Application Note

AN_403

MCCI USB DataPump Mass Storage Protocol Users Guide

Version 1.0

Issue Date: 2017-09-13

This user guide introduces MCCI's portable, generic implementation of the USB Device Working Group Mass Storage Bulk-Only Transport and ATAPI protocols.

Bridgetek Pte Ltd (BRTChip)
178 Paya Lebar Road, #07-03, Singapore 409030
Tel: +65 6547 4827 Fax: +65 6841 6071
Web Site: http://www.brtchip.com

MCCI Legal and Copyright Information

Copyright:

Trademark Information:

The following are registered trademarks of MCCI Corporation:

- MCCI
- TrueCard
- TrueTask
- MCCI USB DataPump
- MCCI Catena

The following are trademarks of MCCI Corporation:

- MCCI Skimmer
- MCCI Wombat
- InstallRight
- MCCI ExpressDisk

All other trademarks, brands and names are the property of their respective owners.

Disclaimer of Warranty:

MCCI Corporation ("MCCI") provides this material as a service to its customers. The material may be used for informational purposes only.

MCCI assumes no responsibility for errors or omissions in the information contained at the world wide web site located at URL address 'http://www.mcci.com/', links reachable from this site, or other information stored on the servers 'www.mcci.com', 'forums.mcci.com', or 'news.mcci.com' (collectively referred to as "Web Site"). MCCI further does not warrant the accuracy or completeness of the information published in the Web Site. MCCI shall not be liable for any special, indirect, incidental, or consequential damages, including without limitation, lost revenues or lost profits, which may result from the use of these materials. MCCI may make changes to this Web Site, or to the products described therein, at any time without notice. MCCI makes no commitment to maintain or update the information at the Web Site.

THESE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Table of Contents

1 Introduction 5
1.1 Overview 5
1.2 Initialization and Setup 5
1.2.1 Protocol Library Initialization 5
1.2.2 Client Instance Initialization 6
2 Data Structures 8
2.1 USBPUMP_PROTOCOL_INIT_NODE 8
2.2 UPROTO_MSCSUBCLASS_ATAPI_CONFIG..........................10
2.3 UPROTO_MSCSUBCLASS_ATAPI_LUN_CONFIG 11
3 Edge-IOCTL (Upcall) services 12
3.1 Edge IOCTL Function 12
3.2 Generic Edge IOCTLS..12
3.2.1 Edge Activate 12
3.2.2 Edge Deactivate.. 13
3.2.3 Edge Bus Event 13
3.2.4 Edge Get Microsoft OS String Descriptor 14
3.2.5 Edge Get Function Section 14
3.3 Storage Specific Edge IOCTLs..15
3.3.1 Edge Storage Read 16
3.3.2 Edge Storage Read Done 16
3.3.3 Edge Storage Write 17
3.3.4 Edge Storage Write Data 17
3.3.5 Edge Storage Get Status.. 18
3.3.6 Edge Storage Reset Device 18
3.3.7 Edge Storage Load or Eject 18
3.3.8 Edge Storage Load or Eject Ex.. 19
3.3.9 Edge Storage Prevent Removal.. 19
3.3.10 Edge Storage Client Command 20
3.3.11 Edge Storage Client Send Done 20
3.3.12 Edge Storage Client Receive Done.. 21

3.3.13 Edge Storage Remove Tag 21
3.3.14 Edge Storage Custom Command 21
3.3.15 Edge Storage Custom Send Done 22
3.3.16 Edge Storage Custom Receive Done 23

4 Downcall Services 24

4.1 Storage Queue Read 24
4.2 Storage Queue Write 24
4.3 Storage Write-Done 25
4.4 Storage Set Current Medium 25
4.5 Storage Set Device Properties 26
4.6 Storage Queue Read V2 26
4.7 Storage Queue Write V2 27
4.8 Storage Write-Done V2 27
4.9 Storage Set Current Medium V2 28
4.10 Storage Queue Read V3 29
4.11 Storage Queue Write V3 29
4.12 Storage Write-Done V3 30
4.13 Storage Set Current Medium V3 30
4.14 Storage Set Device Properties V2 31
4.15 Storage Client Set Mode 32
4.16 Storage Client Sent Data 32
4.17 Storage Client Receive Data 33
4.18 Storage Client Sent Status 34
4.19 Storage Client Get Inquiry Data 34
4.20 Storage Custom Send Status 35
4.20.1 An Example of Supporting Custom SCSI Commands 35
4.21 Storage Custom Send Data 37
4.22 Storage Custom Receive Data 38
4.23 Storage Control Last Lun 38

5 Other Considerations 39

6 Performance Considerations.................................. 40
6.1 Write 40
6.2 Read 40
6.3 General...40
7 Demo Applications... 41
8 Contact Information 45
Appendix A – References 46
Document References 46
Acronyms and Abbreviations...46
Appendix B – List of Tables & Figures 48
List of Tables...48
List of Figures 48
Appendix C – Revision History 49

1 Introduction

The MCCI USB DataPump product is a portable firmware framework for developing USB-enabled devices. As part of the DataPump, MCCI provides a portable, generic implementation of the USB Device Working Group Mass Storage Bulk-Only Transport and ATAPI protocols. We present programming information for integrating this support into user's firmware, to create a USB device that presents a mass-storage class interface to the host PC.

This document does not discuss about host software issues. Because the MCCI implementation complies with the MSC BOT standard, most operating system host drivers will work directly with MCCI's implementation. For information on Microsoft Windows support for MSC, please refer to Microsoft USB Storage FAQ [WINUSBFAQ].

1.1 Overview

The MCCI MSC Protocol Library, in conjunction with the MCCI USB DataPump, provides a straightforward, portable environment for implementing ATAPI compliant mass storage devices over USB using the USB Mass Storage BOT 1.0 protocol. The MCCI MSC Protocol Library can be used to create a stand-alone device, or can be combined with other MCCI- and/or user-provided protocols to create multi-function devices.

This document describes the portions of the MCCI MSC Protocol Library that are visible to an external client. As such, it serves as a Library User's Guide. It is not intended to serve as a stand-alone reference, but should be used in conjunction with the MCCI DataPump User's Guide and the USB MSC BOT Specification [USBMSCBOT], and the relevant ATAPI documentation (see [ATAPI]). The purpose of the MSC Protocol Library is to encapsulate issues regarding USB transactions so that the user can concentrate on the mass-storage portions of a target device.

1.2 Initialization and Setup

When using the DataPump Mass Storage Protocol, the final application consists of two distinct parts. The first part is provided by MCCI and consists of the MCCI USB DataPump libraries and specifically, the MCCI USB MSC Protocol Library. This document uses the name **Protocol** to refer collectively to these components. The second part is provided by the developer and consists of application and device specific modules. This document uses the name **Client** to refer to these components.

1.2.1 Protocol Library Initialization

The Protocol Library code parses the device descriptors, and creates Protocol Instances for each supported Mass Storage Class function. The Protocol Mass Storage Class functions are represented by an interface descriptor with bInterfaceClass 0x08, bInterfaceProtocol 0x50, and bSubClass0x06. These codes indicate to the library:

- that the interface represents a Mass Storage Class device (bInterfaceClass 0x08),
- that the command set for the interface is transported using Bulk Only Transport (bInterfaceProtocol 0x50), and
- that the device is to use the SFF-8020i or MMC-2 command set (as specified by the [SFF-8020i] or [MMC-2] specification).

Each such interface must also supply two bulk endpoint, an IN endpoint and an OUT endpoint. The Protocol Library is not sensitive to the order of the endpoints in the descriptor set, nor to the wMaxPacketSize of the endpoints.

The protocol library assumes that MMC-2 commands are desired. The host will determine this automatically based on the responses generated to "Inquiry" commands.

The following fragment of USBRC code shows how this might be coded:

```
interface 0
        {
        class    0x08           #mass storage class
        subclass 0x06           #ATAPI/SCSI  commands
        protocol 0x50           #bulk-only transport
        name     S_MSCDEV1      #string reference

        endpoints
                    bulk in

                          # Endpoint Companion Descriptor
                          max-burst 15
                          max-streams 0
                          max-sequence 1

                          bulk out

                          # Endpoint Companion Descriptor
                          max-burst 15
                          max-streams 0
                          max-sequence 1                ;
         }
```

The protocol library will create one Protocol Instance for each supported mass-storage interface that it finds in the descriptor set. If a mass storage class interface appears in multiple configurations, then the protocol library will create multiple instances, one for each configuration.

The Protocol Instance code performs all command set decoding, however it contains no code that actually knows how to read and write data blocks. It also requires assistance for obtaining media geometry and other information pertaining to the physical medium. For this purpose, the system integrator must provide client code. This is discussed in the next section.

Finally, the USB DataPump must be instructed to include Mass Storage support in the code being built. This is done using the application initialization vector. See Section 2.1 below.

1.2.2 Client Instance Initialization

Client's code dynamically locates Protocol instances using the USB DataPump object dictionary. When the DataPump is initialized, the modules will create protocol instances, and will give those names.

After the DataPump initializes, the target operating system must discover the available mass-storage instances, and must create client instances. Each client instance registers with a protocol instance. All communications from Client to Protocol is accomplished using a down I/O-control mechanism, known as an **IOCTL**, defined by the DataPump and implemented by the Protocol (See

Section 4). When a function in the Client needs to access a service in the Protocol, then a call is made to the IOCTL mechanism supplied with the appropriate service code.

Because USB device firmware is controlled by the host PC, there is a need for asynchronous communication from the Protocol Instance to the Client Instance. Communications from Protocol to Client are accomplished using an upcall IO-control mechanism, known as an **Edge-IOCTL**. The IOCTLs are defined by the DataPump and are routed by the DataPump to a function supplied by the Client during the initialization process. When a function in the Protocol needs to access a service in the Client, then a call is made to the Edge-IOCTL mechanism supplied with the appropriate service code.

During initialization, the Client will receive control from the platform startup code. The Client is then responsible for enumerating and initializing all instances of the Protocol by repeatedly calling

```
UsbPumpObject_EnumerateMatchingNames(
        ...,
        "storage.*.fn.mcci.com",
        ...)
```

Each time the function returns a non-NULL pointer to a Protocol USBPUMP_OBJECT_HEADER, the Client code must

- Create a matching client instance, with an accompanying USBPUMP_OBJECT_HEADER to represent the Client Instance to the DataPump
- Call UsbPumpObject_Init() to initialize the Client Instance USBPUMP_OBJECT_HEADER and bind it to the Edge-IOCTL function provided by the Client.
- Call UsbPumpObject_FunctionOpen() to open the Protocol object and bind it to the Client Instance object. The USBPUMP_OBJECT_HEADER pointer returned by the call is the reference that the Client Instance will use to access the Protocol Instance thru the IOCTL mechanism.

Applications wishing to make use of the Protocol library should

- include the header file usbmsc10.h, ufnapistorage.h and usbioctl_storage.h
- link with library protomsc.

2 Data Structures

Several data structures are involved in initializing and running the Protocol. The ones that are of interest for the Client are listed below.

2.1 USBPUMP_PROTOCOL_INIT_NODE

This structure is part of the USB_DATAPUMP_APPLICATION_INIT_VECTOR_HDR that the Client passes to the DataPump init function. It is preferably initialized using USBPUMP_PROTOCOL_INIT_NODE_INIT_V2 since this provides backward compatibility with future releases of the DataPump.

This structure is used by the enumerator to match the Protocol against the device, configuration and interface descriptors when locating interfaces to use for the Protocol, and to bind init functions to the Protocol. The fields of interest to the Client are:

`sDeviceClass:`	Normally -1 → allows matching to any device class.
`sDeviceSubClass:`	Normally -1 → allows matching to any device subclass
`sDeviceProtocol:`	Normally -1 → allows matching to any device protocol
`sInterfaceClass:`	USB_bInterfaceClass_MassStorage
`sInterfaceSubClass:`	USB_bInterfaceSubClass_MassStorageATAPI
`sInterfaceProtocol:`	Normally -1 → allows matching no matter what bInterfaceProtocol is used
`sConfigurationValue:`	Normally -1 → allows matching no matter what bConfigurationValue was used in the configuration descriptor
`sInterfaceNumber:`	Normally -1 → allows matching no matter what bInterfaceNumber is on the interface.
`sAlternateSetting:`	Normally -1 → allows matching no matter what bAlternateSetting is on the interface
`sSpeed:`	Always -1 (Reserved for future use)
`uProbeFlags`	Field for probe-control flags
`pProbeFunction:`	Optional pointer to USBPUMP_PROTOCOL_PROBE_FN function. If this function is available and returns FALSE then the pCreateFunction function will not be called prohibiting the creation of the protocol instance. Prototype: __TMS_FNTYPE_DEF (USBPUMP_PROTOCOL_PROBE_FN, __TMS_BOOL, (

```
  __TMS_UDEVICE *,
  __TMS_UINTERFACE *,
  __TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE *,
  __TMS_USBPUMP_OBJECT_HEADER *
  ));
```

Header File: usbprotoinit.h

Functions which are to be used as "probe" functions should be prototyped using this type, by writing:

USBPUMP_PROTOCOL_PROBE_FN MyProbeFunction;

The parameters are:

__TMS_UDEVICE * Pointer to the governing UDEVICE

__TMS_UINTERFACE * It is a pointer to the UINTERFACE under consideration

__TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE * Points to the USBPUMP_PROTOCOL_INIT_NODE in question

__TMS_USBPUMP_OBJECT_HEADER * It is the value returned previously by the USBPUMP_PROTOCOL_INIT_NODE_VECTOR's "setup" function. If no SETUP function was provided, then pProtoInitContext will be NULL.

`pCreateFunction:`

Normally MscSubClass_Atapi_ProtocolCreate – this function will create the appropriate set of protocol objects to implement the appropriate class-level behavior.

Where MscSubClass_Atapi_ProtocolCreate is defined as

```
__TMS_USBPUMP_PROTOCOL_CREATE_FN
    MscSubClass_Atapi_ProtocolCreate;
```

Prototype:

```
__TMS_FNTYPE_DEF (USBPUMP_PROTOCOL_CREATE_FN,
      __TMS_USBPUMP_OBJECT_HEADER *, (
      __TMS_UDEVICE *,
      __TMS_UINTERFACE *,
      __TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE
*,
      __TMS_USBPUMP_OBJECT_HEADER *
      ));
```

Header File: usbprotoinit.h

Each USBPUMP_PROTOCOL_INIT_NODE instance must supply a "create" function pointer. This function is called for each matching UINTERFACE, and is expected to attach a protocol to the underlying UINTERFACE or UINTERFACESET.

Functions which are to be used as "create" functions should be prototyped using this type, by writing:

USBPUMP_PROTOCOL_CREATE_FN MyCreateFunction;

The parameters are:

__TMS_UDEVICE * Pointer to the governing UDEVICE

__TMS_UINTERFACE * Points to the UINTERFACE under consideration

__TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE * Points to the USBPUMP_PROTOCOL_INIT_NODE in question

__TMS_USBPUMP_OBJECT_HEADER * It is the value returned

	previously by the USBPUMP_PROTOCOL_INIT_NODE_VECTOR's "setup" function. If no SETUP function was provided, then pProtoInitContext will be NULL.

`pQualifyAddInterfaceFunction`	Optional add-instance qualifier function. If this function is available and returns TRUE then pAddInterfaceFunction will be called to add the interface. Where, pAddInterfaceFunction is defined as
	__TMS_USBPUMP_PROTOCOL_ADD_INTERFACE_FN
	    *pAddInterfaceFunction;
	Prototype:
	__TMS_FNTYPE_DEF (USBPUMP_PROTOCOL_ADD_INTERFACE_FN,
	    __TMS_BOOL, (
	    __TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE *,
	    __TMS_USBPUMP_OBJECT_HEADER *,
	    __TMS_UDATAPLANE *,
	    __TMS_UINTERFACE *
	    ));
	Header File: usbprotoinit.h
	Functions which are to be used as "add interface" functions should be prototyped using this type, by writing:
	USBPUMP_PROTOCOL_ADD_INTERFACE_FN
	    MyAddInstanceFunction;
	The parameters are:
	__TMS_CONST __TMS_USBPUMP_PROTOCOL_INIT_NODE * It is the pointer to the governing USBPUMP_PROTOCOL_INIT_NODE.
	__TMS_USBPUMP_OBJECT_HEADER * It is the value returned previously by the USBPUMP_PROTOCOL_INIT_NODE_VECTOR's "setup" function. If no SETUP function was provided, then pProtoInitContext will be NULL.
	__TMS_UDATAPLANE * Points to the governing UDATAPLANE
	__TMS_UINTERFACE * Points to the UINTERFACE that is to be added to the protocol instance.

`pAddInterfaceFunction`	Optional function for adding instance
`pOptionalInfo:`	Pointer to UPROTO_MSCSUBCLASS_ATAPI_CONFIG structure (see Section 2.2)

2.2 UPROTO_MSCSUBCLASS_ATAPI_CONFIG

This structure is pointed to by the USBPUMP_PROTOCOL_INIT_NODE. It is preferably initialized using the macro UPROTO_MSCSUBCLASS_ATAPI_CONFIG_INIT_V3 since this provides backward compatibility with future releases of the Protocol.

This structure is used to configure the Protocol. The fields of interest to the Client are:

`pLun`	Pointer to array of LUN configuration structure(UPROTO_MSCSUBCLASS_ATAPI_LUN_CONFIG).

`fEnableDataInStuff`	Flag to Indicating whether data need to be stuffed

Note: Macro UPROTO_MSCSUBCLASS_ATAPI_CONFIG_INIT_V1 is obsolete and should not be used.

2.3 UPROTO_MSCSUBCLASS_ATAPI_LUN_CONFIG

An array if this structure is pointed to by the UPROTO_MSCSUBCLASS_ATAPI_CONFIG. It is preferably initialized using the macro UPROTO_MSCSUBCLASS_ATAPI_LUN_CONFIG_INIT_V1 since this provides backward compatibility with future releases of the Protocol.

This structure is used to configure the Protocol. The fields of interest to the Client are:

`DeviceType:`	USBPUMP_STORAGE_DEVICE_TYPE indicating ATAPI peripheral device type.
`fRemovable:`	Indicating if this device has removable medium or not
`pVendorId:`	Pointer to vendor id string. This is an ANSI string that is used for ATAPI-level Vendor-ID queries, and is not necessarily related to the USB vendor ID.
`pProductId:`	Pointer to product id string. This is an ANSI string that is used for ATAPI-level Product ID queries, and is not necessarily related to the USB product ID.
`pVersion:`	Pointer to version string. This is an ANSI string that is used for ATAPI-level version-number queries, and is not necessarily related to the USB product version number.

3 Edge-IOCTL (Upcall) services

The following section describes the services the Client must provide to the Protocol thru the Edge-IOCTL function given when initializing the Client object using UsbPumpObject_Init() (see Appendix A – Acronyms & Abbreviations).

3.1 Edge IOCTL Function

```
Type name :     USBPUMP_OBJECT_IOCTL_FN

Prototype :     USBPUMP_IOCTL_RESULT OsNone_Ft900_Platform_Ioctl (
                    USBPUMP_OBJECT_HEADER *p,     /* Pointer to target obj */
                    USBPUMP_IOCTL_CODE,           /* IOCTL-code */
                    CONST VOID *,                 /* Pointer to in parameter
*/
                    VOID *                        /* Pointer to out parameter
*/
                    );

Header-file : osnone_ft900_datapump.h
```

3.2 Generic Edge IOCTLS

3.2.1 Edge Activate

IOCTL code	USBPUMP_IOCTL_EDGE_ACTIVATE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_ACTIVATE_ARG *
Field pObject	Pointer to lower-level UPROTO object header
Field pClientContext	Context handle supplied by client when it is connected to the lower-level UPROTO object
Out parameter	USBPUMP_IOCTL_EDGE_ACTIVATE_ARG *
Field fReject	If set TRUE, then the Client would like the Protocol to reject the request, if possible. Note that fReject is an advisory indication, which may be used to flag to the Protocol that the Client cannot actually operate the data streams at this time. Because of hardware or protocol limitations, this might or might not be honored by the lower layers.

	Field is initialized to FALSE by Protocol.
Description	This IOCTL is sent from Protocol to Client whenever the host does something that brings up the logical function. Note that this may be sent when there are no data-channels ready yet. This merely means that the control interface of the function has been configured and is ready to transfer data.
Note	The out parameter is initialized by the Protocol with the same values as the in parameter

3.2.2 Edge Deactivate

IOCTL code	USBPUMP_IOCTL_EDGE_DEACTIVATE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_DEACTIVATE_ARG *
Field pObject	Pointer to lower-level UPROTO object header
Field pClientContext	Context handle supplied by client when it is connected to the lower-level UPROTO object
Out parameter	NULL
Description	The Protocol issues this IOCTL whenever a (protocol-specific) event occurs that deactivates the function. Unlike the ACTIVATE call, the Client has no way to attempt to reject this call. The USB host might have issued a reset -- there's no way to prevent, in general, deactivation.

3.2.3 Edge Bus Event

IOCTL code	USBPUMP_IOCTL_EDGE_BUS_EVENT
In parameter structure	CONST USBPUMP_IOCTL_EDGE_BUS_EVENT_ARG *
Field pObject	Pointer to lower-level UPROTO object header
Field pClientContext	Context handle supplied by client when it is connected to the lower-level UPROTO object
Field EventCode	Instance of UEVENT. The type of event that occurred. This will be one of UEVENT_SUSPEND, UEVENT_RESUME, UEVENT_ATTACH, UEVENT_DETACH, or UEVENT_RESET. [UEVENT_RESET is actually redundant; it will also cause a deactivate event; however this hook may be useful for apps

	that wish to model the USB state.]
Field pEventSpecificInfo	The event-specific information accompanying the UEVENT. Pointer to an Client specific event info. See "ueventnode.h" for details.
Field fRemoteWakeupEnable	Set TRUE if remote-wakeup is enabled.
Out parameter	NULL
Description	Whenever a significant bus event occurs, the Protocol will arrange for this IOCTL to be made to the Client (OS-specific driver). Any events that actually change the state of the Protocol will also cause the appropriate Edge-IOCTL to be performed; SUSPEND and RESUME don't actually change the state of the Protocol (according to the USB core spec).

3.2.4 Edge Get Microsoft OS String Descriptor

IOCTL code	USBPUMP_IOCTL_EDGE_GET_MS_OS_DESC_INFO
In parameter structure	CONST USBPUMP_IOCTL_EDGE_GET_MS_OS_DESC_INFO_ARG *
Field pConfig	pointer of UCONFIG. This is current active configuration. The protocol instance should check this UCONFIG structure to figure out that protocol is part of active configuration. If the protocol object is part of current active configuration, it should return function section of the extended compact ID feature descriptor.
Out parameter	USBPUMP_IOCTL_EDGE_GET_MS_OS_DESC_INFO_ARG *
Field fSupportOsDesc	TRUE if protocol object supports Microsoft OS string descriptor feature.
Description	This IOCTL is sent from DataPump core to the UPROTO/UFUNCTION object. DataPump core send this IOCTL to get information of Microsoft OS string descriptor. This edge IOCTL will be sent only if client enables this feature using USBPUMP_IOCTL_DEVICE_SET_MS_OS_DESCRIPTOR_PROCESS.

3.2.5 Edge Get Function Section

IOCTL code	USBPUMP_IOCTL_EDGE_GET_FUNCTION_SECTION
In parameter structure	CONST USBPUMP_IOCTL_EDGE_GET_FUNCTION_SECTION_ARG *

Field pConfig	pointer of UCONFIG. This is the current active configuration. The protocol instance should check this UCONFIG structure to figure out that protocol is part of the active configuration. If the protocol object is part of the current active configuration, it should return function section of the extended compact ID feature descriptor.
Field pBuffer	function section save buffer pointer and size of buffer.
Field nBuffer	function section save buffer pointer and size of buffer.
Out parameter	USBPUMP_IOCTL_EDGE_GET_FUNCTION_SECTION_ARG *
Field nActual	actual number of written bytes in the buffer.
Description	This IOCTL is sent from a DataPump core to the UPROTO/UFUNCTION object. The DataPump core sends this IOCTL to retrieve all "function section" of the Microsoft extended compact ID feature descriptor if the client enables this feature using USBPUMP_IOCTL_DEVICE_SET_MS_OS_DESCRIPTOR_PROCESS.

3.3 Storage Specific Edge IOCTLs

Field	Description
Field pObject	Pointer to Client object
Field pClientContext	Pointer to Client context

Table 1 Common in parameter fields for all Edge Storage IOCTLs

Field	Description
Field Status[*]	Return status from Client
Field fReject	Set TRUE to reject request. Field initialized to FALSE by Protocol
Note	The out parameter is initialized by the Protocol with the same values as the in parameter

Table 2 Common out parameter fields for all Edge Storage IOCTLs

[*]: This field is not used in "USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND".

3.3.1 Edge Storage Read

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_READ
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_READ_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field wTag	Command Tag
Field Lba	Starting LBA index
Field LbaCount	Number of LBAs to read
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_READ_ARG *
Description	This IOCTL is sent from Protocol to Client (OS-specific driver) whenever the host wants to initialize a read cycle. The Client issues a Storage-Queue-Read call IOCTL (see Section 4.1, Section 4.6 & Section 4.10) back to Protocol when there is data available for the host to read from the Client supplied buffer. The Protocol responds with a Storage-Read-Done call IOCTL (see Section 3.3.2) when buffer has been read, and then it starts all over again with a Storage-Read IOCTL.

3.3.2 Edge Storage Read Done

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_READ_DONE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_READ_DONE_ARG *
Field LUN Index	Index of the Logical Unit (LUN).
Field wTag	Command Tag
Field pBuf	Pointer to buffer that has been read by the host
Field nBytes	Number of bytes to read
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_READ_DONE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever the host has finished reading a buffer provided by the Client thru the Queue-Read call IOCTL (see Section 4.1, Section 4.6 & Section 4.10)

3.3.3 Edge Storage Write

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_WRITE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_WRITE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field wTag	Command Tag
Field Lba	Starting LBA index
Field LbaCount	Number of LBAs to write
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_WRITE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever the host wants to initialize a write cycle. The Client will issue a Storage-Queue-Write IOCTL call (see Section 4.2, Section 4.7 & Section 4.11) back to Protocol with a buffer for the Protocol to write the data to. The Protocol will respond with a Storage-Write-Data IOCTL (see Section 3.3.4) when there is data available in the buffer. Finally the Client issues a Storage-Write-Done IOCTL call (see Section 4.3, Section 4.8 & Section 4.12) when data has been transferred to the Client medium, and it starts all over again with a Storage-Write IOCTL.

3.3.4 Edge Storage Write Data

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_WRITE_DATA
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_WRITE_DATA_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field wTag	Command Tag
Field pBuf	Pointer to buffer where data has been written
Field nBytes	Number of bytes to written
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_WRITE_DATA_ARG *
Description	This IOCTL is sent from Protocol to Client whenever the Protocol has finished writing to the buffer provided by the Client thru the Queue-Write IOCTL call (see Section 4.2)

3.3.5 Edge Storage Get Status

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_GET_STATUS
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_GET_STATUS_ARG *
Field iLun	Index of the Logical Unit (LUN).
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_GET_STATUS_ARG *
Description	This IOCTL is sent from Protocol to Client whenever Protocol wants to read status of Client.

3.3.6 Edge Storage Reset Device

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_RESET_DEVICE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_RESET_DEVICE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_RESET_DEVICE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever Protocol wants to reset Client.

3.3.7 Edge Storage Load or Eject

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_LOAD_OR_EJECT
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_LOAD_OR_EJECT_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field fLoad	set toTRUE if load-media request
Description	This IOCTL is sent from Protocol to Client that has opened/connected to the leaf object. It is sent whenever the Protocol wants to load or eject the Client medium. Note that this IOCTL doesn't say if the medium should be loaded or ejected, it just toggles the status.

3.3.8 Edge Storage Load or Eject Ex

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_LOAD_OR_EJECT_EX
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_LOAD_OR_EJECT_EX_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field PowerConditions	Power Conditions bits of SCSI Start Stop Unit
Field fNoflushOrFL	NO_FLUSH or FL bit of SCSI Start Stop Unit
Field fLoEj	LoEj bit of SCSI Start Stop Unit
Field fStart	Start bit of SCSI Start Stop Unit
Out parameter	USBPUMP_IOCTL_EDGE_STORAGE_LOAD_OR_EJECT_EX_ARG *
Description	This IOCTL is sent from a storage function to the OS-specific driver that has opened/connected to the leaf object. It is sent whenever the host sends a SCSI Start Stop command.

3.3.9 Edge Storage Prevent Removal

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_PREVENT_REMOVAL
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_PREVENT_REMOVAL_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field fPreventRemoval	Set to TRUE if prevent-media-removal request
Description	This IOCTL is sent from a storage function to the OS-specific driver that has opened/connected to the leaf object. It is sent whenever the host wants to prevent the medium from being REMOVED. Note that this is usually used by the host during a write to indicate that there are pending directory data that must be written to the medium before it can be removed.

3.3.10 Edge Storage Client Command

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_COMMAND
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_COMMAND_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field pCbwcbBuf	Pointer to CBWCB buffer from host
Field nCbwcbBuffer	The valid length of the CBWCB in bytes
Field fReject	Set FALSE if the edge accepts the request, TRUE otherwise. If fReject is TRUE, mass storage function will take care of current CBW. If fReject is FALSE and there is no data phase in this command, current CBW will be handled by client and client should send status using USBPUMP_IOCTL_STORAGE_CUSTOM_SEND_STATUS IOCTL. Otherwise client has to prepare send or receive command data.
Description	This IOCTL is sent from a storage function to the OS-specific driver that has opened/connected to the leaf object. It is sent whenever the host sends a CBW.

3.3.11 Edge Storage Client Send Done

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_SEND_DONE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_SEND_DONE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field pBuf	Pointer to buffer with data from client
Field nBuf	The number of bytes sent in buffer
Description	This IOCTL is sent from a storage function to the OS-specific driver (client) that has opened/connected to the leaf object. It is sent whenever the mass storage function sent a buffer.

3.3.12 Edge Storage Client Receive Done

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_RECEIVE_DONE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_RECEIVE_DONE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field pBuf	Pointer to buffer with data from host
Field nBuf	The number of bytes received in buffer
Description	This IOCTL is sent from a storage function to the OS-specific driver (client) that has opened/connected to the leaf object. It is sent whenever the host sends a custom specific CBW.

3.3.13 Edge Storage Remove Tag

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_REMOVE_TAG
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_REMOVE_TAG_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field fAllTag	Remove all tags
Field wTag	TAG in the Command IU
Description	This IOCTL is sent from a storage function to the OS-specific driver that has opened/connected to the leaf object. It is sent whenever the host wants to remove the request with wTag from the client.

3.3.14 Edge Storage Custom Command

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND_ARG *
Field iLun	Index of the Logical Unit (LUN).

Field pCbwcbBuf	Pointer to CBWCB buffer from host
Field nCbwcbBuffer	The valid length of the CBWCB in bytes
Field DataTransferLength	The number of bytes of data that host expects to send/receive during the execution of this command.
Field fDataTransferFromDeviceToHost	Direction of data transfer. This field is valid only when DataTransferLength is not zero. If DataTransferLength is zero, there is no data phase for this command. TRUE: Data-In; FALSE: Data-Out
Field fReject	Set FALSE if the edge accepts the request, TRUE otherwise. If fReject is TRUE, the mass storage function will take care of current CBW. If fReject is FALSE and there is no data phase in this command, the current CBW will be handled by the client and the client should send status using USBPUMP_IOCTL_STORAGE_CUSTOM_SEND_STATUS IOCTL. Otherwise the client has to prepare send or receive command data.
Description	This IOCTL is sent from a storage function to the OS-specific driver (client) that has opened/connected to the leaf object. It is sent whenever the host sends a custom specific CBW.
Notes	See Section 4.20

3.3.15 Edge Storage Custom Send Done

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_SEND_DONE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_SEND_DONE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field pBuf	Pointer to buffer with data from client
Field nBuf	The number of bytes sent in buffer
Description	This IOCTL is sent from a storage function to the OS-specific driver (client) that has opened/connected to the leaf object. It is sent whenever the mass storage function sent a buffer.

3.3.16 Edge Storage Custom Receive Done

IOCTL code	USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_RECEIVE_DONE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_RECEIVE_DONE_ARG *
Field iLun	Index of the Logical Unit (LUN).
Field pBuf	Pointer to buffer with data from host
Field nBuf	The number of bytes received in buffer
Description	This IOCTL is sent from a storage function to the OS-specific driver (client) that has opened/connected to the leaf object. It is sent whenever the host sends a custom specific CBW.

4 Downcall Services

The following section describes the services the Protocol provides to the Client thru library functions provided by the Protocol.

4.1 Storage Queue Read

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueRead(
        USBPUMP_OBJECT_HEADER *                     pObject,
        VOID *                                      pBuf,
        BYTES                                       LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Read IOCTL (See Section 3.3.1), and when data from the medium has been read into a buffer by the Client.

The parameters are:

pObject	This is a pointer to Protocol instance object.
pBuf	Pointer to buffer
LbaCount	Number of LBAs available in buffer

4.2 Storage Queue Write

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueWrite(
        USBPUMP_OBJECT_HEADER *                     pObject,
        VOID *                                      pBuf,
        BYTES                                       LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write IOCTL (see Section 3.3.3), to provide a buffer for the host to write data to.

The parameters are:

pObject	This is a pointer to Protocol instance object.
pBuf	Pointer to buffer
LbaCount	Max number of LBAs to write to buffer

4.3 Storage Write-Done

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_WriteDone(
        USBPUMP_OBJECT_HEADER *                 pObject,
        USBPUMP_STORAGE_STATUS                  Status
        );

Header-file : uclientlibstorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write-Data IOCTL (see Section 3.3.4), when the Client has finished writing data to its medium. This function could be signaled during the transfer of the last chunks of data from the host for appropriate buffer handling to support parallel operation between MSC and MMCSD.

The parameters are:

pObject	This is a pointer to Protocol instance object.
Status	Status of write operation to Client medium

4.4 Storage Set Current Medium

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_SetCurrentMedium(
        USBPUMP_OBJECT_HEADER *                 pObject,
        BOOL                                    fPresent,
        BYTES                                   LbaMax,
        BYTES                                   LbaSize
        );

Header-file : ufnapistorage.h
```

This function is used by the Client when there has been a change of medium status. This function should be called by the Client during initialization to set the state of the medium.

The parameters are:

pObject	This is a pointer to Protocol instance object.
fPresent	Indicates whether medium is present or not
LbaMax	Max number LBAs on current medium
LbaSize	Size in bytes of each LBA

4.5 Storage Set Device Properties

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_SetDeviceProperties(
        USBPUMP_OBJECT_HEADER *                     pObject,
        USBPUMP_STORAGE_DEVICE_TYPE                 DeviceType,
        BOOL                                        fRemovable,
        CONST TEXT *                                pVendorId,
        CONST TEXT *                                pProductId,
        CONST TEXT *                                pVersion
        );

Header-file:  ufnapistorage.h
```

This function is used by the Client when the ATAPI device properties need to be updated.

This information may also be given at startup of Protocol thru the ATAPI configuration structure (see Section 2.2). The parameters are:

pObject	This is a pointer to Protocol instance object.
DeviceType	USBPUMP_STORAGE_DEVICE_TYPE indicating ATAPI peripheral device type.
fRemovable	Indicates if this device has removable medium or not
pVendorId	Pointer to vendor id string. This is an ANSI string that is used for ATAPI-level Vendor-ID queries, and is not necessarily related to the USB vendor ID.
pProductId	Pointer to product id string. This is an ANSI string that is used for ATAPI-level Product ID queries, and is not necessarily related to the USB product ID.
pVersion	Pointer to version string. This is an ANSI string that is used for ATAPI-level version-number queries, and is not necessarily related to the USB product version number.

4.6 Storage Queue Read V2

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueReadV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        VOID *                                      pBuf,
        BYTES                                       LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Read IOCTL (See Section 3.3.1), and when data from the medium has been read into a buffer by the Client.

The parameters are:

pIoObject	This is a pointer to Protocol instance object.
iLun	LUN Index
pBuf	Pointer to buffer
LbaCount	Number of LBAs available in buffer

4.7 Storage Queue Write V2

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueWriteV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        VOID *                                      pBuf,
        BYTES                                       LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write IOCTL (see Section 3.3.3), to provide a buffer for the host to write data to.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index
pBuf	Pointer to buffer
LbaCount	Max number of LBAs to write to buffer

4.8 Storage Write-Done V2

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_WriteDoneV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        USBPUMP_STORAGE_STATUS                      Status
        );
```

```
Header-file : uclientlibstorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write-Data IOCTL (see Section 3.3.4), when the Client has finished writing data to its medium. This function could be signaled during the transfer of last chunks of data from the host for appropriate buffer handling to support parallel operation between MSC and MMCSD.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index
Status	Status of write operation to Client medium

4.9 Storage Set Current Medium V2

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_SetCurrentMediumV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BOOL                                        fPresent,
        BOOL                                        fWriteProtected,
        BYTES                                       LbaMax,
        BYTES                                       LbaSize
        );

Header-file : ufnapistorage.h
```

This function is used by the Client when there has been a change of medium status. This function should be called by the Client during initialization to set the state of the medium.

The parameters are:

pObject	This is a pointer to Protocol instance object.
fPresent	Indicates whether medium is present or not
fWriteProtected	Indicates whether medium is write-protected or not
LbaMax	Max number LBAs on current medium
LbaSize	Size in bytes of each LBA

4.10 Storage Queue Read V3

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueReadV2(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        BYTES                                        iLun,
        UNIT16                                       wTag,
        VOID *                                       pBuf,
        BYTES                                        LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Read IOCTL (See Section 3.3.1), and when data from the medium has been read into a buffer by the Client.

The parameters are:

pIoObject	This is a pointer to Protocol instance object.
iLun	LUN Index
Wtag	Command Tag
pBuf	Pointer to buffer
LbaCount	Number of LBAs available in buffer

4.11 Storage Queue Write V3

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_QueueWriteV2(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        BYTES                                        iLun,
        UINT16                                       wTag,
        VOID *                                       pBuf,
        BYTES                                        LbaCount
        );

Header-file : ufnapistorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write IOCTL (see Section 3.3.3), to provide a buffer for the host to write data to.

The parameters are:

pObject	This is a pointer to Protocol instance object.

iLun	LUN Index
wTag	Command Tag
pBuf	Pointer to buffer
LbaCount	Max number of LBAs to write to buffer

4.12 Storage Write-Done V3

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_WriteDoneV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        UINT16                                      wTag,
        USBPUMP_STORAGE_STATUS                      Status
        );

Header-file : uclientlibstorage.h
```

This function is used by the Client in response to a Protocol initiated Storage-Write-Data IOCTL (see Section 3.3.4), when the Client has finished writing data to its medium. This function could be signaled during the transfer of last chunks of data from the host for appropriate buffer handling to support parallel operations between MSC and MMCSD.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index
wTag	Command Tag
Status	Status of write operation to Client medium

4.13 Storage Set Current Medium V3

```
Prototype :

USBPUMP_IOCTL_RESULT UsbFnApiStorage_SetCurrentMediumV3(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        BOOL                                        fPresent,
        BOOL                                        fWriteProtected,
        BYTES                                       LbaMax,
        BYTES                                       LbaSize
        );
```

```
Header-file : ufnapistorage.h
```

This function is used by the Client when there has been a change of medium status. This function should be called by the Client during initialization to set the state of the medium. This function needs to be called for every LUN affected.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index
fPresent	Indicates whether medium is present or not
fWriteProtected	Indicates whether medium is write-protected or not
LbaMax	Max number LBAs on current medium
LbaSize	Size in bytes of each LBA

4.14 Storage Set Device Properties V2

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_SetDevicePropertiesV2(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        USBPUMP_STORAGE_DEVICE_TYPE                 DeviceType,
        BOOL                                        fRemovable,
        CONST TEXT *                                pVendorId,
        CONST TEXT *                                pProductId,
        CONST TEXT *                                pVersion
        );

Header-file:  ufnapistorage.h
```

This function is used by the Client when the ATAPI device properties need to be updated. This information may also be given at startup of Protocol thru the ATAPI configuration structure (see Section 2.2).

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
DeviceType	USBPUMP_STORAGE_DEVICE_TYPE indicating ATAPI peripheral device type.
fRemovable	Indicates if this device has removable medium or not

pVendorId	Pointer to vendor id string. This is an ANSI string that is used for ATAPI-level Vendor-ID queries, and is not necessarily related to the USB vendor ID.
pProductId	Pointer to product id string. This is an ANSI string that is used for ATAPI-level Product ID queries, and is not necessarily related to the USB product ID.
pVersion	Pointer to version string. This is an ANSI string that is used for ATAPI-level version-number queries, and is not necessarily related to the USB product version number.

4.15 Storage Client Set Mode

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ClientSetMode(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        BOOL                                        fEnableTransparentMode,
        BOOL *                                      fOldMode
        );

Header-file:  ufnapistorage.h
```

This function is used by Client to enable/disable SET_TransparentMode mode. If enabled, the mass storage function will send commands to host using USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_COMMAND.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
fEnableTransparentMode	Current Status
fOldMode	Old Status

4.16 Storage Client Sent Data

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ClientSendData(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        VOID *                                      pBuf,
        BYTES                                       nBuf
        );
```

```
Header-file: ufnapistorage.h
```

This function is used by the Client to send a buffer of data to the host. The mass storage function will send data to the host. When all data was sent, the mass storage function will send notification to the Client using USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_SEND_DONE edge IOCTL.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
pBuf	Pointer to buffer with data to client
nBuf	Number of bytes available in buffer

4.17 Storage Client Receive Data

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ClientReceiveData(
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        VOID *                                      pBuf,
        BYTES                                       nBuf
        );

Header-file: ufnapistorage.h
```

This function is used by the Client to receive data from the host. The mass storage function will receive data from the host. When specified size of data was received, the mass storage function will send notification to the Client using USBPUMP_IOCTL_EDGE_STORAGE_CLIENT_RECEIVE_DONE edge IOCTL.

The parameters are:

pObject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
pBuf	Pointer to buffer with data from client
nBuf	Number of bytes available in buffer

4.18 Storage Client Sent Status

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ClientSendStatus(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        BYTES                                        iLun,
        UINT8                                        bCswStatus,
        USBPUMP_STORAGE_STATUS                       StorageStatus
        );

Header-file: ufnapistorage.h
```

This function is called by the Client to send CSW (Command Status Wrapper) to the host.

The parameters are:

pIobject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
bCswStatus	Status of CSW. Indicates the success or failure of the command. The client shall set this byte to zero if the command completed successfully. A non-zero value shall indicate a failure during command execution.
StorageStatus	Status code of USBPUMP_STORAGE_STATUS.

4.19 Storage Client Get Inquiry Data

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ClientGetInquiryData(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        BYTES                                        iLun,
        VOID *                                       pBuf,
        BYTES                                        nBuf,
        BYTES *                                      pWriteCount
        );

Header-file: ufnapistorage.h
```

This function is called by the Client to get CSW (Command Status Wrapper) status inquiry information.

The parameters are:

pIobject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated

pBuf	Pointer of inquiry buffer
nBuf	Size of inquiry buffer
pWriteCount	Number of written bytes

4.20 Storage Custom Send Status

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_CustomSendStatus (
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        UINT8                                       bCswStatus,
        USBPUMP_STORAGE_STATUS                      StorageStatus
        );

Header-file:  ufnapistorage.h
```

This function is called by client to send CSW (Command Status Wrapper) to the host.

The parameters are:

pIobject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
bCswStatus	Indicates the success or failure of the command. The client shall set this byte to zero if the command completed successfully. A non-zero value shall indicate a failure during command execution.
StorageStatus	Status code of USBPUMP_STORAGE_STATUS.

4.20.1 An Example of Supporting Custom SCSI Commands

Here is an example of using USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND and USBPUMP_IOCTL_STORAGE_CUSTOM_SEND_STATUS to support custom SCSI commands.

In SCSI terminology, the communication takes place between an initiator and a target; the initiator is sending commands in a Command Descriptor Block (CDB), which consists of a one byte operation code followed by five or more bytes containing command-specific characters. At the end of the sequence the target returns a status code byte. Table 3 shows some examples of SCSI commands.

BYTE	Description
00H	Test Unit Ready command. Used to determine if a device is ready to transfer data.
12H	Inquiry. Return basic information of device.
03H	Request sense. Returns any error code from the previous commands that return an error status.
…	…
D6H	Custom SCSI code

Table 3 Example of Standard/Custom SCSI CDB commands

When mass storage protocol received unknown command with dCBWDataTransferLength equal to 0, it will call USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND. Client's IOCTL handler should check command (pCbwcbBuffer[0]) and decide to reject or accept this command. If it accepts this command, client should call UsbFnApiStorage_CustomSendStatus() API. This UsbFnApiStorage_CustomSendStatus() API will send USBPUMP_IOCTL_STORAGE_CUSTOM_SEND_STATUS IOCtl.

Client mass storage IOCTL handler should support USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND IOCtl.

```
USBPUMP_IOCTL_RESULT
MscDemoI_Ramdisk_Ioctl(
    USBPUMP_OBJECT_HEADER *  pDevObjHdr,
    USBPUMP_IOCTL_CODE       Ioctl,
    CONST VOID *             pInParam,
    VOID *                   pOutParam
    )
{
…
case USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND:
        return MscDemoI_Ramdisk_CustomCommand(
            pDevObj,
            pOutParam
            );
…
}
```

In addition, create new routine to handle USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND:

```
USBPUMP_IOCTL_RESULT
MscDemoI_Ramdisk_CustomCommand(
   MSCDEMO_DEVOBJ *                                  pDevObj,
   USBPUMP_IOCTL_EDGE_STORAGE_CUSTOM_COMMAND_ARG *   pOutArg
   )
   {
   MSCDEMO_DEVOBJ_RAMDISK * CONST   pRamDisk = pOutArg->pClientContext;
```

```
USBPUMP_IOCTL_RESULT     Result;
…
/* This is sample code for testing custom SCSI command */
if (pOutArg->pCbwcbBuffer[0] == 0xd6)
    {
    pOutArg->fReject = FALSE;

    Result = UsbFnApiStorage_CustomSendStatus(
            pRamDisk->udrd_DevObj.pIoObject,
            pOutArg->iLun,
            UPROTO_MSCBOT_CSW_STATUS_SUCCESS,
            USBPUMP_STORAGE_STATUS_NONE
            );
    }
else
    {
    pOutArg->fReject = TRUE;
    Result = USBPUMP_IOCTL_RESULT_SUCCESS;
    }

return Result;
}
```

4.21 Storage Custom Send Data

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_CustomSendData (
        USBPUMP_OBJECT_HEADER *                         pIoObject,
        BYTES                                           iLun,
        VOID *                                          pBuf,
        UINT32                                          nBuf
        );

Header-file:  ufnapistorage.h
```

This function is called by the client to send command data to the host.

The parameters are:

pIobject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
pBuf	Indicates the buffer which includes the command data.
nBuf	Size of the command data which will be sent to the host.

Please see an example in Figure 3.

4.22 Storage Custom Receive Data

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_CustomReceiveData (
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BYTES                                       iLun,
        VOID *                                      pBuf,
        UINT32                                      nBuf
        );

Header-file:  ufnapistorage.h
```

This function is called by the client in order to receive command data from the host.

The parameters are:

pIobject	This is a pointer to Protocol instance object.
iLun	LUN Index whose information is to be updated
pBuf	Indicates the buffer which is used to receive command data.
nBuf	Size of the command data from the host.

Please see an example in Figure 4.

4.23 Storage Control Last Lun

```
Prototype:

USBPUMP_IOCTL_RESULT UsbFnApiStorage_ControlLastLun (
        USBPUMP_OBJECT_HEADER *                     pIoObject,
        BOOL                                        fEnableLastLun
        );

Header-file:  ufnapistorage.h

No descriptop??
```

The parameters are:pIobject	This is a pointer to Protocol instance object.
fEnableLastLun	Indicates whether mass storage protocol shows last LUN or not.

5 Other Considerations

[USBMASS] requires that USB Mass Storage devices have unique serial numbers of a specific format. The USB DataPump has complete support for serial numbers, but some platform-specific code is needed to actually provide the serial number to the DataPump.

6 Performance Considerations

6.1 Write

For write, we may not want to signal write complete until we really know that the entire data has been successfully transferred. Instead of signaling the Storage Write-Done function at every Storage-Write-Data IOCTL, it would be appropriate to signal only for the transfer of the last chunks of data. The interim chunks could be handled using Storage QueueWrite indicating the write operation has not yet completed. This maintains parallel operation between USB and MMCSD. For further explanation, refer to Figure 2 and compare the difference with Figure 1.

6.2 Read

The Pre-read could be handled such that the first read can figure out the starting LBA and the count could tell how much data the host is looking for.

6.3 General

We are using an 8KB buffer for the Mass storage interface. It is common for the host to perform a 64KB transfer by splitting it in to 8X8KB iterations of USB/MMCSD transfers. We could save a lot by increasing the buffer size to do a transfer of a bigger chunk of data in one call.

7 Demo Applications

The DataPump Professional and Standard installations contain a RAM-disk demo in `usbkern/app/mscdemo` and `usbkern/proto/msc/applib` that can be used as reference on how to use the MSC protocol.

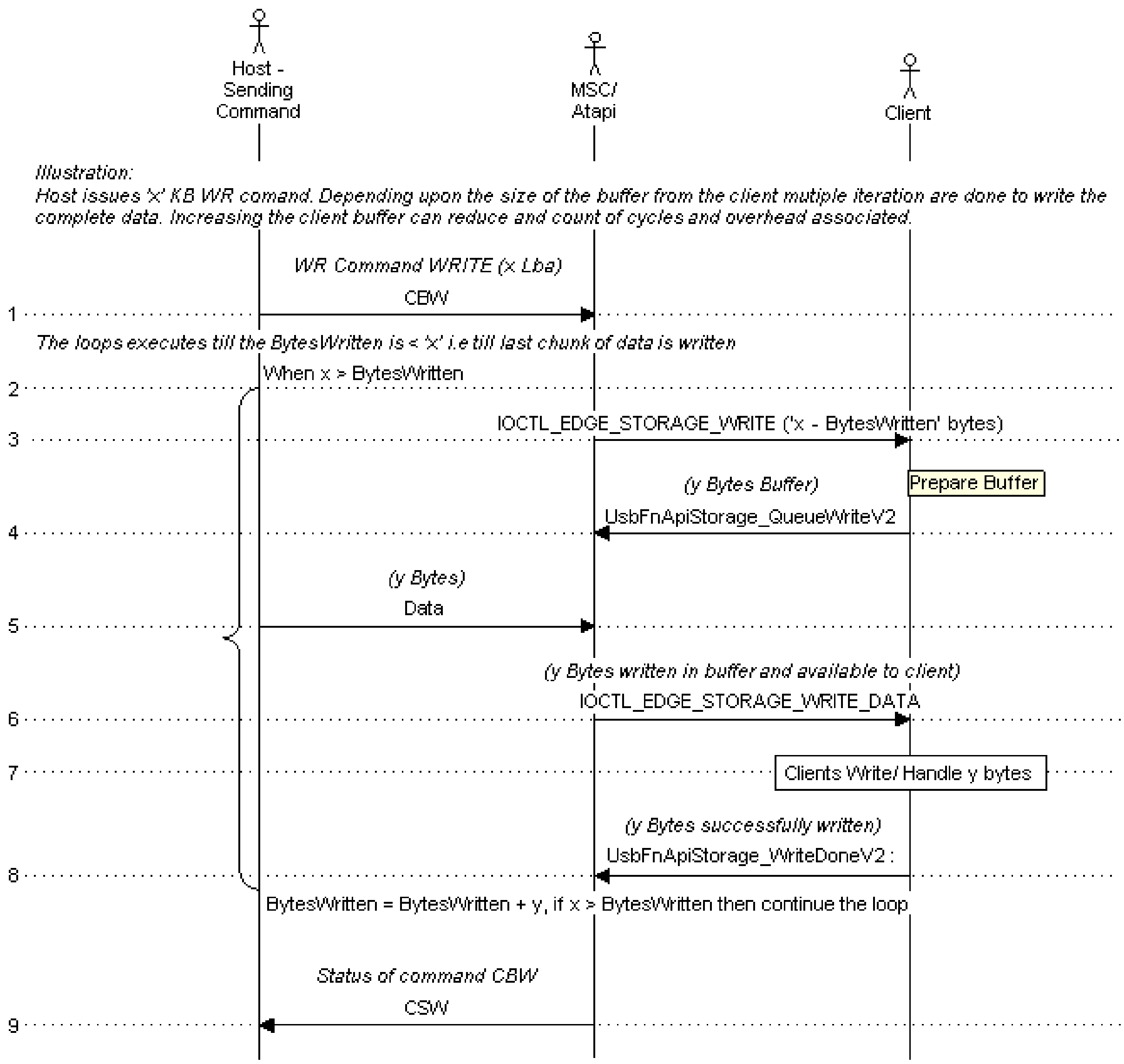

Figure 1 Sequence diagram of Standard procedure for a Write operation

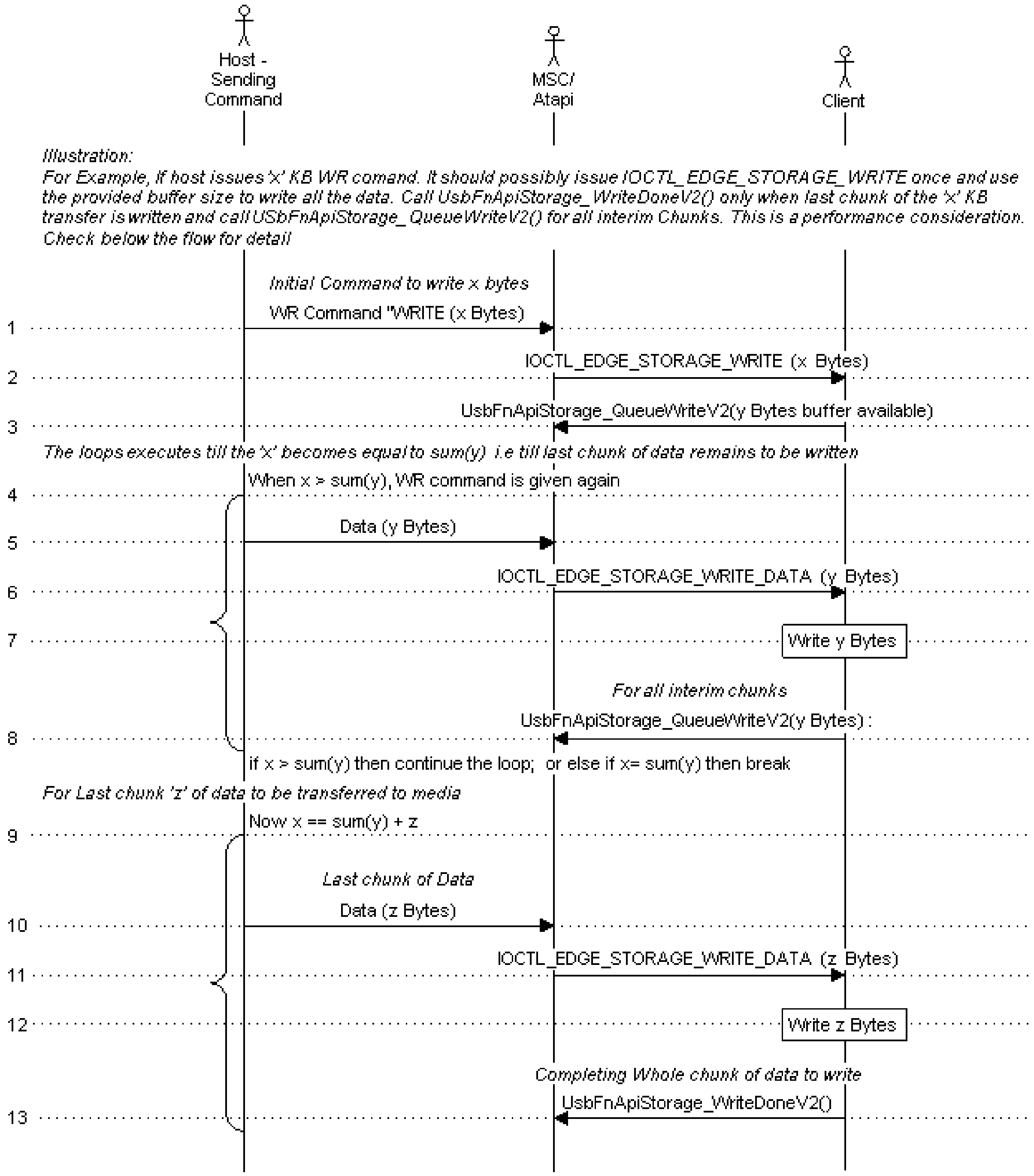

Figure 2 Sequence diagram with Performance consideration for a Write operation

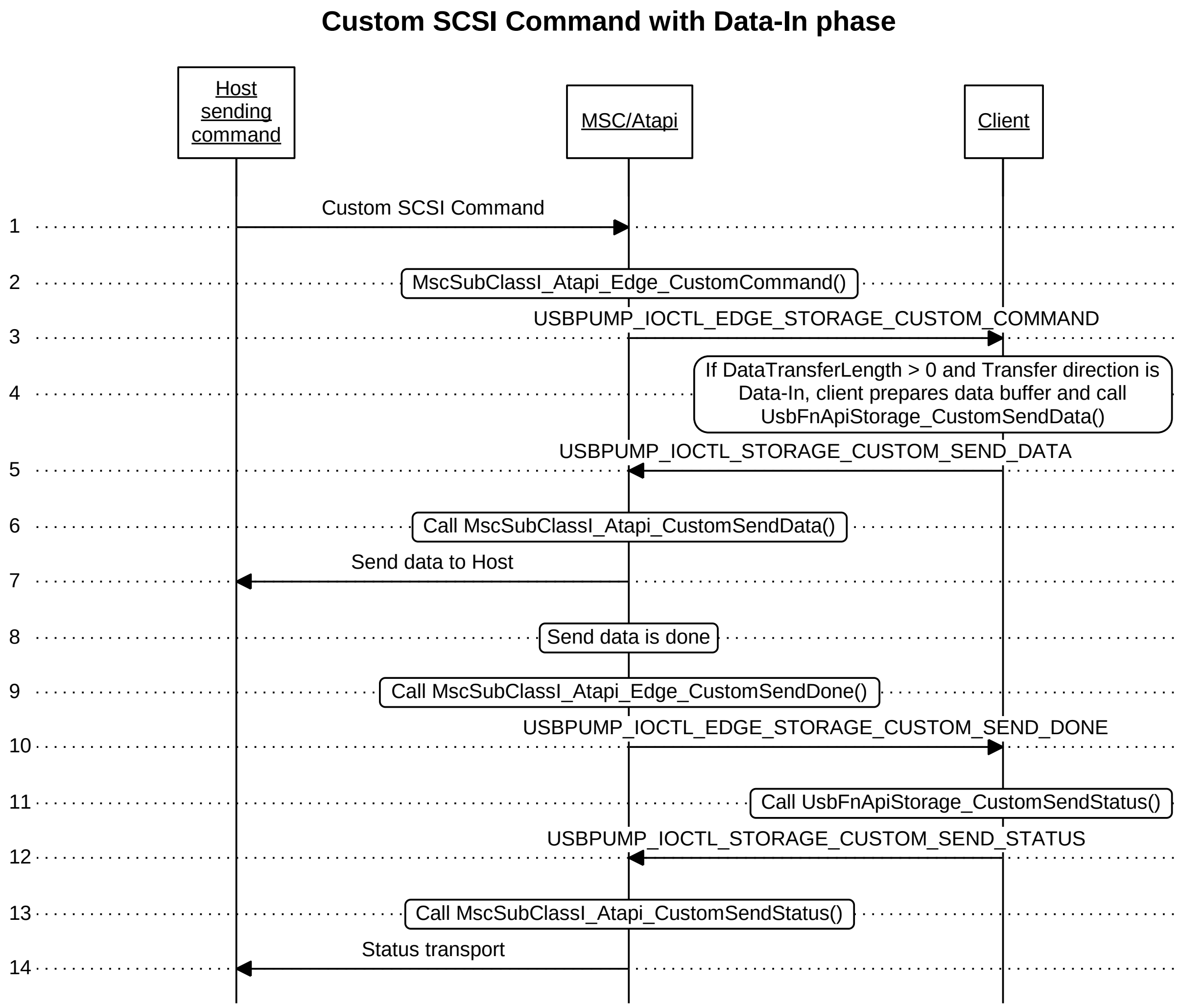

Figure 3 Sequence diagram of Custom SCSI command with Data-In phase

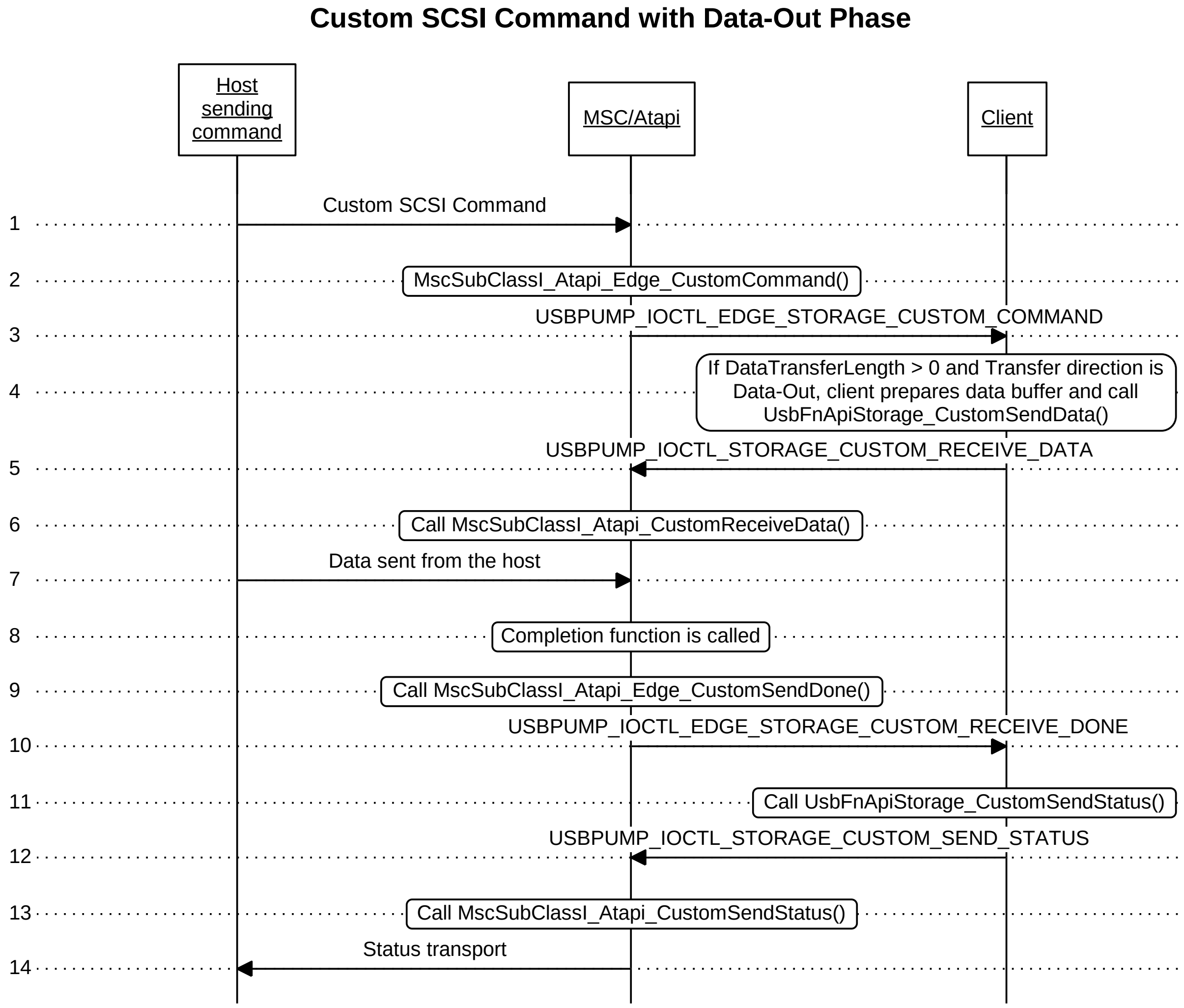

Figure 4 Sequence diagram of Custom SCSI command with Data-Out phase

8 Contact Information

Headquarters – Singapore

Bridgetek Pte Ltd
178 Paya Lebar Road, #07-03
Singapore 409030
Tel: +65 6547 4827
Fax: +65 6841 6071

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Branch Office – Taipei, Taiwan

Bridgetek Pte Ltd, Taiwan Branch
2 Floor, No. 516, Sec. 1, Nei Hu Road, Nei Hu District
Taipei 114
Taiwan, R.O.C.
Tel: +886 (2) 8797 5691
Fax: +886 (2) 8751 9737

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Branch Office - Glasgow, United Kingdom

Bridgetek Pte. Ltd.
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales.emea@brtichip.com
E-mail (Support) support.emea@brtchip.com

Branch Office – Vietnam

Bridgetek VietNam Company Limited
Lutaco Tower Building, 5th Floor, 173A Nguyen Van Troi,
Ward 11, Phu Nhuan District,
Ho Chi Minh City, Vietnam
Tel : 08 38453222
Fax : 08 38455222

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Web Site

http://brtchip.com/

Distributor and Sales Representatives

Please visit the Sales Network page of the Bridgetek Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Document References

AN_402_MCCI_USB_DataPump_UserGuide

AN_400_MCCI_USB_Resource_Compiler_UserGuide

Multi-Media Command Set 2, available at http://www.t10.org/drafts.htm

Universal Serial Bus Specification, version 2.0/3.0 (also referred to as the USB Specification). This specification is available on the World Wide Web site http://www.usb.org.

Universal Serial Bus Mass Storage Class Specification Overview, version 1.4. This specification is available at http://www.usb.org/developers/devclass.

Universal Serial Bus Mass Storage Class Bulk-Only Transport, version 1.0 (also referred to as the MSC BOT Specification, where "BOT" stands for "Bulk-Only Transport"). This specification is available at http://www.usb.org/developers/devclass.

Acronyms and Abbreviations

Terms	Description
ATAPI	"Advanced Technology Attachment Packet Interface". Originally defined for transporting SCSI-like commands over IDE interfaces. The command sets defined by this committee may be used by USB Mass Storage Devices. MCCI's Mass Storage Protocol Library implements this command set.
BOT	Bulk-Only Transport, one of the ways defined by the USB-IF Device Working Group for transporting commands and results between the USB host and a USB mass storage device
CBW	Command Block Wrapper, A structure which maintains the commands send by USB Host during Bulk transfers.
CDB	Command Descriptor Block, A Block (part of CBW) which contains data in the format defined by SCSI command set specification.
CSW	Command Status Wrapper, A structure that maintains the status send by USB Device during Bulk transfers.
IDE	Integrated Device Extension, the original electrical interface and command set used in the IBM PC/AT
LBA	Logical block addressing, is a common scheme used for specifying the location of blocks of data stored on storage devices.
LUN	Logical Unit Number, A number used to identify a logical unit. A logical unit number is assigned when a host scans a SCSI device and discovers a logical unit. An USB Device supports multiple logical units (LUNs) which can operate separately, for example one unit could have an SD card as media and another one could have a RAM disk as media.
MSC	Mass Storage Class – the family of USB class specifications that specify

	standard ways of implementing a mass-storage class device
MMCSD	Multimedia Card (MMC)/Secure Digital (SD), MMC is a memory card standard used for solid-state storage. SD is an extension of MMC
SCSI	Small Computer System Interface. It is set of standards for physically connecting and transferring data between computers and peripheral devices.
SFF-8020i / SFF-8070i	The ATAPI command set for CD-ROMs / for floppies
USB	Universal Serial Bus
USB-IF	USB Implementer's Forum, the consortium that owns the USB specification, and which governs the development of device classes
USBRC	MCCI's USB Resource Compiler, a tool that converts a high-level description of a device's descriptors into the data and code needed to realize that device with the MCCI USB DataPump.

Appendix B – List of Tables & Figures

List of Tables

Table 1 Common in parameter fields for all Edge Storage IOCTLs... 15
Table 2 Common out parameter fields for all Edge Storage IOCTLs... 15
Table 3 Example of Standard/Custom SCSI CDB commands 36

List of Figures

Figure 1 Sequence diagram of Standard procedure for a Write operation 41
Figure 2 Sequence diagram with Performance consideration for a Write operation..................... 42
Figure 3 Sequence diagram of Custom SCSI command with Data-In phase 43
Figure 4 Sequence diagram of Custom SCSI command with Data-Out phase 44

Appendix C – Revision History

Document Title:	AN_403 MCCI USB DataPump Mass Storage Protocol Users Guide
Document Reference No.:	BRT_000124
Clearance No.:	BRT#094
Product Page:	http://brtchip.com/product/
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	Initial release	2017-09-13