Building Blocks for PRU Broad Market Success

Module 4 – Linux Drivers

This session covers the Linux drivers to enable the PRU-ICSS sub-system.

Author: Texas Instruments®, Sitara™ ARM® Processors

October 2014
Creative Commons Attribution-ShareAlike 3.0 (CC BY-SA 3.0)

You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work
- to make commercial use of the work

Under the following conditions:

**Attribution** — You must give the original author(s) credit

**Share Alike** — If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

With the understanding that:

**Waiver** — Any of the above conditions can be waived if you get permission from the copyright holder.

**Public Domain** — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

**Other Rights** — In no way are any of the following rights affected by the license:

**Notice** — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this webpage.

CC BY-SA 3.0 License: http://creativecommons.org/licenses/by-sa/3.0/us/legalcode
ARM + PRU SoC Software Architecture

**ARM Subsystem**
- Cortex-A
- L1 Instruction Cache
- L1 Data Cache
- L2 Data Cache

**Programmable Real-Time Unit (PRU) Subsystem**
- PRU0 (200MHz)
- PRU1 (200MHz)
- Shared RAM
- Instruction RAM
- Data RAM

**Interconnect**
- INTC
- Peripherals

**L3 Interconnect**
- Shared Memory
- Peripherals

**L4 Interconnect**
- Peripherals
- GP I/O

---

Linux

Firmware
What do we need Linux to do?

• Load the Firmware
• Manage resources (memory, CPU, etc.)
• Control execution (start, stop, etc.)
• Send/receive messages to share data
• Synchronize through events (interrupts)
• These services are provided through a combination of remoteproc/rpmsg + virtio transport frameworks
Linux Drivers

Remoteproc
PRU remoteproc Stack

- The remoteproc framework allows different platforms/architectures to control (power on, load firmware, power off) remote processors while abstracting any hardware differences
  - Does not matter what OS (if any) the remote processor is running
- Kernel documentation available in /Documentation/remoteproc.txt
Why Use Remoteproc?

• It already exists
  – Easier to reuse an existing framework than to create a new one

• Easy to implement
  – Requires only a few custom low-level handlers in the Linux driver for a new platform

• Mainline-friendly
  – The core driver has been in mainline for a couple years

• Fairly simple interface for powering up and controlling a remote processor from the kernel

• Enables us to use rpmsg framework for message sharing
How to Use Remoteproc

• Load driver manually or build into kernel
  – Use menuconfig to build into kernel or create a module

• Probe() function automatically looks for firmware in /lib/firmware directory in target filesystem
  – rproc_pru0_fw or rproc_pru1_fw for core 0 and 1, respectively

• Interrupts passed between host application and PRU firmware
  – Application effectively registers to an interrupt
Creating a New Node

• A pruss node is created in the root am33xx Device Tree file

• This passes information about the subsystem on AM335x into the PRU rproc driver during probe() function
  – Primarily register offsets, clock speed, and other non-changing information

• Requires little-to-no interaction on a case-by-case basis
  – All project-dependent settings are configured in Resource Table
Understanding the Resource Table

• What is a Resource Table?
  – A Linux construct used to inform the remoteproc driver about the remote processor’s available resources
  – Typically refers to memory, local peripheral registers, etc.
  – Firmware-dependent

• Why do I need one?
  – Allows the driver to remain generic while still supporting a number of different, often unique remote processors
    • Is flexible enough to allow for the creation of a custom resource type
  – Is not strictly required as the driver can fall back on defaults
    • This severely limits it as the driver may not understand how the PRU firmware wishes to map/handle interrupts
Configuring the Resource Table

• Most projects will not need to touch anything beyond the interrupt and vring configuration

• Typically only need to modify up to three things
  – Event-to-channel mapping
  – Channel-to-host mapping
  – Number and location of vrings
Linux Drivers

Rpsmsg
Application uses /dev/rpmsg-prux interface to send messages

Client drivers specifically for PRU core

Linux frameworks
What Is Rpmsg?

- Rpmsg is a Linux framework designed to allow for message passing between the kernel and a remote processor.

- Kernel documentation available in /Documentation/rpmsg.txt.

- Virtio is a virtualized I/O framework.
  - We will use it to communicate with our virtio device (vdev).
    - There are several ‘standard’ vdevs, but we only use virtio_ring.
    - Virtio_ring (vring) is the transport implementation for virtio.
  - The host and PRU will communicate with one another via the virtio_rings (vrings) and “kicks” for synchronization.
Why Use Rpmsg?

• It already exists
  – Easier to reuse an existing framework than to create a new one

• Mainline-friendly
  – The core driver has been in mainline for at least a couple years

• Ties in with existing remoteproc driver framework

• Fairly simple interface for passing messages between User Space and the PRU firmware

• Allows developers to expose the virtual device (PRU) to User Space or other frameworks

• Provides scalability for integrating individual PRU peripherals with the respective driver sub-systems.
How to Use pru-rpmsg Generic Client Driver

• User Space applications use /dev/rpmsg-prux interface to pass messages to and from PRU

• An example Generic Client Driver is under development
Linux Drivers

Custom Function Drivers
Custom rpmsg Client Drivers

- User Space applications use /dev/rpmsg-pru0 interface to pass messages to and from PRU

- Create different rpmsg client drivers to expose the PRU as other interfaces
  - Firmware based UART, SPI, etc.
  - Allows true PRU firmware enhanced Linux devices
Custom Function Drivers

• Some users may wish to use the PRU as another Linux Device (e.g. as another UART /dev/ttyO6)
  – This will require a custom Linux driver to work in tandem with rproc/rpmsg
  – Customer at this time will have to develop this custom driver themselves or work with a third party to do so

• TI is not initially launching any support for this mechanism
  – We have several different targets in mind (UART, I2C, I2S, SPI, etc…), but these will not be available at release
  – No target date available today, but we will start evaluating after broad market PRU launch
Thank you
Backup Slides
Virtio & Vring

- Virtio is a virtualized I/O framework
  - We will use it to communicate with our virtio device (vdev)
    - There are several ‘standard’ vdevs, but we only use virtio_ring
  - The host and PRU will communicate with one another via the virtio_rings (vrings)
Virtio & Vring

- Virtio_ring (vring) is the transport implementation for virtio

- A vring consists of three primary parts:
  - A descriptor array
  - The available ring
  - The used ring

- In our case the vring contains our list of buffers used to pass data between the host and PRU cores via rpmsg
Virtio & Vring

- The descriptor array
  - This is where the guest chains together length/address pairs
  - Address is the guest-physical address of the buffer
  - Length is the size of the buffer
  - There are two flags: R/W, and whether or not Next is valid
  - Next is used for chaining
    - This is generally used for packet processing (LANs)

<table>
<thead>
<tr>
<th>Address</th>
<th>Length</th>
<th>Flags</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Virtio & Vring

• The available ring
  – This where the guest (PRU) indicates which descriptors are ready for use
  – Consists of a free-running index, an interrupt suppression flag, and an array of indices into the descriptor table (representing the heads of available buffers)
  – Available buffers do not have to be contiguous in memory
Virtio & Vring

• The used ring
  – This is where the host indicates which descriptors chains it has used
  – The used ring is similar to the available ring except it is written by the host as descriptor chains are consumed
  – Consists of a free-running index, an interrupt suppression flag, and an array of indices into the descriptor table (representing the heads of used buffers)
  – Used buffers do not need to be contiguous in memory