

# PHY62XX

## GPIO 应用指南

---

Version 0.6

Phyplus Microelectronics Limited

2019/07/16

## 版本控制信息

版本/状态	作者	参与者	起止日期	备注
V0.4	付晓亮		07/05/2018	文档初稿
V0.5	ZQ		08/09/2018	1.加入上下拉电阻值 2.完善 IO 上电默认配置描述
V0.6	付晓亮		07/16/2019	1.完善 IO 属性说明 2.优化 gpio 做数字按键代码

## 目录

<b>1</b>	<b>简介 .....</b>	<b>1</b>
1.1	GPIO 模式 .....	2
1.1.1	模拟和数字 .....	2
1.1.2	输入和输出 .....	2
1.1.3	上下拉 .....	2
1.1.4	中断和唤醒 .....	2
1.2	功能模式 .....	2
<b>2</b>	<b>API .....</b>	<b>3</b>
2.1	枚举&宏 .....	3
2.1.1	NUMBER_OF_PINS .....	3
2.1.2	NUMBER_OF_IRQ_PINS .....	3
2.1.3	GPIO_Pin_e .....	3
2.1.4	GPIO 输入输出状态 .....	4
2.1.5	GPIO_ioe .....	5
2.1.6	BitAction_e .....	5
2.1.7	IO_Pull_Type_e .....	5
2.1.8	IO_Wakeup_Pol_e .....	5
2.1.9	Fmux_Type_e .....	6
2.2	数据结构 .....	7
2.2.1	gpioin_Hdl_t .....	7
2.2.2	gpioin_Ctx_t .....	7
2.2.3	gpio_Ctx_t .....	7
2.3	API .....	7
2.2.4	int hal_gpio_init(void) .....	7
2.2.5	void hal_gpio_pin_init(GPIO_Pin_e pin, GPIO_ioe type) .....	8
2.2.6	int gpio_pin0to3_pin31to34_control(GPIO_Pin_e pin, uint8_t en) .....	8
2.2.7	int hal_gpio_fmux(GPIO_Pin_e pin, BitAction_e value) .....	9
2.2.8	int hal_gpio_fmux_set(GPIO_Pin_e pin, Fmux_Type_e type) .....	9
2.2.9	int hal_gpio_cfg_analog_io(GPIO_Pin_e pin, BitAction_e value) .....	10
2.2.10	int hal_gpioin_register (GPIO_Pin_e pin, gpioin_Hdl_t posedgeHdl, gpioin_Hdl_t negedgeHdl) .....	10
2.2.11	int hal_gpioin_unregister(GPIO_Pin_e pin) .....	11
2.2.12	int hal_gpio_pull_set(GPIO_Pin_e pin, IO_Pull_Type_e type) .....	11
2.2.13	void hal_gpio_write(GPIO_Pin_e pin, uint8_t en) .....	12
2.2.14	uint32_t hal_gpio_read(GPIO_Pin_e pin) .....	12
2.2.15	int hal_gpioin_enable(GPIO_Pin_e pin) .....	13

2.2.16	int hal_gpioin_disable(GPIO_Pin_e pin) .....	13
2.2.17	int gpio_interrupt_enable(GPIO_Pin_e pin, IO_Wakeup_Pol_e type) .....	14
2.2.18	int gpio_interrupt_disable(GPIO_Pin_e pin).....	14
2.2.19	void io_wakeup_control(GPIO_Pin_e pin, BitAction_e value).....	15
2.2.20	void hal_gpio_wakeup_set (GPIO_Pin_e pin,IO_Wakeup_Pol_e type).....	15
2.2.21	void gpio_sleep_handler(void) .....	15
2.2.22	void gpio_wakeup_handler(void) .....	16
2.2.23	void gpioin_wakeup_trigger(GPIO_Pin_e pin).....	16
2.2.24	void hal_GPIO_IRQHandler(void).....	16
2.2.25	void gpioin_event(uint32 int_status, uint32 polarity) .....	16
2.2.26	void gpioin_event_pin(GPIO_Pin_e pin, IO_Wakeup_Pol_e type).....	17
<b>3</b>	<b>软件应用 .....</b>	<b>18</b>
3.1	数字输出 .....	18
3.2	数字输入 .....	18
3.3	中断 .....	19
3.4	唤醒 .....	21
3.5	脉冲测量 .....	24
3.6	Timer 定时 .....	28
3.7	数字按键 .....	31

## 图表

图表 1 GPIO 上电默认属性配置 .....	2
--------------------------	---

## 1 简介

本文档介绍了 PHY62XX GPIO 模块的原理和使用方法。

GPIO, 全称 General-Purpose Input/Output (通用输入输出), 是一种软件运行期间能够动态配置和控制的通用引脚。

PHY62XX GPIO 最多支持 35 个 IO, IO 上电默认属性见图表 1。

不同 IO 上电后默认属性不同, 除了做 GPIO 模式, IO 还可以配置为功能模式, 即复用为外设模块的驱动引脚, 比如 I2C、UART 等。实际使用时一定要确保 IO 属性是配置正确的。表中 phyplus\_spi\* 是 phyplus 私有协议的 SPI 不是通用 spi。可以通过 IOMUX 功能配置成别的功能 IO。

#	QFN48	QFN32	Default MODE	Default IN_OUT	IRQ	Wakeup	ANA_IO
0	GPIO_P00	√	jtag_dout	OUT	√	√	
1	GPIO_P01	√	jtag_din	IN	√	√	
2	GPIO_P02	√	jtag_tm	IN	√	√	
3	GPIO_P03	√	jtag_clk	IN	√	√	
4	GPIO_P04		GPIO	IN	√	√	
5	GPIO_P05		GPIO	IN	√	√	
6	GPIO_P06		GPIO	IN	√	√	
7	GPIO_P07		GPIO	IN	√	√	
8	TEST_MODE	√					
9	GPIO_P09	√	GPIO	IN	√	√	
10	GPIO_P10	√	GPIO	IN	√	√	
11	GPIO_P11		GPIO	IN	√	√	ADC_CH1N_P11
12	GPIO_P12		GPIO	IN	√	√	ADC_CH1P_P12
13	GPIO_P13		GPIO	IN	√	√	ADC_CH2N_P13
14	GPIO_P14	√	GPIO	IN	√	√	ADC_CH2P_P14
15	GPIO_P15	√	GPIO	IN	√	√	ADC_CH3N_P15
16	GPIO_P16	√	XTALI(ANA)	ANA	√	√	
17	GPIO_P17	√	XTALO(ANA)	ANA	√	√	
18	GPIO_P18	√	GPIO	IN		√	
19	GPIO_P19		GPIO	IN		√	
20	GPIO_P20	√	GPIO	IN		√	ADC_CH3P_P20
21	GPIO_P21		GPIO	IN		√	
22	GPIO_P22		GPIO	IN		√	
23	GPIO_P23	√	GPIO	IN		√	
24	GPIO_P24	√	GPIO	IN		√	
25	GPIO_P25	√	GPIO	IN		√	
26	GPIO_P26		GPIO	IN		√	
27	GPIO_P27		GPIO	IN		√	

28	GPIO_P28		GPIO	IN	√
29	GPIO_P29		GPIO	IN	√
30	GPIO_P30		GPIO	IN	√
31	GPIO_P31	√	phyplus_spi_t_ssn	IN	√
32	GPIO_P32	√	phyplus_spi_t_rx	IN	√
33	GPIO_P33	√	phyplus_spi_t_tx	OUT	√
34	GPIO_P34	√	phyplus_spi_t_sck	IN	√

图表 1 GPIO 上电默认属性配置

## 1.1 GPIO 模式

本文档主要介绍做 GPIO 时的一些注意事项。

### 1.1.1 模拟和数字

35 个 IO 都可以作为数字端口或模拟端口。

### 1.1.2 输入和输出

35 个 IO 作为数字端口使用时，都可以配置端口方向为输入或输出。

### 1.1.3 上下拉

每个 IO 都可以配置上下拉电阻，用来设置引脚的默认状态。

- 浮空:未知态。
- 弱上拉:上拉到 AVDD33，高电平，驱动电流小。上拉电阻 150K 欧姆
- 强上拉:上拉到 AVDD33，高电平，驱动电流大。上拉电阻 10K 欧姆
- 下拉:下拉到地，低电平，下拉电阻 100K 欧姆。

### 1.1.4 中断和唤醒

P00~P17，这 18 个 GPIO 支持中断和唤醒。

P18~P34，这 17 个 GPIO 支持唤醒，不支持中断。

## 1.2 功能模式

GPIO 也可复用为其他外设模块 IO，可见外设文档说明和示例代码。

## 2 API

### 2.1 枚举&宏

#### 2.1.1 NUMBER\_OF\_PINS

GPIO 数量。

#### 2.1.2 NUMBER\_OF\_IRQ\_PINS

GPIO 支持中断引脚数量。

#### 2.1.3 GPIO\_Pin\_e

GPIO 宏定义，GPIO\_DUMMY 为虚拟 pin，一般作无效 pin 使用。

```
typedef enum{  
  
    GPIO_P00    = 0,    P0    = 0,  
    GPIO_P01    = 1,    P1    = 1,  
    GPIO_P02    = 2,    P2    = 2,  
    GPIO_P03    = 3,    P3    = 3,  
    GPIO_P04    = 4,    P4    = 4,  
    GPIO_P05    = 5,    P5    = 5,  
    GPIO_P06    = 6,    P6    = 6,  
    GPIO_P07    = 7,    P7    = 7,  
    TEST_MODE   = 8,    P8    = 8,  
    GPIO_P09    = 9,    P9    = 9,  
    GPIO_P10    = 10,   P10   = 10,  
    GPIO_P11    = 11,   P11   = 11,   Analog_IO_0 = 11,  
    GPIO_P12    = 12,   P12   = 12,   Analog_IO_1 = 12,  
    GPIO_P13    = 13,   P13   = 13,   Analog_IO_2 = 13,  
    GPIO_P14    = 14,   P14   = 14,   Analog_IO_3 = 14,  
    GPIO_P15    = 15,   P15   = 15,   Analog_IO_4 = 15,  
    GPIO_P16    = 16,   P16   = 16,   XTALI = 16,  
    GPIO_P17    = 17,   P17   = 17,   XTALO = 17,  
    GPIO_P18    = 18,   P18   = 18,   Analog_IO_7 = 18,  
    GPIO_P19    = 19,   P19   = 19,   Analog_IO_8 = 19,  
    GPIO_P20    = 20,   P20   = 20,   Analog_IO_9 = 20,  
  
}
```

```

GPIO_P21  =  21,  P21  =  21,
GPIO_P22  =  22,  P22  =  22,
GPIO_P23  =  23,  P23  =  23,
GPIO_P24  =  24,  P24  =  24,
GPIO_P25  =  25,  P25  =  25,
GPIO_P26  =  26,  P26  =  26,
GPIO_P27  =  27,  P27  =  27,
GPIO_P28  =  28,  P28  =  28,
GPIO_P29  =  29,  P29  =  29,
GPIO_P30  =  30,  P30  =  30,
GPIO_P31  =  31,  P31  =  31,
GPIO_P32  =  32,  P32  =  32,
GPIO_P33  =  33,  P33  =  33,
GPIO_P34  =  34,  P34  =  34,

GPIO_DUMMY =  0xff,
}GPIO_Pin_e;

```

#### 2.1.4 GPIO 输入输出状态

命名	含义
GPIO_PIN_ASSI_NONE	空闲。
GPIO_PIN_ASSI_OUT	输出。
GPIO_PIN_ASSI_IN_IRQ	输入，支持中断。
GPIO_PIN_ASSI_IN_WAKEUP	输入，支持唤醒。
GPIO_PIN_ASSI_IN_IRQ_AND_WAKEUP	输入，支持中断和唤醒。



### 2.1.5 GPIO\_ioe

GPIO 配置为输入或输出。

IE	输入。
OEN	输出

### 2.1.6 BitAction\_e

IO 配置为功能模式或 GPIO 模式、模拟端口或数字端口等，传递参数使用，含义为使能和禁止。

Bit_DISABLE	禁止。
Bit_ENABLE	使能。

### 2.1.7 IO\_Pull\_Type\_e

配置 pin 的上下拉模式。

FLOATING	无上下拉，pin 悬空。
WEAK_PULL_UP	弱上拉。
STRONG_PULL_UP	强上拉。
PULL_DOWN	下拉。

### 2.1.8 IO\_Wakeup\_Pol\_e

配置 pin 的中断极性 or 唤醒极性，上升沿或下降沿。

POSEDGE	上升沿触发中断或唤醒。
NEGEDGE	下降沿触发中断或唤醒。

### 2.1.9 Fmux\_Type\_e

配置 pin 的功能设置。

定义	说明
IIC0_SCL	IIC0 时钟
IIC0_SDA	IIC0 数据
IIC1_SCL	IIC1 时钟
IIC1_SDA	IIC1 数据
I2S_SCK	I2S 时钟
I2S_WS	I2S 声道选择
I2S_SDO_0	I2S 数据输出通道 0
I2S_SDI_0	I2S 数据输入通道 0
UART_TX	UART 发送，只支持 GPIOP9
UART_RX	UART 接收，只支持 GPIOP10
PWM0	PWM 通道 0
PWM1	PWM 通道 1
PWM2	PWM 通道 2
PWM3	PWM 通道 3
PWM4	PWM 通道 4
PWM5	PWM 通道 5
SPI_0_SCK	SPI0 时钟
SPI_0_SSN	SPI0 片选
SPI_0_TX	SPI0 发送
SPI_0_RX	SPI0 接收
SPI_1_SCK	SPI1 时钟
SPI_1_SSN,	SPI1 片选
SPI_1_TX	SPI1 发送
SPI_1_RX	SPI1 接收
CHAX	正交解码器
CHBX	正交解码器
CHIX	正交解码器
CHAY	正交解码器
CHBY	正交解码器
CHIY	正交解码器
CHAZ	正交解码器
CHBZ	正交解码器
CHIZ	正交解码器
CLK1P28M	预留

ADCC	预留
I2S_SDO_1	I2S 数据输出通道 1
I2S_SDO_2	I2S 数据输出通道 2
I2S_SDO_3	I2S 数据输出通道 3
I2S_SDI_1	I2S 数据输入通道 1
I2S_SDI_2	I2S 数据输入通道 2
I2S_SDI_3	I2S 数据输入通道 3

## 2.2 数据结构

### 2.2.1 gpioin\_Hdl\_t

GPIO 中断回调函数和唤醒回调函数类型。

### 2.2.2 gpioin\_Ctx\_t

GPIO 模式输入控制结构体。

类型	参数名	说明
bool	enable	引脚输入使能标志。
uint8_t	pin_state	引脚电平状态。
gpioin_Hdl_t	posedgeHdl	上升沿回调函数指针。
gpioin_Hdl_t	negedgeHdl	下降沿回调函数指针。

### 2.2.3 gpio\_Ctx\_t

GPIO 全局控制结构体。

类型	参数名	说明
bool	state	GPIO 模块使能标志位。
uint8_t	pin_assignments	引脚模式配置。
gpioin_Ctx_t	irq_ctx	输入引脚处理结构体。

## 2.3 API

### 2.2.4 int hal\_gpio\_init(void)

GPIO 模块初始化：初始化硬件，使能中断，配置中断优先级等。

该函数需要在系统初始化的时候进行设置，一般是在 `hal_init()` 函数中调用，具体信息请参考例程。

- 参数

无。

- 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

## 2.2.5 void hal\_gpio\_pin\_init(GPIO\_Pin\_e pin,GPIO\_ioe type)

配置 GPIO 输入或输出。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
GPIO_ioe	type	配置 GPIO 为输入或输出。

- 返回值

无。

## 2.2.6 int gpio\_pin0to3\_pin31to34\_control(GPIO\_Pin\_e pin, uint8\_t en)

由于 GPIO\_P00~GPIO\_P03、GPIO\_P31~ GPIO\_P34 上电默认是非 GPIO 模式。  
可通过该接口配置为 GPIO 模式。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。

uint8_t	en	1: 将引脚配置为 GPIO 模式。
		0: 将引脚配置为上电默认复用配置。

#### ● 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

### 2.2.7 int hal\_gpio\_fmux(GPIO\_Pin\_e pin, BitAction\_e value)

配置 IO 为 GPIO 模式还是功能模式。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
BitAction_e	value	Bit_ENABLE: 将 IO 配置为功能模式。 Bit_DISABLE: 将 IO 配置为 GPIO 模式。

#### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

### 2.2.8 int hal\_gpio\_fmux\_set(GPIO\_Pin\_e pin, Fmux\_Type\_e type)

配置 IO 的功能模式。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
Fmux_Type_e	type	IO 的功能模式

#### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

### 2.2.9 int hal\_gpio\_cfg\_analog\_io(GPIO\_Pin\_e pin, BitAction\_e value)

将 GPIO 配置为模拟端口或数字端口。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
BitAction_e	value	Bit_ENABLE：将引脚配置为模拟端口。 Bit_DISABLE：将引脚配置为数字端口。

#### ● 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

### 2.2.10 int hal\_gpioin\_register(GPIO\_Pin\_e pin, gpioin\_Hdl\_t posedgeHdl, gpioin\_Hdl\_t negedgeHdl)

注册 GPIO 的输入模式，该模式下支持中断回调和唤醒回调，包括上升沿回调和下降沿回调。

### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
gpioin_Hdl_t	posedgeHdl	上升沿回调函数，可以为 NULL。
gpioin_Hdl_t	negedgeHdl	下降沿回调函数，可以为 NULL。

### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

## 2.2.11 int hal\_gpioin\_unregister(GPIO\_Pin\_e pin)

注销 GPIO 的输入模式，注销后中断和唤醒的上升沿回调函数和下降沿回调函数无效。

### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

## 2.2.12 int hal\_gpio\_pull\_set(GPIO\_Pin\_e pin,IO\_Pull\_Type\_e type)

设置 IO 的上下拉。

### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
Pull_Type_e	type	IO 上下拉设置。

#### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

### 2.2.13 void hal\_gpio\_write(GPIO\_Pin\_e pin, uint8\_t en)

向某一个 GPIO 写 1 或者 0。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
uint8_t	en	0: 写 0, 其他值: 写 1。

#### ● 返回值

无。

### 2.2.14 uint32\_t hal\_gpio\_read(GPIO\_Pin\_e pin)

读取某一个 GPIO 的值。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。



- 返回值

TRUE	引脚为高电平
FALSE	引脚为低电平

### 2.2.15 int hal\_gpioin\_enable(GPIO\_Pin\_e pin)

GPIO 输入功能使能，此函数会配置输入引脚属性、使能、回调函数等。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

### 2.2.16 int hal\_gpioin\_disable(GPIO\_Pin\_e pin)

GPIO 输入功能禁止。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

PPlus_SUCCESS	成功。
---------------	-----

其他数值

参考<error.h>

### 2.2.17 int gpio\_interrupt\_enable(GPIO\_Pin\_e pin, IO\_Wakeup\_Pol\_e type)

配置 GPIO 中断寄存器，使能中断。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
IO_Wakeup_Pol_e	type	中断触发极性，上升沿还是下降沿。

#### ● 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

### 2.2.18 int gpio\_interrupt\_disable(GPIO\_Pin\_e pin)

配置 GPIO 的中断寄存器，禁止中断。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。

#### ● 返回值

PPlus_SUCCESS	成功。
---------------	-----

其他数值

参考<error.h>

### 2.2.19 void io\_wakeup\_control(GPIO\_Pin\_e pin, BitAction\_e value)

配置 GPIO 唤醒使能或禁止。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
BitAction_e	value	Bit_ENABLE：wakeup 使能。 Bit_DISABLE：wakeup 禁止。

#### ● 返回值

无。

### 2.2.20 void hal\_gpio\_wakeup\_set (GPIO\_Pin\_e pin,IO\_Wakeup\_Pol\_e type)

配置 GPIO 唤醒模式：上升沿唤醒或者下降沿唤醒。

#### ● 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
IO_Wakeup_Pol_e	type	唤醒极性，上升沿触发还是下降沿触发。

#### ● 返回值

无。

### 2.2.21 void gpio\_sleep\_handler(void)

系统进入 sleep 前的回调函数，可以配置唤醒等信息。

#### ● 参数

无。

- 返回值

无。

### 2.2.22 void gpio\_wakeup\_handler(void)

系统从 sleep 唤醒后的回调函数，手动调用唤醒处理函数。

- 参数

无。

- 返回值

无。

### 2.2.23 void gpioin\_wakeup\_trigger(GPIO\_Pin\_e pin)

响应 GPIO 的唤醒事件。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。

- 返回值

无。

### 2.2.24 void hal\_GPIO\_IRQHandler(void)

GPIO 中断处理函数

- 参数

无。

- 返回值

无。

### 2.2.25 void gpioin\_event(uint32 int\_status, uint32 polarity)

GPIO 中断事件处理，遍历响应所有 GPIO 的所有中断事件。

- 参数

类型	参数名	说明
uint32	status	中断标志。
uint32	polarity	上升沿下降沿触发标志。

- 返回值

无。

### 2.2.26 void gpioin\_event\_pin(GPIO\_Pin\_e pin, IO\_Wakeup\_Pol\_e type)

单个 GPIO 的中断事件处理函数，会调用用户定义预设的回调函数。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
IO_Wakeup_Pol_e	type	中断极性，上升沿触发还是下降沿触发。

- 返回值

无。

## 3 软件应用

### 3.1 数字输出

35 个 IO 作为数字输出时，使用简单，可见下面参考代码。

测试参考硬件：PHY6200\_32\_V1.4。

引脚配置：GPIO\_P14 做数字输出引脚。

//示例代码

```
hal_gpio_pin_init(GPIO_P14,OEN);//配置数字输出模式
```

```
while(1)
```

```
{
```

```
    hal_gpio_write(GPIO_P14, 1); //输出高电平
```

```
    WaitMs(50);
```

```
    hal_gpio_write(GPIO_P14, 0); //输出低电平
```

```
    WaitMs(50);
```

```
}
```

### 3.2 数字输入

35 个 IO 作为数字输入时，使用简单，可见下面参考代码。

测试参考硬件：PHY6200\_32\_V1.4。

引脚配置：GPIO\_P14 做数字输入引脚。

//示例代码

```
hal_gpio_pin_init(GPIO_P14,IE);//P14 配置为输入模式

static bool gpiopin_state;

gpiopin_state = hal_gpio_read(GPIO_P14);//读取 P14 引脚电平

LOG("gpiopin_state:%d\n",gpiopin_state);

while(1)
{
    if(gpiopin_state != hal_gpio_read(GPIO_P14))
    {
        gpiopin_state = hal_gpio_read(GPIO_P14);

        LOG("gpiopin_state:%d\n",gpiopin_state);//P14 引脚电平变化时，打印
    }
}
```

### 3.3 中断

P00~P07，P09~P17 作为数字输入时，支持中断，可见下面代码。

测试参考硬件：PHY6200\_32\_V1.4。

P14 做数字输入引脚，中断使能。

P15 做数字输出引脚，以 2s 的周期输出高低电平。

P14、P15 连接，P15 电平变化触发 P14 产生中断。

示例代码：

//P14 的上升沿中断回调函数

```
void pos_callback(IO_Wakeup_Pol_e type)
```

```
{  
    if(POSEDGE == type){  
        LOG("posedge\n");  
    }  
}  
  
//P14 的上升沿中断回调函数  
  
void neg_callback(IO_Wakeup_Pol_e type)  
{  
    if(NEGEDGE == type){  
        LOG("negedge\n");  
    }  
}
```

```
//P15 配置为数字输出功能，以 2s 的周期输出高低电平  
  
hal_gpio_pin_init(GPIO_P15,OEN);  
  
//P14 配置为数字下拉  
  
hal_gpio_pull_set(GPIO_P14,PULL_DOWN);  
  
//配置 P14 回调函数  
  
hal_gpioin_register(GPIO_P14,pos_callback,neg_callback);  
  
while(1)  
{  
    hal_gpio_write(GPIO_P15, 1);  
    WaitMs(1000);  
    hal_gpio_write(GPIO_P15, 0);  
    WaitMs(1000);  
}
```



### 3.4 唤醒

P00~P07, P09~P34 作为数字输入时, 支持唤醒系统。

可参考示例 GPIO 中的 `gpio_wakeup`。

测试参考硬件: PHY6200\_32\_V1.4。

P14 做数字输入引脚, 边沿变化触发唤醒。

OSAL 在没有任务执行时系统会进入 `sleep`, 间隔 1s 后会再次唤醒后扫描任务的事件。

通过 `gpio` 唤醒回调函数可以知道是否是 `gpio` 唤醒了系统。

可以用跳线改变 P14 的状态, 观察 P14 回调函数的执行情况。

```
//OSAL_gpio.c
const pTaskEventHandlerFn tasksArr[] =
{
    LL_ProcessEvent,

    Gpio_Wakeup_ProcessEvent,//事件响应函数入口
};

void osalInitTasks( void )
{
    uint8 taskID = 0;

    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

    LL_Init( taskID++);

    Gpio_Wakeup_Init(taskID);//初始化函数入口
}
```

```
//gpio_demo.c

static uint8 gpio_wakeup_TaskID;

//回调函数

void posedge_callback_wakeup(GPIO_Pin_e pin,IO_Wakeup_Pol_e type){
    if(type == POSEDGE){
        LOG("pos:io:%d type:%d\n",pin,type);
    }
    else{
        LOG("err\n");
    }
}

void negedge_callback_wakeup(GPIO_Pin_e pin,IO_Wakeup_Pol_e type){
    if(type == NEGEDGE){
        LOG("neg:io:%d type:%d\n",pin,type);
    }
    else{
        LOG("err\n");
    }
}

//使用的结构体

typedef struct gpioin_wakeup_t{
    GPIO_Pin_e pin;
    gpioin_Hdl_t posedgeHdl;
    gpioin_Hdl_t negedgeHdl;
}gpioin_wakeup;

#define GPIO_WAKEUP_PIN_NUM 3
```

```

gpioin_wakeup gpiodemo[GPIO_WAKEUP_PIN_NUM] = {
    GPIO_P14,posedge_callback_wakeup,negedge_callback_wakeup,
    GPIO_P23,posedge_callback_wakeup,negedge_callback_wakeup,
    GPIO_P31,posedge_callback_wakeup,negedge_callback_wakeup,
};

//初始化函数

void Gpio_Wakeup_Init(uint8 task_id){
    uint8_t i = 0;
    static bool gpioin_state[GPIO_WAKEUP_PIN_NUM];

    gpio_wakeup_TaskID = task_id;
    LOG("gpio wakeup demo start...\n");

    for(i = 0;i<GPIO_WAKEUP_PIN_NUM;i++){
        hal_gpioin_register(gpiodemo[i].pin,gpiodemo[i].posedgeHdl,gpiodemo[i].negedgeHdl);
        gpioin_state[i] = hal_gpio_read(gpiodemo[i].pin);
        LOG("gpioin_state:%d %d\n",i,gpioin_state[i]);
    }
}

//事件响应函数

uint16 Gpio_Wakeup_ProcessEvent( uint8 task_id, uint16 events ){
    if(task_id != gpio_wakeup_TaskID){

        return 0;
    }

    return 0;
}

```

//注意事项

- 1.P00~P03：默认为 JTAG，可以配置位 GPIO，可唤醒系统。
- 2.P08 为模式选择引脚，不可以做其他用途。
- 3.P04~P07,P11~P15,P18~P30：默认 GPIO，可以唤醒系统。
- 3.P09~P10：默认为 GPIO，SDK 默认将其配置为 UART，如果 P09~P10 做 GPIO 使用时，请注意将 UART 功能映射到其他 io 上。
- 4.P16~P17：默认接 32K 晶振，当使用内部 rc 可以将其设置为 GPIO。
- 5.P31~P34：默认位 SPIF 接口，可以配置位 GPIO，支持唤醒。

### 3.5 脉冲测量

GPIO 中的 pulse\_measure 演示了如何测量脉冲类型和宽度。

测试参考硬件：PHY6200\_32\_V1.4。

P14 做数字输入引脚，采集 P14 上脉冲宽度。

可以手动用跳线改变 P14 的状态，回调函数会输出 P14 上的脉冲类型和长度。

```
//OSAL_gpio.c
const pTaskEventHandlerFn tasksArr[] =
{
    LL_ProcessEvent,

    Pulse_Measure_ProcessEvent, //事件响应函数入口
};
```

```
void osalInitTasks( void )
{
    uint8 taskID = 0;

    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

    LL_Init( taskID++);
    /* Application */

    Pulse_Measure_Init( taskID); //初始化函数入口
}
```

```
//gpio_demo.c

static uint8 pulseMeasure_TaskID; //脉冲测量任务 ID

typedef struct {
    bool        enable;
    bool        pinstate;
    uint32_t    edge_tick;
}gpiopin_Trig_t;

typedef struct {
    GPIO_Pin_e  pin;//测试用到的 IO

    bool        type;//脉冲类型

    uint32_t    ticks;//脉冲长度
}gpiopin_pulse_Width_measure_t;
```

```
gpioin_pulse_Width_measure_t measureResult ={//配置使用的 GPIO

    .pin = GPIO_P14,
};

static gpioin_Trig_t gpioTrig = {

    .enable = FALSE,
    .edge_tick = 0,
};

void plus_edge_callback(void){
    LOG("pulse:%d %d\n",measureResult.type,measureResult.ticks);
}

//回调函数

void pulse_measure_callback(GPIO_Pin_e pin,IO_Wakeup_Pol_e type)
{
    if(gpioTrig.enable == FALSE)
    {
        gpioTrig.enable = TRUE;
        gpioTrig.edge_tick = hal_systick();
        return;
    }

    measureResult.type = type;
    measureResult.ticks = hal_ms_intv(gpioTrig.edge_tick);

    if(measureResult.ticks < 10)
        return;

    plus_edge_callback();
    gpioTrig.edge_tick = hal_systick();
}
```

```

}

void negedge_callback(GPIO_Pin_e pin,IO_Wakeup_Pol_e type)
{
    if(gpioTrig.enable == FALSE)
    {
        gpioTrig.enable = TRUE;
        gpioTrig.edge_tick = hal_systick();
        return;
    }
    measureResult.type = type;
    measureResult.ticks = hal_ms_intv(gpioTrig.edge_tick);

    if(measureResult.ticks < 10)
        return;

    plus_edge_callback();
    gpioTrig.edge_tick = hal_systick();
}

//初始化

void Pulse_Measure_Init( uint8 task_id )
{
    pulseMeasure_TaskID = task_id;

    hal_gpioin_register(measureResult.pin,pulse_measure_callback,pulse_measure_callback);
    gpioTrig.pinstate = hal_gpio_read(measureResult.pin);

    hal_pwrmgr_register(MOD_USR1, NULL, NULL);
    hal_pwrmgr_lock(MOD_USR1);
}

```

```
//事件处理函数

uint16 Pulse_Measure_ProcessEvent(uint8 task_id, uint16 events)
{
    if(task_id != pulseMeasure_TaskID){
        return 0;
    }

    // Discard unknown events
    return 0;
}
```

### 3.6 Timer 定时

示例 GPIO 中的 timer\_Task 演示了定时器的使用。

测试参考硬件：PHY6200\_32\_V1.4。

示例演示了 timer 的使用。

```
osal_start_timerEx( timer_TaskID, TIMER_1S_ONCE , 1000);
```

定时 1000ms，一次有效。

```
osal_start_reload_timer( timer_TaskID, TIMER_2S_CYCLE , 2000);
```

定时 2000ms，循环有效，除非遇到 osal\_stop\_timerEx 来关闭定时器。

```
//OSAL_gpio.c
const pTaskEventHandlerFn tasksArr[] =
{
    LL_ProcessEvent,
    Timer_Demo_ProcessEvent,
};
```



```
void osalInitTasks( void )
{
    uint8 taskID = 0;

    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

    LL_Init( taskID++);
    /* Application */
    Timer_Demo_Init(taskID);
}
```

```
//gpio_demo.c
static uint8 timer_TaskID;

#define TIMER_1S_ONCE                0x0001
#define TIMER_2S_CYCLE                0x0004

//初始化

void Timer_Demo_Init( uint8 task_id ){
    timer_TaskID = task_id;

    osal_start_timerEx( timer_TaskID, TIMER_1S_ONCE , 1000 );
    osal_start_reload_timer( timer_TaskID, TIMER_2S_CYCLE , 2000);
}

//事件处理

uint16 Timer_Demo_ProcessEvent( uint8 task_id, uint16 events ){
    static uint8 count1 = 0,count2 = 0;
    static bool    timer_cycle_enable = TRUE;
```

```

if(task_id != timer_TaskID){
    return 0;
}

if ( events & TIMER_1S_ONCE ){
    LOG("1s:once only mode\n");
    osal_start_timerEx( timer_TaskID, TIMER_1S_ONCE , 1000);

    if(timer_cycle_enable == FALSE){
        if(++count1 >= 10 ){
            osal_start_reload_timer( timer_TaskID, TIMER_2S_CYCLE , 2000);

            LOG("2s:recycle mode start\n");
            timer_cycle_enable = TRUE;
            count1 = 0;
        }
    }

    return (events ^ TIMER_1S_ONCE);
}

if ( events & TIMER_2S_CYCLE ){
    LOG("2s:recycle mode\n");
    if(++count2 >= 5 ){
        osal_stop_timerEx(timer_TaskID, TIMER_2S_CYCLE);

        LOG("2s:recycle mode stop\n");
        timer_cycle_enable = FALSE;
        count2 = 0;
    }

    return (events ^ TIMER_2S_CYCLE);
}

```

```
}  
  
return 0;  
}
```

### 3.7 数字按键

GPIO 做数字输入时，可配置为按键。

key.h 配置项含义：

HAL\_KEY\_NUM：按键数量。

HAL\_KEY\_EVENT：应用层需要分配给 key 一个事件处理，事件处理代码不能修改。

HAL\_KEY\_SUPPORT\_LONG\_PRESS：是否支持长按键。

HAL\_KEY\_DEBOUNCD：按键消抖时间，单位为 ms。

HAL\_KEY\_LONG\_PRESS\_TIME：长按键生效时间，单位为 ms。

gpio 注意事项：

1.P00~P03 默认为 JTAG，做 key 使用时，需要关闭 JTAG 功能。

2.P08 为模式选择引脚，不可做其他用途。

3.P09~P10 默认为 UART，如做 key 使用，可将 UART 映射到其他 io 上。

4.P16~P17 默认接 32K 晶振，如做 key 使用，需要使用内部 rc。

5.P18~P34 支持唤醒，不支持中断，不适合做 key 使用。

按键可配信息：

- 1.使用的引脚。
- 2.空闲时的电平。
- 3.回调函数，回调函数中有短按按下、短按释放、长按释放分支。
- 4.应用层需要分配 `key` 一个事件 `HAL_KEY_EVENT`，事件处理代码不做改动。
- 5.如需长按功能，需要开启 `HAL_KEY_SUPPORT_LONG_PRESS`，并添加长按处理代码。

示例 GPIO 中的 `key_Task` 演示了如何在 OSAL 中如何使用 `key`。

```
/*  
  
配置了两个 key  
  
使用的引脚是 P14 和 P15  
  
空闲时是低电平  
  
回调函数是 key_press_evt  
  
*/  
void Key_Demo_Init(uint8 task_id)  
{  
    uint8_t i = 0;  
  
    key_TaskID = task_id;  
    LOG("gpio key demo start...\n");  
  
    key_state.key[0].pin = GPIO_P14;  
    key_state.key[1].pin = GPIO_P15;
```

```
for(i = 0; i < HAL_KEY_NUM; ++i){
    key_state.key[i].state = HAL_STATE_KEY_IDLE;
    key_state.key[i].idle_level = HAL_LOW_IDLE;

    if(key_state.key[i].pin == GPIO_P16){
        hal_pwrmgr_register(MOD_USR2,NULL,P16_wakeup_handler);
        hal_gpio_cfg_analog_io(key_state.key[i].pin,Bit_DISABLE);
        LOG("P16 is used\n");
    }
    else if(key_state.key[i].pin == GPIO_P17){
        hal_pwrmgr_register(MOD_USR3,NULL,P17_wakeup_handler);
        hal_gpio_cfg_analog_io(key_state.key[i].pin,Bit_DISABLE);
        LOG("P17 is used\n");
    }
    else if((key_state.key[i].pin == GPIO_P09) || (key_state.key[i].pin == GPIO_P10)){
        uart_port_reconfig();
    }
}

key_state.task_id = key_TaskID;
key_state.key_callbank = key_press_evt;
key_init();
}
```

/\*

按键回调函数

HAL\_KEY\_EVT\_PRESS: 短按按下

HAL\_KEY\_EVT\_RELEASE: 短按释放

HAL\_KEY\_EVT\_LONG\_RELEASE：长按释放

```

*/
static void key_press_evt(uint8_t i, key_evt_t key_evt)
{
    LOG("\nkey index:%d gpio:%d ", i, key_state.key[i].pin);
    switch(key_evt)
    {
        case HAL_KEY_EVT_PRESS:
            LOG("key(press down)\n");
#ifdef HAL_KEY_SUPPORT_LONG_PRESS
            osal_start_timerEx(key_TaskID, KEY_DEMO_LONG_PRESS_EVT, HAL_KEY_LONG_PRESS_TIME);
#endif
            break;

        case HAL_KEY_EVT_RELEASE:
            LOG("key(press release)\n");
            break;

#ifdef HAL_KEY_SUPPORT_LONG_PRESS
        case HAL_KEY_EVT_LONG_RELEASE:
            hal_pwrmgr_unlock(MOD_USR1);
            LOG("key(long press release)\n");
            break;
#endif

        default:
            LOG("unexpected\n");
            break;
    }
}

```

```

/*

HAL_KEY_EVENT: key 用到的中间件代码，事件处理代码不可以修改

KEY_DEMO_LONG_PRESS_EVT: 长按按下，可根据需要配置

*/

uint16 Key_ProcessEvent( uint8 task_id, uint16 events )
{
    if(task_id != key_TaskID){
        return 0;
    }

    if( events & HAL_KEY_EVENT){
        //do not modify,key will use it

        for (uint8 i = 0; i < HAL_KEY_NUM; ++i){
            if ((key_state.temp[i].in_enable == TRUE) ||
                (key_state.key[i].state == HAL_STATE_KEY_RELEASE_DEBOUNCE)){
                gpio_key_timer_handler(i);
            }
        }

        return (events ^ HAL_KEY_EVENT);
    }
}

#ifdef HAL_KEY_SUPPORT_LONG_PRESS

    if( events & KEY_DEMO_LONG_PRESS_EVT){

        for (int i = 0; i < HAL_KEY_NUM; ++i){
            if(key_state.key[i].state == HAL_KEY_EVT_PRESS){
                LOG("key:%d gpio:%d    ",i,key_state.key[i].pin);
                LOG("key(long press down)");

                osal_start_timerEx(key_TaskID,KEY_DEMO_LONG_PRESS_EVT,HAL_KEY_LONG_PRESS_TIME);
            }
        }
    }
}

//2s

```

```
        //user app code long press down process
    }
}
return (events ^ KEY_DEMO_LONG_PRESS_EVT);
}
#endif

// Discard unknown events
return 0;
}
```