Introduction to Inter-Processor Communication (IPC) for KeyStone and Sitara™ Devices
Agenda

• IPC 3.x Overview
  – MessageQ Architecture
  – Modules

• Library and daemon

• RPMsg and Resource Table

• Examples

• Benchmarking

• For More Information
IPC 3.x Overview: Architecture
Multicore Processing

TI’s KeyStone and Sitara AM57x devices are built to meet the complex processing needs of modern embedded products. While the number of diverse processors and rich connectivity peripherals on these devices provides flexibility, it also challenges software developers to fully exploit the multicore processing potential.
Why IPC?

To shorten the software development cycle and accelerate the time to market, TI provides a solution for inter-processor communication between homogenous and heterogeneous processor on the device with various transports.
IPC 3.x Overview

- IPC 3.x is an evolution of the IPC product in the TI Processor SDK (Software Development Kit):
  - Abstracts the lower layer of processor fabric connection
  - Provides a set of modules and APIs to facilitate communication:
    - Between cores in a multi-processor environment (inter-processor)
    - To other threads on same processor (inter-process)
    - To peripherals (inter-device)

- IPC can be used to communicate in a multi-processor environment with:
  - Other threads on the same processor
  - Threads on other processor running SYS/BIOS
  - Threads on other processor running an HLOS (e.g., Linux, QNX)
Current IPC Releases

IPC releases newer than IPC 3.40 are now distributed as part of various Processor SDKs. See the Processor SDK for RTOS download pages for a list of supported platforms.
IPC Open Source Project

IPC is an open source project currently managed with git and maintained at http://git.ti.com/ipc/ipcdev. Full source browsing, including all changes, are available at http://git.ti.com/cgit/cgit.cgi/ipc/ipcdev.git.
IPC MessageQ Software Architecture: AK2H

Device 1

ARM Cortex-A15 Linux Thread/Process A

Linux IPC MessageQ

TransportLinuxSrio
TransportLinuxQmss
TransportRpmsg

SYS/BIOS IPC MessageQ

TransportRpmsg
TransportQmss
TransportSrio

DSP X

Device 2

ARM Linux

Linux IPC MessageQ

TransportLinuxQmss

ARM Linux Thread/Process B

TransportLinuxSrio

DSP X

Linux IPC MessageQ

TransportSrio
IPC MessageQ Software Architecture: AM57x

Device 1

ARM Cortex-A15 Linux
Thread/Process A

DSP X / IPU M4

Linux IPC MessageQ

SYS/BIOS IPC MessageQ

TransportRpmqg

Linux Kernel
IPC 3.x Overview: Modules
**IPC SYS/BIOS Modules**

- **ti.sdo.ipc**
  - Ipc
  - MessageQ
  - HeapMemMP
  - GateMP

- **ti.sdo.utils**
  - Notify
  - SharedRegion
  - HeapBufMP
  - MultiProc
  - NameServer
  - List
IPC HLOS Modules

- Ipc
- NameServer
- MessageQ
- MultiProc
- GateMP
IPC Modules Defined

- **Ipc** initializes and synchronizes IPC.
- **MessageQ** sends and receives messages.
- **NameServer** manages the distributed name/value database.
- **Notify** sends and receives event notifications.
- **MultiProc** manages processor identification.
- **SharedRegion** manages shared memory address translation.
- **GateMP** (Multi-Processor Gate) enforces local and remote context protection.
- **Heap*MP** allocates multi-processor memory.
- **ListMP** creates lists of objects, implemented as a doubly-linked list.
Ipc Module

- Ipc is used to initialize IPC and synchronize with other processors.
- An application that uses IPC APIs--such as MessageQ--must include the Ipc module header file and call `Ipc_start()` before any calls to IPC modules.
- Two startup protocols:
  - `Ipc.ProcSync_ALL` option indicates all processors start at same time.
  - `Ipc.ProcSync_PAIR` option indicates host processor starts first.
- Configuration:
  - `Ipc.procSync` property configures startup protocol.
  - When using `Ipc.ProcSync_ALL`, `Ipc_attach` is called internally from `Ipc_start`, application should never call `Ipc_attach`. 
MessageQ Module

- MessageQ module supports the structured sending and receiving of variable length messages.
- Each message queue has one reader and can have many writers. A thread may read from or write to multiple message queues. Timeouts are allowed when receiving messages.
- **Reader**: The single reader thread calls `MessageQ_create()`, `MessageQ_get()`, `MessageQ_free()`, and `MessageQ_delete()`.
- **Writer**: Writer threads call `MessageQ_open()`, `MessageQ_alloc()`, `MessageQ_put()`, and `MessageQ_close()`.
- **Priorities**:
  - `MessageQ_NORMALPRI = 0`
  - `MessageQ_HIGHPRI = 1`
  - `MessageQ_URGENTPRI = 3`
typedef struct {
    Bits32     reserved0;  /*!< reserved for List.elem->next    */
    Bits32     reserved1;  /*!< reserved for List.elem->prev     */
    Bits32     msgSize;    /*!< message size                      */
    Bits16     flags;      /*!< bitmask of transport id, priority */
    Bits16     msgId;      /*!< message id                        */
    Bits16     dstId;      /*!< destination queue id             */
    Bits16     dstProc;    /*!< destination processor id         */
    Bits16     replyId;    /*!< reply id                           */
    Bits16     replyProc;  /*!< reply processor                    */
    Bits16     srcProc;    /*!< source processor                   */
    Bits16     heapId;     /*!< heap id                            */
    Bits16     seqNum;     /*!< sequence number                    */
    Bits16     reserved;   /*!< reserved                           */
} MessageQ_MsgHeader;
Using MessageQ

IPC NameServer

MessageQ_open
MessageQ_alloc
MessageQ_create
MessageQ_Msg
MessageQ_MsgHeader
Data
MessageQ_Msg
MessageQ_MsgHeader
Data
QueueId
MessageQ_Handle
MessageQ_Msg
MessageQ_MsgHeader
Data
MessageQ_Msg
MessageQ_MsgHeader
Data
QueueId
MessageQ_Handle
Msg List
MessageQ_get
MessageQ_put
Transport_put
Transport_isr
Transport
Medium
Processor X (Writer)
Processor Y (Reader)

MessageQ_data
Movement

QueueId

Texas Instruments
• A GateMP instance can be used to enforce both local and remote context protection.

• Entering a GateMP can prevent preemption by another thread/process running on the same processor and simultaneously prevent a remote processor from entering the same gate. GateMPs are typically used to protect reads/writes to a shared resource, such as shared memory.

```c
IArg key;
/* Enter the gate */
key = GateMP_enter(gateHandle);
...
/* Leave the gate */
GateMP_leave(gateHandle, key);
```
MultiProc Module

• IPC MultiProc module assigns unique ID to each processor in system
• Cluster implementation to support multi-device systems
  – Each device must have mutually exclusive base cluster ID
  – All devices must have a MultiProc mapping such that each processor ID is unique

• A system config can be simple, as shown in this example:

```javascript
/* describe the processors in the system */
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
//MultiProc.baseIdOfCluster = 0;
//MultiProc.numProcessors = 2;
MultiProc.setConfig("DSP1", ["HOST", "DSP1"]);
```

• Or a system config can be complex, as shown on the following slide ...
MultiProc Mapping

**Device A MultiProc Mapping**
- numProcessors = 23
- numProcsInCluster = 9
- baseIdOfCluster = 0

**Device B MultiProc Mapping**
- numProcessors = 23
- numProcsInCluster = 9
- baseIdOfCluster = 9

**Device C MultiProc Mapping**
- numProcessors = 23
- numProcsInCluster = 5
- baseIdOfCluster = 18
Library and daemon
IPCC Library Build: products.mak

• The current IPCC release supports multiple devices. The IPCC libraries for each device are NOT pre-built.

• IPCC release contains a **products.mak** file that specifies the necessary paths and options to build IPCC for the various supported OS.

Edit **products.mak** and set the following variables:

- **Variables used by both Linux-side and BIOS-side build scripts:**
  - **PLATFORM** (Optional) identifies the device to build.

- **Variables used by Linux-side build scripts:**
  - **TOOLCHAIN_INSTALL_DIR** identifies the path to the ARM Linux cross-compiler toolchain.
  - **TOOLCHAIN_LONGNAME** defines the long name of the device toolchain (e.g., **arm--linux-gnueabihf**).
  - **KERNEL_INSTALL_DIR** identifies the location of the Linux kernel installation.
  - **DRM_PREFIX** (Optional)
  - **CMEM_INSTALL_DIR** (Optional)

- **Variables used by BIOS-side build scripts:**
  - **XDC_INSTALL_DIR** identifies the path to the TI XDCTools installation (e.g., `c:/ti/xdctools_<version>`).
  - **BIOS_INSTALL_DIR** identifies the path to the TI SYS/BIOS installation (e.g., `c:/ti/bios_<version>`).
  - **ti.targets.<device target and file format>** identifies the path to the TI toolchain for the device.
    (e.g., `c:/ti/CCS/ccsbase/tools/compiler/c6000_<version>`)
DEPOT = /home/user/ti
PLATFORM = DRA7XX
DESTDIR = /home/user/ti/ipc_3_42_00_02/generate

TOOLCHAIN_LONGNAME = arm-linux-gnueabihf
TOOLCHAIN_INSTALL_DIR = /home/user/gcc-linaro-4.9-2015.05-x86_64_arm-linux-gnueabihf
TOOLCHAIN_PREFIX = $(TOOLCHAIN_INSTALL_DIR)/bin/$'(TOOLCHAIN_LONGNAME)-

KERNEL_INSTALL_DIR = /home/user/ti/ti-processor-sdk-linux-am57xx-evm-02.00.02.11/board-support/linux-4.1.18+gitAUTOINC+01c1359baa-g01c1359

CMEM_INSTALL_DIR = /home/user/ti/ludev

XDC_INSTALL_DIR = $(DEPOT)/xdctools_3_32_00_06_core
BIOS_INSTALL_DIR = $(DEPOT)/bios_6_45_01_26
CCS = /home/user/ti/ccsv6_1_2/ccsv6/tools/compiler

ti.targets.elf.C66 = $(CCS)/ti-cgt-c6000_8.1.0
ti.targets.arm.elf.M4 = $(CCS)/ti-cgt-arm_5.2.5
gnu.targets.arm.A15F = $(CCS)/gcc-arm-none-eabi-4_8-2014q3
IPC Library Build Commands

• Linux-side build:
  – The IPC package provides a GNU makefile (ipc-linux.mak) to configure the Linux-side build, using the options and component paths set in the products.mak file. To configure the build, issue the following command:

    <buildhost> make -f ipc-linux.mak config

  – Or to specify a platform:

    <buildhost> make -f ipc-linux.mak config PLATFORM=66AK2G

  – And then
    <buildhost> make; sudo make install

• SYS/BIOS-side build:
  – After editing products.mak, issue the following command:

    <buildhost> make -f ipc-bios.mak all
• IPC provides system-wide services across multiple applications, and utilizes low-level system hardware (e.g. interrupts and shared memory). To facilitate these services, IPC uses a user-space daemon (LAD) and several Linux kernel device drivers.

• At startup, the daemon creates a FIFO (named pipe) to listen for connection requests from other user-mode clients. When a connection request comes in, the daemon opens a dedicated FIFO for sending responses to the client.

• LAD resides on the target filesystem (typically in /usr/bin/) and typically starts running in the background during kernel bootup. LAD logs the IPC state in the file lad.txt in the /var/tmp/LAD/ directory.
 IPC Daemon Initialization

[546.947163] Retrieving command...
[546.947209] LAD_CONNECT:

[546.947217] client FIFO name = /tmp/LAD/1216
[546.947223] client PID = 1216
[546.947233] assigned client handle = 0
[546.947285] FIFO /tmp/LAD/1216 created
[546.948337] FIFO /tmp/LAD/1216 opened for writing
[546.948371] sent response
[546.948378] DONE

[546.948383] Retrieving command...
[546.948436] Sending response...
[546.948469] Retrieving command...
[546.948481] LAD_MULTIPROC_GETCONFIG: calling MultiProc_getConfig()...
[546.948491] MultiProc_getConfig() - 5 procs
[546.948497] # processors in cluster: 5
[546.948502] cluster baseId: 0
[546.948513] ProcId 0 - "HOST"
[546.948519] ProcId 1 - "IPU2"
[546.948525] ProcId 2 - "IPU1"
[546.948530] ProcId 3 - "DSP2"
[546.948535] ProcId 4 - "DSP1"
[546.948540] status = 0
[546.948545] DONE
[546.948549] Sending response...
[546.948669] Retrieving command...
RPMsg and Resource Table
**RPMsg**

- **remoteproc** is a generic kernel component that manages remote processors and enables users access to these remote processors. The main functionalities implemented by remoteproc:
  - Device Loading & Bootup
  - Power Management
  - Exception Management
  - Error Recovery.

- **rpmsg** is a virtio-based messaging bus that allows kernel drivers to communicate with remote processors available on the system.

```
root@k2g-evm:~# lsmod
Module                  Size  Used by
rpmsg_proto             6634  0
virtio_rpmsg_bus        11070 1 rpmsg_proto
omap_remoteproc         9771  1
```


Resource Table

- Resource table is a Linux construct used to inform the remoteproc driver about the remote processor’s available resources, and typically refers to memory and local peripheral registers.

- When a remote processor image is loaded, the Linux kernel remoteproc framework will parse the system resources defined in resource table, which is linked into the remote processor image. Also, the remoteproc allocates rpmsg vring buffers, trace buffer, and configure MMUs according to the resource table.

- See default ./packages/ti/ipc/remoteproc/rsc_table_xyz_dsp.h or ./packages/ti/ipc/remoteproc/rsc_table_xyz_ipu.h

- To customize resource table entries, refer to: http://processors.wiki.ti.com/index.php/IPC_Resource_customTable
IPC Examples
IPC Examples

- A few examples for HLOS and SYS/BIOS running on various devices are available in the IPC/examples directory. For instance:
  - examples/DRA7XX_linux_elf$ ls
    - ex02_messageq ex12_mmrpc ex41_forwardmsg ex68_power
  - examples/DRA7XX_bios_elf$ ls
    - ex01_hello ex02_messageq ex11_ping ex13_notifypeer

- In addition, the IPC release contains source files for functional testing.
  - /packages/ti/ipc/tests/
    - gatemp.c, messageq_multicore.c, messageq_single.c, ping_rpm.c ...
BIOS.addUserStartupFunction('&IpcMgr_ipcStartup');

......
var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');
MessageQ.registerHeapMeta(msgHeap, 0);

/* Setup MessageQ transport */
var VirtioSetup = xdc.useModule('ti.ipc.transports.TransportRpmsgSetup');
MessageQ.SetupTransportProxy = VirtioSetup;

/* Setup NameServer remote proxy */
var NameServer = xdc.useModule("ti.sdo.utils.NameServer");
var NsRemote = xdc.useModule("ti.ipc.namesrv.NameServerRemoteRpmsg");
NameServer.SetupProxy = NsRemote;

/* Enable Memory Translation module that operates on the BIOS Resource Table */
var Resource = xdc.useModule('ti.ipc.remoteproc.Resource');
Resource.customTable = true;
GateMP Examples

./packages/ti/ipc/tests/gatemppapp.cfg

```javascript
/* GateMP host support */
var GateMP = xdc.useModule('ti.sdo.ipc.GateMP');
GateMP.hostSupport = true;
/* shared region configuration */
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
/* configure SharedRegion #0 (IPC) */
var SR0Mem = Program.cpu.memoryMap['SR_0'];
SharedRegion.setEntryMeta(0,
    new SharedRegion.Entry({
        name: "SR0",
        base: SR0Mem.base,
        len: SR0Mem.len,
        ownerProcId: MultiProc.getIdMeta("DSP1"),
        cacheEnable: true,
        isValid: true
    })
);
/*ownerProcId. MultiProc ID of the processor that manages this region. If an owner is specified, the
owner creates a HeapMemMP instance at runtime. The other cores open the same HeapMemMP instance. */
isValid. Boolean to specify whether the region is valid (accessible) or not on this processor. */
```

35
Examples: Image Load and Debug

• The remote processor images are expected in the directory /lib/firmware/ and loaded during Linux kernel bootup via remoteproc module, e.g. on AM57x

<table>
<thead>
<tr>
<th>Core</th>
<th>Binary on the Host</th>
<th>Binary on the Target File System</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPU1</td>
<td>server_ipu1.xem4</td>
<td>/lib/firmware/dra7-ipu1-fw.xem4</td>
</tr>
<tr>
<td>IPU2</td>
<td>server_ipu2.xem4</td>
<td>/lib/firmware/dra7-ipu2-fw.xem4</td>
</tr>
<tr>
<td>DSP1</td>
<td>server_dsp1.xe66</td>
<td>/lib/firmware/dra7-dsp1-fw.xe66</td>
</tr>
<tr>
<td>DSP2</td>
<td>server_dsp2.xe66</td>
<td>/lib/firmware/dra7-dsp2-fw.xe66</td>
</tr>
</tbody>
</table>

• To check the remote processor image boot log:

<table>
<thead>
<tr>
<th>Core</th>
<th>Debug Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPU1</td>
<td>/sys/kernel/debug/remoteproc/remoteproc0/trace0</td>
</tr>
<tr>
<td>IPU2</td>
<td>/sys/kernel/debug/remoteproc/remoteproc1/trace0</td>
</tr>
<tr>
<td>DSP1</td>
<td>/sys/kernel/debug/remoteproc/remoteproc2/trace0</td>
</tr>
<tr>
<td>DSP2</td>
<td>/sys/kernel/debug/remoteproc/remoteproc3/trace0</td>
</tr>
</tbody>
</table>
MessageQBench

• MessageQBench utilizes the IPC stack to communicate from the main processor to the remote processor via the MessageQ interface and measures the time required to complete.

```
target# /usr/bin/MessageQBench
- The application will exchange 1000 messages with a payload of 8 by default with an average round trip time per message. The following is the usage parameters for the application.
- Usage: ./bin/MessageQBench [<numLoops>] [<payloadSize>] [<ProcId>] Defaults: numLoops: 1000; payloadSize: 8, ProcId: 1
- DSP image: messageq_single.xe66
```

• Additional processor synchronization mechanism is required to measure the unidirectional latency.
For More Information:

- IPC Downloads: http://downloads.ti.com/dsps/dspsw/sdo_sb/targetcontent/ipc/
- For questions regarding topics covered in this training, visit the support forums at the TI E2E Community website.