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

MCCI USB DataPump WMC Protocol User's Guide

Engineering Report 950000255
Rev. D
Date: 2011/09/24

PROPRIETARY NOTICE AND DISCLAIMER

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

Document Release History

Rev A	2003/10/16	Original release
Rev. B	2011/04/15	Changed all references to Moore Computer Consultants, Inc. to MCCI Corporation. Changed document numbers to nine digit versions. DataPump 3.0 Updates.
Rev. C	2011/06/13	Updated Reference Documents section and Implementation Overview section.
Rev. D	2011/09/24	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF PROGRAMS

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 Communication Device Class (CDC) - Advanced Control Model (ACM) protocol. We present programming information for integrating this support into user's firmware, to create a USB device that presents an ACM-modem class interface to the host PC.

We do not discuss host software issues other than mentioning that MCCI has CDC-ACM compliant host driver for Microsoft Windows operating systems.

1.1 Glossary

ACM — Abstract Control Model – a device subclass defined by [USBCDC] and further extended by [USBWMC], for handling AT-command modems.

CDC — Communication Device Class – the family of USB class specifications that specify standard ways of implementing communication devices such as modems, Ethernet interfaces, cable modems, ADSL modems, and so forth.

OBEX — Object Exchange protocol – a transport-independent means of exchanging information between light-weight devices, using a protocol similar to HTTP 1.1. Defined by [IrOBEX]

TA — *See* Terminal Adapter

Terminal Adapter — An abstraction from the [USBWMC] specification. Each Terminal Adapter corresponds to a single function in a (possibly multi-function) WMC device. A given Terminal Adapter might be represented on USB as a CDC ACM Modem, as a WMC OBEX function, or as a Device Management function.

USB — Universal Serial Bus

USB-IF — USB Implementers 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.

WMC — Wireless Mobile Communications, a class standard defined in [USBWMC].

1.2 Referenced Documents

[DPIOCTL] — *OVERVIEW-ioctl.txt*, from USB DataPump installation `usbkern/doc/` directory.

[DPOVERVIEW] OVERVIEW-appinit.txt and OVERVIEW-objects.txt available at /usbkern/doc/ in Professional and Standard DataPump installation.

[DPREF] *MCCI USB DataPump User's Guide*, MCCI Engineering Report 950000066

[DPUSBRC] *USBRC User's Guide*, MCCI Engineering Report 950000061

[IrOBEX] *IrDA Object Exchange Protocol IrOBEX, V1.2, March 19, 1999.* This specification is available online from the website http://www.irda.org/

[USBCORE] *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**.

[USBCDC] *Universal Serial Bus Communication Device Class Specification,* version 1.1. This specification is available at **http://www.usb.org/developers/devclass.**

[USBWMC] *Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices, Version 1.0, November 23, 2001*. This specification is available at **http://www.usb.org/developers/devclass.**

1.3 Overview

The MCCI WMC Protocol Library, in conjunction with the MCCI USB DataPump, provides a straightforward, portable environment for implementing CDC-ACM and CDC WMC ACM compliant AT-compatible modem devices and modem-like devices over USB using the following protocols:

- USB CDC 1.1 Abstract Control Model (ACM) protocol [USBCDC] and the USB WMC 1.0 ACM,
- USB WMC 1.0 Object Exchange (OBEX) and
- USB WMC 1.0 Device Management protocols [USBWMC].

The MCCI WMC Protocol Library can be used to create a standalone 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 WMC 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 [DPREF], the USB CDC Specification [USBCDC] and the USB WMC Specification [USBWMC].

The WMC Protocol Library encapsulates issues regarding USB transactions so that the engineer can concentrate on the modem or communication portions of the target device. It handles:

- Enumeration
- Configuration

- Function activation and deactivation
- Standard command decoding and validation
- Generating notifications in response to device-side events
- Demultiplexing encapsulated commands
- Multiplexing encapsulated responses
- Translating the various data-plane protocols into a consistent upper level API for use by clients.
- Presenting USB bus events to clients in a consistent way.

The WMC Protocol Library does not handle the following issues:

- The implementation of the AT interpreter itself.
- The implementation of the OBEX server.
- Allocation of memory for buffering data transmitted and received.

1.3.1 Implementation Overview

When using the DataPump WMC 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 WMC 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.

The WMC Protocol Library was designed to allow a great deal of client code to be shared for all device types. To this end, the Protocol Library provides an abstract interface and hides most of the details from the client.

All of the Protocol objects created by this library share a common set of behaviors. Each object provides at least one USB DataPump `UDATAPLANE` object; each `UDATAPLANE` in turn contains two `UDATASTREAM` objects. `UDATAPLANE` objects are used to model full-duplex data streams to the client software. The client sends and receives data by submitting `UBUFQE` queue elements to a `UDATASTREAM`. The Protocol object arranges to fill or empty the buffer asynchronously, and then indicates completion to the client via a client-supplied callback function.

The implementation of the `UDATAPLANE` and `UDATASTREAM` is hidden from the client. Data can flow either over a bulk pipe, or over the default pipe, using the SEND_ENCAPSULATED_COMMAND and GET_ENCAPSULATED_RESPONSE mechanism defined in [USBWMC]. The Protocol Library handles all details of moving data, so that the client code doesn't care.

The `UDATAPLANE` object includes an up-call facility; registered clients will be notified when the underlying data streams are activated or deactivated due to USB activity. The `UDATASTREAM` semantics also include the ability to "park" `UBUFQE`s when the host is not available; this simplifies client design.

The control plane of the USB interface is managed in response to host activities. Host commands are converted into up-calls from the Protocol to a client-supplied IOCTL function. Some IOCTL operations are specific to the CDC Modem Function – these are used to transfer "baud rate" and "modem status" information that is not relevant to OBEX and Device Management functions. Others are generic, and are used for all functions.

The CDC Modem implementation provides two `UDATAPLANE`s, one for data, one for control. The data `UDATAPLANE` is used with the bulk IN/OUT pair, and the control `UDATAPLANE` is used with encapsulated command transport. A properly coded AT interpreter can use the control `UDATAPLANE` to provide status while on-line, without requiring any additional hardware resources.

The OBEX implementation provides a single data `UDATAPLANE`. This is activated when the host selects alternate setting 1 of the Data class interface. It is intended to be connected to an OBEX server supplied by the Client, but the WMC Protocol Library does not enforce this.

The Device Management implementation provides a single control `UDATAPLANE`; however, this `UDATAPLANE` doesn't require any endpoints, apart from a notification endpoint associated with the control pipe. This interface is suitable for low duty-cycle operations, such as obtaining status from the phone. It is intended to be connected to an AT interpreter supplied by the Client, but the WMC Protocol Library does not enforce this.

Note that the DataPump abstraction that is used with all of these Protocol Objects is called the "USB DataPump Abstract Modem". All three kinds of functions generically use the same Abstract Modem API; but only one (the CDC Modem) actually provides modem functionality.

2. Initialization and Setup

To include WMC support in your application, you must:

1. Supply the appropriate descriptors (See Section 2.1 below)

2. Arrange for the WMC library to be included in your application at link time (See 6.2 below)

3. Arrange for the WMC library to be initialized, by including it in the application initialization vector. (See Section 3.1 below)

4. Provide client code that will search the resulting object table and bind to the DataPump objects created by the WMC library. (See Section 2.2 below)

5. Provide client code that connects the target application environment (AT interpreter, OBEX server) to the DataPump objects during normal USB operations. (See Section 4 and 5 below)

The protocol library will create one Protocol Instance for each supported WMC or CDC function that it finds in the descriptor set. If a WMC or CDC function appears in multiple configurations, then the protocol library will create multiple instances, one instance for each configuration.

The WMC Protocol Instance code performs all command set decoding, however it contains no code that actually knows how to read and write data, decode AT commands, or do OBEX operations. For this purpose, the system integrator must provide client code, and client initialization code.

2.1 USB Descriptor Requirements

The Protocol Library code parses the device's configuration descriptors, and creates Protocol Instances for each supported CDC Subclass function found in the descriptor set. The supported Subclass Functions are found by matching against interface descriptors with bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol as shown in Table 1.

Table 1. Supported Interface Classes

bInterfaceClass	bInterfaceSubClass	bInterfaceProtocol	Description
0x02	0x02	0x01	CDC or WMC Abstract Control Model modem port. According to [USBWMC], bInterfaceProtocol denotes the AT command set; it is normally not used for matching.
0x02	0x09	0x01	Device Management AT-Command port. bInterfaceProtocol is normally not used for matching.
0x02	0x0B	0x00	WMC 1.0 OBEX port. bInterfaceProtocol should be zero for compliance with [USBWMC]

For each matching interface, the CDC Union Descriptor, if any, is parsed to find the Data class interfaces associated with the function.

The Protocol Library is not sensitive to the order of the endpoints in the descriptor set, nor to the wMaxPacketSize of the endpoints.

Since the WMC protocol library works by decoding the USB descriptors and the descriptors for WMC functions are quite complicated, in this section we give examples of how to encode these descriptors.

These examples are all expressed in the USBRC input language, in particular in the latest version of USBRC that ships with DataPump release 3.0. This version of USBRC understands the core chapter 9 descriptors, but has no special knowledge of CDC or WMC. Therefore, the descriptors must be expressed using the "private-descriptor" and "raw" descriptor constructs. Although this is quite straightforward, it means that you must be careful when using these examples – if you change interface numbers, you must change interface numbers consistently throughout the example. For more details on this refer *[DPUSBRC]* documentation.

To help identify the areas that must be changes, we have highlighted the sensitive text in the examples by highlighting in yellow the text that might have to change.

2.1.1 Descriptors for ACM Modem Functions

Program 1 shows how the descriptors for an ACM modem should be coded. The WMC Protocol Library uses some of these descriptors to configure itself; others are ignored. However all the descriptors are needed for a standard-compliant modem.

The requirements of the WMC Protocol Library for an ACM modem are:

1. A Communication class interface with subclass 0x02.

2. A Data class interface.

3. One Interrupt IN endpoint, associated with the Communication class interface. The recommended endpoint wMaxPacketSize is 8 bytes or larger.

4. One Bulk IN endpoint, associated with the Data class interface.

5. One Bulk OUT endpoint, associated with the Data class interface.

6. A CDC Union descriptor, which mentions the Communication class interface and the Data class interface.

MCCI test cases always include substantially the same descriptor set shown in Program 1. Therefore, MCCI recommends that your descriptor set always include the CDC Header descriptor, the ACM Functional Descriptor, and the Call Control descriptor, as shown.

In the example, the USB device hardware allows you to have an IN endpoint and an OUT endpoint with the same endpoint number. Although this is legal according to [USBCORE], many device controllers do not support this feature. You should be careful to verify that the endpoint numbers you give to USBRC will work with your hardware.

Program 1. Descriptor Fragment for ACM Modem

```
#
# The Comm-class control interface
#
interface 1
        {
        class    0x02   #comm class
        subclass 0x02   #ACM
        protocol 0x01   #common AT commands
        name     S_TA1  # string

        # here are the class descriptors.
        private-descriptors
                {
                # header
                raw     {
```

```
                0x24    # interface
                0       # functional descriptor
                word(0x110)     # CDC version 1.1
                };

        # union descriptor.
        raw     {
                0x24    # interface
                6       # “union”
                % index of master interface % 1
                % index of data interface 1 % 2
                };

        # call control
        raw     {
                0x24    # interface/class.
                1       # “call control”
                0x03    # bit 0: can handle call
                        #       control
                        # bit 1: can do call control
                        #       over data ifc
                % index of call control ifc: %
                  2
                };

        # ACM functional descriptor
        raw     {
                0x24    # interface/class
                2       # “acm”
                0x07    # bit 3: NO net connect
                        # bit 2: send break
                        # bit 1: set line/set control
                        #        *get line coding
                        #        *serial_state
                        # bit 0: set/get/clear comm
                        #       feature
                };
        }

    # here is the notification endpoint
    endpoints
        interrupt in packet-size 16
                polling-interval 16
        ;
    }

# data interface TA#1
interface 2
    {
    alternate-setting 0
        class    0x0A   #data class
        subclass 0x00   #
        protocol 0x00   #
        name    S_TA1_DATA_1
        endpoints
                bulk in
                bulk out
```

```
                                ;
                }
```

2.1.2 Descriptors for WMC OBEX Functions

Program 2 shows how the descriptors for a WMC OBEX function should be coded. The WMC Protocol Library uses some of these descriptors to configure itself; others are ignored. However all the descriptors are needed for a standard-compliant OBEX function.

The requirements of the WMC Protocol Library for an OBEX Function are:

1. A Communication class interface with subclass 0x0B.
2. A Data class interface with two alternate settings. Alternate Setting 0 must not have any endpoints.
3. One Bulk IN endpoint, associated with Alternate Setting 1 of the Data class interface.
4. One Bulk OUT endpoint, associated with Alternate Setting 1 of the Data class interface.
5. A CDC Union descriptor, which mentions the Communication class interface and the Data class interface.

MCCI test cases always include substantially the same descriptor set shown in Program 2. Therefore, MCCI recommends that your descriptor set always include the CDC Header descriptor and the OBEX Functional Descriptor, as shown.

In the example, the USB device hardware allows you to have an IN endpoint and an OUT endpoint with the same endpoint number. Although this is legal according to [USBCORE], many device controllers do not support this feature. You should be careful to verify that the endpoint numbers you give to USBRC will work with your hardware.

Program 2. Descriptor Fragment for OBEX Function

```
##################################
#       OBEX Function            #
##################################

#
# The Comm-class control interface
#
interface 3
        {
        class    0x02   #comm class
        subclass 0x0B   #WMC OBEX
        protocol 0x00   #no protocol
        name     S_TA2  # string

        # here are the class descriptors.
        private-descriptors
                {
```

```
                # header
                raw     {
                        0x24    # interface
                        0       # functional descriptor
                        word(0x110)     # CDC version 1.1
                        };

                # obex header
                raw     {
                        0x24    # interface
                        0x15    # OBEX functional descriptor
                        word(0x100)     # OBEX version 1.0
                        };

                # union descriptor.
                raw     {
                        0x24    # interface
                        6       # ``union''
                        % index of primary interface % 3
                        % index of data interface 1 %  4
                        };
                }
        % no endpoints % ;
        }

# OBEX Function, data interface #1
interface 4
        {
        alternate-setting 0
                class    0x0A   #data class
                subclass 0x00   #
                protocol 0x00   #
                name    S_TA2_DATA_DISABLED
                ;

        alternate-setting 1
                class    0x0A   #data class
                subclass 0x00   #
                protocol 0x00   #
                name    S_TA2_DATA_1
                endpoints
                        bulk in
                        bulk out
                        ;
        }
```

2.1.3 Descriptors for WMC Device Management Functions

Program 3 shows how the descriptors for a WMC Device Management Function should be coded. The WMC Protocol Library uses some of these descriptors to configure itself; others are ignored. However all the descriptors are needed for a standard-compliant Device Management Function.

The requirements of the WMC Protocol Library for a Device Management Function are:

1. A Communication class interface with subclass 0x09.

2. One Interrupt IN endpoint, associated with the Communication class interface. The recommended endpoint wMaxPacketSize is 8 bytes or larger.

3. A Device Management functional descriptor, indicating a buffer size of 2048 bytes.

MCCI test cases always include substantially the same descriptor set shown in Program 3. Therefore, MCCI recommends that your descriptor set always include the CDC Header descriptor, as shown.

Program 3. Descriptor Fragment for OBEX Function

```
####################################
#  WMC Device Management Function  #
####################################

#
# The Comm-class control interface
#
interface 5
        {
        class    0x02   #comm class
        subclass 0x09   #Device management
        protocol 0x01   #common AT commands
        name     S_TA3  # string

        # here are the class descriptors.
        private-descriptors
                {
                # header
                raw     {
                        0x24        # interface
                        0           # functional descriptor
                        word(0x110) # CDC version 1.1
                        };

                # Device Management header
                raw     {
                        0x24        # interface
                        0x14        # device mgmt functional
descriptor
                        word(0x100) # device mgt version 1.0
                        word(2048)  # max buffer size
                        };

                }

        # here is the notification endpoint
        endpoints
                interrupt in packet-size 16
                        polling-interval 16
                ;
        }
```

2.2 Client Instance Initialization

Client 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 them names.

Afterwards the DataPump has been initialized, the target operating system must discover the available modem instances, and must create client instances. Each client instance registers with a protocol instance. All communication from Client to Protocol is accomplished using a downcall I/O-control mechanism, known as an **IOCTL**, defined by the DataPump and implemented by the Protocol (see section 5). 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 the host PC controls USB device firmware, 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 I/O-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 (see section 4). 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(
        pDataPumpRootHeader,
        pLastFunctionObject,
        pPattern)
```

pPattern specifies the kind of Protocol object to find. It should be:

`"modem.*.fn.mcci.com"`	to match objects with CDC Modem semantics
`"obex.*.fn.mcci.com"`	to match objects with WMC OBEX semantics
`"devmgmt.*.fn.mcci.com"`	to match objects with WMC Device Management semantics

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.

Please also refer to *[DPOVERVIEW],* `usbkern/doc/OVERVIEW-appinit.txt` and `OVERVIEW-objects.txt` in DataPump Professional and Standard source installation.

Applications wishing to make use of the Protocol library should

- include the header file `wmcclientlib.h`
- link with library `protowmc`.

3. 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. For more details refer *[DPREF].*

3.1 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.

The application initialization code uses this structure 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_Modem`
`sInterfaceSubClass:`	`USB_bInterfaceSubClass_CommACM`, `USB_bInterfaceSubClass_CommOBEX`, or `USB_bInterfaceSubClass_CommDEVMGMT`
`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`	Flags that control the probing of multiple instances. USBPUMP_PROTOCOL_INIT_FLAG_MATCH_IFCNUM USBPUMP_PROTOCOL_INIT_FLAG_IGNORE_STRINGS USBPUMP_PROTOCOL_INIT_FLAG_SEPARATE_ALTSETS USBPUMP_PROTOCOL_INIT_FLAG_NO_BACK_MATCH USBPUMP_PROTOCOL_INIT_FLAG_AUTO_ADD
`pProbeFunction:`	Normally `WmcTaProtocolProbe`. This is an MCCI-supplied `USBPUMP_PROTOCOL_PROBE_FN` function.
`pCreateFunction:`	Normally `WmcTA_ProtocolCreate` – this function will create the appropriate set of protocol objects to implement the appropriate class-level behavior.
`pQualifyAddInterfaceFunction`	Optional add-instance qualifier function. If this function is available and returns TRUE then `pAddInterfaceFunction` will be called to add the interface
`pAddInterfaceFunction`	Optional function for adding instance.
`pOptionalInfo:`	Pointer to the `UPROTO_WMCTA_CONFIG` structure that provides TA-specific information for this TA (see section 3.2). This is normally an MCCI-supplied table, and is either of `(VOID *) &gk_WmcSubClass_Modem_TaConfig`, `(VOID *) &gk_WmcSubClass_Modem_TaConfig`, or `(VOID *) &gk_WmcSubClass_Modem_TaConfig`. These tables are provided by MCCI. The `(VOID *)` is needed for type compatibility. The entry chosen must match the kind of TA being configured (Modem, OBEX, or Device Management).

3.2 UPROTO_WMCTA_CONFIG

This structure is pointed to by the `USBPUMP_PROTOCOL_INIT_NODE`. It is preferably initialized using the macro `UPROTO_WMCTA_CONFIG_INIT_V2` 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:

`SizeSubClass:`	how large is the structure? Sets the size of the subclass structure.

`TagSubClass:`	what is the tag? Sets the tag for the subclass structure.
`pName:`	what's this TA's name? Sets the name to be given to the subclass structure.
`PUpcallTable:`	what's the upcall switch?
`ControlBufferSize:`	how long for setup packets? Sets the maximum SETUP data phase length; a buffer of specified size is allocated.
`InRingbufSize:`	how big are the encaps command In ring buffer? Sets the ring buffer size for control data being sent towards the host.
`OutRingbufSize:`	how big are the encaps command Out ring buffer? Sets the ring buffer size for control data being received from the host.
`pSubClassConfig`	optional pointer to sub-class configuration

4. Edge-IOCTL (Upcall) services

The following section describes the services the Client must provide to the Protocol through the Edge-IOCTL function given when initializing the Client object using `UsbPumpObject_Init()`.

4.1 Edge IOCTL function

```
Type name :     USBPUMP_OBJECT_IOCTL_FN

Prototype :     USBPUMP_IOCTL_RESULT My_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 : usbpumpobject.h
```

4.2 Generic Edge IOCTLs

4.2.1 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 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

4.2.2 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 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, in general, to prevent, deactivation.

4.2.3 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 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 a Client specific event info. See "ueventnode.h" for information, or [DPREF]
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).

4.3 WMC-specific Edge IOCTLs

The IOCTLs in this section are of two kinds: IOCTLs that are specific to modem emulation and used only by the CDC Modem protocol, and IOCTLs that are used by all WMC Protocol implementations to communicate data plane activation/deactivation.

Table 2. Common IN parameter fields for all Edge Modem IOCTLs

Field	Description
USBPUMP_OBJECT_HEADER *pObject	Pointer to the UPROTO object in question
VOID *pClientContext	Pointer to Client context, supplied by client when it connected to the UPROTO.

Table 3. Common OUT parameter fields for all Edge Modem IOCTLs

Field	Description
BOOL fReject	Set TRUE to reject request. Field initialized to FALSE by Protocol. If set TRUE by the client, then the edge driver would like the USB core to reject the request, if possible. Note, though, that due to hardware constraints it

Field	Description
	may not be possible to fail the request.
Note	If the client uses normal DataPump IOCTL processing disciplines, then the out parameter structure will be initialized with the same values as the in parameter

4.3.1 Modem Emulation IOCTLs

4.3.1.1 Edge Modem Set Break

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_ARG *
Field how	Instance of USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_HOW. Kind of break request – one of USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_OFF, USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_ON, USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_TIMED
Field uMillisec	How many milliseconds the break signal should be, if How is USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_TIMED
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_SET_BREAK_ARG *
Description	This IOCTL is sent from Protocol to Client whenever the host changes the break state. If the client doesn't implement this IOCTL, the break state will be tracked by the Protocol Library, but no other action will be taken.

4.3.1.2 Edge Modem Set Control Line State

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_SET_CONTROL_LINE_STATE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_SET_CONTROL_LINE_STATE_ARG *
Field NewValue	8 bit mask indicating the new state of DTR and RTS, as given by the CDC ACM spec [USBCDC] (or in "usbcdc11.h").
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_SET_CONTROL_LINE_STATE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever the host modifies the state of DTR or RTS. The Client should do whatever makes sense to communicate the state change to the higher level.

4.3.1.3 Edge Modem Set Line Coding

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_SET_LINE_CODING
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_SET_LINE_CODING_ARG *
Field pCoding	Pointer to USB_Comm_LINE_CODING structure. The actual coding values: baudrate, bits per char, parity, stop bits.
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_SET_LINE_CODING_ARG *
Description	The host uses a SET_LINE_CODING request to change the baud rate, bits per character, and parity of the connection. This IOCTL is sent from Protocol to Client after basic parameters of the SET_LINE_CODING operation have been verified. The Client has the additional opportunity to reject the operation, by setting fReject TRUE. If this IOCTL is not implemented by the client, any SET_LINE_CODING operation that is legal within the ACM spec is permitted by the base code. In any case, the current values are stored in the WMCSUBCLASS_MODEM structure.

4.3.1.4 Edge Modem Set Acm Feature

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_SET_ACM_FEATURE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_SET_ACM_FEATURE_ARG *
Field FeatureIndex	duration (if How == .._TIMED)
Field FeatureValue	the feature value setting. The value that is being changed.
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_SET_ACM_FEATURE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever host changes the value of a feature.

4.3.2 WMC Protocol Activation/Deactivation IOCTLs

4.3.2.1 Edge Modem Start Plane

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_START_PLANE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_START_PLANE_ARG *

Field iPlane	Instance of USBPUMP_MODEM_PLANE_SELECTOR. Plane selector to select the plane on a multi-data-plane device.
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_START_PLANE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever host starts a plane.

4.3.2.2 Edge Modem Stop Plane

IOCTL code	USBPUMP_IOCTL_EDGE_MODEM_STOP_PLANE
In parameter structure	CONST USBPUMP_IOCTL_EDGE_MODEM_STOP_PLANE_ARG *
Field iPlane	Instance of USBPUMP_MODEM_PLANE_SELECTOR. Plane selector to select the plane on a multi-data-plane device.
Out parameter	USBPUMP_IOCTL_EDGE_MODEM_STOP_PLANE_ARG *
Description	This IOCTL is sent from Protocol to Client whenever host stops a plane.

5. IOCTL (Downcall) Services

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

5.1 Get Plane Info

```
Prototype :

USBPUMP_IOCTL_RESULT WmcClientLib_GetPlaneInfo (
        USBPUMP_OBJECT_HEADER *                    pIoObject,
        BOOL *                                     pfHasControlPlane,
        UINT *                                     pnDataPlanes
        );

Header-file : wmcclientlib.h

Underlying IOCTL :  USBPUMP_IOCTL_MODEM_GET_PLANE_INFO
```

This IOCTL is sent from an OS-specific driver in order to obtain information about the configuration of the specified abstract modem. This function is usually used only during initialization, because the configuration of a given object doesn’t change after initialization

The parameters are:

`pIoObject`	This is a pointer to Protocol instance object.

`pfHasControPlane`	Pointer to a client-supplied BOOL variable. This will be set TRUE if the protocol instance specified by `pObject` has a separate control-oriented data plane (for example, for transporting encapsulated commands)
`pnDataPlanes`	Pointer to a client-supplied integer variable. This will be set to the number of distinct data planes supplied by `pObject`.

5.2 Queue In

```
Prototype :

USBPUMP_IOCTL_RESULT WmcClientLib_QueueIn(
        USBPUMP_OBJECT_HEADER *                 pIoObject,
        UBUFQE *                                pListHead,
        USBPUMP_MODEM_PLANE_SELECTOR            iPlane
        );

Header-file : wmcclientlib.h
```

This function is used by Client to provide a buffer for the host to write data to. The buffer is queued to the specified plane of the abstract modem.

This API is similar to the DataPump core UsbPumpPipe_QueueList() API. It differs in the following ways:

- The uqe_pipe pointer in the UBUFQE is overwritten by this call
- This call will be accepted even when the underlying transport is not available (for example, when no USB cable is plugged in).
- UBUFQEs will only be completed and returned to the client when they are actually filled with data from the host.

The parameters are:

`pIoObject`	This is a pointer to Protocol instance object.
`pListHead`	Pointer to the head of a circularly linked list of UBUFQEs. These UBUFQEs are submitted as a unit to the receive side of the specified data plane. Each UBUFQE points to a buffer that is filled and completed indpendently, in sequence.
`iPlane`	The plane index. This is either USBPUMP_MODEM_PLANE_ENCAPS or USBPUMP_MODEM_PLANE_DATA + (i), to select the plane on a multi-data-plane device. (Since all current implementations are single-plane devices, this is usually simplified to USBPUMP_MODEM_PLANE_DATA.)

Clients using the Abstract Modem API must be aware that the underlying transport is a *character-oriented* transport, not a *record-oriented* transport. This means that clients cannot assume that there is any relationship between USB transfers (which will fill a single UBUFQE) and logical record boundaries. This is not generally a problem for CDC modem and Device Management clients, but it may cause surprises for OBEX implementations.

5.3 Queue Out

```
Prototype :

USBPUMP_IOCTL_RESULT WmcClientLib_QueueOut(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        UBUFQE *                                     pListHead,
        USBPUMP_MODEM_PLANE_SELECTOR                 iPlane
        );

Header-file : wmcclientlib.h
```

This function is used by Client to queue data to be sent to the host over a specific data plane of the function.

The parameters are:

`pIoObject`	This is a pointer to Protocol instance object.
`pListHead`	Pointer to the head of a circularly linked list of UBUFQEs. These UBUFQEs are submitted as a unit to the receive side of the specified data plane. Each UBUFQE points to a buffer that is filled and completed independently, in sequence. The exact semantics are controlled by the uqe_flags field of each UBUFQE.
`iPlane`	The plane index. This is either USBPUMP_MODEM_PLANE_ENCAPS or USBPUMP_MODEM_PLANE_DATA + (i), to select the plane on a multi-data-plane device. (Since all current implementations are single-plane devices, this is usually simplified to USBPUMP_MODEM_PLANE_DATA.

This API is similar to the DataPump core UsbPumpPipe_QueueList() API. It differs in the following ways:

- The uqe_pipe pointer in the UBUFQE is overwritten by this call

However, if a UBUFQE is sent towards the host when the host transport is not available, then it will be competed with an error.

5.4 Set Uart State

```
Prototype :

USBPUMP_IOCTL_RESULT WmcClientLib_SetUartState(
        USBPUMP_OBJECT_HEADER *                      pIoObject,
        UINT                                         ModemMask,
        UINT                                         ModemValue
        );
```

```
Header-file : wmcclientlib.h
```

This function is used by Client to signal a change in the simulated modem-status lines for a given Abstract Modem. If the underlying Protocol is not able to transport modem status lines to the host, then this call will be ignored.

The parameters are:

`pIoObject` This is a pointer to Protocol instance object.

`ModemMask` A bit mask that specifies the specific Abstract Modem status lines that might have changed. If a given bit is 0, then the corresponding bit in `ModemValue` should be ignored. If a given bit is 1, then the corresponding bit in `ModemValue` gives the new value for the bit.

`ModemValue` A bit mask that specifies the current values for the Abstract Modem status, as given in Table 4

Table 4. Abstract Modem Serial Status

Bit	RS-232	Name	Notes
0	DCD	USBPUMP_MODEM_UART_DCD	Data Carrier Detect
1	DSR	USBPUMP_MODEM_UART_DSR	Data Set Ready
2	BREAK	USBPUMP_MODEM_UART_BREAK	Break – this is a transient signal that should be set when the break starts, and cleared when the break clears.
3	RI	USBPUMP_MODEM_UART_RI	Ring Indicator
4	Framing	USBPUMP_MODEM_UART_FE	Framing error – this should be set whenever a character with a framing error is to be sent towards the host. This is rarely used, because of the timing uncertainties. The underlying protocol will automatically clear this bit after it has been passed to the host.
5	Parity	USBPUMP_MODEM_UART_PE	Parity error – this should be set whenever a character with a parity error is to be sent towards the host. This is rarely used, because of the timing uncertainties. The underlying protocol will automatically clear this bit after it has been passed to the host.
6	Overrun	USBPUMP_MODEM_UART_OE	Overrun error – this should be set whenever a hardware overrun occurs for data being received for transmission to the host. This is very rarely used, not because of the timing uncertainties, but because this kind of error is only associated with unreliable UART links. The underlying protocol will automatically clear this bit after it has been passed to the host.

Bit	RS-232	Name	Notes
7	CTS	USBPUMP_MODEM_UART_CTS	Clear to send. Note that this bit cannot be transported by CDC ACM modems, but might be transported by other protocols that implement the Abstract Modem API (such as the MCCI VSP Protocol)

6. Other Considerations

6.1 Serial Numbers

[USBCORE] requires that USB devices either have unique serial numbers, or no serial number at all. 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. See the description of `USBPUMP_IOCTL_GET_SERIALNO` in [DPREF] or [DPIOCTL].

Some AT command interpreter APIs (notably ATI9 in the MCCI reference interpreter) need to return a serial number. For convenience, these implementations can also use `USBPUMP_IOCTL_GET_SERIALNO`.

6.2 Makefile Issues

You must include the library **`protowmc.a`** in the list of input libraries to be searched when linking your application. (This library is named "**`protowmc.lib`**" if you are using Microsoft C). If you are using the MCCI build system, do this by adding the following line to your UsbMakefile.inc:

```
LIBRARY := ${protowmc_a}
```

7. Demo applications

The DataPump Professional and Standard installations contain several demo applications in `usbkern/app/wmcdemo/`... and `usbkern/proto/wmc/applib` that can be used as reference on how to use the Modem protocol.

See especially `usbkern/app/wmcdemo/wmcdemo/`, which demonstrates combining a WMC CDC modem, a WMC OBEX function, and a WMC Device Management function into a single device.

Also see `usbkern/app/wmcdemo/mscacmdemo/`, which demonstrates a simple modem + mass storage combination.