Remote Processor Framework

Introduction

Modern SoCs typically have 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. In a typical configuration, 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.

The remoteproc framework allows different platforms/architectures to control (power on, load firmware, power off) those remote processors while abstracting the hardware differences, so the entire driver doesn't need to be duplicated. In addition, this framework also adds rpmsg virtio devices for remote processors that supports this kind of communication. This way, platform-specific remoteproc drivers only need to provide a few low-level handlers, and then all rpmsg drivers will then just work (for more information about the virtio-based rpmsg bus and its drivers, please read Documentation/staging/rpmsg.rst). Registration of other types of virtio devices is now also possible. Firmwares just need to publish what kind of virtio devices do they support, and then remoteproc will add those devices. This makes it possible to reuse the existing virtio drivers with remote processor backends at a minimal development cost.

User API

```
int rproc_boot(struct rproc *rproc)
```

Boot a remote processor (i.e. load its firmware, power it on, ...).

If the remote processor is already powered on, this function immediately returns (successfully).

Returns 0 on success, and an appropriate error value otherwise. Note: to use this function you should already have a valid rproc handle. There are several ways to achieve that cleanly (devres, pdata, the way remoteproc_rpmsg.c does this, or, if this becomes prevalent, we might also consider using dev_archdata for this).

```
int rproc_shutdown(struct rproc *rproc)
```

Power off a remote processor (previously booted with rproc_boot()). In case @rproc is still being used by an additional user(s), then this function will just decrement the power refcount and exit, without really powering off the device.

Returns 0 on success, and an appropriate error value otherwise. Every call to rproc_boot() must (eventually) be accompanied by a call to rproc_shutdown(). Calling rproc_shutdown() redundantly is a bug.

Note

we're not decrementing the rproc's refcount, only the power refcount. which means that the @rproc handle stays valid even after rproc_shutdown() returns, and users can still use it with a subsequent rproc_boot(), if needed.

```
struct rproc *rproc_get_by_phandle(phandle phandle)
```

Find an rproc handle using a device tree phandle. Returns the rproc handle on success, and NULL on failure. This function increments the remote processor's refcount, so always use rproc_put() to decrement it back once rproc isn't needed anymore.

Typical usage

```
#include <linux/remoteproc.h>

/* in case we were given a valid 'rproc' handle */
int dummy_rproc_example(struct rproc *my_rproc)
{
	int ret;

	/* let's power on and boot our remote processor */
	ret = rproc_boot(my_rproc);
	if (ret) {
		/*
		 * something went wrong. handle it and leave.
		 */
	}

	/*
	 * our remote processor is now powered on... give it some work
	 */
```

```
        /* let's shut it down now */
        rproc_shutdown(my_rproc);
}
```

API for implementors

```
struct rproc *rproc_alloc(struct device *dev, const char *name,
                              const struct rproc_ops *ops,
                              const char *firmware, int len)
```

Allocate a new remote processor handle, but don't register it yet. Required parameters are the underlying device, the name of this remote processor, platform-specific ops handlers, the name of the firmware to boot this rproc with, and the length of private data needed by the allocating rproc driver (in bytes).

This function should be used by rproc implementations during initialization of the remote processor.

After creating an rproc handle using this function, and when ready, implementations should then call rproc_add() to complete the registration of the remote processor.

On success, the new rproc is returned, and on failure, NULL.

> **Note**
>
> **never** directly deallocate @rproc, even if it was not registered yet. Instead, when you need to unroll rproc_alloc(), use rproc_free().

```
void rproc_free(struct rproc *rproc)
```

Free an rproc handle that was allocated by rproc_alloc.

This function essentially unrolls rproc_alloc(), by decrementing the rproc's refcount. It doesn't directly free rproc; that would happen only if there are no other references to rproc and its refcount now dropped to zero.

```
int rproc_add(struct rproc *rproc)
```

Register @rproc with the remoteproc framework, after it has been allocated with rproc_alloc().

This is called by the platform-specific rproc implementation, whenever a new remote processor device is probed.

Returns 0 on success and an appropriate error code otherwise. Note: this function initiates an asynchronous firmware loading context, which will look for virtio devices supported by the rproc's firmware.

If found, those virtio devices will be created and added, so as a result of registering this remote processor, additional virtio drivers might get probed.

```
int rproc_del(struct rproc *rproc)
```

Unroll rproc_add().

This function should be called when the platform specific rproc implementation decides to remove the rproc device. it should _only_ be called if a previous invocation of rproc_add() has completed successfully.

After rproc_del() returns, @rproc is still valid, and its last refcount should be decremented by calling rproc_free().

Returns 0 on success and -EINVAL if @rproc isn't valid.

```
void rproc_report_crash(struct rproc *rproc, enum rproc_crash_type type)
```

Report a crash in a remoteproc

This function must be called every time a crash is detected by the platform specific rproc implementation. This should not be called from a non-remoteproc driver. This function can be called from atomic/interrupt context.

Implementation callbacks

These callbacks should be provided by platform-specific remoteproc drivers:

```
/**
 * struct rproc_ops - platform-specific device handlers
 * @start:    power on the device and boot it
 * @stop:     power off the device
 * @kick:     kick a virtqueue (virtqueue id given as a parameter)
 */
struct rproc_ops {
        int (*start)(struct rproc *rproc);
        int (*stop)(struct rproc *rproc);
        void (*kick)(struct rproc *rproc, int vqid);
};
```

Every remoteproc implementation should at least provide the ->start and ->stop handlers. If rpmsg/virtio functionality is also desired, then the ->kick handler should be provided as well.

The ->start() handler takes an rproc handle and should then power on the device and boot it (use rproc->priv to access platform-specific private data). The boot address, in case needed, can be found in rproc->bootaddr (remoteproc core puts there the ELF entry point). On success, 0 should be returned, and on failure, an appropriate error code.

The ->stop() handler takes an rproc handle and powers the device down. On success, 0 is returned, and on failure, an appropriate error code.

The ->kick() handler takes an rproc handle, and an index of a virtqueue where new message was placed in. Implementations should interrupt the remote processor and let it know it has pending messages. Notifying remote processors the exact virtqueue index to look in is optional: it is easy (and not too expensive) to go through the existing virtqueues and look for new buffers in the used rings.

Binary Firmware Structure

At this point remoteproc supports ELF32 and ELF64 firmware binaries. However, it is quite expected that other platforms/devices which we'd want to support with this framework will be based on different binary formats.

When those use cases show up, we will have to decouple the binary format from the framework core, so we can support several binary formats without duplicating common code.

When the firmware is parsed, its various segments are loaded to memory according to the specified device address (might be a physical address if the remote processor is accessing memory directly).

In addition to the standard ELF segments, most remote processors would also include a special section which we call "the resource table".

The resource table contains system resources that the remote processor requires before it should be powered on, such as allocation of physically contiguous memory, or iommu mapping of certain on-chip peripherals. Remotecore will only power up the device after all the resource table's requirement are met.

In addition to system resources, the resource table may also contain resource entries that publish the existence of supported features or configurations by the remote processor, such as trace buffers and supported virtio devices (and their configurations).

The resource table begins with this header:

```
/**
 * struct resource_table - firmware resource table header
 * @ver: version number
 * @num: number of resource entries
 * @reserved: reserved (must be zero)
 * @offset: array of offsets pointing at the various resource entries
 *
 * The header of the resource table, as expressed by this structure,
 * contains a version number (should we need to change this format in the
 * future), the number of available resource entries, and their offsets
 * in the table.
 */
struct resource_table {
      u32 ver;
      u32 num;
      u32 reserved[2];
      u32 offset[0];
} __packed;
```

Immediately following this header are the resource entries themselves, each of which begins with the following resource entry header:

```
/**
 * struct fw_rsc_hdr - firmware resource entry header
 * @type: resource type
 * @data: resource data
 *
 * Every resource entry begins with a 'struct fw_rsc_hdr' header providing
 * its @type. The content of the entry itself will immediately follow
 * this header, and it should be parsed according to the resource type.
 */
struct fw_rsc_hdr {
      u32 type;
      u8 data[0];
} __packed;
```

Some resources entries are mere announcements, where the host is informed of specific remoteproc configuration. Other entries require the host to do something (e.g. allocate a system resource). Sometimes a negotiation is expected, where the firmware requests a resource, and once allocated, the host should provide back its details (e.g. address of an allocated memory region).

Here are the various resource types that are currently supported:

```
/**
```

```
 * enum fw_resource_type - types of resource entries
 *
 * @RSC_CARVEOUT:   request for allocation of a physically contiguous
 *                 memory region.
 * @RSC_DEVMEM:     request to iommu_map a memory-based peripheral.
 * @RSC_TRACE:            announces the availability of a trace buffer into which
 *                 the remote processor will be writing logs.
 * @RSC_VDEV:       declare support for a virtio device, and serve as its
 *                 virtio header.
 * @RSC_LAST:       just keep this one at the end
 * @RSC_VENDOR_START: start of the vendor specific resource types range
 * @RSC_VENDOR_END:   end of the vendor specific resource types range
 *
 * Please note that these values are used as indices to the rproc_handle_rsc
 * lookup table, so please keep them sane. Moreover, @RSC_LAST is used to
 * check the validity of an index before the lookup table is accessed, so
 * please update it as needed.
 */
enum fw_resource_type {
      RSC_CARVEOUT            = 0,
      RSC_DEVMEM              = 1,
      RSC_TRACE               = 2,
      RSC_VDEV                = 3,
      RSC_LAST                = 4,
      RSC_VENDOR_START        = 128,
      RSC_VENDOR_END          = 512,
};
```

For more details regarding a specific resource type, please see its dedicated structure in include/linux/remoteproc.h.

We also expect that platform-specific resource entries will show up at some point. When that happens, we could easily add a new RSC_PLATFORM type, and hand those resources to the platform-specific rproc driver to handle.

Virtio and remoteproc

The firmware should provide remoteproc information about virtio devices that it supports, and their configurations: a RSC_VDEV resource entry should specify the virtio device id (as in virtio_ids.h), virtio features, virtio config space, vrings information, etc.

When a new remote processor is registered, the remoteproc framework will look for its resource table and will register the virtio devices it supports. A firmware may support any number of virtio devices, and of any type (a single remote processor can also easily support several rpmsg virtio devices this way, if desired).

Of course, RSC_VDEV resource entries are only good enough for static allocation of virtio devices. Dynamic allocations will also be made possible using the rpmsg bus (similar to how we already do dynamic allocations of rpmsg channels; read more about it in rpmsg.txt).