FMC device tree configuration

Contents

Stable: 15.04.2021 - 16:08 / Revision: 15.04.2021 - 16:07

Stable: 15.4.2021 - 15.04 Revision: 15.4.2021 - 15.04

A quality version of this page, approved on *15 April 2021*, was based off this revision.

Contents

1 For ecosystem release v2.1.0

1.1 Article purpose

This article explains how to configure the **FMC** internal peripheral when it is assigned to the Linux®OS. In that case, the FMC NAND Flash controller is controlled by the MTD framework.

The configuration is performed using the device tree mechanism that provides a hardware description of the FMC peripheral, used by the STM32 FMC Linux drivers and by the MTD framework.

1.2 DT bindings documentation

The FMC device tree bindings are composed of:

- generic MTD NAND bindings [1].
- FMC NAND Flash controller driver bindings [2].
- FMC external bus interface driver bindings [3].

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.

1.3.1 DT configuration (STM32 level)

The FMC peripheral node is located in *stm32mp151.dtsi*[4] file.

```
   fmc: memory-controller@58002000 {                          Comments
       #address-cells = <2>;
       #size-cells = <1>;
       compatible = "st,stm32mp1-fmc2-ebi";
       reg = <0x58002000 0x1000>;                             --> This region contains the
register location
       clocks = <&rcc FMC_K>;
       resets = <&rcc FMC_R>;
       status = "disabled";

       ranges = <0 0 0x60000000 0x04000000>, /* EBI CS 1 */ --> External bus interface
region is used to address up to four external devices
                <1 0 0x64000000 0x04000000>, /* EBI CS 2 */
                <2 0 0x68000000 0x04000000>, /* EBI CS 3 */
                <3 0 0x6c000000 0x04000000>, /* EBI CS 4 */
                <4 0 0x80000000 0x10000000>; /* NAND */     --> NAND Flash controller
region is used  to address NAND Flash memory devices

       nand-controller@4,0 {
           #address-cells = <1>;
           #size-cells = <0>;
           compatible = "st,stm32mp1-fmc2-nfc";
           reg = <4 0x00000000 0x1000>,                       --> Regions 1 to 3
respectively contain the data, command and address space for CS0
```

```
                <4 0x08010000 0x1000>,
                <4 0x08020000 0x1000>,
                <4 0x01000000 0x1000>,                          --> Regions 4 to 6 contain
the same areas for CS1
                <4 0x09010000 0x1000>,
                <4 0x09020000 0x1000>;
            interrupts = <GIC_SPI 48 IRQ_TYPE_LEVEL_HIGH>;      --> The interrupt number used
            dmas = <&mdma1 20 0x2 0x12000a02 0x0 0x0 0x0>,      --> DMA specifiers [5]
                   <&mdma1 20 0x2 0x12000a08 0x0 0x0 0x0>,
                   <&mdma1 21 0x2 0x12000a0a 0x0 0x0 0x0>;
            dma-names = "tx", "rx", "ecc";
            status = "disabled";
        };
    };
```

Warning

This device tree part related to the STM32 should be kept as is, customer should not modify it.

1.3.2 DT configuration of the external bus interface controller (board level)

The FMC external bus interface controller may connect up to four external devices.

```
    &fmc {                                                      Comments
        pinctrl-names = "default", "sleep";                     --> For pinctrl
configuration, please refer to Pinctrl device tree configuration
        pinctrl-0 = <&fmc2_pins_b>;
        pinctrl-1 = <&fmc2_sleep_pins_b>;
        status = "okay";                                        --> Enable the node

        ksz8851: ks8851mll@1,0 {
            compatible = "micrel,ks8851-mll";                   --> Configure the external
device
            reg = <1 0x0 0x2>, <1 0x2 0x20000>;
            interrupt-parent = <&gpioc>;
            interrupts = <3 IRQ_TYPE_LEVEL_LOW>;
            bank-width = <2>;

            st,fmc2-ebi-cs-mux-enable;                          --> Configure the
transactions with the external device
            st,fmc2-ebi-cs-transaction-type = <4>;
            st,fmc2-ebi-cs-buswidth = <16>;
            st,fmc2-ebi-cs-address-setup-ns = <5>;
            st,fmc2-ebi-cs-address-hold-ns = <5>;
            st,fmc2-ebi-cs-bus-turnaround-ns = <5>;
            st,fmc2-ebi-cs-data-setup-ns = <45>;
            st,fmc2-ebi-cs-data-hold-ns = <1>;
        };
    };
```

1.3.3 DT configuration of the NAND Flash controller (board level)

The FMC NAND Flash controller may connect to one SLC NAND Flash memory (with a maximum of 2 dies per package).

```
    &fmc {                                                      Comments
        pinctrl-names = "default", "sleep";                     --> For pinctrl
configuration, please refer to Pinctrl device tree configuration
        pinctrl-0 = <&fmc2_pins_a>;
        pinctrl-1 = <&fmc2_sleep_pins_a>;
        status = "okay";                                        --> Enable the node
```

```
        nand-controller@4,0 {
            status = "okay";                                    --> Enable the NAND
controller node

            nand@0 {
                reg = <0>;                                      --> Describe the CS line
assigned to the NAND chip
                nand-on-flash-bbt;                              --> Store the bad block table
on NAND Flash memory
                nand-ecc-strength = <8>;                        --> Number of bits to correct
per ECC step
                nand-ecc-step-size = <512>;                     --> Number of data bytes that
are covered by a single ECC step
                #address-cells = <1>;
                #size-cells = <1>;
            };
        };
    };
```

The supported ECC strength and step size are:

- nand-ecc-strength = <1>, nand-ecc-step-size = <512> (HAMMING).
- nand-ecc-strength = <4>, nand-ecc-step-size = <512> (BCH4).
- nand-ecc-strength = <8>, nand-ecc-step-size = <512> (BCH8).

Warning

It is recommended to check the ECC requirements in the datasheet of the memory provider.

1.3.4 DT configuration examples

The below example shows how to configure the FMC NAND Flash controller when a SLC 8-bit NAND Flash memory device is connected (ECC requirement: 8 bits / 512 bytes).

```
    &fmc {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&fmc2_pins_a>;
        pinctrl-1 = <&fmc2_sleep_pins_a>;
        status = "okay";

        nand-controller@4,0 {
            status = "okay";

            nand: nand@0 {
                reg = <0>;
                nand-on-flash-bbt;
                #address-cells = <1>;
                #size-cells = <1>;

                partition@0 {
                    ...
                };
            };
        };
    };
```

The below example shows how to configure the FMC NAND Flash controller when a SLC 8-bit NAND Flash memory device is connected (ECC requirement: 4 bits / 512 bytes).

```
    &fmc {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&fmc2_pins_a>;
        pinctrl-1 = <&fmc2_sleep_pins_a>;
        status = "okay";

        nand-controller@4,0 {
            status = "okay";

            nand: nand@0 {
                reg = <0>;
                nand-on-flash-bbt;
                nand-ecc-strength = <4>;
                nand-ecc-step-size =
<512>;
                #address-cells = <1>;
                #size-cells = <1>;

                partition@0 {
                    ...
                };
            };
        };
    };
```

1.4 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.

1.5 References

Please refer to the following links for full description:

- Documentation/devicetree/bindings/mtd/nand-controller.yaml (v5.4-stm32mp-r2)
- Documentation/devicetree/bindings/mtd/st,stm32-fmc2-nand.yaml (v5.4-stm32mp-r2)
- Documentation/devicetree/bindings/memory-controllers/st,stm32-fmc2-ebi.yaml (v5.4-stm32mp-r2)
- arch/arm/boot/dts/stm32mp151.dtsi (v5.4-stm32mp-r2)
- Documentation/devicetree/bindings/dma/stm32-mdma.txt (v5.4-stm32mp-r2)

2 For ecosystem release v2.0.0

2.1 Article purpose

This article explains how to configure the **FMC** internal peripheral when it is assigned to the Linux®OS. In that case, it is controlled by the MTD framework.

The configuration is performed using the device tree mechanism that provides a hardware description of the FMC peripheral, used by the STM32 FMC Linux driver and by the MTD framework.

2.2 DT bindings documentation

The FMC device tree bindings are composed of:

- generic MTD nand bindings [1].
- FMC driver bindings [2].

2.3 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.

2.3.1 DT configuration (STM32 level)

The FMC peripheral node is located in *stm32mp151.dtsi*[3] file.

```
   fmc: nand-controller@58002000 {                                  Comments
       compatible = "st,stm32mp15-fmc2";
       reg = <0x58002000 0x1000>,                                   --> First region contains
the register location
             <0x80000000 0x1000>,                                   --> Regions 2 to 4
respectively contain the data, command and address space for CS0
             <0x88010000 0x1000>,
             <0x88020000 0x1000>,
             <0x81000000 0x1000>,                                   --> Regions 5 to 7 contain
the same areas for CS1
             <0x89010000 0x1000>,
             <0x89020000 0x1000>;
       interrupts = <GIC_SPI 48 IRQ_TYPE_LEVEL_HIGH>;               --> The interrupt number used
       dmas = <&mdma1 20 0x10 0x12000A02 0x0 0x0 0>,                --> DMA specifiers [4]
              <&mdma1 20 0x10 0x12000A08 0x0 0x0 0>,
              <&mdma1 21 0x10 0x12000A0A 0x0 0x0 0>;
       dma-names = "tx", "rx", "ecc";
       clocks = <&rcc FMC_K>;
       resets = <&rcc FMC_R>;
       status = "disabled";
   };
```

Warning

This device tree part related to the STM32 should be kept as is, customer should not modify it.

2.3.2 DT configuration (board level)

The FMC peripheral may connect to one SLC NAND Flash memory (with a maximum of 2 dies per package).

```
   &fmc {                                                          Comments
       pinctrl-names = "default", "sleep";                         --> For pinctrl
configuration, please refer to Pinctrl device tree configuration
       pinctrl-0 = <&fmc2_pins_a>;
       pinctrl-1 = <&fmc2_sleep_pins_a>;
       status = "okay";                                            --> Enable the node
       #address-cells = <1>;
       #size-cells = <0>;

       nand: nand@0 {
           reg = <0>;                                              --> Describe the CS line
assigned to the NAND chip
           nand-on-flash-bbt;                                      --> Store the bad block
table on NAND Flash memory
           nand-ecc-strength = <8>;                                --> Number of bits to
correct per ECC step
           nand-ecc-step-size = <512>;                             --> Number of data bytes
that are covered by a single ECC step
           #address-cells = <1>;
           #size-cells = <1>;
       };
   };
```

The supported ECC strength and step size are:

- nand-ecc-strength = <1>, nand-ecc-step-size = <512> (HAMMING).
- nand-ecc-strength = <4>, nand-ecc-step-size = <512> (BCH4).
- nand-ecc-strength = <8>, nand-ecc-step-size = <512> (BCH8).

2.3.3 DT configuration examples

The below example shows how to configure the FMC controller when a SLC 8-bit NAND Flash memory device is connected (ECC requirement: 8 bits / 512 bytes).

```
   &fmc {
       pinctrl-names = "default", "sleep";
       pinctrl-0 = <&fmc2_pins_a>;
       pinctrl-1 = <&fmc2_sleep_pins_a>;
       status = "okay";
       #address-cells = <1>;
       #size-cells = <0>;

       nand: nand@0 {
           reg = <0>;
           nand-on-flash-bbt;
           #address-cells = <1>;
           #size-cells = <1>;

           partition@0 {
               ...
           };
       };
   };
```

The below example shows how to configure the FMC controller when a SLC 8-bit NAND Flash memory device is connected (ECC requirement: 4 bits / 512 bytes).

```
&fmc {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&fmc2_pins_a>;
    pinctrl-1 = <&fmc2_sleep_pins_a>;
    status = "okay";
    #address-cells = <1>;
    #size-cells = <0>;

    nand: nand@0 {
        reg = <0>;
        nand-on-flash-bbt;
        nand-ecc-strength = <4>;
        nand-ecc-step-size = <512>;
        #address-cells = <1>;
        #size-cells = <1>;

        partition@0 {
            ...
        };
    };
};
```

2.4 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.

2.5 References

Please refer to the following links for full description:

- Documentation/devicetree/bindings/mtd/nand-controller.yaml (v5.4-stm32mp-r1)
- Documentation/devicetree/bindings/mtd/stm32-fmc2-nand.txt (v5.4-stm32mp-r1)
- arch/arm/boot/dts/stm32mp151.dtsi (v5.4-stm32mp-r1)
- Documentation/devicetree/bindings/dma/stm32-mdma.txt (v5.4-stm32mp-r1)

Operating System

Memory Technology Device

Device Tree

Flash memories combine high density and cost effectiveness of EPROMs with the electrical erasability of EEPROMs. For this reason, the Flash memory market is one of the most exciting areas of the semiconductor industry today and new applications requiring in system reprogramming, such as cellular telephones, automotive engine management systems, hard disk drives, PC BIOS software for Plug & Play, digital TV, set top boxes, fax and other modems, PC cards and multimedia CD-ROMs, offer the prospect of very high volume demand.

Generic Interrupt Controller

Serial Peripheral Interface

Direct Memory Access

Single-Level Cell is a kind of NAND flash

Elliptic curve cryptography

Error Correction Capability

Stable: 04.02.2020 - 07:47 / Revision: 04.02.2020 - 07:34

A quality version of this page, approved on *4 February 2020*, 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. For more information, please refer to the device tree specification[1]

1.1 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 device tree in the form expected by software components: Linux® Kernel, U-Boot and TF-A.

- **.dtsi:** Source files that can be included from a DTS file.

1.2 Bindings

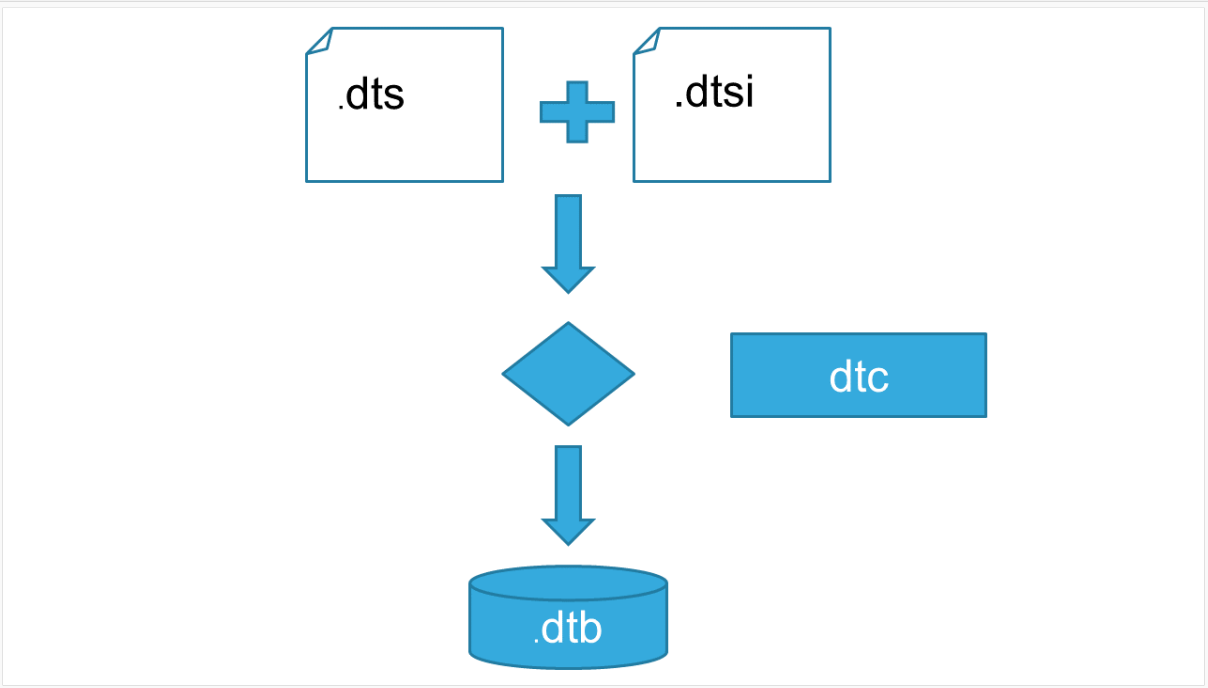
The device tree data structures and properties are named **bindings**. Those bindings are described in:

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

1.3 Build

- A tool named DTC (Device Tree Compiler) allows compiling the DTS sources into a binary.
- input file: the *.dts* file described in section above.
- output file: the *.dtb* file described in section above.
- More information are available in DTC manual[2].

- DTC source code is located here[3]. 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.



Device tree build process

Information

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

1.4 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[3]
- Directly in software components: **Kernel, u-boot, tf-a ...**
- Available in Debian package[4]

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 for Dummies[5] - Free Electrons
- Device Tree Reference[6] - eLinux.org
- Device Tree usage[7] - eLinux.org

4 References

- 1.01.11.2 https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.2(latest) ,Device tree specification
- https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/manual.txt(master) ,DTC manual
- 3.03.1 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 for Dummies, Free Electrons
- Device Tree Reference, eLinux.org
- Device Tree Usage, eLinux.org

Device Tree Source (in software context) or Digital Temperature Sensor (in peripheral context)

Linux® is a registered trademark of Linus Torvalds.

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

Trusted Firmware for Arm Cortex-A

Stable: 19.11.2020 - 10:48 / Revision: 12.11.2020 - 09:10

A quality version of this page, approved on *19 November 2020*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to

- briefly introduce the FMC 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 needed, how to configure the FMC peripheral.

2 Peripheral overview

The **FMC** peripheral includes two memory controllers:

- The NOR/PSRAM memory controller
- The NAND memory controller

2.1 NOR/PSRAM memory controller (or external bus interface controller)

The **FMC** NOR/PSRAM memory controller is used to interface static memory devices, but it is also used to interface Ethernet devices, LCD devices, ...

The **FMC** NOR/PSRAM controller generates the appropriate signal timings to drive the following types of memories:

- Asynchronous SRAM, FRAM and ROM
  - 8 bits
  - 16 bits
- PSRAM (CellularRAM™)
  - Asynchronous mode
  - Burst mode for synchronous accesses with configurable option to split burst access when crossing boundary page for CRAM 1.5.
  - Multiplexed or non-multiplexed
- NOR Flash memory
  - Asynchronous mode
  - Burst mode for synchronous accesses
  - Multiplexed or non-multiplexed

The **FMC** NOR/PSRAM controller supports a wide range of devices through programmable timings among which:

- Programmable wait states (up to 15)
- Programmable bus turnaround cycles (up to 15)
- Programmable output enable and write enable delays (up to 15)
- Independent read and write timings and protocol to support the widest variety of memories and timings
- Programmable continuous clock output.

The **FMC** NOR/PSRAM controller supports up to four external devices.

2.2 NAND Flash controller

The **FMC** NAND Flash controller is used to interface STM32 MPU with SLC 8-bit or 16-bit NAND Flash memory devices.

The **FMC** NAND Flash controller supports:

- Programmable error correction capability (ECC) using BCH8 code, BCH4 code or Hamming code
- Programmable page size of 2048, 4096 and 8192 bytes
- Programmable memory timings
- Multiple dice per package.

2.3 Features

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

2.4 Security support

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

3 Peripheral usage and associated software

3.1 Boot time

The FMC NAND Flash controller is the boot device that supports serial boot for Flash programming with STM32CubeProgrammer.

3.2 Runtime

3.2.1 Overview

The FMC instance can be allocated to the Arm®Cortex®-A7 non-secure core. The FMC NAND Flash controller can be controlled in Linux® by the MTD framework.

Chapter #Peripheral assignment describes which instance can be assigned to which context.

3.2.2 Software frameworks

Domain	Peripheral	Software frameworks		Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)		
Mass storage	FMC		Linux MTD Framework	

3.2.3 Peripheral configuration

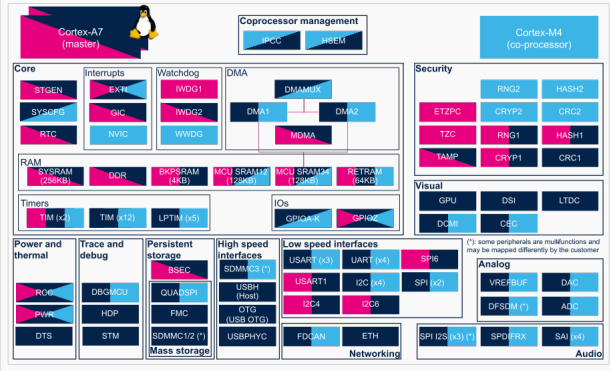
The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals), according to the information given in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

For Linux kernel configuration, please refer to FMC device tree configuration

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)			
Mass storage	FMC	FMC				

4 How to go further

5 References

Read Only Memory

Flash memories combine high density and cost effectiveness of EPROMs with the electrical erasability of EEPROMs. For this reason, the Flash memory market is one of the most exciting areas of the semiconductor industry today and new applications requiring in system reprogramming, such as cellular telephones, automotive engine management systems, hard disk drives, PC BIOS software for Plug & Play, digital TV, set top boxes, fax and other modems, PC cards and multimedia CD-ROMs, offer the prospect of very high volume demand.

Microprocessor Unit

Single-Level Cell is a kind of NAND flash

Elliptic curve cryptography

Error Correction Capability

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.

Open Portable Trusted Execution Environment

Stable: 05.11.2020 - 08:23 / Revision: 21.10.2020 - 11:37

A quality version of this page, approved on *5 November 2020*, was based off this revision.

The Linux®MTD (Memory Technology Device) subsystem provides an abstraction layer for raw Flash memories. It makes it possible to use the same API when working with different Flash types and technologies, e.g. SLC NAND, SPI NOR, ...

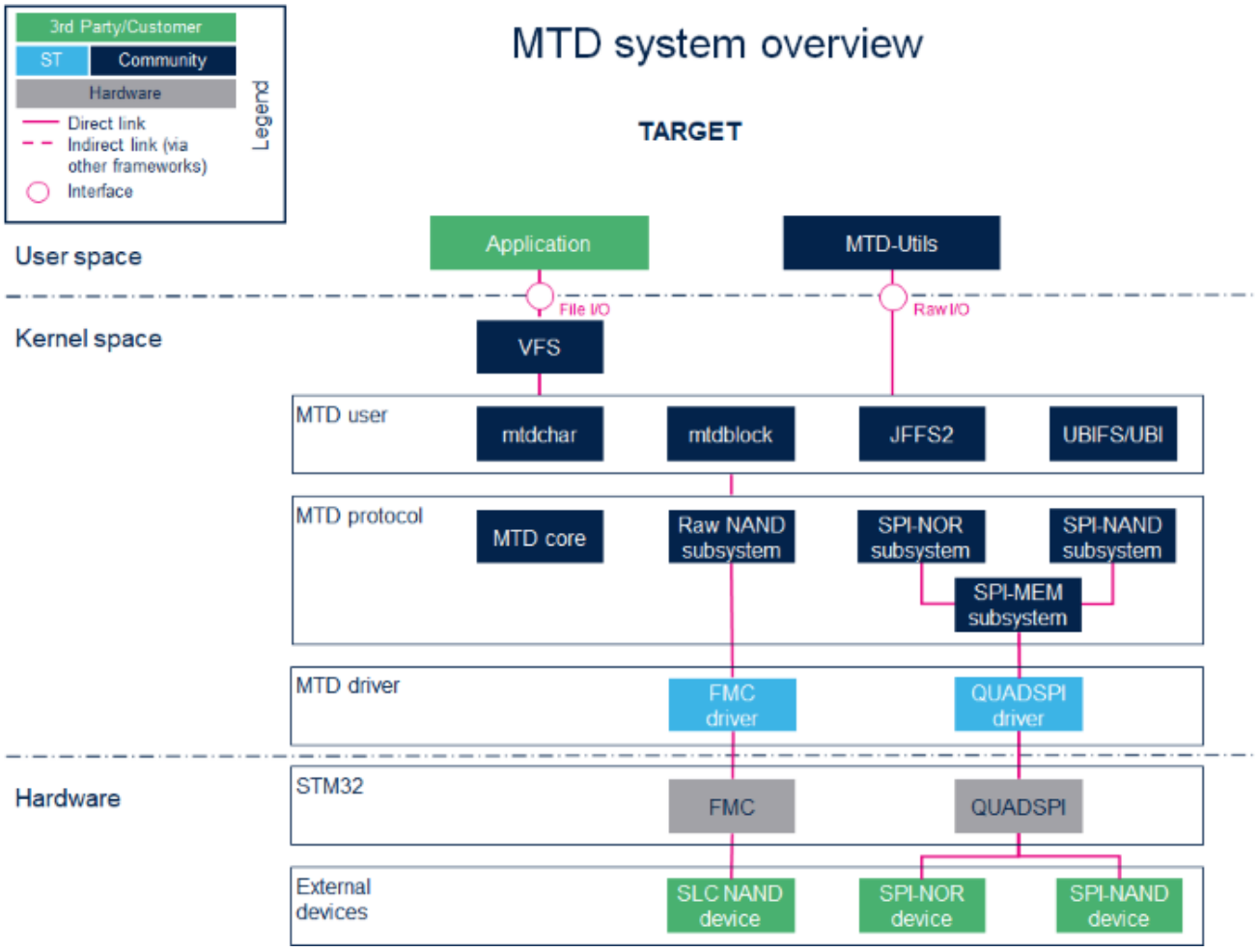
Contents

1 Framework purpose

The purpose of this article is to introduce the MTD Linux subsystem:

- General information
- Main components/stakeholders
- How to use the MTD API

2 System overview



2.1 Component description

- User space applications that perform **file I/O** need to view the Flash memory as if it was a disk, whereas programs that wish to accomplish **raw I/O** access the memory as if it was a character device.
- **VFS** (Kernel space)

Virtual File System. Please refer to the VFS documentation [1].

- **mtdchar** (Kernel space)

Usually referred to as /dev/mtdX. For MTD character devices, please refer to the MTD overview documentation [2].

- **mtdblock** (Kernel space)

Usually referred to as /dev/mtdblockX. Do not use mtdblock unless you know exactly what you are doing. For MTD block devices, please refer to the MTD block documentation [3].

- **JFFS2** (Kernel space)

Journally Flash File System. Please refer to the MTD JFFS2 documentation [4].

- **UBI** (Kernel space)

Unsorted Block Images. Please refer to the MTD UBI documentation [5].

- **UBIFS** (Kernel space)

UBI File System. Please refer to the MTD UBIFS documentation [6].

- **MTD core** (Kernel space)

The MTD core provides an abstraction layer for raw Flash memories.

- **Raw NAND subsystem** (Kernel space)

The Raw NAND protocol is used in the MTD subsystem for interfacing NAND Flash memories.

- **SPI-MEM subsystem** (Kernel space)

The SPI-MEM protocol is used in the MTD subsystem for interfacing all kinds of SPI memories (NORs, NANDs)

- **SPI-NAND subsystem** (Kernel space)

The SPI-NAND protocol is used in the MTD subsystem for interfacing SPI NAND Flash memories.

- **SPI-NOR subsystem** (Kernel space)

The SPI-NOR protocol is used in the MTD subsystem for interfacing SPI NOR Flash memories.

- **FMC driver** (Kernel space) / **FMC** (Hardware)

Please refer to the FMC internal peripheral.

- **QUADSPI driver** (Kernel space) / **QUADSPI** (Hardware)

Please refer to the QUADSPI internal peripheral.

2.2 API description

For the Linux MTD API description, please refer to the MTD API documentation [7].

3 Configuration

3.1 Kernel configuration

MTD is activated by default in ST deliveries. Nevertheless, if a specific configuration is needed, this section indicates how MTD can be activated/deactivated in the kernel.

Activate MTD in the kernel configuration with the Linux Menuconfig tool: Menuconfig or how to configure kernel.

3.1.1 SLC NAND Flash memory

```
[*] Device Drivers --->
    <*> Memory Technology Device (MTD) support --->
        <*> RAW/Parallel NAND Device Support --->
            <*> Support for NAND controller on STM32MP Socs.
```

3.1.2 SPI NOR/NAND Flash memory

```
[*] Device Drivers --->
    <*> Memory Technology Device (MTD) support --->
        Self-contained MTD device drivers --->
            <*> Support most SPI Flash chips (AT26DF, M25P, W25X, ...)
        <*> SPI NAND device Support
        <*> SPI-NOR device support
    <*> SPI support --->
        -*- SPI memory extension
        <*> STMicroelectronics STM32 QUAD SPI controller
```

3.2 Device tree configuration

The DT configuration can be done thanks to STM32CubeMX.

3.2.1 NAND Flash memory

Please refer to the FMC device tree configuration.

3.2.2 SPI NOR/NAND Flash memory

Please refer to the QUADSPI device tree configuration.

4 How to use the framework

A file system, which handles read/write/erase operations, can be used over the MTD Framework. Please refer to the UBIFS support through MTD.

You can also interact with the MTD subsystem using the MTD utilities. The MTD utilities[8] are a set of tools that can be used to perform operations on Flash memories through the MTD character interface.

The most common utilities used are:

- mtdinfo
- flash_erase
- flashcp
- nandwrite
- nanddump

```
root:~# mtdinfo -a
Count of MTD devices:           9
Present MTD devices:            mtd0, mtd1, mtd2, mtd3, mtd4, mtd5, mtd6, mtd7, mtd8
Sysfs interface supported:      yes

mtd0
Name:                           fsbl
Type:                           nand
Eraseblock size:                262144 bytes, 256.0 KiB
Amount of eraseblocks:          8 (2097152 bytes, 2.0 MiB)
Minimum input/output unit size: 4096 bytes
Sub-page size:                  4096 bytes
OOB size:                       224 bytes
Character device major/minor:   90:0
Bad blocks are allowed:         true
Device is writable:             true

mtd1
Name:                           ssbl
Type:                           nand
Eraseblock size:                262144 bytes, 256.0 KiB
Amount of eraseblocks:          8 (2097152 bytes, 2.0 MiB)
Minimum input/output unit size: 4096 bytes
Sub-page size:                  4096 bytes
OOB size:                       224 bytes
Character device major/minor:   90:2
Bad blocks are allowed:         true
Device is writable:             true

mtd2
Name:                           UBI
Type:                           nand
Eraseblock size:                262144 bytes, 256.0 KiB
Amount of eraseblocks:          4078 (1069023232 bytes, 1019.5 MiB)
Minimum input/output unit size: 4096 bytes
Sub-page size:                  4096 bytes
OOB size:                       224 bytes
Character device major/minor:   90:4
Bad blocks are allowed:         true
Device is writable:             true

mtd3
Name:                           fsbl1
```

```
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          4 (262144 bytes, 256.0 KiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:6
Bad blocks are allowed:         false
Device is writable:             true

mtd4
Name:                           fsbl2
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          4 (262144 bytes, 256.0 KiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:8
Bad blocks are allowed:         false
Device is writable:             true

mtd5
Name:                           ssbl
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          32 (2097152 bytes, 2.0 MiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:10
Bad blocks are allowed:         false
Device is writable:             true

mtd6
Name:                           logo
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          4 (262144 bytes, 256.0 KiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:12
Bad blocks are allowed:         false
Device is writable:             true

mtd7
Name:                           nor_user
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          980 (64225280 bytes, 61.2 MiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:14
Bad blocks are allowed:         false
Device is writable:             true

mtd8
Name:                           58003000.qspi
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          1024 (67108864 bytes, 64.0 MiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:16
Bad blocks are allowed:         false
Device is writable:             true
```

5 How to trace and debug the framework

5.1 How to monitor

The sysfs interface provides detail information on each mtd device.

```
root:~# cat /sys/class/mtd/mtd0/name
fsbl
root:~# cat /sys/class/mtd/mtd0/type
nand
root:~# cat /sys/class/mtd/mtd0/erasesize
262144
root:~# cat /sys/class/mtd/mtd0/ecc_strength
8
root:~# cat /sys/class/mtd/mtd0/bad_blocks
0
root:~# cat /sys/class/mtd/mtd0/ecc_failures
0
```

5.2 How to trace

A detail dynamic trace is available here How to use the kernel dynamic debug.

```
root:~# echo "file drivers/mtd/* +p" > /sys/kernel/debug/dynamic_debug/control
```

6 Source code location

The MTD framework is here .

7 To go further

Please refer to the MTD FAQs documentation [9].

8 References

Please refer to the following links for full description:

- VFS
- MTD overview
- MTD block
- MTD JFFS2
- MTD UBI
- MTD UBIFS
- MTD API
- MTD utils
- MTD FAQs

Linux® is a registered trademark of Linus Torvalds.

Memory Technology Device

Application programming interface

Single-Level Cell is a kind of NAND flash

Serial Peripheral Interface

Flash memories combine high density and cost effectiveness of EPROMs with the electrical erasability of EEPROMs. For this reason, the Flash memory market is one of the most exciting areas of the semiconductor industry today and new applications requiring in system reprogramming, such as cellular telephones, automotive engine management systems, hard disk drives, PC BIOS software for Plug & Play, digital TV, set top boxes, fax and other modems, PC cards and multimedia CD-ROMs, offer the prospect of very high volume demand.

Virtual File System

Device Tree

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

Stable: 11.06.2020 - 09:00 / Revision: 10.06.2020 - 15:35

A quality version of this page, approved on *11 June 2020*, was based off this revision.

Contents

1 Purpose

The purpose of this article is to explain how to configure the GPIO internal peripheral through **the pin controller (pinctrl) framework**, **when this peripheral is assigned to Linux®OS (Cortex-A)**. The configuration is performed using the Device tree.

To better understand I/O management, it is recommended to read the Overview of GPIO pins article.

This article also provides an example explaining how to add a new pin in the device tree.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

The Pinctrl device tree bindings are composed of:

- generic DT bindings[1] used by the pinctrl framework.
- vendor pinctrl DT bindings[2] used by the stm32-pinctrl driver: this binding document explains how to write device tree files for pinctrl.

3 DT configuration

3.1 DT configuration (STM32 level)

The pin controller node is located in the pinctrl dtsi file *stm32mp15-pinctrl.dtsi*[3]. See Device tree for more explanations about device tree file split. The pin controller node is composed of three parts:

3.1.1 STM32 pin controller information

```
pinctrl: pin-controller@50002000 {                Comments
         #address-cells = <1>;
         #size-cells = <1>;
         ranges = <0 0x50002000 0xa400>;     -->Provides IP start address and memory map
device size
         interrupt-parent = <&exti>;         -->Provides interrupt parent controller (used
when the GPIO is configured as an external interrupt)
         st,syscfg = <&exti 0x60 0xff>;      -->Provides phandle for IRQ mux selection
         pins-are-numbered;

        ...
};
```

Warning

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

3.1.2 GPIO bank information

```
pinctrl: pin-controller@50002000 {                Comments
         ...

        gpioa: gpio@50002000 {
                gpio-controller;
                #gpio-cells = <2>;
                interrupt-controller;       -->Indicates that this GPIO bank can be used
as interrupt controller
                #interrupt-cells = <2>;
                reg = <0x0 0x400>;          -->Provides offset in pinctrl addres map for
the GPIO bank
                clocks = <&rcc GPIOA>;      -->phandle on GPIO bank clock
                st,bank-name = "GPIOA";
                status = "disabled";
        };

        gpiob: gpio@50003000 {
                gpio-controller;
                #gpio-cells = <2>;
                interrupt-controller;
                #interrupt-cells = <2>;
                reg = <0x1000 0x400>;
                clocks = <&rcc GPIOB>;
                st,bank-name = "GPIOB";
```

```
                status = "disabled";
        };

        ...
};
```

Warning

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

3.1.3 Pin state definition

- Extract of *stm32mp15-pinctrl.dtsi*[3] file:

```
&pinctrl {
        ...
         usart3_pins_a: usart3@0 {                                                  Comme
nts
                pins1 {
                        pinmux = <STM32_PINMUX('B', 10, AF7)>, /* USART3_TX */      --
>Pin muxing information: AF7 (alternate function 7) selected on PB10 pin
                                 <STM32_PINMUX('G', 8, AF8)>; /* USART3_RTS */      --
>Pin muxing information: AF8 (alternate function 8) selected on PG8 pin
                        bias-disable;                                               --
>Generic bindings corresponding to "no pull-up" and "no pull-down"
                        drive-push-pull;                                            --
>Generic bindings to select pin driving information
                        slew-rate = <0>;                                            --
>Generic bindings to select pin speed
                };
                pins2 {
                        pinmux = <STM32_PINMUX('B', 12, AF8)>, /* USART3_RX */
                                 <STM32_PINMUX('I', 10, AF8)>; /* USART3_CTS_NSS */
                        bias-disable;
                };
       };
       ...
};
```

- Refer to GPIO internal peripheral for more details on hardware pin configuration.

3.2 DT configuration (board level)

As seen in Pin controller configuration (pin state definition part), all pin states are defined inside the pin controller node.

Each device that requires pins has to select the desired pin state phandle inside the board device tree file (see Device tree for more explanations about device tree file split).

The STM32MP1 devices feature a lot of possible pin combinations for a given internal peripheral. From one board to another, different sets of pins can consequently be used for an internal peripheral. Note that "_a", "_b" suffixes are used to identify pin muxing combinations in the device tree pinctrl file. The right suffixed combination must then be used in the device tree board file.

- Example:

```
&usart3 {
        ...
        pinctrl-names = "default","sleep";
        pinctrl-0 = <&usart3_pins_a>;
        pinctrl-1 = <&usart3_sleep_pins_a>;
        ...
};
```

3.3 DT configuration examples

3.3.1 How to add new pin states

To add new pin states and affect them to a foo_device, proceed as follows:

1. Find the pins you need:

   In the example below, the foo_device needs to configure PC13, PG8 and PI2.

   AF2 is selected as alternate function on PC13, and AF5 on PG8 and PI2.

   Each pin requires an internal pull-up.

2. Write your pin state phandle in *stm32mp15-pinctrl.dtsi*.

```
 &pinctrl {
         ...
          foo_pins_a: foo@0 {
                 pins {
                         pinmux = <STM32_PINMUX('C', 13, AF2)>,
                                  <STM32_PINMUX('G', 8, AF5)>,
                                   <STM32_PINMUX('I', 2, AF5)>;
                         bias-pull-up;
                 };
        };
        ...
 };
```

   All the possible settings are described in GPIO internal peripheral.

3. Select the pin state phandle required for your device in the board file.

```
&foo {
        ...
        pinctrl-names = "default";
        pinctrl-0 = <&foo_pins_a>;
        ...
};
```

4 How to configure GPIOs 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/pinctrl/pinctrl-bindings.txt , Generic pinctrl device tree bindings
- Documentation/devicetree/bindings/pinctrl/st,stm32-pinctrl.yaml , STM32 pinctrl device tree bindings
- 3.03.1 stm32mp15-pinctrl.dtsi STM32MP15 Pinctrl device tree file

Linux® is a registered trademark of Linus Torvalds.

Operating System

Cortex®

Device Tree

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Transmit

Receive

Compatibility Test Suite (Android specific) or Clear to send (in UART context)

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.

Doubledata rate (memory domain)

Hardware Abstraction Layer

Cortex®

Linux® is a registered trademark of Linus Torvalds.

Trusted Firmware for Arm Cortex-A

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

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Microprocessor Unit