Introduction

SRIO is one of the most high-speed connections available between two devices on the market today. At top speed, you can obtain ~25Gbps bandwidth – not bad for 4 ports transmitting and receiving differential data. The basic idea of how this peripheral works is not difficult to understand. First, we designed this peripheral to meet or exceed the Rapid I/O spec. Second, there are two ways to use SRIO: (1) direct I/O which is similar to doing a DMA transfer from a source on one device to a destination on another device; (2) Message passing. The concepts are relatively easy to understand, but sometimes is almost too abstract to get your hands around. Most users prefer direct I/O due to speed, but we’ll cover both methods in this chapter.

Objectives

- Provide an introduction to what SRIO is and its basic terminology
- Explain an example using Direct I/O and using CSL to program the transfer.
- Discuss what Message Passing is and do a few examples. Then, compare/contrast the two methods.
- Lab: build, load and run a DSK example to watch data transfer from the DSK to the Mezzanine card.

Outline

- Introduction to SRIO
  - Basic Terminology
  - Example – Direct I/O & CSL Programming Model
- Intro to Message Passing
- Message Passing Examples
- Compare/Contrast Direct I/O vs. Message Passing
- Performance Tips, Collateral, Software Support
- Lab 9
Module Topics

Using Serial Rapid I/O (SRIO)......................................................................................................................... 9-1

Module Topics.................................................................................................................................................. 9-2

Introduction to SRIO........................................................................................................................................ 9-3
  Why Use C6455 SRIO ............................................................................................................................... 9-3
  SRIO Topologies ....................................................................................................................................... 9-4
  SRIO Signals and Ports .............................................................................................................................. 9-4

Basic Terminology ........................................................................................................................................ 9-5
  SCR and SRIO ........................................................................................................................................... 9-6
  SRIO Data Path ....................................................................................................................................... 9-6
  SRIO Block Diagram (Direct I/O and Message Passing) ........................................................................ 9-7

Example – Direct I/O and CSL Programming .............................................................................................. 9-8
  Example #1 – Direct I/O & CSL Programming Model ........................................................................... 9-8
  Direct I/O – Load Store Unit (LSU) ....................................................................................................... 9-9
  Step 1 – Initialize the SRIO Module ..................................................................................................... 9-9
  Steps 2-3 – Configure LSU & Execute Command .................................................................................... 9-10
  Step 5 – Issue Doorbell Interrupt to Target .......................................................................................... 9-10

Message Passing ........................................................................................................................................... 9-11
  Introduction & Message Passing Basics ................................................................................................. 9-11
  CPPI – Communication Port Programming ........................................................................................... 9-12

Message Passing Examples ......................................................................................................................... 9-13
  Example #1 – Receive a MSG ............................................................................................................... 9-13
  Example #2 – 3 C6455’s in One System ................................................................................................. 9-14

Compare/Contrast Direct I/O vs. Message Passing ..................................................................................... 9-15

Performance Tips, Collateral & Software Support ...................................................................................... 9-16

Lab 9: Using C6455 SRIO ............................................................................................................................ 9-17
  Lab Overview: .......................................................................................................................................... 9-17

Lab 9 Procedure .......................................................................................................................................... 9-18
  Part 1 – Running the Code ..................................................................................................................... 9-18
Introduction to SRIO

Why Use C6455 SRIO?

Problem: Your application needs to share data at a high-speed to another processor. Dual Access memories are expensive and need external arbitration logic as well as sharing EMIF bandwidth.

Solution: Use SRIO

- Simplifies interconnects while using an industry-supported high-speed protocol
- C6455 is a high performance programmable DSP allowing flexible system partitioning
- No arbitration logic thus your design can be easily expandable for future growth
- All DSPs run in parallel thus maximizing system throughput
- Two upper layer communication solutions allowing you to pick the optimal method for your system's needs
  - MSG Passing
  - Direct I/O
SRIO Topologies

SRIO – Flexible Topologies

- Ring via 2 1x links
- Mesh with 4 1x Links
- 1x Connection to Switch
- 1x Switch Connection w/ Locals
- 5 DSPs – Complete Connection

SRIO Signals and Ports

SRIO – Serial Interface w/Differential Signals

- Actual data rate is (0.80 * raw rate) shown above (due to 8b/10b decoding)
- Total combined data rate is 2 (Tx/Rx) * 4 ports x 3.125 Gbps * 0.8 = 20Gbps
- A Tx/Rx pair on a single device is called a “link” or “lane”
Basic Terminology

Introduction

### Basic Terminology

#### Architecture
- **Port**: Internal to the SRI\O Module on the C6455 (it has 4 physical ports: 0-3)
- **Link/Lane**: Each Rx/Tx pair is a “link” or “lane”
- **Point-to-Point**: Two devices directly connected to each other
- **LSU**: Load/Store Unit – controls the transmission of DirectIO packets, Doorbell and maintenance packets.
- **CPPI**: “Communication Port Programming Interface” used for Message Passing

#### Programming/Features
- **Device ID**: Identifier of an end point connected to the RapidIO interconnect.
- **DoorBell**: Method for the Sender to interrupt the destination (target) device
- **DirectIO**: Sender can write directly to the target device
- **Message Passing**: Method to send packets between src/dst using packet’s dest for a mail box.
- **Packet Forwarding**: Send packets to a target not directly connected to the sender
- **RapidIO (for more info)**: [www.rapidio.org](http://www.rapidio.org)
**SCR & SRIO**

SCR = Switched Central Resource

- SRIO has its own DMA engine – can initiate a transfer (has access to all memory locations)
- SRIO configuration registers are connected to the CFG SCR

**SRIO Data Path**

- **Some Benefits to Know**
  - SRIO can access entire DSP address space from DSP0 to DSP1 or vice versa
  - Interfaces directly to Switch Central Resource (SCR)
  - Can use EDMA to queue multiple transactions (>4KB)
  - DSP0 can boot or reset DSP1 thru SRIO ports and vice versa
  - Can use Direct I/O or Message Passing to pass packets between the 2 DSPs

- **Some Facts to Know**
  - All physical connections are point-to-point
  - Lanes are ALWAYS bi-directional
  - Each lane can run at 3 speeds (1.25, 2.5, 3.125 Gbps) – Tx or Rx
  - Lanes can be used individually (1x link) or can be bundled together for a faster link (4x link)
  - Lanes are 8b/10b encoded
  - Packet payload is 256 bytes (max). Can send up to 4K bytes (16 packets) without CPU/EDMA intervention
Basic Terminology

SRIO Block Diagram (Direct I/O and Message Passing)

SRIO Conceptual block diagram

Note: 4 LSUs (any LSU can be mapped to any port)

- Load/Store Units
- DirectIO
- Doorbell
- Maintenance

- Message Passing (CPPI)

- MAU – Memory Access Unit
- DirectIO

- Message Passing (CPPI)

- DMA Interface & Controller

- Tx Buffer
- Rx Buffer

- Transaction Mapping

- Logical Layer

- Port 0
  - SerDes 0
- Port 1
  - SerDes 1
- Port 2
  - SerDes 2
- Port 3
  - SerDes 3

1.25-3.125Gbps Differential Signals

Technical Training Organization

TT O
Example – Direct I/O and CSL Programming

Example #1 – Direct I/O & CSL Programming Model

SRIO – Direct I/O Example

- Direct I/O Features
  - Direct transfer of data to the target (must be aware of target’s memory map)
  - Must specify: Target (DeviceID), Memory Address (Destination), Command
  - Can read (pull) or write (push) from/to the Target
  - Use CSL to program registers
  - Can send interrupts via DoorBell to target (Doorbells are a simple way to tell the target device that it has data to process)

- Direct I/O Example – Let’s learn how to program this…

SRIO Programming Model

1. Initialize SRIO Module
2. Configure Load Store Unit (LSU)
   - Choose LSU to process
   - Fill Source info
   - Fill Destination info
   - Fill Packet info
   - Either poll for status or use interrupt
3. Execute the command
4. Do step 2 & 3 for each command
5. Signal target DSP via 1 of 16 Doorbells
Direct I/O – Load Store Unit (LSU)

- There are 4 selectable LSUs to process SRI0 commands
  - Reg 0-3: SRC, DST address and size of transfer
  - Reg 4: Various packet info + interrupt CPU/EDMA event when command completes (if necessary)
  - Reg 5: Issues command and triggers the action/transfer
  - Interrupt Req: can be used to interrupt CPU or generate an EDMA event once the command completes (useful if you want to send packets > 4K bytes)
  - OutportId: port # the LSU cmd goes out on – all 4 LSUs can be used by the same port
  - Byte_count: #bytes to send out (max is 4Kbytes) – packet size is 256 bytes

Packet Types
- "Commands"
  - NWRITE
  - NWRITE_R
  - NREAD
  - DOORBELL
  - MESSAGE
  - MAINTENANCE
  - ATOMIC
  - SWRITE

Step 1 – Initialize the SRIO Module

#include <csl_srio.h>

CSL_SrioContext context;
CSL_Status status;
CSL_SrioHandle hSrio;
CSL_SrioObj srioObj;

// Initialization and Open of the SRIO
status = CSL_srioInit(&context);

hSrio = CSL_srioOpen(&srioObj, srioNum, &srioParam, &status);

// Create the setup parameters for the SRIO module. Use default Param srio_Create_Setup (&setup, 1, 1);
// configure the SRIO Hardware with the above setup Parameters
status = CSL_srioHwSetup(hSrio, &setup);

CSL_srioGetHwStatus(hSrio, CSL_SRIO_QUERY_SP_ERR_STAT, &response);
CSL_srioGetHwStatus(hSrio, CSL_SRIO_QUERY_DOORBELL_INTR_STAT, &dbStatus);
CSL_srioHwControl(hSrio, CSL_SRIO_CMD_DOORBELL_INTR_CLEAR, &dbStatus);
CSL_srioHwControl(hSrio, CSL_SRIO_CMD_INT_DST_RATE_CNTL, 0);
### Steps 2-3 – Configure LSU & Execute Command

<table>
<thead>
<tr>
<th>Steps 2 &amp; 3 – Configure LSU &amp; Execute</th>
<th>LSU that processes the Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2</strong> Configure LSU</td>
<td>UInt8 IsuNum = LSU_0; CSL_SrioDirectIO_ConfigXfr IsuConf;</td>
</tr>
</tbody>
</table>

#### Local Device Info
- IsuConf.srcNodeAddr = src;
- IsuConf.outPortId = PORT_0;
- IsuConf.dstNodeAddr.addressLo = dst;
- IsuConf.dstNodeAddr.addressHi = 0;
- IsuConf.xambs = 0;
- IsuConf.idSize = 1;
- IsuConf.dstId = DSP_1;
- IsuConf.pktType = SRIO_PKT_NWRITE;
- IsuConf.byteCnt = len;
- IsuConf.priority = 2;
- IsuConf.hopCount = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = 0;

#### Remote Device Info
- IsuConf.srcNodeAddr = src;
- IsuConf.outPortId = PORT_0;
- IsuConf.xambs = 0;
- IsuConf.idSize = 1;
- IsuConf.dstId = DSP_1;
- IsuConf.pktType = SRIO_PKT_NWRITE;
- IsuConf.byteCnt = len;
- IsuConf.priority = 2;
- IsuConf.hopCount = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = 0;

#### Packet Info
- IsuConf.byteCnt = len;
- IsuConf.priority = 2;
- IsuConf.hopCount = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = 0;

### Step 5 – Issue Doorbell Interrupt to Target

<table>
<thead>
<tr>
<th>Step 5 – Issue Doorbell</th>
<th>LSU that process the Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2</strong> Configure LSU</td>
<td>UInt8 IsuNum = LSU_0; CSL_SrioDirectIO_ConfigXfr IsuConf;</td>
</tr>
</tbody>
</table>

#### Local Device Info
- IsuConf.srcNodeAddr = 0;
- IsuConf.outPortId = PORT_0;
- IsuConf.xambs = 0;
- IsuConf.idSize = 1;
- IsuConf.dstId = DSP_1;
- IsuConf.pktType = SRIO_PKT_DOORBELL;
- IsuConf.byteCnt = 0;
- IsuConf.priority = 2;
- IsuConf.hopCount = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = info;

#### Remote Device Info
- IsuConf.srcNodeAddr = 0;
- IsuConf.outPortId = PORT_0;
- IsuConf.xambs = 0;
- IsuConf.idSize = 1;
- IsuConf.dstId = DSP_1;
- IsuConf.pktType = SRIO_PKT_DOORBELL;
- IsuConf.byteCnt = 0;
- IsuConf.priority = 2;
- IsuConf.hopCount = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = info;

#### Packet Info
- IsuConf.byteCnt = 0;
- IsuConf.intrReq = 0;
- IsuConf.doorbellInfo = info;

| **3** Execute the command | CSL_srioLsuSetup (hSrio, &IsuConf, IsuNum); |

---

9 - 10 | C645x/C642x Integration Workshop - Using Serial Rapid I/O (SRIO)
Message Passing

Introduction & Message Passing Basics

A Message is simply “Data”.

Message Passing is a method to send data from one device to another without requiring the sender to know the target’s memory map.

From a high-level point of view, DSP_0 simply specifies a location (SRC) of MSG0 in DSP_0’s memory map and a mailbox number on DSP_1. DSP_1 fills out a table that connects a specific mailbox to a memory location (DST) on DSP_1.

Needs to know:
- Which DSP_0 mailbox?
- Which DSP_1 mailbox?
- Src addr of MSG0
- Len of MSG0
- Misc: DestID, PortID, etc.

Message Passing – Basics

The Rapid I/O spec dictates that a device must have a minimum of 4 mailboxes.

TI’s implementation uses 4 Tx and 4 Rx mailboxes per physical port (0,1,2,3). In addition, there are 16 Tx and 16 Rx queues that can be assigned to any mailbox.

Similar to Direct I/O, the max payload is 256 bytes. If a MSG is larger than 256 bytes, but smaller than 4KB, the message is broken into multiple segments.

TI’s implementation uses DESCRIPTORS (similar to EMAC descriptors) to describe the transfer (src, dst, len, options). There is one descriptor per MSG (regardless of the number of segments).
CPPI – Communication Port Programming

- The RapidIO **Message Passing** uses CPPI to DMA data between the RapidIO device and the memory system.
- The Buffer Descriptor is used to interface to the CPPI DMA engine:
  - **Transmit**: filled-in buffer descriptor is given to one of the Tx CPPI queues.
  - **Receive**: ISR processes a filled-in buffer descriptor that points to a received message.

**Buffer Descriptor**
- **next**: ptr to next buffer descriptor – allows chaining during xmt/rcv
- **buffer**: ptr to the physical location of the message
- **routing**: routing info (Rx – src device ID, priority, dest mailbox)  
  (Tx – dest device ID, priority, port ID, SSIZE, dest mailbox)
- **len+flags**: length info (Rx – start/end of msg, end of queue, ownership, len, completion code)  
  (Tx – start/end of msg, end of queue, ownership, len, completion code, retry count)
Message Passing Examples

Example #1 – Receive a MSG

- Snapshot of system before MSG comes in.
- CPU assigns a free buffer descriptor and fills it in. The bufferPtr points to the DST location of the MSG on the target device.
- Then, the CPU writes to the RX HDP to point to the first descriptor. The target is now ready to receive MSGs.

When MSG comes in, the RapidIO peripheral DMA’s the MSG into the buffer, updates routing info & flags, updates its HD/P/C, then sets the ICSR bit (bit 0 in this example), then interrupts the processor.

Part of the packet header that is being received contains status, length, etc. regarding the MSG being received. This info is used to update the buffer descriptor.
Example #2 – 3 C6455’s in One System

You can have different queues being processed at the same time...
Compare/Contrast Direct I/O vs. Message Passing

<table>
<thead>
<tr>
<th>Direct I/O</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Performance</td>
<td>Most Flexible</td>
</tr>
<tr>
<td>Target buffering scheme is known at design time</td>
<td>Target buffering scheme is unknown</td>
</tr>
<tr>
<td>Application partitioning is fixed</td>
<td>Target provides abstract buffers – “mailboxes” &amp; “letters”</td>
</tr>
<tr>
<td>Push data flow (NWRITE) has the best performance</td>
<td>Application partitioning is unknown</td>
</tr>
<tr>
<td>Want a low-level interface to data movement</td>
<td>Abstract interface to data movement</td>
</tr>
<tr>
<td>Easier to set up</td>
<td>More difficult to set up, easier once the set up is done.</td>
</tr>
</tbody>
</table>

DSP/BIOS: MSGQ

- BIOS MSGQ Module supports SRIO Message Passing
- Portable code for intra- and inter-processor support
**SRIO Performance - Tips**

- NWRITE is faster than NREAD
- NWRITE_R is faster than NREAD
- Signal integrity is very important (SPRAAA8)
- Can combine all 4 Lanes into one lane for transfer rate improvement. This is a special mode in SRIO. Data is “TDMed” across the 4 ports.
- Can use more than 1 lane to increase the bit rate between 2 devices. This has to be managed by software.
- Internal lookup table for packet forwarding so there may be no need for a switch.

**TI Supplied Collateral**

- SPRU976 – RapidIO Peripheral Module Guide
  - Detailed functional description
- SPRAAA8 – Hi-speed Board Design Guidelines
  - Board layout and stack up requirements
- TI DSP Starter Kit / Evaluation Module
  - Reference design board with two C6455 devices
- Functional Layer CSL for Configuration & Direct IO
- MSGQ Documentation
Lab 9: Using C6455 SRIO

Lab Overview:

The goal of this lab is for you to be familiarized with the process of using the Serial Rapid Input Output (SRIO) for C6455. You will learn to use Direct IO to access the SRIO. To gain this basic knowledge you will:

- Observe and analyze the master-slave example
- Observe the Direct IO programming method used with CSL and SRIO
- Run some tests

Lab9 – SRIO Direct I/O

DSP_0: DSK

- MASTER_WRITE_BUFF
  - 0x90 0000
- MASTER_READ_BUFF
  - 0x90 0100

DSP_1: Mezzanine

- SLAVE_BUFF
  - 0x90 0000

Lab’s Objective

1. Analyze code example
2. View data transfer between DSPs

srio_master.c

- writes 1 SRIO packet (256 bytes) to the slave device
- After write, it sends doorbell 0 to the slave device
- In this file, read the note about the reason for the delay between write and read back completion
- After the delay, master reads 1 SRIO packet (256 bytes) from the slave device
- After read completion, it sends doorbell 1 to the slave device

srio_slave.c

- Loop forever wand wait for Master device to send doorbell interrupts.
- Doorbell interrupts cause slave’s ISR to response
Lab 9 Procedure

Part 1 – Running the Code

You will be running code on the DSK and on the Mezzanine card. It does not matter which one is the master device and which one is the slave device. In the lab procedure, we pick CPU_0 (the DSK) as the master and the CPU_1 (the Mezzanine card) as the slave. This is intended so that in the future, we will port this code example to the audio project we used in previous labs.

1. **Connect C6455 DSK to the Mezzanine EVM card (with power disconnected).**
   - Quit CCS
   - Remove power from the DSK
   - Connect the Mezzanine card.
   - Plug back in the power connection to the DSK.

Note: if the Mezzanine card does not work properly, make sure the card is seated firmly in the socket. If it is, you may need to pull it out slightly to get it to work. We’ve seen this happen on multiple boards. Spectrum Digital is aware of the issue and is working to resolve this.
2. Set up the Parallel Debug Manager

- Start the CCS Setup utility using its desktop icon.

Be aware there are two CCS icons, one for setup, and the other to start the CCS application. You want the Setup CCSStudio v3.2 icon.

When you open CC_Setup, you should see a screen similar to this:

![Screen showing setup utility]

- Clear any old system configurations.

If there are any boards/simulators listed under My System under System Configuration, click the Remove All button to clear the configuration.

- Use the filters to select the correct Factory Board.

To the right of Available Factory Boards, you will see 3 filters (Family, Platform and Endianness). Use the drop down boxes and make the selections shown to select the correct board.

![Screen showing filters]

- Add the proper factory board.

Click on the “C6455 DSK with Mezzanine” and select << Add. This board should now show up under My System.

- Select Save and Quit.

- When prompted to start CCS, click Yes (or click on the CCS 3.2 icon to launch CCS with Parallel Debug Manager).
3. **Connect the boards and open CCS windows for each CPU.**

   Select:
   
   Debug → Connect

   This should connect both boards (no red circle with slash on cpu_0 and cpu_1).

   Select:
   
   Open → cpu_0

   This will open a CCS window pointing to cpu_0 (i.e. the DSK). Now open cpu_1 as well (the CPU on the Mezzanine card). Now you have two CCS windows open – one for each CPU:

   - cpu_0: DSK
   - cpu_1: Mezzanine card

4. **Open master project for cpu_0 (DSK).**

   Make sure you have the active CCS window for cpu_0 on your screen. For cpu_0, open the project srio_master.pjt under the directory path:

   C:\iW64x+\labs\SRIO_MasterSlave_DIO\master

5. **Open slave project for cpu_1 (Mezzanine).**

   Now, switch to the CCS window for cpu_1. Open the project srio_slave.pjt under the directory path:

   C:\iW64x+\labs\SRIO_MasterSlave_DIO\slave
6. **Scan through and analyze the source files**

In master.pjt, there are 3 source files. In slave.pjt, there are 2 source files. Note that the SetUp_Srio.c is the same exact source file for both projects. These files are stored under:

```
C:\iw64x+\labs\SRIO_MasterSlave_DIO\src
```

The code flow under main() in srio_master.c is as follows:

- Master writes 1 SRIO packet (256 bytes) to the slave device
- After write completion, it sends doorbell 0 to the slave device
- After the delay, master reads 1 SRIO packet (256 bytes) from the slave device
- After read completion, it sends doorbell 1 to the slave device

The main code has 2 loops. The inner loop runs 10 times to send 10 SRIO packets then stops. The outer loop waits for a user input. You can trigger this command via any means that you can think off. In this lab, we will use the GEL command which is shown in the steps below.

---

**Note:** For SRIO, you want to optimize it by architecting your system to do only writes, never reads. For example, rather than trying to read a buffer from another device through SRIO, you would instead do a write to let that device know to write that buffer to you. The reason for this is that you get CPU stalls while waiting for each read to complete whereas having the other device write the data into your memory allows the CPU to keep crunching along and then you can get an interrupt at the end of the transfer.

The code flow in srio_slave.c is as follows:

- Loop forever and wait for the Master device to send a doorbell interrupt.

Look into the slave ISR to see how the doorbells are handled.

---

**Note:** This is the portion of the code that you will need to modify to handle doorbell messaging. If you did not have the master do the read, then you can use doorbells to signal the slave to write the data back to the master device.

---

7. **Build, Load, & Run.**

First, build the code on slave side (cpu_1), then run it.

Second, build code on the master side (cpu_0), then run it.

Observe the messages in the stdout for both master & slave CCS windows.
8. **Re-run the test**

Load GEL file for master (cpu_0) project:

File → Load GEL …

C:\iw64x+\labs\SRIO_MasterSlave_DIO\src\Control.GEL

Run GEL command:

GEL → Next Run Dialog – Set_NextRun.

Enter a 1 then click on the Execute button.

You should see another set of 10 runs. Click Done.

9. **Open Memory Windows**

In the master project, open a Memory window address at 0x90 0000. This is the location of MASTER_WRITE_BUFF (pSrioData = 0x90 0000). Look in the header file, srio_Lab.h, and see the definition of MASTER_WRITE_BUFF. Also, look in srio_master.c (line 22) and you can see the pointer (*pSrioData) set to MASTER_WRITE_BUFF.

In the slave project, open a Memory window address at 0x90 0000 (SLAVE_BUFF).

Re-size the memory windows of both CCS windows so that you can see both memory windows (master and slave) at the same time.

10. **View the data transfer**

To see the data transfer, we need to set a breakpoint in both projects. First, if the master and slave are running, halt both processors. The animate key (instead of run) will run to a breakpoint in the slave code, then halt and display the results in the memory window, then it will run again.

Set a breakpoint in srio_master.c at line 40 (while loop). Set a breakpoint in srio_slave.c on line 100 (while loop).

Click on the slave’s (cpu_1) animation button (just underneath the Halt button).

Run the master (cpu_0). If needed, click on Set_NextRun Execute to continue with the transfer.

To observe how the data transfers from the master’s CPU memory to slave’s CPU memory, use the GEL command in the master code to re-run the loop.

---

**You’re Done**