SYS/BIOS Inter-Processor Communication (IPC) and I/O User's Guide

Literature Number: SPRUGO6A
March 2010

Texas Instruments

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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

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 1.20 on targets that use SYS/BIOS.

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.

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

March 29, 2010

Contents

Chapter 1

About IPC

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

Topic **Page**

1.1 What is IPC?

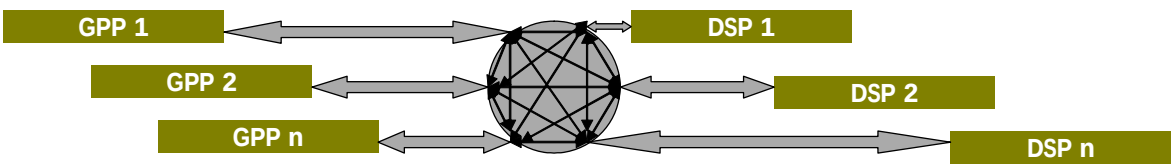
IPC is a component containing packages that are designed to allow communication between processors in a multi-processor environment and communication to peripherals. This communication includes message passing, streams, and linked lists. These work transparently in both uni-processor and multi-processor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

The ti.sdo.ipc package contains modules and interfaces for inter-processor communication. The ti.sdo.io package contains modules and interfaces to support peripheral communication. The ti.sdo.utils package contains utility modules for supporting the ti.sdo.ipc modules and other modules.

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 CCSv4 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)
- ❏ Microsoft Windows Vista
- ❏ Linux (Redhat 4 or 5)

If you are installing separately from CCSv4, 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 CCSv4 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. In general, IPC 1.20 requires the following software versions for other components:

- ❏ SYS/BIOS 6.21 or higher (installed as part of CCStudio 4)
- ❏ XDCtools 3.16 or higher (installed as part of CCStudio 4)

1.3 About this User Guide

- ❏ Chapter 2, "The Input/Output Package", describes the modules in the ti.sdo.io package.
- ❏ Chapter 3, "The Inter-Processor Communication Package“, describes the modules in the ti.sdo.ipc package.
- ❏ Chapter 4, “The Utilities Package“, describes the modules in the ti.sdo.utils package.

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

Note: Please see the release notes included in the package before starting to use the package. 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:

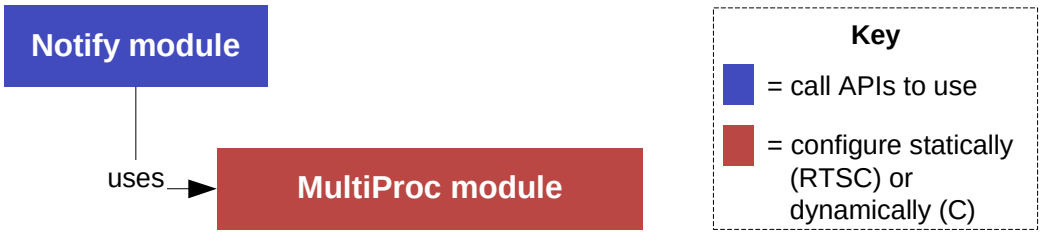
- **Minimal use of IPC.** This scenario performs inter-processor notification. The amount of data passed with a notification is minimal—typically on the order of 32 bits. This scenario is best used for simple synchronization between processors without the overhead and complexity of message-passing infrastructure. 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 sections that follow, diagrams show the 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 a RTSC 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 register callback functions with the Notify module to handle such events either statically or dynamically.

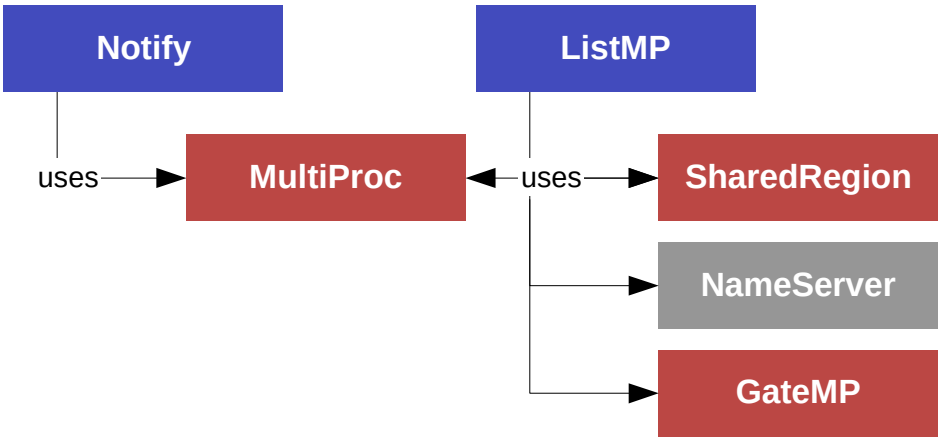
You must statically configure MultiProc module parameters, which are used by the Notify module.

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 3.7, *Notify Mod ule* and Section 4.3, *MultiProc Module*.

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

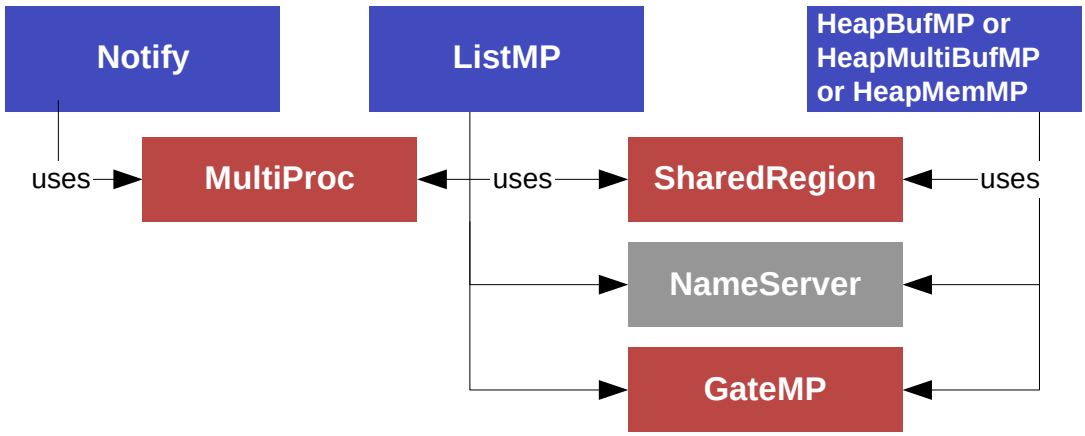
Internally, ListMP can optionally use the NameServer module to manage name/value pairs.

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 3.4, *ListMP Module*, Section 3.6, *GateMP Module*, Section 3.8, *SharedRegion Module*, and Section 4.4, *NameServer Module*.

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.



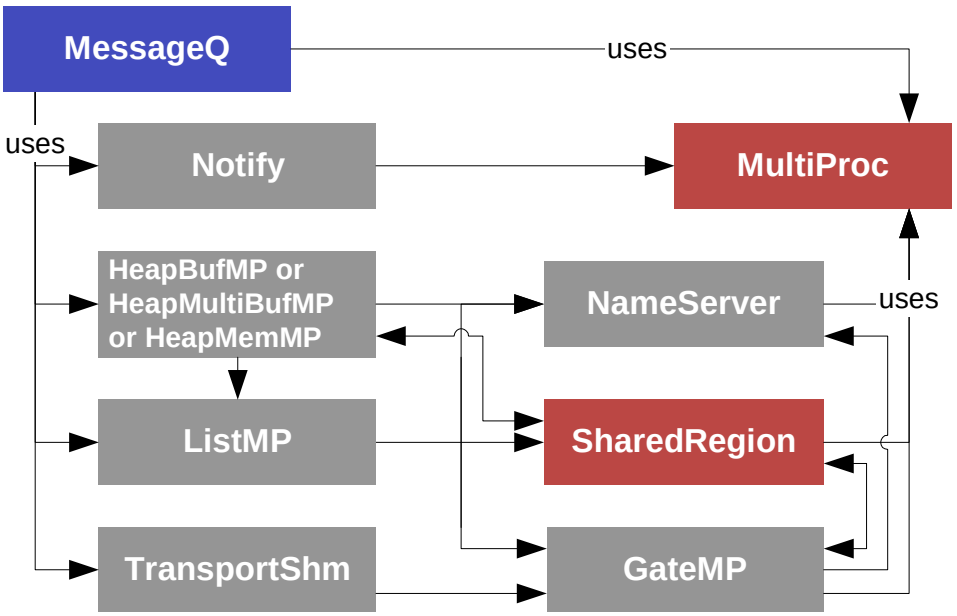
In this scenario, you make API calls to the Notify and ListMP modules and a Heap*MP module.

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 3.5, *Heap*MP Modules* and Section 3.6, *GateMP Module*.

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.

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 3.1.1). 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 CCSv4 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://wiki.davincidsp.com
- ❏ *XDCtools Getting Started Guide:* (XDC_INSTALL_DIR/docs/XDC_Getting_Started_Guide.pdf).
- ❏ *SYS/BIOS 6 Release Notes:* (BIOS_INSTALL_DIR/release_notes.html).
- ❏ *SYS/BIOS 6 Getting Started Guide:* (BIOS_INSTALL_DIR/docs/Bios_Getting_Started_Guide.pdf).

- *XDCtools and SYS/BIOS online documentation:* Open with CCSv4 online help.
- *TMS320 SYS/BIOS 6 User's Guide* (SPRUEX3)
- In CCSv4, templates for projects that use IPC are available when you create a CCS project with RTSC support enabled.

Chapter 2

The Input/Output Package

This chapter describes modules that can be used to handle input and output data.

Topic **Page**

2.1 Modules in IPC's ti.sdo.io Package

The ti.sdo.io package contains the following modules, which are described in this chapter:

- **DriverTable.** See Section 2.3, *Configuring a Driver and Acquiring a Driver Handle*.
- **DriverTypes.** See Section 2.5, *The IDriver Interface*.
- **IConverter.** See Section 2.6, *The IConverter Interface*.
- **IDriver.** See Section 2.5, *The IDriver Interface*.
- **Stream.** See Section 2.4, *Streams*.
- **Transformer.** See Section 2.6, *The IConverter Interface*.
- **Generator.** See Section 2.4.4, *Using Streams with Tasks*.
- **IomAdapter.** See Section 2.7, *The IomAdapter Module*.
- **DriverTemplate.** See Section 2.5.1, *Using the Driver Template Generator*.

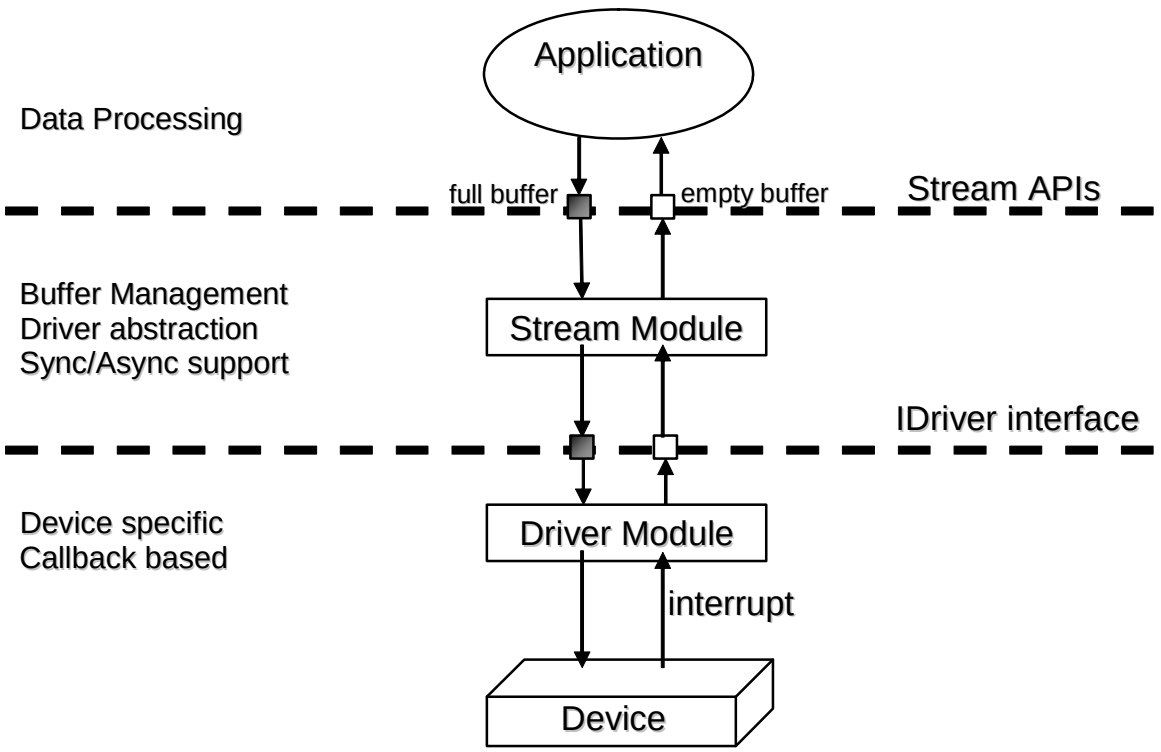
2.2 Overview of Streams

This chapter describes how you can use “streams” for data input and output. You use the Stream module to send data to an output channel or receive data from an input channel.

Streams are an abstraction of device drivers. They allow you to develop applications that do not directly interact with driver code. This allows the application to be developed in a driver-independent and platform-independent manner.

The Stream module also allows your application to process one buffer while the driver is working on another buffer. This improves performance and minimizes the copying of data. The typical use case for the Stream module is that an application gets a buffer of data from an input channel, processes the data and sends the data to an output channel.

The following figure shows a typical flow of data through an output channel.



In this use case, the application calls Stream module APIs. Those APIs manage buffers and the driver abstraction. Internally, the Stream module APIs calls the implementation of the IDriver interface for the driver.

2.3 Configuring a Driver and Acquiring a Driver Handle

Drivers are RTSC modules that inherit from ti.sdo.io.IDriver. A driver module manages peripherals. A driver modules manages all instances of a peripheral. For example, a hypothetical third-party Uart driver manages all UARTs in the system, and a Uart instance equates to a UART device.

An application configures a driver and gets a handle to a driver instance by following these steps:

1) **Enable modules.** The static configuration (cfg file) must include a xdc.useModule for each module it will use. For example, here are the xdc.useModule statements you need for the modules used in this section:

```
var Uart = xdc.useModule('thirdparty.drivers.Uart');
var DriverTable = xdc.useModule('ti.sdo.io.DriverTable');
var Stream = xdc.useModule('ti.sdo.io.Stream');
```

2) **Configure the driver module.** This step is optional because the driver may have defaults that suit the application. The driver is configured as part of the static configuration (cfg file). Typically, module-level parameters apply to all instances of the device driver. For example, DMA support and the number of channels allowed might be two such parameters. The driver may also allow the setting of default channel parameters as part of the static configuration. For example:

```
Uart.edmaSupport = true;
```

3) **Create a driver instance.** An application creates a driver instance to mark a peripheral for use. This step can be performed in the static configuration or at run-time. The driver will support resource management and prevent the peripheral from being used (created) again. Some drivers manage several peripherals; in such cases, the driver allows the application to specify a deviceId as part of its create parameters. A driver may support other create() parameters, which typically apply to all channels supported by an instance.

   Configuration example: This CFG example uses XDCtools to create a UART driver instance called “uart” with the default properties.

```
var uartParams = new Uart_Params;
uartParams.instance.name = "uart";
var uartHdl = Uart.create(uartParams);
```

Run-time example: This C example creates a UART driver instance called "uart" with the default properties.

```
Uart_Handle uartHdl;
Uart_Params uartParams;

Uart_Params_init(&uartParams);
uartParams.instance->name = "uart";
uartHdl = Uart_create(&uartParams, &eb);
```

4) **Add the device driver handle to the driver table.** Before the Stream module APIs can use a device driver instance, the instance needs to be added to a driver table managed by the DriverTable module. This table associates names with drivers. The name has to have a particular format—it has to begin with a "/". (See Section 2.6 for the reasons behind this convention.) This step can be performed in the static configuration or at run-time.

Configuration example: This statement associates the uartHdl you obtained in the previous step with the name "/uart0". Note: A slash "/" is required at the beginning of a driver name.

```
DriverTable.addMeta("/uart0", uartHdl);
```

Run-time example: This C statement associates the uartHdl you obtained in the previous step with the name "/uart0".

```
DriverTable_add("/uart0", uartHdl, &eb);
```

2.4 Streams

This section describes how to create and use streams as abstractions for drivers after you have associated a driver handle with a driver name as described in the previous section.

2.4.1 Creating and Deleting Streams

An application creates a Stream by specifying a device driver by name. The name is the same as the name you associated with a driver in the DriverTable.

One important point is that a Stream instance corresponds to a single channel (either input or output). Several Stream instances can be created using the same driver instance. If the driver instance supports two channels, then two Stream instances can be created using the same driver.

The Stream_create() API creates a Stream instance. It has the following parameters:

```
Stream_Handle Stream_create(
     String                    name,
     UInt                      mode,
     const Stream_Params       *prms,
     xdc_runtime_Error_Block   *eb)
```

The Stream_Params allows you to specify the following parameters:

```
struct Stream_Params {
    xdc_runtime_IInstance_Params   *instance;
    UInt                           maxIssues;
    xdc_runtime_IHeap_Handle       packetHeap;
    xdc_runtime_knl_ISync_Handle   sync;
    UArg                           chanParams;
};
```

The driver **name** and **mode** are required parameters for creating a Stream instance. The driver name must be present in the driver table.

The **mode** is either Stream_INPUT or Stream_OUPUT.

Stream_Params allow you to set optional parameters.

- The "instance" field allows you to specify a name for the Stream instance. The name will show up in the RTA and ROV run-time tools.
- The "maxIssues" field specifies the maximum numbers of buffers that can be issued to the stream using Stream_issue() before a buffer is reclaimed using Stream_reclaim(). This is 2 by default. If you will use Stream_read() and Stream_write() only, you can set this parameter to one.

- The "packetHeap" specifies the heap that the Stream instance can use to allocate IO packets that are used internally to submit IO jobs to the driver. See Section 2.5.6 for details on IO Packets.
- The "sync" field selects a synchronization mechanism for that Stream instance. When this is left at its default of NULL, Stream creates a Semaphore instance for synchronization. See Section 2.4.3 for more on synchronization mechanisms.
- The "chanParams" configures the driver channel and is sent along to the driver's open() call. The chanParams are driver-specific. Typically drivers allow default channel params to be statically configured. To override defaults, the client can pass chanParams down to the driver by setting the chanParams field in Stream.Params.

Stream_create() also takes an **Error_Block** as an argument and can fail. Stream_create() returns a Stream_Handle that can then be used to send or receive data.

See the online documentation for more about Stream_create().

Important Note: A Stream_Handle cannot be used by more than one task simultaneously. One task must finish a Stream_read() or Stream_write() or Stream_issue()/Stream_reclaim() before another Task can use the same Stream_Handle. It is safer if a Stream_Handle is used only by a single task.

Stream_delete() deletes a stream and frees all resources allocated during Stream_create(). This frees up the driver channel in use.

2.4.2 Streams and Error Blocks

A number of functions in the ti.sdo.io package use an Error_Block as an argument. An Error_Block is a structure defined by the xdc.runtime.Error module. You can create and use an Error_Block as follows:

```
#include <xdc/runtime/Error.h>

Error_Block eb;                /* usually global */
...
Error_init(&eb);               /* usually in main() */
...
Stream_issue(handleIn, pbuf, BUFSIZE, NULL, &eb);
```

Your application can then test the value in "eb" to see if what error was returned.

When you are first developing an application, you may want to simply pass NULL instead of &eb for such Error_Block arguments. If an error occurs, the NULL will cause the program to fail with a System_abort(), making it relatively easy to debug.

2.4.3 Stream and the Synchronization Mechanisms

The Stream module uses xdc.runtime.knl.Sync for synchronization. The xdc.runtime.knl.Sync module provides two main functions Sync_signal() and Sync_wait().

These functions take an ISync_Handle as the first parameter. An ISync_Handle must first be obtained by a module that implements the ISync interface. Various implementations of these interfaces handle synchronization of Streams in different ways.

The xdc.runtime.knl package provides the following implementations of the ISync interface:

- **SyncSemThread** (the default) is a BLOCKING implementation that is built on top of the xdc.runtime.knl.SemThread module.
- **SyncGeneric** can be used with any two functions that provide the ISync_signal() and ISync_wait() functionality. See Section 2.4.7.
- **SyncNull** performs no synchronization.

SYS/BIOS contains several implementations of the ISync interface in the ti.sysbios.syncs package:

- **SyncSem** is based on Semaphores and is BLOCKING. See Section 2.4.4.
- **SyncSwi** is based on Swis and is NON-BLOCKING. See Section 2.4.5.
- **SyncEvent** is based on Events and is NON-BLOCKING. See Section 2.4.6.

Instead of tying itself to a particular module for synchronization, the Stream module allows you to pick an ISync implementation module. You can select an ISync implementation for a stream instance using the instance-level configuration parameter "sync".

If you pass NULL for the "sync" parameter to Stream_create(), the Stream module creates a SyncSemThread instance for the Stream instance. This translates to a Semaphore instance for this Stream instance.

The Stream module calls ISync_signal() in the callback path when IO completes. It calls ISync_wait() from Stream_reclaim() when IO has not been completed.

2.4.4 Using Streams with Tasks

Stream_issue() and Stream_reclaim() can be used with Task threads. This is the most common use case. The application first creates a Stream instance. By default Stream uses SyncSemThread.

The application calls Stream_issue() with a buffer and size. It is possible to pass in a semaphore instance owned by the application during Stream_create(). See Section 2.4.4.1 for details.

Stream_issue() sends a buffer to a stream. No buffer is returned, and the application no longer owns the buffer. Stream_issue() returns control to the task without blocking. The buffer has been given to the driver for processing. Internally, Stream uses a DriverTypes_Packet to submit a job to the driver.

At a later time when the application is ready to get the buffer back, Stream_reclaim() is called. This call to Stream_reclaim() blocks if the driver has not completed processing the IO packet.

When the driver has finished processing the buffer, an ISR occurs and the driver performs a callback to Stream. Stream receives the processed IO packet in the callback context, queues it up and calls ISync_signal(). This call is routed to SyncSemThread_signal(), which posts the Semaphore. Control returns to the application task and the application can work on the returned buffer.

If a timeout occurs, the application needs to call Stream_abort() to force the driver to return buffers currently being processed and queued for processing.

Is it possible for Stream_reclaim() to raise an error. This error is reported by the driver in the IO packet. Stream_reclaim() also raises an error if no buffers have been issued.

The Stream_issue/Stream_reclaim APIs provide flexibility by allowing the application to control the number of outstanding buffers at runtime. A client can send multiple buffers to a stream without blocking by using Stream_issue(). The buffers are returned, at the client's request, by calling Stream_reclaim(). This allows the client to choose how deep to buffer a device and when to block and wait for a buffer.

The Stream_issue/Stream_reclaim APIs guarantee that the client's buffers are returned in the order in which they were issued. This allows a client to use memory from any source for streaming. For example, if a SYS/BIOS task receives a large buffer, that task can pass the buffer to the stream in small pieces simply by advancing a pointer through the larger buffer and calling Stream_issue() for each piece. This works because each piece of the buffer is guaranteed to come back in the same order it was sent.

The Stream_issue() API has the following parameters:

```
Void Stream_issue(
    Stream_Handle   handle,
    Ptr             buf,
    SizeT           size,
    UArg            arg,
    Error_Block     *eb);
```

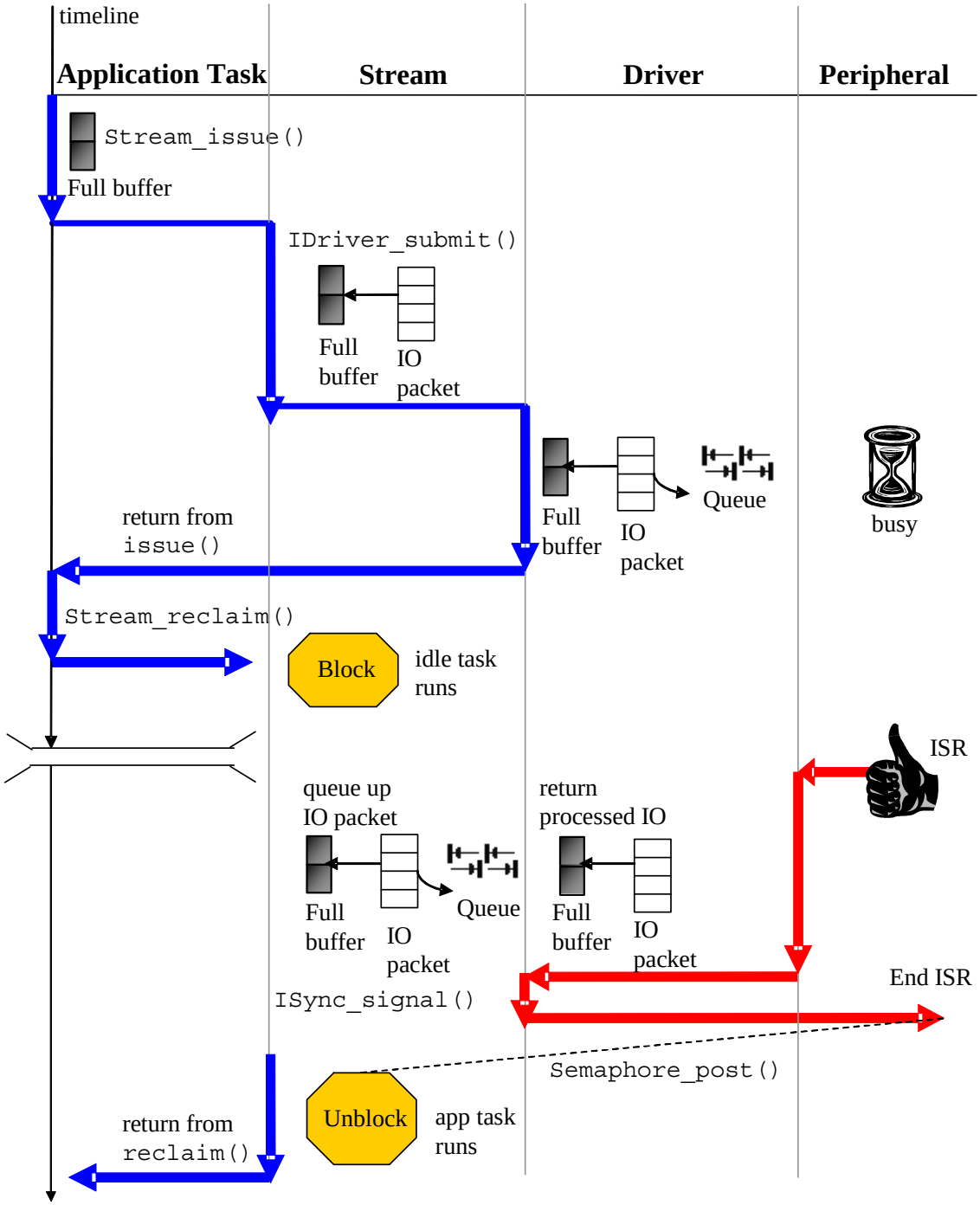
The Stream_reclaim() API has the following parameters:

```
SizeT Stream_reclaim(
    Stream_Handle   handle,
    Ptr             *pbufp,
    UInt            timeout,
    UArg            *parg,
    Error_Block     *eb);
```

See the online documentation for more about these APIs.

The "arg" parameter is not interpreted by SYS/BIOS, but is offered as a service to the application. The arg parameter can be used by the application to associate some data with the buffer—for example, the format of data in the buffer. This arg is passed down to the driver and returned to the application through Stream_reclaim(). The application can use the arg to communicate to the driver. For example, arg could be used to send a timestamp to an output device, indicating exactly when the data is to be rendered. The driver treats the arg as a read-only field.

The following figure shows the control flow for Streams used with Tasks.



The following example uses Stream_issue() and Stream_reclaim() in a Task. "Generator" is a driver provided with SYS/BIOS. All instances are dynamically created in this example.

Configuration code: The following configuration file excerpt includes an xdc.useModule statement for each IO-related module used in the application:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Stream = xdc.useModule('ti.sdo.io.Stream');
var DriverTable = xdc.useModule('ti.sdo.io.DriverTable');
var Generator = xdc.useModule('ti.sdo.io.drivers.Generator');

DriverTable.maxRuntimeEntries = 1;
```

Run-time code: This C code uses Stream_issue() and Stream_reclaim() in a Task:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <xdc/cfg/global.h>

#include <ti/sdo/io/Stream.h>
#include <ti/sdo/io/DriverTable.h>
#include <ti/sdo/io/drivers/Generator.h>

#define BUFSIZE 16
#define NBUFS   5

Void genSine(Ptr bufp, size_t nmaus, UArg arg);
Void myPrintf(Ptr bufp, size_t nmaus, UArg arg);

Int buf[BUFSIZE];
Ptr pbuf = &buf;
Int count = 1;

Stream_Handle handleIn, handleOut;

Generator_ChanParams genParamsIn = {genSine, NULL, FALSE};
Generator_ChanParams genParamsOut = {myPrintf, NULL, FALSE};

Void tsk0Fxn(UArg arg1, UArg arg2);
```

```
/*
 * ======== genSine ========
 * Function to simulate input channel. Generates a sine wave.
 */
Void genSine(Ptr bufp, size_t nmaus, UArg arg)
{
    Int i;
    Int *dst = (Int *)bufp;
    static Int data[16] = {
        0, 11793, 22594, 29956, 32767, 29956, 22594, 11793,
       0, -11793, -22594, -29956, -32767, -29956, -22594, -11793
    };

    for (i = 0; i < nmaus / sizeof(Int); i++) {
        dst[i] = data[i & 0xf];
    }
}

/*
 * ======== myPrintf ========
 * Function to simulate output channel. Prints the buffer
 */
Void myPrintf(Ptr bufp, size_t nmaus, UArg arg)
{
    Int i;
    Int *src = (Int *)bufp;

    for (i = 0; i < (nmaus / sizeof(Int)); i++) {
        System_printf("%d\n", src[i]);
    }
}

/*
 * ======== main ========
 */
Int main(Int argc, Char* argv[])
{
    Stream_Params streamParams;
    Generator_Handle driverHdl;

    /* Create a Generator instance */
    driverHdl = Generator_create(NULL, NULL);

    /* Add Generator instance to DriverTable */
    DriverTable_add("/genDevice",
       Generator_Handle_upCast(driverHdl), NULL);
```

```
    /* Create input stream */
    Stream_Params_init(&streamParams);
    streamParams.chanParams = (UArg)&genParamsIn;
    handleIn = Stream_create("/genDevice", Stream_INPUT,
       &streamParams, NULL);

    /* Create output stream */
    streamParams.chanParams = (UArg)&genParamsOut;
    handleOut = Stream_create("/genDevice", Stream_OUTPUT,
       &streamParams, NULL);

    Task_create(tsk0Fxn, NULL, NULL);

    BIOS_start();

    return(0);
}

/*
 * ======== tsk0Fxn ========
 * Task that owns input channel and output channel stream.
 */
Void tsk0Fxn (UArg arg1, UArg arg2)
{
    for (;;) {
        /* buf gets filled with sine data */
        Stream_issue(handleIn, pbuf, BUFSIZE, NULL, NULL);
        Stream_reclaim(handleIn, &pbuf, BIOS_WAIT_FOREVER,
            NULL, NULL);

        System_printf("Printing sine data %dth time\n", count);

        /* buf filled with sine data is printed out */
        Stream_issue(handleOut, pbuf, BUFSIZE, NULL, NULL);
        Stream_reclaim(handleOut, &pbuf, BIOS_WAIT_FOREVER,
            NULL, NULL);

        count++;

        if (count > NBUFS) {
            System_exit(0);
        }
    }
}
```

2.4.4.1 Using a Semaphore Instance Created by the Application with a Stream Instance

There may be cases where the application does not want Stream to create a Semaphore for synchronization, and instead wants to supply its own Semaphore to Stream. The following code snippet shows how to do this.

```
#include <ti/sysbios/syncs/SyncSem.h>
#include <ti/sysbios/ipc/Semaphore.h>
#include <ti/sdo/io/Stream.h>

/*
* ======== main ========
*/
Int main(Int argc, Char* argv[])
{
   Stream_Params streamParams;
   SyncSem_Params syncParams;
   SyncSem_Handle syncSem;

   /* Create input stream */
   SyncSem_Params_init(&syncParams);
   syncParams.sem = Semaphore_create(0, NULL, NULL);
   syncSem = SyncSem_create(&syncParams, NULL);

   Stream_Params_init(&streamParams);
   streamParams.chanParams = (UArg)&genParamsIn;
   streamParams.sync = SyncSem_Handle_upCast(syncSem);
   handleIn = Stream_create("/genDevice", Stream_INPUT,
       &streamParams, NULL);
}
```

2.4.5 Using Stream with Swis

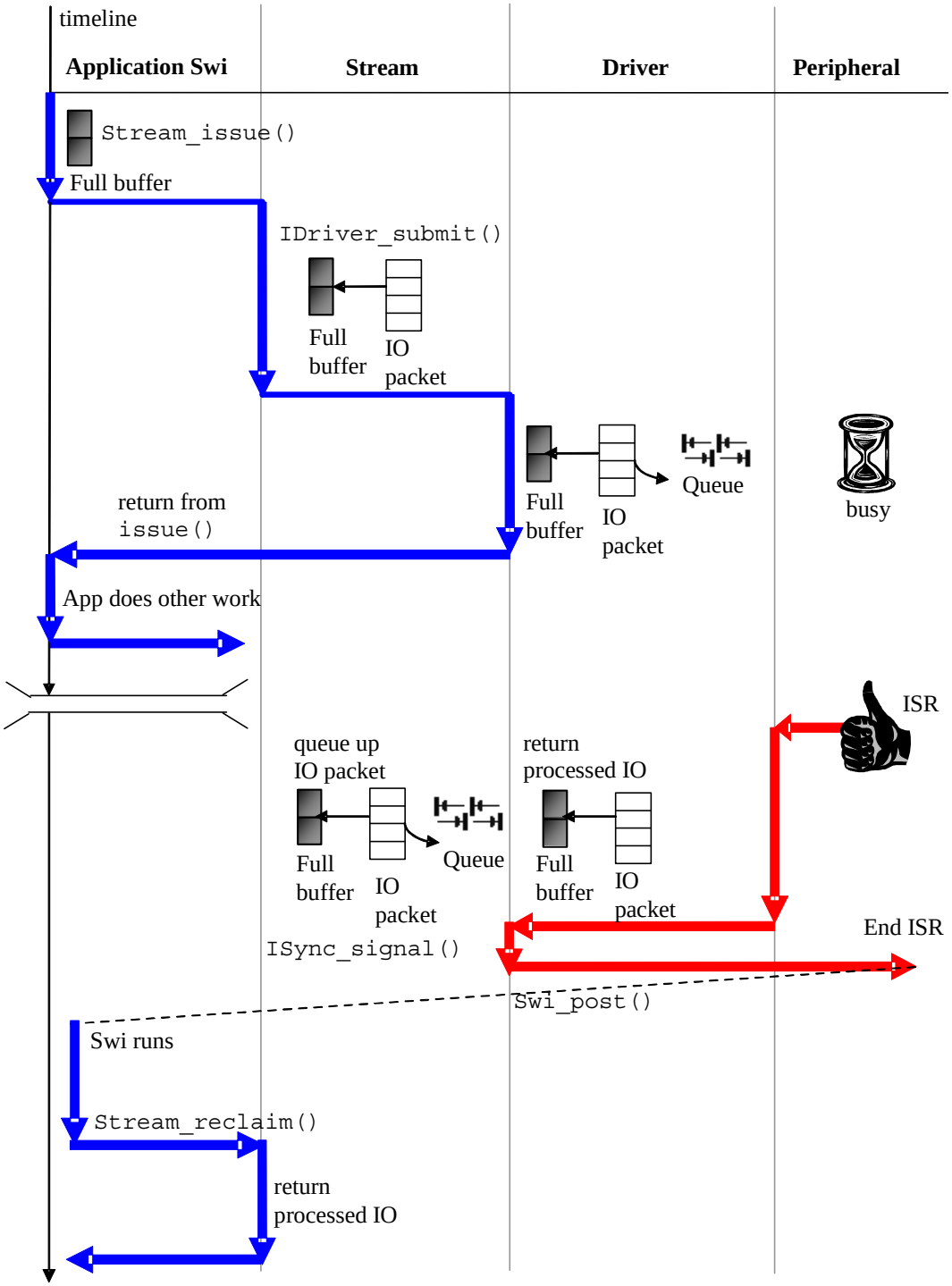
Stream_issue() and Stream_reclaim() can also be used with Swi threads. The client first creates a Stream Instance. The application has to populate the "sync" field in the Stream_Params struct. This sync field needs to be assigned to a SyncSwi instance.

The application needs to follow these steps to get a SyncSwi_Handle.

1) Create a Swi instance.
2) Populate the "swi" field of a SyncSwi_Params struct with the Swi_Handle received in the previous step.
3) Create a SyncSwi instance using the SyncSwi_Params struct.
4) Populate the "sync" field in Stream_Params struct with the SyncSwi_Handle received from the previous step.

The Stream module calls ISync_signal() in the callback path when IO completes. This results in a Swi_post(). The swi runs and calls Stream_reclaim() to get back the processed buffer.

SyncSwi_wait() does nothing and returns FALSE for timeout. The data flow is shown in the following figure.


timeline
Application Swi
Stream
Driver
Peripheral
Stream_issue()
Full buffer
IDriver_submit()
Full buffer
IO packet
Full buffer
IO packet
Queue
busy
return from issue()
App does other work
ISR
queue up IO packet
Full buffer
IO packet
Queue
return processed IO
Full buffer
IO packet
End ISR
ISync_signal()
Swi_post()
Swi runs
Stream_reclaim()
return processed IO

The following example uses Stream_issue() and Stream_reclaim() in a Swi. "Generator" is a driver provided with SYS/BIOS. All instances are statically created in this example. This example uses the genSine and myPrintf functions from the previous section; they are not repeated in this example.

Configuration code: The following configuration file excerpt statically creates all the instances used in the application:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Swi = xdc.useModule('ti.sysbios.knl.Swi');

var Stream = xdc.useModule('ti.sdo.io.Stream');
var Generator = xdc.useModule('ti.sdo.io.drivers.Generator');
var SyncSwi = xdc.useModule('ti.sysbios.syncs.SyncSwi');
var DriverTable = xdc.useModule('ti.sdo.io.DriverTable');

/* create a Swi instance */
var swi0 = Swi.create("&swi0Fxn");

/* create a generator instance and add to DriverTable */
var driverHdl = Generator.create();
DriverTable.addMeta("/genDevice", driverHdl);

/* Prep for SyncSwi instance */
var syncPrms = new SyncSwi.Params();
syncPrms.swi = swi0;

/* input channel creation */
var genParamsIn = new Generator.ChanParams();
genParamsIn.userFxn = "&genSine";
genParamsIn.userArg = null;
genParamsIn.async = true;

var streamPrms = new Stream.Params();
streamPrms.chanParams = genParamsIn;
streamPrms.sync = SyncSwi.create(syncPrms);
Program.global.handleIn = Stream.create("/genDevice",
Stream.INPUT, streamPrms);
```

```
/* output channel creation */
var genParamsOut = new Generator.ChanParams();
genParamsOut.userFxn = "&myPrintf";
genParamsOut.userArg = null;
genParamsOut.async = true;

var streamPrms = new Stream.Params();
streamPrms.chanParams = genParamsOut;
streamPrms.sync = SyncSwi.create(syncPrms);
Program.global.handleOut = Stream.create("/genDevice",
Stream.OUTPUT, streamPrms);
```

Run-time code: This C code uses Stream_issue() and Stream_reclaim() in a Swi:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Swi.h>
#include <ti/sdo/io/Stream.h>

#define BUFSIZE 128
Int buf[BUFSIZE];
Ptr pbuf = &buf;

extern Stream_Handle handleIn, handleOut;
Int count = 0;

/*
 * ======== main ========
 */
Int main(Int argc, Char* argv[])
{
    Stream_issue(handleIn, &buf, BUFSIZE, NULL, NULL);

    BIOS_start();

    return(0);
}
```

```
/*
 * ======== swi0Fxn ========
 */
Void swi0Fxn (UArg arg1, UArg arg2)
{
    Stream_reclaim(handleIn, &pbuf, BIOS_WAIT_FOREVER,
       NULL, NULL);

    /* work on buffer here */

    Stream_issue(handleIn, pbuf, BUFSIZE, NULL, NULL);

    count++;
    if (count >= 5) {
        System_exit(0);
    }
}
```

2.4.6 Using Streams with Events

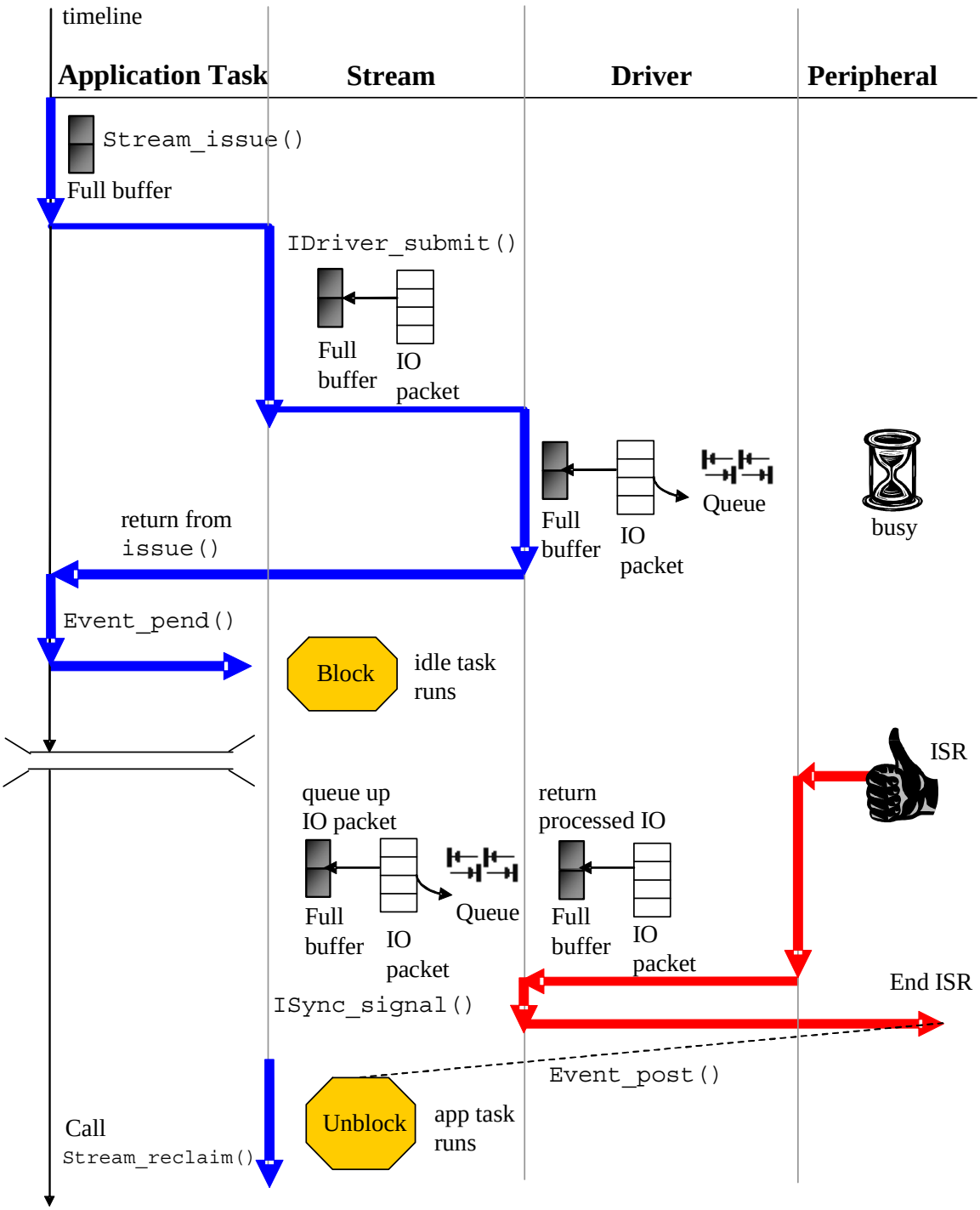
There are cases where a Task needs to wait on multiple events. For example, a Task may need to wait for a buffer from an input channel or a message from the host. In such cases, the Task should use the Event_pend() API to "wait on multiple" items. See the *SYS/BIOS User's Guide* (SPRUEX3) for more information on Events.

In order to use Streams with the Event module, the Stream needs to be created using the following steps:

1) Create an Event instance.
2) Populate the "event" field of the SyncEvent_Params struct with the Event_Handle received from the previous step.
3) Create a SyncEvent instance using the SyncEvent_Params struct.
4) Populate the "sync" field in the Stream_Params struct with the SyncEvent_Handle received from the previous step.

For the example later in this section, the application first primes the channel by calling Stream_issue(). When the worker task runs, it calls Event_pend() and BLOCKS waiting for IO completion.

When IO is complete, the Stream module calls ISync_signal(), which results in an Event_post() call. The Task wakes up, checks which event happened, and calls Stream_reclaim() to get the processed buffer.


timeline
Application Task
Stream
Driver
Peripheral
Stream_issue()
Full buffer
IDriver_submit()
Full buffer
IO packet
Full buffer
IO packet
Queue
busy
return from issue()
Event_pend()
Block
idle task runs
ISR
queue up IO packet
Full buffer
IO packet
Queue
return processed IO
Full buffer
IO packet
End ISR
ISync_signal()
Event_post()
Call Stream_reclaim()
Unblock
app task runs

The following example uses Stream_issue() and Stream_reclaim() with an Event. "Generator" is a driver provided with SYS/BIOS. The Event and the Generator driver are statically created in this example. This example uses the genSine and myPrintf functions from the previous section; they are not repeated in this example.

Configuration code: The following configuration file excerpt statically creates all the Event and Generator instances used in the application:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Stream = xdc.useModule('ti.sdo.io.Stream');
var Event = xdc.useModule('ti.sysbios.ipc.Event');
var SyncEvent = xdc.useModule('ti.sysbios.syncs.SyncEvent');
var Generator = xdc.useModule('ti.sdo.io.drivers.Generator');
var DriverTable = xdc.useModule('ti.sdo.io.DriverTable');

Program.global.evt0 = Event.create();

/* create a Generator instance and add to DriverTable */
var driverHdl = Generator.create();
DriverTable.addMeta("/genDevice", driverHdl);
```

Run-time code: This C code uses Stream_issue() and Stream_reclaim() in a Task using an Event:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/ipc/Event.h>
#include <ti/sysbios/sync/SyncEvent.h>
#include <ti/sdo/io/Stream.h>
#include <ti/sdo/io/drivers/Generator.h>

#define BUFSIZE 128
Int buf1[BUFSIZE];
Int buf2[BUFSIZE];
Ptr pbuf = &buf1;

Stream_Handle handleIn, handleOut;
extern Event_Handle evt0;
```

```
Void genSine(Ptr bufp, size_t nmaus, UArg arg);
Void myPrintf(Ptr bufp, size_t nmaus, UArg arg);
Void tsk0Fxn(UArg arg1, UArg arg2);

Generator_ChanParams genParamsIn = {genSine, NULL, FALSE};
Generator_ChanParams genParamsOut = {myPrintf, NULL, FALSE};

/* ======== main ======== */
Int main(Int argc, Char* argv[])
{
    SyncEvent_Params syncEvtParams;
    Stream_Params streamParams;

    SyncEvent_Params_init(&syncEvtParams);
    syncEvtParams.event = evt0;

    Stream_Params_init(&streamParams);

    streamParams.chanParams = (UArg)&genParamsIn;
    streamParams.sync =
      SyncEvent_Handle_upCast(SyncEvent_create(&syncEvtParams,
      NULL));
    handleIn = Stream_create("/genDevice", Stream_INPUT,
        &streamParams, NULL);

    streamParams.chanParams = (UArg)&genParamsOut;
    streamParams.sync =
      SyncEvent_Handle_upCast(SyncEvent_create(&syncEvtParams,
      NULL));
    handleOut = Stream_create("/genDevice", Stream_OUTPUT,
        &streamParams, NULL);

    Task_create(tsk0Fxn, NULL, NULL);

    Stream_issue(handleIn, &buf1, BUFSIZE, NULL, NULL);
    Stream_issue(handleOut, &buf2, BUFSIZE, NULL, NULL);

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

```
/* ======== tsk0Fxn ======== */
Void tsk0Fxn (UArg arg1, UArg arg2)
{
    Bits32 mask;

    mask = Event_pend(evt0, Event_Id_00 + Event_Id_01,
        Event_Id_NONE, BIOS_WAIT_FOREVER);

   System_printf("Awake after Event_pend, mask = %d \n", mask);

    Stream_reclaim(handleIn, &pbuf, BIOS_NO_WAIT, NULL, NULL);
   Stream_reclaim(handleOut, &pbuf, BIOS_NO_WAIT, NULL, NULL);

    System_exit(0);
}
```

2.4.7 Using Streams with Generic Callbacks

It is possible for the application to provide two functions equivalent to ISync_signal() and ISync_wait() using the xdc.runtime.knl.SyncGeneric module.

One use case for this is when the application will use Stream with a simple callback. The application then provides its callback function as the signal function to SyncGeneric_create(). The Stream module then invokes the application callback when IO completes. The application can reclaim the buffer within the callback function.

2.4.8 Using Stream_read() and Stream_write()

Stream_read() and Stream_write() are simply wrappers on top of the Stream_issue() and Stream_reclaim() APIs. While Stream_issue() and Stream_reclaim() are used for both input and output streams, Stream_read() is used only for input streams, and Stream_write() is used only for output streams.

- Stream_read() is equivalent to a Stream_issue()/Stream_reclaim() pair for an input channel.
- Stream_write() is equivalent to an Stream_issue()/Stream_reclaim() pair for an output channel.

The Stream_read() API has the following parameters:

```
SizeT Stream_read(
    Stream_Handle   handle,
    Ptr             bufp,
    SizeT           size,
    UInt            timeout,
    Error_Block     *eb);
```

The application calls Stream_read() with an empty buffer and a size. The size of the buffer should be greater than or equal to the size requested. Upon returning from Stream_read(), the application has a full buffer. Stream_read() blocks until the buffer is filled or a timeout occurs.

Note: When a Stream is associated with an ISync_Handle that is non-blocking, Stream_read() raises an error.

The Stream_write() API has the following parameters:

```
SizeT Stream_write(
    Stream_Handle   handle,
    Ptr             bufp,
    SizeT           size,
    UInt            timeout,
    Error_Block     *eb);
```

The application calls Stream_write() with a full buffer and a size. The size of the buffer should be greater than or equal to the size given. Upon returning from Stream_write(), the application has a processed buffer. Stream_write() blocks until a buffer is processed or a timeout occurs

Note: When a stream is associated with an ISync_Handle that is non-blocking, Stream_read() raises an error.

2.4.9 Stream_abort() and Error Handling

Stream_abort() is only required if you are using Stream_issue() and Stream_reclaim(). When several buffers have been issued to a Stream instance, the application may want to cancel processing and get back all the buffers. This could be the result of one of the following causes:

- ❏ The application has decided to cancel current IO and work on something else.
- ❏ Stream_reclaim() returned an error.

In such cases, the application needs to call Stream_abort() to force the driver to return all buffers without processing them. After a call to Stream_abort(), the application still needs to call Stream_reclaim() to get back the buffers.

Stream_reclaim() returns the buffers in the order that they were issued. The size field returned by Stream_reclaim() is the size processed by the driver.

2.4.10 Constraints When Using Streams

- A Stream instance can be used only by a single Task.
- Stream_issue() and Stream_reclaim() can only be called by a single thread (Task or Swi) or in the callback context.
- Stream_issue(), Stream_reclaim(), Stream_read(), and Stream_write() cannot be called from Module startup. Some drivers may require that these APIs not be called even from main() if they require hardware interrupts to enable their peripherals.

2.5 The IDriver Interface

The IDriver interface defines the driver interface for SYS/BIOS 6. If you are a driver writer, read the following subsections before implementing a SYS/BIOS 6 driver. The IO modules in SYS/BIOS 6 will only talk to drivers that implement this interface.

The IDriver interface defines four main functions that driver writers need to implement:

- **IDriver_open()**. See Section 2.5.4.
- **IDriver_close()**. See Section 2.5.5.
- **IDriver_submit()**. See Section 2.5.6.
- **IDriver_control()**. See Section 2.5.7.

Drivers should also implement a create() function and a delete() function, but the calling syntax for these functions is not specified in the IDriver interface because the end-user application is responsible for calling these functions. See Section 2.5.2 and Section 2.5.3.

The DriverTypes module is a supporting module that defines types used by all drivers.

2.5.1 Using the Driver Template Generator

The ti.sdo.io.driverTemplate command line tool helps driver writers by generating some starter files. You can invoke the driverTemplate tool as follows:

```
xs ti.sdo.io.driverTemplate -m <DrvMod> -o <outputPath>
```

This tool generates *DrvMod*.xdc, *DrvMod*.xs and *DrvMod*.c files within a package at <*outputPath*>. The template also generates a simple test for the new driver. Driver writers can edit these files to meet their driver requirements.

The generated package can be built using the xdc tools without any changes.

2.5.2 Driver create() Function

For a driver, the syntax of the create() call is not specified in the IDriver interface. The driver owns its create() call and declares it in its own xdc file. Applications need to call the driver-specific create() function directly. The Stream module never calls a driver's create() function directly.

Typically a driver instance equates to a peripheral (for example, uart0). A driver may support several instances (for example, 3 uarts). Typically a driver allows only a certain number of instances to be created.

A call to the driver's create() function does the following:

1) Mark the device inUse.
2) Initialize the instance object
3) Initialize peripheral registers
4) Register interrupts

Typically a driver instance object contains the a deviceId to map the instance to a particular device and channel objects. For example:

```
struct Instance_State {
    UInt    deviceId;
    ChanObj chans[NUMCHANS];
};
```

2.5.3 Driver delete() Function

A driver's delete() function is directly called by the application. The syntax is not specified in the IDriver interface. It should free all resources allocated and acquired by the driver's create() function.

2.5.4 IDriver_open() Function

The driver's open() function is called by Stream_create().

For statically created Stream instances, the driver's open() function is called for all static Stream instances during Stream module startup. Because of this, it is possible that the driver's open() function will be called even before the driver's module startup function. It is therefore necessary that the driver handle this case and return a channel pointer even though the peripheral may not be up and running. Eventually, when the driver's startup gets called, it should raise an error if the peripheral could not be initialized correctly.

The driver's open() function is called using the following parameters:

```
Ptr IDriver_open(
    IDriver_Handle          handle,
    String                  name,
    UInt                    mode,
    UArg                    chanParams,
    DriverTypes.DoneFxn     cbFxn,
    UArg                    cbArg,
    Error.Block             *eb);
```

The driver's open() function can fail (for example, if the channel is in use). When successful, The driver's open() function returns an opaque channel handle, usually a pointer to the channel object.

The driver handle passed to the driver's open() function is read from the driver table by Stream. It is the handle returned by calling the driver specific create function.

The "name" parameter allows for driver-specific configuration—for example, when a channel ID is required. The name will be NULL for most drivers. For example, some drivers support many input and output channels per driver instance. To specify a particular channel, the application would call Stream_create() with a driver name like "rtdx5". The Stream module would then get the driver handle for the rtdx entry in the driver table, and call the driver's open() function with name="5". The driver could parse this name to get the channel ID.

Note: When the name passed to stream has no characters meant for the driver, Stream will set the name to NULL. The driver could then check name as follows: if (name == NULL)...

Most drivers will simply ignore the name.

The "mode" is either DriverTypes.INPUT or DriverTypes.OUTPUT.

The "chanParams" structure is driver specific. When chanParams is NULL, the driver should use the default parameters that were statically configured.

The "cbFxn" function and "cbArg" are used to specify a callback function and argument to indicate the completion of IO after an asynchronous submit call.

The driver should raise an error if open() fails, and the error block should contain a driver-specific error or a generic error defined by DriverTypes.

The driver's open() function should return a driver-specific opaque channel handle.

Typically, a driver performs some or all of the following tasks in its open() function:

- ❏ Mark that channel is in use.
- ❏ Initialize the channel object structure.
- ❏ Initialize the peripheral using chanParams.
- ❏ Enable interrupts for the channel.

The structure of the channel object is defined differently by each driver. The following example shows a sample channel object structure:

```
struct ChanObj {
    Bool                inUse;         /* is channel is use? */
    UInt                mode;           /* input or output */
    List.Handle         pendList;      /* queue io packets */
    DriverTypes.DoneFxn callbackFxn;   /* callback fxn */
    UArg                callbackArg;   /* callback arg *
};
```

While the structure may vary, some fields are necessary in nearly all drivers. These important fields are:

- ❏ **inUse.** Allows the driver to allow only one open call per channel.
- ❏ **mode.** Must have a value of DriverTypes_INPUT, DriverTypes_OUTPUT, or DriverTypes_INOUT.
- ❏ **pendList.** A mini-driver must be able to handle or hold multiple I/O requests due to multiple calls to the driver's submit() function by Stream.
- ❏ **callbackFxn.** The callback function pointer stores which function is called as a result of a completed I/O request. This is typically the callback function implemented as part of Stream.
- ❏ **callbackArg.** The callback argument is a pointer that is an input parameter to the callback function.

2.5.5 IDriver_close() Function

The driver's close() function is called by Stream_delete(). The driver typically needs to perform some or all of the following actions in its close() function:

- ❏ Check to see if the channel can be closed.
- ❏ Release the channel.
- ❏ Disable interrupts for the channel.

The driver's close() function is called using the following syntax:

```
Void IDriver_close(
    IDriver_Handle          handle,
    Ptr                     chanHandle,
    Error.Block             *eb);
```

2.5.6 IDriver_submit() Function

Call to Stream_issue(), Stream_read() and Stream_write() result in a call to the driver's submit() function, which is used to submit jobs to the driver.

The driver's submit() function is called using the following syntax:

```
UInt IDriver_submit(
    IDriver_Handle          handle,
    Ptr                     chanHandle,
    DriverTypes.Packet      *packet,
    Error.Block             *eb);
```

The DriverTypes_Packet has the following fields:

```
typedef struct DriverTypes_Packet {
    List_Elem               link;
    Ptr                     addr;
    SizeT                   origSize;
    SizeT                   size;
    UArg                    arg;
    DriverTypes_PacketCmd   cmd;
    Error_Id                error;
    UArg                    misc;
    Int                     status;
} DriverTypes_Packet;
```

- **link.** This field can be used by driver to queue up IO packets.
- **addr.** This field points to a buffer of data. The driver preserves this field.
- **origSize.** This is the size of data buffer. The driver preserves this field.
- **size.** This is actual size of data written or read. The driver updates this field.
- **arg.** This is used by the end application. The driver preserves this field.
- **cmd.** This is the Packet command. The driver preserves this field.
- **error.** This is filled in by the mini-driver and contains status of IO.

It may be possible for the driver to complete the IO without the use of an asynchronous interrupt. For example, if there is enough room in the peripheral's buffer, and/or depending on the polling mode used. In such cases the driver's submit() function should return the DriverTypes.COMPLETED status.

DriverTypes.ERROR should be returned by the driver's submit() function if there is an error.

When the driver requires an asynchronous event, like an interrupt, to complete the IO, the driver's submit() function should return the DriverTypes.PENDING status. In such cases, the asynchronous event results in a callback. The callback function and callback function argument were specified during the driver's open() call.

If the driver encounters an error after returning DriverTypes_PENDING, the driver should update the packet with the error ID and call the callback.

In the callback, Stream will check for errors in the IO packet. The error in the packet could be driver-specific. In case of success, the Error.Id in the packet should be NULL. The driver needs to update the size field to reflect the actual size of processed data.

In all cases the driver is responsible for raising errors except in the case when the driver's submit() function returns DriverTypes.PENDING. In this case the driver fills the Error.Id in the IO Packet.

The driver is expected to queue up IO packets for transfer if necessary and must not produce an error when given more than one packet.

The driver is non-blocking—that is, it cannot call APIs that block as it is expected that the higher layer will wait for IO to be completed and take action in case of a timeout.

2.5.7 IDriver_control() Function

Stream_control() calls the driver's control() function with driver-specific control commands. The driver's control() function is called using the following syntax:

```
Void IDriver_control(
    IDriver_Handle          handle,
    Ptr                     chanHandle,
    DriverTypes.ControlCmd  cmd,
    UArg                    cmdArgs,
    Error.Block             *eb);
```

For example, the application could call Stream_control() to change channel-specific parameters like the baud rate of a UART channel. The driver should process the command and return values in the argument when necessary.

All drivers can define their own control commands in their spec files.

If the driver does not recognize the control command, it should raise an error. It could use the generic DriverTypes_E_NOTIMPL error message.

A control command of DriverTypes.CHAN_ABORT is used to abort/discard all packets queued up for a channel. This control command must be supported by all drivers. Note that when the driver receives the abort control command, it must abort ALL packets and call the callback for every packet. If a packet is currently in progress, the driver must attempt to shut down DMA, etc. and return the packet. Aborted packets need to be updated with the error field set to DriverTypes.E_Aborted.

2.5.8 Driver ISRs

Many drivers have separate Tx and Rx interrupts. Typically the following activities need to be performed in the ISR(s):

- ❏ Dequeue the IO packet.
- ❏ Set up the next transfer or service request.
- ❏ Call the class driver callback for the completed packet.

2.6 The IConverter Interface

The capabilities of the Stream module play an important role in fostering device independence within SYS/BIOS in that logical devices insulate your application programs from the details of designating a particular device. For example, /dac is a logical device name that does not imply any particular DAC hardware. The device-naming convention adds another dimension to device-independent I/O that is unique to SYS/BIOS—the ability to use a single name to denote a stack of devices.

By stacking certain data-scaling or message-passing modules atop one another, you can create virtual I/O devices that further insulate your applications from the underlying system hardware.

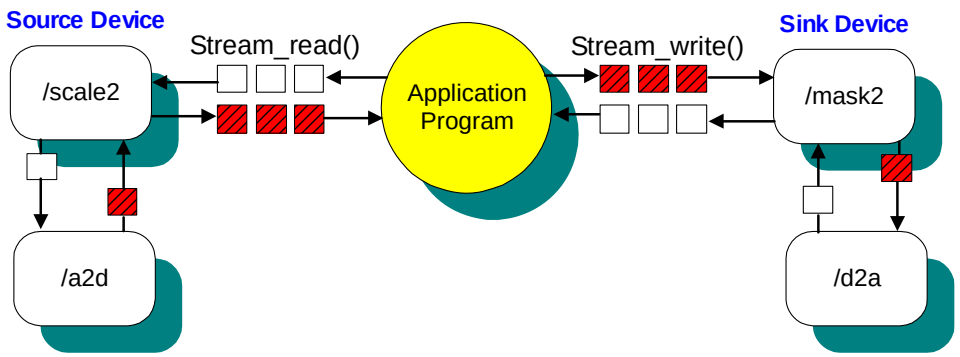
Consider, as an example, a program implementing an algorithm that inputs and outputs a stream of fixed-point data using a pair of A/D-D/A converters. However, the A/D-D/A device can take only the 14 most significant bits of data, and the other two bits have to be 0 if you want to scale up the input data. Instead of cluttering the program with excess code for data conversion and buffering to satisfy the algorithm's needs, we can open a pair of virtual devices that implicitly perform a series of transformations on the data produced and consumed by the underlying real devices as shown.

The virtual input device, /scale2/a2d, actually comprises a stack of two devices, each named according to the prefix of the device name specified in your configuration file.

- **/scale2** designates a device that transforms a fixed-point data stream produced by an underlying device (/a2d) into a stream of scaled fixed-point values.
- **/a2d** designates a device managed by the A/D-D/A device driver that produces a stream of fixed-point input from an A/D converter.

The virtual output device, /mask2/d2a, likewise denotes a stack of two devices.

The following figure shows the flow of empty and full frames through these virtual source and sink devices as the application program calls Stream functions.



Stacking drivers do not implement the IDriver interface. Instead, they implement a different interface called IConverter. Converters are not added to the driver table. Instead, the Stream module maintains a table of converters. Stream searches both the converter table and the driver table for matches. However, note that names must be unique within all the items in both tables. The Stream module searches the converter table first.

Here is an example script showing use of an IConverter module called Transformer. This converter is provided with IPC in the ti.sdo.io.converters package.

Configuration code: The following configuration file excerpt statically creates a rtdxInst driver instance and a Transformer instance called transInst:

```
var Rtdx = xdc.useModule('ti.rtdx.driver.RtdxDrv');
var Transformer = xdc.useModule('ti.sdo.io.converters.Transformer');
var DriverTable = xdc.useModule('ti.sdo.io.DriverTable');
var Stream = xdc.useModule('ti.sdo.io.Stream');

Var rtdxInst = RtdxDrv.create();
DriverTable.addMeta("/in", rtdxInst);

var transParamsIn = new Transformer.Params();
transParamsIn.fxn = Transformer.multiply;
Var transInst = Transformer.create(transParamsIn);
Stream.addMeta("/scale", transInst);
```

Run-time code: This C code fragment creates the stacking driver instance:

```
handleIn = Stream_create("/scale/in", Stream_INPUT, NULL, NULL);
```

2.7 The IomAdapter Module

The IomAdapter module allows legacy IOM drivers to work with SYS/BIOS 6. This module translates the IDriver interface to the old IOM interface.

The IomAdapter module implements the IDriver interface. Each instance of this module represents a legacy IOM driver module.

The following example uses the IomAdapter module with a legacy IOM driver:

Configuration code: The following configuration file excerpt statically creates an IomAdapter instance and adds it to the DriverTable:

```
var iomPrms = new IomAdapter.Params();
iomPrms.iomFxns = $externPtr("UART_FXNS");
iomPrms.initFxn = "&UART_init";
iomPrms.deviceParams = $externPtr("UART_DEVPARAMS");
iomPrms.deviceId = 0;
iomInst = IomAdapter.create(iomPrms);

DriverTable.addMeta("/iomUart", iomInst);
```

Run-time code: This C code fragment creates two Stream instances that use the legacy driver:

```
handleIn = Stream_create("/iomUart", Stream_INPUT, NULL, NULL);
handleOut = Stream_create("/iomUart", Stream_OUTPUT, NULL, NULL);
```

2.7.1 Mapping IOM Functions to IDriver Functions

This table lists the legacy IOM functions and their corresponding IDriver functions:

IOM	IDriver
mdBindDev	Driver specific create()
mdUnbindDev	Driver delete()
mdControlChan	IDriver_control()
mdCreateChan	IDriver_open()
mdDeleteChan	IDriver_close()
mdSubmitChan	IDriver_submit()

2.8 Porting the Stream Module to Another Operating System

The Stream module is a RTSC module that uses the xdc.runtime package. It is independent of SYS/BIOS. This makes it possible to port Stream to other operating systems. Here are the steps required to use the Stream module with other operating systems. These steps assume a TI target and TI build tools.

1) Implement a module called GateInterrupt that inherits the IGate interface. This Gate should disable interrupts when locked and restore interrupts when unlocked. Plug the System gate with this module as follows:

```
System.common$.gate = GateInterrupt.create();
```

2) Implement a module that inherits ITimestampProvider. SYS/BIOS implements a target-specific TimestampProvider that can be identical for another OS.

```
Timestamp = xdc.module('xdc.runtime.Timestamp');
var TimestampDelegate xdc.module ("TimestampProviderMod");
Timestamp.SupportProxy = TimestampDelegate;
```

3) Implement one or several modules that implement the ISync interface. Provide a good default for Stream that is equivalent to SyncSem.

```
var Stream = xdc.module('ti.sdo.io.Stream');
Stream.SyncProxy = xdc.useModule('SyncMod');
```

4) Implement the OS-specific backend of the xdc.runtime.knl package. For example, implement ISemThreadSupport for the SemThread module in xdc.runtime.knl.

Chapter 3

The Inter-Processor Communication Package

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

Topic **Page**

3.1 Modules in the IPC Package

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

Table 3-1. IPC package modules

Module	Module Path	
GateMP	GateMP	Manages gates for mutual exclusion of shared resources by multiple processors and threads. See Section 3.6.
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 3.5.
HeapMemMP	ti.sdo.ipc.heaps. HeapMemMP	Variable-sized shared memory Heaps. See Section 3.5.
HeapMultiBufMP	ti.sdo.ipc.heaps. HeapMultiBufMP	Multiple fixed-sized shared memory Heaps. See Section 3.5.
Ipc	ti.sdo.ipc.Ipc	Provides Ipc_start() function and allows startup sequence configuration. See Section 3.2.
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 3.4.
MessageQ	ti.sdo.ipc.MessageQ	Variable size messaging module. See Section 3.3.
TransportShm	ti.sdo.ipc.transports. TransportShm	Transport used by MessageQ for remote communication with other processors via shared memory. See Section 3.3.11.
Notify	ti.sdo.ipc.Notify	Low-level interrupt mux/demuxer module. See Section 3.7.
NotifyDriverShm	ti.sdo.ipc.notifyDrivers. NotifyDriverShm	Shared memory notification driver used by the Notify module to communicate between a pair of processors. See Section 3.7.
SharedRegion	ti.sdo.ipc.SharedRegion	Maintains shared memory for multiple shared regions. See Section 3.8.

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

Table 3-2. IPC package 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.

3.1.1 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 RTSC-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.

3.1.2 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.

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().

3.1.3 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 != MessageQ_S_SUCCESS) {
        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. 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.

3.2 Ipc Module

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[])
{
    /* Call Ipc_start() */
    Ipc_start();

    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 3.8 for more about the SharedRegion module.

3.2.1 Ipc Module Configuration

In a RTSC 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 3.2.2). For example, the following statements configure functions named myAttachFxn() and myDetachFxn():

```
Ipc.userFxn.attach = '&myAttachFxn';
Ipc.userFxn.detach = '&myDetachFxn';
```

3.2.2 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 only with the Ipc.ProcSync_PAIR configuration setting (the default). The ProcSync_PAIR configuration 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.

3.3 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.

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.)

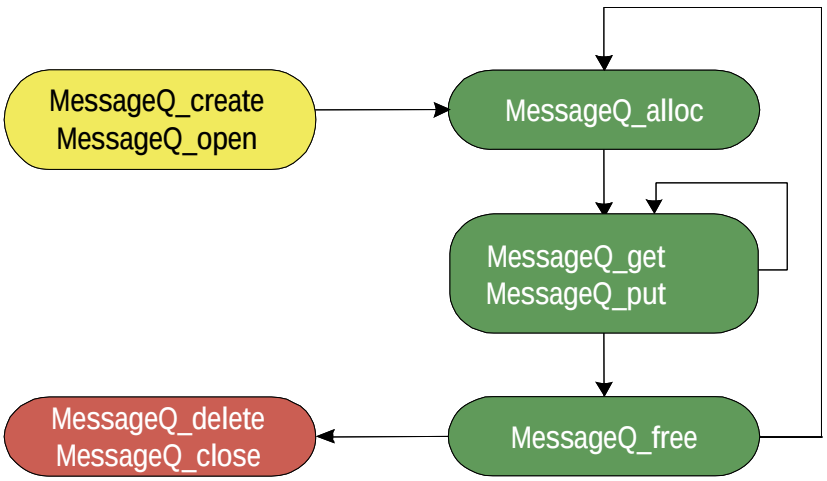
The following are key features of the MessageQ module:

- 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.

3.3.1 Configuring the MessageQ Module

You can configure a number of module-wide parameters for MessageQ in your RTSC 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 parameters 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;
```

3.3.2 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 3.3.9 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. However, the default synchronizer is NULL.

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.

3.3.3 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_FAIL. 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 != MessageQ_S_SUCCESS);
```

3.3.4 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.

3.3.4.1 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 parameter. 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 3.3.4.2, *MessageQ Allocation Without a Heap* for more information.

Heaps can be unregistered via MessageQ_unregisterHeap().

3.3.4.2 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.

3.3.5 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 3.3.10 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 3.3.11 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 3.3.8 for more about message priorities.

3.3.6 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 != MessageQ_S_SUCCESS) {
    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 3.3.8.
- MessageQ_getMsgSize() gets the size of the message in bytes.
- MessageQ_getReplyQueue() gets the ID of the queue provided by MessageQ_setReplyQueue(). See Section 3.3.10.

3.3.7 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.

3.3.8 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 3.3.11, *Remote Communication via Transports* for information about the handling of priority by transports.

3.3.9 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 3.3.11.)

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);
```

3.3.10 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.

3.3.11 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 parameters 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.

3.3.11.1 Custom Transport Implementations

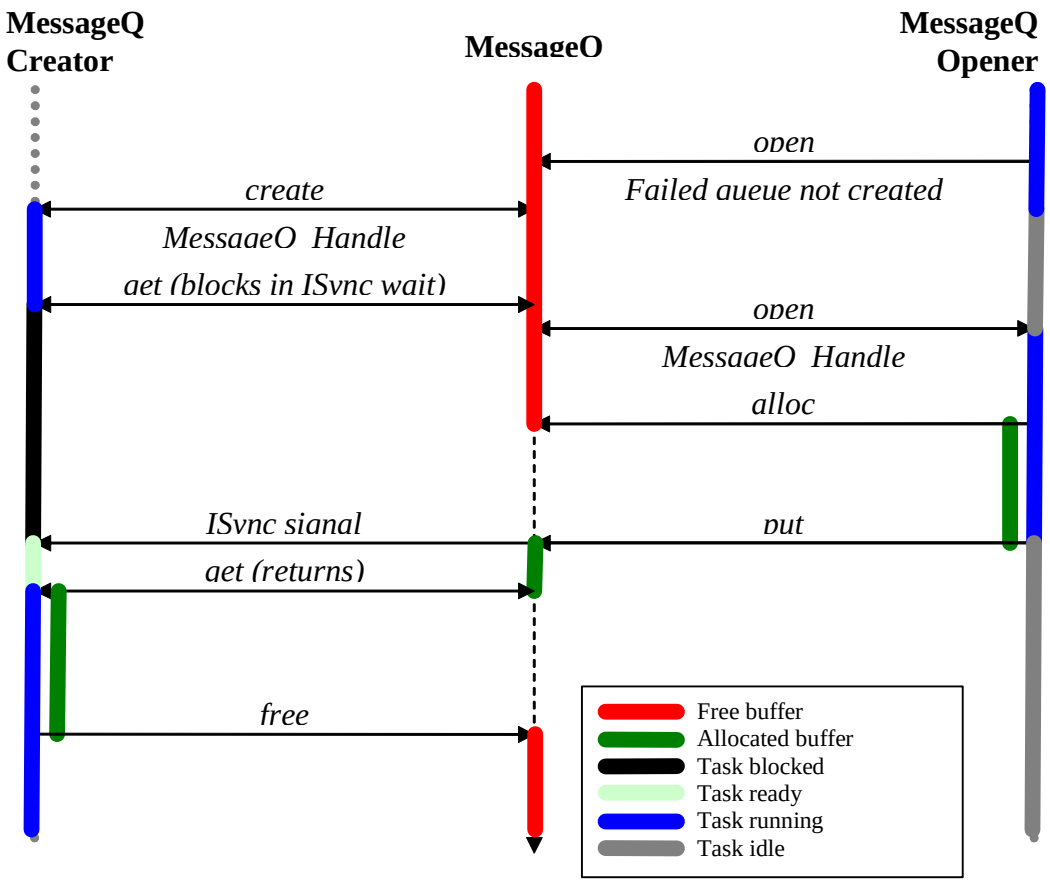
Transports can register and unregister themselves dynamically. That is, if the transport instance is deleted, it should unregister with MessageQ.

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)
```

3.3.12 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.



3.4 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.

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 3.8), 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");
}
```

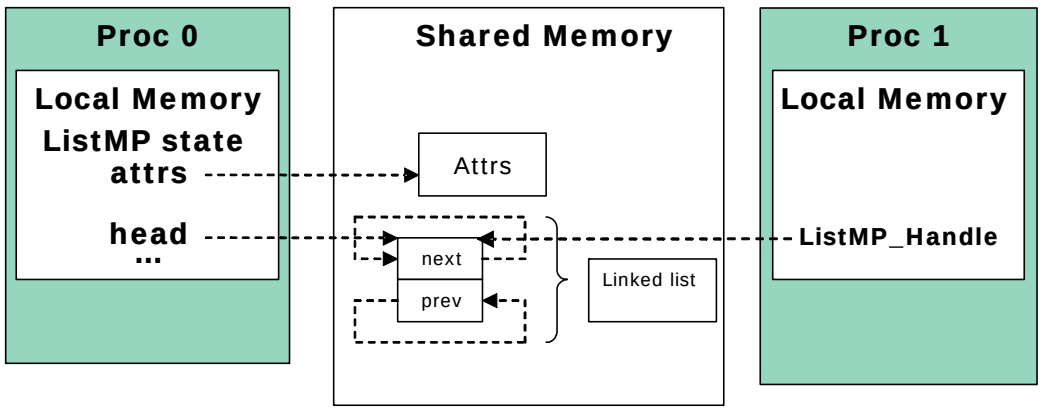
This example places a sequence of even numbers in a ListMP instance:

```
/* 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.

3.5 Heap*MP Modules

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.

3.5.1 Configuring a Heap*MP Module

In addition to configuring Heap*MP instances, you can set module-wide configuration parameters. For example, the maxNameLen parameter lets you set the maximum length of heap names. The track[Max]Allocs module configuration parameter 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.

3.5.2 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 implementation 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.

3.5.3 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);
```

3.5.4 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().

3.5.5 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.

3.5.6 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.

3.5.7 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.

3.5.8 Querying Heap Statistics

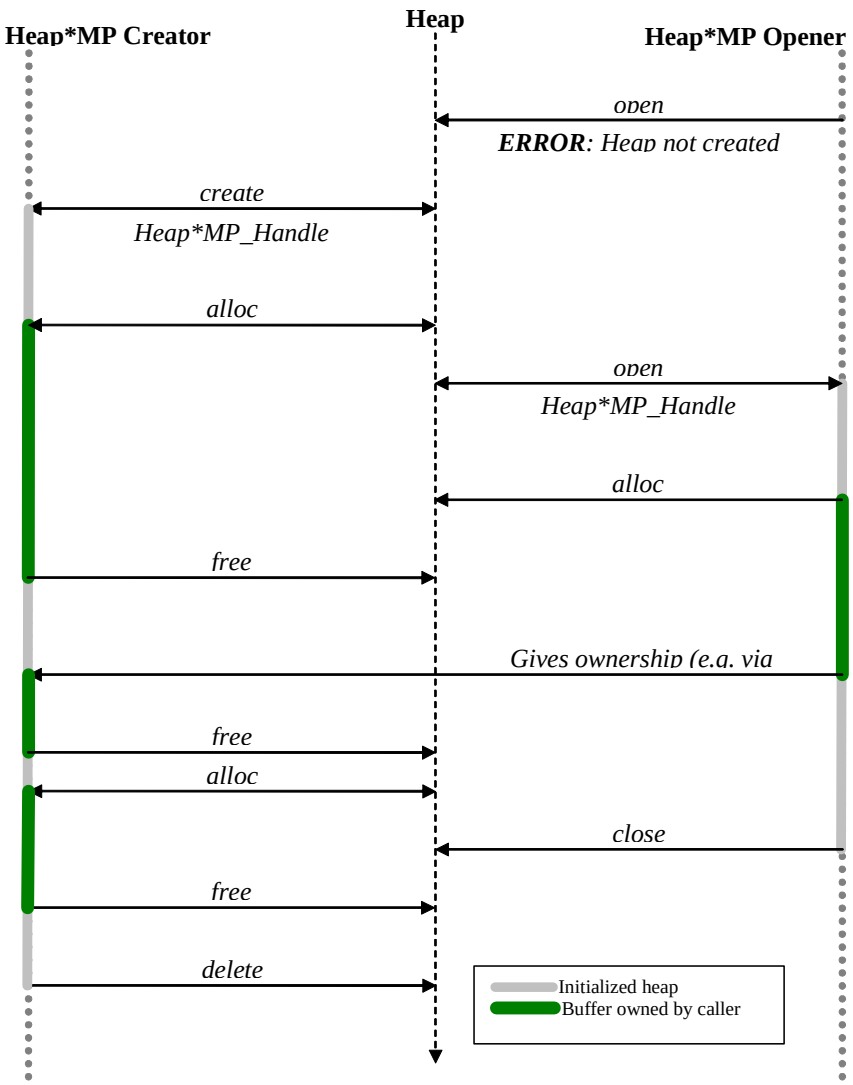
Both heap modules support use of the xdc.runtime.Memory module's Memory_getStats() and Memory_query() functions on the heap.

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.

3.5.9 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.



3.6 GateMP Module

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.

3.6.1 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 parameter. The options for params.remoteProtect are as follows:

- **GateMP_RemoteProtect_NONE.** Creates only the local gate specified by the localProtect parameter.
- **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 parameter. 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.

3.6.2 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.

```
GateMP_Params_init(&gateParams);
while (
    (GateMP_open(&gateHandle, &gateParams)) < 0
);
```

3.6.3 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.

3.6.4 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.

3.6.5 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 parameter 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);
```

3.6.6 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);
```

3.6.7 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.

3.6.8 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 parameter. If a name is supplied, it must be unique for all GateMP instances.

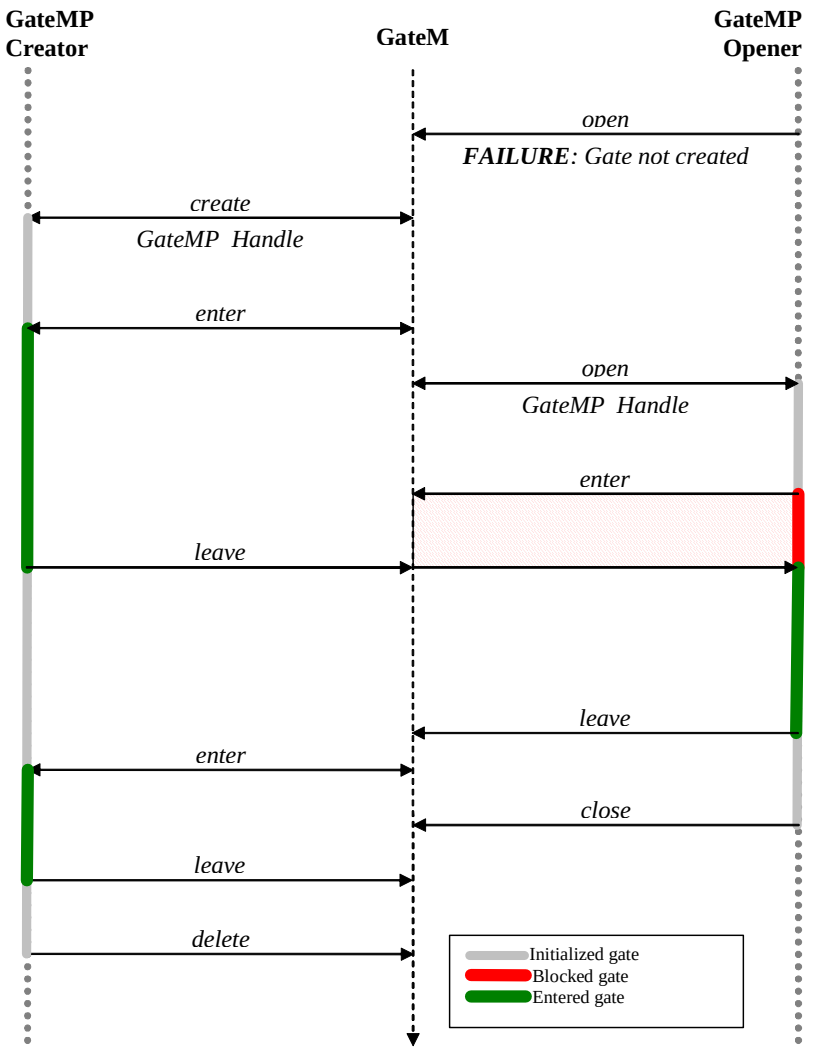
Other modules can use GateMP instances to protect access to their shared memory resources. For example, the NameServer name tables are protected by setting the "gate" parameter of the ti.sdo.utils.NameServer module.

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

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

3.6.9 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.



3.7 Notify Module

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

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 Ipc_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(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");
/* Program.global.EVENTID set to 10
 * in notify_common.cfg.xs */

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.

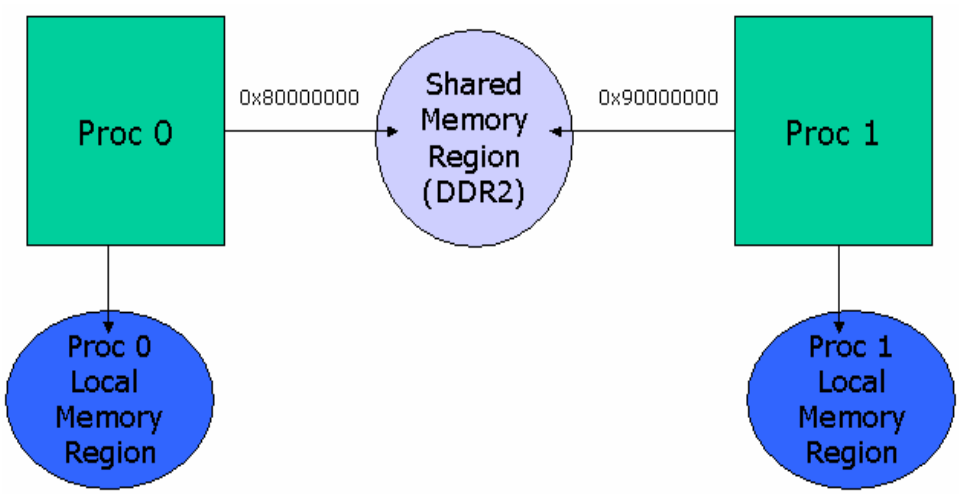
"Loopback" mode, which is enabled by default, allows notifications to be registered and sent locally. This is accomplished by supplying the processor's own MultiProc ID to Notify APIs. Line ID 0 (zero) is always used for local notifications. See the notify_loopback example in <ipc_install_dir>\packages\ti\sdo\ipc\examples\singlecore.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

3.8 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.

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.



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 parameter. 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.

3.8.1 Adding Table Entries Statically

To create a shared region lookup table in the RTSC 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;
```

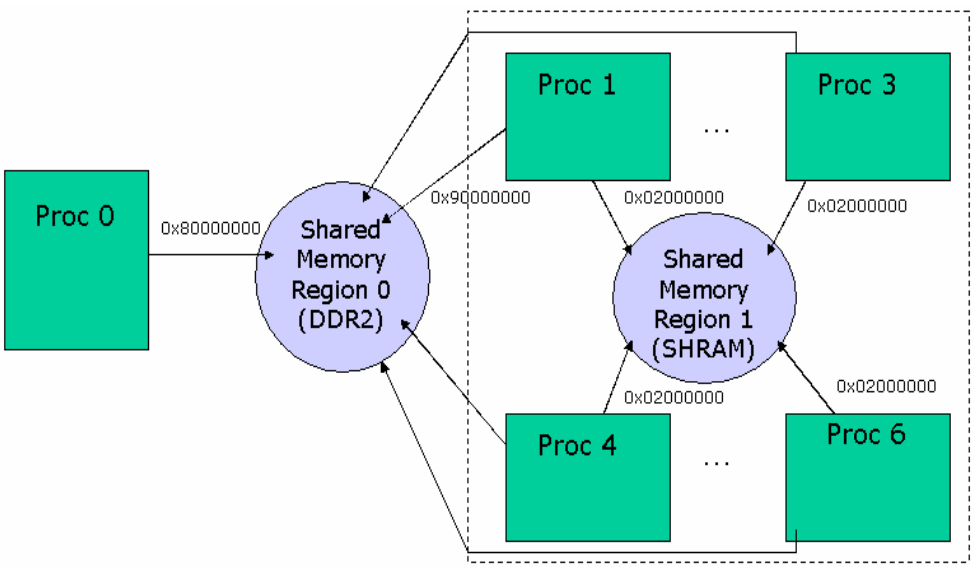
Then, use the SharedRegion.setEntryMeta() method in the configuration file to specify the parameters of the entry.

```
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.

3.8.2 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 parameter, 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.

3.8.3 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);
```

3.8.4 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 4

The Utilities Package

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

Topic **Page**

4.1 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 4.2.
- **MultiProc.** This module stores processor IDs in a centralized location for multi-processor applications. See Section 4.3.
- **NameServer.** This module manages name/value pairs for use by other modules. See Section 4.4.

4.2 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.

4.2.1 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
```

4.2.2 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);
```

4.2.3 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
}
```

4.2.4 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.

4.3 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 MultiProc.setConfig() function tells the MultiProc module:

- The specific processor for which this application is being built
- Which processors out of a set of possible processors on a device are being used by the multi-processor application

Each processor stored 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 RTSC 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());
```

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

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

MultiProc_getNumProcessors() evaluates to the number of processors

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

4.4 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.

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 either statically or dynamically.

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 (statically) or NameServer_ALLOWGROWTH (dynamically), 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.

To create a NameServer instance statically, define a NameServer.params structure and set parameters as needed. Then, use the NameServer.create() method to create the instance.

The following examples create NameServer instances both statically and dynamically. Both instances allow 10 as the maximum number of runtime entries (instead of using ALLOWGROWTH). These examples also specify where to allocate the memory needed for the tables (instead of using the default).

This example configuration code (.cfg) creates a NameServer instance statically:

```
var params = new NameServer.Params;
params.tableHeap = myHeap;
params.maxRuntimeEntries = 10;
var handle = NameServer.create("myTable", params);
```

This example C code creates a NameServer instance dynamically:

```
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 dynamically-created instance. You cannot call NameServer_delete() for a statically-created 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 5

Porting IPC

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

Topic **Page**

5.1 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" for use by MessageQ. The interface definition is in ti.sdo.ipc.interfaces.IMessageQTransport.
- "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.

5.2 Other Porting Tasks

You will likely need to specify custom shared region(s) in your configuration file. For details, see Section 3.8, *SharedRegion Module*.

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

Chapter 6

Optimizing IPC Applications

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

Topic **Page**

6.1 Optimizing Runtime Performance

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

- In CCS, use the whole_program or whole_program_debug RTSC Build Profile to build the application. Doing so causes the build tools to optimize the generated image for both cycle count and local code memory usage. The improvement is often significant.
- 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 3.8.1 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 parameter 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 3.6.1 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.

6.2 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 3.2.1. 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 3.7 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 number of reserved events (default of 3),
 * allowing NameServer and MessageQ to use events. */
Notify.reservedEvents = 0;

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

var NameServerRemoteNotify =
    xdc.useModule('ti.sdo.ipc.NameServerRemoteNotify');
NameServerRemoteNotify.notifyEventId = 0;

var TransportShm =
   xdc.useModule('ti.sdo.ipc.transports.TransportShm');
TransportShm.notifyEventId = 1;
```

- 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",
    });
```

Index

A

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X