Pinctrl overview

Contents

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

Stable: 11.06.2020 - 09:09 | Revision: 10.06.2020 - 13:17

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

Contents

1 Framework purpose

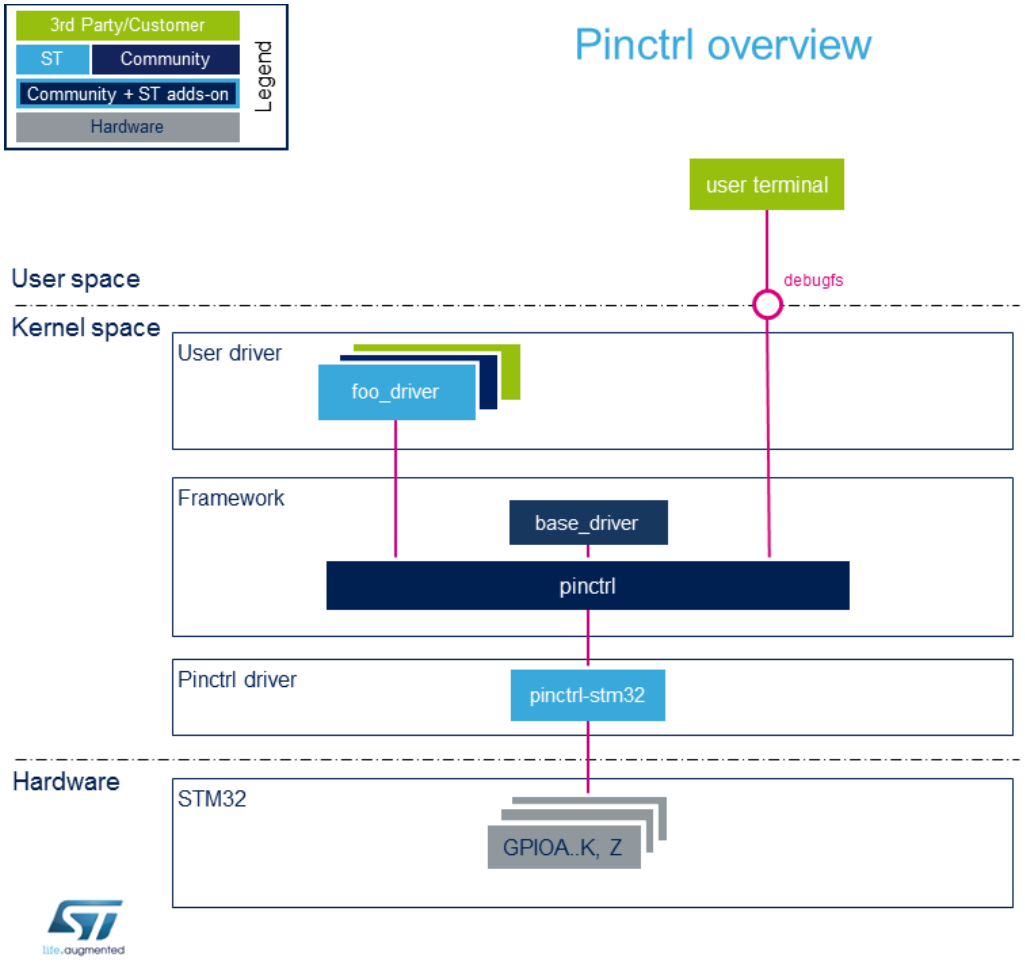
Many of the microprocessor pins (with digital I/O or analog pin type) are multiplexed between different functions: GPIO, alternate function(s).
Pinctrl framework is used to:

- 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**

foo_driver could be any driver that needs to control a GPIO

Stable: 11.06.2020 - 09:05 / Revision: 10.06.2020 - 15:53

You do not have permission to read this page, for the following reason:

The action "Read pages" for the draft version of this page is only available for the groups ST_editors, ST_readers, Selected_editors, sysop, reviewer

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

You do not have permission to read this page, for the following reason:

The action "Read pages" for the draft version of this page is only available for the groups ST_editors, ST_readers, Selected_editors, sysop, reviewer

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

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

Contents

1 Framework purpose

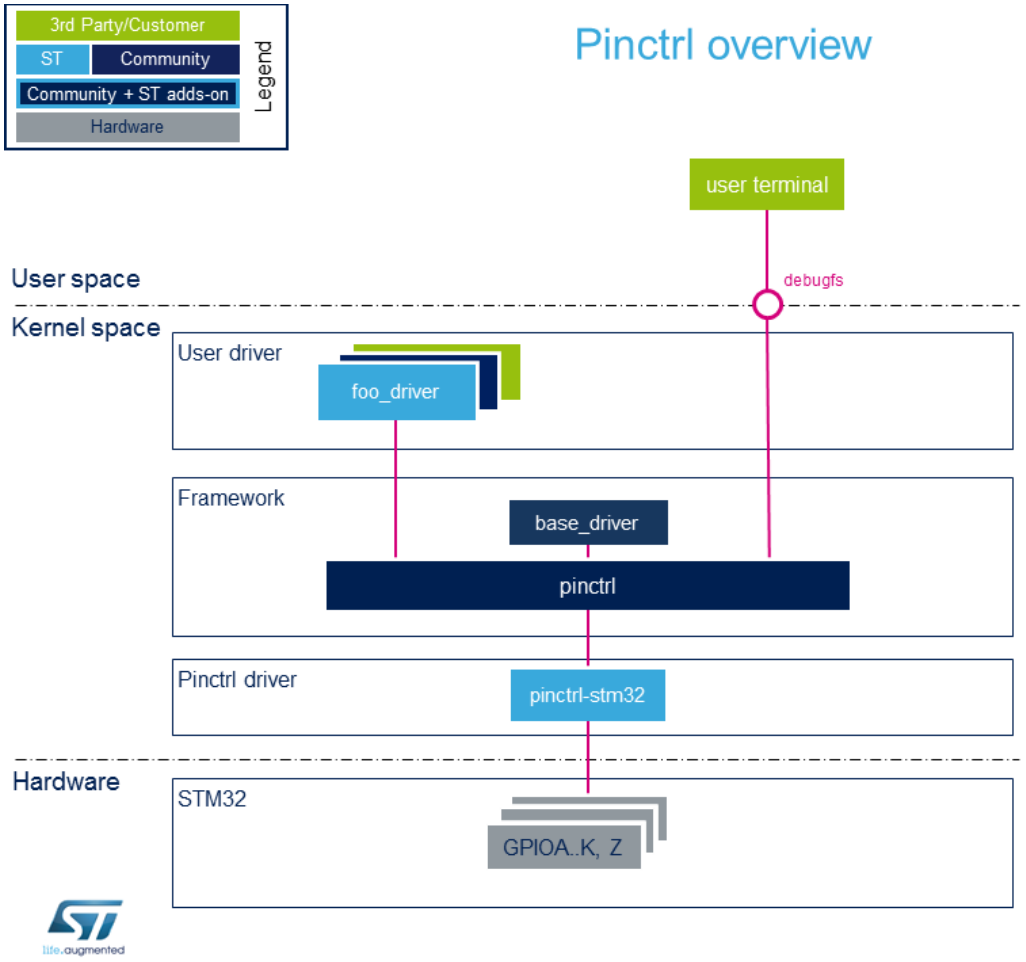
Many of the microprocessor pins (with digital I/O or analog pin type) are multiplexed between different functions: GPIO, alternate function(s).
Pinctrl framework is used to:

- 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**

foo_driver could be any driver that needs to control a GPIO