Reserved memory

Contents

1. Reserved memory 3
2. DDRCTRL and DDRPHYC internal peripherals 6
3. DMA internal peripheral 11
4. Dmaengine overview 16
5. Linux RPMsg framework overview 25
6. Linux remoteproc framework overview 32
7. MCU SRAM internal memory 41
8. RETRAM internal memory 46

Stable: 11.01.2021 - 17:25 / Revision: 11.01.2021 - 17:23

Stable: 11.01.2021 - 17:29 | Revision: 11.01.2021 - 17:29

A quality version of this page, approved on *11 January 2021*, was based off this revision.

1 Article purpose

The **Reserved-memory** mechanism[1] allows reserving memory regions in the kernel. This mechanism is used by drivers to allocate buffers in specific memory regions (such as MCU SRAM) or to get a dedicated memory pool that will not be managed by Linux® conventionnal memory allocator (in DDR).

2 Use cases

In STM32 MPU Linux OS, the **reserved-memory** is used by:

- the dmaengine driver to reserve the region where DMA buffers are allocated, typically MCU SRAM.
- the remoteproc driver to reserve the regions in RETRAM and MCU SRAM where the coprocessor firmware will be loaded.
- the RPMsg driver to reserve the region where RPMsg buffers used for interprocess communication with the coprocessor, are allocated, typically MCU SRAM.
- the Vivante Gcnano driver to reserve the region where the GPU working memory is allocated, typically the DDR.

3 References

- https://www.kernel.org/doc/Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt

Linux® is a registered trademark of Linus Torvalds.

Microprocessor Unit

Operating System

Graphics Processing Units

Stable: 25.09.2020 - 09:43 / Revision: 25.09.2020 - 09:39

A quality version of this page, approved on *25 September 2020*, was based off this revision.

Contents

1 Article purpose 7
2 Peripheral overview 8
2.1 Features 8
2.2 Security support 8
3 Peripheral usage and associated software 9
3.1 Boot time 9
3.2 Runtime 9
3.2.1 Overview 9
3.2.2 Software frameworks 9
3.2.3 Peripheral configuration 9
3.2.4 Peripheral assignment 9
4 References 11

1 Article purpose

The purpose of this article is to:

- briefly introduce the DDRCTRL and DDRPHYC peripherals and their main features
- indicate the level of security supported by those hardware blocks
- explain how they can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the DDRCTRL and DDRPHYC peripherals.

2 Peripheral overview

DDRCTRL and **DDRPHYC** peripherals are used to configure the physical interface to the external DDR memory.

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 actually implemented.

2.2 Security support

DDRCTRL and DDRPHYC are **secure aware** (under ETZPC control).

Access to the DDR memory can be filtered via the TZC controller: for instance, it is possible to forbid access from the Cortex®-M4 to the DDR region used by the Cortex®-A7.

3 Peripheral usage and associated software

3.1 Boot time

DDRCTRL and DDRPHYC are kept secure and used by the FSBL to initialize the access to the DDR where it loads the SSBL (U-Boot) for execution.
STMicroelectronics wishes to make the DDR memory configuration as easy as possible, for this reason a dedicated application note[1] has been published and a **DDR tuning** function is available in STM32CubeMX tool in order to generate the device tree configuration that is given to the FSBL to perform this initialization.

3.2 Runtime

3.2.1 Overview

DDRCTRL and DDRPHYC are accessed at runtime by the secure monitor (from the FSBL or OP-TEE) to put the DDR in self-refresh state before going into Stop or Standby low power mode.
On Standby exit, the ROM code loads the FSBL that again configures the DDRCTRL and DDRPHYC before proceeding with the wake-up procedure.
The TZC controller is configured by TF-A to split the DDR in two regions:

- the first region, the largest one, is reserved for Linux (Cortex-A7 non-secure context)
- the second region, 32 Mbytes wide, is dedicated for OP-TEE (Cortex-A7 secure context). This area is used by its pager as a cache area from which it can load trusted applications that are authenticated in the SYSRAM internal memory before execution.

This split is visible in the overall memory mapping.

3.2.2 Software frameworks

Domain	Peripheral	Software frameworks			Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/RAM	DDR via DDRCTRL	Memory mapping	Memory mapping		

3.2.3 Peripheral configuration

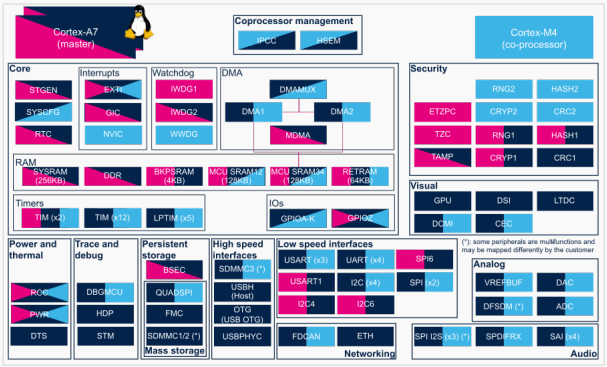
The DDRCTRL and DDRPHYC device tree configuration is generated via STM32CubeMX tool, according to the DDR characteristics (type, size, frequency, speed grade). This configuration is applied during boot time by the FSBL (see Boot chain overview): TF-A or U-Boot SPL.

3.2.4 Peripheral assignment

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	Periphera l	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/RAM	DDR via DDRCTRL	DDR				

4 References

- AN5168 - DDR configuration on STM32MP1 Series MPU

Doubledata rate (memory domain)

Cortex®

Trusted Firmware for Arm Cortex-A

Linux® is a registered trademark of Linus Torvalds.

Open Portable Trusted Execution Environment

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

First Stage Boot Loader

Stable: 13.10.2020 - 08:29 / Revision: 13.10.2020 - 08:29

A quality version of this page, approved on *13 October 2020*, was based off this revision.

Contents

1 Article purpose 12
2 Peripheral overview 13
2.1 Features 13
2.2 Security support 13
3 Peripheral usage and associated software 14
3.1 Boot time 14
3.2 Runtime 14
3.2.1 Overview 14
3.2.2 Software frameworks 14
3.2.3 Peripheral configuration 14
3.2.4 Peripheral assignment 14
4 References 16

1 Article purpose

The purpose of this article is to:

- briefly introduce the DMA peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the DMA peripheral.

2 Peripheral overview

The **DMA** peripheral is used to perform direct accesses from/to a device or a memory. Each DMA instance supports 8 channels. The selection of the device connected to each DMA channel and controlling the DMA transfers is done via the DMAMUX.

Note: Directly accessing DDR from the DMA is not recommended for high-bandwith or latency-critical transfers. This means that DMA transfers configured by the Arm®Cortex®-A7 operating system, that usually target buffers in external memory, require a hardware mechanism to chain the DMA and a MDMA channel in order to achieve the following flow:

DDR<-> MDMA <-> MCU SRAM <-> DMA <-> device

This feature was already present on STM32H7 microcontroller Series. It is documented in application note AN5001[1].

2.1 Features

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

2.2 Security support

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

3 Peripheral usage and associated software

3.1 Boot time

The DMA is not used at boot time.

3.2 Runtime

3.2.1 Overview

Each DMA instance can be allocated to:

- the Arm®Cortex®-A7 non-secure core to be controlled in Linux® by the dmaengine framework

or

- the Arm®Cortex®-M4 to be controlled in STM32Cube MPU Package by the DMA 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)			
Core/DMA	DMA		Linux dmaengine framework	STM32Cube DMA driver	

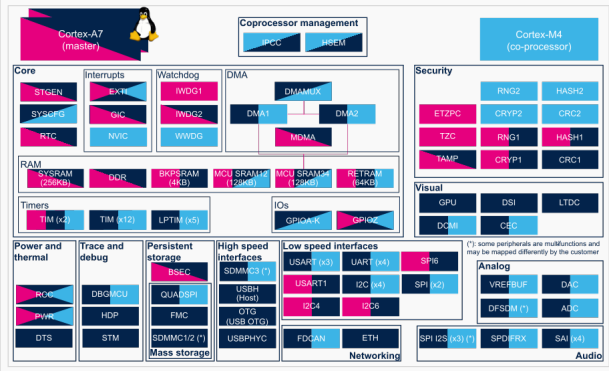
3.2.3 Peripheral configuration

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.

3.2.4 Peripheral assignment

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.



Internal peripherals

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.

Domain	Periphera l	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/DMA	DMA	DMA1				Assignment (single choice)
		DMA2				Assignment (single choice)

4 References

- http://www.st.com/resource/en/application_note/dm00360392.pdf

Direct Memory Access

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

Cortex®

Doubledata rate (memory domain)

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained, system-on-chip (SoC) designs.)

Linux® is a registered trademark of Linus Torvalds.

Microprocessor Unit

Open Portable Trusted Execution Environment

Stable: 26.11.2020 - 13:29 / Revision: 18.11.2020 - 11:04

A quality version of this page, approved on *26 November 2020*, was based off this revision.

This article provides basic information about the DMA engine and how STM32 DMA, DMAMUX and MDMA drivers are plugged into it.

Contents

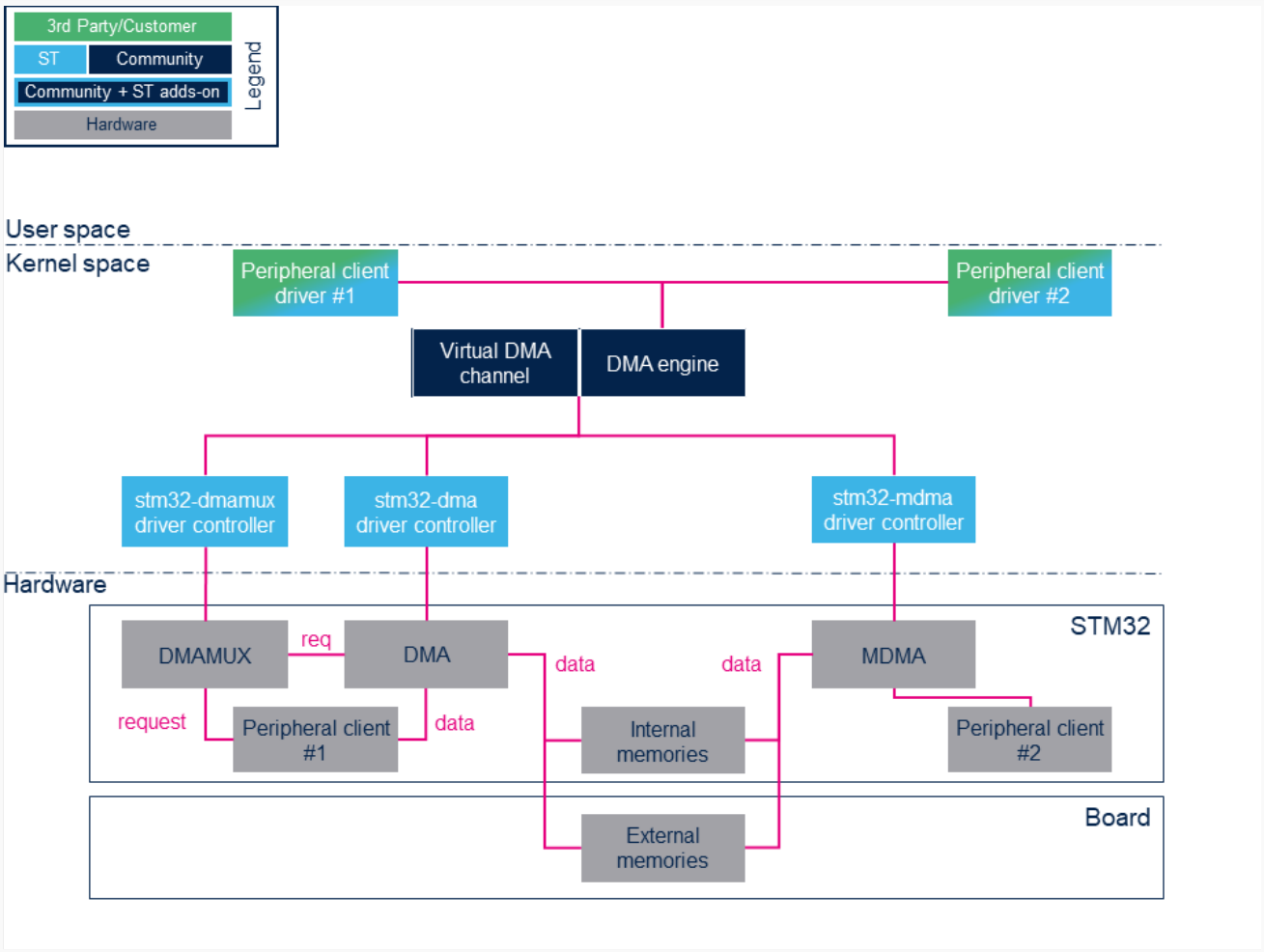
1 Framework purpose 17
2 System overview 18
2.1 Component description 18
2.2 APIs description 19
3 Configuration 20
3.1 Kernel Configuration 20
3.2 Device Tree configuration 20
4 How to trace and debug the framework 21
4.1 How to trace 21
4.2 How to debug 21
4.2.1 devfs 21
4.2.2 Debugfs 22
4.2.3 dmatest 22
5 Source code location 23
6 To go further 24
7 References 25

1 Framework purpose

This article provides basic information about the DMA framework. However it is worth browsing the Kernel documentation related to **DMA concept**[1].

The direct memory access (DMA) is a feature that allows some hardware subsystems to access memory independently from the central processing unit (CPU).
The DMA can transfer data between peripherals and memory or between memory and memory.

2 System overview



DMA Engine

2.1 Component description

- **Peripheral DMA client drivers**:

DMA clients are drivers that are mapped on the **DMA API**[2].

- **DMA engine**:

The DMA engine is the engine core on which all clients rely.

Refer to **DMA provider**[1] for useful information on DMA internal behaviour.

- **Virtual DMA channel support**:

The virtual DMA channel support manages virtual DMA channels and DMA requests queues. This layer is no used by DMA clients.

- **STM32 xDMA driver**:

The STM32 xDMA driver is used to develop the DMA engine API.

- **STM32 DMAMUX driver**:

The STM32 DMAMUX driver request multiplexer allows routing DMA request lines between the device peripherals and the DMA controllers.

- **DMAMUX, DMA and MDMA IP controller**:

This is the STM32 DMA controller that handles data transfers between peripherals and memories or memory and memory connected to the same bus.

DMAMUX (DMA request router): DMAMUX internal peripheral
DMA: DMA internal peripheral
MDMA : MDMA internal peripheral

- **Peripheral clients**:

Peripheral clients are peripherals where at least one DMA request line is mapped on DMAMUX.

- **Memories**:

Memories can be either internal (e.g. SRAM, RETRAM or BCKRAM) or external (DDR).

2.2 APIs description

Please refer to **DMA Engine API Guide**[3] for a clear description of the DMA framework API.

In addition, going through **Dynamic API**[4] provides insight on the DMA memory allocation API. The client has to rely on this API to properly allocate DMA buffers so that they are processed by the DMA engine without any trouble.

The document **Dynamic DMA mapping Guide**[5] can be read in conjunction with the previous one.
It presents some examples and usecases.

3 Configuration

3.1 Kernel Configuration

The DMA engine and driver are enabled throughout menu config (see Menuconfig or how to configure kernel):

For DMA:

```
Device Drivers ->
    [*] DMA Engine support ->
        [*] STMicroelectronics STM32 DMA support
```

For DMAMUX:

```
Device Drivers ->
    [*] DMA Engine support ->
        [*] STMicroelectronics STM32 dma multiplexer support
```

For MDMA

```
Device Drivers ->
    [*] DMA Engine support ->
        [*] STMicroelectronics STM32 master dma support
```

3.2 Device Tree configuration

The DT configuration can be done using the STM32CubeMX.

Refer to the following articles for a description of the DT configuration:

- For DMA: DMA device tree configuration
- For DMAMUX: DMAMUX device tree configuration
- For MDMA: MDMA device tree configuration

4 How to trace and debug the framework

4.1 How to trace

Through menuconfig, enable DMA engine debugging and DMA engine verbose debugging (including STM32 drivers):

```
Device Drivers ->
    [*] DMA Engine support ->
        [*] DMA Engine debugging
        [*]   DMA Engine verbose debugging (NEW)
```

4.2 How to debug

4.2.1 devfs

sysfs entry can be used to browse for available DMA channels.

More information can be found in sysfs.

The following command lists all the registered DMA channels:

```
Board $> ls /sys/class/dma/
dma0chan0   dma0chan13  dma0chan18  dma0chan22  dma0chan27  dma0chan31  dma0chan8
dma1chan3  dma2chan0  dma2chan5
dma0chan1   dma0chan14  dma0chan19  dma0chan23  dma0chan28  dma0chan4   dma0chan9
dma1chan4  dma2chan1  dma2chan6
dma0chan10  dma0chan15  dma0chan2   dma0chan24  dma0chan29  dma0chan5   dma1chan0
dma1chan5  dma2chan2  dma2chan7
dma0chan11  dma0chan16  dma0chan20  dma0chan25  dma0chan3   dma0chan6   dma1chan1
dma1chan6  dma2chan3
dma0chan12  dma0chan17  dma0chan21  dma0chan26  dma0chan30  dma0chan7   dma1chan2
dma1chan7  dma2chan4
```

Each channel is expanded as follows:

```
Board $> ls -la /sys/class/dma/dma0chan0/
total 0
drwxr-xr-x  3 root root    0 Jun  7 21:22 .
drwxr-xr-x 34 root root    0 Jun  7 21:22 ..
-r--r--r--  1 root root 4096 Jun  9 13:11 bytes_transferred
lrwxrwxrwx  1 root root    0 Jun  9 13:11 device -> ../../../58000000.dma
-r--r--r--  1 root root 4096 Jun  9 13:11 in_use
-r--r--r--  1 root root 4096 Jun  9 13:11 memcpy_count
drwxr-xr-x  2 root root    0 Jun  9 13:11 power
lrwxrwxrwx  1 root root    0 Jun  9 13:11 subsystem -> ../../../../../../class/dma
-rw-r--r--  1 root root 4096 Jun  7 21:22 uevent
```

device indicates which DMA driver manages the channel.

echoing **in_use** indicates whether the channel has been allocated or not.

```
Board $> cat /sys/class/dma/dma0chan0
/in_use
1
```

4.2.2 Debugfs

debugfs entries are available. They are documented in *Part III - Debug drivers use of the DMA-API*[4].

4.2.3 dmatest

dmatest can be used to validate or debug DMA engine and driver without using client devices. This module is more a test than a debug module. It performs a memory-to-memory copy using standard DMA engine API.

For details on how to use this kernel module, refer to [6].

5 Source code location

DMA: drivers/dma/stm32-dma.c
MDMA: drivers/dma/stm32-mdma.c
DMAMUX: drivers/dma/stm32-dmamux.c

DMA engine:

- Engine: drivers/dma/dmaengine.c
- Virtual channel support: drivers/dma/virt-dma.c

6 To go further

Very useful documentation can be found at DMAEngine documentation

7 References

- 1.01.1 DMA provider
- DMA API
- DMA Engine API Guide
- 4.04.1 Documentation/DMA-API.txt Dynamic DMA mapping using the generic device
- Documentation/DMA-API-HOWTO.txt Dynamic DMA mapping Guide
- driver-api/dmaengine/dmatest.html

Direct Memory Access

Central processing unit

Application programming interface

Doubledata rate (memory domain)

Device Tree

Device File System (See https://en.wikipedia.org/wiki/Device_file#DEVFS for more details)

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

Stable: 16.02.2021 - 17:39 Revision: 16.02.2021 - 17:00

A quality version of this page, approved on *16 February 2021*, was based off this revision.

This article gives information about the Linux® Remote Processor Messaging (RPMsg) framework. The RPMsg framework is a virtio-based messaging bus that allows a local processor to communicate with remote processors available on the system.

Contents

1 Framework purpose 26
2 System overview 27
2.1 Component description 28
2.2 RPMsg definitions 28
2.3 API description 28
3 Configuration 29
3.1 Kernel configuration 29
3.2 Device tree configuration 29
4 How to use the framework 30
5 How to trace and debug the framework 31
5.1 How to trace 31
6 References 32

1 Framework purpose

The Linux®RPMsg framework is a messaging mechanism implemented on top of the virtio framework[1][2] to communicate with a remote processor. It is based on virtio vrings to send/receive messages to/from the remote CPU over shared memory.

The vrings are uni-directional, one vring is dedicated to messages sent to the remote processor, and the other vring is used for messages received from the remote processor. In addition, shared buffers are created in memory space visible to both processors.

The Mailbox framework is then used to notify cores when new messages are waiting in the shared buffers.

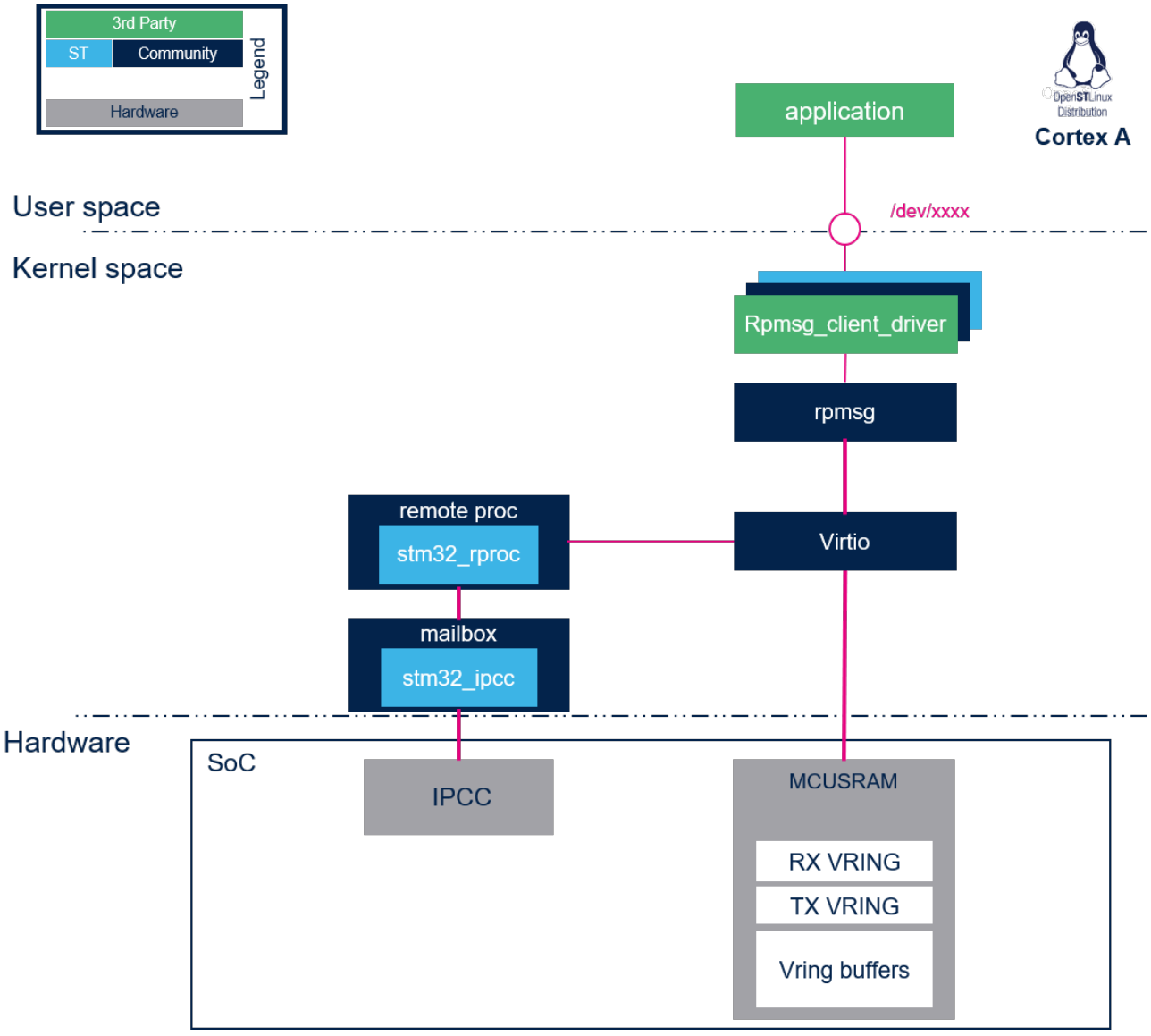
Relying on these frameworks, The RPMsg framework implements a communication based on channels. The channels are identified by a textual name and have a local (“source”) RPMsg address, and a remote (“destination”) RPMsg address”.

On the remote processor side, a RPMSG framework must also be implemented. Several solutions exist, we recommend using OpenAMP

Overviews of the communication mechanisms involved are available at:

- OpenAMP wiki [3][4]

2 System overview



2.1 Component description

- **remoteproc**: The remoteproc framework allows different platforms/architectures to control (power on, load firmware, power off) the remote processors. This framework also adds rpmsg virtio devices for remote processors that support the RPMsg protocol. More details on this framework are available in the remote proc framework page.
- **virtio**: VirtIO framework that supports virtualization. It provides an efficient transport layer based on a shared ring buffer (vring). For more details about this framework please refer to the link below:
  - Virtio: An I/O virtualization framework for Linux [1]
  - virtio introduction - SlideShare [2]
- **rpmsg**: A virtio-based messaging bus that allows kernel drivers to communicate with remote processors available on the system. It provides the messaging infrastructure, facilitating the writing of wire-protocol messages by client drivers. Client drivers can then, in turn, expose appropriate user space interfaces if needed.
- **rpmsg_client_driver** is the client driver that implements a service associated to the remote processor. This driver is probed by the RPMsg framework when an associated service is requested by a remote processor using a "new service announcement" RPMsg message.

2.2 RPMsg definitions

To implement an Rpmsg client, **channel** and **endpoint** concepts need to be understood for a good understanding of the framework.

- RPMsg channel:

An RPMsg client is associated to a communication channel between master and remote processors. This RPMsg client is identified by the textual **service name**, registered in the RPMsg framework. The communication channel is established when a match is found between the local service name registered and the remote service announced.

- RPMsg endpoint:

The RPMsg endpoints provide logical connections through an RPMsg channel. An RPMsg endpoint has a unique source address and associated call back function, allowing the user to bind multiple endpoints on the same channel. When a client driver creates an endpoint with the local address, all the inbound messages with a destination address equal to the endpoint local address are routed to that endpoint. Notice that every channel has a default endpoint, which enables applications to communicate without even creating new endpoints.

2.3 API description

The User API usage is described in Linux kernel RPMsg documentation [5]

3 Configuration

3.1 Kernel configuration

The RPMsg framework is automatically enabled when the remote STM32_RPROC configuration is activated.

3.2 Device tree configuration

No device tree configuration is needed. The Memory region allocated for rpmsg buffers is declared in the remote proc framework device tree.

4 How to use the framework

The Rpmsg framework is used by linux driver client. For details and an example of a simple client, please refer to associated Linux documentation [5]

5 How to trace and debug the framework

5.1 How to trace

RPMsg and virtio dynamic debug traces can be added using the following commands:

```
echo -n 'file virtio_rpmsg_bus.c +p' > /sys/kernel/debug/dynamic_debug/control
echo -n 'file virtio_ring.c +p' > /sys/kernel/debug/dynamic_debug/control
```

6 References

- 1.01.1 An I/O virtualization framework for Linux
- 2.02.1 virtio introduction - SlideShare
- RPMsg Messaging Protocol
- RPMsg Communication Flow
- 5.05.1 Linux kernel rpmsg documentation

Remote Processor Messaging

Central processing unit

Application programming 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.

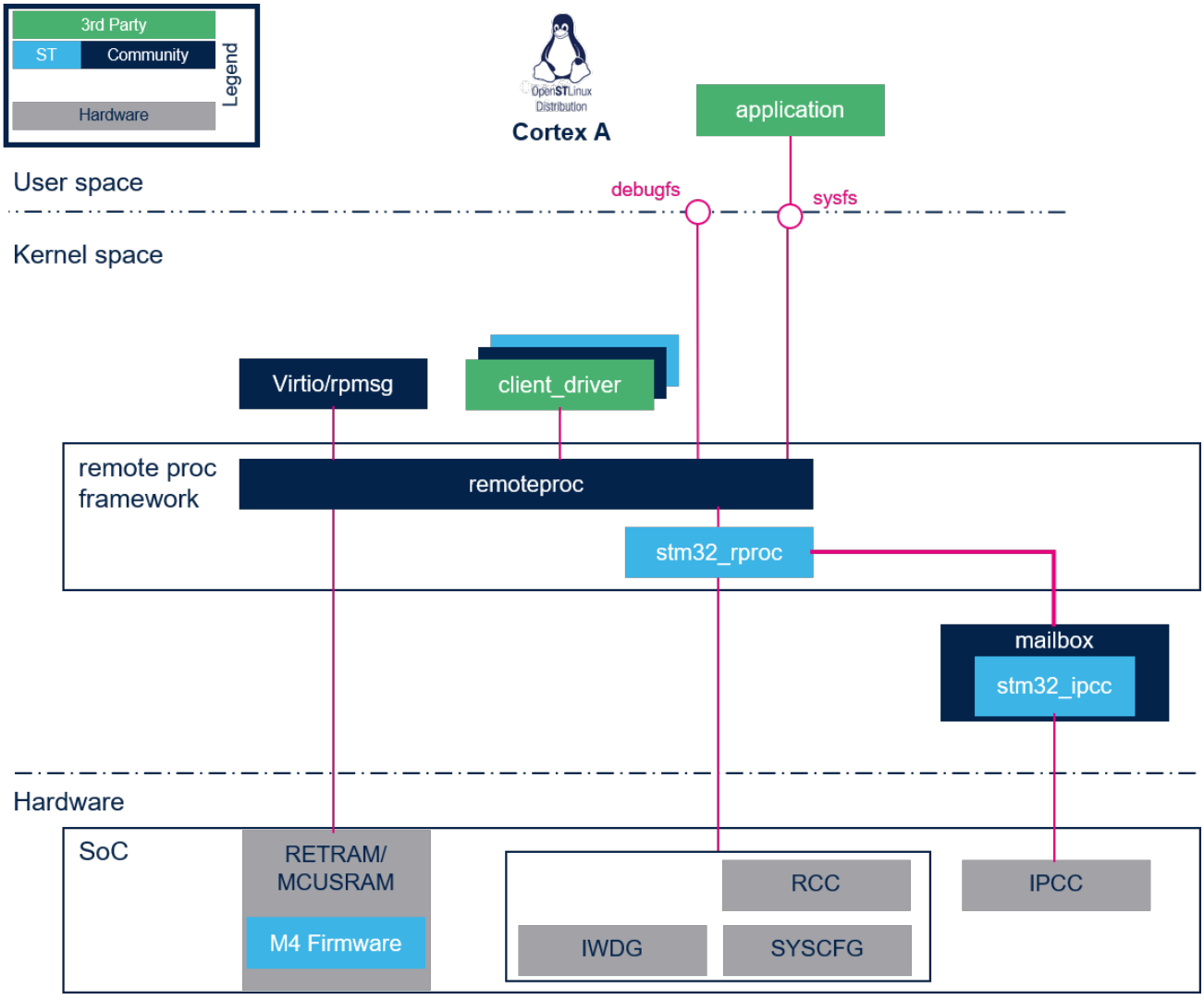
Contents

1 Framework purpose 33
2 System overview 34
2.1 Component description 34
2.2 API description 35
3 Configuration 36
3.1 Kernel configuration 36
3.2 Device tree configuration 36
4 How to use the framework 38
4.1 Remote processor boot 38
4.1.1 Remote processor boot through sysfs 38
4.1.2 Remote processor 'auto' boot (not recommended by STMicroelectronics) 38
4.1.3 Remote processor 'early' boot 39
4.2 Remote processor stop 39
5 How to trace and debug the framework 40
5.1 How to monitor 40
5.2 How to trace 40
6 References 41

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

Stable: 04.02.2020 - 15:59 / Revision: 04.02.2020 - 15:48

A quality version of this page, approved on *4 February 2020*, was based off this revision.

Contents

1 Article purpose 42
2 Peripheral overview 43
2.1 Features 43
2.2 Security support 43
3 Peripheral usage and associated software 44
3.1 Boot time 44
3.2 Runtime 44
3.2.1 Overview 44
3.2.2 Software frameworks 44
3.2.3 Peripheral configuration 44
3.2.4 Peripheral assignment 45

1 Article purpose

The purpose of this article is to

- briefly introduce the MCU RAM memory and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain how to configure the MCU RAM memory.

2 Peripheral overview

The **MCU SRAM** internal memory is 384-Kbyte wide and physically near to the Cortex®-M4 for optimized performances from this core. It is split into four separate banks:

- MCU SRAM1 (128 Kbytes)
- MCU SRAM2 (128 Kbytes)
- MCU SRAM3 (64 Kbytes)
- MCU SRAM4 (64 Kbytes)

Those banks have individual security control (cf. security support below) and automatic clock gating (for power management optimization), but they are not supplied when the system goes to Standby low power mode, so their content is lost in that case.

2.1 Features

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

2.2 Security support

Each MCU SRAM1/SRAM2/SRAM3/SRAM4 bank is **secure aware** (under ETZPC control).

3 Peripheral usage and associated software

3.1 Boot time

The ROM code uses the MCU SRAM1 to store the USB context during a boot on USB for Flash programming (with STM32CubeProgrammer).

Linux remoteproc framework (running on the Cortex®-A7) loads the Cortex®-M4 firmware code into the MCU SRAM, except the exception table that must be loaded in the RETRAM since the Cortex®-M4 is looking for its reset entry point at address 0x00000000. The overall memory mapping is shown in the platform memory mapping section.

3.2 Runtime

3.2.1 Overview

Each MCU SRAM bank can be allocated to:

- the Arm®Cortex®-A7 secure for using in OP-TEE

or

- the Arm®Cortex®-A7 non-secure for using in Linux® with reserved memory, that is used by the dmaengine (for DMA buffers management) or RPMsg for interprocess communication with the coprocessor

and/or

- the Arm®Cortex®-M4 for using in STM32Cube

Notice the **and/or** allocation between Cortex®-A7 non-secure and Cortex®-M4, meaning that it is possible to share banks between those cores, typically to realize inter process communication between RPMsg on Linux side and OpenAMP on STM32Cube side.

The default assignement set in STMicroelectronics distribution is in line with the platform memory mapping, that can be adapted by the platform user.

3.2.2 Software frameworks

Domain	Peripheral	Software frameworks			Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/RAM	MCU SRAM	OP-TEE overview	Linux reserved memory	STM32Cube	

3.2.3 Peripheral configuration

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

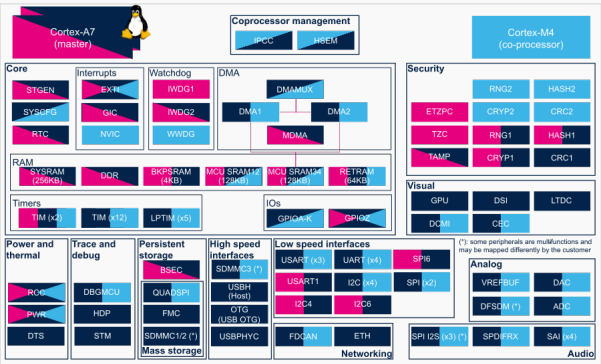
The several SRAM banks are accessible via different address ranges in order to benefit from the Cortex-M4 multiple ports.

3.2.4 Peripheral assignment

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	Periphera l	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/RAM	MCU SRAM	SRAM1				Assignment (between A7 S and A7 NS / M4) Shareable (between A7 NS and M4)
		SRAM2				Assignment (between A7 S and A7 NS / M4) Shareable (between A7 NS and M4)
						Assignment (between A7 S and
		SRAM3				A7 NS / M4) Shareable (between A7 NS and M4)
		SRAM4				Assignment (between A7 S and A7 NS / M4) Shareable (between

Domain	Peripheral	Runtime allocation				Comment
						A7 NS and M4)

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained, system-on-chip (SoC) designs.)

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Cortex®

Linux® is a registered trademark of Linus Torvalds.

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

Open Portable Trusted Execution Environment

Stable: 04.02.2020 - 15:59 / Revision: 04.02.2020 - 15:50

A quality version of this page, approved on *4 February 2020*, was based off this revision.

Contents

1 Peripheral overview 47
1.1 Features 47
1.2 Security support 47
2 Peripheral usage and associated software 48
2.1 Boot time 48
2.2 Runtime 48
2.2.1 Overview 48
2.2.2 Software frameworks 48
2.2.3 Peripheral configuration 48
2.2.4 Peripheral assignment 49

1 Peripheral overview

The **RETRAM** internal memory is 64 Kbytes wide and is physically near to the Arm®Cortex®-M4 for optimized performance from the core. It is located in the VSW power domain, allowing it to be supplied during Standby low power mode, and to retain retention firmware that can be executed very quickly by the Cortex-M4 on wake up from Standby mode.

1.1 Features

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

1.2 Security support

The RETRAM is a **secure** peripheral (under ETZPC control).

2 Peripheral usage and associated software

2.1 Boot time

Linux® remoteproc framework (running on the Cortex-A7) loads the Cortex-M4 firmware to the RETRAM, starting at address 0x00000000. At least, it must load the part of the firmware containing the vector table, since the Cortex-M4 reset entry point is address 0x00000004. The rest of the firmware code is loaded into the MCU SRAM. The overall memory mapping is shown in the platform memory mapping section.

2.2 Runtime

2.2.1 Overview

The Cortex-M4 vector table is mapped from address 0x00000000 (so to the RETRAM) at reset, but it can be remapped by software to any other location by means of the vector table offset register (VTOR). Beyond the reset entry point (0x00000004), the exception table also contains the software entries table used by the NVIC to branch the software execution to the right interrupt service routine.

While going to Standby low power mode, the RETRAM can remain supplied, so it can preserve a (small) Cortex-M4 piece of retention firmware that is executed on wake up when the ROM code (running on Cortex-A7) restarts the Cortex-M4.
All these constraints make the RETRAM the minimum (and default) choice for Cortex-M4 firmware.

RETRAM can be allocated to:

- the Cortex-A7 secure to be used under OP-TEE.

or

- the Cortex-A7 non-secure to be used under Linux as reserved memory.

or

- the Cortex-M4 for use with the STM32Cube MPU Package, either for **runtime firmware** that ca be mapped in both RETRAM and MCU SRAM, or for **retention firmware** that only fits into the RETRAM, but could have some data in MCU SRAM (keeping in mind that these data are lost while entering Standby low power mode).

2.2.2 Software frameworks

Domain	Peripheral	Software frameworks			Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Core/RAM	RETRAM	OP-TEE overview	Linux reserved memory	STM32Cube	

2.2.3 Peripheral configuration

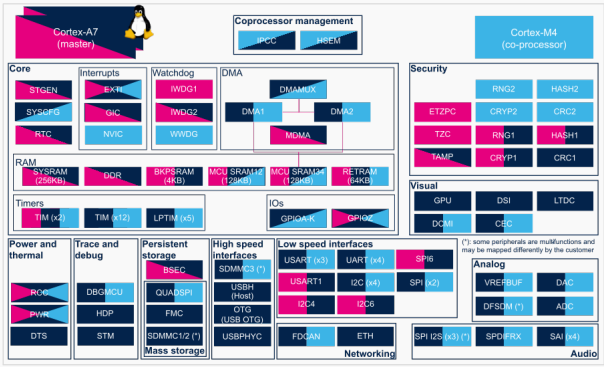
The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (especially for external peripherals), according to the information given in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

2.2.4 Peripheral assignment

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)			
Core/RAM	RETRAM	RETRAM				Assignment (single choice)

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

Cortex®

Linux® is a registered trademark of Linus Torvalds.

Microprocessor Unit

Open Portable Trusted Execution Environment

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)