MDMA device tree configuration

Contents

Stable: 19.03.2021 - 17:20 / Revision: 19.03.2021 - 12:38

Stable: 19.03.2021 - 17:20 / Revision: 19.03.2021 - 12:38

A quality version of this page, approved on *19 March 2021*, was based off this revision.

Contents

1 Article purpose

This article explains how to configure the MDMA internal peripheral when it is assigned to the Linux® OS. In that case, it is controlled by the Dmaengine overview.

The configuration is performed using the Device tree mechanism that provides a hardware description of the MDMA internal peripheral, used by the STM32 MDMA Linux driver and by the DMA framework.

Hardware description is a combination of:

- STM32 MDMA peripheral
- and STM32 MDMA client

2 DT bindings documentation

Complete device tree bindings can be found at this location: [1].

3 DT configuration

This hardware description is a combination of the **STM32 microprocessor** device tree files (*.dtsi* extension) and **board** device tree files (*.dts* extension). See the Device tree for an explanation of the device tree file split.

STM32CubeMX can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.

3.1 DT configuration (STM32 level)

At device level, MDMA is declared as follows:

```
mdma1: dma@58000000 {
    compatible = "st,stm32h7-mdma";
    reg = <0x58000000 0x1000>;
    interrupts = <GIC_SPI 122 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&rcc MDMA>;
    resets = <&scmi0_reset RST_SCMI0_MDMA>;
    #dma-cells = <6>;
    dma-channels = <32>;
    dma-requests = <48>;
};
```

The DTS file is located under arch/arm/boot/dts/stm32mp151.dtsi .

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

No board device tree configuration is required.
The whole configuration remains at STM32 level.

4 How to configure the DT using STM32CubeMX

The STM32CubeMX tool can be used to configure the STM32MPU device and get the corresponding platform configuration device tree files.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

5 References

Please refer to the following links for additional information:

- Documentation/devicetree/bindings/dma/st,stm32-mdma.yaml

Stable: 05.11.2021 - 11:08 / Revision: 05.11.2021 - 11:05

A quality version of this page, approved on *5 November 2021*, was based off this revision.

Contents

1 Purpose

The objective of this chapter is to give general information about the device tree.

An extract of the **device tree specification**[1] explains it as follows:

"A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent. ... Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time."

In other words, a device tree describes the hardware that can not be located by probing.

1.1 Device tree basis

This webinar will give the foundations of device tree applied to STM32MP1 products and boards. This is highly recommended to start from this if you are beginner on this subject.

- Device Tree for STM32MP [2]

1.2 Source files

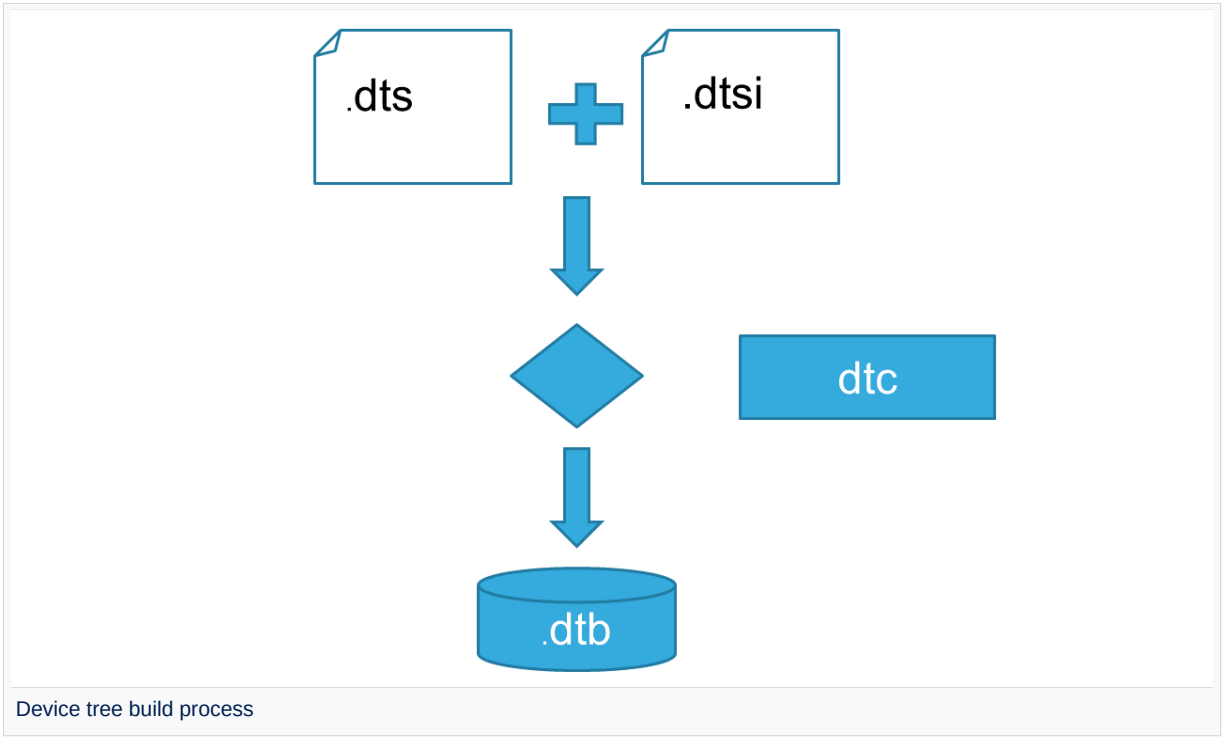
- **.dts:** The device tree source (DTS). This format is a textual representation of a device tree in a form that can be processed by DTC (Device Tree Compiler) into a binary file expected by software components: Linux® Kernel, U-Boot and TF-A.
- **.dtsi:** Source files that can be included from a DTS file.
- **.h:** Header files that can be included from DTS and DTSI files.

1.3 Bindings

The device tree data structures and properties are named **bindings**. Those bindings are described in:

- The Device tree specification[1] for generic bindings.
- The software component documentations:
  - Linux® Kernel: Linux kernel device tree bindings
  - U-Boot: doc/device-tree-bindings/
  - TF-A: TF-A device tree bindings

1.4 Build



Device tree build process

- A tool named DTC[3](Device Tree Compiler) allows compiling the DTS sources into a binary.
  - input file: the *.dts* file described in section above (that includes itself one or several .dtsi and .h files).
  - output file: the *.dtb* file described in section above.

DTC source code is located here[4]. DTC tool is also available directly in particular software components: **Linux Kernel, U-Boot, TF-A ...**. For those components, the device tree building is directly integrated in the component build process.

Information

If dts files use some defines, dts files should be preprocessed before being compiled by DTC.

1.5 Tools

The device tree compiler offers also some tools:

- **fdtdump:** Print a readable version of a flattened device tree file (dtb)
- **fdtget:** Read properties from a device tree
- **fdtput:** Write properties to a device tree
- ...

There are several ways to get those tools:

- In the device tree compiler project source code[4]
- Directly in software components: **Kernel, u-boot, tf-a ...**
- Available in Debian package[5]

2 STM32

For STM32MP1, the device tree is used by three software components: Linux® kernel, U-Boot and TF-A.

The device tree is part of the OpenSTLinux distribution. It can also be generated by STM32CubeMX tool.

To have more information about the device tree usage on STM32MP1 (how the device tree source files are split, how to find the device tree source files per software components, how is STM32CubeMX generating the device tree ...) see STM32MP15 device tree page.

3 How to go further

- Device Tree Reference[6] - eLinux.org
- Device Tree usage[7] - eLinux.org

4 References

- [1.01.1] https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.2(latest) ,Device tree specification
- https://www.youtube.com/watch?v=a9CZ1Uk3OYQ, Device Tree for STM32MP
- https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/manual.txt(master), DTC manual
- [4.04.1] https://git.kernel.org/pub/scm/utils/dtc/dtc.git(master) ,DTC source code
- https://packages.debian.org/search?keywords=device-tree-compiler(master) ,DTC debian package
- Device Tree Reference, eLinux.org
- Device Tree Usage, eLinux.org

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

Stable: 04.10.2021 - 08:05 / Revision: 04.10.2021 - 08:04

A quality version of this page, approved on *4 October 2021*, 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.

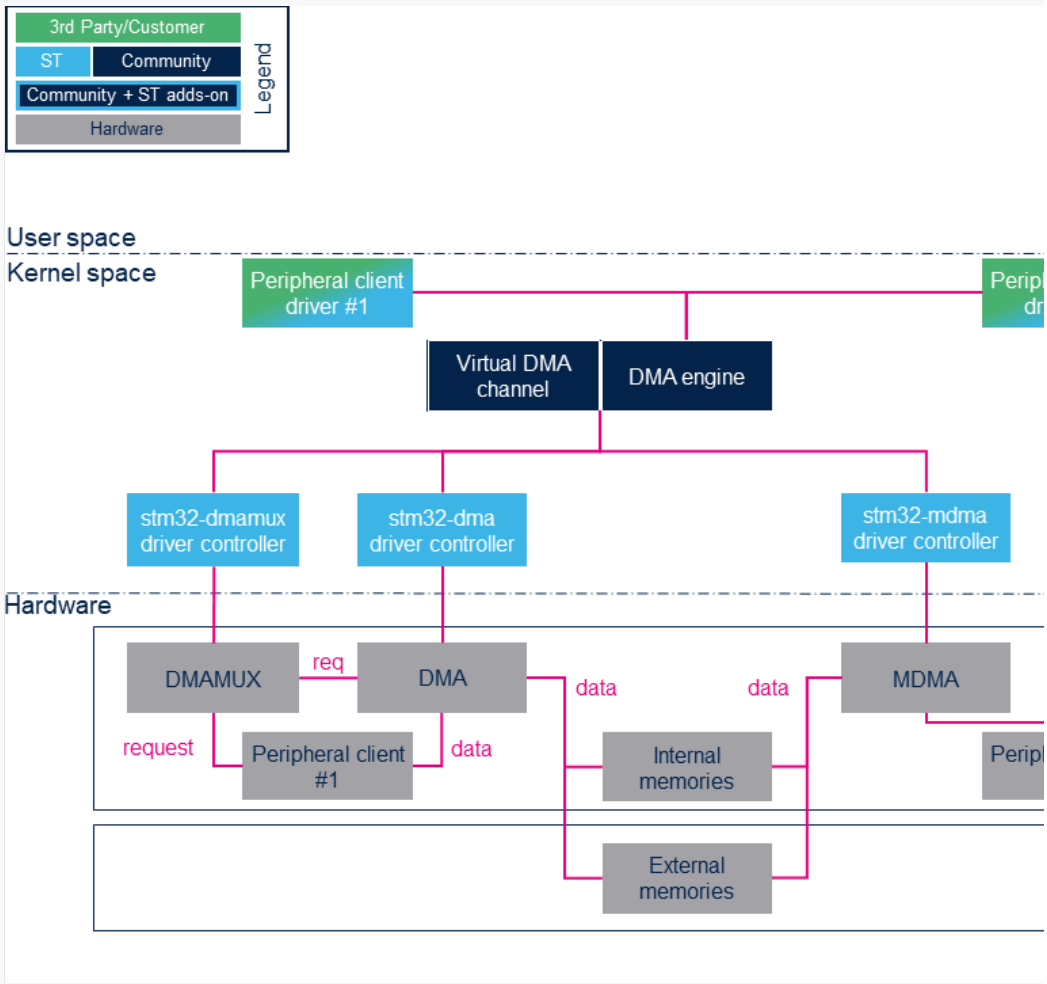
Contents

1 Framework purpose

This article provides basic information about the DMA framework. For additional information, browse 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 behavior.

- **Virtual DMA channel support**:

The virtual DMA channel manages virtual DMA channels and DMA request queues. This layer is not 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 the routing of 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 (such as SRAM, RETRAM or backup RAM) or external (DDR memories).

2.2 API description

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] presents some examples and usecases.
It can be read in conjunction with the previous one.

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 device tree (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 use the framework

Refer to the **DMA Engine API Guide**[3] for an exhaustive description of the DMA engine client API.

4.1 Request a DMA channel

The device tree configuration at STM32 level (arch/arm/boot/dts/stm32mp151.dtsi) contains the "dmas" and "dma-names" properties in the peripheral nodes, that have a request line mapped.

The peripheral drivers just have to request one or more DMA channels. This is generally done during probe.

```
#include <linux/dmaengine.h>
struct dma_chan *dma_request_chan(struct device *dev, const char *name);
```

Thanks to the name, the dmaengine finds a channel that matches the configuration specified in the dmas property.

```
struct dma_chan *chan_rx, *chan_tx;

chan_rx =  dma_request_chan(&pdev->dev, "rx");
chan_tx = dma_request_chan(&pdev->dev, "tx");
```

The returned channel can be null if there are no more available channels or none of them fits the requested configuration. In this case, the peripheral must check the returned channel and switch to Interrupt mode.

4.2 Configure the DMA channel

A part of channel configuration comes from the dmas property in the peripheral device tree node. Refer to the description in DMA controller device tree bindings. dma_slave_config structure is also used to set up the channel. Refer to the dma_slave_config structure definition in include/linux/dmaengine.h for an exhaustive description.

```
struct dma_slave_config {
        enum dma_transfer_direction direction;
        phys_addr_t src_addr;
        phys_addr_t dst_addr;
        enum dma_slave_buswidth src_addr_width;
        enum dma_slave_buswidth dst_addr_width;
        u32 src_maxburst;
        u32 dst_maxburst;
        u32 src_port_window_size;
        u32 dst_port_window_size;
        bool device_fc;
        unsigned int slave_id;
};
```

Source/Destination addresses, Source/Destination address width, Source/Destination maximum burst are used by the DMA controller driver to configure the channel. The user must use dmaengine_slave_config() to set this dma_slave_config structure in the DMA controller driver.

```
struct dma_slave_config config;

/* In case of memory to device (TX) */
memset(&config, 0, sizeof(config));
config.dst_addr = phy_addr + txdr_offset;
config.dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;
config.dst_maxburst = 1;
config.direction = DMA_MEM_TO_DEV;

/* In case of device to memory (RX/Capture) */
memset(&config, 0, sizeof(config));
config.src_addr = phy_addr + rxdr_offset;
config.src_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;
config.src_maxburst = 1;
config.direction = DMA_DEV_TO_MEM;

int dmaengine_slave_config(struct dma_chan *chan, struct dma_slave_config *config);
```

4.3 Configure the DMA transfer

The DMA engine transfer API must be used to prepare the DMA transfer. Three modes are supported by STM32 DMA controller drivers:

- slave_sg: prepares a transfer of a list of scatter-gather buffer from/to a peripheral
- dma_cyclic: prepares a cyclic operation from/to a peripheral until the operation is stopped by the user
- dma_memcpy: prepares a memcpy operation (seldom used except by dmatest)

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
           struct dma_chan *chan, struct scatterlist *sgl,
           unsigned int sg_len, enum dma_data_direction direction,
           unsigned long flags);

struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
           struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
           size_t period_len, enum dma_data_direction direction);

struct dma_async_tx_descriptor *dmaengine_prep_dma_memcpy(
           struct dma_chan *chan, dma_addr_t dst, dma_addr_t src,
           size_t len, unsigned long flags);
```

A peripheral driver completion callback can be set up using the callback* fields of the dma_async_tx_descriptor returned by the dmaengine_prep* function.

```
struct dma_async_tx_descriptor *txdesc;

txdesc = dmaengine_prep_...
txdesc->callback = peripheral_driver_dma_callback;
txdesc->callback_param = peripheral_dev;
```

4.4 Submit the DMA transfer

Once the transfer is prepared, it can be submitted for execution. It is added to the pending queue using dmaengine_submit() used as parameter of dma_submit_error() to digest the returned value.

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
static inline int dma_submit_error(dma_cookie_t cookie)

ret = dma_submit_error(dmaengine_submit(desc));
```

The transfer can then be started using dma_async_issue_pending(). If the channel is idle, the first transfer in the queue is started.

```
void dma_async_issue_pending(struct dma_chan *chan);
```

On completion of each DMA transfer, a DMA interrupt is raised, then the next transfer in the queue is started and a tasklet is triggered. When scheduled, this tasklet calls the peripheral driver completion callback, provided it is set.

4.5 Terminate the DMA transfer

Two variants are available to force the DMA channel to stop an ongoing transfer. No completion callback is called for an incomplete transfer and the data in DMA controller FIFO may be lost. Refer to the **DMA Engine API Guide**[3] for more details.

- dmaengine_terminate_async(): this function can be called from atomic context or from within a completion callback;
- dmaengine_terminate_sync(): this function must not be called from atomic context or from within a completion callback.

```
int dmaengine_terminate_sync(struct dma_chan *chan)
int dmaengine_terminate_async(struct dma_chan *chan)
```

dmaengine_synchronize() must be used after dmaengine_terminate_async() and outside atomic context or completion callback, to synchronize the termination of the DMA channel with the current context. The function waits for the completion of the ongoing transfer and any callback before returning.

```
void dma_release_channel(struct dma_chan *chan)
```

4.6 Release the DMA channel

The peripheral driver can ask for new transfers or simply release the channel if it is no more needed. It is typically done by calling the peripheral driver remove() function.

```
void dma_release_channel(struct dma_chan *chan)
```

5 How to trace and debug the framework

5.1 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)
```

5.2 How to debug

5.2.1 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
```

5.2.2 Debugfs

debugfs entries are available. The user can get information about the DMA devices and the used channels through the /sys /kernel/debug/dmaengine .

```
root@stm32mp1:~# cat /sys/kernel/debug/dmaengine/summary
dma0 (58000000.dma-controller): number of channels: 32
 dma0chan0    | 48000000.dma-controller:ch0
 dma0chan1    | 48000000.dma-controller:ch1
 dma0chan2    | 48000000.dma-controller:ch2
 dma0chan3    | 48000000.dma-controller:ch3
 dma0chan4    | 48000000.dma-controller:ch4
 dma0chan5    | 48000000.dma-controller:ch5
 dma0chan6    | 48000000.dma-controller:ch6
 dma0chan7    | 48000000.dma-controller:ch7
 dma0chan8    | 48001000.dma-controller:ch0
 dma0chan9    | 48001000.dma-controller:ch1
 dma0chan10   | 48001000.dma-controller:ch2
 dma0chan11   | 48001000.dma-controller:ch3
 dma0chan12   | 48001000.dma-controller:ch4
 dma0chan13   | 48001000.dma-controller:ch5
 dma0chan14   | 48001000.dma-controller:ch6
 dma0chan15   | 48001000.dma-controller:ch7
 dma0chan16   | 54002000.hash:in

dma1 (48000000.dma-controller): number of channels: 8
 dma1chan0    | 4000e000.serial:rx (via router: 48002000.dma-router)
 dma1chan1    | 4000e000.serial:tx (via router: 48002000.dma-router)
 dma1chan2    | 4000b000.audio-controller:tx (via router: 48002000.dma-router)
 dma1chan3    | 4000b000.audio-controller:rx (via router: 48002000.dma-router)
 dma1chan4    | 4400b004.audio-controller:tx (via router: 48002000.dma-router)
 dma1chan5    | 4400b024.audio-controller:rx (via router: 48002000.dma-router)

dma2 (48001000.dma-controller): number of channels: 8
```

Other DMA debugfs entries are available when the Linux® kernel is compiled using "Enable debugging of DMA-API usage" configuration. They are documented in *Part III - Debug drivers use of the DMA-API*[4].

5.2.3 dmatest

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

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

6 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

7 To go further

Very useful documentation can be found at DMAEngine documentation

8 References

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

Stable: 13.10.2020 - 08:31 / Revision: 13.10.2020 - 08:31

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

Contents

1 Article purpose

The purpose of this article is to:

- briefly introduce the MDMA 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 MDMA peripheral.

2 Peripheral overview

The **MDMA** is used to perform high-speed data transfers between memory and memory or between peripherals and memory. The MDMA controller offers 32 channels. The selection of the device connected to each channel and controlling DMA transfers is done in MDMA peripheral.

Among all the requestor lines described in the STM32MP15 reference manuals, DMA channels are the only lines that allow to perform transfers with chained DMA and MDMA (refer to DMA internal peripheral article). As a result, when a device is not connected to the MDMA, it is anyway possible to operate in DMA mode via the DMA controller and chain DMA and MDMA.

2.1 Features

Refer to 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 MDMA is a **secure** peripheral. This means that it performs each transfer in the context of the master that requested it:

- a transfer requested by the Arm® Cortex®-A7 **non-secure** core propagates **non-secure accesses** to the targeted device and /or memory.
- a transfer requested by Arm Cortex-A7 **secure** core propagates **secure accesses** to the targeted device and/or memory.

3 Peripheral usage and associated software

3.1 Boot time

The MDMA is used at boot time by the FMC.

3.2 Runtime

3.2.1 Overview

The MDMA is visible from the Arm Cortex-M4 core. However, it is not supported in this context by STM32MPU Embedded Software distribution.

As stated in the 'Security support' chapter above, the MDMA is a secure peripheral. This means that its channels have to be allocated to:

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

and

- the Arm Cortex-A7 secure core to be controlled by the MDMA OP-TEE driver

STM32CubeMX allows to distinguish between non-secure and secure channels, among all the available channels.

3.2.2 Software frameworks

Domain	Peripheral	Software components		Comment
OP-TEE	Linux	STM32Cube		
Core/DMA	MDMA	OP-TEE MDMA driver	Linux dmaengine framework	

3.2.3 Peripheral configuration

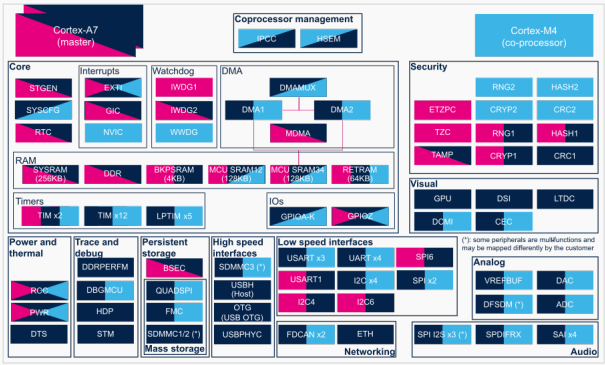
The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals), according to the information given in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

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/DMA	MDMA	MDMA				Shareable (multiple choices supported)

4 How to go further

Not applicable

5 References

Stable: 23.09.2020 - 13:22 / Revision: 12.06.2020 - 13:25

A quality version of this page, approved on *23 September 2020*, was based off this revision.

1 STM32CubeMX overview

This article describes STM32CubeMX, an official STMicroelectronics graphical software configuration tool.

The STM32CubeMX application helps developers to use the STM32 by means of a user interface, and guides the user through to the initial configuration of a firmware project.

It provides the means to:

- configure pin assignments, the clock tree, or internal peripherals
- simulate the power consumption of the resulting project
- configure and tune DDR parameters
- generate HAL initialization code for Cortex-M4
- generate the Device Tree for a Linux kernel, TF-A and U-Boot firmware for Cortex-A7

It uses a rich library of data from the STM32 microcontroller portfolio.

The application is intended to ease the initial development phase by helping developers to select the best product in terms of features and power.

2 STM32CubeMX main features

- Peripheral and middleware parameters

  Presents options specific to each supported software component

- Peripheral assignment to processors

  Allows assignment of each peripheral to Cortex-A Secure, Cortex-A Non-Secure, or Cortex-M processors

- Power consumption calculator

  Uses a database of typical values to estimate power consumption, DMIPS, and battery life

- Code generation

  Makes code regeneration possible, while keeping user code intact

- Pinout configuration

  Enables peripherals to be chosen for use, and assigns GPIO and alternate functions to pins

- Clock tree initialization

  Chooses the oscillator and sets the PLL and clock dividers

- DDR tunning tool

  Ensures the configuration, testing, and tuning of the MPU DDR parameters. Using U-Boot-SPL Embedded Software.

3 How to get STM32CubeMX

Please, refer to the following link STM32CubeMX to find STM32CubeMX, the Release Note, the User Manual and the product specification.

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