# HDP device tree configuration

# Contents



Stable: 21.09.2021 - 14:20 / Revision: 09.09.2021 - 13:05

Stable: 21.09.2021 - 14:20 / Revision: 30.08.2021 - 18:08

A quality version of this page, approved on *21 September 2021*, was based off this revision.

Contents

# 1 Article purpose

This article explains how to configure the HDP driver when the peripheral is assigned to the Linux® OS.

The configuration is performed using the device tree mechanism, which provides a hardware description of the Ethernet peripheral used by STM32 HDP driver

# 2 DT bindings documentation

The *HDP* tree bindings are composed of:

- STM32 HDP device tree bindings [1]

# 3 DT configuration

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

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

## 3.1 DT configuration (STM32 level)

The HDP node is described in the stm32mp151.dtsi [2] file with disabled status and required properties such as:

- The physical base address and size of the device register map
- The HDP clock

```
hdp: hdp@5002a000 {
        compatible = "st,stm32mp1-hdp";
        reg = <0x5002a000 0x400>;
        clocks = <&rcc HDP>;
        clock-names = "hdp";
        status = "disabled";
};
```

The required and optional properties are fully described in the bindings files.

**Warning**

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

## 3.2 HDP DT configuration (board level)

Part of the device tree describes the HDP hardware used on a given board. The DT node (**"hdp"**) must be filled in as follows:

- Enable the HDP block by setting **status = "okay".**
- Configure the pins in use via pinctrl, through **pinctrl-0** (default pins), **pinctrl-1** (sleep pins) and **pinctrl-names**.
- Configure the HDP interface using **muxing-hdp** to indicate which one of the 16 possible output pins is assigned to each HDP output.

```
&hdp {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&hdpx_pins_y>;
        pinctrl-1 = <&hdpx_pins_sleep_y>;
        status = "disabled";

        muxing-hdp = <(STM32_HDP(x, HDPx_value))>;
};
```

## 3.3 DT configuration examples

The example below shows how to configure and enable HDP instances at board level:

```
&hdp {
   pinctrl-names = "default", "sleep";
   pinctrl-0 = <&hdp0_pins_a &hdp6_pins_a &hdp7_pins_a>;                    /* configure
pinctrl for hdp pin 0, 6 and 7*/
   pinctrl-1 = <&hdp0_pins_sleep_a &hdp6_pins_sleep_a &hdp7_pins_sleep_a>;
   status = "okay";                                                         /* enable HDP */


   muxing-hdp = <(STM32_HDP(0, HDP0_GPOVAL_0) |                             /* For HDP pin
0, the signal HDP0_GPOVAL_0 is selected*/
                  STM32_HDP(6, HDP6_GPOVAL_6) |                             /* For HDP pin
6, the signal HDP0_GPOVAL_6 is selected*/
                  STM32_HDP(7, HDP7_GPOVAL_7))>;                            /* For HDP pin
7, the signal HDP0_GPOVAL_7 is selected*/
};
```

List of all possible HDP signals:

```
/* define HDP Pins number*/
HDP0_PWR_PWRWAKE_SYS
HDP0_CM4_SLEEPDEEP
HDP0_PWR_STDBY_WKUP
HDP0_PWR_ENCOMP_VDDCORE
HDP0_BSEC_OUT_SEC_NIDEN
HDP0_RCC_CM4_SLEEPDEEP
HDP0_GPU_DBG7
HDP0_DDRCTRL_LP_REQ
HDP0_PWR_DDR_RET_ENABLE_N
HDP0_GPOVAL_0

HDP1_PWR_PWRWAKE_MCU
HDP1_CM4_HALTED
HDP1_CA7_NAXIERRIRQ
HDP1_PWR_OKIN_MR
HDP1_BSEC_OUT_SEC_DBGEN
HDP1_EXTI_SYS_WAKEUP
HDP1_RCC_PWRDS_MPU
HDP1_GPU_DBG6
HDP1_DDRCTRL_DFI_CTRLUPD_REQ
HDP1_DDRCTRL_CACTIVE_DDRC_ASR
HDP1_GPOVAL_1

HDP2_PWR_PWRWAKE_MPU
HDP2_CM4_RXEV
HDP2_CA7_NPMUIRQ1
HDP2_CA7_NFIQOUT1
HDP2_BSEC_IN_RSTCORE_N
HDP2_EXTI_C2_WAKEUP
HDP2_RCC_PWRDS_MCU
HDP2_GPU_DBG5
HDP2_DDRCTRL_DFI_INIT_COMPLETE
HDP2_DDRCTRL_PERF_OP_IS_REFRESH
HDP2_DDRCTRL_GSKP_DFI_LP_REQ
HDP2_GPOVAL_2

HDP3_PWR_SEL_VTH_VDD_CORE
HDP3_CM4_TXEV
```

```
HDP3_CA7_NPMUIRQ0
HDP3_CA7_NFIQOUT0
HDP3_BSEC_OUT_SEC_DFTLOCK
HDP3_EXTI_C1_WAKEUP
HDP3_RCC_PWRDS_SYS
HDP3_GPU_DBG4
HDP3_DDRCTRL_STAT_DDRC_REG_SELREF_TYPE0
HDP3_DDRCTRL_CACTIVE_1
HDP3_GPOVAL_3

HDP4_PWR_PDDS
HDP4_CM4_SLEEPING
HDP4_CA7_NRESET1
HDP4_CA7_NIRQOUT1
HDP4_BSEC_OUT_SEC_DFTEN
HDP4_BSEC_OUT_SEC_DBGSWENABLE
HDP4_ETH_OUT_PMT_INTR_O
HDP4_GPU_DBG3
HDP4_DDRCTRL_STAT_DDRC_REG_SELREF_TYPE1
HDP4_DDRCTRL_CACTIVE_0
HDP4_GPOVAL_4

HDP5_CA7_STANDBYWFIL2
HDP5_PWR_VTH_VDDCORE_ACK
HDP5_CA7_NRESET0
HDP5_CA7_NIRQOUT0
HDP5_BSEC_IN_PWROK
HDP5_BSEC_OUT_SEC_DEVICEEN
HDP5_ETH_OUT_LPI_INTR_O
HDP5_GPU_DBG2
HDP5_DDRCTRL_CACTIVE_DDRC
HDP5_DDRCTRL_WR_CREDIT_CNT
HDP5_GPOVAL_5

HDP6_CA7_STANDBYWFI1
HDP6_CA7_STANDBYWFE1
HDP6_CA7_EVENT0
HDP6_CA7_DBGACK1
HDP6_BSEC_OUT_SEC_SPNIDEN
HDP6_ETH_OUT_MAC_SPEED_O1
HDP6_GPU_DBG1
HDP6_DDRCTRL_CSYSACK_DDRC
HDP6_DDRCTRL_LPR_CREDIT_CNT
HDP6_GPOVAL_6

HDP7_CA7_STANDBYWFI0
HDP7_CA7_STANDBYWFE0
HDP7_CA7_DBGACK0
HDP7_BSEC_OUT_FUSE_OK
HDP7_BSEC_OUT_SEC_SPIDEN
HDP7_ETH_OUT_MAC_SPEED_O0
HDP7_GPU_DBG0
HDP7_DDRCTRL_CSYSREQ_DDRC
HDP7_DDRCTRL_HPR_CREDIT_CNT
HDP7_GPOVAL_7
```

# 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

- Documentation/devicetree/bindings/soc/stm32/stm32_hdp.txt
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file

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

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

Contents

# 1 Purpose

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

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

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

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

## 1.1 Device tree basis

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

- Device Tree for STM32MP [2]

## 1.2 Source files

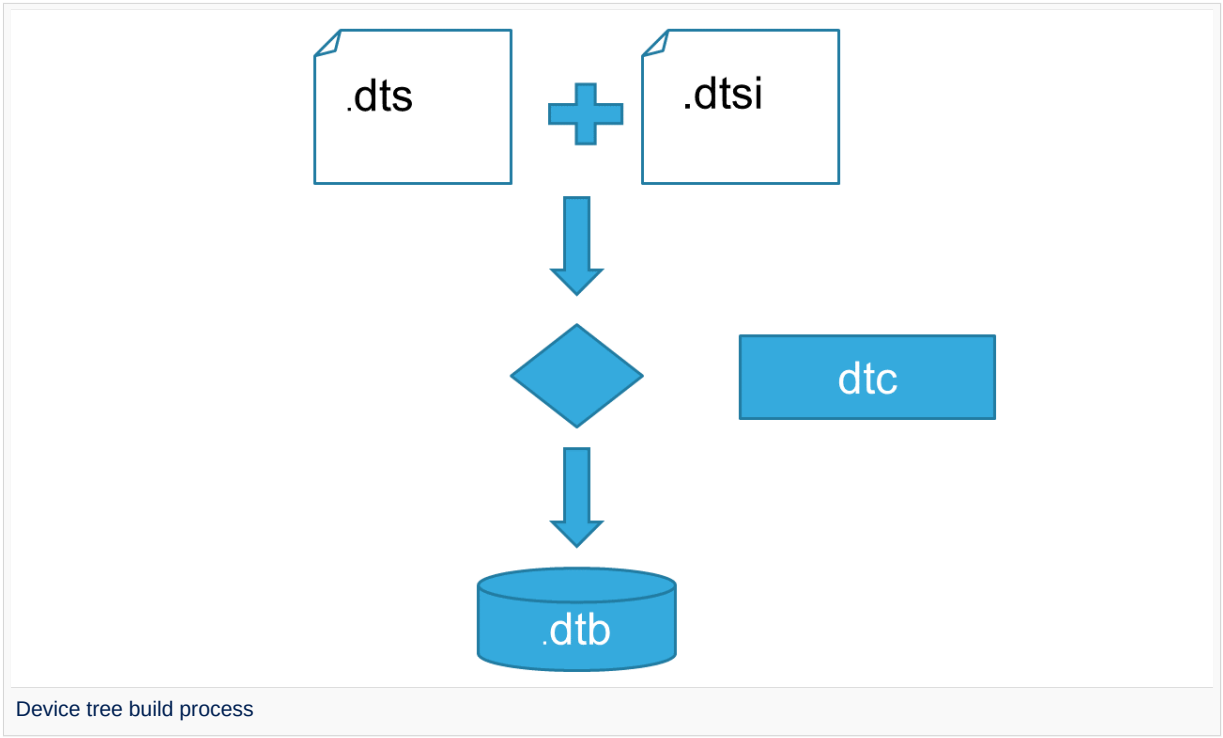
- **.dts:** The device tree source (DTS). This format is a textual representation of a device tree in a form that can be processed by DTC (Device Tree Compiler) into a binary file expected by software components: Linux® Kernel, U-Boot and TF-A.
- **.dtsi:** Source files that can be included from a DTS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

## 1.3 Bindings

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

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

## 1.4 Build




Device tree build process

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

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

**Information**

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

## 1.5 Tools

The device tree compiler offers also some tools:

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

There are several ways to get those tools:

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

# 2 STM32

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

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

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

# 3 How to go further

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

# 4 References


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

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


Stable: 21.09.2021 - 14:25 / Revision: 21.09.2021 - 14:24

A quality version of this page, approved on *21 September 2021*, was based off this revision.

Contents

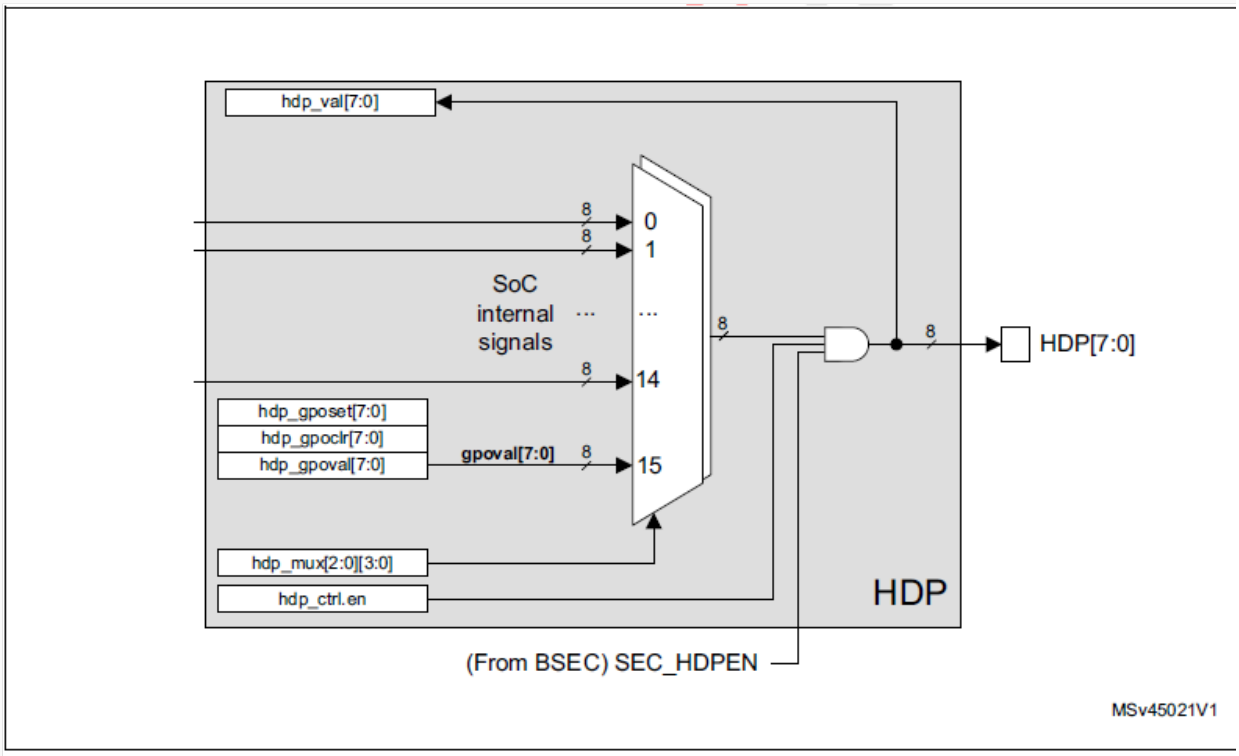
# 1 Article purpose

This article introduces the **Hardware Debug Port** which allows the observation of internal signals. By using multiplexers, up to 16 signals of each of 8-bit output can be observed. The article explains:

- How to configure, use and debug the driver
- The driver structure, and where the source code can be found.

## 2 Short description

- 8 output signals
- One of 16 internal signals with individual control
- 8 software-programmable signals for pinout agnostic code debugging
- Output disabling by security signal




HDP Block Diagram

# 3 Configuration

## 3.1 Kernel configuration

The **HDP** is enabled and ready to be used in all STM32MPU Embedded Software Distributions, via the Linux® kernel configuration **CONFIG_STM32_HDP**, set to disabled by default.

```
Symbol: STM32_HDP
Location:
  Device Drivers
  [*] SOC (System On Chip) specific Drivers
    [*] STMicroelectronics STM32MP157 Hardware Debug Port (HDP) pin control
```

Please refer to the Menuconfig or how to configure kernel article for instructions on modifying the configuration, and recompiling the Linux kernel image in the Distribution Package context.

## 3.2 Device tree

Refer to the HDP device tree configuration article when configuring the HDP Linux kernel driver.

# 4 How to trace and debug

## 4.1 How to monitor

### 4.1.1 How to monitor with debugfs

**sysfs** entry can be used to browse HDP registers.

```
Board $> /sys/kernel/debug/hdp# ls
 ctrl  gpoclr  gposet  gpoval  mux  val
```

See the HDP chapter in the reference manual [1] for further register details.

# 5 Source code location

The HDP Linux driver source code is composed of:

- drivers/soc/st/stm32_hdp.c : handle common resources: registers, clock.

# 6 References

- STM32MP15 reference manuals

Stable: 21.09.2021 - 14:20 / Revision: 09.09.2021 - 13:05

A quality version of this page, approved on *21 September 2021*, was based off this revision.

Contents

# 1 Article purpose

This article explains how to configure the HDP driver when the peripheral is assigned to the Linux® OS.

The configuration is performed using the device tree mechanism, which provides a hardware description of the Ethernet peripheral used by STM32 HDP driver

# 2 DT bindings documentation

The *HDP* tree bindings are composed of:

- STM32 HDP device tree bindings [1]

# 3 DT configuration

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

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

## 3.1 DT configuration (STM32 level)

The HDP node is described in the stm32mp151.dtsi [2] file with disabled status and required properties such as:

- The physical base address and size of the device register map
- The HDP clock

```
hdp: hdp@5002a000 {
        compatible = "st,stm32mp1-hdp";
        reg = <0x5002a000 0x400>;
        clocks = <&rcc HDP>;
        clock-names = "hdp";
        status = "disabled";
};
```

The required and optional properties are fully described in the bindings files.

**Warning**

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

## 3.2 HDP DT configuration (board level)

Part of the device tree describes the HDP hardware used on a given board. The DT node (**"hdp"**) must be filled in as follows:

- Enable the HDP block by setting **status = "okay".**
- Configure the pins in use via pinctrl, through **pinctrl-0** (default pins), **pinctrl-1** (sleep pins) and **pinctrl-names**.
- Configure the HDP interface using **muxing-hdp** to indicate which one of the 16 possible output pins is assigned to each HDP output.

```
&hdp {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&hdpx_pins_y>;
        pinctrl-1 = <&hdpx_pins_sleep_y>;
        status = "disabled";

        muxing-hdp = <(STM32_HDP(x, HDPx_value))>;
};
```

## 3.3 DT configuration examples

The example below shows how to configure and enable HDP instances at board level:

```
&hdp {
   pinctrl-names = "default", "sleep";
   pinctrl-0 = <&hdp0_pins_a &hdp6_pins_a &hdp7_pins_a>;                    /* configure
pinctrl for hdp pin 0, 6 and 7*/
   pinctrl-1 = <&hdp0_pins_sleep_a &hdp6_pins_sleep_a &hdp7_pins_sleep_a>;
   status = "okay";                                                         /* enable HDP */


   muxing-hdp = <(STM32_HDP(0, HDP0_GPOVAL_0) |                             /* For HDP pin
0, the signal HDP0_GPOVAL_0 is selected*/
                  STM32_HDP(6, HDP6_GPOVAL_6) |                             /* For HDP pin
6, the signal HDP0_GPOVAL_6 is selected*/
                  STM32_HDP(7, HDP7_GPOVAL_7))>;                            /* For HDP pin
7, the signal HDP0_GPOVAL_7 is selected*/
};
```

List of all possible HDP signals:

```
/* define HDP Pins number*/
HDP0_PWR_PWRWAKE_SYS
HDP0_CM4_SLEEPDEEP
HDP0_PWR_STDBY_WKUP
HDP0_PWR_ENCOMP_VDDCORE
HDP0_BSEC_OUT_SEC_NIDEN
HDP0_RCC_CM4_SLEEPDEEP
HDP0_GPU_DBG7
HDP0_DDRCTRL_LP_REQ
HDP0_PWR_DDR_RET_ENABLE_N
HDP0_GPOVAL_0

HDP1_PWR_PWRWAKE_MCU
HDP1_CM4_HALTED
HDP1_CA7_NAXIERRIRQ
HDP1_PWR_OKIN_MR
HDP1_BSEC_OUT_SEC_DBGEN
HDP1_EXTI_SYS_WAKEUP
HDP1_RCC_PWRDS_MPU
HDP1_GPU_DBG6
HDP1_DDRCTRL_DFI_CTRLUPD_REQ
HDP1_DDRCTRL_CACTIVE_DDRC_ASR
HDP1_GPOVAL_1

HDP2_PWR_PWRWAKE_MPU
HDP2_CM4_RXEV
HDP2_CA7_NPMUIRQ1
HDP2_CA7_NFIQOUT1
HDP2_BSEC_IN_RSTCORE_N
HDP2_EXTI_C2_WAKEUP
HDP2_RCC_PWRDS_MCU
HDP2_GPU_DBG5
HDP2_DDRCTRL_DFI_INIT_COMPLETE
HDP2_DDRCTRL_PERF_OP_IS_REFRESH
HDP2_DDRCTRL_GSKP_DFI_LP_REQ
HDP2_GPOVAL_2

HDP3_PWR_SEL_VTH_VDD_CORE
HDP3_CM4_TXEV
```

```
HDP3_CA7_NPMUIRQ0
HDP3_CA7_NFIQOUT0
HDP3_BSEC_OUT_SEC_DFTLOCK
HDP3_EXTI_C1_WAKEUP
HDP3_RCC_PWRDS_SYS
HDP3_GPU_DBG4
HDP3_DDRCTRL_STAT_DDRC_REG_SELREF_TYPE0
HDP3_DDRCTRL_CACTIVE_1
HDP3_GPOVAL_3

HDP4_PWR_PDDS
HDP4_CM4_SLEEPING
HDP4_CA7_NRESET1
HDP4_CA7_NIRQOUT1
HDP4_BSEC_OUT_SEC_DFTEN
HDP4_BSEC_OUT_SEC_DBGSWENABLE
HDP4_ETH_OUT_PMT_INTR_O
HDP4_GPU_DBG3
HDP4_DDRCTRL_STAT_DDRC_REG_SELREF_TYPE1
HDP4_DDRCTRL_CACTIVE_0
HDP4_GPOVAL_4

HDP5_CA7_STANDBYWFIL2
HDP5_PWR_VTH_VDDCORE_ACK
HDP5_CA7_NRESET0
HDP5_CA7_NIRQOUT0
HDP5_BSEC_IN_PWROK
HDP5_BSEC_OUT_SEC_DEVICEEN
HDP5_ETH_OUT_LPI_INTR_O
HDP5_GPU_DBG2
HDP5_DDRCTRL_CACTIVE_DDRC
HDP5_DDRCTRL_WR_CREDIT_CNT
HDP5_GPOVAL_5

HDP6_CA7_STANDBYWFI1
HDP6_CA7_STANDBYWFE1
HDP6_CA7_EVENT0
HDP6_CA7_DBGACK1
HDP6_BSEC_OUT_SEC_SPNIDEN
HDP6_ETH_OUT_MAC_SPEED_O1
HDP6_GPU_DBG1
HDP6_DDRCTRL_CSYSACK_DDRC
HDP6_DDRCTRL_LPR_CREDIT_CNT
HDP6_GPOVAL_6

HDP7_CA7_STANDBYWFI0
HDP7_CA7_STANDBYWFE0
HDP7_CA7_DBGACK0
HDP7_BSEC_OUT_FUSE_OK
HDP7_BSEC_OUT_SEC_SPIDEN
HDP7_ETH_OUT_MAC_SPEED_O0
HDP7_GPU_DBG0
HDP7_DDRCTRL_CSYSREQ_DDRC
HDP7_DDRCTRL_HPR_CREDIT_CNT
HDP7_GPOVAL_7
```

# 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

- Documentation/devicetree/bindings/soc/stm32/stm32_hdp.txt
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file

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

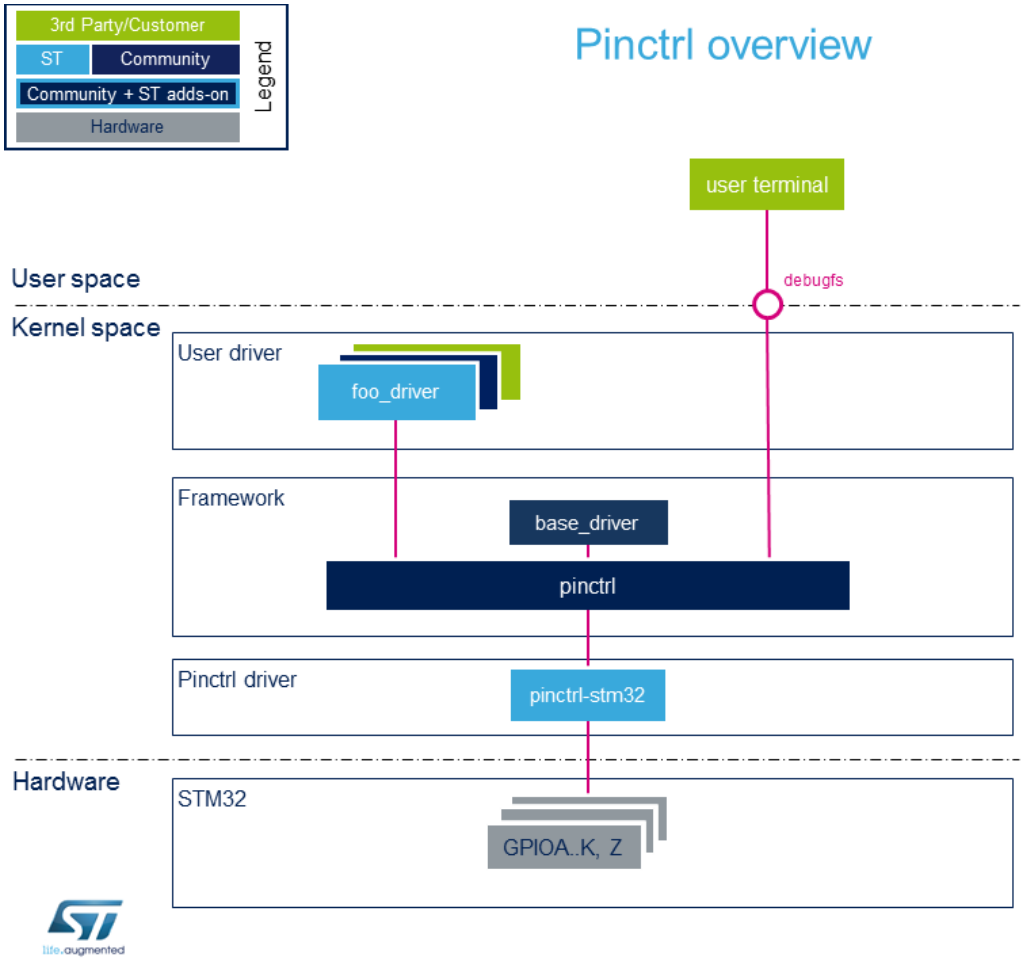
Many of the microprocessor pins (with digital I/O or analog pin type) are multiplexed between different functions: GPIO, alternate function(s).
Pinctrl framework is used to:

- 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**

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

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

# 1 STM32CubeMX overview

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

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

It provides the means to:

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

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

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

# 2 STM32CubeMX main features

- Peripheral and middleware parameters

  Presents options specific to each supported software component

- Peripheral assignment to processors

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

- Power consumption calculator

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

- Code generation

  Makes code regeneration possible, while keeping user code intact

- Pinout configuration

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

- Clock tree initialization

  Chooses the oscillator and sets the PLL and clock dividers

- DDR tunning tool

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

# 3 How to get STM32CubeMX

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

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