Application Report
SPRAAD3A–December 2006

RapidIO MQT

Todd Mullanix DSP Software Development System

ABSTRACT

The RapidIO® Message Queue Transport (MQT) allows applications to communicate to other TI DSP processors via serial RapidIO. The RapidIO MQT plugs into the MSGQ module of DSP/BIOS. The RapidIO MQT module uses the messaging logical layer of RapidIO to send MSGQ messages between processors. The MSGQ module manages the communication with the MQTs, off-loading that burden from the application. The application writer must simply configure the RapidIO MQT as described in the document.

The RapidIO MQT does not manage other parts of the RapidIO peripheral (e.g., Direct I/O or Congestion Control), but does not preclude the application from using them.

Contents

Trademarks

RapidIO is a registered trademark of RapidIO Trade Association.

1 Contents of Deliverable

The following are included in the RapidIO MQT installation

- RapidIO MQT application report (this document)
- RapidIO MQT source code, library and a project file to rebuild it.
- Sample application (source and project file) that uses the RapidIO MQT

2 Requirements

The following are needed to build the MQT and example:

- DSP/BIOS 5.21 or higher installed.
- Chip Support Library (CSL) for desired chip.
- Codegen 6.0.1B2 or higher.
- Code Composer Studio (CCS) 3.2 beta 2 or higher.

Before reading the rest of this document, the reader should have an understanding of the MSGQ APIs and configuration. Refer to the *TMS320 DSP/BIOS User's Guide* (SPRU423) and/or *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403) from DSP/BIOS 5.21 or higher.

3 Installation

Untar the package into a directory (e.g., c:\). You must set up two environment variables:

- CSL_INSTALL_DIR: Location of the CSL package (e.g., c:\csl_3_00_10_1)
- DDK_INSTALL_DIR: Location of the installed RapidIO MQT. (e.g., c:\rapidiomqt_<chip>).

The following are the directories that are installed in DDK_INSTALL_DIR\packages\ti\bios\drivers.

Directory	Description
rapidiomqt	Source code for RapidIO MQT.
rapidiomqt\<chip>	Project file, chip-specific items and library
examples\rapidiomqt	Source code for RapidIO MQT example.
examples\rapidiomqt\<board>	Project file, board specific items and binary.

4 Building

Both the RapidIOMQT library and example application can be rebuilt via CCS with the supplied project files.

5 MQT Usage Overview

The RAPIDIOMQT and MSGQ relies on the application to provide a few globally defined variables. They include:

- MSGQ_config (section MSGQ Configuration)
- RAPIDIOMQT_config (section RapidIO MSGQ System Configuration)
- RAPIDIOMQT instance parameters (section MQT Instance Parameters)

The application is also responsible for taking the RapidIO peripheral out of power-save and all the non-messaging configuration portions of the peripheral (refer to the srio_init() function in the example).

6 Direct I/O

The RAPIDIOMQT does not interact with the Direct I/O portion of RapidIO nor does it preclude its usage. An application can use both Direct I/O directly and the RAPIDIOMQT. The Functional Layer of CSL provides management of Direct I/O.

7 MSGQ Configuration

MSGQ requires the application to supply a global variable called MSGQ_config of type MSGQ_Config. Refer to the *TMS320 DSP/BIOS User's Guide* (SPRU423) and *TMS320C6000 DSP/BIOS 5.21 Application Programming Interface (API) Ref Guide* (SPRU403) for additional details. These documents are included with DSP/BIOS 5.21 or higher.

7.1 MSGQ_Config

The following is the MSGQ_Config structure.

```
typedef struct MSGQ_Config {
    MSGQ_Obj           *msgqQueues;        /* Array of message queue handles */
    MSGQ_TransportObj  *transports;        /* Array of transports            */
    Uint16              numMsgqQueues;     /* Number of message queue handles*/
    Uint16              numProcessors;     /* Number of processors           */
    Uint16              startUninitialized;/* First msgq to init             */
    MSGQ_Queue          errorQueue;        /* Receives async transport errors*/
    Uint16              errorPoolId;       /* Alloc error msgs from poolId   */
} MSGQ_Config;
```

The *transports* array holds all MQT instances. To add a RapidIO MQT instance into an application, the user must add an entry into the transports array. More details in the following sections.

7.2 MQT Instance

There is a 1-1 mapping of MQT instances per other processors in the system. A MQT instance communicates with one remote processor. A MQT instance must have a matching MQT instance on the remote processor. The order of the MQT instance is dictated by the DSP/BIOS processor Id of the remote processor that it communicates with. This order is reflected in the transports array in the MSGQ_Config structure.

Note: MSGQ allows sending a message to another thread on the same processor. Messaging on the same processor is handled via the MSGQ APIs and does not need a MQT.

If there is no physical connection between two processors, there must be a nop MQT (MSGQ_NOTRANSPORT) to that processor.

For example, assume a system has 3 processors that communicate to each other via RapidIO.

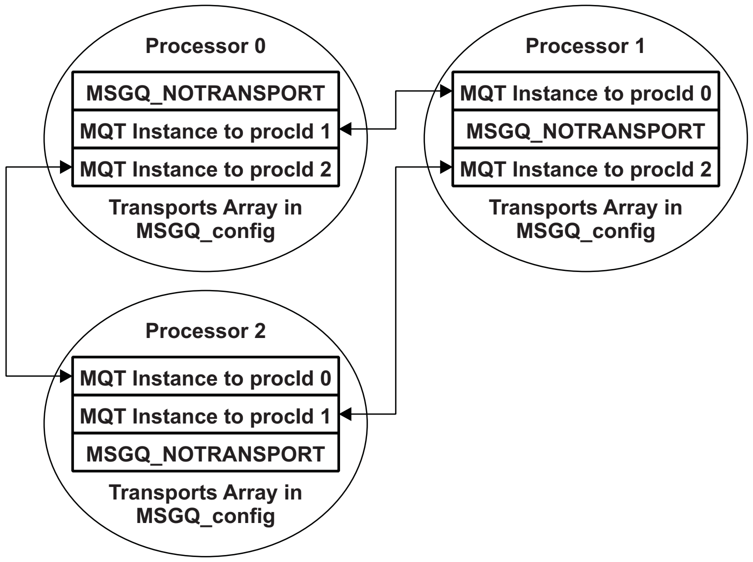

Figure 1. Processors Using RapidIO

7.3 *MSGQ_TransportObj*

The following is the MSGQ_TransportObj structure. When adding a MQT, all fields of this structure must be filled in except the object, which is managed by the RapidIO MQT.

```
typedef struct MSGQ_TransportObj {
    MSGQ_MqtInit        initFxn;    /* Transport init function             */
    MSGQ_TransportFxns *fxns;       /* Transport interface functions       */
    Ptr                 params;     /* Transport-specific setup parameters */
    Ptr                 object;     /* Transport-specific object           */
    Uint16              procId;     /* Processor Id that mqt talks to      */
} MSGQ_TransportObj;
```

The following are the descriptions and the values that the user should use for a RapidIO MQT instance.

Table 1. RapidIO MQT Instance

Field Name	Type	Description	RAPIDIOMQT values
initFxn	MSGQ_MqtInit	MQT's init function	RAPIDIOMQT_init
fxns	MSGQ_TransportFxns	Pointer to the transport's interface functions	&RAPIDIOMQT_FXNS
params	Ptr	MQT's parameters	Refer to section 8 for more details
object	Ptr	State information for the MQT instance	NULL
procId	Uint16	Processor Id that this MQT instance is communicating with	Depends

Note: The MSGQ_config structure and transports array must be persistent for the life of the application. The params structure does not need to be persistent. It is only used during DSP/BIOS initialization.

7.4 Code Example

Here is a code snippet for adding a RapidIO MQT into the MSGQ_config variable. This code assumes this is processor 0 of a three processor system.

Note: The structure of params1 and params2 in the below snippet are discussed in Section 10.

```
#define NUMPROCESSORS 3
static MSGQ_TransportObj transports[NUMPROCESSORS] =
{ MSGQ_NOTRANSPORT,
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params1, NULL, 1},
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params2, NULL, 2}
};

MSGQ_Config MSGQ_config = {msgQueues,         /* Array of message queues     */
                           transports,        /* Array of transports         */
                           NUMMSGQUEUES,      /* # of message queues in array*/
                           NUMPROCESSORS,     /* # of transports in array    */
                           0,                 /* 1st uninitialized msg queue */
                           MSGQ_INVALIDMSGQ,  /* no error handler queue      */
                           POOL_INVALIDID};   /* allocator id for errors     */
```

8 Buffer Descriptors

The main communication between the RapidIO MQT and the RapidIO peripheral is done via CPPI queues. There are 16 Rx and 16 Tx CPPI queues. Buffer descriptors are used to pass messages between the RapidIO peripheral and the RapidIO MQT. The following is a brief overview of how buffer descriptors are used.

Note: The application code does not need to manage or access the buffer descriptors (the RapidIO MQT does that), but the application writer must be familiar with them to configure the RapidIO MQT properly.

8.1 Buffer Descriptor Structure

The following is a CPPI buffer descriptor:

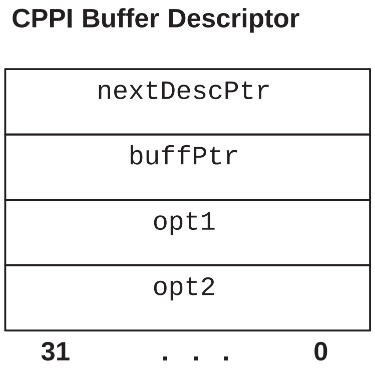

Figure 2. CPPI Buffer Descriptor

Table 2. CPPI Buffer Descriptor

Field Name	Description
nextDescPtr	Pointer to the next buffer descriptor. Used to chain buffer descriptors that are being transmitted or received.
buffPtr	Pointer to a received message in the Rx case, or a message to be sent in Tx case. In either case, for the RapidIO MQT implementation, it points to a MSGQ message.
opt1	Routing information. For example: • Rx: source device id, priority, destination mailbox. • Tx: destination device id, priority, port id, SSIZE, destination mailbox
opt2	Length information. For example • Rx: Start of msg, end of msg, end of queue, ownership, length, completion code • Tx: Start of msg, end of msg, end of queue, ownership, length, completion code, retry count

8.2 Overview

There is a dedicated SRAM location (i.e., the address is 0x02e00000 on the C6455) that is intended for the buffer descriptors. The size of this block is 16384 bytes. The user can select to use a different location if they so desire. The location of the buffer descriptor memory is a configuration parameter (see Section 11, for more details). Cache coherency is an issue if the buffer descriptors are not placed in the reserved location. Refer to Section 19 for more details.

8.3 Rx

During initialization, the RapidIO MQT primes all of the Rx CPPI queues with buffer descriptors that are pointing to empty messages as specified in the configuration. During runtime, incoming RapidIO messages are DMA’d into these empty messages.

8.4 Rx Burst Receives

If the application receives several messages in a burst and exhausts all of the primed Rx buffer descriptors, the RapidIO peripheral drops the msg. Therefore it is important that there are a sufficient number of Rx buffer descriptors allocated.

8.5 Tx

Each RapidIO MQT instance has a Tx CPPI queue that it uses to transmit all its messages to the remote processor.

8.6 Tx Burst Transmits

If the application sends a burst of messages to a remote processor and there are no free buffer descriptors in the associated Tx CPPI Queue, the messages are placed on an internal pending queue. Once a buffer descriptor becomes available, the first message on the pending queue is sent. This is done automatically and does not require any user interaction.

9 MQT Instance Parameters

Each RapidIO MQT instance has its own RAPIDIOMQT_Params. Below is the description of each field in the structure.

```
typedef struct RAPIDIOMQT_Params {
    Uns     maxOutOfOrder;
    Uns     numTxBuffers;
    Uint16  dstDevId;
    Uint8   dstDevId16Bit;
    Uint8   txMbox;
    Uint8   txCppiQueueId;
    Uint8   txCppiQueueWeight;
    Uint8   portId;
    Uint8   retries;
    Uint8   ssize;
} RAPIDIOMQT_Params;
```

Table 3. MQT Instance Parameters

Field Name	Type	Description
maxOutOfOrder	Uns	Maximum depth of messages to wait for an out of order message. See Section 10.1 for more details.
numTxBuffers	Uns	Number of buffer descriptors that this MQT instance will use for transmitting. With more buffer descriptors, there is a smaller chance of messages being placed on a pending queue.
dstDevId	Uint16	Device Id of the destination Rapid IO device.
dstDevId16Bit	Uint8	TRUE: the above dstDevId is 16-bit FALSE: the above dstDevId is 8-bit
txMbox	Uint8	Value to be placed in the mbox field for all the RapidIO messages sent by this MQT instance. Values = 0-3
txCppiQueueId	Uint8	Which Tx CPPI queue that this MQT instance will use. Values = 0-15
txCppiQueueWeight	Uint8	Each Tx CPPI queue has a weighting value. The queues are processed in round-robin. The default is to process one message in a queue, then move to the next queue. This parameter allows a user to specify that the RapidIO peripheral should process more than one message before moving to the next queue. For example, if this parameter is set to ‘5’, the peripheral will process up to 5 messages (if present) on the queue before moving round-robin to the next Tx queue. Values = 0-15
portId	Uint8	When a message is transmitted out this MQT instance, which RapidIO port it will be sent on. Values = 0-3
retries	Uint8	Number of times the RapidIO peripheral will attempt to resend a message.
ssize	Uint8	Segment size for the RapidIO segments. Refer to Section 11.7. 1001b (9) -> 8 byte segments (max MSGQ msg = 128 bytes) 1010b (10) -> 16 byte segments (max MSGQ msg = 256 bytes 1011b (11) -> 32 byte segments (max MSGQ msg = 512 bytes) 1100b (12) -> 64 byte segments (max MSGQ msg = 1024 bytes) 1101b (13) -> 128 byte segments (max MSGQ msg = 2048 bytes) 1110b (14) -> 256 byte segments (max MSGQ msg = 4096 bytes)

Note: The params structure does not have to be persistent during runtime. It is only used during DSP/BIOS initialization (i.e. before the execution of the threads). The configuration parameters cannot be changed during runtime.

9.1 Out of Order Messages

With RapidIO, there is a potential that messages can arrive in a non-FIFO order. The RapidIO MQT corrects the ordering of messages up to a limit. It does not correct lost messages.

The configuration parameter *maxOutOfOrder* specifies how many messages will be delayed while waiting for an out of order message. Take the following examples. Assume the *maxOutOfOrder* parameter is set to 1 and three messages were sent (A, B and C)

Order sent	Order received	Result
A B C	B A C	MQT re-orders the messages correctly
A B C	B C A	MQT drops message A since it was 2 places out of order.

Note: An error message will be log (see Section 20).

Messages may not be coming in a synchronous manner, so the RapidIO MQT sends an internal sync message to the remote processor whenever a message is received out of order. The sync message makes sure that an out of order message is not delayed too long in the case a message is lost. For example, assume the *maxOutOfOrder* parameter is set to 2 and three messages were sent (A, B and C), but A was dropped.

Order sent	Order received	Result
A B C	B C	B and C are sent to the application once the sync reply is received.

Note: An error message will be log (see Section 20).

9.2 Sharing Tx CPPI Queues

Different RapidIO MQT instances can share the same Tx CPPI queue. This configuration may increase message latency since messages may be behind other messages destined for other processors. Using one Tx CPPI queue per MQT instance avoids this issue.

If multiple instances share the same Tx CPPI queue, the weight of the first RapidIO MQT instance is used. The number of Tx buffers, however, is cumulative. For example, assume both RapidIO MQT instances to processor 1 and 2 are using Tx CPPI queue 0 with the following config parameters:

MQT Instance to processor 1	MQT Instance to processor 2
txCppiQueueWeight = 2	txCppiQueueWeight = 1
numTxBuffers = 3	numTxBuffers = 3

Tx CPPI Queue would end up having a weight value of 2. However, the Tx CPPI queue will have a free list of 6 buffer descriptors that are shared between the two MQT instances.

9.3 Code Example

Following is a code snippet for configuring a RapidIO MQT instance. Also shown is the MSGQ_config for completeness. This code assumes this is processor 0 of a three processor system.

```
RAPIDIOMQT_Params params1 = {
    3,      // max num of outstanding our of order msgs
    NUMTXBUFFERS  // numTxBuffers
    0xBEEF, // dstDevId
    TRUE,   // dstDevId16Bit
    3,      // txMbox
    0,      // txCppiQueueId
    2,      // txCppiQueueWeight
    3,      // numTxBuffers
    0,      // portId
    0,      // retries 0 -> infinite
    14      // ssize = 256bytes -> max message size is 4096 bytes
};
RAPIDIOMQT_Params params2 = {
    3,      // max num of outstanding our of order msgs
    NUMTXBUFFERS   // numTxBufffers
    0xFADE, // dstDevId
    TRUE,   // dstDevId16Bit
    2,      // txMbox
    1,      // txCppiQueueId
    1,      // txCppiQueueWeight
    3,      // numTxBuffers
    1,      // portId
    0,      // retries 0 -> infinite
    14      // ssize = 256bytes -> max message size is 4096 bytes
};

#define NUMPROCESSORS 3
static MSGQ_TransportObj transports[NUMPROCESSORS] =
{ MSGQ_NOTRANSPORT,
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params1, NULL, 1},
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params2, NULL, 2}
};

MSGQ_Config MSGQ_config = {msgQueues,         /* Array of message queues     */
                           transports,        /* Array of transports         */
                           NUMMSGQUEUES,      /* # of message queues in array*/
                           NUMPROCESSORS,     /* # of transports in array    */
                           0,                 /* 1st uninitialized msg queue */
                           MSGQ_INVALIDMSGQ,  /* no error handler queue      */
                           POOL_INVALIDID};   /* allocator id for errors     */
```

10 RapidIO MSGQ System Configuration

There are system level parameters for the RapidIO MQT. For example, the Rx CPPI queues are not tied to a specific RapidIO MQT instance. The way these are communicated to the RapidIO MQT is via a required variable called RAPIDIOMQT_config of type RAPIDIOMQT_Config. Below is the RAPIDIOMQT_Config structure. The application must provide the RAPIODIOMQT_config variable in the same way that it provides MSGQ_config variable.

```
typedef struct RAPIDIOMQT_Config {
    Ptr                  bufDescAddr;
    size_t               bufDescSize;
    Uint32               interruptPacingValue;
    Uns                  interruptSrc;
    Uint16               ctrlMsgPoolId;
    RAPIDIOMQT_RxParams *rxQueueParams;
    Uint8                numRxQueueParams;
} RAPIDIOMQT_Config;
```

Field Name	Type	Description
bufDescAddr	Ptr	Base address of the memory that will be used to allocate buffer descriptors. Allocation occurs during initialization. (e.g., reserved buffer descriptor memory on c6455: 0x02e00000)
bufDescSize	size_t	Size of the buffer descriptor space. (e.g.,reserved buffer descriptor memory size on c6455: 16384)
interruptPacingValue	Uint32	Throttling mechanism. How many DMA clock cycles the RapidIO peripheral waits before re-asserting an interrupt (assuming any messages were received or sent).
interruptSrc	Uns	Which interrupt is used for processing the Tx and Rx CPPI queues.
ctrlMsgPoolId	Uint16	A MQT instance needs to send internal messages (e.g., sync, handshakes, locate requests and responses) to its counterpart MQT on the other processor. This parameter specifies from which POOL the MQT instance will allocate. Note, these internal messages are of type RAPIDIOMQT_CtrlMsg. The POOL should have N messages per MQT instance, where is N is equal to or greater than the number of possible concurrent locate requests + the maximum number of out of order messages allowed. The MQT does not send internal messages as the norm during runtime.
rxQueueParams	RAPIDIOMQT_RxParams	Pointer to array of Rx CPPI params. Refer to the RAPIOIOMQT_RxParams below.
numRxQueueParams	Uint8	Number of Rx CPPI queues in the rxQueueParams array.

The following is the RAPIDIOMQT_RxParams structure which is needed in the RAPIDIOMQT_Config structure.

```
typedef struct RAPIDIOMQT_RxParams {
    Uint8   rxCppiQueueId;
    Uint8   rxMbox;
    Uint16  numRxBuffers;
    Uint16  maxMsgSize;
    Uint16  poolId;
} RAPIDIOMQT_RxParams;
```

Field Name	Type	Description
rxCppiQueueId	Uint8	Id of the Rx CPPI queue that will be used to map the below rxMbox to. Values = 0-3
rxMbox	Uint8	Mbox id that will be mapped to the above Rx CPPI Queue. Values = 0-3
numRxBuffers	Uint16	Number of buffer descriptors that the RapidIO ISR will use for receiving for the above Rx CPPI queue. With more buffer descriptors, there a smaller chance of messages being delayed or dropped.
maxMsgSize	Uint16	The maximum size message that will be coming in this Rx CPPI Queue.
poolId	Uint16	The pool that will be used to allocate buffers. The incoming messages will be placed in these buffers.

Note: The above params structures do not have to be persistent during runtime. They are only used during initialization (e.g., before the threads start to execute).

10.1 Mbox <-> Rx CPPI Mapping

The RapidIO MQT allows the use of 4 RapidIO mailbox values. Each incoming message has a mailbox value that is used to determine which Rx CPPI queue it will be placed on. The RapidIO MQT requires that each mailbox value must be placed on a different Rx CPPI queue. For example, all incoming messages with mailbox value 0 are placed on Rx CPPI queue 0, all incoming messages with mailbox value 1 are placed on Rx CPPI queue 1, etc.

Note: The RapidIO MQT’s ISR processes Rx CPPI queue 0 first, then queue 1, etc.

10.2 Code Example

Following is a code snippet for configuring the RapidIO MQT at a system level. Also shown are the MSGQ_config and RapidIO MQT instance configurations for completeness.

```
#define NUMRXMBOX 4
RAPIDIOMQT_RxParams rxParams[NUMRXMBOX] = {
    {0, 0, 2, MSGSIZE, APPPOOLID}, // incoming mbox 0 msgs -> Rx CPPI Queue 0
    {1, 1, 2, MSGSIZE, APPPOOLID}, // incoming mbox 1 msgs -> Rx CPPI Queue 1
    {2, 2, 2, MSGSIZE, APPPOOLID}, // incoming mbox 2 msgs -> Rx CPPI Queue 2
    {3, 3, 2, MSGSIZE, APPPOOLID}  // incoming mbox 3 msgs -> Rx CPPI Queue 3

};

RAPIDIOMQT_Config RAPIDIOMQT_config = {
    (Ptr)0x02e00000,        // bufDescriptorAddr
    16384,                  // bufDescriptorSize
    RAPIDIOMQTISRPACINGVAL, // interruptPacingValue
    0,                      // INTDST0
    MQTCTRLPOOLID,          // ctrlMsgPoolId
    rxParams,               // rxQueueParams
    NUMRXMBOX,              // numRxQueueParams
};


RAPIDIOMQT_Params params1 = {
    3,      // max num of outstanding out of order msgs
    NUMTXBUFFERS,   // numTxBuffers
    0xBEEF, // dstDevId
    TRUE,   // dstDevId16Bit
    3,      // txMbox
    0,      // txCppiQueueId
    2,      // txCppiQueueWeight
    3,      // numTxBuffers
    0,      // portId
    0,      // retries 0 -> infinite
    14,     // ssize = 256bytes -> max message size is 4096 bytes
};
RAPIDIOMQT_Params params2 = {
    3,      // max num of outstanding out of order msgs
    NUMTXBUFFERS,  // numTxBuffers
    0xFADE, // dstDevId
    TRUE,   // dstDevId16Bit
    2,      // txMbox
    1,      // txCppiQueueId
    1,      // txCppiQueueWeight
    3,      // numTxBuffers
    1,      // portId
    0,      // retries 0 -> infinite
    14,     // ssize = 256bytes -> max message size is 4096 bytes
};

#define NUMPROCESSORS 3
static MSGQ_TransportObj transports[NUMPROCESSORS] =
{ MSGQ_NOTRANSPORT,
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params1, NULL, 1},
  {RAPIDIOMQT_init, &RAPIDIOMQT_FXNS, &params2, NULL, 2}
};

MSGQ_Config MSGQ_config = {msgQueues,         /* Array of message queues     */
                           transports,        /* Array of transports         */
                           NUMMSGQUEUES,      /* # of message queues in array*/
                           NUMPROCESSORS,     /* # of transports in array    */
                           0,                 /* 1st uninitialized msg queue */
                           MSGQ_INVALIDMSGQ,  /* no error handler queue      */
                           POOL_INVALIDID};   /* allocator id for errors     */
```

11 Configuration Questions

11.1 How Should I Map My Tx CPPI Queues?

Generally, it is preferable to have one MQT instance per Tx CPPI queue. This prevents messages going to one processor getting delayed by messages going to another processor. If there are more than 16 remote processors in the system that communicate via RapidIO, the application must have the MQT instances share some of the Tx CPPI queues. Refer to section Section 10.2 for the impacts of sharing a Tx CPPI queue.

11.2 How Many Tx Buffer Descriptors Should I Use?

The more Tx Buffer Descriptors you have, the less chance a message will be placed on a pending queue.

11.3 How Should I Map my Rx CPPI Queues?

Each mbox id is mapped to a Rx CPPI queue. For systems with four processors or less, each MQT instances can be configured to use a different txMbox id. This allows the incoming messages for each processor to get mapped to a different Rx CPPI queue. For systems with more than 4 processors, Rx CPPI queues will have incoming messages from multiple processors.

11.4 How Many Rx Buffer Descriptors Should I Use?

There is not an easy answer for this question. It depends on the number of messages flowing through the system, the number of processors in the system, the pacing threshold for the RapidIOMQT ISR, amount of memory in the system, size of the messages, number of Tx buffer descriptors, etc.

If all the Rx buffer descriptors get filled up, the RapidIO peripheral will start rejecting messages. In this case, the message will get dropped.

Note: RapidIOMQT does guarantee that messages are in order, but it does not guarantee the delivery. This was done for performance reasons

So having enough Rx buffer descriptors is very important.

As receiving processor’s interrupt pacing value goes up, more Rx (and Tx) buffer descriptors are needed since the RapidIO MQT ISR latency will be increased.

If the goal of the application is to push as much data across the RapidIO link as possible, messaging is not the best solution. The application should really look into using the Direct I/O logical layer of RapidIO.

11.5 Initialization Handshake

The MQT instances do an initialization handshake with their counterpart MQTs. This handshake allows the MQT instances to set (or reset) internal sequence numbers that are using to maintain in-order delivery. The handshake is managed by the RapidIOMQT PRD (Section 14.3). During start-up, the PRD sends a handshake request to the remote processors. Once all handshakes are completed, the PRD ceases running. Before the handshake is completed with a remote processor, no communication with that processor is allowed. Communication with other processors are that have completed the handshake is allowed even if other processors have not completed the handshake.

There are one direct factor and two indirect factors in determining the timing of the handshake.

- PRD period
- RapidIO peripheral configuration for timeouts
- Number of messages

The smaller the PRD period, the smaller the latency for the handshake to complete once the remote processor is initialized. However, if the RapidIO peripheral configuration for timeouts is large, there can be many outstanding handshake requests. If the number of MSGQ messages in the system is small and the timeout is larger and the PRD period is small, there could be allocation errors. The MQT reports this via MSGQ_MQTERRORALLOC errorType (refer to Section 19).

If the remote processor is not running, the handshake requests transmission timeout in the RapidIO peripheral. This error state is denoted by a bad completion code in the transmit buffer descriptor. The RapidIO MQT reports this via MSGQ_MQTERRORPHYSICAL errorType (refer to Section 19).

11.6 Interrupt Selection

The RapidIO peripheral has multiple interrupts INTDST0-INTDST7. These interrupts can be used for Direct I/O, messaging, port errors and reset.

The user can configure the RAPIDIOMQT to use any one of these interrupts. For example on the C6455, there are 4 different interrupts: INTDST0, INTDST1, INTDST4 and INTDST5 (however INTDST5 is dedicated to reset). To have the RAPIDIOMQT use one of these sources, two things must be done.

- Set the interruptSrc in the RAPIDIOMQT_config structure (Section 10)
- Set the interruptSelectNumber in the HWI configuration (Section 14.4)

The example code uses INTDST0, so the RAPIDIOMQT_config.interruptSrc is set to 0 and the bios.HWI_INT10. interruptSelectNumber is set to 20 (the corresponding selector value for INTDST0). To use INTDST1, the values would be RAPIDIOMQT_config.interruptSrc = 1 and bios.HWI_INT10. interruptSelectNumber = 21.

11.7 Message Sizes

The RAPIDIOMQT does not do segmentation and re-assembly (SAR) of the messages. The maximum size RapidIO message is 4096 bytes. Therefore, the maximum size message that the RAPIDIOMQT supports.

The RapidIO peripheral might break up a message to be transmitted into segments (up to 16). On the receiving side, the peripheral re-assembles the message. The number of segments depends on the size of the message and the requested ssize (segment size) in the RAPIDIOMQT_Params instance configuration.

The supported values for segment size are:

- 1001b (9) -> 8 byte segments (max message = 128 bytes)
- 1010b (10) -> 16 byte segments (max message = 256 bytes)
- 1011b (11) -> 32 byte segments (max message = 512 bytes)
- 1100b (12) -> 64 byte segments (max message = 1024 bytes)
- 1101b (13) -> 128 byte segments (max message = 2048 bytes)
- 1110b (14) -> 256 byte segments (max message = 4096 bytes)

If large messages (e.g., 4096 bytes) are going to be transmitted, the ssize must be set to 1110b. Having a large ssize minimizes the number of segments used to send smaller messages. For example, only two segments are used to send a 512 byte message when ssize is set to 1110b (e.g. 256byte segments). The fewer the segments, the better the performance because of the smaller amount of header overhead and segment formation.

However, there is a downside to having a large ssize, namely memory usage. First some background: on the receiving side, the peripheral looks at the first incoming segment of a message. It determines the amount of memory that is needed to receive the entire message. The peripheral determines this by multiplying the ssize by the number of segments to be received (part of the RapidIO header). The peripheral checks the corresponding Rx CPPI queue to see if there is enough memory. If there is not enough memory, the message is rejected. Otherwise, the segment is processed.

Note: The peripheral does not know size of the last segment (which might be smaller than ssize). This causes the receive buffers, whose size to determined by the maxMsgSize field in RAPIDIOMQT_RxParams, to be multiples of the ssize.

An example can easily show the issue. Have ssize be 1110b (256byte segments, 4096 byte max message). Have the maxMsgSize field be set to 320 bytes. When a 320 byte message is transmitted, the peripheral breaks the message into two segments: one 256 bytes and the other 64 bytes. On the receiving side, the peripheral sees the first segment and determines that it needs 512 bytes to receive the message (ssize * 2 segments). Since the peripheral was primed with 320 byte messages, the transmit fails.

Here are ways to correct the above example:

- Increase the maxMsgSize to a multiple of the ssize, namely 512 bytes. This wastes memory, but has the best performance.
- Decrease the ssize to 1100b (64 byte segments). Now the peripheral makes five 64 byte segments. On the receiving side, the five segments are re-assembled into the 320 byte buffers. Minimal memory waste, but performance impact.

Of course, there can be a compromise in the middle (e.g., set ssize to 1101b (128byte segments) and maxMsgSize to 384 bytes)

12 Interrupt and Peripheral Initialization

The RapidIO MQT initializes the following items of the RapidIO peripheral:

- CPPI Rx and Tx HDP registers
- Rx Mbox <-> Rx CPPI queue mappings
- Mapping the RX and TX CPPI queue to an interrupt (e.g., INTDST0)

The application needs to configure the rest of the RapidIO peripheral. The sample application does this initialization in three places:

- srio_init(): Sets up the RapidIO SerDes registers
- rapidIOTest.tci: Plugs the Rapid messaging ISR and other objects (refer to Section 15).
- configTransports(): Takes the Serial RapidIO (SRIO) peripheral out of power-save

13 RAPIDIOMQT_setCopyFxn API (Advanced Topic)

There is a RAPIDIOMQT_setCopyFxn() API. This allows a user to specify a copy function to be used in the MQT.

Void RAPIDIOMQT_setCopyFxn(RAPIDIOMQT_CopyFxn fxn, Ptr copyHandle);

Parameters

fxn	Function that is called by MQT
copyHandle	Handle that is passed to the copy function

Description

Here is the prototype for the fxn:

typedef Bool (*RAPIDIOMQT_CopyFxn)(Ptr handle, Char *src, Char *dst, Uint16 size);

First some background material.

As part of the RapidIO MQT configuration, you need to specify which pool to use for incoming messages, the worst-case message size and how many messages to prime the receive link list (this is in the RAPIDIOMQT_RxParams structure). The peripheral DMAs incoming data into these messages. Since these messages are then given to the application once they are filled in with incoming data, you need to have many worst-case size messages. This potentially wastes significant memory.

Note: Once a message is received from the peripheral, another worst-case message is allocated and given to the peripheral.

If the application specifies a copy function via RAPIDIOMQT_setCopyFxn, once an application message comes in, the MQT looks in the message to determine its size and its pool id (after it managed cache coherency if needed). The MQT then allocates a “correct-size” message (via MSGQ_alloc) from that pool. The MQT then calls the application specified copy function with the original (src) and new (dst) messages and the size. It also passes the copyHandle. The application can then do a DMA or memcpy into the dst message. This allows for fewer worst-case messages. The down-side is the performance impact of the additional DMA.

The return type of the specified copy function is a Bool. If the copy function copied the data from the dst to the src, the function should return TRUE. If the copy function returns FALSE, it means that the copy did not occur. If the return code is FALSE, the MQT sends the original buffer that the message was received into (e.g., dst) to the application. This return code can be used to minimize the number of copies for large messages (i.e., why copy a received worst-case size message from one large buffer to another).

The MQT never uses the copyHandle directly. The structure of this handle is managed by the application. In the example code, the CSL DAT module is used. No additional information is needed to do the DAT_copy or DAT_wait. If another type of data movement module was used (e.g. EDMA3), the copyHandle could be used to store information (e.g. chan or TCC) to be used when starting the transfer.

Note: All the application messages in the example are the same size, so there is no need to use the copy function, but it is included to demonstrate the concept.

14 Required Statically Configured Objects

There are several statically configured items that the RapidIOMQT requires. They are statically defined to reduce code footprint. The items are discussed below. Note:

Note: The example application statically defines these in rapidiotest.tci.

14.1 LOG Object

The RapidIOMQT has debug trace capabilities. The user must supply a statically configured LOG_Obj called "trace". For example:

```
trace = bios.LOG.create("trace");
trace.bufLen = 1024;
trace.logType = "circular";
```

The LOG object is only needed if the user wants to enable the RAPIDIOMQT trace (refer to Section 16)

14.2 RapidIOMQT SWI

The user must supply a statically created SWI whose function is RAPIDIOMQT_swi and priority is 1. During runtime, the SWI off-loads processing from the RapidIO MQT ISR to minimize interrupt latency. The RapidIO MQT SWI has the following responsibilities:

- Manage cache coherency of incoming messages
- Process incoming locate requests and responses
- Handle incoming application messages (e.g., placing them on the correct message queue destination)
- Manage out of order messages (refer to Section 10.1 for more details).

```
var SWI = bios.SWI.create("RAPIDIOMQT_swiObj");
SWI.priority = 1;
SWI["fxn"] = prog.extern("RAPIDIOMQT_swi");
```

The errorThread is a SWI in the example. This was done for 2 reasons:

1. SWI have higher priority than TSKs.
2. To show that MSGQ can work with SWIs.

14.3 RapidIOMQT PRD

The user must supply a statically created PRD where the configuration is as follows:

```
var PRD = bios.PRD.create("RAPIDIOMQT_prdObj");
PRD.period = 1000;  /* 1 second */
PRD.mode = "one-shot";
PRD["fxn"] = prog.extern("RAPIDIOMQT_prd");
```

During initialization, the MQT instances handshake with the remote side. If the remote side does not respond, the another handshake request is sent. The PRD is responsible for sending the handshake request. It quits running once all the MQTs have completed their startup handshake.

14.4 RapidIOMQT ISR

The user must plug the RapidIO ISR with the function RAPIDIOMQT_isr. For example:

```
bios.HWI_INT10.fxn = prog.extern("RAPIDIOMQT_isr");
bios.HWI_INT10.useDispatcher = true;
bios.HWI_INT10.interruptSelectNumber = 20;
```

14.5 MSGQ and POOL Enabled

The user must enable POOL and MSGQ. For example:

```
bios.MSGQ.ENABLEMSGQ = true;
bios.POOL.ENABLEPOOL = true;
```

15 Example Overview

The example that is included in the package is similar to the standard DSP/BIOS 5.21 (or higher) msgq_tsk2tsk example. This example is intended to run on a two chips with the RapidIO. The biggest change from the 5.21 example is the RapidIO MQT configuration change, the transports array and plugging of the RapidIO ISR. Following is the basic data flow:

```
main()
if processor 0: Open the boss message queue and create the boss thread.
if processor 1: Open the worker message queue and create the worker thread.
Open error message queue and create the error thread.
srio_init to initialize peripheral

workerThread()
    Loop
        MSGQ_get message from the worker thread
        Determine sender
        MSGQ_free message
        Loop number of times requested by the boss
            MSGQ_alloc message
            MSGQ_put message

bossThread()
    MSGQ_locate to locate worker thread
    Loop
        MSGQ_alloc message
        Fill in message with the number of messages to receive.
        MSGQ_put message to worker
        Loop number of times requested by the boss
            MSGQ_get message from the boss queue
            MSGQ_free message

errorThread()
    Loop
        MSGQ_get message from the error queue
        Log MQT error via LOG_printf
```

The example has multiple pools to manage the different types of messages: application, MQT internal control messages and error messages. Having different pools is not required, but makes a system easier to maintain.

The example is intended to run on either board of the EVM (i.e. DSK or mezzanine). To allow a single image to run on either board, there is a GBL_initFxn function in the example that determines which board it is on. Once it determines which board, it sets up the transports table in MSGQ_config accordingly and sets the GBL_procId (if not 0). Using the GBL_initFxn for processor id and transport table management is a short-term work-around. A future release of DSP/BIOS will address single image on multiple processors in a more structured way. An enhancement request has been open (DSP/BIOS tracking number SDSCM00003748). The GBL_initFxn is configured statically in rapidiotest.tci.

Note: GBL_initFxn() runs before DSP/BIOS is initialized.

16 Debug Capabilities

The RapidIO MQT has debug capabilities. If the following compiler option is specified: -d”RAPIDIOMQT_DEBUG”, the RapidIO MQT will include debug information via the LOG module. If this compiler option is not defined, no debug output will be generated. Note: the application needs to supply a LOG_Obj called “trace”

17 Modifying Source

The RapidIO MQT and sample program is shipped with full source, so a user can modify as needed. Some keys points if you modify the files:

- **Hard-coded constants:** There are several hard-coded constants in rapidiomqt.h and _rapidiomqt.h. These values (e.g., RapidIO priority of transmitted messages) could have been configuration parameters, but instead are hard coded constants to minimize footprint and configuration complexity.

18 Cache

Since the CPU Core and the RapidIO peripheral access the same memory, care must be taken to avoid any cache coherency issues. There are two places where cache coherency are an issue: buffer descriptors and messages.

18.1 Buffer Descriptors

As described in Section 9.2, there is a reserved location for buffer descriptors. Since this memory is in non-cacheable, there are no cache coherency issues. The RapidIO peripheral and the core access this memory via the same mechanisms. The buffer descriptors can be placed elsewhere as long as they are not cache-able. The RapidIO MQT does not maintain cache coherency on the buffer descriptors for performance and footprint reasons.

18.2 Messages

If messages are in internal memory, there are no cache coherency issues with the RapidIO peripheral. Similarly, if messages are in external memory and neither L1D nor L2 cache are enabled, there are no cache coherency issues. If the messages that the RapidIO peripheral are interacting with (either sending from or receiving into) are in external memory and there is cache enabled (L1D and/or L2 cache), two actions must be done:

1. Rebuild the RAPIDIOMQT library with the following compiler option: "-dRAPIDIOMQT_CACHECOHERENCY". The MQT performs the necessary cache coherency actions (e.g. write-back cache before giving the message to the peripheral to send out or invalidate cache for incoming messages). The MQT uses DSP/BIOS BCACHE calls.
2. All messages must be aligned on a cache line boundary and their size must be a multiple of a cache line size. So if L1D is enabled (but not L2 cache), the messages that interact with the peripheral must be aligned on a 64 byte boundary and must be a multiple of 64 bytes. If L2 cache is enabled, these values become 128 bytes.

In the example application, both the application messages and the internal MQT control messages are used with the RapidIO peripheral. So they are aligned on a 128 byte boundary and a multiple of 128 bytes. This was unnecessary since all data is currently in internal memory, but was included to help demonstrate the principle.

Note: The application does not take DDR2 out of power-save or configure it. Refer to the CSL examples on how to do this.

19 Errors

Errors may occur in the RapidIO MQT that cannot be communicated to the application via a return code (e.g., errors that occur in the RapidIO MQT ISR). MSGQ has a facility to receive errors messages for MQTs. Refer to the BIOS/DSP documentation for MSGQ_setErrorHandler() and also refer to the RapidIOTest example included in the installation.

The following is the format of the error message:

```
typedef struct MSGQ_AsyncErrorMsg {
    MSGQ_MsgHeader  header;
    MSGQ_MqtError   errorType;
    Uint16          mqtId;
    Uint16          parameter;
} MSGQ_AsyncErrorMsg;
```

Here are the errors that the RapidIO MQT might log and a description of each field.

Note: The mqtId corresponds to the MQT instance that logged the error. If the instance cannot be determine (e.g., the ISR logged the error), a value of 0xFFFF is used for mqtId.

errorType	Description
MSGQ_MQTFAILEDPUT	If a message cannot be placed to the remote or local processor, this error is logged and the message is dropped. Note: this error could signify that an internal message could not be sent also. The msgId of the dropped message is placed in the “parameter” field of the error message.
MSGQ_MQTERRORINTERNAL	Some internal error happened that might affect the health of the system. The unique placement number is in the “parameter” field of the error message. This allows a user to debug the problem.
MSGQ_MQTERRORALLOC	If the MQT cannot allocate a message, this error message is logged. The “parameter” field holds the size of the message that was trying to be allocated.
MSGQ_MQTERRORPHYSICAL	If there is a RapidIO transmission or receiving error, the MQT logs this type of error. The “parameter” field holds the completion code (CC) of the failed transmission. Refer to the RapidIO User Guide for details about the different completion codes.

Note: The user must free all error messages that it receives.

20 Performance

The following benchmarks were done on the 6455EVM running at 1000MHz. The RapidIO linkrate was 3.125Gbps and 1 port was used. All code was compiled with -o2 and no symbols. The RapidIO MQT was built with no debugging trace enabled. L1D and L1P were enabled and the sizes were 32KB. L2 cache was not enabled. All messages are in internal memory and the processors are running at 1GHz.

Note: The CLK ISR was running during the test. This ISR had minimal impact on the results (i.e. < .5%) .

TSK1 allocates a message before entering its main loop. In the loop, TSK1 sends the message to TSK2. TSK2 replies with the same message (i.e. it does not free the message and allocate a new one). This is repeated 10000 times. So for entire test the following APIs are called 20000 times in the application code: MSGQ_put() and MSGQ_get(). TSK1 is timed from when it sends the first message to when it receives the 10000th reply. Time is in CPU cycles. The test was based off the example that is shipped with the RapidIO package.

The below table are results when the two TSKs are on the different processors (i.e. uses RapidIO MQT). Message size does matter in the tests because of the DMA movement done by the RapidIO peripheral and the number of RapidIO segments per message. The TSK based threading model uses semaphores for its MSGQ notification.

Threading Model	#of ping-pongs Messages	Msg Size (bytes)	ISR Pacing	%'CPU Load	Wall Time (in CPU cycles)	Theoretical Time (in CPU cycles)	% Utilization of the Half Duplex Link
1TSK/processor	10,000	64	10	53	67,067,000	4,608,000	6
1TSK/processor	10,000	64	1000	46	78,586,000	4,608,000	7
1TSK/processor	10,000	256	10	42	84,579,000	16,896,000	20
1TSK/processor	10,000	256	1000	42	84,605,000	16,896,000	20
1 TSK/processor	10,000	1024	10	27	141,563,000	67,584,000	48
1 TSK/processor	10,000	1024	1000	27	141,562,000	67,584,000	48
1TSK/processor	10,000	4096	10	11	350,673,000	270,336,000	77
1 TSK/processor	10,000	4096	1000	11	350,664,000	270,336,000	77

The theoretical time is the amount of time (in CPU cycles) to transmit and receive 10000 messages in half duplex. The formula for this is:

(Msg size + RapidIO packet header overhead) * 8bits/byte * # of msgs * 2 / adjusted linkSpeed * CPU speed = Theoretical Time

The adjusted linkSpeed = 8/10 * linkSpeed. This is to account for the 8b/10b encoding scheme.

The " * 2" is because each ping-pong message is sent and received in half duplex.

4096 byte msg size example: (4096 + 128) * 8 * 10,000 * 2 / 2.5Gbps * 1GHz = 270,336,000

The test was repeated using two TSKS on each processor (i.e. two ping-pongs are going over RapidIO now). This was a full-duplex test. Below are the results from this test.

Threading Model	# of Ping-Pongs Messages	Msg Size (bytes)	ISR Pacing	% CPU Load	Wall Time (in CPU cycles)	Theoretical Time (in CPU cycles)	% Utilization of the Full Duplex Link
2 TSKs/processor	10,000	4096	1000	23	400,842,000	270,336,000	67

The theoretical time is still the same since the test allows the RapidIO link to run in full-duplex.

Each RapidIO packet can hold up to 256 bytes. Therefore the larger messages (e.g. 4096 bytes) are broken (by the RapidIO peripheral) into multiple RapidIO packets (e.g. 16 packets for a 4096 byte message). The RapidIO packets header is 8 bytes. From the RapidIOMQT's perspective, message size does not impact performance (except potentially when cache coherency management is included in the MQT).

21 Footprint

Here are the footprint numbers for the RapidIO MQT. These numbers reflect only the MQT’s code and data footprint. They do not include any BIOS APIs (e.g., MSGQ module) or the application’s footprint. There is no debug logging (RAPIDIOMQT_DEBUG is 0) and the optimization is –o2.

```
Section    # bytes
.far       180
.text      9600
.cinit     40
.bss       8
.switch    128
.const     64
Total      10020 bytes
```

22 Endianness

The RAPIDIOMQT does not do any endian or word size conversions on the data or header portion of the messages. Therefore all processors that use the RAPIDIOMQT must have the same type of endianness and word size.

23 Features not Enabled

Several features of the Rapid IO peripheral are not being used by the RapidIO MQT.

- *SourceId matching*. When receiving a message, the Rapid IO peripheral can be configured to only accept messages from a specific device id. The RapidIO MQT does not enable this feature.
- *Re-ordering of the TX CPPI queue*: The Rapid IO peripheral can be configured to process the Tx CPPI queues in non-sequential order (e.g., 11, 5, 7, etc.). The RapidIO MQT does not change the order of the reset values (i.e. 0, 1, 2, … 15)
- *Variable Priority* : The RapidIO MQT always uses the same priority when transmitting a buffer.
- *6-bit mbox ids*: The RapidIO MQT uses 2-bit mbox ids in the RapidIO message. The RapidIO peripheral supports an extended mbox id (6-bits).

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless