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_CyAPI.dll
2. API commands

C++ applications use directly TE_USB_FX2_CyAPI.dll based on CyAPI.lib. 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_CyAPI.dll (C++) (API commands are inserted here in the commands data array of byte)
	CyAPI.lib (C++) (Cypress C++ 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 C++ CyAPI.lib 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:

TimeOut (ms) = [DataLength / DataThroughput] + 1 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_CyAPI.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_CyAPI.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_CyAPI.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
- C++ Run Time: it is contained in
  - Microsoft Visual C++ 2010 x64 Redistributable Setup: vcredist_x64.exe for 64 bit.
  - Microsoft Visual C++ 2010 x86 Redistributable Setup: vcredist_x86.exe for 32 bit

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_CyAPI.dll and CyAPI.lib have been used.

The API for 32 and 64 bit operating systems are located in two different folders:

- TE-USB-Suite/TE_USB_FX2_CyAPI_SampleApplication/TE_USB_FX2_CyAPI_SampleApplication/DLL32/
- TE-USB-Suite/TE_USB_FX2_CyAPI_SampleApplication/TE_USB_FX2_CyAPI_SampleApplication/DLL64/

These folders come from the folder FileToExportForApplication in the project folder TE-USB-Suite/TE_USB_FX2_CyAPI/.

FileToExportForApplication/ contains TE_USB_FX2_CyAPI.h, CyAPI.h and two folders; DLL32/ and DLL64/. DLL32/ and DLL64/ folders contain files with the same name but compiled respectively for 32 or 64 bit operating systems.

You shall select 32 or 64 bit for the compilation. To do this you shall:

1. in “Explore Solution” panel (top right window), right click the second line (between “Solution” and “External Dependencies”)
2. select “Properties”
3. left click "Configuration Management" (top right)
4. in "Active Solution Platform", select Win32 or x64
5. click “Close”
6. click “OK”

To create a program, you shall copy these files to the project folder.

User programs should load these libraries and initialize module connection to get access to API functions. To do this, you shall:

1. copy TE_USB_FX2_CyAPI.h, TE_USB_FX2_CyAPI.dll, TE_USB_FX2_CyAPI.lib, CyAPI.h and CyAPI.lib to the project folder (for example TE-USB-Suite/TE_USB_FX2_CyAPI_SampleApplication/TE_USB_FX2_CyAPI_SampleApplication/);
2. open the C++ project (double click the TE_USB_FX2_CyAPI_SampleApplication icon in the folder TE-USB-Suite/TE_USB_FX2_CyAPI_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 "Header File";
5. select Add. A new window (Add Header File) opens;
6. the term "Look In" shall have automatically selected the correct folder (TE-USB-Suite/TE_USB_FX2_CyAPI_SampleApplication/TE_USB_FX2_CyAPI_SampleApplication/).

If it is not so, you shall select the folder where you have copied the previous DLLs and header files;

7. left click one of the two header files (CyAPI.h or TE_USB_FX2_CyAPI.h);
8. select OK;
9. repeat steps from 4 to 8 for the second header file;
10. in the right panel "Explore Solution", right click "Resource File";
11. select Add. A new window (Add Resource File) opens;
12. the term "Look In" shall have automatically selected the correct folder (TE-USB-Suite/TE_USB_FX2_CyAPI_SampleApplication/TE_USB_FX2_CyAPI_SampleApplication/).
    If is not so, you shall select the folder where you have copied the previous DLLs and header files;
13. left click one of the three DLL files (TE_USB_FX2_CyAPI.dll, TE_USB_FX2_CyAPI.lib or CyAPI.lib);

    Note
    To compile the project it is strictly necessary only TE_USB_FX2_CyAPI.dll, the two lib files are optional.
14. select OK;
15. repeat steps from 10 to 14 for the second and third DLL file (if you want add the *.lib file also).

Exported function list:

- TE_USB_FX2_ScanCards()
- TE_USB_FX2_Open()
- TE_USB_FX2_Close()
- TE_USB_FX2_SendCommand()
- TE_USB_FX2_GetData_InstanceDriverBuffer()
- TE_USB_FX2_GetData()
- TE_USB_FX2_SetData_InstanceDriverBuffer()
- TE_USB_FX2_SetData()

3.1 TE_USB_FX2_ScanCards()

3.1.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_ScanCards(
  CCyUSBDevice *USBDeviceList)
```

3.1.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2_ScanCards(USBDeviceList);

3.1.3 Description

This function takes an already initialized USB device list (USBDeviceList), searches for Trenz Electronic USB FX2 devices (Cypress driver derivative and VID = 0xbd0, PID=0x0300) 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. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyAPI.lib. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices.

   CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI 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

```
TE_USB_FX2_CYAPI int TE_USB_FX2_Open(
  CCyUSBDevice *USBDeviceList, int CardNumber)
```

3.2.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2_Open(USBDeviceList, CardNumber);

3.2.3 Description

This function takes an already initialized 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, USBDeviceList ***is not initialized to null*** (the device list is not erased). An internal operation that closes an handle to the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys) is executed instead (see page 33 of CyAPI.pdf).
If one or more devices are attached and

- if 0 ≤ CardNumber ≤ (number of attached devices – 1), then
  the selected module is not directly given by USBDeviceList (CCyUSBDevice type). An internal operation that opens a handle to CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys) is executed instead (see page 45 of CyAPI.pdf). This handle is internally managed by CyAPI.lib, therefore there is no need to expose them to the user.
- if CardNumber ≥ number of attached devices, then
  USBDeviceList (CyUSBDevice type) ***is not initialized to null*** (the device list is not erased). An internal operation that closes an handle to CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys) is executed instead (see page 33 of CyAPI.pdf).

A more intuitive name for this function would have been TE_USB_FX2_SelectCard().
You can use this function to select the card desired without the need to call Close before.

3.2.4 Parameters

1. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyAPI.lib. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices.

   CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI Programmer's Reference).

2. int CardNumber.

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

3.2.5 Return Value

1. int : integer type

   This function returns true (ST_OK=0) if it is able to find the module selected by CardNumber. If unable to do so, it returns false (ST_ERROR=1).

```
  enum ST_Status
       {
            ST_OK = 0,
            ST_ERROR = 1
        };
```

3.3 *TE_USB_FX2_Close()*

3.3.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_Close(CCyUSBDevice *USBDeviceList)
```

3.3.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2_Close(USBDeviceList);
```

3.3.3 Description

This function takes an already initialized USB device list, searches for Trenz Electronic USB FX2 devices (any Cypress driver derivative and VID = 0xbd0, PID=0x0300) and opens (and immediately after closes) the first device found.
The selected module is not directly given by USBDeviceList (CCyUSBDevice type). An internal operation that opens and immediately after closes an handle to CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys) is executed instead (see page 45 of CyAPI.pdf). The *open* method closes every other handle already opened, and *close* method closes the only handle open; in this way, all handles are closed. These handles are internally managed by CyAPI.lib and there is no need to expose them to the user.
Note. After the execution of this function, no internal handle is open.
This function does not differ much from from its homonym of the previous TE0300DLL.dll; the only difference is that this function closes a handle (like TE0300DLL.dll) to the driver but the handle is not exposed to user (unlike TE0300DLL.dll).

3.3.4 Parameters

1. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyAPI.lib. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices. CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI Programmer's Reference).

2. int CardNumber.

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

3.3.5 Return Value

1. int : integer type

   This function returns true (ST_OK=0) if it is able to find the module selected by CardNumber. If unable to do so, it returns false (ST_ERROR=1).

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

3.4 TE_USB_FX2_SendCommand()

3.4.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_SendCommand(
  CCyUSBDevice *USBDeviceList, byte* Command, long CmdLength,
byte* Reply, long ReplyLength, unsigned long Timeout)
```

3.4.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2_SendCommand(
  USBDeviceList, Command, CmdLength, Reply, ReplyLength, Timeout);
```

3.4.3 Description

This function takes an already initialized USB device list (USBDeviceList 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. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyAPI.lib. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices. CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI Programmer's Reference).

2. byte* Command

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

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

```
Command[0] = I2C_WRITE;
Command[1] = MB_I2C_ADRESS;
Command[2] = I2C_BYTES;
Command[3] = 0;
Command[4] = 0;
Command[5] = 0;
Command[6] = Command2MB;
```

3. long CmdLength

   This parameter 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. byte* Reply

   This parameter (passed by pointer) is the pointer to 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. long ReplyLength.

   This parameter 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.

6. unsigned long Timeout.

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

   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. int : integer type

   This function returns true (ST_OK=0) if it is able to send a command to EP1 and receive a response within 2*Timeout milliseconds. This function returns false (ST_ERROR=1) otherwise.

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

3.5 TE_USB_FX2_GetData_InstanceDriverBuffer()

3.5.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_GetData_InstanceDriverBuffer (
  CCyUSBDevice *USBDeviceList, CCyBulkEndPoint **BulkInEP,
PI_PipeNumber PipeNo,unsigned long Timeout, int BufferSize)
```

3.5.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2_GetData_InstanceDriverBuffer
(USBDeviceList, &BulkInEP, PipeNo, TimeOut, BufferSize);

3.5.3 Description

This function takes an already initialized USB device list (USBDeviceList previously selected by TE_USB_FX2_Open()) and a not initialized CCyBulkEndPoint double pointer, BulkInEP. This function selects the endpoint to use: you shall choose EP6 (0x86) (endpoints EP4(0x84) or EP2(0x82) are also theoretically possible).
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.

TE_USB_FX2_GetData_InstanceDriverBuffer() function instantiates the class used by CyAPI to use bulk endpoint (CCyBulkEndPoint, see pages 9 to 11 of CyAPI.pdf (Cypress CyAPI Programmer's Reference)) and initializes the parameters of this class instantiation. The parameters are :

1. Timeout
2. XMODE_DIRECT (this parameter set the driver to single buffering, instead the slower double buffering)
3. DeviceDriverBufferSize.

The last parameter force the instantiation of the driver buffer (SW side, on the host computer) for the endpoint 0x86; this buffer has a size in byte given by DeviceDriverBufferSize. This value is of great importance because the data throughput is strongly influenced by this parameter (see section 6 TE_USB_FX2_CyAPI.dll: Data Transfer Throughput Optimization).

This function has not been included in TE_USB_FX2_GetData() for throughput reasons; if the driver buffer instantiation were repeated at every data reception, the data throughput would be halved. This function shall be used only one time to instantiate the driver buffer; after instantiation, TE_USB_FX2_GetData() can be used repeatedly without re-instantiating the driver buffer.

```
int RX_PACKET_LEN = 51200;//102400;

int packetlen = RX_PACKET_LEN;
unsigned int packets = 500;//1200;//1200;
unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;
unsigned long TIMEOUT= 18;
byte * data;
byte * data_temp = NULL;
```

```
unsigned int total_cnt = 0;
unsigned int errors = 0;

data = new byte [RX_PACKET_LEN*packets]; //allocate memory

PI_PipeNumber PipeNo = PI_EP6;

//starts test
SendFPGAcommand(USBDeviceList,FX22MB_REG0_START_TX);

CCyBulkEndPoint *BulkInEP = NULL;

TE_USB_FX2_GetData_InstanceDriverBuffer (USBDeviceList,
&BulkInEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);

ElapsedTime.Start(); //StopWatch start
for (unsigned int i = 0; i < packets; i++)
{
  packetlen = TX_PACKET_LEN;
  data_temp = &data[total_cnt];
  if (TE_USB_FX2_GetData(&BulkInEP, data_temp, packetlen))
  {
    cout << "ERROR read" << endl;
    errors++;
    break;
  }
  total_cnt += packetlen;
}
//DEBUG StopWatch
TheElapsedTime = ElapsedTime.Stop(false);

SendFPGAcommand(USBDevicelist,FX22MB_REG0_STOP);
```

3.5.4 Parameters

1. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyAPI.lib. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices. CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI Programmer's Reference).

2. CCyBulkEndPoint **BulkInEP
   This parameter is a double pointer to CCyBulkEndPoint. This parameter is used to pass the used BulkEndPoint parameter to TE_USB_FX2_GetData(). The double pointer is used because, if single pointer were used, the data modification of TE_USB_FX2_GetDataInstanceDriverBuffer() could not be passed over to TE_USB_FX2_GetData.()
3. PI_PipeNumber PipeNo
   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.

4. unsigned long Timeout

It is the integer time value in milliseconds assigned to the synchronous method XferData() of data transfer used by CyAPI.lib. Timeout is the time that is allowed to 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.

5. int BufferSize
   It is the dimension (in bytes) of the driver buffer (SW) used in data reception of a ***single endpoints (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 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). See 6 TE_USB_FX2_CyAPI.dll: Data Transfer Throughput Optimization.

3.5.5 Return Value

1. int : integer type

   This function returns true (ST_OK=0) if the selected BulkEndPoint exists in the firmware. This function returns false (ST_ERROR=1) otherwise.

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

3.6 *TE_USB_FX2_GetData()*

3.6.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_GetData(
  CCyBulkEndPoint **BulkInEP, byte* DataRead, long DataReadLength)
```

3.6.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2_GetData(&BulkInEP, DataRead, DataReadLength);

3.6.3 Description

This function takes an already initialized CCyBulkEndPoint double pointer. The device has been previously selected by TE_USB_FX2_Open(). TE_USB_FX2_GetData() reads data from the USB FX2 microcontroller endpoint EP6 (0x86) and transfers this data to the host computer. This data is generated by the FPGA.

3.6.4 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.

3.6.5 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.

3.6.6 *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_CyAPI.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 CyAPI 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.

3.6.7 Parameters

1. CCyBulkEndPoint **BulkInEP
   This parameter is used to pass to TE_USB_FX2_GetData() the parameter of BulkEndPoint used. This parameter is a double pointer to CCyBulkEndPoint. The double pointer is used because if single pointer is used the data modification of TE_USB_FX2_GetDataInstanceDriverBuffer() cannot be passed to TE_USB_FX2_GetData().
2. byte* DataRead

   C++ applications use directly TE_USB_FX2_CyAPI.dll based on CyAPI.lib. This parameter is passed by pointer to avoid copying back and forth large amount of data between these two DLLs. This parameter points the byte array that, after the function return, will contain the data read from the buffer EP6 of USB FX2 microcontroller. The data contained in EP6 is generated by the FPGA. If no data is contained in EP6, the byte array is left unchanged.
3. long DataReadLength

   This parameter is the length (in bytes) of the previous parameter.

3.6.8 Return Value

1. int: integer type

   This function returns true (ST_OK = 0) if it is able to receive the data from buffer EP6 within Timeout milliseconds. This function returns false (ST_ERROR = 1) otherwise.

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

3.7 *TE_USB_FX2_SetData_InstanceDriverBuffer()*

3.7.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_SetData_InstanceDriverBuffer(
  CCyUSBDevice *USBDeviceList, CCyBulkEndPoint **BulkOutEP,
PI_PipeNumber PipeNo,unsigned long Timeout, int BufferSize)
```

3.7.2 Function Call

Your application program shall call this function like this:

```
TE_USB_FX2_SetData_InstanceDriverBuffer (
 USBDeviceList, &BulkOutEP, PipeNo, Timeout, BufferSize);
```

3.7.3 Description

This function takes an already initialized USB device list (USBDevice previously selected by TE_USB_FX2_Open()) and a not initialized CCyBulkEndPoint double pointer, BulkOutEP. This function selects the endpoint to use: you shall choose EP8 (0x08) (endpoints EP4(0x04) or EP2(0x02) are also theoretically possible).
Currently (April 2012), only endpoint 0x08 is actually implemented in Trenz Electronic USB FPGA modules, so that endpoints EP2 and EP4 cannot be written or , more precisely, they are not even connected to the FPGA. That is why attempting to write them causes a function failure after Timeout expires.

TE_USB_FX2_SetData_InstanceDriverBuffer() function instantiates the class used by CyAPI to use Bulk EndPoint (CCyBulkEndPoint, see pages 9 to 11) and initializes the parameters of this class instantiation. The parameters are :

1. Timeout
2. XMODE_DIRECT (this parameter set the driver to single buffering, instead the slower double buffering)
3. DeviceDriverBufferSize.

The last parameter force the instantiation of the driver buffer (SW side, on the host computer) for the endpoint 0x86; this buffer has a size in byte given by DeviceDriverBufferSize. This value is of great importance because the data throughput is strongly influenced by this parameter (see section 6 TE_USB_FX2_CyAPI.dll: Data Transfer Throughput Optimization).

This function has not been included in TE_USB_FX2_SetData() for throughput reasons; if the driver buffer instantiation were repeated at every data reception, the data throughput would be halved. This function shall be used only one time to instantiate the driver buffer; after instantiation, TE_USB_FX2_SetData() can be used repeatedly without re-instantiating the driver buffer.

```
int TX_PACKET_LEN = 51200;//102400;

int packetlen = TX_PACKET_LEN;
unsigned int packets = 500;//1200;//1200;
unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;
unsigned long TIMEOUT= 18;
byte * data;
byte * data_temp = NULL;
unsigned int total_cnt = 0;
unsigned int errors = 0;
```

```
data = new byte [TX_PACKET_LEN*packets]; //allocate memory

PI_PipeNumber PipeNo = PI_EP8;

//starts test
SendFPGAcommand(USBDeviceList,FX22MB_REG0_START_RX);

CCyBulkEndPoint *BulkOutEP = NULL;

TE_USB_FX2_SetData_InstanceDriverBuffer (USBDeviceList,
&BulkOutEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);

ElapsedTime.Start(); //StopWatch start
for (unsigned int i = 0; i < packets; i++)
{
  packetlen = TX_PACKET_LEN;
  data_temp = &data[total_cnt];
  //cout << "Address &BulkInEP" << &BulkInEP << endl;
  //cout << "Address BulkInEP" << BulkInEP << endl;
  //cout << "Address *BulkInEP" << (*BulkInEP) << endl;
  if (TE_USB_FX2_SetData(&BulkOutEP, data_temp, packetlen))
  {
    cout << "ERROR read" << endl;
    errors++;
    break;
  }
  total_cnt += packetlen;
}
//DEBUG StopWatch
TheElapsedTime = ElapsedTime.Stop(false);

SendFPGAcommand(USBDeviceList,FX22MB_REG0_STOP);
```

3.7.4 Parameters

1. CCyUSBDevice *USBDeviceList

   CCyUSBDevice is a type defined in CyUSB.dll. Its name is misleading because it is not a class that represents a single USB device, but it rather represents a list of USB devices. CCyUSBDevice is the list of devices served by the CyUSB.sys driver (or a derivative like TE_USB_FX2_xx.sys). This parameter is passed by pointer. See page 7 and pages 23-49 of CyAPI.pdf (Cypress CyAPI Programmer's Reference).

2. CCyBulkEndPoint **BulkOutEP
   This parameter is a double pointer to CCyBulkEndPoint. This parameter is used to pass the used BulkEndPoint parameter to TE_USB_FX2_SetData(). The double pointer is used because, if single pointer were used, the data modification of TE_USB_FX2_SetDataInstanceDriverBuffer() could not be passed over to TE_USB_FX2_SetData.()

3. PI_PipeNumber PipeNo
   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.

4. unsigned long Timeout

It is the integer time value in milliseconds assigned to the synchronous method XferData() of data transfer used by CyAPI.lib. TimeOut is the time that is allowed to 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.

5. int BufferSize
   Itis the dimension (in bytes) of the driver buffer (SW) used in data transmission of a ***single endpoints (EP8 0x08 in this case);*** the total buffer size is the sum of BufferSize of every endpoint used. 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 read transactions to an actual value of 18 Mbyte/s). See 6 TE_USB_FX2_CyAPI.dll: Data Transfer Throughput Optimization.

3.7.5 Return Value

1. int : integer type

   This function returns true (ST_OK=0) if the selected BulkEndPoint exists in the firmware. This function returns false (ST_ERROR=1) otherwise.

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

3.8 TE_USB_FX2_SetData()

3.8.1 Declaration

```
TE_USB_FX2_CYAPI int TE_USB_FX2_SetData (
  CCyBulkEndPoint **BulkOutEP, byte* DataWrite, long
DataWriteLength)
```

3.8.2 Function Call

Your application program shall call this function like this:

TE_USB_FX2_SetData (&BulkOutEP, DataWrite ,DataWriteLength);

3.8.3 Description

This function takes an already initialized CCyBulkEndPoint double pointer. The device has been previously selected by TE_USB_FX2_Open(). TE_USB_FX2_SetData() reads data from the host computer and writes them to the USB FX2 microcontroller endpoint EP8 (0x08). This data is then passed over 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.

3.8.4 Expected Data Throughput

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 as normal.

The data throughput is variable in two ways:

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

3.8.5 DataWrite 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 (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.

3.8.6 DataWrite *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 (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_CyAPI.dll: Data Transfer Throughput Optimization.

3.8.7 Parameters

1. CCyBulkEndPoint **BulkOutEP
   This parameter is used to pass to TE_USB_FX2_SetData() the parameter of BulkEndPoint used. This parameter is a double pointer to CcyBulkEndPoint. The double pointer is used because if single pointer is used the data modification of TE_USB_FX2_SetDataInstanceDriverBuffer() cannot be passed to TE_USB_FX2_SetData().
2. byte* DataWrite

   C++ applications use directly TE_USB_FX2_CyAPI.dll based on CyAPI.lib. This parameter is passed by pointer to avoid copying back and forth large amount of data between these two DLLs. This parameter points the byte array that, after the function return, will contain the data written into buffer EP8 of USB FX2 microcontroller. The data contained in EP8 is generated by the host computer.
3. long DataWriteLength

   This parameter is the length (in bytes) of the previous parameter.

3.8.8 Return Value

1. int : integer type

   This function returns true (ST_OK = 0) if it is able to write the data to buffer EP8 within Timeout milliseconds. This function returns false (ST_ERROR = 1) otherwise.

```
enum ST_Status
{
     ST_OK = 0,
     ST_ERROR = 1
};
```

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:

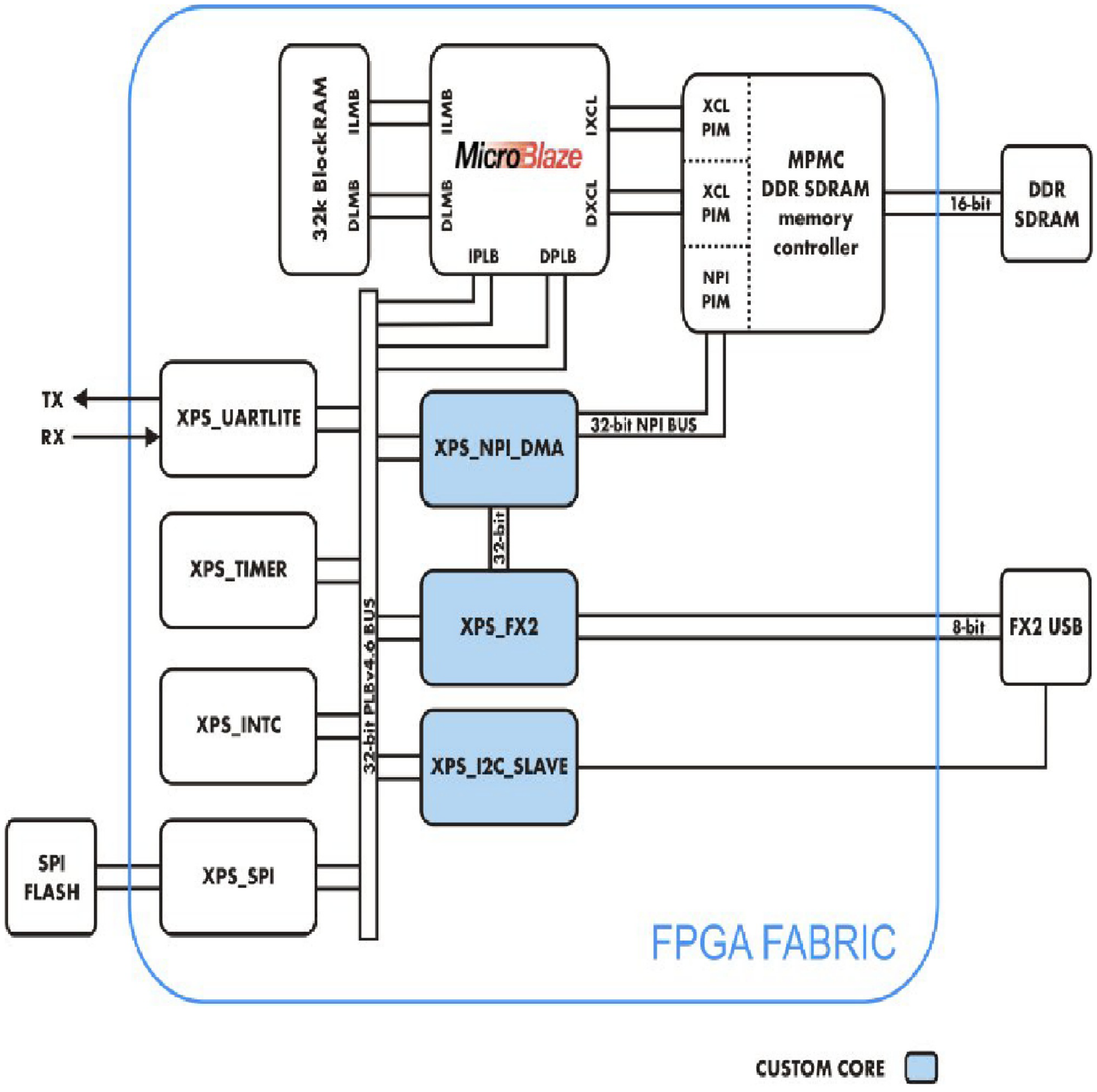
1. XPS_NPI_DMA
2. XPS_FX2
3. XPS_I2C_SLAVE

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_ADRESS	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:

```
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,
};

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

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] = I2C_WRITE;
Command[1] = MB_I2C_ADRESS;
Command[2] = I2C_BYTES;
```

```
Command[3] = 0;
Command[4] = 0;
Command[5] = 0;
Command[6] = 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_ADRESS=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 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 instruction 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_ADRESS=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 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.
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] = I2C_WRITE;
Command[1] = MB_I2C_ADRESS;
Command[2] = I2C_BYTES;
Command[3] = 0;
Command[4] = 0;
Command[5] = 0;
Command[6] = 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_ADRESS=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_ADRESS requesting I2C_BYTES

```
Command[0] = SET_INTERRUPT;
Command[1] = MB_I2C_ADRESS;
Command[2] = I2C_BYTES;

if (TE_USB_FX2_SendCommand(USBDeviceList, Command, CmdLength, Reply,
ReplyLength, 1000))
{
  cout << "Error" << endl;
  return -1;
  }


2)I2C_WRITE with MB_Command FX22MB_REG0_GETVERSION at byte 7

Command[0] = I2C_WRITE; //0xAD;//command I2C_WRITE
//Command[1] = MB_I2C_ADRESS;
//Command[2] = I2C_BYTES;
Command[3] = 0;
Command[4] = 0;
Command[5] = 0;
Command[6] = FX22MB_REG0_GETVERSION;//1; //get FPGA version

if (TE_USB_FX2_SendCommand(USBDeviceList, Command, CmdLength, Reply,
ReplyLength, 1000))
{
  cout << "Error" << endl;
  return -1;
}
```

3)GET_INTERRUPT

```
Command[0] = GET_INTERRUPT; //0xB1;//command GET_INTERRUPT

if (TE_USB_FX2_SendCommand(USBDeviceList, CardNo, cmd, cmd_len, reply,
ReplyLength, 1000))
{
  if ((ReplyLength > 4) && (Reply[0] != 0))
    {
      //Console.WriteLine("INT# : {0}", Reply[0]);
      printf("Major version: %d \n", Reply[1]);
      printf("Minor version: %d \n", Reply[2]);
      printf("Release version: %d \n", Reply[3]);
      printf("Build version: %d \n", 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_ADRESS=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_ADRESS=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)

```
int TX_PACKET_LEN = 51200;//102400;
int packetlen = TX_PACKET_LEN;
unsigned int packets = 500;//1200;//1200;
unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;
unsigned long TIMEOUT= 18;
byte * data;
byte * data_temp = NULL;
unsigned int total_cnt = 0;
unsigned int errors = 0;
bool printout= false;

data = new byte [TX_PACKET_LEN*packets]; //allocate memory

bool bResultDataRead = false;

unsigned int XferSizeRead = 0;

//Shortest and more portable way to select the Address using the
PipeNumber
PI_PipeNumber PipeNo = PI_EP6;
//starts test
SendFPGAcommand(USBDeviceList,FX22MB_REG0_START_TX);

CCyBulkEndPoint *BulkInEP = NULL;

TE_USB_FX2_GetData_InstanceDriverBuffer (USBDeviceList,
&BulkInEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);

//StopWatch start
ElapsedTime.Start();
for (unsigned int i = 0; i < packets; i++)
{
  packetlen = TX_PACKET_LEN;
  data_temp = &data[total_cnt];
  if (TE_USB_FX2_GetData(&BulkInEP, data_temp, packetlen))
  {
    cout << "ERROR read" << endl;
    errors++;
    break;
  }
  total_cnt += packetlen;
}

//DEBUG StopWatch timer
TheElapsedTime = ElapsedTime.Stop(false);
SendFPGAcommand(USBDeviceList,FX22MB_REG0_STOP);
```

4.3.3.2 *Combination 2 (simplified version)*

```
int TX_PACKET_LEN = 51200;//102400;
int packetlen = TX_PACKET_LEN;
unsigned int packets = 500;//1200;//1200;
unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;
unsigned long TIMEOUT= 18;
byte * data;
byte * data_temp = NULL;
unsigned int total_cnt = 0;
unsigned int errors = 0;
bool printout= false;

data = new byte [TX_PACKET_LEN*packets]; //allocate memory

bool bResultDataRead = false;

unsigned int XferSizeRead = 0;

//Shortest and more portable way to select the Address using the
PipeNumber
PI_PipeNumber PipeNo = PI_EP6;

CCyBulkEndPoint *BulkInEP = NULL;

TE_USB_FX2_GetData_InstanceDriverBuffer (USBDeviceList,
&BulkInEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);
 //StopWatch start
ElapsedTime.Start();
for (unsigned int i = 0; i < packets; i++)
{
  packetlen = TX_PACKET_LEN;
  data_temp = &data[total_cnt];
  //starts test
  SendFPGAcommand(USBDeviceList,FX22MB_REG0_START_TX);
  if (TE_USB_FX2_GetData(&BulkInEP, data_temp, packetlen))
  {
    cout << "ERROR read" << endl;
    errors++;
    break;
  }
  total_cnt += packetlen;
  SendFPGAcommand(USBDeviceList,FX22MB_REG0_STOP);
}
 //DEBUG StopWatch timer
TheElapsedTime = ElapsedTime.Stop(false);
```

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_ADRESS=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)

```
int TX_PACKET_LEN = 51200;//102400;
int packetlen = TX_PACKET_LEN;
unsigned int packets = 500; //1200;//1200;
unsigned long TIMEOUT = 1000;
byte * data;
byte * data_temp = NULL;
unsigned int total_cnt = 0;
unsigned int errors = 0;
double TheElapsedTime = 0;

PI_PipeNumber PipeNo = PI_EP8;

data = new byte [TX_PACKET_LEN*packets]; //allocate memory //

ResetFX2FifoStatus(USBDeviceList);
//starts test
SendFPGAcommand(USBDeviceList,FX22MB_REG0_START_RX);

bool bResultDataWrite = false;
byte PipeNoHex = 0x00;

unsigned int XferSizeRead=0;

unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;

// Find a second bulk OUT endpoint in the EndPoints[] array
CCyBulkEndPoint *BulkOutEP = NULL;

TE_USB_FX2_SetData_InstanceDriverBuffer (USBDeviceList,
&BulkOutEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);

ElapsedTime.Start(); //StopWatch start
total_cnt = 0;
for (unsigned int i = 0; i < packets; i++)
{
  long packetlen = RX_PACKET_LEN;
  data_temp = &data[total_cnt];

  if (TE_USB_FX2_SetData(&BulkOutEP, data_temp, packetlen))
  {
    cout << "ERROR" << endl;
    break;
  }
  total_cnt += packetlen;
}
//StopWatch timer
TheElapsedTime = ElapsedTime.Stop(false);
//stops test
SendFPGAcommand(USBDeviceList, FX22MB_REG0_STOP);
```

4.3.4.2 *Combination 2 (simplified version)*

```
int TX_PACKET_LEN = 51200;//102400;
int packetlen = TX_PACKET_LEN;
unsigned int packets = 500; //1200;//1200;
unsigned long TIMEOUT = 1000;
byte * data;
byte * data_temp = NULL;
unsigned int total_cnt = 0;
unsigned int errors = 0;
double TheElapsedTime = 0;

PI_PipeNumber PipeNo = PI_EP8;

data = new byte [TX_PACKET_LEN*packets]; //allocate memory //

ResetFX2FifoStatus(USBDeviceList);
//starts test

bool bResultDataWrite = false;
byte PipeNoHex = 0x00;

unsigned int XferSizeRead=0;

unsigned int DeviceDriverBufferSize = 131072;//409600;//131072;

// Find a second bulk OUT endpoint in the EndPoints[] array
CCyBulkEndPoint *BulkOutEP = NULL;

TE_USB_FX2_SetData_InstanceDriverBuffer (USBDeviceList,
&BulkOutEP, PipeNo, TIMEOUT, DeviceDriverBufferSize);

ElapsedTime.Start(); //StopWatch start
total_cnt = 0;
for (unsigned int i = 0; i < packets; i++)
{
  long packetlen = RX_PACKET_LEN;
  data_temp = &data[total_cnt];
  SendFPGAcommand(USBDeviceList, FX22MB_REG0_START_RX);
  if (TE_USB_FX2_SetData(&BulkOutEP, data_temp, packetlen))
  {
    cout << "ERROR" << endl;
    break;
  }
  SendFPGAcommand(USBDeviceList, FX22MB_REG0_STOP);
  total_cnt += packetlen;
}
//StopWatch timer
TheElapsedTime = ElapsedTime.Stop(false);
//stops test
```

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_ADRESS=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_ADRESS=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_ADRESS=0x3F
3	0x0C (12)	FX2_Parameters.I2C_BYTES=0x0C Number of bytes to write (max 32)
4	0x00	MB_Commands.FX22MB_REG0_NOP
5	0x00	MB_Commands.FX22MB_REG0_NOP
6	0x00	MB_Commands.FX22MB_REG0_NOP
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 (`USBDeviceList`) initialized to null of the class CCyUSBDevice
2. reads and displays the number of Trenz Electronic modules (TE_USB_FX2_ScanCards())
3. selects the first (0) Trenz Electronic module;
   an handle to the device driver is open but not exposed to the user
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;
   the previous handle is automatically closed and another handle (this time associated with the new Trenz Electronic module) to the device driver is opened but not exposed to the user. If the selected module is not attached to the USB bus, the previous handle is automatically closed but a new one is not opened.
7. reads the FPGA firmware version of the second (1) module

```
//This line creates a list of USB device that are used through
//Cypress driver
CCyUSBDevice *USBDeviceList = new
CCyUSBDevice((HANDLE)0,CYUSBDRV_GUID,true);

unsigned long Timeout = 1000;

int  NumberOfCardAttached = TE_USB_FX2_ScanCards(USBDeviceList);
cout << endl << NumberOfCardAttached << endl;

//If you want use the first Trenz Electronic module use 0
if (TE_USB_FX2_Open(USBDeviceList, 0)==0)
  cout << "Module is connected!"  <<endl;
else
  cout << "Module is not connected!" <<endl;

//USBDevice->Open(CypressDeviceNumber); it is used in
//TE_USB_FX2_Open()
int vID = USBDeviceList->VendorID;
int pID = USBDeviceList->ProductID;
cout << "VID" << vID << endl;
cout << "PID" << pID << endl;

byte Command[64], Reply[64];
long CmdLength = 64;
long ReplyLength = 64;

Command[0] = 0x00;//comand read FX2 version

if (!TE_USB_FX2_SendCommand(USBDevicelist, Command, CmdLength,
Reply, ReplyLength, Timeout))
{
  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]);
  }
}
else
  cout << "Error" << endl;

//If you want use the first Trenz Electronic module use 1
if (TE_USB_FX2_Open(USBDeviceList, 1)==0)
  cout << "Module is connected!"  <<endl;
else
  cout << "Module is not connected!" <<endl;

byte Command 1[64], Reply1[64];
long CmdLength1 = 64;
long ReplyLength1 = 64;

Command1[0] = SET_INTERRUPT; //0xB0;//comand SET_INTERRUPT
Command1[1] = MB_I2C_ADRESS; //0x3F;//I2C slave address
Command1[2] = I2C_BYTES;//12;//12 bytes payload

if (TE_USB_FX2_SendCommand(USBDeviceList, Command1, CmdLength1,
Reply1, ReplyLength1, Timeout))
  cout << "Error Send Command SET INTERRUPT" << endl;

Command1[0] = I2C_WRITE; //0xAD;//comand I2C_WRITE
//Command1[1] = MB_I2C_ADRESS; //0x3F;//I2C slave address
//Command1[2] = I2C_BYTES; //12 bytes payload
Command1[3] = 0;
Command1[4] = 0;
Command1[5] = 0;
Command1[6] = FX22MB_REG0_GETVERSION; //1; //get FPGA version

if (TE_USB_FX2_SendCommand(USBDeviceList, Command1, CmdLength1,
Reply1, ReplyLength1, Timeout))
  cout << "Error Send Command Get FPGA Version" << endl;

  Command[0] = GET_INTERRUPT; //0xB1;//comand GET_INTERRUPT

if (!TE_USB_FX2_SendCommand(USBDeviceList, Command1, CmdLength1,
Reply1, ReplyLength1, Timeout))
{
  if ((ReplyLength1 > 4) &&(Reply1[0] != 0))
  {
    printf("Major version: %d \n", Reply1[1]);
    printf("Minor version: %d \n", Reply1[2]);
    printf("Release version: %d \n", Reply1[3]);
    printf("Build version: %d \n", Reply1[4]);
  }
}
else
  cout << "Error, GET INTERRUPT" << endl;
```

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_CyAPI.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)	15.6
8,192	20.4
16,384	26.3
32,768	30.2
65,536	34.1
131,072	36.2
262,144	36.7

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

To change XferSize in C++, the method

```
BulkEndPoint->SetXferSize(DesiredValue);
```

shall be used. Cypress sets DesiredValue to 4,096 bytes by default. This default value is not documented in the CyAPI.lib manual (pag 62 of CyAPI.pdf), but it has been retrieved by using the following C++ instructions:

```
int XferSizeReadValue = BulkEndPoint->GetXferSize();
cout<< "XferSizeReadValue" << XferSizeReadValue <<endl;
```

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	2.32
1,024	4.25
2,048	7.65
4,096	15.22
8,192	20.35
16,384	25.45
32,768	31.05
65,536	35.34
131,072	37.01

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

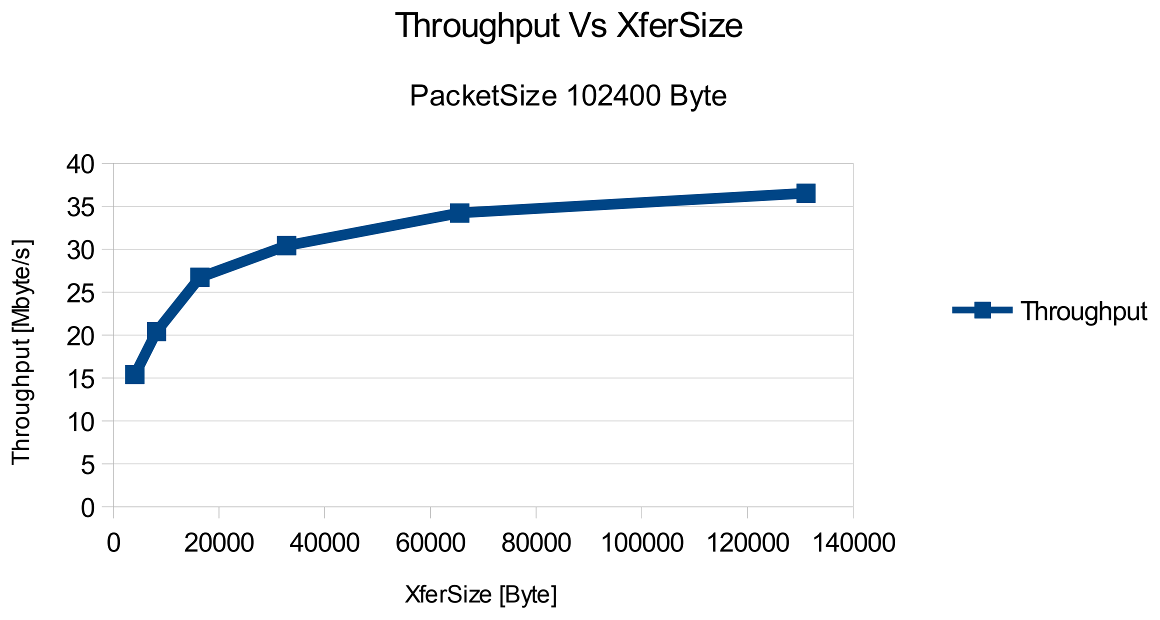


Chart 1: data throughput [Mbyte/s] as a function of XferSize [byte] given PacketSize = 102,400 bytes.

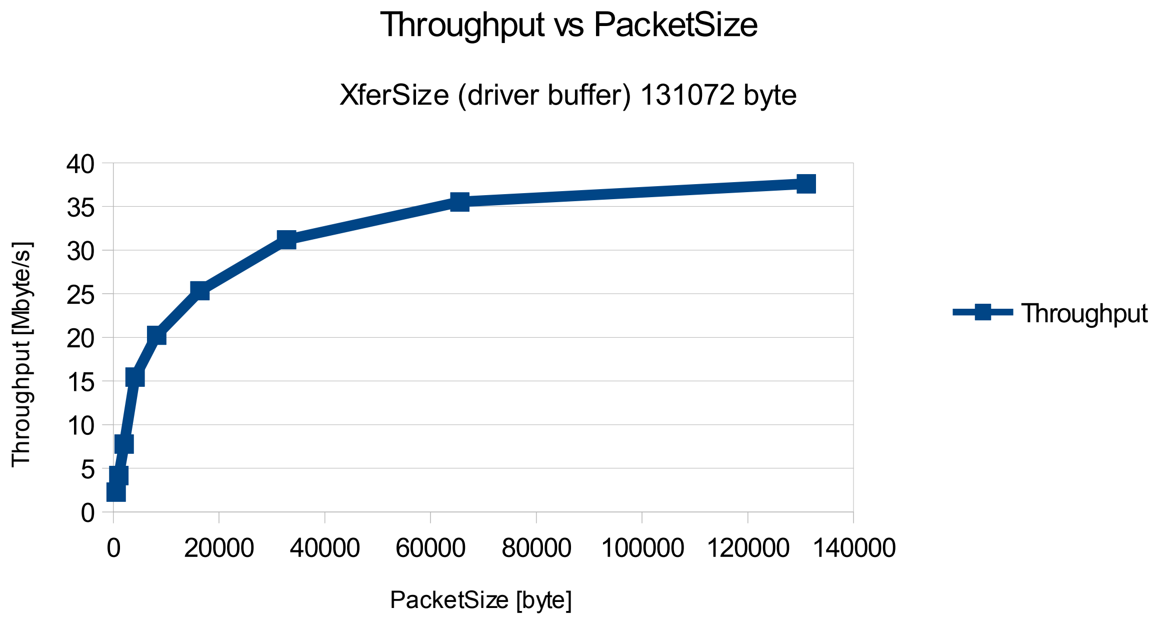


Chart 2: data throughput [Mbyte/s] as a function of PacketSize [byte] given XferSize = 102,400 bytes.

7 Appendix A : Open the Visual Studio 2010 project

The Visual Studio project file *.sln
(TE_USB_FX2_CyUSB_SampleApplication.sln and TE_USB_FX2_CyUSB.sln)
can be open

1. using right click ;
2. and then select “Open with”;
3. and then select “Microsoft Visual C++ 2010 Express” or “Microsoft Visual Studio 2010” (the latter is used if Visual Studio 2010 Professional is installed).

If Visual Studio 2010 Express is used to compile 64 bit C++ programs, Microsoft Windows SDK 7.1 must be installed after the installation of Visual Studio 2010 Express.

After the project file is open you must select the correct parameter (in particular if you use the version of the code from GitHub instead of precompiled software project or create a new softwareproject) for the 32 and 64 bit case.

You must follow this procedure:

1. Open the project
2. wait the end of the parsing (it is shown at lower left with a white “Ready”
3. right-click “Solution 'TE_USB_FX2_CyAPI_SampleApplication'” under “Solution Explorer”
4. a new window pop up (“Solution 'TE_USB_FX2_CyAPI_SampleApplication'” Property Pages”)
5. select “Configuration Properties”
6. left-click “Configuration Manager...”
7. a new window pop up (“Configuration Manager”)
8. for “Active solution configuration” select “Release”
9. for “Active solution platform” select “Win32” (“x64” for 64 bit case)
   If “x64” does not exist you must create this option with <Edit>
10. if are not already selected in the table, select “Release” for “Configuration” and “Win32” for “Platform” (Build must also selected with a “v” shown);
11. left click “Close”
12. the window “Configuration Manager” is closed
13. verify that “Win32” (“x64”) is selected for “Platform”
14. verify that “Release” is selected for “Configuration”
15. in the window “Solution 'TE_USB_FX2_CyAPI_SampleApplication'” Property Pages” select “Apply” and then “Ok”
16. the window “Solution 'TE_USB_FX2_CyAPI_SampleApplication'” Property Pages” is closed
17. right-click “TE_USB_FX2_CyAPI_SampleApplication” under “Solution Explorer”
18. select “Configuration Properties” then “General”
19. a)“Platform Toolset” must be selected “v100” for 32 bit (both Express and Professional) and for 64 bit professional.

b)“Platform Toolset” must be selected “Windows7.1SDK” for 64 bit Express

20. select “Configuration Properties” then “C/C++”, then “Preprocessor”
21. select “Preprocessor Definitions” must be left clicked,
22. left click the black arrow pointing toward the bottom and then select <Edit>
23. a new window pop up (“Preprocessor Definitions”)
24. add “WIN32” and then click return,
25. add “NDEBUG” and then click return
26. add “_CONSOLE” and then click return
27. select “OK”
28. the window “Preprocessor Definitions” is closed
29. select “Configuration Properties” then “C/C++”, then “Linker”
30. select “Input”, then “Ignore specific default libraries”
31. left click the black arrow pointing toward the bottom and then select <Edit>
32. a new window pop up (“Ignore Specific Default Libraries”)
33. add “libcmt.lib” and then click return
34. select “OK”
35. select “Input”, then “Additional Dependencies”
36. left click the black arrow pointing toward the bottom and then select <Edit>
37. a new window pop up (“Additional Dependencies”)
38. add “setupapi.lib” and then click return
39. add “CyApi.lib” and then click return
40. select “OK”
41. select “debugging”, then “Generate Debug”
42. left click the black arrow pointing toward the bottom and then select “Yes(/DEBUG)”
43. click “Apply” and then “OK”.

8 Appendix B : use of pdb file (symbolic debugging)

You can choose to use the pdb file for a debugging based on symbol.

If you compile using pdb file, the compilation is more slow.

If you want deactivate the service or change the directory of pdb files used you must follow this procedure:

1. select “Debug” in the project open
2. select “Options and Settings...” in the list open
3. a new window pop up (“Options”)
4. select “Debugging” then “Symbols”
5. select the “Symbol file (.pdb) locations:” you can choose “Microsoft Symbol Server” or a directory of your choice or nothing (in this latter case, in step of compilation you are informed that Visual Studio is unable to charge the symbols of various DLLs

9 Document Change History

version	date	aut hor	description
0.9	2012-06-01	SP, FDR	Release Preview.
1.0	2012-06-14	SP, FDR	Initial release.
1.1	2012-09-07	SP, FDR	Typo corrected (CyAPI.lib (right) in place of CyAPI.dll (wrong)). Added a clarification in Section 3 at step 13 and 15 about the files necessary for the project compilation.
1.2	2013-04-05	FDR	Improved "Hardware, firmware and software stack" table.

10 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] Introduction to CyAPI.lib Based Application Development Using VC++ March 3, 2011 Document No. 001-61744 Rev. *C 1 , AN61744

http://www.cypress.com/?rID=43538

[3] Cypress CyAPI Programmer's Reference, 2010 Cypress Semiconductor
more recent version inside Cypress Suite USB 3.4.7

http://cosmiac.ece.unm.edu/images/5/50/CyAPI.pdf

[4] Cypress CyUsb.sys Programmer's Reference

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

[5] Plug And Play

http://www.cypress.com/?id=4&rID=37592