

IPC TRAINING

Processor Communication Link

11/13/2014

Version 2.21



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Agenda

- Overview
- IPC Modules
- Configuration
- Scalability
- Optimization
- Footnotes

Overview

- Inter-Processor Communication (IPC)
 - Communication between processors
 - Synchronization between processors
- Two modes
 - Peer-to-peer
 - All cores running TI-RTOS
 - Master-slave
 - Master core running HLOS (e.g. Linux, QNX, Android)
 - Slave cores running TI-RTOS

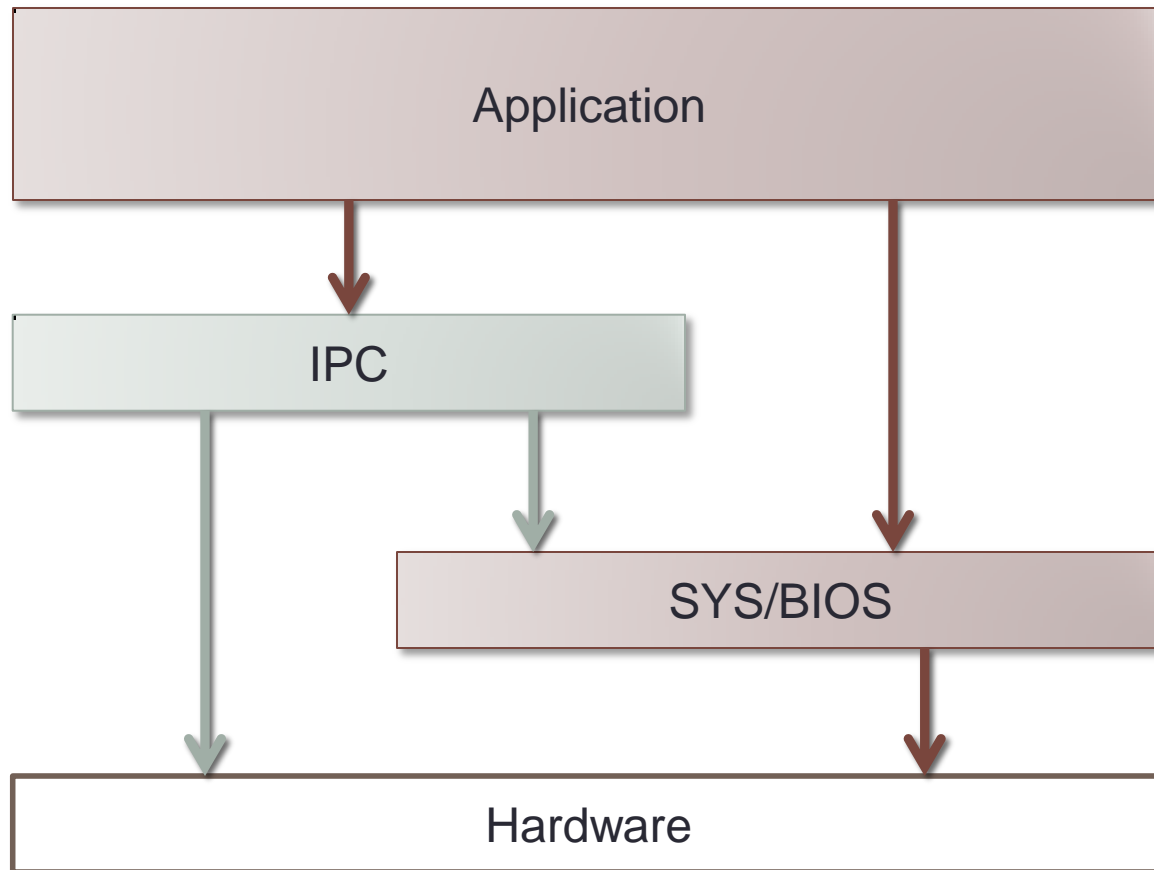
Overview

- Rich device support
 - Homogeneous devices
 - C6472, C6678, ...
 - Heterogeneous devices
 - OMAP5, DRA7XX, TCI6638K2X, OMAP-L138, F28M35, ...

Overview

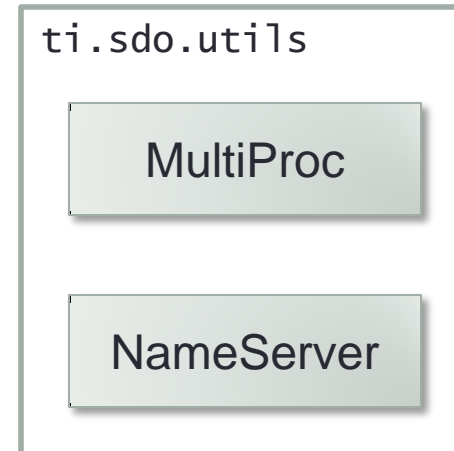
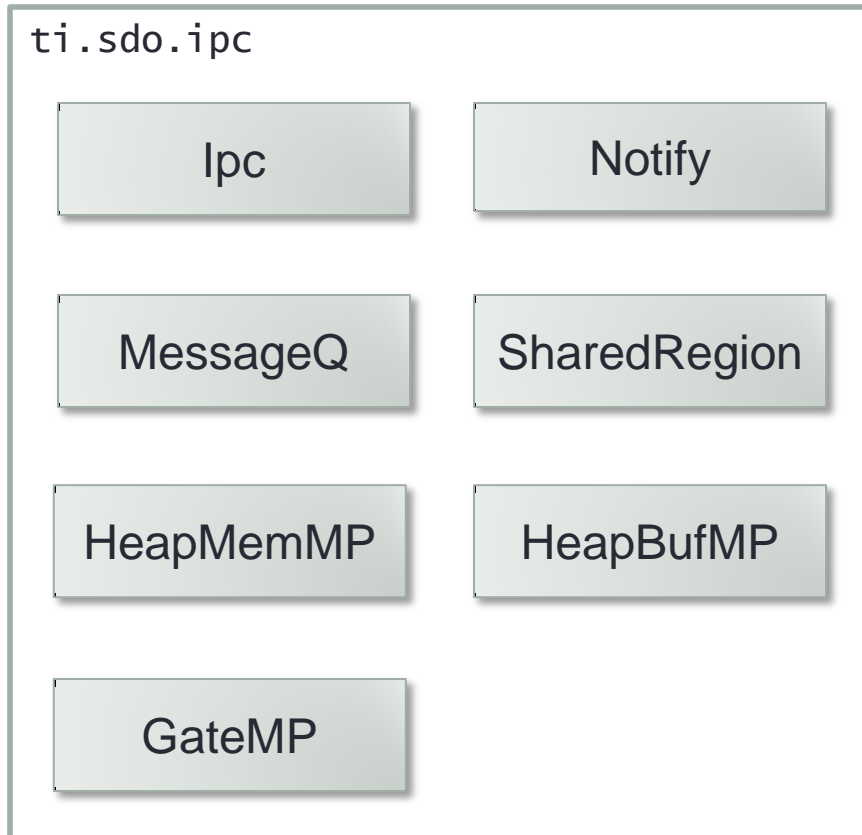
- Flexible design supports many use cases
 - Many thread combinations (Task, Swi, Hwi)
 - Two types of messaging: notification, message queuing
 - Multi- or uni-processor environments
- Hardware abstraction
 - Device-specific support for hardware spinlocks, inter-processor mailbox
 - IPC APIs are same across devices (e.g. GateMP implemented with hardware spinlock or software Peterson algorithm as needed)
- OS-agnostic
 - Same APIs on all operating systems

Overview - Architecture



Overview

- Top-level modules, used by application



Overview - create/open

- Create/open model used to share instances of IPC modules
- Shared objects are “created” by the owner and “opened” by the user(s)
 - Some objects may be opened multiple times (e.g. MessageQ, GateMP).
 - Objects must have system-wide unique names
- Delete/close methods are used to finalize objects
 - Owner should not delete until all users have closed

Overview

- Cache Management
 - Cache coherency operations are performed by the module when shared state is accessed/modified
 - Shared data is padded to prevent sharing cache line with other data
- Data Protection
 - Shared data is protected by a multi-processor gate when accessed/modified

Agenda

- Overview
- **IPC MODULES**
 - Lab 1 – ex01_hello
- Configuration
- Scalability
- Optimization
- Footnotes

IPC Modules

- Ipc – IPC Manager
- MessageQ – send and receive messages
- NameServer – distributed name/value database
- Notify – send and receive event notifications
- MultiProc – processor identification
- SharedRegion – shared memory address translation
- GateMP – protect a critical section
- HeapMemMP, HeapBufMP – multi-processor memory allocator

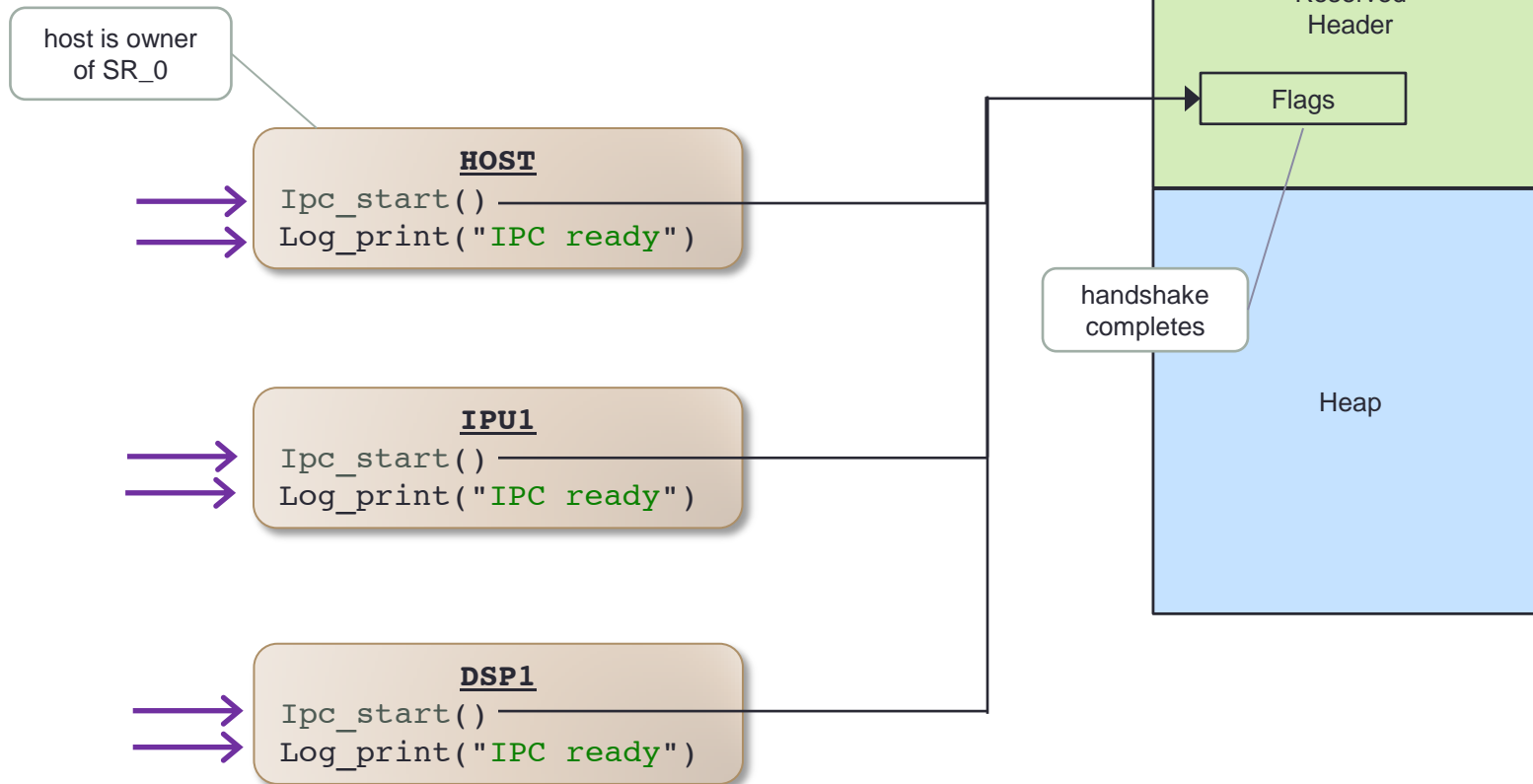
IPC Module

- Ipc – IPC Manager
 - Used to initialize IPC and synchronize with other processors.
 - Application must call `Ipc_start` and `Ipc_attach`.
- Two startup protocols
 - `Ipc.ProcSync_ALL` - all processors start at same time
 - `Ipc.ProcSync_PAIR` – host processor starts first
- Configuration
 - `Ipc.procSync` configures startup protocol
 - When using `Ipc.ProcSync_ALL`, `Ipc_attach` is called internally from `Ipc_start`. Application does not call `Ipc_attach`.

Ipc Module - Ipc.ProcSync_ALL

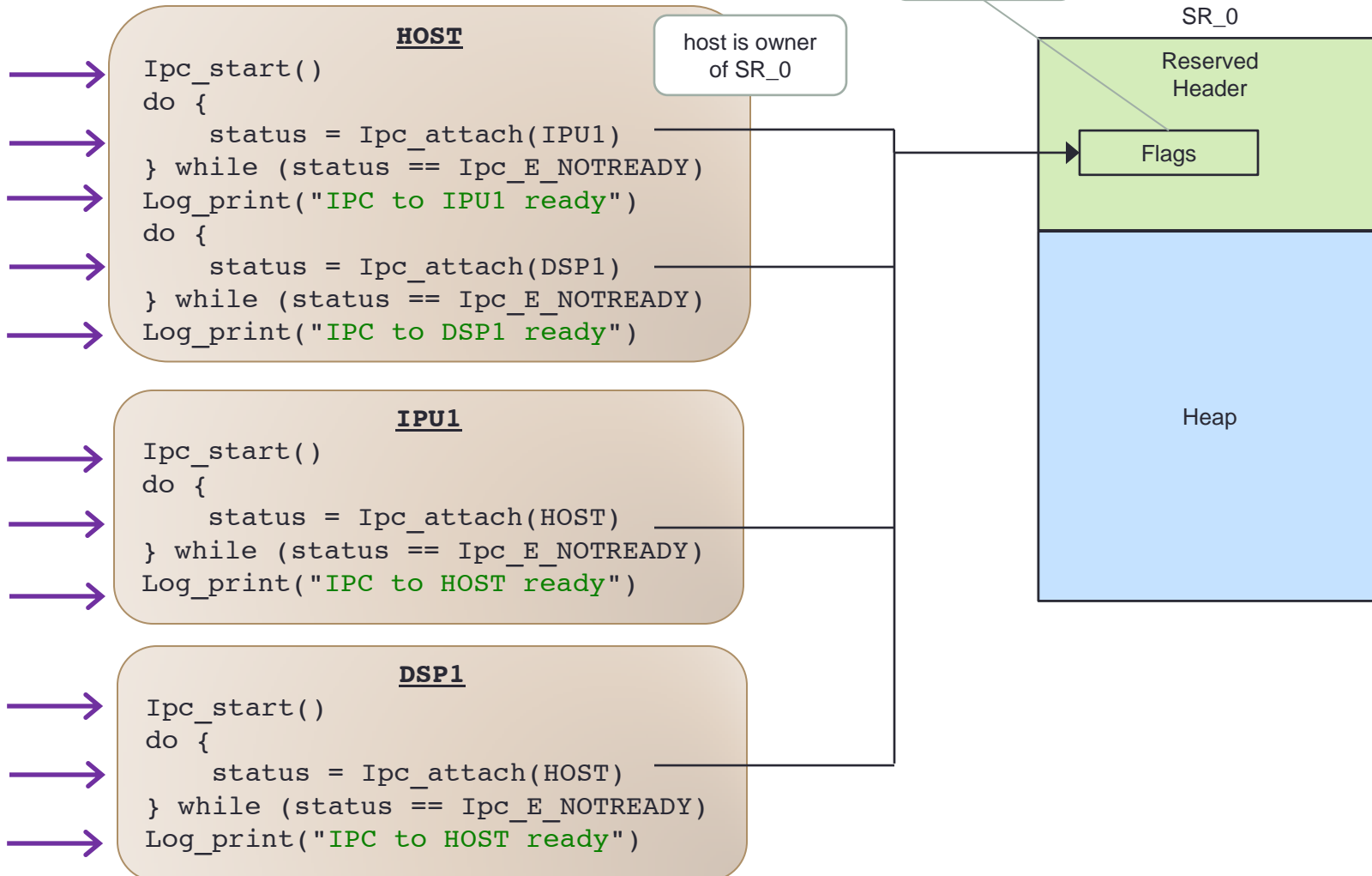
app.cfg

```
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');  
Ipc.procSync = Ipc.ProcSync_ALL;  
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');  
MultiProc.setConfig(proc, ["HOST", "IPU1", "DSP1"]);
```



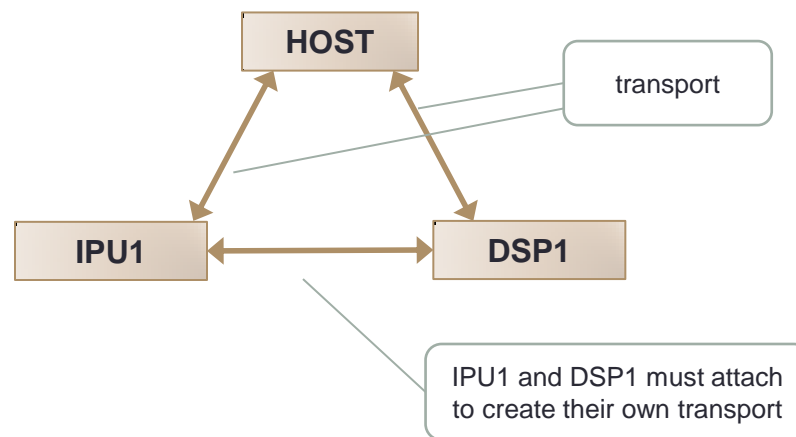
IPC Module - Ipc.ProcSync_PAIR

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



Ipc Module

- Topology
 - Topology expresses which processors communicate with IPC
- Transport created between two processors when they attach
 - In previous example, HOST attached to IPU1 and DSP1.
 - If IPU1 and DSP1 need to communicate with IPC, they must also attach to each other.
- Using `Ipc.ProcSync_ALL` creates transports between all processors.



Ipc Module

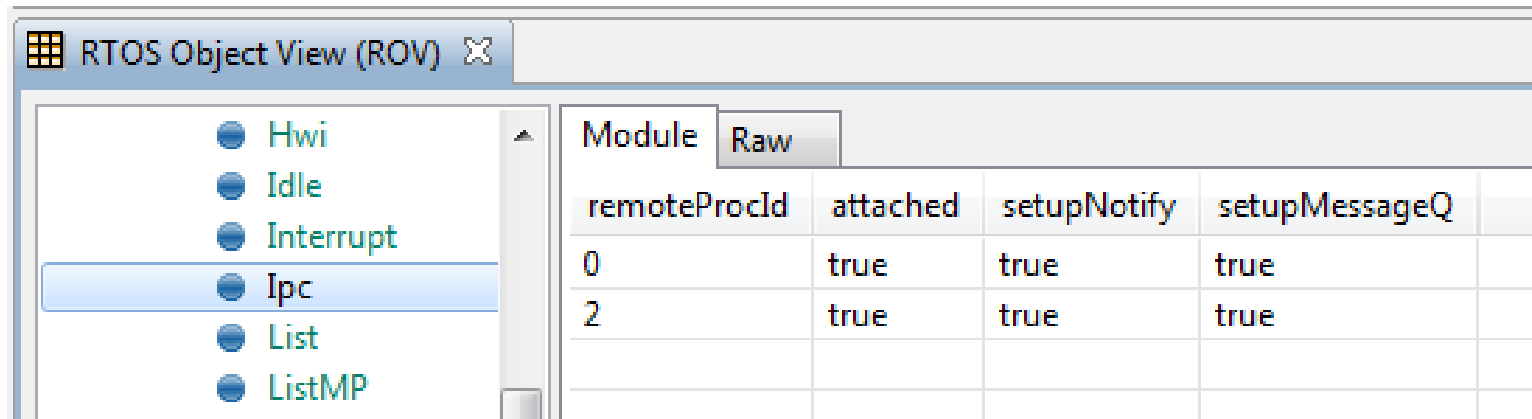
- Why have two modes?
 - Because some applications are static and some are dynamic.
- Static System
 - In a static system all processors are started at the same time. Once the processor is loaded, same application runs forever. IPC startup is part of device boot phase. This is typical for homogeneous devices.
 - Base station, telecommunication, single purpose application.
- Dynamic System
 - In a dynamic system, host processor boots first. Slave processors are booted later depending on application. Slave is shut down when application terminates. Slave reloaded many times, possibly with different executable.
 - Consumer electronics, cell phone.

Ipc Module - API

- API Summary
 - `Ipc_start` – reserve memory, create default gate and heap
 - `Ipc_stop` – release all resources
 - `Ipc_attach` – setup transport between two processors
 - `Ipc_detach` – finalize transport

Ipc Module - ROV

- ROV screen shot

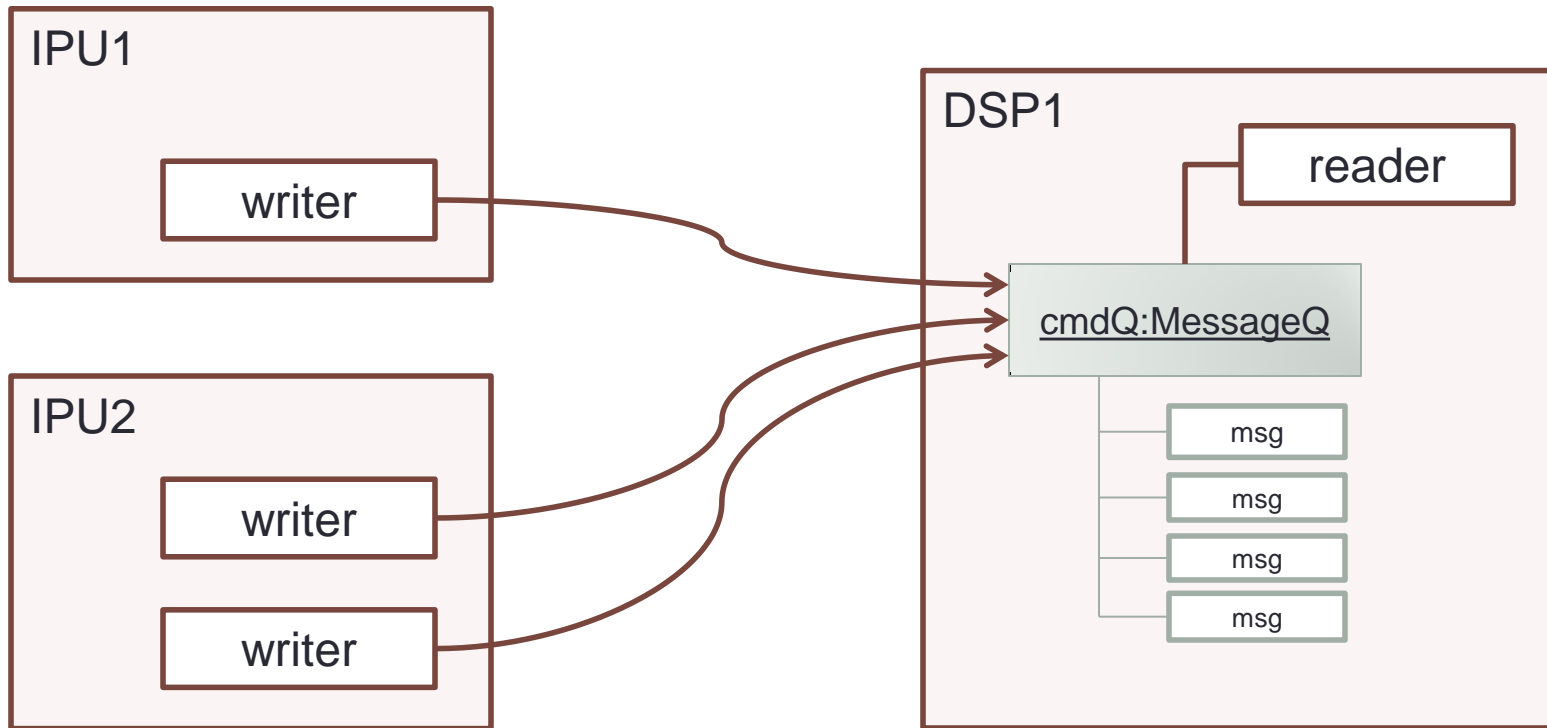


The screenshot shows the RTOS Object View (ROV) window. On the left, a tree view lists several modules: Hwi, Idle, Interrupt, Ipc (selected), List, and ListMP. On the right, a table displays the configuration for the selected Ipc module. The table has two tabs: 'Module' and 'Raw'. The 'Module' tab is active, showing a table with columns: remoteProcId, attached, setupNotify, and setupMessageQ. The table contains two rows of data.

remoteProcId	attached	setupNotify	setupMessageQ
0	true	true	true
2	true	true	true

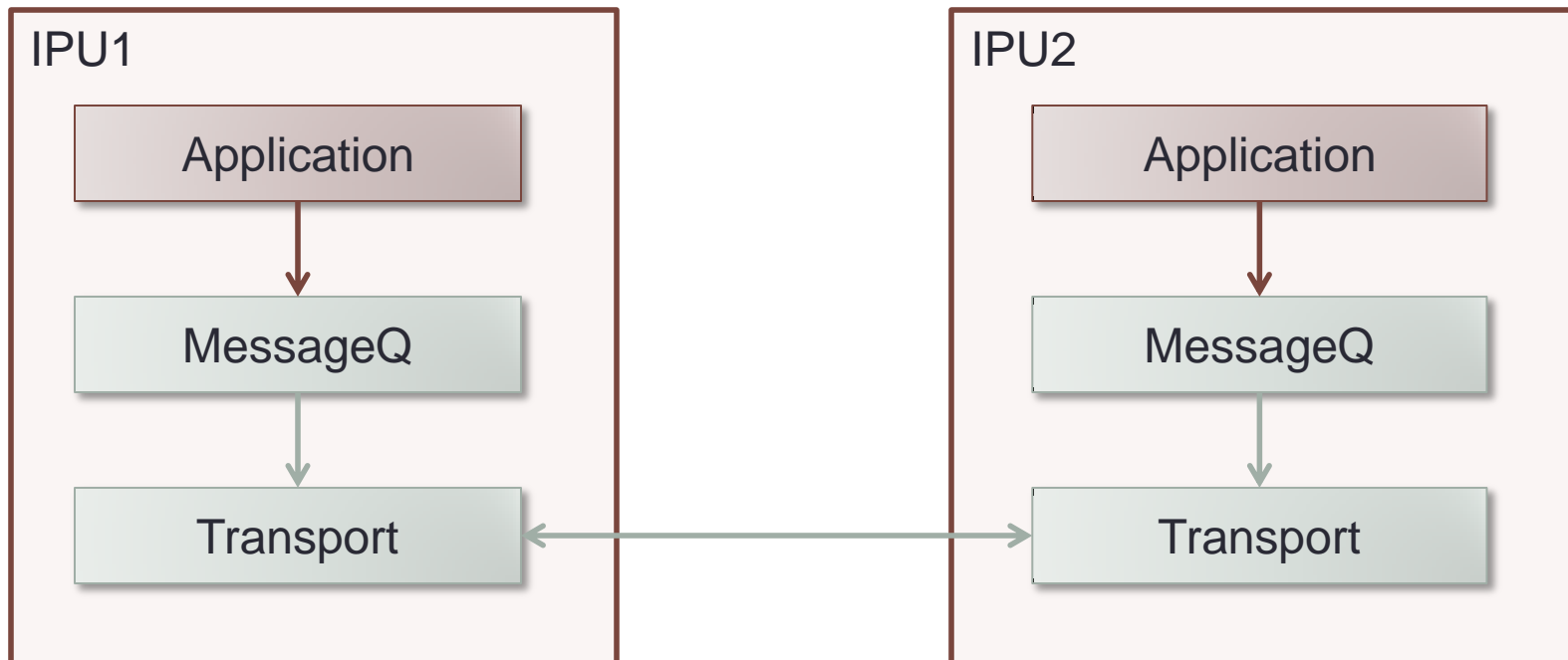
MessageQ

- MessageQ – send and receive messages
- A message queue receives messages
 - Single reader, multiple writers on same message queue
- Message queue instance created by reader, opened by writers



MessageQ – transport

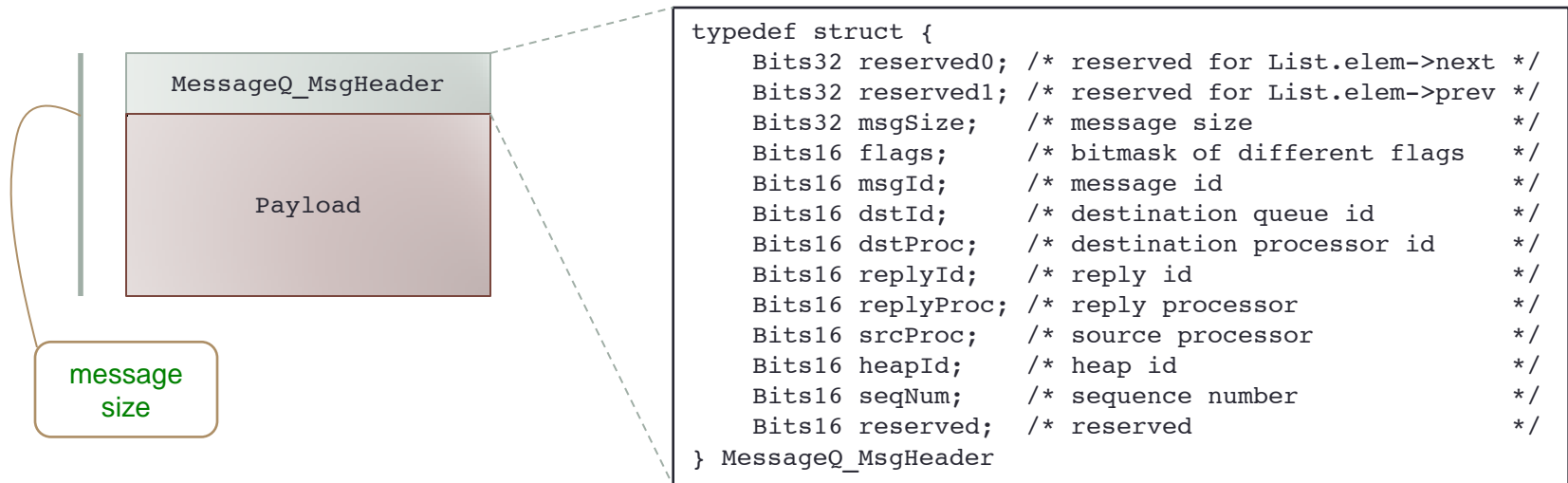
- Transport-independent API, works with local memory, shared memory, copy based transport (SRIO)
- Default transport offers zero-copy transfers via shared memory



MessageQ

- Message structure

- Every message contains an embedded message queue header followed by the payload. Application is responsible for the header allocation.
- Message size must include header and payload sizes.
 - Actual message size is typically padded



```
typedef struct {
    MessageQ_MsgHeader reserved;
    char payload[50];
} App_Msg;
```

MessageQ

- Message types

- MessageQ_MsgHeader is structure type definition.

```
typedef struct {  
    ...  
} MessageQ_MsgHeader;
```

- MessageQ_Msg is pointer to structure type definition.

```
typedef MessageQ_MsgHeader *MessageQ_Msg;
```

- Message Allocation

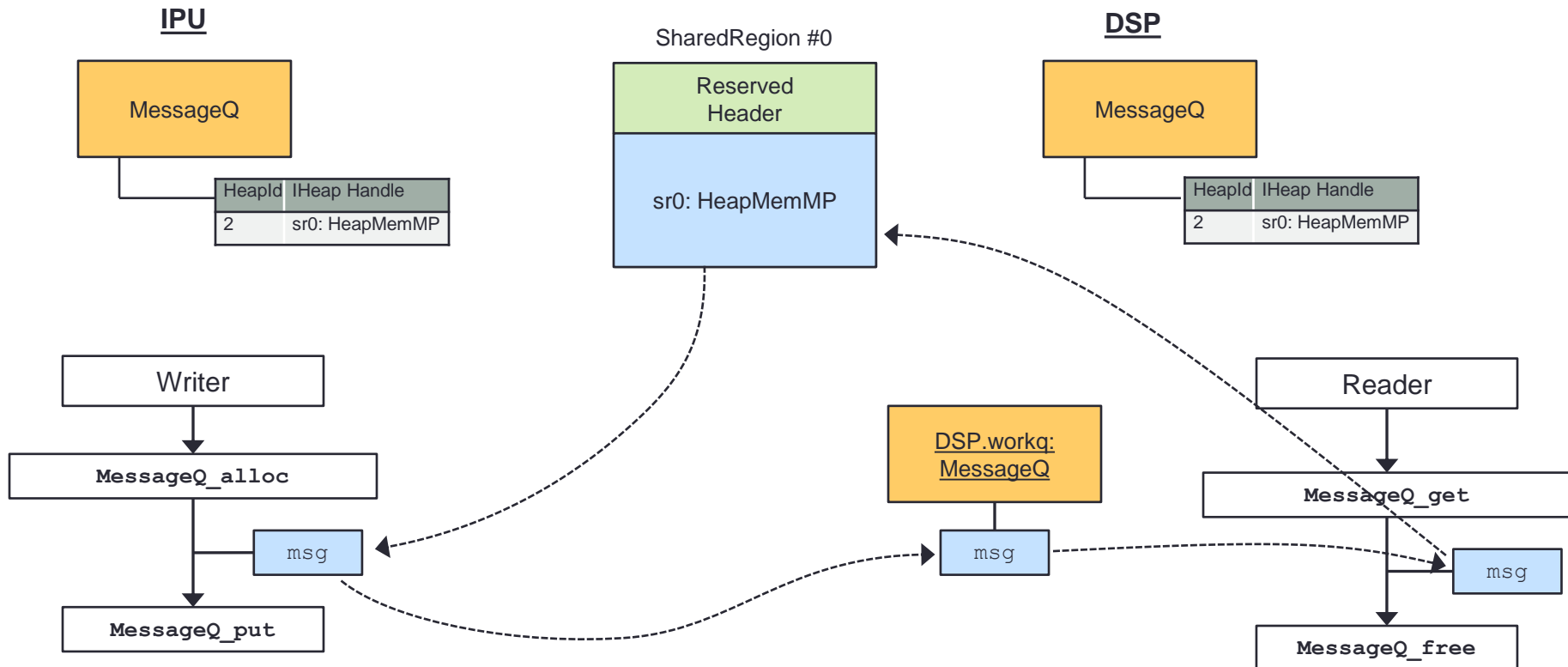
- Message allocation must be large enough to hold the embedded message queue header and your payload. See [Message structure](#).

MessageQ - reader/writer

- IPU → DSP
 - Send a message from IPU to DSP (one-way message)
 - Both processors register a heap with MessageQ module
 - Receiving processor creates a message queue (DSP)
 - Sending processor opens the message queue (IPU)
- Sending processor will...
 - allocate a message
 - write the payload
 - send message
- Receiving processor will...
 - get message
 - read the payload
 - free the message

MessageQ - reader/writer

- IPU → DSP



MessageQ - reader/writer

IPU (writer)

```
#include <xdc/std.h>
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/SharedRegion.h>

#define HEAP_ID 2
#define MSG_SZ sizeof(MessageQ_MsgHeader) + 50

Ptr heap;
MessageQ_Msg msg;
MessageQ_QueueId qid;

heap = SharedRegion_getHeap(0);
MessageQ_registerHeap(HEAP_ID, heap);

do {
    status = MessageQ_open("DSP1.workq", &qid);
    Task_sleep(1);
} while (status == MessageQ_E_NOTFOUND);

while (running) {
    msg = MessageQ_alloc(HEAP_ID, SIZE)
    /* write payload */
    MessageQ_put(qid, msg)
}
```

MessageQ - reader/writer

DSP (reader)

```
#include <xdc/std.h>
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/SharedRegion.h>

#define HEAP_ID 2

Ptr heap;
MessageQ_Handle que;
MessageQ_Msg msg;

heap = SharedRegion_getHeap(0);
MessageQ_registerHeap(HEAP_ID, heap);

que = MessageQ_create("DSP1.workq", NULL);

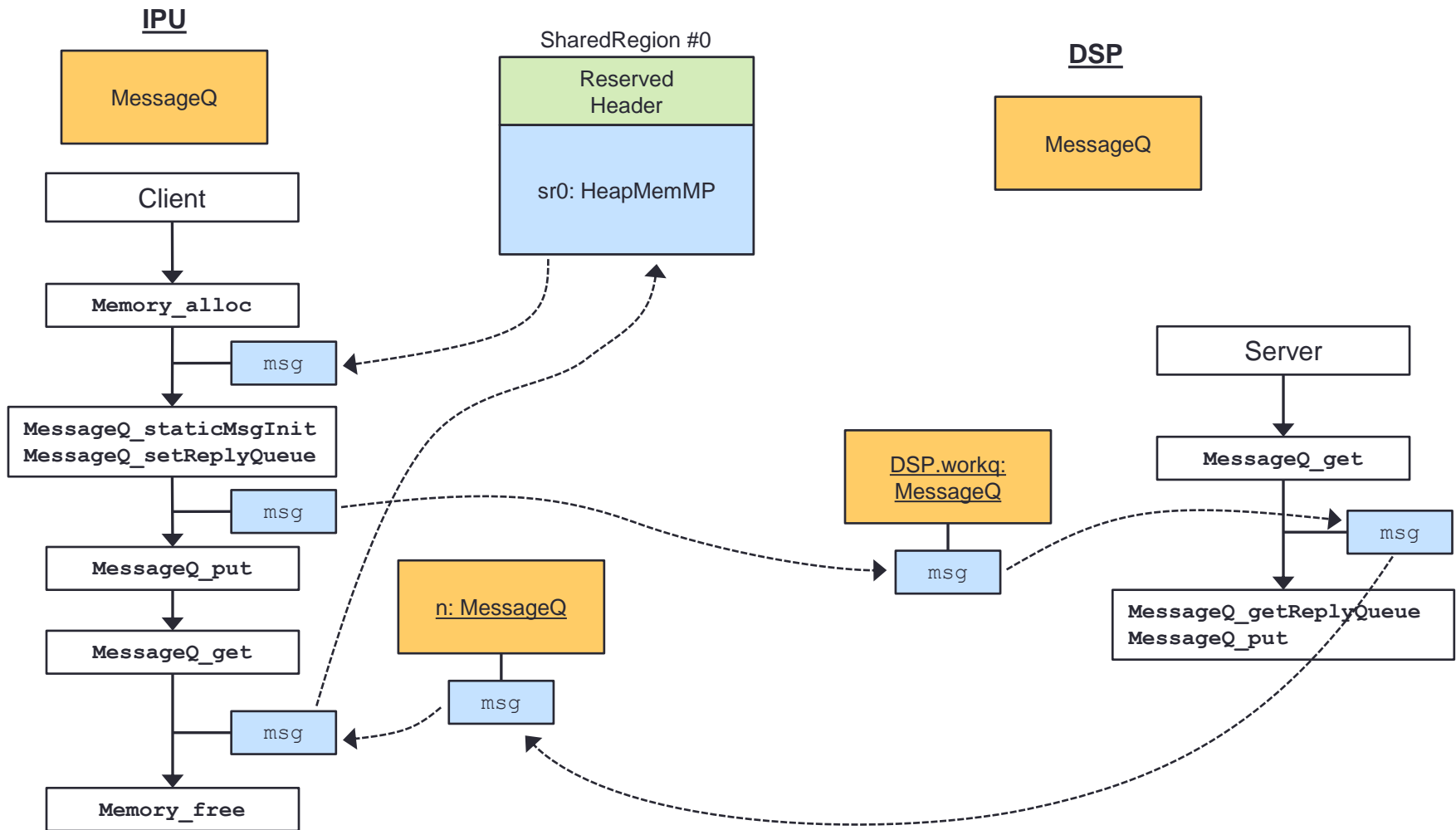
while (running) {
    MessageQ_get(que, &msg, MessageQ_FOREVER);
    /* read payload */
    MessageQ_free(msg);
}
```

MessageQ - client/server

- IPU → DSP → IPU
 - Send a message from IPU to DSP and back again (round trip message)
 - Both processors create a message queue
- IPU processor will...
 - allocate a message
 - write the payload
 - send message
 - wait for return message
 - read payload
 - free the message
- DSP processor will...
 - wait for message
 - read the payload
 - write new payload
 - send message

MessageQ - client/server

- IPU → DSP → IPU



MessageQ - client/server

IPU (client)

```
#include <xdc/std.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/Memory.h>
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/SharedRegion.h>

#define MSG_SZ sizeof(MessageQ_MsgHeader) + 50

IHeap_Handle heap;
MessageQ_Handle ipuQ;
MessageQ_Msg msg;
MessageQ_QueueId dspQ;

heap = (IHeap_Handle)SharedRegion_getHeap(0);
ipuQ = MessageQ_create(NULL, NULL);

do {
    status = MessageQ_open("DSP.workq", &dspQ);
} while (status == MessageQ_E_NOTFOUND);

msg = Memory_alloc(heap, SIZE, 0, NULL);
MessageQ_staticMsgInit(msg, SIZE);
MessageQ_setReplyQueue(ipuQ, msg);
/* write payload */
MessageQ_put(dspQ, msg);

MessageQ_get(ipuQ, &msg, MessageQ_FOREVER);
/* read payload */
Memory_free(heap, msg, SIZE);
```

MessageQ - client/server

DSP (server)

```
#include <xdc/std.h>
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/SharedRegion.h>

MessageQ_Handle dspQ;
MessageQ_QueueId qid;
MessageQ_Msg msg;

dspQ = MessageQ_create("DSP.workq", NULL);

while (running) {
    MessageQ_get(dspQ, &msg, MessageQ_FOREVER);
    /* process payload */
    qid = MessageQ_getReplyQueue(msg);
    MessageQ_put(qid, msg);
}
```

MessageQ

- MessageQ works with any SYS/BIOS threading model:
 - Hwi: hardware interrupts
 - Swi: software interrupts
 - Task: threads that can block and yield
- Variable size messages
- Timeouts are allowed when a Task receives messages
- Message Ownership Rules
 - Acquire ownership with MessageQ_alloc, MessageQ_get
 - Loose ownership with MessageQ_free, MessageQ_put
 - Do not dereference a message when you don't have ownership

MessageQ

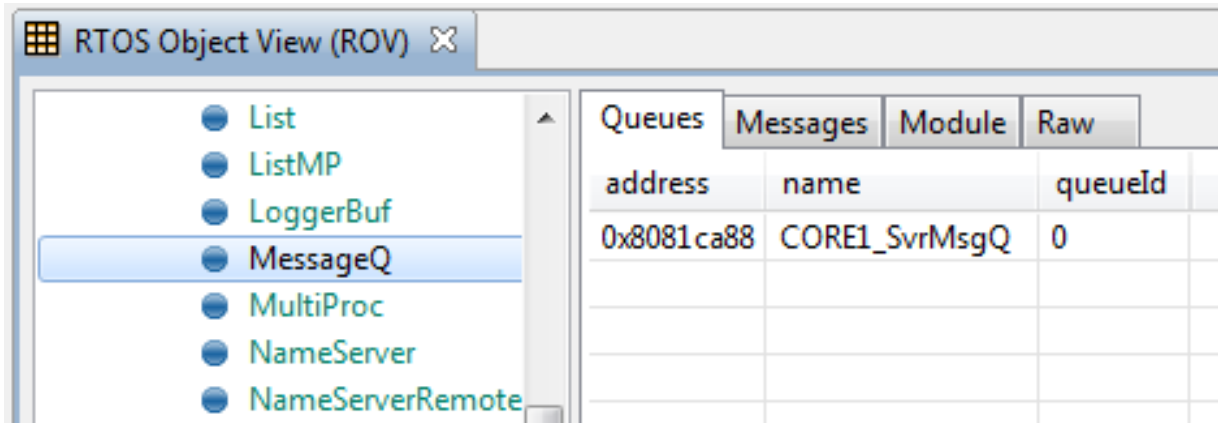
- Uses an IHeap heap implementation to support MessageQ_alloc and MessageQ_free.
- Heaps are coordinated across processors by a common index which is registered using MessageQ_registerHeap API
- Heap ID is stored in message header

MessageQ - API

- API Summary
 - `MessageQ_create` – create a new message queue
 - `MessageQ_open` – open an existing message queue
 - `MessageQ_alloc` – allocate a message from the pool
 - `MessageQ_free` – return message to the pool
 - `MessageQ_put` – send a message
 - `MessageQ_get` – receive a message
 - `MessageQ_registerHeap` – register a heap with MessageQ

MessageQ - ROV

- ROV screen shot



Lab – ex01_hello

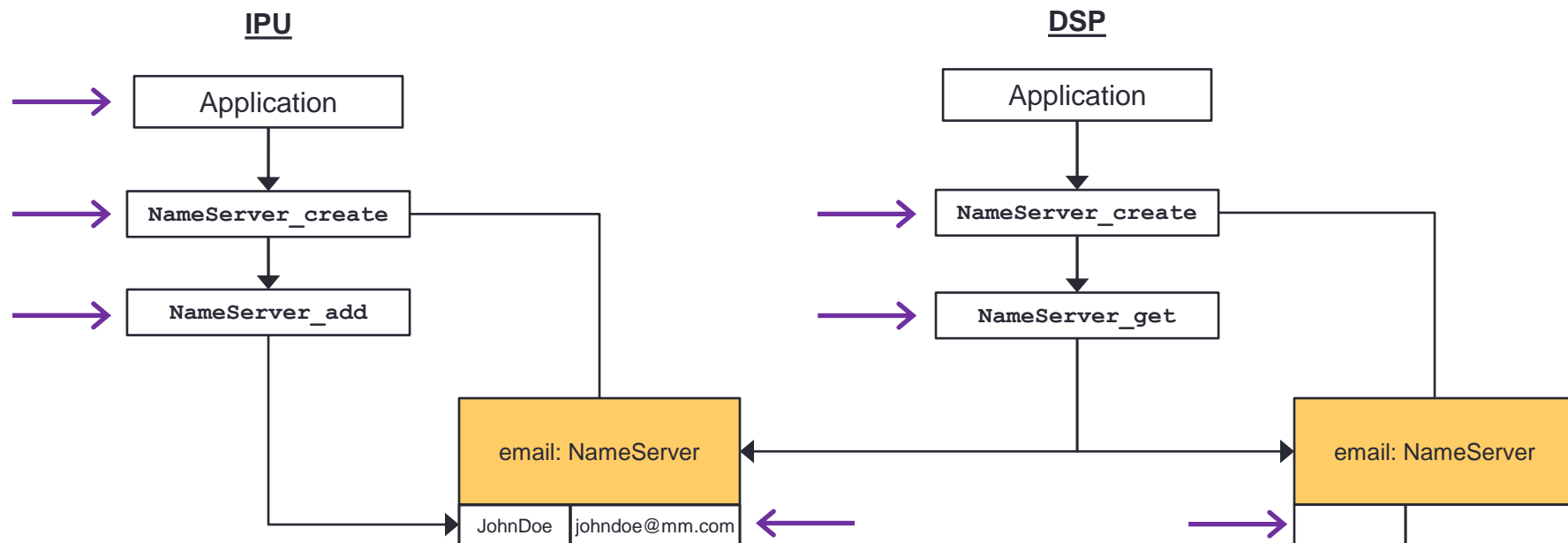
- Please open the PowerPoint slide named IPC_Lab_1_Hello

NameServer Module

- NameServer – distributed name/value database
 - Manages name/value pairs
 - Used for registering data which can be looked up by other processors
- API Summary
 - `NameServer_create` – create a new database instance
 - `NameServer_add` – add a name/value entry into database
 - `NameServer_get` – retrieve value for given name

NameServer

- IPU – create NameServer instance
- DSP – query IPU for name/value pair



NameServer

- IPU – create NameServer instance
- DSP – query IPU for name/value pair

IPU

```
#include <string.h>
#include <xdc/std.h>
#include <ti/ipc/NameServer.h>

NameServer_Handle ns;
Char buf[];
Int len;

ns = NameServer_create("email", NULL);

strcpy(buf, "johndoe@mm.com");
Int len = strlen(buf);

NameServer_add(ns, "JohnDoe", buf, len);
```

DSP

```
#include <xdc/std.h>
#include <ti/ipc/NameServer.h>

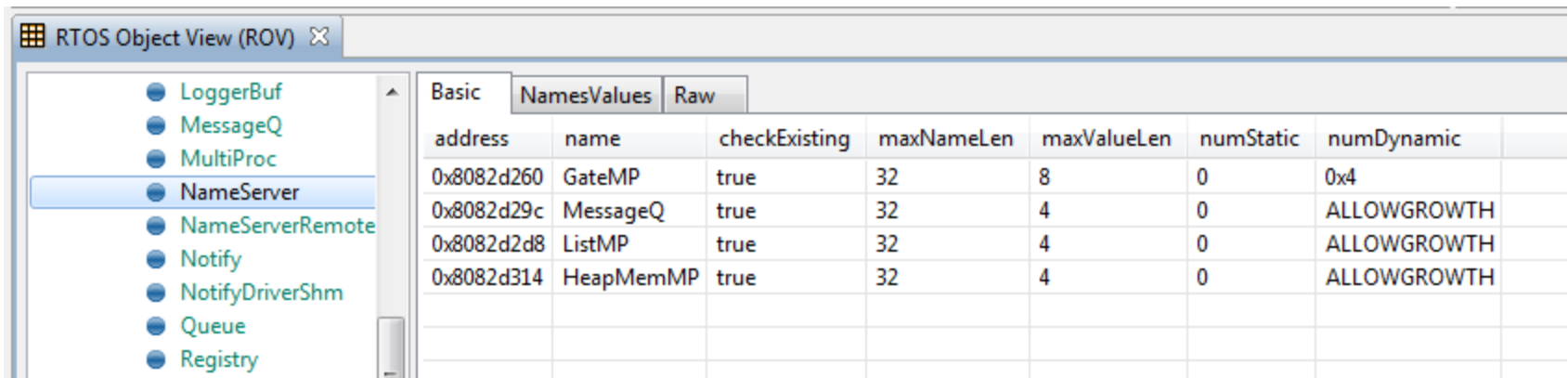
NameServer_Handle ns;
Char buf[32];
Int len;

ns = NameServer_create("email", NULL);

NameServer_get(ns, "JohnDoe", buf, &len, NULL);
```

NameServer Module

- ROV screen shot



The screenshot shows the RTOS Object View (ROV) window with the NameServer module selected in the left-hand tree. The main pane displays a table with columns for address, name, checkExisting, maxNameLen, maxValueLen, numStatic, and numDynamic. The table lists four objects: GateMP, MessageQ, ListMP, and HeapMemMP.

address	name	checkExisting	maxNameLen	maxValueLen	numStatic	numDynamic
0x8082d260	GateMP	true	32	8	0	0x4
0x8082d29c	MessageQ	true	32	4	0	ALLOWGROWTH
0x8082d2d8	ListMP	true	32	4	0	ALLOWGROWTH
0x8082d314	HeapMemMP	true	32	4	0	ALLOWGROWTH

Notify Module

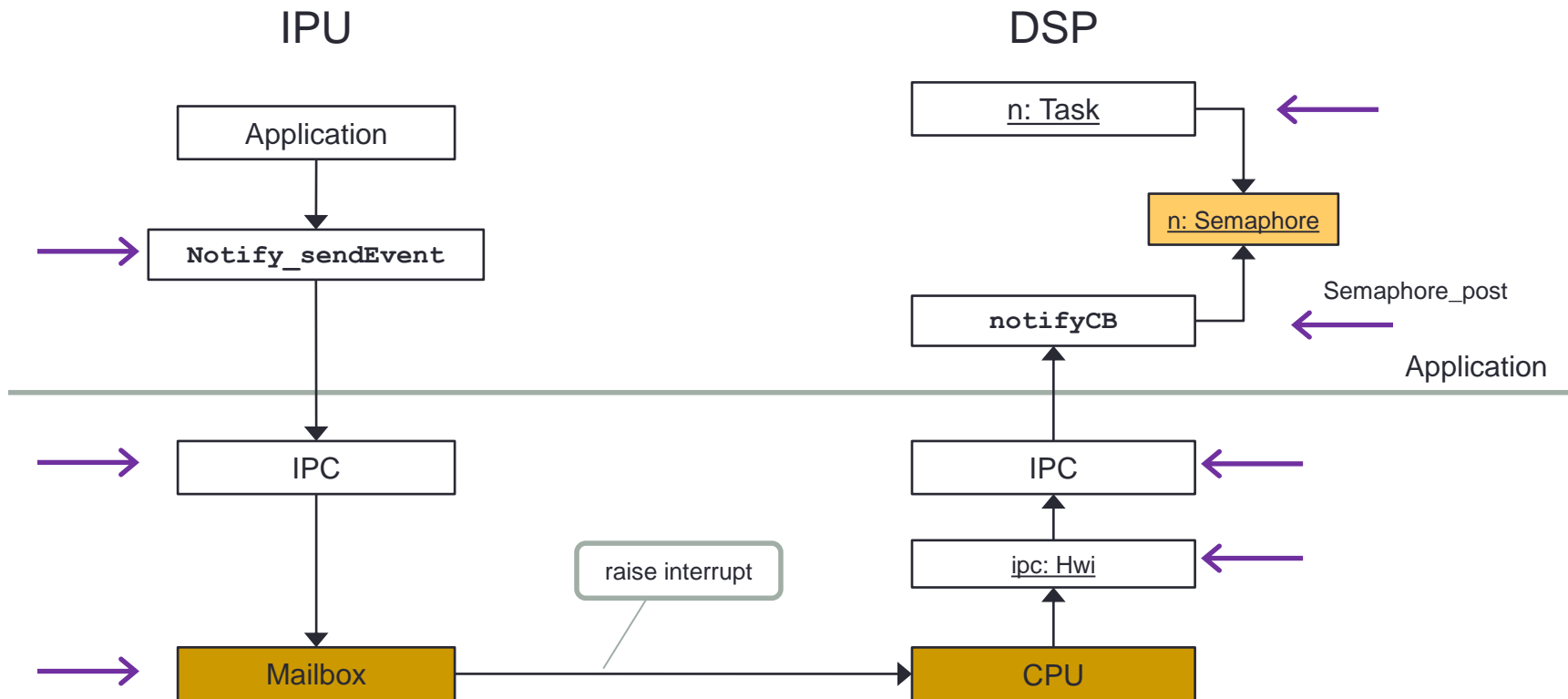
- Notify – send and receive event notifications
 - Inter-processor notifications
 - Multiplex 32 events using single interrupt line
 - Some events are used by IPC
 - Notification is point-to-point
- Callback functions
 - Register for a specific proclid + lineId + eventId triplet
 - Callback can be reused (use proclid and lineId to de-multiplex)
 - Callbacks can be chained (all callbacks are invoked)
 - Callback function receives proclid, eventId, arg, payload
- API Summary
 - `Notify_sendEvent` – raise an event
 - `Notify_registerEvent` – register a callback for an event
 - `Notify_registerEventSingle` – register for exclusive use of event
 - `Notify_FnNotifyCbck` – callback function type definition

Notify

- IPU → DSP
 - IPU sends a notification to the DSP
- IPU processor will...
 - Send a NOP event to test the connection (optional)
 - Send periodic event to the DSP
- DSP processor will...
 - Create a semaphore to synchronize between callback and task
 - Register for event notification
 - Notify callback will post the semaphore
 - Task will run

Notify

- IPU → DSP



Notify

IPU

```
#include <xdc/std.h>
#include <ti/ipc/MultiProc.h>
#include <ti/ipc/Notify.h>

#define EVT 12
#define NOP 0

UInt16 dsp = MultiProc_getId("DSP");
UInt32 payload;

do {
    s = Notify_sendEvent(dsp, 0, EVT, NOP, TRUE);

    if (s == Notify_E_EVTNOTREGISTERED) {
        Task_sleep(1);
    }
} while (s == Notify_E_EVTNOTREGISTERED);

do {
    /* work */
    payload = ...;
    Notify_sendEvent(dsp, 0, EVT, payload, TRUE);
} until(done);
```

Notify

DSP

```
#include <xdc/std.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/ipc/MultiProc.h>
#include <ti/ipc/Notify.h>

#define EVT 12
#define NOP 0

UInt16 IPU = MultiProc_getId("IPU");

Semaphore_Handle sem;
sem = Semaphore_create(0, NULL, NULL);

Notify_registerEvent(IPU, 0, EVT, notifyCB, (UArg)sem);

do {
    Semaphore_pend(sem, BIOS_WAIT_FOREVER);
    /* work */
} until(done);
```

```
Void notifyCB(UInt16 procId, UInt16 lineId,
              UInt32 eventId, UArg arg, UInt32 payload)
{
    Semaphore_Handle sem = (Semaphore_Handle)arg;

    if (payload != NOP) {
        Semaphore_post(sem);
    }
}
```

Notify Module

- ROV screen shot

The screenshot shows the RTOS Object View (ROV) window with the 'Notify' module selected in the left-hand tree. The right-hand pane displays the 'Basic' tab of the configuration table.

address	remoteProcId	remoteProcName	lineId	disabled
0x8081b710	1	CORE1	0	0
0x8081b8f0	0	CORE0	0	0
0x8081bb80	2	CORE2	0	0

The screenshot shows the RTOS Object View (ROV) window with the 'Notify' module selected in the left-hand tree. The right-hand pane displays the 'EventListeners' tab of the configuration table.

eventId	fnNotifyCbck	cbckArg
2	ti_sdo_ipc_transports_TransportShm_notifyFxn_I	0x8081ba8c
4	ti_sdo_ipc_nsremote_NameServerRemoteNotify_cbFxn_I	0x8081b9b0

MultiProc Module

- MultiProc – processor identification
 - Stores processor ID of all processors in the multi-core application
 - Stores processor name
- Processor ID is a number from 0 – (n-1)
- Processor name is defined by IPC
 - IPC Release Notes > Package Reference Guide > ti.sdo.utils.MultiProc > Configuration Settings > MultiProc.setConfig
 - Click on Table of Valid Names for Each Device
- API Summary
 - [MultiProc_getSelf](#) – return your own processor ID
 - [MultiProc_getId](#) – return processor ID for given name
 - [MultiProc_getName](#) – return processor name

SharedRegion Module

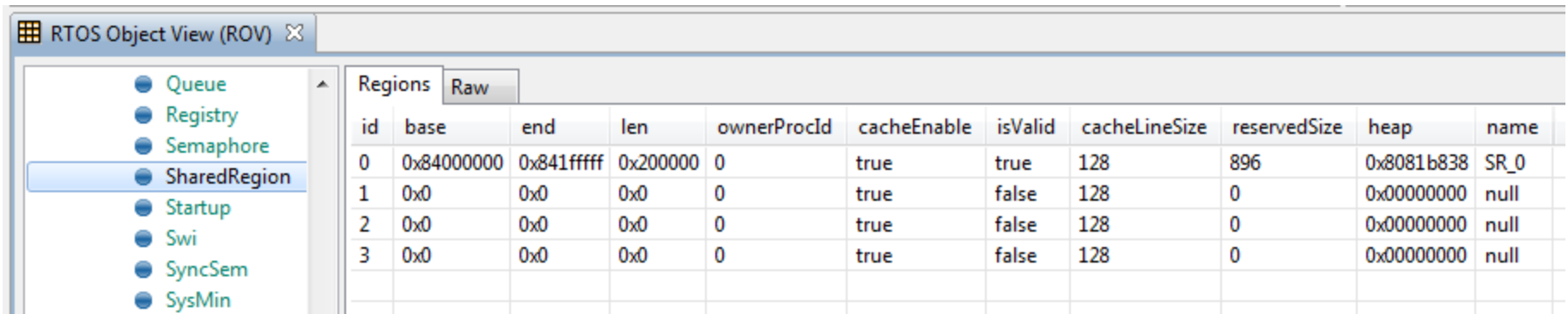
- SharedRegion – shared memory address translation
 - Manages shared memory and its cache configuration
 - Manages shared memory using a memory allocator
 - `SRPtr` is a portable pointer type
- Multiple shared regions are supported
- Each shared region has optional HeapMemMP instance
 - Memory is allocated and freed using this HeapMemMP instance
 - HeapMemMP_create/open and managed internally at IPC initialization
 - `SharedRegion_getHeap` API to get this heap handle

SharedRegion Module

- SharedRegion #0 is special
 - Always contains a heap
 - Contains global state that all cores need to access (reserved header)
 - Must be placed in memory that is accessible by all BIOS cores in the system
- API Summary
 - `SharedRegion_getHeap` – return the heap handle
 - `SharedRegion_getId` – return region ID for given address
 - `SharedRegion_getPtr` – translate SRPtr into local address
 - `SharedRegion_getSRPtr` – translate local address into SRPtr

SharedRegion Module

- ROV screen shot

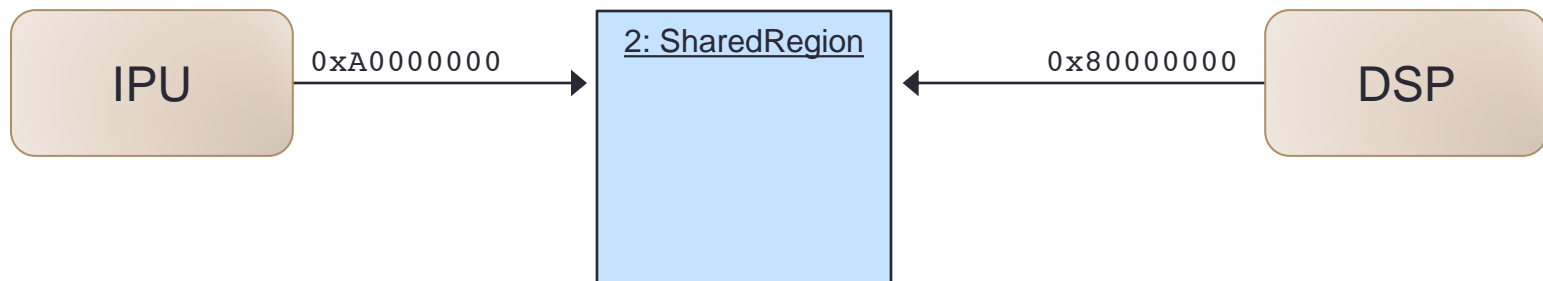


The screenshot shows the RTOS Object View (ROV) window. On the left, a tree view lists various RTOS objects: Queue, Registry, Semaphore, SharedRegion (selected), Startup, Swi, SyncSem, and SysMin. The main area displays a table of SharedRegion objects.

Regions											
Raw											
id	base	end	len	ownerProcId	cacheEnable	isValid	cacheLineSize	reservedSize	heap	name	
0	0x84000000	0x841fffff	0x200000	0	true	true	128	896	0x8081b838	SR_0	
1	0x0	0x0	0x0	0	true	false	128	0	0x00000000	null	
2	0x0	0x0	0x0	0	true	false	128	0	0x00000000	null	
3	0x0	0x0	0x0	0	true	false	128	0	0x00000000	null	

SharedRegion

- Sometimes, shared memory has different address on different processors.
 - Use SharedRegion to translate addresses.



SharedRegion

ipu.cfg

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
var sr2 = new SharedRegion.Entry(
{
    name:          "SR_2",
    base:          0xA0000000,
    len:           0x400000,
    ownerProcId:   0,
    isValid:       true,
    cacheEnable:   false
});
SharedRegion.setEntryMeta(2, sr2);
```

dsp.cfg

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
var sr2 = new SharedRegion.Entry(
{
    name:          "SR_2",
    base:          0x80000000,
    len:           0x400000,
    ownerProcId:   0,
    isValid:       true,
    cacheEnable:   false
});
SharedRegion.setEntryMeta(2, sr2);
```

SharedRegion

IPU

```
#include <xdc/std.h>
#include <ti/ipc/SharedRegion.h>

Ptr ptr = 0xA0004000;
SRPtr srptr = SharedRegion_getSRPtr(ptr, 2);

/* Write SRPtr into message payload
 * and send message to DSP.
 */
msg->srptr = srptr;
MessageQ_put(q, msg);
```

DSP

```
#include <xdc/std.h>
#include <ti/ipc/SharedRegion.h>

/* Receive message, translate
 * embedded SRPtr into local pointer.
 */
MessageQ_get(q, &msg);

SRPtr srptr = msg->srptr;
Ptr ptr = SharedRegion_getPtr(srptr);

/* ptr = 0x80004000 */
```

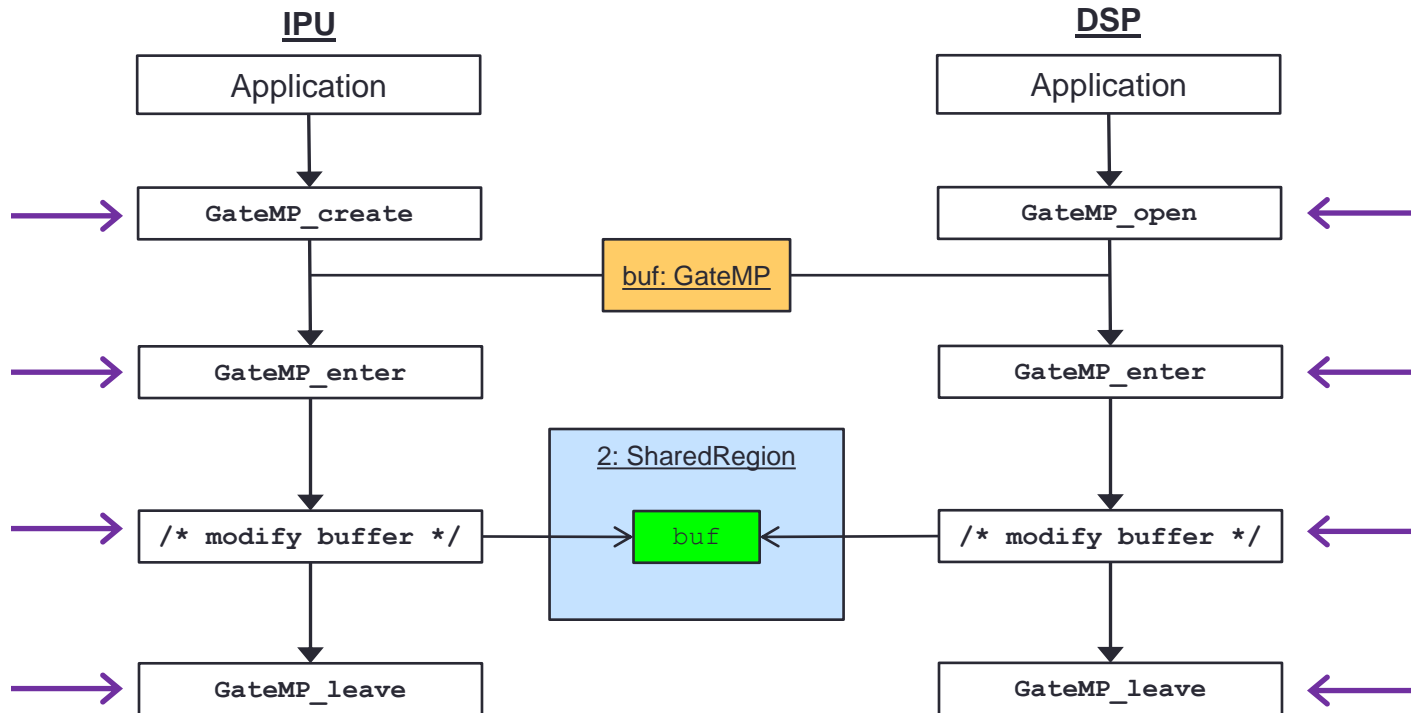
GateMP Module

- GateMP – protect a critical section
 - Multiple processor gate that provides context protection against threads on both local and remote processors
- API Summary
 - `GateMP_create` – create a new instance
 - `GateMP_open` – open an existing instance
 - `GateMP_enter` – acquire the gate
 - `GateMP_leave` – release the gate

GateMP

- Use GateMP instance to protect buffer from concurrent access
- IPU processor will...
 - create a GateMP instance
 - enter the gate
 - modify shared memory
 - leave the gate
- DSP processor will...
 - open the GateMP instance
 - enter the gate
 - modify shared memory
 - leave the gate

GateMP



GateMP Module

IPU

```
#include <xdc/std.h>
#include <ti/ipc/GateMP.h>

GateMP_Params params;
GateMP_Handle gate;

GateMP_Params_init(&params);
params.name = "BufGate";
params.localProtect = GateMP_LocalProtect_NONE;
params.remoteProtect = GateMP_RemoteProtect_SYSTEM;

gate = GateMP_create(params);

GateMP_enter(gate);
/* modify buffer */
GateMP_leave(gate);
```

DSP

```
#include <xdc/std.h>
#include <ti/ipc/GateMP.h>

GateMP_Handle gate;

GateMP_open("BufGate", &gate);

GateMP_enter(gate);
/* modify buffer */
GateMP_leave(gate);
```


HeapMemMP HeapBufMP Modules

- HeapMemMP, HeapBufMP – multi-processor memory allocator
 - Shared memory allocators that can be used by multiple processors
 - HeapMemMP – variable size allocations
 - HeapBufMP – fixed size allocations, deterministic, ideal for MessageQ
- IPC's versions of HeapBuf, adds GateMP and cache coherency to the version provided by SYS/BIOS.
- All allocations are aligned on cache line size.
 - Warning: Small allocations will occupy full cache line.
- Uses GateMP to protect shared state across cores.
- HeapBufMP. All buffers are same size (per instance)
- Every SharedRegion uses a HeapMemMP instance to manage the shared memory

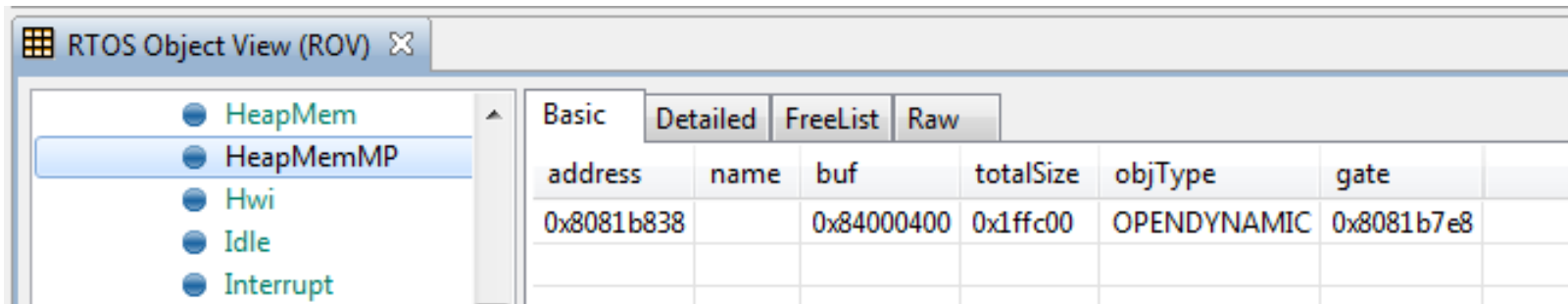
HeapMemMP HeapBufMP Modules

- API Summary

- `HeapMemMP_create` – create a heap instance
- `HeapMemMP_delete` – delete a heap instance
- `HeapMemMP_open` – open a heap instance
- `HeapMemMP_close` – close a heap instance
- `HeapMemMP_alloc` – allocate a block of memory
- `HeapMemMP_free` – return a block of memory to the pool
- `HeapBufMP_create` – create a heap instance
- `HeapBufMP_delete` – delete a heap instance
- `HeapBufMP_open` – open a heap instance
- `HeapBufMP_close` – close a heap instance
- `HeapBufMP_alloc` – allocate a block of memory
- `HeapBufMP_free` – return a block of memory to the pool

HeapMemMP Module

- ROV screen shot



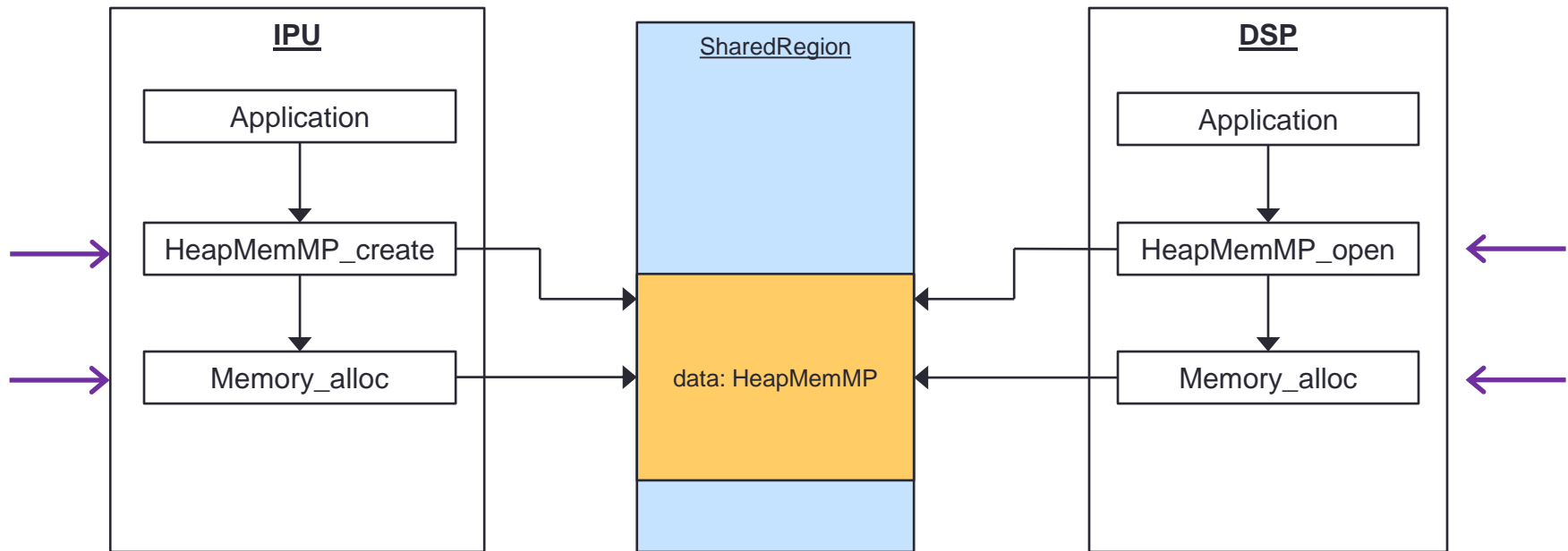
The screenshot shows the RTOS Object View (ROV) window. On the left, a tree view lists several modules: HeapMem, HeapMemMP (selected), Hwi, Idle, and Interrupt. The main area displays the details for the selected module, HeapMemMP, in a table format. The table has columns for address, name, buf, totalSize, objType, and gate. The data row shows: address 0x8081b838, name (empty), buf 0x84000400, totalSize 0x1ffc00, objType OPENDYNAMIC, and gate 0x8081b7e8.

address	name	buf	totalSize	objType	gate
0x8081b838		0x84000400	0x1ffc00	OPENDYNAMIC	0x8081b7e8

HeapMemMP

- Create a heap and share it between IPU and DSP
- IPU processor will...
 - Creates a heap instance
 - Use Memory module to allocate memory
- DSP processor will...
 - Open the heap instance
 - Use Memory module to allocate memory

HeapMemMP Module



HeapMemMP Module

- Casting the heap handle is one of the few places you need to call an IPC Package API.

IPU

```
#include <xdc/std.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/Memory.h>
#include <ti/ipc/HeapMemMP.h>
#include <ti/sdo/ipc/heaps/HeapMemMP.h>

HeapMemMP_Params params;
HeapMemMP_Params_init(&params);
params.name = "DataHeap";
params.regionId = 0;
params.sharedBufSize = 0x100000;

HeapMemMP_Handle handle;
handle = HeapMemMP_create(&params);

IHeap_Handle heap;
heap = HeapMemMP_Handle_upCast(handle);

ptr = Memory_alloc(heap, size, align, NULL);
```

DSP

```
#include <xdc/std.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/Memory.h>
#include <ti/ipc/HeapMemMP.h>
#include <ti/sdo/ipc/heaps/HeapMemMP.h>

HeapMemMP_Handle handle;
HeapMemMP_open("DataHeap", &handle);

IHeap_Handle heap;
heap = HeapMemMP_Handle_upCast(handle);

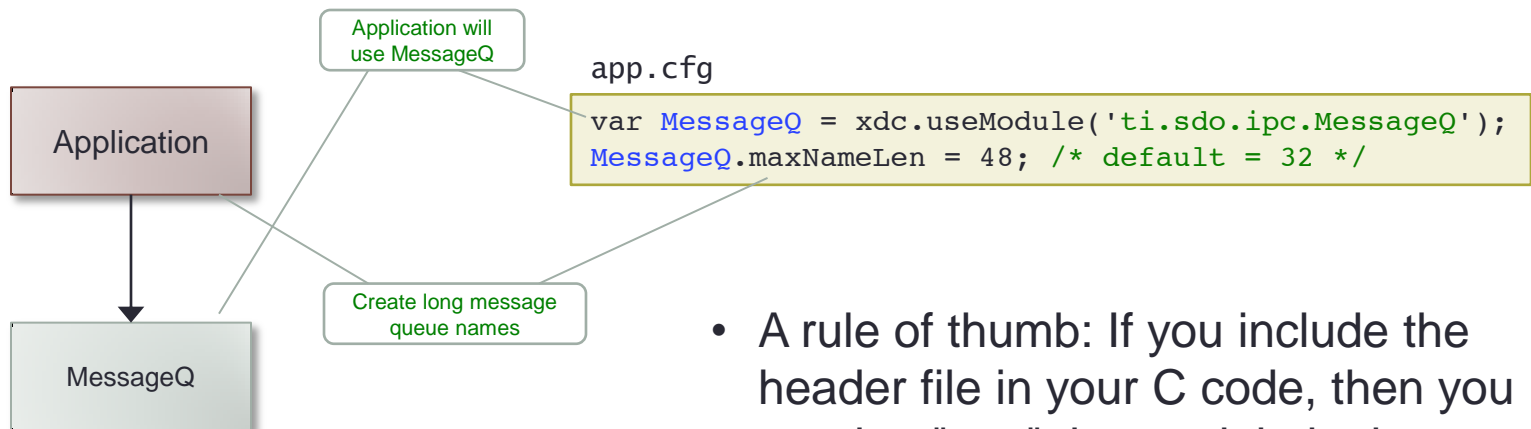
ptr = Memory_alloc(heap, size, align, NULL);
```

Agenda

- Overview
- IPC Modules
- **CONFIGURATION**
- Scalability
- Optimization
- Footnotes

IPC Configuration

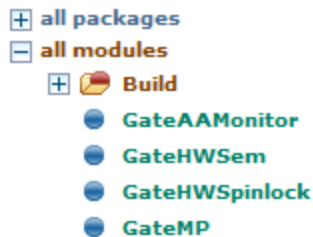
- The RTSC configuration phase is when components are integrated. Each component uses its configuration parameters to express its system requirement.
- The application configuration script is the starting point. It defines which components are needed by the application and configures those components depending on its needs.



- A rule of thumb: If you include the header file in your C code, then you need to "use" the module in the configuration script.

IPC Configuration - documentation

- Navigate to the IPC product folder
 - `C:\Products\ipc_3_30_pp_bb`
- Open the release notes
 - `ipc_3_30_pp_bb_release_notes.html`
- Scroll down to the documentation section
 - Click on [Package Reference Guide \(cdoc\)](#)
 - Tip: the Package Reference Guide is also available in CCS Help.
 - Tip: the Package Reference Guide is also available [on-line](#).
- Open 'all modules' section



- Click on a module (e.g. MessageQ)
- Click on Configuration settings link

`module ti.sdo.ipc.MessageQ`

Message-passing with queuing

- ↓ C synopsis
- Individual elements
- ↓ Configuration settings
- Individual elements

IPC Configuration - documentation

- The documentation shows how to include the module in the application configuration script.
- Scroll down to the '**module-wide config parameters**' for a list of all configuration parameters.

Configuration settings

sourced in ti/sdo/ipc/MessageQ.xdc

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

local proxy modules

```
MessageQ.SetupTransportProxy = ITransportSetup.Module null  
MessageQ.SetupTransportProxy.delegate$ = ITransportSetup.Module null
```

module-wide constants & types

```
const MessageQ.ANY = ~(0);  
const MessageQ.HIGHPRI = 1;  
const MessageQ.NORMALPRI = 0;  
const MessageQ.RESERVEDPRI = 2;  
const MessageQ.URGENTPRI = 3;
```

module-wide config parameters

```
MessageQ.A_cannotFreeStaticMsg = Assert.Desc {  
    msg: "A_cannotFreeStaticMsg: Cannot call MessageQ_free with static msg"  
};
```

IPC Configuration - documentation

- When a config param has a default value, it will be indicated after the type.
- If the config param does not have a default value, this is indicated by the keyword *undefined*. Sometimes default values are computed during the configuration phase.

```
//  
MessageQ.freeHookFxn = Void(*) (Bits16,Bits16) null;  
MessageQ.maxNameLen = UInt 32;  
MessageQ.maxRuntimeEntries = UInt NameServer.ALLOWGROWTH;  
MessageQ.nameTableGate = IGateProvider.Handle null;  
MessageQ.numHeaps = UInt16 8;  
MessageQ.numReservedEntries = UInt 0;  
MessageQ.traceFlag = Bool false,  
  
MessageQ.common$ = Types.Common$ undefined;  
MessageQ.tableSection = String null;
```

- Scroll down to the 'module-wide functions' section. Sometimes you will need to use these to set a config param.

module-wide functions

```
Ipcc.addUserFxn(Ipcc.UserFxn fxn, UArg arg) returns Void  
Ipcc.setEntryMeta(Ipcc.Entry entry) returns Void
```

IPC Configuration - documentation

- Instance Configuration Parameters
 - Some config params are specified when creating an instance. These are listed in the '[per-instance config parameters](#)' section.

per-instance config parameters

```
var params = new NameServer.Params;
params.checkExisting = Bool true;
params.maxNameLen = UInt 16;
params.maxRuntimeEntries = UInt NameServer.ALLOWGROWTH;
params.maxValueLen = UInt 0;
```

- However, IPC modules only support instance creation at run-time. You will need to find the equivalent create parameter in the IPC API Reference Guide.
 - Open the release notes
 - [ipc_3_30_pp_bb_release_notes.html](#)
 - Scroll down to the documentation section.
 - Click on [IPC Application Programming Interface \(API\) Reference Guide \(HTML\)](#)
 - Tip: the IPC API Reference Guide is also available in CCS Help.
 - Tip: the IPC API Reference Guide is also available [on-line](#).

Ipc Module Configuration

- IPC configuration requires the following modules
 - `ti.sdo.ipc.Ipc`
 - `ti.sdo.utils.MultiProc`
 - `ti.sdo.ipc.SharedRegion`
- Define Ipc startup protocol
 - `Ipc.procSync` – controls attach behavior
 - `Ipc.ProcSync_ALL` – Attach to all processors simultaneously.
 - `Ipc.ProcSync_PAIR` – Attach to remote processor one-by-one.
- All processors must use the same startup protocol.

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

Ipc Module Configuration

- SharedRegion #0 Memory Setup
 - On some systems, the SR_0 memory may not be available at boot time. Host processor might map the memory into the slaves MMU. This configuration flag is used to block the slave until the memory is available. `ipc_start` will spin until this flag is set true by host.
 - `Ipc.sr0MemorySetup = true;`
 - `Ipc_start` will access SR_0 memory immediately.
 - `Ipc.sr0MemorySetup = false;`
 - `Ipc_start` will spin until host sets flag to true. Requires symbol address access from host.

Ipc Module Configuration

- Attach and detach hooks
 - You can register hook functions to be called during each attach and detach call. Use the hook function to perform application specific tasks.

```
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');  
var fxn = new Ipc.UserFxn;  
fxn.attach = '&userAttachFxn';  
fxn.detach = '&userDetachFxn';  
Ipc.addUserFxn(fxn, arg);
```

- The hook functions have the following type definitions.

```
Int (*attach)(UArg arg, UInt16 procId);  
Int (*detach)(UArg arg, UInt16 procId);
```

MultiProc Configuration

- Define the processors in the IPC application.
 - This example defines three processors: CORE0, CORE1, CORE2.
 - Name order defines MultiProc ID (zero based counting number)

- CORE0 configuration

```
var procNameAry = ["CORE0", "CORE1", "CORE2" ];  
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');  
MultiProc.setConfig("CORE0", procNameAry);
```

- CORE1 configuration

```
var procNameAry = ["CORE0", "CORE1", "CORE2" ];  
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');  
MultiProc.setConfig("CORE1", procNameAry);
```

- CORE2 configuration

```
var procNameAry = ["CORE0", "CORE1", "CORE2" ];  
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');  
MultiProc.setConfig("CORE2", procNameAry);
```


SharedRegion Configuration

- Define number of shared regions in the system.
 - This config param must be the same across all processors in the system. Increasing the number of regions reduces the maximum size of each region.

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');  
SharedRegion.numEntries = 8;
```

SharedRegion Configuration

- Define number of shared regions in the system.
 - This config param must be the same across all processors in the system. Increasing the number of regions reduces the maximum size of each region.

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');  
SharedRegion.numEntries = 8;
```

- Cache Line Size
 - This value is used to align items on a cache line boundary. For example, memory allocations from the shared region heap will be aligned and sized on this boundary. It must be the same value for all processors using the shared region. It must be the worst case value

```
SharedRegion.cacheLineSize = 128;
```

SharedRegion Configuration

- Define SharedRegion #0
 - The shared region base and size are defined in the platform memory map.

config.bld

```
Build.platformTable["ti.platforms.evm6678:core0"] = {  
    ...  
    externalMemoryMap: [  
        ["SR_0", {  
            name: "SR_0", space: "data", access: "RW",  
            base: 0x84000000, len: 0x200000,  
            comment: "SR#0 Memory (2 MB)"  
        }],  
    ],
```

- Reference the memory map from platform to configure SR_0

app.cfg

```
var SR0Mem = Program.cpu.memoryMap["SR_0"];  
SharedRegion.setEntryMeta(0,  
    new SharedRegion.Entry({  
        name:          "SR_0",  
        base:          SR0Mem.base,  
        len:           SR0Mem.len,  
        ownerProcId:  0,  
        isValid:      true,  
        cacheEnable:  true  
    })  
);
```

SharedRegion Configuration

- Cache setting for a shared region.
 - Reports memory cache setting
 - This config param does not control the cache behavior, it reflects the cache behavior. In other words, if the shared memory is eligible for caching, then this parameter must be set true.
 - Controls IPC cache operations
 - When set to true, IPC will perform the necessary cache operations.
 - Processor relative
 - It may be different on each processor using the same shared region. This comes about because each processor defines its cache behavior.

Build Configuration

- Build module used to configure library type.

```
var Build = xdc.useModule('ti.sdo.ipc.Build');  
Build.LibType = Build.LibType_NonInstrumented;
```

- Build.LibType

- `Build.LibType_Instrumented` – Prebuild library supplied in IPC product. Optimized with logging and asserts enabled.
- `Build.libType_NonInstrumented` – Prebuilt library supplied in IPC product. Optimized without instrumentation.
- `Build.libType_Custom` – Rebuilds IPC libraries from source for each executable. Optimized by default. Use `Build.customCCOpts` to modify compiler options.
- `Build.libType_Debug` – Rebuilds IPC libraries from source for each executable. Non-optimized, useful for debugging IPC sources.

Build Configuration

- Build module used to configure library type.

```
var Build = xdc.useModule('ti.sdo.ipc.Build');  
Build.LibType = Build.LibType_NonInstrumented;
```

- Build.LibType

- `Build.LibType_Instrumented` – Prebuilt library supplied in IPC product. Optimized with logging and asserts enabled.
- `Build.libType_NonInstrumented` – Prebuilt library supplied in IPC product. Optimized without instrumentation.
- `Build.libType_Custom` – Rebuilds IPC libraries from source for each executable. Optimized by default. Use `Build.customCCOpts` to modify compiler options.
- `Build.libType_Debug` – Rebuilds IPC libraries from source for each executable. Non-optimized, useful for debugging IPC sources.
- `Build.libType_PkgLib` – Links with package libraries in the IPC product. These libraries do not ship with the product. You must rebuild the IPC product to generate the package libraries. Useful for development and sharing custom libraries.

MessageQ Configuration

- Message Queue Name Length

- The maximum length of a message queue name is set at configuration time.

```
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');  
MessageQ.maxNameLen = 48;
```

- Number of message queue heaps

- The MessageQ module maintains a table of registered heap handles. The size of this table is set at configuration time. The heapId field in the message queue header is used to index into this table. The heapId must be system wide unique. For example, if ProcA and ProcB share a heap, it might be registered with heapId = 0. If ProcC and ProcD share a different heap, their registered heapId cannot be 0. Keep this in mind when configuring the size of the heap table.

```
MessageQ.numHeaps = 12;
```

Notify Configuration

- Number of events supported by Notify
 - By default, the Notify module will support the maximum number of possible events. You can reduce this to conserve on memory footprint.

```
var Notify = xdc.useModule('ti.sdo.ipc.Notify');  
Notify.numEvents = 16;
```

- Number of reserved events
 - Use this config param to reserve events for middleware modules. IPC already reserves some number, so do not reduce this value. You can increase the value to reserve additional events for your middleware.

```
Notify.reservedEvents = 8;
```


Notify Configuration

- Number of events supported by Notify
 - By default, the Notify module will support the maximum number of possible events. You can reduce this to conserve on memory footprint.

```
var Notify = xdc.useModule('ti.sdo.ipc.Notify');  
Notify.numEvents = 16;
```

GateMP Configuration

- Maximum Name Length
 - The maximum name length for a GateMP instance is controlled by this config param. Although this is a module-wide config parameter, it is used by the GateMP module when creating its private NameServer instance.

```
var GateMP = xdc.useModule('ti.sdo.ipc.GateMP');  
GateMP.maxNameLen = 48;
```

- Device-specific gate delegates offer hardware locking to GateMP
 - GateHWSpinlock for OMAP4, OMAP5, TI81XX, Vayu
 - GateHWSem for C6474, C66x
 - GateAAMonitor for C6472
 - GatePeterson, GatePetersonN for devices that don't have HW locks

GateMP Configuration

- GateMP has three proxies

- `RemoteSystemProxy`
 - `RemoteCustom1Proxy`
 - `RemoteCustom2Proxy`

- Each proxy is assigned a delegate module. The delegate is the one which actually implements the gate. Typically, the system delegate will use hardware support if available and the custom delegates are software implementations.

- ```
GateMP.RemoteSystemProxy = xdc.useModule('ti.sdo.ipc.gates.GateHWSpinlock');
GateMP.RemoteCustom1Proxy = xdc.useModule('ti.sdo.ipc.gates.GatePeterson');
GateMP.RemoteCustom2Proxy = xdc.useModule('ti.sdo.ipc.gates.GateMPSupportNull');
```

- Note: GatePeterson works for only two clients. Use GatePetersonN for three or more clients.

# GateMP Configuration

- The hardware spinlocks are reused as GateMP instances are deleted and re-created.
- When creating a GateMP instance, the remoteProtect create parameter specifies which proxy to use.

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

```
GateMP_Params params;
```

```
GateM_Handle gate;
```

```
GateMP_Params_init(¶ms);
```

```
params.name = "BufGate";
```

```
params.remoteProtect = GateMP_RemoteProtect_CUSTOM1;
```

```
gate = GateMP_create(¶ms);
```

# GateMP Configuration

- On Vayu, GateHWSpinlock is the default remote system delegate. It has a config parameter for specifying the base address of the hardware spin locks. It does not have a default value because it is set internally based on the device. You can override the default by setting it in your config script.

```
var GateHWSpinlock = xdc.useModule('ti.sdo.ipc.gates.GateHWSpinlock');
GateHWSpinlock.baseAddr = 0x4A0F6800;
```

- The number of hardware spinlocks to use is set by an internal config parameter (look at the xdc file to see this). This is also set internally based on the device. However, it is possible to override this in your config script. Warning: this is not a common practice.

```
GateHWSpinlock.numLocks = 64;
```

# Agenda

- Overview
- IPC Modules
- Configuration
- **SCALABILITY**
- Optimization
- Footnotes

# Scalability

- IPC scalability allows you to include as little or as many modules as you need.
- Scalability allows you to manage the IPC footprint (data and code) contributed to your executable.
- Scalability options
  - Utilities only
  - Notify only
  - IPC full

# Scalability – Utilities only

- Application provides its own IPC framework. The [utilities](#) package provides foundational support.
- The application uses only the MultiProc module. Do not use the Ipc module.
- To use the NameServer module, application must provide an INameServerRemote implementation. (Advanced topic.)

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

```
ti.sdo.utils.INameServerRemote
```



# Scalability – Notify only

- At this scalability level, the Notify and MultiProc modules are the only IPC modules used by the application.
- Do not use the Ipc module. No calls to Ipc\_start or Ipc\_attach.
- Call Notify\_attach per transport to enable notify. After this call, the processor is able to receive notify events.
- If needed, the application is responsible to "handshake" between sender and listener.
- Only supported with notify mailbox driver.

# Scalability – Notify only

- Notify only configuration

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

- Notify attach example

```
#include <ti/sdo/ipc/Notify.h>
#include <ti/sdo/utils/MultiProc.h>
```

```
Int procId;
```

```
procId = MultiProc_getId("EVE1");
Notify_attach(procId, 0);
```

# Scalability – IPC full

- This is the default scalability level
- Must configure the following modules
  - `ti.sdo.ipc.Ipc`
  - `ti.sdo.ipc.SharedRegion`
  - `ti.sdo.utils.MultiProc`
- Application is entitled to use all IPC modules.

# Lab – ex13\_notifypeer

- Please open the PowerPoint slide named IPC\_Lab\_3\_Sclability

# Agenda

- Overview
- IPC Modules
- Configuration
- Scalability
- **OPTIMIZATION**
- Footnotes

# IPC Optimization – wiki page

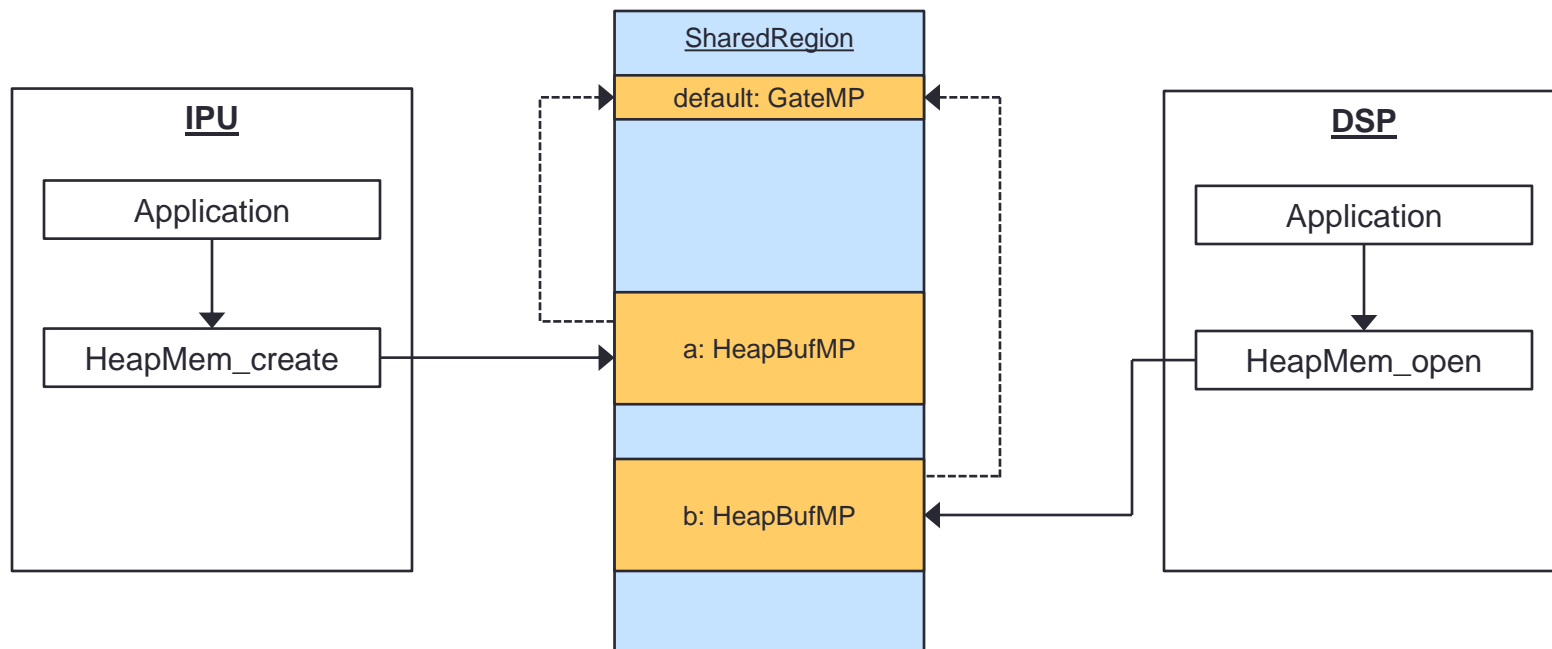
- Through configuration parameters, IPC can be optimized for a given application. There is a good wiki topic on this.
  - [http://processors.wiki.ti.com/index.php/IPC\\_Users\\_Guide/Optimizing\\_IPC\\_Applications](http://processors.wiki.ti.com/index.php/IPC_Users_Guide/Optimizing_IPC_Applications)
- Using dedicated GateMP instances can reduce runtime contention.
- There are a few transports available. They have different restrictions and runtime performance.

# IPC Optimization – heap + gate

- Heaps will use the default gate. When creating two independent heap instances, there would be unnecessary contention for the same gate. To reduce contention for this gate, use a dedicated GateMP instance for your heap.
  - Create the GateMP instance
  - Assign gate instance in heap create parameter
  - Create the heap
  - The dedicated gate is used automatically

# IPC Optimization – heap + gate

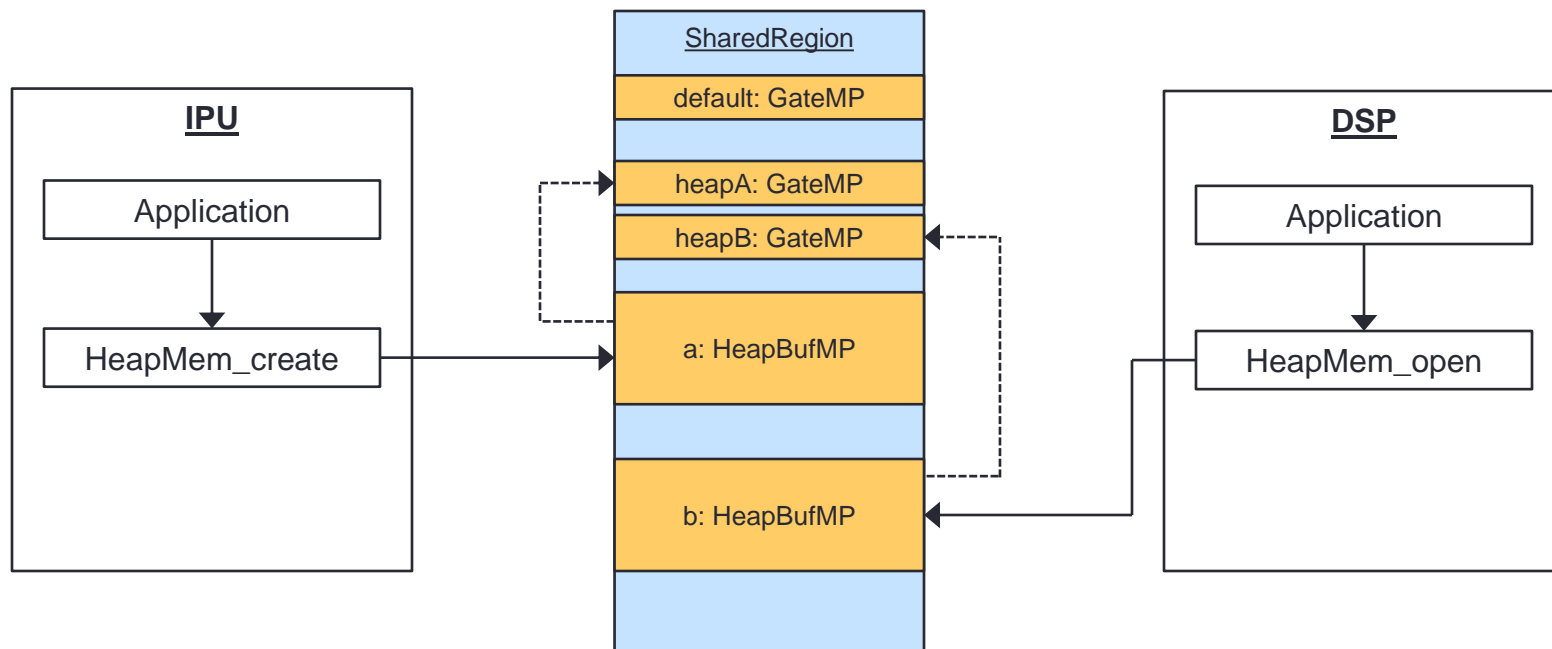
- When using the default gate, two independent heaps will contend for the same gate.





# IPC Optimization – heap + gate

- When using a dedicated gate for each heap, there is no contention for the gate (between the heaps).



# IPC Optimization – heap + gate

- This example creates a heap to be used by MessageQ. A dedicated GateMP instance is used for the heap.

```
#include <xdc/std.h>
#include <ti/ipc/GateMP.h>
#include <ti/ipc/HeapBufMP.h>

HeapBufMP params;
HeapBufMP_Params_init(¶ms);
params.name = "HeapA";
params.blockSize = 128;
params.numBlocks = 32;
params.gate = GateMP_create(NULL);

HeapBufMP_Handle heap;
heap = HeapBufMP_create(params);
MessageQ_registerHeap(0, heap);
```

No need to specify a name for the gate. The default protection will be used.

# IPC Optimization – message transport

- The MessageQ module uses a transport for actual message delivery. There are three available transports, each with different runtime performance characteristics.
  - [TransportShm](#) – slowest, largest data footprint, most robust (default)
  - [TransportShmCirc](#) – medium, fixed length transport buffer
  - [TransportShmNotify](#) – fastest, may cause sender to busy wait
- Each transport has a corresponding setup module. To use a given transport, configure the MessageQ module with the transport's setup module.
  - [TransportShmSetup](#) – the setup module
  - [TransportShmCircSetup](#) – the setup module
  - [TransportShmNotifySetup](#) – the setup module
- Message transport are in the following folder.
  - [ipc\\_3\\_xx\\_pp\\_bb/packages/ti/sdo/ipc/transports](#)
- All MessageQ modules must be configured to use the same transport.

# IPC Optimization – message transport

- Example configuration

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

# IPC Optimization – notify driver

- The Notify module uses a low-level driver to implement the actual signaling between processors. Each driver has a corresponding setup module. To use a given driver, configure the Notify module with the driver's setup module.

- Generic (i.e. software) notify drivers are in the following folder.

```
ipc_3_xx_pp_bb/packages/ti/sdo/ipc/notifyDrivers
```

```
NotifyDriverShm – (default)
```

```
NotifyDriverCirc
```

- Device specific (i.e. hardware) notify drivers are in the family folder.

```
ipc_3_xx_pp_bb/packages/ti/sdo/ipc/family
```

```
ti81xx/NotifyDriverMbx
```

```
vayu/NotifyDriverMbx
```

- All Notify modules must be configured to use the same driver.
- Notify driver list is constantly changing as we add new device support.

# IPC Optimization – notify driver

- The setup modules are always device specific, even when using a generic driver. Look in the family folder to see which setup modules are available for your device.

```
ipc_3_xx_pp_bb/packages/ti/sdo/ipc/family
```

- Here is an example for the C647x device.

```
ipc_3_xx_pp_bb/packages/ti/sdo/ipc/family/c647x
```

```
NotifySetup.xdc – (default)
```

```
NotifyCircSetup.xdc
```

- Here is an example for ti81xx device.

```
ipc_3_xx_pp_bb/packages/ti/sdo/ipc/family/ti81xx
```

```
NotifySetup.xdc – (default)
```

```
NotifyCircSetup.xdc
```

```
NotifyMbxSetup.xdc
```

# IPC Optimization – notify driver

- Example configuration for C647x

```
var Notify = xdc.useModule('ti.sdo.ipc.Notify');
Notify.SetupProxy = xdc.useModule('ti.sdo.ipc.family.c647x.NotifyMbxSetup');
```

# IPC Optimization – Vayu notify driver

- Vayu notify driver configuration is different!
- There is only one notify setup module for Vayu.  
`ti.sdo.ipc.family.vayu.NotifySetup`
- Available notify drivers on Vayu.  
`ti.sdo.ipc.notifyDrivers.NotifyDriverShm` – (default)  
`ti.sdo.ipc.family.vayu.NotifyDriverMbx`
- Notify driver configuration is specified for each connection.



# IPC Optimization – Vayu notify driver

- Example configuration for Vayu (on DSP1)

```
var NotifySetup = xdc.useModule('ti.sdo.ipc.family.vayu.NotifySetup');
```

```
NotifySetup.connections.$add(
 new NotifySetup.Connection({
 procName: "EVE1",
 driver: NotifySetup.Driver_MAILBOX
 })
);
```

```
NotifySetup.connections.$add(
 new NotifySetup.Connection({
 procName: "DSP2",
 driver: NotifySetup.Driver_SHAREDMEMORY
 })
);
```

# Agenda

- Overview
- IPC Modules
- Configuration
- Scalability
- Optimization
- **FOOTNOTES**

# Footnotes — Examples

- Examples are provided in the following location.
  - [ipc\\_3\\_xx\\_pp\\_bb/examples/<platform>](https://github.com/ti/ipc_3_xx_pp_bb/examples/<platform>)
- Examples are platform and OS-specific. Not all examples are provided for all environments.
  - Makefile-based, demonstrating multicore-friendly build model
  - Developed independent of specific SDKs, so memory maps don't always align.
- Use DRA7xx\_bios\_elf for Vayu platform
  - DRA7xx = platform
  - bios = host operating system (i.e. SYS/BIOS on all processors)
  - elf = ELF tool chain

# Footnotes - Error Handling

- Many of the IPC APIs return an integer as a status code. All error values are negative; all success values are zero or positive. You can use a simple test to check for error.

```
Int status;

status = MessageQ_get(...);

if (status < 0) {
 /* error */
}
```

- Some APIs return a handle. If the handle is NULL, an error has occurred.

```
MessageQ_Handle queue;

queue = MessageQ_create(...);

if (queue == NULL) {
 /* error */
}
```

- IPC status codes come in two groups: success and error. The 'S' and 'E' in the status code tells you if it is success or error.

```
MessageQ_S_ALREADYSETUP
MessageQ_E_NOTFOUND
```

# Footnotes - Online Resources

- IPC Documentation
  - IPC API Reference (Doxygen) – [latest release](#)
  - IPC Configuration Reference (Cdoc) – [latest release](#)
- External wiki articles that will continue to evolve:
  - Overview - [http://processors.wiki.ti.com/index.php/IPC\\_3.x](http://processors.wiki.ti.com/index.php/IPC_3.x)
  - Users Guide - [http://processors.wiki.ti.com/index.php/IPC\\_Users\\_Guide](http://processors.wiki.ti.com/index.php/IPC_Users_Guide)
  - Migration - [http://processors.wiki.ti.com/index.php/IPC\\_3.x\\_Migration\\_Guide](http://processors.wiki.ti.com/index.php/IPC_3.x_Migration_Guide)
- Development Repo/products:
  - Development Repository - <http://git.ti.com/cgi/cgit.cgi/ipc/ipcdev.git/>
  - Development Flow - <http://git.ti.com/ipc/pages/Home>
- Product download
  - [http://software-dl.ti.com/dsps/dsps\\_public\\_sw/sdo\\_sb/targetcontent/ipc/index.html](http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ipc/index.html)

Thank You!