GPIO device tree configuration

Contents

Stable: 12.08.2021 - 15:14 / Revision: 12.08.2021 - 15:13

Stable: 12.08.2021 - 10.14 | Revision: 12.08.2021 - 10.15

A quality version of this page, approved on *12 August 2021*, 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 **GPIOLib framework when this peripheral is assigned to Linux®OS**. The configuration is performed using the device tree[1].

To better understand I/O management, it is recommended to read the I/O pins overview article [2].

This article also provides an example explaining how to add a new GPIO 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 STM32 GPIO bindings are composed of:

- Bindings for GPIO controller [3]
- Bindings for devices using a GPIO [4]

It is recommended to read these reference documents if you have never played with GPIO DT.

3 DT configuration

3.1 DT configuration (STM32 level)

The GPIO controller node (gpio controller) is located in the pin controller node that is declared Inside the SoC dtsi file, *stm32mp151.dtsi*[5]. See Device tree for more explanations about device tree files split.

There is one gpio controller node per GPIO bank.

Refer to Pin controller configuration and to GPIO controller node bindings[3] for explanations on the gpio controller node.

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

3.2 DT configuration (board level)

Generic guidelines for adding a GPIO to a client device can be found in the document "GPIO bindings for board" [4].

- Below an example of basic GPIO usage extracted from "GPIO bindings for board" [4]:

GPIO mappings are defined in the ***consumer device node***, through a property named *<function>-gpios*, where <function> is requested by the Linux driver through gpiod_get().

```
foo_device {
                ...
                led-gpios = <&gpioa 15 GPIO_ACTIVE_HIGH>, /* red */
                            <&gpioa 16 GPIO_ACTIVE_HIGH>, /* green */
                            <&gpioa 17 GPIO_ACTIVE_HIGH>; /* blue */

                power-gpios = <&gpiob 1 GPIO_ACTIVE_LOW>;
        };
```

Where:

- *&gpiox*: phandle on gpio-controller node.
- *15*: line offset in gpio-controller.
- *GPIO_ACTIVE_HIGH*: flag to be used for the GPIO. The list of all available GPIO flags can be found in Linux kernel source code[6].

3.3 DT configuration examples

3.3.1 How to add a GPIO dedicated to a device connected on the board

The example below shows how to add a GPIOB5 (PB5 on schematics) to a foo_device.

```
foo_device {
                ...
                x-gpios = <&gpiob 5 (GPIO_ACTIVE_HIGH | GPIO_PULL_UP)>; /* x function */
                ...
        };
```

Note: GPIO_ACTIVE_HIGH means that the GPIO line is driven by the logical level 1 (while GPIO_ACTIVE_LOW means that it is driven by the logical level 0). GPIO_PULL_UP means that the internal pull-up resistor is enabled.

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:

- Device tree, Device tree mechanism
- Overview of GPIO pins, Overview of GPIO pins article
- 3.03.1 Kernel device tree bindings documentation - gpio.txt, GPIO device tree bindings
- 4.04.14.2 GPIO bindings for board
- stm32mp151.dtsi , STM32MP151 device tree file
- include/dt-bindings/gpio/gpio.h

}}

Linux® is a registered trademark of Linus Torvalds.

Operating System

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.)

Device Tree

uniprocessor

Stable: 19.03.2021 - 08:52 / Revision: 19.03.2021 - 08:49

A quality version of this page, approved on *19 March 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. 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

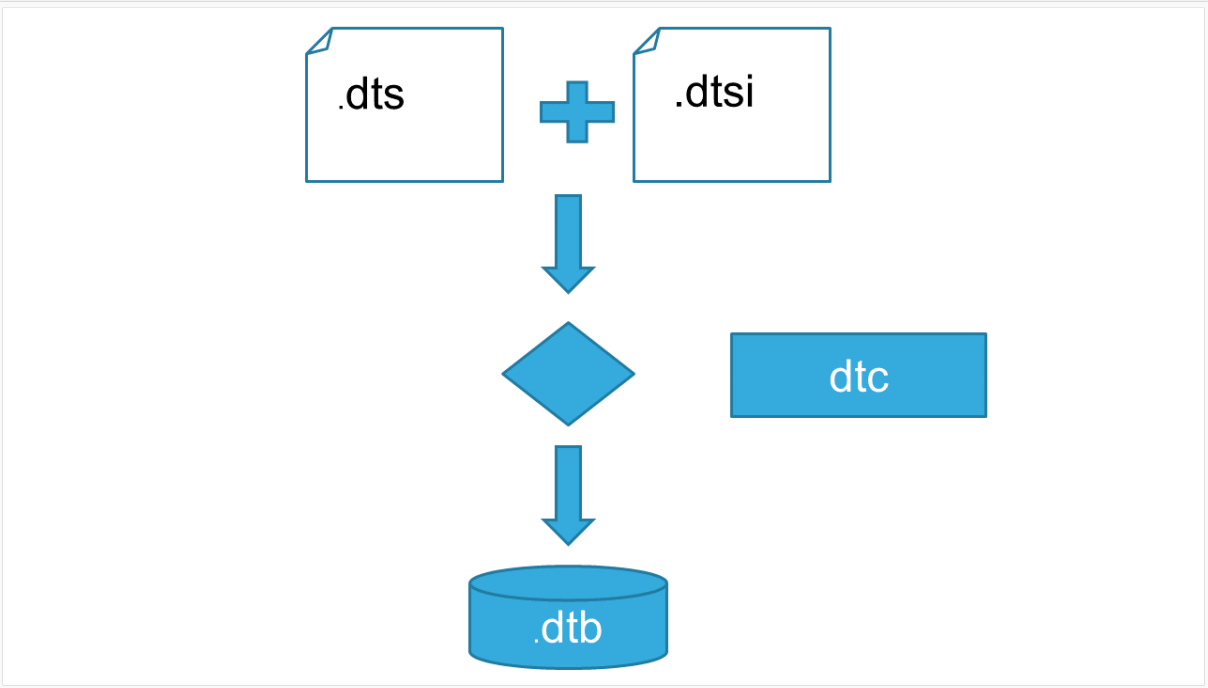
The device tree data structures and properties are named **bindings**. Those bindings are described in:

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

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 STM32MP [5]
- 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
- https://www.youtube.com/watch?v=a9CZ1Uk3OYQ, Device Tree for STM32MP
- 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: 12.08.2021 - 15:30 / Revision: 12.08.2021 - 15:30

A quality version of this page, approved on *12 August 2021*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to

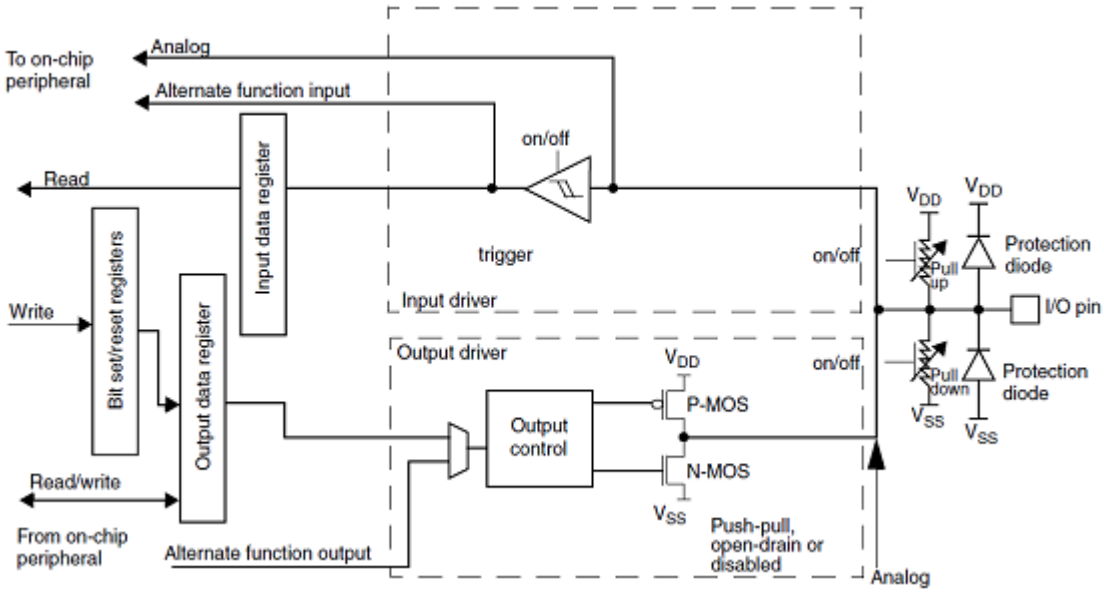
- briefly introduce the GPIO peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain how to configure the GPIO peripheral.

2 Peripheral overview

The **GPIO** peripheral is used to configure the device IO ports, also called pins or pads.
Each GPIO instance controls 8 pins (for GPIOK and GPIOZ) or 16 pins (for GPIOA to GPIOJ).

Every IO port implements the logic shown in the image below, taken from STM32MP15 reference manuals:

- The **IO pin** (on the right) is the physical connection to a chip external ball, soldered on the PCB. The link between each GPIO pin and each ball of the package is given in the datasheet.
- The **Read** and **Write** accesses allow the processor (Arm®Cortex®-A7 or Arm®Cortex®-M4) to configure the peripheral, control the IO pin and get its status.
- **Alternate function** (AF) links allow to connect the IO port to an internal peripheral digital line. In such a case, the IO direction is given by the line purpose: for instance, UART transmit line (TX) is an output.
- **Analog** links allow to connect the IO port to an internal peripheral analog line. In such a case, the IO direction is given by the line purpose: for instance, ADC input line is an input.



Note:

- the pull-up and pull-down resistors are disabled (by hardware) in analog mode.
- at reset, all pins are set in analog input mode to protect the device and minimize the power consumption. All unused pins should be kept in this state.

The pin configuration done by the software consists in:

- setting the **pin mode** in the GPIOx_MODER register:
  - **input** or **output** if the pin is used as general purpose (GPIO), controlled by software.
  - **analog**.
  - **alternate function** (AF).
- selecting the **alternate function** in the GPIOx_AFRH/L register (only when the pin mode is AF):
  - each IO port can support up to 16 alternate functions that are documented in the datasheet.

- setting the **pin characteristics**:
  - **no pull-up and no pull-down** or **pull-up** or **pull-down** in the GPIOx_PUPDR register, needs to be selected to be coherent with the hardware schematics.
  - **push-pull** or **open-drain** in the GPIOx_OTYPER register, needs to be selected to be coherent with the hardware schematics.
  - **output speed** in the GPIOx_OSPEEDR register needs to be tuned to achieve the expected level of performance (rising and falling times) while limiting electromagnetic interferences (EMI) and overconsumption. As example, the table below summarizes the maximum achievable frequency for each supported IO voltage and a 30pF load:

GPIOx_OSPEEDR	Meaning	VDD=3v3	VDD=1v8 HSLV OFF	VDD=1v8 HSLV ON
b00	Low speed	24 MHz	11 MHz	22 MHz
b01	Medium speed	83 MHz	28 MHz	79 MHz
b10	High speed	125 MHz	66 MHz	101 MHz
b11	Very high speed	150 MHz	70 MHz	111 MHz

Note:

- More information is available in the **IO speed settings** chapter of the AN5031[1].
- There are different **IO types** with different characteristics: for instance, all pads are not able to achieve 150 MHz while supplied at 3v3. Refer to the datasheet to get the characteristics for each pin.
- When supplied with VDD=1v8, it is possible to enable the **high speed low voltage** (HSLV) pad mode for FTH (Five volt Tolerant High speed) and FTE (Five volt Tolerant Extended high speed) IO types on some peripherals (SPI, SDMMC, ETH , Dual QUADSPI and TRACE) via SYSCFG_IOCTRLR register in SYSCFG. **Warning**: As it could be destructive if used when VDD>2.7V, the HSLVEN_xx bits set in SYSCFG_IOCTRLR register are not taken account unless the OTP bit PRODUCT_BELOW_2V5 is set.

The table below shows all possible characteristics combinations for each **pin mode**:

pin mode	GPIOx_PUPDR	GPIOx_OTYPER	GPIOx_OSPEEDR
analog	Not applicable	Not applicable	Not applicable
input (GPIO or AF)	**no pull-up and no pull-down** or **pull-down**	Not applicable	Not applicable
output (GPIO or AF) or **bi-directional** (		**push-pull**	cf. the table

pin mode	GPIOx_PUPDR	GPIOx_OTYPER	GPIOx_OSPEEDR
AF)	or **pull-up**	or **open-drain**	above

Note:

- 'Not applicable' means that setting this register has no effect but, in any case, there is no risk for the device.
- On the other hand, leaving a register not initialized whereas it should be, may lead to an unpredictable behavior!

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

The GPIOA to GPIOK peripherals are **non-secure**.
The GPIOZ peripheral is **secure aware**.

3 Peripheral usage and associated software

3.1 Boot time

The STM32CubeMX tool allows to configure in one place the GPIO configuration that is applied at boot time and used at runtime, so it is highly recommended to use it to generate your device tree. Moreover, STM32CubeMX integrates all the information documented in the datasheet, making this configuration step straightforward.

Since a GPIO configuration is done via atomic registers read and write, concurrent accesses from different cores must be avoided and that is why all GPIO configurations are done by the Arm®Cortex®-A7. The strategy is to progressively initialize the GPIO all along the boot chain, as soon as one boot component needs to use them:

- Most of the GPIOs used by the ROM code are directly defined in the ROM code but it is possible to change some pins muxing via dedicated words in BSEC.
- The other boot components are relying on a common binding[2] in the device tree to get the pins configuration:
  - The FSBL configures both secure (GPIOZ) and non-secure (GPIOA to GPIOK) instances.
  - The SSBL and Linux pinctrl configure only non-secure instances, so GPIOA to GPIOK, and only the pins left non-secure in GPIOZ by the FSBL. Linux also initializes the GPIO used by the coprocessor, via its resource manager.

3.2 Runtime

3.2.1 Overview

The **GPIO configuration** must not be done from different cores to avoid concurrent accesses, but this is not the case for the **GPIO using**: each core can manipulate IO on its own since dedicated set/clear registers are available for that.

Nevertheless, beyond the boot time, the GPIO configuration also evolves at runtime: while entering in low power mode, some GPIOs may be put back to analog input mode in order to reduce the power consumption. This is done in two times:

1. the Arm®Cortex®-A7 non-secure takes care of the non-secure GPIO with Linux IOs pins frameworks.
2. the Arm®Cortex®-A7 secure takes care of the secure pins of GPIOZ behind PSCI secure services.

On wakeup, the boot chain restores the GPIO configuration similarly to what is done at boot time.

Let's come back to the runtime allocation...

The GPIOZ pins can individually be:

- set secure and assigned to the Arm®Cortex®-A7 secure for using with OP-TEE

or

- set non-secure and assigned to the Arm®Cortex®-A7 non-secure for using in Linux® with the IOs pins frameworks

or

- set non-secure and assigned to the Arm®Cortex®-M4 for using in STM32Cube with GPIO HAL driver

The GPIOA to GPIOK pins can individually be:

- assigned to the Arm®Cortex®-A7 non-secure for using in Linux® with the IOs pins frameworks

or

- assigned to the Arm®Cortex®-M4 for using in STM32Cube with GPIO HAL driver

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
Core/IOs	GPIO	OP-TEE GPIO driver	Linux IOs pins overview	STM32Cube GPIO 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 by itself can be done via STM32CubeMX tool for all internal peripheral, then it can be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

In Linux kernel, each GPIO bank is declared as a "gpio-controller" in the device tree and each pin can then be used via two different consumer frameworks:

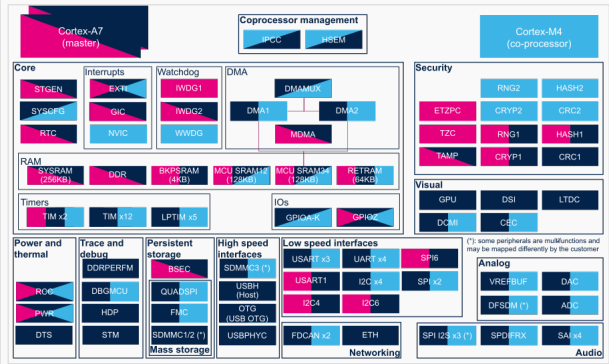
- Pinctrl framework is used to control the alternate function (AF) selection for a given device driver, via the Pinctrl device tree configuration.
- Gpiolib framework is used to control a pin in GPIO mode from another device driver or a user space application: refer to GPIO device tree configuration for further details.

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	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
		GPIOA (16 pins)				Shareable (with pin granularity)

Domain	Peripheral	Runtime allocation				Comment
		GPIOB (16 pins)				Shareable (with pin granularity)
		GPIOC (16 pins)				Shareable (with pin granularity)
		GPIOD (16 pins)				Shareable (with pin granularity)
		GPIOE (16 pins)				Shareable (with pin granularity)
Core/IOs	GPIO	GPIOF (16 pins)				Shareable (with pin granularity)
		GPIOG (16 pins)				Shareable (with pin granularity)
		GPIOH (16 pins)				Shareable (with pin granularity)
		GPIOI (16 pins)				Shareable (with pin granularity)
		GPIOJ (16 pins)				Shareable (with pin granularity)
		GPIOK (8 pins)				Shareable (with pin granularity)
		GPIOZ (8 pins)				Shareable (with pin granularity)

4 How to go further

Not applicable.

5 References

- AN5031
- Documentation/devicetree/bindings/pinctrl/st,stm32-pinctrl.yaml

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.)

input/output

Printed Circuit Board

Cortex®

GPIO alternate function

Transmit

External memory interface

High Speed Low Voltage pin mode

System Configuration

One Time Programmed

Read Only Memory

Open Portable Trusted Execution Environment

Stable: 08.03.2021 - 16:13 / Revision: 16.02.2021 - 17:11

A quality version of this page, approved on *8 March 2021*, was based off this revision.

Contents

1 Article purpose

This article explains how to configure the software that assigns a peripheral to a runtime context.

2 Introduction

A peripheral can be **assigned** to a runtime context via the configuration defined in the device tree. The device tree can be either generated by the STM32CubeMX tool or edited manually.

On STM32MP15 line devices, the assignement can be strengthened by a hardware mechanism: the ETZPC internal peripheral, which is configured by the TF-A boot loader. The ETZPC internal peripheral isolates the peripherals for the Cortex-A7 secure or the Cortex-M4 context. The peripherals assigned to the Cortex-A7 non-secure context are visible from any context, without any isolation.

The components running on the platform after TF-A execution (such as U-Boot, Linux, STM32Cube and OP-TEE) must have a **configuration** that is consistent with the assignment and the isolation configurations.

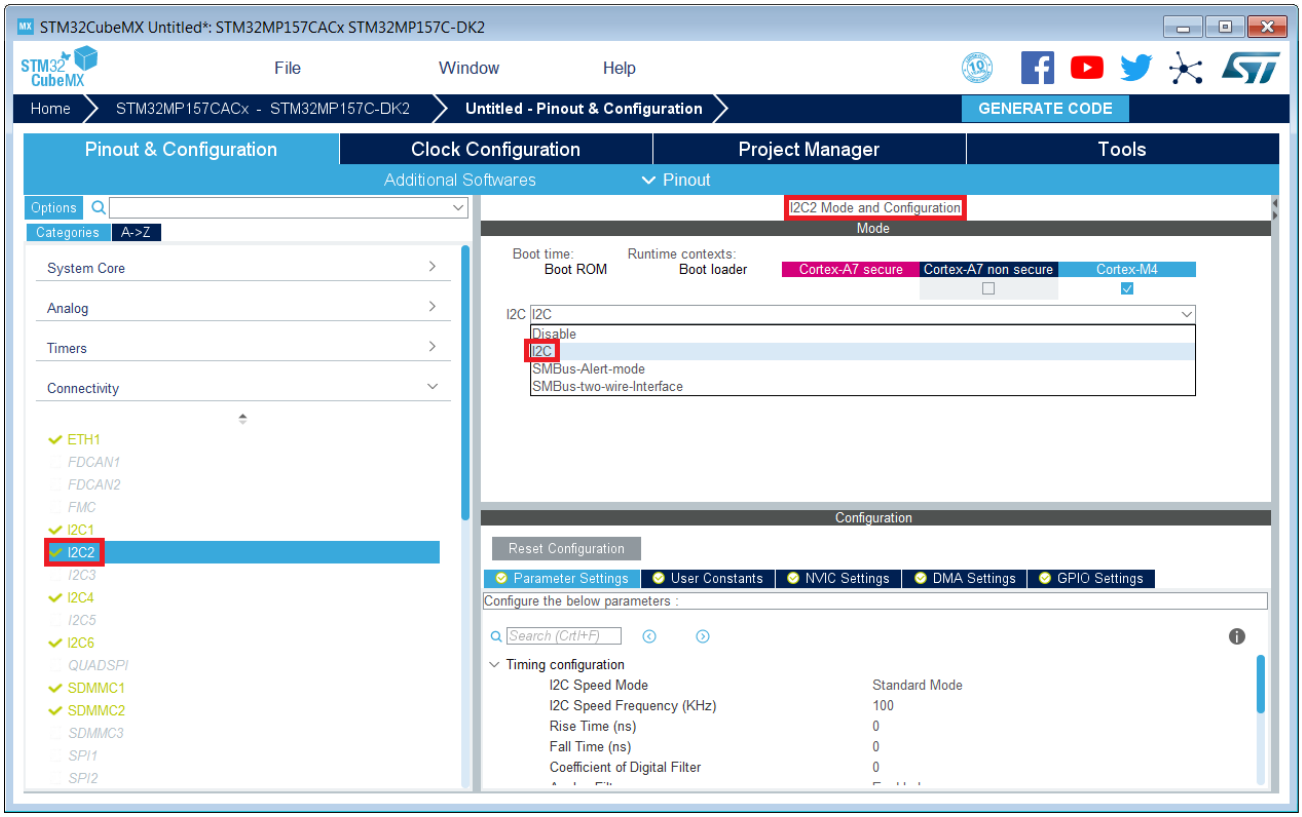
The following sections describe how to configure TF-A, U-Boot, Linux and STM32Cube accordingly.

Beyond the peripherals assignment, explained in this article, it is also important to understand How to configure system resources (i.e clocks, regulator, gpio,...), shared between the Cortex-A7 and Cortex-M4 contexts

3 STM32CubeMX generated assignment

The screenshot below shows the STM32CubeMX user interface:

- **I2C2** peripheral is selected, on the left
- **I2C2 Mode and Configuration** panel, on the right, shows that this I2C instance can be assigned to the Cortex-A7 non-secure or the Cortex-M4 (that is selected) runtime context
- **I2C** mode is enabled in the drop down menu



The context assignment table is displayed inside each peripheral **Mode and Configuration** panel but it is possible to display it for all the peripherals in the **Options** menu via the **Show contexts** option

The **GENERATE CODE** button, on the top right, produces the following:

- The **TF-A device tree** with the ETZPC configuration that isolates the I2C2 instance (in the example) for the Cortex-M4 context. This same device tree can be used by **OP-TEE**, when enabled
- The **U-Boot device tree** widely inherited from the Linux one, just below
- The **Linux kernel device tree** with the I2C node disabled for Linux and enabled for the coprocessor
- The **STM32Cube project** with I2C2 HAL initialization code

The Manual assignment section, just below, illustrates what STM32CubeMX is generating as it follows the same example.

In addition of this generation, the user may have to manually complete the system resources

configuration in the user sections embedded in the STM32CubeMX generated device tree. Refer to How to configure system resources for details.

4 Manual assignment

This section gives step by step instructions, per software components, to manually perform the peripherals assignments.
It takes the same I2C2 example as the previous section, that showed how to use STM32CubeMX, in order to make the move from one approach to the other easier.

> The assignments combinations described in the STM32MP15 peripherals overview article are naturally supported by STM32MPU Embedded Software distribution. Note that the STM32MP15 reference manual may describe more options that would require embedded software adaptations

4.1 TF-A

The assignment follows the ETZPC device tree configuration, with below possible values:

- **DECPROT_S_RW** for the Cortex-A7 secure (Secure OS like OP-TEE)
- **DECPROT_NS_RW** for the Cortex-A7 non-secure (Linux)
  - As stated earlier in this article, there is no hardware isolation for the Cortex-A7 non-secure so this value allows accesses from any context
- **DECPROT_MCU_ISOLATION** for the Cortex-M4 (STM32Cube)

Example:

```
@etzpc: etzpc@5C007000 {
        st,decprot = <
            DECPROT(STM32MP1_ETZPC_I2C2_ID, DECPROT_MCU_ISOLATION, DECPROT_UNLOCK)
        >;
};
```

> The value **DECPROT_NS_RW** can be used with **DECPROT_LOCK** as last parameter. In Cortex-M4 context, this specific configuration allows the generation of an error in the resource manager utility while trying to use on Cortex-M4 side a peripheral that is assigned to the Cortex-A7 non-secure context. If **DECPROT_UNLOCK** is used, then the utility allows the Cortex-M4 to use a peripheral that is assigned to the Cortex-A7 non-secure context.

4.2 U-boot

No specific configuration is needed in U-Boot to configure the access to the peripheral.

> U-Boot does not perform any check with regards to ETZPC configuration before accessing to a peripheral. In case of inconsistency an illegal access is generated.

> U-Boot checks the consistency between ETZPC isolation configuration and Linux kernel device tree configuration to guarantee that Linux kernel do not access an unauthorized device. In order to avoid the access to an unauthorized device, the U-boot fixes up the Linux

kernel device tree to disable the peripheral nodes which are not assigned to the Cortex-A7 non-secure context.

4.3 Linux kernel

Each assignable peripheral is declared twice in the Linux kernel device tree:

- Once in the **soc** node from arch/arm/boot/dts/stm32mp151.dtsi , corresponding to Linux assigned peripherals
  - Example: i2c2
- Once in the **m4_rproc** node from arch/arm/boot/dts/stm32mp15-m4-srm.dtsi , corresponding to the Cortex-M4 context.

Those nodes are disabled, by default.

  - Example: m4_i2c2

In the board device tree file (*.dts), each assignable peripheral has to be enabled only for the context to which it is assigned, in line with TF-A configuration.
As a consequence, a peripheral assigned to the Cortex-A7 secure has both nodes disabled in the Linux device tree.

Example:

```
&i2c2 {
        status = "disabled";
};
...
&m4_i2c2 {
        status = "okay";
};
```

In addition of this assignment, the user may have to complete the system resources configuration in the device tree nodes. Refer to How to configure system resources for details.

4.4 STM32Cube

There is no configuration to do on STM32Cube side regarding the assignment and isolation. Nevertheless, the resource manager utility, relying on ETZPC configuration, can be used to check that the corresponding peripheral is well assigned to the Cortex-M4 before using it.

Example:

```
int main(void)
{
...
 /* Initialize I2C2-------------------------------------------------------- */
 /* Ask the resource manager for the I2C2 resource */
 ResMgr_Init(NULL, NULL);
 if (ResMgr_Request(RESMGR_ID_I2C2, RESMGR_FLAGS_ACCESS_NORMAL | \
                                    RESMGR_FLAGS_CPU1, 0, NULL) != RESMGR_OK)
 {
   Error_Handler();
 }
```

```
...
 if (HAL_I2C_Init(&I2C2) != HAL_OK)
 {
   Error_Handler();
 }
}
```

4.5 OP-TEE

The OP-TEE OS may use STM32MP1 resources. OP-TEE STM32MP1 drivers register the device driver they intend to used in a secure context. This information is used to consolidate system configuration including secure hardening of configurable peripherals.

In most case, the OP-TEE driver probe relies on OP-TEE device tree porperty *secure-status = "okay"*.

Cortex®

Trusted Firmware for Arm® Cortex®-A

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

Linux® is a registered trademark of Linus Torvalds.

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

Open Portable Trusted Execution Environment

Hardware Abstraction Layer

Operating System

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.)

Extended TrustZone Protection Controller

Stable: 14.09.2021 - 07:38 / Revision: 14.09.2021 - 07:37

A quality version of this page, approved on *14 September 2021*, was based off this revision.

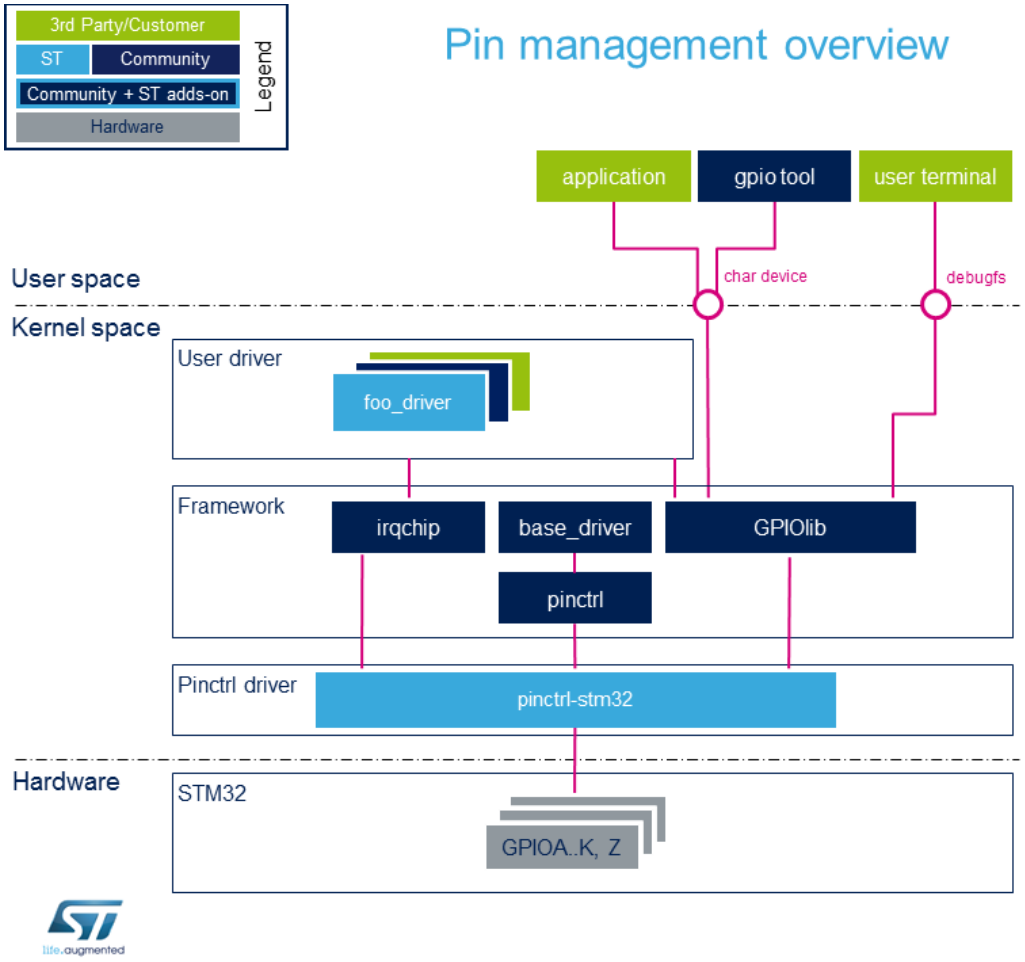
Each STM32 ball/pin is multiplexed in order to support multiple functions. For example, an STM32 pin can operate in three different modes: GPIO, alternate functions or analog. All pin settings are performed via the GPIO internal peripheral, which can be configured through Linux® kernel. This is the purpose of this article.

Contents

1 Purpose

This article introduces the Linux® kernel frameworks dedicated to pin configuration and control. Its purpose is to provide to newcomers some first insights regarding pin management frameworks. Detailed information are provided in the articles referenced in this page.

2 System overview



ST microprocessor pin can be configured in several modes:

- General-purpose input/output (GPIO): controlled by software
- Alternate functions: controlled directly by a hardware block.
- Analog: controlled directly by a hardware block.

Refer to GPIO internal peripheral for more hardware details.

Two frameworks can be used to configure and control a given pin: **pinctrl** and **GPIOLib**. They are selected according to pin usage:

- **pinctrl** is used mainly when a pin is controlled by an internal peripheral. Pinctrl handles the pin configuration and allows assigning a specific feature to a pin. The Pinctrl overview article provides an overview of the pinctrl framework.
- **GPIOLib** is used when a pin needs to be controlled dynamically at runtime (GPIO). GPIOLib is used to control a pin by software. The GPIOLib overview article provides an overview of the GPIOLib framework.

In addition, when a pin is used as external interrupt source, **Irqchip** framework[1] offers an API allowing the configuration of this interrupt.

3 Trainings

Bootlin proposes a detailed presentation of those frameworks: character device interface, *Linux Kernel and Driver Development* training document, see *Introduction to pin muxing* chapter.

4 References

- Interrupt Management training document, *Linux Kernel and Driver Development*

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.)

Linux® is a registered trademark of Linus Torvalds.

Application programming interface

Stable: 06.09.2021 - 16:08 / Revision: 06.09.2021 - 16:03

A quality version of this page, approved on *6 September 2021*, 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 composed of several parts:

3.1.1 STM32 pin controller information

The pin controller node is located in the SOC dtsi file *stm32mp151.dtsi*[3].

```
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;

        ...
};
```

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

The GPIO bank information nodes are located in the SOC dtsi file *stm32mp151.dtsi*[3].

```
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";
        };

        ...
};
```

The GPIO bank definition is completed by the pinctrl package dtsi files, as, for example *stm32mp15xxaa-pinctrl.dtsi*[4].

```
&pinctrl {
        st,package = <STM32MP_PKG_AA>;

        gpioa: gpio@50002000 {
                status = "okay";
                ngpios = <16>;
                gpio-ranges = <&pinctrl 0 0 16>;
        };

        gpiob: gpio@50003000 {
                status = "okay";
                ngpios = <16>;
                gpio-ranges = <&pinctrl 0 16 16>;
        };
```

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

The pin states are defined in the pinctrl dtsi file *stm32mp15-pinctrl.dtsi* [5]

```
&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 stm32mp151.dtsi STM32MP15 SOC device tree file
- stm32mp15xxaa-pinctrl.dtsi STM32MP15 Pinctrl device tree file
- 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