

SocketXpress with TCP/IP accelerator Reference Design

1	System Overview	2
2	Hardware.....	3
2.1	LAXi2TOE	4
2.1.1	AsyncAxiReg	4
2.1.2	UserRegTOE	5
2.2	IPFilter	6
2.3	PktCombined.....	6
2.4	AXI DMA Controller	7
2.5	LAXi2EMAC.....	8
2.6	TOE10GLL-IP.....	8
2.7	AMD 10G/25G Ethernet Subsystem.....	9
3	Kernel Space	10
3.1	DG 10GEMAC driver	10
3.2	DG TOE driver.....	11
4	User Space.....	12
4.1	SocketXpress	12
4.1.1	Socket Creation and Management.....	13
4.1.2	SocketXpress Information and Control	14
4.1.3	Environment Configuration	15
4.1.4	Connection Management	15
4.1.5	Data Transfer Operations.....	16
4.1.6	File Stream Operations	17
4.2	TCP C testing program.....	18
4.2.1	Compilation and Usage.....	18
4.2.2	Throughput Testing Method	19
4.2.3	Data Verification Testing Method	19
5	Revision History	20

SocketXpress with TCP/IP accelerator Reference Design

Rev1.00 23-Jan-2026

This reference design demonstrates how SocketXpress, a custom Linux network socket C library, offloads TCP processing tasks when communicating over network connections. Working seamlessly with Design Gateway's TOE10GLL IP core (TCP Offload Engine), the system enables applications to achieve significantly higher TCP data transfer speeds while maintaining the same user experience and requiring no application recompilation.

1 System Overview

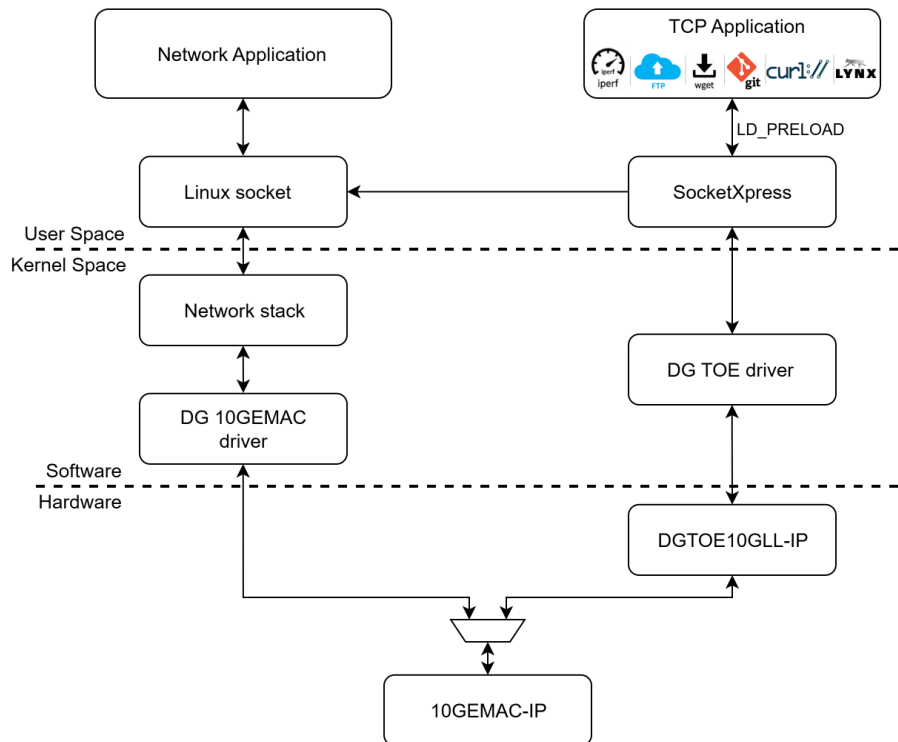


Figure 1 The overview of the SocketXpress with TCP/IP accelerator

Traditional network communication relies on the Linux kernel's network stack to handle TCP protocol processing, including connection management, packet segmentation, acknowledgment handling, and flow control. While this software-based approach provides comprehensive protocol support, it can consume substantial CPU resources during high-throughput data transfers, creating performance bottlenecks on resource-constrained edge devices such as the KR260.

Design Gateway presents a reference design that utilizes high-performance network IP cores, specifically the TOE10GLL-IP and 10GEMAC-IP, implemented in the FPGA hardware logic of the KR260 to fully utilize the 10G Ethernet capabilities on 1 TCP connection. The system provides two communication paths: a hardware TCP offload path using TOE10GLL-IP for maximum single-connection throughput with minimal CPU overhead, and a standard network stack path for multiple connections and comprehensive protocol support.

The SocketXpress library uses LD_PRELOAD mechanism to intercept standard socket API calls, implementing intelligent routing between the hardware acceleration path and standard Linux socket. This approach enables socket-based applications to benefit from hardware acceleration while preserving their original socket-based programming model.

This document is divided into three sections based on system components as shown in Figure 1: Hardware, Kernel space, and User space.

- Hardware: User logic and IP cores for acceleration.
- Kernel space: Device drivers providing interface between hardware accelerators and user space application.
 - a) DG 10GEMAC driver: Ethernet MAC driver with DMA-based packet processing
 - b) DG TOE driver: Manages DMA-based communication with hardware TCP offload engine
- User space: SocketXpress, Custom socket library that interfaces with hardware accelerators.

Each system component's detailed functionality is described in the following sections.

2 Hardware

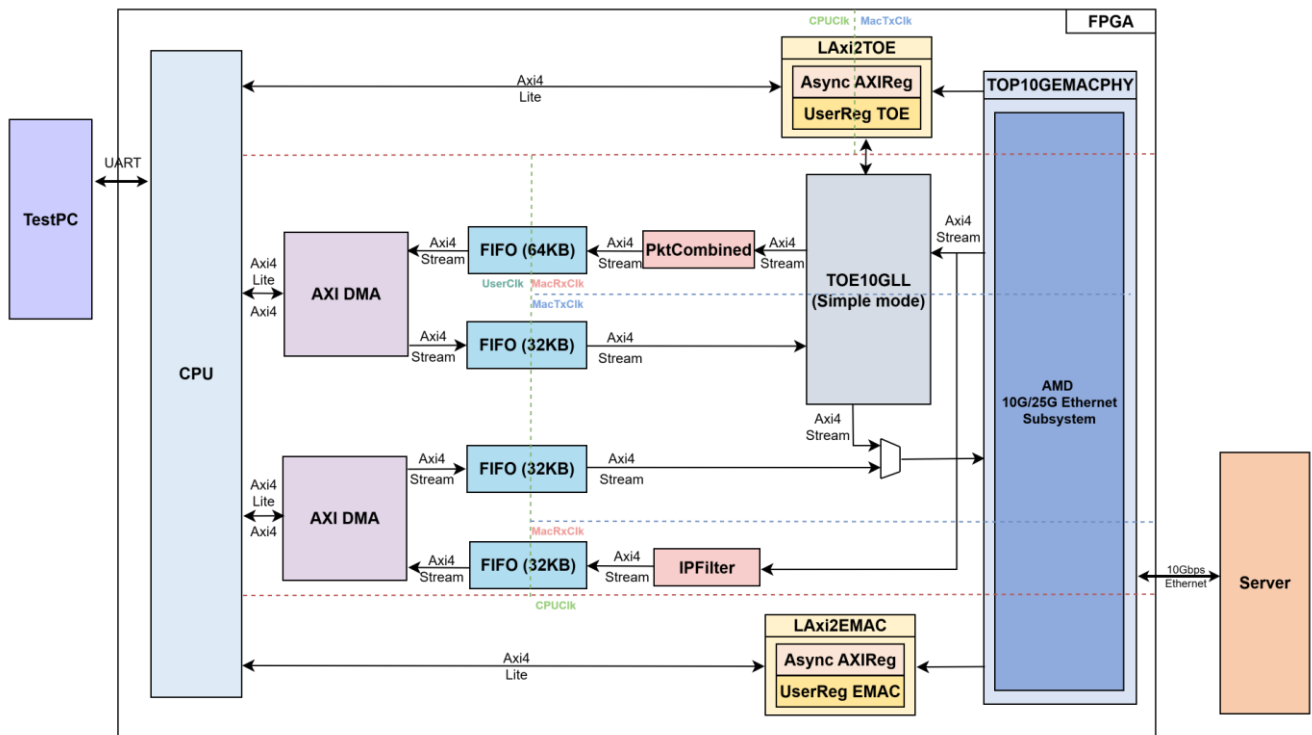


Figure 2 SocketXpress with TCP/IP accelerator reference design block diagram

The hardware is connected to the CPU system via an AXI4-Lite interface for control path and AXI4 interface for data path.

For control path, user-space software interacts with hardware registers via memory mapping. The AXI4-Lite interfaces are implemented using LAXi2EMAC (for 10GEMAC-IP) and LAXi2TOE (for TOE10GLL-IP), as shown in Figure 2.

For Tx data path, packet data is moved from CPU DDR memory by AXI DMA and streamed to FIFO buffers. If it's a TOE data path, the data flows through the TOE10GLL-IP before reaching a 2-to-1 MUX. This MUX gives priority to the TOE path first, then forwards the data to the 10GEMAC-IP for transmission.

For Rx data path, packet data flows from the 10GEMAC-IP to both paths. For the TOE path, data goes through PktCombined, then to FIFO, and finally via DMA to CPU DDR memory. For the normal path, packets are first checked by the IPFilter hardware before proceeding to FIFO and DMA to CPU DDR memory.

The user interface of the TOE10GLL-IP connects to UserRegTOE within the LAXi2TOE module to control and monitor TOE operations through a register map. UserRegTOE interfaces with the CPU through AsyncAXIReg using a register interface, while the CPU connects to AsyncAXIReg via an AXI4-Lite interface.

For the DMA Controllers, they are controlled via AXI4-Lite interfaces and configured with 128-bit memory map and stream data widths, supporting scatter-gather operations with unaligned transfers. The DMA control is implemented through two separate Linux platform drivers.

The TOE10GLL-IP operates in Simple mode and connects to the AMD 10G/25G Ethernet Subsystem through a 32-bit AXI4-Stream interface. An IPFilter module is positioned between the TOE and Ethernet subsystem to filter duplicate TCP connection packets. Additionally, a PktCombined module combines payload packets from the TOE10GLL-IP.

This design includes four clock domains:

- CPUClk: Used for CPU communication via the AXI4-Lite bus.
- UserClk: Used as frequency clock domain for AXI DMA data path.
- MacTxClk: Synchronized with the Tx EMAC interface and the Tx user data interface.
- MacRxClk: Synchronized with the Rx EMAC interface and the Rx user data interface.

Details of each module are provided below.

2.1 LAXi2TOE

The LAXi2Reg module consists of AsyncAxiReg and UserRegTOE. AsyncAxiReg converts AXI4-Lite signals into a simple Register interface with a 32-bit data bus size, similar to AXI4-Lite standard. Additionally, it includes asynchronous logic to handle clock domain crossing between CPUClk and UserClk domains.

2.1.1 AsyncAxiReg

This module is designed to convert the signal interface of AXI4-Lite to be register interface. Also, it enables two clock domains to communicate.

The simple register interface is designed to be compatible with a single-port RAM interface for write transaction. For read transaction, the Register interface is slightly modified from the RAM interface by adding RdReq and RdValid signals to control read latency. Since the address of the Register interface is shared for both write and read transactions, the user cannot perform simultaneous write and read operations. The timing diagram for the Register interface is shown in Figure 3.

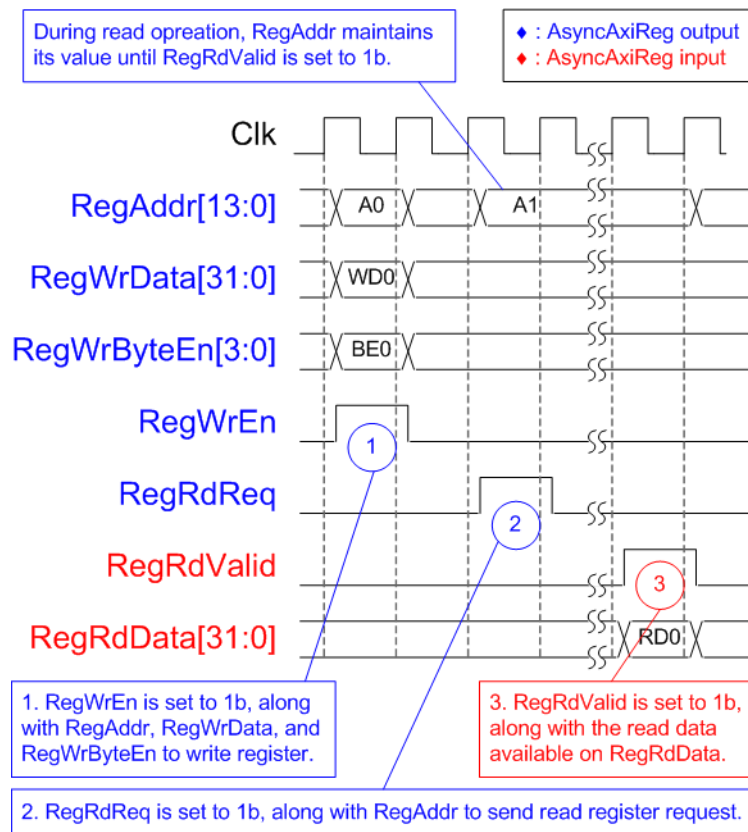


Figure 3 Register Interface Timing Diagram

- 1) To write register, the timing diagram is similar to that of a single-port RAM. The RegWrEn signal is set to 1b, along with a valid RegAddr (Register address in 32-bit units), RegWrData (write data for the register), and RegWrByteEn (write byte enable). The byte enable is four bits wide, where each bit indicates the validity of a specific byte within RegWrData. For example, if RegWrByteEn[0], [1], [2], and [3] are set to 1b, then RegWrData[7:0], [15:8], [23:16], and [31:24] are valid, respectively.
- 2) To read from a register, AsyncAxiReg sets the RegRdReq signal to 1b, along with a valid value for RegAddr. After the read request is processed, the 32-bit data is returned. The slave detects the RegRdReq being asserted to start the read transaction. During the read operation, the address value (RegAddr) remains unchanged until RegRdValid is set to 1b. Once valid, the address is used to select the returned data through multiple layers of multiplexers.

2.1.2 UserRegTOE

For register file, UserReg is designed to write/read registers corresponding with write register access or read register request from AsyncAxiReg module. The memory map inside UserReg module is shown in Table 1.

Table 1 Register map Definition of UserRegTOE

Address offset	Register Name	Description
TOE10GLL-IP register		
0x00000	TOE_RST_INTREG	Wr[0]: Mapped to RstB of TOE10GLL-IP
0x00004	TOE_OPM_INTREG	Wr[16]: Mapped to ARPICMPEn of TOE10GLL-IP Wr[1:0]: Mapped to DstMacMode of TOE10GLL-IP
0x00008	TOE_SML_INTREG	Wr[31:0]: Mapped to SrcMacAddr[31:0] of TOE10GLL-IP
0x0000C	TOE_SMH_INTREG	Wr[15:0]: Mapped to SrcMacAddr[47:32] of TOE10GLL-IP
0x00010	TOE_DMIL_INTREG	Wr[31:0]: Mapped to DstMacAddr[31:0] of TOE10GLL-IP
0x00014	TOE_DMIH_INTREG	Wr[15:0]: Mapped to DstMacAddr[47:32] of TOE10GLL-IP
0x00018	TOE_SIP_INTREG	Wr[31:0]: Mapped to SrcIPAddr of TOE10GLL-IP
0x0001C	TOE_DIP_INTREG	Wr[31:0]: Mapped to DstIPAddr of TOE10GLL-IP
0x00020	TOE_TMO_INTREG	Wr[31:0]: Mapped to TimeOutSet of TOE10GLL-IP
0x00024	TOE_TIC_INTREG	Wr[0]: Set '1' to clear read value of TOE_STS_INTREG[2]
0x00030	TOE_CMD_INTREG	Wr[1:0]: Mapped to TCPCmd of TOE10GLL-IP.
0x00034	TOE_SPN_INTREG	Wr[15:0]: Mapped to TCPSrcPort[15:0] of TOE10GLL-IP
0x00038	TOE_DPN_INTREG	Wr[15:0]: Mapped to TCPDstPort[15:0] of TOE10GLL-IP
0x00040	TOE_VER_INTREG	Rd[31:0]: Mapped to IP version of TOE10GLL-IP
0x00044	TOE_STS_INTREG	Rd[20:16]: Mapped to IPState of TOE10GLL-IP Rd[2]: TOE10GLL-IP Interrupt. Asserted to '1' when IPInt is asserted to '1'. This flag is cleared by TOE_TIC_INTREG. Rd[1]: Mapped to TCPConnOn of TOE10GLL-IP Rd[0]: Mapped to InitFinish of TOE10GLL-IP
0x00048	TOE_INT_INTREG	Rd[31:0]: Mapped to IntStatus of TOE10GLL-IP
0x0004C	TOE_DMOL_INTREG	Rd[31:0]: Mapped to DstMacAddrOut[31:0]
0x00050	TOE_DMOH_INTREG	Rd[15:0]: Mapped to DstMacAddrOut[47:32]
IPFilter register		
0x00054	FILTER_ENABLE	Wr[0]: Mapped to FilterEn of IPFilter module
Ethernet MAC register		
0x00058	EMAC_LINKSTATUS	Rd[0]: Mapped to Link status of Ethernet MAC
AXI4 Stream data FIFO register		
0x00060	DMA_TXFIFO_FLUSH	Wr[0]: set '1' to force AXIS valid signal to '1'
0x00064	DMA_TXFIFO_RDCNT	Rd[31:0]: Mapped to axis_rd_data_count of FIFO
0x00068	DMA_RXFIFO_WRCNT	Rd[31:0]: Mapped to axis_wr_data_count of FIFO

2.2 IPFilter

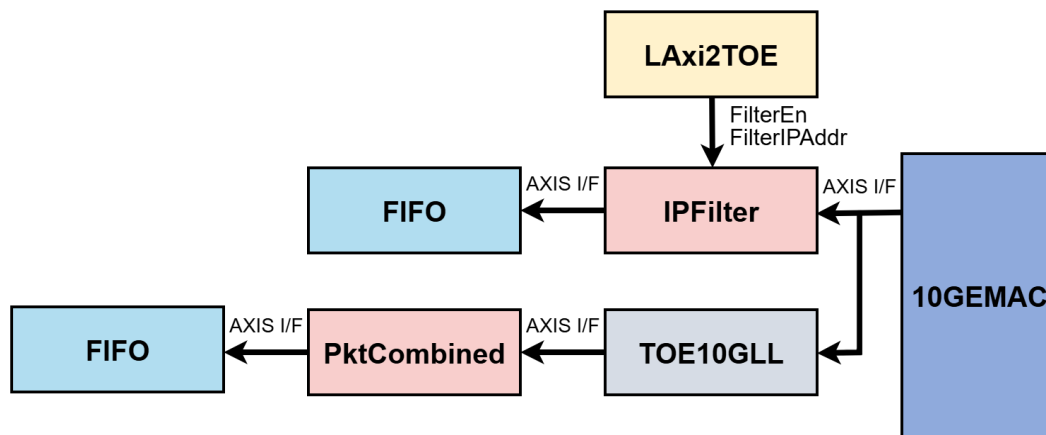


Figure 4 Rx path block diagram

The IPFilter module shown in Figure 4 filters duplicate TCP connection packets from the 10GEMAC-IP before they reach the network stack. It is controlled by two signals: FilterEn and FilterIPAddr. When FilterEn is set to '1' (enabled) and if the destination IP address matches the value stored in FilterIPAddr, the corresponding packet is dropped.

2.3 PktCombined

The PktCombined module illustrated in Figure 4 combines payload packets received from the TOE10GLL-IP to reduce CPU copying operations and lower CPU load. The maximum number of packets to combine is configurable through the generic parameter MaxNumCombinedPkt. PktCombined operates continuously and attempts to combine packets whenever possible by holding the AXI last signal low. The AXI last signal is only asserted to '1' under two conditions: when no incoming packets are available, or when the number of combined packets reaches the MaxNumCombinedPkt limit. Additionally, error packets from the 10GEMAC-IP are dropped within this module.

2.4 AXI DMA Controller

AXI DMA Controller can be generated by using Vivado IP catalog. The user uses the following settings as shown in Figure 5.

- Enable Scatter Gather Engine : Enable
- Width of Buffer Length Register : 16
- Address Width : 32
- Enable Read Channel : Enable
- Enable Write Channel : Enable

Read Channel

- Number of Channel : 1
- Memory Map Data Width : 128
- Stream Data Width : 128
- Max Burst Size : 256
- Allow Unaligned Transfers : Enable

Write Channel

- Number of Channel : 1
- Memory Map Data Width : 128
- Stream Data Width : 128
- Max Burst Size : 256
- Allow Unaligned Transfers : Enable

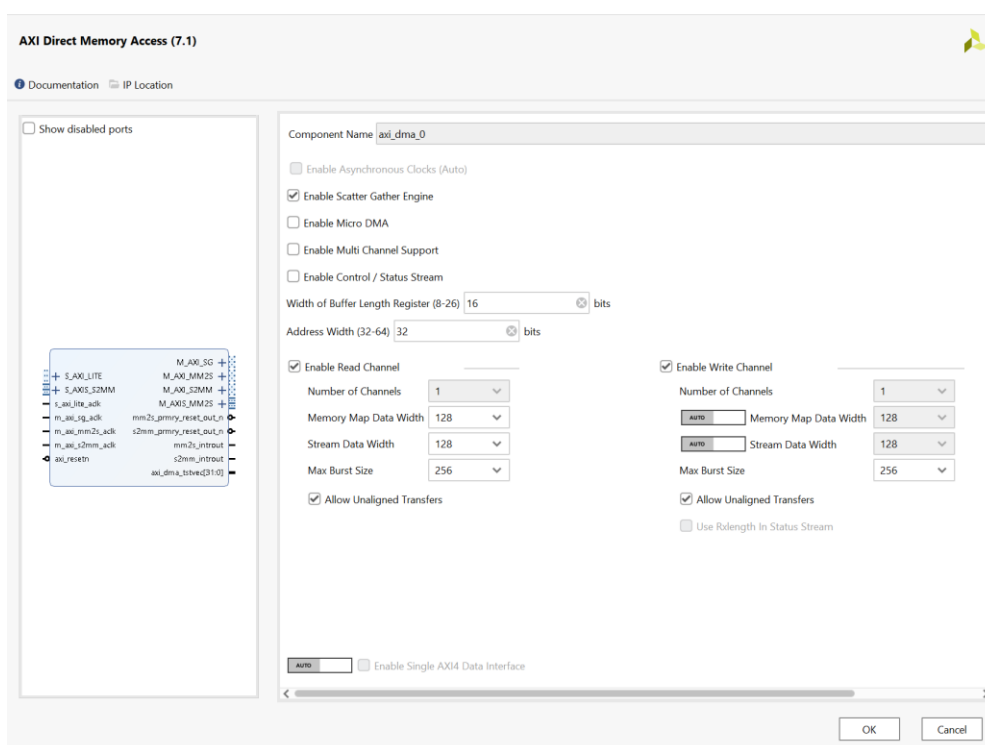


Figure 5 Example AXI DMA configuration page

The example of the AXI DMA Controller in the Ultrascale model is described in the following link:

https://www.xilinx.com/products/intellectual-property/axi_dma.html

2.5 LAXi2EMAC

The LAXi2EMAC module is connected to CPU through AXI4-Lite bus. LAXi2 EMAC consists of AsyncAxiReg and UserRegEMAC. UserRegEMAC is designed to read registers status of AMD 10G/25G Ethernet Subsystem and generate/clear interrupt signal corresponding with write register access or read register request from AsyncAxiReg module. Memory map inside UserRegEMAC module is shown in Table 2.

Interrupt link status is asserted to '1' when detect a link up status changed from AMD 10G/25G Ethernet Subsystem.

Table 2 Register map Definition of UserRegEMAC

Address offset	Register Name	Description
0x00000	EMAC_LINKSTATUS	Rd[0]: Mapped to Link status of Ethernet MAC
0x00004	EMAC_IPVERSION	Rd[31:0]: Mapped to IP version of Ethernet MAC
0x00008	EMAC_CLEAR_IRQ	Wr[0]: Set '1' to clear Interrupt link status

2.6 TOE10GLL-IP

TOE10GLL-IP implements the TCP/IP stack and offload engine for the low latency solution. User interface has two signal groups, i.e., control signals and data signals. The IP can be configured to run in two modes, i.e., Cut-through mode for low-latency application and Simple mode for simple user interface. This reference design shows the usage in Simple mode. More details are described in datasheet.

https://dgway.com/products/IP/Lowlatency-IP/dg_toe10gllip_data_sheet_xilinx_en/

2.7 AMD 10G/25G Ethernet Subsystem

Ethernet Subsystem can be generated by using Vivado IP catalog. The user uses the following settings as shown in Figure 6.

- Select Core : Ethernet MAC+PCS/PMA 32-bit
- Speed : 10.3125G
- Data Path Interface : AXI Stream
- Num of Cores : 1

Read Channel

- Auto Negotiation Logic : None

Read Channel

- Control and Statistic Interface : Control and Status Vectors

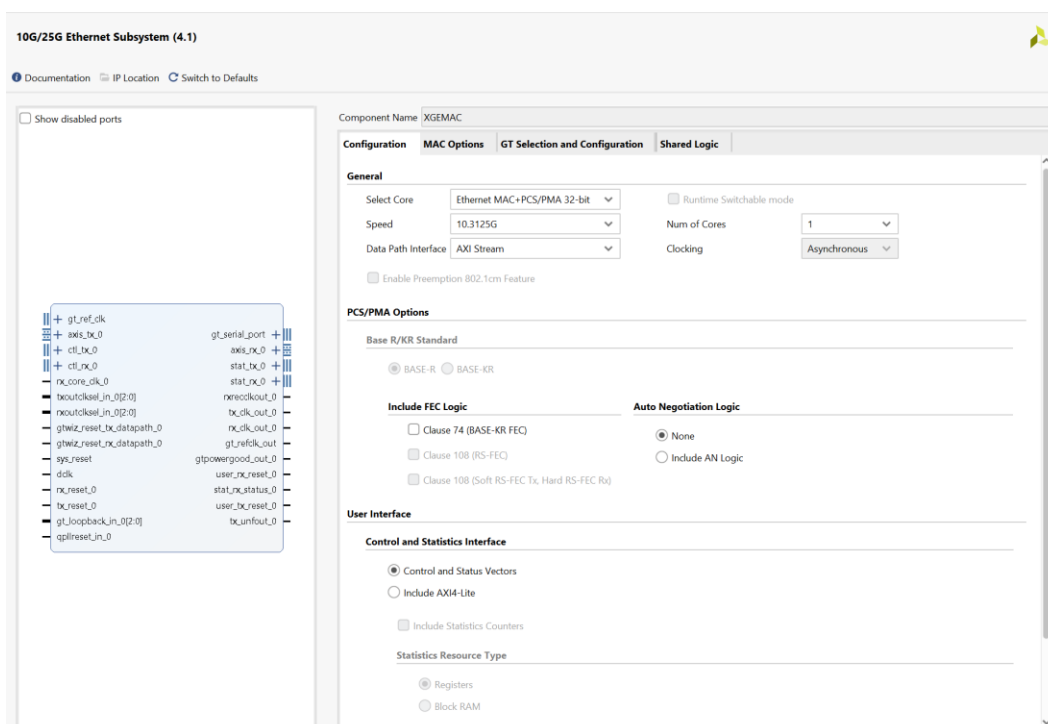


Figure 6 Example of AMD 10G/25G Ethernet Subsystem configuration page

The example of Transceiver wizard in Ultrascale model is described in the following link:

<https://www.amd.com/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-25gemac.html>

3 Kernel Space

This reference design uses the 5.15.0-1027-xilinx-zynqmp kernel image, based on Ubuntu Desktop 22.04 LTS. To facilitate communication between hardware, device driver and user-space software as shown in Figure 7

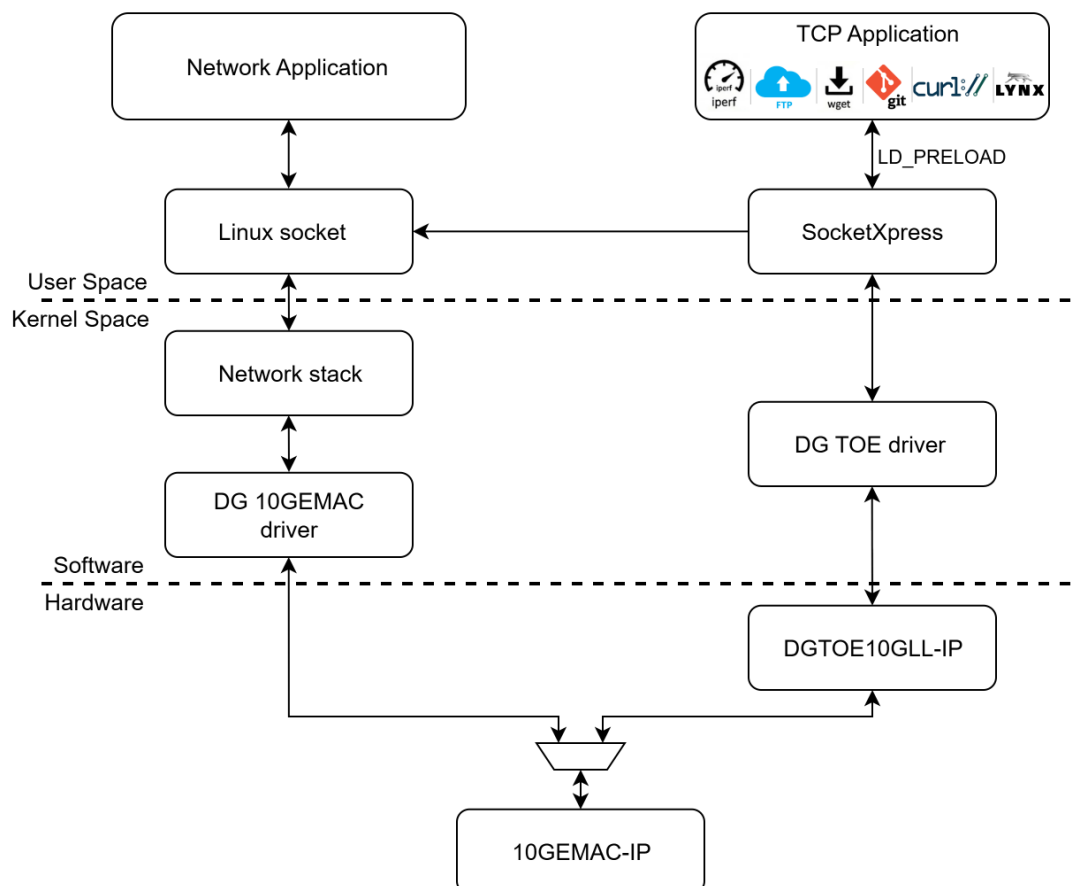


Figure 7 The overview of the SocketXpress with TCP/IP accelerator

The kernel space component consists of two drivers: the DG 10GEMAC driver enables the Linux network stack to communicate with the 10GEMAC-IP, while the DG TOE driver provides a direct interface for user space applications to communicate with the TOE10GLL-IP.

3.1 DG 10GEMAC driver

The DG 10GEMAC driver is modified from the Xilinx driver with unused functions removed and enhanced link status detection capability added. The driver is a Linux network device driver implemented as a platform driver that integrates with the Linux device tree framework for automatic hardware discovery and resource allocation. Upon hardware load, the driver registers with the platform bus and uses device tree matching to detect compatible hardware through the compatible string. The driver provides support for two Ethernet MAC hardware: LL10GEMAC-IP (a low-latency EMAC IP core developed by Design Gateway) and AMD 10G/25G Ethernet Subsystem.

The driver implements a complete network interface that integrates with the Linux network stack. The architecture employs interrupt-driven operation with separate IRQ handling for TX completion, RX packet received, and link status changes, ensuring efficient resource utilization and responsive network performance. The driver supports configurable MTU sizes up to 9000 bytes for jumbo Ethernet frames. Link status monitoring is implemented through IRQ-based detection and reporting, providing real-time network connectivity feedback.

Buffer Descriptor Management and Scatter Gather DMA

The driver employs Scatter Gather (SG) DMA architecture through circular rings of buffer descriptors (BDs) that enable efficient handling of non-contiguous memory buffers. Each BD contains control information, status flags, control flags, and buffer addresses with a crucial "next" pointer that chains descriptors together to form the scatter gather list. The SG DMA capability allows the hardware to automatically process multiple buffer descriptors in sequence without CPU intervention.

Network Stack Integration

The driver integrates seamlessly with the Linux network stack through socket buffer (SKB) management.

For reception, it pre-allocates SKBs using `netdev_alloc_skb()` and maps their data buffers to DMA-accessible addresses stored in RX descriptors. Upon packet arrival, the driver unmaps the DMA buffer, sets the SKB length with `skb_put()`, determines the protocol using `eth_type_trans()`, and delivers the packet to the network stack via `netif_receive_skb()`.

For transmission, the driver handles both linear and fragmented SKBs efficiently. Linear packet data is mapped directly, while fragmented packets use `skb_frag_dma_map()` to map each fragment individually across multiple buffer descriptors. The driver stores SKB pointers within buffer descriptors to enable proper cleanup during TX completion, using `dev_kfree_skb_irq()` in interrupt context to free transmitted packets.

For more details, please refer to:

- Official GitHub Repository: <https://git.launchpad.net/~canonical-kernel/ubuntu/+source/linux-xilinx-zynqmp/+git/jammy/tree/drivers/net/ethernet/xilinx?h=master-next>

3.2 DG TOE driver

The DG TOE driver is a Linux platform driver that enables access to TCP offload functionality by implementing a character device interface (`/dev/dg_stack`) that serves as an interface layer enabling SocketXpress library integration with TOE10GLL-IP core, replacing traditional kernel network stack operations. The driver registers with the platform bus using device tree matching through compatible string.

Buffer Descriptor Management and Zero-Copy Architecture

While employing the same Scatter Gather DMA architecture as the EMAC driver with circular buffer descriptor rings, the TOE driver implements a fundamentally different memory management strategy optimized for zero-copy operations. Instead of allocating individual SKBs for each descriptor, the driver allocates large memory regions for both TX and RX operations. These buffers are then subdivided across multiple buffer descriptors, with each BD pointing to its designated segment within the larger memory block. This approach enables direct user-space access to DMA buffers through memory mapping, eliminating costly data copying between kernel and user space.

Character Device Interface and IOCTL Commands

Rather than integrating with the Linux network stack, the driver provides a user-kernel interface through a character device.

File Operations:

- `poll()` : Monitoring device readiness. Returns `POLLIN` when received data is available and `POLLOUT` when transmit buffer space is available. Return ready state when connection is closed.
- `mmap()` : Memory mapping for zero-copy data access. Exposes TX and RX DMA buffers directly to user space with cached memory access, allowing applications to read and write data without kernel buffer copies.
- `write()` : Buffers data written to the character device. Copies data from user space to an internal kernel buffer for later transmission.

ioctl commands:

- `DG_SEND` : Update user Tx BD buffer pointer to kernel space and update DMA BD pointer to start DMA transmit.
- `DG_RECV` : Retrieve received packet length information in array in size of number of BD
- `DG_UpdateUserRxPtr` : Update user Rx BD buffer pointer to kernel space
- `DG_GetUserRxPtr` : Obtain current Rx BD buffer pointer from kernel space
- `DG_GetDMATxPtr` : Obtain current Tx BD buffer pointer from kernel space
- `DG_IRead/DG_IWrite` : Direct access to hardware registers
- `DG_GET_MAC_ADDR` : Retrieve the device MAC address
- `DG_FLUSH_TX/RX` : Force DMA data buffer flushing and cleanup data in DMA
- `DG_FLUSH_WRBUFFER` : Flush buffered data from `write()` system calls and return it to userspace.

4 User Space

User space includes the SocketXpress library and the TCP testing C program. SocketXpress is a custom Linux network socket C library which works with Design Gateway's TOE10GLL IP core that offloads TCP tasks from CPU, while TCP testing C program is a socket-based C application designed for throughput measurement and data integrity verification. This program serves as a practical demonstration of how existing applications can seamlessly switch from standard Linux socket to SocketXpress library without requiring source code modifications, showcasing the performance benefits of hardware-accelerated TCP processing.

4.1 SocketXpress

The SocketXpress library is a custom Linux network socket C library that provides hardware-accelerated TCP processing through Design Gateway's TOE10GLL IP core. The library uses LD_PRELOAD to intercept standard POSIX socket API calls and standard C library functions, transparently redirecting IPv4 TCP operations to the TOE hardware while maintaining most API compatibility with existing applications. When the TOE device is already in use or unavailable, the library automatically falls back to the original Linux socket implementation. Functions not intercepted by the library, or intercepted functions not called with a SocketXpress file descriptor, continue to operate using the original Linux implementation. The library has been tested with various applications as shown in Table 3

Table 3 Applications Tested with SocketXpress Library

Application Name	Version	Test Scenario
lperf	2.1.5 3.9	Established connection to lperf server and performed bandwidth performance tests
lynx	2.9.0	Browsed web content and navigated to external sites
curl	7.81.0	Retrieved web pages from public websites
Telnet	0.17-44	Connected to remote server and executed basic terminal commands
wget	1.21.2	Downloaded web pages and files from remote servers
ssh	8.9p1	Established secure connection to remote server and executed basic commands
scp	8.9p1	SCP file upload and download operations
ftp	20210827-4	FTP file upload and download operations
git	2.34.1	Cloned remote repository and checked out branches
mysql	8.0.43	Connected to MySQL database server and queried data
links	2.25	Browsed web content and navigated to external sites

Note: The applications listed have been tested for basic usage scenarios. If you encounter any bugs or require support for additional applications, please contact us.

4.1.1 Socket Creation and Management

- **Socket**

```
int socket(int domain, int type, int protocol);
```

The socket creation function implements intelligent routing between DG hardware acceleration and standard Linux socket. When an application requests an IPv4 TCP socket (AF_INET + SOCK_STREAM) and the TOE device is not already in use, the function creates a custom socket by opening character device /dev/dg_stack. For all other socket types or when the TOE device is already in use, the function creates a standard Linux socket using the original socket() system call.

- **Bind**

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The bind operation allows applications to specify source IP address and port for SocketXpress connections. For SocketXpress, the function extracts IPv4 address and port information from the sockaddr_in structure and stores them in variables. These values override the default source IP and port configured through environment variables during subsequent connect or accept operations.

- **Close**

```
int close(int fd);
```

The close operation handles proper cleanup of SocketXpress connections. For SocketXpress, it soft reset the TOE10GLL-IP and flush remaining data in dma buffer, then proceeds to close the actual file descriptor using the original close() system call. This ensures both software state and hardware state are properly reset

4.1.2 SocketXpress Information and Control

- **getpeername**

```
int getpeername(int socket, struct sockaddr *address, socklen_t *address_len);
```

Returns remote peer address information for established SocketXpress connections. The function populates a `sockaddr_in` structure with the target IP address and port stored during connection establishment. Note that in server mode, the TOE10GLL-IP hardware does not provide the target port information, so it remains 0.

- **getsockname**

```
int getsockname(int socket, struct sockaddr *address, socklen_t *address_len);
```

Returns local socket address information for SocketXpress connections. The function populates a `sockaddr_in` structure with the source IP address and port used for the connection.

- **fcntl**

```
int fcntl(int fd, int cmd, ...);
```

The `fcntl` function handles file control operations for SocketXpress, supporting only non-blocking mode configuration through `F_SETFL` with `O_NONBLOCK` flag and status retrieval through `F_GETFL`.

- **setsockopt**

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

The `setsockopt` function provides compatibility with standard socket options for SocketXpress. Only `TCP_NODELAY` option is actually supported and will affect socket behavior.

All other options including `SO_SNDBUF`, `SO_RCVBUF`, `SO_TIMESTAMP`, `SO_REUSEADDR`, and `TCP_MAXSEG` are stored in the `socket_opts` structure but have no functional effect.

- **getsockopt**

```
int getsockopt(int sockfd, int level, int option_name, void *option_value, socklen_t *option_len);
```

The `getsockopt` function retrieves stored socket option values from the `socket_opts` structure. It returns previously set values for all supported options, plus `SO_ERROR` (always 0) and `SO_TYPE` (always `SOCK_STREAM`).

Socket options initialize with:

- `tcp_nodelay` = 0
- `tcp_maxseg` = 1460
- `so_sndbuf` = 65536
- `so_rcvbuf` = 65536
- `so_timestamp` = 0
- `so_reuseaddr` = 0

4.1.3 Environment Configuration

The SocketXpress library supports seamless integration with existing applications through LD_PRELOAD, allowing standard Linux socket applications to use hardware-accelerated TCP processing without recompiling. Applications such as lynx, curl, and custom socket programs can be easily switched from standard Linux socket to the SocketXpress library.

The SocketXpress library replaces standard Linux socket functions using LD_PRELOAD mechanism:

```
> [Environment variables] LD_PRELOAD=libSocketXpress.so <Application>
```

The library supports configuration through environment variables that can be specified as additional parameters in the command line:

- SOURCE_IP=<TOE IP>[/subnet_mask]: Specify source IP address for TOE10GLL operations with optional subnet mask. Used for applications that do not explicitly specify an IP address through bind() operations. Supports CIDR notation (e.g., 192.168.11.11/24). If no subnet mask is specified, defaults to /24
- TARGET_IP=<Host IP>: Specify target IP address when using FPGA as a server mode. This sets the expected client IP address for incoming connections
- SOURCE_PORT=<TOE Port>: Specify source port for TOE10GLL operations (default: 60000). For applications that do not specify a port through bind() operations, this port will be used and automatically incremented for each new connection
- GATEWAY_IP=<Gateway IP>: Specify the gateway IP address when the communication requires routing through a specific gateway. Default: Using Gateway IP from Linux ARP table.
- TX_COMBINE=<1|true>: Control TX packet combining behavior. By default, TX combining is enabled to merge data before transmission for efficiency. Setting this to "0" or "false" disables combining, causing data to be sent immediately without merging or waiting. This provides lower latency but may result in lower overall throughput due to sending smaller packets.
- STS_LOG=<0|false>: Enable internal status logging. When set to "1" or "true", the library prints operational messages to terminal. Default is disable.

These environment variables provide default values that can be overridden by explicit bind() operations or connection parameters.

4.1.4 Connection Management

- **connect**

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The connect operation establishes TCP connections using TOE hardware acceleration. The implementation prevents mode conflicts by checking that the SocketXpress is not already in server mode and handles the complete TOE10GLL-IP initialization phase.

Connection Process:

1. Extracts target IP and port from sockaddr_in structure
2. Configures TOE hardware registers for active open mode
3. Compares source and target IP addresses to determine if they are in the same subnet. If not, the system initiates TOE to send an ARP request to the gateway to obtain the gateway MAC address, then sets TOE to fixed MAC mode using the gateway MAC address
4. Initiates TOE hardware for TCP connection establishment and enables IPFilter hardware
5. Waits for connection completion with signal interrupt support
6. Updates connection state flags and returns 0 on successful completion

Error returns:

- EBUSY : already in server mode
- ENETUNREACH : link is down
- ETIMEDOUT : connection fails
- EINTR : interrupted by signal

- **accept**

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The accept operation implements server-side connection acceptance using TOE hardware. The implementation prevents mode conflicts by checking that the SocketXpress is not already in client mode and handles the complete TOE10GLL-IP initialization for passive connection establishment.

Connection Process:

1. Configures TOE hardware registers for passive open mode with ARP/ICMP response
2. Initiates TOE hardware for TCP connection establishment and enables IPFilter hardware
3. Waits for incoming TCP connection with signal interrupt support
4. Populates client address information in provided sockaddr structure upon successful connection
5. Creates and returns dummy file descriptor (server_fd) for API compatibility

Client Information: When successful, populates the addr structure with the connecting client's IP address. Port information is not available from the TOE hardware and cannot be retrieved (will be set to 0).

Error returns:

- EBUSY : already in server mode
- ENETUNREACH : link is down
- EINTR : interrupted by signal

4.1.5 Data Transfer Operations

- **send and write**

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t write(int fd, const void *buf, size_t count);
```

The send operations implement data transmission through TOE hardware with support for both blocking and non-blocking modes. Since the SocketXpress implementation does not support send flags, both send() and write() functions operate identically, ignoring any flags parameter passed to send().

When a user call send(), it attempts to merge data before copying to the DMA buffer for efficiency. However, when TCP_NODELAY is set through setsockopt(), it copies data immediately without merging. This provides lower latency but may result in lower overall throughput due to sending smaller packets. By default, it operates in blocking mode and continuously retries until all data is transmitted, similar to standard Linux send() behavior. When non-blocking mode is set, it calls the underlying send function once and returns immediately with EAGAIN error, potentially with partial data sent.

- **recv and read**

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t read(int fd, void *buf, size_t count);
```

The receive operations implement data reception from TOE hardware with support for both blocking and non-blocking modes. read() is equivalent to recv() with flags set to 0.

Supported flags:

- MSG_PEEK: Peek at data without removing it from the buffer

When a user calls recv(), it copies data from the DMA receive buffer to the user buffer up to the specified length. By default, it operates in blocking mode and does not return until data becomes available or the connection is closed. When non-blocking mode is set, it returns immediately with EAGAIN error if no data is currently available.

- **sendmsg**

```
*ssize_t sendmsg(int sockfd, const struct msghdr msg, int flags);
```

The sendmsg() operation works similarly to send() but handling multiple data buffers. It processes the msghdr structure by combining all iovec buffers into a single contiguous buffer, then internally calls send() to transmit the data. This allows applications to send data from multiple memory locations in a single system call. It does not support any flags parameter.

- **recvmsg**

```
*ssize_t recvmsg(int sockfd, struct msghdr msg, int flags);
```

The recvmsg() operation works similarly to recv() but handles multiple data buffers. It processes the msghdr structure by allocating a temporary buffer, internally calls recv() to receive the data, then distributes the received bytes sequentially across all iovec buffers. This allows applications to receive data into multiple memory locations in a single system call. It does not support any flags parameter.

4.1.6 File Stream Operations

- **fflush**

```
int fflush(FILE *stream);
```

The fflush() flushes buffered data from a FILE stream and using send() to transfer data through the TOE hardware.

- **getc and fgetc**

```
int fgetc(FILE *stream);
int getc(FILE *stream);
```

The fgetc() and getc() read a single character from receive buffer. getc() operates identically to fgetc(). They internally use recv() to retrieve one byte from the TOE hardware, returning EOF when no data is available.

- **fgets**

```
char *fgets(char *s, int size, FILE *stream);
```

The fgets() reads character-by-character using recv() until a newline is encountered, the size limit is reached, or EOF occurs. The string is null-terminated and includes the newline character if present.

4.2 TCP C testing program

The TCP C testing program is a cross-platform throughput measurement utility designed to demonstrate the performance benefits of the SocketXpress library that works with TOE10GLL-IP compared to standard Linux socket with network stack. This program serves as a practical example of how existing applications can be switched from standard socket implementations to hardware-accelerated TCP processing using TOE10GLL-IP without requiring source code modifications.

The program supports both client and server modes with transmit (TX) and receive (RX) operations. It provides two main testing methodologies: throughput testing for performance measurement and data verification testing for integrity validation.

4.2.1 Compilation and Usage

Compilation

The program can be compiled on both platforms without additional dependencies:

Linux:

```
> gcc -o TCP TCP.c
```

Windows:

```
> gcc -o TCP.exe TCP.c -lws32_
```

Usage

The program demonstrates seamless integration with the SocketXpress library using LD_PRELOAD, allowing switching from standard Linux socket to hardware-accelerated TCP processing without recompiling:

Standard Socket Operation:

```
> ./TCP -c|-s -tx|-rx [options]
```

SocketXpress Library Operation:

```
> LD_PRELOAD=libSocketXpress.so ./TCP -c|-s -tx|-rx [options]
```

Command Line Arguments

The program requires mode and operation selection:

Required Arguments:

- c : Client mode (initiates connections)
- s : Server mode (listens for connections)
- tx : Transmit test (send data)
- rx : Receive test (receive data)

Key Optional Arguments:

- b <IP> : Bind source IP for client mode (default: not bind)
- bp <Port> : Bind source Port for client mode (default: not bind)
- p <port> : Port number (default: 60000)
- i <IP> : Target IP for client mode (default: 127.0.0.1)
- buf <size> : Buffer size in MB (default: 1, max: 1024)
- cs <size> : Send chunk size in bytes (default: 16384, max: 1073741824)
- sb <size> : Socket buffer size in KB (default: 1024, max: 1048576)
- nodelay : Enable TCP_NODELAY

-v : Enable verification (default buffer size will be set to 1GB)
 For TX: sends 32-bit incremental pattern
 For RX: stops after buffer is full and verifies data

4.2.2 Throughput Testing Method

Socket Optimization Configuration

The program implements socket optimization through the `configure_socket_options()` function to enhance throughput performance. When the `-nodelay` flag is specified, the program enables `TCP_NODELAY`.

The program configures socket buffer sizes using `SO_RCVBUF` and `SO_SNDBUF` options, setting both send and receive buffers to the size specified by the `-sb` parameter (default 1MB). Proper socket buffer sizing is critical for achieving optimal throughput, particularly on high-bandwidth networks where insufficient buffering can create bottlenecks.

Transmission Operation

In transmit mode, the program uses the `do_transmit()` function to send data from a pre-allocated buffer in configurable chunk sizes. The transmission continues until manually stopped with `Ctrl+C`, using the same buffer content repeatedly for maximum efficiency while tracking total bytes sent. The transmission process sends data in chunks specified by the `-cs` parameter (default 16KB).

Reception Operation

In receive mode, the program uses the `do_receive()` function to receive data in fixed 1MB chunks for optimal buffer management. The program accumulates total bytes received without storing all data, continuing reception until manually stopped to measure raw network reception performance.

4.2.3 Data Verification Testing Method

The data verification method validates data integrity during transmission, ensuring that TCP offloading maintains complete data accuracy. This mode is enabled with the `-v` flag and automatically sets the buffer size to 1GB for comprehensive testing. For proper data verification testing, both the server and client must be run with the `-v` flag to ensure verification behavior across the entire data path.

Transmission Pattern Generator

On the transmit side, when verification mode is enabled with the `-v` flag, the `fill_buffer_incremental()` function fills the buffer with sequential 32-bit integers. This creates a pattern that can be verified on the receiving end, using the entire buffer space for pattern generation. The function reports the range of values generated from 0 to the maximum count.

The TX side then sends this incremental pattern using configured chunk sizes while maintaining buffer offset tracking to ensure pattern continuity.

Receive Data Verification

On the receive side, when verification mode is enabled with the `-v` flag, the program receives data until the buffer reaches full capacity (1GB by default) and automatically stops reception when buffer is full. The program then calls the `verify_data()` function to validate the received content by comparing it against the expected incremental pattern that was generated on the TX side.

The verification process provides comprehensive integrity analysis by checking every 32-bit value against its expected sequential position. The process reports the total number of errors found during verification and displays the first 10 errors with their positions and values for debugging purposes. The verification concludes with a clear SUCCESS or FAILED status based on data integrity results.

This verification method ensures that hardware-accelerated TCP processing through TOE10GLL-IP maintains complete data integrity while achieving higher performance compared to software-based TCP stacks.

5 Revision History

Revision	Date (D-M-Y)	Description
1.00	23-Jan-26	Initial version release