Application Report
SPRAB09 – October 2008

Shared Memory Message Queue Transport (MQT) and Pool Modules for C647x

Todd Mullanix, Judah Vang — *TI Software Development Organization*

ABSTRACT

The DSP/BIOS MSGQ module sends and receives structured variable-length messages. By using the Shared Memory Message Queue Transport (SMMQT) and Shared Memory Pool (SMPOOL) described in this document, you can use shared memory to allow multiple processors to communicate with each other via the MSGQ module.

This document describes how to configure SMMQT and SMPOOL. It also covers how to install the associated software package, build the libraries and examples, and run the examples.

This document and the example application focus on C647x devices, specifically the C6474 device (three cores). These devices have shared memory between the cores. However, this document is applicable to any device with multiple cores and shared memory.

Contents

1 Introduction

The Shared Memory Message Queue Transport (SMMQT) and the Shared Memory Pool (SMPOOL) allow processors that share memory to communicate with each other via the DSP/BIOS MSGQ module.

To use the SMMQT and SMPOOL modules, you provide simple static configuration information. The interface with these two modules is then managed internally by MSGQ. You must supply three initialized variables: MSGQ_config, POOL_config, and SMMQT_config. This document describes how to initialize these three variables. It also covers how to install the package, build the libraries and examples, and run the example. In addition, it provides application benchmarks.

This document and the example application focus on C647x devices, specifically the C6474 device (three cores). These devices have shared external memory. However, this document is applicable to any device with multiple cores and shared memory.

2 Modules

Several modules allow communication between processors with shared memory. While the application uses only the MSGQ APIs, the following modules are used internally by MSGQ. Each of these modules is discussed in this application note and included in the SMMQT package.

- **SMMQT.** Shared Memory Message Queue Transport
- **SMPOOL.** Shared Memory POOL
- **CRIT.** Critical Region API and interface
- **HSEMCRIT.** Hardware Semaphore Critical Region Implementation

3 Contents of Deliverable

The following items are included in the SMMQT installation provided with this application note:

- *Shared Memory Message Queue Transport (MQT) and Pool Modules* (SPRAAI2) application note (this document)
- SMMQT, SMPOOL, CRIT, and HSEMCRIT source code, libraries, and the project files to rebuild the libraries
- A sample application (source and project file) that uses the above modules

4 Requirements

The following software must be installed in order to build the MQT and example:

- DSP/BIOS 5.33 or higher
- Code Generation Tools (codegen) 6.0.8 or higher
- CCStudio 3.3 or higher

Before reading the rest of this document, you should have an understanding of the MSGQ APIs and their configuration. Refer to the “Message Queues” section of the *TMS320 DSP/BIOS User's Guide* (SPRU423) and the MSGQ topic in the *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403) from DSP/BIOS 5.33 or higher.

5 Installation

Follow these steps to install the software:

1. Untar the package into a directory (for example, into c:\).
2. Define a DDK_INSTALL_DIR environment variable to point to the location of the installed SMMQT package (for example, c:\mqtciv_1_##, where # is a version number digit).

The following directories are installed in DDK_INSTALL_DIR\packages\ti\bios\drivers:

`smmqt`	Source code for Shared Memory MQT
`smmqt\<chip>`	Project file, chip-specific items, and library
`smpool`	Source code for Shared Memory POOL
`smpool\<chip>`	Project file, chip-specific items, and library
`shared`	Source code for critical region module
`hsemcrit`	Source code for Hardware Semaphore CRIT implementation
`hsemcrit\<chip>`	Project file, chip-specific items, and library
`examples\smmqt`	Source code for Shared Memory MQT example
`examples\smmqt\<board>`	Project file, board specific items, and binary

6 Building

You can rebuild all the libraries and the example application via CCStudio by using the supplied project files.

7 Atomically Accessing Shared Memory

When multiple processors access shared memory, there must be a method to guarantee that the processors have no race conditions with the writing (or reading) of the data in the shared memory. In the SMMQT package, a new module called CRIT can be used to allow atomic reading and writing of shared memory. CRIT stands for “critical region”.

The SMMQT and SMPOOL modules use the CRIT module internally to atomically access shared memory. You only need to supply the CRIT object to the SMMQT and SMPOOL module (see Sections 7.2 and 9). The SMMQT and SMPOOL modules handle all management of CRIT objects.

The subsections that follow discuss the CRIT module. Read them if you want to develop your own critical region implementation or use the CRIT module directly in your application.

7.1 CRIT Module

The means for atomically accessing shared memory is hardware dependent. There are different solutions—for example, a hardware semaphore, Atomic Access Monitors, Test and Set hardware instructions, Peterson's Algorithm, and more. The C6474 has Hardware Semaphores.

The CRIT module consists of an interface and four APIs. Since there is an interface, hardware-specific modules can be written and plugged into the CRIT module.

Critical regions cannot be nested. Once a critical region is entered, it cannot be re-entered. Also care must be taken to minimize the duration spent within a critical region. Other processors can be blocked (that is, they spin while waiting for the critical region resource). Obviously, no blocking calls should made within a critical region.

There can be multiple CRIT objects in a system. For example, the SMMQT uses one CRIT object, while each SMPOOL instance uses a different CRIT object. The following is the structure definition of a CRIT object:

```
typedef struct CRIT_Obj {
    const CRIT_Fxns *fxns;     /* Crit interface functions       */
    Ptr              params;   /* Crit-specific setup parameters */
    Ptr              object;   /* Crit-specific object           */
} CRIT_Obj, *CRIT_Handle;
```

The fields in this structure are as follows:

Field Name	Type	Description
fxn	const CRIT_Fxns	CRIT interface functions
params	Ptr	Critical region implementation-specific parameters
object	Ptr	Critical region implementation-specific object

7.1.1 CRIT Interface (CRIT_Fxns)

There are four interface functions in the CRIT module: open(), close(), enter() and leave(). Their prototypes are as follows:

```
/* Typedefs for function prototypes */
typedef Int   (*CRIT_Open) (Ptr *obj, Ptr params, Bool init);
typedef Void  (*CRIT_Close)(Ptr obj);
typedef Uns   (*CRIT_Enter)(Ptr obj);
typedef Void  (*CRIT_Leave)(Ptr obj, Uns key);

/* CRIT interface functions */
typedef struct CRIT_Fxns {
    CRIT_Open     open;     /* Open a CRIT implementation-specific object  */
    CRIT_Close    close;    /* Close a CRIT implementation-specific object */
    CRIT_Enter    enter;    /* Enter critical region managed by object     */
    CRIT_Leave    leave;    /* Leave critical region managed by object     */
} CRIT_Fxns;
```

- **open()** The open() function initializes any state information or hardware as needed. The return code is used to return status. SYS_OK denotes the open was successful.

- **close()** The close() function frees any resources that were allocated in the open() function. The parameter is the implementation-specific object that was returned from the open() function.
- **enter()** The enter() function performs the steps necessary to allow atomic accesses to shared memory. The parameter is the implementation-specific object that was returned from the open() function. The return code is a key that must be passed into the leave() function.
- **leave()** The leave() function performs the steps necessary to release the atomic access to shared memory. The parameters are the implementation-specific object that was returned from the open() function and the key that was returned from the enter() function.

7.1.2 CRIT APIs

The CRIT APIs map 1-to-1 to the interface functions. They are simply macros that allow for more readable code.

7.1.2.1 CRIT_open()

Syntax status = CRIT_open(handle, params, init);

Parameters

```
CRIT_Handle handle;     /* Handle of CRIT object to open */
Ptr params;             /* Implementation-specific parameters */
Bool init;              /* TRUE: initialize critical region. */
                        /* FALSE: do not initialize critical region. */
```

Return Value Int status;

Reentrant yes

Description Opens a CRIT object. Internally, this API calls the implementation-specific open() function. Each CRIT object can be opened once and only once. SYS_OK denotes success.

The CRIT_Obj should be initialized (init == TRUE) by only one processor. The other processors need to call CRIT_open with init set to FALSE. The processor that initializes the CRIT_Obj must be called first.

7.1.2.2 CRIT_close()

Syntax CRIT_close(handle);

Parameters CRIT_Handle handle;

Return Value Void

Reentrant yes

Description Closes a CRIT object. Internally, this API calls the implementation-specific close() function. Each CRIT object can be closed only once.

7.1.2.3 CRIT_enter()

Syntax key = CRIT_enter(handle);

Parameters CRIT_Handle handle;

Return Value Uns key;

Reentrant yes

Description Starts a critical region section. Internally, this API calls the implementation-specific enter() function. The returned key must be used in the CRIT_leave() API when exiting the critical region.

Once this API returns, all threads and other processors are blocked from entering the critical region managed by the CRIT object (until the CRIT_leave() API is called).

Do not call any blocking functions while in a critical region. The duration between the CRIT_enter() and CRIT_leave() should be kept as short as possible, since depending on the specific implementation, interrupts might be disabled on the processor and potentially other processors.

7.1.2.4 CRIT_leave()

Syntax CRIT_leave(handle, key);

Parameters CRIT_Handle handle;

Uns key;

Return Value Void

Reentrant yes

Description Leaves a critical region section. Internally, this API calls the implementation-specific leave() function. The key returned from CRIT_enter must be used as a parameter to the CRIT_leave() API when exiting the critical region.

7.1.3 CRIT Example

The following is an example of entering and leaving a critical region. The CRIT_Obj has already been opened. Note: This code is a simplified version of the SMPOOL alloc function.

```
Uns key;
/* Enter the critical region */
key = CRIT_enter(smpoolHandle->critHandle);

/* Get a free block from the queue and make sure it is valid */
*buf = QUE_dequeue(&(smpoolHandle->queues[i]));
if (*buf == &(smpoolHandle->queues[i])) {
    /* No message. Return an error */
    status = SYS_EALLOC;
}

CRIT_leave(smpoolHandle->critHandle, key);
```

7.2 HSEMCRIT Module

The HSEMCRIT module implements the CRIT interface and manages the atomic accesses via the Hardware Semaphore capabilities in the C6474.

7.2.1 Enter/Leave

In the enter() function, interrupts are disabled. The leave() function restores interrupts. This is done to prevent deadlock conditions.

7.2.2 Parameters

There is one parameter for a HSEMCRIT instance: num. This parameter is the number of the hardware semaphore.

```
typedef struct HSEMCRIT_Params {
    Uns num;     /* hardware semaphore number */
} HSEMCRIT_Params;
```

7.2.3 Objects

There is no placement requirement for the objects.

7.2.4 HSEMCRIT Example

The following is an example that configures an HSEMCRIT object.

```
HSEMCRIT_Params hsemcritParams = {0};
HSEMCRIT_Obj hsemcritObj;
CRIT_Obj critObj = {&HSEMCRIT_FXNS, &hsemcritParams, &hsemcritObj};
```

8 POOL Configuration

The MSGQ module requires the application to supply a global variable called POOL_config of type POOL_Config. For more details, refer to the *TMS320 DSP/BIOS User's Guide* (SPRU423) and the *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403). These documents are included with DSP/BIOS 5.33 or higher.

8.1 SMPOOL

The Shared Memory Pool (SMPOOL) manages memory in shared memory. Internally it handles all the atomic accesses to the shared memory.

Each SMPOOL instance manages multiple fixed-size buffer buckets. The number of buckets (maximum is 8 different size buckets), number of buffers in each bucket, and the size of each buffer in the buckets is configurable. For instance, a single SMPOOL instance can manage 3 buckets, where one bucket has 10 buffers of size 128 bytes, another bucket has 16 buffers of size 256 bytes, and the last bucket has 4 buffers of size 512 bytes.

Internally the buckets are ordered based on the buffer size. When an allocation occurs, the buckets are checked from the smallest buffer to the largest. Once a bucket is found that has a buffer size that is equal to or greater than the request, allocation comes out of that bucket. If that bucket is empty, the allocation fails. The next larger bucket is not used in this case.

Since the buffers are of a fixed size, the allocation is fast. Once the correct bucket is identified, the allocation is basically a QUE_get(). Also, since the buffers have a fixed size, there is no fragmentation.

8.1.1 Parameters

The following is the SMPOOL parameters structure.

```
typedef struct SMPOOL_Params {
    Uns            numBuckets;
    Ptr           *sharedAddrs;
    size_t        *sharedLen;
    Uint16        *bufferSizes;
    CRIT_Handle    critHandle;
    Uint16         procIdToInit;
}SMPOOL_Params;
```

The following are the descriptions of these parameters:

Field Name	Type	Description
numBuckets	Uns	Number of different size buffers that are managed by the POOL instance. The maximum is 8 buckets.
sharedAddrs	Ptr *	Array of addresses pointing to blocks of memory in shared memory. Each block will be sliced up into smaller pieces. The size of these buffers will be determined by bufferSizes. Address must be aligned on an 8-byte boundary. If the addresses are in a cacheable location, they must be on a cache line boundary.
sharedLen	size_t *	Array of sizes. The sizes correspond to the blocks of memory pointed to by sharedAddrs. Lengths must be a multiple of 8 bytes. If the addresses are in a cacheable location, the lengths must be a multiple of a cache line.
bufferSizes	Uint16	Array of buffer sizes. The sharedAddrs blocks are sliced up into multiple buffers of this size. This array must be in shared memory. Sizes must be a multiple of 8 bytes. If the addresses are in a cacheable location, the buffer sizes must be a multiple of a cache line.
critHandle	CRIT_Handle	Pointer to the CRIT object that manages the critical region for this pool. Each instance needs it own unique critHandle. Internally it will call the CRIT_open for the critHandle. If multiple instances shared the same critHandle, the CRIT object would be opened multiple times, which is not supported.
procIdToInit	Uint16	ProcId of processor to do initialization. This processor must run before the other processors that use the same SMPOOL instance.

The order of the buckets does not matter. For example, the first bucket can have a bufferSize of 512 bytes while the next one has a bufferSize of 128 bytes. There can be multiple buckets with the same bufferSize. During initialization, internally the buckets might be re-ordered (in increasing bufferSize) to allow for faster allocations. Buckets with the same bufferSizes are also combined internally.

8.1.2 Object Placement and Alignment

The SMPOOL objects must be placed in shared memory and be aligned on a cache line boundary.

8.1.3 Code Example

The following example configures a SMPOOL instance on two different processors that act on the same shared memory. The instance has two buckets with 100 buffers of size 128 bytes and 10 buffers of size 512 bytes. For this example, assume that the .sharedmem section is in shared external memory.

```
#pragma DATA_SECTION(hsemcritObj,    ".sharedmem")
#pragma DATA_SECTION(buf0,           ".sharedmem")
#pragma DATA_SECTION(buf1,           ".sharedmem")

/* 128 byte cache line boundary */
#pragma DATA_ALIGN(hsemcritObj,    128)
#pragma DATA_ALIGN(buf0,           128)
#pragma DATA_ALIGN(buf1,           128)

/* This array must be placed in shared memory */
HSEMCRIT_Obj hsemcritObj;

HSEMCRIT_Params hsemcritParams = {
    0   /* Hardware Semaphore num */
};

/* Define CRIT Objects and initialize them */
CRIT_Obj critObj = {&HSEMCRIT_FXNS, &hsemcritParams, &hsemcritObj};

#define NUMBUCKETS         2
#define SIZEBUCKET0BUFS  128
#define SIZEBUCKET1BUFS  512
#define NUMBUCKET0BUFS   100
#define NUMBUCKET1BUFS    10
#define SIZEBUCKET0      SIZEBUCKET0BUFS * NUMBUCKET0BUFS
#define SIZEBUCKET1      SIZEBUCKET1BUFS * NUMBUCKET1BUFS
```

```
/* Bucket buffers */
Char buf0[SIZEBUCKET0];
Char buf1[SIZEBUCKET1];
Ptr sharedAddrs[NUMBUCKETS] = {buf0, buf1};
size_t sharedLengths[NUMBUCKETS] = {SIZEBUCKET0, SIZEBUCKET1};
Uint16 sharedMsgSizes[NUMBUCKETS] = {SIZEBUCKET0BUFS, SIZEBUCKET1BUFS};

SMPOOL_Params sharedParams =
    {NUMSHAREDBUCKETS,           /* Number of buffer pools       */
     (Ptr    *)sharedAddrs,      /* Address of each buffer pool  */
     (size_t *)sharedLengths,    /* length of each sharedAddrs   */
     (Uint16 *)sharedMsgSizes,   /* Msg sizes for each pool      */
     &critObj0,                  /* Handle of CRIT object        */
     0};                         /* Processor 0 should init      */
```

8.2 POOL_Config

Section 8.1 talks about the SMPOOL module and its parameters. This section discusses how to plug the instance into the POOL framework. You do this via the POOL_config variable (of type POOL_Config).

```
typedef struct POOL_Config {
    POOL_Obj   *allocators;     /* Array of allocators               */
    Uint16      numAllocators;  /* Number of allocators in the array */
} POOL_Config;
```

The allocators array holds all POOL instances. To add a SMPOOL instance into an application, you must add an entry into the allocators array. More details are provided in the sections that follow.

8.3 POOL_Obj

The following is the POOL_Obj structure. When you add a SMPOOL instance, all fields of this structure must be filled in. The exception is the object, which might be managed by the POOL (this is implementation dependent).

```
typedef struct POOL_Obj {
    POOL_Init     initFxn;  /* Allocator init function             */
    POOL_Fxns    *fxns;     /* Allocator interface functions       */
    Ptr           params;   /* Allocator-specific setup parameters */
    Ptr           object;   /* Allocator-specific object           */
} POOL_Obj, *POOL_Handle;
```

The following are the descriptions and the values that you should use for a POOL instance.

Field Name	Type	Description	SMPOOL values
initFxn	POOL_Init	MQT's init function	SMPOOL_init
fxns	POOL_Fxns *	Pointer to the POOL's interface functions.	&SMPOOL_FXNS
params	Ptr	POOL's parameters	Must supply the address of a SMPOOL_Params variable. The variable must be initialized (discussed in Section 8.1.1)
object	Ptr	State information for the MQT instance	Must supply the address of a SMPOOL_Obj. It does not need to be initialized. The object must be in shared memory and on a cache line boundary.

Note: The POOL_config structure and allocators array must be persistent for the life of the application. The params structure does not need to be persistent after the threads start running. It is only used during DSP/BIOS initialization.

8.4 Code Example

The following code snippet adds a SMPOOL into the POOL_config variable. The variable sharedParams was discussed in Section 8.1.1.

```
#define NUMALLOCATORS 1

#pragma DATA_SECTION(smpoolObj,      ".sharedmem")

/* 128 byte cache line size  */
#pragma DATA_ALIGN(smpoolObj,      128)

static SMPOOL_Obj smpoolObj;

static POOL_Obj allocators[NUMALLOCATORS] =
   {{SMPOOL_init,                      /* Pool init function       */
     (POOL_Fxns *)&SMPOOL_FXNS,        /* Pool interface functions */
     &sharedParams,                    /* Pool configuration       */
     &smpoolObj}};

POOL_Config POOL_config = {allocators, NUMALLOCATORS};
```

9 MSGQ Configuration

MSGQ requires the application to supply a global variable called MSGQ_config of type MSGQ_Config. See the *DSP/BIOS User Guide* (SPRU423) and *API Reference* (SPRU403) for additional details. These documents are included with DSP/BIOS 5.33 or higher.

9.1 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 an SMMQT instance into an application, you must add an entry into the transports array. More details are provided in the following sections.

9.2 MQT Instance

There is a 1-to-1* mapping of MQT instances on all other processors in the system. An MQT instance communicates with one remote processor. An MQT instance must have a matching MQT instance on the remote processor. The order of the MQT instances 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 an MQT.

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

For example, a system might have three processors that communicate to each other via SMMQT.

9.3 MSGQ_TransportObj

The following is the MSGQ_TransportObj structure. When adding an SMMQT instance, all fields of this structure must be filled in except the object, which is managed by the SMMQT.

```
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 you should use for an SMMQT instance.

Field Name	Type	Description	SMMQT values
initFxn	MSGQ_MqtInit	MQT's init function	SMMQT_init
fxns	MSGQ_TransportFxns*	Pointer to the transport's interface functions	&SMMQT_FXNS
params	Ptr	MQT's parameters	NULL (currently no parameters)
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.

9.4 Code Example

The following code snippet adds an SMMQT into the MSGQ_config variable. This code assumes this is processor 0 of a three-processor system.

```
#define NUMPROCESSORS 3
static MSGQ_TransportObj transports[NUMPROCESSORS] =
{  MSGQ_NOTRANSPORT,
  {SMMQT_init, &SMMQT_FXNS, NULL, NULL, 1},
  {SMMQT_init, &SMMQT_FXNS, NULL, 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 SMMQT Instance Parameters

Currently there are no instance parameters for SMMQT.

11 SMMQT System Configuration

There are system-level parameters for SMMQT. These are communicated to SMMQT via a required variable called SMMQT_config of type SMMQT_Config. **Note:** This variable must be stored in shared memory and aligned on a cache line boundary.

The following is the SMMQT_Config structure.

```
typedef struct SMMQT_Config {
    CRIT_Handle      critHandle;
    Uint16           numProc;
    Uint16           ctrlMsgPoolId;
    Uint16           procIdToInit;
    Uint16          reserved;
    QUE_Obj         queArray[6];
    Char            filler[FILLER_FOR_CACHELINESIZE];
} SMMQT_Config;
```

The following are the descriptions of these parameters.

Field Name	Type	Description
critHandle	CRIT_Handle	Pointer to a CRIT object. All SMMQTs share the same CRIT object.
numProc	Uint16	Number of cores on the device.
ctrlMsgPoolId	Uint16	POOL where SMMQT will allocate internal control messages. This must be a POOL that uses shared memory. The internal messages are of type SMMQT_CtrlMsg. The size of this structure is 64 bytes.
procIdToInit	Uint16	ProcId of the processor to do initialization. This processor must run before the other processors that communicate with SMMQT via the same shared memory.
reserved	Uint16	Reserved. Should not be initialized.
queArrays	QUE_Obj array	MQT will initialize.
filler	Char array	Used to make sure the size of the structure is a multiple of a cache line. Does not need to be initialized.

Note: The SMMQT_config structure must be persistent for the life of the application.

11.1 Placement and Alignment

The SMMQT_config variable must be placed in shared memory and must be aligned on a cache line boundary.

11.2 Code Example

The following example for is a code snippet for configuring the SMMQT at a system level. Also shown are the MSGQ_config and SMMQT instance configurations for completeness. This snippet is for core 0 (processor ID 0) of the three-processor system.

```
#define NUMCORES 3

#pragma DATA_SECTION(SMMQT_config,   ".sharedmem")
#pragma DATA_SECTION(hsemcritObj,    ".sharedmem")

/* 128 byte cache line boundary */
#pragma DATA_ALIGN(SMMQT_config,   128)
#pragma DATA_ALIGN(hsemcritObj,    128)

/* This array must be placed in shared memory */
HSEMCRIT_Obj hsemcritObj;

HSEMCRIT_Params hsemcritParams = {
    0   /* Hardware Semaphore num */
};

/* Define CRIT Objects and initialize them */
CRIT_Obj critObj = {&HSEMCRIT_FXNS, &hsemcritParams, &hsemcritObj};

SMMQT_Config SMMQT_config = {
    &critObj1,           /* Handle of CRIT object */
    NUMCORES,            /* Number of cores, in this case same as processors */
    APPPOOLID,           /* ctrlMsgPoolId         */
    0,                   /* procId of processor to do initialization */
};

static MSGQ_TransportObj transports[NUMCORES] =
{ MSGQ_NOTRANSPORT,
  {SMMQT_init,(MSGQ_TransportFxns *)&SMMQT_FXNS, NULL, NULL, 1},
  {SMMQT_init,(MSGQ_TransportFxns *)&SMMQT_FXNS, NULL, NULL, 2}
};

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

12 Initialization Order

Several items in shared memory must be initialized. Therefore, it is critical that the order of the core bring-up is managed. SMMQT and SMPOOL have configuration parameters that specify which processor is going to be started first. The other cores must wait until this core has finished initializing. The SMMQT and SMPOOL initialization is completed by the time main() is called.

If a bootloader is used, there must be some type of mechanism in place to let the loader know when the application on the initializing core has reached main(). For CCStudio, simply make sure the initializing core reaches main() before starting the other cores.

If the initializing core resets but the chip does not reset, SMMQT and SMPOOL detect this and do not re-initialize the shared memory.

13 Endianism

The SMMQT and SMPOOL do not do any endian conversion to the MSGQ_MsgHeader or payload. All processors using SMMQT and SMPOOL must have the same type of endianism.

14 Configuring Required Static Objects

SMMQT requires several statically-configured DSP/BIOS items, which are discussed below. They are statically defined to reduce code footprint. The example application statically defines these in smmqttest.tci.

14.1 Configuring a LOG Object

The SMMQT has debug trace capabilities. You must supply a statically-configured LOG_Obj called “smmqtLogObj”. For example:

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

14.2 Configuring an SMMQT SWI Object

You must supply the SMMQT module with a statically-created SWI object. For example:

```
swi1 = bios.SWI.create("SMMQT_swiObj");
swi1.fxn = prog.extern("SMMQT_swi");
swi1.priority = 1;
```

The priority of the SWI can be changed as needed. In a TSK-based system, the priority of the SMMQT’s SWI does not really matter. In a SWI-based system, having the SMMQT SWI at the highest priority allows it to process (that is, send to the appropriate message Queue) all the incoming messages.

14.3 Plugging the SMMQT ISR

You must plug the SMMQT ISR with the SMMQT_isr function. For example:

```
bios.HWI_INT5.interruptSelectNumber = 84;
bios.HWI_INT5.fxn = prog.extern("SMMQT_isr");
bios.HWI_INT5.useDispatcher = 1;
```

14.4 Enabling the MSGQ and POOL Modules

You must enable the POOL and MSGQ modules. For example:

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

14.5 Setting the DSP/BIOS Processor ID

The SMMQT assumes that each core's GBL.procId is in the same relative order as its DNUM value assigned by the hardware. For example, if core 0's DSP/BIOS processor ID is 5, then core 1's processor ID must be 6, core 2's must be 7, etc. For example:

```
bios.GBL.PROCID = 0;
```

15 Example Overview

The example included in the package is similar to the standard DSP/BIOS 5.33 (or higher) msgq_tsk2tsk example. This example is intended to run on two cores of a C6474. The biggest change from the 5.33 example is the addition of the SMMQT and SMPOOL configuration, the transports array, and the SMMQT ISR.

Core 0 must be run first. Once main() has been called in core 0, then core 1 can run.

The following is the basic data flow:

```
main()
   if core 0: Open the worker message queue and create the worker thread.
   if core 1: Open the boss message queue and create the boss thread.
   Open error message queue and create the error thread.

workerThread()
   Loop
       MSGQ_get message from the worker queue
       Determine sender
       Send specific number of message to sender

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

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

To allow a single image to run on either board, the GBL_initFxn function in the example determines which core it is running on. Once it determines which core it is on, it sets up the transports table in MSGQ_config accordingly and calls GBL_setProcId() to set the processor ID. The GBL_initFxn is configured statically in rapidiotest.tci.

```
bios.GBL.CALLUSERINITFXN = 1;
bios.GBL.USERINITFXN = prog.extern("configTransports");
```

16 Debug Capabilities

The SMMQT has debug capabilities. If the -d"SMMQT_DEBUG" compiler option is specified, the SMMQT includes debug information via the LOG module. If this compiler option is not specified, no debug output is generated.

Note: The application needs to supply a LOG_Obj called "smmqtLogObj" in order for LOG messages to be received.

17 Cache Concerns

It is expected that you have caching enabled. The SMMQT and SMPOOL modules maintain cache coherency as needed. However; you must make sure the following items are aligned on a cache line boundary:

- SMMQT_config
- SMPOOL objects (smpoolObj in example)
- Messages (appBuf and mqtErrorBuf in example)

Additionally, you must make sure that message sizes are a multiple of the size of a cache line.

The size of the SMMQT_config and SMPOOL_Obj structures must also be a multiple of the cache line size. The "_CACHELINE_SIZE" compile option allows you to specify the cache line size. This compile option should be set to the cache line size where the respective objects are going to reside. If no cache is enabled, set the "_CACHELINE_SIZE" to 0 to minimize the data footprint.

For example, the SMMQT (and SMPOOL) C6474 CCStudio project file specifies the following compile option: '-D_CACHELINE_SIZE=128'. This setting is used because the example application has SMMQT_config (and SMPOOL_Obj) in external memory for the C6474.

Note: The L2 cache line size on the C6474 is 128 bytes.

18 Support for Other Chips

The SMMQT and SMPOOL modules can be ported to other C6000 chips. The following are some of the areas that must be modified to port the modules:

- Create a different critical region implementation.
- Modify the cache line sizes as described in Section 17.
- Manage the resets appropriately in SMPOOL and SMMQT.
- Issue the appropriate interrupts in the SMMQT.

19 Errors

Errors may occur in the Shared Memory MQT that cannot be communicated to the application via a return code. (For example, this includes errors that occur in the SMMQT ISR.) MSGQ has a facility to receive errors messages for MQTs. See the DSP/BIOS documentation for details on MSGQ_setErrorHandler(). Also refer to the smmqttest example included in this 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;
```

The following table shows the errors that the SMMQT 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 determined (for example, 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_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_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.

20 Performance Benchmarks

The following benchmarks were found on the EVM6474 running at 1000 MHz. All code was stored in internal memory after being compiled with -o2 and no symbols. The SMMQT was built with no debugging trace enabled. The L1D and L1P caches were enabled with sizes of 32 KB. The L2 cache was not enabled. The _CACHELINE_SIZE compiler option was set to 64. All messages were in external memory. **Note:** The CLK ISR was running during the test. This ISR had minimal impact on the results (that is, <0.5%).

The boss TSK allocates a message before entering its main loop. In the loop, the boss TSK sends a message to the worker TSK. The worker TSK replies with the same message; it does not free the message and allocate a new one. This is repeated 10,000 times. So, for the entire test, the following APIs are called 20,000 times in the application code: MSGQ_put() and MSGQ_get(). The boss TSK is timed from when it sends the first message to when it receives the 10,000th reply. Time is in CPU cycles. The test was based on the example that is shipped with the SMMQT package.

The 2 TSK/processor benchmark has a boss TSK on each core communicating to a worker TSK on the other core. This test essentially allows both cores to run at a higher level of efficiency, for example because the core is not idle while the boss TSK is waiting for a response.

The table below shows the results using two TSKs on different processors (using SMMQT). The TSK-based threading model uses semaphores for its MSGQ notification.

# of TSK/processor	# of msgs	Msg size	CPU load on both processors	Wall time (in CPU cycles)
1 TSK	10,000	128	54	105,130,000
2 TSK	10,000	128	66	194,740,000

21 Footprint

Here are the footprint numbers for the HSEM, SMMQT, and SMPOOL modules. They do not include any DSP/BIOS APIs (for example, the MSGQ module) or the application's footprint.

These numbers were obtained on the EVM6474 with debug logging off (SMMQT_DEBUG==0). The optimization was set to –o2. The _CACHELINE_SIZE compiler option was set to 64. The values are in bytes.

HSEM		**SMPOOL**		**SMMQT**	
.text	160	.text	1856	.text	2912
.const	16	.const	40	.cinit	12
				.const	20
				.far	16
				.bss	4
Total:	**176**	**Total:**	**1896**	**Total:**	**2964**

22 References

For more information, see the following documentation:

- *TMS320 DSP/BIOS User's Guide* (SPRU423)
- *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403, DSP/BIOS 5.33 or higher)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

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
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265