Application Report
Lit. Number – December 2011

SRIO Programming and Performance Data on Keystone DSP

Zhan Xiang, Brighton Feng *Communication Infrastructure*

ABSTRACT

SRIO (Serial RapidIO) on Keystone DSP is a very flexible and powerful; it provides many modes and options for customer to use it, the SRIO user's guide describes a lot about these. This application note provides some complementary information about the programming of the SRIO.

This application report also discusses the performance of SRIO, provides measured performance data achieved under various operating conditions. Some factors affecting SRIO performance are discussed.

Contents

Figures

Tables

1 Introduction

RapidIO is a non-proprietary, high-bandwidth, system-level interconnect. It's a packet-switched interconnect intended primarily as an intra-system interface for chip-to-chip and board-to-board communications at gigabyte-per-second performance levels. The RapidIO peripheral used in KeyStone devices is called Serial RapidIO (SRIO).

SRIO (Serial RapidIO) on Keystone DSP is a very flexible and powerful; it provides many modes and options for customer to use it, the SRIO user's guide describes a lot about these. This application note provides some complementary information about the programming of the SRIO.

Example code/project is provided along with this application note. The example code about programming of the SRIO is based on register layer CSL (Chip Support Layer). Most code use the SRIO register pointer defined as following:

```
#include <ti/csl/cslr_srio.h>
#include <ti/csl/cslr_device.h>
CSL_SrioRegs *  srioRegs =   (CSL_SrioRegs *)CSL_SRIO_CONFIG_REGS;
```

This application report also discusses the performance of SRIO, provides measured performance data achieved under various operating conditions. Some factors affecting SRIO performance are discussed.

2 SRIO configuration

2.1 1x/2x/4x configuration

The KeyStone Family SRIO port configuration use the PLM Port Path Control Register to configure the SRIO port mode, the PATH_CONFIG field is used to configure how many lanes are used, and PATH_MODE field is used to select the combination of 1x, 2x or 4x ports. Figure 6 shows the detail description for the port configuration.

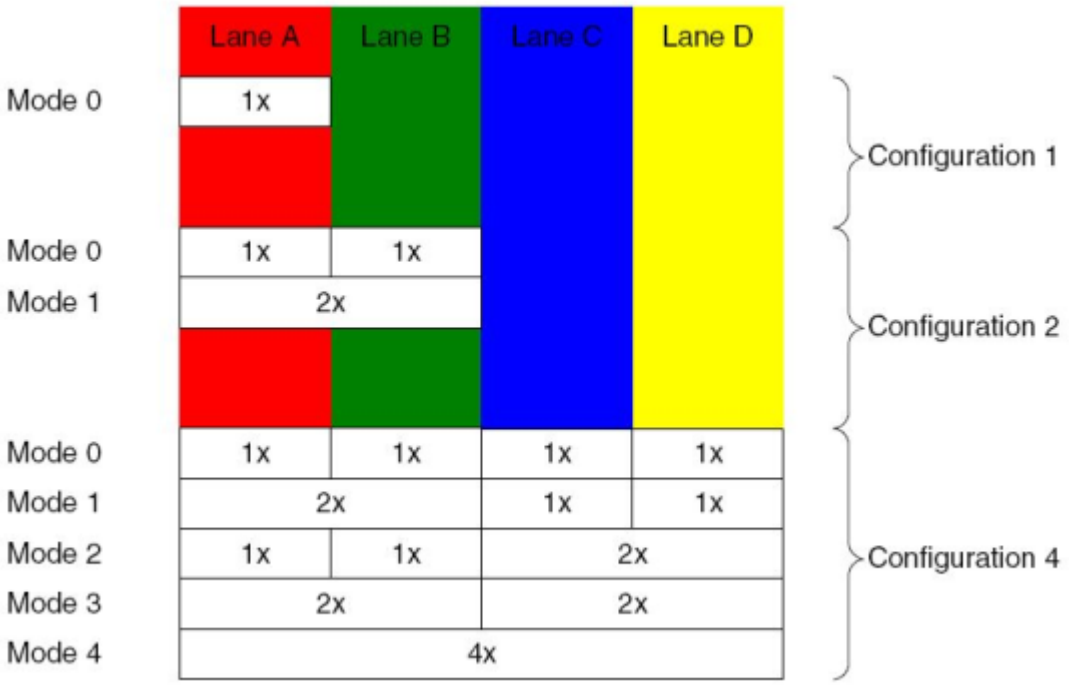

Figure 1 SRIO Port Mode Configuration

Below is the example code to configure it:

```
typedef enum
{
 SRIO_PATH_CTL_1xLaneA =
  (1<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (0<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_1xLaneA_1xLaneB =
  (2<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (0<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_2xLaneAB =
  (2<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (1<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_1xLaneA_1xLaneB_1xLaneC_1xLaneD =
  (4<<CSL SRIO RIO PLM SP PATH CTL PATH CONFIGURATION SHIFT)|
  (0<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_2xLaneAB_1xLaneC_1xLaneD =
  (4<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (1<<CSL SRIO RIO PLM SP PATH CTL PATH MODE SHIFT),

 SRIO_PATH_CTL_1xLaneA_1xLaneB_2xLaneCD =
  (4<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (2<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_2xLaneAB_2xLaneCD =
```

```
  (4<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (3<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT),

 SRIO_PATH_CTL_4xLaneABCD =
  (4<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_CONFIGURATION_SHIFT)|
  (4<<CSL_SRIO_RIO_PLM_SP_PATH_CTL_PATH_MODE_SHIFT)
}SRIO_1x2x4x_Path_Control;

/*configure SRIO 1x 2x or 4x path mode*/
void Keystone_SRIO_set_1x2x4x_Path(
            SRIO_1x2x4x_Path_Control srio_1x2x4x_path_control)
{
 /*This register is a global register, even though it can be accessed
 from any port. So you do not need to program from each port, it is
 basically a single register. */
 srioRegs->RIO_PLM[0].RIO_PLM_SP_PATH_CTL=
  (srioRegs->RIO_PLM[0].RIO_PLM_SP_PATH_CTL&(~SRIO_1x2x4x_PATH_CONTROL_MASK))|
  srio_1x2x4x_path_control;
}
……
 Keystone_SRIO_set_1x2x4x_Path(SRIO_PATH_CTL_1xLaneA);
```

Please note, ports within a path are numbered according to the “lowest” lane that they use:

If lane A is used by a port, the port number is zero; and the port may be 1x, 2x or 4x.

If lane B is the lowest lane, the port number is 1; and the port must be a 1x port.

If lane C is the lowest lane, the port number is 2; and the port may be either 1x or 2x.

If lane D is the lowest lane, the port number is 3; and the port must be a 1x port.

Following figure show the port number for different configuration.

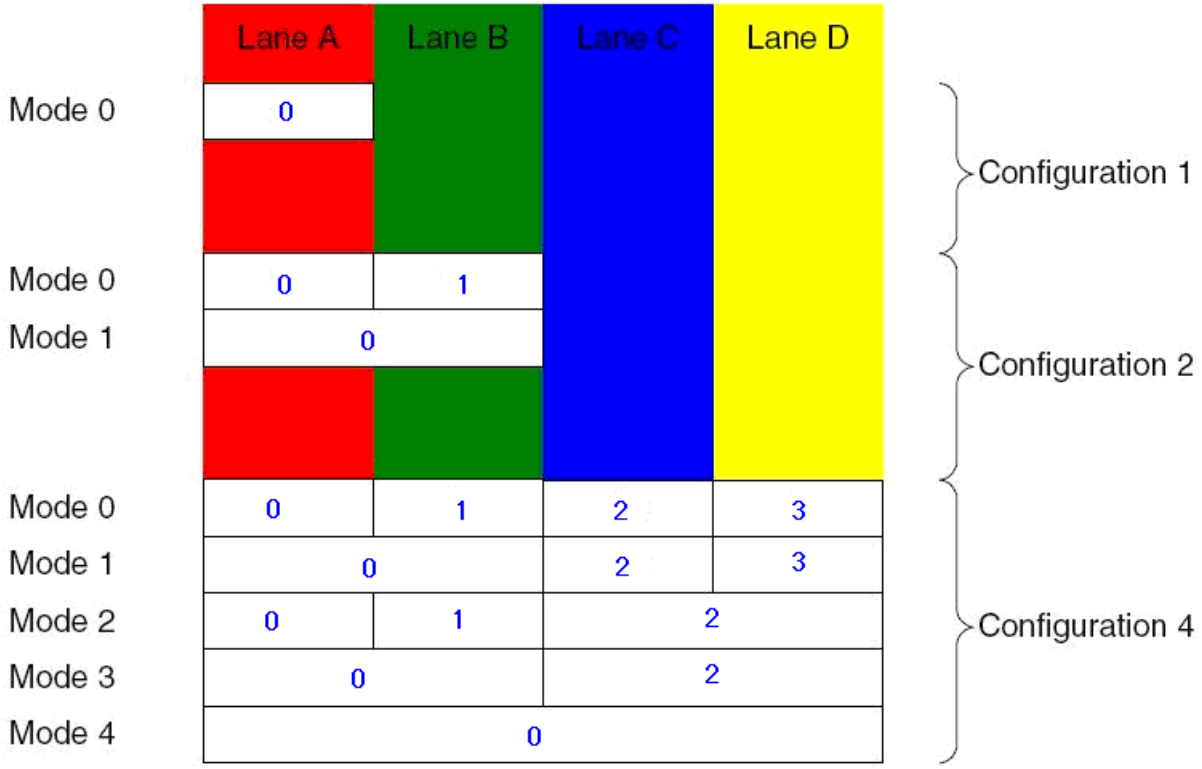

Figure 2 Port number for different configurations

When the SRIO configuration is finished, we must polling the SP_ERR_STATUS registers to check whether the port status is PORT_OK or not, and we shall polling the port based on the configuration.

2.2 Device ID configuration

KeyStone family SRIO support 16 local device ID and 8 multicast device ID. For the local device ID configuration, we should use the TLM Port Base Routing Register (n) Control Registers and TLM Port Base Routing Register (n) Pattern & Match Registers.

The TLM Port Base Routing Register (n) Pattern & Match Registers hold the 15 allowable DestIDs, the first register set is not used; the first DestID is hold in the BASE_ID CSR instead.

Below is the example code to configure the device IDs.

```
typedef struct  {
 /*PATTERN provides the 16 bits compared one-for-one against the inbound destID.*/
 Uint16 idPattern;
/*MATCH indicates which of the 16 bits of the destID is compared against PATTERN.*/
 Uint16 idMatchMask;
 /*maintenance request/reserved packets with destIDs which match this BRR are
 routed to the LLM*/
 Uint8 routeMaintenance;
} SRIO_Device_ID_Routing_Config;

/*configure SRIO device ID*/
```

```
void Keystone_SRIO_set_device_ID(
            SRIO_Device_ID_Routing_Config * device_id_routing_config,
            Uint32 uiDeviceIdNum)
{
 int i;

 /*The TLM_SP(n)_BRR_x_PATTERN_MATCH registers hold the 15 allowable DestIDs,
 note that the first register is not used.  We use the RIO_BASE_ID register
 to hold the first ID */
 srioRegs->RIO_BASE_ID= device_id_routing_config[0].idPattern|/*Large ID*/
  ((device_id_routing_config[0].idPattern&0xFF)<<16);       /*small ID*/

 uiDeviceIdNum= _min2(SRIO_MAX_DEVICEID_NUM, uiDeviceIdNum);
 for(i= 1; i<uiDeviceIdNum; i++)
 {
  /*please note, SRIO block 5~8 must be eanbled for corresponding
  RIO_TLM[0:3] taking effect*/
  srioRegs->RIO_TLM[i/4].brr[i&3].RIO_TLM_SP_BRR_CTL =
   (device_id_routing_config[i].routeMaintenance<<
    CSL_SRIO_RIO_TLM_SP_BRR_1_CTL_ROUTE_MR_TO_LLM_SHIFT)|
   (0<<CSL_SRIO_RIO_TLM_SP_BRR_1_CTL_PRIVATE_SHIFT)|
   (1<<CSL_SRIO_RIO_TLM_SP_BRR_1_CTL_ENABLE_SHIFT);

  srioRegs->RIO_TLM[i/4].brr[i&3].RIO_TLM_SP_BRR_PATTERN_MATCH =
   (device_id_routing_config[i].idPattern<<
    CSL_SRIO_RIO_TLM_SP_BRR_1_PATTERN_MATCH_PATTERN_SHIFT)|
   (device_id_routing_config[i].idMatchMask<<
    CSL_SRIO_RIO_TLM_SP_BRR_1_PATTERN_MATCH_MATCH_SHIFT);
 }
}

/*up to 16 deviceID can be setup here*/
SRIO_Device_ID_Routing_Config dsp0_device_ID_routing_config[]=
{
 /*idPattern        idMatchMask routeMaintenance*/
 {DSP0_SRIO_BASE_ID+0,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+1,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+2,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+3,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+4,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+5,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+6,   0xFFFF,     1},
 {DSP0_SRIO_BASE_ID+7,   0xFFFF,     1},
};
......
 Keystone_SRIO_set_device_ID(&dsp0_device_ID_routing_config,
  sizeof(dsp0_device_ID_routing_config)/sizeof(SRIO_Device_ID_Routing_Config));
```

The four TLM Port Base Routing Register sets belong to four port data paths, so corresponding BLK5~BLK8 must be enabled for the register sets. If only part of port data path blocks is enabled, then, fewer dest ID can be supported. For example, if only BLK5 (port 0 data path) is enabled, then, only 4 dest ID can be supported. Another example, if you use configuration 0, mode 1 (1x mode), normally, you only need enable BLK5, but if you want to support more than 4 IDs, you must enable BLK6~BLK8.

The IDs configured in the register set for one port data path can be used for other ports if the PRIVATE bits = 0.

The PATTERN and MATCH field are 16 bits. In systems that use exclusively 8-bit DeviceIDs, the upper eight bits may be removed from the comparison by appropriately setting the MATCH field. In systems which use a mixture of 8-bit and 16-bit DeviceIDs, the full 16 bits may be used for comparison, in this case, received 8-bit destIDs are prepended with eight bits of zeros by hardware for comparison purposes.

2.3 LSU (Load Store Unit) setup

In the KeyStone family SRIO, Direct IO, door bell and maintenance transfers are implemented in LSU module. There are 8 LSU. Each LSU registers set represents one transfer request, to support multiple pending DirectIO transfer requests, there are two groups of shadow LSU registers, each group have 16 shadow registers, group 0 is shared by LSU0~LSU3, group 1 is shared by LSU4~LSU7. So, we can totally have 16+16+8= 40 LSU transfer requests.

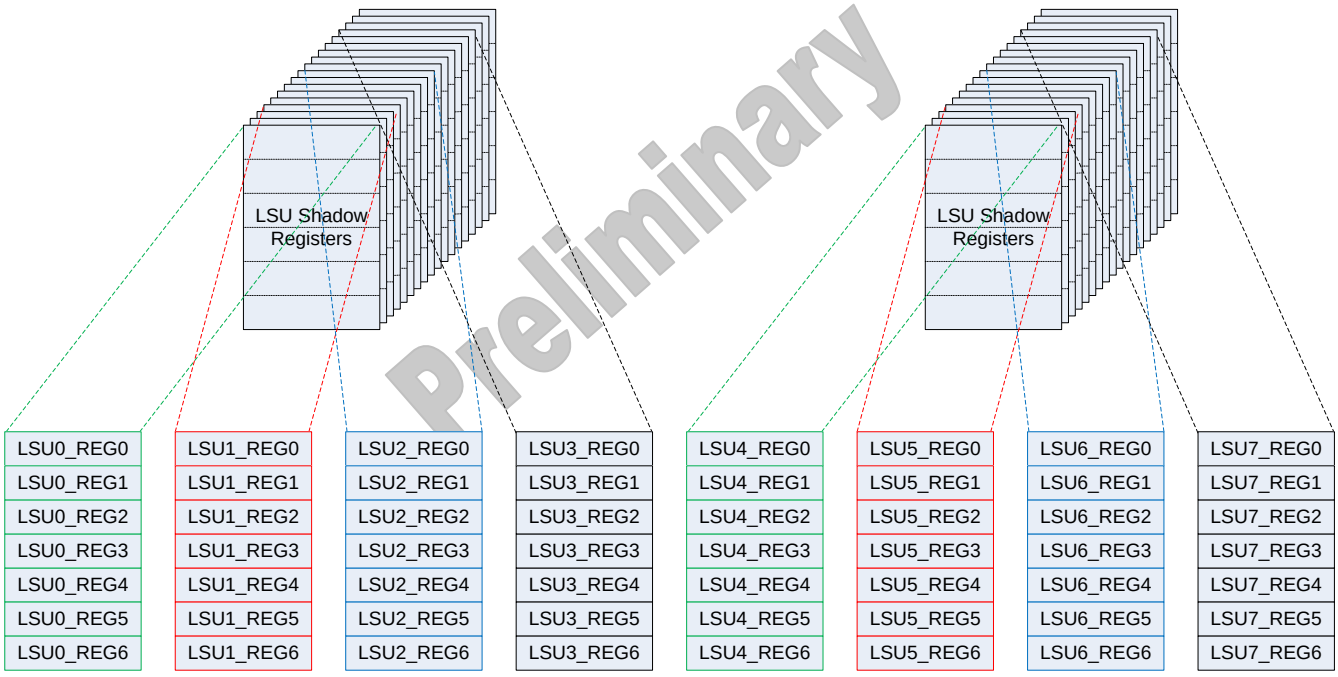

Figure 3 LSU registers

The LSU_SETUP_REG0 is used to setup number of shadow registers for each LSU. Following table shows the shadow registers configuration.

Table 1. Shadow Registers Configuration

Configuration	LSU0/LSU4	LSU1/LSU5	LSU2/LSU6	LSU3/LSU7
0x00	4	4	4	4
0x01	5	5	5	1
0x02	5	5	4	2
0x03	5	5	3	3
0x04	5	4	4	3

0x05	6	6	3	1
0x06	6	6	2	2
0x07	6	5	4	1
0x08	6	5	3	2
0x09	6	4	4	2
0x0A	6	4	3	3
0x0B	7	6	2	1
0x0C	7	5	3	1
0x0D	7	5	2	2
0x0E	7	4	4	1
0x0F	7	4	3	2
0x10	7	3	3	3
0x11	8	6	1	1
0x12	8	5	2	1
0x13	8	4	3	1
0x14	8	4	2	2
0x15	8	3	3	2
0x16	9	5	1	1
0x17	9	4	2	1
0x18	9	3	3	1
0x19	9	3	2	2

LSU_SETUP_REG1 is used to select an interrupt flag bit be set when LSU transfer complete.

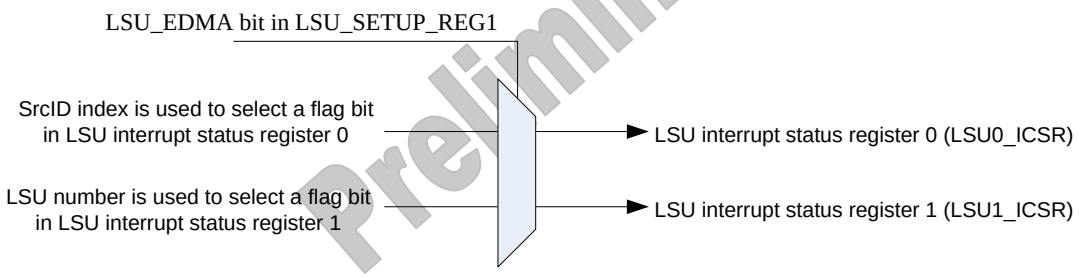

Figure 4 LSU interrupt setup

If the LSU_EDMA bit in LSU_SETUP_REG1 is clear to 0, the Source ID map index of the transfer is used to select a flag bit in LSU interrupt status register 0 (LSU0_ICSR). The source ID index is the index of the 16 device ID in TLM Port Base Routing Register (n) Pattern & Match Registers. For example, if the transfer uses the device ID 1 as source ID, then bit 1 of the LSU interrupt status register 0 (LSU0_ICSR) will be set when the transfer complete.

If the LSU_EDMA bit in LSU_SETUP_REG1 is set to 1, the LSU number of the transfer is used to select a flag bit in LSU interrupt status register 1. For example, if LSU number is 2, then bit 2 of the LSU interrupt status register 1 (LSU1_ICSR) is set.

Please note, the LSU_SETUP_REG0 and LSU_SETUP_REG1 are only programmable when the LSU is disabled while the peripheral is enabled. The following code shows the procedure.

```
typedef enum
{
            SRIO_LSU_SHADOW_REGS_SETUP_4_4_4_4,
```

```
            SRIO_LSU_SHADOW_REGS_SETUP_5_5_5_1,
            SRIO_LSU_SHADOW_REGS_SETUP_5_5_4_2,
            SRIO_LSU_SHADOW_REGS_SETUP_5_5_3_3,
            SRIO_LSU_SHADOW_REGS_SETUP_5_4_4_3,
            SRIO_LSU_SHADOW_REGS_SETUP_6_6_3_1,
            SRIO_LSU_SHADOW_REGS_SETUP_6_6_2_2,
            SRIO_LSU_SHADOW_REGS_SETUP_6_5_4_1,
            SRIO_LSU_SHADOW_REGS_SETUP_6_5_3_2,
            SRIO_LSU_SHADOW_REGS_SETUP_6_4_4_2,
            SRIO_LSU_SHADOW_REGS_SETUP_6_4_3_3,
            SRIO_LSU_SHADOW_REGS_SETUP_7_6_2_1,
            SRIO_LSU_SHADOW_REGS_SETUP_7_5_3_1,
            SRIO_LSU_SHADOW_REGS_SETUP_7_5_2_2,
            SRIO_LSU_SHADOW_REGS_SETUP_7_4_4_1,
            SRIO_LSU_SHADOW_REGS_SETUP_7_4_3_2,
            SRIO_LSU_SHADOW_REGS_SETUP_7_3_3_3,
            SRIO_LSU_SHADOW_REGS_SETUP_8_6_1_1,
            SRIO_LSU_SHADOW_REGS_SETUP_8_5_2_1,
            SRIO_LSU_SHADOW_REGS_SETUP_8_4_3_1,
            SRIO_LSU_SHADOW_REGS_SETUP_8_4_2_2,
            SRIO_LSU_SHADOW_REGS_SETUP_8_3_3_2,
            SRIO_LSU_SHADOW_REGS_SETUP_9_5_1_1,
            SRIO_LSU_SHADOW_REGS_SETUP_9_4_2_1,
            SRIO_LSU_SHADOW_REGS_SETUP_9_3_3_1,
            SRIO_LSU_SHADOW_REGS_SETUP_9_3_2_2
}SRIO_LSU_Shadow_Registers_Setup;

typedef enum {
  LSU_INT_DRIVE_BY_SRCID = 0,
  LSU_INT_DRIVE_BY_LSU_NUM
}SRIO_LSU_Interrupt_setup;

typedef struct
{
    SRIO_LSU_Shadow_Registers_Setup lsuGrp0ShadowRegsSetup;
    SRIO_LSU_Shadow_Registers_Setup lsuGrp1ShadowRegsSetup;
    SRIO_LSU_Interrupt_setup lsuIntSetup[8];
}SRIO_LSU_Cfg;
……
 /*enable globally used blocks including MMR block in SRIO*/
 Keystone_SRIO_GlobalEnable();

 /*The LSU setup registers are only programmable
 while the LSU is disabled while the peripheral is enabled.*/
 if(srio_cfg->lsu_cfg)
 {
  /*setup the shadow registers allocation between LSU*/
  srioRegs->RIO_LSU_SETUP_REG0 =
   (srio cfg->lsu cfg->lsuGrp0ShadowRegsSetup <<
    CSL_SRIO_RIO_LSU_SETUP_REG0_SHADOW_GRP0_SHIFT)|
   (srio_cfg->lsu_cfg->lsuGrp1ShadowRegsSetup <<
    CSL_SRIO_RIO_LSU_SETUP_REG0_SHADOW_GRP1_SHIFT);

  /*setup LSU interrupt based on LSU number or Source ID*/
  cfgValue = 0;
```

```
 for(i=0; i<SRIO_MAX_LSU_NUM; i++)
 {
  cfgValue |= srio_cfg->lsu_cfg->lsuIntSetup[i] << i;
 }
 srioRegs->RIO_LSU_SETUP_REG1 = cfgValue;
}

/*enable other optional blocks*/
Keystone_SRIO_enable_blocks(&srio_cfg->blockEn);
```

2.4 Message and Packet DMA setup

In KeyStone family SRIO, the Type 11 (Message) and Type 9(Data Streaming) uses Packet DMA to transmit and receive data inside the DSP.

There are 16 dedicated queues for SRIO message TX, the queue number is from 672 to 687.

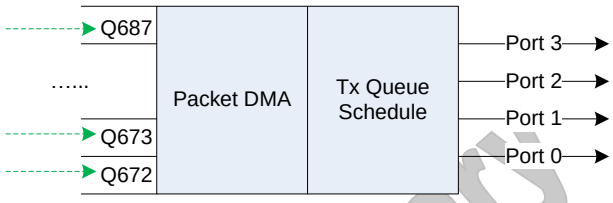

Figure 5 SRIO message TX data path

Since there are 16 TX queue and 4 ports, we need map the TX queue to ports and setup the priorities for each queue. The RIO_TX_QUEUE_SCH_INFO registers are used to configure the TX queue output port and CRF value of priority. The Packet DMA TX_CHANNEL_SCHEDULER_CONFIG_REG is used to configure the Type 11 and Type 9 output packet priority, the following code shows the configuration.

```
typedef struct
{
            Uint8 outputPort;
            Uint8 priority;
            Uint8 CRF;
}SRIO_TX_Queue_Sch_Info;

void Keystone_SRIO_TX_Queue_Cfg(
            SRIO_TX_Queue_Sch_Info * TX_Queue_Sch_Info,
            Uint32 uiNumTxQueue)
{
 int i;
 Uint32 uiMask, uiShift, uiRegIndex;

 /*For SRIO, priority 3 is highest, 0 is lowest
 For PktDMA channel, priority 0 is highest, 3 is lowest*/
 Uint32 mapSrioPriToTxChPri[4]={3,2,1,0};

 uiNumTxQueue= _min2(SRIO_PKTDMA_MAX_CH_NUM, uiNumTxQueue);
 for(i=0; i< uiNumTxQueue; i++)
 {
```

```
  uiRegIndex= i/4;
  uiShift= (i&3)*8;
  uiMask= 0xFF<<uiShift;
  srioRegs->RIO_TX_QUEUE_SCH_INFO[uiRegIndex] =
   (srioRegs->RIO_TX_QUEUE_SCH_INFO[uiRegIndex]&(~uiMask))/*clear the field*/
   |((TX_Queue_Sch_Info[i].CRF
   |(TX_Queue_Sch_Info[i].outputPort<<4))<<uiShift);

  /*PRIO field in TX_QUEUE_SCH_INFOx is read only,
  the priority information comes from the PKTDMA TX channel
  actually takes effect*/
  srioDmaTxChPriority[i] =
   mapSrioPriToTxChPri[TX_Queue_Sch_Info[i].priority];
 }
}
……
/*up to 16 TX queues can be setup here*/
SRIO_TX_Queue_Sch_Info TX_Queue_Sch_Info[]=
{
    /*outputPort*/  /*priority*/    /*CRF*/
    {0,             0,              0},
    {1,             0,              0},
    {2,             0,              0},
    {3,             0,              0},
    {0,             0,              1},
    {1,             0,              1},
    {2,             0,              1},
    {3,             0,              1},
    {0,             1,              0},
    {1,             1,              0},
    {2,             1,              0},
    {3,             1,              0},
    {0,             1,              1},
    {1,             1,              1},
    {2,             1,              1},
    {3,             1,              1}
};
......
 Keystone_SRIO_TX_Queue_Cfg(&TX_Queue_Sch_Info,
  sizeof(TX_Queue_Sch_Info)/sizeof(SRIO_TX_Queue_Sch_Info));
```

Following figures shows the SRIO type 9 and type 11 packet RX data path.

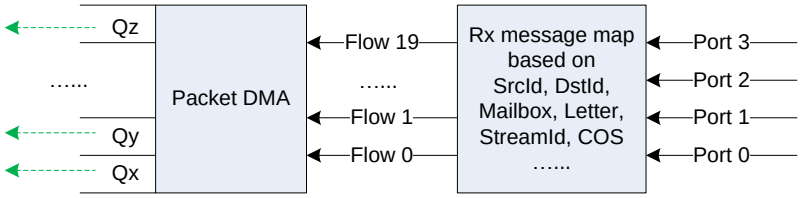

Figure 6 SRIO message RX data path

There are 20 Rx flows. The flows configuration is out of the range of this application note. Please refer to *KeyStone Architecture Multicore Navigator User Guide* (SPRUGR9) for details. The key configuration in SRIO is to map Rx message to a flow according to SrcId, DstId, Mailbox, Letter, StreamId, COS …

The Type 11 use RIO_RXU_MAP_L and RIO_RXU_MAP_H to configure the mapping according to mailbox, letter and device ID…; the Type 9 use RIO_RXU_TYPE9_MAP0, RIO_RXU_TYPE9_MAP1 and RIO_RXU_TYPE9_MAP2 to configure the mapping according to the COS, stream ID and device ID…. The Type 11 and Type 9 share the same RIO_RXU_MAP_QID for RX flow and destination queue selection. Please note, when the destination queue is set as 0x1FFF here, the RX destination queue will be determined by the configuration in RX flow. Below is the example code for message RX configuration.

```
/*
 * This structure is used to determine the
 * receive flow and queue where the packet is pushed to.
 */
typedef struct
{
    /*map matched message to flowID and optional destQuID*/
     Uint8   flowId;
     Uint16  destQuID;

    /*common fields matching*/
     Uint16  dstId;
     Uint8   dstProm;
     Uint16  srcId;
     Uint8   srcProm;
     Uint8   tt;

    /*type 11 message fields matching*/
     Uint8   mbx;
     Uint8   mbxMask;
     Uint8   ltr;
     Uint8   ltrMask;
     Uint8   segMap;

    /*type 9 message fields matching*/
     Uint8   cos;
     Uint8   cosMask;
     Uint16  streamId;
     Uint16  streamMask;

}SRIO_RX_Message_Map;

/*configure the map between message and PacketDMA flow and queue*/
void Keystone_map_SRIO_RX_message(SRIO_RX_Message_Map * srio_message_map,
            Uint32 uiNumMessageMap)
{
 int i;

 uiNumMessageMap= _min2(SRIO_MAX_MSG_MAP_ENTRY_NUM, uiNumMessageMap);
 for(i=0; i< uiNumMessageMap; i++)
 {
  srioRegs->RXU_MAP[i].RIO_RXU_MAP_L=
```

```
  (srio_message_map[i].ltrMask<<CSL_SRIO_RIO_RXU_MAP_L_LTR_MASK_SHIFT)|
  (srio_message_map[i].mbxMask<<CSL_SRIO_RIO_RXU_MAP_L_MBX_MASK_SHIFT)|
  (srio_message_map[i].ltr<<CSL_SRIO_RIO_RXU_MAP_L_LTR_SHIFT)|
  (srio_message_map[i].mbx<<CSL_SRIO_RIO_RXU_MAP_L_MBX_SHIFT)|
  (srio_message_map[i].srcId<<CSL_SRIO_RIO_RXU_MAP_L_SRCID_SHIFT);

  srioRegs->RXU_MAP[i].RIO_RXU_MAP_H=
  (srio_message_map[i].dstId<<CSL_SRIO_RIO_RXU_MAP_H_DEST_ID_SHIFT)|
  (srio_message_map[i].dstProm<<CSL_SRIO_RIO_RXU_MAP_H_DEST_PROM_SHIFT)|
  (srio_message_map[i].tt<<CSL_SRIO_RIO_RXU_MAP_H_TT_MASK)|
  (srio_message_map[i].srcProm<<CSL_SRIO_RIO_RXU_MAP_H_SRC_PROM_SHIFT)|
  (srio_message_map[i].segMap<<CSL_SRIO_RIO_RXU_MAP_H_SEG_MAP_SHIFT);

  srioRegs->RXU_TYPE9_MAP[i].RIO_RXU_TYPE9_MAP0=
  (srio_message_map[i].cosMask<<CSL_SRIO_RIO_RXU_TYPE9_MAP0_COS_MASK_SHIFT)|
  (srio_message_map[i].cos<<CSL_SRIO_RIO_RXU_TYPE9_MAP0_COS_SHIFT)|
  (srio_message_map[i].srcId<<CSL_SRIO_RIO_RXU_TYPE9_MAP0_SRCID_SHIFT);

  srioRegs->RXU_TYPE9_MAP[i].RIO_RXU_TYPE9_MAP1=
  (srio_message_map[i].dstId<<CSL_SRIO_RIO_RXU_TYPE9_MAP1_DEST_ID_SHIFT)|
  (srio_message_map[i].dstProm<<CSL_SRIO_RIO_RXU_TYPE9_MAP1_DEST_PROM_SHIFT)|
  (srio_message_map[i].tt<<CSL_SRIO_RIO_RXU_TYPE9_MAP1_TT_SHIFT)|
  (srio_message_map[i].srcProm<<CSL_SRIO_RIO_RXU_TYPE9_MAP1_SRC_PROM_SHIFT);

  srioRegs->RXU_TYPE9_MAP[i].RIO_RXU_TYPE9_MAP2=
  (srio_message_map[i].streamMask<<CSL_SRIO_RIO_RXU_TYPE9_MAP2_STRM_MASK_SHIFT)|
  (srio_message_map[i].streamId<<CSL_SRIO_RIO_RXU_TYPE9_MAP2_STRM_ID_SHIFT);

  srioRegs->RXU_MAP[i].RIO_RXU_MAP_QID =
  (srio_message_map[i].flowId<<CSL_SRIO_RIO_RXU_MAP_QID_FLOWID_SHIFT)|
  (srio_message_map[i].destQuID<<CSL_SRIO_RIO_RXU_MAP_QID_DEST_QID_SHIFT);
 }
}

/*map RX message to flow or queue according to specific fields in the message.
In this test, we only use destination device ID to determine the RX flow ID,
destination queue is configured in flow conifgurations*/
/*up to 64 map entries can be setup here*/
SRIO_RX_Message_Map DSP0_message_map[]=
{
 /*flowId*/           *destQuID*/    /*dstId*/
 {SRIO RX FLOW CORE0 LL2,0x1FFF,DSP0 SRIO BASE ID+SRIO RX FLOW CORE0 LL2,0,0,1,0,0,0,0,0,0,0,0,0},
 {SRIO_RX_FLOW_CORE1_LL2,0x1FFF,DSP0_SRIO_BASE_ID+SRIO_RX_FLOW_CORE1_LL2,0,0,1,0,0,0,0,0,0,0,0,0},
 {SRIO RX FLOW CORE2 LL2,0x1FFF,DSP0 SRIO BASE ID+SRIO RX FLOW CORE2 LL2,0,0,1,0,0,0,0,0,0,0,0,0},
 {SRIO_RX_FLOW_CORE3_LL2,0x1FFF,DSP0_SRIO_BASE_ID+SRIO_RX_FLOW_CORE3_LL2,0,0,1,0,0,0,0,0,0,0,0,0},
 {SRIO_RX_FLOW_SL2,      0x1FFF,DSP0_SRIO_BASE_ID+SRIO_RX_FLOW_SL2,      0,0,1,0,0,0,0,0,0,0,0,0},
 {SRIO_RX_FLOW_DDR,      0x1FFF,DSP0_SRIO_BASE_ID+SRIO_RX_FLOW_DDR,      0,0,1,0,0,0,0,0,1,0,0,0,0}
};
......
 Keystone_map_SRIO_RX_message(&DSP0_message_map,
  sizeof(DSP0_message_map)/sizeof(SRIO_RX_Message_Map));
```

If error happens during message packet transfer, Packet DMA will discard the packet and send the packet to a garbage queue according to the error type. RIO_GARBAGE_COLL_QID0, RIO_GARBAGE_COLL_QID1 and RIO_GARBAGE_COLL_QID2 registers are used to configure the garbage queue. Below is the example code.

```
typedef struct
{
    SRIO_RX_Message_Map * message_map;
    Uint32 uiNumMessageMap;
    SRIO_TX_Queue_Sch_Info * TX_Queue_Sch_Info;
    Uint32 uiNumTxQueue;
    Uint16 rx_size_error_garbage_Q;
    Uint16 rx_timeout_garbage_Q;
    Uint16 tx_excessive_retries_garbage_Q;
    Uint16 tx_error_garbage_Q;
    Uint16 tx_size_error_garbage_Q;
    SRIO_Datastreaming_Cfg * datastreaming_cfg;
}SRIO_Message_Cfg;

/*Setup garbage queue for error packet*/
void Keystone_SRIO_Garbage_Queue_Cfg(
            SRIO_Message_Cfg * msg_cfg)
{
 if(msg_cfg)
 {
  srioRegs->RIO_GARBAGE_COLL_QID0=
  (msg_cfg->rx_size_error_garbage_Q
   <<CSL_SRIO_RIO_GARBAGE_COLL_QID0_GARBAGE_QID_LEN_SHIFT)
  |(msg_cfg->rx_timeout_garbage_Q
   <<CSL_SRIO_RIO_GARBAGE_COLL_QID0_GARBAGE_QID_TOUT_SHIFT);

  srioRegs->RIO_GARBAGE_COLL_QID1=
  (msg_cfg->tx_excessive_retries_garbage_Q
   <<CSL_SRIO_RIO_GARBAGE_COLL_QID1_GARBAGE_QID_RETRY_SHIFT)
  |(msg_cfg->tx_error_garbage_Q
   <<CSL_SRIO_RIO_GARBAGE_COLL_QID1_GARBAGE_QID_TRANS_ERR_SHIFT);s

  srioRegs->RIO_GARBAGE_COLL_QID2=
  (msg_cfg->tx_size_error_garbage_Q
   <<CSL_SRIO_RIO_GARBAGE_COLL_QID2_GARBAGE_QID_SSIZE_SHIFT);
 }
}
```

2.5 Interrupt setup

The SRIO generates 112 interrupt events; these events are mapped to 24 interrupt outputs which can be routed to DSP core or DMA.

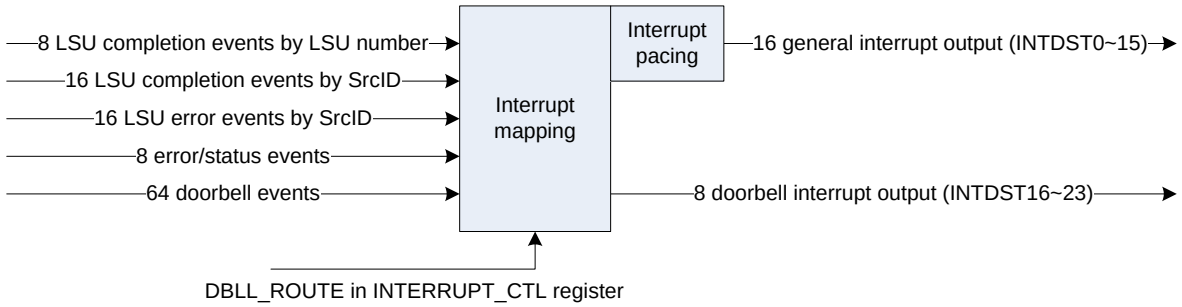

Figure 7 SRIO interrupt control

All the 112 interrupts can be mapped to INTDST0~15. The 64 doorbell events can be mapped to INTDST0~15 or INTDST16~23 depending on the INTERRUPT_CTL register, if INTERRUPT_CTL= 0, the Doorbell events are map to INTDST16~23; if INTERRUPT_CTL= 1, the Doorbell events are map to INTDST0~15.

Please note, interrupt pacing is enabled by default. Interrupt pacing requires DSP core rewrite INTDSTn_RATE_CNT register after each interrupt service to enable the next interrupt, otherwise, the interrupt will not be trigger again regardless of the internal interrupt status changes. This is the common reason user only sees one SRIO interrupt. If interrupt pacing is not desired for a particular INTDST, it can be disabled using INTDST_RATE_DIS register.

All message related interrupt or events are handled by Packet DMA and QMSS (Queue Manager SubSystem), Please refer to *KeyStone Architecture Multicore Navigator User Guide* (SPRUGR9) for details.

Below is the example code to show the SRIO interrupt setup.

```
/*Word index of the Interrupt Routing Registers*/
typedef enum
{
             DOORBELL0_ICRR1    = (0x00/4),
             DOORBELL0 ICRR2    = (0x04/4),
             DOORBELL1_ICRR1    = (0x0C/4),
             DOORBELL1_ICRR2    = (0x10/4),
             DOORBELL2_ICRR1    = (0x18/4),
             DOORBELL2_ICRR2    = (0x1C/4),
             DOORBELL3_ICRR1    = (0x24/4),
             DOORBELL3_ICRR2    = (0x28/4),
             LSU_SRCID_ICRR1    = (0x30/4),
             LSU_SRCID_ICRR2    = (0x34/4),
             LSU_SRCID_ICRR3    = (0x38/4),
             LSU_SRCID_ICRR4    = (0x3C/4),
             LSU_ICRR1          = (0x40/4),
             ERR_RST_EVNT_ICRR1 = (0x50/4),
             ERR_RST_EVNT_ICRR2 = (0x54/4),
             ERR_RST_EVNT_ICRR3 = (0x58/4)
}SRIO_ICRR_Index;

typedef enum
{
    /* SRIO interrupt source constant,
    high 16 bits is the ICRR register index,
    lower 16 bits is the offset of the field in the register*/
    DOORBELL0_0_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0000),
    DOORBELL0_1_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0004),
    DOORBELL0_2_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0008),
    DOORBELL0_3_INT  = ((DOORBELL0_ICRR1 << 16) | 0x000C),
    DOORBELL0_4_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0010),
    DOORBELL0_5_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0014),
    DOORBELL0_6_INT  = ((DOORBELL0_ICRR1 << 16) | 0x0018),
    DOORBELL0_7_INT  = ((DOORBELL0_ICRR1 << 16) | 0x001C),
    DOORBELL0_8_INT  = ((DOORBELL0_ICRR2 << 16) | 0x0000),
    DOORBELL0_9_INT  = ((DOORBELL0_ICRR2 << 16) | 0x0004),
    DOORBELL0_10_INT = ((DOORBELL0_ICRR2 << 16) | 0x0008),
    DOORBELL0_11_INT = ((DOORBELL0_ICRR2 << 16) | 0x000C),
```

```
    DOORBELL0_12_INT = ((DOORBELL0_ICRR2 << 16) | 0x0010),
    DOORBELL0_13_INT = ((DOORBELL0_ICRR2 << 16) | 0x0014),
    DOORBELL0_14_INT = ((DOORBELL0_ICRR2 << 16) | 0x0018),
    DOORBELL0_15_INT = ((DOORBELL0_ICRR2 << 16) | 0x001C),

    DOORBELL1_0_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0000),
    DOORBELL1_1_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0004),
    DOORBELL1_2_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0008),
    DOORBELL1_3_INT  = ((DOORBELL1_ICRR1 << 16) | 0x000C),
    DOORBELL1_4_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0010),
    DOORBELL1_5_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0014),
    DOORBELL1_6_INT  = ((DOORBELL1_ICRR1 << 16) | 0x0018),
    DOORBELL1_7_INT  = ((DOORBELL1_ICRR1 << 16) | 0x001C),
    DOORBELL1_8_INT  = ((DOORBELL1_ICRR2 << 16) | 0x0000),
    DOORBELL1_9_INT  = ((DOORBELL1_ICRR2 << 16) | 0x0004),
    DOORBELL1_10_INT = ((DOORBELL1_ICRR2 << 16) | 0x0008),
    DOORBELL1_11_INT = ((DOORBELL1_ICRR2 << 16) | 0x000C),
    DOORBELL1_12_INT = ((DOORBELL1_ICRR2 << 16) | 0x0010),
    DOORBELL1_13_INT = ((DOORBELL1_ICRR2 << 16) | 0x0014),
    DOORBELL1_14_INT = ((DOORBELL1_ICRR2 << 16) | 0x0018),
    DOORBELL1_15_INT = ((DOORBELL1_ICRR2 << 16) | 0x001C),

    DOORBELL2_0_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0000),
    DOORBELL2_1_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0004),
    DOORBELL2_2_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0008),
    DOORBELL2_3_INT  = ((DOORBELL2_ICRR1 << 16) | 0x000C),
    DOORBELL2_4_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0010),
    DOORBELL2_5_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0014),
    DOORBELL2_6_INT  = ((DOORBELL2_ICRR1 << 16) | 0x0018),
    DOORBELL2_7_INT  = ((DOORBELL2_ICRR1 << 16) | 0x001C),
    DOORBELL2_8_INT  = ((DOORBELL2_ICRR2 << 16) | 0x0000),
    DOORBELL2_9_INT  = ((DOORBELL2_ICRR2 << 16) | 0x0004),
    DOORBELL2_10_INT = ((DOORBELL2_ICRR2 << 16) | 0x0008),
    DOORBELL2_11_INT = ((DOORBELL2_ICRR2 << 16) | 0x000C),
    DOORBELL2_12_INT = ((DOORBELL2_ICRR2 << 16) | 0x0010),
    DOORBELL2_13_INT = ((DOORBELL2_ICRR2 << 16) | 0x0014),
    DOORBELL2_14_INT = ((DOORBELL2_ICRR2 << 16) | 0x0018),
    DOORBELL2_15_INT = ((DOORBELL2_ICRR2 << 16) | 0x001C),

    DOORBELL3_0_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0000),
    DOORBELL3_1_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0004),
    DOORBELL3_2_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0008),
    DOORBELL3_3_INT  = ((DOORBELL3_ICRR1 << 16) | 0x000C),
    DOORBELL3_4_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0010),
    DOORBELL3_5_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0014),
    DOORBELL3_6_INT  = ((DOORBELL3_ICRR1 << 16) | 0x0018),
    DOORBELL3_7_INT  = ((DOORBELL3_ICRR1 << 16) | 0x001C),
    DOORBELL3_8_INT  = ((DOORBELL3_ICRR2 << 16) | 0x0000),
    DOORBELL3 9 INT  = ((DOORBELL3 ICRR2 << 16) | 0x0004),
    DOORBELL3_10_INT = ((DOORBELL3_ICRR2 << 16) | 0x0008),
    DOORBELL3_11_INT = ((DOORBELL3_ICRR2 << 16) | 0x000C),
    DOORBELL3_12_INT = ((DOORBELL3_ICRR2 << 16) | 0x0010),
    DOORBELL3_13_INT = ((DOORBELL3_ICRR2 << 16) | 0x0014),
    DOORBELL3_14_INT = ((DOORBELL3_ICRR2 << 16) | 0x0018),
    DOORBELL3_15_INT = ((DOORBELL3_ICRR2 << 16) | 0x001C),
```

```
    SRCID0_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0000),
    SRCID1_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0004),
    SRCID2_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0008),
    SRCID3_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x000C),
    SRCID4_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0010),
    SRCID5_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0014),
    SRCID6_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x0018),
    SRCID7_Transaction_Complete_OK = ((LSU_SRCID_ICRR1 << 16) | 0x001C),

    SRCID8_Transaction_Complete_OK =  ((LSU_SRCID_ICRR2 << 16) | 0x0000),
    SRCID9_Transaction_Complete_OK =  ((LSU_SRCID_ICRR2 << 16) | 0x0004),
    SRCID10_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x0008),
    SRCID11_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x000C),
    SRCID12_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x0010),
    SRCID13_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x0014),
    SRCID14_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x0018),
    SRCID15_Transaction_Complete_OK = ((LSU_SRCID_ICRR2 << 16) | 0x001C),

    SRCID0_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0000),
    SRCID1_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0004),
    SRCID2_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0008),
    SRCID3_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x000C),
    SRCID4_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0010),
    SRCID5_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0014),
    SRCID6_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x0018),
    SRCID7_Transaction_Complete_ERR = ((LSU_SRCID_ICRR3 << 16) | 0x001C),

    SRCID8_Transaction_Complete_ERR =  ((LSU_SRCID_ICRR4 << 16) | 0x0000),
    SRCID9_Transaction_Complete_ERR =  ((LSU_SRCID_ICRR4 << 16) | 0x0004),
    SRCID10_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x0008),
    SRCID11_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x000C),
    SRCID12_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x0010),
    SRCID13_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x0014),
    SRCID14_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x0018),
    SRCID15_Transaction_Complete_ERR = ((LSU_SRCID_ICRR4 << 16) | 0x001C),

    LSU0_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0000),
    LSU1_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0004),
    LSU2_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0008),
    LSU3_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x000C),
    LSU4_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0010),
    LSU5_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0014),
    LSU6_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x0018),
    LSU7_Transaction_Complete_OK = ((LSU_ICRR1 << 16) | 0x001C),

    Multicast_event        = ((ERR_RST_EVNT_ICRR1<<16)|0),
    Port_write_In_received = ((ERR_RST_EVNT_ICRR1<<16)|4),
    Logical Layer Error    = ((ERR RST EVNT ICRR1<<16)|8),
    Port0_Error            = ((ERR_RST_EVNT_ICRR2<<16)|0),
    Port1_Error            = ((ERR_RST_EVNT_ICRR2<<16)|4),
    Port2_Error            = ((ERR_RST_EVNT_ICRR2<<16)|8),
    Port3_Error            = ((ERR_RST_EVNT_ICRR2<<16)|12),
    Device_Reset           = ((ERR_RST_EVNT_ICRR3<<16)|0)
}SRIO_Interrupt_Source;
```

```
typedef enum
{
    /* SRIO interrupt destination */
    INTDST_0 = 0,
    INTDST_1 = 1,
    INTDST_2 = 2,
    INTDST_3 = 3,
    INTDST_4 = 4,
    INTDST_5 = 5,
    INTDST_6 = 6,
    INTDST_7 = 7,
    INTDST_8 = 8,
    INTDST_9 = 9,
    INTDST_10 = 10,
    INTDST_11 = 11,
    INTDST_12 = 12,
    INTDST_13 = 13,
    INTDST_14 = 14,
    INTDST_15 = 15,
    /* doorbell only */
    INTDST_16 = 0,
    INTDST_17 = 1,
    INTDST_18 = 2,
    INTDST_19 = 3,
    INTDST_20 = 4,
    INTDST_21 = 5,
    INTDST_22 = 6,
    INTDST_23 = 7
}SRIO_Interrupt_Dest;

typedef struct
{
    SRIO_Interrupt_Source interrupt_event;
    SRIO_Interrupt_Dest INTDST_number;
}SRIO_Interrupt_Map;

typedef struct
{
    SRIO_Interrupt_Dest INTDST_number;
    Uint32 interrupt_rate_counter;
}SRIO_Interrupt_Rate;

typedef enum
{
    SRIO_DOORBELL_ROUTE_TO_DEDICATE_INT=0,
    SRIO_DOORBELL_ROUTE_TO_GENERAL_INT
}SRIO Doorbell Route Coutrol;

typedef struct
{
    SRIO_Interrupt_Map * interrupt_map;
    Uint32 uiNumInterruptMap;
    SRIO_Interrupt_Rate * interrupt_rate;
```

```
    Uint32 uiNumInterruptRateCfg;   /*number of INTDST with rate conifguration*/
    SRIO_Doorbell_Route_Coutrol doorbell_route_ctl;
}SRIO_Interrupt_Cfg;

void Keystone_SRIO_Interrupt_init(
            SRIO_Interrupt_Cfg * interrupt_cfg)
{
 Uint32 i;
 Uint32 reg, shift;
 volatile Uint32 * ICRR= (volatile Uint32 *)srioRegs->DOORBELL_ICRR;

 if(NULL == interrupt_cfg)
 return;

 /* Clear all the interrupts */
 for(i=0; i<2; i++)
 {
  srioRegs->LSU_ICSR_ICCR[i].RIO_LSU_ICCR        = 0xFFFFFFFF ;
 }
 for(i=0; i<4; i++)
 {
  srioRegs->DOORBELL_ICSR_ICCR[i].RIO_DOORBELL_ICCR = 0xFFFFFFFF;
 }
 srioRegs->RIO_ERR_RST_EVNT_ICCR = 0xFFFFFFFF;

 if(NULL != interrupt_cfg->interrupt_map)
 {
  for(i=0; i<interrupt_cfg->uiNumInterruptMap; i++)
  {
   /* Get register index for the interrupt source*/
   reg = interrupt_cfg->interrupt_map[i].interrupt_event >> 16;

   /* Get shift value for the interrupt source*/
   shift = interrupt_cfg->interrupt_map[i].interrupt_event & 0x0000FFFF;

   ICRR[reg]= (ICRR[reg]&(~(0xF<<shift))) /*clear the field*/
    |(interrupt_cfg->interrupt_map[i].INTDST_number<<shift);
  }
 }

 srioRegs->RIO_INTERRUPT_CTL = interrupt_cfg->doorbell_route_ctl;

 /*disable interrupt rate control*/
 srioRegs->RIO_INTDST_RATE_DIS= 0xFFFF;
 for(i= 0; i< 16; i++)
 {
  srioRegs->RIO_INTDST_RATE_CNT[i]= 0;
 }

 if(NULL != interrupt_cfg->interrupt_rate)
 {
  /*setup interrupt rate for specific INTDST*/
  for(i= 0; i<interrupt_cfg->uiNumInterruptRateCfg; i++)
  {
   /*enable rate control for this INTDST*/
```

```
   srioRegs->RIO_INTDST_RATE_DIS &=
    ~(1<<interrupt_cfg->interrupt_rate[i].INTDST_number);

   /*set interrupt rate counter for this INTDST*/
   srioRegs->RIO_INTDST_RATE_CNT[i]=
    interrupt_cfg->interrupt_rate[i].interrupt_rate_counter;
  }
 }

 return;
}

SRIO_Interrupt_Map interrupt_map[]=
{
 /*interrupt_event*/ /*INTDST_number*/
 {DOORBELL0_0_INT,       INTDST_16},             /*route to core 0*/
 {DOORBELL0_1_INT,       INTDST_16},             /*route to core 0*/
 {DOORBELL0_2_INT,       INTDST_16},             /*route to core 0*/
 {DOORBELL0_3_INT,       INTDST_16},             /*route to core 0*/
 {DOORBELL0_4_INT,       INTDST_16}              /*route to core 0*/
};
SRIO_Interrupt_Cfg interrupt_cfg;
......
 interrupt_cfg.interrupt_map = interrupt_map;
 interrupt_cfg.uiNumInterruptMap =
  sizeof(interrupt_map)/sizeof(SRIO_Interrupt_Map);

 /*interrupt rate control is not used in this test*/
 interrupt_cfg.interrupt_rate= NULL;
 interrupt_cfg.uiNumInterruptRateCfg= 0;

 interrupt_cfg.doorbell_route_ctl= SRIO_DOORBELL_ROUTE_TO_DEDICATE_INT;
 Keystone_SRIO_Interrupt_init(&interrupt_cfg);
```

2.6 Loopback

The KeyStone family SRIO supports 3 loopback modes; they are useful for debug and test.

2.6.1 Internal Digital loopback

The outbound packets don't forward to the SERDES, the TX data forward to the received buffer directly in digital domain.

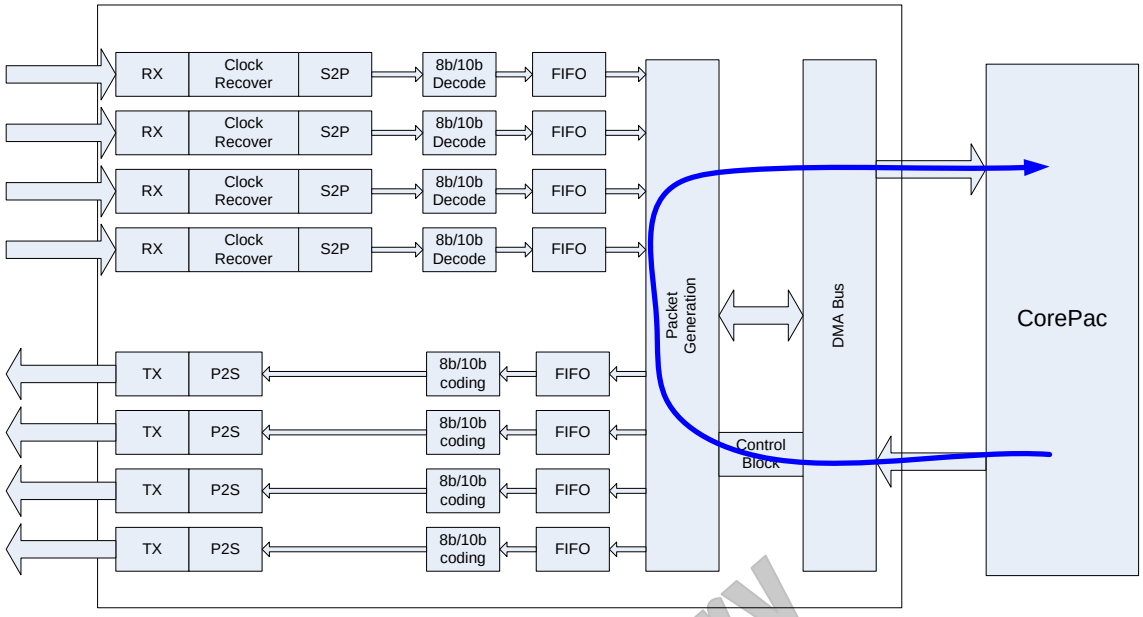

Figure 8 Digital loopback test

The Loopback[3:0] field in Peripheral Settings Control Register 1 is used to set the digital loopback test. Following code shows the configuration:

```
srioRegs->RIO_PER_SET_CNTL1 |=
       (0xF<<CSL_SRIO_RIO_PER_SET_CNTL1_LOOPBACK_SHIFT);
```

2.6.2 Internal SERDES loopback

If the SERDES loopback is configured, a differential current is passed to the receiver directly.

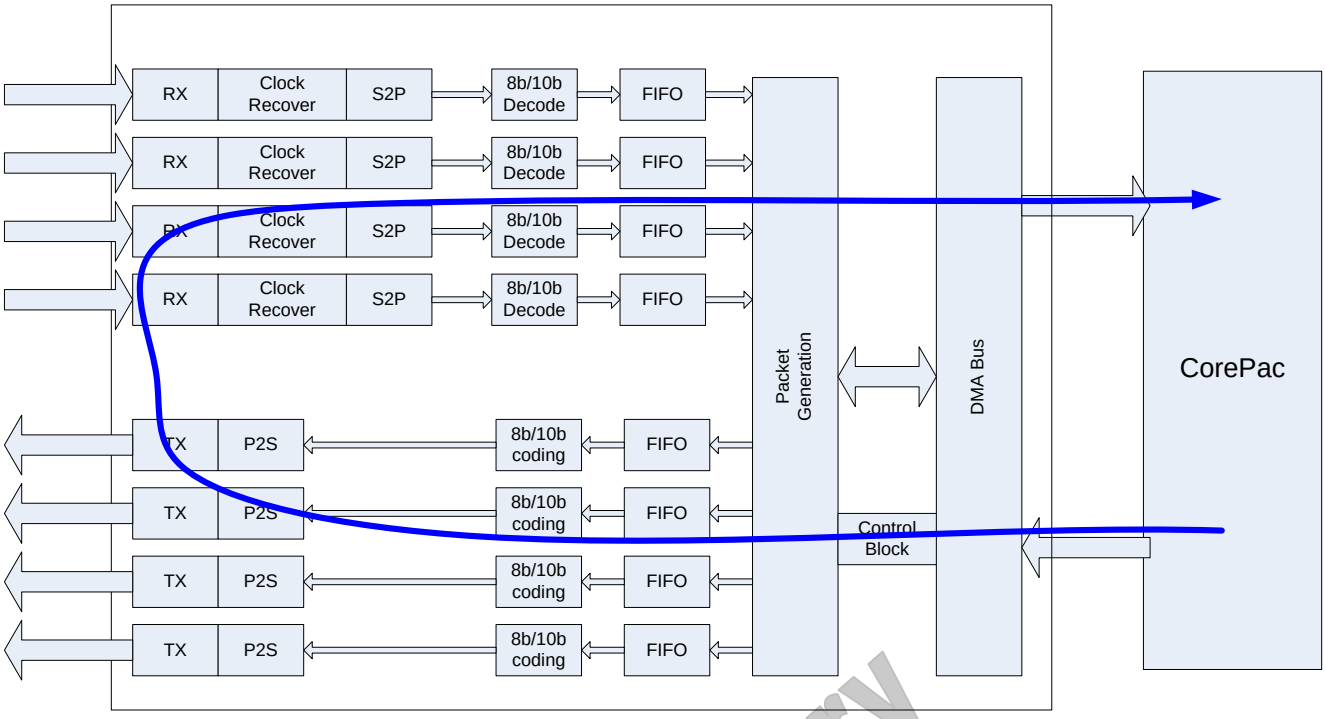

Figure 9 SERDES loopback test

The Loopback field [22:21] in SRIO SERDES Transmit Channel Configuration Registers and Loopback field [24:23] in SRIO SERDES Receive Channel Configuration Registers are used to enable the SERDES loopback test. Following code shows the configuration:

```
serdesRegs->link[i].CFGTX |=
 (serdes_cfg->linkSetup[i]->loopBack<<21);

serdesRegs->link[i].CFGRX |=
 (serdes_cfg->linkSetup[i]->loopBack<<23);
```

2.6.3 External line loopback

If external line loopback is enabled, data received on RX “line” will be directly send back to TX “line”.

Following figure show an external line loopback test. In this test, DSP 1 is configured in line loopback mode to support DSP 0 test the external “line” between two DSPs. In this mode, DSP 1 can not send or receive data.

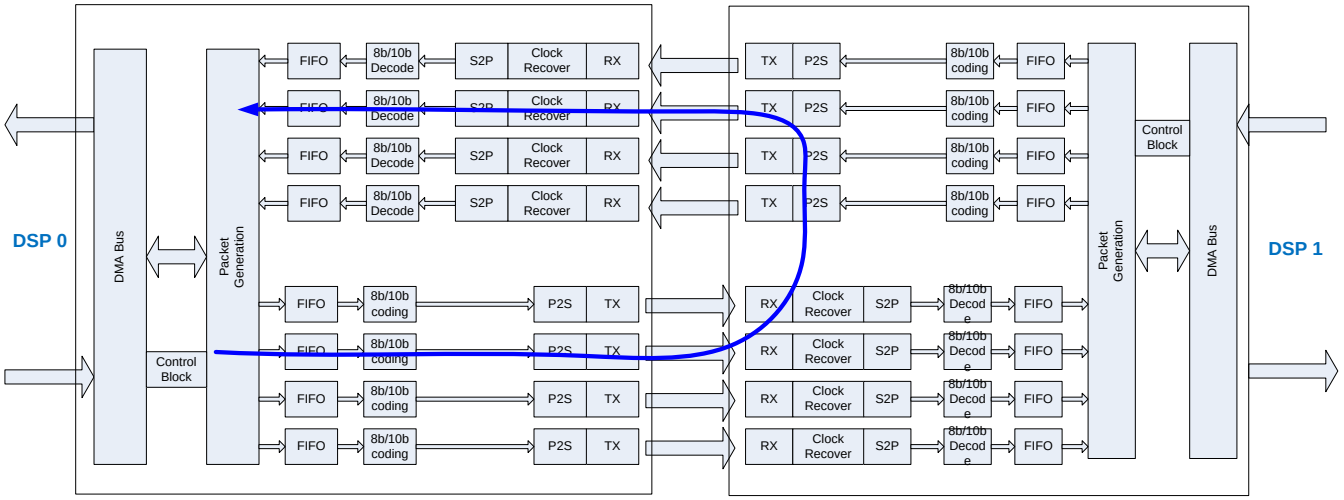

Figure 10 External line loopback test

The PLM Port(n) Implementation Specific Control Register bit 23 LLB_EN is used to control the external line loopback test. Following code shows the configuration:

```
for(i=0; i<SRIO_MAX_PORT_NUM; i++)
{
 srioRegs->RIO_PLM[i].RIO_PLM_SP_IMP_SPEC_CTL=
  (1<<CSL_SRIO_RIO_PLM_SP_IMP_SPEC_CTL_LLB_EN_SHIFT);
}
```

2.7 Packet forwarding

The KeyStone family SRIO supports the packet forwarding function, it can forward the received data from one port to any port. With this function, multiple DSPs can be connected as a daisy chain.

Following figure show an external forwarding back test. In this mode, DSP 1 can send or receive its own data when some packets are forwarding out.

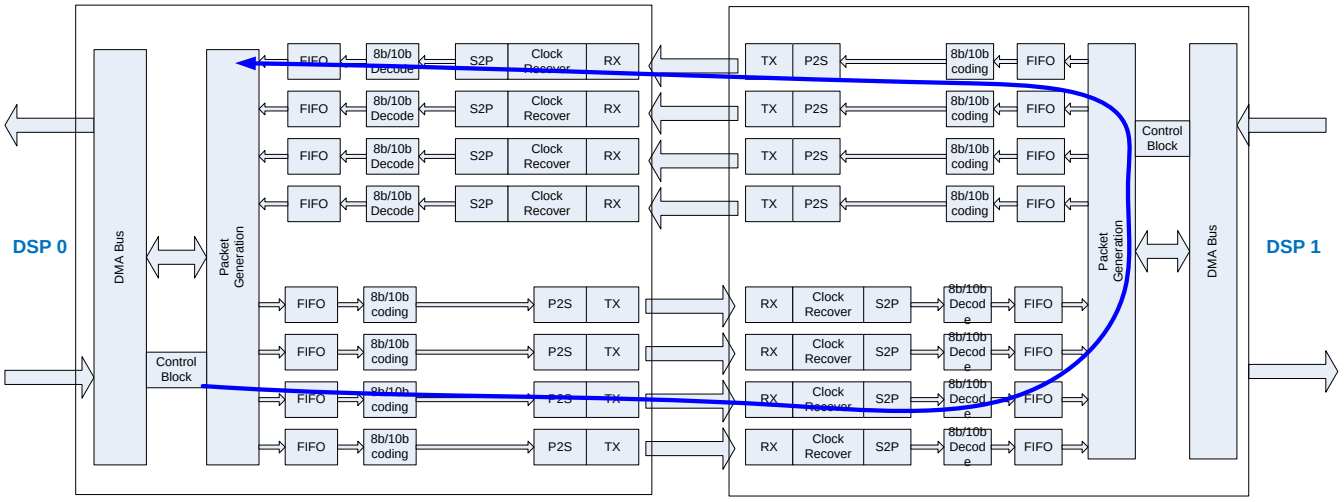

Figure 11 External forwarding back test

The KeyStone family SRIO forwarding function is handled by the MAU (Memory Access Unit) module, it only supports the NREAD, NWRITE, NWRITE_R, SWRITE and Doorbell. All other types are not supported and may cause undefined behavior, include the generation of an ERROR response.

The PF_16b_CNTL[0:7] and PF_8b_CNTL[0:7] control the packet forwarding device ID boundary and output port selection. When the inbound Device ID matches the packet forward entry setup, the packet will forward to the MAU and do further processing.

Following code shows the packet forwarding configuration:

```
typedef struct
{
 Uint16 forwardingID_up_8;          /*Upper 8b DeviceID boundary*/
 Uint16 forwardingID_lo_8;          /*Lower 8b DeviceID boundary*/

 Uint16 forwardingID_up_16;         /*Upper 16b DeviceID boundary*/
 Uint16 forwardingID_lo_16;         /*Lower 16b DeviceID boundary*/

 /*Output port number for packets whose DestID falls within the
 8b or 16b range for this table entry*/
 Uint32 outport;

}SRIO_PktForwarding_Cfg;

/* configure SRIO packet forwarding */
void Keystone_SRIO_packet_forwarding_Cfg(
            SRIO PktForwarding Cfg * PktForwardingEntry cfg,
            Uint32 pktForwardingEntryNum)
{
 int i = 0;

 pktForwardingEntryNum=  min2(SRIO MAX FORWARDING ENTRY NUM,
  pktForwardingEntryNum);
 for(i=0; i<pktForwardingEntryNum; i++)
 {
  srioRegs->PF_CNTL[i].RIO_PF_16B_CNTL =
   (PktForwardingEntry_cfg[i].forwardingID_up_16
    << CSL_SRIO_RIO_PF_16B_CNTL_DEVID_16B_UP_SHIFT)
   |(PktForwardingEntry_cfg[i].forwardingID_lo_16
    << CSL_SRIO_RIO_PF_16B_CNTL_DEVID_16B_LO_SHIFT);

  srioRegs->PF_CNTL[i].RIO_PF_8B_CNTL =
   (PktForwardingEntry_cfg[i].forwardingID_up_8
    << CSL_SRIO_RIO_PF_8B_CNTL_DEVID_8B_UP_SHIFT)
   |(PktForwardingEntry_cfg[i].forwardingID_lo_8
    << CSL_SRIO_RIO_PF_8B_CNTL_DEVID_8B_LO_SHIFT)
   |(PktForwardingEntry_cfg[i].outport
    << CSL_SRIO_RIO_PF_8B_CNTL_OUT_PORT_SHIFT);
 }
}

/*up to 8 entries can be setup here*/
SRIO_PktForwarding_Cfg DSP1_PktForwarding_Cfg[]=
{
 /*ID 8 up*/          /*ID 8 lo */        /*ID 16 up*/        /*ID 16 lo*/ /*outport*/
```

```
 {DSP0_SRIO_BASE_ID+0, DSP0_SRIO_BASE_ID+1, DSP0_SRIO_BASE_ID+0, DSP0_SRIO_BASE_ID+1, 2},
 {DSP0_SRIO_BASE_ID+2, DSP0_SRIO_BASE_ID+2, DSP0_SRIO_BASE_ID+2, DSP0_SRIO_BASE_ID+2, 2},
 {DSP0_SRIO_BASE_ID+3, DSP0_SRIO_BASE_ID+3, DSP0_SRIO_BASE_ID+3, DSP0_SRIO_BASE_ID+3, 3},
 {DSP0_SRIO_BASE_ID+4, DSP0_SRIO_BASE_ID+7, DSP0_SRIO_BASE_ID+4, DSP0_SRIO_BASE_ID+7, 3}
};
……
 Keystone_SRIO_packet_forwarding_Cfg(&DSP1_PktForwarding_Cfg,
  sizeof(DSP1_PktForwarding_Cfg)/sizeof(SRIO_PktForwarding_Cfg));
```

2.8 Rx Mode

How the SRIO handle Rx packets are controlled by three configuration bit.

Table 2. SRIO Rx Mode

Configuration bit	Description
log_tgt_id_dis (PER_SET_CNTL,Bit 27)	0b0 — Supports only packets where DestID equals BASE_ID or any of the other 15 device IDs or 8 multicast IDs. 0b1 — Supports promiscuous ID/Unicast IDs. All packets regardless of DestID are accepted by the logical layer and handled by the appropriate functional block.
mtc_tgt_id_dis (TLM_SP{0..3}_CONTROL, Bit 20)	0b0 — Only accept maintenance packets where DestID equals BASE_ID or any of the other 15 device IDs. 0b1 — All maintenance packets regardless of DestID are accepted.
tgt_id_dis (TLM_SP{0..3}_CONTROL, Bit 21)	0b0 — packet forwarding and multi-cast are not allowed. 0b1 — packet forwarding and multi-cast are allowed

Please note, if log_tgt_id_dis = 1, no matter the tgt_id_dis value, all packets will be received locally, packet forwarding will not work.

Following code shows the Rx mode configuration:

```
typedef struct  {
 Bool accept_maintenance_with_any_ID;

 /*if accept_data_with_any_ID, no packet will be forwarding*/
 Bool support_multicast_forwarding;
}SRIO_Port_RX_Mode;

typedef struct  {
 /*if accept_data_with_any_ID, no packet will be forwarding*/
 Bool accept_data_with_any_ID;
 SRIO_Port_RX_Mode port_rx_mode[4];
}SRIO RX Mode;

/* Rx Mode configuration */
void Keystone_SRIO_RxMode_Setup(SRIO_RX_Mode * rxMode)
{
 int i;

 if(rxMode)
 {
  srioRegs->RIO_PER_SET_CNTL = (srioRegs->RIO_PER_SET_CNTL&
   (~CSL_SRIO_RIO_PER_SET_CNTL_LOG_TGT_ID_DIS_MASK))|
   (rxMode->accept_data_with_any_ID
```

```
    <<CSL_SRIO_RIO_PER_SET_CNTL_LOG_TGT_ID_DIS_SHIFT);

  /*set RX mode for all ports*/
  for(i=0; i<SRIO_MAX_PORT_NUM; i++)
  {
   srioRegs->RIO_TLM[i].RIO_TLM_SP_CONTROL =
    (srioRegs->RIO_TLM[i].RIO_TLM_SP_CONTROL&
    (~(CSL_SRIO_RIO_TLM_SP_CONTROL_TGT_ID_DIS_MASK
    |CSL_SRIO_RIO_TLM_SP_CONTROL_MTC_TGT_ID_DIS_MASK)))
    |(rxMode->port_rx_mode[i].accept_maintenance_with_any_ID
     <<CSL_SRIO_RIO_TLM_SP_CONTROL_MTC_TGT_ID_DIS_SHIFT)
    |(rxMode->port_rx_mode[i].support_multicast_forwarding
     <<CSL_SRIO_RIO_TLM_SP_CONTROL_TGT_ID_DIS_SHIFT);
  }
 }
}
……
SRIO_RX_Mode rxMode;
……
 /*clear configuration structrue to make sure unused field is 0*/
 memset(&rxMode, 0, sizeof(rxMode));
 rxMode.port_rx_mode[0].support_multicast_forwarding= TRUE;
 rxMode.port_rx_mode[1].support_multicast_forwarding= TRUE;
 rxMode.port_rx_mode[2].support_multicast_forwarding= TRUE;
 rxMode.port_rx_mode[3].support_multicast_forwarding= TRUE;
 Keystone_SRIO_RxMode_Setup(&rxMode);
```

3 SRIO transfer programming

3.1 LSU transfer

Direct IO, door bell and maintenance transfers are implemented in LSU module, each LSU registers set represents one transfer request. The programming of the LSU registers is fairly straightforward. As discussed in above section, there are multiple shadow registers sets, we must poll FULL and BUSY bit in LSU_REG6 until there is an shadow register available before we programming the registers. Below is example code to programming the LSU.

```
/** @brief SRIO LSU Transfer
 * This structure is used to configure LSU module for Transfer
 */
typedef struct SRIO_LSU_Transfer
{
    Uint32  rioAddressMSB;
    Uint32  rioAddressLSB_ConfigOffset;
    Uint32  localDspAddress;
    Uint32  bytecount;
    SRIO_Packet_Type   packetType;

    Uint16  dstID;
    Uint16  doorbellInfo;
    Bool    waitLsuReady; /*if BUSY or FULL, should we wait for them?*/

    Uint8   lsuNum;     /*the LSU used for this transfer*/
    Uint8   doorbellValid;
    Uint8   intrRequest;
```

```
    Uint8   supGoodInt;
    Uint8   priority;
    Uint8   outPortID;
    Uint8   idSize;
    Uint8   srcIDMap;
    Uint8   hopCount;

    /*transaction ID returned to caller for completion check */
    Uint8   transactionID;
    /*transaction contextBit returned to caller for completion check */
    Uint8   contextBit;
}SRIO_LSU_Transfer;
/*
the maximum size per LSU transfer block is 1MB, if transfer->byteCount
larger than 1MB, this function only submit transfer for 1MB. After transfer
submission, the transfer->byteCount is decreased to indicate the remaining
bytes, the transfer addresses are increased to indicate the start address
for next possible transfer. The caller can check these to take proper action.
Transcation ID and context bit are recorded in transfer->transactionID
and transfer->contextBit for caller to check completion status.
*/
void Keystone_SRIO_LSU_transfer(SRIO_LSU_Transfer * transfer)
{
 Uint32   lsuNum= transfer->lsuNum;
 Uint32   uiByteCount;
 Uint32   uiReg6;
 unsigned long long ullRioAddress;

 /*check if LSU is busy or full*/
 uiReg6 = srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG6;
 while(uiReg6&(CSL_SRIO_RIO_LSU_REG6_BUSY_MASK
  |CSL_SRIO_RIO_LSU_REG6_FULL_MASK))
 {
  if(FALSE==transfer->waitLsuReady) /*should we wait?*/
   return;
  uiReg6 = srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG6;
 }

 /*record the transcation ID and context bit*/
 transfer->transactionID= uiReg6&
  CSL_SRIO_RIO_LSU_REG6_LTID_MASK;
 transfer->contextBit= (uiReg6&CSL_SRIO_RIO_LSU_REG6_LCB_MASK)>>
  CSL_SRIO_RIO_LSU_REG6_LCB_SHIFT;

 /*the maximum size per LSU transfer block is 1MB, if transfer->byteCount
 larger than 1MB, this function only submit transfer for 1MB*/
 if(transfer->bytecount>=1024*1024)
  uiByteCount= 1024*1024;
 else
  uiByteCount= transfer->bytecount;

 /*submit transfer*/
 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG0= transfer->rioAddressMSB;
 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG1= transfer->rioAddressLSB_ConfigOffset;
 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG2= transfer->localDspAddress;
```

```
 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG3=
  ((uiByteCount&0xFFFFF)<<CSL_SRIO_RIO_LSU_REG3_BYTE_COUNT_SHIFT)|
  (transfer->doorbellValid<<CSL_SRIO_RIO_LSU_REG3_DRBLL_VALUE_SHIFT);

 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG4=
  transfer->intrRequest<<CSL_SRIO_RIO_LSU_REG4_INT_ REQ_SHIFT)|
  (transfer->srcIDMap<<CSL_SRIO_RIO_LSU_REG4_SRCID_MAP_SHIFT)|
  (transfer->supGoodInt<<CSL_SRIO_RIO_LSU_REG4_SUP_GINT_SHIFT)|
  (transfer->dstID<<CSL_SRIO_RIO_LSU_REG4_DESTID_SHIFT)|
  (transfer->idSize<<CSL_SRIO_RIO_LSU_REG4_ID_SIZE_SHIFT)|
  (0<<CSL_SRIO_RIO_LSU_REG4_XAMBS_SHIFT)|
  (transfer->priority<<CSL_SRIO_RIO_LSU_REG4_PRIORITY_SHIFT)|
  (transfer->outPortID<<CSL_SRIO_RIO_LSU_REG4_OUTPORTID_SHIFT);

 srioRegs->LSU_CMD[lsuNum].RIO_LSU_REG5= transfer->packetType|
  (transfer->hopCount<<CSL_SRIO_RIO_LSU_REG5_HOP_COUNT_SHIFT)|
  (transfer->doorbellInfo<<CSL_SRIO_RIO_LSU_REG5_DRBLL_INFO_SHIFT);

 /*update the byte count and address after submit the transfer*/
 transfer->bytecount -= uiByteCount;
 transfer->localDspAddress += uiByteCount;
 ullRioAddress= _itoll(transfer->rioAddressMSB,
 transfer->rioAddressLSB_ConfigOffset)+ uiByteCount;
 transfer->rioAddressLSB_ConfigOffset= _loll(ullRioAddress);
 transfer->rioAddressMSB= _hill(ullRioAddress);
}
```

The LSU transfer completion is checked through LSU_STAT_REG registers. The transaction ID read from LSU_REG6 is used to determine the index of the completion code in LSU_STAT_REG registers; the context bit read from LSU_REG6 is used to determine the validation of the completion code. The context bit toggles for each transaction, for example, for the first transfer with transaction ID = 2, the context bit = 1; then, for the next transfer with transaction ID = 2, the context bit = 0. So, only after the context bit in LSU_STAT_REG register for transaction ID = 2 toggles from 1 to 0, it indicates the second transfer complete, and the completion code is valid. Below is the example code for LSU completion checking.

```
/* LSU states are in 6 LSU state registers as below:
-------------------------------------------------------------------------------------------------------
              |31     28|        24|        20|        16|        12|         8|         4|         0|
--------------|---------|----------|----------|----------|----------|----------|----------|----------|
LSU_STAT_REG0  |Lsu0_Stat7| Lsu0_Stat6| Lsu0_Stat5| Lsu0_Stat4| Lsu0_Stat3| Lsu0_Stat2| Lsu0_Stat1| Lsu0_Stat0|
LSU_STAT_REG1  |Lsu2_Stat0| Lsu1_Stat5| Lsu1_Stat4| Lsu1_Stat3| Lsu1_Stat2| Lsu1_Stat1| Lsu1_Stat0| Lsu0_Stat8|
LSU_STAT_REG2  |Lsu3_Stat3| Lsu3_Stat2| Lsu3_Stat1| Lsu3_Stat0| Lsu2_Stat4| Lsu2_Stat3| Lsu2_Stat2| Lsu2_Stat1|
LSU_STAT_REG3  |Lsu4_Stat7| Lsu4_Stat6| Lsu4_Stat5| Lsu4_Stat4| Lsu4_Stat3| Lsu4_Stat2| Lsu4_Stat1| Lsu4_Stat0|
LSU_STAT_REG4  |Lsu6_Stat0| Lsu5_Stat5| Lsu5_Stat4| Lsu5_Stat3| Lsu5_Stat2| Lsu5_Stat1| Lsu5_Stat0| Lsu4_Stat8|
LSU_STAT_REG5  |Lsu7_Stat3| Lsu7_Stat2| Lsu7_Stat1| Lsu7_Stat0| Lsu6_Stat4| Lsu6_Stat3| Lsu6_Stat2| Lsu6_Stat1|
-------------------------------------------------------------------------------------------------------
below is a table to indicate the index of state for a transaction of a LSU
in the LSU state registers
*/
Uint8 LSU_state_index_table[SRIO_MAX_LSU_NUM][SRIO_LSU0_4_MAX_SHADOW_REG_SET]=
{
            /*LSU0*/ {0, 1, 2, 3, 4, 5, 6, 7, 8},
            /*LSU1*/ {9, 10, 11, 12, 13, 14},
            /*LSU2*/ {15, 16, 17, 18, 19},
            /*LSU3*/ {20, 21, 22, 23},
            /*LSU4*/ {24, 25, 26, 27, 28, 29, 30, 31, 32},
```

```
            /*LSU5*/ {33, 34, 35, 36, 37, 38},
            /*LSU6*/ {39, 40, 41, 42, 43},
            /*LSU7*/ {44, 45, 46, 47}
};
/*wait LSU transfer completion, return completion code*/
Uint32 Keystone_SRIO_wait_LSU_completion(Uint32 lsuNum,
            Uint32 transactionID, Uint32 contextBit)
{
 Uint32 uiStateIndex= LSU_state_index_table[lsuNum][transactionID];
 Uint32 uiCompletionCode;

 do
 {
  uiCompletionCode=(srioRegs->LSU_STAT_REG[uiStateIndex/8]>>
   ((uiStateIndex&7)*4))&0xF;
 }
 while((uiCompletionCode&1) != contextBit);

 return (uiCompletionCode>>1);
}
```

Below is example code for Direct IO transfer.

```
typedef enum
{
            SRIO_PKT_TYPE_NREAD     = 0x24,
            SRIO_PKT_TYPE_NWRITE    = 0x54,
            SRIO_PKT_TYPE_NWRITE_R  = 0x55,
            SRIO_PKT_TYPE_SWRITE    = 0x60,
            SRIO_PKT_TYPE_MTN_READ  = 0x80,
            SRIO_PKT_TYPE_MTN_WRITE = 0x81,
            SRIO_PKT_TYPE9_STREAM   = 0x90,
            SRIO_PKT_TYPE_DOORBELL  = 0xA0,
            SRIO_PKT_TYPE11_MESSAGE = 0xB0
}SRIO_Packet_Type;

//SRIO DirectIO operation
Int32 Keystone_SRIO_DirectIO(Uint32 uiLocalAddress, Uint32 uiRemoteAddress,
            Uint32 uiDestID, Uint32 uiByteCount, Uint32 uiPort, Uint32 uiLSU_No,
            SRIO_Packet_Type packetType)
{
            SRIO_LSU_Transfer lsuTransfer;

    lsuTransfer.rioAddressMSB=0;
    lsuTransfer.rioAddressLSB_ConfigOffset= uiRemoteAddress;
    lsuTransfer.localDspAddress= uiLocalAddress;
    lsuTransfer.bytecount= uiByteCount;
    lsuTransfer.packetType= packetType;
    lsuTransfer.dstID= uiDestID;
    lsuTransfer.doorbellInfo= 0;
    lsuTransfer.waitLsuReady= 1;
    lsuTransfer.lsuNum= uiLSU_No;
    lsuTransfer.doorbellValid = 0;
    lsuTransfer.intrRequest = 0;
    lsuTransfer.supGoodInt = 0;
    lsuTransfer.priority = 0;
```

```
    lsuTransfer.outPortID = uiPort;
    lsuTransfer.idSize = 0;
    lsuTransfer.srcIDMap = 0;
    lsuTransfer.hopCount = 0;

    Keystone_SRIO_LSU_transfer(&lsuTransfer);

    return Keystone_SRIO_wait_LSU_completion(uiLSU_No,
                lsuTransfer.transactionID, lsuTransfer.contextBit);
}
```

3.2 Message transfer

The KeyStone family SRIO Type 11 and Type 9 packets are handled by QMSS and Packet DMA. Please refer to *KeyStone Architecture Multicore Navigator User Guide* (SPRUGR9) for details.

For data receive, if the message Rx map and Rx flow setup properly, once the data is received, the Packet DMA will push the packet to the Rx destination queue, then the application can get the packet from the Rx queue; For data transmit, application just need push the packet to 1 of the 16 TX queue, Packet DMA and SRIO peripheral will transmit the packet according to the TX queue schedule configuration.

The special points for SRIO are the protocol specific fields in the packet descriptor. They are defined as below.

```
/**************************************************************************/
/* Define the bit and word layouts for the SRIO Type 9 message TX descriptor*/
/**************************************************************************/
#ifdef _BIG_ENDIAN
typedef struct
{
  /* word 0 */
  Uint32  SRC_ID    : 16;  //Source Node Id
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.

  /* word 1 */
  Uint32  StreamID  : 16;           //Stream ID for the transaction
  Uint32  reserved0 : 5;
  Uint32  TT        : 2;            //RapidIO tt field, 8 or 16bit IDs
  Uint32  reserved1 : 1;
  Uint32  COS       : 8;            //Class of Service

} SRIO_Type9_Message_TX_Descriptor;
#else
typedef struct
{
  /* word 0 */
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.
  Uint32  SRC_ID    : 16;  //Source Node Id

  /* word 1 */
  Uint32  COS       : 8;            //Class of Service
```

```
  Uint32  reserved1 : 1;
  Uint32  TT        : 2;            // RapidIO tt field, 8 or 16bit IDs
  Uint32  reserved0 : 5;
  Uint32  StreamID  : 16;           //Stream ID for the transaction

} SRIO_Type9_Message_TX_Descriptor;
#endif

/*****************************************************************************/
/* Define the bit and word layouts for the SRIO Type 9 message RX descriptor*/
/*****************************************************************************/
#ifdef _BIG_ENDIAN
typedef struct
{
  /* word 0 */
  Uint32  SRC_ID    : 16;  //Source Node Id
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.

  /* word 1 */
  Uint32  StreamID  : 16;           //Stream ID for the transaction
  Uint32  reserved0 : 1;
  Uint32  PRI       : 4;            //Message Priority (VC||PRIO||CRF)
  Uint32  T         : 1;            //This is TT [0] bit only.
  Uint32  CC        : 2;            //Completion Code
  Uint32  COS       : 8;            //Class of Service

} SRIO_Type9_Message_RX_Descriptor;
#else
typedef struct
{
  /* word 0 */
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.
  Uint32  SRC_ID    : 16;  //Source Node Id

  /* word 1 */
  Uint32  COS       : 8;            //Class of Service
  Uint32  CC        : 2;            //Completion Code
  Uint32  T         : 1;            //This is TT [0] bit only.
  Uint32  PRI       : 4;            //Message Priority (VC||PRIO||CRF)
  Uint32  reserved0 : 1;
  Uint32  StreamID  : 16;           //Stream ID for the transaction

} SRIO_Type9_Message_RX_Descriptor;
#endif

/*****************************************************************************/
/* Define the bit and word layouts for the SRIO Type 11 message RX descriptor*/
/*****************************************************************************/
#ifdef _BIG_ENDIAN
typedef struct
{
  /* word 0 */
  Uint32  SRC_ID    : 16;  //Source Node Id
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.
```

```
  /* word 1 */
  Uint32  reserved0 : 15;
  Uint32  CC        : 2;            //Completion Code
  Uint32  PRI       : 4;            //Message Priority (VC||PRIO||CRF)
  Uint32  TT        : 2;            //RapidIO tt field, 8 or 16bit DeviceIDs
  Uint32  LTR       : 3;            //Destination Letter
  Uint32  MailBox   : 6;            //Destination Mailbox

} SRIO_Type11_Message_RX_Descriptor;
#else
typedef struct
{
  /* word 0 */
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.
  Uint32  SRC_ID    : 16;  //Source Node Id

  /* word 1 */
  Uint32  MailBox   : 6;            //Destination Mailbox
  Uint32  LTR       : 3;            //Destination Letter
  Uint32  TT        : 2;            //RapidIO tt field, 8 or 16bit DeviceIDs
  Uint32  PRI       : 4;            //Message Priority (VC||PRIO||CRF)
  Uint32  CC        : 2;            //Completion Code
  Uint32  reserved0 : 15;

} SRIO_Type11_Message_RX_Descriptor;
#endif

/**************************************************************************/
/* Define the bit and word layouts for the SRIO Type 11 message TX descriptor*/
/**************************************************************************/
#ifdef _BIG_ENDIAN
typedef struct
{
  /* word 0 */
  Uint32  SRC_ID      : 16;  //Source Node Id
  Uint32  Dest_ID     : 16; //Destination Node ID the message was sent to.

  /* word 1 */
  Uint32  reserved0   : 5;
  Uint32  Retry_Count : 6;          //Message Retry Count
  Uint32  SSIZE       : 4;          //standard message payload size.
  Uint32  reserved1   : 6;
  Uint32  TT          : 2;          //RapidIO tt field, 8 or 16bit DeviceIDs
  Uint32  LTR         : 3;          //Destination Letter
  Uint32  MailBox     : 6;          //Destination Mailbox

} SRIO_Type11_Message_TX_Descriptor;
#else
typedef struct
{
  /* word 0 */
  Uint32  Dest_ID   : 16; //Destination Node ID the message was sent to.
  Uint32  SRC_ID    : 16;  //Source Node Id

  /* word 1 */
```

```
  Uint32  MailBox     : 6;          //Destination Mailbox
  Uint32  LTR         : 3;          //Destination Letter
  Uint32  TT          : 2;          //RapidIO tt field, 8 or 16bit DeviceIDs
  Uint32  reserved1   : 6;
  Uint32  SSIZE       : 4;          //standard message payload size.
  Uint32  Retry_Count : 6;          //Message Retry Count
  Uint32  reserved0   : 5;

} SRIO_Type11_Message_TX_Descriptor;
#endif
```

Another point we need pay attention to is, for data transmit, the host packet descriptor field Packet_Type must be written correctly, it is used to identify the SRIO transaction type, the Packet_Type should be set as 30 for the Type 9 is selected; if the Packet_Type is 31, the Type 11 is selected. The application only need to pop a descriptor from the FDQ and fill the PS filed in the descriptor, then push the descriptor to the related Tx queue.

Blow is example code to fill the TX packet descriptor.

```
typedef enum
{
            SRIO_TYPE9_CPPI_PACKET  = 30,
            SRIO_TYPE11_CPPI_PACKET = 31
}SRIO_CPPI_Packet_Type;

//Build SRIO type11 message descriptor
void Keystone_SRIO_Build_Type11_Msg_Desc(
            HostPacketDescriptor * hostDescriptor, Uint32 uiSrcID, Uint32 uiDestID,
            Uint32 uiByteCount, Uint32 uiMailBox, Uint32 uiLetter)
{
 SRIO_Type11_Message_TX_Descriptor * type11MsgTxDesc;

 hostDescriptor->packet type = SRIO TYPE11 CPPI PACKET;
 hostDescriptor->packet_length= uiByteCount;
 hostDescriptor->buffer_len= uiByteCount;
 hostDescriptor->psv_word_count= 2; //SRIO uses 2 Protocol Specific words

 type11MsgTxDesc= (SRIO_Type11_Message_TX_Descriptor *)
    (((Uint32)hostDescriptor)+32);
 type11MsgTxDesc->Dest_ID = uiDestID;
 type11MsgTxDesc->SRC_ID = uiSrcID;
 type11MsgTxDesc->Retry_Count= 1;
 type11MsgTxDesc->SSIZE= SRIO_SSIZE_256_BYTES;
 type11MsgTxDesc->TT = 0;
 type11MsgTxDesc->LTR= uiLetter;
 type11MsgTxDesc->MailBox= uiMailBox;
}

//Build SRIO type9 data stream message Descriptror
void Keystone_SRIO_Build_Type9_Msg_Desc(
            HostPacketDescriptor * hostDescriptor, Uint32 uiSrcID, Uint32 uiDestID,
            Uint32 uiByteCount, Uint32 uiStreamID, Uint32 uiCOS)
{
 SRIO_Type9_Message_TX_Descriptor * type9MsgTxDesc;

 hostDescriptor->packet_type = SRIO_TYPE9_CPPI_PACKET;
```

```
 hostDescriptor->packet_length= uiByteCount;
 hostDescriptor->buffer_len= uiByteCount;
 hostDescriptor->psv_word_count= 2; //SRIO uses 2 Protocol Specific words

 type9MsgTxDesc= (SRIO_Type9_Message_TX_Descriptor *)
    (((Uint32)hostDescriptor)+32);
 type9MsgTxDesc->Dest_ID = uiDestID;
 type9MsgTxDesc->SRC_ID = uiSrcID;
 type9MsgTxDesc->StreamID = uiStreamID;
 type9MsgTxDesc->TT = 0;
 type9MsgTxDesc->COS = uiCOS;
}
```

Please note, the Type 9 packet can support up to 64K bytes payload, and the Type 11 only support 4K bytes payload.

4 Other SRIO programming considerations

4.1 Match ACKID

According to the SRIO specification, ACKID (Acknowledgement ID) is increased for each transaction in both sides of the SRIO interface, if the ACKID of the two sides is different, no data will be accepted.

There are many cases may make the ACKID different, the typical case is, one side of the SRIO interface is reset, but we do not want to reset the other side. Especially, when we debug the SRIO transfer between two DSPs, we may frequently rerun one of the DSP, but do not want to affect the other DSP. For example, when we debug the external forwarding back test described in section 2.7, we may run DSP1 firstly, and then run DSP 0, stop DSP 0, modify code in DSP 0, and then rerun DSP0…

For these cases, we should match the ACKID between the two sides of the SRIO interface throughput software, otherwise, we must reset both sides of the interface to make it work.

Please find the Match ACKID procedure as below:

1. Write 4 to the register RIO_SP_LM_REQ send a "restart-from-error" command, request the inbound ACK_ID of the other side;
2. Check the RIO_SP_LM_RESP, polling the response valid bit till the response has been received correctly;
3. Set the local OUTBOUND_ACKID to be same as the response ACKID;
4. Set the remote outbound ACK_ID through maintenance packet to write the RIO_ACKID_STATUS registers;
5. Read back remote outbound ACK_ID through maintenance packet to verify it match local inbound ACKID

Below is the example code for ACKID matching.

```
/* Make the ACK_ID of both sides match */
```

```
void Keystone_SRIO_match_ACK_ID(Uint32 uiLocalPort,
            Uint32 uiDestID, Uint32 uiRemotePort)
{
 Uint32 uiMaintenanceValue, uiResult;
 Uint32 uiLocal_In_ACK_ID, uiRemote_In_ACK_ID, uiRemote_out_ACK_ID;

 //send a "restart-from-error" commond, request the ACK_ID of the other side
 srioRegs->RIO_SP[uiLocalPort].RIO_SP_LM_REQ=4;

 //wait for link response
 while(0==(srioRegs->RIO_SP[uiLocalPort].RIO_SP_LM_RESP>>
  CSL_SRIO_RIO_SP_LM_RESP_RESP_VALID_SHIFT))
   asm(" nop 5");

 uiRemote_In_ACK_ID= (srioRegs->RIO_SP[uiLocalPort].RIO_SP_LM_RESP&
  CSL_SRIO_RIO_SP_LM_RESP_ACK_ID_STAT_MASK)>>
  CSL_SRIO_RIO_SP_LM_RESP_ACK_ID_STAT_SHIFT;

 //Set the local OUTBOUND_ACKID to be same as the responsed ACKID
 srioRegs->RIO_SP[uiLocalPort].RIO_SP_ACKID_STAT= uiRemote_In_ACK_ID;

 do
 {
  //set the remote OUTBOUND_ACKID to be same as local INBOUND_ACKID
  uiLocal_In_ACK_ID= (srioRegs->RIO_SP[uiLocalPort].RIO_SP_ACKID_STAT&
   CSL_SRIO_RIO_SP_ACKID_STAT_INB_ACKID_MASK)>>
   CSL_SRIO_RIO_SP_ACKID_STAT_INB_ACKID_SHIFT;

  uiMaintenanceValue= ((uiRemote_In_ACK_ID+1)<<
   CSL_SRIO_RIO_SP_ACKID_STAT_INB_ACKID_SHIFT)|uiLocal_In_ACK_ID;

  //set the remote ACK_ID through maintenance packet
  uiResult= Keystone_SRIO_Maintenance(uiLocalPort, uiLocalPort, uiDestID,
   0x148+(0x20*uiRemotePort), GLOBAL_ADDR(&uiMaintenanceValue),
   SRIO_PKT_TYPE_MTN_WRITE);

  if(uiResult)          //fail
   continue;

  //readback the remote ID
  uiResult= Keystone_SRIO_Maintenance(uiLocalPort, uiLocalPort,
   uiDestID, 0x148+(0x20*uiRemotePort), GLOBAL_ADDR(&uiMaintenanceValue),
   SRIO_PKT_TYPE_MTN_READ);
  uiRemote_out_ACK_ID= uiMaintenanceValue&
  CSL_SRIO_RIO_SP_ACKID_STAT_OUTB_ACKID_MASK;
 }while(uiResult|(uiLocal_In_ACK_ID+1 != uiRemote_out_ACK_ID));
}
```

4.2 Soft reset

The KeyStone SRIO can be reset by the software, the application can implement the software reset with the following steps:

1. Set Teardown bit in RX/TX channel global configure registers
2. Flush all LSU transfer, and waiting for completion.
3. Disable the PEREN bit of RIO_PCR register to stop all new logical layer transactions
4. Disable all the SRIO block with BLK_EN and GBL_EN;
5. Disable Serdes by writing 0 to the SRIO SERDES configuration registers;
6. Disable the SRIO through the PSC module;

Below the example code for SRIO reset by software.

```
/*soft shutdown and reset SRIO*/
void Keystone_SRIO_soft_reset()
{
 int i, j, k;

 /*shut down TXU/RXU transaction*/
 for(i=0; i<SRIO_PKTDMA_MAX_CH_NUM; i++)
 {
  KeyStone_pktDma_RxCh_teardown(srioDmaRxChCfgRegs, i);
  KeyStone_pktDma_TxCh_teardown(srioDmaTxChCfgRegs, i);
 }

 for(i= 0; i<SRIO_MAX_LSU_NUM ; i++)
 {
  /*flash LSU transfer for all Source ID*/
  for(j=0; j< SRIO_MAX_DEVICEID_NUM; j++)
  {
   srioRegs->LSU_CMD[i].RIO_LSU_REG6 =
    CSL_SRIO_RIO_LSU_REG6_FLUSH_MASK| /*flash*/
    (j<<CSL_SRIO_RIO_LSU_REG6_SCRID_MAP_SHIFT);

   /*This can take more than one cycle to do the flush. wait for a while*/
   for(k=0; k< 100; k++)
    asm(" nop 5");
  }
 }

 /*disable the PEREN bit of the PCR register to stop all
 new logical layer transactions.*/
 srioRegs->RIO_PCR &= (~CSL_SRIO_RIO_PCR_PEREN_MASK);

 /*Wait one second to finish any current DMA transfer.*/
 for(i=0; i< 100000000; i++)
  asm(" nop 5");

 //reset all logic blocks in SRIO
 Keystone_SRIO_disable_all_blocks();

 //disable Serdes
 Keystone_Serdes_disable(srioSerdesRegs, 4);

 //disable SRIO through PSC
 Keystone_disable_PSC_module(CSL_PSC_PD_SRIO, CSL_PSC_LPSC_SRIO);
```

```
 Keystone_disable_PSC_Power_Domain(CSL_PSC_PD_SRIO);
}
```

When the above procedure is finished, the SRIO can be enabled and initialized again. To rerun the SRIO program without reset the board, we normally call above function as the first step of SRIO initialization, below is the example:

```
void Keystone_SRIO_Init(SRIO_Config * srio_cfg)
{
 if(srioRegs->RIO_PCR&CSL_SRIO_RIO_PCR_PEREN_MASK)
  Keystone_SRIO_soft_reset();        //soft reset SRIO if it is already enabled

 //enable SRIO power and clock domain
 Keystone_enable_PSC_module(CSL_PSC_PD_SRIO, CSL_PSC_LPSC_SRIO);
……
```

4.3 Tricks/Tips

4.3.1 RX buffer allocation based on priority

In SRIO physical layer, each SRIO port has memory, which is maintained as a pool of linked data nodes (similar to a free space linked list in linked memory allocation). Each data node stores 32 bytes of data. Both egress PBM (Physical Buffer Module) and ingress PBM modules have a separate pool of data nodes for storing the outgoing and incoming packets. The number of data nodes for each port is 72.

The PBM ingress watermark registers, specify the no of data nodes that should be available, in order to accept a packet with some particular priority (PRIO+CRF). By default the watermark for low priority is close to the maximum, the SRIO throughput for low priority packet will be limited. Users may adjust this value according to their own application. Below is example code to configure it.


```
/*Allocates ingress Data Nodes and Tags based on priority.
These registers must only be changed while boot_complete is
deasserted or while the port is held in reset.*/
for(i=0;i<SRIO_MAX_PORT_NUM;i++)
{
 if(FALSE == srio_cfg->blockEn.bBLK5_8_Port_Datapath_EN[i])
  continue;

 /*maximum data nodes and tags are 72 (0x48).
 Each data node stores 32 bytes of data*/
 srioRegs->RIO_PBM[i].RIO_PBM_SP_IG_WATERMARK0 =
  (36<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK0_PRIO0_WM_SHIFT)
  |(32<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK0_PRIO0CRF_WM_SHIFT);
 srioRegs->RIO_PBM[i].RIO_PBM_SP_IG_WATERMARK1 =
  (28<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK1_PRIO1_WM_SHIFT)
  |(24<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK1_PRIO1CRF_WM_SHIFT);
 srioRegs->RIO_PBM[i].RIO_PBM_SP_IG_WATERMARK2 =
  (20<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK2_PRIO2_WM_SHIFT)
  |(16<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK2_PRIO2CRF_WM_SHIFT);
 srioRegs->RIO_PBM[i].RIO_PBM_SP_IG_WATERMARK3 =
  (12<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK3_PRIO3_WM_SHIFT)
```

```
   |8<<CSL_SRIO_RIO_PBM_SP_IG_WATERMARK3_PRIO3CRF_WM_SHIFT);
}
```

4.3.2 Check SRIO registers

When we debug SRIO program, we often need check the SRIO register values, the best way to check it in CCS is to watch it through the SRIO register pointer in the watch window, the registers can be show in a struct tree as below.

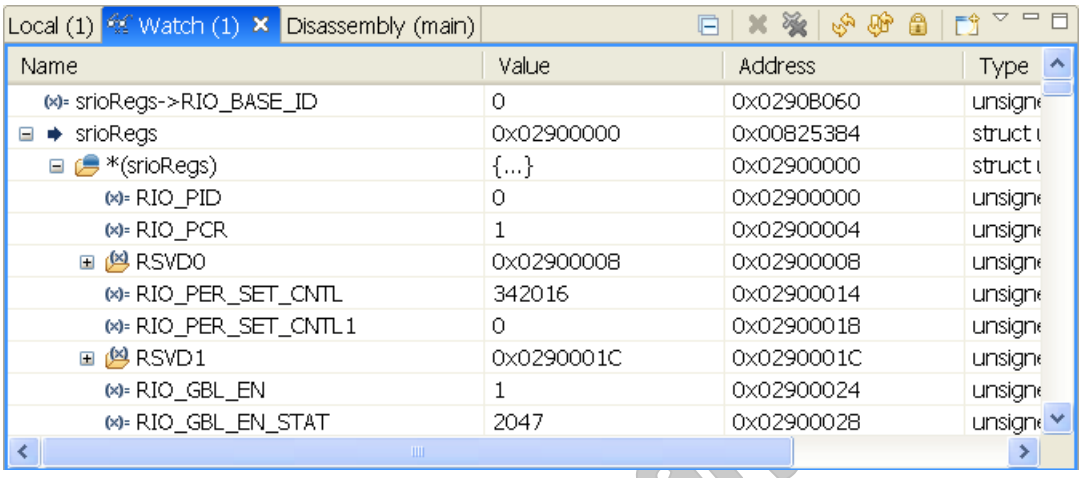

Figure 12 Check SRIO registers in CCS watch Window

5 SRIO performance data

Performance of SRIO is measured on 1GHz TCI6618 EVM.

5.1 SRIO transfer Overhead

In this application report, transfer overhead is defined as the time to transfer minimum data element (1 double-word), it is measured with Serdes loopback test.

Following table shows the average measured overhead for different cases.

Table 3. SRIO Transfer overhead

Transfer type	Average overhead (cycles)	Comments
SWRITE	822	Time between writing LSU registers to detecting the data in destination buffer.
NREAD	1316	
Type 9 message	1014	Time between pushing packet to the TX queue and getting the packet in the RX queue.
type11 message	1011	
Doorbell	929	Time between writing LSU registers to executing first instruction in ISR (Interrupt Service Routine).

Transfer overhead is a big concern for short transfers. Single-element transfer performance is overhead-dominated.

5.2 Throughput of different types

[FIXME] Test condition

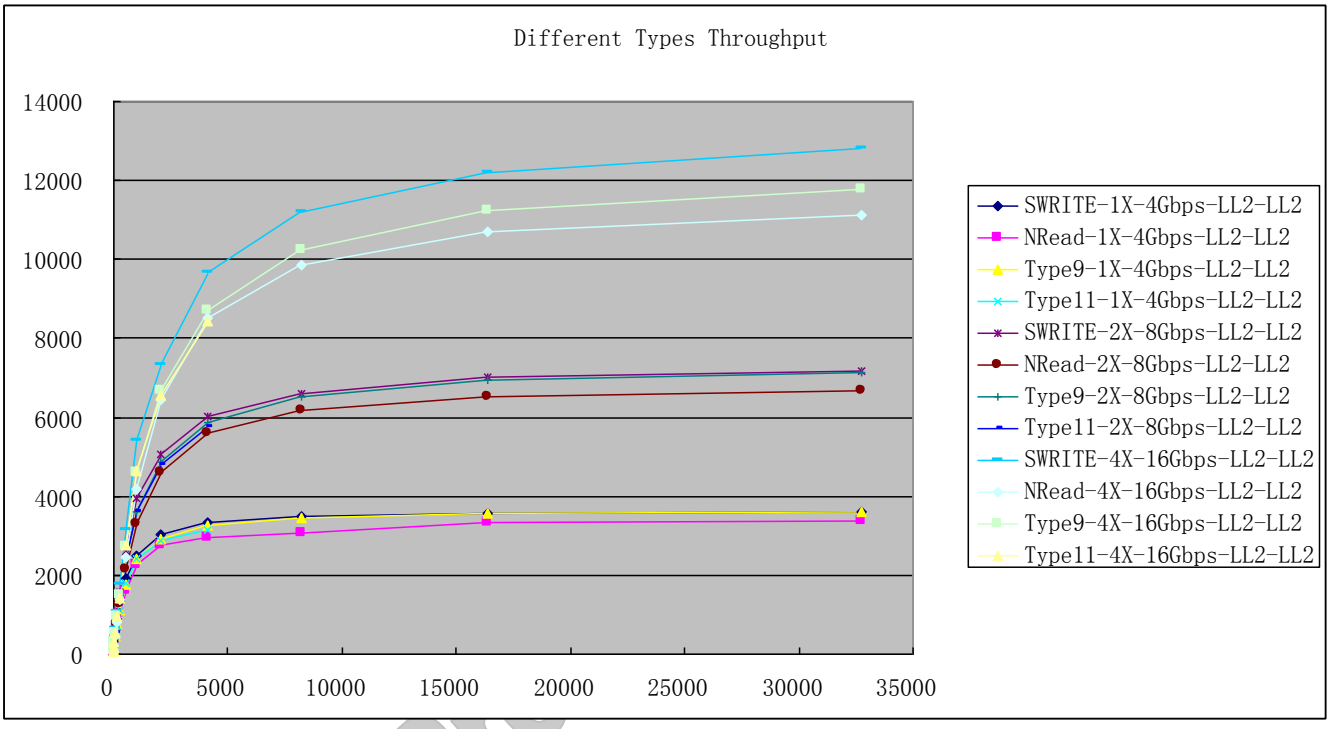

Figure 13 Throughput of different packet types

[FIXME] Only include few curves to make it clear:

SWRITE 4x5Gbps

NREAD 4x5Gbps

Type9 4x5Gbps

Type11 4x5Gbps

[FIXME] summarize how to choose the types for different conditions.

5.3 Throughput of different link configuration

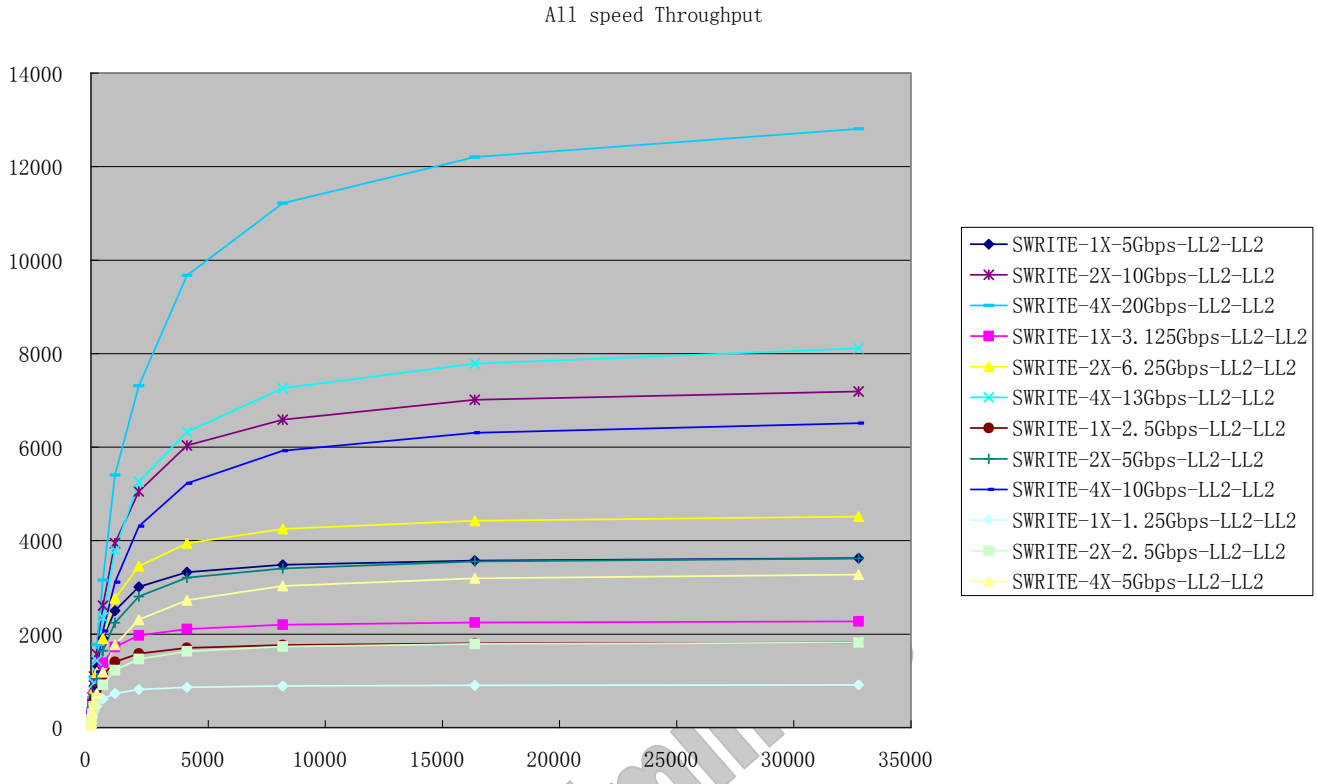

Figure 14 Throughput of different link configuration

[FIXME] use legend like SWRITE 4x5Gbps, SWRITE 2x3.125Gbps.

5.4 Effect of different memory buffer on SRIO throughput

[FIXME] show following curves:

SWRITE 4x5Gbps-LL2-LL2

SWRITE 4x5Gbps-LL2-SL2

SWRITE 4x5Gbps-LL2-DDR

SWRITE 4x5Gbps-SL2-SL2

SWRITE 4x5Gbps-SL2-DDR

SWRITE 4x5Gbps-DDR-DDR

[FIXME] summarize the Effect of different memory buffer on SRIO throughput

Appendix A. Example Project Introduction

The example project for this test is built with CCS4.2.2, it consists of two source code folders as following figure shows.

The files in common folder are the common configuration files which are used to configure the peripherals in the KeyStone devices. It includes the DDR3 configuration functions, Navigator configuration and driver functions, Serdes configuration functions and SRIO configuration and driver functions.

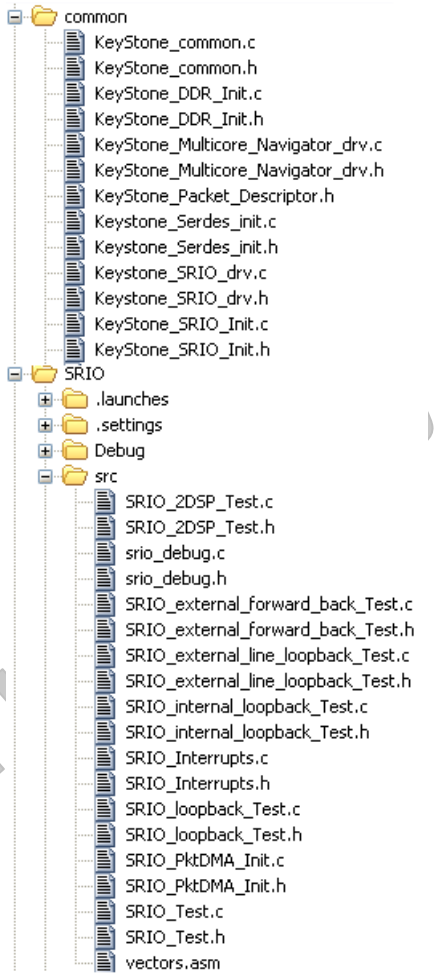

Figure 15 Directory Structure of Example Codes

The files in SRIO/src folder are code for different test case, for example, internal loopback test, external loopback, DSP to DSP test, interrupt test. Packet DMA and QMSS configurations are in the SRIO_PktDMA_Init.c. SRIO_debug.c includes some functions to check and print status registers for debug purpose.

The test case is chosen by below code in SRIO_Test.c.

```
SRIO_Loopback_Mode loopback_mode= SRIO_DIGITAL_LOOPBACK;
```

The loopback mode is defined as below:

```
typedef enum
{
            SRIO_NO_LOOPBACK = 0,
            SRIO_DIGITAL_LOOPBACK,
            SRIO_SERDES_LOOPBACK,
            SRIO_EXTERNAL_LINE_LOOPBACK,
            SRIO_EXTERNAL_FORWARD_BACK
}SRIO_Loopback_Mode;
```

SRIO_NO_LOOPBACK actually selects the test between two DSPs.

For each test mode, the program iterates many transfers with different configurations. In the source code for each test mode, you may change these test configurations.

Below table setup the SRIO packet types are tested.

```
SRIO_Packet_Type packet_type[]=
{
            SRIO_PKT_TYPE_SWRITE   ,
            //SRIO_PKT_TYPE_NWRITE   ,
            //SRIO_PKT_TYPE_NWRITE_R ,
            SRIO_PKT_TYPE_NREAD    ,
            SRIO_PKT_TYPE9_STREAM  ,
            SRIO_PKT_TYPE11_MESSAGE
};
```

Below table setup path configurations are tested.

```
SRIO_1x2x4x_Path_Control internal_path[]=
{
            SRIO_PATH_CTL_1xLaneA,
            SRIO_PATH_CTL_2xLaneAB,
            SRIO_PATH_CTL_4xLaneABCD
};
```

Below array setup all link speeds are tested.

```
float internal_speed[]= {5, 3.125, 2.5, 1.25};
```

Endian mode of this project can also be changed in build properties of CCS.

To rebuild the project after configuration change, you may need change the including path of CSL header files. Only register layer CSL header files are used by this project, PDK is not used by this project. The including path can be changed in compiler include options dialog or be changed in macros.ini file under the SRIO folder, the CSL_INSTALL_PATH in the macros.ini must be changed based on user's system.

The project can be run on TCI6608 EVM or TCI6618 EVM, the code automatically indentify the board, so user do not need modify code between these two boards. Since TCI6608 EVM only has one DSP, so only the digital loopback and Serdes loopback can be run.

To run digital loopback and Serdes loopback, just load the program to one DSP and run it.

On TCI6618 EVM, lane 2 and 3 are connected between two DSPs, so external loopback or transfer between two DSPs can be run on them. The software detects the DSP number automatically, so same program is run on the two DSP for different purpose. The first DSP is master for the tests, and the second DSP is slave for the tests, the second DSP should be run firstly and then the first DSP.

After run the program you should see output in CCS like below.

```
SRIO_DIGITAL_LOOPBACK test start.....................................
Initialize main PLL = x236/29
Initialize DDR PLL = x20/1
configure DDR at 666 MHz
SRIO link speed is 5.000Gbps
SRIO path configuration 1xLaneA
SWRITE   from 0x10802200 to 0x1080a200,      8 bytes,    635 cycles,   100 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,     16 bytes,    794 cycles,   161 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,     32 bytes,    792 cycles,   323 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,     64 bytes,    931 cycles,   549 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,    128 bytes,   1164 cycles,   879 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,    256 bytes,   1568 cycles,  1306 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,    512 bytes,   2075 cycles,  1973 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,   1024 bytes,   3235 cycles,  2532 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,   2048 bytes,   5426 cycles,  3019 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,   4096 bytes,   9939 cycles,  3296 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,   8192 bytes,  18829 cycles,  3480 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,  16384 bytes,  36739 cycles,  3567 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x1080a200,  32768 bytes,  72312 cycles,  3625 Mbps, completion code = 0
SWRITE   from 0x10802200 to 0x c00a000,      8 bytes,    649 cycles,    98 Mbps, completion code = 0
```

References

1. *Serial RapidIO (SRIO) for KeyStone Devices User's Guide*(SPRUGW1)
2. *KeyStone Architecture Multicore Navigator User Guide* (SPRUGR9)
3. *TMS320C66x DSP CorePac User Guide* (SPRUGW0)
4. *TMS320CC66x CPU and Instruction Set Reference Guide* (sprugh7)
5. *TMS320TCI6608 data manual (*SPRS623*)*
6. *TMS320TCI6618 data manual (*SPRS688*)*