

**USB DLL**  
**Programmable Products Inc**  
Version 2.1  
May 12, 2003

## Programmable Products USB DLL

Overview .....	3
DLL Interface .....	4
Open Device .....	4
Open Endpoints .....	4
Perform Access .....	5
Close Device .....	6
Files .....	6
PpiAsyncRead .....	7
PpiAsyncWrite .....	7
PpiCancelDevice .....	8
PpiCloseDevice .....	8
PpiCloseEndPoint .....	8
PpiGetStatus .....	9
PpiGetEndpointDescription .....	10
PpiNumberOfEndpoints .....	10
PpiOpenDevice .....	11
PpiOpenEndpoint .....	11
PpiSyncRead .....	12
PpiSyncWrite .....	13

## Overview

The Programmable Products (PPI) USB DLL is a general purpose USB device driver with a Windows DLL wrapper. The purpose of the wrapper is to provide access to the driver through several simple function calls, which can be done through a wide variety of programming languages.

The DLL provides for multi-threaded access to the USB device, in which each USB Endpoint (a logical connection) is handled in it's own thread. This approach allows the application to make calls to the hardware without having the function block the rest of the program flow. Through the Status function, commands can be monitored and cancelled in the event of an error.

The basic architecture of the DLL can be grouped into two categories: Device, and Endpoint. Device functions are used to connect to the device, while Endpoint functions are used to read or write to the device. An endpoint is either an input, (USB device to PC), or and output (PC to USB device). Each endpoint is numbered from 0 to X, and corresponds to the order in which they are reported in the USB descriptor, which is sent from the device during plug and play.

## **DLL Interface**

The PpiUsb.dll will automatically get copied to the windows/system subdirectory, in which any program can link into. For VC++, an include file (PpiUsbDll.h) can be included. This file provides the required function definitions when accessing the DLL. There is also a function, PpiDllInit(), which should be called at initialization of the application. This function maps the function pointers to the actual functions inside the DLL. If this function fails, typically the problem is the DLL is not located in the Windows/system subdirectory.

To access a device, the following steps must be performed:

1. Open Device
2. Open Endpoints
3. Perform Accesses
4. Close device

### ***Open Device***

In order to open a device, two things must be known: the device logical name, and the instance of the device. The logical name is set in the supplied INF, while the instance is a number representing which instance of the device is to be opened. For example if there is only one device, the instance is 0. If there are two devices, and the application needs to access the second device, a 1 is passed. An example of this function is as follows:

```
if (!PpiOpenDevice("PPI_USB", 0))
{
    MessageBox("Could not Open Device",MB_OK);
    Return FALSE;
}
```

### ***Open Endpoints***

Once the device is opened, the application must then open the endpoints used for communication. If the endpoint number is not known, the PpiNumberOfEndpoints will return the number of endpoints. The PpiGetEndpointDescription can be called to determine the functionality of each endpoint. However, if the endpoint index is known, the application can directly open each desired endpoint. The endpoint index is the order in which the device reports the endpoints in the descriptor. Since the direction of each endpoint is fixed by the device, only the index is needed, not the direction. An example of opening two endpoints is as follows:

```
if (!PpiOpenEndpoint(0))
{
    MessageBox("Could not Open Input Endpoint",MB_OK);
    Return FALSE;
}
```

```

if (!PpiOpenEndpoint(1))
{
    MessageBox("Could not Open Output Endpoint",MB_OK);
    Return FALSE;
}

```

### **Perform Access**

At this point, the application can now perform access to the device. The USB device packet size does not need to be handled at this layer. For example if the device is using USB 1.1, which has a maximum packet size of 64 bytes, and the application needs to transfer 128 bytes. The application only needs to call the transfer function with the number of bytes required to transfer. The DLL and driver will packetize the transfer into two 64-byte transfers.

There are two ways to perform transfers: Synchronous, or Asynchronous. The Synchronous method will block the application calling the transfer function until either the transfer has completed, or the timeout has occurred. If the timeout is set to 0xFFFF, the transfer will halt the application until the transfer has completed, no matter how long the transfer takes. This is often undesirable, since there is no way to cancel the transfer if something went wrong. Only one transfer is allowed to be outstanding at a time. An example of a synchronous write transfer is as follows:

```

UCHAR    ucWrArray;
UINT     uiWrPtr;

uiWrPtr =0;
ucWrArray[uiWrPtr++] = 2;        // Dest
ucWrArray[uiWrPtr++] = 62;      // Length
for (ucLoop = 0; ucLoop < 62; ucLoop++)
    ucWrArray[uiWrPtr++] = ucLoop; // Payload

if (!PpiSyncWrite(1, 64, &ucWrArray[0],0x0100))
{
    MessageBox ("Write Failed",MB_OK);
    Return FALSE;
}

```

Asynchronous transfer will immediately return once the transfer has been submitted. It is very important that the endpoint status is checked to determine if the endpoint is available for the transfer. Once the transfer has been submitted, the application can then check the status of the endpoint to determine if the transfer has completed. Once the transfer has completed, another transfer can be sent. An example of an Asynchronous transfer is as follows:

## Programmable Products USB DLL

```

if (!PpiAsyncRead(0, 128, &ucRdArray[0]))
{
    MessageBox("Read Failed",MB_OK);
    Return FALSE;
}

do
{
    PpiGetStatus(0,&ucStatusArray[0]);
} while ((ucStatusArray[THREAD_OFFSET_OPCODE] !=
THREAD_CMD_DONE) &&
(ucStatusArray[THREAD_OFFSET_OPCODE] !=
THREAD_CMD_FAILED));

if (ucStatusArray[THREAD_OFFSET_OPCODE] ==
THREAD_CMD_FAILED)
    return FALSE;

```

### ***Close Device***

After all transfer has been completed, the driver should be closed before exiting the application. An example of the close function is as follows:

```
PpiCloseDevice();
```

### **Files**

<b>Filename</b>	<b>Location</b>	<b>Description</b>
PPILdr.INF	C:\WINNT\INF	INF file for unconfigured EZUSB
PPILdr.SYS	C:\WINNT\SYSTEM32	Driver to load unconfigured EZUSB device
PPIUSB.DLL	C:\WINNT\SYSTEM	DLL to access device
PPIUSBDLL.H	Project subdirectory	Header file which must be included to access DLL from VC++
PPIUSB.H	Not used	Function prototypes for DLL functions. Only used as a reference.
WINRTUSB.SYS	C:\WINNT\SYSTEM32	Driver for device
WINRTUSB.DLL	C:\WINNT\SYSTEM	Another DLL required for the USB driver
PPI_USB.INF	C:\WINNT\INF	INF file for configured device

Function Descriptions

***PpiAsyncRead***

**Description**

This function will perform an asynchronous read. Once the number of bytes requested has been received, the endpoint status will change from THREAD\_CMD\_START to THREAD\_CMD\_DONE.

**Return BOOL**

TRUE: Transfer has been submitted

FALSE: Transfer has not been submitted. Endpoint may be busy, or not opened.

**Input**

**UCHAR ucEpNum**

Endpoint Number (0 to X)

**ULONG ulLength**

Number of bytes to be read (0 to 0xFFFF)

**UCHAR\* pBuffer**

Pointer to unsigned character buffer in which the read data will be stored.

***PpiAsyncWrite***

**Description**

This function will perform an asynchronous write. Once the requested number of bytes has been sent, the endpoint status will change from THREAD\_CMD\_START to THREAD\_CMD\_DONE.

**Return BOOL**

TRUE: Transfer has been submitted

FALSE: Transfer has not been submitted. Endpoint may be busy, or not opened.

**Input**

**UCHAR ucEpNum**

Endpoint number (0 to X)

**ULONG ulLength**

Number of bytes to write (0 to 0xFFFF)

**UCHAR\* pBuffer**

Pointer to unsigned character buffer that contains the data to be sent.

## ***PpiCancelDevice***

### **Description**

This function will cancel all transfers in progress to the device. This function may be used when transfers cannot be completed.

### **Return BOOL**

TRUE: Transfers stopped

FALSE: Could not stop transfers. Device may be unplugged.

### **No Inputs**

## ***PpiCloseDevice***

### **Description**

This function must be called before exiting the application. The function will first close all endpoints, and then close the device.

### **Return BOOL**

TRUE: Device closed with no errors.

FALSE: Device could not be closed.

### **No Inputs**

## ***PpiCloseEndPoint***

### **Description**

This function will close a single endpoint.

### **Return BOOL**

TRUE: Endpoint was closed

FALSE: Endpoint could not be closed. May need to call PpiCancelDevice.

### **Input**

**UCHAR ucEpNum**

Endpoint number (0 to X)

## ***PpiGetStatus***

### **Description**

This function will return the status of the endpoint. The buffer returned contains the status of the device, thread, and transfer.

### **Return BOOL**

TRUE: Status returned without error  
FALSE: Status could not be returned

### **Input**

#### **UCHAR ucEpNum**

Endpoint number (0 to X)

#### **UCHAR\* pBuffer**

Unsigned character buffer of at least 7 bytes used to return status of device and endpoint.

<b>Byte</b>	<b>Description</b>
0	<b>Endpoint Number</b>
1	<b>Thread Status</b> <ol style="list-style-type: none"> <li>1. THREAD_REQ_RUN                             <ol style="list-style-type: none"> <li>a. Endpoint has been requested to start, but has not yet.</li> </ol> </li> <li>2. THREAD_RSP_RUN                             <ol style="list-style-type: none"> <li>a. Thread is running normally</li> </ol> </li> <li>3. THREAD_REQ_STOP                             <ol style="list-style-type: none"> <li>a. Thread has been request to stop, but has not stopped yet.</li> </ol> </li> <li>4. THREAD_RSP_STOP                             <ol style="list-style-type: none"> <li>a. Thread has been stopped</li> </ol> </li> </ol>
2	<b>OpCode</b> <ol style="list-style-type: none"> <li>1. THREAD_CMD_WRITE                             <ol style="list-style-type: none"> <li>a. Write request</li> </ol> </li> <li>2. THREAD_CMD_READ                             <ol style="list-style-type: none"> <li>a. Read request</li> </ol> </li> <li>3. THREAD_CMD_STATUS                             <ol style="list-style-type: none"> <li>a. Status request</li> </ol> </li> <li>4. THREAD_CMD_DONE                             <ol style="list-style-type: none"> <li>a. Endpoint ready for another command</li> </ol> </li> <li>5. THREAD_CMD_BUSY                             <ol style="list-style-type: none"> <li>a. Endpoint busy</li> </ol> </li> <li>6. THREAD_CMD_START                             <ol style="list-style-type: none"> <li>a. Command has been started, endpoint busy</li> </ol> </li> <li>7. THREAD_CMD_FAILED                             <ol style="list-style-type: none"> <li>a. Command Failed. New command can be sent.</li> </ol> </li> </ol>
3 to 6	<b>Not applicable</b>

## ***PpiGetEndpointDescription***

### **Description**

This function returns the information about a given endpoint, such as the direction, transfer method, and max physical packet size.

### **Return BOOL**

TRUE: Description valid

FALSE: Description could not be returned

### **Input**

**UCHAR ucEpNum**

Endpoint number (0 to X)

**PPI\_EP\_DESCRIPTION \*EpDesc**

Point to endpoint description table

**BOOL bDir**

TRUE = IN

FALSE = OUT

**UCHAR ucEpType**

0 = Unknown

1 = BULK

**UINT uiMaxSize**

64 for USB 1.1

512 for USB 2.0

## ***PpiNumberOfEndpoints***

### **Description**

Determines the number of endpoints used in the device.

### **Returns BOOL**

True: Number of endpoints could be determined

False: Could not determine number of endpoints

### **Input**

**UCHAR \*ucNumEp**

Unsigned character in which the function returns the number of endpoints.

## ***PpiOpenDevice***

### **Description**

This function will open the device based on the name passed, and the instance of the device. By calling this function multiple times, the number of device instances can be determined. However, only one instance can be handled per instance of the DLL. Multiple DLL instance could be used to talk to multiple instances of the device.

### **Return BOOL**

True: Device opened  
False: Device could not be opened.

### **Inputs**

**LPCTSTR pLinkName**

Symbolic name of the device from the INF file.

**ULONG ulDeviceNumber**

Instance number of the device (0 to X)

## ***PpiOpenEndpoint***

### **Description**

This function will open a single endpoint for data transfer. The endpoint is either an input or an output.

### **Returns BOOL**

TRUE: Endpoint was opened and is ready for data transfer  
FALSE: Endpoint could not be opened

### **Input**

**UCHAR ucEpNum**

Number of Endpoint (0 to X)

## ***PpiSyncRead***

### **Description**

This function will transfer 0 to 0xFFFF bytes to the endpoint. The transfer is done synchronously, and will attempt to complete the transfer before the timeout has occurred. If a timeout has occurred the transfer can be cancelled with the PpiCancelDevice function. A timeout of 0xFFFF will prevent anytime from occurring, and will block until the transfer has completed. Only one transfer is allowed at a given time, and will return a FALSE if a second transfer is attempted before the endpoint is back to the THREAD\_CMD\_DONE status.

### **Returns BOOL**

TRUE: Transfer completed  
FALSE: Transfer could not be completed.

### **Input**

**UCHAR ucEpNum**

Endpoint number (0 to X)

**ULONG ulLength**

Number of bytes to transfer

**UCHAR \*pBuffer**

Buffer containing the data to be transferred.

**UINT uiTimeOut**

Timeout period for the transfer.

## ***PpiSyncWrite***

### **Description**

This function will transfer 0 to 0xFFFF bytes to the endpoint. The transfer is done synchronously, and will attempt to complete the transfer before the timeout has occurred. If a timeout has occurred the transfer can be cancelled with the PpiCancelDevice function. A timeout of 0xFFFF will prevent anytime from occurring, and will block until the transfer has completed. Only one transfer is allowed at a given time, and will return a FALSE if a second transfer is attempted before the endpoint is back to the THREAD\_CMD\_DONE status.

### **Returns BOOL**

TRUE: Transfer completed  
FALSE: Transfer could not be completed.

### **Input**

**UCHAR ucEpNum**

Endpoint number (0 to X)

**ULONG ulLength**

Number of bytes to transfer

**UCHAR \*pBuffer**

Buffer containing the data to be transferred.

**UINT uiTimeOut**

Timeout period for the transfer.