PRBMD02 Application Note

GPIO application note

Disclaimer

Liability Disclaimer
K-Solution Consulting Co. Ltd reserves the right to make changes without further notice to the product to improve reliability, function or design. K-Solution Consulting Co. Ltd does not assume any liability arising out of the application or use of any product or circuits described herein.

Life Support Applications
K-Solution Consulting Co. Ltd's products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. K-Solution Consulting Co. Ltd customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify K-Solution Consulting Co. Ltd for any damages resulting from such improper use or sale.

1. Introduction

GPIO, the full name of General-Purpose Input/Output, is a general-purpose pin that can be dynamically configured and controlled during software operation.

The default properties of PRBMD02 GPIO power-on are as follows:

Pin	default mode	Default IN-OUT	IRQ/Wakeup	FULLMUX	ANA	KSCAN
P00	GPIO	IN	√	√		mk_in[0]
P01	GPIO	IN	√			mk_out[0]
P02	SWD_IO		√	√		mk_in[1]
P03	SWD_CLK	IN	√	√		mk_out[1]
TEST_MODE						
P09	GPIO	IN	√	√		mk_out[4]
P10	GPIO	IN	√	√		mk_in[4]
P14	GPIO	IN	√	√	√	mk_out[2]
P15	GPIO	IN	√	√	√	mk_in[2]
P16	XTALI(ANA)	ANA			√	mk_out[10]
P17	XTALO(ANA)	ANA			√	mk_out[9]
P18	GPIO	IN	√	√	√	mk_in[5]
P20	GPIO	IN	√	√	√	mk_out[5]
P23	GPIO	IN	√	√	√	mk_in[6]
P24	GPIO	IN	√	√	√	mk_out[3]
P25	GPIO	IN	√	√	√	mk_in[3]
P31	GPIO	IN	√	√		mk_out[7]
P32	GPIO	IN	√	√		mk_in[7]
P33	GPIO	IN	√	√		mk_out[6]
P34	GPIO	IN	√	√		mk_in[8]

Table 1: GPIO power-on default property configuration

1.1. Special IO

1.1.1. TEST_MODE

TEST_MODE has a special purpose and configures the state of the chip together with P24 and P25. Therefore, TEST_MODE cannot be used in the application code. In addition, the combinations of TEST_MODE, P24, and P25 that need to be supported on the hardware circuit are {0,*,*} and {1,0,0}. The former chip is in Normal Mode to run the program, and the latter The chip is in Program Mode and the program can be programmed.

TEST_MODE	P24	P25	Mode
0	*	*	Normal mode Program mode
1	0	0	Program mode
1	0	1	Scan Module
1	1	0	Bist mode

Table 2: GPIO mode selection

There are two ways to enter program mode:

- TEST_MODE=0, PhyPlusKit needs to run in the single-wire programming stage (UDLL48) or the two-wire programming stage (UXTDWU) first, and the chip switches to Program Mode after the handshake is successful.
- TEST_MODE=1, P24=0, P25=0. After the chip is reset, it detects the above level and enters ProgramMode.

1.1.2. P16, P17

P16 and P17 are used as analog ports by default, and are connected to crystal oscillator and capacitor to form an oscillation circuit. Among them, P16 is XTALI, and P17 is XTALO. Currently, application code cannot use P16 and P17 for other purposes. Even if RC 32K is used in the system, P16 and P17 cannot be used for other purposes.

In the future, P16 and P17 will support other functions, such as GPIO, IOMUX, etc. The documentation will be updated at that time.

1.1.3. P1

P1 does not support FULLMUX function, FULLMUX refers to multiplexing GPIO as other module pins.

1.1.4. P2. P3

P2 (SDW_IO) and P3 (SDW_CLK) can be connected to the debugger. When the debugger is connected to debug the program, the non-debugging functions of these two IO ports will be affected.

Therefore, when using P2 and P3, the hardware should not be connected to the debugger.

1.2. GPIO mode

GPIO mode is the most commonly used mode, which can be configured as output and output high and low levels, and can be configured as input to read external high and low levels.

When configured as an input, interrupts and wakeups are supported.

1.2.1.GPIO output

Configure the corresponding GPIO direction register as output, and write 1 or 0 to the output register to output high or low level.

1.2.2.GPIO input

Configure the corresponding GPIO direction register as an input, and read the value of the input register to obtain the current level state of the GPIO.

If interrupts is used, GPIO interrupt enable function needed to be opened and configure the interrupt generation conditions.

If wake-up is used, the GPIO wake-up enable function needs to be turned on, and the wake-up generation conditions need to be configured.

1.2.3.GPIO retention

When GPIO is output, the retention function can be configured. retention is off by default.

When retention is turned on, the output characteristics and output values of GPIO are not listed when the system sleeps. When retention is off, the GPIO will return to the default input state when the system sleeps.

If there is no retention, after the system sleeps, the high and low state retention of the external circuit of the pin is realized according to the input state of the pin plus the pull-down resistor, and the driving ability is weak at this time. retention is a function that is added to increase the driving capability.

For example, when P00 is running, it is configured as GPIO output and outputs 1. If the system goes to sleep, the GPIO will become input, and 1 will not be output at this time. If you want the GPIO to keep the output 1 state even during sleep, you need to configure the retention function of the GPIO before sleep.

1.2.4.GPIO pull high/low resistance

Each GPIO supports four pull-up and pull-down resistor configurations:

- Floating: Hi-Z

- Strong pull-up: pull-up to AVDD33, high level, large drive current. Pull-up resistor 150kΩ ohm.
- Weak pull-up: pull-up to AVDD33, high level, small drive current. Pull-up resistor 1MΩ ohm.
- Pull-down: pull-down to ground, low level, pull-down resistor 150kΩ ohm.

Hardware default value of pull-up and pull-down resistors:

- P03, P24 and P25: pull down
- Other GPIOs: floating

1.2.5. Interrupt and wakeup

All GPIOs except TEST_MODE, P16, P17 support interrupt and wakeup. Interrupts support level-triggered and edge-triggered, and wake-up supports edge-triggered.

Note: When GPIO is used as a wake-up source, the internal pull-up resistor or pull-down resistor of this pin must be configured, and it cannot be in a high-impedance state.

1.3. FULLMUX mode

All other GPIOs except TEST_MODE, P16, P17, P1 support the GPIO FULLMUX function, and the GPIO can be configured as UART, I2C, PWM and other functions according to the application.

For example, in programming mode, UART uses P9(TX), P10(RX). P9 and P10 here are the GPIO FULLMUX functions. In the application, we can multiplex other GPIOs as UART functions, or multiplex P9, P10 as other functions such as PWM. GPIO and multiplexing relationship can be flexibly configured.

FULLMUX configuration meaning description:

FULLMUX define	FULLMUX value	FULLMUX description
FMUX_IIC0_SCL	0	I2C0 clk pin
FMUX_IIC0_SDA	1	I2C0 data pin
FMUX_IIC1_SCL	2	I2C1 clk pin
FMUX_IIC1_SDA	3	I2C1 data pin
FMUX_UART0_TX	4	UART0 Tx pin
FMUX_UART0_RX	5	UART0 Rx pin
FMUX_RF_RX_EN	6	RF rx function debug pin
FMUX_RF_TX_EN	7	RF tx function debut pin
FMUX_UART1_TX	8	UART1 Tx pin
FMUX_UART1_RX	9	UART1 Rx pin
FMUX_PWM0	10	PWM channel 0
FMUX_PWM1	11	PWM channel 1
FMUX_PWM2	12	PWM channel 2
FMUX_PWM3	13	PWM channel 3
FMUX_PWM4	14	PWM channel 4
FMUX_PWM5	15	PWM channel 5
FMUX_SPI_0_SCK	16	SPI0 clk pin
FMUX_SPI_0_SSN	17	SPI0 CS pin
FMUX_SPI_0_TX	18	SPI0 Tx pin
FMUX_SPI_0_RX	19	SPI0 Rx pin
FMUX_SPI_0_SCK	20	SPI1 clk pin
FMUX_SPI_1_SSN	21	SPI1 CS pin
FMUX_SPI_1_TX	22	SPI1 Tx pin
FMUX_SPI_1_RX	23	SPI1 Rx pin
FMUX_CHAX	24	Rotary encoder CHAX pin
FMUX_CHBX	25	Rotary encoder CHBX pin
FMUX_CHIX	26	Rotary encoder CHIX pin
FMUX_CHAY	27	Rotary encoder CHAY pin
FMUX_CHBY	28	Rotary encoder CHBY pin
FMUX_CHIY	29	Rotary encoder CHIY pin
FMUX_CHAZ	30	Rotary encoder CHAZ pin
FMUX_CHBZ	31	Rotary encoder CHBZ pin
FMUX_CHIZ	32	Rotary encoder CHIZ pin

FULLMUX define	FULLMUX value	FULLMUX description
FMUX_CLK1P28M	33	DMIC clk pin
FMUX_ADCC	34	DMIC data pin
FMUX_ANT_SEL_0	35	Antenna selection 0, for poisitioning
FMUX_ANT_SEL_1	36	Antenna selection 1, for positioning
FMUX_ANT_SEL_2	37	Antenna selection 2, for positioning

Table 3: GPIO FULLMUX selection

1.4. ANALOG mode

Only P11, P14, P15, P16, P17, P18, P20, P23, P24, P25 support the analog function.

32.768K crystal oscillator circuit: P16, P17 are connected to capacitors, 32.768K crystal oscillator constitutes an oscillator circuit, P16, P17 cannot be used for other purposes.

- *VOICE*: VOICE supports DMIC and AMIC. When using the AMIC circuit, the pins used are P18(pga+), P20(pga-), P15(micphone bias), P23(micphone bias reference voltage), among which P23 is optional.
- *ADC*: collects the voltage on the pin. The single-ended supported pins are P11, P23, P24, P14, P15, and P20, and the differential supported pins are P18P25, P14P24, and P20P15.

1.5. KSCAN mode

When the number of keys in the keyboard is large, in order to reduce the occupation of I/O ports, the keys are usually arranged in a matrix form. In a matrix keyboard, each horizontal and vertical line is not directly connected at the intersection, but is connected by a key.

For example: 4*4 matrix keys support 16 keys, which is twice the number of conventional keys. When the number of keys required is relatively large, it is more reasonable to use the matrix method to make the keyboard.

PRBMD02 has a built-in hardware matrix circuit, which is simple to use and efficient to process.

For example: there is a 4*4 matrix button, the pins used by KSCAN are as follows:

row	col	KSCAN	GPIO
row0		mk_in[0]	P0
row1		mk_in[1]	P2
row2		mk_in[2]	P15
row3		mk_in[3]	P25
	col0	mk_out[0]	P1
	col1	mk_out[1]	P3
	col2	mk_out[2]	P14
	col3	mk_out[3]	P24

Table 4: 4 x 4 key metric allocation table

The corresponding relationship between KSCAN mk_in, mk_out and GPIO is shown in Table 1: GPIO power-on default attribute configuration.

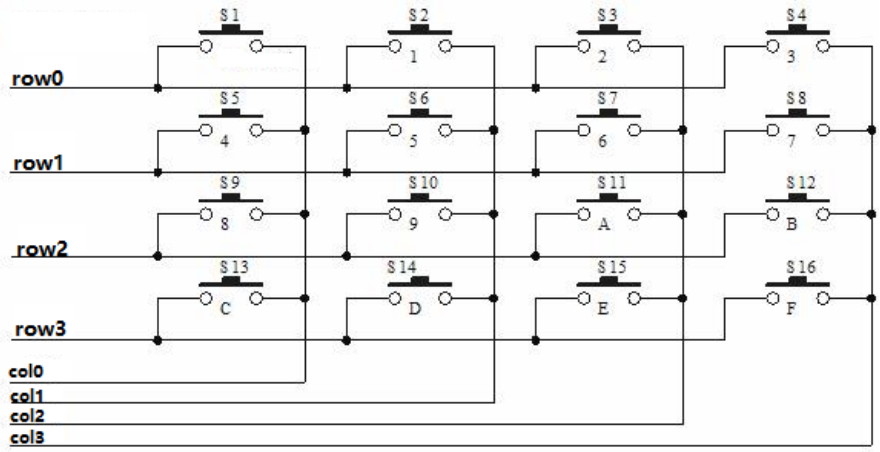

diagram 1 : 4x4 key metric connection

key	row	col
S1	0	0
S2	0	1
S3	0	2
S4	0	3
S5	1	0
S6	1	1
S7	1	2
S8	1	3
S9	2	0
S10	2	1
S11	2	2
S12	2	3
S13	3	0
S14	3	1
S15	3	2
S16	3	3

Table 5: 4*4 Matrix Button Corresponding Diagram

When the KSCAN presses a key, an interrupt will be generated to inform the upper layer of m and n in the key row and column information mk_in[m] and mk_out[n].

2. GPIO typical application

2.1. GPIO output

Configure the corresponding GPIO direction register (swporta_ddr) as an output, and set the output register (swporta_dr) to 0 or 1. The driver has a corresponding API, which can be called directly.

For example: set P0 as an output, and keep outputting 0 and 1.

```
//6220/6250
static void simple_code(void)
{
	int32_t pin = P0;
	gpio_pin_handle_t pin_p = NULL;
	int32_t ret;
	pin_p = csi_gpio_pin_initialize(pin, NULL);
	ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_OUTPUT);
	while(1)
	{
	csi_gpio_pin_write(pin_p,1);
	csi_gpio_pin_write(pin_p,0);
	}
}
```

```
static void simple_code(void)
{
        gpio_pin_e pin = P0;
        hal_gpio_pin_init(pin,GPIO_OUTPUT);
        while(1)
        {
            hal_gpio_fast_write(pin,1);
            hal_gpio_fast_write(pin,0); }
        }
}
```

2.2. GPIO input

Configure the corresponding GPIO direction register (swporta_ddr) as the input, and read the input register (swporta_ddr) to obtain the current level state of the GPIO.

For example, configure P0 as an input and read the current GPIO level state.

```
//6220/6250
static void simple_code(void)
 {
       int32_t pin = P0;
       gpio_pin_handle_t pin_p = NULL;
       int32_t ret;
       bool value;

       pin_p = csi_gpio_pin_initialize(pin, NULL);
       ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_INPUT);
       ret = csi_gpio_pin_read(pin_p, &value);
       printf("pin:%d value:%d\n",pin,value);
}
```

```
//6220/6250
static void simple_code(void)
{
          gpio_pin_e pin = P0;
          bool value;
          hal_gpio_pin_init(pin,GPIO_INPUT);
          value = hal_gpio_read(pin);
          LOG("pin:%d value:%d\n",pin,value);
}
```

2.3.GPIO retention

When GPIO is configured for GPIO output, when the system sleeps, the GPIO output information will be lost. If you want to keep the GPIO output state and keep the output high and low after the system sleeps, you need to use the GPIO retention function.

For example: set P0 as an output, and expect it to persist when the system sleeps.

```
//6220/6250
static void simple_code(void)
 {
       int32_t pin = P0;
       gpio_pin_handle_t pin_p = NULL;
       int32_t ret;
       pin_p = csi_gpio_pin_initialize(pin, NULL);
       ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_OUTPUT);
       csi_gpio_pin_write(pin_p,1);
       phy_gpioretention_register(pin); //enable this pin retention
       //phy_gpioretention_unregister(pin);// disable this pin retention
}
```

```
//6222/6252
static void simple_code(void)
{
	gpio_pin_e pin = P0;
	hal_gpioretention_register(pin);//enable this pin retention
	//hal_gpioretention_unregister(pin);//disable this pin retention
	hal_gpio_write(pin,1);
}
```

2.4. GPIO pull high/low resistance

Each GPIO supports four pull-up and pull-down configurations: floating, strong pull-up, pull-up, pull-down. For example: configure P0 as an input, configure a strong pull-up, and read the current GPIO level state.

```
//6222/6252
static void simple_code(void)
{
	int32 pin = P0;
	gpio_pin_handle_t pin_p = NULL;
	gpio_pupd_e type = GPIO_PULL_UP_S;
	bool value;
	int32 ret;

	pin_p = csi_gpio_pin_initialize(pin, NULL);
	phy_gpio_pull_set(pin,type);
	ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_INPUT);
	ret = csi_gpio_pin_read(pin_p, &value);
	printf("pin:%d value:%d\n",pin,value);
}
```

```
//6222/6252
static void simple_code(void)
{
	gpio_pin_e pin = P0;
	volatile bool value;

	hal_gpio_pull_set(pin,GPIO_PULL_UP_S);
	hal_gpio_pin_init(pin,GPIO_INPUT);
	value = hal_gpio_read(pin);
	LOG("pin:%d value:%d\n",pin,value);
}
```

2.5. Interrupt and wakeup

When using GPIO interrupts, you need to configure the GPIO interrupt generation conditions. After the interrupt is generated, the GPIO driver will respond to the interrupt and call the user-configured callback function.

When using GPIO to wake up, before the system goes to sleep, set the condition to wake up the system according to the current GPIO level state. When the condition occurs, the system wakes up and calls the callback function configured by the user.

For example: configure P0 as an input, support interrupt and wake-up, and support rising and falling edge triggering.

```
//6222/6252
void gpio_event_cb(int idx)
{
      printf("pin:%d\n",idx);
}

static void simple_code(void)
{
      int32 pin = P0;
      gpio_pin_handle_t pin_p = NULL;
      int32 ret;
      bool value;

      drv_pinmux_config(pin, PIN_FUNC_GPIO);
      pin_p = csi_gpio_pin_initialize(pin, gpio_event_cb);
      ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_INPUT);
      ret = csi_gpio_pin_read(pin_p, &value);

      if(value == 0)
        ret = csi_gpio_pin_set_irq(pin_p, GPIO_IRQ_MODE_RISING_EDGE, 1);
      else
        ret = csi_gpio_pin_set_irq(pin_p, GPIO_IRQ_MODE_FALLING_EDGE, 1);
......
}

//6220/6250 wakeup
static void simple_code(void)
{
      int32 pin = P0;
      gpio_pin_handle_t pin_p = NULL;
      int32 ret;
      bool value;

      drv_pinmux_config(pin, PIN_FUNC_GPIO);
      pin_p = csi_gpio_pin_initialize(pin, gpio_event_cb);
      ret = csi_gpio_pin_config_direction(pin_p, GPIO_DIRECTION_INPUT);
      ret = csi_gpio_pin_read(pin_p, &value);

       if(value == 0)
         phy_gpio_wakeup_set(pin,POL_RISING);
       else
         phy_gpio_wakeup_set(pin,POL_FALLING);
…
}
```

2.6. FULLMUX mode

When using GPIO FULLMUX, configure the FULLMUX function and turn on the FULLMUX enable. When not in use, be sure to turn off its FULLMUX enable.

For example, multiplexing P9 and P10 into UART, in which P9 is used as TX, P10 is used as RX, the baud rate is 115200, and UART0 is used.

```
//6222/6252
#define CONSOLE_UART_IDX 0
#define CONSOLE_TXD             P9
#define CONSOLE_RXD             P10
#define CONSOLE_TXD_FUNC        FMUX_UART0_TX
#define CONSOLE_RXD_FUNC        FMUX_UART0_RX
…..
drv_pinmux_config(CONSOLE_TXD, CONSOLE_TXD_FUNC);
drv_pinmux_config(CONSOLE_RXD, CONSOLE_RXD_FUNC); ......
…..
console_init(CONSOLE_UART_IDX, 115200, 0);
…..
```

```
//6222/6252
void dbg_printf_init(void)
 {

      uart_Cfg_t cfg =
      {
       .tx_pin = P9,
       .rx_pin = P10,
       .rts_pin = GPIO_DUMMY,
       .cts_pin = GPIO_DUMMY,
       .baudrate = 115200,
       .use_fifo = TRUE,
       .hw_fwctrl = FALSE,
       .use_tx_buf = FALSE,
       .parity = FALSE,
       .evt_handler = NULL,
      };
      hal_uart_init(cfg, UART0);
}
......
      hal_gpio_fmux_set(pcfg->tx_pin, fmux_tx);  //P9 FMUX_UART0_TX
      hal_gpio_fmux_set(pcfg->rx_pin, fmux_rx);  //P10 FMUX_UART0_RX
......
```

2.7. Analog mode

Please refers to ADC Application note

2.8. KSCAN mode

when using KSCAN mode, the following configurations are needed:

- Pull High/Low resistance
- The pins used in the row are configured as mk_in, and the pins used in the columns are configured as mk_out
- Configure KSCAN related registers

The KSCAN driver has encapsulated the above operations with API, and you can directly configure the corresponding pins and callback functions.

For example: define a 2*2 matrix key, in which the row uses P23, P18, and the column uses P24, P11. When a key is pressed or a key is released, the callback handler is kscan_evt_handler.

```
//6222/6252
#define NUM_KEY_ROWS 2
#define NUM_KEY_COLS 2

KSCAN_ROWS_e rows[NUM_KEY_ROWS] = { KEY_ROW_P23,KEY_ROW_P18};
KSCAN_COLS_e cols[NUM_KEY_COLS] = { KEY_COL_P24,KEY_COL_P11};

static void kscan_evt_handler(kscan_Evt_t* evt)
{
	printf("kscan_evt_handler\n");
	printf("num: ");
	printf("%d",evt->num);
	printf("\n");
	for(uint8_t i=0;i<evt->num;i++)
	{
	  printf("index: ");
	  printf("%d",i);
	  printf(",row: ");
	  printf("%d",evt->keys[i].row);
	  printf(",col: ");
	  printf("%d",evt->keys[i].col);
	  printf(",type: ");
	  printf("%s",evt->keys[i].type == KEY_PRESSED ? "pressed":"released");
	  printf("\n");
	}
}

void KSCAN_Init(void)
{
	printf("KSCAN_Init\n");
	kscan_Cfg_t cfg;
	cfg.ghost_key_state = NOT_IGNORE_GHOST_KEY;
	cfg.key_rows = rows;
	cfg.key_cols = cols;
	cfg.interval = 50;
	cfg.evt_handler = kscan_evt_handler;
	hal_kscan_init(cfg, 0, 0);
}
```

```
//6222/6252
#define NUM_KEY_ROWS 2
#define NUM_KEY_COLS 2

KSCAN_ROWS_e rows[NUM_KEY_ROWS] = {KEY_ROW_P23,KEY_ROW_P18};
KSCAN_COLS_e cols[NUM_KEY_COLS] = {KEY_COL_P24,KEY_COL_P11};

static void kscan_evt_handler(kscan_Evt_t* evt)
{
    LOG("\nkscan_evt_handler\n");
    LOG("num: %d\n",evt->num);
    for(uint8_t i=0;i<evt->num;i++)
    {
      LOG("index: ");
      LOG("%d",i);
      LOG(",row: ");
      LOG("%d",evt->keys[i].row);
      LOG(",col: ");
      LOG("%d",evt->keys[i].col);
      LOG(",type: ");
      LOG("%s",evt->keys[i].type == KEY_PRESSED ? "pressed":"released");
      LOG("\n");
    }

}

static void simple_doce(void)
{
    kscan_Cfg_t cfg;
    cfg.ghost_key_state = NOT_IGNORE_GHOST_KEY;
    cfg.key_rows = rows;
    cfg.key_cols = cols;
    cfg.interval = 50;
    cfg.evt_handler = kscan_evt_handler;
    hal_kscan_init(cfg, task_id, KSCAN_WAKEUP_TIMEOUT_EVT);

}
```