Measurement Automation Monitoring, Dec. 2016, no. 12, vol. 62, ISSN 2450-2855

Krzysztof MROCZEK
INSTITUTE OF RADIOELECTRONICS AND MULTIMEDIA TECHNOLOGY, WARSAW UNIVERSITY OF TECHNOLOGY
Warsaw, Nowowiejska 15/19, 00-665 Poland

An Universal USB 3.0 FIFO Interface For Data Acquisition

Abstract

In this paper, an USB – DAQ interface unit that allows connecting data acquisition (DAQ) application to USB is presented. The unit contains two main components: USB to FIFO IC controller and application controller, designed as VHDL core for FPGA. DAQ logic can be connected to USB through simple I/O and streaming interfaces, thus development time of user application can be reduced. The design was tested with high-speed and SuperSpeed FTDI and Cypress USB – FIFO controllers.

Keywords: data acquisition, USB 3.0, SuperSpeed, FPGA, hardware interface.

1. Introduction

Data acquisition systems (DAQ systems) often require high bandwidth for data transmission between DAQ measurement hardware and processing workstation. In many applications sampling frequency of Analog-to-Digital converters (ADCs) must be set to tens or hundreds of MHz, so large streams of data can be written to memory in real time. Typical signal digitizer contains ADCs channels and logic circuit, including trigger/sample unit and registers. Universal or specialized DAQ boards are also equipped with Digital-to-Analog (DAC) converters. Nowadays USB 3.0 interface is commonly used in modern computers and notebooks and can be considered as a fast communication link for DAQ hardware. The USB 3.0 is backward compatible with USB 2.0 and adds new SuperSpeed link. SuperSpeed bus operates in dual simplex mode with the signal rate of 5 Gb/s. In SuperSpeed protocol layer there are several new elements, e.g. data bursting and asynchronous notifications, which improve bus quality. Real transmission throughput is about 400 MB/s (3.2 Gb/s) for SuperSpeed connection. USB offers plug-in play features, detection of transmission errors and also external connectivity, so it can be attractive for developers of DAQ devices. However, complexity of the protocol implies the use of specialized components, like protocol controller. In software layer, stack of OS drivers is used to communicate with USB devices. Typically, user application sends request to the USB driver for starting data transfer. USB host controller driver (HCD driver) processes the request and initiates the transfer in host controller hardware. The connections are packet oriented. Data on the bus are divided into transaction packets. USB is not memory mapped, like e.g. PCI-Express, therefore DMA transmission must be emulated.

The effective way to implement USB connection is the use of two main components: USB protocol IC and application controller. Several papers, e.g. [8-9], [11-13], describe USB interface designs, where one selected USB – FIFO IC controller was used in a particular project. In this paper an universal USB interface unit is presented. The architecture of USB interface is depicted in Fig. 1. USB to FIFO controller implements USB protocol in hardware and FIFO bridge for local bus. FIFOtoDAQ application controller is the FIFO bus master, performs packet processing and implements I/O interfaces. The main advantage of this configuration is easy integration of user application with the interface module. DAQ logic can be connected without additional effort for USB protocol processing. The interface was tested with popular USB – FIFO controllers from:

– Future Technology Devices International Limited (FTDI): FT 2232H, FT601Q chips,
– Cypress Semiconductor: EZ-USB FX2LP and SuperSpeed FX3 chips.

In this paper, some features of SuperSpeed USB – FIFO controllers, important from implementation point of view, are described. Application parameters of FIFOtoDAQ and evaluation of data throughput for several configurations of the interface are presented.

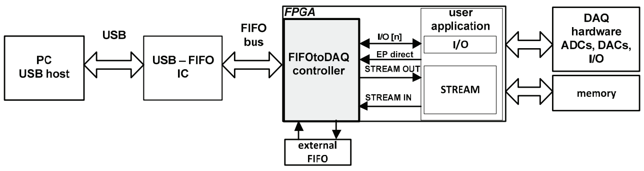


Fig. 1. Architecture of the USB interface

2. FTDI USB 3.0 – FIFO controllers

FTDI FT600/601 USB to FIFO controllers enable easy integration path for new SuperSpeed and existing USB 2.0 projects. USB protocol is implemented in hardware, without additional firmware. There are two FIFO bus modes available by configuration. First "245 synchronous FIFO" mode is similar to [1] with 16- or 32-bit data bus. Two bulk channels (1 IN, 1 OUT endpoint) and 4096 B x 2 (double buffered) memory space per endpoint are available. There are two signals for status reporting: RXF# – status of receive FIFO and TXE# – status of write FIFO. FIFO master can read data from FIFO when RXF# = 0. Read transaction must be finished when RXF# = 1. For IN direction, the master can write data to FIFO when TXE# signal is asserted. Write cycles are invalid when TXE# = 1.

The second "Multi-Channel FIFO" mode gives access to 1, 2 or 4 pair of bulk endpoints.16 kB FIFO buffer is divided between active endpoints with double buffering. The signaling protocol for multi-channel mode is different. FIFO bus master checks status of all endpoints in IDLE phase of the bus. Next, it chooses one of channels and performs data transmission.

It is recommended that bus master performs FIFO write and read in one burst transaction [2-3]. In write operation, bus master should write packets of size equals to FIFO buffer (e.g. 4096 B in 245 mode) except last packet. Writing short packet, that is the packet of size smaller then USB *MaxPacketSize*, will finish the current transfer. Therefore random writes short packets are not optimum. Possibility of controlling the transfers by burst write size distinguishes FT60x controllers from the others [1, 4, 5], which use external signal to commit data to USB. It is important feature for DAQ applications, when data samples from ADC channel are generated periodically. If samples are not delivered continuously, bus master should have memory buffer to be able to perform burst transactions.

FTDI provides Windows driver and ftd3xx.dll library for user applications. USB transfers can be performed synchronous or asynchronous using *FT_WritePipe* and *FT_ReadPipe* functions. Transfer from endpoint to host, initiated by *FT_ReadPipe*, is finished if allocated number of data bytes are received, error is detected or short packet is read. Additionally, read session is finished if master has written *MaxPacketSize* bytes and *Cancel Session On Underrun* option was set to enabled [3]. Separate software thread can be used for queuing of IN or OUT transfers. Queuing of transfers is programmed using the OVERLAPPED parameter in API calls.

3. Cypress EZ-USB FX3 SuperSpeed USB Controller

Cypress FX3 is configurable USB 3.0 peripheral controller based on ARM9 SoC architecture [5]. The block diagram of the chip is depicted in Fig. 2. The controller supports up to 32 endpoints, which are individually configured by firmware as bulk, interrupt, isochronous or control. USB data can be processed by firmware or transmitted to selected peripherals. The DMA controller enables transparent data transmission between USB and external local bus. In this case, only chip initiation and events processing should be programmed in firmware. FX3 SDK includes Eclipse-based software environment for firmware development in C.

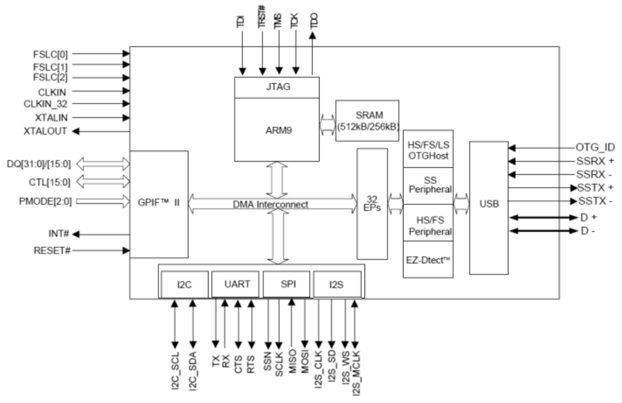


Fig. 2. Block diagram of FX3 controller [5]

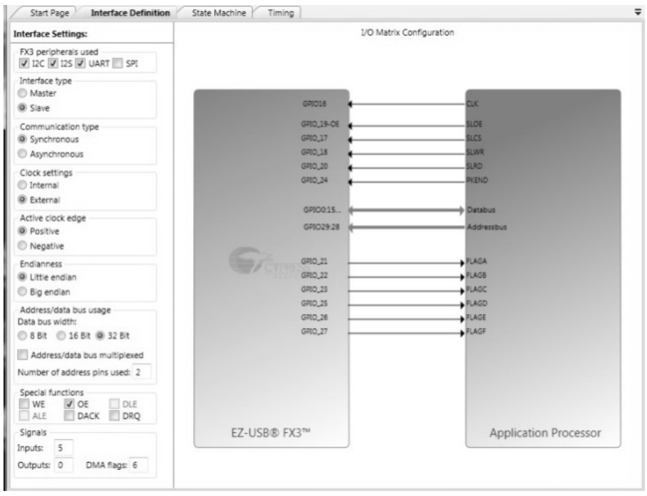
FX3 controller integrates GPIF II interface for local bus implementation. It is a part of larger block called processor interface block (PIB). Functionality of the interface is more advanced compared with previous FX2LP controller. GPIF II has a programmable state machine that allows to define many bus protocols. To implement a bus protocol, set of internal registers should be programmed. The GPIF II Designer tool is used for local bus designing and generation of configuration file for registers configuration. The user protocol is defined as a set of internal states with associated actions. Local bus ports include data bus DQ[31:0] (8 to 32-bit wide), control signals CTL[15:0] and interrupt pin. Bus control signals and internal signals are inputs to this state machine [5].

One of implementation of GPIF II, included in FX3 SDK, is Slave FIFO slave interface [5-6]. In Slave FIFO configuration, external master device writes and reads continuous data stream in way similar to FIFO memory access. The signaling of FIFO can be selected as asynchronous or synchronous. DMA channels are used for data transmission. There are two Slave FIFO configurations having different address bus width. Configuration with 2-bit address gives access to four USB sockets, that is four USB endpoints [6]. Address signal on the FIFO bus selects active GPIF II socket for FIFO transactions. For this configuration, firmware projects *sync_slave_fifo_2bit* and *async_slave_fifo_2bit* can be used as a start point of a design. Fig. 3 shows configuration with 6 FIFO flags, based on the default project. This configuration, denoted as configuration A, was used in interface tests described in chapter 6.

The second configuration of FIFO interface enables maximum 5-bit addressing on the bus. It is usable for applications if more than four USB endpoints is required. There are some limitations for 5-bit addressing, denoted in documentation [6], due to implementation of sockets switching.

Slave FIFO interface must be configured in firmware to be functional. The main steps are described below [5-7].

1. During chip initialization, the *CyFxSlFifoApplnInit* function is called. This function initializes USB and GPIF II interfaces. GPIF II interface protocol is configured according to settings generated using GPIF II Designer tool.


Start Page | Interface Definition | State Machine | Timing

Interface Settings:

FX3 peripherals used: [✓ I2C] [✓ I2S] [✓ UART] SPI

Interface type: Master [✓ Slave]

Communication type: [✓ Synchronous] Asynchronous

Clock settings: Internal [✓ External]

Active clock edge: [✓ Positive] Negative

Endianness: [✓ Little endian] Big endian

Address/data bus usage
Data bus width: 8 Bit 16 Bit [✓ 32 Bit]
Address/data bus multiplexed
Number of address pins used: 2

Special functions: WE [✓ OE] DLE ALE DACK DRQ

Signals
Inputs: 5
Outputs: 0 DMA flags: 6

I/O Matrix Configuration

EZ-USB® FX3™	Application Processor
GPIO16	CLK
GPIO_19-OE	SLOE
GPIO_17	SLCS
GPIO_18	SLWR
GPIO_20	SLRD
GPIO_24	PKEND
GPIO0:15...	Databus
GPIO29:28	Addressbus
GPIO_21	FLAGA
GPIO_22	FLAGB
GPIO_23	FLAGC
GPIO_25	FLAGD
GPIO_26	FLAGE
GPIO_27	FLAGF

Fig. 3. 2-bit slave FIFO project in GPIF II Designer tool

2. After USB SET_CONF event has been received, event callback function is called to configure DMA as it is described in the next steps.
3. Configuration of USB producer (OUT) and consumer (IN) endpoints. Each endpoint is configured using the *CyU3PSetEpConfig* function. This function sets endpoint type and properties, including burst length for SuperSpeed. Burst length must be specified in *SuperSpeed endpoint companion* descriptor.
4. Creation of DMA channels for USB to GPIF II PIB data transmission. Each DMA channel has one data producer socket (source) and one data consumer socket (sink). The configuration of the DMA channel is performed by the *CyU3PDmaChannelCreate* function. The configuration parameters include:
   – USB socket number (mapped to USB endpoint) as the data producer,
   – GPIF II PIB socket number (selected by FIFO address) as the data consumer,
   – DMA byte mode,
   – DMA buffer size equals to:
   MaxPacketSize*DMA_BUF_SIZE, where
   DMA_BUF_SIZE is the SuperSpeed burst length,
   – number of DMA buffers, typically from 2 to 8.
   The function is called with CY_U3P_DMA_TYPE_AUTO parameter and sets automatic DMA channel. In this case, data transmission is fully transparent, without firmware intervention.
5. Creation of DMA channels for GPIF II PIB to USB data transmission, similarly as in step 4.
6. Clearing and enabling of DMA channels.

FIFO master initiates single or burst transactions. End of IN transfer is requested by assertion of PKTEND# signal, that is a request to generate short or zero-length packet. Without using of this signal, USB packets have *MaxPacketSize* bytes length. FIFO status is determined by flags signals FLAGx. The number of flags is limited by number of free GPIF I/O pins. The flags are configured in GPIF II Designer project as [5-6]:

– Dedicated thread flags: empty, full, partial empty, partial full; status of FIFO channel is valid as the status of particular DMA thread.
– Current thread flags: empty, full, partial empty, partial full; the status is valid after setting of FIFO address.

Partial flags must be configured using the *CyU3PGpifSocketConfigure* function. It sets almost full or almost empty threshold for selected GPIF II socket, called watermark [6]. The function also sets minimum burst length for FIFO transactions. The FIFO flags have nonzero latency delay. FIFO master can start transaction if FIFO full or empty flag signals that DMA channel is ready. To determine end of FIFO buffer, partial flags or dedicated counters can be used. Write and read counters can be programmed to count the number of FIFO transaction words in selected channel. FIFO channel should be disabled after:

a) partial empty (read direction) or partial full (write direction) flag is asserted and remaining transactions have been done,
b) channel counter has counted NW = n*MaxPacketSize bytes; the maximum value of NW is equal to DMA buffer size for IN direction; for OUT direction it is the byte count of the buffer available for reading.
c) PKTEND# signal was asserted at the end of write transaction.

The FIFO channel is enabled again after FIFO flag signals that access to FIFO is possible. The time needed for DMA buffer switching is nondeterministic but rather short [6]. Therefore, for FIFO slave configuration, FIFO master should have additional memory buffer. Further information for hardware developers are contained in Cypress documentation.

Cypress provides cyusb3.sys Windows driver and programming libraries for C/C++ and .NET. The best for high speed processing is asynchronous API. Selected numbers of transfers can be queued in separated thread for high throughput.

4. FIFOtoDAQ application controller

Block diagram of the FIFOtoDAQ application controller is depicted in Fig. 4. It is implemented as VHDL IP core. The design is based on previous work [10]. The FIFOtoDAQ acts as the bus master to the USB FIFO slave interface and the controller for the local interfaces. Protocols specific to USB and FIFO are implemented internally.

Currently, FIFOtoDAQ supports the following USB controllers:

1. USB 2.0 – FIFO: FTDI FT2232H, FT232H; Cypress EZ-USB FX2LP, Dn=8.
2. USB 3.0 – FIFO: FTDI FT600, FT601; Cypress EZ-USB FX3, Dn=8, 16, 32.

Clock signal CLK_INTF is required for synchronous FIFO configuration. It could be generated by internal PLL or by FIFO controller. Clock frequency should be set to 100 MHz for USB 3.0 controllers. The FIFOtoDAQ implements the following local interfaces:

1. IO0, …, IO3, interface for emulation of register access, signals:
   IO_write_stb[n], *IO_wait_wr[n]*, *IO_dataw[n](31:0)* – write,
   IO_read_stb[n], *IO_wait_rd[n]*, *IO_datar[n](31:0)* – read,
   IO_adress[n](7:0) – address.
2. STREAM OUT for writing incoming data packets to local memory or processing unit; data are visible as a continuous stream of bytes.
3. STREAM IN; FIFO-like interface for writing continuous data stream to USB, e.g. from ADCs channels.
4. EP Direct. Enables access to additional USB endpoints, not directly used by FIFOtoDAQ. It is used in extended configuration of the core.

Compared with previous work [10], current version of this core adds support for USB 3.0 – FIFO controllers, including:

– basic configuration: one pair of bulk endpoints used by FIFOtoDAQ,
– extended configuration: basic + additional endpoints for FT60x controllers in FT600 mode; basic + additional endpoints for FX3 controllers configured in FIFO 2, 3 or 4 bit mode.

For some applications, the extended configuration may be more flexible because pipes for different data origins can be supported independently.

The USB FIFO master unit is the master for the FIFO slave interface. FIFO transaction signals are generated after a request is asserted by arbitration logic. To achieve high throughput, burst transactions are supported. Prediction of the end of FIFO buffer is implemented using partial flags (only for Cypress controllers) or modulo-NW counters. For FIFO read operation, monitoring of transaction validity (read if not empty) is added. After detection of invalid read, read data are discarded and transaction is finished.

The arbitration block select one of internal requests and connects it to the FIFO master. I/O and STREAM IN packets written to the FIFO are multiplexed. I/O transactions during streaming are supported. Commit request is used to request of PKTEND# or SIWU# assertion. For FT60x controllers, end of IN transfer is forced after burst finishing, when commit request was asserted or the burst size was not integer multiple of *MaxPacketSize*. External requests for EP direct are generated in application specific way.

The packet processing block decodes and executes commands sent by host in packets. Execution of read command with commit flag gives as a result setting of commit request on the end of packet. I/O decoder selects one of local buses or internal register according to command code and address fields.

The stream processing unit reads data written to dual-port FIFO buffer and encodes them into packets. Two buffering modes can be set by software: a) non buffering mode; data read from input FIFO are passed directly to USB master; b) buffering mode; the external FIFO IC is used as a additional data buffer. Currently, the synchronous FIFO ICs with size up to 512 k*8/32 bit are supported.

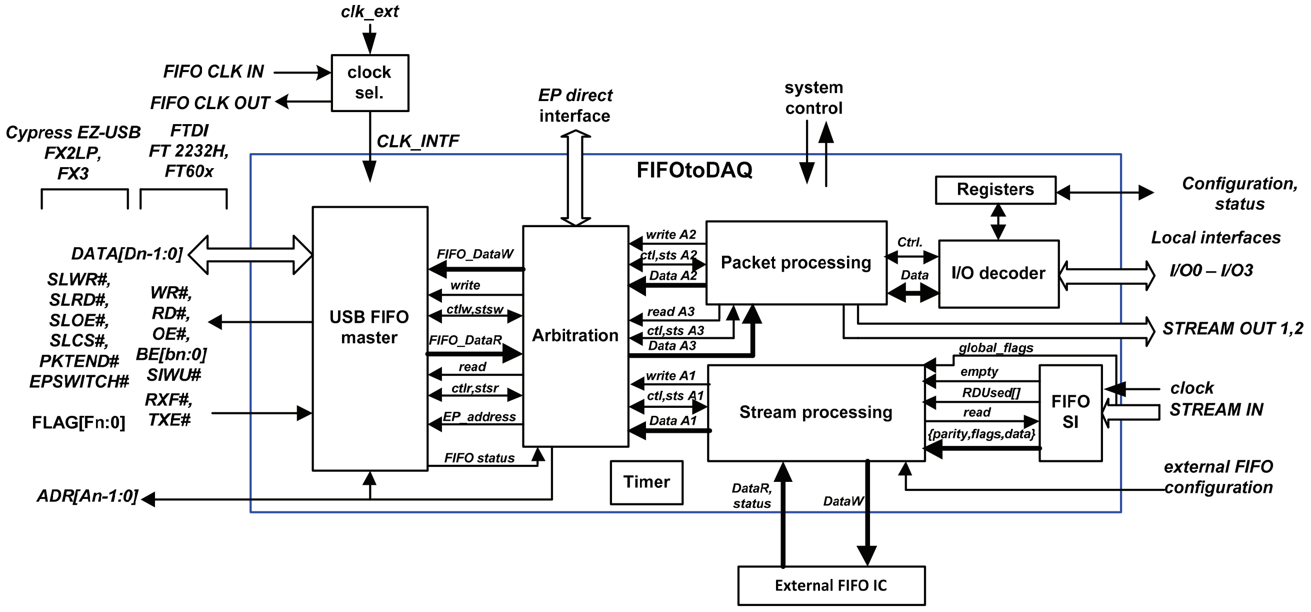


Fig. 4. Block diagram of FIFOtoDAQ core

Communication protocol

Commands and data transmitted from/to USB host are encoded into packets. Local I/O interfaces are accessible as bulk endpoints. The structure of IRP (*I/O request packet*) for OUT direction is shown in Fig. 5. The 4-bytes packet header consist of number of bytes in IRP, followed by encoded commands. The commands can be placed in one or more transaction packets.

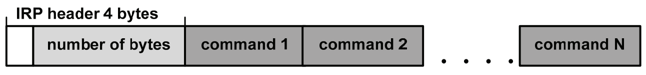


Fig. 5. Structure of IRP sent by USB host

Table 1 summarizes packet formats. The 8-bit sequence number is included in each packet. It is incremented by 1 for next packet. One set of commands (*write*, *read* and *read+commit*) enables access to internal registers; the second (*write I/O(n)*, *read I/O(n)* and *read I/O(n)+commit*) is used for access to the I/On interface. Stream packets are automatically passed to USB master unit if there are data in input FIFO. Table 2 summarizes available options. EP direct channels are supported in application specific way.

Tab. 1. Data encoding in packets

Operation	Request from host	Data to host
write write I/O(n)	packet code, sequence number, start address 8-bit, stop address 8-bit, data 32-bit, …	no response
read read+commit read I/O(n) read I/O(n) + commit	packet code, sequence number, start address 8-bit, stop address 8-bit.	packet code, sequence number, start address 8-bit, stop address 8-bit, data 32-bit, … next bytes optionally fill packet up to NW bytes.
special: CLEAR_CARD, NOP	packet code, sequence number, 2 bytes – 0.	no response
stream_out	packet code, sequence number , 2 bytes – 0, block of data.	no response
stream_in	-	4-8 bytes – header, block of data, 8 bytes – end of packet (pretriggering, status, CRC16)

Tab. 2. STREAM IN channel options

parameter	options
Data bus width	8, 16, 32 bit, configurable
status	4-bit, associated with every data word written to FIFO SI (pretriggering, overrun error, status), summarized as the status field in the packet
packet size	programmable, constant, variable
buffering	1. direct connection from input FIFO to USB 2. data path through external FIFO IC
write request	1. any data are available in FIFO (ALG. 0) 2. there are enough data to form packet (ALG. 2)
finishing of IN transfer	1. filling of transfer buffer 2. write of programmed number of packets 3. as a result of user request 4. internal latency timer expired

Results

Described interface module was evaluated in hardware. The FIFOtoDAQ controller and DAQ logic emulator were implemented in Intel/Altera Cyclone III FPGA. Development modules with FTDI FT601 and Cypress FX3 USB 3.0 controllers were used for communication with USB host. The tests were performed on PC, containing *Intel(R) i7-4790K* 4 GHz processor with 8 GB RAM, Intel*(R) USB 3.0 eXtensible host controller*, working under Windows x64 OS. Transmission throughput over 40 MB/s was measured for USB high-speed connection. Table 3 and 4 show maximum throughputs for SuperSpeed connection. They are obtained for the following conditions:
– continuous data flow in STREAM IN channel,
– queuing of 16 bulk IN transfers, the size of transfer buffer was set from 65536 to 262144 bytes; transfer thread copies data to application buffer (Fig. 6).
– I/O transfers (I/On or EP Direct commands) were performed every 5 ms during input streaming.
The throughput values in meaning ”overrun” were estimated as maximum, when increase of number of overrun errors hasn’t been observed during test. Overrun errors are reported in STREAM IN packets when data can’t be written because FIFO SI is full.The results for STREAM OUT are related to write 10 MB block of data.

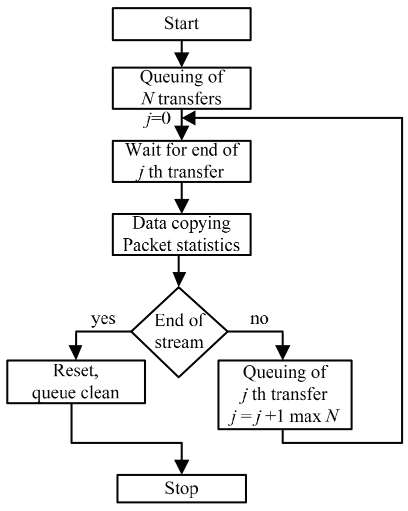

Start
Queuing of N transfers
j=0
Wait for end of j th transfer
Data copying Packet statistics
End of stream
yes
no
Reset, queue clean
Queuing of j th transfer j = j +1 max N
Stop

Fig. 6. Execution flow of software thread for queuing of USB IN transfers

Stable connections for every configuration of the interface were observed during prototyping. ADC data rate can be scaled down to selected value. Data transmission in STREAM IN channel can be performed in parallel with I/O read and write transfers. Maximum throughputs for “overrun” condition are smaller than maximum transmission throughputs. This phenomena shows that loss of data is possible for on-line streaming. Loss of data may occur as an effect of transfers management in the USB host. Moreover, execution of the thread, which queues the transfers, may be delayed by other threads. To overcome this problem, queuing of appropriate number of USB transfers may be tried. Addition of memory buffer in DAQ side is another straightforward solution.

In configuration with FTDI FT601 controller, C++ unmanaged application using FTDI ftd3xx.dll library and ftdibus3.sys driver (v. 1.2.0.5) were applied for the tests. Throughput depends on controller mode and numbers of active pipes. The best performance was noted in the “245 mode” with transfer rates up to 300 – 312 Mbytes/s. In the “Multi-Channel FT600 mode” with 2 pairs of endpoints, the maximum throughputs were smaller and near to 260 – 270 Mbytes/s. The “overrun” values are significantly smaller. All results are obtained for additional buffering with 64 kB external FIFO and 16 kB FIFO SI. To achieve optimum performance, burst write size must be set to endpoint buffer size: 4096 or 2048 bytes, depending on the controller mode. It was programmed by setting SI the write request to ALG. 2 (see Tab. 2) and adjustment of packet size. The throughput can be increased if the *FT_SetStreamPipe* function is used. This function assumes fixed size of data, therefore it wasn’t applied in our tests. Additional I/O and EP Direct transfers noticeably decrease STREAM IN throughput. As a conclusion resulting from these tests, for described FT601 connection, DAQ device should have larger memory buffer to avoid overruns for high data rate.

Tab. 3. Throughput for SuperSpeed connection, FTDI FT601 controller

configuration	throughput (th) Mbytes/s
FTDI FT601 32-bit 100 MHz: FIFOtoDAQ: FIFO SI 4096 x 32 bit; FIFO external 16384 x 32 bit	
FIFO 245 mode, 1 EP IN Bulk, 1 EP OUT Bulk	
STREAM IN maximum	312 1)
STREAM IN overrun	240 < th < 290
STREAM IN + write IO(n) + read IO(n) maximum	300
STREAM IN + write IO(n) + read IO(n) overrun	133 < th < 240
STREAM OUT	335
Multi-Channel FT600 mode, 2 EP IN Bulk, 2 EP OUT Bulk	
STREAM IN maximum	271 1)
STREAM IN overrun	th > 220
STREAM IN + write IO(n) + read IO(n) maximum	260
STREAM IN + write IO(n) + read IO(n) overrun	th > 133
STREAM IN + write EP_Direct + read EP_Direct maximum	260
STREAM IN + write EP_Direct + read EP_Direct overrun	th >133
STREAM OUT	300

1) FT_SetStreamPipe API hasn't been used

The configuration with FX3 controller was tested for two firmware options:

a) Configuration A: synchronous USB-FIFO 32-bit, 2-bit addressing, DMA watermark flags used by FIFOtoDAQ, 1 EP IN, 1 EP OUT for FIFOtoDAQ, 1 EP IN, 1 EP OUT for EP Direct channels, all endpoints are bulk.
b) Configuration B: synchronous USB-FIFO 32-bit, 4-bit addressing, DMA ready flags used, 1 EP IN, 1 EP OUT for FIFOtoDAQ, 4 EPs IN, 4 EPs OUT for EP Direct channels, all endpoints are bulk. EP Direct channels are pooled in hardware.

The size of SuperSpeed burst was set to 16. The size of DMA buffer for IN FIFOtoDAQ channel was set to 8*16*1024 = 131072 bytes. The maximum throughputs were estimated to 389 Mbytes/s. The "overrun" throughputs are near to maximum for SuperSpeed connection. Most often, the results over 380 Mbytes/s were reported. In was observed that more than 16 transfers should be queued to avoid overruns when other OS tasks were running.

Preliminary tests for isochronous pipe for STREAM IN channel have been also performed. As a result of these tests, data rate for isochronous transmission can be set up to 290 Mbytes/s, with possibility of setting of ADC rate to smaller value. More comprehensive conclusions will require additional work.

It is necessary to noting that C# managed application and CyUSB.dll library were used in FX3 configuration. Generally, C# environment wasn't developed for strong real-time applications. Direct access to USB driver from C++ unmanaged code can be selected as the second option.

Tab. 4. Throughput for SuperSpeed connection, Cypress FX3 controller

configuration	throughput (th) Mbytes/s
Cypress CYUSB3014 32-bit 100 MHz, configuration A, configuration B FIFOtoDAQ: FIFO SI 2048 x 32 bit, direct path FIFO SI – FIFO USB writeT: write I/O(n) or write EP Direct; readT: read I/O(n) or read EP Direct;	
STREAM IN maximum	389
STREAM IN overrun	320 < th <389
STREAM IN + writeT + readT maximum	389
STREAM IN + writeT + readT overrun	320 < th <389
STREAM OUT	384

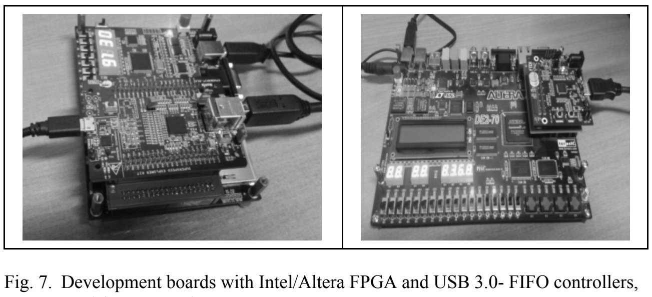


Fig. 7. Development boards with Intel/Altera FPGA and USB 3.0- FIFO controllers, used for prototyping

5. Summary

In this paper universal USB 3.0 FIFO to DAQ interface module has been presented. The goal of this work was developing and prototyping complete USB interface module for many DAQ devices. The FIFOtoDAQ application controller was developed for FPGA projects for this purpose. The design was successfully prototyped with most popular FTDI and Cypress USB 2.0/3.0 FIFO controllers. An user can select one of USB-FIFO controllers and connect DAQ hardware to USB through simple FIFOtoDAQ interfaces. Moreover, real-time transmission of ADCs data can be performed in parallel with bidirectional I/O transfers. Data throughputs were evaluated to over 40 Mbytes/s for USB high-speed and 133 – 389 Mbytes/s for SuperSpeed connections. The important observation from this work is related to finding optimum software-hardware configurations for real-time data transmission over USB. Currently we have good starting point for software development.

6. References

[1] FT2232H Dual High Speed USB To Multip. UART/FIFO IC, FTDI
[2] FT600Q-FT601Q IC Datasheet (USB 3.0 to FIFO Bridge), FTDI
[3] FT600_FT601 USB Bridge chips Integration, app. note an_412, FTDI
[4] EZ-USB® Technical Reference Manual, Cypress Semiconductor
[5] EZ-USB® FX3 Technical Reference Manual, Cypress Semiconductor
[6] Gandhi S., Designing with the EZ-USB® FX3™ Slave FIFO Interface, AN65974, Cypress Semiconductor
[7] Slfifosync firmware example, Cypress EZ-USB® FX3™ SDK
[8] Zabołotny W.M., Kasprowicz G., Low cost USB – local bus interface for FPGA based systems, Proceedings of SPIE Vol. 8454, 2012
[9] Jolfaei F.A., Mohammadizadeh N., Sadri M.S., FaniSani F., High Speed USB 2.0 Interface for FPGA Based Embedded Systems, 4th International Conference on Embedded and Multimedia Computing EM-com, 2009
[10] Mroczek K., USB FIFO interface for FPGA based DAQ applications, 8th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems IDAACS'2015, 24-26 September 2015
[11] Shah K., How to Implement an Image Sensor Interface Using EZ-USB® FX3™ in a USB Video Class (UVC) Framework, AN75779, Cypress Semiconductor
[12] Janßen B., Hübner M., Jaeschke T., An AXI Compatible Cypress EZ-USB FX3 Interface for USB-3.0 SuperSpeed, 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)
[13] Abba A., Caponio F., Cusimano A., Geraci A., Implementation of USB 3.0 Bus Controller in FPGA for Data Transfer in Multi-Channel Application, 2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)

Received: 08.09.2016 Paper reviewed Accepted: 02.11.2016

Krzysztof MROCZEK, PhD

He received PhD degree from Warsaw University of Technology in 2002. His research interests include applications of FPGA, designing of SoPC systems, embedded systems. He works at Institute of Radioelectronics and Multimedia Technology Warsaw Univeristy of Technology.

e-mail: K.Mroczek@ire.pw.edu.pl