

# TOE100G-IP on Silicom NIC reference design

Rev1.0 16-Dec-22

1	Introduction.....	2
2	UserLogic (Hardware) .....	6
2.1	AxiSSw2to1 .....	8
2.2	TOE100G-IP .....	10
2.3	TxDMA2TOEIF .....	11
2.4	TOE2RxDMAIF .....	16
2.5	RegSwitch .....	23
2.6	TOEDMAReg.....	24
3	The host software .....	27
3.1	Framework.....	28
3.1.1	Device interface .....	28
3.1.2	Shell .....	31
3.2	Application .....	38
3.2.1	Display parameters .....	40
3.2.2	Reset IP .....	41
3.2.3	Half duplex test.....	42
3.2.4	Full duplex test.....	46
3.2.5	Function list in application .....	49
4	Test Software on the target.....	57
4.1	“tcpdatatest” for half duplex test.....	57
4.2	“tcp_client_txrx_single” for full duplex test.....	59
5	Revision History .....	61

# 1 Introduction

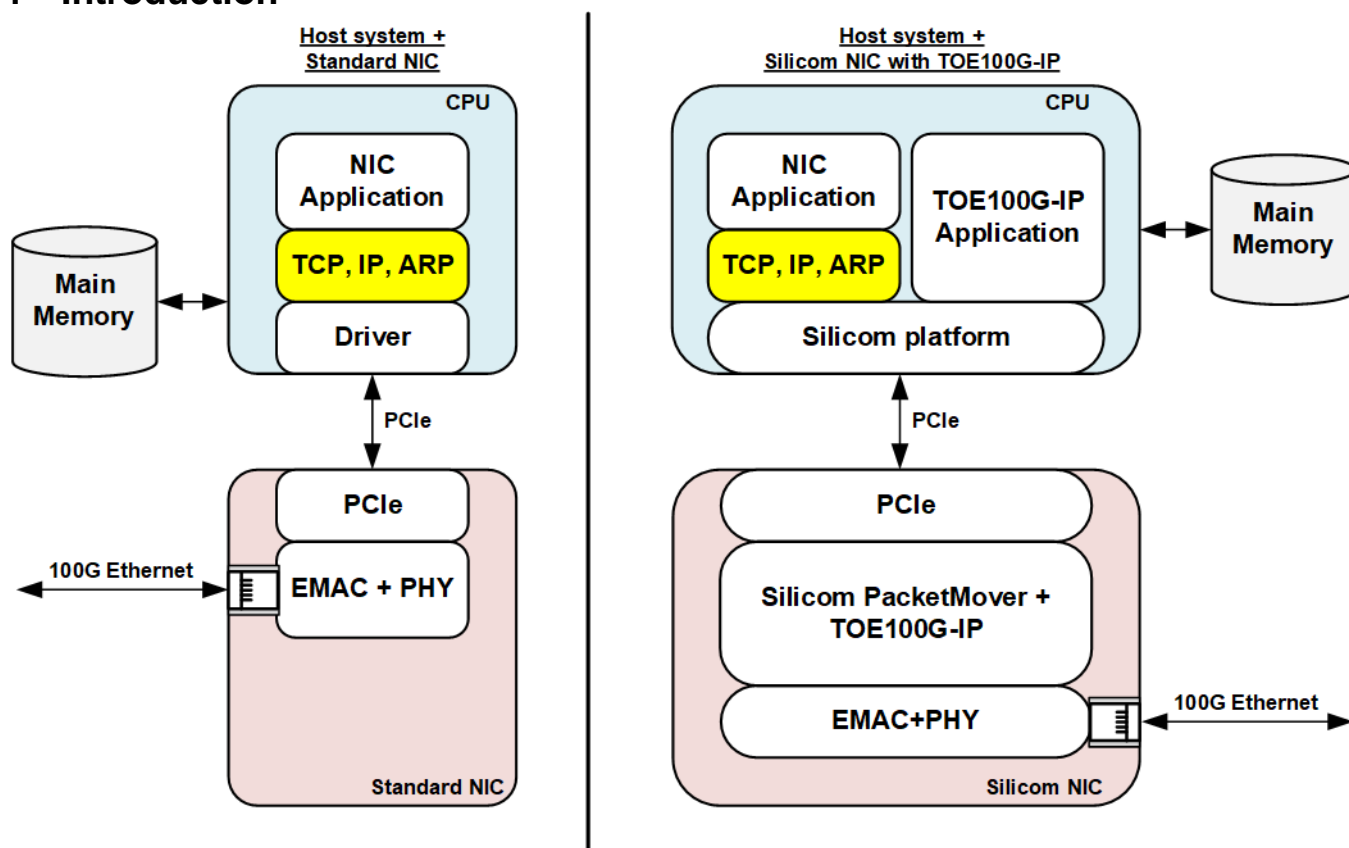


Figure 1-1 Standard NIC and Silicom with TOE100G-IP NIC comparison

The left side of Figure 1-1 shows the host system that plugs-in the standard NIC to PCIe interface. The NIC implements the low-level protocol to handle the Ethernet packet and then transferred to the Main memory via PCIe interface. To control the NIC hardware, the NIC driver provided by the NIC vendor must be run on each OS. The driver is applied by the standard library that implements the standard network protocol such as TCP/IP, UDP, and ARP for building or decoding the Ethernet packet following the standard protocol. Therefore, the application can process the payload data following the requirement without handling the lower-layer protocol. In the system that uses TCP/IP protocol, the Main memory is required for storing TCP payload data or Ethernet packet for transferring data with the Driver. Therefore, CPU has many tasks for handling - process the application layer, TCP protocol, IP protocol, and the driver. Sometimes, ARP protocol must be applied to handle the MAC address table. Consequently, the performance result when running the test application for transferring TCP/IP packet on the standard NIC system shows the limited speed which is much less than the peak bandwidth of 100G Ethernet.

The right side of Figure 1-1 shows the purposed host system that uses Silicom NIC by using FPGA. The FPGA is the programmable logic that can design the offload engine. In this reference design, it implements the full offload engine for TCP/IP protocol by integrating DG TOE100G-IP. Silicom NIC provides the PacketMover framework that applies the full feature of the standard NIC and provides the programmable user logic for customizing the Ethernet packet processor.

This reference design integrates two TOE100G-IPs into the user logic, so TCP/IP packet of two TCP sessions that are controlled by TOE100G-IP can be transferred at the maximum speed of 100G Ethernet (about 1200 MB/s). TOE100G-IP implements TCP/IP protocol of one TCP session by pure-hardware logic. The user data of TOE100G-IP is TCP payload data that is stored to the Main memory via PCIe interface. By using pure-hardware logic, the system does not require the standard library for TCP/IP processing. While other TCP sessions or other protocols are processed by using the standard NIC logic and driver that is provided by Silicom without using TOE100G-IP. Transfer performance by using the standard NIC logic shows less performance. In conclusion, this reference design is the SMART NIC that can configure two high-speed TCP ports for special application while other features of the standard NIC are still supported.

The details of fb2CGhh@KU15P FPGA Card are described in more details from the following site. <https://www.silicom-usa.com/pr/server-adapters/programmable-fpga-server-adapter/fpga-xilinx-based-2/fb2cghhku15p-fpga-card/>

The information about PacketMover framework by Silicom which is the base design can be checked from the following site.

<https://www.silicom.dk/product-details/packetmover-fpga-acceleration/>

*Note: Though TOE100G-IP supports ARP protocol, this reference design forwards the ARP packet to be handled by the standard NIC logic, not TOE100G-IP.*

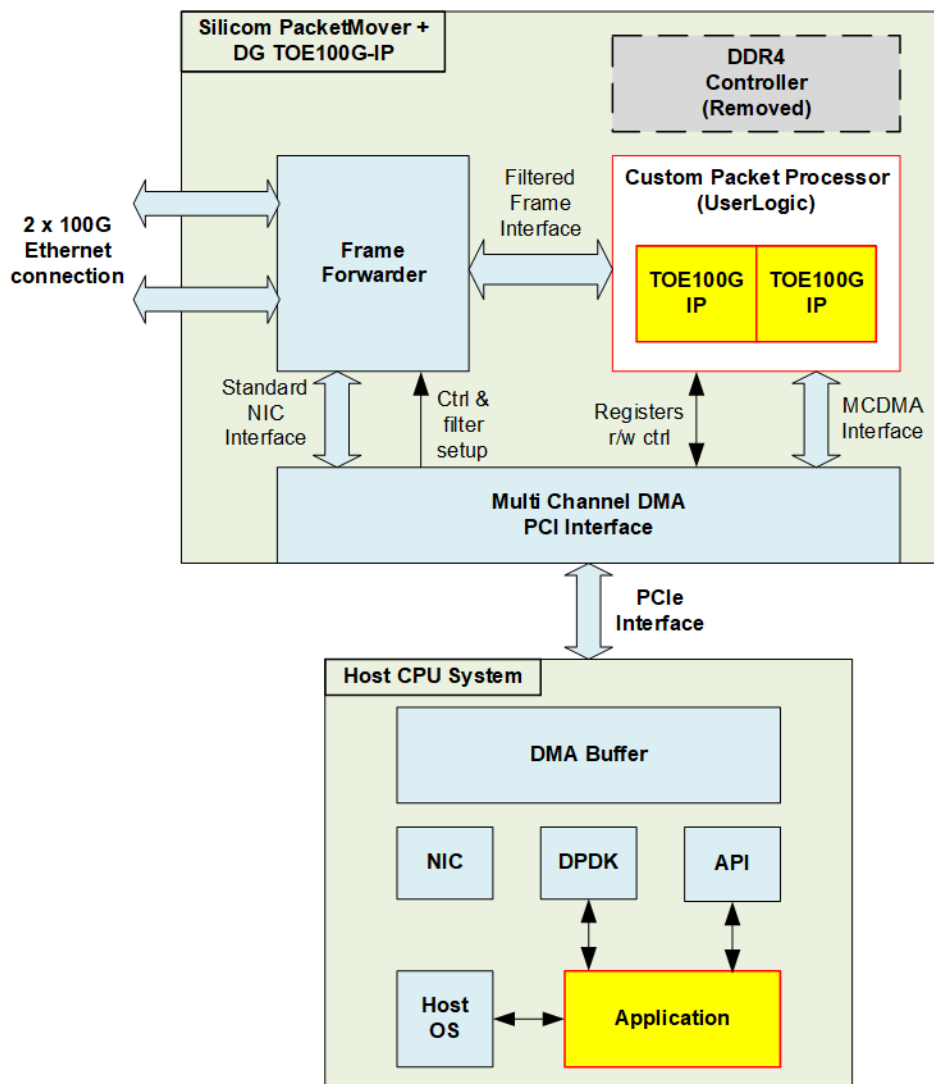


Figure 1-2 Silicom PacketMover with DG TOE100G-IP system

The Silicom PacketMover with DG TOE100G-IP design can be separated to two systems – the hardware system on FPGA card and the host CPU system which runs Ubuntu OS (Linux). The FPGA card connects to the host by using PCIe interface. The PacketMover hardware system is the complete FPGA framework for supporting the standard NIC feature with the Custom Packet Processor. As shown in Figure 1-2, the Frame Forwarder can be configured by the user to determine the destination to forward the received Ethernet packet from 100G Ethernet connection. There are two destinations in the system - Filtered Frame Interface for Custom Packet Processor or Standard NIC Interface. Also, the Frame Forwarder allows the Custom Packet Processor to select the Ethernet connection number (no.0 or no.1 of 100G Ethernet connection) for transmitting Ethernet packet.

Two TOE100G-IPs are applied for processing TCP/IP packet of two TCP sessions. TOE100G-IP is the full offload engine of TCP/IP protocol without CPU and DDR requirement, so DDR4 Controller is removed for resource optimization. The TCP payload data that is user data of TOE100G-IP is transferred with DMA buffer on the host system via MCDMA interface. While the register interface is applied to configure the TOE100G-IP parameters for transferring the Ethernet packet with TCP payload data. Also, this interface is applied to read TOE100G-IP status. The application on the host system is also developed for using two TOE100G-IPs to transfer TCP/IP packet at very high-speed rate.

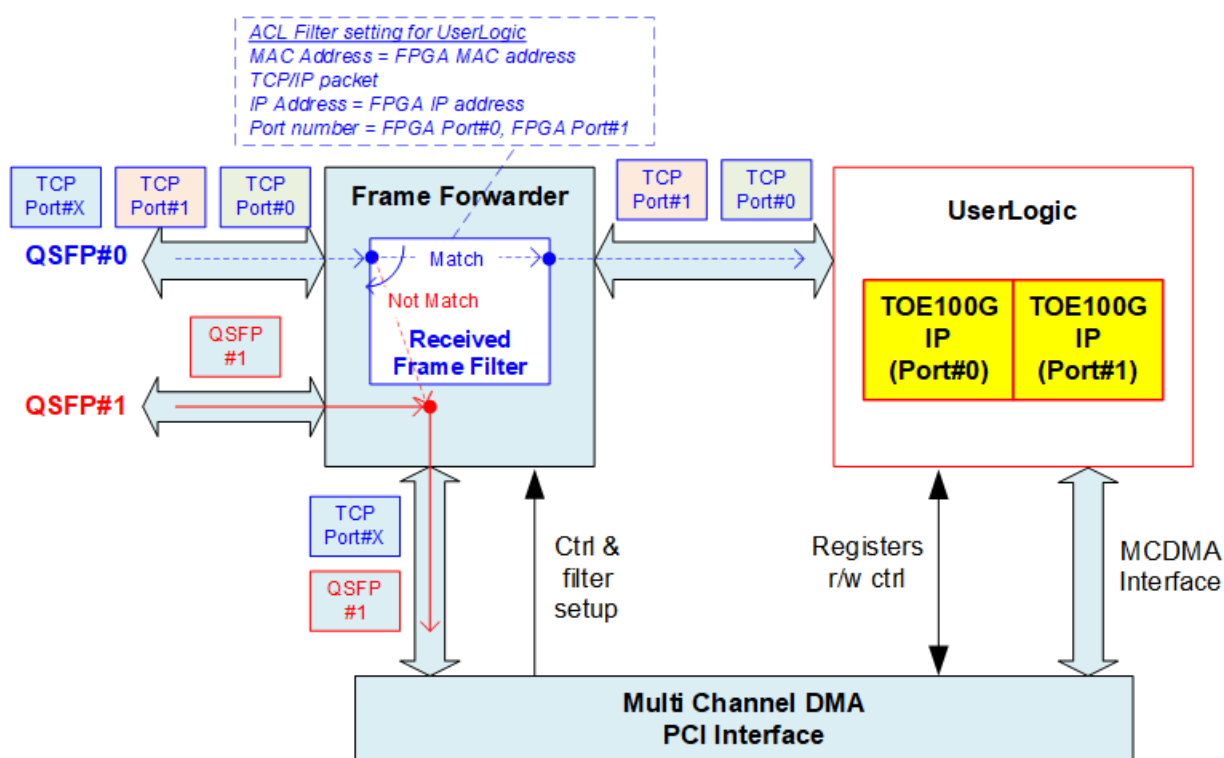


Figure 1-3 ACL Filter setting for UserLogic

Silicom packet mover platform allows the user to configure the received frame filter by setting ACL (Access Control Lists). As shown in Figure 1-3, the rule of this reference design is to forward the received packet of QSFP#0 to the UserLogic (two TOE100G-IPs) when all following conditions are met.

- Target MAC address = FPGA MAC address that is set to TOE100G-IP
- Target IP address = FPGA IP address that is set to TOE100G-IP
- Target Port number = FPGA Port of TOE100G-IP#0 or TOE100G-IP#1
- Protocol type = TCP protocol

If some conditions do not match such as TCP port, the packet is forwarded to Standard NIC interface instead. While all received packets from QSFP#1 connector are forwarded to Standard NIC interface. On the other hand, all transmitted Ethernet packets generated by UserLogic are forwarded to QSFP#0 only while the standard NIC interface has the logic to determine the Ethernet port for forwarding the transmitted packet that is QSFP#0 or QSFP#1. Therefore, all packets that are transferred on QSFP#1 connector are handled by the Standard NIC interface while the packets of two TCP sessions that are transferred on QSFP#0 connector are handled by UserLogic.

Silicom PacketMover with DG TOE100G-IP is run on Ubuntu 20.04 LTS OS and based on Silicom PacketMover Release 1.4.0. To run the demo, the best performance can be achieved when the target system (another side of TOE100G Ethernet connection) is the FPGA card that integrates TOE100G-IP. Up to 12300 MB/s can be achieved by using two FPGA cards with TOE100G-IP transferring data each other. While the performance is much reduced when the target system is the PC with standard NIC. However, the performance by using standard NIC can be improved when using two TCP sessions.

In this document, topic 2 shows the details of the UserLogic hardware design. Topic 3 describes the software implementing on the host CPU system. The last topic is the details of test application on the target system for half-duplex test and full-duplex test.

## 2 UserLogic (Hardware)

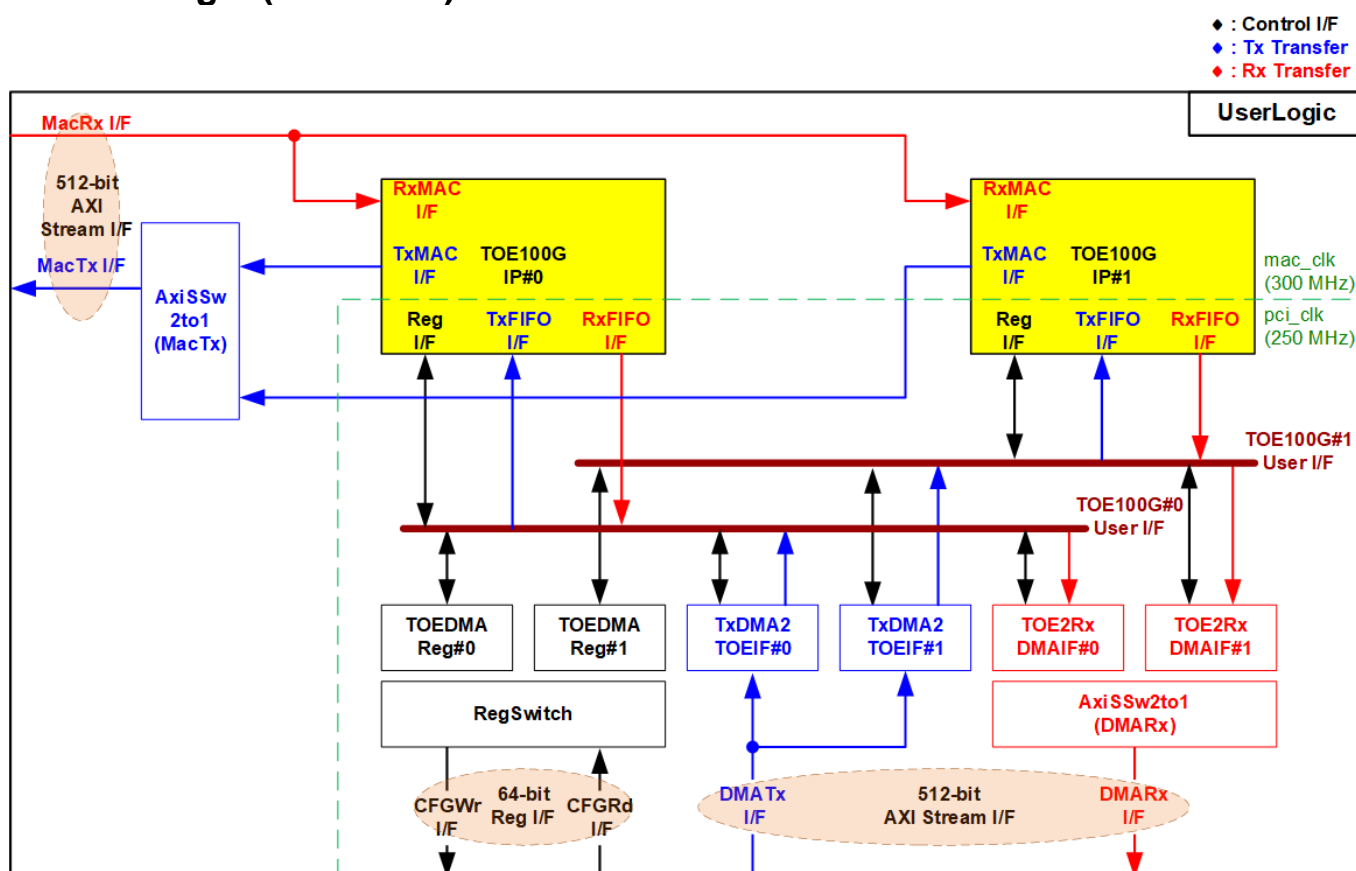


Figure 2-1 UserLogic block diagram

UserLogic has three interfaces. First is 512-bit AXI Stream I/F for transferring Ethernet packet (MacRx I/F and MacTx I/F). Second is 64-bit Reg I/F for parameter setting (CFGWr I/F) and status monitoring (CFGRd I/F). Third is 512-bit AXI Stream I/F for transferring TCP payload data (DMATx I/F and DMARx I/F). The UserLogic consists of two TOE hardware sets for handling TCP/IP packets of two TCP sessions at high-speed performance. Each TOE hardware set consists of TOE100G-IP, TOEDMAReg, TxDMA2TOEIF, and TOE2RxDMAIF. Therefore, when connecting with three interfaces, it requires the switch logics to select one of two TOE hardware sets, i.e., AxiSSw2to1 for both 512-bit AXI Stream I/Fs and RegSwitch for 64-bit Reg I/F.

*Note: UserLogic can be modified to increase or decrease the number of TCP sessions by adjusting the number of the TOE hardware sets. Also, the switch logics must be modified to support the updated number of the TOE hardware sets.*

MacRx I/F directly connects to two TOE100G-IPs without more filtering logics because each TOE100G-IP has its own packet filtering that can be configured the network parameters. While the transmitted Ethernet packet from each TOE100G-IP is fed to AxiSSw2to1 (MacTx) for forwarding the packet to Ethernet MAC via MacTx I/F. If two TOE100G-IPs send the packet at the same time, the transmitted packet of the second TOE100G-IP is transferred after finishing transferring the packet of the first TOE100G-IP.

Before starting the operation, the parameters of each TOE100G-IP and DMA engine (TxDMA2TOEIF and TOE2RxDMAIF) are configured via 64-bit Reg I/F. RegSwitch decodes the address in the request to select the active TOEDMAReg. TOEDMAReg module connects to the control interface of three modules for operating each TCP session, i.e., TOE100G-IP Reg I/F, TxDMA2TOEIF, and TOE2RxDMAIF.

For transmit direction, the host system prepares the transmitted data (TCP payload data) to the Main memory and then sends the command request to UserLogic to start the operation. TxDMA2TOEIF transfers the payload data from the Main memory via 512-bit DMA Tx I/F to Tx FIFO I/F of TOE100G-IP. Next, TOE100G-IP builds and transmits Ethernet packet that includes TCP payload data to AxiSSw2to1 (MacTx) via TxMAC I/F which is the interface of Ethernet MAC. The parameters to run this operation such as the initial address of the main memory, total transmit data size, and transmit packet size are configured by CFGWr I/F before starting the operation. While the status to check the operation progress can be monitored by CFGRd I/F.

On the other hand, the received Ethernet packet from Ethernet MAC is forwarded to TOE100G-IP via MacRx I/F. When the Ethernet packet is valid, TOE100G-IP extracts TCP payload data from the received Ethernet packet and then forwards to TOE2RxDMA I/F. Next, the TOE2RxDMAIF waits until the main memory is free and then uploads TCP payload data to the main memory. After that, the host system reads the payload data from the Main memory for data processing. Similar to transmit operation, the initial address of the main memory and total transfer data size are configured by CFGWr I/F before starting the operation. Also, the operation progress is monitored by CFGRd I/F. To upload the data to the Main memory, AxiSSw2to1 (DMARx) is applied to select the active TOE2RxDMAIF for transferring the data to the Main memory via 512-bit DMARx I/F.

Two clock domains are applied in the UserLogic. The Ethernet MAC I/F uses mac\_clk domain which is equal to 300 MHz while the DMA I/F and CFG I/F use pci\_clk domain which is equal to 250 MHz. Therefore, CDC (clock domain crossing) is implemented inside TOE100G-IP. More details of each hardware module inside the UserLogic are described as follows.

## 2.1 AxiSSw2to1

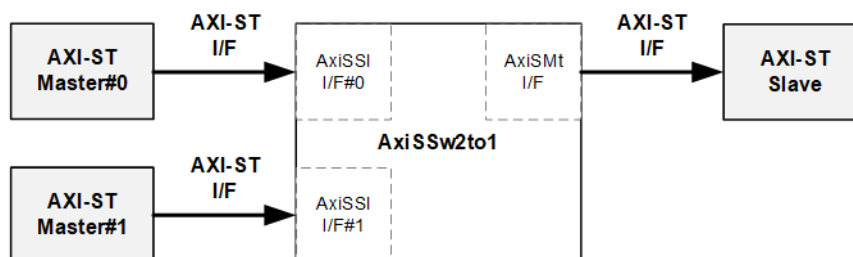


Figure 2-2 AxiSSw2to1 interface

This module is the 2-to-1 switch logic of AXI-ST interface. There are the parameters that can be configured to select the width of the data and user signals. In this reference design, the data width of MacTx interface and DMARx interface are 512 bits while the data width of user signal for MacTx interface and DMARx interface are 1 bit and 81 bits, respectively.

AxiSSw2to1 transfers the data from two Masters (Ch#0 and Ch#1) to one Slave. If two channels send the request to transfer the data at the same time, AxiSSw2to1 selects the higher priority channel and starts transferring the data stream until end of the packet. After that, the priority is switched to another channel and then the data stream of another channel is transferred until end of the packet.

The AxiSSw2to1 logic uses “rChSel” to be the control signal to select the active AxiSSI I/F (the interface to connect to the external Master). When two channels are requested in Idle condition, rChSel changes the value to the new channel after finishing the current channel transferring. More details are shown in Figure 2-3.



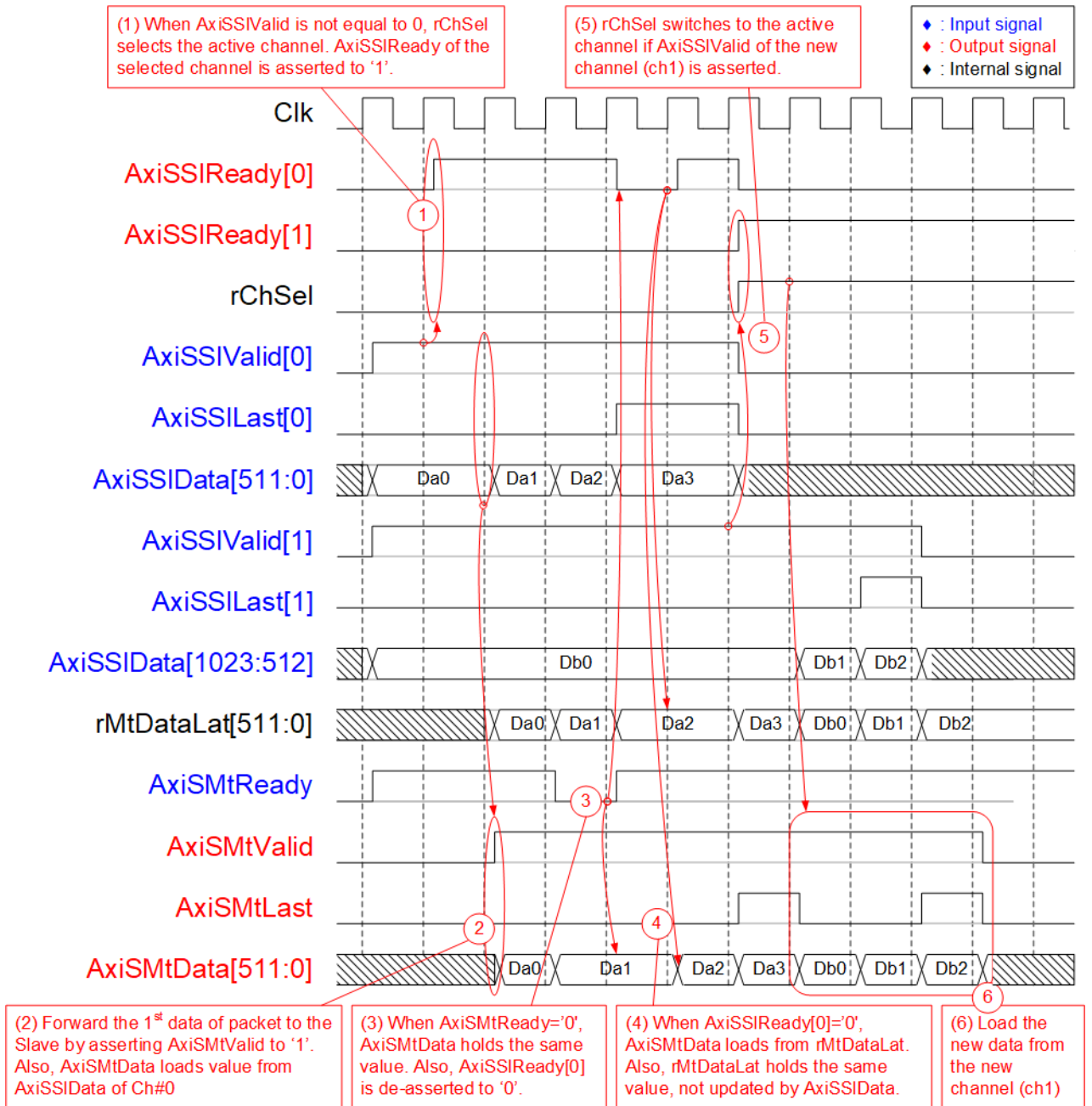


Figure 2-3 AxiSSw2to1 timing diagram

1. When two user sends the new packet by asserting AxiSSIVValid to '1' at the same time and the module is in Idle, rChSel (the signal to indicate the active channel) does not change the value to forward the data from the same channel to the Slave. In Figure 2-3, the Ch#0 is selected, so AxiSSIRReady of the selected channel (Ch#0) is asserted to '1' to accept the first data.
2. The input signals of the selected channel (Ch#0), i.e., AxiSSILast[0] (end-of-packet) and AxiSSIData[511:0] (512-bit data) are loaded to be the output signals to the external Slave via Master I/F (AxiSMtLast and AxiSMtData, respectively). Also, AxiSMtValid is asserted to '1' to start sending the new packet to the Slave.
3. When the Slave is not ready to receive data by de-asserting AxiSMtReady to '0', all output signals of Master I/F hold the same value. Also, AxiSSIRReady of the active channel is de-asserted to '0' to hold the input signals from the Master.
4. After the Slave re-asserts AxiSMtReady to accept the data, the output signals to the Slave load the next value from the internal latch register (rMtDataLat). The internal latch register loads the data from the active source when AxiSSIRReady is asserted to '1' to store the unsend data to the Slave when the Slave pauses data transmission.
5. After the final data of a packet from the active channel is accepted, the next active channel is scanned. If AxiSSIVValid of another channel is asserted, rChSel will switch the value. In Figure 2-3, the next active channel is Ch#1 (rChSel='1') to accept the data from Ch#1.
6. The input signals (AxiSSILast and AxiSSIData) of the active channel (Ch#1) are forwarded to be the output signals of the Slave (AxiSMtLast and AxiSMtData) until the final data of a packet is transferred.

## 2.2 TOE100G-IP

TOE100G-IP implements TCP/IP stack to be the offload engine for transferring TCP/IP packet with the network device. User interface has two signal groups, i.e., control signals and data signals. Register interface is applied to set control registers and monitor status signals while data signals are accessed by using FIFO interface. The interface with 100G EMAC is 512-bit AXI4-ST interface. More details are described in datasheet.

[https://dgway.com/products/IP/TOE100G-IP/dg\\_toe100gip\\_data\\_sheet\\_xilinx.pdf](https://dgway.com/products/IP/TOE100G-IP/dg_toe100gip_data_sheet_xilinx.pdf)

## 2.3 TxDMA2TOEIF

TxDMA2TOEIF is designed to receive data from MCDMA via DMATxI/F and then forward them to TOE100G-IP via Tx FIFO I/F. There is 512 KB buffer in this module to store data for transferring to TOE100G-IP, called TxDMABuf. TxDMABuf is applied to be double buffer by splitting to two areas (Area#0 and Area#1) or 256 KB size per area. The write process of TxDMABuf is handled by CPU while the read process of TxDMABuf for transferring data to TOE100G-IP is run parallelly with the write process, as shown in Figure 2-4.

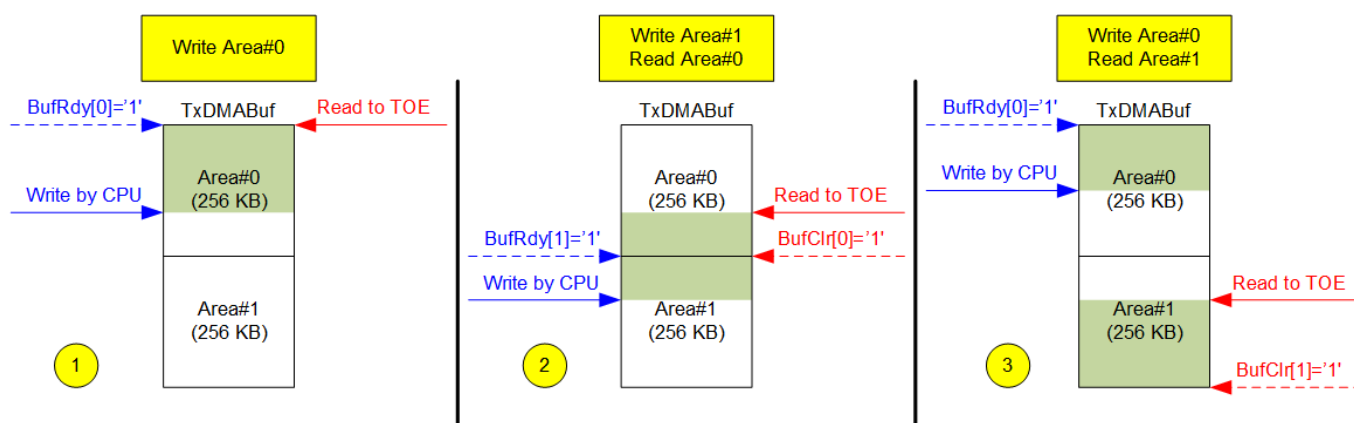


Figure 2-4 Double buffer inside TxDMA2TOEIF

TxDMABuf status is monitored by using two control signals, BufRdy and BufClr which are two-bit signals (each bit is referred to each buffer area). CPU asserts BufRdy[i] to '1' ('i' is the index of TxDMABuf area) to inform that the write process is running at this area. When the read logic detects that BufRdy of the next area is asserted and the write pointer shows some data is available, it starts transferring data of that TxDMABuf area to TOE100G-IP. After finishing reading data in each area, BufClr[i] ('i' is the index of TxDMABuf area) is asserted to '1' for one cycle to clear BufRdy flag to '0'. Therefore, CPU needs to confirm that TxDMABuf area is free (BufRdy[i]='0') before writing the new data and asserting BufRdy[i] to '1'. While the read logic needs to confirm that BufRdy[i]='1' and write pointer shows some data is valid before reading the new data.

The operation is started when the start pulse is asserted along with the total transfer size that must be aligned to 64-byte or 256-bit (the data bus size). After that, the internal signals of this module load their initial value and then the data starts transferring. The data from MCDMA which is stored to TxDMABuf is forwarded to TOE100G-IP until the amount of transferred data is equal to the set value. Finally, Busy flag is de-asserted to '0' after the operation is done.

Each TxDMA2TOEIF uses one DMA channel for transferring TCP payload data of each session with MCDMA. The UserLogic contains two TxDMA2TOEIF modules for supporting two TCP sessions, so two DMA channels – DMA Ch#0 and DMA Ch#8 are applied. User can update the DMA channel number for usage by modifying the parameter assignment to TxDMA2TOEIF module in HDL code.

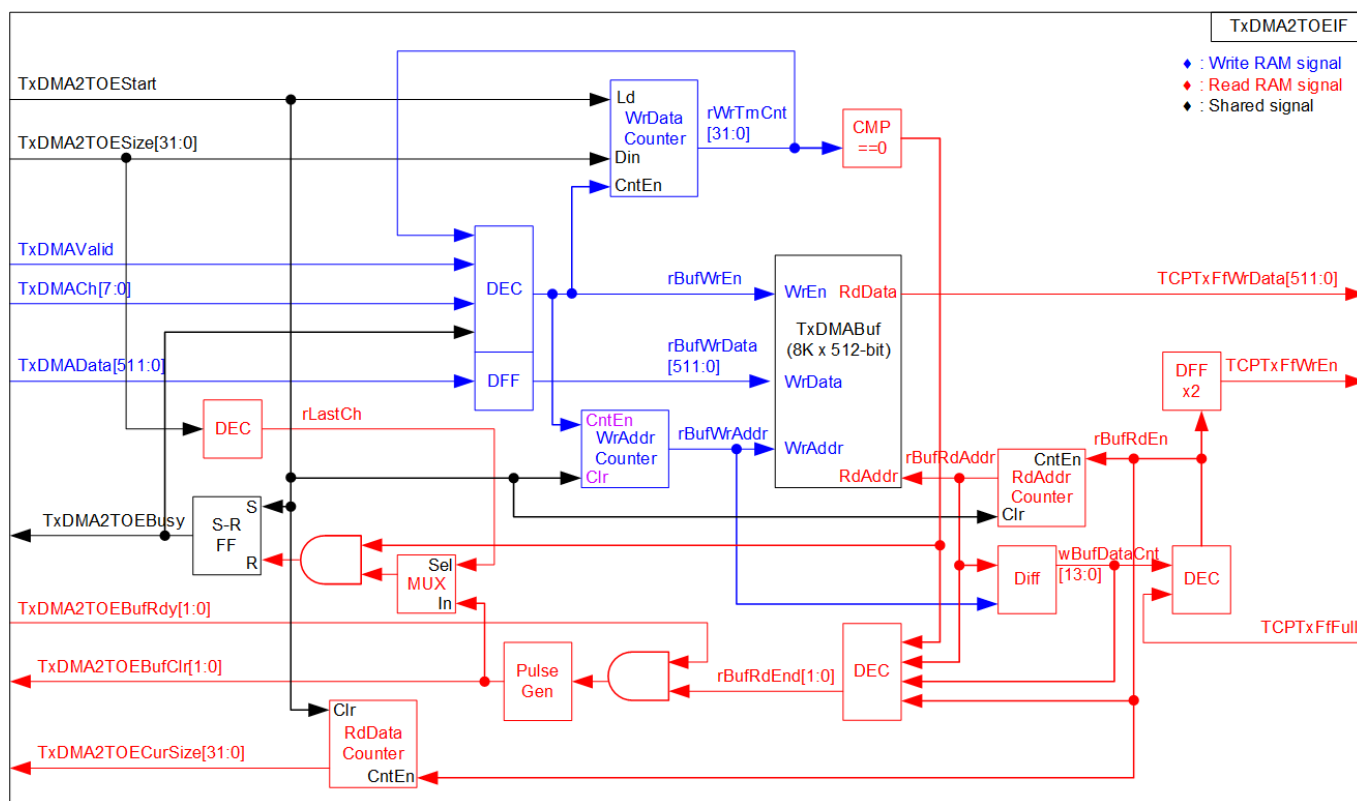


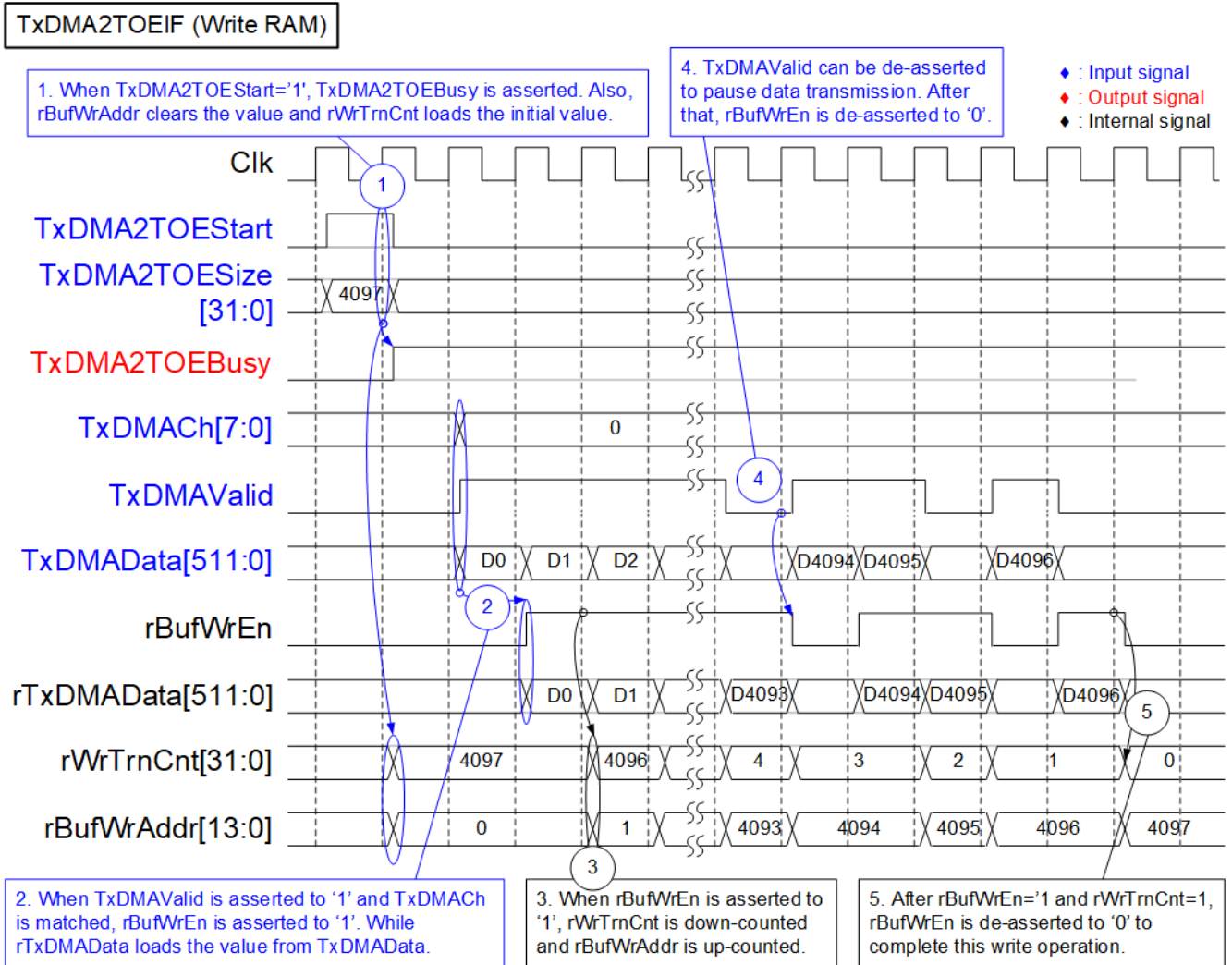
Figure 2-5 TxDMA2TOEIF block diagram

As shown in Figure 2-5, the logic inside TxDMA2TOEIF is divided to two groups, Write RAM group and Read RAM group that are colored by blue and red, respectively. Start flag (TxDMA2TOEStart) is applied to reset and load the internal signals. Total transfer size (TxDMA2TOESize) is loaded to WrData Counter which counts total amount of Write data to TxDMABuf. If the new data (TxDMAData) of the configured DMA channel (TxDMACH) is received by asserting TxDMAValid to '1', the write enable of TxDMABuf (rBufWrEn) is asserted to store the data (rBufWrData) to TxDMABuf. Also, rBufWrEn is applied to be count enable of the WrData counter. However, rBufWrEn is not asserted if the data is received before the start flag is asserted or after the operation is done. The write address of TxDMABuf (rBufWrAddr) is also counted by rBufWrEn to store the new data in the next address.

The write address (rBufWrAddr) and the read address (rBufRdAddr) are used to calculate total amount of data inside TxDMABuf (wBufDataCnt). Though the buffer depth is 8192 which can use 13-bit address signal, the address signal is designed by 14 bits to determine the buffer status that is full or empty. When TxDMABuf is not empty and TOE100G-IP is ready, rBufRdEn is asserted to read the data from TxDMABuf. rBufRdAddr is up-counted to read the next data after rBufRdEn is asserted. The read data from TxDMABuf is transferred to TCPTxFfWrData of TOE100G-IP directly. While the write enable to TOE100G-IP (TCPTxFfWrEn) is created by connecting rBufRdEn with two D Flip-Flops for data synchronizing. After reading the last data of each area, rBufRdEnd of that area is asserted to '1'. This signal is applied to generate a pulse of TxDMA2TOEBufClr.

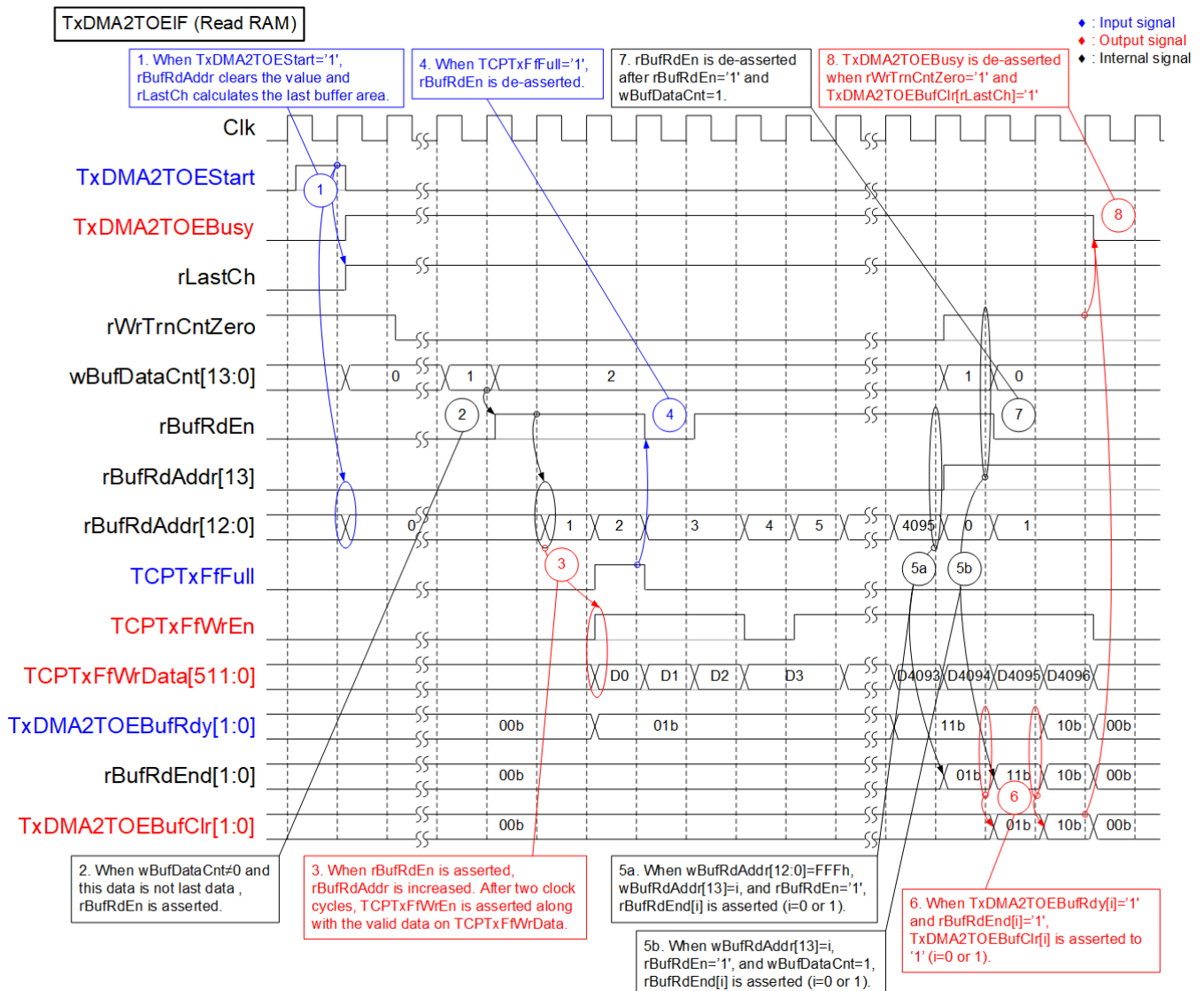
Busy flag (TxDMA2TOEBusy) is asserted after TxDMA2TOEStart is asserted. It is de-asserted after the last data is read and forwarded to TOE100G-IP completely. The last read data is detected by checking WrTrnCnt=0 (no remaining write data) and rBufRdEnd of the last active area (checked by rLastCh) is asserted. Besides, RdData Counter is designed to show the amount of completed data (TxDMA2TOECurSize) for user monitoring.

Timing diagram of the logic for writing RAM is shown in Figure 2-6 while timing diagram of the logic for reading RAM is shown in Figure 2-7. In this example, user sets total transfer size to 4097, so the first buffer area (Area#0) stores 256-Kbyte data (4096 x 512-bit) and the second buffer area (Area#1) stores 64-byte data (512-bit).



**Figure 2-6 TxDMA2TOEIF (Write RAM) timing diagram**

1. When TxDMA2TOEStart is asserted to '1', the internal signals are initialized. rWrTrnCnt loads the initial value from TxDMA2TOESize while rBufWrAddr is reset to 0. Also, TxDMA2TOEBusy is asserted to '1' while operating until the operation is finished.
2. When the new data is received (TxDMAValid='1' and TxDMACH=Set parameter), rBufWrEn is asserted to '1' to write the new rTxDMADData that is loaded by TxDMADData to TxDMABuf.
3. Also, rBufWrEn is applied to be the counter enable of several signals such as rBufWrAddr which is increased to store the next data to the buffer at the next address. Also, rWrTrnCnt is decreased to count the remaining transfer size in Write process.
4. DMA engine can pause data transmission by de-asserting TxDMAValid to '0'. After that, rBufWrEn is de-asserted to '0' in the next cycle to pause writing data to the buffer.
5. After the last data is received and stored to the buffer (rBufWrEn='1' and rWrTrnCnt=1), rBufWrEn is de-asserted to '0'. The write operation is done. rBufWrEn can be re-asserted to '1' when the new request is asserted and rWrTrnCnt loads the new value which is not equal to 0.



**Figure 2-7 TxDMA2TOEIF (Read RAM) timing diagram**

1. When TxDMA2TOEStart is asserted to '1', the internal signals are initialized. rBufRdAddr is reset to 0 and rLastCh calculates the last buffer area used for this request. In this example, the last buffer area is Area#1, so rLastCh is asserted to '1'. rWrTrnCntZero is asserted to '1' when rWrTrnCnt is equal to 0 to inform that the write operation is done.
2. The read operation is started when there is some data stored to the buffer (monitored by wBufDataCnt≠0) and this data is not last data (wBufDataCnt=1 and rBufRdEn='1'). rBufRdEn is asserted to '1' to start reading data from the buffer.
3. The read latency time of the buffer is equal to 2, so the first data (D0) read from the buffer is valid on TCPTxFfWrData after rBufRdEn is asserted for two clock cycles. To synchronous with the data bus, TCPTxFfWrEn is created by adding two-cycle latency time to rBufRdEn. Besides, rBufRdEn is applied to count the read address of the buffer (rBufRdAddr) to point the next address after finishing reading each data from the buffer.
4. If TOE100G-IP is not ready to receive the data by asserting TCPTxFfFull to '1', rBufRdEn is de-asserted to '0' to pause reading data and forwarding data to TOE100G-IP.
5. The end flag of the buffer (rBufRdEnd) has two bits to be the status of two buffer areas (bit[0]: Area#0 and bit[1]: Area#1). The active buffer area for read process can be decoded by rBufRdAddr[13] ('0'-Area#0, '1'-Area#1). rBufRdEnd is asserted by one of two conditions.
  - a. The last address of each buffer area is read (rBufRdAddr[12:0]=FFFh or 4095 and rBufRdEn='1').
  - b. The last data of this request is read by checking the write process is done (rWrTrnCntZero='1') and the last data is read (wBufDataCnt=1 and rBufRdEn='1').
6. Similar to rBufRdEnd, TxDMA2TOEBufClr and TxDMA2TOEBufRdy have two bits to be the status of two buffer areas. TxDMA2TOEBufClr of the active area is asserted to '1' when TxDMA2TOEBufRdy and rBufRdEnd are asserted to '1' (the last data of this active buffer is read). This signal is applied to de-assert TxDMA2TOEBufRdy in the next cycle to allow the write process starts.
7. After the last data is read from the buffer (rBufRdEn='1' and wBufDataCnt=1), rBufRdEn is de-asserted to '0' to stop reading the data from the buffer.
8. If both write process and read process are done, TxDMA2TOEBusy is de-asserted to '0'. The write process is done when rWrTrnCntZero is asserted to '1'. While the read process is done when TxDMA2TOEBufClr of the last active area (decoded by rLastCh) is asserted to '1'.

## 2.4 TOE2RxDMAIF

TOE2RxDMAIF is designed to receive data from TOE100G-IP and then forward to MCDMA for storing TCP payload data to the main memory at Rx buffer area. The Rx buffer area is split to two areas – Area#0 and Area#1 for handling by double buffer style. Rx buffer is controlled by using two flags, BufValid and BufClr.

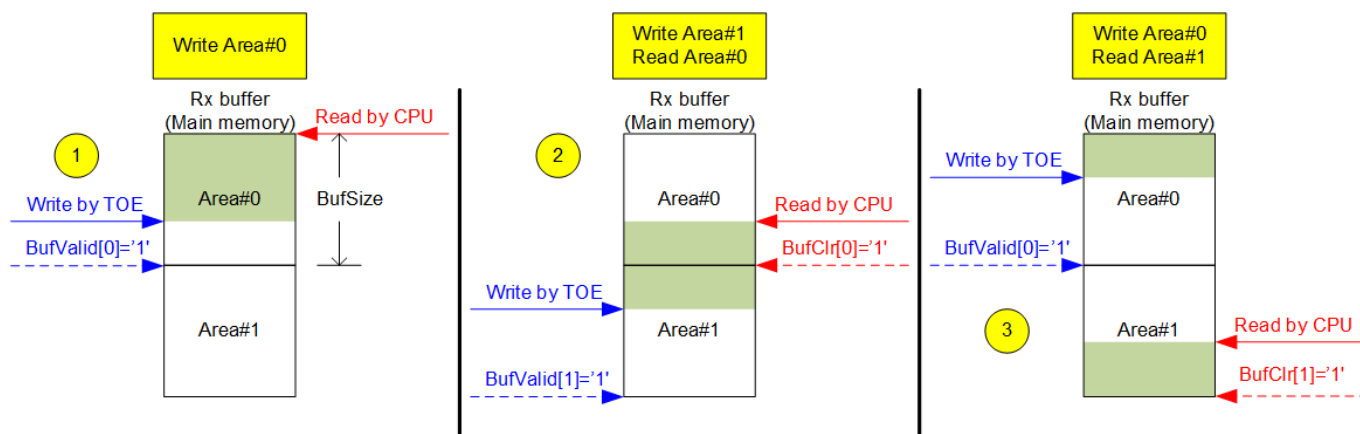


Figure 2-8 Double buffer for Rx buffering from TOE100G-IP

BufValid and BufClr are two-bit signals, bit[0]-Area#0 and [1]-Area#1. The buffer size of each area is the parameter that is configured by the software on the host system. When the hardware finishes writing all data to each Rx buffer area, it asserts BufValid[i] to '1' ('i' is the index of Rx buffer area). After that, the software on the host system that detects the flag asserted starts reading the data from the Rx buffer. After all data of each Rx buffer area is completely read out, the CPU asserts BufClr[i] to '1' ('i' is the index of Rx buffer area). BufClr[i] is applied to de-assert BufValid[i] to '0' to allow the hardware logic to write the new data at this buffer area. The hardware logic always confirms that the active buffer is free (BufValid[i]='0') before starting writing data. While the CPU needs to confirm the active area has the data (BufValid[i]='1') before starting reading data from the Rx buffer.

To transfer the data to DMARx I/F with achieving high throughput, TOE2RxDMAIF fixes the transfer size of each packet to 8 Kbyte data block, except the last data block that may be less than 8 Kbyte size. Therefore, the Rx buffer size must be aligned to 8 Kbyte unit. To start the operation, the start pulse is asserted along with the total transfer size that must be aligned to 64-byte or 256-bit (the data bus size). After that, all received data from TOE100G-IP is transferred to the Rx buffer via DMARx I/F. The operation is done when total amount of transferred data is equal to the set value. While the operation is not completed, busy flag is asserted to '1'.

Similar to TxDMA2TOEIF, each TOE2RxDMAIF uses one DMA channel for transferring data with MCDMA. In this system, two TOE2RxDMAIF are applied for supporting two TCP sessions. Therefore, two DMA channels – DMA Ch#0 and DMA Ch#8 are applied. The DMA channel number for using in each TOE2RxDMAIF can be assigned by the parameters in HDL code.



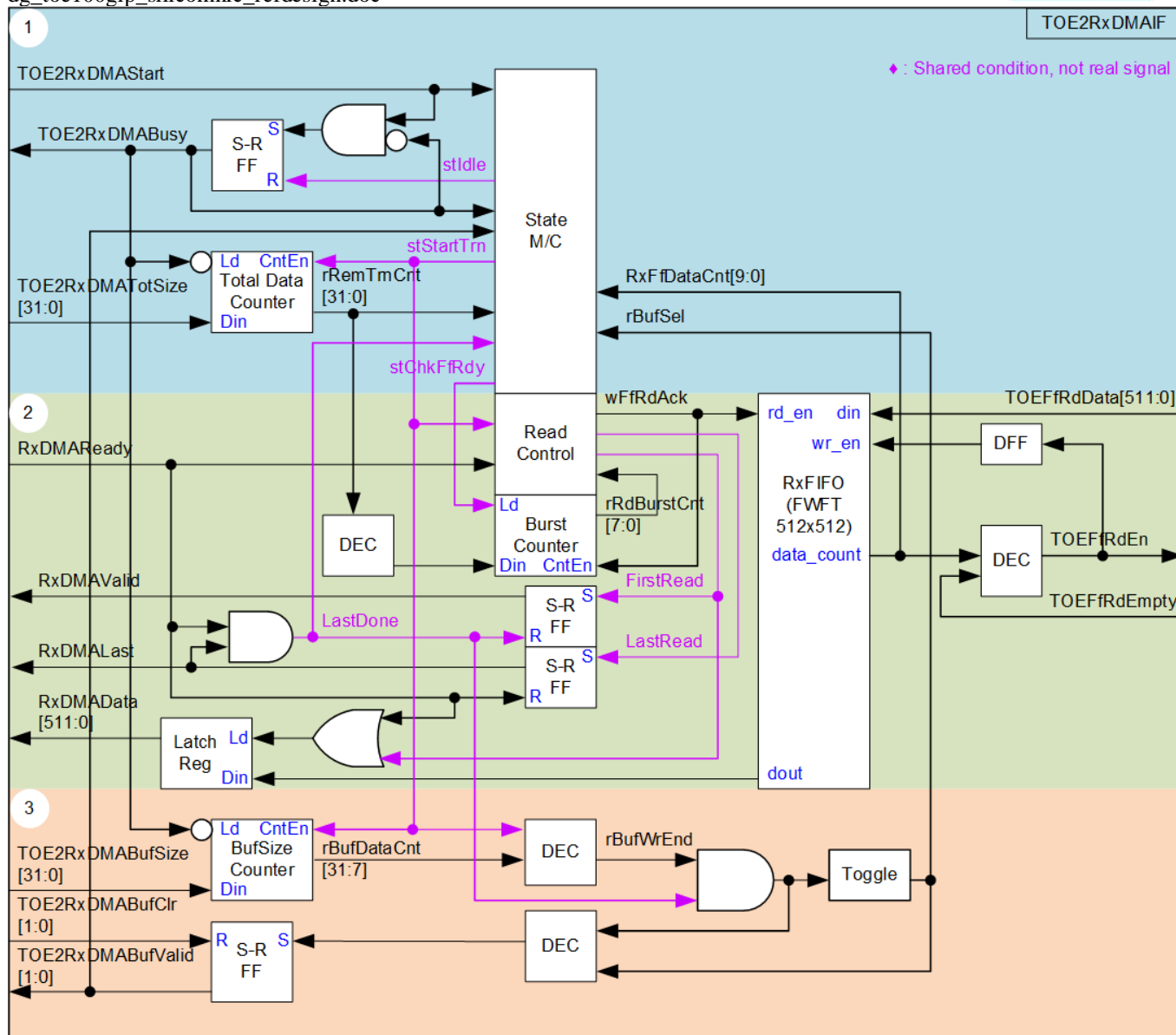


Figure 2-9 TOE2RxDMAIF Block Diagram

Figure 2-9 shows the logic diagram inside TOE2RxDMAIF. It consists of three groups. First is the logic for user interface. Second is the logic for transferring the data from TOE100G-IP to DMARx I/F via RxFIFO. Last is the logic for buffer handling.

The logic has three counters for controlling transfer size. First is “Total Data Counter” which controls total amount of data for this request. The data is transferred from TOE100G-IP to Rx buffer via DMARx I/F which is split to two buffer areas. Therefore, it needs to have a second counter – BufSize Counter to count the data size that is stored to Rx buffer. At the end of each buffer area, the logic needs to check the next buffer status before starting transferring data. The data interface for transferring via DMARx I/F is AXI stream that needs to send the data in packet format. This reference design fixes the packet size to be 8 Kbyte size, except the last transfer which can be less than 8 Kbyte. Therefore, the third counter – Burst Counter is applied to control the data size of each packet. For simple design, the buffer size must be aligned to 8 Kbyte size which is the packet size.

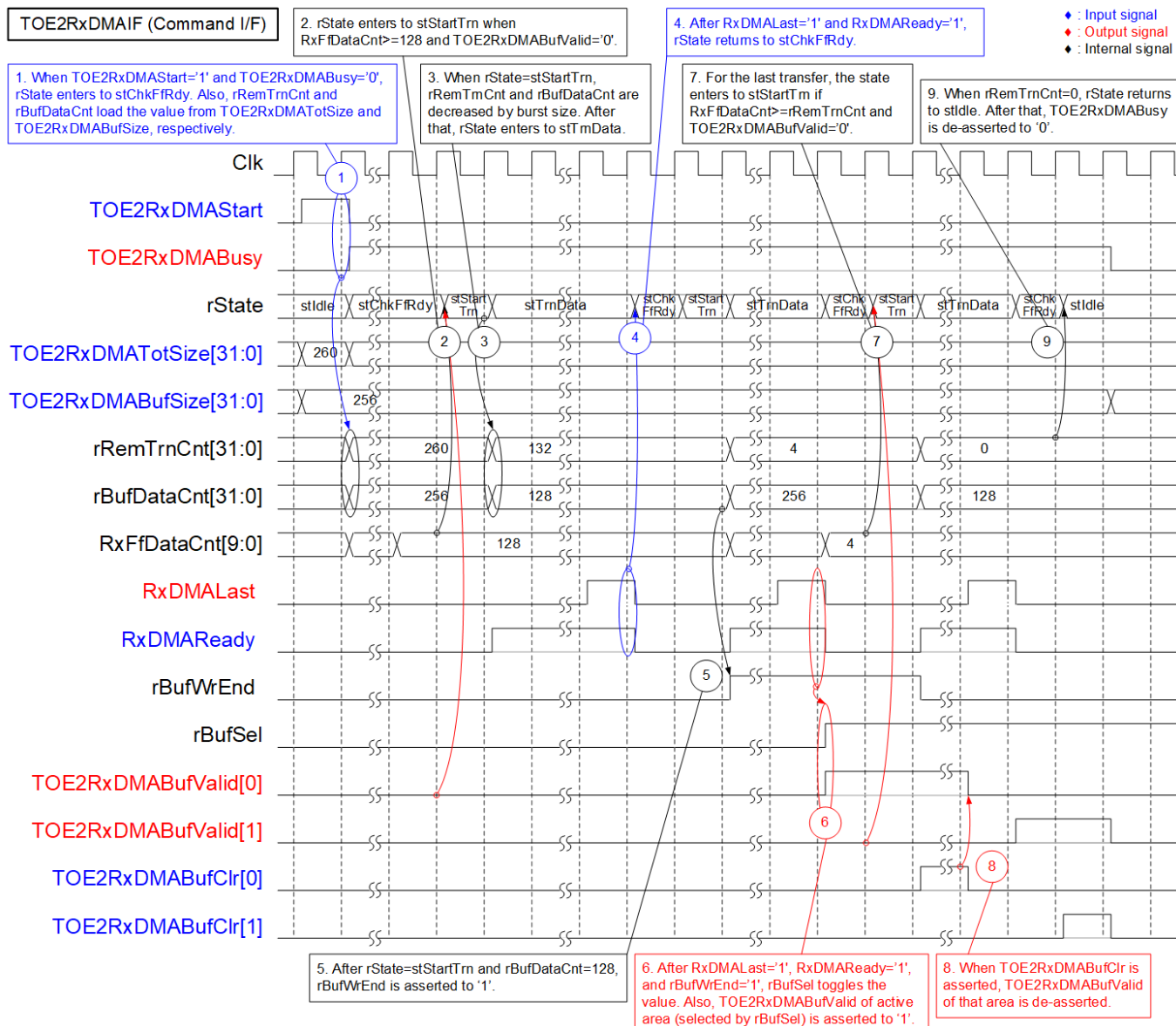
More details of block no.1, no.2, and no.3 in Figure 2-9 are described as follows.

As shown in Block no.1, Start pulse (TOE2RxDMAStart) is accepted when the logic is Idle (busy flag – TOE2RxDMABusy='0'). After that, State machine enters to the next state for starting operation and the busy flag is asserted to '1'. Total transfer size (TOE2RxDMATotSize) is loaded to Total data counter" to check the amount of data in this request. Before starting transferring data in each loop, the state machine waits until the amount of data inside the internal FIFO (RxFIFO) and the remaining transfer size (rRemTrnCnt) are enough. Also, the BufValid flag of the active Rx buffer area, selected by rBufSel, must be de-asserted to '0' (the Rx buffer is free). If both RxFIFO and Rx buffer are ready, the state changes from stChkFfRdy to stStartTrn for transferring data from RxFIFO to DMARx I/F (Rx buffer I/F). Total data counter (rRemTrnCnt) decreases the value to show the remaining transfer size after starting each transfer loop (one loop transfers one packet that is called burst size). The operation is done and the state returns to stIdle when all data is completely transferred. Finally, busy flag is de-asserted to '0'.

Block no.2 shows the data path that is forwarded from TOE100G-IP to DMARx I/F. The FWFT FIFO (RxFIFO) is integrated to be the data buffer. The FIFO depth is 512 while the data width is 512. Therefore, the buffer can store up to 4 packets (when the packet size is 8 Kbytes). On the right side of Block no.2, the data is read from TOE100G-IP when the TOE100G-IP FIFO is not empty (TOEFfRdEmpty='0') and RxFIFO is not full (RxFfDataCnt<504). The read FIFO process is controlled by the state machine. The burst size of each transfer loop is determined by rRemTrnCnt that is always equal to 8 Kbyte size, except the last loop which is equal to rRemTrnCnt. The burst size result is set to the Burst Counter in stChkFfRdy state. After that, the state enters to stStartTrn to start reading the first data from RxFIFO. The data is read from RxFIFO by asserting wFfRdAck to '1' and then forwarded to DMARx I/F by asserting RxDMAValid to '1'. The data transmission can be paused by DMARx I/F when RxDMAReady is de-asserted to '0'. When the last data of each transfer loop is read from RxFIFO, RxDMALast is asserted. After that, the state machine returns to stChkFfRdy to prepare the next loop transfer.

Block no.3 is the logic to select the Rx buffer area by using rBufSel ('0'-Area#0, '1'-Area#1). The buffer size (TOE2RxDMABufSize) is loaded to BufSize counter before the operation is started. The counter shows the remaining transfer size of this buffer area. It decreases by the burst size at the start time of each transfer loop. If the current loop is the last transfer of this buffer area, rBufWrEnd is asserted. After that, the active buffer area (rBufSel) is toggled to switch the active buffer area. After the last data is transferred to DMARx I/F completely, TOE2RxDMABufValid is asserted to '1' to inform that the data of this buffer area is ready for the host system for reading. The host system asserts TOE2RxDMABufClr to '1' when finishing reading data of each area. After that, BufValid is de-asserted to '0' which means this Rx buffer area is free for TOE2RxDMAIF writing the new received data.

Figure 2-10 shows timing diagram for handling the control and status signals of this module when the user sets transfer size to 260 (512-bit unit) and buffer size to 256 (512-bit unit). Therefore, the operation is run for three loops. The first loop and the second loop transfers 128 data (8Kbyte) by using Rx buffer#0. While the final loop transfers the remaining data (4 data) by using Rx buffer#1.



**Figure 2-10 Command I/F of TOE2RxDMAIF Timing diagram**

- 1) When the new command request (TOE2RxDMAStart) is asserted to '1' along with the valid TOE2RxDMATotSize (the total transfer size in 64-byte unit) and TOE2RxDMABufSize (Rx buffer size in 64-byte unit), TOE2RxDMABusy is asserted to '1' to accept the request. Meanwhile, two data counters to count the remaining transfer size (rRemTrnCnt) and the remaining buffer size (rBufDataCnt) load the initial value from TOE2RxDMATotSize and TOE2RxDMABufSize, respectively. The first area of Rx buffer is Rx buffer#0, so rBufSel is reset by '0'. After that, the state enters to stChkFfRdy.
- 2) In stChkFfRdy, the state waits until the amount of data in FIFO is enough (RxFfDataCnt  $\geq$  128 when the current loop is not the last loop) and the Rx buffer is free (TOE2RxDMABufValid[i]='0'; i is the index of Rx buffer area that is assigned by rBufSel). After that, the state enters to stStartTrn.
- 3) The state stays in stStartTrn for one clock cycle for updating the counters. rRemTrnCnt and rBufDataCnt are decreased by the burst size of each loop which is fixed to 128, except the last loop which is equal to the remaining size. After that, the state enters to stTrnData.
- 4) In stTrnData, the data is transferred from FIFO to Rx buffer. When the last data of each transfer loop is completely transferred to Rx buffer (RxDMALast='1' and RxDMAReady='1'), the state returns to stChkFfRdy to check the remaining transfer size. If there is remained data for transferring, it prepares the next loop parameters. Otherwise, the operation is finished (step 9).
- 5) This example shows the operation when the buffer size is set to 256. Therefore, the second transfer loop is the last transfer for this buffer area. rBufWrEnd is asserted to '1' when rBufDataCnt=128.
- 6) When the last data of this buffer area is transferred completely (RxDMALast='1', RxDMAReady='1', and rBufWrEnd='1'), the active buffer is switched (rBufSel is toggled). Also, the buffer valid flag of this Rx buffer area (TOE2RxDMABufValid[0]) is asserted to '1'.
- 7) This example shows the last transfer loop that is not aligned to 128. The remaining transfer size of the last loop is 4, so the burst size is set to 4. In this case, the state changes from stChkFfRdy to stStartTrn when the amount of data in FIFO (RxFfDataCnt) is more than or equal to the last transfer size (rRemTrnCnt).
- 8) If the host system completely reads all data from each Rx buffer area, it will assert TOE2RxDMABufClr of that area to '1'. When BufClr is asserted to '1', the selected Rx buffer area is free by de-asserting TOE2RxDMABufValid to '0'.
- 9) After all data is transferred completely, rRemTrnCnt is equal to 0. The state changes from stChkFfRdy to stIdle. After that, TOE2RxDMABusy is de-asserted to '0'.

Figure 2-11 shows timing diagram of data interface. The data is transferred from Rx FIFO to Rx buffer via DMARx I/F. The example shows the details when transferring 128 data and the beginning time of the last loop which the burst size is equal to 4.

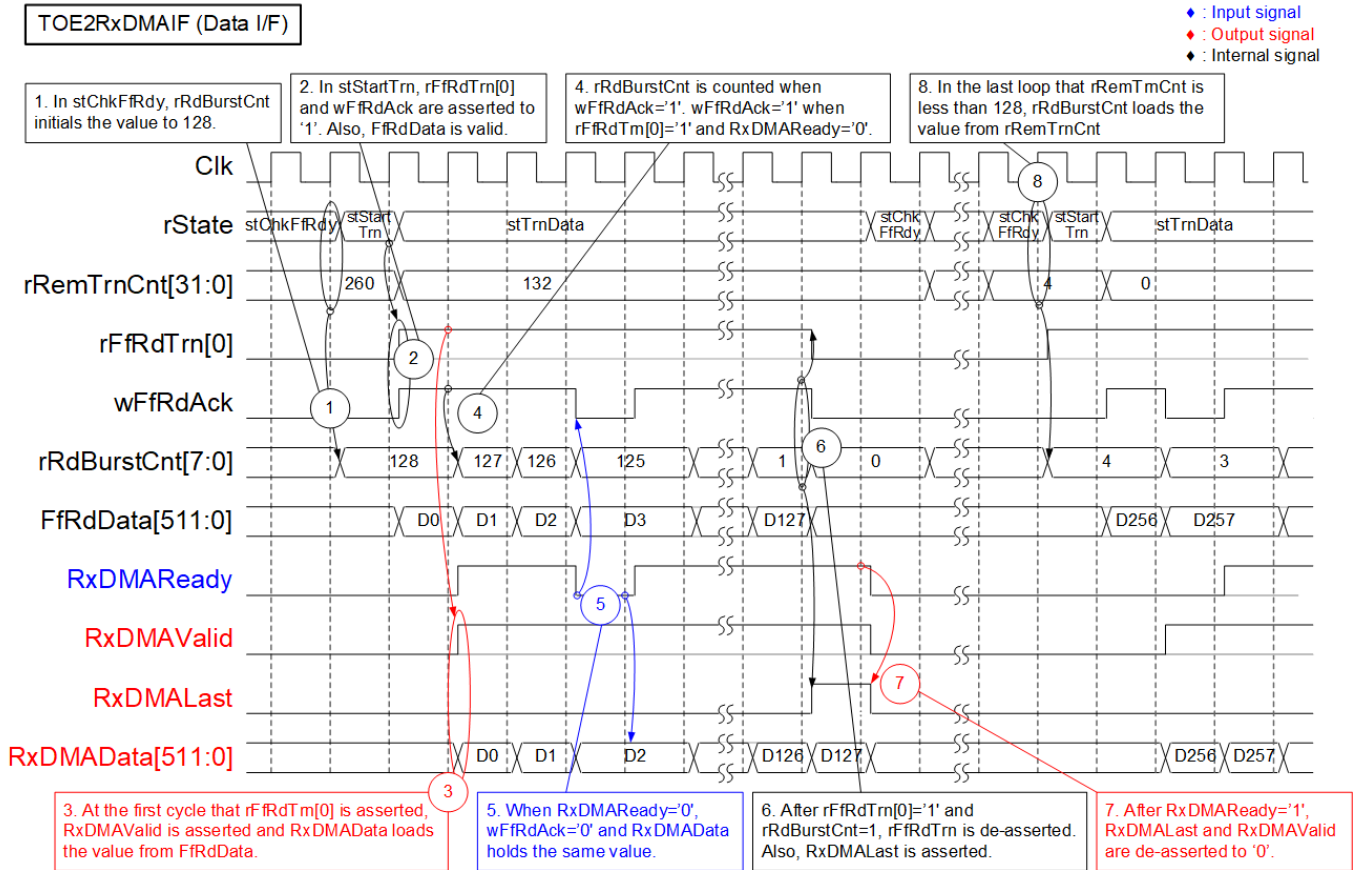


Figure 2-11 Data I/F of TOE2RxDMAIF Timing diagram

- 1) When the state is stChkFfRdy, the burst counter (rRdBurstCnt) calculates the burst size of this transfer loop which is equal to 128, except the last loop.
- 2) After the state changes from stChkFfRdy to stStartTrn, the first data of this transfer loop is read from Rx FIFO by asserting rFfRdTrn[0] and wFfRdAck to '1'. rFfRdTrn[0] is asserted until the last data of this loop is transferred. While wFfRdAck is asserted to read the first data by checking the rising edge of rFfRdTrn[0]. Rx FIFO is FWFT type, so FfRdData is valid at the same clock cycle as wFfRdAck asserted.
- 3) RxDMAData loads the first read data from Rx FIFO (FfRdData) for transferring to the Rx buffer by asserting RxDMAValid to '1'.
- 4) When each data is read from Rx FIFO by asserting wFfRdAck to '1', the burst counter (rRdBurstCnt) is down-counted to show the remaining size in this loop.
- 5) When the Rx buffer is not ready to receive data by de-asserting RxDMAReady to '0', wFfRdAck is de-asserted to '0' to pause reading data from Rx FIFO. Also, RxDMAData holds the same value until RxDMAReady is re-asserted to '1'.
- 6) After the last data of this transfer loop is read from Rx FIFO (wFfRdAck='1' and rRdBurstCnt=1), rFfRdTrn[0] is de-asserted to '0'. Also, RxDMALast is asserted to '1' to inform that the last data is transferred on DMARx I/F.
- 7) After RxDMAReady is asserted to '1' to accept the last data, RxDMAValid and RxDMALast are de-asserted to '0'.
- 8) When the remaining transfer size of the last loop, checked by rRemTrnCnt, is less than 128, rRdBurstCnt loads the value from rRemTrnCnt. In this example, the transfer size of the last is equal to 4, so rRdBurstCnt is set to 4.

## 2.5 RegSwitch

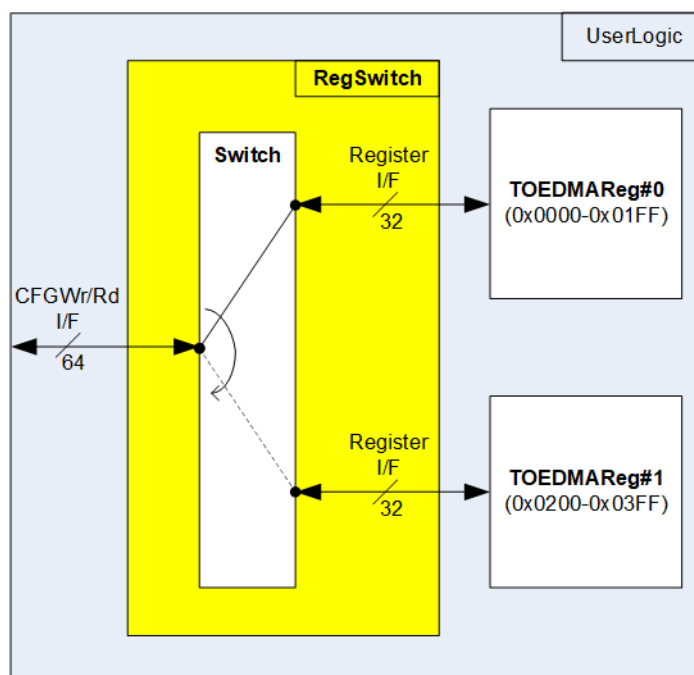


Figure 2-12 RegSwitch block diagram

RegSwitch module is designed to connect two register modules (TOEDMAReg) to CFGWr/Rd I/F. Each TOEDMAReg is mapped to different area, so the switch logic decodes the upper bits of the address for selecting the active module. The data size of CFGWr/Rd interface is 64 bits while the data size of TOEDMAReg is 32 bits. For simple design, this module ignores the upper word (bit[63:32]) for the write access. While zero is padded to the upper word for the read access. The mapped address for each TOEDMAReg is shown as follows.

- 1) 0x0000 – 0x01FF: TOEDMAReg#0
- 2) 0x0200 – 0x02FF: TOEDMAReg#1

## 2.6 TOEDMAReg

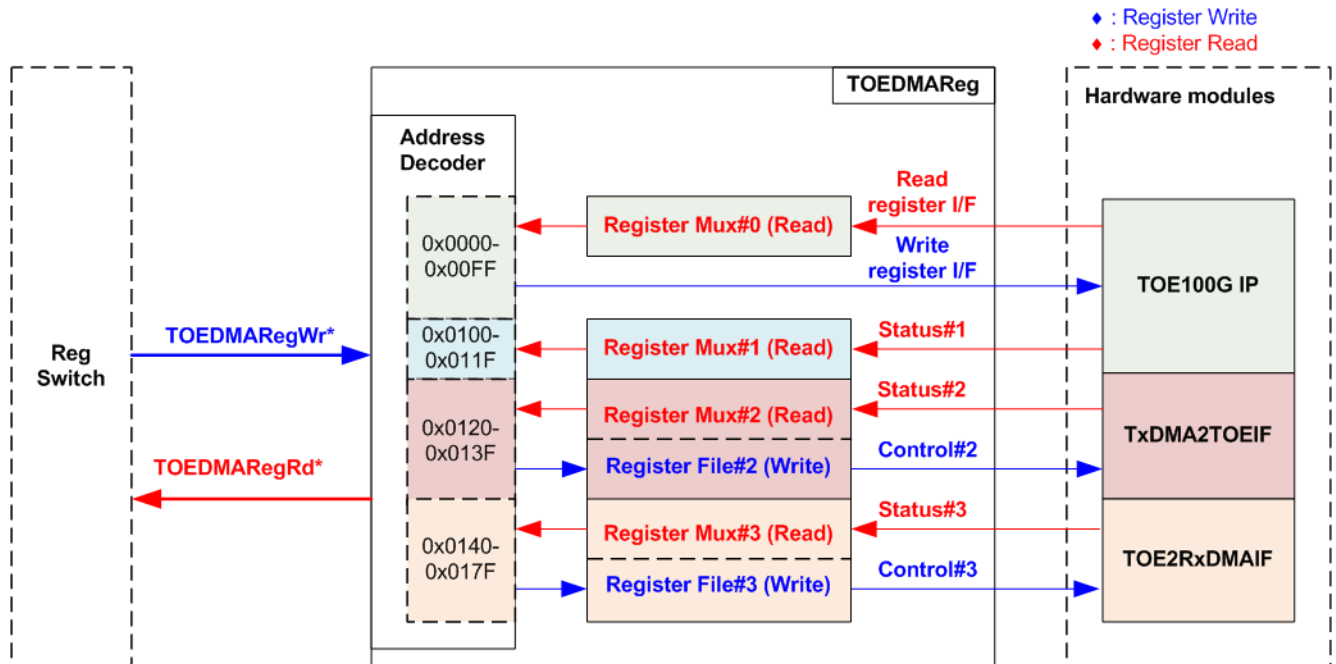


Figure 2-13 TOEDMAReg block diagram

TOEDMAReg consists of many registers for interfacing with the hardware submodules, i.e., TOE100G-IP, TxDMA2TOEIF, and TOE2RxDMAIF. The address for write or read access is decoded by Address decoder to select the active register. There are four address areas, as shown in Figure 2-13.

- 1) 0x0000 – 0x00FF: TOE100G-IP register interface area
- 2) 0x0100 – 0x011F: TOE100G-IP status area
- 3) 0x0120 – 0x013F: TxDMA2TOEIF control and status area
- 4) 0x0140 – 0x017F: TOE2RxDMAIF control and status area

Address decoder decodes the upper bits of TOEDMARegWrAddr and TOEDMARegRdAddr for selecting the active address area while the lower bits is applied to select the active register in each area. There are many status registers in TOEDMAReg, so multi-level multiplexers are applied to select the read value. In this design, the latency time of read data is equal to two clock cycles, so TOEDMARegRdValid is created by adding two D Flip-flops to TOEDMARegRdReq. More details of the address mapping within TOEDMAReg module are shown in Table 2-1.



**Table 2-1 Register map Definition**

Address Wr/Rd	Register Name (Label in the MultiTOE100DMATest.c) Description
<b>BA+0x0000 – BA+0x00FF: TOE100GIP register interface (Write/Read access)</b>	
BA+0x0000	DG_TOEIP_RST_INTREG_OFFSET Mapped to RST register within TOE100G-IP.
BA+0x0008	DG_TOEIP_CMD_INTREG_OFFSET Mapped to CMD register within TOE100G-IP.
BA+0x0010	DG_TOEIP_SML_INTREG_OFFSET Mapped to SML register within TOE100G-IP.
BA+0x0018	DG_TOEIP_SMH_INTREG_OFFSET Mapped to SMH register within TOE100G-IP.
BA+0x0020	DG_TOEIP_DIP_INTREG_OFFSET Mapped to DIP register within TOE100G-IP.
BA+0x0028	DG_TOEIP_SIP_INTREG_OFFSET Mapped to SIP register within TOE100G-IP.
BA+0x0030	DG_TOEIP_DPN_INTREG_OFFSET Mapped to DPN register within TOE100G-IP.
BA+0x0038	DG_TOEIP_SPN_INTREG_OFFSET Mapped to SPN register within TOE100G-IP.
BA+0x0040	DG_TOEIP_TDL_INTREG_OFFSET Mapped to TDL register within TOE100G-IP.
BA+0x0048	DG_TOEIP_TMO_INTREG_OFFSET Mapped to TMO register within TOE100G-IP.
BA+0x0050	DG_TOEIP_PKL_INTREG_OFFSET Mapped to PKL register within TOE100G-IP.
BA+0x0058	DG_TOEIP_PSH_INTREG_OFFSET Mapped to PSH register within TOE100G-IP.
BA+0x0060	DG_TOEIP_WIN_INTREG_OFFSET Mapped to WIN register within TOE100G-IP.
BA+0x0068	DG_TOEIP_ETL_INTREG_OFFSET Mapped to ETL register within TOE100G-IP.
BA+0x0070	DG_TOEIP_SRV_INTREG_OFFSET Mapped to SRV register within TOE100G-IP.
BA+0x0078	DG_TOEIP_VER_INTREG_OFFSET Mapped to VER register within TOE100G-IP.
BA+0x0080	DG_TOEIP_DML_INTREG_OFFSET Mapped to DML register within TOE100G-IP.
BA+0x0088	DG_TOEIP_DMH_INTREG_OFFSET Mapped to DMH register within TOE100G-IP.
<b>BA+0x0100 – BA+0x011F: TOE100GIP status (Write/Read access)</b>	
BA+0x0100	TOE100G-IP status DG_USER_STS_INTREG_OFFSET Rd – [0]: Mapped to ConnOn from TOE100G-IP.
BA+0x0108	Connection interrupt DG_USER_INT_INTREG_OFFSET Wr - [0]: Set '1' to clear the connection interrupt. [8]: Set '1' to clear this bit which shows the latched value of TimerInt. Rd – [0]: Interrupt when ConnOn changes the value. ( '0': ConnOn does not change, '1': ConnOn changes its value) [8]: Interrupt when TimerInt from TOE100GIP is asserted. ( '0': TimerInt is not asserted, '1': TimerInt is asserted to '1') <i>Note: ConnOn value can be read from DG_TOEIP_CONNON_INTREG_OFFSET.</i>

Address Wr/Rd	Register Name (Label in the MultiTOE100DMATest.c) Description
<b>BA+0x0100 – BA+0x011F: TOE100GIP status (Write/Read access)</b>	
BA+0x0110	<p>TOE100G-IP FIFO status DG_USER_FFSTS_INTREG_OFFSET</p> <p>Rd – [5:0]: Mapped to TCPRxFfLastRdCnt from TOE100G-IP. Rd – [15:6]: Mapped to TCPRxFfRdCnt from TOE100G-IP. Rd – [24]: Mapped to TCPTxFfFull from TOE100G-IP.</p>
<b>BA+0x0120 – BA+0x013F: TxDMA2TOEIF (Write/Read access)</b>	
BA+0x0120	<p>TxDMA2TOEIF command DG_TXDMA_COMMAND_INTREG_OFFSET</p> <p>Wr – [0]: Start TxDMA2TOEIF operation. Asserted to '1' to start transferring data on TxDMA2TOEIF. This flag is auto-cleared. Rd – [0]: Busy flag of TxDMA2TOEIF. Asserted to '1' while TxDMA2TOEIF is operating.</p>
BA+0x0128	<p>Total transfer size of TxDMA2TOEIF DG_TXDMA_TRNSIZE_INTREG_OFFSET</p> <p>Wr – [31:0]: Total transfer size in 512-bit unit. Valid range is 1-0xFFFFFFFF. Rd – [31:0]: Current amount of transferred data in 512-bit unit.</p>
BA+0x0130	<p>The control and status of TxDMA buffer inside TxDMA2TOEIF DG_TXDMA_BUFFCTRL_INTREG_OFFSET</p> <p>Wr – [1:0]: Buffer ready status. Asserted to '1' when the TxDMA buffer of that area is ready for reading. Each bit is the index of TxDMA buffer area. [0]-TxDMA buffer#0, [1]-TxDMA buffer#1. Rd – [1:0]: Buffer status. '0'-TxDMA buffer is free or data is not ready, '1'-Data in TxDMA buffer is ready. Each bit is the index of TxDMA buffer area. [0]-TxDMA buffer#0, [1]-TxDMA buffer#1. This flag is de-asserted by the hardware after all data is completely read.</p>
BA+0x0138	<p>The data counter of TxDMA buffer inside TxDMA2TOEIF DG_TXDMA_BUFFCNT_INTREF_OFFSET</p> <p>Rd – [13:0]: Data counter of TxDMA buffer to show the amount of data in 512-bit unit</p>
<b>BA+0x0140 – BA+0x015F: TOE2RxDMAIF (Write/Read access)</b>	
BA+0x0140	<p>TOE2RxDMAIF command DG_RXDMA_COMMAND_INTREG_OFFSET</p> <p>Wr – [0]: Start TOE2RxDMAIF operation. Asserted to '1' to start transferring data on TOE2RxDMAIF. This flag is auto-cleared. Rd – [0]: Busy flag of TOE2RxDMAIF. Asserted to '1' when TOE2RxDMAIF is operating.</p>
BA+0x0148	<p>Total transfer size of TOE2RxDMAIF DG_RXDMA_TRNSIZE_INTREG_OFFSET</p> <p>Wr – [31:0]: Total transfer size in 512-bit unit. Valid range is 1 - 0xFFFFFFFF. Rd – [31:0]: Current amount of transferred data in 512-bit unit.</p>
BA+0x0150	<p>The size of each Rx buffer area DG_RXDMA_BUFFSIZE_INTREG_OFFSET</p> <p>Wr – [31:0]: The size of each Rx buffer area in 512-bit unit. Valid range is 0x80 - 0xFFFFFFFF80. This value must be aligned to 64 or 8Kbyte unit. Rd – [31:0]: The size of each Rx buffer area in 512-bit unit.</p>
BA+0x0158	<p>The control and status of Rx buffer DG_RXDMA_BUFFCTRL_INTREG_OFFSET</p> <p>Wr – [1:0]: Clear buffer. Asserted to '1' to clear the valid flag. After that, the Rx buffer is ready for storing new data. Each bit is the index of Rx buffer area. [0]-Rx buffer#0, [1]-Rx buffer#1. Rd – [1:0]: Buffer valid status. '0'-Rx buffer is free or not ready, '1'-Data in Rx buffer is valid for reading. Each bit is the index of Rx buffer area. [0]-Rx buffer#0, [1]-Rx buffer#1. This flag is asserted by the hardware after all data is completely written.</p>
BA+0x0160	<p>The data counter of FIFO inside TOE2RxDMAIF DG_RXDMA_FFCNT_INTREG_OFFSET</p> <p>Rd – [9:0]: Data counter of FIFO inside TOE2RxDMAIF to show the amount of data in 512-bit unit.</p>

### 3 The host software

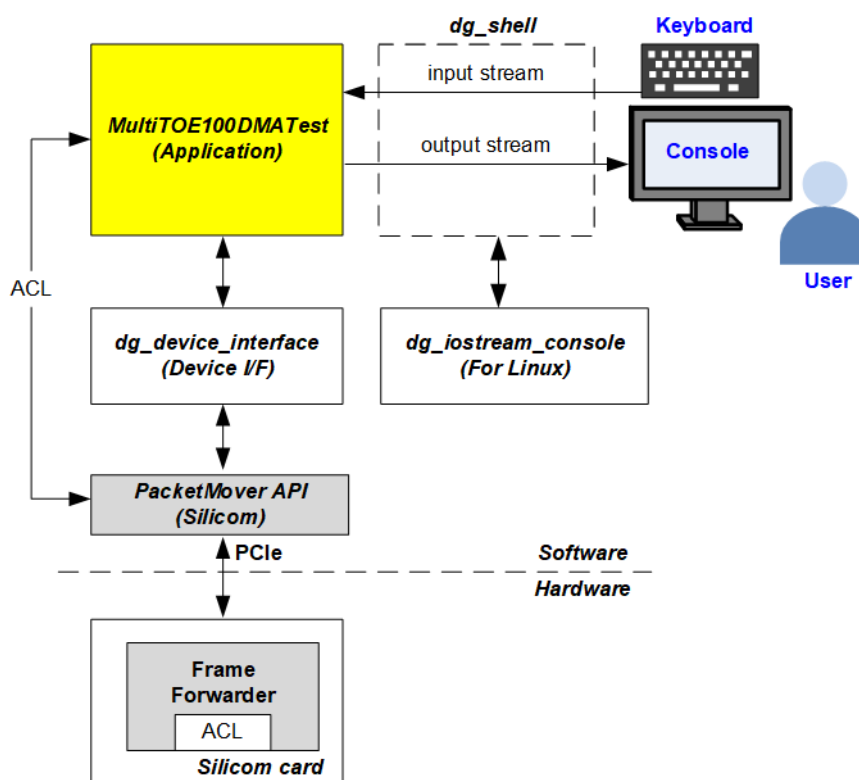


Figure 3-1 The software architecture in TOE100G-IP on Silicom PacketMover platform

To implement TOE100G-IP on Silicom PacketMover platform, the new host software (PacketMover\_dgsoftware) are designed. It consists of two software categories - the application (MultiTOE100DMATest) and the three frameworks (dg\_shell, dg\_iostream\_console, and dg\_device\_interface). “dg\_shell” controls the user input (keyboard) and the output console (monitor) by using “dg\_iostream\_console”. “dg\_iostream\_console” is designed by using specific command for LinuxOS to handle the input stream and the output stream by its own control sequence. While “dg\_device\_interface” is applied to control the hardware interface on Silicom card through PacketMover API. It includes the functions to write/read hardware registers, manage the device, and handle the process for memory allocation.

PacketMover API is a software interface for communicating with the hardware (FPGA) via PCIe interface. Generally, the application uses PacketMover API to access the hardware register via dg\_device\_interface. However, the application uses PacketMover API directly for Access Control List (ACL) to configure the Frame Forwarder. More details about the PacketMover API can be found from the following HTML file which is included in the Silicom released stuff.

>> 1\_4\_0\sw\PacketMover\_SW\_1\_4\_0\doc\index.html

This reference design uses PacketMover API version 1.4.0. Newer or former versions are currently not supported. Please contact our sales if other versions are required.

More details of the software on the demo are describes as follows.

### 3.1 Framework

This topic describes two software frameworks - the device interface and the shell. The device interface framework makes a simple function for utilizing PacketMover API (Silicom) to interface with the hardware. While the shell framework handles the input and output of the console (Linux terminal) for user interface.

#### 3.1.1 Device interface

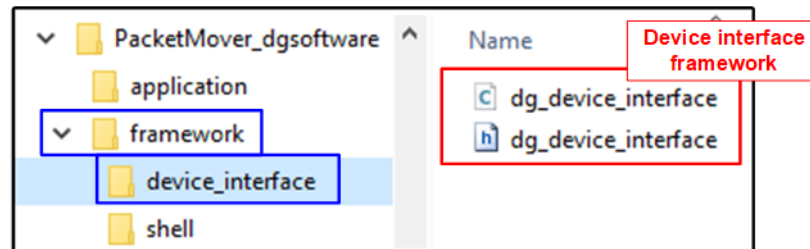


Figure 3-2 Device interface framework

The device interface is used by the application for communicating with the hardware kernels through PacketMover API provided by Silicom. The application uses the device interface to create a connection, allocate the buffer, and write/read the hardware registers. As shown in Figure 3-2, there are two source codes inside device\_interface directories.

- “dg\_device\_interface.h”: Declare functions which are defined in C source file (dg\_device\_interface.c). Also, it declares structs (object of variables) to use in this framework.
- “dg\_device\_interface.c”: Declare functions which are not applied by others. It is designed for the general hardware interface such as connecting the device, accessing the hardware register, connecting user logic with DMA channel, and allocating/de-allocating the memory.

The following functions use “ext\_dma\_example.c” which is Silicom’s example code to be a reference code. These functions are not modified or slightly modified to use the multiple DMA channels.

- static void \_CheckFpgaPanics(SC\_DeviceId deviceId, const char \* fileName, int lineNumber)
- static SC\_Error ErrorHandler(const SC\_ErrorInfo \* pErrorInfo)
- static bool \_NotOverlappingMemory(const char \* name1, uint64\_t start1, uint64\_t end1, const char \* name2, uint64\_t start2, uint64\_t end2)
- static bool ExternalMemoriesSanityCheck(const ExternalMemories \* pExternalMemories)
- static bool AllocateExternalMemories(DeviceInterface\* this, uint32\_t number)
- static bool FreeExternalMemories(DeviceInterface\* this, uint32\_t number)
- int \_\_PrintFunction(void \* pContext, const char \* format, ...)
- static void InitializeCommonInputParameters(DeviceInterface\* this, SL\_DmaChannelCommonInputParameters\* pCommon, uint32\_t number)

*Note: The number of TCP sessions by TOE100G-IP (MAXIMUM\_SESSION\_NUMBER) is defined as the constant in the software header code – “2”. The value must be updated if the user changes the number of supported sessions in the hardware.*

The new function lists of the device interface framework are described as follows.

### Device Connection

DeviceInterface * DevIf_CreateDeviceInterface(void)	
Parameters	None
Return value	The struct (object) of device interface. There is one member called ReturnCode. ReturnCode=True when it is completely connected. Otherwise, ReturnCode is equal to False.
Description	Use PacketMover API functions to connect with the hardware platform to retrieve the device information such as the device ID. The device ID is applied for writing/reading the hardware register. After that, it allocates multiple blocks of external memories for later used in DMA application.

bool DevIf_CloseDeviceInterface(DeviceInterface* this)	
Parameters	this: Pointer to device interface object to be cleaned up
Return value	True: The operation is successful False: The operation failed
Description	Use PacketMover API functions to disconnect the user logic and DMA channel (if there is a connection), free the memory that is used in DMA, close the connection with the hardware platform, and finally destruct the object.

### Write/Read Register

uint32_t DevIf_ReadIntReg(DeviceInterface* this, uint32_t index)	
Parameters	this: Pointer to device interface object index: User logic register index (0-0x1FFFF)
Return value	Read value from the hardware register
Description	Use PacketMover API function to read the data in the hardware register, specified by the index input. The actual address, 8-byte unit, is calculated from the index input (Register index = Register byte address in Table 2-1 / 8). Finally, return the read data to user.

void DevIf_WriteIntReg(DeviceInterface* this, uint32_t index, uint32_t value)	
Parameters	this: Pointer to device interface object index: User logic register index (0-0x1FFFF) value: 32-bit unsigned value for writing to the register
Return value	None
Description	Use PacketMover API function to write the value input to the hardware register, specified by the index input. The actual address, 8-byte unit, is calculated from the index input (Register index = Register byte address in Table 2-1 / 8).

User logic Management

bool DevIf_MountDMAUserLogic(DeviceInterface* this, uint32_t number, uint16_t UserLogicChannelNumber)	
Parameters	this: Pointer to device interface object number: Index of DMA channel in the device interface object UserLogicChannelNumber: Index of user logic channel to create
Return value	True: The operation is successful False: The operation failed
Description	Use PacketMover API functions and the utilization functions to allocate the new user logic channel by using UserLogicChannelNumber input. After that, set and initialize the DMA parameters and options of the DMA channel, selected by number input. Finally, construct and start the DMA channels for both Tx and Rx operation.

bool DevIf_DemountDMAUserLogic(DeviceInterface* this, uint32_t number)	
Parameters	this: Pointer to device interface object number: Index of DMA channel in the device interface object
Return value	True: The operation is successful False: The operation failed
Description	If the DMA channel (indexed by number input) is allocated, use PacketMover API functions and the utilization functions to stop the DMA channel, de-allocate user logic channels, and de-allocate the memories.

### 3.1.2 Shell

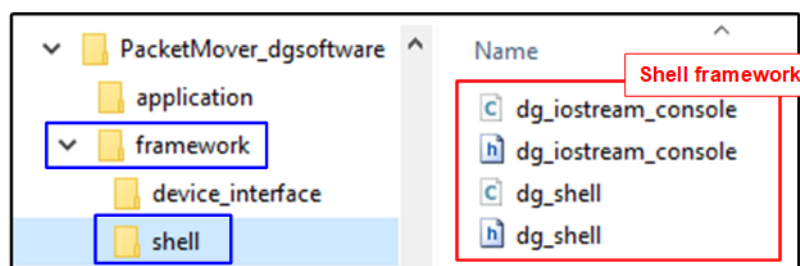


Figure 3-3 Shell framework

The shell framework (`dg_shell`) handles the input and output stream on the Linux terminal (Console). It retrieves keyboard input, manages the input string, parses the input data type, and prints a string out to console. The shell framework uses an I/O stream console library (`dg_iostream_console`) to work with the Linux terminal, i.e., changing the terminal environment, getting the user keyboard input, and pushing the printed string output to terminal.

As shown in Figure 3-3, there are two source codes for handling I/O stream console.

- “`dg_iostream_console.h`”: Declare functions and objects (structs) which are defined and used in C source file (`dg_iostream_console.c`).
- “`dg_iostream_console.c`”: Design the general function to manage the input and output stream on the Linux terminal environment such as writing a string on console, changing the terminal setting for utilizing by the shell, restoring the terminal setting to the original one, and getting the input character from the user through terminal. The function lists of the I/O stream classes are described as follows.

*Note: “KeyPressEnum” is a C enumeration declared in the header file (`dg_iostream_console.h`). It contains the keyboard input type for processing in the shell framework which are NORMAL, BACKSPACE, LEFTARROW, RIGHTARROW, DELETE, TAB, EOL, and CONTROL.*

OutStreamConsole class

void OutStreamConsole_write(OutStreamConsole* self, const char* s, uint32_t numChars)	
Parameters	self: Pointer to OutStreamConsole object (unused) s: Pointer to the character for printing out on the console numChars: The character length of "s"
Return value	None
Description	Call function (fwrite) to write the output (stdout) by the character "s" which specifies the length from "numChars". Next, flush the output to the terminal.

void OutStreamConsole_erase(OutStreamConsole* self, uint32_t numChars)	
Parameters	self: Pointer to OutStreamConsole object (unused) numChars: The number of characters to delete from the terminal
Return value	None
Description	Delete the currently displayed character on the terminal. The number of characters to erase is defined by "numChars".

InStreamConsole class

void InStreamConsole_NewSetting(InStreamConsole* self)	
Parameters	self: Pointer to InStreamConsole object
Return value	None
Description	Change the Linux terminal setting to non-echo mode and to process an input from the terminal without newline character. The original setting is stored to a local variable for restoring later.

void InStreamConsole_RestoreSetting(InStreamConsole* self)	
Parameters	self: Pointer to InStreamConsole object
Return value	None
Description	Restore the Linux terminal setting to the original setting by using a local variable.

bool InStreamConsole_getChar(InStreamConsole* self, char* pChar)	
Parameters	self: Pointer to InStreamConsole object (unused) pChar: Pointer to store the input character
Return value	None
Description	Get a character input from the Linux terminal and write to the pointer. When using this function, it waits until an input is received.

void InStreamConsole_FlushInputStream(InStreamConsole* self)	
Parameters	self: Pointer to InStreamConsole object (unused)
Return value	None
Description	Flush the input stream for the Linux terminal. This function is recommended to use before using "getChar" function.



int InStreamConsole_GetInputCharLength(InStreamConsole* self)	
Parameters	self: Pointer to InStreamConsole object (unused)
Return value	Number of input characters in the Linux terminal buffer
Description	Get the number of user input characters in the Linux terminal buffer.

KeyPressEnum InStreamConsole_getKeyPress(InStreamConsole* self, char c)	
Parameters	self: Pointer to InStreamConsole object c: Character input to determine the character type
Return value	NORMAL: General character that can be printed BACKSPACE, RIGHTARROW, LEFTARROW, DEL, TAB, EOL: Special characters that have specific operation CONTROL: Control character that does not have the operation
Description	Determine the type of the input character and return the value.

The shell framework has two source codes, described as follows.

- “dg\_shell.h”: Similar to the “dg\_istream\_console.h” header file, it declares functions and objects (structs) which are defined and used in C source file (dg\_shell.c).
- “dg\_shell.c”: Design the general function to simplify the input and output console management function and to provide a utility function such as parsing a string to an unsigned integer. Some functions are declared as a static in this file because it is not designed to use or call by user. The function lists of the shell framework are described as follows.

### General Function

void Shell_Initialise(Shell* self, InStreamConsole* inputStream, OutStreamConsole* outputStream)	
Parameters	self: Pointer to Shell object inputStream: The input stream object outputStream: The output stream object
Return value	None
Description	Load the pointers of the input stream object and the output stream object to the local variables for using in the shell framework.

static void Shell_ClearInputBuffer(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Clear the input buffer which is the internal variable.

char * Shell_GetStringPointer(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	Character pointer that points to the first character of the string
Description	Map the internal character buffer that is used to store input stream to be the first pointer of the string.

<b>bool Shell_GetInputLine(Shell* self)</b>	
Parameters	self: Pointer to Shell object
Return value	True: The operation is successful. False: Fail to retrieve an input character using “getChar” function.
Description	Clear the input buffer and the input stream by using “Shell_ClearInputBuffer” and “InStreamConsole_FlushInputStream”. After that, receive the input from the terminal (use “InStreamConsole_getChar”). Return “False” if the operation failed. Otherwise, return “True” and then process the input by using “Shell_ProcessInputChar”. The input is read and processed until end-of-line is detected.

<b>bool Shell_FlushInputBuffer(Shell* self)</b>	
Parameters	self: Pointer to Shell object
Return value	True: The operation is successful (Always returns this value).
Description	Flush the input buffer by calling “InStreamConsole_FlushInputStream”. This function is applied to map the function of “dg_iostream_console” to be used by the application.

<b>bool Shell_IsAnyInputKey(Shell* self)</b>	
Parameters	self: Pointer to Shell object
Return value	True: Some inputs are received from the Linux terminal. False: No received input from the terminal
Description	Read the number of received input from the terminal by using “InStreamConsole_GetInputCharLength”. Return “True” if the length is greater than zero. Otherwise, return “False”.

<b>int Shell_printf(Shell* self, const char* fmt, ...)</b>	
Parameters	self: Pointer to Shell object fmt: String that contains the text to be printed on the console arguments: Additional arguments
Return value	Number of input characters of the output buffer
Description	This function is called to receive the input string with the argument and then calculate the length for writing to the output stream. It is almost similar to standard “printf” function, but displaying to the console through the output stream.

### Input Parsing Function

<b>bool parseUInt32(Shell* self, char* pInputstr, uint32_t* pValue)</b>	
Parameters	self: Pointer to Shell object (unused) pInputstr: Pointer to the input string for processing pValue: Pointer of 32-bit result after parsing
Return value	True: The operation is successful False: Fail to parse the input or other errors
Description	Convert the input string of the decimal value to be 32-bit unsigned value. The input range must not be more than FFFF_FFFFh.

<b>bool Shell_parseHex32(Shell* self, char* pInputstr, uint32_t* pValue)</b>	
Parameters	self: Pointer to Shell object (unused) pInputstr: Pointer to the input string for processing pValue: Pointer of 32-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Validate the input and then convert the input string of the hex value to be 32-bit unsigned value. User must add "0x" before the integer string (0-9 and A-F) for the hexadecimal input. The input range must not be more than FFFF_FFFFh.

<b>bool Shell_parseUInt64(Shell* self, char* pInputstr, uint64_t* pValue)</b>	
Parameters	self: Pointer to Shell object (unused) pInputstr: Pointer to the input string for processing pValue: Pointer of 64-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Convert the input string of the decimal value to be 64-bit unsigned value. The input range must not be more than FFFF_FFFF_FFFF_FFFFh.

<b>bool Shell_parseHex64(Shell* self, char* pInputstr, uint64_t* pValue)</b>	
Parameters	self: Pointer to Shell object (unused) pInputstr: Pointer to the input string for processing pValue: Pointer of 64-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Convert the input string of the hex value to be 64-bit unsigned value. User must add "0x" before the integer string (0-9 and A-F) for the hexadecimal input. The input range must not be more than FFFF_FFFF_FFFF_FFFFh.

<b>bool Shell_get_input_long(Shell* self, uint64_t* pValue)</b>	
Parameters	self: Pointer to Shell object pValue: Pointer of 64-bit result of the input from the terminal
Return value	True: The operation is successful False: Fail to retrieve an input character, to parse the input string, or other errors
Description	Call "Shell_GetInputLine" function to retrieve a string input and then call "Shell_parseUInt64" or "Shell_parseHex64" to parse the input, depending on the input format. Finally, return the result after parsing.

<b>bool Shell_get_ipv4_addr(Shell* self, uint32_t* pValue)</b>	
Parameters	self: Pointer to Shell object pValue: Pointer to return 32-bit IPv4 address value that is received from the terminal
Return value	True: The operation is successful False: Fail to retrieve an input character, to parse the input string, or other errors
Description	Call “Shell_GetInputLine” function to retrieve a string input and then use “Shell_parseIPv4” to verify the input format. Error is returned if the input is invalid. Otherwise, the input is converted to 32-bit unsigned value to be the returned result.

<b>bool Shell_parseIPv4(Shell* self, char* pInputstr, uint32_t* pValue)</b>	
Parameters	self: Pointer to Shell object pInputstr: Pointer to the input string for processing pValue: Pointer to return 32-bit IPv4 address value that is processed from pInputstr
Return value	True: The operation is successful False: Fail to parse the input string or other errors
Description	Check the input that it is IPv4 address string format. If it is valid, the input is converted to be 32-bit unsigned value as a returned result. Otherwise, the error is returned.

Input Key Processing Function

<b>static void Shell_ProcessInputChar(Shell* self, char c)</b>	
Parameters	self: Pointer to Shell object c: Character input
Return value	None
Description	Use “InStreamConsole_getKeyPress” (function of InStreamConsole) to determine the character type. Next, call the function depending on the character type.

<b>static void Shell_ProcessNormalChar(Shell* self, char c)</b>	
Parameters	self: Pointer to Shell object c: Character input
Return value	None
Description	Add the new character to the buffer and then print to the output stream.

<b>static void Shell_ProcessBackspace(Shell* self)</b>	
Parameters	self: Pointer to Shell object
Return value	None
Description	Delete the left side character and then print to the output stream.

static void Shell_ProcessEOL(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Add NULL to the buffer and then print to the output stream. After that, set the local variable that show the end of line flag to be "True". When "Shell_GetInputLine" detects the end of line flag, the input buffer will be cleared.

static void Shell_ProcessTab(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Move the console cursor to the end of line input string.

static void Shell_ProcessLeftArrow(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Move the console cursor to the left side for one position.

static void Shell_ProcessRightArrow(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Move the console cursor to the right side for one position.

static void Shell_ProcessDel(Shell* self)	
Parameters	self: Pointer to Shell object
Return value	None
Description	Delete the character at the current position of the console and then print to the output stream.

### 3.2 Application

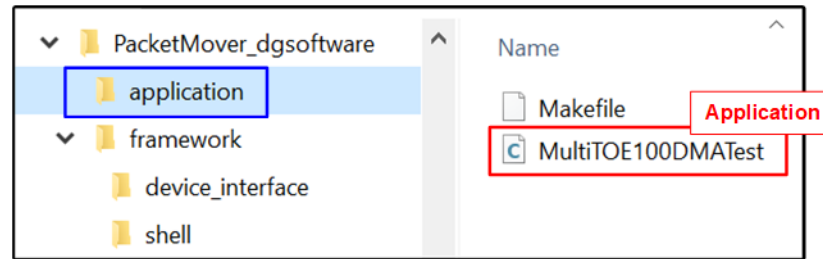


Figure 3-4 Application layer

The source code of the application is “MultiTOE100DMATest.c” file, as shown in Figure 3-4. The main function is operated by following steps.

#### Preparing Linux environment

- 1) Check whether the program is run as root or not. Terminate if it is not with root privilege.
- 2) Examine the physical connection of the network interface (“feth0”). Terminate if the connection is not up or not running.
- 3) Start a signal handler to detect “CTRL+C” input from the user. If the key is found, the termination process is run as below.
  - i) Reset the hardware in user logic area (TOE100G-IPs).
  - ii) Delete the packet filter rules in ACL (Access Control Lists).
  - iii) Close the device interface.
  - iv) Restore the terminal setting to the original setting.
- 4) Start a signal thread handler to detect user signal (“SIGUSR1”) which will be used later in a test sequence for handling multi-threading.
- 5) Setup the system to interface with the Linux terminal.
  - i) Initialize the I/O stream and change terminal setting.
  - ii) Initialize the shell and then connect the shell to I/O stream.

#### Preparing the hardware system

- 1) Retrieve the IPv4 address and MAC address that have been set to the network interface (“feth0”). Next, set the global variables (IPv4 and MAC address) in this application by these retrieved values.
- 2) Connect the hardware platform through device interface framework. Setup the system to interface with the hardware in the target device.
- 3) Examine that TOE100G-IP is available in the hardware kernel and print the IP information.
- 4) Use the Device interface to allocate two user logic channels with DMA features for both Tx and Rx operation. Each user logic channel is applied for one TOE100G-IP connection to operate one TCP session. The memory for DMA transferring is split for Tx operation and Rx operation. Tx buffer for Tx operation uses 256 Kbyte region while Rx buffer for Rx operation uses 1 Mbyte region.

TOE100G-IP operation

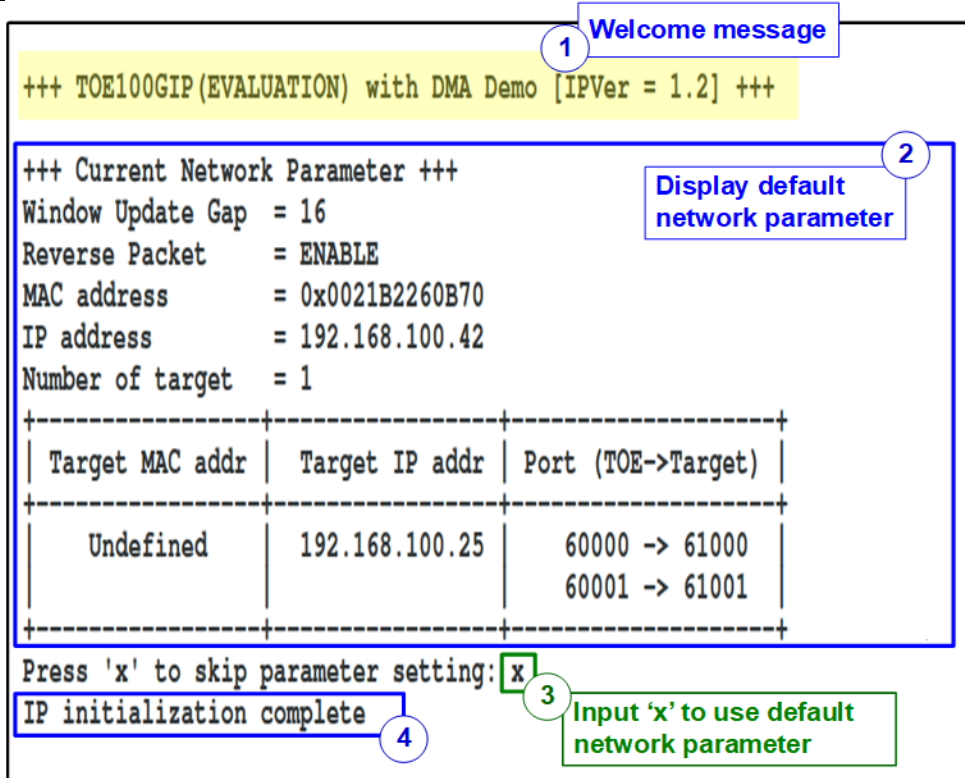


Figure 3-5 System initialization in Client mode by using default parameters

- 1) Start the test operation of TOE100G-IP by displaying the welcome message.
- 2) Display the default parameters in the Client initialization mode. The MAC address and the IP address are the value that retrieve from the network interface. The target MAC address is not yet defined and displayed as “Undefined” while The target IP address shows the default value.
- 3) User enters ‘x’ to start initialization process on Silicom NIC system with TOE100G-IP by using the default parameters or enters other keys to change some parameters (more details are described in topic 3.2.2 Reset menu). After that, start TOE100G-IP initialization by following step.
  - i) Assert reset to TOE100G-IP and then delete ACL filter rules.
  - ii) Load the target MAC address from the ARP table cache. If the target IP address is not found in the table, it will send ARP request to obtain the MAC address.
  - iii) Set the network parameters and test parameters to TOE100G-IP and de-assert reset to TOE100G-IP. After that, the IP starts initialization process.
  - iv) Wait until the TOE100G-IP finishes the initialization process (DG\_TOEIP\_CMD\_INTREG\_OFFSET[TOE index][0]='0').
  - v) Set ACL filter rules.

- 4) Display complete message and the main menu on the console. There are four test operations for user selection. More details of each menu are described as follows.

```

Press 'x' to skip parameter setting: x
IP initialization complete

--- TOE100G-IP menu ---
[0] : Display TCPIP parameters
[1] : Reset TCPIP parameters
[2] : Half duplex Test (TOEIP - Target)
[3] : Full duplex Test (TOEIP <-> Target)
>>>
  
```

Figure 3-6 Main menu of the software application

### 3.2.1 Display parameters

This menu is applied to display current parameters of the system. There are two groups of parameters, common parameter, and target parameters. The parameters of each group are described in more details as follows.

- 1) Common parameters: Windows update threshold, Reverse packet enable, source MAC address, and source IP address.
- 2) Target parameters: destination MAC address, destination IP address, source port, and destination port.

The sequence of display parameters menu are as follows.

- 1) Read all network parameters from each variable in the software application.
- 2) Print out each variable. For the common parameters, it is printed in line format while the target parameters are printed in the table format



### 3.2.2 Reset IP

This menu is applied to change network parameters of TOE100G-IPs such as port numbers. After updating the network parameters on TOE100G-IP, the IP requires to re-initialize. Besides, the software application needs to handle the ACL rules. The sequence to reset IP is as follows.

*Note: Changing IP address and MAC address of TOE100G-IP are not supported because they retrieve from the network interface. It needs to change on the Linux console by “ifconfig” before running the software application.*

- 1) Display current parameter value on the console.
- 2) Ask user to skip (use current parameters) or set new parameter values.
  - i) Press ‘x’ to use the current parameters and skip to step 5.
  - ii) Press other keys to start parameters setting in the next step.
- 3) Receive new common parameters from user, i.e., Window Update Gap and Reverse packet enable. If an input value is invalid, the parameter is not changed.
- 4) Receive the number of targets and parameters of each target from user, i.e., number of sessions, Target IP address, and port numbers. If an input value is invalid, the input is not changed except for the number of sessions which must be inputted by valid value.
- 5) Force reset to all IPs by setting DG\_TOEIP\_RST\_INTREG\_OFFSET[index][0]='1'.
- 6) Delete all ACL rules (packet filter rules) that have been set on the system by the software application. It simply calls “deleteACLFilterRules” (described in topic 3.2.5).
- 7) Load Target MAC address by following step.
  - i) Use “getTargetMacAddress” (described in topic 3.2.5) to get the target MAC address from ARP cache. Continue to step 8 if all target devices can be discovered. Otherwise, proceed the next step.
 

*Note: If the operation is successful, global variables who store the target MAC addresses will be updated.*
  - ii) Use “sendARPRequest” (described in topic 3.2.5) to send the ARP request to the target device to get the target MAC address.
  - iii) Wait for 2 seconds. Continue to next step if the target MAC address is returned from the ARP reply. Otherwise, return to step 7i) to re-process the Target MAC address.
- 8) Set all parameters to TOE100G-IPs register such as DG\_TOEIP\_SML\_INTREG\_OFFSET[index] and DG\_TOEIP\_SMH\_INTREG\_OFFSET[index]. The initialization mode of all TOE100G-IP in this design is fixed to “FIXED MAC” mode, assigned by DG\_TOEIP\_SRV\_INTREG\_OFFSET[index].
- 9) De-assert IP reset by setting DG\_TOEIP\_RST\_INTREG\_OFFSET[index][0]='0'. After that, TOE100G-IP starts the initialization process.
- 10) Wait until all IPs complete the initialization process by monitoring IP busy flag (DG\_TOEIP\_CMD\_INTREG\_OFFSET[index][0]) until all initialization process are completed (busy flag is de-asserted to '0').
- 11) Add new ACL rules with new parameters by using “addACLFilterRules” (described in topic 3.2.5).

### 3.2.3 Half duplex test

This menu transfers data in single direction for the initialized session. The user sets transfer mode to be send data, receive data, or no operation to each session individually. Next, the test parameters are received from user. The operation is cancelled if some inputs are invalid.

*Note: The parameters for send data test and receive data test are shown as follows.*

*Send data test - Total transfer data size, packet size, enable/disable flag of pattern generation, and connection mode (active mode or passive mode).*

*Receive data test – Total transfer data size, enable/disable flag of data verification mode, and connection mode (active mode or passive mode).*

Before transferring data, the software application reads the test parameters to calculate the number of used buffers and the last buffer length for an unaligned-size transfer. Next, the main thread creates a child thread (gen\_txbuf\_data for send data test or ver\_rxbuf\_data for receive data test) to handle Tx buffer for sending the data or Rx buffer for receiving the data. The flow control signals of Tx buffer and Rx buffer are different.

Three counters are applied to be flow control signals of Tx buffer. The first counter is request counter which is applied to start the child thread for writing data to Tx buffer. This counter is increased by the main thread when the Tx buffer area is not full. The second counter is increased by the child thread to be the write counter after finishing filling the data to each Tx buffer area. The child thread starts sending the data when the first counter is more than the second counter. The last counter is completion counter that is increased by the main thread when the hardware finishes reading data of each Tx buffer area, monitored by ready flag that is de-asserted by the hardware. The full status of Tx buffer is calculated from the different value of the request counter and the completion counter.

Similarly, Rx buffer operation is controlled by three counters. First is request counter which is applied to start the child thread for reading data from Rx buffer. This counter is increased by the main thread when the hardware finishes writing data to each Rx buffer area, monitored by valid flag that is asserted by the hardware. Second is the read counter which is increased by the child thread after finishing reading the data of each Rx buffer area. The last counter is completion counter which is increased by the main thread after the main thread asserts clear flag to free Rx buffer area.

To support multiple TCP sessions, the main thread assigns the state variable to define the status for each session, i.e., WAIT\_CONN (waits the new connection created), CONNECTED (the connection is established and data can be transferred), CLOSED (the connection is terminated), and ERROR (error situation is detected). More details of the operation in Half duplex test menu for sending or receiving data are described as follows.

- 1) Display target IP address and port number of each session and then receive test parameters of that session from user. After that, validate all inputs. This step is repeated for operating the next active session until the parameters of all active sessions are received.
- 2) Display the recommended parameters of test application on PC following connection mode.
  - i) For active connection mode, the parameters for running test application on PC by server mode are displayed and then "Press any key to proceed" is displayed. It waits until user runs the test application on PC and then user enters some keys to continue the next step.
  - ii) For passive connection mode, the parameters for running test application on PC by client mode are displayed.
- 3) Open connection following connection mode setting.
  - i) For active open, the software application sets DG\_TOEIP\_CMD\_INTREG\_OFFSET= 2 (Open port) and sets current state variable to WAIT\_CONN.
  - ii) For passive open, the software application sets current state variable to WAIT\_CONN.
- 4) Setup the software and hardware for DMA function to send or receive data with the hardware. Repeat this step to setup all active sessions. The details of Send data test and Receive data test are described as follows.

#### Send data test

- i) Set total transfer size to DG\_TXDMA\_TRANSIZE\_INTREG\_OFFSET.
- ii) Start Tx DMA hardware by setting DG\_TXDMA\_COMMAND\_INTREG\_OFFSET[0]='1'.
- iii) Calculate the amount of data for the last buffer area and the total count of buffer areas for storing all data.
- iv) Store the test parameters to the arguments of the child thread (gen\_txbuf\_data).

#### Receive data test

- i) Set Rx buffer size (DG\_RXDMA\_BUFFSIZE\_INTREG\_OFFSET) to be equal to 1 MB.
  - ii) Set total transfer size to DG\_RXDMA\_TRANSIZE\_INTREG\_OFFSET.
  - iii) Clear the Rx buffer status by setting DG\_RXDMA\_BUFFCTRL\_INTREG\_OFFSET [1:0]="11".
  - iv) Start RX DMA hardware by setting DG\_RXDMA\_COMMAND\_INTREG\_OFFSET [0]='1'.
  - v) Calculate the amount of data for the last buffer area and the total count of buffer areas for storing all data.
  - vi) Store the test parameters to the arguments of the child thread (ver\_rxbuf\_data).
- 5) The initial state variable of both Send data test and Receive data test is equal to WAIT\_CONN. In this state, it waits until ConnOn signal changes its value (DG\_USER\_INT\_INTREG\_OFFSET[0]='1'). After that, the current state variable changes to CONNECTED and creates a child thread (gen\_txbuf\_data or ver\_rxbuf\_data) for sending data or receiving the data in buffers.

- 6) The data can be transferred while the state variable is CONNECTED. The operation of Send data test is completed when total data is transmitted. While the operation of Receive data test is completed when the connection is terminated by the target system. After finishing the operation, the current state variable changes to CLOSED. If some errors are found, the current state variable enters to ERROR. More details of each transfer direction are described as follows.

#### Send data test

The software application to operate Send data test controls the TOE100G-IP operation and the Tx DMA operation independently. The operation is finished when the remaining length variable is equal to 0 and Tx DMA operation is finished.

- i) Read TOE100G-IP busy flag (DG\_TOEIP\_CMD\_INTREG\_OFFSET[0]), the connection status (DG\_USER\_STS\_INTREG\_OFFSET[0]), the remaining transfer length variable, and Tx DMA status (DG\_TXDMA\_COMMAND\_INTREG\_OFFSET[0]).
  - a) If TOE100G-IP busy flag is asserted to '1', skip to step ii). Otherwise, goes to step b).
  - b) If the connection status is OFF, change the state variable to ERROR. Otherwise, goes to step c).
  - c) If the remaining transfer length variable is equal to 0 and Tx DMA status is de-asserted to '0', it means the operation is completed. Run active close command by setting DG\_TOEIP\_CMD\_INTREG\_OFFSET=3 and change the state variable to CLOSED. If TOE100G-IP busy flag after sending close command is not asserted to '1', the state variable is set to ERROR. Continue to step d) if the operation is not finished.
  - d) Read the remaining length variable and calculate total transfer size of TOE100G-IP in this round. Start TOE100G-IP send operation by setting packet size (DG\_TOEIP\_PKL\_INTREG\_OFFSET), total transfer size (DG\_TOEIP\_TDL\_INTREG\_OFFSET), and Send command (DG\_TOEIP\_CMD\_INTREG\_OFFSET=0), respectively. After that, the remaining length is decreased by total transfer size value.
 

*Note: The total transfer size of each round is set to 0xFFFF\_FFC0 (maximum length with 64-byte alignment), except the last round that is set to the remaining length.*
- ii) Run Tx DMA operation by controlling three counters independently, described as follows.
  - a) If the different value of the request counter and the completion counter is less than 2 (the number of Tx buffer area in this reference design), the request counter is increased by the main thread to send the new request to the child thread. Also, the ready flag of the new Tx buffer area (DG\_TXDMA\_BUFFCTRL\_INTREG\_OFFSET[i] where 'i' is an index of Tx buffer area) is asserted to '1' to allow the hardware to read the data.
  - b) The child thread (gen\_txbuf\_data) starts filling the data to the new Tx buffer area when the request counter is more than the write counter. After finishing filling data, the write counter is increased.
  - c) When the hardware finishes reading all data from each Tx buffer area, the ready flag (DG\_TXDMA\_BUFFCTRL\_INTREG\_OFFSET[i] where 'i' is an index of Tx buffer area) will be de-asserted to '0'. If the ready flag of the latest read position is de-asserted to '0', the completion counter is increased.

### Receive data test

The software application to operate Receive data test controls the TOE100G-IP operation and the Rx DMA operation independently. The operation is finished when the connection status is OFF and Rx DMA operation is finished.

- i) Read the connection status (DG\_USER\_STS\_INTREG\_OFFSET[0]) and Rx DMA status (DG\_RXDMA\_COMMAND\_INTREG\_OFFSET[0]). If the connection is OFF and Rx DMA is not busy, it means the read operation is completed and then the state variable enters to CLOSED. Otherwise, continue to step ii) for transferring data.
- ii) Run Rx DMA operation by controlling three counters independently, described as follows.
  - a) When the hardware finishes writing data to read Rx buffer area, it will assert the valid flag (DG\_RXDMA\_BUFFCTRL\_INTREG\_OFFSET[i] where 'i' is an index of Rx buffer area) to '1'. If the main thread detects the new valid flag of the latest position asserted and the different value of the request counter and the completion counter is less than 2 (the total Rx buffer area), the request counter will be increased by the main thread for sending the new request to the child thread.
  - b) The child thread (ver\_rxbuf\_data) starts reading the data from Rx buffer area when the request counter is more than the read counter. The data is verified with the expected value if the verification flag is enabled. After finishing, the read counter is increased.
  - c) If the read counter is more than the completion counter, the main thread sets the register to clear the valid flag of Rx buffer (DG\_RXDMA\_BUFFCTRL\_INTREG\_OFFSET[i] where 'i' is an index of Rx buffer area). After that, the completion counter is increased.

After the state variable of all request sessions are not equal to WAIT\_CONN, the connection status and the progress of the test operation of all request sessions are displayed on the console every second. If the data is still transferred, total amount of transmitted data and received data calculated by the software application are displayed on the console.

- 7) Clear the interrupt from the connection status (DG\_USER\_INT\_INTREG\_OFFSET[0]='1') and wait until the child thread (gen\_txbuf\_data or ver\_rxbuf\_data) finishes the operation.
- 8) Check data verification fail flag for the session that runs Receive data test. Display error message if the error is found. Also, check the remaining data from Tx DMA hardware (DG\_TXDMA\_BUFFCNT\_INTREG\_OFFSET) and Rx DMA hardware (DG\_RXDMA\_FFCNT\_INTREG\_OFFSET). Display warning message if there are remaining data stored in the buffer or FIFO.
- 9) Calculate performance and show test result on the console. After that, de-allocate the memory that are used for the thread arguments.

### 3.2.4 Full duplex test

This menu transfers data between the Silicom NIC system and the target device in both directions. Four inputs are received from user, i.e., total transfer data size, packet size, enable/disable flag of pattern generation/verification, and connection mode (active mode or passive mode). When running the test, the transfer size that is set on both the Silicom NIC system and the target device must be equal. If the target device is PC that runs “tcp\_client\_txrx\_single” application, the connection mode of the Silicon NIC system must be set to passive.

Before transferring data, the software application reads test parameters to calculate the number of used buffers and the last buffer length for an unaligned-size transfer. Next, the main thread creates two child threads (gen\_txbuf\_data and ver\_rxbuf\_data) to handle Tx buffer and Rx buffer individually. Similar to Half duplex test, the flow control signals of Tx buffer and Rx buffer use three different counters. Please read more details of each counter from topic 3.2.3 Half duplex test.

To support multiple TCP sessions, the main thread assigns the state variable to define the status of each session - WAIT\_CONN, CONNECTED, CLOSED, and ERROR. Please see more descriptions of each state from topic 3.2.3 Half duplex test. More details of the operation in Full duplex test menu are described as follows.

- 1) Display target IP address and port number of each session and then receive test parameters of that session from user, similar to step 1) of Half duplex test.
- 2) Display the recommended parameters of test application on PC following connection mode.
  - i) For passive connection mode, the parameters for running test application on PC by client mode are displayed.
  - ii) For active connection mode, “Press any key to proceed” is displayed to wait until user enters some keys to continue the next step.
- 3) Open connection following connection mode setting, similar to step 3) of Half duplex test
- 4) Setup the software and hardware for DMA function to send and receive data with the hardware. Repeat this step to setup all active sessions. Please read more details from step 4) of Half duplex test for both Send data test and Receive data test.
- 5) The initial state variable is set to WAIT\_CONN to wait until ConnOn signal changes its value (DG\_USER\_INT\_INTREG\_OFFSET[0]='1'). After ConnOn is asserted (ON), the current state variable enters to CONNECTED and then two child threads (gen\_txbuf\_data and ver\_rxbuf\_data) are created for sending data and receiving the data in buffers.
- 6) The data can be transferred while the state variable is CONNECTED. The condition to check the operation completion of the active connection mode is different from the passive connection mode. In active connection mode, it waits until there is no remaining transmitted data while the passive connection mode waits until the connection status changes to OFF. Also, both Tx DMA status and Rx DMA status must be Idle for setting the current state variable to CLOSED. If some errors are found, the current state variable changes to ERROR.

The software application controls TOE100G-IP operation for transferring data in both transfer directions. While two child threads for handling Tx DMA operation and Rx DMA operation are run individually. More details for transferring data are described as follows.

- i) Read TOE100G-IP busy flag (DG\_TOEIP\_CMD\_INTREG\_OFFSET[0]), the connection status (DG\_USER\_STS\_INTREG\_OFFSET[0]), the remaining transfer length variable, and Tx DMA/Rx DMA status (DG\_TXDMA/RXDMA\_COMMAND\_INTREG\_OFFSET[0]). The sequence of this step after reading these status signals of active connection mode and passive connection mode is different, described as follows.

#### Active connection mode

- a) If TOE100G-IP busy flag is asserted to '1', skip to step ii). Otherwise, goes to step b).
- b) If the connection status is OFF, change the state variable to ERROR. Otherwise, goes to step c).
- c) If the remaining transfer length variable is equal to 0 and Tx DMA/Rx DMA status are de-asserted to '0', it means the operation is completed. Run active close command by setting DG\_TOEIP\_CMD\_INTREG\_OFFSET=3 and change the state variable to CLOSED. If TOE100G-IP busy flag after sending close command is not asserted to '1', the state variable is set to ERROR. Continue to step d) if the operation is not finished.
- d) Read the remaining length variable and calculate total transfer size of TOE100G-IP in this round. Start TOE100G-IP send operation by setting packet size (DG\_TOEIP\_PKL\_INTREG\_OFFSET), total transfer size (DG\_TOEIP\_TDL\_INTREG\_OFFSET), and Send command (DG\_TOEIP\_CMD\_INTREG\_OFFSET=0), respectively. After that, the remaining length is decreased by total transfer size of this round.

#### Passive connection mode

- a) If the connection status is OFF and Tx DMA/Rx DMA status are de-asserted to '0', it means the operation is completed. Change the state variable to CLOSED.
- b) If the remaining transfer length variable is not equal to 0 and TOE100G-IP busy flag is de-asserted to '0', it means TOE100G-IP is ready to send more data. Calculate total transfer size of TOE100G-IP in this round and start TOE100G-IP send operation by setting packet size (DG\_TOEIP\_PKL\_INTREG\_OFFSET), total transfer size (DG\_TOEIP\_TDL\_INTREG\_OFFSET), and Send command (DG\_TOEIP\_CMD\_INTREG\_OFFSET=0), respectively. After that, the remaining length is decreased by total transfer size of this round.

*Note: The total transfer size of each round for both Passive and Active connection mode is set to the maximum length with packet size alignment (maximum length is 0xFFFF\_FFC0, except the last round that is set to the remaining length).*

- ii) Run Tx DMA operation and Rx DMA operation by controlling three counters, similar to step 6ii) of Half duplex test (both Send data test and Receive data test).

After the state variable of all request sessions are not equal to WAIT\_CONN, the connection status and the progress of the test operation of all request sessions are displayed on the console every second. If the data is still transferred, total amount of transmitted data and received data calculated by the software application are displayed on the console.

- 7) Clear the interrupt from the connection status (DG\_USER\_INT\_INTREG\_OFFSET[0]='1') and wait until the child threads (gen\_txbuf\_data and ver\_rxbuf\_data) finish the operation.
- 8) Check data verification fail flag and Display error message if the error is found. Also, check the remaining data from Tx DMA hardware (DG\_TXDMA\_BUFFCNT\_INTREG\_OFFSET) and Rx DMA hardware (DG\_RXDMA\_FFCNT\_INTREG\_OFFSET). Display warning message if there are remaining data stored in the buffer or FIFO.
- 9) Calculate performance and show test result on the console. After that, de-allocate the memory that are used for the thread arguments.



### 3.2.5 Function list in application

This topic describes the function list to run TOE100G-IP operation.

#### NIC Utilization Function

<code>static bool checkLinkStatus(const char * interfaceName, bool * returnedStatus)</code>	
Parameters	interfaceName: Network interface name (“feth0”) returnedStatus: True – Ethernet link is up, False – Link is down.
Return value	True: The interface operation is successful False: Some errors are found
Description	Use socket and ioctl to check the network interface and the status. False is returned if it fails to handle the socket or the status is unmatched.

<code>static int getTargetMacAddress(uint32_t num_target, uint32_t num_session[], uint32_t * successTargetNum, uint32_t * sizeOfSuccessTargetNum)</code>	
Parameters	num_target: Total number of target devices num_session: Array that represents number of sessions in each target successTargetNum: Array to store target number that is successful sizeOfSuccessTargetNum: Pointer to array size of successTargetNum
Return value	0: The operation is successful -1: All MAC addresses are not fully obtained -2, -3: Fatal failure while opening ARP cache file, handling socket, or parsing string
Description	Open the ARP cache file and skip the first line of the file. Next, iterate through the file line by line to find the MAC address of our target devices. Check the network interface name, target IP address, and hardware type whether it is matched and correct or not. After that, convert the MAC address string from the matched line and load to the target MAC address and global variable. The target number that is found is stored to “successTargetNum” and the size of found target (sizeOfSuccessTargetNum) is updated. Next, scan the next index. Return 0 if all target devices are found. Return -1 when some of them or none are found. Return -3 or -2 when error is found.

<code>static int sendARPRequest(const char * interfaceName, uint32_t sourceIPAddress, uint32_t targetIPAddress)</code>	
Parameters	interfaceName: Network interface name (“feth0”) sourceIPAddress: IPv4 address of the corresponded network interface targetIPAddress: IPv4 address of the target device to discover
Return value	0: The operation is successful -1: The operation failed
Description	Open the socket and retrieve information of network interface such as Ethernet interface index, MAC address for preparing the ARP request packet. Next, construct the ARP request packet and fill the input target IPv4 address for discovering the target device. After that, send the packet through the socket. Finally, clean up the stuff such as socket.

static bool deleteACLFilterRules(void)	
Parameters	None
Return value	True: The operation is successful False: The operation failed
Description	Create a rule expression string for delete action with the rule number. After that, use PacketMover API to convert the string and set the ACL filter rule to the target device. If the packet filter rules have been set more than once, repeat this sequence until the filtered rules are deleted.

static bool addACLFilterRules(void)	
Parameters	None
Return value	True: The operation is successful False: The operation failed
Description	Convert IP and MAC address (global variables) to string format. Next, create a rule expression string to allow the ACL forwarding the packet that matches the given rule. The filter rule has several criteria, i.e., network interface number#0 only ("feth0"), MAC address, IPv4 address, TCP/IP protocol, and source port. Next, use PacketMover API to convert the string and set the ACL filter rule to the target device. If there are multiple sessions, repeat the sequence to set the rules for the next session until all filtered rules are added. Finally, count the number of added filter rules as a global variable to be later used in "deleteACLFilterRules" function.

Termination Function

void cleanup(void)	
Parameters	None
Return value	None
Description	This function is called for safety termination of the application. It resets the hardware system by setting DG_TOEIP_RST_INTREG_OFFSET to '1'. Next, delete the packet filter rules that have been set to ACL. After that, close the target device interface. Finally, restore the terminal setting by using the original setting.

void sigintHandler(void)	
Parameters	None
Return value	None
Description	This function is run to terminate the application when user input "CTRL+C". "cleanup" function is called and then the application is exited.

### General Function

static void get_toeindex(uint32_t *num_target, uint32_t *num_session)	
Parameters	num_target: Pointer to number of targets num_session: Pointer to the first element of the array indicating number of sessions in each target.
Return value	None
Description	Use “num_toe” which is a global variable for total number of active sessions to calculate number of targets and number of sessions in each target. Sessions that have the same target IP address are considered as the same target group and then total number of sessions in that target is more than one.

### Console Display Function

static uint32_t cal_strlen(uint32_t num)	
Parameters	num: 32-bit unsigned integer to be calculated
Return value	String length of integer input. Valid from 1 to 10.
Description	Calculate string length of the input when considered as string.

static void show_ipaddr(uint32_t ip_addr)	
Parameters	ip_addr: An IPv4 address in unsigned integer format to be displayed
Return value	None
Description	Display an IPv4 address from unsigned integer input. For example, the input “0xC0A86401” is printed as “192.168.100.1”.

static void show_perf_line(uint32_t *test_mode)	
Parameters	test_mode: Pointer to array that stores test mode of each session which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3)
Return value	None
Description	Print straight line format for active sessions which is a part of the displayed table.

static void show_perf_header(uint32_t *test_mode)	
Parameters	test_mode: Pointer to array that stores test mode of each session which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3)
Return value	None
Description	Print header of the performance table.

static void show_cursize(uint32_t *test_mode, uint32_t *cur_state, uint64_t *cur_sent, uint64_t *cur_recv)	
Parameters	test_mode: Pointer to array that stores test mode of each session which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3) cur_state: Pointer to array that stores current state in each session cur_sent: Pointer to array of the amount of current transmitted data in each session cur_recv: Pointer to array of the amount of current received data in each session
Return value	None
Description	Print status and current transfer size of the active sessions into the table format

static void show_result(uint32_t *test_mode, uint32_t *cur_state, uint64_t *tot_send, uint64_t *tot_recv, uint64_t *total_len, uint32_t *err_recv_ver)	
Parameters	test_mode: Pointer to array that stores test mode of each session which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3) cur_state: Pointer to array that stores current state in each session tot_send: Pointer to array of total amount of transmitted data in each session tot_recv: Pointer to array of total amount of received data in each session total_len: Pointer to array of setting transfer length in each session err_recv_ver: Pointer to array of received data verification status
Return value	None
Description	Display total amount of transmitted/received data, its status, and the performance of the active sessions. After that, calculate the sum of the amount of transmitted data and received data of each session to calculate total amount of transmitted data and received data for displaying. Finally, display the result (total amount of transmitted data and received data of all sessions) on the console in Byte, KByte, MByte, GByte unit. Also, calculate and display the average performance.

Thread Function

<b>static void sigThreadHandler(int sig)</b>	
Parameters	sig: signal input that sends to this function to be handled
Return value	None
Description	Call the signal handler to catch the signal for killing the thread next time. After that, kill the thread where the signal is detected by using "pthread_exit".

<b>static void *gen_txbuf_data(void *voidArg)</b>	
Parameters	voidArg is the struct that contains following arguments. <ul style="list-style-type: none"> <li>• pTxDmaChannel: Pointer to the Tx DMA channel (Silicom)</li> <li>• buf_reqcnt: Number of buffers that is requested for transferring data (this argument is updated by the main thread while operating)</li> <li>• buf_totalcnt: Total number of buffers to be written and sent</li> <li>• last_buf_len: The length of last buffer in this operation (byte unit)</li> <li>• gen_patt: TRUE-fill the pattern data, FALSE-not fill the pattern data</li> </ul>
Return value	None
Description	<p>To achieve the high-speed performance, the Tx packet size of Tx DMA operation is fixed to 8 Kbytes and Tx buffer size in the hardware is fixed to 256 Kbytes (32 times of packet size). The function is designed by using PacketMover API.</p> <p>The operation is finished when the write counter (buf_wrcnt) is more than or equal to total counter (buf_totalcnt). The operation of this thread is started when the request counter (buf_reqcnt) is more than the write counter (buf_wrcnt). The first operation is setting the buffer size to 256 Kbytes, except the last request that is set by last_buf_len. To transfer data to 256-Kbyte buffer, many packets may be transmitted. The packet size is fixed to 8 Kbyte size, except the last packet size which is set to be equal to the remaining size that may be less than 8 Kbytes. Next, the test data is filled to Tx buffer if gen_patt is set to TRUE. Otherwise, dummy data in Tx buffer is applied instead. After finishing filling data to the current buffer, flush all packets to the hardware. Finally, update the write counter by adding one to show the number of buffers that is completely filled by this child thread. The memory is de-allocated after finishing the operation.</p>

static void *ver_rxbuf_data(void *voidArg)	
Parameters	<p>voidArg is the struct that contains following arguments.</p> <ul style="list-style-type: none"> <li>• gen_patt: TRUE-fill the pattern data, FALSE-not fill the pattern data</li> <li>• pRxDmaChannel: Pointer to the Rx DMA channel (Silicom)</li> <li>• buf_reqcnt: Number of buffers that is requested for transferring data (this argument is updated by the main thread while operating)</li> <li>• buf_rdcnt: Number of buffers that is completely read by this function</li> <li>• buf_totalcnt: Total number of buffers to be received and read</li> <li>• last_buf_len: The length of last buffer in this operation (byte unit)</li> <li>• ver_en: TRUE-verify the pattern data, FALSE-not verify the pattern data</li> <li>• ver_fail: TRUE-verification failed, FALSE-verification successful</li> <li>• tot_recv_count: Amount of received data updated by this function (byte unit)</li> </ul>
Return value	None
Description	<p>Rx buffer size is set to 1 Mbytes (128 of 8192 bytes). The function is designed by using PacketMover API.</p> <p>The operation is finished when the read counter (buf_rdcnt) is more than or equal to total counter (buf_totalcnt). The operation is started when the request counter (buf_reqcnt) is not equal to the read counter (buf_rdcnt). The first operation is setting the buffer size to 1 Mbytes, except the last request that is set by last_buf_len. The data is read and verified when ver_en is set to TRUE. ver_fail is set to TRUE if the received data is not equal to the expected value. When ver_en is set to FALSE, the data is not verified. Each operation loop is completed when total amount of received data in each loop is equal to the buffer size. After receiving all data from the current buffer, update the read counter (buf_rdcnt) by adding one to show the number of buffers that is completely read by this child thread. The memory is de-allocated after finishing the operation.</p>

Miscellaneous Function

static void input_param(void)	
Parameters	None
Return value	None
Description	Receive test parameters from user for test parameters, i.e., Reverse packet enable, Windows update threshold, the number of targets, Target IP address, the number of sessions, FPGA port number, and Target port number. After receiving and verifying all parameters, the current value of all parameters is displayed by calling show_param function.

inline static uint32_t read_conon(uint32_t num)	
Parameters	num: An index of TOE100G-IP
Return value	0: Connection is OFF, 1: Connection is ON
Description	Read value from DG_USER_STS_INTREG_OFFSET register of the specified TOE100G-IP, defined by num input. After that, return only bit0 value to show connection status.

static void toe_tx_send(uint32_t toe_index, uint64_t *total_len, uint32_t *pac_size, uint32_t *round_size)	
Parameters	toe_index: An index of TOE100G-IP total_len: Pointer to the total transfer data size, used to send data pac_size: Pointer to the packet size, used for assignment round_size: Pointer to the round size, used to assign the maximum transfer size per command <i>Note: All size arguments are assigned in 64-byte unit.</i>
Return value	None
Description	Non-blocking function to assign the specific TOE100G-IP for send command. Before setting the transfer size in the IP, firstly set the packet size to the IP. After that, read the remaining transfer size (total_len). Total transfer size that is set to the hardware is equal to round_size when total_len is more than round_size. Otherwise, total transfer size to the hardware is equal to total_len. Finally, write the send command to TOE100G-IP to start the operation.

Test Function

static void show_param(void)	
Parameters	None
Return value	None
Description	Display the current value of the network parameters set to TOE100G-IPs such as IP addresses, MAC addresses, and port numbers, following described in topic 3.2.1.

static bool init_param(void)	
Parameters	None
Return value	True: The operation is successful False: Error is found
Description	This function is called to set the parameters and reset the IPs, following described in topic 3.2.2.

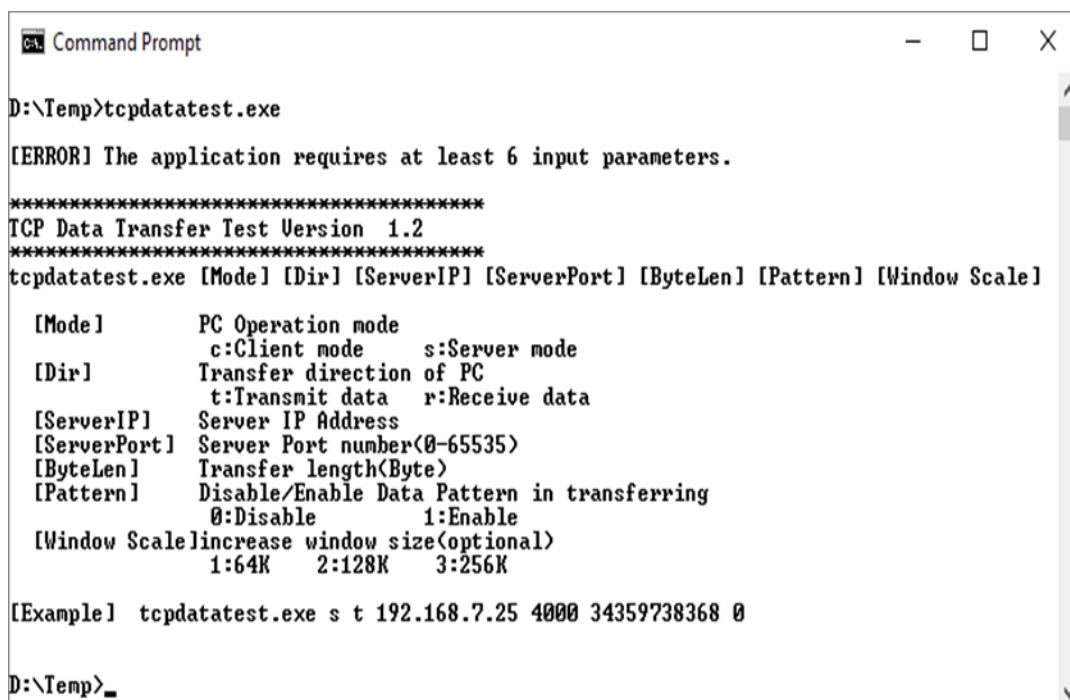
static int dma_half_test(void)	
Parameters	None
Return value	True: The operation is successful False: Receive invalid input or error is found
Description	Run half duplex test with DMA following description in topic 3.2.3. This function uses gen_txbuf_data or ver_rxbuf_data as child thread.

static int dma_txrx_test(void)	
Parameters	None
Return value	True: The operation is successful False: Receive invalid input or error is found
Description	Run Full duplex test with DMA following described in topic 3.2.4. This function uses gen_txbuf_data and ver_rxbuf_data as child threads.



## 4 Test Software on the target

### 4.1 “tcpdatatest” for half duplex test



```

Command Prompt
D:\Temp>tcpdatatest.exe

[ERROR] The application requires at least 6 input parameters.

*****
TCP Data Transfer Test Version 1.2
*****
tcpdatatest.exe [Mode] [Dir] [ServerIP] [ServerPort] [ByteLen] [Pattern] [Window Scale]

[Mode]      PC Operation mode
             c:Client mode    s:Server mode
[Dir]       Transfer direction of PC
             t:Transmit data  r:Receive data
[ServerIP]  Server IP Address
[ServerPort] Server Port number(0-65535)
[ByteLen]   Transfer length(Byte)
[Pattern]   Disable/Enable Data Pattern in transferring
             0:Disable      1:Enable
[Window Scale] increase window size(optional)
             1:64K    2:128K    3:256K

[Example] tcpdatatest.exe s t 192.168.7.25 4000 34359738368 0

D:\Temp>_

```

Figure 4-1 “tcpdatatest” application usage

“tcpdatatest” is designed to run on PC for sending or receiving TCP data via Ethernet as Server or Client mode. PC of this demo should run in Client mode. User sets parameters to select transfer direction and the mode. Six parameters are required as follows.

- 1) Mode: c – PC runs in Client mode and FPGA runs in Server mode
- 2) Dir: t – transmit mode (PC sends data to FPGA)  
r – receive mode (PC receives data from FPGA)
- 3) ServerIP: IP address of FPGA when PC runs in Client mode
- 4) ServerPort: Port number of FPGA when PC runs in Client mode
- 5) ByteLen: Total transfer size in byte unit. This input is used in transmit mode only and ignored in receive mode. In receive mode, the application is closed when the connection is terminated. In transmit mode, ByteLen must be equal to the total transfer size that is set in test menu of FPGA (receive data test in half-duplex test).
- 6) Pattern:
  - 0 – Generate dummy data in transmit mode or disable data verification in receive mode.
  - 1 – Generate incremental data in transmit mode or enable data verification in receive mode.

Note: Window Scale: Optional parameter which is not used in the demo.

### Transmit data mode

Following sequence is the sequence when test application runs in transmit mode.

- 1) Get parameters from the user and verify that all inputs are valid.
- 2) Create the socket and set socket options.
- 3) Create the new connection by using Server IP address and Server port number.
- 4) Allocate memory to be send buffer.
- 5) Skip this step if the dummy pattern is selected. Otherwise, generate the incremental test pattern to send buffer.
- 6) Send data out and read total sent data from the function.
- 7) Calculate remaining transfer size.
- 8) Print total transfer size every second.
- 9) Repeat step 5) – 8) until the remaining transfer size is 0.
- 10) Calculate total performance and print the result on the console.
- 11) Close the socket and free the memory.

### Receive data mode

Following sequence is the sequence when test application runs in receive mode.

- 1) Follow the step 1) – 3) of Transmit data mode.
- 2) Allocate memory to be receive buffer.
- 3) Read data from the receive buffer and increase total amount of received data.
- 4) This step is skipped if data verification is disabled. Otherwise, received data is verified by the incremental pattern. Error message is printed out when data is not correct.
- 5) Print total amount of received data every second.
- 6) Repeat step 3) – 5) until the connection is closed.
- 7) Calculate total performance and print the result on the console.
- 8) Close socket and free the memory.

## 4.2 “tcp\_client\_trx\_single” for full duplex test



```

Command Prompt
D:\Temp>tcp_client_trx_single.exe

*****
TCP Tx Rx Version 1.0
*****
tcp_client_trx_single.exe [ServerIP] [ServerPort] [ByteLen] [Verification]

[ServerIP]      Server IP Address
[ServerPort]    Server Port number(0-65535)
[ByteLen]       Transfer length(Byte)
[Verification] Disable/Enable Verification in transferring
                0:Disable      1:Enable

[Example] tcp_client_trx_single.exe 192.168.40.42 60000 137438953440 0

D:\Temp>

```

Figure 4-2 “tcp\_client\_trx\_single” application usage

“tcp\_client\_trx\_single” is designed to run on PC for sending and receiving TCP data through Ethernet by using the same port number at the same time. The application is run in Client mode, so user needs to input Server parameters (the network parameters of TOE100G-IP). As shown in Figure 4-2, there are four parameters to run the application, described as follow.

- 1) ServerIP : IP address of FPGA
- 2) ServerPort : Port number of FPGA
- 3) ByteLen : Total transfer size in byte unit. This is total amount of transmitted data and received data. This value must be equal to the transfer size set on FPGA for running full-duplex test.
- 4) Verification:
  - 0 – Generate dummy data for sending function and disable data verification for receiving function. This mode is used to check the best performance of full-duplex transfer.
  - 1 – Generate incremental data for sending function and enable data verification for receiving function.

The sequence of test application is as follows.

- 1) Get parameters from the user and verify that the input is valid.
- 2) Create the socket and set socket options.
- 3) Create the new connection by using Server IP address and Server port number.
- 4) Allocate memory for send and receive buffer.
- 5) Generate incremental test pattern to send buffer when the test pattern is enabled. Skip this step if dummy pattern is selected.
- 6) Send data out, read total sent data from the function, and calculate remaining send size.
- 7) Read data from the receive buffer and increase total amount of received data.
- 8) Skip this step if data verification is disabled. Otherwise, data is verified by incremental pattern. Error message is printed out when data is not correct.
- 9) Print total amount of transmitted data and received data every second.
- 10) Repeat step 5) – 9) until total amount of transmitted data and received data are equal to ByteLen, set by user.
- 11) Calculate performance and print the result on the console.
- 12) Close the socket.

## 5 Revision History

Revision	Date	Description
1.0	16-Dec-22	Initial version release