SYS/BIOS Inter-Processor Communication (IPC) 1.25

User's Guide

Literature Number: SPRUGO6E
September 2012

Preface

About This Guide

This document provides an overview of the Inter-Process Communication (IPC) APIs. This version of this document is intended for use with IPC version 1.25 on targets that use SYS/BIOS.

Intended Audience

This document is intended for users of the IPC APIs and creators of implementations of interfaces defined by IPC modules.

This document assumes you have knowledge of inter-process communication concepts and the capabilities of the processors and shared memory available to your application.

Notational Conventions

This document uses the following conventions:

- When the pound sign (#) is used in filenames or directory paths, you should replace the # sign with the version number of the release you are using. A # sign may represent one or more digits of a version number.
- Program listings, program examples, and interactive displays are shown in a `mono-spaced font`. Examples use **`bold`** for emphasis, and interactive displays use **`bold`** to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, DaVinci, the DaVinci logo, XDS, Code Composer, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, SYS/BIOS, RTDX, Online DSP Lab, DaVinci, eXpressDSP, TMS320, TMS320C6000, TMS320C64x, TMS320DM644x, and TMS320C64x+.

MS-DOS, Windows, and Windows NT are trademarks of Microsoft Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Solaris, SunOS, and Java are trademarks or registered trademarks of Sun Microsystems, Inc.

All other brand, product names, and service names are trademarks or registered trademarks of their respective companies or organizations.

September 10, 2012

Contents

Chapter 1

About IPC

This chapter introduces IPC, a set of modules designed to facilitate inter-process communication.

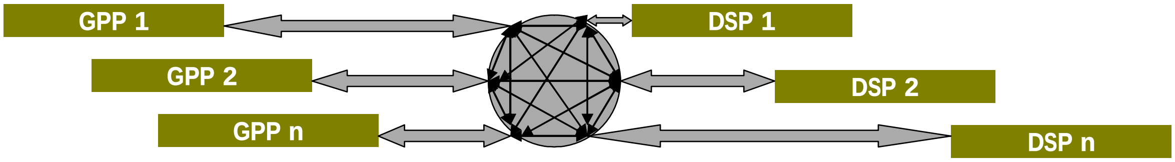
1.1 What is IPC?

IPC is a component containing packages that are designed to allow communication between processors in a multi-processor environment and communication to peripherals. This communication includes message passing, streams, and linked lists. These work transparently in both uni-processor and multi-processor configurations.

IPC is designed for use on processors running SYS/BIOS applications. This is typically a DSP, but may be an ARM device in some cases. Previous versions of SYS/BIOS were called DSP/BIOS. The new name reflects that this operating system can also be use on processors other than DSPs.

IPC can be used to communicate with the following:

- other threads on the same processor
- threads on other processors running SYS/BIOS
- threads on GPP processors running SysLink



IPC was designed with the needs of a wide variety of users in mind. In the interest of providing modules that are usable by all groups, the IPC modules are designed to limit the API actions to the basic functionality required. For example, they do not perform their own resource management. It is the responsibility of the calling thread to manage resources and similar issues that are not managed by the IPC modules.

1.2 Requirements

IPC is installed as part of the Code Composer Studio installation. That installation also installs the versions of XDCtools and SYS/BIOS that you will need.

IPC can be used on hosts running any of the following operating systems:

- Microsoft Windows XP (SP2 or SP3), Vista, or 7
- Linux (Redhat 4 or 5)

If you are installing separately from CCS, see the User_install.pdf file in the <ipc_install_dir>/docs directory for installation information and instructions. This file also provides instructions for building the examples outside the CCS environment.

IPC makes use of the following other software components and tools, which must be installed in order to use IPC. See the IPC release notes for the specific versions required by your IPC installation.

- Code Composer Studio (CCStudio)
- SYS/BIOS (installed as part of CCStudio)
- XDCtools (installed as part of CCStudio)

1.3 About this User Guide

See the installation guide provided with IPC for installation information and instructions.

- Chapter 2, "The Inter-Processor Communication Package," describes the modules in the ti.sdo.ipc package.
- Chapter 3, “The Utilities Package," describes the modules in the ti.sdo.utils package.
- Chapter 4, “Porting IPC," provides an overview of the steps required to port IPC to new devices or systems.
- Chapter 5, “Optimizing IPC Applications," provides hints for improving the runtime performance and shared memory usage of applications that use IPC.
- Appendix 6, "Rebuilding IPC", explains how to rebuild the IPC libraries if you modify the source files.
- Appendix 7, "Using IPC on Concerto Devices", explains how to use IPC if you are designing applications for Concerto F28M35x devices.

Note: Please see the release notes in the installation before starting to use IPC. The release notes contain important information about feature support, issues, and compatibility information for a particular release.

1.4 Use Cases for IPC

You can use IPC modules in a variety of combinations. From the simplest setup to the setup with the most functionality, the use case options are as follows. A number of variations of these cases are also possible:

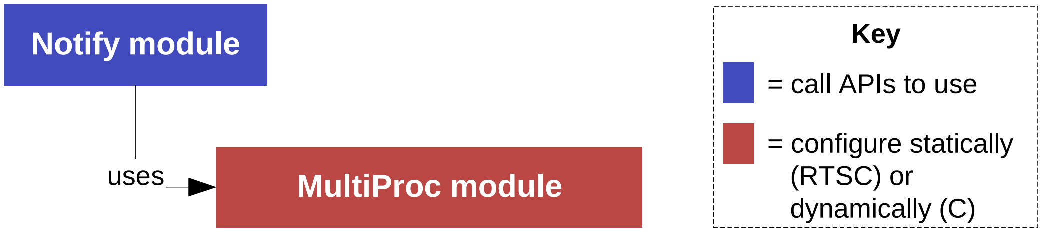
- **Minimal use of IPC.** This scenario performs inter-processor notification. The amount of data passed with a notification is minimal—typically on the order of 32 bits. This scenario is best used for simple synchronization between processors without the overhead and complexity of message-passing infrastructure. The <ipc_install_dir>/packages/ti/sdo/ipc/examples/multicore/<platform_name> directory contains a platform-specific “notify” example for this scenario. See Section 1.4.1.
- **Add data passing.** This scenario adds the ability to pass linked list elements between processors to the previous minimal scenario. The linked list implementation may optionally use shared memory and/or gates to manage synchronization. See Section 1.4.2.
- **Add dynamic allocation.** This scenario adds the ability to dynamically allocate linked list elements from a heap. See Section 1.4.3.
- **Powerful but easy-to-use messaging.** This scenario uses the MessageQ module for messaging. The application configures other modules. However, the APIs for other modules are then used internally by MessageQ, rather than directly by the application. The <ipc_install_dir>/packages/ti/sdo/ipc/examples/multicore/<platform_name> directory contains a platform-specific “message” example for this scenario. See Section 1.4.4.

In the following sections, figures show modules used by each scenario.

- **Blue boxes** identify modules for which your application will call C API functions other than those used to dynamically create objects.
- **Red boxes** identify modules that require only configuration by your application. Static configuration is performed in an XDCtools configuration script (.cfg). Dynamic configuration is performed in C code.
- **Grey boxes** identify modules that are used internally but do not need to be configured or have their APIs called.

1.4.1 Minimal Use Scenario

This scenario performs inter-processor notification using a Notify driver, which is used by the Notify module. This scenario is best used for simple synchronization in which you want to send a message to another processor to tell it to perform some action and optionally have it notify the first processor when it is finished.



In this scenario, you make API calls to the Notify module. For example, the Notify_sendEvent() function sends an event to the specified processor. You can dynamically register callback functions with the Notify module to handle such events.

You must statically configure MultiProc module properties, which are used by the Notify module.

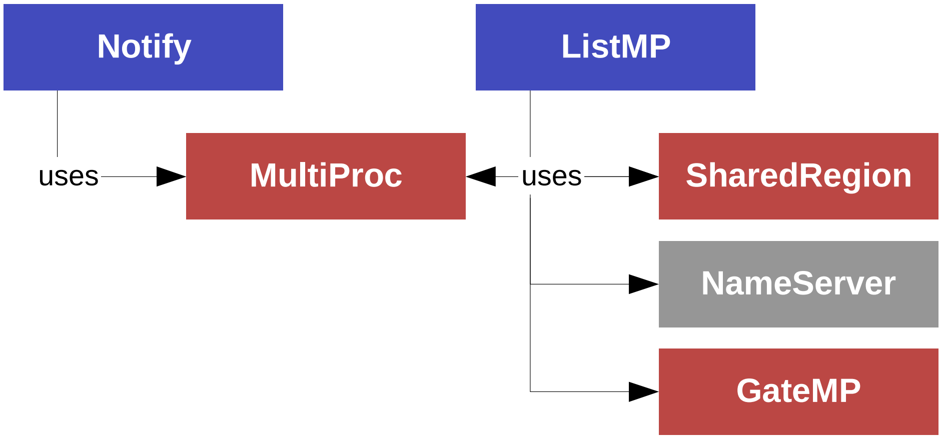
The amount of data passed with a notification is minimal. You can send an event number, which is typically used by the callback function to determine what action it needs to perform. Optionally, a small "payload" of data can also be sent.

The <ipc_install_dir>/packages/ti/sdo/ipc/examples/multicore/ <platform_name> directory contains a platform-specific "notify" example for this scenario. See Section 2.7, *Notify Module* and Section 3.3, *MultiProc Module*.

Note: If you are using a Concerto F28M35x device, this scenario is supported. See Appendix 7 for details.

1.4.2 Data Passing Scenario

In addition to the IPC modules used in the previous scenario, you can use the ListMP module to share a linked list between processors.



In this scenario, you make API calls to the Notify and ListMP modules.

The ListMP module is a doubly-linked-list designed to be shared by multiple processors. ListMP differs from a conventional "local" linked list in the following ways:

- Address translation is performed internally upon pointers contained within the data structure.
- Cache coherency is maintained when the cacheable shared memory is used.
- A multi-processor gate (GateMP) is used to protect read/write accesses to the list by two or more processors.

ListMP uses SharedRegion's lookup table to manage access to shared memory, so configuration of the SharedRegion module is required.

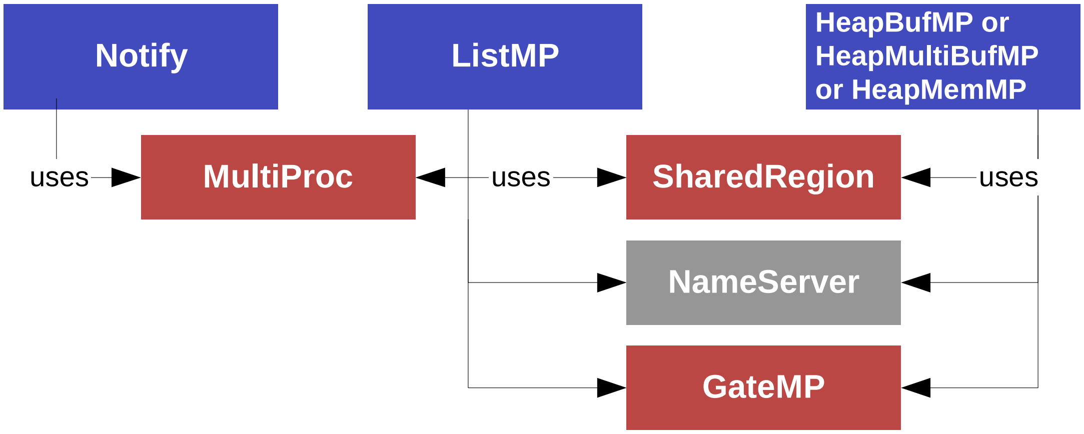
Internally, ListMP can optionally use the NameServer module to manage name/value pairs. The ListMP module also uses a GateMP object, which your application must configure. The GateMP is used internally to synchronize access to the list elements.

See Section 2.4, *ListMP Module*, Section 2.6, *GateMP Module*, Section 2.8, *SharedRegion Module*, and Section 3.4, *NameServer Module*.

Note: If you are using a Concerto F28M35x device, this scenario is not supported due to shared memory limitations. See Appendix 7 for details.

1.4.3 *Dynamic Allocation Scenario*

To the previous scenario, you can add dynamic allocation of ListMP elements using one of the Heap*MP modules.



In this scenario, you make API calls to the Notify and ListMP modules and a Heap*MP module.

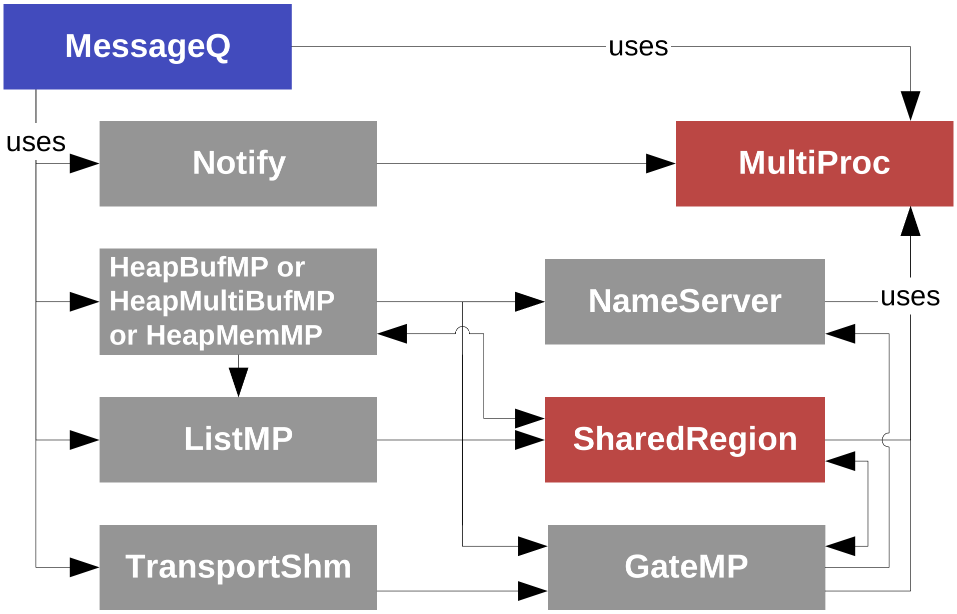
In addition to the modules that you configured for the previous scenario, the Heap*MP modules use a GateMP that you must configure. You may use the same GateMP instance used by ListMP.

See Section 2.5, *Heap*MP Modules* and Section 2.6, *GateMP Module*.

Note: If you are using a Concerto F28M35x device, this scenario is not supported due to shared memory limitations. See Appendix 7 for details.

1.4.4 Powerful But Easy-to-Use Messaging with MessageQ

Finally, to use the most sophisticated inter-processor communication scenario supported by IPC, you can add the MessageQ module.



In this scenario, you make API calls to the MessageQ module for inter-processor communication.

API calls made to the Notify, ListMP, and Heap*MP modules in the previous scenarios are not needed. Instead, your application only needs to configure the MultiProc and SharedRegion modules.

The Ipc_start() API call in your application's main() function takes care of configuring all the modules shown here in gray: the Notify, HeapMemMP, ListMP, TransportShm, NameServer, and GateMP modules.

It is possible to use MessageQ in a single-processor application. In such a case, only API calls to MessageQ and configuration of any xdc.runtime.IHeap implementation are needed.

The <ipc_install_dir>/packages/ti/sdo/ipc/examples/multicore directory contains a "message" example for this scenario.

Note: If you are using a Concerto F28M35x device, this scenario is supported, but fewer modules are used due to shared memory limitations. See Appendix 7 for details.

1.5 Related Documents

To learn more about IPC APIs and the software products used with it, refer to the following API documentation:

IPC online Doxygen-based documentation. Located at <ipc_install_dir>/docs/doxygen/html/index.html. Use this help system to get detailed information about APIs for modules in the ti.ipc package. This system does not contain information about static configuration using XDCtools. This documentation details APIs for all IPC modules that have common header files (see Section 2.1.1). Use this documentation for information about the following aspects of IPC:

- Runtime APIs
- Status codes
- Instance creation parameters
- Type definitions

However, all SYS/BIOS-specific documentation, such as build-time configuration, is located in CDOC (see below).

IPC online CDOC documentation (also called “CDOC”). Open with CCS online help or run <ipc_install_dir>/docs/cdoc/index.html. Use this help system to get information about static configuration of IPC modules and objects using XDCtools and about Error/Assert messages. This help system also contains information about APIs in IPC packages other than ti.ipc and for use if you are building your own modules based on IPC modules and interfaces.

Important: Do not use the CDOC help system to get information about APIs and other aspects of modules in the ti.ipc package. The information in the CDOC system for ti.sdo.ipc package modules does not reflect the interfaces provided by the recommended header files.

- *RTSC-Pedia Wiki:* http://rtsc.eclipse.org/docs-tip
- *Texas Instruments Developer Wiki:* http://processors.wiki.ti.com
- *SYS/BIOS 6 Release Notes:* (BIOS_INSTALL_DIR/Bios_6_##_release_notes.html).
- *SYS/BIOS 6 Getting Started Guide:* (BIOS_INSTALL_DIR/docs/Bios_Getting_Started_Guide.pdf).
- *XDCtools and SYS/BIOS online help:* Open with CCS online help.
- *TMS320 SYS/BIOS 6 User’s Guide* (SPRUEX3)
- In CCS, templates for projects that use IPC are available when you create a CCS project.

Chapter 2

The Inter-Processor Communication Package

This chapter introduces the modules in the ti.sdo.ipc package.

2.1 Modules in the IPC Package

The ti.sdo.ipc package contains the following modules that you may use in your applications:

Module	**Module Path**	
GateMP	GateMP	Manages gates for mutual exclusion of shared resources by multiple processors and threads. See Section 2.6.
HeapBufMP	ti.sdo.ipc.heaps.HeapBufMP	Fixed-sized shared memory Heaps. Similar to SYS/BIOS's ti.sysbios.heaps.HeapBuf module, but with some configuration differences. See Section 2.5.
HeapMemMP	ti.sdo.ipc.heaps.HeapMemMP	Variable-sized shared memory Heaps. See Section 2.5.
HeapMultiBufMP	ti.sdo.ipc.heaps.HeapMultiBufMP	Multiple fixed-sized shared memory Heaps. See Section 2.5.
Ipc	ti.sdo.ipc.Ipc	Provides Ipc_start() function and allows startup sequence configuration. See Section 2.2.
ListMP	ti.sdo.ipc.ListMP	Doubly-linked list for shared-memory, multi-processor applications. Very similar to the ti.sdo.utils.List module. See Section 2.4.
MessageQ	ti.sdo.ipc.MessageQ	Variable size messaging module. See Section 2.3.

Module	Module Path	
TransportShm	ti.sdo.ipc.transports. TransportShm	Transport used by MessageQ for remote communication with other processors via shared memory. See Section 2.3.11.
Notify	ti.sdo.ipc.Notify	Low-level interrupt mux/demuxer module. See Section 2.7.
NotifyDriverShm	ti.sdo.ipc.notifyDrivers. NotifyDriverShm	Shared memory notification driver used by the Notify module to communicate between a pair of processors. See Section 2.7.
SharedRegion	ti.sdo.ipc.SharedRegion	Maintains shared memory for multiple shared regions. See Section 2.8.

Additional modules in the subfolders of the ti.sdo.ipc package contain specific implementations of gates, heaps, notify drivers, transports, and various device family-specific modules.

In addition, the ti.sdo.ipc package defines the following interfaces that you may implement as your own custom modules:

Module	Module Path
IGateMPSupport	ti.sdo.ipc.interfaces.IGateMPSupport
IInterrupt	ti.sdo.ipc.notifyDrivers.IInterrupt
IMessageQTransport	ti.sdo.ipc.interfaces.IMessageQTransport
INotifyDriver	ti.sdo.ipc.interfaces.INotifyDriver
INotifySetup	ti.sdo.ipc.interfaces.INotifySetup

The <ipc_install_dir>/packages/ti/sdo/ipc directory contains the following packages that you may need to know about:

- **examples.** Contains examples.
- **family.** Contains device-specific support modules (used internally).
- **gates.** Contains GateMP implementations (used internally).
- **heaps.** Contains multiprocessor heaps.
- **interfaces.** Contains interfaces.
- **notifyDrivers.** Contains NotifyDriver implementations (used internally).
- **transports.** Contains MessageQ transport implementations that are used internally.

2.1.1 Including Header Files

Applications that use modules in the ti.sdo.ipc or ti.sdo.utils package should include the common header files provided in <ipc_install_dir>/packages/ti/ipc/. These header files are designed to offer a common API for both SYS/BIOS and Linux users of IPC.

The following example C code includes header files applications may need to use. Depending on the APIs used in your application code, you may need to include different XDC, IPC, and SYS/BIOS header files.

```
#include <xdc/std.h>
#include <string.h>

/* ---- XDC.RUNTIME module Headers    */
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/IHeap.h>

/* ----- IPC module Headers           */
#include <ti/ipc/GateMP.h>
#include <ti/ipc/Ipc.h>
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/HeapBufMP.h>
#include <ti/ipc/MultiProc.h>

/* ---- BIOS6 module Headers          */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

/* ---- Get globals from .cfg Header */
#include <xdc/cfg/global.h>
```

Note that the appropriate include file location has changed from previous versions of IPC. The XDCtools-generated header files are still available in <ipc_install_dir>/packages/ti/sdo/ipc/, but these should not directly be included in runtime .c code.

You should search your applications for "ti/sdo/ipc" and "ti/sdo/utils" and change the header file references found as needed. Additional changes to API calls will be needed.

Documentation for all common-header APIs is provided in Doxygen format at <ipc_install_dir>/docs/doxygen/html/index.html.

2.1.2 Standard IPC Function Call Sequence

For instance-based modules in IPC, the standard IPC methodology when creating object dynamically (that is, in C code) is to have the creator thread first initialize a *MODULE*_Params structure to its default values via a *MODULE*_Params_init() function. The creator thread can then set individual parameter fields in this structure as needed. After setting up the *MODULE*_Params structure, the creator thread calls the *MODULE*_create() function to creates the instance and initializes any shared memory used by the instance. If the instance is to be opened remotely, a unique name must be supplied in the parameters.

Other threads can access this instance via the *MODULE*_open() function, which returns a handle with access to the instance. The name that was used for instance creation must be used in the *MODULE*_open() function.

In most cases, MODULE_open() functions must be called in the context of a Task. This is because the thread running the MODULE_open() function needs to be able to block (to pend on a Semaphore in this case) while waiting for the remote processor to respond. The response from the remote processor triggers a hardware interrupt, which then posts a Semaphore to allow to Task to resume execution. The exception to this rule is that MODULE_open() functions do not need to be able to block when opening an instance on the local processor.

When the threads have finished using an instance, all threads that called *MODULE*_open() must call *MODULE*_close(). Then, the thread that called *MODULE*_create() can call *MODULE*_delete() to free the memory used by the instance.

Note that *all* threads that opened an instance must close that instance before the thread that created it can delete it. Also, a thread that calls *MODULE*_create() cannot call *MODULE*_close(). Likewise, a thread that calls *MODULE*_open() cannot call *MODULE*_delete().

2.1.3 Error Handling in IPC

Many of the APIs provided by IPC return an integer as a status code. Your application can test the status value returned against any of the provided status constants. For example:

```
MessageQ_Msg     msg;
MessageQ_Handle  messageQ;
Int              status;

...
status = MessageQ_get(messageQ, &msg, MessageQ_FOREVER);
    if (status < 0) {
        System_abort("Should not happen\n");
    }
```

Status constants have the following format: `MODULE_[S|E]_CONDITION`. For example, Ipc_S_SUCCESS, MessageQ_E_FAIL, and SharedRegion_E_MEMORY are status codes that may be returned by functions in the corresponding modules.

Success codes always have values greater or equal to zero. For example, Ipc_S_SUCCESS=0 and Ipc_S_ALREADYSETUP=1; both are success codes. Failure codes always have values less than zero. Therefore, the presence of an error can be detected by simply checking whether the return value is negative.

Other APIs provided by IPC return a handle to a created object. If the handle is NULL, an error occurred when creating the object. For example:

```
messageQ = MessageQ_create(DSP_MESSAGEQNAME, NULL);
if (messageQ == NULL) {
    System_abort("MessageQ_create failed\n");
}
```

Refer to the Doxygen documentation for status codes returned by IPC functions.

2.2 Ipc Module

Note: The Ipc module is not used on Concerto F28M35x devices. Instead, the IpcMgr module (in the ti.sdo.ipc.family.f28m35x package) is used to configure the devices as described in Section B.2. Concerto applications should *not* call any Ipc or IpcMgr APIs at runtime.

The main purpose of the Ipc module is to initialize the various subsystems of IPC. All applications that use IPC modules must call the Ipc_start() API, which does the following:

- Initializes a number of objects and modules used by IPC
- Synchronizes multiple processors so they can boot in any order

An application that uses IPC APIs—such as MessageQ, GateMP, and ListMP—must include the Ipc module header file and call Ipc_start() in the main() function. If the main() function calls any IPC APIs, the call to Ipc_start() must be placed before any calls to IPC modules. For example:

```
#include <ti/ipc/Ipc.h>

...

Int main(Int argc, Char* argv[])
{
    Int status;

    /* Call Ipc_start() */
    status = Ipc_start();
    if (status < 0) {
        System_abort("Ipc_start failed\n");
    }

    BIOS_start();
    return (0);
}
```

By default, Ipc_start() internally calls Notify_start() if it has not already been called. Ipc_start() then loops through the defined SharedRegions so that it can set up the HeapMemMP and GateMP instances used internally by the IPC modules. It also sets up MessageQ transports to remote processors.

The SharedRegion with an index of 0 (zero) is used by IPC_start() to create resource management tables for internal use by other IPC modules. Thus SharedRegion "0" must be accessible by all processors. See Section 2.8 for more about the SharedRegion module.

2.2.1 Ipc Module Configuration

In an XDCtools configuration file, you configure the Ipc module for use as follows:

```
Ipc = xdc.useModule('ti.sdo.ipc.Ipc');
```

You can configure what the Ipc_start() API will do—which modules it will start and which objects it will create—by using the Ipc.setEntryMeta method in the configuration file to set the following properties:

- **setupNotify.** If set to false, the Notify module is not set up. The default is true.

- **setupMessageQ.** If set to false, the MessageQ transport instances to remote processors are not set up and the MessageQ module does not attach to remote processors. The default is true.

For example, the following statements from the notify example configuration turn off the setup of the MessageQ transports and connections to remote processors:

```
/* To avoid wasting shared memory for MessageQ transports */
for (var i = 0; i < MultiProc.numProcessors; i++) {
    Ipc.setEntryMeta({
        remoteProcId: i,
        setupMessageQ: false,
    });
}
```

You can configure how the IPC module synchronizes processors by configuring the Ipc.procSync property. For example:

```
Ipc.procSync = Ipc.ProcSync_ALL;
```

The options are:

- **Ipc.ProcSync_ALL.** If you use this option, the Ipc_start() API automatically attaches to and synchronizes all remote processors. If you use this option, your application should never call Ipc_attach(). Use this option if all IPC processors on a device start up at the same time and connections should be established between every possible pair of processors.
- **Ipc.ProcSync_PAIR.** (Default) If you use this option, you must explicitly call Ipc_attach() to attach to a specific remote processor. If you use this option, Ipc_start() performs system-wide IPC initialization, but does not make connections to remote processors. Use this option if any or all of the following are true:
  - — You need to control when synchronization with each remote processor occurs.
  - — Useful work can be done while trying to synchronize with a remote processor by yielding a thread after each attempt to Ipc_attach() to the processor.
  - — Connections to some remote processors are unnecessary and should be made selectively to save memory.
- **Ipc.ProcSync_NONE.** If you use this option, Ipc_start() doesn't synchronize any processors before setting up the objects needed by other modules. Use this option with caution. It is intended for use in cases where the application performs its own synchronization and you want to avoid a potential deadlock situation with the IPC synchronization.

  If you use the ProcSync_NONE option, Ipc_start() works exactly as it does with ProcSync_PAIR.

  However, in this case, Ipc_attach() does not synchronize with the remote processor. As with other ProcSync options, Ipc_attach() still sets up access to GateMP, SharedRegion, Notify, NameServer, and MessageQ transports, so your application must still call Ipc_attach() for each remote processor that will be accessed. Note that an Ipc_attach() call for a remote processor whose ID is less than the local processor's ID must occur *after* the corresponding remote processor has called Ipc_attach() to the local processor. For example, processor #2 can call Ipc_attach(1) only after processor #1 has called Ipc_attach(2).

You can configure a function to perform custom actions in addition to the default actions performed when attaching to or detaching from a remote processor. These functions run near the end of Ipc_attach() and near the beginning of Ipc_detach(), respectively (see Section 2.2.2). Such functions must be non-blocking and must run to completion. The following example configures two attach functions and two detach functions. Each set of functions will be passed a different argument:

```
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');

var fxn = new Ipc.UserFxn;
fxn.attach = '&userAttachFxn1';
fxn.detach = '&userDetachFxn1';
Ipc.addUserFxn(fxn, 0x1);

fxn.attach = '&userAttachFxn2';
fxn.detach = '&userDetachFxn2';
Ipc.addUserFxn(fxn, 0x2);
```

2.2.2 Ipc Module APIs

In addition to the Ipc_start() API, which all applications that use IPC modules are required to call, the Ipc module also provides the following APIs for processor synchronization:

- **Ipc_attach()** Creates a connection to the specified remote processor.
- **Ipc_detach()** Deletes the connection to the specified remote processor.

You must call Ipc_start() on a processor before calling Ipc_attach().

Note: Call Ipc_attach() to the processor that owns shared memory region 0 (usually the processor with id = 0) before making a connection to any other remote processor. For example, if there are three processors configured with MultiProc, #1 should attach to #0 before it can attach to #2.

Use these functions *unless* you are using the Ipc.ProcSync_ALL configuration setting. With that option, Ipc_start() automatically attaches to and synchronizes all remote processors, and your application should never call Ipc_attach().

The Ipc.ProcSync_PAIR configuration option expects that your application will call Ipc_attach() for each remote processor with which it should be able to communicate.

Processor synchronization means that one processor waits until the other processor signals that a particular module is ready for use. Within Ipc_attach(), this is done for the GateMP, SharedRegion (region 0), and Notify modules and the MessageQ transports.

You can call the Ipc_detach() API to delete internal instances created by Ipc_attach() and to free the memory used by these instances.

2.3 MessageQ Module

The MessageQ module supports the structured sending and receiving of variable length messages. It is OS independent and works with any threading model. For each MessageQ you create, there is a single reader and may be multiple writers.

Note: MessageQ use is the same with Concerto F28M35x devices as for other devices. See Section 5.3.2 for information about the TransportCirc driver used with the MessageQ module when you are using Concerto devices.

MessageQ is the recommended messaging API for most applications. It can be used for both homogeneous and heterogeneous multi-processor messaging, along with single-processor messaging between threads.

With the additional setup now performed automatically by Ipc_start()— the creation of transports, initialization of shared memory, and more—configuration of objects used by MessageQ is much easier than in previous versions of IPC.

(The MessageQ module in IPC is similar in functionality to the MSGQ module in DSP/BIOS 5.x.)

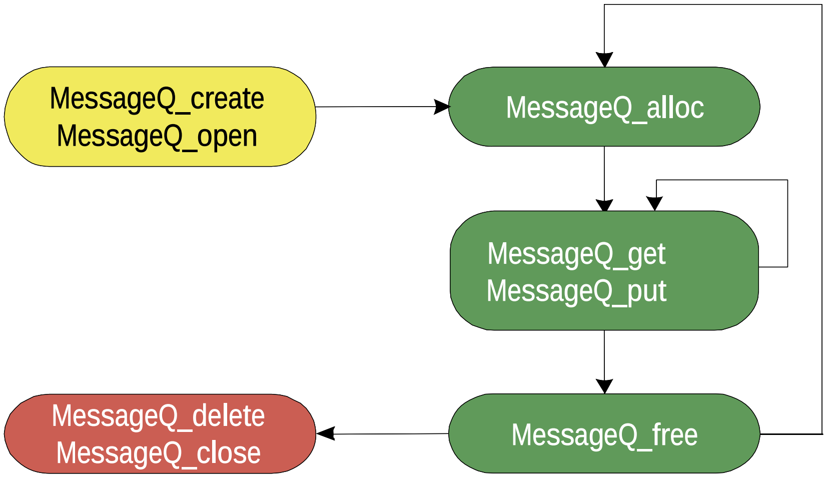
The following are key features of the MessageQ module:

- Writers and readers can be relocated to another processor with no runtime code changes.
- Timeouts are allowed when receiving messages.
- Readers can determine the writer and reply back.
- Receiving a message is deterministic when the timeout is zero.
- Messages can reside on any message queue.
- Supports zero-copy transfers.
- Messages can be sent and received from any type of thread.
- The notification mechanism is specified by the application.
- Allows QoS (quality of service) on message buffer pools. For example, using specific buffer pools for specific message queues.

Messages are sent and received via a message queue. A reader is a thread that gets (reads) messages from a message queue. A writer is a thread that puts (writes) a message to a message queue. Each message queue has one reader and can have many writers. A thread may read from or write to multiple message queues.

- **Reader.** The single reader thread calls MessageQ_create(), MessageQ_get(), MessageQ_free(), and MessageQ_delete().
- **Writer.** Writer threads call MessageQ_open(), MessageQ_alloc(), MessageQ_put(), and MessageQ_close().

The following figure shows the flow in which applications typically use the main runtime MessageQ APIs:



Conceptually, the reader thread owns a message queue. The reader thread creates a message queue. Writer threads then open a created message queue to get access to them.

2.3.1 Configuring the MessageQ Module

You can configure a number of module-wide properties for MessageQ in your XDCtools configuration file. If you are configuring the MessageQ module, you must enable the module as follows:

```
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');
```

Module-wide configuration properties you can set are as follows. The default values are shown in the following statements. See the IPC online documentation for details.

```
// Maximum length of MessageQ names
MessageQ.maxNameLen = 32;

// Max number of MessageQs that can be dynamically created
MessageQ.maxRuntimeEntries = 10;

// Number of heapIds in the system
MessageQ.numHeaps = 0;

// Section name used to place the names table
MessageQ.tableSection = null;
```

2.3.2 Creating a MessageQ Object

You can create message queues dynamically. Static creation is not supported. A MessageQ object is not a shared resource. That is, it resides on the processor that creates it.

The reader thread creates a message queue. To create a MessageQ object dynamically, use the MessageQ_create() C API, which has the following syntax:

```
MessageQ_Handle MessageQ_create(String           name,
                                MessageQ_Params  *params);
```

When you create a queue, you specify a name string. This name will be needed by the MessageQ_open() function, which is called by threads on the same or remote processors that want to send messages to the created message queue. While the name is not required (that is, it can be NULL), an unnamed queue cannot be opened.

An ISync handle is associated with the message queue via the synchronizer parameter (see Section 2.3.9 for details).

If the call is successful, the MessageQ_Handle is returned. If the call fails, NULL is returned.

You initialize the params struct by using the MessageQ_Params_init() function, which initializes the params structure with the default values. A NULL value for params can be passed into the create call, which results in the defaults being used. The default synchronizer is SyncSem.

The following code creates a MessageQ object using SyncSem as the synchronizer.

```
MessageQ_Handle   messageQ;
MessageQ_Params   messageQParams;
SyncSem_Handle    syncSemHandle;

...

syncSemHandle = SyncSem_create(NULL, NULL);
MessageQ_Params_init(&messageQParams);
messageQParams.synchronizer = SyncSem_Handle_upCast(syncSemHandle);
messageQ = MessageQ_create(CORE0_MESSAGEQNAME, &messageQParams);
```

In this example, the `CORE0_MESSAGEQNAME` constant is defined in the message_common.cfg.xs configuration file.

2.3.3 Opening a Message Queue

Writer threads open a created message queue to get access to them. In order to obtain a handle to a message queue that has been created, a writer thread must call MessageQ_open(), which has the following syntax.

```
Int MessageQ_open(String name, MessageQ_QueueId *queueId);
```

This function expects a name, which must match with the name of the created object. Internally MessageQ calls NameServer_get() to find the 32-bit queueId associated with the created message queue. NameServer looks both locally and remotely.

If no matching name is found on any processor, MessageQ_open() returns MessageQ_E_NOTFOUND. If the open is successful, the Queue ID is filled in and MessageQ_S_SUCCESS is returned.

The following code opens the MessageQ object created by the processor.

```
MessageQ_QueueId   remoteQueueId;
Int                status;

...

/* Open the remote message queue. Spin until it is ready. */
do {
   status = MessageQ_open(CORE0_MESSAGEQNAME, &remoteQueueId);
}
while (status < 0);
```

2.3.4 Allocating a Message

MessageQ manages message allocation via the MessageQ_alloc() and MessageQ_free() functions. MessageQ uses Heaps for message allocation. MessageQ_alloc() has the following syntax:

```
MessageQ_Msg MessageQ_alloc(UInt16        heapId,
                            UInt32        size);
```

The allocation size in MessageQ_alloc() must include the size of the message header, which is 32 bytes.

The following code allocates a message:

```
#define MSGSIZE                  256
#define HEAPID                     0
...
MessageQ_Msg     msg;

...

msg = MessageQ_alloc(HEAPID, sizeof(MessageQ_MsgHeader));
if (msg == NULL) {
    System_abort("MessageQ_alloc failed\n");
}
```

Once a message is allocated, it can be sent on any message queue. Once the reader receives the message, it may either free the message or re-use the message.

Messages in a message queue can be of variable length. The only requirement is that the first field in the definition of a message must be a MsgHeader structure. For example:

```
typedef struct MyMsg {
    MessageQ_MsgHeader header;      // Required
    SomeEnumType       type         // Can be any field
    ...                             // ...
} MyMsg;
```

The MessageQ APIs use the MessageQ_MsgHeader internally. Your application should not modify or directly access the fields in the MessageQ_MsgHeader structure.

2.3.4.1 MessageQ Allocation and Heaps

All messages sent via the MessageQ module must be allocated from a xdc.runtime.IHeap implementation, such as ti.sdo.ipc.heaps.HeapBufMP. The same heap can also be used for other memory allocation not related to MessageQ.

The MessageQ_registerHeap() API assigns a MessageQ heapId to a heap. When allocating a message, the heapId is used, not the heap handle. The heapIds should start at zero and increase. The maximum number of heaps is determined by the numHeap module configuration property. See the online documentation for MessageQ_registerHeap() for details.

```
/* Register this heap with MessageQ */
status = MessageQ_registerHeap( HeapBufMP_Handle_upCast(heapHandle), HEAPID);
```

If the registration fails (for example, the heapId is already used), this function returns FALSE.

An application can use multiple heaps to allow an application to regulate its message usage. For example, an application can allocate critical messages from a heap of fast on-chip memory and non-critical messages from a heap of slower external memory. Additionally, heaps MessageQ uses can be shared with other modules and/or the application.

MessageQ alternatively supports allocating messages without the MessageQ_alloc() function. See Section 2.3.4.2, *MessageQ Allocation Without a Heap* for more information.

Heaps can be unregistered via MessageQ_unregisterHeap().

2.3.4.2 MessageQ Allocation Without a Heap

It is possible to send MessageQ messages that are allocated statically instead of being allocated at run-time via MessageQ_alloc(). However the first field of the message must still be a MsgHeader. To make sure the MsgHeader has valid settings, the application must call MessageQ_staticMsgInit(). This function initializes the header fields in the same way that MessageQ_alloc() does, except that it sets the heapId field in the header to the MessageQ_STATICMSG constant.

If an application uses messages that were not allocated using MessageQ_alloc(), it cannot free the messages via the MessageQ_free() function, even if the message is received by a different processor. Also, the transport may internally call MessageQ_free() and encounter an error.

If MessageQ_free() is called on a statically allocated message, it asserts that the heapId of the message is not MessageQ_STATICMSG.

2.3.5 Sending a Message

Once a message queue is opened and a message is allocated, the message can be sent to the MessageQ via the MessageQ_put() function, which has the following syntax.

```
Int MessageQ_put(MessageQ_QueueId queueId,
                 MessageQ_Msg     msg);
```

For example:

```
status = MessageQ_put(remoteQueueId, msg);
```

Opening a queue is not required. Instead the message queue ID can be "discovered" via the MessageQ_getReplyQueue() function (see Section 2.3.10 for more information), which returns the 32-bit queueId.

```
MessageQ_QueueId replyQueue;
MessageQ_Msg     msg;

/* Use the embedded reply destination */
replyMessageQ = MessageQ_getReplyQueue(msg);
if (replyMessageQ == MessageQ_INVALIDMESSAGEQ) {
    System_abort("Invalid reply queue\n");
}

/* Send the response back */
status = MessageQ_put(replyQueue, msg);
    if (status < 0) {
        System_abort("MessageQ_put was not successful\n");
    }
```

If the destination queue is local, the message is placed on the appropriate priority linked list and the ISync signal function is called. If the destination queue is on a remote processor, the message is given to the proper transport and returns. See Section 2.3.11 for more information.

If MessageQ_put() succeeds, it returns MessageQ_S_SUCCESS. If MessageQ_E_FAIL is returned, an error occurred and the caller still owns the message.

There can be multiple senders to a single message queue. MessageQ handles the thread safety.

Before you send a message, you can use the MessageQ_setMsgId() function to assign a numeric value to the message that can be checked by the receiving thread.

```
/* Increment...the remote side will check this */
msgId++;
MessageQ_setMsgId(msg, msgId);
```

You can use the MessageQ_setMsgPri() function to set the priority of the message. See Section 2.3.8 for more about message priorities.

2.3.6 Receiving a Message

To receive a message, a reader thread calls the MessageQ_get() API.

```
Int MessageQ_get(MessageQ_Handle handle,
                 MessageQ_Msg    *msg,
                 UInt            timeout)
```

If a message is present, it returned by this function. In this case the ISync's wait() function is not called.

For example:

```
/* Get a message */
status = MessageQ_get(messageQ, &msg, MessageQ_FOREVER);
if (status < 0) {
     System_abort("Should not happen; timeout is forever\n");
}
```

If no message is present and no error occurs, this function blocks while waiting for the timeout period for the message to arrive. If the timeout period expires, MessageQ_E_FAIL is returned. If an error occurs, the msg argument will be unchanged.

After receiving a message, you can use the following APIs to get information about the message from the message header:

- MessageQ_getMsgId() gets the ID value set by MessageQ_setMsgId(). For example:

```
/* Get the id and increment it to send back */
msgId = MessageQ_getMsgId(msg);
msgId += NUMCLIENTS;
MessageQ_setMsgId(msg, msgId);
```

- MessageQ_getMsgPri() gets the priority set by MessageQ_setMsgPri(). See Section 2.3.8.
- MessageQ_getMsgSize() gets the size of the message in bytes.
- MessageQ_getReplyQueue() gets the ID of the queue provided by MessageQ_setReplyQueue(). See Section 2.3.10.

2.3.7 Deleting a MessageQ Object

MessageQ_delete() frees a MessageQ object stored in local memory. If any messages are still on the internal linked lists, they will be freed. The contents of the handle are nulled out by the function to prevent use after deleting.

```
Void MessageQ_delete(MessageQ_Handle *handle);
```

The queue array entry is set to NULL to allow re-use.

Once a message queue is deleted, no messages should be sent to it. A MessageQ_close() is recommended, but not required.

2.3.8 Message Priorities

MessageQ supports three message priorities as follows:

- MessageQ_NORMALPRI = 0
- MessageQ_HIGHPRI = 1
- MessageQ_URGENTPRI = 3

You can set the priority level for a message before sending it by using the MessageQ_setMsgPri function:

```
Void MessageQ_setMsgPri(MessageQ_Msg        msg,
                        MessageQ_Priority   priority)
```

Internally a MessageQ object maintains two linked lists: normal and high-priority. A normal priority message is placed onto the "normal" linked list in FIFO manner. A high priority message is placed onto the "high-priority" linked list in FIFO manner. An urgent message is placed at the beginning of the high linked list.

Note: Since multiple urgent messages may be sent before a message is read, the order of urgent messages is not guaranteed.

When getting a message, the reader checks the high priority linked list first. If a message is present on that list, it is returned. If not, the normal priority linked list is checked. If a message is present there, it is returned. Otherwise the synchronizer's wait function is called.

See Section 2.3.11, *Remote Communication via Transports* for information about the handling of priority by transports.

2.3.9 Thread Synchronization

MessageQ supports reads and writes of different thread models. It can work with threading models that include SYS/BIOS's Hwi, Swi, and Task threads.

This flexibility is accomplished by using an implementation of the xdc.runtime.knl.ISync interface. The creator of the message queue must also create an object of the desired ISync implementation and assign that object as the "synchronizer" of the MessageQ. Each message queue has its own synchronizer object.

An ISync object has two main functions: signal() and wait(). Whenever MessageQ_put() is called, the signal() function of the ISync implementation is called. If MessageQ_get() is called when there are no messages on the queue, the wait() function of the ISync implementation is called. The timeout passed into the MessageQ_get() is directly passed to the ISync wait() API.

Important: Since ISync implementations must be binary, the reader thread must drain the MessageQ of all messages before waiting for another signal.

For example, if the reader is a SYS/BIOS Swi, the instance could be a SyncSwi. When a MessageQ_put() is called, the Swi_post() API would be called. The Swi would run and it must call MessageQ_get() until no messages are returned. If the Swi does not get all the messages, the Swi will not run again, or at least will not run until a new message is placed on the queue.

The calls to ISync functions occurs directly in MessageQ_put() when the call occurs on the same processor where the queue was created. In the remote case, the transport calls MessageQ_put(), which is then a local put, and the signal function is called. (See Section 2.3.11.)

The following are ISync implementations provided by XDCtools and SYS/BIOS:

- **xdc.runtime.knl.SyncNull.** The signal() and wait() functions do nothing. Basically this implementation allows for polling.
- **xdc.runtime.knl.SyncSemThread.** An implementation built using the xdc.runtime.knl.Semaphore module, which is a binary semaphore.
- **xdc.runtime.knl.SyncGeneric.xdc.** This implementation allows you to use custom signal() and wait() functions as needed.
- **ti.sysbios.syncs.SyncSem.** An implementation built using the ti.sysbios.ipc.Semaphore module. The signal() function runs a Semaphore_post(). The wait() function runs a Semaphore_pend().
- **ti.sysbios.syncs.SyncSwi.** An implementation built using the ti.sysbios.knl.Swi module. The signal() function runs a Swi_post(). The wait() function does nothing and returns FALSE if the timeout elapses.
- **ti.sysbios.syncs.SyncEvent.** An implementation built using the ti.sysbios.ipc.Event module. The signal() function runs an Event_post(). The wait() function does nothing and returns FALSE if the timeout elapses. This implementation allows waiting on multiple events.

The following code from the "message" example creates a SyncSem instance and assigns it to the synchronizer field in the MessageQ_Params structure before creating the MessageQ instance:

```
#include <ti/sysbios/syncs/SyncSem.h>
...

MessageQ_Params   messageQParams;
SyncSem_Handle    syncSemHandle;

/* Create a message queue using SyncSem as synchronizer */
syncSemHandle = SyncSem_create(NULL, NULL);
MessageQ_Params_init(&messageQParams);
messageQParams.synchronizer = SyncSem_Handle_upCast(syncSemHandle);
messageQ = MessageQ_create(CORE1_MESSAGEQNAME, &messageQParams, NULL);
```

2.3.10 ReplyQueue

For some applications, doing a MessageQ_open() on a queue is not realistic. For example, a server may not want to open all the clients' queues for sending responses. To support this use case, the message sender can embed a reply queueId in the message using the MessageQ_setReplyQueue() function.

```
Void MessageQ_setReplyQueue(MessageQ_Handle handle,
                            MessageQ_Msg msg)
```

This API stores the message queue's queueId into fields in the MsgHeader.

The MessageQ_getReplyQueue() function does the reverse. For example:

```
MessageQ_QueueId replyQueue;
MessageQ_Msg     msg;
...

/* Use the embedded reply destination */
replyMessageQ = MessageQ_getReplyQueue(msg);
if (replyMessageQ == MessageQ_INVALIDMESSAGEQ) {
   System_abort("Invalid reply queue\n");
}
```

The MessageQ_QueueId value returned by this function can then be used in a MessageQ_put() call.

The queue that is embedded in the message does not have to be the sender's queue.

2.3.11 Remote Communication via Transports

MessageQ is designed to support multiple processors. To allow this, different transports can be plugged into MessageQ.

In a multi-processor system, MessageQ communicates with other processors via ti.sdo.ipc.interfaces.IMessageQTransport instances. There can be up to two IMessageQTransport instances for each processor to which communication is desired. One can be a normal-priority transport and the other for handling high-priority messages. This is done via the priority parameter in the transport create() function. If there is only one register to a remote processor (either normal or high), all messages go via that transport.

There can be different transports on a processor. For example, there may be a shared memory transport to processor A and an sRIO one to processor B.

When your application calls Ipc_start(), the default transport instance used by MessageQ is created automatically. Internally, transport instances are responsible for registering themselves with MessageQ via the MessageQ_registerTransport() function.

IPC provides an implementation of the IMessageQTransport interface called ti.sdo.ipc.transports.TransportShm (shared memory). You can write other implementations to meet your needs.

When a transport is created via a transport-specific create() call, a remote processor ID (defined via the MultiProc module) is specified. This ID denotes which processor this instance communicates with. Additionally there are configuration properties for the transport—such as the message priority handled—that can be defined in a Params structure. The transport takes these pieces of information and registers itself with MessageQ. MessageQ now knows which transport to call when sending a message to a remote processor.

Trying to send to a processor that has no transport results in an error.

2.3.11.1 Custom Transport Implementations

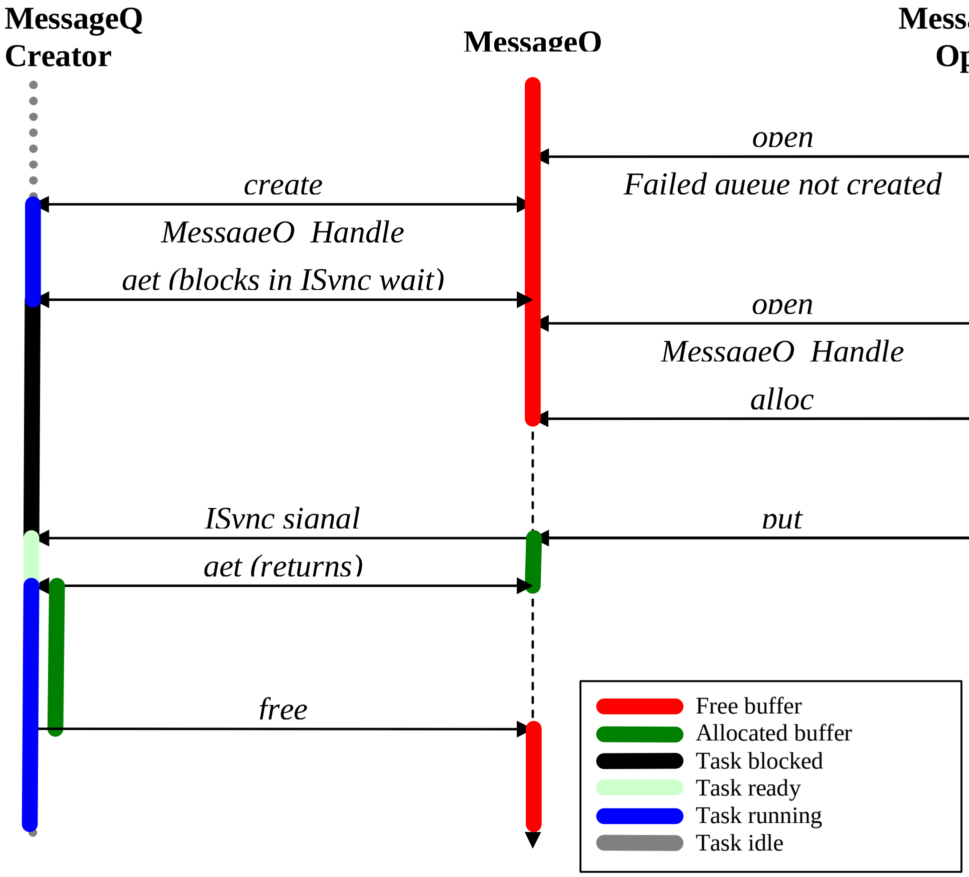
Transports can register and unregister themselves dynamically. That is, if the transport instance is deleted, it should unregister with MessageQ.

When receiving a message, transports need to form the MessageQ_QueueId that allows them to call MessageQ_put(). This is accomplished via the MessageQ_getDstQueue() API.

```
MessageQ_QueueId MessageQ_getDstQueue(MessageQ_Msg msg)
```

2.3.12 Sample Runtime Program Flow (Dynamic)

The following figure shows the typical sequence of events when using a MessageQ. A message queue is created by a Task. An open on the same processor then occurs. Assume there is one message in the system. The opener allocates the message and sends it to the created message queue, which gets and frees it.



2.4 ListMP Module

The ti.sdo.ipc.ListMP module is a linked-list based module designed to be used in a multi-processor environment. It is designed to provide a means of communication between different processors.

Note: The ListMP module is not supported for Concerto F28M35x devices.

ListMP uses shared memory to provide a way for multiple processors to share, pass, or store data buffers, messages, or state information. ListMP is a low-level module used by several other IPC modules, including MessageQ, HeapBufMP, and transports, as a building block for their instance and state structures.

A common challenge that occurs in a multi-processor environment is preventing concurrent data access in shared memory between different processors. ListMP uses a multi-processor gate to prevent multiple processors from simultaneously accessing the same linked-list. All ListMP operations are atomic across processors.

You create a ListMP instance dynamically as follows:

1. Initialize a ListMP_Params structure by calling ListMP_Params_init().
2. Specify the name, regionId, and other parameters in the ListMP_Params structure.
3. Call ListMP_create().

ListMP uses a ti.sdo.utils.NameServer instance to store the instance information. The ListMP name supplied must be unique for all ListMP instances in the system.

```
ListMP_Params params;
GateMP_Handle gateHandle;
ListMP_Handle handle1;

/* If gateHandle is NULL, the default remote gate will be
   automatically chosen by ListMP */
gateHandle = GateMP_getDefaultRemote();
ListMP_Params_init(&params);
params.gate = gateHandle;
params.name = "myListMP";
params.regionId = 1;
handle1 = ListMP_create(&params, NULL);
```

Once created, another processor or thread can open the ListMP instance by calling ListMP_open().

```
while (ListMP_open("myListMP", &handle1, NULL) < 0) {
    ;
}
```

ListMP uses SharedRegion pointers (see Section 2.8), which are portable across processors, to translate addresses for shared memory. The processor that creates the ListMP instance must specify the shared memory in terms of its local address space. This shared memory must have been defined in the SharedRegion module by the application.

The ListMP module has the following constraints:

- ListMP elements to be added/removed from the linked-list must be stored in a shared memory region.
- The linked list must be on a worst-case cache line boundary for all the processors sharing the list.
- ListMP_open() should be called only when global interrupts are enabled.

A list item must have a field of type ListMP_Elem as its first field. For example, the following structure could be used for list elements:

```
typedef struct Tester {
    ListMP_Elem elem;
    Int         scratch[30];
    Int         flag;
} Tester;
```

Besides creating, opening, and deleting a list instance, the ListMP module provides functions for the following common list operations:

- **ListMP_empty().** Test for an empty ListMP.
- **ListMP_getHead().** Get the element from the front of the ListMP.
- **ListMP_getTail().** Get the element from the end of the ListMP.
- **ListMP_insert().** Insert element into a ListMP at the current location.
- **ListMP_next().** Return the next element in the ListMP (non-atomic).
- **ListMP_prev().** Return previous element in the ListMP (non-atomic).
- **ListMP_putHead().** Put an element at the head of the ListMP.
- **ListMP_putTail().** Put an element at the end of the ListMP.
- **ListMP_remove().** Remove the current element from the middle of the ListMP.

This example prints a "flag" field from the list elements in a ListMP instance in order:

```
System_printf("On the List: ");
testElem = NULL;
while ((testElem = ListMP_next(handle, (ListMP_Elem *)testElem)) != NULL) {
    System_printf("%d ", testElem->flag);
}
```

This example prints the same items in reverse order:

```
System_printf("in reverse: ");
elem = NULL;
while ((elem = ListMP_prev(handle, elem)) != NULL) {
    System_printf("%d ", ((Tester  *)elem)->flag);
}
```

This example determines if a ListMP instance is empty:

```
if (ListMP_empty(handle1) == TRUE) {
    System_printf("Yes, handle1 is empty\n");
}
```

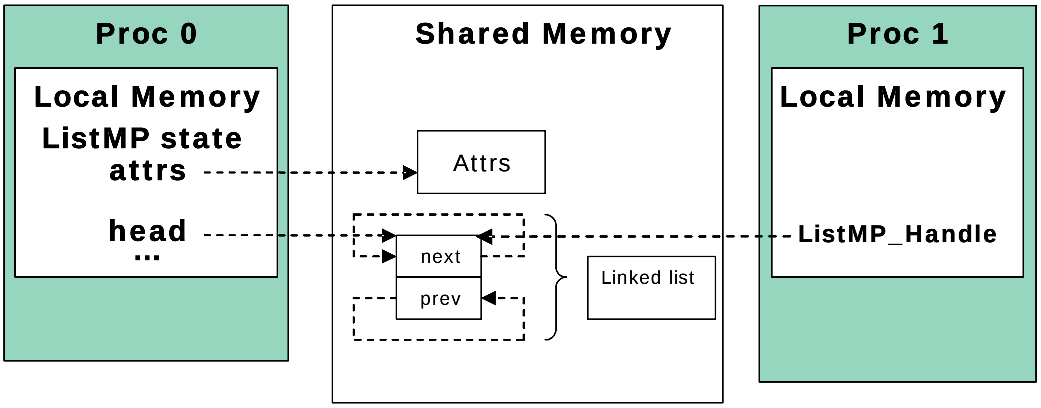
This example places a sequence of even numbers in a ListMP instance:

```
/* Add 0, 2, 4, 6, 8 */
for (i = 0; i < COUNT; i = i + 2) {
    ListMP_putTail(handle1, (ListMP_Elem *)&(buf[i]));
}
```

The instance state information contains a pointer to the head of the linked-list, which is stored in shared memory. Other attributes of the instance stored in shared memory include the version, status, and the size of the shared address.

Other processors can obtain a handle to the linked list by calling ListMP_open().

The following figure shows local memory and shared memory for processors Proc 0 and Proc 1, in which Proc 0 calls ListMP_create() and Proc 1 calls ListMP_open().



The cache alignment used by the list is taken from the SharedRegion on a per-region basis. The alignment must be the same across all processors and should be the worst-case cache line boundary.

2.5 Heap*MP Modules

Note: The Heap*MP modules are not supported for Concerto F28M35x devices because Concerto does not support shared memory heaps.

The ti.sdo.ipc.heaps package provides three implementations of the xdc.runtime.IHeap interface.

- **HeapBufMP.** Fixed-size memory manager. All buffers allocated from a HeapBufMP instance are of the same size. There can be multiple instances of HeapBufMP that manage different sizes. The ti.sdo.ipc.heaps.HeapBufMP module is modeled after SYS/BIOS 6's HeapBuf module (ti.sysbios.heaps.HeapBuf).
- **HeapMultiBufMP.** Each instance supports up to 8 different fixed sizes of buffers. When an allocation request is made, the HeapMultiBufMP instance searches the different buckets to find the smallest one that satisfies the request. If that bucket is empty, the allocation fails. The ti.sdo.ipc.heaps.HeapMultiBufMP module is modeled after SYS/BIOS 6's HeapMultiBuf module (ti.sysbios.heaps.HeapMultiBuf).

- **HeapMemMP.** Variable-size memory manager. HeapMemMP manages a single buffer in shared memory from which blocks of user-specified length are allocated and freed. The ti.sdo.ipc.heaps.HeapMemMP module is modeled after SYS/BIOS 6's HeapMem module (ti.sysbios.heaps.HeapMem).

The main addition to these modules is the use of shared memory and the management of multi-processor exclusion.

The SharedRegion modules, and therefore the MessageQ module and other IPC modules that use SharedRegion, use a HeapMemMP instance internally.

The following subsections use "Heap*MP" to refer to the HeapBufMP, HeapMultiBufMP, and HeapMemMP modules.

2.5.1 Configuring a Heap*MP Module

In addition to configuring Heap*MP instances, you can set module-wide configuration properties. For example, the maxNameLen property lets you set the maximum length of heap names. The track[Max]Allocs module configuration property enables/disables tracking memory allocation statistics.

A Heap*MP instance uses a NameServer instance to manage name/value pairs.

The Heap*MP modules make the following assumptions:

- The SharedRegion module handles address translation between a virtual shared address space and the local processor's address space. If the memory address spaces are identical across all processors, or if a single processor is being used, no address translation is required and the SharedRegion module must be appropriately configured.
- Both processors must have the same endianness.

2.5.2 Creating a Heap*MP Instance

Heaps can be created dynamically. You use the Heap*MP_create() functions to dynamically create Heap*MP instances. As with other IPC modules, before creating a Heap*MP instance, you initialize a Heap*MP_Params structure and set fields in the structure to the desired values. When you create a heap, the shared memory is initialized and the Heap*MP object is created in local memory. Only the actual buffers and some shared information reside in shared memory.

The following code example initializes a HeapBufMP_Params structure and sets fields in it. It then creates and registers an instance of the HeapBufMP module.

```
/* Create the heap that will be used to allocate messages. */
HeapBufMP_Params_init(&heapBufMPParams);
heapBufMPParams.regionId      = 0;        /* use default region */
heapBufMPParams.name          = "myHeap";
heapBufMPParams.align         = 256;
heapBufMPParams.numBlocks     = 40;
heapBufMPParams.blockSize     = 1024;
heapBufMPParams.gate          = NULL;     /* use system gate */
heapHandle = HeapBufMP_create(&heapBufMPParams);
if (heapHandle == NULL) {
    System_abort("HeapBufMP_create failed\n");
}

/* Register this heap with MessageQ */
MessageQ_registerHeap(HeapBufMP_Handle_upCast(heapHandle), HEAPID);
```

The parameters for the various Heap*MP implementations vary. For example, when you create a HeapBufMP instance, you can configure the following parameters after initializing the HeapBufMP_Params structure:

- **regionId.** The index corresponding to the shared region from which shared memory will be allocated.
- **name.** A name of the heap instance for NameServer (optional).
- **align.** Requested alignment for each block.
- **numBlocks.** Number of fixed size blocks.
- **blockSize.** Size of the blocks in this instance.
- **gate.** A multiprocessor gate for context protection.
- **exact.** Only allocate a block if the requested size is an exact match. Default is false.

Of these parameters, the ones that are common to all three Heap*MP implementations are gate, name and regionId.

*2.5.3 Opening a Heap*MP Instance*

Once a Heap*MP instance is created on a processor, the heap can be opened on another processor to obtain a local handle to the same shared instance. In order for a remote processor to obtain a handle to a Heap*MP that has been created, the remote processor needs to open it using Heap*MP_open().

The Heap*MP modules use a NameServer instance to allow a remote processor to address the local Heap*MP instance using a user-configurable string value as an identifier. The Heap*MP name is the sole parameter needed to identify an instance.

The heap must be created before it can be opened. An open call matches the call’s version number with the creator's version number in order to ensure compatibility. For example:

```
HeapBufMP_Handle heapHandle;
...

/* Open heap created by other processor. Loop until open. */
do {
    status = HeapBufMP_open("myHeap", &heapHandle);
}
while (status < 0);

/* Register this heap with MessageQ */
MessageQ_registerHeap(HeapBufMP_Handle_upCast(heapHandle), HEAPID);
```

2.5.4 Closing a Heap*MP Instance

Heap*MP_close() frees an opened Heap*MP instance stored in local memory. Heap*MP_close() may only be used to finalize instances that were opened with Heap*MP_open() by this thread. For example:

```
HeapBufMP_close(&heapHandle);
```

Never call Heap*MP_close() if some other thread has already called Heap*MP_delete().

2.5.5 Deleting a Heap*MP Instance

The Heap*MP creator thread can use Heap*MP_delete() to free a Heap*MP object stored in local memory and to flag the shared memory to indicate that the heap is no longer initialized. Heap*MP_delete() may not be used to finalize a heap using a handle acquired using Heap*MP_open()—Heap*MP_close() should be used by such threads instead.

2.5.6 Allocating Memory from the Heap

The HeapBufMP_alloc() function obtains the first buffer off the heap's freeList.

The HeapMultiBufMP_alloc() function searches through the buckets to find the smallest size that honors the requested size. It obtains the first block on that bucket.

If the "exact" field in the Heap*BufMP_Params structure was true when the heap was created, the alloc only returns the block if the blockSize for a bucket is the exact size requested. If no exact size is found, an allocation error is returned.

The HeapMemMP_alloc() function allocates a block of memory of the requested size from the heap.

For all of these allocation functions, the cache coherency of the message is managed by the SharedRegion module that manages the shared memory region used for the heap.

2.5.7 Freeing Memory to the Heap

The HeapBufMP_free() function returns an allocated buffer to its heap.

The HeapMultiBufMP_free() function searches through the buckets to determine on which bucket the block should be returned. This is determined by the same algorithm as the HeapMultiBufMP_alloc() function, namely the smallest blockSize that the block can fit into.

If the "exact" field in the Heap*BufMP_Params structure was true when the heap was created, and the size of the block to free does not match any bucket's blockSize, an assert is raised.

The HeapMemMP_free() function returns the allocated block of memory to its heap.

For all of these deallocation functions, cache coherency is managed by the corresponding Heap*MP module.

2.5.8 Querying Heap Statistics

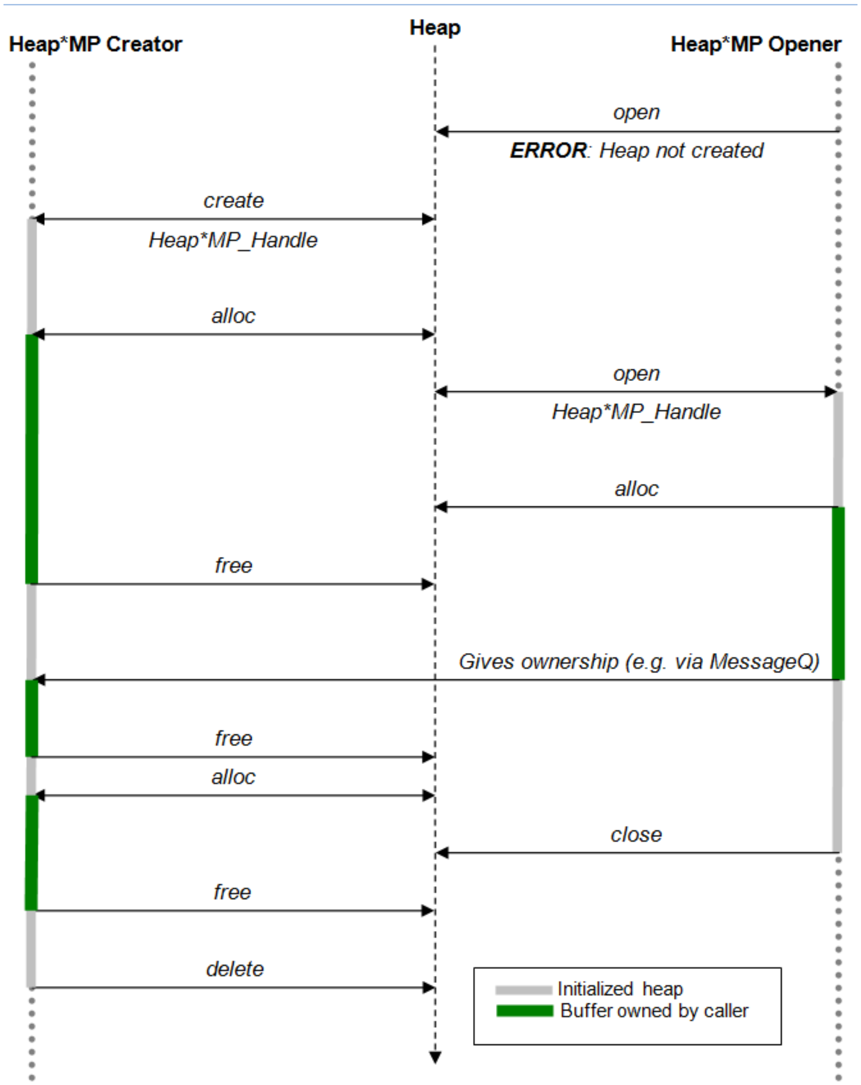
Both heap modules support use of the xdc.runtime.Memory module’s Memory_getStats() and Memory_query() functions on the heap.

In addition, the Heap*MP modules provide the Heap*MP_getStats(), Heap*MP_getExtendedStats(), and Heap*MP_isBlocking() functions to enable you to gather information about a heap.

By default, allocation tracking is often disabled in shared-heap modules for performance reasons. You can set the HeapBufMP.trackAllocs and HeapMultiBufMP.trackMaxAllocs configuration properties to true in order to turn on allocation tracking for their respective modules. Refer to the CDOC documentation for further information.

2.5.9 Sample Runtime Program Flow

The following diagram shows the program flow for a two-processor (or two-thread) application. This application creates a Heap*MP instance dynamically.



2.6 GateMP Module

Note: The GateMP module is not supported for Concerto F28M35x devices.

A GateMP instance can be used to enforce both local and remote context protection. That is, entering a GateMP can prevent preemption by another thread running on the same processor and simultaneously prevent a remote processor from entering the same gate. GateMP's are typically used to protect reads/writes to a shared resource, such as shared memory.

2.6.1 Creating a GateMP Instance

As with other IPC modules, GateMP instances can only be created dynamically.

Before creating the GateMP instance, you initialize a GateMP_Params structure and set fields in the structure to the desired values. You then use the GateMP_create() function to dynamically create a GateMP instance.

When you create a gate, shared memory is initialized, but the GateMP object is created in local memory. Only the gate information resides in shared memory.

The following code creates a GateMP object:

```
GateMP_Params gparams;
GateMP_Handle gateHandle;

...

GateMP_Params_init(&gparams);
gparams.localProtect = GateMP_LocalProtect_THREAD;
gparams.remoteProtect = GateMP_RemoteProtect_SYSTEM;
gparams.name = "myGate";
gparams.regionId = 1;
gateHandle = GateMP_create(&gparams, NULL);
```

A gate can be configured to implement remote processor protection in various ways. This is done via the params.remoteProtect configuration property. The options for params.remoteProtect are as follows:

- **GateMP_RemoteProtect_NONE.** Creates only the local gate specified by the localProtect property.
- **GateMP_RemoteProtect_SYSTEM.** Uses the default device-specific gate protection mechanism for your device. Internally, GateMP automatically uses device-specific implementations of multi-processor mutexes implemented via a variety of hardware mechanisms. Devices typically support a single type of system gate, so this is usually the correct configuration setting for params.remoteProtect.
- **GateMP_RemoteProtect_CUSTOM1 and GateMP_RemoteProtect_CUSTOM2.** Some devices support multiple types of system gates. If you know that GateMP has multiple implementations of gates for your device, you can use one of these options.

Several gate implementations used internally for remote protection are provided in the ti.sdo.ipc.gates package.

A gate can be configured to implement local protection at various levels. This is done via the params.localProtect configuration property. The options for params.localProtect are as follows:

- **GateMP_LocalProtect_NONE.** Uses the XDCtools GateNull implementation, which does not offer any local context protection. For example, you might use this option for a single-threaded local application that still needs remote protection.
- **GateMP_LocalProtect_INTERRUPT.** Uses the SYS/BIOS GateHwi implementation, which disables hardware interrupts.
- **GateMP_LocalProtect_TASKLET.** Uses the SYS/BIOS GateSwi implementation, which disables software interrupts.

- **GateMP_LocalProtect_THREAD.** Uses the SYS/BIOS GateMutexPri implementation, which is based on Semaphores. This option may use a different gate than the following option on some operating systems. When using SYS/BIOS, they are equivalent.
- **GateMP_LocalProtect_PROCESS.** Uses the SYS/BIOS GateMutexPri implementation, which is based on Semaphores.

Other fields you are required to set in the GateMP_Params structure are:

- **name.** The name of the GateMP instance.
- **regionId.** The ID of the SharedRegion to use for shared memory used by this GateMP instance.

2.6.2 Opening a GateMP Instance

Once an instance is created on a processor, the gate can be opened on another processor to obtain a local handle to the same instance.

The GateMP module uses a NameServer instance to allow a remote processor to address the local GateMP instance using a user-configurable string value as an identifier rather than a potentially dynamic address value.

```
status = GateMP_open("myGate", &gateHandle);
if (status < 0) {
    System_printf("GateMP_open failed\n");
}
```

2.6.3 Closing a GateMP Instance

GateMP_close() frees a GateMP object stored in local memory.

GateMP_close() should never be called on an instance whose creator has been deleted.

2.6.4 Deleting a GateMP Instance

GateMP_delete() frees a GateMP object stored in local memory and flags the shared memory to indicate that the gate is no longer initialized.

A thread may not use GateMP_delete() if it acquired the handle to the gate using GateMP_open(). Such threads should call GateMP_close() instead.

2.6.5 Entering a GateMP Instance

Either the GateMP creator or opener may call GateMP_enter() to enter a gate. While it is necessary for the opener to wait for a gate to be created to enter a created gate, it isn't necessary for a creator to wait for a gate to be opened before entering it.

GateMP_enter() enters the caller's local gate. The local gate (if supplied) blocks if entered on the local processor. If entered by the remote processor, GateMP_enter() spins until the remote processor has left the gate.

No matter what the params.localProtection configuration property is set to, after GateMP_enter() returns, the caller has exclusive access to the data protected by this gate.

A thread may reenter a gate without blocking or failing.

GateMP_enter() returns a "key" that is used by GateMP_leave() to leave this gate; this value is used to restore thread preemption to the state that existed just prior to entering this gate.

```
IArg key;

...
/* Enter the gate */
key = GateMP_enter(gateHandle);
```

2.6.6 Leaving a GateMP Instance

GateMP_leave() may only called by a thread that has previously entered this gate via GateMP_enter().

After this method returns, the caller must not access the data structure protected by this gate (unless the caller has entered the gate more than once and other calls to leave remain to balance the number of previous calls to enter).

```
IArg key;

...
/* Leave the gate */
GateMP_leave(gateHandle, key);
```

2.6.7 Querying a GateMP Instance

GateMP_query() returns TRUE if a gate has a given quality, and FALSE otherwise, including cases when the gate does not recognize the constant describing the quality. The qualities you can query are:

- **GateMP_Q_BLOCKING.** If GateMP_Q__BLOCKING is FALSE, the gate never blocks.
- **GateMP_Q_PREEMPTING.** If GateMP_Q_PREEMPTING is FALSE, the gate does not allow other threads to preempt the thread that has already entered the gate.

2.6.8 NameServer Interaction

The GateMP module uses a ti.sdo.utils.NameServer instance to store instance information when an instance is created and the name parameter is non-NULL. The length of this name is limited to 16 characters (by default) including the null terminator ('\0'). This length can be increased by configuring the GateMP.maxNameLen module configuration property. If a name is supplied, it must be unique for all GateMP instances.

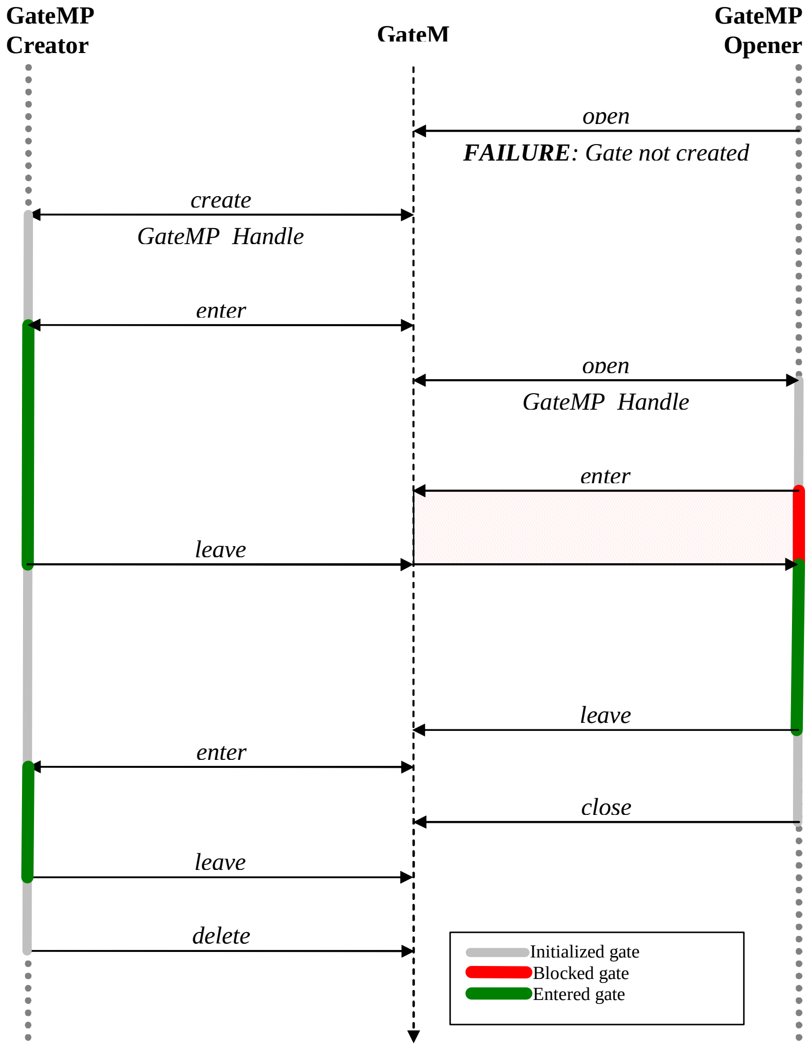
Other modules can use GateMP instances to protect access to their shared memory resources. For example, the NameServer name tables are protected by setting the "gate" property of the ti.sdo.utils.NameServer module.

These examples set the "gate" property for various modules:

```
heapBufMPParams.gate   = GateMP_getDefaultRemote();
listMPParams.gate    = gateHandle;
```

2.6.9 Sample Runtime Program Flow (Dynamic)

The following diagram shows the program flow for a two-processor (or two-thread) application. This application creates a Gate dynamically.



2.7 Notify Module

The ti.sdo.ipc.Notify module manages the multiplexing/demultiplexing of software interrupts over hardware interrupts.

Note: Notify use is the same with Concerto F28M35x devices as for other devices. See Section 5.3.1 for information about the NotfiyDriverCirc driver used with the Notify module when you are using Concerto devices.

In order to use any Notify APIs, you must call the Ipc_start() function first, usually within main(). This sets up all the necessary Notify drivers, shared memory, and interprocessor interrupts. However, note that if Ipc.setupNotify is set to FALSE, you will need call Notify_start() outside the scope of Ipc_start().

To be able to receive notifications, a processor registers one or more callback functions to an eventId by calling Notify_registerEvent(). The callback function must have the following signature:

```
Void cbFxn(UInt16 procId, UInt16 lineId, UInt32 eventId, UArg arg, UInt32 payload);
```

The Notify_registerEvent() function (like most other Notify APIs) uses a ti.sdo.utils.MultiProc ID and line ID to target a specific interrupt line to/from a specific processor on a device.

```
Int status;
armProcId = MultiProc_getId("ARM");

Ipc_start();

/* Register cbFxn with Notify. It will be called when ARM
 * sends event number EVENTID to line #0 on this processor.
 * The argument 0x1010 is passed to the callback function. */
status = Notify_registerEvent(armProcId, 0, EVENTID,
                 (Notify_FnNotifyCbck)cbFxn, 0x1010);
if (status < 0) {
    System_abort("Notify_registerEvent failed\n");
}
```

The line ID number is typically 0 (zero), but is provided for use on systems that have multiple interrupt lines between processors.

When using Notify_registerEvent(), multiple callbacks may be registered with a single event. If you plan to register only one callback function for an event on this processor, you can call Notify_registerEventSingle() instead of Notify_registerEvent(). Better performance is provided with Notify_registerEventSingle(), and a Notify_E_ALREADYEXISTS status is returned if you try to register a second callback for the same event.

Once an event has been registered, a remote processor may "send" an event by calling Notify_sendEvent(). If the specified event and interrupt line are both enabled, all callback functions registered to the event will be called sequentially.

```
while (seq < NUMLOOPS) {
    Semaphore_pend(semHandle, BIOS_WAIT_FOREVER);
    /* Semaphore_post is called by callback function*/
    status = Notify_sendEvent(armProcId, 0, EVENTID, seq, TRUE);
}
```

In this example, the seq variable is sent as the "payload" along with the event. The payload is limited to a fixed size of 32 bits.

Since the fifth argument in the previous example call to Notify_sendEvent() is TRUE, if any previous event to the same event ID was sent, the Notify driver waits for an acknowledgement that the previous event was received.

A specific event may be disabled or enabled using the Notify_disableEvent() and Notify_enableEvent() calls. All notifications on an entire interrupt line may be disabled or restored using the Notify_disable() and Notify_restore() calls. The Notify_disable() call does not alter the state of individual events. Instead, it just disables the ability of the Notify module to receive events on the specified interrupt line.

"Loopback" mode, which is enabled by default, allows notifications to be registered and sent locally. This is accomplished by supplying the processor's own MultiProc ID to Notify APIs. Line ID 0 (zero) is always used for local notifications. See the notify_loopback example in <ipc_install_dir>\packages\ti\sdo\ipc\examples\singlecore. It is important to be aware of some subtle (but important) differences between remote and local notifications:

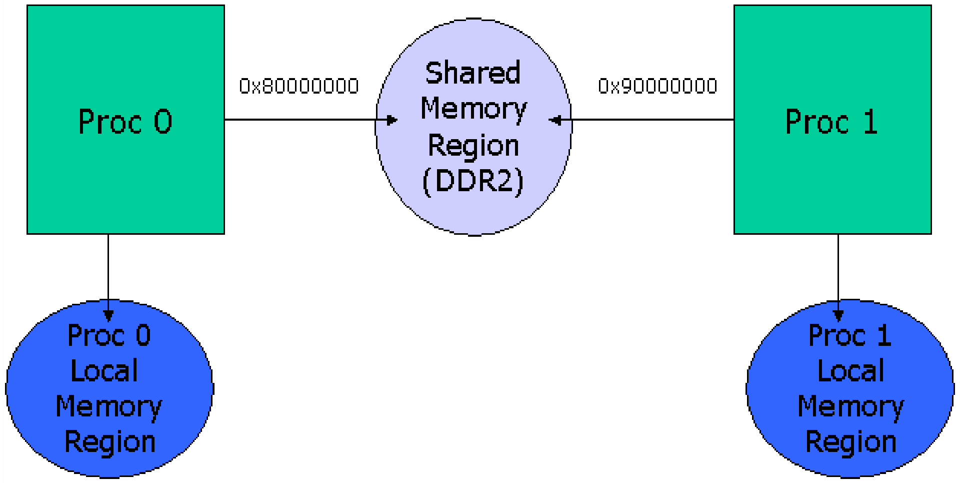
- Loopback callback functions execute in the context of the same thread that called Notify_sendEvent(). This is in contrast to callback functions called due to another processor's sent notification—such "remote" callback functions execute in an ISR context.
- Loopback callback functions execute with interrupts disabled.
- Disabling the local interrupt line causes all notifications that are sent to the local processor to be lost. By contrast, a notification sent to an enabled event on a remote processor that has called Notify_disableEvent() results in a pending notification until the disabled processor has called Notify_restore().
- Local notifications do not support events of different priorities. By contrast, Notify driver implementations may correlate event IDs with varying priorities.

2.8 SharedRegion Module

The SharedRegion module is designed to be used in a multi-processor environment where there are memory regions that are shared and accessed across different processors.

Note: The SharedRegion module is not used on Concerto F28M35x devices. Instead, the IpcMgr module (in the ti.sdo.ipc.family.f28m35x package) is used to configure access to shared memory by Concerto devices. See Section B.2.

In an environment with shared memory regions, a common problem is that these shared regions are memory mapped to different address spaces on different processors. This is shown in the following figure. The shared memory region "DDR2" is mapped into Proc0's local memory space at base address 0x80000000 and Proc1's local memory space at base address 0x90000000. Therefore, the pointers in "DDR2" need to be translated in order for them to be portable between Proc0 and Proc1. The local memory regions for Proc0 and Proc1 are not shared thus they do not need to be added to the SharedRegion module.


Proc 0
0x80000000
Shared Memory Region (DDR2)
0x90000000
Proc 1
Proc 0 Local Memory Region
Proc 1 Local Memory Region

On systems where address translation is not required, translation is a noop, so performance is not affected.

The SharedRegion module itself does not use any shared memory, because all of its state is stored locally. The APIs use the system gate for thread protection.

This module creates a shared memory region lookup table. The lookup table contains the processor's view of every shared region in the system. In cases where a processor cannot view a certain shared memory region, that shared memory region should be left invalid for that processor. Each processor has its own lookup table.

Each processor's view of a particular shared memory region can be determined by the same region ID across all lookup tables. At runtime, this table, along with the shared region pointer, is used to do a quick address translation.

The lookup table contains the following information about each shared region:

- **base.** The base address of the region. This may be different on different processors, depending on their addressing schemes.
- **len.** The length of the region. This should be should be the same across all processors.
- **ownerProcId.** MultiProc ID of the processor that manages this region. If an owner is specified, the owner creates a HeapMemMP instance at runtime. The other cores open the same HeapMemMP instance.
- **isValid.** Boolean to specify whether the region is valid (accessible) or not on this processor.
- **cacheEnable.** Boolean to specify whether a cache is enabled for the region on the local processor.
- **cacheLineSize.** The cache line size for the region. It is *crucial* that the value specified here be the same on all processors.
- **createHeap.** Boolean to specify if a heap is created for the region.
- **name.** The name associated with the region.

The maximum number of entries in the lookup table is statically configurable using the SharedRegion.numEntries property. Entries can be added during static configuration or at runtime. When you add or remove an entry in one processor's table, you must update all of the remaining processors' tables to keep them consistent. The larger the maximum number of entries, the longer it will take to traverse the lookup table when searching for the index. Therefore, keep the lookup table small for better performance and footprint.

Because each processor stores information about the caching of a shared memory region in the SharedRegion lookup table, other modules can (and do) make use of this caching information to maintain coherency and alignment when using items stored in shared memory.

In order to use the SharedRegion module, the following must be true:

- The SharedRegion.numEntries property must be the same on all processors.
- The size of a SharedRegion pointer is 32-bits wide.
- The SharedRegion lookup table must contain at least 1 entry for address translation to occur.
- Shared memory regions must not overlap each other from a single processor's viewpoint.
- Regions are not allowed to overlap from a single processor's view.

- The SharedRegion with an index of 0 (zero) is used by IPC_start() to create resource management tables for internal use by other IPC modules. Thus SharedRegion "0" must be accessible by all processors. Your applications can also make use of SharedRegion "0", but must be aware of memory limitations.

2.8.1 Adding Table Entries Statically

To create a shared region lookup table in the XDCtools configuration, first determine the shared memory regions you plan to use.

Next, specify the maximum number of entries in the lookup table with the SharedRegion.numEntries property. You can specify a value for the SharedRegion.cacheLineSize configuration property, which is the default cache line size if no size is specified for a region. You can also specify the value of the SharedRegion.translate property, which should only be set to false if all shared memory regions have the same base address on all processors. Setting the translate property to false improves performance because no address translation is performed. For example:

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
SharedRegion.cacheLineSize = 32;
SharedRegion.numEntries = 4;
SharedRegion.translate = true;
```

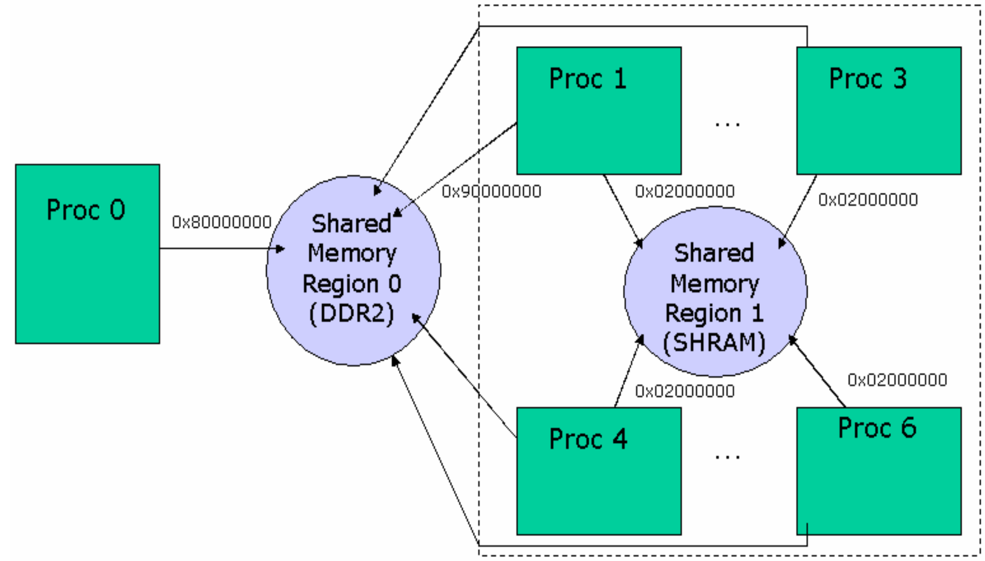
Then, use the SharedRegion.setEntryMeta() method in the configuration file to specify the parameters of the entry.

```
var SHAREDMEM      = 0x0C000000;
var SHAREDMEMSIZE  = 0x00200000;

SharedRegion.setEntryMeta(0,
    { base: SHAREDMEM,
      len: SHAREDMEMSIZE,
      ownerProcId: 0,
      isValid: true,
      cacheEnable: true,
      cacheLineSize: 128,
      createHeap: true,
      name: "internal_shared_mem"   });
```

If, during static configuration, you don't know the base address for every processor, you should set the "isValid" field for an entry for which you don’t yet know the base address to "false". Storing this information will allow it to be completed at runtime.

The following figure shows the configuration of a SharedRegion table for the system in the following figure. This system has seven processors and two shared memory regions. Region 0 ("ext") is accessible by all processors. Region 1 ("local") is accessible only by processors 1 to 6.



If the "createHeap" field is set to true, a HeapMemMP instance is created within the SharedRegion.

2.8.2 Modifying Table Entries Dynamically

In the application's C code, a shared memory region can be modified in the SharedRegion table by calling SharedRegion_setEntry().

Typically, applications configure SharedRegion table entries statically as described in the previous section, and only modify the table entries dynamically in applications where it is possible for shared memory region availability to change dynamically.

The call to SharedRegion_setEntry() must specify all the fields in the SharedRegion_Entry structure. The index specified must be the same across all processors for the same shared memory region. The index also must be smaller than the maxNumEntries property, otherwise an assert will be triggered.

```
typedef struct SharedRegion_Entry {
    Ptr base;
    SizeT len;
    UInt16 ownerProcId;
    Bool isValid;
    Bool cacheEnable;
    SizeT cacheLineSize;
    Bool createHeap;
    String name;
} SharedRegion_Entry;
```

You can use the SharedRegion_getEntry() API to fill the fields in a SharedRegion_Entry structure. Then, you can modify fields in the structure and call SharedRegion_setEntry() to write the modified fields back to the SharedRegion table.

If you want to reuse an index location in the SharedRegion table, you can call SharedRegion_clear() on all processors to erase the existing entry at that index location.

2.8.3 Using Memory in a Shared Region

Note that the SharedRegion with an index of 0 (zero) is used by IPC_start() to create resource management tables for internal use by the GateMP, NameServer, and Notify modules. Thus SharedRegion "0" must be accessible by all processors.

This example allocates memory from a SharedRegion:

```
buf = Memory_alloc(SharedRegion_getHeap(0), sizeof(Tester) * COUNT, 128, NULL);
```

2.8.4 Getting Information About a Shared Region

The shared region pointer (SRPtr) is a 32-bit portable pointer composed of an ID and offset. The most significant bits of a SRPtr are used for the ID. The ID corresponds to the index of the entry in the lookup table. The offset is the offset from the base of the shared memory region. The maximum number of table entries in the lookup table determines the number of bits to be used for the ID. An increase in the id means the range of the offset would decrease. The ID is limited to 16-bits.

Here is sample code for getting the SRPtr and then getting the real address pointer back.

```
SharedRegion_SRPtr srptr;
UInt16 id;

// Get the id of the address if id is not already known.
id = SharedRegion_getId(addr);

// Get the shared region pointer for the address
srptr = SharedRegion_getSRPtr(addr, id);

// Get the address back from the shared region pointer
addr = SharedRegion_getPtr(srptr);
```

In addition, you can use the SharedRegion_getIdByName() function to pass the name of a SharedRegion and receive the ID number of the region.

You can use the SharedRegion_getHeap() function to get a handle to the heap associated with a region using the heap ID.

You can retrieve a specific shared region's cache configuration from the SharedRegion table by using the SharedRegion_isCacheEnabled() and SharedRegion_getCacheLineSize() APIs.

Chapter 3

The Utilities Package

This chapter introduces the modules in the ti.sdo.utils package.

Topic **Page**

3.1 Modules in the Utils Package

The ti.sdo.utils package contains modules that are used as utilities by other modules in the IPC product.

- **List.** This module provides a doubly-linked list manager for use by other modules. See Section 3.2.
- **MultiProc.** This module stores processor IDs in a centralized location for multi-processor applications. See Section 3.3.
- **NameServer.** This module manages name/value pairs for use by other modules. See Section 3.4.

3.2 List Module

The ti.sdo.utils.List module provides support for creating lists of objects. A List is implemented as a doubly-linked list, so that elements can be inserted or removed from anywhere in the list. Lists do not have a maximum size.

Note: List module use is the same for Concerto F28M35x devices as for other devices.

3.2.1 Basic FIFO Operation of a List

To add a structure to a List, its first field needs to be of type List_Elem. The following example shows a structure that can be added to a List. A List has a "head", which is the front of the list. List_put() adds elements to the back of the list, and List_get() removes and returns the element at the head of the list. Together, these functions support a FIFO queue.

Run-time example: The following example demonstrates the basic List operations—List_put() and List_get().

```
/* This structure can be added to a List because the first
 * field is a List_Elem. Declared globally. */
typedef struct Rec {
    List_Elem elem;
    Int data;
} Rec;

...

List_Handle myList;                /* in main() */
Rec r1, r2;
Rec* rp;

r1.data = 100;
r2.data = 200;

/* No parameters are needed to create a List. */
myList = List_create(NULL, NULL);

/* Add r1 and r2 to the back of myList. */
List_put(myList, &(r1.elem));
List_put(myList, &(r2.elem));

/* get the records and print their data */
while ((rp = List_get(myList)) != NULL) {
    System_printf("rec: %d\n", rp->data);
}
```

The example prints the following:

```
rec: 100
rec: 200
```

3.2.2 Iterating Over a List

The List module also provides several APIs for looping over a List.

List_next() with NULL returns the element at the front of the List (without removing it). List_next() with an elem returns the next elem. NULL is returned when the end of the List is reached.

Similarly, List_prev() with NULL returns the tail. List_prev() with an elem returns the previous elem. NULL is returned when the beginning of the List is reached.

Run-time example: The following example demonstrates one way to iterate over a List once from beginning to end. In this example, "myList" is a List_Handle.

```
List_Elem   *elem = NULL;
Rec* rp;

...

/* To start the search at the beginning of the List */
rp = NULL;

/* Begin protection against modification of the List */
key = Gate_enterSystem();

while ((elem = List_next(myList, elem)) != NULL) {
     System_printf("rec: %d\n", rp->data);
}
/* End protection against modification of the List */
Gate_leaveSystem(key);
```

3.2.3 Inserting and Removing List Elements

Elements can also be inserted or removed from anywhere in the middle of a List using List_insert() and List_remove(). List_insert() inserts an element in front of the specified element. Use List_putHead() to place an element at the front of the List and List_put() to place an element at the end of the List.

List_remove() removes the specified element from whatever List it is in.

Note that List does not provide any APIs for inserting or removing elements at a given index in the List.

Run-time example: The following example demonstrates List_insert() and List_remove():

```
/* Insert r2 in front of r1 in the List. */
List_insert(myList, &(r1.elem), &(r2.elem));

/* Remove r1 from the List. */
List_remove(myList, &(r1.elem));
```

Run-time example: The following example treats the List as a LIFO stack using List_putHead() and List_get():

```
List_Elem  elem[NUMELEM];
List_Elem *tmpElem;

// push onto the top (i.e. head)
for (i = 0; i < NUMELEM; i++) {
    List_putHead(listHandle, &(elem[i]));
}

// remove the buffers in FIFO order.
while((tmpElem = List_get(listHandle)) != NULL) {
    // process tmpElem
}
```

3.2.4 Atomic List Operations

Lists are commonly shared across multiple threads in the system, which might lead to concurrent modifications of the List by different threads, which would corrupt the List. List provides several "atomic" APIs that disable interrupts before operating on the List. These APIs are List_get() List_put(), List_putHead(), and List_empty().

An atomic API completes in core functionality without being interrupted. Therefore, atomic APIs are thread-safe. An example is List_put(). Multiple threads can call this API at the same time. The threads do not have to manage the synchronization.

Other APIs—List_prev(), List_next(), List_insert(), and List_remove()—should be protected by the application.

3.3 MultiProc Module

Many IPC modules require the ability to uniquely specify and identify processors in a multi-processor environment. The MultiProc module centralizes processor ID management into one module. Most multi-processor IPC applications require that you configure this module using the MultiProc.setConfig() function in the *.cfg script. The setConfig() function tells the MultiProc module:

- The specific processor for which this application is being built.
- The processors in this cluster. A "cluster" is a set of processors within a system that share some memory and for which notification between those processors is needed.

Note: MultiProc module use is the same for Concerto F28M35x devices as for other devices.

Most systems contain a single cluster. For systems with multiple clusters, you also need to configure the numProcessors and baseIdOfCluster properties. See Section 3.3.1 for examples that configure systems with multiple clusters.

Each processor reference by the MultiProc module can be uniquely identified by either its name string or an integer ranging from 0 to MultiProc.maxProcessors - 1.

The following XDCtools configuration statements set up a MultiProc array. At runtime, the "DSP" processor running this configuration gets assigned an ID value of 2. The other processors in the system are "VIDEO" with a processor ID of 0 and "DSS" with a processor ID of 1.

```
/* DSP will get assigned processor id 2. */
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
MultiProc.setConfig("DSP", ["VIDEO", "DSS", "DSP"]);
```

The ID is a software-only setting. It does not correlate to hardware core IDs or any other type of hardware identification. For devices with more than one core, each core must have its own unique processor ID. The ID is also independent of any OS setting.

The processor ID is not always known at configuration time. It might need to be determined at initialization time via a GPIO pin, flash setting, or some other method. You can call the MultiProc_setLocalId() API (with the restriction that it must be called before module startup) to set the processor ID. However, other modules that use MultiProc need to know that the static ID will be changed during initialization. Setting the local name to NULL in the MultiProc.setConfig statement in the configuration indicates that the MultiProc_setLocalId() API will be used at runtime. Other modules that use MultiProc should act accordingly by deferring processing until the actual ID is known.

For example, the following fragment of configuration code requires that the MultiProc_setLocalId() API be run during startup to fill in the NULL processor name.

```
/* Specify startup function */
var Startup = xdc.useModule('xdc.runtime.Startup');
Startup.firstFxns.$add('&setMyId');

/* Specify MultiProc config; current processor unknown */
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
MultiProc.setConfig(null, ["CORE0", "CORE1", "CORE2"]);
```

Then, the application code could contain the following setMyID() function to be run at startup:

```
Void setMyId()
{
    UInt16 procId;
    Int    status;

    //
    // Board specific determination of processor id.
    // Example: GPIO_READ reads register of GPIO pin 5
    //
    if (GPIO_READ(5) == 0) {
        procId = 0;
    }
    else {
        procId = 1;
    }

    MultiProc_setLocalId(procId);
}
```

Your application can query the MultiProc table using various runtime APIs.

At runtime, the MultiProc_getId() call returns the MultiProc ID for any processor name. At config-time, the MultiProc.getIdMeta() call returns the same value. For example:

```
core1ProcId = MultiProc_getId("CORE1");
```

MultiProc_self() returns the processor ID of the processor running the API. For example:

```
System_printf("My MultiProc id = %d\n", MultiProc_self());
```

MultiProc_getBaseIdOfCluster() returns the MultiProc ID of the base processor in the cluster to which this processor belongs.

The MultiProc_getName() API returns that processor name if given the MultiProc ID. For example:

```
core0Name = MultiProc_getName(0);
```

MultiProc_getNumProcessors() evaluates to the total number of processors.

```
System_printf("Number of processors in the system = %d\n",
              MultiProc_getNumProcessors() );
```

MultiProc_getNumProcsInCluster() returns the number of processors in the cluster to which this processor belongs.

3.3.1 Configuring Clusters With the MultiProc Module

A "cluster" is a set of processors within a system that share some memory and for which notification between those processors is needed. If your system has multiple clusters, you need to configure the MultiProc module’s numProcessors and baseIdOfCluster properties in addition to calling the MultiProc.setConfig() function.

Notifications are not supported between different clusters.

For example, in a system with two ’C6678 devices that each use eight homogeneous cores, you could configure the first ’C6678 device as follows:

```
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
MultiProc.baseIdOfCluster = 0;
MultiProc.numProcessors = 16;
MultiProc.setConfig(null, ["CORE0", "CORE1", "CORE2",
         "CORE3", "CORE4", "CORE5", "CORE6", "CORE7"]);
```

You could configure the second ’C6678 device as follows:

```
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
MultiProc.baseIdOfCluster = 8;
MultiProc.numProcessors = 16;
MultiProc.setConfig(null, ["CORE0", "CORE1", "CORE2",
         "CORE3", "CORE4", "CORE5", "CORE6", "CORE7"]);
```

Notice that the MultiProc.numProcessors property specifies the total number of processors in the system, while the length of the array passed to setConfig() specifies the number of processors in the cluster. (If you are not using multiple clusters, the numProcessors property is configured automatically.)

The MultiProc.baseIdOfCluster property is set to the MultiProc ID number you want to use for the first processor in the array for this cluster. For example, if there are 8 processors in a cluster, the baseIdOfCluster property should be 0 for the first cluster and 8 for the second cluster.

The Ipc_start() and Ipc_attach() APIs can only be used to attach and synchronizes with processors in the same cluster.

To create a connection between cores in different clusters, you must manually create a connection using the MessageQ and ti.sdo.ipc.NameServerMessageQ modules. The NameServerMessageQ module supports NameServer requests between different clusters by using MessageQ, which in turns uses the MessageQ transport to send a NameServer request.

You can control the timeout period for the NameServerMessageQ module by configuring its timeoutInMicroSecs parameter, which defaults to 1 second. If a response is not received within the timeout period, the NameServer request returns a failure status. The NameServerRemoteNotify module also has a timeoutInMicroSecs parameter that you can configure; it defaults to wait forever.

Creating a connection between cores in different clusters allows you to call MessageQ_open() even for a core on a different cluster. Note that these calls must occur after the MessageQ heap has been registered, because they allocate memory from the heap.

Once the connection has been created, MessageQ can be used between different processors on different clusters just as it is used between different processors in the same cluster.

The following example function creates a NameServerMessageQ and TransportXXX to communicate remotely with a processor in a different cluster. The "remoteProcId" would be specified to be the MultiProc ID of the processor in the system. "TransportXXX" must be a copy-based transport that does not require any shared memory. You would need to create such a transport, because IPC does not provide one.

```
Void myRemoteCreateFunction(Uint16 remoteProcId)
{
    NameServerMessageQ_Params  nsParams;
    NameServerMessageQ_Handle  nsHandle;
    TransportXXX_Handle        tranHandle;
    TransportXXX_Params        tranParams;
    Error_Block eb;

    Error_init(&eb);

    /*
     *  Note: You must register a MessageQ heap prior to
     *  calling NameServerMessageQ_create().
     */

    /* init nsParams */
    NameServerMessageQ_Params_init(&nsParams);

    /* create driver to remote processor */
    nsHandle = NameServerMessageQ_create(
        remoteProcId, /* MultiProc ID of proc on 2nd cluster */
        &nsParams,
        &eb);
    if (nsHandle == NULL) {
        SYS_abort("NameServerMessageQ_create() failed");
    }

    /* initialize the transport parameters */
    TransportXXX_Params_init(&tranParams);

    tranHandle = TransportXXX_create(
       remoteProcId, /* MultiProc ID of proc on 2nd cluster */
       &tranParams,
       &eb);
    if (tranHandle == NULL) {
        SYS_abort("TransportXXX_create() failed");
    }
}
```

3.4 NameServer Module

The NameServer module manages local name/value pairs. This enables an application and other modules to store and retrieve values based on a name.

Note: NameServer module use is essentially the same for Concerto F28M35x devices as for other devices.

The NameServer module maintains thread-safety for its APIs. However, NameServer APIs cannot be called from an interrupt (that is, Hwi context). They can be called from Swis and Tasks.

This module supports different lengths of values. The NameServer_add() and NameServer_get() functions support variable-length values. The NameServer_addUInt32() function is optimized for UInt32 variables and constants.

The NameServer module currently does not perform any endian or word size conversion. Also there is no asynchronous support at this time.

You can create NameServer instances dynamically or statically.

To create a NameServer instance statically, you can add statements similar to the following to your XDCtools configuration script:

```
var NameServer = xdc.useModule('ti.sdo.utils.NameServer');

var nameServerParams = new NameServer.Params;
nameServerParams.maxRuntimeEntries = 10;
nameServerParams.maxNameLen = 32;
var nameServer0 = NameServer.create(“nameServer0”, nameServerParams);
```

If you want to specify the heap to be used by the NameServer module and a NameServer instance, use configuration statements similar to the following:

```
var NameServer = xdc.useModule('ti.sdo.utils.NameServer');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

var heapParams = new HeapMem.Params;
heapParams.size = 1024;
var heapMem = HeapMem.create(heapParams);

var nameServerParams = new NameServer.Params;
nameServerParams.tableHeap = heapMem;
var nameServer = NameServer.create("staticNameServer", nameServerParams);
```

To create a NameServer instance dynamically, initialize a NameServer_Params structure with NameServer_Params_init() and customize the values as needed. The parameters include the following:

- **checkExisting.** If true, NameServer check to see if a name already exists in the name/value table before adding it.
- **maxNameLen.** Specify the maximum length, in characters, of the name field in the table.
- **maxRuntimeEntries.** Specify the maximum number of name/value pairs this table can hold. If you set this parameter to NameServer_ALLOWGROWTH, then NameServer allows dynamic growth of the table.

- **maxValueLen.** Specify the maximum length, in MAUs, of the value field in the table.
- **tableHeap.** The heap to allocate the name/value table from when allocating dynamically. If this parameter is NULL, the heap used for object allocation is also used here.

After setting parameters, use NameServer_create() to create an instance. Each NameServer instance manages its own name/value table.

The following C example creates a NameServer instance dynamically. The instance allows a maximum of 10 runtime entries (instead of using ALLOWGROWTH). This example also specifies where to allocate the memory needed for the tables (instead of using the default).

```
NameServer_Handle NSHandle;
NameServer_Params params;

NameServer_Params_init(&params);
params.tableHeap = HeapStd_Handle_upCast(myHeap);
params.maxRuntimeEntries = 10;
NSHandle = NameServer_create("myTable", &params);
if (NSHandle == NULL) {
    // manage error
}
```

This example C code adds and removes entries at run-time:

```
Ptr key;

key = NameServer_addUInt32(NSHandle, "volume", 5);
if (key == NULL) {
    // manage error
}

NameServer_removeEntry(NSHandle, key);
// or
NameServer_remove(NSHandle, "volume");
```

The following example searches the NameServer instance pointed to by "handle" on the specified processor for a name-value pair with the name stored in nameToFind. It returns the value of the pair to valueBuf.

```
/* Search NameServer */
status = NameServer_get(NSHandle, nameToFind, valueBuf, sizeof(UInt32), procId);
```

Using different parameters for different table instances allows you to meet requirements like the following:

- **Size differences.** The maxValueLen parameter specifies the maximum length, in MAUs, of the value field in the table. One table could allow long values (for example, > 32 bits), while another table could be used to store integers. This customization enables better memory usage.
- **Performance.** Multiple NameServer tables can improve the search time when retrieving a name/value pair.
- **Relax name uniqueness.** Names in a specific table must be unique, but the same name can be used in different tables.

When you call NameServer_delete(), the memory for the name/values pairs is freed. You do not need to call NameServer_remove() on the entries before deleting a NameServer instance.

In addition to the functions mentioned above, the NameServer module provides the following APIs:

- **NameServer_get()** Retrieves the value portion of a local name/value pair from the specified processor.
- **NameServer_getLocal()** Retrieves the value portion of a local name/value pair.
- **NameServer_remove()** Removes a name/value pair from the table given a name.
- **NameServer_removeEntry()** Removes an entry from the table given a pointer to an entry.

NameServer maintains the name/values table in local memory, not in shared memory. However the NameServer module can be used in a multiprocessor system. The module communicates with other processors via NameServer Remote drivers, which are implementations of the INameServerRemote interface. The communication to the other processors is dependent on the Remote drivers implementation. When a remote driver is created, it registers with NameServer via the NameServer_registerRemoteDriver() API.

The NameServer module uses the MultiProc module to identify different processors. Which remote processors to query and the order in which they are queried is determined by the procId array passed to the NameServer_get() function.

Chapter 4

Porting IPC

This chapter provides an overview of the steps required to port IPC to new devices or systems.

4.1 Interfaces to Implement

When porting IPC to new devices, you may need to create custom implementations of the following interfaces. You may find that the provided implementations of these interfaces meet your needs, so don't assume that you will need to create custom implementation in all cases.

- "IInterrupt" for use by Notify. The interface definition is in ti.sdo.ipc.notifyDrivers.IInterrupt.
- "IGateMPSupport" for use by GateMP. The interface definition is in ti.sdo.ipc.interfaces.IGateMPSupport.
- "IMessageQTransport" and "ITransportSetup" for use by MessageQ. Interface definitions are in ti.sdo.ipc.interfaces.IMessageQTransport and ti.sdo.ipc.interfaces.ITransportSetup.
- "INotifyDriver" for use by Notify. The interface definition is in ti.sdo.ipc.interfaces.INotifyDriver.
- "INotifySetup" module, which defines interrupt mappings, for use by Notify. The interface definition is in ti.sdo.ipc.interfaces.INotifySetup.

For details about the interfaces, see the IPC online documentation.

4.2 Other Porting Tasks

You will likely need to specify custom shared region(s) in your configuration file. For details, see Section 2.8, *SharedRegion Module*.

Optionally, you may implement custom Heaps and hardware-specific versions of other IPC modules.

Chapter 5

Optimizing IPC Applications

This chapter provides hints for improving the runtime performance and shared memory usage of applications that use IPC.

Topic Page

5.1 Compiler and Linker Optimization

You can optimize your application for better performance and code size or to give you more debugging information by selecting different ways of compiling and linking your application. For example, you can do this by linking with versions of the SYS/BIOS and IPC libraries that were compiled differently.

The choices you can make related to compiler and linker optimization are located in the following places:

- **RTSC Build-Profile.** You see this field when you are creating a new CCS project or modifying the CCS Build settings. We recommend that you use the "release" setting. The "release" option is preferred even when you are creating and debugging an application; the "debug" option is mainly intended for internal use by Texas Instruments. The "release" option results in a somewhat smaller executable that can still be debugged. This build profile primarily affects how Codec Engine and some device drivers are built.

Note: The "whole_program" and "whole_program_debug" options for the RTSC Build-Profile have been deprecated, and are no longer recommended. The option that provides the most similar result is to set the BIOS.libType configuration property to BIOS.LibType_Custom.

- **CCS Build Configuration.** This setting in the CCS Build settings allows you to choose between and customize multiple build configurations. Each configuration can have the compiler and linker settings you choose.

- **BIOS.libType configuration property.** You can set this property in XGCONF or by editing the .cfg file in your project. This property lets you select from two pre-compiled versions of the SYS/BIOS and IPC libraries or to have a custom version of the SYS/BIOS and IPC libraries compiled based on the needs of your application. See the table and discussion that follow for more information.

The options for the BIOS.libType configuration property are as follows:

BIOS.libType	Compile Time	Logging	Code Size	Run-Time Performance
Instrumented (BIOS.LibType_Instrumented)	Fast	On	Good	Good
Non-Instrumented (BIOS.LibType_NonInstrumented)	Fast	Off	Better	Better
Custom (BIOS.LibType_Custom)	Fast (slow first time)	As configured	Best	Best
Debug (BIOS.LibType_Debug)	Slower	As configured	--	--

- **Instrumented.** (default) This option links with pre-built SYS/BIOS (and IPC) libraries that have instrumentation available. All Asserts and Diags settings are checked. Your configuration file can enable or disable various Diags and logging related settings. However, note that the checks to see if Diags are enabled before outputting a Log event are always performed, which has an impact on performance even if you use the ALWAYS_ON or ALWAYS_OFF setting. The resulting code size when using this option may be too large to fit on some targets, such as C28x and MSP430. This option is easy to use and debug and provides a fast build time.
- **Non-Instrumented.** This option links with pre-built SYS/BIOS (and IPC) libraries that have instrumentation turned off. No Assert or Diag settings are checked, and logging information is not available at run-time. The checking for Asserts and Diags is compiled out of the libraries, so run-time performance and code size are optimized. Checking of Error_Blocks and handling errors in ways other than logging an event are still supported. This option is easy to use and provides a fast build time.
- **Custom.** This option builds custom versions of the SYS/BIOS (and IPC) libraries that contain the modules and APIs that your application needs to access. If you have not used a particular module in your .cfg file or your C code (and it is not required internally by a SYS/BIOS module that is used), that module is not contained in the custom libraries compiled for your application. This option provides the best run-time performance and best code size given the needs of your application. Instrumentation is available to whatever extent your application configures it.

  The first time you build a project with the custom libType, the build will be longer. The custom libraries are stored in the "src" directory of your project. Subsequent builds may be faster; libraries do not need to be rebuilt unless you change one of the few configuration properties that affect the build settings, or you use an additional module that wasn't already used in the previous configuration.

  Note: If you disable SYS/BIOS Task or Swi scheduling, you must use the "custom" option in order to successfully link your application.

The custom option uses program optimization that removes many initialized constants and small code fragments (often "glue" code) from the final executable image. Such classic optimizations as constant folding and function inlining are used, including across module boundaries. The custom build preserves enough debug information to make it still possible to step through the optimized code in CCS and locate global variables.

- **Debug.** This option is not recommended; it is intended for internal use by Texas Instruments developers.

The following example statements set the BIOS.libType configuration property:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
```

If you use the custom option for the BIOS.libType, you can also set the BIOS.customCCOpts property to customize the C compiler command-line options used when compiling the SYS/BIOS libraries. If you want to change this property, it is important to first examine and understand the default command-line options used to compile the SYS/BIOS libraries for your target. You can see the default in XGCONF or by placing the following statement in your configuration script and building the project:

```
print("customCCOpts =", BIOS.customCCOpts);
```

Be careful not to cause problems for the SYS/BIOS compilation when you modify this property. For example, the --program_level_compile option is required. (Some --define and --include_path options are used on the compiler command line but are not listed in the customCCOpts definition; these also cannot be removed.)

For example, to create a debuggable custom library, you can remove the -o3 option from the BIOS.customCCOpts definition by specifying it with the following string for a C64x+ target:

```
BIOS.customCCOpts = "-mv64p --abi=eabi -q -mi10 -mo -pdr -pden -pds=238 -pds=880
  -pds1110  --embed_inline_assembly --program_level_compile -g";
```

5.2 Optimizing Runtime Performance

You can use one or more of the following techniques to improve the runtime performance of IPC applications:

- After you have finished debugging an application, you can disable asserts and logging with the following configuration statements:

  ```
  var Diags = xdc.useModule("xdc.runtime.Diags");
  var Defaults = xdc.useModule('xdc.runtime.Defaults');
  Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;
  Defaults.common$.logger = null;
  ```

- If shared memory has the same address on all processors, you can use the following configuration statement to set the SharedRegion.translate property to false. See Section 2.8.1 for more about SharedRegion configuration.

  ```
  SharedRegion.translate = false;
  ```

- Ensure that code, data, and shared data are all placed in cacheable memory. Refer to the SYS/BIOS documentation for information on how to configure a cache. See the *TI SYS/BIOS Real-time Operating System v6.x User's Guide* (SPRUEX3) for details.

- You can reduce contention between multiple processors and multiple threads by creating a new gate for use by a new IPC module instance. Leaving the params.gate property set to NULL causes the default system GateMP instance to be used for context protection. However, in some cases it may be optimal to create a new GateMP instance and supply it to the instance creation. See Section 2.6.1 for more information. For example:

```
GateMP_Params gateParams;
GateMP_Handle gateHandle;
HeapBufMP_Params heapParams;

GateMP_Params_init(&gateParams);
gateHandle = GateMP_create(&gateParams);

HeapBufMP_Params_init(&heapParams);
heapParams.gate = gateHandle;
```

- If a unicache is shared between two cores in shared memory and you expect to share certain IPC instances (such as a GateMP or ListMP) solely between those two cores, you may be able to improve performance by creating a SharedRegion with cache disabled for use between those two cores only. Since region 0 needs to be accessible by all cores on a system, region 1 can be created with a cache line size of 0 and a cacheEnable configuration of FALSE. Any IPC instance created within a SharedRegion inherits the cache settings (the cacheEnabled flag and the cacheLineSize) from this region. Therefore, unnecessary cache operations can be avoided by creating an instance in region 1.

  The following configuration statements create a SharedRegion with the cache disabled (on OMAP4430):

```
SharedRegion.setEntryMeta(1, /* Create shared region 1 */
   { base: 0x86000000,
     len: 0x10000,
     ownerProcId: 0,
     isValid: true,
      cacheEnabled: false, /* Cache operations unneeded */
     cacheLineSize: 0, /* Cache padding unneeded */
     name: "DDR2",
   });
```

  The following C code creates a HeapBufMP instance in this SharedRegion:

```
HeapBufMP_Params heapParams;
HeapBufMP_Handle heapHandle;

HeapBufMP_Params_init(&heapParams);
heapParams.regionId = 1;

heapHandle = HeapBufMP_create(&heapParams);
```

  This heap can be used by either of the Cortex M3 cores on an OMAP4430, because they both share a unicache. Do not use this heap (or anything else belonging to a SharedRegion with caching disabled) from any other processor if the shared memory belonging to the SharedRegion is cacheable.

5.3 Optimizing Notify and MessageQ Latency

By default, IPC applications are configured to use the ti.sdo.ipc.notifyDrivers.NotifyDriverShm Notify driver and the ti.sdo.ipc.transports.TransportShm MessageQ transport. These modules are used by default because they offer backward compatibility with older IPC/SysLink releases. In addition, these modules may offer functionality not supported by their newer, lower-latency counterparts.

If your application does not need functionality provided only by the default Notify drivers or MessageQ transport, you can reduce the latency by switching to alternative MessageQ transports and/or Notify drivers.

5.3.1 Choosing and Configuring Notify Drivers

To switch to a different Notify driver, set the Notify.SetupProxy configuration to the family-specific Notify setup module. For example, the following statements configure an application on the DM6446 to use the NotifyDriverCirc driver for that device:

```
var Notify = xdc.useModule(‘ti.sdo.ipc.Notify’);
Notify.SetupProxy = xdc.useModule(‘ti.sdo.ipc.family.dm6446.NotifyCircSetup’);
```

IPC provides the following Notify drivers. Each has a corresponding setup module that should be used as the Notify.SetupProxy module.

Modules and Description	Supports Disabling/ Enabling Events	Latency
ti.sdo.ipc.notifyDrivers.**NotifyDriverShm** ti.sdo.ipc.family.<*family*>.NotifySetup This shared-memory Notify driver offers room for a single pending notification in shared memory per event. This is the default driver on non-F28M35x devices.	Yes	Default
ti.sdo.ipc.notifyDrivers.**NotifyDriverCirc** ti.sdo.ipc.family.<*family*>.NotifyCircSetup This shared-memory Notify driver uses a circular buffer to store notifications. Unlike NotifyDriverShm, this driver stores all notifications in the same circular buffer (whose size is configurable).	No	Better than NotifyDriverShm
ti.sdo.ipc.family.ti81xx.**NotifyDriverMbx** ti.sdo.ipc.family.ti81xx.NotifyMbxSetup This TI81xx-only Notify driver uses the hardware mailbox. This driver is not usable by other devices. Notifications are stored in hardware mailbox queues present on TI81xx devices.	No	Better than NotifyDriverCirc and NotifyDriverShm
ti.sdo.ipc.family.f28m35x.**NotifyDriverCirc** ti.sdo.ipc.family.f28m35x.NotifyCircSetup This F28M35x-only (Concerto) shared-memory Notify driver uses a circular buffer to store notifications. This is the only Notify driver that works with the F28M35x device.	No	-- only Notify driver for F28M35x --

5.3.2 Choosing and Configuring MessageQ Transports

Similarly, to use an alternative MessageQ transport, configure the MessageQ.SetupTransportProxy property to use the transport's corresponding Transport Setup proxy. For example, to use the TransportShmNotify module, use the following configuration:

```
var MessageQ = xdc.module('ti.sdo.ipc.MessageQ');
MessageQ.SetupTransportProxy =
      xdc.module('ti.sdo.ipc.transports.TransportShmNotifySetup');
```

Unlike the Notify setup modules, Transport setup modules are generally not family-specific; most are located in the ti.sdo.ipc.transports package.

IPC provides the following transports. Each has a corresponding setup module for use as the MessageQ.SetupTransportProxy module.

Modules and Description	Transport Speed
ti.sdo.ipc.transports.**TransportShm** ti.sdo.ipc.transports.TransportShmSetup This shared-memory MessageQ transport uses ListMP to temporarily queue messages in shared memory before the messages are moved to the destination queue. This transport is typically slowest because of the overhead of queuing messages using a linked list. This is the default MessageQ transport on non-F28M35x devices.	Slowest
ti.sdo.ipc.transports.**TransportShmCirc** ti.sdo.ipc.transports.TransportShmCircSetup This shared-memory MessageQ transport uses a fixed-length circular buffer to temporarily queue messages in shared memory before the messages are moved to the destination queue. This transport is typically faster than TransportShm because of the efficiencies gained by using a circular buffer instead of a linked list.	Medium
ti.sdo.ipc.transports.**TransportShmNotify** ti.sdo.ipc.transports.TransportShmNotifySetup This shared-memory MessageQ transport does no buffering before the messages are moved to the destination queue. Because of the lack of buffering, this transport tends to offer lower MessageQ latency than either TransportShm or TransportShm. However, If messages aren't received quickly enough by the receiver, the sender may spin while waiting for the receiver to move the message to its local queue.	Fastest, but depends on fast processing of messages by receiver

5.4 Optimizing Shared Memory Usage

You can use one or more of the following techniques to reduce the shared memory footprint of IPC applications:

- If some connections between processors are not needed, it is not necessary to attach to those cores. To selectively attach between cores, use pair-wise synchronization as described in Section 2.2.1. Your C code must call Ipc_attach() for processors you want to connect to if you are using pair-wise synchronization. The following configuration statement causes the Ipc module to expect pair-wise synchronization.

```
Ipc.procSync = Ipc.ProcSync_PAIR;
```

At run-time, only call Ipc_attach() to a remote processor if one or more of the following conditions is true:

- The remote processor is the owner of region 0.
- It is necessary to send Notifications between this processor and the remote processor.
- It is necessary to send MessageQ messages between this processor and the remote processor.
- It is necessary for either the local or remote processor to open a module instance using *MODULE*_open() that has been created on the other processor.

- Configure the Ipc.setEntryMeta property to disable components of IPC that are not required. For example, if an application uses Notify but not MessageQ, disabling MessageQ avoids the creation of MessageQ transports during Ipc_attach().

```
/* To avoid wasting shared mem for MessageQ transports */
for (var i = 0; i < MultiProc.numProcessors; i++) {
    Ipc.setEntryMeta({
        remoteProcId: 1,
        setupMessageQ: false,
    });
}
```

- Configure Notify.numEvents to a lower number. The default value of 32 is often significantly more than the total number of Notify events required on a system. See Section 2.7 for more information.

  For example, a simple MessageQ application may simply use two events (one for NameServer and one for the MessageQ transport). In this case, we can optimize memory use with the following configuration:

```
var Notify = xdc.useModule('ti.sdo.ipc.Notify');

/* Reduce the total number of supported events from 32 to 2 */
Notify.numEvents = 2;

var NameServerRemoteNotify = xdc.useModule('ti.sdo.ipc.NameServerRemoteNotify');
NameServerRemoteNotify.notifyEventId = 1;

var TransportShm = xdc.useModule('ti.sdo.ipc.transports.TransportShm');
TransportShm.notifyEventId = 0;
```

- Reduce the cacheLineSize property of a SharedRegion to reflect the actual size of the cache line. IPC uses the cacheLineSize setting to pad data structures in shared memory. Padding is required so that cache write-back and invalidate operations on data in shared memory do not affect the cache status of adjacent data. The larger the cacheLineSize setting, the more shared memory is used for the sole purpose of padding. Therefore, the cacheLineSize setting should optimally be set to the actual size of the cache line. The default cacheLineSize for SharedRegion is 128. Using the correct size has both performance and size benefits.

The following example (for C6472) sets the cacheLineSize property to 64 because the shared L2 memory has this cache line size.

```
SharedRegion.setEntryMeta(0,
    { base: SHAREDMEM,
      len: SHAREDMEMSIZE,
      ownerProcId: 0,
      isValid: true,
      cacheLineSize: 64, /* SL2 cache line size = 64 */
      name: "SL2_RAM",
    });
```

5.5 Optimizing Local Memory Usage

If the Custom1 and Custom2 GateMP proxies will never be used, make sure they are both plugged with the ti.sdo.ipc.gates.GateMPSupportNull GateMP delegate. By default, GateMP plugs the Custom1 proxy with the GatePeterson delegate. A considerable amount of local memory is reserved for use by GatePeterson. You can plug the Custom1 proxy with the GateMPSupportNull delegate by adding the following configuration statements to your application:

```
var GateMP = xdc.useModule('ti.sdo.ipc.GateMP');
GateMP.RemoteCustom1Proxy = xdc.useModule('ti.sdo.ipc.gates.GateMPSupportNull');
```

5.6 Optimizing Code Size

This section provides tips and suggestions for minimizing the code size of a SYS/BIOS-based application that uses IPC.

- For a number of ways to configure SYS/BIOS that reduce code size by using custom built SYS/BIOS libraries and by disabling various features, see Section E.3 of the *TI SYS/BIOS Real-time Operating System v6.x User’s Guide* (SPRUEX3). In particular, after you have debugged your code, disabling Asserts as follows helps reduce the size of your code.

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtimg.Diags');
Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;
```

- The NotifyDriverCirc notification driver and the TransportShmNotify or TransportShmCirc MessageQ transports described in Section 5.3 use less code space than the default Notify driver and MessageQ transport.
- You can reduce code size by not using the HeapBufMP Heap implementation. Since IPC uses the HeapMemMP implementation internally, using HeapMemMP in your application does not increase the code size. However, you should be aware that, depending on how your application uses heaps, HeapMemMP may lead to problems with heap fragmentation. See Section 2.5 for more about Heap implementations.

Appendix A

Rebuilding IPC

This appendix describes how to rebuild the IPC source code.

Topic **Page**

A.1 Overview

The IPC product includes source files and build scripts that allow you to modify the IPC sources and rebuild its libraries. You can do this in order to modify, update, or add functionality. If you edit the IPC source code and/or corresponding build scripts, you must also rebuild IPC in order to create new libraries containing these modifications.

Note that you can cause the IPC (and SYS/BIOS) libraries to be rebuilt as part of the application build within CCS. The custom-built libraries will be stored with your CCS project and will contain only modules and APIs that your application needs to access. You can cause such a custom build to occur by configuring the BIOS.libType property as follows. See the *SYS/BIOS 6 User's Guide* (SPRUEX3) for details.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
```

Caution: This appendix provides details about rebuilding the IPC source code. We strongly recommend that you copy the IPC installation to a directory with a different name and rebuild that copy, rather than rebuilding the original installation.

For information about building IPC applications (that is, applications that use IPC), see the *SYS/BIOS 6 User's Guide* (SPRUEX3).

A.2 Prerequisites

In order to rebuild IPC, the SYS/BIOS, XDCtools, and IPC products must all be installed. The SYS/BIOS installation location must be referenced in the definition of the XDCPATH environment variable.

It is important to build IPC with compatible versions of XDCtools and SYS/BIOS. To find out which versions are compatible, see the “Dependencies” section of the Release Notes in the top-level directory of your IPC installation.

Note: You should generally avoid installing the various Texas Instruments tools and source distributions in directories that have spaces in their paths.

A.3 Build Procedure

Rebuilding IPC itself from the provided source files is straightforward, whether you are using the TI compiler toolchain or the GNU GCC toolchain.

IPC ships with an `ipc.mak` file in the top-level installation directory. This makefile enables you to easily (re)build IPC using your choice of compilers and desired "targets". A target incorporates a particular ISA and a runtime model.

The instructions in this section can be used to build IPC on Windows or Linux. If you are using a Windows machine, you can use the regular DOS command shell provided with Windows. However, you may want to install a Unix-like shell, such as Cygwin.

For Windows users, the XDCtools top-level installation directory contains `gmake.exe`, which is used in the commands that follow to run the Makefile. The gmake utility is a Windows version of the standard GNU "make" utility provided with Linux.

If you are using Linux, change the "gmake" command to "make" in the commands that follow.

For these instructions, suppose you have the following directories, where ## is part of the version:

- `<ccs_install_dir>/bios_6_##_##_##` — The location where you installed SYS/BIOS.
- `<ccs_install_dir>/xdctools_3_##_##_##` — The location where you installed XDCtools.

The following steps refer to the top-level directory of the XDCtools installation as *`<xdc_install_dir>`*. They refer to the top-level directory of the SYS/BIOS installation as *`<bios_install_dir>`*.

Follow these steps to rebuild IPC:

1. If you have not already done so, install XDCtools and SYS/BIOS.
2. Make a copy of the IPC installation that you will use when rebuilding. This leaves you with an unmodified installation as a backup. The full path to this directory cannot contain any spaces. For example, use commands similar to the following on Windows:

```
mkdir c:\ti\copy-ipc_1_##_##_##

copy c:\ti\ipc_1_##_##_## c:\ti\copy-ipc_1_##_##_##
```

Or, use the a command similar to the following on Linux:

```
cp -r $BASE/ipc_1_##_##_##/* $BASE/copy-ipc_1_##_##_##
```

3. Make sure you have access to compilers for any targets for which you want to be able be able to build applications using the rebuilt IPC. Note the path to the directory containing the executable for each compiler. These compilers can include Texas Instruments compilers, GCC compilers, and any other command-line compilers for any targets supported by IPC.
4. If you are using Windows and the gmake utility provided in the top-level directory of the XDCtools installation, you should add the `<xdc_install_dir>` to your PATH environment variable so that the gmake executable can be found.
5. You may remove the top-level doc directory located in `copy-ipc_1_##_##_##/docs` if you need to save disk space.
6. At this point, you may want to add the remaining files in the copy of the IPC installation tree to your Software Configuration Management (SCM) system.
7. Open the `copy-ipc_1_##_##_##/ipc.mak` file with a text editor, and make the following changes for any options you want to hardcode in the file. (You can also set these options on the command line if you want to override the settings in the `ipc.mak` file.)
   - Ignore the lines near the beginning of the file that specify subdirectories within the IPC product. These definitions are used internally, but few users will have a need to change them.
   - Specify the location of XDCtools. For example:

     ```
     # Set up dependencies
     XDC_INSTALL_DIR ?= <ccs_install_dir>/xdctools_3_##_##_##
     BIOS_INSTALL_DIR ?= <ccs_install_dir>/bios_6_##_##_##
     ```

   - Specify the location of the compiler executable for all targets you want to be able to build. Use only the directory path; do not include the name of the executable file. Any targets for which you do not specify a compiler location will be skipped during the build. For example, on Linux you might specify the following:

     ```
     ti.targets.C28_float ?= /opt/ti/ccsv5/tools/compiler/c2000
     ti.targets.arm.elf.M3 ?= /opt/ti/ccsv5/tools/compiler/tms470
     gnu.targets.arm.M3 ?= $TOOLS/gcc/bin/arm-none-eabi-gcc
     ```

     Similarly, on Windows you might specify the following compiler locations:

     ```
     ti.targets.C28_float ?= c:/ti/ccsv5/tools/compiler/c2000
     ti.targets.arm.elf.M3 ?= c:/ti/ccsv5/tools/compiler/tms470
     gnu.targets.arm.M3 ?= c:/tools/gcc/bin/arm-none-eabi-gcc
     ```

   - If you need to add any repositories to your XDCPATH (for example, to reference the packages directory of another component), you should edit the XDCPATH definition.
   - You can uncomment the line that sets XDCOPTIONS to “v” if you want more information output during the build.
8. Clean the IPC installation with the following commands. (If you are running the build on Linux, change all "`gmake`" commands to "`make`".)

   ```
   cd <ipccopy_install_dir>

   gmake -f ipc.mak clean
   ```

9. Run the `ipc.mak` file to build IPC as follows. (Remember, if you are running the build on Linux, change all "`gmake`" commands to "`make`".)

```
gmake -f ipc.mak
```

10. If you want to specify options on the command line to override the settings in `ipc.mak`, use a command similar to the following.

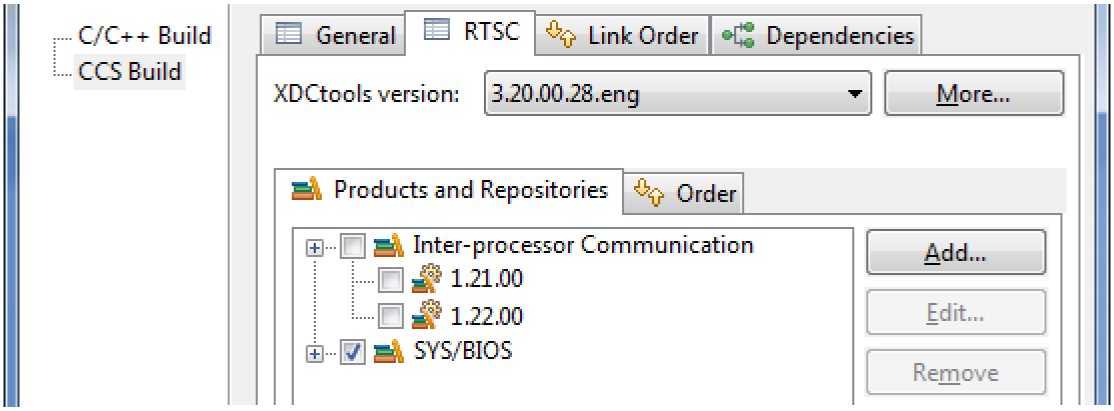
```
gmake -f ipc.mak XDC_INSTALL_DIR=<xdc_install_dir> gnu.targets.arm.M3=<compiler_path>
```

For details about the XDCPATH environment variable, see http://rtsc.eclipse.org/docs-tip/Managing_the_Package_Path in the RTSC-pedia. For more about the “xdc” command line, see http://rtsc.eclipse.org/docs-tip/Command_-_xdc.

A.4 Building Your Project Using a Rebuilt IPC

To build your application using the version of IPC you have rebuilt, you must point your project to this rebuilt version by following these steps:

1. Open CCS and select the application project you want to rebuild.
2. Right-click on your project and choose **Build Properties**. If you have a configuration project that is separate from your application project, open the build properties for the configuration project.
3. In the **CCS Build** category of the Properties dialog, choose the **RTSC** tab.
4. Under the **Products and Repositories** tab, uncheck *all* the boxes for IPC. This ensures that no version is selected.

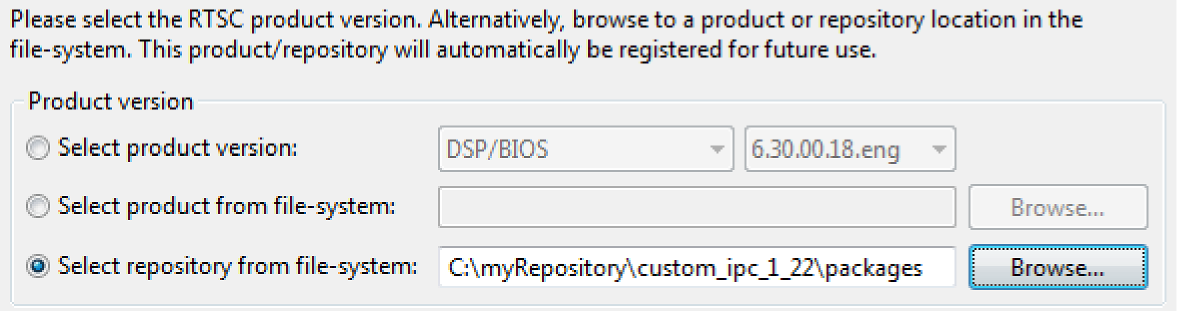


XDCtools version: 3.20.00.28.eng

□ Inter-processor Communication
  □ 1.21.00
  □ 1.22.00
☑ SYS/BIOS

5. Click the **Add** button next to the **Products and Repositories** tab.

6. Choose **Select repository from file-system**, and browse to the “packages” directory of the location where you copied and rebuilt IPC. For example, the location may be C:\myIpcBuilds\custom_ipc_1_22_##-##\packages.



Please select the RTSC product version. Alternatively, browse to a product or repository location in the file-system. This product/repository will automatically be registered for future use.

Product version

○ Select product version: DSP/BIOS 6.30.00.18.eng

○ Select product from file-system:

● Select repository from file-system: C:\myRepository\custom_ipc_1_22\packages

7. Click **OK** to apply these changes to the project.
8. You may now rebuild your project using the re-built version of IPC.

Appendix B

Using IPC on Concerto Devices

This appendix provides target-specific information about using IPC on Concerto devices.

B.1 Overview

SYS/BIOS supports both the ARM M3 and the ‘C28x cores on Concerto F28M35x devices. This allows you to use the same SYS/BIOS and IPC APIs on both processors and to use IPC for communication between the two cores. The following table identifies which IPC modules are used and not used with Concerto devices:

Table 5-1: IPC modules used with Concerto

Modules Used with Concerto	Modules Not Used or Supported	Notes
MessageQ		MessageQ usage is the same.
Notify		Notify usage is the same.
MultiProc		MultiProc configuration is the same.
IpcMgr (in ti.sdo.ipc.family.f28m35x)	Ipc	IpcMgr must be configured in place of the Ipc module when using Concerto. See Section B.2.
NotifyDriverCirc (in ti.sdo.ipc.family.f28m35x)	NotifyDriverShm NotifyDriverMbx	See Section 5.3.1.
TransportCirc (in ti.sdo.ipc.family.f28m35x)	TransportShm, TransportShmNotify	See Section 5.3.2.
List		List usage is the same.
NameServer		NameServer usage is the same.
	GateMP	Shared gates are not supported with Concerto.

Table 5-1: IPC modules used with Concerto

Modules Used with Concerto	Modules Not Used or Supported	Notes
HeapBuf (from SYS/BIOS)	Heap*MP	Shared heaps are not supported with Concerto.
	ListMP	Shared lists are not supported with Concerto.
	SharedRegion	IpcMgr is used instead of SharedRegion to specify the location of shared memory with Concerto.

In addition, you should be aware of the following special issues:

- No caching is performed on Concerto devices. Ignore any information about caching in the IPC documentation.
- Concerto provides a shared timestamp counter that can be read by either core. SYS/BIOS manages this counter with the ti.sysbios.family.[c28|arm].f28m35x.TimestampProvider modules. The Timestamp_get32() APIs use this counter to provide a common timestamp on both M3 and C28x cores. This is useful when logging on both cores and debugging multi-core issues.

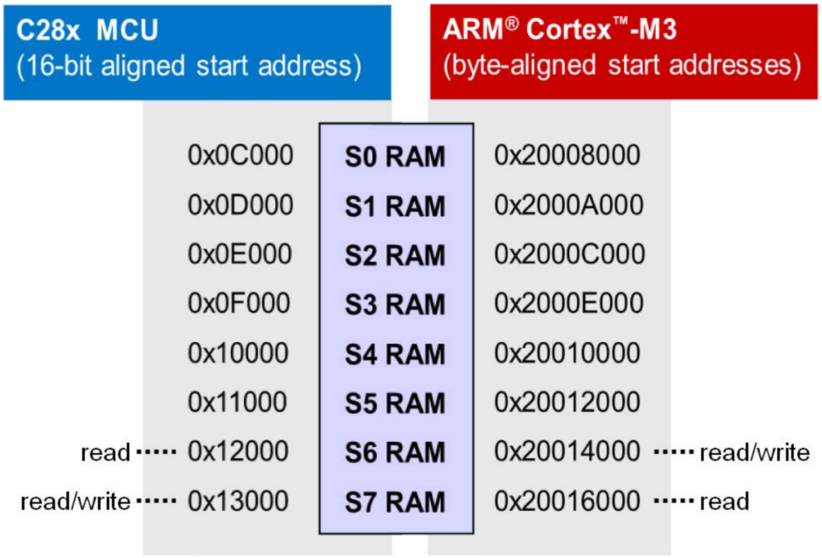
B.2 Configuring Applications with IpcMgr

The ti.sdo.ipc.family.f28m35x.IpcMgr module is used only for Concerto F28M35x devices. You use IpcMgr instead of the ti.sdo.ipc.Ipc module. That is, your application should not call Ipc_start() or Ipc_attach().

The IpcMgr module statically configures which shared memory segments to enable between the M3 and ‘C28 processors. No IpcMgr APIs need to be called at runtime. Instead, the drivers for IPC are created during this module's startup function, which runs internally. The internal startup function also synchronizes the M3 and ‘C28 processors.

Concerto devices have 8 segments of shared RAM. Each segment has 8 KB. Only one core can have read/write access to a shared memory segment at a time. The other core has read access to that segment. When configuring M3 and ‘C28 applications, you must specify the shared memory read and write addresses that IPC should use. Your application can use other shared memory segments as needed.

For example, suppose you want to configure the Concerto with the M3 processor writing to the S6 segment of shared RAM and the ‘C28x writing to the S7 segment of shared RAM. The following diagram shows the local addresses used to point to the start of the shared memory segment from both processors:



	C28x MCU (16-bit aligned start address)		ARM® Cortex™-M3 (byte-aligned start addresses)	
	0x0C000	S0 RAM	0x20008000	
	0x0D000	S1 RAM	0x2000A000	
	0x0E000	S2 RAM	0x2000C000	
	0x0F000	S3 RAM	0x2000E000	
	0x10000	S4 RAM	0x20010000	
	0x11000	S5 RAM	0x20012000	
read	0x12000	S6 RAM	0x20014000	read/write
read/write	0x13000	S7 RAM	0x20016000	read

The IpcMgr module configuration for such an ‘C28 application would be as follows:

```
var IpcMgr = xdc.useModule('ti.sdo.ipc.family.f28m35x.IpcMgr');
IpcMgr.readAddr  = 0x12000;     /* S6 RAM */
IpcMgr.writeAddr = 0x13000;     /* S7 RAM */
```

The corresponding configuration for the M3 application would be:

```
var IpcMgr = xdc.useModule('ti.sdo.ipc.family.f28m35x.IpcMgr');
IpcMgr.readAddr  = 0x20016000;  /* S7 RAM */
IpcMgr.writeAddr = 0x20014000;  /* S6 RAM */
IpcMgr.sharedMemoryOwnerMask = 0x80;
```

The readAddr and writeAddr segments specified for a processor must be different. The readAddr segment on one processor must correspond to the writeAddr segment on the other processor. The memory addresses you use must be the physical addresses understood by the local core.

By default, the M3 has write access to all segments initially. IPC’s IpcMgr module provides a sharedMemoryOwnerMask that the M3 core must set to provide write access to the ‘C28 core. This mask writes to the M3’s MSxMSEL register. This register determines which processor has write access to each of the 8 shared RAM segments. In the previous example, the M3 application sets the sharedMemoryOwnerMask to 0x80, which sets the bit for the S7 RAM segment to “1”, allowing the ‘C28 to write to that segment.

Additional configuration properties you can set for the IpcMgr module include:

- **sharedMemoryEnable.** This property lets the M3 processor disable one or more shared RAM segments. By default, all segments are enabled. This property writes to the MEMCNF register from the M3. To disable a shared RAM segment, set the corresponding bit to 0. You cannot load data into a disabled RAM segment. (Do not use in C28 applications.)

- **sharedMemoryAccess.** This property lets the M3 processor specify the type of access the owner can have to shared RAM segments. This property writes to the MSxSRCR register from the M3. (Do not use in C28 applications.)

  By default, the segment owner has fetch, DMA write, and CPU write access to all segments owned. You should *not* disable fetch or CPU write access for the two segments used by IpcMgr. DMA write access is not used by IpcMgr.

  The IpcMgr.sharedMemoryAccess configuration property is an array of eight 32-bit masks. Mask[0] corresponds to the S0 shared RAM segment, and so on. In each mask, bits 0 through 2 are used to control fetch (bit 0), DMA write (bit 1), and CPU write (bit 2) access. All other bits are ignored. By default, all three types of access are allowed, which corresponds to a bit setting of zero (0). Setting a bit to 1 disables that type of access for the shared RAM segment corresponding to that mask. For example, the following statements remove DMA write and CPU write access for the S4 segment:

```
var IpcMgr = xdc.useModule('ti.sdo.ipc.family.f28m35x.IpcMgr');
IpcMgr.sharedMemoryAccess[4] = 0x6;
```

- **IpcMgr.ipcSetFlag.** This property determines which flag generates an IPC interrupt. The default is 3. You can use a value from 0 to 3, but the value must be the same on both processors.

In addition, the IpcMgr module provides configuration properties that set the number of Notify and MessageQ messages stored in the circular buffers and the largest MessageQ size (in bytes) supported by the transport. These can be modified to reduce shared memory use if the application passes relatively few messages between the processors.

- **IpcMgr.numNotifyMsgs.** By default, the Notify driver's circular buffer can hold up to 32 messages, which means there can be up to 31 outstanding notifications. You can change this value by setting the IpcMgr.numNotifyMsgs property to some other power of 2.

  The IpcMgr.numNotifyMsgs property affects the size of the shared memory circular buffer used to store notifications regardless of the event IDs. Changing this value allows you to optimize either the memory use or the performance. That is, with fewer messages, the buffer is smaller but there is a higher chance that the system will have a full circular buffer and need to wait for space to be freed.

  Note that the IpcMgr.numNotifyMsgs property is different from the Notify.numEvents property. The Notify.numEvents property determines the number of unique event IDs that can be used in a system. When this property is set to the default value of 32, Notify event IDs can range from 0 to 31.

- **IpcMgr.numMessageQMsgs.** By default, the MessageQ transport's circular buffer can hold up to 4 messages. The number of MessageQ messages must be a power of 2. If your application does not use MessageQ, you should set this property to 0 in order to reduce the application's memory footprint.

- **IpcMgr.messageQSize.** By default, each message in the MessageQ transport's circular buffer can hold 128 bytes. If your application stores less information in each MessageQ message, you should set this property to reduce the application's memory footprint.

If you want to know how much memory is used by IpcMgr, open the .map file that is created when you build your application, and search for the section names that contain ti.sdo.ipc.family.f28m35x.IpcMgr.readSect and ti.sdo.ipc.family.f28m35x.IpcMgr.writeSect.

B.3 Examples for Concerto

IPC provides Notify and MessageQ examples for both the ARM M3 and the C28x cores. Both are dual-core examples in which the same program (with a slightly different configuration) is executed on both cores.

The Notify example uses the ti.ipc.Notify module to send a notification back and forth between the M3 and the C28 a number of times (10 by default). When a processor receives an event, it posts a Semaphore that allows a Task function to continue running and send a reply notification to the other processor.

The MessageQ example uses the ti.ipc.MessageQ module to send messages between processors. Each processor creates its own MessageQ first, and then tries to open the remote processor's MessageQ.

If you compare the CFG files for the Concerto examples with the examples for some other device, you will notice that the Concerto configuration is simpler because it does not need to synchronize the processors or configure the NotifySetup and SharedRegion modules.

If you compare the C code for the Concerto examples with the examples for some other device, you will find the following categories of differences:

- The Concerto MessageQ example allocates MessageQ messages using the ti.sysbios.heaps.HeapBuf module instead of the ti.sdo.ipc.HeapBufMP module, because HeapBufMP is not supported for Concerto.
- The Concerto examples do not call Ipc_start() or include the ti.sdo.ipc.Ipc module.
- In the Concerto examples, the other processor is called the “remote” processor instead of the “next” processor, since there are only two processors.

Index

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components which meet ISO/TS16949 requirements, mainly for automotive use. Components which have not been so designated are neither designed nor intended for automotive use; and TI will not be responsible for any failure of such components to meet such requirements.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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