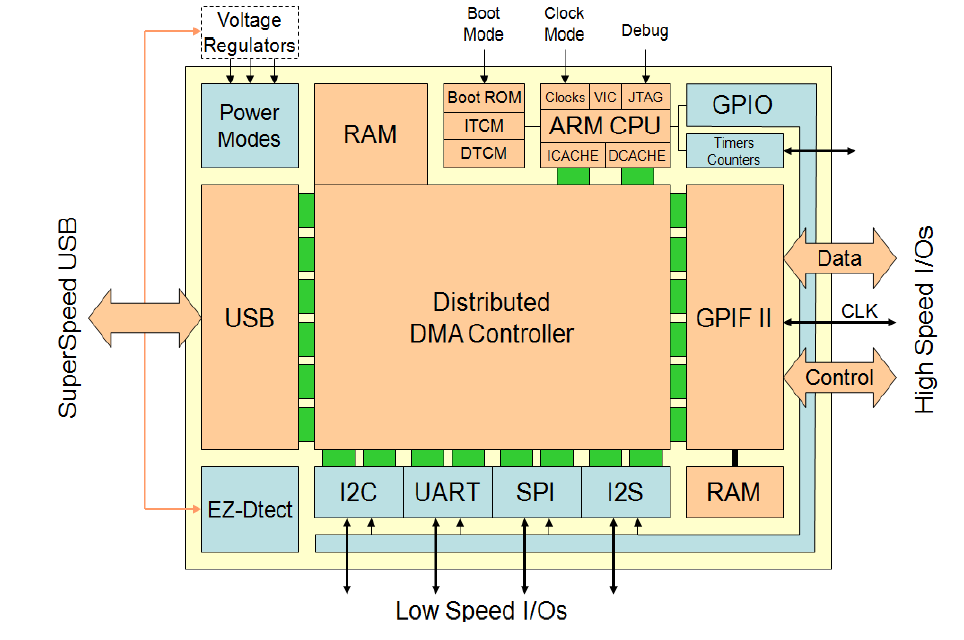
### ARM-RTOS project

Liqing Shen

## Description of the processor



This application utilize a CYUSB3014 ASIC for Superspeed USB application. The USB block provides de-serialization of the USB bus data into endpoints. The distributed DMA buffer in the chip allows the communication between I/O and endpoints without CPU intervention. The center piece of this ASIC is a ARM9 core processor which will be the focus of this project. The firmware of ARM core processor is built upon a multi-threading RTOS kernel which allows each process to be compartmentalized. The chip vendor provides APIs which does all the heavy lifting for programmer to develop firmware.

## Description of I/O resource configuration

The ASIC I/O can be configured either as GPIO or dedicated I/O for GPIF, I2C, I2S, SPI, UART interface. The I/O initialization can be found in main() function in cyfxslfifosync.c

io\_cfg.s0Mode = *CY\_U3P\_SPORT\_INACTIVE*;

io\_cfg.s1Mode = *CY\_U3P\_SPORT\_INACTIVE*;

io\_cfg.useUart = CyTrue;

io\_cfg.useI2C = CyTrue;

io\_cfg.useI2S = CyTrue;

io\_cfg.useSpi = CyTrue;

**#if** (CY\_FX\_SLFIFO\_GPIF\_16\_32BIT\_CONF\_SELECT == 0)

io\_cfg.isDQ32Bit = CyFalse;

io\_cfg.lppMode = *CY\_U3P\_IO\_MATRIX\_LPP\_DEFAULT*;

The above source code disable the USB storage I/O device (since we are not designing a USB storage device), enable Uart (for debug output), I2C (for communicating with EEPROM), SPI (for communicating with FPGA). Besides those serial bus, we also configure a 16 bit parallel bus to stream high speed data to FPGA (the bus protocol can be customized by a dedicated GPIF designer software provided by vendor). The above are dedicated I/O. We also need to enable GPIO 33,35,36,38,39. These I/Os can be customized for special functions such as reading a button or switch state, turning on/off LEDs. In this example, we are using it for a slow speed SPI interface to configure FPGA devices. This allows FPGA firmware be downloaded through USB by PC software application.

The io\_cfg variable is passed to an API to configure I/O

**CyU3PDeviceConfigureIOMatrix** (&io\_cfg);

## The structure of the RTOS code

The main function only configures the I/O and then enter RTOS kernel. This is a non-return function which enable RTOS scheduler to schedule the thread. The thread is created in CyFxApplicationDefine() in cyfxslfifosync.c The code is as follows

CyU3PThreadCreate (&slFifoAppThread, /\* Slave FIFO app thread structure \*/

"21:Slave\_FIFO\_sync", /\* Thread ID and thread name \*/

SlFifoAppThread\_Entry, /\* Slave FIFO app thread entry function \*/

0, /\* No input parameter to thread \*/

ptr, /\* Pointer to the allocated thread stack \*/

CY\_FX\_SLFIFO\_THREAD\_STACK, /\* App Thread stack size \*/

CY\_FX\_SLFIFO\_THREAD\_PRIORITY, /\* App Thread priority \*/

CY\_FX\_SLFIFO\_THREAD\_PRIORITY, /\* App Thread pre-emption \*/

CYU3P\_NO\_TIME\_SLICE, /\* No time slice for the application thread \*/

CYU3P\_AUTO\_START /\* Start the thread immediately \*/

);

The configuration of thread basically allocate the thread in the memory and tell the scheduler the time slice, start condition and priority. The thread process is in SlFifoAppThread Entry() function where initialize the UART function for debugging output and main application involving 16 bit parallel bus and other serial bus, which will be broken down later.

There are other functions such as CyFxSlFifoApplnUSBEventCB() handles USB events. In this function, the main application of the thread will be started or stopped in case of different USB events. The start function of the thread process is defined in CyFxSlFifoApplnStart() which initialize endpoints and DMA for parallel bus and other serial bus data transfer. The stop function is defined in CyFxSlFifoApplnStop () which will flush endpoint and destroy DMA channel created.

There are other functions handling USB events and USB standard and customized request.

CyFxApplnLPMRqtCB() is the functions called when USB enter low power mode (standby). IT is an empty function basically do nothing. CyFxSlFifoApplnUSBSetupCB() handles standard USB request from USB driver for enumeration as well as customized request from PC software application.

Other functions are sub functions of the thread application. cyfxgpif\_syncsf.h file contains the definition of all the macros used in cyfxslfifosync.c program. cyfxgpif\_syncsf.h file is the generated header file by GPIF designer which implements proprietary bus protocol. cyfxslfifousbdscr.c file contains USB device description which has lots of standard USB stuff. Other files are generated automatically by compilers, API licence and RTOS kernel. Modification of those files are prohibited.

## Description of SPI-DMA mode configuration

As mentioned earlier, the DMA bus linked USB endpoints to either serial or parallel bus for data transfer. Since the I/O initialization activates the dedicated I/O for SPI transfer. We need to initialize the SPI module and also create DMA channels for SPI read and write operation. The source code is in CySpiInit() function which is pasted below.

CyU3PSpiConfig\_t spiConfig;

**CyU3PMemSet** ((uint8\_t \*)&spiConfig, 0, **sizeof**(spiConfig));

spiConfig.isLsbFirst = CyFalse;

spiConfig.cpol = CyTrue;

spiConfig.ssnPol = CyFalse;

spiConfig.cpha = CyTrue;

spiConfig.leadTime = *CY\_U3P\_SPI\_SSN\_LAG\_LEAD\_HALF\_CLK*;

spiConfig.lagTime = *CY\_U3P\_SPI\_SSN\_LAG\_LEAD\_HALF\_CLK*;

spiConfig.ssnCtrl = *CY\_U3P\_SPI\_SSN\_CTRL\_FW*; /\*\*< SSN is controlled by API and is not at clock boundaries. \*/

spiConfig.clock = 25000000;

spiConfig.wordLen = 8;

Status = **CyU3PSpiSetConfig** (&spiConfig, NULL);

//CheckStatus ("configure SPI module", Status);

CyU3PDmaChannelConfig\_t dmaConfig;

**CyU3PMemSet** ((uint8\_t \*)&dmaConfig, 0, **sizeof**(dmaConfig)); // No buffers need to be allocated as this channel will be used only in override mode.

dmaConfig.size = pageLen;

dmaConfig.count = 0;

dmaConfig.prodAvailCount = 0;

dmaConfig.dmaMode = *CY\_U3P\_DMA\_MODE\_BYTE*;

dmaConfig.prodHeader = 0;

dmaConfig.prodFooter = 0;

dmaConfig.consHeader = 0;

dmaConfig.notification = 0;

dmaConfig.cb = NULL;

/\* Channel to write to SPI slave. \*/

dmaConfig.prodSckId = *CY\_U3P\_CPU\_SOCKET\_PROD*;

dmaConfig.consSckId = *CY\_U3P\_LPP\_SOCKET\_SPI\_CONS*;

Status = **CyU3PDmaChannelCreate** (&glSpiTxHandle, *CY\_U3P\_DMA\_TYPE\_MANUAL\_OUT*, &dmaConfig);

/\* Channel to read from SPI slave. \*/

dmaConfig.prodSckId = *CY\_U3P\_LPP\_SOCKET\_SPI\_PROD*;

dmaConfig.consSckId = *CY\_U3P\_CPU\_SOCKET\_CONS*;

Status = **CyU3PDmaChannelCreate** (&glSpiRxHandle, *CY\_U3P\_DMA\_TYPE\_MANUAL\_IN*, &dmaConfig);

The SPI is configured most significant bit transfer first and each transfer words is 8 bit long. The SPI clock frequency is set 25MHz. These are important information FPGA designer to develop FPGA RTL logic to interface with the SPI bus.

Two DMA channels created. One for SPI read and the other for SPI write. Both DMA buffer has the same size (256 in this example). The DMA channel transferring data to SPI slave device needs to link CPU producer socket to peripheral’s SPI consumer socket. The DMA channel transferring data from SPI slave device needs link peripheral’s SPI producer socket to CPU consumer socket. To simply put, a socket contains the DMA buffer description, buffer size and buffer starting address for DMA channel to transfer data to the correct memory space defined by the ARM processor. Please refer to FX3 programmer manual for detail description of socket and DMA. After that, two handles are returned (glSpiTxHandle, glSpiRxHandle) which can be passed to other APIs.

After the initialization, CyFxSpiTransfer() is the function to handle SPI transfer between USB to SPI slave device (FPGA). In this application, it allows user to read and write SPI data through the customized USB request defined in CyFxSlFifoApplnUSBSetupCB(). The USB request data is available in the control endpoint 0x01.

CyFxSpiWaitForStatus() is the sub-function of CyFxSpiTransfer() which keeps polling the acknowledgement from SPI slave device after the SPI transfer request is sent. It cannot infinitely wait for the status because it will be stuck forever if SPI slave device is not responding.

## Description of Bulk streaming data auto DMA configuration

In the same manner, the DMA can also link endpoints to parallel data bus for high speed data streaming. The difference here is that it needs to have bulk endpoints for user application to send and received data. (The previous SPI DMA transfer allows user to read and send data using customized USB request, which is available to user application in control endpoint 0x00). So first, we need to create bulk endpoint for software applications to read/write data.

**CyU3PMemSet** ((uint8\_t \*)&epCfg, 0, **sizeof** (epCfg));

epCfg.enable = CyTrue;

epCfg.epType = *CY\_U3P\_USB\_EP\_BULK*;

epCfg.burstLen = 1;

epCfg.streams = 0;

epCfg.pcktSize = size;

/\* Producer endpoint configuration \*/

apiRetStatus = **CyU3PSetEpConfig**(CY\_FX\_EP\_PRODUCER, &epCfg);

/\* Consumer endpoint configuration \*/

apiRetStatus = **CyU3PSetEpConfig**(CY\_FX\_EP\_CONSUMER, &epCfg);

Here we create a producer bulk endpoint and a consumer bulk endpoint. Next we are going to create DMA channel.

dmaCfg.size = 16384;

dmaCfg.count = 4;

dmaCfg.prodSckId = CY\_FX\_PRODUCER\_USB\_SOCKET;

dmaCfg.consSckId = CY\_FX\_CONSUMER\_PPORT\_SOCKET;

dmaCfg.dmaMode = *CY\_U3P\_DMA\_MODE\_BYTE*;

dmaCfg.notification = *0*;

dmaCfg.cb = NULL;

dmaCfg.prodHeader = 0;

dmaCfg.prodFooter = 0;

dmaCfg.consHeader = 0;

dmaCfg.prodAvailCount = 0;

apiRetStatus = **CyU3PDmaChannelCreate** (&glChHandleSlFifoUtoP,

*CY\_U3P\_DMA\_TYPE\_AUTO*, &dmaCfg);

dmaCfg.prodSckId = CY\_FX\_PRODUCER\_PPORT\_SOCKET;

dmaCfg.consSckId = CY\_FX\_CONSUMER\_USB\_SOCKET;

dmaCfg.cb = CyFxSlFifoPtoUDmaCallback;

apiRetStatus = **CyU3PDmaChannelCreate** (&glChHandleSlFifoPtoU,

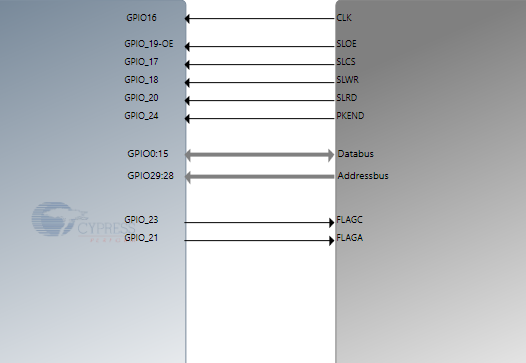
*CY\_U3P\_DMA\_TYPE\_AUTO*, &dmaCfg);

We created two DMA channels. They are both running in automode, which means there will not be a called back function the processor need to handle after each transfer event. We can change it to manual mode and write call back functions to add overhead text for example, to the data buffer. The first DMA channel created connect USB producer socket to processor consumer socket, which means it transfer the data from PC software application to the parallel data bus. The other DMA channel connect USB consumer socket to processor producer socket, which means it transfer the data from parallel data bus to PC software application.

#### Description of GPIF interface

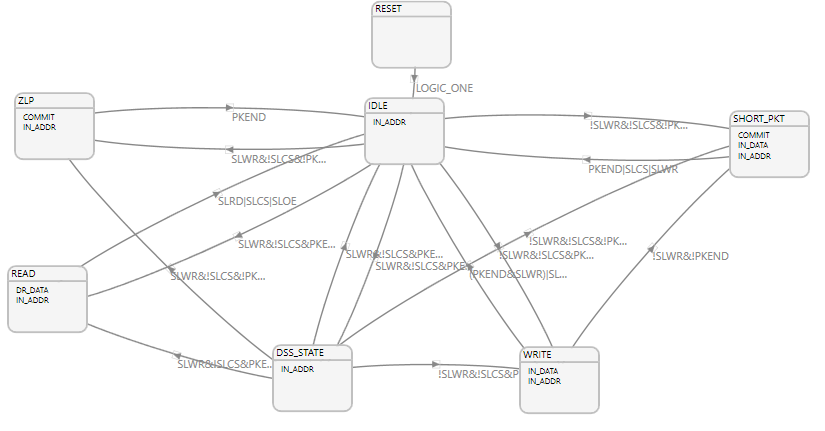
Now that we have created endpoint, the corresponding socket and DMA channel. We need configure the parallel data bus protocol via GPIF designer provided by the development tool and move the data to the socket on the other side of DMA channel.

We can customize the parallel bus protocol depending on the device we are communicating with. The bus connection of this example is shown in the picture below.

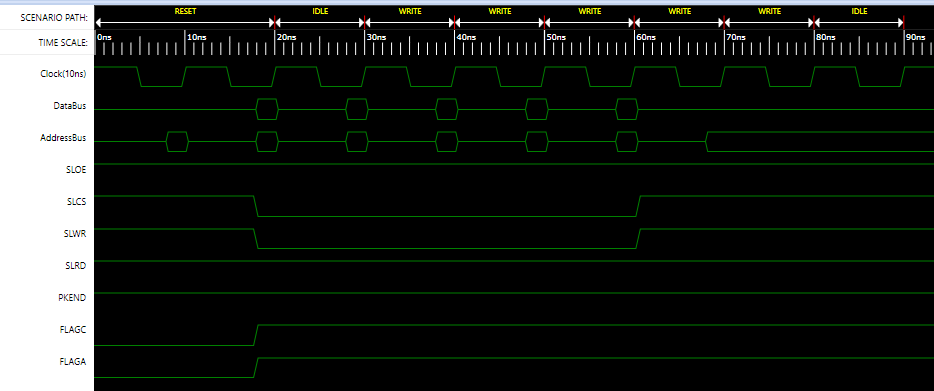


These ARM processor is slave device while the external device is the master device, which issue the CLK signal as well as flow control and addressing signals. The ARM processor also need 2 flags to tell the external device of the empty/full status of internal data buffer.

The bus protocol can be developed using the state machine creator in the tool which is similar to FPGA programming. The state machine flow chart is created and shown below.

What is worth mentioning is the Read state and Write state transfer between external data and internal GPIF thread (It is not the thread defined in software). The thread number is addressed by external processor using address-bus. In this example, the write thread number is 0 and the read thread number is 3.

The tool also provides timing simulation to verify the logic designed. The timing of the bus we created is shown as follows



The tool will compile our design and generate header file cyfxgpif\_syncsf.h to be included in the project. Now we need to link the data from data bus to the socket we intended before. (CY\_FX\_PRODUCER\_PPORT\_SOCKET and CY\_FX\_CONSUMER\_PPORT\_SOCKET). The code is given below. It is in the CyFxSlFifoApplnInit() application.

**CyU3PGpifSocketConfigure** (0,CY\_FX\_PRODUCER\_PPORT\_SOCKET,3,CyFalse,1);

**CyU3PGpifSocketConfigure** (3,CY\_FX\_CONSUMER\_PPORT\_SOCKET,3,CyFalse,1);

This map the GPIF thread 0 to CY\_FX\_PRODUCER\_PPORT\_SOCKET and GPIF thread 3 to CY\_FX\_CONSUMER\_PPORT\_SOCKET. It completes the other side the DMA channel we created before and thus the communication path between PC software to the external device is established.

## Description of other functions

There are other functions such as GPIO and I2C. The GPIO is programmed as a second SPI protocol as it is used to transfer the configuration data to FPGA configuration flash. The I2C allows the data transfer between an EPPROM to PC software. Please refer to the source code for detail.

#### Conclusion

The CYUSB3014 de-serialize the USB data to endpoint enumerated by the USB driver which allows data transfer between PC software and CYUSB3014. The internal DMA bus and ARM core processor can link the data to various bus to communicate to the FPGAs, image sensors, and other microprocessors. The example explores SPI DMA transfer, bulk data streaming, GPIF designer and GPIO functions.

## 