Intro to:
Inter-Processor Communications (IPC)
Agenda

• Basic Concepts
• IPC Services
• Setup and Examples
• IPC Transports
• Lab or Demo
Basic Concepts

• Basic Concepts
• IPC Services
• Setup and Examples
• IPC Transports
• Lab or Demo
IPC – Definition

- IPC = Inter-Processor Communication
- While this definition is rather generic, it really means:
  
  “Transporting data and/or signals between threads of execution”

- These threads could be located anywhere:

  How would YOU solve this problem?
IPC – Possible Solutions

How do you transport the data and signal?

- **Manual**: Using shared memory and an INT (*possible contention*)
- **Auto**: Using existing RTOS/Framework Utilities (i.e. IPC)

![Diagram showing IPC between threads and cores](image)

What solutions exist in TI’s RTOS to perform IPC?
IPC – RTOS/Framework Solutions

- **SAME CorePac:** TI’s RTOS (SYS/BIOS) supports several services for inter-thread communication (*e.g.* semaphores, queues, mailboxes, etc.).

- **DIFFERENT CorePac:** The IPC framework supports communications between CorePacs via several transports.

- **DIFFERENT DEVICE:** IPC transports can also be implemented between devices.

- **KEY:** Same IPC APIs can be used for local or remote communications.

What kinds of transports are possible?

- **Solutions**
  - SYS/BIOS (or IPC)
  - IPC + transport
  - IPC + transport

---

[Diagram showing communication between threads and devices]
IPC – Transports

- IPC supports several transports:
  - CorePac → CorePac  (Shared Memory Model, Multicore Navigator)
  - Device → Device  (Serial Rapid I/O)

- Chosen at configuration; **Same code** regardless of location

What services does IPC provide?
IPC Services

• Basic Concepts

• **IPC Services**
  – Notify
  – Data Passing
  – Message Queue
  – Support Utilities

• Setup and Examples

• IPC Transports

• Lab or Demo
The IPC Package is a library that contains many user services:

**Transport – Shared Memory**
- **SIMPLE** – Data Passing + Notify
  "Send a 32-bit message and notify via INT"
- **STATIC** – Data Passing via Linked Lists + Notify
  "Pass linked list elements to/from fixed memory. Notify via INT."
- **DYNAMIC** – Data Passing via Heaps + Notify
  "Pass dynamically created linked list elements. Notify via INT."

**Multiple Transport – Shared Mem, Multicore Navigator, SRIO**
- **MESSAGE QUEUE** – the ultimate IPC capability
  "Configure transport, use simple APIs to send MSGs + notify"

Let's examine how a simple NOTIFY works first...
IPC Services - Notify

• Basic Concepts
• IPC Services
  – Notify
  – Data Passing
  – Message Queue
  – Support Utilities
• Setup and Examples
• IPC Transports
• Lab or Demo
Using Notify – Concepts

- Notify is the **SIMPLEST** form of IPC communication.
- Use: Send simple 32-bit payload to another CorePac.
- Sender: Transmit payload and eventId (INT) to destination CorePac.
- Receiver: Receive payload/eventId and run associated callback fxn.

Let's see an example...

![Diagram of IPC communication between CorePacs and threads]
Using Notify – Example

- Notify APIs interact with IPC Notify module.
- Configuration is done via the IPC MultiProc module.
- Notify manager handles multiple interrupt events via single INT line.

```
// SENDER
Void Notify_sendEvent(dstId, lineId, eventId, payload, waitClear);

// RECEIVER
Void Notify_registerEvent(srcId, lineId, eventId, cbFxn, cbArg);
```

- When RECEIVER gets this eventId from the SENDER, the cbFxn is called and passes the srcId, eventId, payload and cbArg.
Example Callback Function

```c
/*
* ======== cbFxns ========
* This fxn was registered with Notify. It is called when any event is sent to this CPU.
*/

void cbFxns(UInt16 procId, UInt16 lineId, UInt32 eventId, UArg arg, UInt32 payload)
{
    /* The payload is a sequence number. */
    recvProcId = procId;
    seq = payload;
    Semaphore_post(semHandle);
}
```
IPC Services - Data Passing

• Basic Concepts
• **IPC Services**
  – Notify
  – **Data Passing**
  – Message Queue
  – Support Utilities
• Setup and Examples
• IPC Transports
• Lab or Demo
Data Passing – Concepts

- Uses *doubly linked lists* to share data between CorePacs
- Use: Send more complex data elements to another CorePac.
- Linked lists can be configured statically or dynamically.
- Not typically used by itself – but as a building block for MessageQ

Let’s look at the static version first…
Data Passing – Static

- Data Passing uses a *double linked list* that can be shared between CorePacs; Linked list is defined **STATICALLY**.
- ListMP handles address translation and cache coherency.
- GateMP protects read/write accesses.
- ListMP is typically used by MessageQ (covered later), not by itself.

```
Notify

User APIs

ListMP

MultiProc

Uses

Shared Region

Uses

GateMP

NameServer

Cfg
```

Now, the dynamic version...
Data Passing – Dynamic

- Data Passing uses a double linked list that can be shared between CPUs. Linked list is defined DYNAMICALLY (via heap).
- Same as previous, except linked lists are allocated from Heap
- Typically not used alone – but as a building block for MessageQ

And last...let's look at MessageQ...
IPC Services – Message Queue

- Basic Concepts
- **IPC Services**
  - Notify
  - Data Passing
  - **Message Queue**
  - Support Utilities
- Setup and Examples
- IPC Transports
- Lab or Demo
MessageQ – Concepts

- Supports structured sending/receiving of variable-length messages
- Default “transport” is shared memory, but other transports (Multicore Navigator, SRIO) can be configured.
- APIs do NOT change based on transport – only the CFG (init) code
- Supports SINGLE reader, multiple WRITER (READER owns mailbox)

Let's see an example...
Using MessageQ (1/4)

- MessageQ transactions begin with **READER** creating a MessageQ.
- **READER**’s attempt to get a message results in a block (when semaphore specified), since no messages are in the queue yet.

What happens next?
Using MessageQ (2/4)

CorePac 1 - WRITER

- WRITER begins by opening MessageQ created by READER.
- WRITER gets a message block from a heap and fills it, as desired.
- WRITER puts the message into the MessageQ.

CorePac 2 - READER

- MessageQ_create("myQ", ...);
- MessageQ_get("myQ", &msg...);

How does the READER get unblocked?
Using MessageQ (3/4)

**CorePac 1 - WRITER**
- `MessageQ_open` (“myQ”, …);
- `msg = MessageQ_aloc` (heap, size,…);
- `MessageQ_put` (“myQ”, msg, …);
- `MessageQ_close` (“myQ”, …);

**CorePac 2 - READER**
- `MessageQ_create` (“myQ”, …);
- `MessageQ_get` (“myQ”, &msg…);
- ***PROCESS MSG***
- `MessageQ_free` (“myQ”, …);
- `MessageQ_delete` (“myQ”, …);

- **Once WRITER** puts msg in MessageQ, READER is unblocked.
- **READER** can now read/evaluate the received message.
- **READER** frees message back to Heap.
- **READER** can optionally delete the created MessageQ, if desired.
Using MessageQ (4/4)

- The message object manages queuing of messages passed.
- An **ALLOCATOR** mechanism gets the buffers (standard is HeapMemMP).
- A **TRANSPORT** mechanism can be specified during init (default is shared memory, but can also use Multicore Navigator or SRIO).

---

Which IPC modules does MessageQ use?
MessageQ – Configuration

- All API calls use the MessageQ module in IPC.
- User must also configure MultiProc and SharedRegion modules.
- All other configuration/setup is performed by MessageQ.

---

Some final notes on MessageQ...
MessageQ – Miscellaneous Notes

- O/S independent; If one CorePac is running LINUX and using SYS/Link, the API calls do not change.
- Messages can be allocated statically or dynamically.
- Works with all threading modules (HWI, SWI, Task)
- Can synchronize via semaphores, SWIs, events, wait
- Timeouts are allowed when Tasks receives message.
- User can specify three priority levels (normal, high, urgent).
IPC Services – Support Utilities

- Basic Concepts
- **IPC Services**
  - Notify
  - Data Passing
  - Message Queue
  - **Support Utilities**
- Setup and Examples
- IPC Transports
- Lab or Demo
IPC Support Utilities (1/2)

- IPC contains several utilities, most of which do NOT need to be configured by the user.
- Here is a short description of each IPC utility:

**IPC**
- Initializes IPC subsystems
- All applications that use IPC must call IPC_start()
- `setupNotify` and `setupMessageQ` specify whether to set up these IPC modules

**MultiProc**
- Lightweight module that simply stores processor IDs

**SharedRegion**
- Manages shared memory using HeapMemMP allocator
- Handles address translation for shared memory regions

More utilities...
IPC contains several utilities, most of which do NOT need to be configured by the user.

Here is a short description of each IPC utility:

**ListMP**
- Provides linked list in shared memory
- Uses multi-processor gate (GateMP) to prevent collisions on lists

**GateMP**
- Multi-processor gate that provides local (against other threads on local core) and remote context protection

**HeapMemMP**
- Traditional heap; Supports variable-sized alloc/free
- All allocations are aligned on cache line boundaries
Setup and Examples

• Basic Concepts
• IPC Services
• Setup and Examples
• IPC Transports
• Lab or Demo
IPC – Tools/Setup Required

- IPC is a package (library) that is installed with the MCSDK.
- IPC also requires SYS/BIOS (threading) and XDC tools (packaging) – installed with the MCSDK (supported by SYS/BIOS ROV and RTA).
- IPC will run on the latest TI multicore devices (C667x, C665x), as well as C647x and ARM+DSP devices.

Users can either install these packages separately or via the MCSDK install.

What IPC examples exist in the MCSDK?
IPC – Examples

- IPC examples are installed along with the MCSDK in the PDK folder.
- Simply import these projects into CCS and analyze them.
- Some users start with this example code and modify as necessary.

- QMSS – Multicore Navigator (Queue Manager Subsystem)
- SHM – Shared Memory
- SRIO – SRIO (loopback)
- SRIO – Chip to Chip
IPC Transports

• Basic Concepts
• IPC Services
• Setup and Examples
• IPC Transports
• Lab or Demo
An IPC TRANSPORT is a...

“combination of physical H/W and driver code that allows two threads to communicate on the same device or across devices.”

IPC supports three different transports:

- **Shared Memory**
  - This is the default transport.
  - Uses on-chip shared memory resources and interrupt lines to signal the availability of data.

- **Multicore Navigator**
  - Available on C667x and C665x processors.
  - Uses queues and descriptors plus built-in signaling to transmit/receive “packets” or msgs between threads.

- **SRIO**
  - Hardware serial peripheral that connects two or more DEVICES together.

For MessageQ, only the configuration (init) code changes. All other code (e.g. put/get) remains unchanged.

Let's look briefly at the Multicore Navigator first...
The Multicore Navigator is an innovative packet-based infrastructure that facilitates data movement and multicore control.

Provides a highly efficient inter-core communication mechanism. H/W queues and packet DMA are the basic building blocks for IPC.

Refer to the MCSDK examples for required init/CFG code.
The **SRIO** transport enables MessageQ to send data between tasks, cores and devices via the SRIO IP block.

Refer to the MCSDK examples for setup code required to use MessageQ over this transport.
Lab/Demo

- Basic Concepts
- IPC Services
- Setup and Examples
- IPC Transports
- Lab or Demo