**Cfg文件**

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

/\*

 \* Get the list of names that the build device supports.

 \* I.e. ["CORE0", "CORE1", "CORE2" ... ]

 \*/

**var** nameList = MultiProc.getDeviceProcNames();

/\*

 \* Since this is a single-image example, we don't (at build-time) which

 \* processor we're building for. We therefore supply 'null'

 \* as the local procName and use MultiProc\_setLocalId to set the procId

 \* at runtime.

 \*/

MultiProc.setConfig(**null**, nameList);

**var** System = xdc.useModule('xdc.runtime.System');

**var** SysStd = xdc.useModule('xdc.runtime.SysStd');

System.SupportProxy = SysStd;

/\* Modules explicitly used in the application \*/

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

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

**var** BIOS = xdc.useModule('ti.sysbios.BIOS');

BIOS.heapSize = 0x8000;

**var** Task = xdc.useModule('ti.sysbios.knl.Task');

**var** tsk0 = Task.create('&tsk0\_func');

tsk0.instance.name = "tsk0";

/\* To avoid wasting shared memory for MessageQ transports \*/

**for** (**var** i = 0; i < MultiProc.numProcessors; i++) {

 Ipc.setEntryMeta({

 remoteProcId: i,

 setupMessageQ: **false**,

 });

}

/\* Synchronize all processors (this will be done in Ipc\_start) \*/

Ipc.procSync = Ipc.ProcSync\_ALL;

/\* Shared Memory base address and length \*/

**var** SHAREDMEM = 0x0C000000;

**var** SHAREDMEMSIZE = 0x00200000;

/\*

 \* Need to define the shared region. The IPC modules use this

 \* to make portable pointers. All processors need to add this

 \* call with their base address of the shared memory region.

 \* If the processor cannot access the memory, do not add it.

 \*/

**var** SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');

SharedRegion.setEntryMeta(0,

 { base: SHAREDMEM,

 len: SHAREDMEMSIZE,

 ownerProcId: 0,

 isValid: **true**,

 name: "DDR2\_RAM",

 });

/\* Create a semaphore with count 0 \*/

**var** Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');

Program.global.semHandle = Semaphore.create(0);

/\*

 \* @(#) ti.sdo.ipc.examples.multicore.evm667x; 1, 0, 0, 0,1; 5-22-2012 16:36:06; /db/vtree/library/trees/ipc/ipc-h32/src/ xlibrary

 \*/

IPC.c

**#include** <string.h> /\* for memcpy() \*/

**#include** <xdc/std.h>

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

**#include** <xdc/runtime/Assert.h>

**#include** <xdc/runtime/Memory.h>

**#include** <xdc/runtime/Startup.h>

**#include** <ti/sysbios/BIOS.h>

**#include** <ti/sysbios/hal/Cache.h>

**#include** <ti/sysbios/hal/Hwi.h>

**#include** <ti/sysbios/knl/Task.h>

**#include** <ti/sdo/ipc/\_Ipc.h>

**#include** <ti/sdo/ipc/\_Notify.h>

**#include** <ti/sdo/ipc/\_GateMP.h>

**#include** <ti/sdo/ipc/\_SharedRegion.h>

**#include** <ti/sdo/utils/\_MultiProc.h>

**#include** <ti/sdo/utils/\_NameServer.h>

**#include** <ti/sdo/ipc/\_MessageQ.h>

**#include** "package/internal/Ipc.xdc.h"

**#ifdef** \_\_ti\_\_

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_attach);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_detach);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_isAttached);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_readConfig);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_start);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_stop);

 **#pragma** FUNC\_EXT\_CALLED(Ipc\_writeConfig);

**#endif**

/\*

 \* For no MMU case, it should be set to TRUE always.

 \* For MMU, it would be set by IPC link between local and HOST processors.

 \*/

**extern** \_\_FAR\_\_ Bits32 Ipc\_sr0MemorySetup;

/\*

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Common Header Functions

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \*/

/\*

 \* ======== Ipc\_attach ========

 \*/

Int **Ipc\_attach**(UInt16 remoteProcId)

{

 Int i;

 Ptr sharedAddr;

 SizeT memReq;

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*slave;

 ti\_sdo\_ipc\_Ipc\_ProcEntry \*ipc;

 Error\_Block eb;

 SharedRegion\_Entry entry;

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 Int status;

 UInt hwiKey;

 /\* Assert remoteProcId is in our cluster and isn't our own \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 Assert\_isTrue(remoteProcId != **MultiProc\_self**(),

 ti\_sdo\_ipc\_Ipc\_A\_invArgument);

 /\* Check whether Ipc\_start has been called. If not, fail. \*/

 **if** (Ipc\_module->ipcSharedAddr == NULL) {

 **return** (Ipc\_E\_FAIL);

 }

 /\* for checking and incrementing attached below \*/

 hwiKey = Hwi\_disable();

 /\* Make sure its not already attached \*/

 **if** (Ipc\_module->procEntry[clusterId].attached) {

 Ipc\_module->procEntry[clusterId].attached++;

 /\* restore interrupts and return \*/

 Hwi\_restore(hwiKey);

 **return** (Ipc\_S\_ALREADYSETUP);

 }

 /\* restore interrupts \*/

 Hwi\_restore(hwiKey);

 /\* get region 0 information \*/

 SharedRegion\_getEntry(0, &entry);

 /\* Make sure we've attached to owner of SR0 if we're not owner \*/

 **if** ((**MultiProc\_self**() != entry.ownerProcId) &&

 (remoteProcId != entry.ownerProcId) &&

 !(Ipc\_module->procEntry[ti\_sdo\_utils\_MultiProc\_getClusterId(

 entry.ownerProcId)].attached)) {

 **return** (Ipc\_E\_FAIL);

 }

 /\* Init error block \*/

 Error\_init(&eb);

 /\* determine the slave's slot \*/

 slave = Ipc\_getSlaveAddr(remoteProcId, Ipc\_module->ipcSharedAddr);

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)slave, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 /\* get the attach paramters associated with remoteProcId \*/

 ipc = &(Ipc\_module->procEntry[clusterId]);

 /\* Synchronize the processors. \*/

 status = Ipc\_procSyncStart(remoteProcId, Ipc\_module->ipcSharedAddr);

 **if** (status < 0) {

 **return** (status);

 }

 /\* must be called before SharedRegion\_attach \*/

 status = ti\_sdo\_ipc\_GateMP\_attach(remoteProcId,

 Ipc\_module->gateMPSharedAddr);

 **if** (status < 0) {

 **return** (status);

 }

 /\* retrieves the SharedRegion Heap handles \*/

 status = ti\_sdo\_ipc\_SharedRegion\_attach(remoteProcId);

 **if** (status < 0) {

 **return** (status);

 }

 /\* attach Notify if not yet attached and specified to set internal setup \*/

 **if** (!(Notify\_intLineRegistered(remoteProcId, 0)) &&

 (ipc->entry.setupNotify)) {

 /\* call Notify\_attach \*/

 memReq = Notify\_sharedMemReq(remoteProcId, Ipc\_module->ipcSharedAddr);

 **if** (memReq != 0) {

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\*

 \* calloc required here due to race condition. Its possible

 \* that the slave, who creates the instance, tries a sendEvent

 \* before the master has created its instance because the

 \* state of memory was enabled from a previous run.

 \*/

 sharedAddr = Memory\_calloc(SharedRegion\_getHeap(0),

 memReq,

 SharedRegion\_getCacheLineSize(0),

 &eb);

 /\* make sure alloc did not fail \*/

 **if** (sharedAddr == NULL) {

 **return** (Ipc\_E\_MEMORY);

 }

 /\* if cache enabled, wbInv the calloc above \*/

 **if** (cacheEnabled) {

 Cache\_wbInv(sharedAddr, memReq, Cache\_Type\_ALL, TRUE);

 }

 /\* set the notify SRPtr \*/

 slave->notifySRPtr = SharedRegion\_getSRPtr(sharedAddr, 0);

 }

 **else** {

 /\* get the notify SRPtr \*/

 sharedAddr = SharedRegion\_getPtr(slave->notifySRPtr);

 }

 }

 **else** {

 sharedAddr = NULL;

 slave->notifySRPtr = 0;

 }

 /\* call attach to remote processor \*/

 status = Notify\_attach(remoteProcId, sharedAddr);

 **if** (status < 0) {

 **if** (**MultiProc\_self**() < remoteProcId && sharedAddr != NULL) {

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0), sharedAddr, memReq);

 }

 **return** (Ipc\_E\_FAIL);

 }

 }

 /\* Must come after GateMP\_start because depends on default GateMP \*/

 **if** (!(ti\_sdo\_utils\_NameServer\_isRegistered(remoteProcId)) &&

 (ipc->entry.setupNotify)) {

 memReq = ti\_sdo\_utils\_NameServer\_SetupProxy\_sharedMemReq(

 Ipc\_module->ipcSharedAddr);

 **if** (memReq != 0) {

 **if** (**MultiProc\_self**() < remoteProcId) {

 sharedAddr = Memory\_alloc(SharedRegion\_getHeap(0),

 memReq,

 SharedRegion\_getCacheLineSize(0),

 &eb);

 /\* make sure alloc did not fail \*/

 **if** (sharedAddr == NULL) {

 **return** (Ipc\_E\_MEMORY);

 }

 /\* set the NSRN SRPtr \*/

 slave->nsrnSRPtr = SharedRegion\_getSRPtr(sharedAddr, 0);

 }

 **else** {

 /\* get the NSRN SRPtr \*/

 sharedAddr = SharedRegion\_getPtr(slave->nsrnSRPtr);

 }

 }

 **else** {

 sharedAddr = NULL;

 slave->nsrnSRPtr = 0;

 }

 /\* call attach to remote processor \*/

 status = ti\_sdo\_utils\_NameServer\_SetupProxy\_attach(remoteProcId,

 sharedAddr);

 **if** (status < 0) {

 **if** (**MultiProc\_self**() < remoteProcId && sharedAddr != NULL) {

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0), sharedAddr, memReq);

 }

 **return** (Ipc\_E\_FAIL);

 }

 }

 /\* Must come after GateMP\_start because depends on default GateMP \*/

 **if** (!(ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_isRegistered(remoteProcId)) &&

 (ipc->entry.setupMessageQ)) {

 memReq = ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_sharedMemReq(

 Ipc\_module->ipcSharedAddr);

 **if** (memReq != 0) {

 **if** (**MultiProc\_self**() < remoteProcId) {

 sharedAddr = Memory\_alloc(SharedRegion\_getHeap(0),

 memReq, SharedRegion\_getCacheLineSize(0), &eb);

 /\* make sure alloc did not fail \*/

 **if** (sharedAddr == NULL) {

 **return** (Ipc\_E\_MEMORY);

 }

 /\* set the transport SRPtr \*/

 slave->transportSRPtr = SharedRegion\_getSRPtr(sharedAddr, 0);

 }

 **else** {

 /\* get the transport SRPtr \*/

 sharedAddr = SharedRegion\_getPtr(slave->transportSRPtr);

 }

 }

 **else** {

 sharedAddr = NULL;

 slave->transportSRPtr = 0;

 }

 /\* call attach to remote processor \*/

 status = ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_attach(remoteProcId,

 sharedAddr);

 **if** (status < 0) {

 **if** (**MultiProc\_self**() < remoteProcId && sharedAddr != NULL) {

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0), sharedAddr, memReq);

 }

 **return** (Ipc\_E\_FAIL);

 }

 }

 /\* writeback invalidate slave's shared memory if cache enabled \*/

 **if** (cacheEnabled) {

 **if** (**MultiProc\_self**() < remoteProcId) {

 Cache\_wbInv((Ptr)slave, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 }

 /\* Call user attach fxns \*/

 **for** (i = 0; i < ti\_sdo\_ipc\_Ipc\_numUserFxns; i++) {

 **if** (ti\_sdo\_ipc\_Ipc\_userFxns[i].userFxn.attach) {

 status = ti\_sdo\_ipc\_Ipc\_userFxns[i].userFxn.attach(

 ti\_sdo\_ipc\_Ipc\_userFxns[i].arg, remoteProcId);

 **if** (status < 0) {

 **return** (status);

 }

 }

 }

 /\* Finish the processor synchronization \*/

 status = ti\_sdo\_ipc\_Ipc\_procSyncFinish(remoteProcId,

 Ipc\_module->ipcSharedAddr);

 **if** (status < 0) {

 **return** (status);

 }

 /\* for atomically incrementing attached \*/

 hwiKey = Hwi\_disable();

 /\* now attached to remote processor \*/

 Ipc\_module->procEntry[clusterId].attached++;

 /\* restore interrupts \*/

 Hwi\_restore(hwiKey);

 **return** (status);

}

/\*

 \* ======== Ipc\_isAttached ========

 \*/

Bool **Ipc\_isAttached**(UInt16 remoteProcId)

{

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 /\* Assert remoteProcId is in our cluster \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 **if** (remoteProcId == **MultiProc\_self**()) {

 **return** (FALSE);

 }

 **else** {

 **return** (Ipc\_module->procEntry[clusterId].attached);

 }

}

/\*

 \* ======== Ipc\_detach ========

 \*/

Int **Ipc\_detach**(UInt16 remoteProcId)

{

 Int i;

 UInt16 baseId = **MultiProc\_getBaseIdOfCluster**();

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 Ptr notifySharedAddr;

 Ptr nsrnSharedAddr;

 Ptr msgqSharedAddr;

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*slave, \*master;

 SharedRegion\_Entry entry;

 ti\_sdo\_ipc\_Ipc\_ProcEntry \*ipc;

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 Int status = Ipc\_S\_SUCCESS;

 UInt hwiKey;

 /\* Assert remoteProcId is in our cluster and isn't our own \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 Assert\_isTrue(remoteProcId != **MultiProc\_self**(),

 ti\_sdo\_ipc\_Ipc\_A\_invArgument);

 /\* for checking and incrementing attached below \*/

 hwiKey = Hwi\_disable();

 **if** (Ipc\_module->procEntry[clusterId].attached > 1) {

 /\* only detach if attach count reaches 1 \*/

 Ipc\_module->procEntry[clusterId].attached--;

 Hwi\_restore(hwiKey);

 **return** (Ipc\_S\_BUSY);

 }

 **else** **if** (Ipc\_module->procEntry[clusterId].attached == 0) {

 /\* already detached, restore interrupts and return success \*/

 Hwi\_restore(hwiKey);

 **return** (Ipc\_S\_SUCCESS);

 }

 /\* restore interrupts \*/

 Hwi\_restore(hwiKey);

 /\* get region 0 information \*/

 SharedRegion\_getEntry(0, &entry);

 /\*

 \* Make sure we detach from all other procs in cluster before

 \* detaching from owner of SR 0.

 \*/

 **if** (remoteProcId == entry.ownerProcId) {

 **for** (i = 0; i < ti\_sdo\_utils\_MultiProc\_numProcsInCluster; i++, baseId++) {

 **if** ((baseId != **MultiProc\_self**()) && (baseId != entry.ownerProcId) &&

 (Ipc\_module->procEntry[i].attached)) {

 **return** (Ipc\_E\_FAIL);

 }

 }

 }

 /\* get the paramters associated with remoteProcId \*/

 ipc = &(Ipc\_module->procEntry[clusterId]);

 /\* determine the slave's slot \*/

 slave = Ipc\_getSlaveAddr(remoteProcId, Ipc\_module->ipcSharedAddr);

 /\* determine the master's slot \*/

 master = ti\_sdo\_ipc\_Ipc\_getMasterAddr(remoteProcId,

 Ipc\_module->ipcSharedAddr);

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)slave, reservedSize, Cache\_Type\_ALL, TRUE);

 Cache\_inv((Ptr)master, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\* check to make sure master is not trying to attach \*/

 **if** (master->startedKey == ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 **else** {

 /\* check to make sure slave is not trying to attach \*/

 **if** (slave->startedKey == ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 /\* The slave processor waits for master to finish its detach sequence \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 **if** (master->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCDETACH) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 /\* Call user detach fxns \*/

 **for** (i = 0; i < ti\_sdo\_ipc\_Ipc\_numUserFxns; i++) {

 **if** (ti\_sdo\_ipc\_Ipc\_userFxns[i].userFxn.detach) {

 status = ti\_sdo\_ipc\_Ipc\_userFxns[i].userFxn.detach(

 ti\_sdo\_ipc\_Ipc\_userFxns[i].arg, remoteProcId);

 **if** (status < 0) {

 **return** (status);

 }

 }

 }

 **if** ((ipc->entry.setupMessageQ) &&

 (ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_isRegistered(remoteProcId))) {

 /\* call MessageQ\_detach for remote processor \*/

 status = ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_detach(remoteProcId);

 **if** (status < 0) {

 **return** (Ipc\_E\_FAIL);

 }

 **if** (slave->transportSRPtr) {

 /\* free the memory if slave processor \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\* get the pointer to MessageQ transport instance \*/

 msgqSharedAddr = SharedRegion\_getPtr(slave->transportSRPtr);

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0),

 msgqSharedAddr,

 ti\_sdo\_ipc\_MessageQ\_SetupTransportProxy\_sharedMemReq(

 msgqSharedAddr));

 /\* set pointer for MessageQ transport instance back to NULL \*/

 slave->transportSRPtr = NULL;

 }

 }

 }

 **if** ((ipc->entry.setupNotify) &&

 (ti\_sdo\_utils\_NameServer\_isRegistered(remoteProcId))) {

 /\* call NameServer\_SetupProxy\_detach for remote processor \*/

 status = ti\_sdo\_utils\_NameServer\_SetupProxy\_detach(remoteProcId);

 **if** (status < 0) {

 **return** (Ipc\_E\_FAIL);

 }

 **if** (slave->nsrnSRPtr) {

 /\* free the memory if slave processor \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\* get the pointer to NSRN instance \*/

 nsrnSharedAddr = SharedRegion\_getPtr(slave->nsrnSRPtr);

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0),

 nsrnSharedAddr,

 ti\_sdo\_utils\_NameServer\_SetupProxy\_sharedMemReq(

 nsrnSharedAddr));

 /\* set pointer for NSRN instance back to NULL \*/

 slave->nsrnSRPtr = NULL;

 }

 }

 }

 **if** ((ipc->entry.setupNotify) &&

 (Notify\_intLineRegistered(remoteProcId, 0))) {

 /\* call Notify\_detach for remote processor \*/

 status = ti\_sdo\_ipc\_Notify\_detach(remoteProcId);

 **if** (status < 0) {

 **return** (Ipc\_E\_FAIL);

 }

 **if** (slave->notifySRPtr) {

 /\* free the memory if slave processor \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\* get the pointer to Notify instance \*/

 notifySharedAddr = SharedRegion\_getPtr(slave->notifySRPtr);

 /\* free the memory back to SharedRegion 0 heap \*/

 Memory\_free(SharedRegion\_getHeap(0),

 notifySharedAddr,

 Notify\_sharedMemReq(remoteProcId, notifySharedAddr));

 /\* set pointer for Notify instance back to NULL \*/

 slave->notifySRPtr = NULL;

 }

 }

 }

 **if** (**MultiProc\_self**() < remoteProcId) {

 slave->configListHead = ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR;

 slave->startedKey = ti\_sdo\_ipc\_Ipc\_PROCSYNCDETACH;

 **if** (cacheEnabled) {

 Cache\_wbInv((Ptr)slave, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 }

 **else** {

 master->configListHead = ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR;

 master->startedKey = ti\_sdo\_ipc\_Ipc\_PROCSYNCDETACH;

 **if** (cacheEnabled) {

 Cache\_wbInv((Ptr)master, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 }

 /\* attached must be decremented atomically \*/

 hwiKey = Hwi\_disable();

 /\* now detached from remote processor \*/

 Ipc\_module->procEntry[clusterId].attached--;

 /\* restore interrupts \*/

 Hwi\_restore(hwiKey);

 **return** (status);

}

/\*

 \* ======== Ipc\_readConfig ========

 \*/

Int **Ipc\_readConfig**(UInt16 remoteProcId, UInt32 tag, Ptr cfg, SizeT size)

{

 Int status = Ipc\_E\_FAIL;

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 **volatile** ti\_sdo\_ipc\_Ipc\_ConfigEntry \*entry;

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 /\* Assert that the remoteProc in our cluster and isn't our own \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 **if** (cacheEnabled) {

 Cache\_inv(Ipc\_module->procEntry[clusterId].remoteConfigList,

 SharedRegion\_getCacheLineSize(0),

 Cache\_Type\_ALL,

 TRUE);

 }

 entry = (ti\_sdo\_ipc\_Ipc\_ConfigEntry \*)((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)

 Ipc\_module->procEntry[clusterId].remoteConfigList)->first;

 **while** ((SharedRegion\_SRPtr)entry != ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR) {

 entry = (ti\_sdo\_ipc\_Ipc\_ConfigEntry \*)

 SharedRegion\_getPtr((SharedRegion\_SRPtr)entry);

 /\* Traverse the list to find the tag \*/

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)entry,

 size + **sizeof**(ti\_sdo\_ipc\_Ipc\_ConfigEntry),

 Cache\_Type\_ALL,

 TRUE);

 }

 **if** ((entry->remoteProcId == **MultiProc\_self**()) &&

 (entry->localProcId == remoteProcId) &&

 (entry->tag == tag)) {

 **if** (size == entry->size) {

 **memcpy**(cfg, (Ptr)((UInt32)entry + **sizeof**(ti\_sdo\_ipc\_Ipc\_ConfigEntry)),

 entry->size);

 **return** (Ipc\_S\_SUCCESS);

 }

 **else** {

 **return** (Ipc\_E\_FAIL);

 }

 }

 entry = (ti\_sdo\_ipc\_Ipc\_ConfigEntry \*)entry->next;

 }

 **return** (status);

}

/\*

 \* ======== Ipc\_start ========

 \*/

Int **Ipc\_start**()

{

 Int i;

 UInt16 baseId = **MultiProc\_getBaseIdOfCluster**();

 SharedRegion\_Entry entry;

 Ptr ipcSharedAddr;

 Ptr gateMPSharedAddr;

 GateMP\_Params gateMPParams;

 Int status;

 /\* Check whether Ipc\_start has been called. If so, succeed. \*/

 **if** (Ipc\_module->ipcSharedAddr != NULL) {

 **return** (Ipc\_S\_ALREADYSETUP);

 }

 **if** (ti\_sdo\_ipc\_Ipc\_generateSlaveDataForHost) {

 /\* get Ipc\_sr0MemorySetup out of the cache \*/

 Cache\_inv(&Ipc\_sr0MemorySetup,

 ti\_sdo\_ipc\_SharedRegion\_cacheLineSize,

 Cache\_Type\_ALL,

 TRUE);

 /\* check Ipc\_sr0MemorySetup variable \*/

 **if** (Ipc\_sr0MemorySetup == 0x0) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 /\* get region 0 information \*/

 SharedRegion\_getEntry(0, &entry);

 /\* if entry is not valid then return \*/

 **if** (entry.isValid == FALSE) {

 **return** (Ipc\_E\_NOTREADY);

 }

 /\*

 \* Need to reserve memory in region 0 for processor synchronization.

 \* This must done before SharedRegion\_start().

 \*/

 ipcSharedAddr = ti\_sdo\_ipc\_SharedRegion\_reserveMemory(

 0, Ipc\_getRegion0ReservedSize());

 /\* must reserve memory for GateMP before SharedRegion\_start() \*/

 gateMPSharedAddr = ti\_sdo\_ipc\_SharedRegion\_reserveMemory(0,

 ti\_sdo\_ipc\_GateMP\_getRegion0ReservedSize());

 /\* Init params for default gate (must match those in GateMP\_start()) \*/

 **GateMP\_Params\_init**(&gateMPParams);

 gateMPParams.localProtect = *GateMP\_LocalProtect\_TASKLET*;

 **if** (ti\_sdo\_utils\_MultiProc\_numProcessors > 1) {

 gateMPParams.remoteProtect = *GateMP\_RemoteProtect\_SYSTEM*;

 }

 **else** {

 gateMPParams.remoteProtect = *GateMP\_RemoteProtect\_NONE*;

 }

 /\* reserve memory for default gate before SharedRegion\_start() \*/

 ti\_sdo\_ipc\_SharedRegion\_reserveMemory(0, **GateMP\_sharedMemReq**(&gateMPParams));

 /\* clear the reserved memory \*/

 ti\_sdo\_ipc\_SharedRegion\_clearReservedMemory();

 /\* Set shared addresses \*/

 Ipc\_module->ipcSharedAddr = ipcSharedAddr;

 Ipc\_module->gateMPSharedAddr = gateMPSharedAddr;

 /\* create default GateMP, must be called before SharedRegion start \*/

 status = ti\_sdo\_ipc\_GateMP\_start(Ipc\_module->gateMPSharedAddr);

 **if** (status < 0) {

 **return** (status);

 }

 /\* create HeapMemMP in each SharedRegion \*/

 status = ti\_sdo\_ipc\_SharedRegion\_start();

 **if** (status < 0) {

 **return** (status);

 }

 /\* Call attach for all procs if procSync is ALL \*/

 **if** (ti\_sdo\_ipc\_Ipc\_procSync == *ti\_sdo\_ipc\_Ipc\_ProcSync\_ALL*) {

 /\* Must attach to owner first to get default GateMP and HeapMemMP \*/

 **if** (**MultiProc\_self**() != entry.ownerProcId) {

 **do** {

 status = Ipc\_attach(entry.ownerProcId);

 } **while** (status == Ipc\_E\_NOTREADY);

 **if** (status < 0) {

 /\* Ipc\_attach failed. Get out of Ipc\_start \*/

 **return** (status);

 }

 }

 /\* Loop to attach to all other processors in cluster \*/

 **for** (i = 0; i < ti\_sdo\_utils\_MultiProc\_numProcsInCluster; i++, baseId++) {

 **if** ((baseId == **MultiProc\_self**()) || (baseId == entry.ownerProcId)) {

 **continue**;

 }

 /\* Skip the processor if there are no interrupt lines to it \*/

 **if** (Notify\_numIntLines(baseId) == 0) {

 **continue**;

 }

 /\* call Ipc\_attach for every remote processor \*/

 **do** {

 status = Ipc\_attach(baseId);

 } **while** (status == Ipc\_E\_NOTREADY);

 **if** (status < 0) {

 /\* Ipc\_attach failed. Get out of Ipc\_start \*/

 **return** (status);

 }

 }

 }

 **return** (status);

}

/\*

 \* ======== Ipc\_stop ========

 \*/

Int **Ipc\_stop**()

{

 /\* clear local module state \*/

 Ipc\_module->gateMPSharedAddr = NULL;

 Ipc\_module->ipcSharedAddr = NULL;

 /\* reset Shared Region 0 reservedSize and heap handle \*/

 ti\_sdo\_ipc\_SharedRegion\_resetInternalFields(0);

 /\* set sr0MemorySetup back to 0 \*/

 **if** (ti\_sdo\_ipc\_Ipc\_generateSlaveDataForHost) {

 Ipc\_sr0MemorySetup = 0;

 Cache\_wbInv(&Ipc\_sr0MemorySetup,

 ti\_sdo\_ipc\_SharedRegion\_cacheLineSize,

 Cache\_Type\_ALL,

 TRUE);

 }

 **return** (Ipc\_S\_SUCCESS);

}

/\*

 \* ======== Ipc\_writeConfig ========

 \*/

Int **Ipc\_writeConfig**(UInt16 remoteProcId, UInt32 tag, Ptr cfg, SizeT size)

{

 Int status = Ipc\_S\_SUCCESS;

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 ti\_sdo\_ipc\_Ipc\_ConfigEntry \*entry;

 Error\_Block eb;

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 SizeT cacheLineSize = SharedRegion\_getCacheLineSize(0);

 /\* Assert that the remoteProc in our cluster \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 Error\_init(&eb);

 /\* Allocate memory from the shared heap (System Heap) \*/

 entry = Memory\_alloc(SharedRegion\_getHeap(0),

 size + **sizeof**(ti\_sdo\_ipc\_Ipc\_ConfigEntry),

 cacheLineSize,

 &eb);

 **if** (entry == NULL) {

 **return** (Ipc\_E\_FAIL);

 }

 entry->remoteProcId = remoteProcId;

 entry->localProcId = **MultiProc\_self**();

 entry->tag = tag;

 entry->size = size;

 **memcpy**((Ptr)((UInt32)entry + **sizeof**(ti\_sdo\_ipc\_Ipc\_ConfigEntry)), cfg, size);

 /\* Create a linked-list of config \*/

 **if** (((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)

 Ipc\_module->procEntry[clusterId].localConfigList)->first

 == ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR) {

 entry->next = ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR;

 ((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)

 Ipc\_module->procEntry[clusterId].localConfigList)->first =

 (Bits32)SharedRegion\_getSRPtr(entry, 0);

 }

 **else** {

 entry->next = ((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)

 Ipc\_module->procEntry[clusterId].localConfigList)->first;

 ((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)

 Ipc\_module->procEntry[clusterId].localConfigList)->first =

 (Bits32)SharedRegion\_getSRPtr(entry, 0);

 }

 **if** (cacheEnabled) {

 Cache\_wbInv(Ipc\_module->procEntry[clusterId].localConfigList,

 SharedRegion\_getCacheLineSize(0),

 Cache\_Type\_ALL,

 FALSE);

 Cache\_wbInv(entry, size + **sizeof**(ti\_sdo\_ipc\_Ipc\_ConfigEntry), Cache\_Type\_ALL,

 TRUE);

 }

 **return** (status);

}

/\*

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Module Functions

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \*/

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_dummy ========

 \*/

Void **ti\_sdo\_ipc\_Ipc\_dummy**()

{

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_getEntry ========

 \*/

Void **ti\_sdo\_ipc\_Ipc\_getEntry**(ti\_sdo\_ipc\_Ipc\_Entry \*entry)

{

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(entry->remoteProcId);

 /\* Assert remoteProcId is in our cluster \*/

 Assert\_isTrue(clusterId < ti\_sdo\_utils\_MultiProc\_numProcsInCluster,

 ti\_sdo\_utils\_MultiProc\_A\_invalidMultiProcId);

 /\* Get the setupNotify flag \*/

 entry->setupNotify =

 Ipc\_module->procEntry[clusterId].entry.setupNotify;

 /\* Get the setupMessageQ flag \*/

 entry->setupMessageQ =

 Ipc\_module->procEntry[clusterId].entry.setupMessageQ;

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_setEntry ========

 \*/

Void **ti\_sdo\_ipc\_Ipc\_setEntry**(ti\_sdo\_ipc\_Ipc\_Entry \*entry)

{

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(entry->remoteProcId);

 /\* Set the setupNotify flag \*/

 Ipc\_module->procEntry[clusterId].entry.setupNotify =

 entry->setupNotify;

 /\* Set the setupMessageQ flag \*/

 Ipc\_module->procEntry[clusterId].entry.setupMessageQ =

 entry->setupMessageQ;

}

/\*

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Internal Functions

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \*/

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_getMasterAddr ========

 \*/

Ptr **ti\_sdo\_ipc\_Ipc\_getMasterAddr**(UInt16 remoteProcId, Ptr sharedAddr)

{

 Int slot;

 UInt16 masterId;

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*master;

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 /\* determine the master's procId and slot \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 masterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 slot = ti\_sdo\_utils\_MultiProc\_getClusterId(**MultiProc\_self**());

 }

 **else** {

 masterId = ti\_sdo\_utils\_MultiProc\_getClusterId(**MultiProc\_self**());

 slot = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 }

 /\* determine the reserve address for master between self and remote \*/

 master = (ti\_sdo\_ipc\_Ipc\_Reserved \*)((UInt32)sharedAddr +

 ((masterId \* reservedSize) +

 (slot \* **sizeof**(ti\_sdo\_ipc\_Ipc\_Reserved))));

 **return** ((Ptr)master);

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_getRegion0ReservedSize ========

 \*/

SizeT **ti\_sdo\_ipc\_Ipc\_getRegion0ReservedSize**(Void)

{

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 /\* Calculate the total amount to reserve \*/

 reservedSize = reservedSize \* ti\_sdo\_utils\_MultiProc\_numProcsInCluster;

 **return** (reservedSize);

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_getSlaveAddr ========

 \*/

Ptr **ti\_sdo\_ipc\_Ipc\_getSlaveAddr**(UInt16 remoteProcId, Ptr sharedAddr)

{

 Int slot;

 UInt16 slaveId;

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*slave;

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 /\* determine the slave's procId and slot \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 slaveId = ti\_sdo\_utils\_MultiProc\_getClusterId(**MultiProc\_self**());

 slot = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId) - 1;

 }

 **else** {

 slaveId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 slot = ti\_sdo\_utils\_MultiProc\_getClusterId(**MultiProc\_self**()) - 1;

 }

 /\* determine the reserve address for slave between self and remote \*/

 slave = (ti\_sdo\_ipc\_Ipc\_Reserved \*)((UInt32)sharedAddr +

 ((slaveId \* reservedSize) +

 (slot \* **sizeof**(ti\_sdo\_ipc\_Ipc\_Reserved))));

 **return** ((Ptr)slave);

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_procSyncStart ========

 \* The owner of SharedRegion 0 writes to its reserve memory address

 \* in region 0 to let the other processors know it has started.

 \* It then spins until the other processors start.

 \* The other processors write their reserve memory address in

 \* region 0 to let the owner processor know they've started.

 \* The other processors then spin until the owner processor writes

 \* to let them know that its finished the process of synchronization

 \* before continuing.

 \*/

Int **ti\_sdo\_ipc\_Ipc\_procSyncStart**(UInt16 remoteProcId, Ptr sharedAddr)

{

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*self, \*remote;

 ti\_sdo\_ipc\_Ipc\_ProcEntry \*ipc;

 UInt16 clusterId = ti\_sdo\_utils\_MultiProc\_getClusterId(remoteProcId);

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 /\* don't do any synchronization if procSync is NONE \*/

 **if** (ti\_sdo\_ipc\_Ipc\_procSync == *ti\_sdo\_ipc\_Ipc\_ProcSync\_NONE*) {

 **return** (Ipc\_S\_SUCCESS);

 }

 /\* determine self and remote pointers \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 self = Ipc\_getSlaveAddr(remoteProcId, sharedAddr);

 remote = ti\_sdo\_ipc\_Ipc\_getMasterAddr(remoteProcId, sharedAddr);

 }

 **else** {

 self = ti\_sdo\_ipc\_Ipc\_getMasterAddr(remoteProcId, sharedAddr);

 remote = Ipc\_getSlaveAddr(remoteProcId, sharedAddr);

 }

 /\* construct the config list \*/

 ipc = &(Ipc\_module->procEntry[clusterId]);

 ipc->localConfigList = (Ptr)&self->configListHead;

 ipc->remoteConfigList = (Ptr)&remote->configListHead;

 ((ti\_sdo\_ipc\_Ipc\_ConfigHead \*)ipc->localConfigList)->first =

 ti\_sdo\_ipc\_SharedRegion\_INVALIDSRPTR;

 **if** (cacheEnabled) {

 Cache\_wbInv(ipc->localConfigList, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 **if** (**MultiProc\_self**() < remoteProcId) {

 /\* set my processor's reserved key to start \*/

 self->startedKey = ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART;

 /\* write back my processor's reserve key \*/

 **if** (cacheEnabled) {

 Cache\_wbInv((Ptr)self, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 /\* wait for remote processor to start \*/

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)remote, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 **if** (remote->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 **else** {

 /\* wait for remote processor to start \*/

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)remote, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 **if** ((self->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART) &&

 (remote->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART)) {

 **return** (Ipc\_E\_NOTREADY);

 }

 /\* set my processor's reserved key to start \*/

 self->startedKey = ti\_sdo\_ipc\_Ipc\_PROCSYNCSTART;

 /\* write my processor's reserve key back \*/

 **if** (cacheEnabled) {

 Cache\_wbInv((Ptr)self, reservedSize, Cache\_Type\_ALL, TRUE);

 /\* wait for remote processor to finish \*/

 Cache\_inv((Ptr)remote, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 **if** (remote->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCFINISH) {

 **return** (Ipc\_E\_NOTREADY);

 }

 }

 **return** (Ipc\_S\_SUCCESS);

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_procSyncFinish ========

 \* Each processor writes its reserve memory address in SharedRegion 0

 \* to let the other processors know its finished the process of

 \* synchronization.

 \*/

Int **ti\_sdo\_ipc\_Ipc\_procSyncFinish**(UInt16 remoteProcId, Ptr sharedAddr)

{

 **volatile** ti\_sdo\_ipc\_Ipc\_Reserved \*self, \*remote;

 SizeT reservedSize = ti\_sdo\_ipc\_Ipc\_reservedSizePerProc();

 Bool cacheEnabled = SharedRegion\_isCacheEnabled(0);

 UInt oldPri;

 /\* don't do any synchronization if procSync is NONE \*/

 **if** (ti\_sdo\_ipc\_Ipc\_procSync == *ti\_sdo\_ipc\_Ipc\_ProcSync\_NONE*) {

 **return** (Ipc\_S\_SUCCESS);

 }

 /\* determine self and remote pointers \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 self = Ipc\_getSlaveAddr(remoteProcId, sharedAddr);

 remote = ti\_sdo\_ipc\_Ipc\_getMasterAddr(remoteProcId, sharedAddr);

 }

 **else** {

 self = ti\_sdo\_ipc\_Ipc\_getMasterAddr(remoteProcId, sharedAddr);

 remote = Ipc\_getSlaveAddr(remoteProcId, sharedAddr);

 }

 /\* set my processor's reserved key to finish \*/

 self->startedKey = ti\_sdo\_ipc\_Ipc\_PROCSYNCFINISH;

 /\* write back my processor's reserve key \*/

 **if** (cacheEnabled) {

 Cache\_wbInv((Ptr)self, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 /\* if slave processor, wait for remote to finish sync \*/

 **if** (**MultiProc\_self**() < remoteProcId) {

 **if** (BIOS\_getThreadType() == BIOS\_ThreadType\_Task) {

 oldPri = Task\_getPri(Task\_self());

 }

 /\* wait for remote processor to finish \*/

 **while** (remote->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCFINISH &&

 remote->startedKey != ti\_sdo\_ipc\_Ipc\_PROCSYNCDETACH) {

 /\* Set self priority to 1 [lowest] and yield cpu \*/

 **if** (BIOS\_getThreadType() == BIOS\_ThreadType\_Task) {

 Task\_setPri(Task\_self(), 1);

 Task\_yield();

 }

 /\* Check the remote's sync flag \*/

 **if** (cacheEnabled) {

 Cache\_inv((Ptr)remote, reservedSize, Cache\_Type\_ALL, TRUE);

 }

 }

 /\* Restore self priority \*/

 **if** (BIOS\_getThreadType() == BIOS\_ThreadType\_Task) {

 Task\_setPri(Task\_self(), oldPri);

 }

 }

 **return** (Ipc\_S\_SUCCESS);

}

/\*

 \* ======== ti\_sdo\_ipc\_Ipc\_reservedSizePerProc ========

 \*/

SizeT **ti\_sdo\_ipc\_Ipc\_reservedSizePerProc**(Void)

{

 SizeT reservedSize = **sizeof**(ti\_sdo\_ipc\_Ipc\_Reserved) \*

 ti\_sdo\_utils\_MultiProc\_numProcsInCluster;

 SizeT cacheLineSize = SharedRegion\_getCacheLineSize(0);

 /\* Calculate amount to reserve per processor \*/

 **if** (cacheLineSize > reservedSize) {

 /\* Use cacheLineSize if larger than reservedSize \*/

 reservedSize = cacheLineSize;

 }

 **else** {

 /\* Round reservedSize to cacheLineSize \*/

 reservedSize = \_Ipc\_roundup(reservedSize, cacheLineSize);

 }

 **return** (reservedSize);

}

/\*

 \*/

/\*

 \* @(#) ti.sdo.ipc; 1, 0, 0, 0,1; 5-22-2012 16:18:04; /db/vtree/library/trees/ipc/ipc-h32/src/ xlibrary

 \*/