Remote Processor Messaging (rpmsg) Framework

> **Note**
>
> This document describes the rpmsg bus and how to write rpmsg drivers. To learn how to add rpmsg support for new platforms, check out remoteproc.txt (also a resident of Documentation/).

Introduction

Modern SoCs typically employ heterogeneous remote processor devices in asymmetric multiprocessing (AMP) configurations, which may be running different instances of operating system, whether it's Linux or any other flavor of real-time OS.

OMAP4, for example, has dual Cortex-A9, dual Cortex-M3 and a C64x+ DSP. Typically, the dual cortex-A9 is running Linux in a SMP configuration, and each of the other three cores (two M3 cores and a DSP) is running its own instance of RTOS in an AMP configuration.

Typically AMP remote processors employ dedicated DSP codecs and multimedia hardware accelerators, and therefore are often used to offload CPU-intensive multimedia tasks from the main application processor.

These remote processors could also be used to control latency-sensitive sensors, drive random hardware blocks, or just perform background tasks while the main CPU is idling.

Users of those remote processors can either be userland apps (e.g. multimedia frameworks talking with remote OMX components) or kernel drivers (controlling hardware accessible only by the remote processor, reserving kernel-controlled resources on behalf of the remote processor, etc..).

Rpmsg is a virtio-based messaging bus that allows kernel drivers to communicate with remote processors available on the system. In turn, drivers could then expose appropriate user space interfaces, if needed.

When writing a driver that exposes rpmsg communication to userland, please keep in mind that remote processors might have direct access to the system's physical memory and other sensitive hardware resources (e.g. on OMAP4, remote cores and hardware accelerators may have direct access to the physical memory, gpio banks, dma controllers, i2c bus, gptimers, mailbox devices, hwspinlocks, etc..). Moreover, those remote processors might be running RTOS where every task can access the entire memory/devices exposed to the processor. To minimize the risks of rogue (or buggy) userland code exploiting remote bugs, and by that taking over the system, it is often desired to limit userland to specific rpmsg channels (see definition below) it can send messages on, and if possible, minimize how much control it has over the content of the messages.

Every rpmsg device is a communication channel with a remote processor (thus rpmsg devices are called channels). Channels are identified by a textual name and have a local ("source") rpmsg address, and remote ("destination") rpmsg address.

When a driver starts listening on a channel, its rx callback is bound with a unique rpmsg local address (a 32-bit integer). This way when inbound messages arrive, the rpmsg core dispatches them to the appropriate driver according to their destination address (this is done by invoking the driver's rx handler with the payload of the inbound message).

User API

```
int rpmsg_send(struct rpmsg_channel *rpdev, void *data, int len);
```

sends a message across to the remote processor on a given channel. The caller should specify the channel, the data it wants to send, and its length (in bytes). The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel's src and dst addresses.

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio's used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_sendto(struct rpmsg_channel *rpdev, void *data, int len, u32 dst);
```

sends a message across to the remote processor on a given channel, to a destination address provided by the caller.

The caller should specify the channel, the data it wants to send, its length (in bytes), and an explicit destination address.

The message will then be sent to the remote processor to which the channel belongs, using the channel's src address, and the user-provided dst address (thus the channel's dst address will be ignored).

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio's used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_send_offchannel(struct rpmsg_channel *rpdev, u32 src, u32 dst,
                                                    void *data, int len);
```

sends a message across to the remote processor, using the src and dst addresses provided by the user.

The caller should specify the channel, the data it wants to send, its length (in bytes), and explicit source and destination addresses. The message will then be sent to the remote processor to which the channel belongs, but the channel's src and dst addresses will be ignored (and the user-provided addresses will be used instead).

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio's used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysend(struct rpmsg_channel *rpdev, void *data, int len);
```

sends a message across to the remote processor on a given channel. The caller should specify the channel, the data it wants to send, and its length (in bytes). The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel's src and dst addresses.

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysendto(struct rpmsg_channel *rpdev, void *data, int len, u32 dst)
```

sends a message across to the remote processor on a given channel, to a destination address provided by the user.

The user should specify the channel, the data it wants to send, its length (in bytes), and an explicit destination address.

The message will then be sent to the remote processor to which the channel belongs, using the channel's src address, and the user-provided dst address (thus the channel's dst address will be ignored).

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysend_offchannel(struct rpmsg_channel *rpdev, u32 src, u32 dst,
                                                    void *data, int len);
```

sends a message across to the remote processor, using source and destination addresses provided by the user.

The user should specify the channel, the data it wants to send, its length (in bytes), and explicit source and destination addresses. The message will then be sent to the remote processor to which the channel belongs, but the channel's src and dst addresses will be ignored (and the user-provided addresses will be used instead).

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
struct rpmsg_endpoint *rpmsg_create_ept(struct rpmsg_device *rpdev,
                                        rpmsg_rx_cb_t cb, void *priv,
                                        struct rpmsg_channel_info chinfo);
```

every rpmsg address in the system is bound to an rx callback (so when inbound messages arrive, they are dispatched by the rpmsg bus using the appropriate callback handler) by means of an rpmsg_endpoint struct.

This function allows drivers to create such an endpoint, and by that, bind a callback, and possibly some private data too, to an rpmsg address (either one that is known in advance, or one that will be dynamically assigned for them).

Simple rpmsg drivers need not call rpmsg_create_ept, because an endpoint is already created for them when they are probed by the rpmsg bus (using the rx callback they provide when they registered to the rpmsg bus).

So things should just work for simple drivers: they already have an endpoint, their rx callback is bound to their rpmsg address, and when relevant inbound messages arrive (i.e. messages which their dst address equals to the src address of their rpmsg channel), the driver's handler is invoked to process it.

That said, more complicated drivers might do need to allocate additional rpmsg addresses, and bind them to different rx callbacks. To accomplish that, those drivers need to call this function. Drivers should provide their channel (so the new endpoint would bind to the same remote processor their channel belongs to), an rx callback function, an optional private data (which is provided back when the rx callback is invoked), and an address they want to bind with the callback. If addr is RPMSG_ADDR_ANY, then rpmsg_create_ept will dynamically assign them an available rpmsg address (drivers should have a very good reason why not to always use RPMSG_ADDR_ANY here).

Returns a pointer to the endpoint on success, or NULL on error.

```
void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);
```

destroys an existing rpmsg endpoint. user should provide a pointer to an rpmsg endpoint that was previously created with rpmsg_create_ept().

```
int register_rpmsg_driver(struct rpmsg_driver *rpdrv);
```

registers an rpmsg driver with the rpmsg bus. user should provide a pointer to an rpmsg_driver struct, which contains the driver's ->probe() and ->remove() functions, an rx callback, and an id_table specifying the names of the channels this driver is interested to be probed with.

```
void unregister_rpmsg_driver(struct rpmsg_driver *rpdrv);
```

unregisters an rpmsg driver from the rpmsg bus. user should provide a pointer to a previously-registered rpmsg_driver struct. Returns 0 on success, and an appropriate error value on failure.

Typical usage

The following is a simple rpmsg driver, that sends an "hello!" message on probe(), and whenever it receives an incoming message, it dumps its content to the console.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rpmsg.h>

static void rpmsg_sample_cb(struct rpmsg_channel *rpdev, void *data, int len,
                                                void *priv, u32 src)
{
      print_hex_dump(KERN_INFO, "incoming message:", DUMP_PREFIX_NONE,
                                                16, 1, data, len, true);
}

static int rpmsg_sample_probe(struct rpmsg_channel *rpdev)
{
      int err;

      dev_info(&rpdev->dev, "chnl: 0x%x -> 0x%x\n", rpdev->src, rpdev->dst);

      /* send a message on our channel */
      err = rpmsg_send(rpdev, "hello!", 6);
      if (err) {
              pr_err("rpmsg_send failed: %d\n", err);
              return err;
      }

      return 0;
}

static void rpmsg_sample_remove(struct rpmsg_channel *rpdev)
{
      dev_info(&rpdev->dev, "rpmsg sample client driver is removed\n");
}

static struct rpmsg_device_id rpmsg_driver_sample_id_table[] = {
      { .name = "rpmsg-client-sample" },
      { },
};
MODULE_DEVICE_TABLE(rpmsg, rpmsg_driver_sample_id_table);

static struct rpmsg_driver rpmsg_sample_client = {
      .drv.name       = KBUILD_MODNAME,
      .id_table       = rpmsg_driver_sample_id_table,
      .probe          = rpmsg_sample_probe,
      .callback       = rpmsg_sample_cb,
      .remove         = rpmsg_sample_remove,
};
module_rpmsg_driver(rpmsg_sample_client);
```

> **Note**
>
> a similar sample which can be built and loaded can be found in samples/rpmsg/.

Allocations of rpmsg channels

At this point we only support dynamic allocations of rpmsg channels.

This is possible only with remote processors that have the VIRTIO_RPMSG_F_NS virtio device feature set. This feature bit means that the remote processor supports dynamic name service announcement messages.

When this feature is enabled, creation of rpmsg devices (i.e. channels) is completely dynamic: the remote processor announces the existence of a remote rpmsg service by sending a name service message (which contains the name and rpmsg addr of the remote service, see struct rpmsg_ns_msg).

This message is then handled by the rpmsg bus, which in turn dynamically creates and registers an rpmsg channel (which represents the remote service). If/when a relevant rpmsg driver is registered, it will be immediately probed by the bus, and can then start sending messages to the remote service.

The plan is also to add static creation of rpmsg channels via the virtio config space, but it's not implemented yet.