Linux remoteproc framework overview

Stable: 02.10.2019 - 15:51 / Revision: 02.10.2019 - 15:46

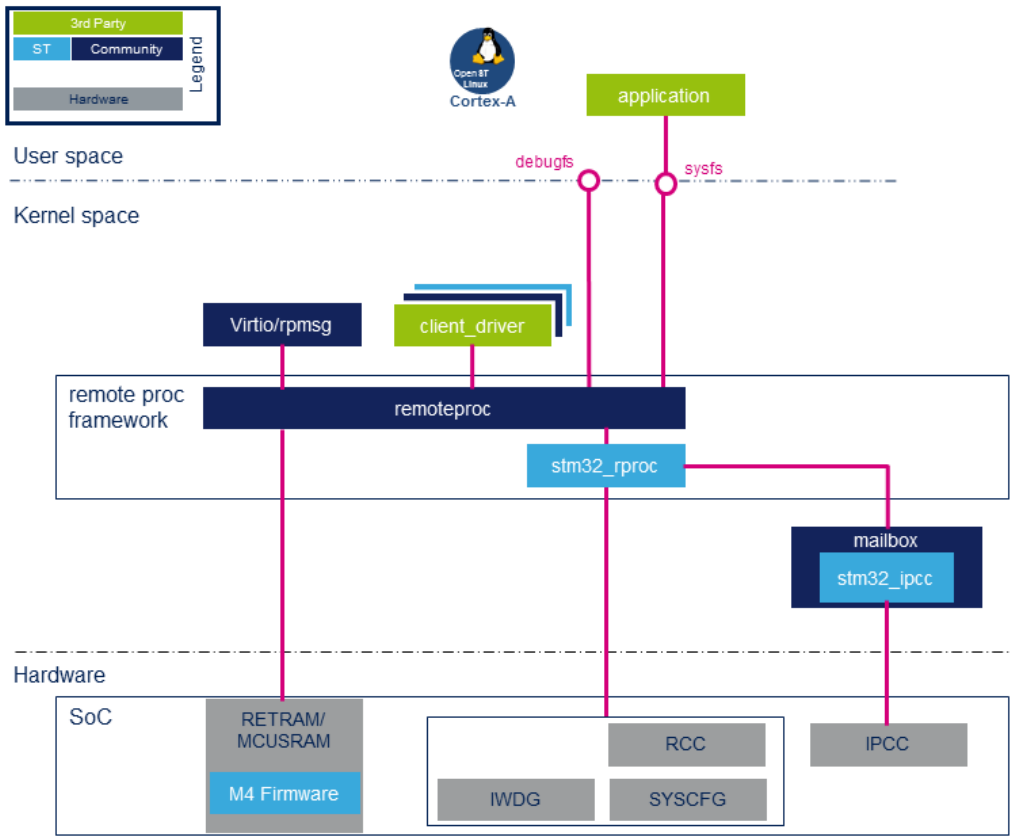
This article gives information about the Linux® remoteproc framework.

Contents

1 Framework purpose

The remote processor (RPROC) framework allows the different platforms/architectures to control (power on, load firmware, power off) remote processors while abstracting the hardware differences. In addition it offers services to monitor and debug the remote coprocessor.

2 System overview



2.1 Component description

remoteproc: this is the remote processor framework generic part. Its role is to:

- Load the ELF firmware in the remote processor memory.
- Parse the firmware resource table to set associated resources (such as IPC, memory carveout and traces).
- Control the remote processor execution (start, stop...).
- Provide a service to monitor and debug the remote firmware.

stm32_rproc: this is the remote processor platform driver. Its role is to:

- Register the vendor specific functions (callback) to the RPROC framework.
- Handle the platform resources associated to the remote processor (such as registers, watchdogs, reset, clock and memories).
- Forward notifications (kicks) to the remote processor through the mailbox framework.

2.2 API description

The API usage and remote processor binary firmware structure (resource table, ...) are described in the Linux kernel remoteproc documentation [1].

3 Configuration

3.1 Kernel configuration

Activate the remoteproc driver and framework in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel.

```
Device drivers  --->
   Remoteproc drivers  --->
     <*> Support for Remote Processor subsystem
     <*> STM32 remoteproc support
```

3.2 Device tree configuration

The remote processor device node must be declared and enabled in the Linux kernel device tree. Here is an extract of the STM32MP1 evaluation board device tree:

```
 /* Memory region declaration, containing vring and rpmsg buffers */
 reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        /* RETRAM memory region reserved for firmware code and data */
        retram: retram@0x38000000 {
                compatible = "shared-dma-pool";
                reg = <0x38000000 0x10000>;
                no-map;
        };
        /* MCUSRAM memory region reserved for firmware code and data */
        mcusram: mcusram@0x10000000 {
                compatible = "shared-dma-pool";
                reg = <0x10000000 0x40000>;
                no-map;
        };
        /* MCUSRAM aliased memory region reserved for firmware code and data */
        mcusram2: mcusram2@0x30000000 {
                compatible = "shared-dma-pool";
                reg = <0x30000000 0x40000>;
                no-map;
        };

        /* memory region reserved for virtio ring buffer descriptor 0*/
        vdev0vring0: vdev0vring0@10040000 {
                compatible = "shared-dma-pool";
                reg = <0x10040000 0x2000>;
                no-map;
        };
        /* memory region reserved for virtio ring buffer descriptor 1*/
        vdev0vring1: vdev0vring1@10042000 {
                compatible = "shared-dma-pool";
                reg = <0x10042000 0x2000>;
```

```
                no-map;
        };
        /* memory region reserved for virtio buffers associated to the vrings*/
        vdev0buffer: vdev0buffer@10044000 {
                compatible = "shared-dma-pool";
                reg = <0x10044000 0x4000>;
                no-map;
        };
};

 /* stm32 M4 remoteproc device */
 m4_rproc: m4@0 {
        compatible = "st,stm32mp1-rproc";
        ranges = <0x00000000 0x38000000 0x10000>,
                 <0x30000000 0x30000000 0x60000>,
                 <0x10000000 0x10000000 0x60000>;
        memory-region = <&retram>, <&mcusram>, <&mcusram2>, <&vdev0vring0>,
                        <&vdev0vring1>, <&vdev0buffer>;
        mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
        mbox-names = "vq0", "vq1", "shutdown";
        resets = <&rcc_rst MCU_R>;
        reset-names = "mcu_rst";
        st,hold_boot = <&rcc 0x10C 0x1>;
        st,smc_boot;

        status = "okay";
};
```

'memory-region' properties define the RETRAM and MCUSRAM base addresses and sizes in RETRAM and MCUSRAM, from the Arm® Cortex®-A point of view. The firmware memory mapping must be set according to these values in the linker script.

For additional details, please refer to STM32MP15 Memory mapping.

4 How to use the framework

4.1 Remote processor boot

There are three possibilities to load and start the remote processor firmware:

- Start the firmware through the SysFS interface.
- Automatically start the firmware on remoteproc driver probing (not recommended by STMicroelectronics).
- Early boot the firmware during boot time (before Linux boot).

4.1.1 Remote processor boot through sysfs

- The firmware components are stored in the file system, by default in the **/lib/firmware/** folder. Optionally another location can be set. In this case the remoteproc framework parses this new path in priority.

Below the command for adding a new path for firmware parsing:

```
Board $> echo -n <firmware_path> > /sys/module/firmware_class/parameters/path
```

This path is common for all firmwares loaded by Linux (Bluetooth, Wifi...)

- If the firmware elf filename differs from the default one (rproc-%s-fw), set the name with the following command: (replace **X** with remoteproc instance number: 0 by default)

```
Board $> echo -n <firmware_name.elf> > /sys/class/remoteproc/remoteprocX/firmware
```

- To start the firmware, use the following command:

```
Board $> echo start >/sys/class/remoteproc/remoteprocX/state
```

Based on the above commands, a userland service can be implemented to automatically load the firmware during the userland initialization phase.

4.1.2 Remote processor 'auto' boot (not recommended by STMicroelectronics)

The remote processor can be automatically booted during platform boot. To do this, the following conditions must be fulfilled:

- The firmware must be present in **/lib/firmware** before the remoteproc driver is probed.
- The filesystem on Linux (Cortex-A) must be available before the remoteproc driver is probed. However, in normal conditions, the remoteproc driver is probed before the filesystem is mounted, and the firmware is consequently not available during the Linux driver probing phase. Possible solutions could be:
  - to use an initramfs[2]
  - or to put the remoteproc driver in the module.
- The firmware must be named **rproc-%s-fw**, where **%s** corresponds to the name of the remoteproc node in the device tree. For example, for **rproc-m4-fw.elf**, the remoteproc device tree must be defined as follows:

```
m4 {
        compatible = "st,stm32mp1-rproc";
[...]
        status = "okay";
};
```

- The "auto_boot" property has to be defined in the remoteproc node device tree:

```
m4 {
        compatible = "st,stm32mp1-rproc";
[...]
        auto_boot;
        status = "okay";
};
```

4.1.3 Remote processor 'early' boot

The coprocessor can be started by the second stage bootloader (eg U-Boot). This mode allows to start the coprocessor firmware before the Linux one. For instance, it can be used to execute first actions for projects that have hard constraints on boot time. On Linux boot, the remoteproc framework attaches itself to the firmware by parsing the resource table, based on the information added by the bootloader in the Linux device tree (resource table address and size). Refer to How to start the coprocessor from the bootloader for details on this mode.

4.2 Remote processor stop

It is possible to stop the remote processor firmware through the SysFS interface. On stop request, the stm32_rproc driver:

- informs the remote firmware relying on the "shutdown" channel of the the IPCC mailbox. This mechanism allows the remote processor firmware to shut down properly.
- resets the coprocessor, on "shutdown" message acknowledgement or after a timeout of 500 ms.

The use of the IPCC "shutdown" channel is optional. If the mailbox channel is not declared in the device trree, the remote processor is immediately reset, without informing firmware of the remote processor.

To stop the firmware, use the following command:

```
Board $> echo stop >/sys/class/remoteproc/remoteprocX/state
```

5 How to trace and debug the framework

5.1 How to monitor

- The remoteproc firmware state can be monitored using following command:

```
Board $> cat /sys/class/remoteproc/remoteprocX/state
```

5.2 How to trace

- remoteproc framework and driver debug traces can be added in the kernel logs thanks to the dynamic debug mechanism:

```
Board $> echo -n 'file stm32_rproc.c +p' > /sys/kernel/debug/dynamic_debug/control
Board $> echo -n 'file remoteproc*.c +p' > /sys/kernel/debug/dynamic_debug/control
```

- A log buffer can de defined in the remoteproc firmware and declared in the resource table. If the feature is activated on the remote firmware, log traces can be dumped from the trace buffer using the following command:

```
Board $>cat /sys/kernel/debug/remoteproc/remoteprocX/trace0
```

6 References

1. ↑ Linux kernel remoteproc documentation
2. ↑ ramfs-rootfs-initramfs Linux documentation