

Using Serial Rapid I/O (SRIO)

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

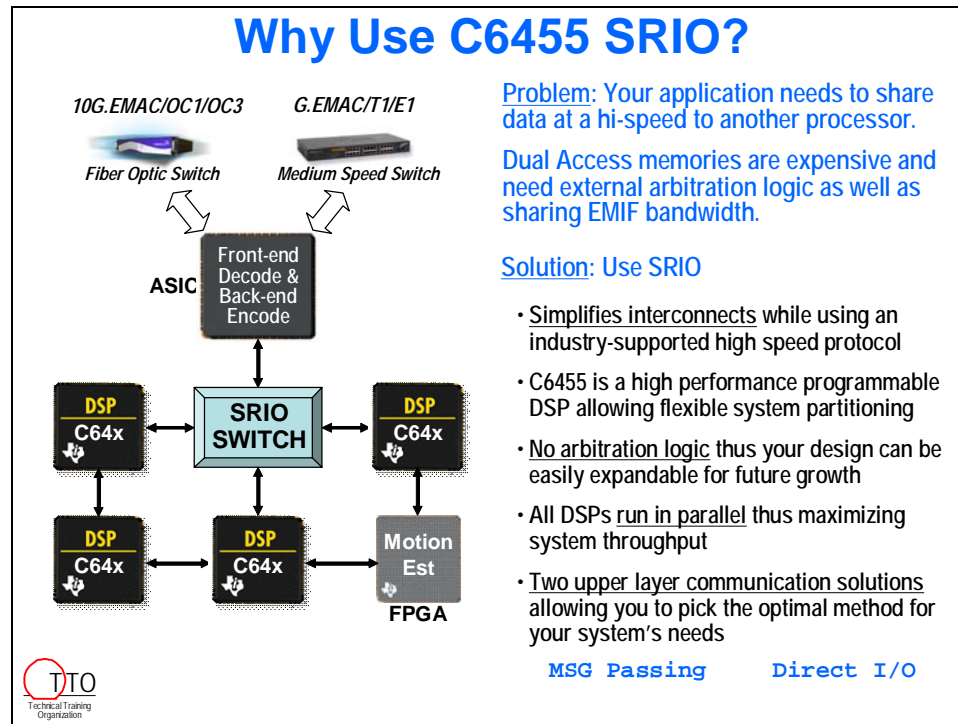


Module Topics

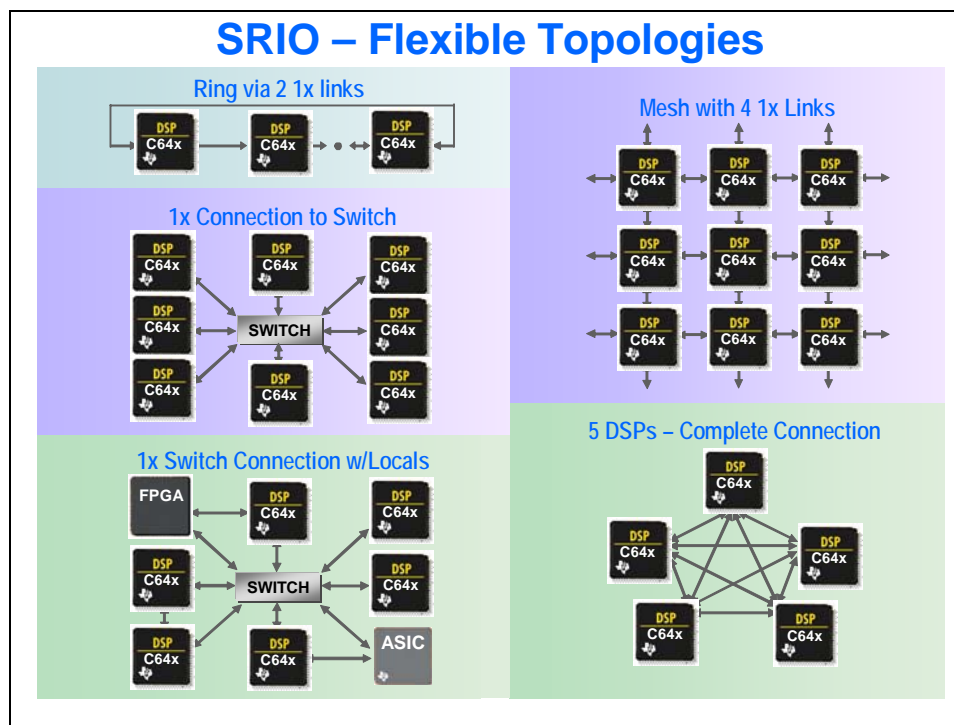
Using Serial Rapid I/O (SRIO).....	9-1
<i>Module Topics.....</i>	9-2
<i>Introduction to SRIO.....</i>	9-3
Why Use C6455 SRIO	9-3
SRIO Topologies	9-4
SRIO Signals and Ports	9-4
<i>Basic Terminology.....</i>	9-5
Introduction	9-5
SCR and SRIO.....	9-6
SRIO Data Path	9-6
SRIO Block Diagram (Direct I/O and Message Passing)	9-7
<i>Example – Direct I/O and CSL Programming</i>	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
<i>Message Passing.....</i>	9-11
Introduction & Message Passing Basics	9-11
CPPI – Communication Port Programming.....	9-12
<i>Message Passing Examples</i>	9-13
Example #1 – Receive a MSG.....	9-13
Example #2 – 3 C6455's in One System	9-14
<i>Compare/Contrast Direct I/O vs. Message Passing</i>	9-15
<i>Performance Tips, Collateral & Software Support.....</i>	9-16
<i>Lab 9: Using C6455 SRIO</i>	9-17
Lab Overview:	9-17
<i>Lab 9 Procedure</i>	9-18
Part 1 – Running the Code.....	9-18

Introduction to SRIO

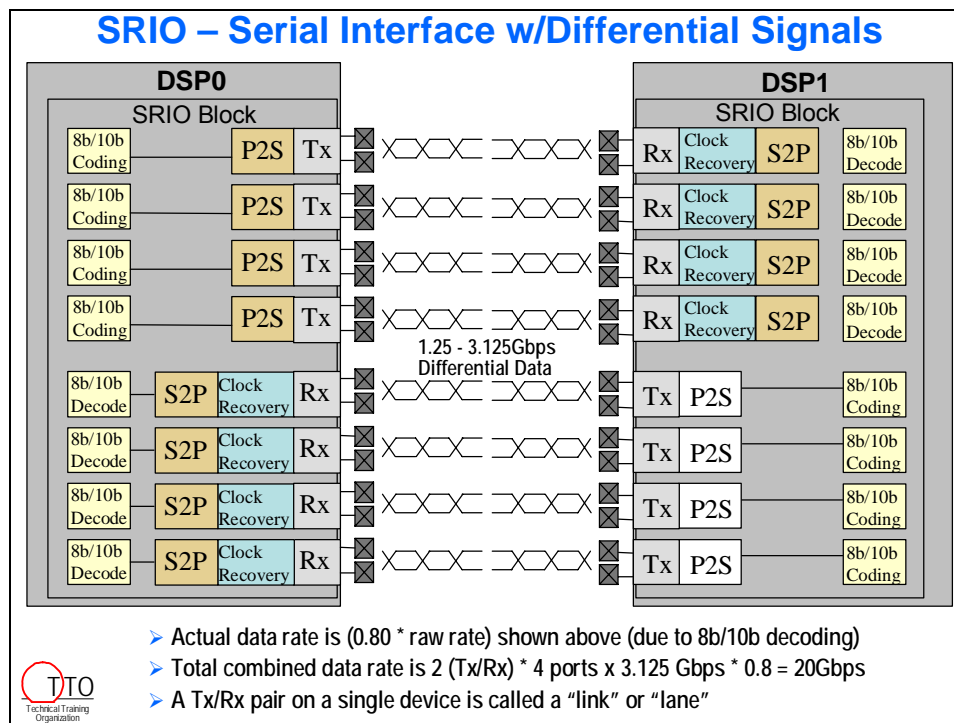
Why Use C6455 SRIO



SRIO Topologies



SRIO Signals and Ports



Basic Terminology

Introduction


Basic Terminology

Architecture

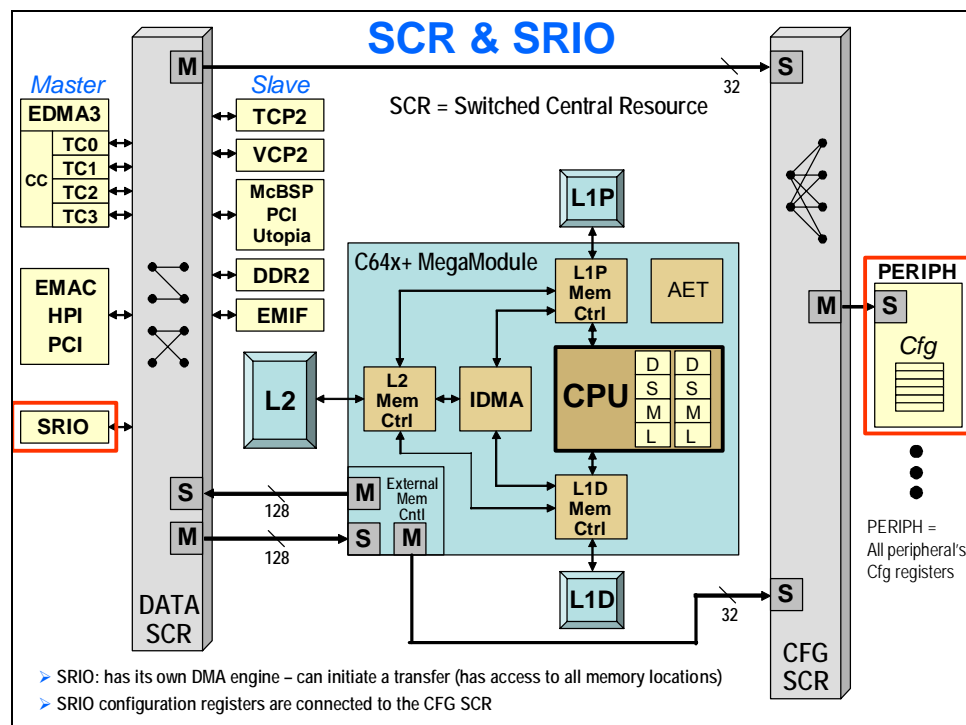
- **Port:** Internal to the SRIO Module on the C6455 (it has 4 physical ports: 0-3)
- **Link/Lane:** Each Rx/Tx pair is a “link” or “lane”
- **Point-to-Point:** Two devices directly connected to each other
- **LSU:** Load/Store Unit – controls the transmission of DirectIO packets, Doorbell and maintenance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

Programming/Features

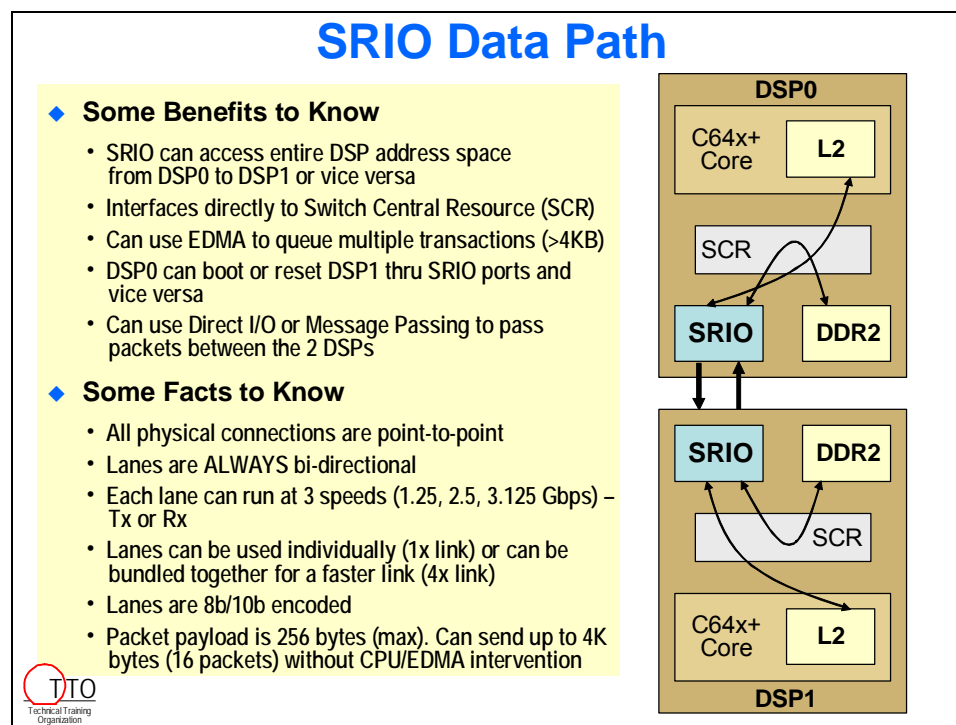
- **Device ID:** Identifier of an end point connected to the RapidIO interconnect.
- **DoorBell:** Method for the Sender to interrupt the destination (target) device
- **DirectIO:** Sender can write directly to the target device
- **Message Passing:** Method to send packets between src/dst using packet’s 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



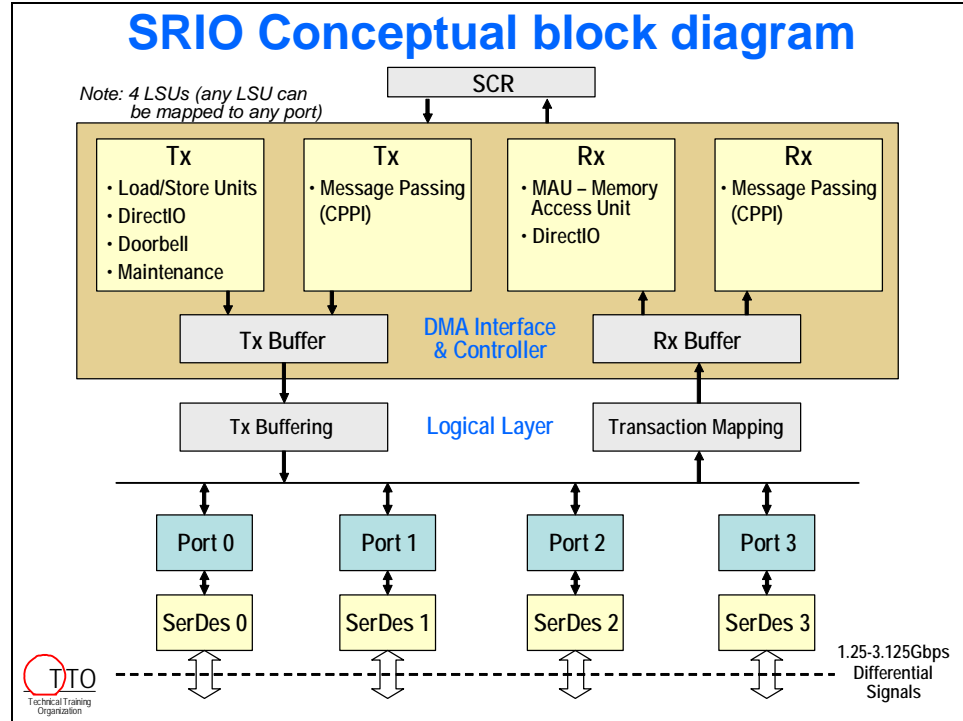
SCR and SRIO



SRIO Data Path

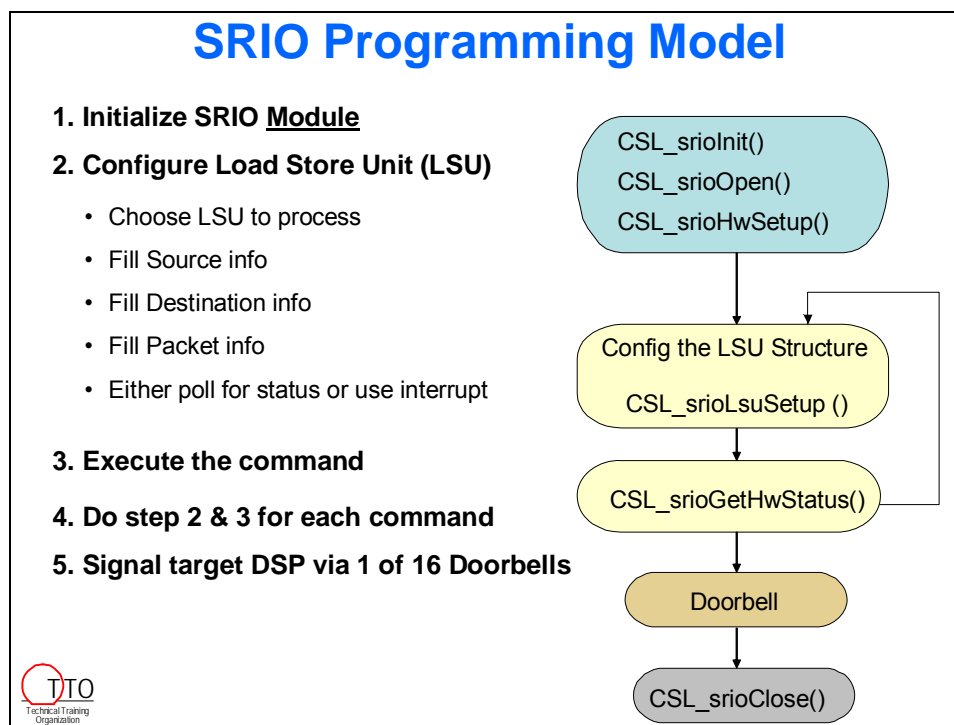
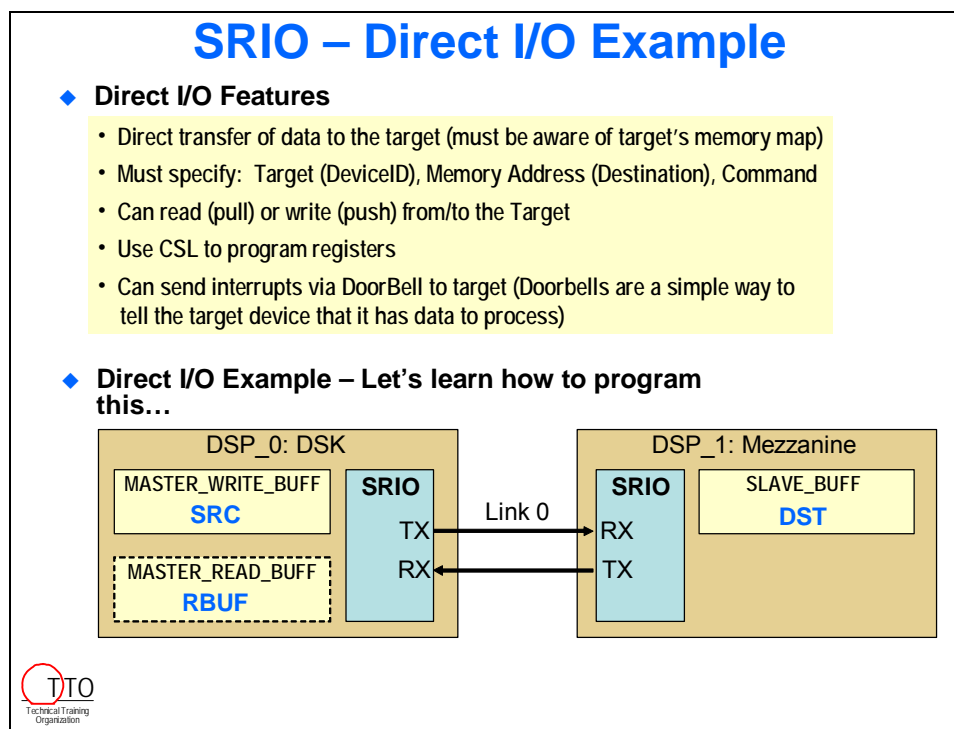


SRIO Block Diagram (Direct I/O and Message Passing)



Example – Direct I/O and CSL Programming

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



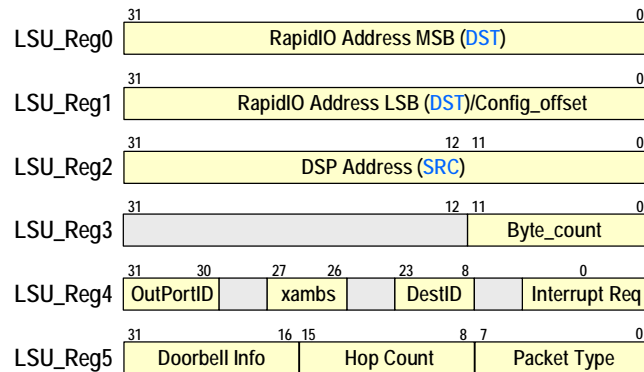
Direct I/O – Load Store Unit (LSU)

Direct I/O – Load Store Unit (LSU)

- ◆ There are 4 selectable LSUs to process SRIO 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

Step 1 – Initialize SRIO Module

1 Initialize SRIO Module

Function call to create setup structure for SRIO H/W
(Note: not a CSL function call.
You can modify to fit your needs)

Verify h/w was set up properly

Set other control bits

```
#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_INTDST_RATE_CNTL, 0);
```

Steps 2-3 – Configure LSU & Execute Command


Steps 2 & 3 – Configure LSU & Execute

2 Configure LSU	LSU that processes the Command	<pre> UInt8 IsuNum = LSU_0; CSL_SrioDirectIO_ConfigXfr IsuConf; </pre>
	Local Device Info	<pre> IsuConf.srcNodeAddr = src; IsuConf.outPortId = PORT_0; </pre>
	Remote Device Info	<pre> IsuConf.dstNodeAddr.addressLo = dst; IsuConf.dstNodeAddr.addressHi = 0; IsuConf.xambs = 0; IsuConf.idSize = 1; IsuConf.dstId = DSP_1; </pre>
	Packet Info	<pre> IsuConf.pktType = SRIO_PKT_NWRITE; IsuConf.byteCnt = len; IsuConf.priority = 2; IsuConf.hopCount = 0; </pre>
	Signaling Mechanism	<pre> IsuConf.intrReq = 0; IsuConf.doorbellInfo = 0; </pre>
3 Execute the command		<pre> CSL_srioLsuSetup (hSrio, &IsuConf, IsuNum); </pre>

Step 5 – Issue Doorbell Interrupt to Target

Step 5 – Issue Doorbell

2 Configure LSU	LSU that process the Command	<pre> UInt8 IsuNum = LSU_0; CSL_SrioDirectIO_ConfigXfr IsuConf; </pre>
	Local Device Info	<pre> IsuConf.srcNodeAddr = 0; IsuConf.outPortId = PORT_0; IsuConf.xambs = 0; </pre>
	Remote Device Info	<pre> IsuConf.idSize = 1; IsuConf.dstId = DSP_1; IsuConf.pktType = SRIO_PKT_DOORBELL; IsuConf.byteCnt = 0; IsuConf.priority = 2; </pre>
	Packet Info	<pre> IsuConf.hopCount = 0; IsuConf.intrReq = 0; IsuConf.doorbellInfo = info; </pre>
3 Execute the command		<pre> CSL_srioLsuSetup (hSrio, &IsuConf, IsuNum); </pre>



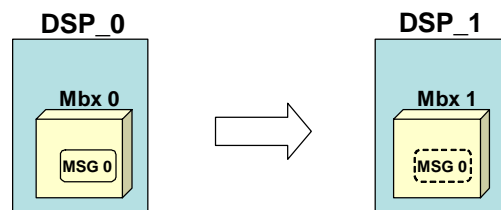
TTO
Technical Training
Organization

Message Passing

Introduction & Message Passing Basics

Intro to Message Passing

- 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.

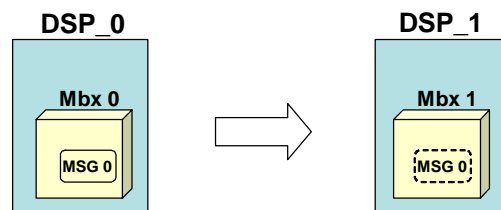
Needs to know:

- How each mailbox is mapped to a specific destination address in memory
- If mailbox1 gets a message, e.g., it places the data (MSG) at a specific address



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)



Tx Buffer Descriptor

- Pointer to next descriptor
- Pointer to SRC location of MSG 0
- Routing info (DST deviceID, portID, segment size, DST mailbox)
- MSG length + flags/status

Rx Buffer Descriptor

- Pointer to next descriptor
- Pointer to DST location of MSG 0
- Routing info (SRC deviceID, DST mailbox)
- MSG length + flags/status



CPPI – Communication Port Programming

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.

CPPI Buffer Descriptor

next
buffer
routing
Len+flags

◆ 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)

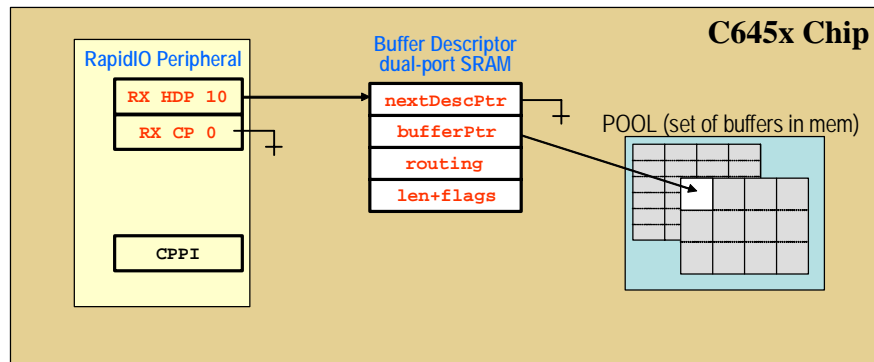


Message Passing Examples

Example #1 – Receive a MSG

Example #1 – Receive 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.



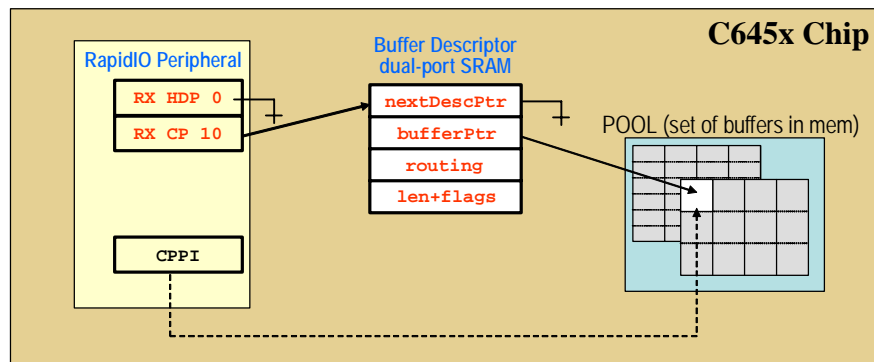
RX HDP x: Header Descriptor Pointer Per Queue (points to the top of the first descriptor)

RX CP x: Completion Pointer Per Queue (points to the top of the descriptor that has been completed)



Example #1 – Receive MSG

- When MSG comes in, the RapidIO peripheral DMA's the MSG into the buffer, updates routing info & flags, updates its HDP/CP, 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.

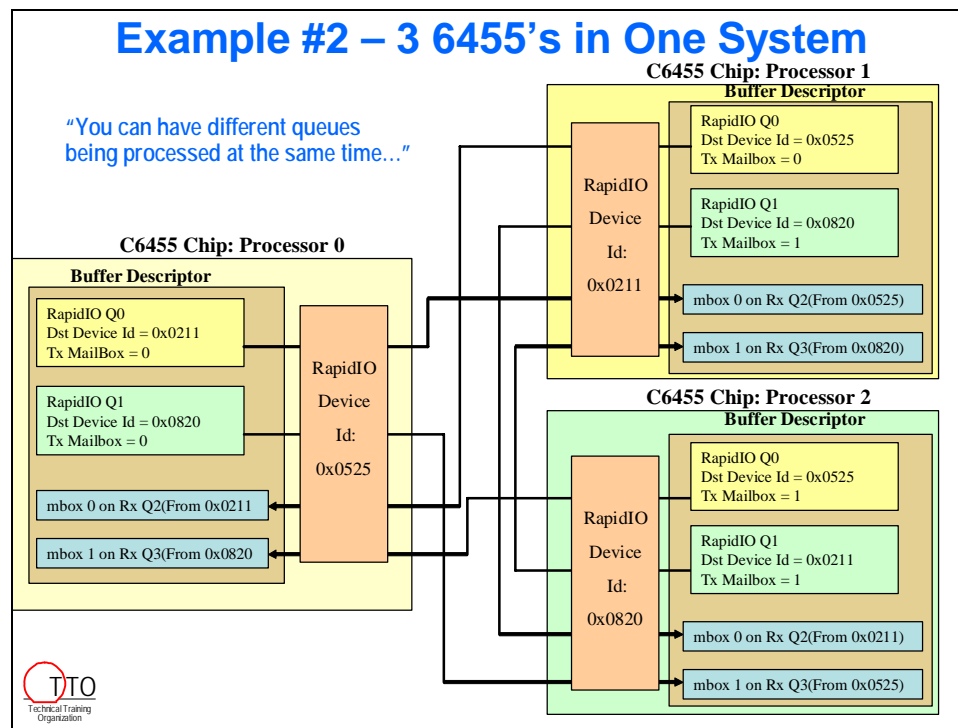


RX HDP x: Header Descriptor Pointer Per Queue (points to the top of the first descriptor)

RX CP x: Completion Pointer Per Queue (points to the top of the descriptor that has been completed)



Example #2 – 3 C6455's in One System



Compare/Contrast Direct I/O vs. Message Passing

Direct I/O vs. Message Passing

Direct I/O

- ◆ Best Performance
- ◆ Target buffering scheme is known at design time
- ◆ Application partitioning is fixed
- ◆ Push data flow (NWRITE) has the best performance
- ◆ Want a low-level interface to data movement
- ◆ Easier to set up

Message Passing

- ◆ Most Flexible
- ◆ Target buffering scheme is unknown
- ◆ Target provides abstract buffers – “mailboxes” & “letters”
- ◆ Application partitioning is unknown
- ◆ Abstract interface to data movement
- ◆ More difficult to set up, easier once the set up is done.

DSP/BIOS: MSGQ

- BIOS MSGQ Module supports SRIO Message Passing
- Portable code for intra- and inter-processor support
- www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html

Performance Tips, Collateral & Software 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.**

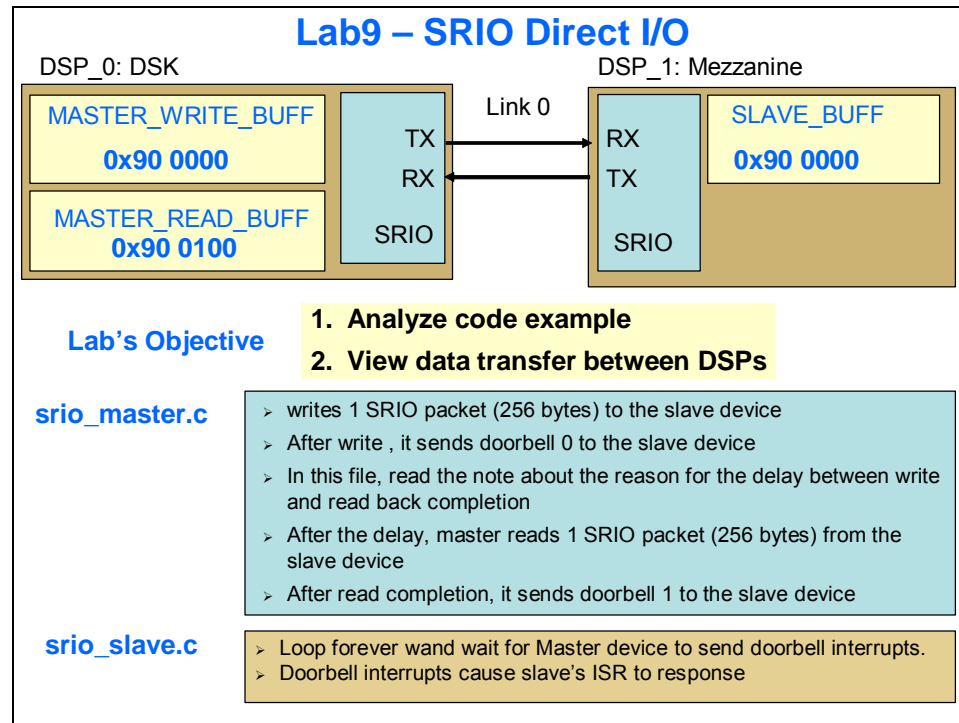


TI Supplied Collateral

- ◆ **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

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.

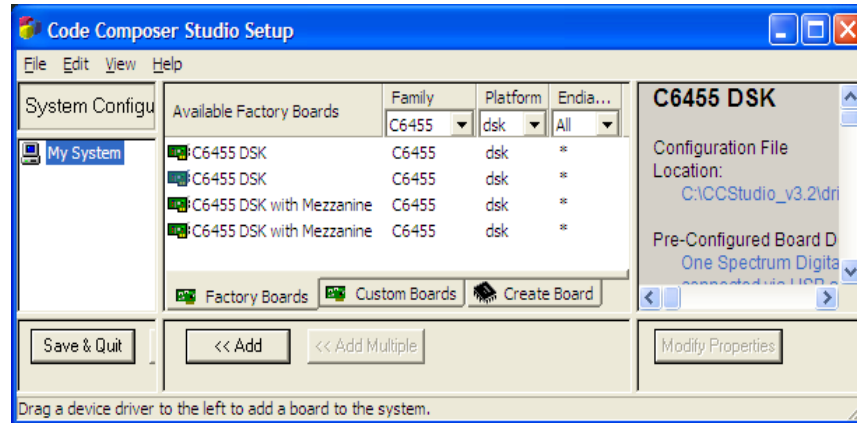
Note: if the Mezzanine card does not work properly, make sure the card is seated firmly in the socket. If it is, you may need to pull it out slightly to get it to work. We've seen this happen on multiple boards. Spectrum Digital is aware of the issue and is working to resolve this.

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:

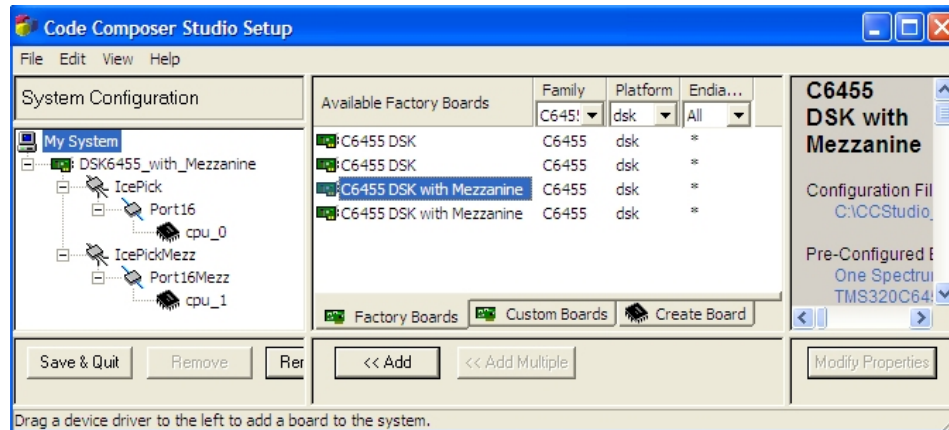


- **Clear any old system configurations.**

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.



- **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