C# TE_USB_FX2 API

reference manual

General Index

1 Introduction

This document describes the API supported by standard Trenz Electronic FPGA modules equipped with Cypress EZ-USB FX2 microcontroller (currently: TE0300, TE0320 and TE0630).

This document describes two different sets of API:

1. TE_USB_FX2_CyUSB.dll
2. API commands

C# applications use directly TE_USB_FX2_CyUSB.dll based on CyUSB.dll. To avoid copying back and forth large amount of data between these two DLLs, data is passed by reference and not by value.

Hardware / Firmware	**Software**
FPGA-MicroBlaze (response to some API commands) Defined in MB_Commands	Sample application (C#) (the programmer shall know how to use API commands)
FPGA to USB_FX2 communication	TE_USB_FX2_CyUSB.dll (C#) (API commands are inserted here in the commands data array of byte)
	CyUSB.dll (C#) (Cypress .NET DLL)
USB_FX2 Firmware (able to execute API commands and send binary code responses) Defined in FX2_Commands	TE_USB_FX2_xx.sys (derivative of CyUSB.sys: Cypress EZ-USB FX2 driver derivate)
EndPoint USB FX2 Buffer (API commands are a set of byte in the buffer)	Driver Buffer (size determined by BufferSize parameter) (API commands are a set of byte in the buffer)
USB Cable and Tx/Rx Circuits	

Hardware, firmware and software stack.

1.1 API Functions (First API Set)

The first set of API is a set of DLL functions mainly used to communicate between the host computer and the EZ-USB FX2 microcontroller endpoints; this API uses the Cypress .NET CyUSB.dll as a basis. In fact, one API function (namely TE_USB_FX2_SendCommand()) is able to communicate with the MicroBlaze implemented on the FPGA.

These API functions have some parameters to set: Timeout , BufferSize and others.

1.1.1 Synchronous Functions

These functions use a synchronous version for the data transfer (Cypress XferData()). They perform synchronous (i.e. blocking) I/O operations and do not return until the transaction completes or the endpoint TimeOut has elapsed. A synchronous operation (aka blocking operation) is an operation that owns (in an exclusive way) resources and CPU until its job is done. A synchronous function monopolizes resources until its end, even during idle time.

If the program uses the synchronous XferData() function (both in C# with TE_USB_FX2_CyUSB.dll and C++ with TE_USB_FX2_CyAPI.dll), the array of data to transfer is the only one that is subdivided into packets (with packet length ≤ MaxPktSize = 512 byte) and scheduled over the USB buffer for data transmission. Until the data array is completely transferred, no other data array can be scheduled into packets over the USB, even if there is free packet time to be used by other data. The array of data passed to the XferData() function is the only owner of the USB bus until all data of this array are transferred (successfully or unsuccessfully). While using XferData() method, the OS will schedule the next XferData() only after the previous XferData() completes, leading to delay.

XferData() just calls asynchronous functions BeginDataXfer(), WaitForXfer() and FinishDataXfer() in sequence and does error handling accordingly. WaitForXfer() is the one which implements the timeout period for larger transfers. Cypress recommends the following: "You will usually want to use the synchronous XferData method rather than the asynchronous BeginDataXfer / WaitForXfer / FinishDataXfer approach.".

The API uses the synchronous version because it is more suitable to be included in a DLL and it is already fast. With synchronous version, the API functions are simpler to use.

1.1.2 Timeout Setting

Timeout is the time that is allowed to the function for sending/receiving the data packet passed to the function; this timeout shall be large enough to allow the data/command transmission/reception. Otherwise the transmission/reception will fail.

TimeOut shall be set according to the following formula:

$$\text{TimeOut (ms)} = [\text{DataLength} / \text{DataThroughput}] + 1\ \text{ms}.$$

Note: TimeOut is integer so you shall round up the result.

For write transactions, assume DataThroughput ≈ 20 Mbyte/s (it is lower than actual value, give some margin).

For read transactions, assume DataThroughput ≈ 30 Mbyte/s (it is lower than actual value, give some margin).

For SendCommand() assume DataThroughput ≈ 1 Mbyte/s (close to actual value).

These values have been verified for a Core i7 processor at 2.20 GHz with Microsoft Windows 7. Other configuration may require others value. An AMD Athlon II at 1.30 GHz with Microsoft Windows 7 might require much (e.g. two or three times) larger values. If your host computer is not highly responsive, you should set TimeOut to even larger values : e.g 20, 50, 200, 1000 ms (the less responsive the host computer is, the higher the recommended values shall be).

1.1.3 BufferSize (also called XferSize)

BufferSize is the size of the buffer used in data/command transmission/reception of a ***single endpoint;*** the total buffer size is the sum of BufferSize of every endpoint used. See section 6 TE_USB_FX2_CyUSB.dll: Data Transfer Throughput Optimization for some insights into this kind of influence

BufferSize has a strong influence on DataThroughput. If BufferSize is too small, the DataThroughput can be 1/3 to 1/2 of the maximum value (36 Mbyte/s for read and 25 Mbyte/s for write transactions). If the BufferSize has a large value (a roomy buffer), the application should be able to cope with the non-deterministic behavior of C# without losing packets.

1.1.4 PacketSize

PacketSize is the size of packets used in data/command transmission/reception of a ***single endpoint***. See section 6 TE_USB_FX2_CyUSB.dll: Data Transfer Throughput Optimization for further insights on this influence.

PacketSize has also a strong influence on DataThroughput. If PacketSize is too small (512 byte for example) you can achieve very low data throughput (2.2 Mbyte/s) even if you use a large BufferSize (driver buffer size = 131,072 byte).

1.2 MicroBlaze API Commands (Second API Set)

The second set of API is API commands. They are binary data that are sent/received by the EZ-USB FX2 microcontroller. API commands provide an easy way to create a communication interface with Trenz Electronic FPGA modules.

API commands are sent using a function of the first API set: TE_USB_FX2_SendCommand(). This function is able to pass the API commands (of the second API set) to the MicroBlaze embedded processor and receive the response binary code of using endpoint EP1.

A combination of TE_USB_FX2_SendCommand() and TE_USB_FX2_GetData() functions is able to read data from FPGA RAM.

A combination of TE_USB_FX2_SendCommand() and TE_USB_FX2_SetData() functions is able to write data to FPGA RAM.

2 Requirements

When using TE_USB_FX2_CyUSB.dll API, a host computer should meet the following requirements:

- Operating system: Microsoft Windows 2000, Microsoft Windows XP, Microsoft Windows Vista, Microsoft Windows 7
- USB driver: Trenz Electronic USB FX2 driver
- Interface: USB 2.0 host
- .NET Framework version $\geq$ 4.0.30319

See your module user manual for dedicated driver installation instructions.

3 API Functions

In order to provide a user interface for driver functions, dynamic link library TE_USB_FX2_CyUSB.dll and CyUSB.dll have been used. User program should load these libraries and initialize module connection to get access to API functions. To do this, you shall:

1. copy TE_USB_FX2_CyUSB.dll and CyUSB.dll to the project folder (for example TE-USB-Suite/TE_USB_FX2_SampleApplication/TE_USB_FX2_SampleApplication/);
2. open the C# project (double click the TE_USB_FX2_CyUSB_SampleApplication icon in the folder TE-USB-Suite/TE_USB_FX2_CyUSB_SampleApplication/);
3. open "Explore Solution" if it is not already open (Ctrl+W or left click "Visualize > Explore Solution");
4. in the right panel "Explore Solution", right click "Reference";
5. select "Add Reference". A new window (Add Reference) opens;
6. select the fourth sheet (Browse). The term "Look In" shall have automatically selected the correct folder (TE-USB-Suite/TE_USB_FX2_SampleApplication/). If is not so, you shall select the folder where you have copied the previous DLLs;
7. left click one of the two DLLs;
8. select OK;
9. repeat steps from 4 to 8 for the second DLL.

Exported function list:

- TE_USB_FX2_ScanCards()
- TE_USB_FX2_Open()
- TE_USB_FX2_Close()
- TE_USB_FX2_SendCommand()
- TE_USB_FX2_GetData()
- TE_USB_FX2_SetData()

3.1 TE_USB_FX2_ScanCards()

3.1.1 Declaration

```
public static int TE_USB_FX2_ScanCards(
  ref USBDeviceList USBdevList)
```

3.1.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_ScanCards(ref USBdevList);

3.1.3 Description

This function takes (a null initialized or an already initialized) USB device list, (re-)creates a USB device list, searches for Trenz Electronic USB FX2 devices (Cypress driver derivative and VID = 0xbd0, PID=0x0300) devices and counts them.

This function returns the number of Trenz Electronic USB FX2 devices attached to the USB bus of the host computer.

3.1.4 Parameters

1. ref USBDeviceList USBdevList

   USBDeviceList is a type defined in CyUSB.dll.

   USBdevList is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2.sys). This parameter is passed by reference (ref). See page 139-140 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference).

3.1.5 Return Value

1. int : integer type.

   This function returns the number of USB devices attached to the host computer USB bus.

3.2 *TE_USB_FX2_Open()*

3.2.1 Declaration

```
public static bool TE_USB_FX2_Open(
  ref CyUSBDevice TE_USB_FX2_USBDevice, ref USBDeviceList
USBdevList, int CardNumber)
```

3.2.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_Open
( ref TE_USB_FX2_USBDevice, ref USBdevList, CardNumber);
```

3.2.3 Description

This function takes (a null initialized or an already initialized) USB device list, (re-)creates a USB device list , searches for Trenz Electronic USB FX2 devices (Cypress driver derivative and VID = 0xbd0, PID=0x0300) and counts them.
If no device is attached, TE_USB_FX2_USB_device (CyUSBDevice type) is initialized to null.
If one or more devices are attached and

- if 0 ≤ CardNumber ≤ (number of attached devices – 1), then TE_USB_FX2_USBDevice (CyUSBDevice type) will point to and will be initialized according to the selected device.
- if CardNumber ≥ number of attached devices, then TE_USB_FX2_USBDevice (CyUSBDevice type) is initialized to null.

A more intuitive name for this function would have been TE_USB_FX2_SelectCard().

3.2.4 Parameters

1. ref CyUSBDevice TE_USB_FX2_USBDevice

   TE_USB_FX2_USBDevice is the module selected by this function. This is the most useful value returned by this function. This parameter is passed by reference (ref). See pages 70-93 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference).

2. ref USBDeviceList USBdevList

   USBDeviceList is a type defined in CyUSB.dll. USBdevList is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2.sys). This parameter is passed by reference (ref). See page 139-140 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference)

3. int CardNumber

   This is the number of the selected Trenz Electronic USB FX2 device.

3.2.5 Return Value

1. bool : logical type

   This function returns true if it is able to find the module selected by CardNumber. If unable to do so, it returns false.

3.3 TE_USB_FX2_Close()

3.3.1 Declaration

```
public static bool TE_USB_FX2_Close(ref USBDeviceList USBdevList)
```

3.3.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_Close
( ref USBdevList);
```

3.3.3 Description

This function takes an already initialized USB device list and disposes it.

Due to the fact that we are coding C# here, the device list can or cannot be erased; this is in the scope of the garbage collector and it cannot be forced by the user. Sometimes it is erased instantly, some other times it is never erased, until the user closes the application program that uses this data.

Use of TE_USB_FX2_Close() function is NOT recommended for new software projects. Users may use this function only just before exiting their applications. If users use this function anywhere else, they shall

- manage System.ObjectDisposedException exceptions (try – catch) or
- avoid using disposed resources.

Note: USBdevList is disposed, not set to null.

```
try
{
  Application Code
}
catch (System.ObjectDisposedException)
{
  Console.WriteLine("TE_USB_FX2_USBDevice disposed: you have used
the wrong procedure!");
}
```

If you want to close the current USB device (card) without opening another one, you shall use TE_USB_FX2_Open() with a device number (CardNumber) that certainly does not exist (e.g. CardNumber = 200, because there can be a maximum of 127 USB devices connected to a single host controller). The reason of this behavior is due to the CyUSB.dll as explained by Cypress document CyUSB.NET.pdf, pages 132-133 and pages 139-140: “You should never invoke the Dispose method of a USBDevice directly. Rather, the appropriate technique is to call the Dispose method of the USBDeviceList object that contains the USBDevice objects”

This function differs from its homonym of the previous TE0300DLL.dll in that it does not close a Handle but disposes (erases) all USB devices in the list.

A more intuitive name for this function would have been TE_USB_FX2_CloseAll or TE_USB_FX2_Dispose.

3.3.4 Parameters

1. ref USBDeviceList USBdevList

   USBDeviceList is a type defined in CyUSB.dll. USBdevList is the list of Trenz Electronic USB FX2 devices attached to the USB bus host computer. This parameter is passed by reference (ref). See page 139-140 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference).

3.3.5 Return Value

1. bool : logical type

   This function returns true if it is able to dispose the list. If unable to do so, it returns false.

3.4 TE_USB_FX2_SendCommand()

3.4.1 Declaration

```
public static bool TE_USB_FX2_SendCommand(ref CyUSBDevice
TE_USB_FX2_USBDevice, ref byte[] Command, ref int CmdLength,ref
byte[] Reply, ref int ReplyLength, uint Timeout)
```

3.4.2 Function Call

Your application program shall call this function like this:

> TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand
> (ref TE_USB_FX2_USBDevice, ref Command, ref CmdLength, ref Reply, ref ReplyLength, Timeout);

3.4.3 Description

This function takes an already initialized USB device (previously selected by TE_USB_FX2_Open()) and sends a command (API command) to the USB FX2 microcontroller (USB FX2 API command) or to the MicroBlaze embedded processor (MicroBlaze API command) through the USB FX2 microcontroller endpoint EP1 buffer.
This function is normally used to send 64 bytes packets to the USB endpoint EP1 (0x01).
This function is also able to obtain the response of the USB FX2 microcontroller or MicroBlaze embedded processor through the USB FX2 microcontroller endpoint EP1 (0x81).

3.4.4 Parameters

1. ref CyUSBDevice TE_USB-FX2_USBDevice

   CyUSBDevice is a type defined in CyUSB.dll. This parameter points to the module selected by TE_USB_FX2_Open(). This parameter is passed by reference (ref). See pages 70-93 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference)

2. ref byte[] Command

   This parameter is passed by reference (ref). It is the byte array that contains the commands to send to USB FX2 microcontroller (FX2_Commands) or to the MicroBlaze embedded processor (MB_Commands).

   The byte array shall be properly initialized using instructions similar to the following ones:

```
Command[0] = (byte)FX2_Commands.I2C_WRITE;
Command[1] = (byte)FX2_Commands.MB_I2C_ADDRESS;
Command[2] = (byte)FX2_Commands.I2C_BYTES;
Command[3] = (byte)0;
Command[4] = (byte)0;
Command[5] = (byte)0;
Command[6] = (byte)Command2MB;
```

3. ref int CmdLength

   This parameter (passed by reference (ref)) is the length (in bytes) of the previous byte array; it is the length of the packet to transmit to USB FX2 controller endpoint EP1 (0x01). It is typically initialized to 64 bytes.

4. ref byte[] Reply

   This parameter (passed by reference (ref)) is the byte array that contains the response to the command sent to the USB FX2 microcontroller (FX2_Commands) or to the MicroBlaze embedded processor (MB_Commands).

5. ref int ReplyLength

   This parameter (passed by reference (ref)) is the length (in bytes) of the previous byte array; it is the length of the packet to transmit to the USB FX2 microcontroller endpoint EP1 (0x81). It is typically initialized to 64 byes, but normally the meaningful bytes are less. The parameter is a reference, meaning that the method can modify its value. The number of bytes actually received is passed back in ReplyLength.

6. uint Timeout

   The unsigned integer value is the time in milliseconds assigned to the synchronous method XferData() of data transfer used by CyUSB.dll.

   Timeout is the time that is allowed to the function for sending/receiving the data packet passed to the function; this timeout shall be large enough to allow the data/command transmission/reception. Otherwise the transmission/reception will fail. See 1.1.2 Timeout Setting.

3.4.5 Return Value

1. bool : logical type

   This function returns true if it is able to send a command to EP1 and receive a response within 2*Timeout milliseconds. This function returns false otherwise.

3.5 TE_USB_FX2_GetData()

3.5.1 Declaration

```
public static bool TE_USB_FX2_GetData(
  ref CyUSBDevice TE_USB_FX2_USBDevice, ref byte[] DataRead, ref
int DataReadLength, int PipeNo, uint Timeout, int BufferSize)
```

3.5.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_GetData
(ref TE_USB_FX2_USBDevice, ref DataRead, ref DataReadLength, PI_EP6, Timeout,
BufferSize);
```

3.5.3 Description

This function takes an already initialized USB Device (previously selected by TE_USB_FX2_Open()) and reads data from USB FX2 microcontroller endpoint EP6 (0x86) (endpoints EP4(0x84) or EP2(0x82) are also theoretically possible). Data comes from the FPGA. Currently (April 2012), only endpoint 0x86 is actually implemented in Trenz Electronic USB FPGA modules, so that endpoints EP2 and EP4 cannot be read or , more precisely, they are not even connected to the FPGA. That is why attempting to read them causes a function failure after Timeout expires.

3.5.4 Expected Data Throughput

The maximum data throughput expected (with a DataReadLength= 120*10^6) is 37 Mbyte/s (PacketSize = BufferSize = 131,072), but in fact this value is variable between 31-36 Mbyte/s (the mean value seems 33.5 Mbyte/s); so if you measure this range of values, the data reception can be considered as normal.

The data throughput is variable in two ways:

1. depends on the used host computer;
2. varies with every function call.

3.5.5 DataRead Size Shall Not Be Too Large

TE_USB_FX2_GetData() seems unable to use too large arrays or, more precisely, this fact seems variable by changing host computer. To be safe, do not try to transfer in a single packet very large data (e.g. 120 millions of byte); transfer the same data with many packets instead (1,200 packets * 100,000 byte) and copy the data in a single large data array if necessary (with Buffer.BlockCopy()). Buffer.BlockCopy seems not to hinder throughput too much (max 2 Mbyte/s)

3.5.5.1 *Reduced version (pseudo code)*

```
PACKETLENGTH=100000;
packets=1200;
byte[] data = new byte[packetlen*packets];
byte[] buffer = new byte[packetlen];

for (int i = 0; i < packets; i++)
{
  TE_USB_FX2_GetData(ref TE_USB_FX2_USBDevice, ref buffer, ref
packetlen, PI_EP6,  TIMEOUT_MS,BUFFER_SIZE)

  Buffer.BlockCopy(buffer, 0, data, total_cnt, packetlen);

  total_cnt += packetlen;
}
```

3.5.5.2 *Expanded version (code)*

```
PACKETLENGTH=100000;
packets=1200;
byte[] data = new byte[packetlen*packets];
byte[] buffer = new byte[packetlen];

//starts test: the FPGA start to write data in the buffer EP6 of
FX2 chip
SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_TX, TIMEOUT_MS);

test_cnt = 0;
total_cnt = 0;
for (int i = 0; i < packets; i++)
{
  //buffer = &data[total_cnt];
  packetlen = PACKETLENGTH;
  //fixed (byte* buffer = &data[total_cnt])
  bResultXfer = TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_GetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP6,
TIMEOUT_MS,BUFFER_SIZE);
  Buffer.BlockCopy(buffer, 0, data, total_cnt, packetlen);
  if (bResultXfer == false)
  {
    //cout << "ERROR" << endl;
    Console.WriteLine("Error Get Data");
    SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);
    return;
  }
  total_cnt += packetlen;
}
//stop test: the FPGA start to write data in the buffer EP6 of
//FX2 chip
SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);
```

3.5.6 DataRead Size Shall Not Be Too Small

There are two reasons why DataRead size shall not be too small.

The first reason is described in section 1.1.4 PacketSize. PacketSize has also a strong influence on DataThroughput. If PacketSize is too small (e.g. 512 byte), you can have very low DataThroughput (2.2 Mbyte/s) even if you use a large driver buffer (driver buffer size = 131,072 bytes). See section 6 TE_USB_FX2_CyUSB.dll: Data Transfer Throughput Optimization.

The second reason is that probably the FPGA imposes your minimum packet size. In a properly used read test mode (using FX22MB_REG0_START_TX and therefore attaching the FPGA), TE_USB_FX2_GetData() is unable to read less than 1024 byte. In a improperly used read test mode (not using FX22MB_REG0_START_TX and therefore detaching the FPGA), TE_USB_FX2_GetData() is able to read a packet size down to 64 byte. The same CyUSB method XferData() used (under the hood) in TE_USB_FX2_SendCommand() is able to read a packet size of 64 byte. These facts prove that the minimum packet size is imposed by FPGA. To be safe, we recommend to use this function with a size multiple of 1 kbyte.

3.5.7 Parameters

1. ref CyUSBDevice TE_USB-FX2_USBDevice

   This parameter points to the module selected by TE_USB_FX2_Open(). This parameter is passed by reference (ref). See pages 70-93 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference)

2. ref byte[] DataRead

   This parameter is passed by reference (ref). C# applications use directly TE_USB_FX2_CyUSB.dll based on CyUSB.dll. To avoid copying back and forth large amount of data between these two DLLs, data is passed by reference rather than by value. This parameter points to the byte array that, after the function returns, will contain the data read from the buffer EP6 of the USB FX2 microcontroller. The data contained in EP6 generated by the FPGA. If no data is contained in EP6, the byte array is left unchanged.

3. ref int DataReadLength

   This parameter is the length (in bytes) of the previous byte array; it is the length of the packet read from the USB FX2 microcontroller endpoint EP6 (0x86). It is typically PacketLength. This parameter is passed by reference (ref).

4. int PipeNumber

   This parameter is the value that identifies the endpoint used for data transfer. It is called PipeNumber because it identifies the buffer (pipe) used by the USB FX2 microcontroller.

5. uint Timeout

   It is the integer time value in milliseconds assigned to the synchronous method XferData() of data transfer used by CyUSB.dll. Timeout is the time that is allowed to the function for sending/receiving the data packet passed to the function; this timeout shall be large enough to allow data/command transmission/reception.. Otherwise the transmission/reception will fail. See 1.1.2 Timeout Setting.

6. int BufferSize

   It is the dimension (in bytes) of the driver buffer (SW) used in data reception of a single endpoint (EP6 0x86 in this case)***single endpoint (EP6 0x86 in this case)***; the total buffer size is the sum of BufferSize of every endpoint used. BufferSize has a strong influence on DataThroughput. If the BufferSize is too small, DataThroughput can be 1/3-1/2 of the maximum value (from a maximum value of 36 Mbyte/s for read transactions to an actual value of 18 Mbyte/s). If BufferSize has a large value (a roomy buffer), the program shall be able to cope with the non-deterministic behavior of C# without losing packets.

3.5.8 Return Value

1. bool : logical type

This function returns true if it is able to receive the data from buffer EP6 within Timeout milliseconds. This function returns false otherwise.

3.6 *TE_USB_FX2_SetData()*

3.6.1 Declaration

```
public static bool TE_USB_FX2_SetData(ref CyUSBDevice
TE_USB_FX2_USBDevice, ref byte[] DataWrite, ref int
DataWriteLength, int PipeNo, uint Timeout, int BufferSize)
```

3.6.2 Function Call

Your application program shall call this function like this:

> TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SetData
> (ref TE_USB_FX2_USBDevice, ref DataWrite, ref DataWriteLength, PI_EP8, Timeout,
> BufferSize);

3.6.3 Description

This function takes an already initialized USB device (CyUSBDevice is a type defined in CyUSB.dll), selected by TE_USB_FX2_Open(), and writes data to the USB FX2 microcontroller endpoint EP8 (0x08). This data is then passed to the FPGA.

If there is not a proper connection (not using FX22MB_REG0_START_RX) between FPGA and USB FX2 microcontroller, the function can experience a strange behavior. For example, a very low throughput (9-10 Mbyte/s even if a 22-24 Mbyte/s are expected) is measured or the function fails returning false. These happen because buffer EP8 (the HW buffer, not the SW buffer of the driver whose size is given by BufferSize parameter) is already full (it is not properly read/emptied by the FPGA) and no longer able to receive further packets.

3.6.4 Data throughput expected

The maximum data throughput expected (with a DataWriteLength= 120*10^6) is 24 Mbyte/s (PacketSize = BufferSize =131,072) but in fact this value is variable between 22-24 Mbyte/s (the mean value seems 24 Mbyte/s); so if you measure this range of values, the data reception can be considered normal.

The data throughput is variable in two way:

1. depends on which host computer is used (on some host computers this value is even higher: 29 Mbyte/s)
2. vary with every function call

3.6.5 DataWrite size shall not be too large

TE_USB_FX2_SetData() seems unable to use too large arrays or, more precisely, this fact seems variable by changing host computer. To be safe, do not try to transfer in a single packet very large data (120 millions of byte); transfer the same data with many packets (1,200 packets * 100,000 byte) and copy the data in a single large data array if necessary (with Buffer.BlockCopy()). Buffer.BlockCopy seems not to hinder throughput too much (max 2 Mbyte/s).

3.6.5.1 Reduced version (pseudo code)

```
PACKETLENGTH=100000;
packets=1200;
```

```
byte[] data = new byte[packetlen*packets];
byte[] buffer = new byte[packetlen];

for (int i = 0; i < packets; i++)
{
  Buffer.BlockCopy(data, total_cnt, buffer, 0, packetlen);

  TE_USB_FX2_SetData(ref TE_USB_FX2_USBDevice, ref buffer, ref
packetlen, PI_EP8,  TIMEOUT_MS,BUFFER_SIZE);
  total_cnt += packetlen;
}
```

3.6.5.2 Expanded version (code)

```
SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_RX, TIMEOUT_MS);

//ElapsedTime.Start(); //StopWatch start
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

for (int i = 0; i < packets; i++)
{
  packetlen = PACKETLENGTH;
  Buffer.BlockCopy(data, total_cnt, buffer, 0, packetlen);
  if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP8,
TIMEOUT_MS, BUFFER_SIZE) == false) errors++;
  else total_cnt += packetlen;
}
//total_cnt += (packetlen * packets);
stopWatch.Stop();
```

3.6.6 DataWrite size shall not be too small

The reason is described in section 1.1.4 PacketSize.

PacketSize has also a strong influence on DataThroughput. If PacketSize is too small (512 byte for example) you can achieve very low data throughput (2.2 Mbyte/s) even if you use a large driver buffer (driver buffer size = 131,072 byte). See 6 TE_USB_FX2_CyUSB.dll: Data Transfer Throughput Optimization.

3.6.7 Parameters

1. ref CyUSBDevice TE_USB-FX2_USBDevice

   This parameter is passed by reference (ref). It points to the module selected by TE_USB_FX2_Open(). See pages 70-93 of CyUSB.NET.pdf (Cypress CyUSB .NET DLL Programmer's Reference)

2. ref byte[] DataWrite

   This parameter is passed by reference (ref). C# applications use directly TE_USB_FX2_CyUSB.dll based on CyUSB.dll. To avoid copying back and forth large amount of data between these two DLLs, data is passed by reference and not by value.

   This parameter points to the byte array that contains the data to be written to buffer EP8

(0x08) of USB FX2 microcontroller. Data contained in EP8 are then read by the FPGA.

3. ref int DataWriteLength

   This parameter is passed by reference (ref). This parameter is the length (in bytes) of the previous byte array; it is the length of the packet read from FX2 USB endpoint EP6 (0x86). Normally it is PacketLength.

4. int PipeNumber

   This parameter is the value that identify the endpoint used for the data transfer. It is called PipeNumber because it identifies the buffer (pipe) used by the USB FX2 microcontroller.

5. uint Timeout.

   The unsigned integer value is the time in milliseconds assigned to the synchronous method XferData() of data transfer used by CyUSB.dll.

   Timeout is the time that is allowed to the function for sending/receiving the data packet passed to the function; this timeout shall be large enough to allow the data/command transmission/reception. Otherwise the transmission/reception will fail. See 1.1.2 Timeout Setting.

6. int BufferSize

   The integer value is the dimension (in bytes) of the driver buffer (SW) used in data transmission of a ***single endpoint (EP8 0x08 in this case);*** the total buffer size is the sum of all BufferSize of every endpoint used.

   The BufferSize has a strong influence on DataThroughput. If BufferSize is too small, DataThroughput can be 1/3-1/2 of the maximum value (from a maximum value of 24 Mbyte/s for write transactions to an actual value of 14 Mbyte/s). If BufferSize has a large value (a roomy buffer), the program shall be able to cope with the non-deterministic behavior of C# without losing packets.

3.6.8 Return Value

1. bool: logical type

   This function returns true if it is able to write data to buffer EP8 within Timeout milliseconds. This function returns false otherwise.

4 API Commands

4.1 Introduction

This introduction has been taken from "TE03xx Series Application Notes".

4.1.1 Reference Architecture

The Xilinx FPGA itself on the Trenz Electronic USB FX2 family by default is blank and has no architecture. To define an FPGA functionality, a logic architecture should be defined and loaded into the device. The reference design system was built using Xilinx Embedded Development Kit (EDK). Basically, it is an embedded system with a MicroBlaze 32-bit soft microprocessor. The MicroBlaze initializes and sets up the system. The XPS_I2C_SLAVE block sends commands coming from the USB bus towards the MicroBlaze processor (low speed communication channel). The horsepower for high bandwidth data streaming is a Multiport Memory Controller (MPMC). A custom-built DMA (direct memory access) engine (XPS_NPI_DMA) streams data between multiple sources and external RAM simultaneously. Standard EDK cores are used to implement a serial interface (XPS_UARTLITE), an SPI FLASH interface (XPS_SPI), a timer / counter block (XPS_TIMER) and an interrupt controller (XPS_INTC).

When data is sent from the USB-host to the USB FX2 family high-speed endpoint (high speed communication channel), it is automatically stored into the RAM by the DMA at a specified buffer location. The reference design software running on the MicroBlaze verifies the transferred data at the end of transmission and sends to the USB host a notification about the data test (pass/fail).

When data is sent form the Trenz Electronic USB FX2 family high-speed endpoint to the USB host, it is automatically fetched from the RAM via the DMA engine and forwarded to the XPS_FX2 core in 1-kbyte packets. MicroBlaze does the throttling to prevent XPS_FX2 TX FIFO overflow.

4.1.2 Custom Logic Block

The instructions contained in this document can be applied to all reference designs. Besides standard IP cores, they contain three custom IP cores:

1. XPS_NPI_DMA
2. XPS_FX2
3. XPS_I2C_SLAVE

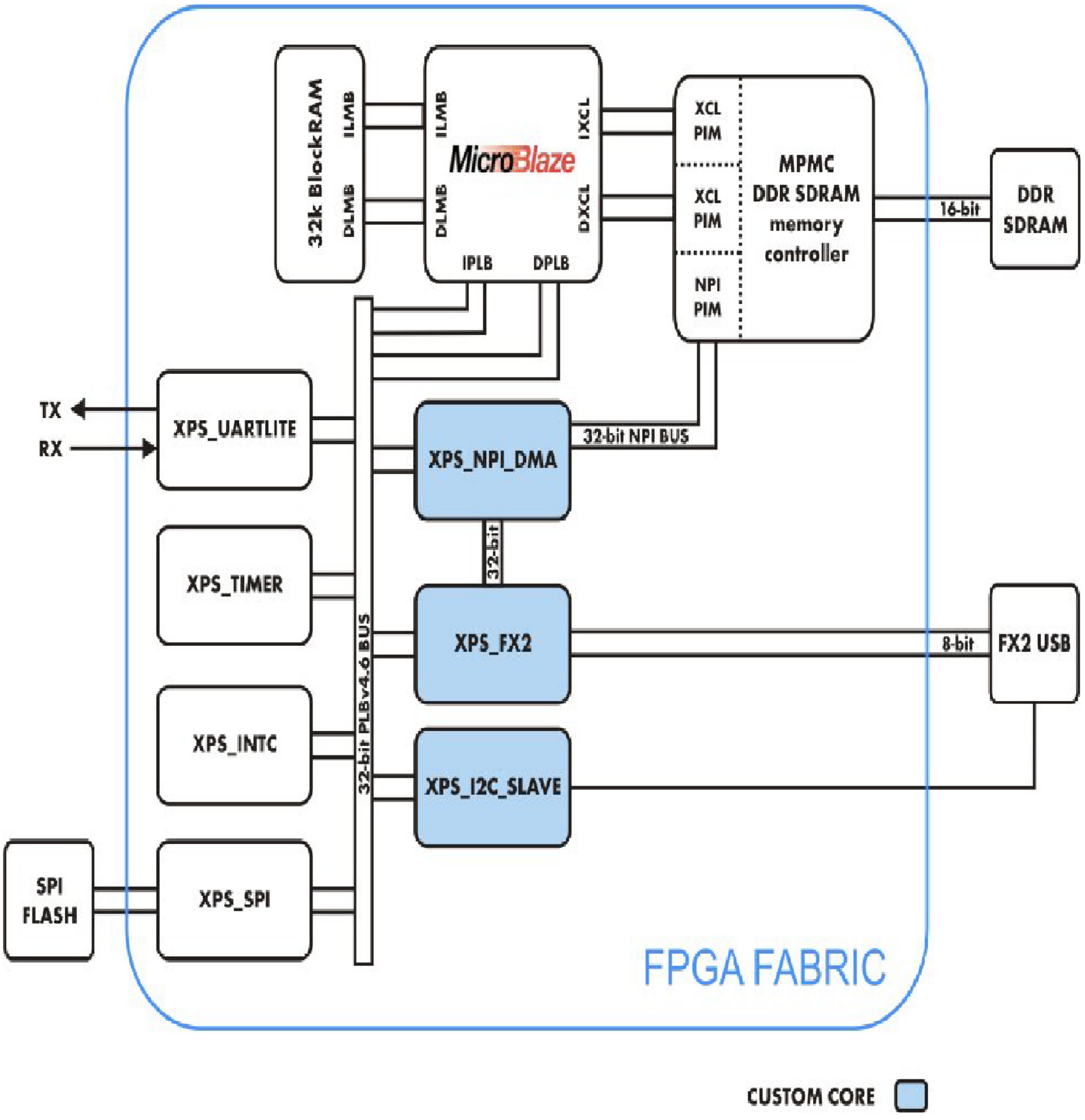
XPS_NPI_DMA is a high speed DMA (direct memory access) engine which connects to the MPMC (Multi-Port Memory Controller) VFBC (Video Frame Buffer Controller) port. It enables high speed data streaming to/from external memory (DDR SDRAM). It can be controlled by a processor using 6 x 32-bit memory mapped registers attached to the PLB (peripheral local bus). For more information about registers, see the Xilinx MPMC Product Specification (mpmc.pdf), "Video Frame Buffer Controller PIM" section .

XPS_FX2 *is a logic block for high speed bidirectional communication between the FPGA and a host PC. It contains two 2 kB FIFOs for data buffering. For more information about the 5 x 32-bit memory mapped registers see the #project_root#/pcores/xps_fx2_v1_00_a/doc.*

XPS_I2C_SLAVE is a logic block for low speed bidirectional communication between the FPGA and a host PC. It is usually used for command, settings and status communication. It contains 6 x 32-bit memory mapped registers:

- 3 for PC -> FPGA communication (FX2MB regs)
- 3 for FPGA -> PC communication (MB2FX2 regs)

When the PC sends commands to the MicroBlaze (MB) soft embedded processor, an interrupt is triggered. When the MB writes data to MB2FX2_reg0, the interrupt (INT0) is sent to the Cypress EZ-USB FX2LP USB microcontroller. When the FX2 microcontroller receives an interrupt, it reads all MB2FX2 regs.



The commands described in Table 1 are binary data packets sent/received by the USB FX2 microcontroller through endpoint 1. Endpoint 1 accepts 64 byte packets with a predefined structure. These command are sent using the API function TE_USB_FX2_SendCommand().

ID	Name	Description
0x00	READ_VERSION	Return 4 bytes representing FX2 firmware version
0xA0	INITIALIZE	Initialize FX2 to initial state
0xA1	READ_STATUS	Return 5 bytes of FX2 status
0xA4	RESET_FIFO	Reset selected FX2 FIFO
0xA5	FLASH_READ	Read data from SPI Flash
0xA6	FLASH_WRITE	Write data to SPI Flash
0xA7	FLASH_ERASE	Erase entire SPI Flash
0xA8	EEPROM_READ	Read data from I2C EEPROM
0xA9	EEPROM_WRITE	Write data from I2C EEPROM
0xAC	FIFO STATUS	Return FIFO status for all endpoints
0xAD	I2C_WRITE	Write data to I2C interface
0xAE	I2C_READ	Read data from I2C interface
0xAF	POWER	Control FPGA power supply
0xAA	FLASH_WRITE_COMMAND	Write SPI Flash command
0xB0	SET_INTERRUPT	Set parameters for interrupt handler
0xB1	GET_INTERRUPT	Return interrupt statistic information

Table 1 : USB FX2 API command list (commands accepted by the USB FX2 microcontroller firmware).

There are also some MicroBlaze commands that can be customized by users. Table 2 lists and describes briefly the default MicroBlaze commands accepted by the default MicroBlaze embedded processor implemented in Trenz Electronic USB FX2 FPGA modules.

ID	Name	Description
0x00	FX22MB_REG0_NOP	No operation
0x01	FX22MB_REG0_GETVERSION	Return 4 bytes representing FPGA firmware version
0x02	FX22MB_REG0_START_TX	Start read data integrity test of data transmitted from EP6 of FX2 to computer.
0x03	FX22MB_REG0_START_RX	Start write data integrity test of data transmitted from computer to EP8 of FX2
0x04	FX22MB_REG0_STOP	Stop both the test started by 0x02 and 0x03
0x05	FX22MB_REG0_PING	This command send a ping request. A “pong” 0x706F6E67 value shall be returned.

Table 2: MicroBlaze API commands list.

Table 3 lists some important parameters used in FX2 API commands, in case they are required by a MicroBlaze command.

ID	Name	Description
0x0C	I2C_BYTES	Number of bytes (12)
0x3F	MB_I2C_ADDRESS	Address of MicroBlaze over I2C

Table 3: USB FX2 API parameter list.

When writing applications, users shall include, typically at the beginning of their programs, code sections similar to the three following ones:

```
public enum FX2_Commands
{
  READ_VERSION = 0x00,
  INITALIZE = 0xA0,
  READ_STATUS = 0xA1,
  WRITE_REGISTER = 0xA2,
  READ_REGISTER = 0xA3,
  RESET_FIFO_STATUS = 0xA4,
  FLASH_READ = 0xA5,
  FLASH_WRITE = 0xA6,
  FLASH_ERASE = 0xA7,
  EEPROM_READ = 0xA8,
  EEPROM_WRITE = 0xA9,
  GET_FIFO_STATUS = 0xAC,
  I2C_WRITE = 0xAD,
  I2C_READ = 0xAE,
  POWER_ON = 0xAF,
  FLASH_WRITE_COMMAND = 0xAA,
  SET_INTERRUPT = 0xB0,
  GET_INTERRUPT = 0xB1,
};

public enum FX2_Parameters
{
  I2C_BYTES = 0x0C,
  MB_I2C_ADDRESS = 0x3F
};

public enum MB_Commands
{
  FX22MB_REG0_NOP = 0,
  FX22MB_REG0_GETVERSION = 1,
  FX22MB_REG0_START_TX = 2,
  FX22MB_REG0_START_RX = 3,
  FX22MB_REG0_STOP = 4,
  FX22MB_REG0_PING = 5
};
```

The byte array shall be properly initialized by using instructions similar to the following ones:

```
Command[0] = (byte)FX2_Commands.I2C_WRITE;
Command[1] = (byte)FX2_Parameters.MB_I2C_ADDRESS;
Command[2] = (byte)FX2_Parameters.I2C_BYTES;
Command[3] = (byte)0;
```

```
Command[4] = (byte)0;
Command[5] = (byte)0;
Command[6] = (byte)Command2MB;
```

4.2 USB FX2 API Commands

The first byte sent by TE_USB_FX2_SendCommand() is the USB FX2 API Command.

4.2.1 READ_VERSION

This command returns 4 bytes representing the USB FX2 firmware version.

Byte	Value	Description
1	0x00	READ_VERSION command ID
From 2 to 64	-	Not used

Table 4: READ_VERSION Command Packet Layout.

Byte	Description
1	FX2 Firmware version major number
2	FX2 Firmware version minor number
3	Device Major Number
4	Device Minor Number
From 5 to 64	Not Used

Table 5: READ_VERSION Reply Packet Layout.

4.2.2 INITIALIZE

This command runs the USB FX2 initialization process.

Byte	Value	Description
1	0xA0	INITIALIZE command ID
2	0x01	FIFO mode
From 3 to 64	-	Not used

Table 6: INITIALIZE Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.3 READ_STATUS

This command returns 5 bytes representing the USB FX2 status.

Byte	Value	Description
1	0xA1	READ_STATUS command ID
From 2 to 64	-	Not used

Table 7: READ_STATUS Command Packet Layout.

Byte	Description
1	FIFO error
2	Current mode
3	Flash busy
4	FPGA program
5	Booting
From 6 to 64	Not used

Table 8: READ_STATUS Reply Packet Layout.

4.2.4 RESET_FIFO

This command resets the FIFO of the selected endpoint (all endpoints if zero is selected).

Byte	Value	Description
1	0xA4	RESET_FIFO command ID
2	0/2/4/6/8	Endpoint number. 0 means all endpoints, not control endpoint.
From 3 to 64	-	Not used

Table 9: READ_VERSION Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.5 FLASH_READ

This command reads data (from 1 to 64 bytes) from the requested SPI Flash address.

Byte	Value	Description
1	0xA5	FLASH_READ command ID
2	Sector	Flash sector to read (address [23:16])
3	AddrHigh	High part of address (address [15:8])
4	AddrLow	Low part of address (address [7:0])
5	size	Number of bytes to read (max 64)
From 6 to 64	-	Not used

Table 10: FLASH_READ Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.6 FLASH_WRITE

This command writes data (from 1 to 59 bytes) to the requested SPI Flash address. Afterwards, it writes USB FX2 firmware, reads back data from Flash and returns it in a reply packet.

Byte	Value	Description
1	0xA6	FLASH_WRITE command ID
2	Sector	Flash sector to read (address [23:16])
3	AddrHigh	High part of address (address [15:8])
4	AddrLow	Low part of address (address [7:0])
5	size	Number of bytes to read (max 59)
From 6 to size+5	data	Data to write (size bytes)
From size+6 to 64	-	Not used

Table 11: FLASH_WRITE Command Packet Layout.

Byte	Description
From 1 to size	Readback result
From size to 64	Not used

Table 12: FLASH_WRITE Reply Packet Layout.

4.2.7 FLASH_ERASE

This command starts an entire Flash erase process. A full Flash erase process may take up to 30 seconds for M25PS32 SPI Flash chip (check your SPI Flash data sheet for actual time values). To control Flash busy status, use READ_STATUS command.

Byte	Value	Description
1	0xA7	FLASH_ERASE command ID
From 2 to 64	-	Not used

Table 13: FLASH_ERASE Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.8 EEPROM_READ

This command reads data (from 1 to 64 bytes) from requested EEPROM address.

Byte	Value	Description
1	0xA8	EEPROM_READ command ID
2	AddrHigh	High part of address (address [15:8])
3	AddrLow	Low part of address (address [7:0])
4	size	Number of bytes to read (max 64)
From 5 to 64	-	Not used

Table 14: EEPROM_READ Command Packet Layout.

Reply packet contains requested data.

4.2.9 EEPROM_WRITE

This command writes data (from 1 to 60 bytes) to the requested EEPROM address. Afterwards, it writes USB FX2 firmware, reads back data from EEPROM and returns it in a reply packet.

Byte	Value	Description
1	0xA9	EEPROM_WRITE command ID
2	AddrHigh	High part of address (address [15:8])
3	AddrLow	Low part of address (address [7:0])
4	size	Number of bytes to write (max 60)
From 5 to size+4	data	Data to write (size bytes)
From size+5 to 64	-	Not used

Table 15: EEPROM_WRITE Command Packet Layout.

Byte	Description
From 1 to size	Readback result
From size to 64	Not used

Table 16: EEPROM_WRITE Reply Packet Layout.

4.2.10 FIFO_STATUS

This command returns the FIFO status of all used endpoints. Status is the value of EP2CS, EP4CS, EP6CS and EP8CS USB FX2 registers. See USB FX2 documentation for detailed information.

Byte	Value	Description
1	0xAC	FIFO_STATUS command ID
From 2 to 64	-	Not used

Table 17: FIFO_STATUS Command Packet Layout.

Byte	Description
1	FX2 EP2CS Register value
2	FX2 EP4CS Register value
3	FX2 EP6CS Register value
4	FX2 EP8CS Register value
From 5 to 64	Not used

Table 18: FIFO_STAUS Reply Packet Layout.

4.2.11 I2C_WRITE

This command writes data (from 1 to 32 bytes) to the requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	Address	I2C Address MB_I2C_ADDRESS=0x3F
3	size	Number of bytes to write (max 32)
From 4 to size+3	data	Data to write (size bytes)
From size+4 to 64	-	Not used

Table 19: I2C_WRITE Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.12 I2C_READ

This command reads data (from 1 to 32 bytes) from the requested I2C address.

Byte	Value	Description
1	0xAE	I2C_READ command ID
2	Address	I2C Address
3	size	Number of bytes to write (max 32)
From 4 to 64	-	Not used

Table 20: EEPROM_WRITE Command Packet Layout.

Reply packet contains requested data.

4.2.13 POWER

This command controls some FPGA power supply sources.

Byte	Value	Description
1	0xAF	POWER command ID
2	power	0 = Power OFF state, 1 = Power ON state
From 3 to 64	-	Not used

Table 21: POWER Command Packet Layout.

Byte	Description
1	0 = Power OFF state, 1 = Power ON state
From 2 to 64	Not used

Table 22: POWER Reply Packet Layout.

4.2.14 FLASH_WRITE_COMMAND

This command sends instructions to the SPI Flash. See SPI Flash data sheet for detailed command description.

Byte	Value	Description
1	0xAA	FLASH_WRITE_COMMAND command ID
2	Write length	Write command length
3	Read length	Read command length
From 4 to write length +3	command	Write command
From write length+4 to 64	-	Not used

Table 23: FLASH_WRITE_COMMAND Command Packet Layout.

Byte	Description
From 1 to read length	SPI Data Out sequence
From read length +1 to 64	Not used

Table 24: FLASH_WRITE_COMMAND Reply Packet Layout.

4.2.15 SET_INTERRUPT

This command sets address and number of bytes to read from I2C bus when interrupt request is received.

Byte	Value	Description
1	0xB0	SET_INTERRUPT command ID
2	Address	I2C Address MB_I2C_ADDRESS=0x3F
3	size	Number of bytes to write (max 32)
From 4 to 64	-	Not used

Table 25: SET_INTERRUPT Command Packet Layout.

Reply packet doesn't contain any usable information.

4.2.16 GET_INTERRUPT

This command pulls the number of received interrupts and received data (number of bytes set by SET_INTERRUPT command) from the USB FX2.

Byte	Value	Description
1	0xB1	GET_INTERRUPT command ID
From 2 to 64	-	Not used

Table 26: GET_INTERRUPT Command Packet Layout.

Byte	Description
1	Interrupt number Zero means that GET_INTERRUPT has not been able to retrieve data because the interrupt created by SET_INTERRUPT has not yet been serviced.
From 2 to size+1	Interrupt data
From size+2 to 64	Not used

Table 27: GET_INTERRUPT Reply Packet Layout.

4.3 MicroBlaze API Commands

These commands differ from USB FX2 API commands because they are executed by the MicroBlaze and shall be sent with the I2C_WRITE USB FX2 API command; more precisely, after it in the Command byte array. I2C_WRITE USB FX2 API command (with the Commmand byte array) is itself a parameter of USB FX2 API function TE_USB_FX2_SendCommand().

The byte array shall be properly initialized using instructions similar to the ones listed below:

```
Command[0] = (byte)FX2_Commands.I2C_WRITE;
Command[1] = (byte)FX2_Parameters.MB_I2C_ADDRESS;
Command[2] = (byte)FX2_Parameters.I2C_BYTES;
Command[3] = (byte)0;
Command[4] = (byte)0;
Command[5] = (byte)0;
Command[6] = (byte)Command2MB;
```

Command2MB it is one of the commands listed in Table 2. This command writes data (from 1 to 32 bytes) to the requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	Command2MB	MB_Commands to send to the MicroBlaze
From 8 to 64	-	Not used

Table 28: MB_Command Packet Layout.

Reply packet doesn't contain any usable information.

4.3.1 FX22MB_REG0_NOP

This command is used as No Operation.

4.3.2 FX22MB_REG0_GETVERSION

This command is used to request the FPGA firmware version. This function is not able to return directly 4 bytes representing the FPGA firmware version. The procedure requested is the following:

1. SET_INTERRUPT on MB_I2C_ADDRESS requesting I2C_BYTES
2. I2C_WRITE with MB_Command FX22MB_REG0_GETVERSION at byte 7
3. GET_INTERRUPT

4.3.2.1 Code Form

//1)SET_INTERRUPT on MB_I2C_ADDRESS requesting I2C_BYTES

```
Command[0] = (byte)FX2_Commands.SET_INTERRUPT;
Command[1] = (byte)FX2_Parameters.MB_I2C_ADDRESS;
Command[2] = (byte)FX2_Parameters.I2C_BYTES;

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref TE_USB_FX2_USBDevice, ref
Command, ref CmdLength, ref Reply, ref ReplyLength, Timeout) = FALSE) return FALSE;
```

2)I2C_WRITE with MB_Command FX22MB_REG0_GETVERSION at byte 7

```
Command[0] = (byte)FX2_Commands.I2C_WRITE; //0xAD;//command I2C_WRITE
//Command[1] = (byte)FX2_Parameters.MB_I2C_ADDRESS;
//Command[2] = (byte)FX2_Parameters.I2C_BYTES;
Command[3] = (byte)0;
Command[4] = (byte)0;
Command[5] = (byte)0;
Command[6] = (byte)MB_Commands.FX22MB_REG0_GETVERSION;//1; //get FPGA version

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref TE_USB_FX2_USBDevice, ref
Command, ref CmdLength, ref Reply, ref ReplyLength, Timeout) = FALSE) return FALSE;
```

3)GET_INTERRUPT

```
Command[0] = (byte)FX2_Commands.GET_INTERRUPT; //0xB1;//command GET_INTERRUPT

TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref TE_USB_FX2_USBDevice, ref Command,
ref CmdLength, ref Reply, ref ReplyLength, Timeout)

if ((ReplyLength > 4) && (Reply[0] != 0))
{
  //Console.WriteLine("INT# : {0}", Reply[0]);
  Console.WriteLine("Major version: {0}", Reply[1]);
  Console.WriteLine("Minor version: {0}", Reply[2]);
  Console.WriteLine("Release version: {0}", Reply[3]);
  Console.WriteLine("Build version: {0}", Reply[4]);
}
```

4.3.2.2 Table Form

This command sets address and number of bytes to read from the I2C bus when an interrupt request is received.

Byte	Value	Description
1	0xB0	SET_INTERRUPT command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C	Number of bytes to write (max 32)
From 4 to 64	-	Not used

Table 29: SET_INTERRUPT Command Packet Layout.

Reply packet doesn't contain any usable information.

This command writes data (12 bytes) to requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	0x01	MB_Commands.FX22MB_REG0_GETVERSION It request to the MicroBlaze the return of 4 bytes representing FPGA firmware version
From 8 to 64	-	Not used

Table 30: FX22MB_REG0_GETVERSION MicroBlaze command.

Reply packet doesn't contain any usable information.

This command pulls the number of received interrupts and received data (number o bytes set by SET_INTERRUPT command) from USB FX2.

Byte	Value	Description
1	0xB1	GET_INTERRUPT command ID
From 2 to 64	-	Not used

Table 31: GET_INTERRUPT Command Packet Layout.

Byte	Description
1, reply[0]	Interrupt number. If zero means that GET_INTERRUPT has not been able to retry data because the interrupt created by SET_INTERRUPT has not yet been serviced.
2, reply[1]	Interrupt data [0] : Major Version
3, reply[2]	Interrupt data [1] : Minor Version
4, reply[3]	Interrupt data [2] : Release Version
5, reply[4]	Interrupt data [3] : Build Version
From 6 to 64	Not used

Table 32: GET_INTERRUPT Reply Packet Layout.

4.3.3 FX22MB_REG0_START_TX

This command starts reading data integrity test for the data transmitted from EP6 of the USB FX2 to the host computer.

This MicroBlaze command does not require the use of SET_INTERRUPT before and GET_INTERRUPT after. It is instead required to send this command before starting data transmission from the USB FX2 to the host computer. It is also required to use FX22MB_REG0_STOP after the data transmission is ended.

4.3.3.1 Combination 1 (simplified version)

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

//starts test
SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_TX, TIMEOUT_MS);

test_cnt = 0;
total_cnt = 0;
for (int i = 0; i < packets; i++)
{
  packetlen = PACKETLENGTH;
  TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_GetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP6,
TIMEOUT_MS,BUFFER_SIZE);
  Buffer.BlockCopy(buffer, 0, data, total_cnt, packetlen);
  total_cnt += packetlen;
}

SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);

stopWatch.Stop();
TimeSpan ts = stopWatch.Elapsed;
```

4.3.3.2 *Combination 2 (simplified version)*

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

//starts test

test_cnt = 0;
total_cnt = 0;
for (int i = 0; i < packets; i++)
{
  SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_TX, TIMEOUT_MS);
  packetlen = PACKETLENGTH;
  TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_GetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP6,
TIMEOUT_MS,BUFFER_SIZE);
  Buffer.BlockCopy(buffer, 0, data, total_cnt, packetlen);
  total_cnt += packetlen;

  SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);
}

stopWatch.Stop();
TimeSpan ts = stopWatch.Elapsed;
```

Note: If you use combination 2,

- the data throughput is halved with regard to combination 1 and
- the test will fail because it is not the way it is supposed to be used.

This command writes data (12 bytes) to the requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	0x02	MB_Commands.FX22MB_REG0_START_TX Start read data integrity test of data transmitted from EP6 of FX2 to computer.
From 8 to 64	-	Not used

Table 33: FX22MB_REG0_START_TX MicroBlaze command.

Reply packet doesn't contain any usable information.

4.3.4 FX22MB_REG0_START_RX

This command starts writing data integrity test for the data transmitted from the host computer to EP8 of the USB FX2.

This MicroBlaze command does not require the use of SET_INTERRUPT before and GET_INTERRUPT after. It is instead required to send this command before starting data transmission from the host computer to the USB FX2. It is also required to use FX22MB_REG0_STOP after the data transmission is ended.

4.3.4.1 Combination 1 (simplified version)

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

//starts test
SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_TX, TIMEOUT_MS);

test_cnt = 0;
total_cnt = 0;
for (int i = 0; i < packets; i++)
{
  packetlen = PACKETLENGTH;

  Buffer.BlockCopy(data, total_cnt, buffer, 0, packetlen);
  TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP6,
TIMEOUT_MS,BUFFER_SIZE);
  total_cnt += packetlen;
}

SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);

stopWatch.Stop();
TimeSpan ts = stopWatch.Elapsed;
```

4.3.4.2 Combination 2 (simplified version)

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

//starts test

test_cnt = 0;
total_cnt = 0;
for (int i = 0; i < packets; i++)
{
  SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_START_TX, TIMEOUT_MS);
  packetlen = PACKETLENGTH;
  Buffer.BlockCopy(data, total_cnt, buffer, 0, packetlen);
  TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SetData(ref
TE_USB_FX2_USBDevice, ref buffer, ref packetlen, PI_EP6,
TIMEOUT_MS,BUFFER_SIZE);
  total_cnt += packetlen;

  SendFPGAcommand(ref TE_USB_FX2_USBDevice,
MB_Commands.FX22MB_REG0_STOP, TIMEOUT_MS);
}

stopWatch.Stop();
TimeSpan ts = stopWatch.Elapsed;
```

Note: If you make the combination 2

- data throughput is halved with regard to combination 1 and
- the test will fail because it is not the way it is supposed to be used.

This command writes data (12 bytes) to the requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	0x03	MB_Commands.FX22MB_REG0_START_RX Start read data integrity test of data transmitted from computer to EP8 of FX2.
From 8 to 64	-	Not used

Table 34: FX22MB_REG0_START_RX MicroBlaze command.

Reply packet doesn't contain any usable information

4.3.5 FX22MB_REG0_STOP

This command stops both the test started by FX22MB_REG0_START_TX (0x02, read test) and FX22MB_REG0_START_RX (0x03, write test).

This MicroBlaze command does not require the use of SET_INTERRUPT before and GET_INTERRUPT after. It is instead required to send this command after

the data transmission (form the USB FX2 to the host computer or from the host computer to the USB FX2) is ended. See 4.3.3 FX22MB_REG0_START_TX and 4.3.4 FX22MB_REG0_START_RX for more information.

This command writes data (12 bytes) to the requested I2C address.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	0x04	MB_Commands.FX22MB_REG0_STOP Stop both the test started by FX22MB_REG0_START_TX (0x02, read test) and FX22MB_REG0_START_RX (0x03, write test)
From 8 to 64	-	Not used

Table 35: FX22MB_REG0_STOP MicroBlaze command.

Reply packet doesn't contain any usable information.

4.3.6 *FX22MB_REG0_PING*

This command writes data (12 bytes) to the requested I2C address. A “pong” 0x706F6E67 value shall be returned.

Byte	Value	Description
1	0xAD	I2C_WRITE command ID
2	0x3F	I2C Address MB_I2C_ADDRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	-
5	0x00	-
6	0x00	-
7	0x05	MB_Commands.FX22MB_REG0_PING
From 8 to 64	-	Not used

Table 36: FX22MB_REG0_PING MicroBlaze command.

Reply packet contains the value 0x706F6E67.

5 API Usage Example Program

5.1 First Example: select module, read firmware version, read VID/PID

This example program implements a simple console that

1. creates an instance (TE_USB_FX2_USBDevice) initialized to null of the class CyUSBDevice
2. reads and displays the number of Trenz Electronic modules (TE_USB_FX2_ScanCards())
3. selects the first (0) Trenz Electronic module:
   modifies the instance of the class CyUSBDevice, making this instance different from null and properly initialized with the data of the selected Trenz Electronic module. If no module is attached, the instance remains initialized to null.
4. reads and displays VID and PID of the selected module
5. reads the firmware version of the selected module
6. selects the second (1) Trenz Electronic module:
   modifies the instance of class CyUSBDevice to represent the new selected Trenz Electronic module. If the selected module is not attached, the instance is reinitialized to null and the reference to the previous module (0) is lost.
7. reads the FPGA firmware version of the second (1) module

```
// This line creates a concrete class that is able to manage one
single FX2
//device (one single Trenz module)
CyUSBDevice TE_USB_FX2_USBDevice = null;

//This line creates a list of USB device that are used through
Cypress driver
USBDeviceList USBdevList = new USBDeviceList(CyConst.DEVICES);

//This line read the number of modules (cards) that are identified
by VID and PID //as Trenz Electronic modules
int NumberOfCardAttached =
TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_ScanCards(ref USBdevList);
Console.WriteLine("The number of card is {0} ",
NumberOfCardAttached);

//If you want use the first Trenz Electronic module use 0
if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_Open(ref
TE_USB_FX2_USBDevice, ref USBdevList, 0) == false)
Console.WriteLine("Module is not connected!");
else Console.WriteLine("Module is connected!");

byte[] Command = new byte[64];
byte[] Reply = new byte[64];
int CmdLength = 64;
int ReplyLength = 64;
// Timeout of 1000 milliseconds
uint TIMEOUT_MS = 1000;
```

```
if (TE_USB_FX2_USBDevice == null)
{
  Console.WriteLine("Error,no device is selected");
  return;
}
else
{
  UInt16 VID = TE_USB_FX2_USBDevice.VendorID;
  Console.WriteLine("VID {0:X4} e ", VID);

  UInt16 PID = TE_USB_FX2_USBDevice.ProductID;
  Console.WriteLine("PID {0:X4} e ", PID);
}

//comand read FX2 version
Command[0] = (byte)FX2_Commands.READ_VERSION;

Console.WriteLine("cmd[0] {0:X2} ", cmd[0]);

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref
TE_USB_FX2_USBDevice, ref Command, ref CmdLength, ref Reply, ref
ReplyLength, TIMEOUT_MS) == true)
{
  if (ReplyLength >= 4)
  {
    //printf("Major version: %d \n", reply[0]);
    //printf("Minor version: %d \n", reply[1]);
    //printf("Device hi: %d \n", reply[2]);
    //printf("Device lo: %d \n", reply[3]);
    Console.WriteLine("Major version: {0}", Reply[0]);
    Console.WriteLine("Minor version: {0}", Reply[1]);
    Console.WriteLine("Device hi: {0}", Reply[2]);
    Console.WriteLine("Device lo: {0}", Reply[3]);
     }
}
else
//cout << "Error" << endl;
Console.WriteLine("Error");

//If you want use the first Trenz Electronic module use 1
if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_Open(ref
TE_USB_FX2_USBDevice, ref USBdevList, 1) == false)
Console.WriteLine("Module is not connected!");
else Console.WriteLine("Module is connected!");

if (TE_USB_FX2_USBDevice == null)
{
  Console.WriteLine("Error,no device is selected");
  return;
}

//byte cmd[64], reply[64];
```

```
byte[] Command1 = new byte[64];
byte[] Reply1 = new byte[64];
int CmdLength1 = 64;
int ReplyLength1 = 64;

uint TIMEOUT_MS1 = 1000;

//Here two alternatives to initialize Command1 are shown
byte MB_I2C_ADRESS = 0x3f;
byte I2C_BYTES = 12;

Command1[0] = (byte)FX2_Commands.SET_INTERRUPT;
Command1[1] = MB_I2C_ADRESS;
Command1[2] = I2C_BYTES;

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref
TE_USB_FX2_USBDevice, ref Command1, ref Cmdlength1, ref Reply1,
ref ReplyLength1, TIMEOUT_MS) == false)
{
  //cout << "Error" << endl;
  Console.WriteLine("Error Send Command SET INTERRUPT");
  return;
}

Command1[0] = 0xAD;//comand I2C_WRITE
Command1[3] = 0;
Command1[4] = 0;
Command1[5] = 0;
Command1[6] = 1; //get FPGA version

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref
TE_USB_FX2_USBDevice, ref Command1, ref CmdLength1, ref Reply1,
ref ReplyLength1, TIMEOUT_MS) == false)
{
  //cout << "Error" << endl;
  Console.WriteLine("Error Send Command Get FPGA Version");
  return;
}

Command1[0] = 0xB1;//comand (byte)FX2_Commands.GET_INTERRUPT

if (TE_USB_FX2.TE_USB_FX2.TE_USB_FX2_SendCommand(ref
TE_USB_FX2_USBDevice, ref Command1, ref CmdLength1, ref Reply1,
ref ReplyLength1, TIMEOUT_MS) == true)
{
  if ((ReplyLength1 > 4) && (Reply1[0] != 0))
  {
    //Console.WriteLine("INT# : {0}", reply[0]);
    Console.WriteLine("Major version: {0}", Reply1[1]);
    Console.WriteLine("Minor version: {0}", Reply1[2]);
    Console.WriteLine("Release version: {0}", Reply1[3]);
    Console.WriteLine("Build version: {0}", Reply1[4]);
  }
}
else Console.WriteLine("Error, GET INTERRUPT");
```

5.2 Second Example: Read Test

See section 4.3.3 FX22MB_REG0_START_TX.

5.3 Third Example: Write Test

See section 4.3.4 FX22MB_REG0_START_RX.

6 TE_USB_FX2_CyUSB.dll: Data Transfer Throughput Optimization

6.1 Introduction

XferSize is the dimension (in bytes) of the buffer reserved (on the host computer) for the data transfer over the USB channel between one USB FX2 endpoint and the host computer.

PacketSize is the dimension (in bytes) of the data array to be transferred over the USB channel between one USB FX2 endpoint and the host computer. This data array is subdivided into packets of dimension ≤ MaxPktSize = 512 and scheduled for transmission over the USB channel.

6.2 XferSize (driver buffer size) Influence

Given a PacketSize of 102,400 bytes (it can be subdivided into 200 USB packets of 512 bytes), the influence of XferSize (driver buffer size reserved for data communication) on the throughput is reported in the following table.

XferSize (bytes)	**Throughput (Mbyte/s)**
PacketSize = 102,400 bytes	
4,096 (Cypress Default)	15.4
8,192	20.4
16,384	26.7
32,768	30.4
65,536	34.2
131,072	36.5
262,144	36.8

Table 37: data throughput as a function of XferSize given PacketSize = 102,400 bytes.

To change XferSize in C#, the method

```
    endpointIdentifier.XferSize = DesiredValue;
```

shall be used. Cypress sets DesiredValue to 4,096 bytes by default. This default value is not documented in the CyUSB.dll manual (pag 110-111 of CyUSB.NET.pdf), but it has been retrieved by using the following C# instructions:

```
int XferSizeReadValue = outEndPointPipeNo.XferSize;
Console.WriteLine("XferSize {0} ", XferSizeReadValue);
```

6.3 PacketSize (transfer data size) Influence

Given an XferSize (driver buffer size) of 131,072 bytes, the influence of PacketSize on the throughput is reported in the following table.

Packet Size (bytes)	Throughput (Mbyte/s)
XferSize = 131,072 Bytes	
512	2.26
1,024	4.15
2,048	7.78
4,096	15.44
8,192	20.23
16,384	25.34
32,768	31.18
65,536	35.52
131,072	37.06

Table 38: data throughput as a function of PacketSize given XferSize = 102,400 bytes.

To transfer the array of data with dimension PacketSize in C#, the method XferData() shall be used.

6.4 Conclusion

If a higher throughput is desired,

1. the value of XferSize shall be greater than the default one
2. the data to be transferred shall be organized in large data array(s)

Recommended values are :

- XferSize = 131,072 bytes and PacketSize = 131,072 bytes
  for a throughput of ≈ 37 Mbyte/s.
- XferSize = 65,536 bytes and PacketSize = 65,536 bytes
  for a throughput of ≈ 35 Mbyte/s.

6.5 Appendix : Charts

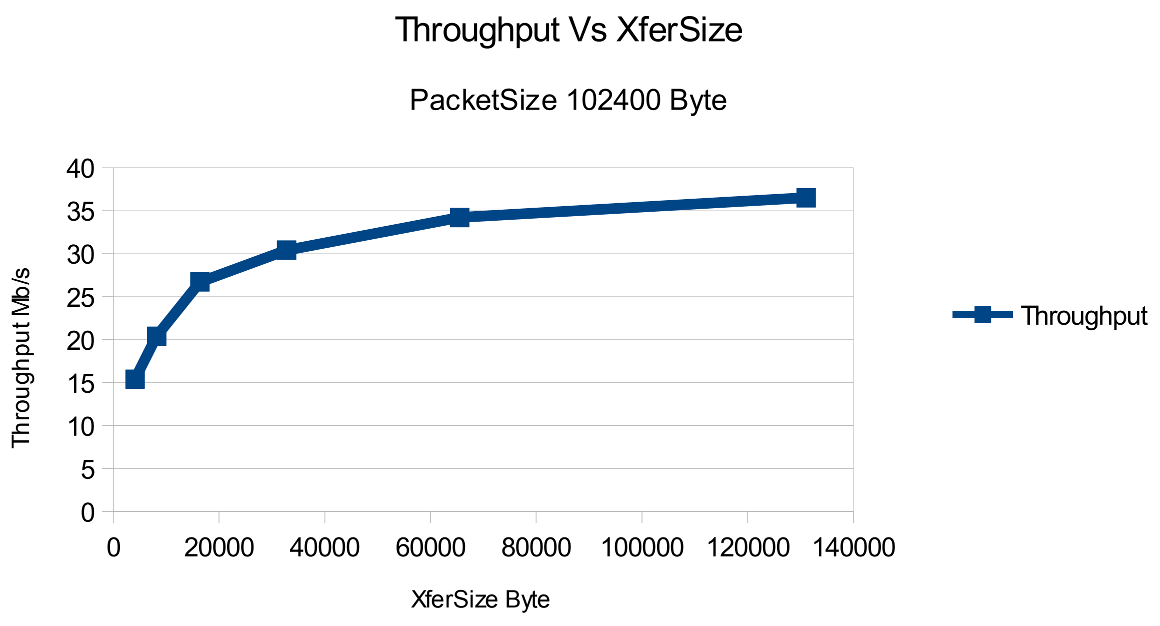


Chart 1: data throughput [Mbyte/s] as a function of XferSize [byte] given PacketSize = 102,400 bytes.

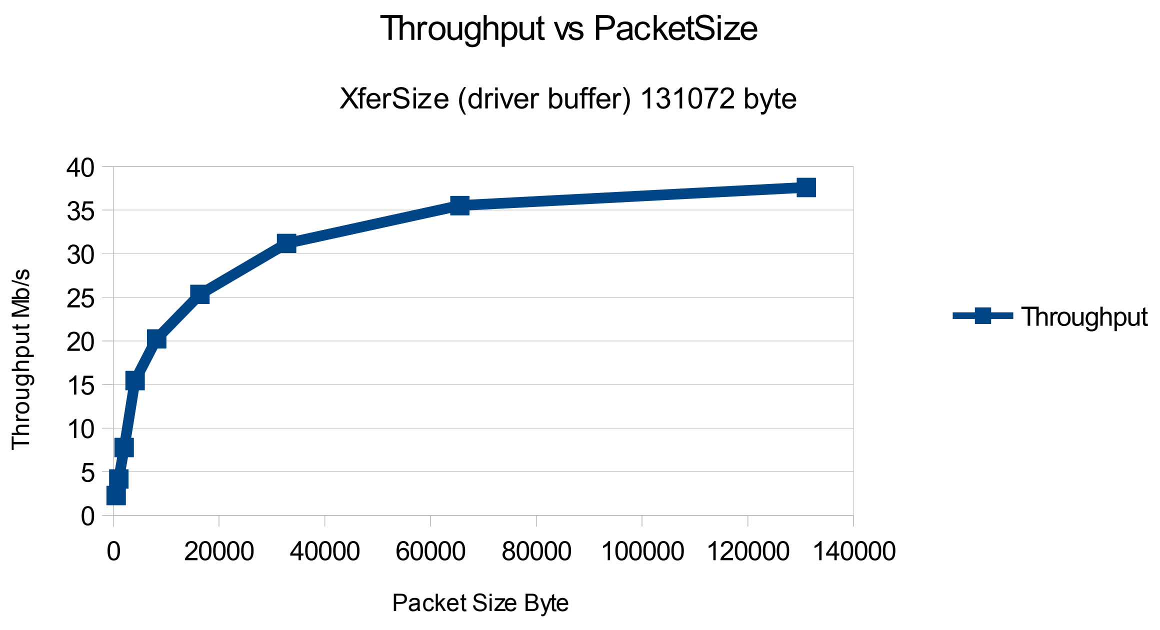


Chart 2: data throughput [Mbyte/s] as a function of PacketSize [byte] given XferSize = 102,400 bytes.

7 Document Change History

version	date	author	description
0.9	2012-06-01	SP, FDR	Release Preview.
1.0			Initial release.
1.2	2013-04-05	FDR	Improved "Hardware, firmware and software stack" table.

8 Bibliography

[1] TE03xx Series Application Notes, Xilinx Spartan-3* Industrial-Grade FPGA Micromodules

AN-TE03xx (v2.01) April 6, 2011

http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0300_series/TE0300/documents/AN-TE03xx.pdf

[2] AN14557 - Introduction to CyUSB.dll Based Application Development Using C#

Last Updated: 02/20/2012

http://www.cypress.com/?rID=12974

[3] Cypress CyUSB Programmer's Reference, 2010 Cypress Semiconductor

more recent version inside Cypress Suite USB 3.4.7

http://cosmiac.ece.unm.edu/images/b/be/CyUSB_NET.pdf

[4] Cypress CyUsb.sys Programmer's Reference

http://www.cypress.com/?docID=26658