CBA4 API/SDK

1 Introduction

Congratulations on your CBAIV purchase. This document will provide the necessary information to build the framework for a custom interface with your CBAIV. Drivers for your CBAIV can be obtained at http://www.westmountainradio.com/. Use of the MPUSBAPI.dll is documented in the Appendix.

This package also contains a samble application written in VB.net that demonstrates how to communicate with the CBAIV.

2 Safety

The West Mountain Radio CBA is inherently safe, but the batteries in testing are not. The CBA device itself will not allow you to exceed the maximum ratings of the device (with the exception of maximum voltage, which is 55V). The CBA will also automatically shutdown if it gets too hot.

Anything that stores energy is potentially dangerous. Some types of batteries are safer than others, but all are capable of causing explosions or fires.

For technical information on any battery, refer to the Battery Manufacturers website to get detailed information on the rating of the battery. Most battery labels have limited information, but the manufacturer's website may have more detailed information. General information for batteries is also available on the West Mountain Radio CD. Select Battery Information and FAQs.

Primary Safety Considerations:

- Never discharge a battery at a higher discharge rate than designed for.
- Never use a battery that has poorly insulated, frayed wiring, or exposed metal parts.
- Never connect a battery with reverse polarity.
- Never test a battery near flammable materials.
- Never allow a battery reach a temperature that causes it to get so hot that it may be too hot to touch.
- Recharge batteries immediately with an appropriate battery charger after the test is completed. Some types do not like to remain discharged for extended periods of time.
- Never test or charge batteries while unattended.
- Always have a fire extinguisher nearby.

CBA Operating Requirements:

- Never connect a battery with the polarity reversed.
- Always connect the USB cable BEFORE connecting the battery.
- Be aware the heatsink of the CBA is at battery potential (positive terminal of battery).

- Never connect a battery to the CBA if it is connected to something else.
- Contact West Mountain Radio for support before connecting to a USB hub.
- Operate the CBA in a cool dry location.
- Never allow metal parts or wiring of the battery to come in contact with the metal heat sink.
- Never allow the cooling fan to be blocked or jammed.
- Do not ignore the warnings from the CBA Software.
- The CBA can get hot. Do not touch the metal heat sink while, or shortly after, conducting a high power test.

Lead Acid Battery Safety Warnings:

Adhere to all safety considerations with lead acid batteries especially while charging. These batteries, especially automotive and marine types, give off explosive hydrogen gas when charged. A nearby or internal spark or flame can cause a lead acid battery to explode sending liquid acid and lead shrapnel in all directions. This is particularly dangerous and frequently causes blindness or severe scarring injuries.

Never use a standard lead acid battery without proper ventilation. Sealed lead acid batteries such as gelled or AGM (absorbed glass mat) types are much safer. Automotive and marine types are not as safe and should be used in protective ventilated housings.

Never make the last connection to a lead acid battery that may cause a spark on the terminal. Always connect the load or charger last, and at a safe distance away from the battery.

NiCad, NiMh, and Alkaline Safety Warnings:

Adhere to all safety considerations, as these types of batteries can be dangerous also. If they are shorted out or charged/discharged at too high of a rate, they can overheat and explode. A single D size NiCad cell can actually melt a 10 penny nail (it is not recommended to attempt.)

Lithium Safety Warnings:

Reprinted with permission of the Academy of Model Aeronautics

Words in bold *and* ( ) were added by West Mountain Radio

Lithium Battery Fires

Lithium batteries are becoming very popular for powering the control and power systems in our models. This is true because of their very high energy density **(amp-hrs/wt. ratio)** compared to NiCads or other batteries. With high energy comes increased risk in their use. The, principal, risk is FIRE which can result from improper charging, crash damage, or shorting the batteries. All vendors of these batteries warn their customers of this danger and recommend extreme caution in their use. In spite of this many fires have occurred as a result of the use of Lithium Polymer batteries, resulting in loss of models, automobiles, and other property. Homes and garages and workshops have also burned. A lithium battery fire is very hot (several thousand degrees) and is an excellent initiator for ancillary **(resulting)** fires. Fire occurs due to contact between Lithium and oxygen in the air. It does not need any other source of ignition, or fuel to start, and burns almost explosively.

These batteries must be used in a manner that precludes ancillary fire. The following is

recommended:

1. Store, and charge **(discharge)**, in a fireproof container; never in your model.
2. Charge **(discharge)**, in a protected area devoid of combustibles. Always stand watch over the charging **(discharging)**, process. Never leave the charging process unattended.
3. In the event of damage from crashes, etc, carefully remove to a safe place for at least a half hour to observe. Physically damaged cells could erupt into flame, and, after sufficient time to ensure safety, should be discarded in accordance with the instructions which came with the batteries. Never attempt to charge **(discharge)** a cell with physical damage, regardless of how slight.
4. Always use chargers designed for the specific purpose, preferably having a fixed setting for your particular pack. Many fires occur in using selectable/adjustable chargers improperly set. Never attempt to charge Lithium cells with a charger which is not specifically designed for charging Lithium cells. Never use chargers designed for Nickel Cadmium batteries.
5. Use charging systems that monitor and control the charge state of each cell in the pack. Unbalanced cells can lead to disaster if it permits overcharge of a single cell in the pack. If the batteries show any sign of swelling, discontinue charging, and remove them to a safe place outside as they could erupt into flames.
6. Most important: NEVER PLUG IN A BATTERY AND LEAVE IT TO CHARGE **(DISCHARGE)**, UNATTENDED OVERNIGHT. Serious fires have resulted from this practice.
7. Do not attempt to make your own battery packs from individual cells. These batteries CANNOT be handled and charged casually such as has been the practice for years with other types of batteries. The consequence of this practice can be very serious resulting in major property damage and/ or personal harm.

Primary Safety Considerations:

- Never discharge a battery at a higher discharge rate than it is designed for.
- Never use a battery that has poorly insulated or frayed wiring or exposed metal parts.
- Never connect a battery with reverse polarity.
- Be aware the heatsink of the CBA is at battery potential (positive terminal of battery).
- Never test a battery near flammable materials.
- Never allow a battery reach a temperature that causes it to get so hot that it may be too hot to touch.
- Recharge batteries immediately with an appropriate battery charger after tests are completed. Some types do not like to remain discharged for extended periods of time.
- Never test or charge batteries while unattended.
- Always have a fire extinguisher nearby.

Liability Disclaimer

West Mountain Radio will not be liable to you (whether under the law of contract, the law of torts or otherwise) in relation to the contents of, or use of, or otherwise in connection with, the CBA ("product"):

- To the extent that the product may cause any bodily or equipment damage;
- For any indirect, special or consequential loss of any kind;
- For any business losses, loss of revenue, income, profits or anticipated savings, loss of contracts or business relationships, loss of reputation or goodwill, or loss or corruption of information or data.

By using this product, you agree to the limitations of liability set forth in this disclaimer are reasonable.

3 CBA4 USB Protocol

3.1 VID/PID

VID = 0x2405
PID = 0x0005

3.2 Transfer Method

Bulk, Endpoint 1 ("\\MCHP_EP"). Use Microchip's mpusbapi.dll.

Max packet size is 60 bytes.

All multi-byte values are sent little endian (LSB first).

3.3 Command-Set

3.3.1 Set Status (0x53)

Sent by PC to change operational parameter of device.

Payload

	0x53	FLAGS	LOAD	FAN	LED1	LED2	IOTRIS	IOPORT	VSTOP
bytes	1	2	4	1	1	1	1	1	4

FLAGS sigifies a series of control flags.

Bit0 – Clear if you don't want to update test is running bit and LOAD value.
Bit1 – Test is running (set), Test is idle (clear). If test is idle, LOAD is ignored. [DEFAULT is clear]
Bit2 – PWMs are disabled because battery voltage is less than 0.5V. This is a read-only bit and automatically sets/clears depending on the current level of the battery.
Bit 3 - Calibration status. This bit is set if calibration values in unit appear good. This bit is clear if unit needs to be calibrated. This is a read-only value.

Bit 4 – actual current of test is limited because we are exceeding maximum power of device. This is a read-only value.
Bit 5 – Test aborted because we hit high temperature. This is a read-only value.
Bit 6 – If set, use VSTOP paramater. This bit is always assumed clear if payload doesn't contain enough bytes for VSTOP paramater.
Bit 7 – If set, unit stop because we hit VSTOP. This is a read-only value.
Bit 8 – Reserved
Bit 9 – Use fast DAC. See section 3.3.7 of this manual, otherwise leave clear.
Bit 10 – Use fast ADC. See section 3.3.5 of this manual, otherwise leave clear.
Bit 11 – Use thermistor voltage. See EXT_TEMP in section 3.3.3 of this manual.

LOAD is the new desired setpoint, in micro-amps. [DEFAULT is 0]

FAN is new desired fan setting:
0b00 – do not change from previous setting
0b01 – automatic control by firmware (on if unit is hot) [DEFAULT]
0b10 – off
0b11 – on

LED1 and LED2 are led settings:
0b00 – do not change from previous setting
0b01 – automatic control by firmware (LED1 on if USB is connected and drivers loaded, LED2 is on if FLAGS.1 is set) [DEFAULT]
0b10 – off
0b11 – on

IOTRIS controls the input/output direction of the general purpose IO pins. If bit7 is clear, PIC will ignore payload of this byte. Bit0 is IO pin 0, Bit1 is IO pin 1, etc. Bit clear (0) is for output, Bit set (1) is for input. [DEFAULT is 0x7F]

IOPORT controls the new output state if the bit is in output mode (if pin is input mode, that bit is gnored). If bit7 is clear, PIC will ignore payload of this byte. Bit clear (0) is for low, bit set (1) is for high.

VSTOP is microvolts ands sets an automatic stop voltage. This paramter is only used if bit6 of FLAGS is set. Normally the PC user software using a USB connection will not use this VSTOP and that bit in FLAGS will be clear.

If the PIC does not receive this command every 2 seconds, the PIC will automatically switch to DEFAULT setting. This will stop any and all tests.

The CBA will react immediately to this command to apply the new settings. There are some hardware limitations that do limit speed. On original CBAIVs, it takes the DAC 2-3 seconds to achieve the desired LOAD setpoint. On newer CBAIVs, it takes the DAC around 50ms to achieve the desired LOAD setpoint. You can tell if you have the newer faster DACs by going to "Tools-CBA Devices" from the WMR CBA software and seeing if there is an F at the end of the firmware verison (F indicates it is the newer faster DACs).

3.3.2 Get Config (0x43)

Sent by the PC when it wants a config packet from the device.

3.3.3 Send Status (0x73)

Sent by the PIC every ~150ms, regardless of what operational mode.

	0x73	FLAGS	LOAD	FAN	LED1	LED2	IOTRIS
bytes	1	2	4	1	1	1	1
	IOPORT	INT_ TEMP	EXT_TEMP	DETECT	VOLTAGE	VSTOP	TIME
	1	2	2	4	4	4	4

FLAGS is the same as the 'Set Status' command.

LOAD is the same as the 'Set Status' command, represents what LOAD we are aiming for.

FAN bits 0..6 are the same as 'Set Status' command. Bit7 is is set if the firmware currently has the fan running, clear if fan is not running.

LED1 and LED2 bits 0..6 are the same as 'Set Status' command. Bit7 is set if the firmware currently has the LED lit, clear if LED is dark.

IOPORT and IOTRIS are the same as 'Set Status' command. If pin is in output mode, IOTRIS bit reflects current output latch setting.

INT_TEMP is temperature, in tenths degrees fahrenheit. Anything below 0 is reported as 0.

EXT_TEMP is temperature, in tenths degrees fahrenheit. Anything below 0 is reported as 0. Bits will be all 1 if it doesn't appear that there is a temperature sensor attached. If bit 10 of FLAGS is set, then instead of temperature the raw ADC value will be sent – this is so customer can use their own thermistor (thermistor acts as a pull-down resistor, with a 680 ohm resistor acting as a pull-up to 3.3V).

DETECT is the current detected through the unit, in micro-amps.

VOLTAGE is the voltage detected on the unit, in micro-volts.

VSTOP is the VSTOP setting, as configured by the 'Set Status' command.

TIME is the time, in seconds, that test has been running and drawing current. This will stop incrementing once test has stopped or was automatically aborted by the firmware.

This message can be requested or sent before data has been updated. For example, the DETECT and VOLTAGE are only updated approximately every 270ms (as of CBAIV firmware 4.10).

3.3.4 Send Config (0x63)

Sent by the PIC as response to 'Get Config' message.

	0x63	HW_VER	FW_VER_MAJ	FW_VER_MIN	SER#
bytes	1	1	1	1	4
	MAX_LOAD	MIN_LOAD	MAX_VOLT	MAX_POWER	DATE

bytes	4	4	2	2	5
	FLAGS	MAX_LOAD2	MIN_LOAD2		
bytes	1	4	4		

HW_VER is the hardware revision of the device.

FW_VER is the firware version of the device.

SER# is the serial number of this device.

MAX_LOAD and MIN_LOAD are the max load setpoints (in micro amps) supported by this device. If bit0 of FLAGS is set, the max power isn't linear and ranges from MAX_LOAD:MIN_LOAD to MAX_LOAD2:MIN_LOAD2. If bit0 of FLAGS is clear, max power is linear and ranges from MAX_LOAD to MIN_LOAD.

FLAGS is a bitmap:

bit0 – if set, max power isn't linear. When determine max power MAX_LOAD2 and MIN_LOAD2 also need to be used (see above paragraph).

Some hardware may not send FLAGS byte.

MAX_LOAD2/MIN_LOAD2 – see above 2 paragraphs. Some hardware may not send these bytes.

MAX_VOLT is the maximum voltage that can be handled by this device.

MAX_POWER is the maximum power (in watts) that can be handled by this device.

DATE is the date timestamp that the device was calibrated. This matches time_t in C, which is seconds since January 1st 1970. The only exception to this value from time_t is that this is 5 bytes for a bigger range.

3.3.5 Many Samples (0x6D)

This packet is sent periodically if bit 10 of FLAGS is set, or if a Get Many Samples message is received (see section 3.3.6).

	0x6D	SEQ	NUM	VOLTS0	CURRENT0	VOLTS1	CURRENT1
bytes	1	1	1	4	4	4	4
	VOLTS2	CURRENT2	VOLTS3	CURRENT3	VOLTS4	CURRENT4	
	4	4	4	4	4	4	

This is used to send sample data at a faster fixed rate, whereas the sample data sent in the Send Status message (section 3.3.3) is sent at a slower rate and uses a slower sample rate. The data saved to this message is done at 20Hz (50ms per sample).

VOLTSx is measured voltage, in microvolts. CURRENTx is the test current, in microamps. VOLTS0/CURRENT0 is the oldest data, VOLTS4/CURRENT4 is the newest data.

SEQ is the sequence number for this block of data. When 5 new sets of samples have finished (5 * 50ms = 250ms), a new block of data is created with a higher sequence number and then sent over

USB. The device holds the current block of data and 3 older blocks of data (for 1s of history), the Get Many Samples request (section 3.3.6) can be used to request older data.

NUM is the number of blocks of data that can be retrieved that happened after this. Every 250ms when a new block of data has been filled and transmitted over USB, the NUM will always be 0. However if a Get Many Samples request is sent (section 3.3.6) for older data, this can be used to see how many blocks exist after this block.

3.3.6 Get Many Samples (0x4D)

This can be sent by the software to request older Many Samples data (section 3.3.5).

	0x4D	SEQ
bytes	1	1

SEQ is the sequence number of data that is requested. See SEQ and NUM of section 3.3.5 for more information.

3.3.7 Many Setpoints (0x7D)

If bit 9 of FLAGS is set, then the LOAD parameter of the Set Status request (section 3.3.3) is ignored and instead this request is used to pre-program the desired test current draw in steps. This can be used because the latency of sending the Set Status (3.3.1) may not provide enough precise real time control of the unit.

	0x6D	SEQ	LOAD0	LOAD1	LOAD2	LOAD3	LOAD4
bytes	1	1	4	4	4	4	4

LOAD5	LOAD6	LOAD7	LOAD8	LOAD9
4	4	4	4	4

LOADx is the desired test current, in microamps.

Each LOADx represents a 100ms step (LOAD0 is at time 0, LOAD1 is at time 0.1s, LOAD2 is at time 0.2s, etc).

SEQ – the implemantation of this field is difficult to describe and it appears that the firmware has a flaw in handling it. Basically, if you send a message with a SEQ of 0 then LOAD0-LOAD9 represent 1 second of current draws. The next message sent should be a SEQ of 1, then LOAD0-4 of this message should match LOAD5-9 of the last message and will be ignored; LOAD5-9 of the SEQ=1 message represent the next 500ms of data and is appended to this setpoint list. Each message should then have an incrementing SEQ, with a 1s of setpoint data but only 500ms of it is appended to the setpoint list.

4 CBA Charger

CBA Charger uses an interface very similar to the CBAIV.

4.1 VID/PID

VID = 0x2405
PID = 0x0004

4.2 Transfer Method

Bulk, Endpoint 1 ("\\MCHP_EP"). Use Microchip's mpusbapi.dll.

Max packet size is 60 bytes.

All multi-byte values are sent little endian (LSB first).

4.3 Command-Set

4.3.1 Set Charger Status (0x53)

Control LED and Relays on charger board.

	0x53	STATUS	LED1	LED2	LED3	OUTA	OUTB	OUTC	XOR	~XOR
bytes	*1*	*2*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

If this is not heard within 2s, unit will reset. This will prevent relays from being closed if the software crashes.

STATUS is used for any status or control flags. Right now this is unused and you should read/write 0s here.

LEDx are led settings:
- 0b00 – do not change from previous setting
- 0b01 – automatic control by firmware [DEFAULT]
- 0b10 – off
- 0b11 – on

LED1 is green, LED2 is yellow, LED3 is red.

OUTx control the 3 relays:
- 0b00 – do not change from previous setting
- 0b01 – do not change from previous setting
- 0b10 – off (open)
- 0b11 – on (closed)

XOR is XOR of all data (0x53, LEDx, OUTx).

~XOR is inverted XOR.

4.3.2 Send Charger Status (0x73)

This is sent every ~150ms by the charge controller.

	0x73	STATUS	LED1	LED2	LED3	OUTA	OUTB	OUTC	IN	LOAD
bytes	*1*	*2*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*4*

Status is the same as 'Set Charger Status (0x53)' message.

LEDx is the same as LEDx in 'Set Charger Status (0x53)' message.

OUTx is the same as OUTx in 'Set Charger Status (0x53)' message.

IN is the input status of INA and INB.

Bit0 = High in INA is asserted

Bit1 = High if INB is asserted

LOAD is the current detected current passing through the unit, in micro-amps.

4.3.3 Get Charger Config (0x43)

Sent by the PC when it wants a config packet from the device.

4.3.4 Send Charger Config (0x63)

Sent as a response to Get Charger Config (0x43) message. The format of this response is the same as the format of the CBAIV's Send Config (0x63) message. See Section 3.3.4 of this document for more information.

5 Appendix

```
/********************************************************************
 *
 *              MPUSBAPI Library
 *
 ********************************************************************
 * FileName:       mpusbapi.h
 * Dependencies:   None
 * Compiler:       C++
 * Company:        Copyright (C) 2004 by Microchip Technology, Inc.
 *
 * Software License Agreement
 *
 * The software supplied herewith by Microchip Technology Incorporated
 * (the ❽Company❾) for its PICmicro® Microcontroller is intended and
 * supplied to you, the Company❼s customer, for use solely and
 * exclusively on Microchip PICmicro Microcontroller products. The
 * software is owned by the Company and/or its supplier, and is
 * protected under applicable copyright laws. All rights are reserved.
 * Any use in violation of the foregoing restrictions may subject the
 * user to criminal sanctions under applicable laws, as well as to
 * civil liability for the breach of the terms and conditions of this
```

```
* license.
*
* THIS SOFTWARE IS PROVIDED IN AN ❽AS IS❾ CONDITION. NO WARRANTIES,
* WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
* TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT,
* IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR
* CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
*
* revision          Date       Comment
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* v0.0.0.0          9/2/04     Implemented MPUSBGetDeviceLink()
* v1.0.0.0          11/19/04   Original version 1.00 completed
* v1.0.1.0          03/24/08   Slight update to fix issue which
*                              was requiring admin run level to
*                              work correctly.
* v1.1.0.0          05/22/08   Added the support for the
*                              following functions:
*                               MPUSBSetConfiguration()
*                               MPUSBGetDeviceDescriptor()
*                               MPUSBGetConfigurationDescriptor()
*                               MPUSBGetStringDescriptor()
*                              Modified MPUSBGetDLLVersion return
*                               format.
*************************************************************************/

#ifndef _MPUSBAPI_H_
#define _MPUSBAPI_H_

#define MPUSB_FAIL              0
#define MPUSB_SUCCESS           1

#define MP_WRITE                0
#define MP_READ                 1

// MAX_NUM_MPUSB_DEV is an abstract limitation.
// It is very unlikely that a computer system will have more
// then 127 USB devices attached to it. (single or multiple USB hosts)
#define MAX_NUM_MPUSB_DEV       127

////////////////////////////////////////////////////////////////////////////
//    MPUSBGetDLLVersion : get mpusbapi.dll revision level
//
//    Input:
//          None
//    Output:
//          32-bit revision level MMmmddii
//            MM - Major release
//            mm - Minor release
//            dd - dot release or minor fix
```

```
//              ii - test release revisions
//     Note:
//           The formatting changed between revisions v1.0.1.0 of the
//             driver and v1.1.0.0.  The output of this function was
//             previously MMMMmmmm and did not match the .DLL file
//             numbering format.  The format of this fuction was changed to
//             match how the .DLL file number generation works.
//
DWORD (*MPUSBGetDLLVersion)(void);

////////////////////////////////////////////////////////////////////////////
//     MPUSBGetDeviceCount : Returns the number of devices with matching VID & PID
//
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
//
DWORD (*MPUSBGetDeviceCount)(PCHAR pVID_PID);

////////////////////////////////////////////////////////////////////////////
//     MPUSBOpen : Returns the handle to the endpoint pipe with matching VID & PID
//
//  All pipes are opened with the FILE_FLAG_OVERLAPPED attribute.
//  This allows MPUSBRead,MPUSBWrite, and MPUSBReadInt to have a time-out value.
//
//  Note: Time-out value has no meaning for Isochronous pipes.
//
//  instance - An instance number of the device to open.
//           Typical usage is to call MPUSBGetDeviceCount first to find out
//           how many instances there are.
//           It is important to understand that the driver is shared among
//           different devices. The number of devices returned by
//           MPUSBGetDeviceCount could be equal to or less than the number
//           of all the devices that are currently connected & using the
//           generic driver.
//
//           Example:
//           if there are 3 device with the following PID&VID connected:
//           Device Instance 0, VID 0x04d8, PID 0x0001
//           Device Instance 1, VID 0x04d8, PID 0x0002
//           Device Instance 2, VID 0x04d8, PID 0x0001
//
//           If the device of interest has VID = 0x04d8 and PID = 0x0002
//           Then MPUSBGetDeviceCount will only return '1'.
//           The calling function should have a mechanism that attempts
//           to call MPUSBOpen up to the absolute maximum of MAX_NUM_MPUSB_DEV
//           (MAX_NUM_MPUSB_DEV is defined in _mpusbapi.h).
//           It should also keep track of the number of successful calls
//           to MPUSBOpen(). Once the number of successes equals the
//           number returned by MPUSBGetDeviceCount, the attempts should
```

```
//          be aborted because there will no more devices with
//          a matching vid&pid left.
//
//  pVID_PID - A string containing the PID&VID value of the target device.
//          The format is "vid_xxxx&pid_yyyy". Where xxxx is the VID value
//          in hex and yyyy is the PID value in hex.
//          Example: If a device has the VID value of 0x04d8 and PID value
//                  of 0x000b, then the input string should be:
//                  "vid_04d8&pid_000b"
//
//  pEP     - A string of the endpoint number on the target endpoint to open.
//          The format is "\\MCHP_EPz". Where z is the endpoint number in
//          decimal.
//          Example: "\\MCHP_EP1"
//
//          This arguement can be NULL. A NULL value should be used to
//          create a handles for non-specific endpoint functions.
//          MPUSBRead, MPUSBWrite, MPUSBReadInt are endpoint specific
//          functions.
//          All others are not.
//          Non-specific endpoint functions will become available in the
//          next release of the DLL.
//
//          Note: To use MPUSBReadInt(), the format of pEP has to be
//              "\\MCHP_EPz_ASYNC". This option is only available for
//              an IN interrupt endpoint. A data pipe opened with the
//              "_ASYNC" keyword would buffer the data at the interval
//              specified in the endpoint descriptor upto the maximum of
//              100 data sets. Any data received after the driver buffer
//              is full will be ignored.
//              The user application should call MPUSBReadInt() often
//              enough so that the maximum limit of 100 is never reached.
//
//  dwDir   - Specifies the direction of the endpoint.
//          Use MP_READ for MPUSBRead, MPSUBReadInt
//          Use MP_WRITE for MPUSBWrite
//
//  dwReserved Future Use
//
//  Summary of transfer type usage:
//
//  ============================================================================
  =====
//  Transfer Type       Functions                   Time-Out Applicable?
//
//  ============================================================================
  =====
//  Interrupt - IN      MPUSBRead, MPUSBReadInt         Yes
//  Interrupt - OUT     MPUSBWrite                      Yes
//  Bulk - IN           MPUSBRead                       Yes
```

```
//  Bulk - OUT          MPUSBWrite                  Yes
//  Isochronous - IN    MPUSBRead                    No
//  Isochronous - OUT   MPUSBWrite                   No
//
=============================================================================
=====
//
//      Note that "input" and "output" refer to the parameter designations in calls
//      to this function, which are the opposite of common sense from the
//      perspective of an application making the calls.
//
HANDLE (*MPUSBOpen)(DWORD instance,         // Input
              PCHAR pVID_PID,           // Input
              PCHAR pEP,                // Input
              DWORD dwDir,              // Input
              DWORD dwReserved);        // Input <Future Use>

////////////////////////////////////////////////////////////////////////////
//      MPUSBRead :
//
//  handle  - Identifies the endpoint pipe to be read. The pipe handle must
//            have been created with MP_READ access attribute.
//
//  pData   - Points to the buffer that receives the data read from the pipe.
//
//  dwLen   - Specifies the number of bytes to be read from the pipe.
//
//  pLength - Points to the number of bytes read. MPUSBRead sets this value to
//            zero before doing any work or error checking.
//
//  dwMilliseconds
//          - Specifies the time-out interval, in milliseconds. The function
//            returns if the interval elapses, even if the operation is
//            incomplete. If dwMilliseconds is zero, the function tests the
//            data pipe and returns immediately. If dwMilliseconds is INFINITE,
//            the function's time-out interval never elapses.
//
//      Note that "input" and "output" refer to the parameter designations in calls
//      to this function, which are the opposite of common sense from the
//      perspective of an application making the calls.
//
DWORD (*MPUSBRead)(HANDLE handle,           // Input
              PVOID pData,              // Output
              DWORD dwLen,              // Input
              PDWORD pLength,           // Output
              DWORD dwMilliseconds);    // Input

////////////////////////////////////////////////////////////////////////////
//      MPUSBWrite :
//
```

```
//  handle  - Identifies the endpoint pipe to be written to. The pipe handle
//            must have been created with MP_WRITE access attribute.
//
//  pData   - Points to the buffer containing the data to be written to the pipe.
//
//  dwLen   - Specifies the number of bytes to write to the pipe.
//
//  pLength - Points to the number of bytes written by this function call.
//            MPUSBWrite sets this value to zero before doing any work or
//            error checking.
//
//  dwMilliseconds
//          - Specifies the time-out interval, in milliseconds. The function
//            returns if the interval elapses, even if the operation is
//            incomplete. If dwMilliseconds is zero, the function tests the
//            data pipe and returns immediately. If dwMilliseconds is INFINITE,
//            the function's time-out interval never elapses.
//
//      Note that "input" and "output" refer to the parameter designations in calls
//      to this function, which are the opposite of common sense from the
//      perspective of an application making the calls.
//
DWORD (*MPUSBWrite)(HANDLE handle,          // Input
              PVOID pData,              // Input
              DWORD dwLen,              // Input
              PDWORD pLength,           // Output
              DWORD dwMilliseconds);    // Input

////////////////////////////////////////////////////////////////////////////
//      MPUSBReadInt :
//
//  handle  - Identifies the endpoint pipe to be read. The pipe handle must
//            have been created with MP_READ access attribute.
//
//  pData   - Points to the buffer that receives the data read from the pipe.
//
//  dwLen   - Specifies the number of bytes to be read from the pipe.
//
//  pLength - Points to the number of bytes read. MPUSBRead sets this value to
//            zero before doing any work or error checking.
//
//  dwMilliseconds
//          - Specifies the time-out interval, in milliseconds. The function
//            returns if the interval elapses, even if the operation is
//            incomplete. If dwMilliseconds is zero, the function tests the
//            data pipe and returns immediately. If dwMilliseconds is INFINITE,
//            the function's time-out interval never elapses.
//
//      Note that "input" and "output" refer to the parameter designations in calls
//      to this function, which are the opposite of common sense from the
```

```
//     perspective of an application making the calls.
//
DWORD (*MPUSBReadInt)(HANDLE handle,       // Input
                PVOID pData,           // Output
                DWORD dwLen,           // Input
                PDWORD pLength,        // Output
                DWORD dwMilliseconds);  // Input

////////////////////////////////////////////////////////////////////////////
//     MPUSBClose : closes a given handle.
//
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
//
BOOL (*MPUSBClose)(HANDLE handle);

////////////////////////////////////////////////////////////////////////////
//     MPUSBGetDeviceDescriptor : Returns the Device Descriptor Data
//
//  handle  - Identifies the endpoint pipe to be read. The pipe handle must
//            have been created with MP_READ access attribute.
//  pDevDsc - pointer to where the resulting descriptor should be copied.
//  dwLen   - the available data in the pDevDsc buffer
//  pLength - a pointer to a DWORD that will be updated with the amount of data
//            actually written to the pDevDsc buffer.  This number will be
//            less than or equal to dwLen.
//
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
//
DWORD (*MPUSBGetDeviceDescriptor)(HANDLE handle,      // Input
                        PVOID pDevDsc,      // Output
                        DWORD dwLen,        // Input
                        PDWORD pLength);    // Output

////////////////////////////////////////////////////////////////////////////
//     MPUSBGetConfigurationDescriptor : Returns the Configuration Descriptor
//
//  handle  - Identifies the endpoint pipe to be read. The pipe handle must
//            have been created with MP_READ access attribute.
//  bIndex  - the index of the configuration descriptor desired.  Valid input
//            range is 1 - 255.
//  pDevDsc - pointer to where the resulting descriptor should be copied.
//  dwLen   - the available data in the pDevDsc buffer
//  pLength - a pointer to a DWORD that will be updated with the amount of data
//            actually written to the pDevDsc buffer.  This number will be
//            less than or equal to dwLen.
//
```

```
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
//
DWORD (*MPUSBGetConfigurationDescriptor)(HANDLE handle,       // Input
                        UCHAR bIndex,       // Input
                        PVOID pDevDsc,      // Output
                        DWORD dwLen,        // Input
                        PDWORD pLength);    // Output

////////////////////////////////////////////////////////////////////////////
//     MPUSBGetStringDescriptor : Returns the requested string descriptor
//
//  handle  - Identifies the endpoint pipe to be read. The pipe handle must
//            have been created with MP_READ access attribute.
//  bIndex  - the index of the configuration descriptor desired.  Valid input
//            range is 0 - 255.
//  wLangId - the language ID of the string that needs to be read
//  pDevDsc - pointer to where the resulting descriptor should be copied.
//  dwLen   - the available data in the pDevDsc buffer
//  pLength - a pointer to a DWORD that will be updated with the amount of data
//            actually written to the pDevDsc buffer.  This number will be
//            less than or equal to dwLen.
//
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
//
DWORD (*MPUSBGetStringDescriptor)(HANDLE handle,       // Input
                        UCHAR bIndex,       // Input
                        USHORT wLangId,     // Input
                        PVOID pDevDsc,      // Output
                        DWORD dwLen,        // Input
                        PDWORD pLength);    // Output

////////////////////////////////////////////////////////////////////////////
//     MPUSBSetConfiguration : Sets the device configuration through a USB
//           SET_CONFIGURATION command.
//
//  handle  - Identifies the endpoint pipe to be written. The pipe handle must
//            have been created with MP_WRITE access attribute.
//
//  bConfigSetting
//          - Denotes the configuration number that needs to be set.  If this
//            number does not fall in the devices allowed configurations then
//            this function will return with MP_FAIL
//
//     Note that "input" and "output" refer to the parameter designations in calls
//     to this function, which are the opposite of common sense from the
//     perspective of an application making the calls.
```

```
//
DWORD (*MPUSBSetConfiguration)(HANDLE handle,        // Input
                    USHORT bConfigSetting);          // Input
#endif
```