Building Virtual Drivers with RPMsg:
Key Design Principles, Challengs & Trade-offs

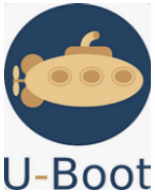
Date: 2026-05-18

Beleswar Prasad Padhi

About us: TI Processors and Open Source

Upstream FIRST!

- ✔ Upstream FIRST mentality!
- ✔ Decades of contribution and collaboration
- ✔ Focus on long term, sustainable and quality products
- ✔ Upstream & opensource ecosystem in device architecture
- ✔ Ingrained culture to give back to the community



Introduction to the Speakers

Beleswar Prasad Padhi, *Senior Software Engineer at Texas Instruments*

Beleswar is a Senior Software Engineer at Texas Instruments, actively working on Upstream Linux Kernel and U-Boot. His work mainly focuses on Remoteproc, RPMsg, Mailbox, Virtio subsystems, as well as boot-time optimizations. He was listed among the top contributors for Linux 6.18 LTS release. He is a FOSS Enthusiast and has contributed to several Open Source projects like Linux, U-Boot, ZephyrRTOS, Vim, Armbian, Metasploit and Mandiant. He was a mentee under Google Summer of Code program twice.

Overview

- Problem Statement
- Primer on RPMsg and VirtIO
  - Channel vs Endpoint
  - Static and Dynamic Channels/Endpoints
- Design Principles
- Comparison with VirtIO
  - Latency
  - Resource usage
  - Usecase suitability
- Challenges
- Upstreaming Lessons

Texas Instruments

Problem Statement

- Peripheral sharing is becoming more important in Heterogenous SoCs
- Automotive use cases
  - Safety oriented: Require to run on safety core
  - Time critical: Require to run on real-time core
  - Complex: 3D surround view, EdgeAI can not be run standalone from a single core
- Increasing need of driving the same peripheral by a safety component and a non safety component

Texas Instruments

Problem Statement

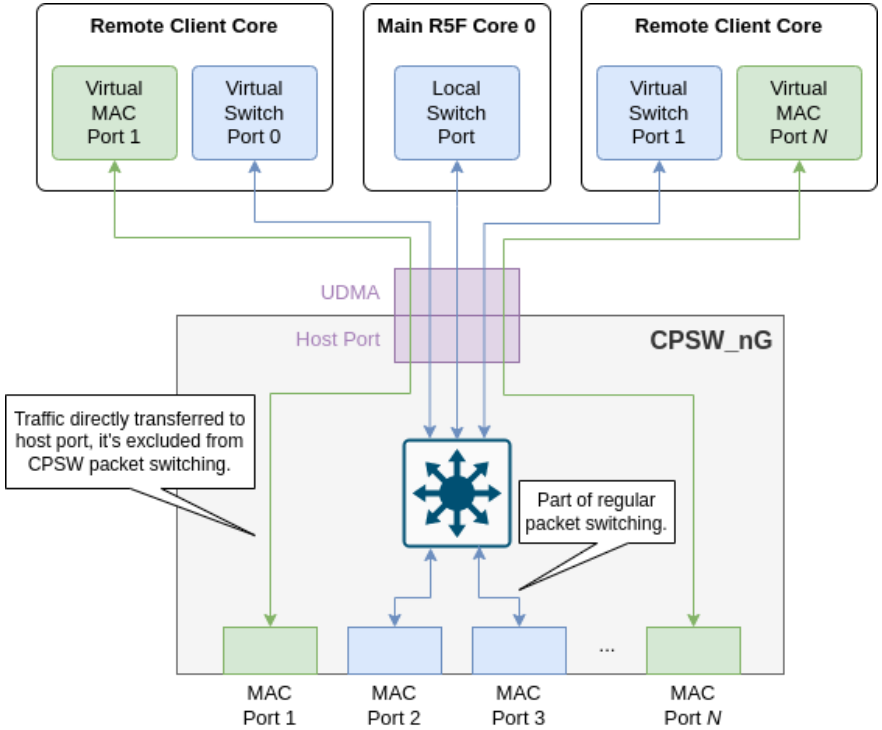


Fig.1: Ethernet traffic tunelling

Fig.2: Automotive cluster tell-tales

Texas Instruments

Problem Statement

- Peripheral sharing is becoming more important in Heterogenous SoCs
- Automotive use cases
  - Safety oriented: Require to run on safety core
  - Time critical: Require to run on real-time core
  - Complex: 3D surround view, EdgeAI can not be run standalone from a single core
- Increasing need of driving the same peripheral by a safety component and a non safety component
- **How can this be enabled in an efficient way ensuring synchronization, promising throughput and error handling?**

Texas Instruments

Solutions for Peripheral Sharing

- Build Virtual drivers with a communication mechanism between Heterogeneous cores
- Backend transport mechanism can be via VirtIO or RPMsg
- What is VirtIO?
  - Raw & efficient transport mechanism to move data from point A to point B
  - Works with (atleast) a pair of virtqueues in shared memory, TX and RX VRINGs
  - Data buffer sizes can be as huge as 2^{16} bytes
  - Cannot differentiate between multiple services
  - Requires separate set of VRINGs for each service/device

Texas Instruments

Solutions for Peripheral Sharing

- Build Virtual drivers with a communication mechanism between Heterogeneous cores
- Backend transport mechanism can be via VirtIO or RPMsg
- What is RPMsg?
  - Protocol layer built on top of VirtIO for IPC in AMP systems
  - Appends a metadata to each message to achieve multiplexing
  - Data buffer sizes are fixed to 512 bytes by design
  - Can differentiate between multiple services over the single set of VRINGs

Texas Instruments

Solutions for Peripheral Sharing | RPMsg

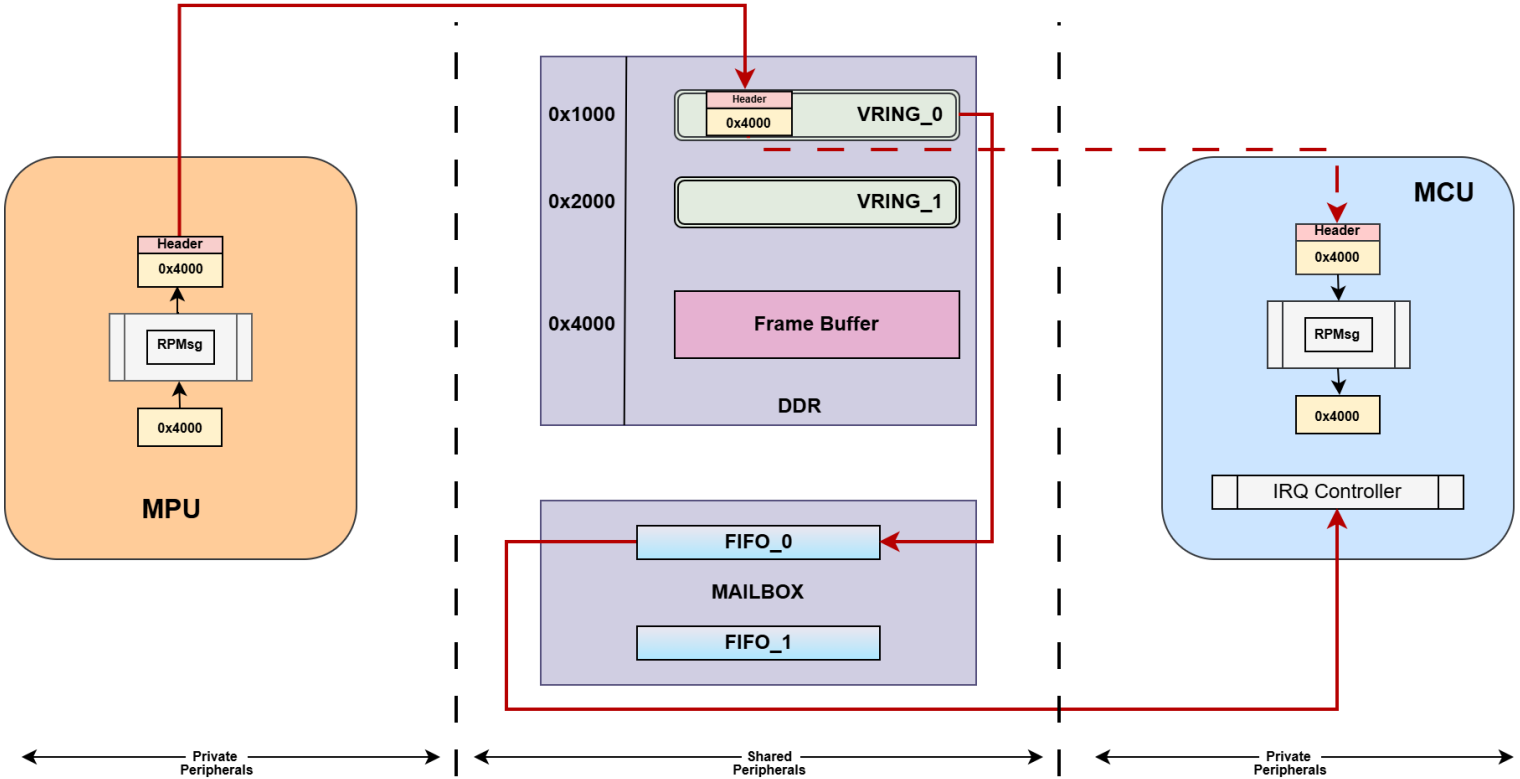


Fig.3: Data flow with RPMsg from MPU to MCU

Texas Instruments

Solutions for Peripheral Sharing | RPMsg

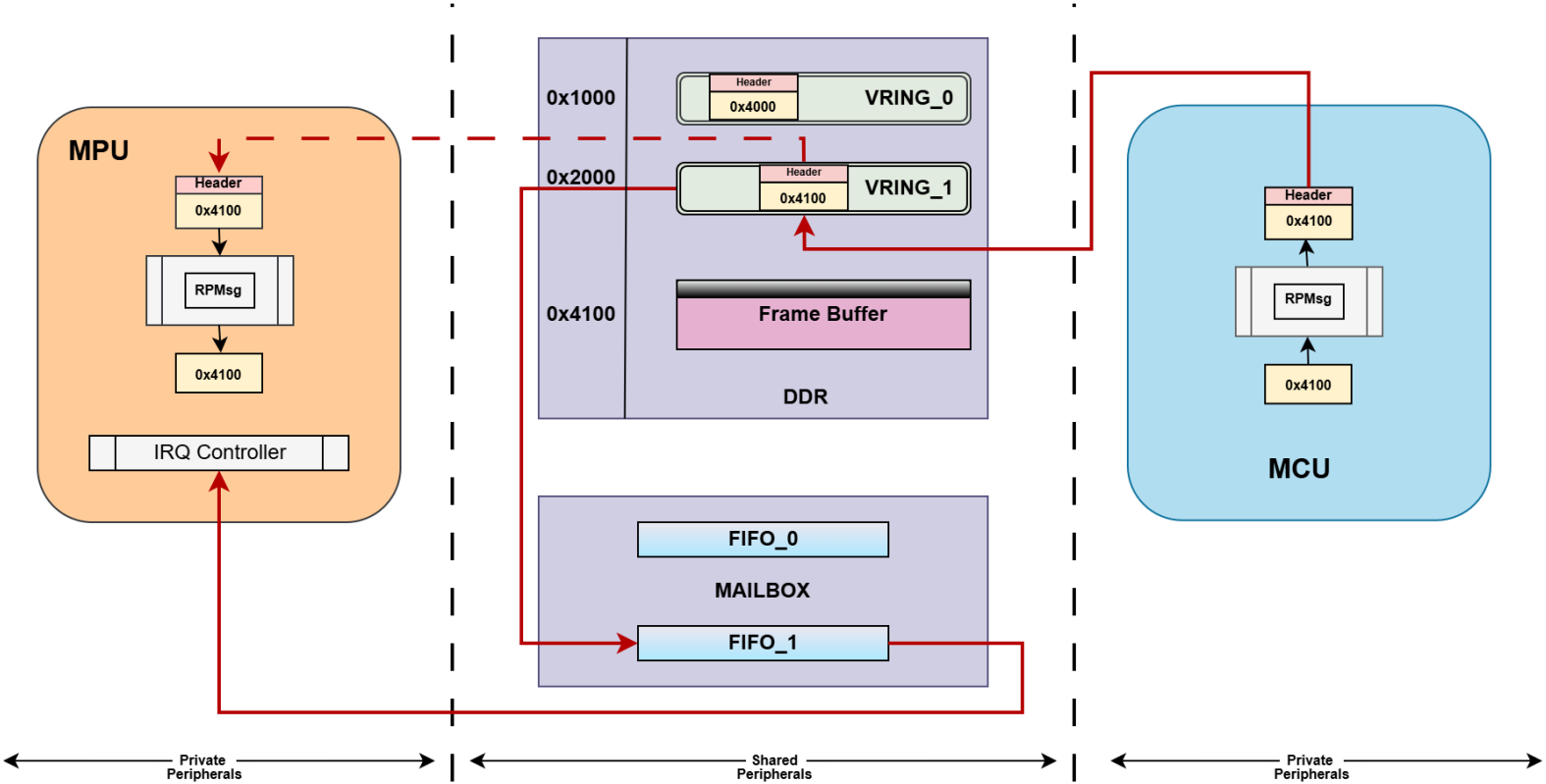


Fig.4: Data flow with RPMsg from MCU to MPU

Texas Instruments

Solutions for Peripheral Sharing | Virtio

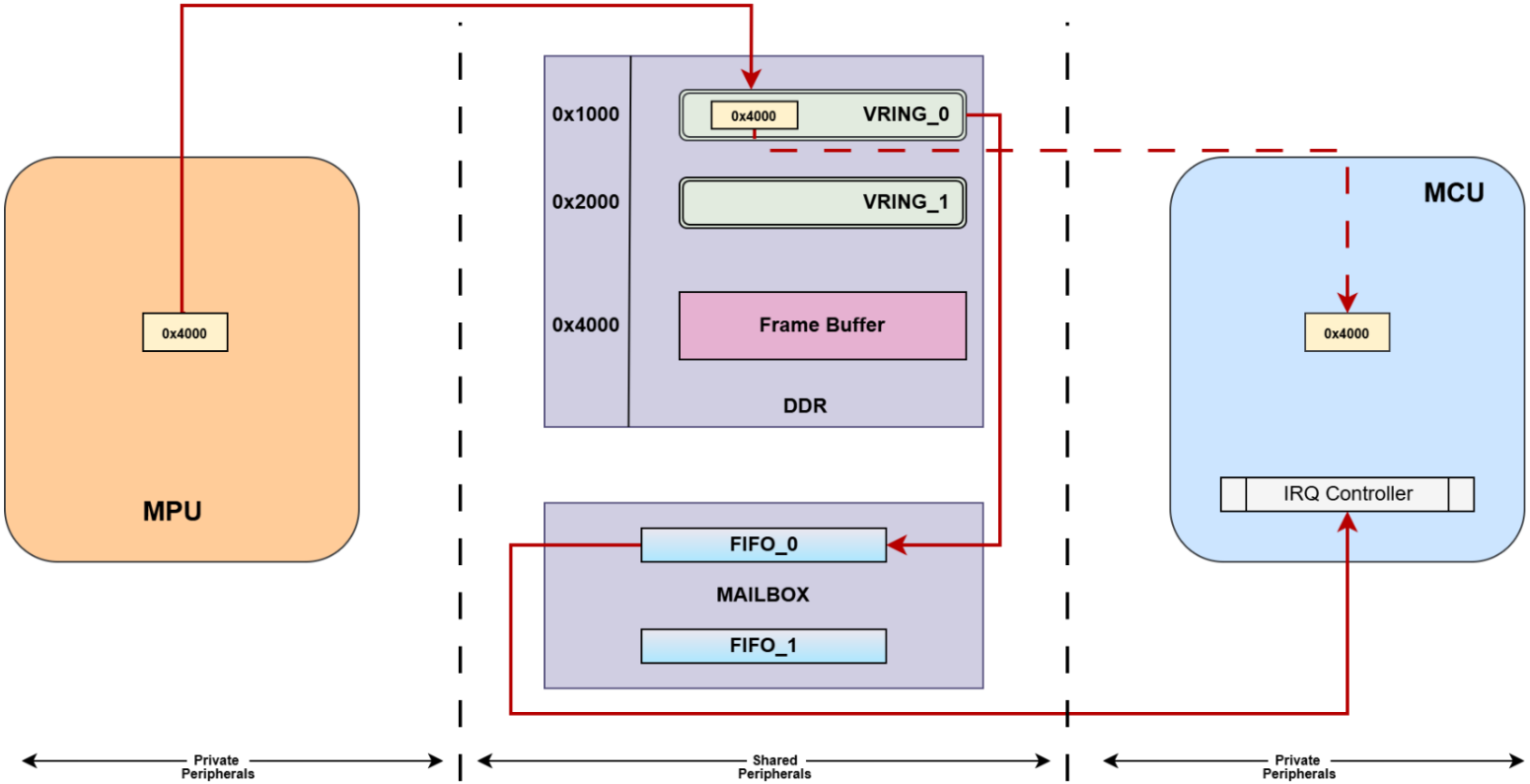


Fig.5: Data flow with Virtio from MCU to MPU

Texas Instruments

Solutions for Peripheral Sharing | Virtio

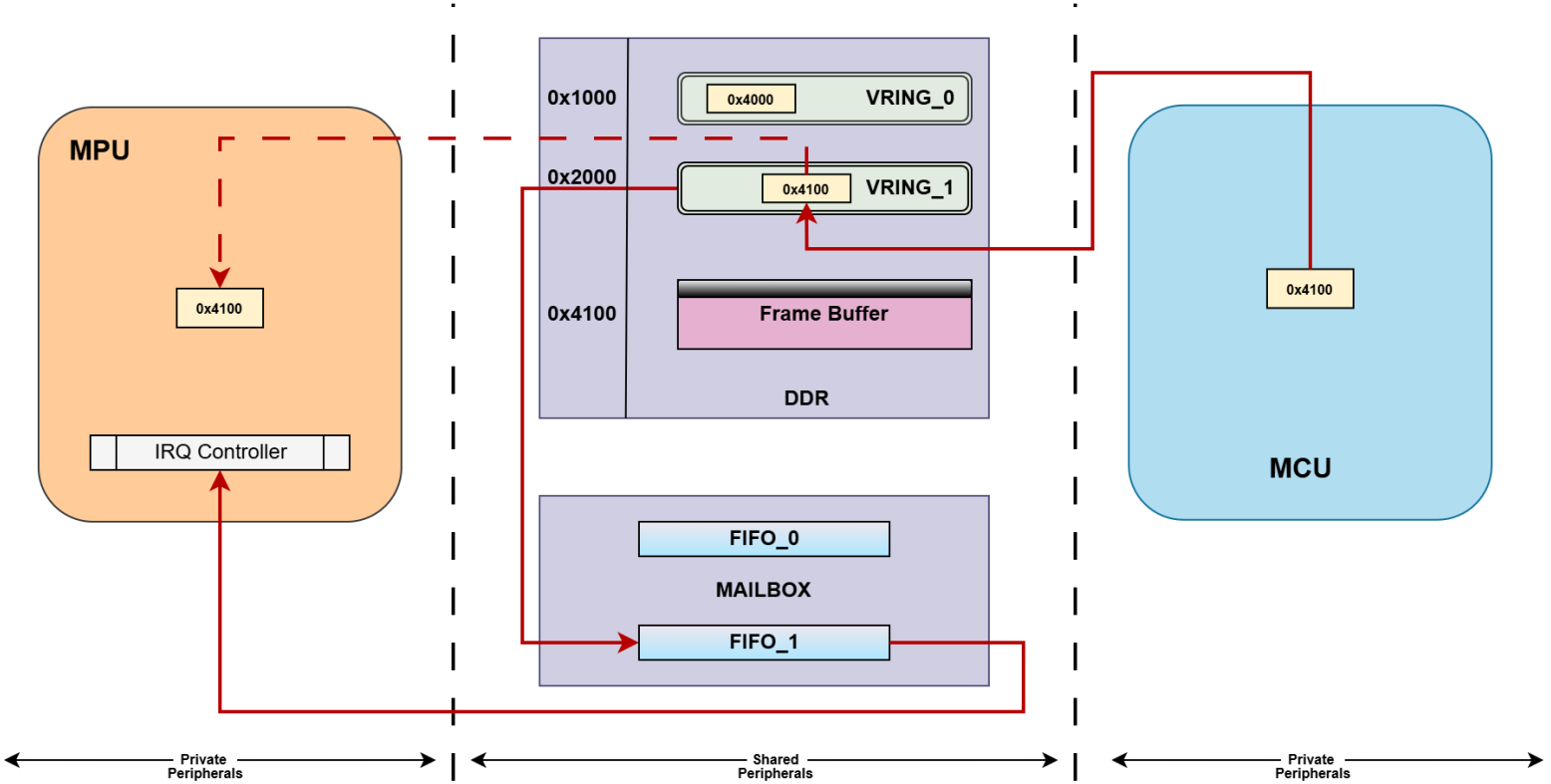


Fig.6: Data flow with Virtio from MPU to MCU

RPMsg Channel vs Endpoint

- Channel is the logical link between two communicating entities
- Endpoint is the final port through which data transfers happen
- Each endpoint is assigned a unique address and bound to a callback function

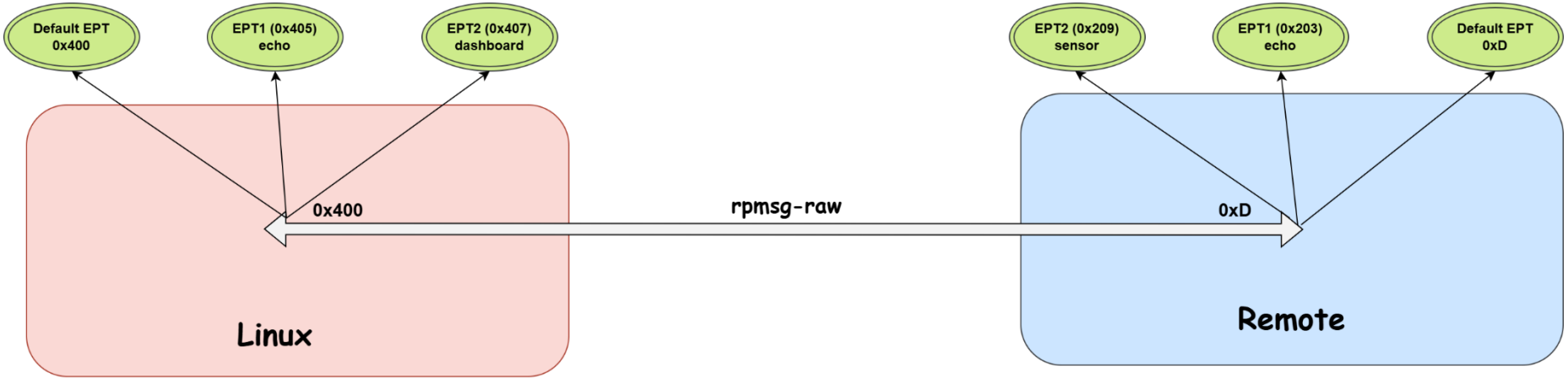


Fig. 7: RPMsg channels and endpoints

Texas Instruments

RPMsg Channel vs Endpoint

```
struct rpmsg_device {
        struct device dev;
        struct rpmsg_device_id id;
        const char *driver_override;
        u32 src;
        u32 dst;
        struct rpmsg_endpoint *ept;
        bool announce;
        bool little_endian;
        const struct rpmsg_device_ops *ops;
};
```

Fig. 8: Data Structure of RPMsg channel

```
struct rpmsg_endpoint {
        struct rpmsg_device *rpdev;
        struct kref refcount;
        rpmsg_rx_cb_t cb;
        rpmsg_flowcontrol_cb_t flow_cb;
        struct mutex cb_lock;
        u32 addr;
        void *priv;

        const struct rpmsg_endpoint_ops *ops;
};
```

Fig. 9: Data Structure of RPMsg endpoint

Texas Instruments

RPMsg Packet layout

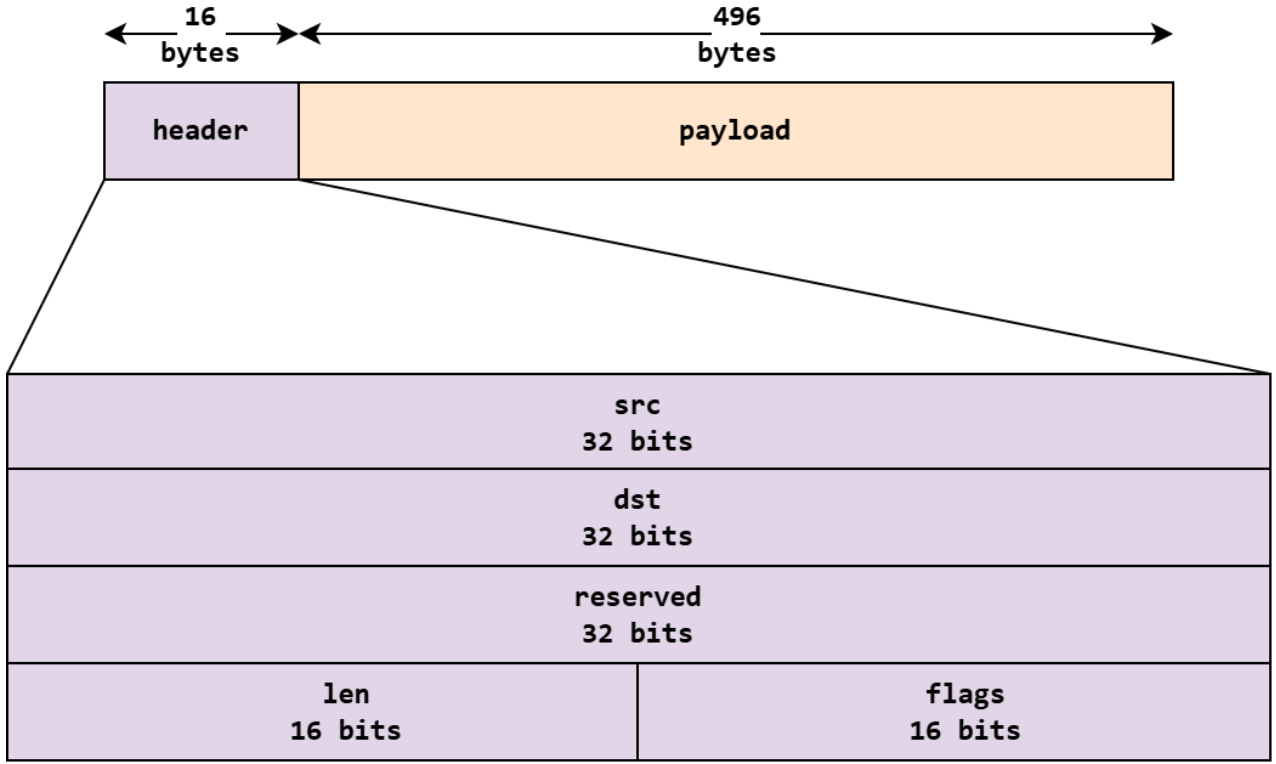


Fig. 10: RPMsg Packet layout

Texas Instruments

Timing Diagram

Texas Instruments

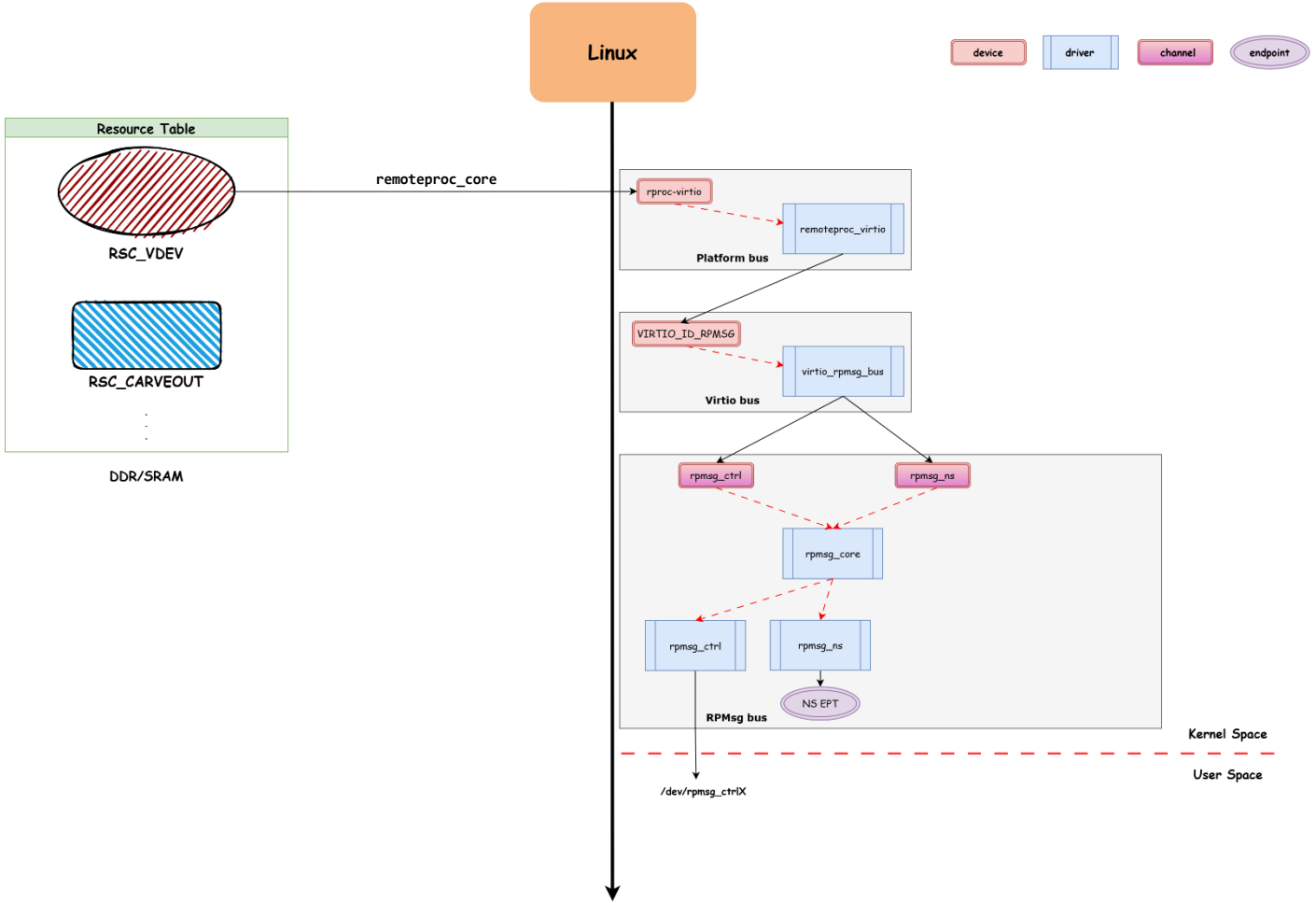


Fig. 11: Creation of static endpoints and channels from RSC table

Texas Instruments

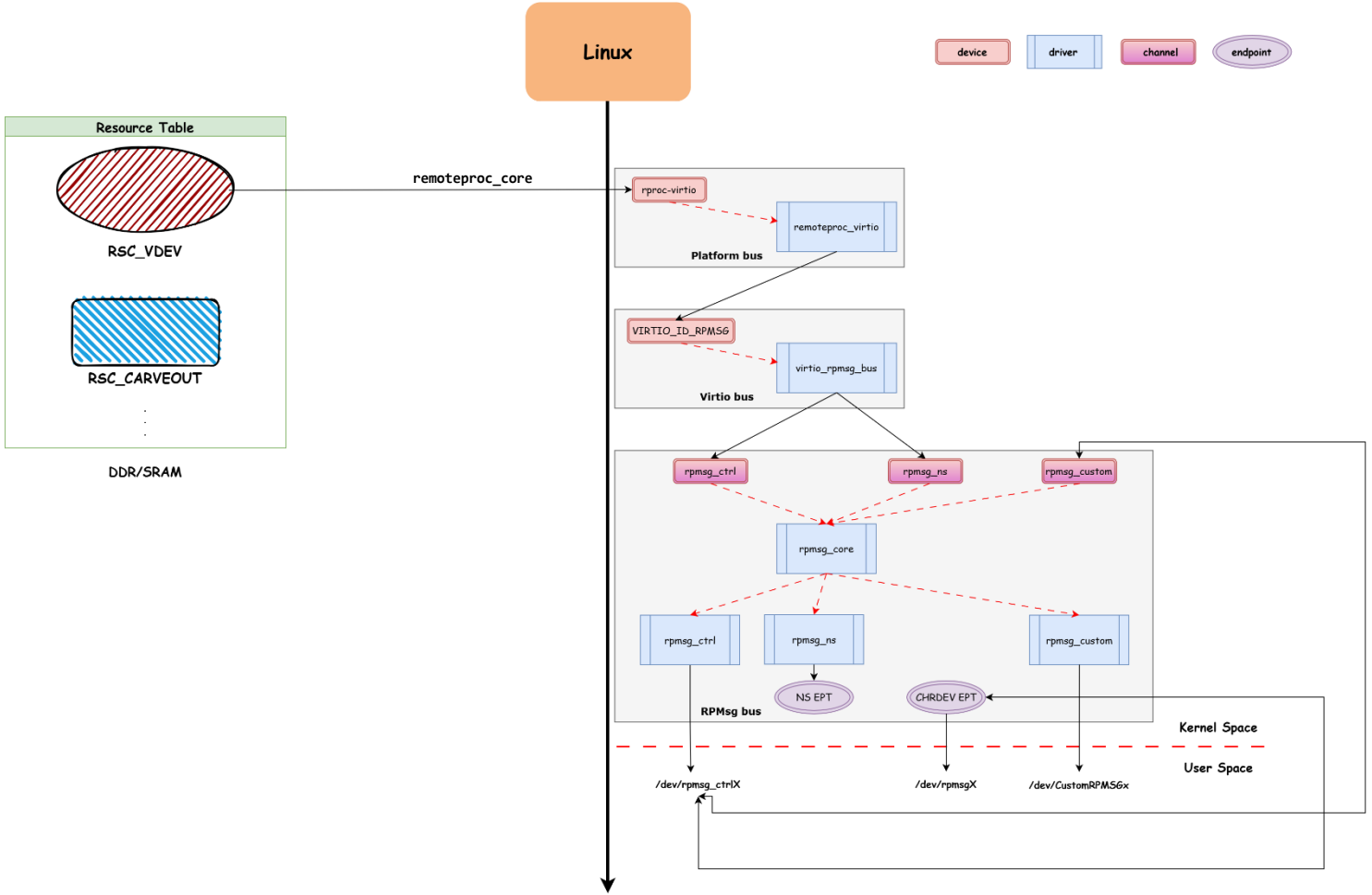


Fig. 12: Creation of endpoints and channels from userspace

Texas Instruments

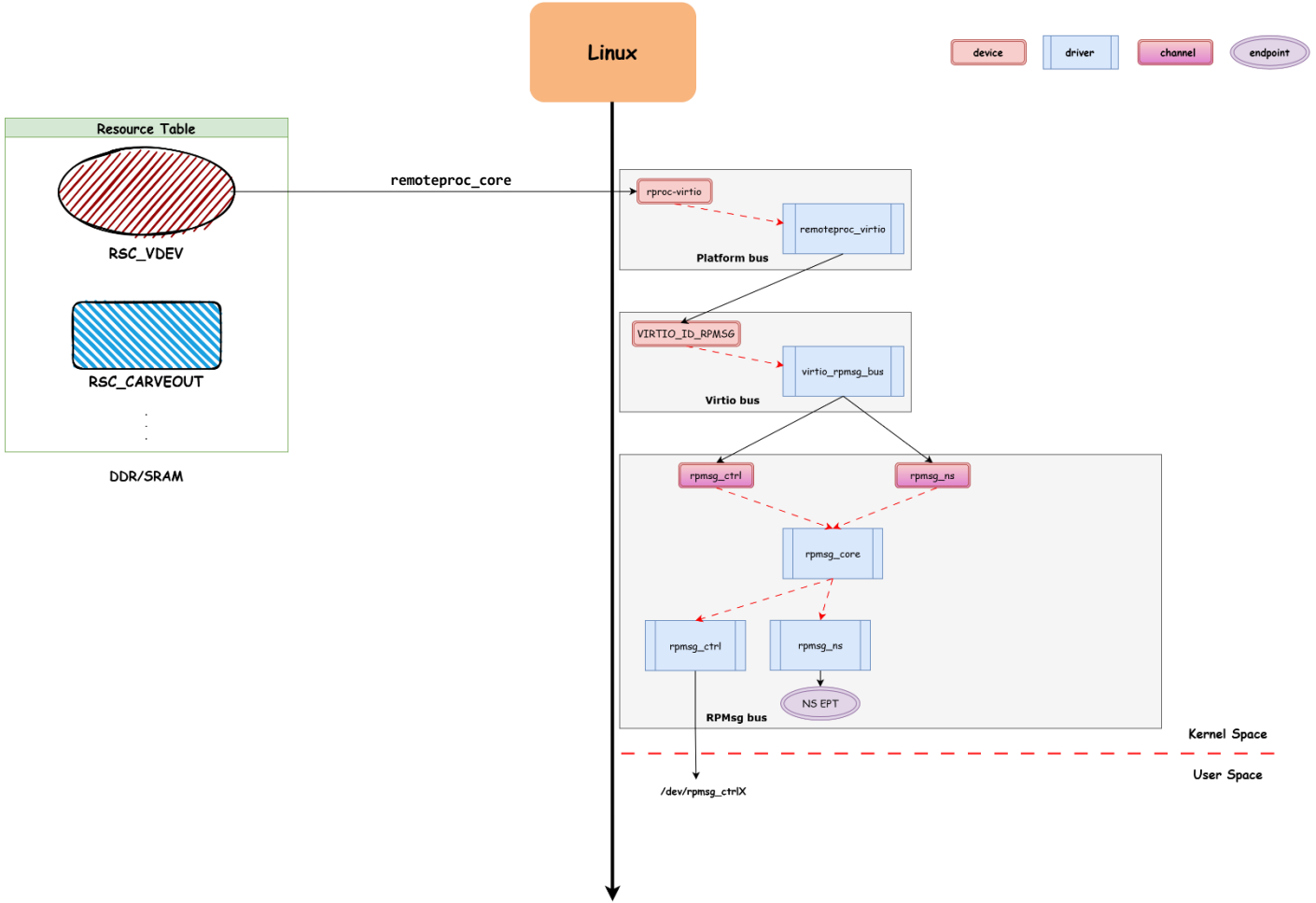


Fig. 13: Creation of static endpoints and channels from RSC table

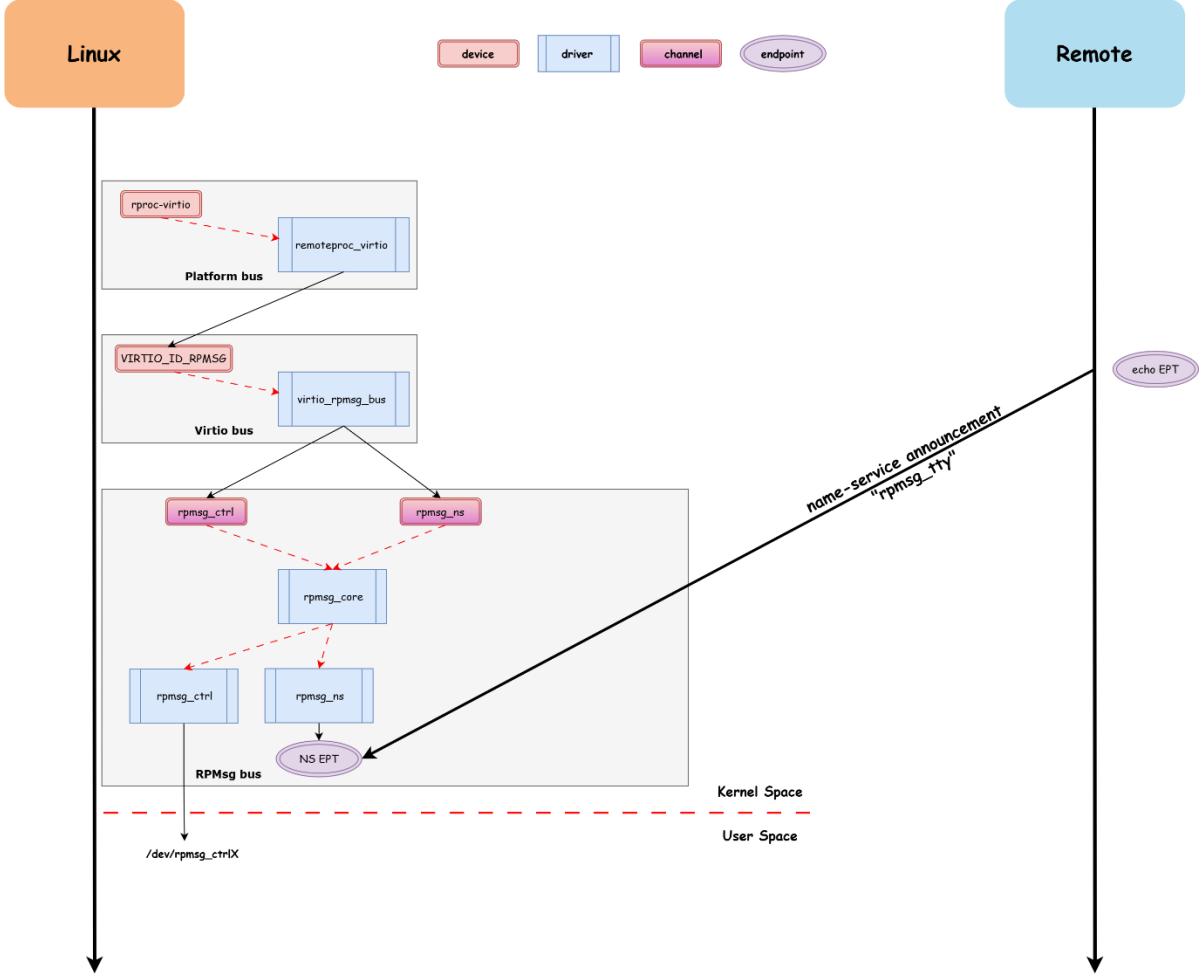


Fig. 14: Remote processor announcing RPMsg channel

Texas Instruments

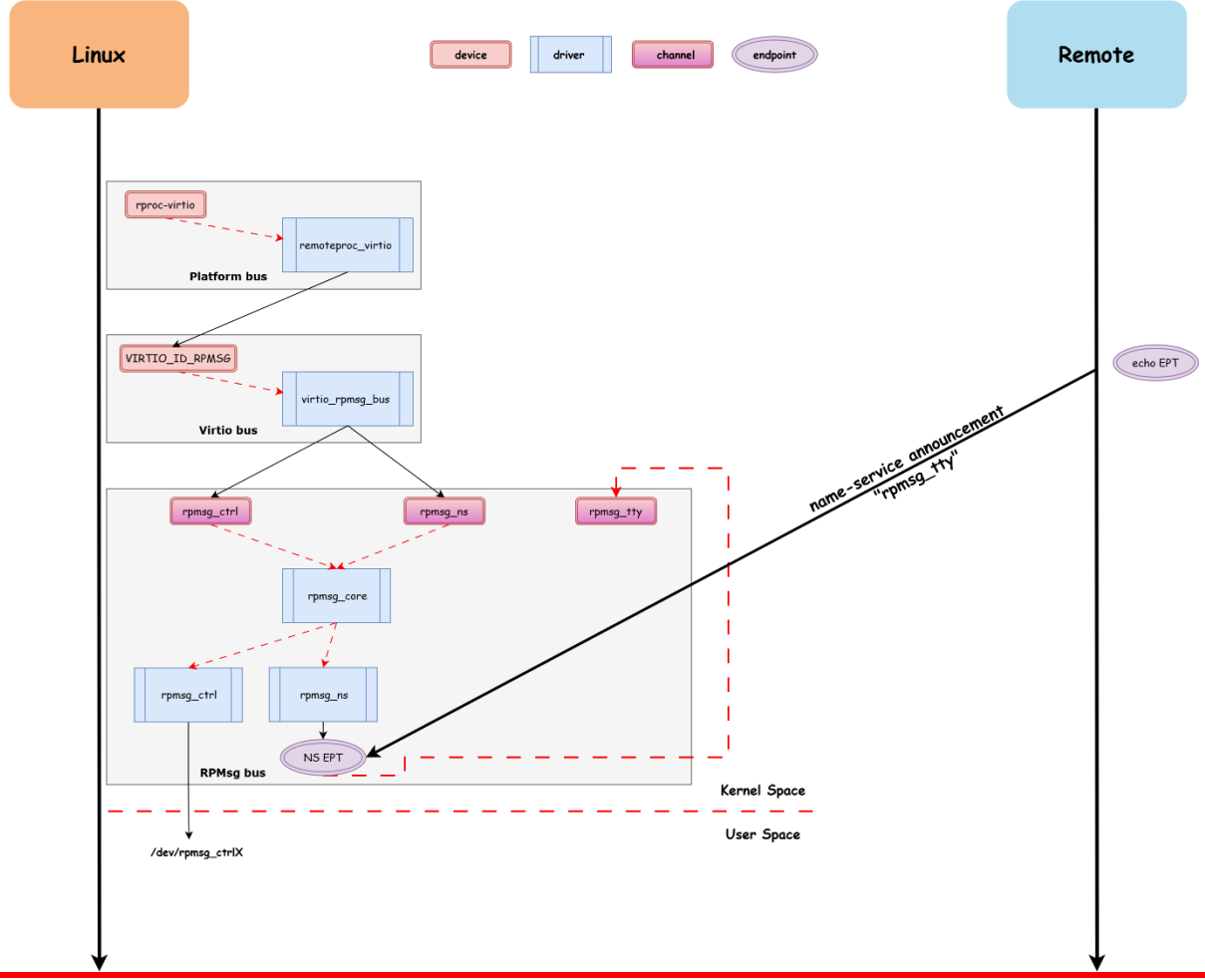


Fig. 15: rpmsg_ns creating RPMsg device

Texas Instruments

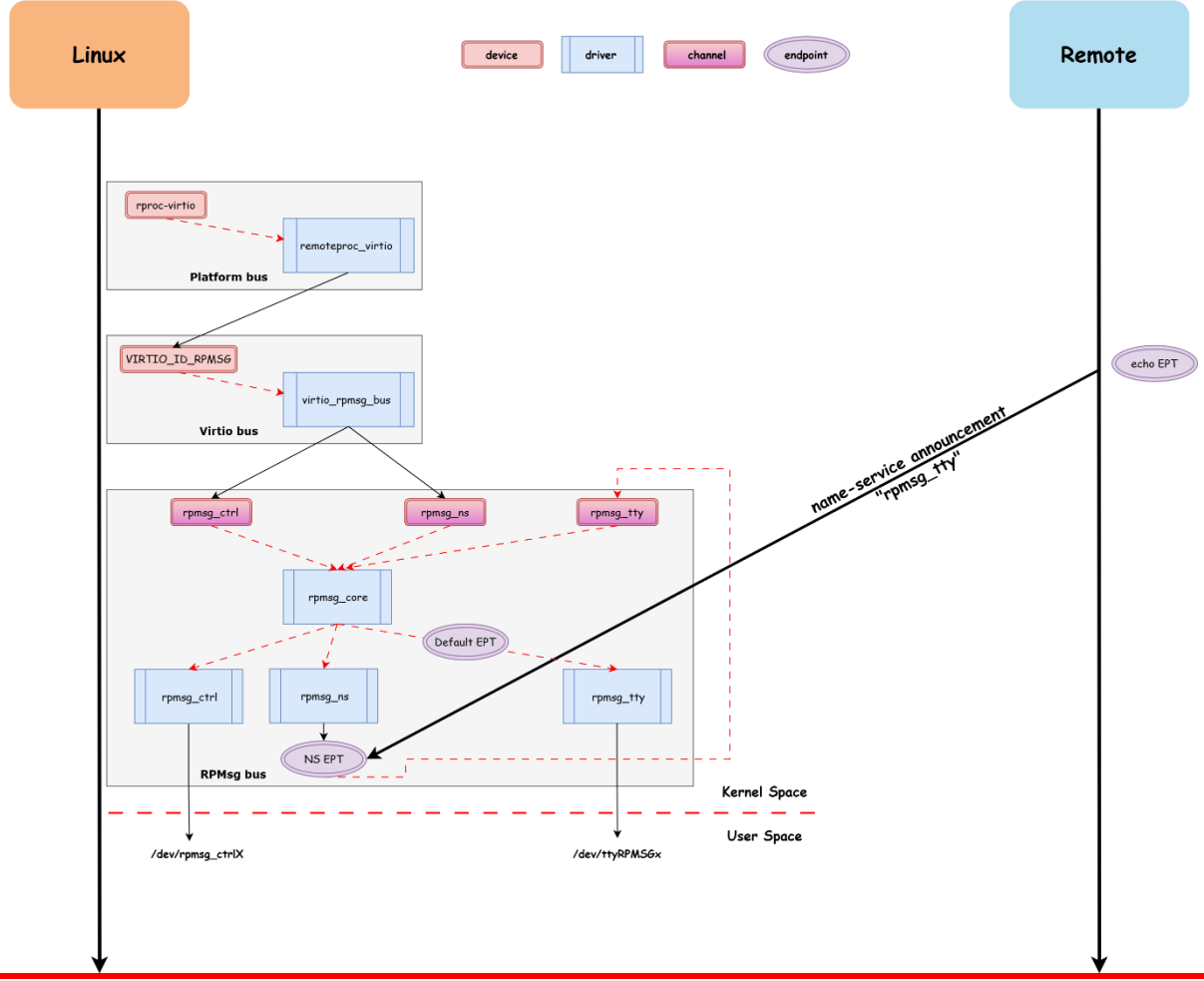


Fig. 16: rpmsg_core creating a default endpoint and binding the driver

Texas Instruments

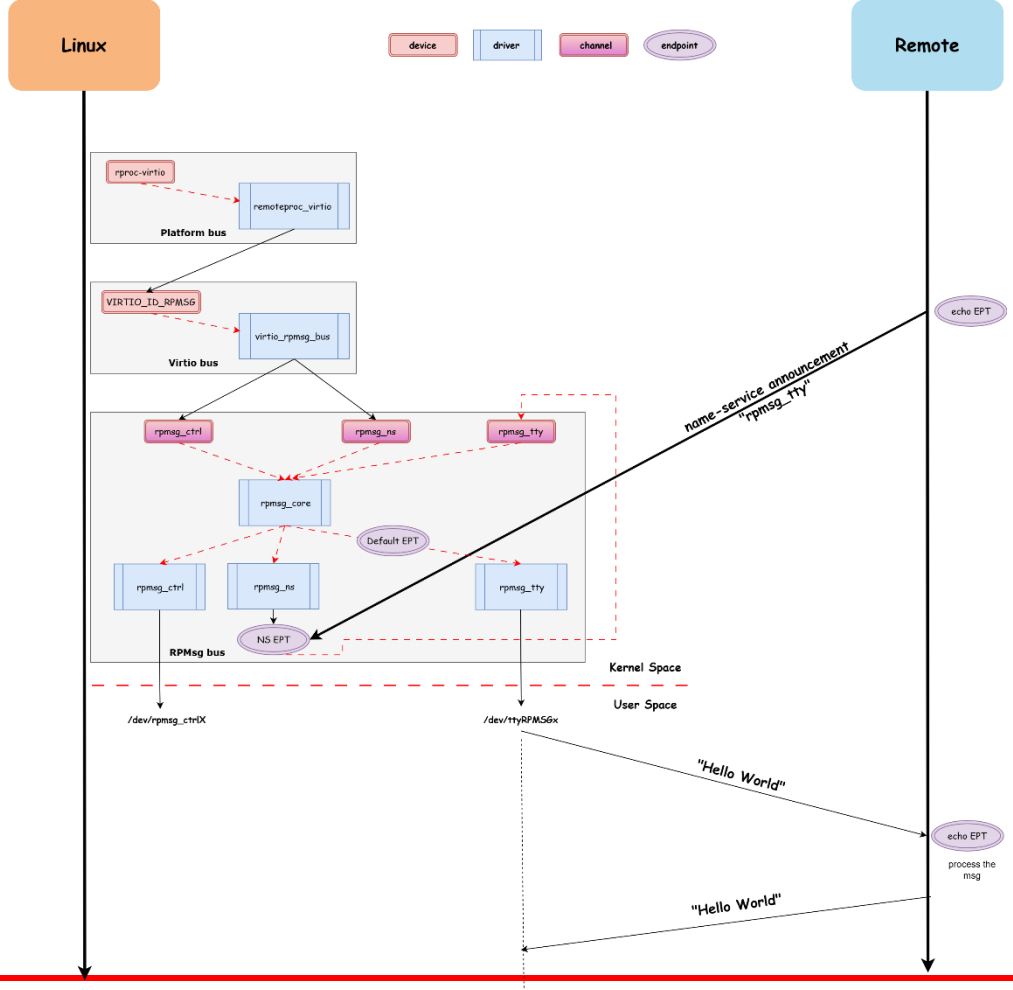


Fig. 17: IPC through virtual device exposed to userspace

Static & Dynamic Endpoints/Channels

- Dynamic Endpoints
  - Created at runtime
  - Dynamic address allocation
  - Pass RPMSG_ADDR_ANY flag while creating ept
  - Ex: Default ept of RPMsg channel, epts created through rpmsg_ctrl device
- Static Endpoints
  - Hardcoded in source
  - Fixed address allocation
  - Pass u32 addr while creating ept
  - Ex: Name service reserved ept (53)
- Dynamic Channel
  - Announced at runtime
  - Dynamic address allocation
  - Ex: rpmsg-tty, channels created through rpmsg_ctrl device
- Static Channel
  - Hardcoded in source
  - Fixed address allocation
  - Ex: rpmsg_ctrl, rpmsg_ns

Texas Instruments

Design Principles

Texas Instruments

Design Principles #1 | DT Enumeration

- Only needed so other nodes can take a reference
- *Do not* enumerate devices in DT if the peripheral is not a provider (e.g. serial)
  - This allows to make the name service completely dynamic
- For provider devices (e.g. gpio, i2c), create a topology of rpmsg -> channel -> devices under remote processor node

```
cm33: remoteproc-cm33 {
        compatible = "fsl,imx8ulp-cm33";
        status = "disabled";
+
+       rpmsg {
+               rpmsg-io {
+                       #address-cells = <1>;
+                       #size-cells = <0>;
+
+                       rpmsg_gpioa: gpio@0 {
+                               compatible = "rpmsg-gpio";
+                               reg = <0>;
+                               gpio-controller;
+                               #gpio-cells = <2>;
+                               #interrupt-cells = <2>;
+                               interrupt-controller;
+                       };
+
+                       rpmsg_gpiob: gpio@1 {
+                               compatible = "rpmsg-gpio";
+                               reg = <1>;
+                               gpio-controller;
+                               #gpio-cells = <2>;
+                               #interrupt-cells = <2>;
+                               interrupt-controller;
+                       };
+               };
+       };
};
```

Fig. 18: Devicetree topology for provider peripherals

Texas Instruments

Design Principles #2 | Decouple Remoteproc from RPMsg

- **remoteproc ->** should only be responsible to power on the remote processor
- Should *not* have any knowledge of the services offered by the remote processor

Don’ts

- Peripheral specific initialization in remoteproc
- Registering remoteproc in the rpmsg bus and the actual rpmsg driver in the platform bus
- Binding application running on the remote processor to the remoteproc driver
- Forces to make the remoteproc driver as a “server” of all its bound rpmsg drivers

Do’s

- Model everything in individual RPMsg drivers
- Remoteproc driver has no knowledge of services offered by application

Design Principles #2 | Decouple Remoteproc from RPMsg

- Example of bad design:

```
+static const struct of_device_id imx_rpmsg_gpio_dt_ids[] = {
+       { .compatible = "fsl,imx-rpmsg-gpio" },
+       { /* sentinel */ }
+};
+
+static struct platform_driver imx_rpmsg_gpio_driver = {
+       .driver = {
+               .name = "gpio-imx-rpmsg",
+               .of_match_table = imx_rpmsg_gpio_dt_ids,
+       },
+       .probe = imx_rpmsg_gpio_probe,
+};
+
+static int __init gpio_imx_rpmsg_init(void)
+{
+       return platform_driver_register(&imx_rpmsg_gpio_driver);
+}
+
+device_initcall(gpio_imx_rpmsg_init);
```

Fig. 19: rpmsg-gpio driver registered on the platform bus

```
+static char *channel_device_map[][2] = {
+       {"rpmsg-io-channel", "fsl,imx-rpmsg-gpio"},
+       {"rpmsg-i2c-channel", "fsl,imx-rpmsg-i2c"},
+};
+
+static int imx_rpmsg_endpoint_cb(struct rpmsg_device *rpdev,
+       void *data, int len, void *priv, u32 src)
+{
+       struct imx_rpmsg_driver_data *drvdata;
+
+       drvdata = dev_get_drvdata(&rpdev->dev);
+       if (drvdata && drvdata->rx_callback)
+               return drvdata->rx_callback(rpdev, data, len, priv, src);

+               rp_driver->rpdrv.drv.name = channel_device_map[i][0];
+               rp_driver->rpdrv.id_table = rpdev_id;
+               rp_driver->rpdrv.probe = imx_rpmsg_endpoint_probe;
+               rp_driver->rpdrv.remove = imx_rpmsg_endpoint_remove;
+               rp_driver->rpdrv.callback = imx_rpmsg_endpoint_cb;
+               rp_driver->driver_data = driver_data;
+
+               register_rpmsg_driver(&rp_driver->rpdrv);

+       drvdata->rpdev = rpdev;
+       auxdata[0].compatible = channel_device_map[i][1];
+       auxdata[0].platform_data = drvdata;
+       dev_set_drvdata(dev, drvdata);
+
+       of_platform_populate(drvdata->channel_node, NULL, auxdata, dev);
```

Fig. 20: remoteproc driver registered on the rpmsg bus

Texas Instruments

Design Principles #2 | Decouple Remoteproc from RPMsg

- Example of good design:

```
+static const struct of_device_id rpmsg_gpio_dt_ids[] = {
+       { .compatible = "rpmsg-gpio" },
+       { /* sentinel */ }
+};
+
+static struct rpmsg_device_id rpmsg_gpio_channel_id_table[]
+       { .name = "rpmsg-io" },
+       { },
+};
+MODULE_DEVICE_TABLE(rpmsg, rpmsg_gpio_channel_id_table);
+
+static struct rpmsg_driver rpmsg_gpio_channel_client = {
+       .callback       = rpmsg_gpio_channel_callback,
+       .id_table       = rpmsg_gpio_channel_id_table,
+       .probe          = rpmsg_gpio_channel_probe,
+       .drv            = {
+               .name   = KBUILD_MODNAME,
+               .of_match_table = rpmsg_gpio_dt_ids,
+       },
+};
+module_rpmsg_driver(rpmsg_gpio_channel_client);
```

Nothing in remoteproc

Fig. 21: rpmsg-gpio driver registered on the rpmsg bus

Texas Instruments

Design Principles #3 | Channels

Don’ts

- Announcing a single name service for all types of peripherals controlled by remote processor. Ex- rpmsg-remoteproc
- Tight coupling of services offered by remoteproc
- Forces to maintain redundant info in message payload: msg_type

Do’s

- Announce a name service for each device type supported. Ex-
  - rpmsg-gpio
  - rpmsg-tty
  - rpmsg-i2c
- Loosely coupled system
- Allows lifecycle of each device type to be controlled independently

Texas Instruments

Design Principles #4 | Endpoints

Don’ts

- Having a single endpoint for multiple instances of the same device
- Forces to maintain redundant info in message payload: device index
- Forces to maintain redundant info in internal driver structures: map of instance index to per device pointer
- Forces to maintain separate IDR in driver for indexes

Do’s

- One dedicated endpoint per device instance
- Endpoint address can be re-used as device index
- Firmware can announce new channel request per device instance as a default endpoint gets created per request:
  - rpmsg-tty1
  - rpmsg-tty2

Design Principles #4 | Endpoints

- How does multiple name service announcements work?
  - rpmsg bus does not allow the creation of duplicate name-service
  - Traditionally, firmware needs to announce “rpmsg-tty1”, “rpmsg-tty2”
  - Which can lead to hardcodings in Linux driver

```
diff --git a/drivers/tty/rpmsg_tty.c b/drivers/tty/rpmsg_tty.c
index c5fd6d9b380fc..9513b71b95c6b 100644
--- a/drivers/tty/rpmsg_tty.c
+++ b/drivers/tty/rpmsg_tty.c
@@ -215,7 +215,9 @@ static void rpmsg_tty_remove(struct rpmsg_device *rpdev)
 }

 static struct rpmsg_device_id rpmsg_driver_tty_id_table[] = {
-       { .name = "rpmsg-tty" },
+       { .name = "rpmsg-tty0" },
+       { .name = "rpmsg-tty1" },
+       { .name = "rpmsg-tty2" },
        { },
 };
```

Fig. 22: Traditional way of channel enumeration in rpmsg driver

Texas Instruments

Design Principles #4 | Endpoints

- How does multiple name service announcements work?
  - Ongoing discussions to introduce wildcard matching for name service
  - Allows to match any instance of the name service without driver hard-codings

```
diff --git a/drivers/rpmsg/rpmsg_core.c b/drivers/rpmsg/rpmsg_core.c
index e330ec4dfc33..bfd25978fa35 100644
--- a/drivers/rpmsg/rpmsg_core.c
+++ b/drivers/rpmsg/rpmsg_core.c
@@ -399,7 +399,25 @@ ATTRIBUTE_GROUPS(rpmsg_dev);
 static inline int rpmsg_id_match(const struct rpmsg_device *rpdev,
                                  const struct rpmsg_device_id *id)
 {
-       return strncmp(id->name, rpdev->id.name, RPMSG_NAME_SIZE) == 0;
+       size_t len = min_t(size_t, strlen(id->name), RPMSG_NAME_SIZE);
+
+       /*
+        * Allow for wildcard matches.  For example if rpmsg_driver::id_table
+        * is:
+        *
+        * static struct rpmsg_device_id rpmsg_driver_sample_id_table[] = {
+        *      { .name = "rpmsg-client-sample" },
+        *      { },
+        * }
+        *
+        * Then it is possible to support "rpmsg-client-sample*", i.e:
+        *      rpmsg-client-sample
+        *      rpmsg-client-sample_instance0
+        *      rpmsg-client-sample_instance1
+        *      ...
+        *      rpmsg-client-sample_instanceX
+        */
+       return strncmp(id->name, rpdev->id.name, len) == 0;
```

Fig. 23: Patch showing wildcard name service matching

Texas Instruments

Design Principles #5 | Send Function

- Send Functions can be invoked in interrupt context and time-sensitive paths

Don’ts

- Using *rpmsg_send()* functions
- Blocking call which can wait upto 15s for a TX buffer

Do’s

- Using *rpmsg_trysend()* functions
- Non blocking call, returns immediately with error

Texas Instruments

Design Principles #6 | Receive Function

- The rpmsg receive callback function is invoked in interrupt context
- As soon as this function returns, the sent buffer is recycled and ready to be used again for TX/RX

Don’ts

- Storing a pointer to received message and waking up sleeping task and returning from cb()
- Doing entire processing in this function.

Do’s

- memcpy the received buffer into a local var and wake up the sleeping thread with pointer to the local var.

Texas Instruments

Design Principles #7 | Synchronization

- RPMsg packets can arrive out of order at destination due to various reasons like
  - Availability of virtqueue
  - Roundtrip message latency
  - Task wakeup time etc.
- Without synchronization, it is possible a dependent message gets processed before its parent message leading to wrong programming.
- 2 ways to implement that:
  - Maintaining seq numbers in the payloads
  - Acquiring a lock before sending a message and releasing it after receiving &
  - Acquiring a lock before processing a message in rpmsg_cb and releasing it after processing

Texas Instruments

Design Principles #7 | Synchronization

- Example scenario without locking:
  - T1: rpmsg_send()
  - T1: virtqueue_get_buf()
    - T2: rpmsg_send()
    - T2: virtqueue_get_buf()
    - T2: virtqueue_add_outbuf() // Message is inserted in available queue
  - T1: virtqueue_add_outbuf() // Message is inserted in available queue
  - T1: virtqueue_kick()
    - T2: virtqueue_kick()

- T2's message gets processed first at remote side which can lead to erroneous programming.

Texas Instruments

Design Principles #8 | Sharing Addresses OTW

- Different processors may have different memory maps

Don’ts

- Sending Physical Addresses over the wire
- Can be sniffed to leak Physical Addresses
- Remote processor is not to be trusted to access all the DDR

Do’s

- Sending Device Addresses over the wire
- Remoteproc framework has helpers to convert DA to VA and vice versa
- Ensures each processing entity receives addresses corresponding to its own memory map

Texas Instruments

Design Principles #9 | Re-using virtio design

- A virtio counterpart already exists for almost all of the peripherals

Don’ts

- Re-invent new stuff in RPMsg-based design

Do’s

- Adopt almost everything from virtio counterpart, including message and error formats
- Keeps the maintenance relatively easier

Texas Instruments

Design Principles #10 | Logging

- Incase the remote processor crashes, printing error messages can cause lot of prints

Don’ts

- Excessive logging (like msg drop)
- Using dev_info(), dev_dbg() versions
- Risk of DoS and log flooding

Do’s

- Logging only necessary events
- Using the ratelimited() version for logging to avoid printing same thing and filling the kernel buffer.

Texas Instruments

Comparative Study with VirtIO | Application

- rpmsg-tty was already upstreamed in both Linux and Zephyr
- virtio_console driver interface was available in Linux, but not in Zephyr
- Implemented a virtio_console device interface in Zephyr:
  - Registers VIRTIO_ID_RPROC_SERIAL VDEV in resource table
  - 16 descriptors per VRING
  - Receive a msg from Linux, memcpy into TX Vring and echo back.

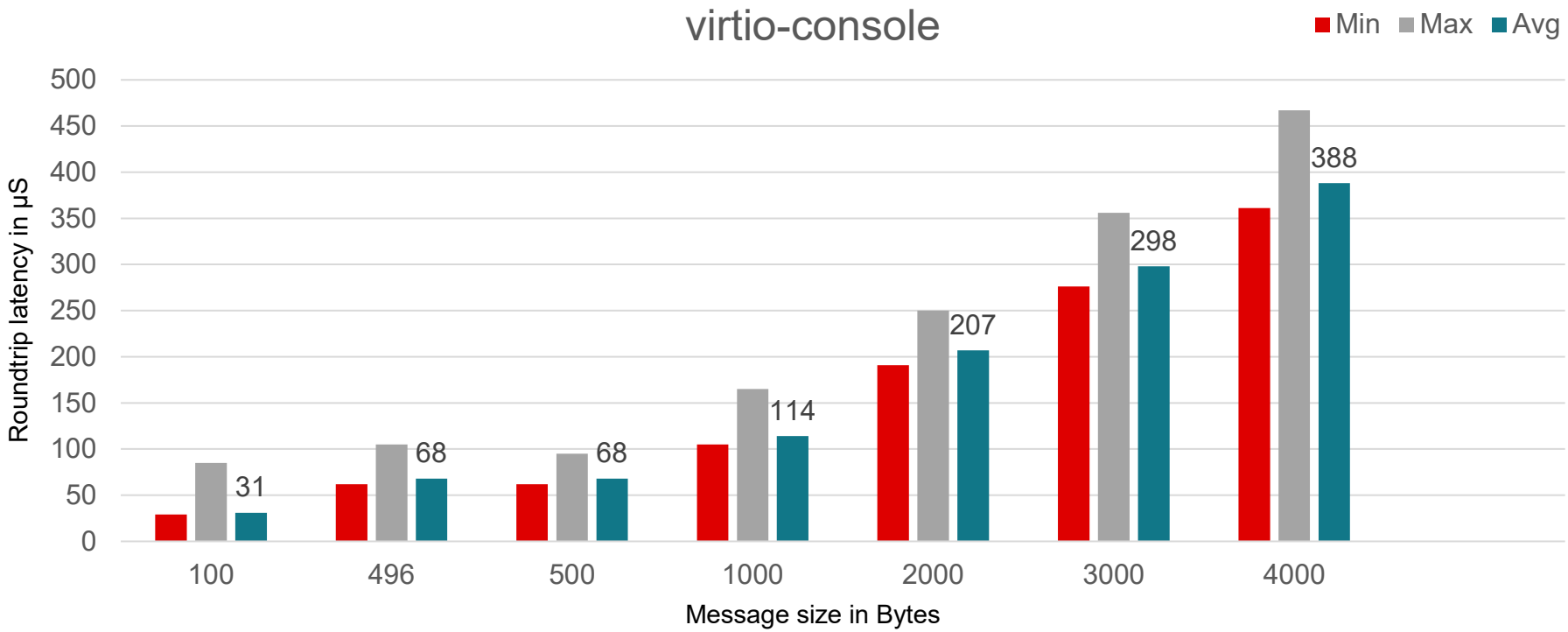
Texas Instruments

Comparative Study with VirtIO | Tracing latencies

- Platform: TI AM62x SK running Zephyr on Cortex M4
- Kernel space:
  - Get timestamp in rpmsg/virtio_send() function after doing copy_from_user()
  - Get timestamp in rpmsg_recv_single()
  - Calculate difference and expose to userspace
- User space:
  - Open the console port device for read and write
  - Write 1000 messages to fd
  - Read fd and validate echo responses of all 1000 messages
- Ensures we are capturing *only* the roundtrip latency for message transmission

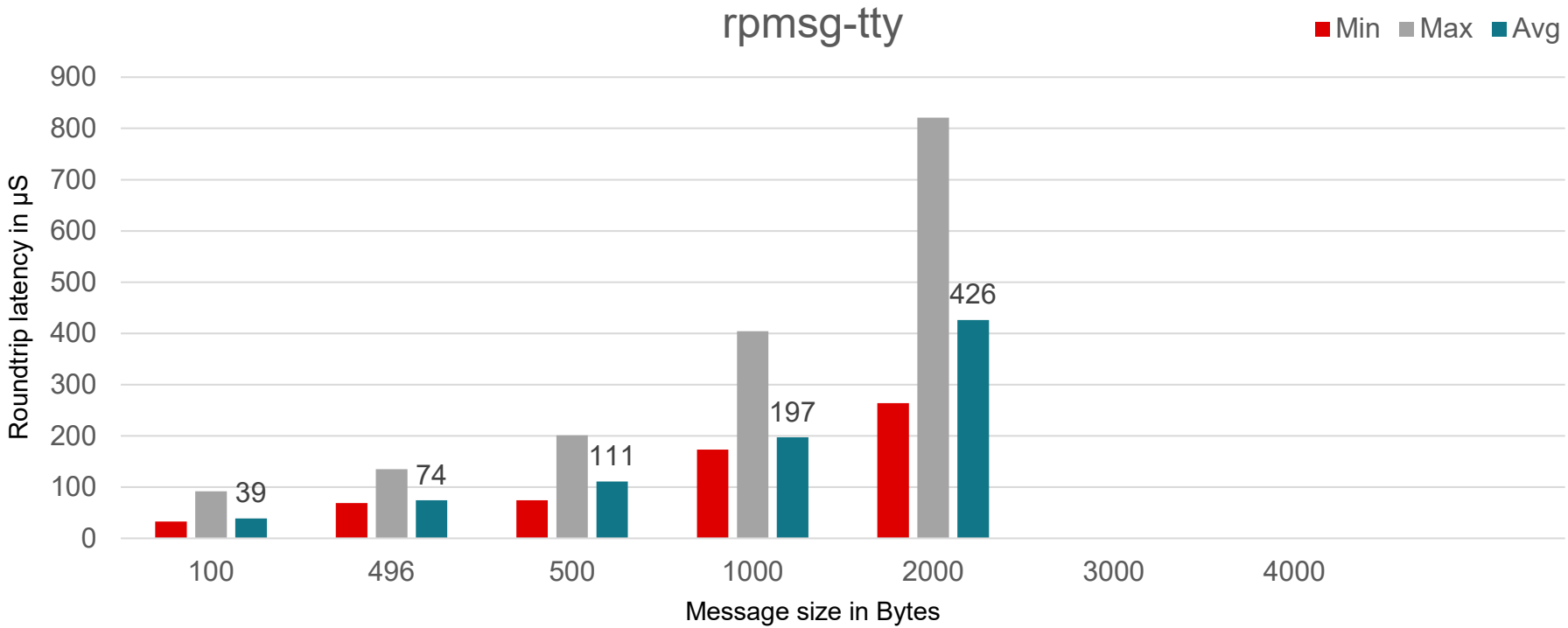
Texas Instruments

Comparative Study with VirtIO | Latencies



Trend: virtio-console is almost constant, overhead is memcpy

Comparative Study with VirtIO | Latencies



Trend: rpmsg-tty grows linearly with msg size

Comparative Study with VirtIO | Latencies

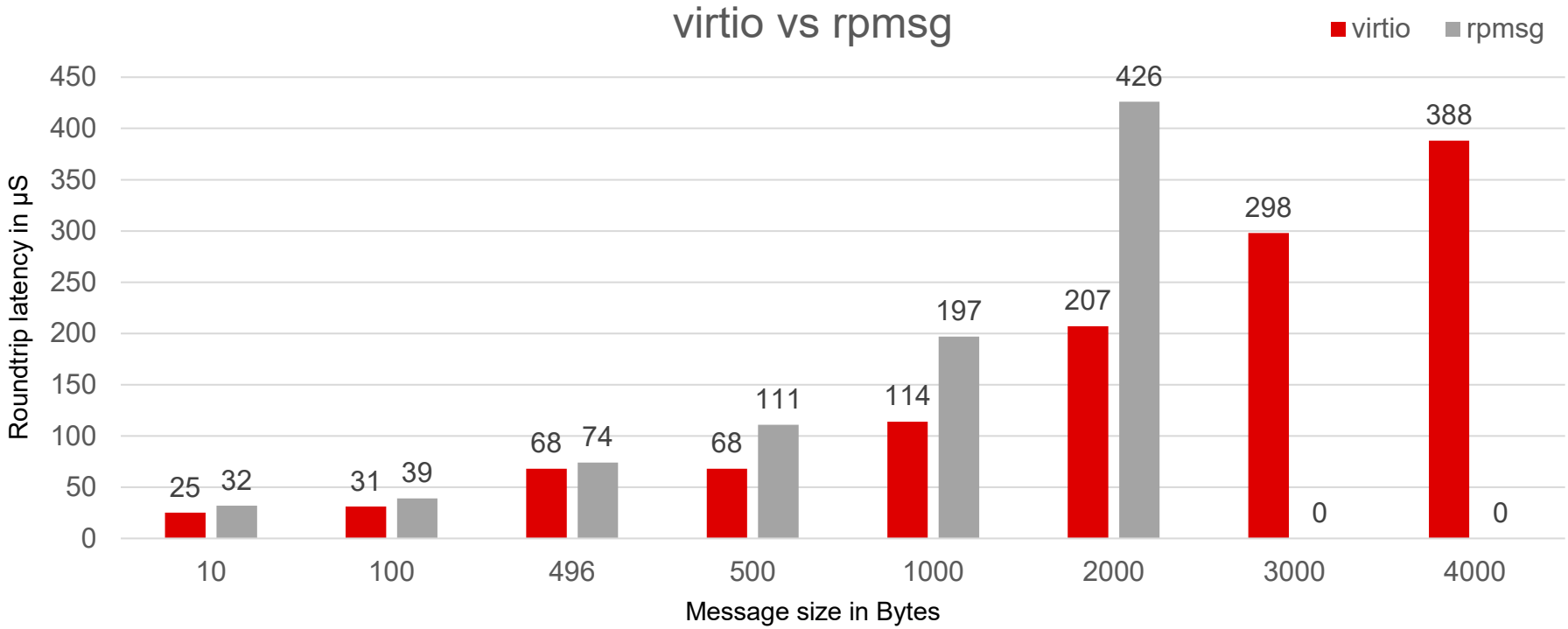


Fig. 25: Chart showing virtio vs rpmsg latencies

Comparative Study with VirtIO | Starvation in RPMsg

- RPMsg can introduce a risk of starvation incase of heavy traffic flow
- Bottleneck = fragmentation of descriptor due to small size (512 bytes)

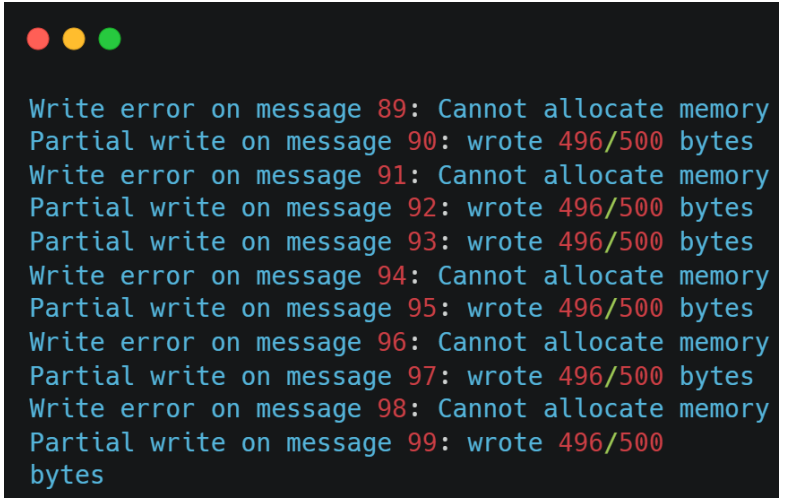


Fig. 26: Logs showing no available buffer for TX

Comparative Study with VirtIO | Resources

- Virtio requires atleast a pair of VRINGS per **device instance**
  - Can be more depending on the specific virtio implementation
  - The VRINGs can not be re-used for other devices and must be dedicated to a singe instance
- Virtio requires a dedicated mailbox instance as well for each device instance
- RPMsg consumes less resources and provides multiplexing
- RPMsg always requires only a single pair of VRINGs and a single mailbox instance regardless of the number of device instances it services

Comparative Study with VirtIO | Trade-offs

Trade-offs	Virtio	RPMsg
Speed	Faster	Slower
Throughput	High	Less
Memory	High	Less
Starvation	Unlikely	Possible

Comparative Study with VirtIO | When to use what?

- Higher performance? -> Virtio
- Low latency requirement? -> Virtio
- Robustness? -> Virtio
- No memory constraint? -> Virtio
- Maintainability? -> Virtio
- Re-usability? -> Virtio
- Low memory constraint? -> RPMsg
- Single remote processor servicing multiple other things? -> RPMsg

Texas Instruments

Challenges #1 | Service dormant until remoteproc

- The name service is only announced by the remote processor when the IPC stack has been setup by Linux:
  - Registering the VIRTIO_RPMSG device
  - Allocating the VRINGs
  - Pre-filling half of the descriptors for RX VRING so remote processor can send messages
- This can be a blocker for Early Boot requirements:
  - Early Ethernet
  - Early Splash
- “Early Boot & Late Attach” can help here, but the service has to be driven by remote processor

Challenges #2 | Static Configuration for providers

- Defining RPMsg nodes in DT makes it a static configuration, tightly coupling Linux with that specific firmware.
- What happens if the remote processor is brought down?
- Or decide to run another firmware from userspace which does not advertise this service?
- At the same time, we do need this supplier device to be available in DT so that it can be referenced by other consumer nodes. Circular problem?

Texas Instruments

Upstreaming Lessons

- Evaluate virtio-interface for the usecase with latency and memory constraints
- If RPMsg interface is really needed:
  - Submit an RFC first with Upstream Linux
  - Expect firmware rework
  - Propose vendor-agnostic generic drivers!
  - Implies less places to fix bugs
  - Easier to maintain in long-term than maintaining individual drivers

Texas Instruments

Improvements

- Explore the usage of chained descriptors in RPMsg
- Instead of returning 512 as the MTU, allow chaining multiple descriptors next to each other.
- This allows to transfer larger memory buffers upto 2^{16} (64KiB) size
- Explore configuring the size of RPMsg packets, currently hardcoded to 512 Bytes, can be increased?

Texas Instruments

References

- rpmsg-gpio implementation - https://lore.kernel.org/all/20260422212849.1240591-4-shenwei.wang@nxp.com/
- rpmsg-tty implementation - https://lore.kernel.org/all/20211008153446.23188-1-arnaud.pouliquen@foss.st.com/
- rpmsg-net implementation - https://lore.kernel.org/all/20250911113612.2598643-1-danishanwar@ti.com/
- rpmsg-char implementation - https://github.com/torvalds/linux/blob/master/drivers/rpmsg/rpmsg_char.c
- virtio-console implementation - https://github.com/torvalds/linux/blob/master/drivers/char/virtio_console.c

Texas Instruments

Credits and Acknowledgement

- Texas Instruments Inc.
- The Linux Foundation.
- Mathieu Poirier
- Bjorn Andersson
- Arnaud Pouliquen
- Andrew Lunn
- Shenwei Wang
- Md Danish Anwar
- Ravi Gunasekaran
- Yojita Mallik

Q&A

- Contact Information:
  - Beleswar Padhi <b-padhi@ti.com>
- Also on IRC @ libera.chat #linux-ti

Learn more about TI products

- https://www.ti.com/linux
- https://www.ti.com/processors
- https://www.ti.com/edgeai

Why choose TI MCUs and processors?

Scalability

Our products offer scalable performance that can adapt and grow as the needs of your customers evolve.

Efficiency

We design products that extend battery life, maximize performance for every watt expended, and unlock the highest levels of system efficiency.

Affordability

We strive to make innovation accessible to all by creating cost-effective products that feature state-of-the-art technology and package designs.

Availability

Our investment in internal manufacturing capacity provides greater assurance of supply, supporting your growth for decades to come.