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 DataPump Embedded Host Generic Class Driver User's Guide

Engineering Report 950000692
Rev. B
Date: 2011/09/30

PROPRIETARY NOTICE AND DISCLAIMER

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

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

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

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

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

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

Document Release History

Rev. A	2010/08/25	Initial Release
Rev. B	2011/09/30	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Purpose

This document describes the Generic Class Driver (GCD) API provided by the MCCI USB DataPump Embedded USB host stack. The GCD is a general purpose USB driver that provides access to any USB device. It does not implement any USB device class specific functionality such as MSC (Mass Storage Device Class), HID (Human Interface Device) class, CDC (Communications Device Class), or SIC (Still Image Class), and can be used to communicate with any USB device.

1.2 Scope

The embedded GCD provides all necessary services for a client software module to access a USB device. The GCD provides the logic to communicate with USB devices of all types.

The interface to the driver is non-blocking to allow the USB subsystem to continue executing during long running I/O operations. A client prepares a request block containing a pointer to a callback routine that is invoked upon the completion of the asynchronous request. Calls to the Class Driver return immediately, and the callback routine is invoked upon completion of the request.

This document assumes familiarity with the MCCI USB DataPump.

1.3 Glossary

{HOME}	Home directory of the Class Driver. This is usbkern/host/class/generic.
ClassKit	Class Driver Development Kit. This is the software component that provides common routines to Class Drivers.
Class Driver	MCCI USB DataPump Embedded Host Class Driver. This provides low-level access to USB devices.
Class Driver Framework	The source tree that provides a skeletal structure of a Class Driver that is based on the ClassKit. This is generated from the Sample Class Driver.
Device	The hardware component that provides the USB descriptors and data to Class Driver.
FSM	Finite State Machine
GCD	Generic Class Driver

1.4 Referenced Documents

[MOB] MCCI Object Brokerage Specification, MCCI Engineering report 950000961

[USBDI] MCCI USB DataPump Embedded USBDI, MCCI Engineering report 950000325

[CLASSKIT] MCCI USB DataPump Embedded Host Class Driver Development Guide, MCCI Engineering report 950000761

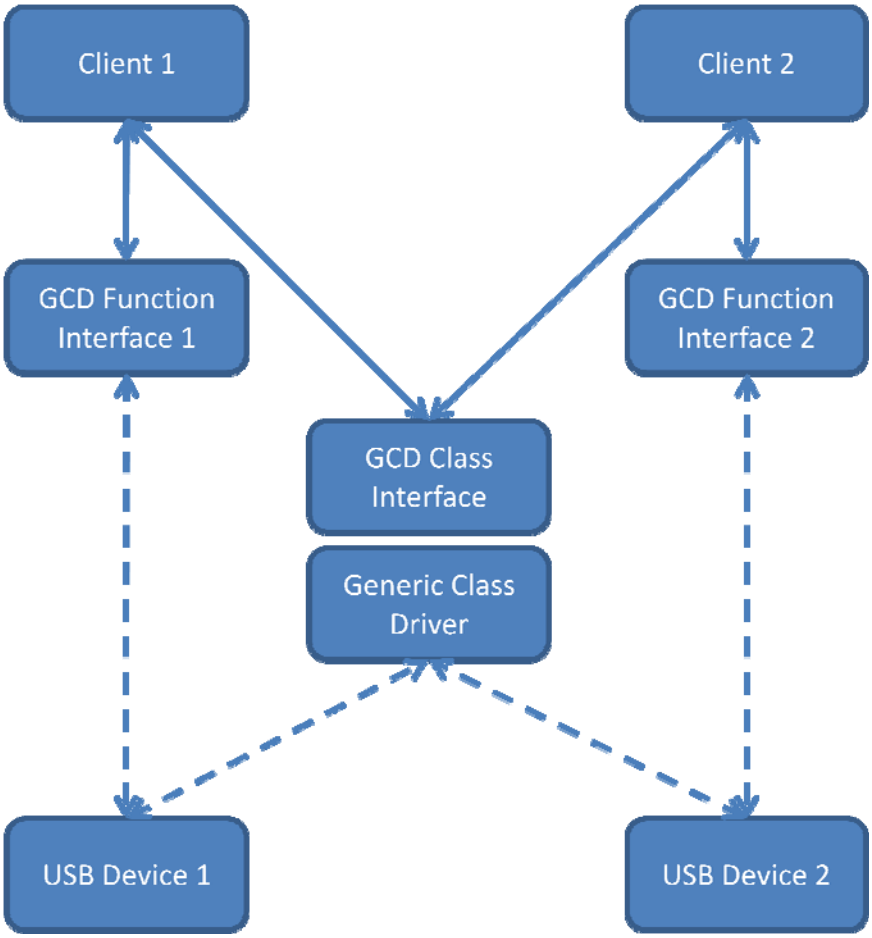
[USB2.0/3.0] Universal Serial Bus Specification, 2.0/3.0 (http://www.usb.org)

2 Client Implementation and Use of GCD

2.1 Introduction

A client of the Generic Class Driver accesses USB devices which are bound to the GCD through the GCD Class Interface and the Function Interface.

Figure 1 Conceptual Diagram



In order to initialize the GCD and use it, the following steps need to be performed:

- Configure the match string list, the GCD class configuration and the GCD private configuration
- Add the configuration and initialization information to ClassDriverInitNode table
- Find the GCD class object and open the GCD class interface
- Enumerate the function vector currently bound to the GCD or wait for the device arrival event
- Open the GCD function interface
- Access and control the USB device through the GCD function interface

Detailed descriptions of these steps are found in following sections.

2.2 Configuration and Initialization

To create a Generic Class Driver (hereafter, GCD) in the MCCI USB DataPump Embedded USB host stack, a client must initialize the GCD before using it. The following code illustrates how to initialize the GCD.

2.2.1 Customize Match List Entries

```
/*
|| Match list entries for generic device class driver
*/
static CONST USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY
sk_vUsbPumpUsbdiGeneric_Matches[] =
          {
          USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
              "vid=040e/f60f;*",       /* catena1610 loopback application */
              USBPUMP_USBDI_PRIORITY_VIDPID
              ),
          USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
              "vid=040e/f602;*",       /* catena1650 loopback application */
              USBPUMP_USBDI_PRIORITY_VIDPID
              ),
          USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
              "*;fc=06/01/01;*",       /* SIC interface */
              USBPUMP_USBDI_PRIORITY_FN_CSP
              )
          };

CONST USBPUMP_USBDI_INIT_MATCH_LIST
sk_UsbPumpUsbdiGeneric_InitMatchList =
          USBPUMP_USBDI_INIT_MATCH_LIST_INIT_V1(
              sk_vUsbPumpUsbdiGeneric_Matches
              );
```

2.2.2 Configuration for GCD Class

In the GCD class configuration, you can set the match list entries and driver class name, function instance name, and the maximum number of instances for this Class Driver.

```
/*
|| Configuration for GCD Class
*/
CONST USBPUMP_USBDI_DRIVER_CLASS_CONFIG gk_UsbPumpUsbdiGeneric_ClassConfig =
          USBPUMP_USBDI_DRIVER_CLASS_CONFIG_INIT_V1(
              &gk_UsbPumpUsbdiGeneric_InitMatchList,
              /* driver class name */ USBPUMP_USBDI_CLASS_GENERIC_NAME,
              /* function instance name */ USBPUMP_USBDI_FUNCTION_GENERIC_NAME,
              /* number of instances */ 2
              );
```

2.2.3 Configuration for GCD Private

In the GCD private configuration, you can set the maximum number of client sessions, configurations, interfaces, alternative settings, pipes and client requests, and the maximum size of the configuration bundle of the USB devices that GCD supports. The client sessions include both class sessions and function sessions. Refer to section 5.1.1.

```
/*
|| Private Configuration for generic device class driver
*/
static
CONST USBPUMP_USBDI_CLASS_GENERIC_CONFIG sk_UsbPumpUsbdGeneric_PrivateConfig =
          USBPUMP_USBDI_CLASS_GENERIC_CONFIG_INIT_V1(
              /* number of client sessions */4,
              /* number of configurations */ 1,
              /* number of interfaces */ 5,
              /* number of alternative settings */ 15,
              /* number of pipes */ 30,
              /* number of requests */ 4,
              /* maximum size of configuration bundle */ 512
              );
```

2.2.4 Add the initialization information for GCD to ClassDriverInitNode Table

In the class driver initialization node, put the GCD initialization function which GDC provides, class and private configurations (Refer to section 2.2.2 and 2.2.3), and debug flags.

```
/*
|| This table provides the initialization information for the class drivers
*/
static
CONST USBPUMP_HOST_DRIVER_CLASS_INIT_NODE sk_ClassDriverInitNodes[] =
```

```
        {
        USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
            /* pProbeFn */ NULL,
            /* pInitFn */ UsbPumpUsbdiClassGeneric_Initialize,
            /* pConfig */ &gk_UsbPumpUsbdiGeneric_ClassConfig,
            /* pPrivateConfig */ &sk_UsbPumpUsbdGD_PrivateConfig,
            /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS
            )
        };
```

2.3 Start Client

In the host initialization completion function, call the client start routine.

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

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

        /*
        || Create sample Generic client object
        */
        UsbPumpSampleGcd_Client_Create(
            pPlatform,
            UDMASK_ERRORS | UDMASK_FLOW
            );

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

2.4 Find Generic Class Driver Object

A client that intends to use the GCD API must open a session to the GCD. To open a session, the client must have the GCD object. The client has to enumerate the GCD object using the Standard API, UsbPumpObject_EnumerateMatchingNames().

```
USBPUMP_CLASS_GCD_CLIENT_DATA *
UsbPumpSampleGcd_Client_Create(
          UPLATFORM*  pPlatform,
          UINT32      DebugFlags
          )
          {
          USBPUMP_OBJECT_ROOT *            pRootObject;
          USBPUMP_OBJECT_HEADER *          pClassObject;
          USBPUMP_CLASS_GCD_CLIENT_DATA *  pClient;
          USTAT                            Status;

          pRootObject = UsbPumpObject_GetRoot(&pPlatform->upf_Header);

          /*
          || Create a sample GCD client object
          */
          pClient = UsbPumpPlatform_Malloc(pPlatform, sizeof(*pClient));
          if (pClient == NULL)
              {
              TTUSB_OBJPRINTF((
                  &pPlatform->upf_Header,
                  UDMASK_ANY | UDMASK_ERRORS,
                  "?UsbPumpSampleGcd_Client_Create:"
                  " Memory (%x bytes) allocation failed\n",
                  sizeof(*pClient)
                  ));
              return NULL;
              }

          pClient->pPlatform = pPlatform;
          UsbPumpObject_Init(
              &pClient->ObjectHeader,
              pPlatform->upf_Header.pClassParent,
              /* Generic Class Driver Sample Client */
              UHIL_MEMTAG('G', 'c', 'd', 'S'),
              sizeof(*pClient),
              "sample.gcd.client.mcci.com",
              &pPlatform->upf_Header,
              NULL
              );
          pClient->ObjectHeader.ulDebugFlags |= DebugFlags;
          pClient->FunctionHandle = NULL;
```

```
UsbPumpTimer_Initialize(
    pPlatform,
    &pClient->Timer,
    UsbPumpSampleGcd_Client_SuspendTimeout
    );

/*
|| Find the Generic Class Driver object
*/
pClassObject = NULL;
pClassObject = UsbPumpObject_EnumerateMatchingNames(
            &pPumpRoot->Header,
            pClassObject,
            USBPUMP_OBJECT_NAME_ENUM_HOST_GCD
            );

/*
|| If the GCD object is found, open a class session
*/
if (pClassObject == NULL)
    {
    TTUSB_OBJPRINTF((
        &pClient->ObjectHeader, UDMASK_ERRORS,
        "?UsbPumpSampleGcd_Client_Create: "
        " Failed to enumerate Generic Class driver\n"
        ));
    UsbPumpObject_DeInit(&pClient->ObjectHeader);
    UsbPumpPlatform_Free(pPlatform, pClient, sizeof(*pClient));
    return NULL;
    }
else
    {
    pClient->pClassObject = pClassObject;
    Status = UsbPumpSampleGcd_Client_OpenSession(pClient, pClassObject);
    }

if (Status != USTAT_OK)
    {
    TTUSB_OBJPRINTF((
        &pClient->ObjectHeader,
        UDMASK_ERRORS,
        "?UsbPumpSampleGcd_Client_Create:"
        " Failed to open a class session(%s)\n",
        UsbPumpStatus_Name(Status)
        ));
    UsbPumpObject_DeInit(&pClient->ObjectHeader);
    UsbPumpPlatform_Free(pPlatform, pClient, sizeof(*pClient));
    return NULL;
    }
```

```
        return pClient;
        }
```

2.5 Open a Class Session

A client has to open a Class Session (a session to the GCD Class) to use the Class Interface of the GCD and post the class events of the driver such as DEVICE_ARRIVAL, DEVICE_DEPARTURE, FUNCTION_OPEN, and FUNCTION_CLOSE. For details of class events, refer to section 6.1.

The following code illustrates how to open a Class Session to the GCD.

```
static
USTAT UsbPumpSampleGcd_Client_OpenSession(
        USBPUMP_CLASS_GCD_CLIENT_DATA *      pGcdClient,
        USBPUMP_OBJECT_HEADER *          pGcdClassObjectHeader
        )
        {
        VOID *  pOpenRequestMemory;

        /*
        || Allocate memory for Open Request (UsbPumpObject_OpenSession)
        */
        pOpenRequestMemory =
            UsbPumpPlatform_Malloc(
                pGcdClient->pPlatform,
                USBPUMP_API_OPEN_REQUEST_MEMORY_SIZE
                );
        if (pOpenRequestMemory == NULL)
            {
            TTUSB_OBJPRINTF((
                pGcdClient->pClassObject,
                UDMASK_ERRORS,
                "?UsbPumpSampleGcd_Client_OpenSession:"
                " Memory (%x bytes) allocation failed\n",
                USBPUMP_API_OPEN_REQUEST_MEMORY_SIZE
                ));
            Return USTAT_NO_MEMORY;
            }

        UsbPumpObject_OpenSession(
            pGcdClassObjectHeader,
            pOpenRequestMemory,
            USBPUMP_API_OPEN_REQUEST_MEMORY_SIZE,
            UsbPumpSampleGcd_Client_OpenSession_Callback,
            pGcdClient,/* pCallBackContext */
            &gk_UsbPumpUsbdiClassGeneric_Guid,
```

```
            NULL,   /* pClientObject -- OPTIONAL */
            &pGcdClient->ClassInCall.GenericCast,
            sizeof(pGcdClient->ClassInCall),
            pGcdClient, /* pClientHandle */
            &sk_UsbPumpUsbdiClassGdi_ClientOutCall.GenericCast,
        sizeof(sk_UsbPumpUsbdiClassGdi_ClientOutCall)/* sizeOutCallApiBuffer */
            );
        return USTAT_OK;
        }

static VOID
UsbPumpSampleGcd_Client_OpenSession_Callback(
        VOID *      pClientContext,
        USBPUMP_SESSION_HANDLE      SessionHandle,
        UINT32      Status,
        VOID *      pOpenRequestMemory,
        RECSIZE     sizeOpenRequestMemory
        )
        {
        USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pClientContext;

        if (Status == USBPUMP_USBDI_GENERIC_STATUS_OK)
            {
            TTUSB_OBJPRINTF((
                pGcdClient->pClassObject,
                UDMASK_ANY,
                "UsbPumpSampleGcd_Client_OpenSession_Callback:"
                " OpenSession STATUS_OK %p\n",
                SessionHandle
                ));
            pGcdClient->SessionHandle = SessionHandle;
            }
        else
            {
            TTUSB_OBJPRINTF((
                pGcdClient->pClassObject,
                UDMASK_ERRORS,
                "UsbPumpSampleGcd_Client_OpenSession_Callback:"
                " OpenSession failed %x\n",
                UsbPumpUsbdiClassGeneric_StatusName(Status),
                Status
                ));
            }

        if (pOpenRequestMemory)
            {
            UsbPumpPlatform_Free(
                pGcdClient->pPlatform,
                pOpenRequestMemory,
```

```
                sizeOpenRequestMemory
                );
            }
        }
```

Refer to [MOB] in the referenced documents section for the details of `UsbPumpObject_OpenSession()`.

The way to handle buffers for In-Calls and Out-Calls is shown below.

1. A client allocates its own buffer to store the Class In-Calls of the GCD. The size of the Class In-Calls is `sizeof(USBPUMP_USBDI_CLASS_GENERIC_INCALL)`.

2. The client calls `UsbPumpObject_OpenSession()` with the pointer of the buffer, the pointer of the Class Out-Calls structure, and the GUID of the interface of the GCD class object.

3. The GCD copies its own Class In-Calls buffer into the buffer the client provides. And it copies the Class Out-Calls buffer the client provides into the buffer of the GCD. The GCD uses the Class Out-Calls for sending class event notifications.

4. The GCD calls the completion routine (i.e., the callback function) to pass the Class In-Calls buffer to the client. The client uses the buffer to call operations of the Class In-Calls.

Upon a completion of opening a Class Session, the callback routine that the client provided is invoked with the client handle, a session handle pointer (class handle), and a status code. If the status code is USBPUMP_USBDI_GENERIC_STATUS_OK, the session is open and the requested Class In-Calls are ready for use.

After opening a session, the GCD sends class notifications to the client through the Notification Class Out-Call that the client passed when it calls UsbPumpObject_OpenSession(). For the details of this Class Out-Call, refer to section 3.2.1 in [CLASSKIT]. The client must implement the Notification Class Out-Call to process the class notifications. The class notifications that the GCD sends are DEVICE_ARRIVAL, DEVICE_DEPARTURE, FUNCTION_OPEN, and FUNCTION_CLOSE.

The DEVICE_ARRIVAL class notification means that a USB device that matches the GCD's match list entries was enumerated and ready to use. If the client wishes to access the USB device through the GCD interface, the client should retrieve the pointer of the function instance to the USB device from the notification information. The client then opens a Function Session (a session to the GCD function instance) to use the Function Interface of the GCD. For the details of opening a Function Session, refer to section 2.7 in this document. An example of the client code for processing the DEVICE_ARRIVAL class notification is provided below:

```
case USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL:
        {
        CONST USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL_INFO * pEventInfo =
```

```
            pNotification;

            TTUSB_OBJPRINTF((
                pGcdClient->pClassObject,
                UDMASK_ANY,
                " UsbPumpSampleGcd_Client_Notification:"
                " A function instance(%p) is arriving.\n",
                pEventInfo->pFunction
                ));

            /*
            || If the client already opened a function session, just ignore
            || this event notification because this sample client cares
            || only one function instance.
            */
            if (pGcdClient->FunctionHandle)
                break;

            /*
            || Retrieve GCD function instance from Notification Information
            || and store it in the client context. This function instance
            || will be used to open a function session to use the function
            || interface of the Generic Class Driver
            */  pGcdClient->pGcdFunction = pEventInfo->pFunction;
        pGcdClient->hDefaultPipe =
            pGcdClient->pGcdFunction->Function.PortInfo.hDefaultPipe;

        /* Call OpenFunction */
        UsbPumpSampleGcd_Client_OpenFunction(pGcdClient, pClientHandle);


        break;
        }
```

The DEVICE_DEPARTURE class notification means that a USB device bound to the GCD has been unplugged. If the client has already opened a Function Session to the USB device that has been unplugged, the client should close the Function Session by invoking Close Function Function In-Call. For the details of this Function In-Call, refer to section 2.8 and 4.1.1.2 in this document. An example of the client code for processing the DEVICE_DEPARTURE class notification is provided below:

```
case USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE:
        /* Call CloseFunction */
            if ((pEventInfo->pFunction == pGcdClient->pFunction) &&
                pGcdClient->FunctionHandle)
                {
                UsbPumpSampleGcd_Client_CloseFunction(pGcdClient);
                pGcdClient->pFunction = NULL;
                pGcdClient->hDefaultPipe = NULL;
```

```
            }
    break;
```

The FUNCTION_OPEN and CLOSE class notifications are sent when a Function Session is opened or closed for the GCD, respectively.

2.6 Enumerate Function Vector Bound to GCD

The client can obtain the number of function instances that are bound to the GCD by invoking GetNumDevs Class In-Call and enumerating a vector of them using GetBoundDevs Class In-Call. The client can open Function Sessions by using the function instances returned. For the details of this Class In-Call, refer to section 4.1.1 in this document.

2.7 Open a Function Session

The client needs to open a Function Session to a specific function instance that is bound to a USB device. After getting the Function Session, the client can access the USB device by invoking the Function In-Calls. For the details of this Class In-Call, refer to section 4.1.1 in this document.

2.8 Access and Control USB Device Using Function In-Calls

Upon completion of OpenFunction operation, the callback routine that the client passed is invoked. If the status code is successful, the client has a function handle and a Function In-Call buffer. The client can use function operations such as sending GetDescription(Device) request and reading data from a Bulk endpoint using function pointers of the Function In-Call buffer and the function handle. For the details of GCD Function In-Calls, refer to section 4.2.1 in this document.

3 Generic Class Driver Memory Requirement

Below is the equation to calculate the memory requirements. The equation returns the number of bytes required. The equation is specific to the Catena platform and the Microsoft Visual C 6.0 compiler, but is typical of memory use on 32-bit platforms.

(Approximately) RequiredMemory $\doteqdot$

472 + /* GCD Class overhead */

NumInstances * (

920 + /* GCD Function overhead */

(MaxSession * 16) +

(NumConfig * 32) +

(NumIfc * 32) +

(NumAlt * 32) +

(NumPipe * 80) +

(NumRequest * 224) +

MaxConfigDescSize

)

The table below shows the memory requirements for some possible configurations of the Generic Class Driver. The first configuration could be used to support one PictBridge device containing one Interrupt IN endpoint and one pair of Bulk IN/OUT endpoints, and one Client which uses the GCD class and function interfaces. The second one provides more general Generic Class Device support.

In this release, it is not possible to configure the Composite driver to request and release memory dynamically – all required memory is allocated during initialization.

Table 1 Generic Class Driver Memory Requirements

Configuration	Variable	Comment	Required Memory
Single PictBridge Device	NumInstances = 1	Only want to support one instance	2592 bytes
	MaxSession = 2	Support one class session and one function session for one client	
	NumConfig = 1	One configuration	
	NumIfc = 1	One interface	
	NumAlt = 1	One alternative setting	
	NumPipe = 3	Three pipes (Bulk IN/OUT + Interrupt IN)	
	NumRequest = 3	Support three requests at the same time	
	Max ConfigDescSize = 128	Maximum configuration bundle size	
General-purpose	NumInstances = 2	Only want to support one instance	16896 bytes

Configuration	Variable	Comment	Required Memory
	MaxSession = 4	Support one class session and one function session for one client	
	NumConfig = 1	One configuration	
	NumIfc = 5	One interface	
	NumAlt = 10	One alternative setting	
	NumPipe = 20	Three pipes (Bulk IN/OUT + Interrupt IN)	
	NumRequest = 20	Support three requests at the same time	
	Max ConfigDescSize = 512	Maximum configuration bundle size	

4 GCD Interfaces

The GCD provides two types of interfaces, the GCD Class interface and the Function Interface. The GCD object has one GCD class interface. The GCD object can have multiple Function Interfaces one for each GCD function instance. To use the GCD class interface, a client must open a class session to the GCD object using UsbPumpObject_OpenSession(). For the details of UsbPumpObject_OpenSession(), refer to [MOB] in the referenced documents section. And to use the GCD function interface, the client must open a function session to a GCD function instance using OpenFunction() Class In-Call. Refer to [ClassKit].

4.1 Class Interface

4.1.1 Class In-Calls

In the OpenSession routine and its completion routine, the GCD provides the Class In-Calls to the client and the client provides the Class Out-Calls to the driver. The client uses the Class In-Calls to retrieve the number of the function instances and the function instance list, learn features of USBD/GCD, and open a Function Session to a specific function instance.

4.1.1.1 CloseSession Operation

This operation closes the session that the client opened using UsbPumpObject_OpenSession(). For the details of this operation, refer to section 3.1.1 in [CLASSKIT] and [MOB].

4.1.1.2 OpenFunction Operation

This Class In-Call opens a Function Session to a specific function instance to use Function In-Calls. In this operation, Function In-Calls and Out-Calls are exchanged between the client and the function instance. A function handle, necessary to use Function In-Calls, is provided in the OpenFunction's completion routine (callback). Refer to section 4.2 below in this document for the detailed description of the Function Interface.

For the details of the OpenFunction operation, refer to section 3.1.2 in [CLASSKIT].

4.1.1.3 GetNumDevices Operation

This Class In-Call returns the number of function instances that are bound to the GCD. For the details of this operation, refer to section 3.1.3 in [CLASSKIT].

4.1.1.4 GetBoundDevices Operation

This Class In-Call returns the vector of the function instances that will be returned to the callback routine. For the details of this operation, refer to section 3.1.4 in [CLASSKIT].

4.1.1.5 GetUsbdFeature Operation

Description

Note: This operation has yet to be implemented. Currently, default return for this operation call is TRUE.

The client uses this operation to learn the features of the USBD where the GCD resides. Refer to [USBDI] for details of USBD. For example, if the client wishes to know whether the USBD supports isochronous transfers, the client simply calls this operation with a feature name string such as "ISOCH_SUPPORT". If the USBDI supports this feature, TRUE is returned via the callback routine of this operation. Otherwise, FALSE is returned.

This API will return one of the following return codes through its callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`

Declaration of Prototype

```
typedef VOID
USBPUMP_USBDI_GENERIC_GET_USBD_FEATURE_FN(
         USBPUMP_SESSION_HANDLE           SessionHandle,
         USBPUMP_USBDI_GENERIC_GET_USBD_FEATURE_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         CONST TEXT *             pFeatureName,
         SIZE_T                   sizeFeatureName
```

```
        );
```

Parameters

- `SessionHandle` is the class session handle returned from `UsbPumpObject_OpenSession()`.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of the GetUsbdFeature operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pFeatureName` is the feature name string such as "ISOCH_SUPPORT".
- `sizeFeatureName` is the length of the feature name string.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_GET_USBD_FEATURE_CB_FN,
__TMS_VOID,
        (
        __TMS_VOID *                /* pClientContext */,
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */,
        __TMS_BOOL                  /* fFeaturePresent */
        ));
```

Parameters

- `pClientContext` is the pointer to the client information that is provided by the Client.
- `ErrorCode` is the result code of the operation.
- `fFeaturePresent` is a Boolean value that indicates whether the USBD supports the feature.

4.1.1.6 GetGenDrvFeature Operation

Description

Note: This operation has yet to be implemented. Currently, default return for this operation call is TRUE.

The client uses this operation to learn the features of the Generic Class Driver. For example, if the client wishes to know whether the GCD supports isochronous transfers, the client simply calls this operation with a feature name string such as "ISOCH_SUPPORT". If the GCD supports this feature, TRUE is returned via the callback routine of this operation. Otherwise, FALSE is returned.

This API will return one of the following return codes through its callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`

Declaration of Prototype

```
typedef VOID
USBPUMP_USBDI_GENERIC_GET_GENERIC_DRIVER_FEATURE_FN,(
        USBPUMP_SESSION_HANDLE          SessionHandle,
        USBPUMP_USBDI_GENERIC_GET_GENERIC_DRIVER_FEATURE_CB_FN * pCallBack,
        VOID *                  pCallBackContext,
        CONST TEXT *            pFeatureName,
        SIZE_T                  sizeFeatureName
        );
```

Parameters

- `SessionHandle` is the class session handle returned from `UsbPumpObject_OpenSession()`.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of the GetUsbdFeature operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pFeatureName` is the feature name string such as "ISOCH_SUPPORT".
- `sizeFeatureName` is the length of the feature name string.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_GET_GENERIC_DRIVER_FEATURE_CB_FN,
__TMS_VOID,
        (
        __TMS_VOID *                /* pClientContext */,
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */,
        __TMS_BOOL                  /* fFeaturePresent */
        ));
```

Parameters

- `pClientContext` is the pointer to the client information that is provided by the Client.
- `ErrorCode` is the result code of the operation.
- `fFeaturePresent` is a Boolean value that indicates whether the Generic Class Driver supports the feature.

4.1.2 Class Out-Calls

4.1.2.1 Notification Operation

The client has to implement this operation and provide the pointer to the operation when calling UsbPumpObject_OpenSession(). If a class notification is available in the GCD, this operation is invoked to send the notification to the client. For the details of this Out-Call, refer to section 3.2.1 in [CLASSKIT].

4.2 Function Interface

4.2.1 Function In-Calls

In the OpenFunction Class In-Call and its completion routine, the GCD function instance provides the Function In-Calls to the client and the client provides the Function Out-Calls to the driver function instance. The client uses the Function In-Calls to access and control the USB device that is bound to the GCD function instance.

4.2.1.1 CloseFunction Operation

The client calls this operation to close the Function Session that is opened via OpenFunction Class In-Call if the client doesn't want to use the session any longer. For the details of this operation, refer to section 3.3.1 in [CLASSKIT].

4.2.1.2 CancelRequest Operation

Description

The client uses this operation to cancel a pending request. This operation doesn't have any callback routine. For the details of the request, refer to the sections further below.

This API will return one of the following return codes:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_ALREADY_COMPLETED`

Declaration of Prototype

```
typedef USBPUMP_USBDI_GENERIC_STATUS
USBPUMP_USBDI_GENERIC_CANCEL_REQUEST_FN(
          USBPUMP_SESSION_HANDLE   FunctionHandle,
          VOID *                   pRequestHandle
          );
```

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.

- `pRequestHandle` is the request handle that the client wishes to cancel. The request handle will be returned when the client invokes a Function In-Call that returns a request handle such as Read Bulk/Interrupt Pipe Function In-Call.

Callback Routine

None.

4.2.1.3 GetDeviceState Operation

Description

The client uses this operation to check the state of the device that is bound to the GCD function instance. This operation will return the state of the device via the callback routine that the client provided. The information about the device state is as follows:

```
struct __TMS_STRUCTNAME(USBPUMP_USBDI_GENERIC_DEVICE_STATE)
        {
        __TMS_BOOL  fRemoved;
        __TMS_BOOL  fStopped;
        __TMS_BOOL  fFunctionOpened;
        };
```

- `fRemoved` indicates if the device is already removed or not.

- `fStopped` indicates if the device is already stopped or not.

- `fFunctionOpened` indicates if the function instance to the device is already opened or not.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`

Declaration of Prototype

```
typedef VOID *                  // pRequestHandle
USBPUMP_USBDI_GENERIC_GET_DEVICE_STATE_FN(
        USBPUMP_SESSION_HANDLE          FunctionHandle,
        USBPUMP_USBDI_GENERIC_GET_DEVICE_STATE_CB_FN * pCallBack,
        VOID *                  pCallBackContext,
        USBPUMP_USBDI_GENERIC_DEVICE_STATE * pDeviceState
        );
```

Return Value

- `pRequestHandle` is always NULL because this request cannot be cancelled.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pDeviceState` is a buffer that the client provides to make this operation fill the device state information.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_GET_DEVICE_STATE_CB_FN,
__TMS_VOID,
        (
        __TMS_VOID *                /* pClientContext */,
        __TMS_VOID *                /* pRequestHandle */,
        __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */,
        __TMS_USBPUMP_USBDI_GENERIC_DEVICE_STATE *   /* pDeviceState */
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` that the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pDeviceState` is the buffer that contains the device state information.

4.2.1.4 GetDeviceDescriptor Operation

Description

The client uses this operation to get the USB device descriptor of the target device. This operation will return the result code of the operation and the USB device descriptor, if the operation is successful, via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- USBPUMP_USBDI_GENERIC_STATUS_OK
- USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE
- USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED
- USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER
- USBPUMP_USBDI_GENERIC_STATUS_BUSY
- Error codes from USBD layer (Refer to section 7.2)

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_GET_DEVICE_DESCRIPTOR_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_GET_DEVICE_DESCRIPTOR_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         VOID *                   pBuffer,
         BYTES                    sizeBuffer
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pBuffer` is a buffer which will store the USB device descriptor.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to.

Callback Routine

```
__TMS_FNTYPE_DEF(                                 \
USBPUMP_USBDI_GENERIC_GET_DEVICE_DESCRIPTOR_CB_FN,             \
__TMS_VOID,                                       \
         (                                        \
         __TMS_VOID *                /* pClientContext */,    \
         __TMS_VOID *                /* pRequestHandle */,    \
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */, \
         __TMS_VOID *                /* pBuffer */,       \
         __TMS_BYTES                 /* sizeBuffer */,    \
         __TMS_BYTES                 /* nBytesRead */ \
```

```
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pBuffer` is a buffer which contains the USB descriptor that was transferred from the target device. If the size of the buffer that the client provided is less than the size of the USB descriptor of the target device, the buffer will contain a part of the descriptor. If the client wants to get the complete descriptor, it must call this operation again with a buffer which size is equal to *bLength* of the descriptor.
- `sizeBuffer` is the size of the buffer that the client provided.
- `nBytesRead` is the size of the data that the target device actually transferred.

Example

```
pGcdClient->FunctionInCall.GenDrv.pGetDeviceDescriptorFn(
          pGcdClient->FunctionHandle,
          UsbPumpSampleGcd_Client_GetDevDescCbFn,
          pGcdClient,
          pBuffer,
          sizeBuffer
          );

static VOID
UsbPumpSampleGcd_Client_GetDevDescCbFn(
          VOID *       pCallbackCtx,
          VOID *       pRequestHandle,
          USBPUMP_USBDI_GENERIC_STATUS     ErrorCode,
          VOID *       pBuffer,
          BYTES        sizeBuffer,
          BYTES        nBytesRead
          )
          {
          USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pCallbackCtx;
          USBPUMP_UNREFERENCED_PARAMETER(pRequestHandle);
          USBPUMP_UNREFERENCED_PARAMETER(sizeBuffer);
          USBPUMP_UNREFERENCED_PARAMETER(pBuffer);
          USBPUMP_UNREFERENCED_PARAMETER(nBytesRead);

          if (ErrorCode != USBPUMP_USBDI_GENERIC_STATUS_OK)
              {
```

```
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ERRORS,
            "?UsbPumpSampleGcd_Client_GetDevDescCbFn:"
            " ErrorCode:%s(%d)\n",
            UsbPumpUsbdiClassGeneric_StatusName(ErrorCode),
            ErrorCode
            ));
        }
    else
        {
        BYTES i;
        USBIF_DEVDESC * CONST pDevDesc = pBuffer;

        pGcdClient->iManufacturer = pDevDesc->iManufacturer;
        pGcdClient->iProduct = pDevDesc->iProduct;
        pGcdClient->iSerialNumber = pDevDesc->iSerialNumber;
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ANY,
            " UsbPumpSampleGcd_Client_GetDevDescCbFn:"
            " pBuffer:%p, sizeBuffer:%d, nBytesRead:%d\n",
    pBuffer, sizeBuffer, nBytesRead
            ));
        for (i=0; i<nBytesRead; i++)
            {
            TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                UDMASK_ANY,
                "%02x %s",
                ((UINT8 *) pBuffer)[i],
                (((i + 1) % 8)?"":"\n")
                ));
            }
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ANY,
            "\n"
            ));
        UsbPumpSampleGcd_Client_GetConfigDesc(
            pGcdClient,
            pGcdClient->Buffer,
            sizeof(pGcdClient->Buffer),
            0
            );
        }
    }
```

4.2.1.5 GetConfigDescriptor Operation

Description

The client uses this operation to get the USB configuration descriptor of the target device. This operation will return the result code of the operation and the USB configuration descriptor, if the operation was successful via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_GET_CONFIG_DESCRIPTOR_FN(
          USBPUMP_SESSION_HANDLE           FunctionHandle,
          USBPUMP_USBDI_GENERIC_GET_CONFIG_DESCRIPTOR_CB_FN * pCallBack,
          VOID *                  pCallBackContext,
          VOID *                  pBuffer,
          BYTES                   sizeBuffer,
          UINT                    iConfig
          );
```

Return Values

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pBuffer` is a buffer which will store the USB configuration descriptor.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to.
- `iConfig` is a configuration number.

Callback Routine

```
__TMS_FNTYPE_DEF(                              \
```

```
USBPUMP_USBDI_GENERIC_GET_CONFIG_DESCRIPTOR_CB_FN,              \
__TMS_VOID,                                     \
        (                                       \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_VOID *                /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesRead */ \
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pBuffer` is a buffer which contains the USB descriptor that was transferred from the target device. If the size of the buffer that the client provided is less than the size of the USB descriptor of the target device, the buffer will contain a part of the descriptor. If the client wants to get the complete descriptor, it must call this operation again with a buffer with a size equal to *bLength* of the descriptor.
- `sizeBuffer` is the size of the buffer that the client provided.
- `nBytesRead` is the size of the data that the target device actually transferred.

Example

This operation is very similar to the GetDeviceDescriptor operation above except `iConfig.` Refer to the example in section 4.2.1.4 in this document.

4.2.1.6 GetConfigTree Operation

Description

The client uses this operation to retrieve the configuration tree of the target device. The configuration tree is a tree that describes the possible operating configurations of the device. This operation will return the result code of the operation and the configuration tree, the pointer of USBPUMP_USBDI_CFG_NODE structure, if the operation was successful via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- USBPUMP_USBDI_GENERIC_STATUS_OK
- USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE
- USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED
- USBPUMP_USBDI_GENERIC_STATUS_BUFFER_TOO_SMALL
- USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_GET_CONFIG_TREE_FN(
         USBPUMP_SESSION_HANDLE          FunctionHandle,
         USBPUMP_USBDI_GENERIC_GET_CONFIG_TREE_CB_FN * pCallBack,
         VOID *                  pCallBackContext,
         VOID *                  pBuffer,
         BYTES                   sizeBuffer
         );
```

Return Value

- `pRequestHandle` is always NULL because this request cannot be cancelled.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `pBuffer` is a buffer which will store the configuration tree.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to.

Callback Routine

```
__TMS_FNTYPE_DEF(                              \
USBPUMP_USBDI_GENERIC_GET_CONFIG_TREE_CB_FN,                  \
__TMS_VOID,                                    \
         (                                     \
         __TMS_VOID *                /* pClientContext */,    \
         __TMS_VOID *                /* pRequestHandle */,    \
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */, \
         __TMS_VOID *                /* pBuffer */,       \
         __TMS_BYTES                 /* sizeBuffer */,    \
         __TMS_BYTES                 /* nBytesRead */ \
         ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer which contains the configuration tree to the target device. If the size of the buffer that the client provided is less than the size of the configuration tree, the buffer will NULL and `nBytesRead` will let the client know the size needed. If the client wants to get the configuration tree, it must call this operation again with a buffer which size is equal to `nBytesRead`.

- `sizeBuffer` is the size of the buffer that the client provided.

- `nBytesRead` is the size of the data that the target device actually transferred. If the size of the buffer is too small to store the configuration tree, this parameter indicates the buffer size needed to store the tree. In this case, the result code is USBPUMP_USBDI_GENERIC_STATUS_BUFFER_TOO_SMALL.

Example

```
pGcdClient->FunctionInCall.GenDrv.pGetConfigTreeFn(
          pGcdClient->FunctionHandle,
          UsbPumpSampleGcd_Client_GetConfigTreeCbFn,
          pGcdClient,
          pBuffer,
          sizeBuffer
          );

static VOID
UsbPumpSampleGcd_Client_GetConfigTreeCbFn(
          VOID *       pCallbackCtx,
          VOID *       pRequestHandle,
          USBPUMP_USBDI_GENERIC_STATUS     ErrorCode,
          VOID *       pBuffer,
          BYTES        sizeBuffer,
          BYTES        nBytesRead
          )
          {
          USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pCallbackCtx;

          USBPUMP_UNREFERENCED_PARAMETER(pRequestHandle);
          USBPUMP_UNREFERENCED_PARAMETER(sizeBuffer);


          if (ErrorCode != USBPUMP_USBDI_GENERIC_STATUS_OK)
              {
              TTUSB_OBJPRINTF((pGcdClient->pClassObject,
```

```
            UDMASK_ERRORS,

            "?UsbPumpSampleGcd_Client_GetConfigTreeCbFn:"
            " ErrorCode:%s(%d)\n",
        UsbPumpUsbdiClassGeneric_StatusName(ErrorCode),
            ErrorCode
            ));
        if (ErrorCode == USBPUMP_USBDI_GENERIC_STATUS_BUFFER_TOO_SMALL)
            {
            pGcdClient->pConfigRoot =
                UsbPumpPlatform_Malloc(
                    pGcdClient->pPlatform,
                    nBytesRead
                    );

            UsbPumpSampleGcd_Client_GetConfigTree(                  pGcdClient,
                pGcdClient->pConfigRoot,
                nBytesRead
                );
            }
        }
    else
        {
        USBPUMP_USBDI_CFG_NODE *     pConfigRoot = pBuffer;
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ANY,
            " UsbPumpSampleGcd_Client_GetConfigTreeCbFn:"
            " pConfigRoot:%p, sizeBuffer:%d, nBytesRead:%d\n",
        pConfigRoot, sizeBuffer, nBytesRead
            ));
        UsbPumpSampleGcd_Client_ParseConfigTree(
            pGcdClient,
            pConfigRoot
            );

        UsbPumpSampleGcd_Client_WriteCtrl(
            pGcdClient,
            pGcdClient->hDefaultPipe,
            /* bmReqType */ USB_bmRequestType_HSIFC,
            /* bRequest */ USB_bRequest_SET_INTERFACE,
            /* wValue */ TEST_bAlternateSetting,
            /* wIndex */ TEST_bInterfaceNumber,
            NULL,
            0,
            /* Timeout */ 500 /* ms */
            );
        }
    }
```

```
static VOID
UsbPumpSampleGcd_Client_ParseConfigTree(
          USBPUMP_CLASS_GCD_CLIENT_DATA * pGcdClient,
          USBPUMP_USBDI_CFG_NODE *     pConfigRoot
          )
          {
          UINT32      iPipe = 0;
          UINT32      hPipeIndex = 0;
          do
              {
              USBPUMP_USBDI_IFC_NODE * pIfcNode =
                  USBPUMP_USBDI_CFG_NODE_IFC(
                      pConfigRoot
                      );
              TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                  UDMASK_ANY,
                  " UsbPumpSampleGcd_Client_ParseConfigTree:"
                  " pConfigRoot:%p, bConfigurationValue:%x\n",
                  pConfigRoot, pConfigRoot->bConfigurationValue
                  ));
              do
                  {
                  USBPUMP_USBDI_ALTSET_NODE * pAltNode =
                      USBPUMP_USBDI_IFC_NODE_ALTSET(pIfcNode);
                  TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                      UDMASK_ANY,
                      "  UsbPumpSampleGcd_Client_ParseConfigTree:"
                      " pIfcNode:%p, bInterfaceNumber:%x,"
                      " bNumAltSettings:%x\n",
                      pIfcNode, pIfcNode->bInterfaceNumber,
                      pIfcNode->bNumAltSettings
                      ));
                  do
                      {
                      USBPUMP_USBDI_PIPE_NODE * pPipeNode =
                          USBPUMP_USBDI_ALTSET_NODE_PIPE(pAltNode);
                      TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                          UDMASK_ANY,
                          "   UsbPumpSampleGcd_Client_ParseConfigTree:"
                          " pAltNode:%p, bAlternateSetting:%x\n",
                          pAltNode,
                          pAltNode->bAlternateSetting
                          ));
                      for (; iPipe < pAltNode->bNumPipes; ++pPipeNode, ++iPipe)
                          {
                          pGcdClient->hPipes[hPipeIndex++] = pPipeNode->hPipe;
                          TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                              UDMASK_ANY,
                              "    UsbPumpSampleGcd_Client_ParseConfigTree:"
```

```
                                " pPipeNode:%p, hPipe:%p, dwMaxTransferSize:%x"
                                " wMaxStreams:%u\n",
                                pPipeNode,
                                pPipeNode->hPipe,
                                pPipeNode->dwMaxTransferSize,
                                pPipeNode->wMaxStreamID
                                ));

                            if (pPipeNode->wMaxStreamID != 0)
                                {
                                pGcdClient->MaxStreamID =
                                    pPipeNode->wMaxStreamID;

                                /* Save Out/In pipe index */
                                if (pPipeNode->bEndpointAddress & 0x80)
                                    pGcdClient->InStreamPipe = pPipeNode->hPipe;
                                else
                                    pGcdClient->OutStreamPipe = pPipeNode->hPipe;
                                }
                            else
                                {
                                pGcdClient->MaxStreamID = 0;
                                }
                            }
                        } while ((pAltNode = USBPUMP_USBDI_ALTSET_NODE_NEXT(pAltNode))
!= NULL);
                    } while ((pIfcNode = USBPUMP_USBDI_IFC_NODE_NEXT(pIfcNode)) !=
NULL);
                } while ((pConfigRoot = USBPUMP_USBDI_CFG_NODE_NEXT(pConfigRoot)) !=
NULL);
            pGcdClient->nPipes = hPipeIndex;
            }
```

4.2.1.7 ReadControlPipe Operation

Description

The client uses this operation to send an arbitrary USB command to the target device and optionally receive data returned from the device. If the buffer that the client provided is not NULL the CONTROL IN transfer consists of three stages: SETUP, DATA(IN), and STATUS(OUT). Otherwise (i.e. buffer==NULL), it will consist of two stages: SETUP and STATUS(IN). This operation will return the result code of the operation and the data transferred from the device, if the operation was successful via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`

- USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE
- USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED
- USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER
- USBPUMP_USBDI_GENERIC_STATUS_BUSY
- Error codes from USBD layer (Refer to section 7.2)

Declaration of Prototype

```
typedef VOID *                      // pRequestHandle
USBPUMP_USBDI_GENERIC_READ_CONTROL_PIPE_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_READ_CONTROL_PIPE_CB_FN *  pCallBack,
         VOID *                   pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE        hPipe,
         UINT8                    bmRequestType,
         UINT8                    bRequest,
         UINT16                   wValue,
         UINT16                   wIndex,
         VOID *                   pBuffer,
         UINT16                   sizeBuffer,
         USBPUMP_USBDI_GENERIC_TIMEOUT        milliseconds
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.

- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.

- `pCallBackContext` is provided by the Client to be used for the callback routine.

- `hPipe` is the PIPE handle that the client sends a USB device request to. The PIPE usually is the target device's Default Control Pipe.

- `bmRequestType` is the type of the USB device request. Refer to section 9.3 in [USB2.0/3.0].

- `bRequest` specifies the particular request. Refer to section 9.3 in [USB2.0/3.0].

- `wValue` vary according to the request. It is used to pass a parameter to the device, specific to the request. Refer to section 9.3 in [USB2.0/3.0].

- `wIndex` vary according to the request. It is used to pass a parameter to the device, specific to the request. Refer to section 9.3 in [USB2.0/3.0].

- `pBuffer` is a buffer which will store the data from the target device. If this is NULL, this operation performs two stages CONTROL-IN transfer.

- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This parameter determines the `wLength` of the USB device request.

- `milliseconds` is the request timeout in ms.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_READ_CONTROL_PIPE_CB_FN,                  \
__TMS_VOID,                                         \
        (                                           \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_VOID *                /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesRead */     \
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer which contains the data returned from the target device in the data stage.

- `sizeBuffer` is the size of the buffer that the client provided.

- `nBytesRead` is the size of the data that the target device actually transferred.

4.2.1.8 WriteControlPipe Operation

Description

The client uses this operation to send an arbitrary USB command with optional additional data to the target device. If the buffer that the client provided is not NULL the CONTROL OUT transfer consists of three stages: SETUP, DATA(OUT), and STATUS(IN). Otherwise (i.e.

buffer==NULL), it will consist of two stages: SETUP and STATUS(IN). This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_WRITE_CONTROL_PIPE_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_WRITE_CONTROL_PIPE_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE        hPipe,
         UINT8                    bmRequestType,
         UINT8                    bRequest,
         UINT16                   wValue,
         UINT16                   wIndex,
         CONST VOID *             pBuffer,
         BYTES                    sizeBuffer,
         USBPUMP_USBDI_GENERIC_TIMEOUT        milliseconds
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the PIPE handle that the client sends a USB device request to. The PIPE usually is the target device's Default Control Pipe.

- `bmRequestType` is the type of the USB device request. Refer to section 9.3 in [USB2.0/3.0].

- `bRequest` specifies the particular request. Refer to section 9.3 in [USB2.0/3.0].

- `wValue` varies according to the request. It is used to pass a parameter to the device, specific to the request. Refer to section 9.3 in [USB2.0/3.0].

- `wIndex` varies according to the request. It is used to pass a parameter to the device, specific to the request. Refer to section 9.3 in [USB2.0/3.0].

- `pBuffer` is a buffer which contains data that the client wishes to transfer to the target device with the USB device request. If this is NULL, this operation performs two stages CONTROL-OUT transfer.

- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This parameter determines the `wLength` of the USB device request.

- `milliseconds` is the request timeout in ms.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_WRITE_CONTROL_PIPE_CB_FN,                     \
__TMS_VOID,                                         \
        (                                           \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_CONST __TMS_VOID *        /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesWritten */  \
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer that the client provided.

- `sizeBuffer` is the size of the buffer that the client provided.

- nBytesWritten is the size of the data which is transferred to the target device actually.

Example

This example illustrates how to send a SetInterface request to the target device to select an alternate setting for the specified interface. In this example, the client tries to select the alternate setting 2 of the interface 1. For the details of the SetInterface request, refer to section 9.4.10 in [USB2.0/3.0].

```
UsbPumpSampleGcd_Client_WriteCtrl(
          pGcdClient,
          pGcdClient->hDefaultPipe,
          /* bmReqType */ USB_bmRequestType_HSIFC,
          /* bRequest */ USB_bRequest_SET_INTERFACE,
          /* wValue */ 2  /* Alternate Setting */,
          /* wIndex */ 1  /* Interface */,
          NULL        /* pGcdClient->pBuffer */,
          0           /* sizeof(pGcdClient->pBuffer) */,
          /* Timeout */ 500 /* ms */
          );

static VOID
UsbPumpSampleGcd_Client_WriteCtrl(
          USBPUMP_CLASS_GCD_CLIENT_DATA * pGcdClient,
          USBPUMP_USBDI_PIPE_HANDLE    hPipe,
          UINT8               bmRequestType,
          UINT8               bRequest,
          UINT16              wValue,
          UINT16              wIndex,
          VOID *              pBuffer,
          BYTES               sizeBuffer,
          USBPUMP_USBDI_GENERIC_TIMEOUT    milliseconds
          )
          {
          VOID *  pRequestHandle;

          pRequestHandle =
              pGcdClient->FunctionInCall.GenDrv.pWriteControlPipeFn(
                  pGcdClient->FunctionHandle,
                  UsbPumpSampleGcd_Client_WriteCtrlCbFn,
                  pGcdClient,
                  hPipe,
                  bmRequestType,
                  bRequest,
                  wValue,
                  wIndex,
                  pBuffer,
                  sizeBuffer,
                  milliseconds
              );
```

```
        if (!pRequestHandle)
            {
            TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                UDMASK_ERRORS,
                "?UsbPumpSampleGcd_Client_WriteCtrl:"
                " ERROR - pRequestHandle is NULL!\n"
                ));
            return;
            }

        /*
        || This Function In-Call returns the pointer of the request handle
        || which is used to cancel the request.
        */
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ANY,
            " UsbPumpSampleGcd_Client_WriteCtrl:"
            " pRequestHandle=%p\n",
            pRequestHandle
            ));
        }

static VOID
UsbPumpSampleGcd_Client_WriteCtrlCbFn(
        VOID *      pCallbackCtx,
        VOID *      pRequestHandle,
        USBPUMP_USBDI_GENERIC_STATUS     ErrorCode,
        VOID *      pBuffer,
        BYTES       sizeBuffer,
        BYTES       nBytesWritten
        )
        {
        USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pCallbackCtx;
        USBPUMP_UNREFERENCED_PARAMETER(pRequestHandle);
        USBPUMP_UNREFERENCED_PARAMETER(pBuffer);
        USBPUMP_UNREFERENCED_PARAMETER(sizeBuffer);
        USBPUMP_UNREFERENCED_PARAMETER(nBytesWritten);

        if (ErrorCode != USBPUMP_USBDI_GENERIC_STATUS_OK)
            {
            TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                UDMASK_ERRORS,
                "?UsbPumpSampleGcd_Client_WriteCtrlCbFn:"
                " ErrorCode:%s(%d)\n",
                UsbPumpUsbdiClassGeneric_StatusName(ErrorCode),
                ErrorCode
                ));
            }
        else
```

```
        {
        TTUSB_OBJPRINTF((pGcdClient->pClassObject,
            UDMASK_ANY,
            " UsbPumpSampleGcd_Client_WriteCtrlCbFn:"
            " pBuffer:%p, sizeBuffer:%d, nBytesWritten:%d\n",
            pBuffer, sizeBuffer, nBytesWritten
            ));
        UsbPumpSampleGcd_Client_GenerateTestData(
            pGcdClient->Buffer,
            sizeof(pGcdClient->Buffer)
            );

        UsbPumpSampleGcd_Client_StartLoopbackTest(
            pGcdClient,
            LOOPBACK_TEST_ITERATIONS
            );
        }
    }
```

4.2.1.9 ReadBulkIntPipe Operation

Description

The client uses this operation to receive data from a BULK or INTERRUPT pipe of the target device. This operation performs a BULK IN or INTERRUPT IN transfer depending on the type of the pipe that the pipe handle refers to. This operation will return the result code of the operation and data transferred from the pipe via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                  // pRequestHandle
USBPUMP_USBDI_GENERIC_READ_BULKINT_PIPE_FN(
        USBPUMP_SESSION_HANDLE          FunctionHandle,
        USBPUMP_USBDI_GENERIC_READ_BULKINT_PIPE_CB_FN *  pCallBack,
        VOID *                  pCallBackContext,
        USBPUMP_USBDI_PIPE_HANDLE       hPipe,
        VOID *                  pBuffer,
        BYTES                   sizeBuffer,
        USBPUMP_USBDI_GENERIC_TIMEOUT       milliseconds
```

```
        );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client wish to receive data from. The type of the pipe must be either BULK or INTERRUPT.
- `pBuffer` is a buffer that will store data transferred from the target pipe. This cannot be NULL.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate very large transfers.
- `milliseconds` is the request timeout in ms.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_READ_BULKINT_PIPE_CB_FN,                  \
__TMS_VOID,                                         \
        (                                           \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_VOID *                /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesRead */     \
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pBuffer` is a buffer that the client provided.
- `sizeBuffer` is the size of the buffer that the client provided.
- `nBytesRead` is the size of the data which is transferred from the target device actually.

Example

Refer to section 4.2.1.10 Write Bulk/Interrupt Pipe for the example of this operation.

4.2.1.10 WriteBulkIntPipe Operation

Description

The client uses this operation to send data to a BULK or INTERRUPT pipe of the target device. This operation performs a BULK OUT or INTERRUPT OUT transfer depending on the type of the pipe that the pipe handle refers to. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_WRITE_BULKINT_PIPE_FN(
         USBPUMP_SESSION_HANDLE          FunctionHandle,
         USBPUMP_USBDI_GENERIC_WRITE_BULKINT_PIPE_CB_FN * pCallBack,
         VOID *                  pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE       hPipe,
         CONST VOID *            pBuffer,
         BYTES                   sizeBuffer,
         USBPUMP_USBDI_GENERIC_TIMEOUT       milliseconds,
         BOOL                fFullTransfer
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client will send data to. The type of the pipe must be either BULK or INTERRUPT.
- `pBuffer` is a buffer that contains data which will be transferred to the target pipe. This cannot be NULL.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate very large transfers.
- `milliseconds` is the request timeout in ms.
- `fFullTransfer` is a flag to set/clear the TRANSFER_FLAG_POST_BREAK in the URB. If this flag is set, and the transfer would not end with a short packet, (that is, the size of the buffer is N * Max Packet Size of the pipe), then the USBD will ensure that a ZLP will be sent as part of the transfer. For the details of the URB concept and the flag, refer to section 4 in [USBDI].

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_WRITE_BULKINT_PIPE_CB_FN,                     \
__TMS_VOID,                                         \
        (                                           \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_CONST __TMS_VOID *        /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesWritten */  \
        ))
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pBuffer` is a buffer that the client provided.
- `sizeBuffer` is the size of the buffer that the client provided.
- nBytesWritten is the size of the data which is transferred to the target device actually.

Example

Let's assume the target device implements a loop back protocol so that if the host sends data to a specific BULK or INTERRUPT OUT pipe, the device will return the exact same data to the host via the other BULK or INTERRUPT IN pipe. In this example, the client sends randomly generated data to a specific BULK OUT pipe (The pipe index is 8.), receives the data from the other BULK IN pipe (The pipe index is 9. The pipe index 8 and 9 are pair endpoints.), then checks if the sent data and received data are identical.

```
/* Step 1: Generate a random data and Send it to the pipe index 8.  */
#define   TEST_OUT_PIPE_INDEX  8
#define   TEST_IN_PIPE_INDEX   (TEST_OUT_PIPE_INDEX + 1)
UsbPumpSampleGcdClient_GenerateTestData(
          pGcdClient->Buffer,
          sizeof(pGcdClient->Buffer)
          );
pGcdClient->FunctionInCall.GenDrv.pWriteBulkIntPipeFn(
          pGcdClient->FunctionHandle,
          UsbPumpSampleGcd_Client_WriteBulkIntCbFn,
                  hPipe,
                  pBuffer,
                  sizeBuffer,
                  milliseconds,
                  fFullTransfer
          );

/*
|| Step 2: In the callback routine of the writing operation,
|| read data from the pipe index 9.
*/
static VOID
UsbPumpSampleGcd_Client_WriteBulkIntCbFn(
          VOID *      pCallbackCtx,
          VOID *      pRequestHandle,
          USBPUMP_USBDI_GENERIC_STATUS     ErrorCode,
```

```
        VOID CONST *        pBuffer,
        BYTES       sizeBuffer,
        BYTES       nBytesWritten
        )
        {
        USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pCallbackCtx;
        USBPUMP_UNREFERENCED_PARAMETER(pRequestHandle);
        USBPUMP_UNREFERENCED_PARAMETER(pBuffer);
        USBPUMP_UNREFERENCED_PARAMETER(sizeBuffer);
        USBPUMP_UNREFERENCED_PARAMETER(nBytesWritten);

        if (ErrorCode != USBPUMP_USBDI_GENERIC_STATUS_OK)
            {
            TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                UDMASK_ERRORS,
                "?UsbPumpSampleGcd_Client_WriteBulkIntCbFn:"
                " ErrorCode:%s(%d)\n",
                UsbPumpUsbdiClassGeneric_StatusName(ErrorCode),
                ErrorCode
                ));
            }
        else
            {
            TTUSB_OBJPRINTF((pGcdClient->pClassObject,
                UDMASK_ENTRY,
                " UsbPumpSampleGcd_Client_WriteBulkIntCbFn:"
                " pBuffer:%p, sizeBuffer:%d, nBytesWritten:%d\n",
                pBuffer, sizeBuffer, nBytesWritten
                ));
            UsbPumpSampleGcd_Client_ClearTestData(
                pGcdClient->Buffer,
                sizeof(pGcdClient->Buffer)
                );
            UsbPumpSampleGcd_Client_ReadBulkInt(
                pGcdClient,
                pGcdClient->hPipes[TEST_IN_PIPE_INDEX],
                pGcdClient->Buffer,
                sizeof(pGcdClient->Buffer),
                /* Timeout */ 5000 /* ms */
                );
            }
        }

/*
|| Step 3: In the callback routine of the reading operation,
|| check the sent data and the received data.
*/
static VOID
UsbPumpSampleGcd_Client_ReadBulkIntCbFn(
```

```
VOID *       pCallbackCtx,
VOID *       pRequestHandle,
USBPUMP_USBDI_GENERIC_STATUS     ErrorCode,
VOID *       pBuffer,
BYTES        sizeBuffer,
BYTES        nBytesWritten
)
{
USBPUMP_CLASS_GCD_CLIENT_DATA * CONST    pGcdClient = pCallbackCtx;
USBPUMP_UNREFERENCED_PARAMETER(pRequestHandle);
USBPUMP_UNREFERENCED_PARAMETER(pBuffer);
USBPUMP_UNREFERENCED_PARAMETER(sizeBuffer);
USBPUMP_UNREFERENCED_PARAMETER(nBytesWritten);

if (ErrorCode != USBPUMP_USBDI_GENERIC_STATUS_OK)
    {
    TTUSB_OBJPRINTF((pGcdClient->pClassObject,
        UDMASK_ERRORS,
        "?UsbPumpSampleGcd_Client_ReadBulkIntCbFn:"
        " ErrorCode:%s(%d)\n",
        UsbPumpUsbdiClassGeneric_StatusName(ErrorCode),
        ErrorCode
        ));
    }
else
    {
    BOOL fResult;

    fResult = UsbPumpSampleGcd_Client_CheckTestData(
            pBuffer,
            nBytesWritten
            );

    USBPUMP_DEBUG_PARAMETER(fResult);

    TTUSB_OBJPRINTF((pGcdClient->pClassObject,
        UDMASK_ANY,
        " UsbPumpSampleGcd_Client_ReadBulkIntCbFn:"
        " pBuffer:%p, sizeBuffer:%d, nBytesWritten:%d,"
        " check:%d\n",
        pBuffer, sizeBuffer, nBytesWritten, fResult
        ));

    UsbPumpSampleGcd_Client_ResetPipe(
        pGcdClient,
        pGcdClient->hPipes[TEST_OUT_PIPE_INDEX],
        RESET_PIPE_FLAGS
        );
    }
```

```
}
```

4.2.1.11 ReadStreamPipe Operation

Description

The client uses this operation to receive data from a bulk STREAM pipe of the target device. This operation performs an STREAM IN transfer. This operation will return the result code of the operation and data transferred from the pipe via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_READ_STREAM_PIPE_FN(
        USBPUMP_SESSION_HANDLE                              FunctionHandle,
        USBPUMP_USBDI_GENERIC_READ_STREAM_PIPE_CB_FN *      pCallBack,
        VOID *                                              pCallBackContext,
        USBPUMP_USBDI_PIPE_HANDLE                           hPipe,
        VOID *                                              pBuffer,
        BYTES                                               sizeBuffer,
        USBPUMP_USBDI_GENERIC_TIMEOUT                       milliseconds,
        UINT16                                              StreamID
        );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.

- `hPipe` is the handle of the pipe that the client will receive data from. The type of the pipe must be STREAM.

- `pBuffer` is a buffer that will store data transferred from the target pipe. This cannot be NULL.

- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate very large transfers.

- `Milliseconds` is the request timeout in ms.

- `StreamID` is the request timeout in ms.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_READ_STREAM_PIPE_CB_FN,                   \
__TMS_VOID,                                         \
         (                                          \
         __TMS_VOID *                /* pClientContext */,    \
         __TMS_VOID *                /* pRequestHandle */,    \
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */, \
         __TMS_VOID *                /* pBuffer */,       \
         __TMS_BYTES                 /* sizeBuffer */,    \
         __TMS_BYTES                 /* nBytesRead */     \
         ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer that the client provided. This buffer contains the data from the target stream pipe.

- `sizeBuffer` is the size of the buffer that the client provided. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.

- `nBytesRead` is the size of the data which is transferred from the target device actually.

4.2.1.12 WriteStreamPipe Operation

Description

The client uses this operation to send data to a bulk STREAM pipe of the target device. This operation performs an STREAM OUT transfer. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_WRITE_STREAM_PIPE_FN(
        USBPUMP_SESSION_HANDLE                              FunctionHandle,
        USBPUMP_USBDI_GENERIC_WRITE_ISOCH_PIPE_CB_FN *      pCallBack,
        VOID *                                              pCallBackContext,
        USBPUMP_USBDI_PIPE_HANDLE                           hPipe,
        CONST VOID *                                        pBuffer,
        BYTES                                               sizeBuffer,
        USBPUMP_USBDI_GENERIC_TIMEOUT                       milliseconds,
        BOOL                                                fFullTransfer,
        UINT16                                              StreamID
        );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client will receive data from. The type of the pipe must be either BULK or INTERRUPT.
- `pBuffer` is a buffer that contains data which will be transferred to the target pipe. This cannot be NULL.

- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.

- `milliseconds` is the request timeout in ms.

- `fFullTransfer` is a flag to set/clear the TRANSFER_FLAG_POST_BREAK in the URB. If this flag is set, and the transfer would not end with a short packet, (that is, the size of the buffer is N * Max Packet Size of the pipe), then the USBD will ensure that a ZLP will be sent as part of the transfer. For the details of the URB concept and the flag, refer to section 4 in [USBDI].

- `StreamID` is the Stream ID.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_WRITE_STREAM_PIPE_CB_FN,                  \
__TMS_VOID,                                         \
         (                                          \
         __TMS_VOID *                /* pClientContext */,    \
         __TMS_VOID *                /* pRequestHandle */,    \
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */, \
         __TMS_CONST __TMS_VOID *         /* pBuffer */,       \
         __TMS_BYTES                 /* sizeBuffer */,    \
         __TMS_BYTES                 /* nBytesWritten */  \
         ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer that the client provided.

- `sizeBuffer` is the size of the buffer that the client provided. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.

- `nBytesWritten` is the size of the data which is transferred to the target device actually.

4.2.1.13 ReadIsochPipe Operation

Description

The client uses this operation to receive data from an ISOCHRONOUS pipe of the target device. This operation performs an ISOCHRONOUS IN transfer. This operation will return the result code of the operation and data transferred from the pipe via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_READ_ISOCH_PIPE_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_READ_ISOCH_PIPE_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE        hPipe,
         VOID *                   pBuffer,
         BYTES                    sizeBuffer,
         USBPUMP_USBDI_GENERIC_TIMEOUT        milliseconds,
         USBPUMP_ISOCH_PACKET_DESCR *     pIsochDescr,
         BYTES                    IsochDescrSize,
         UINT32                   IsochStartFrame,
         BOOL                fAsap
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.

- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.

- `pCallBackContext` is provided by the Client to be used for the callback routine.

- `hPipe` is the handle of the pipe that the client will receive data from. The type of the pipe must be either BULK or INTERRUPT.

- `pBuffer` is a buffer that will store data transferred from the target pipe. This cannot be NULL.

- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate very large transfers.

- `milliseconds` is the request timeout in ms.

- `pIsochDescr` is a pointer to buffer containing the packet-by-packet isochronous descriptor information. Refer to section 6.3.1 and Table 6 in [USBDI] for a description of this structure and its use.

- `IsochDescrSize` is the size of the buffer that `pIsochDescr` refers to.

- `IsochStartFrame` is the frame to use as the starting frame for the transfer.

- `fAsap` – If this flag is set, the transfer in this URB will be started as soon as possible. Otherwise, the starting frame number is taken from `IsochStartFrame`.

Callback Routine

```
__TMS_FNTYPE_DEF(                                   \
USBPUMP_USBDI_GENERIC_READ_ISOCH_PIPE_CB_FN,                    \
__TMS_VOID,                                         \
        (                                           \
        __TMS_VOID *                /* pClientContext */,    \
        __TMS_VOID *                /* pRequestHandle */,    \
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */, \
        __TMS_VOID *                /* pBuffer */,       \
        __TMS_BYTES                 /* sizeBuffer */,    \
        __TMS_BYTES                 /* nBytesRead */,    \
        __TMS_USBPUMP_ISOCH_PACKET_DESCR *      /* pIsochDescr */,     \
        __TMS_BYTES                 /* IsochDescrSize */,    \
        __TMS_UINT32                /* IsochStartFrame */,   \
        __TMS_BYTES                 /* nIsochErrs */ \
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.

- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

- `pBuffer` is a buffer that the client provided. This buffer contains the data from the target isochronous pipe.

- `sizeBuffer` is the size of the buffer that the client provided. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.
- `nBytesRead` is the size of the data which is transferred from the target device actually.
- `pIsochDescr` is a pointer to buffer containing the packet-by-packet isochronous descriptor information.
- `IsochDescrSize` is the size of the buffer that `pIsochDescr` refers to.
- `IsochStartFrame` is the actual starting frame number.
- `nIsochErrs` is the number of the isochronous transfer errors.

4.2.1.14 WriteIsochPipe Operation

Description

The client uses this operation to send data to an ISOCHRONOUS pipe of the target device. This operation performs an ISOCHRONOUS OUT transfer. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_WRITE_ISOCH_PIPE_FN(
         USBPUMP_SESSION_HANDLE          FunctionHandle,
         USBPUMP_USBDI_GENERIC_WRITE_ISOCH_PIPE_CB_FN * pCallBack,
         VOID *                  pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE       hPipe,
         CONST VOID *            pBuffer,
         BYTES                   sizeBuffer,
         USBPUMP_USBDI_GENERIC_TIMEOUT       milliseconds,
         USBPUMP_ISOCH_PACKET_DESCR *    pIsochDescr,
         BYTES                   IsochDescrSize,
         UINT32                  IsochStartFrame,
         BOOL               fAsap
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client will receive data from. The type of the pipe must be either BULK or INTERRUPT.
- `pBuffer` is a buffer that contains data which will be transferred to the target pipe. This cannot be NULL.
- `sizeBuffer` is the size of the buffer that `pBuffer` refers to. This cannot be zero. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.
- `milliseconds` is the request timeout in ms.
- `pIsochDescr` is a pointer to buffer containing the packet-by-packet isochronous descriptor information. Refer to section 6.3.1 and Table 6 in [USBDI] for a description of this structure and its use.
- `IsochDescrSize` is the size of the buffer that `pIsochDescr` refers to.
- `IsochStartFrame` is the frame to use as the starting frame for the transfer.
- `fAsap` – If this flag is set, the transfer in this URB shall be started as soon as possible. Otherwise, the starting frame number is taken from `IsochStartFrame`.

Callback Routine

```
__TMS_FNTYPE_DEF(                                \
USBPUMP_USBDI_GENERIC_WRITE_ISOCH_PIPE_CB_FN,                  \
__TMS_VOID,                                      \
         (                                       \
         __TMS_VOID *                /* pClientContext */,    \
         __TMS_VOID *                /* pRequestHandle */,    \
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */, \
         __TMS_CONST __TMS_VOID *         /* pBuffer */,       \
         __TMS_BYTES                 /* sizeBuffer */,    \
         __TMS_BYTES                 /* nBytesWritten */,     \
```

```
__TMS_USBPUMP_ISOCH_PACKET_DESCR *       /* pIsochDescr */,     \
__TMS_BYTES                  /* IsochDescrSize */,    \
__TMS_UINT32                 /* IsochStartFrame */,   \
__TMS_BYTES                  /* nIsochErrs */ \
));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `pBuffer` is a buffer that the client provided.
- `sizeBuffer` is the size of the buffer that the client provided. The size of the buffer can be bigger than 64K because MCCI's USBD can accommodate large transfers.
- `nBytesWritten` is the size of the data which is transferred to the target device actually.
- `pIsochDescr` is a pointer to buffer containing the packet-by-packet isochronous descriptor information.
- `IsochDescrSize` is the size of the buffer that `pIsochDescr` refers to.
- `IsochStartFrame` is the actual starting frame number.
- `nIsochErrs` is the number of the isochronous transfer errors.

4.2.1.15 AbortPipe Operation

Description

The client uses this operation to abort all pending I/O for a specific pipe. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_ABORT_PIPE_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_ABORT_PIPE_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE        hPipe
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client wishes to abort.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_ABORT_PIPE_CB_FN,
__TMS_VOID,
         (
         __TMS_VOID *                /* pClientContext */,
         __TMS_VOID *                /* pRequestHandle */,
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */,
         __TMS_USBPUMP_USBDI_PIPE_HANDLE          /* hPipe */
         ));
```

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `hPipe` is the handle of the pipe which was aborted by a client request.

4.2.1.16 ResetPipe Operation

Description

The client uses this operation to reset-pipe operation to USBD for a specific pipe. This operation clears out stall conditions and so forth. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_RESET_PIPE_FN(
         USBPUMP_SESSION_HANDLE           FunctionHandle,
         USBPUMP_USBDI_GENERIC_RESET_PIPE_CB_FN * pCallBack,
         VOID *                   pCallBackContext,
         USBPUMP_USBDI_PIPE_HANDLE        hPipe,
         UINT32                   ResetPipeFlags
         );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.
- `hPipe` is the handle of the pipe that the client wishes to reset.
- `ResetPipeFlags` is the reset pipe control flag.

Callback Routine

```
__TMS_FNTYPE_DEF(
```

```
USBPUMP_USBDI_GENERIC_RESET_PIPE_CB_FN,
__TMS_VOID,
        (
        __TMS_VOID *                /* pClientContext */,
        __TMS_VOID *                /* pRequestHandle */,
        __TMS_USBPUMP_USBDI_GENERIC_STATUS      /* ErrorCode */,
        __TMS_USBPUMP_USBDI_PIPE_HANDLE         /* hPipe */
        ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.
- `hPipe` is the handle of the pipe which was reset by a client request.

4.2.1.17 CyclePort Operation

Description

The client uses this operation to simulate a device removal and re-insertion at the root hub. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns the request handle immediately and the client can cancel the request using the request handle.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`
- `USBPUMP_USBDI_GENERIC_STATUS_BUSY`
- `Error codes from USBD layer (Refer to section 7.2)`

Declaration of Prototype

```
typedef VOID *                  // pRequestHandle
USBPUMP_USBDI_GENERIC_CYCLE_PORT_FN(
        USBPUMP_SESSION_HANDLE          FunctionHandle,
        USBPUMP_USBDI_GENERIC_CYCLE_PORT_CB_FN * pCallBack,
        VOID *                  pCallBackContext
        );
```

Return Value

- `pRequestHandle` is the request handle for this operation and is returned to the client immediately. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document for canceling a request.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_CYCLE_PORT_CB_FN,
__TMS_VOID,
         (
         __TMS_VOID *                /* pClientContext */,
         __TMS_VOID *                /* pRequestHandle */,
         __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */
         ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.

4.2.1.18 SuspendDevice Operation

Description

The client uses this operation to suspend the device which is bound to the GCD function instance. This operation will return the result code of the operation via the callback routine that the client provided. This operation returns immediately and the device has not been suspened yet. When the client receives USBPUMP_USBDI_GENERIC_EVENT_DEVICE_SUSPENDED function event notification, the device is in the suspend mode.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`

- USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_SUSPEND_DEVICE_FN(
          USBPUMP_SESSION_HANDLE           FunctionHandle,
          USBPUMP_USBDI_GENERIC_SUSPEND_DEVICE_FN * pCallBack,
          VOID *                  pCallBackContext
          );
```

Return Value

- `pRequestHandle` is always NULL because this request cannot be cancelled.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_SUSPEND_DEVICE_CB_FN,
__TMS_VOID *,                     /* pRequestHandle */
          (
          __TMS_VOID *                /* pClientContext */,
          __TMS_VOID *                /* pRequestHandle */,
          __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */
          ));
```

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.
- `ErrorCode` is the result code of the operation.

4.2.1.19 ResumeDevice Operation

Description

The client uses this operation to resume the device which is bound to the GCD function instance. This operation will return the result code of the operation via the callback routine that

the client provided. This operation returns immediately and the device has not been resumed yet. When the client receives USBPUMP_USBDI_GENERIC_EVENT_DEVICE_RESUMED function event notification, the device is in the normal mode.

This API will return one of the following return codes through the callback routine:

- `USBPUMP_USBDI_GENERIC_STATUS_OK`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_FUNCTION_HANDLE`
- `USBPUMP_USBDI_GENERIC_STATUS_FUNCTION_NOT_OPENED`
- `USBPUMP_USBDI_GENERIC_STATUS_INVALID_PARAMETER`

Declaration of Prototype

```
typedef VOID *                    // pRequestHandle
USBPUMP_USBDI_GENERIC_RESUME_DEVICE_FN(
          USBPUMP_SESSION_HANDLE           FunctionHandle,
          USBPUMP_USBDI_GENERIC_RESUME_DEVICE_CB_FN * pCallBack,
          VOID *                   pCallBackContext
          );
```

Return Value

- `pRequestHandle` is always NULL because this request cannot be cancelled.

Parameters

- `FunctionHandle` is the function session handle returned from OpenFunction Class In-Call.
- `pCallBack` is the pointer to the callback routine provided by the Client to return the result of this operation.
- `pCallBackContext` is provided by the Client to be used for the callback routine.

Callback Routine

```
__TMS_FNTYPE_DEF(
USBPUMP_USBDI_GENERIC_RESUME_DEVICE_CB_FN,
__TMS_VOID,
          (
          __TMS_VOID *                /* pClientContext */,
          __TMS_VOID *                /* pRequestHandle */,
          __TMS_USBPUMP_USBDI_GENERIC_STATUS       /* ErrorCode */
          ));
```

Parameters

- `pClientContext` is the `pCallBackContext` which the Client provided to the operation.
- `pRequestHandle` is the request handle for this operation and is returned to the client. The client can cancel using this request handle. Refer to section 4.2.1.2 in this document.

- `ErrorCode` is the result code of the operation.

4.2.2 Function Out-Calls

4.2.2.1 Notification Operation

The client has to implement this operation and provide the pointer to the function when calling the open function Class In-Call. If a function notification is available in the GCD, this operation will be invoked to send the notification to the client. For the details of this Out-Call, refer to section 3.2.1 in [CLASSKIT].

5 Generic Class Driver API

5.1 GCD Configuration API

5.1.1 USBPUMP_USBDI_CLASS_GENERIC_CONFIG_INIT_V1

Description

The client uses this macro to initialize a GCD private configuration structure at compile-time.

Declaration of Prototype

```
#define   __TMS_USBPUMP_USBDI_CLASS_GENERIC_CONFIG_INIT_V1(     \
          a_MaxSession,                                 \
          a_NumConfig,                      \
          a_NumIfc,                             \
          a_NumAlt,                         \
          a_NumPipe,                            \
          a_NumRequest,                                 \
          a_MaxConfigDescSize                           \
```

) **Parameters**

- `a_MaxSession` is the maximum number of client sessions that the GCD supports. The client sessions includes both class sessions and function sessions. For example, if the a_MaxSession is 10, the GCD supports up to 5 class sessions and 5 function session, or 1 class session and 9 function sessions.

- `a_NumConfig` is the number of configurations which are owned by the USB device that GCD supports.

- `a_NumIfc` is the number of interfaces which are owned by the USB device that GCD supports. This means the number of the all interfaces the USB device has. For

example, if the number of configurations is 2 and each configuration has 3 interfaces, this argument has to be 6 (2 * 3).

- `a_NumAlt` is the number of alternative settings which are owned by the USB device that GCD supports. This also means the number of the all alternative settings the USB device has. For example, if the number of configurations is 2, each configuration has 3 interfaces and each interface has 2 alternative settings, this argument has to be 12 (2 * 3 * 2). And because an interface has at least one alternative setting by default, the a_MaxAlt must not be less than a_MaxIfc.

- `a_NumPipe` is the number of pipes which are owned by the USB device that GCD supports. This also means the number of the all pipes the USB device has. For example, if the number of configurations is 2, each configuration has 3 interfaces, each interface has 2 alternative settings and each alternative setting has 4 pipes, this argument has to be 48 (2 * 3 * 2 * 4). This argument doesn't count the default control pipe.

- `a_NumRequest` is the number of client requests that GCD is able to handle at the same time. This is optional so you can pass zero for this. If this is zero, the GCD sets the default value for this. (c.f. Default NumRequest = a_NumPipe / a_NumConfig)

- a_MaxConfigDescSize is the maximum size (bytes) of the configuration bundle the GCD supports.

Example

```
/*
|| Configuration for generic device class driver
*/
       static
       CONST USBPUMP_USBDI_CLASS_GENERIC_CONFIG sk_UsbPumpUsbdGeneric_PrivateConfig =
          USBPUMP_USBDI_CLASS_GENERIC_CONFIG_INIT_V1(
              /* number of client sessions */ 4,
              /* number of configurations */ 1,
              /* number of interfaces */ 5,
              /* number of alternative settings */5,
              /* number of pipes */30,
              /* number of requests */4,
              /* maximum size of configuration bundle */ 512
              );
```

5.1.2 USBPUMP_USBDI_CLASS_GENERIC_CONFIG_SETUP_V1

Description

The client uses this macro to initialize a GCD private configuration structure at run-time.

Declaration of Prototype

```
#define   __TMS_USBPUMP_USBDI_CLASS_GENERIC_CONFIG_SETUP_V1(
```

```
a_pConfig,
a_MaxSession,                           \
a_NumConfig,                    \
a_NumIfc,                           \
a_NumAlt,                       \
a_NumPipe,                          \
a_NumRequest,                           \
a_MaxConfigDescSize                     \
)
```

Parameters

- `a_pConfig` is a pointer to GCD private configuration structure that will be configured by this macro in run-time. The type of this parameter must be USBPUMP_USBDI_CLASS_GENERIC_CONFIG.

- `a_MaxSession, a_NumConfig, a_NumIfc, a_NumAlt, a_NumPipe, a_NumRequest, a_MaxConfigDescSize` – Refer to parameters in section 5.1.1.

Example

```
USBPUMP_USBDI_CLASS_GENERIC_CONFIG   GcdConfig;

USBPUMP_USBDI_CLASS_GENERIC_CONFIG_SETUP_V1(
    &GcdConfig,
    /* number of client sessions */ 4,
    /* number of configurations */ 1,
    /* number of interfaces */ 5,
    /* number of alternative settings */ 10,
    /* number of pipes */ 20,
    /* number of requests */ 20,
    /* maximum size of configuration bundle */ 512
    );
```

5.2 GCD API Functions

5.2.1 UsbPumpUsbdiClassGeneric_Initialize

Description

This API function initializes the generic class driver. This function creates the generic class driver, along with all the idle instance objects for the generic class, and registers them all with USBDI. The client does not need to call this function explicitly but needs to set the function pointer of this function to `USBPUMP_HOST_DRIVER_CLASS_INIT_NODE.` (Refer to section 2.2.4.)

5.2.2 UsbPumpUsbdiClassGeneric_StatusName

Description

This API function returns a string name for a specific status code returned from GCD Class and Function In-Calls. Refer to section 6 for the GCD status code list.

Declaration of Prototype

```
__TMS_CONST __TMS_TEXT *
UsbPumpUsbdiClassGeneric_StatusName(
          __TMS_USBPUMP_USBDI_GENERIC_STATUS
          );
```

Return Value

- A string name for a GCD status code passed as an argument.

Parameters

- A GCD status code returned from GCD Class and Function In-Call.

6 Generic Class Driver Event Notifications

A client of GCD passes a Class Out-Call buffer which contains only the function pointer to the Class Event Notification function when it calls UsbPumpObject_OpenSession(). And it passes a Function Out-Call buffer which contains only the function pointer to the Function Event Notification function. The GCD notifies the client of class eventst through the Class Out-Call, and function events through the Function Out-Call.

6.1 Class Event Notifications

Table 2 Class Event Notifications

Event Code	Description
USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL	A USB device for GCD is attached to this host. The device is ready to use. A client can open a function session for this device instance to control it using GCD Function In-Calls.
USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE	A USB device for GCD is detached from this host. The device can not be used any longer. A client which opened a function session to this device instance should close the function session.
USBPUMP_CLASSKIT_EVENT_FUNCTION_OPEN	A function session to the device instance to which a client opened a class session has been opened. If another client opened the function session to the device instance, this client is not able to open a function session until the function session is closed.

Event Code	Description
USBPUMP_CLASSKIT_EVENT_FUNCTION_CLOSE	A function session to the device instance to which a client opened a class session has been closed. If another client closed the function session to the device instance, this client is able to open a function session.

6.2 Function Event Notifications

Table 3 Function Event Notifications

Event Code	Description
USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL	Refer to section 6.1.
USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE	Refer to section 6.1.
USBPUMP_CLASSKIT_EVENT_FUNCTION_OPEN	Refer to section 6.1.
USBPUMP_CLASSKIT_EVENT_FUNCTION_CLOSE	Refer to section 6.1.
USBPUMP_USBDI_GENERIC_EVENT_DEVICE_SUSPENDED	The USB device to which this client opens a function session is suspended. To escape from the suspend mode, the client should call ResumeDevice Function In operation (refer to section 4.2.1.17) to resume the device.
USBPUMP_USBDI_GENERIC_EVENT_DEVICE_RESUMED	The USB device to which this client opens a function session is resumed from suspend mode.

7 Generic Class Driver Status Codes

7.1 Generic Class Driver Status Codes

Following GCD status codes are returned by the GCD Class and Function In-Calls. To get string forms of GCD status codes for debugging purpose, use UsbPumpUsbdiClassGeneric_StatusName() GCD API function (refer to section 5.2.2). The prefix of the status codes in below table is "USBPUMP_USBDI_GENERIC_STATUS_".

Table 4 Generic Class Driver Status Codes

Status Code	Description
OK	The GCD returns this status code when it handles an In-Call successfully.

Status Code	Description
INVALID_PARAMETER	This status code is returned at the open session API and open function API when a client passed invalid parameters like NULL In-Call buffer.
ARG_AREA_TOO_SMALL	This status code is returned when the open request memory is too small.
BUFFER_TOO_SMALL	This status code is returned at the open session API when the In-Call buffer size or the Out-Call buffer size are too small.
NOT_SUPPORTED	This status code is returned when a client invokes an unsupported In-Call.
NO_MORE_SESSIONS	The GCD returns this status code when a client tries to open a class or function session and there is no free session to open. The maximum number of sessions is configured by the Class Driver configuration.
INVALID_SESSION_HANDLE	The GCD returns this status code if the return value of UsbPumpClassKitI_ValidateSessionHandle() to the session passed into a Class In-Call is NULL. For the session handle validation API, refer to section 3.7.1 in [ClassKit].
INVALID_FUNCTION_HANDLE	The GCD returns this status code if the return value of UsbPumpClassKitI_ValidateSessionHandle() to the session passed into a Function In-Call, is NULL.
FUNCTION_ALREADY_OPENED	The GCD returns this status code if a client tries to open a function session to a specific function instance when another client has already opened a session to the specific function instance.
FUNCTION_NOT_OPENED	The GCD returns this status code when a client calls a Function In-Call using a function session that is already closed or never opened.
INTERNAL_ERROR	The GCD returns this status code when it encounters unknown internal error.
NO_MEMORY	The GCD returns this status code if it fails to allocate the client request.
ALREADY_COMPLETED	The GCD returns this status code when a client tries to cancel a request that is already completed.

7.2 Error Codes From USBD

Following GCD status codes are also returned by the GCD Class and Function In-Calls but these status codes contains USBD layer error code information. To get string forms of GCD status codes for debugging purpose, use UsbPumpUsbdiClassGeneric_StatusName() GCD API function (refer to section 5.2.2). The prefix of the status codes in below table is "USBPUMP_USBDI_GENERIC_STATUS_".

Table 5 Generic Class Driver USBD Error Codes

Status Code	Description
USBD_USTAT_BUSY	This status code is not an error code and indicates that the operation is now in process in USBD layer.
USBD_USTAT_KILL	This status code indicates that the operation was cancelled in USBD layer.
USBD_USTAT_IOERR	This status code indicates that some kind of unrecoverable device error occurred in USBD layer.
USBD_USTAT_STALL	This status code is returned when a STALL pid was received, or the specified pipe is stalled in USBD layer.
USBD_USTAT_LENGTH_OVERRUN	This status code is returned when the amount of data returned by the device exceeded the maximum packet size or the remaining buffer size, as applicable in USBD layer.
USBD_USTAT_LENGTH_UNDERRUN	This status code is returned when short packet was received, and short packets were not indicated to be OK on the transfer in USBD layer.
USBD_USTAT_INVALID_PARAM	This status code is returned when an invalid parameter was received in USBD layer.
USBD_USTAT_NOHW	This status code is returned when the host controller hardware has been removed from the system, powered down or made inaccessible in some way in USBD layer.
USBD_USTAT_IN_USE	This status code is returned when the operation could not be performed because some other operation is referencing the element. For example, this will be returned if the client attempts to do a SET_CONFIG while I/O is in progress in USBD layer.
USBD_USTAT_NO_MEMORY	This status code is returned when the operation could not be performed because an attempt to allocate from a memory pool failed in USBD layer.
USBD_USTAT_NO_BANDWIDTH	This status code is returned when the operation could not be performed because there's not enough bus bandwidth availalble in USBD layer.
USBD_USTAT_NO_BUS_POWER	This status code is returned when the operation could not be performed because there's not enough bus power available in USBD layer.
USBD_USTAT_INTERNAL_ERROR	This status code is returned when the operation could not be performed due to an internal consistency-check failure of some kind in USBD layer.