Freescale Semiconductor
Application Note

AN2741
Rev. 1, 08/2004

Using the 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 RapidIO Message Unit on the PowerQUICC III™. It has been written for and tested on the MPC8540 and MPC8560 processors, but may also apply to other members of the PowerQUICC III family. This document is not intended as a replacement for the PowerQUICC III 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 and doorbell messages), and provides example code. These extracted code segments are part of a suite of simple applications 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 Messaging Unit,” summarizes the messaging unit implemented on the PowerQUICC III.

Section 4, “Example Software Extracts,” contains software extracts to demonstrate the use of the messaging unit on the PowerQUICC III.

2 RapidIO Messaging

The RapidIO specification includes a message passing logical specification.[2] In the message passing model, processing elements are only allowed to access memory which is 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 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.[2]

2.2 Doorbells

The doorbell operation, consisting of doorbell 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.[2]

3 PowerQUICC III Messaging Unit

The RapidIO messaging unit on PowerQUICC III is compliant with the message passing logical specification.[2] The messaging unit can be examined as two separate entities: the data message controller and the doorbell message controller. The following sections summarize the details of the messaging unit. Additional details can be found in the PowerQUICC III Reference Manual.[1]

3.1 Data Message Controller

The data message controller provides support for up to 256 bytes for each message packet (segment), and up to 16 segments per message. This gives support for the maximum message size of 4 Kbytes set by the RapidIO specifications.[2]

The data message controller can be examined in two parts: the outbox controller (outbound mailbox) and the inbox controller (inbound mailbox).

3.1.1 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 directly program the outbox controller registers with the address, size and destination of the outgoing message.

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.

Upon 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 can generate five different interrupts, which can be individually enabled.

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

3.1.2 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 when that descriptor is processed, the information contained in it is transferred into the outbox registers. The format of the descriptor is shown below:

Table 1. Outbound Message Unit Descriptor Summary

Offset	Descriptor Field	Description
0x00	Reserved	—
0x04	Source address	Contains the source address of the message operation. After the message controller reads the descriptor from memory, this field is loaded into the source address register
0x08	Destination port	Contains the destination port of the message operation. After the message controller reads the descriptor from memory, this field is loaded into the destination port register.
0x0C	Destination attributes	Contains transaction attributes of the message operation. After the message controller reads the descriptor from memory, this field is loaded into the Destination Attributes Register
0x10	Reserved	—
0x14	Reserved	—

Table 1. Outbound Message Unit Descriptor Summary (continued)

Offset	Descriptor Field	Description
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	—

3.1.3 Inbox Controller

The inbox controller is responsible for receiving incoming data messages and storing them in local memory.

The software must reserve an area of memory which will be used to contain all incoming messages (the outbound message queue contains message descriptors, the incoming 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 determine 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.

This queueing system means that only one inbound message 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 message at the top of the queue and causing the 'head' pointer to be incremented.

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 inserted into the body of the message by the sending processor, in some format understood by the receiver.

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 three interrupts:

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

3.2 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.[2] Although both outbound and inbound doorbells will be covered in this section, only the inbound doorbells are handled as part of the message unit. The outbound doorbells are created through the address translation and mapping unit (ATMU) part of the RapidIO unit.[1]

3.2.1 Generation of Outbound Doorbells

Outbound doorbells are not generated through the message unit, but through an ATMU window. This is the same mechanism which is used to generate maintenance read/write, NREAD, NWRITE and other transactions. To generate doorbells an ATMU window must be created which maps a certain area of the local memory map to doorbell transactions. After the initialization of this window, writes into that area of memory create doorbell packets using the information in the ATMU registers to provide the destination and so forth.

The operation of the ATMU will not be discussed fully in this document, but will only be discussed as directly relating to the creation of outgoing doorbells. Further information on the setup and operation of the ATMU is available in the PowerQUICC III Reference Manual.[1]

3.2.2 Inbound Doorbell Reception

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

The software must declare 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 determines 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.

This queueing system means that 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 three interrupts:

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

3.2.3 Doorbell Format

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

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.[2]

Table 3. Target Information Definition

Bits	Name	Description
0–23	—	Reserved
24–31	TID	Target ID field from the received doorbell packet

The source information field contains the source ID field from the received doorbell packet, and the 16 bits of information passed in this doorbell.[2]

Table 4. Source Information Definition

Bits	Name	Description
0–7	—	Reserved
8–15	SID	Source ID field from the received doorbell packet
16–31	INFO	Info field from the received doorbell packet

4 Example Software Extracts

The following section contains a number of software extracts, which demonstrate the procedures required to initialize the inbox and outbox 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 PowerQUICC III.[3]

Throughout these examples there are a number of repeating assumptions:

- `rioport` is a pointer to a structure which 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 which use polling to monitor the inbound and outbound data queues.
- There are 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]

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

To enable direct mode outgoing messages, this software ensures that the message unit is not busy processing a previously initiated message (by examining the message unit busy bit), then sets the message unit transfer mode bit (OMR[MUTM]), to indicate direct mode. Most of the bits in the OMR (descriptor snoop enable, circular descriptor queue size, queue overflow interrupt enable, queue full interrupt enable, and queue empty interrupt enable) are meaningless when operating in direct mode. Therefore there is no option to set them in this extract.

```
/* Error interrupt is the only outbox interrupt which has meaning in direct mode */
#define OB_EIE 0

/* setting MUTM bit configures the outbox in direct mode */
#define MUTM 1
```

```
STATUS enable_direct_ob_msgs(void)
{

      /* first, check that the message unit is not busy */
      if(rioport->osr & OSR_MUB_Mask)
               return MESSAGE_UNIT_BUSY;

      /* configure the message controller in direct mode - do not start */
      rioport->omr = ((OB_EIE << OMR_EIE_Shift)
                        |(MUTM  << OMR_MUTM_Shift));

      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 already 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 (OMR[MUS]) starts the transaction. The hardware does not clear OMR[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.

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_direct_msg` function to include EOMIE and priority; thereby permitting different settings for different messages.

```
/* define the value to be programmed into the EOMIE bit, to determine if an interrupt should
be generated at the end of each message transmission */
#define EOMIE 0

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

STATUS send_direct_msg(

      u32 dest_id,
      u32 bytecount,
      u32 dest_mailbox,
      u32 *message)

{

      /* first, check that the message unit is not busy */
      if(rioport->osr & OSR_MUB_Mask)
               return MESSAGE_UNIT_BUSY;

      /* load the start address of the message
      Software must ensure that the message is double-word aligned
      Enable snooping of the local processor for the message */
      rioport->osar = message | OSAR_SNEN_Mask;

      /* load the destination mailbox */
      rioport->odpr = (dest_mailbox << ODPR_Mailbox_Shift);
      /* load the destination ID, message prority, and End-of-message interrupt
      enable */
      rioport->odatr = ((EOMIE << ODATR_Eomie_Shift)
                                           |(PRIORITY << ODATR_Dtflowlvl_Shift)
                                           |(dest_id << ODATR_DtgtRoute_Shift));


      /* odcr register actually contains a double word count, shifted 3 bits left. This is
      equivalent to the byte count, assuming that there are check elsewhere in the software to
      ensure it will only  produce messages of valid sizes. */
      rioport->odcr = bytecount;

      /*ensure that message unit start bit is clear */
      rioport->omr = rioport->omr & (~OMR_MUS_Mask) ;

      /*ensure write to OMR is complete*/
      asm("sync");

      /*then initiate this message transfer */
      rioport->omr = rioport->omr | OMR_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                 reserved1;
         u32                 osar;
         u32                 odpr;
         u32                 odatr;
         u32                 reserved2;
         u32                 reserved3;
         u32                 odcr;
         u32                 reserved4;
}msg_desc;
```

4.2.2 Enable Normal Chaining Outgoing Messages

This example declares a static array to be used as the outbound queue. This outbound message queue will contain a number of message descriptors, each consisting eight 32-bit fields (see Section 3.1.2, “Descriptor Format”). This message queue must be aligned on a 32-byte boundary.

The starting address of this 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.

```
/* in this particular example, create an outbound queue which will contain up to
32 message descriptors */
#define MESSAGES_IN_IBQ 32

/* when informing the Operating Mode Register (OMR) of the number of messages in the queue, an
encoded value is used. Value 4 indicates a 32 message queue */
#define OBMSG_CIRQ_SIZE  4

/* define the descriptor snooping enable bit */
#define DES_SEN 1

/* define the transfer mode. 0=chaining, 1=direct */
#define MUTM 0

/* define these constants to enable or disable the various interrupts */
/* the example application from which this code was extracted does not
used any interrupts */
#define QOIE 0              /* do not enable queue overflow interrupt */
#define QFIE 0              /* do not enable queue full interrupt */
#define QEIE 0              /* do not enable queue empty interrupt */
#define OB_EIE 0            /* do not enable error interrupt */


/* need to set the start bit */
#define MUS 1

STATUS enable_chaining_ob_msgs(void)

{

      /* create a buffer which will be used as the outbound queue. This queue will
      hold the descriptors for the outbound messages. Each descriptor consists eight 32-bit
      values. Software must ensure that each descriptor is aligned on a 32-byte boundary
      */
      static u32 ob_message_q[MESSAGES_IN_IBQ * 8] __attribute__ ((aligned (32)));


      /* before attempting to set up the outbound message unit, ensure that it
      is not already busy with some other task */
      if(rioport->osr & OSR_MUB_Mask)
               return MESSAGE_UNIT_BUSY;


      /* initialise the head and tail (enqueue and dequeue) pointers of the hardware managed
      circular queue. */
      rioport->odqdpar = (u32)ob_message_q;
      rioport->odqepar = (u32)ob_message_q;


      /* enable the message controller */
      rioport->omr = ((DES_SEN << OMR_Des_Sen_Shift)
               |(OBMSG_CIRQ_SIZE << OMR_Cirq_Size_Shift)
               |(QOIE << OMR_QOIE_Shift)
               |(QEIE << OMR_QEIE_Shift)
               |(QFIE << OMR_QFIE_Shift)
               |(OB_EIE << OMR_EIE_Shift)
               |(MUTM  << OMR_MUTM_Shift)
               |(MUS  << OMR_MUS_Shift));

      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.

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_chaining_msg` function to include EOMIE and priority; thereby permitting different settings for different messages.

```
/* define the value to be programmed into the EOMIE bit, to determine if an interrupt should
be generated at the end of each message transmission */
#define EOMIE 0

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

STATUS send_chaining_msg(

      u32 dest_id,
      u32 bytecount,
      u32 dest_mailbox,
      u32 *message)

{

      msg_desc *descriptor;

      #ifdef PARANOID
      /*check that the bytecount is one of the allowed values */
      switch(bytecount)
      {

           case 8:
           case 16:
           case 32:
           case 64:
           case 128:
           case 256:
           case 512:
           case 1024:
           case 2048:
           case 4096:
                break;
           default:
                return ERR_INVALID_PARAMETER;
                break;

      }


      /* the message must be aligned on a double word boundary. If paranoid,
      check for this */
      if(((u32)message & MASK_DOUBLEWORD_ALIGNED) != (u32)message)
                return ERR_INVALID_PARAMETER;

      #endif /* PARANOID */

      /* before adding a message to the outbound queue, first ensure that the queue is not full
```

```
        */
        if(rioport->osr & OSR_QF_Mask)
                return OUTBOUND_QUEUE_FULL;

        /* get the address of the descriptor to use */
        descriptor = (msg_desc *)(rioport->odqepar);

        /* fill this descriptor with all the information we need
        osar field contains source address, and a bit which will enable snooping of the local
        processor to retrieve the data */
        descriptor->osar = (u32)message | OSAR_SNEN_Mask;

        /* odpr field sets the destination mailbox number */
        descriptor->odpr = (dest_mailbox << ODPR_Mailbox_Shift);

        /* the odatr field contains the EOMIE bit, which determines if an interrupt should be
        generated when this message is complete; the DFTFLOWLVL field which indicates the priority
        of the transaction and the destination ID to which the message should be sent */
        descriptor->odatr = ((EOMIE << ODATR_Eomie_Shift)
                              |(PRIORITY << ODATR_Dtflowlvl_Shift)
                              |(dest_id << ODATR_DtgtRoute_Shift));

        /* odcr field contains a double-word count, shifted 3 bits left. Assuming that the
        software will only produce messages of valid sizes the bytecount can be written directly
        into the double-word count field. */
        descriptor->odcr = bytecount;

        /* ensure that the initialisation of the descriptor fields is complete before loading
        that descriptor into the queue */
        asm("sync");

        /* inform the outbound message unit that a new descriptor has been added to the outbound
        queue */
        rioport->omr = rioport->omr | OMR_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 holds all the incoming messages (unlike the outbound queue which holds the message descriptors), and must be sized according to the number and maximum size of the messages which will be incoming. This message queue must be double-word (8 byte) aligned.

This function declares an area of memory for use as an inbound message queue 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 inbox controller is then initialized and enabled with the relevant mode parameters.

```
/* although the inbound controller is capable of handling max message of 4kbytes,
it is possible to limit the size of message it can accept. This is set in the Inbound
Mode Register using an encoded value representing the size. 0x0A = max 4k messages. Also
need to know the max size of incoming message to correctly reserve memory */
#define IBMSG_FRM_SIZE 0x0A
```

```
#define MAX_MSG 4096

/* similarly the number of messages in the inbound queue must be set. The IMR uses a coded value
to represent the number of messages (4=32 messages in queue). The actual number is required
when declaring the memory. */
#define IBMSG_CIRQ_SIZE 4
#define MESSAGES_IN_IBQ 32

/* define the snoop enable bit to be set */
#define SEN 1

/* define the mailbox enable bit to be set */
#define ME 1


/* the application from which this code is extracted does not use interrupts */
#define QFIE 0              /*do not enable queue full interrupts */
#define MIQIE 0             /*do not enable message-in-queue interrupts */
#define IB_EIE 0            /*do not enable error interrupts */


STATUS enable_ib_msgs(void)
{

      /* declare an array of bytes, equal in size to the max message size multiplied by the
      number of messages. Must be aligned on double-word boundary */
      static u8 message_q[MAX_MSG * MESSAGES_IN_IBQ] __attribute__ ((aligned (8)));

      /* before attempting to set up the inbound message unit, ensure that it is not already
      busy with some other task */
      if(rioport->osr & ISR_MB_Mask)
               return MESSAGE_UNIT_BUSY;

      /*set up the head and tail pointers for the queue. */
      rioport->ifqepar = (u32)message_q;
      rioport->ifqdpar = (u32)message_q;

      /*set up the inbound mode register with the relevant setting */
      rioport->imr = ((SEN << IMR_Sen_Shift)
                        |(IBMSG_FRM_SIZ << IMR_Frm_Size_Shift)
                        |(IBMSG_CIRQ_SIZE << IMR_Cirq_Size_Shift)
                        |(QFIE << IMR_QFIE_Shift )
                        |(MIQIE << IMR_MIQIE_Shift )
                        |(IB_EIE << IMR_EIE_Shift )
                        |(ME  << IMR_ME_Shift)) ;

      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 to be read, it determines its 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.

```
STATUS get_ib_msg(u32 ** message_ptr)
{

      /*if there is a message in the queue, fill message_ptr with its address */
      if(rioport->isr & ISR_MIQ_Mask)
      {
               *message_ptr = (u32 *)rioport->ifqdpar;
               return DETECTED_INCOMING;
      }
      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()
{

      /* release the message at the head of the queue */
      rioport->imr = rioport->imr | IMR_MI_Mask;

}
```

4.4 Transmitting Doorbells

4.4.1 Creating a Window for Outgoing Doorbells

Doorbells are sent using a mechanism which is completely different from the sending of data messages, and more like the creation of maintenance writes or NWRITEs. To enable the sending of doorbells, an ATMU window (defined by base address, translation address, and attributes registers) needs to be created. In the application from which this code is extracted, RapidIO Outbound Window 8 is used for this purpose, but any window could be used. In this example, the destination ID for the doorbell is initially setup as 0. However, this is re-programmed with the correct destination ID in the `send_db` function below.

It is assumed that the `BASE_ADDRESS_FOR_DOORBELLS` corresponds to an area of the memory map, mapped to the RapidIO interface by a local area window, and not already covered by another RapidIO outbound window. Refer to the PowerQUICC III Reference Manual for additional information on local area window and RapidIO outbound window setup.[1]

```
/*define the parameters which will be used to set up the attributes register for the outbound
doorbell window */
#define DB_WINDOW_ENABLE 1          /* this window will be enabled */
#define PRIORITY_LOW 0              /* priority of packets will be set to low */
#define PCI 0                       /* do not follow PCI transaction ordering rules */
#define RIO_NREAD 4                 /* any reads within this window will generate NREAD
                                             transactions */
#define RIO_DOORBELL 2              /* any writes within this window will generate
                                             DOORBELL transactions */
#define WINDOW_SIZE_4k 0x0B         /* this window size is 4k, the minimum window size*/
```

```
void setup_db_window(void)
{

      /* Base Address of the memory area which will be mapped to doorbells*/
      rioport->rowbar8 = BASE_ADDRESS_FOR_DOORBELLS >> Address_to_BA_Shift;

      /* for a doorbell window, the translation address window only contains the
      destination id. In this software, initially set this destination id to 0 and
      the function to send the message will set this up for every doorbell sent */
      rioport->rowtar8 = 0;

      /* setup the window attributes using the definitions above */
      rioport->rowar8 = ((DB_WINDOW_ENABLE << ROW_En_Shift)

                                    |(PRIORITY_LOW << ROW_Tflowvl_Shift)
                                    |(PCI << ROW_PCI_Shift)
                                    |(RIO_NREAD << ROW_ReadTType_Shift)
                                    |(RIO_DOORBELL << ROW_WriteTType_Shift)
                                    |(WINDOW_SIZE_4k << ROW_WindSize_Shift)) ;


}
```

4.4.2 Sending Doorbells

This function assumes that the `setup_db_window` function has already been called to set up the outbound window. This software function expects two parameters: the destination ID of the processor to which the doorbell should be sent, and the 16 bits of information which should be inserted into the doorbell packet's info field.

The `rowtar` is updated to reflect the new destination ID, then a write is executed into a memory location within this window to create the outgoing doorbell.

```
void send_db(u32 dest_id, u16 data)
{

      u16 *mem;

      /* initialise a pointer to the start of the doorbell window. The actual address
      within that window in meaningless */
      mem = (u16 *)BASE_ADDRESS_FOR_DOORBELLS;

      /*update the rowtar register with the new destination id information */
      rioport->rowtar8 = dest_id << ROW_DEV_ID_Shift;

      /*ensure that the update of the rowtar is complete before writing the data*/
      asm("sync");

      /* the write of the 16-bit data value into the area covered by this window
      generates the outgoing doorbell */
      *mem = data;

}
```

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.

```
/*in this example, inbound doorbell queue can contain 8 doorbells. The DOORBELLS_IN_IBQ
constant is defined to calculate the size of the array required. The OMR_CIRC_SIZE is an
encoded value to inform the doorbell controller of the size of the doorbell queue,
encoded value 2 represents 8 doorbell queue */
#define DB_IN_IBQ 8
#define DB_CIRQ_SIZE 2

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


/*define constants which will enable/disable doorbell interrupts 0=disabled*/
#define DB_QFIE 0
#define DB_DIQIE 0
#define DB_EIE 0

/*define a constant which will be used to enable the incoming doorbells*/
#define DB_DE 1

STATUS enable_ib_dbs(void)
{

      /* declare an array of u32s, 2x number of doorbells in queue
      Must be aligned on double-word boundary */
      static u32 doorbell_q[DB_IN_IBQ * 2] __attribute__ ((aligned (8)));

      /* check that the doorbell unit is not busy before setting up */
      if(rioport->dsr & DSR_DB_Mask)
               return MESSAGE_UNIT_BUSY;

      /*set up the head and tail pointers for the queue. */
      rioport->dqepar = (u32)doorbell_q;
      rioport->dqdpar = (u32)doorbell_q;

      /*set up the doorbell mode register with the relevant setting */
      rioport->dmr = ((DB_SEN << DMR_Sen_Shift)
                        |(DB_CIRQ_SIZE << DMR_Cirq_Size_Shift)
                        |(DB_QFIE << DMR_QFIE_Shift)
                        |(DB_DIQIE << DMR_DIQIE_Shift)
                        |(DB_EIE << DMR_EIE_Shift)
                        |(DB_DE  << DMR_DE_Shift)) ;

      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.

```
STATUS get_ib_db(u32 *target_info, u32 *source_info)
{

      u32 *doorbell;

      if(rioport->dsr & DSR_DIQ_Mask)
      {
          /*read the incoming message address from the dequeue pointer
          fill parameters with doorbell information */
          doorbell = (u32 *)rioport->dqdpar;
          *target_info = *doorbell++;
          *source_info = *doorbell;
          asm("sync");


          /*have effectively made a copy, so set the doorbell
          increment bit to release incoming doorbell from queue */
          rioport->dmr = rioport->dmr |  DMR_DI_Mask;

          return DETECTED_INCOMING;

      }
      /* if no message in incoming queue, return */
      else
          return INBOUND_QUEUE_EMPTY;

}
```

5 References

1. *MPC8540 PowerQUICC III Integrated Host Processor Reference Manual*, Rev. 1, 7/2004.
   or *MPC8560 PowerQUICC III Integrated Communications Processor Reference Manual*, Rev. 1, 7/2004.
2. RapidIO Trade Association, *RapidIO Interconnect Specification*, Rev 1.2, 06/2002.
3. Open Source Technology Group (OSTG), www.sourceforge.net/projects/u-boot, 2004.

6 Revision History

Table 5 provides a revision history for this application note.

Table 5. Document Revision History

Revision Number	Date	Substantive Change(s)
1	08/23/04	Initial release.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.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

Europe, Middle East, and Africa:
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-0047 Japan
0120-191014
+81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
852-26668334

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

AN2741
Rev. 1
08/2004