OTG device tree configuration

Contents

Stable: 26.03.2021 - 15:53 / Revision: 19.03.2021 - 13:15

Stable: 26.03.2021 - 10:39 / Revision: 10.03.2021 - 13:13

A quality version of this page, approved on *26 March 2021*, was based off this revision.

Contents

1 Article purpose

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

The configuration is performed using the device tree mechanism.

It is used by *OTG Linux driver*[1] which registers the relevant information in the USB framework.

2 DT bindings documentation

The *Platform DesignWare HS OTG USB 2.0 controller device tree bindings*[2] document represents the OTG (DRD) controller.

The *Generic USB device tree bindings*[3] document represents generic USB properties, proposed by USB framework such as maximum speed, dr_mode...

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)

The **usbotg_hs** DT node is declared in stm32mp151.dtsi[4].

It is composed of a set of properties, used to describe the OTG controller: registers address, clocks, resets, interrupts...

```
usbotg_hs: usb-otg@49000000 {
        compatible = "snps,dwc2";
        reg = <0x49000000 0x10000>;
        clocks = <&rcc USBO_K>;
        clock-names = "otg";
        resets = <&rcc USBO_R>;
        reset-names = "dwc2";
        interrupts = <GIC_SPI 98 IRQ_TYPE_LEVEL_HIGH>;
        dr_mode = "otg";                                    /* dr_mode[3] can be
overwritten at board level to set a particular mode */
        status = "disabled";
};
```

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)

Follow the sequences described in the below chapters to configure and enable the OTG on your board.

OTG supports two PHY interfaces that can be statically selected via DT:

- full-speed PHY, embedded with the OTG controller
- high-speed USBPHYC HS PHY that can be assigned to either the USBH or the OTG controller.

When operating in "otg" or "host" mode, an external charge pump, e.g. 5V regulator must be specified.

Please refer to Regulator overview for additional information on regulators configuration.

3.2.1 DT configuration using full-speed USB PHY

- Enable the OTG by setting **status = "okay".**
- Use embedded full-speed PHY by setting **compatible = "st,stm32mp1-fsotg", "snps,dwc2"**
- Configure full-speed PHY pins (OTG ID, DM, DP) as analog via pinctrl, through **pinctrl-0** and **pinctrl-names**.

- Optionally set dual-role mode through **dr_mode = "peripheral"**, **"host"** or **"otg"** (default to "otg" in case it is not set)
- Optionally set vbus voltage regulator for otg and host modes, through **vbus-supply = <&your_regulator>**

3.2.2 DT configuration using high-speed USB PHY

- Enable the OTG by setting **status = "okay".**
- Select USBPHYC port#2 by setting **phys = <&usbphyc_port1 0>;** and **phy-names = "usb2-phy";**
- Optionally configure OTG ID pin as analog via pinctrl, through **pinctrl-0** and **pinctrl-names**.
- Optionally set dual-role mode through **dr_mode = "peripheral"**, **"host"** or **"otg"** (default to "otg" in case it is not set)
- Optionally set vbus voltage regulator for otg and host modes, through **vbus-supply = <&your_regulator>**

Please refer to USBPHYC device tree configuration for additional information on USBPHYC port#2 configuration

3.3 DT configuration examples

3.3.1 DT configuration example as full-speed OTG, with micro-B connector

The example below shows how to configure full-speed OTG, with the ID pin to detect role (peripheral, host):

- OTG ID and data (DM, DP) pins: use Pinctrl device tree configuration to configure PA10, PA11 and PA12 as analog input.
- Use integrated *full-speed USB PHY* by setting **compatible**
- Dual-role (dr_mode) is "otg" (e.g. the default as unspecified)
- Use vbus voltage regulator

```
# part of pin-controller dt node
usbotg_hs_pins_a: usbotg_hs-0 {
        pins {
                pinmux = <STM32_PINMUX('A', 10, ANALOG)>; /* OTG_ID */            /*
configure 'PA10' as ANALOG */

        };
};

usbotg_fs_dp_dm_pins_a: usbotg-fs-dp-dm-0 {
        pins {
                pinmux = <STM32_PINMUX('A', 11, ANALOG)>, /* OTG_FS_DM */
                         <STM32_PINMUX('A', 12, ANALOG)>; /* OTG_FS_DP */
        };
};
```

```
&usbotg_hs {
        compatible = "st,stm32mp1-fsotg", "snps,dwc2";                          /* Use
full-speed integrated PHY */
        pinctrl-names = "default";
        pinctrl-0 = <&usbotg_hs_pins_a &usbotg_fs_dp_dm_pins_a>;                /*
configure OTG ID and full-speed data pins */
        vbus-supply = <&vbus_otg>;                                              /*
voltage regulator to supply Vbus */
        status = "okay";
};
```

3.3.2 DT configuration example as high speed OTG, with micro-B connector

The example below shows how to configure high-speed OTG, with the ID pin to detect role (peripheral, host):

- OTG ID pin: use Pinctrl device tree configuration to configure PA10 as analog input.
- Use *USB HS PHY port#2*, with the UTMI switch that selects the OTG controller
- Dual-role mode (dr_mode) is "otg" (e.g. the default as unspecified)
- Use vbus voltage regulator

```
# part of pin-controller dt node
usbotg_hs_pins_a: usbotg_hs-0 {
        pins {
                pinmux = <STM32_PINMUX('A', 10, ANALOG)>; /* OTG_ID */            /*
configure 'PA10' as ANALOG */

        };
};
```

```
&usbotg_hs {
        compatible = "st,stm32mp1-hsotg", "snps,dwc2";
        pinctrl-names = "default";
        pinctrl-0 = <&usbotg_hs_pins_a>;                                          /*
configure OTG_ID pin */
        phys = <&usbphyc_port1 0>;                                                /* 0: UT
MI switch selects the OTG controller */
        phy-names = "usb2-phy";
        vbus-supply = <&vbus_otg>;                                                /*
voltage regulator to supply Vbus */
        status = "okay";                                                          /*
enable OTG */
};
```

3.3.3 DT configuration example as high speed OTG, with Type-C connector

The example below shows how to configure high-speed OTG with a Type-C connector. Type-C is managed by an external controller which detects connection and data role (peripheral, host) and implements Linux USB role switch class:

- Use *USB HS PHY port#2*, with the UTMI switch that selects the OTG controller
- Dual-role mode (dr_mode) is "otg" (e.g. the default as unspecified)
- Add usb-role-switch property to OTG controller node: it indicates that the device is capable of assigning the USB data role (USB host or USB device) for a given USB connector.
- Add a connector subnode to the Type-C controller node, with a port child node pointing to the OTG controller endpoint and add a port child node to the OTG controller node, pointing to the Type-C controller endpoint: Type-C controller driver will be able to get the USB role switch to inform it of a role change.

```
#example of Type-C controller node
stusb1600@28 {
        compatible = "st,stusb1600";
        reg = <0x28>;
        interrupts = <11 IRQ_TYPE_EDGE_FALLING>;
        interrupt-parent = <&gpioi>;
        pinctrl-names = "default";
        pinctrl-0 = <&stusb1600_pins_a>;
        status = "okay";
        vdd-supply = <&vin>;
```

```
        connector {
                compatible = "usb-c-connector";
                label = "USB-C";
                power-role = "dual";
                power-opmode = "default";
```

```
                port {
                        con_usbotg_hs_ep: endpoint {
                                remote-endpoint = <&usbotg_hs_ep>;                    /*
point the OTG controller endpoint node */
                        };
                };
        };
};
```

```
&usbotg_hs {
        compatible = "st,stm32mp1-hsotg", "snps,dwc2";
        phys = <&usbphyc_port1 0>;                                                    /* 0: UT
MI switch selects the OTG controller */
        phy-names = "usb2-phy";
        usb-role-switch;                                                              /* see
USB generic bindings [5] */
        status = "okay";                                                              /*
enable OTG */

        port {
                usbotg_hs_ep: endpoint {
                        remote-endpoint = <&con_usbotg_hs_ep>;                      /*
point the Type-C controller endpoint node */
                };
        };
};
```

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:

- drivers/usb/dwc2/ , DesignWare HS OTG Controller driver
- Documentation/devicetree/bindings/usb/dwc2.yaml , Platform DesignWare HS OTG USB 2.0 controller device tree bindings
- 3.03.1 Documentation/devicetree/bindings/usb/generic.txt , Generic USB device tree bindings
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file
- Documentation/devicetree/bindings/usb/generic.txt , USB generic bindings

Operating System

USB On-The-Go (Capability/type of USB port, acting primarily as USB device, to also act as USB host. Also known as USB OTG.)

Device Tree

High Speed (MIPI® Alliance DSI standard)

Dual-Role Device (USB standard defines host and device roles. OTG controllers support both roles and can be called Dual-Role Devices controllers.)

Generic Interrupt Controller

Serial Peripheral Interface

USB 2.0 Transceiver Macrocell Interface

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

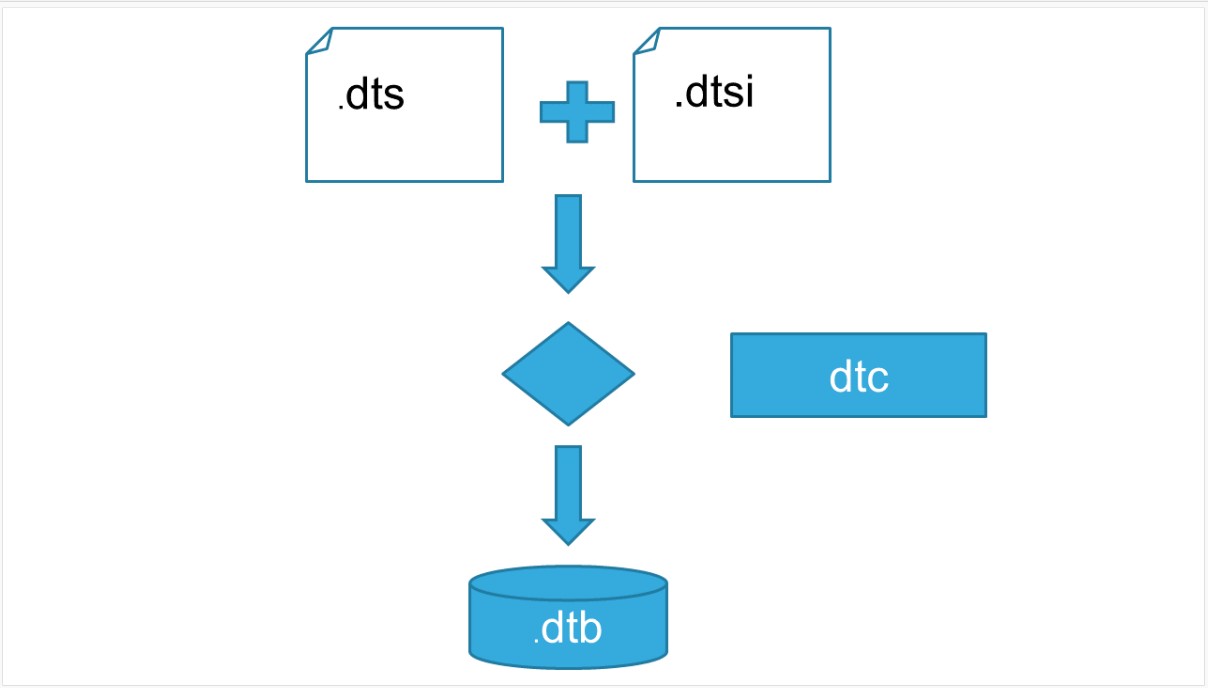
The device tree data structures and properties are named **bindings**. Those bindings are described in:

- 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: 26.03.2021 - 15:54 / Revision: 18.03.2021 - 14:35

A quality version of this page, approved on *26 March 2021*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to

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

2 Peripheral overview

The **OTG** peripheral is used to interconnect other systems with STM32 MPU devices, using USB standard.

2.1 Features

The **OTG** peripheral is a USB Dual-Role Device (DRD) controller that supports both device and host functions.
In Host mode, it supports high-speed (480 Mbit/s), full-speed (12 Mbit/s) and low-speed (1.5 Mbit/s).
In Peripheral mode, high-speed and full-speed are supported, not low-speed.
The **OTG** peripheral embeds a full-speed PHY and supports a UTMI interface connected to internal HS PHY.

The **OTG** peripheral is fully compliant with

- *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification*[1], Revision 2.0, May 8, 2009
- *Universal Serial Bus Revision 2.0 Specification*[2], Revision 2.0, April 27, 2000
- *USB 2.0 Link Power Management Addendum Engineering Change Notice to the USB 2.0 specification*[3], July 16, 2007
- *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*[4], Version 1.05, March 29, 2001
- *UTMI+ Specification*[5], Revision 1.0, February 25, 2004

Refer to STM32MP15 reference manuals for the complete hardware feature list, and to the software components (introduced below) to know which features are supported.

2.2 Security support

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

3 Peripheral usage and associated software

3.1 Boot time

The OTG peripheral is used by ROM code, FSBL and SSBL in device mode (DFU) to support serial boot for flash programming with STM32CubeProgrammer.
The SSBL can use it in host mode (mass storage), for instance to boot on a kernel stored on a USB key, or after a kernel panic to perform the crash dump saving to the USB key.

3.2 Runtime

3.2.1 Overview

The OTG peripheral can be allocated to the Arm®Cortex®-A7 non-secure core to be used under Linux® with USB framework.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
High speed interface	OTG (USB OTG)		Linux USB 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 by itself can be performed via the STM32CubeMX tool for all internal peripherals. It can then be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

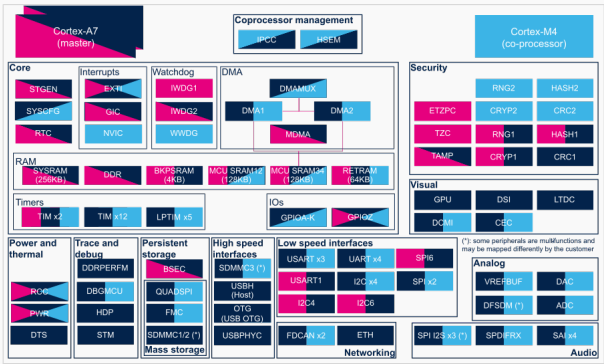
For **Linux** kernel configuration, please refer to OTG device tree configuration.
For **U-boot** configuration, please refer to Configure USB OTG node in U-Boot.

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)			
High speed interface	OTG (USB OTG)	OTG (USB OTG)				

4 References

- On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification
- Universal Serial Bus Revision 2.0 Specification
- ECN USB 2.0 Link Power Management Addendum
- USB 2.0 Transceiver Macrocell Interface (UTMI) Specification
- UTMI+ Specification

USB On-The-Go (Capability/type of USB port, acting primarily as USB device, to also act as USB host. Also known as USB OTG.)

Microprocessor Unit

Dual-Role Device (USB standard defines host and device roles. OTG controllers support both roles and can be called Dual-Role Devices controllers.)

USB 2.0 Transceiver Macrocell Interface

Device Firmware Upgrade

Cortex®

Linux® is a registered trademark of Linus Torvalds.

Open Portable Trusted Execution Environment

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

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: 11.06.2020 - 09:03 / Revision: 10.06.2020 - 15:17

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

This article explains how the Linux®**pinctrl** framework manages IOs/pins, how to configure it, and how to use it.

Contents

1 Framework purpose

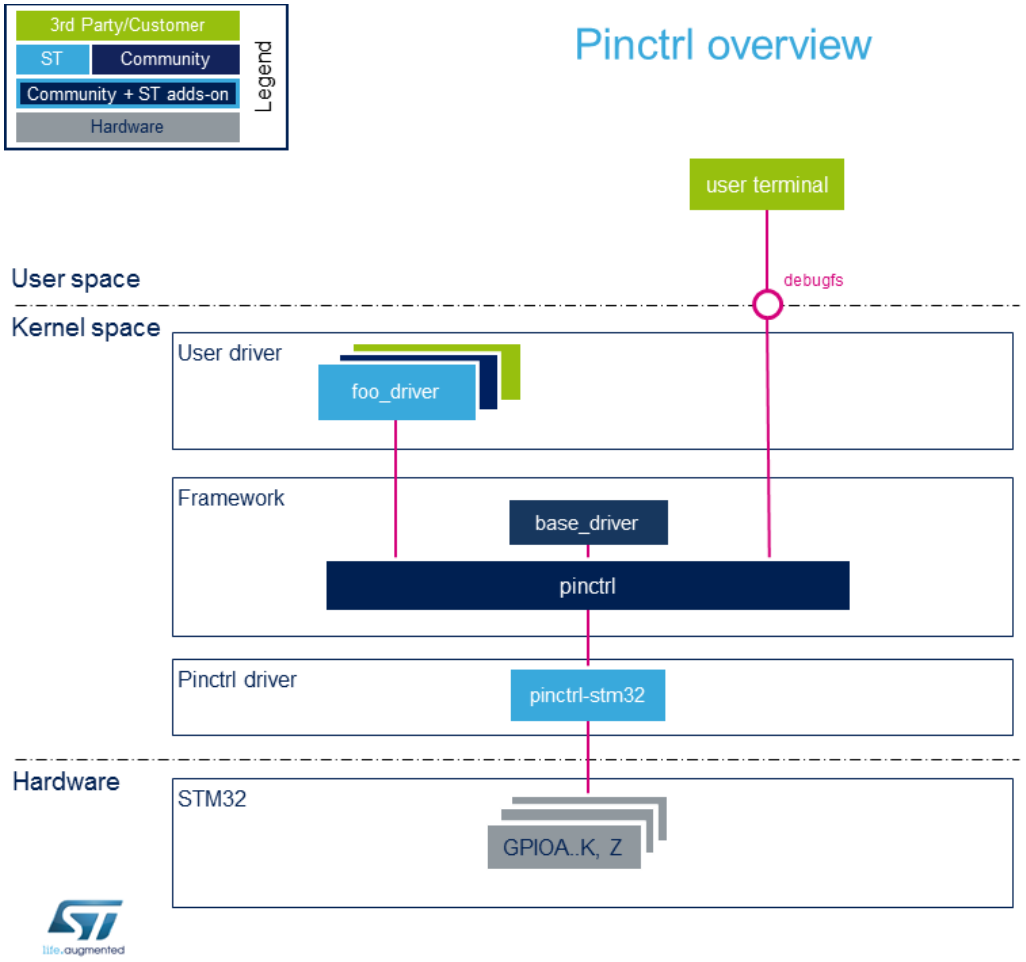
Many of the microprocessor pins (with digital I/O or analog pin type) are multiplexed between different functions: GPIO, alternate function(s).
Pinctrl framework is used to:

- Configure pin hardware settings: multiplexing, pull-up/pull-down, open-drain ...
- Provide information through debugfs

Pinctrl framework is the Linux framework to configure and control the microprocessor pins. There are 2 ways to use it:

- A pin (or group of pins) is controlled by a hardware block, then pinctrl will apply the pin configuration given by the device tree (it just applies devicetree configuration)
- A pin needs to be controlled by software (typically a GPIO), then GPIOLib framework will be used to control this pin on top of pinctrl framework. Refer to GPIOLib overview.

2 System overview



2.1 Component description

- **Pinctrl:** the pinctrl framework **core**, its role is to:
  - provide API to other drivers
  - call specific vendor callback for pin configuration (muxing end setting)
  - create logical pin mapping and guarantee pin exclusivity for a device.
- **Pinctrl-stm32:** microprocessor **specific** pinctrl driver, its role is to:
  - register vendor specific functions (callback) to pinctrl framework
  - access to hardware registers to configure pins (muxing and all pins capabilities)
  - provide other services described in GPIOLib overview.
- **Base driver:** generic kernel driver in charge of getting pin information through the device tree for a device and to register those pins to the pinctrl framework.
- **Foo_driver:**
  - Foo_driver could be any driver that needs specific pins configuration. Note that "default" pins configuration is managed by the kernel base before foo_driver probe. No action is needed by the foo driver.

- this configuration is described in the device tree file. See Pinctrl device tree configuration.

- **debugfs:**
  - provides debug interface available through user terminal, including pin configurations, muxing... See How_to_monitor_with_debugfs.

2.2 API description

- **Kernel space API**: Pinctrl API provides API interface to user driver.
  - Main useful API functions are:

    -*devm_pinctrl_get()*: call to get all pinctrl information.

    -*pinctrl_lookup_state()*: call to obtain a pinctrl state struct from a name.

    -*pinctrl_select_state()*: call to select a pinctrl state struct. After a call to this function, pins are configured.
  - Possible standard state names are: **"default"**, **"init"**, **"sleep"** and **"idle"**.
  - Pinctrl API functions to control those standard states are:

    -*pinctrl_pm_select_sleep_state*: call to select **"sleep"** state defined in device tree.

    -*pinctrl_pm_select_idle_state*: call to select **"idle"** state defined in device tree.

    -*pinctrl_pm_select_default_state*: call to select **"default"** state in device tree
  - See pinctrl kernel documentation[1] for more API function descriptions.

- **debugfs:**

  See How_to_monitor_with_debugfs

3 Configuration

3.1 Kernel configuration

Pinctrl framework and driver are enabled by default.

3.2 Device tree configuration

Refer to Pinctrl device tree configuration.

4 How to use the framework

For a device, there are two ways to use pinctrl framework:

- standard pinctrl utilization
- custom (+standard) pinctrl utilization

4.1 Standard

- To simplify kernel development and avoid code duplication, Linux kernel is in charge to call pinctrl framework to apply pin states (pins configuration). It is possible when standard entries are used in device tree for "pinctrl-names". Possible standard names are: **"default"**, **"init"**, **"sleep"** and **"idle"**.

- **Device tree part:** when using this approach, since Kernel base driver calls pinctrl framework, the user has to write device tree configuration. It means:

  - **Write pin states**: pin states nodes are defined inside the pin controller device node that contains several information about pin configuration. It can be for one pin or a group of pins. This information is not generic and depends on each pin controller driver. See Pinctrl device tree configuration for details.

```
pincontroller {
        foo_state_pins_a {
                "pins configuration: muxing, pull-up/pull-down, ..."
        };

        foo_state_pins_b {
                "pins configuration: muxing, pull-up/pull-down, ..."
        };

        foo_state_pins_sleep_b {
                "pins configuration: muxing, pull-up/pull-down, ..."
        };
}
```

  - **Invoke pin configuration inside user device node**.

```
foo_device {                                    comments
       ...
        pinctrl-names = "default";               -->Standard name known by Linux Kernel.
It's mapped on pinctrl-0 state.
        pinctrl-0 = <&foo_state_pins_a>;         -->Phandle to a pin state node(see above).
       ...
};
```

If needed two pin nodes *foo_state_pins_a* and *foo_state_pins_b* can be used for a same state:

```
foo_device {
        ...
        pinctrl-names = "default";
        pinctrl-0 = <&foo_state_pins_a &foo_state_pins_b>;
        ...
};
```

Two differents states **"default"** and **"sleep"** can also be defined. First name **"default"** is mapped to the first state "pinctrl-0", second name **"sleep"** is mapped to the second state "pinctrl-1":

```
foo_device {
        ...
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&foo_state_pins_a>;
        pinctrl-1 = <&foo_state_pins_sleep_a>;
        ...
};
```

- **Base driver part**[2]

The base driver is in charge to **register** pin states to devices that use standard names as **"default"**, **"idle"**, **"sleep"**, **"init"**. This driver is in charge to **select** **"default"** and **"init"** state:

-If **"default"** state is defined in device tree, this state is selected before the driver probe.

-If **"init"** and **"default"** state are defined, the **"init"** state is selected before the driver probe and the **"default"** state is selected after the driver probe. It is mainly used to avoid glitches.

- **Foo driver part**

As explain above the base driver is in charge to select **"default"** and **"init"** states at probe time. To select **"idle"** and **"sleep"** states, the foo driver has to call pinctrl framework API:

"sleep" and **"idle"** states are mainly used for power management. Indeed to reduce leakage and power consumption, pin settings are changed when the device is not in use. In this case *pinctrl_pm_select_sleep_state* and *pinctrl_pm_select_idle_state* functions can be used. When the device is used again, **"default"** state has to be restored, then *pinctrl_pm_select_default_state* is used.

4.2 Custom

- Sometimes, using standard pin states (managed by base driver and not by concerned foo_driver) is not enough. Foo_driver may need to control pin states at runtime. In such a case it will be up to foo_driver to call framework API.
- The custom pinctrl usage may cohabit with the standard usage explained in previous section.
- Extracted from documentation[1], here is an example on how to use 2 different configurations inside a device driver:
  - **device tree part**

    - **Write pin states**: pin states nodes are defined inside the pin controller device node that contains several information about pin configuration. It can be for one pin or a group of pins. This information is not generic and depends on each pin controller driver. See Pinctrl device tree configuration for details.

```
pincontroller {
        foo_state_pins_a {
                "pins configuration: muxing, pull-up/pull-down, ..."
        };

        foo_state_pins_b {
                "pins configuration: muxing, pull-up/pull-down, ..."
        };
}
```

-Invoke pin configuration inside user device node.

```
       foo_device {
                pinctrl-names = "state-A", "state-B";
                pinctrl-0 = <&state_pins_A>;
                pinctrl-1 = <&state_pins_B>;
        };
```

- foo driver part

  - Initialization part:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;
struct pinctrl_state *s1, *s2;

foo_probe()
{
        /* Setup */
        p = devm_pinctrl_get(&device);
        if (IS_ERR(p))
                ...

        s1 = pinctrl_lookup_state(foo->p, "state-A");
        if (IS_ERR(s1))
                ...

        s2 = pinctrl_lookup_state(foo->p, "state-B");
        if (IS_ERR(s2))
                ...
}
```

- Runtime usage: each state can be selected at runtime.

```
foo_switch()
{
        /* Select pinctrl state A */
        ret = pinctrl_select_state(s1);
        if (ret < 0)
            ...

        ...

        /* select pinctrl state B */
```

```
        ret = pinctrl_select_state(s2);
        if (ret < 0)
            ...

        ...
}
```

- See mmci driver example for a real use case (search for "pinctrl_select_state").

5 How to trace and debug the framework

5.1 How to monitor

5.1.1 How to monitor with debugfs

Some information about pin controller / pins states / pins configurations is available in Debugfs interface. There are two levels of information:

1 Generic information:

```
Board $> ls -l /sys/kernel/debug/pinctrl/
    |
    +---pinctrl-devices      | List of pin ctroller devices.
    +---pinctrl-handles      | List of all pin states registered.
    +---pinctrl-maps         | List of all pin states registered per pin used.
    +---soc:pin-controller-z@54004000    | Folder which contains pins information for a
pin controller.
    +---soc:pin-controller@50002000    | Folder which contains pins information for a pin
controller.
```

2 Pin controller information:

```
Board $> ls -l /sys/kernel/debug/pinctrl/soc:pin-controller@50002000
    |
    +---gpio-ranges          | Provides mapping between logical address space and pins
address space for GPIOs.
    +---pinconf-config       | Provides modified pins at runtime.  Not supported.
    +---pinconf-groups       | Provides pin config settings per pin group.
    +---pinconf-pins         | Provides all pins settings. It reflects the hardware
values.
    +---pingroups            | Provides registered pin groups.
    +---pinmux-functions     | Provides all possibles muxing available.
    +---pinmux-pins          | Provides a list for each pin the muxing selected and the
device which use the pin.
    +---pins                 | Provides list of all pins.
```

5.2 How to trace

- The following extract of kernel log shows that pin controller is well probed:

```
[    0.353613] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOA bank added
[    0.360539] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOB bank added
[    0.367344] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOC bank added
[    0.374199] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOD bank added
[    0.381016] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOE bank added
[    0.387850] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOF bank added
[    0.394625] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOG bank added
[    0.401463] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOH bank added
[    0.408257] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOI bank added
[    0.415098] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOJ bank added
[    0.421889] stm32mp157-pinctrl soc:pin-controller@50002000: GPIOK bank added
[    0.428444] stm32mp157-pinctrl soc:pin-controller@50002000: Pinctrl STM32 initialized
[    0.436604] stm32mp157-pinctrl soc:pin-controller-z@54004000: GPIOZ bank added
[    0.443222] stm32mp157-pinctrl soc:pin-controller-z@54004000: Pinctrl STM32 initialized
```

By default there is no indication in the log that the pin default state has been correctly applied to the device by the base driver. If an issue occurs (like a conflict) the device probe will fail with an error.

- If more kernel logs are needed, use pinctrl dynamic debug:

```
Board $> dmesg -n8
Board $> echo "file drivers/pinctrl* +p" > /sys/kernel/debug/dynamic_debug/control
```

- Since main pin states are applied when devices are probed (meaning before userland prompt) the dynamic printk may need to be enabled in command line:

```
root=/dev/mmcblk0p5 rootwait rw earlyprintk console=ttyS3,115200 loglevel=8 dyndbg="file drivers/pinctrl/* +p"
```

5.3 How to debug

Our pin controller is configured in *strict mode* (meaning that a pin can be requested by only one device). So if a device cannot request a pin during kernel boot, the device tree should be controlled to check if the pin is not affected to two different devices.

Another kind of problem may be that a pin configuration does not fit with the design. In this case, first check the pinconf-pins file in debugfs to verify that the pin hardware settings correspond to the settings defined in the device tree for the same pins. If everything matches, compare the settings with the board schematic in search for missing or unaligned settings, in particular regarding pull-up/pull-down/open-drain ... See GPIO internal peripheral article.

6 Source code location

Source files are located inside kernel Linux.

- **Pinctrl core part**: generic core[3], generic pinconf[4] and generic pinmux[5]
- **STM32 pinctrl vendor part**: folder to STM32 dedicated pinctrl functions[6]
- **base driver part**[2]

7 To go further

7.1 Configure pins for a new board

To configure a new board, two scenarios are possible:

- Pins/groups for device/internal peripherals are already defined: in this case, you only have to select the right group for your device according to schematics.
- Pins/groups for device/internal peripherals are NOT already defined: In this case, you have to define your pins/groups settings inside pincontroller and to select it in your device node according to schematics.
- Please refer to Pinctrl device tree configuration example

Or you can use STM32CubeMX to select your pins and to generate the devicetree accordingly.

7.2 Trainings

More details about pinctrl framework [7]

8 References

- 1.01.1 Documentation/driver-api/pinctl.rst Pinctrl documentation
- 2.02.1 drivers/base/pinctrl.c Pinctrl base driver source
- Pinctrl framework source - core.c Sources of generic pinctrl framework
- Pinctrl framework source - pinconf.c Sources of generic pin configuration
- Pinctrl framework source - pinmux.c Sources of generic pin muxing
- STM32 vendor specific folder Provides all vendor specifics functions
- character device interface, *Linux Kernel and Driver Development* training document, see Introduction to pin muxing **chapter**

Linux® is a registered trademark of Linus Torvalds.

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.)

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

Application programming interface

foo_driver could be any driver that needs to control a GPIO

Stable: 11.06.2020 - 12:50 / Revision: 11.06.2020 - 12:12

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

This article gives information about the Linux® regulator framework.

Contents

1 Article Purpose

This article aims to explain how to use regulators:

- how to configure a regulator on a Linux BSP
- how to access a regulator from a kernel space

This article is applicable for the Linux kernel version 4.10 and later.

2 System overview

Some documentation on the Linux regulator framework is provided with the kernel source code:overview.rst [1]

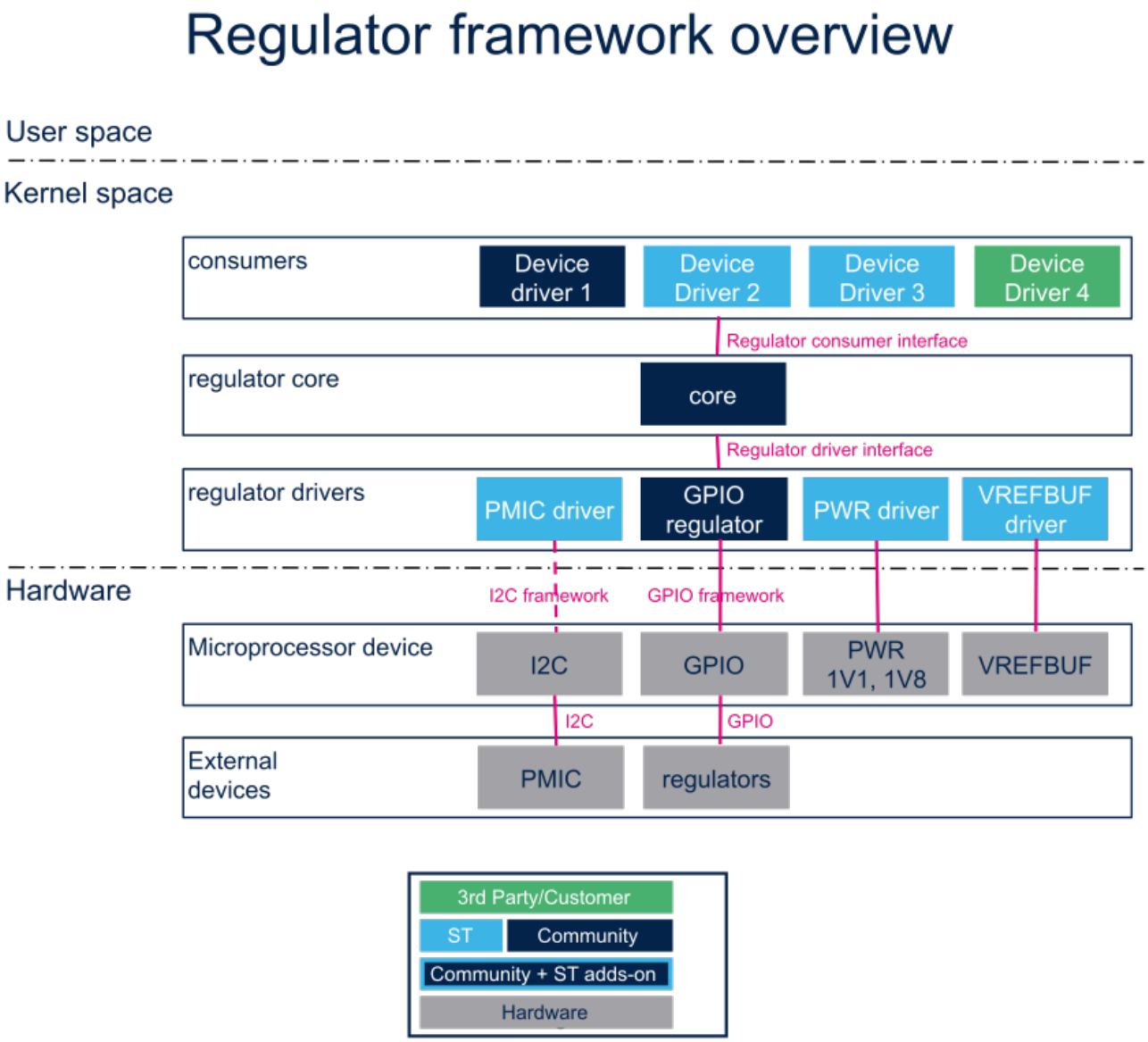
2.1 Overview

The power supplies can be provided by various blocks:

- External single regulators:
  - Low-dropout regulators (LDO)
  - BUCKs (DC-to-DC power converter)
  - Switches
- A Power Management Integrated Circuit (PMIC) that integrates several LDO and BUCKS
- Internal regulators from the microprocessor device internal blocks:
  - PWR peripheral [2]
  - VREFBUF peripheral [3]

All the regulators are implemented and controlled under the standard Linux regulator framework.

2.2 Components Description



2.2.1 External devices: external regulators, PMIC

This corresponds to physical components that provide the various power supplies on the board.

2.2.2 Microprocessor device internal regulators

This corresponds to the regulators integrated to the microprocessor device. Those regulators supply mainly the USB and ADC peripherals.

2.2.3 Regulator drivers

A regulator that can be controlled (enable/disable, adjust voltage...) needs a driver to operate. This is the role of the regulator driver.

A driver can also send notifications like over current or over temperature.

Notes:

- The kernel contains generic drivers for GPIO controlled regulators.
- The PMIC uses a specific driver.
- The internal regulators of the microprocessor device are implemented in the STM32 machine.
- kernel Documentation regulator.txt [4]

2.2.4 Regulator framework core

The core manages all the regulators. A consumer request is not handled directly by a regulator driver. It is handled by the core that can arbitrate requests between consumers in order to save power.

2.2.5 Regulator consumers

The devices correspond to internal or external peripherals of the microprocessor device (ADC, SDCARD, USB, ETHERNET...)

Each peripheral that needs a power supply to operate must enable it. When a regulator is not used, it is disabled by the core.

The consumer interface allows to control a regulator (enable/disable, set voltage...), and to register to a notification service.

- kernel Documentation consumer.txt [5]

2.2.6 Sysfs interface

The regulator framework offers a sysfs interface that can be used for monitoring. It is not possible to control a regulator via the sysfs.

2.2.7 Configfs interface

Most of the regulator configurations are described in the device-tree (configfs)

3 How to find the source code

Everything is part of the kernel source code:

- The regulator framework core and drivers code are located in drivers/regulator directory
  - Driver interface: driver.h
  - Consumer interface: consumer.h

4 Regulator configuration

4.1 Kernel configuration

The configuration is done using the standard menuconfig. Most configurations are available under Device Drivers / Voltage and Current Regulator Support

4.2 Device tree configuration

The device tree describes regulators and consumers:

- A regulator provides a supply.
- A consumer uses a supply.

When possible, the supply name comes from the electrical schematics of the board.

4.2.1 Some regulator drivers

Binding Doc:regulator.yaml

4.2.1.1 Gpio controlled regulator

usb otg vbus:

```
    vbus_otg: regulator-vbus_otg {
        compatible = "regulator-fixed";
        regulator-name = "vbus_otg";
        regulator-min-microvolt = <5000000>;
        regulator-max-microvolt = <5000000>;
        gpio = <&gpioz 4 0>;
        enable-active-high;
    };
```

Binding Doc:fixed-regulator.yaml

sdcard level shifter:

```
    sd_switch: regulator-sd_switch {
        compatible = "regulator-gpio";
        regulator-name = "sd_switch";
        regulator-min-microvolt = <1800000>;
        regulator-max-microvolt = <2900000>;
        regulator-type = "voltage";
        regulator-always-on;

        gpios = <&gpiof 14 GPIO_ACTIVE_HIGH>;
        gpios-states = <0>;
        states = <1800000 0x1 2900000 0x0>;
    };
```

Binding Doc:gpio-regulator.yaml

4.2.1.2 PMIC

```
pmic: stpmu1@33 {
    compatible = "st,stpmu1";
    reg = <0x33>;
    interrupts = <0 2>;
    interrupt-parent = <&gpioa>;
    interrupt-controller;
    #interrupt-cells = <2>;
    status = "okay";

    regulators {
        compatible = "st,stpmu1-regulators";

        vddcore: buck1 {
            regulator-compatible = "buck1";
            regulator-name = "vddcore";
            regulator-min-microvolt = <800000>;
            regulator-max-microvolt = <1350000>;
            regulator-always-on;
            regulator-initial-mode = <2>;
        };

        vdd_ddr: buck2 {
            regulator-compatible = "buck2";
            regulator-name = "vdd_ddr";
            regulator-min-microvolt = <1350000>;
            regulator-max-microvolt = <1350000>;
            regulator-always-on;
            regulator-initial-mode = <2>;
        };
        ...
    };
```

4.2.1.3 Microcontroller device internal regulator

VREFBUF[3] regulator:

```
vrefbuf: vrefbuf@50025000 {
    compatible = "st,stm32-vrefbuf";
    reg = <0x50025000 0x8>;
    regulator-min-microvolt = <1500000>;
    regulator-max-microvolt = <2500000>;
    clocks = <&rcc_clk VREF>;
    status = "disabled";
};
```

Binding Doc:st,stm32-vrefbuf.txt

4.2.2 Consumers

See below some examples of consumers.

The SDMMC needs 2 power supply:

```
&sdmmc1 {
    vmmc-supply = <&vdd_sd>;
    vqmmc-supply = <&sd_switch>;
};
```

The name before "-supply" is not free. vmmc and vqmmc are imposed by the consumer driver. They should be aligned with the name used in the data sheet of the driven component.

The USBPHY is supplied by vdd_usb:

```
&usbphyc {
    vdd-supply = <&vdd_usb>;
};
```

The DAC is supplied by vdda:

```
&dac {
    pinctrl-names = "default";
    pinctrl-0 = <&dac_ch1_pins &dac_ch2_pins>;
    vref-supply = <&vdda>;
    status = "okay";
    ...
};
```

The regulators can be consumers. This is used to define power domains:

```
pmic: stpmu1@33 {
    compatible = "st,stpmu1";
    ...

    regulators {
        compatible = "st,stpmu1-regulators";

        ldo1-supply = <&v3v3>;
        ldo2-supply = <&v3v3>;
        ldo5-supply = <&v3v3>;
        ldo6-supply = <&v3v3>;
        vref_ddr-supply = <&vdd_ddr>;
        vbus_otg-supply = <&bst_out>;
        sw_out-supply = <&bst_out>;
        ...
    };
};
```

Enabling ldo1 will enable v3v3 automatically.

5 Power management

The regulator framework handles the power management at runtime and during suspend.

5.1 Runtime

- The consumers should disable the regulators that are not needed.
- The core disables a regulator as soon as it is not requested by any consumer.

This can be avoided by the usage of "regulator-always-on" property in the device-tree.

5.2 Suspend

The regulator framework offers the possibility to define suspend states for regulators. This is only possible if the driver allows it. The regulator suspend sate is no more handled by the linux kernel in OpenSTLinux distribution.

regulator-state-standby, regulator-state-mem, regulator-state-disk are used to define the state of the regulators during suspend.

```
regulator-state-standby {
    regulator-on-in-suspend;
    regulator-suspend-microvolt = <900000>;
    regulator-mode = <8>;
};
regulator-state-mem {
    regulator-off-in-suspend;
};
```

- The regulator runtime strategy does not apply to suspend. With "regulator-on-in-suspend", the regulator is enabled in suspend even if no consumer uses it.
- "regulator-always-on" does not apply to suspend states.

6 How to trace and debug the framework

6.1 How to trace

The regulator framework provides debugfs tools. The most important one is regulator/regulator_summary:

```
Board $> cat /sys/kernel/debug/regulator/regulator_summary
 regulator                       use open bypass voltage current     min     max
 ---------------------------------------------------------------------------------
  regulator-dummy                  0    6      0     0mV     0mA     0mV     0mV
     vddcore                       0    0      0  1200mV     0mA   800mV  1350mV
     vdd_ddr                       0    1      0  1350mV     0mA  1350mV  1350mV
        vtt_ddr                    0    0      0   675mV     0mA   675mV   675mV
     vdd                           0    1      0  3300mV     0mA  3300mV  3300mV
        58007000.sdmmc                                              3300mV  3300mV
     v3v3                          1    5      0  3300mV     0mA  3300mV  3300mV
        58007000.sdmmc                                              3300mV  3300mV
        vdda                       0    2      0  2900mV     0mA  2900mV  2900mV
           40017000.dac                                                0mV     0mV
           48003000.adc                                                0mV     0mV
        v2v8                       0    0      0  2800mV     0mA  2800mV  2800mV
        vdd_sd                     0    1      0  2900mV     0mA  2900mV  2900mV
           58005000.sdmmc                                           2900mV  2900mV
        v1v8                       0    0      0  1800mV     0mA  1800mV  1800mV
     vdd_usb                       0    0      0  3300mV     0mA  3300mV  3300mV
     bst_out                       0    2      0  5000mV     0mA     0mV     0mV
        vbus_otg                   0    0      0  5000mV     0mA     0mV     0mV
        vbus_sw                    0    0      0  5000mV     0mA     0mV     0mV
  sd_switch                        0    1      0  2900mV     0mA  1800mV  2900mV
     58005000.sdmmc                                              2700mV  2900mV
  reg11                            0    0      0  1100mV     0mA  1100mV  1100mV
  reg18                            0    0      0  1800mV     0mA  1800mV  1800mV
  usb33                            0    0      0  3300mV     0mA  3300mV  3300mV
  vref_ddr                         0    0      0   675mV     0mA     0mV     0mV
```

Notes:

- use: counts the "enable" calls made by the consumers
- open: is the number of consumers that get the regulator
- vdd_sd is a consumer for v3v3
- 58005000.sdmmc is a consumer for v3v3, vdd_sd, sd_switch
- when regulator_always_on property is set, use is equal to ZERO (but the regulator is enabled...)

7 References

- kernel documentation overview
- PWR internal peripheral
- 3.03.1 VREFBUF internal peripheral
- Driver API documentation
- Consumer API documentation

Linux® is a registered trademark of Linus Torvalds.

Board support package

Low-dropout regulator

Power Management Integrated Circuit

voltage reference buffer (STM32 specific)

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

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.)

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

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

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

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

Stable: 20.10.2021 - 09:21 / Revision: 20.10.2021 - 09:17

A quality version of this page, approved on *20 October 2021*, was based off this revision.

This article gives information about the Linux® USB framework.
It explains how to activate USB interface and, based on examples, how to access it from user space.

Contents

1 Framework purpose

The USB (universal serial bus) Linux® framework supports many types of:

- host controllers and peripheral devices
- gadget drivers and classes to be used within a peripheral

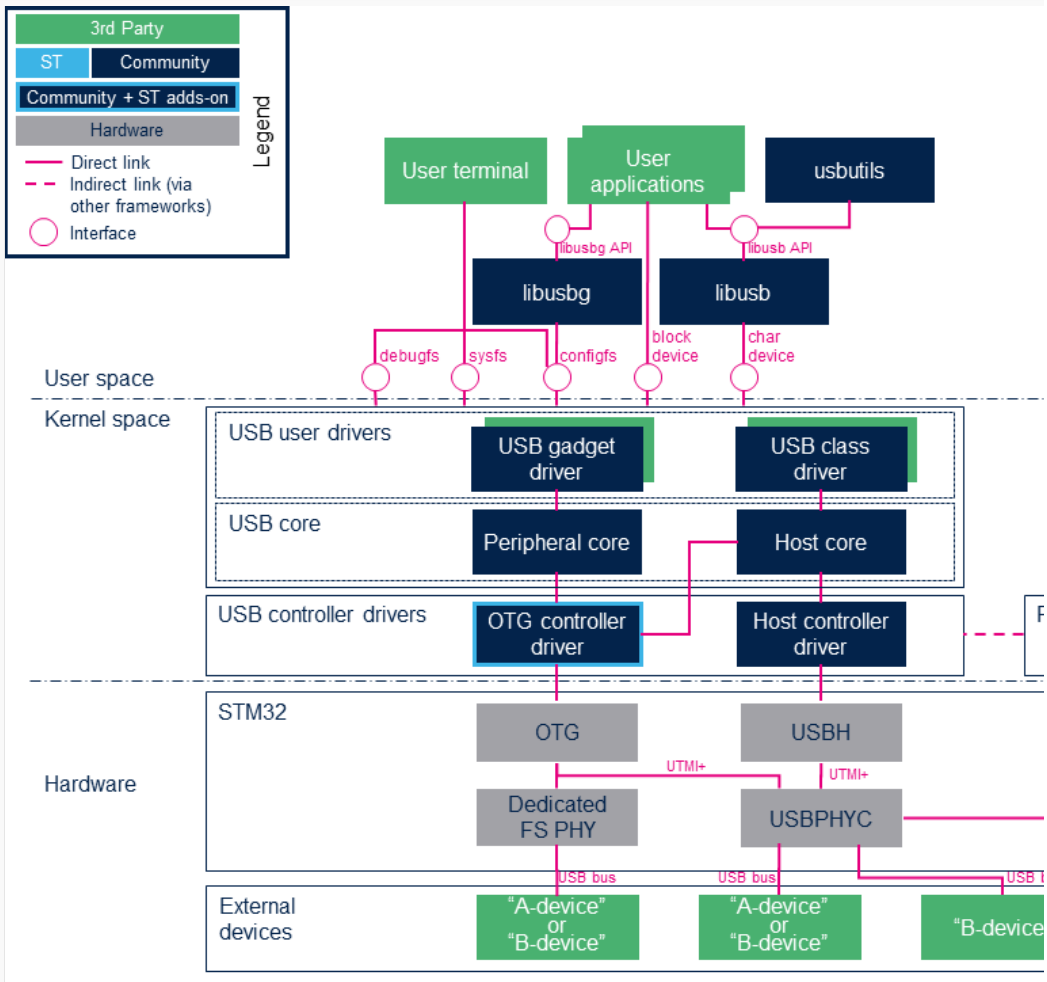
Linux can be used on the host machine. In this case various types of peripherals can be plugged in, such as:

- Mass storage (hard drive, USB stick..)
- HID (keyboard, mouse..)

Linux can also be used as a device on the peripheral side, using gadget drivers. In this case, it can act as:

- USB mass storage (e.g. to export some partitions, filesystem)
- ethernet card
- serial interface
- ...

2 System overview



USB Implementation architecture

2.1 Component description

- **USB userland** (*User space*)
  - *Host-Side* userland

  - libusb[1] is a userland library that provides access to USB devices.

  - usbutils[2] is a set of USB utilities for collecting information about the USB devices that are connected to the USB host. Note that **usbutils** depends on **libusb**.

  - One of the well-known utility is **lsusb**, used to display information about USB buses and the devices connected to them.

  - *Gadget* userland

  - libusbg[3] is a userland library that provides routines for creating and parsing USB gadget devices using the configfs API.

  - Gadget configfs provides configuration interface available through user terminal, used to configure USB Gadget.

  - Common userland

  - **sysfs** provides an information interface available through the user terminal. See How to monitor with sysfs below.

  - **debugfs** provides a debugging interface available through the user terminal. See How to monitor with debugfs below.

- **USB framework** (*Kernel space*): composed of two parts, USB *Host-Side* and USB *Gadget*, which rely on the USB core with specific APIs to support USB host and devices controllers
  - Host-Side provides API interface to class drivers and forwards the request from class drivers to host controller driver.
  - Gadget requires a peripheral controller and the gadget driver to use it.
- **USB controller drivers** (*Kernel space*)
  - *USB Host controller drivers* such as STM32 USBH USB Host controllers in the USB Host-Side framework. STM32 USBH uses kernel community drivers (kernel space), based on the USB framework.

  - Enhanced Host Controller Interface (EHCI) driver and Generic platform ehci driver

  - Open Host Controller Interface (OHCI) driver and Generic platform ohci driver

  - *USB OTG controller drivers* such as STM32 OTG USB OTG controllers in the USB Host-Side framework when they are used either in **otg** or **host** mode, and/or in the USB Gadget framework when they are used either in **otg** or **peripheral** mode. STM32 OTG uses kernel community driver (kernel space), based on the USB framework.

  - DesignWare HS OTG Controller driver

  - USB controller drivers can rely on Generic PHY framework to manage the physical layer for USB data transmissions. STM32 USBPHYC PHY provider is a *PHY driver* in the Generic PHY framework:

  - STM32 USBPHYC driver

- **USB hardware controllers** (*Hardware*)

USB controllers such as STM32 USBH internal peripheral and STM32 OTG internal peripheral, using an on-chip High-Speed UTMI+ PHY (STM32 USBPHYC internal peripheral), or on-chip Full-Speed PHY for STM32 OTG internal peripheral.

- **USB devices** (*External USB devices*)
  - USB OTG specification[4] defines two roles for USB devices: A-Device and B-Device. STM32 OTG controller, depending on the USB connector which is used, can accept both A-Device and B-Device, while STM32 USBH Host controller only manages B-Device:

  - **A-Device** is a **power supplier** acting as a **USB Host** (e.g. a PC)

  - **B-Device** is a **power consumer** and acts as a **USB Peripheral** (e.g. a USB key).

2.2 API description

See USB kernel documentation for more details on API functions.

3 Configuration

3.1 Kernel configuration

USB support, STM32 USBH driver and STM32 OTG driver are activated by default in ST deliveries. Nevertheless, if a specific configuration is required, this section indicates how the USB framework can be activated/deactivated in the kernel.

Activate USB support (CONFIG_USB=y) in the kernel configuration with the Linux Menuconfig tool: Menuconfig or how to configure kernel then select:

```
Device Drivers  --->
 [*] USB support  --->
```

Then activate USB controllers drivers.

To activate the STM32 USBH driver, select:

```
Device Drivers  --->
   --- USB support
   <*>   Support for Host-side USB
   <*>   EHCI HCD (USB 2.0) support
   <*>     Generic EHCI driver for a platform device
   <*>   OHCI HCD (USB 1.1) support
   <*>     Generic OHCI driver for a platform device
```

To activate the STM32 OTG driver, select:

```
Device Drivers  --->
   --- USB support
   <*>   Support for Host-side USB
   <*>   USB Gadget Support  --->
   <*>   DesignWare USB2 DRD Core Support
           DWC2 Mode Selection (Dual Role mode)  --->
```

Then to activate the STM32 USBPHYC driver, select:

```
PHY Subsystem  --->
  -*- PHY Core
  <*> STMicroelectronics STM32 USB HS PHY Controller driver
```

3.2 Device tree configuration

Detailed DT configurations for STM32 USB internal peripherals:

- for STM32 USBH Host controller: USBH device tree configuration
- for STM32 OTG controller: OTG device tree configuration
- for STM32 USBPHYC PHY: USBPHYC device tree configuration

4 How to use the framework

4.1 How to list USB devices

lsusb displays information about the attached USB buses and devices.
In the example above, we have an onboard hub, a USB mouse and USB keyboard plugged into the hub.

```
Board $> lsusb                                                    /* root hubs
correspond to STM32 USB controllers (USBH,  OTG) */
Bus 002 Device 005: ID 413c:2003 Dell Computer Corp. Keyboard
Bus 002 Device 004: ID 046d:c016 Logitech, Inc. Optical Wheel Mouse
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 002: ID 0424:2514 Standard Microsystems Corp. USB 2.0 Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

```
Board $> lsusb -t                                                  /* lsusb -t shows
the USB class, the driver used and the number of ports and speed of each USB devices */
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci-platform/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
        |__ Port 1: Dev 5, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 3: Dev 4, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=dwc2/1p, 480M
```

To limit **lsusb** to the USB keyboard:

```
Board $> lsusb -s 002:005                                        /* lsusb -s [Bus]:
[Device] */
Bus 002 Device 005: ID 413c:2003 Dell Computer Corp. Keyboard
Board $> lsusb -d 413c:2003                                        /* lsusb -d [ID] */
Bus 002 Device 005: ID 413c:2003 Dell Computer Corp. Keyboard
```

To limit **lsusb** to the USB keyboard and display its descriptors:

```
Board $> lsusb -D /dev/bus/usb/002/005                          /* lsusb -D /dev/bus/usb/
[Bus]/[Device] */
Device: ID 413c:2003 Dell Computer Corp. Keyboard
Device Descriptor:
 ...
```

4.2 How to mount a USB key (mass-storage)

```
Board $> mkdir /usb
Board $> mount /dev/sdxx /usb
```

4.3 How to configure USB Gadget through configfs

See USB Gadget configfs documentation for an introduction to USB gadget configfs structure and how to use it to configure Linux USB Gadget.

Here is an example to configure USB Gadget through configfs to use the OTG as a USB Ethernet Gadget with Remote NDIS (RNDIS). See: stm32_usbotg_eth_config.sh .

5 How to trace and debug the framework

5.1 How to monitor

5.1.1 How to monitor with debugfs

Please refer to the USB devices chapter[5] to decode the output.

```
Board $> cat /sys/kernel/debug/usb/devices

T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480  MxCh= 1
B:  Alloc=  0/800 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=1d6b ProdID=0002 Rev= 4.14
S:  Manufacturer=Linux 4.14.0 dwc2_hsotg
S:  Product=DWC OTG Controller
S:  SerialNumber=49000000.usb-otg
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   4 Ivl=256ms

T:  Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  4 Spd=480  MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=05e3 ProdID=0723 Rev=94.54
S:  Manufacturer=Generic
S:  Product=USB Storage
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=500mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

T:  Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480  MxCh= 2
B:  Alloc=  0/800 us ( 0%), #Int=  2, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=1d6b ProdID=0002 Rev= 4.14
S:  Manufacturer=Linux 4.14.0 ehci_hcd
S:  Product=EHCI Host Controller
S:  SerialNumber=5800d000.usbh-ehci
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   4 Ivl=256ms

T:  Bus=02 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=480  MxCh= 4
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=02 MxPS=64 #Cfgs=  1
P:  Vendor=0424 ProdID=2514 Rev= b.b3
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  2mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=01 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=256ms
I:* If#= 0 Alt= 1 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=02 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=256ms

T:  Bus=02 Lev=02 Prnt=02 Port=03 Cnt=01 Dev#=  5 Spd=1.5  MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=413c ProdID=2003 Rev= 1.00
S:  Manufacturer=Dell
S:  Product=Dell USB Keyboard
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr= 70mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=01 Driver=usbhid
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=24ms
```

5.1.2 How to monitor with sysfs

5.1.2.1 USB buses monitoring with sysfs

Please refer to What are the sysfs structures for Linux USB?[6].

```
Board $> ls /sys/bus/usb/devices/
1-0:1.0  1-1  1-1:1.0  2-0:1.0  2-1  2-1.4  2-1.4:1.0  2-1:1.0  usb1  usb2
```

The names that begin with **usb** refer to USB controllers.

The device naming scheme is the following:

- bus-port.port.port... (**1-1**, **2-1**, or **2-1.4** in the example above)

The interfaces are indicated by suffixes in the following form:

- :config.interface (1-1**:1.0**, 2-1**:1.0**, 2-1.4**:1.0** in the example above)

Each interface corresponds to an entry in sysfs and can have its own driver.

5.1.2.2 USB Gadget monitoring with sysfs

Once the USB Gadget is configured, USB Device Controller sysfs is populated. See Documentation/ABI/stable/sysfs-class-udc for a description of each file.

```
Board $> ls /sys/class/udc/49000000.usb-otg/
a_alt_hnp_support  device           is_selfpowered  srp
a_hnp_support      function         maximum_speed   state
b_hnp_enable       is_a_peripheral  power           subsystem
current_speed      is_otg           soft_connect    uevent
```

5.2 How to trace

5.2.1 How to trace with usbmon

usbmon [7] collects traces of the input/output on the USB bus.
It relies on a kernel part and on a user part, and reports the requests made by USB device drivers to the Host controller drivers.
Activate USBMON support (CONFIG_USB_MON=y) in the kernel configuration with Linux Menuconfig tool: Menuconfig or how to configure kernel.
A usbmon entry is created in debugfs. It includes several files.
The file names consist of a number (the USB bus - 0 relates to all buses) and a letter (s, u or t). The s file contains a generic event overview. The t (deprecated) and u files will stream trace data.
To gather debug data, either use the master file 0u (to capture data from all devices) or find out the bus to which your device is connected and use the corresponding bus file. For example, if the device is connected to bus 1:

```
Board $> cat /sys/kernel/debug/usb/usbmon>1u > bus1data.log
```

To stop the capture, just type (CTRL+C) to kill the command. You can then analyze the log with vUSBAnalyzer graphical tool on your Linux host.

5.2.2 How to trace using a protocol analyzer

A USB protocol analyzer is a USB traffic sniffer that decodes USB descriptors and displays bus states and packets sent. Refer to you USB protocol analyzer user manual.

5.3 How to debug

5.3.1 Activating USB framework debug messages

A detailed dynamic trace is available in How to use the kernel dynamic debug

```
Board $> echo  "file usb* +p" > /sys/kernel/debug/dynamic_debug/control
```

This command enables all the traces related to the USB core and drivers at runtime.
A finer selection can be made by choosing only the files to trace.

> Reminder: *loglevel* needs to be increased to 8 either by using boot arguments or by sending the *dmesg -n 8* command from the console

5.3.2 EHCI/OHCI driver debugfs entry

EHCI/OHCI drivers export a debugfs entry when CONFIG_DYNAMIC_DEBUG is enabled.

```
Board $> ls /sys/kernel/debug/usb/ohci/5800c000.usbh-ohci/
async  periodic  registers
Board $> ls /sys/kernel/debug/usb/ehci/5800d000.usbh-ehci/
async  bandwidth  periodic  registers
```

- **async** dumps a snapshot of the async schedule.
- **bandwith** dumps the bandwidth allocation
- **periodic** dumps a snapshot of the periodic schedule.
- **registers** dumps the USB controller registers

5.3.3 DWC2 driver debug messages and debugfs entry

To get the verbose messages from the DWC2 driver used by STM32 OTG, activate "Enable Debugging Messages" in the Linux kernel via the menuconfig Menuconfig or how to configure kernel.

```
Device Drivers  --->
   [*] USB support
   <*>   Support for Host-side USB
   <*>   USB Gadget Support  --->
   <*>   DesignWare USB2 DRD Core Support
   [*]     Enable Debugging Messages
   [*]       Enable Verbose Debugging Messages
   [ ]     Enable Missed SOF Tracking
   [*]     Enable Debugging Messages For Periodic Transfers
```

This can be done manually in your kernel .config file:

```
CONFIG_USB_SUPPORT=y
CONFIG_USB_DWC2=y
CONFIG_USB_DWC2_DEBUG=y
CONFIG_USB_DWC2_VERBOSE=y
CONFIG_USB_DWC2_DEBUG_PERIODIC=y
```

The debug support for DWC2 driver (CONFIG_USB_DWC2_DEBUG) compiles all the files located in Linux kernel drivers/usb/dwc2/ folder with DEBUG flag.

Reminder: *loglevel* needs to be increased to 8 by using either boot arguments or the *dmesg -n 8* command through the console

The DWC2 driver also exports a debugfs entry that contains useful information:

```
Board $> ls /sys/kernel/debug/usb/49000000.usb-otg/
dr_mode  ep0  ep1in  ep1out  ep2in  ep2out  ep3in  ep3out  ep4in  ep4out  ep5in  ep5out
ep6in  ep6out  ep7in  ep7out  ep8in  ep8out  fifo  hw_params  params  regdump state
testmode
```

- **dr_mode** indicates the working mode of the USB controller. It can be "host", "peripheral" or "otg". The value is set through a device tree property.
- **ep*** files show the state of the given endpoint.
- **fifo** shows the FIFO information for the overall FIFO and all the periodic transmission FIFOs.
- **hw_params** shows the parameters read from USB controller registers.
- **params** shows the parameters used by the driver.
- **regdump** dumps all the USB controller registers.
- **state** shows the overall state of the hardware controller and some general information on the available endpoints.
- **testmode** shows/sets usb test mode ("test_j", "test_k", "test_se0_nak", "test_packet", "test_force_enable").

6 Source code location

The source files are located inside the Linux kernel.

- The **USB framework** is under drivers/usb/
- The drivers used for STM32 USBH are under drivers/usb/host/ehci-platform.c , drivers/usb/host/ehci-hcd.c and drivers/usb /host/ohci-platform.c , drivers/usb/host/ohci-hcd.c
- The driver used for STM32 OTG is under drivers/usb/dwc2/

7 References

- libusb: a cross-platform library to access USB devices
- usbutils: USB utilities for Linux, including lsusb
- libusbg: a C library encapsulating the kernel USB gadget-configfs userspace API functionality
- On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification
- Linux USB API: The Linux-USB Host Side API - The USB devices
- What are the sysfs structures for Linux USB?
- usbmon

Linux® is a registered trademark of Linus Torvalds.

Human Interface Device (for USB, Bluetooth...)

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

Application programming interface

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

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

USB On-The-Go (Capability/type of USB port, acting primarily as USB device, to also act as USB host. Also known as USB OTG.)

USB 2.0 Transceiver Macrocell Interface

Enhanced Host Controller Interface

Open Host Controller Interface

Dual-Role Device (USB standard defines host and device roles. OTG controllers support both roles and can be called Dual-Role Devices controllers.)

High Speed (MIPI® Alliance DSI standard)

Device Tree

Stable: 25.09.2020 - 09:43 / Revision: 25.09.2020 - 09:37

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

Contents

1 Article purpose

The purpose of this article is to

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

2 Peripheral overview

The **USBH** peripheral is used to interconnect other systems with STM32 MPU devices, using USB standard.

2.1 Features

The **USBH** peripheral is a USB Host controller supporting high-speed (480 Mbit/s) using an **EHCI** controller, and full- and low-speeds (12 and 1.5 Mbit/s) through **OHCI** controller.
The **USBH** peripheral has two physical ports providing a **UTMI+** physical interface, mapped on an on-chip 2-port high-speed UTMI+ PHY.
It supports the standard registers used for low- and full-speed operation (**OHCI** model) and high-speed operation (**EHCI** model) and the power management feature called Link Power Management (LPM).

The supported standards are:

- *Universal Serial Bus Revision 2.0 Specification*[1], Revision 2.0, April 27, 2000
- *USB 2.0 Link Power Management Addendum Engineering Change Notice to the USB 2.0 specification*[2], July 16, 2007
- *Enhanced Host Controller Interface Specification for Universal Serial Bus*[3], Revision 1.0, March 12, 2002
- *EHCI v1.1 Addendum*[4], August 2008
- *Open Host Controller Interface Specification for USB*[5], Release 1.0a, September 14, 1999
- *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*[6], Version 1.05, March 29, 2001
- *UTMI+ Specification*[7], Revision 1.0, February 25, 2004

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 **USBH** peripheral is a **non-secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

The **USBH** peripheral is usually not used at boot time. But it may be used by the SSBL (see Boot chain overview), for example to boot a kernel stored on a USB key (mass storage).

3.2 Runtime

3.2.1 Overview

The **USBH** peripheral can be allocated to the Arm®Cortex®-A7 non-secure core to be used under Linux® with USB framework.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
High speed interface	USBH (USB Host)		Linux USB 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 by itself can be performed via the STM32CubeMX tool for all internal peripherals. It can then be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

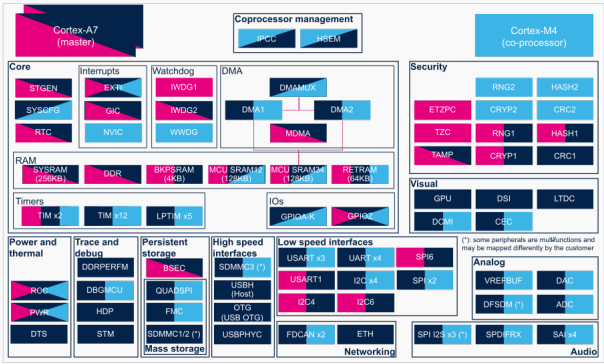
For Linux kernel configuration, please refer to USBH device tree configuration.

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.

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
			Cortex-M4	

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	(STM32Cube)			
High speed interface	USBH (USB Host)	USBH (USB Host)				

4 References

- Universal Serial Bus Revision 2.0 Specification
- ECN USB 2.0 Link Power Management Addendum
- Enhanced Host Controller Interface Specification for Universal Serial Bus
- Enhanced Host Controller Interface Specification: Addendum
- Open Host Controller Interface Specification for USB
- USB 2.0 Transceiver Macrocell Interface (UTMI) Specification
- UTMI+ Specification

USB Host (STM32 specific)

Microprocessor Unit

Enhanced Host Controller Interface

Open Host Controller Interface

USB 2.0 Transceiver Macrocell Interface

Second Stage Boot Loader

Open Portable Trusted Execution Environment

Stable: 20.10.2021 - 10:07 / Revision: 19.07.2021 - 15:25

A quality version of this page, approved on *20 October 2021*, was based off this revision.

Contents

1 Article purpose

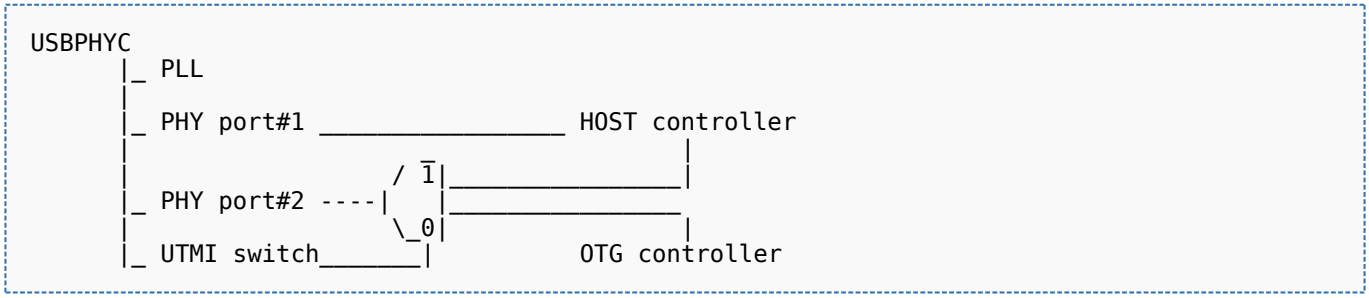
This article explains how to configure the **USBPHYC** internal peripheral when it is assigned to the Linux®OS. In that case, it is controlled by the PHY framework.

The configuration is performed using the device tree mechanism.

It is used by the *USBPHYC Linux driver*[1] which registers the relevant information in PHY framework.

2 DT bindings documentation

USBPHYC device tree bindings[2] describe all the required and optional functions.



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)

The USBPHYC node is declared in stm32mp151.dtsi[3].

- root node e.g. **usbphyc** describes the USBPHYC hardware block parameters such as registers, clocks, resets and supplies.
- child nodes e.g. **usbphyc_port0** and **usbphyc_port1** describe the two high speed PHY ports: *port#1* and *port#2*.

```
usbphyc: usbphyc@address {
        compatible = "st,stm32mp1-usbphyc";
        ...                                          /* usbphyc resources: registers,
clocks, resets and supplies */
        usbphyc_port0: usb-phy@0 {
                ...                                  /* usbphyc HS PHY port#1 */
        };
        usbphyc_port1: usb-phy@1 {
                ...                                  /* usbphyc HS PHY port#2 */
        };
};
```

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)

Follow the sequences described in the below chapters to configure and enable the USBPHYC on your board.

The '**usbphyc' root node** must be filled in:

- Enable the USBPHYC block by setting **status = "okay".**

The **child nodes** for each port must be filled in:

- Configure the USBPHYC 3V3 regulator[4] by setting **phy-supply = <&your_regulator>**.

USB HS PHY ports require an external 3V3 power supply to be provided at VDD3V3_USBHS pin.

- Optional, **for ecosystem release v2.1.0** , you may configure the VBUS 5V regulator[4] by using a connector node with **vbus-supply = <&your_regulator>**. Have a look at connector bindings [5]

The **child nodes** for each port may also be tuned:

- Optional: create a **usb_phy_tuning** node that can take optional parameters in DT root folder ('/')
- Optional: add '**st,phy-tuning = <&usb_phy_tuning>**' in '**usbphyc_port0**' and/or '**usbphyc_port1**' node to use this tuning.

It may be necessary to adjust the phy settings to compensate parasitics, which can be due to USB connector/receptacle, routing, ESD protection component.

Optional tuning parameter list is available in *USBPHYC device tree bindings*[2].

3.3 DT configuration example

The example below shows how to enable and configure USBPHYC ports in the board file.

For ecosystem release v2.1.0

```
&usbphyc {
        status = "okay";                                /* enable USB HS PHY controller */
};

&usbphyc_port0 {
        phy-supply = <&vdd_usb>;                        /* references the 3V3 voltage
regulator on the user board */
        st,phy-tuning = <&usb_phy_tuning>;              /* optional USB HS PHY port#1 tuning
*/
        connector {
                compatible = "usb-a-connector";
                vbus-supply = <&vbus_sw>;               /* references the optional 5V voltage
regulator on the user board */
        };
};

&usbphyc_port1 {
        phy-supply = <&vdd_usb>;                        /* references the 3V3 voltage
regulator on the user board */
        st,phy-tuning = <&usb_phy_tuning>;              /* optional USB HS PHY port#2 tuning
*/
};
```

```
/ {                                                     /* optional USB HS PHY tuning
example, to be added in DT root node, e.g. '/' */
        usb_phy_tuning: usb-phy-tuning {
                st,current-boost = <2>;
                st,no-lfs-fb-cap;
                st,hs-dc-level = <2>;
                st,hs-rftime-reduction;
                st,hs-current-trim = <5>;
                st,hs-impedance-trim = <0>;
                st,squelch-level = <1>;
                st,no-hs-ftime-ctrl;
                st,hs-tx-staggering;
        };
};
```

For ecosystem release v1.0.0 to v2.1.0

```
&usbphyc {
        status = "okay";                                /* enable USB HS PHY controller */
};

&usbphyc_port0 {
```

```
        phy-supply = <&vdd_usb>;                    /* references the 3V3 voltage
regulator on the user board */
        st,phy-tuning = <&usb_phy_tuning>;          /* optional USB HS PHY port#1 tuning
*/
};

&usbphyc_port1 {
        phy-supply = <&vdd_usb>;                    /* references the 3V3 voltage
regulator on the user board */
        st,phy-tuning = <&usb_phy_tuning>;          /* optional USB HS PHY port#2 tuning
*/
};
```

```
/ {                                                 /* optional USB HS PHY tuning
example, to be added in DT root node, e.g. '/' */
        usb_phy_tuning: usb-phy-tuning {
                st,current-boost = <2>;
                st,no-lfs-fb-cap;
                st,hs-dc-level = <2>;
                st,hs-rftime-reduction;
                st,hs-current-trim = <5>;
                st,hs-impedance-trim = <0>;
                st,squelch-level = <1>;
                st,no-hs-ftime-ctrl;
                st,hs-tx-staggering;
        };
};
```

Static configuration of the UTMI switch to assign the **port#2** to either USBH or OTG is done by the **PHY user node**[6]:

- Please refer to USBH_device_tree_configuration
- Please refer to OTG_device_tree_configuration

usbphyc_port1 user must configure an additional specifier for UTMI switch: **0 to select OTG**, **1 to select USBH**

Abstract of the example to configure port#2, to be assigned to the USBH:

```
&usbh_ehci {
        phys = <&usbphyc_port0>, <&usbphyc_port1 1>; /* 1: UTMI switch selects the USBH */
        phy-names = "usb", "usb";
        ...
}
```

Abstract of the example to configure port#2, to be assigned to the OTG:

```
&usbotg_hs {
        phys = <&usbphyc_port1 0>;                  /* 0: UTMI switch selects the OTG */
        phy-names = "usb2-phy";
        ...
}
```

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:

- drivers/phy/st/phy-stm32-usbphyc.c , STM32 USB PHY Controller driver
- 2.02.1 Documentation/devicetree/bindings/phy/phy-stm32-usbphyc.yaml , USBPHYC device tree bindings
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file
- 4.04.1 Regulator overview
- Documentation/devicetree/bindings/connector/usb-connector.yaml
- Documentation/devicetree/bindings/phy/phy-bindings.txt ,PHY generic bindings

Operating System

Device Tree

USB 2.0 Transceiver Macrocell Interface

USB On-The-Go (Capability/type of USB port, acting primarily as USB device, to also act as USB host. Also known as USB OTG.)

High Speed (MIPI® Alliance DSI standard)

USB Host (STM32 specific)

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

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

Contents

1 Article purpose

The purpose of this article is to

- briefly introduce the USBPHYC 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 USBPHYC peripheral.

2 Peripheral overview

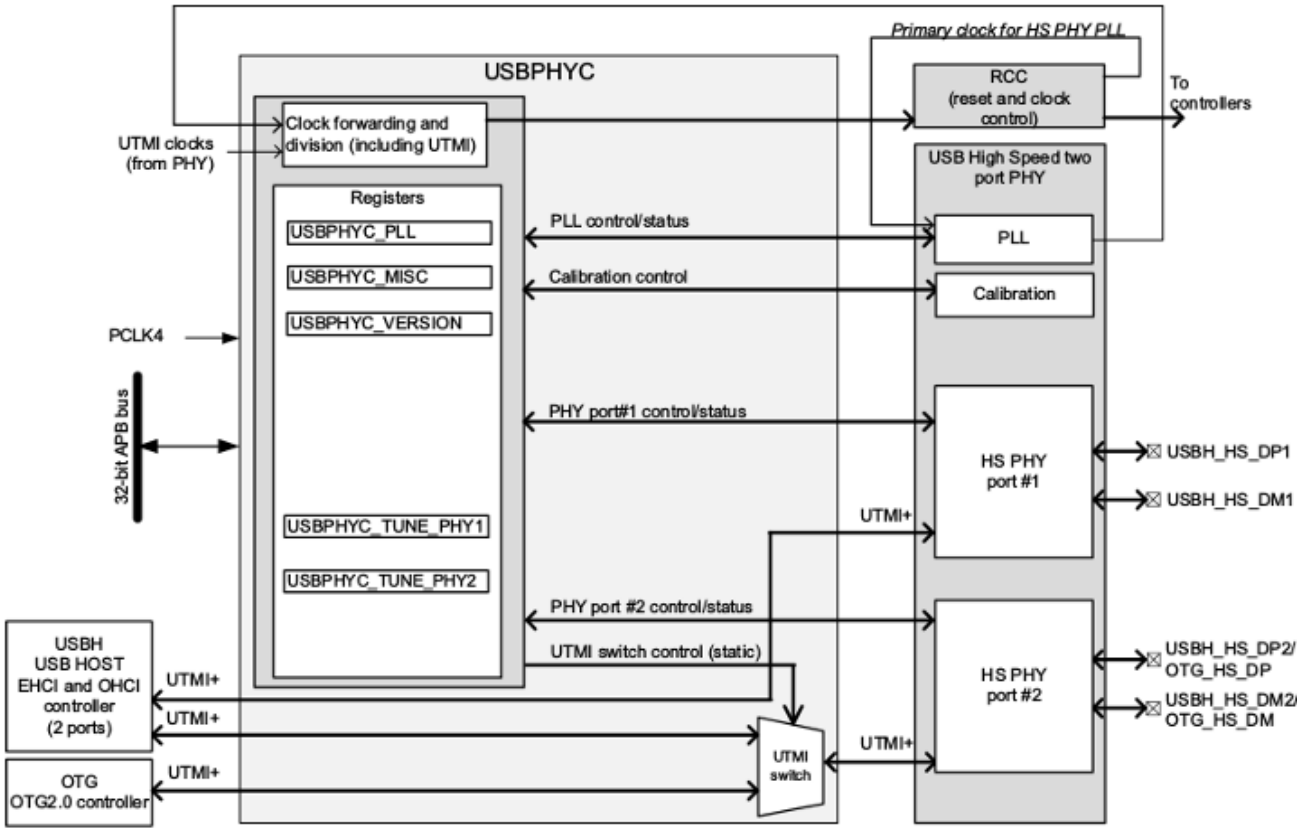
The **USBPHYC** peripheral is a block that contains a dual port USB high-speed UTMI+ PHY and a UTMI switch. It makes the interface between:

- the internal USB controllers (USBH and OTG)
- the external USB physical lines (DP, DM)

2.1 Features

The **USBPHYC** peripheral:

- controls a two port high-speed PHY:
  - Port1 connected to the USBH controller
  - Port2 connected via the UTMI+switch to the USBH or to the OTG controller
- sets the PLL values
- performs other controls (and monitoring) on the PHY.



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 **USBPHYC** is a **non-secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

USBPHYC instances are boot devices that support Flash programming with STM32CubeProgrammer.

The USBPHYC peripheral is used by ROM code, FSBL and SSBL when using OTG in Device mode (DFU).

The SSBL can use OTG in Host mode or USBH (mass storage). The USBPHYC peripheral can be used to boot on a kernel stored on a USB key, or after a kernel panic to perform the crash dump saving to the USB key.

3.2 Runtime

3.2.1 Overview

The USBPHYC peripheral can be allocated to the the Arm®Cortex®-A7 non-secure core to be used under Linux® with PHY framework.

The peripheral assignment chapter describes which peripheral instance can be assigned to which context.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
High-speed interface	USBPHYC (USB HS PHY controller)		Linux PHY 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.

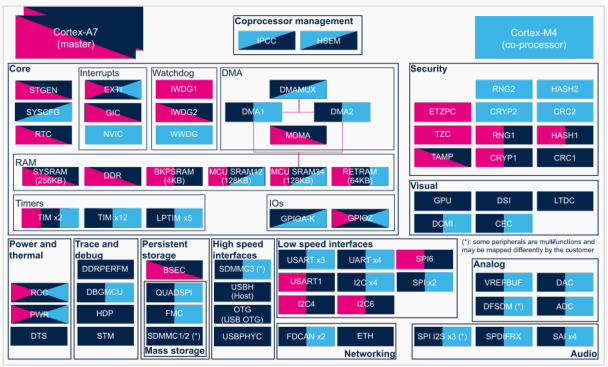
For Linux kernel configuration, please refer to USBPHYC device tree configuration.

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.

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)			
High speed interface	USBPHYC (USB HS PHY controller)	USBPHYC (USB HS PHY controller)				

USB 2.0 Transceiver Macrocell Interface

Device Firmware Upgrade

Open Portable Trusted Execution Environment

High Speed (MIPI® Alliance DSI standard)