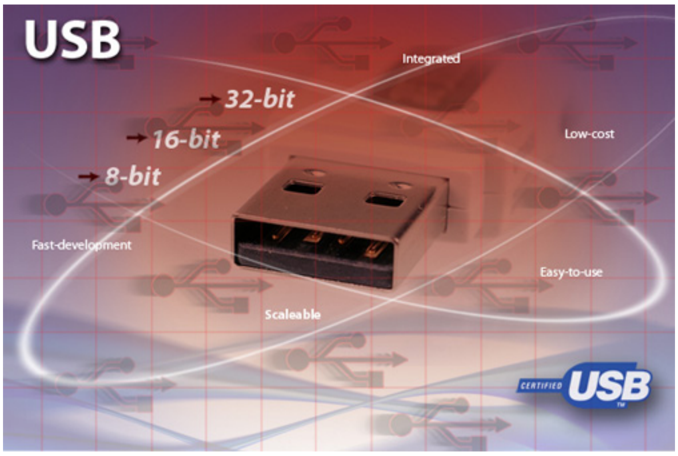
MCHPFSUSB Library Help

Table of Contents

1 Introduction

MCHPFSUSB v2.9c

for Microchip PIC18/PIC24F/PIC32MX Microcontrollers



MCHPFSUSB is a distribution package containing a variety of USB related firmware projects, USB drivers and resources intended for use on the PC. The MCHPFSUSB firmware examples include projects for USB peripheral/device, Embedded Host, OTG, and Dual Role.

2 Software License Agreement

MICROCHIP IS WILLING TO LICENSE THE ACCOMPANYING SOFTWARE AND DOCUMENTATION TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE FOLLOWING TERMS. TO ACCEPT THE TERMS OF THIS LICENSE, CLICK "I ACCEPT" AND PROCEED WITH THE DOWNLOAD OR INSTALL. IF YOU DO NOT ACCEPT THESE LICENSE TERMS, CLICK "I DO NOT ACCEPT," AND DO NOT DOWNLOAD OR INSTALL THIS SOFTWARE.

NON-EXCLUSIVE SOFTWARE LICENSE AGREEMENT

This Nonexclusive Software License Agreement ("Agreement") is a contract between you, your heirs, successors and assigns ("Licensee") and Microchip Technology Incorporated, a Delaware corporation, with a principal place of business at 2355 W. Chandler Blvd., Chandler, AZ 85224-6199, and its subsidiary, Microchip Technology (Barbados) II Incorporated (collectively, "Microchip") for the accompanying Microchip software including, but not limited to, Graphics Library Software, IrDA Stack Software, MCHPFSUSB Stack Software, Memory Disk Drive File System Software, mTouch(TM) Capacitive Library Software, Smart Card Library Software, TCP/IP Stack Software, MiWi(TM) DE Software, and/or any PC programs and any updates thereto (collectively, the "Software"), and accompanying documentation, including images and any other graphic resources provided by Microchip ("Documentation").

1. Definitions. As used in this Agreement, the following capitalized terms will have the meanings defined below:

a. "Microchip Products" means Microchip microcontrollers and Microchip digital signal controllers.

b. "Licensee Products" means Licensee products that use or incorporate Microchip Products.

c. "Object Code" means the Software computer programming code that is in binary form (including related documentation, if any), and error corrections, improvements, modifications, and updates.

d. "Source Code" means the Software computer programming code that may be printed out or displayed in human readable form (including related programmer comments and documentation, if any), and error corrections, improvements, modifications, and updates.

e. "Third Party" means Licensee’s agents, representatives, consultants, clients, customers, or contract manufacturers.

f. "Third Party Products" means Third Party products that use or incorporate Microchip Products.

2. Software License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to:

a. use the Software in connection with Licensee Products and/or Third Party Products;

b. if Source Code is provided, modify the Software, provided that no Open Source Components (defined in Section 5 below) are incorporated into such Software in such a way that would affect Microchip’s right to distribute the Software with the limitations set forth herein and provided that Licensee clearly notifies Third Parties regarding such modifications;

c. distribute the Software to Third Parties for use in Third Party Products, so long as such Third Party agrees to be bound by this Agreement (in writing or by "click to accept") and this Agreement accompanies such distribution;

d. sublicense to a Third Party to use the Software, so long as such Third Party agrees to be bound by this Agreement (in writing or by "click to accept");

e. with respect to the TCP/IP Stack Software, Licensee may port the ENC28J60.c, ENC28J60.h, ENCX24J600.c, and ENCX24J600.h driver source files to a non-Microchip Product used in conjunction with a Microchip ethernet controller;

f. with respect to the MiWi (TM) DE Software, Licensee may only exercise its rights when the Software is embedded on a Microchip Product and used with a Microchip radio frequency transceiver or UBEC UZ2400 radio frequency transceiver which are integrated into Licensee Products or Third Party Products.

For purposes of clarity, Licensee may NOT embed the Software on a non-Microchip Product, except as described in this Section.

3. Documentation License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to use the Documentation in support of Licensee's authorized use of the Software

4. Third Party Requirements. Licensee acknowledges that it is Licensee’s responsibility to comply with any third party license terms or requirements applicable to the use of such third party software, specifications, systems, or tools. Microchip is not responsible and will not be held responsible in any manner for Licensee’s failure to comply with such applicable terms or requirements.

5. Open Source Components. Notwithstanding the license grant in Section 1 above, Licensee further acknowledges that certain components of the Software may be covered by so-called "open source" software licenses ("Open Source Components"). Open Source Components means any software licenses approved as open source licenses by the Open Source Initiative or any substantially similar licenses, including without limitation any license that, as a condition of distribution of the software licensed under such license, requires that the distributor make the software available in source code format. To the extent required by the licenses covering Open Source Components, the terms of such license will apply in lieu of the terms of this Agreement. To the extent the terms of the licenses applicable to Open Source Components prohibit any of the restrictions in this Agreement with respect to such Open Source Components, such restrictions will not apply to such Open Source Component.

6. Licensee Obligations. Licensee will not: (i) engage in unauthorized use, modification, disclosure or distribution of Software or Documentation, or its derivatives; (ii) use all or any portion of the Software, Documentation, or its derivatives except in conjunction with Microchip Products or Third Party Products; or (iii) reverse engineer (by disassembly, decompilation or otherwise) Software or any portion thereof. Licensee may not remove or alter any Microchip copyright or other proprietary rights notice posted in any portion of the Software or Documentation. Licensee will defend, indemnify and hold Microchip and its subsidiaries harmless from and against any and all claims, costs, damages, expenses (including reasonable attorney's fees), liabilities, and losses, including without limitation product liability claims, directly or indirectly arising from or related to the use, modification, disclosure or distribution of the Software, Documentation, or any intellectual property rights related thereto; (ii) the use, sale and distribution of Licensee Products or Third Party Products; and (iii) breach of this Agreement.

7. Confidentiality. Licensee agrees that the Software (including but not limited to the Source Code, Object Code and library files) and its derivatives, Documentation and underlying inventions, algorithms, know-how and ideas relating to the Software and the Documentation are proprietary information belonging to Microchip and its licensors ("Proprietary Information"). Except as expressly and unambiguously allowed herein, Licensee will hold in confidence and not use or disclose any Proprietary Information and will similarly bind its employees and Third Party(ies) in writing. Proprietary Information will not include information that: (i) is in or enters the public domain without breach of this Agreement and through no fault of the receiving party; (ii) the receiving party was legally in possession of prior to receiving it; (iii) the receiving party can demonstrate was developed by the receiving party independently and without use of or reference to the disclosing party's Proprietary Information; or (iv) the receiving party receives from a third party without restriction on disclosure. If Licensee is required to disclose Proprietary Information by law, court order, or government agency, License will give Microchip prompt notice of such requirement in order to allow Microchip to object or limit such disclosure. Licensee agrees that the provisions of this Agreement regarding unauthorized use and nondisclosure of the Software, Documentation and related Proprietary Rights are necessary to protect the legitimate business interests of Microchip and its licensors and that monetary damage alone cannot adequately compensate Microchip or its licensors if such provisions are violated. Licensee, therefore, agrees that if Microchip alleges that Licensee or Third Party has breached or violated such provision then Microchip will have the right to injunctive relief, without the requirement for the posting of a bond, in addition to all other remedies at law or in equity.

8. Ownership of Proprietary Rights. Microchip and its licensors retain all right, title and interest in and to the Software and Documentation including, but not limited to all patent, copyright, trade secret and other intellectual property rights in the Software, Documentation, and underlying technology and all copies and derivative works thereof (by whomever produced). Licensee and Third Party use of such modifications and derivatives is limited to the license rights described in Sections this Agreement.

9. Termination of Agreement. Without prejudice to any other rights, this Agreement terminates immediately, without notice by Microchip, upon a failure by Licensee or Third Party to comply with any provision of this Agreement. Upon termination, Licensee and Third Party will immediately stop using the Software, Documentation, and derivatives thereof, and immediately destroy all such copies.

10. Warranty Disclaimers. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. MICROCHIP AND ITS LICENSORS ASSUME NO RESPONSIBILITY FOR THE ACCURACY, RELIABILITY OR APPLICATION OF THE SOFTWARE OR DOCUMENTATION. MICROCHIP AND ITS LICENSORS DO NOT WARRANT THAT THE SOFTWARE WILL MEET REQUIREMENTS OF LICENSEE OR THIRD PARTY, BE UNINTERRUPTED OR ERROR-FREE. MICROCHIP AND ITS LICENSORS HAVE NO OBLIGATION TO CORRECT ANY DEFECTS IN THE SOFTWARE.

11. Limited Liability. IN NO EVENT WILL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER ANY LEGAL OR EQUITABLE THEORY FOR ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS. The aggregate and cumulative liability of Microchip and its licensors for damages hereunder will in no event exceed $1000 or the amount Licensee paid Microchip for the Software and Documentation, whichever is greater. Licensee acknowledges that the foregoing limitations are reasonable and an essential part of this Agreement.

12. General. THIS AGREEMENT WILL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF ARIZONA AND THE UNITED STATES WITHOUT REGARD TO CONFLICTS OF LAWS PROVISIONS. Licensee agrees that any disputes arising out of or related to this Agreement, Software or Documentation will be brought in the courts of the State of Arizona. This Agreement will constitute the entire agreement between the parties with respect to the subject matter hereof. It will not be modified except by a written agreement signed by an authorized representative of the Microchip. If any provision of this Agreement will be held by a court of competent jurisdiction to be illegal, invalid or unenforceable, that provision will be limited or eliminated to the minimum extent necessary so that this Agreement will otherwise remain in full force and effect and enforceable. No waiver of any breach of any provision of this Agreement will constitute a waiver of any prior, concurrent or subsequent breach of the same or any other provisions hereof, and no waiver will be effective unless made in writing and signed by an authorized representative of the waiving party. Licensee agrees to comply with all export laws and restrictions and regulations of the Department of Commerce or other United States or foreign agency or authority. The indemnities, obligations of confidentiality, and limitations on liability described herein, and any right of action for breach of this Agreement prior to termination, will survive any termination of this Agreement. Any prohibited assignment will be null and void. Use, duplication or disclosure by the United States Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer-Restricted Rights clause of FAR 52.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clauses in the NASA FAR Supplement. Contractor/manufacturer is Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85224-6199.

If Licensee has any questions about this Agreement, please write to Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85224-6199 USA. ATTN: Marketing.

License Rev. No. 04-091511

3 Release Notes

3.1 What's New

Find out what is new for this stack release.

Description

New to v2.9c

- PC
  - Fixed HID boot loader executable issue on Windows systems
  - LibUSB example now works on Windows 7 and 64-bit machines
- Device
  - Fixed issue with some dsPIC33E projects not building correctly

New to v2.9b

- Host
  - Added MIDI host support
  - Bug fixes to various demos and client drivers
- Device
  - Addition of DTS support for CDC driver
  - Bug fixes to various demos
  - Added example showing how to connect to custom HID, LibUSB, WinUSB, and MCHPUSB demos from an Android v3.1+ host.

New to v2.9a

- PC Utilities
  - Bug fixes to cross-platform HID boot loader.

New to v2.9

- Device
  - Bug fixes and enhancements
  - Addition of PHDC class
- Host/OTG/Dual Role
  - Bug fixes and enhancements
  - Addition of Android host mode accessory support for OpenAccessory framework
- PC Utilities
  - Cross-platform custom HID application
  - Cross-platform HID boot loader

For more information about changes in this revision please refer to the Revision History (see page 12) section.

For potential migration questions, please refer to the Library Migration (see page 17) section.

3.2 What's Next

Find out what the USB development team is working on and what will be out in the near future.

Description

The following are the projects that are being worked on. These may not be released in the next release but are in development

- General improvements to the documentation and demos.

3.3 Support

Find out how to get help with your USB design, support questions, or USB training.

Description

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

• Product Support - Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software

• General Technical Support - Frequently Asked Questions (FAQs), technical support requests (http://support.microchip.com), online discussion groups/forums (http://forum.microchip.com, or more specifically the USB forum topic), Microchip consultant program member listing

• Business of Microchip - Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Development Systems Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com, click on Customer Change Notification and follow the registration instructions.

Additional Support

Users of Microchip products can receive assistance through several channels:

• Distributor or Representative

• Local Sales Office

• Field Application Engineer (FAE)

• Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is available on our website.

Technical support is available through the web site at: http://support.microchip.com

Training

• Regional Training Centers: http://www.microchip.com/rtc

• MASTERs Conference: http://www.microchip.com/masters

• Webseminars: http://techtrain.microchip.com/webseminars/QuickList.aspx

3.4 Online Reference and Resources

This section includes useful links to online USB development resources.

Description

Note: Newer versions of the documents below may be available. Please check www.microchip.com for the latest version.

USB Design Center

http://www.microchip.com/usb

Application Notes

Microchip USB Device Firmware Framework User's Guide

AN950 – Power Management for PIC18 USB Microcontrollers with nanoWatt Technology

AN956 – Migrating Applications to USB from RS-232 UART with Minimal Impact on PC Software

AN1140 – USB Embedded Host Stack

AN1141 – USB Embedded Host Stack Programmer's Guide

AN1142 – USB Mass Storage Class on an Embedded Host

AN1143 – Generic Client Driver for a USB Embedded Host

AN1144 - USB Human Interface Device Class on an Embedded Host

AN1145 – Using a USB Flash Drive on an Embedded Host

AN1189 – Implementing a Mass Storage Device Using the Microchip

AN1212 – Using USB Keyboard with an Embedded Host

AN1233 – USB Printer Class on an Embedded Host

USB Demonstration Videos

http://www.youtube.com/watch?v=ljF4KQ2mfD0

http://www.youtube.com/watch?v=cmtjKUv_yPs&feature=related

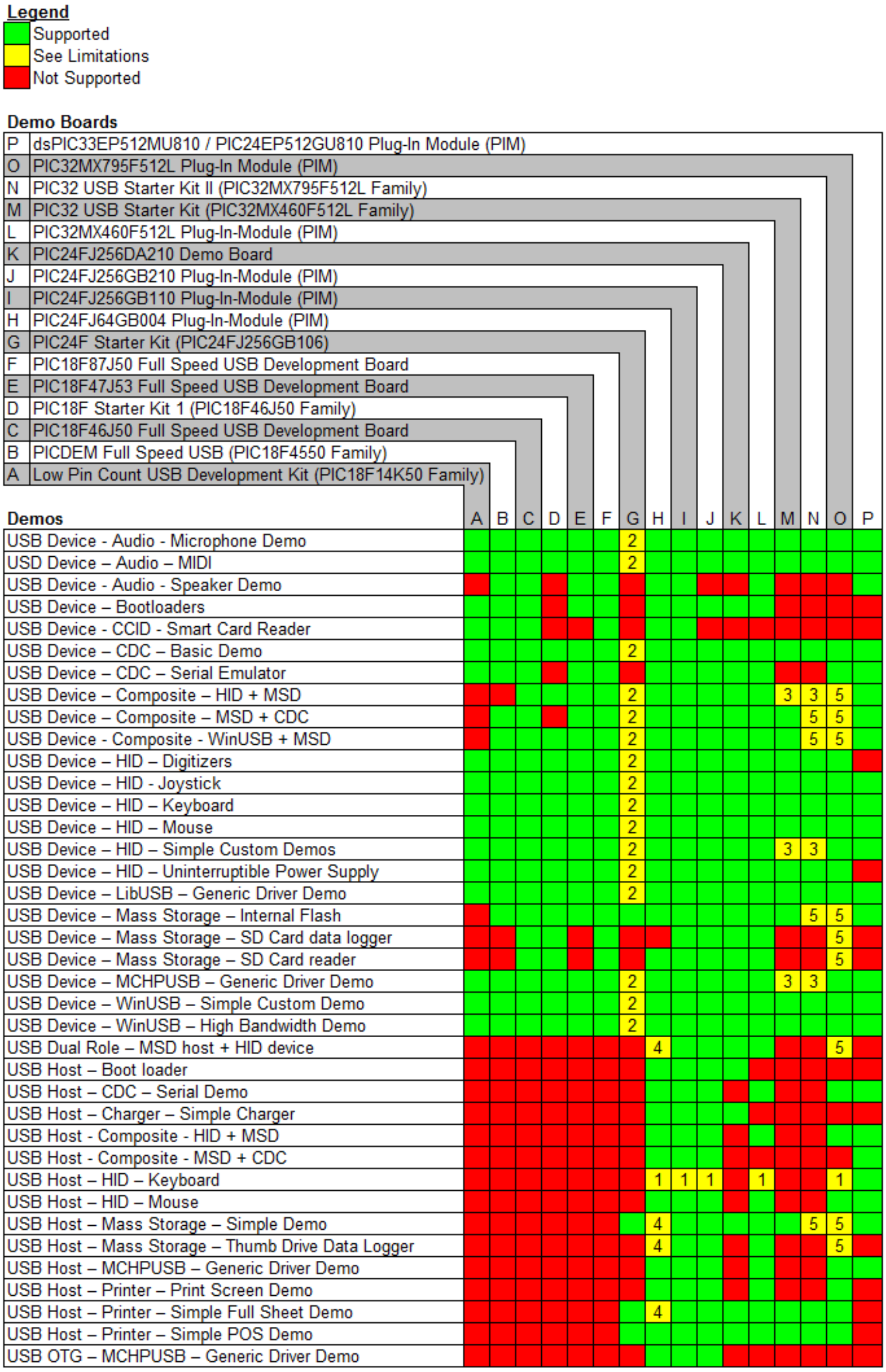
http://www.youtube.com/watch?v=BOosLeO7D58&feature=related

3.5 Demo Board Support and Limitations

This section shows which demos are supported on each of the USB demo boards.

Description

This section shows which demos are supported on each of the USB demo boards.



Legend

- Supported
- See Limitations
- Not Supported

Demo Boards

P	dsPIC33EP512MU810 / PIC24EP512GU810 Plug-In Module (PIM)
O	PIC32MX795F512L Plug-In Module (PIM)
N	PIC32 USB Starter Kit II (PIC32MX795F512L Family)
M	PIC32 USB Starter Kit (PIC32MX460F512L Family)
L	PIC32MX460F512L Plug-In-Module (PIM)
K	PIC24FJ256DA210 Demo Board
J	PIC24FJ256GB210 Plug-In-Module (PIM)
I	PIC24FJ256GB110 Plug-In-Module (PIM)
H	PIC24FJ64GB004 Plug-In-Module (PIM)
G	PIC24F Starter Kit (PIC24FJ256GB106)
F	PIC18F87J50 Full Speed USB Development Board
E	PIC18F47J53 Full Speed USB Development Board
D	PIC18F Starter Kit 1 (PIC18F46J50 Family)
C	PIC18F46J50 Full Speed USB Development Board
B	PICDEM Full Speed USB (PIC18F4550 Family)
A	Low Pin Count USB Development Kit (PIC18F14K50 Family)

Demos	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
USB Device - Audio – Microphone Demo							2									
USD Device – Audio – MIDI							2									
USB Device - Audio - Speaker Demo																
USB Device – Bootloaders																
USB Device - CCID - Smart Card Reader																
USB Device – CDC – Basic Demo							2									
USB Device – CDC – Serial Emulator																
USB Device – Composite – HID + MSD							2						3	3	5	
USB Device – Composite – MSD + CDC							2							5	5	
USB Device - Composite - WinUSB + MSD							2							5	5	
USB Device – HID – Digitizers							2									
USB Device – HID - Joystick							2									
USB Device – HID – Keyboard							2									
USB Device – HID – Mouse							2									
USB Device – HID – Simple Custom Demos							2						3	3		
USB Device – HID – Uninterruptible Power Supply							2									
USB Device – LibUSB – Generic Driver Demo							2									
USB Device – Mass Storage – Internal Flash														5	5	
USB Device – Mass Storage – SD Card data logger															5	
USB Device – Mass Storage – SD Card reader															5	
USB Device – MCHPUSB – Generic Driver Demo							2						3	3		
USB Device – WinUSB – Simple Custom Demo							2									
USB Device – WinUSB – High Bandwidth Demo							2									
USB Dual Role – MSD host + HID device								4							5	
USB Host – Boot loader																
USB Host – CDC – Serial Demo																
USB Host – Charger – Simple Charger																
USB Host - Composite - HID + MSD																
USB Host - Composite - MSD + CDC																
USB Host – HID – Keyboard								1	1	1		1			1	
USB Host – HID – Mouse																
USB Host – Mass Storage – Simple Demo								4						5	5	
USB Host – Mass Storage – Thumb Drive Data Logger								4							5	
USB Host – MCHPUSB – Generic Driver Demo																
USB Host – Printer – Print Screen Demo																
USB Host – Printer – Simple Full Sheet Demo								4								
USB Host – Printer – Simple POS Demo																
USB OTG – MCHPUSB – Generic Driver Demo																

Limitations

1) "Neither compound nor composite devices are supported. Some keyboards are either compound or composite.

The "~" prints as an arrow character instead ("->"). This is an effect of the LCD screen on the Explorer 16. The ascii character for "~" is remapped in the LCD controller.

The "\" prints as a "¥" character instead. This is an effect of the LCD screen on the Explorer 16. The ascii character for "\" is remapped in the LCD controller.

Backspace and arrow keys may have issues on Explorer 16 boards with certain LCD modules"

2) The PIC24F starter kit does not have a physical push button. The board uses capacitive touch buttons instead. The cap touch functionality has not been added to the demos yet so the functionality required by the demos is not currently available.

3) PIC32 USB Starter kit does not have a potentiometer, a temperature sensor, or a 4th LED on the board. Demos using these features do not function in their full capacity.

4) Due to the size of this demo, optimizations must be enabled in the compiler in order for this demo to work on the specified hardware platform. Optimizations are not available on all versions of the compilers.

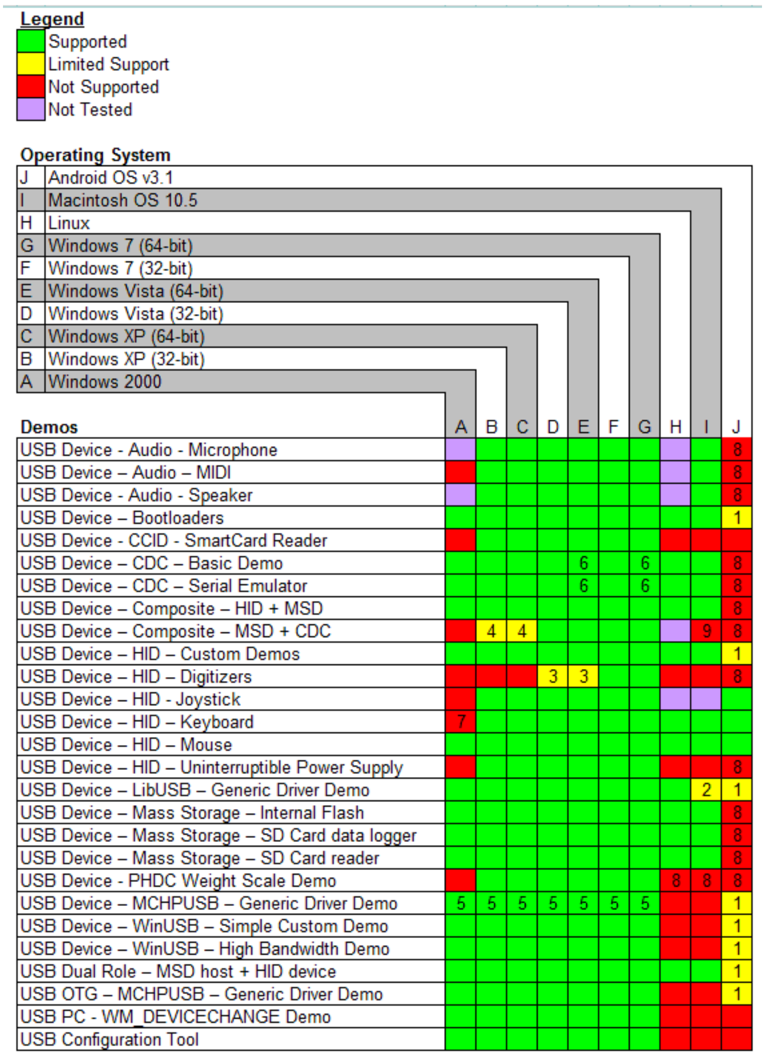
5) The PIC32MX795F512L Family devices have a register bit named READ. This conflicts with a definition in the MDD library. The MDD Library READ definition should not be used. Instead a 'r' should be used.

3.6 Operating System Support and Limitations

This section describes which operating systems support each of the provided demos.

Description

This section describes which operating systems support each of the provided demos.



Limitations

1) These devices enumerate successfully by the OS but currently there is not an example program to interface these devices.

2) Devices that implement the LibUSB demo will enumerate successfully on Macintosh based operating systems (provided the correct drivers are installed). Currently there is not an example program to communicate to these devices on these operating systems in this installation.

3) Only single touch gestures are supported in Windows Vista. For the multi touch demo only the single touch gestures will work as a gesture. The multi touch gestures in Vista will appears as two separate touch events that do not produce a usable pattern.

4) When used with Windows XP SP2 or earlier, this demo requires a Microsoft hotfix in order to run properly. This hotfix is linked from the demo folder. Windows XP SP3 works properly without needing any hotfix.

5) When adding a VID/PID string to the “%DESCRIPTION%=DriverInstall” and “%DESCRIPTION%=DriverInstall64” sections in the mchpusb.inf file, remove one or more of the pre-existing VID/PID strings from the list. There is a limit to the maximum number of VID/PID strings that can be supported simultaneously. If the list contains too many entries, the following error message will occur when installing the driver under Vista: "The Data Area Passed to a System Call Is Too Small"

6) The CDC PC example code does not run as implemented on the 64-bit version of the Windows Vista operating system with some versions of the .net framework. The .NET SerialPort object does not appear to receive data as implemented in these examples in the early versions of the .net framework for Vista.

7) The HID keyboard example does not work as implemented on the Windows 2000 operating system or any earlier revisions of the Windows operating systems.

8) Firmware successfully enumerates but test machine was unable to verify functionality. This is either due to lack of support in the OS for these types of devices or lack of an Application that uses these devices.

9) This demo uses the USB IAD specification. Some versions of Macintosh OSX do not support IAD.

3.7 Tool Information

Specifies the versions of the tools used to test this release.

Description

This release was tested with the following tools:

Compiler	Version
MPLAB C18	3.40
MPLAB C30	3.30c
MPLAB C32	2.01

IDE	Version
MPLAB	8.76
MPLAB X	beta 7.02

Some demos in this release require the full versions of the above compilers (the boot loaders and a few of the demo applications). For most demos, either the commercial version, or the evaluation version can be used to build the example projects. Some The compilers may be obtained from http://www.microchip.com/c18, http://www.microchip.com/c30, and http://www.microchip.com/c32.

3.8 Revision History

This section describes in more detail the changes made between versions of the MCHPFSUSB stack. This section generally discusses only changes made to the core files (those found in the <install directory>\Microchip folder). This section generally doesn't include changes to the demo projects unless those changes are important to know about. This section also doesn't encompass minor changes to the stack files such as arrangement or locations of definitions or any other organizational changes.

For more information about how to compare the actual source of two different revisions, please see the Appendix - Using a diff (see page 844) tool section of this document.

3.8.1 v2.9b

1. UART RX functionality fixed on several demos using the PIC24FJ256DA210 development board.
   - Stack files affected: none
2. Race condition fixed in Android OpenAccessory framework that could lead to the accessory not attaching periodically.
   - Stack files affected: usb_host_android_protocol_v1.c
3. Added Android Accessory workaround for when Android device attaches in accessory mode without first attaching as the manufacturer's mode (happens when accessory is reset but not detached from bus).
   - Stack files affected: usb_host_android_protocol_v1.c, usb_host_android.c, usb_host_android.h
4. Fixed issue where non-supported Android protocol versions would try to enumerate.
   - Stack files affected: usb_host_android.c
5. PIC18F Starter Kit MSD SD card reader demo not working correctly.
   - Stack files affected: none
6. Null pointer dereference on Android OpenAccessory detach event.
   - Stack files affected: usb_host_android_protocol_v1.c
7. Removed the restriction of MSD drives with the VID = 0x0930 and PID = 0x6545 for the USB MSD host data logging demo. These drives now show no issues with recent robustness enhancements in the past several releases.
   - Stack files affected: none
8. Link issues on Linux and Macintosh machines for PIC18 demos. The latest versions of the C18 compiler for Linux and Macintosh change the linker and library file capitalization scheme resulting in link errors when using older linker files. Linker files updated to use latest capitalization scheme.
   - Stack files affected: none
9. Cleaned up the configuration bits sections for several processors in several demos.
   - Stack files affected: none
10. CCID demo descriptors updated to enable operation on Macintosh machines.
    - Stack files affected: none
11. Update the precompiled MSD library to support .elf files.
    - Stack files affected: none
12. PCL5 printer host would send out a 0-length packet if an empty string was passed to it. This results in some PCL5 printers to lock up. The updated driver will not send out a text string to a printer if it is empty.
    - Stack files affected: none

13. USB_HID_FEATURE_REPORT was assigned the incorrect value.
    - Stack files affected: usb_host_hid.c
14. Some CDC device demos had incorrect USB_MAX_NUM_INT definition.
    - Stack files affected: none
15. Added examples showing how to connect to various USB demos with the Android USB host API.
    - Stack files affected: none
16. Optional support for DTS signalling added
    - Stack files affected: usb_function_cdc.c, usb_function_cdc.h
17. Added MIDI host support
    - Stack files affected: usb_host_midi.c, usb_host_midi.h
18. Added Android OpenAccessory boot loader example
    - Stack files affected: none
19. Fixed issues with PIC32 support with the MSD host boot loader. Now supports C32 versions 2.x and later.
    - Stack files affected: none

3.8.2 v2.9a

1. Fixes issues in the cross-platform HID boot loader that caused certain hex files not to work if the various sections in the hex file were not order in increasing address in the .hex file.
    - Stack files affected: none
2. Added UART output support for PIC24FJ256DA210 Development Board in Host – Printer Full sheet demo.
    - Stack files affected: none

3.8.3 v2.9

1. Adds PHDC peripheral support.
2. Adds Android accessory support for host mode accessories.
3. Added MPLAB X project files for most demo projects.
4. Added code to allow subclass 0x05 (SFF-8070i devices) to enumerate to the MSD host. Support limited to devices that use SCSI command set only.
    - Stack files affected: usb_host_msd.c
5. Added additional logic to MSD SCSI host code to improve support for various MSD devices by trying to reset various error conditions that may occur.
    - Stack files affected: usb_host_msd_scsi.c
6. Fixed issue with CDC host where SET_CONTROL_LINE_STATE command response was formatted incorrectly.
    - Stack files affected: usb_host_cdc.c
7. Added support for both input and output functionality in the Audio host driver.
    - Stack files affected: usb_host_audio.c
8. Added support for SOF, 1 millisecond timer, and data transfer event notifications to USB host drivers.
    - Stack files affected: usb_host.c

9. Added mechanism for a host client driver to override or reject the stacks selection for the class driver associated with an attached device.
   - Stack files affected: usb_host.c, usb_common.h
10. Fixed an issue with STALL handling behavior on non-EP0 endpoints for PIC24 and PIC32 devices.
    - Stack files affected: usb_device.c
11. Fixed an issue where some variables/flags were not getting re-initialized correctly after a set configuration event leading to communication issues when ping-pong is enabled and multiple set configuration commands are received.
    - Stack files affected: usb_device.c
12. Added mechanism to get the handle for the next available ping-pong transfer.
    - Stack files affected: usb_device.h
13. Fixed incorrect value for USB_CDC_CONTROL_LINE_LENGTH (see page 428) Stack files affected: usb_host_cdc.h
14. Updated MSD device driver to pass command verifier tests.
    - Stack files affected: usb_device_msd.c, usb_device_msd.h
15. Change to CDC device driver to allow handling of terminated transfers.
    - Stack files affected: usb_device_cdc.c

3.8.4 v2.8

1. Fixed issue with SetFeature(ENDPOINT_HALT) handling in the device stack. Error could cause one packet of data to get lost per endpoint after clearing a ENDPOINT_HALT event on an endpoint. Issue could also cause the user to lose control of endpoints that may not have been enabled before the SetFeature(ENDPOINT_HALT) was received. Parts of the issue described in the following forum thread: http://www.microchip.com/forums/tm.aspx?m=503200.
   - Stack files affected: usb_device.c
2. Fixed stability issue in device stack when interrupts enabled related to the improper enabling of the interrupt control bits in an interrupt context.
   - Stack files affected: usb_device.c
3. Fixed issue STALLs were not handled correctly when event transfers are enabled. This could result in the attached device remaining in a non-responsive state where their endpoints are STALLed.
   - Stack files affected: usb_host_msd.c
4. Fixed issue where MSD function driver could not always reinitialize itself to a known state.
   - Stack files affected: usb_function_msd.c
5. Added USBCtrlEPAllowStatusStage (see page 182)(), USBDeferStatusStage (see page 187)(), USBCtrlEPAllowDataStage (see page 181)(), USBDeferOUTDataStage (see page 185)(), USBOUTDataStageDeffered(), USBDeferInDataStage(), and USBINDataStageDeferred (see page 208)() functions. These functions allow users to defer the handling of control transfers received in interrupt context until a later point of time.
   - Stack files affected: usb_device.c, usb_device.h
6. Fixed issue in PIC18F starter kit SD-card bootloader issue. Bootloader could have errors loading hex files if there was an hex entry starting at an odd address with an even number of bytes in the payload.
   - Stack files affected: none
7. Reorganization of many of the definitions and data types.
   - Stack files affected: usb_hal_pic18.h, usb_hal_pic24.h, usb_hal_pic32.h, usb_device_local.h, usb_device.c, usb_device.h
8. Changed the behavior of the PIC24F HID bootloader linker scripts. The remapping.s file is no longer required. Interrupt vector remapping is now handled by the provided linker scripts (no customization required). Applications should be able to run with the bootloader linker script when either programmed or loaded through the bootloader allowing for more easy development and debugging. Interrupt latency should also be the same when using the bootloader or the debugger. For

more information about usage, please refer to the HID bootloader documentation.

9. Changed the behavior of the PIC32 HID bootloader linker scripts. The dual-linker script requirement has been replaced by a single required linker script that should be attached to the application project. Applications should be able to run with the bootloader linker script when either programmed or loaded through the bootloader allowing for more easy development and debugging. Interrupt latency should also be the same when using the bootloader or the debugger. For more information about usage, please refer to the HID bootloader documentation.
10. Added files for the PIC18F starter kit contest winners. Located in "<INSTALL_DIRECTORY>/PIC18F Starter Kit 1/Demos/Customer Submissions/Contest 1"
11. Added initial support for the PIC24FJ256DA210 development board (http://www.microchipdirect.com/ProductSearch.aspx?keywords=DM240312)
12. Added initial support for the PIC24FJ256GB210 Plug-in module (http://www.microchipdirect.com/ProductSearch.aspx?Keywords=MA240021).

3.8.5 v2.7a

1. Fixed USBSetBDTAddress() macro, so that it correctly loads the entire U1BDTPx register set, enabling the BDT to be anywhere in RAM. Previous implementation wouldn't work on a large RAM device if the linker decided to place the BDT[] array at an address > 64kB.
   - Stack files affected: usb_hal_pic32.h
2. Fixed initialization issue where HID parse result information wasn't cleared before loading with new parse result data.
   - Stack files affected: usb_host_hid_parser.c
3. Update to support the PIC18F47J53 A1 and later revision devices.
   - Stack files affected: usb_device.c
4. Fixed an error on 16-bit and 32-bit processors where a word access could be performed on a byte pointer resulting in possible address errors with odd aligned pointers.
   - Stack files affected: usb_device.c
5. Fixed issue where the USBSleepOnSuspend() function would cause the USB communication to fail after being called when _IPL is equal to 0.
   - Stack files affected: usb_hal_pic24.c
6. Fixed issue where placing the micro in idle mode would cause the host stack to stop sending out SOF packets.
   - Stack files affected: usb_host.c
7. Fixed several issues in the USBConfig.exe
8. Made changes to the starting address of the HID bootloader for PIC32. Reduced the size used by the bootloader. Also added application linker scripts for each processor.
9. Added a three point touch digitizer example
10. Updated some of the PC examples to build and run properly in the 2010 .net Express versions.
11. Added information and batch file showing how to enter a special mode of device manager that allows removal/uninstallation of devices that are not currently attached to the system.

3.8.6 v2.7

1. Fixed error where USBHandleGetAddr (see page 206)() didn't convert the return address from a physical address to a virtual address for PIC32.

- Stack files affected: usb_device.h

2. Added macro versions of USBDeviceAttach (see page 188)() and USBDeviceDetach (see page 189)() so they will compile without error when using polling mode.
   - Stack files affected: usb_device.h
3. Fixes issue in dual role example where a device in polling mode can still have interrupts enabled from the host mode causing an incorrect vectoring to the host interrupt controller while in device mode.
   - Stack files affected: usb_hal_pic18.h, usb_hal_pic24.h, usb_hal-pic32.h, usb_device.c
4. Modified the SetConfigurationOptions() function for PIC32 to explicitly reconfigure the pull-up/pull-down settings for the D+/D- pins in case the host code leaves the pull-downs enabled when running in a dual role configuration.
   - Stack files affected: usb_hal_pic32.h
5. Fixed error where the USB error interrupt flag was not getting cleared properly for PIC32 resulting in extra error interrupts (http://www.microchip.com/forums/tm.aspx?m=479085).
   - Stack files affected: usb_device.c
6. Updated the device stack to move to the configuration state only after the user event completes.
   - Stack files affected: usb_device.c
7. Fixed error in the part support list of the variables section where the address of the CDC variables are defined. The PIC18F2553 was incorrectly named PIC18F2453 and the PIC18F4558 was incorrectly named PIC18F4458 (http://www.microchip.com/forums/fb.aspx?m=487397).
   - Stack files affected: usb_function_cdc.c
8. Fixed an error where the USBHostClearEndpointErrors (see page 293)() function didn't properly return USB_SUCCESS if the errors were successfully cleared (http://www.microchip.com/forums/fb.aspx?m=490651).
   - Stack files affected: usb_host.c
9. Fixed issue where deviceInfoHID[i].rptDescriptor was incorrectly freed twice. The second free results in possible issues in future malloc() calls in the C32 compiler.
   - Stack files affected: usb_host_hid.c
10. Fixed an issue where the MSD client driver would issue a transfer events to an incorrect/invalid client driver number when transfer events are enabled.
    - Stack files affected: usb_host_msd.c
11. Fixed issue where a device that is already connected to the embedded host when the system is initialized may not enumerate.
    - Stack files affected: usb_host.c
12. Fixed issue where the embedded host or OTG device did not properly check bmRequestType when it thinks that a HALT_ENDPOINT request was sent to the device. This resulted in the DTS bits for the attached device getting reset causing possible communication issues.
    - Stack files affected: usb_host.c
13. Changed how the bus sensing works. In previous revisions it was impossible to use the USBDeviceDetach (see page 189) to detach from the bus if the bus voltage was still present. This is now possible. It was also possible to move the device to the ATTACHED state in interrupt mode even if the bus voltage wasn't available. This is now prohibited unless VBUS is present.
    - Stack files affected: usb_device.c
14. Added USBSleepOnSuspend() function. This function shows how to put the PIC24F to sleep while the USB module is in suspend and have the USB module wake up the device on activity on the bus.
    - Stack files affected: usb_hal_pic24.h, usb_hal_pic24.c
15. Modified the code to allow connection of USB-RS232 dongles that do not fully comply with CDC specifications.
    - Stack files affected: usb_host_cdc.h, usb_host_cdc.c, usb_host_cdc_interface.c, usb_host_interface.h
16. Modified API USBHostCDC_Api_Send_OUT_Data to allow data transfers more than 256 bytes.
    - Stack files affected: usb_host_cdc.h, usb_host_cdc.c, usb_host_cdc_interface.c, usb_host_interface.h
17. Improved error case handling when the host sends more OUT bytes in a control transfer than the firmware was

expecting to receive (based on the size parameter when calling USBEP0Receive (see page 196)()).

- Stack files affected: usb_device.c

18. Added CCID (Circuit Cards Interface Device) class device/function support.
    - Stack Files affected: usb_function_ccid.h, usb_function_ccid.c
19. Added Audio v1 class embedded host support.
    - Stack files affected: usb_host_audio_v1.h, usb_host_audio_v1.c

3.9 Library Migration

3.9.1 From v2.9a to v2.9b

No changes required.

3.9.2 From v2.9 to v2.9a

No changes required.

3.9.3 From v2.8 to v2.9

No changes required.

3.9.4 From v2.7a to v2.8

1. HID Bootloader for PIC32 devices
    - An error was fixed in PIC32 bootloader. The previous implementations placed the interrupt vector table on a 1K-page aligned boundary. This table should be on a such a boundary. The user reset vector and the interrupt vector section addresses were switched to meet this requirement. Applications/bootloaders using the old reset vector will not work with applications/bootloaders using the new bootloader linker files.

3.9.5 From v2.7 to v2.7a

1. HID Bootloader for PIC32 devices
    - The PIC32 bootloader was changed in this revision. The memory region used by the HID bootloader was reduced. This could result in issues loading application projects built with the new linker scripts on a system with the old bootloader. It

could also result in issues loading an old application with the new bootloader.

3.9.6 From v2.6a to v2.7

No changes required.

3.9.7 From v2.6 to v2.6a

1. HID Bootloader for PIC24F devices
   - The HID Bootloader for PIC24F has been reworked for the v2.6a release. The change involve how interrupt remapping is handled and how applications relocate their code to make room for the bootloader. Applications built with the v2.6 or earlier PIC24F compiler should continue using the v2.6 bootloader and support files. It is recommended for new projects that new bootloader and support files should be used.
   - In previous revisions of the stack there was a “PIC24F HID Bootloader Remapping.s” file that was added to any PIC24F project to relocate the application code out of the bootloader space. These files have been deprecated and should not be used with the new revision of the bootloader. Instead there is a custom linker script (boot_hid_p24fjxxxGBxxx.gld) file in the HID bootloader folder specifically designed for the application. These are located in the “Application Files” folder in each of the respective bootloader folders. Copy this file from this folder into the application folder and add it to the target project. All of the possible interrupts should already be remapped. To use an interrupt, merely define the interrupt handler as you normally would if you weren’t using a bootloader.
   - The bootloader for PIC18 and PIC32 devices were not modified.

3.9.8 From v2.5 to v2.6

1. Include Files
   - The files that must be included into a project has changed from v2.5 to v2.6.
   - Version v2.5 of the MCHPFSUSB stack required multiple include files in order to work properly in device mode. The usb_device.h, usb.h, usb_config.h, and class specific files (i.e. - “./usb/usb_function_msd.h”) had to be included in all of the application files that accessed the USB stack as well as other common include files like the GenericTypeDefs.h and Compiler.h files.
   - In MHCPFSUSB v2.6, only the usb.h file and the class specific files (i.e. - “./usb/usb_function_msd.h”) must be included in the project. The usb_device.h and usb_config.h files should no longer be included in the application specific files.

2. Include Search Paths and Build Directory Policy
   - The preferred include path list has changed since the initial v2.x release. MPLAB now support compiling projects with respect to the project file instead of the source file. This is now the preferred method. With this modification the required include paths are the following:
     - .
     - ../Microchip/Include
   - If your project file located in a different format than the example projects, please add or remove the appropriate path modifiers such that the include path indirectly points to the /Microchip/Include folder.
   - To change the build directory policy and set the include paths, go to the “Project->Build Options->Project” menu. On the directories tab, select the include directories from the show directories drop down box.

3. Disabling Interrupt Handlers
   - In MCHPFSUSB v2.6, the interrupt handler routines are disabled through the usb_config.h file using the following definitions:
     - USB_DISABLE_SET_CONFIGURATION_HANDLER
     - USB_DISABLE_SUSPEND_HANDLER
     - USB_DISABLE_WAKEUP_FROM_SUSPEND_HANDLER
     - USB_DISABLE_SOF_HANDLER
     - USB_DISABLE_ERROR_HANDLER
     - USB_DISABLE_NONSTANDARD_EP0_REQUEST_HANDLER
     - USB_DISABLE_SET_DESCRIPTOR_HANDLER
     - USB_DISABLE_TRANSFER_COMPLETE_HANDLER
   - Defining any of these definitions in the usb_config.h file will disable the callback from the stack during these events. Please note that some of these events are required to be USB compliant. For example all USB devices must go into suspend mode when requested. The suspend handler is how the stack notifies the user that the bus has requested the device to go into suspend mode.
   - Also note that some device classes or demos may require certain handlers to be available in order to operate properly. For example, the audio class demo uses the start of frames provided by the SOF handler to properly synchronize the audio data playback.

4 Demos

4.1 Device - Audio Microphone Basic Demo

This demo shows how to implement a simple USB microphone. This demo uses a pre-recorded sound file in flash and plays that file when a pushbutton is pressed.

Description

4.1.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3

PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.1.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*

- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.1.3 Running the Demo

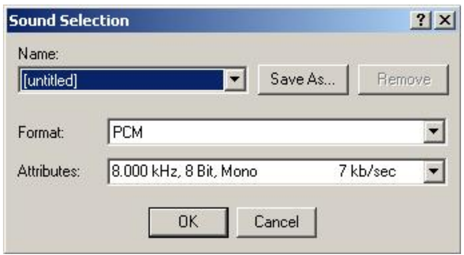
This demo uses the selected hardware platform as a USB Microphone Device. The demo emulates a PCM, 16 bits/Sample, 8000 Samples/ second, mono Microphone. Connect the device to the computer. Open a sound recording software package. Each sound recording software interface is different so the following instructions may not apply the to software package you are using. Please refer to the user’s manual for the software package you are using for more details of how to configure that tool for Sound recording.

Using Sound Recorder [Windows Computers]

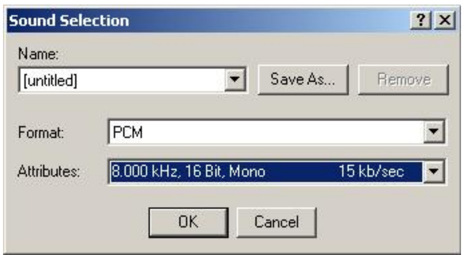
Open Sound Recorder from Start->Programs->Accessories->Entertainement->Sound Recorder. Click on File-> Properties.

Now the ‘Properties for Sound’ Window gets opened as shown below. Click on ‘Convert Now’ button.

This opens up the 'Sound Selection' window as shown below.



Change the 'Attributes' to "8.00kHz, 16 Bit, Mono 15kb/sec" in the 'Sound Selection' Window.



Click on OK button on the 'Sound Selection' Window. Click OK button on the 'Properties for Sound' Window.

Click on the Record Button on the Sound Recorder.

At this point you can press the pushbutton on the demo board and it will record a voice that is stored in the USB device. Once you finish with the recording click on the 'Play' button to play the recorded voice which can be heard through your computer Speaker.

Demo Board (click link for board information)	**Button**
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2

PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.2 Device - Audio MIDI Demo

This demo shows how to implement a simple bi-directional USB MIDI device.

4.2.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1

PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.2.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

<u>PIC24F Starter Kit</u>

- No hardware related configuration or jumper setting changes are necessary.

<u>PIC32 USB Starter Kit</u>

- No hardware related configuration or jumper setting changes are necessary.

<u>PIC32 USB Starter Kit II</u>

- No hardware related configuration or jumper setting changes are necessary.

4.2.3 Running the Demo

This demo uses the selected hardware platform as a USB MIDI device. Connect the device to the computer. Open a MIDI recording software package. Each MIDI recording software interface is different so the following instructions may not apply the to software package you are using. Please refer to the user’s manual for the software package you are using for more details of how to configure that tool for a USB MIDI input.

In this demo each time you press the button on the board, it will cycle through a series of notes.

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

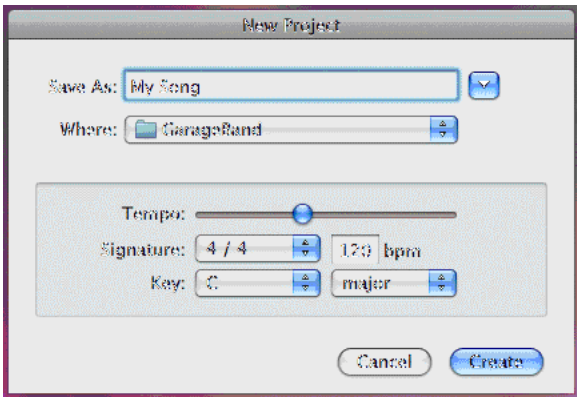
1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

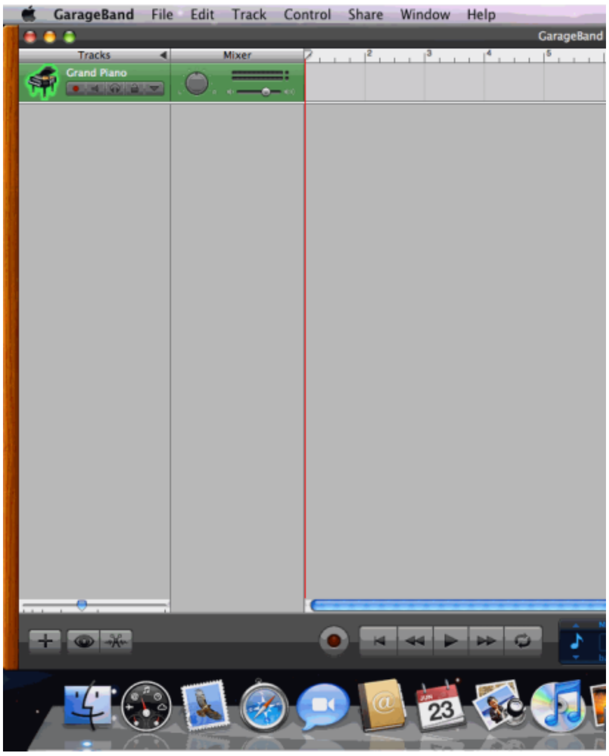
4.2.3.1 Garage Band '08 [Macintosh Computers]

Open Garage Band. If you haven’t opened Garage Band before you will see an opening window. Select “Create New Music Project”

The next window will prompt you for information about the song. Change any of the information is desired. Click “Create” when done.

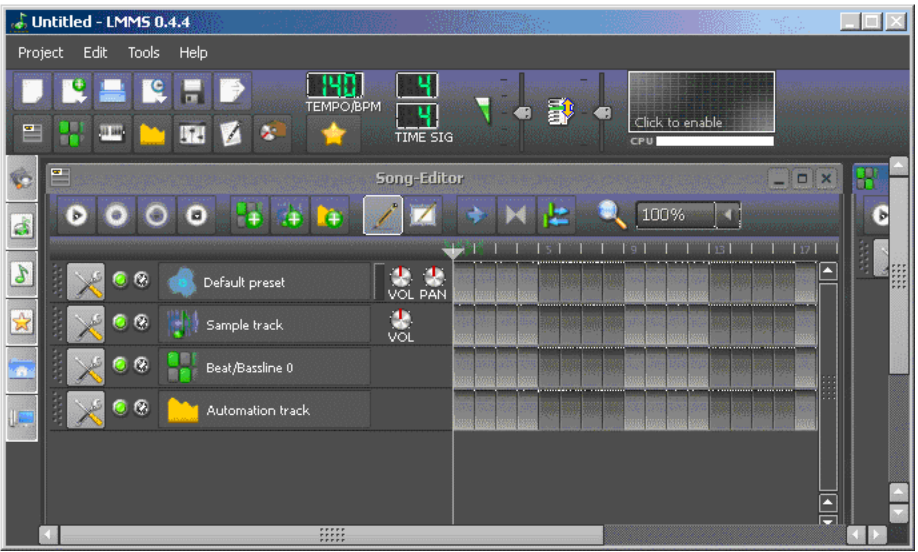


The Garage Band main window will open. In this window there should be a single default track if the USB device is already attached. At this point you can press the pushbutton (see page 26) on the demo board and it will cycle through a series of notes and play these notes through the computer speakers.

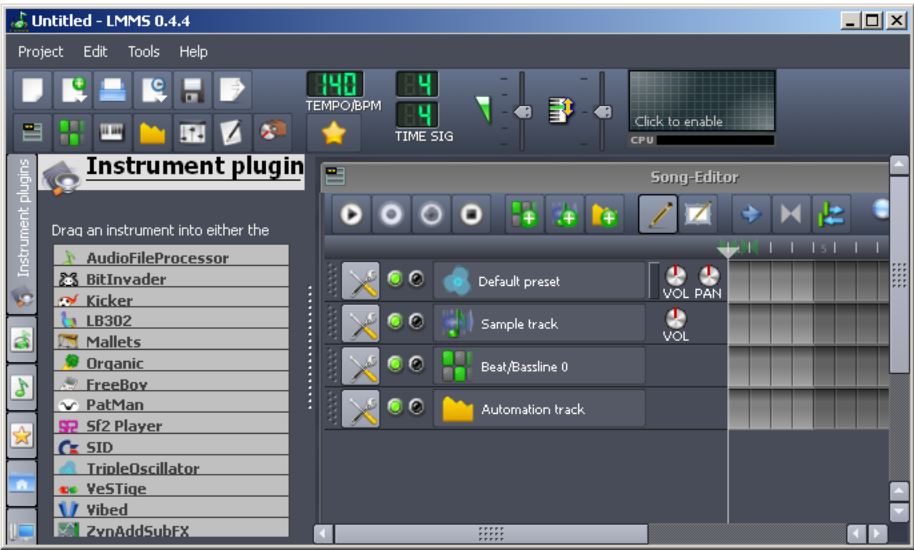

GarageBand
File
Edit
Track
Control
Share
Window
Help
GarageBand
Tracks
Mixer
Grand Piano

4.2.3.2 Using Linux MultiMedia Studio (LMMS) [Linux and Windows Computers]

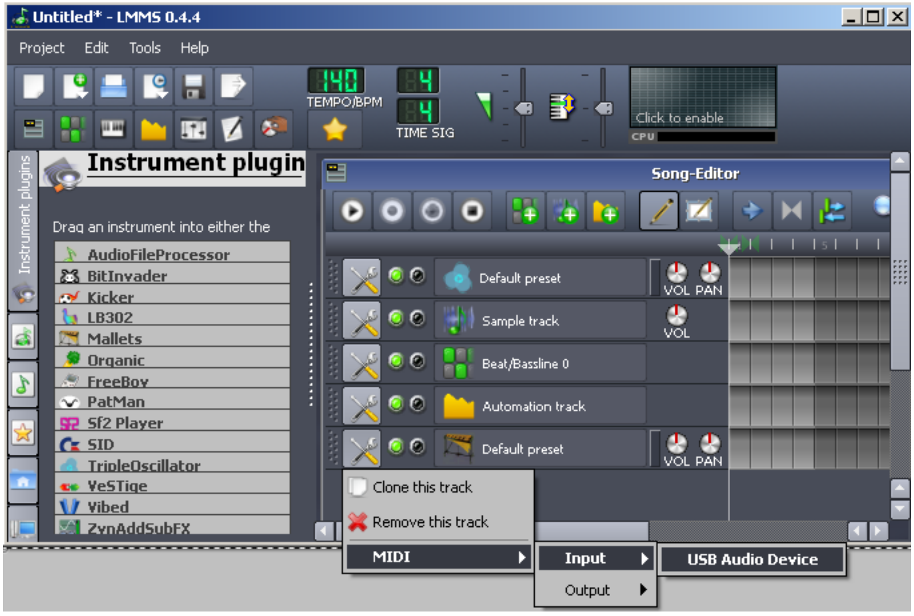
In this example we will be using Linux MultiMedia Studio (LMMS) available at http://sourceforge.net/projects/lmms/. Install LMMS. Attach the demo board to the computer. Make sure to attach the USB Audio MIDI example board to the computer before opening LMMS as LMMS polls for USB MIDI devices upon opening but may not find the devices attached after the program is opened.



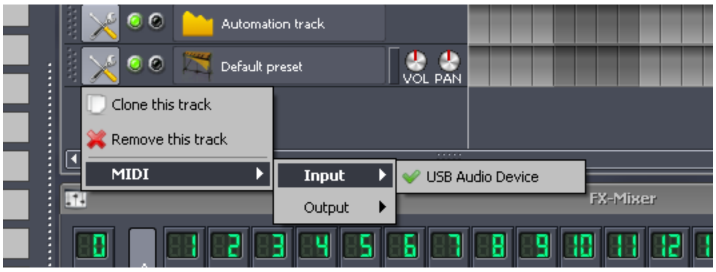
Click on the instrument plug-in button and click and drag the desired instrument plug in to the song editor window.



Once the new instrument is available in the song editor window, “click on the actions” for this track button. Select the “MIDI > Input > USB Audio Device” option.



If you open this option again you should see a green check mark indicating that the device is selected as the input.



At this point you can press the pushbutton on the demo board (see page 26) and it will cycle through a series of notes and play these notes through the computer speakers.

4.3 Device - Audio Speaker Demo

4.3.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PICDEM FS USB (see page 161)	DM163025	1
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	1, 2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	1, 2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	1, 2
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 3

PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 3
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1, 3
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1, 3
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1, 3
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1, 3
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1, 3

Notes:

1. These boards require the Speech Playback PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.3.2 Configuring the Hardware

PICDEM FS USB:

1. If header J6 is not populated on the board, you will need to populate it with a female header
2. Connect the Speech Playback Board.

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC18F46J50 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F47J53 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F87J50 Plug-In-Module:*
  1. *Short JP1 such that the "R" and the "U" options are shorted together.*
  2. *Short JP4. This allows the demo board to be powered through the USB bus power.*
  3. *Short JP5. This enabled the LED operation on the board.*

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board

4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.3.3 Running the Demo

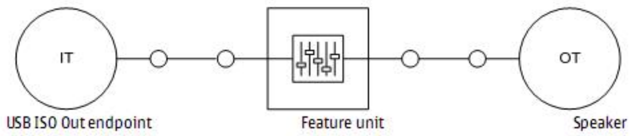
This demo functions as a speaker when plugged into a computer. Using any feature on the computer that normal produces sound on the speaker will work with this demo.

Please note that some applications lock into a sound source when they open or close (such as some web browsers or plug-ins), so that if you plug in the speaker with the webpage or video already playing, the sound might not get redirected to the USB based speakers until you close and reopen the browser.

The audio device created in this demo has the following characteristics:

- Sampling rate of 48 KHz
- 1 Channel (Mono)
- PCM Format - 16 bits per Sample
- Asynchronous Audio Endpoint

And the following audio topology:



The feature unit only supports the Mute control.

4.4 Device - Boot Loader - HID

In many types of applications, it is often desirable to be able to field update the firmware used on the flash microcontroller, such as to perform bug fixes, or to provide new features. Microchip's flash memory based USB microcontrollers have self programming capability, and are therefore able to perform self updates of application firmware. This can be achieved by downloading a new firmware image (.hex file) through the USB port, and using the microcontroller's self programming ability to update the flash memory.

As of this release the "HID Bootloader" is intended to be used with all PIC18 and PIC24F released Microchip USB flash microcontrollers.

The bootloader comes with full firmware and PC software source code, and is intended to be easily modified to support future Microchip USB microcontrollers. The PC software is designed to be independent of the microcontroller device being used, so only one PC application is needed to update any of the microcontroller devices.

4.4.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.4.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*

4.4.3 Running the Demo

All variants of the HID Bootloader firmware are intended to interface with the “HIDBootLoader.exe” PC application.

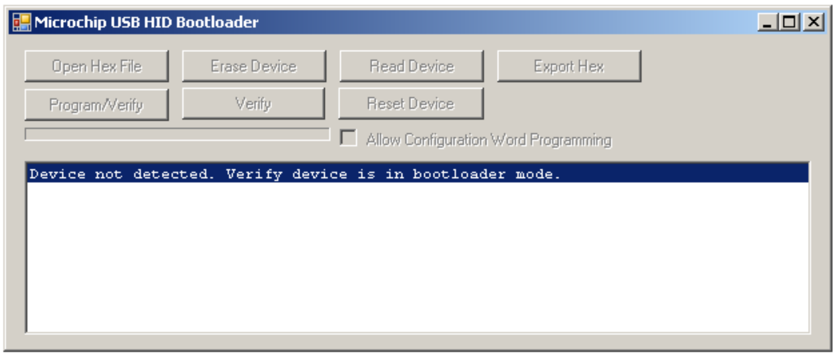
Before you can run the HIDBootLoader.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft’s website. Users of Windows Vista® and Windows 7 operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for the HIDBootLoader.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the “<Install Directory>\USB USB Device - Bootloaders\HID - Bootloader\HID Bootloader - PC Software” directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from

Microsoft's website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft's instructions for integrating it with the development environment.

It is not necessary to install either Microsoft Visual C++ 2005 or the Platform SDK in order to use the HID Bootloader. These are only required in order to modify or recompile the PC software source code.

To run the application, simply double click on the executable, which can be found in the following directory: "<Install Directory>\USB USB Device - Bootloaders\HID – Bootloader". Upon launching the application, a window like that shown below should appear:



If the application fails to launch, but instead causes a non-descript error message pop up box to appear, it is likely that the .NET framework redistributable has not been installed. Please install the .NET framework and try again.

Upon launch, the HIDBootLoader.exe program will do a search, looking for HID class devices with VID = 0x04D8, and PID = 0x003C. This is the same VID/PID that is used in the HID Bootloader firmware projects, which is found in the following directory: "<Install Directory>\USB Device - Bootloaders\HID - Bootloader\HID Bootloader - Firmware for (microcontroller family name)". When commercializing a product that will be using this bootloader, it is important to change the VID/PID in both the firmware and the PC application source code.

In order to use the bootloader, you will need to program a device with the bootloader firmware. If using a Microchip demo board, such as the PIC18F46J50 FS USB Demo Board (also known as "PIC18F46J50 PIM") (www.microchipDIRECT.com part number MA180024), precompiled demo .hex files can be used (without any modifications). These pre-compiled .hex files are located in the "<Install Directory>\USB Precompiled Demos" folder. After the HID bootloader firmware (ex: the .hex file named "USB Device - HID - HID Bootloader - C18 – PIC(device name).hex" has been programmed, continuously hold down the relevant pushbutton on the demo board, and then tap and release the MCLR pushbutton. After exiting from MCLR reset, the bootloader firmware will make a quick check of the pushbutton I/O pin state. If the pushbutton is pressed, it will stay in the bootloader.

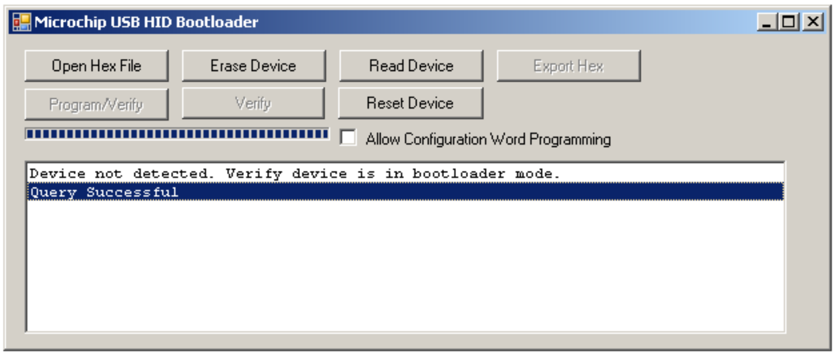
By default, the I/O pin that gets checked after exiting from reset will be:

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6[1]
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256DA210 Development Board (see page 167)	S1

Notes:

1) This is the button number on the Explorer 16.

Assuming that the device is connected correctly, and in bootload mode, the HIDBootLoader.exe application should automatically detect the device. The application uses WM_DEVICECHANGE messages in order to make for a smooth plug and play experience. Once the application detects the device, some of the buttons in the application should automatically become enabled.



At this point, “main application” firmware images can be loaded and programmed using the bootloader. The main application should not try to put code in addresses 0x00-0xFFF, because the bootloader will not attempt to program these locations (which is where the bootloader firmware resides). Therefore, when building the main application hex files, a modified linker script should be used. The “rm18f87j50.lkr” file included in the various USB device projects (such as in the “HID Mouse” project) shows an example of how this can be done.

By default, most of the pre-compiled demo .hex files are pre-configured to be useable with the HID Bootloader. Therefore, the pre-compiled demo firmware files, such as the “USB Device - HID - Mouse - C18 - PIC18F87J50 PIM.hex” can be directly programmed with the bootloader.

After an appropriate hex file has been programmed, simply reset the microcontroller (without holding down the bootloader entry pushbutton) to exit the bootloader and begin running the main application code. The main application firmware should begin running.

NOTE: The “USB Device - Mass Storage - SD Card reader” and “USB Device - Mass Storage - SD Card data logger” demos make use of the SD Card PICtail™ Daughter Board (Microchip Direct: AC164122). This PICtail uses the RB4 I/O pin for the card detect (CD) signal when used with the PIC18F87J50 FS USB Demo Board (PIM), and is actively driven by the PICtail. The active drive overpowers the pull up resistor on the RB4 pushbutton (on the PIC18F87J50 FS USB Demo Board). As a result, if the PIC18F87J50 is programmed with the HID bootloader, and an SD Card is installed in the socket when the microcontroller comes out of reset, the firmware will immediately enter the bootloader (irrespective of the RB4 pushbutton state). To exit the bootloader firmware, remove the SD Card from the SD Card socket, and tap the MCLR button. When the SD Card is not plugged in, the PICtail will drive the card detect signal (which is connected to RB4) logic high, which will enable the bootloader to exit to the main application after coming out of reset. Once the main application firmware is operating, the SD Card can be plugged in. The SD Card is “hot-swappable” and should be recognized by the host upon insertion. To avoid this inconvenience when using the bootloader with the PICtail, it is suggested to modify the bootloader firmware to use some other I/O pin for bootloader entry, such as RB0 (which has a pushbutton on it on the HPC Explorer board).

4.4.4 Implementation and Customization Details

4.4.4.1 Configuration Bits

Typically, when downloading new firmware images into the microcontroller, the configuration bit settings do not need to be modified. In some applications, it is sometimes desirable to be able to program new configuration bit settings into the microcontroller. Doing so entails a small amount of risk however, since it is potentially possible to program a new .hex file containing configuration bit settings that would be incompatible with USB operation (for example, if the oscillator settings are completely wrong). It is therefore generally recommended not to check the “Allow Configuration Word Programming” check box, unless strictly necessary. Special considerations should be kept in mind regarding the “Allow Configuration Word Programming” check box:

On currently supported PIC18xxJxx devices, the configuration words are stored in flash memory at the end of the implemented program memory space. However, the minimum erase page size is currently fixed at 1024 bytes for the currently supported microcontrollers. Therefore, if the “Allow Configuration Word Programming” box is left unchecked, then the last page of program memory will not get erase and will not get updated by the bootloader. If the main application firmware .hex file contains program code on the last page of implemented flash memory, it will not get updated. This can however be worked around, simply by checking the “Allow Configuration Word Programming” check box. The bootloader firmware will then erase and reprogram the last 1024 byte page of flash memory (which contains the configuration words).

4.4.4.2 Vendor ID (VID) and Product ID (PID)

When commercializing a product that will be using a USB bootloader, always make sure to use a unique VID and PID combination. Do not use the default VID/PID combination (from the bootloader firmware and PC application) in your commercialized product. If a PC has two devices, both containing the same bootloader with VID/PID = 0x04D8/0x003C, one made by manufacturer A (ex: a keyboard), and another device made by manufacturer B (ex: a CDC serial emulation device), then it is not certain which device the HID Bootloader PC application will connect to. The HID Bootloader PC application will search the system for any devices attached with matching VID/PID, but if there is more than one simultaneously attached, it will connect to the first one it finds. This could potentially lead to inadvertent flash updating of the wrong product, leading to unexpected and undesired consequences. By using a unique VID/PID for each product line of a given type, this ensures that the HID bootloader PC application will only find the correct device. To change the VID and PID in the bootloader firmware, simply change the USB device descriptor and rebuild the firmware. To change the HID Bootloader PC application, change the “MY_DEVICE_ID” string at the top of Form1.h, so that the VID/PID matches the firmware and then rebuild the project. The PC application is built in Microsoft Visual C++ 2005 .NET express edition. Microsoft currently distributes the express editions of Visual Studio languages for free download on their website.

4.4.4.3 Part Specific Details

4.4.4.3.1 PIC18F

Software entry to boot loader from application:

In the MCHPFSUSB v2.4 release, the PIC18F87J50 family and PIC18F46J50 family versions of the HID bootloader firmware also contains an alternative software only entry method into the bootloader. If executing the main application (non-bootloader) software, the main application may enter the bootloader by:

1. Clearing the global interrupt enable bit (INTCON<GIEH>

2. Execute the instruction: "_asm goto 0x001C _endasm"

It is not necessary to have the I/O pin in the logic low state when using this software entry method.

Memory Map Overview:

As configured by default, the HID bootloader firmware uses the below memory mapping. The memory map can readily be modified by editing the HID bootloader firmware project. It should not be necessary to modify the PC application source code to change the memory map.

0x000-0xFFF - Occupied by the HID bootloader firmware

- 0x08 (high priority interrupt vector) contains a "goto 0x1008" instruction
- 0x18 (low priority interrupt vector) contains a "goto 0x1018" instruction
- 0x1C is a main application firmware software only entry point into the bootloader (this entry point is currently implemented on the PIC18F87J50 family and PIC18F46J50 family versions of the firmware)
- 0x1000-(end of device flash memory) – Available for use by the main application firmware
- If programming in C18, normally should place a "goto _startup" instruction at address 0x1000, to allow the C initializer to run

Vector Remapping:

As currently configured, the bootloader occupies the address range 0x00-0xFFF (on PIC18), which means it occupies the PIC18 reset, high priority, and low priority interrupt vector locations. The bootloader firmware itself does not enable or use interrupts. In order to make interrupts available for use by the main application firmware, the interrupt vectors are effectively "remapped" by placing goto instructions at the actual vector locations. In other words:

Address 0x08 (high priority interrupt vector), contains a "goto 0x1008".

Address 0x18 (low priority interrupt vector), contains a "goto 0x1018".

For example, if a high priority interrupt is enabled and used in the main application firmware, the following will occur:

1. Main application enables the interrupt source.

2. Sometime later, the interrupt event occurs.

3. Microcontroller PC jumps to 0x08.

4. Microcontroller executes a "goto 0x1008".

5. Microcontroller executes the main application interrupt handler routine, which has an entry point at address 0x1008. (Note: The interrupt handler routine itself is not required to be at address 0x1008, instead another bra/goto may optionally be located at 0x1008 to get to the real handler routine)

4.4.4.3.2 PIC24F

Please refer to the PIC24F Boot Loader Implementation Specific Details (see page 824) appendix (see page 821) section for more information about how the boot loader works and fits in a PIC24F specifically.

4.5 Device - Boot Loader - MCHPUSB

The "MCHPUSB Bootloader" is a custom device class (requires driver installation) bootloader. The HID Bootloader is superior in a number of ways, and if developing a new application, it is recommended to consider developing with the HID bootloader instead. The MCHPUSB bootloader only supports the following microcontrollers: PIC18F4550, PIC18F4455, PIC18F2550, PIC18F2455, PIC18F4553, PIC18F4458, PIC18F2553, PIC18F2458, PIC18F4450, PIC18F2450.

4.5.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PICDEM FS USB (see page 161)	DM163025	

4.5.2 Configuring the Demo

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

4.5.3 Running the Demo

The MCHPUSB bootloader uses the PICDEM FS USB Demo Tool (pdfsusb.exe) for downloading/programming new firmware images from the PC. This program can be found in the following directory: "<install directory>\USB Tools\Pdfsusb". Documentation describing how to use this tool is found in chapter 3 of the PICDEM FS USB Demo Board User's Guide (DS51526). This document can be found in the following directory, "<install directory>\Microchip\USB\Documentation\Board Information\51526b.pdf". (Note: A newer version of this document may exist, please check the Microchip website. The 51526b.pdf version of the document is written with the assumption that the user is working with MCHPFSUSB v1.x, which uses a somewhat different directory structure compared to that of MCHPFSUSB v2.2)

4.5.4 Implementation and Customization Details

Two USB Stacks Approach:

The bootloader firmware contains all of the code needed for self programming, as well as all of the necessary code to enumerate as a custom (vendor) class USB device (which uses the mchpusb.sys custom driver).

The MCHPUSB bootloader firmware is an entirely stand alone MPLAB IDE based project. The "main application" firmware should be a separate MPLAB IDE based project altogether. The main application firmware is intended to be entirely independent of the bootloader. This requires that the main application should also contain a fully functional and complete USB stack. However, only one of the USB stacks is used at any given time.

With this approach, the main application firmware need not be a custom class device (nor does it need to be a “composite” device). In order to switch between the main application and the USB bootloader, the device “functionally detaches” itself from the USB bus (by temporarily turning off the pull up resistor), and then re-enumerates as the other firmware project.

Bootloader Entry Method:

As currently configured, the bootloader firmware resides in program memory in address range 0x00-0x7FF. Almost immediately after coming out of reset, the bootloader firmware checks I/O pin RB4 (which happens to have a pushbutton attached to it on the PICDEM™ FS USB Demo Board). If the pushbutton is not pressed, the bootloader will immediately exit the bootloader and go to the main application firmware “reset vector”.

In other words, the bootloader effectively does this:

//Device powers up, and comes out of POR

if(RB4 pushbutton is not pressed) --> goto 0x800 //main application “reset vector”

if(RB4 pushbutton is pressed) --> goto/stay in main bootloader project.

Effectively, the “reset” vector for the main application firmware is at address 0x800. In the main application firmware project, the user should place a “goto _startup” at address 0x800. This will allow the C initializer code to execute, which will initialize things like the software stack pointers and any user “idata” variables. For an example, see one of the USB device firmware projects, such as the “HID - Mouse” project. The PICDEM FSUSB version of this project is already configured to allow the generated .hex file to function along with the USB bootloader project.

Vector Remapping:

As currently configured, the bootloader occupies the address range 0x00-0x7FF, which means it occupies the PIC18 reset, high priority, and low priority interrupt vector locations. The bootloader firmware itself does not enable or use interrupts. In order to make interrupts available for use by the main application firmware, the interrupt vectors are effectively “remapped” by placing goto instructions at the actual vector locations. In other words:

Address 0x08 (high priority interrupt vector), contains a “goto 0x808”.

Address 0x18 (low priority interrupt vector), contains a “goto 0x818”.

For example, if a high priority interrupt is enabled and used in the main application firmware, the following will occur:

1. Main application enables the interrupt source.

2. Sometime later, the interrupt event occurs.

3. Microcontroller PC jumps to 0x08.

4. Microcontroller executes a “goto 0x808”.

5. Microcontroller executes the main application interrupt handler routine, which has an entry point at address 0x808. (Note: The interrupt handler routine itself might not be at address 0x808, but another bra/goto may be located at 0x808 to get to the real routine)

4.6 Device - CCID Smart Card Reader

4.6.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	1
PICDEM FS USB (see page 161)	DM163025	1
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	1, 2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	1, 2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 3

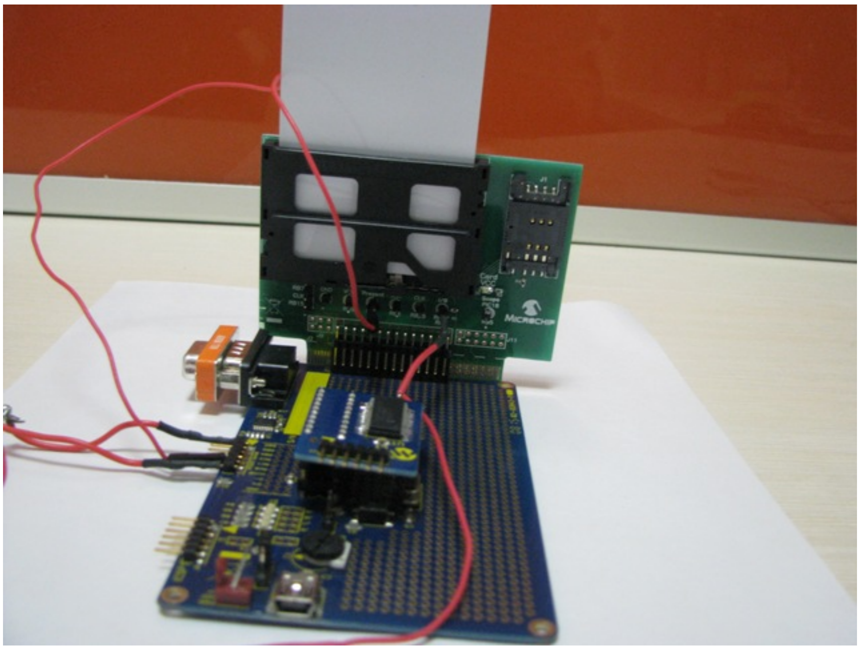
Notes:

1. These boards require the Smart/Sim Card PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.6.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.
3. One side of J4 port pins of the SC (Smart/Sim Card) PICTail Board match with the J11 port of LPC board. Insert the matching side of J4 port of SC PICTail board into the J11 port of LPC board. Make sure that the Smart Card Connector is facing towards the LPC board. Insert the Smart Card in SC PICTail board.
4. Short Tx & Rx line of the UART at J13 port using a wire and connect it to I/O pin of SC PICTail board.
5. Connect RB6 (of J13 port) to “Card Present” signal pin of SC PICTail board.



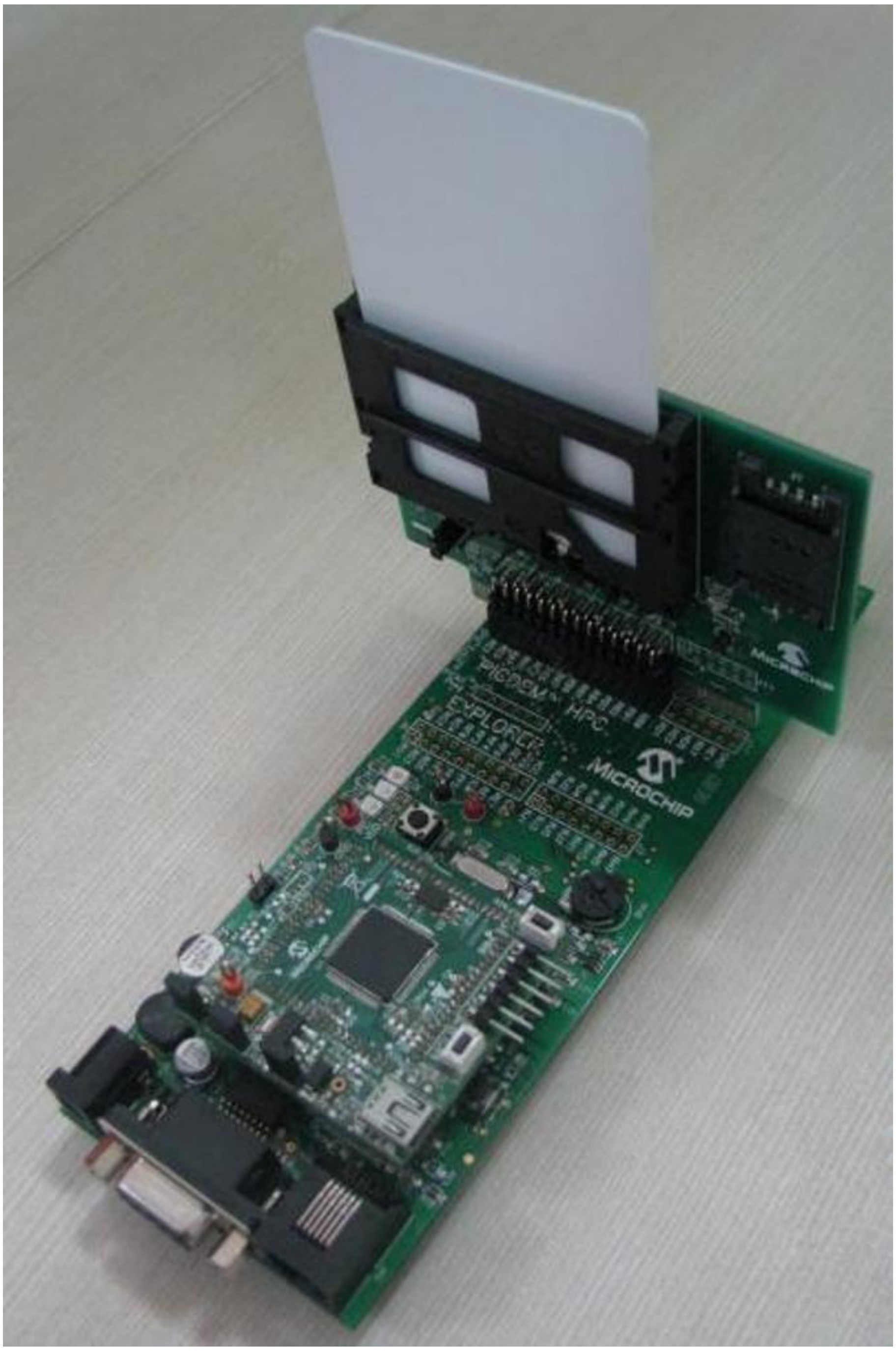
PICDEM FS USB:

1. If header J6 is not populated on the board, you will need to populate it with a female header
2. Connect the Speech Playback Board.
3. The Jumper JP11 needs to be open in this board. In some revision of the board it may necessary to cut the PCB track that is shorting the jumper.
4. Insert the J2 port of SC (Smart/Sim Card) PICTail card into J3 port of PICDEM FSUSB board as per the pin configuration. Insert the Smart Card in SC PICTail board.

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. Insert the J4 port of SC (Smart/Sim Card) PICTail Board to the J3 port of HPC Explorer board. Make sure that the Smart Card Connector is facing towards the HPC Explorer board. Insert the Smart Card in SC PICTail Board.



3. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC18F46J50 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F47J53 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F87J50 Plug-In-Module:*
  1. *Short JP1 such that the "R" and the "U" options are shorted together.*
  2. *Short JP4. This allows the demo board to be powered through the USB bus power.*

3. *Short JP5. This enabled the LED operation on the board.*

Explorer 16 Based Demos

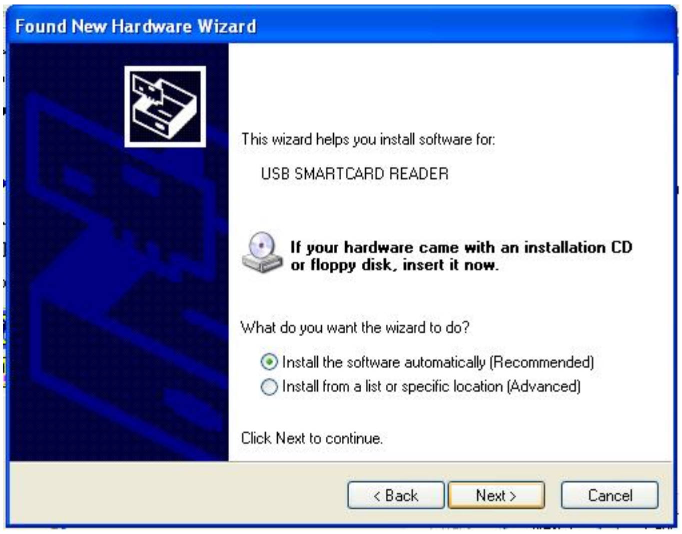
For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Insert the Smart/Sim Card daughter board into the first PICTail+ connector of the Explorer 16 (J5)

4.6.3 Running the Demo

This demo allows the selected hardware platform as a USB CCID Smart Card Reader to the host. In order to run this demo first compile and program the target device. Attach the device to the host. If the host is a Windows PC and this is the first time you have plugged this device into the computer then you may be prompted with “New hardware found” wizard.

Click on Next to continue.



Select “Install the Software automatically” and click on next.

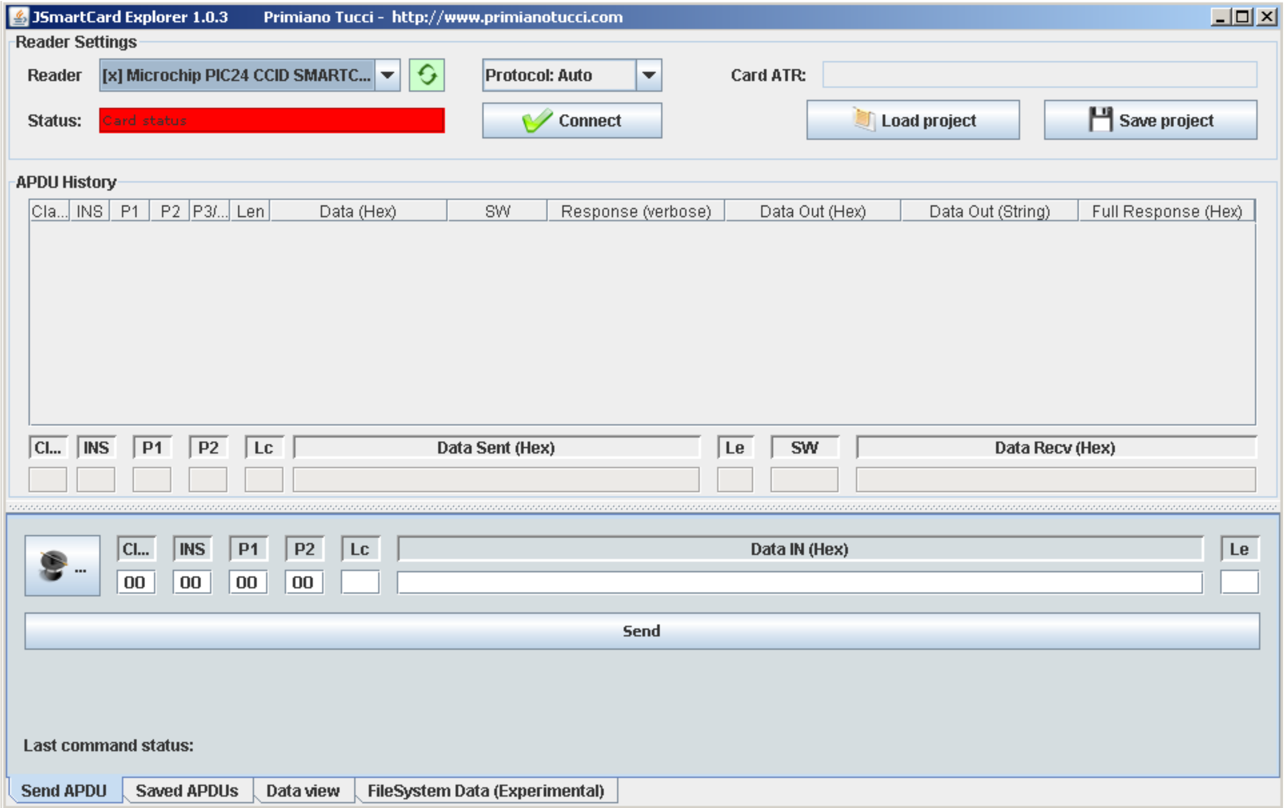
Click on Next to continue.

Click on Finish to complete the installation.

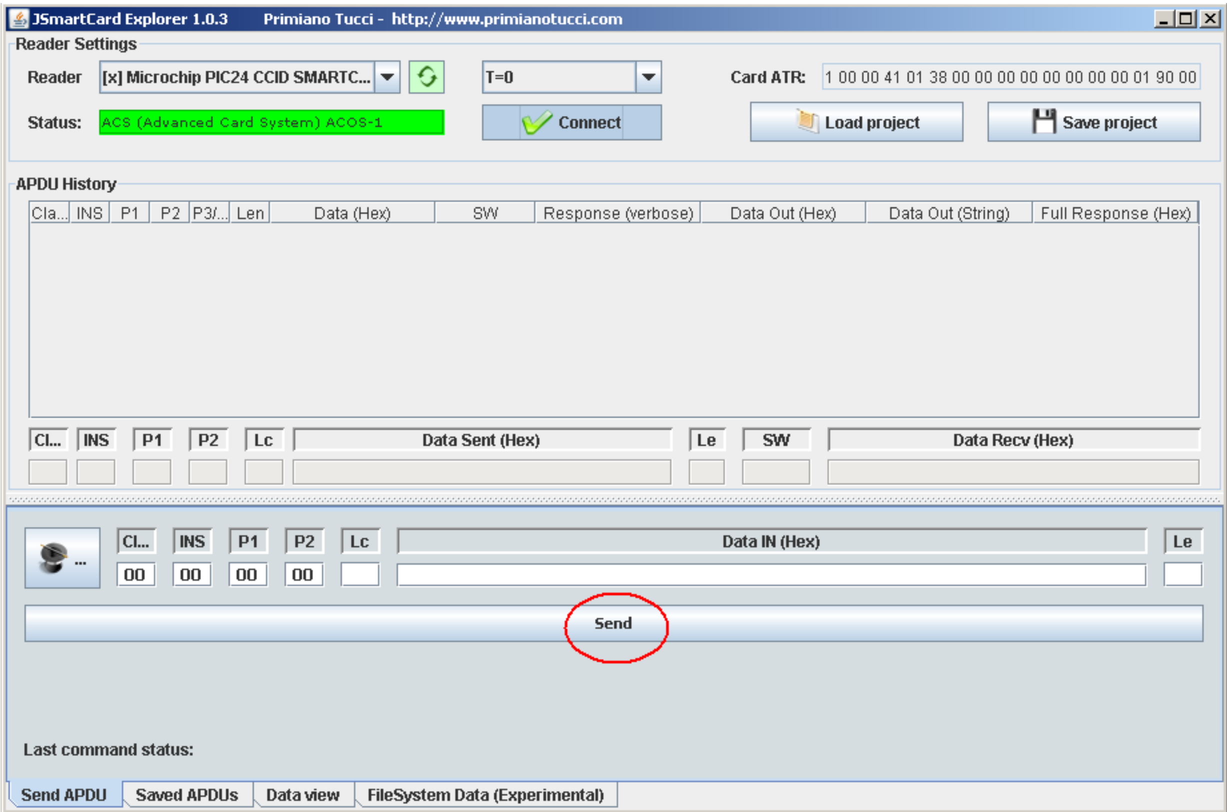
Note: Microsoft states that Usbccid.sys is compliant with Microsoft Windows 2000, Windows XP, and Microsoft Windows Server 2003 operating systems, and is available on Windows Update (http://www.microsoft.com/whdc/device/input/smartcard/usb_ccid.mspx). You might need to do a ‘Windows update’ if you windows computer does not have usbccid driver (or software required to install a usbccid driver) currently.

Using jSmartCardExplorer

Download SmartCardExplorer from http://www.primianotucci.com/default.php?view=112. Attach the demo board to the Computer. Ensure that the Smart Card is inserted on the SC Pictail Card. Launch the jSmartCardExplorer Application.



Select Protocol T=0 or T=1 based on the type of Smart card you insert and click on Connect Button. If the Smart Card is inserted on the SC Pictail Card, the 'Status' field turns green and the ATR of the Smart Card is displayed on the 'Card ATR' field.



The APDU can be send to the Smart Card by clicking on the 'Send' Button in the Send APDU section. The Command and

Data fields need to be filled before sending an APDU to the Smart Card. Please refer the Smart Card reference manual from the manufacturer of the Card for the Command list supported by the Card. The Response from the Smart Card is displayed in the APDU History Section.

A few sample commands for the ACS ACOS3 card is listed below.

Command	CLA	INS	P1	P2	LC	DATA	LE	Description
Select File	80	A4	00	00	02	FF00	00	Selects file FF00
Read Record	80	B2	00	00	00	--	08	Reads 8 bytes from record number 0 of FF00 file

4.7 Device - CDC Basic Demo

This example shows how to create a basic CDC demo. CDC devices appear like COM ports on the host computer and be communicated with via regular terminal software.

4.7.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3

PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.7.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*

- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

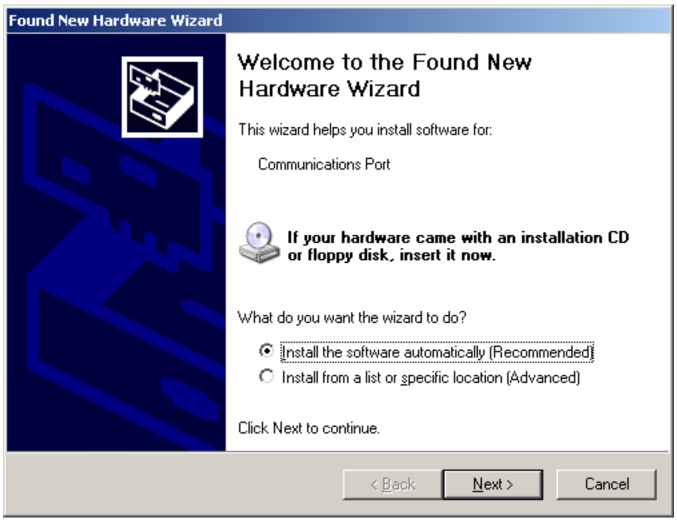
- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

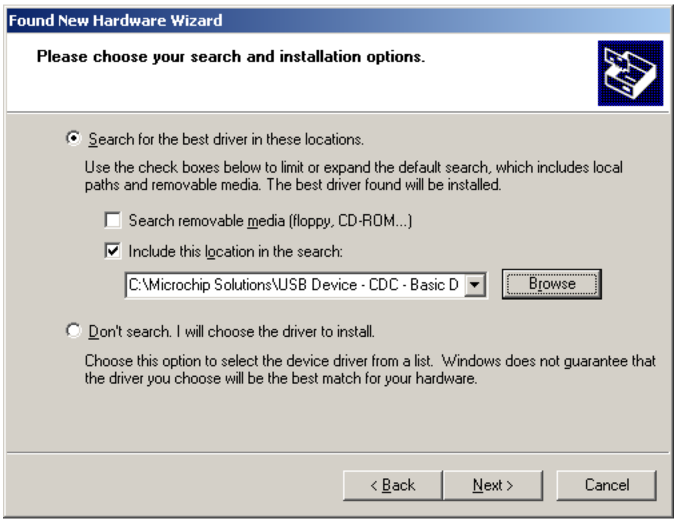
- No hardware related configuration or jumper setting changes are necessary.

4.7.3 Running the Demo

This demo allows the device to appear like a serial (COM) port to the host. In order to run this demo first compile and program the target device. Attach the device to the host. If the host is a PC and this is the first time you have plugged this device into the computer then you may be asked for a .inf file.



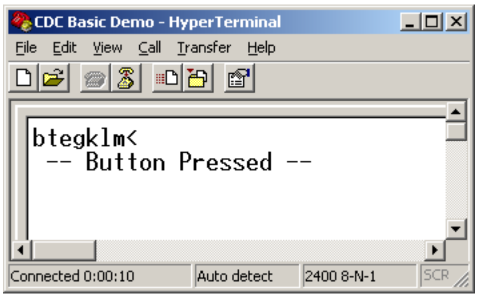
Select the “Install from a list or specific location (Advanced)” option. Point to the “<Install Directory>\USB Device - CDC – Basic Demo\inf\win2k_winxp” directory.



Once the device is successfully installed, open up a terminal program, such as hyperterminal. Select the appropriate COM port. On most machines this will be COM5 or higher.

Once connected to the device, there are two ways to run this example project. Typing a key in the terminal window will result in the device echoing that key plus one. So if the user presses “a”, the device will echo “b”. If the pushbutton is pressed the device will echo “ – Button Pressed – “ to the terminal window.

Note: Some terminal programs, like hyperterminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again in order to reconnect to the device.



Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)

PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.8 Device - CDC USB to UART Converter Demo

This demo shows how to use the CDC class to create a USB to UART bridge device. For a more simple starting point in using CDC based solutions, please consider using the Device - CDC Basic Demo (see page 50) as a starting point instead of this demo.

4.8.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	1
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	1, 2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	1, 2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	1, 2
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 3
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 3
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1, 3
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1, 3
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1, 3
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1, 3
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1, 3

Notes:

1. These boards require the Speech Playback PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.8.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No board specific settings are required

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC18F46J50 Plug-In-Module:*
    1. *Short JP2 such that the "R" and the "U" options are shorted together.*
    2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F47J53 Plug-In-Module:*
    1. *Short JP2 such that the "R" and the "U" options are shorted together.*
    2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F87J50 Plug-In-Module:*
    1. *Short JP1 such that the "R" and the "U" options are shorted together.*
    2. *Short JP4. This allows the demo board to be powered through the USB bus power.*
    3. *Short JP5. This enabled the LED operation on the board.*

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
    1. *Set switch S1 to the "PGX1" setting*
    2. *Short J1 pin 1 (marked "POT") to the center pin*
    3. *Short J2 pin 1 (marked "Temp") to the center pin*
    4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
    1. *Short JP1 "U" option to the center pin*

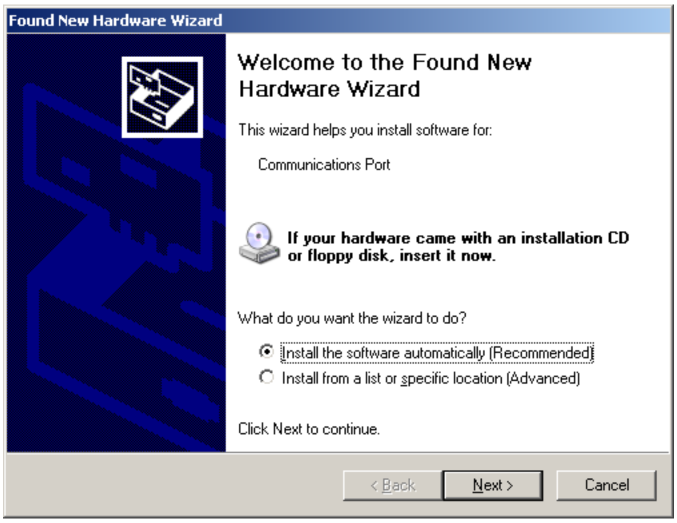
2. *Short JP2 "U" option to the center pin*
3. *Short JP3 "U" option to the center pin*
4. *Short JP4*

- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

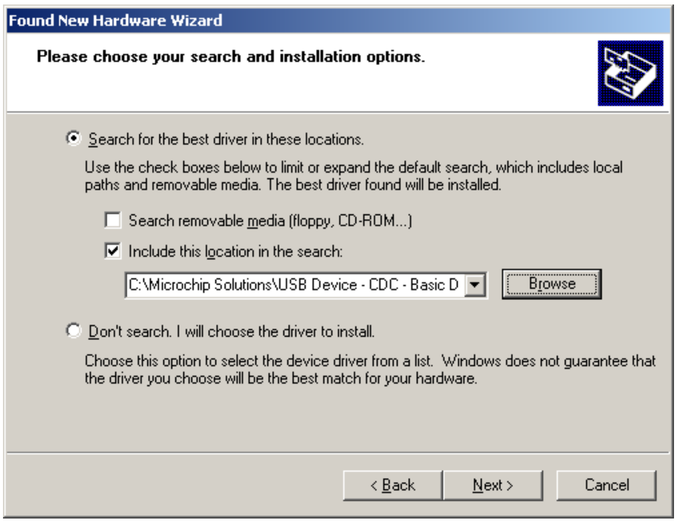
4.8.3 Running the Demo

This demo allows the device to appear like a serial (COM) port to the host. This demo will take data sent over the USB CDC interface and send it on the UART of the microcontroller.

In order to run this demo first compile and program the target device. Attach the device to the host with the USB cable. Also connect the RS232 port of the demo board to a computer. This computer and be the same computer as the USB connection or it can be a different computer. If the host is a PC and this is the first time you have plugged this device into the computer then you may be asked for a .inf file.



Select the “Install from a list or specific location (Advanced)” option. Point to the “<Install Directory>\USB Device - CDC – Serial Emulator\inf\” directory



Once the device is successfully installed, open up a terminal program, such as hyperterminal. Select the appropriate COM port for the USB virtual COM port. On most machines this will be COM5 or higher. On the computer where the RS232 cable it attached, open a second terminal program. Select the hardware COM port associated with that computer. Please insure that the baud rate for both terminal windows is the same.

Once everything is configured correctly, typing a key in one terminal window will result in the same data to show up in the second terminal window.

Note: Some terminal programs, like hyperterminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again in order to reconnect to the device.

4.9 Device - Composite - HID + MSD Demo

This document describes how to run the Composite HID + MSD demo. Composite devices allow a single USB peripheral to appear like two different devices/function on the computer.

4.9.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1

PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.9.2 Configuring the Demo

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board

3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.9.3 Running the Demo

This demo uses the selected hardware platform as both a flash drive using the internal flash as storage and a custom HID device. It will appear to the computer as if two USB devices were attached.

For details how to run the demo for each of the functions please see the respective getting started documents: “USB Device – Mass Storage – Internal Flash” (see page 90) and “USB Device – HID – Simple Custom Demo” (see page 76).

NOTE: the “USB Device – HID – Simple Custom Demo” application is expecting a PID of 0x003F. Because these two different applications can’t have the same PID, this demo uses PID 0x0054. The PC application that corresponds to this application is only looking for devices with PID 0x003F so the PC application will not be able connect to this demo without modification. Modify the MY_DEVICE_ID field to "Vid_04d8&Pid_0054" and recompile

NOTE: that for the PIC24F Starter Kit 1 the pushbutton functionality is not implemented.

4.10 Device - Composite - MSD + WinUSB Demo

This document describes how to run the Composite WinUSB + MSD demo. Composite devices allow a single USB peripheral to appear like two different devices/function on the computer.

4.10.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.10.2 Configuring the Demo

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*

- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.10.3 Running the Demo

This demo uses the selected hardware platform as both a flash drive using the internal flash as storage and WinUSB class USB device. It will appear to the computer as if two USB devices were attached.

For details how to run the demo for each of the functions please see the respective getting started documents: “USB Device – Mass Storage – Internal Flash” (see page 90) and “USB Device – WinUSB – Generic Driver Demo” (see page 113).

For PIC24F Starter Kit 1, “Get pushbutton State” functionality is not implemented.

For PIC18F Starter Kit 1, “Toggle LED” functionality is not implemented.

4.11 Device - HID Digitizer Demos

These are examples of HID digitizers. There are single, and various multi-point touch examples.

4.11.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1

PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.11.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.11.3 Running the Demo

These demos use the selected hardware platform as a USB HID class digitizer device. The Single-Touch demo is a HID class pen digitizer demo, which emulates a pen digitizer touch screen capable of sensing a single contact point. The Multi-Touch demo emulates a touch sensitive touch screen, capable of sensing two simultaneous contact points. The multi-touch demo can potentially be expanded to support additional simultaneous contacts (by modifying the HID report descriptor), however, the standard built in gestures that are recognized by the Microsoft Windows 7 platform only use one or two contacts.

To use the Single-Touch pen digitizer demo, plug the demo board into a free USB port on a Windows Vista or Windows 7 machine. The device should automatically enumerate as a HID class pen digitizer device, and certain additional functions and capabilities built into the operating system will become activated. No manual USB driver installation is necessary, as the built in HID class drivers are used for this device.

To use the Multi-Touch digitizer demo, plug the demo board into a free USB port on a Windows 7 machine. Windows 7 has significantly more "Windows Touch" capabilities than Vista. Although the device will enumerate and provide limited functionality on Windows Vista, multi-touch gestures will not be recognized unless run on Windows 7.

Since the standard demo boards that these demos are meant to be run on do not have an actual touch sensitive contact area, the firmware demos emulate the data that would be generated by a real touch screen. Both demo projects use a single user pushbutton. By pressing the button, the firmware will send a flurry of USB packets to the host, which contain contact position data that is meant to mimic an actual "gesture" of various types. Each subsequent press of the pushbutton will advance the internal state machine, and cause the firmware to send a gesture to the PC.

To use the demos, it is best to have Microsoft Internet Explorer installed on the machine (although some demo functions can be observed using the pen flick practice area available from the control panel). The latest versions of Internet Explorer (when run on the proper OS: preferably Windows 7, but some function on Windows Vista) supports recognition and use of certain basic gestures, such as "back", "forward", as well as certain scroll and zoom operations. To see a full detailed description of how best to use Internet Explorer or the pen flick practice area, see the detailed comments at the top of SingleTouch.c file (for the Single Touch pen digitizer demo: "<Install Directory>\USB Device - HID - Digitizers\Single Touch – Firmware"). For details on how best to use and what to expect with the multi-touch demo, see the detailed comments at the top of the MultiTouch.c file (<Install Directory>\ "USB Device - HID - Digitizers\Multi Touch – Firmware").

Other Info: Windows 7 adds support for Windows messages such as "WM_GESTURE" and "WM_TOUCH". These messages can be used to help build customized "touch enabled" PC applications. Documentation for these messages can be found in MSDN.

The following Microsoft developer blog contains useful additional information relating to Windows Touch:

http://blogs.msdn.com/e7/archive/2009/03/25/touching-windows-7.aspx

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.12 Device - HID Joystick Demo

This demo shows how to create a USB joystick

4.12.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited

capacity or has the hardware feature emulated in software.

3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.12.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*

- *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
- *Open jumpers J6, J7, J8, J9, and J10*

- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

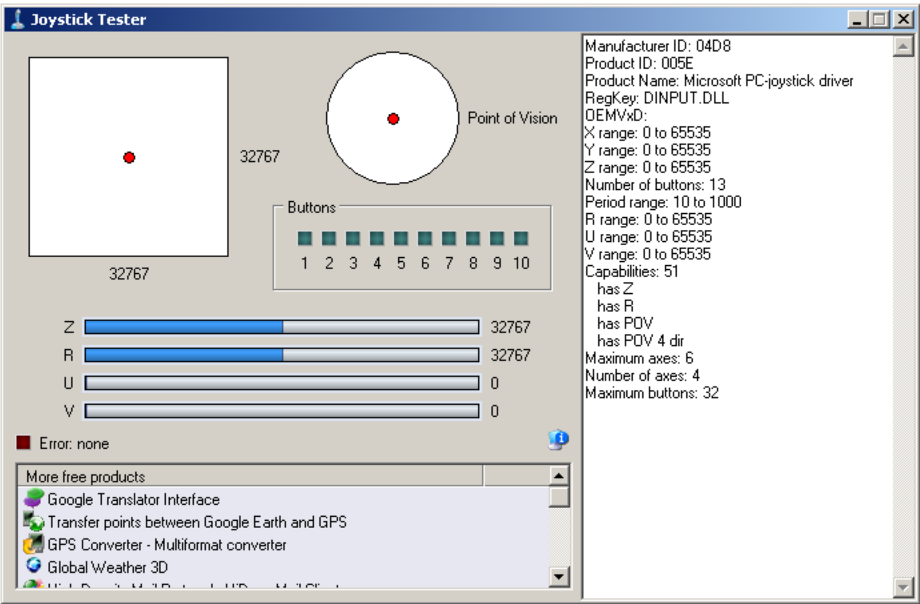
- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

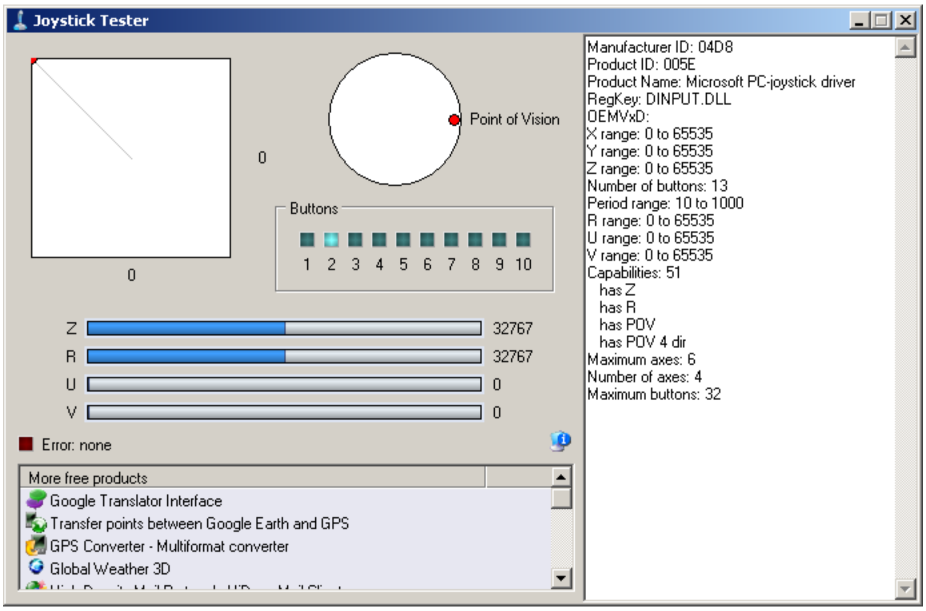
4.12.3 Running the Demo

This demo uses the selected hardware platform as a USB Joystick. To test the joystick feature, go to the “<Install Directory\USB Device – HID - Joystick” directory and open the JoystickTester.exe:



Pressing the button will cause the device to:

- Indicate that the “x” button is pressed, but none others;
- Move the hat switch to the "east" position;
- Move the X and Y coordinates to the their extreme values;



Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.13 Device - HID Keyboard Demo

This example shows how to create a USB keyboard and how to send data to the host.

4.13.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.13.2 Configuring the Hardware

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.

2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*

- *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.13.3 Running the Demo

This demo uses the selected hardware platform as a USB keyboard. Before pressing the button, select a window in which it is safe to type text freely. Pressing the button will cause the device to print a character on the screen.

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.14 Device - HID Mouse Demo

This demo is a simple mouse demo that causes the mouse to move in a circle on the screen.

4.14.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.14.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*

- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.14.3 Running the Demo

This demo uses the selected hardware platform as a USB mouse. Before connecting the board to the computer through the USB cable please be aware that the device will start moving the mouse cursor around on the computer. There are two ways to stop the device from making the cursor to continue to move. The first way is to disconnect the device from the computer. The second is to press the correct button on the hardware platform. Pressing the button again will cause the mouse cursor to start moving in a circle again.

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.15 Device - HID Simple Custom Demo

4.15.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.

3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.15.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*

  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.15.3 Running the Demo

This demo uses the selected hardware platform as a HID class USB device, but uses the HID class for general purpose I/O operations. Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol is however quite flexible, and can be adapted and used to send/receive general purpose data to/from a USB device. Using the HID class for general purpose I/O operations is quite advantageous, in that it does not require any kind of custom driver installation process. HID class drivers are already provided by and are distributed with common operating systems. Therefore, upon plugging in a HID class device into a typical computer system, no user installation of drivers is required, the installation is fully automatic.

HID devices primarily communicate through one interrupt IN endpoint and one interrupt OUT endpoint. In most applications, this effectively limits the maximum achievable bandwidth for full speed HID devices to 64kBytes/s of IN traffic, and 64kBytes/s of OUT traffic (64kB/s, but effectively “full duplex”).

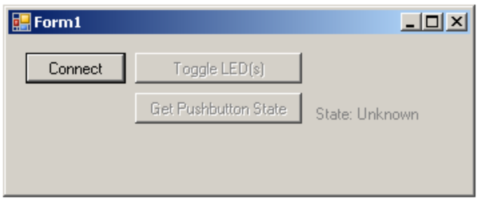
The GenericHIDSimpleDemo.exe program, and the associated firmware demonstrate how to use the HID protocol for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the GenericHIDSimpleDemo.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft’s website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for GenericHIDSimpleDemo.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the “<Install Directory>\ USB Device - HID - Custom Demos\Generic HID - Simple Demo - PC Software” directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from Microsoft’s website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft’s instructions for integrating it with the development environment.

It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the GenericHIDSimpleDemo.exe program. These are only required if the source code will be modified or compiled.

To launch the application, simply double click on the executable “GenericHIDSimpleDemo.exe” in the “<Install Directory>\USB Device - HID - Custom Demos” directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

In order to begin sending/receiving packets to the device, you must first find and “connect” to the device. As configured by default, the application is looking for HID class USB devices with VID = 0x04D8 and PID = 0x003F. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. If you plug in a USB device programmed with the correct precompiled .hex file, and hit the “Connect” button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the HID class USB peripheral device. The data will arrive on the interrupt OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the “Toggle LED(s)” command, and should respond appropriately by controlling the LED(s) on the demo board.

The “Get Pushbutton State” button will send one packet of data over the USB to the peripheral device (to the interrupt OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

Demo Board (click link for board information)	**Button**
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6[1]
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A[2]
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3[1]
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3[1]
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3[1]
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3[1]
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

The PC then requests a packet of data from the device (which will be taken from the interrupt IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the "Get Pushbutton State" button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

Running the demo on an Android v3.1+ device

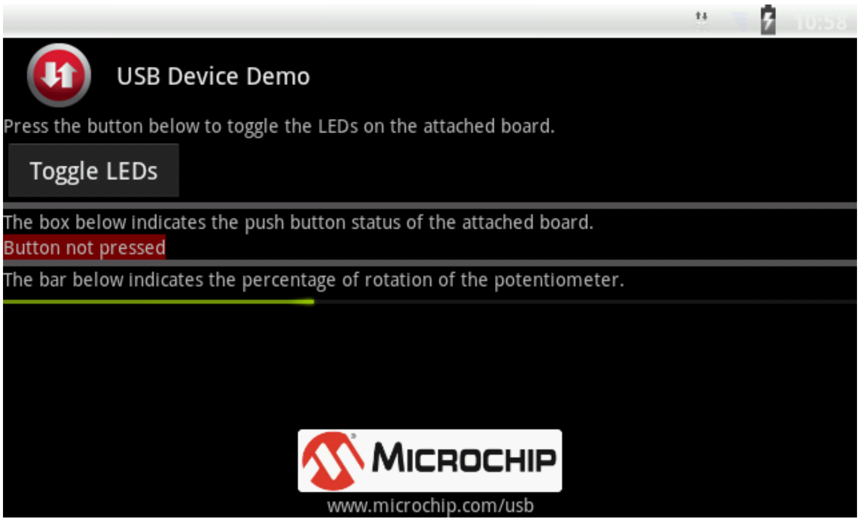
There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

1. The demo application can be downloaded from Microchip's Android Marketplace page:
   https://market.android.com/developer?pub=Microchip+Technology+Inc

2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:
   - http://developer.android.com/index.html
   - http://developer.android.com/sdk/index.html
   - http://developer.android.com/sdk/installing.html

While there are no devices attached, the Android application will indicate that no devices are attached.

When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



4.16 Device - LibUSB Generic Driver Demo

4.16.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2

PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.16.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board

4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.16.3 Running the Demo

When running this demo, the following push buttons are used. Please refer to each of the following sections for a description of how to run the demo on various operating systems:

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1

PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.16.3.1 Windows

This demo uses the selected hardware platform as a Libusb class USB device. Libusb-Win32 is a USB Library for the Windows operating systems. The library allows user space applications to access any USB device on Windows in a generic way without writing any line of kernel driver code. This driver allows users to have access to interrupt, bulk, and control transfers directly.

The SimpleLibUSBDemo.exe program and the associated firmware demonstrate how to use the Libusb device drivers for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the SimpleLibUSBDemo.exe executable, you will need to have the Microsoft® .NET Framework Version 3.5 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for SimpleLibUSBDemo.exe file was created in Microsoft Visual C++® 2008 Express Edition. The source code can be found in the "<Install Directory>\ USB Device - Libusb - Generic Driver Demo\ Libusb Simple Demo - Windows Application\Libusb Simple Demo - PC Application - MS VC++ 2008 Express" directory. Microsoft currently distributes Visual C++ 2008 Express Edition for free, and can be downloaded from Microsoft's website.

To launch the application, simply double click on the executable "SimpleLIbusbDemo.exe" in the "<Install Directory>\USB Device - Libusb - Generic Driver Demo\Windows Application" directory. A window like that shown below should appear:

If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 3.5 Redistributable Package has not yet been installed. Please install it and try again.

In order to begin sending/receiving packets to the device, you must first find and "connect" to the device. As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0204. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. To run the demo program the USB device with the correct precompiled .hex file. If you are connecting the device for the first time, Windows XP and Vista pop up a window asking you to install the driver for the device. For Windows 7, the device will be listed as an unknown USB device in the device manager and will need its driver updated. When asked for the driver or updating through the device manager, point it to the inf file provided along with the demo. Windows takes a while to install the driver for the USB device that is just plugged in. Make sure the Device manager lists the USB device under the 'Libusb Demo Devices'.

If a different VID/PID combination from the default is desired, then the descriptors in the firmware must be changed as well as the inf file. The easiest way to change the inf file is to use the utility provided with the LibUSB download for windows on the LibUSB website. This utility can create a new inf file based on a connected device. So make sure to change the VID/PID combination first in the firmware, connect the device, and then run the inf file creator utility. After completing the utility, a new signed driver with inf file is created.

Once the driver is installed hit the "Connect" button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the Custom class USB peripheral device. The data will arrive on the Bulk OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the "Toggle LED(s)" command, and should respond appropriately by controlling the LED(s) on the demo board.

The "Get Pushbutton State" button will send one packet of data over the USB to the peripheral device (to the Bulk OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the Bulk IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the “Get Pushbutton State” button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

In order to build the application, copy the file <libusb-win32 unzipped folder>\ libusb-win32-device-bin-0.1.12.1\lib\msvc\libusb.lib and paste to ‘lib’ folder of the VC++. Also copy the file

<libusb-win32 unzipped folder>\ libusb-win32-device-bin-0.1.12.1\ include\usb.h and paste to the “<Install Directory>\USB Device - Libusb - Generic Driver Demo\Windows Application\Microsoft VC++ 2008 Express\SimpleLibusbDemo’ folder.

4.16.3.2 Linux

The SimpleLibUSBDemo program and the associated firmware demonstrate how to use the Libusb device drivers for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the SimpleLibUSBDemo executable, you will need to have the libusb 0.1 driver installed on your computer. The libusb can be downloaded from sourceforge.net.

The source code for SimpleLibUSBDemo.exe file was created using QT3 Designer. The source code can be found in the “<Install Directory>\ USB Device - Libusb - Generic Driver Demo\Libusb Simple Demo - Linux Application\ Libusb Simple Demo - Linux Application -QT3” directory.

To launch the application, open the Terminal and navigate to the “<Install Directory>\USB Device - LibUSB - Generic Driver Demo\Linux Application” directory and execute the following commands

1. chmod a+x SimpleLibusbDemo_Linux (This command gives executable right to the file on this Linux computer
2. sudo ./SimpleLibusbDemo_Linux.

Enter the Super user password when requested. A window like that shown below should appear:

In order to begin sending/receiving packets to the device, you must first find and “connect” to the device. As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0204. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. To run the demo program the USB device with the correct precompiled .hex file. If you are connecting the device for the first time, Windows pops up a window asking you to install the driver for the device. When asked for the driver point it to the inf file provided along with the demo. Windows takes while to install the driver for the USB device that is just plugged in. Open the Device manager and ensure that the USB device is listed under the ‘Libusb Demo Devices’. Once the driver is installed hit the “Connect” button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the Custom class USB peripheral device. The data will arrive on the Bulk OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the “Toggle LED(s)” command, and should respond appropriately by controlling the LED(s) on the demo board.

The “Get Pushbutton State” button will send one packet of data over the USB to the peripheral device (to the Bulk OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the Bulk IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the “Get Pushbutton State” button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

In order to build the application navigate to the “<Install Directory>\USB Device - LibUSB - Generic Driver Demo\Linux Application\Qt3” directory and execute the command “make”.

4.16.3.3 Android 3.1+

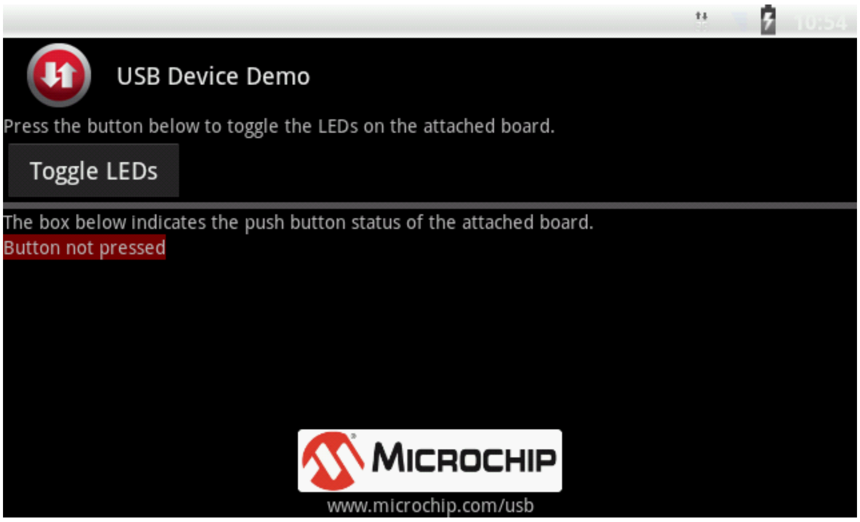
There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

1. The demo application can be downloaded from Microchip's Android Marketplace page: https://market.android.com/developer?pub=Microchip+Technology+Inc

2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:
   - http://developer.android.com/index.html
   - http://developer.android.com/sdk/index.html
   - http://developer.android.com/sdk/installing.html

While there are no devices attached, the Android application will indicate that no devices are attached.

When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



4.17 Device - Mass Storage - Internal Flash Demo

This demo uses the selected hardware platform as an drive on the computer using the internal flash of the device as the drive storage media. Connect the hardware platform to a computer through a USB cable.

The device should appear as a new drive on the computer named “Drive Name”. The volume label or file information can be changed in the Files.c file located in the project directory.

4.17.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2

PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.17.2 Configuring the Demo

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*

    - *Set switch S1 to the "PGX1" setting*
    - *Short J1 pin 1 (marked "POT") to the center pin*
    - *Short J2 pin 1 (marked "Temp") to the center pin*
    - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
    - *Short JP1 "U" option to the center pin*
    - *Short JP2 "U" option to the center pin*
    - *Short JP3 "U" option to the center pin*
    - *Short JP4*
- *PIC24EP512GU810 PIM*
    - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
    - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
    - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
    - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
    - *Open J10*
    - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.17.3 Running the Demo

This demo uses the selected hardware platform as an drive on the computer using the internal flash of the device as the drive storage media. Connect the hardware platform to a computer through a USB cable.

The device should appear as a new drive on the computer named “Drive Name”. The volume label or file information can be changed in the Files.c file located in the project directory.

4.17.3.1 Troubleshooting

Issue 1: The device appears correctly in the device manager, but no new drive letters appear on a Windows® operating system based machine.

Solution: See Microsoft knowledge base article 297694: http://support.microsoft.com/kb/297694

If there is a drive letter conflict (ex: because a network drive has been mapped to a letter low in the alphabet), on some operating systems the newly attached USB drive may not appear. If this occurs, either obtain the hotfix from Microsoft, or remap the conflicting mapped network drive to a letter at the end of the alphabet (ex: Z:).

Issue 2: The device enumerates correctly and I can access the new drive. Even though the drive is not full yet, when I try to write to the drive, I get an error message something like, “Cannot copy (some name): The directory or file cannot be created.”

Solution: In order to copy new files onto the drive volume, both the file contents themselves must be copied to the drive, and the FAT table must also be updated in order to accommodate the new file name and path. Even if the drive has plenty of free space available, the FAT table may have reached its limit. In order to keep the default demos small, the FAT table is configured to be only 512 bytes long. This is not very large, and can easily be exceeded, especially if the files on the drive have long file names. In order to use the remaining space available on the drive, it is recommended to keep the individual file names as short as possible to minimize their size in the FAT table. Alternatively, the firmware can be modified so that the FAT table is larger, and therefore able to accommodate more file name and path characters.

Issue 3: When I try to format the drive, I get an error message and the drive does not get formatted properly.

Solution: By default, common Windows based operating systems will try to place a large FAT table on the newly formatted disk (larger than the default 512 bytes of the demo firmware). If the FAT table is larger than the total drive space, the drive cannot be formatted. In order to successfully format the drive, an alternative method of formatting will be needed that places a smaller FAT table on the drive. For example, the drive can be effectively reformatted by reprogramming the microcontroller with the original HEX file. Alternatively, if the firmware is modified to increase the total drive space, the Windows operating system managed FAT table may be able to fit. Unfortunately, this will shrink the effective drive size, making less of it available for actual file data.

Issue 4: When I format the drive, the drive size shrinks.

Solution: See the solution to issue #3 above.

4.18 Device - Mass Storage - SD Card Reader

This demo shows how to implement a simple SD card reader

4.18.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	1, 2

PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	1, 2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	1, 2
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 3
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 3
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1, 3
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1, 3
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1, 3
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1, 3
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1, 3

Notes:

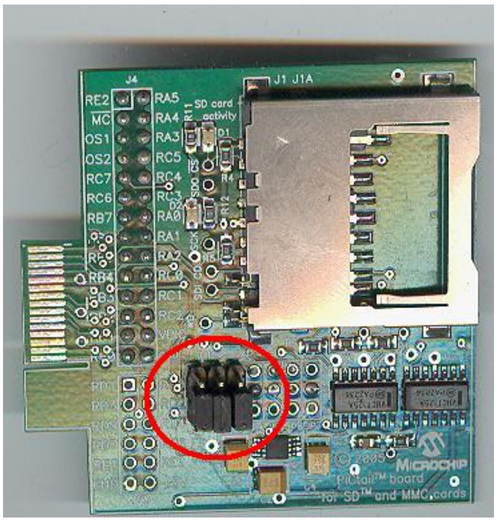
1. These boards require the SD Card PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.18.2 Configuring the Demo

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. On the SD Card PICTail™ Plus board, short JP1, JP2, and JP3 on the side farthest from the SD Card holder. Depending on the revision of the board you have the silk-screen on the board may incorrectly label the top as the “HPC-EXP” setting. Please ignore this silk screen and place the jumpers as described above and seen below.



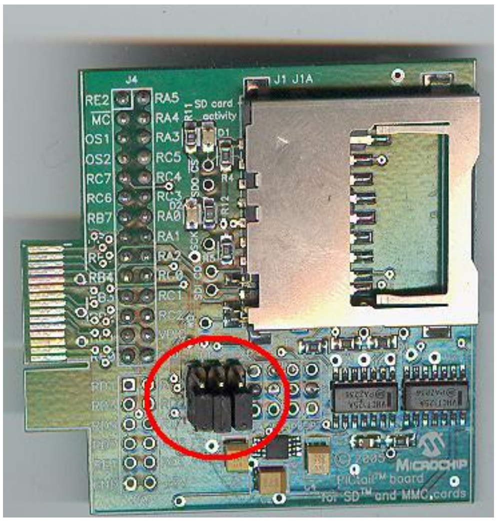
3. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC18F46J50 Plug-In-Module:*
    1. *Short JP2 such that the "R" and the "U" options are shorted together.*
    2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F47J53 Plug-In-Module:*
    1. *Short JP2 such that the "R" and the "U" options are shorted together.*
    2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F87J50 Plug-In-Module:*
    1. *Short JP1 such that the "R" and the "U" options are shorted together.*
    2. *Short JP4. This allows the demo board to be powered through the USB bus power.*
    3. *Short JP5. This enabled the LED operation on the board.*

<u>Explorer 16 Based Demos</u>

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. On the SD Card PICTail™ Plus board, short JP1, JP2, and JP3 on the side farthest from the SD Card holder. Depending on the revision of the board you have the silk-screen on the board may incorrectly label the top as the “HPC-EXP” setting. Please ignore this silk screen and place the jumpers as described above and seen below.



7. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
    1. *Set switch S1 to the "PGX1" setting*
    2. *Short J1 pin 1 (marked "POT") to the center pin*
    3. *Short J2 pin 1 (marked "Temp") to the center pin*
    4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
    1. *Short JP1 "U" option to the center pin*
    2. *Short JP2 "U" option to the center pin*
    3. *Short JP3 "U" option to the center pin*

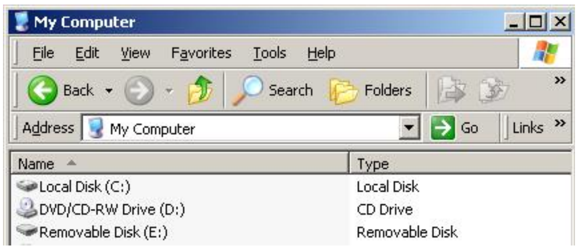
4. *Short JP4*

- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.18.3 Running the Demo

Connect the hardware platform to a computer through a USB cable. If the device was attached to the computer while the data logging occurred, you may need to remove the SD card from the card slot or disconnect and reconnect the device from the computer for the files to appear. Most computers are not expecting the files on an attached drive to change if they are not making the change so some operating systems will not look for additional drive changes.

The device should appear as a new drive on the computer named “Removable Drive”.



If no SD Card is inserted in the SD Card PICTail Plus, the following dialog will pop-up.

Once a compatible card is inserted in the card reader, files can be read, deleted, and manipulated like any other drive on the computer.

4.19 Device - Mass Storage - SD Card Data Logger

This demo shows how to data log to an SD card using the Microchip MDD file system and present the data to the PC using the Mass Storage data class to appear like an SD card reader.

4.19.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	1, 2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	1, 2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	1, 2
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 3
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 3
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1, 3
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1, 3
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1, 3
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1, 3
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1, 3

Notes:

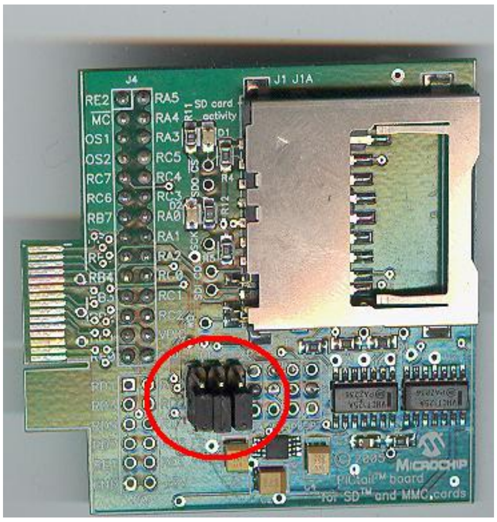
1. These boards require the SD Card PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.19.2 Configuring the Demo

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. On the SD Card PICTail™ Plus board, short JP1, JP2, and JP3 on the side farthest from the SD Card holder. Depending on the revision of the board you have the silk-screen on the board may incorrectly label the top as the “HPC-EXP” setting. Please ignore this silk screen and place the jumpers as described above and seen below.



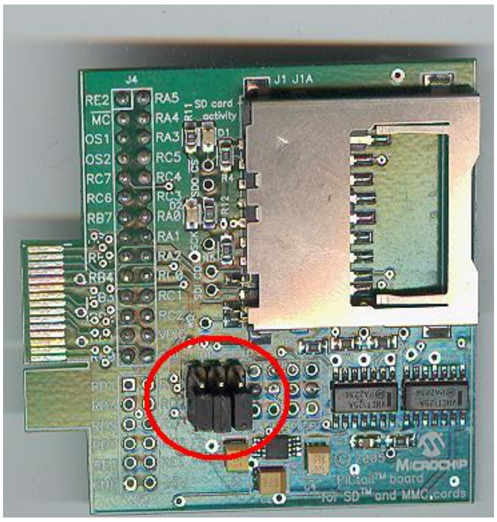
3. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC18F46J50 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F47J53 Plug-In-Module:*
  1. *Short JP2 such that the "R" and the "U" options are shorted together.*
  2. *Short JP3. This allows the demo board to be powered through the USB bus power.*
- *PIC18F87J50 Plug-In-Module:*
  1. *Short JP1 such that the "R" and the "U" options are shorted together.*
  2. *Short JP4. This allows the demo board to be powered through the USB bus power.*
  3. *Short JP5. This enabled the LED operation on the board.*

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. On the SD Card PICTail™ Plus board, short JP1, JP2, and JP3 on the side farthest from the SD Card holder. Depending on the revision of the board you have the silk-screen on the board may incorrectly label the top as the “HPC-EXP” setting. Please ignore this silk screen and place the jumpers as described above and seen below.



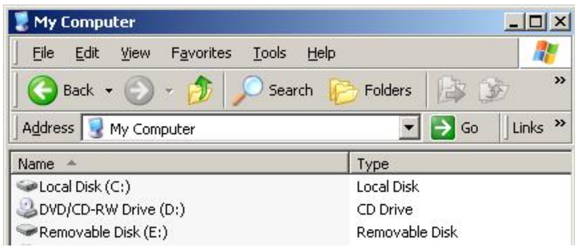
7. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.19.3 Running the Demo

Logging Data:

Make sure that there is a FAT or FAT32 formatted SD card in the card reader. This can be done by either connecting the device to a regulator SD card reader or connecting the hardware platform to the computer through the USB cable. The device should appear as a new drive on the computer named “Removable Drive”.



Power the demo board if it is not powered already.

Press and hold down the specified button below. This will cause the unit to soft detach from the computer (if it is attached) and start to log data to the card.

Demo Board (click link for board information)	Button
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6[1]
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3[1]
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3[1]
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3[1]
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3[1]

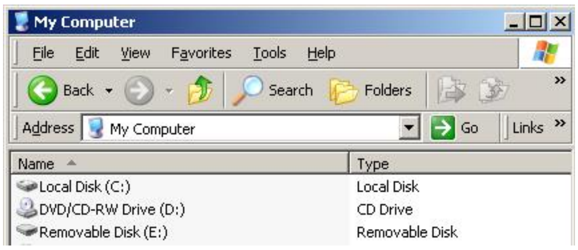
Notes:

1) This is the button number on the Explorer 16.

Reading the Data:

Connect the hardware platform to a computer through a USB cable. If the device was attached to the computer while the data logging occurred, you may need to remove the SD card from the card slot or disconnect and reconnect the device from the computer for the files to appear. Most computers are not expecting the files on an attached drive to change if they are not making the change so some operating systems will not look for additional drive changes.

The device should appear as a new drive on the computer named “Removable Drive”.

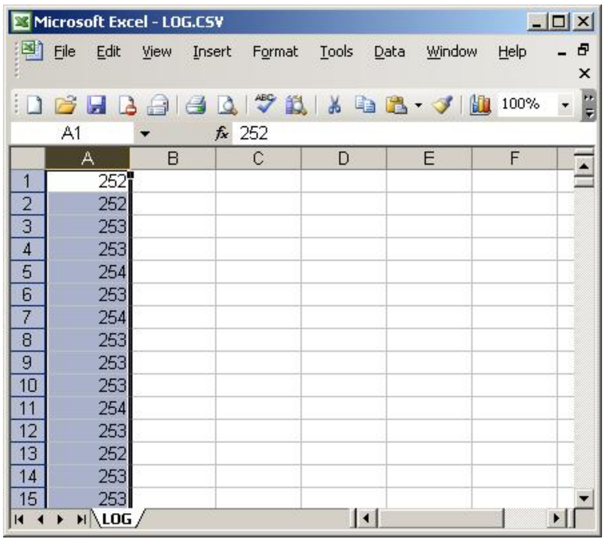


If no SD Card is inserted in the SD Card PICTail Plus, the following dialog will pop-up.

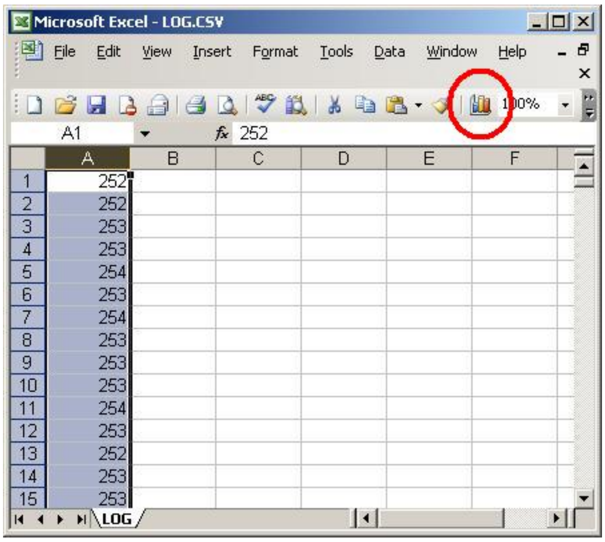
Once a compatible card is inserted in the card reader, files can be read, deleted, and manipulated like any other drive on the computer. If the instructions in the “Logging Data” are performed, there should be a “LOG.CSV” file on the card.

This file can be read by a simple text editor program or graphical/statistical programs, like Microsoft® Excel®.

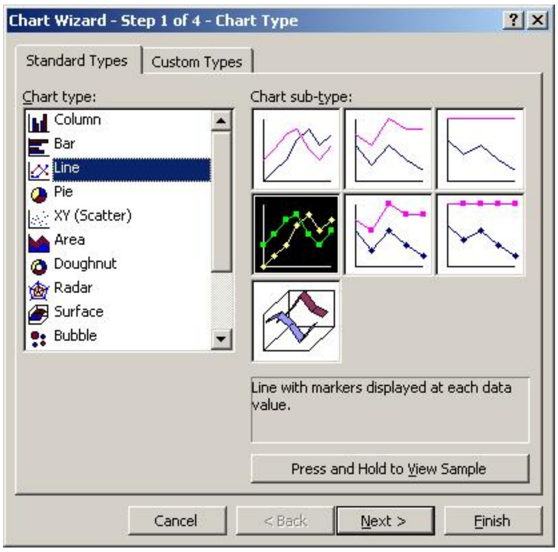
To plot the data in Excel, select the entire column that contains the data.



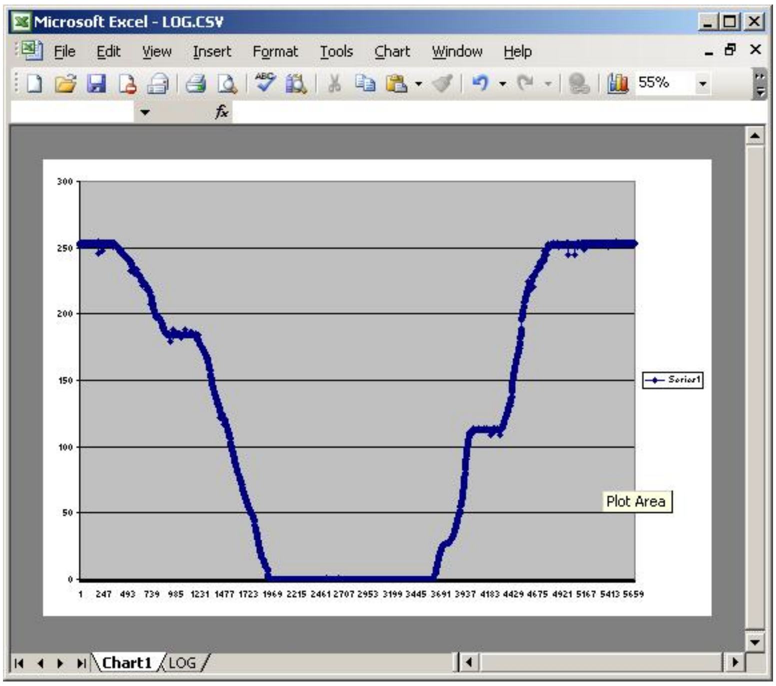
Click on the chart wizard button.



Select the “Line” option chart.



Click “Next” and “Finish” until the chart is generated.



Toubleshooting Tips:

Issue 1: The device appears correctly in the device manager, but no new drive letters appear on a Windows® operating system based machine.

Solution: See Microsoft knowledge base article 297694: http://support.microsoft.com/kb/297694

If there is a drive letter conflict (ex: because a network drive has been mapped to a letter low in the alphabet), on some operating systems, the newly attached USB drive may not appear. If this occurs, either obtain the hotfix from Microsoft, or remap the conflicting mapped network drive to a letter at the end of the alphabet (ex: Z:).

NOTE WHEN USING THE HID BOOTLOADER (for PIC18F87J50 PIM): The "USB Device - Mass Storage - SD Card reader" and "USB Device - Mass Storage - SD Card data logger" demos make use of the SD Card PICtail Daughter Board (Microchip® Direct: AC164122). This PICtail uses the RB4 I/O pin for the card detect (CD) signal, and is actively driven by the PICtail. The active drive overpowers the pull up resistor on the RB4 pushbutton (on the PIC18F87J50 FS USB Plug-In Module board). As a result, if the PIC18F87J50 is programmed with the HID bootloader, and an SD Card is installed in the socket when the microcontroller comes out of reset, the firmware will immediately enter the bootloader (irrespective of the RB4 pushbutton state). To exit the bootloader firmware, remove the SD Card from the SD Card socket, and tap the MCLR button. When the SD Card is not plugged in, the PICtail will drive the card detect signal (which is connected to RB4) logic high, which will enable the bootloader to exit to the main application after coming out of reset. Once the main application firmware is operating, the SD Card can be plugged in. The SD Card is "hot-swappable" and should be recognized by the host upon insertion. To avoid this inconvenience when using the bootloader with the PICtail, it is suggested to modify the bootloader firmware to use some other I/O pin for bootloader entry, such as RB0 (which has a pushbutton on it on the HPC Explorer board).

4.20 Device - MCHPUSB Generic Driver Demo

4.20.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1

PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.20.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*

- *Short J1 pin 1 (marked "POT") to the center pin*
- *Short J2 pin 1 (marked "Temp") to the center pin*
- *Short J3 pin 1 (marked "EEPROM CS") to the center pin*

- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.20.3 Running the Demo

When running this demo, the following push buttons are used. Please refer to each of the following sections for a description of how to run the demo on various operating systems:

Demo Board (click link for board information)	**Button**
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A(2)

PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

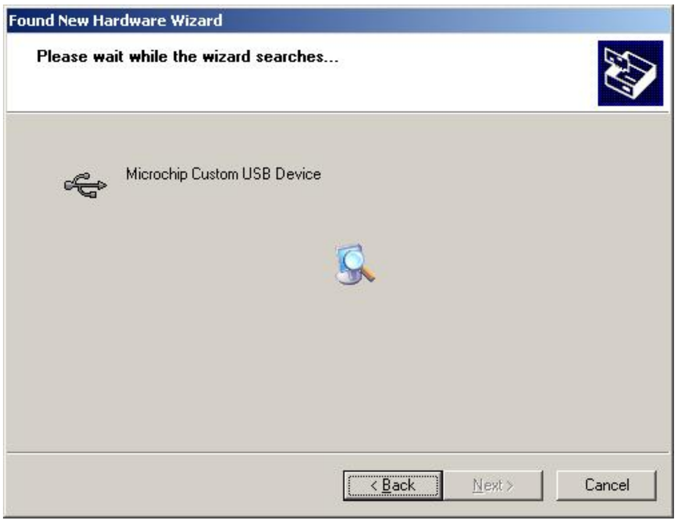
2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.20.3.1 Installing Windows Drivers

The generic driver (custom class) demo uses a custom class driver. Like any custom driver when first plugged into a computer, a driver needs to be installed. When the device is plugged in to the computer the following window will pop-up:

Continue by selecting either options and clicking next.

If the driver has been installed on the computer before the installation process may complete itself without further action.



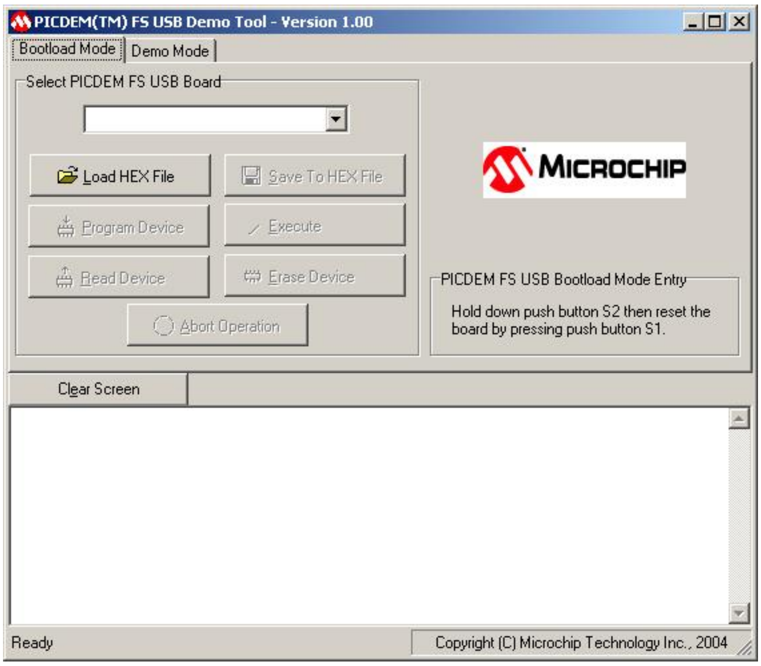
If the driver has not been installed before on the computer, then the driver will need to be installed. The Found New Hardware Wizard will be looking for a *.inf file with a matching VID/PID as the newly attached USB device. The driver can be found in the following location: "<Install Directory>\USB Tools\MCHPUSB Custom Driver\MCHPUSB Driver\Release". Point the install wizard to this directory. The install wizard should then continue and finally complete.

Some example PC applications which interface with the driver can be found at "<Install Directory>\USB Tools\MCHPUSB Custom Driver\Mpusbapi". PC applications can be written to either directly interface with the custom class USB driver (by using standard I/O functions like CreateFile(), ReadFile(), WriteFile(), CloseHandle()), or indirectly through the use of mpusbapi.dll. Mpusbapi.dll is a dynamic linked library file, which makes the process of interfacing with the custom class USB driver (and therefore, your USB device) somewhat simpler.

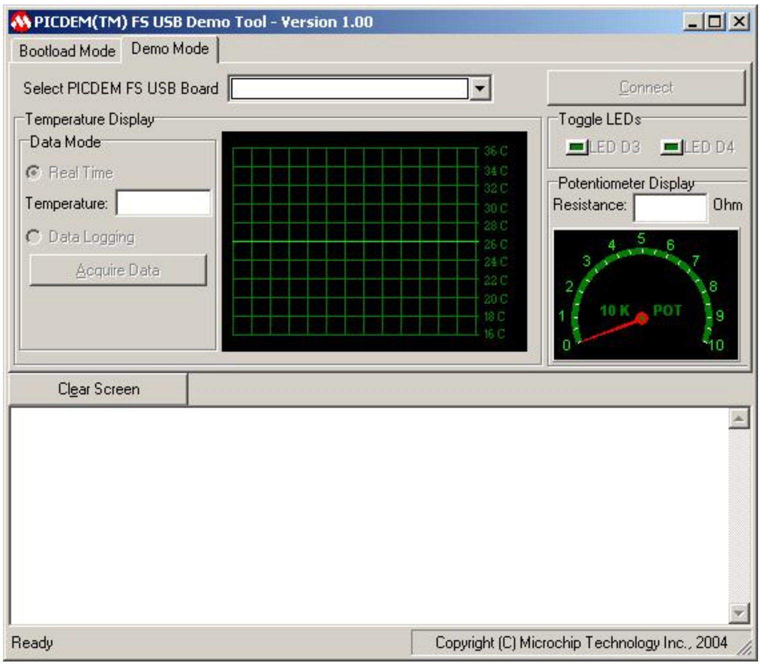
4.20.3.2 PDFSUSB

The example application can be found in the “<Install Directory>\USB Device - MCHPUSB - Generic Driver Demo\PC Software\Pdfsusb” directory.

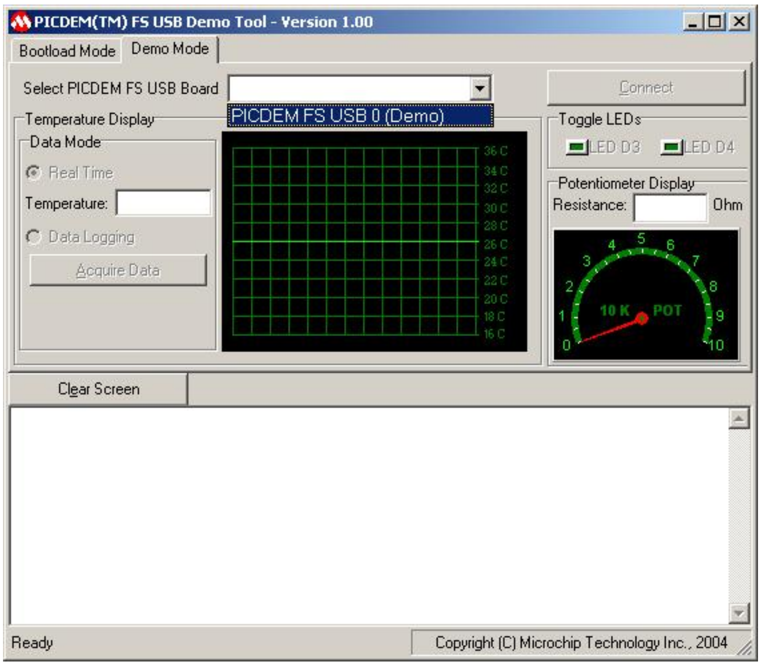
When the application is first launched it will look like the following.



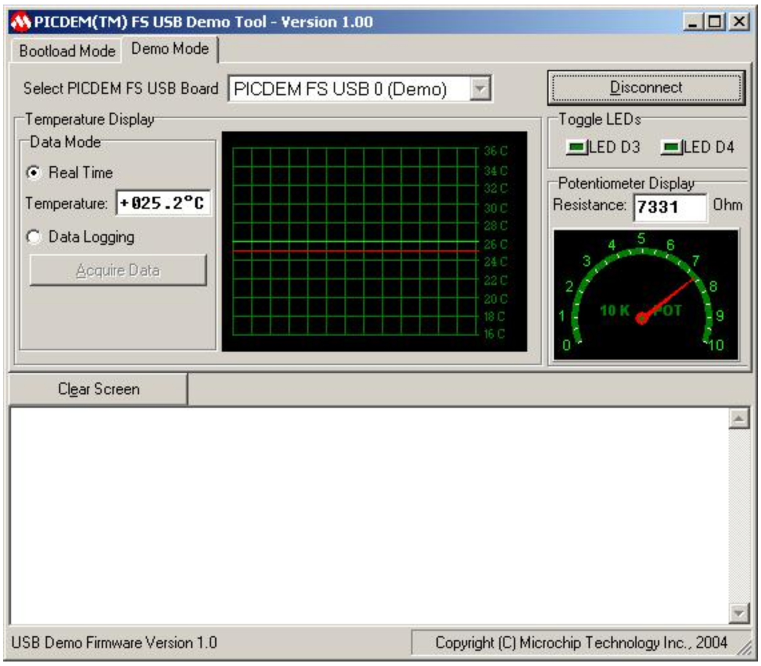
Select the “Demo Mode” tab.



In the listbox at the top of the application, select the “PICDEM FS USB…” option. If this option is not available then the device is either not connected to the computer, the driver was not installed correctly, or the firmware programmed into the device was not the correct project needed to interface with the generic driver.



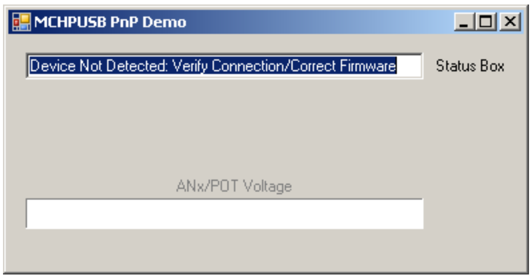
With the listbox selecting the "PICDEM FS USB…" click the connect button. Once the button is clicked the application should start reading the potentiometer and temperature data from the hardware. The application can also change the state of the LEDs. NOTE: the Low Pin Count USB Development Kit does not have an on board temperature sensor. This feature is not currently implemented. Clicking LED3 button will toggle LED D7 on the Explorer 16 board. Clicking LED4 button will toggle LED D8 on the Explorer 16 board. While using Explorer 16 and dsPIC33EP512MU810 or PIC24EP512GU810 PIM, the temperature sensor and potentiometer interface are not supported.



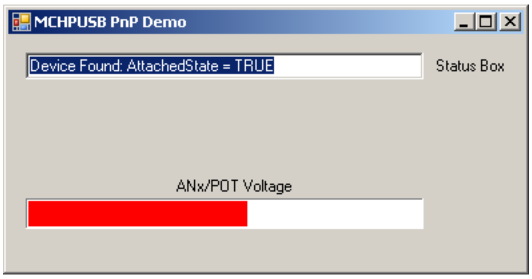
4.20.3.3 MCHPUSB PnP Demo

The example application can be found in the "<Install Directory>\USB Device - MCHPUSB - Generic Driver Demo\PC Software\Visual C++ 2005 Express" directory.

When the application is launched and the MCHPUSB custom device is not attached, it will look like the following:



Once the device is attached the application will reflect that the device is attached and look like the following. Moving the potentiometer will cause the status bar of the application to move to reflect the current value. While using Explorer 16 with dsPIC33EP512MU810 or PIC24EP512GU810 PIM, the potentiometer interface is not supported.



4.20.3.4 Running the Demo (Android v3.1+)

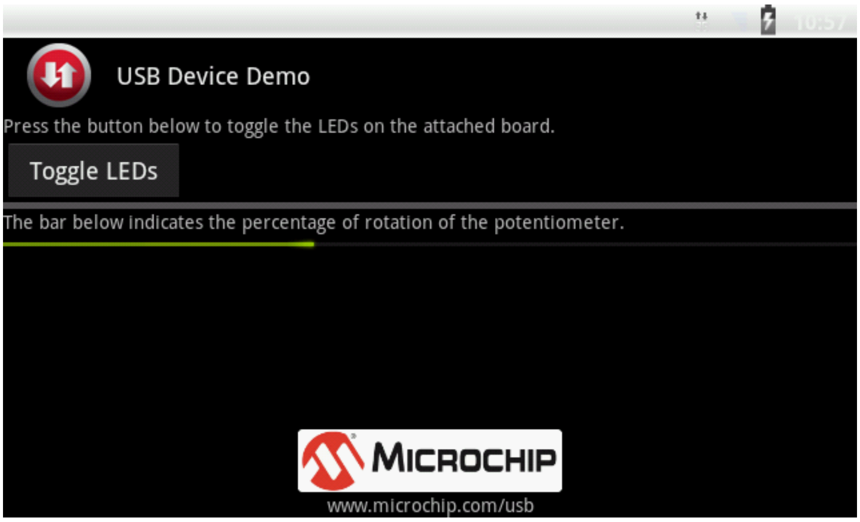
There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

1. The demo application can be downloaded from Microchip's Android Marketplace page: https://market.android.com/developer?pub=Microchip+Technology+Inc

2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:
   - http://developer.android.com/index.html
   - http://developer.android.com/sdk/index.html
   - http://developer.android.com/sdk/installing.html

While there are no devices attached, the Android application will indicate that no devices are attached.

When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



4.21 Device - WinUSB Generic Driver Demo

4.21.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1

PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.21.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*

  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.21.3 Running the Demo

When running this demo, the following push buttons are used. Please refer to each of the following sections for a description of how to run the demo on various operating systems:

Demo Board (click link for board information)	**Button**
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6(1)
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)
PIC24FJ256DA210 Development Board (see page 167)	S1

PIC24F Starter Kit (see page 168)	N/A(2)
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3(1)
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.21.3.1 Windows

This demo uses the selected hardware platform as a WinUSB class USB device. WinUSB is a vender specific driver produced by Microsoft for use with Windows® XP service pack 2 and Windows Vista® operating systems. This driver allows users to have access to interrupt, bulk, and control transfers directly.

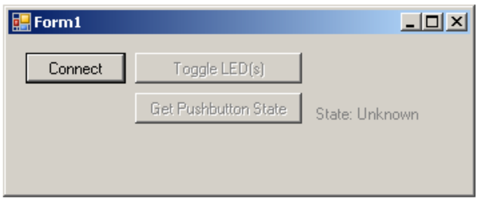
The SimpleWinUSBDemo.exe program, and the associated firmware demonstrate how to use the WinUSB device drivers for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the SimpleWinUSBDemo.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for SimpleWinUSBDemo.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the "<Install Directory>\ USB Device - WinUSB - Generic Driver Demo\WinUSB Simple Demo - PC Application - MS VC++ 2005 Express" directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from Microsoft's website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft's instructions for integrating it with the development environment.

It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the SimpleWinUSBDemo.exe program. These are only required if the source code will be modified or compiled.

To launch the application, simply double click on the executable "SimpleWinUSBDemo.exe" in the "<Install Directory>\USB Device - WinUSB - Generic Driver Demo" directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

In order to begin sending/receiving packets to the device, you must first find and "connect" to the device. As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0053. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. If you plug in a USB device programmed with the correct precompiled .hex file, and hit the "Connect" button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the HID class USB peripheral device. The data will arrive on the interrupt OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the "Toggle LED(s)" command, and should respond appropriately by controlling the LED(s) on the demo board.

Please see the "Running the Demo" (see page 115) section for information about specific board limitations. Not all demo features may be available on all boards.

The "Get Pushbutton State" button will send one packet of data over the USB to the peripheral device (to the interrupt OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the interrupt IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the “Get Pushbutton State” button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

4.21.3.2 Android v3.1+

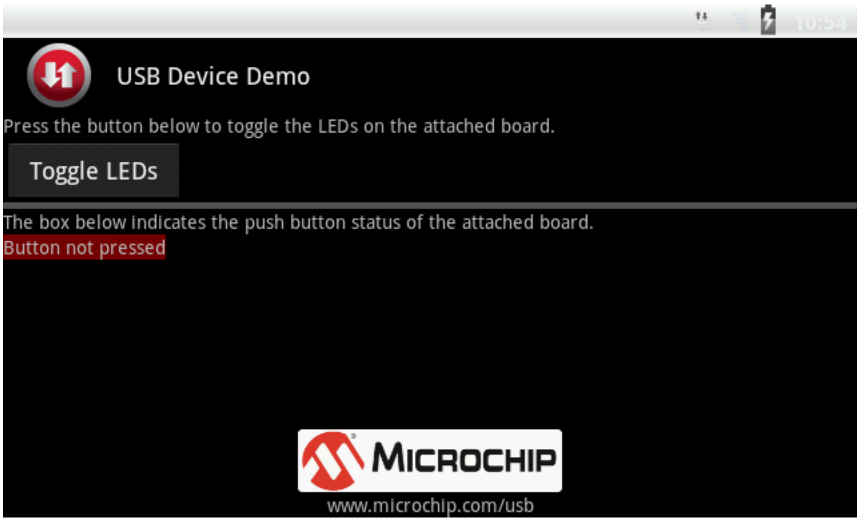
There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

1. The demo application can be downloaded from Microchip's Android Marketplace page:
   https://market.android.com/developer?pub=Microchip+Technology+Inc

2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:
   - http://developer.android.com/index.html
   - http://developer.android.com/sdk/index.html
   - http://developer.android.com/sdk/installing.html

While there are no devices attached, the Android application will indicate that no devices are attached.

When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



4.22 Device - WinUSB High Bandwidth Demo

4.22.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1

dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.22.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.

6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.22.3 Running the Demo

This demo uses the selected hardware platform as a WinUSB class USB device. WinUSB is a vender specific driver produced by Microsoft for use with Windows® XP service pack 2 and later operating systems. This driver allows users to have access to interrupt, bulk, and control transfers directly.

The HighBandwidthWinUSB.exe program, and the associated firmware demonstrate how to use the WinUSB device drivers for USB Bulk data transfers. Total Time taken to transmit the data & data transmission rate (Bytes/Sec) is shown in the GUI once the data transmission of 9,60,000 bytes is completed from the PC side.

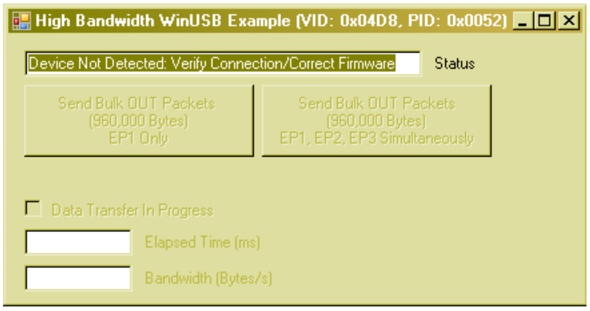
Before you can run the HighBandwidthWinUSB.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for HighBandwidthWinUSB.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the "<Install Directory>\ USB Device - WinUSB - High Bandwidth Demo\WinUSB High

Bandwidth Demo - PC Application - MS VC++ 2005 Express" directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from Microsoft's website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft's instructions for integrating it with the development environment.

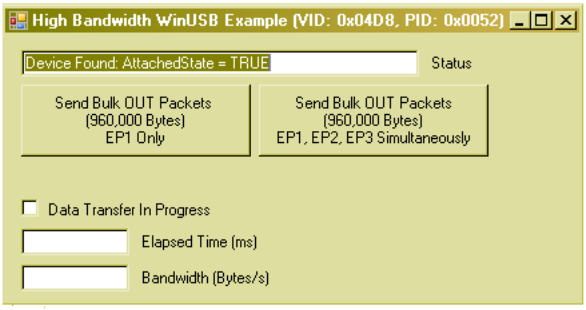
It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the HighBandwidthWinUSB.exe program. These are only required if the source code will be modified or compiled.

To launch the application, simply double click on the executable "HighBandwidthWinUSB.exe" in the "<Install Directory>\USB Device - WinUSB - High Bandwidth Demo" directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0052. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. Once the device flashed with corresponding firmware is connected to the PC, the below window appears:



Hitting the "Send Bulk OUT Packets" tab will transmit 960,000 bytes of data on the USB bus to the corresponding endpoints (EP1 Only or EP1,EP2, EP3 Simultaneously depending upon the button pressed in the GUI). Elapsed Time (ms) & Bandwidth (Bytes/Sec) are displayed in the GUI once the data transmission is complete.

4.23 Device - Personal Healthcare Device Class (PHDC) - Weight Scale Demo

4.23.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
Low Pin Count USB Development Kit (see page 160)	DM164127	
PICDEM FS USB (see page 161)	DM163025	
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	MA180024	
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	MA180029	
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	MA180021	
PIC18F Starter Kit (see page 165)	DM180021	
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	3
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.23.2 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No hardware related configuration or jumper setting changes are necessary.

PIC18F46J50 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F47J53 Plug-In-Module:

1. Short JP2 such that the "R" and the "U" options are shorted together.
2. Short JP3. This allows the demo board to be powered through the USB bus power.

PIC18F87J50 Plug-In-Module:

1. Short JP1 such that the "R" and the "U" options are shorted together.
2. Short JP4. This allows the demo board to be powered through the USB bus power.
3. Short JP5. This enabled the LED operation on the board.

PIC18F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.23.3 Running the Demo

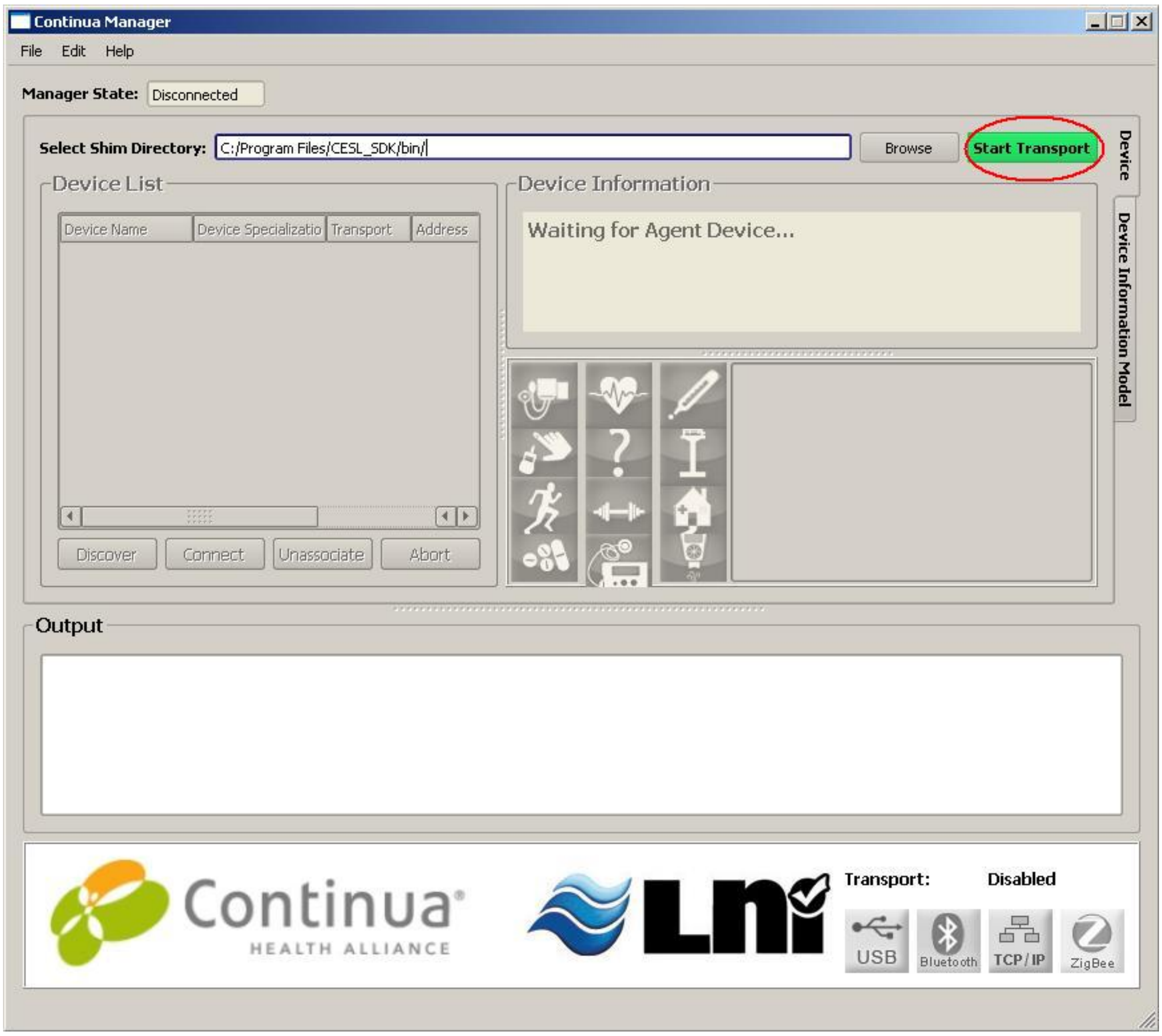
The user needs to install the Continua Manager GUI in order to see the measured data which would be transmitted from the device. To obtain the Continua manager GUI one needs to be a member of the Continua org. In order to run this demo first compile and program the target device. Attach the device to the host. If the host is a PC and this is the first time you have plugged this device into the computer then you may be asked for a .inf file. The .inf file is provided by Continua org. The demo user needs to be a member of the Continua org to access the CESL software and the Continua Agent USB driver files provided by them.

Select the “Install from a list or specific location (Advanced)” option. Point to the “<Install Directory>\USB\Device - PHDC - Weighing Scale\Driver and INF” directory.

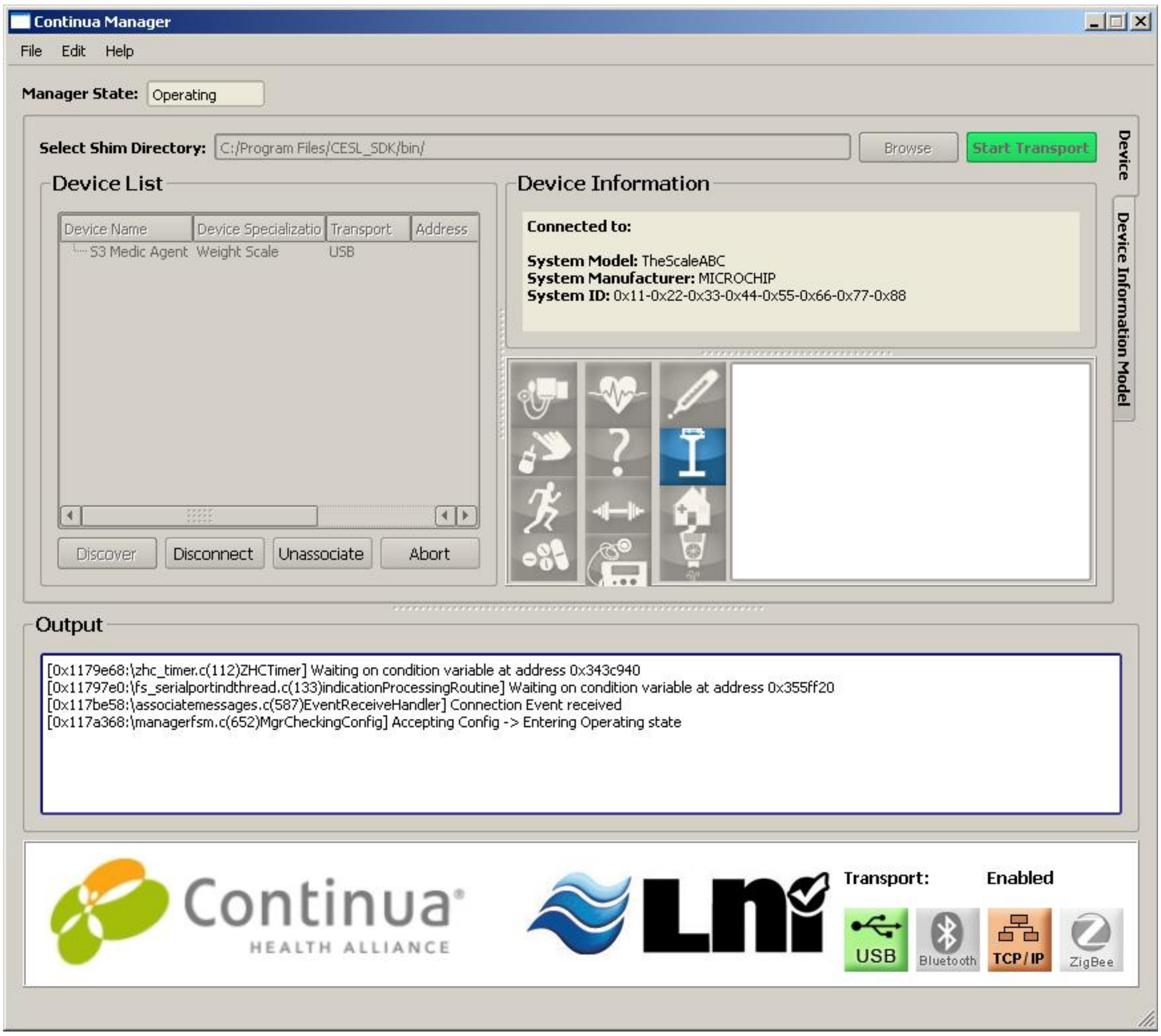
Once the device is successfully installed, open up the Continua Manager GUI. On the GUI the “Start Transport” button has to be pressed to connect with the device. The device now will be connected to manager.

It needs two push buttons on the device board to showcase the PHDC application. The PHDC application provided emulates a weigh scale. Each press on a push button performs specific tasks. One of the push button toggles between “Unassociated” and “Operating” state. The other one on assertion sends the measured data to the Continua Manager. The Weigh Scale measurement will be sent only when both Manager and agent are in Operating States.

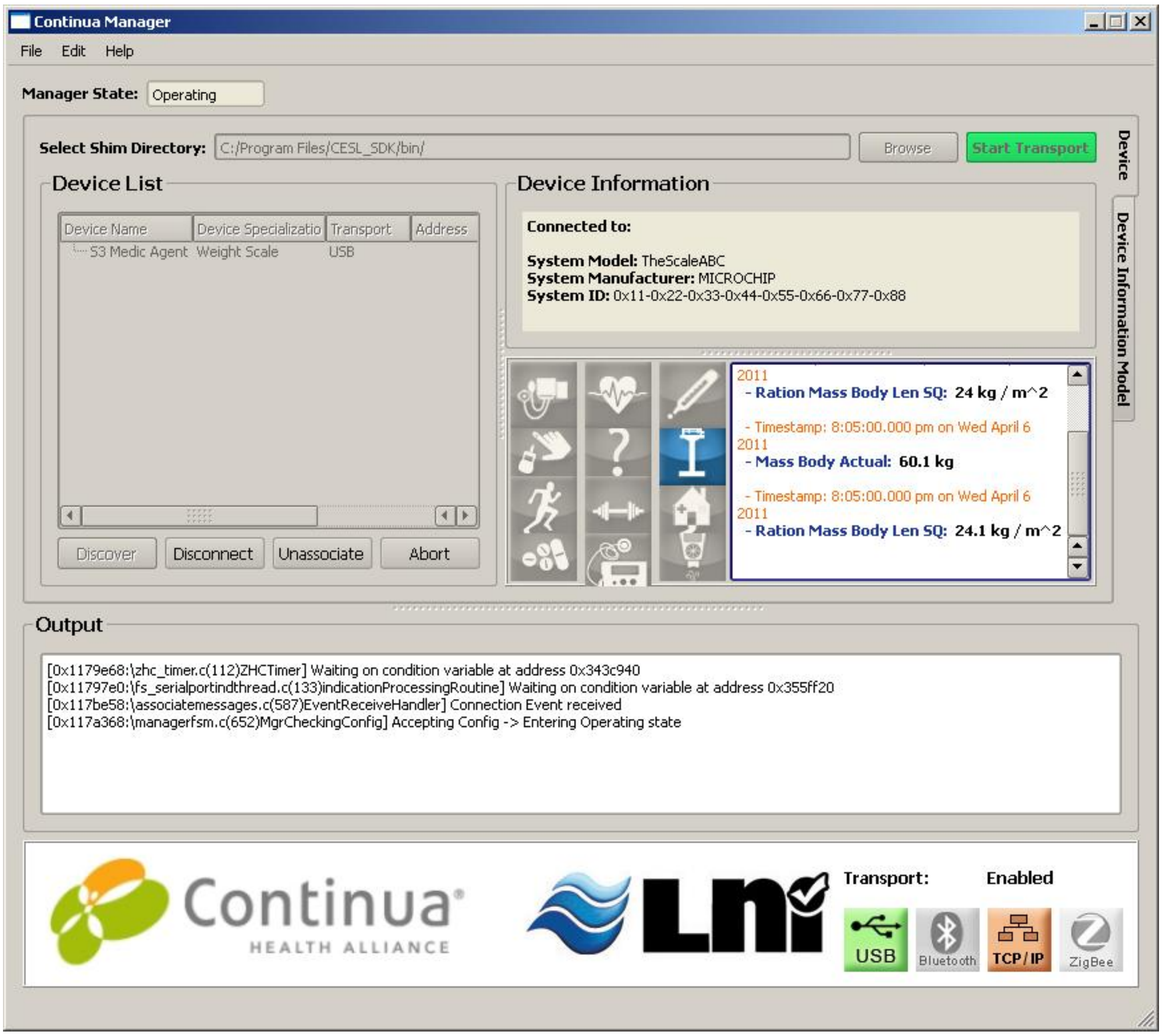
In some of the Microchip USB demo boards there is only one Pushbutton. In those demo boards with only one pushbutton the Agent (PHDC Device) Connects to the Manager when the pushbutton is pressed for the First time. When the pushbutton is pressed for the second time the device sends measured data to the device. When the pushbutton is pressed for the third time the device Disconnects from the Manager. Examples for demo boards with only one pushbutton are PIC18F46J50 PIM, PIC18F46J50 PIM, PIC18F87J50 PIM, PIC18F Starter Kit and Low Pin Count USB Development Kit.



Click on Start Transport Button



Press Push Button on the device to enter in Operating state. The Continua Manager indicates that it is in Operating State.



Press the pushbutton on the device to Send Measured Data to the Manager. The continua Manager displays the received data from the Agent (PHDC device).

When running this demo, the following push buttons are used. Please refer to each of the following sections for a description of how to run the demo on various operating systems:

Demo Board (click link for board information)	Button
Low Pin Count USB Development Kit (see page 160)	S1
PICDEM FS USB (see page 161)	S2
PIC18F46J50 Plug-In-Module (PIM) (see page 162)	S2
PIC18F47J53 Plug-In-Module (PIM) (see page 163)	S2
PIC18F87J50 Plug-In-Module (PIM) (see page 164)	S4
PIC18F Starter Kit (see page 165)	S1
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	S6[1]
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3[1]
PIC24FJ256DA210 Development Board (see page 167)	S1
PIC24F Starter Kit (see page 168)	N/A[2]
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	S3[1]
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	S3[1]

PIC32 USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)
PIC32 USB Starter Kit (see page 169)	SW1
PIC32 USB Starter Kit II (see page 170)	SW1

Notes:

1) This is the button number on the Explorer 16.

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.24 Host - Audio MIDI Demo

4.24.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1, 2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1, 2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1, 2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1, 2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1, 2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1, 2

Notes:

1. These boards require the Speech Playback PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.24.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.

2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.24.3 Running the Demo

This is a simple demo to show how an embedded MIDI host can be implemented. When a USB MIDI device is attached to the host, the demo host application polls for input data from the device and converts the data from MIDI USB packets to MIDI UART packets and sends the UART packets over the UART bus. And when a UART MIDI device is connected to the host, the demo host application polls for input data from that device and converts the data from MIDI UART packets to MIDI USB packets, and sends the USB packets over the USB bus.

To test this demo, connect a USB MIDI device to the development board, and either connect the board to a computer via RS-232 or a MIDI UART device. The USB MIDI packets will be converted to UART MIDI packets and will be sent to the attached UART MIDI device, or if connected to a computer, send to a terminal program.

To test with a terminal program, make sure that a baud rate of 32150 is used, and the terminal program reads hex, not ascii. The packets sent will be shown in the terminal following the MIDI protocol, which is explained here:

http://www.srm.com/qtma/davidsmidispec.html

4.25 Host - HID - Keyboard Demo

This demo shows how to interface to USB keyboards. Many USB barcode scanners also appear as a USB keyboard.

4.25.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.25.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*

- *Short J2 pin 1 (marked "Temp") to the center pin*
- *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.25.3 Running the Demo

When the device is programmed correctly with the HID host keyboard application the LCD screen on the Explorer 16 should read “Device Detached” if there is no device attached to the USB port. At this point plug in a USB keyboard, bar code scanner that supports HID keyboard emulation, or magnetic card reader that supports HID keyboard emulation. Type a key on the keyboard. This character should be printed on the LCD screen. Pressing the “ESC” key will clear the screen and return the cursor to the first position.

Limitations:

- Neither compound nor composite devices are supported. Some keyboards are either compound or composite.
- The “~” prints as an arrow character instead (“->”). This is an effect of the LCD screen on the Explorer 16. The ascii character for “~” is remapped in the LCD controller.
- The “\” prints as a “¥” character instead. This is an effect of the LCD screen on the Explorer 16. The ascii character for “\” is remapped in the LCD controller.
- Backspace and arrow keys may have issues on Explorer 16 boards with certain LCD modules

4.26 Host - HID - Mouse Demo

4.26.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board does not contain all of the hardware features to run all of the features of the demo, but will work in a limited capacity or has the hardware feature emulated in software.
3. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.26.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*

- *Short JP4*
- *PIC24EP512GU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.26.3 Running the Demo

When the device is programmed correctly with the HID host mouse application the LCD screen on the Explorer 16 should read “Device Detached” if there is no device attached to the USB port. At this point plug in a USB mouse. As you move the mouse the X & Y co-ordinates of the mouse are displayed on the LCD display mounted on the Explorer 16 demo board. The display also toggles the status of Left/Right click status received from the mouse.

Limitations:

- Composite and compound device are not currently supported. These devices may not enumerate or operate correctly. Devices with built in USB hubs are likely compound device. Many multimedia devices with mouse as one of the interface are composite devices.

4.27 Host - Boot Loader - Thumb Drive Boot Loader

4.27.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.27.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ256GB210 PIM*
  - *Short JP1 "U" option to the center pin*
  - *Short JP2 "U" option to the center pin*
  - *Short JP3 "U" option to the center pin*
  - *Short JP4*
- *PIC32MX795F512L PIM*
  - *Open J10*
  - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24FJ256DA210 Development Board Based Demos

For all of the PIC24FJ256DA210 demo board based demos, please follow the following instructions

1. Short JP13 between the S1 and RG8 taps.
2. Short JP14 between the S2 and RE9 taps.
3. Short JP15 between the S3 and RB5 taps.
4. Short JP6
5. Open JP5 and JP7

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.27.3 Running the Demo

Once the image.hex file is created through the process listed above, copy this file onto a thumbdrive. Insert the thumbdrive into the USB A connector. Press and hold the boot load button specified in the table below while resetting the board by pressing the MCLR button or applying power to the board.

An LED on the board should illuminate if this is done correctly (see table below). The LED should turn itself off once the bootload process is complete. At this point of time the application should be running. If you wish to abort a bootload sequence you can press the abort switch or power cycle the board.

Demo Board (click link for board information)	Boot Load Button	Abort Button	LED
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	S3(1)	S6(1)	D5(1)
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	S3(1)	S6(1)	D5(1)
PIC24FJ256DA210 Development Board (see page 167)	S3	S2	D1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	S3(1)	S6(1)	D5(1)
PIC32 USB Starter Kit II (see page 170)	SW1	SW2	LED3

Notes:

1) This is the button/LED number on the Explorer 16 (see page 172).

2) This demo board only has capacitive touch buttons. At this time the button feature of this demo does not work on this board.

4.27.3.1 Creating a Hex File to Load

Any application in the MCHPFSUSB release can be used to create a hex file the can be loaded with this bootloader.

In order to create a new hex file to bootload, follow these steps:

MPLAB 8:

1. First open up the target project.
2. Insure that the correct processor is selected in the “Configure->Select Device” option of MPLAB 8
3. If there is a linker script already attached to the project, remove it from the project by right clicking on it and selecting remove.
4. Copy the appropriate linker file to the demo project folder. The linker files for the thumbdrive bootloader can be found in the “<install directory>\USB Host - Bootloaders\Mass Storage Bootloader\Application Files\Linker Files” folder. For PIC32 target device refer to Customizing the Boot loader and Target application Linker Scripts for PIC32 devices (see page 139).
5. Add this file to the project by right clicking on the “linker scripts” folder in the project window, select “Add Files…”.
6. Compile the project.
7. Rename the resulting .hex file to “image.hex”. It is important that the name be exact. The bootloader is looking for that file specifically by name.

MPLAB X:

1) first open up the target project.

2) Select the configuration that corresponds to your demo board.

3) Copy the appropriate linker file to the demo project folder. The linker files for the thumbdrive bootloader can be found in the “<install directory>\USB Host - Bootloaders\Mass Storage Bootloader\Application Files\Linker Files” folder. For PIC32 target device refer to Customizing the Boot loader and Target application Linker Scripts for PIC32 devices (see page 139).

4) Add this file to the project by right clicking on the “linker scripts” folder in the project window, select “Add Existing Item…”. Select the correct file that corresponds to the processor that is being used on the demo board.

5) Compile the project.

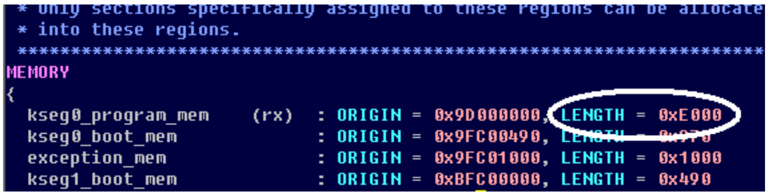
6) Rename the resulting .hex file to “image.hex”. It is important that the name be exact. The bootloader is looking for that file specifically by name.

4.27.3.2 Customizing the Boot Loader and Target Application Linker Scripts for PIC32 devices

The PIC32 USB MSD Host boot loader uses the PIC32MX795F512L device by default. To use the boot loader with another PIC32 device (for example, say the PIC32MX564F128H device), the following steps should be followed.

For Boot Loader Applications:

1. Copy the procdefs.ld file from \Program Files\Microchip\mplabc32\v2.01\pic32mx\lib\proc\32MX564F128H folder to the boot loader project folder (the folder that contains USB Host - Mass Storage Bootloader - C32.mcp file). You may have to overwrite the existing procdefs.ld file.
2. Open the procdefs.ld file in a text editor. Change the length of the kseg0_program_mem section to 0xE000. See the figure below. This is the memory allocated for the boot loader code with compiler size optimization enabled. If the compiler size optimization is disabled, this number should be changed to 0x17000. If the size is not increased, the linker would indicate an error saying that it cannot fit the code within the specified memory region.



3. Save and close procdefs.ld file. Build the USB Host - Mass Storage Bootloader - C32.mcp project and program the boot loader on to the device.

For Target Applications:

1. Copy the procdefs.ld file from \Program Files\Microchip\mplabc32\v2.01\pic32mx\lib\proc\32MX564F128H folder to the target application folder (the folder that contains the project to be boot loaded).
2. Open the procdefs.ld file in a text editor. Multiple changes are needed to the application procdefs.ld file.
   1. Change _RESET_ADDR to 0x9D00F000
   2. Change _BEV_EXCPT_ADDR to 0x9D00F380
   3. Change _DBG_EXCPT_ADDR to 0x9D00F480
   4. Change the exception_mem origin to start from 0x9D00E000. This origin address is obtained by adding the size of boot loader application (which is 0xE000 in our case) to the actual start of the KSEG0 program memory (that is 0x9D000000).
   5. Change the kseg1_boot_mem origin to start from 0x9D00F000
   6. Change the kseg0_boot_mem origin to start from 0x9D00F490
   7. Change the kseg0_program_mem origin to start from 0x9D00FA00. Change the length to 0x10600 (i.e. 0x20000 (the total memory on device) - 0xFA00). The changes are shown in the screen shot below.

```
/*************************************************************************
 * Memory Address Equates
 *************************************************************************/
_RESET_ADDR              = 0x9D00F000;
_BEV_EXCPT_ADDR          = 0x9D00F380;
_DBG_EXCPT_ADDR          = 0x9D00F480;
_DBG_CODE_ADDR           = 0xBFC02000;
_DBG_CODE_SIZE           = 0xFF0     ;
_GEN_EXCPT_ADDR          = _ebase_address + 0x180;

/*************************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for orphaned sections.
 * Only sections specifically assigned to these regions can be allocated
 * into these regions.
 *************************************************************************/
MEMORY
{
  kseg0_program_mem    (rx)  : ORIGIN = 0x9D00FA00, LENGTH = 0x10600
  kseg0_boot_mem             : ORIGIN = 0x9D00F490, LENGTH = 0x970
  exception_mem              : ORIGIN = 0x9D00E000, LENGTH = 0x1000
  kseg1_boot_mem             : ORIGIN = 0x9D00F000, LENGTH = 0x490
```

3. Save and close the procdefs.ld file. Compile and build the target application. This completes the steps required for using the boot loader with another PIC32 device.

4.28 Host - CDC Serial Demo

This demo shows how to interface to USB CDC devices. This typically includes many cell phone models and USB modems.

4.28.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.28.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.28.3 Running the Demo

This is a simple demo to show how an embedded CDC host can be implemented. When a CDC-RS232 device is attached to the host, the demo host application polls for input data from the device and displays the data on the LCD mounted on the explorer 16 board. When a switch SW6 on explorer 16 board is pressed a string “**** Test Data *****” is sent to the attached device to simulate the OUT transfer.

4.29 Host - Composite - CDC + MSD

4.29.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.29.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*

4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*

- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.29.3 Running the Demo

This demo supports a composite device with MSD and CDC interfaces only. Please refer CDC host (see page 140) and MSD host (see page 146)documentation to understand the individual host requirements. The device end can be programmed on any of the development boards for which "USB Device - Composite - MSD + CDC" demo is implemented. The composite host demo works in two steps.

Step 1 - The MSD interface opens a file "test.txt" with text "This is from Composite Host." as the content of the file.

Step 2 - On pressing switch 'S6' on host i.e Explorer 16 board the demo send a character on the CDC interface. The same character is echoed back by the device firmware. On reception the character is displayed on the LCD display on the Explorer 16 demo board.

4.30 Host - Composite - HID + MSD

4.30.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2

PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.30.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.30.3 Running the Demo

This demo supports a composite device with HID and MSD interfaces only. Please refer HID host and MSD host documentation to understand the individual host requirements. The device end can be programmed on any of the development boards for which "USB Device - Composite - HID + MSD (see page 57)" demo is implemented. The composite host demo works in two steps.

Step 1 - The MSD interface opens a file "test.txt" with text "This is from Composite Host." as the content of the file.

Step 2 - The HID interface gets POT value from the device and displays it on the LCD mounted on the Explorer 16 board. On press of switch 'S6' on Explorer 16 board HID interface sends a command to toggle the LED's on the device board.

4.31 Host - MCHPUSB - Generic Driver Demo

This demo shows how to interface to Vendor class devices. It uses "Device - MCHPUSB - Generic Driver Demo" (see page 103) as the target device.

4.31.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.31.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.31.3 Running the Demo

This demo will require two Microchip devices to run. One will act as the USB host and the other will run the USB peripheral.

Program the peripheral with the firmware found in the “USB Device – MCHPUSB – Generic driver demo” (see page 103) folder. Program the host with the firmware found in the “USB Host – MCHPUSB – Generic driver demo” folder.

Power the host. The LCD screen should show the revision number of the MCHPUSB class driver. Plug in the peripheral device. The LCD screen should now update with the potentiometer and temperature data from the attached peripheral.

4.32 Host - Mass Storage (MSD) - Simple Demo

4.32.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1
PIC24FJ256DA210 Development Board (see page 167)	DM240312	
PIC24F Starter Kit (see page 168)	DM240011	
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	1
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	1
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	1
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	1
PIC32 USB Starter Kit (see page 169)	DM320003-1	2
PIC32 USB Starter Kit II (see page 170)	DM320003-2	

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.
2. This board is no longer sold. It was replaced by the PIC32 USB Starter Kit II.

4.32.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  - *Set switch S1 to the "PGX1" setting*
  - *Short J1 pin 1 (marked "POT") to the center pin*
  - *Short J2 pin 1 (marked "Temp") to the center pin*
  - *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*

    - *Short JP1 "U" option to the center pin*
    - *Short JP2 "U" option to the center pin*
    - *Short JP3 "U" option to the center pin*
    - *Short JP4*
- *PIC24EP512GU810 PIM*
    - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
    - *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
    - *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
    - *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
    - *Open J10*
    - *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

PIC24F Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit

- No hardware related configuration or jumper setting changes are necessary.

PIC32 USB Starter Kit II

- No hardware related configuration or jumper setting changes are necessary.

4.32.3 Running the Demo

This demo is a simple example of how to write files to a thumb drive through the Microchip MDD file system library. When a thumb drive is plugged in the code will create a text file on the drive. This process only takes a brief moment. After connecting the thumb drive to the board and waiting for a couple of seconds, remove the drive and plug it back into a computer. There should be an additional text file created named “test.txt”.

Limitations:

- Due to the size of this demo, optimizations must be enabled in the compiler in order for this demo to work on the certain hardware platforms. Optimizations are not available on all versions of the compilers.
- The PIC32MX795F512L Family devices have a register bit named READ. This conflicts with a definition in the MDD library. The MDD Library READ definition should not be used. Instead a 'r' should be used.

4.33 Host - Mass Storage - Thumb Drive Data Logger

This demo shows how to create a console based interface to a system that can read, write, and log data to a thumb drive. This demo is great for showing the various capabilities of the MDD library and the the USB host stack, but is a fairly complex demo. For a more simple introduction (see page 1) to interfacing to a thumb drive, consider starting from the Host - Mass Storage - Simple Demo (see page 146) instead of this demo.

4.33.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.33.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

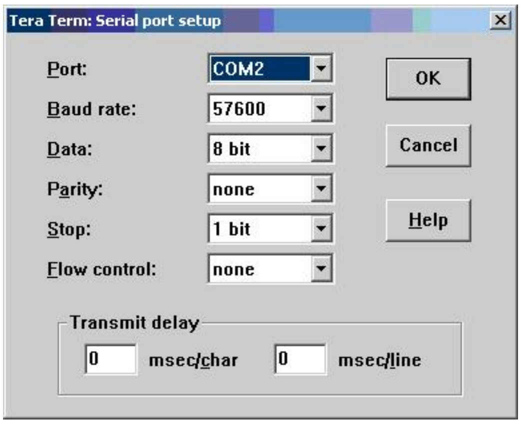
1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
    1. *Set switch S1 to the "PGX1" setting*
    2. *Short J1 pin 1 (marked "POT") to the center pin*
    3. *Short J2 pin 1 (marked "Temp") to the center pin*
    4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
    1. *Short JP1 "U" option to the center pin*
    2. *Short JP2 "U" option to the center pin*
    3. *Short JP3 "U" option to the center pin*
    4. *Short JP4*

- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

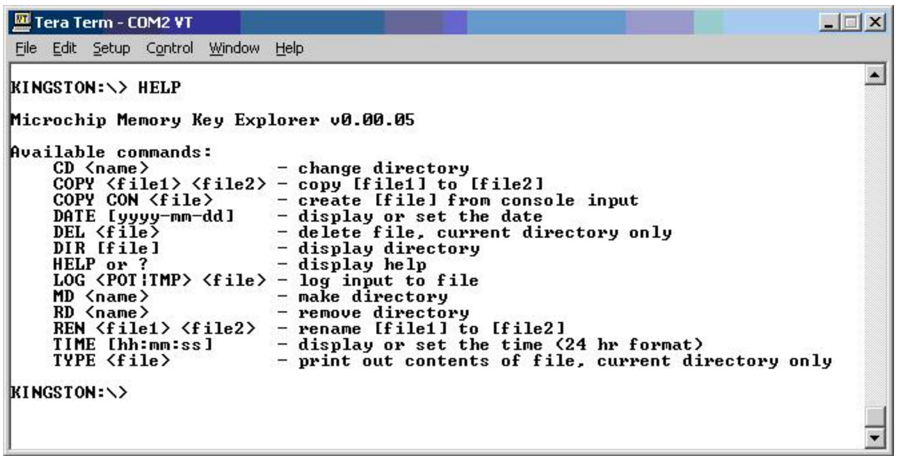
4.33.3 Running the Demo

Once the device is programmed with the correct firmware, connect a serial cable from the Explorer 16 to the computer. Open up a terminal program with the following settings: baud rate – 57600, data – 8bit, parity – none, stop – 1 bit, and flow control – none.



Connect power to the Explorer 16. If power was already connected to the board before the serial port was connected, either cycle power, press reset, or press enter. This should give a “>” prompt. Connect a USB Thumb Drive to the host connector. If the prompt does not appear then verify that you programmed the firmware correctly and that you have the correct serial port settings.

Type “HELP” in the prompt for a list of the available commands.



4.34 Host - Printer - Print Screen Demo

4.34.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1, 2
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	2
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	2
PIC24EP512GU810 Plug-In-Module (PIM) (see page 168)	MA240025-1	2
dsPIC33EP512MU810 Plug-In-Module (PIM) (see page 169)	MA330025-1	2
PIC32 USB Plug-In-Module (PIM) (see page 169)	MA320002	2
PIC32 CAN-USB Plug-In-Module (PIM) (see page 169)	MA320003	2

Notes:

1. This configuration requires optimizations in order to fit the memory of the part. Not all versions of the compilers support all optimization levels.
2. This board can not be used by itself. It requires an Explorer 16 (DM240001), a USB PICTail+ Daughter Board (AC164131), and a Graphics PICTail+ daughter board 3.2" (AC164127-3) in order to operate.

4.34.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP2 and JP3 on the USB PICTail+ board
3. Open JP1 and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

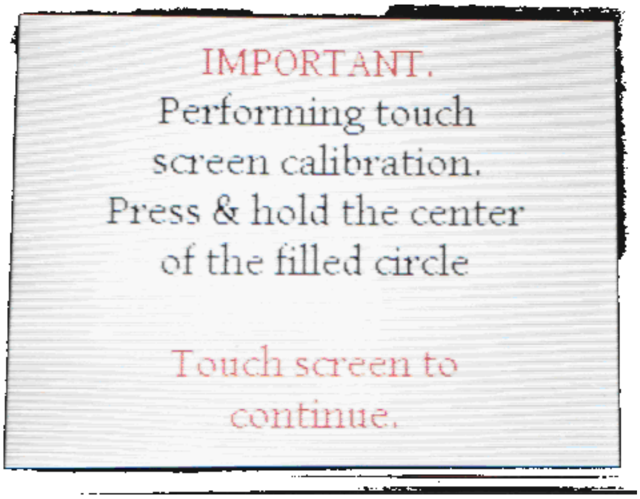
- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*

4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*

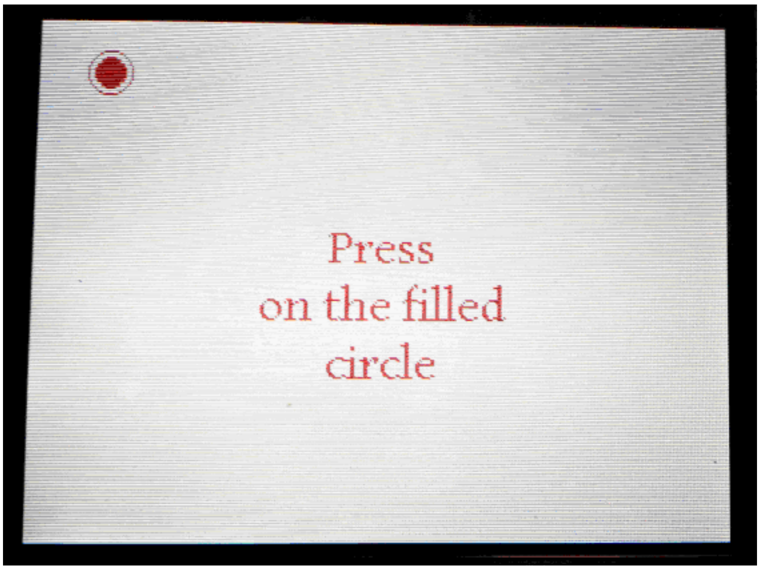
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*
- *PIC24EP512GU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *dsPIC33EP512MU810 PIM*
  1. *Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5*
  2. *Open jumpers J6, J7, J8, J9, and J10*
- *PIC32MX795F512L PIM*
  1. *Open J10*
  2. *Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2*

4.34.3 Running the Demo

The first step required before running this demo is to calibrate the touch screen. Touch screen calibration is done by holding the touch screen down and power cycling the board. This will bring you to the touch screen calibration screen.

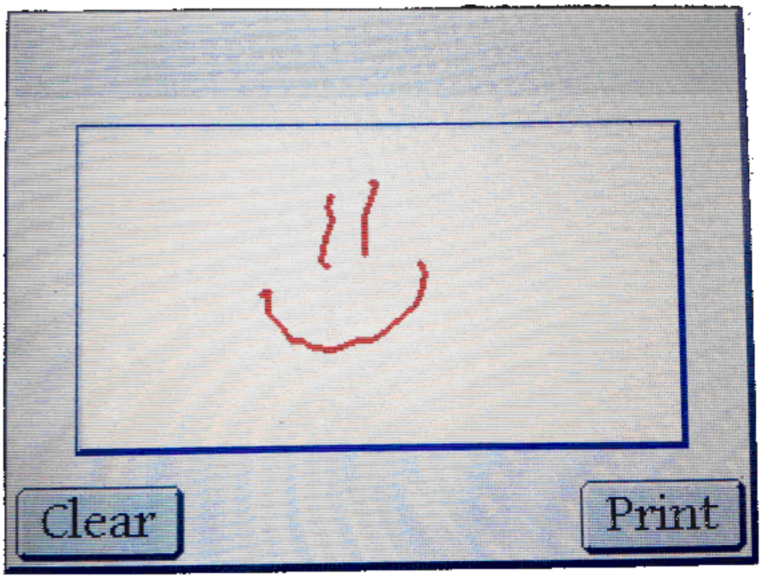


Follow the instructions on the screen.



Once the touch screen calibration is complete, plug in a printer into the board and reset the board. If the following screen comes up, then the printer that was plugged in is not supported or there is no printer plugged in. Please plug in a printer and reset the demo.

If the printer is successfully loaded then the following screen should come up. You can then draw on the screen and press print. When you press print the attached printer should print exactly what is on the screen and clear the screen.



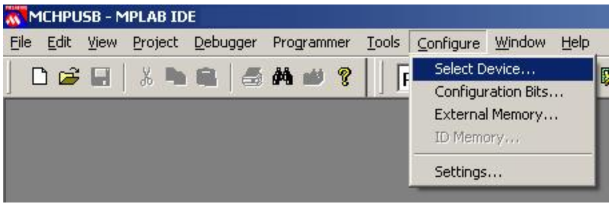
4.35 Loading a precompiled demo

This section describes how to load a pre-compiled demo hex file.

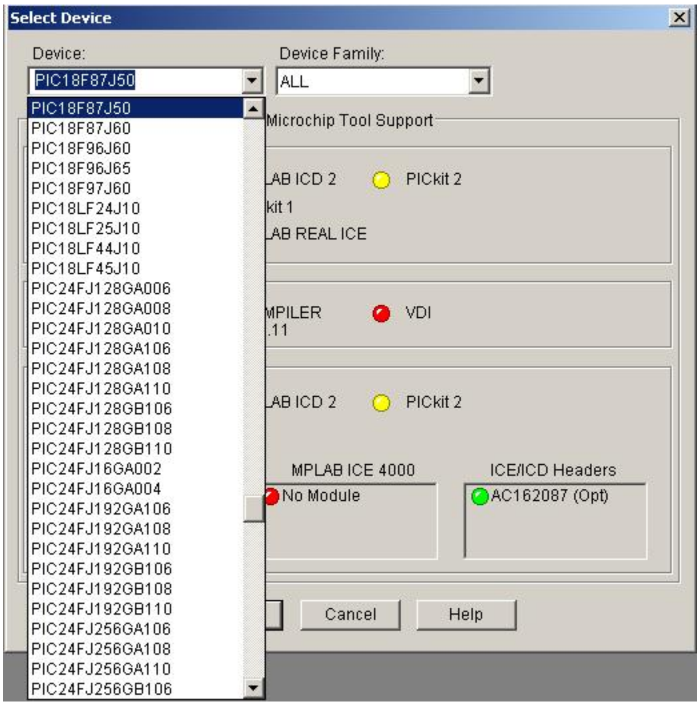
4.35.1 MPLAB 8

Selecting a target device:

1. Open MPLAB® IDE
2. Select Configure->Select Device

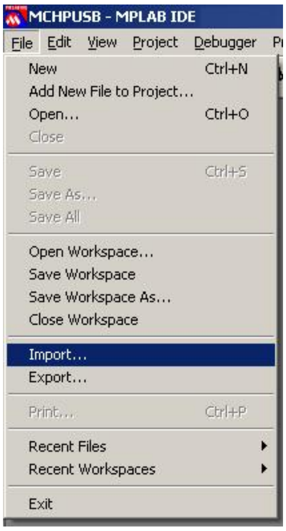


3. Select the target device from the drop down menu



Importing a hex file:

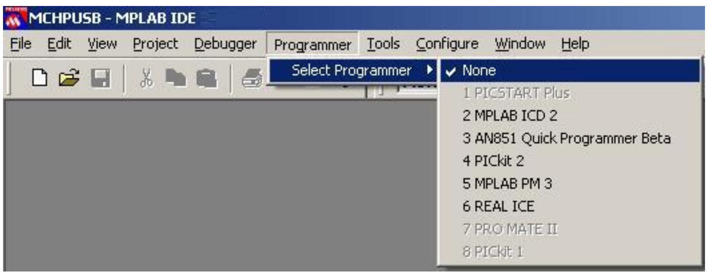
1. Make sure that the correct device is selected before the hex file is imported (see above for instructions).
2. Select File->Import



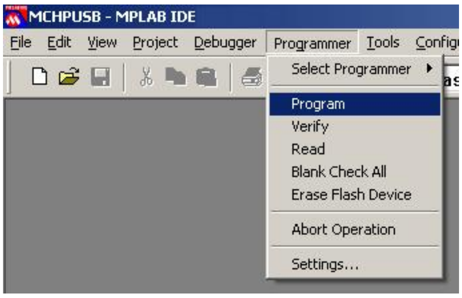
3. Select the desired hex file

Programming the device:

1. Once the correct device is selected and a hex file is imported it is ready to be programmed into a device. Select Programmer->Select Programmer->… and select the programmer that you have available and connected to the computer.



2. After the programmer is enabled connect it to the target device.
3. Select Programmer->Program to program the device.



Running the device:

1. After the device is programmed, some programmers hold the device in reset. The easiest way to get the device running despite what programmer is being used is to disconnect the programmer from the target device.

4.36 PC - WM_DEVICECHANGE Demo

This demo uses the "Device - HID Simple Custom Demo" (see page 76) as the firmware. Please refer to that demos documentation for the supported boards and setups.

Before you can run the WM_DEVICECHANGE_Demo.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for the WM_DEVICECHANGE_Demo.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the "<Install Directory>\USB PC - WM_DEVICECHANGE Demo\WM_DEVICECHANGE Demo - PC Software" directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from Microsoft's website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft's instructions for integrating it with the development environment.

It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the WM_DEVICECHANGE_Demo.exe file. These are only required if the source code will be modified or compiled.

To run the demo, simply run the executable by double clicking on it. The executable can be found in the "<Install Directory>\USB PC - WM_DEVICECHANGE Demo" directory. If the application launches successfully, a window similar to that shown below should appear:

4.37 OTG - MCHPUSB Device/MCHPUSB Host Demo

This demo shows how to implement an OTG device. Both the device and the host are both MCHPUSB custom class demos. More information about the device portion of the demo and how it would work on other hosts can be found in the Device - MCHPUSB Generic Driver Demo (see page 103) section. More information about the host portion of this demo can be found in the Host - MCHPUSB Generic Driver Demo (see page 145) section.

4.37.1 Supported Demo Boards

Demo Board (click link for board information)	Part Number (click link for ordering information)	Notes
PIC24FJ64GB004 Plug-In-Module (PIM) (see page 166)	MA240019	1
PIC24FJ256GB110 Plug-In-Module (PIM) (see page 166)	MA240014	1
PIC24FJ256GB210 Plug-In-Module (PIM) (see page 166)	MA240021	1

Notes:

1. This board can not be used by itself. It requires an Explorer 16 (DM240001) and a USB PICTail+ Daughter Board (AC164131) in order to operate.

4.37.2 Configuring the Demo

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP4 on the USB PICTail+ board
3. Open JP1, JP2, and JP3 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:

- *PIC24FJ64GB004 PIM*
  1. *Set switch S1 to the "PGX1" setting*
  2. *Short J1 pin 1 (marked "POT") to the center pin*
  3. *Short J2 pin 1 (marked "Temp") to the center pin*
  4. *Short J3 pin 1 (marked "EEPROM CS") to the center pin*
- *PIC24FJ256GB210 PIM*
  1. *Short JP1 "U" option to the center pin*
  2. *Short JP2 "U" option to the center pin*
  3. *Short JP3 "U" option to the center pin*
  4. *Short JP4*

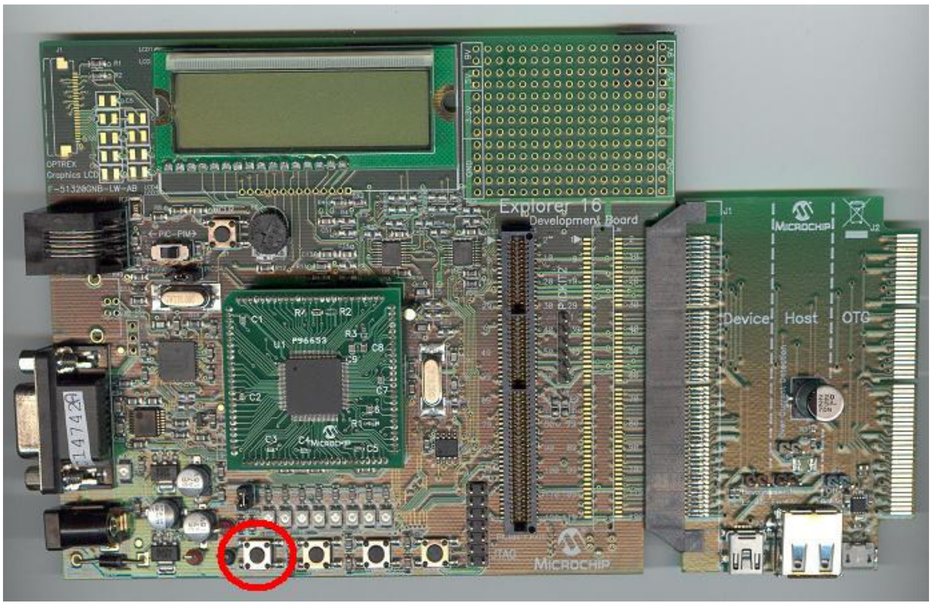
4.37.3 Running the Demo

This demo will require two Microchip devices to run. One will act as the USB host and the other will run the USB peripheral.

To demo the full OTG functionality, program 2 sets of boards with the code provided in the “USB OTG – MCHPUSB – Generic driver demo” folder.

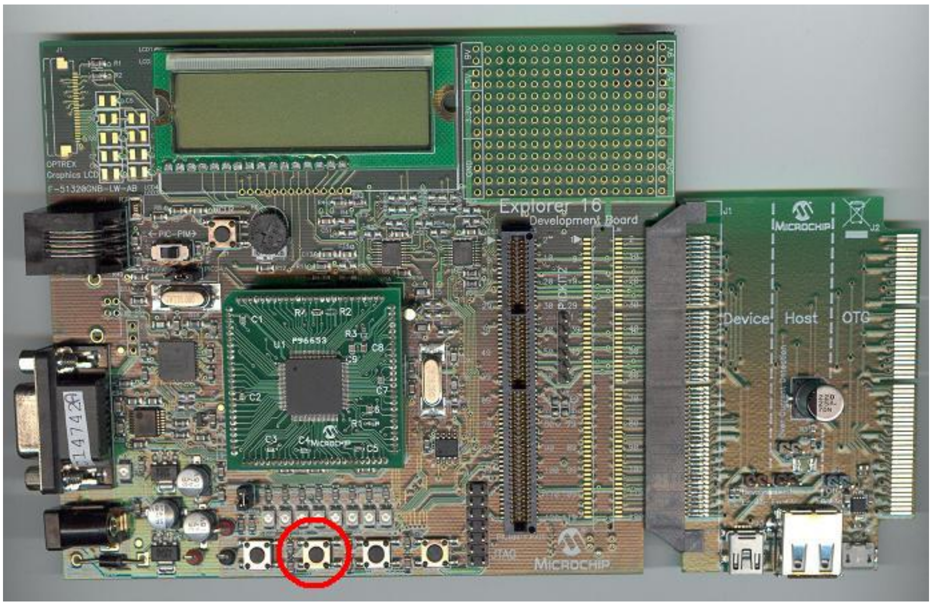
Power both boards. For this demo you will need a micro-A to micro-B cable. At the time of this release these cables are rare and still hard to find. When purchasing a cable please insure that the ID pin on both connectors is correct. On the A side of the cable the ID pin should be less than 10 ohms to ground. On the B-side of the cable the ID pin should be more than 100k ohms to gnd. There are several manufactures that produce micro-A to micro-B cables that are not OTG compliant and these cables have the ID pin connected straight through the cable instead of grounded on one end and floating on the other as the OTG specification calls for.

When the A end of the cable is plugged into either of the boards, that board will become host. When the B end of the cable is connected to either of the boards, that board will become peripheral. On the host board, press the S3 button (seen below). This will cause the host to start supplying 5v to the peripheral. At this point the peripheral should start to operate.



Changing the POT on the peripheral board will cause the LCD on the host to reflect the new value. Similarly the temperature will be affected. Pressing S3 again will cause the host to power down the peripheral. With the power to the peripheral off, press the S3 button on the peripheral. This initiates a Session Request Protocol (SRP). This requests that the host power the bus again. At this point the device should run again without enabling the power from the device.

To switch roles, while both devices are power and the host is communicating to the peripheral, press the S6 button (seen below) on the host (the A device). This initiates a Host Negotiate Protocol (HNP). This causes the A device to become the peripheral and the B device to become the host. The demo is run the same as before except that the POT on the A device is now read on the LCD of the B device. To return to the original configuration, press the S6 button on the B device (now the host). This will suspend the bus and return host control to the A device.



5 Demo Board Information

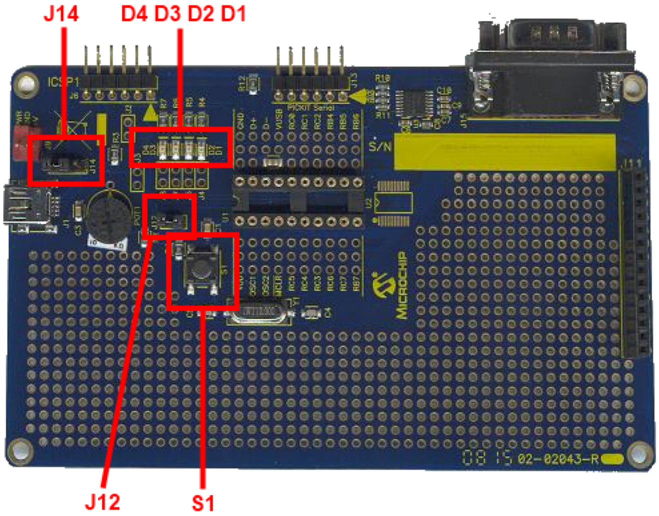
This section gives a brief introduction (see page 1) and links to more information for the USB demo boards.

Description

5.1 Low Pin Count USB Development Board

Overview

This board features the PIC18F14K50 microcontroller. This controller has 20 pins, 16KB of flash, 768 bytes of RAM and an 8-bit core running up to 12MIPS.



J12 - Shorts the VUSB pin to Vdd rail.

J14 - Selects the power source for the board. Short pins 1 and 2 to power from J9. Short pins 2 and 3 to power from the USB VBUS line.

S1 - Application button. Connected to RA3

D1 - Application LED. Connected to RC0

D2 - Application LED. Connected to RC1

D3 - Application LED. Connected to RC2

D4 - Application LED. Connected to RC3

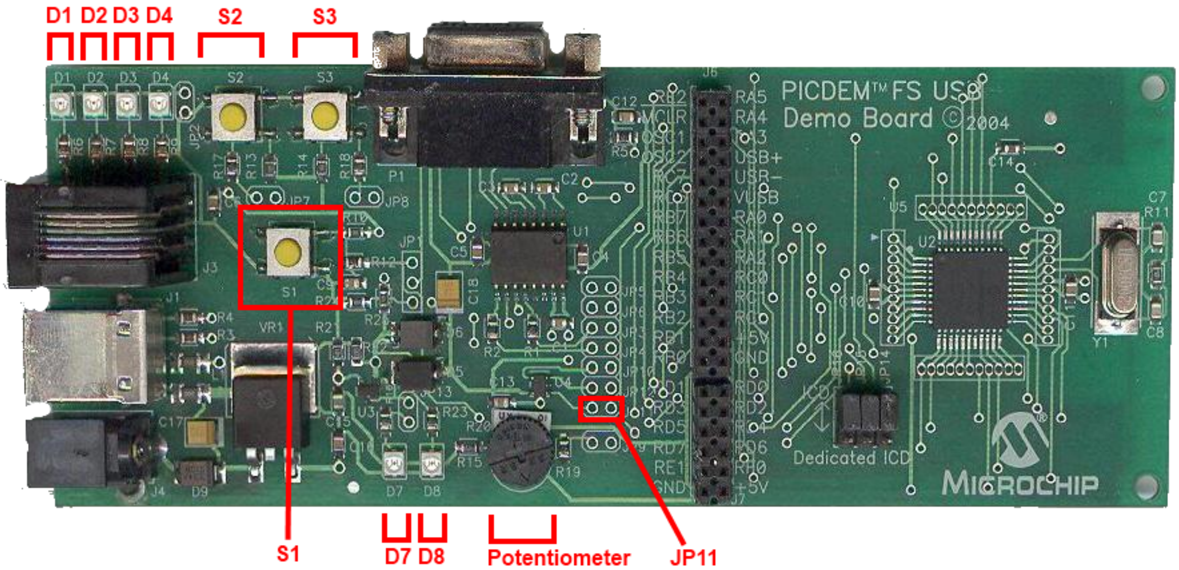
More Information

Product webpage

PIC18F14K50 webpage

5.2 PICDEM FS USB Board

Overview



S1 - MCLR reset button

S2 - Application button

S3 - Application button

D1 - Application LED

D2 - Application LED

D3 - Application LED

D4 - Application LED

D7 - Bus powered indicator - When this LED is illuminated, the board is being powered by the USB bus.

D8 - Self powered indicator - When this LED is illuminated, the board is being powered by an external power supply.

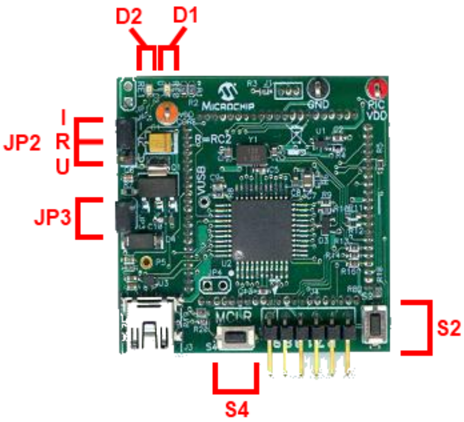
JP11 - connects RB2 of the microcontroller to the temperature sensor on the board (U4). On some revisions of the board there is a trace shorting this jumper that needs to be cut in order to open this jumper.

More Information

Product website

5.3 PIC18F46J50 Plug-In-Module (PIM)

Overview



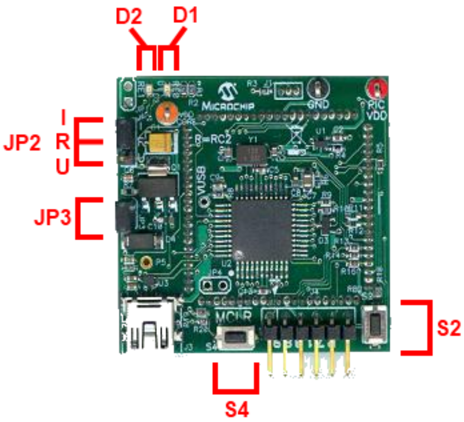
- JP2 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RC2. "I" is an abbreviation referring to the "ICE" female header pin for the RC2 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RC2 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RC2 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller.s 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP3 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP3 provides a means of preventing the board from consuming USB power.
- S2 - Switch for application use. Tied to RB2.
- S4 - MCLR reset switch
- D1 - LED for application use. Tied to RE0.
- D2 - LED for application use. Tied to RE1.

More Information

Product webpage

5.4 PIC18F47J53 Plug-In-Module (PIM)

Overview



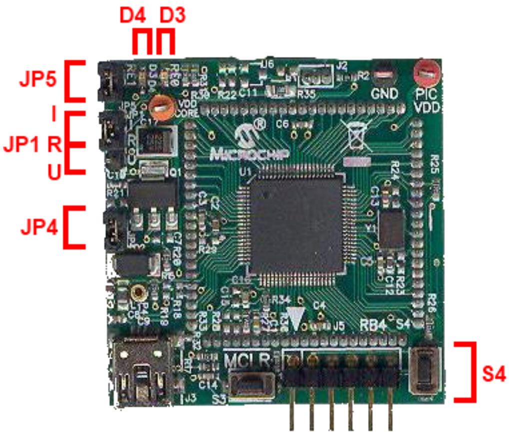
- JP2 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RC2. "I" is an abbreviation referring to the "ICE" female header pin for the RC2 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RC2 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RC2 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller.s 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP3 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP3 provides a means of preventing the board from consuming USB power.
- S2 - Switch for application use. Tied to RB2.
- S4 - MCLR reset switch
- D1 - LED for application use. Tied to RE0.
- D2 - LED for application use. Tied to RE1.

More Information

Product website

5.5 PIC18F87J50 Plug-In-Module (PIM) Demo Board

Overview



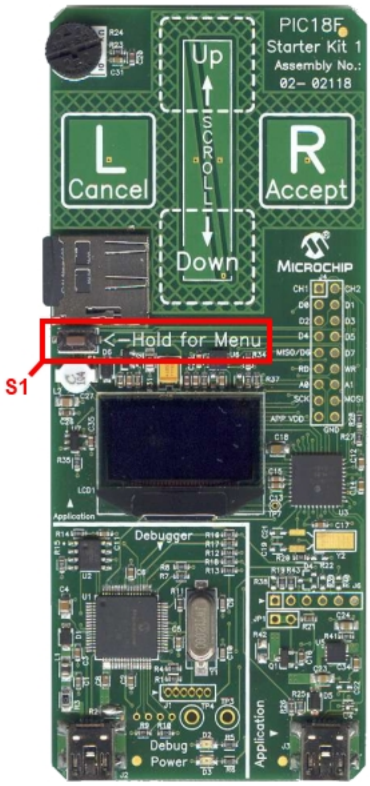
- JP1 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RB5. "I" is an abbreviation referring to the "ICE" female header pin for the RB5 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RB5 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RB5 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller.s 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP4 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP4 provides a means of preventing the board from consuming USB power.
- JP5 - This jumper provides a means of removing the LED pin loading on the RE0 and RE1 pins.
- S4 - Switch for application use. Tied to RB4.
- D3 - LED for application use. Tied to RE0.
- D4 - LED for application use. Tied to RE1.

More Information

- Product webpage

5.6 PIC18 Starter Kit

Overview



S1 - Application switch. Connected to RB0.

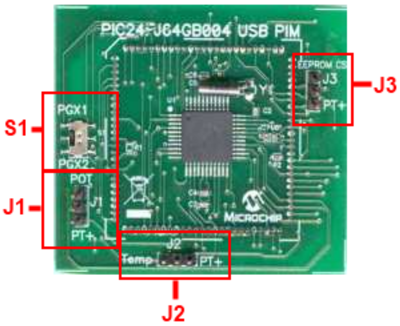
More Information

Product Website

Introduction Video

5.7 PIC24FJ64GB004 Plug-In-Module (PIM)

Overview



S1 - Select which programming pins are going to be used on the microcontroller. The "PGX1" setting must be used for USB operation.

J1 - A/D setting for RC1 (center tap). Setting the jumper to "POT" connects the pin to the potentiometer on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

J2 - A/D setting for RC0 (center tap). Setting the jumper to "Temp" connects the pin to the temperature sensor on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

J3 - I/O selection for RA8 (center tap). Setting the jumper to "EEPROM CS" connects the pin to the chip select line of the EEPROM on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

More Information

Plug-In-Module (PIM) Information Sheet

5.8 PIC24FJ256GB110 Plug-In-Module (PIM)

Overview

The PIC24FJ256GB110 Plug-In-Module (PIM) is not a standalone board. It requires the use of the Explorer 16 (see page 172) (DM240001). For USB applications the USB PICTail plus daughter board (see page 171) (AC164131) is also required.

More Information

Information sheet

5.9 PIC24FJ256GB210 Plug-In-Module (PIM)

Overview

The PIC24FJ256GB210 Plug-In-Module (PIM) is not a standalone board. It requires the use of the Explorer 16 (see page 172) (DM240001). For USB applications the USB PICTail plus daughter board (see page 171) (AC164131) is also required.

For USB operation, jumpers JP1, JP2, and JP3 should be shorted from pins 1 to 2.

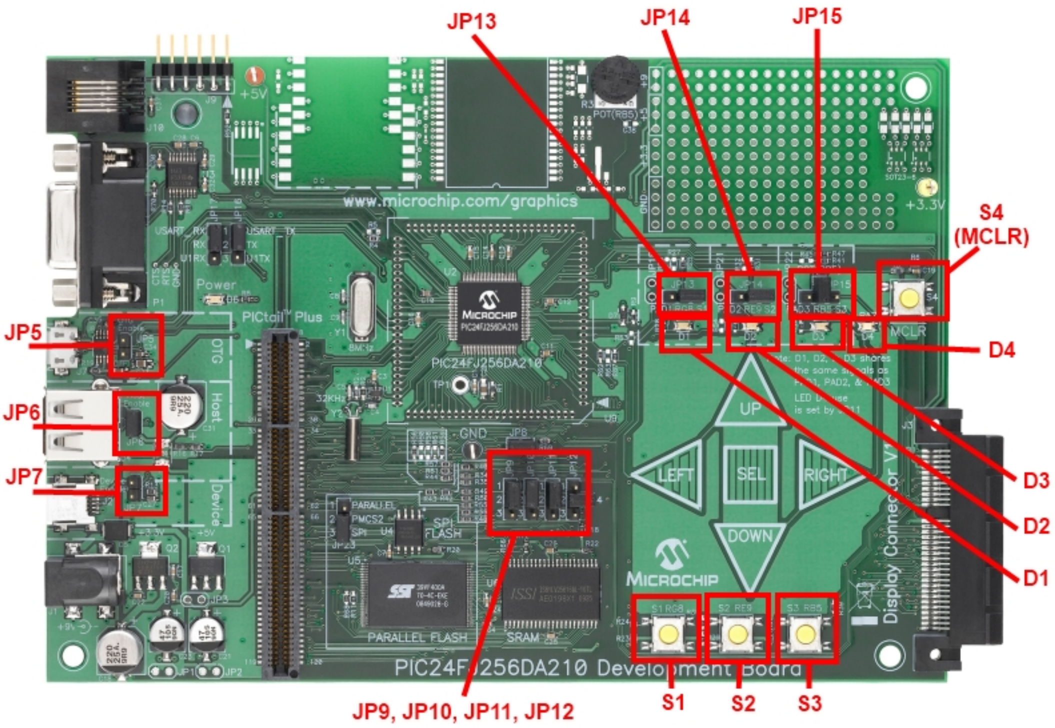
More Information

Information sheet

Ordering information

5.10 PIC24FJ256DA210 Development Board

Overview



S1 - Application switch. Tied to RG8 when JP13 is shorted from S1 to RG8 settings.

S2 - Application switch. Tied to RE9 when JP14 is shorted from S1 to RE9 settings.

S3 - Application switch. Tied to RB5 when JP15 is shorted from S1 to RB5 settings.

S4 - MCLR reset button. Resets the microcontroller on the board.

D1 - Application LED. Connected to RG8 when JP13 is shorted from PAD1 to RG8.

D2 - Application LED. Connected to RE9 when JP14 is shorted from PAD2 to RE9.

D3 - Application LED. Connected to RB5 when JP15 is shorted from PAD3 to RB5.

D4 - Application LED. Connected to RA7 when JP11 is shorted from 1 to 2.

JP5 - Connect USB OTG port to VBUS.

JP6 - Connect USB Host port to VBUS.

JP7 - Connect USB Device port to VBUS.

JP11 - Functionality selection for RA7.

JP13 - Functionality selection for RG8.

JP14 - Functionality selection for RE9.

JP15 - Functionality selection for RB5.

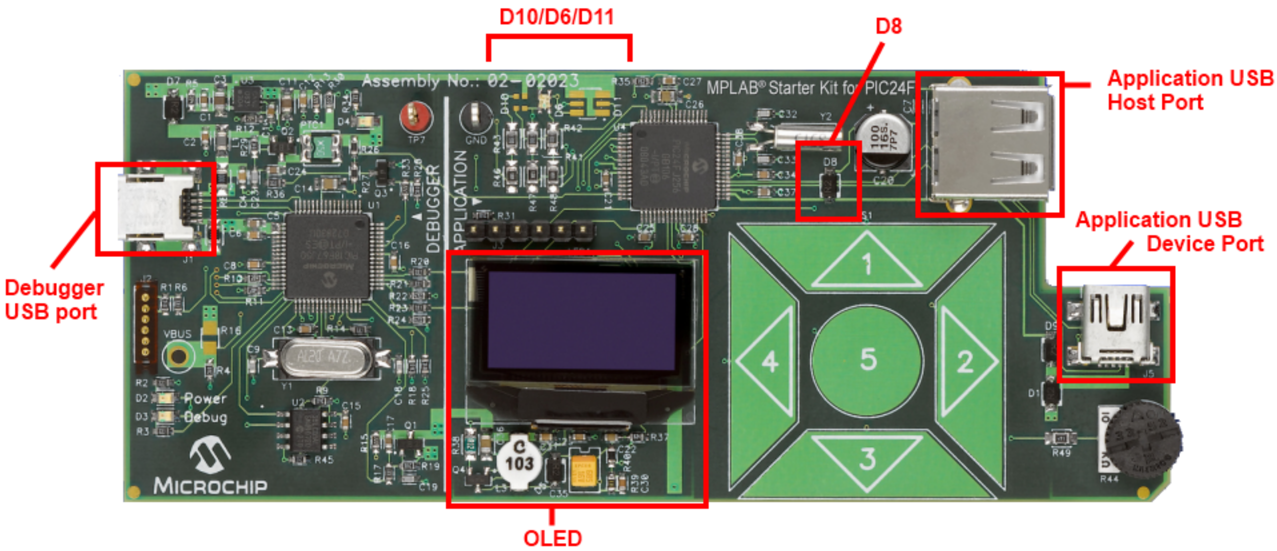
More Information

Product Webpage

Ordering Information

5.11 PIC24F Starter Kit

Overview



D8 - For dual role examples on the PIC24F starter kit, D8 needs to be removed. D8 allows the firmware to verify that the 5v has been delivered to the application USB host port. This, however, is also tied to the application USB device port. With the diode in place the controller can not determine if the 5v it sees is from the USB host port being powered or from the USB device port on an attachment to a USB host.

More Information

Product Website

Ordering Information

Introduction Video

5.12 PIC24EP512GU810 Plug-In-Module (PIM)

More Information

Information Sheet

5.13 dsPIC33EP512MU810 Plug-In-Module (PIM)

More Information

Information Sheet

5.14 PIC32MX460F512L Plug-In-Module (PIM)

More Information

Schematic

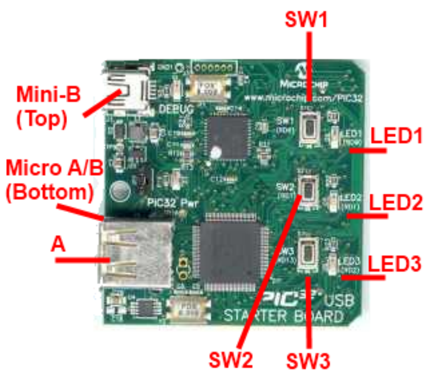
5.15 PIC32MX795F512L Plug-In-Module (PIM)

More Information

Information Sheet

5.16 PIC32 USB Starter Kit

Overview



SW1 - Application switch. Tied to RD6.

SW2 - Application switch. Tied to RD7.

SW3 - Application switch. Tied to RD13.

LED1 - Application LED. Tied to RD0.

LED2 - Application LED. Tied to RD1.

LED3 - Application LED. Tied to RD2.

This board has 3 USB connectors on it.

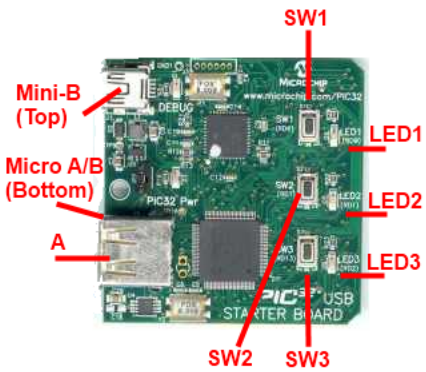
- The mini-B connector is for on-board debugger. It is located on the top of the board.
- The micro-A/B connector is for USB OTG operation on the target microcontroller. It is located on the bottom side of the board.
- The A connector is for USB host support on the target microcontroller.

More Information

The PIC32 USB Starter Kit is no longer sold. It has been replaced by the PIC32 USB Starter Kit II (see page 170).

5.17 PIC32 USB Starter Kit II

Overview



SW1 - Application switch. Tied to RD6.

SW2 - Application switch. Tied to RD7.

SW3 - Application switch. Tied to RD13.

LED1 - Application LED. Tied to RD0.

LED2 - Application LED. Tied to RD1.

LED3 - Application LED. Tied to RD2.

This board has 3 USB connectors on it.

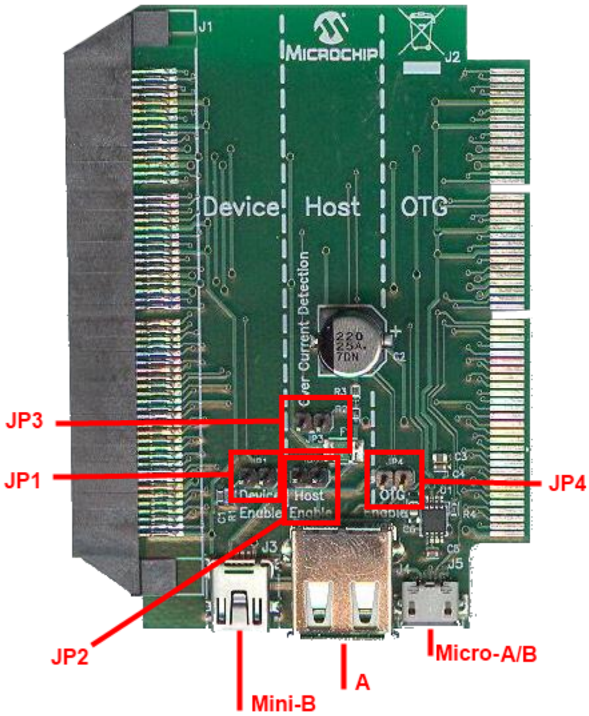
- The mini-B connector is for on-board debugger. It is located on the top of the board.
- The micro-A/B connector is for USB OTG operation on the target microcontroller. It is located on the bottom side of the board.
- The A connector is for USB host support on the target microcontroller.

More Information

Product webpage

5.18 USB PICTail Plus Daughter Board

Overview



JP1 - Connects the VBUS pin of the mini-B connector to the VBUS pin of the microcontroller.

JP2 - Connects the VBUS pin of the A connector (and associated circuitry) to the VBUS pin of the microcontroller.

JP3 - Connects the VBUS voltage detection resistor divider circuit to the microcontroller (pin varies depending on the processor module).

JP4 - Connects the VBUS pin of the micro-A/B connector (and associated circuitry) to the VBUS pin of the microcontroller.

This board has 3 USB connectors on it.

- The mini-B connector is for USB device operation. For use in this mode JP1 should be shorted and JP2, JP3, and JP4 should be open.
- The A connector is for USB host support. JP2 should be short for this mode. JP3 can be shorted to enable VBUS voltage sensing. Some demos may require this feature. JP1 and JP4 should be open.
- The micro-A/B connector is for USB OTG operation. JP4 should be short and JP1, JP2, and JP3 should be open.

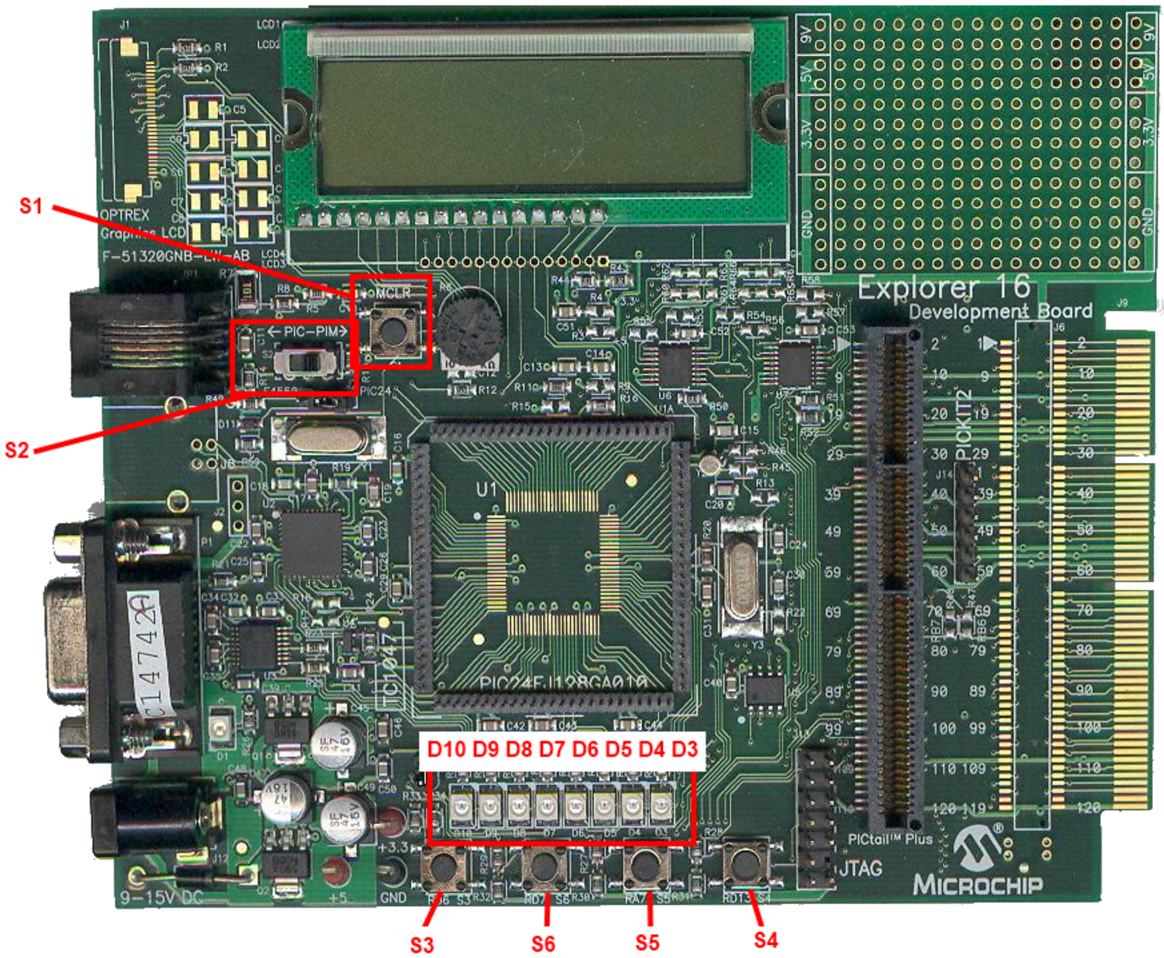
More Information

Product website

Ordering information

5.19 Explorer 16

Overview:



S1 - Reset button (MCLR)

S2 - Processor switch. This switch determines which processor is running, the processor on the board or the processor on the Plug-In-Module (PIM).

S3, S4, S5, S6 - Application switches. For information about what pin is connected to this switch, please refer to the information for the PIM in use.

D3 through D10 - Application LEDs. For information about what pin is connected to this LED, please refer to the information for the PIM in use.

More Information:

Product webpage

6 PC Tools and Example Code

Find out what PC tools and example code are available for development with the MCHPFSUSB Library.

Description

Microchip General Purpose (Custom/Vendor Class) USB Driver

See: <Install Directory>\USB Tools\MCHP Custom Driver\MCHPUSB Driver\Release

Microchip provides a general purpose Windows driver which can be used by Windows applications to interface with a custom class USB device. This driver will not be necessary in many USB applications, such as USB HID class devices, which would normally use built in HID class drivers which distribute with the OS.

For USB applications that do not readily fit within the constraints of these other device class options, Microchip's general purpose driver may be used. Windows applications can access USB devices either by directly interfacing with the driver (mchpusb.sys), or they may indirectly use the driver through a pre-compiled library.

The custom class firmware examples are intended to be used with the general purpose USB driver.

After installation, the release notes for the general purpose USB driver are located at: <Install Directory>\Microchip\USB\Utilities\MCHP Custom Driver\MCHPUSB Driver\MCHPUSB Driver Release Notes.htm

MPUSBAPI Library and DLL Source

See: <Install Directory>\USB Tools\MCHPUSB Custom Driver\Mpusbapi

A custom class Windows application using the Microchip General Purpose USB driver may interface directly with the driver (mchpusb.sys). Doing so directly requires more effort and more time to learn than using a pre-compiled library that exposes a simple to use API including basic functions like open(), read(), write(), and close().

The MPUSBAPI.DLL file is a library which provides a number of functions including the basic ones needed for reading and writing to a USB device. A list of the functions available, and the calling conventions for those functions is currently documented in the form of inline comments in the source code for the DLL file. The DLL is compiled using Borland® C++ Builder™ 6 development environment, and the source code is provided in the "<install directory>\Microchip\USB\Utilities\MCHPUSB Custom Driver\Mpusbapi\Dll\Borland_C\Source" directory.

A load time linking and a run time linking example showing how to use the DLL are included in "<install directory>\Microchip\USB\Utilities\MCHPUSB Custom Driver\Mpusbapi\Example Applications\Borland_C" directory.

PICDEM FS USB Demo Tool "Pdfsusb"

See: <Install Directory>\USB Tools\Pdfsusb

This computer program demonstrates basic USB communication using the Microchip Custom class driver with a Windows GUI based application. The USB Device – MCHPUSB – Generic Driver Demo Firmware is intended to be used in conjunction with the "PICDEM FS USB Demo Tool" which can be launched by executing the PDFSUSB.exe file. The features and use of this application are described in the PICDEM FS USB Demonstration Board User's Guide (DS51526).

This application was originally intended to be used with the PICDEM FS USB Demo Board, but it can be used with the other available USB platforms as well. The demo tool makes use of hardware features, such as a temperature sensor and potentiometer which are not found on all of the hardware platforms. In order to use the demo tool with the PIC18F87J50 PIM, the PIM should be used while it is plugged into the HPC Explorer board. The HPC Explorer board has the needed potentiometer, temperature sensor, and additional LEDs.

In order to use the PICDEM FS USB Demo Tool with any of the hardware platforms, the board will need to be programmed with the code generated by the Custom class device example project or from the custom class precompiled examples.

USBConfig.exe Tool

See: <Install Directory>\USB Tools\USBConfig Tool

Each of the firmware projects requires a usb_config.h that defines several macros that the USB stack uses to know how it should perform. In the case of the embedded host applications there is also a .c file that needs to be create that describes the Targeted Peripheral List (TPL). The TPL is a list of supported devices. This .c file also contains various information that the stack needs to know in order to load and execute the correct client drivers for these devices.

The USBConfig.exe tool is a simple to use interface to help generate the files required by the USB stack.

At the moment the USBConfig.exe tool is only functional for the embedded host examples.

Driver Management Tool

See: <Install Directory>\USB PC - Driver Management Tool

This tool provides an example of how to install a USB driver from within a PC application. This is useful for operating systems like Windows Vista or Windows 7 where the operating system doesn't always ask the user for the driver files if they are not pre-installed.

7 Application Programming Interface (API)

7.1 Device/Peripheral

7.1.1 Device Stack

7.1.1.1 Interface Routines

Functions

	Name	Description
	USB_APPLICATION_EVENT_HANDLER (see page 179)	This function is called whenever the USB stack wants to notify the user of an event.
	USBCancelIO (see page 180)	This function cancels the transfers pending on the specified endpoint. This function can only be used after a SETUP packet is received and before that setup packet is handled. This is the time period in which the EVENT_EP0_REQUEST is thrown, before the event handler function returns to the stack.
	USBCtrlEPAllowDataStage (see page 181)	This function allows the data stage of either a host-to-device or device-to-host control transfer (with data stage) to complete. This function is meant to be used in conjunction with either the USBDeferOUTDataStage (see page 185)() or USBDeferINDataStage (see page 183)(). If the firmware does not call either USBDeferOUTDataStage (see page 185)() or USBDeferINDataStage (see page 183)(), then the firmware does not need to manually call USBCtrlEPAllowDataStage(), as the USB stack will call this function instead.
	USBCtrlEPAllowStatusStage (see page 182)	This function prepares the proper endpoint 0 IN or endpoint 0 OUT (based on the controlTransferState) to allow the status stage packet of a control transfer to complete. This function gets used internally by the USB stack itself, but it may also be called from the application firmware, IF the application firmware called the USBDeferStatusStage (see page 187)() function during the initial processing of the control transfer request. In this case, the application must call the USBCtrlEPAllowStatusStage() once, after it has fully completed processing and handling the data stage portion of the request. If the application firmware has no need for delaying... more (see page 182)
	USBDeferINDataStage (see page 183)	This function will cause the USB hardware to continuously NAK the IN token packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to process and prepare the IN data packets that will eventually be sent to the host. This is also useful, if the firmware needs to process/prepare the IN data in a different context than what the USBDeviceTasks (see page 192)() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the... more (see page 183)
	USBDeferOUTDataStage (see page 185)	This function will cause the USB hardware to continuously NAK the OUT data packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to prepare the RAM buffer that will eventually be used to receive the data from the host. This is also useful, if the firmware wishes to receive the OUT data in a different context than what the USBDeviceTasks (see page 192)() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches... more (see page 185)

	USBDeferStatusStage (see page 187)	Calling this function will prevent the USB stack from automatically enabling the status stage for the currently pending control transfer from completing immediately after all data bytes have been sent or received. This is useful if a class handler or USB application firmware project uses control transfers for sending/receiving data over EP0, but requires time in order to finish processing and/or to consume the data. For example: Consider an application which receives OUT data from the USB host, through EP0 using control transfers. Now assume that this application wishes to do something time consuming with this data (ex: transmit it... more (see page 187)
	USBDeviceAttach (see page 188)	Checks if VBUS is present, and that the USB module is not already initalized, and if so, enables the USB module so as to signal device attachment to the USB host.
	USBDeviceDetach (see page 189)	This function configures the USB module to "soft detach" itself from the USB host.
	USBDeviceInit (see page 191)	This function initializes the device stack it in the default state. The USB module will be completely reset including all of the internal variables, registers, and interrupt flags.
	USBDeviceTasks (see page 192)	This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState (see page 200)() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the... more (see page 192)
	USBEnableEndpoint (see page 194)	This function will enable the specified endpoint with the specified options
	USBEP0Receive (see page 196)	Sets the destination, size, and a function to call on the completion of the next control write.
	USBEP0SendRAMPtr (see page 197)	Sets the source, size, and options of the data you wish to send from a RAM source
	USBEP0SendROMPtr (see page 198)	Sets the source, size, and options of the data you wish to send from a ROM source
	USBEP0Transmit (see page 199)	Sets the address of the data to send over the control endpoint
	USBGetDeviceState (see page 200)	This function will return the current state of the device on the USB. This function should return CONFIGURED_STATE before an application tries to send information on the bus.
	USBGetNextHandle (see page 201)	Retrieves the handle to the next endpoint BDT entry that the USBTransferOnePacket (see page 215)() will use.
	USBGetRemoteWakeupStatus (see page 203)	This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.
	USBGetSuspendState (see page 204)	This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus.
	USBHandleBusy (see page 205)	Checks to see if the input handle is busy
	USBHandleGetAddr (see page 206)	Retrieves the address of the destination buffer of the input handle
	USBHandleGetLength (see page 207)	Retrieves the length of the destination buffer of the input handle

	USBINDataStageDeferred (see page 208)	Returns TRUE if a control transfer with IN data stage is pending, and the firmware has called USBDeferINDataStage (see page 183)(), but has not yet called USBCtrlEPAllowDataStage (see page 181)(). Returns FALSE if a control transfer with IN data stage is either not pending, or the firmware did not call USBDeferINDataStage (see page 183)() at the start of the control transfer. This function (macro) would typically be used in the case where the USBDeviceTasks (see page 192)() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more (see page 208)
	USBIsBusSuspended (see page 209)	This function indicates if the USB bus is in suspend mode.
	USBIsDeviceSuspended (see page 210)	This function indicates if the USB module is in suspend mode.
	USBRxOnePacket (see page 211)	Receives the specified data out the specified endpoint
	USBSoftDetach (see page 212)	This function performs a detach from the USB bus via software.
	USBOUTDataStageDeferred (see page 213)	Returns TRUE if a control transfer with OUT data stage is pending, and the firmware has called USBDeferOUTDataStage (see page 185)(), but has not yet called USBCtrlEPAllowDataStage (see page 181)(). Returns FALSE if a control transfer with OUT data stage is either not pending, or the firmware did not call USBDeferOUTDataStage (see page 185)() at the start of the control transfer. This function (macro) would typically be used in the case where the USBDeviceTasks (see page 192)() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more (see page 213)
	USBStallEndpoint (see page 214)	Configures the specified endpoint to send STALL to the host, the next time the host tries to access the endpoint.
	USBTransferOnePacket (see page 215)	Transfers a single packet (one transaction) of data on the USB bus.
	USBTxOnePacket (see page 217)	Sends the specified data out the specified endpoint

Description

7.1.1.1.1 USB_APPLICATION_EVENT_HANDLER Function

This function is called whenever the USB stack wants to notify the user of an event.

File

usb_device.h

C

```
BOOL USB_APPLICATION_EVENT_HANDLER(
    BYTE address,
    USB_EVENT event,
    void * pdata,
    WORD size
);
```

Returns

None

Description

This function is called whenever the USB stack wants to notify the user of an event. This function should be implemented by the user.

Example Usage:

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	the address of the device when the event happened
BYTE event	The event input specifies which event happened. The possible options are listed in the USB_DEVICE_STACK_EVENTS (see page 220) enumeration.

Function

BOOL USB_APPLICATION_EVENT_HANDLER(BYTE address, USB_EVENT event, void *pdata, WORD size);

7.1.1.1.2 USBCancelIO Function

File

usb_device.h

C

```
void USBCancelIO(
    BYTE endpoint
);
```

Description

This function cancels the transfers pending on the specified endpoint. This function can only be used after a SETUP packet is received and before that setup packet is handled. This is the time period in which the EVENT_EP0_REQUEST is thrown, before the event handler function returns to the stack.

Remarks

None

Parameters

Parameters	Description
BYTE endpoint	the endpoint number you wish to cancel the transfers for

Function

void USBCancelIO(BYTE endpoint)

7.1.1.1.3 USBCtrlEPAllowDataStage Function

This function allows the data stage of either a host-to-device or device-to-host control transfer (with data stage) to complete. This function is meant to be used in conjunction with either the USBDeferOUTDataStage (see page 185)() or USBDeferINDataStage (see page 183)(). If the firmware does not call either USBDeferOUTDataStage (see page 185)() or USBDeferINDataStage (see page 183)(), then the firmware does not need to manually call USBCtrlEPAllowDataStage(), as the USB stack will call this function instead.

File

usb_device.h

C

```
void USBCtrlEPAllowDataStage();
```

Preconditions

A control transfer (with data stage) should already be pending, if the firmware calls this function. Additionally, the firmware should have called either USBDeferOUTDataStage (see page 185)() or USBDeferINDataStage (see page 183)() at the start of the control transfer, if the firmware will be calling this function manually.

Function

void USBCtrlEPAllowDataStage(void);

7.1.1.1.4 USBCtrlEPAllowStatusStage Function

This function prepares the proper endpoint 0 IN or endpoint 0 OUT (based on the controlTransferState) to allow the status stage packet of a control transfer to complete. This function gets used internally by the USB stack itself, but it may also be called from the application firmware, IF the application firmware called the USBDeferStatusStage (see page 187)() function during the initial processing of the control transfer request. In this case, the application must call the USBCtrlEPAllowStatusStage() once, after it has fully completed processing and handling the data stage portion of the request.

If the application firmware has no need for delaying control transfers, and therefore never calls USBDeferStatusStage (see page 187)(), then the application firmware should not call USBCtrlEPAllowStatusStage().

File

usb_device.h

C

```
void USBCtrlEPAllowStatusStage();
```

Remarks

None

Preconditions

None

Function

void USBCtrlEPAllowStatusStage(void);

7.1.1.1.5 USBDeferINDataStage Function

This function will cause the USB hardware to continuously NAK the IN token packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to process and prepare the IN data packets that will eventually be sent to the host. This is also useful, if the firmware needs to process/prepare the IN data in a different context than what the USBDeviceTasks (see page 192)() function executes at.

Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the data stage of the control transfer, even if the firmware has not (yet) called the USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)() API function. However, the application firware must still (eventually, once it is ready) call one of the aforementioned API functions.

Example Usage:

1. Host sends a SETUP packet to the device, requesting a device to host control transfer, with data stage.
2. USBDeviceTasks (see page 192)() executes, and then calls the USBCBCheckOtherReq() callback event handler. The USBCBCheckOtherReq() calls the application specific/device class specific handler that detects the type of control transfer.
3. If the firmware needs more time to prepare the first IN data packet, or, if the firmware wishes to process the command in a different context (ex: if USBDeviceTasks (see page 192)() executes as an interrupt handler, but the IN data stage data needs to be prepared in the main loop context), then it may call USBDeferINDataStage(), in the context of the USBCBCheckOtherReq() handler function.
4. If the firmware called USBDeferINDataStage() in step #3 above, then the hardware will NAK the IN token packets sent by the host, for the IN data stage.
5. Once the firmware is ready, and has successfully prepared the data to be sent to the host in fulfillment of the control transfer, it should then call USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)(), to prepare the USB stack to know how many bytes to send to the host, and from what source location.
6. The firmware should now call USBCtrlEPAllowDataStage (see page 181)(). This will allow the data stage to complete. The USB stack will send the data buffer specified by the USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)() function, when it was called.
7. Once all data has been sent to the host, or if the host performs early termination, the status stage (a 0-byte OUT packet) will complete automatically (assuming the firmware did not call USBDeferStatusStage (see page 187)() during step #3).

File

usb_device.h

C

```
void USBDeferINDataStage();
```

Remarks

Section 9.2.6 of the official USB 2.0 specifications indicates that the USB device must return the first IN data packet within 500ms of the start of the control transfer. In order to meet this specification, the firmware must call USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)(), and then call USBCtrlEPAllowDataStage (see page 181)(), in less than 500ms from the start of the control transfer.

If the firmware calls USBDeferINDataStage(), it must eventually call USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)(), and then call USBCtrlEPAllowDataStage (see page 181)(). If it does not do this, the control transfer will never be able to complete.

The firmware should never call both USBDeferINDataStage() and USBDeferOUTDataStage (see page 185)() during the same control transfer. These functions are mutually exclusive (a control transfer with data stage can never contain both IN and OUT data packets during the data stage).

Preconditions

Before calling USBDeferINDataStage(), the firmware should first verify that the control transfer has a data stage, and that it is of type device-to-host (IN).

Function

void USBDeferINDataStage(void);

7.1.1.1.6 USBDeferOUTDataStage Function

This function will cause the USB hardware to continuously NAK the OUT data packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to prepare the RAM buffer that will eventually be used to receive the data from the host. This is also useful, if the firmware wishes to receive the OUT data in a different context than what the USBDeviceTasks (see page 192)() function executes at.

Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the data stage of the control transfer, even if the firmware has not (yet) called the USBEP0Receive (see page 196)() API function. However, the application firware must still (eventually, once it is ready) call one of the aforementioned API function.

Example Usage:

1. Host sends a SETUP packet to the device, requesting a host to device control transfer, with data stage (OUT data packets).
2. USBDeviceTasks (see page 192)() executes, and then calls the USBCBCheckOtherReq() callback event handler. The USBCBCheckOtherReq() calls the application specific/device class specific handler that detects the type of control transfer.
3. If the firmware needs more time before it wishes to receive the first OUT data packet, or, if the firmware wishes to process the command in a different context, then it may call USBDeferOUTDataStage(), in the context of the USBCBCheckOtherReq() handler function.
4. If the firmware called USBDeferOUTDataStage() in step #3 above, then the hardware will NAK the OUT data packets sent by the host, for the OUT data stage.
5. Once the firmware is ready, it should then call USBEP0Receive (see page 196)(), to prepare the USB stack to receive the OUT data from the host, and to write it to the user specified buffer.
6. The firmware should now call USBCtrlEPAllowDataStage (see page 181)(). This will allow the data stage to complete. Once all OUT data has been received, the user callback function (provided by the function pointer provided when calling USBEP0Receive (see page 196)()) will get called.
7. Once all data has been received from the host, the status stage (a 0-byte IN packet) will complete automatically (assuming the firmware did not call USBDeferStatusStage (see page 187)() during step #3).

File

usb_device.h

C

```c
void USBDeferOUTDataStage();
```

Remarks

Section 9.2.6 of the official USB 2.0 specifications indicates that the USB device must be able to receive all bytes and complete the control transfer within a maximum of 5 seconds.

If the firmware calls USBDeferOUTDataStage(), it must eventually call USBEP0Receive (see page 196)(), and then call USBCtrlEPAllowDataStage (see page 181)(). If it does not do this, the control transfer will never be able to complete. This will break the USB connection, as the host needs to be able to communicate over EP0, in order to perform basic tasks including enumeration.

The firmware should never call both USBDeferINDataStage (see page 183)() and USBDeferOUTDataStage() during the same control transfer. These functions are mutually exclusive (a control transfer with data stage can never contain both IN and OUT data packets during the data stage).

Preconditions

Before calling USBDeferOUTDataStage(), the firmware should first verify that the control transfer has a data stage, and that it is of type host-to-device (OUT).

Function

void USBDeferOUTDataStage(void);

7.1.1.1.7 USBDeferStatusStage Function

Calling this function will prevent the USB stack from automatically enabling the status stage for the currently pending control transfer from completing immediately after all data bytes have been sent or received. This is useful if a class handler or USB application firmware project uses control transfers for sending/receiving data over EP0, but requires time in order to finish processing and/or to consume the data.

For example: Consider an application which receives OUT data from the USB host, through EP0 using control transfers. Now assume that this application wishes to do something time consuming with this data (ex: transmit it to and save it to an external EEPROM device, connected via SPI/I2C/etc.). In this case, it would typically be desireable to defer allowing the USB status stage of the control transfer to complete, until after the data has been fully sent to the EEPROM device and saved.

If the USB class handler firmware that processes the control transfer SETUP packet determines that it will need extra time to complete the control transfer, it may optionally call USBDeferStatusStage(). If it does so, it is then the responsibility of the application firmware to eventually call USBCtrlEPAllowStatusStage (see page 182)(), once the firmware has finished processing the data associated with the control transfer.

If the firmware call USBDeferStatusStage(), but never calls USBCtrlEPAllowStatusStage (see page 182)(), then one of two possibilities will occur.

1. If the "USB_ENABLE_STATUS_STAGE_TIMEOUTS" option is commented in usb_config.h, then the status stage of the control transfer will never be able to complete. This is an error case and should be avoided.
2. If the "USB_ENABLE_STATUS_STAGE_TIMEOUTS" option is enabled in usb_config.h, then the USBDeviceTasks (see page 192)() function will automatically call USBCtrlEPAllowStatusStage (see page 182)(), after the "USB_STATUS_STAGE_TIMEOUT" has elapsed, since the last quanta of "progress" has occurred in the control transfer. Progress is defined as the last successful transaction completing on EP0 IN or EP0 OUT. Although the timeouts feature allows the status stage to [eventually] complete, it is still preferable to manually call USBCtrlEPAllowStatusStage (see page 182)() after the application firmware has finished processing/consuming the control transfer data, as this will allow for much faster processing of control transfers, and therefore much higher data rates and better user responsiveness.

File

usb_device.h

C

```
void USBDeferStatusStage();
```

Remarks

If this function is called, is should get called after the SETUP packet has arrived (the control transfer has started), but before the USBCtrlEPServiceComplete() function has been called by the USB stack. Therefore, the normal place to call USBDeferStatusStage() would be from within the USBCBCheckOtherReq() handler context. For example, in a HID application using control transfers, the USBDeferStatusStage() function would be called from within the USER_GET_REPORT_HANDLER or USER_SET_REPORT_HANDLER functions.

Preconditions

None

Function

void USBDeferStatusStage(void);

7.1.1.1.8 USBDeviceAttach Function

Checks if VBUS is present, and that the USB module is not already initalized, and if so, enables the USB module so as to signal device attachment to the USB host.

File

usb_device.h

C

```
void USBDeviceAttach();
```

Description

This function indicates to the USB host that the USB device has been attached to the bus. This function needs to be called in order for the device to start to enumerate on the bus.

Remarks

See also the USBDeviceDetach (see page 189)() API function documentation.

Preconditions

Should only be called when USB_INTERRUPT is defined. Also, should only be called from the main() loop context. Do not call USBDeviceAttach() from within an interrupt handler, as the USBDeviceAttach() function may modify global interrupt enable bits and settings.

For normal USB devices: Make sure that if the module was previously on, that it has been turned off for a long time (ex: 100ms+) before calling this function to re-enable the module. If the device turns off the D+ (for full speed) or D- (for low speed) ~1.5k ohm pull up resistor, and then turns it back on very quickly, common hosts will sometimes reject this event, since no human could ever unplug and reattach a USB device in a microseconds (or nanoseconds) timescale. The host could simply treat this as some kind of glitch and ignore the event altogether.

Function

void USBDeviceAttach(void)

7.1.1.1.9 USBDeviceDetach Function

This function configures the USB module to "soft detach" itself from the USB host.

File

usb_device.h

C

```
void USBDeviceDetach();
```

Description

This function configures the USB module to perform a "soft detach" operation, by disabling the D+ (or D-) ~1.5k pull up resistor, which lets the host know the device is present and attached. This will make the host think that the device has been unplugged. This is potentially useful, as it allows the USB device to force the host to re-enumerate the device (on the firmware has re-enabled the USB module/pull up, by calling USBDeviceAttach (see page 188)(), to "soft re-attach" to the host).

Remarks

If the application firmware calls USBDeviceDetach(), it is strongly recommended that the firmware wait at least >= 80ms before calling USBDeviceAttach (see page 188)(). If the fimeware performs a soft detach, and then re-attaches too soon (ex: after a few micro seconds for instance), some hosts may interpret this as an unexpected "glitch" rather than as a physical removal/re-attachment of the USB device. In this case the host may simply ignore the event without re-enumerating the device. To ensure that the host properly detects and processes the device soft detach/re-attach, it is recommended to make sure the device remains detached long enough to mimic a real human controlled USB unplug/re-attach event (ex: after calling USBDeviceDetach(), do not call USBDeviceAttach (see page 188)() for at least 80+ms, preferrably longer.

Neither the USBDeviceDetach() or USBDeviceAttach (see page 188)() functions are blocking or take long to execute. It is the application firmware's responsibility for adding the 80+ms delay, when using these API functions.

The Windows plug and play event handler processing is fairly slow, especially in certain versions of Windows, and for certain USB device classes. It has been observed that some device classes need to provide even more USB detach dwell interval (before calling USBDeviceAttach (see page 188)()), in order to work correctly after re-enumeration. If the USB device is a CDC class device, it is recommended to wait at least 1.5 seconds or longer, before soft re-attaching to the host, to provide the plug and play event handler enough time to finish processing the removal event, before the re-attach occurs.

If the application is using the USB_POLLING mode option, then the USBDeviceDetach() and USBDeviceAttach (see page 188)() functions are not available. In this mode, the USB stack relies on the "#define USE_USB_BUS_SENSE_IO" and "#define USB_BUS_SENSE" options in the HardwareProfile – [platform name].h file.

When using the USB_POLLING mode option, and the "#define USE_USB_BUS_SENSE_IO" definition has been commented out, then the USB stack assumes that it should always enable the USB module at pretty much all times. Basically, anytime the application firmware calls USBDeviceTasks (see page 192)(), the firmware will automatically enable the USB module. This mode would typically be selected if the application was designed to be a purely bus powered device. In this case, the application is powered from the +5V VBUS supply from the USB port, so it is correct and sensible in this type of application to power up and turn on the USB module, at anytime that the microcontroller is powered (which implies the USB cable is attached and the host is also powered).

In a self powered application, the USB stack is designed with the intention that the user will enable the "#define USE_USB_BUS_SENSE_IO" option in the HardwareProfile – [platform name].h file. When this option is defined, then the USBDeviceTasks (see page 192)() function will automatically check the I/O pin port value of the designated pin (based on the #define USB_BUS_SENSE option in the HardwareProfile – [platform name].h file), every time the application calls USBDeviceTasks (see page 192)(). If the USBDeviceTasks (see page 192)() function is executed and finds that the pin defined by the #define USB_BUS_SENSE is in a logic low state, then it will automatically disable the USB module and tri-state the D+ and D- pins. If however the USBDeviceTasks (see page 192)() function is executed and finds the pin defined by the #define USB_BUS_SENSE is in a logic high state, then it will automatically enable the USB module, if it has not already been enabled.

Preconditions

Should only be called when USB_INTERRUPT is defined. See remarks section if USB_POLLING mode option is being used (usb_config.h option).

Additionally, this function should only be called from the main() loop context. Do not call this function from within an interrupt handler, as this function may modify global interrupt enable bits and settings.

Function

void USBDeviceDetach(void)

7.1.1.1.10 USBDeviceInit Function

File

usb_device.h

C

```
void USBDeviceInit();
```

Description

This function initializes the device stack it in the default state. The USB module will be completely reset including all of the internal variables, registers, and interrupt flags.

Remarks

None

Preconditions

This function must be called before any of the other USB Device functions can be called, including USBDeviceTasks (see page 192)().

Function

void USBDeviceInit(void)

7.1.1.1.11 USBDeviceTasks Function

This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState (see page 200)() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the faster of: either once every 9.8ms, or as often as needed to make sure that the hardware USTAT FIFO never gets full. A good rule of thumb is to call USBDeviceTasks() at a minimum rate of either the frequency that USBTransferOnePacket (see page 215)() gets called, or, once/1.8ms, whichever is faster. See the inline code comments near the top of usb_device.c for more details about minimum timing requirements when calling USBDeviceTasks().

When the USB stack is operated in "USB_INTERRUPT" mode, it is not necessary to call USBDeviceTasks() from the main loop context. In the USB_INTERRUPT mode, the USBDeviceTasks() handler only needs to execute when a USB interrupt occurs, and therefore only needs to be called from the interrupt context.

File

usb_device.h

C

```
void USBDeviceTasks();
```

Description

This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState (see page 200)() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the faster of: either once every 9.8ms, or as often as needed to make sure that the hardware USTAT FIFO never gets full. A good rule of thumb is to call USBDeviceTasks() at a minimum rate of either the frequency that USBTransferOnePacket (see page 215)() gets called, or, once/1.8ms, whichever is faster. See the inline code comments near the top of usb_device.c for more details about minimum timing requirements when calling USBDeviceTasks().

When the USB stack is operated in "USB_INTERRUPT" mode, it is not necessary to call USBDeviceTasks() from the main loop context. In the USB_INTERRUPT mode, the USBDeviceTasks() handler only needs to execute when a USB interrupt occurs, and therefore only needs to be called from the interrupt context.

Typical usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks(); //Takes care of enumeration and other USB events
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == TRUE))
        {
            //Either the device is not configured or we are suspended,
            // so we don't want to execute any USB related application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run USB and non-USB related user
            //application code.
            UserApplication();
        }
    }
}
```

Remarks

USBDeviceTasks() does not need to be called while in the USB suspend mode, if the user application firmware in the USBCBSuspend() callback function enables the ACTVIF USB interrupt source and put the microcontroller into sleep mode. If the application firmware decides not to sleep the microcontroller core during USB suspend (ex: continues running at full frequency, or clock switches to a lower frequency), then the USBDeviceTasks() function must still be called periodically, at a rate frequent enough to ensure the 10ms resume recovery interval USB specification is met. Assuming a worst case primary oscillator and PLL start up time of <5ms, then USBDeviceTasks() should be called once every 5ms in this scenario.

When the USB cable is detached, or the USB host is not actively powering the VBUS line to +5V nominal, the application firmware does not always have to call USBDeviceTasks() frequently, as no USB activity will be taking place. However, if USBDeviceTasks() is not called regularly, some alternative means of promptly detecting when VBUS is powered (indicating host attachment), or not powered (host powered down or USB cable unplugged) is still needed. For self or dual self/bus powered USB applications, see the USBDeviceAttach (see page 188)() and USBDeviceDetach (see page 189)() API documentation for additional considerations.

Preconditions

Make sure the USBDeviceInit (see page 191)() function has been called prior to calling USBDeviceTasks() for the first time.

Function

void USBDeviceTasks(void)

7.1.1.1.12 USBEnableEndpoint Function

This function will enable the specified endpoint with the specified options

File

usb_device.h

C

```
void USBEnableEndpoint(
    BYTE ep,
    BYTE options
);
```

Returns

None

Description

This function will enable the specified endpoint with the specified options.

Typical Usage:

```
void USBCBInitEP(void)
{
    USBEnableEndpoint(MSD_DATA_IN_EP,USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENABLED|US
B_DISALLOW_SETUP);
    USBMSDInit();
}
```

In the above example endpoint number MSD_DATA_IN_EP is being configured for both IN and OUT traffic with handshaking enabled. Also since MSD_DATA_IN_EP is not endpoint 0 (MSD does not allow this), then we can explicitly disable SETUP packets on this endpoint.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	the endpoint to be configured
BYTE options	optional settings for the endpoint. The options should be ORed together to form a single options string. The available optional settings for the endpoint. The options should be ORed together to form a single options string. The available options are the following: • USB_HANDSHAKE_ENABLED enables USB handshaking (ACK, NAK) • USB_HANDSHAKE_DISABLED disables USB handshaking (ACK, NAK) • USB_OUT_ENABLED enables the out direction • USB_OUT_DISABLED disables the out direction • USB_IN_ENABLED enables the in direction • USB_IN_DISABLED disables the in direction • USB_ALLOW_SETUP enables control transfers • USB_DISALLOW_SETUP disables control transfers • USB_STALL_ENDPOINT STALLs this endpoint

Function

void USBEnableEndpoint(BYTE ep, BYTE options)

7.1.1.1.13 USBEP0Receive Function

Sets the destination, size, and a function to call on the completion of the next control write.

File

usb_device.h

C

```
void USBEP0Receive(
    BYTE* dest,
    WORD size,
    void (*function)
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
dest	address of where the incoming data will go (make sure that this address is directly accessable by the USB module for parts with dedicated USB RAM this address must be in that space)
size	the size of the data being received (is almost always going tobe presented by the preceeding setup packet SetupPkt.wLength)
(*function)	a function that you want called once the data is received. If this is specificed as NULL then no function is called.

Function

void USBEP0Receive(BYTE* dest, WORD size, void (*function))

7.1.1.1.14 USBEP0SendRAMPtr Function

Sets the source, size, and options of the data you wish to send from a RAM source

File

usb_device.h

C

```
void USBEP0SendRAMPtr(
    BYTE* src,
    WORD size,
    BYTE Options
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
src	address of the data to send
size	the size of the data needing to be transmitted
options	the various options that you want when sending the control data. Options are: • USB_EP0_ROM (see page 226) • USB_EP0_RAM (see page 225) • USB_EP0_BUSY (see page 221) • USB_EP0_INCLUDE_ZERO (see page 222) • USB_EP0_NO_DATA (see page 223) • USB_EP0_NO_OPTIONS (see page 224)

Function

void USBEP0SendRAMPtr(BYTE* src, WORD size, BYTE Options)

7.1.1.1.15 USBEP0SendROMPtr Function

Sets the source, size, and options of the data you wish to send from a ROM source

File

usb_device.h

C

```
void USBEP0SendROMPtr(
    BYTE* src,
    WORD size,
    BYTE Options
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
src	address of the data to send
size	the size of the data needing to be transmitted
options	the various options that you want when sending the control data. Options are: • USB_EP0_ROM (see page 226) • USB_EP0_RAM (see page 225) • USB_EP0_BUSY (see page 221) • USB_EP0_INCLUDE_ZERO (see page 222) • USB_EP0_NO_DATA (see page 223) • USB_EP0_NO_OPTIONS (see page 224)

Function

void USBEP0SendROMPtr(BYTE* src, WORD size, BYTE Options)

7.1.1.1.16 USBEP0Transmit Function

Sets the address of the data to send over the control endpoint

File

usb_device.h

C

```
void USBEP0Transmit(
    BYTE options
);
```

Remarks

None

Preconditions

None

Paramters: options - the various options that you want when sending the control data. Options are: USB_EP0_ROM (see page 226) USB_EP0_RAM (see page 225) USB_EP0_BUSY (see page 221) USB_EP0_INCLUDE_ZERO (see page 222) USB_EP0_NO_DATA (see page 223) USB_EP0_NO_OPTIONS (see page 224)

Function

void USBEP0Transmit(BYTE options)

7.1.1.1.17 USBGetDeviceState Function

This function will return the current state of the device on the USB. This function should return CONFIGURED_STATE before an application tries to send information on the bus.

File

usb_device.h

C

```
USB_DEVICE_STATE USBGetDeviceState();
```

Description

This function returns the current state of the device on the USB. This function is used to determine when the device is ready to communicate on the bus. Applications should not try to send or receive data until this function returns CONFIGURED_STATE.

It is also important that applications yield as much time as possible to the USBDeviceTasks (see page 192)() function as possible while the this function returns any value between ATTACHED_STATE through CONFIGURED_STATE.

For more information about the various device states, please refer to the USB specification section 9.1 available from www.usb.org.

Typical usage:

```
void main(void)
{
    USBDeviceInit()
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == TRUE))
        {
            //Either the device is not configured or we are suspended
            //  so we don't want to do execute any application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run user application code.
            UserApplication();
        }
    }
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_DEVICE_STATE (see page 219)	the current state of the device on the bus

Function

USB_DEVICE_STATE (see page 219) USBGetDeviceState(void)

7.1.1.1.18 USBGetNextHandle Function

Retrieves the handle to the next endpoint BDT entry that the USBTransferOnePacket (see page 215)() will use.

File

usb_device.h

C

```
USB_HANDLE USBGetNextHandle(
    BYTE ep_num,
    BYTE ep_dir
);
```

Description

Retrieves the handle to the next endpoint BDT that the USBTransferOnePacket (see page 215)() will use. Useful for initialization and when ping pong buffering will be used on application endpoints.

Remarks

This API is useful for initializing USB_HANDLEs during initialization of the application firmware. It is also useful when ping-pong bufferring is enabled, and the application firmware wishes to arm both the even and odd BDTs for an endpoint simultaneously. In this case, the application firmware for sending data to the host would typically be something like follows:

```
    USB_HANDLE Handle1;
USB_HANDLE Handle2;
USB_HANDLE* pHandle = &Handle1;
BYTE UserDataBuffer1[64];
BYTE UserDataBuffer2[64];
BYTE* pDataBuffer = &UserDataBuffer1[0];

//Add some code that loads UserDataBuffer1[] with useful data to send,
//using the pDataBuffer pointer, for example:
//for(i = 0; i < 64; i++)
//{
//    *pDataBuffer++ = [useful data value];
//}

//Check if the next USB endpoint BDT is available
if(!USBHandleBusy(USBGetNextHandle(ep_num, IN_TO_HOST))
{
    //The endpoint is available.  Send the data.
    *pHandle = USBTransferOnePacket(ep_num, ep_dir, pDataBuffer, bytecount);
    //Toggle the handle and buffer pointer for the next transaction
    if(pHandle == &Handle1)
    {
        pHandle = &Handle2;
        pDataBuffer = &UserDataBuffer2[0];
    }
    else
    {
        pHandle = &Handle1;
        pDataBuffer = &UserDataBuffer1[0];
    }
}

//The firmware can then load the next data buffer (in this case
//UserDataBuffer2)with useful data, and send it using the same
//process.  For example:

//Add some code that loads UserDataBuffer2[] with useful data to send,
//using the pDataBuffer pointer, for example:
//for(i = 0; i < 64; i++)
//{
//    *pDataBuffer++ = [useful data value];
//}

//Check if the next USB endpoint BDT is available
if(!USBHandleBusy(USBGetNextHandle(ep_num, IN_TO_HOST))
```

```
{
    //The endpoint is available.  Send the data.
    *pHandle = USBTransferOnePacket(ep_num, ep_dir, pDataBuffer, bytecount);
    //Toggle the handle and buffer pointer for the next transaction
    if(pHandle == &Handle1)
    {
        pHandle = &Handle2;
        pDataBuffer = &UserDataBuffer2[0];
    }
    else
    {
        pHandle = &Handle1;
        pDataBuffer = &UserDataBuffer1[0];
    }
}
```

Preconditions

Will return NULL if the USB device has not yet been configured/the endpoint specified has not yet been initalized by USBEnableEndpoint (see page 194)().

Parameters

Parameters	Description
BYTE ep_num	The endpoint number to get the handle for (valid values are 1-15, 0 is not a valid input value for this API)
BYTE ep_dir	The endpoint direction associated with the endpoint number to get the handle for (valid values are OUT_FROM_HOST and IN_TO_HOST).

Return Values

Return Values	Description
USB_HANDLE (see page 227)	Returns the USB_HANDLE (see page 227) (a pointer) to the BDT that will be used next time the USBTransferOnePacket (see page 215)() function is called, for the given ep_num and ep_dir

Function

USB_HANDLE (see page 227) USBGetNextHandle(BYTE ep_num, BYTE ep_dir)

7.1.1.1.19 USBGetRemoteWakeupStatus Function

This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.

File

usb_device.h

C

```
BOOL USBGetRemoteWakeupStatus();
```

Description

This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.

If a device does not support remote wakeup (the Remote wakeup bit, bit 5, of the bmAttributes field of the Configuration descriptor is set to 1), then it should not send a remote wakeup command to the PC and this function is not of any use to the device. If a device does support remote wakeup then it should use this function as described below.

If this function returns FALSE and the device is suspended, it should not issue a remote wakeup (resume).

If this function returns TRUE and the device is suspended, it should issue a remote wakeup (resume).

A device can add remote wakeup support by having the _RWU symbol added in the configuration descriptor (located in the usb_descriptors.c file in the project). This done in the 8th byte of the configuration descriptor. For example:

```
ROM BYTE configDescriptor1[]={
  0x09,                           // Size
  USB_DESCRIPTOR_CONFIGURATION,   // descriptor type
  DESC_CONFIG_WORD(0x0022),       // Total length
  1,                              // Number of interfaces
  1,                              // Index value of this cfg
  0,                              // Configuration string index
  _DEFAULT | _SELF | _RWU,        // Attributes, see usb_device.h
  50,                             // Max power consumption in 2X mA(100mA)

  //The rest of the configuration descriptor should follow
```

For more information about remote wakeup, see the following section of the USB v2.0 specification available at www.usb.org:

- Section 9.2.5.2
- Table 9-10
- Section 7.1.7.7
- Section 9.4.5

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	Remote Wakeup has been enabled by the host
FALSE	Remote Wakeup is not currently enabled

Function

BOOL USBGetRemoteWakeupStatus(void)

7.1.1.1.20 USBGetSuspendState Function

This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus.

File

usb_device.h

C

```
BOOL USBGetSuspendState();
```

Description

This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus. This function can be used by the application to skip over section of code that do not need to exectute if the device is unable to send data over the bus. This function can also be used to help determine when it is legal to perform USB remote wakeup signalling, for devices supporting this feature.

Typical usage:

```
void main(void)
{
    USBDeviceInit()
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBGetSuspendState() == TRUE))
        {
            //Either the device is not configured or we are suspended
            //  so we don't want to do execute any application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run user application code.
            UserApplication();
        }
    }
}
```

Remarks

This function is the same as USBIsBusSuspended (see page 209)().

Preconditions

None

Return Values

Return Values	Description
TRUE	the USB port this device is attached to is suspended.
FALSE	the USB port this device is attached to is not suspended.

Function

BOOL USBGetSuspendState(void)

7.1.1.1.1.21 USBHandleBusy Function

Checks to see if the input handle is busy

File

usb_device.h

C

```
BOOL USBHandleBusy(
    USB_HANDLE handle
);
```

Description

Checks to see if the input handle is busy

Typical Usage

```
//make sure that the last transfer isn't busy by checking the handle
if(!USBHandleBusy(USBGenericInHandle))
{
    //Send the data contained in the INPacket[] array out on
    //  endpoint USBGEN_EP_NUM
    USBGenericInHandle = USBGenWrite(USBGEN_EP_NUM,(BYTE*)&INPacket[0],sizeof(INPacket));
}
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
USB_HANDLE handle	handle of the transfer that you want to check the status of

Return Values

Return Values	Description
TRUE	The specified handle is busy
FALSE	The specified handle is free and available for a transfer

Function

BOOL USBHandleBusy(USB_HANDLE (see page 227) handle)

7.1.1.1.22 USBHandleGetAddr Function

Retrieves the address of the destination buffer of the input handle

File

usb_device.h

C

```
WORD USBHandleGetAddr(
    USB_HANDLE
);
```

Description

Retrieves the address of the destination buffer of the input handle

Remarks

None

Preconditions

None

Parameters

Parameters	Description
USB_HANDLE handle	the handle to the transfer you want the address for.

Return Values

Return Values	Description
WORD	address of the current buffer that the input handle points to.

Function

WORD USBHandleGetAddr(USB_HANDLE (see page 227))

7.1.1.1.23 USBHandleGetLength Function

Retrieves the length of the destination buffer of the input handle

File

usb_device.h

C

```
WORD USBHandleGetLength(
    USB_HANDLE handle
);
```

Description

Retrieves the length of the destination buffer of the input handle

Remarks

None

Preconditions

None

Parameters

Parameters	Description
USB_HANDLE handle	the handle to the transfer you want the address for.

Return Values

Return Values	Description
WORD	length of the current buffer that the input handle points to. If the transfer is complete then this is the length of the data transmitted or the length of data actually received.

Function

WORD USBHandleGetLength(USB_HANDLE (see page 227) handle)

7.1.1.1.24 USBINDataStageDeferred Function

Returns TRUE if a control transfer with IN data stage is pending, and the firmware has called USBDeferINDataStage (see page 183)(), but has not yet called USBCtrlEPAllowDataStage (see page 181)(). Returns FALSE if a control transfer with IN data stage is either not pending, or the firmware did not call USBDeferINDataStage (see page 183)() at the start of the control transfer.

This function (macro) would typically be used in the case where the USBDeviceTasks (see page 192)() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main loop context.

In this scenario, typical usage would be:

1. Host starts a control transfer with IN data stage.
2. USBDeviceTasks (see page 192)() (in this scenario, interrupt context) executes.
3. USBDeviceTasks (see page 192)() calls USBCBCheckOtherReq(), which in turn determines that the control transfer is class specific, with IN data stage.
4. The user code in USBCBCheckOtherReq() (also in interrupt context, since it is called from USBDeviceTasks (see page 192)(), and therefore executes at the same priority/context) calls USBDeferINDataStage (see page 183)().
5. Meanwhile, in the main loop context, a polling handler may be periodically checking if(USBINDataStageDeferred() == TRUE). Ordinarily, it would evaluate false, but when a control transfer becomes pending, and after the USBDeferINDataStage (see page 183)() macro has been called (ex: in the interrupt context), the if() statement will evaluate true. In this case, the main loop context can then take care of sending the data (when ready), by calling USBEP0SendRAMPtr (see page 197)() or USBEP0SendROMPtr (see page 198)() and USBCtrlEPAllowDataStage (see page 181)().

File

usb_device.h

C

```
BOOL USBINDataStageDeferred();
```

Function

BOOL USBINDataStageDeferred(void);

7.1.1.1.25 USBIsBusSuspended Function

This function indicates if the USB bus is in suspend mode.

File

usb_device.h

C

```
BOOL USBIsBusSuspended();
```

Returns

None

Description

This function indicates if the USB bus is in suspend mode. This function is typically used for checking if the conditions are consistent with performing a USB remote wakeup sequence.

Typical Usage:

```
if((USBIsBusSuspended() == TRUE) && (USBGetRemoteWakeupStatus() == TRUE))
{
    //Check if some stimulus occured, which will be used as the wakeup source
    if(sw3 == 0)
    {
        USBCBSendResume();  //Send the remote wakeup signalling to the host
    }
}
// otherwise do some other application specific tasks
```

Remarks

The USBIsBusSuspended() function relies on the USBBusIsSuspended boolean variable, which gets updated by the USBDeviceTasks (see page 192)() function. Therefore, in order to be sure the return value is not "stale", it is suggested to make sure USBDeviceTasks (see page 192)() has executed recently (if using USB polling mode).

Preconditions

None

Function

BOOL USBIsBusSuspended(void);

7.1.1.1.26 USBIsDeviceSuspended Function

This function indicates if the USB module is in suspend mode.

File

usb_device.h

C

```
BOOL USBIsDeviceSuspended();
```

Returns

None

Description

This function indicates if the USB module is in suspend mode. This function does NOT indicate that a suspend request has been received. It only reflects the state of the USB module.

Typical Usage:

```
if(USBIsDeviceSuspended() == TRUE)
{
    return;
}
// otherwise do some application specific tasks
```

Remarks

None

Preconditions

None

Function

BOOL USBIsDeviceSuspended(void)

7.1.1.1.27 USBRxOnePacket Function

Receives the specified data out the specified endpoint

File

usb_device.h

C

```
USB_HANDLE USBRxOnePacket(
    BYTE ep,
    BYTE* data,
    WORD len
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
ep	The endpoint number you want to receive the data on.
data	Pointer to a user buffer where the data will go when
it arrives from the host. Note	This RAM must be USB module accessible.
len	The len parameter should always be set to the maximum endpoint packet size, specified in the USB descriptor for this endpoint. The host may send <= the number of bytes as the endpoint size in the endpoint descriptor. After the transaction is complete, the application firmware can call USBHandleGetLength (see page 207)() to determine how many bytes the host actually sent in the last transaction on this endpoint.

Return Values

Return Values	Description
USB_HANDLE (see page 227)	Returns a pointer to the BDT entry associated with the transaction. The firmware can check for completion of the transaction by using the USBHandleBusy (see page 205)() function, using the returned USB_HANDLE (see page 227) value.

Function

USB_HANDLE (see page 227) USBRxOnePacket(BYTE ep, BYTE* data, WORD len)

7.1.1.1.28 USBSoftDetach Function

This function performs a detach from the USB bus via software.

File

usb_device.h

C

```
void USBSoftDetach();
```

Returns

None

Description

This function performs a detach from the USB bus via software.

Remarks

Caution should be used when detaching from the bus. Some PC drivers and programs may require additional time after a detach before a device can be reattached to the bus.

Preconditions

None

Function

void USBSoftDetach(void);

7.1.1.1.29 USBOUTDataStageDeferred Function

Returns TRUE if a control transfer with OUT data stage is pending, and the firmware has called USBDeferOUTDataStage (see page 185)(), but has not yet called USBCtrlEPAllowDataStage (see page 181)(). Returns FALSE if a control transfer with OUT data stage is either not pending, or the firmware did not call USBDeferOUTDataStage (see page 185)() at the start of the control transfer.

This function (macro) would typically be used in the case where the USBDeviceTasks (see page 192)() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main loop context.

In this scenario, typical usage would be:

1. Host starts a control transfer with OUT data stage.
2. USBDeviceTasks (see page 192)() (in this scenario, interrupt context) executes.
3. USBDeviceTasks (see page 192)() calls USBCBCheckOtherReq(), which in turn determines that the control transfer is class specific, with OUT data stage.
4. The user code in USBCBCheckOtherReq() (also in interrupt context, since it is called from USBDeviceTasks (see page 192)(), and therefore executes at the same priority/context) calls USBDeferOUTDataStage (see page 185)().

5. Meanwhile, in the main loop context, a polling handler may be periodically checking if(USBOUTDataStageDeferred() == TRUE). Ordinarily, it would evaluate false, but when a control transfer becomes pending, and after the USBDeferOUTDataStage (see page 185)() macro has been called (ex: in the interrupt context), the if() statement will evaluate true. In this case, the main loop context can then take care of receiving the data, by calling USBEP0Receive (see page 196)() and USBCtrlEPAllowDataStage (see page 181)().

File

usb_device.h

C

```
BOOL USBOUTDataStageDeferred();
```

Function

BOOL USBOUTDataStageDeferred(void);

7.1.1.1.30 USBStallEndpoint Function

Configures the specified endpoint to send STALL to the host, the next time the host tries to access the endpoint.

File

usb_device.h

C

```
void USBStallEndpoint(
    BYTE ep,
    BYTE dir
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	The endpoint number that should be configured to send STALL.
BYTE dir	The direction of the endpoint to STALL, either IN_TO_HOST or OUT_FROM_HOST.

Function

void USBStallEndpoint(BYTE ep, BYTE dir)

7.1.1.1.1.31 USBTransferOnePacket Function

Transfers a single packet (one transaction) of data on the USB bus.

File

usb_device.h

C

```
USB_HANDLE USBTransferOnePacket(
    BYTE ep,
    BYTE dir,
    BYTE* data,
    BYTE len
);
```

Description

The USBTransferOnePacket() function prepares a USB endpoint so that it may send data to the host (an IN transaction), or receive data from the host (an OUT transaction). The USBTransferOnePacket() function can be used both to receive and send data to the host. This function is the primary API function provided by the USB stack firmware for sending or receiving application data over the USB port.

The USBTransferOnePacket() is intended for use with all application endpoints. It is not used for sending or receiving applicaiton data through endpoint 0 by using control transfers. Separate API functions, such as USBEP0Receive (see page 196)(), USBEP0SendRAMPtr (see page 197)(), and USBEP0SendROMPtr (see page 198)() are provided for this purpose.

The USBTransferOnePacket() writes to the Buffer Descriptor Table (BDT) entry associated with an endpoint buffer, and sets the UOWN bit, which prepares the USB hardware to allow the transaction to complete. The application firmware can use the USBHandleBusy (see page 205)() macro to check the status of the transaction, to see if the data has been successfully transmitted yet.

Typical Usage

```
//make sure that the we are in the configured state
if(USBGetDeviceState() == CONFIGURED_STATE)
{
    //make sure that the last transaction isn't busy by checking the handle
    if(!USBHandleBusy(USBInHandle))
    {
        //Write the new data that we wish to send to the host to the INPacket[] array
        INPacket[0] = USEFUL_APPLICATION_VALUE1;
        INPacket[1] = USEFUL_APPLICATION_VALUE2;
        //INPacket[2] = ... (fill in the rest of the packet data)

        //Send the data contained in the INPacket[] array through endpoint "EP_NUM"
        USBInHandle =
USBTransferOnePacket(EP_NUM,IN_TO_HOST,(BYTE*)&INPacket[0],sizeof(INPacket));
    }
}
```

Remarks

If calling the USBTransferOnePacket() function from within the USBCBInitEP() callback function, the set configuration is still being processed and the USBDeviceState may not be == CONFIGURED_STATE yet. In this special case, the USBTransferOnePacket() may still be called, but make sure that the endpoint has been enabled and initialized by the USBEnableEndpoint (see page 194)() function first.

Preconditions

Before calling USBTransferOnePacket(), the following should be true.

1. The USB stack has already been initialized (USBDeviceInit (see page 191)() was called).
2. A transaction is not already pending on the specified endpoint. This is done by checking the previous request using the USBHandleBusy (see page 205)() macro (see the typical usage example).

3. The host has already sent a set configuration request and the enumeration process is complete. This can be checked by verifying that the USBGetDeviceState (see page 200)() macro returns "CONFIGURED_STATE", prior to calling USBTransferOnePacket().

Parameters

Parameters	Description
BYTE ep	The endpoint number that the data will be transmitted or received on
BYTE dir	The direction of the transfer This value is either OUT_FROM_HOST or IN_TO_HOST
BYTE* data	For IN transactions: pointer to the RAM buffer containing
the data to be sent to the host. For OUT transactions	pointer to the RAM buffer that the received data should get written to.
BYTE len	Length of the data needing to be sent (for IN transactions). For OUT transactions, the len parameter should normally be set to the endpoint size specified in the endpoint descriptor.

Return Values

Return Values	Description
USB_HANDLE (see page 227)	handle to the transfer. The handle is a pointer to the BDT entry associated with this transaction. The
status of the transaction (ex	if it is complete or still pending) can be checked using the USBHandleBusy (see page 205)() macro and supplying the USB_HANDLE (see page 227) provided by USBTransferOnePacket().

Function

USB_HANDLE (see page 227) USBTransferOnePacket(BYTE ep, BYTE dir, BYTE* data, BYTE len)

7.1.1.1.32 USBTxOnePacket Function

Sends the specified data out the specified endpoint

File

usb_device.h

C

```
USB_HANDLE USBTxOnePacket(
    BYTE ep,
    BYTE* data,
    WORD len
);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
ep	the endpoint number you want to send the data out of
data	pointer to a user buffer that contains the data that you wish to
send to the host. Note	This RAM buffer must be accessible by the USB module.
len	the number of bytes of data that you wish to send to the host,
in the next transaction on this endpoint. Note	this value should always be less than or equal to the endpoint size, as specified in the USB endpoint descriptor.

Return Values

Return Values	Description
USB_HANDLE (see page 227)	Returns a pointer to the BDT entry associated with the transaction. The firmware can check for completion of the transaction by using the USBHandleBusy (see page 205)() function, using the returned USB_HANDLE (see page 227) value.

Function

USB_HANDLE (see page 227) USBTxOnePacket(BYTE ep, BYTE* data, WORD len)

7.1.1.2 Data Types and Constants

Enumerations

	Name	Description
	USB_DEVICE_STATE (see page 219)	USB Device States as returned by USBGetDeviceState (see page 200)(). Only the defintions for these states should be used. The actual value for each state should not be relied upon as constant and may change based on the implementation.
	USB_DEVICE_STACK_EVENTS (see page 220)	USB device stack events description here - DWF

Macros

	Name	Description
	USB_EP0_BUSY (see page 221)	The PIPE is busy
	USB_EP0_INCLUDE_ZERO (see page 222)	include a trailing zero packet
	USB_EP0_NO_DATA (see page 223)	no data to send
	USB_EP0_NO_OPTIONS (see page 224)	no options set
	USB_EP0_RAM (see page 225)	Data comes from ROM
	USB_EP0_ROM (see page 226)	Data comes from RAM
	USB_HANDLE (see page 227)	USB_HANDLE is a pointer to an entry in the BDT. This pointer can be used to read the length of the last transfer, the status of the last transfer, and various other information. Insure to initialize USB_HANDLE objects to NULL so that they are in a known state during their first usage.

Description

7.1.1.2.1 USB_DEVICE_STATE Enumeration

File

usb_device.h

C

```
typedef enum {
  DETACHED_STATE,
  ATTACHED_STATE,
  POWERED_STATE,
  DEFAULT_STATE,
  ADR_PENDING_STATE,
  ADDRESS_STATE,
  CONFIGURED_STATE
} USB_DEVICE_STATE;
```

Members

Members	Description
DETACHED_STATE	Detached is the state in which the device is not attached to the bus. When in the detached state a device should not have any pull-ups attached to either the D+ or D- line.
ATTACHED_STATE	Attached is the state in which the device is attached ot the bus but the hub/port that it is attached to is not yet configured.
POWERED_STATE	Powered is the state in which the device is attached to the bus and the hub/port that it is attached to is configured.
DEFAULT_STATE	Default state is the state after the device receives a RESET command from the host.
ADR_PENDING_STATE	Address pending state is not an official state of the USB defined states. This state is internally used to indicate that the device has received a SET_ADDRESS command but has not received the STATUS stage of the transfer yet. The device is should not switch addresses until after the STATUS stage is complete.
ADDRESS_STATE	Address is the state in which the device has its own specific address on the bus.
CONFIGURED_STATE	Configured is the state where the device has been fully enumerated and is operating on the bus. The device is now allowed to excute its application specific tasks. It is also allowed to increase its current consumption to the value specified in the configuration descriptor of the current configuration.

Description

USB Device States as returned by USBGetDeviceState (see page 200)(). Only the defintions for these states should be used. The actual value for each state should not be relied upon as constant and may change based on the implementation.

7.1.1.2.2 USB_DEVICE_STACK_EVENTS Enumeration

File

usb_device.h

C

```
typedef enum {
  EVENT_CONFIGURED,
  EVENT_SET_DESCRIPTOR,
  EVENT_EP0_REQUEST,
  EVENT_ATTACH,
  EVENT_TRANSFER_TERMINATED
} USB_DEVICE_STACK_EVENTS;
```

Members

Members	Description
EVENT_CONFIGURED	Notification that a SET_CONFIGURATION() command was received (device)
EVENT_SET_DESCRIPTOR	A SET_DESCRIPTOR request was received (device)
EVENT_EP0_REQUEST	An endpoint 0 request was received that the stack did not know how to handle. This is most often a request for one of the class drivers. Please refer to the class driver documenation for information related to what to do if this request is received. (device)
EVENT_ATTACH	Device-mode USB cable has been attached. This event is not used by the Host stack. The client driver may provide an application event when a device attaches.
EVENT_TRANSFER_TERMINATED	A user transfer was terminated by the stack. This event will pass back the value of the handle that was terminated. Compare this value against the current valid handles to determine which transfer was terminated.

Description

USB device stack events description here - DWF

7.1.1.2.3 USB_EP0_BUSY Macro

File

usb_device.h

C

```
#define USB_EP0_BUSY 0x80      //The PIPE is busy
```

Description

The PIPE is busy

7.1.1.2.4 USB_EP0_INCLUDE_ZERO Macro

File

usb_device.h

C

```
#define USB_EP0_INCLUDE_ZERO 0x40     //include a trailing zero packet
```

Description

include a trailing zero packet

7.1.1.2.5 USB_EP0_NO_DATA Macro

File

usb_device.h

C

```
#define USB_EP0_NO_DATA 0x00      //no data to send
```

Description

no data to send

7.1.1.2.6 USB_EP0_NO_OPTIONS Macro

File

usb_device.h

C

```
#define USB_EP0_NO_OPTIONS 0x00      //no options set
```

Description

no options set

7.1.1.2.7 USB_EP0_RAM Macro

File

usb_device.h

C

```
#define USB_EP0_RAM 0x01      //Data comes from ROM
```

Description

Data comes from ROM

7.1.1.2.8 USB_EP0_ROM Macro

File

usb_device.h

C

```
#define USB_EP0_ROM 0x00     //Data comes from RAM
```

Description

Data comes from RAM

7.1.1.2.9 USB_HANDLE Macro

File

usb_device.h

C

```
#define USB_HANDLE void*
```

Description

USB_HANDLE is a pointer to an entry in the BDT. This pointer can be used to read the length of the last transfer, the status of the last transfer, and various other information. Insure to initialize USB_HANDLE objects to NULL so that they are in a known state during their first usage.

7.1.1.3 Macros

Macros

	Name	Description
	DESC_CONFIG_BYTE (see page 229)	The DESC_CONFIG_BYTE() macro is implemented for convinence. The DESC_CONFIG_BYTE() macro provides a consistant macro for use with a byte when generating a configuratin descriptor when using either the DESC_CONFIG_WORD (see page 231)() or DESC_CONFIG_DWORD (see page 230)() macros.
	DESC_CONFIG_DWORD (see page 230)	The DESC_CONFIG_DWORD() macro is implemented for convinence. Since the configuration descriptor array is a BYTE array, each entry needs to be a BYTE in LSB format. The DESC_CONFIG_DWORD() macro breaks up a DWORD into the appropriate BYTE entries in LSB.
	DESC_CONFIG_WORD (see page 231)	The DESC_CONFIG_WORD() macro is implemented for convinence. Since the configuration descriptor array is a BYTE array, each entry needs to be a BYTE in LSB format. The DESC_CONFIG_WORD() macro breaks up a WORD into the appropriate BYTE entries in LSB. Typical Usage:

Description

7.1.1.3.1 DESC_CONFIG_BYTE Macro

File

usb_device.h

C

```
#define DESC_CONFIG_BYTE(a) (a)
```

Description

The DESC_CONFIG_BYTE() macro is implemented for convinence. The DESC_CONFIG_BYTE() macro provides a consistant macro for use with a byte when generating a configuratin descriptor when using either the DESC_CONFIG_WORD (see page 231)() or DESC_CONFIG_DWORD (see page 230)() macros.

7.1.1.3.2 DESC_CONFIG_DWORD Macro

File

usb_device.h

C

```
#define DESC_CONFIG_DWORD(a) (a&0xFF),((a>>8)&0xFF),((a>>16)&0xFF),((a>>24)&0xFF)
```

Description

The DESC_CONFIG_DWORD() macro is implemented for convinence. Since the configuration descriptor array is a BYTE array, each entry needs to be a BYTE in LSB format. The DESC_CONFIG_DWORD() macro breaks up a DWORD into the appropriate BYTE entries in LSB.

7.1.1.3.3 DESC_CONFIG_WORD Macro

File

usb_device.h

C

```
#define DESC_CONFIG_WORD(a) (a&0xFF),((a>>8)&0xFF)
```

Description

The DESC_CONFIG_WORD() macro is implemented for convinence. Since the configuration descriptor array is a BYTE array, each entry needs to be a BYTE in LSB format. The DESC_CONFIG_WORD() macro breaks up a WORD into the appropriate BYTE entries in LSB. Typical Usage:

```
ROM BYTE configDescriptor1[]={
    0x09,                           // Size of this descriptor in bytes
    USB_DESCRIPTOR_CONFIGURATION,   // CONFIGURATION descriptor type
    DESC_CONFIG_WORD(0x0022),       // Total length of data for this cfg
```

7.1.2 Audio Function Driver

7.1.2.1 Interface Routines

Functions

	Name	Description
	USBCheckAudioRequest (see page 233)	This routine checks the setup data packet to see if it knows how to handle it

Description

7.1.2.1.1 USBCheckAudioRequest Function

This routine checks the setup data packet to see if it knows how to handle it

File

usb_function_audio.h

C

```
void USBCheckAudioRequest();
```

Description

This routine checks the setup data packet to see if it knows how to handle it

Remarks

None

Preconditions

None

Function

void USBCheckAudioRequest(void)

7.1.2.2 Data Types and Constants

7.1.3 CCID (Smart/Sim Card) Function Driver

7.1.3.1 Interface Routines

Functions

	Name	Description
	USBCCIDBulkInService (see page 236)	USBCCIDBulkInService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.
	USBCCIDInitEP (see page 237)	This function initializes the CCID function driver. This function should be called after the SET_CONFIGURATION command.
	USBCCIDSendDataToHost (see page 238)	USBCCIDSendDataToHost writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).
	USBCheckCCIDRequest (see page 239)	This routine checks the setup data packet to see if it knows how to handle it

Description

7.1.3.1.1 USBCCIDBulkInService Function

USBCCIDBulkInService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

File

usb_function_ccid.h

C

```
void USBCCIDBulkInService();
```

Description

USBCCIDBulkInService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

Typical Usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == TRUE))
        {
            //Either the device is not configured or we are suspended
            //  so we don't want to do execute any application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Run application code.
            UserApplication();

            //Keep trying to send data to the PC as required
            USBCCIDBulkInService();
        }
    }
}
```

Remarks

None

Preconditions

None

Function

void USBCCIDBulkInService(void)

7.1.3.1.2 USBCCIDInitEP Function

This function initializes the CCID function driver. This function should be called after the SET_CONFIGURATION command.

File

usb_function_ccid.h

C

```
void USBCCIDInitEP();
```

Description

This function initializes the CCID function driver. This function sets the default line coding (baud rate, bit parity, number of data bits, and format). This function also enables the endpoints and prepares for the first transfer from the host.

This function should be called after the SET_CONFIGURATION command. This is most simply done by calling this function from the USBCBInitEP() function.

Typical Usage:

```
void USBCBInitEP(void)
{
    USBCCIDInitEP();
}
```

Remarks

None

Preconditions

None

Function

void USBCCIDInitEP(void)

7.1.3.1.3 USBCCIDSendDataToHost Function

USBCCIDSendDataToHost writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).

File

usb_function_ccid.h

C

```
void USBCCIDSendDataToHost(
    BYTE * pData,
    WORD len
);
```

Description

USBCCIDSendDataToHost writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. USBCCIDBulkInService (see page 236)() must be called periodically to keep sending blocks of data to the host.

Parameters

Parameters	Description
BYTE *data	pointer to a RAM array of data to be transfered to the host
WORD length	the number of bytes to be transfered

Function

void USBCCIDSendDataToHost(BYTE *data, WORD length)

7.1.3.1.4 USBCheckCCIDRequest Function

File

usb_function_ccid.h

C

```
void USBCheckCCIDRequest();
```

Description

This routine checks the setup data packet to see if it knows how to handle it

Remarks

None

Preconditions

None

Function

void USBCheckCCIDRequest(void)

7.1.4 CDC Function Driver

7.1.4.1 Interface Routines

Functions

	Name	Description
	CDCInitEP (see page 241)	This function initializes the CDC function driver. This function should be called after the SET_CONFIGURATION command (ex: within the context of the USBCBInitEP() function).
	CDCTxService (see page 242)	CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.
	getsUSBUSART (see page 243)	getsUSBUSART copies a string of BYTEs received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.
	putrsUSBUSART (see page 244)	putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.
	putsUSBUSART (see page 245)	putsUSBUSART writes a string of data to the USB including the null character. Use this version, 'puts', to transfer data from a RAM buffer.
	putUSBUSART (see page 246)	putUSBUSART writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).
	USBCheckCDCRequest (see page 247)	This routine checks the most recently received SETUP data packet to see if the request is specific to the CDC class. If the request was a CDC specific request, this function will take care of handling the request and responding appropriately.

Macros

	Name	Description
	CDCSetBaudRate (see page 248)	This macro is used set the baud rate reported back to the host during a get line coding request. (optional)
	CDCSetCharacterFormat (see page 249)	This macro is used manually set the character format reported back to the host during a get line coding request. (optional)
	CDCSetDataSize (see page 250)	This function is used manually set the number of data bits reported back to the host during a get line coding request. (optional)
	CDCSetLineCoding (see page 251)	This function is used to manually set the data reported back to the host during a get line coding request. (optional)
	CDCSetParity (see page 252)	This function is used manually set the parity format reported back to the host during a get line coding request. (optional)
	USBUSARTIsTxTrfReady (see page 253)	This macro is used to check if the CDC class is ready to send more data.

Description

7.1.4.1.1 CDCInitEP Function

This function initializes the CDC function driver. This function should be called after the SET_CONFIGURATION command (ex: within the context of the USBCBInitEP() function).

File

usb_function_cdc.h

C

```
void CDCInitEP();
```

Description

This function initializes the CDC function driver. This function sets the default line coding (baud rate, bit parity, number of data bits, and format). This function also enables the endpoints and prepares for the first transfer from the host.

This function should be called after the SET_CONFIGURATION command. This is most simply done by calling this function from the USBCBInitEP() function.

Typical Usage:

```
void USBCBInitEP(void)
{
    CDCInitEP();
}
```

Remarks

None

Preconditions

None

Function

void CDCInitEP(void)

7.1.4.1.2 CDCTxService Function

CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

File

usb_function_cdc.h

C

```
void CDCTxService();
```

Description

CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state (after the CDCIniEP() function has already executed). This function is needed, in order to advance the internal software state machine that takes care of sending multiple transactions worth of IN USB data to the host, associated with CDC serial data. Failure to call CDCTxService() perioidcally will prevent data from being sent to the USB host, over the CDC serial data interface.

Typical Usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == TRUE))
        {
            //Either the device is not configured or we are suspended
            //  so we don't want to do execute any application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Keep trying to send data to the PC as required
            CDCTxService();

            //Run application code.
            UserApplication();
        }
    }
}
```

Remarks

None

Preconditions

CDCIniEP() function should have already exectuted/the device should be in the CONFIGURED_STATE.

Function

void CDCTxService(void)

7.1.4.1.3 getsUSBUSART Function

getsUSBUSART copies a string of BYTEs received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

File

usb_function_cdc.h

C

```
BYTE getsUSBUSART(
    char * buffer,
    BYTE len
);
```

Returns

BYTE - Returns a byte indicating the total number of bytes that were actually received and copied into the specified buffer. The returned value can be anything from 0 up to the len input value. A return value of 0 indicates that no new CDC bulk OUT endpoint data was available.

Description

getsUSBUSART copies a string of BYTEs received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

Typical Usage:

```
BYTE numBytes;
BYTE buffer[64]

numBytes = getsUSBUSART(buffer,sizeof(buffer)); //until the buffer is free.
if(numBytes > 0)
{
    //we received numBytes bytes of data and they are copied into
    //  the "buffer" variable.  We can do something with the data
    //  here.
}
```

Preconditions

Value of input argument 'len' should be smaller than the maximum endpoint size responsible for receiving bulk data from USB host for CDC class. Input argument 'buffer' should point to a buffer area that is bigger or equal to the size specified by 'len'.

Parameters

Parameters	Description
buffer	Pointer to where received BYTEs are to be stored
len	The number of BYTEs expected.

Function

BYTE getsUSBUSART(char *buffer, BYTE len)

7.1.4.1.4 putrsUSBUSART Function

putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.

File

usb_function_cdc.h

C

```
void putrsUSBUSART(
    const ROM char * data
);
```

Description

putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.

Typical Usage:

```
    if(USBUSARTIsTxTrfReady())
    {
        putrsUSBUSART("Hello World");
    }
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. CDCTxService (see page 242)() must be called periodically to keep sending blocks of data to the host.

Preconditions

USBUSARTIsTxTrfReady (see page 253)() must return TRUE. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTEs.

Parameters

Parameters	Description
const ROM char *data	null-terminated string of constant data. If a null character is not found, 255 BYTEs of data will be transferred to the host.

Function

void putrsUSBUSART(const ROM char *data)

7.1.4.1.5 putsUSBUSART Function

putsUSBUSART writes a string of data to the USB including the null character. Use this version, 'puts', to transfer data from a RAM buffer.

File

usb_function_cdc.h

C

```
void putsUSBUSART(
    char * data
);
```

Description

putsUSBUSART writes a string of data to the USB including the null character. Use this version, 'puts', to transfer data from a RAM buffer.

Typical Usage:

```
    if(USBUSARTIsTxTrfReady())
    {
        char data[] = "Hello World";
        putsUSBUSART(data);
    }
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. CDCTxService (see page 242)() must be called periodically to keep sending blocks of data to the host.

Preconditions

USBUSARTIsTxTrfReady (see page 253)() must return TRUE. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTEs.

Parameters

Parameters	Description
char *data	null-terminated string of constant data. If a null character is not found, 255 BYTEs of data will be transferred to the host.

Function

void putsUSBUSART(char *data)

7.1.4.1.6 putUSBUSART Function

putUSBUSART writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).

File

usb_function_cdc.h

C

```
void putUSBUSART(
    char * data,
    BYTE Length
);
```

Description

putUSBUSART writes an array of data to the USB. Use this version, is capable of transfering 0x00 (what is typically a NULL character in any of the string transfer functions).

Typical Usage:

```
if(USBUSARTIsTxTrfReady())
{
    char data[] = {0x00, 0x01, 0x02, 0x03, 0x04};
    putUSBUSART(data,5);
}
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. CDCTxService (see page 242)() must be called periodically to keep sending blocks of data to the host.

Preconditions

USBUSARTIsTxTrfReady (see page 253)() must return TRUE. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTEs.

Parameters

Parameters	Description
char *data	pointer to a RAM array of data to be transfered to the host
BYTE length	the number of bytes to be transfered (must be less than 255).

Function

void putUSBUSART(char *data, BYTE length)

7.1.4.1.7 USBCheckCDCRequest Function

File

usb_function_cdc.h

C

```
void USBCheckCDCRequest();
```

Description

This routine checks the most recently received SETUP data packet to see if the request is specific to the CDC class. If the request was a CDC specific request, this function will take care of handling the request and responding appropriately.

Remarks

This function does not change status or do anything if the SETUP packet did not contain a CDC class specific request.

Preconditions

This function should only be called after a control transfer SETUP packet has arrived from the host.

Function

void USBCheckCDCRequest(void)

7.1.4.1.8 CDCSetBaudRate Macro

This macro is used set the baud rate reported back to the host during a get line coding request. (optional)

File

usb_function_cdc.h

C

```
#define CDCSetBaudRate(baudRate) {line_coding.dwDTERate.Val=baudRate;}
```

Description

This macro is used set the baud rate reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetBaudRate(19200);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
DWORD baudRate	The desired baudrate

Function

void CDCSetBaudRate(DWORD baudRate)

7.1.4.1.9 CDCSetCharacterFormat Macro

This macro is used manually set the character format reported back to the host during a get line coding request. (optional)

File

usb_function_cdc.h

C

```
#define CDCSetCharacterFormat(charFormat) {line_coding.bCharFormat=charFormat;}
```

Description

This macro is used manually set the character format reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetCharacterFormat(NUM_STOP_BITS_1);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE charFormat	number of stop bits. Available options are: • NUM_STOP_BITS_1 (see page 255) - 1 Stop bit • NUM_STOP_BITS_1_5 (see page 256) - 1.5 Stop bits • NUM_STOP_BITS_2 (see page 257) - 2 Stop bits

Function

void CDCSetCharacterFormat(BYTE charFormat)

7.1.4.1.10 CDCSetDataSize Macro

This function is used manually set the number of data bits reported back to the host during a get line coding request. (optional)

File

usb_function_cdc.h

C

```
#define CDCSetDataSize(dataBits) {line_coding.bDataBits=dataBits;}
```

Description

This function is used manually set the number of data bits reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetDataSize(8);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE dataBits	number of data bits. The options are 5, 6, 7, 8, or 16.

Function

void CDCSetDataSize(BYTE dataBits)

7.1.4.1.11 CDCSetLineCoding Macro

This function is used to manually set the data reported back to the host during a get line coding request. (optional)

File

usb_function_cdc.h

C

```
#define CDCSetLineCoding(baud,format,parity,dataSize) {\
            CDCSetBaudRate(baud);\
            CDCSetCharacterFormat(format);\
            CDCSetParity(parity);\
            CDCSetDataSize(dataSize);\
        }
```

Description

This function is used to manually set the data reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetLineCoding(19200, NUM_STOP_BITS_1, PARITY_NONE, 8);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
DWORD baud	The desired baudrate
BYTE format	number of stop bits. Available options are: • NUM_STOP_BITS_1 (see page 255) - 1 Stop bit • NUM_STOP_BITS_1_5 (see page 256) - 1.5 Stop bits • NUM_STOP_BITS_2 (see page 257) - 2 Stop bits
BYTE parity	Type of parity. The options are the following: • PARITY_NONE (see page 260) • PARITY_ODD (see page 261) • PARITY_EVEN (see page 258) • PARITY_MARK (see page 259) • PARITY_SPACE (see page 262)
BYTE dataSize	number of data bits. The options are 5, 6, 7, 8, or 16.

Function

void CDCSetLineCoding(DWORD baud, BYTE format, BYTE parity, BYTE dataSize)

7.1.4.1.12 CDCSetParity Macro

This function is used manually set the parity format reported back to the host during a get line coding request. (optional)

File

usb_function_cdc.h

C

```
#define CDCSetParity(parityType) {line_coding.bParityType=parityType;}
```

Description

This macro is used manually set the parity format reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetParity(PARITY_NONE);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE parityType	Type of parity. The options are the following: • PARITY_NONE (see page 260) • PARITY_ODD (see page 261) • PARITY_EVEN (see page 258) • PARITY_MARK (see page 259) • PARITY_SPACE (see page 262)

Function

void CDCSetParity(BYTE parityType)

7.1.4.1.13 USBUSARTIsTxTrfReady Macro

This macro is used to check if the CDC class is ready to send more data.

File

usb_function_cdc.h

C

```
#define USBUSARTIsTxTrfReady (cdc_trf_state == CDC_TX_READY)
```

Description

This macro is used to check if the CDC class handler firmware is ready to send more data to the host over the CDC bulk IN endpoint.

Typical Usage:

```
if(USBUSARTIsTxTrfReady())
{
    putrsUSBUSART("Hello World");
}
```

Remarks

Make sure the application periodically calls the CDCTxService (see page 242)() handler, or pending USB IN transfers will not be able to advance and complete.

Preconditions

The return value of this function is only valid if the device is in a configured state (i.e. - USBDeviceGetState() returns CONFIGURED_STATE)

Function

BOOL USBUSARTIsTxTrfReady(void)

7.1.4.2 Data Types and Constants

Macros

	Name	Description
	NUM_STOP_BITS_1 (see page 255)	1 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()
	NUM_STOP_BITS_1_5 (see page 256)	1.5 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()
	NUM_STOP_BITS_2 (see page 257)	2 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()
	PARITY_EVEN (see page 258)	even parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()
	PARITY_MARK (see page 259)	mark parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()
	PARITY_NONE (see page 260)	no parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()
	PARITY_ODD (see page 261)	odd parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()
	PARITY_SPACE (see page 262)	space parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

Description

7.1.4.2.1 NUM_STOP_BITS_1 Macro

File

usb_function_cdc.h

C

```
#define NUM_STOP_BITS_1 0   //1 stop bit - used by CDCSetLineCoding() and
CDCSetCharacterFormat()
```

Description

1 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()

7.1.4.2.2 NUM_STOP_BITS_1_5 Macro

File

usb_function_cdc.h

C

```
#define NUM_STOP_BITS_1_5 1   //1.5 stop bit - used by CDCSetLineCoding() and
CDCSetCharacterFormat()
```

Description

1.5 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()

7.1.4.2.3 NUM_STOP_BITS_2 Macro

File

usb_function_cdc.h

C

```
#define NUM_STOP_BITS_2 2   //2 stop bit - used by CDCSetLineCoding() and
CDCSetCharacterFormat()
```

Description

2 stop bit - used by CDCSetLineCoding (see page 251)() and CDCSetCharacterFormat (see page 249)()

7.1.4.2.4 PARITY_EVEN Macro

File

usb_function_cdc.h

C

```
#define PARITY_EVEN 2 //even parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

even parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

7.1.4.2.5 PARITY_MARK Macro

File

usb_function_cdc.h

C

```
#define PARITY_MARK 3 //mark parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

mark parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

7.1.4.2.6 PARITY_NONE Macro

File

usb_function_cdc.h

C

```
#define PARITY_NONE 0 //no parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

no parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

7.1.4.2.7 PARITY_ODD Macro

File

usb_function_cdc.h

C

```
#define PARITY_ODD 1 //odd parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

odd parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

7.1.4.2.8 PARITY_SPACE Macro

File

usb_function_cdc.h

C

```
#define PARITY_SPACE 4 //space parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

space parity - used by CDCSetLineCoding (see page 251)() and CDCSetParity (see page 252)()

7.1.5 HID Function Driver

7.1.5.1 Interface Routines

Macros

	Name	Description
	HIDRxHandleBusy (see page 264)	Retreives the status of the buffer ownership
	HIDRxPacket (see page 265)	Receives the specified data out the specified endpoint
	HIDTxHandleBusy (see page 266)	Retreives the status of the buffer ownership
	HIDTxPacket (see page 267)	Sends the specified data out the specified endpoint

Description

7.1.5.1.1 HIDRxHandleBusy Macro

Retreives the status of the buffer ownership

File

usb_function_hid.h

C

```
#define HIDRxHandleBusy(handle) USBHandleBusy(handle)
```

Description

Retreives the status of the buffer ownership. This function will indicate if the previous transfer is complete or not.

This function will take the input handle (pointer to a BDT entry) and will check the UOWN bit. If the UOWN bit is set then that indicates that the transfer is not complete and the USB module still owns the data memory. If the UOWN bit is clear that means that the transfer is complete and that the CPU now owns the data memory.

For more information about the BDT, please refer to the appropriate datasheet for the device in use.

Typical Usage:

```
if(!HIDRxHandleBusy(USBOutHandle))
{
    //The data is available in the buffer that was specified when the
    //  HIDRxPacket() was called.
}
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
USB_HANDLE handle	the handle for the transfer in question. The handle is returned by the HIDTxPacket (see page 267)() and HIDRxPacket (see page 265)() functions. Please insure that USB_HANDLE (see page 227) objects are initialized to NULL.

Return Values

Return Values	Description
TRUE	the HID handle is still busy
FALSE	the HID handle is not busy and is ready to receive additional data.

Function

BOOL HIDRxHandleBusy(USB_HANDLE (see page 227) handle)

7.1.5.1.2 HIDRxPacket Macro

Receives the specified data out the specified endpoint

File

usb_function_hid.h

C

```
#define HIDRxPacket USBRxOnePacket
```

Description

Receives the specified data out the specified endpoint.

Typical Usage:

```
//Read 64-bytes from endpoint HID_EP, into the ReceivedDataBuffer array.
//  Make sure to save the return handle so that we can check it later
//  to determine when the transfer is complete.
USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	the endpoint you want to receive the data into
BYTE* data	pointer to where the data will go when it arrives
WORD len	the length of the data that you wish to receive

Return Values

Return Values	Description
USB_HANDLE (see page 227)	a handle for the transfer. This information should be kept to track the status of the transfer

Function

USB_HANDLE (see page 227) HIDRxPacket(BYTE ep, BYTE* data, WORD len)

7.1.5.1.3 HIDTxHandleBusy Macro

Retreives the status of the buffer ownership

File

usb_function_hid.h

C

```
#define HIDTxHandleBusy(handle) USBHandleBusy(handle)
```

Description

Retreives the status of the buffer ownership. This function will indicate if the previous transfer is complete or not.

This function will take the input handle (pointer to a BDT entry) and will check the UOWN bit. If the UOWN bit is set then that indicates that the transfer is not complete and the USB module still owns the data memory. If the UOWN bit is clear that means that the transfer is complete and that the CPU now owns the data memory.

For more information about the BDT, please refer to the appropriate datasheet for the device in use.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if(!HIDTxHandleBusy(USBInHandle))
{
    //Send the data contained in the ToSendDataBuffer[] array out on
    //  endpoint HID_EP
    USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],sizeof(ToSendDataBuffer));
}
```

Remarks

None

Preconditions

None.

Parameters

Parameters	Description
USB_HANDLE handle	the handle for the transfer in question. The handle is returned by the HIDTxPacket (see page 267)() and HIDRxPacket (see page 265)() functions. Please insure that USB_HANDLE (see page 227) objects are initialized to NULL.

Return Values

Return Values	Description
TRUE	the HID handle is still busy
FALSE	the HID handle is not busy and is ready to send additional data.

Function

BOOL HIDTxHandleBusy(USB_HANDLE (see page 227) handle)

7.1.5.1.4 HIDTxPacket Macro

Sends the specified data out the specified endpoint

File

usb_function_hid.h

C

```
#define HIDTxPacket USBTxOnePacket
```

Description

This function sends the specified data out the specified endpoint and returns a handle to the transfer information.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if(!HIDTxHandleBusy(USBInHandle))
{
    //Send the data contained in the ToSendDataBuffer[] array out on
    //  endpoint HID_EP
    USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],sizeof(ToSendDataBuffer));
}
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	the endpoint you want to send the data out of
BYTE* data	pointer to the data that you wish to send
WORD len	the length of the data that you wish to send

Return Values

Return Values	Description
USB_HANDLE (see page 227)	a handle for the transfer. This information should be kept to track the status of the transfer

Function

USB_HANDLE (see page 227) HIDTxPacket(BYTE ep, BYTE* data, WORD len)

7.1.5.2 Data Types and Constants

Macros

	Name	Description
	BOOT_INTF_SUBCLASS (see page 269)	HID Interface Class SubClass Codes
	BOOT_PROTOCOL (see page 270)	Protocol Selection
	HID_PROTOCOL_KEYBOARD (see page 271)	This is macro HID_PROTOCOL_KEYBOARD.
	HID_PROTOCOL_MOUSE (see page 272)	This is macro HID_PROTOCOL_MOUSE.
	HID_PROTOCOL_NONE (see page 273)	HID Interface Class Protocol Codes

Description

7.1.5.2.1 BOOT_INTF_SUBCLASS Macro

File

usb_function_hid.h

C

```
#define BOOT_INTF_SUBCLASS 0x01
```

Description

HID Interface Class SubClass Codes

7.1.5.2.2 BOOT_PROTOCOL Macro

File

usb_function_hid.h

C

```
#define BOOT_PROTOCOL 0x00
```

Description

Protocol Selection

7.1.5.2.3 HID_PROTOCOL_KEYBOARD Macro

File

usb_function_hid.h

C

```
#define HID_PROTOCOL_KEYBOARD 0x01
```

Description

This is macro HID_PROTOCOL_KEYBOARD.

7.1.5.2.4 HID_PROTOCOL_MOUSE Macro

File

usb_function_hid.h

C

```
#define HID_PROTOCOL_MOUSE 0x02
```

Description

This is macro HID_PROTOCOL_MOUSE.

7.1.5.2.5 HID_PROTOCOL_NONE Macro

File

usb_function_hid.h

C

```
#define HID_PROTOCOL_NONE 0x00
```

Description

HID Interface Class Protocol Codes

7.1.6 MSD Function Driver

7.1.6.1 Interface Routines

Functions

	Name	Description
	MSDTasks (see page 275)	This is function MSDTasks.
	USBCheckMSDRequest (see page 276)	
	USBMSDInit (see page 277)	This is function USBMSDInit.

Description

7.1.6.1.1 MSDTasks Function

File

usb_function_msd.h

C

```
BYTE MSDTasks();
```

Description

This is function MSDTasks.

7.1.6.1.2 USBCheckMSDRequest Function

File

usb_function_msd.h

C

```
void USBCheckMSDRequest();
```

Section

Public Prototypes

7.1.6.1.3 USBMSDInit Function

File

usb_function_msd.h

C

```
void USBMSDInit();
```

Description

This is function USBMSDInit.

7.1.6.2 Data Types and Constants

Types

	Name	Description
	LUN_FUNCTIONS (see page 279)	LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

Description

7.1.6.2.1 LUN_FUNCTIONS Type

LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

File

usb_function_msd.h

C

```
typedef struct LUN_FUNCTIONS@1 LUN_FUNCTIONS;
```

Description

LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

Typical Usage:

```
LUN_FUNCTIONS LUN[MAX_LUN + 1] =
{
    {
        &MDD_SDSPI_MediaInitialize,
        &MDD_SDSPI_ReadCapacity,
        &MDD_SDSPI_ReadSectorSize,
        &MDD_SDSPI_MediaDetect,
        &MDD_SDSPI_SectorRead,
        &MDD_SDSPI_WriteProtectState,
        &MDD_SDSPI_SectorWrite
    }
};
```

In the above code we are passing the address of the SDSPI functions to the corresponding member of the LUN_FUNCTIONS structure. In the above case we have created an array of LUN_FUNCTIONS structures so that it is possible to have multiple physical layers by merely increasing the MAX_LUN variable and by adding one more set of entries in the array. Please take caution to insure that each function is in the the correct location in the structure. Incorrect alignment will cause the USB stack to call the incorrect function for a given command.

See the MDD File System Library for additional information about the available physical media, their requirements, and how to use their associated functions.

7.1.7 Personal Healthcare Device Class (PHDC) Function Driver

7.1.7.1 Interface Routines

Functions

	Name	Description
	USBDevicePHDCCheckRequest (see page 281)	This routine checks the setup data packet to see if it is class specific request or vendor specific request and handles it
	USBDevicePHDCInit (see page 282)	This function initializes the PHDC function driver. This function should be called after the SET_CONFIGURATION command.
	USBDevicePHDCReceiveData (see page 283)	USBDevicePHDCReceiveData copies a string of BYTEs received through USB PHDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.
	USBDevicePHDCSendData (see page 284)	USBDevicePHDCSendData writes an array of data to the USB.
	USBDevicePHDCTxRXService (see page 285)	USBDevicePHDCTxRXService handles device-to-host transaction(s) and host-to-device transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.
	USBDevicePHDCUpdateStatus (see page 286)	USBDevicePHDCUpdateStatus Function Gets the current status of an Endpoint and holds the status in variable phdcEpDataBitmap. The Status is sent to the host upon the "Get Data Status" request from the host.

Description

7.1.7.1.1 USBDevicePHDCCheckRequest Function

File

usb_function_phdc.h

C

```
void USBDevicePHDCCheckRequest();
```

Description

This routine checks the setup data packet to see if it is class specific request or vendor specific request and handles it

Remarks

None

Preconditions

None

Function

void USBDevicePHDCCheckRequest(void)

7.1.7.1.2 USBDevicePHDCInit Function

This function initializes the PHDC function driver. This function should be called after the SET_CONFIGURATION command.

File

usb_function_phdc.h

C

```
void USBDevicePHDCInit(
    USB_PHDC_CB
);
```

Description

This function initializes the PHDC function driver. This function sets the default line coding (baud rate, bit parity, number of data bits, and format). This function also enables the endpoints and prepares for the first transfer from the host.

This function should be called after the SET_CONFIGURATION command. This is most simply done by calling this function from the USBCBInitEP() function.

Typical Usage:

```
void USBCBInitEP(void)
{
    PHDCInitEP();
}
```

Remarks

None

Preconditions

None

Function

void PHDCInitEP(void)

7.1.7.1.3 USBDevicePHDCReceiveData Function

USBDevicePHDCReceiveData copies a string of BYTEs received through USB PHDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

File

usb_function_phdc.h

C

```
UINT8 USBDevicePHDCReceiveData(
    UINT8 qos,
    UINT8 * buffer,
    UINT16 len
);
```

Description

USBDevicePHDCReceiveData copies a string of BYTEs received through USB PHDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

Typical Usage:

```
BYTE numBytes;
BYTE buffer[64]

numBytes = USBDevicePHDCReceiveData(buffer,sizeof(buffer)); //until the buffer is free.
if(numBytes > 0)
{
    //we received numBytes bytes of data and they are copied into
    //  the "buffer" variable.  We can do something with the data
    //  here.
}
```

Preconditions

Value of input argument 'len' should be smaller than the maximum endpoint size responsible for receiving bulk data from USB host for PHDC class. Input argument 'buffer' should point to a buffer area that is bigger or equal to the size specified by 'len'.

Parameters

Parameters	Description
qos	quality of service
buffer	Pointer to where received BYTEs are to be stored
len	The number of BYTEs expected.

Function

UINT8 USBDevicePHDCReceiveData(UINT8 qos, UINT8 *buffer, UINT16 len)

7.1.7.1.4 USBDevicePHDCSendData Function

USBDevicePHDCSendData writes an array of data to the USB.

File

usb_function_phdc.h

C

```
void USBDevicePHDCSendData(
    UINT8 qos,
    UINT8 * data,
    UINT16 length,
    BOOL memtype
);
```

Description

USBDevicePHDCSendData writes an array of data to the USB.

Typical Usage:

```
    if(USBUSARTIsTxTrfReady())
    {
        char data[] = {0x00, 0x01, 0x02, 0x03, 0x04};
        USBDevicePHDCSendData(1,data,5);
    }
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. USBDevicePHDCTxRXService (see page 285)() must be called periodically to keep sending blocks of data to the host.

Preconditions

USBUSARTIsTxTrfReady (see page 253)() must return TRUE. This indicates that the last transfer is complete and is ready to receive a new block of data.

Parameters

Parameters	Description
qos	Quality of service information
*data	pointer to a RAM array of data to be transfered to the host
length	the number of bytes to be transfered.

Function

void USBDevicePHDCSendData(UINT8 qos, UINT8 *data, UINT8 Length)

7.1.7.1.5 USBDevicePHDCTxRXService Function

USBDevicePHDCTxRXService handles device-to-host transaction(s) and host-to-device transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

File

usb_function_phdc.h

C

```
void USBDevicePHDCTxRXService(
    USTAT_FIELDS* event
);
```

Description

USBDevicePHDCTxRXService handles device-to-host transaction(s) and host-to-device transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

Typical Usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == TRUE))
        {
            //Either the device is not configured or we are suspended
            //  so we don't want to do execute any application code
            continue;   //go back to the top of the while loop
        }
        else
        {
            //Keep trying to send data to the PC as required
            USBDevicePHDCTxRXService();

            //Run application code.
            UserApplication();
        }
    }
}
```

Remarks

None

Preconditions

None

Function

void USBDevicePHDCTxRXService(void)

7.1.7.1.6 USBDevicePHDCUpdateStatus Function

USBDevicePHDCUpdateStatus Function Gets the current status of an Endpoint and holds the status in variable phdcEpDataBitmap. The Status is sent to the host upon the "Get Data Status" request from the host.

File

usb_function_phdc.h

C

```
void USBDevicePHDCUpdateStatus(
    WORD EndpointNo,
    BIT Status
);
```

Description

USBDevicePHDCUpdateStatus Function helps to handle the "Get Data Status" PHDC specfic request received from the Host as mentioned in the section 7.1.2 of the Personal Healthcare Devices Specification. This function Gets the current status of an Endpoint and holds the status in variable phdcEpDataBitmap.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
WORD EndpointNo	The number of the endpoint, for which the status is requested.
BIT Status	Current status of the Endpoint.

Function

void USBDevicePHDCUpdateStatus (WORD EndpointNo, BIT Status)

7.1.8 Vendor Class (Generic) Function Driver

7.1.8.1 Interface Routines

Macros

	Name	Description
	USBGenRead (see page 288)	Receives the specified data out the specified endpoint
	USBGenWrite (see page 289)	Sends the specified data out the specified endpoint

Description

7.1.8.1.1 USBGenRead Macro

Receives the specified data out the specified endpoint

File

usb_function_generic.h

C

```
#define USBGenRead(ep,data,len) USBRxOnePacket(ep,data,len)
```

Description

Receives the specified data out the specified endpoint.

Typical Usage:

```
//Read 64-bytes from endpoint USBGEN_EP_NUM, into the OUTPacket array.
//  Make sure to save the return handle so that we can check it later
//  to determine when the transfer is complete.
if(!USBHandleBusy(USBOutHandle))
{
    USBOutHandle = USBGenRead(USBGEN_EP_NUM,(BYTE*)&OUTPacket,64);
}
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	the endpoint you want to receive the data into
BYTE* data	pointer to where the data will go when it arrives
WORD len	the length of the data that you wish to receive

Return Values

Return Values	Description
USB_HANDLE (see page 227)	a handle for the transfer. This information should be kept to track the status of the transfer

Function

USB_HANDLE (see page 227) USBGenRead(BYTE ep, BYTE* data, WORD len)

7.1.8.1.2 USBGenWrite Macro

Sends the specified data out the specified endpoint

File

usb_function_generic.h

C

```
#define USBGenWrite(ep,data,len) USBTxOnePacket(ep,data,len)
```

Description

This function sends the specified data out the specified endpoint and returns a handle to the transfer information.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if(!USBHandleBusy(USBGenericInHandle))
{
    //Send the data contained in the INPacket[] array out on
    //  endpoint USBGEN_EP_NUM
    USBGenericInHandle = USBGenWrite(USBGEN_EP_NUM,(BYTE*)&INPacket[0],sizeof(INPacket));
}
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE ep	the endpoint you want to send the data out of
BYTE* data	pointer to the data that you wish to send
WORD len	the length of the data that you wish to send

Return Values

Return Values	Description
USB_HANDLE (see page 227)	a handle for the transfer. This information should be kept to track the status of the transfer

Function

USB_HANDLE (see page 227) USBGenWrite(BYTE ep, BYTE* data, WORD len)

7.2 Embedded Host API

These are the various client drivers that are available for use with the USB Embedded Host driver.

Description

7.2.1 Embedded Host Stack

The USB Embedded Host driver provides low-level USB functionality for all host client drivers.

Description

The USB Embedded Host driver provides low-level USB functionality for all host client drivers. This layer is responsible for enumerating devices, managing data transfers, and detecting device detach.

Typically, only host client drivers will interact with this layer. Applications can be configured to receive some events from this layer, such as EVENT_REQUEST_POWER and EVENT_RELEASE_POWER.

See *AN1140 USB Embedded Host Stack* for more information about this layer. See *AN1141 USB Embedded Host Stack Programmer's Guide* for more information about creating a client driver that uses this layer.

7.2.1.1 Interface Routines

Functions

	Name	Description
	USB_HOST_APP_EVENT_HANDLER (see page 292)	This is a typedef to use when defining the application level events handler.
	USBHostClearEndpointErrors (see page 293)	This function clears an endpoint's internal error condition.
	USBHostDeviceSpecificClientDriver (see page 294)	This function indicates if the specified device has explicit client driver support specified in the TPL.
	USBHostDeviceStatus (see page 295)	This function returns the current status of a device.
	USBHostInit (see page 300)	This function initializes the variables of the USB host stack.
	USBHostRead (see page 301)	This function initiates a read from the attached device.
	USBHostResetDevice (see page 303)	This function resets an attached device.
	USBHostResumeDevice (see page 304)	This function issues a RESUME to the attached device.
	USBHostSetDeviceConfiguration (see page 305)	This function changes the device's configuration.
	USBHostSetNAKTimeout (see page 307)	This function specifies NAK timeout capability.
	USBHostSuspendDevice (see page 308)	This function suspends a device.
	USBHostTerminateTransfer (see page 309)	This function terminates the current transfer for the given endpoint.
	USBHostTransferIsComplete (see page 310)	This function initiates whether or not the last endpoint transaction is complete.
	USBHostVbusEvent (see page 312)	This function handles Vbus events that are detected by the application.
	USBHostWrite (see page 313)	This function initiates a write to the attached device.

Macros

	Name	Description
	USBHostGetCurrentConfigurationDescriptor (see page 296)	This function returns a pointer to the current configuration descriptor of the requested device.
	USBHostGetDeviceDescriptor (see page 297)	This function returns a pointer to the device descriptor of the requested device.
	USBHostGetStringDescriptor (see page 298)	This routine initiates a request to obtains the requested string descriptor.

Description

7.2.1.1.1 USB_HOST_APP_EVENT_HANDLER Function

This is a typedef to use when defining the application level events handler.

File

usb_host.h

C

```
BOOL USB_HOST_APP_EVENT_HANDLER(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is implemented by the application. The function name can be anything - the macro USB_HOST_APP_EVENT_HANDLER must be set in usb_config.h to the name of the application function.

In the application layer, this function is responsible for handling all application-level events that are generated by the stack. See the enumeration USB_EVENT for a complete list of all events that can occur. Note that some of these events are intended for client drivers (e.g. EVENT_TRANSFER), while some are intended for for the application layer (e.g. EVENT_UNSUPPORTED_DEVICE).

If the application can handle the event successfully, the function should return TRUE. For example, if the function receives the event EVENT_VBUS_REQUEST_POWER and the system can allocate that much power to an attached device, the function should return TRUE. If, however, the system cannot allocate that much power to an attached device, the function should return FALSE.

Remarks

If this function is not provided by the application, then all application events are assumed to function without error.

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the USB device generating the event
USB_EVENT event	Event that occurred
void *data	Optional pointer to data for the event
DWORD size	Size of the data pointed to by *data

Return Values

Return Values	Description
TRUE	Event was processed successfully
FALSE	Event was not processed successfully

Function

BOOL USB_HOST_APP_EVENT_HANDLER (BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.1.1.2 USBHostClearEndpointErrors Function

This function clears an endpoint's internal error condition.

File

usb_host.h

C

```
BYTE USBHostClearEndpointErrors(
    BYTE deviceAddress,
    BYTE endpoint
);
```

Description

This function is called to clear the internal error condition of a device's endpoint. It should be called after the application has dealt with the error condition on the device. This routine clears internal status only; it does not interact with the device.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of device
BYTE endpoint	Endpoint to clear error condition

Return Values

Return Values	Description
USB_SUCCESS	Errors cleared
USB_UNKNOWN_DEVICE	Device not found
USB_ENDPOINT_NOT_FOUND	Specified endpoint not found

Function

BYTE USBHostClearEndpointErrors(BYTE deviceAddress, BYTE endpoint)

7.2.1.1.3 USBHostDeviceSpecificClientDriver Function

This function indicates if the specified device has explicit client driver support specified in the TPL.

File

usb_host.h

C

```
BOOL USBHostDeviceSpecificClientDriver(
    BYTE deviceAddress
);
```

Description

This function indicates if the specified device has explicit client driver support specified in the TPL. It is used in client drivers' USB_CLIENT_INIT (see page 321) routines to indicate that the client driver should be used even though the class, subclass, and protocol values may not match those normally required by the class. For example, some printing devices do not fulfill all of the requirements of the printer class, so their class, subclass, and protocol fields indicate a custom driver rather than the printer class. But the printer class driver can still be used, with minor limitations.

Remarks

This function is used so client drivers can allow certain devices to enumerate. For example, some printer devices indicate a custom class rather than the printer class, even though the device has only minor limitations from the full printer class. The printer client driver will fail to initialize the device if it does not indicate printer class support in its interface descriptor. The printer client driver could allow any device with an interface that matches the printer class endpoint configuration, but both printer and mass storage devices utilize one bulk IN and one bulk OUT endpoint. So a mass storage device would be erroneously initialized as a printer device. This function allows a client driver to know that the client driver support was specified explicitly in the TPL, so for this particular device only, the class, subclass, and protocol fields can be safely ignored.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of device

Return Values

Return Values	Description
TRUE	This device is listed in the TPL by VID andPID, and has explicit client driver support.
FALSE	This device is not listed in the TPL by VID and PID.

Function

BOOL USBHostDeviceSpecificClientDriver(BYTE deviceAddress)

7.2.1.1.4 USBHostDeviceStatus Function

This function returns the current status of a device.

File

usb_host.h

C

```
BYTE USBHostDeviceStatus(
    BYTE deviceAddress
);
```

Description

This function returns the current status of a device. If the device is in a holding state due to an error, the error is returned.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_DEVICE_ATTACHED	Device is attached and running
USB_DEVICE_DETACHED	No device is attached
USB_DEVICE_ENUMERATING	Device is enumerating
USB_HOLDING_OUT_OF_MEMORY	Not enough heap space available
USB_HOLDING_UNSUPPORTED_DEVICE	Invalid configuration or unsupported class
USB_HOLDING_UNSUPPORTED_HUB	Hubs are not supported
USB_HOLDING_INVALID_CONFIGURATION	Invalid configuration requested
USB_HOLDING_PROCESSING_CAPACITY	Processing requirement excessive
USB_HOLDING_POWER_REQUIREMENT	Power requirement excessive
USB_HOLDING_CLIENT_INIT_ERROR	Client driver failed to initialize
USB_DEVICE_SUSPENDED	Device is suspended
Other	Device is holding in an error state. The return value indicates the error.

Function

BYTE USBHostDeviceStatus(BYTE deviceAddress)

7.2.1.1.5 USBHostGetCurrentConfigurationDescriptor Macro

File

usb_host.h

C

```
#define USBHostGetCurrentConfigurationDescriptor( deviceAddress) (
pCurrentConfigurationDescriptor )
```

Returns

BYTE * - Pointer to the Configuration Descriptor.

Description

This function returns a pointer to the current configuration descriptor of the requested device.

Remarks

This will need to be expanded to a full function when multiple device support is added.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of device

Function

BYTE * USBHostGetCurrentConfigurationDescriptor(BYTE deviceAddress)

7.2.1.1.6 USBHostGetDeviceDescriptor Macro

File

usb_host.h

C

```
#define USBHostGetDeviceDescriptor( deviceAddress ) ( pDeviceDescriptor )
```

Returns

BYTE * - Pointer to the Device Descriptor.

Description

This function returns a pointer to the device descriptor of the requested device.

Remarks

This will need to be expanded to a full function when multiple device support is added.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of device

Function

BYTE * USBHostGetDeviceDescriptor(BYTE deviceAddress)

7.2.1.1.7 USBHostGetStringDescriptor Macro

This routine initiates a request to obtains the requested string descriptor.

File

usb_host.h

C

```
#define USBHostGetStringDescriptor( deviceAddress, stringNumber, LangID, stringDescriptor,
stringLength, clientDriverID ) \
        USBHostIssueDeviceRequest( deviceAddress, USB_SETUP_DEVICE_TO_HOST |
USB_SETUP_TYPE_STANDARD | USB_SETUP_RECIPIENT_DEVICE,    \
                USB_REQUEST_GET_DESCRIPTOR, (USB_DESCRIPTOR_STRING << 8) |
stringNumber,                                            \
                LangID, stringLength, stringDescriptor, USB_DEVICE_REQUEST_GET,
clientDriverID )
```

Description

This routine initiates a request to obtains the requested string descriptor. If the request cannot be started, the routine returns an error. Otherwise, the request is started, and the requested string descriptor is stored in the designated location.

Example Usage:

```
USBHostGetStringDescriptor(
    deviceAddress,
    stringDescriptorNum,
    LangID,
    stringDescriptorBuffer,
    sizeof(stringDescriptorBuffer),
    0xFF
    );

while(1)
{
    if(USBHostTransferIsComplete( deviceAddress , 0, &errorCode, &byteCount))
    {
        if(errorCode)
        {
            //There was an error reading the string, bail out of loop
        }
        else
        {
            //String is located in specified buffer, do something with it.

            //The length of the string is both in the byteCount variable
            //  as well as the first byte of the string itself
        }
        break;
    }
    USBTasks();
}
```

Remarks

The returned string descriptor will be in the exact format as obtained from the device. The length of the entire descriptor will be in the first byte, and the descriptor type will be in the second. The string itself is represented in UNICODE. Refer to the USB 2.0 Specification for more information about the format of string descriptors.

Preconditions

None

Parameters

Parameters	Description
deviceAddress	Address of the device
stringNumber	Index of the desired string descriptor

LangID	The Language ID of the string to read (should be 0 if trying to read the language ID list
*stringDescriptor	Pointer to where to store the string.
stringLength	Maximum length of the returned string.
clientDriverID	Client driver to return the completion event to.

Return Values

Return Values	Description
USB_SUCCESS	The request was started successfully.
USB_UNKNOWN_DEVICE	Device not found
USB_INVALID_STATE	We must be in a normal running state.
USB_ENDPOINT_BUSY	The endpoint is currently processing a request.

Function

BYTE USBHostGetStringDescriptor (BYTE deviceAddress, BYTE stringNumber,

BYTE LangID, BYTE *stringDescriptor, BYTE stringLength,

BYTE clientDriverID)

7.2.1.1.8 USBHostInit Function

This function initializes the variables of the USB host stack.

File

usb_host.h

C

```
BOOL USBHostInit(
    unsigned long flags
);
```

Description

This function initializes the variables of the USB host stack. It does not initialize the hardware. The peripheral itself is initialized in one of the state machine states. Therefore, USBHostTasks() should be called soon after this function.

Remarks

If the endpoint list is empty, an entry is created in the endpoint list for EP0. If the list is not empty, free all allocated memory other than the EP0 node. This allows the routine to be called multiple times by the application.

Preconditions

None

Parameters

Parameters	Description
flags	reserved

Return Values

Return Values	Description
TRUE	Initialization successful
FALSE	Could not allocate memory.

Function

BOOL USBHostInit(unsigned long flags)

7.2.1.1.9 USBHostRead Function

This function initiates a read from the attached device.

File

usb_host.h

C

```
BYTE USBHostRead(
    BYTE deviceAddress,
    BYTE endpoint,
    BYTE * data,
    DWORD size
);
```

Description

This function initiates a read from the attached device.

If the endpoint is isochronous, special conditions apply. The pData and size parameters have slightly different meanings, since multiple buffers are required. Once started, an isochronous transfer will continue with no upper layer intervention until USBHostTerminateTransfer (see page 309)() is called. The ISOCHRONOUS_DATA_BUFFERS structure should not be manipulated until the transfer is terminated.

To clarify parameter usage and to simplify casting, use the macro USBHostReadIsochronous() when reading from an isochronous endpoint.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE endpoint	Endpoint number
BYTE *pData	Pointer to where to store the data. If the endpoint is isochronous, this points to an ISOCHRONOUS_DATA_BUFFERS structure, with multiple data buffer pointers.
DWORD size	Number of data bytes to read. If the endpoint is isochronous, this is the number of data buffer pointers pointed to by pData.

Return Values

Return Values	Description
USB_SUCCESS	Read started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlRead to read from a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must read from an IN endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Read is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

BYTE USBHostRead(BYTE deviceAddress, BYTE endpoint, BYTE *pData, DWORD size)

7.2.1.1.10 USBHostResetDevice Function

This function resets an attached device.

File

usb_host.h

C

```
BYTE USBHostResetDevice(
    BYTE deviceAddress
);
```

Description

This function places the device back in the RESET state, to issue RESET signaling. It can be called only if the state machine is not in the DETACHED state.

Remarks

In order to do a full clean-up, the state is set back to STATE_DETACHED rather than a reset state. The ATTACH interrupt will automatically be triggered when the module is re-enabled, and the proper reset will be performed.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Device cannot RESUME unless it is suspended

Function

BYTE USBHostResetDevice(BYTE deviceAddress)

7.2.1.1.11 USBHostResumeDevice Function

This function issues a RESUME to the attached device.

File

usb_host.h

C

```
BYTE USBHostResumeDevice(
    BYTE deviceAddress
);
```

Description

This function issues a RESUME to the attached device. It can called only if the state machine is in the suspend state.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Device cannot RESUME unless it is suspended

Function

BYTE USBHostResumeDevice(BYTE deviceAddress)

7.2.1.1.12 USBHostSetDeviceConfiguration Function

This function changes the device's configuration.

File

usb_host.h

C

```
BYTE USBHostSetDeviceConfiguration(
    BYTE deviceAddress,
    BYTE configuration
);
```

Description

This function is used by the application to change the device's Configuration. This function must be used instead of USBHostIssueDeviceRequest(), because the endpoint definitions may change.

To see when the reconfiguration is complete, use the USBHostDeviceStatus (see page 295)() function. If configuration is still in progress, this function will return USB_DEVICE_ENUMERATING.

Remarks

If an invalid configuration is specified, this function cannot return an error. Instead, the event USB_UNSUPPORTED_DEVICE will the sent to the application layer and the device will be placed in a holding state with a USB_HOLDING_UNSUPPORTED_DEVICE error returned by USBHostDeviceStatus (see page 295)().

Preconditions

The host state machine should be in the running state, and no reads or writes should be in progress.

Example

```
rc = USBHostSetDeviceConfiguration( attachedDevice, configuration );
if (rc)
{
    // Error - cannot set configuration.
}
else
{
    while (USBHostDeviceStatus( attachedDevice ) == USB_DEVICE_ENUMERATING)
    {
        USBHostTasks();
    }
}
if (USBHostDeviceStatus( attachedDevice ) != USB_DEVICE_ATTACHED)
{
    // Error - cannot set configuration.
}
```

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE configuration	Index of the new configuration

Return Values

Return Values	Description
USB_SUCCESS	Process of changing the configuration was started successfully.
USB_UNKNOWN_DEVICE	Device not found
USB_INVALID_STATE	This function cannot be called during enumeration or while performing a device request.
USB_BUSY	No IN or OUT transfers may be in progress.

Function

BYTE USBHostSetDeviceConfiguration(BYTE deviceAddress, BYTE configuration)

7.2.1.1.13 USBHostSetNAKTimeout Function

This function specifies NAK timeout capability.

File

usb_host.h

C

```
BYTE USBHostSetNAKTimeout(
    BYTE deviceAddress,
    BYTE endpoint,
    WORD flags,
    WORD timeoutCount
);
```

Description

This function is used to set whether or not an endpoint on a device should time out a transaction based on the number of NAKs received, and if so, how many NAKs are allowed before the timeout.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE endpoint	Endpoint number to configure
WORD flags	Bit 0: • 0 = disable NAK timeout • 1 = enable NAK timeout
WORD timeoutCount	Number of NAKs allowed before a timeout

Return Values

Return Values	Description
USB_SUCCESS	NAK timeout was configured successfully.
USB_UNKNOWN_DEVICE	Device not found.
USB_ENDPOINT_NOT_FOUND	The specified endpoint was not found.

Function

BYTE USBHostSetNAKTimeout(BYTE deviceAddress, BYTE endpoint, WORD flags,

WORD timeoutCount)

7.2.1.1.14 USBHostSuspendDevice Function

This function suspends a device.

File

usb_host.h

C

```
BYTE USBHostSuspendDevice(
    BYTE deviceAddress
);
```

Description

This function put a device into an IDLE state. It can only be called while the state machine is in normal running mode. After 3ms, the attached device should go into SUSPEND mode.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device to suspend

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Cannot suspend unless device is in normal run mode

Function

BYTE USBHostSuspendDevice(BYTE deviceAddress)

7.2.1.1.15 USBHostTerminateTransfer Function

This function terminates the current transfer for the given endpoint.

File

usb_host.h

C

```
void USBHostTerminateTransfer(
    BYTE deviceAddress,
    BYTE endpoint
);
```

Returns

None

Description

This function terminates the current transfer for the given endpoint. It can be used to terminate reads or writes that the device is not responding to. It is also the only way to terminate an isochronous transfer.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE endpoint	Endpoint number

Function

void USBHostTerminateTransfer(BYTE deviceAddress, BYTE endpoint)

7.2.1.1.16 USBHostTransferIsComplete Function

This function initiates whether or not the last endpoint transaction is complete.

File

usb_host.h

C

```
BOOL USBHostTransferIsComplete(
    BYTE deviceAddress,
    BYTE endpoint,
    BYTE * errorCode,
    DWORD * byteCount
);
```

Description

This function initiates whether or not the last endpoint transaction is complete. If it is complete, an error code and the number of bytes transferred are returned.

For isochronous transfers, byteCount is not valid. Instead, use the returned byte counts for each EVENT_TRANSFER event that was generated during the transfer.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful
- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE endpoint	Endpoint number
BYTE *errorCode	Error code indicating the status of the transfer. Only valid if the transfer is complete.
DWORD *byteCount	The number of bytes sent or received. Invalid for isochronous transfers.

Return Values

Return Values	Description
TRUE	Transfer is complete.
FALSE	Transfer is not complete.

Function

BOOL USBHostTransferIsComplete(BYTE deviceAddress, BYTE endpoint,

BYTE *errorCode, DWORD *byteCount)

7.2.1.1.17 USBHostVbusEvent Function

This function handles Vbus events that are detected by the application.

File

usb_host.h

C

```
BYTE USBHostVbusEvent(
    USB_EVENT vbusEvent,
    BYTE hubAddress,
    BYTE portNumber
);
```

Description

This function handles Vbus events that are detected by the application. Since Vbus management is application dependent, the application is responsible for monitoring Vbus and detecting overcurrent conditions and removal of the overcurrent condition. If the application detects an overcurrent condition, it should call this function with the event EVENT_VBUS_OVERCURRENT with the address of the hub and port number that has the condition. When a port returns to normal operation, the application should call this function with the event EVENT_VBUS_POWER_AVAILABLE so the stack knows that it can allow devices to attach to that port.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
USB_EVENT vbusEvent	Vbus event that occured. Valid events: • EVENT_VBUS_OVERCURRENT • EVENT_VBUS_POWER_AVAILABLE
BYTE hubAddress	Address of the hub device (USB_ROOT_HUB for the root hub)
BYTE portNumber	Number of the physical port on the hub (0 - based)

Return Values

Return Values	Description
USB_SUCCESS	Event handled
USB_ILLEGAL_REQUEST	Invalid event, hub, or port

Function

BYTE USBHostVbusEvent(USB_EVENT vbusEvent, BYTE hubAddress,

BYTE portNumber)

7.2.1.1.18 USBHostWrite Function

This function initiates a write to the attached device.

File

usb_host.h

C

```
BYTE USBHostWrite(
    BYTE deviceAddress,
    BYTE endpoint,
    BYTE * data,
    DWORD size
);
```

Description

This function initiates a write to the attached device. The data buffer pointed to by *data must remain valid during the entire time that the write is taking place; the data is not buffered by the stack.

If the endpoint is isochronous, special conditions apply. The pData and size parameters have slightly different meanings, since multiple buffers are required. Once started, an isochronous transfer will continue with no upper layer intervention until USBHostTerminateTransfer (see page 309)() is called. The ISOCHRONOUS_DATA_BUFFERS structure should not be manipulated until the transfer is terminated.

To clarify parameter usage and to simplify casting, use the macro USBHostWriteIsochronous() when writing to an isochronous endpoint.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE endpoint	Endpoint number
BYTE *data	Pointer to where the data is stored. If the endpoint is isochronous, this points to an ISOCHRONOUS_DATA_BUFFERS structure, with multiple data buffer pointers.
DWORD size	Number of data bytes to send. If the endpoint is isochronous, this is the number of data buffer pointers pointed to by pData.

Return Values

Return Values	Description
USB_SUCCESS	Write started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlWrite to write to a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must write to an OUT endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Write is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

BYTE USBHostWrite(BYTE deviceAddress, BYTE endpoint, BYTE *data, DWORD size)

7.2.1.2 Data Types and Constants

Macros

	Name	Description
	USB_NUM_BULK_NAKS (see page 323)	Define how many NAK's are allowed during a bulk transfer before erroring.
	USB_NUM_COMMAND_TRIES (see page 324)	During enumeration, define how many times each command will be tried before giving up and resetting the device.
	USB_NUM_CONTROL_NAKS (see page 325)	Define how many NAK's are allowed during a control transfer before erroring.
	USB_NUM_ENUMERATION_TRIES (see page 326)	Define how many times the host will try to enumerate the device before giving up and setting the state to DETACHED.
	USB_NUM_INTERRUPT_NAKS (see page 327)	Define how many NAK's are allowed during an interrupt OUT transfer before erroring. Interrupt IN transfers that are NAK'd are terminated without error.
	TPL_SET_CONFIG (see page 328)	Bitmask for setting the configuration.
	TPL_CLASS_DRV (see page 329)	Bitmask for class driver support.
	TPL_ALLOW_HNP (see page 330)	Bitmask for Host Negotiation Protocol.

Structures

	Name	Description
	_CLIENT_DRIVER_TABLE (see page 317)	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
	_HOST_TRANSFER_DATA (see page 318)	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.
	_USB_TPL (see page 320)	Targeted Peripheral List This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.
	CLIENT_DRIVER_TABLE (see page 317)	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
	HOST_TRANSFER_DATA (see page 318)	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.
	USB_TPL (see page 320)	Targeted Peripheral List This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.

Types

	Name	Description
	USB_CLIENT_INIT (see page 321)	This is a typedef to use when defining a client driver initialization handler.
	USB_CLIENT_EVENT_HANDLER (see page 322)	This is a typedef to use when defining a client driver event handler.

Unions

	Name	Description
	TRANSFER_ATTRIBUTES (see page 319)	This is type TRANSFER_ATTRIBUTES.

Description

7.2.1.2.1 CLIENT_DRIVER_TABLE Structure

File

usb_host.h

C

```
typedef struct _CLIENT_DRIVER_TABLE {
  USB_CLIENT_INIT Initialize;
  USB_CLIENT_EVENT_HANDLER EventHandler;
  USB_CLIENT_EVENT_HANDLER DataEventHandler;
  DWORD flags;
} CLIENT_DRIVER_TABLE;
```

Members

Members	Description
USB_CLIENT_INIT Initialize;	Initialization routine
USB_CLIENT_EVENT_HANDLER EventHandler;	Event routine
USB_CLIENT_EVENT_HANDLER DataEventHandler;	Data Event routine
DWORD flags;	Initialization flags

Description

Client Driver Table Structure

This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.

7.2.1.2.2 HOST_TRANSFER_DATA Structure

File

usb_host.h

C

```
typedef struct _HOST_TRANSFER_DATA {
  DWORD dataCount;
  BYTE * pUserData;
  BYTE bEndpointAddress;
  BYTE bErrorCode;
  TRANSFER_ATTRIBUTES bmAttributes;
  BYTE clientDriver;
} HOST_TRANSFER_DATA;
```

Members

Members	Description
DWORD dataCount;	Count of bytes transferred.
BYTE * pUserData;	Pointer to transfer data.
BYTE bEndpointAddress;	Transfer endpoint.
BYTE bErrorCode;	Transfer error code.
TRANSFER_ATTRIBUTES bmAttributes;	INTERNAL USE ONLY - Endpoint transfer attributes.
BYTE clientDriver;	INTERNAL USE ONLY - Client driver index for sending the event.

Description

Host Transfer Information

This structure is used when the event handler is used to notify the upper layer of transfer completion.

7.2.1.2.3 TRANSFER_ATTRIBUTES Union

File

usb_host.h

C

```
typedef union {
  BYTE val;
  struct {
    BYTE bfTransferType : 2;
    BYTE bfSynchronizationType : 2;
    BYTE bfUsageType : 2;
  }
} TRANSFER_ATTRIBUTES;
```

Members

Members	Description
BYTE bfTransferType : 2;	See USB_TRANSFER_TYPE_* for values.
BYTE bfSynchronizationType : 2;	For isochronous endpoints only.
BYTE bfUsageType : 2;	For isochronous endpoints only.

Description

This is type TRANSFER_ATTRIBUTES.

7.2.1.2.4 USB_TPL Structure

File

usb_host.h

C

```
typedef struct _USB_TPL {
  union {
    DWORD val;
    struct {
      WORD idVendor;
      WORD idProduct;
    }
    struct {
      BYTE bClass;
      BYTE bSubClass;
      BYTE bProtocol;
    }
  } device;
  BYTE bConfiguration;
  BYTE ClientDriver;
  union {
    BYTE val;
    struct {
      BYTE bfAllowHNP : 1;
      BYTE bfIsClassDriver : 1;
      BYTE bfSetConfiguration : 1;
    }
  } flags;
} USB_TPL;
```

Members

Members	Description
WORD idVendor;	Vendor ID
WORD idProduct;	Product ID
BYTE bClass;	Class ID
BYTE bSubClass;	SubClass ID
BYTE bProtocol;	Protocol ID
BYTE bConfiguration;	Initial device configuration
BYTE ClientDriver;	Index of client driver in the Client Driver table
BYTE bfAllowHNP : 1;	Is HNP allowed?
BYTE bfIsClassDriver : 1;	Client driver is a class-level driver
BYTE bfSetConfiguration : 1;	bConfiguration is valid

Description

Targeted Peripheral List

This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.

7.2.1.2.5 USB_CLIENT_INIT Type

This is a typedef to use when defining a client driver initialization handler.

File

usb_host.h

C

```
typedef BOOL (* USB_CLIENT_INIT)(BYTE address, DWORD flags, BYTE clientDriverID);
```

Description

This routine is a call out from the host layer to a USB client driver. It is called when the system has been configured as a USB host and a new device has been attached to the bus. Its purpose is to initialize and activate the client driver.

Remarks

There may be multiple client drivers. If so, the USB host layer will call the initialize routine for each of the clients that are in the selected configuration.

Preconditions

The device has been configured.

Parameters

Parameters	Description
BYTE address	Device's address on the bus
DWORD flags	Initialization flags
BYTE clientDriverID	ID to send when issuing a Device Request via USBHostIssueDeviceRequest() or USBHostSetDeviceConfiguration (see page 305)().

Return Values

Return Values	Description
TRUE	Successful
FALSE	Not successful

Function

BOOL (*USB_CLIENT_INIT) (BYTE address, DWORD flags, BYTE clientDriverID)

7.2.1.2.6 USB_CLIENT_EVENT_HANDLER Type

This is a typedef to use when defining a client driver event handler.

File

usb_host.h

C

```
typedef BOOL (* USB_CLIENT_EVENT_HANDLER)(BYTE address, USB_EVENT event, void *data, DWORD
size);
```

Description

This data type defines a pointer to a call-back function that must be implemented by a client driver if it needs to be aware of events on the USB. When an event occurs, the Host layer will call the client driver via this pointer to handle the event. Events are identified by the "event" parameter and may have associated data. If the client driver was able to handle the event, it should return TRUE. If not (or if additional processing is required), it should return FALSE.

Remarks

The application may also implement an event handling routine if it requires knowledge of events. To do so, it must implement a routine that matches this function signature and define the USB_HOST_APP_EVENT_HANDLER (see page 292) macro as the name of that function.

Preconditions

The client must have been initialized.

Parameters

Parameters	Description
BYTE address	Address of device where event occurred
USB_EVENT event	Identifies the event that occured
void *data	Pointer to event-specific data
DWORD size	Size of the event-specific data

Return Values

Return Values	Description
TRUE	The event was handled
FALSE	The event was not handled

Function

BOOL (*USB_CLIENT_EVENT_HANDLER) (BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.1.2.7 USB_NUM_BULK_NAKS Macro

File

usb_host.h

C

```
#define USB_NUM_BULK_NAKS 10000   // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during a bulk transfer before erroring.

7.2.1.2.8 USB_NUM_COMMAND_TRIES Macro

File

usb_host.h

C

```
#define USB_NUM_COMMAND_TRIES 3   // During enumeration, define how many
```

Description

During enumeration, define how many times each command will be tried before giving up and resetting the device.

7.2.1.2.9 USB_NUM_CONTROL_NAKS Macro

File

usb_host.h

C

```
#define USB_NUM_CONTROL_NAKS 20  // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during a control transfer before erroring.

7.2.1.2.10 USB_NUM_ENUMERATION_TRIES Macro

File

usb_host.h

C

```
#define USB_NUM_ENUMERATION_TRIES 3   // Define how many times the host will try
```

Description

Define how many times the host will try to enumerate the device before giving up and setting the state to DETACHED.

7.2.1.2.11 USB_NUM_INTERRUPT_NAKS Macro

File

usb_host.h

C

```
#define USB_NUM_INTERRUPT_NAKS 3   // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during an interrupt OUT transfer before erroring. Interrupt IN transfers that are NAK'd are terminated without error.

7.2.1.2.12 TPL_SET_CONFIG Macro

File

usb_host.h

C

```
#define TPL_SET_CONFIG 0x04                                // Bitmask for setting the configuration.
```

Description

Bitmask for setting the configuration.

7.2.1.2.13 TPL_CLASS_DRV Macro

File

usb_host.h

C

```
#define TPL_CLASS_DRV 0x02                    // Bitmask for class driver support.
```

Description

Bitmask for class driver support.

7.2.1.2.14 TPL_ALLOW_HNP Macro

File

usb_host.h

C

```
#define TPL_ALLOW_HNP 0x01                    // Bitmask for Host Negotiation Protocol.
```

Description

Bitmask for Host Negotiation Protocol.

7.2.1.3 Macros

Macros

	Name	Description
	INIT_CL_SC_P (see page 332)	Set class support in the TPL (non-OTG only).
	INIT_VID_PID (see page 333)	Set VID/PID support in the TPL.

Description

7.2.1.3.1 INIT_CL_SC_P Macro

File

usb_host.h

C

```
#define INIT_CL_SC_P(c,s,p) {((c)|((s)<<8)|((p)<<16))}  // Set class support in the TPL
(non-OTG only).
```

Description

Set class support in the TPL (non-OTG only).

7.2.1.3.2 INIT_VID_PID Macro

File

usb_host.h

C

```
#define INIT_VID_PID(v,p) {((v)|((p)<<16))}           // Set VID/PID support in the TPL.
```

Description

Set VID/PID support in the TPL.

7.2.2 Audio Client Driver

7.2.2.1 Interface Routines

Functions

	Name	Description
	USBHostAudioV1DataEventHandler (see page 335)	This function is the data event handler for this client driver.
	USBHostAudioV1EventHandler (see page 336)	This function is the event handler for this client driver.
	USBHostAudioV1Initialize (see page 337)	This function is the initialization routine for this client driver.
	USBHostAudioV1ReceiveAudioData (see page 338)	This function starts the reception of streaming, isochronous audio data.
	USBHostAudioV1SetInterfaceFullBandwidth (see page 339)	This function sets the full bandwidth interface.
	USBHostAudioV1SetInterfaceZeroBandwidth (see page 340)	This function sets the zero bandwidth interface.
	USBHostAudioV1SetSamplingFrequency (see page 341)	This function sets the sampling frequency for the device.
	USBHostAudioV1SupportedFrequencies (see page 343)	This function returns a pointer to the list of supported frequencies.
	USBHostAudioV1TerminateTransfer (see page 345)	This function terminates an audio stream.

Description

7.2.2.1.1 USBHostAudioV1DataEventHandler Function

This function is the data event handler for this client driver.

File

usb_host_audio_v1.h

C

```
BOOL USBHostAudioV1DataEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is the data event handler for this client driver. It is called by the host layer when isochronous data events occur.

Remarks

The client driver does not need to process the data. Just pass the event up to the application layer.

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
WORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostAudioV1DataEventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.2.1.2 USBHostAudioV1EventHandler Function

This function is the event handler for this client driver.

File

usb_host_audio_v1.h

C

```
BOOL USBHostAudioV1EventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
WORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostAudioV1EventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.2.1.3 USBHostAudioV1Initialize Function

This function is the initialization routine for this client driver.

File

usb_host_audio_v1.h

C

```
BOOL USBHostAudioV1Initialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	ID to send when issuing a Device Request via USBHostIssueDeviceRequest() or USBHostSetDeviceConfiguration (see page 305)().

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostAudioV1Initialize(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.2.1.4 USBHostAudioV1ReceiveAudioData Function

This function starts the reception of streaming, isochronous audio data.

File

usb_host_audio_v1.h

C

```
BYTE USBHostAudioV1ReceiveAudioData(
    BYTE deviceAddress,
    ISOCHRONOUS_DATA * pIsochronousData
);
```

Description

This function starts the reception of streaming, isochronous audio data.

Remarks

Some devices require other operations between setting the full bandwidth interface and starting the streaming audio data. Therefore, these two functions are broken out separately.

Preconditions

USBHostAudioV1SetInterfaceFullBandwidth (see page 339)() must be called to set the device to its full bandwidth interface.

Parameters

Parameters	Description
BYTE deviceAddress	Device address
ISOCHRONOUS_DATA *pIsochronousData	Pointer to an ISOCHRONOUS_DATA structure, containing information for the application and the host driver for the isochronous transfer.

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_AUDIO_DEVICE_NOT_FOUND	No device with specified address
USB_AUDIO_DEVICE_BUSY	Device is already receiving audio data or setting an interface.
Others	See USBHostIssueDeviceRequest() errors.

Function

BYTE USBHostAudioV1ReceiveAudioData(BYTE deviceAddress,

ISOCHRONOUS_DATA *pIsochronousData)

7.2.2.1.5 USBHostAudioV1SetInterfaceFullBandwidth Function

This function sets the full bandwidth interface.

File

usb_host_audio_v1.h

C

```
BYTE USBHostAudioV1SetInterfaceFullBandwidth(
    BYTE deviceAddress
);
```

Description

This function sets the full bandwidth interface. This function should be called before calling USBHostAudioV1ReceiveAudioData (see page 338)() to receive the audio stream. Upon completion, the event EVENT_AUDIO_INTERFACE_SET (see page 350) will be generated.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_AUDIO_DEVICE_NOT_FOUND	No device with specified address
USB_AUDIO_DEVICE_BUSY	Device is already receiving audio data or setting an interface.
Others	See USBHostIssueDeviceRequest() errors.

Function

BYTE USBHostAudioV1SetInterfaceFullBandwidth(BYTE deviceAddress)

7.2.2.1.6 USBHostAudioV1SetInterfaceZeroBandwidth Function

This function sets the zero bandwidth interface.

File

usb_host_audio_v1.h

C

```
BYTE USBHostAudioV1SetInterfaceZeroBandwidth(
    BYTE deviceAddress
);
```

Description

This function sets the full bandwidth interface. This function can be called after calling USBHostAudioV1TerminateTransfer (see page 345)() to terminate the audio stream. Upon completion, the event EVENT_AUDIO_INTERFACE_SET (see page 350) will be generated.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_AUDIO_DEVICE_NOT_FOUND	No device with the specified address.
Others	See USBHostIssueDeviceRequest()

Function

BYTE USBHostAudioV1SetInterfaceZeroBandwidth(BYTE deviceAddress)

7.2.2.1.7 USBHostAudioV1SetSamplingFrequency Function

This function sets the sampling frequency for the device.

File

usb_host_audio_v1.h

C

```
BYTE USBHostAudioV1SetSamplingFrequency(
    BYTE deviceAddress,
    BYTE * frequency
);
```

Description

This function sets the sampling frequency for the device. If the exact frequency is not supported by the device, the device will round it to the closest supported value.

IMPORTANT: If the request is initiated successfully, the frequency value must remain valid until the EVENT_AUDIO_FREQUENCY_SET (see page 349) event is received. Therefore, this value cannot be a local (stack) variable. The application can either use a global variable for this value, or it can use the function USBHostAudioV1SupportedFrequencies (see page 343)() to obtain a pointer to the number and list of supported frequencies, and pass a pointer to the desired frequency in this list.

Remarks

If a global variable is used to old the frequency, it can be declared as a DWORD. Since PIC Microcontrollers are little endian machines, a pointer to the DWORD can be used as the frequency parameter:

```
DWORD   desiredFrequency = 44100;   // Hertz
```

```
rc = USBHostAudioV1SetSamplingFrequency( deviceAddress, (BYTE *)(&desiredFrequency) );
```

Preconditions

None

Example

```
BYTE  numFrequencies;
BYTE  *ptr;

ptr = USBHostAudioV1SupportedFrequencies( deviceAddress );
if (ptr)
{
    numFrequencies = *ptr;
    ptr++;
    if (numFrequencies == 0)
    {
        // Continuous sampling, minimum and maximum are specified.
        DWORD   minFrequency;
        DWORD   maxFrequency;

        minFrequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
        ptr += 3;
        maxFrequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
        if ((minFrequency <= desiredFrequency) && (desiredFrequency <= maxFrequency))
        {
            rc = USBHostAudioV1SetSamplingFrequency( deviceAddress, &desiredFrequency );
        }
        else
        {
            // Desired frequency out of range
        }
    }
    else
    {
        // Discrete sampling frequencies are specified.
        DWORD frequency;
```

```
        while (numFrequencies)
        {
            frequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
            if (frequency == desiredFrequency)
            {
                rc = USBHostAudioV1SetSamplingFrequency( deviceAddress, ptr );
                continue;
            }
            numFrequencies--;
            ptr += 3;
        }
        if (numFrequencies == 0)
        {
            // Desired frequency not found.
        }
    }
}
```

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE *frequency	Pointer to three bytes that specify the desired
sampling frequency. NOTE	If the request is initiated successfully, this location must remain valid until the EVENT_AUDIO_FREQUENCY_SET (see page 349) event is received.

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
Others	See USBHostIssueDeviceRequest() errors.

Function

BYTE USBHostAudioV1SetSamplingFrequency(BYTE deviceAddress, BYTE *frequency)

7.2.2.1.8 USBHostAudioV1SupportedFrequencies Function

This function returns a pointer to the list of supported frequencies.

File

usb_host_audio_v1.h

C

```
BYTE * USBHostAudioV1SupportedFrequencies(
    BYTE deviceAddress
);
```

Returns

This function returns a BYTE pointer to the list of supported frequencies. The first byte of this list is the number of supported frequencies. Each supported frequency is then listed, with three bytes for each frequency.

Description

This function returns a pointer to the list of supported frequencies. It is intended to be used with the function USBHostAudioV1SetSamplingFrequency (see page 341)() to set the device's sampling frequency.

Remarks

None

Preconditions

None

Example

```
BYTE  numFrequencies;
BYTE  *ptr;

ptr = USBHostAudioV1SupportedFrequencies( deviceAddress );
if (ptr)
{
    numFrequencies = *ptr;
    ptr++;
    if (numFrequencies == 0)
    {
        // Continuous sampling, minimum and maximum are specified.
        DWORD   minFrequency;
        DWORD   maxFrequency;

        minFrequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
        ptr += 3;
        maxFrequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
        if ((minFrequency <= desiredFrequency) && (desiredFrequency <= maxFrequency))
        {
            rc = USBHostAudioV1SetSamplingFrequency( deviceAddress, &desiredFrequency );
        }
        else
        {
            // Desired frequency out of range
        }
    }
    else
    {
        // Discrete sampling frequencies are specified.
        DWORD frequency;

        while (numFrequencies)
        {
            frequency = *ptr + (*(ptr+1) << 8) + (*(ptr+2) << 16);
            if (frequency == desiredFrequency)
            {
                rc = USBHostAudioV1SetSamplingFrequency( deviceAddress, ptr );
                continue;
```

```
                }
                numFrequencies--;
                ptr += 3;
            }
            if (numFrequencies == 0)
            {
                // Desired frequency not found.
            }
        }
    }
```

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Function

BYTE * USBHostAudioV1SupportedFrequencies(BYTE deviceAddress)

7.2.2.1.9 USBHostAudioV1TerminateTransfer Function

This function terminates an audio stream.

File

usb_host_audio_v1.h

C

```
void USBHostAudioV1TerminateTransfer(
    BYTE deviceAddress
);
```

Returns

None

Description

This function terminates an audio stream. It does not change the device's selected interface. The application may wish to call USBHostAudioV1SetInterfaceZeroBandwidth (see page 340)() after this function to set the device to the zero bandwidth interface.

Between terminating one audio stream and starting another, the application should call USBHostIsochronousBuffersReset() to reset the data buffers. This is done from the application layer rather than from this function, so the application can process all received audio data.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Function

void USBHostAudioV1TerminateTransfer(BYTE deviceAddress)

7.2.2.2 Data Types and Constants

Macros

	Name	Description
	EVENT_AUDIO_ATTACH (see page 347)	An audio device has attached. The returned data pointer points to a USB_AUDIO_V1_DEVICE_ID structure.
	EVENT_AUDIO_DETACH (see page 348)	An audio device has detached. The returned data pointer points to a byte with the previous address of the detached device.
	EVENT_AUDIO_FREQUENCY_SET (see page 349)	This event is returned after the sampling frequency is set via USBHostAudioV1SetSamplingFrequency (see page 341)(). The returned data pointer points to a HOST_TRANSFER_DATA (see page 318) structure, with the error code for this request.
	EVENT_AUDIO_INTERFACE_SET (see page 350)	This event is returned after the full or zero bandwidth interface has been set. The returned data pointer is NULL, but the size is the error code from the transfer.
	EVENT_AUDIO_NONE (see page 351)	No event occured (NULL event).
	EVENT_AUDIO_OFFSET (see page 352)	If the application has not defined an offset for audio events, set it to 0.
	EVENT_AUDIO_STREAM_RECEIVED (see page 353)	An audio stream data packet has been received. The returned data pointer points to a HOST_TRANSFER_DATA (see page 318) structure, with information about the most recent transfer. One event will be returned for each transfer, so the application will know how much data was actually received in each transfer. If there was a bus error, both the returned data pointer and the size will be zero.

Description

7.2.2.2.1 EVENT_AUDIO_ATTACH Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_ATTACH EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 1
```

Description

An audio device has attached. The returned data pointer points to a USB_AUDIO_V1_DEVICE_ID structure.

7.2.2.2.2 EVENT_AUDIO_DETACH Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_DETACH EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 2
```

Description

An audio device has detached. The returned data pointer points to a byte with the previous address of the detached device.

7.2.2.2.3 EVENT_AUDIO_FREQUENCY_SET Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_FREQUENCY_SET EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 4
```

Description

This event is returned after the sampling frequency is set via USBHostAudioV1SetSamplingFrequency (see page 341)(). The returned data pointer points to a HOST_TRANSFER_DATA (see page 318) structure, with the error code for this request.

7.2.2.2.4 EVENT_AUDIO_INTERFACE_SET Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_INTERFACE_SET EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 5
```

Description

This event is returned after the full or zero bandwidth interface has been set. The returned data pointer is NULL, but the size is the error code from the transfer.

7.2.2.2.5 EVENT_AUDIO_NONE Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_NONE EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 0
```

Description

No event occured (NULL event).

7.2.2.2.6 EVENT_AUDIO_OFFSET Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_OFFSET 0
```

Description

If the application has not defined an offset for audio events, set it to 0.

7.2.2.2.7 EVENT_AUDIO_STREAM_RECEIVED Macro

File

usb_host_audio_v1.h

C

```
#define EVENT_AUDIO_STREAM_RECEIVED EVENT_AUDIO_BASE + EVENT_AUDIO_OFFSET + 3
```

Description

An audio stream data packet has been received. The returned data pointer points to a HOST_TRANSFER_DATA (see page 318) structure, with information about the most recent transfer. One event will be returned for each transfer, so the application will know how much data was actually received in each transfer. If there was a bus error, both the returned data pointer and the size will be zero.

7.2.3 Audio MIDI Client Driver

7.2.3.1 Interface Functions

Functions

	Name	Description
	USBHostMIDIRead (see page 358)	This function will attempt to read length number of bytes from the attached MIDI device located at handle, and will save the contents to ram located at buffer.
	USBHostMIDITransferIsComplete (see page 361)	This routine indicates whether or not the last transfer over endpointIndex is complete.
	USBHostMIDIWrite (see page 362)	This function will attempt to write length number of bytes from memory at location buffer to the attached MIDI device located at handle.

Macros

	Name	Description
	USBHostMIDIDeviceDetached (see page 355)	This interface is used to check if the device has been detached from the bus.
	USBHostMIDIEndpointDirection (see page 356)	This function retrieves the endpoint direction of the endpoint at endpointIndex for device that's located at handle.
	USBHostMIDINumberOfEndpoints (see page 357)	This function retrieves the number of endpoints for the device that's located at handle.
	USBHostMIDISizeOfEndpoint (see page 359)	This function retrieves the endpoint size of the endpoint at endpointIndex for device that's located at handle.
	USBHostMIDITransferIsBusy (see page 360)	This interface is used to check if the client driver is currently busy transferring data over endponitIndex for the device at handle.

Description

7.2.3.1.1 USBHostMIDIDeviceDetached Macro

File

usb_host_midi.h

C

```
#define USBHostMIDIDeviceDetached(a) ( (((a)==NULL) ? FALSE : TRUE )
```

Description

This interface is used to check if the device has been detached from the bus.

Remarks

None

Preconditions

None

Example

```
if (USBHostMIDIDeviceDetached( deviceAddress ))
{
    // Handle detach
}
```

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info

Return Values

Return Values	Description
TRUE	The device has been detached, or an invalid handle is given.
FALSE	The device is attached

Function

BOOL USBHostMIDIDeviceDetached(void* handle)

7.2.3.1.2 USBHostMIDIEndpointDirection Macro

File

usb_host_midi.h

C

```
#define USBHostMIDIEndpointDirection(a,b) (((MIDI_DEVICE*)a)->endpoints[b].endpointAddress
& 0x80) ? IN : OUT
```

Returns

MIDI_ENDPOINT_DIRECTION - Returns the direction of the endpoint (IN or OUT)

Description

This function retrieves the endpoint direction of the endpoint at endpointIndex for device that's located at handle.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info
BYTE endpointIndex	the index of the endpoint whose direction is requested

Function

MIDI_ENDPOINT_DIRECTION USBHostMIDIEndpointDirection(void* handle, BYTE endpointIndex)

7.2.3.1.3 USBHostMIDINumberOfEndpoints Macro

File

usb_host_midi.h

C

```
#define USBHostMIDINumberOfEndpoints(a) ((MIDI_DEVICE*)a)->numEndpoints
```

Returns

BYTE - Returns the number of endpoints for the device at handle.

Description

This function retrieves the number of endpoints for the device that's located at handle.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info

Function

BYTE USBHostMIDINumberOfEndpoints(void* handle)

7.2.3.1.4 USBHostMIDIRead Function

File

usb_host_midi.h

C

```
BYTE USBHostMIDIRead(
    void* handle,
    BYTE endpointIndex,
    void * buffer,
    WORD length
);
```

Description

This function will attempt to read length number of bytes from the attached MIDI device located at handle, and will save the contents to ram located at buffer.

Remarks

None

Preconditions

The device must be connected and enumerated. The array at *buffer should have at least length number of bytes available.

Example

```
if (!USBHostMIDITransferIsBusy( deviceHandle, currentEndpoint )
{
    USBHostMIDIRead( deviceHandle, currentEndpoint, &buffer, sizeof(buffer));
}
```

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info
BYTE endpointIndex	the index of the endpoint whose direction is requested
void* buffer	Pointer to the data buffer
WORD length	Number of bytes to be read

Return Values

Return Values	Description
USB_SUCCESS	The Read was started successfully
(USB error code)	The Read was not started. See USBHostRead (see page 301)() for a list of errors.

Function

BYTE USBHostMIDIRead(void* handle, BYTE endpointIndex, void *buffer, WORD length)

7.2.3.1.5 USBHostMIDISizeOfEndpoint Macro

File

usb_host_midi.h

C

```
#define USBHostMIDISizeOfEndpoint(a,b) ((MIDI_DEVICE*)a)->endpoints[b].endpointSize
```

Returns

DWORD - Returns the number of bytes for the endpoint (4 - 64 bytes per USB spec)

Description

This function retrieves the endpoint size of the endpoint at endpointIndex for device that's located at handle.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info
BYTE endpointIndex	the index of the endpoint whose direction is requested

Function

DWORD USBHostMIDISizeOfEndpoint(void* handle, BYTE endpointIndex)

7.2.3.1.6 USBHostMIDITransferIsBusy Macro

This interface is used to check if the client driver is currently busy transferring data over endponitIndex for the device at handle.

File

usb_host_midi.h

C

```
#define USBHostMIDITransferIsBusy(a,b) ((MIDI_DEVICE*)a)->endpoints[b].busy
```

Description

This interface is used to check if the client driver is currently busy receiving or sending data from the device at the endpoint with number endpointIndex. This function is intended for use with transfer events. With polling, the function USBHostMIDITransferIsComplete (see page 361)() should be used.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostMIDITransferIsBusy( handle, endpointIndex ))
{
    USBHostMIDIRead( handle, endpointIndex, &buffer, sizeof( buffer ) );
}
```

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info
BYTE endpointIndex	the index of the endpoint whose direction is requested

Return Values

Return Values	Description
TRUE	The device is receiving data or an invalid handle is given.
FALSE	The device is not receiving data

Function

BOOL USBHostMIDITransferIsBusy(void* handle, BYTE endpointIndex)

7.2.3.1.7 USBHostMIDITransferIsComplete Function

This routine indicates whether or not the last transfer over endpointIndex is complete.

File

usb_host_midi.h

C

```
BOOL USBHostMIDITransferIsComplete(
    void* handle,
    BYTE endpointIndex,
    BYTE * errorCode,
    DWORD * byteCount
);
```

Description

This routine indicates whether or not the last transfer over endpointIndex is complete. If it is, then the returned errorCode and byteCount are valid, and reflect the error code and the number of bytes received.

This function is intended for use with polling. With transfer events, the function USBHostMIDITransferIsBusy (see page 360)() should be used.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
void* handle	Pointer to a structure containing the Device Info
BYTE endpointIndex	index of endpoint in endpoints array
BYTE *errorCode	Error code of the last transfer, if complete
DWORD *byteCount	Bytes transferred during the last transfer, if complete

Return Values

Return Values	Description
TRUE	The IN transfer is complete. errorCode and byteCount are valid.
FALSE	The IN transfer is not complete. errorCode and byteCount are invalid.

Function

BOOL USBHostMIDITransferIsComplete(void* handle, BYTE endpointIndex,

BYTE *errorCode, DWORD *byteCount);

7.2.3.1.8 USBHostMIDIWrite Function

File

usb_host_midi.h

C

```
BYTE USBHostMIDIWrite(
    void* handle,
    BYTE endpointIndex,
    void * buffer,
    WORD length
);
```

Description

This function will attempt to write length number of bytes from memory at location buffer to the attached MIDI device located at handle.

Remarks

None

Preconditions

The device must be connected and enumerated. The array at *buffer should have at least length number of bytes available.

Example

```
if (!USBHostMIDITransferIsBusy( deviceHandle, currentEndpoint )
{
    USBHostMIDIWrite( deviceAddress, &buffer, sizeof(buffer) );
}
```

Parameters

Parameters	Description
handle	Pointer to a structure containing the Device Info
endpointIndex	Index of the endpoint
buffer	Pointer to the data being transferred
length	Size of the data being transferred

Return Values

Return Values	Description
USB_SUCCESS	The Write was started successfully
(USB error code)	The Write was not started. See USBHostWrite (see page 313)() for a list of errors.

Function

BYTE USBHostMIDIWrite(void* handle, BYTE endpointIndex, void *buffer, WORD length)

7.2.3.2 Data Types and Constants

Macros

	Name	Description
	EVENT_MIDI_ATTACH (see page 364)	This event indicates that a MIDI device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a MIDI_DEVICE_ID structure, and size is the size of the MIDI_DEVICE_ID structure.
	EVENT_MIDI_DETACH (see page 365)	This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.
	EVENT_MIDI_OFFSET (see page 366)	This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the MIDI events to resolve conflicts in event number.
	EVENT_MIDI_TRANSFER_DONE (see page 367)	This event indicates that a previous write/read request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.

Description

7.2.3.2.1 EVENT_MIDI_ATTACH Macro

File

usb_host_midi.h

C

```
#define EVENT_MIDI_ATTACH (EVENT_AUDIO_BASE+EVENT_MIDI_OFFSET+0)
```

Description

This event indicates that a MIDI device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a MIDI_DEVICE_ID structure, and size is the size of the MIDI_DEVICE_ID structure.

7.2.3.2.2 EVENT_MIDI_DETACH Macro

File

usb_host_midi.h

C

```
#define EVENT_MIDI_DETACH (EVENT_AUDIO_BASE+EVENT_MIDI_OFFSET+1)
```

Description

This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.

7.2.3.2.3 EVENT_MIDI_OFFSET Macro

File

usb_host_midi.h

C

```
#define EVENT_MIDI_OFFSET 0
```

Description

This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the MIDI events to resolve conflicts in event number.

7.2.3.2.4 EVENT_MIDI_TRANSFER_DONE Macro

File

usb_host_midi.h

C

```
#define EVENT_MIDI_TRANSFER_DONE (EVENT_AUDIO_BASE+EVENT_MIDI_OFFSET+2)
```

Description

This event indicates that a previous write/read request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.

7.2.4 Android Accessory Client Driver

7.2.4.1 Interface Routines

Functions

	Name	Description
	AndroidAppIsReadComplete (see page 369)	Check to see if the last read to the Android device was completed
	AndroidAppIsWriteComplete (see page 370)	Check to see if the last write to the Android device was completed
	AndroidAppRead (see page 371)	Attempts to read information from the specified Android device
	AndroidAppStart (see page 372)	Sets the accessory information and initializes the client driver information after the initial power cycles.
	AndroidAppWrite (see page 373)	Sends data to the Android device specified by the passed in handle.
	AndroidTasks (see page 374)	Tasks function that keeps the Android client driver moving

Description

7.2.4.1.1 AndroidAppIsReadComplete Function

Check to see if the last read to the Android device was completed

File

usb_host_android.h

C

```
BOOL AndroidAppIsReadComplete(
    void* handle,
    BYTE* errorCode,
    DWORD* size
);
```

Description

Check to see if the last read to the Android device was completed. If complete, returns the amount of data that was sent and the corresponding error code for the transmission.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful
- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

Transfer has previously been requested from an Android device.

Parameters

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 379) event
BYTE* errorCode	a pointer to the location where the resulting error code should be written
DWORD* size	a pointer to the location where the resulting size information should be written

Return Values

Return Values	Description
TRUE	Transfer is complete.
FALSE	Transfer is not complete.

Function

BOOL AndroidAppIsReadComplete(void* handle, BYTE* errorCode, DWORD* size)

7.2.4.1.2 AndroidAppIsWriteComplete Function

Check to see if the last write to the Android device was completed

File

usb_host_android.h

C

```
BOOL AndroidAppIsWriteComplete(
    void* handle,
    BYTE* errorCode,
    DWORD* size
);
```

Description

Check to see if the last write to the Android device was completed. If complete, returns the amount of data that was sent and the corresponding error code for the transmission.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful
- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

Transfer has previously been sent to Android device.

Parameters

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 379) event
BYTE* errorCode	a pointer to the location where the resulting error code should be written
DWORD* size	a pointer to the location where the resulting size information should be written

Return Values

Return Values	Description
TRUE	Transfer is complete.
FALSE	Transfer is not complete.

Function

BOOL AndroidAppIsWriteComplete(void* handle, BYTE* errorCode, DWORD* size)

7.2.4.1.3 AndroidAppRead Function

Attempts to read information from the specified Android device

File

usb_host_android.h

C

```
BYTE AndroidAppRead(
    void* handle,
    BYTE* data,
    DWORD size
);
```

Description

Attempts to read information from the specified Android device. This function does not block. Data availability is checked via the AndroidAppIsReadComplete (see page 369)() function.

Remarks

None

Preconditions

A read request is not already in progress and an Android device is attached.

Parameters

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 379) event
BYTE* data	a pointer to the location of where the data should be stored. This location should be accessible by the USB module
DWORD size	the amount of data to read.

Return Values

Return Values	Description
USB_SUCCESS	Read started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlRead to read from a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must read from an IN endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Read is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.
USB_ERROR_BUFFER_TOO_SMALL (see page 382)	The buffer passed to the read function was smaller than the endpoint size being used (buffer must be larger than or equal to the endpoint size).

Function

BYTE AndroidAppRead(void* handle, BYTE* data, DWORD size)

7.2.4.1.4 AndroidAppStart Function

Sets the accessory information and initializes the client driver information after the initial power cycles.

File

usb_host_android.h

C

```
void AndroidAppStart(
    ANDROID_ACCESSORY_INFORMATION* accessoryInfo
);
```

Description

Sets the accessory information and initializes the client driver information after the initial power cycles. Since this resets all device information this function should be used only after a compete system reset. This should not be called while the USB is active or while connected to a device.

Remarks

None

Preconditions

USB module should not be in operation

Parameters

Parameters	Description
ANDROID_ACCESSORY_INFORMATION *info	the information about the Android accessory

Function

void AndroidAppStart(ANDROID_ACCESSORY_INFORMATION (see page 376) *info)

7.2.4.1.5 AndroidAppWrite Function

Sends data to the Android device specified by the passed in handle.

File

usb_host_android.h

C

```
BYTE AndroidAppWrite(
    void* handle,
    BYTE* data,
    DWORD size
);
```

Description

Sends data to the Android device specified by the passed in handle.

Remarks

None

Preconditions

Transfer is not already in progress. USB module is initialized and Android device has attached.

Parameters

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 379) event
BYTE* data	the data to send to the Android device
DWORD size	the size of the data that needs to be sent

Return Values

Return Values	Description
USB_SUCCESS	Write started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlWrite to write to a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must write to an OUT endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Write is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

BYTE AndroidAppWrite(void* handle, BYTE* data, DWORD size)

7.2.4.1.6 AndroidTasks Function

Tasks function that keeps the Android client driver moving

File

usb_host_android.h

C

```
void AndroidTasks();
```

Description

Tasks function that keeps the Android client driver moving. Keeps the driver processing requests and handling events. This function should be called periodically (the same frequency as USBHostTasks() would be helpful).

Remarks

This function should be called periodically to keep the Android driver moving.

Preconditions

AndroidAppStart (see page 372)() function has been called before the first calling of this function

Function

void AndroidTasks(void)

7.2.4.2 Data Type and Constants

Structures

	Name	Description
	ANDROID_ACCESSORY_INFORMATION (see page 376)	This structure contains the informatin that is required to successfully create a link between the Android device and the accessory. This information must match the information entered in the accessory filter in the Android application in order for the Android application to access the device. An instance of this structure should be passed into the AndroidAppStart (see page 372)() at initialization.

Description

7.2.4.2.1 ANDROID_ACCESSORY_INFORMATION Structure

File

usb_host_android.h

C

```
typedef struct {
  char* manufacturer;
  BYTE manufacturer_size;
  char* model;
  BYTE model_size;
  char* description;
  BYTE description_size;
  char* version;
  BYTE version_size;
  char* URI;
  BYTE URI_size;
  char* serial;
  BYTE serial_size;
} ANDROID_ACCESSORY_INFORMATION;
```

Members

Members	Description
char* manufacturer;	String: manufacturer name
BYTE manufacturer_size;	length of manufacturer string
char* model;	String: model name
BYTE model_size;	length of model name string
char* description;	String: description of the accessory
BYTE description_size;	length of the description string
char* version;	String: version number
BYTE version_size;	length of the version number string
char* URI;	String: URI for the accessory (most commonly a URL)
BYTE URI_size;	length of the URI string
char* serial;	String: serial number of the device
BYTE serial_size;	length of the serial number string

Description

This structure contains the informatin that is required to successfully create a link between the Android device and the accessory. This information must match the information entered in the accessory filter in the Android application in order for the Android application to access the device. An instance of this structure should be passed into the AndroidAppStart (see page 372)() at initialization.

7.2.4.3 Macros

Macros

	Name	Description
	ANDROID_BASE_OFFSET (see page 378)	Defines the event offset for the Android specific events. If not defined, then a default of 0 is used.
	EVENT_ANDROID_ATTACH (see page 379)	This event is thrown when an Android device is attached and successfully entered into accessory mode already. The data portion of this event is the handle that is required to communicate to the device and should be saved so that it can be passed to all of the transfer functions. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.
	EVENT_ANDROID_DETACH (see page 380)	This event is thrown when an Android device is removed. The data portion of the event is the handle of the device that has been removed. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.
	NUM_ANDROID_DEVICES_SUPPORTED (see page 381)	Defines the number of concurrent Android devices this implementation is allowed to talk to. This definition is only used for implementations where the accessory is the host and the Android device is the slave. This is also most often defined to be 1. If this is not defined by the user, a default of 1 is used. This option is only used when compiling the source version of the library. This value is set to 1 for pre-compiled versions of the library.
	USB_ERROR_BUFFER_TOO_SMALL (see page 382)	Error code indicating that the buffer passed to the read function was too small. Since the USB host can't control how much data it will receive in a single packet, the user must provide a buffer that is at least the size of the endpoint of the attached device. If a buffer is passed in that is too small, the read will not start and this error is returned to the user.
	ANDROID_INIT_FLAG_BYPASS_PROTOCOL (see page 383)	This defintion is used in the usbClientDrvTable[] in the flags field in order to bypass the Android accessory initialization phase. This should be used only when a device is known to already be in accessory mode (in protocol v1 if the VID/PID are already matching the accessory mode VID/PID). In some cases an Android device doesn't exit accessory mode and thus those other protocol commands will not work. This flag must be used to save those devices

Description

7.2.4.3.1 ANDROID_BASE_OFFSET Macro

File

usb_host_android.h

C

```
#define ANDROID_BASE_OFFSET 0
```

Description

Defines the event offset for the Android specific events. If not defined, then a default of 0 is used.

7.2.4.3.2 EVENT_ANDROID_ATTACH Macro

File

usb_host_android.h

C

```
#define EVENT_ANDROID_ATTACH ANDROID_EVENT_BASE + 0
```

Description

This event is thrown when an Android device is attached and successfully entered into accessory mode already. The data portion of this event is the handle that is required to communicate to the device and should be saved so that it can be passed to all of the transfer functions. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.

7.2.4.3.3 EVENT_ANDROID_DETACH Macro

File

usb_host_android.h

C

```
#define EVENT_ANDROID_DETACH ANDROID_EVENT_BASE + 1
```

Description

This event is thrown when an Android device is removed. The data portion of the event is the handle of the device that has been removed. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.

7.2.4.3.4 NUM_ANDROID_DEVICES_SUPPORTED Macro

File

usb_host_android.h

C

```
#define NUM_ANDROID_DEVICES_SUPPORTED 1
```

Description

Defines the number of concurrent Android devices this implementation is allowed to talk to. This definition is only used for implementations where the accessory is the host and the Android device is the slave. This is also most often defined to be 1. If this is not defined by the user, a default of 1 is used.

This option is only used when compiling the source version of the library. This value is set to 1 for pre-compiled versions of the library.

7.2.4.3.5 USB_ERROR_BUFFER_TOO_SMALL Macro

File

usb_host_android.h

C

```
#define USB_ERROR_BUFFER_TOO_SMALL USB_ERROR_CLASS_DEFINED + 0
```

Description

Error code indicating that the buffer passed to the read function was too small. Since the USB host can't control how much data it will receive in a single packet, the user must provide a buffer that is at least the size of the endpoint of the attached device. If a buffer is passed in that is too small, the read will not start and this error is returned to the user.

7.2.4.3.6 ANDROID_INIT_FLAG_BYPASS_PROTOCOL Macro

File

usb_host_android.h

C

```
#define ANDROID_INIT_FLAG_BYPASS_PROTOCOL 0x00000001
```

Description

This defintion is used in the usbClientDrvTable[] in the flags field in order to bypass the Android accessory initialization phase. This should be used only when a device is known to already be in accessory mode (in protocol v1 if the VID/PID are already matching the accessory mode VID/PID). In some cases an Android device doesn't exit accessory mode and thus those other protocol commands will not work. This flag must be used to save those devices

7.2.4.4 Internal Members

7.2.5 CDC Client Driver

This is a CDC client driver for use with the USB Embedded Host driver.

Description

Communication Device Class (CDC) Host

CDC - Overview

Several type of communication can benefit from USB. Communication Device Class specification provides common specification for communication devices. There are three classes that make up the definition for communications devices:

* Communications Device Class

* Communications Interface Class

* Data Interface Class.

The Communications Device Class is a device-level definition and is used by the host to properly identify a communications device that may present several different types of interfaces.

The Communications Interface Class defines a general-purpose mechanism that can be used to enable all types of communications services on the Universal Serial Bus (USB). This interface consist of two elements, a management element and a notification element. The management element configures and controls the device, it consist of endpoint 0. Notification element is optional and is used to handle transport events. In the current stack notification element is not implemented.

The Data Interface Class defines a general-purpose mechanism to enable bulk or isochronous transfer on the USB when the data does not meet the requirements for any other class. This interface is used to transmit/receive data to/from the device. The type of endpoints belonging to a Data Class interface are restricted to being either isochronous or bulk, and are expected to exist in pairs of the same type (one In and one Out). Current version of the stack is tested for Bulk transfers.

Class-Specific Codes

This section lists the codes for the Communications Device Class, Communications Interface Class and Data Interface Class, including subclasses and protocols supported in the current version of the stack. The current version of the stack supports RS232 emulation over USB. Below is the list of codes to support this functionality.

The following table defines the Communications Device Class code:

Code	Class
0x02	Communications Device Class

Communication Interface Codes

The following table defines the Communications Class code:

Code	Class
0x02	Communications Interface Class

CDC specification mentions various subclass , current version of the Microchip CDC host stack supports below mentioned subclasses. The following table defines the currently supported Subclass codes for the Communications Interface Class:

Code	SubClass
0x02	Abstract Control Model

The following table defines supported Communications Class Protocol Codes:

Code	Protocol
0x01	AT Commands: V.250 etc.

Data Interface Code

The following table defines the Data Interface Class code:

Code	Class
0x0A	Data Interface Class

No specific Subclass and Protocol codes are required to achieve RS232 functionality over USB.

Communication and Data Transfer Handling

Communication Management : The CDC client deriver takes care of enumerating the device connected on the bus. The application must define Line Coding parameters in file usb_config.h . USBConfig utility can be used to set these parameters. If the connected device complies with the setting then the device is successfully attached else the device is not attached onto the bus. If the application needs to change the setting dynamically after the device has been successfully enumerated , interface function USBHostCDC_Api_ACM_Request (see page 387)()can be used to do so. Following standard requests are currently implemented:

Request	Summary
SendEncapsulatedCommand	Issues a command in the format of the supported control protocol.
GetEncapsulatedResponse	Requests a response in the format of the supported control protocol.
SetLineCoding	Configures DTE rate, stop-bits, parity, and number-of-character bits.
GetLineCoding	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
SetControlLineState	[V24] signal used to tell the DCE device the DTE device is now present.

Data transfers : Once the device is attached the application is ready to start data transfers. Usually two endpoints one in each direction are supported by the device.

* To receive data from the device the application must set up a IN request at the rate depending on the baudrate settings. Application can use a timer interrupt to precisely set up the request. Function USBHostCDC_Api_Get_IN_Data (see page 388)()is used to setup the request. Maximum of 64 bytes can be received in single transfer.

* To transmit data to the device application must set up a OUT request. Function USBHostCDC_Api_Send_OUT_Data()is used to setup out request. Any amount of data can be transferred to the device. The Client driver takes care of sending the data in 64 bytes packet.

* USBHostCDC_ApiTransferIsComplete (see page 389)() is used to poll for the status of previous transfer.

* USBHostCDC_ApiDeviceDetect() is used to get the status of the device. If the device is ready for new transfer then the function returns TRUE.

7.2.5.1 Interface Routines

Functions

	Name	Description
	USBHostCDC_Api_ACM_Request (see page 387)	This function can be used by application code to dynamically access ACM specific requests. This function should be used only if apllication intends to modify for example the Baudrate from previouly configured rate. Data transmitted/received to/from device is a array of bytes. Application must take extra care of understanding the data format before using this function.
	USBHostCDC_Api_Get_IN_Data (see page 388)	This function is called by application to receive Input data over DATA interface. This function setsup the request to receive data from the device.
	USBHostCDC_ApiTransferIsComplete (see page 389)	This function is called by application to poll for transfer status. This function returns true in the transfer is over. To check whether the transfer was successfull or not , application must check the error code returned by reference.
	USBHostCDCDeviceStatus (see page 390)	This function determines the status of a CDC device.
	USBHostCDCEventHandler (see page 391)	This function is the event handler for this client driver.
	USBHostCDCInitAddress (see page 392)	This function intializes the address of the attached CDC device.
	USBHostCDCInitialize (see page 393)	This function is the initialization routine for this client driver.
	USBHostCDCResetDevice (see page 395)	This function starts a CDC reset.
	USBHostCDCTransfer (see page 397)	This function starts a CDC transfer.
	USBHostCDCTransferIsComplete (see page 398)	This function indicates whether or not the last transfer is complete.

Macros

	Name	Description
	USBHostCDCRead_DATA (see page 394)	This function intiates a read request from a attached CDC device.
	USBHostCDCSend_DATA (see page 396)	This function intiates a write request to a attached CDC device.

Description

7.2.5.1.1 USBHostCDC_Api_ACM_Request Function

File

usb_host_cdc_interface.h

C

```
BYTE USBHostCDC_Api_ACM_Request(
    BYTE requestType,
    BYTE size,
    BYTE* data
);
```

Description

This function can be used by application code to dynamically access ACM specific requests. This function should be used only if apllication intends to modify for example the Baudrate from previouly configured rate. Data transmitted/received to/from device is a array of bytes. Application must take extra care of understanding the data format before using this function.

Remarks

None

Preconditions

Device must be enumerated and attached successfully.

Parameters

Parameters	Description
BYTE size	Number bytes to be transferred.
BYTE *data	Pointer to data being transferred.

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND (see page 436)	No device with specified address
USB_CDC_DEVICE_BUSY (see page 432)	Device not in proper state for performing a transfer
USB_CDC_COMMAND_FAILED (see page 426)	Request is not supported.
USB_CDC_ILLEGAL_REQUEST (see page 453)	Requested ID is invalid.

Function

BYTE USBHostCDC_Api_ACM_Request(BYTE requestType, BYTE size, BYTE* data)

7.2.5.1.2 USBHostCDC_Api_Get_IN_Data Function

File

usb_host_cdc_interface.h

C

```
BOOL USBHostCDC_Api_Get_IN_Data(
    BYTE no_of_bytes,
    BYTE* data
);
```

Description

This function is called by application to receive Input data over DATA interface. This function setsup the request to receive data from the device.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE no_of_bytes	Number of Bytes expected from the device.
BYTE* data	Pointer to application receive data buffer.

Return Values

Return Values	Description
TRUE	Transfer request is placed successfully.
FALSE	Transfer request failed.

Function

BOOL USBHostCDC_Api_Get_IN_Data(BYTE no_of_bytes, BYTE* data)

7.2.5.1.3 USBHostCDC_ApiTransferIsComplete Function

File

usb_host_cdc_interface.h

C

```
BOOL USBHostCDC_ApiTransferIsComplete(
    BYTE* errorCodeDriver,
    BYTE* byteCount
);
```

Description

This function is called by application to poll for transfer status. This function returns true in the transfer is over. To check whether the transfer was successfull or not , application must check the error code returned by reference.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE *errorCodeDriver	returns.
BYTE *byteCount	Number of bytes transferred.

Return Values

Return Values	Description
TRUE	Transfer is has completed.
FALSE	Transfer is pending.

Function

BOOL USBHostCDC_ApiTransferIsComplete(BYTE* errorCodeDriver,BYTE* byteCount)

7.2.5.1.4 USBHostCDCDeviceStatus Function

This function determines the status of a CDC device.

File

usb_host_cdc.h

C

```
BYTE USBHostCDCDeviceStatus(
    BYTE deviceAddress
);
```

Description

This function determines the status of a CDC device.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	address of device to query

Return Values

Return Values	Description
USB_CDC_DEVICE_NOT_FOUND (see page 436)	Illegal device address, or the device is not an CDC
USB_CDC_INITIALIZING (see page 454)	CDC is attached and in the process of initializing
USB_PROCESSING_REPORT_DESCRIPTOR (see page 579)	CDC device is detected and report descriptor is being parsed
USB_CDC_NORMAL_RUNNING (see page 461)	CDC Device is running normal, ready to send and receive reports
USB_CDC_DEVICE_HOLDING (see page 434)	Device is holding due to error
USB_CDC_DEVICE_DETACHED (see page 433)	CDC detached.

Function

BYTE   USBHostCDCDeviceStatus(BYTE deviceAddress)

7.2.5.1.5 USBHostCDCEventHandler Function

This function is the event handler for this client driver.

File

usb_host_cdc.h

C

```
BOOL USBHostCDCEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
DWORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostCDCEventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.5.1.6 USBHostCDCInitAddress Function

This function intializes the address of the attached CDC device.

File

usb_host_cdc.h

C

```
BOOL USBHostCDCInitAddress(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function intializes the address of the attached CDC device. Once the device is enumerated without any errors, the CDC client call this function. For all the transfer requesets this address is used to indentify the CDC device.

Remarks

None

Preconditions

The device has been enumerated without any errors.

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	Client driver identification for device requests

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostCDCInitAddress(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.5.1.7 USBHostCDCInitialize Function

This function is the initialization routine for this client driver.

File

usb_host_cdc.h

C

```
BOOL USBHostCDCInitialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated.For a CDC device we need to look into CDC descriptor, interface descriptor and endpoint descriptor.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	Client driver identification for device requests

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostCDCInitialize(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.5.1.8 USBHostCDCRead_DATA Macro

This function intiates a read request from a attached CDC device.

File

usb_host_cdc.h

C

```
#define USBHostCDCRead_DATA( address,interface,size,data,endpointData) \
         USBHostCDCTransfer( address,0,1,interface, size,data,endpointData)
```

Description

This function starts a CDC read transfer.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
address	Device address
interface	interface number of the requested transfer
size	Number of bytes to be read from the device
data	address of location where received data is to be stored
endpointDATA	endpoint details on which the transfer is requested

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND (see page 436)	No device with specified address
USB_CDC_DEVICE_BUSY (see page 432)	Device not in proper state for performing a transfer

Function

USBHostCDCRead_DATA(address,interface,size,data,endpointData)

7.2.5.1.9 USBHostCDCResetDevice Function

This function starts a CDC reset.

File

usb_host_cdc.h

C

```
BYTE USBHostCDCResetDevice(
    BYTE deviceAddress
);
```

Description

This function starts a CDC reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Reset started
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_ILLEGAL_REQUEST (see page 621)	Device is in an illegal state for reset

Function

BYTE USBHostCDCResetDevice(BYTE deviceAddress)

7.2.5.1.10 USBHostCDCSend_DATA Macro

This function intiates a write request to a attached CDC device.

File

usb_host_cdc.h

C

```
#define USBHostCDCSend_DATA( address,interface,size,data,endpointData) \
         USBHostCDCTransfer( address,0,0,interface, size,data,endpointData)
```

Description

This function starts a CDC write transfer.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
address	Device address
interface	interface number of the requested transfer
size	Number of bytes to be transfered to the device
data	address of location where the data to be transferred is stored
endpointDATA	endpoint details on which the transfer is requested

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND (see page 436)	No device with specified address
USB_CDC_DEVICE_BUSY (see page 432)	Device not in proper state for performing a transfer

Function

USBHostCDCSend_DATA(address,interface,size,data,endpointData)

7.2.5.1.11 USBHostCDCTransfer Function

This function starts a CDC transfer.

File

usb_host_cdc.h

C

```
BYTE USBHostCDCTransfer(
    BYTE deviceAddress,
    BYTE request,
    BYTE direction,
    BYTE interfaceNum,
    WORD size,
    BYTE * data,
    BYTE endpointDATA
);
```

Description

This function starts a CDC transfer. A read/write wrapper is provided in application interface file to access this function.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE request	Request type for Communication Interface
BYTE direction	1=read, 0=write
BYTE interfaceNum	interface number of the requested transfer
BYTE size	Byte size of the data buffer
BYTE *data	Pointer to the data buffer
BYTE endpointDATA	endpoint details on which the transfer is requested

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND (see page 436)	No device with specified address
USB_CDC_DEVICE_BUSY (see page 432)	Device not in proper state for performing a transfer

Function

USBHostCDCTransfer(BYTE deviceAddress, BYTE direction, BYTE reportid, BYTE size, BYTE *data)

7.2.5.1.12 USBHostCDCTransferIsComplete Function

This function indicates whether or not the last transfer is complete.

File

usb_host_cdc.h

C

```
BOOL USBHostCDCTransferIsComplete(
    BYTE deviceAddress,
    BYTE * errorCode,
    BYTE * byteCount
);
```

Description

This function indicates whether or not the last transfer is complete. If the functions returns TRUE, the returned byte count and error code are valid. Since only one transfer can be performed at once and only one endpoint can be used, we only need to know the device address.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE *errorCode	Error code from last transfer
DWORD *byteCount	Number of bytes transferred

Return Values

Return Values	Description
TRUE	Transfer is complete, errorCode is valid
FALSE	Transfer is not complete, errorCode is not valid

Function

BOOL USBHostCDCTransferIsComplete(BYTE deviceAddress,

BYTE *errorCode, DWORD *byteCount)

7.2.5.2 Data Types and Constants

Macros

	Name	Description
	DEVICE_CLASS_CDC (see page 412)	CDC Interface Class Code
	EVENT_CDC_COMM_READ_DONE (see page 413)	A CDC Communication Read transfer has completed
	EVENT_CDC_COMM_WRITE_DONE (see page 414)	A CDC Communication Write transfer has completed
	EVENT_CDC_DATA_READ_DONE (see page 415)	A CDC Data Read transfer has completed
	EVENT_CDC_DATA_WRITE_DONE (see page 416)	A CDC Data Write transfer has completed
	EVENT_CDC_NAK_TIMEOUT (see page 417)	CDC device NAK timeout has occurred
	EVENT_CDC_NONE (see page 418)	No event occured (NULL event)
	EVENT_CDC_OFFSET (see page 419)	If the application has not defined an offset for CDC events, set it to 0.
	EVENT_CDC_RESET (see page 420)	CDC reset complete
	USB_CDC_ABSTRACT_CONTROL_MODEL (see page 421)	Abstract Control Model
	USB_CDC_ATM_NETWORKING_CONTROL_MODEL (see page 422)	ATM Networking Control Model
	USB_CDC_CAPI_CONTROL_MODEL (see page 423)	CAPI Control Model
	USB_CDC_CLASS_ERROR (see page 424)	CDC Class Error Codes
	USB_CDC_COMM_INTF (see page 425)	Communication Interface Class Code
	USB_CDC_COMMAND_FAILED (see page 426)	Command failed at the device.
	USB_CDC_COMMAND_PASSED (see page 427)	Command was successful.
	USB_CDC_CONTROL_LINE_LENGTH (see page 428)	Number of bytes Control line transfer
	USB_CDC_CS_ENDPOINT (see page 429)	This is macro USB_CDC_CS_ENDPOINT.
	USB_CDC_CS_INTERFACE (see page 430)	Functional Descriptor Details Type Values for the bDscType Field
	USB_CDC_DATA_INTF (see page 431)	Data Interface Class Codes
	USB_CDC_DEVICE_BUSY (see page 432)	A transfer is currently in progress.
	USB_CDC_DEVICE_DETACHED (see page 433)	Device is detached.
	USB_CDC_DEVICE_HOLDING (see page 434)	Device is holding due to error
	USB_CDC_DEVICE_MANAGEMENT (see page 435)	Device Management
	USB_CDC_DEVICE_NOT_FOUND (see page 436)	Device with the specified address is not available.
	USB_CDC_DIRECT_LINE_CONTROL_MODEL (see page 437)	Direct Line Control Model
	USB_CDC_DSC_FN_ACM (see page 438)	ACM - Abstract Control Management
	USB_CDC_DSC_FN_CALL_MGT (see page 439)	This is macro USB_CDC_DSC_FN_CALL_MGT.
	USB_CDC_DSC_FN_COUNTRY_SELECTION (see page 440)	This is macro USB_CDC_DSC_FN_COUNTRY_SELECTIO N.
	USB_CDC_DSC_FN_DLM (see page 441)	DLM - Direct Line Managment
	USB_CDC_DSC_FN_HEADER (see page 442)	bDscSubType in Functional Descriptors
	USB_CDC_DSC_FN_RPT_CAPABILITIES (see page 443)	This is macro USB_CDC_DSC_FN_RPT_CAPABILITIES.
	USB_CDC_DSC_FN_TEL_OP_MODES (see page 444)	This is macro USB_CDC_DSC_FN_TEL_OP_MODES.

	USB_CDC_DSC_FN_TELEPHONE_RINGER (see page 445)	This is macro USB_CDC_DSC_FN_TELEPHONE_RINGER.
	USB_CDC_DSC_FN_UNION (see page 446)	This is macro USB_CDC_DSC_FN_UNION.
	USB_CDC_DSC_FN_USB_TERMINAL (see page 447)	This is macro USB_CDC_DSC_FN_USB_TERMINAL.
	USB_CDC_ETHERNET_EMULATION_MODEL (see page 448)	Ethernet Emulation Model
	USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL (see page 449)	Ethernet Networking Control Model
	USB_CDC_GET_COMM_FEATURE (see page 450)	Returns the current settings for the communications feature.
	USB_CDC_GET_ENCAPSULATED_REQUEST (see page 451)	Requests a response in the format of the supported control protocol.
	USB_CDC_GET_LINE_CODING (see page 452)	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
	USB_CDC_ILLEGAL_REQUEST (see page 453)	Cannot perform requested operation.
	USB_CDC_INITIALIZING (see page 454)	Device is initializing.
	USB_CDC_INTERFACE_ERROR (see page 455)	The interface layer cannot support the device.
	USB_CDC_LINE_CODING_LENGTH (see page 456)	Number of bytes Line Coding transfer
	USB_CDC_MOBILE_DIRECT_LINE_MODEL (see page 457)	Mobile Direct Line Model
	USB_CDC_MULTI_CHANNEL_CONTROL_MODEL (see page 458)	Multi-Channel Control Model
	USB_CDC_NO_PROTOCOL (see page 459)	No class specific protocol required For more.... see Table 7 in USB CDC Specification 1.2
	USB_CDC_NO_REPORT_DESCRIPTOR (see page 460)	No report descriptor found
	USB_CDC_NORMAL_RUNNING (see page 461)	Device is running and available for data transfers.
	USB_CDC_OBEX (see page 462)	OBEX
	USB_CDC_PHASE_ERROR (see page 463)	Command had a phase error at the device.
	USB_CDC_REPORT_DESCRIPTOR_BAD (see page 464)	Report Descriptor for not proper
	USB_CDC_RESET_ERROR (see page 465)	An error occurred while resetting the device.
	USB_CDC_RESETTING_DEVICE (see page 466)	Device is being reset.
	USB_CDC_SEND_BREAK (see page 467)	Sends special carrier modulation used to specify [V24] style break.
	USB_CDC_SEND_ENCAPSULATED_COMMAND (see page 468)	Issues a command in the format of the supported control protocol.
	USB_CDC_SET_COMM_FEATURE (see page 469)	Controls the settings for a particular communications feature.
	USB_CDC_SET_CONTROL_LINE_STATE (see page 470)	V24] signal used to tell the DCE device the DTE device is now present.
	USB_CDC_SET_LINE_CODING (see page 471)	Configures DTE rate, stop-bits, parity, and number-of-character bits.
	USB_CDC_TELEPHONE_CONTROL_MODEL (see page 472)	Telephone Control Model
	USB_CDC_V25TER (see page 473)	Common AT commands ("Hayes(TM)")
	USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL (see page 474)	Wireless Handset Control Model

Structures

	Name	Description
	_COMM_INTERFACE_DETAILS (see page 402)	This structure stores communication interface details of the attached CDC device

	_DATA_INTERFACE_DETAILS (see page 403)	This structure stores data interface details of the attached CDC device
	_USB_CDC_ACM_FN_DSC (see page 404)	Abstract Control Management Functional Descriptor
	_USB_CDC_CALL_MGT_FN_DSC (see page 405)	Call Management Functional Descriptor
	_USB_CDC_DEVICE_INFO (see page 407)	This structure is used to hold information about an attached CDC device
	_USB_CDC_HEADER_FN_DSC (see page 409)	Header Functional Descriptor
	_USB_CDC_UNION_FN_DSC (see page 411)	Union Functional Descriptor
	COMM_INTERFACE_DETAILS (see page 402)	This structure stores communication interface details of the attached CDC device
	DATA_INTERFACE_DETAILS (see page 403)	This structure stores data interface details of the attached CDC device
	USB_CDC_ACM_FN_DSC (see page 404)	Abstract Control Management Functional Descriptor
	USB_CDC_CALL_MGT_FN_DSC (see page 405)	Call Management Functional Descriptor
	USB_CDC_DEVICE_INFO (see page 407)	This structure is used to hold information about an attached CDC device
	USB_CDC_HEADER_FN_DSC (see page 409)	Header Functional Descriptor
	USB_CDC_UNION_FN_DSC (see page 411)	Union Functional Descriptor

Unions

	Name	Description
	_USB_CDC_CONTROL_SIGNAL_BITMAP (see page 406)	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
	_USB_CDC_LINE_CODING (see page 410)	This is type USB_CDC_LINE_CODING.
	USB_CDC_CONTROL_SIGNAL_BITMAP (see page 406)	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
	USB_CDC_LINE_CODING (see page 410)	This is type USB_CDC_LINE_CODING.

Description

7.2.5.2.1 COMM_INTERFACE_DETAILS Structure

File

usb_host_cdc.h

C

```
typedef struct _COMM_INTERFACE_DETAILS {
  BYTE interfaceNum;
  BYTE noOfEndpoints;
  USB_CDC_HEADER_FN_DSC Header_Fn_Dsc;
  USB_CDC_ACM_FN_DSC ACM_Fn_Desc;
  USB_CDC_UNION_FN_DSC Union_Fn_Desc;
  USB_CDC_CALL_MGT_FN_DSC Call_Mgt_Fn_Desc;
  WORD endpointMaxDataSize;
  WORD endpointInDataSize;
  WORD endpointOutDataSize;
  BYTE endpointPollInterval;
  BYTE endpointType;
  BYTE endpointIN;
  BYTE endpointOUT;
} COMM_INTERFACE_DETAILS;
```

Members

Members	Description
BYTE interfaceNum;	communication interface number
BYTE noOfEndpoints;	Number endpoints for communication interface Functional Descriptor Details
USB_CDC_HEADER_FN_DSC Header_Fn_Dsc;	Header Function Descriptor
USB_CDC_ACM_FN_DSC ACM_Fn_Desc;	Abstract Control Model Function Descriptor
USB_CDC_UNION_FN_DSC Union_Fn_Desc;	Union Function Descriptor
USB_CDC_CALL_MGT_FN_DSC Call_Mgt_Fn_Desc;	Call Management Function Descriptor Endpoint Descriptor Details
WORD endpointMaxDataSize;	Max data size for a interface.
WORD endpointInDataSize;	Max data size for a interface.
WORD endpointOutDataSize;	Max data size for a interface.
BYTE endpointPollInterval;	Polling rate of corresponding interface.
BYTE endpointType;	Endpoint type - either Isochronous or Bulk
BYTE endpointIN;	IN endpoint for comm interface.
BYTE endpointOUT;	IN endpoint for comm interface.

Description

This structure stores communication interface details of the attached CDC device

7.2.5.2.2 DATA_INTERFACE_DETAILS Structure

File

usb_host_cdc.h

C

```
typedef struct _DATA_INTERFACE_DETAILS {
  BYTE interfaceNum;
  BYTE noOfEndpoints;
  WORD endpointInDataSize;
  WORD endpointOutDataSize;
  BYTE endpointType;
  BYTE endpointIN;
  BYTE endpointOUT;
} DATA_INTERFACE_DETAILS;
```

Members

Members	Description
BYTE interfaceNum;	Data interface number
BYTE noOfEndpoints;	number of endpoints associated with data interface
WORD endpointInDataSize;	Max data size for a interface.
WORD endpointOutDataSize;	Max data size for a interface.
BYTE endpointType;	Endpoint type - either Isochronous or Bulk
BYTE endpointIN;	IN endpoint for comm interface.
BYTE endpointOUT;	IN endpoint for comm interface.

Description

This structure stores data interface details of the attached CDC device

7.2.5.2.3 USB_CDC_ACM_FN_DSC Structure

File

usb_host_cdc.h

C

```
typedef struct _USB_CDC_ACM_FN_DSC {
  BYTE bFNLength;
  BYTE bDscType;
  BYTE bDscSubType;
  BYTE bmCapabilities;
} USB_CDC_ACM_FN_DSC;
```

Members

Members	Description
BYTE bFNLength;	Size of this functional descriptor, in bytes.
BYTE bDscType;	CS_INTERFACE
BYTE bDscSubType;	Abstract Control Management functional descriptor subtype as defined in [USBCDC1.2].
BYTE bmCapabilities;	The capabilities that this configuration supports. (A bit value of zero means that the request is not supported.)

Description

Abstract Control Management Functional Descriptor

7.2.5.2.4 USB_CDC_CALL_MGT_FN_DSC Structure

File

usb_host_cdc.h

C

```
typedef struct _USB_CDC_CALL_MGT_FN_DSC {
  BYTE bFNLength;
  BYTE bDscType;
  BYTE bDscSubType;
  BYTE bmCapabilities;
  BYTE bDataInterface;
} USB_CDC_CALL_MGT_FN_DSC;
```

Members

Members	Description
BYTE bFNLength;	Size of this functional descriptor, in bytes.
BYTE bDscType;	CS_INTERFACE
BYTE bDscSubType;	Call Management functional descriptor subtype, as defined in [USBCDC1.2].
BYTE bmCapabilities;	The capabilities that this configuration supports:
BYTE bDataInterface;	Interface number of Data Class interface optionally used for call management.

Description

Call Management Functional Descriptor

7.2.5.2.5 USB_CDC_CONTROL_SIGNAL_BITMAP Union

File

usb_host_cdc.h

C

```
typedef union _USB_CDC_CONTROL_SIGNAL_BITMAP {
  BYTE _byte;
  struct {
    unsigned DTE_PRESENT : 1;
    unsigned CARRIER_CONTROL : 1;
  }
} USB_CDC_CONTROL_SIGNAL_BITMAP;
```

Members

Members	Description
unsigned DTE_PRESENT : 1;	0] Not Present [1] Present
unsigned CARRIER_CONTROL : 1;	0] Deactivate [1] Activate

Description

This is type USB_CDC_CONTROL_SIGNAL_BITMAP.

7.2.5.2.6 USB_CDC_DEVICE_INFO Structure

File

usb_host_cdc.h

C

```
typedef struct _USB_CDC_DEVICE_INFO {
  BYTE* userData;
  WORD reportSize;
  WORD remainingBytes;
  WORD bytesTransferred;
  union {
    struct {
      BYTE bfDirection : 1;
      BYTE bfReset : 1;
      BYTE bfClearDataIN : 1;
      BYTE bfClearDataOUT : 1;
    }
    BYTE val;
  } flags;
  BYTE driverSupported;
  BYTE deviceAddress;
  BYTE errorCode;
  BYTE state;
  BYTE returnState;
  BYTE noOfInterfaces;
  BYTE interface;
  BYTE endpointDATA;
  BYTE commRequest;
  BYTE clientDriverID;
  COMM_INTERFACE_DETAILS commInterface;
  DATA_INTERFACE_DETAILS dataInterface;
} USB_CDC_DEVICE_INFO;
```

Members

Members	Description
BYTE* userData;	Data pointer to application buffer.
WORD reportSize;	Total length of user data
WORD remainingBytes;	Number bytes remaining to be transferrerd in case user data length is more than 64 bytes
WORD bytesTransferred;	Number of bytes transferred to/from the user's data buffer.
BYTE bfDirection : 1;	Direction of current transfer (0=OUT, 1=IN).
BYTE bfReset : 1;	Flag indicating to perform CDC Reset.
BYTE bfClearDataIN : 1;	Flag indicating to clear the IN endpoint.
BYTE bfClearDataOUT : 1;	Flag indicating to clear the OUT endpoint.
BYTE driverSupported;	If CDC driver supports requested Class,Subclass & Protocol.
BYTE deviceAddress;	Address of the device on the bus.
BYTE errorCode;	Error code of last error.
BYTE state;	State machine state of the device.
BYTE returnState;	State to return to after performing error handling.
BYTE noOfInterfaces;	Total number of interfaces in the device.
BYTE interface;	Interface number of current transfer.
BYTE endpointDATA;	Endpoint to use for the current transfer.
BYTE commRequest;	Current Communication code
BYTE clientDriverID;	Client driver ID for device requests.
COMM_INTERFACE_DETAILS commInterface;	This structure stores communication interface details.

DATA_INTERFACE_DETAILS dataInterface;	This structure stores data interface details.

Description

This structure is used to hold information about an attached CDC device

7.2.5.2.7 USB_CDC_HEADER_FN_DSC Structure

File

usb_host_cdc.h

C

```
typedef struct _USB_CDC_HEADER_FN_DSC {
  BYTE bFNLength;
  BYTE bDscType;
  BYTE bDscSubType;
  BYTE bcdCDC[2];
} USB_CDC_HEADER_FN_DSC;
```

Members

Members	Description
BYTE bFNLength;	Size of this functional descriptor, in bytes.
BYTE bDscType;	CS_INTERFACE
BYTE bDscSubType;	Header. This is defined in [USBCDC1.2], which defines this as a header.
BYTE bcdCDC[2];	USB Class Definitions for Communications Devices Specification release number in binary-coded decimal.

Description

Header Functional Descriptor

7.2.5.2.8 USB_CDC_LINE_CODING Union

File

usb_host_cdc.h

C

```
typedef union _USB_CDC_LINE_CODING {
  struct {
    BYTE _byte[USB_CDC_LINE_CODING_LENGTH];
  }
  struct {
    DWORD_VAL dwDTERate;
    BYTE bCharFormat;
    BYTE bParityType;
    BYTE bDataBits;
  }
} USB_CDC_LINE_CODING;
```

Members

Members	Description
DWORD_VAL dwDTERate;	Data terminal rate, in bits per second.
BYTE bCharFormat;	Stop bits 0:1 Stop bit, 1:1.5 Stop bits, 2:2 Stop bits
BYTE bParityType;	Parity 0:None, 1:Odd, 2:Even, 3:Mark, 4:Space
BYTE bDataBits;	Data bits (5, 6, 7, 8 or 16)

Description

This is type USB_CDC_LINE_CODING.

7.2.5.2.9 USB_CDC_UNION_FN_DSC Structure

File

usb_host_cdc.h

C

```
typedef struct _USB_CDC_UNION_FN_DSC {
  BYTE bFNLength;
  BYTE bDscType;
  BYTE bDscSubType;
  BYTE bMasterIntf;
  BYTE bSaveIntf0;
} USB_CDC_UNION_FN_DSC;
```

Members

Members	Description
BYTE bFNLength;	Size of this functional descriptor, in bytes.
BYTE bDscType;	CS_INTERFACE
BYTE bDscSubType;	Union Descriptor Functional Descriptor subtype as defined in [USBCDC1.2].
BYTE bMasterIntf;	Interface number of the control (Communications Class) interface
BYTE bSaveIntf0;	Interface number of the subordinate (Data Class) interface

Description

Union Functional Descriptor

7.2.5.2.10 DEVICE_CLASS_CDC Macro

File

usb_host_cdc.h

C

```
#define DEVICE_CLASS_CDC 0x02 // CDC Interface Class Code
```

Description

CDC Interface Class Code

7.2.5.2.11 EVENT_CDC_COMM_READ_DONE Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_COMM_READ_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 2   // A CDC Communication Read transfer has completed
```

Description

A CDC Communication Read transfer has completed

7.2.5.2.12 EVENT_CDC_COMM_WRITE_DONE Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_COMM_WRITE_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 3   // A CDC
Communication Write transfer has completed
```

Description

A CDC Communication Write transfer has completed

7.2.5.2.13 EVENT_CDC_DATA_READ_DONE Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_DATA_READ_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 4   // A CDC Data Read
transfer has completed
```

Description

A CDC Data Read transfer has completed

7.2.5.2.14 EVENT_CDC_DATA_WRITE_DONE Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_DATA_WRITE_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 5   // A CDC Data
Write transfer has completed
```

Description

A CDC Data Write transfer has completed

7.2.5.2.15 EVENT_CDC_NAK_TIMEOUT Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_NAK_TIMEOUT EVENT_CDC_BASE + EVENT_CDC_OFFSET + 7   // CDC device NAK
timeout has occurred
```

Description

CDC device NAK timeout has occurred

7.2.5.2.16 EVENT_CDC_NONE Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_NONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 0   // No event occured (NULL
event)
```

Description

No event occured (NULL event)

7.2.5.2.17 EVENT_CDC_OFFSET Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_OFFSET 0
```

Description

If the application has not defined an offset for CDC events, set it to 0.

7.2.5.2.18 EVENT_CDC_RESET Macro

File

usb_host_cdc.h

C

```
#define EVENT_CDC_RESET EVENT_CDC_BASE + EVENT_CDC_OFFSET + 6   // CDC reset complete
```

Description

CDC reset complete

7.2.5.2.19 USB_CDC_ABSTRACT_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_ABSTRACT_CONTROL_MODEL 0x02 // Abstract Control Model
```

Description

Abstract Control Model

7.2.5.2.20 USB_CDC_ATM_NETWORKING_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_ATM_NETWORKING_CONTROL_MODEL 0x07 // ATM Networking Control Model
```

Description

ATM Networking Control Model

7.2.5.2.21 USB_CDC_CAPI_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_CAPI_CONTROL_MODEL 0x05 // CAPI Control Model
```

Description

CAPI Control Model

7.2.5.2.22 USB_CDC_CLASS_ERROR Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_CLASS_ERROR USB_ERROR_CLASS_DEFINED
```

Description

CDC Class Error Codes

7.2.5.2.23 USB_CDC_COMM_INTF Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_COMM_INTF 0x02 // Communication Interface Class Code
```

Description

Communication Interface Class Code

7.2.5.2.24 USB_CDC_COMMAND_FAILED Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_COMMAND_FAILED (USB_CDC_CLASS_ERROR | 0x01) // Command failed at the device.
```

Description

Command failed at the device.

7.2.5.2.25 USB_CDC_COMMAND_PASSED Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_COMMAND_PASSED USB_SUCCESS                // Command was successful.
```

Description

Command was successful.

7.2.5.2.26 USB_CDC_CONTROL_LINE_LENGTH Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_CONTROL_LINE_LENGTH 0x00   // Number of bytes Control line transfer
```

Description

Number of bytes Control line transfer

7.2.5.2.27 USB_CDC_CS_ENDPOINT Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_CS_ENDPOINT 0x25
```

Description

This is macro USB_CDC_CS_ENDPOINT.

7.2.5.2.28 USB_CDC_CS_INTERFACE Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_CS_INTERFACE 0x24
```

Description

Functional Descriptor Details Type Values for the bDscType Field

7.2.5.2.29 USB_CDC_DATA_INTF Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DATA_INTF 0x0A
```

Description

Data Interface Class Codes

7.2.5.2.30 USB_CDC_DEVICE_BUSY Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DEVICE_BUSY (USB_CDC_CLASS_ERROR | 0x04) // A transfer is currently in
progress.
```

Description

A transfer is currently in progress.

7.2.5.2.31 USB_CDC_DEVICE_DETACHED Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DEVICE_DETACHED 0x50    // Device is detached.
```

Description

Device is detached.

7.2.5.2.32 USB_CDC_DEVICE_HOLDING Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DEVICE_HOLDING 0x54     // Device is holding due to error
```

Description

Device is holding due to error

7.2.5.2.33 USB_CDC_DEVICE_MANAGEMENT Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DEVICE_MANAGEMENT 0x09 // Device Management
```

Description

Device Management

7.2.5.2.34 USB_CDC_DEVICE_NOT_FOUND Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DEVICE_NOT_FOUND (USB_CDC_CLASS_ERROR | 0x03) // Device with the specified
address is not available.
```

Description

Device with the specified address is not available.

7.2.5.2.35 USB_CDC_DIRECT_LINE_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DIRECT_LINE_CONTROL_MODEL 0x01 // Direct Line Control Model
```

Description

Direct Line Control Model

7.2.5.2.36 USB_CDC_DSC_FN_ACM Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_ACM 0x02    // ACM - Abstract Control Management
```

Description

ACM - Abstract Control Management

7.2.5.2.37 USB_CDC_DSC_FN_CALL_MGT Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_CALL_MGT 0x01
```

Description

This is macro USB_CDC_DSC_FN_CALL_MGT.

7.2.5.2.38 USB_CDC_DSC_FN_COUNTRY_SELECTION Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_COUNTRY_SELECTION 0x07
```

Description

This is macro USB_CDC_DSC_FN_COUNTRY_SELECTION.

7.2.5.2.39 USB_CDC_DSC_FN_DLM Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_DLM 0x03     // DLM - Direct Line Managment
```

Description

DLM - Direct Line Managment

7.2.5.2.40 USB_CDC_DSC_FN_HEADER Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_HEADER 0x00
```

Description

bDscSubType in Functional Descriptors

7.2.5.2.41 USB_CDC_DSC_FN_RPT_CAPABILITIES Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_RPT_CAPABILITIES 0x05
```

Description

This is macro USB_CDC_DSC_FN_RPT_CAPABILITIES.

7.2.5.2.42 USB_CDC_DSC_FN_TEL_OP_MODES Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_TEL_OP_MODES 0x08
```

Description

This is macro USB_CDC_DSC_FN_TEL_OP_MODES.

7.2.5.2.43 USB_CDC_DSC_FN_TELEPHONE_RINGER Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_TELEPHONE_RINGER 0x04
```

Description

This is macro USB_CDC_DSC_FN_TELEPHONE_RINGER.

7.2.5.2.44 USB_CDC_DSC_FN_UNION Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_UNION 0x06
```

Description

This is macro USB_CDC_DSC_FN_UNION.

7.2.5.2.45 USB_CDC_DSC_FN_USB_TERMINAL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_DSC_FN_USB_TERMINAL 0x09
```

Description

This is macro USB_CDC_DSC_FN_USB_TERMINAL.

7.2.5.2.46 USB_CDC_ETHERNET_EMULATION_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_ETHERNET_EMULATION_MODEL 0x0C // Ethernet Emulation Model
```

Description

Ethernet Emulation Model

7.2.5.2.47 USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL 0x06 // Ethernet Networking Control Model
```

Description

Ethernet Networking Control Model

7.2.5.2.48 USB_CDC_GET_COMM_FEATURE Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_GET_COMM_FEATURE 0x03    // Returns the current settings for the
communications feature.
```

Description

Returns the current settings for the communications feature.

7.2.5.2.49 USB_CDC_GET_ENCAPSULATED_REQUEST Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_GET_ENCAPSULATED_REQUEST 0x01    // Requests a response in the format of
the supported control protocol.
```

Description

Requests a response in the format of the supported control protocol.

7.2.5.2.50 USB_CDC_GET_LINE_CODING Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_GET_LINE_CODING 0x21    // Requests current DTE rate, stop-bits, parity,
and number-of-character bits.
```

Description

Requests current DTE rate, stop-bits, parity, and number-of-character bits.

7.2.5.2.51 USB_CDC_ILLEGAL_REQUEST Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_ILLEGAL_REQUEST (USB_CDC_CLASS_ERROR | 0x0B) // Cannot perform requested operation.
```

Description

Cannot perform requested operation.

7.2.5.2.52 USB_CDC_INITIALIZING Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_INITIALIZING 0x51    // Device is initializing.
```

Description

Device is initializing.

7.2.5.2.53 USB_CDC_INTERFACE_ERROR Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_INTERFACE_ERROR (USB_CDC_CLASS_ERROR | 0x06) // The interface layer cannot
support the device.
```

Description

The interface layer cannot support the device.

7.2.5.2.54 USB_CDC_LINE_CODING_LENGTH Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_LINE_CODING_LENGTH 0x07    // Number of bytes Line Coding transfer
```

Description

Number of bytes Line Coding transfer

7.2.5.2.55 USB_CDC_MOBILE_DIRECT_LINE_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_MOBILE_DIRECT_LINE_MODEL 0x0A // Mobile Direct Line Model
```

Description

Mobile Direct Line Model

7.2.5.2.56 USB_CDC_MULTI_CHANNEL_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_MULTI_CHANNEL_CONTROL_MODEL 0x04 // Multi-Channel Control Model
```

Description

Multi-Channel Control Model

7.2.5.2.57 USB_CDC_NO_PROTOCOL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_NO_PROTOCOL 0x00    // No class specific protocol required
```

Description

No class specific protocol required For more.... see Table 7 in USB CDC Specification 1.2

7.2.5.2.58 USB_CDC_NO_REPORT_DESCRIPTOR Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_NO_REPORT_DESCRIPTOR (USB_CDC_CLASS_ERROR | 0x05) // No report descriptor found
```

Description

No report descriptor found

7.2.5.2.59 USB_CDC_NORMAL_RUNNING Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_NORMAL_RUNNING 0x53    // Device is running and available for data
transfers.
```

Description

Device is running and available for data transfers.

7.2.5.2.60 USB_CDC_OBEX Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_OBEX 0x0B // OBEX
```

Description

OBEX

7.2.5.2.61 USB_CDC_PHASE_ERROR Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_PHASE_ERROR (USB_CDC_CLASS_ERROR | 0x02) // Command had a phase error at
the device.
```

Description

Command had a phase error at the device.

7.2.5.2.62 USB_CDC_REPORT_DESCRIPTOR_BAD Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_REPORT_DESCRIPTOR_BAD (USB_CDC_CLASS_ERROR | 0x05) // Report Descriptor for
not proper
```

Description

Report Descriptor for not proper

7.2.5.2.63 USB_CDC_RESET_ERROR Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_RESET_ERROR (USB_CDC_CLASS_ERROR | 0x0A) // An error occurred while
resetting the device.
```

Description

An error occurred while resetting the device.

7.2.5.2.64 USB_CDC_RESETTING_DEVICE Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_RESETTING_DEVICE 0x55    // Device is being reset.
```

Description

Device is being reset.

7.2.5.2.65 USB_CDC_SEND_BREAK Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_SEND_BREAK 0x23    // Sends special carrier modulation used to specify
[V24] style break.
```

Description

Sends special carrier modulation used to specify [V24] style break.

7.2.5.2.66 USB_CDC_SEND_ENCAPSULATED_COMMAND Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_SEND_ENCAPSULATED_COMMAND 0x00    // Issues a command in the format of the
supported control protocol.
```

Description

Issues a command in the format of the supported control protocol.

7.2.5.2.67 USB_CDC_SET_COMM_FEATURE Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_SET_COMM_FEATURE 0x02    // Controls the settings for a particular
communications feature.
```

Description

Controls the settings for a particular communications feature.

7.2.5.2.68 USB_CDC_SET_CONTROL_LINE_STATE Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_SET_CONTROL_LINE_STATE 0x22    // [V24] signal used to tell the DCE device
the DTE device is now present.
```

Description

V24] signal used to tell the DCE device the DTE device is now present.

7.2.5.2.69 USB_CDC_SET_LINE_CODING Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_SET_LINE_CODING 0x20    // Configures DTE rate, stop-bits, parity, and
number-of-character bits.
```

Description

Configures DTE rate, stop-bits, parity, and number-of-character bits.

7.2.5.2.70 USB_CDC_TELEPHONE_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_TELEPHONE_CONTROL_MODEL 0x03 // Telephone Control Model
```

Description

Telephone Control Model

7.2.5.2.71 USB_CDC_V25TER Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_V25TER 0x01     // Common AT commands ("Hayes(TM)")
```

Description

Common AT commands ("Hayes(TM)")

7.2.5.2.72 USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL Macro

File

usb_host_cdc.h

C

```
#define USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL 0x08 // Wireless Handset Control Model
```

Description

Wireless Handset Control Model

7.2.6 Charger Client Driver

This Client Driver gives an application the ability to charge the rechargeable batteries of many USB devices.

Description

USB devices that obey the USB specification will only draw 100mA from the bus until they are enumerated and less than 2.5mA when the bus is idle. In some situations this is not sufficient for a quick charge of the batteries. These devices often have a mode where they request more than 100mA but require permission from the host before drawing that current. This client driver simply allows a device to enumerate for the purpose of allowing this charging rate.

This client driver can be utilized for any device where the VID and PID are known. But the stack also contains a provision to allow a client driver to be used for any VID and PID by specifying a VID of 0xFFFF and a PID of 0xFFFF in the TPL. BE SURE THAT THIS IS THE LAST ENTRY IN THE TPL.

Chargers and devices can also follow the USB Battery Charging specification (http://www.usb.org/developers/devclass_docs/batt_charging_1_0.zip). Not all devices implement this specification, however, so some devices will not be able to charge with chargers using this specification.

7.2.6.1 Interface Routines

Functions

	Name	Description
	USBHostChargerDeviceDetached (see page 476)	This interface is used to check if the devich has been detached from the bus.
	USBHostChargerEventHandler (see page 477)	This routine is called by the Host layer to notify the charger client of events that occur.
	USBHostChargerGetDeviceAddress (see page 478)	This interface is used get the address of a specific generic device on the USB.

Description

7.2.6.1.1 USBHostChargerDeviceDetached Function

File

usb_host_charger.h

C

```
BOOL USBHostChargerDeviceDetached(
    BYTE deviceAddress
);
```

Description

This interface is used to check if the devich has been detached from the bus.

Remarks

None

Preconditions

None

Example

```
if (USBHostChargerDeviceDetached( deviceAddress ))
{
    // Handle detach
}
```

Parameters

Parameters	Description
deviceAddress	USB Address of the device.

Return Values

Return Values	Description
TRUE	The device has been detached, or an invalid deviceAddress is given.
FALSE	The device is attached

Function

BOOL USBHostChargerDeviceDetached(BYTE deviceAddress)

7.2.6.1.2 USBHostChargerEventHandler Function

This routine is called by the Host layer to notify the charger client of events that occur.

File

usb_host_charger.h

C

```
BOOL USBHostChargerEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This routine is called by the Host layer to notify the charger client of events that occur. If the event is recognized, it is handled and the routine returns TRUE. Otherwise, it is ignored and the routine returns FALSE.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of device with the event
USB_EVENT event	The bus event that occured
void *data	Pointer to event-specific data
DWORD size	Size of the event-specific data

Return Values

Return Values	Description
TRUE	The event was handled
FALSE	The event was not handled

Function

BOOL USBHostChargerEventHandler (BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.6.1.3 USBHostChargerGetDeviceAddress Function

File

usb_host_charger.h

C

```
BOOL USBHostChargerGetDeviceAddress(
    USB_CHARGING_DEVICE_ID * pDevID
);
```

Description

This interface is used get the address of a specific generic device on the USB.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
USB_CHARGING_DEVICE_ID  deviceID;
BYTE                    deviceAddress;

deviceID.vid          = 0x1234;
deviceID.pid          = 0x5678;

if (USBHostChargerGetDeviceAddress(&deviceID))
{
    deviceAddress = deviceID.deviceAddress;
}
```

Parameters

Parameters	Description
pDevID	Pointer to a structure containing the Device ID Info (VID, PID, and device address).

Return Values

Return Values	Description
TRUE	The device is connected
FALSE	The device is not connected.

Function

BOOL USBHostChargerGetDeviceAddress(USB_CHARGING_DEVICE_ID *pDevID)

7.2.6.2 Data Type and Constants

Macros

	Name	Description
	EVENT_CHARGER_ATTACH (see page 480)	This event indicates that a device has been attached for charging. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a USB_CHARGING_DEVICE_ID structure, and size is the size of the USB_CHARGING_DEVICE_ID structure.
	EVENT_CHARGER_DETACH (see page 481)	This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.
	EVENT_CHARGER_OFFSET (see page 482)	This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.
	USB_MAX_CHARGING_DEVICES (see page 483)	Max Number of Supported Devices This value represents the maximum number of attached devices this client driver can support. If the user does not define a value, it will be set to 1.

Description

7.2.6.2.1 EVENT_CHARGER_ATTACH Macro

File

usb_host_charger.h

C

```
#define EVENT_CHARGER_ATTACH (EVENT_CHARGER_BASE+EVENT_CHARGER_OFFSET+0)
```

Description

This event indicates that a device has been attached for charging. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a USB_CHARGING_DEVICE_ID structure, and size is the size of the USB_CHARGING_DEVICE_ID structure.

7.2.6.2.2 EVENT_CHARGER_DETACH Macro

File

usb_host_charger.h

C

```
#define EVENT_CHARGER_DETACH (EVENT_CHARGER_BASE+EVENT_CHARGER_OFFSET+1)
```

Description

This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.

7.2.6.2.3 EVENT_CHARGER_OFFSET Macro

File

usb_host_charger.h

C

```
#define EVENT_CHARGER_OFFSET 0
```

Description

This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.

7.2.6.2.4 USB_MAX_CHARGING_DEVICES Macro

File

usb_host_charger.h

C

```
#define USB_MAX_CHARGING_DEVICES 1
```

Description

Max Number of Supported Devices

This value represents the maximum number of attached devices this client driver can support. If the user does not define a value, it will be set to 1.

7.2.7 Generic Client Driver

This is a generic client driver for use with the USB Embedded Host driver.

Description

Many USB applications do not fall under the category of an existing class. For these applications, the developer can create a custom driver, and utilize the Generic client driver to communicate with the device.

The Generic class offers simple wrappers to USB functions, with additional device management support.

See *AN1143 - USB Generic Client on an Embedded Host* for more information about this client driver.

7.2.7.1 Interface Routines

Functions

	Name	Description
	USBHostGenericEventHandler (see page 486)	This routine is called by the Host layer to notify the general client of events that occur.
	USBHostGenericGetDeviceAddress (see page 487)	This interface is used get the address of a specific generic device on the USB.
	USBHostGenericInit (see page 489)	This function is called by the USB Embedded Host layer when a "generic" device attaches.
	USBHostGenericRead (see page 490)	Use this routine to receive from the device and store it into memory.
	USBHostGenericRxIsComplete (see page 492)	This routine indicates whether or not the last IN transfer is complete.
	USBHostGenericTxIsComplete (see page 494)	This routine indicates whether or not the last OUT transfer is complete.
	USBHostGenericWrite (see page 495)	Use this routine to transmit data from memory to the device.

Macros

	Name	Description
	USBHostGenericDeviceDetached (see page 485)	This interface is used to check if the devich has been detached from the bus.
	USBHostGenericGetRxLength (see page 488)	This function retrieves the number of bytes copied to user's buffer by the most recent call to the USBHostGenericRead (see page 490)() function.
	USBHostGenericRxIsBusy (see page 491)	This interface is used to check if the client driver is currently busy receiving data from the device.
	USBHostGenericTxIsBusy (see page 493)	This interface is used to check if the client driver is currently busy transmitting data to the device.

Description

7.2.7.1.1 USBHostGenericDeviceDetached Macro

File

usb_host_generic.h

C

```
#define USBHostGenericDeviceDetached(a) ( (((a)==gc_DevData.ID.deviceAddress) &&
gc_DevData.flags.initialized == 1) ? FALSE : TRUE )
```

Description

This interface is used to check if the devich has been detached from the bus.

Remarks

None

Preconditions

None

Example

```
if (USBHostGenericDeviceDetached( deviceAddress ))
{
    // Handle detach
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device.

Return Values

Return Values	Description
TRUE	The device has been detached, or an invalid deviceAddress is given.
FALSE	The device is attached

Function

BOOL USBHostGenericDeviceDetached(BYTE deviceAddress)

7.2.7.1.2 USBHostGenericEventHandler Function

This routine is called by the Host layer to notify the general client of events that occur.

File

usb_host_generic.h

C

```
BOOL USBHostGenericEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This routine is called by the Host layer to notify the general client of events that occur. If the event is recognized, it is handled and the routine returns TRUE. Otherwise, it is ignored and the routine returns FALSE.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of device with the event
USB_EVENT event	The bus event that occured
void *data	Pointer to event-specific data
DWORD size	Size of the event-specific data

Return Values

Return Values	Description
TRUE	The event was handled
FALSE	The event was not handled

Function

BOOL USBHostGenericEventHandler (BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.7.1.3 USBHostGenericGetDeviceAddress Function

File

usb_host_generic.h

C

```
BOOL USBHostGenericGetDeviceAddress(
    GENERIC_DEVICE_ID * pDevID
);
```

Description

This interface is used get the address of a specific generic device on the USB.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
GENERIC_DEVICE_ID   deviceID;
WORD                serialNumber[] = { '1', '2', '3', '4', '5', '6' };
BYTE                deviceAddress;

deviceID.vid          = 0x1234;
deviceID.pid          = 0x5678;
deviceID.serialNumber = &serialNumber;

if (USBHostGenericGetDeviceAddress(&deviceID))
{
    deviceAddress = deviceID.deviceAddress;
}
```

Parameters

Parameters	Description
GENERIC_DEVICE_ID* pDevID	Pointer to a structure containing the Device ID Info (VID, PID, serial number, and device address).

Return Values

Return Values	Description
TRUE	The device is connected
FALSE	The device is not connected.

Function

BOOL USBHostGenericGetDeviceAddress(GENERIC_DEVICE_ID (see page 498) *pDevID)

7.2.7.1.4 USBHostGenericGetRxLength Macro

File

usb_host_generic.h

C

```
#define USBHostGenericGetRxLength(a) ( (API_VALID(a)) ? gc_DevData.rxLength : 0 )
```

Returns

Returns the number of bytes most recently received from the Generic device with address deviceAddress.

Description

This function retrieves the number of bytes copied to user's buffer by the most recent call to the USBHostGenericRead (see page 490)() function.

Remarks

This function can only be called once per transfer. Subsequent calls will return zero until new data has been received.

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device

Function

DWORD USBHostGenericGetRxLength(BYTE deviceAddress)

7.2.7.1.5 USBHostGenericInit Function

This function is called by the USB Embedded Host layer when a "generic" device attaches.

File

usb_host_generic.h

C

```
BOOL USBHostGenericInit(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This routine is a call out from the USB Embedded Host layer to the USB generic client driver. It is called when a "generic" device has been connected to the host. Its purpose is to initialize and activate the USB Generic client driver.

Remarks

Multiple client drivers may be used in a single application. The USB Embedded Host layer will call the initialize routine required for the attached device.

Preconditions

The device has been configured.

Parameters

Parameters	Description
BYTE address	Device's address on the bus
DWORD flags	Initialization flags
BYTE clientDriverID	ID to send when issuing a Device Request via USBHostIssueDeviceRequest(), USBHostSetDeviceConfiguration (see page 305)(), or USBHostSetDeviceInterface().

Return Values

Return Values	Description
TRUE	Initialization was successful
FALSE	Initialization failed

Function

BOOL USBHostGenericInit (BYTE address, DWORD flags, BYTE clientDriverID)

7.2.7.1.6 USBHostGenericRead Function

File

usb_host_generic.h

C

```
BYTE USBHostGenericRead(
    BYTE deviceAddress,
    void * buffer,
    DWORD length
);
```

Description

Use this routine to receive from the device and store it into memory.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostGenericRxIsBusy( deviceAddress ))
{
    USBHostGenericRead( deviceAddress, &buffer, sizeof(buffer) );
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device.
BYTE *buffer	Pointer to the data buffer
DWORD length	Number of bytes to be transferred

Return Values

Return Values	Description
USB_SUCCESS	The Read was started successfully
(USB error code)	The Read was not started. See USBHostRead (see page 301)() for a list of errors.

Function

void USBHostGenericRead(BYTE deviceAddress, BYTE *buffer, DWORD length)

7.2.7.1.7 USBHostGenericRxIsBusy Macro

This interface is used to check if the client driver is currently busy receiving data from the device.

File

usb_host_generic.h

C

```
#define USBHostGenericRxIsBusy(a) ( (API_VALID(a)) ? ((gc_DevData.flags.rxBusy == 1) ? TRUE
: FALSE) : TRUE )
```

Description

This interface is used to check if the client driver is currently busy receiving data from the device. This function is intended for use with transfer events. With polling, the function USBHostGenericRxIsComplete (see page 492)() should be used.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostGenericRxIsBusy( deviceAddress ))
{
    USBHostGenericRead( deviceAddress, &buffer, sizeof( buffer ) );
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device

Return Values

Return Values	Description
TRUE	The device is receiving data or an invalid deviceAddress is given.
FALSE	The device is not receiving data

Function

BOOL USBHostGenericRxIsBusy(BYTE deviceAddress)

7.2.7.1.8 USBHostGenericRxIsComplete Function

This routine indicates whether or not the last IN transfer is complete.

File

usb_host_generic.h

C

```
BOOL USBHostGenericRxIsComplete(
    BYTE deviceAddress,
    BYTE * errorCode,
    DWORD * byteCount
);
```

Description

This routine indicates whether or not the last IN transfer is complete. If it is, then the returned errorCode and byteCount are valid, and reflect the error code and the number of bytes received.

This function is intended for use with polling. With transfer events, the function USBHostGenericRxIsBusy (see page 491)() should be used.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of the attached peripheral
BYTE *errorCode	Error code of the last transfer, if complete
DWORD *byteCount	Bytes transferred during the last transfer, if complete

Return Values

Return Values	Description
TRUE	The IN transfer is complete. errorCode and byteCount are valid.
FALSE	The IN transfer is not complete. errorCode and byteCount are invalid.

Function

BOOL USBHostGenericRxIsComplete(BYTE deviceAddress, BYTE *errorCode,

DWORD *byteCount)

7.2.7.1.9 USBHostGenericTxIsBusy Macro

This interface is used to check if the client driver is currently busy transmitting data to the device.

File

usb_host_generic.h

C

```
#define USBHostGenericTxIsBusy(a) ( (API_VALID(a)) ? ((gc_DevData.flags.txBusy == 1) ? TRUE
: FALSE) : TRUE )
```

Description

This interface is used to check if the client driver is currently busy transmitting data to the device. This function is intended for use with transfer events. With polling, the function USBHostGenericTxIsComplete (see page 494)() should be used.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostGenericTxIsBusy( deviceAddress ) )
{
    USBHostGenericWrite( deviceAddress, &buffer, sizeof( buffer ) );
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device

Return Values

Return Values	Description
TRUE	The device is transmitting data or an invalid deviceAddress is given.
FALSE	The device is not transmitting data

Function

BOOL USBHostGenericTxIsBusy(BYTE deviceAddress)

7.2.7.1.10 USBHostGenericTxIsComplete Function

This routine indicates whether or not the last OUT transfer is complete.

File

usb_host_generic.h

C

```
BOOL USBHostGenericTxIsComplete(
    BYTE deviceAddress,
    BYTE * errorCode
);
```

Description

This routine indicates whether or not the last OUT transfer is complete. If it is, then the returned errorCode is valid, and reflect the error code of the transfer.

This function is intended for use with polling. With transfer events, the function USBHostGenericTxIsBusy (see page 493)() should be used.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of the attached peripheral
BYTE *errorCode	Error code of the last transfer, if complete

Return Values

Return Values	Description
TRUE	The OUT transfer is complete. errorCode is valid.
FALSE	The OUT transfer is not complete. errorCode is invalid.

Function

BOOL USBHostGenericTxIsComplete(BYTE deviceAddress, BYTE *errorCode)

7.2.7.1.11 USBHostGenericWrite Function

File

usb_host_generic.h

C

```
BYTE USBHostGenericWrite(
    BYTE deviceAddress,
    void * buffer,
    DWORD length
);
```

Description

Use this routine to transmit data from memory to the device.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostGenericTxIsBusy( deviceAddress ))
{
    USBHostGenericWrite( deviceAddress, &buffer, sizeof(buffer) );
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device.
BYTE *buffer	Pointer to the data buffer
DWORD length	Number of bytes to be transferred

Return Values

Return Values	Description
USB_SUCCESS	The Write was started successfully
(USB error code)	The Write was not started. See USBHostWrite (see page 313)() for a list of errors.

Function

void USBHostGenericWrite(BYTE deviceAddress, BYTE *buffer, DWORD length)

7.2.7.2 Data Types and Constants

Macros

	Name	Description
	EVENT_GENERIC_ATTACH (see page 499)	This event indicates that a Generic device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a GENERIC_DEVICE_ID (see page 498) structure, and size is the size of the GENERIC_DEVICE_ID (see page 498) structure.
	EVENT_GENERIC_DETACH (see page 500)	This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.
	EVENT_GENERIC_OFFSET (see page 501)	This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.
	EVENT_GENERIC_RX_DONE (see page 502)	This event indicates that a previous read request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the receive buffer, and size is the actual number of bytes read from the device.
	EVENT_GENERIC_TX_DONE (see page 503)	This event indicates that a previous write request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.
	USB_GENERIC_EP (see page 504)	This is the default Generic Client Driver endpoint number.

Types

	Name	Description
	GENERIC_DEVICE (see page 497)	Generic Device Information This structure contains information about an attached device, including status flags and device identification.
	GENERIC_DEVICE_ID (see page 498)	Generic Device ID Information This structure contains identification information about an attached device.

Description

7.2.7.2.1 GENERIC_DEVICE Type

File

usb_host_generic.h

C

```
typedef struct _GENERIC_DEVICE GENERIC_DEVICE;
```

Description

Generic Device Information

This structure contains information about an attached device, including status flags and device identification.

7.2.7.2.2 GENERIC_DEVICE_ID Type

File

usb_host_generic.h

C

```
typedef struct _GENERIC_DEVICE_ID GENERIC_DEVICE_ID;
```

Description

Generic Device ID Information

This structure contains identification information about an attached device.

7.2.7.2.3 EVENT_GENERIC_ATTACH Macro

File

usb_host_generic.h

C

```
#define EVENT_GENERIC_ATTACH (EVENT_GENERIC_BASE+EVENT_GENERIC_OFFSET+0)
```

Description

This event indicates that a Generic device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a GENERIC_DEVICE_ID (see page 498) structure, and size is the size of the GENERIC_DEVICE_ID (see page 498) structure.

7.2.7.2.4 EVENT_GENERIC_DETACH Macro

File

usb_host_generic.h

C

```
#define EVENT_GENERIC_DETACH (EVENT_GENERIC_BASE+EVENT_GENERIC_OFFSET+1)
```

Description

This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.

7.2.7.2.5 EVENT_GENERIC_OFFSET Macro

File

usb_host_generic.h

C

```
#define EVENT_GENERIC_OFFSET 0
```

Description

This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.

7.2.7.2.6 EVENT_GENERIC_RX_DONE Macro

File

usb_host_generic.h

C

```
#define EVENT_GENERIC_RX_DONE (EVENT_GENERIC_BASE+EVENT_GENERIC_OFFSET+3)
```

Description

This event indicates that a previous read request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the receive buffer, and size is the actual number of bytes read from the device.

7.2.7.2.7 EVENT_GENERIC_TX_DONE Macro

File

usb_host_generic.h

C

```
#define EVENT_GENERIC_TX_DONE (EVENT_GENERIC_BASE+EVENT_GENERIC_OFFSET+2)
```

Description

This event indicates that a previous write request has completed. These events are enabled if USB Embedded Host transfer events are enabled (USB_ENABLE_TRANSFER_EVENT is defined). When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.

7.2.7.2.8 USB_GENERIC_EP Macro

File

usb_host_generic.h

C

```
#define USB_GENERIC_EP 1
```

Description

This is the default Generic Client Driver endpoint number.

7.2.8 HID Client Driver

This client driver provides USB Embedded Host support for HID devices.

Description

This client driver provides USB Embedded Host support for HID devices. Common HID devices include mice, keyboards, and bar code scanners. Many other USB peripherals also use the HID class to transfer data, since it provides a simple, flexible interface and does not require a custom Windows driver when used with a PC.

See *AN1144 - USB HID Class on an Embedded Host* and *AN1212 - Using USB Keyboard with an Embedded Host* for more information.

7.2.8.1 Interface Routines

Functions

	Name	Description
	USBHostHID_ApiFindBit (see page 507)	This function is used to locate a specific button or indicator. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application
	USBHostHID_ApiFindValue (see page 508)	Find a specific Usage Value. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application.
	USBHostHID_ApiGetCurrentInterfaceNum (see page 509)	This function reurns the interface number of the cuurent report descriptor parsed. This function must be called to fill data interface detail data structure and passed as parameter when requesinf for report transfers.
	USBHostHID_ApiImportData (see page 511)	This function can be used by application to extract data from the input reports. On receiving the input report from the device application can call the function with required inputs 'HID_DATA_DETAILS (see page 548)'.
	USBHostHID_HasUsage (see page 516)	This function is used to locate the usage in a report descriptor. Function will look into the data structures created by the HID parser and return the appropriate location.
	USBHostHIDDeviceDetect (see page 517)	This function determines if a HID device is attached and ready to use.
	USBHostHIDDeviceStatus (see page 518)	
	USBHostHIDEventHandler (see page 519)	This function is the event handler for this client driver.
	USBHostHIDInitialize (see page 520)	This function is the initialization routine for this client driver.
	USBHostHIDResetDevice (see page 522)	This function starts a HID reset.
	USBHostHIDResetDeviceWithWait (see page 523)	This function resets a HID device, and waits until the reset is complete.
	USBHostHIDTasks (see page 524)	This function performs the maintenance tasks required by HID class
	USBHostHIDTerminateTransfer (see page 525)	This function terminates a transfer that is in progress.
	USBHostHIDTransfer (see page 526)	This function starts a HID transfer.
	USBHostHIDTransferIsComplete (see page 527)	This function indicates whether or not the last transfer is complete.

Macros

	Name	Description
	USBHostHID_ApiGetReport (see page 510)	This macro provides legacy support for an older API function.
	USBHostHID_ApiSendReport (see page 512)	This macro provides legacy support for an older API function.
	USBHostHID_ApiTransferIsComplete (see page 513)	This macro provides legacy support for an older API function.
	USBHostHID_GetCurrentReportInfo (see page 514)	This function returns a pointer to the current report info structure.
	USBHostHID_GetItemListPointers (see page 515)	This function returns a pointer to list of item pointers stored in a structure.

	USBHostHIDRead (see page 521)	This function starts a Get report transfer reuest from the device, utilizing the function USBHostHIDTransfer (see page 526)();
	USBHostHIDWrite (see page 528)	This function starts a Set report transfer request to the device, utilizing the function USBHostHIDTransfer (see page 526)();

Description

7.2.8.1.1 USBHostHID_ApiFindBit Function

File

usb_host_hid.h

C

```
BOOL USBHostHID_ApiFindBit(
    WORD usagePage,
    WORD usage,
    HIDReportTypeEnum type,
    BYTE* Report_ID,
    BYTE* Report_Length,
    BYTE* Start_Bit
);
```

Description

This function is used to locate a specific button or indicator. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application

Remarks

Application event handler with event 'EVENT_HID_RPT_DESC_PARSED (see page 545)' is called. Application is suppose to fill in data details in structure 'HID_DATA_DETAILS (see page 548)'. This function can be used to the get the details of the required usages.

Preconditions

None

Parameters

Parameters	Description
WORD usagePage	usage page supported by application
WORD usage	usage supported by application
HIDReportTypeEnum type	report type Input/Output for the particular usage
BYTE* Report_ID	returns the report ID of the required usage
BYTE* Report_Length	returns the report length of the required usage
BYTE* Start_Bit	returns the start bit of the usage in a particular report

Return Values

Return Values	Description
TRUE	If the required usage is located in the report descriptor
FALSE	If the application required usage is not supported by the device(i.e report descriptor).

Function

BOOL USBHostHID_ApiFindBit(WORD usagePage,WORD usage, HIDReportTypeEnum (see page 557) type,

BYTE* Report_ID, BYTE* Report_Length, BYTE* Start_Bit)

7.2.8.1.2 USBHostHID_ApiFindValue Function

File

usb_host_hid.h

C

```
BOOL USBHostHID_ApiFindValue(
    WORD usagePage,
    WORD usage,
    HIDReportTypeEnum type,
    BYTE* Report_ID,
    BYTE* Report_Length,
    BYTE* Start_Bit,
    BYTE* Bit_Length
);
```

Description

Find a specific Usage Value. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application.

Remarks

Application event handler with event 'EVENT_HID_RPT_DESC_PARSED (see page 545)' is called. Application is suppose to fill in data details structure 'HID_DATA_DETAILS (see page 548)' This function can be used to the get the details of the required usages.

Preconditions

None

Parameters

Parameters	Description
WORD usagePage	usage page supported by application
WORD usage	usage supported by application
HIDReportTypeEnum type	report type Input/Output for the particular usage
BYTE* Report_ID	returns the report ID of the required usage
BYTE* Report_Length	returns the report length of the required usage
BYTE* Start_Bit	returns the start bit of the usage in a particular report
BYTE* Bit_Length	returns size of requested usage type data in bits

Return Values

Return Values	Description
TRUE	If the required usage is located in the report descriptor
FALSE	If the application required usage is not supported by the device(i.e report descriptor).

Function

BOOL USBHostHID_ApiFindValue(WORD usagePage,WORD usage,

HIDReportTypeEnum (see page 557) type,BYTE* Report_ID,BYTE* Report_Length,BYTE*

Start_Bit, BYTE* Bit_Length)

7.2.8.1.3 USBHostHID_ApiGetCurrentInterfaceNum Function

File

usb_host_hid.h

C

```
BYTE USBHostHID_ApiGetCurrentInterfaceNum();
```

Description

This function reurns the interface number of the cuurent report descriptor parsed. This function must be called to fill data interface detail data structure and passed as parameter when requesinf for report transfers.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	Transfer is complete, errorCode is valid
FALSE	Transfer is not complete, errorCode is not valid

Function

BYTE USBHostHID_ApiGetCurrentInterfaceNum(void)

7.2.8.1.4 USBHostHID_ApiGetReport Macro

File

usb_host_hid.h

C

```
#define USBHostHID_ApiGetReport( r, i, s, d ) USBHostHIDRead( 1, r, i, s, d )
```

Description

This macro provides legacy support for an older API function.

7.2.8.1.5 USBHostHID_ApiImportData Function

File

usb_host_hid.h

C

```
BOOL USBHostHID_ApiImportData(
    BYTE * report,
    WORD reportLength,
    HID_USER_DATA_SIZE * buffer,
    HID_DATA_DETAILS * pDataDetails
);
```

Description

This function can be used by application to extract data from the input reports. On receiving the input report from the device application can call the function with required inputs 'HID_DATA_DETAILS (see page 548)'.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE *report	Input report received from device
WORD reportLength	Length of input report report
HID_USER_DATA_SIZE *buffer	Buffer into which data needs to be populated
HID_DATA_DETAILS *pDataDetails	data details extracted from report descriptor

Return Values

Return Values	Description
TRUE	If the required data is retrieved from the report
FALSE	If required data is not found.

Function

BOOL USBHostHID_ApiImportData(BYTE *report, WORD reportLength,

HID_USER_DATA_SIZE *buffer, HID_DATA_DETAILS (see page 548) *pDataDetails)

7.2.8.1.6 USBHostHID_ApiSendReport Macro

File

usb_host_hid.h

C

```
#define USBHostHID_ApiSendReport( r, i, s, d ) USBHostHIDWrite( 1, r, i, s, d )
```

Description

This macro provides legacy support for an older API function.

7.2.8.1.7 USBHostHID_ApiTransferIsComplete Macro

File

usb_host_hid.h

C

```
#define USBHostHID_ApiTransferIsComplete( e, c ) USBHostHIDTransferIsComplete( 1, e, c )
```

Description

This macro provides legacy support for an older API function.

7.2.8.1.8 USBHostHID_GetCurrentReportInfo Macro

File

usb_host_hid.h

C

```
#define USBHostHID_GetCurrentReportInfo (&deviceRptInfo)
```

Returns

BYTE * - Pointer to the report Info structure.

Description

This function returns a pointer to the current report info structure.

Remarks

None

Preconditions

None

Function

BYTE* USBHostHID_GetCurrentReportInfo(void)

7.2.8.1.9 USBHostHID_GetItemListPointers Macro

File

usb_host_hid.h

C

```
#define USBHostHID_GetItemListPointers (&itemListPtrs)
```

Returns

BYTE * - Pointer to list of item pointers structure.

Description

This function returns a pointer to list of item pointers stored in a structure.

Remarks

None

Preconditions

None

Function

BYTE* USBHostHID_GetItemListPointers()

7.2.8.1.10 USBHostHID_HasUsage Function

File

usb_host_hid_parser.h

C

```
BOOL USBHostHID_HasUsage(
    HID_REPORTITEM * reportItem,
    WORD usagePage,
    WORD usage,
    WORD * pindex,
    BYTE* count
);
```

Description

This function is used to locate the usage in a report descriptor. Function will look into the data structures created by the HID parser and return the appropriate location.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
HID_REPORTITEM *reportItem	Report item index to be searched
WORD usagePage	Application needs to pass the usagePage as the search criteria for the usage
WORD usage	Application needs to pass the usageto be searched
WORD *pindex	returns index to the usage item requested.
BYTE* count	returns the remaining number of reports

Return Values

Return Values	Description
BOOL	FALSE - If requested usage is not found
TRUE	if requested usage is found

Function

BOOL USBHostHID_HasUsage(HID_REPORTITEM (see page 553) *reportItem, WORD usagePage,

WORD usage, WORD *pindex, BYTE* count)

7.2.8.1.11 USBHostHIDDeviceDetect Function

File

usb_host_hid.h

C

```
BOOL USBHostHIDDeviceDetect(
    BYTE deviceAddress
);
```

Description

This function determines if a HID device is attached and ready to use.

Remarks

This function replaces the USBHostHID_ApiDeviceDetect() function.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of the attached device.

Return Values

Return Values	Description
TRUE	HID present and ready
FALSE	HID not present or not ready

Function

BOOL USBHostHIDDeviceDetect(BYTE deviceAddress)

7.2.8.1.12 USBHostHIDDeviceStatus Function

File

usb_host_hid.h

C

```
BYTE USBHostHIDDeviceStatus(
    BYTE deviceAddress
);
```

Description

This function determines the status of a HID device.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	address of device to query

Return Values

Return Values	Description
USB_HID_DEVICE_NOT_FOUND (see page 565)	Illegal device address, or the device is not an HID
USB_HID_INITIALIZING (see page 569)	HID is attached and in the process of initializing
USB_PROCESSING_REPORT_DESCRIPTOR (see page 579)	HID device is detected and report descriptor is being parsed
USB_HID_NORMAL_RUNNING (see page 573)	HID Device is running normal, ready to send and receive reports
USB_HID_DEVICE_HOLDING (see page 563)	Driver has encountered error and could not recover
USB_HID_DEVICE_DETACHED (see page 562)	HID detached.

Function

BYTE USBHostHIDDeviceStatus(BYTE deviceAddress)

7.2.8.1.13 USBHostHIDEventHandler Function

This function is the event handler for this client driver.

File

usb_host_hid.h

C

```
BOOL USBHostHIDEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
DWORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostHIDEventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.8.1.14 USBHostHIDInitialize Function

This function is the initialization routine for this client driver.

File

usb_host_hid.h

C

```
BOOL USBHostHIDInitialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated.For a HID device we need to look into HID descriptor, interface descriptor and endpoint descriptor.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	Client driver identification for device requests

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostHIDInitialize(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.8.1.15 USBHostHIDRead Macro

This function starts a Get report transfer reuest from the device, utilizing the function USBHostHIDTransfer (see page 526)();

File

usb_host_hid.h

C

```
#define USBHostHIDRead( deviceAddress,reportid,interface,size,data) \
         USBHostHIDTransfer( deviceAddress,1,interface,reportid,size,data)
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE reportid	Report ID of the requested report
BYTE interface	Interface number
BYTE size	Byte size of the data buffer
BYTE *data	Pointer to the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_HID_DEVICE_NOT_FOUND (see page 565)	No device with specified address
USB_HID_DEVICE_BUSY (see page 561)	Device not in proper state for performing a transfer
Others	Return values from USBHostRead (see page 301)()

Function

BYTE USBHostHIDRead(BYTE deviceAddress,BYTE reportid, BYTE interface,

BYTE size, BYTE *data)

7.2.8.1.16 USBHostHIDResetDevice Function

This function starts a HID reset.

File

usb_host_hid.h

C

```
BYTE USBHostHIDResetDevice(
    BYTE deviceAddress
);
```

Description

This function starts a HID reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Reset started
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_ILLEGAL_REQUEST (see page 621)	Device is in an illegal state for reset

Function

BYTE USBHostHIDResetDevice(BYTE deviceAddress)

7.2.8.1.17 USBHostHIDResetDeviceWithWait Function

File

usb_host_hid.h

C

```
BYTE USBHostHIDResetDeviceWithWait(
    BYTE deviceAddress
);
```

Description

This function resets a HID device, and waits until the reset is complete.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Address of the device to reset.

Return Values

Return Values	Description
USB_SUCCESS	Reset successful
USB_HID_RESET_ERROR (see page 576)	Error while resetting device
Others	See return values for USBHostHIDResetDevice (see page 522)() and error codes that can be returned in the errorCode parameter of USBHostHIDTransferIsComplete (see page 527)();

Function

BOOL USBHostHIDResetDeviceWithWait(BYTE deviceAddress)

7.2.8.1.18 USBHostHIDTasks Function

This function performs the maintenance tasks required by HID class

File

usb_host_hid.h

C

```
void USBHostHIDTasks();
```

Returns

None

Description

This function performs the maintenance tasks required by the HID class. If transfer events from the host layer are not being used, then it should be called on a regular basis by the application. If transfer events from the host layer are being used, this function is compiled out, and does not need to be called.

Remarks

None

Preconditions

USBHostHIDInitialize (see page 520)() has been called.

Function

void USBHostHIDTasks(void)

7.2.8.1.19 USBHostHIDTerminateTransfer Function

This function terminates a transfer that is in progress.

File

usb_host_hid.h

C

```
BYTE USBHostHIDTerminateTransfer(
    BYTE deviceAddress,
    BYTE direction,
    BYTE interfaceNum
);
```

Description

This function terminates a transfer that is in progress.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE direction	Transfer direction. Valid values are: • 1 = In (Read) • 0 = Out (Write)
BYTE interfaceNum	Interface number

Return Values

Return Values	Description
USB_SUCCESS	Transfer terminated
USB_HID_DEVICE_NOT_FOUND (see page 565)	No device with specified address

Function

BYTE USBHostHIDTerminateTransfer(BYTE deviceAddress, BYTE direction, BYTE interfaceNum)

7.2.8.1.20 USBHostHIDTransfer Function

This function starts a HID transfer.

File

usb_host_hid.h

C

```
BYTE USBHostHIDTransfer(
    BYTE deviceAddress,
    BYTE direction,
    BYTE interfaceNum,
    WORD reportid,
    WORD size,
    BYTE * data
);
```

Description

This function starts a HID transfer. A read/write wrapper is provided in application interface file to access this function.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE direction	1=read, 0=write
BYTE interfaceNum	Interface number
BYTE reportid	Report ID of the requested report
BYTE size	Byte size of the data buffer
BYTE *data	Pointer to the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_HID_DEVICE_NOT_FOUND (see page 565)	No device with specified address
USB_HID_DEVICE_BUSY (see page 561)	Device not in proper state for performing a transfer
Others	Return values from USBHostIssueDeviceRequest(), USBHostRead (see page 301)(), and USBHostWrite (see page 313)()

Function

USBHostHIDTransfer(BYTE deviceAddress, BYTE direction, BYTE interfaceNum,

BYTE reportid, BYTE size, BYTE *data)

7.2.8.1.21 USBHostHIDTransferIsComplete Function

This function indicates whether or not the last transfer is complete.

File

usb_host_hid.h

C

```
BOOL USBHostHIDTransferIsComplete(
    BYTE deviceAddress,
    BYTE * errorCode,
    BYTE * byteCount
);
```

Description

This function indicates whether or not the last transfer is complete. If the functions returns TRUE, the returned byte count and error code are valid. Since only one transfer can be performed at once and only one endpoint can be used, we only need to know the device address.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE *errorCode	Error code from last transfer
DWORD *byteCount	Number of bytes transferred

Return Values

Return Values	Description
TRUE	Transfer is complete, errorCode is valid
FALSE	Transfer is not complete, errorCode is not valid

Function

BOOL USBHostHIDTransferIsComplete(BYTE deviceAddress,

BYTE *errorCode, DWORD *byteCount)

7.2.8.1.22 USBHostHIDWrite Macro

This function starts a Set report transfer request to the device, utilizing the function USBHostHIDTransfer (see page 526)();

File

usb_host_hid.h

C

```
#define USBHostHIDWrite( address,reportid,interface,size,data) \
              USBHostHIDTransfer( address,0,interface,reportid,size,data)
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE reportid	Report ID of the requested report
BYTE interface	Interface number
BYTE size	Byte size of the data buffer
BYTE *data	Pointer to the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_HID_DEVICE_NOT_FOUND (see page 565)	No device with specified address
USB_HID_DEVICE_BUSY (see page 561)	Device not in proper state for performing a transfer
Others	Return values from USBHostIssueDeviceRequest(), and USBHostWrite (see page 313)()

Function

BYTE USBHostHIDWrite(BYTE deviceAddress,BYTE reportid, BYTE interface,

BYTE size, BYTE *data)

7.2.8.2 Data Types and Constants

Enumerations

	Name	Description
	HIDReportTypeEnum (see page 557)	This is type HIDReportTypeEnum.
	USB_HID_RPT_DESC_ERROR (see page 578)	HID parser error codes This enumerates the error encountered during the parsing of report descriptor. In case of any error parsing is sttopped and the error is flagged. Device is not attched successfully.

Macros

	Name	Description
	DEVICE_CLASS_HID (see page 533)	HID Interface Class Code
	DSC_HID (see page 534)	HID Descriptor Code
	DSC_PHY (see page 535)	Pysical Descriptor Code
	DSC_RPT (see page 536)	Report Descriptor Code
	EVENT_HID_ATTACH (see page 537)	A HID device has attached. The returned data pointer points to a USB_HID_DEVICE_ID (see page 564) structure.
	EVENT_HID_BAD_REPORT_DESCRIPTOR (see page 538)	There was a problem parsing the report descriptor of the attached device. Communication with the device is not allowed, and the device should be detached.
	EVENT_HID_DETACH (see page 539)	A HID device has detached. The returned data pointer points to a byte with the previous address of the detached device.
	EVENT_HID_NONE (see page 540)	No event occured (NULL event)
	EVENT_HID_OFFSET (see page 541)	If the application has not defined an offset for HID events, set it to 0.
	EVENT_HID_READ_DONE (see page 542)	#define EVENT_HID_TRANSFER EVENT_HID_BASE + EVENT_HID_OFFSET (see page 541) + 3 // Unused - value retained for legacy. A HID Read transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA (see page 555) structure, with information about the transfer.
	EVENT_HID_RESET (see page 543)	HID reset complete. The returned data pointer is NULL.
	EVENT_HID_RESET_ERROR (see page 544)	An error occurred while trying to do a HID reset. The returned data pointer is NULL.
	EVENT_HID_RPT_DESC_PARSED (see page 545)	A Report Descriptor has been parsed. The returned data pointer is NULL. The application must collect details, or simply return TRUE if the application is already aware of the data format.
	EVENT_HID_WRITE_DONE (see page 546)	A HID Write transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA (see page 555) structure, with information about the transfer.
	USB_HID_CLASS_ERROR (see page 558)	
	USB_HID_COMMAND_FAILED (see page 559)	Command failed at the device.
	USB_HID_COMMAND_PASSED (see page 560)	Command was successful.
	USB_HID_DEVICE_BUSY (see page 561)	A transfer is currently in progress.
	USB_HID_DEVICE_DETACHED (see page 562)	Device is detached.

	USB_HID_DEVICE_HOLDING (see page 563)	Device is holding due to error
	USB_HID_DEVICE_NOT_FOUND (see page 565)	Device with the specified address is not available.
	USB_HID_ILLEGAL_REQUEST (see page 568)	Cannot perform requested operation.
	USB_HID_INITIALIZING (see page 569)	Device is initializing.
	USB_HID_INTERFACE_ERROR (see page 570)	The interface layer cannot support the device.
	USB_HID_NO_REPORT_DESCRIPTOR (see page 572)	No report descriptor found
	USB_HID_NORMAL_RUNNING (see page 573)	Device is running and available for data transfers.
	USB_HID_PHASE_ERROR (see page 574)	Command had a phase error at the device.
	USB_HID_REPORT_DESCRIPTOR_BAD (see page 575)	Report Descriptor for not proper
	USB_HID_RESET_ERROR (see page 576)	An error occurred while resetting the device.
	USB_HID_RESETTING_DEVICE (see page 577)	Device is being reset.
	USB_PROCESSING_REPORT_DESCRIPTOR (see page 579)	Parser is processing report descriptor.

Structures

	Name	Description
	_HID_COLLECTION (see page 547)	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
	_HID_DATA_DETAILS (see page 548)	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue (see page 508)
	_HID_GLOBALS (see page 550)	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
	_HID_ITEM_INFO (see page 551)	HID Item Information This structure contains information about each Item of the report descriptor.
	_HID_REPORT (see page 552)	HID Report details This structure contains information about each report exchanged with the device.
	_HID_REPORTITEM (see page 553)	HID Report Details This structure contains information about each Report encountered in the report descriptor.
	_HID_STRINGITEM (see page 554)	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	_HID_TRANSFER_DATA (see page 555)	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE (see page 542) or EVENT_HID_WRITE_DONE (see page 546)).
	_HID_USAGEITEM (see page 556)	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
	_USB_HID_DEVICE_ID (see page 564)	HID Device ID Information This structure contains identification information about an attached device.

	_USB_HID_DEVICE_RPT_INFO (see page 566)	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during th ecourse of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much inportance to the application.
	_USB_HID_ITEM_LIST (see page 571)	List of Items This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrive data from the report received from the device. Application can also access these details to retreive the intended information incase provided interface function fail to do so.
	HID_COLLECTION (see page 547)	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
	HID_DATA_DETAILS (see page 548)	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue (see page 508)
	HID_DESIGITEM (see page 549)	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	HID_GLOBALS (see page 550)	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
	HID_ITEM_INFO (see page 551)	HID Item Information This structure contains information about each Item of the report descriptor.
	HID_REPORT (see page 552)	HID Report details This structure contains information about each report exchanged with the device.
	HID_REPORTITEM (see page 553)	HID Report Details This structure contains information about each Report encountered in the report descriptor.
	HID_STRINGITEM (see page 554)	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	HID_TRANSFER_DATA (see page 555)	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE (see page 542) or EVENT_HID_WRITE_DONE (see page 546)).
	HID_USAGEITEM (see page 556)	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
	USB_HID_DEVICE_ID (see page 564)	HID Device ID Information This structure contains identification information about an attached device.
	USB_HID_DEVICE_RPT_INFO (see page 566)	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during th ecourse of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much inportance to the application.

	USB_HID_ITEM_LIST (see page 571)	List of Items This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrive data from the report received from the device. Application can also access these details to retreive the intended information incase provided interface function fail to do so.

Description

7.2.8.2.1 DEVICE_CLASS_HID Macro

File

usb_host_hid.h

C

```
#define DEVICE_CLASS_HID 0x03 /* HID Interface Class Code */
```

Description

HID Interface Class Code

7.2.8.2.2 DSC_HID Macro

File

usb_host_hid.h

C

```
#define DSC_HID 0x21 /* HID Descriptor Code */
```

Description

HID Descriptor Code

7.2.8.2.3 DSC_PHY Macro

File

usb_host_hid.h

C

```
#define DSC_PHY 0x23 /* Pysical Descriptor Code */
```

Description

Pysical Descriptor Code

7.2.8.2.4 DSC_RPT Macro

File

usb_host_hid.h

C

```
#define DSC_RPT 0x2200 /* Report Descriptor Code */
```

Description

Report Descriptor Code

7.2.8.2.5 EVENT_HID_ATTACH Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_ATTACH EVENT_HID_BASE + EVENT_HID_OFFSET + 7
```

Description

A HID device has attached. The returned data pointer points to a USB_HID_DEVICE_ID (see page 564) structure.

7.2.8.2.6 EVENT_HID_BAD_REPORT_DESCRIPTOR Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_BAD_REPORT_DESCRIPTOR EVENT_HID_BASE + EVENT_HID_OFFSET + 9
```

Description

There was a problem parsing the report descriptor of the attached device. Communication with the device is not allowed, and the device should be detached.

7.2.8.2.7 EVENT_HID_DETACH Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_DETACH EVENT_HID_BASE + EVENT_HID_OFFSET + 8
```

Description

A HID device has detached. The returned data pointer points to a byte with the previous address of the detached device.

7.2.8.2.8 EVENT_HID_NONE Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_NONE EVENT_HID_BASE + EVENT_HID_OFFSET + 0
```

Description

No event occured (NULL event)

7.2.8.2.9 EVENT_HID_OFFSET Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_OFFSET 0
```

Description

If the application has not defined an offset for HID events, set it to 0.

7.2.8.2.10 EVENT_HID_READ_DONE Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_READ_DONE EVENT_HID_BASE + EVENT_HID_OFFSET + 4
```

Description

#define EVENT_HID_TRANSFER EVENT_HID_BASE + EVENT_HID_OFFSET (see page 541) + 3 // Unused - value retained for legacy. A HID Read transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA (see page 555) structure, with information about the transfer.

7.2.8.2.11 EVENT_HID_RESET Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_RESET EVENT_HID_BASE + EVENT_HID_OFFSET + 6
```

Description

HID reset complete. The returned data pointer is NULL.

7.2.8.2.12 EVENT_HID_RESET_ERROR Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_RESET_ERROR EVENT_HID_BASE + EVENT_HID_OFFSET + 10
```

Description

An error occurred while trying to do a HID reset. The returned data pointer is NULL.

7.2.8.2.13 EVENT_HID_RPT_DESC_PARSED Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_RPT_DESC_PARSED EVENT_HID_BASE + EVENT_HID_OFFSET + 1
```

Description

A Report Descriptor has been parsed. The returned data pointer is NULL. The application must collect details, or simply return TRUE if the application is already aware of the data format.

7.2.8.2.14 EVENT_HID_WRITE_DONE Macro

File

usb_host_hid.h

C

```
#define EVENT_HID_WRITE_DONE EVENT_HID_BASE + EVENT_HID_OFFSET + 5
```

Description

A HID Write transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA (see page 555) structure, with information about the transfer.

7.2.8.2.15 HID_COLLECTION Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_COLLECTION {
  DWORD data;
  WORD usagePage;
  BYTE firstUsageItem;
  BYTE usageItems;
  BYTE firstReportItem;
  BYTE reportItems;
  BYTE parent;
  BYTE firstChild;
  BYTE nextSibling;
} HID_COLLECTION;
```

Members

Members	Description
DWORD data;	Collection raw data
WORD usagePage;	Usage page associated with current level of collection
BYTE firstUsageItem;	Index of First Usage Item in the current collection
BYTE usageItems;	Number of Usage Items in the current collection
BYTE firstReportItem;	Index of First report Item in the current collection
BYTE reportItems;	Number of report Items in the current collection
BYTE parent;	Index to Parent collection
BYTE firstChild;	Index to next child collection in the report descriptor
BYTE nextSibling;	Index to next child collection in the report descriptor

Description

HID Collection Details

This structure contains information about each collection encountered in the report descriptor.

7.2.8.2.16 HID_DATA_DETAILS Structure

File

usb_host_hid.h

C

```
typedef struct _HID_DATA_DETAILS {
  WORD reportLength;
  WORD reportID;
  BYTE bitOffset;
  BYTE bitLength;
  BYTE count;
  BYTE signExtend;
  BYTE interfaceNum;
} HID_DATA_DETAILS;
```

Members

Members	Description
WORD reportLength;	reportLength - the expected length of the parent report.
WORD reportID;	reportID - report ID - the first byte of the parent report.
BYTE bitOffset;	BitOffset - bit offset within the report.
BYTE bitLength;	bitlength - length of the data in bits.
BYTE count;	count - what's left of the message after this data.
BYTE signExtend;	extend - sign extend the data.
BYTE interfaceNum;	interfaceNum - informs HID layer about interface number.

Description

HID Data Details

This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue (see page 508)

7.2.8.2.17 HID_DESIGITEM Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_STRINGITEM {
  BOOL isRange;
  WORD index;
  WORD minimum;
  WORD maximum;
} HID_STRINGITEM, HID_DESIGITEM;
```

Members

Members	Description
BOOL isRange;	If range of String Item is valid
WORD index;	String index for a String descriptor; allows a string to be associated with a particular item or control
WORD minimum;	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap
WORD maximum;	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap

Description

HID String Item Details

This structure contains information about each Report encountered in the report descriptor.

7.2.8.2.18 HID_GLOBALS Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_GLOBALS {
  WORD usagePage;
  LONG logicalMinimum;
  LONG logicalMaximum;
  LONG physicalMinimum;
  LONG physicalMaximum;
  LONG unitExponent;
  LONG unit;
  WORD reportIndex;
  BYTE reportID;
  BYTE reportsize;
  BYTE reportCount;
} HID_GLOBALS;
```

Members

Members	Description
WORD usagePage;	Specifies current Usage Page
LONG logicalMinimum;	This is the minimum value that a variable or array item will report
LONG logicalMaximum;	This is the maximum value that a variable or array item will report
LONG physicalMinimum;	Minimum value for the physical extent of a variable item
LONG physicalMaximum;	Maximum value for the physical extent of a variable item
LONG unitExponent;	Value of the unit exponent in base 10
LONG unit;	Unit values
WORD reportIndex;	Conter to keep track of report being processed in the parser
BYTE reportID;	Report ID. All the reports are preceded by a single byte report ID
BYTE reportsize;	Size of current report in bytes
BYTE reportCount;	This field determines number of fields in the report

Description

HID Global Item Information

This structure contains information about each Global Item of the report descriptor.

7.2.8.2.19 HID_ITEM_INFO Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_ITEM_INFO {
  union {
    struct {
      BYTE ItemSize : 2;
      BYTE ItemType : 2;
      BYTE ItemTag : 4;
    }
    BYTE val;
  } ItemDetails;
  union {
    LONG sItemData;
    DWORD uItemData;
    BYTE bItemData[4];
  } Data;
} HID_ITEM_INFO;
```

Members

Members	Description
BYTE ItemSize : 2;	Numeric expression specifying size of data
BYTE ItemType : 2;	This field identifies type of item(Main, Global or Local)
BYTE ItemTag : 4;	This field specifies the function of the item
BYTE val;	to access the data in byte format
LONG sItemData;	Item Data is stored in signed format
DWORD uItemData;	Item Data is stored in unsigned format

Description

HID Item Information

This structure contains information about each Item of the report descriptor.

7.2.8.2.20 HID_REPORT Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_REPORT {
  WORD reportID;
  WORD inputBits;
  WORD outputBits;
  WORD featureBits;
} HID_REPORT;
```

Members

Members	Description
WORD reportID;	Report ID of the associated report
WORD inputBits;	If input report then length of report in bits
WORD outputBits;	If output report then length of report in bits
WORD featureBits;	If feature report then length of report in bits

Description

HID Report details

This structure contains information about each report exchanged with the device.

7.2.8.2.21 HID_REPORTITEM Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_REPORTITEM {
  HIDReportTypeEnum reportType;
  HID_GLOBALS globals;
  BYTE startBit;
  BYTE parent;
  DWORD dataModes;
  BYTE firstUsageItem;
  BYTE usageItems;
  BYTE firstStringItem;
  BYTE stringItems;
  BYTE firstDesignatorItem;
  BYTE designatorItems;
} HID_REPORTITEM;
```

Members

Members	Description
HIDReportTypeEnum reportType;	Type of Report Input/Output/Feature
HID_GLOBALS globals;	Stores all the global items associated with the current report
BYTE startBit;	Starting Bit Position of the report
BYTE parent;	Index of parent collection
DWORD dataModes;	this tells the data mode is array or not
BYTE firstUsageItem;	Index to first usage item related to the report
BYTE usageItems;	Number of usage items in the current report
BYTE firstStringItem;	Index to first srting item in the list
BYTE stringItems;	Number of string items in the current report
BYTE firstDesignatorItem;	Index to first designator item
BYTE designatorItems;	Number of designator items in the current report

Description

HID Report Details

This structure contains information about each Report encountered in the report descriptor.

7.2.8.2.22 HID_STRINGITEM Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_STRINGITEM {
  BOOL isRange;
  WORD index;
  WORD minimum;
  WORD maximum;
} HID_STRINGITEM, HID_DESIGITEM;
```

Members

Members	Description
BOOL isRange;	If range of String Item is valid
WORD index;	String index for a String descriptor; allows a string to be associated with a particular item or control
WORD minimum;	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap
WORD maximum;	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap

Description

HID String Item Details

This structure contains information about each Report encountered in the report descriptor.

7.2.8.2.23 HID_TRANSFER_DATA Structure

File

usb_host_hid.h

C

```
typedef struct _HID_TRANSFER_DATA {
  DWORD dataCount;
  BYTE bErrorCode;
} HID_TRANSFER_DATA;
```

Members

Members	Description
DWORD dataCount;	Count of bytes transferred.
BYTE bErrorCode;	Transfer error code.

Description

HID Transfer Information

This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE (see page 542) or EVENT_HID_WRITE_DONE (see page 546)).

7.2.8.2.24 HID_USAGEITEM Structure

File

usb_host_hid_parser.h

C

```
typedef struct _HID_USAGEITEM {
  BOOL isRange;
  WORD usagePage;
  WORD usage;
  WORD usageMinimum;
  WORD usageMaximum;
} HID_USAGEITEM;
```

Members

Members	Description
BOOL isRange;	True if Usage item has a valid MAX and MIN range
WORD usagePage;	Usage page ID asscociated with the Item
WORD usage;	Usage ID asscociated with the Item
WORD usageMinimum;	Defines the starting usage associated with an array or bitmap
WORD usageMaximum;	Defines the ending usage associated with an array or bitmap

Description

HID Report Details

This structure contains information about each Usage Item encountered in the report descriptor.

7.2.8.2.25 HIDReportTypeEnum Enumeration

File

usb_host_hid_parser.h

C

```
typedef enum {
  hidReportInput,
  hidReportOutput,
  hidReportFeature,
  hidReportUnknown
} HIDReportTypeEnum;
```

Description

This is type HIDReportTypeEnum.

7.2.8.2.26 USB_HID_CLASS_ERROR Macro

File

usb_host_hid.h

C

```
#define USB_HID_CLASS_ERROR USB_ERROR_CLASS_DEFINED
```

Section

HID Class Error Codes

7.2.8.2.27 USB_HID_COMMAND_FAILED Macro

File

usb_host_hid.h

C

```
#define USB_HID_COMMAND_FAILED (USB_HID_CLASS_ERROR | HID_COMMAND_FAILED) // Command failed
at the device.
```

Description

Command failed at the device.

7.2.8.2.28 USB_HID_COMMAND_PASSED Macro

File

usb_host_hid.h

C

```
#define USB_HID_COMMAND_PASSED USB_SUCCESS                    // Command was
successful.
```

Description

Command was successful.

7.2.8.2.29 USB_HID_DEVICE_BUSY Macro

File

usb_host_hid.h

C

```
#define USB_HID_DEVICE_BUSY (USB_HID_CLASS_ERROR | 0x04)                // A transfer is
currently in progress.
```

Description

A transfer is currently in progress.

7.2.8.2.30 USB_HID_DEVICE_DETACHED Macro

File

usb_host_hid.h

C

```
#define USB_HID_DEVICE_DETACHED 0x50    // Device is detached.
```

Description

Device is detached.

7.2.8.2.31 USB_HID_DEVICE_HOLDING Macro

File

usb_host_hid.h

C

```
#define USB_HID_DEVICE_HOLDING 0x54     // Device is holding due to error
```

Description

Device is holding due to error

7.2.8.2.32 USB_HID_DEVICE_ID Structure

File

usb_host_hid.h

C

```
typedef struct _USB_HID_DEVICE_ID {
  WORD vid;
  WORD pid;
  BYTE deviceAddress;
  BYTE clientDriverID;
} USB_HID_DEVICE_ID;
```

Members

Members	Description
WORD vid;	Vendor ID of the device
WORD pid;	Product ID of the device
BYTE deviceAddress;	Address of the device on the USB
BYTE clientDriverID;	Client driver ID for device requests

Description

HID Device ID Information

This structure contains identification information about an attached device.

7.2.8.2.33 USB_HID_DEVICE_NOT_FOUND Macro

File

usb_host_hid.h

C

```
#define USB_HID_DEVICE_NOT_FOUND (USB_HID_CLASS_ERROR | 0x03)              // Device with
the specified address is not available.
```

Description

Device with the specified address is not available.

7.2.8.2.34 USB_HID_DEVICE_RPT_INFO Structure

File

usb_host_hid_parser.h

C

```
typedef struct _USB_HID_DEVICE_RPT_INFO {
  WORD reportPollingRate;
  BYTE interfaceNumber;
  BOOL haveDesignatorMax;
  BOOL haveDesignatorMin;
  BOOL haveStringMax;
  BOOL haveStringMin;
  BOOL haveUsageMax;
  BOOL haveUsageMin;
  WORD designatorMaximum;
  WORD designatorMinimum;
  WORD designatorRanges;
  WORD designators;
  WORD rangeUsagePage;
  WORD stringMaximum;
  WORD stringMinimum;
  WORD stringRanges;
  WORD usageMaximum;
  WORD usageMinimum;
  WORD usageRanges;
  BYTE collectionNesting;
  BYTE collections;
  BYTE designatorItems;
  BYTE firstUsageItem;
  BYTE firstDesignatorItem;
  BYTE firstStringItem;
  BYTE globalsNesting;
  BYTE maxCollectionNesting;
  BYTE maxGlobalsNesting;
  BYTE parent;
  BYTE reportItems;
  BYTE reports;
  BYTE sibling;
  BYTE stringItems;
  BYTE strings;
  BYTE usageItems;
  BYTE usages;
  HID_GLOBALS globals;
} USB_HID_DEVICE_RPT_INFO;
```

Members

Members	Description
WORD reportPollingRate;	This stores the pollrate for the input report. Application can use this to decide the rate of transfer
BYTE interfaceNumber;	This stores the interface number for the current report descriptor
BOOL haveDesignatorMax;	True if report descriptor has a valid Designator Max
BOOL haveDesignatorMin;	True if report descriptor has a valid Designator Min
BOOL haveStringMax;	True if report descriptor has a valid String Max
BOOL haveStringMin;	True if report descriptor has a valid String Min
BOOL haveUsageMax;	True if report descriptor has a valid Usage Max
BOOL haveUsageMin;	True if report descriptor has a valid Usage Min
WORD designatorMaximum;	Last designator max value
WORD designatorMinimum;	Last designator min value
WORD designatorRanges;	Last designator range
WORD designators;	This tells toatal number of designator items
WORD rangeUsagePage;	current usage page during parsing

WORD stringMaximum;	current string maximum
WORD stringMinimum;	current string minimum
WORD stringRanges;	current string ranges
WORD usageMaximum;	current usage maximum
WORD usageMinimum;	current usage minimum
WORD usageRanges;	current usage ranges
BYTE collectionNesting;	this number tells depth of collection nesting
BYTE collections;	total number of collections
BYTE designatorItems;	total number of designator items
BYTE firstUsageItem;	index of first usage item for the current collection
BYTE firstDesignatorItem;	index of first designator item for the current collection
BYTE firstStringItem;	index of first string item for the current collection
BYTE globalsNesting;	On encountering every PUSH item , this is incremented , keep track of current depth of Globals
BYTE maxCollectionNesting;	Maximum depth of collections
BYTE maxGlobalsNesting;	Maximum depth of Globals
BYTE parent;	Parent collection
BYTE reportItems;	total number of report items
BYTE reports;	total number of reports
BYTE sibling;	current sibling collection
BYTE stringItems;	total number of string items , used to index the array of strings
BYTE strings;	total sumber of strings
BYTE usageItems;	total number of usage items , used to index the array of usage
BYTE usages;	total sumber of usages
HID_GLOBALS globals;	holds cuurent globals items

Description

Report Descriptor Information

This structure contains top level information of the report descriptor. This information is important and is used to understand the information during th ecourse of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much inportance to the application.

7.2.8.2.35 USB_HID_ILLEGAL_REQUEST Macro

File

usb_host_hid.h

C

```
#define USB_HID_ILLEGAL_REQUEST (USB_HID_CLASS_ERROR | 0x0B) // Cannot perform requested
operation.
```

Description

Cannot perform requested operation.

7.2.8.2.36 USB_HID_INITIALIZING Macro

File

usb_host_hid.h

C

```
#define USB_HID_INITIALIZING 0x51    // Device is initializing.
```

Description

Device is initializing.

7.2.8.2.37 USB_HID_INTERFACE_ERROR Macro

File

usb_host_hid.h

C

```
#define USB_HID_INTERFACE_ERROR (USB_HID_CLASS_ERROR | 0x06)              // The interface
layer cannot support the device.
```

Description

The interface layer cannot support the device.

7.2.8.2.38 USB_HID_ITEM_LIST Structure

File

usb_host_hid_parser.h

C

```
typedef struct _USB_HID_ITEM_LIST {
  HID_COLLECTION * collectionList;
  HID_DESIGITEM * designatorItemList;
  HID_GLOBALS * globalsStack;
  HID_REPORTITEM * reportItemList;
  HID_REPORT * reportList;
  HID_STRINGITEM * stringItemList;
  HID_USAGEITEM * usageItemList;
  BYTE * collectionStack;
} USB_HID_ITEM_LIST;
```

Members

Members	Description
HID_COLLECTION * collectionList;	List of collections, see HID_COLLECTION (see page 547) for details in the structure
HID_DESIGITEM * designatorItemList;	List of designator Items, see HID_DESIGITEM (see page 549) for details in the structure
HID_GLOBALS * globalsStack;	List of global Items, see HID_GLOBALS (see page 550) for details in the structure
HID_REPORTITEM * reportItemList;	List of report Items, see HID_REPORTITEM (see page 553) for details in the structure
HID_REPORT * reportList;	List of reports , see HID_REPORT (see page 552) for details in the structure
HID_STRINGITEM * stringItemList;	List of string item , see HID_STRINGITEM (see page 554) for details in the structure
HID_USAGEITEM * usageItemList;	List of Usage item , see HID_USAGEITEM (see page 556) for details in the structure
BYTE * collectionStack;	stores the array of parents ids for the collection

Description

List of Items

This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrive data from the report received from the device. Application can also access these details to retreive the intended information incase provided interface function fail to do so.

7.2.8.2.39 USB_HID_NO_REPORT_DESCRIPTOR Macro

File

usb_host_hid.h

C

```
#define USB_HID_NO_REPORT_DESCRIPTOR (USB_HID_CLASS_ERROR | 0x05)          // No
report descriptor found
```

Description

No report descriptor found

7.2.8.2.40 USB_HID_NORMAL_RUNNING Macro

File

usb_host_hid.h

C

```
#define USB_HID_NORMAL_RUNNING 0x53    // Device is running and available for data
transfers.
```

Description

Device is running and available for data transfers.

7.2.8.2.41 USB_HID_PHASE_ERROR Macro

File

usb_host_hid.h

C

```
#define USB_HID_PHASE_ERROR (USB_HID_CLASS_ERROR | HID_PHASE_ERROR)    // Command had a
phase error at the device.
```

Description

Command had a phase error at the device.

7.2.8.2.42 USB_HID_REPORT_DESCRIPTOR_BAD Macro

File

usb_host_hid.h

C

```
#define USB_HID_REPORT_DESCRIPTOR_BAD (USB_HID_CLASS_ERROR | 0x07)            // Report
Descriptor for not proper
```

Description

Report Descriptor for not proper

7.2.8.2.43 USB_HID_RESET_ERROR Macro

File

usb_host_hid.h

C

```
#define USB_HID_RESET_ERROR (USB_HID_CLASS_ERROR | 0x0A) // An error occurred while
resetting the device.
```

Description

An error occurred while resetting the device.

7.2.8.2.44 USB_HID_RESETTING_DEVICE Macro

File

usb_host_hid.h

C

```
#define USB_HID_RESETTING_DEVICE 0x55    // Device is being reset.
```

Description

Device is being reset.

7.2.8.2.45 USB_HID_RPT_DESC_ERROR Enumeration

File

usb_host_hid_parser.h

C

```
typedef enum {
  HID_ERR = 0,
  HID_ERR_NotEnoughMemory,
  HID_ERR_NullPointer,
  HID_ERR_UnexpectedEndCollection,
  HID_ERR_UnexpectedPop,
  HID_ERR_MissingEndCollection,
  HID_ERR_MissingTopLevelCollection,
  HID_ERR_NoReports,
  HID_ERR_UnmatchedUsageRange,
  HID_ERR_UnmatchedStringRange,
  HID_ERR_UnmatchedDesignatorRange,
  HID_ERR_UnexpectedEndOfDescriptor,
  HID_ERR_BadLogicalMin,
  HID_ERR_BadLogicalMax,
  HID_ERR_BadLogical,
  HID_ERR_ZeroReportSize,
  HID_ERR_ZeroReportID,
  HID_ERR_ZeroReportCount,
  HID_ERR_BadUsageRangePage,
  HID_ERR_BadUsageRange
} USB_HID_RPT_DESC_ERROR;
```

Members

Members	Description
HID_ERR = 0	No error
HID_ERR_NotEnoughMemory	If not enough Heap can be allocated, make sure sufficient dynamic memory is aloocated for the parser
HID_ERR_NullPointer	Pointer to report descriptor is NULL
HID_ERR_UnexpectedEndCollection	End of collection not expected
HID_ERR_UnexpectedPop	POP not expected
HID_ERR_MissingEndCollection	No end of collection found
HID_ERR_MissingTopLevelCollection	Atleast one collection must be present
HID_ERR_NoReports	atlest one report must be present
HID_ERR_UnmatchedUsageRange	Either Minimum or Maximum for usage range missing
HID_ERR_UnmatchedStringRange	Either Minimum or Maximum for string range missing
HID_ERR_UnmatchedDesignatorRange	Either Minimum or Maximum for designator range missing
HID_ERR_UnexpectedEndOfDescriptor	Report descriptor not formatted properly
HID_ERR_BadLogicalMin	Logical Min greater than report size
HID_ERR_BadLogicalMax	Logical Max greater than report size
HID_ERR_BadLogical	If logical Min is greater than Max
HID_ERR_ZeroReportSize	Report size is zero
HID_ERR_ZeroReportID	report ID is zero
HID_ERR_ZeroReportCount	Number of reports is zero
HID_ERR_BadUsageRangePage	Bad Usage page range
HID_ERR_BadUsageRange	Bad Usage range

Description

HID parser error codes

This enumerates the error encountered during the parsing of report descriptor. In case of any error parsing is sttopped and the error is flagged. Device is not attched successfully.

7.2.8.2.46 USB_PROCESSING_REPORT_DESCRIPTOR Macro

File

usb_host_hid.h

C

```
#define USB_PROCESSING_REPORT_DESCRIPTOR 0x52    // Parser is processing report descriptor.
```

Description

Parser is processing report descriptor.

7.2.9 Mass Storage Client Driver

This client driver provides USB Embedded Host support for mass storage devices.

Description

This client driver provides USB Embedded Host support for mass storage devices. Mass storage devices use USB Bulk transfers to efficiently transfer large amounts of data. Bulk transfers may utilize all remaining bandwidth on the bus after all of the Control, Interrupt, and Isochronous transfers for the frame have completed. The exact amount of time required for a bulk transfer will depend on the amount of other traffic that is on the bus. Therefore, Bulk transfers should be used only for non-time critical operations.

This implementation of the Mass Storage Class provides support for the Bulk Only Transport.

See *AN1142 - USB Mass Storage Class on an Embedded Host* for more information about the Mass Storage Class and this client driver.

7.2.9.1 Interface Routines

Functions

	Name	Description
	USBHostMSDDeviceStatus (see page 581)	This function determines the status of a mass storage device.
	USBHostMSDEventHandler (see page 582)	This function is the event handler for this client driver.
	USBHostMSDInitialize (see page 583)	This function is the initialization routine for this client driver.
	USBHostMSDResetDevice (see page 585)	This function starts a bulk-only mass storage reset.
	USBHostMSDSCSIEventHandler (see page 586)	This function is called when various events occur in the USB Host Mass Storage client driver.
	USBHostMSDSCSIInitialize (see page 587)	This function is called when a USB Mass Storage device is being enumerated.
	USBHostMSDSCSISectorRead (see page 588)	This function reads one sector.
	USBHostMSDSCSISectorWrite (see page 589)	This function writes one sector.
	USBHostMSDTerminateTransfer (see page 590)	This function terminates a mass storage transfer.
	USBHostMSDTransfer (see page 591)	This function starts a mass storage transfer.
	USBHostMSDTransferIsComplete (see page 592)	This function indicates whether or not the last transfer is complete.

Macros

	Name	Description
	USBHostMSDRead (see page 584)	This function starts a mass storage read, utilizing the function USBHostMSDTransfer (see page 591)();
	USBHostMSDWrite (see page 593)	This function starts a mass storage write, utilizing the function USBHostMSDTransfer (see page 591)();

Description

7.2.9.1.1 USBHostMSDDeviceStatus Function

File

usb_host_msd.h

C

```
BYTE USBHostMSDDeviceStatus(
    BYTE deviceAddress
);
```

Description

This function determines the status of a mass storage device.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	address of device to query

Return Values

Return Values	Description
USB_MSD_DEVICE_NOT_FOUND (see page 618)	Illegal device address, or the device is not an MSD
USB_MSD_INITIALIZING (see page 622)	MSD is attached and in the process of initializing
USB_MSD_NORMAL_RUNNING (see page 625)	MSD is in normal running mode
USB_MSD_RESETTING_DEVICE (see page 629)	MSD is resetting
USB_MSD_DEVICE_DETACHED (see page 617)	MSD detached. Should not occur
USB_MSD_ERROR_STATE (see page 620)	MSD is holding due to an error. No communication is allowed.
Other	Return codes from USBHostDeviceStatus (see page 295)() will also be returned if the device is in the process of enumerating.

Function

BYTE USBHostMSDDeviceStatus(BYTE deviceAddress)

7.2.9.1.2 USBHostMSDEventHandler Function

This function is the event handler for this client driver.

File

usb_host_msd.h

C

```
BOOL USBHostMSDEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
WORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostMSDEventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.9.1.3 USBHostMSDInitialize Function

This function is the initialization routine for this client driver.

File

usb_host_msd.h

C

```
BOOL USBHostMSDInitialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated. For a mass storage device, we need to make sure that we have room for a new device, and that the device has at least one bulk IN and one bulk OUT endpoint.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	ID to send when issuing a Device Request via USBHostSendDeviceRequest(), USBHostSetDeviceConfiguration (see page 305)(), or USBHostSetDeviceInterface().

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostMSDInitialize(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.9.1.4 USBHostMSDRead Macro

File

usb_host_msd.h

C

```
#define USBHostMSDRead(
deviceAddress,deviceLUN,commandBlock,commandBlockLength,data,dataLength ) \
        USBHostMSDTransfer( deviceAddress, deviceLUN, 1, commandBlock, commandBlockLength,
data, dataLength )
```

Description

This function starts a mass storage read, utilizing the function USBHostMSDTransfer (see page 591)();

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE deviceLUN	Device LUN to access
BYTE *commandBlock	Pointer to the command block for the CBW
BYTE commandBlockLength	Length of the command block
BYTE *data	Pointer to the data buffer
DWORD dataLength	Byte size of the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_DEVICE_BUSY (see page 616)	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN (see page 623)	Specified LUN does not exist

Function

BYTE USBHostMSDRead(BYTE deviceAddress, BYTE deviceLUN, BYTE *commandBlock,

BYTE commandBlockLength, BYTE *data, DWORD dataLength);

7.2.9.1.5 USBHostMSDResetDevice Function

This function starts a bulk-only mass storage reset.

File

usb_host_msd.h

C

```
BYTE USBHostMSDResetDevice(
    BYTE deviceAddress
);
```

Description

This function starts a bulk-only mass storage reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Return Values

Return Values	Description
USB_SUCCESS	Reset started
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_ILLEGAL_REQUEST (see page 621)	Device is in an illegal state for reset

Function

BYTE USBHostMSDResetDevice(BYTE deviceAddress)

7.2.9.1.6 USBHostMSDSCSIEventHandler Function

File

usb_host_msd_scsi.h

C

```
BOOL USBHostMSDSCSIEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This function is called when various events occur in the USB Host Mass Storage client driver.

Remarks

None

Preconditions

The device has been initialized.

Parameters

Parameters	Description
BYTE address	Address of the device
USB_EVENT event	Event that has occurred
void *data	Pointer to data pertinent to the event
DWORD size	Size of the data

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

BOOL USBHostMSDSCSIEventHandler(BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.9.1.7 USBHostMSDSCSIInitialize Function

File

usb_host_msd_scsi.h

C

```
BOOL USBHostMSDSCSIInitialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This function is called when a USB Mass Storage device is being enumerated.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of the new device
DWORD flags	Initialization flags
BYTE clientDriverID	ID for this layer. Not used by the media interface layer.

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

BOOL USBHostMSDSCSIInitialize(BYTE address, DWORD flags, BYTE clientDriverID)

7.2.9.1.8 USBHostMSDSCSISectorRead Function

This function reads one sector.

File

usb_host_msd_scsi.h

C

```
BYTE USBHostMSDSCSISectorRead(
    DWORD sectorAddress,
    BYTE * dataBuffer
);
```

Description

This function uses the SCSI command READ10 to read one sector. The size of the sector was determined in the USBHostMSDSCSIMediaInitialize() function. The data is stored in the application buffer.

Remarks

The READ10 command block is as follows:

```
    Byte/Bit    7       6       5       4       3       2       1       0
       0                    Operation Code (0x28)
       1        [    RDPROTECT      ]  DPO     FUA      -     FUA_NV    -
       2        [ (MSB)
       3                        Logical Block Address
       4
       5                                                            (LSB) ]
       6        [         -         ][          Group Number              ]
       7        [ (MSB)         Transfer Length
       8                                                            (LSB) ]
       9        [                    Control                              ]
```

Preconditions

None

Parameters

Parameters	Description
DWORD sectorAddress	address of sector to read
BYTE *dataBuffer	buffer to store data

Return Values

Return Values	Description
TRUE	read performed successfully
FALSE	read was not successful

Function

BYTE USBHostMSDSCSISectorRead(DWORD sectorAddress, BYTE *dataBuffer)

7.2.9.1.9 USBHostMSDSCSISectorWrite Function

This function writes one sector.

File

usb_host_msd_scsi.h

C

```
BYTE USBHostMSDSCSISectorWrite(
    DWORD sectorAddress,
    BYTE * dataBuffer,
    BYTE allowWriteToZero
);
```

Description

This function uses the SCSI command WRITE10 to write one sector. The size of the sector was determined in the USBHostMSDSCSIMediaInitialize() function. The data is read from the application buffer.

Remarks

To follow convention, this function blocks until the write is complete.

The WRITE10 command block is as follows:

```
Byte/Bit    7       6       5       4       3       2       1       0
   0                    Operation Code (0x2A)
   1        [    WRPROTECT      ]  DPO     FUA      -     FUA_NV    -
   2        [ (MSB)
   3                        Logical Block Address
   4
   5                                                          (LSB) ]
   6        [         -         ][          Group Number            ]
   7        [ (MSB)         Transfer Length
   8                                                          (LSB) ]
   9        [                    Control                            ]
```

Preconditions

None

Parameters

Parameters	Description
DWORD sectorAddress	address of sector to write
BYTE *dataBuffer	buffer with application data
BYTE allowWriteToZero	If a write to sector 0 is allowed.

Return Values

Return Values	Description
TRUE	write performed successfully
FALSE	write was not successful

Function

BYTE USBHostMSDSCSISectorWrite(DWORD sectorAddress, BYTE *dataBuffer, BYTE allowWriteToZero)

7.2.9.1.10 USBHostMSDTerminateTransfer Function

File

usb_host_msd.h

C

```
void USBHostMSDTerminateTransfer(
    BYTE deviceAddress
);
```

Returns

None

Description

This function terminates a mass storage transfer.

Remarks

After executing this function, the application may have to reset the device in order for the device to continue working properly.

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address

Function

void USBHostMSDTerminateTransfer(BYTE deviceAddress)

7.2.9.1.11 USBHostMSDTransfer Function

This function starts a mass storage transfer.

File

usb_host_msd.h

C

```
BYTE USBHostMSDTransfer(
    BYTE deviceAddress,
    BYTE deviceLUN,
    BYTE direction,
    BYTE * commandBlock,
    BYTE commandBlockLength,
    BYTE * data,
    DWORD dataLength
);
```

Description

This function starts a mass storage transfer. Usually, applications will probably utilize a read/write wrapper to access this function.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE deviceLUN	Device LUN to access
BYTE direction	1=read, 0=write
BYTE *commandBlock	Pointer to the command block for the CBW
BYTE commandBlockLength	Length of the command block
BYTE *data	Pointer to the data buffer
DWORD dataLength	Byte size of the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_DEVICE_BUSY (see page 616)	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN (see page 623)	Specified LUN does not exist

Function

BYTE USBHostMSDTransfer(BYTE deviceAddress, BYTE deviceLUN,

BYTE direction, BYTE *commandBlock, BYTE commandBlockLength,

BYTE *data, DWORD dataLength)

7.2.9.1.12 USBHostMSDTransferIsComplete Function

This function indicates whether or not the last transfer is complete.

File

usb_host_msd.h

C

```
BOOL USBHostMSDTransferIsComplete(
    BYTE deviceAddress,
    BYTE * errorCode,
    DWORD * byteCount
);
```

Description

This function indicates whether or not the last transfer is complete. If the functions returns TRUE, the returned byte count and error code are valid. Since only one transfer can be performed at once and only one endpoint can be used, we only need to know the device address.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE *errorCode	Error code from last transfer
DWORD *byteCount	Number of bytes transferred

Return Values

Return Values	Description
TRUE	Transfer is complete, errorCode is valid
FALSE	Transfer is not complete, errorCode is not valid

Function

BOOL USBHostMSDTransferIsComplete(BYTE deviceAddress,

BYTE *errorCode, DWORD *byteCount)

7.2.9.1.13 USBHostMSDWrite Macro

File

usb_host_msd.h

C

```
#define USBHostMSDWrite(
deviceAddress,deviceLUN,commandBlock,commandBlockLength,data,dataLength ) \
        USBHostMSDTransfer( deviceAddress, deviceLUN, 0, commandBlock, commandBlockLength,
data, dataLength )
```

Description

This function starts a mass storage write, utilizing the function USBHostMSDTransfer (see page 591)();

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE deviceAddress	Device address
BYTE deviceLUN	Device LUN to access
BYTE *commandBlock	Pointer to the command block for the CBW
BYTE commandBlockLength	Length of the command block
BYTE *data	Pointer to the data buffer
DWORD dataLength	Byte size of the data buffer

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND (see page 618)	No device with specified address
USB_MSD_DEVICE_BUSY (see page 616)	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN (see page 623)	Specified LUN does not exist

Function

BYTE USBHostMSDWrite(BYTE deviceAddress, BYTE deviceLUN, BYTE *commandBlock,

BYTE commandBlockLength, BYTE *data, DWORD dataLength);

7.2.9.2 Data Types and Constants

Macros

	Name	Description
	DEVICE_CLASS_MASS_STORAGE (see page 596)	Class code for Mass Storage.
	DEVICE_INTERFACE_PROTOCOL_BULK_ONLY (see page 597)	Protocol code for Bulk-only mass storage.
	DEVICE_SUBCLASS_CD_DVD (see page 598)	SubClass code for a CD/DVD drive (not supported).
	DEVICE_SUBCLASS_FLOPPY_INTERFACE (see page 599)	SubClass code for a floppy disk interface (not supported).
	DEVICE_SUBCLASS_RBC (see page 600)	SubClass code for Reduced Block Commands (not supported).
	DEVICE_SUBCLASS_REMOVABLE (see page 601)	SubClass code for removable media (not supported).
	DEVICE_SUBCLASS_SCSI (see page 602)	SubClass code for a SCSI interface device (supported).
	DEVICE_SUBCLASS_TAPE_DRIVE (see page 603)	SubClass code for a tape drive (not supported).
	EVENT_MSD_MAX_LUN (see page 604)	Set maximum LUN for the device
	EVENT_MSD_NONE (see page 605)	No event occured (NULL event)
	EVENT_MSD_OFFSET (see page 606)	If the application has not defined an offset for MSD events, set it to 0.
	EVENT_MSD_RESET (see page 607)	MSD reset complete
	EVENT_MSD_TRANSFER (see page 608)	A MSD transfer has completed
	MSD_COMMAND_FAILED (see page 609)	Transfer failed. Returned in dCSWStatus.
	MSD_COMMAND_PASSED (see page 610)	Transfer was successful. Returned in dCSWStatus.
	MSD_PHASE_ERROR (see page 611)	Transfer phase error. Returned in dCSWStatus.
	USB_MSD_CBW_ERROR (see page 612)	The CBW was not transferred successfully.
	USB_MSD_COMMAND_FAILED (see page 613)	Command failed at the device.
	USB_MSD_COMMAND_PASSED (see page 614)	Command was successful.
	USB_MSD_CSW_ERROR (see page 615)	The CSW was not transferred successfully.
	USB_MSD_DEVICE_BUSY (see page 616)	A transfer is currently in progress.
	USB_MSD_DEVICE_DETACHED (see page 617)	Device is detached.
	USB_MSD_DEVICE_NOT_FOUND (see page 618)	Device with the specified address is not available.
	USB_MSD_ERROR (see page 619)	Error code offset.
	USB_MSD_ERROR_STATE (see page 620)	Device is holding due to a MSD error.
	USB_MSD_ILLEGAL_REQUEST (see page 621)	Cannot perform requested operation.
	USB_MSD_INITIALIZING (see page 622)	Device is initializing.
	USB_MSD_INVALID_LUN (see page 623)	Invalid LUN specified.
	USB_MSD_MEDIA_INTERFACE_ERROR (see page 624)	The media interface layer cannot support the device.
	USB_MSD_NORMAL_RUNNING (see page 625)	Device is running and available for data transfers.
	USB_MSD_OUT_OF_MEMORY (see page 626)	No dynamic memory is available.
	USB_MSD_PHASE_ERROR (see page 627)	Command had a phase error at the device.
	USB_MSD_RESET_ERROR (see page 628)	An error occurred while resetting the device.

	USB_MSD_RESETTING_DEVICE (see page 629)	Device is being reset.

Description

7.2.9.2.1 DEVICE_CLASS_MASS_STORAGE Macro

File

usb_host_msd.h

C

```
#define DEVICE_CLASS_MASS_STORAGE 0x08    // Class code for Mass Storage.
```

Description

Class code for Mass Storage.

7.2.9.2.2 DEVICE_INTERFACE_PROTOCOL_BULK_ONLY Macro

File

usb_host_msd.h

C

```
#define DEVICE_INTERFACE_PROTOCOL_BULK_ONLY 0x50    // Protocol code for Bulk-only mass
storage.
```

Description

Protocol code for Bulk-only mass storage.

7.2.9.2.3 DEVICE_SUBCLASS_CD_DVD Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_CD_DVD 0x02     // SubClass code for a CD/DVD drive (not supported).
```

Description

SubClass code for a CD/DVD drive (not supported).

7.2.9.2.4 DEVICE_SUBCLASS_FLOPPY_INTERFACE Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_FLOPPY_INTERFACE 0x04    // SubClass code for a floppy disk
interface (not supported).
```

Description

SubClass code for a floppy disk interface (not supported).

7.2.9.2.5 DEVICE_SUBCLASS_RBC Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_RBC 0x01    // SubClass code for Reduced Block Commands (not
supported).
```

Description

SubClass code for Reduced Block Commands (not supported).

7.2.9.2.6 DEVICE_SUBCLASS_REMOVABLE Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_REMOVABLE 0x05    // SubClass code for removable media (not
supported).
```

Description

SubClass code for removable media (not supported).

7.2.9.2.7 DEVICE_SUBCLASS_SCSI Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_SCSI 0x06    // SubClass code for a SCSI interface device
(supported).
```

Description

SubClass code for a SCSI interface device (supported).

7.2.9.2.8 DEVICE_SUBCLASS_TAPE_DRIVE Macro

File

usb_host_msd.h

C

```
#define DEVICE_SUBCLASS_TAPE_DRIVE 0x03    // SubClass code for a tape drive (not
supported).
```

Description

SubClass code for a tape drive (not supported).

7.2.9.2.9 EVENT_MSD_MAX_LUN Macro

File

usb_host_msd.h

C

```
#define EVENT_MSD_MAX_LUN EVENT_MSD_BASE + EVENT_MSD_OFFSET + 3   // Set maximum LUN for
the device
```

Description

Set maximum LUN for the device

7.2.9.2.10 EVENT_MSD_NONE Macro

File

usb_host_msd.h

C

```
#define EVENT_MSD_NONE EVENT_MSD_BASE + EVENT_MSD_OFFSET + 0   // No event occured (NULL
event)
```

Description

No event occured (NULL event)

7.2.9.2.11 EVENT_MSD_OFFSET Macro

File

usb_host_msd.h

C

```
#define EVENT_MSD_OFFSET 0
```

Description

If the application has not defined an offset for MSD events, set it to 0.

7.2.9.2.12 EVENT_MSD_RESET Macro

File

usb_host_msd.h

C

```
#define EVENT_MSD_RESET EVENT_MSD_BASE + EVENT_MSD_OFFSET + 2   // MSD reset complete
```

Description

MSD reset complete

7.2.9.2.13 EVENT_MSD_TRANSFER Macro

File

usb_host_msd.h

C

```
#define EVENT_MSD_TRANSFER EVENT_MSD_BASE + EVENT_MSD_OFFSET + 1   // A MSD transfer has
completed
```

Description

A MSD transfer has completed

7.2.9.2.14 MSD_COMMAND_FAILED Macro

File

usb_host_msd.h

C

```
#define MSD_COMMAND_FAILED 0x01    // Transfer failed. Returned in dCSWStatus.
```

Description

Transfer failed. Returned in dCSWStatus.

7.2.9.2.15 MSD_COMMAND_PASSED Macro

File

usb_host_msd.h

C

```
#define MSD_COMMAND_PASSED 0x00    // Transfer was successful. Returned in dCSWStatus.
```

Description

Transfer was successful. Returned in dCSWStatus.

7.2.9.2.16 MSD_PHASE_ERROR Macro

File

usb_host_msd.h

C

```
#define MSD_PHASE_ERROR 0x02     // Transfer phase error. Returned in dCSWStatus.
```

Description

Transfer phase error. Returned in dCSWStatus.

7.2.9.2.17 USB_MSD_CBW_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_CBW_ERROR (USB_MSD_ERROR | 0x04)            // The CBW was not 
transferred successfully.
```

Description

The CBW was not transferred successfully.

7.2.9.2.18 USB_MSD_COMMAND_FAILED Macro

File

usb_host_msd.h

C

```
#define USB_MSD_COMMAND_FAILED (USB_MSD_ERROR | MSD_COMMAND_FAILED)// Command failed at the
device.
```

Description

Command failed at the device.

7.2.9.2.19 USB_MSD_COMMAND_PASSED Macro

File

usb_host_msd.h

C

```
#define USB_MSD_COMMAND_PASSED USB_SUCCESS                          // Command was
successful.
```

Description

Command was successful.

7.2.9.2.20 USB_MSD_CSW_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_CSW_ERROR (USB_MSD_ERROR | 0x05)              // The CSW was not
transferred successfully.
```

Description

The CSW was not transferred successfully.

7.2.9.2.21 USB_MSD_DEVICE_BUSY Macro

File

usb_host_msd.h

C

```
#define USB_MSD_DEVICE_BUSY (USB_MSD_ERROR | 0x07)            // A transfer is currently
in progress.
```

Description

A transfer is currently in progress.

7.2.9.2.22 USB_MSD_DEVICE_DETACHED Macro

File

usb_host_msd.h

C

```
#define USB_MSD_DEVICE_DETACHED 0x50    // Device is detached.
```

Description

Device is detached.

7.2.9.2.23 USB_MSD_DEVICE_NOT_FOUND Macro

File

usb_host_msd.h

C

```
#define USB_MSD_DEVICE_NOT_FOUND (USB_MSD_ERROR | 0x06)              // Device with the
specified address is not available.
```

Description

Device with the specified address is not available.

7.2.9.2.24 USB_MSD_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_ERROR USB_ERROR_CLASS_DEFINED            // Error code offset.
```

Description

Error code offset.

7.2.9.2.25 USB_MSD_ERROR_STATE Macro

File

usb_host_msd.h

C

```
#define USB_MSD_ERROR_STATE 0x55     // Device is holding due to a MSD error.
```

Description

Device is holding due to a MSD error.

7.2.9.2.26 USB_MSD_ILLEGAL_REQUEST Macro

File

usb_host_msd.h

C

```
#define USB_MSD_ILLEGAL_REQUEST (USB_MSD_ERROR | 0x0B)              // Cannot perform
requested operation.
```

Description

Cannot perform requested operation.

7.2.9.2.27 USB_MSD_INITIALIZING Macro

File

usb_host_msd.h

C

```
#define USB_MSD_INITIALIZING 0x51    // Device is initializing.
```

Description

Device is initializing.

7.2.9.2.28 USB_MSD_INVALID_LUN Macro

File

usb_host_msd.h

C

```
#define USB_MSD_INVALID_LUN (USB_MSD_ERROR | 0x08)              // Invalid LUN specified.
```

Description

Invalid LUN specified.

7.2.9.2.29 USB_MSD_MEDIA_INTERFACE_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_MEDIA_INTERFACE_ERROR (USB_MSD_ERROR | 0x09)            // The media
interface layer cannot support the device.
```

Description

The media interface layer cannot support the device.

7.2.9.2.30 USB_MSD_NORMAL_RUNNING Macro

File

usb_host_msd.h

C

```
#define USB_MSD_NORMAL_RUNNING 0x52    // Device is running and available for data
transfers.
```

Description

Device is running and available for data transfers.

7.2.9.2.31 USB_MSD_OUT_OF_MEMORY Macro

File

usb_host_msd.h

C

```
#define USB_MSD_OUT_OF_MEMORY (USB_MSD_ERROR | 0x03)            // No dynamic memory is
available.
```

Description

No dynamic memory is available.

7.2.9.2.32 USB_MSD_PHASE_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_PHASE_ERROR (USB_MSD_ERROR | MSD_PHASE_ERROR)   // Command had a phase
error at the device.
```

Description

Command had a phase error at the device.

7.2.9.2.33 USB_MSD_RESET_ERROR Macro

File

usb_host_msd.h

C

```
#define USB_MSD_RESET_ERROR (USB_MSD_ERROR | 0x0A)              // An error occurred while resetting the device.
```

Description

An error occurred while resetting the device.

7.2.9.2.34 USB_MSD_RESETTING_DEVICE Macro

File

usb_host_msd.h

C

```
#define USB_MSD_RESETTING_DEVICE 0x53     // Device is being reset.
```

Description

Device is being reset.

7.2.10 Printer Client Driver

This client driver provides USB Embedded Host support for printer devices.

Description

Many USB printers utilize the USB Printer Class to communicate with a USB Host. This class defines the USB transfer type, the endpoint structure,a device requests that can be performed. The actual commands sent to the printer, however, are dictated by the printer language used by the particular printer.

Many different printer languages are utilized by the wide variety of printers on the market. Typically, low end printers receive printer-specific binary data, utilizing the processing power of the USB Host to perform all of the complex calculations required to translate text and graphics to a simple binary representation. This works well when a PC is the USB Host, but it is not conducive to an embedded application with limited resources.

Many printers on the market use a command based printer language, relying on the printer itself to interpret commands to produce the desired output. Some languages are standardized across printers from a particular manufacturer, and some are used across multiple manufacturer. This method lends itself better to embedded applications by allowing the printer to take on some of the computational overhead. Microchip provides support for some printer languages, including ESC/POS, PostScript, and PCL 5. Additional printer language can be implemented. Refer to the USB Embedded Host Printer Class application notes for more details on implementing printer language support.

Printer support is loosely divided into two categories: full sheet and point-of-sale (POS). Full sheet printers print on standard letter sized paper, and use printer languages such as PostScript and PCL 5. POS printers typically print on paper rolls, and use printer languages such as ESC/POS. The difference between printing on these two types of printers will be shown below.

Coordinate System - Full Sheet Printers

Locations on the printed page are specified in terms of (X,Y) coordinates. The (0,0) location on the page is located at the upper left corner, in either portrait or landscape mode. Ascending values of X proceed right across the page, and ascending values of Y proceed down the page. The scale of the coordinate system is 72 dots per inch, giving a maximum position of (611,791) in the portrait orientation, and (791,611) in the landscape orientation.

Standard vs. Page Mode - POS Printers

All POS printers support Standard Mode printing. In this mode, the printer prints the output as soon as it receives the command. Output is printed one line at a time, and vertical position is determined simply by the previous lines that were printed. Many POS printers also support Page Mode, where an entire "page" can be described before it is printed. Currently, only Standard Mode is supported.

Colors - Full Sheet Printers

Currently, only black and white printing is supported. All printing is performed with opaque colors; if a a white object is printed over a previously printed black object, the white object will be visible on the printed output. Therefore, it is important to consider the order in which the objects should be printed.

Colors - POS Printers

Most POS printers support only one color. Some printers, particularly impact printers that utilize a printer ribbon, can print in two colors. If the printer supports two color printing, use the commands USB_PRINTER_POS_COLOR_BLACK and USB_PRINTER_POS_COLOR_RED to specify the color.

Printing Commands

The application receives the event EVENT_PRINTER_ATTACH (see page 687) when a USB printer successfully attaches. The application can then send printing commands to the printer, utilizing either USBHostPrinterCommand (see page 636)() or USBHostPrinterCommandWithReadyWait (see page 639)().

Starting a Print Job

To start a print job, issue the command USB_PRINTER_JOB_START (USB_PRINTER_COMMAND (see page 742) enum). This will reset the printer back to its default settings.

Page Orientation - Full Sheet Printers

The page orientation must be set immediately after starting the print job, before any other printing commands. It cannot be changed in the middle of a page. To set portrait orientation, issue the command USB_PRINTER_ORIENTATION_PORTRAIT (USB_PRINTER_COMMAND (see page 742) enum). To set landscape orientation, issue the command USB_PRINTER_ORIENTATION_LANDSCAPE (USB_PRINTER_COMMAND (see page 742) enum). The default orientation is portrait.

Set Position - Full Sheet Printers

Many printer commands will be performed at the current location of the printer cursor. To move the printer cursor to the desired location, issue the command USB_PRINTER_SET_POSITION (USB_PRINTER_COMMAND (see page 742) enum).

Set Justification - POS Printers

Set the horizontal justification of the printed items to left, center, or right justification by issuing the command USB_PRINTER_POS_JUSTIFICATION_LEFT, USB_PRINTER_POS_JUSTIFICATION_CENTER, or USB_PRINTER_POS_JUSTIFICATION_RIGHT (USB_PRINTER_COMMAND (see page 742) enum).

Stop the Job

To finish the print job, issue the command USB_PRINTER_JOB_STOP (USB_PRINTER_COMMAND (see page 742) enum). This will print the page and reset the printer for the next job.

Selecting Fonts - Full Sheet Printers

Before printing text, select the desired font by issuing the command USB_PRINTER_FONT_NAME (USB_PRINTER_COMMAND (see page 742) enum). The available fonts, supported by most printers, are listed in the USB_PRINTER_FONTS (see page 753) enumeration. Select the size of the font in points by issuing the command USB_PRINTER_FONT_SIZE (USB_PRINTER_COMMAND (see page 742) enum). The font can be made italic by issuing the command USB_PRINTER_FONT_ITALIC (USB_PRINTER_COMMAND (see page 742) enum), and returned to upright by issuing the command USB_PRINTER_FONT_UPRIGHT (USB_PRINTER_COMMAND (see page 742) enum). The font can be made bold by issuing the command USB_PRINTER_FONT_BOLD (USB_PRINTER_COMMAND (see page 742) enum), and returned to medium weight be issuing the command USB_PRINTER_FONT_MEDIUM (USB_PRINTER_COMMAND (see page 742) enum).

Note: When the printer receives the font selection commands described above, the printer will select its best matching internal font. Some printers may not be able to support all fonts at all sizes and with all italic and bold combinations. Be sure to test the output on the target printer to ensure that the output appears as desired. In general, PostScript printers provide the best font support.

Bitmapped fonts are currently not supported.

Selecting Fonts - POS Printers

Before printing text, select the desired font by issuing the command USB_PRINTER_FONT_NAME (USB_PRINTER_COMMAND (see page 742) enum). The available fonts, supported by most printers, are listed in the USB_PRINTER_FONTS_POS (see page 754) enumeration. The font name also includes the font size, so the command USB_PRINTER_FONT_SIZE (USB_PRINTER_COMMAND (see page 742) enum) is not supported by POS printers. The font can be made bold by issuing the command USB_PRINTER_FONT_BOLD (USB_PRINTER_COMMAND (see page 742) enum), and returned to medium weight be issuing the command USB_PRINTER_FONT_MEDIUM (USB_PRINTER_COMMAND (see page 742) enum). Use the command USB_PRINTER_POS_FONT_UNDERLINE (USB_PRINTER_COMMAND (see page 742) enum) to enable and disable underlining. If the printer has to ability to do reverse text printing (white characters on a black background), use the command USB_PRINTER_POS_FONT_REVERSE (USB_PRINTER_COMMAND (see page 742) enum) to enable and disable reverse text printing.

User defined characters and fonts are not currently supported.

Printing Text - Full Sheet Printers

To print text, first set the printer cursor to the desired location, and select the desired font, as described above. Initialize text printing by issuing the command USB_PRINTER_TEXT_START (USB_PRINTER_COMMAND (see page 742) enum). After this command, issue the command USB_PRINTER_TEXT (USB_PRINTER_COMMAND (see page 742) enum) with the desired text string to print. Then issue the command USB_PRINTER_TEXT_STOP (USB_PRINTER_COMMAND (see page 742) enum).

This sequence of commands is required in order to support the various ways that the different printer languages handle text printing. Do not insert any other printer commands in this sequence, or the print will fail.

Different printers handle an embedded carriage returns and line feeds differently. For maximum compatibility across all printers, print each line of text separately.

Printing Text - POS Printers

The three command sequence described above can also be for printing text to POS printers. To simplify text printing in standard mode, use the command USB_PRINTER_POS_TEXT_LINE (USB_PRINTER_COMMAND (see page 742) enum) to print a single, null terminated string.

Printing Bitmapped Images

The printer languages supplied with USB Embedded Host Printer Class Support can print bitmapped images that are compatible with the Microchip Graphics Library bitmapped images. The images must be specified with one bit per pixel. A bit value of 0 indicates the color black, and a bit value of 1 indicates the color white. Images are opaque, not transparent. Image data begins at the top left corner, with the data proceeding from left to right, then top to bottom.

To print a bitmapped image, issue the command USB_PRINTER_IMAGE_START (USB_PRINTER_COMMAND (see page 742) enum) to initialize the image print. Be sure to examine the structure USB_PRINTER_IMAGE_INFO (see page 762), required by this command, and fill in all of the members appropriately for the image. Note that the position of the image on the paper is specified in the structure, so the printer cursor does not have to be explicitly set before this command. Next, send the image data to the printer, one line at a time. For each line, issue the command USB_PRINTER_IMAGE_DATA_HEADER (USB_PRINTER_COMMAND (see page 742) enum), then issue the command USB_PRINTER_IMAGE_DATA (USB_PRINTER_COMMAND (see page 742) enum) with a pointer to the row of bitmapped data. Be sure to correctly indicate the source of the data (RAM, ROM, or external memory). After all of the data has been transferred, issue the command USB_PRINTER_IMAGE_STOP (USB_PRINTER_COMMAND (see page 742) enum) to terminate image printing.

This sequence of commands is required in order to support the various ways that the different printer languages handle bitmapped image printing. Do not insert any other printer commands in this sequence, or the print will fail.

POS printers require image data in a slightly modified format from full sheet printers. Refer to the function USBHostPrinterPOSImageDataFormat (see page 655)() for further information about printing to POS printers.

NOTE: Some printer languages use the reverse polarity to specify black and white. For compatibility, the printer language drivers will automatically convert the image data to the format required by the particular printer, as long as the image data is located in ROM (USB_PRINTER_TRANSFER_FROM_ROM (see page 772)) or it is copied from a RAM buffer (USB_PRINTER_TRANSFER_COPY_DATA (see page 770)). If the data is to be sent directly from its original RAM location, the data must already be in the format required by the printer language. Refer to the main printer language documentation to see if the default polarity differs from 0=black, 1=white.

Vector Graphics - Full Sheet Printers

Some printer languages offer the ability to perform vector graphics, or the ability to print shapes such as lines and arcs via special commands instead of bitmaps. If vector graphics support is enabled, many additional commands are available to easily print shapes. Refer to the enumeration USB_PRINTER_COMMAND (see page 742) for the list of commands and which ones are supported only if vector graphics is enabled.

Multiple Page Output - Full Sheet Printers

If the print job contains multiple pages, issue the command USB_PRINTER_EJECT_PAGE (USB_PRINTER_COMMAND (see page 742) enum) to print and eject the current page. After this command, previous settings for orientation, font, and line type settings should be assumed to be undefined. Re-issue these commands at the beginning of each page to ensure correct output.

See *AN1233 - USB Printer Class on an Embedded Host* for more information about the Printer Class and this client driver.

7.2.10.1 Interface Routines

Functions

	Name	Description
	PrintScreen (see page 635)	This routine will extract the image that is currently on the specified portion of the graphics display, and print it at the specified location.
	USBHostPrinterCommand (see page 636)	This is the primary user interface function for the printer client driver. It is used to issue all printer commands.
	USBHostPrinterCommandReady (see page 638)	This interface is used to check if the client driver has space available to enqueue another transfer.
	USBHostPrinterDeviceDetached (see page 641)	This interface is used to check if the device has been detached from the bus.
	USBHostPrinterEventHandler (see page 642)	This routine is called by the Host layer to notify the printer client of events that occur.
	USBHostPrinterGetRxLength (see page 643)	This function retrieves the number of bytes copied to user's buffer by the most recent call to the USBHostPrinterRead (see page 659)() function.
	USBHostPrinterGetStatus (see page 644)	This function issues the Printer class-specific Device Request to obtain the printer status.
	USBHostPrinterInitialize (see page 645)	This function is called by the USB Embedded Host layer when a printer attaches.
	USBHostPrinterLanguageESCPOS (see page 646)	This function executes printer commands for an ESC/POS printer.
	USBHostPrinterLanguageESCPOSIsSupported (see page 648)	This function determines if the printer with the given device ID string supports the ESC/POS printer language.
	USBHostPrinterLanguagePCL5 (see page 649)	This function executes printer commands for a PCL 5 printer.
	USBHostPrinterLanguagePCL5IsSupported (see page 651)	This function determines if the printer with the given device ID string supports the PCL 5 printer language.
	USBHostPrinterLanguagePostScript (see page 652)	This function executes printer commands for a PostScript printer.
	USBHostPrinterLanguagePostScriptIsSupported (see page 654)	This function determines if the printer with the given device ID string supports the PostScript printer language.
	USBHostPrinterPOSImageDataFormat (see page 655)	This function formats data for a bitmapped image into the format required for sending to a POS printer.
	USBHostPrinterRead (see page 659)	Use this routine to receive from the device and store it into memory.
	USBHostPrinterReset (see page 660)	This function issues the Printer class-specific Device Request to perform a soft reset.
	USBHostPrinterRxIsBusy (see page 661)	This interface is used to check if the client driver is currently busy receiving data from the device.
	USBHostPrinterWrite (see page 662)	Use this routine to transmit data from memory to the device. This routine will not usually be called by the application directly. The application will use the USBHostPrinterCommand (see page 636)() function, which will call the appropriate printer language support function, which will utilize this routine.
	USBHostPrinterWriteComplete (see page 663)	This interface is used to check if the client driver is currently transmitting data to the printer, or if it is between transfers but still has transfers queued.

Macros

	Name	Description
	USBHostPrinterCommandWithReadyWait (see page 639)	This function is intended to be a short-cut to perform blocking calls to USBHostPrinterCommand (see page 636)(). While there is no space available in the printer queue (USBHostPrinterCommandReady (see page 638)() returns FALSE), USBTasks() is called. When space becomes available, USBHostPrinterCommand (see page 636)() is called. The return value from USBHostPrinterCommand (see page 636)() is returned in the returnCode parameter.
	USBHostPrinterPosition (see page 657)	This function is used to simplify the call to the printer command USB_PRINTER_SET_POSITION by generating the value needed for the specified (X,Y) coordinate.
	USBHostPrinterPositionRelative (see page 658)	This function is used to simplify the call to some of the printer graphics commands by generating the value needed for the specified change in X and change in Y coordinates.

Description

7.2.10.1.1 PrintScreen Function

This routine will extract the image that is currently on the specified portion of the graphics display, and print it at the specified location.

File

usb_host_printer_primitives.h

C

```
SHORT PrintScreen(
    BYTE address,
    USB_PRINT_SCREEN_INFO * printScreenInfo
);
```

Description

This routine is intended for use in an application that is using the Graphics Library to control a graphics display. This routine will extract the image that is currently on the specified portion of the graphics display, and print it at the specified location. Since the display may be in color and the printer can print only black and white, the pixel color to interpret as black must be specified in the USB_PRINT_SCREEN_INFO (see page 741) structure.

The function can be compiled as either a blocking function or a non-blocking function. When compiled as a blocking function, the routine will wait to enqueue all printer instructions. If an error occurs, then this function will return the error. If all printer instructions are enqueued successfully, the function will return -1. When compiled as a non-blocking function, this function will return 0 if the operation is proceeding correctly but has not yet completed. The application must continue to call this function, with the same parameters, until a non-zero value is returned. A value of -1 indicates that all printer instructions have been enqueued successfully. Any other value is an error code, and the state machine will be set back to the beginning state.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	USB address of the printer.
USB_PRINT_SCREEN_INFO *printScreenInfo	Information about the screen area to print, how to interpret the screen image, and how and where to print the image. Note that the width and height members of the structure do not need to be filled in by the application.

Return Values

Return Values	Description
0	Non-blocking configuration only. Image output is not yet complete, but is proceeding normally.
(-1)	Image output was completed successfully.
other	Printing was aborted due to an error. See the return values for USBHostPrinterCommand (see page 636)(). Note that the return code USB_PRINTER_SUCCESS will not be returned. Instead, (-1) will be returned upon successful completion.

Function

SHORT PrintScreen(BYTE address, USB_PRINT_SCREEN_INFO (see page 741) *printScreenInfo)

7.2.10.1.2 USBHostPrinterCommand Function

This is the primary user interface function for the printer client driver. It is used to issue all printer commands.

File

usb_host_printer.h

C

```
BYTE USBHostPrinterCommand(
    BYTE deviceAddress,
    USB_PRINTER_COMMAND command,
    USB_DATA_POINTER data,
    DWORD size,
    BYTE flags
);
```

Returns

See the USB_PRINTER_ERRORS (see page 752) enumeration. Also, refer to the printer language command handler function, such as USBHostPrinterLanguagePostScript (see page 652)().

Description

This is the primary user interface function for the printer client driver. It is used to issue all printer commands. Each generic printer command is translated to the appropriate command for the supported language and is enqueued for transfer to the printer. Before calling this routine, it is recommended to call the function USBHostPrinterCommandReady (see page 638)() to determine if there is space available in the printer's output queue.

Remarks

When developing new commands, keep in mind that the function USBHostPrinterCommandReady (see page 638)() will be used before calling this function to see if there is space available in the output transfer queue. USBHostPrinterCommandReady (see page 638)() will routine TRUE if a single space is available in the output queue. Therefore, each command can generate only one output transfer.

Preconditions

None

Example

```
if (USBHostPrinterCommandReady( address ))
{
    USBHostPrinterCommand( address, USB_PRINTER_JOB_START, USB_NULL, 0, 0 );
}
```

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Function

BYTE USBHostPrinterCommand(BYTE deviceAddress, USB_PRINTER_COMMAND (see page 742) command,

USB_DATA_POINTER (see page 736) data, DWORD size, BYTE flags)

7.2.10.1.3 USBHostPrinterCommandReady Function

File

usb_host_printer.h

C

```
BOOL USBHostPrinterCommandReady(
    BYTE deviceAddress
);
```

Description

This interface is used to check if the client driver has space available to enqueue another transfer.

Remarks

Use the definitions USB_DATA_POINTER_RAM (see page 737)() and USB_DATA_POINTER_ROM (see page 738)() to cast data pointers. For example:

```
    USBHostPrinterCommand( address, USB_PRINTER_TEXT, USB_DATA_POINTER_RAM(buffer),
strlen(buffer), 0 );
```

This routine will return TRUE if a single transfer can be enqueued. Since this routine is the check to see if USBHostPrinterCommand (see page 636)() can be called, every command can generate at most one transfer.

Preconditions

None

Example

```
if (USBHostPrinterCommandReady( address ))
{
    USBHostPrinterCommand( address, USB_PRINTER_JOB_START, USB_NULL, 0, 0 );
}
```

Parameters

Parameters	Description
deviceAddress	USB Address of the device

Return Values

Return Values	Description
TRUE	The printer client driver has room for at least one more transfer request, or the device is not attached. The latter allows this routine to be called without generating an infinite loop if the device detaches.
FALSE	The transfer queue is full.

Function

BOOL USBHostPrinterCommandReady(BYTE deviceAddress)

7.2.10.1.4 USBHostPrinterCommandWithReadyWait Macro

File

usb_host_printer.h

C

```
#define USBHostPrinterCommandWithReadyWait( returnCode, deviceAddress, command, data, size,
flags ) \
        {
                                                                                        
\
            while (!USBHostPrinterCommandReady( deviceAddress ))
USBTasks();                                 \
            *(returnCode) = USBHostPrinterCommand( deviceAddress, command, data, size,
flags );          \
        }
```

Returns

See the USB_PRINTER_ERRORS (see page 752) enumeration. Also, refer to the printer language command handler function, such as USBHostPrinterLanguagePostScript (see page 652)().

Description

This function is intended to be a short-cut to perform blocking calls to USBHostPrinterCommand (see page 636)(). While there is no space available in the printer queue (USBHostPrinterCommandReady (see page 638)() returns FALSE), USBTasks() is called. When space becomes available, USBHostPrinterCommand (see page 636)() is called. The return value from USBHostPrinterCommand (see page 636)() is returned in the returnCode parameter.

Remarks

Use the definitions USB_DATA_POINTER_RAM (see page 737)() and USB_DATA_POINTER_ROM (see page 738)() to cast data pointers. For example:

```
    USBHostPrinterCommandWithReadyWait( &rc, address, USB_PRINTER_TEXT,
USB_DATA_POINTER_RAM(buffer), strlen(buffer), 0 );
```

In the event that the device detaches during this routine, USBHostPrinterCommandReady (see page 638)() will return TRUE, and this function will return USB_PRINTER_UNKNOWN_DEVICE.

Preconditions

None

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Function

BYTE USBHostPrinterCommandWithReadyWait(BYTE &returnCode,

BYTE deviceAddress, USB_PRINTER_COMMAND (see page 742) command,

USB_DATA_POINTER (see page 736) data, DWORD size, BYTE flags)

7.2.10.1.5 USBHostPrinterDeviceDetached Function

File

usb_host_printer.h

C

```
BOOL USBHostPrinterDeviceDetached(
    BYTE deviceAddress
);
```

Description

This interface is used to check if the device has been detached from the bus.

Remarks

The event EVENT_PRINTER_DETACH (see page 688) can also be used to detect a detach.

Preconditions

None

Example

```
if (USBHostPrinterDeviceDetached( deviceAddress ))
{
    // Handle detach
}
```

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device.

Return Values

Return Values	Description
TRUE	The device has been detached, or an invalid deviceAddress is given.
FALSE	The device is attached

Function

BOOL USBHostPrinterDeviceDetached(BYTE deviceAddress)

7.2.10.1.6 USBHostPrinterEventHandler Function

This routine is called by the Host layer to notify the printer client of events that occur.

File

usb_host_printer.h

C

```
BOOL USBHostPrinterEventHandler(
    BYTE address,
    USB_EVENT event,
    void * data,
    DWORD size
);
```

Description

This routine is called by the Host layer to notify the printer client of events that occur. If the event is recognized, it is handled and the routine returns TRUE. Otherwise, it is ignored and the routine returns FALSE.

This routine can notify the application with the following events:

- EVENT_PRINTER_ATTACH (see page 687)
- EVENT_PRINTER_DETACH (see page 688)
- EVENT_PRINTER_TX_DONE (see page 694)
- EVENT_PRINTER_RX_DONE (see page 692)
- EVENT_PRINTER_REQUEST_DONE (see page 690)
- EVENT_PRINTER_UNSUPPORTED (see page 696)

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Address of device with the event
USB_EVENT event	The bus event that occured
void *data	Pointer to event-specific data
DWORD size	Size of the event-specific data

Return Values

Return Values	Description
TRUE	The event was handled
FALSE	The event was not handled

Function

BOOL USBHostPrinterEventHandler (BYTE address, USB_EVENT event,

void *data, DWORD size)

7.2.10.1.7 USBHostPrinterGetRxLength Function

File

usb_host_printer.h

C

```
DWORD USBHostPrinterGetRxLength(
    BYTE deviceAddress
);
```

Returns

Returns the number of bytes most recently received from the Printer device with address deviceAddress.

Description

This function retrieves the number of bytes copied to user's buffer by the most recent call to the USBHostPrinterRead (see page 659)() function.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device

Function

DWORD USBHostPrinterGetRxLength(BYTE deviceAddress)

7.2.10.1.8 USBHostPrinterGetStatus Function

This function issues the Printer class-specific Device Request to obtain the printer status.

File

usb_host_printer.h

C

```
BYTE USBHostPrinterGetStatus(
    BYTE deviceAddress,
    BYTE * status
);
```

Returns

See the return values for the USBHostIssueDeviceRequest() function.

Description

This function issues the Printer class-specific Device Request to obtain the printer status. The returned status should have the following format, per the USB specification. Any deviation will be due to the specific printer implementation.

- Bit 5 - Paper Empty; 1 = paper empty, 0 = paper not empty
- Bit 4 - Select; 1 = selected, 0 = not selected
- Bit 3 - Not Error; 1 = no error, 0 = error
- All other bits are reserved.

The *status parameter is not updated until the EVENT_PRINTER_REQUEST_DONE (see page 690) event is thrown. Until that point the value of *status is unknown.

The *status parameter will only be updated if this function returns USB_SUCCESS. If this function returns with any other error code then the EVENT_PRINTER_REQUEST_DONE (see page 690) event will not be thrown and the status field will not be updated.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
deviceAddress	USB Address of the device
*status	pointer to the returned status byte

Function

BYTE USBHostPrinterGetStatus(BYTE deviceAddress, BYTE *status)

7.2.10.1.9 USBHostPrinterInitialize Function

This function is called by the USB Embedded Host layer when a printer attaches.

File

usb_host_printer.h

C

```
BOOL USBHostPrinterInitialize(
    BYTE address,
    DWORD flags,
    BYTE clientDriverID
);
```

Description

This routine is a call out from the USB Embedded Host layer to the USB printer client driver. It is called when a "printer" device has been connected to the host. Its purpose is to initialize and activate the USB Printer client driver.

Remarks

Multiple client drivers may be used in a single application. The USB Embedded Host layer will call the initialize routine required for the attached device.

Preconditions

The device has been configured.

Parameters

Parameters	Description
BYTE address	Device's address on the bus
DWORD flags	Initialization flags
BYTE clientDriverID	Client driver identification for device requests

Return Values

Return Values	Description
TRUE	Initialization was successful
FALSE	Initialization failed

Function

BOOL USBHostPrinterInitialize (BYTE address, DWORD flags, BYTE clientDriverID)

7.2.10.1.10 USBHostPrinterLanguageESCPOS Function

This function executes printer commands for an ESC/POS printer.

File

usb_host_printer_esc_pos.h

C

```
BYTE USBHostPrinterLanguageESCPOS(
    BYTE address,
    USB_PRINTER_COMMAND command,
    USB_DATA_POINTER data,
    DWORD size,
    BYTE transferFlags
);
```

Description

This function executes printer commands for an ESC/POS printer. When the application issues a printer command, the printer client driver determines what language to use to communicate with the printer, and transfers the command to that language support routine. As much as possible, commands are designed to produce the same output regardless of what printer language is used.

Not all printer commands support data from both RAM and ROM. Unless otherwise noted, the data pointer is assumed to point to RAM, regardless of the value of transferFlags. Refer to the specific command to see if ROM data is supported.

Remarks

When developing new commands, keep in mind that the function USBHostPrinterCommandReady (see page 638)() will be used before calling this function to see if there is space available in the output transfer queue. USBHostPrinterCommandReady (see page 638)() will routine TRUE if a single space is available in the output queue. Therefore, each command can generate only one output transfer.

Multiple printer languages may be used in a single application. The USB Embedded Host Printer Client Driver will call the routine required for the attached device.

Preconditions

None

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Return Values

Return Values	Description
USB_PRINTER_SUCCESS	The command was executed successfully.

USB_PRINTER_UNKNOWN_DEVICE	A printer with the indicated address is not attached
USB_PRINTER_TOO_MANY_DEVICES	The printer status array does not have space for another printer.
USB_PRINTER_OUT_OF_MEMORY	Not enough available heap space to execute the command.
other	See possible return codes from the function USBHostPrinterWrite (see page 662)().

Function

BYTE USBHostPrinterLanguageESCPOS(BYTE address,

USB_PRINTER_COMMAND (see page 742) command, USB_DATA_POINTER (see page 736) data, DWORD size, BYTE transferFlags)

7.2.10.1.11 USBHostPrinterLanguageESCPOSIsSupported Function

File

usb_host_printer_esc_pos.h

C

```
BOOL USBHostPrinterLanguageESCPOSIsSupported(
    char * deviceID,
    USB_PRINTER_FUNCTION_SUPPORT * support
);
```

Description

This function determines if the printer with the given device ID string supports the ESC/POS printer language.

Remarks

The caller must first locate the "COMMAND SET:" section of the device ID string. To ensure that only the "COMMAND SET:" section of the device ID string is checked, the ";" at the end of the section should be temporarily replaced with a NULL. Otherwise, this function may find the printer language string in the comments or other section, and incorrectly indicate that the printer supports the language.

Device ID strings are case sensitive.

See the file header comments for this file (usb_host_printer_esc_pos.h) for cautions regarding dynamic printer language selection with POS printers.

Preconditions

None

Parameters

Parameters	Description
char *deviceID	Pointer to the "COMMAND SET:" portion of the device ID string of the attached printer.
USB_PRINTER_FUNCTION_SUPPORT *support	Pointer to returned information about what types of functions this printer supports.

Return Values

Return Values	Description
TRUE	The printer supports ESC/POS.
FALSE	The printer does not support ESC/POS.

Function

BOOL USBHostPrinterLanguageESCPOSIsSupported(char *deviceID,

USB_PRINTER_FUNCTION_SUPPORT (see page 755) *support)

7.2.10.1.12 USBHostPrinterLanguagePCL5 Function

This function executes printer commands for a PCL 5 printer.

File

usb_host_printer_pcl_5.h

C

```
BYTE USBHostPrinterLanguagePCL5(
    BYTE address,
    USB_PRINTER_COMMAND command,
    USB_DATA_POINTER data,
    DWORD size,
    BYTE transferFlags
);
```

Description

This function executes printer commands for a PCL 5 printer. When the application issues a printer command, the printer client driver determines what language to use to communicate with the printer, and transfers the command to that language support routine. As much as possible, commands are designed to produce the same output regardless of what printer language is used.

Not all printer commands support data from both RAM and ROM. Unless otherwise noted, the data pointer is assumed to point to RAM, regardless of the value of transferFlags. Refer to the specific command to see if ROM data is supported.

Remarks

When developing new commands, keep in mind that the function USBHostPrinterCommandReady (see page 638)() will be used before calling this function to see if there is space available in the output transfer queue. USBHostPrinterCommandReady (see page 638)() will routine TRUE if a single space is available in the output queue. Therefore, each command can generate only one output transfer.

Multiple printer languages may be used in a single application. The USB Embedded Host Printer Client Driver will call the routine required for the attached device.

Preconditions

None

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Return Values

Return Values	Description
USB_PRINTER_SUCCESS	The command was executed successfully.

USB_PRINTER_UNKNOWN_DEVICE	A printer with the indicated address is not attached
USB_PRINTER_TOO_MANY_DEVICES	The printer status array does not have space for another printer.
USB_PRINTER_OUT_OF_MEMORY	Not enough available heap space to execute the command.
other	See possible return codes from the function USBHostPrinterWrite (see page 662)().

Function

BYTE USBHostPrinterLanguagePCL5(BYTE address,

USB_PRINTER_COMMAND (see page 742) command, USB_DATA_POINTER (see page 736) data, DWORD size, BYTE transferFlags)

7.2.10.1.13 USBHostPrinterLanguagePCL5IsSupported Function

This function determines if the printer with the given device ID string supports the PCL 5 printer language.

File

usb_host_printer_pcl_5.h

C

```
BOOL USBHostPrinterLanguagePCL5IsSupported(
    char * deviceID,
    USB_PRINTER_FUNCTION_SUPPORT * support
);
```

Description

This function determines if the printer with the given device ID string supports the PCL 5 printer language.

Unfortunately, printer language support is not always advertised correctly by the printer. Some printers advertise only PCL 6 support when they also support PCL 5. Therefore, this routine will return TRUE if any PCL language support is advertised. It is therefore highly recommended to test the target application with the specific printer(s) that will be utilized.

Remarks

The caller must first locate the "COMMAND SET:" section of the device ID string. To ensure that only the "COMMAND SET:" section of the device ID string is checked, the ";" at the end of the section should be temporarily replaced with a NULL. Otherwise, this function may find the printer language string in the comments or other section, and incorrectly indicate that the printer supports the language.

Device ID strings are case sensitive.

Preconditions

None

Parameters

Parameters	Description
char *deviceID	Pointer to the "COMMAND SET:" portion of the device ID string of the attached printer.
USB_PRINTER_FUNCTION_SUPPORT *support	Pointer to returned information about what types of functions this printer supports.

Return Values

Return Values	Description
TRUE	The printer supports PCL 5.
FALSE	The printer does not support PCL 5.

Function

BOOL USBHostPrinterLanguagePCL5IsSupported(char *deviceID,

USB_PRINTER_FUNCTION_SUPPORT (see page 755) *support)

7.2.10.1.14 USBHostPrinterLanguagePostScript Function

This function executes printer commands for a PostScript printer.

File

usb_host_printer_postscript.h

C

```
BYTE USBHostPrinterLanguagePostScript(
    BYTE address,
    USB_PRINTER_COMMAND command,
    USB_DATA_POINTER data,
    DWORD size,
    BYTE transferFlags
);
```

Description

This function executes printer commands for a PostScript printer. When the application issues a printer command, the printer client driver determines what language to use to communicate with the printer, and transfers the command to that language support routine. As much as possible, commands are designed to produce the same output regardless of what printer language is used.

Not all printer commands support data from both RAM and ROM. Unless otherwise noted, the data pointer is assumed to point to RAM, regardless of the value of transferFlags. Refer to the specific command to see if ROM data is supported.

Remarks

When developing new commands, keep in mind that the function USBHostPrinterCommandReady (see page 638)() will be used before calling this function to see if there is space available in the output transfer queue. USBHostPrinterCommandReady (see page 638)() will routine TRUE if a single space is available in the output queue. Therefore, each command can generate only one output transfer.

Multiple printer languages may be used in a single application. The USB Embedded Host Printer Client Driver will call the routine required for the attached device.

Preconditions

None

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Return Values

Return Values	Description
USB_PRINTER_SUCCESS	The command was executed successfully.

USB_PRINTER_UNKNOWN_DEVICE	A printer with the indicated address is not attached
USB_PRINTER_TOO_MANY_DEVICES	The printer status array does not have space for another printer.
USB_PRINTER_OUT_OF_MEMORY	Not enough available heap space to execute the command.
other	See possible return codes from the function USBHostPrinterWrite (see page 662)().

Function

BYTE USBHostPrinterLanguagePostScript(BYTE address,

USB_PRINTER_COMMAND (see page 742) command, USB_DATA_POINTER (see page 736) data, DWORD size, BYTE transferFlags)

7.2.10.1.15 USBHostPrinterLanguagePostScriptIsSupported Function

File

usb_host_printer_postscript.h

C

```
BOOL USBHostPrinterLanguagePostScriptIsSupported(
    char * deviceID,
    USB_PRINTER_FUNCTION_SUPPORT * support
);
```

Description

This function determines if the printer with the given device ID string supports the PostScript printer language.

Remarks

The caller must first locate the "COMMAND SET:" section of the device ID string. To ensure that only the "COMMAND SET:" section of the device ID string is checked, the ";" at the end of the section should be temporarily replaced with a NULL. Otherwise, this function may find the printer language string in the comments or other section, and incorrectly indicate that the printer supports the language.

Device ID strings are case sensitive.

Preconditions

None

Parameters

Parameters	Description
char *deviceID	Pointer to the "COMMAND SET:" portion of the device ID string of the attached printer.
USB_PRINTER_FUNCTION_SUPPORT *support	Pointer to returned information about what types of functions this printer supports.

Return Values

Return Values	Description
TRUE	The printer supports PostScript.
FALSE	The printer does not support PostScript.

Function

BOOL USBHostPrinterLanguagePostScriptIsSupported(char *deviceID,

USB_PRINTER_FUNCTION_SUPPORT (see page 755) *support)

7.2.10.1.16 USBHostPrinterPOSImageDataFormat Function

This function formats data for a bitmapped image into the format required for sending to a POS printer.

File

usb_host_printer_esc_pos.h

C

```
USB_DATA_POINTER USBHostPrinterPOSImageDataFormat(
    USB_DATA_POINTER image,
    BYTE imageLocation,
    WORD imageHeight,
    WORD imageWidth,
    WORD * currentRow,
    BYTE byteDepth,
    BYTE * imageData
);
```

Returns

The function returns a pointer to the next byte of image data.

Description

This function formats data for a bitmapped image into the format required for sending to a POS printer. Bitmapped images are stored one row of pixels at a time. Suppose we have an image with vertical black bars, eight pixels wide and eight pixels deep. The image would appear as the following pixels, where 0 indicates a black dot and 1 indicates a white dot:

```
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
```

The stored bitmap of the data would contain the data bytes, where each byte is one row of data:

```
0x55 0x55 0x55 0x55 0x55 0x55 0x55 0x55
```

When printing to a full sheet printer, eight separate USB_PRINTER_IMAGE_DATA_HEADER / USB_PRINTER_IMAGE_DATA command combinations are required to print this image.

POS printers, however, require image data formated either 8 dots or 24 dots deep, depending on the desired (and supported) vertical print density. For a POS printer performing an 8-dot vertical density print, the data needs to be in this format:

```
0x00 0xFF 0x00 0xFF 0x00 0xFF 0x00 0xFF
```

When printing to a POS printer, only one USB_PRINTER_IMAGE_DATA_HEADER / USB_PRINTER_IMAGE_DATA command combination is required to print this image.

This function supports 8-dot and 24-dot vertical densities by specifying the byteDepth parameter as either 1 (8-dot) or 3 (24-dot).

Remarks

This routine currently does not support 36-dot density printing. Since the output for 36-dot vertical density is identical to 24-dot vertical density, 24-dot vertical density should be used instead.

This routine does not yet support reading from external memory.

Preconditions

None

Example

The following example code will send a complete bitmapped image to a POS printer.

```
    WORD                    currentRow;
    BYTE                    depthBytes;
    BYTE                    *imageDataPOS;
    USB_PRINTER_IMAGE_INFO  imageInfo;
    BYTE                    returnCode;
    #if defined (__C30__)
        BYTE __prog__       *ptr;
        ptr = (BYTE __prog__ *)logoMCHP.address;
    #elif defined (__PIC32MX__)
        const BYTE          *ptr;
        ptr = (const BYTE *)logoMCHP.address;
    #endif

    imageInfo.densityVertical   = 24;   // 24-dot density
    imageInfo.densityHorizontal = 2;    // Double density

    // Extract the image height and width
    imageInfo.width     = ((WORD)ptr[5] << 8) + ptr[4];
    imageInfo.height    = ((WORD)ptr[3] << 8) + ptr[2];

    depthBytes          = imageInfo.densityVertical / 8;
    imageDataPOS        = (BYTE *)malloc( imageInfo.width *
                                    depthBytes );

    if (imageDataPOS == NULL)
    {
        // Error - not enough heap space
    }

    USBHostPrinterCommandWithReadyWait( &returnCode,
          printerInfo.deviceAddress, USB_PRINTER_IMAGE_START,
          USB_DATA_POINTER_RAM(&imageInfo),
          sizeof(USB_PRINTER_IMAGE_INFO ),
          0 );

    ptr += 10; // skip the header info

    currentRow = 0;
    while (currentRow < imageInfo.height)
    {
        USBHostPrinterCommandWithReadyWait( &returnCode,
          printerInfo.deviceAddress,
          USB_PRINTER_IMAGE_DATA_HEADER, USB_NULL,
          imageInfo.width, 0 );

        ptr = USBHostPrinterPOSImageDataFormat(
          USB_DATA_POINTER_ROM(ptr),
          USB_PRINTER_TRANSFER_FROM_ROM, imageInfo.height,
          imageInfo.width, &currentRow, depthBytes,
          imageDataPOS ).pointerROM;

        USBHostPrinterCommandWithReadyWait( &returnCode,
          printerInfo.deviceAddress, USB_PRINTER_IMAGE_DATA,
          USB_DATA_POINTER_RAM(imageDataPOS), imageInfo.width,
          USB_PRINTER_TRANSFER_COPY_DATA);
    }

    free( imageDataPOS );

    USBHostPrinterCommandWithReadyWait( &returnCode,
          printerInfo.deviceAddress, USB_PRINTER_IMAGE_STOP,
          USB_NULL, 0, 0 );
```

Function

USB_DATA_POINTER (see page 736) USBHostPrinterPOSImageDataFormat(USB_DATA_POINTER (see page 736) image,

BYTE imageLocation, WORD imageHeight, WORD imageWidth, WORD *currentRow,

BYTE byteDepth, BYTE *imageData)

7.2.10.1.17 USBHostPrinterPosition Macro

This function is used to simplify the call to the printer command USB_PRINTER_SET_POSITION by generating the value needed for the specified (X,Y) coordinate.

File

usb_host_printer.h

C

```
#define USBHostPrinterPosition( X, Y ) (((DWORD)(X) << 16) | ((DWORD)(Y) & 0xFFFF))
```

Returns

DWORD value that can be used in the USBHostPrinterCommand (see page 636)() function call with the command USB_PRINTER_SET_POSITION.

Description

This function is used to simplify the call to the printer command USB_PRINTER_SET_POSITION by generating the value needed for the specified (X,Y) coordinate. The USB_PRINTER_SET_POSITION command requires that the (X,Y) coordinate be passed in the (DWORD) size parameter of the USBHostPrinterCommand (see page 636)() function. This function takes the specified coordinate and packs it in the DWORD as required.

Remarks

None

Preconditions

None

Example

```
USBHostPrinterCommand( printer, USB_PRINTER_SET_POSITION, USB_NULL,
                USBHostPrinterPosition( 100, 100 ), 0 );
```

Parameters

Parameters	Description
X	X coordinate (horizontal)
Y	Y coordinate (vertical)

Function

DWORD USBHostPrinterPosition(WORD X, WORD Y)

7.2.10.1.18 USBHostPrinterPositionRelative Macro

This function is used to simplify the call to some of the printer graphics commands by generating the value needed for the specified change in X and change in Y coordinates.

File

usb_host_printer.h

C

```
#define USBHostPrinterPositionRelative( dX, dY ) (((DWORD)(dX) << 16) | ((DWORD)(dY) &
0xFFFF))
```

Returns

DWORD value that can be used in the USBHostPrinterCommand (see page 636)() function call with the commands USB_PRINTER_GRAPHICS_MOVE_RELATIVE and USB_PRINTER_GRAPHICS_LINE_TO_RELATIVE.

Description

This function is used to simplify the call to some of the printer graphics commands by generating the value needed for the specified change in X and change in Y coordinates. The USB_PRINTER_GRAPHICS_MOVE_RELATIVE and the USB_PRINTER_GRAPHICS_LINE_TO_RELATIVE commands requires that the change in the (X,Y) coordinates be passed in the (DWORD) size parameter of the USBHostPrinterCommand (see page 636)() function. This function takes the specified coordinate changes and packs them in the DWORD as required.

Remarks

None

Preconditions

None

Example

```
USBHostPrinterCommand( printer, USB_PRINTER_GRAPHICS_LINE_TO_RELATIVE, USB_NULL,
                USBHostPrinterPositionRelative( 0, -100 ), 0 );
```

Parameters

Parameters	Description
dX	Change in the X coordinate (horizontal)
dY	Change in the Y coordinate (vertical)

Function

DWORD USBHostPrinterPositionRelative(SHORT dX, SHORT dY)

7.2.10.1.19 USBHostPrinterRead Function

File

usb_host_printer.h

C

```
BYTE USBHostPrinterRead(
    BYTE deviceAddress,
    void * buffer,
    DWORD length,
    BYTE transferFlags
);
```

Description

Use this routine to receive from the device and store it into memory.

Remarks

None

Preconditions

The device must be connected and enumerated.

Example

```
if (!USBHostPrinterRxIsBusy( deviceAddress ))
{
    USBHostPrinterRead( deviceAddress, &buffer, sizeof(buffer), 0 );
}
```

Parameters

Parameters	Description
deviceAddress	USB Address of the device.
buffer	Pointer to the data buffer
length	Number of bytes to be transferred
transferFlags	Flags for how to perform the operation

Return Values

Return Values	Description
USB_SUCCESS	The Read was started successfully
(USB error code)	The Read was not started. See USBHostRead (see page 301)() for a list of errors.

Function

BYTE USBHostPrinterRead(BYTE deviceAddress, BYTE *buffer, DWORD length,

BYTE transferFlags)

7.2.10.1.20 USBHostPrinterReset Function

File

usb_host_printer.h

C

```
BYTE USBHostPrinterReset(
    BYTE deviceAddress
);
```

Returns

See the return values for the USBHostIssueDeviceRequest() function.

Description

This function issues the Printer class-specific Device Request to perform a soft reset.

Remarks

Not all printers support this command.

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device

Function

BYTE USBHostPrinterReset(BYTE deviceAddress)

7.2.10.1.21 USBHostPrinterRxIsBusy Function

This interface is used to check if the client driver is currently busy receiving data from the device.

File

usb_host_printer.h

C

```
BOOL USBHostPrinterRxIsBusy(
    BYTE deviceAddress
);
```

Description

This interface is used to check if the client driver is currently busy receiving data from the device.

Remarks

None

Preconditions

None

Example

```
if (!USBHostPrinterRxIsBusy( deviceAddress ))
{
    USBHostPrinterRead( deviceAddress, &buffer, sizeof( buffer ) );
}
```

Parameters

Parameters	Description
deviceAddress	USB Address of the device

Return Values

Return Values	Description
TRUE	The device is receiving data or an invalid deviceAddress is given.
FALSE	The device is not receiving data

Function

BOOL USBHostPrinterRxIsBusy(BYTE deviceAddress)

7.2.10.1.22 USBHostPrinterWrite Function

File

usb_host_printer.h

C

```
BYTE USBHostPrinterWrite(
    BYTE deviceAddress,
    void * buffer,
    DWORD length,
    BYTE flags
);
```

Description

Use this routine to transmit data from memory to the device. This routine will not usually be called by the application directly. The application will use the USBHostPrinterCommand (see page 636)() function, which will call the appropriate printer language support function, which will utilize this routine.

Remarks

None

Preconditions

The device must be connected and enumerated.

Parameters

Parameters	Description
BYTE deviceAddress	USB Address of the device.
void *buffer	Pointer to the data buffer
DWORD length	Number of bytes to be transferred
BYTE transferFlags	Flags for how to perform the operation

Return Values

Return Values	Description
USB_SUCCESS	The Write was started successfully.
USB_PRINTER_UNKNOWN_DEVICE	Device not found or has not been initialized.
USB_PRINTER_BUSY	The printer's output queue is full.
(USB error code)	The Write was not started. See USBHostWrite (see page 313)() for a list of errors.

Function

BYTE USBHostPrinterWrite(BYTE deviceAddress, void *buffer, DWORD length,

BYTE transferFlags)

7.2.10.1.23 USBHostPrinterWriteComplete Function

File

usb_host_printer.h

C

```
BOOL USBHostPrinterWriteComplete(
    BYTE deviceAddress
);
```

Description

This interface is used to check if the client driver is currently transmitting data to the printer, or if it is between transfers but still has transfers queued.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
deviceAddress	USB Address of the device

Return Values

Return Values	Description
TRUE	The device is done transmitting data or an invalid deviceAddress is given.
FALSE	The device is transmitting data or has a transfer in the queue.

Function

BOOL USBHostPrinterWriteComplete(BYTE deviceAddress)

7.2.10.2 Data Types and Constants

Enumerations

	Name	Description
	USB_PRINTER_COMMAND (see page 742)	USB Printer Client Driver Commands The main interface to the USB Printer Client Driver is through the function USBHostPrinterCommand (see page 636)(). These are the commands that can be passed to that function.
	USB_PRINTER_ERRORS (see page 752)	Printer Errors These are errors that can be returned by the printer client driver. Note that USB Embedded Host errors can also be returned.
	USB_PRINTER_FONTS (see page 753)	Printer Fonts This enumeration defines the various printer fonts. If new fonts are added, they must be added at the end of the list, just before the USB_PRINTER_FONT_MAX_FONT definition, as the printer language support files may utilize them for indexing purposes.
	USB_PRINTER_FONTS_POS (see page 754)	POS Printer Fonts This enumeration defines the various printer fonts used by POS printers. If new fonts are added, they must be added at the end of the list, just before the USB_PRINTER_FONT_POS_MAX_FONT definition, as the printer language support files may utilize them for indexing purposes.
	USB_PRINTER_POS_BARCODE_FORMAT (see page 767)	Bar Code Formats These are the bar code formats for printing bar codes on POS printers. They are used in conjuction with the USB_PRINTER_POS_BARCODE command (USB_PRINTER_COMMAND (see page 742)). Bar code information is passed using the sBarCode structure within the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. The exact values to send for each bar code type can vary for the particular POS printer, and not all printers support all bar code types. Be sure to test the output on the target printer, and adjust the values specified in usb_host_printer_esc_pos.c if necessary. Refer to the printer's technical documentation for the required values. Do not alter this... more (see page 767)

Macros

	Name	Description
	_USB_HOST_PRINTER_PRIMITIVES_H (see page 672)	This is macro _USB_HOST_PRINTER_PRIMITIVES_H.
	BARCODE_CODE128_CODESET_A_CHAR (see page 673)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE A. This code set should be used if control characters (0x00-0x1F) are included in the data.
	BARCODE_CODE128_CODESET_A_STRING (see page 674)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE A. This code set should be used if control characters (0x00-0x1F) are included in the data.

	BARCODE_CODE128_CODESET_B_CHAR (see page 675)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE B. This code set should be used if lower case letters and higher ASCII characters (0x60-0x7F) are included in the data.
	BARCODE_CODE128_CODESET_B_STRING (see page 676)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE B. This code set should be used if lower case letters and higher ASCII characters (0x60-0x7F) are included in the data.
	BARCODE_CODE128_CODESET_C_CHAR (see page 677)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE C. This code set can be used only if the data values are between 0 and 99 (0x00-0x63).
	BARCODE_CODE128_CODESET_C_STRING (see page 678)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE C. This code set can be used only if the data values are between 0 and 99 (0x00-0x63).
	BARCODE_CODE128_CODESET_CHAR (see page 679)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the first data byte of the bar code data, which begins the character code specification. The next byte must be 'A', 'B', or 'C'.
	BARCODE_CODE128_CODESET_STRING (see page 680)	For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the first data byte of the bar code data, which begins the character code specification. The next byte must be 'A', 'B', or 'C'.
	BARCODE_TEXT_12x24 (see page 681)	For use with POS printers. May not be valid for all printers. Print the bar code text in 12x24 dot font. Do not alter this value.
	BARCODE_TEXT_18x36 (see page 682)	For use with POS printers. May not be valid for all printers. Print the bar code text in 18x36 dot font. Do not alter this value.
	BARCODE_TEXT_ABOVE (see page 683)	For use with POS printers. Print readable text above the bar code. Do not alter this value.
	BARCODE_TEXT_ABOVE_AND_BELOW (see page 684)	For use with POS printers. Print readable text above and below the bar code. Do not alter this value.
	BARCODE_TEXT_BELOW (see page 685)	For use with POS printers. Print readable text below the bar code. Do not alter this value.
	BARCODE_TEXT_OMIT (see page 686)	For use with POS printers. Do not print readable bar code text. Do not alter this value.

	EVENT_PRINTER_ATTACH (see page 687)	This event indicates that a Printer device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, the *data parameter points to a structure of the type USB_PRINTER_DEVICE_ID (see page 751), which provides important information about the attached printer. The size parameter is the size of this structure.
	EVENT_PRINTER_DETACH (see page 688)	This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.
	EVENT_PRINTER_OFFSET (see page 689)	This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.
	EVENT_PRINTER_REQUEST_DONE (see page 690)	This event indicates that the printer request has completed. These requests occur on endpoint 0 and include getting the printer status and performing a soft reset.
	EVENT_PRINTER_REQUEST_ERROR (see page 691)	This event indicates that a bus error occurred while trying to perform a device request. The error code is returned in the size parameter. The data parameter is returned as NULL.
	EVENT_PRINTER_RX_DONE (see page 692)	This event indicates that a previous read request has completed. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the receive buffer, and size is the actual number of bytes read from the device.
	EVENT_PRINTER_RX_ERROR (see page 693)	This event indicates that a bus error occurred while trying to perform a read. The error code is returned in the size parameter. The data parameter is returned as NULL.
	EVENT_PRINTER_TX_DONE (see page 694)	This event indicates that a previous write request has completed. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.
	EVENT_PRINTER_TX_ERROR (see page 695)	This event indicates that a bus error occurred while trying to perform a write. The error code is returned in the size parameter. The data parameter is returned as NULL.
	EVENT_PRINTER_UNSUPPORTED (see page 696)	This event indicates that a printer has attached for which we do not have printer language support. Therefore, we cannot talk to this printer. This event can also occur if there is not enough dynamic memory available to read the device ID string.

	LANGUAGE_ID_STRING_ESCPOS (see page 697)	This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive. See the function USBHostPrinterLanguageESCPOSIsSupported (see page 648)() for more information about using or changing this string. Dynamic language determination is not recommended when using POS printers.
	LANGUAGE_ID_STRING_PCL (see page 698)	This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive. Some printers that report only PCL 6 support also support PCL 5. So it is recommended to use "PCL" as the search string, rather than "PCL 5", and verify that the correct output is produced by the target printer.
	LANGUAGE_ID_STRING_POSTSCRIPT (see page 699)	This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive.
	LANGUAGE_SUPPORT_FLAGS_ESCPOS (see page 700)	These are the support flags that are set for this language.
	LANGUAGE_SUPPORT_FLAGS_PCL3 (see page 701)	These are the support flags that are set for the PCL 3 version of this language.
	LANGUAGE_SUPPORT_FLAGS_PCL5 (see page 702)	These are the support flags that are set for the PCL 5 version of this language.
	LANGUAGE_SUPPORT_FLAGS_POSTSCRIPT (see page 703)	These are the support flags that are set for this language.
	PRINTER_COLOR_BLACK (see page 704)	Indicates a black line for drawing graphics objects.
	PRINTER_COLOR_WHITE (see page 705)	Indicates a white line for drawing graphics objects.
	PRINTER_DEVICE_REQUEST_GET_DEVICE_ID (see page 706)	bRequest value for the GET_DEVICE_ID USB class-specific request.
	PRINTER_DEVICE_REQUEST_GET_PORT_STATUS (see page 707)	bRequest value for the GET_PORT_STATUS USB class-specific request.
	PRINTER_DEVICE_REQUEST_SOFT_RESET (see page 708)	bRequest value for the SOFT_RESET USB class-specific request.
	PRINTER_FILL_CROSS_HATCHED (see page 709)	Indicates a cross-hatched fill for graphics objects. Requires a specified line spacing and angle.
	PRINTER_FILL_HATCHED (see page 710)	Indicates a hatched fill for graphics objects. Requires a specified line spacing and angle.
	PRINTER_FILL_SHADED (see page 711)	Indicates a shaded fill for filled graphics objects. Requires a specified fill percentage.
	PRINTER_FILL_SOLID (see page 712)	Indicates a solid color fill for filled graphics objects.
	PRINTER_LINE_END_BUTT (see page 713)	Drawn lines will have a butt end.
	PRINTER_LINE_END_ROUND (see page 714)	Drawn lines will have a round end.
	PRINTER_LINE_END_SQUARE (see page 715)	Drawn lines will have a square end.
	PRINTER_LINE_JOIN_BEVEL (see page 716)	Drawn lines will be joined with a bevel.
	PRINTER_LINE_JOIN_MITER (see page 717)	Drawn lines will be joined with a miter.
	PRINTER_LINE_JOIN_ROUND (see page 718)	Drawn lines will be joined with a round.

	PRINTER_LINE_TYPE_DASHED (see page 719)	Indicates a dashed line for drawing graphics objects.
	PRINTER_LINE_TYPE_DOTTED (see page 720)	Indicates a dotted line for drawing graphics objects.
	PRINTER_LINE_TYPE_SOLID (see page 721)	Indicates a solid line for drawing graphics objects.
	PRINTER_LINE_WIDTH_NORMAL (see page 722)	Indicates a normal width line for drawing graphics objects.
	PRINTER_LINE_WIDTH_THICK (see page 723)	Indicates a thick line for drawing graphics objects.
	PRINTER_PAGE_LANDSCAPE_HEIGHT (see page 724)	The height of the page in points when in landscape mode.
	PRINTER_PAGE_LANDSCAPE_WIDTH (see page 725)	The width of the page in points when in landscape mode.
	PRINTER_PAGE_PORTRAIT_HEIGHT (see page 726)	The height of the page in points when in portrait mode.
	PRINTER_PAGE_PORTRAIT_WIDTH (see page 727)	The width of the page in points when in portrait mode.
	PRINTER_POS_BOTTOM_TO_TOP (see page 728)	POS print direction bottom to top, starting at the bottom left corner.
	PRINTER_POS_DENSITY_HORIZONTAL_DOUBLE (see page 729)	Image print with double horizontal density.
	PRINTER_POS_DENSITY_HORIZONTAL_SINGLE (see page 730)	Image print with single horizontal density.
	PRINTER_POS_DENSITY_VERTICAL_24 (see page 731)	Image print with 24-dot vertical density.
	PRINTER_POS_DENSITY_VERTICAL_8 (see page 732)	Image print with 8-dot vertical density.
	PRINTER_POS_LEFT_TO_RIGHT (see page 733)	POS print direction left to right, starting at the top left corner.
	PRINTER_POS_RIGHT_TO_LEFT (see page 734)	POS print direction right to left, startin at the bottom right corner.
	PRINTER_POS_TOP_TO_BOTTOM (see page 735)	POS print direction top to bottom, starting at the top right corner.
	USB_DATA_POINTER_RAM (see page 737)	Use this definition to cast a pointer being passed to the function USBHostPrinterCommand (see page 636)() that points to data in RAM.
	USB_DATA_POINTER_ROM (see page 738)	Use this definition to cast a pointer being passed to the function USBHostPrinterCommand (see page 636)() that points to data in ROM.
	USB_MAX_PRINTER_DEVICES (see page 739)	Max Number of Supported Devices This value represents the maximum number of attached devices this class driver can support. If the user does not define a value, it will be set to 1. Currently this must be set to 1, due to limitations in the USB Host layer.
	USB_NULL (see page 740)	Use this definition to pass a NULL pointer to the function USBHostPrinterCommand (see page 636)().
	USB_PRINTER_FUNCTION_SUPPORT_POS (see page 756)	Constant to use to set the supportsPOS member of the USB_PRINTER_FUNCTION_SUPPORT (see page 755) union.
	USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS (see page 757)	Constant to use to set the supportsVectorGraphics member of the USB_PRINTER_FUNCTION_SUPPORT (see page 755) union.

	USB_PRINTER_TRANSFER_COPY_DATA (see page 770)	This flag indicates that the printer client driver should make a copy of the data passed to the command. This allows the application to reuse the data storage immediately instead of waiting until the transfer is sent to the printer. The client driver will allocate space in the heap for the data copy. If there is not enough available memory, the command will terminate with a USB_PRINTER_OUT_OF_MEMORY error. Otherwise, the original data will be copied to the temporary data space. This temporary data will be freed upon completion, regardless of whether or not the command was performed successfully. NOTE: If... more (see page 770)
	USB_PRINTER_TRANSFER_FROM_RAM (see page 771)	This flag indicates that the source of the command data is in RAM. The application can then choose whether or not to have the printer client driver make a copy of the data.
	USB_PRINTER_TRANSFER_FROM_ROM (see page 772)	This flag indicates that the source of the command data is in ROM. The data will be copied to RAM, since the USB Host layer cannot read data from ROM. If there is not enough available heap space to make a copy of the data, the command will fail. If using this flag, do not set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag.
	USB_PRINTER_TRANSFER_NOTIFY (see page 773)	This flag indicates that the application layer wants to receive an event notification when the command completes.
	USB_PRINTER_TRANSFER_STATIC_DATA (see page 774)	This flag indicates that the data will not change in the time between the printer command being issued and the data actually being sent to the printer.
	USBHOSTPRINTER_SETFLAG_COPY_DATA (see page 775)	Use this macro to set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag in a variable.
	USBHOSTPRINTER_SETFLAG_NOTIFY (see page 776)	Use this macro to set the USB_PRINTER_TRANSFER_NOTIFY (see page 773) flag in a variable.
	USBHOSTPRINTER_SETFLAG_STATIC_DATA (see page 777)	Use this macro to clear the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag in a variable.

Structures

	Name	Description
	_USB_PRINTER_DEVICE_ID (see page 751)	Printer Device ID Information This structure contains identification information about an attached device.
	USB_PRINT_SCREEN_INFO (see page 741)	Print Screen Information This structure is designed for use when the USB Embedded Host Printer support is integrated with the graphics library. The structure contains the information needed to print a portion of the graphics screen as a bitmapped graphic image.
	USB_PRINTER_DEVICE_ID (see page 751)	Printer Device ID Information This structure contains identification information about an attached device.

	USB_PRINTER_IMAGE_INFO (see page 762)	Bitmapped Image Information This structure contains the information needed to print a bitmapped graphic image. When using a full sheet printer, utilize the resolution and the scale members to specify the size of the image. Some printer languages (e.g. PostScript) utilize a scale factor, while others (e.g. PCL 5) utilize a dots-per-inch resolution. Also, some printers that utilize the resolution specification support only certain values for the resolution. For maximum compatibility, specify both members of this structure. The following table shows example values that will generate similarly sized output.
	USB_PRINTER_INTERFACE (see page 764)	USB Printer Interface Structure This structure represents the information needed to interface with a printer language. An array of these structures must be created in usb_config.c, so the USB printer client driver can determine what printer language to use to communicate with the printer.
	USB_PRINTER_SPECIFIC_INTERFACE (see page 769)	USB Printer Specific Interface Structure This structure is used to explicitly specify what printer language to use for a particular printer, and what print functions the printer supports. It can be used when a printer supports multiple languages with one language preferred over the others. It is required for printers that do not support the GET_DEVICE_ID printer class request. These printers do not report what printer languages they support. Typically, these printers also do not report Printer Class support in their Interface Descriptors, and must be explicitly supported by their VID and PID in the TPL. This structure links the... more (see page 769)

Types

	Name	Description
	USB_PRINTER_LANGUAGE_HANDLER (see page 765)	This is a typedef to use when defining a printer language command handler.
	USB_PRINTER_LANGUAGE_SUPPORTED (see page 766)	This is a typedef to use when defining a function that determines if the printer with the given "COMMAND SET:" portion of the device ID string supports the particular printer language.

Unions

	Name	Description
	USB_DATA_POINTER (see page 736)	This type is used to represent a generic RAM or ROM pointer when passed to the function USBHostPrinterCommand (see page 636)() or a printer language function of the type USB_PRINTER_LANGUAGE_HANDLER (see page 765). Note that the caller must indicate whether the point is actually pointing to RAM or to ROM, so we can tell which pointer is valid. Not all printer commands can actually use data in ROM. Refer to the specific printer command in the USB_PRINTER_COMMAND (see page 742) enumeration for more information.
	USB_PRINTER_FUNCTION_SUPPORT (see page 755)	Printer Device Function support Information This structure contains information about the functions that the attached printer supports. See the related constants for setting these flags via the val member: • USB_PRINTER_FUNCTION_SUPPORT_POS (see page 756) • USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS (see page 757)

	USB_PRINTER_GRAPHICS_PARAMETERS (see page 758)	USB Printer Graphics Parameter Structures This union can be used to declare a variable that can hold the parameters for any printer graphics or POS printer command (USB_PRINTER_COMMAND (see page 742)). The union allows a single variable to be declared and then reused for any printer graphics command.

Description

7.2.10.2.1 _USB_HOST_PRINTER_PRIMITIVES_H Macro

File

usb_host_printer_primitives.h

C

```
#define _USB_HOST_PRINTER_PRIMITIVES_H
```

Description

This is macro _USB_HOST_PRINTER_PRIMITIVES_H.

7.2.10.2.2 BARCODE_CODE128_CODESET_A_CHAR Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_A_CHAR 'A'
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE A. This code set should be used if control characters (0x00-0x1F) are included in the data.

7.2.10.2.3 BARCODE_CODE128_CODESET_A_STRING Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_A_STRING "A"
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE A. This code set should be used if control characters (0x00-0x1F) are included in the data.

7.2.10.2.4 BARCODE_CODE128_CODESET_B_CHAR Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_B_CHAR 'B'
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE B. This code set should be used if lower case letters and higher ASCII characters (0x60-0x7F) are included in the data.

7.2.10.2.5 BARCODE_CODE128_CODESET_B_STRING Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_B_STRING "B"
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE B. This code set should be used if lower case letters and higher ASCII characters (0x60-0x7F) are included in the data.

7.2.10.2.6 BARCODE_CODE128_CODESET_C_CHAR Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_C_CHAR 'C'
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE C. This code set can be used only if the data values are between 0 and 99 (0x00-0x63).

7.2.10.2.7 BARCODE_CODE128_CODESET_C_STRING Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_C_STRING "C"
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the second data byte of the bar code data to specify character code set CODE C. This code set can be used only if the data values are between 0 and 99 (0x00-0x63).

7.2.10.2.8 BARCODE_CODE128_CODESET_CHAR Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_CHAR '{'
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the first data byte of the bar code data, which begins the character code specification. The next byte must be 'A', 'B', or 'C'.

7.2.10.2.9 BARCODE_CODE128_CODESET_STRING Macro

File

usb_host_printer.h

C

```
#define BARCODE_CODE128_CODESET_STRING "{"
```

Description

For use with POS printers that support Code128 bar codes (extended barcodes). This is the value of the first data byte of the bar code data, which begins the character code specification. The next byte must be 'A', 'B', or 'C'.

7.2.10.2.10 BARCODE_TEXT_12x24 Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_12x24 1
```

Description

For use with POS printers. May not be valid for all printers. Print the bar code text in 12x24 dot font. Do not alter this value.

7.2.10.2.11 BARCODE_TEXT_18x36 Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_18x36 0
```

Description

For use with POS printers. May not be valid for all printers. Print the bar code text in 18x36 dot font. Do not alter this value.

7.2.10.2.12 BARCODE_TEXT_ABOVE Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_ABOVE 1
```

Description

For use with POS printers. Print readable text above the bar code. Do not alter this value.

7.2.10.2.13 BARCODE_TEXT_ABOVE_AND_BELOW Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_ABOVE_AND_BELOW 3
```

Description

For use with POS printers. Print readable text above and below the bar code. Do not alter this value.

7.2.10.2.14 BARCODE_TEXT_BELOW Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_BELOW 2
```

Description

For use with POS printers. Print readable text below the bar code. Do not alter this value.

7.2.10.2.15 BARCODE_TEXT_OMIT Macro

File

usb_host_printer.h

C

```
#define BARCODE_TEXT_OMIT 0
```

Description

For use with POS printers. Do not print readable bar code text. Do not alter this value.

7.2.10.2.16 EVENT_PRINTER_ATTACH Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_ATTACH (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+0)
```

Description

This event indicates that a Printer device has been attached. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, the *data parameter points to a structure of the type USB_PRINTER_DEVICE_ID (see page 751), which provides important information about the attached printer. The size parameter is the size of this structure.

7.2.10.2.17 EVENT_PRINTER_DETACH Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_DETACH (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+1)
```

Description

This event indicates that the specified device has been detached from the USB. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to a BYTE that contains the device address, and size is the size of a BYTE.

7.2.10.2.18 EVENT_PRINTER_OFFSET Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_OFFSET 0
```

Description

This is an optional offset for the values of the generated events. If necessary, the application can use a non-zero offset for the generic events to resolve conflicts in event number.

7.2.10.2.19 EVENT_PRINTER_REQUEST_DONE Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_REQUEST_DONE (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+4)
```

Description

This event indicates that the printer request has completed. These requests occur on endpoint 0 and include getting the printer status and performing a soft reset.

7.2.10.2.20 EVENT_PRINTER_REQUEST_ERROR Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_REQUEST_ERROR (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+7)
```

Description

This event indicates that a bus error occurred while trying to perform a device request. The error code is returned in the size parameter. The data parameter is returned as NULL.

7.2.10.2.21 EVENT_PRINTER_RX_DONE Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_RX_DONE (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+3)
```

Description

This event indicates that a previous read request has completed. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the receive buffer, and size is the actual number of bytes read from the device.

7.2.10.2.22 EVENT_PRINTER_RX_ERROR Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_RX_ERROR (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+6)
```

Description

This event indicates that a bus error occurred while trying to perform a read. The error code is returned in the size parameter. The data parameter is returned as NULL.

7.2.10.2.23 EVENT_PRINTER_TX_DONE Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_TX_DONE (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+2)
```

Description

This event indicates that a previous write request has completed. When USB_HOST_APP_EVENT_HANDLER (see page 292) is called with this event, *data points to the buffer that completed transmission, and size is the actual number of bytes that were written to the device.

7.2.10.2.24 EVENT_PRINTER_TX_ERROR Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_TX_ERROR (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+5)
```

Description

This event indicates that a bus error occurred while trying to perform a write. The error code is returned in the size parameter. The data parameter is returned as NULL.

7.2.10.2.25 EVENT_PRINTER_UNSUPPORTED Macro

File

usb_host_printer.h

C

```
#define EVENT_PRINTER_UNSUPPORTED (EVENT_PRINTER_BASE+EVENT_PRINTER_OFFSET+8)
```

Description

This event indicates that a printer has attached for which we do not have printer language support. Therefore, we cannot talk to this printer. This event can also occur if there is not enough dynamic memory available to read the device ID string.

7.2.10.2.26 LANGUAGE_ID_STRING_ESCPOS Macro

File

usb_host_printer_esc_pos.h

C

```
#define LANGUAGE_ID_STRING_ESCPOS "ESC"
```

Description

This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive. See the function USBHostPrinterLanguageESCPOSIsSupported (see page 648)() for more information about using or changing this string. Dynamic language determination is not recommended when using POS printers.

7.2.10.2.27 LANGUAGE_ID_STRING_PCL Macro

File

usb_host_printer_pcl_5.h

C

```
#define LANGUAGE_ID_STRING_PCL "PCL"
```

Description

This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive. Some printers that report only PCL 6 support also support PCL 5. So it is recommended to use "PCL" as the search string, rather than "PCL 5", and verify that the correct output is produced by the target printer.

7.2.10.2.28 LANGUAGE_ID_STRING_POSTSCRIPT Macro

File

usb_host_printer_postscript.h

C

```
#define LANGUAGE_ID_STRING_POSTSCRIPT "POSTSCRIPT"
```

Description

This is the string that the printer language support determination routine will look for to determine if the printer supports this printer language. This string is case sensive.

7.2.10.2.29 LANGUAGE_SUPPORT_FLAGS_ESCPOS Macro

File

usb_host_printer_esc_pos.h

C

```
#define LANGUAGE_SUPPORT_FLAGS_ESCPOS USB_PRINTER_FUNCTION_SUPPORT_POS
```

Description

These are the support flags that are set for this language.

7.2.10.2.30 LANGUAGE_SUPPORT_FLAGS_PCL3 Macro

File

usb_host_printer_pcl_5.h

C

```
#define LANGUAGE_SUPPORT_FLAGS_PCL3 0
```

Description

These are the support flags that are set for the PCL 3 version of this language.

7.2.10.2.31 LANGUAGE_SUPPORT_FLAGS_PCL5 Macro

File

usb_host_printer_pcl_5.h

C

```
#define LANGUAGE_SUPPORT_FLAGS_PCL5 USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS
```

Description

These are the support flags that are set for the PCL 5 version of this language.

7.2.10.2.32 LANGUAGE_SUPPORT_FLAGS_POSTSCRIPT Macro

File

usb_host_printer_postscript.h

C

```
#define LANGUAGE_SUPPORT_FLAGS_POSTSCRIPT USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS
```

Description

These are the support flags that are set for this language.

7.2.10.2.33 PRINTER_COLOR_BLACK Macro

File

usb_host_printer.h

C

```
#define PRINTER_COLOR_BLACK 0
```

Description

Indicates a black line for drawing graphics objects.

7.2.10.2.34 PRINTER_COLOR_WHITE Macro

File

usb_host_printer.h

C

```
#define PRINTER_COLOR_WHITE 1
```

Description

Indicates a white line for drawing graphics objects.

7.2.10.2.35 PRINTER_DEVICE_REQUEST_GET_DEVICE_ID Macro

File

usb_host_printer.h

C

```
#define PRINTER_DEVICE_REQUEST_GET_DEVICE_ID 0x00
```

Description

bRequest value for the GET_DEVICE_ID USB class-specific request.

7.2.10.2.36 PRINTER_DEVICE_REQUEST_GET_PORT_STATUS Macro

File

usb_host_printer.h

C

```
#define PRINTER_DEVICE_REQUEST_GET_PORT_STATUS 0x01
```

Description

bRequest value for the GET_PORT_STATUS USB class-specific request.

7.2.10.2.37 PRINTER_DEVICE_REQUEST_SOFT_RESET Macro

File

usb_host_printer.h

C

```
#define PRINTER_DEVICE_REQUEST_SOFT_RESET 0x02
```

Description

bRequest value for the SOFT_RESET USB class-specific request.

7.2.10.2.38 PRINTER_FILL_CROSS_HATCHED Macro

File

usb_host_printer.h

C

```
#define PRINTER_FILL_CROSS_HATCHED 3
```

Description

Indicates a cross-hatched fill for graphics objects. Requires a specified line spacing and angle.

7.2.10.2.39 PRINTER_FILL_HATCHED Macro

File

usb_host_printer.h

C

```
#define PRINTER_FILL_HATCHED 2
```

Description

Indicates a hatched fill for graphics objects. Requires a specified line spacing and angle.

7.2.10.2.40 PRINTER_FILL_SHADED Macro

File

usb_host_printer.h

C

```
#define PRINTER_FILL_SHADED 1
```

Description

Indicates a shaded fill for filled graphics objects. Requires a specified fill percentage.

7.2.10.2.41 PRINTER_FILL_SOLID Macro

File

usb_host_printer.h

C

```
#define PRINTER_FILL_SOLID 0
```

Description

Indicates a solid color fill for filled graphics objects.

7.2.10.2.42 PRINTER_LINE_END_BUTT Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_END_BUTT 0
```

Description

Drawn lines will have a butt end.

7.2.10.2.43 PRINTER_LINE_END_ROUND Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_END_ROUND 1
```

Description

Drawn lines will have a round end.

7.2.10.2.44 PRINTER_LINE_END_SQUARE Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_END_SQUARE 2
```

Description

Drawn lines will have a square end.

7.2.10.2.45 PRINTER_LINE_JOIN_BEVEL Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_JOIN_BEVEL 0
```

Description

Drawn lines will be joined with a bevel.

7.2.10.2.46 PRINTER_LINE_JOIN_MITER Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_JOIN_MITER 1
```

Description

Drawn lines will be joined with a miter.

7.2.10.2.47 PRINTER_LINE_JOIN_ROUND Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_JOIN_ROUND 2
```

Description

Drawn lines will be joined with a round.

7.2.10.2.48 PRINTER_LINE_TYPE_DASHED Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_TYPE_DASHED 1
```

Description

Indicates a dashed line for drawing graphics objects.

7.2.10.2.49 PRINTER_LINE_TYPE_DOTTED Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_TYPE_DOTTED 2
```

Description

Indicates a dotted line for drawing graphics objects.

7.2.10.2.50 PRINTER_LINE_TYPE_SOLID Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_TYPE_SOLID 0
```

Description

Indicates a solid line for drawing graphics objects.

7.2.10.2.51 PRINTER_LINE_WIDTH_NORMAL Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_WIDTH_NORMAL 0
```

Description

Indicates a normal width line for drawing graphics objects.

7.2.10.2.52 PRINTER_LINE_WIDTH_THICK Macro

File

usb_host_printer.h

C

```
#define PRINTER_LINE_WIDTH_THICK 1
```

Description

Indicates a thick line for drawing graphics objects.

7.2.10.2.53 PRINTER_PAGE_LANDSCAPE_HEIGHT Macro

File

usb_host_printer.h

C

```
#define PRINTER_PAGE_LANDSCAPE_HEIGHT 612
```

Description

The height of the page in points when in landscape mode.

7.2.10.2.54 PRINTER_PAGE_LANDSCAPE_WIDTH Macro

File

usb_host_printer.h

C

```
#define PRINTER_PAGE_LANDSCAPE_WIDTH 792
```

Description

The width of the page in points when in landscape mode.

7.2.10.2.55 PRINTER_PAGE_PORTRAIT_HEIGHT Macro

File

usb_host_printer.h

C

```
#define PRINTER_PAGE_PORTRAIT_HEIGHT 792
```

Description

The height of the page in points when in portrait mode.

7.2.10.2.56 PRINTER_PAGE_PORTRAIT_WIDTH Macro

File

usb_host_printer.h

C

```
#define PRINTER_PAGE_PORTRAIT_WIDTH 612
```

Description

The width of the page in points when in portrait mode.

7.2.10.2.57 PRINTER_POS_BOTTOM_TO_TOP Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_BOTTOM_TO_TOP 1
```

Description

POS print direction bottom to top, starting at the bottom left corner.

7.2.10.2.58 PRINTER_POS_DENSITY_HORIZONTAL_DOUBLE Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_DENSITY_HORIZONTAL_DOUBLE 2
```

Description

Image print with double horizontal density.

7.2.10.2.59 PRINTER_POS_DENSITY_HORIZONTAL_SINGLE Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_DENSITY_HORIZONTAL_SINGLE 1
```

Description

Image print with single horizontal density.

7.2.10.2.60 PRINTER_POS_DENSITY_VERTICAL_24 Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_DENSITY_VERTICAL_24 24
```

Description

Image print with 24-dot vertical density.

7.2.10.2.61 PRINTER_POS_DENSITY_VERTICAL_8 Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_DENSITY_VERTICAL_8 8
```

Description

Image print with 8-dot vertical density.

7.2.10.2.62 PRINTER_POS_LEFT_TO_RIGHT Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_LEFT_TO_RIGHT 0
```

Description

POS print direction left to right, starting at the top left corner.

7.2.10.2.63 PRINTER_POS_RIGHT_TO_LEFT Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_RIGHT_TO_LEFT 2
```

Description

POS print direction right to left, startin at the bottom right corner.

7.2.10.2.64 PRINTER_POS_TOP_TO_BOTTOM Macro

File

usb_host_printer.h

C

```
#define PRINTER_POS_TOP_TO_BOTTOM 3
```

Description

POS print direction top to bottom, starting at the top right corner.

7.2.10.2.65 USB_DATA_POINTER Union

File

usb_host_printer.h

C

```
typedef union {
  void * pointerRAM;
  const void * pointerROM;
} USB_DATA_POINTER;
```

Members

Members	Description
void * pointerRAM;	Pointer to data in RAM.
const void * pointerROM;	Pointer to data in ROM.

Description

This type is used to represent a generic RAM or ROM pointer when passed to the function USBHostPrinterCommand (see page 636)() or a printer language function of the type USB_PRINTER_LANGUAGE_HANDLER (see page 765). Note that the caller must indicate whether the point is actually pointing to RAM or to ROM, so we can tell which pointer is valid. Not all printer commands can actually use data in ROM. Refer to the specific printer command in the USB_PRINTER_COMMAND (see page 742) enumeration for more information.

7.2.10.2.66 USB_DATA_POINTER_RAM Macro

File

usb_host_printer.h

C

```
#define USB_DATA_POINTER_RAM(x) ((USB_DATA_POINTER)(void *)x)
```

Description

Use this definition to cast a pointer being passed to the function USBHostPrinterCommand (see page 636)() that points to data in RAM.

7.2.10.2.67 USB_DATA_POINTER_ROM Macro

File

usb_host_printer.h

C

```
#define USB_DATA_POINTER_ROM(x) ((USB_DATA_POINTER)(const void *)x)
```

Description

Use this definition to cast a pointer being passed to the function USBHostPrinterCommand (see page 636)() that points to data in ROM.

7.2.10.2.68 USB_MAX_PRINTER_DEVICES Macro

File

usb_host_printer.h

C

```
#define USB_MAX_PRINTER_DEVICES 1
```

Description

Max Number of Supported Devices

This value represents the maximum number of attached devices this class driver can support. If the user does not define a value, it will be set to 1. Currently this must be set to 1, due to limitations in the USB Host layer.

7.2.10.2.69 USB_NULL Macro

File

usb_host_printer.h

C

```
#define USB_NULL (USB_DATA_POINTER)(void *)NULL
```

Description

Use this definition to pass a NULL pointer to the function USBHostPrinterCommand (see page 636)().

7.2.10.2.70 USB_PRINT_SCREEN_INFO Structure

File

usb_host_printer_primitives.h

C

```
typedef struct {
  WORD xL;
  WORD yT;
  WORD xR;
  WORD yB;
  WORD colorBlack;
  USB_PRINTER_FUNCTION_SUPPORT printerType;
  USB_PRINTER_IMAGE_INFO printerInfo;
} USB_PRINT_SCREEN_INFO;
```

Members

Members	Description
WORD xL;	X-axis position of the left side of the screen image.
WORD yT;	Y-axis position of the top of the screen image.
WORD xR;	X-axis position of the right side of the screen image.
WORD yB;	Y-axis position of the bottom of the screen image.
WORD colorBlack;	Screen color that should be printed as black.
USB_PRINTER_FUNCTION_SUPPORT printerType;	The capabilities of the current printer, so we know what structure members are valid.
USB_PRINTER_IMAGE_INFO printerInfo;	Store all the info needed to print the image. The width and height parameters will be determined by the screen coordinates specified above. The application must provide the other values.

Description

Print Screen Information

This structure is designed for use when the USB Embedded Host Printer support is integrated with the graphics library. The structure contains the information needed to print a portion of the graphics screen as a bitmapped graphic image.

7.2.10.2.71 USB_PRINTER_COMMAND Enumeration

File

usb_host_printer.h

C

```
typedef enum {
  USB_PRINTER_ATTACHED,
  USB_PRINTER_DETACHED,
  USB_PRINTER_TRANSPARENT,
  USB_PRINTER_JOB_START,
  USB_PRINTER_JOB_STOP,
  USB_PRINTER_ORIENTATION_PORTRAIT,
  USB_PRINTER_ORIENTATION_LANDSCAPE,
  USB_PRINTER_FONT_NAME,
  USB_PRINTER_FONT_SIZE,
  USB_PRINTER_FONT_ITALIC,
  USB_PRINTER_FONT_UPRIGHT,
  USB_PRINTER_FONT_BOLD,
  USB_PRINTER_FONT_MEDIUM,
  USB_PRINTER_EJECT_PAGE,
  USB_PRINTER_TEXT_START,
  USB_PRINTER_TEXT,
  USB_PRINTER_TEXT_STOP,
  USB_PRINTER_SET_POSITION,
  USB_PRINTER_IMAGE_START,
  USB_PRINTER_IMAGE_DATA_HEADER,
  USB_PRINTER_IMAGE_DATA,
  USB_PRINTER_IMAGE_STOP,
  USB_PRINTER_VECTOR_GRAPHICS_START,
  USB_PRINTER_GRAPHICS_LINE_TYPE,
  USB_PRINTER_GRAPHICS_LINE_WIDTH,
  USB_PRINTER_GRAPHICS_LINE_END,
  USB_PRINTER_GRAPHICS_LINE_JOIN,
  USB_PRINTER_GRAPHICS_FILL_TYPE,
  USB_PRINTER_GRAPHICS_COLOR,
  USB_PRINTER_GRAPHICS_MOVE_TO,
  USB_PRINTER_GRAPHICS_MOVE_RELATIVE,
  USB_PRINTER_GRAPHICS_LINE,
  USB_PRINTER_GRAPHICS_LINE_TO,
  USB_PRINTER_GRAPHICS_LINE_TO_RELATIVE,
  USB_PRINTER_GRAPHICS_ARC,
  USB_PRINTER_GRAPHICS_CIRCLE,
  USB_PRINTER_GRAPHICS_CIRCLE_FILLED,
  USB_PRINTER_GRAPHICS_BEVEL,
  USB_PRINTER_GRAPHICS_BEVEL_FILLED,
  USB_PRINTER_GRAPHICS_RECTANGLE,
  USB_PRINTER_GRAPHICS_RECTANGLE_FILLED,
  USB_PRINTER_GRAPHICS_POLYGON,
  USB_PRINTER_VECTOR_GRAPHICS_END,
  USB_PRINTER_POS_START,
  USB_PRINTER_POS_PAGE_MODE,
  USB_PRINTER_POS_STANDARD_MODE,
  USB_PRINTER_POS_FEED,
  USB_PRINTER_POS_TEXT_LINE,
  USB_PRINTER_POS_CUT,
  USB_PRINTER_POS_CUT_PARTIAL,
  USB_PRINTER_POS_JUSTIFICATION_CENTER,
  USB_PRINTER_POS_JUSTIFICATION_LEFT,
  USB_PRINTER_POS_JUSTIFICATION_RIGHT,
  USB_PRINTER_POS_FONT_REVERSE,
  USB_PRINTER_POS_FONT_UNDERLINE,
  USB_PRINTER_POS_COLOR_BLACK,
  USB_PRINTER_POS_COLOR_RED,
  USB_PRINTER_POS_BARCODE,
  USB_PRINTER_POS_END
} USB_PRINTER_COMMAND;
```

Members

Members	Description
USB_PRINTER_ATTACHED	This command is used internally by the printer client driver. Applications do not issue this command. This command informs the language support code that a new device has attached. Some language support requires the maintenance of certain information about the printering status. This command, with the USB_PRINTER_DETACHED command, allows the language support information to be maintained properly as printers are attached and detached. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_DETACHED	This command is used internally by the printer client driver. Applications do not issue this command. This command informs the language support code that a device has detached. Some language support requires the maintenance of certain information about the printering status. This command, with the USB_PRINTER_ATTACHED command, allows the language support information to be maintained properly as printers are attached and detached. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_TRANSPARENT	This command instructs the printer driver to send the buffer directly to the printer, without interpretation by the printer driver. This is normally used only when debugging new commands. The data parameter should point to the data to be sent, and size should indicate the number of bytes to send. This command supports sending data from either RAM or ROM. If the data is in ROM, be sure to set the USB_PRINTER_TRANSFER_FROM_ROM (see page 772) flag. If the data is in RAM but the application may overwrite it, set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag to tell the printer client driver to make a local copy of the data, allowing the application to overwrite the original buffer when USBHostPrinterCommand (see page 636)() terminates.
USB_PRINTER_JOB_START	This command should be issued at the beginning of every print job. It ensures that the printer is set back to a default state. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_JOB_STOP	This command should be issued at the end of every print job. It ejects the currently printing page, and ensures that the printer is set back to a default state. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_ORIENTATION_PORTRAIT	This command sets the current page orientation to portrait. This command must be issued immediately after the USB_PRINTER_JOB_START and USB_PRINTER_EJECT_PAGE commands in order for the command to take effect properly. Only one orientation command should be sent per page, or the output may not be properly generated. The default orientation is portrait. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_ORIENTATION_LANDSCAPE	This command sets the current page orientation to landscape. This command must be issued immediately after the USB_PRINTER_JOB_START and USB_PRINTER_EJECT_PAGE commands in order for the command to take effect properly. Only one orientation command should be sent per page, or the output may not be properly generated. The default orientation is portrait. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.

USB_PRINTER_FONT_NAME	This command selects the text font. To make usage easier, the size parameter is used to hold the font name indication. The data pointer should be passed in as USB_NULL (see page 740). Refer to the enums USB_PRINTER_FONTS (see page 753) and USB_PRINTER_FONTS_POS (see page 754) for the valid values for the font name. With POS printers, the font name also indicates the font size.
USB_PRINTER_FONT_SIZE	(Full sheet printers only.) This command selects the font size in terms of points. To make usage easier, the size parameter is used to hold the font size. The data pointer should be passed in as USB_NULL (see page 740). For POS printers, the size is specified as a scale factor. The value of bits [3:0] plus one is the vertical scale, and the value of bits [7:4] plus one is the horizontal scale. Each direction can be scaled a maximum of x10. For example, the value 0x00 is x1 scaling in both directions, and 0x95 is x10 scaling horizontally and x6 scaling vertically.
USB_PRINTER_FONT_ITALIC	This command sets the current font to italic. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_FONT_UPRIGHT	This command sets the current font to upright (not italic). The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_FONT_BOLD	This command sets the current font to bold. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_FONT_MEDIUM	This command sets the current font to regular weight (not bold). The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_EJECT_PAGE	This command ejects the currently printing page. The command USB_PRINTER_JOB_STOP will also eject the page. After this command, the selected paper orientation (portrait or landscape) and selected font must be reset. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_TEXT_START	This command initiates a text print. To print text, first issue a USB_PRINTER_TEXT_START command. Then issue a USB_PRINTER_TEXT command with the text to be printed, setting the transferFlags parameter correctly for the location of the source text (RAM, ROM, or external memory. Finally, use the USB_PRINTER_TEXT_STOP command to terminate the text print. For best compatibility across printers, do not insert other commands into this sequence. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_TEXT	This command specifies text to print. The data parameter should point to the buffer of data to send, and size should indicate how many bytes of data to print. This command supports printing text from either RAM or ROM. Be sure to set the transferFlags parameter correctly for the location of the data source. If the data is in RAM but the application may overwrite it, set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag to tell the printer client driver to make a local copy of the data, allowing the application to overwrite the original buffer when USBHostPrinterCommand (see page 636)() terminates. Refer to the USB_PRINTER_TEXT_START command for the sequence required to print text.

USB_PRINTER_TEXT_STOP	This command terminates a text print. Refer to the USB_PRINTER_TEXT_START command for the sequence required to print text. The data and size parameters are not used by this command, and can be passed as USB_NULL (see page 740) and 0 respectively. For POS printers, size is the number of lines to feed after the print. To get the best result, a minimum of one line is recommended. The data parameter is not used, and can be passed as USB_NULL (see page 740).
USB_PRINTER_SET_POSITION	This command sets the current printing position on the page. Refer to the documentation for a description of the page coordinates. Both X and Y coordinates are passed in the size parameter. The X coordinate is passed in the most significant (upper) WORD, and the Y coordinate is passed in the least significant (lower) WORD. The macro USBHostPrinterPosition (see page 657)(X, Y) can be used to create the paramater. POS printers support specifying the Y-axis position while in page mode only. The data pointer should be passed in as USB_NULL (see page 740).
USB_PRINTER_IMAGE_START	This command is used to initialize the printing of a bitmapped image. This command requires a pointer to a variable of type USB_PRINTER_IMAGE_INFO (see page 762). To print a bitmapped image, obtain the information required by the USB_PRINTER_IMAGE_INFO (see page 762) structure, and issue this command. Each row of bitmapped data can now be sent to the printer. For each row, first issue the USB_PRINTER_IMAGE_DATA_HEADER command. Then issue the USB_PRINTER_IMAGE_DATA command, with the transferFlags parameter set appropriately for the location of the bitmapped data. After all rows of data have been sent, terminate the image print with the USB_PRINTER_IMAGE_STOP command. Be sure that adequate heap space is available, particularly when printing from ROM or external memory, and when printing to a POS printer. When printing images on POS printers, ensure that the dot density capabilities of the printer are set correctly. If they are not, the printer will print garbage characters. Refer to the Printer Client Driver section of the Help file for more information about printing images.
USB_PRINTER_IMAGE_DATA_HEADER	This command is issued before each row of bitmapped image data. The size parameter is the width of the image in terms of pixels. The *data parameter is not used and should be passed in as USB_NULL (see page 740). Refer to the USB_PRINTER_IMAGE_START command for the sequence required to print an image. When printing images on POS printers, ensure that the dot density capabilities of the printer are set correctly. If they are not, the printer will print garbage characters.
USB_PRINTER_IMAGE_DATA	This command is issued for each row of bitmapped image data. The *data parameter should point to the data, and size should be the number of bits of data to send to the printer, which should match the value passed in the USB_PRINTER_IMAGE_DATA_HEADER command. This command supports reading image data from either RAM or ROM. Be sure to set the transferFlags parameter appropriately to indicate the location of the bitmapped data. If the data is in RAM but the application may overwrite it, set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag to tell the printer client driver to make a local copy of the data, allowing the application to overwrite the original buffer when USBHostPrinterCommand (see page 636)() terminates. Refer to the USB_PRINTER_IMAGE_START command for the sequence required to print an image. Be sure that adequate heap space is available, particularly when printing from ROM or external memory, and when printing to a POS printer. When printing images on POS printers, ensure that the dot density capabilities of the printer are set correctly. If they are not, the printer will print garbage characters. Refer to the Printer Client Driver section of the Help file for more information about printing images.

USB_PRINTER_IMAGE_STOP	This command is used to terminate printing a bitmapped image. Refer to the USB_PRINTER_IMAGE_START command for the sequence required to print an image.
USB_PRINTER_VECTOR_GRAPHICS_START	Commands between USB_PRINTER_VECTOR_GRAPHICS_START and USB_PRINTER_VECTOR_GRAPHICS_END are valid only with printers that support vector graphics. This support is determined by the interface function of the type USB_PRINTER_LANGUAGE_SUPPORTED (see page 766) that is specified in the usb_config.c configuration file.
USB_PRINTER_GRAPHICS_LINE_TYPE	(Vector graphics support required.) This command sets the line type for drawing graphics. The line type indication is passed in the size parameter. Valid values are PRINTER_LINE_TYPE_SOLID (see page 721), PRINTER_LINE_TYPE_DOTTED (see page 720), and PRINTER_LINE_TYPE_DASHED (see page 719). The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_LINE_WIDTH	(Vector graphics support required.) This command sets the width of the line for drawing vector graphics. The width indication is passed in the size parameter. For full sheet printers, valid values are PRINTER_LINE_WIDTH_NORMAL (see page 722) and PRINTER_LINE_WIDTH_THICK (see page 723). For POS printers, the size is specified in dots (1-255). The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_LINE_END	(Vector graphics support required.) This command sets the style of the end of the lines used for drawing vector graphics. The style indication is passed in the size parameter. Valid values are PRINTER_LINE_END_BUTT (see page 713), PRINTER_LINE_END_ROUND (see page 714), and PRINTER_LINE_END_SQUARE (see page 715). The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_LINE_JOIN	(Vector graphics support required.) This commands sets the style of how lines are joined when drawing vector graphics. The style indication is passed in the size parameter. Valid values are PRINTER_LINE_JOIN_BEVEL (see page 716), PRINTER_LINE_JOIN_MITER (see page 717), and PRINTER_LINE_JOIN_ROUND (see page 718). The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_FILL_TYPE	(Vector graphics support required.) This command sets the fill type for drawing filled vector graphics. The data pointer should point to a data structure that matches the sFillType structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. Valid values for the fillType member are: • PRINTER_FILL_SOLID (see page 712). Other structure members are ignored. • PRINTER_FILL_SHADED (see page 711). 0 <= shading <= 100 • PRINTER_FILL_HATCHED (see page 710). The spacing is specified in points, angle is specified in degrees. • PRINTER_FILL_CROSS_HATCHED (see page 709). The spacing is specified in points, angle is specified in degrees.
USB_PRINTER_GRAPHICS_COLOR	(Vector graphics support required.) This command sets the color of the line for drawing vector graphics. The color indication is passed in the size parameter. Valid values are PRINTER_COLOR_BLACK (see page 704) and PRINTER_COLOR_WHITE (see page 705). The data pointer parameter is not used and should be set to USB_NULL (see page 740).

USB_PRINTER_GRAPHICS_MOVE_TO	(Vector graphics support required.) This command moves the graphics pen to the specified position. The position is specified in terms of points. The X-axis position value is passed in the most significant word of the size parameter, and the Y-axis position value is passed in the least significant word of the size parameter. POS printers support specifying the Y-axis position while in page mode only. The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_MOVE_RELATIVE	(Vector graphics support required.) This command moves the graphics pen to the specified relative position. The change in position is specified in terms of points. The X-axis position change is passed in the most significant word of the size parameter, and the Y-axis position change is passed in the least significant word of the size parameter. POS printers do not support specifying the Y-axis position.
USB_PRINTER_GRAPHICS_LINE	(Vector graphics support required.) This command draws a line from one specified x,y position to another specified x,y position. The data pointer should point to a data structure that matches the sLine structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_LINE_TO	(Vector graphics support required.) This command draws a line from the current x,y position to the specified x,y position. The new x,y position is passed in the size parameter. The X-axis position value is passed in the most significant word of the size parameter, and the Y-axis position value is passed in the least significant word of the size parameter. The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_LINE_TO_RELATIVE	(Vector graphics support required.) This command draws a line from the current x,y position to the x,y position defined by the indicated displacement. The x and y displacements are passed in the size parameter. The X-axis displacement is passed in the most significant word of the size parameter, and the Y-axis displacement is passed in the least significant word of the size parameter. The data pointer parameter is not used and should be set to USB_NULL (see page 740).
USB_PRINTER_GRAPHICS_ARC	(Vector graphics support required.) This command draws an arc, or a piece of a circle. The data pointer should point to a data structure that matches the sArc structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. This command can print only one arc of a circle, unlike the Graphics library, which can print multiple separated arcs of the same circle.
USB_PRINTER_GRAPHICS_CIRCLE	(Vector graphics support required.) This command draws a circle using the current pen color and width. The inside of the circle is not filled. To draw a filled circle, use the command USB_PRINTER_GRAPHICS_CIRCLE_FILLED. The data pointer should point to a data structure that matches the sCircle structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_CIRCLE_FILLED	(Vector graphics support required.) This command draws a filled circle using the current pen color. To draw the outline of a circle, use the command USB_PRINTER_GRAPHICS_CIRCLE_FILLED. The data pointer should point to a data structure that matches the sCircle structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_BEVEL	(Vector graphics support required.) This command draws an outlined bevel (rectangle with rounded corners) using the current pen color and width. The inside of the bevel is not filled. To draw a filled bevel, use the command USB_PRINTER_GRAPHICS_BEVEL_FILLED. The data pointer should point to a data structure that matches the sBevel structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.

USB_PRINTER_GRAPHICS_BEVEL_FILLED	(Vector graphics support required.) This command draws a filled bevel using the current pen color. To draw the outline of a bevel, use the command USB_PRINTER_GRAPHICS_BEVEL_FILLED. The data pointer should point to a data structure that matches the sBevel structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_RECTANGLE	(Vector graphics support required.) This command draws a rectangle using the current pen color and width. The inside of the rectangle is not filled. To draw a filled rectangle, use the command USB_PRINTER_GRAPHICS_RECTANGLE_FILLED. The data pointer should point to a data structure that matches the sRectangle structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_RECTANGLE_FILLED	(Vector graphics support required.) This command draws a filled rectangle using the current pen color. To draw the outline of a rectangle, use the command USB_PRINTER_GRAPHICS_RECTANGLE. The data pointer should point to a data structure that matches the sRectangle structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union.
USB_PRINTER_GRAPHICS_POLYGON	(Vector graphics support required.) This command draws the outline of a polygon with a specified number of sides, using the current pen color and width. The data pointer should point to a data structure that matches the sPolygon structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. This structure contains the number of verticies of the polygon and a pointer to an array containing x,y coordinates of the verticies. Line segments are drawn to each vertex in the order that they appear in the array.
USB_PRINTER_VECTOR_GRAPHICS_END	Commands between USB_PRINTER_VECTOR_GRAPHICS_START and USB_PRINTER_VECTOR_GRAPHICS_END are valid only with printers that support vector graphics. This support is determined by the interface function of the type USB_PRINTER_LANGUAGE_SUPPORTED (see page 766) that is specified in the usb_config.c configuration file.
USB_PRINTER_POS_START	Commands between USB_PRINTER_POS_START and USB_PRINTER_POS_END are valid only with point-of-sale printers. This support is determined by the interface function of the type USB_PRINTER_LANGUAGE_SUPPORTED (see page 766) that is specified in the usb_config.c configuration file.
USB_PRINTER_POS_PAGE_MODE	(POS support required.) This command sets the printer into page mode. In this mode, print commands are retained by the printer until it receives the USB_PRINTER_EJECT_PAGE command. This allows the application to create more sophisticated output. The data pointer should point to a data struture that matches the sPage structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. This structure contains the horizontal and vertical starting point, the horizontal and vertical print length, and the print direction and starting point. Valid values for the print direction and starting point are: • PRINTER_POS_LEFT_TO_RIGHT (see page 733) • PRINTER_POS_BOTTOM_TO_TOP (see page 728) • PRINTER_POS_RIGHT_TO_LEFT (see page 734) • PRINTER_POS_TOP_TO_BOTTOM (see page 735)
USB_PRINTER_POS_STANDARD_MODE	(POS support required.) This command sets the printer into standard mode. In this mode, print commands are processed and printed immediately. This is typically the default mode for a POS printer.

USB_PRINTER_POS_FEED	(POS support required.) This command feeds the specified number of lines, as dictated by the size parameter (between 0 and 255). The data parameter is not used, and should be set to USB_NULL (see page 740).
USB_PRINTER_POS_TEXT_LINE	(POS support required.) This command is a simplified method of printing a text line to a POS printer. This command prints a single, null terminated string and feeds a specified number of lines after the print. The data pointer must point to a null terminated string located in RAM. Printing strings from ROM is not supported. The size parameter should be set to the number of lines to feed after the text is printed.
USB_PRINTER_POS_CUT	(POS support required.) This command cuts the paper completely. The data parameter is not used, and should be passed as USB_NULL (see page 740). The least significant byte of the size parameter indicates the number of vertical motion units (printer dependent, typical values are 1/360 inch to 1/144 inch) to feed before the cut. Not all POS printer models support this command.
USB_PRINTER_POS_CUT_PARTIAL	(POS support required.) This command cuts the paper, leaving one point uncut. The data parameter is not used, and should be passed as USB_NULL (see page 740). The least significant byte of the size parameter indicates the number of vertical motion units (printer dependent, typical values are 1/360 inch to 1/144 inch) to feed before the cut. Not all POS printer models support this command.
USB_PRINTER_POS_JUSTIFICATION_CENTER	(POS support required.) This command sets the printing justification to the center of the print area. The data and size parameters are not used, and should be set to USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_POS_JUSTIFICATION_LEFT	(POS support required.) This command sets the printing justification to the left side of the print area. The data and size parameters are not used, and should be set to USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_POS_JUSTIFICATION_RIGHT	(POS support required.) This command sets the printing justification to the right side of the print area. The data and size parameters are not used, and should be set to USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_POS_FONT_REVERSE	(POS support required.) This command enables or disables white/black reverse printing of characters. When enabled, characters are printed in white on a black background, and underlining is not performed. To enable reverse printing, set the size parameter to 1. To disable reverse printing, set the size parameter to 0. (Only the least significant bit of the size parameter is examined.) The data parameter is not used, and should be set to USB_NULL (see page 740). Not all POS printer models support this command.
USB_PRINTER_POS_FONT_UNDERLINE	(POS support required.) This command enables or disables underlining. Underlining is not performed if reverse printing is enabled. To enable underlining, set the size parameter to 1. To disable underlining, set the size parameter to 0. (Only the least significant bit of the size parameter is examined.) The data parameter is not used, and should be set to USB_NULL (see page 740).
USB_PRINTER_POS_COLOR_BLACK	(POS support required.) This command changes the print color to black. This command is available only with printers that support two color printing. The data and size parameters are not used, and should be set to USB_NULL (see page 740) and 0 respectively.
USB_PRINTER_POS_COLOR_RED	(POS support required.) This command changes the print color to red. This command is available only with printers that support two color printing. The data and size parameters are not used, and should be set to USB_NULL (see page 740) and 0 respectively.

USB_PRINTER_POS_BARCODE	(POS support required.) This command prints a bar code. Not all POS printers provide bar code support, and the types of bar codes supported may vary; check the technical documentation for the desired target printer. The data pointer should point to a data structure that matches the sBarCode structure in the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. This structure contains the type of bar code (as specified by the USB_PRINTER_POS_BARCODE_FORMAT (see page 767) enumeration) and the bar code data.
USB_PRINTER_POS_END	Commands between USB_PRINTER_POS_START and USB_PRINTER_POS_END are valid only with point-of-sale printers. This support is determined by the interface function of the type USB_PRINTER_LANGUAGE_SUPPORTED (see page 766) that is specified in the usb_config.c configuration file.

Description

USB Printer Client Driver Commands

The main interface to the USB Printer Client Driver is through the function USBHostPrinterCommand (see page 636)(). These are the commands that can be passed to that function.

7.2.10.2.72 USB_PRINTER_DEVICE_ID Structure

File

usb_host_printer.h

C

```
typedef struct _USB_PRINTER_DEVICE_ID {
  WORD vid;
  WORD pid;
  USB_PRINTER_FUNCTION_SUPPORT support;
  BYTE deviceAddress;
} USB_PRINTER_DEVICE_ID;
```

Members

Members	Description
WORD vid;	Vendor ID of the device
WORD pid;	Product ID of the device
USB_PRINTER_FUNCTION_SUPPORT support;	Function support flags.
BYTE deviceAddress;	Address of the device on the USB

Description

Printer Device ID Information

This structure contains identification information about an attached device.

7.2.10.2.73 USB_PRINTER_ERRORS Enumeration

File

usb_host_printer.h

C

```
typedef enum {
  USB_PRINTER_SUCCESS = 0,
  USB_PRINTER_BUSY = USB_ERROR_CLASS_DEFINED,
  USB_PRINTER_UNKNOWN_COMMAND,
  USB_PRINTER_UNKNOWN_DEVICE,
  USB_PRINTER_OUT_OF_MEMORY,
  USB_PRINTER_TOO_MANY_DEVICES,
  USB_PRINTER_BAD_PARAMETER
} USB_PRINTER_ERRORS;
```

Members

Members	Description
USB_PRINTER_SUCCESS = 0	The command was successful.
USB_PRINTER_BUSY = USB_ERROR_CLASS_DEFINED	The command cannot be performed because the printer client driver's command queue is full. Use the function USBHostPrinterCommandReady (see page 638)() to determine if there is space available in the queue.
USB_PRINTER_UNKNOWN_COMMAND	An invalid printer command was requested. Refer to the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands.
USB_PRINTER_UNKNOWN_DEVICE	A device with the indicated address is not attached or is not a printer.
USB_PRINTER_OUT_OF_MEMORY	Not enough free heap space is available to perform the command.
USB_PRINTER_TOO_MANY_DEVICES	The number of attached printers exceeds the maximum specified by USB_MAX_PRINTER_DEVICES (see page 739). Refer to the USB configuration tool.
USB_PRINTER_BAD_PARAMETER	An invalid or out of range parameter was passed. Run time checking of graphics coordinates must be enabled by defining PRINTER_GRAPHICS_COORDINATE_CHECKING.

Description

Printer Errors

These are errors that can be returned by the printer client driver. Note that USB Embedded Host errors can also be returned.

7.2.10.2.74 USB_PRINTER_FONTS Enumeration

File

usb_host_printer.h

C

```
typedef enum {
  USB_PRINTER_FONT_AVANT_GARDE = 0,
  USB_PRINTER_FONT_BOOKMAN,
  USB_PRINTER_FONT_COURIER,
  USB_PRINTER_FONT_HELVETICA,
  USB_PRINTER_FONT_HELVETICA_NARROW,
  USB_PRINTER_FONT_NEW_CENTURY_SCHOOLBOOK,
  USB_PRINTER_FONT_PALATINO,
  USB_PRINTER_FONT_TIMES_NEW_ROMAN,
  USB_PRINTER_FONT_MAX_FONT
} USB_PRINTER_FONTS;
```

Members

Members	Description
USB_PRINTER_FONT_AVANT_GARDE = 0	Avant Garde font
USB_PRINTER_FONT_BOOKMAN	Bookman font
USB_PRINTER_FONT_COURIER	Courier font
USB_PRINTER_FONT_HELVETICA	Helvetica font
USB_PRINTER_FONT_HELVETICA_NARROW	Helvetica Narrow font
USB_PRINTER_FONT_NEW_CENTURY_SCHOOLBOOK	New Century Schoolbook font
USB_PRINTER_FONT_PALATINO	Palatino font
USB_PRINTER_FONT_TIMES_NEW_ROMAN	Times New Roman font
USB_PRINTER_FONT_MAX_FONT	Font out of range

Description

Printer Fonts

This enumeration defines the various printer fonts. If new fonts are added, they must be added at the end of the list, just before the USB_PRINTER_FONT_MAX_FONT definition, as the printer language support files may utilize them for indexing purposes.

7.2.10.2.75 USB_PRINTER_FONTS_POS Enumeration

File

usb_host_printer.h

C

```
typedef enum {
  USB_PRINTER_FONT_POS_18x36,
  USB_PRINTER_FONT_POS_18x72,
  USB_PRINTER_FONT_POS_36x36,
  USB_PRINTER_FONT_POS_36x72,
  USB_PRINTER_FONT_POS_12x24,
  USB_PRINTER_FONT_POS_12x48,
  USB_PRINTER_FONT_POS_24x24,
  USB_PRINTER_FONT_POS_24x48,
  USB_PRINTER_FONT_POS_MAX_FONT
} USB_PRINTER_FONTS_POS;
```

Members

Members	Description
USB_PRINTER_FONT_POS_18x36	Character size 18x36
USB_PRINTER_FONT_POS_18x72	Character size 18x36, double height
USB_PRINTER_FONT_POS_36x36	Character size 18x36, double width
USB_PRINTER_FONT_POS_36x72	Character size 18x36, double height and width
USB_PRINTER_FONT_POS_12x24	Character size 12x24
USB_PRINTER_FONT_POS_12x48	Character size 12x24, double height
USB_PRINTER_FONT_POS_24x24	Character size 12x24, double width
USB_PRINTER_FONT_POS_24x48	Character size 12x24, double height and width
USB_PRINTER_FONT_POS_MAX_FONT	Font out of range

Description

POS Printer Fonts

This enumeration defines the various printer fonts used by POS printers. If new fonts are added, they must be added at the end of the list, just before the USB_PRINTER_FONT_POS_MAX_FONT definition, as the printer language support files may utilize them for indexing purposes.

7.2.10.2.76 USB_PRINTER_FUNCTION_SUPPORT Union

File

usb_host_printer.h

C

```
typedef union {
  WORD val;
  struct {
    WORD supportsVectorGraphics : 1;
    WORD supportsPOS : 1;
  } supportFlags;
} USB_PRINTER_FUNCTION_SUPPORT;
```

Members

Members	Description
WORD val;	The WORD representation of the support flags.
struct { WORD supportsVectorGraphics : 1; WORD supportsPOS : 1; } supportFlags;	Various printer function support flags.
WORD supportsVectorGraphics : 1;	The printer supports vector graphics.
WORD supportsPOS : 1;	The printer is a POS printer.

Description

Printer Device Function support Information

This structure contains information about the functions that the attached printer supports. See the related constants for setting these flags via the val member:

- USB_PRINTER_FUNCTION_SUPPORT_POS (see page 756)
- USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS (see page 757)

7.2.10.2.77 USB_PRINTER_FUNCTION_SUPPORT_POS Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_FUNCTION_SUPPORT_POS 0x0002
```

Description

Constant to use to set the supportsPOS member of the USB_PRINTER_FUNCTION_SUPPORT (see page 755) union.

7.2.10.2.78 USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_FUNCTION_SUPPORT_VECTOR_GRAPHICS 0x0001
```

Description

Constant to use to set the supportsVectorGraphics member of the USB_PRINTER_FUNCTION_SUPPORT (see page 755) union.

7.2.10.2.79 USB_PRINTER_GRAPHICS_PARAMETERS Union

File

usb_host_printer.h

C

```
typedef union {
  struct {
    WORD xL;
    WORD yT;
    WORD xR;
    WORD yB;
    WORD r1;
    WORD r2;
    WORD octant;
  } sArc;
  struct {
    BYTE height;
    BYTE type;
    BYTE textPosition;
    BYTE textFont;
    BYTE * data;
    BYTE dataLength;
    union {
      BYTE value;
      struct {
        BYTE bPrintCheckDigit : 1;
      } bits;
    } flags;
  } sBarCode;
  struct {
    WORD xL;
    WORD yT;
    WORD xR;
    WORD yB;
    WORD r;
  } sBevel;
  struct {
    WORD x;
    WORD y;
    WORD r;
  } sCircle;
  struct {
    WORD fillType;
    WORD spacing;
    WORD angle;
    WORD shading;
  } sFillType;
  struct {
    WORD x1;
    WORD y1;
    WORD x2;
    WORD y2;
  } sLine;
  struct {
    WORD startPointHorizontal;
    WORD startPointVertical;
    WORD lengthHorizontal;
    WORD lengthVertical;
    BYTE printDirection;
  } sPage;
  struct {
    SHORT numPoints;
    WORD * points;
  } sPolygon;
  struct {
    WORD xL;
    WORD yT;
```

```
        WORD xR;
        WORD yB;
    } sRectangle;
} USB_PRINTER_GRAPHICS_PARAMETERS;
```

Members

Members	Description
struct { WORD xL; WORD yT; WORD xR; WORD yB; WORD r1; WORD r2; WORD octant; } sArc;	This structure is used by the USB_PRINTER_GRAPHICS_ARC command (USB_PRINTER_COMMAND (see page 742)).
WORD xL;	X-axis position of the upper left corner.
WORD yT;	Y-axis position of the upper left corner.
WORD xR;	X-axis position of the lower right corner.
WORD yB;	Y-axis position of the lower right corner.
WORD r1;	The inner radius of the two concentric cicles that defines the thickness of the arc.
WORD r2;	The outer of radius the two concentric circles that defines the thickness of the arc.
WORD octant;	Bitmask of the octant that will be drawn. Moving in a clockwise direction from x = 0, y = +radius • bit0 : first octant • bit1 : second octant • bit2 : third octant • bit3 : fourth octant • bit4 : fifth octant • bit5 : sixth octant • bit6 : seventh octant • bit7 : eigth octant

struct {
BYTE height;
BYTE type;
BYTE textPosition;
BYTE textFont;
BYTE * data;
BYTE dataLength;
union {
BYTE value;
struct {
BYTE bPrintCheckDigit : 1;
} bits;
} flags;
} sBarCode;
	This structure is used by the USB_PRINTER_POS_BARCODE command (USB_PRINTER_COMMAND (see page 742)).

BYTE height;	Bar code height in dots.
BYTE type;	Bar code type. See the USB_PRINTER_POS_BARCODE_FORMAT (see page 767) enumeration.

BYTE textPosition;	Position of the readable text. Valid values are BARCODE_TEXT_OMIT (see page 686), BARCODE_TEXT_ABOVE (see page 683), BARCODE_TEXT_BELOW (see page 685), BARCODE_TEXT_ABOVE_AND_BELOW (see page 684).
BYTE textFont;	Font of the readable text. Valid values are dependent on the particular POS printer (BARCODE_TEXT_12x24 (see page 681) and BARCODE_TEXT_18x36 (see page 682) for ESC/POS).
BYTE * data;	Pointer to the bar code data.
BYTE dataLength;	Number of bytes of bar code data.
BYTE bPrintCheckDigit : 1;	Whether or not to print an optional check digit. Valid for Code39 (USB_PRINTER_POS_BARCODE_FORMAT (see page 767) USB_PRINTER_POS_BARCODE_CODE39) and CODABAR (USB_PRINTER_POS_BARCODE_FORMAT (see page 767) USB_PRINTER_POS_BARCODE_CODABAR) formats only.
struct { WORD xL; WORD yT; WORD xR; WORD yB; WORD r; } sBevel;	This structure is used by the USB_PRINTER_GRAPHICS_BEVEL and USB_PRINTER_GRAPHICS_BEVEL_FILLED commands (USB_PRINTER_COMMAND (see page 742)).
WORD xL;	X-axis position of the left side of the bevel.
WORD yT;	Y-axis position of the top of the bevel.
WORD xR;	X-axis position of the right side of the bevel.
WORD yB;	Y-axis position of the bottom of the bevel.
WORD r;	The radius of the cicle that defines the rounded corner
struct { WORD x; WORD y; WORD r; } sCircle;	This structure is used by the USB_PRINTER_GRAPHICS_CIRCLE and USB_PRINTER_GRAPHICS_CIRCLE_FILLED commands (USB_PRINTER_COMMAND (see page 742)).
WORD x;	X-axis position of the center of the circle.
WORD y;	Y-axis position of the center of the circle.
WORD r;	Radius of the circle.
struct { WORD fillType; WORD spacing; WORD angle; WORD shading; } sFillType;	This structure is used by the USB_PRINTER_GRAPHICS_FILL_TYPE command (USB_PRINTER_COMMAND (see page 742)).
WORD fillType;	The type of fill. See USB_PRINTER_GRAPHICS_FILL_TYPE for valid values.
WORD spacing;	Line spacing for hatched fill (if supported).
WORD angle;	Line angle for hatched fill (if supported).
WORD shading;	Shading level for shaded fill. Printer support may be limited.
struct { WORD x1; WORD y1; WORD x2; WORD y2; } sLine;	This structure is used by the USB_PRINTER_GRAPHICS_LINE command (USB_PRINTER_COMMAND (see page 742)).
WORD x1;	X-axis position of the first point.
WORD y1;	Y-axis position of the first point.
WORD x2;	X-axis position of the second point.
WORD y2;	Y-axis position of the second point.

struct { WORD startPointHorizontal; WORD startPointVertical; WORD lengthHorizontal; WORD lengthVertical; BYTE printDirection; } sPage;	This structure is used by POS printers and the USB_PRINTER_POS_PAGE_MODE command (USB_PRINTER_COMMAND (see page 742)).
WORD startPointHorizontal;	The horizontal page starting point.
WORD startPointVertical;	The vertical page starting point.
WORD lengthHorizontal;	The horizontal print length.
WORD lengthVertical;	The vertical print length.
BYTE printDirection;	The print direction and starting point.
struct { SHORT numPoints; WORD * points; } sPolygon;	This structure is used by the USB_PRINTER_GRAPHICS_POLYGON command (USB_PRINTER_COMMAND (see page 742)).
SHORT numPoints;	The number of points of the polygon.
WORD * points;	The array of polygon points {x1, y1, x2, y2, ... xn, yn}.
struct { WORD xL; WORD yT; WORD xR; WORD yB; } sRectangle;	This structure is used by the USB_PRINTER_GRAPHICS_RECTANGLE and USB_PRINTER_GRAPHICS_RECTANGLE_FILLED commands (USB_PRINTER_COMMAND (see page 742)).
WORD xL;	X-axis position of the left side of the rectangle.
WORD yT;	Y-axis position of the top of the rectangle.
WORD xR;	X-axis position of the right side of the rectangle.
WORD yB;	Y-axis position of the bottom of the rectangle.

Description

USB Printer Graphics Parameter Structures

This union can be used to declare a variable that can hold the parameters for any printer graphics or POS printer command (USB_PRINTER_COMMAND (see page 742)). The union allows a single variable to be declared and then reused for any printer graphics command.

7.2.10.2.80 USB_PRINTER_IMAGE_INFO Structure

File

usb_host_printer.h

C

```
typedef struct {
  WORD width;
  WORD height;
  WORD positionX;
  WORD positionY;
  union {
    struct {
      WORD resolution;
      float scale;
    }
    struct {
      BYTE densityVertical;
      BYTE densityHorizontal;
    }
  }
} USB_PRINTER_IMAGE_INFO;
```

Members

Members	Description
WORD width;	The width of the image in pixels.
WORD height;	The height of the image in pixels.
WORD positionX;	The position of the image on the X axis.
WORD positionY;	The position of the image on the Y axis.
WORD resolution;	(Full sheet printers only.) The resolution of the printed image. This parameter is not supported by all printer languages.
float scale;	(Full sheet printers only.) The scaling of the printed image. Both the X axis and the Y axis are scaled by this amount. This parameter is not supported by all printer languages.
BYTE densityVertical;	(POS printers only.) The vertical dot density of the bit image. Valid values are printer dependent. See above.
BYTE densityHorizontal;	(POS printers only.) The horizontal dot density of the bit image. Valid values are 1 (single) and 2 (double). See above.

Description

Bitmapped Image Information

This structure contains the information needed to print a bitmapped graphic image.

When using a full sheet printer, utilize the resolution and the scale members to specify the size of the image. Some printer languages (e.g. PostScript) utilize a scale factor, while others (e.g. PCL 5) utilize a dots-per-inch resolution. Also, some printers that utilize the resolution specification support only certain values for the resolution. For maximum compatibility, specify both members of this structure. The following table shows example values that will generate similarly sized output.

```
Resolution (DPI)  Scale
----------------  -----
       75          1.0
      100          0.75
      150          0.5
      200          0.37
      300          0.25
      600          0.13
```

When using a POS printer, utilize the densityVertical and densityHorizontal members to specify the size of the image. The densityHorizontal can be either single (1) or double (2). The valid values for densityVertical are printer dependent. Most printers support 8-dot, many support 8 and 24-dot, and a few support 8, 24, and 36-dot (represented by the values 8, 24, and 36 respectively). This value affects how the bit image data is sent to the printer. The set of allowable values must be configured correctly, since the image configuration method differs depending on the set of allowed values. To maintain the aspect ratio, the following selections are recommended:

```
Supported Horizontal Densities  densityVertical  densityHorizontal
------------------------------------------------------------------
8-dot                                  8             1 (single)
8 and 24-dot                          24             2 (double)
8, 24, and 36-dot                     24             2 (double)
```

The 36-bit density is not recommended, as it requires a great deal of available heap space, is not supported by the USBHostPrinterPOSImageDataFormat (see page 655)() function, and produces the same output as the 24-dot density print.

7.2.10.2.81 USB_PRINTER_INTERFACE Structure

File

usb_host_printer.h

C

```
typedef struct {
  USB_PRINTER_LANGUAGE_HANDLER languageCommandHandler;
  USB_PRINTER_LANGUAGE_SUPPORTED isLanguageSupported;
} USB_PRINTER_INTERFACE;
```

Members

Members	Description
USB_PRINTER_LANGUAGE_HANDLER languageCommandHandler;	Function in the printer language support file that handles all printer commands.
USB_PRINTER_LANGUAGE_SUPPORTED isLanguageSupported;	Function in the printer language support file that determines if the printer supports this particular printer language.

Description

USB Printer Interface Structure

This structure represents the information needed to interface with a printer language. An array of these structures must be created in usb_config.c, so the USB printer client driver can determine what printer language to use to communicate with the printer.

7.2.10.2.82 USB_PRINTER_LANGUAGE_HANDLER Type

This is a typedef to use when defining a printer language command handler.

File

usb_host_printer.h

C

```
typedef BYTE (* USB_PRINTER_LANGUAGE_HANDLER)(BYTE address, USB_PRINTER_COMMAND command,
USB_DATA_POINTER data, DWORD size, BYTE flags);
```

Description

This data type defines a pointer to a call-back function that must be implemented by a printer language driver. When the user calls the printer interface function, the appropriate language driver with this prototype will be called to generate the proper commands for the requested operation.

Not all printer commands support data from both RAM and ROM. Unless otherwise noted, the data pointer is assumed to point to RAM, regardless of the value of transferFlags. Refer to the specific command to see if ROM data is supported.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BYTE address	Device's address on the bus
USB_PRINTER_COMMAND command	Command to execute. See the enumeration USB_PRINTER_COMMAND (see page 742) for the list of valid commands and their requirements.
USB_DATA_POINTER data	Pointer to the required data. Note that the caller must set transferFlags appropriately to indicate if the pointer is a RAM pointer or a ROM pointer.
DWORD size	Size of the data. For some commands, this parameter is used to hold the data itself.
BYTE transferFlags	Flags that indicate details about the transfer operation. Refer to these flags • USB_PRINTER_TRANSFER_COPY_DATA (see page 770) • USB_PRINTER_TRANSFER_STATIC_DATA (see page 774) • USB_PRINTER_TRANSFER_NOTIFY (see page 773) • USB_PRINTER_TRANSFER_FROM_ROM (see page 772) • USB_PRINTER_TRANSFER_FROM_RAM (see page 771)

Return Values

Return Values	Description
USB_PRINTER_SUCCESS	The command was executed successfully.
USB_PRINTER_UNKNOWN_DEVICE	A printer with the indicated address is not attached
USB_PRINTER_TOO_MANY_DEVICES	The printer status array does not have space for another printer.
USB_PRINTER_OUT_OF_MEMORY	Not enough available heap space to execute the command.
other	See possible return codes from the function USBHostPrinterWrite (see page 662)().

Function

BYTE (*USB_PRINTER_LANGUAGE_HANDLER) (BYTE address,

USB_PRINTER_COMMAND (see page 742) command, USB_DATA_POINTER (see page 736) data, DWORD size, BYTE flags)

7.2.10.2.83 USB_PRINTER_LANGUAGE_SUPPORTED Type

This is a typedef to use when defining a function that determines if the printer with the given "COMMAND SET:" portion of the device ID string supports the particular printer language.

File

usb_host_printer.h

C

```
typedef BOOL (* USB_PRINTER_LANGUAGE_SUPPORTED)(char *deviceID,
USB_PRINTER_FUNCTION_SUPPORT *support);
```

Description

This data type defines a pointer to a call-back function that must be implemented by a printer language driver. When the user calls a function of this type, the language driver will return a BOOL indicating if the language driver supports a printer with the indicated "COMMAND SET:" portion of the device ID string. If the printer is supported, this function also returns information about the types of operations that the printer supports.

Remarks

The caller must first locate the "COMMAND SET:" section of the device ID string. To ensure that only the "COMMAND SET:" section of the device ID string is checked, the ";" at the end of the section should be temporarily replaced with a NULL. Otherwise, this function may find the printer language string in the comments or other section, and incorrectly indicate that the printer supports the language.

Device ID strings are case sensitive.

Preconditions

None

Parameters

Parameters	Description
char *deviceID	Pointer to the "COMMAND SET:" portion of the device ID string of the attached printer.
USB_PRINTER_FUNCTION_SUPPORT *support	Pointer to returned information about what types of functions this printer supports.

Return Values

Return Values	Description
TRUE	The printer language can be used with the attached printer.
FALSE	The printer language cannot be used with the attached printer.

Function

BOOL (*USB_PRINTER_LANGUAGE_SUPPORTED) (char *deviceID,

USB_PRINTER_FUNCTION_SUPPORT (see page 755) *support)

7.2.10.2.84 USB_PRINTER_POS_BARCODE_FORMAT Enumeration

File

usb_host_printer.h

C

```
typedef enum {
  USB_PRINTER_POS_BARCODE_UPC_A = 0,
  USB_PRINTER_POS_BARCODE_UPC_E,
  USB_PRINTER_POS_BARCODE_EAN13,
  USB_PRINTER_POS_BARCODE_EAN8,
  USB_PRINTER_POS_BARCODE_CODE39,
  USB_PRINTER_POS_BARCODE_ITF,
  USB_PRINTER_POS_BARCODE_CODABAR,
  USB_PRINTER_POS_BARCODE_CODE93,
  USB_PRINTER_POS_BARCODE_CODE128,
  USB_PRINTER_POS_BARCODE_EAN128,
  USB_PRINTER_POS_BARCODE_MAX
} USB_PRINTER_POS_BARCODE_FORMAT;
```

Members

Members	Description
USB_PRINTER_POS_BARCODE_UPC_A = 0	UPC-A bar code format. Typically used for making products with a unique code,as well as for coupons, periodicals, and paperback books. The data for this bar code must consist of 11 values from '0' to '9' (ASCII), and the data length for this bar code must be 11. The first digit is the number system character: • 0, 6, 7 Regular UPC codes • 2 Random weight items • 3 National Drug Code and National Health Related Items Code • 4 In-store marking of non-food items • 5 Coupons • 1, 8, 9 Reserved A check digit will be automatically calculated and appended. For more information, refer to the UPC Symbol Specification Manual from the Uniform Code Council.
USB_PRINTER_POS_BARCODE_UPC_E	UPC-E bar code format. Similar to UPC-A but with restrictions. Data lengths of 6, 7, or 11 bytes are supported. Not all printers support the 6 or 7 byte widths; 11 byte data is recommended. If the data length is not 6, then the first the first digit (the number system character) must be set to '0'. If 11 data bytes are presented, the printer will generate a shortened 6-digit code. The check digit will be automatically calculated and appended.
USB_PRINTER_POS_BARCODE_EAN13	EAN/JAN-13 bar code format. Similar to UPC-A, but there are 12 numeric digits plus a checksum digit. The check digit will be automatically calculated and appended.
USB_PRINTER_POS_BARCODE_EAN8	EAN/JAN-8 bar code format. Similar to UPC-E, but there are 7 numeric digits, and the first digit (the number system character) must be set to '0'. The check digit will be automatically calculated and appended.
USB_PRINTER_POS_BARCODE_CODE39	CODE39 bar code format. Typically used in applications where the data length may change. This format uses encoded numeric characters, uppercase alphabet characters, and the symbols '-' (dash), '.' (period), ' ' (space), '$' (dollar sign), '/' (forward slash), '+' (plus), and '%' (percent). If the bPrintCheckDigit flag is set, then the check digit will be automatically calculated and appended. Otherwise, no check digit will be printed.
USB_PRINTER_POS_BARCODE_ITF	ITF, or Interleaved 2 of 5, bar code format. Used in applications that have a fixed data length for all items. Only the digits 0-9 can be encoded, and there must be an even number of digits.

USB_PRINTER_POS_BARCODE_CODABAR	Codabar bar code format. Useful in applications that contain mostly numeric digits and variable data sizes. This format utilizes the digits 0-9, letters A-D (used as start/stop characters), '-' (dash), '$' (dollar sign), ':' (colon), '/' (forward slash), '.' (period), and '+' (plus). If the bPrintCheckDigit flag is set, then the check digit will be automatically calculated and appended. Otherwise, no check digit will be printed.
USB_PRINTER_POS_BARCODE_CODE93	(Available only if the printer supports extended bar code formats.) CODE93 bar code format. Used in applications that require heavy error checking. It has a variable data size, and uses 128-bit ASCII characters. The start code, stop code, and check digits are added automatically.
USB_PRINTER_POS_BARCODE_CODE128	(Available only if the printer supports extended bar code formats.) Code 128 bar code format. Used in applications that require a large amount of variable length data and extra error checking. It uses 128-bit ASCII plus special symbols. The first two data bytes must be the code set selection character. The first byte must be BARCODE_CODE128_CODESET (0x7B), and the second byte must be 'A', 'B', or 'C'. In general Code A should be used if the data contains control characters (0x00 - 0x1F), and Code B should be used if the data contains lower case letters and higher ASCII values (0x60-0x7F). If an ASCII '{' (left brace, 0x7B) is contained in the data, it must be encoded as two bytes with the value 0x7B.
USB_PRINTER_POS_BARCODE_EAN128	NOT YET SUPPORTED (Available only if the printer supports extended bar code formats.) EAN-128 or UCC-128 bar code format. Used in shipping applications. Refer to the Application Standard for Shopping Container Codes from the Uniform Code Council.
USB_PRINTER_POS_BARCODE_MAX	Bar code type out of range.

Description

Bar Code Formats

These are the bar code formats for printing bar codes on POS printers. They are used in conjuction with the USB_PRINTER_POS_BARCODE command (USB_PRINTER_COMMAND (see page 742)). Bar code information is passed using the sBarCode structure within the USB_PRINTER_GRAPHICS_PARAMETERS (see page 758) union. The exact values to send for each bar code type can vary for the particular POS printer, and not all printers support all bar code types. Be sure to test the output on the target printer, and adjust the values specified in usb_host_printer_esc_pos.c if necessary. Refer to the printer's technical documentation for the required values. Do not alter this enumeration.

7.2.10.2.85 USB_PRINTER_SPECIFIC_INTERFACE Structure

File

usb_host_printer.h

C

```
typedef struct {
  WORD vid;
  WORD pid;
  WORD languageIndex;
  USB_PRINTER_FUNCTION_SUPPORT support;
} USB_PRINTER_SPECIFIC_INTERFACE;
```

Members

Members	Description
WORD vid;	Printer vendor ID.
WORD pid;	Printer product ID.
WORD languageIndex;	Index into the usbPrinterClientLanguages[] array of USB_PRINTER_INTERFACE (see page 764) structures defined in usb_config.c.
USB_PRINTER_FUNCTION_SUPPORT support;	Support flags that are set by this printer.

Description

USB Printer Specific Interface Structure

This structure is used to explicitly specify what printer language to use for a particular printer, and what print functions the printer supports. It can be used when a printer supports multiple languages with one language preferred over the others. It is required for printers that do not support the GET_DEVICE_ID printer class request. These printers do not report what printer languages they support. Typically, these printers also do not report Printer Class support in their Interface Descriptors, and must be explicitly supported by their VID and PID in the TPL. This structure links the VID and PID of the printer to the index in the usbPrinterClientLanguages[] array of USB_PRINTER_INTERFACE (see page 764) structures in usb_config.c that contains the appropriate printer language functions.

7.2.10.2.86 USB_PRINTER_TRANSFER_COPY_DATA Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_TRANSFER_COPY_DATA 0x01
```

Description

This flag indicates that the printer client driver should make a copy of the data passed to the command. This allows the application to reuse the data storage immediately instead of waiting until the transfer is sent to the printer. The client driver will allocate space in the heap for the data copy. If there is not enough available memory, the command will terminate with a USB_PRINTER_OUT_OF_MEMORY error. Otherwise, the original data will be copied to the temporary data space. This temporary data will be freed upon completion, regardless of whether or not the command was performed successfully. NOTE: If the data is located in ROM, the flag USB_PRINTER_TRANSFER_FROM_ROM (see page 772) must be used instead.

7.2.10.2.87 USB_PRINTER_TRANSFER_FROM_RAM Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_TRANSFER_FROM_RAM 0x00
```

Description

This flag indicates that the source of the command data is in RAM. The application can then choose whether or not to have the printer client driver make a copy of the data.

7.2.10.2.88 USB_PRINTER_TRANSFER_FROM_ROM Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_TRANSFER_FROM_ROM 0x04
```

Description

This flag indicates that the source of the command data is in ROM. The data will be copied to RAM, since the USB Host layer cannot read data from ROM. If there is not enough available heap space to make a copy of the data, the command will fail. If using this flag, do not set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag.

7.2.10.2.89 USB_PRINTER_TRANSFER_NOTIFY Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_TRANSFER_NOTIFY 0x02
```

Description

This flag indicates that the application layer wants to receive an event notification when the command completes.

7.2.10.2.90 USB_PRINTER_TRANSFER_STATIC_DATA Macro

File

usb_host_printer.h

C

```
#define USB_PRINTER_TRANSFER_STATIC_DATA 0x00
```

Description

This flag indicates that the data will not change in the time between the printer command being issued and the data actually being sent to the printer.

7.2.10.2.91 USBHOSTPRINTER_SETFLAG_COPY_DATA Macro

File

usb_host_printer.h

C

```
#define USBHOSTPRINTER_SETFLAG_COPY_DATA(x) {x |= USB_PRINTER_TRANSFER_COPY_DATA;}
```

Description

Use this macro to set the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag in a variable.

7.2.10.2.92 USBHOSTPRINTER_SETFLAG_NOTIFY Macro

File

usb_host_printer.h

C

```
#define USBHOSTPRINTER_SETFLAG_NOTIFY(x) {x |= USB_PRINTER_TRANSFER_NOTIFY;}
```

Description

Use this macro to set the USB_PRINTER_TRANSFER_NOTIFY (see page 773) flag in a variable.

7.2.10.2.93 USBHOSTPRINTER_SETFLAG_STATIC_DATA Macro

File

usb_host_printer.h

C

```
#define USBHOSTPRINTER_SETFLAG_STATIC_DATA(x) {x &= ~USB_PRINTER_TRANSFER_COPY_DATA;}
```

Description

Use this macro to clear the USB_PRINTER_TRANSFER_COPY_DATA (see page 770) flag in a variable.

7.3 On-The-Go (OTG)

This module provides support for USB OTG (On-The-Go) functionality.

Description

USB OTG (On-The-Go)

USB OTG was defined by the USB-IF to standardize connectivity in mobile devices. USB OTG allows devices to be dual role (Host or Peripheral) and dynamically switch between the two. For example, you could have all in one product, a device that is a peripheral when plugged into a PC, a device that is an embedded host when plugged into a digital camera, a device that is an embedded host when plugged into a printer, and a device that is an embedded host when plugged into a keyboard.

A USB OTG device uses a Micro A/B style receptacle. When a Micro A plug is inserted, the device will take on the default role of being a host. When a Micro B plug is inserted, the device will take on the default role of being a peripheral. When no plug is inserted, the device will take on the role of being a peripheral.

The USB OTG layer provides an interface for a USB OTG device to dynamically switch roles between either being an embedded host or a peripheral. The USB OTG layer is called into by the the Embedded Host and Peripheral Device Stacks. The USB OTG layer is responsible for switching roles using the Host Negotiation Protocol (HNP), requesting sessions using the Session Request Protocol(SRP), providing role status to the application, and displaying any errors.

Switching Roles using Host Negotiation Protocol (HNP)

Switching Roles is easily accomplished using the USBOTGSelectRole (see page 785)() function call. This function is called on the A-side Host when it is ready to become a peripheral and give the B-side peripheral the opportunity to become Host.

Requesting Sessions using Session Request Protocol (SRP)

If the A-side Host has ended a session (turned off VBUS power), the B-side can request a new VBUS session. This is easily accomplished by using the USBOTGRequestSession (see page 783)() function call. This function should only be called on a B-side peripheral.

Main Application

The main application should have the following code at a minimum for initialization, re-initialization of the system when a role switch occurs, maintaining the stack tasks, and maintaining the application tasks.

```
InitializeSystem();
USBOTGInitialize();

while(1)
{
    //If Role Switch Occurred Then
```

```
    if (USBOTGRoleSwitch())
    {
        //Re-Initialize
        InitializeSystem();

        //Clear Role Switch Flag
        USBOTGClearRoleSwitch();
    }

    //If currently a Peripheral and HNP is not Active Then
    if (USBOTGCurrentRoleIs() == ROLE_DEVICE && !USBOTGHnpIsActive())
    {
        //Call Device Tasks
        USBDeviceTasks();

        //Call Process IO
        ProcessIO();
    }

    //If currently a Host and HNP is not Active Then
    else if (USBOTGCurrentRoleIs() == ROLE_HOST && !USBOTGHnpIsActive())
    {
        //Call Host Tasks
        USBHostTasks();

        //Call Manage Demo
        ManageDemoState();
    }
}
```

See *AN1140 USB Embedded Host Stack* for more information about the Embedded Host Stack layer.

See the Microchip USB Device Firmware Framework User's Guide from the www.microchip.com/usb **Documentation** link for more information about the USB Device Stack layer.

7.3.1 Interface Routines

Functions

	Name	Description
	USBOTGClearRoleSwitch (see page 779)	This function clears the RoleSwitch variable. After the main function detects the RoleSwitch and re-initializes the system, this function should be called to clear the RoleSwitch flag
	USBOTGCurrentRoleIs (see page 780)	This function returns whether the current role is ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)
	USBOTGDefaultRoleIs (see page 781)	This function returns whether the default role is ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)
	USBOTGInitialize (see page 782)	This function initializes an OTG application and initializes a default role of Host or Device
	USBOTGRequestSession (see page 783)	This function requests a Session from an A side Host using the Session Request Protocol (SRP). The function will return TRUE if the request was successful or FALSE otherwise.
	USBOTGRoleSwitch (see page 784)	This function returns whether a role switch occurred or not. This is used by the main application function to determine when to reinitialize the system (InitializeSystem())
	USBOTGSelectRole (see page 785)	This function initiates a role switch via the Host Negotiation Protocol (HNP). The parameter role that is passed to this function is the desired role to switch to.
	USBOTGSession (see page 786)	This function starts, ends, or toggles a VBUS session.

Description

7.3.1.1 USBOTGClearRoleSwitch Function

File

usb_otg.h

C

```
void USBOTGClearRoleSwitch();
```

Description

This function clears the RoleSwitch variable. After the main function detects the RoleSwitch and re-initializes the system, this function should be called to clear the RoleSwitch flag

Remarks

None

Preconditions

None

Function

void USBOTGClearRoleSwitch()

7.3.1.2 USBOTGCurrentRoleIs Function

File

usb_otg.h

C

```
BYTE USBOTGCurrentRoleIs();
```

Description

This function returns whether the current role is ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)

Remarks

None

Preconditions

None

Return Values

Return Values	Description
BYTE	ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)

Function

BYTE USBOTGCurrentRoleIs()

7.3.1.3 USBOTGDefaultRoleIs Function

File

usb_otg.h

C

```
BYTE USBOTGDefaultRoleIs();
```

Description

This function returns whether the default role is ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)

Remarks

If using a Micro AB USB OTG Cable, the A-side plug of the cable when plugged in will assign a default role of ROLE_HOST (see page 815). The B-side plug of the cable when plugged in will assign a default role of ROLE_DEVICE (see page 814).

If using a Standard USB Cable, ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814) needs to be manually configured in usb_config.h.

Both of these items can be easily configured using the USB Config Tool which will automatically generate the apropriate information for your application

Preconditions

None

Return Values

Return Values	Description
BYTE	ROLE_HOST (see page 815) or ROLE_DEVICE (see page 814)

Function

BYTE USBOTGDefaultRoleIs()

7.3.1.4 USBOTGInitialize Function

File

usb_otg.h

C

```
void USBOTGInitialize();
```

Description

This function initializes an OTG application and initializes a default role of Host or Device

Remarks

#define USB_MICRO_AB_OTG_CABLE should be commented out in usb_config.h if not using a micro AB OTG cable

Preconditions

None

Function

void USBOTGInitialize()

7.3.1.5 USBOTGRequestSession Function

File

usb_otg.h

C

```
BOOL USBOTGRequestSession();
```

Description

This function requests a Session from an A side Host using the Session Request Protocol (SRP). The function will return TRUE if the request was successful or FALSE otherwise.

Remarks

This function should only be called by a B side Device.

Preconditions

None

Function

void USBOTGRequestSession()

7.3.1.6 USBOTGRoleSwitch Function

File

usb_otg.h

C

```
BOOL USBOTGRoleSwitch();
```

Description

This function returns whether a role switch occurred or not. This is used by the main application function to determine when to reinitialize the system (InitializeSystem())

Remarks

None

Preconditions

None

Return Values

Return Values	Description
BOOL	TRUE or FALSE

Function

BOOL USBOTGRoleSwitch()

7.3.1.7 USBOTGSelectRole Function

File

usb_otg.h

C

```
void USBOTGSelectRole(
    BOOL role
);
```

Description

This function initiates a role switch via the Host Negotiation Protocol (HNP). The parameter role that is passed to this function is the desired role to switch to.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BOOL role	ROLE_DEVICE (see page 814) or ROLE_HOST (see page 815)

Function

void USBOTGSelectRole(BOOL role)

7.3.1.8 USBOTGSession Function

File

usb_otg.h

C

```
BOOL USBOTGSession(
    BYTE Value
);
```

Description

This function starts, ends, or toggles a VBUS session.

Remarks

This function should only be called by an A-side Host

Preconditions

This function assumes I/O controlling DC/DC converter has already been initialized

Parameters

Parameters	Description
Value	START_SESSION (see page 816), END_SESSION (see page 801), TOGGLE_SESSION (see page 817)

Return Values

Return Values	Description
TRUE	Session Started, FALSE - Session Not Started

Function

void USBOTGSession(BYTE Value)

7.3.2 Data Types and Constants

Macros

	Name	Description
	CABLE_A_SIDE (see page 788)	Cable Defines
	CABLE_B_SIDE (see page 789)	This is macro CABLE_B_SIDE.
	DELAY_TA_AIDL_BDIS (see page 790)	This is macro DELAY_TA_AIDL_BDIS.
	DELAY_TA_BDIS_ACON (see page 791)	This is macro DELAY_TA_BDIS_ACON.
	DELAY_TA_BIDL_ADIS (see page 792)	150
	DELAY_TA_WAIT_BCON (see page 793)	This is macro DELAY_TA_WAIT_BCON.
	DELAY_TA_WAIT_VRISE (see page 794)	This is macro DELAY_TA_WAIT_VRISE.
	DELAY_TB_AIDL_BDIS (see page 795)	100

	DELAY_TB_ASE0_BRST (see page 796)	This is macro DELAY_TB_ASE0_BRST.
	DELAY_TB_DATA_PLS (see page 797)	This is macro DELAY_TB_DATA_PLS.
	DELAY_TB_SE0_SRP (see page 798)	This is macro DELAY_TB_SE0_SRP.
	DELAY_TB_SRP_FAIL (see page 799)	This is macro DELAY_TB_SRP_FAIL.
	DELAY_VBUS_SETTLE (see page 800)	This is macro DELAY_VBUS_SETTLE.
	END_SESSION (see page 801)	This is macro END_SESSION.
	OTG_EVENT_CONNECT (see page 802)	This is macro OTG_EVENT_CONNECT.
	OTG_EVENT_DISCONNECT (see page 803)	OTG Events
	OTG_EVENT_HNP_ABORT (see page 804)	This is macro OTG_EVENT_HNP_ABORT.
	OTG_EVENT_HNP_FAILED (see page 805)	This is macro OTG_EVENT_HNP_FAILED.
	OTG_EVENT_NONE (see page 806)	This is macro OTG_EVENT_NONE.
	OTG_EVENT_RESUME_SIGNALING (see page 807)	This is macro OTG_EVENT_RESUME_SIGNALING.
	OTG_EVENT_SRP_CONNECT (see page 808)	This is macro OTG_EVENT_SRP_CONNECT.
	OTG_EVENT_SRP_DPLUS_HIGH (see page 809)	This is macro OTG_EVENT_SRP_DPLUS_HIGH.
	OTG_EVENT_SRP_DPLUS_LOW (see page 810)	This is macro OTG_EVENT_SRP_DPLUS_LOW.
	OTG_EVENT_SRP_FAILED (see page 811)	This is macro OTG_EVENT_SRP_FAILED.
	OTG_EVENT_SRP_VBUS_HIGH (see page 812)	This is macro OTG_EVENT_SRP_VBUS_HIGH.
	OTG_EVENT_SRP_VBUS_LOW (see page 813)	This is macro OTG_EVENT_SRP_VBUS_LOW.
	ROLE_DEVICE (see page 814)	Role Defines
	ROLE_HOST (see page 815)	This is macro ROLE_HOST.
	START_SESSION (see page 816)	Session Defines
	TOGGLE_SESSION (see page 817)	This is macro TOGGLE_SESSION.
	USB_OTG_FW_DOT_VER (see page 818)	Firmware version, dot release number.
	USB_OTG_FW_MAJOR_VER (see page 819)	Firmware version, major release number.
	USB_OTG_FW_MINOR_VER (see page 820)	Firmware version, minor release number.

Description

7.3.2.1 CABLE_A_SIDE Macro

File

usb_otg.h

C

```
#define CABLE_A_SIDE 0
```

Description

Cable Defines

7.3.2.2 CABLE_B_SIDE Macro

File

usb_otg.h

C

```
#define CABLE_B_SIDE 1
```

Description

This is macro CABLE_B_SIDE.

7.3.2.3 DELAY_TA_AIDL_BDIS Macro

File

usb_otg.h

C

```
#define DELAY_TA_AIDL_BDIS 255
```

Description

This is macro DELAY_TA_AIDL_BDIS.

7.3.2.4 DELAY_TA_BDIS_ACON Macro

File

usb_otg.h

C

```
#define DELAY_TA_BDIS_ACON 1
```

Description

This is macro DELAY_TA_BDIS_ACON.

7.3.2.5 DELAY_TA_BIDL_ADIS Macro

File

usb_otg.h

C

```
#define DELAY_TA_BIDL_ADIS 10//150
```

Description

150

7.3.2.6 DELAY_TA_WAIT_BCON Macro

File

usb_otg.h

C

```
#define DELAY_TA_WAIT_BCON 1100
```

Description

This is macro DELAY_TA_WAIT_BCON.

7.3.2.7 DELAY_TA_WAIT_VRISE Macro

File

usb_otg.h

C

```
#define DELAY_TA_WAIT_VRISE 100
```

Description

This is macro DELAY_TA_WAIT_VRISE.

7.3.2.8 DELAY_TB_AIDL_BDIS Macro

File

usb_otg.h

C

```
#define DELAY_TB_AIDL_BDIS 10 //100
```

Description

100

7.3.2.9 DELAY_TB_ASE0_BRST Macro

File

usb_otg.h

C

```
#define DELAY_TB_ASE0_BRST 100
```

Description

This is macro DELAY_TB_ASE0_BRST.

7.3.2.10 DELAY_TB_DATA_PLS Macro

File

usb_otg.h

C

```
#define DELAY_TB_DATA_PLS 6
```

Description

This is macro DELAY_TB_DATA_PLS.

7.3.2.11 DELAY_TB_SE0_SRP Macro

File

usb_otg.h

C

```
#define DELAY_TB_SE0_SRP 2
```

Description

This is macro DELAY_TB_SE0_SRP.

7.3.2.12 DELAY_TB_SRP_FAIL Macro

File

usb_otg.h

C

```
#define DELAY_TB_SRP_FAIL 5100
```

Description

This is macro DELAY_TB_SRP_FAIL.

7.3.2.13 DELAY_VBUS_SETTLE Macro

File

usb_otg.h

C

```
#define DELAY_VBUS_SETTLE 500
```

Description

This is macro DELAY_VBUS_SETTLE.

7.3.2.14 END_SESSION Macro

File

usb_otg.h

C

```
#define END_SESSION 1
```

Description

This is macro END_SESSION.

7.3.2.15 OTG_EVENT_CONNECT Macro

File

usb_otg.h

C

```
#define OTG_EVENT_CONNECT 1
```

Description

This is macro OTG_EVENT_CONNECT.

7.3.2.16 OTG_EVENT_DISCONNECT Macro

File

usb_otg.h

C

```
#define OTG_EVENT_DISCONNECT 0
```

Description

OTG Events

7.3.2.17 OTG_EVENT_HNP_ABORT Macro

File

usb_otg.h

C

```
#define OTG_EVENT_HNP_ABORT 8
```

Description

This is macro OTG_EVENT_HNP_ABORT.

7.3.2.18 OTG_EVENT_HNP_FAILED Macro

File

usb_otg.h

C

```
#define OTG_EVENT_HNP_FAILED 9
```

Description

This is macro OTG_EVENT_HNP_FAILED.

7.3.2.19 OTG_EVENT_NONE Macro

File

usb_otg.h

C

```
#define OTG_EVENT_NONE 2
```

Description

This is macro OTG_EVENT_NONE.

7.3.2.20 OTG_EVENT_RESUME_SIGNALING Macro

File

usb_otg.h

C

```
#define OTG_EVENT_RESUME_SIGNALING 11
```

Description

This is macro OTG_EVENT_RESUME_SIGNALING.

7.3.2.21 OTG_EVENT_SRP_CONNECT Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_CONNECT 7
```

Description

This is macro OTG_EVENT_SRP_CONNECT.

7.3.2.22 OTG_EVENT_SRP_DPLUS_HIGH Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_DPLUS_HIGH 3
```

Description

This is macro OTG_EVENT_SRP_DPLUS_HIGH.

7.3.2.23 OTG_EVENT_SRP_DPLUS_LOW Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_DPLUS_LOW 4
```

Description

This is macro OTG_EVENT_SRP_DPLUS_LOW.

7.3.2.24 OTG_EVENT_SRP_FAILED Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_FAILED 10
```

Description

This is macro OTG_EVENT_SRP_FAILED.

7.3.2.25 OTG_EVENT_SRP_VBUS_HIGH Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_VBUS_HIGH 5
```

Description

This is macro OTG_EVENT_SRP_VBUS_HIGH.

7.3.2.26 OTG_EVENT_SRP_VBUS_LOW Macro

File

usb_otg.h

C

```
#define OTG_EVENT_SRP_VBUS_LOW 6
```

Description

This is macro OTG_EVENT_SRP_VBUS_LOW.

7.3.2.27 ROLE_DEVICE Macro

File

usb_otg.h

C

```
#define ROLE_DEVICE 0
```

Description

Role Defines

7.3.2.28 ROLE_HOST Macro

File

usb_otg.h

C

```
#define ROLE_HOST 1
```

Description

This is macro ROLE_HOST.

7.3.2.29 START_SESSION Macro

File

usb_otg.h

C

```
#define START_SESSION 0
```

Description

Session Defines

7.3.2.30 TOGGLE_SESSION Macro

File

usb_otg.h

C

```
#define TOGGLE_SESSION 2
```

Description

This is macro TOGGLE_SESSION.

7.3.2.31 USB_OTG_FW_DOT_VER Macro

File

usb_otg.h

C

```
#define USB_OTG_FW_DOT_VER 0        // Firmware version, dot release number.
```

Description

Firmware version, dot release number.

7.3.2.32 USB_OTG_FW_MAJOR_VER Macro

File

usb_otg.h

C

```
#define USB_OTG_FW_MAJOR_VER 1          // Firmware version, major release number.
```

Description

Firmware version, major release number.

7.3.2.33 USB_OTG_FW_MINOR_VER Macro

File

usb_otg.h

C

```
#define USB_OTG_FW_MINOR_VER 0          // Firmware version, minor release number.
```

Description

Firmware version, minor release number.

8 Appendix (Frequently Asked Questions, Important Information, Reference Material, etc.)

8.1 Using breakpoints in USB host applications

This section describes how to use breakpoints when running a USB host application without causing communication issues.

Description

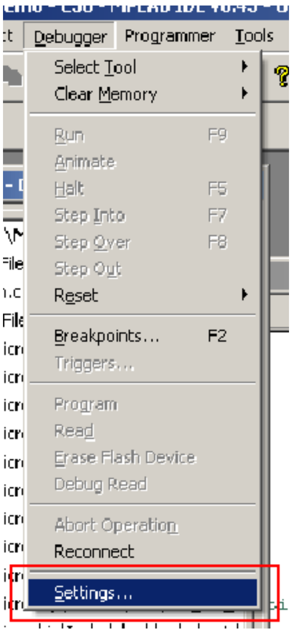
This section describes how to use breakpoints when running a USB host application without causing communication issues.

USB has a periodic packet that is sent on the bus once every millisecond, called the start of frame (SOF) packet, that is used to keep the bus from going into an idle/suspended state. When a the microcontroller hits a breakpoint, both the CPU and the modules on the device stop operation. This will cause the attached USB device to enter the suspend mode. Some programmers implement a method that allows specified peripherals to continue to run even after a breakpoint occurs. This section describes how to enable this feature for the USB peripheral on PIC24F and PIC32 devices.

MPLAB v8.x

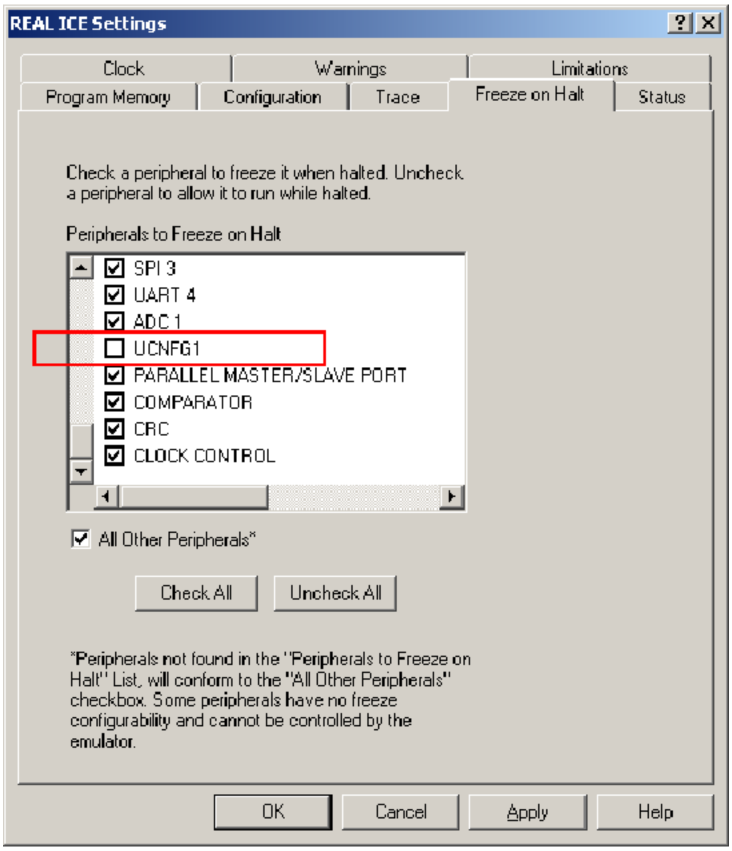
1) Select the desired debugger from the debugger menu

2) Go to the “Debugger->Settings” menu option

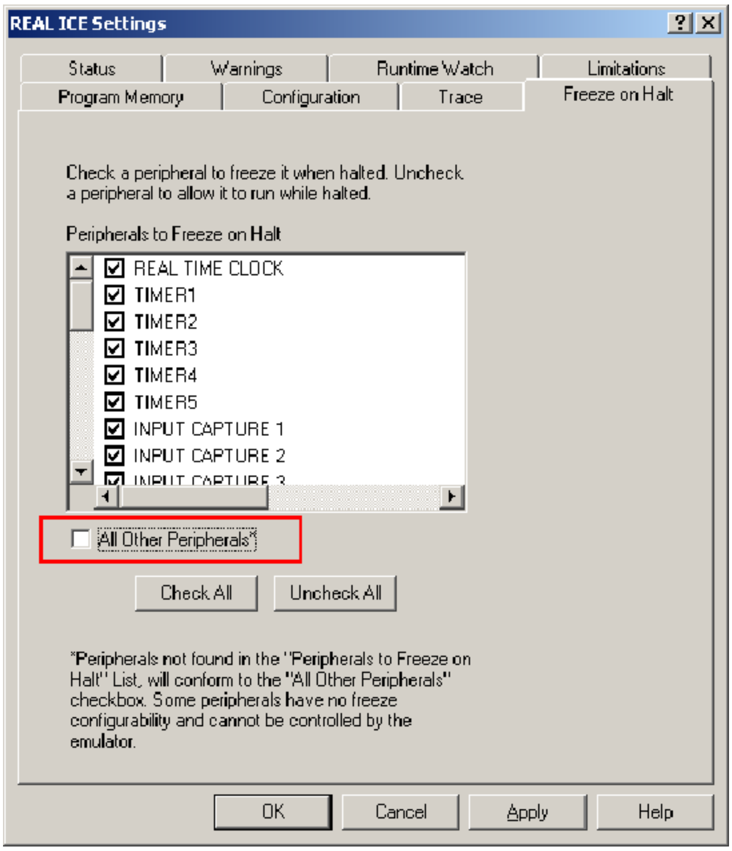


3) Go to the Freeze on Halt tab. For PIC24F devices, uncheck the UCNFG1 box. For PIC32 devices, uncheck the “All other peripherals” box located below the scrolling menu.

PIC24F

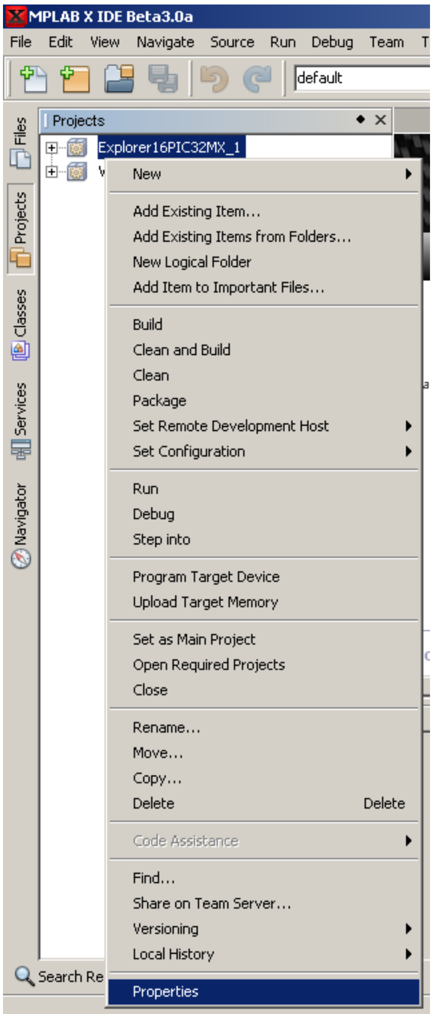


PIC32



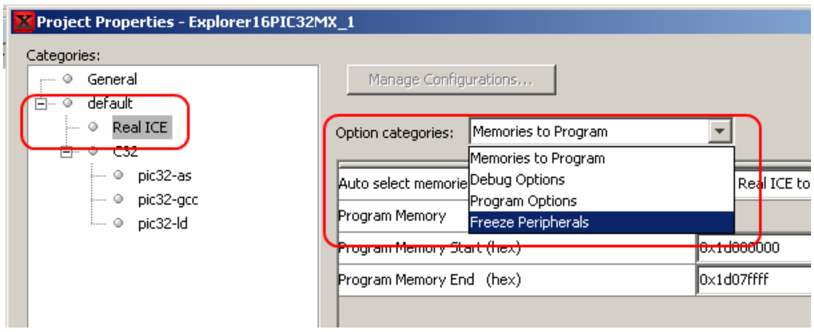
MPLAB X

1) In the projects window, right click on the project you are working on and select properties from the menu that appears.

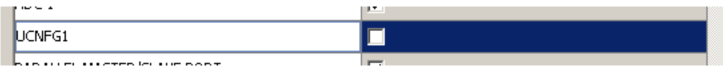


2) In the properties window, select the debugger that you are currently using from the Categories navigation pane.

3) In the resulting form, select "Freeze Peripherals" in the "Option Categories" drop down box.



4) In the resulting list uncheck the box corresponding to the USB peripheral. If there is not one on the list, uncheck "All other peripherals". Please note that on PIC24F the USB module may be named UCNFG1.



8.2 Bootloader Details

This section covers some of the implementation and usage details about the boot loaders.

Description

The detailed descriptions of the boot loader implementations are very part specific. They often involve modified linker scripts and discussions of part specific features and architectural differences (like interrupts and resets). For this reason this section is broken down into sections for each processor product line.

8.2.1 PIC24F Implementation Specific Details

This section covers the PIC24F product line USB boot loaders.

Description

8.2.1.1 Adding a boot loader to your project

This section covers how to add a boot loader to your application.

Description

The boot loader implementations available in the MCHPFSUSB take a two application approach. What this means is that the boot loader and the application are development, compiled, and loaded separately.

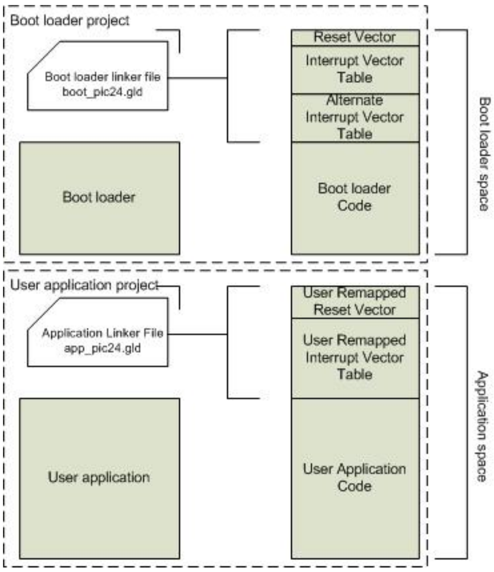
With this approach there are two separate linker scripts that are required, one for the boot loader, and one for the application.

For the PIC24F applications intended to be used with a boot loader, all that is required is to attach the specific linker file designed for the applications of that boot loader to the project.

- No modifications are required to the linker file. Just attach the provided application linker file to the application project without modification.
- No modifications are required to the application code. Just write your code as you always would and attach the provided application linker file to the application project without modification.

The required application linker files are found in the folders that contain the targeted boot loaders. These linker files can be referenced directly from the application projects, or can be copied locally to the project folder.

These provided linker files generate the required code to handle the reset and interrupt remapping sections that are required.



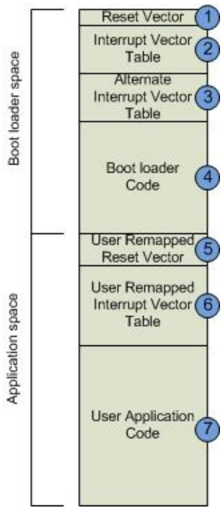
8.2.1.2 Memory Map

This section discusses the various memory regions in the PIC24F device and how they are arranged between the boot loaders and the target applications.

Description

The PIC24F boot loaders have several different special memory regions. Some of these regions are defined by the hardware. Others are part of the boot loader implementation and usage. This section discusses what each of these memory regions are. For more information about how these sections are implemented or how to change them, please refer to the Understanding and Customizing the Boot Loader Implementation (see page 830) section.

The different memory regions are shown below:



1) Reset Vector - the reset vector is defined by the hardware. This is located at address 0x0000. Any reset of the CPU will go to the reset vector. The main responsibility of the reset vector is to jump to the code that needs to be run. In the case of the boot loader, this means jumping to the boot loader code (section 4).

2) The interrupt vector table (IVT) is another section that is defined by the PIC24F hardware. The IVT is a fixed set of addresses that specify where the CPU should jump to in the case of an interrupt event. Each interrupt has it's own vector in the table. When that interrupt occurs, the CPU fetches the address in the table corresponding to that interrupt and jumps to that address.

3) The alternate interrupt vector table (AIVT) behaves just like the IVT. The user must set a bit to select if they are using the IVT or AIVT for their interrupt handling. The IVT is the default. For the current boot loader applications, the AIVT is either used by the boot loader or is not remapped to user space so the AIVT is not available for application use.

4) The boot loader code - This section is where the boot loader code resides. This section handles all of the loading of the new application code.

5) User Remapped Reset Vector - This is a section that is defined by the boot loader. The boot loader must always know how to exit to the application on startup. The User Remapped Reset Vector is used as a fixed address that the boot loader can jump to in order to start an application. The application must place code at this address that starts their application. In the PIC24F implementations this is handled by the application linker file.

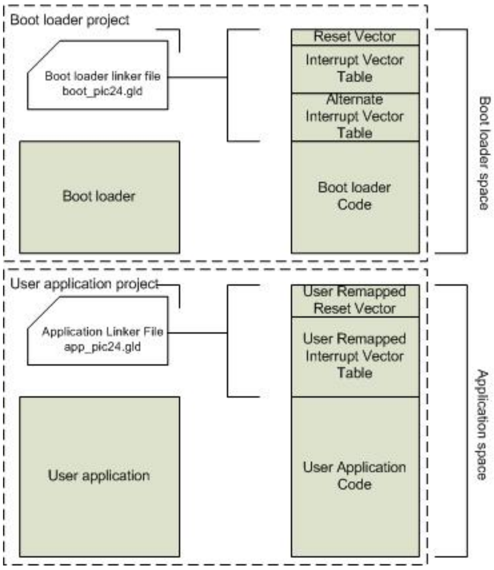
6) User Remapped Interrupt Vectors - Since the IVT is located in the boot loader space, the boot loader must remap all of the interrupts to the application space. This is done using the User remapped interrupt vectors. The IVT in the boot loader will jump to a specific address in the User remapped interrupt vector. The User remapped interrupt vector table jumps to the interrupt handler code defined in the user code. In the PIC24F implementations this table is generated by the application linker file and doesn't require any user modifications.

7) The user application code - this is the main application code for the project that needs to be loaded by the boot loader. In the PIC24F implementation, only the application linker file for the specific boot loader needs to be added to the project. No

other files are required. No changes or additions are required to the user application code either in order to get the code working.

The boot loader and application linker files provided with the MCHPFSUSB enforce all of the memory regions specified above. If an application tries to specify an address outside of the valid range, the user should get a linker error.

Separate linker files are required for the boot loader and the application. These linker files generate the material required for several of the different memory regions in the device. Below is a diagram showing which sections of the final device image are created by the linker files. All of the regions of the device are specified within one of the two linker files. This image merely shows where the content for each of those regions is generated.



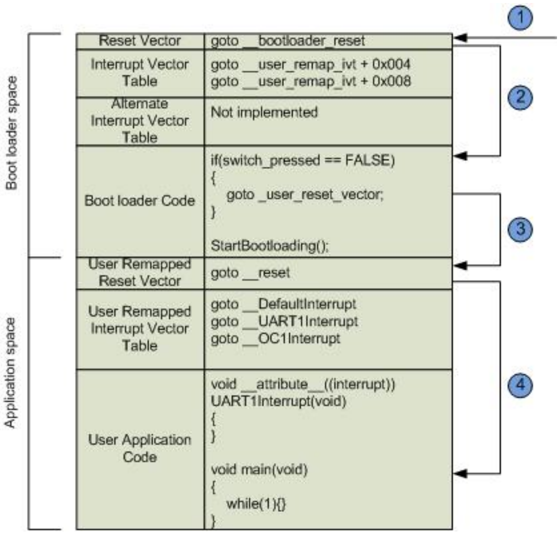
8.2.1.3 Startup Sequence and Reset Remapping

This section discusses how the device comes out of reset and how the control passes between the boot loader and the application.

Description

Before continuing with this section, please review the preceding sections to understand some of the implementation details that aren't discussed in detail in this section. Some of the implementation details of how this works is described the Understanding and Customizing the Boot Loader Implementation (see page 830) section. This section covers the basic flow and how it passes between the boot loader and the application.

In the boot loader implementations provided in MCHPFSUSB library, the boot loader controls the reset vector. This is true for the PIC24F boot loaders as well. The reset vector resides within the boot loader memory space. This means that the boot loader must jump to the target application. This processes in show below in the following diagram and described in the following paragraphs.



1) On PIC24F devices, when a reset occurs the hardware automatically jumps to the reset vector. This is located at address 0x0000. This address resides within the boot loader memory. The compiler/linker for the boot loader code places a 'goto' instruction at the reset vector to the boot loader startup code.

2) The 'goto" instruction at the reset address will jump to the main() function for the boot loader.

3) In the boot loader startup sequence there is a check to determine if the boot loader should run or if the boot loader should jump to the application instead. In the provided examples the code checks a switch to determine if it should remain in the boot loader. If the switch is not pressed then the boot loader jumps to the user_remapped_reset_vector. At this point the control of the processor has just changed from the boot loader to the application.

4) The code at the user_remapped_reset_vector is controlled by the application project, not the boot loader. This vector effectively emulates the behavior that the normal reset vector would if a boot loader wasn't used. In this case it should jump to the startup code for the application. This is done by modified linker script for the application.

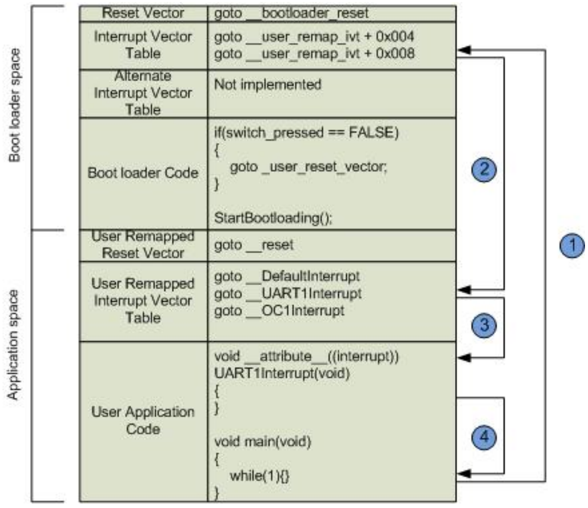
8.2.1.4 Interrupt Remapping

This section discusses how interrupts are handled between the boot loader and application.

Description

Before continuing with this section, please review the preceding sections to understand some of the implementation details that aren't discussed in detail in this section. Some of the implementation details of how this works is described the Understanding and Customizing the Boot Loader Implementation (see page 830) section. This section covers the basic flow and how it passes between the boot loader and the application.

In the boot loader implementations provided in MCHPFSUSB library, the boot loader controls the interrupt vectors for PIC24F devices. The hardware interrupt vector table resides within the boot loader memory space. This means that the boot loader must jump to the appropriate user target application interrupt handler when an interrupt occurs. This processes in show below in the following diagram and described in the following paragraphs.



1) During the course of normal code execution, an interrupt occurs. The CPU vectors to the interrupt vector table (IVT) as described in the appropriate PIC24F datasheet.

2) The IVT is located in boot loader space, but the application needs to handle the interrupt. The boot loader jumps to the correct entry in the User Remapped Interrupt Vector Table. At this point the CPU is jumping from the boot loader memory space to the application memory space and effectively transferring control to the application.

3) At the entry in the User Remapped Interrupt Vector table there is placed a 'goto' instruction that will jump to the appropriate interrupt handler if one is defined in your application and to the default interrupt if there isn't a handler defined. In this way the behavior of the application with or without the boot loader is identical. The User Remapped Interrupt Vector table is created by the application linker file for the specific boot loader in use. This table is automatically generated and doesn't need to be modified. More about how this table is generated can be found in the Understanding and Customizing the Boot Loader Implementation (see page 830).

4) Finally once the interrupt handler code is complete, the code will return from the interrupt handler. This will return the CPU to the instruction that the interrupt occurred before.

8.2.1.5 Understanding and Customizing the Boot Loader Implementation

This sections discusses the customizations that have been made from the default linker scripts in order to make the boot loader work and how to customize these implementations if you wish to change the behavior or location of the boot loader.

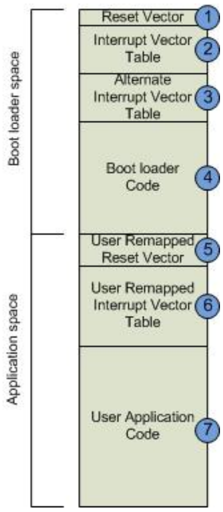
Description

8.2.1.5.1 Memory Region Definitions

This section describes how each of the memory regions gets defined.

Description

First let's take a look how each of the memory regions are defined. The address ranges for each of the regions seen in the diagram below must be defined in either the application linker file or the boot loader linker files.



Below is an excerpt from one of the HID boot loader linker files. This is from the linker script for the boot loader itself so this will be covering sections (1), (2), (3), and (4).

```
/*
** Memory Regions
*/
MEMORY
{
  data  (a!xr) : ORIGIN = 0x800,         LENGTH = 0x4000
  reset        : ORIGIN = 0x0,           LENGTH = 0x4
  ivt          : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt         : ORIGIN = 0x104,         LENGTH = 0xFC
  program (xr) : ORIGIN = 0x400,         LENGTH = 0x1000
  config4      : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3      : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2      : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1      : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

The region named "reset" is defined to start at address 0x0 and has a length of 0x4. This means that the first two instructions of the device are used for the reset vector. This is just enough for one 'goto' instruction. This corresponds to hardware implementation and should not be changed. This defines section (1).

Section (2) is the IVT table. This is defined with the "ivt" memory entry. It starts at address 0x4 and is 0xFC bytes long. This corresponds to hardware implementation and should not be changed.

Section (3) is the AIVT table. This is defined with the "aivt" memory entry. It starts at address 0x104 and is 0xFC bytes long. This corresponds to hardware implementation and should not be changed.

Section (4) is the section for the boot loader code. This section is covered by the "program" entry in the memory table. This section starts at address 0x400 and is 0x1000 bytes long in this example (ends at 0x1400). As you can see with this section it has been decreased from the total size of the device to limit the boot loader code to this specific area. This is how the linker knows where the boot loader code is allowed to reside.

Looking in the corresponding application linker file will result in a similar table.

```
/*
** Memory Regions
*/
```

```
MEMORY
{
  data  (a!xr)   : ORIGIN = 0x800,         LENGTH = 0x4000
  reset          : ORIGIN = 0x0,           LENGTH = 0x4
  ivt            : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt           : ORIGIN = 0x104,         LENGTH = 0xFC
  app_ivt        : ORIGIN = 0x1400,        LENGTH = 0x10C
  program (xr)   : ORIGIN = 0x1510,        LENGTH = 0x296E8
  config4        : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3        : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2        : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1        : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

Note that the "reset", "ivt", and "aivt" sections are all still present in the application linker script. These sections remain here so that applications compiled with the boot loader can be programmed with or without the boot loader. This aids in the development of the application without having to use the boot loader while maintaining identical interrupt latency and memory positioning.

Sections (5) and (6) are created in the special "app_ivt" section. The following discussion topic describes how the content of this section is created. This entry in the memory table is how the space for that area is allocated. Note that the "app_ivt" section starts at address 0x1400 (the same address that the boot loader ended at). Since different parts have different number of interrupts, the size of the "app_ivt" section may change.

The "program" memory section has changed for the application space. It starts at address 0x1510 in this example. This will vary from part to part based on the size of the "app_ivt" section. The "program" memory section corresponds to the user application code (section (7)). Note that it takes up the rest of the memory of the device that is available to load.

8.2.1.5.2 Special Region Creation

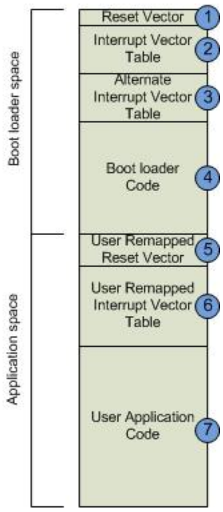
This section covers how each of the special memory regions are created/populated within the linker files.

Description

The Memory Region Definitions (see page 831) section described how each of the memory regions are defined. This allocates the room for each of the memory regions.

This discussion covers how the values of some of the special memory regions are created/populated. Please refer to the earlier sections for an understanding of how the reset and interrupt remapping works before proceeding through this section.

Let's take a look at each of the memory regions in order. Please note that there are two linker scripts, one for the boot loader and one for the application. In order for some of these section definitions to make sense, we will be showing excerpts from either or both of these files for any given section. Please pay close attention to which linker file we are referring to when we show an example.



1) Section (1) is the reset vector. This belongs to the boot loader space so this is located in the boot loader linker file. What we need at the reset vector is a jump to the start of the boot loader code. In the boot loader linker script:

```
/*
** Reset Instruction
*/
.reset :
{
      SHORT(ABSOLUTE(__reset));
      SHORT(0x04);
      SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
      SHORT(0);
} >reset
```

The code in this section generates a "goto __reset" instruction located in the "reset" memory section. This will cause the CPU to jump to the boot loader startup code after any device reset. This is common code that is present in any default linker script for PIC24F.

2) The second section is the IVT. In the IVT we need to jump to the user's remapped IVT table.

```
__APP_IVT_BASE = 0x1400;
.ivt __IVT_BASE :
  {
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x004); /* __ReservedTrap0*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x008); /* __OscillatorFail*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x00C); /* __AddressError*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x010); /* __StackError*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x014); /* __MathError*/
...
    LONG(ABSOLUTE(__DEFAULT_VECTOR)); /* __Interrupt116 not implemented */
    LONG(ABSOLUTE(__DEFAULT_VECTOR)); /* __Interrupt117 not implemented */
```

```
  } >ivt
```

This linker code will place the _APP_IVT_BASE constant + an offset address at each of the IVT vector entries. This will cause the CPU to jump to the specified vector in the user's remapped IVT table.

Note that each entry is 4 bytes away from the previous entry. Is is because the resulting remapped IVT will need to use "goto" instructions at each entry in order to reach the desired handler. The "goto" instruction takes two instruction words at 2 bytes of memory address each.

3) Section (3), the AIVT, is either not used or is used by the boot loader and shouldn't be used by the application. If the boot loader requires interrupts, then it uses the AIVT and switches to AIVT interrupts before starting and switches back to the IVT before jumping to the customer code. No linker modifications are required here. For boot loaders that don't require interrupts, some have the AIVT section removed since they are not remapped to the user space and not used by the boot loader.

4) Section (4), the boot loader code - the only modification required in the linker script for the boot loader code is the changes to the memory region definitions discussed previously in the Memory Region Definitions (see page 831) section.

5) Section (5) is the user remapped reset. This is the address where the boot loader jumps upon completion. This address needs to be at a fixed location in code that both the boot loader and the application know about. At this address there needs to be a jump to the user application code. In the application linker script:

```
  .application_ivt __APP_IVT_BASE :
  {
    SHORT(ABSOLUTE(__reset)); SHORT(0x04); SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__ReservedTrap0) ? ABSOLUTE(__ReservedTrap0) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__ReservedTrap0) ?
(ABSOLUTE(__ReservedTrap0) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__OscillatorFail) ?
(ABSOLUTE(__OscillatorFail) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__AddressError) ? ABSOLUTE(__AddressError) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__AddressError) ?
(ABSOLUTE(__AddressError) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
```

This section of code has been added to the default linker script. This creates a section in code located at __APP_IVT_BASE address. In this case the __APP_IVT_BASE address is also defined in the application linker file:

```
__APP_IVT_BASE = 0x1400;
```

This address must match exactly between the boot loader code, boot loader linker file, and the application linker file. If any of these do not match then the linkage between the interrupt remapping or reset remapping will not work and the application will fail to run properly.

The first entry in this table is the user remapped reset. This code generates a "goto __reset" at address __APP_IVT_BASE. This allows the boot loader to jump to this fixed address to then jump to the start of the user code (located at the __reset label).

6) Section (6) is the remapped IVT table. This section allows the interrupt to be remapped from the boot loader space to the application space. In order to do this the boot loader must either know the exact address of every interrupt handler, or must have another jump table that it jumps to in order to redirect it to the correct interrupt handler. The second approach is the one used in the implemented boot loaders. This is implemented in the following table:

```
  .application_ivt __APP_IVT_BASE :
  {
    SHORT(ABSOLUTE(__reset)); SHORT(0x04); SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__ReservedTrap0) ? ABSOLUTE(__ReservedTrap0) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__ReservedTrap0) ?
(ABSOLUTE(__ReservedTrap0) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__OscillatorFail) ?
(ABSOLUTE(__OscillatorFail) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
```

```
    SHORT(DEFINED(__AddressError) ? ABSOLUTE(__AddressError) :
ABSOLUTE(__DefaultInterrupt));  SHORT(0x04);  SHORT(DEFINED(__AddressError) ?
(ABSOLUTE(__AddressError) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
```

This first entry in the table is the remapped reset vector that we just discussed. The second entry in the table is the first possible interrupt. In this case it is the ReservedTrap0 interrupt. This line of linker code will look for the __ReservedTrap0 interrupt function. If it exists it will insert a "goto __ReservedTrap0" at the second address in this table. If it doesn't find the __ReservedTrap0 function, it will put a "goto __DefaultInterrupt" at this entry in the table. In this way just by defining the appropriate interrupt handler function in the application code, the linker will automatically create the jump table entry .required.

Looking at an example application_ivt table as generated by the linker script where the ReservedTrap0 interrupt is not defined and the OscillatorFail and AddressError handlers are defined, starting at address _APP_IVT_BASE you will have the following entries in program memory:

goto __reset

goto __DefaultInterrupt

goto __OscillatorFail

goto __AddressError

...

7) Section (7), the user application code - the only modification to the linker script required for the application code is the changes to the memory region definitions discussed previously in the Memory Region Definitions (see page 831) section.

8.2.1.5.3 Changing the memory foot print of the boot loader

This section covers how to modify how much memory is used by the boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader.

Description

This section covers how to modify how much memory is used by the boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader.

Each boot loader has different memory sections and implementations so there is a section covering each boot loader individually. Please refer to the section corresponding to the boot loader in question.

8.2.1.5.3.1 HID boot loader

This section covers how to modify the size of the HID boot loader.

Description

This section covers how to modify the size of the HID boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader. The boot loaders provided by default assume full optimizations and may not work with compilers that don't have access to full optimizations.

Please read all of the other topics in the PIC24F boot loader section before proceeding in this topic. This topic will show where the modifications need to be made and how they need to match up, but will not describe what the sections that are being modified are or how they are implemented. This information is in previous sections.

There are three places that require corresponding changes: the boot loader linker script, the application linker script, and the boot loader code. You may wish to make copies of the original files so that you preserve the original non-modified files.

In the following examples we will be increasing the size of the boot loader from 0x1400 to 0x2400 in length.

First start by determining the size that you want the boot loader to be. This must be a multiple of an erase page. On many PIC24F devices there is a 512 instruction word erase page (1024 addresses per page). Please insure that the address you select for the end of the boot loader corresponds to a page boundary. There are several ways to determine the size of the boot loader application. Below is an example of one method.

1) Remove the boot loader linker script provided if it is causing link errors due either to optimization settings or added code.

2) Build the project

3) Open the memory window and find the last non-blank address in the program memory space.

4) Find the next flash erase page address after this address. Add any additional buffer room that you might want for future boot loader development, growth, or changes. Use this address as your new boot loader end address.

Once the end address of the boot loader is known, start by modifying the boot loader linker script program memory region to match that change. The boot loader linker script can either be found in the folder containing the boot loader project file or in a folder that is specified for boot loader linker scripts. In the linker script find the memory regions.

```
MEMORY
{
  data  (a!xr) : ORIGIN = 0x800,         LENGTH = 0x4000
  reset        : ORIGIN = 0x0,           LENGTH = 0x4
  ivt          : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt         : ORIGIN = 0x104,         LENGTH = 0xFC
  program (xr) : ORIGIN = 0x400,         LENGTH = 0x2000
  config4      : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3      : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2      : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1      : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

Change the LENGTH field of the program memory section to match the new length. Note that this is length and not the end address. To get the end address, please add LENGTH + ORIGIN.

Next, locate the __APP_IVT_BASE definition in the linker file. Change this to equal the end address of your boot loader.

```
__APP_IVT_BASE = 0x2400;
```

Once the length of the boot loader is changed, you will need to make similar changes in the application boot loader linker script. The application boot loader linker scripts are typically found in a folder with the boot loader project. In the application linker file, locate the memory regions section. In this section there are three items that need to change.

1. The first is the ORIGIN of the app_ivt section. This needs to be modified to match the new end address of the boot loader.
2. Second, move the ORIGIN of the program memory section to the ORIGIN of app_ivt + the LENGTH of the app_ivt section so that the program memory starts immediately after the app_ivt section.
3. Last, change the LENGTH field of the program section so that it goes to the end of the program memory of the device.

Remember that the LENGTH field is the length starting from the origin and not the end address. An easy way to make sure that this address is correct is by just subtracting off from the LENGTH the same amount that was added to the ORIGIN.

```
MEMORY
{
  data  (a!xr)   : ORIGIN = 0x800,         LENGTH = 0x4000
  reset          : ORIGIN = 0x0,           LENGTH = 0x4
  ivt            : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt           : ORIGIN = 0x104,         LENGTH = 0xFC
  app_ivt        : ORIGIN = 0x2400,        LENGTH = 0x110
  program (xr)   : ORIGIN = 0x2510,        LENGTH = 0x286E8
  config4        : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3        : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2        : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1        : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

The final changes that needs to be made are in the boot loader code itself. Open up the boot loader project.

1. Find the ProgramMemStart definition in the main.c file. Change the start address to match the new address.

```
#define ProgramMemStart                    0x00002400
```

2. Next find the #ifdef section that applies to the device that you are working with. This section will contain definitions used by the boot loader to determine what memory is should erase and re-write.

```
#if defined(__PIC24FJ256GB110__) || defined(__PIC24FJ256GB108__) ||
defined(__PIC24FJ256GB106__)
    #define BeginPageToErase             5          //Bootloader and vectors occupy first
six 1024 word (1536 bytes due to 25% unimplemented bytes) pages
    #define MaxPageToEraseNoConfigs      169        //Last full page of flash on the
PIC24FJ256GB110, which does not contain the flash configuration words.
    #define MaxPageToEraseWithConfigs    170        //Page 170 contains the flash
configurations words on the PIC24FJ256GB110.  Page 170 is also smaller than the rest of the
(1536 byte) pages.
    #define ProgramMemStopNoConfigs      0x0002A800 //Must be instruction word aligned
address.  This address does not get updated, but the one just below it does:
                                                    //IE: If AddressToStopPopulating =
0x200, 0x1FF is the last programmed address (0x200 not programmed)
    #define ProgramMemStopWithConfigs    0x0002ABF8 //Must be instruction word aligned
address.  This address does not get updated, but the one just below it does: IE: If
AddressToStopPopulating = 0x200, 0x1FF is the last programmed address (0x200 not programmed)
    #define ConfigWordsStartAddress      0x0002ABF8 //0x2ABFA is start of CW3 on
PIC24FJ256GB110 Family devices
    #define ConfigWordsStopAddress       0x0002AC00
```

3. Modify the BeginPageToErase to indicate which page is the first page it should erase. This will be the ProgramMemStart/Page Size. In this case we are starting at 0x2400 and each page is 0x400 so this should now be 9.

```
    #define BeginPageToErase             9
```

4. Locate the start of the main() function. In the first few lines of code there is a check to determine of the code should stay in the boot loader or jump to the application code. Change the address in the "goto" statement to match the new end of the boot loader and start of the application.

```
__asm__("goto 0x2400");
```

This should be all of the changes required in order to change the size of the HID boot loader.

Please note that since the boot loader and the application code are developed as two separate applications, they do not need to use the same optimization settings.

8.2.1.5.3.2 MSD boot loader

This section covers how to modify the size of the MSD boot loader.

Description

This section covers how to modify the size of the MSD boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader. The boot loaders provided by default assume full optimizations and may not work with compilers that don't have access to full optimizations.

Please read all of the other topics in the PIC24F boot loader section before proceeding in this topic. This topic will show where the modifications need to be made and how they need to match up, but will not describe what the sections that are being modified are or how they are implemented. This information is in previous sections.

There are three places that require corresponding changes: the boot loader linker script, the application linker script, and the boot loader code. You may wish to make copies of the original files so that you preserve the original non-modified files.

In the following examples we will be increasing the size of the boot loader from 0xA000 to 0xC000.

First start by determining the size that you want the boot loader to be. This must be a multiple of an erase page. On many PIC24F devices there is a 512 instruction word erase page (1024 addresses per page). Please insure that the address you select for the end of the boot loader corresponds to a page boundary. There are several ways to determine the size of the boot loader application. Below is an example of one method.

1) Remove the boot loader linker script provided if it is causing link errors due either to optimization settings or added code.

2) Build the project

3) Open the memory window and find the last non-blank address in the program memory space.

4) Find the next flash erase page address after this address. Add any additional buffer room that you might want for future boot loader development, growth, or changes. Use this address as your new boot loader end address.

Once the end address of the boot loader is known, start by modifying the boot loader linker script program memory region to match that change. The boot loader linker script can either be found in the folder containing the boot loader project file or in a folder that is specified for boot loader linker scripts. In the linker script find the memory regions.

```
MEMORY
{
  data  (a!xr) : ORIGIN = 0x800,         LENGTH = 0x4000
  reset        : ORIGIN = 0x0,           LENGTH = 0x4
  ivt          : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt         : ORIGIN = 0x104,         LENGTH = 0xFC
  program (xr) : ORIGIN = 0x400,         LENGTH = 0xBC00
  config4      : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3      : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2      : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1      : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

Change the LENGTH field of the program memory section to match the new length. Note that this is length and not the end address. To get the end address, please add LENGTH + ORIGIN.

Next, locate the __APP_IVT_BASE definition in the linker file. Change this to equal the end address of your boot loader.

```
__APP_IVT_BASE = 0xC000;
```

Once the length of the boot loader is changed, you will need to make similar changes in the application boot loader linker script. The application boot loader linker scripts are typically found in a folder with the boot loader project. In the application linker file, locate the memory regions section. In this section there are three items that need to change.

1. The first is the ORIGIN of the app_ivt section. This needs to be modified to match the new end address of the boot loader.
2. Second, move the ORIGIN of the program memory section to the ORIGIN of app_ivt + the LENGTH of the app_ivt section so that the program memory starts immediately after the app_ivt section.
3. Last, change the LENGTH field of the program section so that it goes to the end of the program memory of the device.

Remember that the LENGTH field is the length starting from the origin and not the end address. An easy way to make sure that this address is correct is by just subtracting off from the LENGTH the same amount that was added to the ORIGIN.

```
MEMORY
{
  data  (a!xr)   : ORIGIN = 0x800,         LENGTH = 0x4000
  reset          : ORIGIN = 0x0,           LENGTH = 0x4
  ivt            : ORIGIN = 0x4,           LENGTH = 0xFC
  aivt           : ORIGIN = 0x104,         LENGTH = 0xFC
  app_ivt        : ORIGIN = 0xC000,        LENGTH = 0x110
  program (xr)   : ORIGIN = 0xC110,        LENGTH = 0x1EAE8
  config4        : ORIGIN = 0x2ABF8,       LENGTH = 0x2
  config3        : ORIGIN = 0x2ABFA,       LENGTH = 0x2
  config2        : ORIGIN = 0x2ABFC,       LENGTH = 0x2
  config1        : ORIGIN = 0x2ABFE,       LENGTH = 0x2
}
```

The final changes that needs to be made are in the boot loader code itself. Open up the boot loader project.

1. Find the processor section in boot_config.h that applies to the processor that you are using. This section should contain a definition APPLICATION_ADDRESS. This address indicates the address where the user application reset vector resides. Change this address to match the new user reset address.

```
#define APPLICATION_ADDRESS          0xC000ul
```

2. Next find the PROGRAM_FLASH_BASE. This address specifies the starting address of memory where the device will start to erase and reprogram. Please make sure to make this address at the start of an erase page on the device.

```
#define PROGRAM_FLASH_BASE           0xC000ul
```

3. Next modify the PROGRAM_FLASH_LENGTH to match the new flash length. This length specifies the number of bytes starting from the PROGRAM_FLASH_BASE that is valid for the application.

```
#define PROGRAM_FLASH_LENGTH         0x1EB00
```

This should be all of the changes required in order to change the size of the HID boot loader.

Please note that since the boot loader and the application code are developed as two separate applications, they do not need to use the same optimization settings.

8.2.2 Important Considerations

There are some important topics that need to be considered when developing an application for a boot loader. This section will cover some of these topics that need to be considered.

8.2.2.1 Configuration Bits

This section covers some topics related to configuration bits.

Description

Matching configuration bits between application and boot loader

While the code code space of the application and the boot loader are separate, they both must run on the same device. As such they both use the same configuration bits. Some boot loaders are able to update configuration bits. Others are not. If the boot loader is able to load configuration bits, the application designer should be careful to select a setting that works for both the application as well as the boot loader. It is possible to change the configuration bits into a setting that isn't compatible with the boot loader.

For example: if the boot loader wasn't design to handle a watch dog timer (doesn't clear the watch dog timer) and the application enables the watch dog timer through the configuration bits, it is possible that the boot loader will never be able to load a new set of code again because it gets a watch dog timer event before the loading of a file is complete.

The previous example is just one of many ways that changes in configuration bits by the loaded application files can be dangerous to future use of the boot loader. Caution should be used when changing the configuration bits if the boot loader enables that feature.

8.2.2.2 Boot Loader Entry

This section covers topics related to boot loader entry.

Description

While some applications it is desirable to enter the boot loader after starting the application, designers should consider have a mechanism to enter the boot loader without jumping to the application in case of an application failure.

Example: An application wants to run and when the user selects that they want to load a new firmware, they go to the boot loader. If there is a power failure during the loading process it is possible that there isn't a valid application image that will run successfully. If there isn't a mechanism to detect this failure or a method to enter the boot loader directly without going to the application, then this device could be rendered useless.

Even if the main method of boot loader entry is directly from the application, a secondary method that doesn't require an application should be considered for such circumstances.

8.2.2.3 Interrupts

This section covers some topics related to interrupts.

Description

PIC24F

Because the interrupts must be remapped from the the IVT to the application space, there is additional latency from the time that the interrupt is generated to the time that the first line in the interrupt handler is executed. There is one additional inserted "goto" instruction resulting in a 2 cycle increased interrupt latency.

Note that the provided application linker files allow projects built for the PIC24F boot loaders to be either programmed or boot loaded. In both cases the interrupt latency and memory organization are identical allowing users to develop their application without having to use the boot loader but having identical performance and memory usage as if they were using the boot loader.

The PIC24F boot loaders only remap the main interrupt vector table (IVT). They don't remap the alternate interrupt vector table (AIVT). Please see the PIC24F Implementation details for more information.

PIC32MX

The PIC32MX processor has a programmable interrupt vector table address, both the boot loader and the application projects can have their own interrupt vector tables. The boot loader and application code vector from their interrupt tables directly thus there are no special requirements or changes in latency for the PIC32MX family while using the USB boot loaders.

8.3 Notes on .inf Files

Describes important information about .inf file usage and behavior.

Description

Upon initially plugging in a USB device, in some cases Windows will prompt the user for a driver. Rather than having users manually copy .sys files (driver binary files) into important system directories (such as within the “\Windows\system32\” directory structure) and manually add registry entries, Windows automates the driver installation process through the use of .INF files. INF files are plain text (can be edited with notepad) installation instruction script files.

Some types of USB devices will not require .INF files or user provided drivers (for example, a HID class mouse). For these types of devices, the operating system makes use of drivers already built into/distributed with the operating system, so no user provided driver or .INF file is necessary.

For other types of devices, Windows will prompt the user for a driver. In these cases, point Windows to the .INF file relevant for the USB device. All of the example projects included in the MCHPFSUSB framework which need an INF file are provided with an example INF file. The INF file will need slight modification (most importantly to change the VID and PID) before commercial distribution.

The INF file for the custom demo can be found in <Install Directory>\USB Tools\MCHPUSB Custom Driver\MCHPUSB Driver\Release.

The INF file for the CDC demos can be found in <Install Directory>\USB Tools\USB CDC Serial Demo\inf\win2k_winxp.

8.4 Vendor IDs (VID) and Product IDs (PID)

Describes important information about Vendor IDs (VID) and Product IDs (PID).

Description

Every USB product line must have a unique combination of VID and PID. All firmware examples use Microchip's VID (0x04d8) and a unique PID. Prior to manufacturing and marketing a new USB product, the VID and PID need to be changed. New VID and PID numbers can be obtained by purchasing a VID from the USB Implementers Forum:

http://www.usb.org/developers/vendor

Alternatively, Microchip has a free VID sublicensing program. An application form for obtaining a PID (for use with Microchip’s VID: 0x04d8) from Microchip can be obtained by clicking here for the direct link.

Once a new VID/PID combination is obtained, both the firmware and the .INF file (when applicable) will need to be updated.

To modify the VID/PID in one of the example USB firmware projects, open the usb_descriptors.c file (found in each of the demo folders). They should appear in the table used for the USB Device Descriptor. Change both values as needed.

To modify the VID/PID in the .INF file, open the relevant INF file and search for the “[DeviceList]” sections. There are two sections, one for 32-bit and one for 64-bit, both sections should be identical. In these sections, some text will appear with the form “USB\VID_xxxx&PID_yyyy”. Update the “xxxx” and “yyyy” sections with the new hexadecimal format VID/PID values.

8.5 Using a diff tool

This section will cover the basics of using a diff tool to compare two different sets of code to evaluate the differences. This section will cover a couple of different tools available and their basic functionality. This section doesn't cover all of the features available in each tool nor does this section cover all of the possible diff tools available.

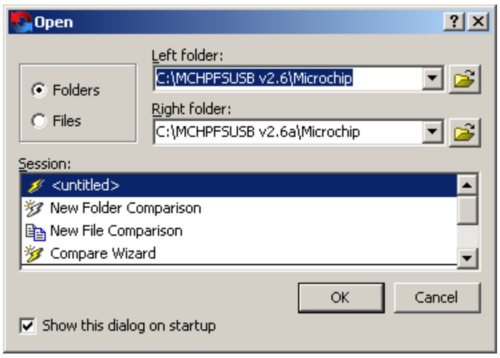
8.5.1 Beyond Compare

Beyond Compare is a commercial available differencing software from Scooter Software, Inc (http://www.scootersoftware.com/).

This demonstration is based on Beyond Compare v2.2.7. There may be interface or features changes in other versions of this software. Please refer to the software's documentation for a more detailed and updated description of the functionality.

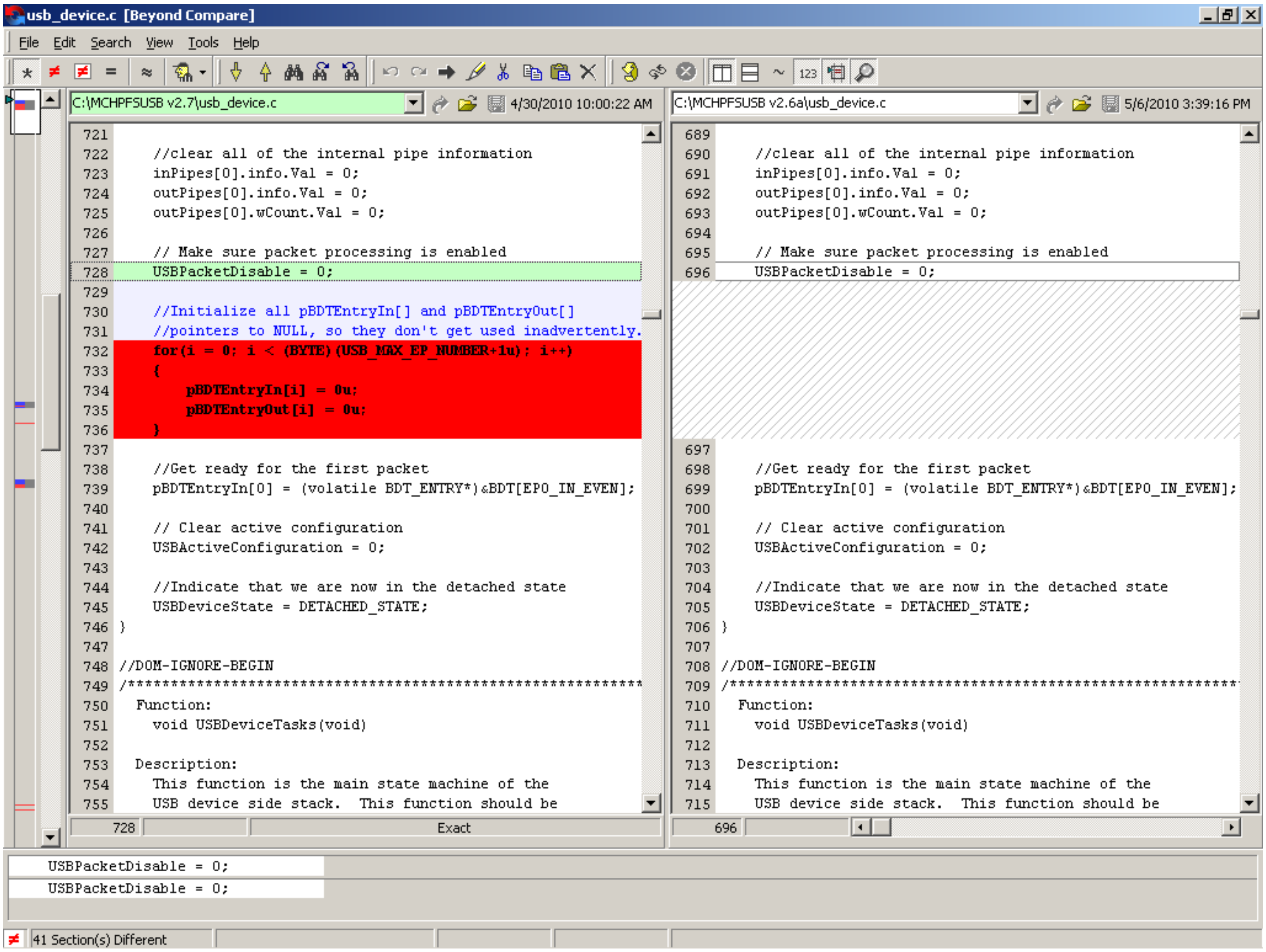
Creating a comparison

There are a couple of ways to create a comparison with Beyond Compare. The first is to open the Beyond Compare program from the Start menu. This opens a window that allows you to either compare two files or two folders.



Beyond compare also has a right click menu option that allows the user to right click on two files and compare them to each other.

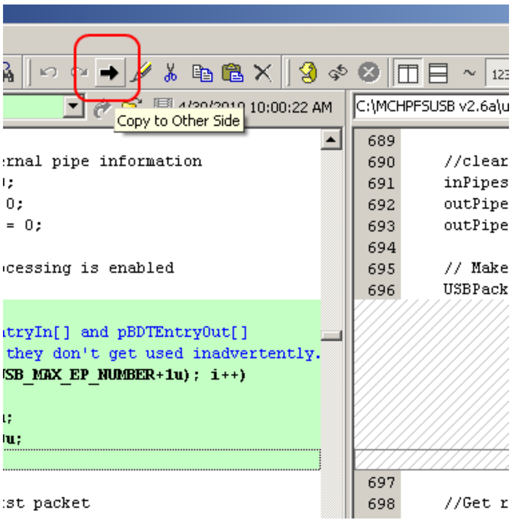
When two files are compared together, the differences between the two files are highlighted. On the left of the main windows there is also a difference navigation bar that shows where the differences are in the file. You can click on this bar to go to that location in the code.



<u>Moving changes</u>

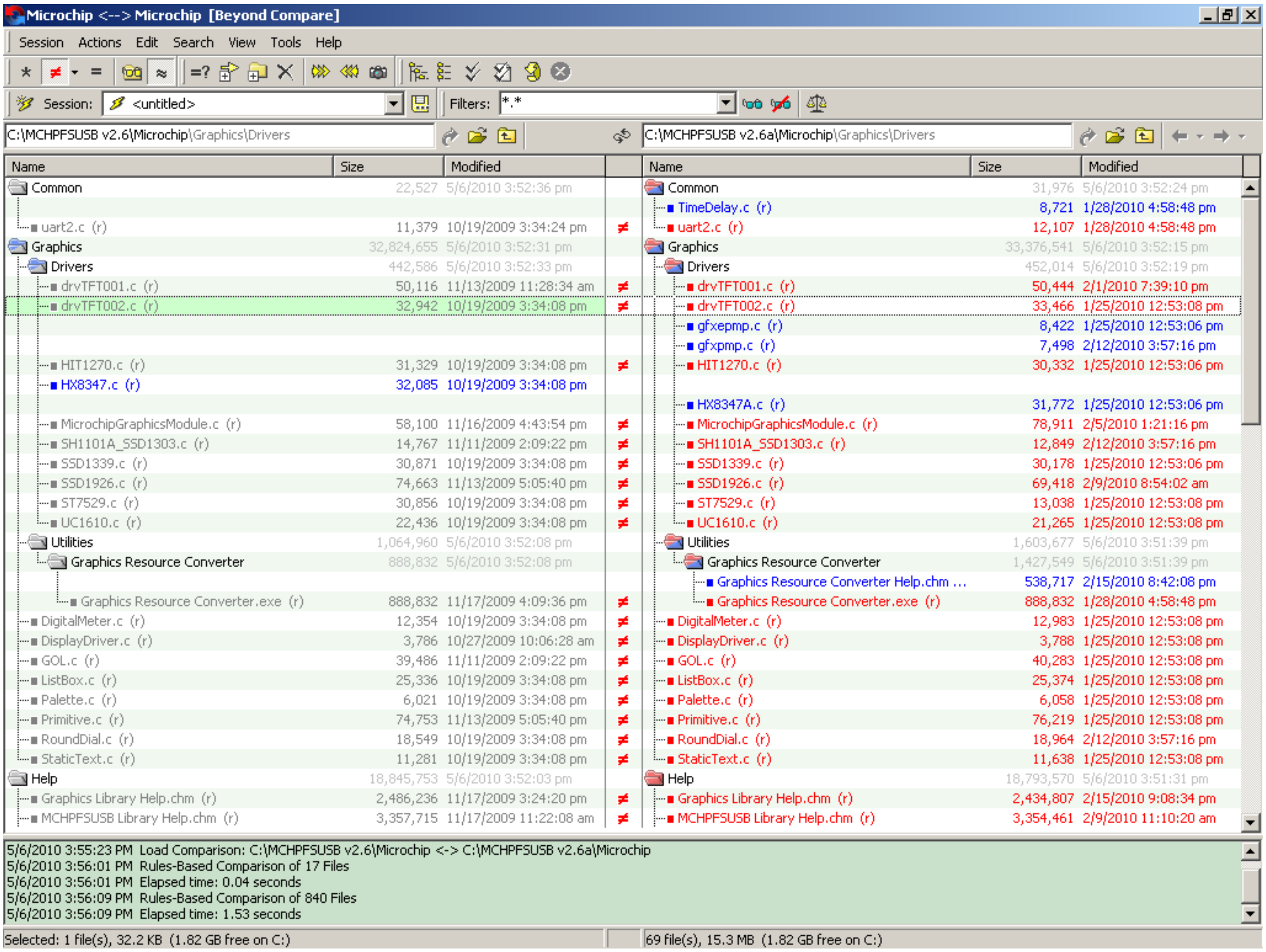
Once a difference between two files is detected, it is easy to move that change from one file to the other. In Beyond

Compare simply highlight the lines that need to move, and click on the "Copy to other side" button that is shown below.



Comparing Folders

A feature of Beyond Compare is that it allows you to compare two folders against each other. Once two folders are selected in the program (or through the right-click menu option discussed before), the folders are compared against each other. At this point of time the contents of the folders aren't compared to see if they are different, just if the files are present or not.



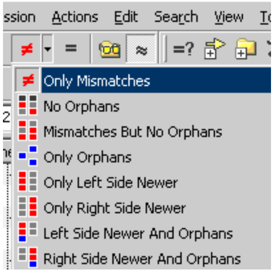
Double clicking on a file will open a file difference instance showing the difference between the two files.

To compare all of the contents against each other you can select all of the files by expanding all of the folders,

selecting all of the files,

and running the file comparison tool.

To see only files that are different, select the "Only mismatch" from the comparison tool.



8.5.2 MPLAB X (NetBeans)

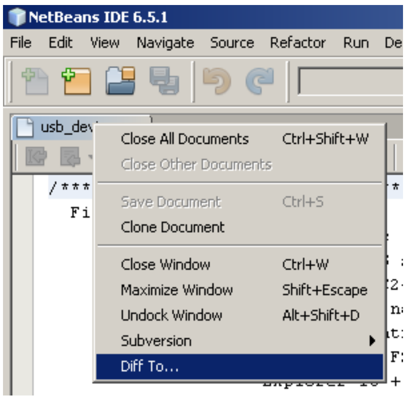
MPLAB X (NetBeans) is an open source IDE that has built in differencing functionality (www.microchip.com/mplabx and http://netbeans.org/).

This demonstration is based on NetBeans v6.5.1 but also applies to MPLAB X beta 7.02. There may be interface or features changes in other versions of this software. Please refer to the software's documentation for a more detailed and updated description of the functionality.

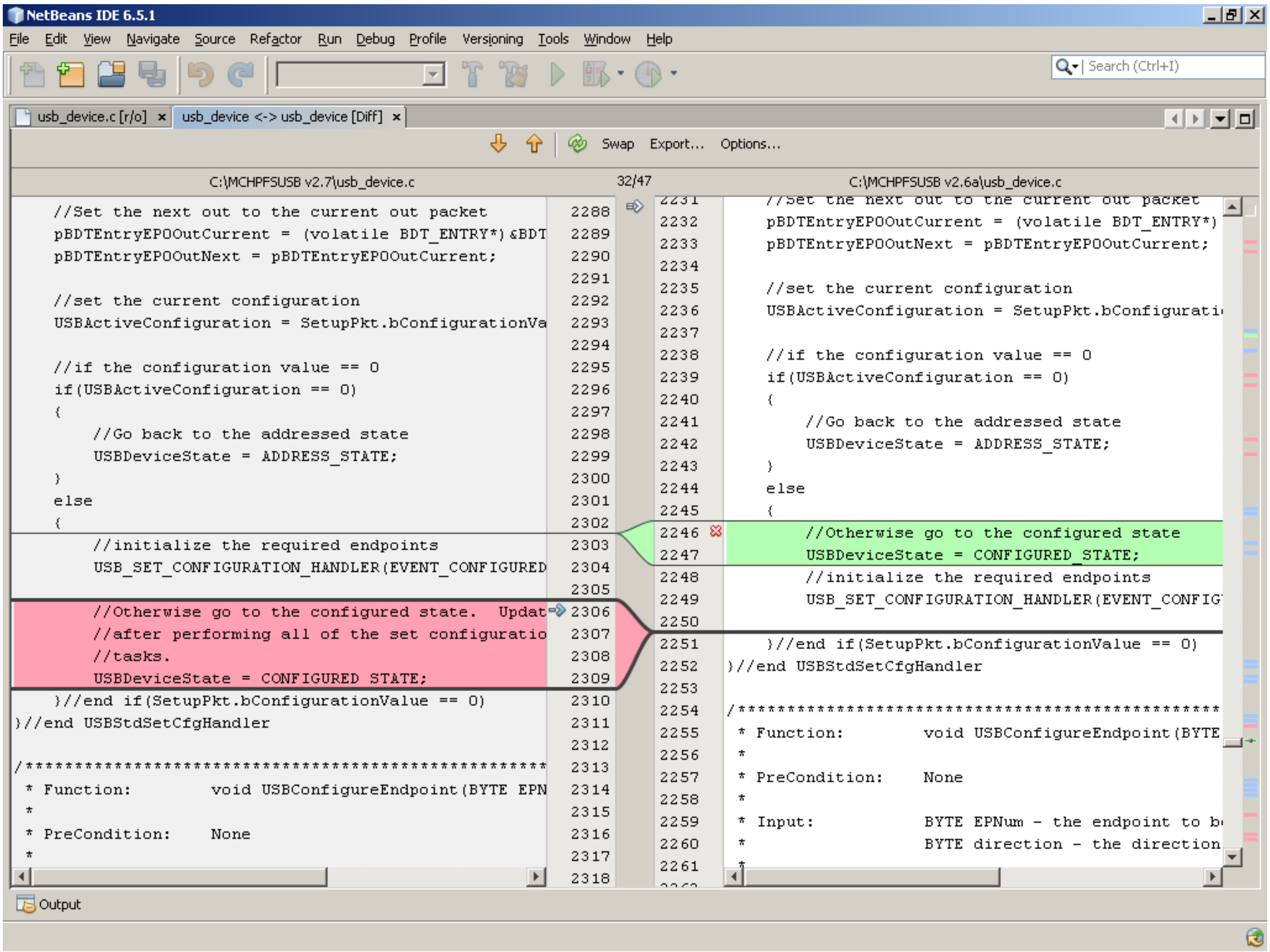
Creating a comparison

To compare two files in MPLAB X (NetBeans), one of the files must first be opened. This can be done using the "File->Open File" menu option.

Once one file is open, right click on the tab with the file name. In this menu there should be a "Diff to..." option that is available.

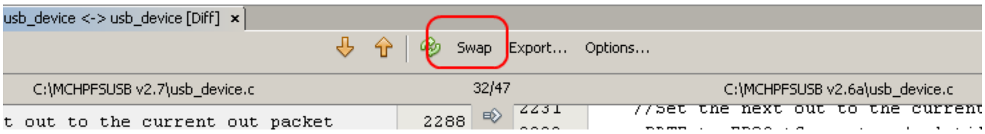


This option will open a new file window. In this file window select the file that you want to compare against. Once this file is selected the two files will sit side by side. On the right hand side of the window there is a difference navigation bar. By clicking on any of these highlighted sections, it will bring you to the difference in the two files.

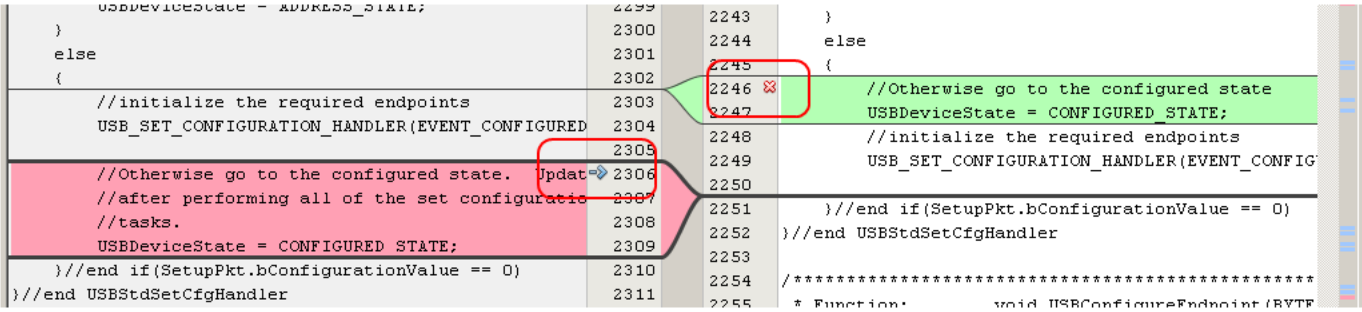


Moving changes

Changes in MPLAB X (NetBeans) are always made from the left file to the right file. If the files were opened in the opposite order as the changes that need to be made, you can click the "Swap" button that is just above both of the two files. This will exchange the position of the two files.



To move a change, click on the icon that is near the center bar between the differences. An arrow "->" indicator on the left window indicates that the changes in that section will be moved from the left file to the right file. An "X" found on the right file in a green section indicates that the source found in that section will be deleted when you click the "X".



Please note that changes made to the files via the MPLAB X (NetBeans) comparison tool may be final. You may not be able to undo these effects to please use caution when making changes.

9 Trademark Information

The Microchip name and logo, the Microchip logo, MPLAB, and PIC are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

PICDEM and PICTail are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

SD is a trademark of the SD Association in the U.S.A and other countries

Index

_

A

B

C

D

E

F

G

H

P

R

S

T

U

V

W