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 Embedded Mass Storage Class Driver V2 User's Guide

Engineering Report 950001350

Rev. A

Date: 2014-03-18

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.

Document Release History

Rev A	2014-03-18	Initial Release

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Purpose

This documentation describes the Mass Storage Class Driver (MSD) V2 API provided by the MCCI USB DataPump Embedded Host / On-The-Go host stack.

1.2 Scope

The embedded Mass Storage Class driver (MSD) supports devices compliant with the **USB Mass Storage Class Bulk-Only Transport** specification. Concepts from the USB and MSC (Mass Storage Class) specifications are used but not explained in this documentation.

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

ClassKit Class Driver Development Kit. This is the software component that provides common routines to Class Drivers.

Device The hardware component that provides the USB descriptors and data to the MSC driver.

Driver The software component that provides low-level access to MSC devices.

MSC Mass Storage Class

MSD Mass Starage class Driver

LUN Logical unit. This is typically defined as a “drive” of a Mass Storage Class Device.

SCSI Small computer system interface. A protocol designed to provide an efficient peer-to-peer I/O bus with up to 16 devices, including one or more hosts.

SRB SCSI Request Block. A structure that describes the SCSI request issued to the MSC driver. The request must supply a properly formatted CDB.

CDB SCSI Command Data Block. A structure that describes the SCSI command issued to the device. Refer to the SCSI-2 specification described in the referenced documents section.

1.4 Referenced Documents

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

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

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

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

[USBMSC-BOT] *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, version 1.0. This specification is available on the World Wide Web site http://www.usb.org/.

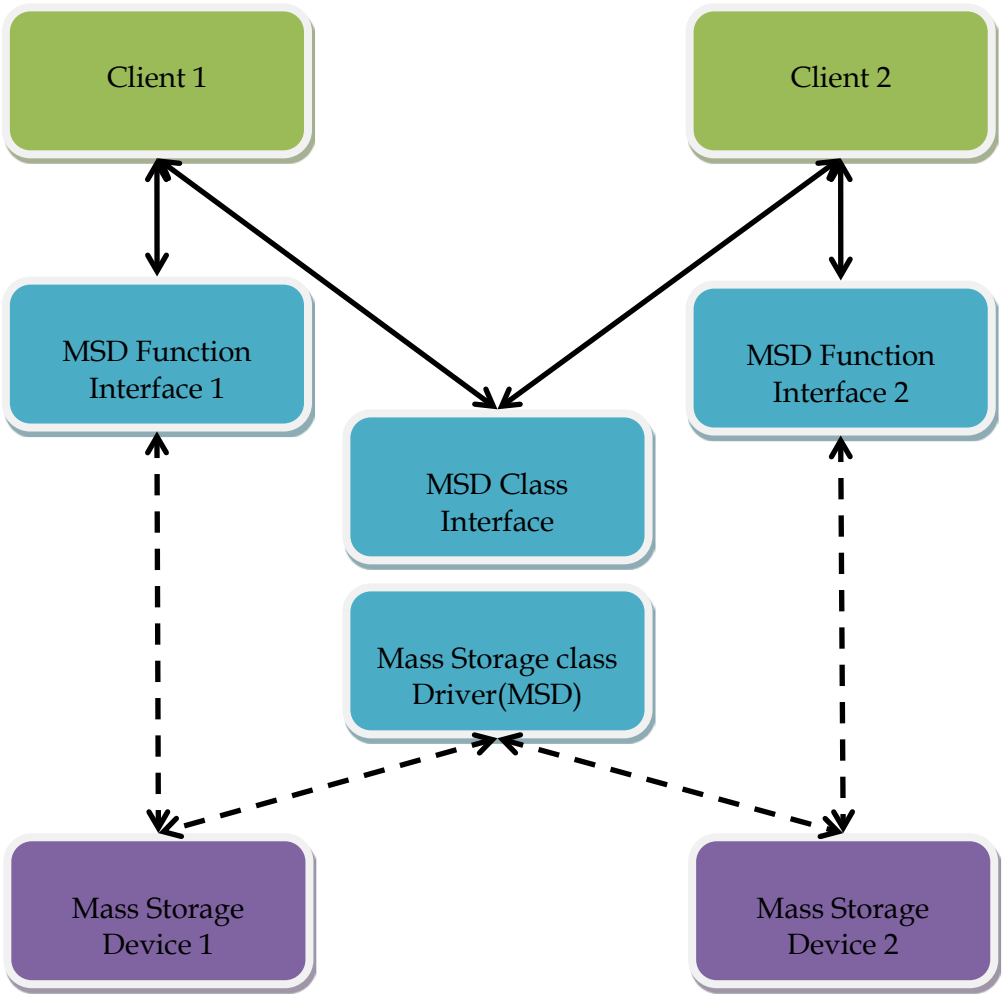
[SCSI-2] *Small Computer System Interface X3T9.2*, version 10L. This specification describes the format of the SCSI Command Data Block required as part of the SRB sent to the MSC driver. This specification is available on the World Wide Web site http://global.ihs.com/

2 Client Implementation and Use of MSC Driver

2.1 Introduction

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

Figure 1 Conceptual Diagram



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

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

Detailed descriptions of these steps are found in following sections.

2.2 Configuration and Initialization

To create a Mass Storage class Driver (hereafter, MSD) in the MCCI USB DataPump Embedded USB host stack, a client must initialize the MSD before using it. The following code illustrates how a client initialize the MSD.

2.2.1 Customize Match List Entries

```
/*
|| Match list entries for mass storage device class driver
*/
static CONST
USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY
sk_vUsbPumpUsbdiMsd_Matches[] =
   {
   USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
        "*;fc=08/06/50;*",   /* msc,scsi,bulko */
        USBPUMP_USBDI_PRIORITY_FN_CSP
        ),
   USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
        "*;fc=08/02/50;*",   /* msc,atapi,bulko */
        USBPUMP_USBDI_PRIORITY_FN_CSP
        ),
   USBPUMP_USBDI_INIT_MATCH_LIST_ENTRY_INIT_V1(
        "*;fc=08/05/50;*",   /* msc,fdd,bulko */
        USBPUMP_USBDI_PRIORITY_FN_CSP
        )
   };

CONST
USBPUMP_USBDI_INIT_MATCH_LIST
gk_UsbPumpUsbdiMsd_InitMatchList =
   USBPUMP_USBDI_INIT_MATCH_LIST_INIT_V1(
        sk_vUsbPumpUsbdiMsd_Matches
        );
```

2.2.2 Configuration for MSD Class

In the MSD 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. If a client want to use the Mass Storage Class driver default configuration, a client can use `gk_UsbPumpUsbdiMsd_ClassConfigDefault`. This default configuration defines as below.

```
/*
|| Configuration for MSD Class
*/
```

```
CONST
USBPUMP_USBDI_DRIVER_CLASS_CONFIG
gk_UsbPumpUsbdiMsd_ClassConfigDefault =
   USBPUMP_USBDI_DRIVER_CLASS_CONFIG_INIT_V1(
       &gk_UsbPumpUsbdiMsd_InitMatchList,
       NULL, /* use implementation default driver class name */
       NULL, /* use default function instance name */
       USBPUMP_USBDI_CLASS_MSC_MAX_INSTANCE
       );
```

2.2.3 Configuration for MSD Private

In the MSD private configuration, you can set the maximum number of client sessions. The client sessions include both class sessions and function sessions. Refer to section 5.1.1.

If a client wants to use the MSD default configuration, a client can use `gk_UsbPumpUsbdiClassMsd_ConfigDeafult`. This default configuration defines as below. It has maximum two sessions and it can be override by using compile-time parameter.

```
/*
|| Private Configuration for generic device class driver
*/
#ifndefUSBPUMP_USBDI_CLASS_MSD_CONFIG_MAX_SESSION   /* PARAM */
# define USBPUMP_USBDI_CLASS_MSD_CONFIG_MAX_SESSION  2
#endif

CONST
USBPUMP_USBDI_CLASS_MSD_CONFIG
gk_UsbPumpUsbdiClassMsd_ConfigDeafult =
   USBPUMP_USBDI_CLASS_MSD_CONFIG_INIT_V1(
       USBPUMP_USBDI_CLASS_MSD_CONFIG_MAX_SESSION
       );
```

2.2.4 Add the MSD initialization information to the ClassDriverInitNode Table

In the class driver initialization node, put the MSD initialization function which MSD 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 */ UsbPumpUsbdiClassMsd_Initialize,
```

```
        /* pConfig */ &gk_UsbPumpUsbdiMsd_ClassConfigDefault,
        /* pPrivateConfig */ &gk_UsbPumpUsbdiClassMsd_ConfigDeafult,
        /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS | UDMASK_USBDI
        )
    };
```

2.3 Start Client

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

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

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

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

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

2.4 Find Mass Storage Class Driver Object

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

```
USBPUMP_CLASS_MSD_CLIENT *
UsbPumpSampleMsd_Client_Create(
   UPLATFORM *  pPlatform,
   UINT32       DebugFlags,
   BYTES        DataBufferSize
   )
   {
   USBPUMP_OBJECT_ROOT *        pRootObject;
   USBPUMP_OBJECT_HEADER *      pClassObject;
   USBPUMP_CLASS_MSD_CLIENT *   pMsdClient;
   USTAT                Status;

   pRootObject = UsbPumpObject_GetRoot(&pPlatform->upf_Header);

   /*
   || Find the Mass Storage Class Driver object
   */
   pClassObject = UsbPumpObject_EnumerateMatchingNames(
               &pRootObject->Header,
               NULL,
               USBPUMP_USBDI_CLASS_MSD_NAME
               );

   /*
   || If the MSD object is found, open a class session
   */
   if (pClassObject == NULL)
       {
       TTUSB_PLATFORM_PRINTF((
           pPlatform,
           UDMASK_ERRORS,
           "?" FUNCTION ":"
           " Failed to enumerate mass storage class driver\n"
           ));
       return NULL;
       }

   /*
   || Create a sample MSD client object
   */
   pMsdClient = UsbPumpPlatform_Malloc(pPlatform, sizeof(*pMsdClient));
   if (pMsdClient == NULL)
       {
       TTUSB_PLATFORM_PRINTF((
           pPlatform,
           UDMASK_ANY | UDMASK_ERRORS,
           "?" FUNCTION ":"
           " Memory (%x bytes) allocation failed\n",
           sizeof(*pMsdClient)
```

```
            ));
        return NULL;
        }

    UsbPumpObject_Init(
        &pMsdClient->ObjectHeader,
        pPlatform->upf_Header.pClassParent,
        /* Mass Storage Class Driver Sample Client */
        UHIL_MEMTAG('M', 's', 'd', 'C'),
        sizeof(*pMsdClient),
        "sample.msd.client.mcci.com",
        &pPlatform->upf_Header,
        NULL
        );

    pMsdClient->pPlatform = pPlatform;
    pMsdClient->ObjectHeader.ulDebugFlags |= DebugFlags;
    pMsdClient->pClassObject = pClassObject;
    pMsdClient->DataBufferSize = DataBufferSize;

    /*
    || Open a MSC driver session.
    */
    Status = UsbPumpMsdClient_OpenSession(
            pMsdClient,
            pClassObject
            );

    if (Status != USTAT_OK)
        {
        TTUSB_OBJPRINTF((
            &pMsdClient->ObjectHeader,
            UDMASK_ERRORS,
            "?" FUNCTION ":"
            " Failed to open a class session(%s)\n",
            UsbPumpStatus_Name(Status)
            ));

        UsbPumpObject_DeInit(&pMsdClient->ObjectHeader);
        UsbPumpPlatform_Free(
            pPlatform,
            pMsdClient,
            sizeof(*pMsdClient)
            );
        return NULL;
        }

    return pMsdClient;
    }
```

2.5 Open a Class Session

A client has to open a Class Session (a session to the MSD Class) to use the Class Interface of the MSD 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.

When a client open a MSC driver session successfully, a client needs to save the handle of the opened session. A client can use the Mass Storage Class driver session APIs using this open session handle.

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

```
static USTAT
UsbPumpMsdClient_OpenSession(
   USBPUMP_CLASS_MSD_CLIENT *   pMsdClient,
   USBPUMP_OBJECT_HEADER *      pMsdClassObject
   )
   {
   VOID *       pOpenRequestMemory;
   CONST RECSIZE    Size = UsbPumpObject_SizeOpenSessionRequestMemory(
                    pMsdClassObject
                    );

   /*
   || Allocate Open Request memory for UsbPumpObject_OpenSession()
   */
   pOpenRequestMemory = UsbPumpPlatform_Malloc(
                pMsdClient->pPlatform,
                Size
                );
   if (pOpenRequestMemory == NULL)
       {
       TTUSB_OBJPRINTF((
           &pMsdClient->ObjectHeader,
           UDMASK_ERRORS,
           "?" FUNCTION ":"
           " Memory (%x bytes) allocation failed\n",
           Size
           ));
       return USTAT_NO_MEMORY;
       }

   UsbPumpObject_OpenSession(
       pMsdClassObject,
       pOpenRequestMemory,
       USBPUMP_API_OPEN_REQUEST_MEMORY_SIZE,
       UsbPumpMsdClient_OpenSessionCallback,
       pMsdClient,  /* pCallBackContext */
```

```
            &gk_UsbPumpUsbdiClassMsd_Guid,
            NULL,       /* pClientObject -- OPTIONAL */
            &pMsdClient->InCall.GenericCast,
            sizeof(pMsdClient->InCall),
            pMsdClient, /* pClientHandle */
            &sk_UsbPumpUsbdiMsdClient_ClassOutCall.GenericCast,
            sizeof(sk_UsbPumpUsbdiMsdClient_ClassOutCall)
            );

        return USTAT_OK;
        }

    static VOID
    UsbPumpMsdClient_OpenSessionCallback(
        VOID *          pClientContext,
        USBPUMP_SESSION_HANDLE   SessionHandle,
        UINT32          Status,
        VOID *          pOpenRequestMemory,
        RECSIZE         sizeOpenRequestMemory
        )
        {
        USBPUMP_CLASS_MSD_CLIENT * CONST pMsdClient = pClientContext;

        if (Status == USBPUMP_CLASSKIT_STATUS_OK)
            {
            TTUSB_OBJPRINTF((
                &pMsdClient->ObjectHeader,
                UDMASK_ANY,
                " " FUNCTION ":"
                " OpenSession STATUS_OK %p\n",
                SessionHandle
                ));
            /*
            || Store the session handle to the MSD object. This will be
            || used to call Class In-Calls such as OpenFunction() and
            || GetNumDevices().
            */
            pMsdClient->SessionHandle = SessionHandle;
            }
        else
            {
            TTUSB_OBJPRINTF((
                &pMsdClient->ObjectHeader,
                UDMASK_ERRORS,
                "?" FUNCTION ":"
                " OpenSession failed(%d)\n",
                Status
                ));
            }
```

```
    if (pOpenRequestMemory)
        {
        UsbPumpPlatform_Free(
            pMsdClient->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.

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

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 MSD class object.

The MSD 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 MSD. The MSD uses the Class Out-Calls for sending class event notifications.

The MSD 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_CLASSKIT_STATUS_OK`, the session is open and the requested Class In-Calls are ready for use.

After opening a session, the MSD 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 MSD sends are

USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL,
USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE,
USBPUMP_CLASSKIT_EVENT_FUNCTION_OPEN,
USBPUMP_CLASSKIT_EVENT_FUNCTION_CLOSE.

The `USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL` class notification means that a USB device that matches the MSD's match list entries was enumerated and ready to use. If the client wishes to access the USB device through the MSD 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 MSD function instance) to use the Function Interface of the MSD. 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 `USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL` class notification is provided below:

```
static VOID
UsbPumpMsdClient_ClassNotification(
   VOID *              pClientHandle,
   USBPUMP_CLASSKIT_NOTIFICATION    NotificationId,
   CONST VOID *        pNotification,
   BYTES               nNotification
   )
   {
   USBPUMP_CLASS_MSD_CLIENT * CONST pMsdClient = pClientHandle;

   USBPUMP_UNREFERENCED_PARAMETER(nNotification);

   switch (NotificationId)
       {
   case USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL:
       UsbPumpMsdClient_DeviceArrival(pMsdClient, pNotification);
       break;

   case USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE:
       UsbPumpMsdClient_DeviceDeparture(pMsdClient, pNotification);
       break;

   case USBPUMP_CLASSKIT_EVENT_FUNCTION_OPEN:
   case USBPUMP_CLASSKIT_EVENT_FUNCTION_CLOSE:
       break;

   default:
       break;
       }
   }

static VOID
UsbPumpMsdClient_DeviceArrival(
   USBPUMP_CLASS_MSD_CLIENT *pMsdClient,
   CONST USBPUMP_CLASSKIT_EVENT_DEVICE_ARRIVAL_INFO *pInfo
   )
   {
   USBPUMP_CLASS_MSD_CLIENT_DEVICE *    pMsdDevice;

   TTUSB_OBJPRINTF((
       &pMsdClient->ObjectHeader,
       UDMASK_ENTRY | UDMASK_ANY,
       "+" FUNCTION ":"
```

```
        " pFunction(%p)\n",
        pInfo->pFunction
        ));

    pMsdDevice = UsbPumpPlatform_MallocZero(
            pMsdClient->pPlatform,
            sizeof(*pMsdDevice)
            );
    if (pMsdDevice == NULL)
        {
        TTUSB_OBJPRINTF((
            &pMsdClient->ObjectHeader,
            UDMASK_ERRORS,
            "?UsbPumpMsdClient_ClassNotification:"
            " pMsdDevice allocation failed.\n"
            ));
        return;
        }

    pMsdDevice->pMsdClient = pMsdClient;
    pMsdDevice->pFunction = pInfo->pFunction;

    USBPUMPLIB_LIST_INSERT_NODE_TAIL(
        USBPUMP_CLASS_MSD_CLIENT_DEVICE,
        &pMsdClient->pMsdDeviceHead,
        pMsdDevice,
        pNext,
        pLast
        );

    (*pMsdClient->InCall.Msd.pOpenFunctionFn)(
        pMsdClient->SessionHandle,
        UsbPumpMsdClient_OpenCallback,
        pMsdDevice,
        pMsdDevice->pFunction,
        &pMsdDevice->InCall.ClassKitCast,
        sizeof(pMsdClient->InCall),
        pMsdDevice,
        &sk_UsbPumpUsbdiMsdClient_FunctionOutCall,
        sizeof(sk_UsbPumpUsbdiMsdClient_FunctionOutCall)
        );
    }
```

The `USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE` class notification means that a USB device bound to the MSD 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 `USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE` class notification is provided below:

```
static VOID
UsbPumpMsdClient_DeviceDeparture(
   USBPUMP_CLASS_MSD_CLIENT *pMsdClient,
   CONST USBPUMP_CLASSKIT_EVENT_DEVICE_DEPARTURE_INFO *pInfo
   )
   {
   USBPUMP_CLASS_MSD_CLIENT_DEVICE *    pMsdDevice;

   TTUSB_OBJPRINTF((
       &pMsdClient->ObjectHeader,
       UDMASK_ENTRY | UDMASK_ANY,
       "+" FUNCTION ":"
       " pFunction(%p)\n",
       pInfo->pFunction
       ));

   USBPUMPLIB_LIST_FOREACH_BEGIN(
       USBPUMP_CLASS_MSD_CLIENT_DEVICE,
       pMsdClient->pMsdDeviceHead,
       pMsdDevice
       )

       if (pMsdDevice->pFunction == pInfo->pFunction)
           {
           (*pMsdDevice->InCall.Msd.pCloseFn)(
               pMsdDevice->SessionHandle,
               UsbPumpMsdClient_CloseCallback,
               pMsdDevice
               );
           return;
           }

   USBPUMPLIB_LIST_FOREACH_END(
       pMsdDevice,
       pNext
       )
   }
```

The `USBPUMP_CLASSKIT_EVENT_FUNCTION_OPEN` and `FUNCTION_CLOSE` class notifications are sent when a Function Session is opened or closed for the MSD, respectively.

2.6 Enumerate Function Vector Bound to MSD

The client can obtain the number of function instances that are bound to the MSD by invoking GetNumDevices Class In-Call and enumerating a vector of them using GetBoundDevices 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 calling pGetDeviceInfoFn() to get attached mass storage device information and calling pGetLunInfoFn() to get mass storage device LUN information. For the details of MSD 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  ≒

        112 + /* MSD Class overhead */

        NumSession * 28 + /* MSD Session overhead */

        NumInstances * (

                544 + /* MSD Function overhead */

                64  + /* MSD config tree node */

                200 + /* MSD internal URB */

                256 + /* MSD MaxConfigDescSize */

                (MaxNumLUN * 56) + /* MSD LUN handling overhead */

                (2048 or 1024) /* MSD data buffer size, 2048=super-speed, 1024=others */

                );
```

4 MSD Interfaces

The MSD provides two types of interfaces, the MSD Class interface and the Function Interface. The MSD object has one MSD class interface. The MSD object can have multiple Function Interfaces one for each MSD function instance. To use the MSD class interface, a client must open a class session to the MSD object using `UsbPumpObject_OpenSession()`. For the details of `UsbPumpObject_OpenSession()`, refer to [MOB] in the referenced documents section. And to use the MSD function interface, the client must open a function session to a MSD 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 MSD 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/MSD, 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 MSD. 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 GetDriverFeature 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 Mass Storage Class Driver. For example, if the client wishes to know whether the MSD supports super-speed device, the client simply calls this operation with a feature name string such as "support super-speed". If the MSD supports this feature, it returns TRUE. Otherwise, FALSE is returned.

Declaration of Prototype

```
typedef BOOL
(*USBPUMP_USBDI_MSD_CLASS_GET_DRIVER_FEATURE_FN)(
   USBPUMP_SESSION_HANDLE   SessionHandle,
   CONST TEXT *             pFeatureName,
   SIZE_T                   sizeFeatureName
   );
```

Parameters

`SessionHandle` is the class session handle returned from `UsbPumpObject_OpenSession()`.

`pFeatureName` is the feature name string such as "support super-speed".

`sizeFeatureName` is the length of the feature name string.

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 MSD, 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 MSD 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 MSD 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 GetDeviceInfo Operation

Description

The client uses this operation to get the mass storage device information of the attached device. 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_CLASSKIT _STATUS_OK
USBPUMP_CLASSKIT_STATUS_INVALID_FUNCTION_HANDLE
USBPUMP_CLASSKIT_STATUS_INVALID_PARAMETER
```

Declaration of Prototype

```
typedef USBPUMP_CLASSKIT_STATUS
(*USBPUMP_USBDI_MSD_FUNCTION_GET_DEVICE_INFO_FN)(
   USBPUMP_SESSION_HANDLE   FunctionHandle,
   UINT *                   pNumberOfLun,
   UINT8 *                  pSerialNo,   -- Optional
   BYTES                    nSerialNo
   );
```

Parameters

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

`pNumberOfLun` is pointer of the number of LUN. The MSD function will return how many LUNs are available on the attached mass storage device.

`pSerialNo` is the mass storage device serial number save buffer. This optional input parameter and client can pass NULL.

`nSerialNo` is size of input pSerialNo buffer.

Callback Routine

None.

Sample Code

```
static VOID
UsbPumpMsdClient_OpenCallback(
   VOID *           pCallbackCtx,
   USBPUMP_CLASSKIT_STATUS  ErrorCode,
   USBPUMP_SESSION_HANDLE   SessionHandle
   )
   {
   USBPUMP_CLASS_MSD_CLIENT_DEVICE * CONST  pMsdDevice = pCallbackCtx;
   USBPUMP_CLASS_MSD_CLIENT_LUN *       pMsdLun;
   USBPUMP_CLASS_MSD_CLIENT * CONST
       pMsdClient = pMsdDevice->pMsdClient;
   UINTi;

   if (ErrorCode != USBPUMP_CLASSKIT_STATUS_OK)
       {
       TTUSB_OBJPRINTF((
           &pMsdClient->ObjectHeader,
           UDMASK_ERRORS,
           "?" FUNCTION ": ErrorCode(%d)\n",
           ErrorCode
           ));
       return;
       }

   pMsdDevice->SessionHandle = SessionHandle;

   TTUSB_OBJPRINTF((
       &pMsdClient->ObjectHeader,
       UDMASK_FLOW,
       " " FUNCTION ":"
       " SessionHandle:%p\n",
       SessionHandle
       ));

   (*pMsdDevice->InCall.Msd.pGetDeviceInfoFn)(
       pMsdDevice->SessionHandle,
       &pMsdDevice->nLun,
       NULL,   /* pSerialNo -- OPTIONAL */
```

```
            0   /* nSerialNo */
            );

        TTUSB_OBJPRINTF((
            &pMsdClient->ObjectHeader,
            UDMASK_FLOW,
            " " FUNCTION ":"
            " Number of LUNs = %u\n",
            pMsdDevice->nLun
            ));

    /* Allocate LUN handling structure */
    }
```

4.2.1.3 GetLunInfo Operation

Description

The client uses this operation to get the LUN information of the attached mass storage device. 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_CLASSKIT _STATUS_OK
USBPUMP_CLASSKIT_STATUS_INVALID_FUNCTION_HANDLE
USBPUMP_CLASSKIT_STATUS_INVALID_PARAMETER
```

Declaration of Prototype

```
    typedef USBPUMP_CLASSKIT_STATUS
    (*USBPUMP_USBDI_MSD_FUNCTION_GET_DEVICE_INFO_FN)(
        USBPUMP_SESSION_HANDLE       FunctionHandle,
        UINT                         Lun,
        USBPUMP_USBDI_MSD_LUN_INFO * pLunInfo
        );

    typedef struct _USBPUMP_USBDI_MSD_LUN_INFO
        {
        CHAR    VendorId[8+1];
        CHAR    ProductId[16+1];
        CHAR    ProductRevision[4+1];

        UINT8   DeviceType;
        UINT8   MediaType;
        BOOL    fRemovableMedia;
        } USBPUMP_USBDI_MSD_LUN_INFO;
```

Parameters

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

`Lun` is LUN index number.

`pLunInfo` is the output buffer to save LUN information. The `DeviceType` represents specified LUN device type and possible device types are UNKNOWN, DISK, CD, OPTICAL, FLASH. The MediaType represents current media type of the LUN and possible media types are NO_MEDIA, DISK, CD, DVD, FLASH_SD, FLASH_COMPACTF, FLASH_MEMSTICK, FLASH_SMARTMD, FLASH_XD, FLASH_UNKNOWN.

Callback Routine

None.

Sample Code

```
static VOID
UsbPumpMsdClient_OpenCallback(
   VOID *           pCallbackCtx,
   USBPUMP_CLASSKIT_STATUS  ErrorCode,
   USBPUMP_SESSION_HANDLE   SessionHandle
   )
   {
   /* find out how many LUNs are available */

   /* Allocate LUN handling structure */
   pMsdLun = UsbPumpPlatform_MallocZero(
           pMsdClient->pPlatform,
           sizeof(*pMsdLun) * pMsdDevice->nLun
           );
   if (pMsdLun == NULL)
       {
       TTUSB_OBJPRINTF((
           &pMsdClient->ObjectHeader,
           UDMASK_ERRORS,
           "?" FUNCTION ":"
           " %d MsdLun allocation failed.\n",
           pMsdDevice->nLun
           ));
       return;
       }

   pMsdDevice->pLun = pMsdLun;

   for (i = 0; i < pMsdDevice->nLun; ++pMsdLun, ++i)
       {
       pMsdLun->pMsdDevice = pMsdDevice;
```

```
            pMsdLun->Lun = i;

            (*pMsdDevice->InCall.Msd.pGetLunInfoFn)(
                pMsdDevice->SessionHandle,
                i,
                &pMsdLun->LunInfo
                );

            TTUSB_OBJPRINTF((
                &pMsdClient->ObjectHeader,
                UDMASK_FLOW,
                " " FUNCTION ":"
                " Lun(%x) Vendor(%s) Product(%s)\n",
                i,
                pMsdLun->LunInfo.VendorId,
                pMsdLun->LunInfo.ProductId
                ));
            }
        }
```

4.2.1.4 GetMediaInfo Operation

Description

The client uses this operation to get current media information of the specified LUN of the attached mass storage device. 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_CLASSKIT _STATUS_OK
USBPUMP_CLASSKIT_STATUS_INVALID_FUNCTION_HANDLE
USBPUMP_CLASSKIT_STATUS_INVALID_PARAMETER
```

Declaration of Prototype

```
typedef USBPUMP_CLASSKIT_STATUS
(*USBPUMP_USBDI_MSD_FUNCTION_GET_MEDIA_INFO_FN)(
   USBPUMP_SESSION_HANDLE           FunctionHandle,
   UINT                             Lun,
   USBPUMP_USBDI_MSD_MEDIA_INFO *   pMediaInfo
   );

typedef struct _USBPUMP_USBDI_MSD_MEDIA_INFO
   {
   BOOL         fHasMedia;
   BOOL         fWriteProtected;
   UINT32       SizeOfBlock;
```

```
    UINT32      TotalBlocks;
    } USBPUMP_USBDI_MSD_MEDIA_INFO;
```

Parameters

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

`Lun` is LUN index number.

`pMediaInfo` is the output buffer to save current media information. It has media present flag, write protected flag and media capacity information.

Callback Routine

None.

Sample Code

```
static VOID
UsbPumpMsdClient_OpenCallback(
   VOID *          pCallbackCtx,
   USBPUMP_CLASSKIT_STATUS  ErrorCode,
   USBPUMP_SESSION_HANDLE   SessionHandle
   )
   {
   /* Find out how many LUNs are available, allocate LUN handling structure */

   for (i = 0; i < pMsdDevice->nLun; ++pMsdLun, ++i)
       {
       pMsdLun->pMsdDevice = pMsdDevice;
       pMsdLun->Lun = i;

       (*pMsdDevice->InCall.Msd.pGetMediaInfoFn)(
           pMsdDevice->SessionHandle,
           i,
           &pMsdLun->MediaInfo
           );

       TTUSB_OBJPRINTF((
           &pMsdClient->ObjectHeader,
           UDMASK_FLOW,
           "  SizeOfBlock(%x) TotalBlocks(%x) fReadOnly(%x)\n",
           pMsdLun->MediaInfo.SizeOfBlock,
           pMsdLun->MediaInfo.TotalBlocks,
           pMsdLun->MediaInfo.fWriteProtected
           ));

       pMsdLun->nBuffer = pMsdClient->DataBufferSize;
       pMsdLun->pBuffer = UsbPumpPlatform_Malloc(
```

```
                    pMsdClient->pPlatform,
                    pMsdLun->nBuffer
                    );
            if (pMsdLun->pBuffer == NULL)
                {
                TTUSB_OBJPRINTF((
                    &pMsdClient->ObjectHeader,
                    UDMASK_ERRORS,
                    "?" FUNCTION ":"
                    " MsdLun %d bytes buffer allocation failed.\n",
                    pMsdLun->nBuffer
                    ));
                continue;
                }

            if (pMsdLun->MediaInfo.fHasMedia)
                {
                UsbPumpMsdClient_StartRead(pMsdDevice, pMsdLun);
                }
            }
        }
```

4.2.1.5 SubmitRequest Operation

Description

The client uses this operation to submit MSD request to the mass storage function driver. The USBPUMP_USBDI_CLASS_MSD_REQUEST::pDoneFn function will be called when this request is completed. For the details of the request, refer to the sections further below.

This API has no return value and the USBPUMP_USBDI_CLASS_MSD_REQUEST::pDoneFn will be called later.

Declaration of Prototype

```
typedef VOID
(*USBPUMP_USBDI_MSD_FUNCTION_SUBMIT_REQUEST_FN)(
   USBPUMP_SESSION_HANDLE               FunctionHandle,
   USBPUMP_USBDI_CLASS_MSD_REQUEST *    pMsdRequest
   );

typedef struct _ USBPUMP_USBDI_CLASS_MSD_REQUEST
   {
   USBPUMP_USBDI_CLASS_MSD_REQUEST_CODE Request;
   USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT Status;

   UINT8                                Lun;
```

```
    UINT32                                  StartLBA;
    UINT16                                  NumberOfLBA;

    USBPUMP_TIMER_TIMEOUT                   Timeout;

    VOID *                                  pBuffer;
    USBPUMP_BUFFER_HANDLE                   hBuffer;
    BYTES                                   nBuffer;
    BYTES                                   nActual;

    USBPUMP_USBDI_CLASS_MSD_REQUEST_DONE_FN *pDoneFn;
    VOID *                                  pDoneInfo;

    USBPUMP_USBDI_CLASS_MSD_REQUEST *    pNext;
    USBPUMP_USBDI_CLASS_MSD_REQUEST *    pLast;

    /* Internal usage */
    USBPUMP_USBDI_CLASS_MSD_REQUEST_PRIVATE  Private;
    } USBPUMP_USBDI_CLASS_MSD_REQUEST;
```

Parameters

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

`pMsdRequest` is the MSD request block. Clients need to prepare MSD request block using UsbPumpUsbdMsd_PrepareRequest() macro function and submit MSD request. Refer to section 4.2.1.5.1 below in this documentation for the detailed description of the MSD request block.

Callback Routine

None.

4.2.1.5.1 USBPUMP_USBDI_CLASS_MSD_REQUEST

This structure encapsulates a request to the mass storage class driver. The portable MSD class API uses special request packets to convey requests to the MSD class driver.

Declaration of structure

```
typedef struct _ USBPUMP_USBDI_CLASS_MSD_REQUEST
   {
   USBPUMP_USBDI_CLASS_MSD_REQUEST_CODE Request;
   USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT Status;

   UINT8                                Lun;

   UINT32                               StartLBA;
   UINT16                               NumberOfLBA;
```

```
USBPUMP_TIMER_TIMEOUT               Timeout;

VOID *                              pBuffer;
USBPUMP_BUFFER_HANDLE               hBuffer;
BYTES                               nBuffer;
BYTES                               nActual;

USBPUMP_USBDI_CLASS_MSD_REQUEST_DONE_FN *pDoneFn;
VOID *                              pDoneInfo;

USBPUMP_USBDI_CLASS_MSD_REQUEST *   pNext;
USBPUMP_USBDI_CLASS_MSD_REQUEST *   pLast;

/* Internal usage */
USBPUMP_USBDI_CLASS_MSD_REQUEST_PRIVATE  Private;
} USBPUMP_USBDI_CLASS_MSD_REQUEST;
```

Contents:

```
USBPUMP_USBDI_CLASS_MSD_REQUEST_CODE Request;
    Request code of this MSD request. Defined request codes are

    USBPUMP_USBDI_CLASS_MSD_REQUEST_READ
        Data read request from given LUN.

    USBPUMP_USBDI_CLASS_MSD_REQUEST_WRITE
        Data write request to given LUN.

USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT Status;
    On completion, set to the completion code for this request.
    The common MSD entry logic sets this to USBPUMP_USBDI_CLASS_
    MSD_REQUEST_STATUS_BUSY when the request is accepted, and to
    USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT_XXX when the request
    is done.

UINT8   Lun;
    Logical unit number.

USBPUMP_TIMER_TIMEOUT   Timeout;
    The timeout, expressed in milliseconds.  A timeout value
    of 0 means no timeout.
    |USBPUMP_USBDI_CLASS_MSD_TIMEOUT_MAX| is defined as the maximum
    value of Timeout, and |USBPUMP_USBDI_CLASS_MSD_TIMEOUT_NONE|
    is defined to be zero.

VOID *          pBuffer;
USBPUMP_BUFFER_HANDLE   hBuffer;
BYTES           nBuffer;
```

```
BYTES           nActual;
    Data buffer information. The pBuffer is pointer of data buffer,
    the nBuffer is data buffer size in bytes. The hBuffer is handle
    of the data buffer. The nActual is actual data transfer size
    in bytes and MSD will update this value when complete.


USBPUMP_USBDI_CLASS_MSD_REQUEST *pNext, *pLast;
    Fields reserved for use by the transient owner of the
    request packet.  This is the client prior to queuing
    and as soon as the packet is completed; and the
    MSD or its delegates from the time the packet is queued
    until the moment the completion routine is called.

USBPUMP_USBDI_CLASS_MSD_REQUEST_DONE_FN pDoneFn;
    The function to be called when the request is completed.
    The function is of type:

        VOID (*pDoneFn)(
            USBPUMP_USBDI_FUNCTION *pFunction,
            USBPUMP_USBDI_CLASS_MSD_REQUEST *pRequest,
            VOID *pDoneInfo,
            USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT Status
            );

    The parameters are provided for convenience; on typical
    RISC processors, the parameters will be passed in
    registers, saving the code that would otherwise be
    required to fetch the result.

VOID *pDoneInfo;
    Completion information.
```

Private Contents:

```
UCALLBACKCOMPLETION  Private.Callback.Event;
    DataPump callback event.

VOID *   Private.Callback.pFunction;
    Save function pointer.

UINT8    Private.Command.Data;
    The data direction of the request. Defined data directions are

    USBPUMP_USBDI_CLASS_MSD_REQUEST_DATA_NONE
        The request has no data phase.

    USBPUMP_USBDI_CLASS_MSD_REQUEST_DATA_IN
        The request has data IN phase.
```

```
            USBPUMP_USBDI_CLASS_MSD_REQUEST_DATA_OUT
                The request has data OUT phase.

        UINT8   Private.Command.bBlockLength;
            Length of the command block in bytes.

        UINT8   Private.Command.Block[16];
            Command block buffer.
```

API Function:

```
#defineUsbPumpUsbdMsd_PrepareRequest(
        pRequest,
        RequestCode,
        Lun,
        StartLBA,
        NumberOfLBA,
        Timeout,
        pBuffer,
        hBuffer,
        nBuffer,
        pDoneFn,
        pDoneInfo
        )
```

Parameters:

`pRequest` is the MSD request structure pointer to initialize.
RequestCode is the MSD request code. Currently supported request codes are
USBPUMP_USBDI_CLASS_MSD_REQUEST_READ
USBPUMP_USBDI_CLASS_MSD_REQUEST_WRITE
`Lun` is LUN index number and it is zero-based.
`StartLBA` is starting logical block address of the specified LUN.
`NumberOfLBA` is number of LBA to read or write operation.
`Timeout` is timeout value for this request in mili-seconds.
`pBuffer` is address of the data buffer.
`hBuffer` is handle of the data buffer. This is optional.
`nBuffer` is size of the data buffer.
`pDoneFn` is completion callback function pointer. This function will be called when data transfer request is done.
`pDoneInfo` is completion callback function context pointer.

Sample code:

```
static VOID
UsbPumpMsdClient_StartRead(
   USBPUMP_CLASS_MSD_CLIENT_DEVICE *    pMsdDevice,
```

```
USBPUMP_CLASS_MSD_CLIENT_LUN *       pMsdLun
)
{
UINT32  StartLBA;
UINT16  NumberOfLBA;

if ((StartLBA = pMsdLun->StartLBA) == 0)
    {
    NumberOfLBA = 1;
    pMsdLun->StartLBA += 32;
    }
else
    {
    NumberOfLBA = pMsdLun->nBuffer / pMsdLun->MediaInfo.SizeOfBlock;
    pMsdLun->StartLBA += 256;
    if (pMsdLun->StartLBA > pMsdLun->MediaInfo.TotalBlocks)
        pMsdLun->StartLBA = 256;
    }

UsbPumpUsbdMsd_PrepareRequest(              \
    &pMsdLun->Request,                      \
    USBPUMP_USBDI_CLASS_MSD_REQUEST_READ,   \
    pMsdLun->Lun,                           \
    StartLBA,                               \
    NumberOfLBA,                            \
    5000,   /* Timeout: 5 sec */            \
    pMsdLun->pBuffer,                       \
    NULL,   /* hBuffer */                   \
    NumberOfLBA * pMsdLun->MediaInfo.SizeOfBlock,\
    UsbPumpMsdClient_ReadDone,              \
    pMsdLun                                 \
    );

(*pMsdDevice->InCall.Msd.pSubmitRequestFn)(
    pMsdDevice->SessionHandle,
    &pMsdLun->Request
    );
}
```

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 MSD, 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 Mass Storage Class Driver API

5.1 MSD Configuration API

5.1.1 USBPUMP_USBDI_CLASS_MSD_CONFIG_INIT_V1

Description

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

Declaration of Prototype

```
#define USBPUMP_USBDI_CLASS_MSD_CONFIG_INIT_V1(
        MaxSession
        )
```

Parameters

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

Example

```
/*
|| Configuration for generic device class driver
*/
CONST USBPUMP_USBDI_CLASS_MSD_CONFIG gk_UsbPumpUsbdiClassMsd_ConfigDeafult =
   USBPUMP_USBDI_CLASS_MSD_CONFIG_INIT_V1(
        /* number of client sessions */ 4
        );
```

5.1.2 USBPUMP_USBDI_CLASS_MSD_CONFIG_SETUP_V1

Description

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

Declaration of Prototype

```
#define   USBPUMP_USBDI_CLASS_MSD_CONFIG_SETUP_V1(
          pConfig,
          MaxSession,
          )
```

Parameters

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

`MaxSession` – Refer to parameter in section 5.1.1.

Example

```
USBPUMP_USBDI_CLASS_MSD_CONFIG   MsdConfig;

USBPUMP_USBDI_CLASS_MSD_CONFIG_SETUP_V1(
    &MsdConfig,
    /* number of client sessions */ 4
    );
```

5.2 MSD API Functions

5.2.1 UsbPumpUsbdiClassMsd_Initialize

Description

This API function initializes the mass storage class driver. This function creates the mass storage class driver, along with all the idle instance objects for the mass storage 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.)

6 Mass Storage Class Driver Event Notifications

A client of MSD 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 MSD 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 the MSD 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 MSD Function In-Calls.
USBPUMP_CLASSKIT_EVENT_	A USB device for the MSD is detached from this host. The device can not be

Event Code	Description
DEVICE_DEPARTURE	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.
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_USBDI_MSD_EVENT_MEDIA_CHANGE	The mass storage device media changed notification event. When the MSD detects attached USB mass storage device's media changes, it will send this event notification to the client.

7 Mass Storage Class Driver Status Codes

7.1 Mass Storage Class Driver Status Codes

Following MSD status codes are returned by the MSD Class and Function In-Calls. The prefix of the status codes in below table is "USBPUMP_CLASSKIT_STATUS_".

Table 4 Mass Storage Class Driver Status Codes

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

Status Code	Description
NOT_SUPPORTED	This status code is returned when a client invokes an unsupported In-Call.
NO_MORE_SESSIONS	The MSD 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 MSD 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 MSD 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 MSD 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 MSD returns this status code when a client calls a Function In-Call using a function session that is already closed or never opened.

7.2 Mass Storage Request Status Codes

Following MSD status codes are returned by the MSD SubmitRequest operation. The prefix of the status codes in below table is "USBPUMP_USBDI_CLASS_MSD_REQUEST_STAT_".

Table 4 Mass Storage Class Driver Status Codes

Status Code	Description
OK	The MSD returns this status code when it handles a MSD request successfully.
BUSY	This status code is set internally when the MSD received a MSD request from client. If this status code is set in the MSD request block, the MSD is in progress this MSD request block.
ERROR	This status code is returned if error happen.
NO_MEDIA	This status code is returned if there is no media.
NO_DEVICE	This status code is returned if there is no USB mass storage device.

Status Code	Description
PROTECTED	This status code is returned if specified mass storage device is write protected and MSD request is write operation.
INVALID	The status code is returned if a MSD request is invalid format.