Intro to:
Inter-Processor Communications (IPC)
Agenda

- Basic Concepts
- IPC Services
- Setup and Examples
- IPC Transports
- For More Information
Basic Concepts

- Basic Concepts
- IPC Services
- Setup and Examples
- IPC Transports
- For More Information
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:

```
Thread 1                      Data                      Thread 2
CorePac 0  →  CorePac 0
CorePac 0  →  CorePac 1
Device 0  →  Device 1
```

How would YOU solve this problem?
IPC – Possible Solutions

◆ How do you transport the data and signal?
  - **Manual**: Develop your own protocol using device resources
  - **Auto**: Using existing RTOS/Framework Utilities (i.e., IPC)

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.

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

- **Thread 1**
  - CorePac 0 → CorePac 0
  - CorePac 0 → CorePac 1
  - Device 0 → Device 1

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

*Texas Instruments*
Current IPC implementation uses several transports:

- **CorePac → CorePac** (Shared Memory Model, Multicore Navigator)
- **Device → Device** (Serial Rapid I/O)

Chosen at configuration; **Same code** regardless of thread location.
IPC Services

- Basic Concepts
- **IPC Services**
  - Message Queue
  - Notify
  - Data Passing
  - Support Utilities
- Setup and Examples
- IPC Transports
- For More Information
IPC Services

- The IPC package is a set of APIs
- MessageQ uses the modules below ...
- But each module can also be used independently.

![Diagram of IPC Services](image-url)
IPC Services – Message Queue

◆ Basic Concepts
◆ IPC Services
  • Message Queue
  • Notify
  • Data Passing
  • Support Utilities
◆ Setup and Examples
◆ IPC Transports
◆ For More Information
MessageQ – Highest Layer API

- SINGLE reader, multiple WRITERS model (READER owns queue/mailbox)
- Supports structured sending/receiving of variable-length messages, which can include (pointers to) data
- Uses all of the IPC services layers along with IPC Configuration & Initialization
- APIs do not change if the message is between two threads:
  - On the same core
  - On two different cores
  - On two different devices
- APIs do NOT change based on transport – only the CFG (init) code
  - Shared memory
  - Multicore Navigator
  - SRIO
MessageQ and Messages

Q How does the writer connect with the reader queue?
   A MultiProc and name server keep track of queue names and core IDs.

Q What do we mean when we refer to structured messages with variable size?
   A Each message has a standard header and data. The header specifies the size of payload.

Q How and where are messages allocated?
   A List utility provides a double-link list mechanism. The actual allocation of the memory is done by HeapMP, SharedRegion, and ListMP.

Q If there are multiple writers, how does the system prevent race conditions (e.g., two writers attempting to allocate the same memory)?
   A GateMP provides hardware semaphore API to prevent race conditions.

Q What facilitates the moving of a message to the receiver queue?
   A This is done by Notify API using the transport layer.

Q Does the application need to configure all these modules?
   A No. Most of the configuration is done by the system. More details later.
MessageQ transactions begin with READER creating a MessageQ.

READER’s attempt to get a message results in a block (unless timeout was specified), since no messages are in the queue yet.
Using MessageQ (2/3)

CorePac 1 - WRITER

```c
MessageQ_open("myQ", ...);
msg = MessageQ_alloc(heap, size, ...);
MessageQ_put("myQ", msg, ...);
```

CorePac 2 - READER

```c
MessageQ_create("myQ", ...);
MessageQ_get("myQ", &msg...);
```

- **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.

**How does the READER get unblocked?**
Once **WRITER** puts `msg` in MessageQ, **READER** is unblocked.

**READER** can now read/process the received message.

**READER** frees message back to Heap.

**READER** can optionally delete the created MessageQ, if desired.
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 automatically by MessageQ.
MessageQ – Miscellaneous Notes

- O/S independent:
  - If one CorePac is running LINUX and using SysLink, the API calls do not change.
  - SysLink is runtime software that provides connectivity between processors (running Linux, SYSBIOS, etc.)

- Messages can be allocated statically or dynamically.

- Timeouts are allowed when a task receives a message.

- User can specify three priority levels:
  - Normal
  - High
  - Urgent
More Information About MessageQ

- All structures and function descriptions can be found within the release:

```
\ipc_U_ZZ_YY_XX\docs\doxygen\html\_message_q_8h.html
```
IPC Services - Notify

- Basic Concepts
- **IPC Services**
  - Message Queue
  - Notify
  - Data Passing
  - Support Utilities
- Setup and Examples
- IPC Transports
- For More Information
Using Notify – Concepts

- In addition to moving MessageQ messages, Notify:
  - Can be used independently of MessageQ
  - Is a simpler form of IPC communication
  - Is used ONLY with shared memory transport
Notify Model

- Comprised of **SENDER** and **RECEIVER**.
- The **SENDER** API requires the following information:
  - Destination (**SENDER** ID is implicit)
  - 16-bit Line ID
  - 32-bit Event ID
  - 32-bit payload (For example, a pointer to message handle)
- The **SENDER** API generates an interrupt (an event) in the destination.
- Based on Line ID and Event ID, the **RECEIVER** schedules a pre-defined call-back function.
During initialization, link the Event ID and Line ID with call back function.

Void Notify_registerEvent(srcId, lineId, eventId, cbFxn, cbArg);

During run time, the Sender sends a notify to the Receiver.

Void Notify_sendEvent(dstId, lineId, eventId, payload, waitClear);

Based on the Sender’s Source ID, Event ID, and Line ID, the call-back function that was registered during initialization is called with the argument payload.
Notify Implementation

Q How are interrupts generated for shared memory transport?
   A The IPC hardware registers are a set of 32-bit registers that generate
      interrupts. There is one register for each core.

Q How are the notify parameters stored?
   A List utility provides a double-link list mechanism. The actual
      allocation of the memory is done by HeapMP, SharedRegion, and
      ListMP.

Q How does the notify know to send the message to the correct
   destination?
   A MultiProc and name server keep track of the core ID.

Q Does the application need to configure all these modules?
   A No. Most of the configuration is done by the system.
Example Callback Function

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

Uint32 recvProcId;
Uint32 seq;

void cbFxn(UInt16 procId, UInt16 lineId, UInt32 eventId, UArg arg, UInt32 payload)
{
    /* The payload is a sequence number. */
    recvProcId = procId;
    seq = payload;
    Semaphore_post(semHandle);
}
```
More Information About Notify

- All structures and function descriptions can be found within the release:

  `\ipc_U_ZZ YY XX\docs\doxygen\html\_notify_8h.html`
IPC Services - Data Passing

- Basic Concepts
- IPC Services
  - Message Queue
  - Notify
  - Data Passing
  - Support Utilities
- Setup and Examples
- IPC Transports
- For More Information
Data Passing Using Shared Memory (1/2)

- When there is a need to allocate memory that is accessible by multiple cores, shared memory is used.
- However, the MPAX register for each core might assign a different logical address to the same physical shared memory address.
- The Shared Region module provides a translation look-up table that resolves the logical/physical address issue without user intervention.
Data Passing Using Shared Memory (2/2)

- Messages are created and freed, but not necessarily in consecutive order:
  - HeapMP provides a dynamic heap utility that supports create and free based on double link list architecture.
  - ListMP provides a double link list utility that makes it easy to create and free messages for static memory. It is used by the HeapMP for dynamic cases.

- To protect the above utilities from race conditions (e.g., multiple cores try to create messages at the same time):
  - GateMP provides hardware semaphore protection.
  - GateMP can also be used by non-IPC applications to assign hardware semaphores.
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 not by itself.

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

![Diagram](image-url)
Data Passing: Additional Transports

- Multicore Navigator
  - Messages are assigned to descriptors.
  - Monolithic descriptors provide a pointer to the message.
- SRIO Type 11 has a TI-developed message protocol that moves data and interrupts between devices.
- Implementation of these transports will be discussed later.
IPC Services – Support Utilities

◆ Basic Concepts
◆ IPC Services
  • Message Queue
  • Notify
  • Data Passing
  • Support Utilities
◆ Setup and Examples
◆ IPC Transports
◆ For More Information
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:
  - Applications using IPC must call `IPC_start()`.
  - In the configuration file, `setupNotify` and `setupMessageQ` specify whether to set up these IPC modules.

**MultiProc**
- Manages processor IDs and core names

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

More utilities...
IPC Support Utilities (2/2)

- 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:

  **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
- For More Information
IPC – Tools/Setup Required

◆ IPC is a package (library) that is installed with the MCSDK. Note: Installing IPC independent of MCSDK is strongly discouraged.

◆ IPC also requires SYS/BIOS (threading) and XDC tools (packaging) – installed with the MCSDK (supported by SYS/BIOS ROV and RTA).

◆ IPC is supported on the latest KeyStone multicore devices (C667x, C665x), as well as C647x and ARM+DSP devices.

What IPC examples exist in the MCSDK?
IPhone – 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
- For More Information
IPC Transports – Intro

◆ 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 all KeyStone SoC devices
  - Uses queues and descriptors plus built-in signaling to transmit/receive data and messages 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...
Data Passing Using Multicore Navigator

The Multicore Navigator:

- Is an innovative packet-based infrastructure that facilitates data movement and multicore control
- Provides a highly efficient inter-core communication mechanism
- Provides H/W queues and packet DMA that are the basic building blocks for IPC

Examples of required init/CFG code are provided with the MCSDK.

Now consider the Multicore Navigator implementation …
IPC Transports – Multicore Navigator (1/3)

- MessageQ_alloc allocates a heapMP buffer for the MessageQ_msg from MSMC.
- The buffer contains the MessageQ_Header plus the data.
- When MessageQ_Put is called with the MessageQ_msg, it also calls the TransportQmss function `NotifyQmss_sendEvent`.

```
TX Core

msg = MessageQ_alloc

MessageQ_put

NotifyQmss_sendEvent

Get a descriptor

Attach ptr/msg to descriptor

Qmss_queuePush
(pushes descriptor directly to RxQueue of remote core)

RX Core

“get the pointer to Msg”

MessageQ_get

QMSS-received descriptor

NotifyDriverQmss_isr

Multicore Navigator

Queue Manager  Packet DMA
```
**IPC Transports – Multicore Navigator (2/3)**

- The TransportQmss function `NotifyQmss_sendEvent`:
  - gets a descriptor
  - Adds the pointer to the MessageQ_msg (located in MSMC and containing the data) to the descriptor.
- `Qmss_queuePush` pushes the descriptor into a queue.
The Interrupt generator queue calls `notifyDriverQmss_isr`.

The ISR schedules a QMSS receive descriptor that sends a message to the queue of Receiver.

**TX Core**
- `msg = MessageQ_alloc`
- `MessageQ_put`
- `NotifyQmss_sendEvent`
- Get a descriptor
- Attach ptr/msg to descriptor
- `Qmss_queuePush`
  (pushes descriptor directly to RxQueue of remote core)

**RX Core**
- "get the pointer to Msg"
- `MessageQ_get`
- QMSS-receive descriptor
- `NotifyDriverQmss_isr`
The **SRIO** (Type 11) 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.
From a messageQ standpoint, the SRIO transport works the same as the QMSS transport. At the transport level, it is also somewhat the same.

The SRIO transport copies the messageQ message into the SRIO data buffer.

It will then pop a SRIO descriptor and put a pointer to the SRIO data buffer into the descriptor.
The transport then passes the descriptor to the SRIO LLD via the Srio_sockSend API.

SRIO then sends and receives the buffer via the SRIO PKTDMA.

The message is then queued on the Receiver side.

Chip V  CorePac W
- msg = MessageQ_alloc
- MessageQ_put(queueId, msg)
- TransportSrio_put
- Srio_sockSend(pkt, dstAddr)
  
SRIO x4

Chip X  CorePac Y
- "get Msg from queue"
- MessageQ_get(queueHndl, rxMsg)
- MessageQ_put(queueId, rxMsg)
- TransportSrio_isr
  
SRIO x4
Configure the Transport Layer

- Most of the transport changes are in the CFG file.
- The XDC tools build the configuration and initialization code. User involvement is only in changing the CFG file.
- Some additional include and initialization is needed in the code.

Now consider the differences between shared memory and Navigator transport from the application perspective …
Configuration: Shared Memory CFG File

```javascript
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');
var Notify = xdc.module('ti.sdo.ipc.Notify');
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');
Notify.SetupProxy = xdc.module(Settings.getNotifySetupDelegate());
MessageQ.SetupTransportProxy = xdc.module(Settings.getMessageQSetupDelegate());

/* Use shared memory IPC */
MessageQ.SetupTransportProxy = xdc.module('ti.sdo.ipc.transports.TransportShmSetup');

Program.global.TRANSPORTSETUP = MessageQ.SetupTransportProxy.delegate$.name;
```
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');
var TransportQmss = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmss');

/* use IPC over QMSS */
MessageQ.SetupTransportProxy = xdc.useModule(Settings.getMessageQSetupDelegate());
var TransportQmssSetup = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmssSetup');
MessageQ.SetupTransportProxy = TransportQmssSetup;

TransportQmssSetup.descMemRegion = 0;
Program.global.descriptorMemRegion = TransportQmssSetup.descMemRegion;

Program.global.numDescriptors = 8192;
Program.global.descriptorSize = cacheLineSize; // multiple of cache line size

TransportQmss.numDescriptors = Program.global.numDescriptors;
TransportQmss.descriptorIsInSharedMem = true;
TransportQmss.descriptorSize = Program.global.descriptorSize;
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');
var Ipc = xdc.useModule('ti.sdo.ipc.Ipc');
var TransportSrio = xdc.useModule('ti.transport.ipc.srio.transports.TransportSrio');

/* use IPC over SRIO */
MessageQ.SetupTransportProxy = xdc.useModule( Settings.getMessageQSetupDelegate() );
var TransportSrioSetup =
    xdc.useModule('ti.transport.ipc.srio.transports.TransportSrioSetup');
MessageQ.SetupTransportProxy = TransportSrioSetup;

TransportSrioSetup.messageQHeapId = 1; /* Sized specifically to handle receive side packets. */
    /* Heap should */
    /* not be used by any other application or module */
TransportSrioSetup.descMemRegion = 0;
TransportSrioSetup.numRxDescBuffs = 256; /* Should be sized large enough so that multiple */
    /* packets can be queued on receive side and still */
    /* have buffs available for incoming packets */
Configuration: Navigator Initialization (1/2)

Additional Include:

/* QMSS LLD*/
#include <ti/drv/qmss/qmss_drv.h>
#include <ti/drv/qmss/qmss_firmware.h>

/* CPPI LLD */
#include <ti/drv/cppi/cppi_drv.h>
#include <ti/transport/ipc/examples/common/bench_common.h>
#include <ti/transport/ipc/qmss/transports/TransportQmss.h>
Configuration: Navigator Initialization (2/2)

Add an initialization routine:

```c
Int main(Int argc, Char* argv[])
{

    if (selfId == 0)
    {
        /* QMSS, and CPPI system wide initializations are run on
         * this core */
        result = systemInit();
        if (result != 0)
        {
            System_printf("Error (%d) while initializing QMSS\n", result);
        }
    }

    // Note that systemInit is provided by TI
```
## IPC Transport Details

### Throughput (Mb/second)

<table>
<thead>
<tr>
<th>Message Size (Bytes)</th>
<th>Shared Memory</th>
<th>Multicore Navigator</th>
<th>SRIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>23.8</td>
<td>34.6</td>
<td>4.1</td>
</tr>
<tr>
<td>256</td>
<td>125.8</td>
<td>184.4</td>
<td>21.2</td>
</tr>
<tr>
<td>1024</td>
<td>503.2</td>
<td>737.4</td>
<td>-</td>
</tr>
</tbody>
</table>

### Benchmark Details
- IPC Benchmark Examples from MCSDK
- CPU Clock – 1 GHz
- Header Size – 32 bytes
- SRIO – Loopback Mode
- Messages allocated up front
<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shared Memory</strong></td>
<td>• Single device only</td>
</tr>
<tr>
<td>• Simplest to implement</td>
<td>• Requires Notify module and API call</td>
</tr>
<tr>
<td>• Moderate throughput</td>
<td>• if doorbell required</td>
</tr>
<tr>
<td>• Possible contention</td>
<td>• with other tasks</td>
</tr>
<tr>
<td>with other tasks</td>
<td>• using the same shared memory</td>
</tr>
<tr>
<td><strong>Multicore Navigator</strong></td>
<td>• Single device only</td>
</tr>
<tr>
<td>• Highest throughput</td>
<td></td>
</tr>
<tr>
<td>• Dedicated resources</td>
<td></td>
</tr>
<tr>
<td>• Consumes least CPU cycles</td>
<td></td>
</tr>
<tr>
<td>• Interrupt generated when</td>
<td></td>
</tr>
<tr>
<td>data is available</td>
<td></td>
</tr>
<tr>
<td><strong>SRIO</strong></td>
<td>• Lowest throughput</td>
</tr>
<tr>
<td>• Can be used across devices</td>
<td></td>
</tr>
</tbody>
</table>
For More Information

• For more information, refer to the BIOS MCSDK User Guide on the Embedded Processors Wiki.

• For questions regarding topics covered in this training, visit the support forums at the TI E2E Community website.