Freescale Semiconductor
Application Note

Document Number: AN2923
Rev. 0, 09/2005

Using the Serial RapidIO Messaging Unit on PowerQUICC™ III

by Lorraine McLuckie
Freescale Semiconductor, Inc.
East Kilbride, Scotland

Contents

1 Introduction

This application note is provided to assist those engineers wishing to use the serial RapidIO message unit on the PowerQUICC™ III. It has been written for and tested on the MPC8548 processor, but may also apply to other members of the PowerQUICC III family. This document is not intended as a replacement for the MPC8548 reference manual and should be read in conjunction with that document.[1]

This document summarizes the features and uses of the RapidIO messaging unit (including data messages, doorbell messages and inbound port-writes) and provides example code. These extracted code segments are part of a simple application, written to run on top of U-Boot, to prove the functionality of the messaging unit.[2]

The following section descriptions provide an overview of this document.

Section 2, “RapidIO Messaging,” summarizes the relevant aspects of the RapidIO message passing logical specification.

Section 3, “PowerQUICC III Serial RapidIO Messaging Unit,” summarizes the messaging unit implemented on the MPC8548.

Section 4, “Example Software Extracts,” contains software extracts to demonstrate the use of the messaging units on the MPC8548.

Section 5, “References,” provides a list of references corresponding to the numbered notes throughout this document.

Section 6, “Revision History,” gives the revision history for this document.

2 RapidIO Messaging

The RapidIO specification includes a message passing logical specification.[3] In the message passing model, processing elements are only allowed to access memory local to themselves, and communication between processing elements is handled by specialized hardware and controlled by software.

For two processors to communicate, the sending processor uses a local message passing device that reads a section of the sender’s local memory and moves that information across the RapidIO interconnect to the receiving processor’s message passing device. If enabled, the receiver’s message passing device then stores that information in the receiving processor’s local memory and informs the receiving processor that a message has arrived. The receiving processor then accesses its local memory to read the message.

The logical specification defines two kinds of message passing transactions: data messages and doorbell messages.

2.1 Data Messages

The data message operation, consisting of message request and response transactions, is used by a processing element’s message passing hardware to send a data message to other processing elements.

A data message operation can consist of up to 16 individual message transactions. Message transaction data payloads are always multiples of 8 bytes (although not all multiples of 8 bytes are permitted), up to a maximum of 256 bytes. Therefore the maximum data message operation size is 4 Kbytes.

Data messages use the type 11 packet format.[3]

2.2 Doorbells

The doorbell operation, consisting of doorbell request and response transactions, is used by a processing element to send a very short message to another processor. The doorbell transaction contains a 16-bit information field to hold the information destined for the recipient; it does not have a separate data payload field.

Doorbells use the type 10 packet format.[3]

2.3 Port-Writes

Port-writes are not actually part of the RapidIO messaging specification, but are covered in the Input/Output logical specification. However, in the PowerQUICCIII devices, port writes are processed by the message unit and therefore will be covered in this document.

The maintenance port-write operation does not have guaranteed delivery and does not have an associated response. This operation can be used for sending messages, such as error indicators or status information, from a device that does not have an endpoint (for example, a switch). A port-write request to a queue that is full or busy may be discarded.

Port-writes use the type 8 packet format.[3]

3 PowerQUICC III Serial RapidIO Messaging Unit

The serial RapidIO messaging unit on PowerQUICC III is compliant with the message passing logical specification.[3] The messaging unit can be examined as three separate entities: the data message controllers, the doorbell message controller, and the inbound port-write controller. The following sections summarize the details of the messaging unit. Full details can be found in the PowerQUICC III MPC9548 reference manual.[1]

3.1 Comparison with Parallel RapidIO Messaging Unit

Parallel and serial RapidIO are very similar, in terms of software, as the software has no direct control over the RapidIO physical layer. However, there are a few notable differences between the operation of the serial RapidIO messaging unit on the MPC8548 and the parallel RapidIO messaging unit on other PowerQUICC III processors (such as MPC8540 and MPC8560). The serial RapidIO messaging unit features two data message controllers, whereas the parallel RapidIO unit contains only one. Also, in the parallel RapidIO implementation, doorbells are not transmitted through the messaging unit. In the serial RapidIO implementation, doorbells are transmitted in a similar manner to the data messages. Refer to the equivalent application note for the parallel RapidIO implementation for further detail.[4]

3.2 Data Message Controllers

The MPC8548 features two data message controllers. Each of these data message controllers operates independently, and all data message controller registers are duplicated.

The data message controllers provide support for up to 256 bytes for each message transaction (segment), and up to 16 segments per message. This gives support for the maximum message size of 4 Kbytes set by the RapidIO specifications.[3]

Each data message controller can be examined in two parts: the outbox controller and the inbox controller.

3.2.1 Outbox Controller

The outbox controller is responsible for sending messages from local memory. The outbox controller supports two modes of operation: direct and chaining modes.

In direct mode, messages must be sent one at a time. Software must first ensure that the message unit is not busy with a previously initiated message, then program the outbox controller registers with the address, size, and destination of the outgoing message. A 0-to-1 transition of the start bit in the mode register will then cause the data message to be transmitted.

In chaining mode, the software must reserve an area of memory to store message descriptors. The address of this memory is then passed to the outbox controller, which will access it as a circular queue.

When the software wishes to add a new message to the queue, it first ensures that the queue is not full, then reads the address of the next available descriptor from the circular queue. The information regarding the location, size, and destination of the outgoing message is then written into that descriptor. The software must then set the increment bit in the outbound mode register to inform the queuing mechanism that this descriptor should be added to the queue.

On completion of one outgoing message, the outbox controller will automatically start processing the next descriptor in the queue until the queue is empty.

Multi-segment messages are transparent to the software. Messages with payload greater than 256 bytes must be divided into segments to be transmitted across RapidIO. However, the outbox controller automatically handles this segmentation and the software is not aware of (or able to control) it.

The outbox controller also supports the ability to multicast a single-segment 256-byte message to up to 32 different destination IDs. Please note that this is not related to the multicast extensions described in the RapidIO specification.

The outbox can generate an interrupt, from five sources, which can be individually enabled.

- Queue overflow interrupt
- Queue full interrupt
- Queue empty interrupt
- End-of-message interrupt
- Error interrupt

3.2.2 Descriptor Format

In normal chaining mode, message descriptors are used to contain all the information relevant to an outgoing message. The descriptor is added to the outbound queue, and as that descriptor is processed, the information contained in it is transferred into the outbox registers. The format of the descriptor is shown in Table 1.

Table 1. Outbound Message Unit Descriptor Summary

Offset	Descriptor Field	Description
0x00	Source extended address	Contains the source address of message operations with local addresses greater than 32 bits. After the message controller reads the descriptor from memory, this field is loaded into the source extended address register.
0x04	Source address	Contains the source address of the message operation and a snoop enable bit. After the message controller reads the descriptor from memory, this field is loaded into the source address register.
0x08	Destination port	Contains the destination ID and mailbox of the message operation. After the message controller reads the descriptor from memory, this field is loaded into the destination port register.

Table 1. Outbound Message Unit Descriptor Summary (continued)

Offset	Descriptor Field	Description
0x0C	Destination attributes	Contains transaction attributes of the message operation (that is, multicast mode, end-of-message interrupt enable and priority). After the message controller reads the descriptor from memory, this field is loaded into the destination attributes register.
0x10	Multicast group	Contains the logical multicast group. Groups are defined as a list of 32 consecutive device IDs.
0x14	Multicast list	Contains a bit vector list by device ID.
0x18	Double-word count	Contains the number of double-words for the message operation. After the message controller reads the descriptor from memory, this field is loaded into the double-word count register.
0x1C	Reserved	—

3.2.3 Multicast Mode

The data message units support a “multicast” mode, which is not related to the multicast mode described in the multicast extensions to the RapidIO specification.[5] Rather, this mode permits users to efficiently send the same data message to multiple endpoints in the system. This mode can only be used for single-segment (up to 256-byte) data messages.

When multicast mode is enabled (multicast mode is a bit within the destination attributes word of the descriptor, or in the destination attributes register), the message unit looks to two additional registers for information regarding the destinations to which the data message should be sent. These registers are the outbound message multicast group register (OMxMGR) and the outbound message multicast list register (OMxMLR).

The multicast mode splits the destination IDs into groups of 32. For example, group 0 contains target deviceIDs 0,1,..31, and group 1 contains target deviceIDs 32, 33,..63. Each multicast operation can transmit to device IDs within a single group, and the OMxMGR determines which group is to be addressed.

The multicast operation can transmit the data message to any number of the 32 deviceIDs within the chosen group. The OMxMLR contains a single bit corresponding to each deviceID within a group and, if that bit is set, the data messages will be sent to that DeviceID.

For example, if OMxMGR selects group 0, then bit 0 in the OMxMLR corresponds to deviceID 0, and bit 1 corresponds to deviceID 1. If OMxMGR selects group 1, then bit 0 in the OMxMLR corresponds to deviceID 32, and bit 1 corresponds to deviceID 33. It is not valid to have none of the bits set. If OMxMGR is set to 0x00000000, bit 0 is assumed to be set.

3.2.4 Inbox Controller

The inbox controllers are responsible for receiving incoming data messages and storing them in local memory. In the MPC8548, all incoming data messages destined for mailbox 0 will be directed to the inbound controller of message unit 0, and all incoming messages destined for mailbox 1,2 or 3 will be directed to the inbound controller of message unit 1.

The software must reserve an area of memory, per data message controller, which will be used to contain all incoming messages (the outbound message queue contains message descriptors, and the inbound message queue contains actual messages). The address of this memory is passed to the inbox controller which will access this area as a circular queue.

When the software determines that there is an inbound message in the queue, it can read the start address of that message by reading the pointer to the "head" of the circular queue. To release that message from the inbound queue, the software must set the increment bit, which causes the hardware to increment the head (or dequeue) pointer to the next message.

Because of this queueing system, only one inbound message per message controller can be processed by the software at any time. Multiple reads to the head pointer register will return the same value, until the increment bit is set, releasing the message at the top of the queue and causing the "head" pointer to be incremented.

Multi-segment messages are transparent to the software. Messages with data payload greater than 256 bytes must be divided into segments to be transmitted across RapidIO. However, the inbox controller automatically handles the re-assembly of these segments, and the software is not aware of the segmentation and re-assembly process.

The inbox controller can generate an interrupt, from three sources:

- Message-in-queue interrupt
- Queue full interrupt
- Error interrupt

3.2.5 Message Format

Only the payload of the incoming data message appears in the inbound queue. There is no mechanism for the inbox controller to inform the software of the size or the source of the incoming message. If this information is required, it should be encapsulated into the body of the message, in some format understood by the receiver.

3.3 Doorbell Controller

The PowerQUICC III supports the RapidIO doorbell message type, which contains no separate data payload field, but can pass data in a 16-bit information field within the packet header.[3]

3.3.1 Generation of Outbound Doorbells

The mechanism for generating outbound doorbells is similar to the mechanism for generating direct mode data messages.

Software must first ensure that the outbound doorbell unit is not busy with a previously initiated doorbell, then program the outbound doorbell controller registers with the destination and contents of the doorbell message. A 0-to-1 transition of the start bit in the mode register will cause the doorbell to be transmitted.

3.3.2 Inbound Doorbell Reception

The mechanism for receiving doorbells is very similar to the mechanism used for receiving data messages.

The software must allocate an area of memory to contain the inbound doorbell queue. The address of this memory is passed to the inbound doorbell controller which accesses it as a circular queue.

When the software determines that there is a doorbell in the inbound queue, it reads the start address of that message by reading the inbound 'head' (or dequeue) pointer.

To release that message from the inbound queue, the software must set the increment bit, which causes the hardware to increment the 'head' pointer to the next doorbell in the queue.

Because of this queueing system, only one inbound doorbell can be processed by the user at any time. Multiple reads to the 'head' pointer register will return the same value, until the increment bit is set, releasing the doorbell at the top of the queue and causing the 'head' pointer to be incremented.

The doorbell controller can generate an interrupt from three sources:

- Doorbell-in-queue interrupt
- Queue full interrupt
- Error interrupt

3.3.3 Doorbell Format

The incoming doorbells appear in the queue as two 32-bit values, the first containing target information, the second containing source information. See Table 2.

Table 2. Doorbell Entry Format

Offset	Local Memory
0x00	Target info
0x04	Source info

The target information field contains the Target ID field from the received doorbell packet (TID).[3] See Figure 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
—															
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
—								TID							

Figure 1. Target Information Definition

The source information field contains the source ID field from the received doorbell packet (SID) and the 16 bits of information passed in this doorbell (INFO).[3] See Figure 2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
—								SID							
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
INFO															

Figure 2. Source Information Definition

3.4 Inbound Port-Write Controller

The PowerQUICC III serial RapidIO interface does not have the capability to generate port-write transactions. However, it can detect port-writes which arrive from other points in the system (for example, switches).

Unlike the detection of incoming data messages of doorbells, the incoming port-write mechanism has a fixed-length, single-entry queue. To enable the detection of incoming port-writes, the software must reserve 64-bytes of data (64-byte aligned) and program this into the port-write base address register (PIWBAR). When an incoming port-write is detected, the contents of the port-write can be read directly from this memory location. To release the single entry queue, in order that it may detect any other incoming port-writes, the software must write a bit in the mode register to clear the queue.

The inbound port-write mechanism can generate interrupt from two sources:

- Queue full interrupt
- Error interrupt

4 Example Software Extracts

The following section contains a number of software extracts, which demonstrate the procedures required to initialize the inbox, outbox, and port-write controllers, transmit and receive data messages, and transmit and receive doorbells. All of these examples were written as part of a suite of simple applications, which run on U-Boot to demonstrate the operation of RapidIO on MPC8548.[2]

Throughout these examples there are a number of assumptions:

- `local_srio` is a pointer to a structure that contains all the register offsets to the RapidIO registers. This pointer has been initialized to point to the RapidIO registers of the local device.
- These examples do not use interrupts; they are extracted from simple applications that use polling to monitor the inbound and outbound data queues.
- There is a series of constants used of the form REG_FIELD_Shift. These are defined as the number of left shifts required to move a value into the correct FIELD of register REG. For example `ODATR_DtgtRoute_Shift` is defined as the number of left shifts required to place a value into the DtgtRoute field of the ODATR register (= 2). For actual shift values, refer to register definitions in the PowerQUICC III reference manual.[1]
- There are a series of constants used of the form REG_BIT_Mask. These are defined as all zero, with the exception of a set bit in the position corresponding to that bit definition. For example `OSR_MUB_Mask` has a bit set in the position corresponding to the MUB bit in the OSR register (= 0x0000_0004). For actual bit values, refer to register definitions in the PowerQUICC III reference manual.[1]
- It is assumed that all internal addressing is 32-bit; extended addressing is not used.
- It is assumed that device is operating in small transport mode; all deviceIDs are in the range 0–256.
- The data message functions are passed a pointer to the start of the data message controller to be used. Therefore, they can easily be used for either of the data message controllers on the MPC8548.

4.1 Transmitting a Data Message in Direct Mode

In direct mode, the software is directly responsible for programming the outbox registers with the location, size, and destination of the outgoing message, before initiating the transfer.

4.1.1 Enable Direct Mode Outgoing Messages

This function is passed a pointer to the start of the register block of the data message unit to be enabled. To enable direct mode outgoing messages, this software ensures that the message unit is not busy processing a previously initiated message by examining the outbound message unit busy bit (OMnSR[MUB]). It then sets the message unit transfer mode bit (OMnMR[MUTM]) to indicate direct mode. Most of the bits in the mode register (OMnMR), are meaningless in this context. This function also initializes the retry threshold, which is the number of times the message unit will retry a message before an error is indicated.

```
/*define the number of times a message will be retried before generating an error */

#define RETRY_THRESH 4


int enable_ob_msg_dm(data_msg_cntrllr *msg_cntrllr)

{

         /* ensure that the message unit is not already busy */

         if(msg_cntrllr->omsr & OMSR_MUB_Mask )

                  return MESSAGE_UNIT_BUSY;
```

```
        /* set the number of times a message will be retried before
           generating an error */
        msg_cntrllr->omretcr = RETRY_THRESH;

        /* set the mode to direct mode, all other bits in the mode
           register become meaningless */
        msg_cntrllr->ommr = OMMR_MUTM_Mask;

        return SUCCESS;
}
```

4.1.2 Send a Direct Mode Message

The software should only program the registers for a direct transfer if the message unit is not busy transferring a previously initiated message. If the message unit is not busy, the source address, size, and so on of the message are loaded directly into the outbox registers.

A 0 to 1 transition of the message unit start bit (OMMR[MUS]) starts the transaction. The hardware does not clear OMMR[MUS] when it completes the transmission of the message. The software must clear and then set this bit to ensure a 0-to-1 transition.

The function has the option to work in multicast mode. As multicast messaging can only support single segment messages, the code checks for this and returns an error if the message is too large.

```
/* disable the end-of-message interrupt */
#define EOMIE 0

/* the priority of the message transaction */
#define PRIORITY 0

int send_msg_dm(data_msg_cntrllr *msg_cntrllr, u32 multicast, u32 dest_id, u32 bytecount,
                 u32 dest_mailbox, u32 *message, u32 mcgroup, u32 mclist)
{
        u32 attributes;

        /* check if the message unit us already busy*/
        if(msg_cntrllr->omsr & OMSR_MUB_Mask)
                 return MESSAGE_UNIT_BUSY;
```

```
if(multicast)
{
         /* if operating in multicast mode, enable the multicast bit
          multicast mode can only support single segment messages */
         if(bytecount > 0x100)
                  return INVALID_MESSAGE_SIZE;
         dest_id = 0;
         attributes = ((OMDATR_MM_Mask)
                  | (EOMIE << OMDATR_EOMIE_Shift)
                  | (PRIORITY << OMDATR_Dtflowlvl_Shift));
}
else
{
         attributes = ((EOMIE << OMDATR_EOMIE_Shift)
                  | (PRIORITY << OMDATR_Dtflowlvl_Shift));

}



/* fill the registers with the correct info */
msg_cntrllr->omsar = (u32)message | (MSG_SNEN << OMSAR_SNEN_Shift);
msg_cntrllr->omdpr = (dest_id << OMDPR_TgtRoute_Shift)
                  | (dest_mailbox << OMDPR_Mailbox_Shift) ;
msg_cntrllr->omdatr = attributes;
msg_cntrllr->omdcr = bytecount;
msg_cntrllr->ommgr = mcgroup;
msg_cntrllr->ommlr = mclist;

/* ensure that the start bit is clear */
msg_cntrllr->ommr = msg_cntrllr->ommr & (~OMMR_MUS_Mask);

/* ensure that the write to the mode register is complete */
asm("sync");
```

```
        /* start the transfer */
        msg_cntrllr->ommr |= OMMR_MUS_Mask;


        return SUCCESS;


}
```

4.2 Transmitting Data Messages in Chaining Mode

4.2.1 Descriptor Structure

In the example software a data structure type was declared to reference the contents of the descriptors.

```
/* declare a data type which represents the format of a message descriptor */
typedef struct {
        u32     omesar;
        u32     omsar;
        u32     omdpr;
        u32     omdatr;
        u32     ommcgroup;
        u32     ommclist;
        u32     omdcr;
        u32     reserved;
}msg_desc;
```

4.2.2 Enable Normal Chaining Outgoing Messages

This example enables the outbound message unit in normal chaining mode. This function assumes that the memory required for the queue has been allocated elsewhere, and this function receives a pointer to the start of that memory block. This function also accepts a parameter that indicates how many descriptors can be held in that queue.

The starting address of the queue is passed to the outbox controller enqueue and dequeue pointers; thereby allocating it as the circular outbound queue. The final step is to enable the outbound controller with all the relevant mode parameters, including the size of the descriptor queue.

```
/* enable descriptor snooping */
#define DES_SEN 1


/* the constants to enable and disable interrupts */
#define QOIE 0
```

```
#define QFIE 0

#define QEIE 0


/* the number of times a message will be retried before generating an error */

#define RETRY_THRESH 2


int enable_ob_msg_ch(data_msg_cntrllr *msg_cntrllr, u32 *ob_msg_q, u32 desc_in_q)

{

        volatile u32 *tmp;

        u32 cirqsize;


        switch(desc_in_q)

        {

                case 2:             cirqsize = 0;

                                    break;

                case 4:             cirqsize = 1;

                                    break;

                case 8:             cirqsize = 2;

                                    break;

                case 16:            cirqsize = 3;

                                    break;

                case 32:            cirqsize = 4;

                                    break;

                case 64:            cirqsize = 5;

                                    break;

                case 128:           cirqsize = 6;

                                    break;

                case 256:           cirqsize = 7;

                                    break;

                case 512:           cirqsize = 8;

                                    break;

                case 1024:          cirqsize = 9;

                                    break;

                case 2048:          cirqsize = 10;

                                    break;
                default:          return(INVALID_MSG_PARAM);
        }
        /* ensure the message unit is not already busy */
        if(msg_cntrllr->omsr & OMSR_MUB_Mask)
                return MESSAGE_UNIT_BUSY;

        /* put message unit into direct mode, clears any previous chain mode setup */
        msg_cntrllr->ommr = OMMR_MUTM_Mask;

        /* init the head and tail pointers */
        msg_cntrllr->omdqdpar = (u32)ob_msg_q;
        msg_cntrllr->omdqepar = (u32)ob_msg_q;

        /* set the number of times a message will be retried before
           generating an error */
        msg_cntrllr->omretcr = RETRY_THRESH;

        /* set the mode */
        msg_cntrllr->ommr = ((SERVICE_CONTROL << OMMR_SCTL_Shift)
                          | (DES_SEN << OMMR_Des_Sen_Shift)
                          | (cirqsize << OMMR_Cirq_Size_Shift)
                          | (QOIE << OMMR_QOIE_Shift)
                          | (QFIE << OMMR_QFIE_Shift)
                          | (QEIE << OMMR_QEIE_Shift)
                          | (O_EIE << OMMR_EIE_Shift));

        /* ensure that the write to the mode register has completed*/
        asm("sync");

        /* start the message unit */
        msg_cntrllr->ommr |= OMMR_MUS_Mask;

        return SUCCESS;

}
```

4.2.3 Add Message to the Outbound Queue

This function adds a message into the outbound message queue. It checks that the message queue is not full, and then reads the enqueue pointer to determine the address of the next available descriptor. The message information is written to this descriptor, and the message unit increment bit is set to add that descriptor into the queue.

This function also ensures that, if multicast is being used, the message size is no greater than a single segment of 256 bytes.

In this implementation, the end of message interrupt enable and the message priority are set once, by defining constants. These parameters are then common to all messages sent using this routine. It would also be possible to expand the number of parameters passed to the `send_ob_msg_ch` function to include EOMIE and priority, thereby permitting different settings for different messages.

```
/* disable the end-of-message interrupt */
#define EOMIE 0

/* the priority of the message transaction */
#define PRIORITY 0

/* enable message snooping  */
#define MSG_SNEN 1



/* Given the destination, content and length of an outbound message, this function
/* will attempt to create the relevant descriptor and place it in the outbound queue.
/* It will return an error is the message queue was already full. */
int send_msg_ch(data_msg_cntrllr *msg_cntrllr, u32 multicast, u32 dest_id, u32 bytecount,
                 u32 dest_mailbox, u32 *message, u32 mcgroup, u32 mclist)
{
        msg_desc *descriptor;
        u32 attributes;

        /* check for outbound queue full */
        if(msg_cntrllr->omsr & OMSR_QF_Mask)
                return MESSAGE_QUEUE_FULL;

        if(multicast)
        {
                /* if operating in multicast mode, enable the multicast bit
                   multicast mode can only support single segment messages*/
                if(bytecount > 0x100)
                         return INVALID_MESSAGE_SIZE;

                dest_id = 0;
                attributes = ((OMDATR_MM_Mask)
                         | (EOMIE << OMDATR_EOMIE_Shift)
                         | (PRIORITY << OMDATR_Dtflowlvl_Shift));
        }
        else
        {
                attributes = ((EOMIE << OMDATR_EOMIE_Shift)
                         | (PRIORITY << OMDATR_Dtflowlvl_Shift));

        }

        /* get descriptor from queue */
        descriptor = (msg_desc *)msg_cntrllr->omdqepar;

        /* fill the descriptor with the correct info */
        descriptor->omesar = 0;
        descriptor->omsar = (u32)message | (MSG_SNEN << OMSAR_SNEN_Shift);
        descriptor->omdpr = (dest_id << OMDPR_TgtRoute_Shift)
                         | (dest_mailbox << OMDPR_Mailbox_Shift) ;
        descriptor->omdatr = attributes;
        descriptor->omdcr = bytecount;
        descriptor->ommcgroup = mcgroup;
        descriptor->ommclist = mclist;



        /* ensure that the writes to the descriptor are complete */
        asm("sync");
        /* add the descriptor to the queue */
        msg_cntrllr->ommr |= OMMR_MUI_Mask;


        return SUCCESS;



}
```

4.3 Receiving Data Messages

4.3.1 Enable Incoming Data Messages

This function enables incoming data messages. The inbound queue is declared externally and this function accepts a pointer to the start of that memory area, and information regarding the size of the queue. The inbound queue holds all the incoming messages (unlike the outbound queue that holds the message descriptors) and must be sized according to the number and maximum size of the messages that will be incoming. This message queue must be double-word (8 byte) aligned.

This function passes the address of the queue memory to the enqueue and dequeue pointers. The inbox controller will access that memory as a circular queue. The inbox controller is then initialized and enabled with the relevant mode parameters, including information about the size of the queue.

```
/* the number of messages in queue before MIQ is set*/
#define MIQ_THRESH 0



/* enable snooping */
#define IBMSG_SEN 1


/* enable/disable the interrupts */
#define IB_QFIE 0
#define IB_MIQIE 0


int  enable_ib_msgs(data_msg_cntrllr *msg_cntrllr, u8 *message_q, u32 max_ib_msg,
                          u32 msgs_in_q)
{
        u32 frmsize, qsize;
        switch(max_ib_msg)
        {
                case 8:             frmsize = 2;
                                    break;
                case 16:            frmsize = 3;
                                    break;
                case 32:            frmsize = 4;
                                    break;
                case 64:            frmsize = 5;
                                    break;
                case 128:           frmsize = 6;
                                    break;
                case 256:           frmsize = 7;
                                    break;
                case 512:           frmsize = 8;
                                    break;
                case 1024:          frmsize = 9;
                                    break;
                case 2048:          frmsize = 10;
                                    break;
                case 4096:          frmsize = 11;
                                    break;
                default:            return(INVALID_MSG_PARAM);
        }

        switch(msgs_in_q)
        {
                case 2:             qsize = 0;
                                    break;
                case 4:             qsize = 1;
                                    break;
                case 8:             qsize = 2;
                                    break;
                case 16:            qsize = 3;
                                    break;
        case 32:          qsize = 4;
                          break;
        case 64:          qsize = 5;
                          break;
        case 128:         qsize = 6;
                          break;
        case 256:         qsize = 7;
                          break;
        case 512:         qsize = 8;
                          break;
        case 1024:        qsize = 9;
                          break;
        case 2048:        qsize = 10;
                          break;
        default:          return(INVALID_MSG_PARAM);
}



/* ensure message unit is not busy with other task*/
if(msg_cntrllr->imsr & IMSR_MB_Mask)
        return MESSAGE_UNIT_BUSY;

/* check if inbound messages are already enabled*/
if(msg_cntrllr->immr & IMMR_ME_Mask)
        return SUCCESS;



/* load the head and tail pointers for the queue */
msg_cntrllr->imfqdpar = (u32)message_q;
msg_cntrllr->imfqepar = (u32)message_q;



/* set the mode */
msg_cntrllr->immr = ((MIQ_THRESH << IMMR_MIQ_Thresh_Shift)
                  | (IBMSG_SEN << IMMR_SEN_Shift)
```

```
                                    | (frmsize << IMMR_Frm_Size_Shift)
                                    | (qsize << IMMR_Cirq_Size_Shift)
                                    | (IB_QFIE << IMMR_QFIE_Shift)
                                    | (IB_MIQIE << IMMR_MIQIE_Shift)
                                    | (IB_EIE << IMMR_EIE_Shift));

        /* ensure that this update has been completed */
        asm("sync");

        /* enable the inbound messages */
        msg_cntrllr->immr |= IMMR_ME_Mask;

        return SUCCESS;
}
```

4.3.2 Read Address of Data Message from Inbound Queue

This example polls the inbound status register to determine if there is an inbound message waiting to be read. If there is a message waiting, it reads start address.

On return from this function, assuming that there was a message to be read, the `message_ptr` parameter will contain its address. However, it is important to note that this message has not been copied or removed from the queue. This message is still at the top of the queue and any further calls to this function will continue to return the same address until this message is released.

```
int get_ib_msg(data_msg_cntrllr *msg_cntrllr, u32 ** message_ptr)
{
        /* check if there is an inbound message in the queue */
        if(msg_cntrllr->imsr & IMSR_MIQ_Mask)
        {
                /* read the incoming message address from the tail pointer */
                *message_ptr = (u32*)msg_cntrllr->imfqdpar;
                return MESSAGE_READ;
        }
        else
                return INBOUND_QUEUE_EMPTY;
}
```

4.3.3 Release Data Message from Inbound Queue

Once the software has processed an incoming message and has no further use for it, the software must release that message from the queue by setting the mailbox increment bit.

```
void release_ib_msg(data_msg_cntrllr *msg_cntrllr)
{
        /* release this message */
        msg_cntrllr->immr |= IMMR_MI_Mask;
}
```

4.4 Transmitting Doorbells

This function, to transmit a doorbell message, accepts two parameters: the destination ID to which the doorbell should be sent, and the 16 bits of data to be sent. The function ensures that the outbound doorbell unit is not busy, loads the information about the doorbell into the appropriate registers, then starts the doorbell unit to transmit the doorbell message.

```
/* enable/disable end of doorbell interrupt */
#define EODIE 0

/* transaction flow priority. 0 = low priority */
#define DTFLOWLVL 0

int send_db(u32 dest, u32 data)
{
        /* ensure that the doorbell unit is not busy */
        if(local_srio->odsr & ODSR_DUB_Mask)
                return DOORBELL_UNIT_BUSY;

        /* clear start bit */
        local_srio->odmr &= (~(ODMR_DUS_Mask));
        asm("sync");

        /* initialise the information about the doorbell*/
        local_srio->oddpr = (dest << ODDPR_TGTROUTE_Shift);
        local_srio->oddatr = ((EODIE << ODDATR_EODIE_Shift)
                        | (DTFLOWLVL << ODDATR_DTFLOWLVL_Shift)
                        | ((data & 0xFFFF) << ODDATR_INFO_Shift));
        asm("sync");

        /* start the doorbell unit */
        local_srio->odmr |= ODMR_DUS_Mask;

        return SUCCESS;

}
```

4.5 Receiving Doorbells

4.5.1 Enable Incoming Doorbells

This function enables incoming doorbell messages. It declares an area of memory for use as an inbound message queue (64 bits for each doorbell) and passes the address of this memory to the enqueue and dequeue pointers. The inbox controller will access that memory as a circular queue. The doorbell controller is then initialized and enabled with the relevant mode parameters.

```
/* number of doorbells in queue before the doorbell in queue
bit is set. DIQ Thresh of 0 = threshold of 1 doorbell */
#define DB_DIQ_THRESH 0

/* set snoop enable for the incoming doorbells */
#define DB_SEN 1

/* number of doorbells in the inbound queue
   Cirq size of 1 = 4 doorbell queue */
#define DB_CIRQ_SIZE 1
#define DB_QUEUE_SIZE 4

/* interrupts enable/disable */
#define DB_QFIE 0
#define DB_DIQIE 0

/* enable the incoming doorbells */
#define DB_DE 1
```

```
int enable_ib_dbs(void)
{
        /* allocate the memory to store the incoming doorbells */
        static u8 doorbell_q[DB_QUEUE_SIZE *8] __attribute__ ((aligned (64)));

        /* check that the doorbell unit is not busy before setting up */
        if(local_srio->odsr & IDSR_DB_Mask)
                return DOORBELL_UNIT_BUSY;

        /* set up the head and tail pointers for the queue. Assume
           allocated memory within 32-bit address space */
        local_srio->idqdpar = (u32)doorbell_q;
        local_srio->idqepar = (u32)doorbell_q;

        /* ensure these instructions have been completed before continuing */
        asm("sync");

        /* set up the mode register with the relevant setting */
        local_srio->idmr = ((DB_DIQ_THRESH << IDMR_DIQ_Thresh_Shift)
                          | (DB_SEN << IDMR_SEN_Shift)
                          | (DB_CIRQ_SIZE << IDMR_Cirq_Size_Shift)
                          | (DB_QFIE << IDMR_QFIE_Shift )
                          | (DB_DIQIE << IDMR_DIQIE_Shift ));

        /* ensure that this has completed before continuing */
        asm("sync");
        /* start the inbound doorbell unit */
        local_srio->idmr |= IDMR_DE_Mask;

        return SUCCESS;

}
```

4.5.2 Read and Release Doorbell from Inbound Queue

The doorbell contains only two 32-bit fields of information. The function below reads both of these fields from the inbound doorbell queue. As this effectively creates a local copy of the doorbell information, the doorbell can immediately be released from the inbound queue.

```
int get_ib_db(u32 *target_info, u32 *source_info)
{
        volatile u32 *doorbell;

        /* determine if the doorbell threshold has been reached */
        if(local_srio->idsr & IDSR_DIQ_Mask)
        {
                /* read the incoming doobell address from the tail pointer */
                doorbell = (u32*)(local_srio->idqdpar);
                *target_info = *doorbell++;
                *source_info = *doorbell;

                /* ensure the doorbell has been read before releasing it */
                asm("sync");

                /* set the increment bit to release this doorbell */
                local_srio->idmr |= IDMR_DI_Mask;

                return DOORBELL_RECEIVED;
        }
        else
        {

                return IB_Q_EMPTY;
        }



}
```

4.6 Detecting Incoming Port-Writes

4.6.1 Enabling Inbound Port-Write Detection

This function allocates a 64-byte (and 64-byte aligned) buffer which will be used to store the information from any incoming port-write operations, and declares a pointer which will enable other functions to access this memory location. After ensuring that the port-write mechanism is not busy, the function loads the address of the buffer into the port-write base address register, initializes the pointer to be used by other functions, sets up the mode with the appropriate parameters, and then enables the incoming port-write detection.

```
static u8 buffer[64] __attribute__ ((aligned (64)));
u8 *portwritebuffer;

/* set up the incoming port write mechanism */
int enable_ib_pw()
{

        /*ensure unit is not busy before chaning anything */
        if(local_srio->ipwsr & IPWSR_PWB_Mask)
                return PW_UNIT_BUSY;

        /*if already enabled, return success */
        if(local_srio->ipwmr & IPWMR_PWE_Mask)
                return SUCCESS;

        /*load the address to which data should be written */
        local_srio->ipwqbar = (u32)buffer;

        /*create a pointer for the code to use */
        portwritebuffer = buffer ;

        /*set the mode */
        local_srio->ipwmr = ((PW_SEN << IPWMR_SEN_Shift)
                          | (PW_QFIE << IPWMR_QFIE_Shift));

        /*ensure that the mode has been set up */
        asm("sync");

        /*enable the inbound port write detection */
        local_srio->ipwmr |= IPWMR_PWE_Mask;

        return SUCCESS;

}
```

4.6.2 Checking for Incoming Port-Writes

An incoming port-write can be detected checking the queue-full bit in the port-write status register. This is shown here in a function to maintain consistency with the remainder of the document.

```
int get_ib_pw(void)
{
        if(local_srio->ipwsr & IPWSR_QF_Mask)
                return IB_PORTWRITE_DETECTED;
        else
                return IB_PORTWRITE_Q_EMPTY;
}
```

4.6.3 Releasing Port-Writes from the Queue

Port-writes are removed from the queue by setting the clear-queue bit in the port-write mode registers. This is shown here in a function to maintain consistency with the remainder of the document.

```
void release_ib_pw(void)
{
        local_srio->ipwmr |= IPWMR_CQ_Mask;
}
```

4.6.4 Reading Information from Port-Writes

The inbound port-writes go into a single entry queue; users can read the information directly from that queue. In the functions above, the `portwritebuffer` pointer has been declared for that purpose.

5 References

1. Freescale Semiconductor, Inc. *MPC8548E PowerQUICC III™ Integrated Host Processor Reference Manual*, Rev. 1, 7/2005, order #MPC8548RM.
2. Open Source Technology Group (OSTG). www.sourceforge.net/projects/u-boot, 2004.
3. RapidIO Trade Association. *RapidIO Interconnect Specification*, Rev 1.2, 06/2002.
4. Freescale Semiconductor, Inc. (Lorraine McLuckie). *Using the RapidIO Messaging Unit on PowerQUICC III™*, Rev 1, 08/2004, order #AN2741.
5. RapidIO Trade Association. *RapidIO Part XI: Multicast Extensions Specification*, Rev 1.3, 06/2004.

6 Revision History

Table 3 provides a revision history for this application note.

Table 3. Document Revision History

Revision Number	Date	Substantive Change(s)
0	09/16/2005	Initial release.

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 2666 8080
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN2923
Rev. 0
09/2005