

MSP430™ USB HID Windows API Programmer's Guide

*William Goh, Keith Quiring**MSP430 Applications*

1 Introduction

The USB Human Interface Device (HID) class is one of the most widely-supported device classes among operating systems. Although originally developed for mice and keyboards, it has many advantages in general-purpose usage.

A common choice for general-purpose USB applications is the virtual COM port, which can be implemented using the Communications Device Class (CDC). COM ports are easy to implement on host platforms, and they are well understood by developers. A downside is that USB virtual COM port solutions for Windows machines require the end user to go through a device installation process. In addition to being a minor hassle, this process can go wrong if the user chooses the wrong options. Further, some users in corporate environments do not have administrative rights, so they are not able to do device installations without the help of a network administrator. In contrast, an HID device loads silently, bypassing all these issues.

HID does have a few disadvantages. Compared to virtual COM ports, many software engineers are unfamiliar with its use. HID relies on "reports" to carry the data, and often these complex formats provide no real value to a general-purpose application. The bandwidth on basic HID implementations is limited to 64KB/sec.

To make up for these disadvantages, TI provides a Windows HID API and demo project. It is streamlined for use with the *datapipe interface* that can be implemented by the MSP430 HID API stack, which eliminates the need to create HID reports. It works with composite HID devices, allowing multiple HID interfaces to easily be put in composite within the USB device, multiplying the bandwidth.

Used together, the MSP430 HID API stack and Windows HID API form an end-to-end solution that is more advantageous than virtual COM ports for many general-purpose applications.

For the USB support software for MSP430 MCUs, see the MSP430 USB Developers Package at <http://focus.ti.com/docs/toolsw/folders/print/msp430usbdevpack.html>.

2 Implementation

This Windows MSP430 HID API is hereafter referred to as *the Windows API*.

2.1 Overview

Figure 1 shows the position of this API within a Windows software stack and in relation to the USB HID API on the MSP430 device.

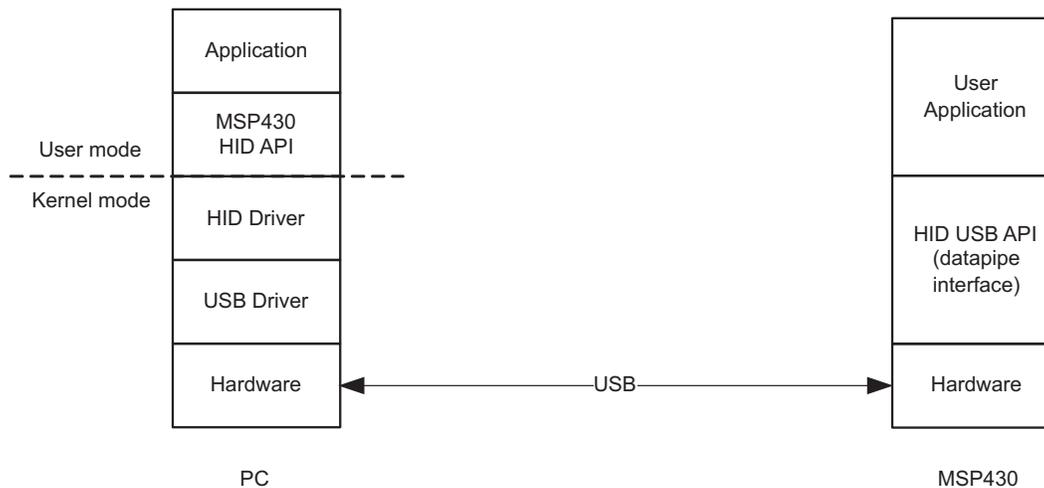


Figure 1. Windows and MSP430 Software Stacks

The software is designed to be used with the USB HID API stack firmware for the MSP430, and specifically the datapipe interface provided by the API stack. Used together, the API stack and Windows API abstract the software developer from HID reports, allowing the developer to think of the data interface as an unformatted stream of data, similar to a COM port.

To accomplish this abstraction, the datapipe interface implements a very simple HID report structure, which the developer does not need to modify. The developer uses simple commands to send or receive data blocks of any size, and the API uses the report as a data packet to transfer this data. (See the *MSP430 USB CDC/HID API Stack Programmer's Guide* for more information about the datapipe interface.)

2.2 File Organization

Table 1 shows the files in the API and demo application.

Table 1. Source Code Files

File	Description
<i>Demo Application</i>	
*.cpp	Various C++ files
<i>API</i>	
hiddevice.c	Code implementations of the API functions calls and supporting function calls.
hiddevice.h	Definitions for the API function calls that can be used by the application.

2.3 System Requirements

The Windows API, as distributed, is written for Visual Studio 2008, using the Microsoft Foundation Class (MFC) library. As such, it is usable on a wide variety of Windows platforms, including all flavors of Windows Vista/XP/7. The API is written in C, and the demo application is written in C++. The Express version of Visual Studio is sufficient for the compiling the API. However, the full version is required to compile the demo application.

The API also requires the Windows Driver Kit (WDK) to be installed. This is available from Microsoft at no cost. Version 7 is recommended, because it results in speed gains relative to previous versions.

To link the API to the WDK:

1. Download and install the latest version of WDK from Microsoft.
2. Right click on HidDevice.c file → Properties
3. Navigate to C/C++ → General → Additional Include Directories
4. Include:
 - (a) c:\<WinDDK Install>\<Build Version>\inc\api\
 - (b) c:\<WinDDK Install>\<Build Version>\inc\crt\
5. Navigate to Linker → General → Additional Include Directories
6. Include c:\<WinDDK Install>\<Build Version>\lib\wpx\i386

2.4 MSP430 USB API Stacks

The MSP430 USB API is designed for easy creation of USB devices without having to learn USB in detail. Accompanying it is the MSP430 USB Descriptor Tool, which automatically:

- Configures the API stack for any combination of USB interfaces (single or in composite)
- Creates USB descriptors that work the first time

The API for HID allows two kinds of HID implementations. One is for traditional HID devices, which require detailed knowledge of HID reports. The other is datapipe HID devices, which is TI's implementation of a method which uses HID as a simple data carrier. This avoids any need for the designer to create HID reports.

To the MSP430 application, the HID-datapipe interface looks and feels much like a COM port:

- It is relatively unformatted, allowing the designer to apply formatting of his or her choosing
- It handles data chunks of any size (not limited by USB packet sizes).

In fact, the API for HID-datapipe is almost identical to the one for CDC (which is used to produce a virtual COM port on the host).

For more information about the MSP430 USB API stacks, the download link can be found on <http://www.ti.com/430usb>.

2.5 How Windows Maps Physical USB HID Devices to the Host Application

Every USB device contains a vendor ID (VID) and a product ID (PID). USB hosts use the VID/PID combination to identify product types. It stands to reason that all products of a given PID from a vendor of a given VID are functionally identical. The host associates all devices with that VID/PID with a particular USB device class – for example, HID or CDC. Whenever it encounters this VID/PID, it loads that particular driver.

Above the Windows HID driver, things look slightly different. The HID driver maintains a list of "HID devices" on the bus, selectable by a generic index, and makes this list available to the host application above it. The "HID devices" in this list are logical entities, not physical USB devices. These logical entities don't necessarily map directly to the physical devices on the bus.

As an example, it is possible that more than one "HID device" with a given VID/PID may be reported to the host application. This could mean there are multiple physical devices of a given product type attached to this host; or it might be that there is a composite device that has multiple HID interfaces (which can be thought of as logical devices). Further, if there are 4+ "HID devices", it is possible that both these factors are involved, as in [Figure 2](#).

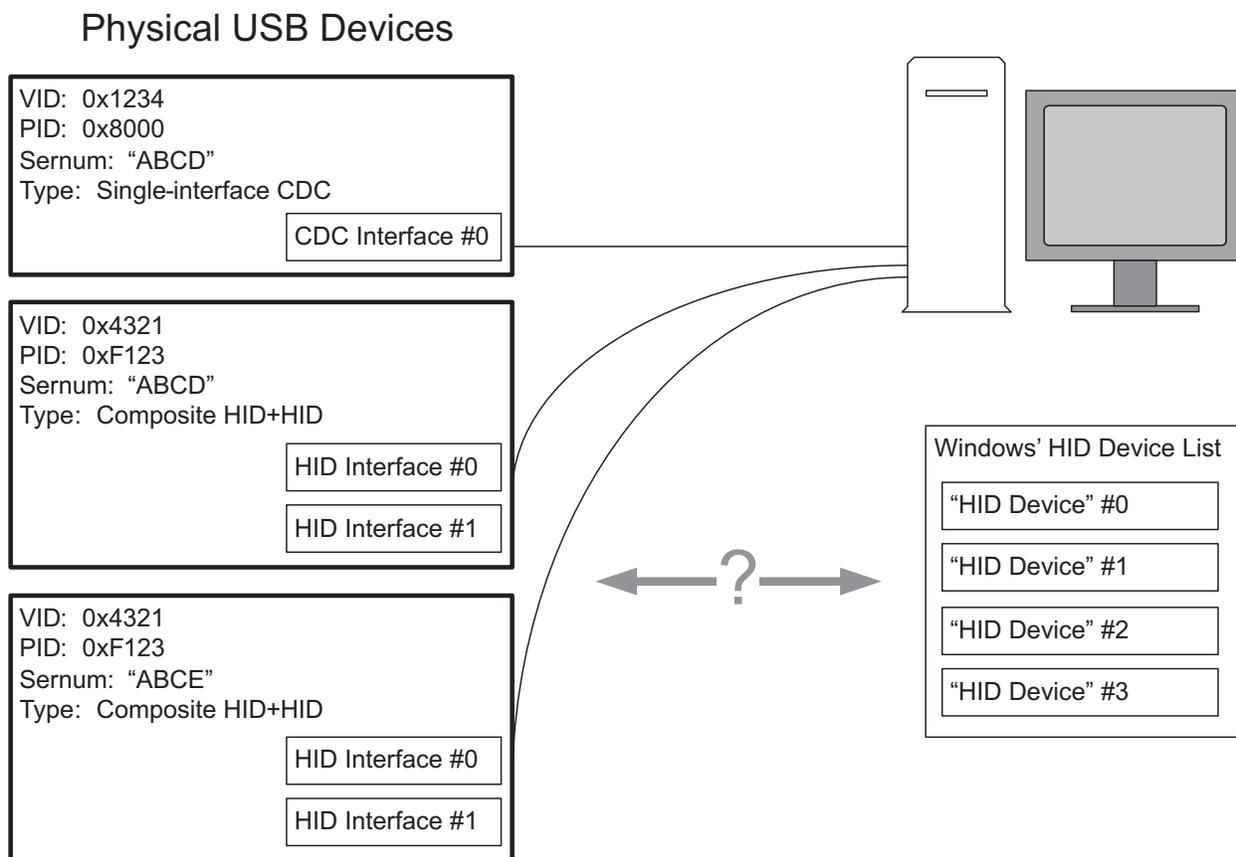


Figure 2. Mapping of "HID Interfaces" to "HID Devices" on a Windows System

In this figure, the Windows application cannot immediately know what these "HID devices" represent in the real world.

If the host application wants to be functional in this kind of multiple-device situation – or further, to be able to take advantage of multiple HID datapipes on a physical device – it needs to probe further. One important piece of information it needs to gather from each HID device is the serial number (marked as "sernum" in [Figure 2](#)). A USB device can optionally report a serial number that identifies a unique piece of physical equipment among all others containing the same VID/PID. (The MSP430 USB API stacks can automatically report a serial number, if selected to do so with the Descriptor Tool.)

Once the host application has the VID, PID, and serial number of every HID device it sees on the bus, it can begin to understand the situation on the bus. If it finds two HID devices with a given VID/PID, with the serial numbers also the same, then it can know this is a composite HID device that has two HID interfaces. If, instead, these two HID devices have different serial numbers, then the application is looking at two physical single-interface HID devices.

With these techniques, the developer can then design devices with multiple HID interfaces, using each one as a datapipe for a particular kind of information – the equivalent of multiple COM ports. Among other benefits, this helps overcome the bandwidth limitation of a single HID interface (64 kbps). As many as eight HID interfaces can be run in parallel on an MSP430, providing 512-kbps bandwidth.

It should be added that if Windows recognizes in the USB descriptors that this device is a mouse or keyboard during initial enumeration, then Windows itself becomes the "host application" for this HID device. It issues report requests and uses the data to control the mouse on the screen or input data for text entry.

2.6 Locating a Specific HID Device/Interface on the System and Opening It

Throughout the rest of this document, the following terms are used:

- *HID device*: A device in Windows HID device list
- *Physical USB device*: An actual piece of USB hardware, with its own unique address on the USB bus
- *HID interface*: An interface declared within the USB descriptors of a physical USB device. It may be the only interface on the device, or it may be one of several as part of a composite USB device. An HID interface is associated with an HID device in Windows.

As the host application developer sets about accessing the USB device, he or she generally knows the VID/PID pair of the device to which it is being associated, because both are usually designed by the same party. The process therefore begins with calling `HID_GetSerNums()`. This function receives a VID/PID as parameters and returns a list of serial numbers, each of which represents a physical USB device on the bus associated with that VID/PID. The application may choose to interact with only one of these physical devices, or it could be given capability of interacting with multiple devices.

The application developer also generally knows how many interfaces each of these physical devices has. If somehow this is not known, the function `HID_GetNumOfInterfaces()` can be called. When `HID_Open()` is called to open the device, it is necessary to know the total number of HID interfaces on the physical device.

When the VID, PID, and serial number are used to filter the HID device list, all that remains is a list that represents all of the HID interfaces in the physical USB device described by those parameters. This list is usually one or two, but can be as many as eight on an MSP430. The order of this list is the same as the order in which the HID interfaces were listed in the physical device's USB descriptors. In this way, the host application and MSP430 application using the API stack can "find" each other, forming a complete data link.

Armed with the VID/PID, serial number, and index of the desired HID interface within the USB device, the application can call `HID_Open()` to open a connection to the HID interface on the physical USB device.

When the application is finished using the device (for example when exiting the program), the application should close any open HID devices with `HID_Close()`.

2.7 Sending/Receiving Data

Once the HID interface is opened – and assuming this interface is set up as a "datapipe" interface on the MSP430 HID API – data chunks of any size can be sent/received with the API using the `HID_writeFile()` and `HID_readFile()` calls. The calls automatically handle packetization into one or more HID reports.

If implementing a "traditional" HID device rather than "datapipe" – meaning the report format has been customized and the HID-traditional function calls are being used in the MSP430 HID application – then it is still possible to use this API as an example, but it will be necessary to customize the API code.

2.8 Detecting the Dynamic Connection/Disconnection of HID Devices

Unlike with traditional COM ports, USB applications must be aware that the user may detach (or attach) a device at any time. This is the dynamic "plug-and-play" aspect of USB. Windows handles this through notifications, implemented in the MFC library. This occurs outside the API, because the notifications go directly to the application's window object; but when these notifications are received, the application can call API handlers specifically provided for this purpose. The demo application shows how this can be done.

These are the steps required for an application to handle these notifications:

1. At the beginning of the application, the application should register its window to receive device change notifications by using `HID_RegisterForDeviceNotification()`.
2. Put `ON_WM_DEVICECHANGE` into the message map for the window. (Make sure this is outside the `AFX_MSG_MAP`-commented section.)
3. Write a function `OnDeviceChange()` as a public function for the window. This message handler will be called by the MFC framework in response to an `ON_WM_DEVICECHANGE` notification.

The definition of the function should be as shown:

```
afx_msg BOOL OnDeviceChange(  
    UINT nEventType,  
    DWORD_PTR dwData  
);
```

This function needs to determine if any of the "HID devices" used by this application were removed from the system. To that end, it can call `HID_IsDeviceAffected()` for each open HID device to determine if they have been removed.

When exiting the application, the application should be sure to unregister it using `HID_UnregisterForDeviceNotification()`.

3 Function Call Reference

3.1 Device Connection Management and Initialization Calls

Table 2 summarizes the device connection management calls.

Table 2. Device Connection Management Call Summary

Function	Description
VOID HID_Init()	Initializes the HID device data structure used to store information about the HID device/interface.
DWORD HID_GetNumOfInterfaces()	Returns the number of HID devices on the system associated with a particular VID/PID and serial number.
BYTE HID_Open()	Opens a handle to a particular device.
BYTE HID_Close()	Closes the handle to a particular device.
BYTE HID_GetSerNums()	Obtains the serial numbers associated with a particular VID/PID.

3.1.1 VOID HID_Init(struct strHidDevice* pstrHidDevice)

Description

This function initializes pstrHidDevice. It must be called one time for every strHidDevice structure instance before using it.

Parameters

Table 3. Parameters for HID_Init()

strHidDevice* pstrHidDevice	The structure to be initialized.
-----------------------------	----------------------------------

3.1.2 DWORD HID_GetSerNums(WORD vid, WORD pid, struct strTrackSerialNumbers *serialNumList)

Description

Returns the number of physical USB devices on the system associated with this combination of vid and pid. (These devices may be of type HID or otherwise.) If no devices are connected, it returns zero. If devices are connected, the passed strTrackSerialNumbers structure is populated with the serial numbers corresponding to the physical devices found, and the function returns the total number of serial numbers in the list.

If the returned result is more than one, this means there are multiple physical devices with the combination of vid/pid.

Parameters

Table 4. Parameters for HID_GetSerNums()

WORD vid	16-bit vendor ID of the device to be found.
WORD pid	16-bit product ID of the device to be found.
struct strTrackSerialNumbers *serialNumList	The structure to contain the list of serial numbers associated with vid/pid. The function populates the structure with the serial numbers found.
returns	0: If no physical USB devices with this VID/PID are attached. non-0: The number of physical USB devices on the system with this VID/PID.

Table 5. strTrackSerialNumbers Structure Definition

Field	Description
DWORD deviceNum	Index number representing the physical USB device.
char serialNum[SERNUM_LEN];	String containing the serial number of the detected physical devices.

3.1.3 **DWORD HID_GetNumOfInterfaces(WORD vid, WORD pid, DWORD numSerNums)**

Description

Returns the number of HID interfaces that exist within each physical USB device identified by vid and pid. If no such devices are connected, the function returns zero.

numSerNums is the number of physical devices on the system associated with this VID/PID. This function accomplishes its goal by scanning the Windows HID device list (that is, the logical HID devices registered on the system), counting those associated with this VID/PID, and then dividing it by numSerNums. numSerNums can be found by calling HID_GetSerNums().

A single-interface HID device always causes the value 1 to be returned.

Parameters

Table 6. Parameters for HID_GetNumOfInterfaces()

UINT vid	16-bit vendor ID of the device to be found.
UINT pid	16-bit product ID of the device to be found.
DWORD numSerNums	Total number of connected physical devices with this VID/PID
returns	0: If no USB devices with this VID/PID are attached. non-0: The number of USB devices on the system with this VID/PID

3.1.4 **BYTE HID_Open(struct strHidDevice* pstrHidDevice, WORD vid, WORD pid, DWORD deviceIndex, char serialNumber[SERNUM_LEN], DWORD totalDevNum, DWORD totalSerNum)**

Description

Attempts to locate a device of index deviceIndex within the list of devices on the system that are associated with this VID/PID and serial number. If found, it opens a handle to the device and stores it within the pstrHidDevice structure, and loads the other fields within the structure as well.

To accomplish its goal, HID_Open() needs two additional parameters. One is the totalDevNum, which is the number of HID interfaces on the device described by vid and pid. Usually the developer knows this already. If not, then HID_GetNumOfInterfaces() can be called. Another required parameter is totalSerNum, which is the total number of serial numbers (physical devices) on the system matching this VID/PID.

deviceIndex is a number from 0 to totalDevNum-1. Once vid, pid, and serialNumber are used to filter the Windows HID device list, the list is effectively reduced to HID devices representing HID interfaces contained within a particular physical USB device. All that remains is to select one of those interfaces, and this is what deviceIndex does. The ordering of these HID devices (zero through totalDevNum) is the same as the order in which the HID interfaces were declared in the device's USB descriptors (see [Section 2.5](#)).

If the device described by vid and pid has only one HID interface in it, deviceIndex can simply be 0.

The function assumes that this device is in fact an HID device, and not a USB device of another type.

Parameters

Table 7. Parameters for HID_Open()

strHidDevice* pstrHidDevice	The structure to contain the newly-opened device.
WORD vid	16-bit vendor ID of the device to be opened.
WORD pid	16-bit product ID of the device to be opened.
DWORD deviceIndex	The index within the available devices of this VID/PID and serial number.
char serialNumber[SERNUM_LEN]	Char string of size SERNUM_LEN (40) of the serial number of the device to be found.
DWORD totalDevNum	The total number of HID interfaces of a the physical device described by this VID/PID and serial number.
DWORD totalSerNum	The total number of available physical USB devices for this VID/PID
returns	HID_DEVICE_SUCCESS. The device has been found and opened. HID_DEVICE_ALREADY_OPENED. The device has already been opened. HID_DEVICE_NOT_FOUND. No device identified by this VID/PID/index could be located.

3.1.5 BYTE HID_Close(struct strHidDevice* pstrHidDevice)

Description

Closes the device associated with this pstrHidDevice structure, and instructs the system to close the handle. This should always be performed by the application when it is done with the device.

Parameters

Table 8. Parameters for HID_Close()

strHidDevice* pstrHidDevice	The structure to contain the newly-opened device.
returns	HID_DEVICE_HANDLE_ERROR. The device handle is invalid. HID_DEVICE_SUCCESS. The device handle is successfully closed. HID_DEVICE_NOT_OPENED. Attempted to close a device handle that was not opened before.

3.1.6 BYTE HID_GetVersionNumber(struct strHidDevice* pstrHidDevice, USHORT * VersionNumber)

Description

Places the device release number associated with pstrHidDevice into VersionNumber. This is the value reported in the bcdDevice field of a physical USB device's device descriptor.

Parameters

Table 9. Parameters for HID_GetVersionNumber()

strHidDevice* pstrHidDevice	The structure containing the HID device information.
USHORT* VersionNumber	The device release number.
returns	HID_DEVICE_HANDLE_ERROR. The device handle is invalid. HID_DEVICE_SUCCESS

3.2 Sending/Receiving Data

The calls shown in [Table 10](#) pertain to sending/receiving data.

Table 10. Device Connection Management Call Summary

Function	Description
BYTE HID_WriteFile()	Sends data to the device.
BYTE HID_ReadFile()	Receives data from the device.

3.2.1 BYTE HID_WriteFile(struct strHidDevice* pstrHidDevice, BYTE* buffer, DWORD bufferSize, DWORD* bytesSent)

Description

Sends bufferSize bytes of data stored in buffer to the HID device designated by pstrHidDevice.

This function sends the data using HID reports as "packets". If any of the attempts to send a HID report times out, then the function returns HID_DEVICE_TRANSFER_TIMEOUT.

There is no inherent limit to the number of bytes that can be sent, other than the limitation imposed by bufferSize being a DWORD value. Packetization is handled automatically.

Parameters

Table 11. Parameters for HID_WriteFile()

strHidDevice* pstrHidDevice	The structure containing the HID device information.
BYTE* buffer	An array of data to be sent
DWORD bufferSize	Number of bytes to be sent, starting from address buffer.
DWORD bytesSent	Number of bytes actually sent (in the event of an error)
returns	HID_DEVICE_NOT_OPENED. The HID device failed to open. HID_DEVICE_TRANSFER_TIMEOUT. A report request timed out. HID_DEVICE_TRANSFER_FAILED. The transmission failed for unspecified reasons. HID_DEVICE_SUCCESS. All the data was successfully sent.

3.2.2 BYTE HID_ReadFile(struct strHidDevice* pStrHidDevice, BYTE* buffer, DWORD bufferSize, DWORD* bytesReturned)

Description

Obtains bufferSize bytes of payload data from the HID device designated by pStrHidDevice, and stores it in buffer. It does this by reading as many HID reports as necessary until bufferSize bytes have been returned.

If the function returns HID_DEVICE_SUCCESS, then bytesReturned will equal bufferSize. bytesReturned will only be less than bufferSize in the event of one of the other return codes.

This function receives the data by using HID reports as "packets". If any attempt to fetch a packet from the USB device times out, according to the value pStrHidDevice.uGetReportTimeout, then the function returns HID_DEVICE_TRANSFER_TIMEOUT.

There is no inherent limit to the number of bytes that can be received, other than the limitation imposed by bufferSize being a DWORD value. Packetization is handled automatically.

Parameters

Table 12. Parameters for HID_ReadFile()

strHidDevice* pstrHidDevice	The structure containing the HID device information.
BYTE* buffer	An array to contain the received data
DWORD bufferSize	Size of buffer, indicating the maximum amount of bytes intended to be received
DWORD* bytesReturned	The number of bytes actually received (in the event of an error)
returns	HID_DEVICE_NOT_OPENED. The HID device failed to open. HID_DEVICE_TRANSFER_TIMEOUT. A report request timed out. HID_DEVICE_TRANSFER_FAILED. The transmission failed for unspecified reasons. HID_DEVICE_SUCCESS. The correct number of bytes were received.

3.3 Plug and Play Management

The calls shown in [Table 13](#) help the application manage the dynamic attachment/detachment of USB devices.

Table 13. Device Connection Management Call Summary

Function	Description
BYTE HID_RegisterForDeviceNotification()	Sends data to the device.
BYTE HID_UnregisterForDeviceNotification()	Receives data from the device.
BYTE HID_IsDeviceAffected()	Determines whether a particular device is still on the system.

3.3.1 BYTE HID_RegisterForDeviceNotification(HWND hWnd, HDEVNOTIFY* diNotifyHandle)

Description

Registers the window pointed to by handle hWnd to receive notifications when devices are added to or removed from the system. Steps should be taken to equip the application to respond to system notifications, as described in [Section 2.6](#).

A device notification handle is returned in diNotifyHandle. This should be stored by the application, to be used when calling HID_UnregisterForDeviceNotification().

Parameters

Table 14. Parameters for HID_RegisterForDeviceNotification()

HWND hWnd	Handle to the main window
HDEVNOTIFY* diNotifyHandle	Device notification handle.
Returns	HID_DEVICE_HANDLE_ERROR. The device handle is invalid. HID_DEVICE_SUCCESS

3.3.2 BYTE HID_UnregisterForDeviceNotification(HDEVNOTIFY* diNotifyHandle)

Description

Unregisters the window pointed to by the device notification handle diNotifyHandle, which was returned by HID_RegisterForDeviceNotification. After this call, Windows will no longer notify the application when devices are added to or removed from the system. The function returns true whether the handle is valid or not.

Parameters

Table 15. Parameters for HID_UnregisterForDeviceNotification()

HDEVNOTIFY* diNotifyHandle	Device notification handle.
Returns	HID_DEVICE_SUCCESS

3.3.3 BOOL IsDeviceAffected(struct strHidDevice* pstrHidDevice)

Description

The application should call this function in response to an ON_WM_DEVICECHANGE notification received from the system. It indicates whether the event refers to device associated with pstrHidDevice.

Parameters

Table 16. Parameters for HID_IsThisDeviceAffected()

strHidDevice* pstrHidDevice	The structure containing the HID device information.
Returns	TRUE or FALSE

4 Demo Application

A simple demo application is provided which shows how to use the API. The application can be thought of as similar to a COM port terminal application, like Hyperterminal. It sends and receives chunks of data in a similar fashion, except it does over an HID interface.

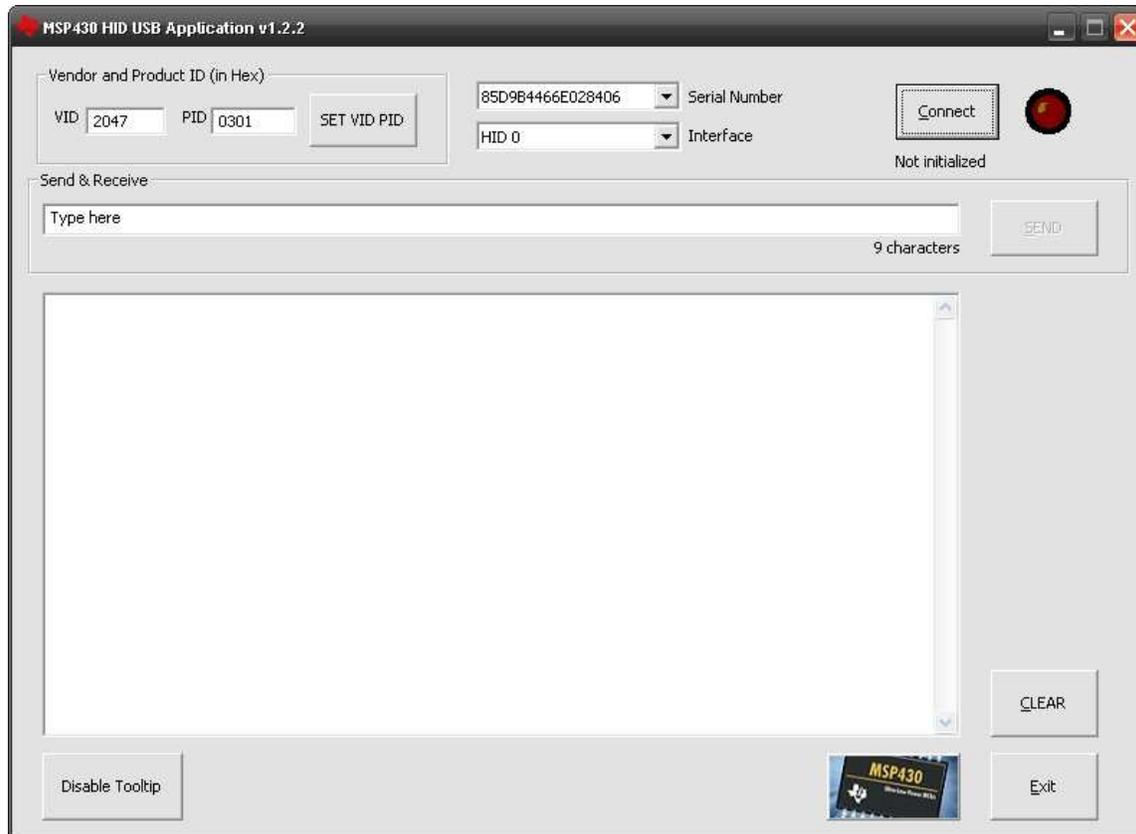


Figure 3. Demo Application Window

To operate the program, attach an MSP430 running the HID API stack, using the datapipe function calls. Several examples of this are provided with the API stack distribution. There are examples of single-interface HID, a CDC+HID composite device, and an HID+HID composite device. All of these can be used with this demo application.

The VID and PID fields in the GUI select the VID/PID for which the application should search. After entering them, press the "Set VID/PID" button.

When the VID/PID has been set, the application determines how many physical devices are attached to the system matching this VID/PID. It finds their serial numbers and posts them in the "Serial Number" combo box. It automatically selects the first serial number in the list, and proceeds to determine the number of HID interfaces present on that device. It loads the "Interface" combo box with a string for each one of these interfaces: "HID 0", "HID 1", etc. If there are any HID interfaces found, then "HID 0" is selected by default.

Press the "Connect" button. This opens a data connection with the USB device. If successful, the string below the button indicates it as such. If it is not successful, try the following:

- Check the Windows Device Manager to ensure that the device successfully enumerated on the system as an HID device
- Ensure the device is equipped with the MSP430 HID API stack, running an HID-datapipe application
- Ensure that the VID/PID shown in descriptors.h is the same pair entered into the demo app's fields, and that the "Set VID/PID" button was pressed.

Once the connection is initialized, data can be sent to the device by entering text and pressing "Send". Data received by the application from the device at any time is displayed in the large text field. The receive window can be cleared with the "Clear" button.

If the HID device is removed from the bus while another device is being accessed, or if another HID device of this VID/PID is added, the application automatically updates the pulldown menu list.

5 MSP430 USB Tool Suite

This API is part of a complete suite of tools TI provides to make USB easier on the MSP430, including:

- MSP430 USB API Stacks
 - CDC (Communications Device Class)
 - HID (Human Interface Device class)
 - MSC (Mass Storage Class)
- MSP430 USB Descriptor Tool
 - Automatically configures the USB API stacks for any combination of USB interfaces (single or composite USB device), with USB descriptors that work the first time
- MSP430 USB Field Firmware Update Starter Project
 - A Windows Visual Studio Express project for updating firmware in the field on the MSP430 over USB from a PC

6 References

1. *MSP430 USB API Stack Programmer's Guide* (accompanies the API stack download; see the link to the MSP430 USB Developer's Package, located at <http://www.ti.com/430usb>)
2. Microsoft Developers Network reference for HID:
<http://msdn.microsoft.com/en-us/library/dd446410.aspx>
3. If support is needed, go to <http://www.ti.com/msp430> and see the support options.

Appendix A HID Interface Data Structure: strHidDevice

The API uses a data structure to contain all information relating to the HID device, defined in hiddevice.h (see [Table 17](#)). An instance of the structure is passed into any API call that involves the device.

Table 17. strHidDevice Structure Definition

Field	Description	Data Origin
HANDLE hndHidDevice	Handle to this HID device.	Written by HID_Open(). Set to INVALID_HANDLE_VALUE during HID_Close().
BOOL bDeviceOpen	Boolean indicating that the device is opened, and that the handle in hndHidDevice is valid.	Written by HID_Open() and HID_Close().
UINT uGetReportTimeout	Timeout value for reading data, in milliseconds. This is the amount of time HID_ReadFile() will wait for any given report from the device within a data read operation.	Must be initialized by the application. The demo application does this in UsbAppDlg.cpp.
UINT uSetReportTimeout	Timeout value for writing data, in milliseconds. This is the amount of time HID_WriteFile() will wait for a device to receive any given report within a data send operation.	Must be initialized by the application. The demo application does this in UsbAppDlg.cpp.
OVERLAPPED oRead	Asynchronous I/O structure	Initialized by HID_Init()
OVERLAPPED oWrite	Asynchronous I/O structure	Initialized by HID_Init()
WORD wInReportBufferLength	Maximum length of an input report.	Initialized by HID_Init(), derived from the device's USB descriptors.
WORD wOutReportBufferLength	Maximum length of an output report.	Initialized by HID_Init(), derived from the device's USB descriptors.
BYTE inBuffer[256]	Buffer to receive incoming data.	Written by HID_readFile(), read by the application.
WORD inBufferUsed	The number of bytes in inBuffer[].	Written by HID_readFile(), read by the application

Appendix B Format of Reports on HID-Datapipe Devices

USB devices running the MSP430 USB API stacks and creating a HID-datapipe interface use a HID report formatted as shown in [Table 18](#). This is defined in the USB descriptors located in the physical USB device. This Windows API uses the same format, and in fact assumes the HID device uses this format.

Table 18. HID Datapipe Report Structure

Field	Size
Report ID	1 byte (0x3F)
Size	1 byte
Data	N-2 bytes, where N is the payload size of the USB packets (that is, the MAX_PACKET_SIZE value defined in the descriptors.h file in the USB API stacks).

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Transportation and Automotive	www.ti.com/automotive
Video and Imaging	www.ti.com/video
Wireless	www.ti.com/wireless-apps

TI E2E Community Home Page

e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2011, Texas Instruments Incorporated