RapidIO for Linux

Matt Porter
MontaVista Software, Inc.
mporter@mvista.com
mporter@kernel.crashing.org

Abstract

RapidIO is a switched fabric interconnect standard intended for embedded systems. Providing a message based interface, it is currently capable of speeds up to 10Gb/s full duplex and is available in many form factors including ATCA for telecom applications. In this paper, the author introduces a RapidIO subsystem for the Linux kernel. The implementation provides support for discovery and enumeration of devices, management of resources, and a consistent access mechanism for drivers and other kernel facilities. As an example of the use of the subsystem feature set, the author presents a Linux network driver implementation which communicates via RapidIO message packets.

1 Introduction to RapidIO

1.1 Busses and Switched Fabrics

To date, most well known system interconnect technologies have been shared memory bus designs. ISA, PCI, and VMEbus are all examples of shared memory bus systems. A shared memory bus interconnect will have a specific bus width which is measured by the number of data lines routed to each participant on the bus. An arbitration mechanism is required to determine which participant owns the bus for purposes of asserting an address-data cycle onto the bus. Other participants will decode the address-data cycle and latch data locally if the address cycle is intended for them. In the shared memory bus architecture, there is one global address space shared amongst all participants.

Switched fabric interconnect technology has been around for some time with proprietary implementations like StarFabric. In recent years though, standardized switched fabrics like HyperTransport, Infiniband, PCI Express, and RapidIO have become more familiar names. A switched fabric interconnect is usually modeled much like a switched network architecture. However, it provides features that a chip to chip or intra-chassis conventional shared memory bus standard would provide. Each node has at least one link that can be connected point to point or into a switch element. An implementation-specific routing method determines packet routing in the network. Typically, a switched fabric interconnect incorporates some method of sending messages and events through the network. In some cases, the switched fabric will implement memory mapped I/O over the network.

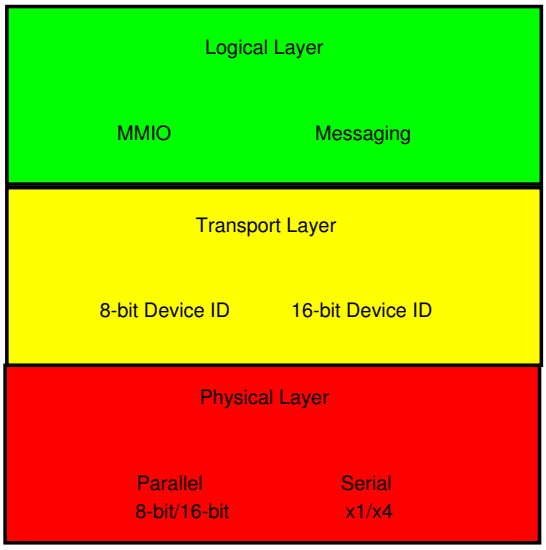

Figure 1: RapidIO Layers

1.2 RapidIO Overview

The RapidIO interconnect technology was originally created by Motorola for use in embedded computing systems. Motorola (now Freescale Semiconductor) later created the RapidIO Trade Association (RTA) to guide future development of the specification. A number of embedded silicon vendors are active members of the RTA and are now shipping or announcing RapidIO devices.

The RapidIO specification is divided into three distinct layers. These layers are illustrated in Figure 1.

1. Logical Layer
   Provides methods for memory mapped I/O (MMIO) and message-based access to devices. MMIO allows for accesses within a local memory space to generate read/write transactions within the address space of a remote device. Each device has a unique RapidIO address space that can range from 34-bits to 66-bits in size. RapidIO provides a messaging model with mailbox and doorbell facilities. A mailbox is a hardware port which can send and receive messages up to 4KB in size. A doorbell is a specialized message port which can be used for event notifications similar to message signaled interrupts.

2. Transport Layer
   Implements the device ID routing methodology. In RapidIO, packets are routed by a unique device ID. Two different sizes of device IDs are defined , either a small (8-bit) or large (16-bit) device ID. The small device ID allows a maximum of 256 devices whereas the large device ID allows a maximum of 65536 devices.

3. Physical Layer
   Offers either parallel or serial implementations for the physical interconnect. The parallel version is available in 8-bit or 16-bit configurations with full duplex speeds up to 8 Gb/s and 16 Gb/s, respectively. The serial implementation offers lane configurations of x1 or x4. In the x1 configuration, the single lane offers up to 3.125 Gb/s full duplex data throughput. In the x4 configuration, each lane offers up to 2.5Gb/s full duplex data throughput resulting in 10 Gb/s total bandwidth.

1.3 RapidIO versus PCI Express

RapidIO is often compared to PCI Express because of the popularity of PCI Express in the commodity PC workstation/server market. On the surface, both seem very similar, offering features that improve upon the use of conventional PCI as a system interconnect. RapidIO, however, is designed with some features targeted at specific embedded system needs that will likely facilitate its inclusion in many applications. There are now embedded processors available which include both PCI Express and RapidIO support on the chip.

PCI Express and RapidIO have similar data rate capabilities. PCI Express offers lane configurations of x1 through x32 where each lane offers 2 Gb/s full duplex data throughput. In a typical PC implementation there is a single x16 slot for graphics that handles 32 Gb/s full duplex and multiple x1 slots which can handle 4 Gb/s full duplex.

Both interconnects also have similar discovery models. A separate set of transactions is used to access configuration space registers. Configuration space accesses are used to determine existence of nodes in the system and additional information about those nodes.

A major difference between PCI Express and RapidIO is in system topology capability. PCI Express is backward compatible with PCI and therefore depends on the host and multiple slave device model. RapidIO is designed for multiple hosts in the system performing redundant discovery. In addition, it can be configured in any network topology allowing direct node to node communication.

Device addressing is very different as well. In PCI Express, the globally shared address space with hierarchical windows of address space is retained from PCI. This is important for backward compatibility of software and allows routing of packets via base address assignments. RapidIO's device ID based routing simplifies changes to the network due to device failure or hot plug events.

PCI Express does not offer a standardized messaging facility. Most modern distributed applications are based on message passing architectures.

2 RapidIO Hardware

The current generation RapidIO parts use an 8-bit wide parallel physical layer. These parts can support up to 8Gb/s full duplex data throughput. The first RapidIO processor elements (endpoints with a processor) are Freescale's MPC8540 and MPC8560 Systems-on-a-Chip (SoC). The first RapidIO switch is the Tundra Tsi500.

The first commercially available system with these parts is the STx GP3 HIPPS2 development platform. This system includes one or more STx GP3 boards containing the MPC8560 processor and a HIPPS2 RapidIO backplane with two Tsi500 switches. The Linux RapidIO subsystem is being developed using this platform with two STx GP3 boards plugged into the HIPPS2 backplane.

3 Linux RapidIO Subsystem

3.1 Subsystem Overview

Due to the discovery mechanism similarities between PCI and RapidIO, the RapidIO subsystem has a structure which is similar to that of the PCI subsystem. The subsystem hooks into the standard Linux Device Model (LDM) in a similar fashion to other busses in the kernel. RapidIO specific device and bus types are defined and registered with the LDM. The core subsystem is designed such that there is a clear separation between the generic subsystem interfaces and architecture specific interfaces which support RapidIO. Finally, a set of subsystem device driver interfaces is defined to abstract access to facilities by device drivers.

3.2 Subsystem Core

The core of the Linux RapidIO subsystem revolves around four major components.

1. Master Port. A master port is an interface which allows RapidIO transactions to be transmitted and received in a system. A master port provides a bridge from a processor running Linux into the switched fabric network.

2. Device. A RapidIO device is any endpoint or switch on the network.

3. Switch. A RapidIO switch is a special class of device which routes packets between point to point connections to reach their final destination.

4. Network. A RapidIO network comprises a set of endpoints and switches that are interconnected.

Each of these components is mapped into a subsystem structure. The RapidIO subsystem uses these structures as the root handle for manipulating the hardware components abstracted by the structures.

`struct rio_mport` (Figure 2) contains information regarding a specific master port. Master port specific resources such as inbound mailboxes and doorbells are contained in this structure. If a master port is defined as a enumerating host, then the structure will contain a unique host device ID. The host device ID is used for multi-host locking purposes during enumeration.

`struct rio_switch` (Figure 3) contains information about a RapidIO switch device. The structure is populated during enumeration and discovery of the system with information such as the number of hops to the switch and the routing table present in the switch. In addition, pointers to switch specific routing table operations reside here.

`struct rio_dev` (Figure 4) contains information about an endpoint or switch that is part of the RapidIO system. Fields are present to cache many common configuration space registers.

`struct rio_net` (Figure 5) contains information about a specific RapidIO network known to the system. It defines a list of all devices that are part of the network. Another list tracks all of the local processor master ports that can access this network. The `hport` field points to the default master port which is used to communicate with devices within the network.

3.3 Subsystem Initialization

In order to initialize the RapidIO subsystem, an architecture must register at least one master port to send and receive transactions within the RapidIO network. A `subsys_initcall()` is registered which is responsible for any arch-specific RapidIO initialization. This includes hardware initialization and registration of active master ports in the system. The final step of the initcall is to execute `rio_init_mports()` which performs enumeration and discovery on all registered master ports.

3.4 Enumeration and Discovery

The enumeration and discovery process is implemented to comply with the multiple host enumeration algorithm detailed in the *RapidIO Interconnect Specification: Annex I* [1]. Enumeration is performed by a master port which is designated as a host port. A host port is defined as a master port which has a host device ID greater than or equal to zero. A host device ID is assigned to a master port in a platform specific manner or can be passed on the command line.

```
struct rio_mport {
    struct list_head dbells;         /* list of doorbell events */
    struct list_head node;           /* node in global list of ports */
    struct list_head nnode;          /* node in net list of ports */
    struct resource iores;
    struct resource riores[RIO_MAX_MPORT_RESOURCES];
    struct rio_msg inb_msg[RIO_MAX_MBOX];
    struct rio_msg outb_msg[RIO_MAX_MBOX];
    int host_deviceid;               /* Host device ID */
    struct rio_ops *ops;             /* maintenance transaction functions */
    unsigned char id;                /* port ID, unique among all ports */
    unsigned char index;             /* port index, unique among all port
                                                interfaces of the same type */
    unsigned char    name[40];
};
```

Figure 2: struct rio_mport

```
struct rio_switch {
    struct list_head node;
    u16 switchid;
    u16 hopcount;
    u16 destid;
    u16 route_table[RIO_MAX_ROUTE_ENTRIES];
    int (*add_entry)(struct rio_mport *mport, u16 destid, u8 hopcount,
            u16 table, u16 route_destid, u8 route_port);
    int (*get_entry)(struct rio_mport *mport, u16 destid, u8 hopcount,
            u16 table, u16 route_destid, u8 *route_port);
};
```

Figure 3: struct rio_switch

```
struct rio_dev {
    struct list_head global_list;    /* node in list of all RIO devices */
    struct list_head net_list;       /* node in per net list */
    struct rio_net *net;             /* RIO net this device resides in */
    u16 did;
    u16 vid;
    u32 device_rev;
    u16 asm_did;
    u16 asm_vid;
    u16 asm_rev;
    u16 efptr;
    u32 pef;
    u32 swpinfo;                     /* Only used for switches */
    u32 src_ops;
    u32 dst_ops;
    struct rio_switch *rswitch;      /* RIO switch info */
    struct rio_driver *driver;       /* RIO driver claiming this device */
    struct device dev;               /* LDM device structure */
    struct resource riores[RIO_MAX_DEV_RESOURCES];
    u16 destid;
};
```

Figure 4: struct rio_dev

```
struct rio_net {
    struct list_head node;           /* node in list of networks */
    struct list_head devices;        /* list of devices in this net */
    struct list_head mports;         /* list of ports accessing net */
    struct rio_mport *hport;         /* primary port for accessing net */
    unsigned char id;                /* RIO network ID */
};
```

Figure 5: struct rio_net

During enumeration, maintenance transactions are used to access the configuration space of devices. A maintenance transaction has two components to address a device, a device ID and a hopcount. The device ID is normally used for endpoint devices to determine if they should accept a packet. It is a requirement for all devices to ignore the device ID and accept any transaction during enumeration. Switches are a different case, however, as they do not implement a device ID. Transactions which reach a switch device must have their hopcount set appropriately. If a maintenance transaction with a hopcount of 0 reaches a switch, then the switch will process the packet against its own configuration space. If a maintenance transaction has a hopcount greater than 0, then the switch decrements the hopcount in the packet and forwards it along according to the route set for the corresponding device ID in the packet.

The enumeration process walks the network depth first. Like PCI enumeration, this is easily implemented by recursion. When a device is found, the Host Device ID Lock Register is written to ensure that the enumerator has exclusive enumeration ownership of the device. The device's capabilities are then queried to determine if it is a switch or endpoint device.

If the device is an endpoint, it is allocated a new unique device ID and this value is written to the endpoint. A new `rio_dev` is allocated and initialized.

If the device is a switch, its vendor and device ID are queried against a table of known RapidIO switches. A switch table entry has a set of switch routing operations which are specific to the located switch. The routing operations are used to read and write route entries in the switch. New `rio_dev` and `rio_switch` structures are then allocated and initialized.

Enumeration past a switch device is accomplished by iterating over each active switch port on the switch. For each active link, a route to a fake device ID (0xFF for 8-bit systems and 0xFFFF for 16-bit systems) is written to the route table. The algorithm recurses by calling itself with hopcount + 1 and the fake device ID in order to access the device on the active port. While traversing the network, the current allocated device ID is tracked. When the depth first traversal completes, the recursion unwinds and permanent routes are written into the switch routing tables. The device IDs that were found beyond a switch port are assigned route entries pointing to the port which they were found behind.

When the host has completed enumeration of the entire network it calls `rio_clear_locks()` to clean up. For each device in the system, it writes a magic "enumeration complete" value to the Component Tag Register. This register is essentially a scratch pad register reserved for enumeration housekeeping. After this process, all Host Device ID Lock Registers are cleared. Remote nodes that are to initiate passive discovery of the network wait for the magic value to appear in the Component Tag Register and then begin discovery.

The discovery process is similar to the enumeration process that has already been described. However, the discovery process is performed passively. This means that all devices in the network are traversed without modifying device IDs or routing tables. This is necessary in the case where there are multiple enumeration capable endpoints in the system. Typically, only one or two processors with endpoints will be designated as enumerating hosts. Out of the competing enumeration hosts, only one host can win. The losing hosts and other non-enumerating processors are forced to wait until enumeration is complete. At that point, they may traverse the network to find all devices without disturbing the network configuration. When discovery completes, the Linux

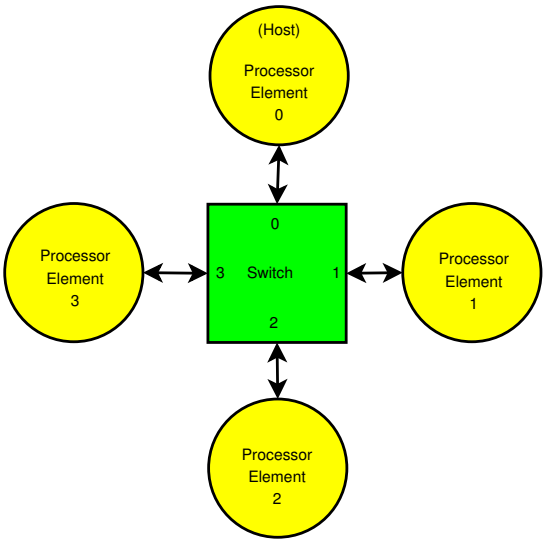

Figure 6: Example RapidIO System

RapidIO subsystem will have a complete view of all RapidIO devices in the network.

In the passive discovery process, the network is walked depth first as with enumeration. However, the existing route table entries are utilized to generate transactions that pass through a switch. When an endpoint device is discovered, a `rio_dev` is allocated but the device ID is retrieved from the value written in the Base Device ID Register. When a switch device is found, discovery iterates over each active switch port as with enumeration. However, in order to generate transactions for devices beyond that switch port, the routing table is scanned for an entry which is routed out that switch port. Using the device ID associated with the switch port, discovery issues a transaction with the associated device ID and a hopcount equal to the number of hops into the network. The process continues in a similar manner as described with enumeration until all devices have been discovered.

3.5 Enumeration and Discovery Example

Figure 6 illustrates a typical RapidIO system. There are four processor elements (PEs) numbered zero through three. Each PE provides RapidIO endpoint functionality and is connected to each of four ports on the switch in the center. PE 0 is the only designated enumerating host in the system and is assigned a host device ID of 0. PEs 1-3 do not perform enumeration, but rather wait for the signal indicating that enumeration has been completed by PE 0.

PE 0 begins enumeration by attempting to obtain the host device ID lock on the adjoining device. The transaction to configuration space is issued with a hopcount of 0 and a device ID of 0xFF. Since the hopcount of the transaction is 0, the switch will process the request and allow PE 0 to obtain the lock. Once the lock is obtained, PE 0 queries the device to learn that it is a switch and allocates `rio_dev` and `rio_switch` structures.

PE 0 queries the switch to determine that there are 4 ports with active links present. PE 0 then begins a loop to iterate over the 4 active ports, skipping the input port which it is using to access the switch device. For each active switch port, PE 0 performs the following:

1. Writes a route entry that assigns device ID 0xFF to the current active switch port.

2. Issues configuration space transactions with a hopcount of 1 to access the devices that are one hop from PE 0:

   - Obtains the host device ID lock for each device.
   - Queries the device to determine that it is an endpoint and allocates a `rio_dev` structure.

- Assigns the next available device ID to the endpoint. PEs 1-3 are assigned device IDs 0x01-0x03, respectively.

3. Assigns route entries corresponding to the switch ports where the PEs were discovered. Route entries for device IDs 0x01-0x03 are assigned to switch ports 1-3, respectively.

After this process completes, PE 0 writes the magic "enumeration complete" value into the Component Tag Register on each device. This is followed by PE 0 releasing the host device ID lock on each device in the system. Once PEs 1-3 detect that enumeration is complete, they are free to begin their discovery process.

3.6 Driver Interface

RapidIO device drivers are provided a specific set of functions to use in their implementation. In order to guarantee proper functioning of the subsystem, drivers may not access hardware resources directly.

Configuration space access is managed similar to configuration space access in the PCI subsystem.

- `rio_config_read_8()`
  `rio_config_read_16()`
  `rio_config_read_32()`
  `rio_config_write_8()`
  `rio_config_write_16()`
  `rio_config_write_32()`

  Read or write a specific size at an offset of a device.

- `rio_local_config_read_8()`
  `rio_local_config_read_16()`
  `rio_local_config_read_32()`
  `rio_local_config_write_8()`
  `rio_local_config_write_16()`
  `rio_local_config_write_32()`

  Read or write a specific size at an offset of the local master port's configuration space.

Several calls handle the ownership and initialization of mailbox and doorbell resources on a master port or remote device.

- `rio_request_outb_mbox()`
  `rio_request_inb_mbox()`

  Claim ownership of an outbound or inbound mailbox, initialize the mailbox for processing of messages, and register a notification callback. The outbound mailbox callback provides a interrupt context event when a message has been sent. The inbound mailbox callback provides an event when a message has been received.

- `rio_release_outb_mbox()`
  `rio_release_inb_mbox()`

  Give up ownership of an outbound or inbound mailbox and unregister notification callback.

- `rio_request_outb_dbell()`

  Claim ownership of a range of doorbells on a remote device. Ownership is only valid for the local processor.

- `rio_request_inb_dbell()`

  Claim ownership of a range of doorbells on the inbound doorbell queue, initialize the doorbell queue, and register a callback. The doorbell callback provides an event when a doorbell within the registered range is received.

- `rio_release_outb_dbell()`

  Give up ownership of a range of doorbells on a remote device.

- `rio_release_inb_dbell()`

  Give up ownership of a range of doorbells on the inbound doorbell queue.

Several calls provide access to doorbell and message queues.

- `rio_send_doorbell()`

  Send a doorbell message to a specific device.

- `rio_add_outb_message()`

  Add a message to an outbound mailbox queue.

- `rio_add_inb_buffer()`

  Add an empty buffer to an inbound mailbox queue.

- `rio_get_inb_message()`

  Get the next available message from an inbound mailbox queue.

3.7 Architecture Interface

Every architecture must provide implementations for a set of RapidIO functions. These functions manage hardware-specific features of configuration space access, mailbox access, and doorbell access.

- `rio_ops.lcwrite()`
  `rio_ops.lcread()`
  `rio_ops.cwrite()`
  `rio_ops.cread()`

  Hardware specific implementations for generation of read and write transactions to configuration space. These master port specific routines are assigned to a `struct rio_ops` which is in turn bound to a `struct rio_mport`. These low-level operations are used by the driver interface configuration space access routines.

- `rio_ops.dsend()`

  Hardware specific implementation for generation of a doorbell write transaction. This master port specific routine is assigned to a `struct rio_mport` and used by the `rio_send_doorbell()` call.

- `rio_hw_open_outb_mbox()`
  `rio_hw_open_inb_mbox()`

  Hardware specific initialization for outbound and inbound mailbox queues.

- `rio_hw_close_outb_mbox()`
  `rio_hw_close_inb_mbox()`

  Hardware specific cleanup for outbound and inbound mailbox queues.

- `rio_hw_add_outb_message()`

  Hardware specific implementation to add a message buffer to the outbound mailbox queue.

- `rio_hw_add_inb_buffer()`

  Hardware specific implementation to add an empty buffer to the inbound mailbox queue.

- `rio_hw_get_inb_message()`

  Hardware specific implementation to get the next available inbound message.

An architecture must also implement interrupt handlers for mailbox and doorbell queue events. Typically, inbound doorbell and mailbox hardware will generate a hardware interrupt to indicate that a message has arrived. Outbound doorbell hardware will typically generate a hardware interrupt when a message has

been successfully sent. The architecture interrupt handler must process the event in an appropriate manner for the message type and acknowledge the hardware interrupt.

For inbound doorbell messages, the handler must extract the doorbell message info and check for a callback that has been registered for the doorbell message it has received. If a callback has been registered (using `rio_request_inb_dbell()`) for a doorbell range that includes the received doorbell message, the callback is executed. The callback indicates the source, destination, and 16-bit info field (the doorbell message) that was received.

A mailbox interrupt handler must execute the registered callback for the mailbox that generated the hardware interrupt. It may be required to do some hardware-specific ring buffer management and must acknowledge the hardware interrupt. The callback is registered using `rio_request_inb_mbox()` or `rio_request_outb_mbox()`

3.8 Device Model

The RapidIO subsystem ties into the Linux Device Model in a similar way to most other device subsystems. A RapidIO bus is registered with the device subsystem and each RapidIO device is registered as a child of that bus. RapidIO specific `match` and `dev_attrs` implementations are provided.

`rio_match_bus()` implementation is a simple device to driver matching implementation. It compares vendor and device IDs of a candidate RapidIO device to determine if a driver will claim ownership of the device.

The `rio_dev_attrs[]` implementation exports all of the common register fields in the `rio_dev` structure to sysfs. In addition to the standard `dev_attrs` sysfs support, a *config* node is exported similar to the same node in the PCI subsystem. It provides userspace access to the 2MB configuration space on each RapidIO device.

RapidIO specific implementations of `probe()`, `remove()`, and driver register/unregister are also provided.

4 RapidIO Messaging Network Driver (*rionet*)

4.1 *rionet* Overview

With the subsystem in place, a driver is still needed to make use of the new functionality. Since the first RapidIO parts available are processors with RapidIO interfaces, a network driver to provide communication over the RapidIO switched fabric makes good sense. The RapidIO messaging model makes this easy since managing outbound and inbound messages is much like a managing a modern descriptor-based network controller.

4.2 *rionet* Features

rionet has the following features:

- Ethernet driver model for simplicity
- Dynamic discovery of network peers using doorbell messages
- Unique MAC address generation based on RapidIO device ID
- Maximum MTU of 4082
- Uses standard RapidIO subsystem message model to work on any RapidIO endpoints with mailboxes and doorbells

4.3 *rionet* Implementation

The *rionet* driver is initialized with a `rio_register_driver()` call. The `id_table` is configured to match all RapidIO devices so that the `rionet_probe()` call will qualify *rionet* devices. The probe routine verifies that the device has mailbox and doorbell capabilities. If the device is mailbox and doorbell capable, then it is added to a list of potential *rionet* peers. If at least one potential peer is found, the local RapidIO device is queried for its device ID. The MAC address is generated by concatenating 3 bytes of a well known Ethernet test network address with a 1 byte zero pad and finally the 2 byte device ID of the local device.

When *rionet* is opened, it requests a range of doorbell messages and registers a doorbell callback to process doorbell events. Two messages, `RIONET_JOIN` and `RIONET_LEAVE`, are defined to manage the active peer discovery process. For each device in the potential peer list, the `RIONET_JOIN` and `RIONET_LEAVE` outbound doorbell resources are claimed. After verifying that the potential peer device has initialized inbound doorbell service, a `RIONET_JOIN` doorbell is sent to it.

The doorbell event handler processes a `RIONET_JOIN` doorbell by doing the following:

1. Adds the originating device ID to the active peer list.
2. Sends a `RIONET_JOIN` doorbell as a reply to the originator.

If a `RIONET_LEAVE` doorbell is received, the originating device ID is removed from the active peer list.

rionet is designed such that it defaults to the maximum allowable MTU size. With a maximum RapidIO message payload of 4096 bytes, the default MTU size is 4082 after allowing for the 14 byte Ethernet header overhead. Due to the inclusion of the RapidIO device ID in the generated MAC address, Ethernet packets in this driver contain all the information required to send the packets over RapidIO.

The `hard_start_xmit()` implementation in *rionet* is similar to any standard Ethernet driver except that it must verify that a destination node is active before queuing a packet. The active peer list that was created during the *rionet* discovery process is used for this verification. The least significant 2 bytes of the destination MAC address are used to index into the active peer list to verify that the node is active. If the node is active, then the packet is queued for transmission using `rio_add_outb_message()`. Housekeeping for freeing of completed skbs is handled using the outbound mailbox transmission complete event. This is similar to how a standard Ethernet driver uses a direct hardware interrupt event for TX complete events.

Ethernet packet reception is also very similar to standard Ethernet drivers. In this case, it is driven from the inbound mailbox event handler. This callback is executed when the hardware mailbox receives an inbound message in its queue. `rio_get_inb_message()` is used to retrieve the next inbound Ethernet packet from the inbound mailbox queue. As skbs are consumed, a ring refill function adds additional empty skbs to the inbound mailbox queue using `rio_add_inb_buffer()`.

The result is an Ethernet compatible driver which can be used to leverage the huge set of TCP/IP userspace applications for development, testing, and deployment. The Ethernet implementation allows routing between *rionet* and wired Ethernet networks, opening up

many interesting application possibilities. It is possible to provide the root file system to nodes via NFS over RapidIO. Coupling this with firmware support for booting over RapidIO, it is possible to boot an entire network of RapidIO processor devices over the RapidIO network.

5 Going Forward

Although the Linux RapidIO subsystem encapsulates much of the hardware functionality of RapidIO, a few areas have been left incomplete. The following features are in development or planned for development.

- In the future, the Linux RapidIO subsystem will add an interface for managing MMIO regions which are mapped to per-device address spaces. As a part of this effort, mmapable sysfs nodes for each region will be exported for use from userspace.

- Although parallel RapidIO provided the first available RapidIO hardware, 16-bit device ID addressable serial RapidIO is the direction where all future hardware is heading. The subsystem is being extended to handle 16-bit device IDs and the serial RapidIO physical layer.

- In order to make use of the standardized error reporting facilities in RapidIO, an interface will be required to register and process Port Write Events. These are unsolicited transactions which are reported to a specified host in RapidIO Typically, they will be used for error reporting.

6 Conclusion

Today, the Linux RapidIO subsystem provides a complete layer for initialization of a RapidIO network and a driver interface for message passing based drivers. The message passing network driver, *rionet*, provides a simple mechanism for application developers to take advantage of RapidIO messaging. As new RapidIO devices are released, *rionet* will serve as a reference driver for authors of new RapidIO device drivers.

References

[1] RapidIO Trade Association. RapidIO Interconnect Specification. `http://www.rapidio.org`.

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*