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

MCCI USBSERI API

Engineering Report 950386
Rev. A
Date: 7/24/2006

Copyright © 2006
All rights reserved

PROPRIETARY NOTICE AND DISCLAIMER

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

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

MCCI, TrueCard, TrueTask, MCCI Catena, and MCCI USB DataPump are registered trademarks of Moore Computer Consultants, Inc.

MCCI Instant RS-232, MCCI Wombat and InstallRight Pro are trademarks of Moore Computer Consultants, Inc.

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

Copyright © 2006 by Moore Computer Consultants, Incorporated

Document Release History

07/24/2006	Rev. A	Original release

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This document describes and specifies the USBSERI API to the MCCI DataPump WMC protocol library. The purpose of the USBSERI API addition is to simplify the WMC protocol client interface, also to lower the integration effort.

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

1.2 Referenced Documents

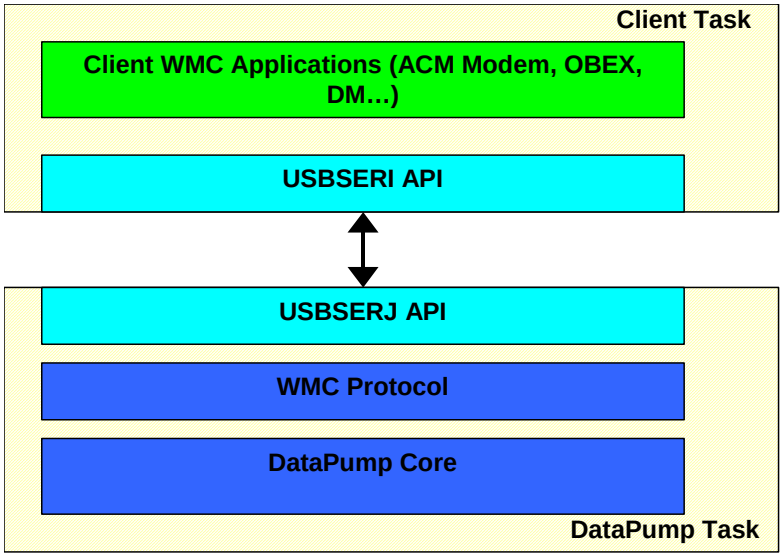
[MCCIWMC] MCCI DataPump WMC protocol users guide, MCCI Engineering Report 950255 revision A

1.3 Architecture

There are two layers of USB APIs being added on top of the existing WMC protocol. A lower USBSERJ API locates on top of the WMC protocol to serve as the interface communicating with the upper client task. Another upper USBSERI API locates at the bottom of the client task to interact with WMC protocol through USBSERJ API.

Figure 1 shows the USBSERI and USBSERJ APIs in the system architecture.

Figure 1. USBSERI/USBSERJ block diagram



Notice that since the USBSERJ layer handles internal communications between the WMC protocol and USBSERI API to the client applications, it is not going to be exposed to the client code thus will not be covered in this document.

1.4 Overview

The USBSERI API contains all necessary interfaces communicating with the host WMC functions. The API is divided into four major function categories: Basic operations, status control, event callback and abstract OS-specific events.

Basic operations contain basic device open, close and IO functions. Status control functions work with flow and WMC port status control. Event callback contains callback function with status codes for outgoing (from USBSERI to client) events. A set of standardized modem status control codes inherited from the MCCI DataPump WMC protocol is applied to the event system.

The abstract OS-specific events are for internal USBSERI and USBSERJ task control that will not be exposed for external use. They serve as the inter-task synchronization process lock to make sure internal operations happen in desired order. The way to implement the set of event functions is OS-dependent.

2 Data types

2.1 Modem status codes

The following signals define the simulated "modem status" bits that are communicated to the host over the USB UART. These lines might not be supported on all kinds of USB transports; for example, none of these are supported on a "diagnostic" or "device management" port; and CTS is not supported on ACM modems.

Bits 0..6 deliberately match the meanings assigned by the CDC ACM spec. Bit 7 is deliberately reserved and will not be used. Bits 8..31 are local extensions to the spec, for signals that we might need to transport, notably CTS.

```
#define   USBSERI_MODEM_STATE_DCD      0x01u   /* DCD line to host */
#define   USBSERI_MODEM_STATE_DSR      0x02u   /* DSR line to host */
#define   USBSERI_MODEM_STATE_BREAK    0x04u   /* BREAK status to host */
#define   USBSERI_MODEM_STATE_RI       0x08u   /* RI line to host */
#define   USBSERI_MODEM_STATE_FE       0x10u   /* Framing Error status to host */
#define   USBSERI_MODEM_STATE_PE       0x20u   /* Parity Error status to host */
#define   USBSERI_MODEM_STATE_OE       0x40u   /* Overrun Error status to host */
#define   USBSERI_MODEM_STATE_RSV7     0x80u   /* <<reserved>> */
#define   USBSERI_MODEM_STATE_CTS      0x100u  /* CTS line to host */
#define   USBSERI_MODEM_STATE_MASK_ACM 0x7Fu   /* a mask of the ACM-defined bits */
```

Similarly, we have DTR and RTS lines in the simulated "modem control" bits that are communicated from the host. Again, not all USB transport protocols will support these signals. If they are not supported, the USB UART will assert DTR and RTS as soon as the USB UART function is activated by the host, and will de-assert when the function is deactivated.

```
#define   USBSERI_MODEM_CONTROL_DTR    0x01u   /* DTR line from host */
#define   USBSERI_MODEM_CONTROL_RTS    0x02u   /* RTS line from host */
```

2.2 USBSERI_EVENT_CODE_MAP structure

The `USBSERI_EVENT_CODE_MAP` structure is defined to standardize outgoing event communication from USBSERI API to the client supplied event function. The event codes are brought in from the outside world using a struct that must be filled in and supplied by the client. Each value may be -1 indicating that no event of the kind is to be actually sent, otherwise a value in 0 to 255 giving the desired event code for the event.

A client function of type `USB_SerialEventHandler_t` (see section 2.3) will need to be defined and registered to receive the event with the specified `USBSERI_EVENT_CODE_MAP`.

```
typedef struct USBSERI_EVENT_CODE_MAP
          {
          int RxDataAvailable;
          int DtrChangeHigh;
```

```
int DtrChangeLow;
int EscapeDataConnect;
int TxNotFull;
int BreakReceived;
int StartPlane;
int StopPlane;
} USBSERI_EVENT_CODE_MAP;
```

There are eight events being covered in the event map; see description in Table 1 for details.

Table 1 USBSERI_EVENT_CODE_MAP members

Member	Description
`RxDataAvailable`	Indicates data received from host.
`DtrChangeHigh`	Indicates DTR status has been changed to high.
`DtrChangeLow`	Indicates DTR status has been changed to low.
`EscapeDataConnect`	Indicates escape signal has been received.
`TxNotFull`	Indicates Tx pipe not full so the client can move data to outgoing pipe.
`BreakReceived`	Indicates break signal has been received.
`StartPlane`	Indicates a specific control / data plane is now started.
`StopPlane`	Indicates a specific control / data plane is now stopped.

2.3 Abstract event structure

The abstract event structure is allocated for the event control block of abstract OS-specific events. The size of the control block is defined in `USBSERI_OS_ABSTRACT_EVENT_SIZE`.

```
#define   USBSERI_OS_ABSTRACT_EVENT_SIZE   16
```

The event control block size is determined by specific implementation chosen on target OS. Thus it will usually require a pre-definition in the buildset.var file to overwrite the default event size given. Adding definition of the line below will modify the control block size to a user-defined value.

```
CFLAGS_PORT_OS_[Target OS] += -DUSBSERI_OS_ABSTRACT_EVENT_SIZE=[Value]
```

The tokens for [Target OS] and [Value] shall be replaced by target OS abbreviation and control block size respectively.

The abstract event structure allocates a size of `USBSERI_OS_ABSTRACT_EVENT_SIZE` to be filled in by the actual event control structure.

```
typedef struct USBSERI_OS_ABSTRACT_EVENT
          {
          void*Filler[USBSERI_OS_ABSTRACT_EVENT_SIZE / sizeof(void*)];
          } USBSERI_OS_ABSTRACT_EVENT;
```

3 Basic operations API

3.1 USBSERI Open

Client calls USB_OpenDevice to open a WMC port. A port instantce number `uInstance` shall be supplied when calling the USBSERI open API. The open API will return a port handle if the open action succeeds, or NULL if it fails.

The port handle will be used in the subsequent operations to the port as the identifier.

```
void *
USB_OpenDevice(
          unsigned uInstance
          );
```

3.2 USBSERI Close

Client calls USB_CloseDevice to close a WMC port. A port handle to the port to be closed shall be supplied to `hPort`.

```
void
USB_CloseDevice(
          void *hPort
          );
```

3.3 USBSERI Read

Client calls `USB_Read` to read from a WMC port. A port handle to the port to read from shall be supplied to `hPort`.

This routine receives data from the USBSERI internal buffers into a user-supplied buffer, designated by `pBuffer` and `nBuffer`.

If no data is available, no bytes will be copied, and the result will be zero. If `nBuffer` is less than the data available in the USB buffers, only copy the first `nBuffer` bytes. If zero is returned, the caller should wait for a while before calling this again. (For example, wait for a call back to the registered event notification function, with event code `RxDataAvailable`).

As data is received, it is removed from the internal buffers. As UBUFQEs become empty, they are returned to the DataPump to be re-filled.

```
unsigned short
USB_Read(
          void *          hPort,
          unsigned char *  pBuffer,
          unsigned short   nBuffer
          );
```

3.4 USBSERI Write

Client calls USB_Write to write to a WMC port. A port handle to the port to write to shall be supplied to `hPort`.

This routine copies the relevant data from the user-supplied buffer to the USB subsystem internal buffers for that port. If maxData is bigger than the remaining space in the internal buffers, only copy as much data as can be accommodated in the existing space. Therefore, if the buffers are full, this routine will copy no data, and will return zero. If zero is returned, the caller should wait for a while before calling this again. (For example, wait for a call back to the event handler function with event code TxNotFull.

```
UINT16
USB_Write(
          void *      hPort,
          const UINT8 *dataPtr,
          UINT16      dataSize
          );
```

4 Status control API

4.1 Rx flow control

Client calls `USB_EnableRxDataFlow` to control flow from the receiving end of a WMC port. A port handle to the port to write to shall be supplied to `hPort`.

If `fEnable` is TRUE, the event of `RxDataAvailable` is sent when data arrives. If `fEnable` is FALSE, the event of `RxDataAvailable` is not sent when data arrives.

```
int
USB_EnableRxDataFlow(
          void *  hPort,
          int fEnable
          );
```

4.2 Control lines query

Client calls `USB_QueryDtrRts` to get current DTR and RTS values of a WMC port. A port handle to the port to write to shall be supplied to `hPort`.

The DTR and RTS value is returned by the call, see section 2.1 for the bit masks of `USBSERI_MODEM_CONTROL_DTR` and `USBSERI_MODEM_CONTROL_RTS`.

```
unsigned
USB_QueryDtrRts(
          void *hPort
          );
```

4.3 Tx buffer query

Client calls `USB_QueryFreeTxSpace` to get space available in the TX buffers from a WMC port. A port handle to the port to write to shall be supplied to `hPort`.

Number of bytes available in the Tx buffer is returned with the call to `USB_QueryFreeTxSpace`.

```
unsigned short
USB_QueryFreeTxSpace(
          void *  hPort
          );
```

4.4 Modem status line control

Client calls `USB_SetClearModemStatus` to change modem status lines of a WMC port. A port handle to the port to write to shall be supplied to `hPort`.

This routine modifies the modem status lines of the port according to the value of the `uBitsToChange` and `uNewBits` parameters. For each bit that's set in `uBitsToChange`, the corresponding bit in `uNewBits` is copied into the modem's simulated modem status register. If any values change, then a status notification is forwarded to the host. See section 2.1 for the definition of modem status codes to supply to `uBitsToChange` and `uNewBits` variables.

```
void USB_SetClearModemStatus(
          void *hPort,
          UINT32 uBitsToChange,
          UINT32 uNewBits
          );
```

5 Event callback API

5.1 Event callback function definition

The event callback function from USBSERI API shall be defined as `USB_SerialEventHandler_t` type. The callback function of `USB_SerialEventHandler_t` type will reside in client code to handle events from USBSERI API.

```
typedef void
USB_SerialEventHandler_t(
          void *           /*hPort*/,
          void *           /*pClientContext*/,
          unsigned char    /*event code */,
          void *           /*pEventData (optional)*/
          );
```

5.2 Event callback function registration

Client calls `USB_RegisterEventNotify` function to register event callback function defined by the implementation of `USB_SerialEventHandler_t` prototype. A port handle to the port to write to shall be supplied to `hPort`. A `pClientContext` of a client instance shall also be supplied.

The function `pHandler` is registered as the event handling function for the given WMC port. It will be called with the client context specified by the `pClientContext` value. `pEventCodeMap` specifies the numerical value of the event codes. See section 2.2 for the definition of `USBSERI_EVENT_CODE_MAP` structure.

```
int
USB_RegisterEventNotify(
          void *                          hPort,
          USB_SerialEventHandler_t *      pHandler,
          void *                          pClientContext,
          const USBSERI_EVENT_CODE_MAP *  pEventCodeMap
          );
```

6 Abstract OS-specific events

6.1 Event initialization

USBSERI API uses `Usbseri_OS_InitializeEvent` to initialize a event to wait on another task. The pointer to abstract event control structure `pEvent` will be passed in along with the event size `sizeEvent`. An actual `pEvent` will be filled in once the event is successfully initialized. This `pEvent` will be used as a reference in all the abstract event functions.

Notice that this function along with all other abstract events shall be implemented outside of USBSERI API (usually by customer) due to its OS and implementation dependent nature.

```
void
Usbseri_OS_InitializeEvent(
          USBSERI_OS_ABSTRACT_EVENT *pEvent,
          unsigned sizeEvent
          );
```

6.2 Event de-initialization

USBSERI API uses `Usbseri_OS_DeinitializeEvent` to de-initialize a event to wait on another task. The event of `pEvent` will be de-initialized after the call to this function.

Notice that this function along with all other abstract events shall be implemented outside of USBSERI API (usually by customer) due to its OS and implementation dependent nature.

```
void
Usbseri_OS_DeinitializeEvent(
          USBSERI_OS_ABSTRACT_EVENT *pEvent
          );
```

6.3 Waiting on event

USBSERI API uses `Usbseri_OS_WaitForEvent` to wait on arrival of a specific event before it proceeds to the next statement.

Notice that this function along with all other abstract events shall be implemented outside of USBSERI API (usually by customer) due to its OS and implementation dependent nature.

```
int
Usbseri_OS_WaitForEvent(

          USBSERI_OS_ABSTRACT_EVENT *pEvent
          );
```

6.4 Setting event

USBSERI API uses `Usbseri_OS_SetEvent` to flag the arrival of a specific event, to remove the process lock set by `Usbseri_OS_WaitForEvent` to wait for the event arrival.

Notice that this function along with all other abstract events shall be implemented outside of USBSERI API (usually by customer) due to its OS and implementation dependent nature.

```
void
Usbseri_OS_SetEvent(
```

```
        USBSERI_OS_ABSTRACT_EVENT *pEvent
        );
```

7 Sequence diagrams

Sequence diagrams for the USBSERI API are shown below to illustrate the major stages of DataPump and client application initialization, i.e., DataPump initialization, client ports instances association through USBSERI API, and ports activation by host. Interaction with Nucleus RTOS is included as an example of OS dependent operations, which shall be replaced with corresponding targeted OS events and functions.

7.1 DataPump and protocol clients initialization

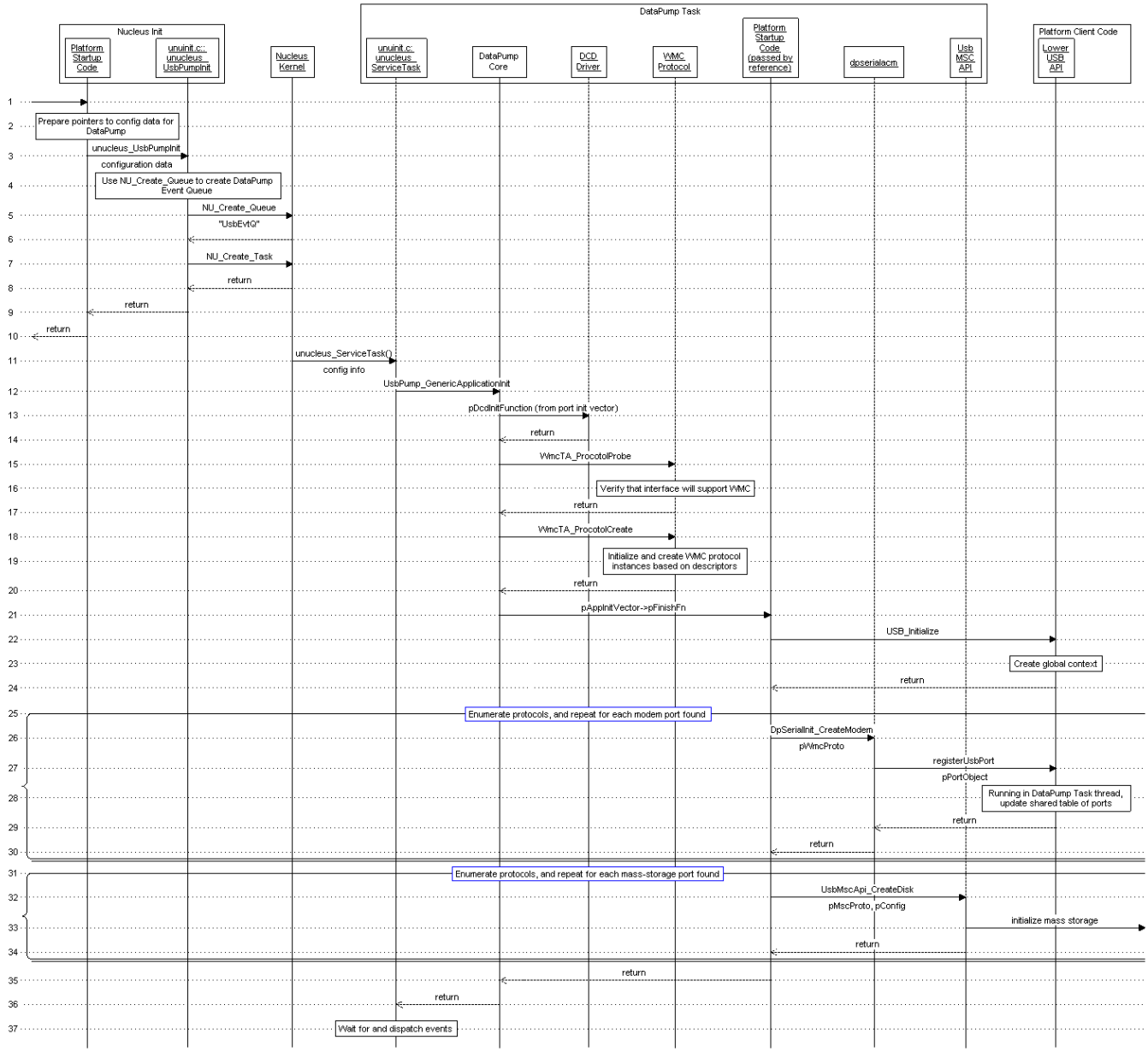
To startup USB function, DataPump task is first initialized under target platform and OS environment. As shown on the left side of Figure 2, Nucleus OS, as an example, is responsible for providing OS-related support such as event queue, interrupt and DataPump task services (line 1 to line 10).

After DataPump task is created, it then initializes DCD (Device Controller Driver) layer from the port initialization vector. Then DataPump verifies WMC protocol and then creates WMC protocol instances according to the TAs defined by descriptors (line 13 to line 20).

Then in the platform startup code when creating function clients, DataPump first calls USBSERI API initialization function once, to initialize USBSERI API so WMC port objects can later be created (see line 22). It then does the actual creation of the WMC port objects by calling `CreateModem` API function once for each WMC port instance found, by looping through `UsbPumpObject_EnumerateMatchingNames` (line 25 to line 30).

Other protocol clients such as mass storage are also created during the above matching loop (line 31 to line 34).

Figure 2. Sequence diagram for DataPump and protocol instances initialization



7.2 Client ports instance association through USBSERI API

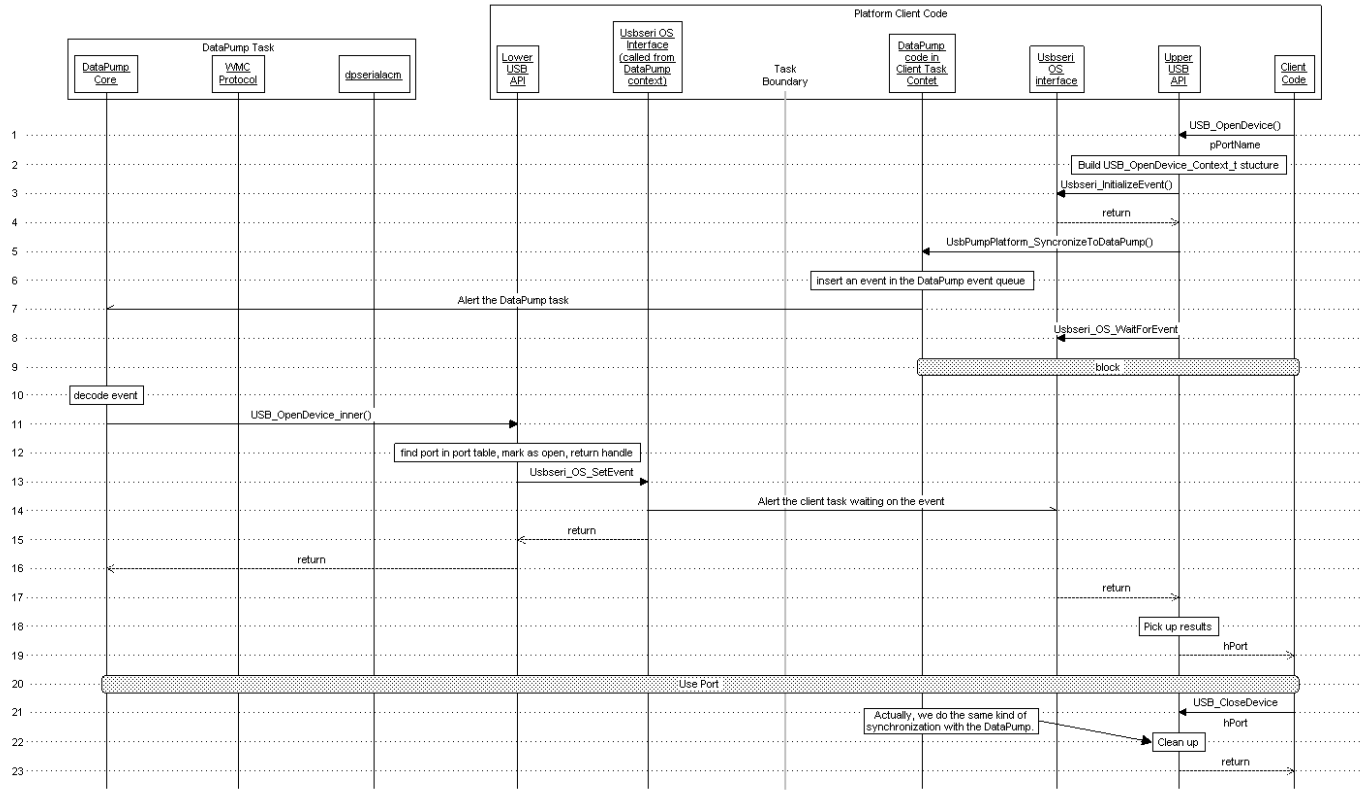
DataPump WMC ports instances interact with UART or modem objects in the client code through USBSERI API. To associate each WMC protocol instance created with a specified UART or modem object from the client application, an OpenDevice action is required to pass back the WMC port instance as a reference in later port operations.

First a call to `USB_OpenDevice` is initiated in client application, with the sequence number of WMC port instance, which the client UART object would like to associate with. Then a return

structure of `USB_OpenDevice_Context_t` is created to contain the opened handle to WMC port instance, to be passed back to client application (line 1 to line 2).

An event for OpenDevice is then created and registering to DataPump. After DataPump task receives the event, USBSERJ API will then process the rest of the OpenDevice operations inside DataPump and then pass back the WMC port handle to client application (line 3 to line 19).

Figure 3. Sequence diagram for client ports instance association



7.3 Host opens WMC ports

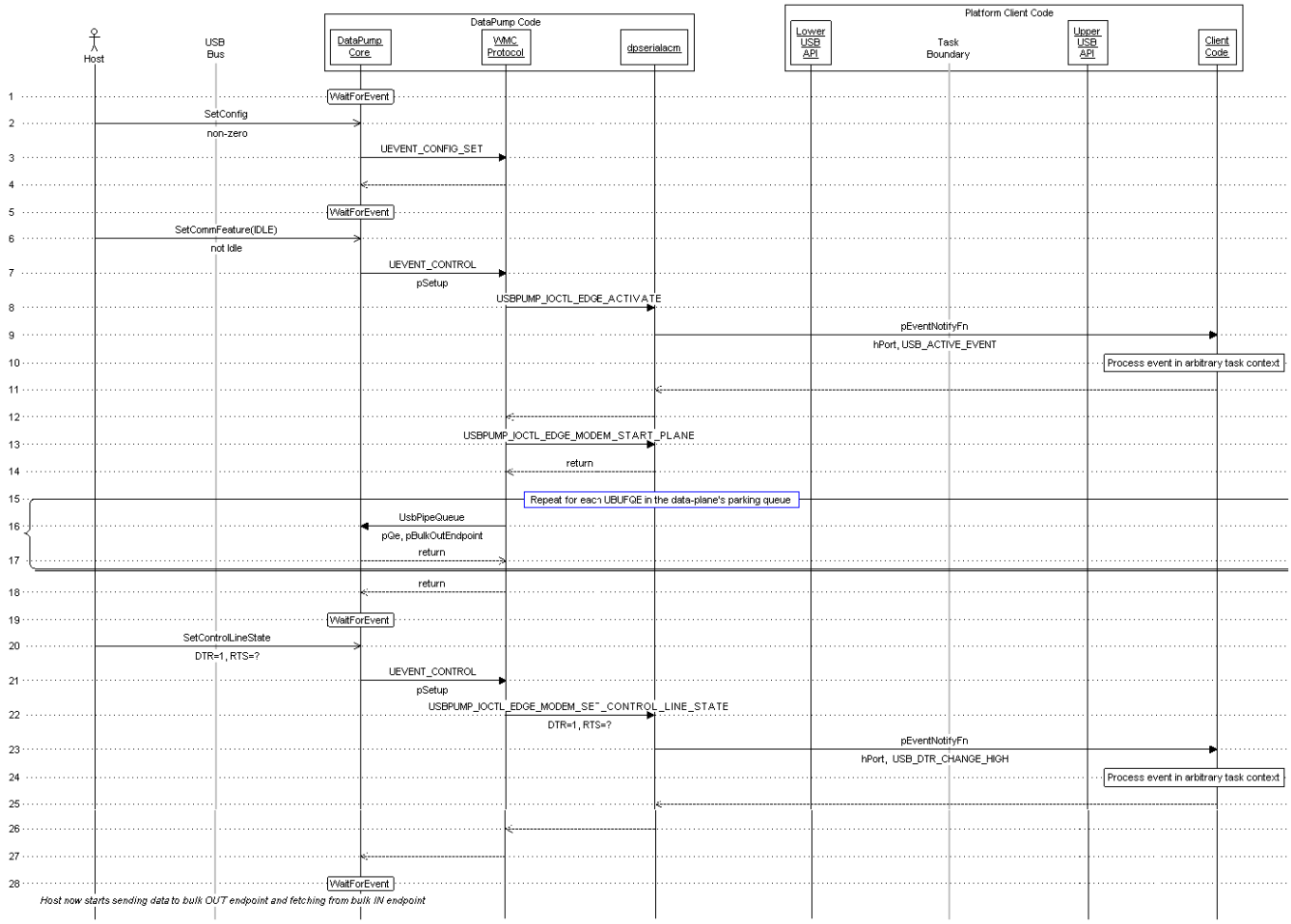
A number of events occur before data can begin to flow when the host opens a WMC port. Figure 4 shows the typical behavior when a Windows host with MCCI WMC drivers trying to open a WMC port.

When host starts to do SetConfig and SetCommFeature out of the idle state, DataPump WMC protocol instance analyzes the command and then uses the activation event to notify the client task through USBSERI API (line 1 to line 11).

Then the modem plane is started and the UBUFQE buffer is enqueued for the data stream associated with the particular pipe (line 13 to line 17).

A SetControlLineState event may also be passed from the host to activate transmission, the event will be passed through USBSERI API to client for further processing (line 20 to line 27).

Figure 4. Sequence diagram for host opening WMC ports



8 Other considerations

No other considerations.

9 API Location

See usbkern/api/serial for the implementation of USBSERI API, and usbkern/api/serial/i/usbserifc.h for the prototypes.