GPIO Programming Guide

Programming Guide

Customer Support

MT6000

Doc No: CS6000-AW2A-PGD-V1.1EN
Version: V1.1
Release date: 2017-7-24
Classification: internal

GPIO
Programming Guide

MediaTek Inc.

Postal address
No. 1, Dusing 1st Rd. , Hsinchu Science Park, Hsinchu City, Taiwan 30078

MTK support office address
No. 1, Dusing 1st Rd. , Hsinchu Science Park, Hsinchu City, Taiwan 30078

Internet
http://www.mediatek.com/

Document Revision History

Revision	Date	Author	Description
V1.0	2017-01-3	Xj wang	Initial Release
V1.1	2017-07-24	Xj wang	Add information for mt6763

Table of Contents

Lists of Tables

Lists of Figures

1 Introduction

1.1 Purpose

This document provides the programming guidelines for the GPIO module. It describes how to control GPIO in Linux kernel driver.

1.2 Scope

The document provides the programming details of the GPIO.

It is applying in Linux kernel-3.18 and later versions.

1.3 Who Should Read This Document

This document is primarily intended for:

- Engineers with technical knowledge of the GPIO.
- Customers who use the GPIO.

1.4 How to Use This Manual

This segment explains how information is distributed in this document, and presents some cues and examples to simplify finding and understanding information in this document. Table 1-1 presents an overview of the chapters and appendices in this document.

Table 1-1. Chapter Overview

#	Chapter	Contents
1	Introduction	Describes the scope and layout of this document.
2	References	List the references document
3	Definitions	Definitions in this document
4	Abbreviations	Abbreviations in this document
5	Overview	Brief description of this module
6	GPIO	Detail description of gpio
7	Usage guide	Linux gpio usage guide
8	examples	Provide an example about using the module
9	Frequently Asked Questions	How to get gpio current status

1.4.1 Terms and Conventions

This document uses special terms and typographical conventions to help you easily identify various information types in this document. These cues are designed to simply finding and understanding the information this document contains.

Table 1-2. Conventions

Convention	Usage	Example
[1]	Serial number of a document in the order of appearance in the References topic	Chapter 3: Pinctrl: the pin control subsystem in Linux.[2]
void xx(zz)	Source code	int gpio_setting(struct device *dev) {}
☞	Important	

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

[1] Linux kernel gpio document:

https://android.googlesource.com/kernel/mediatek/+/android-4.4.4_r3/Documentation/gpio.txt

[2] Linux kernel pinctrl document:

https://android.googlesource.com/kernel/mediatek/+/android-4.4.4_r3/Documentation/pinctrl.txt

[3] Linux devicetree usage document:

https://android.googlesource.com/kernel/mediatek/+/android-4.4.4_r3/Documentation/devicetree/usage-model.txt

3 Definitions

For the purposes of the present document, the following terms and definitions apply:

$(platform): the platform name, example: mt6757.

$(project): the project name of you, example: evb6757_64_op01.

GPIO: General Purpose Input/Output, it is a flexible software-controlled digital signal.

GPIOlib: it is an optional implementation framework making it easier for platforms to support different kinds of GPIO controller using the same programming interface.[1]

Pinctrl: the pin control subsystem in Linux.[2]

Devicetree: Linux kernel manages the device model.

4 Abbreviations

Please note the abbreviations and their explanations provided in Table 4-1. They are used in many fundamental definitions and explanations in this document and are specific to the information that this document contains.

Table 4-1. Abbreviations

Abbreviations	Explanation
MTK	MediaTek, Asia's largest fabless IC design company.
GPIO	General Purpose Input Output

5 Overview

Gpio driver has three parts: Linux pinctrl model, Linux GPIOlib model and Mediatek GPIO driver.

5.1 Architecture(before mt6763)

The Mediatek GPIO driver has three parts: pinctrl driver, GPIOlib driver and common driver.

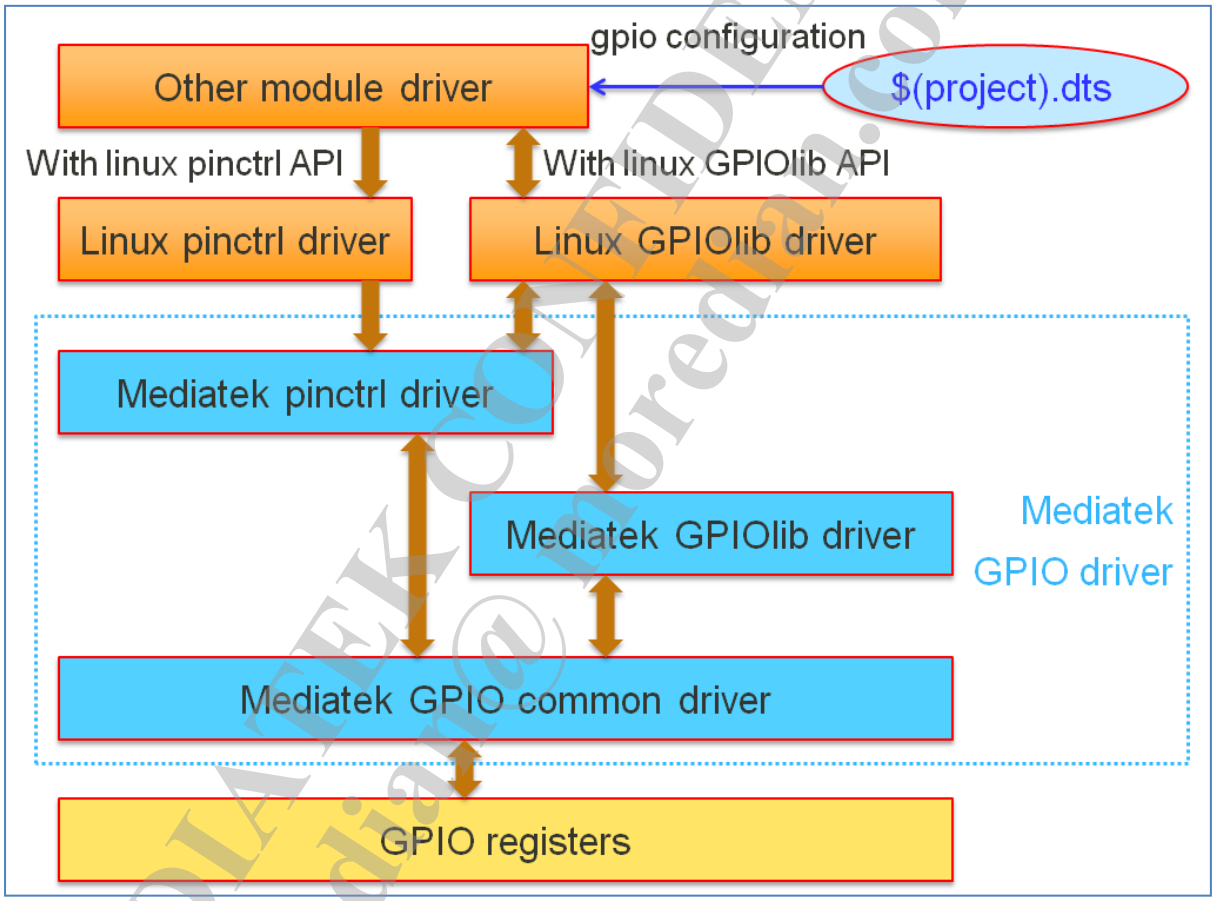

Figure 5-1. GPIO driver architecture (before mt6763)

5.2 Source Code Organization(before mt6763)

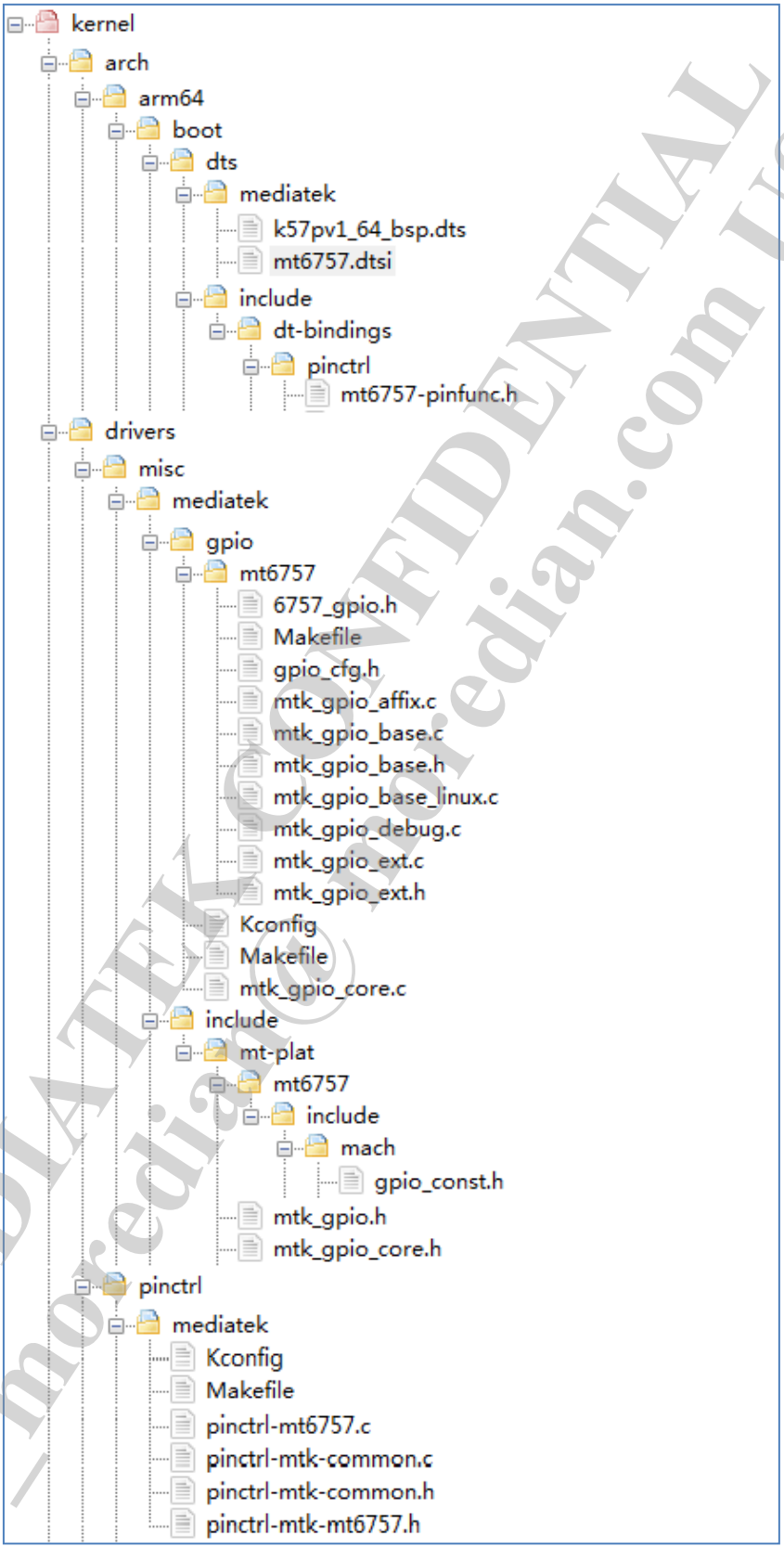

Figure 5-2. Source Code Directory Structure (before mt6763)

The description of the directories and their subdirectories is given below:

Kernel	Contains the top-level source directory
Kernel/arch/arm64/boot/dts	Contains GPIO configure in devicetree
Kernel/driver/misc/medatek/gpio	Contains Mediatek GPIOlib driver and common driver source code
Kernel/driver/pinctrl/Mediatek	Contains Mediatek pinctrl driver source code

5.3 Architecture(mt6763 and future)

The Mediatek GPIO driver has three parts: pinctrl driver, GPIOlib driver and common driver.

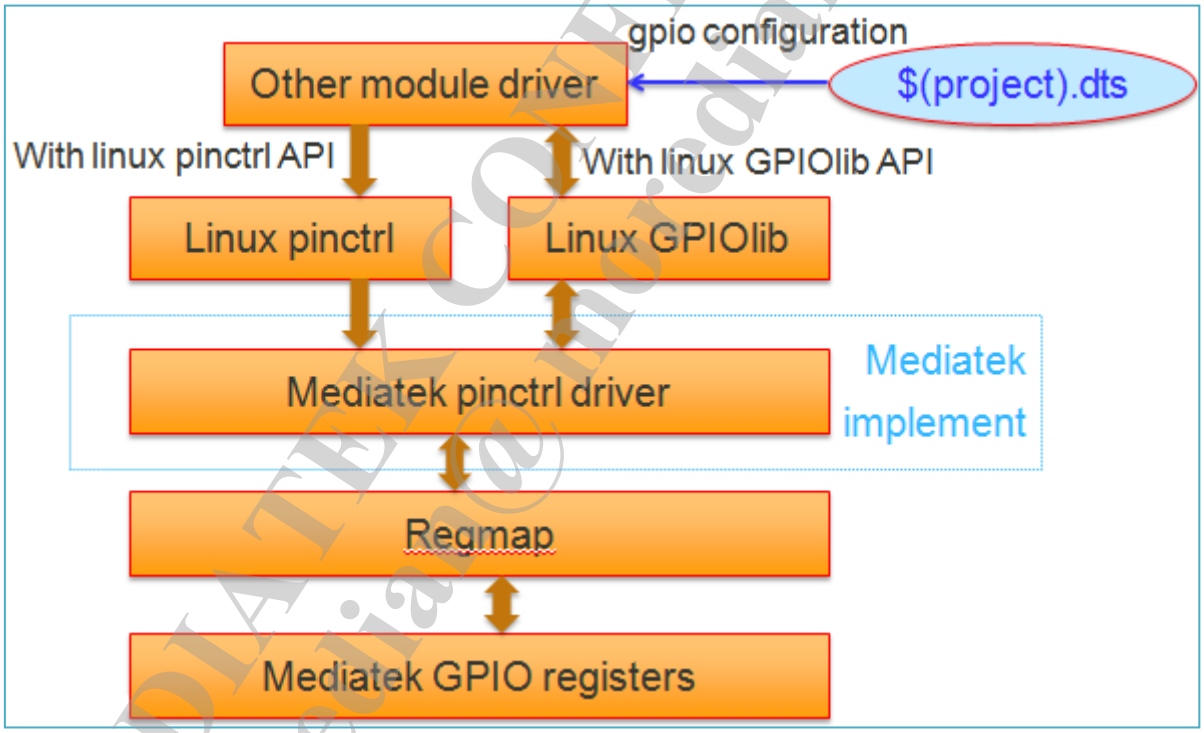

Figure 5-3. GPIO driver architecture (mt6763 and future)

5.4 Source Code Organization(mt6763 and future)

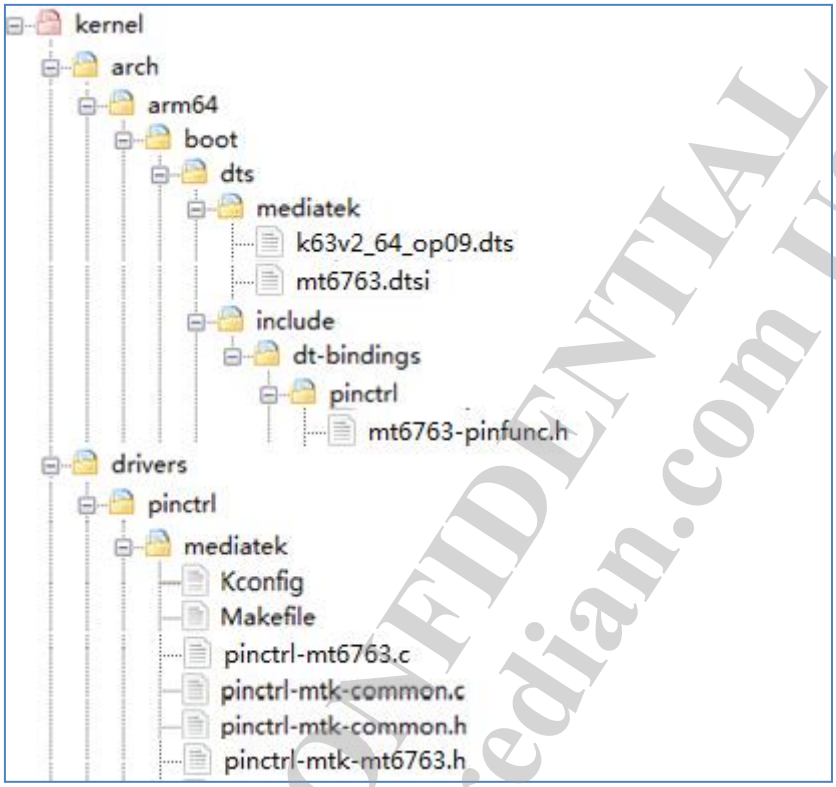

Figure 5-4. Source Code Directory Structure (mt6763 and future)

The description of the directories and their subdirectories is given below:

Kernel	Contains the top-level source directory
Kernel/arch/arm64/boot/dts	Contains GPIO configure in devicetree
Kernel/driver/pinctrl/Mediatek	Contains Mediatek pinctrl driver source code

6 GPIO

6.1 Data Structure

This part introduction the data structure of Mediatek GPIO driver.

6.1.1 mtk_pinctrl structure

The mtk_pinctrl structure describes the chip pin controller information. It has the following header-file definition:

```
struct mtk_pinctrl {
        struct regmap           *regmap1;
        struct regmap           *regmap2;
        struct pinctrl_desc pctl_desc;
        struct device         *dev;
        struct gpio_chip        *chip;
        struct mtk_pinctrl_group          *groups;
        unsigned                          ngroups;
        const char        **grp_names;
        struct pinctrl_dev     *pctl_dev;
        const struct mtk_pinctrl_devdata  *devdata;
        void __iomem                      *eint_reg_base;
        struct irq_domain       *domain;
        int                               *eint_dual_edges;
        u32 *wake_mask;
        u32 *cur_mask;
};
```

The **regmap1** field contains registor information of pin controller on chip.

The **regmap2** field is reserved.

The **ptcl_desc** field is pin controller descriptor for Linux pin control subsystem.

The **dev** field is the basic device structure of Linux device subsystem.

The **chip** field is the abstract a GPIO controller of Linux GPIOlib subsystem.

The **groups** field describe the pin group information.

The **ngroups** field is the count of pin group on this chip.

The **grp_names** field contains all group name of pin controller.

The **pctl_dev** field is pin control class device of Linux pin control subsystem.

The **devdata** field is the specific information of pin controller on this chip.

The **eint_reg_base** field is eint controller register base address on this chip.

The **domain** field is the eint controller domain.

The **eint_dual_edges** field describe this eint pin whether setting both trigger.

The **wake_mask** field describe this eint pin whether setting ability of wake up system.

The **cur_mask** field describe this eint pin whether enabled.

6.1.2 mtk_pinctrl_group structure

The mtk_pinctrl_group structure describe the group information of the pin controller. It has the following header-file definition:

```
struct mtk_pinctrl_group {
        const char *name;
        unsigned long          config;
        unsigned   pin;
};
```

The **name** field is the group name.

The **config** field is the group configure information.

The **pin** field is the group number.

6.1.3 mtk_pinctrl_devdata structure

The mtk_pinctrl_devdata structure describe the pin information and control call back function of the pin controller. It has the following header-file definition:

```
struct mtk_pinctrl_devdata {
        const struct mtk_desc_pin         *pins;
        unsigned int                                        npins;
        const struct mtk_drv_group_desc              *grp_desc;
        unsigned int              n_grp_cls;
        const struct mtk_pin_drv_grp     *pin_drv_grp;
        unsigned int              n_pin_drv_grps;
        int (*spec_pull_set)(struct regmap *reg, unsigned int pin,
                              unsigned char align, bool isup, unsigned int arg);
```

```
int (*spec_ies_smt_set)(struct regmap *reg, unsigned int pin,
                        unsigned char align, int value, enum pin_config_param arg);
void (*spec_pinmux_set)(struct regmap *reg, unsigned int pin,
                        unsigned int mode);
void (*spec_dir_set)(unsigned int *reg_addr, unsigned int pin);
int (*spec_pull_get)(struct regmap *reg, unsigned int pin);
int (*spec_ies_get)(struct regmap *reg, unsigned int pin);
int (*spec_smt_get)(struct regmap *reg, unsigned int pin);
int (*spec_set_gpio_mode)(unsigned long pin, unsigned long mode);
int (*mt_set_gpio_dir)(unsigned long pin, unsigned long dir);
int (*mt_get_gpio_dir)(unsigned long pin);
int (*mt_get_gpio_out)(unsigned long pin);
int (*mt_set_gpio_out)(unsigned long pin, unsigned long output);
int (*mt_set_gpio_driving)(unsigned long pin, unsigned long strength);
int (*mt_get_gpio_in)(unsigned long pin);
int (*mt_set_gpio_ies)(unsigned long pin, unsigned long enable);
int (*mt_set_gpio_smt)(unsigned long pin, unsigned long enable);
int (*mt_set_gpio_slew_rate)(unsigned long pin, unsigned long enable);
int (*mt_set_gpio_pull_enable)(unsigned long pin, unsigned long enable);
int (*mt_set_gpio_pull_select)(unsigned long pin, unsigned long select);
int (*mt_set_gpio_pull_resistor)(unsigned long pin, unsigned long resistors);
unsigned int dir_offset;
unsigned int ies_offset;
unsigned int smt_offset;
unsigned int pullen_offset;
unsigned int pullsel_offset;
unsigned int drv_offset;
unsigned int dout_offset;
unsigned int din_offset;
unsigned int pinmux_offset;
unsigned short type1_start;
unsigned short type1_end;
unsigned char  port_shf;
unsigned char  port_mask;
unsigned char  port_align;
```

```
    struct mtk_eint_offsets eint_offsets;
    unsigned int        ap_num;
    unsigned int        db_cnt;
};
```

The **pins** field is the pin information of pin controller on this chip.

The **npins** field is the pin count of pin controller.

The **grp_desc** field is the group driving strength information, reserved.

The **n_grp_cls** field is reserved.

The **pin_drv_grp** field is the pin driving strength information in a group, reserved.

The **n_pin_drv_grps** field is reserved.

The **spec_pull_set** field is call back function for setting pull function of special pin.

The **spec_ies_smt_set** field is call back function for setting input enable and Schmitt function of special pin.

The **spec_pinmux_set** field is call back function for setting pin mux mode of special pin. Reserved.

The **spec_dir_set** field is call back function for setting direction of special pin.

The **spec_pull_get** field is call back function for getting pull function of special pin.

The **spec_ies_get** field is call back function for getting input enable configure of special pin.

The **spec_smt_get** field is call back function for getting Schmitt trigger configure of special pin.

The **spec_set_gpio_mode** field is call back function for setting pin mux mode of special pin.

The **mt_set_gpio_dir** field is call back function for setting direction of general pin.

The **mt_get_gpio_dir** field is call back function for getting direction configure of general pin.

The **mt_get_gpio_out** field is call back function for getting output value configure of general pin.

The **mt_set_gpio_out** field is call back function for setting output of general pin.

The **mt_set_gpio_driving** field is call back function for setting driving strength of general pin.

The **mt_get_gpio_in** field is call back function for getting input value.

The **mt_set_gpio_ies** field is call back function for setting input enable of general pin.

The **mt_set_gpio_smt** field is call back function for setting Schmitt trigger of general pin.

The **mt_set_gpio_slew_rate** field is reserved.

The **mt_set_gpio_pull_enable** field is call back function for setting pull function enable of general pin.

The **mt_set_gpio_pull_select** field is call back function for setting pull up/down selection of general pin.

The **mt_set_gpio_pull_resistor** field is reserved.

The **dir_offset** field is reserved.

The **ies_offset** field is reserved.

The **smt_offset** field is reserved.

The **pullen_offset** field is reserved.

The **pullsel_offset** field is reserved.

The **drv_offset** field is reserved.

The **dout_offset** field is reserved.

The **din_offset** field is reserved.

The **pinmux_offset** field is reserved.

The **type1_start** field is reserved.

The **type1_end** field is reserved.

The **port_shf** field is reserved.

The **port_mask** field is reserved.

The **port_align** field is reserved.

The **eint_offsets** field is reserved.

The **ap_num** field is reserved.

The **db_cnt** field is reserved.

6.2 Variables

There is no special variable in Mediatek GPIO driver.

6.3 Functions

This part introduces Linux pinctrl function, Linux GPIOlib function and Mediatek GPIO common driver function.

6.3.1 Linux pinctrl function

These functions are Linux pinctrl driver functions. These are provided by Linux in head file: Linux/pinctrl/consumer.h . So, you must include this head file when you use these functions.

Function #1: **devm_pinctrl_get**

The main purpose of the devm_pinctrl_get is getting the pinctrl structure of this device.

- Definition

struct pinctrl *devm_pinctrl_get(struct device *dev);

- Parameters

Table 6-1. Parameters of devm_pinctrl_get

Parameters	Direction (IN/OUT)	Description
dev	IN	the device to obtain the handle for pinctrl

- Return Values

  On success, devm_pinctrl_get () returns a pinctrl structure handle. On errors, returns NULL.

Function #2: **pinctrl_lookup_state**

The main purpose of the pinctrl_lookup_state is retrieving a state handle from a pinctrl handle.

- Definition

  struct pinctrl_state *pinctrl_lookup_state(struct pinctrl *p, const char *name);

- Parameters

Table 6-2. Parameters of pinctrl_lookup_state

Parameters	Direction (IN/OUT)	Description
p	IN	the pinctrl handle to retrieve the state from
name	IN	the state name to retrieve

- Return Values

  On success, pinctrl_lookup_state () returns a pinctrl_state structure handle. On errors, returns NULL.

Function #3: **pinctrl_select_state**

The main purpose of the pinctrl_select_state is selecting/activating a pinctrl state to hardware.

- Definition

  int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *state);

- Parameters

Table 6-3. Parameters of pinctrl_select_state

Parameters	Direction (IN/OUT)	Description
p	IN	the pinctrl handle for the device that requests configuration
state	IN	the state handle to select/activate

- Return Values

  On success, pinctrl_select_state () returns a 0. On errors, returns a negative value.

If you want to get more pinctrl function, please see the head file: Linux/pinctrl/consumer.h

6.3.2 Linux GPIOlib function

These functions are Linux GPIO driver functions. These are provided by Linux in head file: Linux/gpio.h or asm-generic/gpio.h . So, you must include head file: Linux/gpio.h when you use these functions.

Function #1: **gpio_request**

The main purpose of the gpio_request is to request a gpio for operation.

- Definition

  int gpio_request(unsigned gpio, const char *label);

- Parameters

Table 6-4. Parameters of gpio_request

Parameters	Direction (IN/OUT)	Description
Gpio	IN	The gpio number that will be requested
label	IN	The label for this request

- Return Values

  On success, gpio_request () returns a 0. On errors, returns a negative value.

Function #2: **gpio_free**

The main purpose of the gpio_free is to free a gpio resource.

- Definition

  void gpio_free(unsigned gpio);

- Parameters

Table 6-5. Parameters of gpio_free

Parameters	Direction (IN/OUT)	Description
gpio	IN	The gpio number that will be free

- Return Values

  No return value.

Function #3: **gpio_direction_input**

The main purpose of the gpio_direction_input is to configure the gpio direction as input.

- Definition

  int gpio_direction_input(unsigned gpio);

- Parameters

Table 6-6. Parameters of gpio_direction_input

Parameters	Direction (IN/OUT)	Description
gpio	IN	The gpio number that will be operated

- Return Values
  On success, gpio_direction_input () returns a 0. On errors, returns a negative value.

Function #4: **gpio_direction_output**

The main purpose of the gpio_direction_output is to configure a gpio as output and set the output value.

- Definition

  int gpio_direction_output(unsigned gpio, int value);

- Parameters

Table 6-7. Parameters of gpio_direction_output

Parameters	Direction (IN/OUT)	Description
gpio	IN	The gpio number that will be operated
value	IN	The output value of the gpio

- Return Values
  On success, gpio_direction_output () returns a 0. On errors, returns a negative value.

Function #5: **gpio_set_value**

The main purpose of the gpio_set_value is to set the output value of gpio.

- Definition

  void gpio_set_value(unsigned int gpio, int value);

- Parameters

Table 6-8. Parameters of gpio_set_value

Parameters	Direction (IN/OUT)	Description
gpio	IN	The gpio number that will be operated
value	IN	The output value of the gpio

- Return Values
  No return value.

Function #6: **gpio_get_value**

The main purpose of the gpio_get_value is to get the input value of this gpio.

- Definition

  int gpio_get_value(unsigned int gpio);

- Parameters

Table 6-9. Parameters of gpio_get_value

Parameters	Direction (IN/OUT)	Description
gpio	IN	The gpio number that will be operated

- Return Values
  On success, gpio_get_vlaue () returns a 0 or 1, it is the gpio input value. On errors, returns a negative value.

Function #7: **of_get_named_gpio**

The main purpose of the of_get_named_gpio is to get a GPIO number to use with GPIO API . You must include the head file: Linux/of_gpio.h if you want to use it.

- Definition

  int of_get_named_gpio(struct device_node *np, const char *propname, int index);

- Parameters

Table 6-10. Parameters of of_get_named_gpio

Parameters	Direction (IN/OUT)	Description
np	IN	device node to get GPIO from
propname	IN	Name of property containing gpio specifier(s)
index	IN	index of the GPIO

- Return Values
  On success, of_get_named_gpio () returns GPIO number to use with Linux generic GPIO API. On errors, returns a negative value.

6.3.3 Mediatek GPIO common driver function(before mt6763)

These functions are Mediatek GPIO driver function, it is used by Mediatek pinctrl driver and Mediatek GPIOlib driver. Generally, user cannot use these functions.

Mt6763 and future don't have these GPIO API.

Function #1: **mt_set_gpio_mode**

The main purpose of the mt_set_gpio_mode is setting the pin mux of pin.

- Definition

  int mt_set_gpio_mode(unsigned long pin, unsigned long mode);

- Parameters

Table 6-11. Parameters of mt_set_gpio_mode

Parameters	Direction (IN/OUT)	Description

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
mode	IN	The pin mux mode that will be setting

- Return Values
  On success, mt_set_gpio_mode () returns a 0. On errors, returns a negative value.

Function #2: **mt_get_gpio_mode**

The main purpose of the mt_get_gpio_mode is getting the pin mux selection of pin.

- Definition

  int mt_get_gpio_mode(unsigned long pin);

- Parameters

Table 6-12. Parameters of mt_get_gpio_mode

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values
  On success, mt_get_gpio_mode () returns a positive value , it is current pin mux mode. On errors, returns a negative value.

Function #3: **mt_set_gpio_dir**

The main purpose of the mt_set_gpio_dir is setting the pin direction(input or output).

- Definition

  int mt_set_gpio_dir(unsigned long pin, unsigned long dir);

- Parameters

Table 6-13. Parameters of mt_set_gpio_dir

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
dir	IN	The pin direction that will be setting.(0: input; 1: output)

- Return Values
  On success, mt_set_gpio_dir () returns a 0. On errors, returns a negative value.

Function #4: **mt_get_gpio_dir**

The main purpose of the mt_get_gpio_dir is getting the direction of pin.

- Definition

  int mt_get_gpio_dir(unsigned long pin);

- Parameters

Table 6-14. Parameters of mt_get_gpio_dir

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values
  On success, mt_get_gpio_dir () returns a 0 or 1, it is current pin direction, 0: input and 1: output. On errors, returns a negative value.

Function #5: **mt_set_gpio_out**

The main purpose of the mt_set_gpio_out is setting the pin output value. It is just valid when pin direction is output.

- Definition

  int mt_set_gpio_out(unsigned long pin, unsigned long output);

- Parameters

Table 6-15. Parameters of mt_set_gpio_out

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
output	IN	The pin output value that will be setting.(0: low; 1: high)

- Return Values
  On success, mt_set_gpio_out () returns a 0. On errors, returns a negative value.

Function #6: **mt_get_gpio_out**

The main purpose of the mt_get_gpio_out is getting the current output value of pin.

- Definition

  int mt_get_gpio_out(unsigned long pin);

- Parameters

Table 6-16. Parameters of mt_get_gpio_out

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values
  On success, mt_get_gpio_out () returns a 0 or 1, it is the current output value, 0: low and 1:high. On errors, returns a negative value.

Function #7: **mt_get_gpio_in**

The main purpose of the mt_get_gpio_in is getting current input value of pin.

- Definition

  int mt_get_gpio_in(unsigned long pin);

- Parameters

Table 6-17. Parameters of mt_get_gpio_in

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values

  On success, mt_get_gpio_mode () returns a 0 or 1, it is the current input value, 0: low and 1:high. On errors, returns a negative value.

Function #8: **mt_set_gpio_pull_enable**

The main purpose of the mt_set_gpio_pull_enable is to enable or disable pull function of pin.

- Definition

  int mt_set_gpio_pull_enable(unsigned long pin, unsigned long enable);

- Parameters

Table 6-18. Parameters of mt_set_gpio_pull_enable

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
enable	IN	Enable or disable pull function.(0: disable; 1: enable)

- Return Values

  On success, mt_set_gpio_pull_enable () returns a 0. On errors, returns a negative value.

Function #9: **mt_get_gpio_pull_enable**

The main purpose of the mt_get_gpio_pull_enable is getting the current status of pin pull function.

- Definition

  int mt_get_gpio_pull_enable(unsigned long pin);

- Parameters

Table 6-19. Parameters of mt_get_gpio_pull_enable

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values

On success, mt_get_gpio_pull_enable () returns a 0 or 1, it is current status of pull function, 0: disabled and 1: enabled. On errors, returns a negative value.

Function #10: **mt_set_gpio_pull_select**

The main purpose of the mt_set_gpio_pull_select is setting the pull direction of pin.(pull up or pull down). It is just valid when pull function is enabled.

- Definition

  int mt_set_gpio_pull_select(unsigned long pin, unsigned long select);

- Parameters

Table 6-20. Parameters of mt_set_gpio_pull_select

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
select	IN	Pull direction that will be setting. (0: pull down; 1: pull up)

- Return Values

  On success, mt_set_gpio_pull_select () returns a 0. On errors, returns a negative value.

Function #11: **mt_get_gpio_pull_select**

The main purpose of the mt_get_gpio_pull_select is getting the pull direction of pin.(pull up or pull down). Definition

int mt_get_gpio_pull_select(unsigned long pin);

- Parameters

Table 6-21. Parameters of mt_get_gpio_pull_select

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values

  On success, mt_get_gpio_pull_select () returns a 0 or 1, it is pull direction, 0: pull down and 1: pull up. On errors, returns a negative value.

Function #12: **mt_set_gpio_ies**

The main purpose of the mt_set_gpio_ies is to enable or disable input function of pin.(enable or disable).

- Definition

  int mt_set_gpio_ies(unsigned long pin, unsigned long enable);

- Parameters

Table 6-22. Parameters of mt_set_gpio_ies

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
enable	IN	Enable or disable input function. (0: disable; 1: enable)

- Return Values
  On success, mt_set_gpio_ies () returns a 0. On errors, returns a negative value.

Function #13: **mt_get_gpio_ies**

The main purpose of the mt_get_gpio_ies is getting the input function status of pin.(enabled or disabled).

- Definition

  int mt_get_gpio_ies(unsigned long pin);

- Parameters

Table 6-23. Parameters of mt_get_gpio_ies

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values
  On success, mt_get_gpio_ies () returns a 0 or 1, it is the current input function status, 0: disabled or 1: enabled. On errors, returns a negative value.

Function #14: **mt_set_gpio_smt**

The main purpose of the mt_set_gpio_smt is to enable or disable Schmitt trigger function of pin.(enable or disable).

- Definition

  int mt_set_gpio_smt(unsigned long pin, unsigned long enable);

- Parameters

Table 6-24. Parameters of mt_set_gpio_smt

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated
enable	IN	Enable or disable Schmitt trigger function. (0: disable; 1: enable)

- Return Values
  On success, mt_set_gpio_smt () returns a 0. On errors, returns a negative value.

Function #15: **mt_get_gpio_smt**

The main purpose of the mt_get_gpio_smt is getting the Schmitt trigger function status of pin.(enabled or disabled).

- Definition

  int mt_get_gpio_smt(unsigned long pin);

- Parameters

Table 6-25. Parameters of mt_get_gpio_smt

Parameters	Direction (IN/OUT)	Description
pin	IN	The pin number that will be operated

- Return Values

  On success, mt_get_gpio_smt () returns a 0 or 1, it is current Schmitt trigger function status, 0: disabled and 1: enabled. On errors, returns a negative value.

7 Usage guide

7.1 Pinctrl usage guide

Generally, pinctrl is used to control gpio. It has more functions than GPIOlib, but it cannot get this gpio input value.

7.1.1 Pinctrl node setting in devicetree

There are three parts setting about pinctrl in devicetree: pin controller node, pinctrl node and device node.

Pin controller node describes the gpio controller information of this chip. It is configured in file: $(platform).dtsi by Mediatek correctly and user does not need to modify it.

For example: (mt6757.dtsi)

```
pio: pinctrl@10005000 {
        compatible = "mediatek,mt6757-pinctrl";
        reg = <0 0x10005000 0 0x1000>;
        mediatek,pctl-regmap = <&syscfg_pctl_a>;
        pins-are-numbered;
        gpio-controller;
        #gpio-cells = <2>;
};
```

Pinctrl node describes the specific pin information that you want to configure. It is sub node of pin controller node. Generally, it is configured in file: $(project).dts by user.

For example: (evb6757_64.dts)

```
/* sensor gpio standization */
&pio {
        alsps_intpin_cfg: alspspincfg {
                pins_cmd_dat {
                        pins = <PINMUX_GPIO6__FUNC_GPIO6>;
                        slew-rate = <0>;
                        bias-pull-up = <00>;
                };
        };
        alsps_intpin_default: alspsdefaultcfg {
        };
};
```

Device node describes the device information that wants to control gpio. Generally, it is configured in file: $(project).dts by user.

For example: (evb6757_64.dts)

```
&als {
        pinctrl-names = "pin_default", "pin_cfg";
        pinctrl-0 = <&alsps_intpin_default>;
        pinctrl-1 = <&alsps_intpin_cfg>;
        status = "okay";
};
```

7.1.2 pinctrl node format in devicetree

Format in device node.

It contains two parts: pinctrl-names and pinctrl-xx (xx is a number).

```
&als {
        pinctrl-names = "pin_default", "pin_cfg";
        pinctrl-0 = <&alsps_intpin_default>;
        pinctrl-1 = <&alsps_intpin_cfg>;
        status = "okay";
};
```

pinctrl-names configure the pinctrl state name. It is used to match the pinctrl node setting and for search by driver code.

pinctrl-names can contain multiple name, every name match a pinctrl-xx. The order is pinctrl-names from left to right match pinctrl-0, pinctrl-1,….

pinctrl-xx is the pinctrl state, its value point the pinctrl node label.

Format in pinctrl node as below.

```
&pio {
        label: name {
                pins_cmd_dat {
                        pin information
                };
        };
};
```

Pio: the pin controller label.

Label: this pinctrl node label. It is used by other node.

Name: this pinctrl node name.

Pins_cmd_dat: one gpio setting node.

Pin information: one gpio specific setting.

Table 7-1. pin information can be configured

property	Description	example
pins	The pin number and pin mux mode	pins = <PINMUX_GPIO6__FUNC_GPIO6>;
slew-rate	Gpio direction,(0:input; 1:output)	slew-rate = <0>;
bias-disable	Pull function disabled	bias-disable;
bias-pull-down	Pull down, (property value is reserved)	bias-pull-down = <00>;
bias-pull-up	Pull up, (property value is reserved)	bias-pull-up = <00>;
output-high	Output high,(it is valid when slew-rate = <1>)	output-high;
output-low	Output low,(it is valid when slew-rate = <1>)	output-low;
input-enable	Enable input function, (ies = 1)	input-enable;
input-disable	Disable input function, (ies = 0)	input-disable;
input-schmitt-enable	Schmitt function setting, (0:disable; 1: enable)	input-schmitt-enable = <0>;

For example1: configure gpio6 is gpio mode and pull up.

```
&pio {
        alsps_intpin_cfg: alspspincfg {
                pins_cmd_dat {
                        pins = <PINMUX_GPIO6__FUNC_GPIO6>;
                        slew-rate = <0>;
                        bias-pull-up = <00>;
                };
        };
};
```

For example2: configure gpio6 is gpio mode and output low.

```
&pio {
        alsps_intpin_cfg: alspspincfg {
                pins_cmd_dat {
                        pins = <PINMUX_GPIO6__FUNC_GPIO6>;
                        slew-rate = <1>;
                        output-low;
                };
        };
};
```

One pinctrl state can contain several pinctrl nodes. For example:

```
test {
        compatible = "test";
        pinctrl-names = "active";
        pinctrl-0 = <&i2c_active &int_active>;
};
```

One pinctrl node can contain several pin setting. For example:

```
&pio {
    i2c_active: i2c_active {
        pins_cmd_dat1 {
            pins = <PINMUX_GPIO93__FUNC_SCL0>;
            slew-rate = <0>;
            bias-disable;
        };
        pins_cmd_dat2 {
            pins = <PINMUX_GPIO92__FUNC_SDA0>;
            slew-rate = <0>;
            bias-disable;
        };
    };
    int_active: int_active {
        pins_cmd_dat {
            pins = <PINMUX_GPIO11__FUNC_GPIO11>;
            slew-rate = <0>;
            bias-pull-up = <00>;
        };
    };
};
```

7.1.3 pinctrl usage in driver code

There are three steps for use pinctrl in driver code. Get pinctrl, look up the pinctrl state and select pinctrl state.

Step 1: get pinctrl in device node.

Use pinctrl API: **devm_pinctrl_get**() obtain pinctrl from device node.

Step 2: look up pinctrl state in pinctrl.

Use pinctrl API: **pinctrl_lookup_state**() obtain pinctrl state from pinctrl. The pinctrl state name need match the pinctrl-names in device node.

Step 3: select pinctrl state to set the gpio hardware.

Use pinctrl API: **pinctrl_select_state**() to set the gpio hardware.

For example:

```
int gpio_setting(struct device *dev)
{
	int ret = 0;
	struct pinctrl *pinctrl;
	struct pinctrl_state *pins_default;
	struct pinctrl_state *pins_cfg;
	pinctrl = devm_pinctrl_get(dev); /*step1: get pinctrl from device node */
	if (IS_ERR(pinctrl)) {
		ret = PTR_ERR(pinctrl);
		return ret;
	}
	pins_default = pinctrl_lookup_state(pinctrl, "pin_default"); /*step2: look up pinctrl state */
	if (IS_ERR(pins_default)) {
		ret = PTR_ERR(pins_default);
	}
	pins_cfg = pinctrl_lookup_state(pinctrl, "pin_cfg");  /*step2: look up pinctrl state */
	if (IS_ERR(pins_cfg)) {
		ret = PTR_ERR(pins_cfg);
	} else
		pinctrl_select_state(pinctrl, pins_cfg);  /*step3: select pinctrl state to set gpio hardware*/
	return ret;
}
```

7.2 GPIOlib usage guide

GPIOlib is Linux GPIO subsystem. It provides some API to control GPIO. Generally, there are four steps for use GPIOlib API:

Step1: get gpio number.

Step2: request gpio.

Step3: control gpio.

Step4: free gpio.

7.2.1 Get gpio number

Generally, we configure the gpio number in devicetree, and get the gpio number in driver code.

In devicetree, you could add a property for configure gpio num. The format is:

```
property = <&pio_label gpionumber flags>;
```

For example:

```
gpiolib_test: gpio_test {
        compatible = "gpiolib_usage_test";
        gpio_num = <&pio 11 0>;          /*gpio number configure*/
        status = "okay";
};
```

In driver code, you could use API: **of_get_named_gpio**() to get the gpio number that is configured in devicetree.

For example:

```
#include <Linux/of_gpio.h>
int get_gpio_number(void)
{
        struct device_node *node = NULL;
        int gpio = 0;
        node = of_find_compatible_node(NULL, NULL, "gpiolib_usage_test");
        if (node == NULL) {
                pr_err("%s: node: gpiolib_usage_test  not find!\n", __func__);
                return -EFAULT;
        }
        gpio = of_get_named_gpio(node, "gpio_num", 0);
        return gpio;
}
```

7.2.2 Request gpio

Generally, you could use API: **gpio_request**() to request gpio.

For example:

```
ret = gpio_request(gpio, "gpiolib_test");
```

7.2.3 Control gpio

You could control gpio according to your ideas with GPIOlib API.

For example:

```
gpio_direction_output(gpio, 0);   /*set gpio as output pin*/
gpio_set_value(gpio, 1);          /*set gpio output high*/
gpio_direction_input(gpio);       /*set gpio as input pin*/
value = gpio_get_value(gpio);     /*get gpio input value*/
```

7.2.4 Free gpio

You should free gpio with API: **gpio_free**() when you no longer use it.

For example:

```
gpio_free(gpio);
```

8 Examples

8.1 Use pinctrl control GPIO to generate PWM waveform.

This example is using pinctrl control gpio251 to generate PWM waveform.

8.1.1 Devicetree setting and driver code

Devicetree setting: add the below code in $(project).dts.

```
gpio_pwm{
        compatible = "gpio_pwm";
        pinctrl-names = "output_low", "output_high";
        pinctrl-0 = <&gpio_output_low>;
        pinctrl-1 = <&gpio_output_high>;
        status = "okay";
};
&pio {
        gpio_output_low:output_low {
                pin_cmd_dat {
                        pins = <PINMUX_GPIO251__FUNC_GPIO251>;
                        slew-rate = <1>;
                        output-low;
                };
        };
        gpio_output_high:output_high {
                pin_cmd_dat {
                        pins = <PINMUX_GPIO251__FUNC_GPIO251>;
                        slew-rate = <1>;
                        output-high;
                };
        };
};
```

Driver code: gpio_pwm_pinctrl.c

```
#include <Linux/module.h>
#include <Linux/platform_device.h>
#include <Linux/of.h>
#include <Linux/delay.h>
#include <Linux/kthread.h>
#include <Linux/pinctrl/consumer.h>

static struct task_struct *gpio_pwm_task;
static struct pinctrl *pctrl;
static struct pinctrl_state *output_low;
static struct pinctrl_state *output_high;
#define PERIOD 1000
static int gpio_pwm_thread(void *data)
{
        if (IS_ERR(pctrl)) {
                pr_err("%s pctrl is NULL!\n", __func__);
                return 0;
        }
        while (1) {
                if (!IS_ERR(output_low)) {
                        /*control GPIO hardware*/
                        pinctrl_select_state(pctrl, output_low);
                }
                msleep(PERIOD / 2);
                if (!IS_ERR(output_high)) {
                        pinctrl_select_state(pctrl, output_high);
                }
                msleep(PERIOD / 2);
        }
        return 0;
}
static int gpio_pwm_probe(struct platform_device *pdev)
{
        int ret = 0;
```

```
	/*get pinctrl from device*/
	pctrl = devm_pinctrl_get(&pdev->dev);
	if (IS_ERR(pctrl)) {
		pr_err("%s devm_pinctrl_get failed!\n", __func__);
		ret = -EFAULT;
		goto out;
	}
	/*look up pinctrl state: output_low*/
	output_low = pinctrl_lookup_state(pctrl, "output_low");
	if (IS_ERR(output_low)) {
		pr_err("%s pinctrl_lookup_state ouput_low failed!\n", __func__);
		ret = -EFAULT;
		goto out_pctrl;
	}
	/*look up pinctrl state: output_high*/
	output_high = pinctrl_lookup_state(pctrl, "output_high");
	if (IS_ERR(output_low)) {
		pr_err("%s pinctrl_lookup_state ouput_high failed!\n", __func__);
		ret = -EFAULT;
		goto out_pctrl;
	}
	gpio_pwm_task = kthread_run(gpio_pwm_thread, NULL, "gpio_pwm_thread");
	if (IS_ERR(gpio_pwm_task)) {
		pr_err("%s kthread_run failed!\n", __func__);
		ret = -EFAULT;
		goto out_pctrl;
	}
	return 0;

out_pctrl:
	devm_pinctrl_put(pctrl);
	pctrl = NULL;
out:
	return ret;
}
```

```
static int gpio_pwm_remove(struct platform_device *pdev)
{
	if (!IS_ERR(gpio_pwm_task)) {
		kthread_stop(gpio_pwm_task);
		gpio_pwm_task = NULL;
	}
	if (!IS_ERR(pctrl)) {
		devm_pinctrl_put(pctrl);
		pctrl = NULL;
		output_high = NULL;
		output_low = NULL;
	}
	return 0;
}

#if defined(CONFIG_OF)
static const struct of_device_id gpio_pwm_of_match[] = {
	{ .compatible = "gpio_pwm",	},
	{},
};
#endif

static struct platform_driver gpio_pwm_driver = {
	.probe = gpio_pwm_probe,
	.remove = gpio_pwm_remove,
	.driver = {
		.name = "gpio_pwm",
#if defined(CONFIG_OF)
		.of_match_table = gpio_pwm_of_match,
#endif
	}
};

static int __init gpio_pwm_init(void)
{
```

```
        return platform_driver_register(&gpio_pwm_driver);
}
static void __exit gpio_pwm_exit(void)
{
        platform_driver_unregister(&gpio_pwm_driver);
}
module_init(gpio_pwm_init);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("gpio_pwm_dirver");
MODULE_AUTHOR("Mediatek");
```

8.2 Use GPIOlib control GPIO to generate PWM waveform

This example is using GPIOlib API control gpio251 to generate PWM waveform.

8.2.1 Devicetree setting and driver code

Devicetree setting: add the below code in $(project).dts.

```
        gpio_pwm{
                compatible = "gpio_pwm";
                gpio_num = <&pio 251 0>;
                status = "okay";
        };
```

Driver code: (gpio_pwm_gpiolib.c)

```
        #include <Linux/module.h>
        #include <Linux/platform_device.h>
        #include <Linux/of.h>
        #include <Linux/delay.h>
        #include <Linux/kthread.h>
        #include <Linux/gpio.h>
        #include <Linux/of_gpio.h>

        static struct task_struct *gpio_pwm_task;
        static int gpio_num;
        #define PERIOD 500
```

```
static int gpio_pwm_thread(void *data)
{
	int ret = 0;

	ret = gpio_request(gpio_num, "gpio_pwm");		/*request gpio*/
	if (ret < 0) {
		pr_err("%s gpio_request failed, gpio=%d\n", __func__, gpio_num);
		return 0;
	}
	gpio_direction_output(gpio_num, 0);		/*set direction is output*/
	while (1) {
		gpio_set_value(gpio_num, 0);	/*output 0*/
		msleep(PERIOD / 2);
		gpio_set_value(gpio_num, 1);	/*output 1*/
		msleep(PERIOD / 2);
	}
	gpio_free(gpio_num); /*free gpio*/
	return 0;
}

static int gpio_pwm_probe(struct platform_device *pdev)
{
	int ret = 0;
	struct device_node *node = NULL;

	/*get gpio num*/
	if (IS_ERR(pdev->dev.of_node))
		node = of_find_compatible_node(NULL, NULL, "gpio_pwm");
	else
		node = pdev->dev.of_node;

	if (IS_ERR(node)) {
		pr_err("%s get gpio_pwm_node failed!\n", __func__);
		ret = -EFAULT;
		goto out;
```

```
		}
		gpio_num = of_get_named_gpio(node, "gpio_num", 0);
		if (gpio_num < 0) {
			pr_err("%s get gpio_num failed!\n", __func__);
			ret = gpio_num;
			goto out;
		}
		gpio_pwm_task = kthread_run(gpio_pwm_thread, NULL, "gpio_pwm_thread");
		if (IS_ERR(gpio_pwm_task)) {
			pr_err("%s kthread_run failed!\n", __func__);
			ret = -EFAULT;
			goto out;
		}
		return 0;

out:
		return ret;
}
static int gpio_pwm_remove(struct platform_device *pdev)
{
		if (!IS_ERR(gpio_pwm_task)) {
			kthread_stop(gpio_pwm_task);
			gpio_pwm_task = NULL;
		}
		return 0;
}

#if defined(CONFIG_OF)
static const struct of_device_id gpio_pwm_of_match[] = {
		{ .compatible = "gpio_pwm",	},
		{},
};
#endif

		static struct platform_driver gpio_pwm_driver = {
```

```
		.probe = gpio_pwm_probe,
		.remove = gpio_pwm_remove,
		.driver = {
			.name = "gpio_pwm",
#if defined(CONFIG_OF)
			.of_match_table = gpio_pwm_of_match,
#endif
		}
};

static int __init gpio_pwm_init(void)
{
		return platform_driver_register(&gpio_pwm_driver);
}
static void __exit gpio_pwm_exit(void)
{
		platform_driver_unregister(&gpio_pwm_driver);
}
module_init(gpio_pwm_init);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("gpio_pwm_dirver");
MODULE_AUTHOR("Mediatek");
```

9 Frequently Asked Questions

9.1 Debugging Related Questions

9.1.1 How to check gpio current status? (before mt6763)

The gpio current status can be dumped by adb command as below:

```
adb shell "cat /sys/class/misc/mtgpio/pin"
```

The dump result is:

```
PIN: [MODE] [PULL_SEL] [DIN] [DOUT] [PULL EN] [DIR] [IES] [SMT]
0:00001011
1:01000111
2:00001011
3:01101011
4:00001010
.....
```

The first line is the format.

Table 9-1. pin status dump information (before mt6763)

item	Description
PIN	The pin number
MODE	Current gpio mux mode
PULL_SEL	Current pull direction. It is valid when PULL_EN=1.(0: pull down; 1: pull up)
DIN	Current input value. (0: low; 1: high)
DOUT	Current output value. It is valid when DIR=1. (0: low; 1: high)
PULL_EN	Pull function status. (0: disabled; 1: enabled)
DIR	Pin direction.(0: input; 1: output)
IES	Input function status. (0: disabled; 1: enabled)
SMT	Schmitt trigger function status. (0: disabled; 1: enabled)

For example:

4:00001010

It means GPIO4 current status is : MODE 0 (gpio mode), output low, pull down.

9.1.2 How to check gpio current status? (mt6763 and future)

The gpio current status can be dumped by adb command as below:

```
adb shell "cat /sys/devices/platform/xxx.pinctrl/mtgpio "
```

xxx is the gpio register base address.

Example: mt6763 is 10005000, mt6758 is 10050000.

The dump result is:

```
PIN: [MODE] [DIR] [DOUT] [DIN] [PULL_EN] [PULL_SEL] [IES] [SMT] [DRIVE] ( [R1] [R0] )
   0: 800011110
   1: 000100110
   ...
  29: 110011114 10
  30: 100011113 10
```

The first line is the format.

Table 9-2. pin status dump information (mt6763 and future)

item	Description
PIN	The pin number
MODE	Current gpio mux mode
DIR	Pin direction.(0: input; 1: output)
DOUT	Current output value. It is valid when DIR=1. (0: low; 1: high)
DIN	Current input value. (0: low; 1: high)
PULL_EN	Pull function status. (0: disabled; 1: enabled)
PULL_SEL	Current pull direction. It is valid when PULL_EN=1.(0: pull down; 1: pull up)
IES	Input function status. (0: disabled; 1: enabled)
SMT	Schmitt trigger function status. (0: disabled; 1: enabled)
DRIVE	The pin's driving strength
R1 R0	The pull resistor select. (some pins can select the pull resistor value)

For example:

1:000100110

It means GPIO1 current status is : MODE 0 (gpio mode), input high , pull disable driving strength is step 0.