Lime Microsystems Limited

Surrey Tech Centre
Occam Road
The Surrey Research Park
Guildford, Surrey GU2 7YG
United Kingdom

Tel: +44 (0) 1483 685 063
Fax: +44 (0) 1428 656 662
e-mail: enquiries@limemicro.com

Stream Board Communications

Chip version:	LMS7002Mr2
Chip revision:	02
Document version:	03
Document revision:	03
Last modified:	03/04/2015 12:22:35

Contents

Revision History

Version v01r00
Released: 19 Jan, 2015
Initial version.

Version v01r01
Released: 03 Feb, 2015
Added FPGA programming description and example, changed document structure.

Version v01r02
Released: 18 Mar, 2015
Chapter 3 added.

Version v01r03
Released: 23 Mar, 2015
Chapter 4 added.
Chapter 2 updated.

1

IQ Samples data formats for PC and FX3 communications

1.1 Uncompressed data format

Stream board samples uses 1 bit for I or Q channel selection and 12 bits for amplitude. One pair of I and Q samples makes up one frame. Each frame is transferred using 4 bytes as shown in Table 1.This data format is used for uploading and downloading samples.

Table 1 Samples data buffer

Buffer byte index	Data (8-bits)	Frame
0	[7:0] I0 LSB	0
1	0x00 + [3:0] I0 MSB	
2	[7:0] Q0 LSB	
3	0x10 + Q0 MSB	
4	[7:0] I1 LSB	1
5	0x00 + [3:0] I1 MSB	
6	[7:0] Q1 LSB	
7	0x10 + Q1 MSB	
...	...	...

2

Stream Board PC and FX3 communications

Stream board can be connected to the PC by USB 2.0 or USB 3.0. The software can perform SPI registers writing and reading operations over USB device's control endpoints using LMS64C control protocol, see [1] for more information. The control endpoints should be configured as shown in Table 2.

Table 2 USB control endpoints configuration

Field	In Endpoint	Out Endpoint
Request Code	0xC0	0xC1
Value	0x0000	0x0000
Index	0x0000	0x0000

The following commands are supported in Stream board bridge MCU (FX3):

- CMD_GET_INFO
- CMD_LMS_RST
- CMD_SI5351_WR
- CMD_SI5351_RD
- CMD_LMS6002_WR
- CMD_LMS6002_RD
- CMD_LMS7002_WR
- CMD_LMS7002_RD
- CMD_ADF4002_WR
- LMS_BRDSPI16_WR
- LMS_BRDSPI16_RD
- CMD_ALTERA_FPGA_GW_WR

1.1 Getting board information

Send a CMD_GET_INFO command to the bridge MCU to get an infomation about the hardware as described in [1]. In case of Stream board information is returned by MCU as shown in Table 3.

Table 3 Board information

Field	Value
FW_ver	2
Dev_type	8
Protocol_ver	1
HW_ver	2
EXP_ID	0

1.2 Reseting LMS chip

Send a CMD_LMS_RST command to the bridge MCU to to reset the LMS chip as described in [1].

1.3 Communicating with LMS transceivers

Variuos expansion boards may be connected to the Stream board. Expansion boards may be populated with LMS6002D or LMS7002M transceivers. The commands CMD_LMS6002_WR, CMD_LMS6002_RD, CMD_LMS7002_WR and CMD_LMS7002_RD are designed to setup and read setup data to these transceivers. Check [1] for more information.

1.4 Communicating with Si5351

Stream board has Si5351 clock synthesizer onboard. The commands CMD_SI5351_WR and CMD_SI5351_RD are dedicated for Si5351 clock synthesizer setup. See [1] for more information.

1.5 Communicating with ADF4002

Stream board has ADF4002 PLL onboard. The command CMD_ADF4002_WR is dedicated for ADF4002 setup. It is not possible to read the configuration data from ADF4002 hence no corresponding command. See [1] for more information.

1.6 FPGA modules setup

FPGA gateware on Stream board is controlled by 16 bit memory registers, which can be read or written by using LMS64C data protocol and CMD_BRDSPI16_WR and CMD_BRDSPI16_RD commands as described in [1]. The SPI registers are shown in Table 4.

Table 4 Stream board registers

Address (15 bits)	Bits	Description
0x0005	15 – 5	**Unused**
	4	CH_SEL: Selects Rx channel for streaming to FX3 0 – DIQ1(**default**) 1 – DIQ2
	3	STREAM_RXDSRC: Selects Rx data source: 0 – NCO(**default**) 1 – ADC
	2	STREAM_RXEN: Enables data streaming from FPGA to FX3 0 – Disabled(**default**) 1 – Enabled
	1	STREAM_TXEN: Enables data streaming from RAM 0 – Disabled 1 – Enabled (**default**)
	0	STREAM_LOAD: Enables data loading 0 – Enabled(**default**) 1 – Disabled **Default**: 0000000 00000010

1.7 FPGA programming

FPGA programming is performed by sending data to USB device's control end point. Data needs to be encapsulated in LMS64C packet, using CMD_ALTERA_FPGA_GW_WR command, as described in [1].

Example C++ program for FPGA programming is given in 1.10.3 chapter.

1.8 IQ samples writing

IQ Samples writing is performed by sending data to USB device's bulk end point, which has address 0x01. Before that Stream board has to be configured to accept incoming data.

The general steps for IQ samples writing are:

1. Enable STREAM_LOAD and disable STREAM_TXEN, STREAM_RXEN.
2. Transfer samples data to USB Endpoint whose address is 0x01. Samples writing can be done either in one transaction, or several successive transactions.
3. Disable STREAM_LOAD and enable STREAM_TXEN.

Minimal C++ program for writing IQ samples is given in 21.10.1 chapter.

1.9 IQ samples reading

IQ Samples reading is performed by reading data from USB device's bulk end point, which has address 0x81. Before that Stream board has to be configured to send out data samples.

The general steps for IQ samples reading are:

1. Enable STREAM_RXEN.
2. Read data samples from USB Endpoint whose address is 0x81. Reading can be repeated indefinitely.
3. After reading is done. Disable STREAM_RXEN.

Minimal C++ program for reading IQ samples is given in 21.10.2 chapter.

1.10 FX3 communication examples

1.10.1 IQ Samples writing example

This program is written in C++ on Windows operating system. USB communications are performed using CyAPI library, it can be downloaded from EZ-USB FX3 Software Development Kit for Windows. The program reads given WFM file, converts it to 12 bit values and uploads to Stream board.

```
#include <windows.h>
#include "CyAPI/CyAPI.h"
#include <iostream>
#include <stdio.h>
#include <vector>

#define CTR_W_REQCODE 0xC1
#define CTR_W_VALUE 0x0000
#define CTR_W_INDEX 0x0000

#define CTR_R_REQCODE 0xC0
#define CTR_R_VALUE 0x0000
#define CTR_R_INDEX 0x0000
#define CMD_BRDSPI_WR 0x55//16 bit spi for stream
#define CMD_BRDSPI_RD 0x56//16 bit spi for stream
using namespace std;

int ReadWFM(const char *filename, std::vector< std::pair<int, int> > &iq_pairs)
{
    FILE *fpin;
    unsigned char c1, c2, c3, c4;
    double iin, qin; // IQ inputs
    int iint, qint; // IQ integer versions
    int cnt = 0;

    fpin = fopen(filename, "rb");
    if( fpin == NULL)
    {
        printf("Input file can not be opened.");
        return -1;
    }
    while( fscanf(fpin, "%c%c%c%c", &c1, &c2, &c3, &c4) == 4 )
    {
        cnt++;
        c1 &= 0xFF;
        c2 &= 0xFF;
        c3 &= 0xFF;
        c4 &= 0xFF;
        if( c1&0x80 ) iin = (double)(-1*(1<<15) + ((c1&0x7F)<<8) + c2);
        else iin = (double)(((c1&0x7F)<<8) + c2);
        if( c3&0x80 ) qin = (double)(-1*(1<<15) + ((c3&0x7F)<<8) + c4);
        else qin = (double)(((c3&0x7F)<<8) + c4);
        iint = (int)(iin);
        qint = (int)(qin);
        iint = iint >> 4;
        qint = qint >> 4;
        iq_pairs.push_back( std::pair<int, int>(iint, qint) );
    };
    fclose(fpin);
    return 0;
}

int main(int argc, char** argv)
{
    if(argc < 2)
```

```
    {
        cout << "wfm filename required\n";
        return 0;
    }
    vector< pair<int,int> > iq_pairs;
    if(ReadWFM(argv[1], iq_pairs) != 0)
    {
        cout << "Error reading wfm file\n" << endl;
        return -1;
    }

    const long dataBufLength = iq_pairs.size()*2*sizeof(short);
    unsigned char *dataBuf = new unsigned char[dataBufLength];
    //construct data buffer
    int bufPos = 0;
    for(int i=0; i<iq_pairs.size() && bufPos < dataBufLength; ++i)
    {
        short i_sample = iq_pairs[i].first;
        short q_sample = iq_pairs[i].second;
        dataBuf[bufPos]   = i_sample & 0xFF; // I LSB
        dataBuf[bufPos+1] = (i_sample>>8) & 0x0F; // iq select 0, I MSB
        dataBuf[bufPos+2] = q_sample & 0xFF; // Q LSB
        dataBuf[bufPos+3] = 0x10 | ((q_sample>>8) & 0x0F); // iq select 1, Q MSB
        bufPos += 4;
    }

    CCyUSBDevice USBDevice;
    //end points for spi communications
    CCyControlEndPoint *InCtrlEndPt = NULL;
    CCyControlEndPoint *OutCtrlEndPt = NULL;
    //end point for samples uploading
    CCyUSBEndPoint *OutEndPt = NULL;

    if( USBDevice.DeviceCount() == 0)
    {
        cout << "USB device not found\n";
        return -1;
    }
    //expecting that only one device is connected
    else if(USBDevice.Open(0))
    {
        InCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);
        OutCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);
        InCtrlEndPt->ReqCode = CTR_R_REQCODE;
        InCtrlEndPt->Value = CTR_R_VALUE;
        InCtrlEndPt->Index = CTR_R_INDEX;
        OutCtrlEndPt->ReqCode = CTR_W_REQCODE;
        OutCtrlEndPt->Value = CTR_W_VALUE;
        OutCtrlEndPt->Index = CTR_W_INDEX;

        for (int i=0; i<USBDevice.EndPointCount(); ++i)
        {
            if(USBDevice.EndPoints[i]->Address == 0x01)
            {
                OutEndPt = USBDevice.EndPoints[i];
                long len = OutEndPt->MaxPktSize * 64;
                OutEndPt->SetXferSize(len);
                break;
            }
        }
    }
    else
    {
        cout << "Failed to open USB device\n";
        return -2;
    }

    if(InCtrlEndPt && OutCtrlEndPt && OutEndPt)
```

```
    {
        unsigned char ctrbuf[64];
        unsigned char inbuf[64];
        long len = 64;
        //Read memory register
        memset(ctrbuf, 0, 64);
        memset(inbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_RD;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        memset(ctrbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_WR;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        ctrbuf[10] = inbuf[10];
        //enable STREAM_LOAD, disable STREAM_TXEN, STREAM_RXEN
        ctrbuf[11] = inbuf[11] & ~0x7;
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        //transfer data buffer to board
        long bToSend = dataBufLength;
        OutEndPt->XferData(dataBuf, bToSend);

        //Read memory register
        memset(ctrbuf, 0, 64);
        memset(inbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_RD;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        memset(ctrbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_WR;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        ctrbuf[10] = inbuf[10];
        ctrbuf[11] = inbuf[11] | 0x3; //disable STREAM_LOAD, enable STREAM_TXEN
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);
    }
    else
    {
        cout << "Failed to find USB end points\n";
        return -3;
    }
    return 0;
}
```

1.10.2 IQ Samples reading example

This program is written in C++ on Windows operating system. USB communications are performed using CyAPI library, it can be downloaded from EZ-USB FX3 Software Development Kit for Windows. The program reads single buffer of IQ samples from Stream board.

```
#include <windows.h>
#include "CyAPI/CyAPI.h"
#include <iostream>
#include <vector>

#define CTR_W_REQCODE 0xC1
#define CTR_W_VALUE 0x0000
#define CTR_W_INDEX 0x0000

#define CTR_R_REQCODE 0xC0
#define CTR_R_VALUE 0x0000
#define CTR_R_INDEX 0x0000
#define CMD_BRDSPI_WR 0x55//16 bit spi for stream
#define CMD_BRDSPI_RD 0x56//16 bit spi for stream
using namespace std;

int main(int argc, char** argv)
{
    CCyUSBDevice USBDevice;
    //end points for spi communications
    CCyControlEndPoint *InCtrlEndPt = NULL;
    CCyControlEndPoint *OutCtrlEndPt = NULL;
    //end point for samples uploading
    CCyUSBEndPoint *InEndPt = NULL;

    if( USBDevice.DeviceCount() == 0)
    {
        cout << "USB device not found\n";
        return -1;
    }
    //expecting that only one device is connected
    else if(USBDevice.Open(0))
    {
        InCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);
        OutCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);
        InCtrlEndPt->ReqCode = CTR_R_REQCODE;
        InCtrlEndPt->Value = CTR_R_VALUE;
        InCtrlEndPt->Index = CTR_R_INDEX;
        OutCtrlEndPt->ReqCode = CTR_W_REQCODE;
        OutCtrlEndPt->Value = CTR_W_VALUE;
        OutCtrlEndPt->Index = CTR_W_INDEX;

        for (int i=0; i<USBDevice.EndPointCount(); ++i)
        {
            if(USBDevice.EndPoints[i]->Address == 0x81)
            {
                InEndPt = USBDevice.EndPoints[i];
                long len = InEndPt->MaxPktSize * 64;
                InEndPt->SetXferSize(len);
                break;
            }
        }
    }
    else
    {
        cout << "Failed to open USB device\n";
        return -2;
    }
```

```
    if(InCtrlEndPt && OutCtrlEndPt && InEndPt)
    {
        const int buffer_size = 4096*64; // single transaction size
        unsigned char *buffer = NULL;
        buffer = new unsigned char[buffer_size];
        if(buffer == 0)
        {
            cout << "error allocating buffers\n";
            return -4;
        }
        memset(buffer, 0, buffer_size);
        unsigned char ctrbuf[64];
        unsigned char inbuf[64];
        long len = 64;
        //Read memory register
        memset(ctrbuf, 0, 64);
        memset(inbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_RD;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        memset(ctrbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_WR;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        ctrbuf[10] = 0;
        ctrbuf[11] = inbuf[11] | 0x4; //Enable STREAM_RXEN
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        //read IQ samples to buffer
        long bToSend = buffer_size;
        InEndPt->XferData(buffer, bToSend); //this is blocking operation, if board
         //is not sending any data, program will be stuck at this point

        short tempInt = 0;
        short* buf;
        buf = (short*)buffer;

        vector<short> iq_samples;
        int framestart = 0; // can be either 1 or 0, depending from chip parameters
        //frame start might change it's position in data buffer, so it is
        //needed to find the start of the first frame for each transfer
        unsigned int fs = 0;
        for(fs=0; fs<buffer_size/sizeof(short); ++fs)
        {
            if( ((buf[fs]&0x1000)>0) == framestart)
                break;
            ++fs;
        }
        for(; fs<buffer_size/sizeof(short); ++fs)
        {
            tempInt = buf[fs] & 0x0FFF;
            //shifting 4 bits to left then right, to restore number sign
            tempInt = tempInt << 4;
            tempInt = tempInt >> 4;
            iq_samples.push_back(tempInt);
        }
        delete[] buffer;
```

```
        //Here iq_samples will be ready for further processing
        //Inside the iq_samples, depending on detected frame start position,
        //there might be non equal count of I and Q samples.

        //Read memory register
        memset(ctrbuf, 0, 64);
        memset(inbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_RD;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);

        //write memory register
        memset(ctrbuf, 0, 64);
        ctrbuf[0] = CMD_BRDSPI_WR;
        ctrbuf[1] = 0;
        ctrbuf[2] = 0x01;
        ctrbuf[8] = 0x00;
        ctrbuf[9] = 0x05;
        ctrbuf[10] = 0;
        ctrbuf[11] = inbuf[11] & ~0x4; // Disable STREAM_RXEN
        len = 64;
        OutCtrlEndPt->Write(ctrbuf, len);
        len = 64;
        InCtrlEndPt->Read(inbuf, len);
    }
    else
    {
        cout << "Failed to find USB end points\n";
        return -3;
    }
    return 0;
}
```

1.10.3 FPGA programming example

This program is written in C++ on Windows operating system. USB communications are performed using CyAPI library, it can be downloaded from EZ-USB FX3 Software Development Kit for Windows. The program reads given RBF file, and uploads it to FPGA.

```
#include <windows.h>
#include "CyAPI/CyAPI.h"
#include <iostream>
#include <fstream>
#include <vector>

#define CTR_W_REQCODE 0xC1
#define CTR_W_VALUE 0x0000
#define CTR_W_INDEX 0x0000

#define CTR_R_REQCODE 0xC0
#define CTR_R_VALUE 0x0000
#define CTR_R_INDEX 0x0000

#define STATUS_COMPLETED_CMD 0x01
#define BITSTREAM_FILE "bitstream.rbf"
using namespace std;

int main(int argc, char** argv)
{
```

```
    CCyUSBDevice USBDevice;
    //end points for spi communications
    CCyControlEndPoint *InCtrlEndPt = NULL;
    CCyControlEndPoint *OutCtrlEndPt = NULL;

    if( USBDevice.DeviceCount() == 0)
    {
        cout << "USB device not found\n";
        return -1;
    }
    //expecting that only one device is connected
    else if(USBDevice.Open(0))
    {
        InCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);
        OutCtrlEndPt = new CCyControlEndPoint(*USBDevice.ControlEndPt);

        InCtrlEndPt->ReqCode = CTR_R_REQCODE;
        InCtrlEndPt->Value = CTR_R_VALUE;
        InCtrlEndPt->Index = CTR_R_INDEX;
        OutCtrlEndPt->ReqCode = CTR_W_REQCODE;
        OutCtrlEndPt->Value = CTR_W_VALUE;
        OutCtrlEndPt->Index = CTR_W_INDEX;
    }
    else
    {
        cout << "Failed to open USB device\n";
        return -2;
    }

    if(InCtrlEndPt && OutCtrlEndPt)
    {
        unsigned char *m_data = NULL; //bitstream data
        long m_data_size = 0; // bitstream file size
        //First load bitstream file into memory
        fstream fin;
        fin.open(BITSTREAM_FILE, ios::in|ios::binary);
        if(!fin.good())
        {
            fin.close();
            cout << "File " << BITSTREAM_FILE << " not found\n";
            return -2;
        }
        fin.seekg(0, ios_base::end);
        m_data_size = fin.tellg();
        m_data = new unsigned char[m_data_size];
        fin.seekg(0, ios_base::beg);
        fin.readsome((char*)m_data, m_data_size); //read file into memory
        fin.close();

        //set programming mode
        // 0-bitstream to FPGA
        // 1-bitstream to FLASH
        // 2-bitstream from FLASH
        int prog_mode = 0;

        //Split bitstream data into 32 byte chunks and send to board
        int pktSize = 32;
        int data_left = m_data_size; //data left to send
        unsigned char* data_src = m_data; //data source pointer

        //total number of packet that will be sent, +1 programming end packet
        int portionsCount = m_data_size/pktSize + (m_data_size%pktSize > 0) + 1;
        int portionNumber; //packet index counter
        int status = 0;

        unsigned char ctrbuf[64];
        unsigned char inbuf[64];
        memset(ctrbuf, 0, 64);
```

```
        ctrbuf[0] = 0x53; // CMD_ALTERA_FPGA_GW_WR = 0x53
        ctrbuf[1] = 0;
        ctrbuf[2] = 56; // indicate that 56 data bytes will be in the packet

        for(portionNumber=0; portionNumber<portionsCount; ++portionNumber)
        {
            int offset = 8; //control packet offset to data field
            memset(&ctrbuf[offset], 0, 56); //set packet data to zeros
            ctrbuf[offset+0] = prog_mode;

            //set set bitstream chunk index
            ctrbuf[offset+1] = (portionNumber >> 24) & 0xFF;
            ctrbuf[offset+2] = (portionNumber >> 16) & 0xFF;
            ctrbuf[offset+3] = (portionNumber >> 8) & 0xFF;
            ctrbuf[offset+4] = portionNumber & 0xFF;

            //set how many bytes of bitstream will be in the packet
            //on the end of programming last packet with data_cnt 0 needs to be sent
            unsigned char data_cnt = data_left > pktSize ? pktSize : data_left;
            ctrbuf[offset+5] = data_cnt;

            memcpy(&ctrbuf[offset+24], data_src, data_cnt); //copy chunk of
bitstream to packet
            data_src+=data_cnt; //advance the source pointer

            cout << "portion: " << portionNumber << "/" << portionsCount << endl;
            //send packet to board
            long len = 64;
            OutCtrlEndPt->Write(ctrbuf, len);
            len = 64;
            memset(inbuf, 0, 64);
            InCtrlEndPt->Read(inbuf, len);

            //calculate data left to send
            data_left -= data_cnt;
            status = inbuf[1]; //check sent packet status
            if(status != STATUS_COMPLETED_CMD)
            {
                //error occoured, terminate programming
                cout << "Programming failed! Status: " << (int)status << endl;
                return -1;
            }
        }
        cout << "Programming finished, " << m_data_size << " bytes sent!\n";
        delete []m_data;
        return 0;
    }
    else
    {
        cout << "Failed to find USB end points\n";
        return -3;
    }
    return 0;
}
```

3

Stream Board FX3 and FPGA communications

1.1 Introduction

Communication between FX3 USB microcontroller and FPGA are performed using GPIF II interface configured as slave FIFO with 16 bit width data bus. Implementation is based on AN65974 application note from Cypress.

1.2 GPIF II related connections between FX3 and FPGA

FX3 and FPGA GPIF II related connections are as shown in Figure 1.

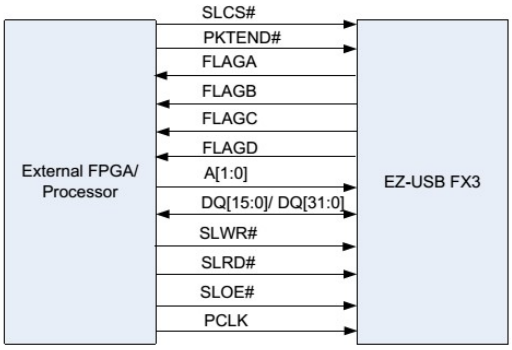

Figure 1 FX3 and FPGA GPIF II related connection

Detailed description of the signals is provided in Table 5.

Table 5 FX3 and FPGA GPIF II related connection details

Signal Name	Schematic Signal Name	FPGA Pin	Signal Description
SLCS#	FX3_CTL[0]	L7	Chip select signal for the Slave FIFO interface. It must be asserted to access Slave FIFO.
SLWR#	FX3_CTL[1]	M4	Write strobe for the Slave FIFO interface. It must be asserted for performing write transfers to Slave FIFO.

Signal Name	Schematic Signal Name	FPGA Pin	Signal Description
SLRD#	FX3_CTL[3]	M7	Read strobe for the Slave FIFO interface. It must be asserted for performing read transfers from Slave FIFO.
SLOE#	FX3_CTL[2]	M8	Output enable signal. It causes the data bus of the Slave FIFO interface to be driven by FX3. It must be asserted for performing read transfers from Slave FIFO.
FLAGA, FLAGB, FLAGC, FLAGD	FX3_CTL[4], FX3_CTL[5], FX3_CTL[6], FX3_CTL[8]	M5, P7, M3, N8	Flag outputs from FX3. The FLAGs indicate the availability of an FX3 socket. FLAGA, FLAGB are used for Slave FIFO write operation and FLAGC, FLAGD are used for Slave FIFO read operation.
A[1:0]	FX3_CTL[11], FX3_CTL[12]	N7, N5	2-bit address bus of Slave FIFO.
DQ[15:0]	FX3_DQ[0..15]	R2, R1, V2, V1, U2, U1, R5, T3, P4, P5, N2, P2, P1, M1, N1, P3	16-bit data bus of Slave FIFO.
PKTEND#	FX3_CTL[7]	L6	Assert to write a short packet or a zero-length packet to Slave FIFO.
PCLK	FX3_PCLK	M2	Slave FIFO interface clock.

1.3 StreamIN data transfers (from FPGA to FX3)

StreamIN data transmits the data from FPGA to FX3 and then to the PC.

To implement StreamIN data transfers some signals, listed in Table 5, may be connected to the high or low level, as shown in Table 6. note please that active (asserted) signal level is low.

Table 6 Used and unused signals to drive GPIF II interface from FPGA for StreamIN

Signal Name	Connect to	Explanation
SLCS#	Low	Assert to access Slave FIFO.
SLWR#	From FPGA data write logic.	Assert for performing write transfers to Slave FIFO.
SLRD#	High	Read strobe for the Slave FIFO interface is not used for StreamIN transfers.
SLOE#	High	Data are driven from FPGA.
FLAGA	To FPGA data write logic.	FLAGA indicates Slave FIFO full when asserted.
FLAGB	To FPGA data write logic.	FLAGB is partial flag and indicates Slave FIFO water level. Becomes asserted when there is room for 2 words to write.
FLAGC	To FPGA data write logic.	Not used.
FLAGD	To FPGA data write logic.	Not used.
A[1:0]	Low, Low	2-bit address bus of Slave FIFO is "00".
DQ[15:0]	From FPGA data write logic.	16-bit data bus of Slave FIFO.
PKTEND#	High	Do not use a short packet or a zero-length packet feature.
PCLK	From FPGA	Slave FIFO interface clock, provided from FPGA.

Check an application note AN65974 from Cypress, chapter " Design Example 1: Interfacing an Xilinx FPGA to FX3's Synchronous Slave FIFO Interface" for more detailed explanation on how to implement StreamIN interface between FX3 and FPGA.

1.4 StreamOUT data transfers (from FX3 to FPGA)

StreamOUT data transmits the data from PC to FX3 and then to FPGA.

To implement StreamIN data transfers some signals, listed in Table 5, may be connected to the high or low level, as shown in Table 7. note please that active (asserted) signal level is low.

Table 7 Used and unused signals to drive GPIF II interface from FPGA for StreamOUT

Signal Name	Connect to	Explanation
SLCS#	Low	Assert to access Slave FIFO.
SLWR#	High	Write strobe for the Slave FIFO interface is not used for StreamOUT transfers.
SLRD#	From FPGA data read logic.	Assert for performing read transfers from Slave FIFO.
SLOE#	From FPGA data read logic.	Assert when data are driven from FX3.
FLAGA	To FPGA data write logic.	Not used.
FLAGB	To FPGA data write logic.	Not used.
FLAGC	To FPGA data write logic.	FLAGC indicates Slave FIFO empty when asserted.
FLAGD	To FPGA data write logic.	FLAGD is partial flag and indicates Slave FIFO water level. Becomes asserted when there 0 not read words.
A[1:0]	High, High	2-bit address bus of Slave FIFO is "11".
DQ[15:0]	To FPGA data read logic.	16-bit data bus of Slave FIFO.
PKTEND#	High	Do not use a short packet or a zero-length packet feature.
PCLK	From FPGA	Slave FIFO interface clock, provided from FPGA.

Check an application note AN65974 from Cypress, chapter " Design Example 1: Interfacing an Xilinx FPGA to FX3's Synchronous Slave FIFO Interface" for more detailed explanation on how to implement StreamIN interface between FX3 and FPGA.

4

References

1. Lime Microsystems, LMS64C Control Protocol, Description.