TNT 2 Digital acquisition cards

Communicating with a TNT card

Sender :	Marc RICHER
Writer :	Marc RICHER
Created	11 april 2003
Updated	09 September 2005
Document reference	GTI/PHYNUC/TNT2_DOC
Document version	1.0
Document state	Work in progress
Document name	CardCommunication.doc
Number of pages	23
Related pieces	0

SUMMARY

1 REFERENCE DOCUMENTS....................2

2 OBJECT....................2

3 GETTING STARTED....................3

4 COMMUNICATION WITH A TNT CARD....................3

4.1 Presentation of the card....................3
4.2 Some USB communication elements....................7
4.3 Communication model....................9
4.4 Particular USB informations & codes....................14
4.5 Communication examples....................17
4.6 USB host programming....................20
4.7 Tricks & tips....................21

1 Reference documents

- [1] *'Digital Signal Processing for segmented HPGe Detectors Preprocessing Algorithms and Pulse Shape Analysis'*, Dissertation submitted to the Combined Faculties for the Natural Sciences and for Mathematics of the Ruperto-Carola University of Heidelberg, Germany for the degree of Doctor of Natural Sciences, Martin Lauer.

- [2] '*Digital synthesis of pulse shapes in real time for high resolution radiation spectroscopy*' Jordanov, Knoll. Nuclear Instruments & methods in Physics Research. A345 (1994) 337-345.

- [3] *Benjamin Carniol Ph. Thesis*

- [4] *Strahinja Lukic Ph. Thesis*

- [5] *Paper de Garcia*

- [6] *'Universal Serial Bus Specification'* Revision 2.0 from the 27 April 2000. http://www.usb.org

- [7] Documentation of the Cypress USB2 development kit CY3681

- [8] ' USB in a Nutshell ' - http://www.beyondlogic.org

2 Object

Working around the « Tracking gamma – AGATA » project and the Grace-NTOF project the development of a new digital acquisition card has been started at IReS - IN2P3 at the end of the year 2001. The TNT card (Treatment for Numerical Tracking or Treatment for NTOF) came out of development phase during 2003.

This card is controlled from a PC through USB2 (Universal Serial Bus, version 2).
The TNT card has 2 channel sampled at 65 Mhz and a FPGA in order to process sampled data and accomplish communication tasks.
During 2004 a second card version has been designed (TNT2) which has some improvements : 4 channel sampled up to 100 Mhz and 2 FPGA on board.

The aim of this document is to present

- the general communication model that has been designed for the TNT2 (inspired by the one used for the TNT1) and how it is relayed at USB level,
- the fonctionnalities (from a "user" point of view) of a TNT2 card ,
- an example of a control & readout software for all TNT cards: the TUC (Tnt Usb Control) software.

Chapter 4 contain some VHDL specific description but for more hardware documentation, see the document written by Patrice MEDINA : "TNT2 hardware description".

3 Getting started

Basically, the aim of all TNT card models is to digitize some input signal (up to 4 inputs) at some sampling period (up to 100 Mhz) with 14 bits per sampled points and to achieve some processing based on these sampled data. Processing is accomplished by FPGA components which have some external memory ressources. The communication with a card is done via USB/USB2 (and/or Ethernet network in case of TNT2-DJ).

The first main operation mode, the oscilloscope mode, consist to memorize some amount of sampled points for each input channel (an 'oscillogram') and to readout them to the control PC. Some triggering shemes can be set up in order to define some event selection criteria.

For the second main operation mode, the energy mode, the context is to process some detector output (mainly Ge detectors) which are used in nuclear physics experiments. When some physics events occurs, a side effect is the apparition of gamma-rays which interact with detectors. The result of this interaction is the appearance of voltage impulses at the detector's outputs. The height of these impulses is proportional to the amount of the gamma-ray energy released during the interaction with the detector. Knowing precisely the interaction energy is the key in order to start some analysis (constitution of energy spectra,...) about what occured during the physic experiment which is part of physicist research work.
At any time, when some impulse comes over a card input, the card will construct some associated trapezoidal shape whose flat top height is proportional to the initial impulse height. The flat top height value is called 'energy'.
Each time a trigger occurs on a card channel, the card will construct a so called 'energy event' or 'multiparametric event' or 'information tuple' :
Channel number, trigger counter, timestamp and energy value
The successive 'energy events' are sent to the PC in order to save them to file and/or to construct some energy histograms.

4 Communication with a TNT card

4.1 Presentation of the card

The block diagram on next page presents a simple overall view of the TNT2 acquisition card.

PCM56
5-20 Mev
5-20 Mev
5-20 Mev
AD6645
Buffers
Channel 1

PCM56
5-20 Mev
5-20 Mev
5-20 Mev
AD6645
Buffers
Channel 2

PCM56
5-20 Mev
5-20 Mev
5-20 Mev
AD6645
Buffers
Channel 3

PCM56
5-20 Mev
5-20 Mev
5-20 Mev
AD6645
Buffers
Channel 4

4 Logical Inputs
Nim/LVTTL
Nim/LVTTL
Nim/LVTTL
Nim/LVTTL
LVTTL/NIM
LVTTL/NIM
LVTTL/NIM
LVTTL/NIM
4 Logical Outputs

4 LVDS Inputs
4 LVDS Outputs

LVTTL/NIM
LVTTL/NIM
LVTTL/NIM
LVTTL/NIM
4 Local Trigger

Memory 1
512k x 64 bits
Memory 2
512k x 64 bits
Memory 3
512k x 32bits

VIRTEX II
3000 FG656

Memory
Memory
Memory
XF04
LOADER
SPARTAN2
Memory
Memory

AD9767
Analog Inpections

4 LOGICALS
INSPECTIONS

LVTTL/NIM
Sel
Q
Vir
Clock Selector

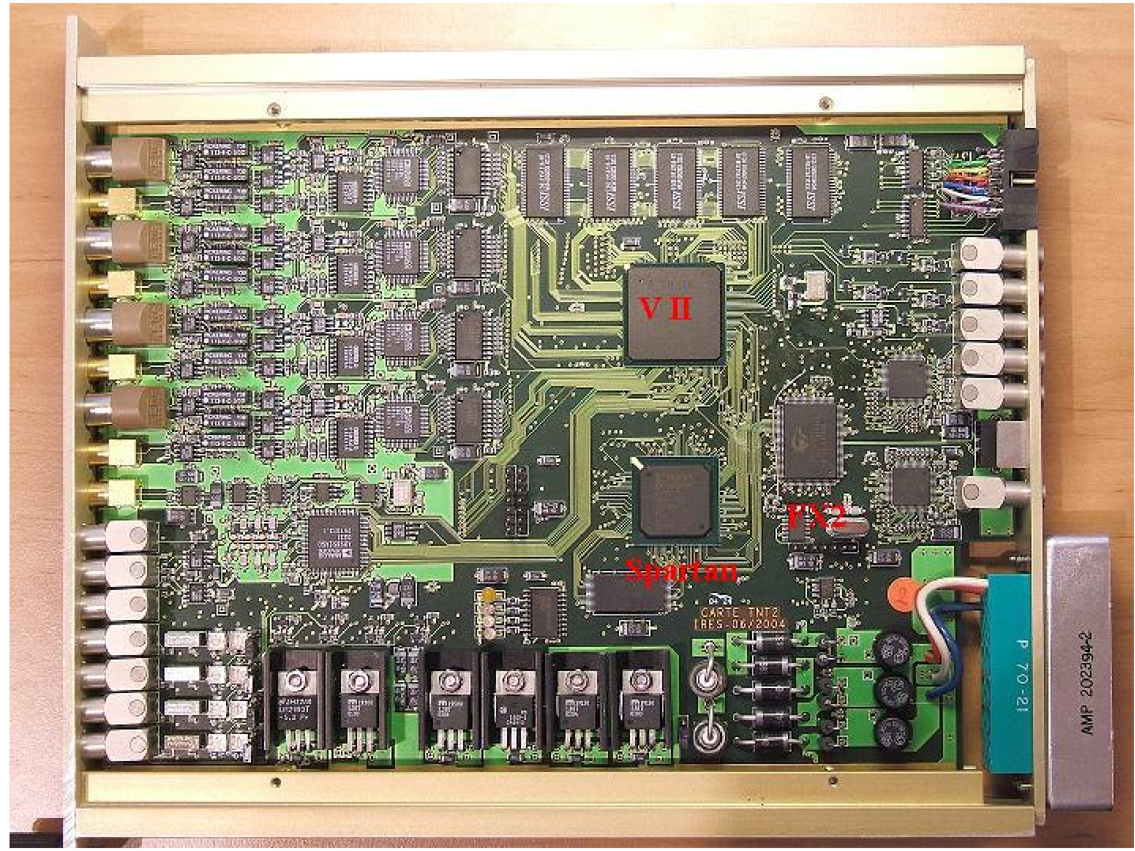
DDS
AD9852
Clock

CY680013
FX2
USB2 Interface

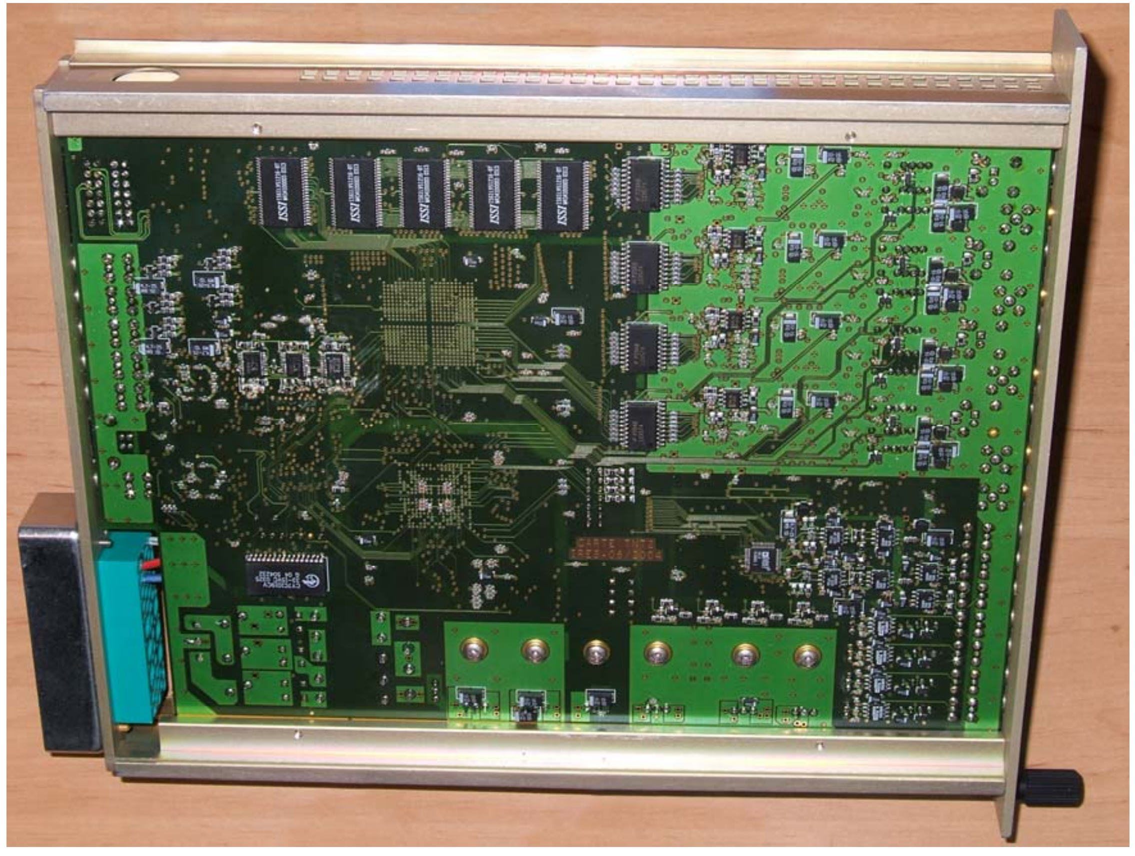
We distinguish these components:

- Four input channels sampled by ADC's whose default frequency is 100 MHz. Each channel has a commun and a differential input
- Four logical NIM inputs and four logical NIM outputs. Some gate and/or delay can be applied to each of them
- Four LVDS output and four LVDS input
- Two analogical inspection outputs
- Four logical inspections outputs
- A Direct Digital Synthesis –DDS for constructing some clock signal and a clock selector allowing to use different signal sources
    - from a 48 Mhz signal from the virtex2
    - from a 20 mhz quartz
    - from a special dedicated NIM input
    - from a special dedicated LVDS input
- The FX2 CY68013 from Cypress for the USB2 interface. This chip act like a kind of bridge between the FPGA's and the USB bus. It implement different mechanisms in order to
    - Receive USB data from the PC host and deposit it in a specific FIFO. The chip handles the USB low level automatically : USB frames are received, tokens and header of packets analysed, payload data is extracted, CRC is done, and some acknowledgment is sent back to the Host.
    - Receive some data from the FPGA's that is ready for a transfer to the PC. Up to four packet, with a maximum size of 512 bytes (64 in case of USB1) are available for the FPGA's.
    - Handle the standard USB requests over control transfers that every USB compliant device must support. Those requests takes place always on Endpoint n°0, and are codified on one byte. Two main groups are defined :
        - 'Standard Device request' (ex: "GET_STATUS", "GET_DESCRIPTOR",..). These requests are strictly defined in the official USB specifications, every device has to handle them coherently.
        - 'Vendor specific request' : free to use as needed for specific purposes.

  At power on reset the FX2 load and execute some software code stored in an I2C EEPROM. If no valid EEPROM content is found, the FX2 is able to supply some native minimalist USB parameters and actions that enable a connections on the bus and allow some software download via the USB. Some specific software can erase, read or write to the EEPROM.
- Two FPGA's which can communicate with the FX2 :
    - A Virtex II which is in charge of managing all I/O, calculation and memory access
        - At power on, the Virtex II load and execute some software stored in 3 EEPROM accessible via the JTAG interface.
        - It can also reboot on a flash EEPROM which is accessible through the USB via the Spartan
    - A Spartan which is in charge of rebooting the Virtex II and controlling some parameters. He can also access to the flash EEPROM of the Virtex II
- The mechanical dimension of the TNT card fits in a slot of a NIM rack.
- For debugging purposes, four LED's are accessible on the card.



V II
FX2
Spartan
CARTE TNT2
IRES-06/2004
P 70-21
AMP 202394-2



Front face		Rear face	
	4 input channel sampled at 100 Mhz over 14 bits. Common and differential input for each one.	4 LVDS outputs and 4 LVDS inputs	
		USB connector	
	External clock input		
	2 analogical inspection outputs 4 logical inspection outputs		
		Alimentation block which has to be connected on a NIM rack.	
	4 NIM inputs , 4 NIM outputs		

4.2 Some USB communication elements

- The universal serial bus USB has a star topology : the host PC in the middle and all USB connected device around. Thus the master which is the PC has an USB controler implementation which is of type UHCI - Universal Host Controller Interface or OHCI-Open Host Controller Interface in USB1 and EHCI-Enhanced Host Controller Interface) in USB2.

- All communications are initiated by the bus master, the PC. Communications which transfer data from the PC to the device are called « OUT communication», those transfering data from the device to the PC are « IN communication».

- The USB2.0 communication protocol has 3 been designed with different theorical bandwidth :
  - ➢ 480 Mbits/s in « High speed » mode ➔ 60 Mo/s
  - ➢ 12 Mbits/s in « Full speed » mode ➔ 1,5 Mo/s
  - ➢ 1,5 Mbits/s in « Low speed » mode ➔ 192 Ko/s

  TNT cards operate mainly in high speed mode. 480 Mbits is the theorical upper limit of transfer capability which include a lot of informations needed for the low level USB protocol. By considering only payload data, in high speed, the theorical maximum transfer rate is about 53 MB/s.

- Up to 127 devices can be connected on the same bus. A USB cable cannot exceed 5 meters (but some special repeater cable exists).

- The plug&play feature of USB means that a connection / disconnection to/from the bus is detected and handled by the bus host PC. Once connected a USB device send a couple of identifiers to the bus master : the « Vendor Identification » (VID) and the « Product identification » (PID) numbers. Thanks to these identifiers, the host know how to handle this new connected device and may load some associated driver. The master allocate a bus adress to the device thus becoming operationnal on the bus. For 'classical' devices referring to some predefined class of devices (mouse, a keyboard, USB key,...) the operating system has in most cases some appropriate driver ready to load. For some specific devices (scanner, camera, a TNT card), when first connected on the bus, the host may ask the user for some driver files which are essential in this case.

- From a logical point of view, a USB device present some 'endpoints' which are sources or sinks of communication streams. Endpoints are identified with a number, starting at 0, with a direction (IN or OUT), a communication type (bulk, isochronous, interrupt or control). By sending or receiving data between host endpoint and device endpoint, some logical connection is defined and called 'pipe' which is also identified by some sequence number.

- Four different communication types exists in the USB, sorted in two groups :
  - The first group that contain only the '**Control'** communication type is for message transfers. It's purpose is to support control, configuration and parameter settings. By default, every device must be able to handle some request over his endpoint 0 :
    - USB 'Standard Device requests'
      - GET_STATUS
      - CLEAR
      - SET_FEATURE
      - GET_DESCRIPTOR and SET_DESCRIPTOR
      - GET_CONFIGURATION and SET_ CONFIGURATION
      - GET_INTERFACE and SET_ INTERFACE
      - SET_ADRESS
      - SYNCH_FRAME

      These request are defined in the USB specifications and must be handled correctly by every device.
    - 'Vendor specific request' : USB device designers can implement here some specific request for ther needs.
  - For data transfers
    - 'Bulk' data transfers : the integrity of data is guaranteed, (data will ber e-sent if some error is detected) but not the transfer rate which can change during the transfer. This kind of transfer is typically used for print services.
    - 'Isochronous' data transfers : the data rate is guaranteed but data may be lost or corrupted. No retry action is made in cas of error detection. Some bandwidth is reserved by the bus controler for such transfers. This kind of transfer is typically used for multimédia needs (video, music,..) where having some constant data flow is crucial.
    - 'Interrupt' data transfers: on a regular basis, the host ask the device if some data has to be transferred. It's a kind of polling strategy : USB mousse and keyboards typically uses these transfer types.

  - At lowest level, every transfer is made of frames : data are send or received by packets, on a regular basis establishing some transaction feature which involve these kinds of packets :
    - Token packets which contain the destination adress, the transfer direction and may be some data

- Eventualy some data packets : one or more data packets encapsulating the payload data to transmit. Depending of the type of transfer and of the USB mode choosen, the data packet size can change. For example, for bulk transfers, the maximum packet size is 512 bytes for USB2.
- Status packet which is used for acquitment of a transfer and for error detection.

This low level is completely and automatically handled by the FX2 chip of Cypress.

Frames rates is about 1ms. In high speed, there are some micro-frames every 125µs. Within a micro-frame, it is theorically possible to carry up to 13 bulk data packet, which gives in one second :

(13 * 512) / 125µs = 53,24 MB of payload data / s.

4.3 Communication model

The following diagram present the four involved actors that has to be in relation together in order to establish a communication with an acquisition card:

Communication model between PC and card 1/2

Digital card

HOST PC

FX2

VIRTEX II

HI-SPEED CERTIFIED USB

C++, TCL, VB, Java,..

1,7 Ghz

Keil C / assembler

48 Mhz

VHDL

48-360 Mhz

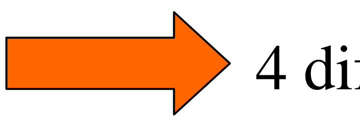
SPARTAN

VHDL

48-360 Mhz

GUI

FX2 "Slave FIFO" feature
2*8 bits ports I/O
+ 5 bits CTL



4 different softwares!

For a communication between the acquisition card and the supervision station four different softwares has to be to designed and needs to communicate together :

- **The control and supervision software** implement some GUI screens in order to present all the parameters values of the card, elaborate some orders that have to be sent to the card (new parameters values, start of an acquisition, readout of parameters values), and receive acquisition data send by the card in order to visualize it or/and to store it on disk. This software can be made with any kind of programming tools / language that is able to manage interactive GUI and make some call to external libraries (DLL files on windows platform) or some call to the kernel (Linux case)

- **The FX2 software** that define and initialize the appropriate configuration at USB level and then start the chosen mechanism / mode for interfacing with outer world (Virtex II and Spartan). It will define some pipe associated with a specific endpoint, choose a transfer type for those pipes, and some specific vendor request in order to get some information's about what happens in the FX2…Basically this software is in charge of relaying communication between the host PC and the card. It has to be written in a Keil C language which is dedicated for the 8051 micro controller in the FX2 component. Under windows operating systems, this software can be directly downloaded via USB with the 'Control Panel'software shipped with the FX2 development kit.

- **The Virtex II software** communicate with the FX2, drive the ADC's, store some data, communicate with other devices / card or components and manage the majors parmaters of the card. It is written in VHDL, compiled and then downloaded
  - through a JTAG connector into 3 serial connected EEPROM's
  - through USB to the Spartan and then to a flash EEPROM.

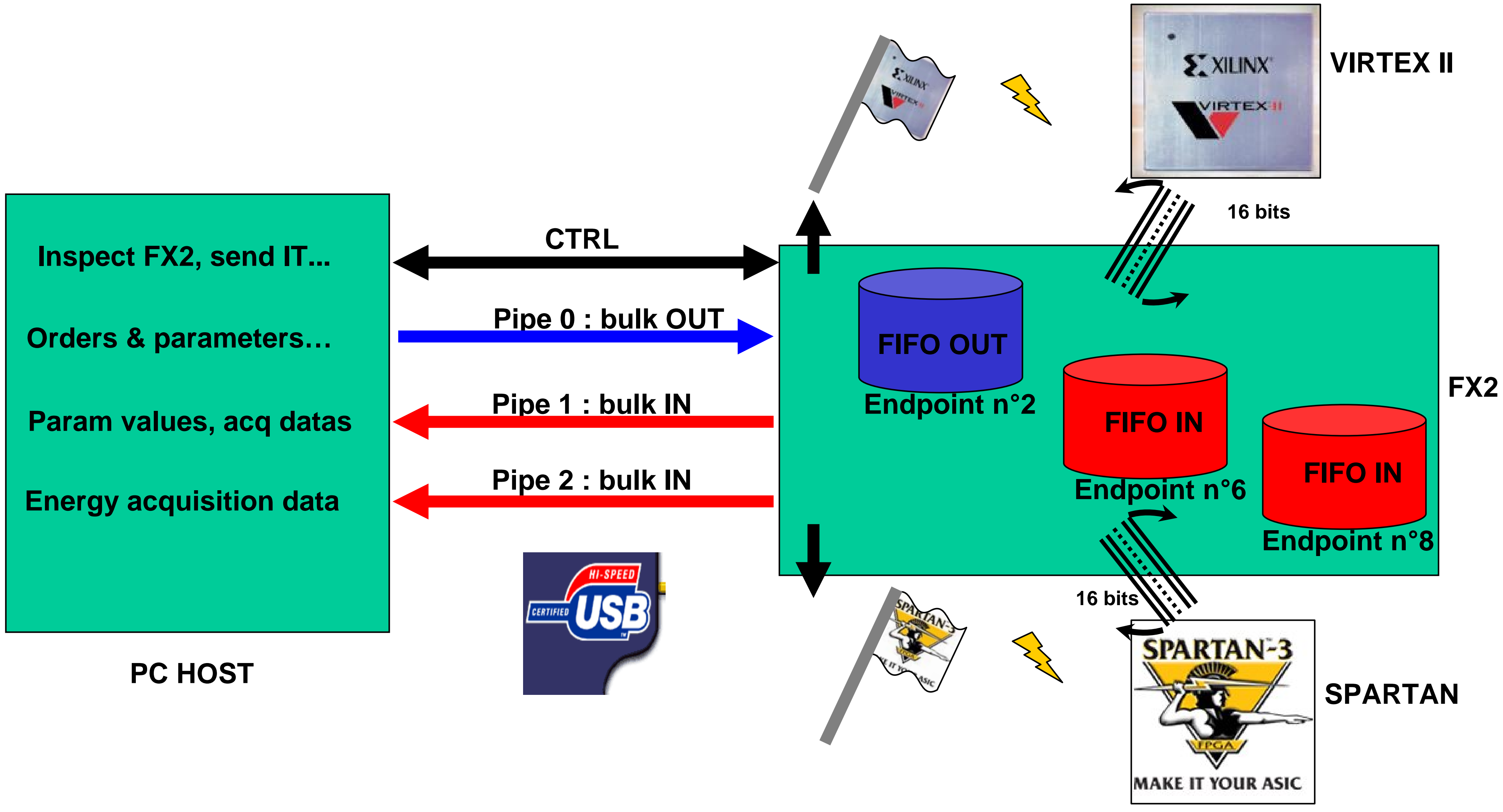
- **The Spartan software** communicate also with the FX2, and is in charge of reading / writing the flash EEPROM of the Virtex II, make reboot the Virtex II on the 3 serial EEPROM's or on the flash EEPROM and manage some minor parameters on the card.

The two FPGA needs to communicate with the same component, thus thay have to share the same communication lines through the FX2.
The USB dedicated component propose a simple I/O mechanism called **'Slave FIFO'** that has been choosen for the TNT communication implementation :

- For every defined USB 'endpoind', the FX2 present some associated FIFO
  - If the 'endpoint' is OUT defined, data send by the PC to the card is placed into a FIFO
  - If the 'endpoint' is IN defined, FPGA's on the card can write data to the PC directly in the FIFO
- FX2 automatically transfer payload data from the USB to the OUT FIFO, by doing all USB low level protocol tasks
- FX2 automatically transfer data from the IN FIFO to the USB as soon as some amount of data is written or is explicitly 'ready for transfer' committed (see later), by doing all USB low level protocol tasks
- FX2 has 25 pins/bits for handling access these FIFO's :
  - 2 pins/bits for selecting a precise FIFO number
  - 3 state flag for the current selected FIFO : FIFO full, FIFO empty or FIFO at a specific level
  - 4 bits in order to implement the protocol for reading out or writing in some bytes of data (this protocol is explained in the Cypress documentation)
  - 16 pins/bits for reading / writing some data in a FIFO (16 bits data bus)

By using this 'Slave FIFO' feature of the FX2, this communication model has been implemented which covers all communication needs to a TNT card :

Communication model between PC and card 2/2



The FX2 software defines and configures three USB communication channels:

- Pipe n°0 with endpoint n°2 that is of type BULK OUT, with a size of 1536 bytes (up to three 512 bytes packets)
- Pipe n°1, with endpoint n°6 that is of type BULK IN, with a of size 1536 bytes (up to three 512 bytes packets)
- Pipe n°2 with endpoint n°8 that is of type BULK IN, with a size of 1024 bytes (up to two 512 bytes packets)

In the TNT communication model, some orders / parameters are only for the Spartan, some orders / parameters are designed only for the Virtex : when addressing something to the card, there is always only one receiver on the card : the Spartan or the Virtex II but never both.

The endpoint n°0 /Endpoint n°2 is used **for sending orders and parameters to the card**. All the different parameters / orders are encoded in a specific way (see Order & parameters encoding), thus a small amount of bytes is send by the host over USB, using the pipe n°0, and then put down in the associated FIFO by the FX2. By doing this, the empty and full flag of this FIFO's are updated.

In order to inform the dedicated receiver of these orders / parameters (Spartan or Virtex II) that some information is available, ready to be read and analysed, some control transfer using specific vendor requests are send over **endpoint 0** to the FX2 by using an USB 'control' communication.
The FX2 identifies the code of the request and generate some signal to the Spartan or to the Virtex II. This will act as raising something like a 'flag' to one of the FPGA's, in order to alert it.

Some process in the FPGA's is always looking for his 'flag'. Each of the FPGA's has its own 'flag' designed. As soon as one FPGA's recognize his flag being raised, he react in an interrupt like manner : he **immediately stop any activity in progress** and stay in a interrupt state, waiting for the lowering of his flag.

By sending another specific Vendor request to the FX2, he will now lower the flag that has been raised just before. As in some kind of a 'release' state, the addressed FPGA now connects to the endpoint n°2 FIFO OUT buffer and starts to read out all bytes (until the empty flag has been raised by the FX2).

The FPGA analyse the orders / parameters received and proceed to the appropriate task.

If any data / information has **to be send back to the host PC via USB**, the FPGA switch the FX2 data bus to the FIFO IN buffer which is associated with pipe n°1, endpoint 6 BULK IN and simply write it in the buffer until all the data or 3*512 =1536 bytes (fulfilling the endpoint buffer in this case) has been written.
In some case where energy calculation is wanted, the pipe n°2, endpoint 8 BULK IN can also be used by the Virtex II for acquisition data readout.

The FX2 handles automatically the interface between the USB bus and the three FIFO's. For example, every time data is sent over pipe n°0 into endpoint 2, it is immediately available for a FPGA, status flags being instantaneously updated.
When a FPGA writes some data in EP6/8 IN FIFO, any USB request asking for an amount of bytes over this pipe will be serve by the FX2 in this way:

- a soon as 512 bytes (256 words on the data I/O bus) has been written in the FIFO by the FPGA, this packet is immediately available as a USB packet for any pending request from the host which has asked for 512 bytes (or more) over this endpoint
- By writing the last byte to the FIFO, unless it has just fulfilled an entire packet, the FPGA can activate a special line/bit in the writing protocol in order to ask the FX2 to commit this 'short' packet to the USB. Thus, a host query for some bytes (even greater than the short packet size) will be served by this short packet. The host software can

detect that the initial amount of bytes asked to the USB host controller has been served by a smaller quantity.

If the FPGA wants to write 1538 bytes for example, he fill the FIFO up to 1536 bytes, then has to wait until a 512 bytes packet (or more) has been taken by some USB bulk request IN, and then can write the two last bytes with a **short packet** signal.

The two FPGA will never try to access some endpoint FIFO at the same time.

In order to get some state information and to control what happens in the FX2 some other 'Vendor request' has been defined (control transfer with a specific code). See §4.4 below

From the USB point of view, mainly most communication implementations **needs only 3 basic USB actions** :

- Send a byte array to the TNT card over a bulk pipe (orders and parameters on endpoint n°2)
- Receive a byte array from the TNT card over another bulk pipe (data over endpoint n°6 IN or over EP8 IN)
- Send some few control transfer containing a vendor request to the card

4.4 Particular USB informations & codes

USB characteristic	Value	Description
VID	0x0999	Important identification number used by operating systems
PID	0x8888	Id
String descriptors : Index=1 Index=2 Index=3	IRES/LEPSI/IN2P3/CNRS TNT2-01/2005 001x	The index 3 descriptor contains the identification number of the card (0003) which is also marked at the bottom of the front panel of the card. The card number is very important for the TUC software.
Defined Pipes	Pipe 0 BLK EP2 OUT MaxPktSz: 0x200 Pipe 1 BLK EP6 IN MaxPktSize: 0x200 Pipe 2 BLK EP8 IN MaxPktSize: 0x200	3 defined USB channel of communication for data transfers

Some control transfers are used for monitoring and controlling the FX2. A control transfer has a USB setup stage that contain an 8 bytes setup packet. This setup bytes gives informations about the type of request wanted and looks like:

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bit-Map	Bit 7 : Data Phase Transfer Direction 0 = Host to Device **1 = Device to Host** Bit 6..5 : Type of request 0 = Standard 1 = Class **2 = Vendor** 3 = Reserved Bit 4..0 : Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	Value	Code of request wanted: **see below**.
2	wValue	2	Value	Value : **always 0**
4	wIndex	2	Index or Offset	Index: **always 0**
6	wLength	2	Count	Number of bytes to transfer : **see below**

Depending of the value of the field "bRequest", different type of control transfer are defined :

- Values between 0x00 and 0x0C are reserved as standard device request by USB specifications
- Other values are commonly called "Vendor requests"
  - codes between 0xA0 and 0xAF are reserved for Cypress purposes,
  - codes starting at 0xB0 has been defined and implemented for specific TNT features (see table below). All these "TNT vendor request" are control transfers with an IN direction and waiting in return 1 or 3 bytes of data

TNT Vendor request code	Description
0xB0	FX2 puts both FPGA's in some interrupt state. Return 0xB0.
0xB1	FX2 puts Spartan in some interrupt state while the Virtex II will be released. Return 0xB1.

0xB2	FX2 puts Spartan in some release state while the Virtex II will be interrupted. Return 0xB2.
0xB3	FX2 puts Spartan in some release state while the Virtex II will be interrupted. The Spartan wait for a VHDL compiled file contents over EP2OUT and write it in the flash EEPROM of the Virtex II. Return 0xB3.
0xB4	FX2 return the number of bytes (hexadecimal encoding) over 2 bytes received in EP2OUT ready to be read by a FPGA and 0xB4.
0xB6	FX2 return 1 bytes containing the number of USB packets received over EP2OUT and ready to be read by an FPGA.
0xB7	FX2 return the value of the EMPTY and the FULL flags for EP2OUT and 0xB7. The different values are : 0xEE 0x0F 0xB7➔ empty = yes, full=no 0x0E 0x0F 0xB7➔ empty = no, full=no 0x0E 0xFF 0xB7➔ empty = no, full=yes
0xB8	FX2 return the number of bytes (hexadecimal encoding) over 2 bytes received in the packet which is in the course of filling on EP6IN and 0xB8. A non-zero value indicates that bytes has been written but not ready committed for the USB.
0xBA	FX2 return 1 byte value indicating the number of packets ready for USB on EP6IN. As this endpoint has 3 packets buffering the return values are between 0 (EP6IN is empty) and 3 (EP6IN is full).
0xBB	FX2 return 3 bytes indicating the value of the EMPTY and the FULL flags for EP6IN and 0xBB. The different values are : 0xEE 0x0F 0xBB➔ empty = yes, full=no 0x0E 0x0F 0xBB ➔ empty = no, full=no 0x0E 0xFF 0xBB ➔ empty = no, full=yes
0xBC	FX2 commit the actual packet in the course of filling as ready to be serve to some data request over EP6IN. Return 0xBC.
0xBD	FX2 return 3 bytes indicating the value of the EMPTY and the FULL flags for EP8IN and 0xBD. The different values are : 0xEE 0x0F 0xBD➔ empty = yes, full=no 0x0E 0x0F 0xBD ➔ empty = no, full=no 0x0E 0xFF 0xBD ➔ empty = no, full=yes

0xBE	FX2 return the number of bytes (hexadecimal encoding) over 2 bytes received in the packet which is in the course of filling on EP8IN and 0xB8. A non-zero value indicates that bytes has been written but not ready committed for the USB.
0xBF	FX2 commit the actual packet in the course of filling as ready to be serve to some data request over EP8IN. Return 0xBF.
Others codes	Depending on debugging and development purposes..

See the "Technical reference manual of FX2" for more detailed information about FX2 registers.

4.5 Communication examples

Informations sent to a FPGA on the card consists always of an array of bytes :

- The first word (2 bytes) contains an order code
- Words which are eventually after a first word contains parameter values. A single parameter is encoded over a certain number of bits at a precise word number after the first word.

A full parameter set (including the order code) for the Virtex II is over N bytes (actually, N=81).
A full parameter set (including the order code) for the Spartan is over M bytes (actually, M=9).

- **Get parameter set from the card** :
  - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state.
    - Empty the buffer from EP6IN from previous data: while the result of the vendor request with code 0xBA is not 00, make a bulk transfer request for 512 bytes on EP6IN. Depending of what kind of previous activity the card had, there should be maximum 3 bulk transfers.
  - Encode order 'Readout parameter set' over 2 bytes.
  - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT
  - Send off to the FX2 a vendor request with code 0xB2 in order to RELEASE the spartan and place the Virtex II in the INTERRUPT state.
  - The Spartan switch to a IN data transfer phase with the FX2 :
    - He switch the data bus to the FIFO associated with EP6IN,
    - He writes the M bytes containing every parameters values he actually had in memory.
  - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state.
  - Encode order 'Readout parameter set' over 2 bytes.
  - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT
  - Send off to the FX2 a vendor request with code 0xB1 in order to RELEASE the Virtex II and place the Spartan in the INTERRUPT state
  - The Virtex II switch to a IN data transfer phase with the FX2 :
    - He switch the data bus to the FIFO associated with EP6IN,
    - He writes the N bytes containing every parameters values he actually had in memory.

        - He send a short packet signal to the FX2, allowing this short packet to be serve by the FX2 to any bulk request for Q bytes (Q>=M+N) bytes over EP6IN
    - Send a request for 512 bytes (for example), the request should complete immediately returning M+N bytes

- **Update the parameter set of the Virtex II or Spartan** :
    - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state.
        - Empty the buffer from EP6IN from previous data: while the result of the vendor request with code 0xBA is not 00, make a bulk transfer request for 512 bytes on EP6IN. Depending of what kind of previous activity the card had, there should be maximum 3 bulk transfers.
    - Encode the full parameter set over N (Virtex II) or M (Spartan). Encode order 'Readout parameter set' over the first word of the parameter set.
    - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT

    - Send off to the FX2 a vendor request with code
        - 0xB1 in order to RELEASE the Virtex II and place the Spartan in the INTERRUPT state

        or
        - 0xB2 in order to RELEASE the Spartan and place the Virtex II in the INTERRUPT state.

    - The addressed FPGA switch to a OUT data transfer phase with the FX2 :
        - He switch the data bus to the FIFO associated with EP2OUT,
        - Until the EMPTY flag raised, he read the data in the FIFO (N or M bytes) containing the parameter set bytes and update every parameter value in memory with new values received

- **Start of an acquisition :**
    - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state.
    - Empty all FX2's IN buffers (EP6IN and EP8OUT) which may contain data from a previous order. Use for example a the vendor request with code 0xBA : while the result is not 00, make a bulk transfer request for 512 bytes on EP6IN. Depending of what kind of previous activity the card had, there should be maximum 3 bulk transfers.
    - Encode order 'Update parameter set" together with every Spartan parameters over M bytes
    - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT
    - Send off to the FX2 a vendor request with code 0xB2 in order to RELEASE the spartan and place the Virtex II in the INTERRUPT state.
    - The Spartan switch to a OUT data transfer phase with the FX2 :
        - He switch the data bus to the FIFO associated with EP2OUT,
        - Until the EMPTY flag raised, he read the data in the FIFO (M bytes) containing orders and parameters values to take into account for the acquisition (overwriting old values he actually had in memory).
    - Encode order "start acquisition of mode Z" together with every parameters over N bytes.
    - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT.
    - Send off to the FX2 a vendor request with code 0xB1 in order to RELEASE the Virtex II and place the Spartan in the INTERRUPT state
    - The Virtex II switch to a OUT data transfer phase with the FX2 :
        - He switch the data bus to the FIFO associated with EP2OUT,

- Until the EMPTY flag raised, he read the data in the FIFO (N bytes) containing orders and parameters values to take into account for the acquisition (overwriting old values he actually had in memory).

- The Virtex start to run an acquisition of the desired type / mode

- **Get acquisition data :**
  - It is assuming that the card has already started some acquisition.

  - Make a request for a bulk transfer of data to the FX2 over EP6IN or EP8IN depending of the mode of the acquisition. For example, the TUC control software ask always for 65024 bytes of raw acquisition data over EP6IN and 64512 bytes over EP8IN.
  - As soon as the card has some data to send back to the PC, the Virtex II switch to a data transfer phase with the FX2 :
    - He switch the data bus to the FIFO associated with EP6IN or EP8IN,
    - He starts to write some data until the flag FULL of the FIFO raised. A maximum of 1536 bytes can be written on EP6IN and 1024 bytes on EP8IN.
    - If there are more bytes to transfer, the Virtex II wait until some other USB request over EP6IN has been made by the host that will lower down the FULL FLAG of the endpoint FIFO
  - The card continues to work in acquisition mode, waiting for the next trigger signal.
  - As soon as the byte size of readouted events has reach the number of bytes asked by the host PC, the request started in the second point above is completed, thus allowing the host program to manipulate the byte arrays received as wanted : storage on disk, visualization,….

  - Loop back to the second point (request for a bulk transfer of data to the FX2 over EP6IN or EP8IN) if necessary.

- **Stop of an acquisition :**
  - It is possible to only stop making request of bytes over EP6IN or EP8IN, in order to only stop the 'thread' activity on the host. In this case, the card still continue to work in acquisition mode but as there is no more activity on the host that clear out the EP6IN / EP8IN FIFO buffer on the FX2, the Virtex II / Spartan will simply be blocked during a data transfer phase to the FX2, waiting for the lowering of the FULL flag. Of course, before reading out some new data, it will be necessary to clear the remaining data bytes in the EP6 / EP8 buffer.
  - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state. The card will stop any ongoing activity.

- **Get the VHDL description of the card:**

  The VHDL software contains some short version information as a fixed array of bytes containing ASCII coded characters.
  Each of the FPGA has his own description string.

  - Send off to the FX2 a vendor request with code 0xB0 in order to place the 2 FPGA's in the INTERRUPT state.
    - Empty all FX2's IN buffers (EP6IN and EP8OUT) which may contain data from a previous order. Use for example a the vendor request with code 0xBA : while the result is not 00, make a bulk transfer request for 512 bytes on EP6IN. Depending of what kind of previous activity the card had, there should be maximum 3 bulk transfers.
  - Encode order 'Readout VHDL description" over one word.
  - Send off this byte sequence over USB to the FX2 through a Bulk transfer on EP2OUT
  - Send off to the FX2 a vendor request with code

    - 0xB1 in order to RELEASE the Virtex II and place the Spartan in the INTERRUPT state

    or

    - 0xB2 in order to RELEASE the Spartan and place the Virtex II in the INTERRUPT state.
  - The addressed FPGA switch to a OUT data transfer phase with the FX2 :
    - He switch the data bus to the FIFO associated with EP2OUT,
    - Until the EMPTY flag raised, he read the data in the FIFO (2 bytes) containing containing the order to readout the VHDL description bytes
  - After analysing the order, the addressed FPGA switch to a IN data transfer phase with the FX2 :
    - He switch the data bus to the FIFO associated with EP6IN,
    - He writes the VHDL description P bytes.
    - If the addressed FPGA is the Virtex II, he will send a short packet signal, allowing this short packet to be serve by the FX2 to any bulk request for R (R>=P) bytes over EP6IN
  - If the addressed FPGA is the Spartan : send off to the FX2 a vendor request with code 0xBC in order to commit the packet being in course of filling.
  - Send a request for 512 bytes (for example), the request should complete immediately returning P bytes

  Example of VHDL description:

    SPARTAN:V.1.0 - 09/2004
    This is sent in raw form as these 28 bytes:
    53 50 41 52 54 41 4E 3A 56 2E 31 2E 30 20 2D 20 30 39 2F 32 30 30 34 3C 62 72 3E 20

    T.N.T – V.1.0 – IRES – IN2P3 – Fev. 2003
    This is sent in raw form as these 40 bytes:
    54 2E 4E 2E 54 20 2D 20 56 2E 31 2E 30 20 2D 20
    49 52 45 53 20 2D 20 49 4E 32 50 33 20 2D 20 46
    65 76 2E 20 32 30 30 33

See "0**Erreur ! Source du renvoi introuvable.Erreur ! Source du renvoi introuvable.**" for detailed informations about parameter sets of the FPGA's.

4.6 USB host programming

Programming some software for accessing TNT cards is slightly different under Windows operating systems and under linux operating systems.

Windows operating systems :

- The use of a specific driver is required (under such systems, direct I/O calls to devices are not authorized : a driver must be used in order to relay communications between 'user mode' programs and the operating system kernel)
- As, the TNT cards has been developed by using a USB2 development kit of the Cypress company, this kit is shipped with a specific driver which can be used for devices using the Cypress chip FX2. This generic driver is called 'General Purpose Driver".
- By using the drivers sources shipped aloung with the USB2 development kit of Cypress, IReS has recompiled a new driver which is exactly the same as the GPD of Cypress, except that it's attached to USB devices that has
  - VID=0x0999
  - PID=0x8888

- User softwares communicate with the driver through some functions calls which are in the Win32 API (USER32.DLL and KERNEL32.DLL) :
  - CreateFile(..) : the first step of a USB communication with a TNT card is the opening of some kind of communication channel with the USB plugged device. This is done by some « CreateFile()» function call with a specific argument string :
    - «Ezusb-x », x is a sequence number allocated by the operating system when the device is plugged on the bus. If no other USB device with same VID/PID is alreday connected, this number is zero (0). If n devices were already connected, the new connected one will received the n+1 number allocated. This function return some 'handle' to the TNT card which will be used later by other communications calls.
  - DeviceIoControl(..). Some calls of this function can be now made as needed by the USB communications need. By using different parameters values, it's possible to send some bytes to the card with a BULK OUT transfer on pipe n°2, or to pass some standard « GET_DEVICE_DESCRIPTOR » request (USB "Control message"), for example.
  - CloseHandle(). As no more communication is needed, this function call simply close the communication channel that was opened before. Argument of this function is the handle that was initially returned by the CreateFile() function call.
  - Every king of programming language or tool can be used as it is possible to make some calls to a external library (DLL file)
  - Refer to the Cypress documentation of the GPD driver ().

Linux operating systems :

- These systems were not especially studied as the development kit of Cypress was for Windows operating systems and as "**Erreur ! Source du renvoi introuvable.**" uses, under linux, some high level java API which mask the USB low level. By looking in the source of this API, it seems that at USB low level, every communication consist of some basic function calls to the linux kernel API:
  - Open ()
  - Ioctl()
  - Close ()

  It seems to be similar as under Windows systems. For IOCTL() calls a lot of codes and arguments values are avalaible for every type of USB communication : refer to the <linux/usbdevice_fs.h> file pour getting some explications
- Thus, the use of a specific driver is NOT required !
- When connecting some TNT card to the PC, the I/O subsystem receive the VID/PID sent by the card and search for a specific installed driver for such ID values. As no one exists, the 3 standard kernel API calls should complete.
  - VID=0x0999
  - PID=0x8888

4.7 Tricks & tips

- At power on, all parameters on the card has zero values

- The card search his parameters in a FX2's buffer. Each parameter has a precise position in the byte / word array which has to be sent to the card. In order to modify some parameter stored in the 25th word, for example, it is necessary to collect and encode at least all previous parameters which are defined in the previous words. Those who are defined after the 25th word must not absolutely be sent gain to the card : she will keep in memory her older values.

- If fewer parameters words than the defined total number are sent to the card, only those parameters that are defined in the words sent are updated.

- The FX2 has been configured for a SLAVE FIFO mode : the Virtex II can use some specific control lines in order to
  - connect to a specific FX2 buffer
  - implement some given read and write protocol
  - inspect the state of the connected buffer : empty, not empty, full, not full,...

- The data bus which connect the Virtex II and the FX2 has a 16 bit width and is clocked at 48 Mhz. The reading and writing protocols for the FX2 buffers are implemented in VHDL as state machines. 3 states are necessary in order to read or write a word from the FX2 data bus, thus allowing theorically about : 16Mhz *2 bytes =30.51 MB/s transfer rate between the card and the PC USB host.

- Sampled points are readout over 16 bits : ADC samples data over 14 bits, signed in a two's complement. A 15th bit give hint about ADC overflow state and a 16th bit is used for data tagging (see Data readouts formats).

- How to stop a pending request of data over pipe n°1 – Endpoint 6 IN ? This can often happen when an acquisition has been started but with no trigger type chosen, or with wrong trigger parameters in regards of the input signal.
  - Then any request over endpoint n°6 IN will never complete, thus placing the host program which is in charge of managing this request of bytes in a 'frozen' state, infinitely waiting for the completion f the request.
  - If this happens, depending of the kind of USB controller is on the host PC main board, there are 2 ways to get out of this blocking state:
    - Launch another thread / program which open a new communication channel with the card. Send a vendor request (USB control transfer) with IN direction, 1 byte asked in return and with code 0xBC. The initial request over EP6IN complete then with zero bytes in return.
    - Send some control transfer in order to abort the pipe and then reset it. This depends of the capability and supporting of such action by the USB host controller and the driver. On windows platform, with the GPD driver from Cypress, it is possible to launch such actions.

- Sampled data from ADC's is directly written in memory, based on the desired number of acquisition points

- Only one oscillogram is stored in the card and then transferred. It must be completely send over USB before the card is able to catch another one.

- If two distinct TNT cards are used in a "Oscilloscope mode", even if they are supposed to trigger at the same time by using an external trigger, the duration of the readout phase is not exactly known and couldn't be precisely controlled. It depends mainly of the PC USB host configuration : USB host controller implementation and type of driver used. Even if we consider that both cards start exactly at the same time to readout their event, it could happen that the first card which has finished the readout will catch a new event, while the readout process of the other card is still in progress. For some specific needs in the past, some VETO like feature was implemented in the TNT card (each card send to the other one a signal when she is able to trigg again after readout) but is no more available.

- A summary of the different USB pipes / endpoints used for the differents data transfers is shown below :

USB pipes / Endpoints used for data transfers Vs Card feature	EP2 Bulk OUT Pipe 0	EP6 Bulk IN pipe 1	EP 8 Bulk IN pipe n°2
Send order / parameter to the card	**X**		
Get parameters value from the card	**X**		
Oscilloscope mode		**X**	
ADC histogramer mode		**X**	
Energy histogramer mode			**X**
Oscillo+energy histogramer mode		**X**	**X**
Test counter mode		**X**	

- Under windows OS, a TNT card will appear in the device manager screen as:

"IReS TNT v1.0 acquisition card"

If, for any reason, the FX2 doesn't boot correctly over his EEPROM, the card will appear as:

"Cypress EZ-USB FX2 (68613) - EEPROM missing"

When launching the Cypress « Control Panel » software this text will appears :

EZ-USB Control Panel - built 15:43:04 Apr 29 2002
Get PipeInfo
Interface Size 16

- Under windows OS, for being sure that a TNT card is connected on the USB bus, launch the Cypress « Control Panel » software, this text will appears :

Get PipeInfo
Interface Size 56
Pipe: 0 Type: BLK Endpoint: 2 OUT MaxPktSize: 0x200
Pipe: 1 Type: BLK Endpoint: 6 IN MaxPktSize: 0x200

If for any reason, the card doesn't achieve to negociate a USB2 protocol with the PC bus master, 'MaxPktSize' field will have : 0x40. In USB1, the bulk packet size is about 64 bytes while it is 512 on the USB2.

- When asking some information to the Spartan FPGA, it will be necessary to pass some vendor request (USB 'control message') with a 0xBC code in order to commit the current filling packet being in course of filling : on the contrary of the Virtex II, the Spartan FPGA will never writes shortpacket himself.