This project demonstrate the usage of **CyU3PGpifWriteDataWords()** and **CyU3PGpifReadDataWords()** APIs.

Project Setup:

Two FX3 DVKs were connected back to back over GPIF II interface using inter connection board with one FX3 programmed as master and the other as slave. The bidirectional communication between master and slave is **asynchronous** and makes use of writing to EGRESS_DATA_REGISTER and reading from INGRESS_DATA_REGISTER as opposed to using Sockets and associated buffers. Figure 1 shows the Master-Slave interface signals.

Signal	**Direction & Polarity**	**Purpose**
CE	Master to Slave, Active low	Chip Enable. For the Master to read from or write to slave, it has to keep this signal asserted
WE	Master to Slave, Active low	Write Enable. Data from EGRESS_DATA_REGISTER in master is driven into the data bus when CE is kept asserted and WE transitions from Low to High
RE	Master to Slave, Active low	Read Enable. Data From EGRESS_DATA_REGISTER in slave is driven into the data bus when CE is asserted and RE transitions from Low to High.
MDONE	Master to Slave, Active low	Master asserts this signal after driving the last word of data onto the data bus. The Slave on detecting this signal, generates an interrupt and commits all the read data to its USB end.
WriteFlag	Slave to Master, Active low	The Slave asserts this signal low after master asserts WE and CE.The Master can drive the next word into the data bus only when this signal is deasserted (This is to ensure that master doesn't start a new write cycle before Slave completes its read cycle.)
ReadFlag	Slave to Master, Active low	Master cannot read data from data bus when this signal is asserted. (This is again similar to Writeflag except that this ensures flow control in the Slave Write – Master Read cycle)
SDONE	Slave to Master, Active High	Slave asserts this signal after driving the last word of data onto the data bus. The Master on detecting this signal, generates an interrupt and commits all the read data to its USB end. *(There is no specific reason behind having this signal as Active high. The user can have it configured as active low as well and accordingly complementing all the check*

		conditions for this signal in the state machine as well.)

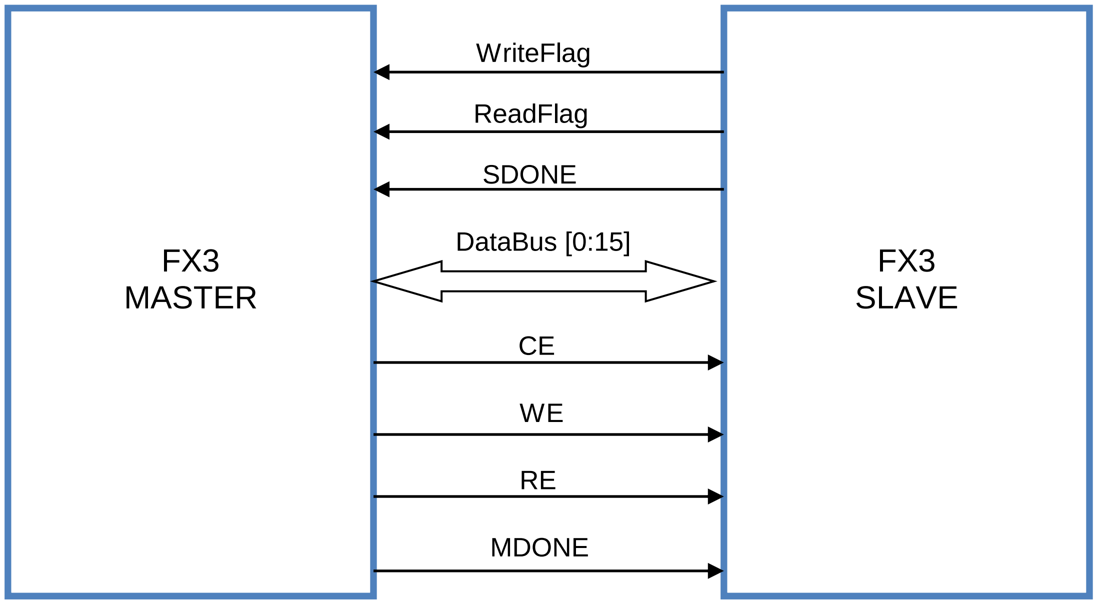


Fig 1: Master-Slave Asynchronous Interface

Note: In the demo project, we have used 16 bit Data bus while Data being put into the bus is 32 bit from EGRESS register. We don't see data loss however because, the data that we write to the EGRESS register is actually a byte value which is converted to uint_32 (padded with 24 zeros in the front). Added benefit of choosing lower bus width is availability of few extra pins which can be used as GPIOs. Data bus width has to be accordingly changed based on the size of data being written to the GPIF EGRESS_DATA_REGISTER i.e. if you are writing a 32 bit value to the register using CyU3PGpifWriteDataWords API, the data bus width has to be chosen as 32 bit too to avoid data loss.

Firmware Flow:

Firmware wise, both master and Slave are the same except for their state machines.

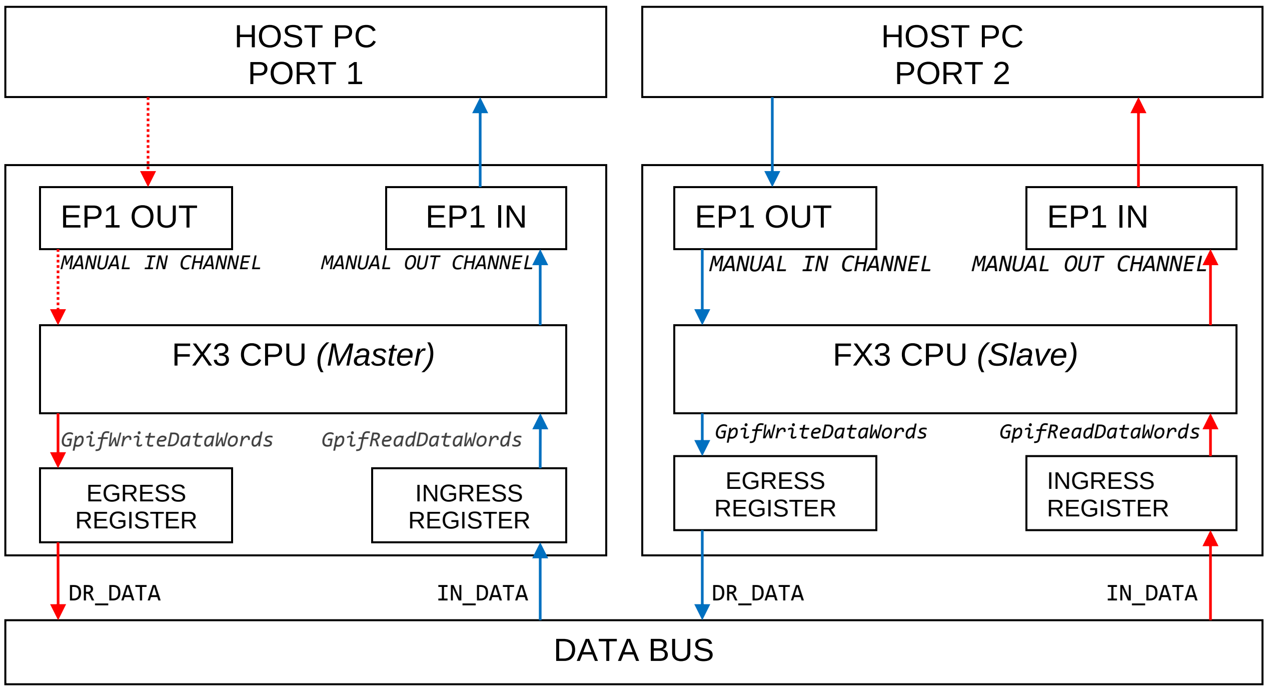


Fig 2: Bidirectional Data transfer.

* Red arrows for Master to Slave .
*Blue arrows for Slave to Master.

Lets look at data transfer from host → master → slave → host. Data transfer from host → slave → master → host is similar.

- In the master side, configure a CY_U3P_DMA_TYPE_MANUAL_IN channel between UIB (producer) and CPU(consumer) with 1 buffer of size 512 bytes or 1024 bytes depending on High speed or Super speed port used (buffer count and buffer size can be chosen as per need). The channel is also configured with a callback enabled. On Producer event, ie whenever host sends data to USB end (sent from control center), the callback function is executed. Inside the callback function, the GPIF data counter is initialized with count limit = total no of bytes received from host. The purpose of this counter is to keep track of how many write cycles are required (one for each byte of data) and when count hit occurs, MDONE gpio is driven.

- In the Slave side, configure a CY_U3P_DMA_TYPE_MANUAL_OUT channel between CPU(producer) and UIB(consumer) with 1 buffer of size 512 bytes or 1024 bytes depending on High speed or Super speed port used (buffer count and buffer size can be chosen as per need). In the **SlFifoAppThread_Entry** function, the register read operation is done using **CyU3PGpifReadDataWords()** repeatedly in a ‘infinite for loop’ and the read values stored in a globally defined array **tmp_data[512]. tmp_data** size again can be changed as needed. When

Slave reads the MDONE signal to be asserted (active low), then it generates a interrupt CPU operation (INTR_CPU) and the contents of tmp_data are manually committed to USB end by the CPU through GPIF registered callback function in firmware.

- This is how the API **CyU3PReturnStatus_t CyU3PGpifWriteDataWords (uint32_t threadIndex, CyBool_t selectThread, uint32_t numWords, uint32_t buffer_p, uint32 t waitOption)** works.

  Step 1: Check for valid thread number (0,1,2 or 3) in **threadIndex.**
  Step 2: If the thread has to be activated, check if software based thread selection is permitted (if **selectThread** is set to Cytrue**)** and do so.
  Step 3: If the register is already free, we can write the first word without waiting for an event decrement **numWords** by 1
  Step 4: If numWords after decrement is non-zero or if register is not free in step 3,
  If (numWords)
  {
  Step 4.1: Make sure that the EG_DATA_EMPTY interrupt for this thread has been enabled.
  Step 4.2: For each word, wait for the empty event (wait period specified in **waitOption**) and then copy the data. If time out occurs before copying the data to register, return with error status.
  Step 4.3: Write the data into the register and then update the data valid flag.
  Step 4.4: Decrement **numWords.** If **numWords** is non-zero, go to beginning of step 4.
  Step 4.5: return CY_U3P_SUCCESS
  }

- This is how the API **CyU3PReturnStatus_t CyU3PGpifReadDataWords (uint32_t threadIndex, CyBool_t selectThread, uint32_t numWords, uint32_t buffer_p, uint32 t waitOption)** works.

  Step 1 and Step 2 same as above.
  Step 3: If the register already has data, read the first word from it directly without waiting for an event. Decrement **numWords** by 1.
  Step 4: If numWords after decrement is non-zero or if register is empty in step 3,
  If (numWords)
  {
  Step 4.1: Make sure that the IN_DATA_READY interrupt for this thread has been enabled.
  Step 4.2: For each word, wait for the data ready event and copy the data into the buffer. If time out occurs before copying the data to register, return with error status.
  Step 4.3: Read the data into the buffer and clear the data valid flag.

Step 4.4: Decrement **numWords.** If **numWords** is non-zero, go to beginning of step 4.
Step 4.5: return CY_U3P_SUCCESS
}

State machines (If the transition equations are not clear, refer the attached project)
Also to find the required setting for each action, double click on the action in the GPIF II designer project attached with the MEMO.

Part 1: Master Write - Slave Read

Slave Side:

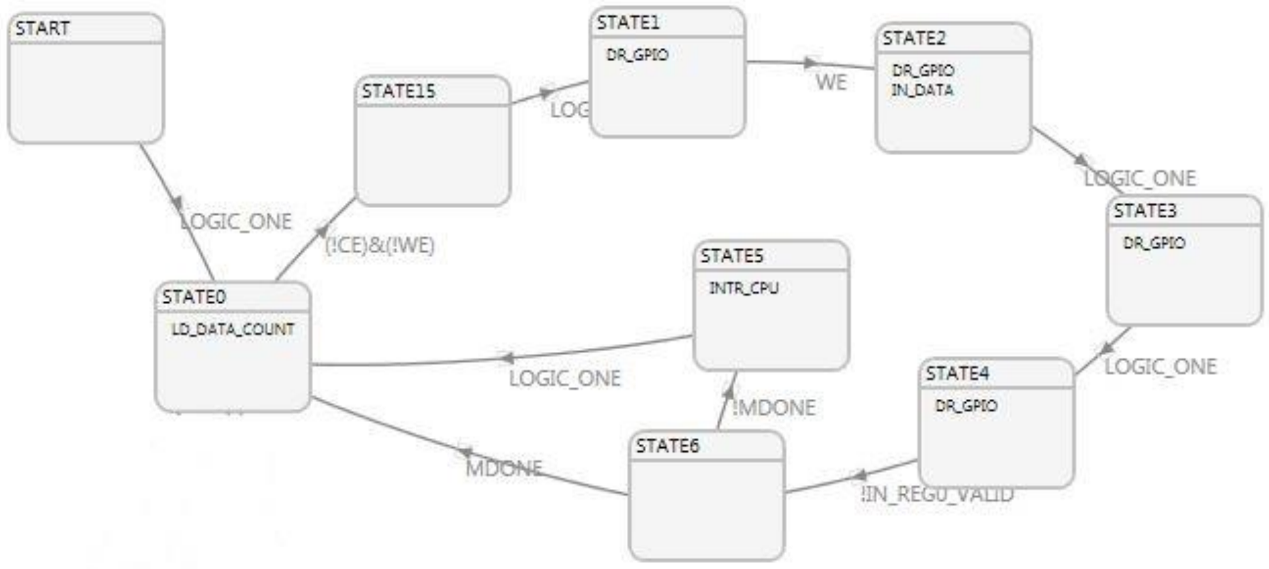


Fig 3: Master Write-Slave Read (Slave State machine)

Master Side:

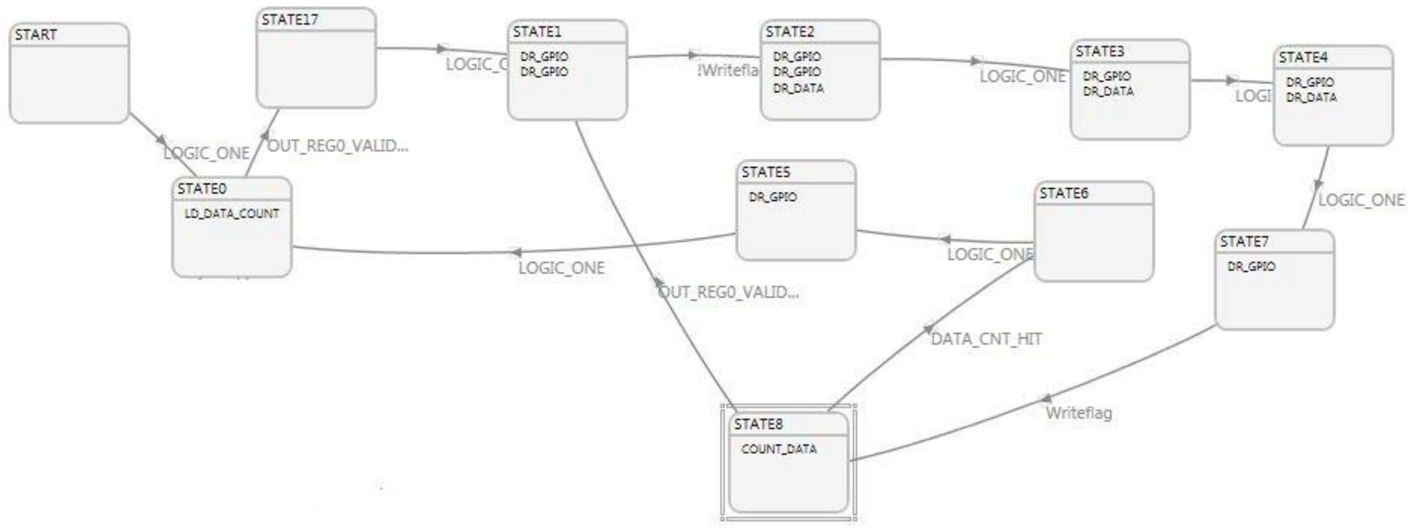


Fig 4: Master Write – Slave Read (Master State machine)

It is recommended to program the master FX3 first and the slave FX3 next .
Reason: If Slave is programmed first, before FX3 master is programmed, CE and WE which are inputs to the Slave are low (the pins were probed by connecting FX3 to FPGA interconnection board and voltage level of all input pins were measured using oscilloscope and found to be low) and the slave leaves its ideal state0 and goes to state1. As soon as master is programmed, it pulls the WE to high and the slave traverses one complete cycle without any valid data being driven by the master.

However, during test, no problem was observed whether slave was programmed first or the master first.

Initially both Master and Slave are in State0.
When Master has Valid data in its register, OUT_REG_VALID flag is high. Master goes to next state which is state17 if WriteFlag is also high in addition to OUT_REG_VALID. WriteFlag is signal from Slave to Master for flow control. It prevents Master to write the second word of data before Slave has finished reading the first word of data from its INGRESS_DATA_REGISTER.

When Master Drives CE AND WE low, Slave goes and waits in State1. Master then drives the data onto the bus and pulls WE high. This transition of WE from low to high makes slave to move to next state and read the data from the bus. When the register value is successfully read within the Slave using CyU3PReadDataWords() API, IN_REG_VALID flag goes low.

In the Master, the data counter is loaded with upper limit being the total number of bytes to be transferred. When DATA_COUNT_HIT occurs, the master drives the signal MDONE low. When MDONE is driven low, the slave generates an GPIF state machine interrupt which causes all the read data which was stored in an array till now to be committed manually to the USB end using MANUAL OUT channel.

Note: when the MDONE signal is asserted low by master as soon as the data count hit occurs, Slave would have not yet read its INGRESS_DATA_REGISTER content. Therefore, an additional GpifReadDataWords() API is called in the GPIF interrupt callback before entire data is committed to the USB end.

Part 2: Slave Write – Master Read

Slave Side

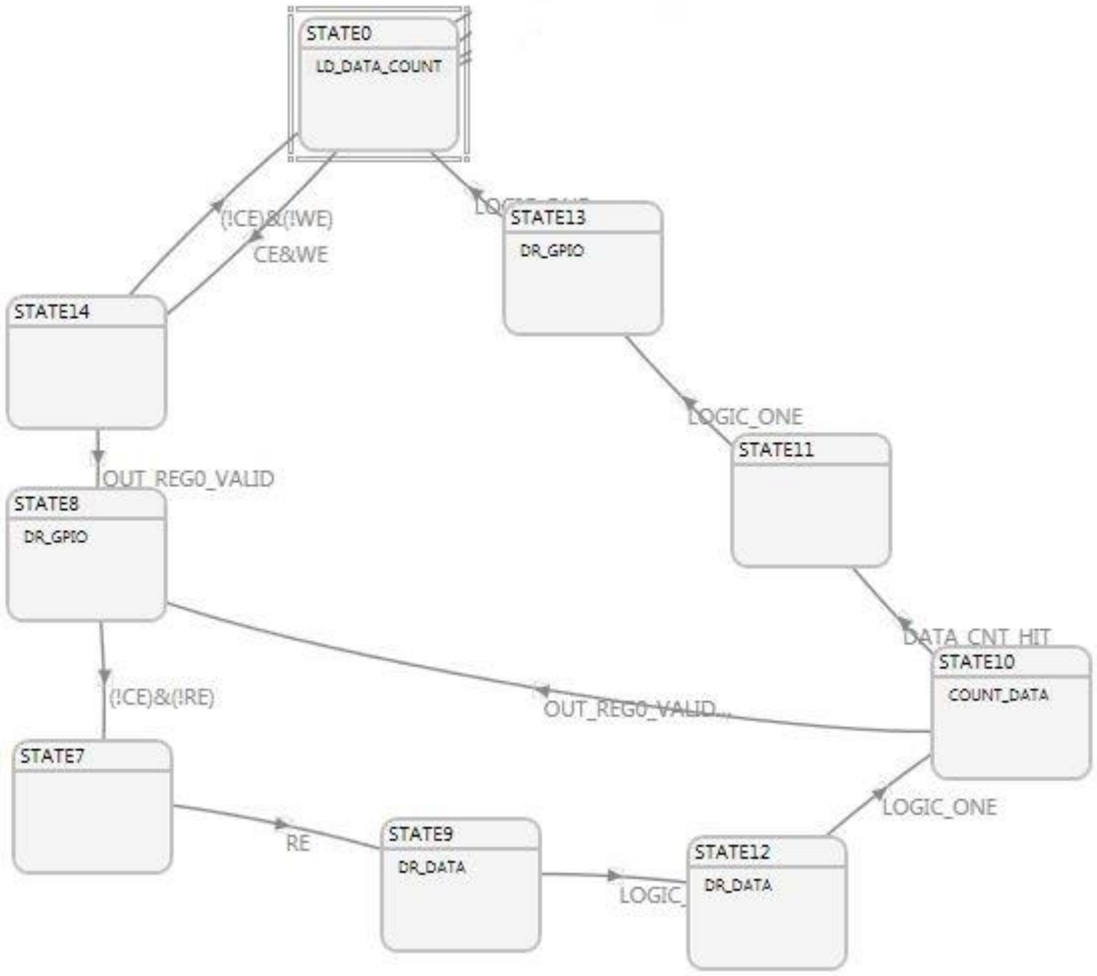


Fig 5: Slave Write – Master Read (Slave state machine)

Master Side

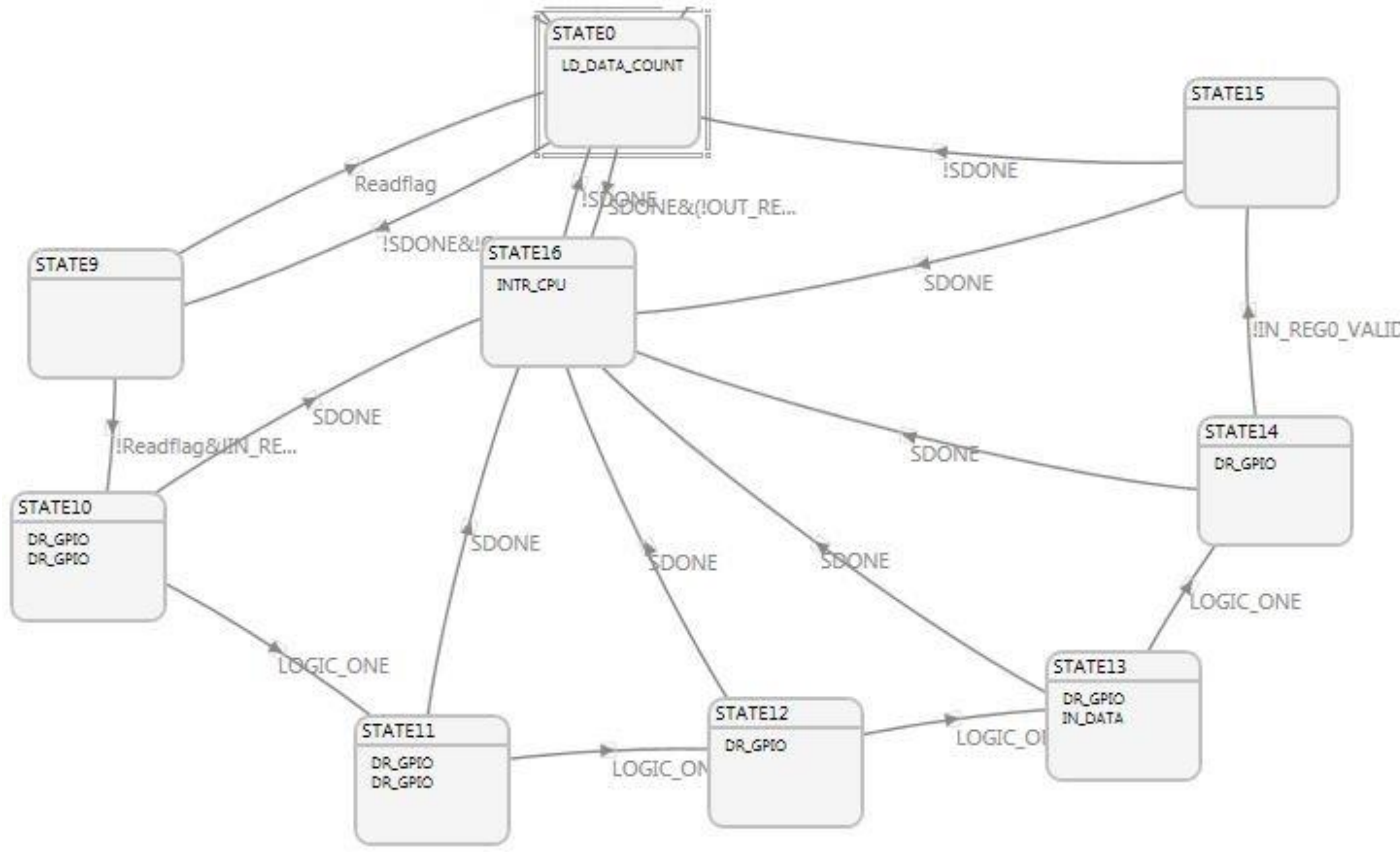


Fig 6: Slave Write – Master Read (Master State machine)

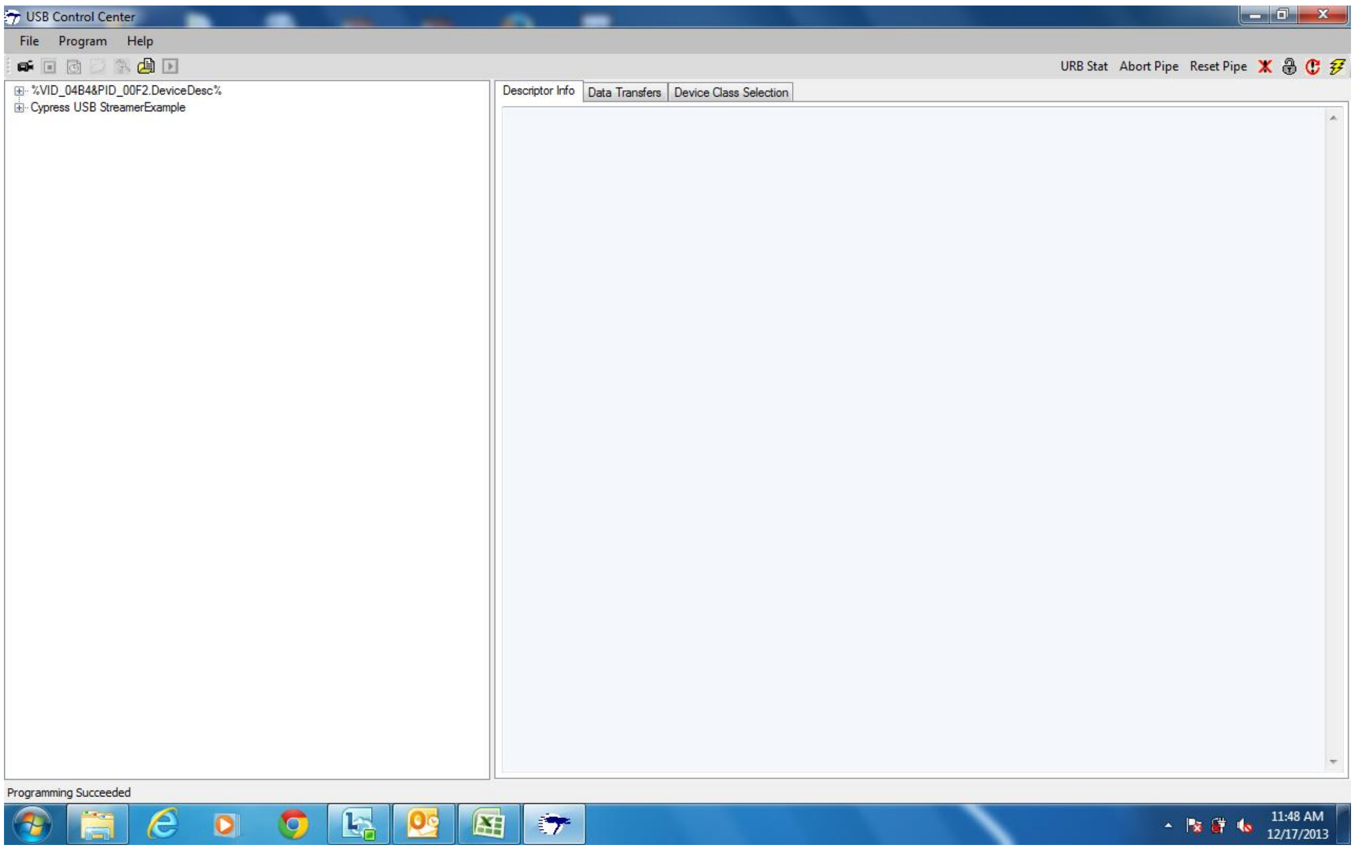
Note: State9 in the master and State14 in the Slave were added to create mirror states as per GPIF II designer requirements.

Initially, both Master and Slave are in state0. Slave will be switching between State0 and State14 till its OUT_REG_VALID flag is set. Slave then drives ReadFlag signal indicating that there is data to be read by the Master. Master which intitially switches between State0 and State9 goes to state10 when ReadFlag goes low and IN_REG_VALID is low (ensuring that previous read is successfully completed). Slave then drives the data onto the bus when CE is low and RE transitions from low to high. When all the data has been driven on the bus, Slave drives SDONE signal (configured as active high). Master on seeing the SDONE go high, generates Interrupt Callback where data is committed to USB end. (Extra GpifReadDataWords() is done in the callback for the same reason as described in Part 1 Master Write – Slave Read Section)

Project Execution:

Download the project files from this link :

- Open the Control Center and load the firmware onto FX3 slave (SlaveFifoAsync-b2b) and master (GPIF_Async) connected using Back to Back interconnect board
- The Slave enumerates with VID/PID = 0x04B4/0x00F2. The master enumerates as “Cypress USB StreamerExample” with VID/PID = 0x04B4/0x00F4. Both master and slave are configured with one IN and one OUT endpoint.

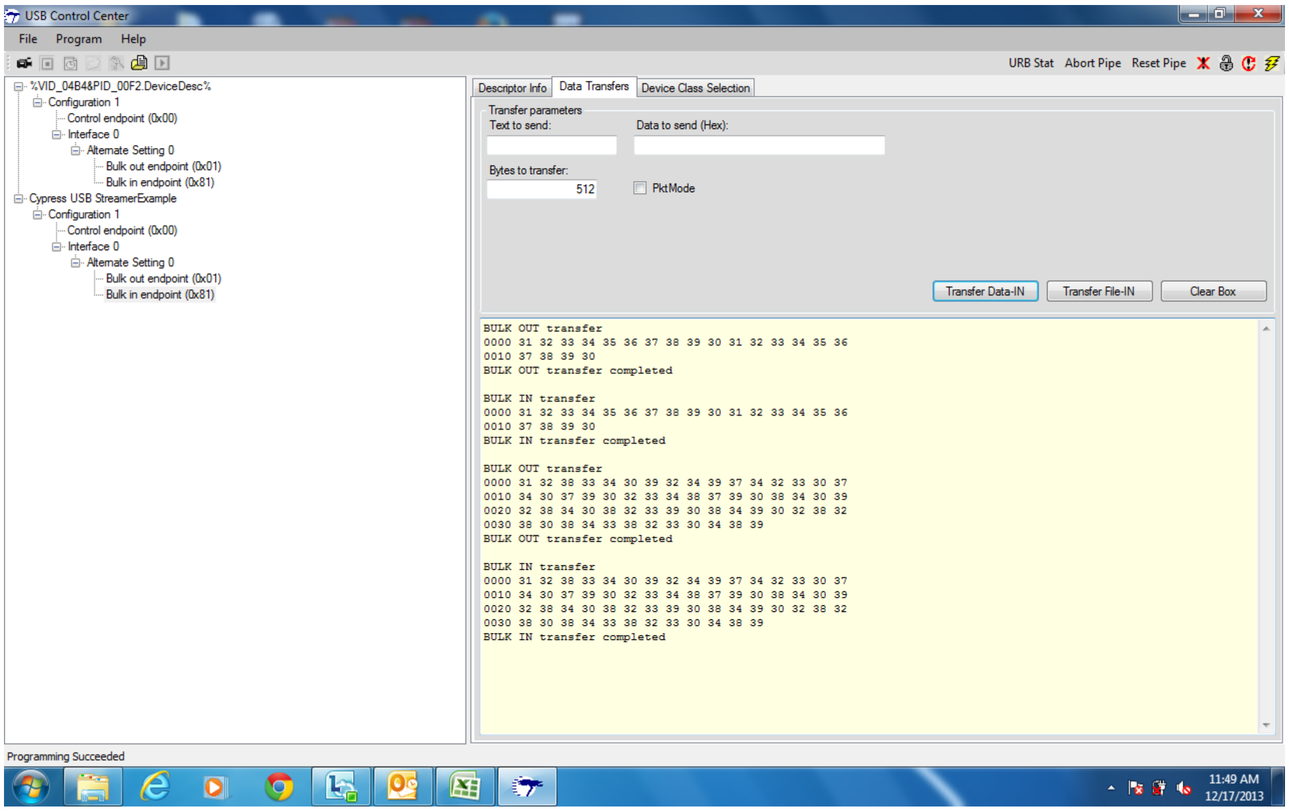


%VID_04B4&PID_00F2.DeviceDesc%
Cypress USB StreamerExample

Programming Succeeded

Fig 7: Master and Slave FX3 successfully programmed

- Select Data Transfer tab. Select Master OUT endpoint (EP1 OUT) and transfer some data. Select Slave IN endpoint (EP1 IN). Read the IN data and check if IN data is same as the OUT data. Repeat the same for Slave OUT and Master IN.



```
%VID_04B4&PID_00F2.DeviceDesc%
  Configuration 1
    Control endpoint (0x00)
    Interface 0
      Alternate Setting 0
        Bulk out endpoint (0x01)
        Bulk in endpoint (0x81)
Cypress USB StreamerExample
  Configuration 1
    Control endpoint (0x00)
    Interface 0
      Alternate Setting 0
        Bulk out endpoint (0x01)
        Bulk in endpoint (0x81)
```

Bytes to transfer: 512

```
BULK OUT transfer
0000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36
0010 37 38 39 30
BULK OUT transfer completed

BULK IN transfer
0000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36
0010 37 38 39 30
BULK IN transfer completed

BULK OUT transfer
0000 31 32 38 33 34 30 39 32 34 39 37 34 32 33 30 37
0010 34 30 37 39 30 32 33 34 38 37 39 30 38 34 30 39
0020 32 38 34 30 38 32 33 39 30 38 34 39 30 32 38 32
0030 38 30 38 34 33 38 32 33 30 34 38 39
BULK OUT transfer completed

BULK IN transfer
0000 31 32 38 33 34 30 39 32 34 39 37 34 32 33 30 37
0010 34 30 37 39 30 32 33 34 38 37 39 30 38 34 30 39
0020 32 38 34 30 38 32 33 39 30 38 34 39 30 32 38 32
0030 38 30 38 34 33 38 32 33 30 34 38 39
BULK IN transfer completed
```

Programming Succeeded

Fig 8: Successful Bidirectional transfers are verified

Note:

Adding UART debug prints indiscriminately in callback functions (be it USB callback or GPIF state machine callback) may lead to FX3 getting stuck or not functioning as desired.

Since the above APIs are related to the state machine design, (WriteDataWords writes to register which is emptied by DR_DATA action in the state machine; IN_DATA action in state machine fills the INGRESS_DATA_REGISTER and ReadDataWords API empties the register content), you need to choose appropriate **waitOption** parameter with the APIs

In the Asynchronous State machine implementation, some states have been repeated to ensure the other device doesn't miss out on that signal (for Eg: the master drives the same data on the bus for few cycles so that Slave can sample it atleast once) and Firmware delay (CyU3PBusyWait) has also been introduced to give sufficient time for ReadDataWords API to successfully read the data.

Both WriteDataWords and ReadDataWords are Slow APIs and state machine should take into consideration that. A lot of delay has been added in the state machines (by repeating the states) to prevent data loss. The state machine is needs to be optimized further by reducing unnecessary delay.