

SPC1068 公共系统指南（时钟及 IO）使用指南

概述

SPC1068 是基于 ARM Cortex M3 构架的复杂 MCU，片上集成了众多先进的外设，可以实现很多复杂的功能。这些外设都要用到的芯片资源包括 CPU、时钟及 IO，作为 SPC1068 的入门指导，这份文档重点介绍芯片的时钟及 IO。

注意： 本文档主要以 SPC1068 为例进行介绍。

目录

1	SPC1068 时钟	7
1.1	PLL 时钟	8
2	SPC1068 GPIO	11
2.1	IO 口读写功能	12
2.2	IO 口中断功能	14
2.3	IO 口抗尖峰功能	16

SPIN TROL

图片列表

图 2-1: SPC1068 时钟系统框图	7
图 2-2: PLL 功能框图.....	8
图 3-1: GPIO 框图	11
图 3-2: GPIO 读写配置图	12
图 3-3: GPIO 中断配置图	14
图 3-4: Deglitch 滤波	16

SPIN
TROL

表格列表

表 2-1: PLL 时钟频率和分频系数选择.....	8
表 3-1: GPIO 函数.....	11
表 3-2: Deglitch 时钟控制寄存器定义.....	16

SPIN TROL

版本历史

版本	日期	作者	状态	变更
C/0	2024-02-27	周佳莉	Released	首次发布。

SPIN
TROL

术语或缩写

术语或缩写	描述
MCU	Microcontroller Unit, 微控制单元

SPIN TROL

1 SPC1068 时钟

SPC1068 的时钟系统的时钟源共有三种不同的选择：

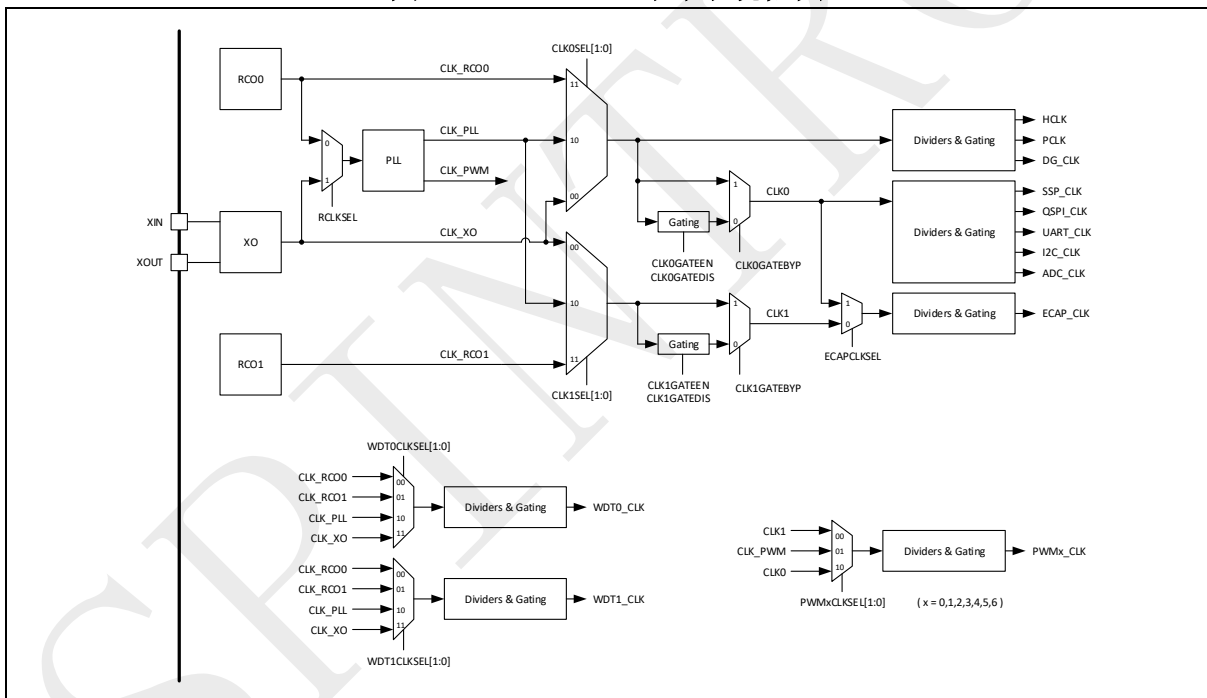
内部时钟

- RCO0: 24M Hz
- RCO1: 24M Hz

外部时钟 XO

- 无源晶振：输入范围 4 MHz ~ 10 MHz
- 时钟信号输入（有源晶振或其他时钟信号）：4 MHz ~ 56 MHz

图 1-1: SPC1068 时钟系统框图



1.1 PLL 时钟

用户可以直接使用上述三种时钟作为 SPC1068 的系统时钟源，但这样做无疑浪费了 SPC1068 的性能，因为 SPC1068 是在 150MHz 的系统时钟下工作的。如果要让 SPC1068 工作在超过或低于外部时钟源的速度时，用户需要用到 SPC1068 内部的 PLL 电路。

从图 2-2 可以看出，PLL 电路的输入可以选为内部（默认）或外部的时钟源，然后通过分频、锁相等步骤，获取到其最终频率。PLL 时钟频率的计算公式如下：

$$f_{PLL} = \frac{f_{IN}}{NIN} \times \left(NFB_INT + \frac{NFB_FRAC}{65536} \right) \quad (1)$$

其中， $4MHz \leq \frac{f_{IN}}{NIN} \leq 8MHz$ 。

图 1-2: PLL 功能框图

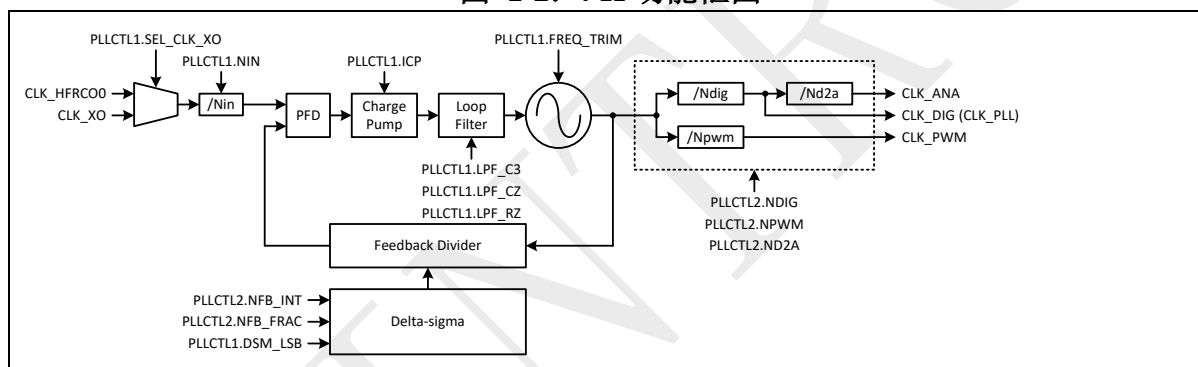


表 1-1: PLL 时钟频率和分频系数选择

Desired Output (FDIG/FPWM)	Frequency	Dividing Ratio (NDIG/NPWM)
400MHz ~ 600MHz		1
200MHz ~ 300MHz		2
150MHz ~ 200MHz		3
100MHz ~ 150MHz		4

例 1: 使用内部 RC 时钟，获取 150MHz 系统频率的配置

SPC1068 内部 RC 时钟的频率为 24MHz，根据公式 (1) 的要求，需要把 NIN 选为 3~6，这里选择为 4，而 NDIG 需要选为 3 或 4，我们先选择为 3：

$NIN = 3$

$NDIG = 3$

根据计算公式 (1)，得出 $\left(NFB_INT + \frac{NFB_FRAC}{65536} \right) = 75$ ，于是选择


```
NFB_INT = 75
```

```
NFB_FRAC = 0
```

具体配置代码如下：

Example Code

```
/* Select RCO0 as PLL input clock source */
CLOCK->PLLCTL0.bit.RCLKSEL = PLLCTL0_BIT_RCLKSEL_RCO0;
/* NIN = 3 */
CLOCK->PLLCTL0.bit.NIN = 3;
/* NDIG = 3 */
CLOCK->PLLCTL1.bit.NDIG = 3;

/* PLL闭环 */
CLOCK->PLLCTL0.bit.LOOPCLOSE = 1;
/* NFB_INT = 150 */
CLOCK->PLLCTL1.bit.NFBINTG = 75;
/* NFB_FRAC = 0 */
CLOCK->PLLCTL1.bit.NFBFRAC = 0;

/* PLL Digital clock output enable */
CLOCK->PLLCTL0.bit.DCLKEN = 1;
/* PLL enable */
CLOCK->PLLCTL0.bit.EN = 1;

/* Wait PLL ready */
while(CLOCK->CLKSTS.bit.PLLDRDY == 0) {};

/* Set PLL as the system clock */
CLOCK->GLBCLKCTL.bit.CLK0SEL = GLBCLKCTL_BIT_CLK0SEL_PLL;
CLOCK->GLBCLKCTL.bit.CLK1SEL = GLBCLKCTL_BIT_CLK1SEL_PLL;
```

或者简单的使用函数库函数：

```
void CLOCK_InitWithRCO(CLOCK_HCLKSelEnum u32HCLKSel);
```

来进行系统时钟设置，该函数覆盖的应用场景为：

- 使用内部 RCO 时钟源
- 目标主频为 MHz 的整数倍

这时，用户只需要简单的调用函数

```
CLOCK_InitWithRCO(CLOCK_HCLK_150MHZ);
```

就可以完成上述设置。

例 2：使用外部 8MHz 无源晶振，获取 150MHz 系统频率的配置

SPC1068 使用外部晶振的频率为 8MHz，根据公式（1）的要求，需要把 NIN 选为 1 或 2，而 NDIG 需要选为 3 或 4，我们先选择：

NIN = 1

NDIG = 3

根据计算公式（1），得出 $(NFB_INT + \frac{NFB_FRAC}{65536}) = 75$ ，于是选择：

NFB_INT = 75

NFB_FRAC = 0

具体配置代码如下：

Example Code

```

/* Select XO as PLL input clock source */
CLOCK->PLLCTL0.bit.RCLKSEL = PLLCTL0_BIT_RCLKSEL_XO;
/* NIN = 1 */
CLOCK->PLLCTL0.bit.NIN = 1;
/* NDIG = 3 */
CLOCK->PLLCTL1.bit.NDIG = 3;

/* PLL闭环 */
CLOCK->PLLCTL0.bit.LOOPCLOSE = 1;
/* NFB_INT = 150 */
CLOCK->PLLCTL1.bit.NFBINTG = 75;
/* NFB_FRAC = 0 */
CLOCK->PLLCTL1.bit.NBFRAC = 0;

/* PLL Digital clock output enable */
CLOCK->PLLCTL0.bit.DCLKEN = 1;
/* PLL enable */
CLOCK->PLLCTL0.bit.EN = 1;

/* Wait PLL ready */
while (CLOCK->CLKSTS.bit.PLLRDY == 0) {};

/* Set PLL as the system clock */
CLOCK->GLBCLKCTL.bit.CLK0SEL = GLBCLKCTL_BIT_CLK0SEL_PLL;
CLOCK->GLBCLKCTL.bit.CLK1SEL = GLBCLKCTL_BIT_CLK1SEL_PLL;

```

2 SPC1068 GPIO

SPC1068 的 GPIO 是一种多功能的 IO 接口，从下图可以看出，IO 口的功能主要可以分为两部分：

- GPIO 功能：在 CPU 的控制下接收或输出高低电平
- 与其他外设相连接：把 IO 口上的电平状态传送给与其相连的外设，或者把指定的外设输出连接到 IO 口

除此之外，IO 口还有产生 IO 中断的能力。

图 2-1: GPIO 框图

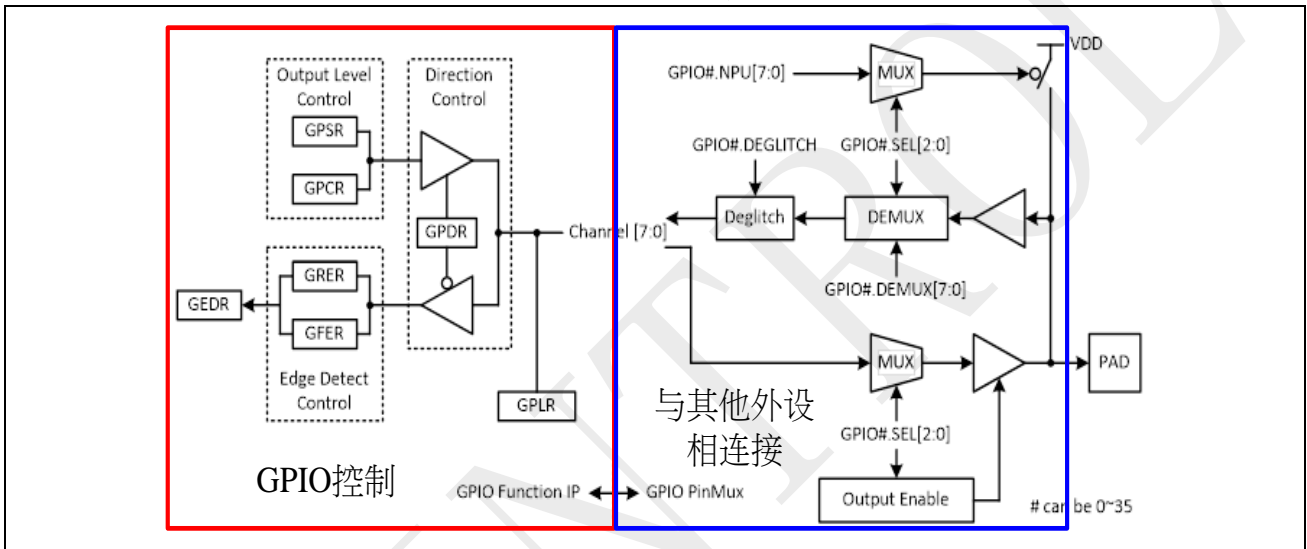
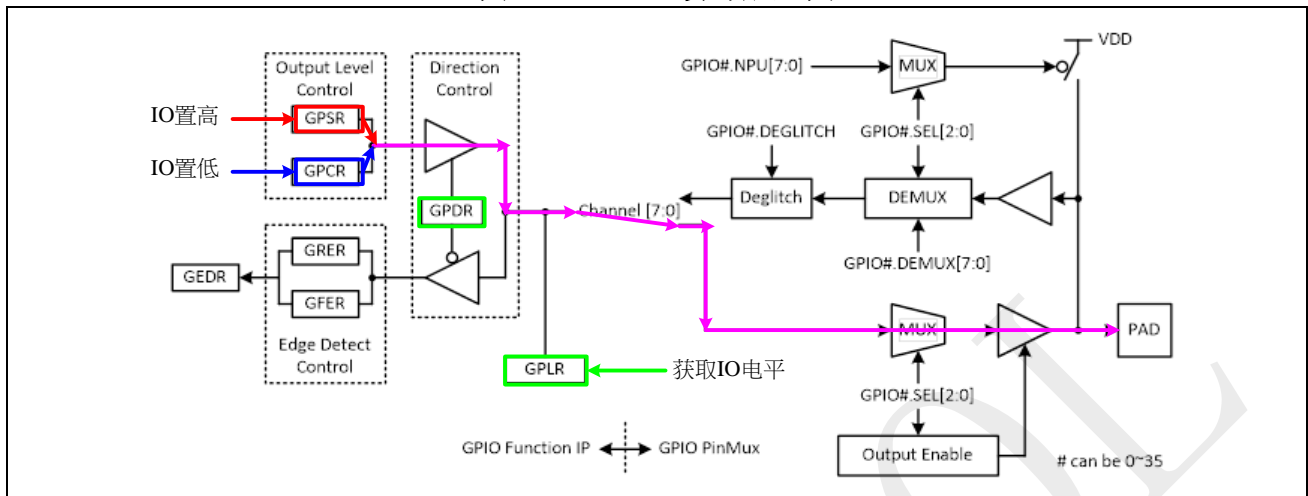


表 2-1: GPIO 函数

API 名称	API 功能描述
GPIO_WritePin(ePinNum, bitVal)	把序号为 ePinNum 的 IO 口设置为 bitVal 指定的电平状态
GPIO_ReadPin(ePinNum)	获取序号为 ePinNum 的 IO 口电平高低状态
GPIO_TogglePin(ePinNum)	转换序号为 ePinNum 的 IO 口电平高低状态
GPIO_SetPinDir(GPIO_PinEnum ePinNum, GPIO_DirEnum eDir)	设置序号为 ePinNum 的 IO 口为输入或者输出管脚
GPIO_SetEdgeIntMode(GPIO_PinEnum ePinNum, GPIO_EdgeIntEnum eIntType)	设置序号为 ePinNum 的 IO 事件类型： (1) 上升沿事件 (2) 下降沿事件 (3) 双边沿事件 (4) 不产生中断
GPIO_EnableEdgeInt(ePinNum)	使能序号为 ePinNum 的 IO 口边沿触发中断功能
GPIO_DisableEdgeInt(ePinNum)	关闭序号为 ePinNum 的 IO 口边沿触发中断功能
GPIO_GetEdgeIntStatus(ePinNum)	获取序号为 ePinNum 的 IO 口边沿触发中断状态
GPIO_ClearEdgeInt(ePinNum)	清除序号为 ePinNum 的 IO 口边沿触发中断状态标志
GPIO_EnableDeglitch(ePinNum)	使能序号为 ePinNum 的 IO 口的 Deglitch 功能
GPIO_DisableDeglitch(ePinNum)	关闭序号为 ePinNum 的 IO 口的 Deglitch 功能
GPIO_SetDeglitchWindow(eWindow)	设置序号为 ePinNum 的 IO 口的 Deglitch 功能窗口大小

2.1 IO 口读写功能

图 2-2: GPIO 读写配置图



SPC1068 的所有 IO 都可以设定为 GPIO 状态，即用 CPU 去控制 IO 口的电平状态，并通过 CPU 读取 IO 口的状态。如果需要设定 IO 口为 GPIO 输出功能，需要把 GPIO Pin Configuration Register 的 Channel[7:0] 设置为 0，即选择 Channel[0]，同时通过 GPDR 寄存器把 IO 口设置为输出。这两个设置现在可以用以下两个封装好的函数来实现：

```
GPIO_SetPinChannel(GPIO_X, PINMUX_CHANNEL_0) // 选定 GPIO_X 为 Channel[0]
```

```
GPIO_SetPinDir(GPIO_X, GPIO_OUTPUT/ GPIO_INPUT) // 设定 GPIO_X 为输入或输出
```

当设置成为 GPIO 状态时候，信号的通路如图 2-2 所示的彩色路线所示。CPU 可以通过设定以下两个寄存器改写 IO 口的状态：

GPSR：设定 IO 电平状态为高

GPCR：设定 IO 电平状态为低

现在这两个寄存器的设置已经被封装成函数：

```
GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_HIGH/ GPIO_LEVEL_LOW)
```

而 IO 口电平状态的获取，则可以通过寄存器 GPLR 来得知，对这个寄存器的访问，现在也封装成函数：

```
GPIO_ReadPin(GPIO_X)
```

当返回为 GPIO_LEVEL_HIGH 时，电平为高，反之，若返回为 GPIO_LEVEL_LOW，则电平为低。

在下面的例子中，会根据一个 IO 口的电平状态，去更改另一个 IO 口反转的频率，进而去控制一个 LED 灯的闪烁。

配置代码如下：

Example Code

```
/* Select GPIO pin */
GPIO_InX = GPIO_1;
GPIO_OutX = GPIO_19;

/* Set GPIO pin function */
GPIO_SetPinAsGPIO(GPIO_InX);
GPIO_SetPinAsGPIO(GPIO_OutX);

/* Set GPIO direction */
GPIO_SetPinDir(GPIO_InX, GPIO_INPUT);
GPIO_SetPinDir(GPIO_OutX, GPIO_OUTPUT);

while(1)
{
    /* If GPIO_InX = 1, toggle GPIO_X every 100 ms */
    if(GPIO_ReadPin(GPIO_InX) == GPIO_LEVEL_HIGH)
    {
        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_HIGH);
        Delay_ms(100);

        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_LOW);
        Delay_ms(100);

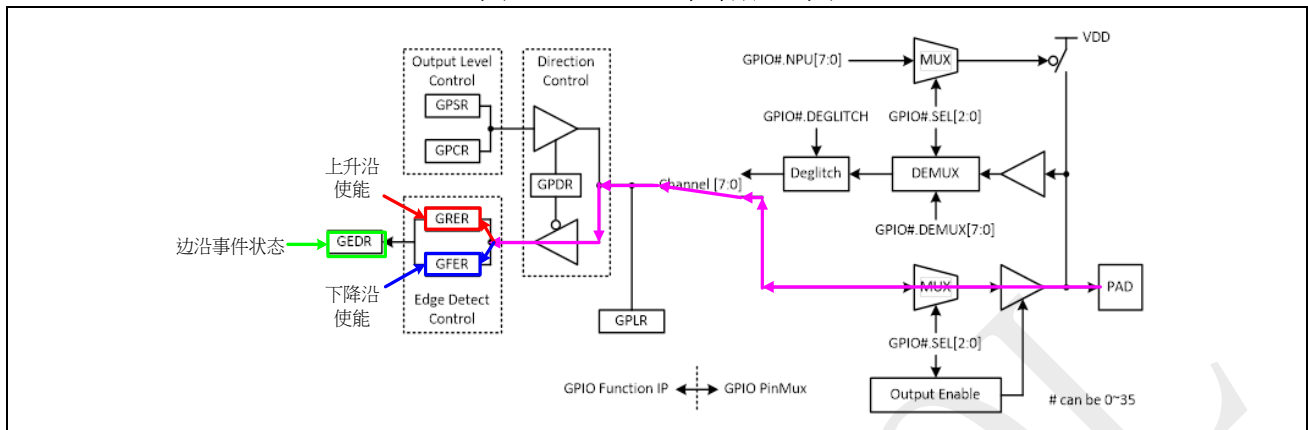
        printf("Toggle GPIO_%d every 200 ms\n", GPIO_OutX);
    }
    else /* If GPIO_InX = 0, toggle GPIO_X every 500 ms */
    {
        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_HIGH);
        Delay_ms(500);

        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_LOW);
        Delay_ms(500);

        printf("Toggle GPIO_%d every 1 second\n", GPIO_OutX);
    }
}
```

2.2 IO 口中断功能

图 2-3: GPIO 中断配置图



SPC1068 的所有 GPIO 都可以使能 IO 口中断功能，其中所有的 IO 口中断都可以设定为：

- 上升沿触发中断
- 下降沿触发中断
- 上升沿和下降沿触发中断

具体的信号通路如上图中的彩色路径，设置 IO 口为 GPIO 功能，通过 GRER/GFER 两个寄存器分别设定 IO 口为上升沿触发中断/下降沿触发中断，这个功能现在也封装成为了函数，可以按照以下三步进行调用：

- 设置边沿触发类型 `GPIO_SetEdgeIntMode(GPIO_InX, GPIO_INT_BOTH_EDGES)`
- 使能边沿触发中断 `GPIO_EnableEdgeInt(GPIO_InX)`
- 使能 CPU 中断 `NVIC_EnableIRQ(GPIO_EDGE_IRQn)`

下面的例子和上一个例子非常相似，也是通过对 GPIO1 的电平判断来实现对 GPIO19 的转换频率进行控制，所不同的是，上一个例子是通过软件查询的办法来实现，这个例子里将采用 IO 中断的方式。

具体代码如下：

Example Code

```

{
GPIO_InX = GPIO_1;
GPIO_OutX = GPIO_19;

/* Set GPIO pin function */
GPIO_SetPinAsGPIO(GPIO_InX);
GPIO_SetPinAsGPIO(GPIO_OutX);
/* Set GPIO direction */
GPIO_SetPinDir(GPIO_InX, GPIO_INPUT);
GPIO_SetPinDir(GPIO_OutX, GPIO_OUTPUT);
/* GPIO Interrupt configuration: Rising and Falling edge will trigger interrupt */
GPIO_SetEdgeIntMode(GPIO_InX, GPIO_INT_BOTH_EDGES);
    
```

```
GPIO_EnableEdgeInt(GPIO_InX);
/* Enable GPIO Edge-trigger interrupt */
NVIC_EnableIRQ(GPIO_EDGE_IRQn);

while(1)
{
    /* If GPIO_InX = 1, toggle GPIO_OutX every 100 ms */
    if(u8Flag)
    {
        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_HIGH);
        Delay_ms(100);

        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_LOW);
        Delay_ms(100);

        printf("Toggle GPIO_%d every 200 ms\n", GPIO_OutX);
    }
    /* If GPIO_InX = 0, toggle GPIO_OutX every 500 ms */
    else
    {
        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_HIGH);
        Delay_ms(500);

        GPIO_WritePin(GPIO_OutX, GPIO_LEVEL_LOW);
        Delay_ms(500);

        printf("Toggle GPIO_%d every 1 second\n", GPIO_OutX);
    }
}

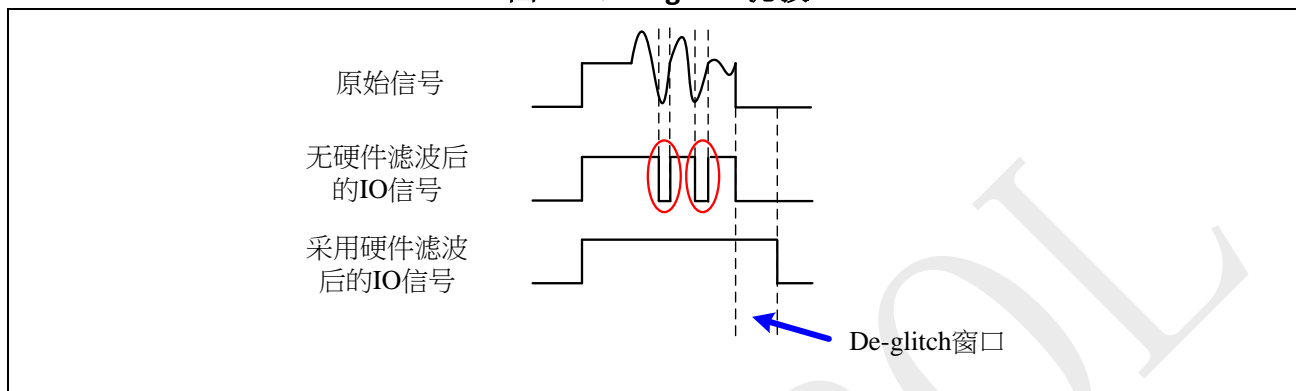
/* GPIO Edge Interrupt Handler */
void GPIOEdge_IRQHandler()
{
    if(GPIO_ReadPin(GPIO_InX))
        u8Flag = 1;
    else
        u8Flag = 0;

    GPIO_ClearEdgeInt(GPIO_InX);
}
```

2.3 IO 口抗尖峰功能

在实际应用过程中，常常遇到 IO 信号不是非常稳定，容易被外部强信号干扰。这时候一般的解决方法是对外部信号进行滤波，这要增加外部的电容或电阻。

图 2-4: Deglitch 滤波



在 SPC1068 中，IO 口带有对外部信号进行硬件滤波的单元，可以消除尖峰干扰带来的影响。这个可以通过简单的通过配置 De-glitch 寄存器来实现。

比如在上图中，原始信号是一个开关信号，但是开关过程中有可能带有图中的很强的干扰信号，如果不经过滤波，IO 口采集到的信号就会出现波动，如果用这个信号来做保护或其他重要的实时功能，可能会造成系统的误动作。

如果使用了 De-glitch 的滤波功能，就能起到消除尖峰，抗干扰的作用。可以看出在第三个信号中，由于前两个信号宽度过低，所以被硬件滤波处理掉，而稳定的信号在最终反映在 IO 的值上。

这个 De-glitch 窗口的大小，可以通过调整 Deglitch Clock 分频值来设定，这个也要根据实际信号情况来确定。可供调整的值，请见下表：

表 2-2: Deglitch 时钟控制寄存器定义

Bits	Field Name	Type	Reset	Description
31:3	RESERVED_31_3	RO	0x0	Reserved
2:0	DIV	RW	0x0	De-glitch clock dividing ratio from PCLK 000: Divide by 1 001: Divide by 2 010: Divide by 4 011: Divide by 8 100: Divide by 16 101: Divide by 32 110: Divide by 64 111: Divide by 128

比如在上表中，如果选定 DIV= 0b010，那么 SPC1068 的 IO 单元会对外部信号进行 4 个时钟周期的滤波，可以消除 4 个时钟周期以内的干扰。

在下面的例子里，GPIO28 被配置成为 PWM 输出，GPIO1 配置为 IO 输入，把 PWM 输出到 IO 输入。在 De-glitch 打开的情况下，若 PWM 的频率过高，则信号稳定的时间会很多，若短到少于 De-glitch 配置的宽度，则无法激发 IO 口中断，若 PWM 频率降低，信号稳定时间超过 De-glitch 配置的宽度，则能够正常激发 IO 口中断。

具体代码实现如下：

Example Code

```
#define GPIO_InX      (GPIO_1)
#define GPIO_OutX     (GPIO_19)

{
    uint16_t u16PrdVal = 0;

    /* Enable PWM1 Clock */
    CLOCK_EnableModule(PWM1_MODULE);

    /* Set GPIO as PWM1 output pin */
    GPIO_SetPinChannel(GPIO_28, GPIO28_PWM1A);

    /* Set frequency of PWM1 */
    u16PrdVal = 500; /* It can enter interrupt */
    // u16PrdVal = 40; /* It can not enter interrupt */
    PWM1->TBPRD.all = u16PrdVal;
    PWM1->CMPA.all = u16PrdVal / 2; /* PWM Duty = 50% */

    /* Following 3 configurations are by default */
    PWM1->TBCTL.bit.PRDLTD = TBCTL_BIT_PRDLTD_TBPRD_FROM_SHADOW;
    PWM1->CMPCTL.bit.SHDWAMODE = CMPCTL_BIT_SHDWAMODE_SHADOW_MODE;
    PWM1->CMPCTL.bit.LOADAMODE = CMPCTL_BIT_LOADAMODE_LOAD_ON_ZERO;

    /* Configure output action */
    PWM1->AQCTLA.bit.CAU = AQCTLA_BIT_CAU_TOGGLE;
    PWM1->AQCTLA.bit.PRD = AQCTLA_BIT_PRD_TOGGLE;

    /* Start PWM1 */
    PWM_CounterRunControl(PWM1, COUNT_UP_AND_RUN);

    /* Set GPIO pin function */
    GPIO_SetPinAsGPIO(GPIO_InX);
    GPIO_SetPinAsGPIO(GPIO_OutX);

    /* Set GPIO direction */
    GPIO_SetPinDir(GPIO_InX, GPIO_INPUT);
    GPIO_SetPinDir(GPIO_OutX, GPIO_OUTPUT);

    /* De-glitch settings, if IO signal state in 128 PCLK cycles, then
    it is assumed valid */
    GPIO_SetDeglitchWindow(GPIO_DEGLITCH_WINDOW_128);
    GPIO_EnableDeglitch(GPIO_InX);
}
```

```
/* GPIO Interrupt configuration: Rising edge will trigger
interrupt */
GPIO_SetEdgeIntMode(GPIO_InX, GPIO_INT_RISING_EDGE);
GPIO_EnableEdgeInt(GPIO_InX);

/* Enable GPIO Edge-triggered interrupt in CPU side */
NVIC_EnableIRQ(GPIO_EDGE_IRQn);

while(1)
{
}

void GPIOEdge_IRQHandler()
{
/* Toggle GPIO */
GPIO_TogglePin(GPIO_OutX);

printf("x");

/* Clear interrupt flag */
GPIO_ClearEdgeInt(GPIO_InX);
}
```