Coprocessor management troubleshooting grid

Contents

Stable: 07.12.2020 - 14:11 / Revision: 07.12.2020 - 14:09

Stable: 07.12.2020 - 14:11 / Revision: 07.12.2020 - 14:00

A quality version of this page, approved on *7 December 2020*, was based off this revision.

Some typical issues related to the management of a coprocessor are listed below. Solutions or debugging methods are proposed for these issues.

If your issue is not listed, try looking in the articles in the Coprocessor management Linux, Coprocessor management STM32Cube or troubleshooting grids categories.

1 Coprocessor firmware loading and control

Symptom	Resolution

The coprocessor traces are not available on Linux® side
 	
`Board $> cat /sys/kernel/debug /remoteproc/remoteproc0/trace0`
`No such file or directory`
	This may happen for two reasons:
	• The firmware does not include any resource table or the resource table does not define any trace.
	Update the firmware resource table and rebuild the firmware
	• The ".resource_table" section is empty or not defined in the elf file.
	Use the following command to verify it in the generated elf file:
	
	`PC $> readelf -l <elf file>`
	`...`
	`02 .data . resource_table .bss . _user_heap_stack`
	`....`

When starting the coprocessor from the bootloader (u-boot):
 	
`"unsupported fw ver: 0`
`Remote Processor 0 resource table Not found : 0x00000000-0x0 "`
	The firmware does not include any resource table. This is only a warning that does not prevent the firmware from starting properly. "rproc load_rsc" step can be bypassed.

2 Inter processor communication

Symptom	Resolution
Frozen firmware as consequence of a deadlock in OpenAMP during IP communication with the the main processor.	This Issue probably comes from rpmsg_virtio_rx_callback or rpmsg_virtio_send_offchannel_raw (rpmsg_virtio.c) functions that are called in interrupt context. These functions use a mutex lock in rpmsg_device struct when accessing the index of the virtio queue index. Rework your code so that these functions are not called in interrupt context.
Linux kernel trace: `stm32-ipcc 4c001000.mailbox: Try increasing MBOX_TX_QUEUE_LEN`	On each IPCC interrupt, the coprocessor treats all the buffered RPMsgs (one IPCC event for several RPMsgs). On Linux side, one IPCC signal is programmed for each RPM sg sent. This can result is an overflow warning on Linux since too many IPCC events are queued. No RPMsgs are dropped but this message could be interpreted as the coprocessor reaches its capacity to treat the received messages in time. Consider reworking the code so that the coprocessor processes messages more efficiently or decreasing the rate of messages sent by Linux.
Linux kernel trace (example with rpmsg_tty driver): `rpmsg_tty virtio0.rpmsg-tty-channel. -1.0: timeout waiting for a tx buffer`	This message means that there is no more TX buffer available to transmit messages to the remote processor. This may happen for two reasons: • The firmware implementation is incomplete and the coprocessor does not process any IP CC interrupt (eg interrupt disabled or interrupt handler not defined). Fix the firmware code: refer to IPCC_internal_peripheral for details on the peripheral. • The coprocessor is busy, frozen or crashed. This can be confirmed by performing a debug tool analysis.
Linux kernel trace:	There is no more space in the TTY temporary buffer on reception. The root cause is probably that the Linux application did not

Symptom	Resolution
`rpmsg_tty virtio0.rpmsg-tty-channel. -1.0: No memory for tty_prepare_flip_string`	read the tty device on time to treat the incoming TTY stream. Consider reworking the Linux application so that it takes less time to process messages or decreasing the message exchange rate.
Linux kernel trace: `remoteproc remoteproc0: stm32_rproc_kick: failed (<mbx>, <error value>)`	The Linux remoteproc driver cannot use the IPCC mailbox. This may happen for two reasons: • The Linux kernel is built without the support of the stm32 IPCC mailbox: to enable it, refer to IPCC configuration. • The mailboxes are not or incorrectly defined in the Linux kernel DeviceTree: fix it as described in mailbox DeviceTree.
stm32Cube error: VIRT_UART_Transmit() returns an error until a first message is received from the Cortex-A7.	Before the Cortex-M4 can send any message, the RPMsg protocol requires that the Linux application has sent a first message (to provide its address to the Cortex-M4). The Linux and/or the Cortex-M4 applications shall be reworked to follow this constraint.

Linux® is a registered trademark of Linus Torvalds.

Transmit

Inter-Processor Communication Controller

Remote Processor Messaging

TeleTYpewriter

Universal Asynchronous Receiver/Transmitter

Cortex®

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 08:08

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the Linux®**coprocessor management** software frameworks such as: mailbox, RPMsg, and so on.

It is recommended to first read the Coprocessor management overview article.

Linux® is a registered trademark of Linus Torvalds.

Remote Processor Messaging

Pages in category "Coprocessor management Linux"

The following 12 pages are in this category, out of 12 total.

- Coprocessor management overview
- Coprocessor management troubleshooting grid
- Coprocessor resource table
- Exchanging buffers with the coprocessor
- How to assign an internal peripheral to a runtime context
- How to configure system resources
- How to exchange large data buffers with the coprocessor - principle
- How to protect the coprocessor firmware
- Linux Mailbox framework overview
- Linux remoteproc framework overview
- Linux RPMsg framework overview
- Resource manager for coprocessing

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 08:10

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the STM32Cube MPU **coprocessor management** software components.

It is recommended to first read the Coprocessor management overview article.

Microprocessor Unit

Pages in category "Coprocessor management STM32Cube"

The following 12 pages are in this category, out of 12 total.

- Coprocessor management overview
- Coprocessor management troubleshooting grid
- Coprocessor power management
- Coprocessor resource table
- Exchanging buffers with the coprocessor
- How to assign an internal peripheral to a runtime context
- How to configure system resources
- How to exchange large data buffers with the coprocessor - principle
- How to protect the coprocessor firmware
- How to retrieve Cortex-M4 logs after crash
- How to start the coprocessor from the bootloader
- Resource manager for coprocessing

Stable: 17.06.2020 - 15:27 / Revision: 16.01.2020 - 13:51

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to a troubleshooting grid.

Pages in category "Troubleshooting grids"

The following 9 pages are in this category, out of 9 total.

- ALSA troubleshooting grid
- Audio troubleshooting grid
- Coprocessor management troubleshooting grid
- DRM KMS troubleshooting grid
- GPU troubleshooting grid
- GStreamer troubleshooting grid
- Networking troubleshooting grid
- Visual troubleshooting grid
- Wayland Weston troubleshooting grid

Stable: 24.01.2020 - 09:04 / Revision: 24.01.2020 - 09:03

A quality version of this page, approved on *24 January 2020*, was based off this revision.

Contents

1 Role of the resource table

The resource table is a global variable declared as a structure in the coprocessor firmware. This table contains resources that the remote processor requires before being powered on, such as the allocation of a physically contiguous memory. In addition, 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/or supported Virtio devices used for the IPC.

This table must be defined in a specific data section of the coprocessor firmware, parsed by the RemoteProc Linux® framework during the firmware load phase to:

- Allocate memories defined in the resource table carveout section (not used in the ST Arm®Cortex®-M4 firmware).
- Load the RPMsg and Virtio frameworks to support messaging services.
- Provide a user sysfs interface to access coprocessor traces for debug.

2 How to define the resource table

The resource table must be part of the firmware's ELF image, in order to be accessible by the Linux RemoteProc framework. This resource table can be a default table or customized depending on the enabled features.

In the STM32MCU cube firmware package, the resource table is defined in rsc_table.c

2.1 Default table

For the Cortex-M firmware that does not require interaction with the Linux OS, a default structure must be declared in the Cortex -M firmware

```
struct remote_resource_table __resource __attribute__((used)) rproc_resource = {
        .version = 1,
        .num = 0,
        .reserved = {0, 0},
        .offset = { 0 },
};
```

2.2 How to add trace for the log buffer

This feature allows to dump Cortex-M firmware traces on the linux side. For this a **system_log_buf** buffer defined in log.c file and declared in the resource table allows Linux to dump the associated memory area (under "__LOG_TRACE_IO_" preprocessor definition).

```
 1 const struct shared_resource_table __resource __attribute__((used)) resource_table = {
 2         .version = 1,
 3 #if defined (__LOG_TRACE_IO_)
 4         .num = 2,
 5 #else
 6         .num = 1,
 7 #endif
 8         .reserved = {0, 0},
 9         .offset = {
10                 offsetof(struct shared_resource_table, vdev),
11 #if defined (__LOG_TRACE_IO_)
12                 offsetof(struct shared_resource_table, cm_trace),
13 #endif
14         },
15         .......
16 #if defined (__LOG_TRACE_IO_)
17         .cm_trace = {
18                 RSC_TRACE,
19                 (uint32_t)system_log_buf, SYSTEM_TRACE_BUF_SZ, 0, "cm4_log",
20         },
21 #endif
22  };
```

These logs can be retrieved after a firmware crash: refer to How to retrieve Cortex-M4 logs after crash for more information.

2.3 How to add RPMsg inter-processor communication

The messaging service is enabled by declaring:

- The rpmsg Virtio device for control,
- The rpmsg Virtio ring buffers for message management.

These structures are used by the Linux RemoteProc framework. The Remoteproc framework is in charge of allocating buffers associated to Vring (for both direction) in the shared memory and of providing information in the rpmsg_vring structures.

```
 1 #define NUM_VRINGS                  0x02  /* number of Vring used , must be fixed to 2
(one for TX one for RX) */
 2 #define VRING_ALIGN                 0x1000 /* must be fixed to 0x1000 (Linux
constraint) */
 3 #define VRING_TX                    -1  /* allocated by master processor */
 4 #define VRING_RX                    -1  /* allocated by master processor */
 5 #define VRING_SIZE                  8  /* number of 512 bytes buffers associated to a
Vring: can be customized */
 6
 7 struct remote_resource_table __resource __attribute__((used)) rproc_resource = {
 8          .version = 1,
 9          .num = 1,  /* rely to number of offsetof structures declared in .offset */
10          .reserved = {0, 0},
11          .offset = {
12                  offsetof(struct remote_resource_table, rpmsg_vdev),
13          },
14          /* Virtio device entry */
15         .rpmsg_vdev= {
16                 RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0, 0,
17                 NUM_VRINGS, {0, 0},
18         },
19
20         /* Vring rsc entry - part of vdev rsc entry */
21         .rpmsg_vring0 = {VRING_TX, VRING_ALIGN, VRING_SIZE, 1, 0},
22         .rpmsg_vring1 = {VRING_RX, VRING_ALIGN, VRING_SIZE, 2, 0},
23  };
```

Inter-Processor Communication

Remote Processor Messaging

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Executable and linkable file

Operating System

input/output

Transmit

Receive

Stable: 22.01.2020 - 16:10 / Revision: 22.01.2020 - 10:51

A quality version of this page, approved on *22 January 2020*, was based off this revision.

Contents

1 Article purpose

The inter-processor communication controller (IPCC) is used to exchange data between two processors. It provides a non blocking signaling mechanism to post and retrieve information in an atomic way. Note that shared memory buffers are allocated in the MCU SRAM, which is not part of the IPCC block.

2 Peripheral overview

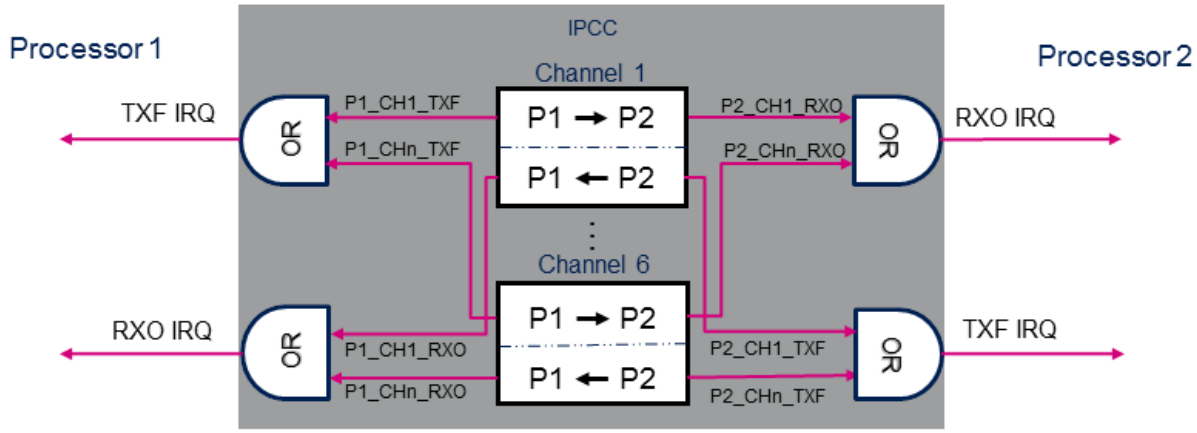
The **IPCC** peripheral provides a hardware support to manage inter-processor communication between two processor instances. Each processor owns specific register bank and interrupts.

The IPCC provides the signaling for **six bidirectional** channels.
Each channel is divided into two subchannels that offer a unidirectional signaling from the "sender" processor to the "receiver" processor:

- P1_TO_P2 subchannel
- P2_TO_P1 subchannel

A subchannel consists in:

- One flag that toggles between occupied and free: the flag is set to occupied by the "sender" processor and cleared by the "receiver" processor.
- Two associated interrupts (shared with the other channels):
  - RXO: RX channel occupied, connected to the "receiver" processor.
  - TXF: TX channel free, connected to the "sender" processor.
- Two associated interrupt masks multiplexing channel IRQs.



The IPCC supports the following channel operating modes:

- **Simplex communication mode**:
  - Only one subchannel is used.
  - Unidirectional messages: once the "sender" processor has posted the communication data in the memory, it sets the channel status flag to occupied. The "receiver" processor clears the flag when the message is treated.
- **Half-duplex communication mode**:
  - Only one subchannel is used.
  - Bidirectional messages: once the "sender" processor has posted the communication data in the memory, it sets the channel status flag to occupied. The "receiver" processor clears the flag when the message is treated and the response is available in shared memory.

- **Full-duplex communication mode**:
  - The subchannels are used in Asynchronous mode.
  - Any processor can post asynchronously a message by setting the subchannel status flag to occupied. The "receiver" processor clears the flag when the message is treated. This mode can be considered as a combination of two simplex modes on a given channel.

2.1 Features

Refer to STM32MP15 reference manuals for the complete features list, and to the software components, introduced below, to see which features are implemented.

2.2 Security support

The IPCC is a **non-secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

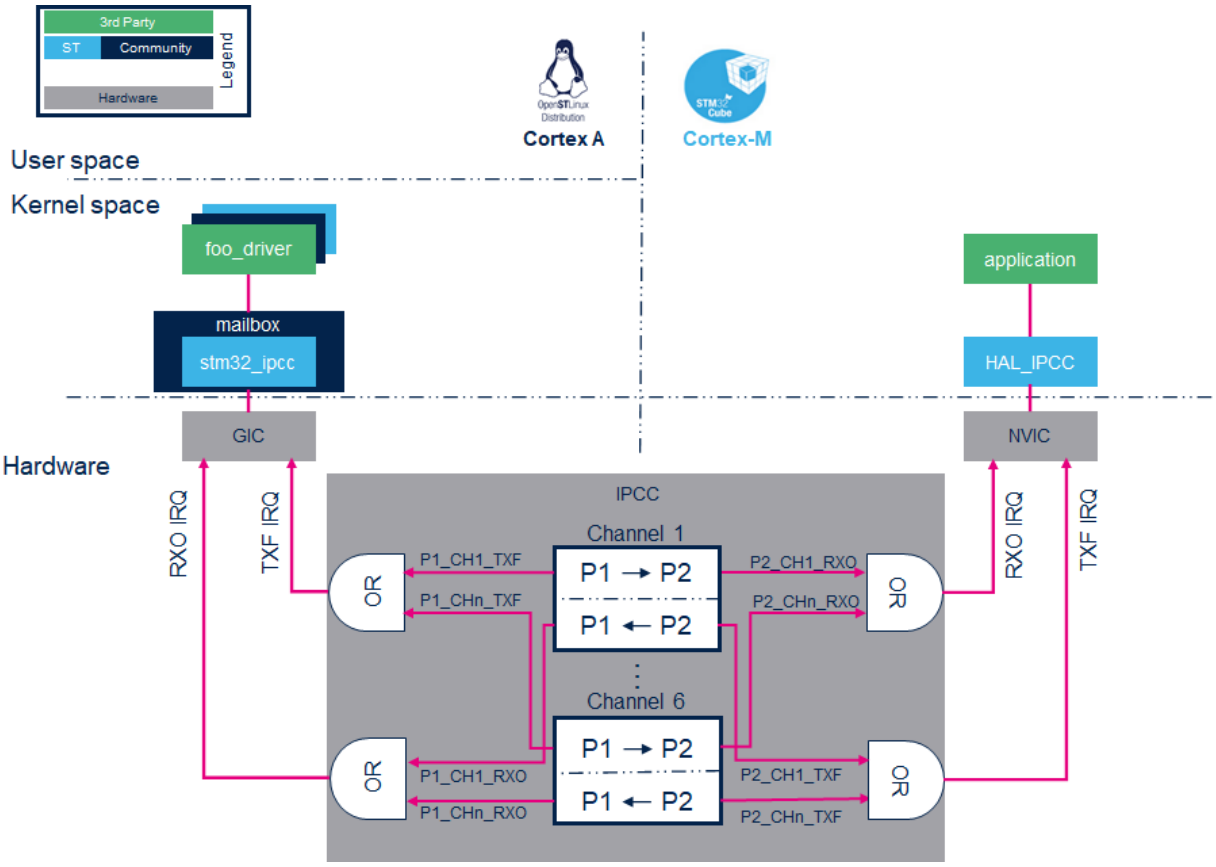
The IPCC is not used at boot time.

3.2 Runtime

3.2.1 Overview

STMicroelectronics distribution uses the IPCC peripheral for inter-processor communication with the following configuration:

- IPCC processor 1 interface is assigned to Arm®Cortex®-A7 non-secure context and handled by Linux mailbox framework.
- IPCC processor 2 interface is assigned to Arm®Cortex®-M4 context and handled by the IPCC HAL driver.



3.2.2 Software frameworks

Domain	Peripheral	Software frameworks	Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)	

Domain	Peripheral	Software frameworks			Comment
Coprocessor	IPCC		Linux mailbox framework	STM32Cube IPCC driver	

3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals) according to the information given in the corresponding software framework article.

The IPCC peripheral is shared between the Arm Cortex-A and Cortex-M contexts. A particular attention must therefore be paid to have a complementary configuration on both contexts. In STMicroelectronics distribution, the IPCC is configured as described below. To ensure the coherency of the system, it is recommended to keep this configuration unchanged in your implementation.

- Processor interface

Processor	Context	
Cortex-A7 non-secure (Linux)	**Cortex-M4** (STM32Cube)	
Processor 1 interface		
Processor 2 interface		

- Channel allocation

Chann	Mode	Usage	Software client frameworks	
Cortex-A7 non-secure (Linux)	**Cortex-M4** (STM32Cube)			
Channel 1	Full-duplex comm	RPMsg transfer from Cortex-M to Cortex-A • The Cortex-M core uses this channel to indicate that a message is available	RPMsg framework	OpenAMP

Channel	Mode	Usage	Software client frameworks	
	unication	• The Cortex-A core uses this channel to indicate that the message is treated		
Channel 2	Full-duplex communication	RPMsg transfer from Cortex-A to Cortex-M • The Cortex-A core uses this channel to indicate that a message is available • The Cortex-M core uses this channel to indicate that the message is treated	RPmsg framework	OpenAMP
Channel 3	Simplex communication	Cortex-M4 shutdown request	RemoteProc framework	CprocSync cube utility
Channel 4		free		
Channel 5		free		
Channel 6		free		

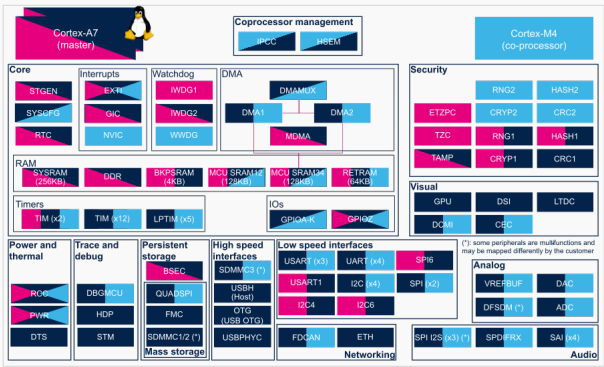
3.2.4 Peripheral assignment

It does not make sense to allocate the IPCC to a single runtime execution context. It is consequently enabled by default for both cores in the STM32CubeMX.

Check boxes illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

- means that the peripheral can be assigned () to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to How to assign an internal peripheral to a runtime context for more information on how to assign peripherals manually or via STM32CubeMX.
The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possiblities might be described in STM32MP15 reference manuals .



Internal peripherals

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Coprocessor	IPCC	IPCC				Shared (none or both)

4 References

Inter-Processor Communication Controller

Receive

Transmit

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cortex®

Open Portable Trusted Execution Environment

Linux® is a registered trademark of Linus Torvalds.

Remote Processor Messaging

Stable: 30.01.2020 - 13:51 Revision: 30.01.2020 - 13:49

A quality version of this page, approved on *30 January 2020*, was based off this revision.

This article gives information about the Linux® mailbox framework. The mailbox framework is involved in interprocessor communication in heterogeneous multicore systems.

Contents

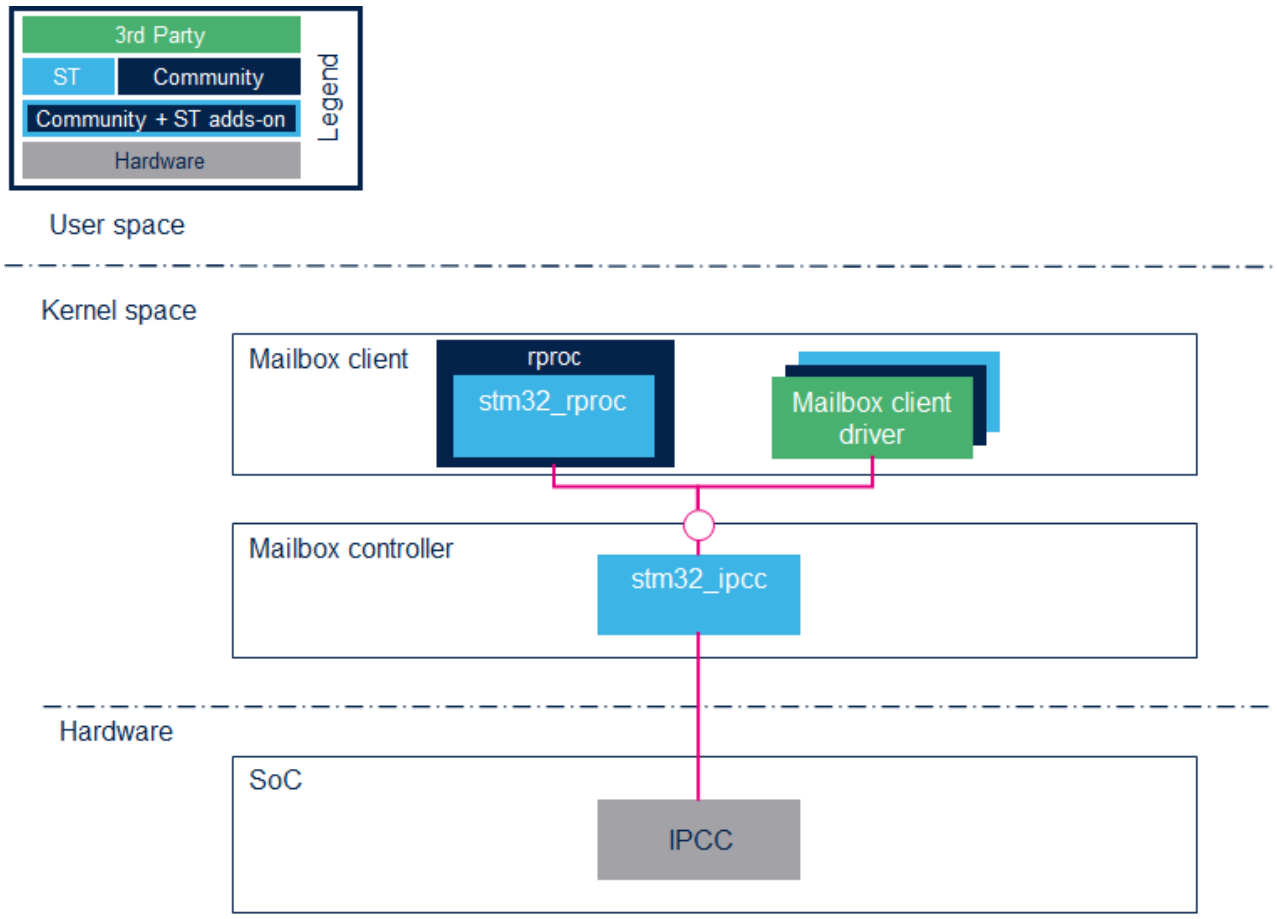
1 Framework purpose

The mailbox is used in interprocessor communication to exchange messages or signals between the host and the coprocessor cores. The mailbox framework is based on:

- A **mailbox controller** that is platform dependent:
  - It is in charge of configuring and handling IRQ from the IPCC peripheral.
  - It provides a generic API to the mailbox client.
- A **mailbox client** that is in charge of the message to send or receive.

A general presentation of the mailbox framework is available in the Linux mailbox documentation [1].

2 System overview



2.1 Component description

- **Mailbox controller**

  The mailbox controller is the **stm32_ipcc**. It configures and controls the IPCC peripheral

- **Mailbox client**

  The user can define his own mailbox client.

  For example, the RPMsg framework uses mailbox for the interprocessor communication.

  In this case the mailbox client is the remoteproc driver that forwards services from/to the RPMsg framework.

2.2 API description

The APIs are described in the Linux documentation:

- Mailbox client API [2]
- Mailbox controller API [3]

3 Configuration

Activate **stm32 IPCC** mailbox in kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel

```
Device drivers  --->
    -*- Mailbox Hardware Support  --->
        <*> STM32 IPCC Mailbox
```

4 Device tree configuration

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

```
ipcc: mailbox@4c001000 {
        compatible = "st,stm32-ipcc";
        #mbox-cells = <1>;
        reg = <0x4c001000 0x400>;
        interrupts-extended = <&intc GIC_SPI 100 IRQ_TYPE_NONE>,
                              <&intc GIC_SPI 101 IRQ_TYPE_NONE>,
                              <&exti 62 1>;
        interrupt-names = "rx", "tx", "wakeup";
        clocks = <&rcc_clk IPCC>;
        wakeup-source;

        Status = "Okay";
 };
```

Then client has to reserve channels. Here is an example of channel allocation for the remoteproc node:

```
&m4_rproc {
       memory-region = <&ipc_share>;
       mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
       mbox-names = "vq0", "vq1", "init_shdn";
       status = "okay";
};
```

5 How to trace and debug the framework

5.1 How to trace

Dynamic debug traces can be added using the following commands:

```
echo -n 'file stm32-ipcc.c +p' > /sys/kernel/debug/dynamic_debug/control
echo -n 'file mailbox.c +p' > /sys/kernel/debug/dynamic_debug/control
```

6 References

- Linux Mailbox documentation
- Mailbox client API
- Mailbox controller API

Inter-Processor Communication Controller

Application programming interface

Remote Processor Messaging

Generic Interrupt Controller

Serial Peripheral Interface

Stable: 07.12.2020 - 10:37 / Revision: 07.12.2020 - 10:35

A quality version of this page, approved on *7 December 2020*, was based off this revision.

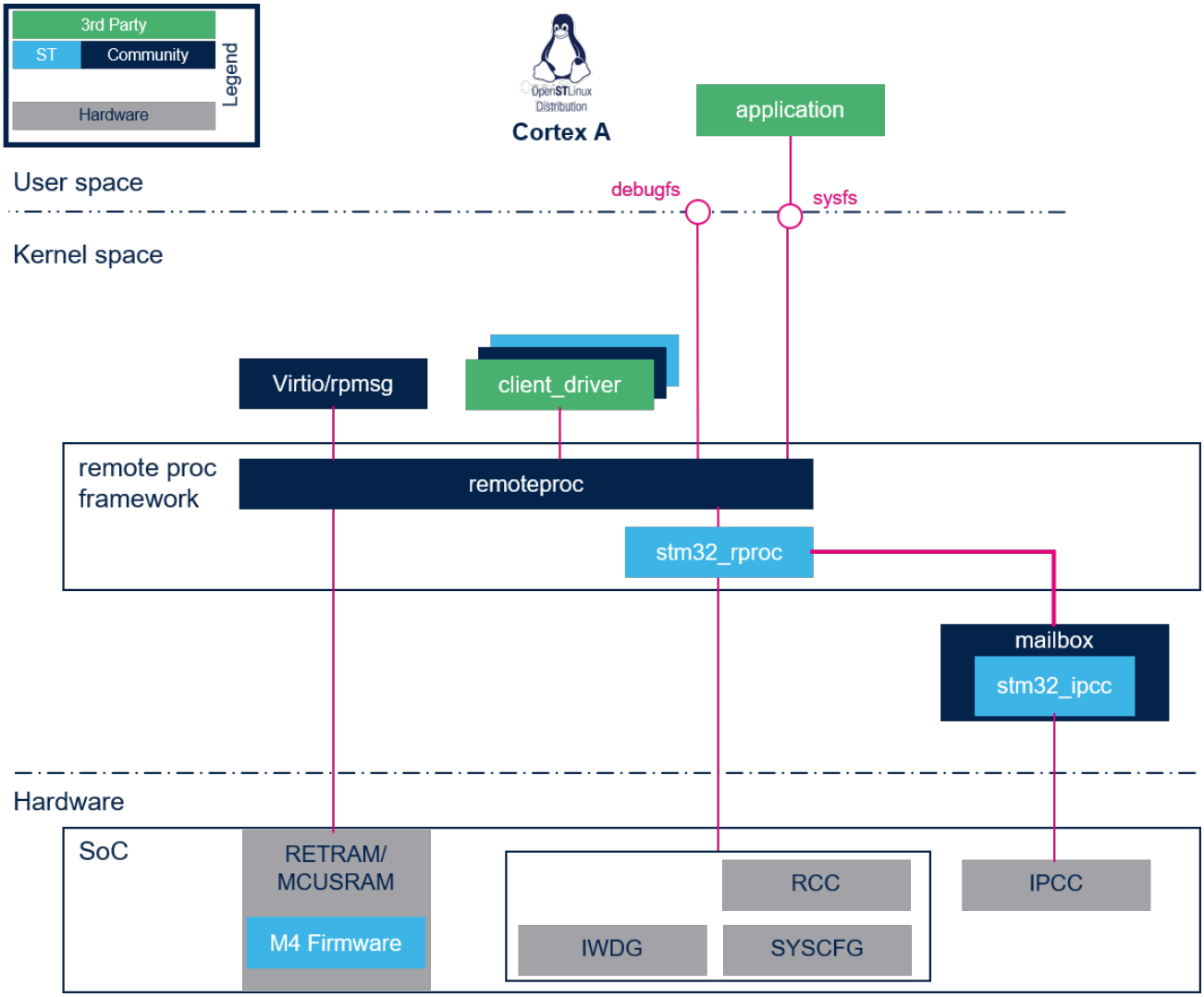
This article gives information about the Linux® remoteproc framework.

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

Warning

The remoteproc framework needs the mailbox framework to be configured. Refer to mailbox kernel configuration for details.

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 *STM32 remoteproc bindings*[2] documentation deals with all required or optional STM32 remoteproc DT properties.

It also introduces *memory regions* properties that define the RETRAM and MCUSRAM base addresses and sizes in RETRAM and MCUSRAM, from the Arm®Cortex®-A point of view..
Simplified example:

```
/* Memory region declaration, containing vring and rpmsg buffers */
reserved-memory {
        /* RETRAM memory region reserved for firmware code and data */
        retram: retram@0x38000000 {
                reg = <0x38000000 0x10000>;
        };
        /* MCUSRAM memory region reserved for firmware code and data */
        mcusram: mcusram@0x10000000 {
                reg = <0x10000000 0x40000>;
        };
        /* MCUSRAM aliased memory region reserved for firmware code and data */
        mcusram2: mcusram2@0x30000000 {
                reg = <0x30000000 0x40000>;
        };
```

```
 /* stm32 M4 remoteproc device */
 m4_rproc: m4@0 {
        ...
        memory-region = <&retram>, <&mcusram>, <&mcusram2>, <&vdev0vring0>,
                        <&vdev0vring1>, <&vdev0buffer>;
        ...
};
```

Information

The firmware memory mapping must be set according to these values in the STM32Cube firmware 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
```

Warning

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
```

Information

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[3]
  - or compile remoteproc as a module and not as kernel built-in driver.

```
*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, 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 backup registers (Cortex-M4 state and resource table address). 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.

Information

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

- Linux kernel remoteproc documentation
- Documentation/devicetree/bindings/remoteproc/stm32-rproc.txt , Linux Foundation, STM32 remoteproc DT bindings
- ramfs-rootfs-initramfs Linux documentation

Executable and linkable file

Inter-Processor Communication

Application programming interface

Device Tree

Cortex®

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Initial ramdisk (https://en.wikipedia.org/wiki/Initial_ramdisk)

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Inter-Processor Communication Controller