IDT®
Tsi56x Software Initialization Design Note

80B8010_DN002_06

September 2, 2009

6024 Silver Creek Valley Road San Jose, California 95138
Telephone: (408) 284-8200 • FAX: (408) 284-3572
Printed in U.S.A.

1. Tsi56xx Software Initialization Design Note

The Tsi56xx is a high performance multi-port switch, based on the Serial RapidIO standard. The RapidIO ports operate with a 1-bit or 4-bit lane width up to 3.125 Gbps. This design note describes the software initialization for the Tsi56xx internal registers.

This document should be used with the *Tsi564A User Manual* or the *Tsi568A User Manual*. It contains the following information:

2. RapidIO System Initialization and Routing

The RapidIO interconnect standard targets both memory-mapped distributed memory systems and message based systems.

A system must be initialized and packets properly routed for a RapidIO system to operate.

2.1 RapidIO System Software Initialization

In a RapidIO system, the host must configure the system. The following steps are required for a system configuration:

The steps in **bold** are explained in this document. For further information on the other steps please see *RapidIO Interconnect Specification (Version 1.2)* and the specific RapidIO processing element (endpoint) user manual.

1. Determine if the host is synchronized and aligned with the adjacent device
2. Interrogate adjacent device (through the RIO_PE_FEAT register) to determine if it is a switch or a processing element.
   a. If the device is a switch the following steps are performed:
   - Determine the number of ports on the switch (through the RIO_SW_PORT register)
   - **Determine which ports of the switch are synchronized and aligned**
   - **Set-up routing tables**
   - Discover other devices connected to the switch (repeat step 2)

   b. If the device is a processing element it must be configured. Endpoint configuration is device specific. Refer to the endpoint documentation for information.

2.2 RapidIO System Packet Routing

There are typically two types of elements in a RapidIO system: switches and processing elements. System packet routing in a RapidIO system uses device identifier (device ID) based packet routing to direct packets through the switches to a given processing elements. Each processing element has one or more unique device IDs. The transaction type (tt) field defines the size of the device ID. For more information see the *RapidIO Interconnect Specification: Common Transport Layer.*

The Tsi56xx supports both 8-bit and 16-bit device ID fields (tt = 00b and tt=01b). Up to 2^8 (256) devices can attach to a switching fabric with 8-bit device ID fields and 2^{16} (65536) for 16-bit device IDs.

2.2.1 Processing Element Packet Routing

When a processing element generates a packet, it inserts the device ID of the destination processing element into the packet header. Packets are routed through the RapidIO switch based on the destination device ID. The device ID of the source processing element is also inserted into the packet header, for use by the destination processing element when generating response packets.

When the destination of a request packet generates a response packet, it swaps the source and destination fields from the request, making the original source the new destination and itself the new source.

2.2.2 Switch (Tsi56xx) Internal Packet Routing

In the Tsi56xx, as in all RapidIO switch processing elements, packet routing through the switching fabric is implementation dependent. However, the RapidIO Specification enables the reading of packet routing information (refer *RapidIO Interconnect Specification (Version 1.2) for more information).*

> Internal packet routing refers to routing between an ingress (inbound) port and an egress (outbound) port.

In the Tsi56xx implementation, the device performs an exit port look-up for each packet entering the device for packet routing. Devices connected to the Tsi56xx ports can be other RapidIO switch devices or endpoint devices.

When the Tsi56xx receives a maintenance packet the Hop Count field is reviewed to determine if the packet is forwarded or destined for the Tsi56xx. A hop count of 0 indicates the maintenance packet is destined for the Tsi56xx. For more information on how the Tsi56xx deals with maintenance packets, see the *Tsi564A User Manual* or *Tsi568A User Manual*, available at www.idt.com.

3. Tsi56xx Software Initialization

Initializing the Tsi56xx allows packets to traverse the switch. This section describes the software initialization for the Tsi56xx.

3.1 Tsi56xx Registers

The Tsi56xx contains registers that enable a processing element to determine its capabilities and set-up any required configuration (such as routing tables) through maintenance read and write operations. All registers are 32 bits wide and are accessed in 32-bit quantities.

> Maintenance transactions refer to a specific transaction type that access the address mapped locations within a device ID mapped device (such as the Tsi56xx).

3.1.1 Programming Registers with Maintenance Packets

The Tsi56xx registers can be accessed by a host through the use of maintenance packets.

If the Tsi56xx receives a maintenance packet with a hop count of 0, the maintenance packet is destined for its registers (either a read or write). If the hop count is not 0, the Tsi56xx decrements the hop count, reads the destination ID, and forwards the packet to the port mapped in its lookup table.

> Hop counts are used to specify the number of switches (or hops) into the network from the issuing processing element that is being addressed. Whenever a switch processing element that does not have an associated device ID receives a maintenance packet it examines the hop_count field. If the received hop_count is zero, the access is for that switch. If the hop_count is not zero, it is decremented and the packet is sent out of the switch according to the destinationID field.
>
> This method allows easy access to any intervening switches in the path between two addressable processing elements.

3.1.2 Register Set-up using EEPROM

The Tsi56xx registers can be initialized after power-up by an on-board EEPROM. The set-up of the Tsi56xx using the EEPROM is the same as other register programming, however the source of the data is the EEPROM. For further information on loading the registers from an EEPROM, refer to the *Tsi564A User Manual* or *Tsi568A User Manual*.

3.1.3 Programming with JTAG

The Tsi56xx includes a JTAG interface to help debug new RapidIO designs. This interface also enables accesses to the Tsi56xx registers using the JTAG internal register access command (IRAC). The JTAG interface can program the lookup tables (see "Tsi56xx Lookup Tables" on page 7), I^2C devices, and all internal Tsi56xx registers.

For more information on the JTAG interface, refer to the *Tsi564A User Manual* or *Tsi568A User Manual*.

3.2 Tsi56xx Port Status

Before performing read or write transactions from a host device beyond the Tsi56xx, the host must first determine if a given port on the Tsi56xx is initialized. To determine if a port is initialized the host software must read the SP{0..15}_ERR_STATUS register. The port is initialized if the PORT_OK bit is set and PORT_UNINT is clear.

3.3 Tsi56xx Lookup Tables

The Tsi56xx has a flat lookup table (LUT) architecture which has 512 destination IDs. These destination IDs can be mapped to user selectable egress ports. Destination IDs that fall outside the 0 -511 destination ID range are sent to the egress port identified in the RIO Route LUT Attributes CSR register.

3.3.1 Registers Used in Lookup Table Configuration

The Tsi56xx's RapidIO interface is compliant with the *RapidIO Interconnect Specification (Version 1.2)*. The following RapidIO standard registers are used by the Tsi56xx for programming the lookup tables:

- RIO Route Config DestID CSR
- RIO Route Config Output Port CSR
- RIO Route LUT Size CAR
- RIO Route LUT Attributes CSR

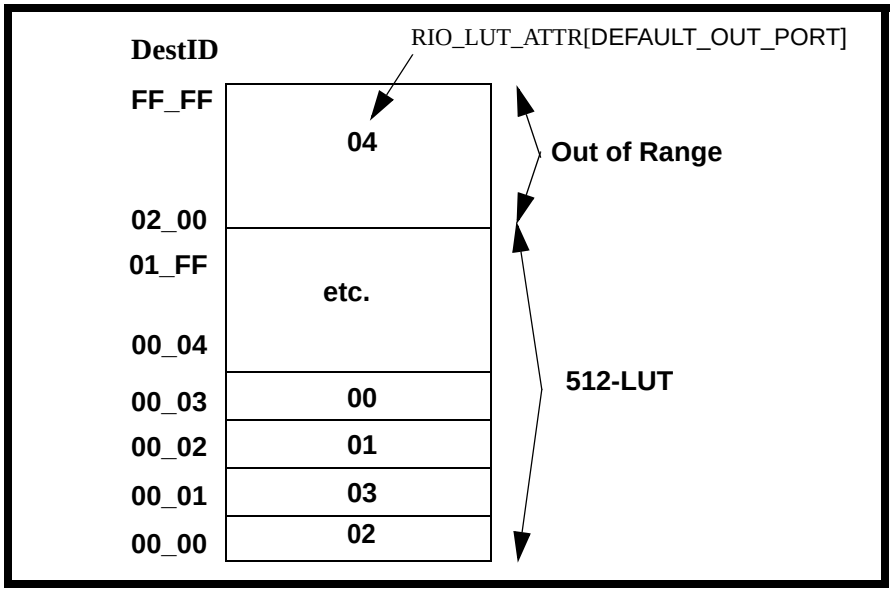
3.3.2 Programming the Tsi56xx Lookup Tables

Each of the ports on the Tsi56xx has its own lookup table. Each lookup table can be programmed with different values which allows each port to route packets differently. The lookup table maps the packet to the correct output port based on the destination ID.

All ports can be programmed using the using the RapidIO standard registers. To individually program the lookup tables, the IDT-specific registers must be used (see “Verifying / Reading a Lookup Table Entry - Example Three” on page 10)

Figure 1 shows an example of a lookup table. In this example, a destination ID of 0x0002, or 0x02, is routed by the switch to output port 1. A destination ID of 0x0003, or 0x03, is routed out port 0 and destination IDs greater than 0x1FF are routed out port 4.

Figure 1: Lookup Table Example



The IDT-specific registers (RIO Port x Route Config DestID CSR and RIO Port x Route Config Output Port CSR) allows the lookup tables on all the ports to be configured either independently or simultaneously. The IDT-specific registers also provide an auto-increment bit so that the destination ID in the RIO Port x Route Config DestID CSR is automatically incremented after each read or write. This allows sequential access to the lookup table entries using half the number of register operations of the RapidIO standard interface.

Lookup Table Configuration Examples

The Tsi56xx lookup tables can be configured through an external EEPROM or through software maintenance writes to the Tsi56xx registers.

The entire lookup table must be configured on each port to avoid undefined lookup table entries which may cause non-deterministic behavior.

The following sequence is used to program the lookup tables:

1. Write the destination ID to be configured or queried to the RIO Route Config DestID CSR register.
2. Read the RIO Route Config Output Port CSR register to determine the current egress port for the destination ID, or write this register to change the configuration of the destination ID.

Adding a Lookup Table Entry - Example One

In the following example, a routing entry is added to all ports to route destination ID 0x98 to output port 0x4.

To add a lookup table entry, perform the following steps:

1. Write to the RIO Route Config DestID CSR with a value of 0x00000098
   - Destination ID 0x98
2. Write to the RIO Route Config Output Port CSR a value of 0x00000004
   - Port 0x4

Adding a Lookup Table Entry - Example Two

In the following example, a routing entry is added to port 0x5 to route destination ID 0x20 to output port 0x3.

To add a lookup table entry, perform the following steps:

1. Write to the RIO Route Config DestID CSR with a value of 0x80000020
   - Destination ID 0x20
   - Auto-increment 0x1

— For more information on the Auto-increment functionality, refer to *Tsi564A User Manual* or *Tsi568A User Manual*.)

2. Write to the RIO Route Config Output Port CSR a value of 0x00000003
   - Port 0x3

In this example, if a further write to RIO Route Config Output Port CSR was performed, the output port for Destination ID 0x20 would be configured.

Verifying / Reading a Lookup Table Entry - Example Three

In the following example, the output port for Destination ID 0x54 is read.

To verify and read a lookup table entry, perform the following steps:

1. Write to the RIO Route Config DestID CSR with a value of 0x00000054
   - Destination ID 0x54
2. Read to the value in RIO Route Config Output Port CSR
   - value represents the output port for packets with Destination ID=0x54

The value reported back only reports back the value in port 0.

3.3.3 Lookup Table Entry States

A lookup table entry can be in one of three states: mapped, unmapped, and parity error. After any reset all lookup table entries are undefined (an unknown state). All lookup table entries must be programmed to a known value after reset to achieve predictable operation. A lookup table entry that routes packets to a port that exists within the Tsi56xx is *mapped*. A lookup table entry that routes packets to a port that does not exist within the Tsi56xx is *unmapped*. When a lookup table entry's parity is incorrect, the lookup table entry is in a parity error state. Table 1 shows the possible lookup table states.

Table 1: Lookup Table States

Lookup Table Entry State	Action on Packet Arrival
Mapped	Packet is routed to the specified output port
Unmapped	Packet recorded in error capture registers, packet discarded, RIO Port x Error and Status CSR IMP_SPEC_ERR is set, a port write can be generated (if enabled) and an interrupt can be generated (if enabled).
Parity Error	Packet recorded in error capture registers, packet discarded, RIO Port x Error and Status CSR IMP_SPEC_ERR is set, a port write is generated and an interrupt is generated.
UnProgrammed (Undefined)	Non-deterministic operation may occur.

When a port value for a lookup table entry is unmapped, the default port is used for routing the packet as defined in RIO Route LUT Attributes CSR register. If the default port value is unmapped, packets routed using the default port value are discarded and the IMP_SPEC_ERR bit is set in the RIO Port x Error Detect CSR register.

Lookup table entries can be programmed through the standard RapidIO compliant interface or through a IDT-specific registers (RIO Port x Route Config DestID CSR and RIO Port x Route Config Output Port CSR) which can auto-increment entries reducing the number of maintenance cycles required. For further information on this interface see the *Tsi568A User Manual*.

For further information on parity see *Tsi568A User Manual*.

3.3.4 Routing Destination IDs Based on the Lookup Table Entries

The Tsi56xx routes packets with the varying destination ID possibilities in the following way:

- If a packet's destination ID is in the range of 0 - 511, and its lookup table entry has been programmed, the packet is routed out the specified port.
- If the table entry is programmed to the unmapped state (output port is set to 0xFF), the packet is routed using the default egress port defined in RIO Route LUT Attributes CSR register.
- If a packet's destination ID is greater than 511, the packet is also routed using the default egress port defined in RIO Route LUT Attributes CSR.
- If the default egress port in RIO Route LUT Attributes CSR is unmapped, the Tsi56xx discards the packet and sets the IMP_SPEC_ERR bit in the RIO Port x Error Detect CSR register.

All lookup table entries are in an unknown state after power-up. All entries should be programmed to a mapped or unmapped state to ensure predictable operation. It is strongly recommended that the value 0xFF be used as the port value for writing unmapped lookup table entries. An unmapped lookup table entry returns the value of 0xFF as the port value when read.

- Packets with 16 bit destination IDs from 0x0000 through 0x00FF are routed using the same lookup table entries as those for the 8 bit destination IDs 0x00 through 0xFF.

A. Example Source Code

A.1 Tsi568/Tsi564 Routing Table Code

```
/*****************************************************************************
* Function: void Tsi568SetRoute(int routeId, int hopCount, int destId, int rioInterface, int portN)
*
* Purpose:  configure device(routeId,hopCount) to route destID coming from portN to rioInterface
*           if portN = -1, it configures all ports the same.
*
* Returns:  nothing
*
*****************************************************************************/
void Tsi568SetRoute(int routeId, int hopCount, int destId, int rioInterface, int portN)
{
   if (portN != -1) {
   /* First read the look-up table register.  Then mask out current
    * destination port value.  Once that has been masked, write the
    * new destination by performing a bitwise OR with the new value.
    * Then write the new register value to the look-up table register.
    */
      maintWrite(routeId, hopCount, Tsi568_SP0_ROUTE_CFG_DESTID + 0x100*portN, destId);
      maintWrite(routeId, hopCount, Tsi568_SP0_ROUTE_CFG_PORT + 0x100*portN, rioInterface);
   } else {
      /* set all ports in the switch the same */
      maintWrite(routeId, hopCount,Tsi568_SPBC_ROUTE_CFG_DESTID, destId);
      maintWrite(routeId, hopCount, Tsi568_SPBC_ROUTE_CFG_PORT, rioInterface);
   }
   return;
}


/*****************************************************************************
* Function: int Tsi568GetRoute(int routeId, int hopCount, int destId, int portN)
*
* Purpose:  returns port number where destID is routed to coming from portN on device(routeId,hopCount)
*
* Returns:  rioInterface
*
*****************************************************************************/
int Tsi568GetRoute(int routeId, int hopCount, int destId, int portN)
{
   maintWrite(routeId, hopCount, Tsi568_SP0_ROUTE_CFG_DESTID + 0x100*portN, destId);
   return maintRead(routeId, hopCount, Tsi568_SP0_ROUTE_CFG_PORT + 0x100*portN);
}
```

A.2 Example RapidIO Discovery Code

```
extern struct rioSwitch {
    UINT32 SwitchIdentity;  /* Switch Identity */
    UINT16 hopCount;        /* Hop Count to reach this switch */
    UINT16 DeviceID;        /* Associated Device ID in the path to this switch */
    UINT16 SwitchTag;/* Value stored in Switch component tag (used for Port writes) */
    UINT8  RouteTable[512]; /* Switch Routing Table Entries*/
    UINT16 destID[16];      /* destID of end point connected to specified port of the switch */
    UINT32 EPIdentity[16];  /* Identity of end point connected to specified port of the switch */
} rioSwitch;

/* Currently available Device ID and Switch ID to be assigned to the
 * end point device  */
static UINT16 nextAvailDeviceID = 1;
static UINT16 nextAvailSwitchID = 1;

static struct rioSwitch Switches[256];


/*****************************************************************************
* Function: int Rio_Init_and_Discovery(void)
*
* Purpose:  discovery and set-up of a RapidIO network
*           This function performs the initial set-up prior to the
*           discovery done in rio_discover()
*
* Returns:  Next Available Device ID (ie Destination ID)
*
*****************************************************************************/
int Rio_Init_and_Discovery(void)
{
    int i,j;

    nextAvailDeviceID = 1;
    nextAvailSwitchID = 1;

/* Initialize the Switches structure to default values */
    for(i=1; i<256; i++)
    {
     for(j=0; j<512; j++)  Switches[i].RouteTable[j] = 0xee;
     for(j=0; j<16; j++)   Switches[i].destID[j] = 0xeeee;
     for(j=0; j<16; j++)   Switches[i].EPIdentity[j] = 0xeeee;
     Switches[i].DeviceID = 0xeeee;
     Switches[i].hopCount = 0xeeee;
     Switches[i].SwitchIdentity = 0xeeee;
    }

    /* set device id we running on to 1 */
    setLocalRioDevID(nextAvailDeviceID);
    Switches[1].DeviceID = nextAvailDeviceID;
    nextAvailDeviceID++;

    // discover and set-up RIO network
    rio_discover(0,1,1,0);
```

```
// back fill first switch LUTs.
    rio_discover_full_routing(nextAvailSwitchID);

printf("\n\n=============================================\n");
    print_switch_info();
    printf("=============================================\n\n\n");

    return (nextAvailDeviceID-1);
}

/*****************************************************************************
* Function: int rio_discover(int hc, int curDeviceID, int curSwitchID, int arg)
*
* Purpose:  discovery and set-up of a RapidIO network
*
* Returns:  1 - device found.
*           0 - switch found.
*
*****************************************************************************/

int rio_discover(int hc, int curDeviceID, int curSwitchID, int verbose)
{
    UINT32 features, ports, inb_port;
    int i,j, retval;
    int devIDbehindTheSwitch;

    /* That algorithm assumes that there is no loops in the system */
    printf("(RIO_DISCOVER) NextAvailDestID= %x, current hopCount= %x\n\r", nextAvailDeviceID, hc);

    features = maintRead(0xfe, hc, RIO_PE_FEATURES_CAR);

    if (RIO_PE_FEATURES_CAR_SWITCH == (features & RIO_PE_FEATURES_CAR_SWITCH))
    { /* if neighbour device is a switch */
        inb_port = maintRead(0xfe, hc, RIO_SW_PORT_INFO_CAR);
        ports = (inb_port & RIO_SW_PORT_INFO_CAR_PORT_TOTAL) >> 8;
        inb_port &= RIO_SW_PORT_INFO_CAR_PORT_NUM;
        printf("(RIO_DISCOVER) Switch(%i) found, inb_port = %x\n\r",curSwitchID, inb_port);

      if (!isDiscovered(0xfe, hc))
 {
                setDiscoveredBit(0xfe, hc);

                /* Record the switch device identity */
                Switches[curSwitchID].SwitchIdentity = maintRead(0xfe, hc, RIO_DEV_ID_CAR);

                /* Set component tag within switch - use the curSwitchID */
                maintWrite(0xfe, hc, RIO_COMP_TAG_CSR, curSwitchID);
                Switches[curSwitchID].SwitchTag = curSwitchID;

                nextAvailSwitchID ++;           /* Increment the available switch ID */

                /* Initialize the current switch routing table to add entries for all previously discovered devices so that
                        they are routed correctly. Start with the host device ID (0x00) and end with DeviceID-1. */
                /* Synchronize the current switch routing table with the global table */

                for (i=1; i<nextAvailDeviceID; i++)
```

```
		{
			SetRoute(0xfe, hc, i, inb_port, -1);
			Switches[curSwitchID].RouteTable[i] = inb_port;
		}

		/* Update the hopCount to reach the current switch */
		Switches[curSwitchID].hopCount = hc;

		if (Switches[curSwitchID].hopCount != 0)
			Switches[curSwitchID].destID[inb_port] = 0x101;
		else
			Switches[curSwitchID].destID[inb_port] = 0x1;

		for (i = 0; i<ports; i++)
		{
			if (verbose)
			 {		printf("Switch %i ",curSwitchID); }
			if ((i != inb_port) && (isPortTrained(0xfe, hc, i,verbose)))
			{
			  printf("(RIO_DISCOVER) Exploring port %x of the switch(%i)\n\r", i,curSwitchID);
			  /* set the route from/to that port and destID 0xff */
			  SetRoute(0xfe, hc, 0xfe, i, inb_port);

			  devIDbehindTheSwitch = nextAvailDeviceID;
			  if (rio_discover(hc+1, nextAvailDeviceID, nextAvailSwitchID, i) == 0) /* found switch */
			  {

				/* If more than one end point device was found, update the current switch routing table
entries
					beginning with the curDeviceID entry and ending with the DeviceID-1 entry. */
				/* Synchronize the current switch routing table with the global table */
				//if ((nextAvailDeviceID-1) > devIDbehindTheSwitch)
				if ((nextAvailDeviceID) > devIDbehindTheSwitch)
				{
					printf("(RIO_DISCOVER) Found IDs behind switch(%i) \n\r",curSwitchID+1);
				  for (j=devIDbehindTheSwitch; j<nextAvailDeviceID; j++)
				  {
					SetRoute(0xfe, hc, j, i, -1);
					Switches[curSwitchID].RouteTable[j] = i;
				  }
				  Switches[curSwitchID].destID[i] = (0x100) | (0xff & devIDbehindTheSwitch);
				}
			  }
			  else /* found endpoint */
			  {
				SetRoute(0xfe, hc, nextAvailDeviceID-1, i, -1);
				Switches[curSwitchID].RouteTable[nextAvailDeviceID-1] = i;
				Switches[curSwitchID].destID[i] = nextAvailDeviceID-1;


				/* Update the associated Device ID in the path. */
				if (Switches[curSwitchID].DeviceID == 0xeeee)
				  Switches[curSwitchID].DeviceID = nextAvailDeviceID-1;
			  }
		}
		if ((i == inb_port) & verbose)
```

```
                {
                        printf("port %x is my port \n\r",i);
                }
        }
      }
      else
      {
              printf("(RIO_DISCOVER) Switch already found \n\r");
      }
         retval = 0;
        }
        else
        { /* endpoint */
          UINT32 devID;
          printf("(RIO_DISCOVER) Endpoint found, destID = %x, hc= %x  %x  %x\n\r", curDeviceID, hc);
          setDiscoveredBit(0xfe, hc);
          devID = initRioDevice(0xfe, hc, 1);
          setDestId(curDeviceID, hc);
          nextAvailDeviceID++;
          retval = 1;
        }
        return retval;
      }

      /*****************************************************************************
      * Function: int rio_discover_full_routing(int SwitchIDnum)
      *
      * Purpose:populates the switches strucutre for a new switch in a system
      *
      * Returns:  0
      *
      *****************************************************************************/
      int rio_discover_full_routing(int SwitchIDnum)
      {
         int i,j,k;

         for (i=1; i<SwitchIDnum; i++)
           for (k=0; k<16; k++)
             if ((Switches[i].destID[k] != 0xeeee) && ((Switches[i].destID[k] & 0x100) == 0))
             { /* endpoint */
               for(j=1;j<SwitchIDnum;j++)
                 if (j!=i)
                 {
                   Switches[j].RouteTable[Switches[i].destID[k]] =  Switches[j].RouteTable[Switches[i].DeviceID];
                   SetRoute(Switches[j].DeviceID, Switches[j].hopCount, Switches[i].destID[k],
Switches[j].RouteTable[Switches[i].DeviceID], -1);
                 }
             }
         return 0;
      }
      /*****************************************************************************
      * Function: void print_switch_info()
      *
      * Purpose:  prints out the information stored in the Switches structure
      *
      * Returns:  none.
```

```
*
*****************************************************************************/
void print_switch_info()
{
   int i,j;

   for(i=1; i<nextAvailSwitchID; i++)
   {
     printf("switch[%x] Identity=%x CompID=%x HopCnt=%x RouteID=%x \n\r", i,
                          Switches[i].SwitchIdentity,
                          Switches[i].SwitchTag,
                          Switches[i].hopCount,
                          Switches[i].DeviceID);
     //for(j=0; j<512; j++)
     //  if (0xee != Switches[i].RouteTable[j])
     //    printf("destID %x routed to %x port\n\r", j, Switches[i].RouteTable[j]);

     for(j=0; j<16; j++)
       if (0xeeee != Switches[i].destID[j])
         printf("\t device  with destID %x is connected to %x port\n\r",Switches[i].destID[j], j);
   }
   return;
}
/*****************************************************************************
* Function: int isPortTrained( int destId, int hopCount, int port, int verbose)
*
* Purpose:  Check if a given port(port) on a serial switch(destID,hop)
*
* Returns:  1 - port is trained and there is a partner
*           0 - port is not trained
*
*****************************************************************************/
int isPortTrained( int destId, int hopCount, int port, int verbose)
{
   int portStatus;

   portStatus = maintRead( destId, hopCount, RIO_EXT_FE(0x100, RIO_EF_PORT_ERR_STAT_CSR(port)));

   if ((portStatus & RIO_PORT_ERR_STAT_PORT_OK) == RIO_PORT_ERR_STAT_PORT_OK) return 1;
   else {
            if (verbose)
            {       printf("port %x is not trained\n", port); }
      return 0;
   }
}

/*****************************************************************************
* Function: int getNextPort(int hc, int direction, int current_port)
*
* Purpose:
*
* Returns:
*****************************************************************************/
int getNextPort(int hc, int direction, int current_port)
{
   int retval = 0;
```

```
        int i;
        printf("LOOPBACKS!\n\r");
        switch (direction)
        {
           case DIRECTION_COUNTERCLOCKWISE:
             switch (switchType(0xfe, hc))
             {
               case RIO_SWITCH_TSI500:
                 retval = (current_port - 1) & 0x3;
                 break;
               case RIO_SWITCH_TSI568:
               case RIO_SWITCH_TSI564:
                 i = current_port - 1;
                 while ((maintRead(0xfe, hc, Tsi568_SP0_ERR_STATUS + 0x20*i) &
    Tsi568_SP0_ERR_STATUS_PORT_OK) == 0)
                  { i--; }
                 retval =  i;
                 break;
               default:
                 break;
             }
             break;
           case DIRECTION_CLOCKWISE:
             switch (switchType(0xfe, hc))
             {
               case RIO_SWITCH_TSI500:
                 retval = (current_port + 1) & 0x3;
                 break;
               case RIO_SWITCH_TSI568:
               case RIO_SWITCH_TSI564:
                  i = current_port + 1;
                while ((maintRead(0xfe, hc, Tsi568_SP0_ERR_STATUS + 0x20*i) & Tsi568_SP0_ERR_STATUS_PORT_OK)
== 0)
                 { i++; }
                 retval =  i;
                 break;
               default:
                 break;
             }
             break;
           default:
           break;
        }
        return retval;
    }
```

CORPORATE HEADQUARTERS
6024 Silver Creek Valley Road
San Jose, CA 95138

for SALES:
800-345-7015 or 408-284-8200
fax: 408-284-2775
www.idt.com

for Tech Support:
408-360-1533
sRIO@idt.com
Document: 80B8010_DN002_06