Remote Processor (Linux remoteproc) Guide

Remote Processor In AMP/HMP systems

Version 1.0.0

daybreakOs

History

Contents

1. Remote Processor:

Ref: The official kernel documentation for remote processor can be found at:

https://docs.kernel.org/staging/remoteproc.html j

1.1. Introduction:

The ***remoteproc*** framework in the Linux Kernel is primarily concerned with managing remote processor (or coprocessors) in a heterogeneous system. These systems often feature a primary CPU and additional processors, such as DSPs (Digital Signal Processors), GPU's or other specialized co-processors that run that run independently of the main CPU but still need to communicate with it.

To understand the `remoteproc` system and its kernel implementation, it's important to cover a few key historical and conceptual aspects:

1.1.1. History of Heterogeneous Computers:

Heterogeneous Systems: Modern computing increasingly use heterogeneous computing, where the main processor (often general-purpose CPU like ARM, x86 or PowerPC) work alongside specialized processors like GPUs, DSPs, FPGAs Or NPUs (neural processing units). These specialized processors are optimized for specific tasks such as video encoding/decoding, machine learning, signal processing, or graphics rendering. **Co-Processors:**: These coprocessors run in parallel along with the main CPU but typically not directly managed by the main operating system. Instead they often required specific device drivers or custom SW to communicate with the main system.

1.1.2. Earlier Linux support for Remote Processors:

- **Legacy of Remote Processor Management**: Initially, Linux was designed to manage only the main CPU. As heterogeneous systems grew in popularity, it became apparent that there was no unified way to handle remote processors (e.g., DSPs or specialized accelerators). This led to the development of `remoteproc`, which provides a framework for managing remote processors in Linux.
- **RemoteProc Goals**: The goal of the `remoteproc` subsystem is to provide a standardized way for the Linux kernel to manage these remote processors, load their firmware, and set up communication between the main CPU and remote processor(s). The subsystem helps decouple the details of remote processor management from the core Linux kernel, enabling cleaner, more maintainable code.

1.1.3. Architecture of `remoteproc`

The `remoteproc` subsystem abstracts out the complexities of managing remote processors and provides a consistent interface for both hardware and software:

- **Firmware Loading**: Many remote processors require firmware to function. The `remoteproc` subsystem manages the loading and unloading of this firmware. This is an essential feature because remote processors typically don't boot from the main system's firmware.

  - This is the core function, as Remote Processors do not have their own boot ROM that can load an OS from a general-purpose storage device like a SSD or eMMC, Instead Linux host is responsible for the following :
    1. Locate the appropriate firmware (@ /lib/firmware/)
    2. Loading this firmware (often an ELF executable) into the remote processors's dedicated memory regions.
    3. Initiating the remote processor's execution of this firmware.

    `remoteproc` abstracts this process, allowing the system to bring up various remote processors with their specific firmwares.

- **State Management**: Remote processors can be in different states, such as powered off, idle, running, or suspended. `remoteproc` provides mechanisms for state transitions and event handling, which helps coordinate the operation of remote processors in a complex system. `remoteproc` provides the internal state machine and the external API's (`rproc_boot()`, `rproc_shutdown()`..etc) that allow the host to control and query the remote processors operational state. This is crucial for power mgmt, error handling and coordinating tasks between the host and the remote processor. It's to note here that the event handling would refer to how the `remoteproc` driver reacts to events like remote processor crashing or requesting a state change.

- **Memory Management**: Remote processors often need memory that is distinct from the main system memory. The `remoteproc` subsystem handles memory regions that are shared between the main CPU and remote processors. I.E Remote processors often have their own physical memory or they might share portion of the main system's memory. `remoteproc` handles the complexities of :
  1. Defining and reserving memory regions for the remote processor's exclusive use (for its code, data , stack and heap).
  2. Setting up shared memory regions that both the host and the remote processor can access. This is particularly important for communication mechanisms like `rpmsg`, where message buffer reside in shared memory.
  3. Ensuring proper cache coherency or flushing when shared memory is involved, depending on the architecture.

1.1.4. The `remoteproc` Kernel API

The `remoteproc` subsystem provides a set of kernel APIs to interact with remote processors. Key functionalities include:

- **Initialization**: The `remoteproc` API handles the setup and initialization of remote processors, including memory management and device configuration.
  1. Loading firmware (generally ELF file) into the remote processor's memory.
  2. Configuring memory regions that the remote processor will use.
  3. Initializing resources like shared memory or communication channels.

- **State Transitions**: The API allows the kernel to move the remote processor between various states, such as powered off, halted, running, or suspended.

```
//remoteproc.h
enum rproc_state {
  RPROC_OFFLINE = 0,
  RPROC_SUSPENDED = 1,
  RPROC_RUNNING = 2,
  RPROC_CRASHED = 3,
  RPROC_DELETED = 4,
  RPROC_ATTACHED  = 5,
  RPROC_DETACHED  = 6,
  RPROC_LAST  = 7,
};
```

1. `RPROC_OFFLINE`: remote processor is powered down or not initialized.
2. `RPROC_RUNNING`: remote processor is executing its firmware.
3. `RPROC_SUSPENDED`: remote processor is in low-power state.

   `remoteproc` API's allows transition between these above states.

- **Interrupt Handling**: Remote processors often generate interrupts, which need to be handled by the main system. The `remoteproc` subsystem includes support for handling interrupts coming from remote processors. These interrupts signal events or completion of tasks to the main CPU. `remoteproc` drivers handle the registration and processing of interrupts originating from remote processor, oftern translating them into actions on the host side.

- **Communication Mechanisms**: Many systems use a form of inter-process communication (IPC) between the main CPU and the remote processor. `remoteproc` often interfaces with other subsystems (like `rpmsg`, which is used for messaging between processors) to facilitate this communication. I.E `remoteproc` does not define the communication protocol, it provides the foundation upon which communication can be built. It is very frequently interfaces with:
  1. `rpmsg`: Remote processor messaging: Most common and widely used messaging framework that sits on top of `remoteproc`. `rpmsg` provides a standard way for applications on the host and remote processor to send and receive messages. `remoteproc` provides the shared memory and interrupt mechanisms that `rpmsg` utilizes.
  2. Other custom or simple shared mem mechanisms might also be employed depending on the specific use case, but `rpmsg` is preferred and more feature-rich solution.

1.1.5. The `remoteproc` Kernel API

The `remoteproc` subsystem provides a set of kernel APIs to interact with remote processors. Key functionalities include:

- **Initialization**: The `remoteproc` API handles the setup and initialization of remote processors, loading firmware, setting up memory and configuring necessary hardware and device configurations.

- **State Transitions**: The API allows the kernel to move the remote processor between various states, such as powered off, halted, running, or suspended.

- **Interrupt Handling**: Remote processors often generate interrupts, which need to be handled by the main system. The `remoteproc` subsystem includes support for handling interrupts coming from remote processors.
- **Communication Mechanisms**: Many systems use a form of inter-process communication (IPC) between the main CPU and the remote processor. `remoteproc` often interfaces with other subsystems (like `rpmsg`, which is used for messaging between processors) to facilitate this communication. While not a communication protocol itself `remoteproc` provides the underlying infrastructure (like shared memory and interrupt notification) that communication frameworks like `rpmsg` utilize to enable IPC between the host and remote processor.

1.1.6. Integration with Other Kernel Subsystems

- **RPMsg (Remote Processor Messaging)**: One common method of communication between a main processor and a remote processor is via the `rpmsg` subsystem. The `remoteproc` subsystem often works in tandem with `rpmsg` to facilitate message passing, data transfer, and synchronization between processors.
- **Devicetree**: In many embedded systems, the devicetree is used to describe the hardware configuration of the system, including remote processors. The `remoteproc` subsystem is often integrated with devicetree, allowing for automatic initialization and configuration of remote processors at boot time.
- **PM (Power Management)**: Power management is crucial when dealing with remote processors, especially in mobile and embedded systems where power consumption is a concern. The `remoteproc` subsystem integrates with the Linux power management infrastructure to ensure that remote processors are powered on or off as necessary.

1.1.7. Real-World Use Cases:

- **System-on-Chip (SoC) Designs**: Modern SoCs (such as those used in smartphones, tablets, and embedded systems) often contain multiple processors, including general-purpose CPUs, GPUs, DSPs, and NPUs. Managing these processors efficiently is essential for system performance, power management, and functionality.
- **Embedded Systems**: Embedded systems often use remote processors to handle specialized tasks, such as video encoding/decoding or sensor data processing. In these systems, the remoteproc subsystem ensures that the remote processors are initialized and managed correctly.
- **Accelerators in AI/ML**: The rise of machine learning and AI has spurred the need for specialized accelerators like NPUs or TPUs. These processors require efficient integration into the Linux kernel, and the `remoteproc` subsystem plays a key role in managing them.

1.1.8. Key Concepts to Understand

To dive deeper into the `remoteproc` documentation, you'll need a good grasp of the following concepts:

- **Firmware Management**: Understanding how firmware is loaded, executed, and unloaded is fundamental. In some cases, remote processors must load their firmware at runtime, which is managed by the kernel's `remoteproc` subsystem.
- **State Transitions**: The ability to transition a remote processor between various states (e.g., halted, running) is central to its management.
- **Interprocessor Communication**: Understanding how data is exchanged between the main CPU and remote processors (via IPC, `rpmsg`, or memory sharing) is crucial to understanding how remote processors fit into the system.
- **Platform-Specific Code**: In some cases, supporting a specific remote processor requires writing platform-specific code in the kernel, especially when dealing with non-standard hardware.
- **Synchronization**: Remote processors may need to synchronize with the main CPU or with each other, so understanding synchronization techniques in the Linux kernel (e.g., mutexes, spinlocks) is important.

1.1.9. Key Documentation Sections:

- **Introduction to RemoteProc**: Provides an overview of the subsystem's purpose and general usage.
- **APIs and Functions**: Lists the various kernel functions and structures for interacting with remote processors.
- **DeviceTree Integration**: Explains how to integrate remote processors into the device-tree for hardware description and initialization.
- **Use Cases and Examples**: Includes examples of how the `remoteproc` subsystem is used in real-world scenarios.

2. Beyond Linux:

The **Remote processor subsystems** (`remoteproc`) is not execlusive to Linux Kernerl, it is simply **one implementation** of a general concept: managing and communicating with **remote (or auxiliary) processor** in a hetrogeneous system (HMP) or AMPs.

2.1. HMP/AMP:

In modern HMP/AMP systems we typically have

- A main processor (ARM Cortex-A … running Linux or RTOS)
- One or more remoteprocessors (DSPs or Cortex-M cores, NPUs)

Managing these remote processors involves:

- Loading firmware
- Booting/Stopping the processor
- Managing shared memory
- Inter-Processor Communication (IPC)

And Linux remoteproc subsystem does the above and provides drivers and APIs to do all of this within the Linux Kernel.

2.2. Other RTOS with Remote processor framework:

- Generally Other RTOS or any Rust based OS this has to be implemented but this would come with the possible challenges in implementing remoteproc functionality.
  1. **FW loading**: RTOS must support reading and parsing firmware images possbly from flash or over the air.
  2. **Boot Control**: Must have HW specific mechanism to control the remote processor (ex: kick off a Cortex-M core from a Cortex-A)
  3. **Memory Mapping**: Need to setup shared memory correctly between the processors.
  4. **IPC**: Need to Implement a messaging protocol like `rpmsg` or a custom one.

3. What is virtIO?

VirtIO is a **paravirtualized device framework** — designed for VMs but applicable to any system that needs virtual devices. Originally used in QEMU/KVM for devices like:

- `virtio-net` (network)
- `virtio-blk` (block device)
- `virtio-console` / `virtio-serial`

The **RPMsg protocol itself is built on virtIO**:

> `RPMsg` = `virtio` + `shared memory` + `mailbox` + `message protocol`

This sub-system can also extend by implementation of virtio devices. Note: **you can design a communication system using virtIO devices** (like `virtio-net`, `virtio-console`, etc.) instead of RPMsg + mailbox. In fact, **RPMsg already uses virtIO under the hood.**

3.1. Designing with virtIO Devices

3.1.1. Example 1: `virtio-net`

- You could expose a **virtual network device** to the RTOS.
- RTOS sends/receives packets as if through a NIC — Linux receives them via the `virtio-net` interface.
- You'd need:
  1. A shared memory ring buffer for TX/RX descriptors
  2. A mailbox or interrupt to notify the other core
  3. A virtIO-net frontend on the RTOS side (there are examples in Zephyr!)

3.1.2. Example 2: `virtio-serial` or `virtio-console`

- This mimics a UART or console channel.
- Works well for command/control or log data.
- Simpler than `virtio-net`, but still needs virtIO framework.

4. OpenAMP:

OpenAMP (Open Asymmetric Multi-Processing) is an open-source framework that provides the software components needed to enable the development of applications for

Asymmetric Multiprocessing (AMP) systems. In an AMP system, multiple processors, often with different architectures (e.g., a powerful application processor running Linux and a real-time microcontroller or DSP), coexist on the same System-on-Chip (SoC) and work together on different tasks.

The core purpose of OpenAMP is to **standardize the interactions between these diverse operating environments**, abstracting away the underlying hardware complexities to simplify heterogeneous system development.

4.1. Why OpenAMP? The Problem It Solves

Modern SoCs are increasingly complex and heterogeneous, featuring:

- **General-Purpose Processors (GPPs)**: Often running rich operating systems like Linux for complex applications, networking, and user interfaces.
- **Real-Time Processors (RTPs) / Microcontrollers (MCUs) / DSPs**: Designed for deterministic, low-latency tasks like motor control, sensor data processing, or audio/video encoding.
- **Special-purpose Accelerators**: Hardware blocks for specific functions (e.g., image processing, AI inference).

The challenge lies in making these disparate components work together efficiently and reliably. Before OpenAMP, communication and coordination between such processors often involved ad-hoc, vendor-specific solutions, leading to:

- **Complex and non-portable code**: Each new SoC or processor combination required significant re-engineering.
- **Difficult resource management**: Ensuring proper allocation and sharing of memory, peripherals, and interrupts was a manual and error-prone process.
- **Limited interoperability**: Different OSes and bare-metal environments struggled to communicate effectively.

OpenAMP addresses these issues by providing a standardized framework.

4.2. Key Capabilities and Components of OpenAMP

OpenAMP focuses on two primary capabilities, realized through its core components:

4.2.1. Life Cycle Management (LCM) - Powered by `remoteproc`

- **Purpose**: To enable a “master” processor (typically the GPP running Linux) to control the boot-up, shutdown, and overall operational state of “remote” processors.
- **Mechanism**: OpenAMP leverages and is compliant with the Linux kernel’s **`remoteproc`** framework.
  - The `remoteproc` component in OpenAMP provides APIs and functionality for the remote (non-Linux) side to interact with the Linux `remoteproc` driver.
  - This includes:
    1. **Firmware Loading**: The master loads the remote processor’s firmware (often an ELF file) into its dedicated memory.

2. **Boot/Shutdown**: The master initiates the remote processor's execution or brings it down.
3. **Resource Table**: The remote firmware provides a `resource_table` (a data structure embedded in its ELF file) that describes the resources it needs (e.g., memory regions, virtio devices for communication) to the master's `remoteproc` driver. This allows the master to properly configure and set up the environment for the remote.

4.2.2. Inter-Processor Communication (IPC) - Powered by `RPMsg` (Remote Processor Messaging)

- **Purpose**: To facilitate efficient and standardized message-based communication between the different software contexts running on heterogeneous cores.
- **Mechanism**: OpenAMP uses and is compliant with the Linux kernel's **`rpmsg`** framework.
  1. `RPMsg` builds on top of the **VirtIO** standard (specifically `virtio-ring` and `virtio-rpmsg`). VirtIO provides a standardized, efficient mechanism for I/O virtualization, typically used between a guest OS and a hypervisor. OpenAMP adapts this concept for inter-processor communication in an AMP system.
  2. **Virtqueues**: `RPMsg` uses shared memory regions called "virtqueues" (or vrings) for message exchange. These are circular buffers that allow producers and consumers to pass data efficiently without excessive copying.
  3. **Channels and Endpoints**: `RPMsg` operates on the concept of named "channels" (services) and "endpoints" (logical connections within a channel). Applications register for specific channels to send and receive messages.
  4. **Lightweight Messaging**: `RPMsg` is designed for low-latency, small-to-medium sized messages. For larger data transfers, it can be used to coordinate shared memory buffers, where only pointers to the data are exchanged via `RPMsg`, and the actual bulk data transfer happens directly in shared memory.

4.3. Supporting Component: `libmetal`

OpenAMP heavily relies on `libmetal`. While not a core part of the `remoteproc` or `rpmsg` **protocol** itself, `libmetal` acts as a crucial **hardware and OS abstraction layer**.

- **Hardware Abstraction**: It provides a common set of APIs for low-level operations that vary between different SoCs and processors, such as:
  - Memory allocation and mapping (e.g., accessing specific physical memory addresses).
  - Cache management (flushing, invalidating caches for shared memory).
  - Interrupt handling (registering interrupt handlers, enabling/disabling interrupts).
  - Synchronization primitives (mutexes, atomics).
- **OS Abstraction**: `libmetal` allows OpenAMP to be portable across various operating environments (Linux kernel, Linux userspace, various RTOSes like FreeRTOS, Zephyr, bare-metal). It provides a generic interface that is then implemented specifically for each target OS/platform.

4.4. How OpenAMP Works (Typical Flow)

1. **System Design**: Define the system topology, partition hardware resources (memory, peripherals, interrupts) between the master and remote processors, and determine shared memory layouts.
2. **Firmware Development**: Develop the remote processor's firmware (e.g., a FreeRTOS application or bare-metal code). This firmware must include a `.resource_table` section, detailing its resource requirements and virtio device configurations for `rpmsg`.
3. **Host (Linux) Configuration**: **The Linux kernel's `remoteproc` driver is configured via Device Tree to recognize the remote processor and its associated resources.** The `rpmsg` bus driver will then detect the `rpmsg` virtio devices announced in the remote's resource table.
4. **Boot Sequence**: **The Linux `remoteproc` driver (or a user-space application controlling `remoteproc` via sysfs) loads the remote firmware.** It boots the remote processor by releasing it from reset or powering it on. **The remote processor initializes, including its `libmetal` and OpenAMP components, and sets up its `rpmsg` virtio devices based on its `resource_table`.** A "service announcement" message is typically sent from the remote to the host over `rpmsg`, signaling its readiness and available services.
5. **Communication**: Once the `rpmsg` channels are established, applications on both the host and remote can send and receive messages using `rpmsg` APIs. **For example, a Linux application can send a command to the remote, and the remote can send back status updates or processed data.** Proxy mechanisms are also supported, allowing remote applications to "redirect" standard C library calls (like `printf`, `open`, `read`, `write`) to the Linux host for execution, simplifying development on resource-constrained remote processors.

4.5. Use Cases and Benefits

OpenAMP is widely adopted in heterogeneous embedded systems for various applications:

- **Industrial Automation**: Real-time control loops on MCUs communicating with Linux-based HMI/network gateways.
- **Automotive (ADAS/Infotainment)**: Safety-critical functions on dedicated MCUs interacting with complex infotainment systems on GPPs.
- **Embedded Vision and AI**: Offloading intensive image processing or AI inference to DSPs/accelerators while a GPP manages the overall application.
- **Networking and Telecom (5G Infrastructure)**: Real-time packet processing on dedicated cores, managed by a Linux system for control plane operations.
- **Mixed-Criticality Systems**: Separating safety-critical tasks (on an RTOS/bare-metal) from non-critical functions (on Linux), while maintaining communication.

Benefits of using OpenAMP:

- **Standardization**: Promotes interoperability and reduces vendor lock-in.
- **Modularity**: Decouples processor management from communication protocols.
- **Portability**: `libmetal` enables easier porting to different hardware and OS environments.

- **Efficiency**: `virtio` and `rpmsg` provide a low-overhead, high-throughput communication mechanism.
- **Simplified Development**: Abstracting complexities allows developers to focus on application logic.

In essence, OpenAMP acts as the glue that enables sophisticated heterogeneous SoCs to unlock their full potential by providing a robust and standardized framework for managing and communicating between diverse processing units.

5. `remoteproc + openAMP`

5.1. Best Practice AMP Design: `remoteproc + OpenAMP`

5.1.1. Flow Summary

Component	Role
Linux (host)	Boots first, loads firmware to remote core via `remoteproc`
Remote OS	Typically runs RTOS or bare-metal firmware with `OpenAMP + libmetal`
IPC Layer	Communication is done via **RPMsg** (message queues over shared memory)
Signaling	Mailbox, IPI, or interrupt is used to notify the other processor

5.1.2. Why This Design Works

- **Decouples** Linux from RTOS runtime logic
- Lets Linux **control lifecycle** of remote core (start/stop/reset)
- Uses **resource table** to describe memory regions, RPMsg channels, etc.
- **OpenAMP** abstracts the complexity on the remote side (virtqueues, shared memory)
- **RPMsg** gives you a clean message-passing API (no raw memory hacks)

5.2. Component Breakdown

5.2.1. On Linux Host

- Kernel driver: `remoteproc` (loads and boots remote firmware)
- IPC driver: `rpmsg` (handles communication and creates `/dev/rpmsg`)
- Device Tree: describes memory regions, firmware location, mailbox, etc.

5.2.2. On Remote (RTOS/Bare-Metal)

- **OpenAMP stack**:
  - Uses `libmetal` for HAL (memory map, IRQ, etc.)
  - Implements `remoteproc` (minimal stub for boot sync)
  - Implements `rpmsg` transport (virtqueues, ring buffers)
- Firmware: includes `resource_table`, which tells Linux what features/channels exist

5.3. Example: Resource Table (Simplified)

```
struct my_resource_table {
    struct resource_table base;
    uint32_t offset[1]; // One entry

    struct fw_rsc_vdev rpmsg_vdev; // VirtIO device
    struct fw_rsc_vdev_vring vring0;
    struct fw_rsc_vdev_vring vring1;
};
```

This tells Linux:

- "I'm an RPMsg-capable firmware"
- "Here's the shared memory and virtqueues"
- "Let's talk over a specific endpoint"

5.4. Communication: RPMsg Channels

- RTOS calls : `rpmsg_send() , rpmsg_recv()`, etc.
- Linux gets a ` /dev/rpmsg_ctrlX ` and then dynamically creates endpoints like:
  - /dev/rpmsgX
- You can write/read like a file, or use netlink-style sockets in user space.

5.5. Benefits of This Design

Feature	Benefit
Scalable	Add more endpoints easily
OS-Agnostic Remote	RTOS, Zephyr, FreeRTOS, etc.
Message-based	No need for memory locks
Fast development	Works out-of-the-box on STM32MP1, TI, etc.
Industry-proven	Used in automotive, medical, industrial

5.6. Potential Challenges

Challenges	Solution
Debugging remote OS	Use semihosting, JTAG, or serial logs
Synchronization	Use RPMsg events, not shared memory locks
DMA-coherence	Use cache management or non-cached buffers

5.7. Real-World Usage:

- **STM32MP1**: Cortex-A7 Linux host + Cortex-M4 Zephyr/FreeRTOS with OpenAMP
- **TI Sitara AM64x**: Linux A53 host + R5F RTOS with OpenAMP
- **Xilinx Zynq MPSoC** : A53 + R5F With openAMP and remoteproc.

5.8. Recommended Dev Boards to Try

SoC/Board	Notes
STM32MP157-DK2	Ideal starter for Linux + OpenAMP
TI AM64x LaunchPad	Heterogeneous cores + RPMsg ready

SoC/Board	Notes
BeagleBone AI-64	AM62x platform with remoteproc