SITARA™ ARM® PROCESSORS

BOOt camp

Enhancing Real-time Capabilities with the PRU

Love Linux. Need hard real-time? Seems like these might not go together, but with the PRU (Programmable Real-time Unit) and a Cortex-A running Linux, you might be surprised.

Author: Ron Birkett, Sitara™ ARM® Processors

October 2014

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

creative commons

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 web page.

CC BY-SA 3.0 License:
http://creativecommons.org/licenses/by-sa/3.0/us/legalcode

CC BY-SA

Discussion Topics

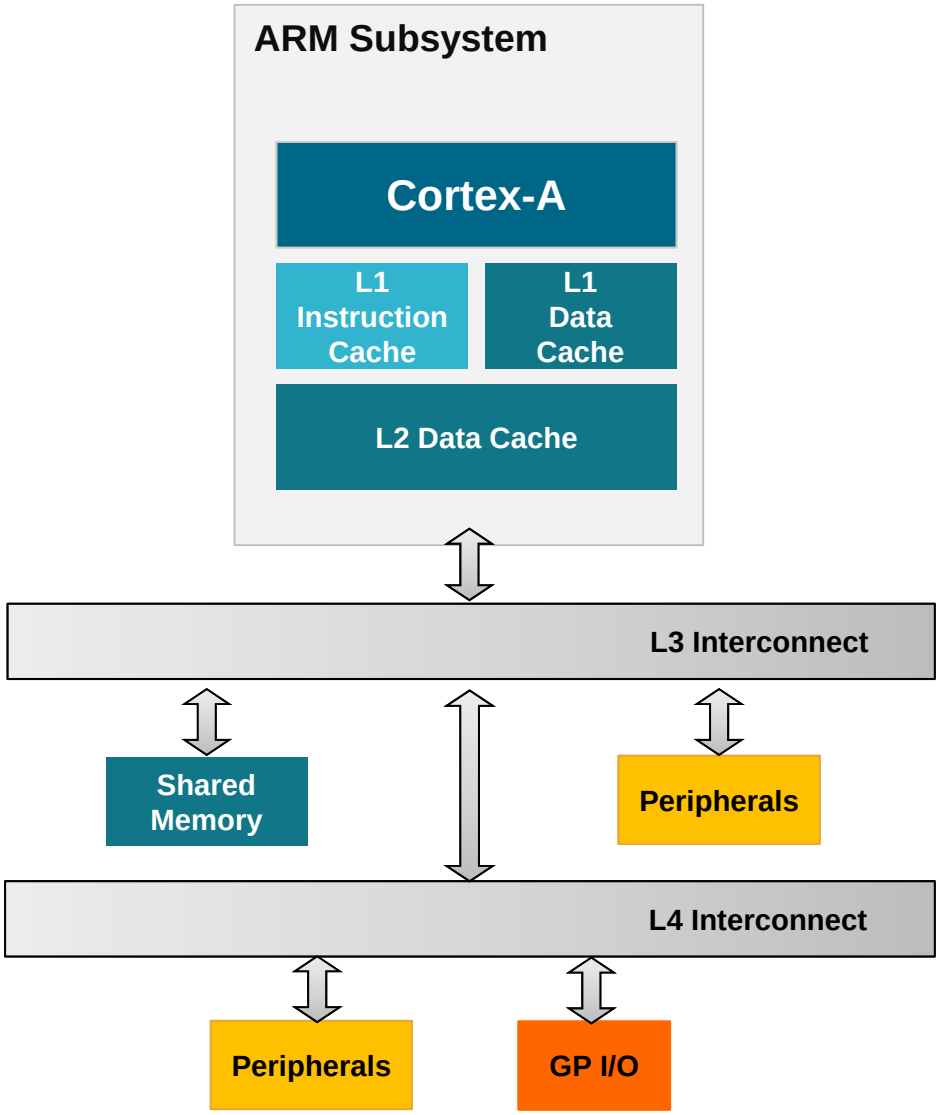
- What is Real-Time?
- PRU Hardware Overview
- Linux ←→ PRU
- Summary

What is “Real-time”?

- “Real-time” is relative
- Real-time programs must guarantee response within strict time constraints (i.e. “deadlines”)
- Real-time deadlines must be met, regardless of system load
- High Performance ≠ Real-time
- For our definition, we’ll constrain “real-time” to deterministic, ultra-low-latency response

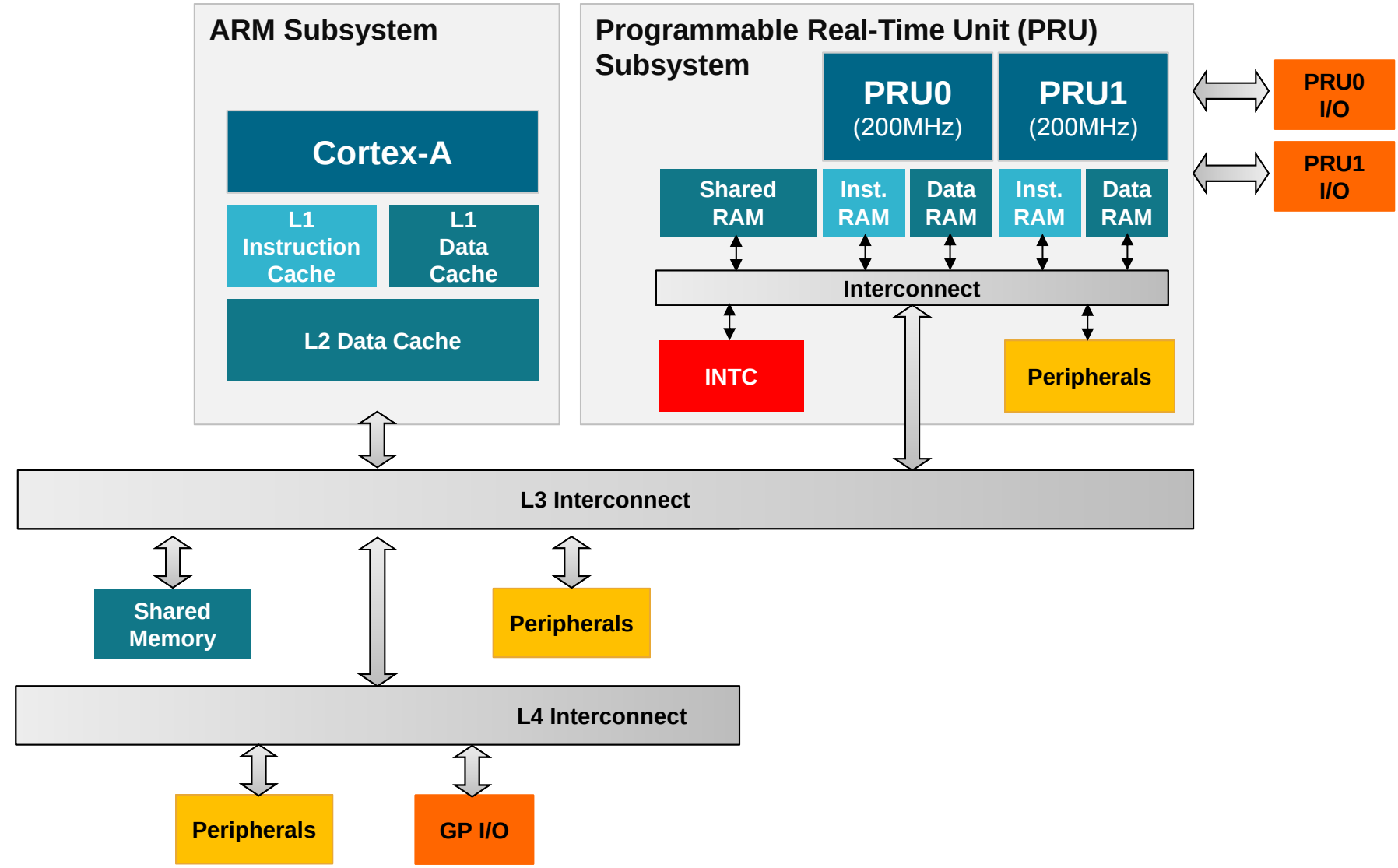
PRU Hardware Overview

ARM SoC Architecture



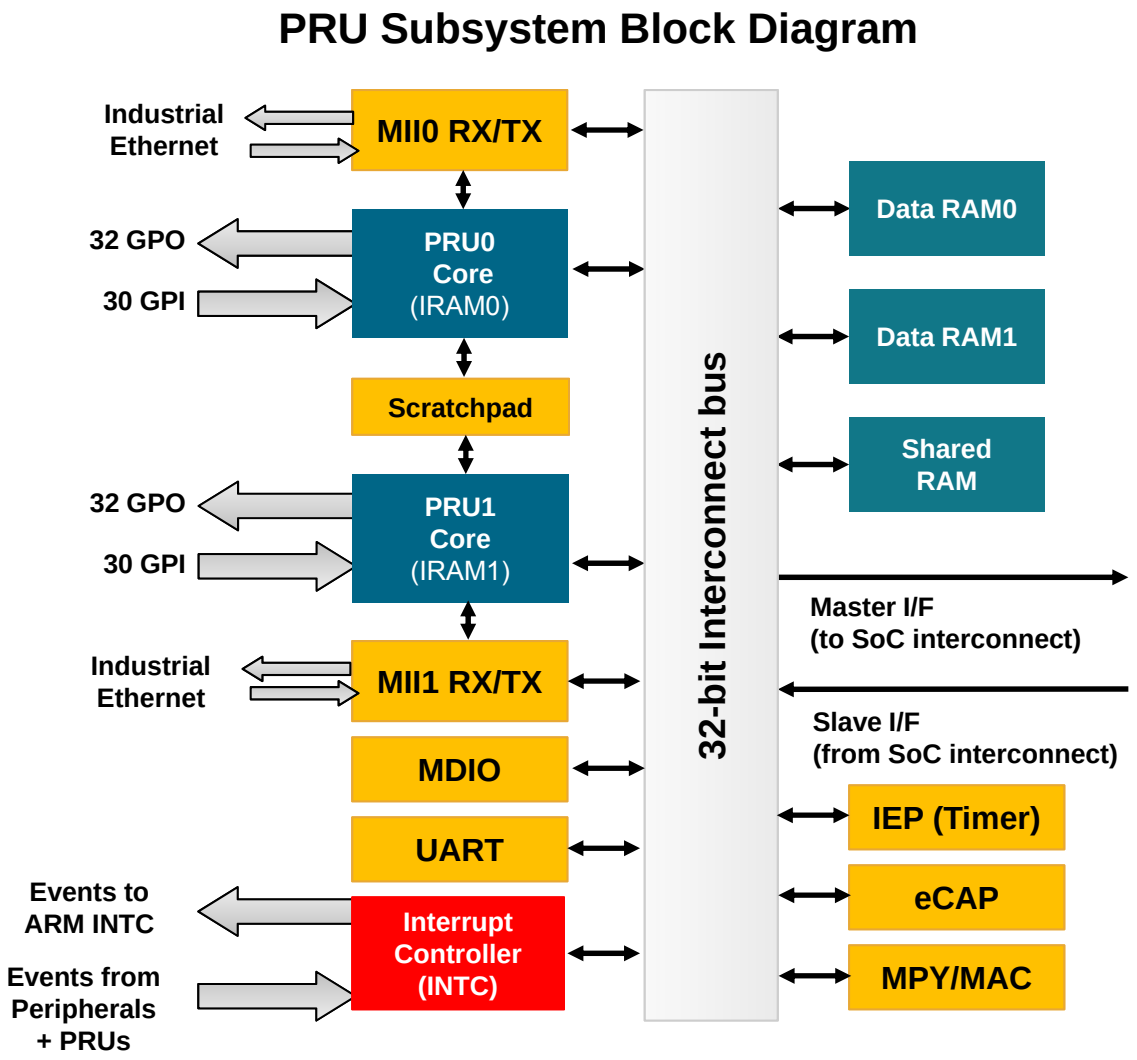
- L1 D/I caches:
  - Single cycle access
- L2 cache:
  - Min latency of 8 cycles
- Access to on-chip SRAM:
  - 20 cycles
- Access to shared memory over L3 Interconnect:
  - 40 cycles

ARM + PRU SoC Architecture

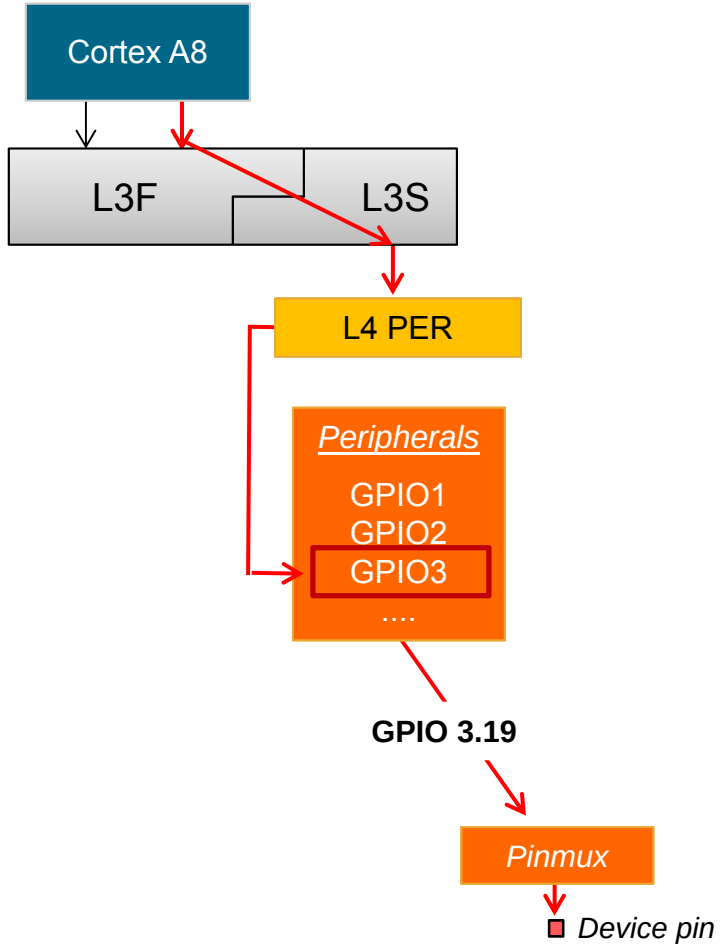


Programmable Real-Time Unit (PRU) Subsystem

- Programmable Real-Time Unit (PRU) is a low-latency microcontroller subsystem
- Two independent PRU execution units
  - 32-Bit RISC architecture
  - 32 General Purpose Registers
  - 200MHz – 5ns per instruction
  - Single cycle execution - No pipeline
  - Dedicated instruction and data RAM per core
  - Shared RAM
- Includes Interrupt Controller for system event handling
- Fast I/O interface
  - Up to 30 inputs and 32 outputs on external pins per PRU unit

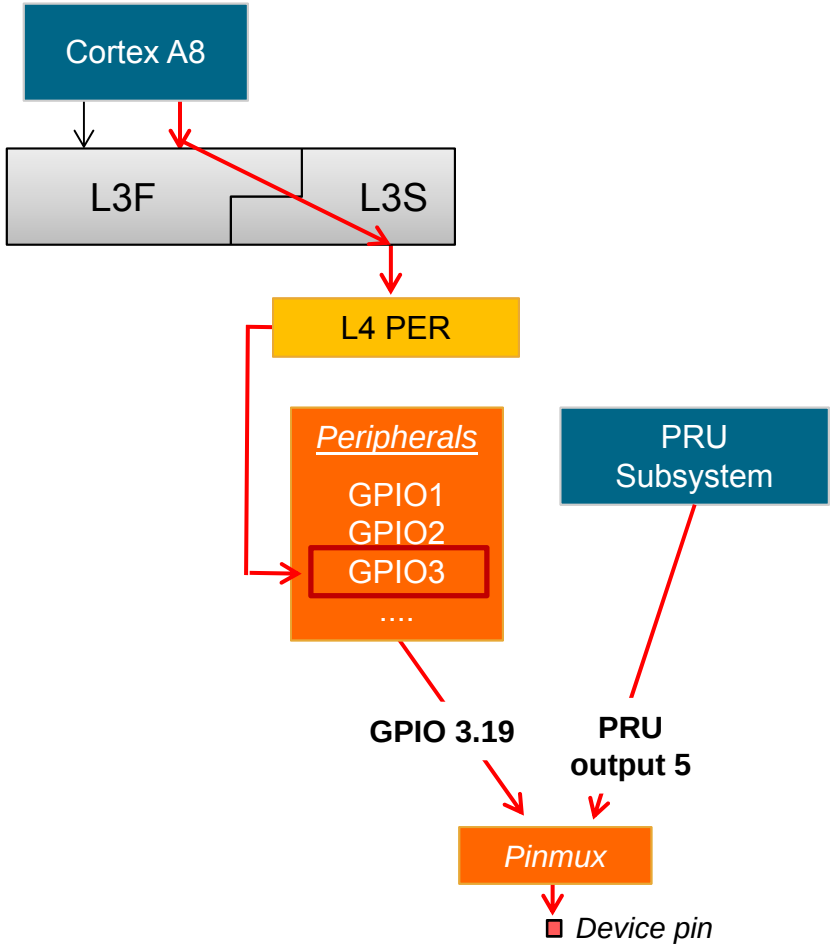


Fast I/O Interface



Fast I/O Interface

- Reduced latency through direct access to pins
  - Read or toggle I/O within a single PRU cycle
  - Detect and react to I/O event within two PRU cycles
- Independent general purpose inputs (GPIs) and general purpose outputs (GPOs)
  - PRU R31 directly reads from up to 30 GPI pins
  - PRU R30 directly writes up to 32 PRU GPOs
- Configurable I/O modes per PRU core
  - GP input modes
    - Direct connect
    - 16-bit parallel capture
    - 28-bit shift
  - GP output modes
    - Direct connect
    - Shift out



GPIO Toggle: Bench measurements

ARM GPIO Toggle

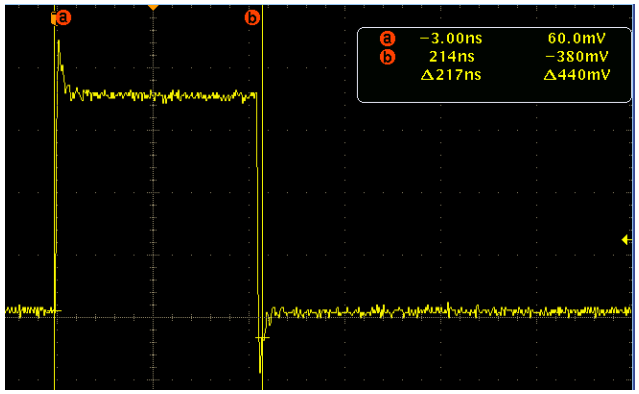
```
int main(){

    // Configure GPIO module, pinmuxing, etc.

    // Toggle system-level GPIO 3.19 from ARM core
    BitToggle(GPIO_INSTANCE_ADDRESS+GPIO_SETDATAOUT,
              GPIO_INSTANCE_ADDRESS+GPIO_CLEARDATAOUT);

    while();
}

unsigned long BitToggle(unsigned long val1, unsigned long val2){
    asm(
        " mov r2, #0x00080000" "\n\t"
        "str     r2,[r0]" "\n\t"        // Set GPIO 3.19
        "str     r2,[r1]" "\n\t"        // Clear GPIO 3.19
    );
    return val1;
}
```



~200ns

PRU IO Toggle:

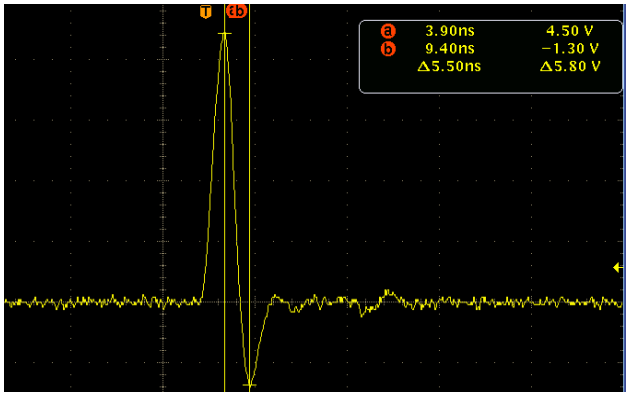
```
.origin 0
.entrypoint PRU_GPIO_TOGGLE


PRU_GPIO_TOGGLE:

    // Set PRU GPO 5
    SET       R30, R30, 5

    // Clear PRU GPO 5
    CLR       R30, R30, 5

    HALT
```

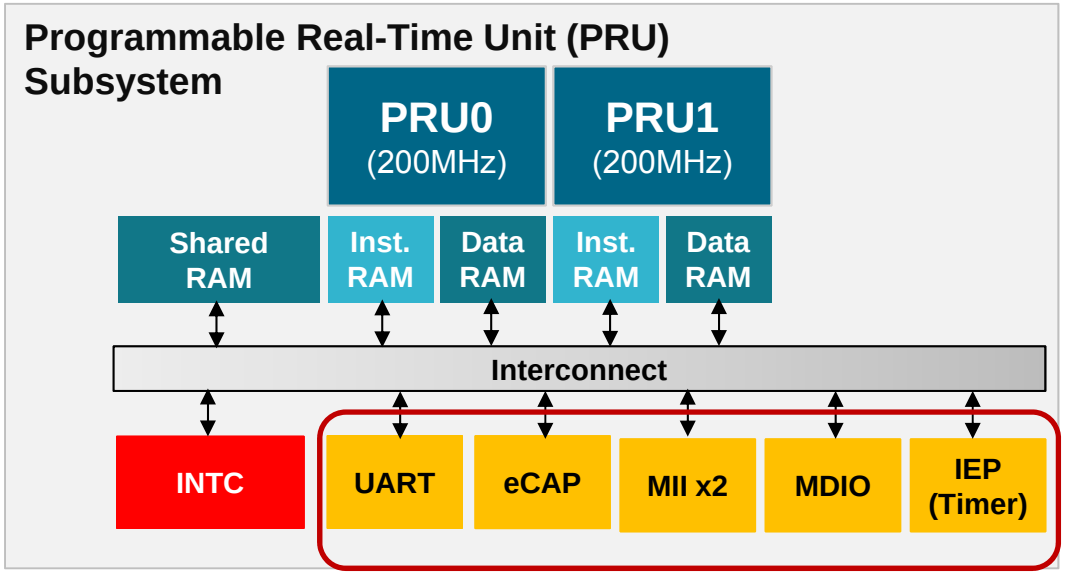


~5ns = ~40x Faster

Integrated Peripherals

- Provide reduced PRU read/write access latency compared to external peripherals
- Local peripherals don't need to go through external L3 or L4 interconnects
- Can be used by PRU or by the ARM as additional hardware peripherals on the device
- Integrated peripherals:
  - PRU UART
  - PRU eCAP
  - PRU MDIO
  - PRU MII_RT
  - PRU IEP

  } Real-time Ethernet specific modules



PRU “Interrupts”

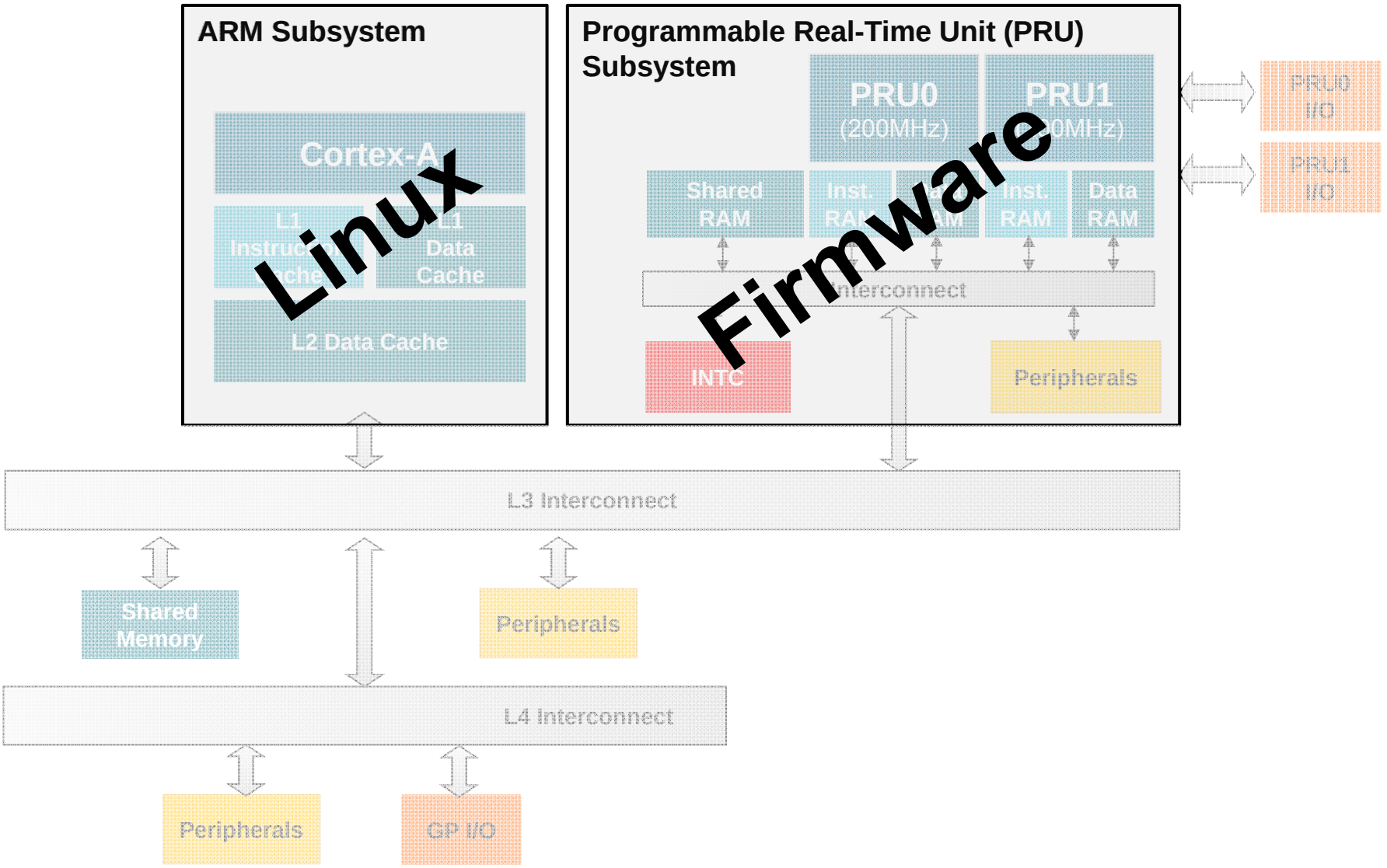
- The PRU does not support asynchronous interrupts.
  - However, specialized h/w and instructions facilitate efficient polling of system events.
  - The PRU-ICSS can also generate interrupts for the ARM, other PRU-ICSS, and sync events for EDMA.
- From UofT CSC469 lecture notes, “*Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring.*
  - *Interrupts win if processor has other work to do and event response time is not critical*
  - *Polling can be better if processor has to respond to an event ASAP*”
- Asynchronous interrupts can introduce jitter in execution time and generally reduce determinism. The PRU is optimized for highly deterministic operation.

PRU Summary and Update

- Cortex-A arch designed more for performance than real-time
- PRU designed for low-latency and deterministic, making real-time easier to deal with
- Using both for their designed purposes is an elegant system design
- Available on AM335x and AM437x, and planned for future devices
- New C Compiler and Register Header files for the PRU make firmware development easier than ever
- Upstream work in Linux frameworks to add PRU support makes interfacing with Linux easier as well

Linux Drivers to Interface with PRU

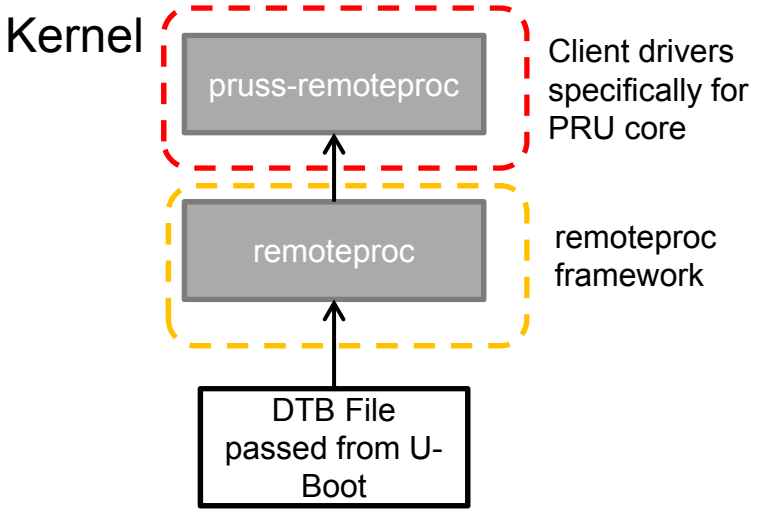
ARM + PRU SoC *Software* Architecture



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

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

Creating a New Node in DT

- 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 scalable TLV (table, length, value) Table 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
    - Necessary for ARM/PRU communication

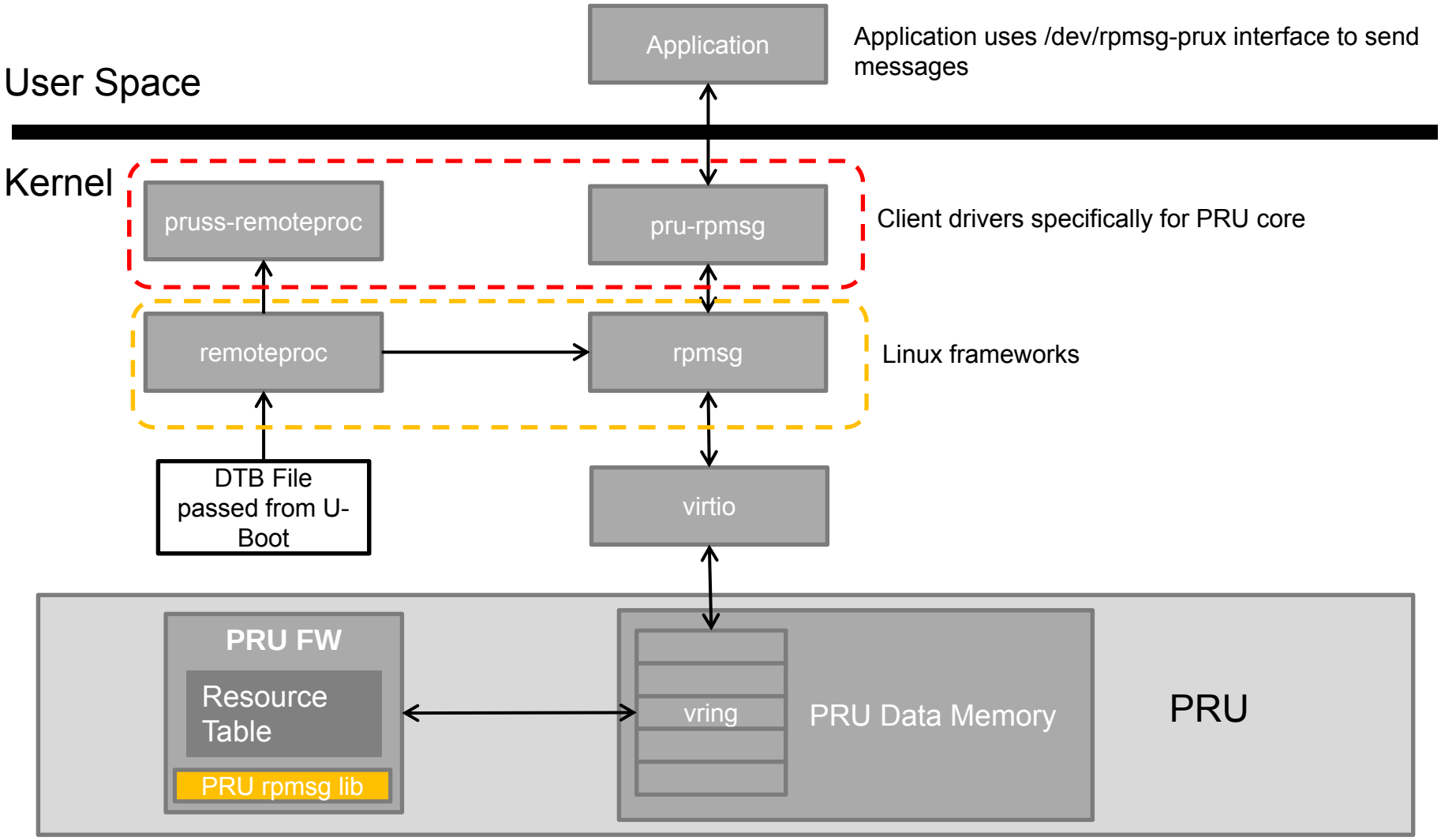
Why Use Remoteproc?

- 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

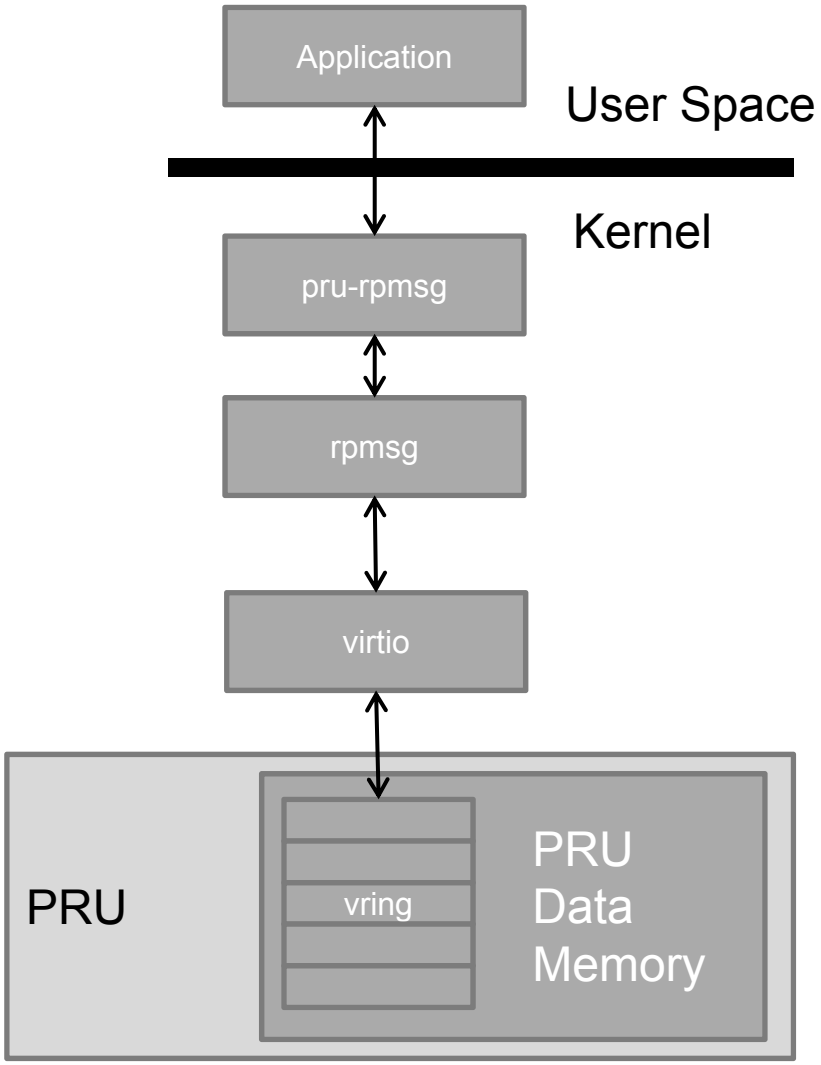
- Load driver manually or build into kernel
  - Use menuconfig to build into kernel or create a module
- Probe() function automatically looks for firmware in /lib/firmware directory in target filesystem
  - rproc_pru0_fw or rproc_pru1_fw for core 0 and 1, respectively
- Interrupts passed between host application and PRU firmware
  - Application effectively registers to an interrupt

PRU rpmsg Stack



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

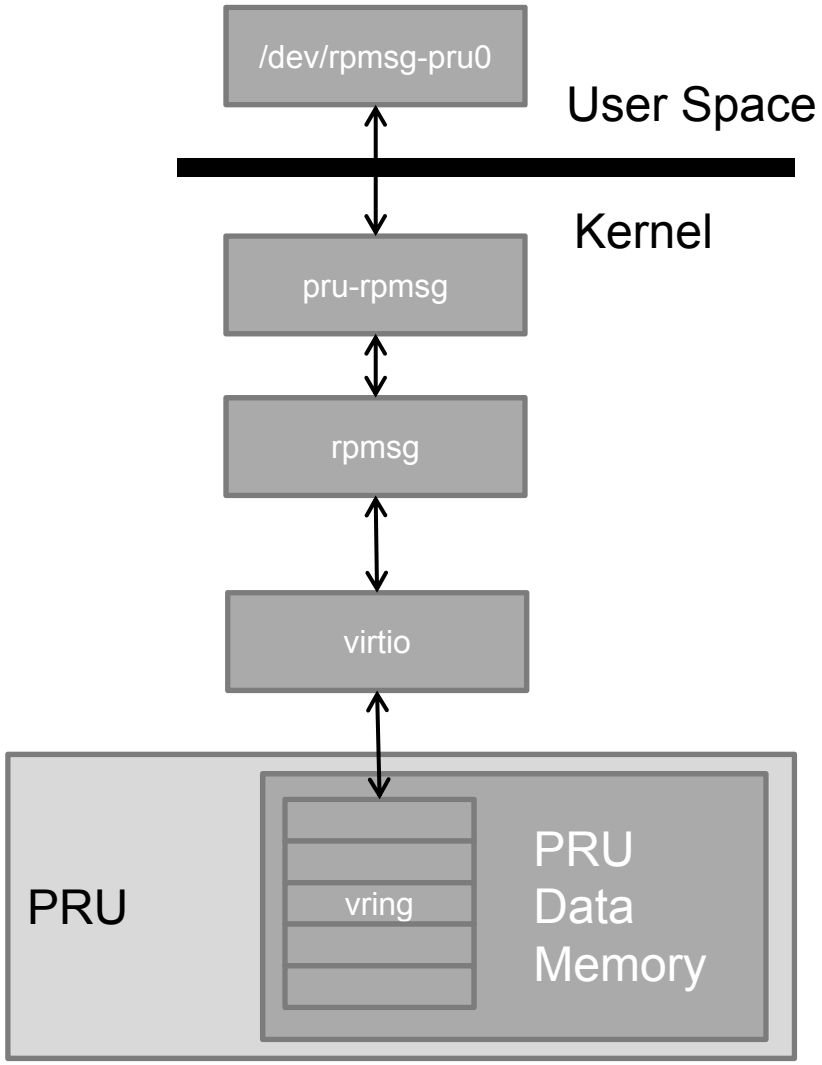


Why Use Rpmsg?

- 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.

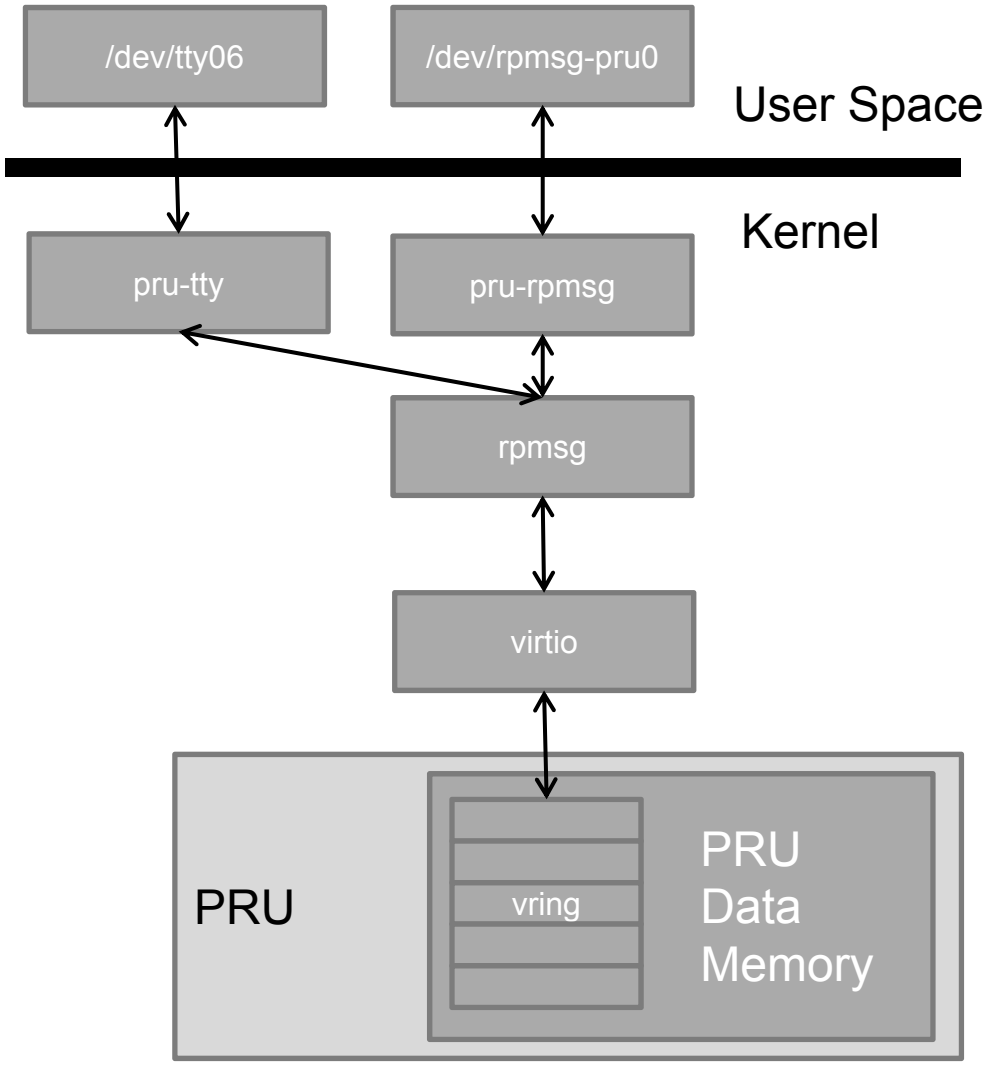
How to Use pru-rpmsg Generic Client Driver

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



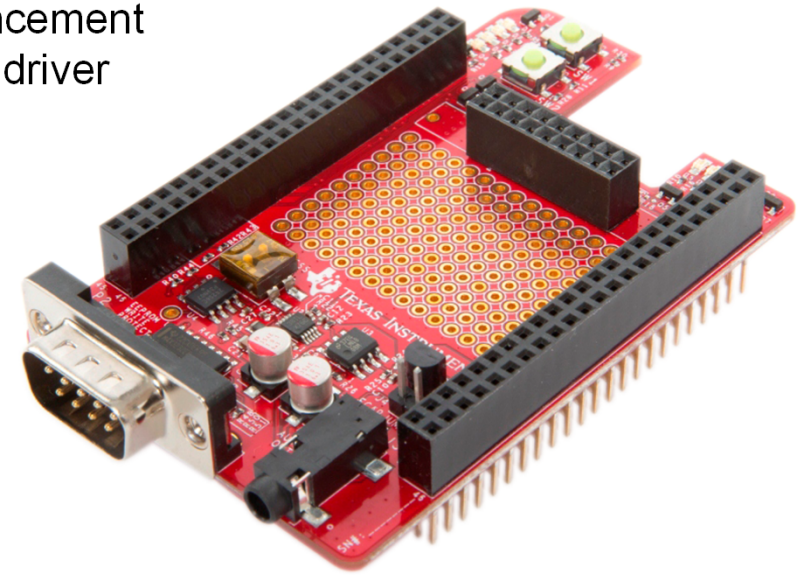
Custom rpmsg Client Drivers

- 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



What's happening now?

- Ongoing work to improve remoteproc/rpmsg for the PRU
  - Move from using hardware mailboxes for kick to PRU events
  - Allow vring buffers to be specified in PRU Data Memory as opposed to DDR
  - Allow smaller buffers to be used for vrings
  - Creation of generic pru-rpmsg client driver to expose PRU to User Space
  - Provide example firmware
    - Broadcast dynamic name service announcement message to connect to appropriate client driver
    - Send/Receive data via vrings
    - Use PRU events to manage kicks
- The PRU BeagleBone Cape
  - Easy way to experiment with PRU
  - Labs available with release
  - Available Nov. 4, 2014

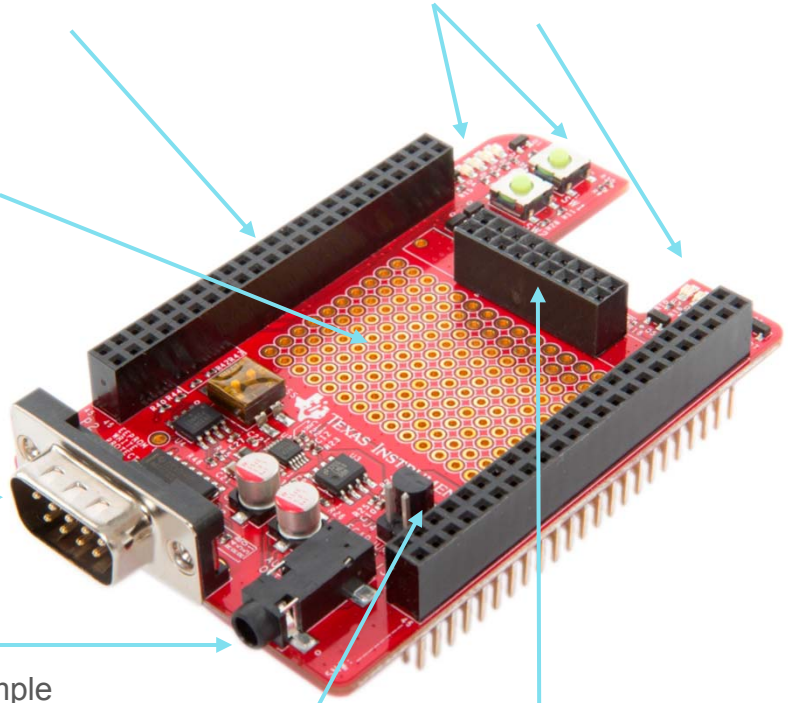


Programmable Real-time Unit (PRU) Cape
Inexpensive and easy PRU evaluation

Expansion connectors
Connectivity to BeagleBone Black

LEDs and push buttons
For PRU GPIO testing

Proto board area
For prototyping your own design



UART
Hardware UART in PRU subsystem

Audio output
Sine wave tone example using the PRU shift-out hardware

Temperature sensor
Sensor monitoring example using 1-wire interface

LCD connector
- For optional character display (not included)
- Compatible with Newhaven NHD-0208AZ-RN-YBW-33V

Features & Benefits
- BeagleBone Black compatible plug-in board (“cape”)
- Quick development and evaluation of the PRU integrated in Sitara processors
- Leverages sample code included in the TI Linux Software Development Kit
- The PRU core is optimized for deterministic, real-time processing, direct access to I/Os and ultra-low-latency requirements

Schedule
- TI Design available now: http://www.ti.com/tool/TIDEP0017
- Available for $39 from TI Store and Distributors in Nov 2014

Thank you

Backup Slides

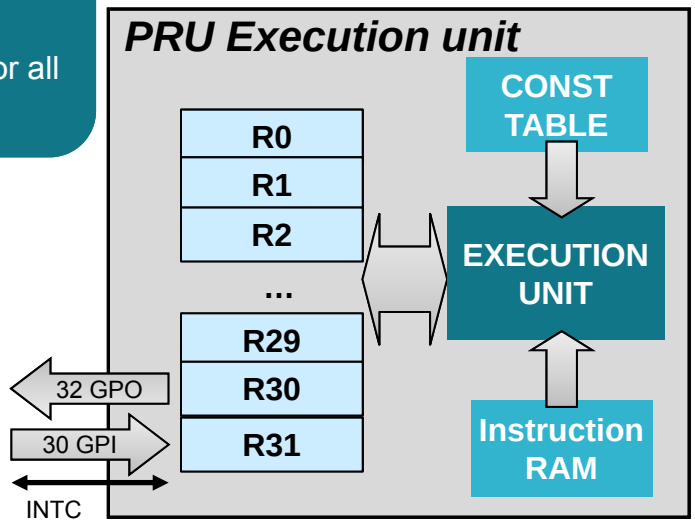
Features & Benefits

Feature	Benefit
Each PRU has dedicated instruction and data memory and can operate independently or in coordination with the ARM or the other PRU core	Use each PRU for a different task; use PRUs in tandem for more advanced tasks
Access all SoC resources (peripherals, memory, etc.)	Direct access to buffer data; leverage system peripherals for various implementations
Interrupt controller for monitoring and generating system events	Communication with higher level software running on ARM; detection of peripheral events
Dedicated, fast input and output pins	Input/output interface implementation; detect and react to I/O event within two PRU cycles
Small, deterministic instruction set with multiple bit-manipulation instructions	Easy to use; fast learning curve

PRU Functional Block Diagram

General Purpose Registers

- All instructions are performed on registers and complete in a single cycle
- Register file appears as linear block for all register to memory operations



Constant Table

- Ease SW development by providing freq used constants
- Peripheral base addresses
- Few entries programmable

Execution Unit

- Logical, arithmetic, and flow control instructions
- Scalar, no Pipeline, Little Endian
- Register-to-register data flow
- Addressing modes: Ld Immediate & Ld/St to Mem

Special Registers (R30 and R31)

- R30
  - Write: 32 GPO
- R31
  - Read: 30 GPI + 2 Host Int status
  - Write: Generate INTC Event

Instruction RAM

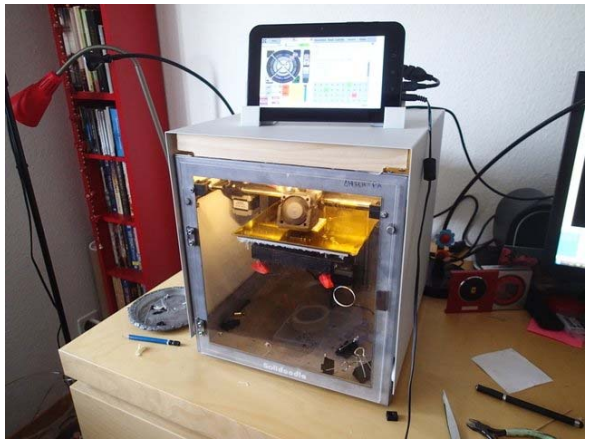
- Typical size is a multiple of 4KB (or 1K Instructions)
- Can be updated with PRU reset

Use Cases Examples

- Industrial Protocols
- ASRC
- 10/100 Switch
- Smart Card
- DSP-like functions
- Filtering
- FSK Modulation
- LCD I/F
- Camera I/F
- RS-485
- UART
- SPI
- Monitor Sensors
- I2C
- Bit banging
- Custom/Complex PWM
- Stepper motor control

Not all use cases are feasible on PRU

- Development complexity
- Technical constraints (i.e. running Linux on PRU)



Development Complexity

C Compiler

- Developed and maintained by TI CGT team
  - Remains very similar to other TI compilers
- Full support of C/C++
- Adds PRU-specific functionality
  - Can take advantage of PRU architectural features automatically
  - Contains several intrinsics
    - List can be found in Compiler documentation
- Full instruction-set Assembler for hand-tuned routines

TI PRU CGT Assembler vs PASM

- **Advantages of using TI PRU Assembler over PASM**
  - **The biggest advantage is that the TI PRU linker produces ELF files that enable source-level debugging within CCS. No more debugging in disassembly window!!**
  - **The TI PRU assembler uses the same shell as other TI compilers. Customers only need to learn one set of conventions, directives, etc.**
  - **TI PRU assembler will be maintained in the future, while PASM will not be updated anymore.**
  - **The TI PRU assembler uses the powerful TI linker which allows more flexibility then PASM and facilitates linking PRU programs with host CPU image for runtime loading and symbol sharing.**

- **Disadvantages of using TI PRU Assembler over PASM**
  - **Have to learn new directives if already used to PASM**
  - **TI PRU assembler requires more command line options and a linker command file.**
  - **Some porting effort required for reusing legacy PASM projects.**

There are some differences in the instructions and directives supported TI PRU Assembler versus PASM. Theses are listed in the TI PRU Compiler package release notes which is located at the root of the install folder.

TI PRU CGT Assembly vs C

- Advantages of coding in Assembly over C
  - Code can be tweaked to save every last cycle and byte of RAM
  - No need to rely on the compiler to make code deterministic
  - Easily make use of scratchpad

- Advantages of coding in C over Assembly
  - More code reusability
  - Can directly leverage kernel headers for interaction with kernel drivers
  - Optimizer is extremely intelligent at optimizing routines
    - “Accelerating” math via MAC unit, implementing LOOP instruction, etc.
  - Not mutually exclusive - inline Assembly can be easily added to a C project

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

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