AW31N 用户手册

珠海市杰理科技股份有限公司

Zhuhai Jieli Technologyco.,LTD

2024年4月

目录

1 序言

这款芯片专为智能家居和物联网控制等无线数传智能设备设计，其卓越性能源自集成的192 MIPS 32位高性能处理器，并配备了高效的Cache。芯片内置的蓝牙BLE收发器严格遵循v5.4规范，不仅支持超低功耗蓝牙广播和连接待机功能，还能实现组网传输，展现出色的低延时和远距离数据传输能力，充分满足各类无线数传和智能控制的数据传输需求。此外，该芯片还配备了先进的PMU模块，集成了LDO与DCDC技术，并提供了多种低功耗工作模式，实现低至100 nA的深度睡眠电流，有效延长设备使用寿命。同时，芯片内置了丰富的外设接口，包括USB、按键ADC、IIC、SPI、TIMER、UART、CAN和红外收发等，为各种智能设备提供了全面的连接和控制解决方案。

2 总体介绍

2.1 概述

SYSTEM

- 32bit CPU 160MHz
- Support MATH/AES128
- I-cache
- Support EMU
- On-chip SRAM 32kbyte
- Support MPU
- Support UDMA
- Built-In Flash
- 24MHz crystal oscillator
- Internal low jitter low power RC oscillator
- Internal PLL

Bluetooth

- BLE5.4 +2.4GHz-Proprietary (QDID 223418)
- Support AoA Transmitter
- Support long range BLE
- Maximum transmitting power 9dBm
- Receiver sensitivity
  -96dBm @BLE-1Mbps
  -93dBm @BLE-2Mbps
  -98dBm @BLE-S2
  -103.5dBm @BLE-S8

Peripherals

- 1 x Full speed USB
- 4 x Multi-function 32bit timer
- 1 x IR RX/TX
- 3 x UART interface
- 1 x IIC Master/Slave interface
- 2 x SPI Master/Slave interface
- 1 x QDEC
- 4 x MCPWM
- 2 x LEDC
- 1 x 10bit ADC(15 Channel)
- 12 x GPIO Support function remapping

PMU

- Support temperature sensor
- VPWR range 2.7V to 5.5V
- IOVDD range 1.8V to 3.6V
- Deep sleep mode (IOVDD @3.0V)
  170nA (External wakeup)
  1.37uA (32kHz RC OSC+wakeup)
  2.66uA (32kHz RC OSC+wakeup+16k retention SRAM)

Packages

- QFN20(3mm*3mm)

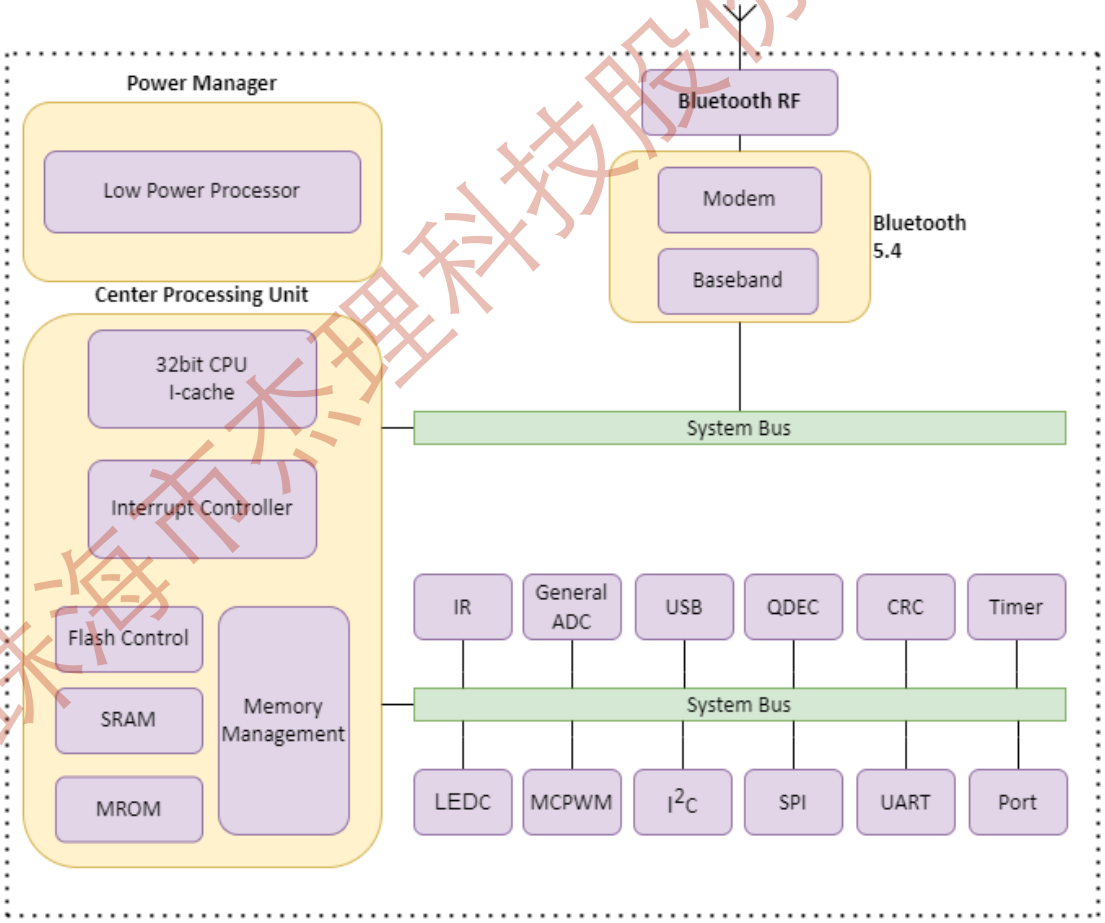
Temperature

- Operating temperature
  TC = -20°C to +85°C(standard range)
  TC = -40°C to +105°C(extended range)
- Storage temperature -65°C to +150°C

Applications

- Mouse devices
- Non-audio remote controller
- Selfie stick
- Page turner
- Adaptive USB
- Bluetooth moudle
- Price tag and other diversified IOT product

2.2 系统框图



3 CLOCK_SYSTEM

3.1 原生时钟源

本芯片具备的原生时钟源如下表所示：

时钟	概述
rc_250k	内置RC振荡器，用于复位系统和watch dog功能时钟。
rc_16m	内置RC振荡器，用于系统启动的初始时钟。
lrc_200k	内置低温度电压漂移RC振荡器，用于低功耗计数和PLL参考时钟。
erc_32k	内置RC振荡器，用于低功耗模式下提供时钟
btosc_24m	外挂12-26MHz晶振，可用于系统时钟和PLL参考时钟。

3.2 衍生时钟源

来自片内的SYSPLL，SYSPLL工作频率范围为96MHz~240MHz，输出PLL_D1P0，PLL_D1P5，PLL_D2P0，PLL_D2P5，PLL_D3P5的时钟，如果SYSPLL配置为192MHz，则能输出192MHz、128MHz、120MHz、76.8MHz和54.86MHz；每个时钟都有独立的使能端。在不使用该时钟时，软件应关闭其使能端以防止额外电源消耗。

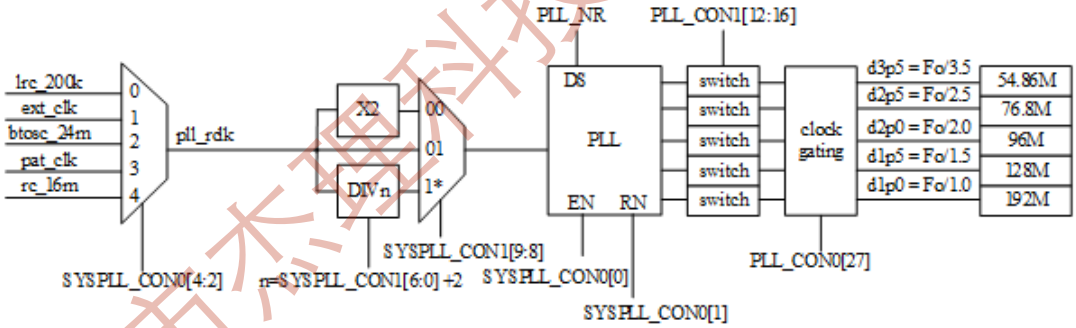


图1、SYSPLL时钟结构框图

Fout = Fref / n * m;　（n = SYSPLL_CON1[6:0]+2，m = PLL_NR[11:0]+2）

Fout 建议值为192MHz，±10%

reference clock	ref mode	NF	NR vco 192
lrc_200k	200K	1	960
ext_clk(4M)	2M	2	96
btosc_24m	12M	2	16
rc_16m	2M	8	16

表1、SYSPLL参考时钟配置表

3.3 标准时钟

除上述的原生时钟外，芯片内还有频率固定的标准时钟，分别是从rc_250k分频来的wclk；pll_96m、std_48m、std_24m和std_12m，其时钟结构如下图所示。

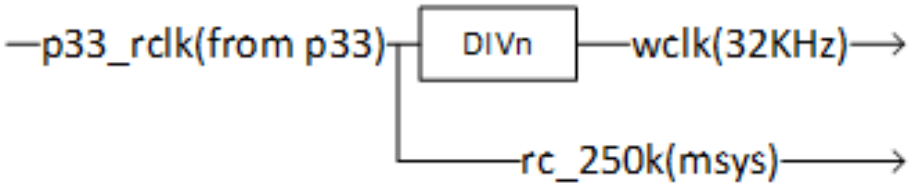


图2、系统rc_250k时钟结构

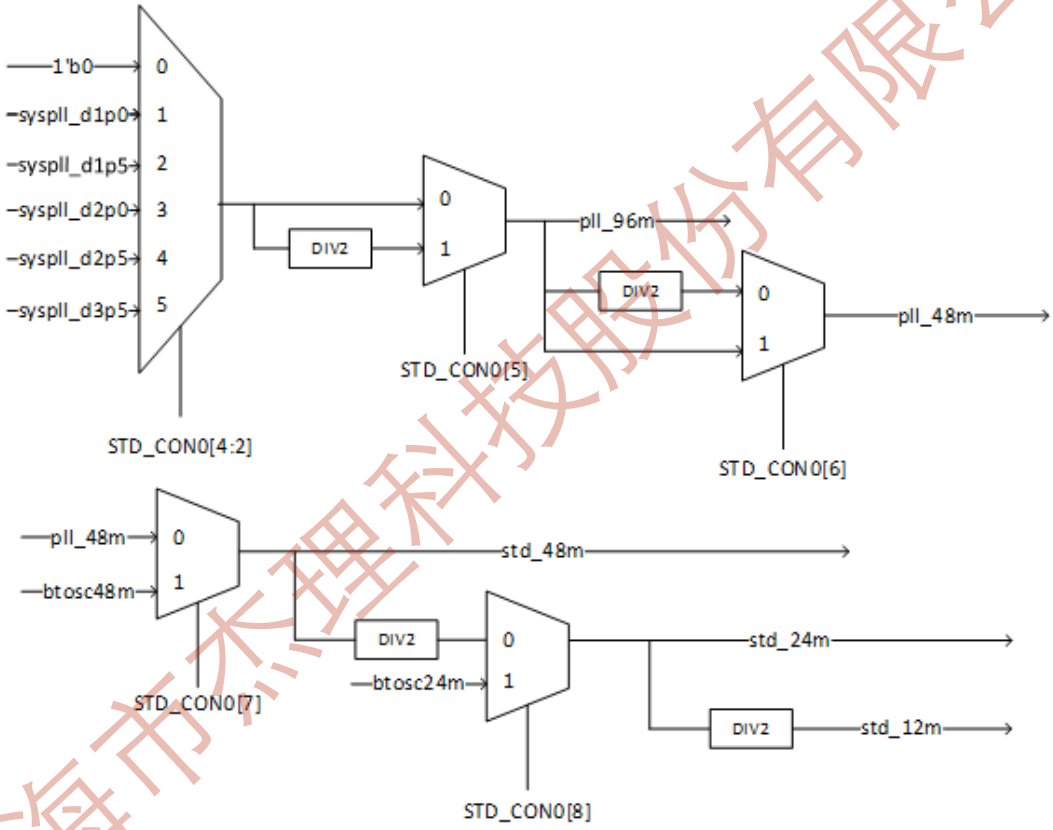
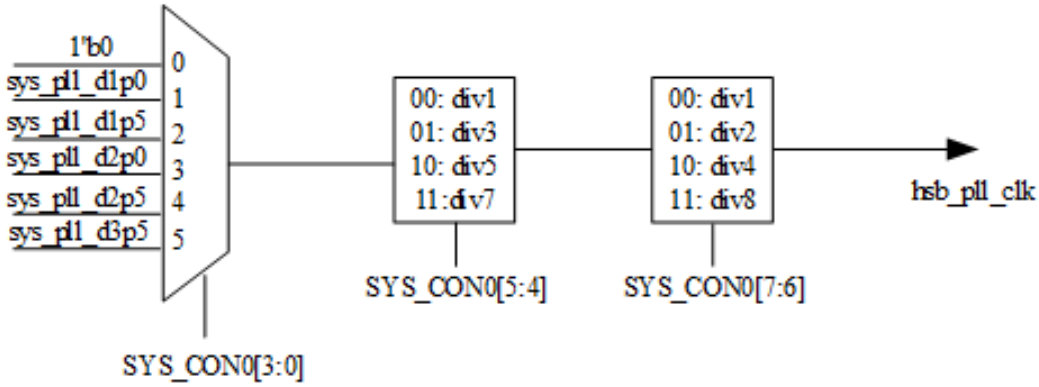


图3、STD_CLK时钟结构图

3.4 系统时钟

系统时钟部分电路框图如下，系统运行过程中可随时更改hsb_clk，lsb_clk的分频值，但需确保在任何时刻，各分频时钟都不会超过其允许的最高运行频率。



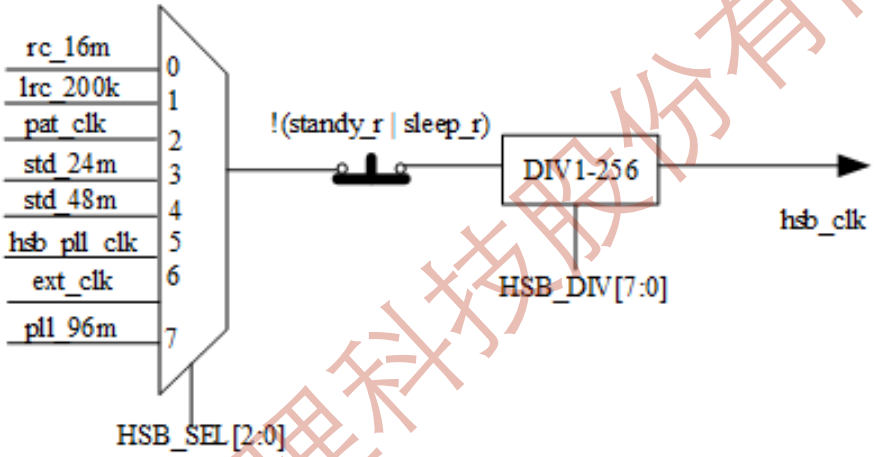
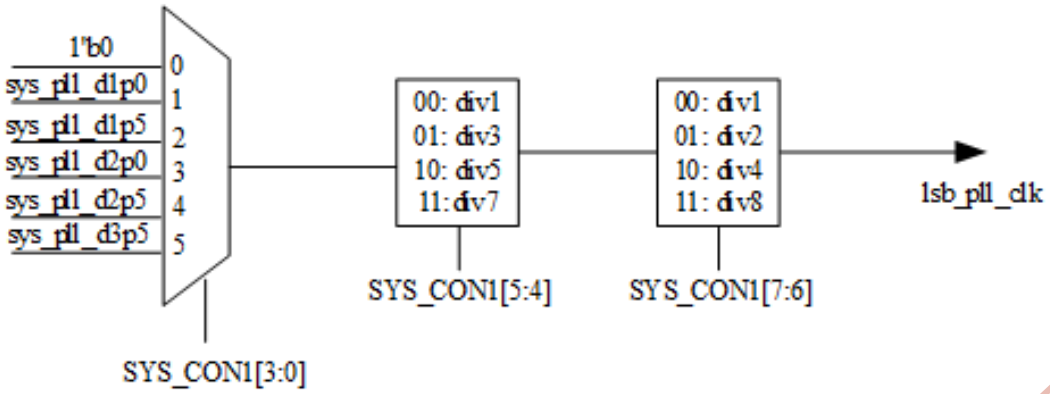


图4、hsb_clk时钟结构图



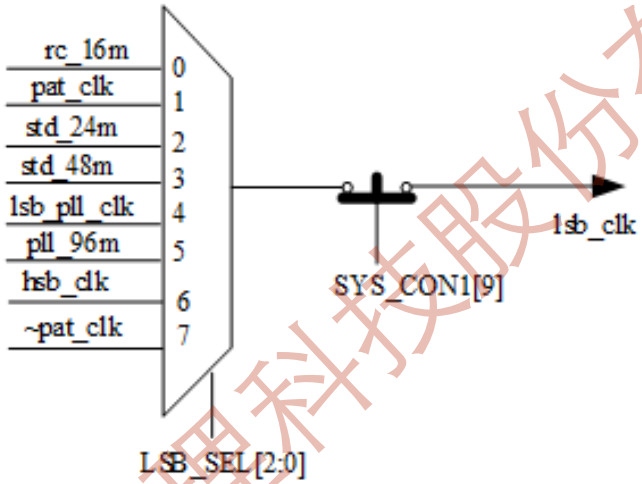
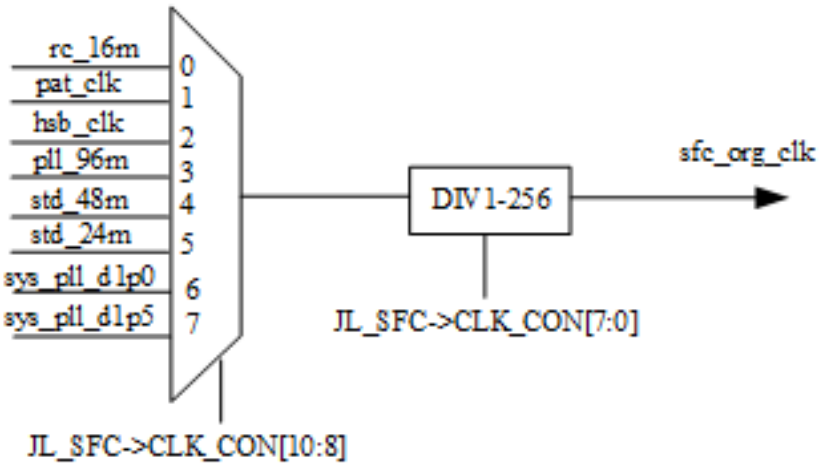


图5、lsb_clk时钟电路

【注意】：LSB_SEL 不能连续切换，两次切换中间需要等待5个LSB_CLK；

3.5 异步外设时钟



【注意】：SFC_CLK可动态切换，但不能连续切换，两次切换中间需要等待大于5个SFC_CLK；

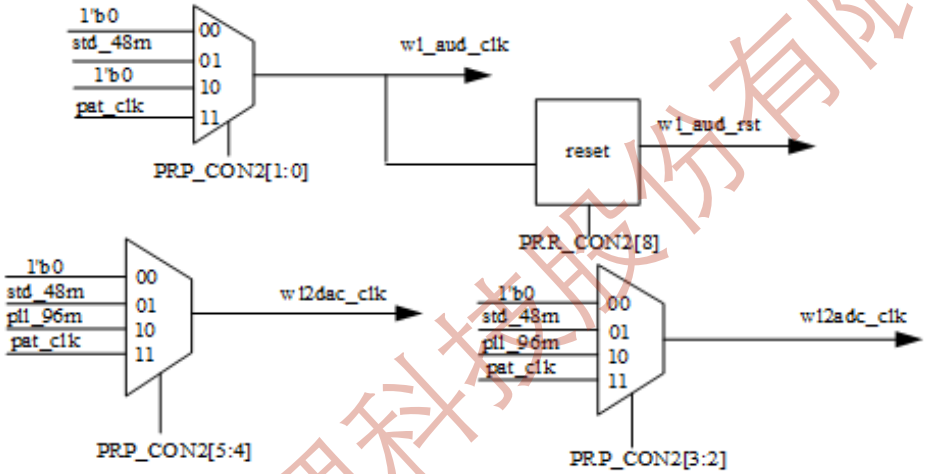


图6、高速外设时钟结构图

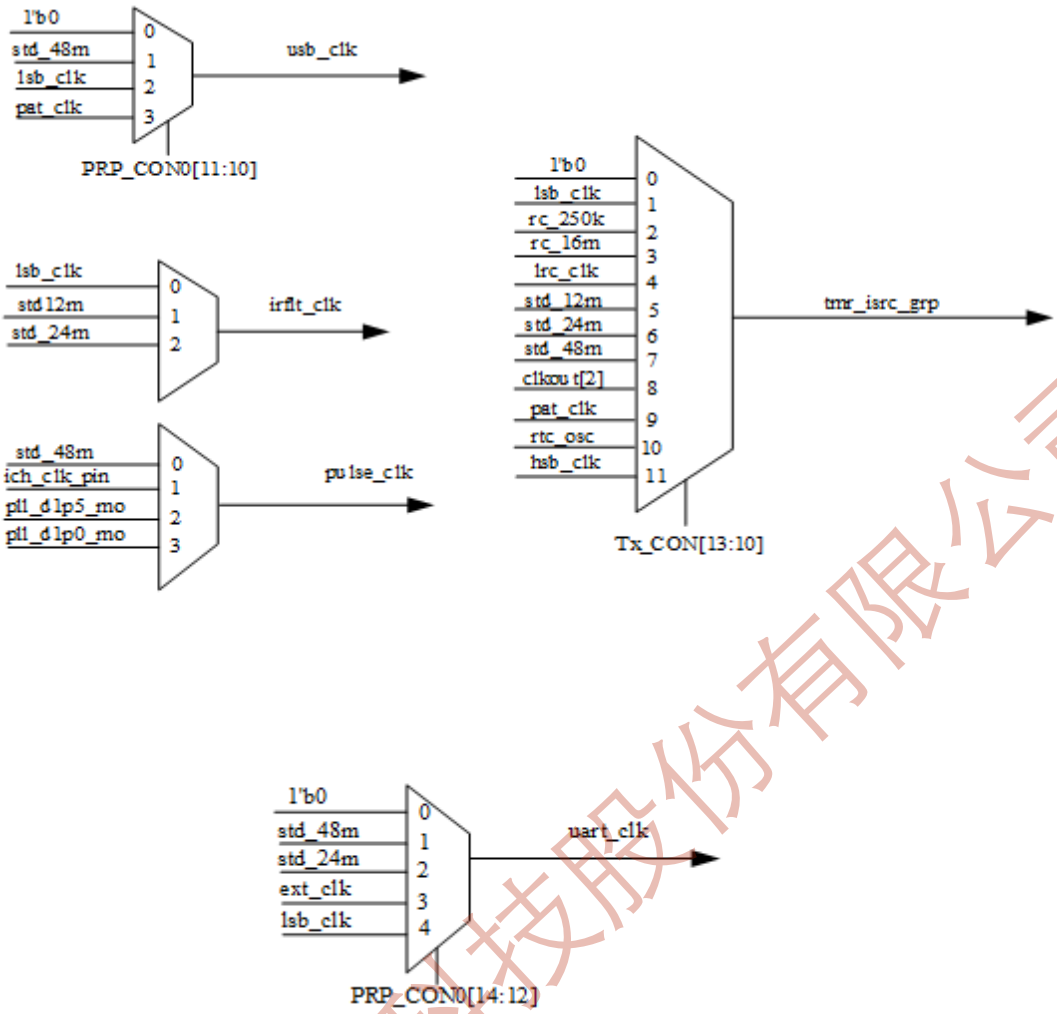


图7、低速外设时钟结构图

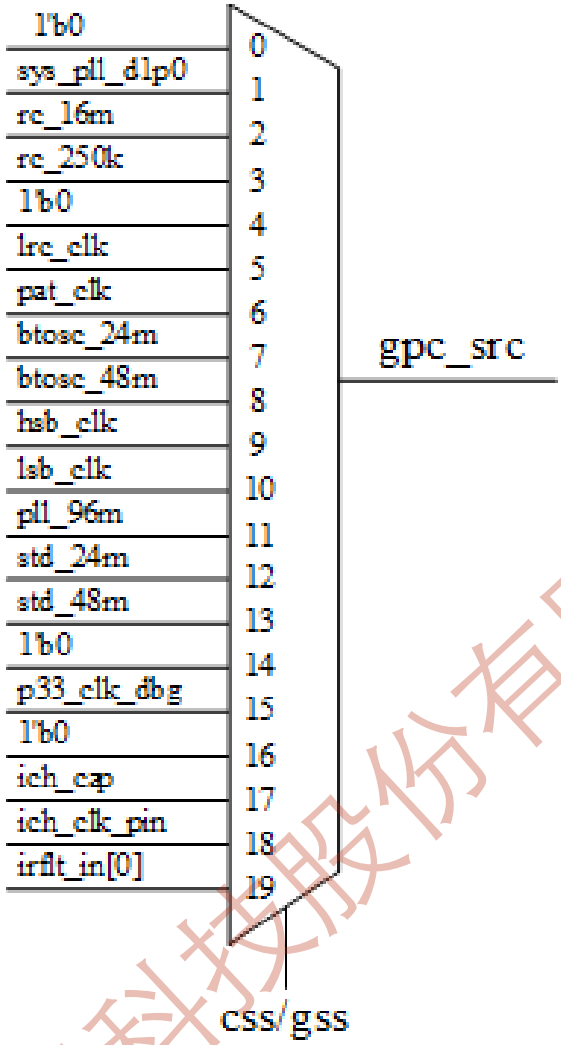


图8、gpcnt计数时钟源

1. 该芯片有3组时钟输出，可以通过crossbar将内部的时钟信号驱动至片外，用于测试或特殊用途。

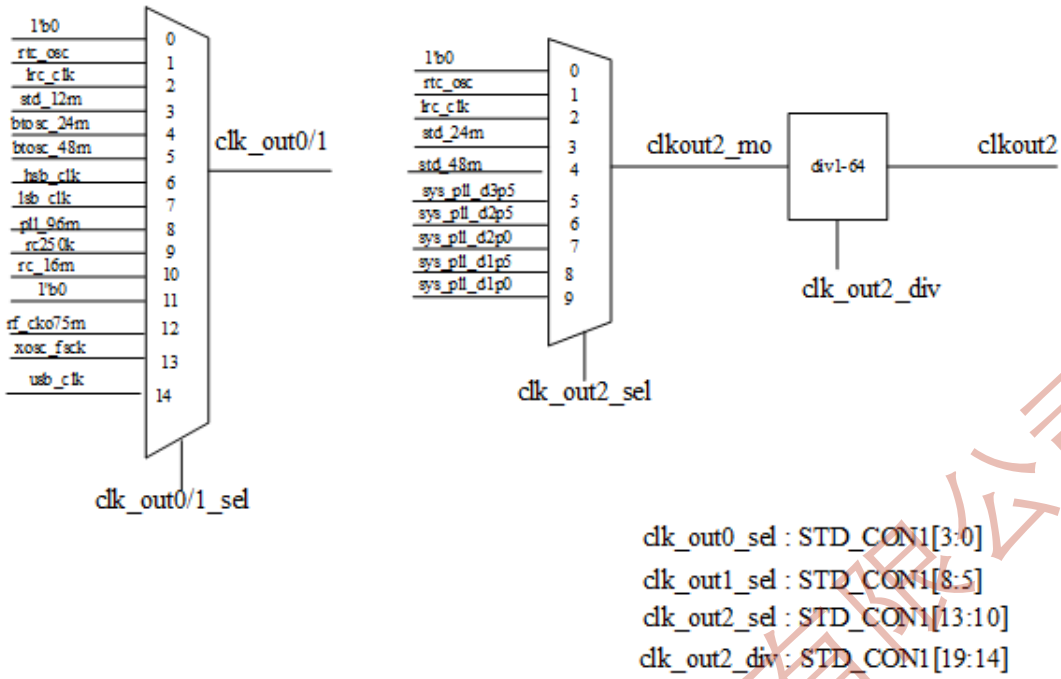


图9、IO输出时钟电路

3.6 数字模块控制寄存器

3.6.1 SYS_CON0：system config register 0

Bit	Name	RW	Default	RV	Description
31-11	RESERVED	-	0	-	预留
8	HSB_SFR_CKMD	rw	0	-	hsb_sfr_clk时钟模式选择： 0：有SFR写的时候，hsb_sfr_clk才使能 1：无论有无SFR写，hsb_sfr_clk都使能
7-4	HSB_PLL_DIV	rw	0	-	HSB_PLL时钟分频，见图4
3-0	HSB_PLL_SEL	rw	0	-	HSB_PLL时钟选择，见图4

3.6.2 SYS_CON1：system config register 1

Bit	Name	RW	Default	RV	Description
31-10	RESERVED	-	0	-	预留
9	LSB_CKEN	rw	1	-	LSB时钟使能，默认开
8	LSB_SFR_CKMD	rw	0	-	lsb_sfr_clk时钟模式选择： 0：有SFR写的时候，lsb_sfr_clk才使能 1：无论有无SFR写，lsb_sfr_clk都使能

Bit	Name	RW	Default	RV	Description
7-4	LSB_PLL_DIV	rw	0	-	LSB_PLL时钟分频，见图5
3-0	LSB_PLL_SEL	rw	0	-	LSB_PLL时钟选择，见图5

3.6.3 HSB_DIV：hsb clock div register

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	0	-	预留
7-0	HSB_DIV	rw	0	-	HSB时钟分频，见图4

3.6.4 HSB_SEL：hsb clock sel register

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	0	-	预留
2-0	HSB_SEL	rw	0	-	HSB时钟选择，见图4

3.6.5 LSB_SEL：lsb clock sel register

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	0	-	预留
2-0	LSB_SEL	rw	0	-	LSB时钟选择，见图5

3.6.6 STD_CON0：standard clock register 0

Bit	Name	RW	Default	RV	Description
31-7	RESERVED	-	0	-	预留
8	STD_24M_SEL	rw	0	-	STD_24M时钟源选择，见图3
7	STD_48M_SEL	rw	0	-	STD_48M时钟源选择，见图3
6	PLL_48M_DS	rw	0	-	PLL_48M DIV2使能选择，见图3
5	PLL_96M_DS	rw	0	-	PLL_96M DIV2使能选择，见图3
4-2	PLL_96M_SEL	rw	0	-	PLL_96M输入时钟源选择，见图3
1	RC_250K_EN	rw	1	-	片内RC（250k）振荡器使能（PMU低功耗模式无效） 0: 关闭片内RC振荡器 1: 打开片内RC振荡器

Bit	Name	RW	Default	RV	Description
0	RC_16M_EN	rw	1	-	片内RC（16M）振荡器使能（PMU低功耗模式无效） 0: 关闭片内RC振荡器 1: 打开片内RC振荡器

3.6.7 STD_CON1：standard clock register 1

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	-	-	预留
19-14	CLKOUT2_DIV	rw	1	-	CLK_OUT2输出时钟分频选择，见图9
13-10	CLKOUT2_SEL	rw	0	-	CLK_OUT2输出时钟源选择，见图9
8-5	CLKOUT1_SEL	rw	0	-	CLK_OUT1输出时钟选择，见图9
3-0	CLKOUT0_SEL	rw	0	-	CLK_OUT0输出时钟选择，见图9

3.6.8 PRP_CON0： peripheral clock config register 0

Bit	Name	RW	Default	RV	Description
31-22	RESERVED	-	-	-	预留
16-15	MBIST_CKEN	rw	0	-	mbist_clk选择 1：hsb_pll_clk 2：pat_clk
14-12	UART_CKSEL	rw	0	-	uart_clk来源选择，见图7
11-10	USB_CKSEL	rw	0	-	usb_clk来源选择，见图7
9-6	RESERVED	-	-	-	预留
5-2	RESERVED	-	-	-	预留
1	RESERVED	-	-	-	预留
0	AXI_EN	rw	1	-	axi_clk使能： 0：不使能 1：使能

3.6.9 PRP_CON1： peripheral clock config register 1

Bit	Name	RW	Default	RV	Description
31-0	RESERVED	-	0	-	预留

3.6.10 PRP_CON2： peripheral clock config register 2

Bit	Name	RW	Default	RV	Description
31-6	RESERVED	-	0	-	预留
8	WL_REG_RSTN	rw	0	-	wl_aud复位
5-4	WL_DAC_CKSEL	rw	0	-	wl_dac_clk来源选择，见图6
3-2	WL_ADC_CKSEL	rw	0	-	wl_adc_clk来源选择，见图6
1-0	WL_CKSEL	rw	0	-	wl_clk来源选择，见图6

3.6.11 SYSPLL_CON0：pll control register 0

Bit	Name	RW	Default	RV	Description
31-28	RESERVED	-	-	-	预留
27	PLL_CKEN	rw	1	-	PLL时钟输出使能 需要先关闭数字使能信号（PLL_CKEN为0）再关闭PLL（PLL_EN）
26	PLL_TEST_EN	rw	0	-	测试使能
25-24	PLL_TEST_SEL	rw	0	-	测试功能选择
23	PLL_LDO_BYPASS	rw	0	-	测试功能
22-21	RESERVED	-	-	-	预留
20-17	PLL_LDO12A_S	rw	0	0x8	PLL模拟控制位，选择LDO的输出电压
16-14	PLL_IVCOS	rw	0	0x3	PLL模拟控制位，选择CCO的偏置电流
13-11	PLL_LPFR2	rw	0	-	PLL模拟控制位，对应环带滤波器电阻 0：R2=15K 1：R2=18K (对应24M模式) 2：R2=24K (对应12M模式) 3：R2=27K 4：R2=84K (对应2M模式) 5：R2=96K 6：R2=381K 7：R2=500K (对应200k模式)
9-8	PLL_ICPS	rw	0	-	PLL模拟控制位，选择电荷泵的电流 0: 对应200k Hz模式 1：对应2M Hz模式 2：对应12M Hz和24M Hz模式

Bit	Name	RW	Default	RV	Description
7-6	PLL_PFDS	rw	0	0x1	PLL模拟控制位，选择PFD的延迟时间
5	RESERVED	-	-	-	预留
4-2	PLL_REF_SEL	rw	0	-	PLL输入参考时钟选择，见图5
1	PLL_RST	rw	0	-	PLL模块复位，需在PLL EN使能10uS之后才能释放 0: 复位 1: 释放
0	PLL_EN	rw	0	-	PLL模块使能 0: 关闭 1: 打开

3.6.12 SYSPLL_CON1： pll control register 1

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	-	-	预留
19	PLL_VCTRL_OE	rw	0	-	PLL测试信号
18	PLL_CKSYN_EN	rw	0	-	CKSYN信号使能
17	RESERVED	-	-	-	预留
16	PLL_CKOUT_D3P5_OE	rw	0	-	3.5分频时钟输出使能
15	PLL_CKOUT_D2P5_OE	rw	0	-	2.5分频时钟输出使能
14	PLL_CKOUT_D2P0_OE	rw	0	-	2分频时钟输出使能
13	PLL_CKOUT_D1P5_OE	rw	0	-	1.5分频时钟输出使能
12	PLL_CKOUT_D1P0_OE	rw	0	-	1分频时钟输出使能
11-10	RESERVED	-	-	-	预留
9-8	PLL_ REFDSEN	rw	0x0	-	PLL参考时钟分频使能（见图1） 00: PLL内参考时钟分频器X2； 01: PLL内参考时钟分频器关闭（DIV1）； 1*: PLL内参考时钟分频器打开（DIV2-129）
7	RESERVED	-	-	-	预留
6-0	PLL_REFDS	rw	0x0	-	PLL参考时钟分频值，n= PLL_REFDS + 2。 （见图1）

3.6.13 SYSPLL_NR： pll config register NR

Bit	Name	RW	Default	RV	Description
31-12	RESERVED	-	-	-	预留
11-0	PLL_NR	rw	0	-	PLL参考时钟反馈分频(相当倍频)，m= PLL_DS+2 。见图1

4 IO_Wakeup

4.1 概述

IO Wakeup是一个异步事件唤醒模块，它可以将处于standby/sleep状态下的系统唤醒，进入正常工作模式。当系统处于正常模式时，则Wakeup源作为中断源，Wakeup模块至多有32个唤醒事件来源（具体数目参照下面IO 唤醒源）；

4.2 IO唤醒源

事件0：PA0

事件1： PA1

事件2：PA2

事件3：PA3

事件4：PA4

事件5：PA5

事件6：PA6

事件7：PA7

事件8：PA8

事件9：PA9

事件10：PA10

事件11：PA11

事件12：FUSBDP

事件13：FUSBDM

事件14：GP_ICH0

事件15：GP_ICH1

事件16：GP_ICH2

事件17：GP_ICH3

4.3 数字模块控制寄存

4.3.1 WKUP_CON0： wakeup enable control register

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
31-0	WKUP_EN	rw	0x0	-	对应唤醒源使能， 0：关闭唤醒功能 1：打开唤醒功能

4.3.2 WKUP_CON1：wakeup edge control register

Bit	Name	RW	Default	RV	Description
31-0	WKUP_EDGE	rw	0x0	-	对应唤醒源边沿选择 0：上升沿唤醒 1：下降沿唤醒

4.3.3 WKUP_CON2：wakeup clear pending control register

Bit	Name	RW	Default	RV	Description
31-0	WKUP_CPND	w	0x0	-	对应唤醒源中断清除，写1清零

4.3.4 WKUP_CON3：wakeup pending control register

Bit	Name	RW	Default	RV	Description
31-0	WKUP_PND	r	0x0	-	对应唤醒源中断标记

5 IOMC

5.1 数字模块控制寄存器

IOMC（IO remapping control）控制寄存器主要用于控制IO除crossbar之外的一些拓展功能的配置，一般仅有连接固定 IO 的外设或其他特殊外设的mapping控制位于IOMC控制寄存器；

5.1.1 IOMC0：iomc control register 0

Bit	Name	RW	Default	RV	Description
31-11	RESERVED	-	-	-	预留
10-8	SPI_ICK_SEL	rw	0	-	SPI1输入时钟延迟级数选择
7	SPI_DUPLEX	rw	0	-	spi1从机输入时钟选择 0：spi1_clki 1：spi0_clko
6	SPI0_IOS	-	-	-	spi0固定IO选择 0：A组 1：B组
5-4	RESERVED	-	-	-	预留
3	PCAP_MUX_EDGE	rw	0	-	pcap_ich输入极性反转 0：不变 1：取反
2-0	RESERVED	-	-	-	预留

5.1.2 OCH_CON0：output channel control register0

Bit	Name	RW	Default	RV	Description
31-24	RESERVED	-	-	-	预留

Bit	Name	RW	Default	RV	Description
23-18	OCH3_SEL	rw	0	-	通用输出通道选择 0：tmr0_pwm 1：tmr1_pwm 2：tmr2_pwm 3：tmr3_pwm 6：uart1_rts 7：clkout0 8：clkout1 9：clkout2 8：gp_ich0 9：gp_ich1 10-14: for test 15：usb_dbg_out 16：p33_clk_dbg 17：p33_sig_dbg[0] 18：p33_sig_dbg[1] 19：lpctm_dbg_out clk_dbg20-27：for test 28：spi0_cso 29：spi1_cso
17-12	OCH2_SEL	rw	0	-	同OCH3_SEL
11-6	OCH1_SEL	rw	0	-	同OCH3_SEL
5-0	OCH0_SEL	rw	0	-	同OCH3_SEL

5.1.3 OCH_CON1：output channel control register1

Bit	Name	RW	Default	RV	Description
31-24	RESERVED	-	-	-	预留
23-18	OCH7_SEL	rw	0	-	同OCH3_SEL
17-12	OCH6_SEL	rw	0	-	同OCH3_SEL
11-6	OCH5_SEL	rw	0	-	同OCH3_SEL
5-0	OCH4_SEL	rw	0	-	同OCH3_SEL

5.1.4 ICH_CON0：input channel control register0

Bit	Name	RW	Default	RV	Description
31-28	ICH7_SEL (TMR3_CAP)	rw	0	-	功能输入通道选择 0-3：gp_ich0-gp_ich3 4：tmr2_pwm 5：tmr3_pwm

Bit	Name	RW	Default	RV	Description
27-24	ICH6_SEL (TMR2_CAP)	rw	0	-	同ICH7选项
23-20	ICH5_SEL (TMR1_CAP)	rw	0	-	同ICH7选项
19-16	ICH4_SEL (TMR0_CAP)	rw	0	-	同ICH7选项
15-12	ICH3_SEL (TMR3_CIN)	rw	0	-	同ICH7选项
11-8	ICH2_SEL (TMR2_CIN)	rw	0	-	同ICH7选项
7-4	ICH1_SEL (TMR1_CIN)	rw	0	-	同ICH7选项
3-0	ICH0_SEL (TMR0_CIN)	rw	0	-	同ICH7选项

5.1.5 ICH_CON1：input channel control register1

Bit	Name	RW	Default	RV	Description
31-28	ICH15_SEL (IRFLT_IN3)	rw	0	-	同ICH7选项
27-24	ICH14_SEL (IRFLT_IN2)	rw	0	-	同ICH7选项
23-20	ICH13_SEL (IRFLT_IN1)	rw	0	-	同ICH7选项
19-16	ICH12_SEL (IRFLT_IN0)	rw	0	-	同ICH7选项
15-12	ICH11_SEL (MCPWM1_FP)	rw	0	-	同ICH7选项
11-8	ICH10_SEL (MCPWM0_FP)	rw	0	-	同ICH7选项
7-4	ICH9_SEL (MCPWM1_CK)	rw	0	-	同ICH7选项
3-0	ICH8_SEL (MCPWM0_CK)	rw	0	-	同ICH7选项

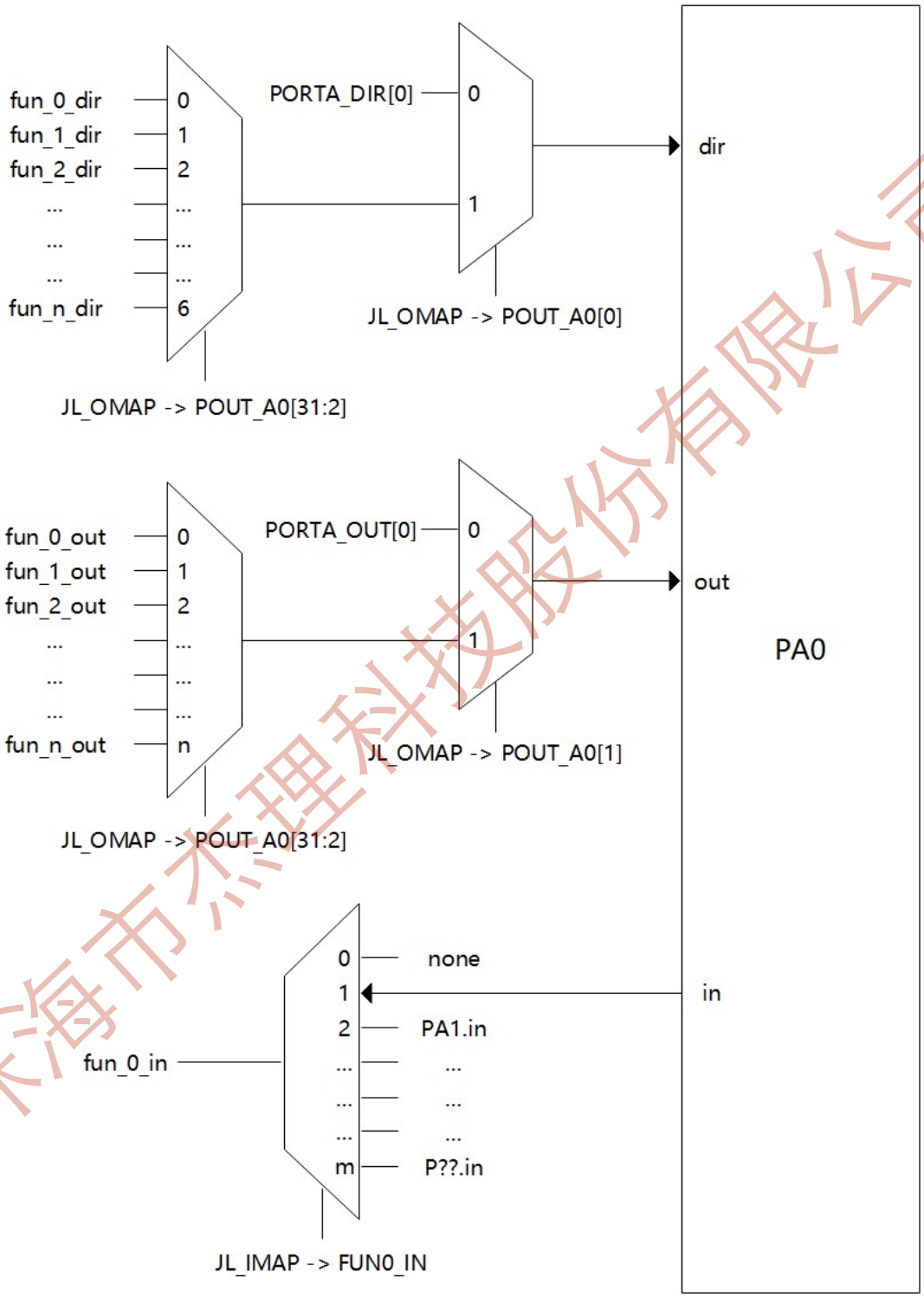
5.1.6 ICH_CON2：input channel control register2

Bit	Name	RW	Default	RV	Description
31-28	RESERVED	-	-	-	预留
27-24	ICH21_SEL (SPI1_CS_IN)	rw	0	-	同ICH7选项
23-20	ICH21_SEL (SPI0_CS_IN)	rw	0	-	同ICH7选项

Bit	Name	RW	Default	RV	Description
19-16	ICH20_SEL (EXTCLK_IN)	rw	0	-	同ICH7选项
15-12	ICH19_SEL (PCLK_IN)	rw	0	-	同ICH7选项
11-8	ICH18_SEL (PCAP_IN)	rw	0	-	同ICH7选项
7-4	ICH17_SEL (WLC_EXT_ACT_IN)	rw	0	-	同ICH7选项
3-0	ICH16_SEL (UART1_CTS)	rw	0	-	同ICH7选项

6 PORT Control Crossbar

6.1 Crossbar功能结构图



6.2 寄存器功能

6.2.1 JL_OMAP-> POUT

Bit	Name	RW	Description
[0]	FUN_DIR_EN	rw	Crossbar 方向控制使能： 0：CPU 控制 1：外设控制 （注：若所选外设无方向控制功能，则都由 CPU 控制）
[1]	FUN_OUT_EN	rw	Crossbar 数据控制使能： 0：CPU 数据 1：外设数据
[31:2]	FUN_SELECT	rw	Crossbar 输出功能选择： （见功能表或头文件）

6.2.2 JL_IMAP-> FUN_IN

Bit	Name	RW	Description
[31:0]	INPUT_SELECT	rw	Crossbar 输出引脚选择： （见功能表或头文件）

7 PORT_CONTROL

7.1 概述

该模块是一个IO多功能复用器，可以将芯片内部需要输出信号mapping到特定IO口上，也可以将特定IO的输入信号mapping到指定的内部模块。从而实现更灵活的IO可编程。

7.2 数字模块控制寄存器

7.2.1 PORTX_IN：portx input control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_IN	r	-	-	获取portx组的输入值，只有在对应的GPIO设置为输入使能（DIE）有效时数值才正确 0：输入为0 1：输入为1

7.2.2 PORTX_OUT：portx output control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_OUT	RW	0x0	-	控制portx组的输出值，只有在对应的GPIO设置为输出方向（DIR）有效时输出数值才正确 0：输出为0 1：输出为1

7.2.3 PORTX_DIR：portx direction control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_DIR	RW	0xff	-	控制portx组的输入输出方向 0：设置为输出方向 1：设置为输入方向

7.2.4 PORTX_DIE：portx input enable control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_DIE	RW	0xff	-	控制portx组的输入使能 0：禁止输入使能 1：允许输入使能

7.2.5 PORTX_DIEH：portx input enable p33 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_DIEH	RW	0xff	-	控制portx组的P33输入使能 0：禁止输入使能 1：允许输入使能

7.2.6 PORTX_PU0：portx pull up0 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_PU0	RW	0x0	-	控制portx组的上拉电阻值，与PORTX_PU1组成4个档位

7.2.7 PORTX_PU1：portx pull up1 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_PU0	RW	0x0	-	控制portx组的上拉电阻值，与PORTX_PU0组成4个档位{pu1,pu0} 00：无上拉电阻 01：10KΩ 10：100KΩ 11：1MΩ（mclr上拉电阻不受该位控制，由p33控制）

注意：PORTUSB不存在PU1寄存器，写入无效，读取为不定值。IO上拉电阻的值固定，具体参考下文USB_IO CONTROL的内容

7.2.8 PORTX_PD0：portx pull down0 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_PD0	RW	0x0	-	控制portx组的下拉电阻值，与PORTX_PD1组成4个档位

7.2.9 PORTX_PD1：portx pull down1 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留

Bit	Name	RW	Default	RV	Description
15-0	PORTX_PD1	RW	0x0	-	控制portx组的下拉电阻值，与PORTX_PD0组成4个档位{pd1,pd0} 00：无下拉电阻 01：10KΩ 10：100KΩ 11：1MΩ

注意：PORTUSB不存在PD1寄存器，写入无效，读取为不定值。IO上拉电阻的值固定，具体参考下文USB_IO CONTROL的内容

7.2.10 PORTX_HD0：portx high driver0 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_HD0	RW	0x0	-	控制portx组的强驱动电流档位，与PORTX_HD1组成4个档位

注意：PORTUSB不存在HD0寄存器，写入无效，读取为不定值。

7.2.11 PORTX_HD1：portx high driver1 control register：

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	PORTX_HD1	RW	0x0	-	控制portx组的强驱动电流档位，与PORTX_HD0组成4个档位{hd1,hd0}

注意：PORTUSB不存在HD1寄存器，写入无效，读取为不定值

7.2.12 PORTX_SPL：portx pull-down resistor enable in output control register:

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0x0	-	预留
15-0	SPL	RW	0x0	-	置1使能，当使能时： 在IO为输出状态时，PORTX_IN寄存器持续读取从PAD上的电压值，即输出回读功能 在IO为输出状态时保持上下拉电阻功能，

7.2.13 PORTX_BSR：portx bit set/clear resistor enable in output control register:

Bit	Name	RW	Default	RV	Description
31-16	CLR	W	0x0	-	设置portx组的输出0，只有在对应的GPIO设置为输出方向（DIR）有效时输出数值才正确 0：无效 1：有效

Bit	Name	RW	Default	RV	Description
15-0	SET	W	0x0	-	设置portx组的输出1，只有在对应的GPIO设置为输出方向（DIR）有效时输出数值才正确 0：无效 1：有效

7.2.14 POUT：port output control register

Bit	Name	RW	Default	RV	Description
31-2	FUN_SEL	rw	0x0	-	Crossbar输出功能选择： 0：GP_OCH0 1：GP_OCH1 2：GP_OCH2 3：GP_OCH3 4：GP_OCH4 5：GP_OCH5 6：GP_OCH6 7：GP_OCH7 8：FO_LEDC_DOUT0 9：FO_LEDC_DOUT1 10：FO_I2C_SCL 11：FO_I2C_SDA 12：FO_UART0_TX 13：FO_UART1_TX 14：FO_UART2_TX 15：FO_MCPWM_H0 16：FO_MCPWM_L0 17：FO_MCPWM_H1 18：FO_MCPWM_L1 19：FO_SPI1_CLK 20：FO_SPI1_DA0 21：FO_SPI1_DA1 22：FO_SPI1_DA2 23：FO_SPI1_DA3
DAT	FUN_OUT_EN	rw	0	-	Crossbar数据控制使能 0：CPU数据 1：外设数据
0	FUN_DIR_EN	rw	0	-	Crossbar方向控制使能 0：CPU控制 1：外设控制 【注意】：若所选外设无方向控制功能，则由CPU控制

7.2.15 FUN_IN：port input control register

Bit	Name	RW	Default	RV	Description
31-0	INPUT_SEL	rw	0x0	-	Crossbar输入引脚选择 1：PA0 2：PA1 3：PA2 4：PA3 5：PA4 6：PA5 7：PA6 8：PA7 9：PA8 10：PA9 11：PA10 12：PA11 13：PA12 14：USBDP 15：USBDM

USB_IO CONTROL

1. JL_PORTUSB->DIE：
2. JL_PORTUSB->DIEH：
3. JL_PORTUSB->DIR：
4. JL_PORTUSB->PU0：DP上拉1.5K，DM上拉180K(PU1没有用到)
5. JL_PORTUSB->PD0：DP 下拉15K，DM下拉15K(PD1没有用到)
6. JL_PORTUSB->IN：
7. JL_PORTUSB->OUT：
8. JL_PORTUSB->SPL：

上述USB IO寄存器跟普通IO寄存器功能一样，不过USB 只有两个IO,BIT(0)设置DP，BIT(1)设置DM

7.2.16 JL_PORTUSB->CON： usb io control register 0

Bit	Name	RW	Default	RV	Description
31-13	RESERVED	-	-	-	预留
12-11	DBG_SEL	rw	0x0	-	测试信号选择： 00：1'b0 01：usb_diff 10：usb_dp 11：usb_dm
10-9	RESERVED	-	-	-	预留
8	IO_MODE	rw	0x1	-	IO模式使能 0：USB模式 1：普通IO模式
7	CHKDPO	r	0x1	-	CHKDPO经系统时钟采样后的输入
6	DIDF	r	0	-	差分输入DIDF经lsb_clk采样后的输入
5	RESERVED	-	-	-	预留
4	PDCHKDP	rw	0	-	DP 外接下拉检查使能 0：disable 1：enable
3	RESERVED	-	-	-	预留
2	SR0	rw	0x0	-	输出驱动能力选择
1	RESERVED	-	-	-	预留
0	RCVEN	rw	0	-	USB_PHY使能(差分输入使能，使用USB功能时需打开这一BIT)

7.3 IO模拟状态设置

当IO用于模拟功能时，需要正确设置IO寄存器，确保模拟功能不受IO影响。正确配置如下：

如将PA0设置为模拟功能。

```
JL_PORTA->DIR |= BIT(0);
JL_PORTA->DIE &= ~BIT(0);
JL_PORTA->DIEH &= ~BIT(0);
JL_PORTA->PU0 &= ~BIT(0);
JL_PORTA->PU1 &= ~BIT(0);
JL_PORTA->PD0 &= ~BIT(0);
JL_PORTA->PD1 &= ~BIT(0);
```

8 ADC

8.1 概述

10Bit ADC(A/D转换器)，其时钟最大不可超过1MHz。

8.2 寄存器说明

8.2.1 ADC_CON: ADC control register

Bit	Name	RW	Default	RV	Description
31	DONE_PND	r	0	-	只读，采样完成pnd；kst后完成一次该位置1
30	DONE_PND_CLR	w	-	-	只写，写‘1’清除DONE_PND位，写‘0’无效
29	DONE_PND_IE	rw	0	-	DONE_PND中断允许
28-24	RESERVED	-	-	-	预留
23-21	ADC_MUX_SEL	rw	0x0	0	ADC模式选择： 001：ADC采集IO通道电压 010：ADC采集内部模拟通道电压 111：将选通的内部模拟电压通过IO口输出
20-18	ADC_ASEL	rw	0x0	-	内部模拟通道选择： 000: BT_TEST 001: PMU_TEST 010: PLL_TEST 011: LRC32K_TEST
17	ADC_CLKEN	rw	0	0	ADC 时钟使能
16	ADCISEL	rw	0		ADC CMP power select
15-12	WAIT_TIME	rw	x	-	启动延时控制，实际启动延时为此数值乘8个ADC时钟

Bit	Name	RW	Default	RV	Description
11-8	CH_SEL	rw	x	-	IO通道选择： 0000：选择PA0 0001：选择PA1 0010：选择PA2 0011：选择PA3 0100：选择PA4 0101：选择PA5 0110：选择PA6 0111：选择PF2 1000：选择PA8 1001：选择PA9 1010：选择PA10 1011：选择PA11 1100：选择FSPG 1101：选择USBDP 1110：选择USBDM
7	PND	r	1	-	PND：只读，中断请求位，当ADC完成一次转换后，此位会被设置为‘1’，需由软件清‘0’
6	CPND	w	x	-	CPND：只写，写‘1’清除中断请求位，写‘0’无效
5	ADC_IE	rw	0	-	ADC_IE：ADC中断允许
4	ADC_EN	rw	0	-	ADC_EN： ADC控制器使能
3	ADC_AE	rw	0	-	ADC_AE： ADC 模拟模块常使能
2-0	ADC_BAUD	rw	0x0	-	ADC_BAUD： ADC时钟频率选择 000：LSB时钟1分频 001：LSB时钟6分频 010：LSB时钟12分频 011：LSB时钟24分频 100：LSB时钟48分频 101：LSB时钟72分频 110：LSB时钟96分频 111：LSB时钟128分频

8.2.2 ADC_RES： ADC result register

Bit	Name	RW	Default	RV	Description
31-10	RESERVED	-	-	-	预留
9-0	RES	r	x	-	ADC结果

有关模拟通道配置，请参考相关文档：

	SFR	文档

	SFR	文档
BT		
PMU	P33_ANA_CON4	pmu3333_analog.doc
AUDIO	DAA_CON0	Audio_Onchip.doc
X32K	P3_OSL_CON	

9 ADVANCED ENCRYPTION STANDARD(aes)

9.1 概述

高级加密标准AES128为最常见的对称加密算法。该模块具有CCM功能。

9.2 特殊注意点

9.3 1.3控制寄存器

9.3.1 AES_CON: AES control register

Bit	Name	RW	Default	RV	Description
31	CCM_PND	r			CCM中断标志位 0：CCM模块IDLE或者正在运算 1：CCM模块结束运算
30	CLR_CCM_PND	w			写1清空CCM_PND
29-19	-	-			预留
18	CCM_DEC_KST	w			CCM解密启动 写1：启动 写0：N/A
17	-	-			预留
16	CCM_ENC_KST	-			CCM加密启动 写1：启动 写0：N/A
15-12	-	-			预留
11	HEAD_LEN	rw			HEADER的长度 0：长度为0 1：长度为1
10	KEY_IE	rw			KEY完成中断允许
9	CCM_IE	rw			CCM中断允许
8	ENC_IE	rw			ENC中断允许
7	AES_PND	r			0: IDLE 1: AES模块产生中断
6	CLR_PND	w			写1清空AES_PND

Bit	Name	RW	Default	RV	Description
5	KEY_STATE	r			0:表示正在做密钥扩展运算； 1:密钥已经准备好可以启动加解密
4	KEY_STAR	w			写1： 把AES_KEY输入的128bit数据输入到密钥模块， 并启动密钥扩展运算，KEY_STATE 由1变0； 写0：无效
3	-	-			预留
2	ENC_STATE	r			1:表示正在做加密运算； 0：加密模块空闲
1-0	IS_ENC	w			01:ENC other:Reserved

9.3.2 AES_DATIN : 加解密数据输入

Bit	Name	RW	Default	RV	Description
31-0	AES_DATIN	rw	0x0	-	加解密数据输入，先输入高word 再输入低word

9.3.3 AES_KEY: 加解密KEY输入

Bit	Name	RW	Default	RV	Description
31-0	AES_KEY	rw	0x0	-	加解密原始密钥输入，先输入高word 再输入低word

9.3.4 AES_ENCRES0: 加密结果寄存器0

Bit	Name	RW	Default	RV	Description
31-0	AES_ENCRES	r	0x0	-	加密结果，ENC_STATE为0时有效

9.3.5 AES_ENCRES1: 加密结果寄存器1

Bit	Name	RW	Default	RV	Description
31-0	AES_ENCRES1	r	0x0	-	加密结果，ENC_STATE为0时有效

9.3.6 AES_ENCRES2 加密结果寄存器2

Bit	Name	RW	Default	RV	Description
31-0	AES_ENCRES2	r	0x0	-	加密结果，ENC_STATE为0时有效

9.3.7 AES_ENCRES3: 加密结果寄存器3

Bit	Name	RW	Default	RV	Description
31-0	AES_ENCRES3	r	0x0	-	加密结果，ENC_STATE为0时有效

9.3.8 AES_ NONCE: NONCE寄存器（16个byte）

Bit	Name	RW	Default	RV	Description
31-0	AES_NONCE	r	0x0	-	设置CCM的NONCE值，共16byte， 先写最高4byte，写4次完成初始化

9.3.9 AES_ HEADER: HEADER寄存器

Bit	Name	RW	Default	RV	Description
31-8					预留
7-0	AES_HEADER	W			设置CCM的header值

9.3.10 AES_SRCADR:数据起始地址

Bit	Name	RW	Default	RV	Description
31-0	AES_SRCADR	Rw	0x0	-	设置AES取数据运算的起始地址

9.3.11 AES_DSTADR:目标起始地址

Bit	Name	RW	Default	RV	Description
31-0	AES_DSTADR	Rw	0x0	-	设置AES目标的起始地址

9.3.12 AES_CTCNT

Bit	Name	RW	Default	RV	Description
31-8					
7-0	AES_CTCNT				设置AES从MEMORY里拿取多少byte数据来运算，推荐值是27和251

9.3.13 AES_TAGLEN

Bit	Name	RW	Default	RV	Description
31-0	AES_TAGLEN	Rw	0x0	-	设置TAG的长度，范围从4~16，只能是2的倍数

TAGRES0 TAG结果的0~3个字节

TAGRES1 TAG结果的4~7个字节

TAGRES2 TAG结果的8~11个字节

TAGRES3 TAG结果的12~15个字节

9.3.14 AES_TAGRES0

Bit	Name	RW	Default	RV	Description
31-0	AES_TAGRES0	Rw	0x0	-	TAG结果的0~3个字节

9.3.15 AES_TAGRES1

Bit	Name	RW	Default	RV	Description
31-0	AES_TAGRES1	Rw	0x0	-	TAG结果的4~7个字节

9.3.16 AES_TAGRES2

Bit	Name	RW	Default	RV	Description
31-0	AES_TAGRES2	Rw	0x0	-	TAG结果的8~11个字节

9.3.17 AES_TAGRES3

Bit	Name	RW	Default	RV	Description
31-0	AES_TAGRES3	rw	0x0	-	TAG结果的12~15个字节

9.4 1.4代码示例

假设使用的参数为：

加解密使用相同KEY：0X4c683841_39f574d8_36bcf34e_9dfb01bf

加密原始数据：0X02132435_46576879_acbdcedf_e0f10213

解密原始数据：0X99ad1b52_26a37e3e_058e3b8e_27c2c666

```
AES_KEY = 0X4c683841;
AES_KEY = 0X39f574d8;
AES_KEY = 0x36bcf34e;
AES_KEY = 0x9dfb01bf;
AES_CON |= (1<<4);          // set key
while( AES_CON & (1<<5) );      // wait key set ok
AES_DAT3 = 0X02132435;
AES_DAT2 = 0X46576879;
AES_DAT1 = 0Xacbdcedf;
AES_DAT0 = 0Xe0f10213;
AES_CON |= (1<<0)   ;      //START ENC
while( AES_CON & (1<<2) );     //wait enc
```

加密结果：{AES_ENCRES3, AES_ENCRES2, AES_ENCRES1, AES_ENCRES0}

= 0X99ad1b52_26a37e3e_058e3b8e_27c2c666

CCM使用方法：

假设使用的参数为：

加解密使用相同KEY： 0x99ad1b5226a37e3e58e3b8e27c2c666;

加密原始数据：0X02132435_46576879_acbdcedf_e0f10213

解密原始数据：0X99ad1b52_26a37e3e_058e3b8e_27c2c666

```
JL_AES->KEY = 0x99ad1b52;
JL_AES->KEY = 0x26a37e3e;
JL_AES->KEY = 0x58e3b8e;
JL_AES->KEY = 0x27c2c666;
JL_AES->CON  = BIT(4);
while((JL_AES->CON & 0x20) == 0x20);
   unsigned int *p = (unsigned int *)0x1f04000;
    ptr[i++] = 0x17;
    ptr[i++] = 0x00;
    ptr[i++] = 0x37;
    ptr[i++] = 0x36;
    ptr[i++] = 0x35;
    ptr[i++] = 0x34;
    ptr[i++] = 0x33;
    ptr[i++] = 0x32;
    ptr[i++] = 0x31;
    ptr[i++] = 0x30;
    ptr[i++] = 0x41;
    ptr[i++] = 0x42;
    ptr[i++] = 0x43;
    ptr[i++] = 0x44;
    ptr[i++] = 0x45;
    ptr[i++] = 0x46;
    ptr[i++] = 0x47;
    ptr[i++] = 0x48;
    ptr[i++] = 0x49;
    ptr[i++] = 0x4a;
    ptr[i++] = 0x4b;
    ptr[i++] = 0x4c;
    ptr[i++] = 0x4d;
    ptr[i++] = 0x4e;
    ptr[i++] = 0x4f;
    ptr[i++] = 0x50;
    ptr[i++] = 0x51;
    ptr[i++] = 0x00;
    ptr[i++] = 0x00;
    ptr[i++] = 0x00;
    ptr[i++] = 0x00;
    ptr[i++] = 0x00;
  }
  JL_AES->TAGLEN = 4;
  JL_AES->NONCE = 0xde;
  JL_AES->NONCE = 0xafbabeba;
```

```
JL_AES->NONCE = 0xdcab2400;
JL_AES->NONCE = 0x00000001;
JL_AES->HEADER = 0x2;
JL_AES->SRCADR = (unsigned int)ptr;
JL_AES->DSTADR = (unsigned int)dptr;
JL_AES->CON |= BIT(16);
while((JL_AES->CON & BIT(31)) == 0);
JL_AES->CON |= BIT(30);
if(JL_AES->TAGRES0 != 0x89b96088)   //tag在这里输出
  return -1;
JL_AES->SRCADR = (unsigned int)dptr;
JL_AES->DSTADR = (unsigned int)ptr;
JL_AES->CON |= BIT(18);
while((JL_AES->CON & BIT(31)) == 0);
JL_AES->CON |= BIT(30);
if(JL_AES->TAGRES0 != 0x89b96088)   //tag在这里输出
  return -1;
```

加密结果：=0xf38881e7_bd94c9c3_69b9a668_46dd4786_aa8c39ce_540d0dae_3adcdf

解密结果：=0x17003736_35343332_31304142_43444546_4748494a_4b4c4d4e_4f5051

10 DMA_GEN请求发生器

10.1 概述

DMA_GEN请求发生器会在其DMA请求触发输入上出现触发事件后产生 DMA 请求。

DMA_GEN请求发生器在硬件上可配通道数量。DMA请求触发输入并行连接到所有通道。

DMA_GEN请求发生器预留了64个触发源，通道x的 DMA请求触发输入通过寄存器GEN_SEL0和GEN_SEL1选择。

DMA请求触发输入上的触发事件可以是上升沿、下降沿或任一边沿。有效边沿通过GEN_POL字段选择。

触发事件发生后，相应通道开始在其输出上生成 DMA请求。每完成一次DMA 请求（即得到DMA_ACK时），DMA_GEN请求发生器内置的DMA 请求计数器减 1，同时产生下一次DMA请求，直达达到预设的请求数量。

在DMA_GEN请求发生器生成 DMA请求过程中，若有新的请求触发，则触发溢出标志OV_PND会被置1，当对应的中断使能OV_IE为1的前提下，将产生中断请求。当发生触发溢出事件时，说明触发频率过高，DMA_GEN请求发生器来不及响应，使用人员应该通过软件降低触发频率。

10.2 数字模块控制寄存器

10.2.1 JL_DMAGEN->JL_DMAGEN_CHx.CON0: dma gererator channel x config register

Bit	Name	RW	Default	RV	Description
31	OV_PND	r	0	-	触发溢出异常标志
30	OV_CLR	w	0	-	触发溢出异常清零
29	OV_IE	rw	0	-	触发溢出异常使能
28-8	RESERVED	-	-	-	预留
7-3	REQ_NUM	rw	0x0	-	一次触发产生的DMA请求数目。 实际上产生的DMA请求数目为REQ_NUM+1；
2-1	GEN_POL	rw	0x0	-	触发极性选择： 00：无触发检测和生成 01：上升沿 10：下降沿 11：上升沿或下降沿
0	GEN_EN	rw	0	-	通道使能

10.2.2 JL_DMAGEN->JL_DMAGEN_CHx.SEL0: dma gererator channel x source select register0

Bit	Name	RW	Default	RV	Description
31-0	GEN_SEL0	rw	0	-	GEN_SEL0[31:0]的每一位分别对应选择第31-0号触发源

10.2.3 JL_DMAGEN->JL_DMAGEN_CHx.SEL1: dma gererator channel x source select register1

Bit	Name	RW	Default	RV	Description
31-0	GEN_SEL1	rw	0	-	GEN_SEL1[31:0]的每一位分别对应选择第63-32号触发源

10.3 DEMO CODE

以下例子为DMA_GEN请求发生器联动UDMA进行DMA传输的例子。

```
void dma_ch0_cfg()
{
    JL_UDMA->JL_UDMA_CH0.SADR = 0x3f00000;  // 设置源地址为0x300011
    JL_UDMA->JL_UDMA_CH0.DADR = 0x3f01002;  // 设置目标地址为0x301002
    SFR(JL_UDMA->JL_UDMA_CH0.CON1, 16, 16, 0);// 设置循环次数为0次
    SFR(JL_UDMA->JL_UDMA_CH0.CON1, 0, 16, 16); // 设置DMA的传输数据量为16byte
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 1, 3, 0);  // 选择0号外设源
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 4, 2, 3);  // 设置通道优先级为最高
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 6, 1, 0);  // 设置DMA传输为外设传输模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 7, 1, 0);  // 设置DMA通道为单次模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 8, 1, 1);  // 设置源地址为指针递增模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 9, 1, 1);  // 设置目标地址为指针递增模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 10, 2, 2); // 设置源内存数据位宽为32
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 12, 2, 2); // 目标内存数据位宽为32
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 15, 2, 0); // 数据扩展模式为位封装模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 25, 1, 1); // 使能传输完成中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 28, 1, 0); // 禁用传输一半中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 31, 1, 1); // 使能传输错误中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 0, 1, 1);  // 打开DMA通道使能
}
void mian()
{
    dma_ch0_cfg();
    JL_DMAGEN->JL_DMAGEN_CH0.SEL1 = 0x0;
    JL_DMAGEN->JL_DMAGEN_CH0.SEL0 = BIT(1);      // 选择触发源1
    SFR(JL_DMAGEN->JL_DMAGEN_CH0.CON0, 1, 2, 1); // 选择上升沿触发
    SFR(JL_DMAGEN->JL_DMAGEN_CH0.CON0, 3,  5, 3); // DMA请求个数为3
    SFR(JL_DMAGEN->JL_DMAGEN_CH0.CON0, 29, 1, 1); // 开启触发溢出异常使能
    SFR(JL_DMAGEN->JL_DMAGEN_CH0.CON0, 0, 1, 1); // 使能DMA请求发生器通道0
    delay(10);
}
```

10.4 附录 DMA_GEN触发源映射

触发源序号	选择控制寄存器	触发源
0	GEN_SEL0[0]	TRIG_IIC0_ TASK （IIC触发源对应IIC所有PND置位条件）
1	GEN_SEL0[1]	TRIG_IIC0_ RESTART
2	GEN_SEL0[2]	TRIG_IIC0_ STOP
3	GEN_SEL0[3]	TRIG_IIC0_ RXACK
4	GEN_SEL0[4]	TRIG_IIC0_ RXNACK
5	GEN_SEL0[5]	TRIG_IIC0_ ADR_MATCH
6	GEN_SEL0[6]	TRIG_IIC0_ RXBYTE_DONE
7	GEN_SEL0[7]	TRIG_IIC0_ TXTASK_LOAD
8	GEN_SEL0[8]	TRIG_IIC0_ RXTASK_LOAD
9	GEN_SEL0[9]	-
10	GEN_SEL0[10]	TMR_PR_MATCH[0]
11	GEN_SEL0[11]	TMR_PR_MATCH[1]
12	GEN_SEL0[12]	TMR_PR_MATCH[2]
13	GEN_SEL0[13]	TMR_PR_MATCH[3]
14	GEN_SEL0[14]	TMR_CPT_ACTIVE[1]
15	GEN_SEL0[15]	TMR_CPT_ACTIVE[2]
16	GEN_SEL0[16]	TMR_CPT_ACTIVE[3]
17	GEN_SEL0[17]	TMR_CPT_ACTIVE[4]
18	GEN_SEL0[18]	TMR_PWM_OUT[0]
19	GEN_SEL0[19]	TMR_PWM_OUT[1]
20	GEN_SEL0[20]	TMR_PWM_OUT[2]
21	GEN_SEL0[21]	TMR_PWM_OUT[3]
22	GEN_SEL0[22]	MC_PWM_H[0]
23	GEN_SEL0[23]	MC_PWM_H[1]
24	GEN_SEL0[24]	MC_PWM_H[2]
25	GEN_SEL0[25]	MC_PWM_L[0]
26	GEN_SEL0[26]	MC_PWM_L[1]

触发源序号	选择控制寄存器	触发源
27	GEN_SEL0[27]	MC_PWM_L[2]
28	GEN_SEL0[28]	MC_TMR_OVF[0]
29	GEN_SEL0[29]	MC_TMR_OVF[1]
30	GEN_SEL0[30]	MC_TMR_OVF[2]
31	GEN_SEL0[31]	MC_TMR_ZERO[0]
32	GEN_SEL1[0]	MC_TMR_ZERO[1]
33	GEN_SEL1[1]	MC_TMR_ZERO[2]
34	GEN_SEL1[2]	MC_FPIN_EDGE[0]
35	GEN_SEL1[3]	MC_FPIN_EDGE[1]
36	GEN_SEL1[4]	MC_FPIN_EDGE[2]
37	GEN_SEL1[5]	ADC_TRI
38	GEN_SEL1[6]	LPCTM0_DMA_REQ
39	GEN_SEL1[7]	
40	GEN_SEL1[8]	
41	GEN_SEL1[9]	
42	GEN_SEL1[10]	
43	GEN_SEL1[11]	
44	GEN_SEL1[12]	
45	GEN_SEL1[13]	
46	GEN_SEL1[14]	
47	GEN_SEL1[15]	
48	GEN_SEL1[16]	
49	GEN_SEL1[17]	
50	GEN_SEL1[18]	
51	GEN_SEL1[19]	
52	GEN_SEL1[20]	

触发源序号	选择控制寄存器	触发源
53	GEN_SEL1[21]	
54	GEN_SEL1[22]	
55	GEN_SEL1[23]	
56	GEN_SEL1[24]	
57	GEN_SEL1[25]	
58	GEN_SEL1[26]	
59	GEN_SEL1[27]	
60	GEN_SEL1[28]	
61	GEN_SEL1[29]	
62	GEN_SEL1[30]	
63	GEN_SEL1[31]	

11 GPCNT

11.1 概述

GPCNT为时钟脉冲计数器，用于计算两个时钟周期的比例，即用一个已知时钟（主时钟）计算另一个时钟（次时钟）的周期。

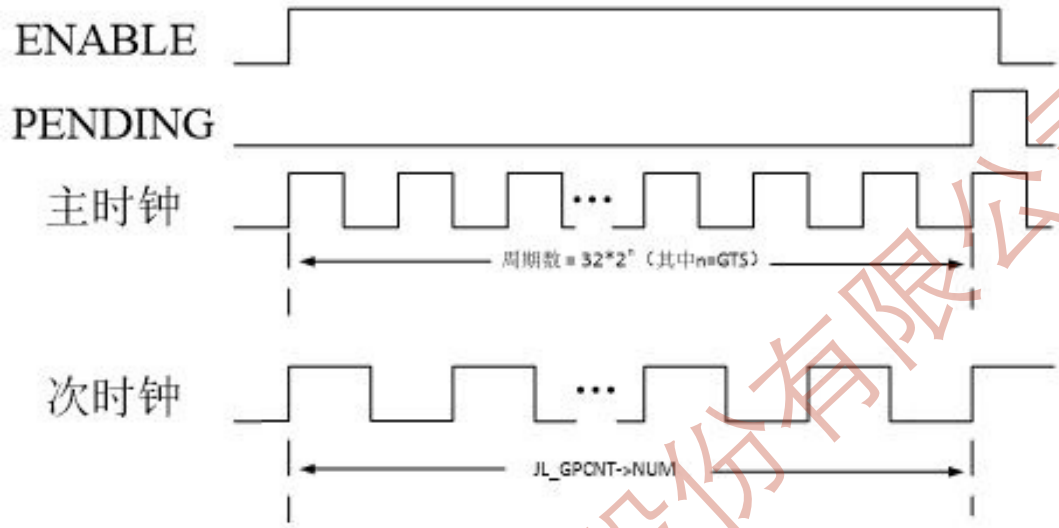


图 1-1 GPCNT时钟计算示意图

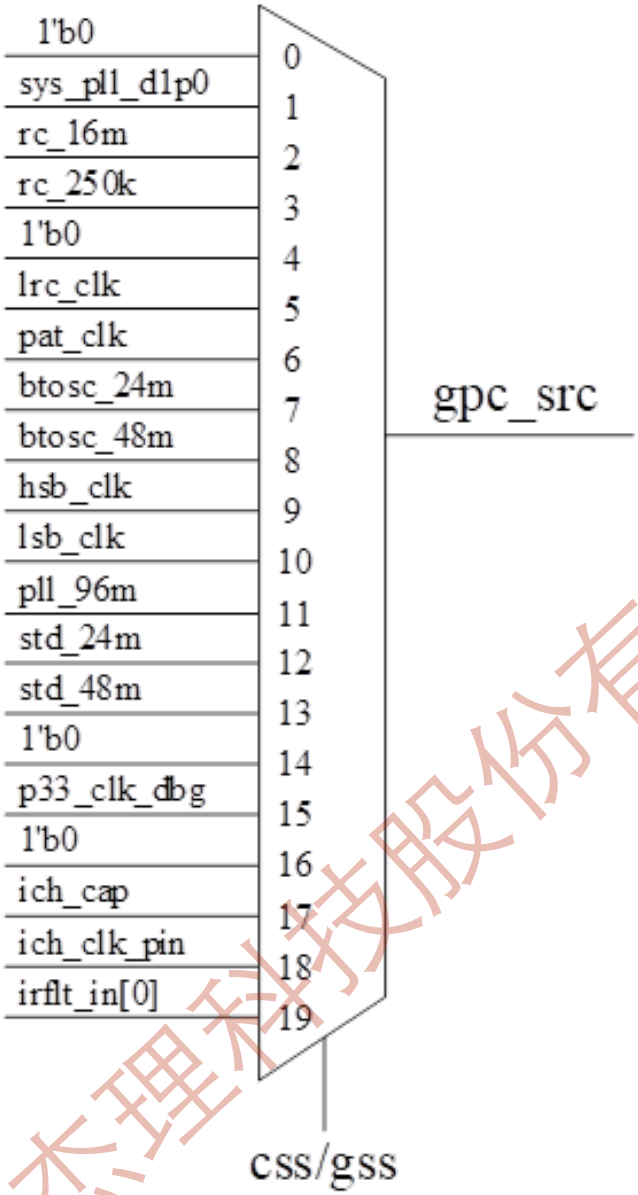


图 1-2 GPCNT时钟源选择示意图

11.2 寄存器说明

11.2.1 JL_GPCNT->CON: GPCNT configuration register

Bit	Name	RW	Default	RV	Description
31	PND	r	0	-	中断请求标志（当JL_GPCNT->CON[0]置1后， 主时钟达到JL_GPCNT->CON[11:8]设定的周期数后， PENDING置1，并请求中断）： 0: 无PENDING; 1: 有PENDING

Bit	Name	RW	Default	RV	Description
30	CLR_PND	w	x	-	清除中断请求标志位： 0: 无效； 1: 清PENDING
29-20	RESERVED	-	-	-	预留
19-16	GTS	rw	0x0	-	主时钟周期数选择： 主时钟周期数 = 32*2^n （其中n=GTS）
15-13	RESERVED	-	-	-	预留
12-8	GSS	rw	0x0	-	主时钟选择（计数时钟）：见图1-2
7-6	RESERVED	-	-	-	预留
5-1	CSS	rw	0x0	-	次时钟选择（被计数时钟）：见图1-2
0	ENABLE	rw	0	-	GPCNT模块使能位： 0: 不使能； 1: 使能

【注意】：需配置好其他位，才将ENABLE置1。

11.2.2 JL_GPCNT->NUM: The number of GPCNT clock cycle register

Bit	Name	RW	Default	RV	Description
31-0	NUM	r	0x0	-	在JL_GPCNT->CON[0]置1到 PENDING置1（中断到来）之间，次时钟跑的周期数。

12 IIC

12.1 概述

IIC接口是一个可兼容大部分IIC总线标准的串行通讯接口。在上面传输的数据以Byte（8bit）为最小单位，且永远是MSB在前。

1. IIC接口主要支持特性：

1. 主模式和从模式，不支持多机功能；
2. 支持标准速度模式（Standard-mode，Sm，高达100kHz），快速模式（Fast-mode，Fm，高达400kHz），快速增强模式（Fast-mode Plus，Fm+，高达1MHz）；
3. 7bit主从机寻址模式，可配置从机地址应答，广播地址响应；
4. 总线时钟延展功能可配置；
5. 总线数据建立时间和保持时间可配置；
6. 基于便捷的事务操作驱动总线；
7. 数据接收发送各独立具有1字节缓冲；
8. 可配置总线信号数字滤波器；

2. 主机：

1. IIC接口SCL时钟由本机产生，提供给片外IIC设备使用；
2. IIC接口SCL时钟可配置，SCL时钟周期为：

TIIC_BUS_SCL = TIIC_clk /(2 * BUAD_CNT) + T电阻上拉

3. 从机：

1. IIC接口SCL时钟由片外 IIC设备产生，经内部IIC_clk滤波采样后给本机使用；
2. IIC总线上每一个start/restart位后接从机地址，此时当外部IIC设备所发的地址与我们匹配时（开启广播地址响应，在接收到广播地址时也会响应），可配置硬件自动回应（ack位为“0”）；

12.2 特殊注意点

时钟要求：作为主机时，BAUD、TSU 、THD、FLT_SEL值设置应当满足以下关系

BAUD_CNT > (SETUP_CNT + HOLD_CNT + FLT_SEL+5)

作为从机时，SCL 时钟低电平时间与TSU 、THD、FLT_SEL值设置应当满足以下关系

TSCL_L > (SETUP_CNT + HOLD_CNT + FLT_SEL + 5) * TI2C_clk

IO 设置：在选择IO作为IIC接口的SCL、SDA输入输出时，必须配置该两个IO的SPL寄存器位为 1；

事务寄存器：写入事务后需等待硬件将写入的事务加载后（tx_task_load/rx_task_load 置位即表示对应事务被硬件加载，task_done pnd置位即表示事务已被硬件加载且执行完成）才可写入新的事务；

【注意】：IIC的通信频率不仅与所配置的波特率有关，还会受到IIC总线上拉时间的影响，请尽可能符合IIC总线硬件电气要求。

12.3 数字模块控制寄存器

12.3.1 IIC_CON0: iic control register 0

Bit	Name	RW	Default	RV	Description
31-10	RESERVED	-	-	-	预留
10	BADDR_RESP_EN	rw	0	-	广播地址响应使能： 1：响应总线上的广播地址； 0：不响应总线上的广播地址
9	AUTO_SLV_TX	rw	0	-	从机自动发送数据事务使能（主机模式下无效）： 0：不使能； 1：使能 从机接收匹配的地址且为从机接收数据时，接下来硬件会自动填充SEND_DATA事务操作，使得硬件能不间断的发送数据；（作为从机与不支持时钟延展的主机通信时必须使能，其他工作场景依据需求使能）
8	AUTO_SLV_RX	rw	0	-	从机自动接收数据事务使能（主机模式下无效）： 0：不使能； 1：使能 从机接收匹配的地址且为从机接收数据时，接下来 硬件会自动填充RECV_DATA事务操作，使得在软件读取数据时，硬件接口能继续不间断接收数据；（作为从机与不支持时钟延展的主机通信时必须 使能，其他工作场景依据需求使能）
7	AUTO_ADR_RESP	rw	0	-	从机接收地址硬件自动响应使能： 作为从机时，在接收到地址数据后（接收的第一字节数据） 0：不自动进行响应，即不对接收到的地址数据进行NACK/ACK，需cpu读取接收数据后，再执行NACK/ACK事务； 1：自动进行响应，接收到匹配的地址（含广播地址） 自动响应ACK；接收到不匹配的地址，自动响应NACK （从机与不支持时钟延展的主机通信时必须使能，其他工作场景依据需求使能）
6	IGNORE_NACK	rw	0	-	NACK调试模式使能 0：在发送数据中遇到NACK后会停止接收发送； 1：在发送数据中遇到NACK后不会中断接收发送事务(仅用于调试)；

Bit	Name	RW	Default	RV	Description
5	NO_STRETCH	rw	0	-	IIC主从时钟延展功能选择 0：支持时钟延展； 1：不支持时钟延展
4-3	FLT_SEL	rw	0	-	IIC接口数字滤波器选择，无特殊情况软件配置值为2'b10 2'b11：滤除3*Tiic_baud_clk 以下尖峰脉宽；* *2'b10：滤除2*Tiic_baud_clk 以下尖峰脉宽； 2'b01：滤除Tiic_baud_clk 以下尖峰脉宽； 2'b00：关闭数字滤波器
2	SLAVE MODE	rw	0	-	IIC接口主从机模式： 0：主机模式； 1：从机模式
1	I2C_RST	rw	0	-	IIC接口复位（使能后再释放复位） 0：IIC接口复位； 1：IIC接口释放
0	EN	rw	0	-	IIC接口使能。 0：关闭IIC接口 1：打开IIC接口

12.3.2 IIC_TASK: iic task register

Bit	Name	RW	Default	RV	Description
31-4	RESERVED	-	-	-	预留
12	SLAVE_CALL	r	-	-	从机地址匹配（仅debug，用户正常流程不可使用）
11	BC_CALL	r	-	-	广播地址匹配（仅debug，用户正常流程不可使用）
10	SLAVE_RW	r	-	-	从机读写状态（仅debug，用户正常流程不可使用）
9-6	RUNNING_TASK	r	-	-	正在运行事务（仅debug，用户正常流程不可使用）
5	TASK_RUN_RDY	r	-	-	事务运行状态（仅debug，用户正常流程不可使用）
4	TASK_LOAD_RDY	r	-	-	事务加载状态（仅debug，用户正常流程不可使用）

Bit	Name	RW	Default	RV	Description
3-0	RUN_TASK	w	0x0	-	事务操作：以下9种事务操作涵盖了主从机接收发送时的总线行为序列（具体解析见下节） 0x0：SEND_RESET 0x1：SEND_ADDR 0x2：SEND_DATA 0x3：SEND_ACK 0x4：SEND_NACK 0x5：SEND_STOP 0x6：SEND_NACK_STOP 0x7：RECV_DATA 0x8：RECV_DATA_WITH_ACK 0x9：RECV_DATA_WITH_NACK

12.3.3 IIC_PND: iic pending register

Bit	Name	RW	Default	RV	Description
31-29	RESERVED	-	-	-	预留
28	RXTASK_LOAD_PND	r	0x0	-	硬件将当前的接收相关的task事务加载完成
27	TXTASK_LOAD_PND	r	0x0	-	硬件将当前的发送相关的task事务加载完成
26	RXBYTE_DONE_PND	r	0x0	-	接收到1byte的数据
25	ADR_MATCH_PND	r	0x0	-	作为从机接收到与ADDR相符的地址（开始广播地址响应使能后，匹配广播地址也会起该PND）
24	RXNACK_PND	r	0x0	-	发送地址/数据后接收到NACK
23	RXACK_PND	r	0x0	-	发送地址/数据后接收到ACK
22	STOP_PND	r	0x0	-	接收到总线STOP信号序列
21	RESTART_PND	r	0x0	-	接收到总线RESTART信号序列
20	TASK_PND	r	0x0	-	事务执行完成PND
19	RESERVED	-	-	-	预留
18	RXTASK_LOAD_CLR	w	-	-	写“1”清除对应的PND
17	TXTASK_LOAD_CLR	w	-	-	写“1”清除对应的PND
16	RXDATA_DONE_CLR	w	-	-	写“1”清除对应的PND
15	ADR_MATCH_CLR	w	-	-	写“1”清除对应的PND
14	RXNACK_CLR	w	-	-	写“1”清除对应的PND

Bit	Name	RW	Default	RV	Description
13	RXACK_CLR	w	-	-	写“1”清除对应的PND
12	STOP_CLR	w	-	-	写“1”清除对应的PND
11	RESTART_CLR	w	-	-	写“1”清除对应的PND
10	TASK_CLR	w	-	-	写“1”清除对应的PND
9	RESERVED	-	-	-	预留
8	RXTASK_LOAD_IE	rw	0		对应PND的中断使能
7	TXTASK_LOAD_IE	rw	0		对应PND的中断使能
6	RXDATA_DONE_IE	rw	0		对应PND的中断使能
5	ADR_MATCH_IE	rw	0		对应PND的中断使能
4	RXNACK_IE	rw	0		对应PND的中断使能
3	RXACK_IE	rw	0	-	对应PND的中断使能
2	STOP_IE	rw	0	-	对应PND的中断使能
1	RESTART_IE	rw	0	-	对应PND的中断使能
0	TASK_IE	rw	0	-	对应PND的中断使能

12.3.4 IIC_TXBUF: iic tx buff register

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	-
7-0	TX_BUFF	rw	-	-	待发送数据

12.3.5 IIC_RXBUF: iic tx buff register

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	-
7-0	RX_BUFF	r	-	-	已接收数据

12.3.6 IIC_ADDR: iic device address register

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	-

Bit	Name	RW	Default	RV	Description
7-1	IIC DEVICE ADDR	rw	0x00	-	IIC 设备地址（仅支持7bit模式）
0	W/R OPTION	rw	0x0	-	作为主机发送地址时的读写标志： 0：发送数据到从机； 1：读取从机的数据；

12.3.7 IIC_BAUD: iic baud clk register

Bit	Name	RW	Default	RV	Description
31-12	RESERVED	-	-	-	-
11-0	BAUD_CNT	rw	-	-	IIC 总线传输时钟SCL速率配置： FIIC_BUS_SCL = FIIC_clk / BUAD_CNT *注意：不可设置为0*

12.3.8 IIC_TSU: iic setup time register

Bit	Name	RW	Default	RV	Description
31-7	RESERVED	-	-	-	-
6-0	SETUP_CNT	rw	-	-	IIC 总线SDA信号更新后到SCL时钟上升的最少时间配置： Tsetup = TIIC_clk * SETUP_CNT 无特殊情况软件配置值为2；

12.3.9 IIC_THD: iic hold time register

Bit	Name	RW	Default	RV	Description
31-7	RESERVED	-	-	-	-
6-0	HOLD_CNT	rw	-	-	IIC 总线SCL时钟下降到SDA信号更新前的最少时间配置： Thold = TIIC_clk * HOLD_CNT 无特殊情况软件配置值为2；

12.3.10 IIC_DBG: iic debug register

Bit	Name	RW	Default	RV	Description
31-16	SDA_DBG	r	-	-	SDA相关调试信号
15-0	SCL_DBG	r	-	-	SCL相关调试信号

12.3.11 IIC_TRIG_EN: iic task register

Bit	Name	RW	Default	RV	Description
31-9	RESERVED	-	-	-	预留
8	RXTASK_LOAD_TRIG_EN	rw	0		对应送到UDMA_GEN 源头的使能
7	TXTASK_LOAD_TRIG_EN	rw	0		对应送到UDMA_GEN 源头的使能
6	RXDATA_DONE_TRIG_EN	rw	0		对应送到UDMA_GEN 源头的使能
5	ADR_MATCH_TRIG_EN	rw	0		对应送到UDMA_GEN 源头的使能
4	RXNACK_TRIG_EN	rw	0		对应送到UDMA_GEN 源头的使能
3	RXACK_TRIG_EN	rw	0	-	对应送到UDMA_GEN 源头的使能
2	STOP_TRIG_EN	rw	0	-	对应送到UDMA_GEN 源头的使能
1	RESTART_TRIG_EN	rw	0	-	对应送到UDMA_GEN 源头的使能
0	TASK_TRIG_EN	rw	0	-	对应送到UDMA_GEN 源头的使能

12.4 基本事务操作

IIC_TASK 事务寄存器允许写入以下9种事务操作值来驱动IIC接口做出相应总线行为序列：

1. 0x0：SEND_RESET

工作在主机模式下，向IIC总线发送START 和 STOP；

2. 0x1：SEND_ADDR

工作在主机模式下，向IIC总线发送START 和 ADDR的数据，并接收从机回复的ACK/NACK；

3. 0x2：SEND_DATA

工作在主机模式下，向IIC总线发送TX BUFF的数据，并接收从机回复的ACK/NACK；

4. 0x3：SEND_ACK

工作在主/从机模式下，向IIC总线发送ACK；

5. 0x4：SEND_NACK

工作在主/从机模式下，向IIC总线发送NACK；

6. 0x5：SEND_STOP

工作在主机模式下，向IIC总线发送STOP；

7. 0x6：SEND_NACK_STOP

工作在主机模式下，向IIC总线发送NACK和STOP；

8. 0x7：RECV_DATA

工作在主机/从机模式下，从IIC总线上接收一个字节数据，但不进行ACK/NACK（持续拉低SCL线，总线上的设备需支持时钟延展），直到SEND_ACK/SEND_NACK的事务被填充，SCL才会撤销拉低；

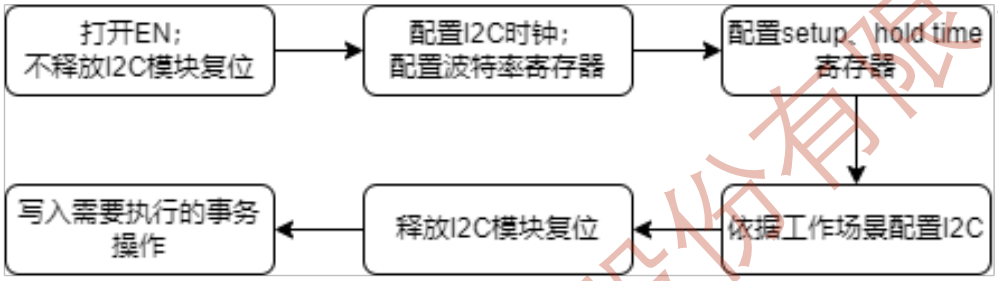
9. 0x8：RECV_DATA_WITH_ACK

工作在主机/从机模式下，从IIC总线上接收一个字节数据，且随后向IIC总线发送ACK；

10. 0x9：RECV_DATA_WITH_NACK

工作在主机/从机模式下，从IIC总线上接收一个字节数据，且随后向IIC总线发送NACK；

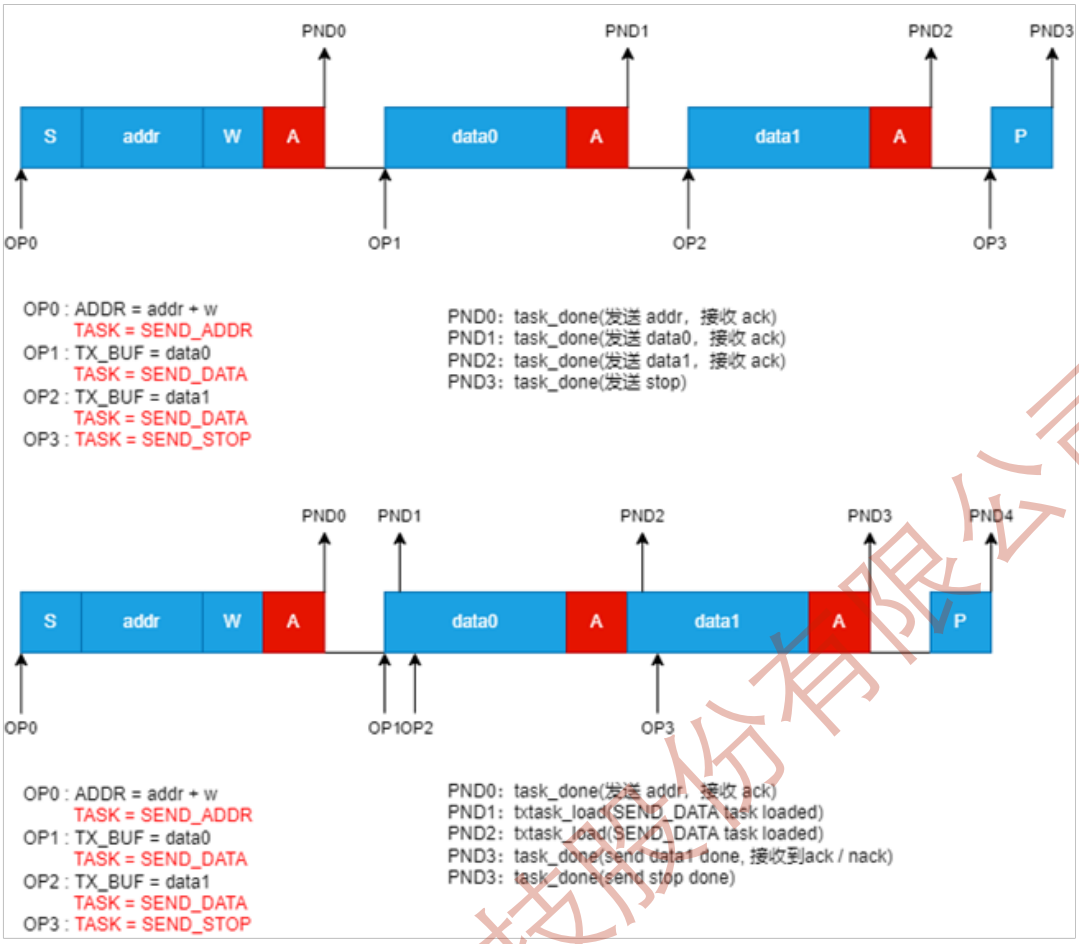
12.5 基本工作场景使用流程



1. 主机发送

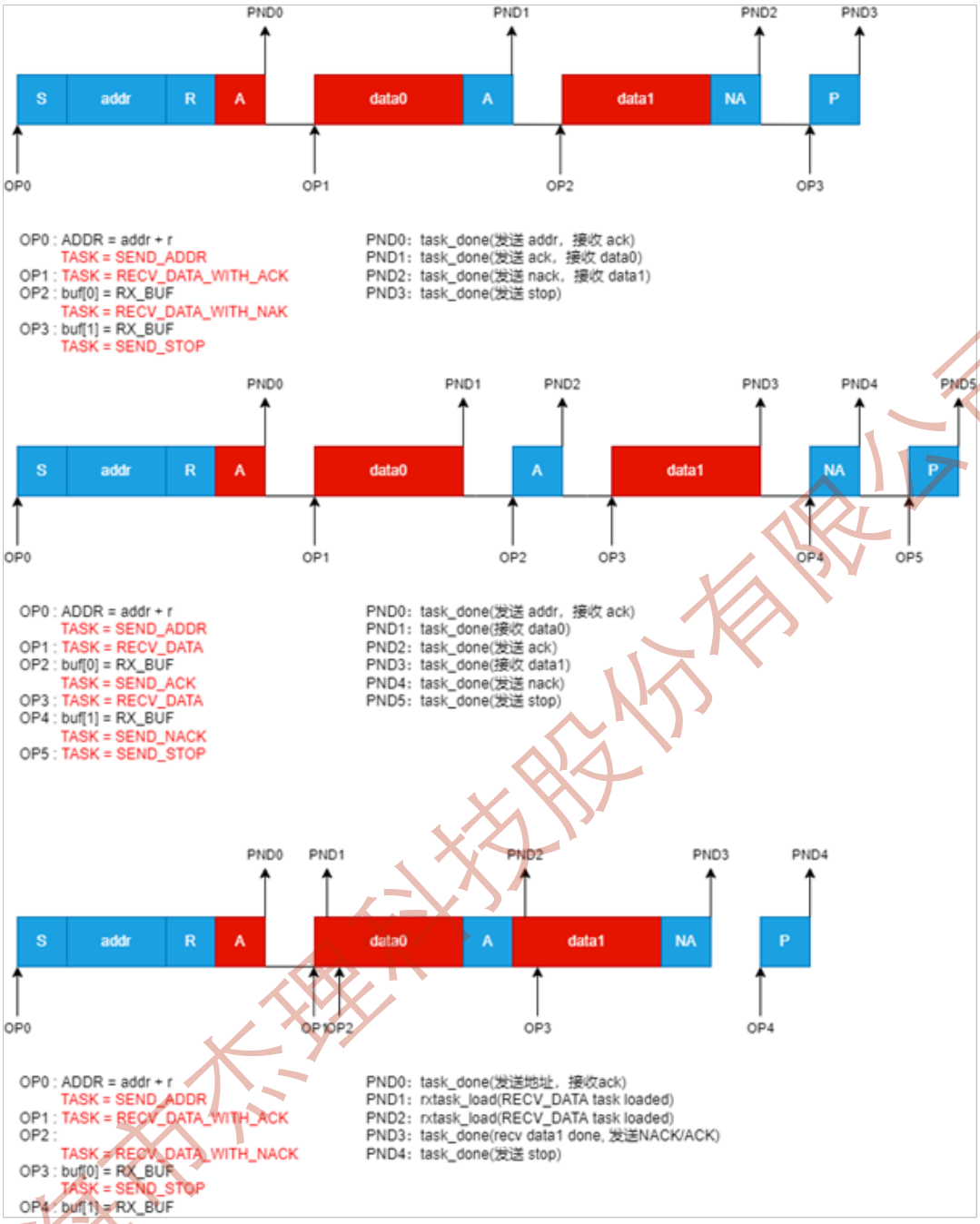
主机发送具有字节断续和字节连续发送两种工作场景。

字节连续发送：用于传输数据量较大的（中途无需进行数据处理）场景，主机在发送地址并且匹配后通过txtask_pnd来预先（提前约8个I2C_SCL时钟周期）填充下一事务达到连续发送。



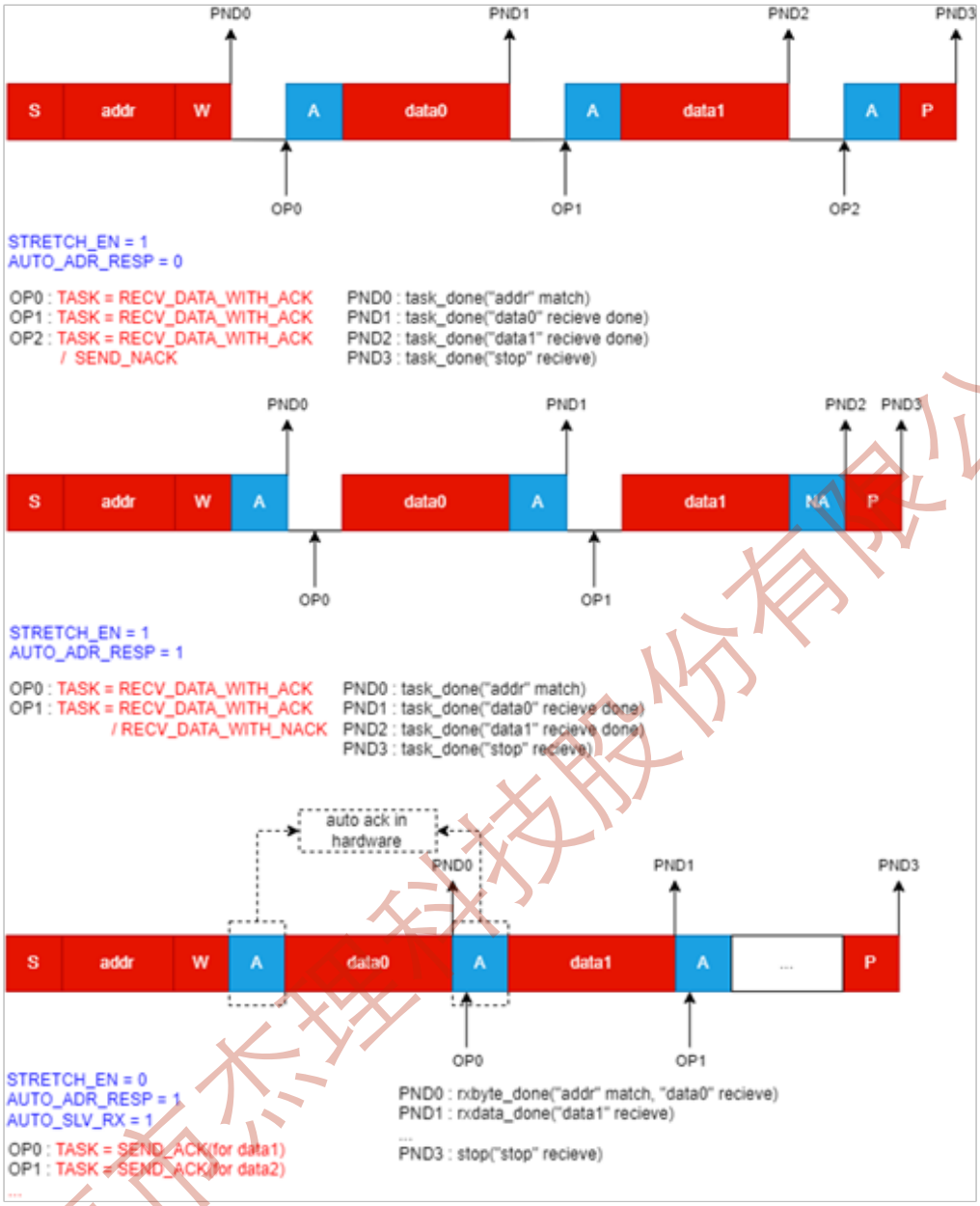
2. 主机接收

与主机发送相似，主机发送同样具有字节断续和字节连续发送两种工作场景；

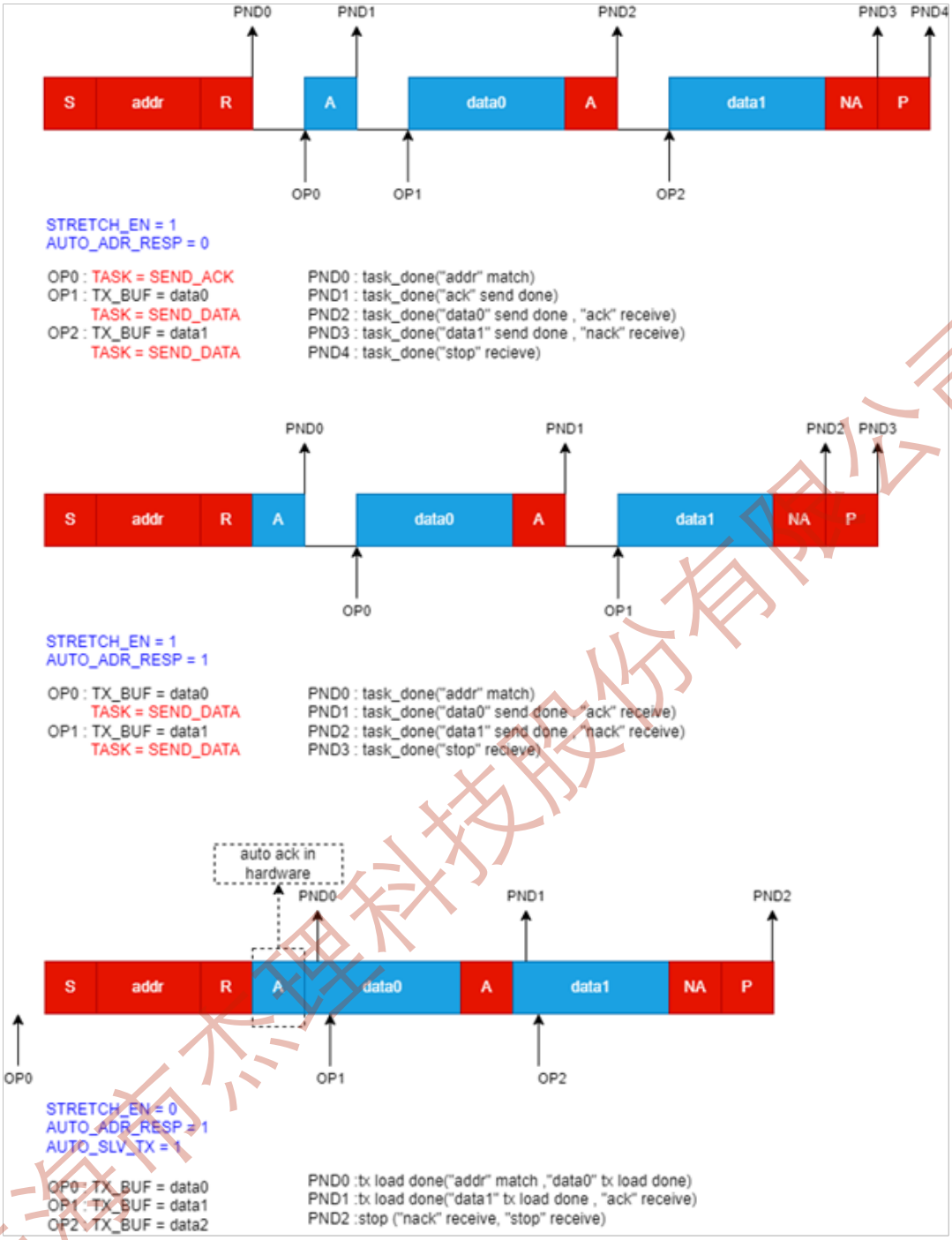


3. 主机其他操作（用于复位总线上其他从设备）

4. 从机接收

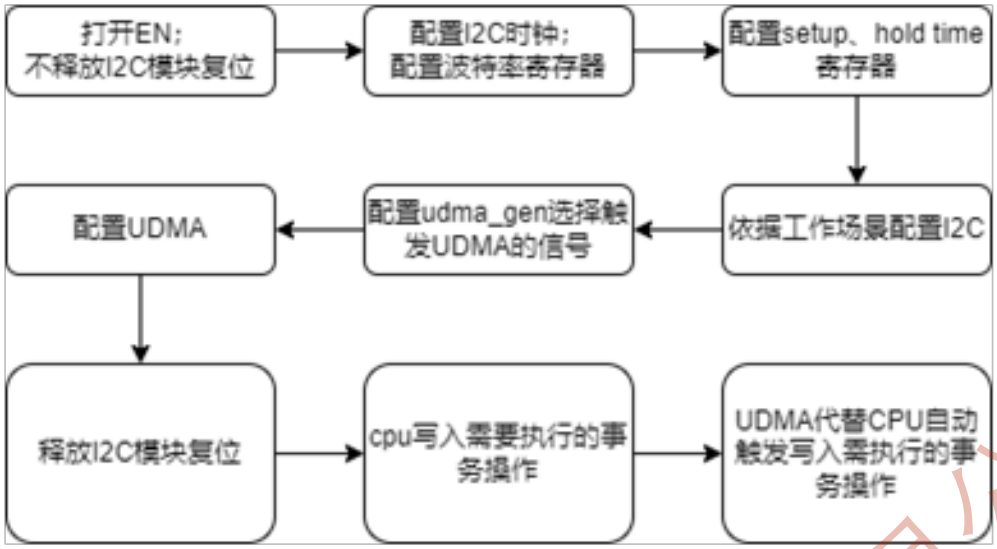


5. 从机发送



UDMA工作场景使用流程

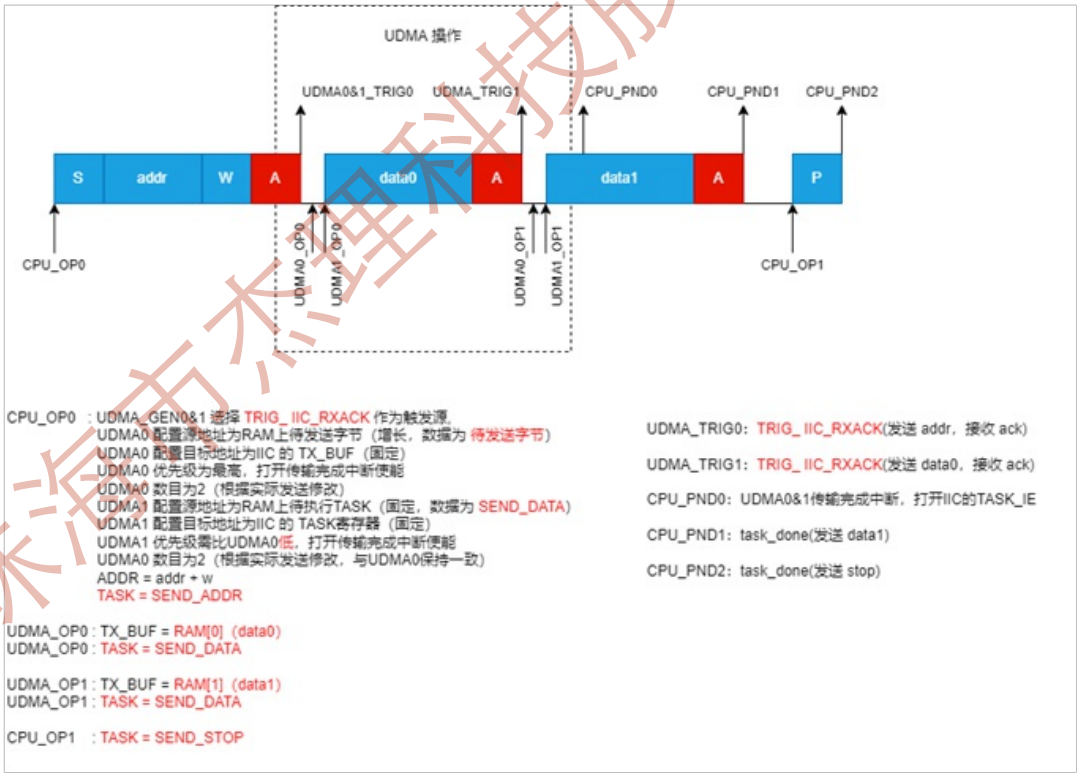
对于UDMA介入的IIC工作场景基本使用流程如下，各个具体场景的具体配置会有细微差别，本质上UDMA在代替CPU完成sfr读写操作；



以下介绍在作为主机/从机（与不支持时钟延展主机通信）使用UDMA完成发送接收的具体流程；

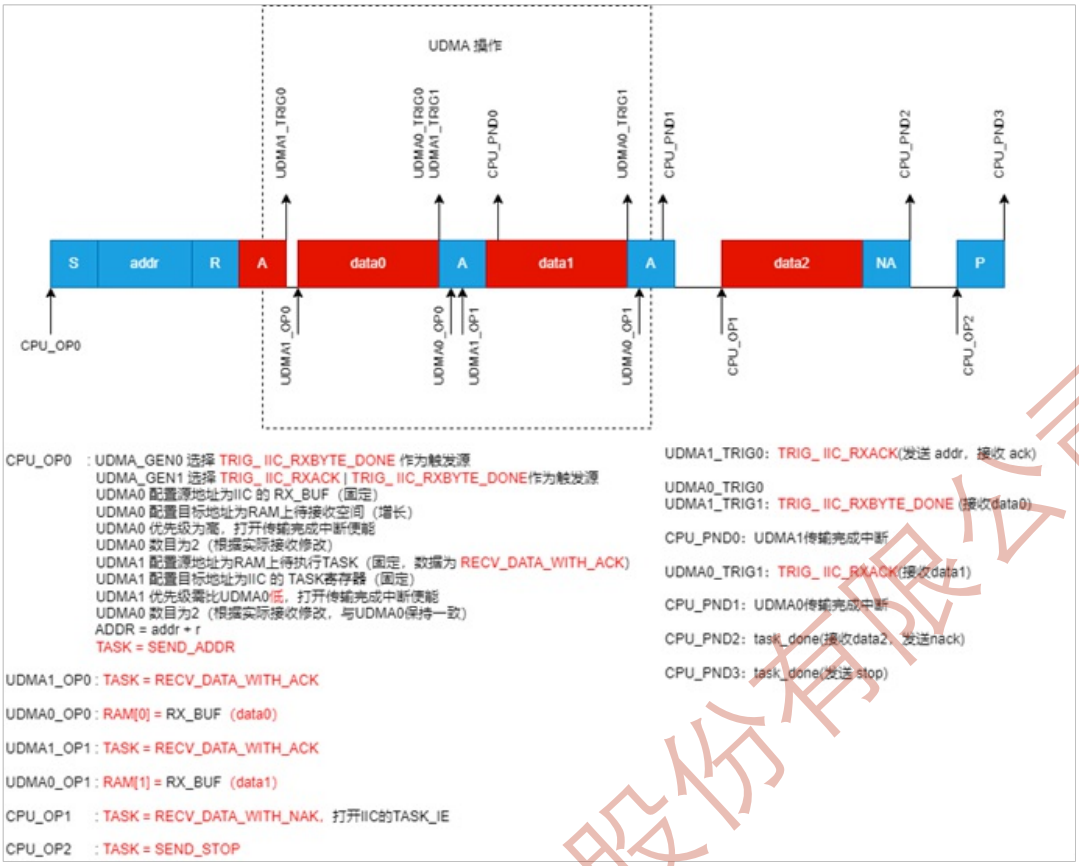
1. 主机发送

主机发送使用一个UDMA进行TX数据填充，使用一个UDMA进行TASK填充，可以不需要CPU中断介入完成数据阶段的字节连续发送，但对于地址和最后的STOP 发送仍需CPU介入操作；



2. 主机接收

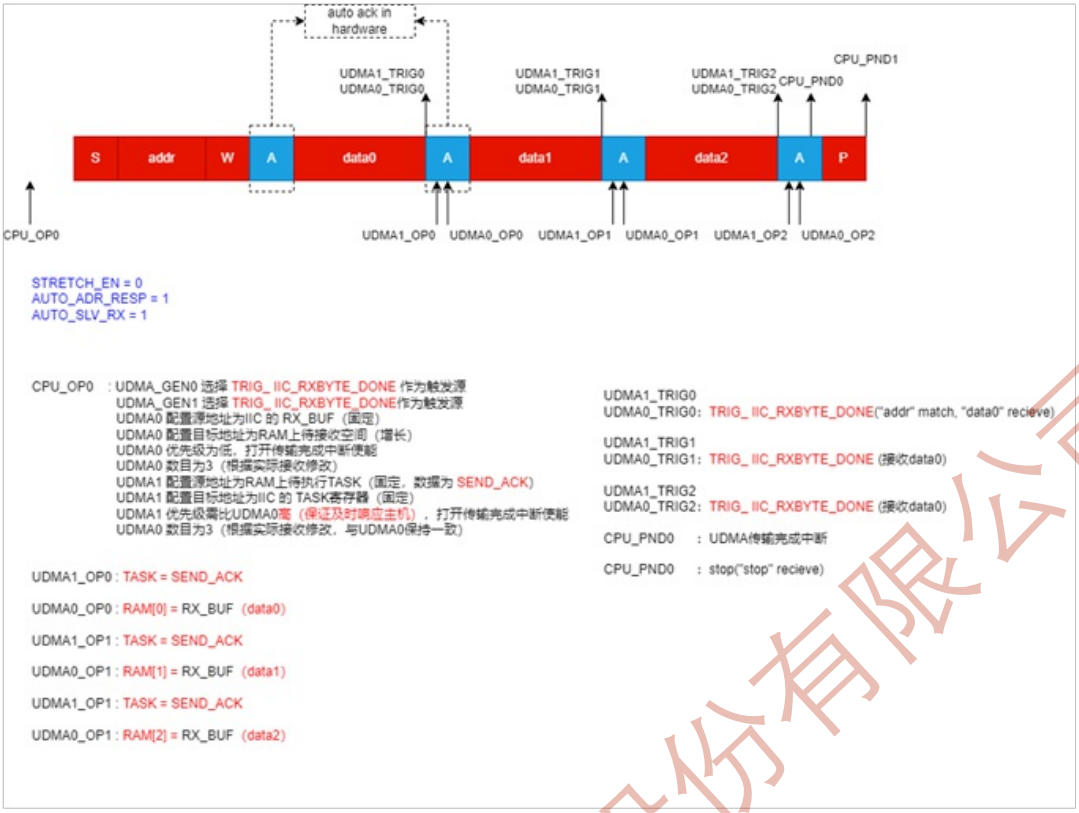
主机接收与主机发送类似，使用一个UDMA进行RX数据转移，使用一个UDMA进行TASK填充，可以不需要CPU中断介入完成数据阶段的字节连续接收，但对于地址、最后一个字节数据和STOP 发送仍需CPU介入操作；



3. 从机接收

从机接收，使用一个UDMA进行RX数据转移，使用一个UDMA进行TASK填充，可以不需要CPU中断介入完成数据阶段的字节连续接收，但在对于最后的STOP仍需CPU介入操作；

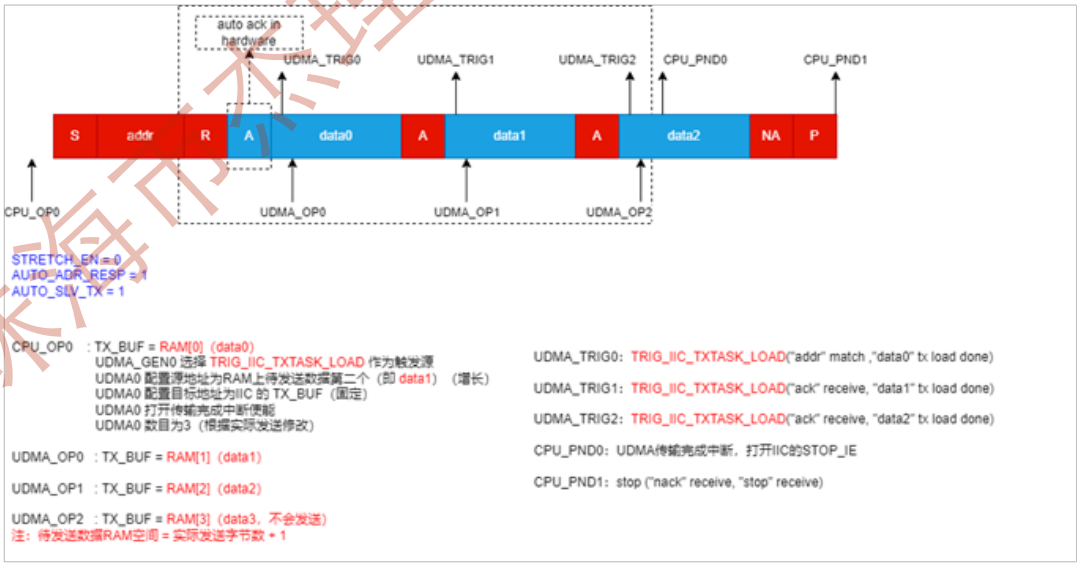
注意：需配置成非时钟延展，从机自动接收数据事务使能，从机接收地址硬件自动响应使能



4. 从机发送

从机发送，使用一个UDMA进行TX数据转移，可以不需要CPU中断介入完成数据阶段的字节连续接收，但在对于最后的STOP仍需CPU介入操作；

注意：需配置成非时钟延展，从机自动接收数据事务使能，从机自动发送数据事务使能



13 红外过滤（IRFLT）

13.1 概述

IRFLT是一个专用的硬件模块，用于去除掉红外接收头信号上的窄脉冲信号，提升红外接收解码的质量。IRFLT使用一个固定的时基对红外信号进行采样，必须连续4次采样均为‘1’时，输出信号才会变为‘1’；必须连续4次采样均为‘0’时，输出信号才会变为‘0’。换言之，脉宽小于3倍时基的窄脉冲将被滤除。改变该时基的产生可兼容不同的系统工作状态，也可在一定范围内调整对红外信号的过滤效果。通过对IOMC（IO re-mapping）寄存器的配置，可以将IRFLTx插入到对应TIMERx的捕获引脚之前。

例如通过IOMC寄存器选择了IRFLT0并且IRFLT_EN被使能之后，IO口的信号会先经过IRFLT0进行滤波，然后再送至TIMER0中进行边沿捕获。

1. 硬件接口

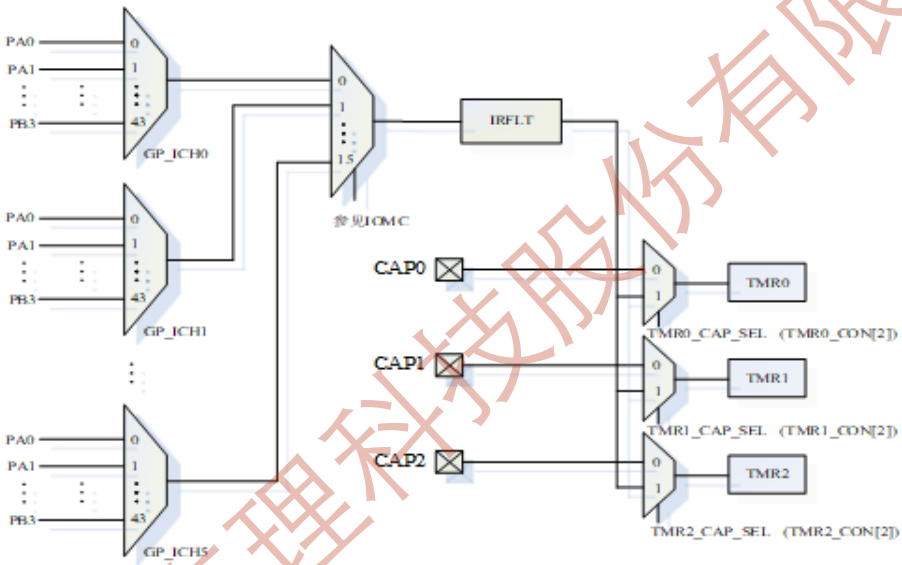


图 1-1硬件接口示意图

2. 时基选择

PSEL选定的分频倍数N和TSRC选定的驱动时钟的周期Tc共同决定了IRFLT用于采样红外接收信号的时基Ts：

$$Ts = Tc * N$$

又如，当选择24MHz的系统时钟，并且分频倍数为512时，Ts = 21.3uS。根据IRFLT的工作规则，所有小于(21.3*3=63.9uS)的窄脉冲信号，均会被滤除。

13.2 输入时钟源结构

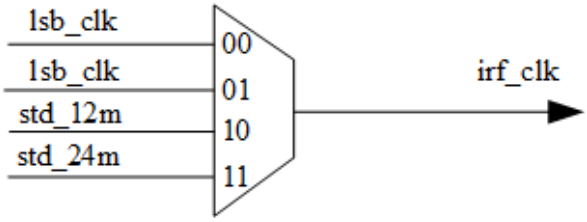


图1、irflt_clk时钟结构

13.3 寄存器说明

13.3.1 IRFLT_CON: irda filter contrl register

Bit	Name	RW	Default	RV	Description
7-4	PSEL	rw	x	-	PSEL[3-0]：时基发生器分频选择 0000：分频倍数为1； 0001：分频倍数为2； 0010：分频倍数为4； 0011：分频倍数为8； 0100：分频倍数为16； 0101：分频倍数为32； 0110：分频倍数为64； 0111：分频倍数为128； 1000：分频倍数为256； 1001：分频倍数为512； 1010：分频倍数为1024； 1011：分频倍数为2048； 1100：分频倍数为4096； 1101：分频倍数为8192； 1110：分频倍数为16384； 1111：分频倍数为32768；
3-2	TSRC	rw	x	-	TSRC[1-0]：时基发生器驱动源选择，见图 1-1
1	RESERVED	-	-	-	预留
0	IRFLT_EN	rw	0	-	IRFLT_EN： IRFLT使能 0：关闭IRFLT； 1：打开IRFLT；

14 LED_CONTROLER

14.1 概述

LEDC应用于LED灯带驱动和控制。通过配置，可以使led灯带显示不同颜色变化。支持dma数据传输，支持两路独立显示驱动。

14.2 特殊注意点

【注意】：使用该模块前需将CLK_SEL置0，然后再进行相关配置。否则可能出现异常。

14.3 数字模块控制寄存器

14.3.1 LEDC_CLK：ledc clk control register

Bit	Name	RW	Default	RV	Description
7-2	RESERVED	-	-	-	预留
1-0	CLK_SEL	rw	0x0	-	0：内部状态机复位，无时钟 1：lsb_clk 2：std_48m 3：std_24m

14.3.2 LEDx_CON：ledc(x) control register

Bit	Name	RW	Default	RV	Description
31-16	LEDC_BAUD	rw	x	-	LEDC输出波特率， CNT_CLK=BAUD_CLK/（LEDC_BAUD+1）， 每个LEDx波特率可单独配置
7	PND	r	0	-	中断标志，当ledc完成一次配置后，该位置1
6	CPND	w	0	1	用于清除中断标志，写1
5	IE	rw	0	1	硬件中断使能
4	IDLE_LEVEL	rw	x	-	设置ledc空闲状态输出信号电平 0：空闲输出0电平（OUT_INV=0） 1：空闲输出1电平（OUT_INV=0）
3	OUT_INV	rw	x	-	设置输出信号反转 0：信号不反转 1：信号反转

Bit	Name	RW	Default	RV	Description
2-1	BYTE_INV	rw	x	-	调整dma数据反向 0：不取反 1：1byte取反（0xaa<->0x55） 2：2byte取反（0xaabb<->0xdd55） 3：4byte取反（0xaabbccdd<->0xbb33dd55）
0	EN	rw	0	-	模块使能，置1启动

14.3.3 LEDx_FD：ledc(x) frame depth

Bit	Name	RW	Default	RV	Description
31-0	FD_LEN	rw	x	-	不重复n个LED数据（LED_BIT）循环， 设置值为（不重复个数 * 单个led位数）， 例如10个不重复RGB led灯： 10*24=240。

14.3.4 LEDx_LP：ledc(x) loop

Bit	Name	RW	Default	RV	Description
15-0	LP_LEN	r	0x0	-	FD_LEN重复次数，0则不重复

14.3.5 LEDx_TIX：ledc(x) high/low level timing parameters

Bit	Name	RW	Default	RV	Description
31-24	T1H	rw	x	-	输出1码高电平时间，单位CNT_CK。实际值为写入值加1
23-16	T1L	rw	x	-	输出1码低电平时间，单位CNT_CK。实际值为写入值加1
15-8	T0H	rw	x	-	输出0码高电平时间，单位CNT_CK。实际值为写入值加1
7-0	T0L	rw	x	-	输出0码低电平时间，单位CNT_CK。实际值为写入值加1

14.3.6 LEDx_RSTX：ledc(x) reset timing parameters

Bit	Name	RW	Default	RV	Description
31-8	RSTL	rw	x	-	每帧一开始复位有效时间长度，单位CNT_CK
7-0	RSTH	rw	x	-	每帧最后一bit到起pnding的时间间隔，单位CNT_CK。 0为不输出，否则实际值为写入值加1

14.3.7 LEDx_ADR：ledc(x) dma adr

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
31-0	ADR	w	x	-	数据输入基地址,必须为32位对齐地址值，只写，读为0。

14.4 LEDC控制流程 (IC内部)

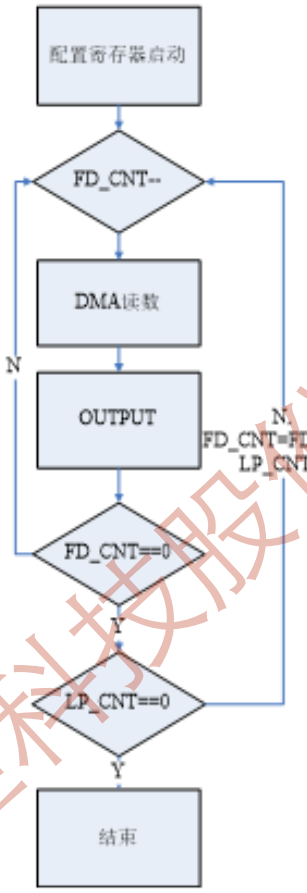


图1 LEDC控制流程图

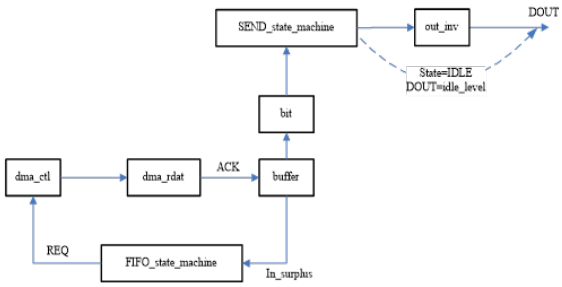


图2 状态机流程

14.5 数据通路

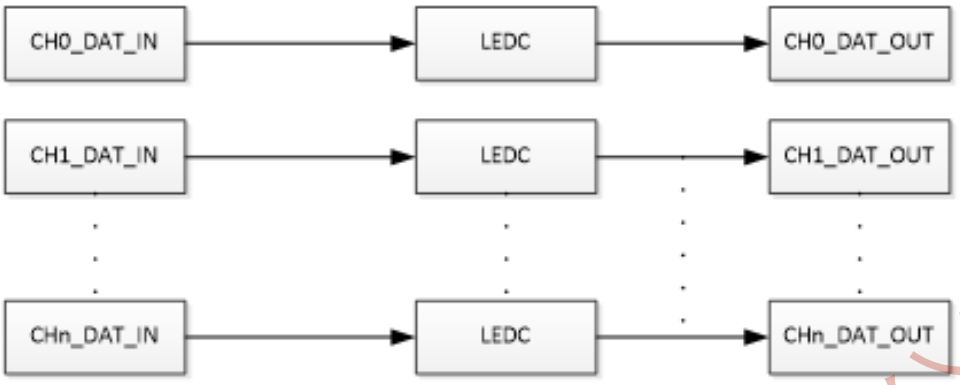


图3 数据通路图

14.6 DEMO CODE

例1配置CH0通道：

```
JL_LEDCK->CLK = 0;//make the clk_sel = 0;
SFR(JL_LED0->CON,5,1,1);//enable ie
SFR(JL_LED0->CON,4,1,x);//config idle_level
SFR(JL_LED0->CON,3,1,x);//config out_inv
SFR(JL_LED0->CON,1,2,x);//config byte_inv
JL_LED0->FD = XXXX;//config fd_len
JL_LED0->LP = XXXX;//config lp_len
SFR(JL_LED0->TIX,24,8,x);//config t1h
SFR(JL_LED0->TIX,16,8,x);//config t1l
SFR(JL_LED0->TIX,8,8,x);//config t0h
SFR(JL_LED0->TIX,0,8,x);//config t0l
SFR(JL_LED0->RSTX,8,24,x);//config rstl
SFR(JL_LED0->RSTX,0,8,x);//config rsth
SFR(JL_LED0->ADR,0,32,x);//config dma address
SFR(JL_LED0->CON,0,1,1);//enable ch0
while(!(JL_LED0->CON & BIT(7));
SFR(JL_LED0->CON,6,1,1);//clear pnd
```

15 LOW POWER MODE

15.1 概述

Low Power Mode具体可分为Idle/Standby/Sleep三种类型的低功耗模式，用于在不同需求下让系统进入低功耗的非工作状态，以尽可能的节省电能。在指定的唤醒事件发生时，系统会退出当前的低功耗模式，进入正常工作状态。

唤醒事件分为两大类：

1，任一外设的中断请求（pending）出现，且当时该模块中断使能打开时，发生第一类唤醒事件。

2，当某WAKEUP功能被使能（请参考Wakeup部分文档）且发生了相应的唤醒事件，或RTC的2mS/500mS/闹钟唤醒被使能，或LVD被使能且检测到低电压时，发生第二类唤醒事件。

Idle状态下(执行CPU“idle”指令进入)，CPU停止工作，其他被使能的设备则正常工作。当发生第一类或第二类唤醒事件时，CPU将从Idle状态被唤醒。若总中断有使能，则进入相应中断入口（如果由第二类唤醒，则不会进中断），执行该中断服务程序。中断返回后，CPU返回进入Idle的后一条地址处继续执行指令。

Standby状态下，所有数字设备停止工作，但被使能的模拟设备仍然正常工作。当发生第二类唤醒事件时，CPU将从Standby状态被唤醒，继续从进入Standby的后一条地址处继续执行指令，同时所有的其他数字外设也回到正常工作状态。

Sleep状态下，所有数字和时钟相关的模拟设备（包括PLL，XOSC，EOSC0，EOSC1，HTC，RC）均停止工作。当发生第二类唤醒事件时，CPU将从Sleep状态被唤醒，继续从进入Sleep的后一条地址处继续执行指令，同时所有的其他数字和模拟设备也回到正常工作状态。

15.2 寄存器说明

1. PWR_CON: power control register

参考“Reset_Source”文档

16 MCPWM

16.1 概述

MCPWM功能模块包括：n个timer时基模块，n对独立的PWM通道，n路故障保护输入。框图如下：

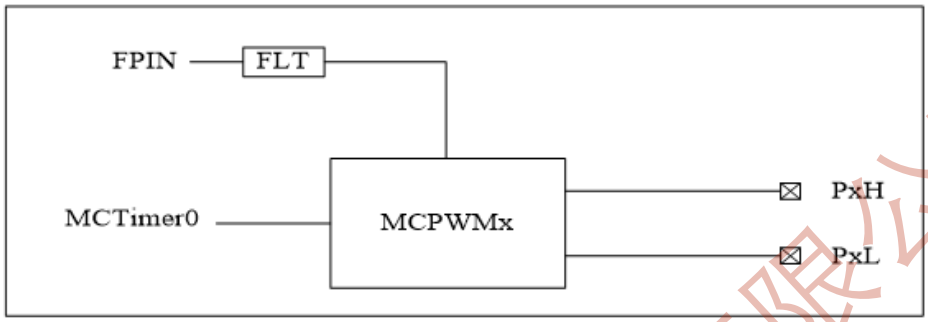


图1-1 MCPWM结构图

16.2 定时器 MCTIMER

1. 概述

   mctimer是16位定时器，可作为pwmx的时基控制

2. 模块特性 (1). 16位定时功能 (2). 带缓冲的周期寄存器 (3). 支持多种工作模式：递增、递增-递减、外部引脚控制递增或递减 (4). 多种中断模式：上溢出中断、下溢出中断、上下溢出中断

3. 注意：MCPWM增加了一BIT（MCPWM_CON BIT (16))用于开关MCPWM时钟，防止MCPWM模块不用时还有漏电；使用MCPWM之前需要打开这一BIT。

16.3 TIMER模块寄存器

16.3.1 TMRx_CON：Timer contrl register

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	-	-	预留
15	INCF	rw	0	-	递增递减标志位 0：递减 1：递增
13	UFPND	r	0	-	递减借位标志 0：没借位 1：已发生借位

Bit	Name	RW	Default	RV	Description
12	OFPN	r	0	-	计数溢出（TMRXCNT == TMRXPR）标志 0：没溢出 1：已发生溢出
11	UFCLR	w	-	1	清除UFPND标志位 0：无效 1：清除UFPND
10	OFCLR	w	-	1	清除OFPND标志位 0：无效 1：清除OFPND
9	UFIE	rw	0	1	定时递减借位中断使能 0：禁止 1：允许
8	OFIE	rw	0	1	计数溢出中断使能 0：禁止 1：允许
7	CKSRC	rw	0	-	定时器时钟源 0：内部时钟 1：外部引脚时钟
6-3	CKPS	rw	0	-	时钟预分频设置，TCK/（2^TCKPS）
2	RESERVED	-	-	-	预留
1-0	MODE	rw	0	-	定时器工作模式 00：保持计数值 01：递增模式 10：递增-递减循环模式 11：递减模式

16.3.2 TMRx_CNT: counter initial value register

Bit	Name	RW	Default	RV	Description
15-0	TMR_CNT	rw	x	-	计数/定时初值

16.3.3 TMR_PR: counter target value register

Bit	Name	RW	Default	RV	Description
15-0	TMR_PR	rw	x	-	计数/定值目标值

16.4 模块引脚

PWM0：PWMCH0H、PWMCH0L

PWM1：PWMCH1H、PWMCH1L

PWM2：PWMCH2H、PWMCH2L

……

故障输入引脚：FPIN

16.5 模块特性

1. 边沿对齐和中心对齐输出模式
2. 可运行过程中更改PWM频率、占空比
3. 多种更新频率/占空比模式：上溢出重载、下溢出重载、上下溢出重载
4. 灵活配置每对通道的有效电平状态
5. 可编程死区控制
6. 硬件故障输入引脚

16.6 PWM模块控制寄存器

16.6.1 CHx_CON0: pwm control register0

Bit	Name	RW	Default	RV	Description
15-12	DTCKPS	rw	0	-	死区时钟预分频，Tsys/（2^ DTCKPS）
11-7	DTPR	rw	0	-	死区时间控制 死区时间：Tsys/(2^ DTCKPS) × (DTPR+1)
6	DTEN	rw	0	-	死区允许控制，对应PWMCHxH、PWMCHxL 0：禁止 1：允许
5	L_INV	rw	0	-	对应PWMCHXL输出反向控制 0：反向禁止 1：反向允许
4	H_INV	rw	0	-	对应PWMCHXH输出反向控制 0：反向禁止 1：反向允许
3	L_EN	rw	0	-	对应PWMCHXL输出允许控制 0：禁止PWMCHXL 1：允许PWMCHXL
2	H_EN	rw	0	-	对应PWMCHxH输出允许控制 0：禁止PWMCHxH 1：允许PWMCHxH

Bit	Name	RW	Default	RV	Description
1-0	CMP_LD	rw	0	-	CHx_CMP重新载入控制 00：时基TMRX_CNT等于“0”载入 01：时基TMRX_CNT等于“0”或者等于TMRX_PR时候载入 10：时基TMRX_CNT等于TMRX_PR时候载入 11：立即载入

16.6.2 CHX_CON1: pwm control register1

Bit	Name	RW	Default	RV	Description
15	FPND	r	0	-	故障保护输入标志，只读，写无效 读0：未发生保护 读1：已发生保护，模块的PWM引脚会变成高阻态
14	FCLR	w	-	-	清除FPND标志位，只写，读为“0” 写0：无效 写1：清除FPND
13-12	RESERVED	-	-	-	预留
11	INTEN	rw	0	1	FPND中断允许 0：禁止 1：允许
10-8	TMRSEL	rw	0	-	选择TMR0-7作为PWM时基 0-7：选择时基
7-5	RESERVED	-	-	-	预留
4	FPINEN	rw	0	1	故障保护输入允许控制 0：禁止保护 1：允许保护
3	FPINAUTO	rw	0	1	故障自动保护控制 0：禁止自动保护 1：允许自动保护， 当检测到故障引脚有效信号时， 自动把PWM引脚设成高阻态，直到软件清除FPND。
2-0	FPINSEL	rw	0	-	故障保护输入引脚选择 0-7：选择FPIN作为故障保护输入 (0-5为io输入，6-7固定为0)

16.6.3 CHX_CMPH: pwm high port compare register

Bit	Name	RW	Default	RV	Description
15-0	MMRX_PRH	rw	-	-	带缓冲的16位比较寄存器，对应PWMCHXH引脚的占空比控制

16.6.4 CHX_CMPL: pwm low port compare register

Bit	Name	RW	Default	RV	Description
15-0	TMRX_PRL	rw	-	-	带缓冲的16位比较寄存器，对应PWMCHXL引脚的占空比控制

16.6.5 FPIN_CON：input filter control register

Bit	Name	RW	Default	RV	Description
23-16	EDGE	rw	0	-	FPINx边沿选择 0：下降沿 1：上升沿
15-8	FLT_EN	rw	0	-	FLT_EN：FPINx滤波使能开关 0：滤波关闭 1：滤波开启
7-6	RESERVED	-	-	-	预留
5-0	FLT_PR	rw	0	-	FLT_PR：滤波宽度选择 滤波宽度=16 × FLT_PR × lsb_clk

16.6.6 MCPWM_CON：mcpwm control register

Bit	Name	RW	Default	RV	Description
16	CLK_EN	rw	0	-	mcpwm 模块时钟使能， 配置mcpwm模块之前先配该位为1
15-8	TMR_EN	rw	0	-	定时器计数开关控制(tmr7-0) 0：定时器计数关闭 1：定时器计数打开
7-0	PWM_EN	rw	0	-	模块开关控制（pwm7-0） 0：模块关闭 1：模块开启

17 PULSE COUNTER

17.1 概述

以下为pulse_counter/触摸键的寄存器说明及使用方法。其工作原理如下所示：

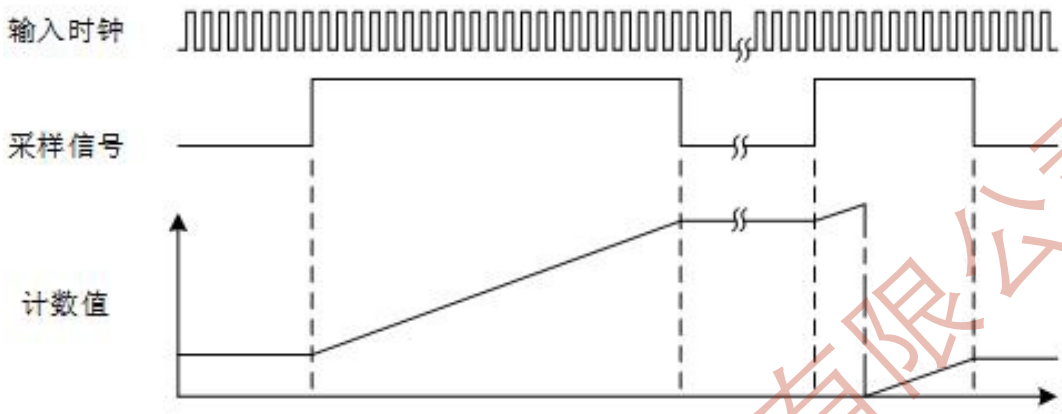


图 1-1 pulse counter计数原理

17.2 寄存器说明

17.2.1 PL_CNT_CON

Bit	Name	RW	Default	RV	Description
31-4	RESERVED	-	-	-	预留
3-2	CLK_SEL	rw	0x0	-	pulse counter 时钟选择 00:选择std_48m作为计数时钟； 01: 选择ich_clk_pin作为计数时钟，具体可参考IOMC； 10: 选择pll_d1p5_mo作为计数时钟； 11: 选择pll_d1p0_mo作为计数时钟； 备注：时钟选择的频率越大，pl_cnt_value计数值会越大， 分辨率会越高，触摸键的灵敏度也越高
1	EN	rw	0	-	触摸键使能，当cap_mux_in为1时，PLCNTVL累加计数，计满后归0再重新累加 0: 触摸键禁止； 1: 触摸键使能
0	TEST_EN	rw	0	0	触摸键测试使能（此位专用于测试） 0: 测试模式未使能； 1: 测试模式使能；

【注意】：当cap_mux_in选择TMRx_PWMOUT, pulse counter时钟选择clk_mux_in时，可以用内部固定周期的PWM，计算芯片IO上不确定的时钟频率。(具体cap_mux选择参考IOMC文档)

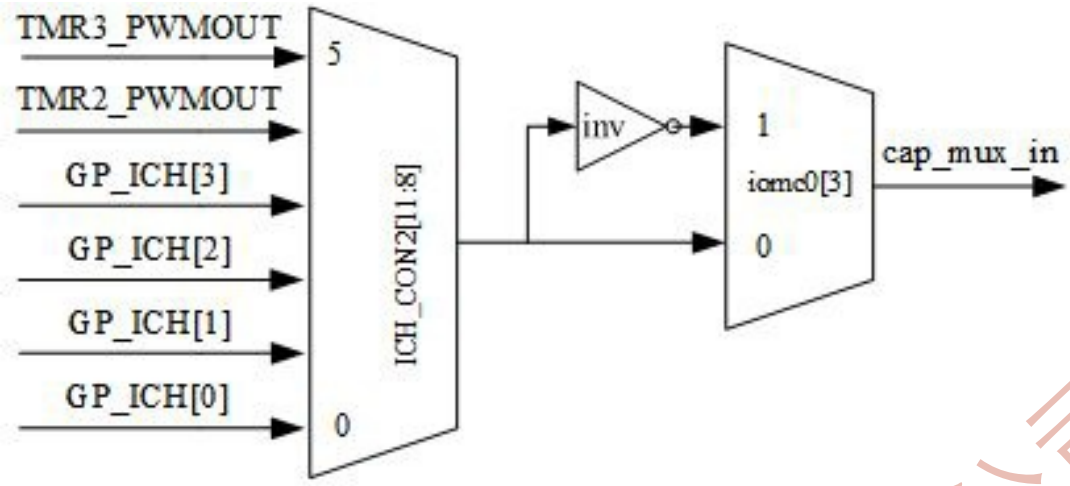


图 1-2 cap_mux_in电路控制图

17.2.2 PL_CNT_VAL

Bit	Name	RW	Default	RV	Description
31-0	VALUE	r	-	-	32位counter递增计数值，默认值为随机数

【注意】：芯片上电时Pluse Counter的PL_CNT_VAL都会产生一个随机的初始值，因此每次启动Pluse Counter前都需要读取一次

17.3 示例代码

以下以PA1为例:

1. 变量定义：

```
int Touchkey_value_old, Touchkey_value_new, Touchkey_value_delta;   //32bit
```

2. 选择检测管脚

```
JL_IMAP->FI_GP_ICH1 = PA1_IN; // PA1选通为ICH1
JL_IOMC->ICH_CONx = (ICH1)<<8; //PLUSE COUNTER 选择使用ICH1作为输入
```

3. 初始化计数器配置

```
JL_PCNT->CON  = 0;      //
JL_PCNT->CON |= (PCCON_CLOCK_3)<<2;   //选择触摸键时钟
JL_PCNT->CON |= BIT(1);     //使能计数器
```

4. 检测电容

```
Touchkey_value_old = JL_PCNT->VAL;  //读取旧值
JL_PORTA->DIR |= BIT(   1) ;            //对应管脚输入使能
While(PORTA_IN & BIT(1));
Touchkey_value_new = JL_PCNT->VAL;
// 判断是否溢出
if(Touchkey_value_old > Touchkey_value_new)
```

```
Touchkey_value_new += 0x10000;
Touchkey_value_delta = Touchkey_value_new - Touchkey_value_old;
```

18 PWM LED灯

18.1 概述

PWM LED是通过脉冲宽度调节控制LED的亮度与亮灭的一个模块。放在DVDD，不可以在软关机下工作。

PWM LED模块可以配置两个周期大小不同的PWM0、PWM1，分别控制LED的亮度（PWM0）与亮灭（PWM1）。

18.2 特性

1. 驱动档位特性：

1. PWM推高电平: 高阻，上拉，输出1强驱0，输出1强驱1，输出1强驱2，输出1强驱3，保持，输出1强驱2，输出1强驱1，输出1强驱0，上拉，高阻， 保持（循环以上步骤）

2. PWM推低电平: 高阻，下拉，输出0强驱0，输出0强驱1，输出0强驱2，输出0强驱3，保持，输出0强驱2，输出0强驱1，输出0强驱0，下拉，高阻， 保持（循环以上步骤）

3. 最高驱动档位可设， 强驱及高阻保持时间可设， 驱动切换等待时间可设（所有等待时间为同样的n个PWM0_CLK）

2. 推灯功能：

1. 固定亮度

A.（单LED）单闪；双闪（双闪的两个时间长度可调，两个间隔可调）；

B.（双LED）单闪 变色 单闪；

C. 双闪 变色 双闪（双闪的两个时间长度可调，两个间隔可调）；

D. n闪 变色 n闪（n<7）；

2. 呼吸灯

A.（单LED）灭灯时间长度可调，亮灯时间长度可调，亮灯快慢可调，灭灯快慢可调；

B.（双LED）（灭灯时间长度可调，亮灯快慢可调，灭灯快慢可调）呼吸 变色 呼吸；

无灯—A灯逐渐变到最亮—最亮变色—B灯由最亮逐渐变到最暗—变色（无灯）—A灯逐渐变到最亮…；

无灯—A 灯呼吸—无灯—A灯逐渐变到 最亮—最亮变色—B灯由最亮逐渐变到最暗—无灯—B 灯呼吸—变色（无灯）—A灯呼吸…；

【注意】：单IO推两个灯电路原理图。在IO为高阻时，IOVDD的电压，不满足两个串联LED的压降，A、B灯均不点亮；IO输出高电平（IOVDD电平），A灯不亮，B灯被点亮；IO输出高电平（IOVDD电平），A灯被点亮亮，B灯不亮；当A、B灯交替点亮的频率大于人眼分辨极限，即可被认为两灯同时被点亮。

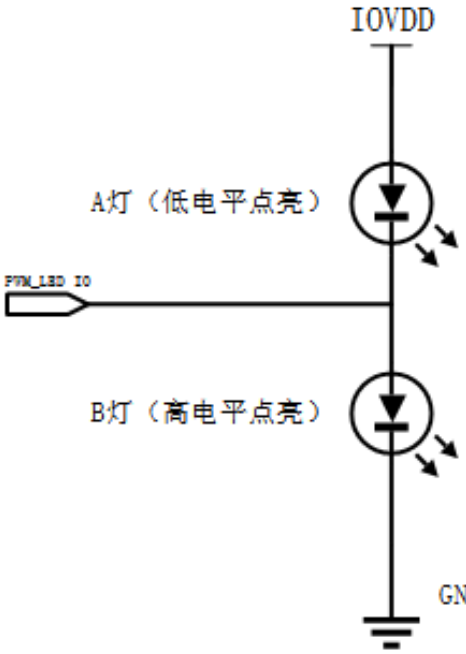


图 1-1 时钟控制框图

18.3 特殊注意点

1. 所有IO 可用（除PD口）。

2. PDOWN（蓝牙sniff）可用，SOFF不可用。

3. 目前只放了一个模块，只支持单IO单LED、单IO双LED。

4. IO在普通IO和LED模块IO间切换时，会有一个时钟的波形突变，可能会造成LED灯闪亮，IO配置需依照以下顺序，来降低该突变影响：

1. 普通IO→LED模块输出IO

xxxx //LED模块配置

JL_PORTX->DIE = 0x00;
JL_PORTX->DIEH = 0x00; // 关闭所使用IO的DIE和DIEH JL_PORTX->DIR = 0xff; // 所使用IO 设置为输入 JL_PORTX->PU0 = 0xff; // 打开所使用IO的上拉 JL_PORTX->PD0 = 0xff; // 打开所使用IO的下拉 JL_PORTX->DIR = 0x00; // 所使用IO 设置为输出 JL_PLED->CON0 |= BIT(0); // 使能LED模块，开始驱动LED

2. LED模块输出IO→普通IO

JL_PORTX->DIR = 0xff; // 所使用IO 设置为输入 JL_PORTX->PU0 = 0x00; // 恢复所使用IO的上拉设置 JL_PORTX->PD0 = 0x00; // 恢复所使用IO的下拉 JL_PLED->CON0 &= ~BIT(0); // 关闭LED模块

18.4 寄存器并接

1. PWM_ PRD_DIV ：pwm1_clk分频比控制器，12bit

PWM_ PRD_DIV[11:0] = {P3_PWM_CON3[3:0],PWM_ PRD_DIVL[7:0]}；

2. PWM_BRI_PRD[9:0]： 亮度周期，10bit

PWM_BRI_PRD [9:0] = { P3_PWM_BRI_PRDH [1:0], P3_PWM_BRI_PRDL [7:0]}；

3. P3_PWM_BRI_DUTY0 [9:0]： 推高电平点亮的灯，亮度占空比，10bit

P3_PWM_BRI_DUTY0 [9:0] = { P3_PWM_BRI_PRD0H [1:0], P3_PWM_BRI_PRD0L [7:0]}；

4. P3_PWM_BRI_DUTY1 [9:0]：推低电平点亮的灯，亮度占空比，10bit

P3_PWM_BRI_DUTY1[9:0] = { P3_PWM_BRI_PRD1H [1:0], P3_PWM_BRI_PRD1L [7:0]}；

5. 呼吸灯，灭灯延时计数器，16bit

{PWM_DUTY3[7:0] ,PWM_DUTY2[7:0]}；

18.5 时钟控制

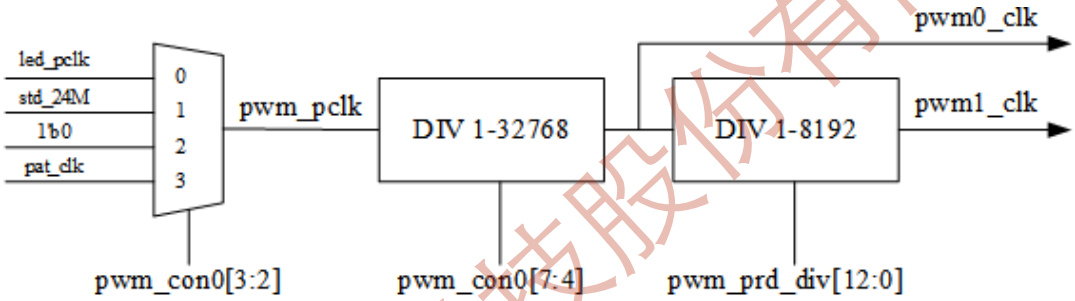


图 1-2 时钟控制框图

其中PWM0由pwm0_clk驱动，周期PRD0由PWM_BRI_PRD0/1 10bit决定，有一个循环计数bright_cnt决定显示亮度。

其中PWM1由pwm1_clk驱动，周期PRD1 最多256个时钟，有一个循环计数counter_cnt。

特别地，当呼吸灯功能打开，PWM0、PWM1分别控制LED的亮度级数变化及最亮和最暗保持的时间。

led_pclk的时钟来自P33的模块，用于低功耗模式时保持推灯模块工作。

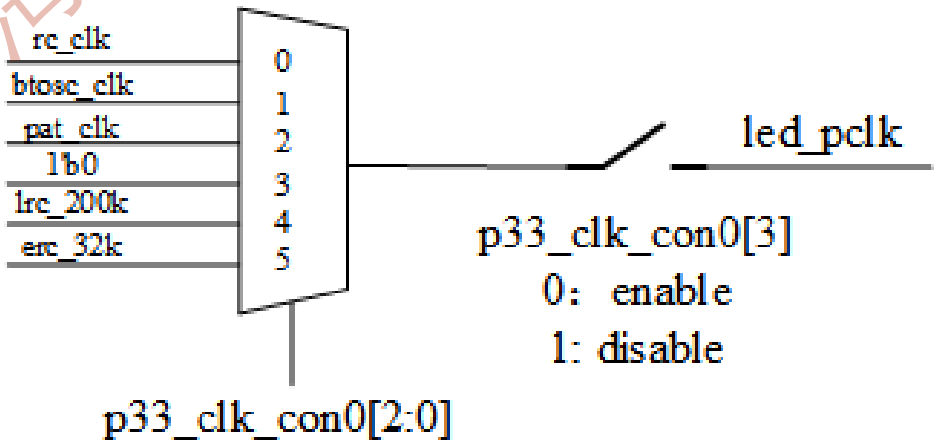


图 1-3 时钟控制框图

18.6 二级亮灭控制

PWM1可以控制LED在一个PRD1周期亮灭一次或两次。

下图以一个LED灯PWM电平举例：

输出高电平亮

单周期双闪

单灯

非呼吸灯模式

配置：

PWM_DUTY3_EN = 0； PWM_DUTY2_EN = 1； PWM_DUTY1_EN = 1； PWM_DUTY0_EN = 1；
PWM1_PRD_SEL = 0； //PWM1 PRD 为256 pwm1_clk PWM1_INV = 0; //固定为0不变，亮灭不取反，如下图，周期开始时是从灯灭开始 PWM0_INV = 0； //固定为0不变，灯亮度控制 OUT_LOGIC = 1；//固定为1不变，PWM输出逻辑控制 BREATHE_EN = 0; SHIFT_DUTY = 0;

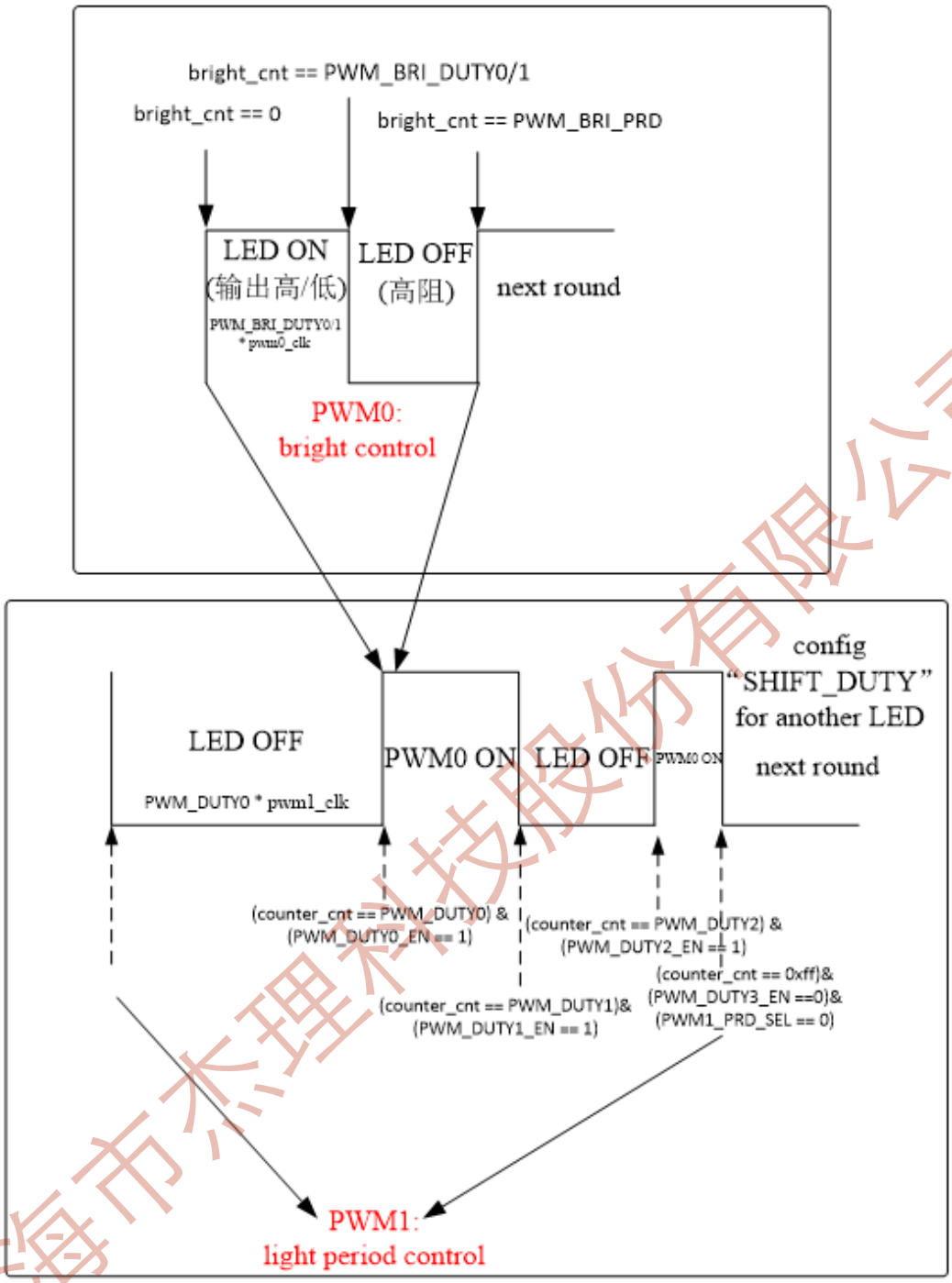
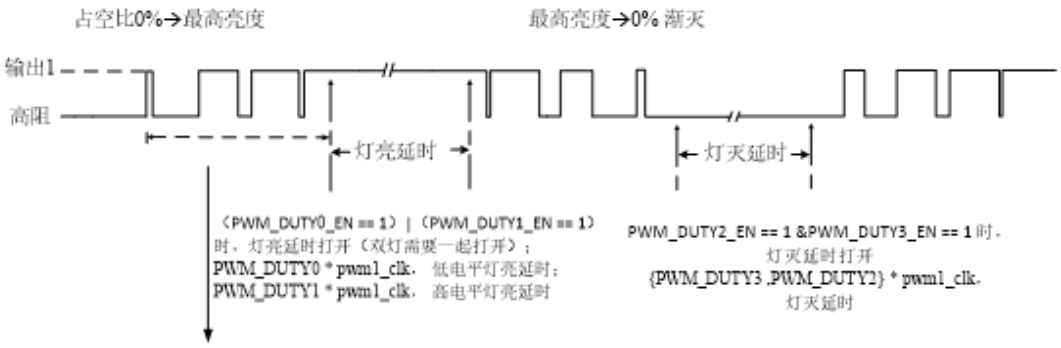


图 1-3 二级亮灭控制框图

【注意】：需要同时推两个颜色的灯时，请把PWM1的周期设置得尽可能短，小于人眼分辨极限即可。

18.7 呼吸灯控制



灯亮度变化的时间 T = pwm1_clk * PWM_BRI_DUTYx
（PWM_BRI_DUTY0是高电平，PWM_BRI_DUTY1是低电平）。
例如：pwm0_clk 周期25uS，pwm1_clk 周期5mS，

PWM_ PRD_DIV = 200； PWM_BRI_PRD = 200，
PWM_BRI_DUTY0 = 150，
PWM_BRI_DUTY1 = 170；

那么高电平灯从灭到亮度（150/200）时间
T0 = pwm1_clk * PWM_BRI_DUTY0= 750mS；
级数 N0 = T0/pwm0_clk*PWM_BRI_PRD=150
高电平灯灭灯时间一致。

那么低电平灯从灭到亮度（170/200）时间
T1 = pwm1_clk * PWM_BRI_DUTY1= 850mS；
级数 N1 = T1/pwm0_clk*PWM_BRI_PRD=170
低电平灯灭灯时间一致。

以上双灯亮灭转换时间不一致，可以用不同的高电平灯亮的延时、低电平灯亮的延时来平衡：
tmp = PWM_BRI_DUTY0 - PWM_BRI_DUTY1；
PWM_DUTY2 = PWM_DUTY0 + tmp；

图 1-4 呼吸灯控制示意图

18.8 控制寄存器说明

18.8.1 PWM_CON0: PWM control register 0

Bit	Name	RW	Default	RV	Description
7-4	PWM_PSET	rw	0x0		PWM_PSET: PWM pre_scaler setting。设置PWM0的分频器除法因子 [1:0]: 00:div1 01:div4 10:div16 11:div64 [2] : 0: x1 1: x2 [3] : 0: x1 1: x256 计算方式：DIV = [1:0]的divx 乘以 [2]倍数 乘以 [3]倍数。

Bit	Name	RW	Default	RV	Description
3-2	PWM_SSEL	rw	0x0		PWM_SSEL: PWM时钟源的选择。 00： led_clk (P3_CLK_CON0[2-0]： 0:rc_250k; 1:btosc; 2: isp_clk; 3:1'b0; 4:lrc_200k; 5:erc_32k; P3_CLK_CON0[3]: 0: enable 1: disable) 01： std_24MHz 11： pat_clk
1	BREATHE_EN	rw	0		呼吸灯使能。 0：呼吸灯功能关闭 1：呼吸灯功能打开
0	PWM_EN	rw	0		PWM使能。 0：模块总使能关闭 1：模块总使能打开

18.8.2 PWM_CON1: PWM control register 1

Bit	Name	RW	Default	RV	Description
7	PWM_DUTY3_EN	rw	0	-	亮灭DUTY控制3开关/呼吸灯空周期延时（高8bit）开关 0：亮灭DUTY控制3关闭（BREATHE_EN == 0 & PWM1_PRD_SEL == 0）/ 呼吸灯空周期延时关闭（BREATHE_EN == 1） 1：亮灭DUTY控制3开启（BREATHE_EN == 0 & PWM1_PRD_SEL == 0）/ 呼吸灯空周期延时开启（BREATHE_EN == 1）
6	PWM_DUTY2_EN	rw	0	-	亮灭DUTY控制2开关/呼吸灯空周期延时（低8bit）开关 0：亮灭DUTY控制2关闭（BREATHE_EN == 0）/ 呼吸灯满周期延时关闭（BREATHE_EN == 1） 1：亮灭DUTY控制2开启（BREATHE_EN == 0）/ 呼吸灯满周期延时开启（BREATHE_EN == 1）

Bit	Name	RW	Default	RV	Description
5	PWM_DUTY1_EN	rw	0	-	亮灭DUTY控制1开关/呼吸灯最高亮度延时 (高电平灯) 开关 0：亮灭DUTY控制1关闭（BREATHE_EN == 0）/ 呼吸灯最高亮度延时 (高电平灯)关闭（BREATHE_EN == 1） 1：亮灭DUTY控制1开启（BREATHE_EN == 0）/ 呼吸灯最高亮度延时 (高电平灯)开启（BREATHE_EN == 1）
4	PWM_DUTY0_EN	rw	0	-	亮灭DUTY控制0开关/呼吸灯最高亮度延时 (低电平灯) 开关 0：亮灭DUTY控制0关闭（BREATHE_EN == 0）/ 呼吸灯最高亮度延时 (低电平灯)关闭（BREATHE_EN == 1） 1：亮灭DUTY控制0开启（BREATHE_EN == 0）/ 呼吸灯最高亮度延时 (低电平灯)开启（BREATHE_EN == 1）
3	PWM1_PRD_SEL	rw	0	-	PWM1亮灭周期选择： 0：PWM1周期为0xff 1：PWM1周期为PWM_DUTY3
2	PWM_OUTPUT_INV	rw	0	-	输出电平取反（周期变色SHIFT_DUTY==0时可用） 0：不取反。 1：取反
1	PWM1_INV	rw	0	0	PWM1输出的LED亮灭波形取反（默认由0开始）， 可以控制亮灭周期是从灯亮开始还是灯灭开始，软件常设为0。 0：PWM1输出的LED亮度波形不取反（灭灯开始的PWM1周期） 1：PWM1输出的LED亮度波形取反（亮灯开始的PWM1周期）
0	PWM0_INV	rw	0	0	PWM0输出的LED亮度波形取反（默认由1开始）， 软件常设为0 0：PWM0输出的LED亮度波形不取反 1：PWM0输出的LED亮度波形取反

18.8.3 PWM_CON2: PWM control register 2

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
7-4	SHIFT_DUTY	rw	0x0	-	LED周期变色（输出电平取反），非呼吸灯 0000：不变色 0001：1个PWM1周期变色 … 1111：15个PWM1周期变色 LED周期变色，呼吸灯PWM_CON2[4]常设为0 0000：不变色 0010：1个周期变色（ 1个周期为：最暗到最亮，再到最暗） 0100：2个周期变色 … 1110：7个周期变色
3-2	IO_DRV_SHIFT_CLK	rw	0x0	-	输出驱动切换时钟个数（PWM0_CLK） 00: 1 01: 2 10: 4 11: 8（干扰少，但输出会延时）
1-0	IO_MAX_DRV	rw	0x0	-	IO 最大输出强度 00: 2.4mA（8mA mos + 120Ω res） 01: 8mA（8mA mos ） 10: 18.4mA（24mA mos + 120Ω res） 11: 24mA（24mA mos ）

18.8.4 PWM_CON3: PWM control register 3

Bit	Name	RW	Default	RV	Description
7	PENDING	r	0	-	亮灭周期结束，呼吸灯周期结束，pengding 置1
6	CLEAR PENDING	w	0	-	清pending 0：无效 1：清pending
5	IE	rw	0	-	PWM LED pending 中断使能。
4	OUT_LOGIC	rw	0	1	输出逻辑， 软件常设为1 0：LED在低电平时为亮（PWM0 | PWM1）。 1：LED在高电平时为亮（PWM0 & PWM1）。
3-0	PWM_PRD_DIVH	rw	0x0	-	pwm1 clk 分频比控制高位 PWM_PRD_DIV[11:8]

18.8.5 PWM_BRI_PRDL: PWM0 brightness period register

Bit	Name	RW	Default	RV	Description
7-0	PWM_BRI_PRD	rw	0xff	-	亮度周期低7bit

18.8.6 PWM_BRI_PRDH: PWM0 brightness period register

Bit	Name	RW	Default	RV	Description
1-0	PWM_BRI_PRD	rw	0xff	-	亮度周期高2bit

18.8.7 PWM_BRI_DUTY0L: PWM0 brightness duty register

Bit	Name	RW	Default	RV	Description
7-0	PWM_BRI_DUTY0	rw	0x0	-	亮度控制低7bit，例： 高电平灯亮 PWM_BRI_DUTY0 == 0； 灯最暗 PWM_BRI_DUTY0 == 1； 灯亮度增强一档 … PWM_BRI_DUTY0 == PWM_BRI_PRD； 灯最亮

18.8.8 PWM_BRI_DUTY0H: PWM0 brightness duty register

Bit	Name	RW	Default	RV	Description
1-0	PWM_BRI_DUTY0	rw	0x0	-	亮度控制高2bit，例： 高电平灯亮

18.8.9 PWM_BRI_DUTY1L: PWM0 brightness duty register

Bit	Name	RW	Default	RV	Description
7-0	PWM_BRI_DUTY1	rw	0x0	-	亮度控制低7bit，例： 低电平灯亮 PWM_BRI_DUTY0 == 0； 灯最暗 PWM_BRI_DUTY0 == 1； 灯亮度增强一档 … PWM_BRI_DUTY0 == PWM_BRI_PRD； 灯最亮

18.8.10 PWM_BRI_DUTY1H: PWM0 brightness duty register

Bit	Name	RW	Default	RV	Description
1-0	PWM_BRI_DUTY1	rw	0x0		亮度控制高2bit， 例： 低电平灯亮

18.8.11 PWM_ PRD_DIVL: PWM1 period divider register

Bit	Name	RW	Default	RV	Description
7-0	PWM_PRD_DIV	rw	0x0		PWM时钟的分频因子设置。

PWM_ PRD_DIV ： pwm1_clk分频比控制器，12bit

PWM_ PRD_DIV[11:0] = {P3_PWM_CON3[3:0],PWM_PRD_DIVL[7:0]}；

0x000: div = 1;

0x001: div = 2;

… …

0xfff: div = 4096;

18.8.12 PWM_DUTY0: PWM duty0 register

Bit	Name	RW	Default	RV	Description
7-0	PWM_DUTY0	rw	0x0		BREATHE_EN == 0 & PWM_DUTY0_EN == 1: 亮灭DUTY控制1 BREATHE_EN == 1 & PWM_DUTY0_EN == 1: 呼吸灯最高亮度延时(低电平灯)

18.8.13 PWM_DUTY1: PWM duty1 register

Bit	Name	RW	Default	RV	Description
7-0	PWM_DUTY1	rw	0x0		BREATHE_EN == 0 & PWM_DUTY1_EN == 1: 亮灭DUTY控制1 BREATHE_EN == 1 & PWM_DUTY1_EN == 1: 呼吸灯最高亮度延时(高电平灯)

18.8.14 PWM_DUTY2: PWM duty2 register

Bit	Name	RW	Default	RV	Description
7-0	PWM_DUTY2	rw	0x0		BREATHE_EN == 0 & PWM_DUTY2_EN == 1: 亮灭DUTY控制2 BREATHE_EN == 1 & PWM_DUTY2_EN == 1: 呼吸灯灭灯延时低8bit

18.8.15 PWM_DUTY3: PWM duty3 register

Bit	Name	RW	Default	RV	Description
7-0	PWM_DUTY3	rw	0x0		BREATHE_EN == 0 & PWM1_PRD_SEL == 0 & PWM_DUTY3_EN == 1: 亮灭DUTY控制3 BREATHE_EN == 0 & PWM1_PRD_SEL == 1: 亮灭PWM1周期PWM1_PRD BREATHE_EN == 1 & PWM_DUTY3_EN == 1: 呼吸灯灭灯延时高8bit

18.8.16 PWM_CON4 : counter_cnt read control

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
7-6	RESERVED	-	-	-	预留
5	RD_DONE	r	1		读取完成标记位。读流程如下： u16 pwm_cnt_read (void){ u16 cnt; PWM_CON4 ### 1.8.
4	RD_KICK	w	0		读取rtc_dat
3-0	RESERVED	-	-	-	预留

18.8.17 CNT_RD : counter_cnt read data

Bit	Name	RW	Default	RV	Description
17	BREATHE BRIGHTNESS	r	x		LED breathe brightness： 0： LED breath brightness decrease 1： LED breath brightness increase
16	LED LEVEL	r	x		LED output level： 0： hign level ON LED period 1： low level ON LED period
15:8	COUNTER_CNT	r	0x0		counter_cnt[15:8]:only use in breathe mode, meaningless in other case
7-0	COUNTER_CNT	r	0x0		counter_cnt[7:0]:use in all case

下图是呼吸灯状态下，CNT_RD的图解：

1. PWM_DUTY0_EN == 1； PWM_DUTY1_EN == 1；PWM_DUTY2_EN == 1；

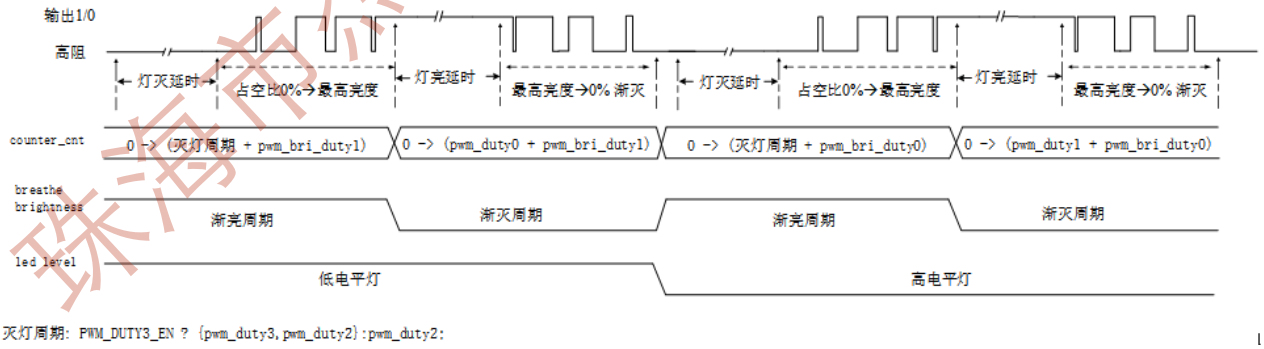
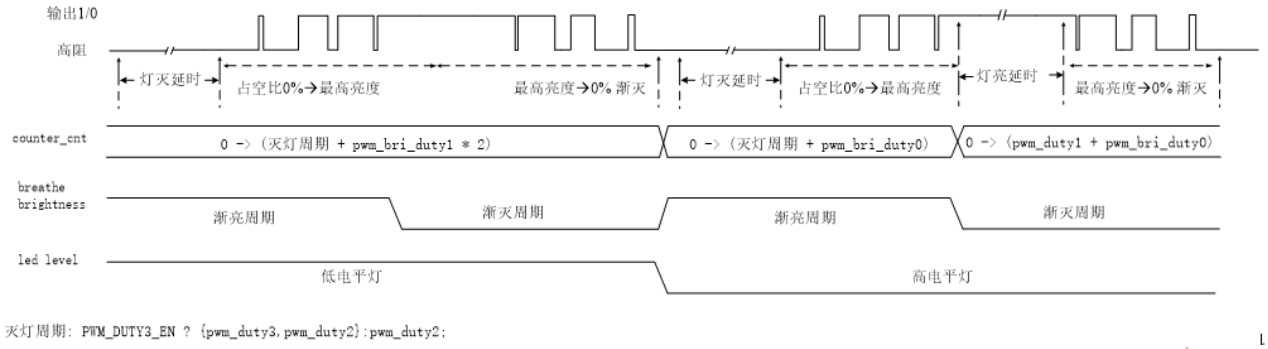


图 1-5 CNT_RD亮灭延时全开示意图

2. PWM_DUTY0_EN == 0； PWM_DUTY1_EN == 1；PWM_DUTY2_EN == 1；



灭灯周期: PWM_DUTY3_EN ? {pwm_duty3, pwm_duty2} : pwm_duty2;

图 1-6 CNT_RD亮灭延时半开示意图

3. PWM_DUTY0_EN == 0； PWM_DUTY1_EN == 0；PWM_DUTY2_EN == 0；

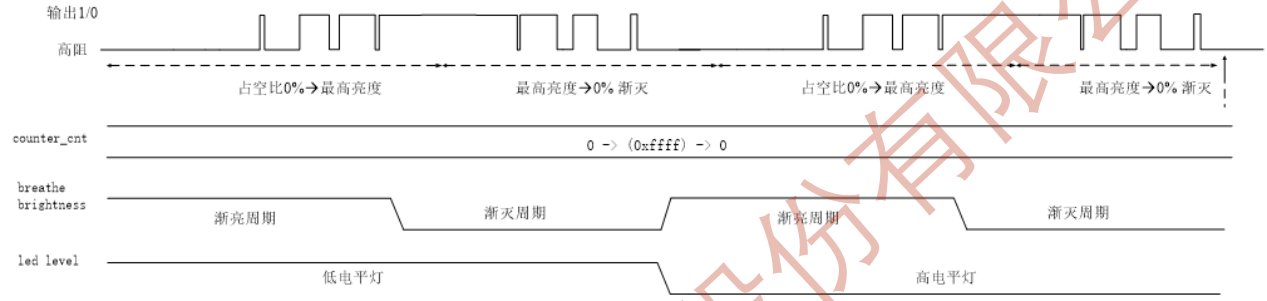


图 1-7 CNT_RD亮灭延时关闭示意图

18.8.18 CNT_DEC : counter_cnt decrease

Bit	Name	RW	Default	RV	Description
7-0	CNT DEC	rw	0x0		counter_cnt decrease in clocks below: 0: function disable 1: counter_cnt stop 1 clk every 2clk 2: counter_cnt stop 1 clk every 3clk 3: counter_cnt stop 1 clk every 4clk …

18.8.19 CNT_SYNC : counter_cnt sync

Bit	Name	RW	Default	RV	Description
7-0	CNT SYNC	w	x		counter_cnt sync

use in more than one chip PWM LED period sync.

for example in TWS earphone:

L earphone clock frequency is 30k.

R earphone clock frequency is 32k.

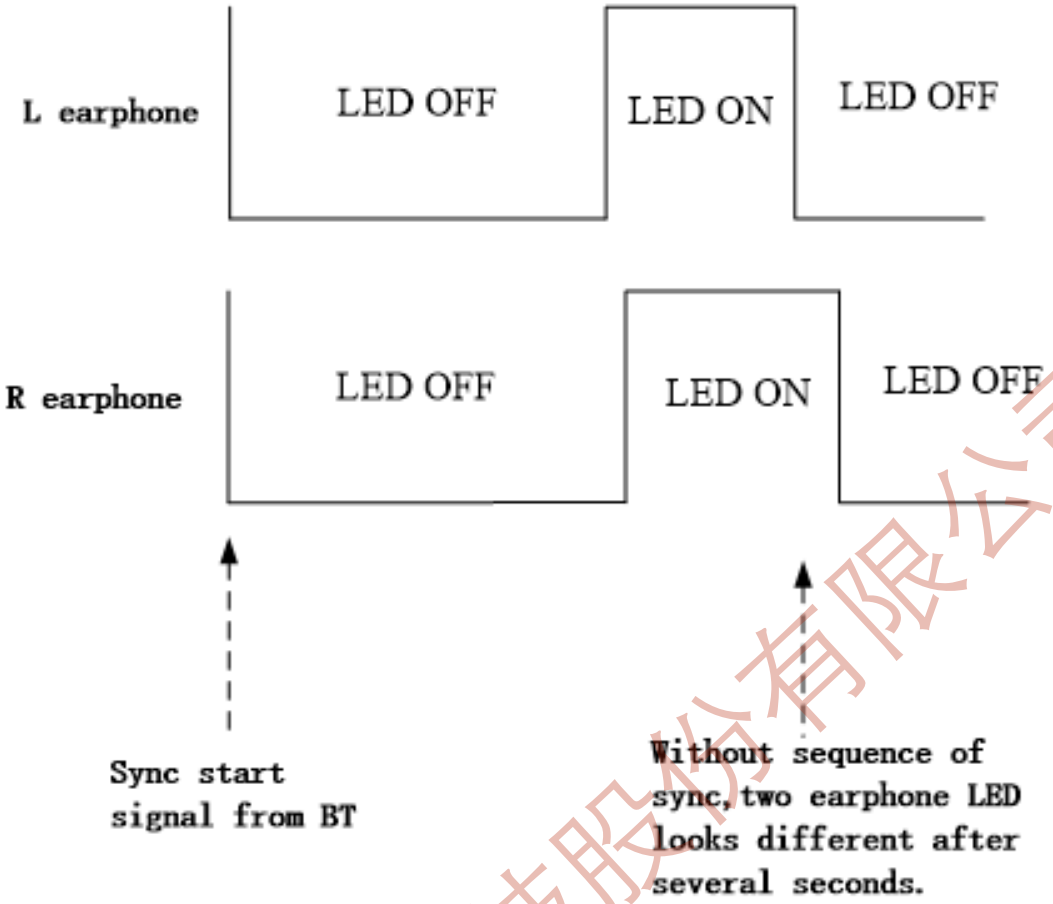


图 1-8 TWS LED without SYNC

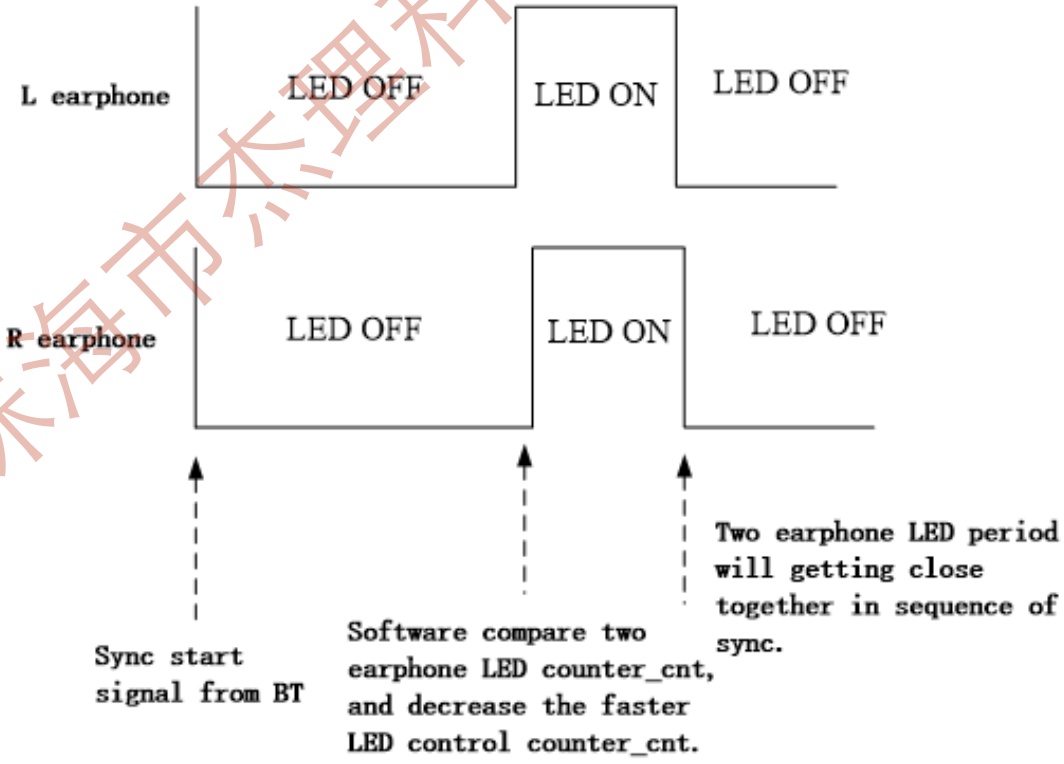


图 1-9 TWS LED SYNC start

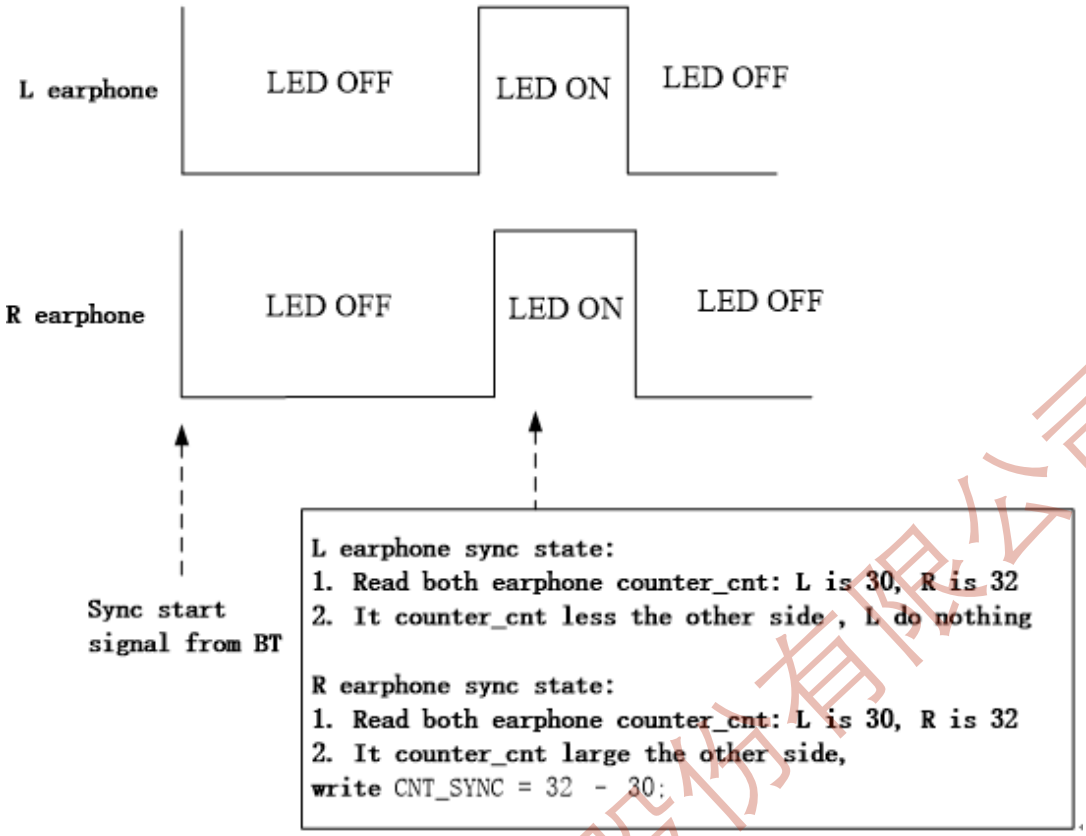


图 1-10 TWS LED SYNC strategy

19 QDEC

19.1 概述

QDEC（quadrature decoder）是一个用于正交编码器检测的模块，支持两线输入。兼容4 phase 、2 phase 、1 phase 三种模式， 通常旋转编码器使用 1phase 即可，鼠标滑轮等使用了 2 phase，分辨率要求更高的可使用 4 phase 模式；同时本模块自带定时器，可独立定时检测。

19.2 特殊注意点

19.3 数字模块控制寄存器

19.3.1 QDECx_CON: QDEC control register

Bit	Name	RW	Default	RV	Description
15-11	RESERVED	-	-	-	预留
10	QDEC_INT_MODE	rw	0	0	1： 根据 QDEC_SMP 配置产生定时标志与中断 0： 当检测到输入信号发生变化时产生标志与中断
9-8	QDEC_MODE	rw	0	0	QDEC_MODE 0 0：1 phase 模式，普通旋转编码器 0 1：2 phase 模式，通常鼠标滑轮 1 0：4 phase 模式，分辨率最高
7	PND	r	0	0	PND，只读。写入无效
6	CPND	w	0	0	CPND写入1清除PND，并且计步器结果输送到寄存器 QDECx_DAT， QDECx_DBE 后重新计数； 读取 QDECx_DAT， QDECx_DBE 前需要把CPND写入1； 读出永远为0
5-2	QDEC_SPND	rw	0x0	0x0	QDEC_SPD，采样时钟设置 sr_clk = lsb_clk / (2^QDEC_SPND+1) 采样时钟要求，是编码器变化输出信号发生变化最短时间频率的8 倍以上，一般 8 - 32 倍比较合适； 常见编码器可设置在介于2k - 500Hz之间；
1	QDEC_POL	rw	0	0	1: 把输入信号极性反转，会影响输入信号第一个步数的识别（仅影响1phase和2phase模式，不影响4phase模式）
0	EN	rw	0	0	QDEC_EN 0：模块关闭 1：模块打开

19.3.2 QDECx_SMP: QDEC sample register

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
7-0	QDEC_SMP	rw	0x0	0x0	当 QDEC_INT_MODE 为 1时， 定时器时间为 Tsmp = (QDEC_SMP+1)/sr_clk

19.3.3 QDECx_DAT: QDEC control register

Bit	Name	RW	Default	RV	Description
7-0	QDEC_DAT	r	0x0	0x0	编码器步数计算器输出结果，此寄存器为8位有符号数，表示编码器正反的步数。CPND写入1清除PND时，步数计算器输出到QDEC_DAT 后自动清0 重新计算。通常读取QDEC_DAT 前 需 CPND写入1 获取计步器结果；如果长时间不读取数据清0，会导致步数计算器溢出无法正确识别步数跳转； 需要注意内核总线同步问题， 想在CPND写入1后马上读结果，需要要加asm（“csync”）；

19.3.4 QDECx_DBE: QDEC double tansition error

Bit	Name	RW	Default	RV	Description
7-0	QDEC_DBE	r	0x0	0x0	编码器步数错误计算器输出结果； 此寄存器为8位无符号数记录检测到编码; 器输入信号发生非连续跳步行进次数， 出现这些情况时，会影响计步器输出结果QDEC_DAT的准确性；通常 QDEC_SPND 设置采样率过慢导致跟不上信号变化，或者过快受编码器模块输出信号毛刺影响。当然编码器出现损坏，输出信号有时不满足正交要求也可能导致问题发生。同样CPND写入1清除PND 时，步数错误计算器结果输出到QDEC_DBE 后自动清0 重新计算，读取QDEC_DBE前需要CPND写入1； 需要注意内核总线同步问题， 想在CPND写入1后马上读结果，需要要加asm（“csync”）；

19.4 DEMO CODE

```
void get_qdec_res( void)
{
    char qdec_cnt;
    unsigned char qdec_error_cnt;
    JL_QDEC0->CON0 |= BIT(6);
    asm(“csync”);
    qdec_cnt = JL_QDEC0->DAT;
    qdec_error_cnt = JL_QDEC0->DBE;
}
```

20 RAND64

20.1 概述

RAND64是一个产生64位伪随机序列的模块，它由片内RC振荡器驱动，RC振荡器振荡频率的不稳定性进一步确保了生成序列的随机性。

由于片内RC振荡器的振荡频率大约为250KHz，即每个周期4uS，因此RAND64生成的序列约每4uS自动更新一次。若软件有读取RAND64寄存器（RAND64H或RAND64L）的动作，则生成序列将立即更新，以确保每次读取均能获得不同的序列。

20.2 控制寄存器

寄存器列表	宽度	读写	复位值
RAND64H	32bit	只读	不确定
RAND64L	32bit	只读	不确定

21 SPI

21.1 概述

SPI接口是一个标准的遵守SPI协议的串行通讯接口，在上面传输的数据以Byte（8bit）为最小单位，且永远是MSB在前。

SPI接口支持主机和从机两种模式。

主机：SPI 接口时钟由本机产生，提供给片外SPI设备使用。

从机：SPI接口时钟由片外 SPI设备产生，提供给本机使用。

工作于主机模式时，SPI接口的驱动时钟可配置，范围为 系统时钟~系统时钟/256。

工作于从机模式时，SPI接口的驱动时钟频率无特殊要求，但数据速率需要进行限制，否则易出现接收缓冲覆盖错误，在系统不繁忙情况下可以接近系统时钟速率。

SPI接口可独立地选择在SPI时钟的上升沿或下降沿更新数据，在SPI时钟的上升沿或下降沿采样数据。

SPI接口支持单向（Unidirection）和双向（Bidirection）模式

单向模式：使用SPICK和SPIDAT两组连线，其中SPIDAT为双向信号线，同一时刻数据只能单方向传输。

双向模式：使用SPICK，SPIDI和SPIDO三组连线，同一时刻数据双向传输。但DMA不支持双向数据传输，当在本模式下使能DMA时，也只有一个方向的数据能通过DMA和系统进行传输。

SPI单向模式支持1bit data、2bit data和4bit data模式，即：

1bit data模式：串行数据通过一根DAT线传输，一个字节数据需8个SPI时钟。

2bit data模式：串行数据通过两根DAT线传输，一个字节数据需4个SPI时钟。

4bit data 模式：串行数据通过四根DAT线传输，一个字节数据需2个SPI时钟。

SPI双向模式只支持1bit data模式，即：

1bit data模式：串行数据通过两根DAT线进行双向传输，一个字节数据需8个SPI时钟。

SPI接口在发送方向上为单缓冲，在上一次传输未完成之前，不可开始下一次传输。在接收方向上为双缓冲，如果在下一次传输完成时CPU还未取走本次的接收数据，那么本次的接收数据将会丢失。

SPI接口的发送寄存器和接收寄存器在物理上是分开的，但在逻辑上它们一起称为SPIBUF寄存器，使用相同的SFR地址。当写这个SFR地址时，写入至发送寄存器。当读这个SFR地址时，从接收寄存器读出。

SPI传输支持由CPU直接驱动，写SPIBUF的动作将启动一次Byte传输。

SPI传输也支持DMA操作，但DMA操作永远是单方向的，即一次DMA要么是发送一包数据，要么是接收一包数据，不能同时发送并且接收一包数据，即使在双向模式下也是这样。每次DMA操作支持的数据量为1-(2^32-1)Byte。写SPIADR的动作将启动一次DMA传输。

21.2 特殊注意点

【注意】：该模块所有说明及配置仅作为SPIx模块使用，不适用到SFC中

21.3 数字模块控制寄存器

21.3.1 SPIx_CON: SPIx control register 0

Bit	Name	RW	Default	RV	Description
31-22	RESERVED	-	-	-	预留
21	OF_PND	r	0	-	dma fifo overflow中断，当spi做从机dma接收时，fifo满且dma响应慢， 而新接收的数据到来时导致fifo发出溢出就会产生of_pnd 【注意】：仅做debug使用！！！
20	OF_PND_CLR	w	0	-	清除of_pnd，写1清零
19	OF_IE	rw	0	1	of_pnd使能位
18	UF_PND	r	0	-	dma fifo underflow中断，当spi做从机dma发送时，fifo空且dma响应慢， 而主机端请求新数据时导致fifo读空就会产生uf_pnd 【注意】：仅做debug使用
17	UF_PND_CLR	w	0	-	清除uf_pnd，写1清零
16	UF_IE	rw	0	1	uf_pnd使能位
15	PND	r	0	0	中断请求标志，当1Byte传输完成或DMA传输完成时会被硬件置1。 有3种方法清除此标志 向PCLR写入‘1’ 写SPIBUF寄存器来启动一次传输 写SPICNT寄存器来启动一次DMA
14	CPND	w	0	1	软件在此位写入‘1’将清除PND中断请求标志。
13	IE	rw	0	1	SPI 中断使能 0: 禁止SPI中断 1: 允许中断允许
12	DIR	rw	0	0	在单向模式或DMA操作时设置传输的方向 0: 发送数据 1: 接收数据
11-10	DATW	r	0x0	0x0	SPI数据宽度设置 00: 1bit数据宽度 01: 2bit 数据宽度 10: 4bit数据宽度 11：NA，不可设置为此项
9-8	RESERVED	-	-	-	预留

Bit	Name	RW	Default	RV	Description
7	CSID	rw	0	0	SPICS信号极性选择 0: SPICS空闲时为0电平 1: SPICS空闲时为1电平
6	CKID	rw	0	0	SPICK信号极性选择 0: SPICK空闲时为0电平 1: SPICK空闲时为1电平
5	UE	rw	0	0	更新数据边沿选择 0: 在SPICK的上升沿更新数据 1: 在SPICK的下降沿更新数据
4	SE	rw	0	0	采样数据边沿选择 0: 在SPICK的上升沿采样数据 1: 在SPICK的下降沿采样数据
3	BDIR	rw	0	0	单向/双向模式选择 0: 单向模式，数据单向传输，同一时刻只能发送或者接收数据。 数据传输方向因收发而改变，所以由硬件控制，不受写IO口DIR影响。 1: 双向模式，数据双向传输，同时收发数据，但DMA只支持一个方向的数据传输。 数据传输方向设置后不改变，所以由软件控制，通过写IO口DIR控制。
2	CSE	rw	0	1	SPICS信号使能，always set为1
1	SLAVE	rw	0	0	从机模式
0	SPIEN	rw	0	0	SPI接口使能 0: 关闭SPI接口 1: 打开SPI接口

21.3.2 SPIx_BAUD: SPIx baudrate setting register

Bit	Name	RW	Default	RV	Description
7-0	SPI_BAUD	rw	0x0	0x0	SPI主机时钟设置寄存器 SPICK = system clock / (SPIBAUD + 1)

21.3.3 SPIx_BUF: SPIx buffer register

Bit	Name	RW	Default	RV	Description
7-0	SPI_BUF	rw	0x0	0x0	发送寄存器和接收寄存器共用此SFR地址。 写入至发送寄存器，从接收寄存器读出。

21.3.4 SPIx_ADR: SPIx DMA address register

Bit	Name	RW	Default	RV	Description
31-0	SPI_ADR	rw	0x0	0x0	SPI DMA起始地址寄存器，只写，读出为不确定值

21.3.5 SPIx_CNT: SPIx DMA counter register

Bit	Name	RW	Default	RV	Description
31-0	SPI_CNT	rw	0x0	0x0	SPI DMA计数寄存器，读出为当下剩余读写数量， 此寄存器用于设置DMA操作的数目（按Byte计）并启动DMA传输。 如，需启动一次512Byte的DMA传输，写入0x0200，此写入动作将启动本次传输。

21.3.6 SPIx_CON1: SPIx control1 register

Bit	Name	RW	Default	RV	Description
31-2	RESERVED	-	-	-	预留
4	SPI_9B_DATA	rw	0	-	在9bit模式下，发送命令时，为0； 发送参数/数据时，为1。
3	SPI_9B_MODE	rw	0	-	0：关闭9bit模式 1：打开9bit模式
2	MIX_MODE	rw	0	-	3wire模式混合接线
1-0	SPI_BIT_MODE	rw	0	0	设置spi输入输出位流模式，默认0为标准格式 0：【7,6,5,4,3,2,1,0】 1：【0,1,2,3,4,5,6,7】 2：【3,2,1,0,7,6,5,4】 3：【4,5,6,7,0,1,2,3】

21.4 传输波形

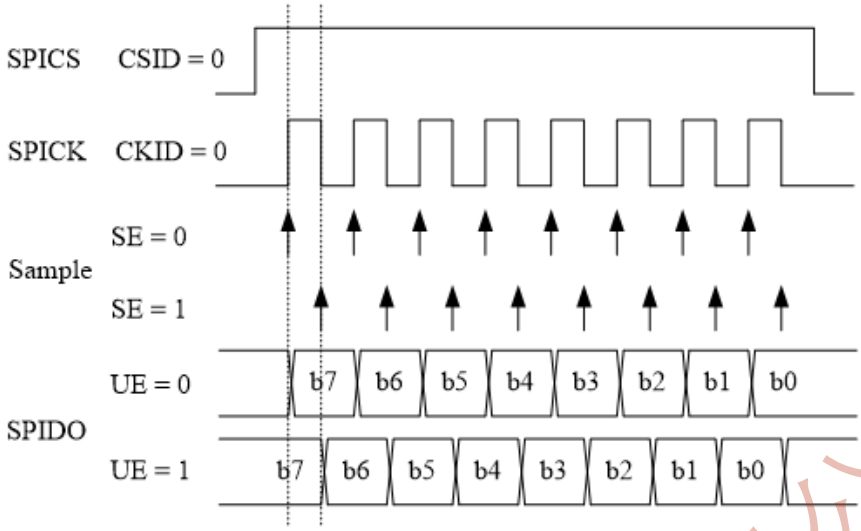


图 1-1 传输波形1

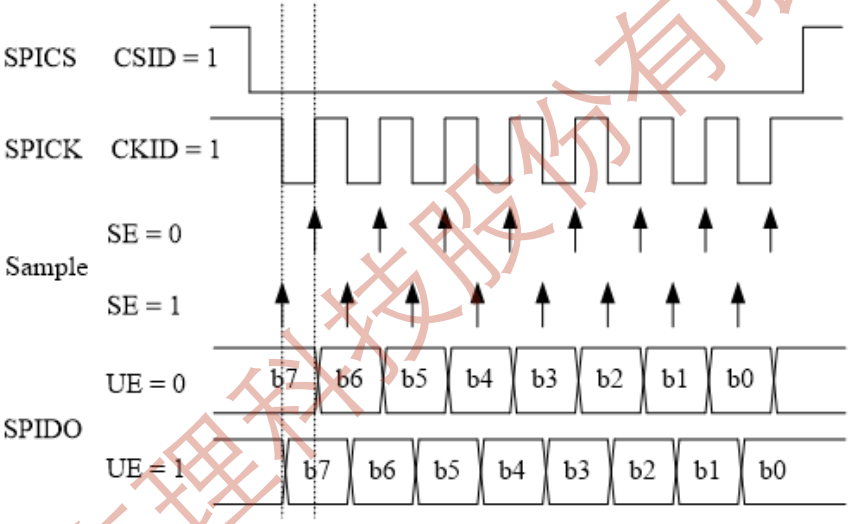


图 1-2 传输波形2

22 32位定时器(TIMER32)

22.1 概述

Timer16是一个集合了定时/计数/捕获功能于一体的多功能32位定时器。它的驱动源可以选择片内时钟或片外信号。它带有一个可配置的最高达64的异步预分频器，用于扩展定时时间或片外信号的最高频率。它还具有上升沿/下降沿捕获功能，可以方便的对片外信号的高电平/低电平宽度进行测量。

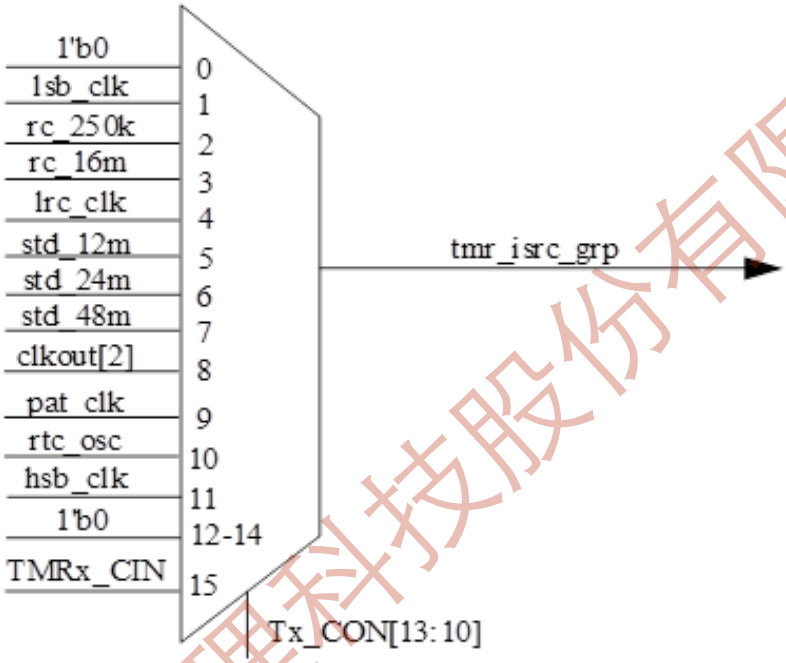


图 1-1 TIMER时钟源选择示意图

22.2 控制寄存器

寄存器列表	TIMER0	TIMER1	TIMER2	TIMER3		
Tx_CON	T0_CON	T1_CON	T2_CON	T3_CON		
Tx_CNT	T0_CNT	T1_CNT	T2_CNT	T3_CNT		
Tx_PRD	T0_PRD	T1_PRD	T2_PRD	T3_PRD		
Tx_PWM	T0_PWM	T1_PWM	T2_PWM	T3_PWM		

22.3 寄存器说明

22.3.1 Tx_CON： timer x control register

Bit	Name	RW	Default	RV	Description
31-20	RESERVED	-	-	-	预留
19	RMT_TX	Rw	0	-	红外发送模式，需要配合UDMA使用
18	RMT_RX	Rw	0	-	红外接收模式，需要配合UDMA使用
16	DUAL_EDGE_EN	rw	0	-	DUAL_EDGE_EN:双边沿捕获模式， 当MODE=2或3时， 使能该位即变成双边沿捕获模式， 在上沿和下沿会TxCNT的值写入TxPR。
15	PND	r	0	-	PND： 中断请求标志， 当timer溢出或产生捕获动作时会被硬件置1， 需要由软件清0。
14	PCLR	w	x	-	PCLR： 软件在此位写入‘1’将清除PND中断请求标志。
13-10	SSEL	rw	0x0	-	SSEL3-0：timer驱动源选择见图1-2 【注意】：选中的时钟需要使用上面PSET分频到(lsb_clk/2)以下！
9	PWM_INV	rw	0	-	PWM_INV：PWM信号输出反向。
8	PWM_EN	rw	0	-	PWM_EN： PWM信号输出使能。 此位置1后， 相应IO口的功能将会被PWM信号输出替代。
7-4	PSET	rw	0x0	-	PSET3-0：预分频选择位 0000：预分频1 0001：预分频4 0010：预分频16 0011：预分频64 0100：预分频12 *0101：预分频42* 0110：预分频162 *0111：预分频642* 1000：预分频1256 *1001：预分频4256* 1010：预分频16256 *1011：预分频64256* 1100：预分频12256 1101：预分频42256 1110：预分频162256 1111：预分频642256
3-2	CSEL	rw	0x0	-	捕获模式端口选择： 0：IO mux in（crossbar） 1：IRFLT_OUT

Bit	Name	RW	Default	RV	Description
1-0	MODE	rw	0x0	-	MODE1-0：工作模式选择 00： timer关闭； 01： 定时/计数模式； 10： IO口上升沿捕获模式（当IO上升沿到来时，把TxCNT的值捕捉到TxPR中）； 11： IO口下降沿捕获模式（当IO下降沿到来时，把TxCNT的值捕捉到TxPR中）。

22.3.2 Tx_CNT： timer x counter register

Bit	Name	RW	Default	RV	Description
31-0	Tx_CNT	rw	x	-	Timer32的计数寄存器

22.3.3 Tx_PR： timer x period register

Bit	Name	RW	Default	RV	Description
31-0	Tx_PR	rw	x	-	Timer32的周期寄存器

在定时/计数模式下，当Tx_CNT == Tx_PR时，Tx_CNT会被清0。

在上升沿/下降沿捕获模式下，TxPR是作为捕获寄存器使用的，当捕获发生时，Tx_CNT的值会被复制到Tx_PR中。而此时Tx_CNT自由的由0->（232-1）->0计数，不会和Tx_PR进行比较清0。

22.3.4 Tx_PWM： timer x PWM register

Bit	Name	RW	Default	RV	Description
31-0	Tx_PWM	rw	x	-	Timer32的PWM设置寄存器

在PWM模式下，此寄存器的值决定PWM输出的占空比。占空比N的计算公式如下：

$$N = (Tx\_PWM / Tx\_PR) * 100\%$$

此寄存器带有缓冲，写此寄存器的动作不会导致不同步状态产生的PWM波形占空比瞬间过大或过小的问题。只有在MODE=01的情况下才会输出PWM。

22.4 红外遥控模式（RMT）

22.4.1 红外接收

通过UDMA 与 TIMER 的结合使用，实现RMT 接收功能。

该模式下TIMER设置为双边沿捕获模式，TIMER->PWM寄存器用于设置超时时间，TIMER->PRD 寄存器用于接收数据buff，接收的数据通过TIMER 捕获触发UDMA搬运到memory里。

配置步骤：

a. 配置timer 捕获输入 io, 并且通过 irdflt 模块滤波；

b. 配置timer 计算器时钟，用于记录脉宽宽度；

c. 配置TIMER->PWM 接收超时时间, TIMER->CNT=0；当TIMER->CNT 计数等于 TIMER->PWM 时硬件自动把TIMER->CNT清理0，并且产生标志位；

d. 配置timer 为双边沿捕获模式，并且 rmt_mode_rx 配置为 1；

e. 当捕获到边沿时，把前面的电平宽度和电平值写到 TIMER->PRD ={TIMER->CNT, 电平} 并且可以配置TIMER 捕获触发UDMA把更新好的TIMER->PRD 写到 memory；第一个捕获的电平数据视为无效数据；根据需要可以把 TIMER->PRD按照低 8bit/16bit/32bit写到memory. 同时硬件把 TIMER->CNT 自动清0；

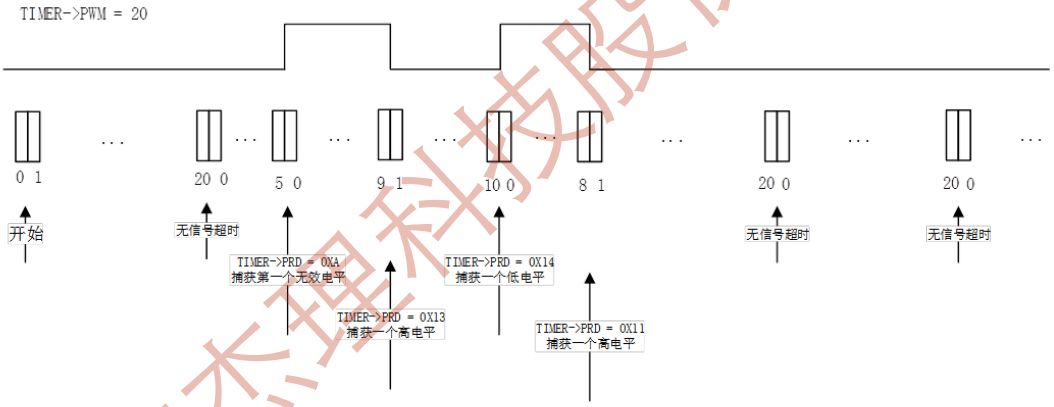
f. 当接收到一个超时信号后，视为红外接收结束。

g. 捕获的电平可以通过UDMA 写到memroy：

当按照8位数据写时,TIMER->PWM 超时值不应该超过0X7F；超时机制同时也是一种数据保护机制，当红外脉冲长度大于超时值0X7F时，生成超时信号表明数据溢出错误；

当按照16位数据写时,TIMER->PWM 超时值不应该超过0X7FFF；

当按照32位数据写时,TIMER->PWM 超时值不应该超过0X7FFFFFFF；



22.4.2 红外发送

通过UDMA 与 TIMER 的结合使用，实现RMT 发送功能。

该模式下TIMER设置为定时模式，TIMER->PWM寄存器用于做发送数据buff，通过TIMER定时触发UDMA把要发送的数据从memory搬运到TIMER->PWM寄存器。

配置步骤：

a. 配置timer PWM 输出 io, 做RMT发送，关闭pwm_en

b. 配置timer 计算器时钟，用于设置脉宽宽度

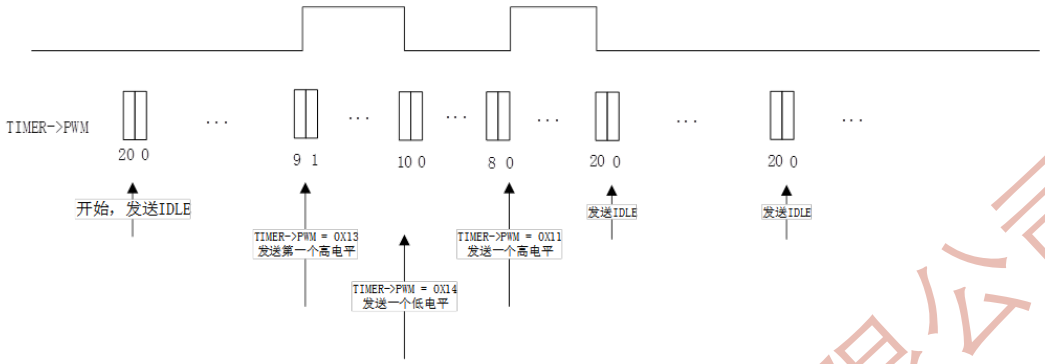
c. 配置TIMER->PWM 作为发送数据BUF, 第一次配置为{发送IDEL时间，IDEL电平}

d. 配置TIMER->PRD = 0，清除发送时间，同时配置TIMER->CNT = 0；

e. 配置timer 为定时器模式，并且 rmt_mode_tx 配置为 1

f. 当CNT=PRD时，把TIMER->PWM[0] 作为发送电平从TIMER PWM 口发出,并更新定时时间为TIMER->PRD =TIMER->PWM[31:1]，TIMER->CNT 自动清零重新计数; 同时可以配置TIMER 定时触发UDMA把 memory数据跟新到TIMER->PWM 用于下一次发送；

g. 发送完数据后需要再发送一个IDEL电平。



23 UDMA

23.1 概述

基础直接存储器存取 (Ultimate Direct Memory Access, UDMA) 用来提供在外设和存储器之间或者存储器和存储器之间的高速数据传输。无须CPU干预，数据都可以通过DMA进行快速地传输。这样节省的CPU资源可供其它操作使用。

23.2 主要特性

UDMA有以下特性：

1. 通道个数可配，每个通道支持8个外设源。
2. 4级可配优先级：最高、高、中等、低。
3. 相同优先级通道的仲裁方式：轮询仲裁。
4. 源和目标数据区的数据宽度各自可配（8bit/16bit/32bit）。
5. 每个通道都有4个事件标志：单次传输/每次循环传输一半、单次传输/每次循环传输完成、总循环传输完成以及配置异常；可用于中断。
6. 传输数据量可编程；最大传输数据量：65535。
7. 在源地址和目标地址符合所配的数据宽度时，可以设定任意的传输数目。
8. 支持指针增量模式/固定地址模式。
9. 支持循环传输模式/正常传输模式。循环模式支持最大循环次数：65535。
10. 当传输出错时，相应通道使能位将被硬件清零。
11. 每个通道支持软件触发的DMA传输，可用于M2M传输。
12. 支持位封装模式（M2M）/零扩展模式/符号扩展模式。

23.3 DMA功能说明

23.3.1 结构图

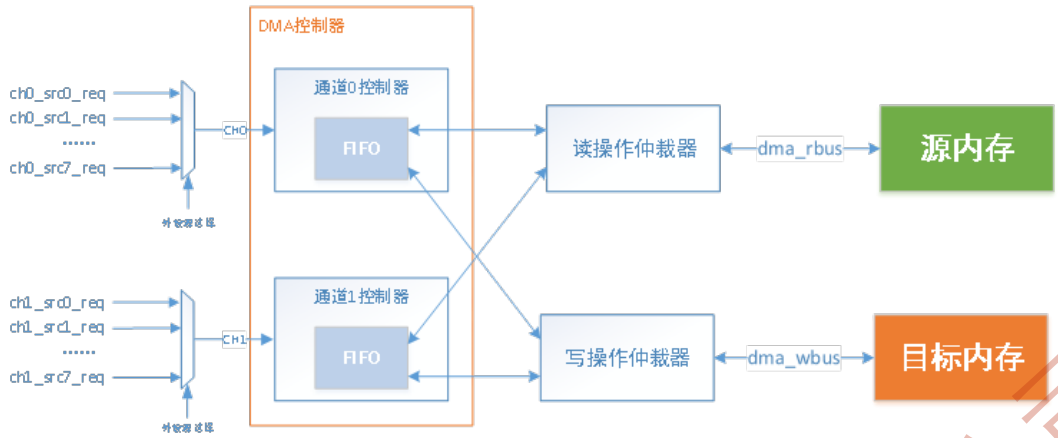


图 1-1 UDMA结构图

图1-1从左到右显示了UDMA中DMA传输所经过的各个过程：

1. 每组dma通道有8个外设源，通过寄存器选择出一个外设源；
2. 寄存器配置传输通道的信息，包括源地址CHX_SADR、目标地址CHX_DADR、控制寄存器DMA_CON0和控制寄存器DMA_CON1；
3. 根据各个通道的DMA配置情况，在通道使能的情况下，开始P2M/M2P/P2P/M2M的传输；
4. 每组DMA通道将产生一组cbn形式的读总线和写总线。所有读总线和写总线将进入仲裁器进行仲裁，最终得到一组读总线和一组写总线。
5. 每组DMA通道在得到仲裁许可之后，读总线将从源内存读取数据并存入通道的FIFO，在外设模式下DMA控制器根据外设的请求，或在M2M模式下DMA自身产生的请求，往目的内存发出写总线信息，完成写操作。

23.3.2 通道信号选择

每个DMA通道可硬件连接8个外设源。一个DMA通道同时只能选择一个外设源进行传输，可通过寄存器DMA_CON0的DMA_SEL位选择对应外设源。

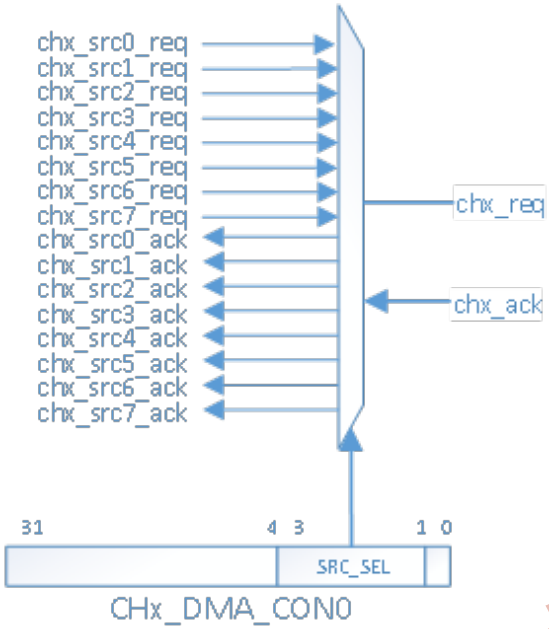


图 1-2 通道信号选择

附录为本项目DMA硬件映射。

23.3.3 通道仲裁

UDMA中提供了4个可配的优先级：特高优先级、高优先级、中优先级和低优先级。相同优先级下的不同通道，将根据轮询规则进行仲裁，保证不会有某个通道长时间占据总线的情况出现。

23.3.4 指针递增

根据DMA_CON0的SCR_MODE和DCR_MODE，可以分别控制源地址和目标地址是否自动向后递增或保持常量。若设置为地址自增，则在每一次完成传输之后都将自动增加所设数据位宽对应的增量。若数据位宽为8，则地址增量为1；若数据位宽为16，则地址增量为2；若数据位宽为32，则地址增量为4。

23.3.5 循环模式

循环模式可用于处理循环缓冲区和连续数据流（例如 ADC 扫描模式）。可以使用DMA_CON0中的CYC_MODE来使能循环模式。

循环模式分为无线循环和可配次数循环。无线循环功能需要配置DMA_CON0的INF_MODE为1，配置后会一直不断循环执行DMA传输。可配次数循环功能需要将DMA_CON0的INF_MODE配为0，并且配置DMA_CON1中的CYC_TIME为需要循环的次数。当循环次数达到预设的值时，DMA传输停止。

当激活循环模式时，要传输的数据项的数目在数据流配置阶段自动用设置的初始值进行加载，并继续响应 DMA 请求。

23.3.6 传输方式

外设传输模式（P2M/M2P/P2P）

使能这种模式时，外设的每一次请求，数据流会启动数据源到FIFO的传输，当FIFO饱和或者读到的数据已满足目标内存位宽时，将会启动FIFO到目标内存的传输。

如果传输次数达到预定数量，则传输终止，并且通道的DMA_EN会有硬件自动清零。

存储器传输模式（M2M）

DMA通道在没有外设请求触发的情况下，可以用于进行存储器到存储器之间的传输。

通过将配置DMA_CON0的DMA_EN和DMA_TYPE来使能数据流时，数据流会立即开始填充FIFO。在FIFO中已经存入数据后，FIFO的内容便会移出，并存储到目标中。

如果传输次数达到预定数量，则传输终止，并且通道的DMA_EN会有硬件自动清零。

23.3.7 可编程数据位宽、扩展模式、字节序

在源地址和目标地址配置符合对齐要求的情况下（即32位数据位宽要求地址以word为单位对齐，16位数据位宽要求地址以half-word为单位对齐，8为数据位宽要求地址以byte为单位对齐），源内存地址、目标内存地址以及传输数量可以任意配置，DMA控制器会根据配置的扩展模式，自动对数据进行扩展并更新地址，以实现正确读写内存的目的。扩展模式可由EXT_MODE配置。

位封装模式（M2M）

只有M2M支持位封装模式。在位封装模式下，DMA_SIZE的单位为“byte”，即从源内存读取DMA_SIZE个byte的数据写到目标地址。可以设置任意源/目标位宽组合，任意传输byte数目。DMA从源内存读取数据后，按先后顺序将读到的数据以小端模式排序写入目标内存。

表1-1列举了在位封装模式下，不同数据位宽和传输数量组合情况下dma控制器对源内存和目标内存的读写情况。

表 1-1 位封装模式（M2M）

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
8	8	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，将B0写入@d_0x0 2、 从@s_0x1读取B1，将B1写入@d_0x1 3、 从@s_0x2读取B2，将B2写入@d_0x2 4、 从@s_0x3读取B3，将B3写入@d_0x3	@d_0x0/B0 @d_0x1/B1 @d_0x2/B2 @d_0x3/B3
8	16	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，从@s_0x1读取B1，将B1B0写入@d_0x0 2、 从@s_0x2读取B2，从@s_0x3读取B3，将B3B2写入@d_0x2	@d_0x0/B1B0 @d_0x2/B3B2

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
8	32	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，从@s_0x1读取B1， 从@s_0x2读取B2，从@s_0x3读取B3， 将B3B2B1B0写入@d_0x0	@d_0x0/B3B2B1B0
16	8	4	@s_0x0/B1B0 @s_0x2/B3B2	1、 从@s_0x0读取B1B0，将B0写入@d_0x0 2、 从@s_0x0读取B1B0，将B1写入@d_0x1 3、 从@s_0x2读取B3B2，将B2写入@d_0x2 4、 从@s_0x2读取B3B2，将B3写入@d_0x3	@d_0x0/B0 @d_0x1/B1 @d_0x2/B2 @d_0x3/B3
16	16	4	@s_0x0/B1B0 @s_0x2/B3B2	1、 从@s_0x0读取B1B0，将B1B0写入@d_0x0 2、 从@s_0x2读取B3B2，将B3B2写入@d_0x2	@d_0x0/B1B0 @d_0x2/B3B2
16	32	4	@s_0x0/B1B0 @s_0x2/B3B2	1、 从@s_0x0读取B1B0，从@s_0x2读取B3B2， 将B3B2B1B0写入@d_0x0	@d_0x0/B3B2B1B0
32	8	4	@s_0x0/B3B2B1B0	1、 从@s_0x0读取B3B2B1B0，将B0写入@d_0x0 2、 从@s_0x0读取B3B2B1B0，将B1写入@d_0x1 3、 从@s_0x0读取B3B2B1B0，将B2写入@d_0x2 4、 从@s_0x0读取B3B2B1B0，将B3写入@d_0x3	@d_0x0/B0 @d_0x1/B1 @d_0x2/B2 @d_0x3/B3
32	16	4	@s_0x0/B3B2B1B0	1、 从@s_0x0读取B3B2B1B0，将B1B0写入@d_0x0 2、 从@s_0x0读取B3B2B1B0，将B3B2写入@d_0x2	@d_0x0/B1B0 @d_0x2/B3B2
32	32	4	@s_0x0/B3B2B1B0	1、 从@s_0x0读取B3B2B1B0，将B3B2B1B0写入@d_0x0	@d_0x0/B3B2B1B0

【注意】

1. 上表中传输数量的单位是byte，源内存地址与目标内存地址以byte为单位。

2. 只有M2M传输支持位封装模式。

3. 当传输数量与目标内存数据位宽不对齐时，例如16to32的传输中，传输数量为3byte，则DMA的操作流程为：从@s_0x0读取B1B0，从@s_0x1读取B3B2，将B2B1B0写入@d_0x0，而@d_0x0的最高byte数据不会被改变。即：传输数量设置多少byte，就只会写入多少byte的数据。

零扩展模式

在零扩展模式下，DMA_SIZE的单位为“次”，即从源内存读取DMA_SIZE次数据写到目标地址。可以设置任意源/目标位宽组合，任意传输次数。DMA从源内存读取数据后，将对数据进行零扩展后再写入目标内存。

表1-2列举了在零扩展模式下，不同数据位宽和传输数量组合情况下dma控制器对源内存和目标内存的读写情况。

表 1-2 零扩展模式

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
8	8	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，将B0写入@d_0x0 2、从@s_0x1读取B1，将B1写入@d_0x1 3、从@s_0x2读取B2，将B2写入@d_0x2 4、 从@s_0x3读取B3，将B3写入@d_0x3	@d_0x0/B0 @d_0x1/B1 @d_0x2/B2 @d_0x3/B3
8	16	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，将00B0写入@d_0x0 2、 从@s_0x1读取B1，将00B1写入@d_0x2 3、从@s_0x2读取B2，将00B2写入@d_0x4 4、从@s_0x3读取B3，将00B3写入@d_0x6	@d_0x0/00B0 @d_0x2/00B1 @d_0x4/00B2 @d_0x6/00B3

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
8	32	4	@s_0x0/B0 @s_0x1/B1 @s_0x2/B2 @s_0x3/B3	1、 从@s_0x0读取B0，将000000B0写入@d_0x0 2、 从@s_0x1读取B1，将000000B1写入@d_0x4 3、 从@s_0x2读取B2，将000000B2写入@d_0x8 4、 从@s_0x3读取B3，将000000B3写入@d_0xc	@d_0x0/000000B0 @d_0x4/000000B1 @d_0x8/000000B2 @d_0xc/000000B3
16	8	4	@s_0x0/B1B0 @s_0x2/B3B2 @s_0x4/B5B4 @s_0x6/B7B6	1、 从@s_0x0读取B1B0，将B0写入@d_0x0 2、 从@s_0x2读取B3B2，将B2写入@d_0x1 3、 从@s_0x4读取B5B4，将B4写入@d_0x2 4、 从@s_0x6读取B7B6，将B6写入@d_0x3	@d_0x0/B0 @d_0x1/B2 @d_0x2/B4 @d_0x3/B6
16	16	4	@s_0x0/B1B0 @s_0x2/B3B2 @s_0x4/B5B4 @s_0x6/B7B6	1、 从@s_0x0读取B1B0，将B1B0写入@d_0x0 2、 从@s_0x2读取B3B2，将B3B2写入@d_0x2 3、 从@s_0x4读取B5B4，将B5B4写入@d_0x4 4、 从@s_0x6读取B7B6，将B7B6写入@d_0x6	@d_0x0/B1B0 @d_0x2/B3B2 @d_0x4/B5B4 @d_0x6/B7B6

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
16	32	4	@s_0x0/B1B0 @s_0x2/B3B2 @s_0x4/B5B4 @s_0x6/B7B6	1、 从@s_0x0读取B1B0，将0000B1B0写入@d_0x0 2、 从@s_0x2读取B3B2，将0000B3B2写入@d_0x4 3、 从@s_0x4读取B5B4，将0000B5B4写入@d_0x8 4、 从@s_0x6读取B7B6，将0000B7B6写入@d_0xc	@d_0x0/0000B1B0 @d_0x4/0000B3B2 @d_0x8/0000B5B4 @d_0xc/0000B7B6
32	8	4	@s_0x0/B3B2B1B0 @s_0x4/B7B6B5B4 @s_0x8/BbBaB9B8 @s_0xc/BfBeBdBc	1、 从@s_0x0读取B3B2B1B0，将B0写入@d_0x0 2、 从@s_0x4读取B7B6B5B4，将B4写入@d_0x1 3、 从@s_0x8读取BbBaB9B8，将B8写入@d_0x2 4、 从@s_0xc读取BfBeBdBc，将Bc写入@d_0x3	@d_0x0/B0 @d_0x1/B4 @d_0x2/B8 @d_0x3/Bc
32	16	4	@s_0x0/B3B2B1B0 @s_0x4/B7B6B5B4 @s_0x8/BbBaB9B8 @s_0xc/BfBeBdBc	1、 从@s_0x0读取B3B2B1B0，将B1B0写入@d_0x0 2、 从@s_0x4读取B7B6B5B4，将B5B4写入@d_0x2 3、 从@s_0x8读取BbBaB9B8，将B9B8写入@d_0x4 4、 从@s_0xc读取BfBeBdBc，将BdBc写入@d_0x6	@d_0x0/B1B0 @d_0x2/B5B4 @d_0x4/B9B8 @d_0x6/BdBc

源内存数据位宽	目的内存数据位宽	传输数量	源内存 地址与数据	传输操作	目标内存 地址与数据
32	32	4	@s_0x0/B3B2B1B0 @s_0x4/B7B6B5B4 @s_0x8/BbBaB9B8 @s_0xc/BfBeBdBc	1、 从@s_0x0读取B3B2B1B0，将B3B2B1B0写入@d_0x0 2、 从@s_0x4读取B7B6B5B4，将B7B6B5B4写入@d_0x4 3、 从@s_0x8读取BbBaB9B8，将BbBaB9B8写入@d_0x8 4、 从@s_0xc读取BfBeBdBc，将BfBeBdBc写入@d_0xc	@d_0x0/B3B2B1B0 @d_0x4/B7B6B5B4 @d_0x8/BbBaB9B8 @d_0xc/BfBeBdBc

符号扩展模式

符号扩展模式与零扩展模式类似，区别是符号扩展模式将读到的数据进行符号扩展后再写入到目标内存，此外无其他差异，这里不再赘述。

23.3.8 DMA中断与异常

对于每个DMA通道，可在发生以下事件时产生中断：

l 循环模式下总循环传输完成

l 单次传输/每次循环传输完成

l 单次传输/每次循环传输到一半

在使能对应的中断后，DMA控制器将在满足中断的情况下向CPU发出中断。CPU在处理完中断后，需要往对应的中断清零位置1以清掉中断。

循环模式下，在使能了对应中断后，每一次循环传输将在传输到一半或传输完成时产生中断。如果在循环模式下需要用到中断，请及时响应中断并在响应后清掉中断。

中断事件	中断标志	中断使能控制位	中断清零位
循环模式下总循环传输完成	DMA_CCIF	DMA_CCIE	DMA_CCFC
单次传输/每次循环传输完成	DMA_TCIF	DMA_TCIE	DMA_TCFC
单次传输/每次循环传输一半	DMA_THIF	DMA_THIE	DMA_THFC

对于每个DMA通道，当发生以下情况之一，并且在异常开关打开的情况下（相关开关见system_EMU.doc），将触发emu异常：

l 在通道使能的情况下配置该通道的SADR/ DADR/ CON1；

l 在通道使能的情况下，SRC_SIZE/DST_SIZE/EXT_MODE配置为0b11；

l 在通道使能的情况下，P2M/M2P/P2P模式下，EXT_MODE配置为0b00；

l 在通道使能的情况下，源/目标内存地址与源/目标内存的数据位宽不对齐。

值得注意的是，不管是否使能了传输完成中断，只要在传输过程中触发传输异常，或者传输完成，dma通道使能将被硬件自动关闭。如果需要再次进行传输，需要重新配置。

23.3.9 DMA传输停止控制

DMA控制器中提供了传输停止的选项。如果在DMA达到配置的传输数据量之前将通道的DMA_STOP置1，则DMA停止传输。传输停止的过程可能要花费一些时间（需要首先完成正在进行的传输）。当传输完全停止时，传输完成标志DMA_TCF将变为1；在循环模式下，循环完成标志DMA_CCF也将变为1。传输停止后，本次DMA实际的传输数量可以通过读取JL_UDMA->JL_UDMA_CHx.CON1，结合配置的数目计算出来。

传输停止功能的加入，可以满足外设传输的数据量的数目未知时的情况。在该情况下，建议将DMA_SIZE配置为最大值，即0xffff，以方便大数据量的传输。

23.3.10 DMA传输配置流程

配置某一DMA通道x时需遵循下面的顺序：

1. 如果使能了通道x，需要读取DMA_EN和DMA_SIZE，若DMA_EN为1且DMA_SIZE不为0，说明此时该通道正在进行DMA传输，此时需要等到传输完成，待其传输完成后，再次读取DMA_SIZE确保为0、DMA_EN为0，才可以进行新的DMA传输配置，或启用DMA传输停止功能，等待DMA_EN为0才可重新配置。
2. 根据对齐要求，配置该通道的源地址CHx_SADR和目标地址CHx_DADR。
3. 配置传输数据量。
4. 外设源选择
5. 配置通道优先级
6. 配置DMA传输类型（1为M2M模式，0为外设传输模式）。
7. 配置INF_MODE（1为无限循环模式，0为可配循环次数模式）。
8. 配置循环次数。
9. 配置循环模式（1为循环模式，0为单次模式）。
10. 配置源地址和目标地址的地址递增模式（1为地址递增，0为固定地址）。
11. 配置源内存和目标内存的数据位宽。
12. 配置中断使能。
13. 使能DMA通道

【注意】DMA通道使能请在配置完其他寄存器后再使能。

23.4 _DEMO_CODE提供了代码示例。

23.5 数字模块控制寄存器

23.5.1 JL_UDMA->JL_UDMA_CHx.SADR：dma通道x源数据区基址寄存器（x为通道序号）

Bit	Name	RW	Default	RV	Description
31-0	CHx_SADR	rw	-	-	通道x源数据区基址

23.5.2 JL_UDMA->JL_UDMA_CHx.DADR：dma通道x目标数据区基址寄存器（x为通道序号）

Bit	Name	RW	Default	RV	Description
31-0	CHx_DADR	rw	-	-	通道x目标数据区基址

23.5.3 JL_UDMA->JL_UDMA_CHx.CON0：dma通道x控制寄存器0（x为通道序号）

Bit	Name	RW	Default	RV	Description
31	DMA_EXCPT	r	0	-	DMA传输异常，触发异常的情况见[1.3.8节] (#_DMA中断与异常)
30	DMA_CCF	r	0	-	DMA循环模式下总循环传输完成标志
29	DMA_TCF	r	0	-	DMA单次传输/每次循环传输完成标志
28	DMA_THF	r	0	-	DMA单次传输/每次循环传输一半标志
27	DMA_CCFC	w	0	-	DMA循环模式下总循环传输完成标志清零
26	DMA_TCFC	w	0	-	DMA单次传输/每次循环传输完成标志清零
25	DMA_THFC	w	0	-	DMA单次传输/每次循环传输一半标志清零
24	DMA_CCIE	rw	0	-	DMA循环模式下总循环传输完成中断使能
23	DMA_TCIE	rw	0	-	DMA单次传输/每次循环传输完成中断使能
22	DMA_THIE	rw	0	-	DMA单次传输/每次循环传输一半中断使能
21-18	RESERVED	-	-	-	预留
17	INF_MODE	rw	0	-	无限循环模式。在CYC_MODE=1的情况下， INF_MODE=0：可配次数循环； INF_MODE=1：无限循环。
16-15	EXT_MODE	rw	0	-	DMA数据扩展模式选择 00：位封装模式（只有M2M支持） 01：0扩展模式 10：符号扩展模式 11：预留

Bit	Name	RW	Default	RV	Description
14	DMA_STOP	rw	0	-	DMA传输停止
13-12	DST_SIZE	rw	0	-	目标数据区数据宽度 00：8bit 01：16bit 10：32bit 11：保留
11-10	SRC_SIZE	rw	0	-	源数据区数据宽度 00：8bit 01：16bit 10：32bit 11：保留
9	DCR_MODE	rw	0	-	目标数据区指针自增模式： 0：地址固定 1：地址自增
8	SCR_MODE	rw	0	-	源数据区指针自增模式： 0：地址固定 1：地址自增
7	CYC_MODE	rw	0	-	循环模式 0：非循环模式 1：循环模式
6	DMA_TYPE	rw	0	-	传输类型选择 0：PDMA 1：MDMA 默认为PDMA，支持P2M/M2P/P2P； 当该位被置1时为MDMA，SRC_SEL数值多少都无效，并且立刻发出M2M传输。
5-4	DMA_PRI	rw	0	-	通道优先级： 00：低 01：中 10：高 11：最高
3-1	SRC_SEL	rw	0	-	外设源选择
0	DMA_EN	rw	0	-	DMA使能

23.5.4 JL_UDMA->JL_UDMA_CHx.CON1：dma通道x控制寄存器1（x为通道序号）

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
31-16	CYC_TIME	rw	0	-	通道x循环次数。当配置为循环模式时，该寄存器为循环的 次数，每完成一次减1，当值为0时停止循环。若在启动DMA前配置为普通循环模式，并且将CYC_TIME配置为0， 则会传输最大循环次数（65536）。
15-0	DMA_SIZE	rw	0	-	数据传输数量。在位封装模式下，单位为“byte”，其他模式下单位为“次”，最大数据传输数量为65535

23.6 DEMO CODE

下面示例为配置DMA通道0并启动位封装模式DMA传输，CPU等待DMA传输完成后再将进行数据比对的例子。

```
void dma_ch0_cfg()
{
    JL_UDMA->JL_UDMA_CH0.SADR = 0x300011;    // 设置源地址为0x300011
    L_UDMA->JL_UDMA_CH0.DADR = 0x301002;    // 设置目标地址为0x301002
    SFR(JL_UDMA->JL_UDMA_CH0.CON1, 16, 16, 3);// 设置循环次数为3次
    SFR(JL_UDMA->JL_UDMA_CH0.CON1, 0, 16, 13); // 设置DMA的传输数据量为13byte
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 1, 3, 1);  // 选择1号外设源
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 4, 2, 3);  // 设置通道优先级为最高
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 6, 1, 0);  // 设置DMA传输为外设传输模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 7, 1, 1);  // 设置DMA通道为循环模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 8, 1, 1);  // 设置源地址为指针递增模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 9, 1, 1);  // 设置目标地址为指针递增模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 10, 2, 0); // 设置源内存数据位宽为8
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 12, 2, 1); // 目标内存数据位宽为16
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 15, 2, 0); // 数据扩展模式为位封装模式
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 25, 1, 1); // 使能传输完成中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 28, 1, 1); // 使能传输一半中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 31, 1, 1); // 使能传输错误中断
    SFR(JL_UDMA->JL_UDMA_CH0.CON0, 0, 1, 1);  // 打开DMA通道使能
}
int main(void)
{
  dma_ch0_cfg();
  lp_waiting((int*)(& JL_UDMA->JL_UDMA_CH0.CON0), BIT(30), BIT(28), 34);
  u8 *sptr = (u8 *)0x300011;
  u8 *dptr = (u8 *)0x301002;
  u8 cmp = 0x0;
  for(int i=0; i<13; i++) {
    if(*sptr!=*dptr) cmp++;
        sptr++;
        dptr++;
}
if(cmp) printf("DMA transfer data error!!!");
  else printf("DMA transfer correct!!!");
}
```

23.7 附录：项目DMA硬件映射

表 1-3 DMA硬件映射（TODO）

外设源	通道0	通道1	通道2	通道3
src0	DMA_GEN[0]	DMA_GEN[1]	DMA_GEN[2]	DMA_GEN[3]
src1				
src2				
src3				
src4				
src5				
src6				
src7				

23.8 附录：UDMA与DMA_GEN的联动

23.8.1 联动框图

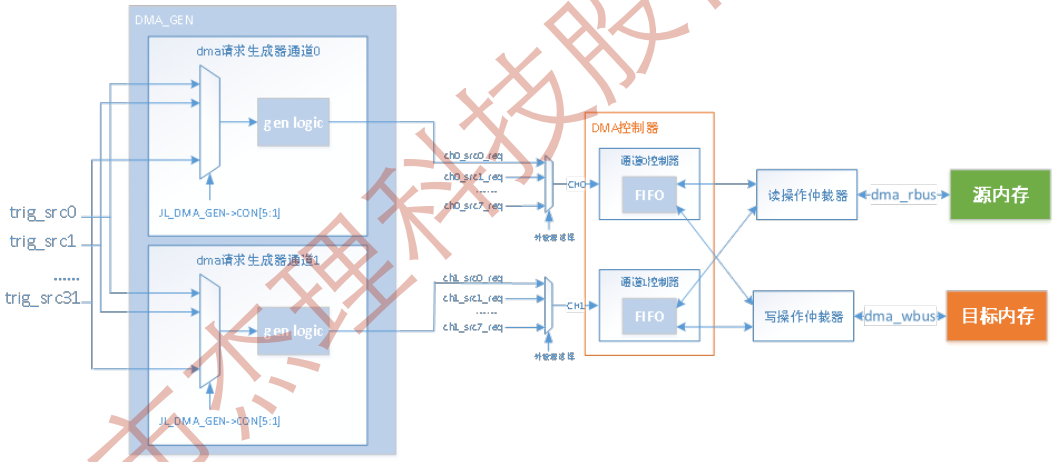


图 1-3 UDMA与DMA_GEN的联动

23.8.2 注意事项

UDMA联动DMA_GEN使用，若在无限循环模式下（CYC_MODE=1并且INF_MODE=1）停止传输，首先应该停止触发，再将UDMA相应通道的DMA_STOP置1。如果先将UDMA相应通道的DMA_STOP置1再停止触发，可能触发DMA_GEN的“触发频率过高”异常。

24 UART

24.1 概述

支持接收带循环Buffer的DMA模式和普通模式。

UART在DMA接收的时候有一个循环Buffer，UTx_RXSADR表示它的起始，UTx_RXEADR表示它的结束。同时，在接收过程中，会有一个超时计数器（UTx_OTCNT），如果在指定的时间里没有收到任何数据，则超时中断就会产生。超时计数器是在收到数据的同时自动清空。

24.2 特殊注意点

【注意】：

1. OT_PND触发流程，满足一下（1-4）顺序，则可产生OT_PND： (1). 写UTx_OTCNT，数值不为0启动OT功能； (2). 收到n个byte数据； (3). 从最后一个下降沿开始，等待OT超时（每来一个下降沿都会重新计时，时钟与接收baud_clk一致）； (4). 当OTCNT与超时时间相等，OT_PND置1（只能通过CLR_OTPND清零）。
2. 读接收的数据数量时，先将RDC该bit写1，接着asm("csync")清空流水线，然后软件查询RDC_OVER置1，置1后即可读取HRXCNT数值，RDC_OVER只在RDC写1之后生效，平常为无效状态（不管0或1）；

24.3 数字模块控制寄存器

24.3.1 UTx_CON0: uart x control register 0

Bit	Name	RW	Default	RV	Description
31-16	RESERVED	-	0	-	预留
15	TPND	r	1	-	TPND:TX Pending
14	RPND	r	0	-	RPND:RX Pending& Dma_Wr_Buf_Empty， 数据接收不完Pending不会为1
13	CLRTPND	r	0	-	CLRTPND： 清空TX Pending
12	CLRRPND	r	0	-	CLRRPND： 清空 RX Pending
11	OTPND	r	0	-	OTPND:OverTime Pending
10	CLR_OTPND	w	0	-	CLR_OTPND: 清空OTPND
9	RESERVED	-	0	-	预留
8	RDC_OVER	r	0	-	用于判断写RDC后数据是否已存入内存标志，RDC写1时会清零，当数据存入内存后置1。
7	RDC	w	0	-	RDC： 写1时，将已经收到的数目写到UTx_HRXCNT，已收到的数目清零写0无效

Bit	Name	RW	Default	RV	Description
6	RX_MODE	rw	0	-	RXMODE（*）： 读模式选择 0: 普通模式，不用DMA； 1: DMA模式。
5	OT_IE	rw	0	1	OTIE:OT中断允许 0:不允许； 1: 允许。
4	DIVS	rw	0	-	DIVS:前3分频选择，0为4分频，1为3分频
3	RXIE	rw	0	1	RXIE:RX中断允许 当RX Pending为1，而且RX中断允许为1，则会产生中断
2	TXIE	rw	0	1	TXIE:TX中断允许 当TX Pending为1，而且TX中断允许为1，则会产生中断
1	UTRXEN	rw	0	1	UTRXEN:UART模块接收使能
0	UTTXEN	rw	0	1	UTTXEN:UART模块发送使能

24.3.2 UTx_CON1: uart x control register 1

Bit	Name	RW	Default	RV	Description
15	CTSPND	r	0	0	CTSPND: CTS中断pending
14	RTSPND	r	0	0	RTSPND: RST pending
13	CLR_CTSPND	w	0	-	CLR_CTSPND：清除CTS pending
12	CLR_RTSPND	w	0	-	CLRRTS： 清除RTS 0: N/A 1: 清空RTS
11-10	RESERVED	-	0	-	预留
9	TX_INV	rw	0	0	Tx发送数据电平取反 0：不取反 1：取反
8	RX_INV	rw	0	0	Rx接收数据电平取反 0：不取反 1：取反
7	CTS_INV	rw	0	-	CTS流控有效电平选择（对方允许我方发送数据） 0：高电平有效 1：低电平有效

Bit	Name	RW	Default	RV	Description
6	RTS_INV	rw	0	-	RTS流控有效电平选择（允许对方发送数据） 0：高电平有效 1：低电平有效
5	RX_BYPASS	rw	1	1	Rx接收数据通路直通使能 0：滤波输入 1：直通输入
4	RX_DISABLE	r	1	0	关闭数据接收 0：开启输入（正常模式） 1：关闭输入（输入固定为1）
3	CTSIE	rw	0	-	CTSIE: CTS中断使能 0：禁止中断； 1：中断允许
2	CTSE	rw	0	-	CTSE:CTS使能 0：禁止CTS硬件流控制 1： 允许CTS硬件流控制
1	RESERVED	-	0	-	预留
0	RTSE	rw	0	-	RTSE:RTS使能 0: 禁止RTS硬件流控制 1: 允许RTS硬件流控制

24.3.3 UTx_CON2: uart x control register 2

Bit	Name	RW	Default	RV	Description
15-3	RESERVED	-	0	-	预留
8	CHK_PND	r	0	-	校验中断标志
7	CLR_CHKPND	w	0	-	校验中断CHK_PND清除，置1清除
6	CHK_IE	rw	0	1	校验中断使能，置1使能
5-4	CHECK_MODE	rw	0	-	校验模式选择： 0：常0 1：常1 2：偶校验 3：奇校验
3	CHECK_EN	rw	0	0	校验功能使能位，置1使能
2	RB8	r	0	0	RB8:9bit模式时，RX接收的第9位
1	RESERVED	-	-	-	预留
0	M9EN	rw	0	0	M9EN：9bit模式使能

24.3.4 UTx_BAUD: uart x baudrate register

Bit	Name	RW	Default	RV	Description
15-0	UTx_BAUD	w	0x0	0x0	注释如下所示

uart 的UTx_DIV的整数部分

串口频率分频器因子(UTx_DIV)的整数部分

当DIVS=0时，

Baudrate = Freq_sys / ((UTx_BAUD+1) *4 + BAUD_FRAC)

当DIVS=1时，

Baudrate = Freq_sys / ((UTx_BAUD+1) *3 + BAUD_FRAC)
(Freq_sys是apb_clk,指慢速设备总线的时钟，非系统时钟)

24.3.5 UTx_BUF: uart x data buffer register

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	0	-	预留
7-0	UT_BUF	rw	0x0	-	uart的收发数据寄存器 写UTx_BUF可启动一次发送； 读UTx_BUF可获得已接收到的数据。

24.3.6 UTx_TXADR: uart x TX DMA buffer register

Bit	Name	RW	Default	RV	Description
31-25	RESERVED	-	0	-	预留
24-0	UTx_TXADR	w	0x0	0x0	DMA发送数据的起始地址

24.3.7 UTx_TXCNT: uart x TX DMA counter register

Bit	Name	RW	Default	RV	Description
31-0	UTx_TXCNT	w	0x0	0x0	写UTx_TXCNT，控制器产生一次DMA的操作，同时清空中断， 当uart需要发送的数据达到UTx_TXCNT的值， 控制器会停止发送数据的操作，同时产生中断（UTx_CON[15]）。

24.3.8 UTx_RXCNT: uart x RX DMA counter register

Bit	Name	RW	Default	RV	Description

Bit	Name	RW	Default	RV	Description
31-0	UTx_RXCNT	w	0x0	0x0	写UTx_RXCNT，控制器产生一次DMA的操作， 同时清空中断， 当uart需要接收的数据达到UTx_RXCNT的值， 控制器会停止接收数据的操作，同时产生中断 （UTx_CON[14]）。

24.3.9 UTx_RXSADR: uart x RX DMA start address register

Bit	Name	RW	Default	RV	Description
31-25	RESERVED	-	0	-	预留
24-0	UTx_RXSADR	w	0x0	0x0	DMA接收数据时，循环buffer的开始地址， 需4byte地址对齐

24.3.10 UTx_RXEADR: uart x RX DMA end address register

Bit	Name	RW	Default	RV	Description
31-25	RESERVED	-	0	-	预留
24-0	UTx_RXEADR	w	0x0	0x0	DMA接收数据时，循环buffer的结束地址。 需4byte地址对齐 【注意】：UTx_RXSADR=UTx_RXEADR时没有循环buffer，接收地址会递增

24.3.11 UTx_HRXCNT: uart x have RX DMA counter register

Bit	Name	RW	Default	RV	Description
31-0	UTx_HRXCNT	r	0x0	0x0	当设这RDC（UTx_CON[7]）=1时， 串口设备会将当前总共收到的字节数记录到 UTx_HRXCNT里。

24.3.12 UTx_OTCNT: uart x Over Timer counter register

Bit	Name	RW	Default	RV	Description
31-0	UTx_OTCNT	w	0x0	0x0	设置串口设备在等待RX下降沿的时间， 在设置的时间内没收到RX下降沿，则产生OT_PND Time(ot)= Time(rx_baud_clk) * UTx_OTCNT; 例如：接收波特率时间为100ns，UTx_OTCNT = 10,那么，OT的时间就为1000ns

24.3.13 UTx_ERR_CNT: uart x error byte counter register

Bit	Name	RW	Default	RV	Description
31-0	UTx_ERR_CNT	r	0x0	-	校验错误数量计数，当打开CHECK_EN后， 当第一次校验出错时， 该寄存器会记录当前byte对应的数量。 【注意】：只记录第一次出错的数量，清除CHK_PND会重新记录。

25 USB BRIDGE

25.1 概述

1. 负责控制系统与USB模块之间的通讯，包括：

1. 接收CPU的命令，控制SIE；

2. 管理endpoint mapping；

3. 负责SIE和memory的通讯；

4. 控制USB PHY。

25.2 特殊注意点

【注意】：目前总共4个Endpoint，为EP0~EP3，其中对应的深度为，EP0:64、EP1 TX/RX：1024、EP2 TX/RX：1024、EP3 TX/RX：1024.

【时钟要求】： $6/f_{hsb\_clk} + 3.5/f_{hsb\_clk} < 583ns$

当lsb_clk = 48M时，hsb_clk >=8M

当lsb_clk = 24M时，hsb_clk >=11M

25.3 寄存器列表

寄存器列表	USB			
USB_CONx	USB_CON0			
EPx_CNT	EP0_CNT	EP1_CNT	EP2_CNT	
EP0_ADR	EP0_ADR			
EPx_TADR	EP1_TADR	EP2_TADR		
EPx_RADR	EP1_RADR	EP2_RADR		
USB_IO_CONx	USB_IO_CON0			
USB_SIE	FADDR	POWER	INTRTX1	INTRTX2
	INTRRX1	INTRRX2	INTRUSB	INTRTX1E
	INTRTX2E	INTRRX1E	INTRRX2E	INTRRX2E
	FRAME1	FRAME2	DEVCTL	
USB_EP0	CSR0		COUNT0	
	TXMAXP	TXCSR1	TXCSR2	RXMAXP

USB_EPX				
	RXCSR1	RXCSR2	RXCOUNT1	RXCOUNT2
	TXTYPE	TXINTERVAL	RXTYPE	RXINTERVAL
TXDLY_CON				
EPx_RX_LEN	EP1_RX_LEN	EP2_RX_LEN		
EP1_MTX_PRD				
EP1_MTX_NUM				
EP1_MRX_PRD				
EP1_MRX_NUM				

Endpoint大小：

EP号	BUFFER大小(BYTE)
EP0	64
EP1_TX	1024
EP1_ RX	1024
EP2_TX	1024
EP2 _RX	1024
EP3_TX	1024
EP3_RX	1024

25.4 数字模块控制寄存器

25.4.1 USB_CON0：usb control register 0

Bit	Name	RW	Default	RV	Description
31-29	RESERVED	-	-	-	预留
28	TX_BLOCK_EN	rw	0	0	发送块使能
27	RESERVED	-	-	-	预留
26	RESERVED	-	-	-	预留

Bit	Name	RW	Default	RV	Description
25	EP2_RLIM_EN	rw	0	-	EP2写限制使能
24	EP1_RLIM_EN	rw	0	-	EP1写限制使能
23	RESERVED	-	-	-	预留
22	RESERVED	-	-	-	预留
21	EP2_DISABLE	rw	0	0	关闭端点2，不会返回ack, nak, stall
20	EP1_DISABLE	rw	0	0	关闭端点1，不会返回ack, nak, stall
19	RST_STL	rw	0	-	EP0从stall状态复位为idle状态
18	LOWP_MD_	rw	0	0	低有效，默认为使用低功耗模式 0：使用低功耗模式，即系统时钟可跑低于48M；1：不使用低功耗模式，即要用USB模块时系统得跑48M或48M以上；
17-15	RESERVED	-	-	-	预留
14	SIE_PND	r	0	0	SIE中断请求标志
13	SOF_PND	r	0	0	SOF中断请求标志
12	CLR_SOF	w	0	1	写1清除SOF中断请求标志，写0无效
11	SIEIE	rw	0	0	SIE中断使能
10	SOFIE	rw	0	0	SOF中断使能
6-9	RESERVED	-	-	-	预留
5	VBUS	rw	0	0	USB 电源
4	CID	rw	0	0	USB工作模式： 0：host 1：device
3	TM1	rw	0	0	用于缩短检测连接时间(short connect timeout)： 0：disable 1：enable
2	USB_NRST	rw	0	0	USB模块复位： 0：reset 1：release reset
1	LOW_SPEED	rw	0	0	低速USB_DMA使能： 0：disable 1：enable
0	RESERVED	-	-	-	预留

25.4.2 USB_CON1： usb control register 1

Bit	Name	RW	Default	RV	Description
31-6	RESERVED	-	-	-	预留

Bit	Name	RW	Default	RV	Description
5	EP1_MRX_PND	r	0	-	ep1连续接收完成标志位
4	EP1_MTX_PND	r	0	-	ep1连续发送完成标志位
3	EP1_MRX_PND_CLR	w	0	-	清除ep1连续接收完成标志位
2	EP1_MTX_PND_CLR	w	0	-	清除ep1连续发送完成标志位
1	EP1_MRX_EN	rw	0	-	ep1连续接收使能
0	EP1_MTX_EN	rw	0	-	ep1连续发送使能 【注意】： 1. 连续收发期间，最后一包数据完成才会起正常传输中断，期间如stall等异常中断也会起来； 2. 连续收发的数据存档地址是连续的；

25.4.3 EP0_CNT, EP1_CNT, EP2_CNT

Bit	Name	RW	Default	RV	Description
31-0	EP(n)_CNT	w	0x0	0x0	usb endpoint 0-3发送数据个数，单位为byte

25.4.4 EP0_ADR

Bit	Name	RW	Default	RV	Description
31-0	EP0_ADR	w	0x0	0x0	usb endpoint 0发送/接收数据的起始地址

25.4.5 EP1_TADR , EP2_TADR

Bit	Name	RW	Default	RV	Description
31-0	EP(n)_TADR	w	0x0	0x0	usb endpoint 1-3发送数据起始地址

25.4.6 EP1_RADR , EP2_RADR

Bit	Name	RW	Default	RV	Description
31-0	EP(n)_RADR	w	0x0	0x0	usb endpoint 1-3接收数据起始地址

USB_IO CONTROL

25.4.7 JL_PORTUSB->DIE

25.4.8 JL_PORTUSB->DIEH

25.4.9 JL_PORTUSB->DIR

25.4.10 JL_PORTUSB->PU0：DP上拉1.5K，DM上拉180K

25.4.11 JL_PORTUSB->PD0：DP 下拉15K，DM下拉15K

25.4.12 JL_PORTUSB->IN

25.4.13 JL_PORTUSB->OUT

25.4.14 JL_PORTUSB->SPL

25.4.15 以上USB IO寄存器和普通IO用法是类似的，其中BIT(0)是配置DP的，BIT(1)是配置DM的

25.4.16 JL_PORTUSB->CON：usb io control register 0

Bit	Name	RW	Default	RV	Description
31-13	RESERVED	-	-	-	预留
12-11	DBG_SEL	rw	0x0	-	测试信号选择： 00：1'b0 01：usb_diff 10：usb_dp 11：usb_dm
10-9	RESERVED	-	-	-	预留
8	IO_MODE	rw	0x1	-	IO模式使能 0：USB模式 1：普通IO模式
7	CHKDPO	r	0x1	-	CHKDPO经系统时钟采样后的输入
6	DIDF	r	0	-	差分输入DIDF经lsb_clk采样后的输入
5	RESERVED	-	-	-	预留
4	PDCHKDP	rw	0x0	-	DP 外接下拉检查使能 0：disable 1：enable
3	DM_ADC_EN	rw	0x0	-	DM到SARADC通路en,默认断开
2	SR0	rw	0x0	-	输出驱动能力选择
1	DP_ADC_EN	rw	0x0	-	DP到SARADC通路en,默认断开
0	RCVEN	rw	0x0	-	USB_PHY使能(差分输入使能，使用USB功能时需打开这一BIT)

25.4.17 SOF_STA_CON：sof control register（用于sdc与usb复用IO时使用)

Bit	Name	RW	Default	RV	Description
31-4	PRD	rw	0x0	-	设置令牌包与数据包的延迟时钟个数
3	sof_sta2	r	0	-	用于切换IO状态标志,在发送SOF前自动切为USB模式
2	sof_sta1	r	0	-	用于SDC停时钟标志，1停时钟（硬件停)

Bit	Name	RW	Default	RV	Description
1	sof_sta0	r	0	-	CPU读取该位为高时不能有新的SDC操作
0	force_txsof_en	rw	0	-	发送sof时停止SDC的时钟使能

25.4.18 TXDLY_CON：tx delay control register

Bit	Name	RW	Default	RV	Description
31-16	PRD	rw	0x0	-	设置令牌包与数据包的延迟时钟个数
15-3	RESERVED	-	-	-	预留
2	PND	r	0	-	延迟时钟数等于数据包的延迟功能
1	CLR_PND	w	0	-	清除pending
0	CLK_DIS	rw	0	-	1：开启令牌包与数据包的延迟功能 0：关闭令牌包与数据包的延迟功能

25.4.19 EPx_RX_LEN： EP1_RX_LEN，EP2_RX_LEN

Bit	Name	RW	Default	RV	Description
31-11	RESERVED	-	-	-	预留
10-0	EPx_RX_LEN	rw	0x0	-	EP（x）写数据最大地址长度（EPx_BASE_ADR到EPx_BASE_ADR+LEN-1），写满len个数后，dma停止接收数据。 注意：每一次新的传输，dma地址都会回到BASE_ADR

25.4.20 EP1_MTX_PRD

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	预留
7-0	EP1_MTX_PRD	rw	0x0	-	EP1连续发送的包数

25.4.21 EP1_MTX_NUM

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	预留
7-0	EP1_MTX_NUM	r	0x0	-	EP1连续发送中已发送包数，写mtx_pnd_clr会清0

25.4.22 EP1_MRX_PRD

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	预留
7-0	EP1_MRX_PRD	rw	0x0	-	EP1连续接收的包数

25.4.23 EP1_MRX_NUM

Bit	Name	RW	Default	RV	Description
31-8	RESERVED	-	-	-	预留
7-0	EP1_MRX_NUM	r	0x0	-	EP1连续发送中已接收包数，写mrx_pnd_clr会清0

25.5 DEMO CODE

25.5.1 主机模式下的例程：

```
#include "includes.h"
#define set_bit(x,y) x|=(1<<y)
#define clr_bit(x,y) x&=~(1<<y)
#define rever_bit(x,y) x^=(1<<y)
#define get_bit(x,y) ((x)>>(y)&1)
#define EP0    1
#define EP1    1
#define EP2    1
#define EP3    1
//-----------------------------------------------//
// interrupt
//-----------------------------------------------//
__attribute__((interrupt("")))
void sie0_isr(void)
{
  JL_TIMER1->CNT = JL_USB_SIE->INTRTX1;
  JL_TIMER1->CNT = JL_USB_SIE->INTRRX1;
  JL_TIMER1->CNT = JL_USB_SIE->INTRUSB;
  JL_TIMER2->CNT++;
}
__attribute__((interrupt("")))
void sof0_isr(void)
{
  JL_USB->CON0 |= BIT(12);
}
#define usb0_con0_init                          \
           /*EP4_RLIM_EN  1bit RW */(0 << 27) | \
           /*EP3_RLIM_EN  1bit RW */(0 << 26) | \
           /*EP2_RLIM_EN  1bit RW */(0 << 25) | \
           /*EP1_RLIM_EN  1bit RW */(0 << 24) | \
           /*EP4_DISABLE  1bit RW */(0 << 23) | \
           /*EP3_DISABLE  1bit RW */(0 << 22) | \
           /*EP2_DISABLE  1bit RW */(0 << 21) | \
```

```
            /*EP1_DISABLE  1bit RW */(0 << 20) | \
            /*RST_STL    1bit RW */(0 << 19) | \
            /*LOWP_MD    1bit RW */(0 << 18) | \
            /*SE_DP     1bit RW */(0 << 17) | \
            /*SE_DM     1bit RW */(0 << 16) | \
            /*CHCDPO_S    1bit RW */(0 << 15) | \
            /*SIE_PND    1bit RO */(0 << 14) | \
            /*SOF_PND    1bit RO */(0 << 13) | \
            /*CLR_SOFR    1bit WO */(1 << 12) | \
            /*SIEIE     1bit RW */(1 << 11) | \
            /*SOFIE     1bit RW */(1 << 10) | \
            /*PDCHKDP    1bit RW */(1 << 9) | \
            /*USB_TEST    1bit RW */(1 << 6) | \
            /*VBUS      1bit RW */(1 << 5) | \
            /*CID      1bit RW */(0 << 4) | \
            /*TIM1      1bit RW */(1 << 3) | \
            /*USB_NRST    1bit RW */(1 << 2) | \
            /*LOW_SPEED   1bit RW */(0 << 1) | \
            /*PHY_ON     1bit RW */(1 << 0)

#define usb0_io_con0_init                   \

            /*IO_PU_MODE   1bit RW */(0 << 14) | \
            /*SR       1bit RW */(0 << 13) | \
            /*IO_MODE    1bit RW */(0 << 12) | \
            /*DMDIEH     1bit RW */(1 << 11) | \
            /*DPDIEH     1bit RW */(1 << 10) | \
            /*DMDIE     1bit RW */(1 << 9) | \
            /*DPDIE     1bit RW */(1 << 8) | \
            /*DMPD      1bit RW */(0 << 7) | \
            /*DPPD      1bit RW */(0 << 6) | \
            /*DMPU      1bit RW */(0 << 5) | \
            /*DPPU      1bit RW */(0 << 4) | \
            /*DMIE      1bit RW */(0 << 3) | \
             /*DPIE      1bit RW */(0 << 2) | \
            /*DMOUT     1bit RW */(0 << 1) | \
            /*DPOUT     1bit RW */(0 << 0)
#define LS0_DEV  ( JL_USB_SIE->DEVCTL & BIT(5) )
#define FS0_DEV  ( JL_USB_SIE->DEVCTL & BIT(6) )
#define DMA_ADR0  0x100000
void out_flag(int flag)
{
  JL_PORTA->DIR &= ~(BIT(4)|BIT(5)|BIT(6)|BIT(7)|BIT(8));
  SFR(JL_PORTA->OUT, 4, 5, flag);
}
void usb_clk_int(void)
{
  //pll initial
  SFR(JL_PLL0->CON3, 0, 5, 0b11111);//oe: 4p5 | 3p5 | 2p5 | 1p5 | 1p0
  SFR(JL_CLOCK->CLK_CON2, 8, 2, 0);//0:std_48m 1:osc_clk 2:lsb_clk 3:1'b1
}
//-----------------------------------------------//
// main
//-----------------------------------------------//
//int main (void)
void usb_mst (void)
{
  int i;
```

```
  u8 volatile *ep0_adr0;
  u8 volatile *ep1_tadr0, *ep1_radr0;
  u8 volatile *ep2_tadr0, *ep2_radr0;
  u8 volatile *ep3_tadr0, *ep3_radr0;
  u8 volatile *ep4_tadr0, *ep4_radr0;
  u8 dev_req[8] = {0x80, 0x06, 0x00, 0x01, 0x00, 0x00, 0x12, 0x00};
  JL_TIMER2->CNT = 0;
  out_flag(0);
  enable_int();
  INTALL_HWI(28,sof0_isr,0);
  INTALL_HWI(29,sie0_isr,0);
  usb_clk_int();
  JL_USB->CON0 = usb0_con0_init;
  JL_USB_IO->CON0 = usb0_io_con0_init;
  JL_USB_SIE->INTRTX1E = 0x1f;
  JL_USB_SIE->INTRRX1E = 0x1f;
  JL_USB_SIE->DEVCTL |= BIT(0);
  while(!(LS0_DEV | FS0_DEV));
  JL_USB_SIE->POWER |= BIT(3);//reset
  delay(100);
  JL_USB_SIE->POWER = 0;
//-----------------------------------------------------------//
// SetUp
//-----------------------------------------------------------//
#if EP0
//setup
  ep0_adr0 = (u8 *)DMA_ADR0;
  for(i=0; i<8; i++)
  {
    *(ep0_adr0++) = dev_req[i];
  }
  JL_USB->EP0_ADR = DMA_ADR0;
  JL_USB->EP0_CNT = 0x08;
  JL_USB_EP0->CSR0 |= (BIT(3) | BIT(1));
  while(JL_USB_EP0->CSR0 & BIT(1));
  JL_USB->TXDLY_CON &= ~BIT(0);
//data
  ep0_adr0 = (u8 *)DMA_ADR0;
  JL_USB->EP0_ADR = DMA_ADR0;
  JL_USB_EP0->CSR0 |= BIT(5);//send stall
  while(!(JL_USB_EP0->CSR0 & BIT(0)));
  JL_USB_EP0->CSR0 |= BIT(6);//clear RxPktRdy
  for(i=0; i<8; i++)
  {
   if(*(ep0_adr0++) == i)
    JL_TIMER0->PRD = 0x69;
   else
    JL_TIMER0->PRD = 0x404;
  }
  if(JL_TIMER0->PRD == 0x69)
    out_flag(0b00001);
  else
    out_flag(0b00000);
//state
  JL_USB_EP0->CSR0 = (BIT(6) | BIT(1));
  while(JL_USB_EP0->CSR0 & BIT(1));
  JL_USB_EP0->CSR0 = 0x00;
#endif
```

```
//-------------------------------------------------------------//
// EP1
//-------------------------------------------------------------//
#if EP1
  JL_USB_SIE->FADDR = 1;
  JL_USB_EP1->RXTYPE = 0x11;//iso
  JL_USB_EP1->TXTYPE = 0x21;//bulk
  JL_USB_EP1->TXMAXP = 0x80;
  JL_USB_EP1->RXMAXP = 0x80;
//rx
  ep1_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP1_RADR = DMA_ADR0;
  JL_USB_EP1->RXCSR1 = BIT(5);//IN
  while(!(JL_USB_EP1->RXCSR1 & BIT(0)));
  JL_USB_EP1->RXCSR1 = 0;
  for(i=0; i<1023; i++)
  {
    if(*(ep1_radr0++) == 0x58)
     JL_TIMER0->PRD = 0xe1;
    else
     JL_TIMER0->PRD = 0x404;
  }
  if(JL_TIMER0->PRD == 0xe1)
    out_flag(0b00011);
  else
    out_flag(0b00001);
//tx
  ep1_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP1_TADR = DMA_ADR0;
  for(i=0; i<64*2; i++)
  {
    *(ep1_tadr0++) = 0xaa;
  }
  delay(100);
  JL_USB->EP1_CNT = 64;
  JL_USB_EP1->TXCSR1 = BIT(0);
  while(JL_USB_EP1->TXCSR1 & BIT(0));
  JL_USB->EP1_TADR = DMA_ADR0 +64;
  JL_USB->EP1_CNT = 64;
  JL_USB_EP1->TXCSR1 = BIT(0);
  while(JL_USB_EP1->TXCSR1 & BIT(0));
#endif
//-------------------------------------------------------------//
// EP2
//-------------------------------------------------------------//
#if EP2
  JL_USB_SIE->FADDR = 2;
  JL_USB_EP2->RXTYPE = 0x32;//interrupt
  JL_USB_EP2->TXTYPE = 0x32;//interrupt
  JL_USB_EP2->TXMAXP = 0x80;
  JL_USB_EP2->RXMAXP = 0x80;
//tx
  ep2_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP2_TADR = DMA_ADR0;
  for(i=0; i<64; i++)
  {
    *(ep2_tadr0++) = 0xaa;
  }
```

```
  delay(100);
  JL_USB->EP2_CNT = 64;
  JL_USB_EP2->TXCSR1 = BIT(0);
  while(JL_USB_EP2->TXCSR1 & BIT(0));
//rx
  ep2_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP2_RADR = DMA_ADR0;
  JL_USB_EP2->RXCSR1 = BIT(5);//IN
  while(!(JL_USB_EP2->RXCSR1 & BIT(0)));
  JL_USB_EP2->RXCSR1 = 0;
  for(i=0; i<64; i++)
  {
    if(*(ep2_radr0++) == 0x58)
     JL_TIMER0->PRD = 0xe2;
    else
     JL_TIMER0->PRD = 0x404;
  }
  if(JL_TIMER0->PRD == 0xe2)
    out_flag(0b00111);
  else
    out_flag(0b00011);
#endif
//-------------------------------------------------------------//
// EP3
//-------------------------------------------------------------//
#if EP3
  JL_USB_SIE->FADDR = 3;
  JL_USB_EP3->RXTYPE = 0x23;//interrupt
  JL_USB_EP3->TXTYPE = 0x23;//interrupt
  JL_USB_EP3->TXMAXP = 0x80;
  JL_USB_EP3->RXMAXP = 0x80;
//tx
  ep3_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP3_TADR = DMA_ADR0;
  for(i=0; i<64; i++)
  {
    *(ep3_tadr0++) = 0xaa;
  }
  delay(100);
  JL_USB->EP3_CNT = 64;
  JL_USB_EP3->TXCSR1 = BIT(0);
  while(JL_USB_EP3->TXCSR1 & BIT(0));
//rx
  ep3_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP3_RADR = DMA_ADR0;
  JL_USB_EP3->RXCSR1 = BIT(5);//IN
  while(!(JL_USB_EP3->RXCSR1 & BIT(0)));
  JL_USB_EP3->RXCSR1 = 0;
  for(i=0; i<64; i++)
  {
    if(*(ep3_radr0++) == 0x58)
     JL_TIMER0->PRD = 0xe3;
    else
     JL_TIMER0->PRD = 0x404;
  }
  if(JL_TIMER0->PRD == 0xe3)
    out_flag(0b01111);
  else
```

```
    out_flag(0b00111);
#endif
}
```

25.5.2 从机模式下的例程：

```
#include "includes.h"
#define set_bit(x,y) x|=(1<<y)
#define clr_bit(x,y) x&=~(1<<y)
#define rever_bit(x,y) x^=(1<<y)
#define get_bit(x,y) ((x)>>(y)&1)
#define EP0    1
#define EP1    1
#define EP2    1
#define EP3    1
//-----------------------------------------------//
// interrupt
//-----------------------------------------------//
__attribute__((interrupt("")))
void sie0_isr(void)
{
  JL_TIMER1->CNT = JL_USB_SIE->INTRTX1;
  JL_TIMER1->CNT = JL_USB_SIE->INTRRX1;
  JL_TIMER1->CNT = JL_USB_SIE->INTRUSB;
  JL_TIMER2->CNT++;
}
__attribute__((interrupt("")))
void sof0_isr(void)
{
  JL_USB->CON0 |= BIT(12);
}
#define usb0_con0_init                          \
          /*EP4_RLIM_EN  1bit RW */(0 << 27) | \
          /*EP3_RLIM_EN  1bit RW */(0 << 26) | \
          /*EP2_RLIM_EN  1bit RW */(0 << 25) | \
          /*EP1_RLIM_EN  1bit RW */(0 << 24) | \
          /*EP4_DISABLE  1bit RW */(0 << 23) | \
          /*EP3_DISABLE  1bit RW */(0 << 22) | \
          /*EP2_DISABLE  1bit RW */(0 << 21) | \
          /*EP1_DISABLE  1bit RW */(0 << 20) | \
          /*RST_STL    1bit RW */(0 << 19) | \
          /*LOWP_MD    1bit RW */(0 << 18) | \
          /*SE_DP      1bit RW */(0 << 17) | \
          /*SE_DM      1bit RW */(0 << 16) | \
          /*CHCDPO_S     1bit RW */(0 << 15) | \
          /*SIE_PND    1bit RO */(0 << 14) | \
          /*SOF_PND    1bit RO */(0 << 13) | \
          /*CLR_SOFR     1bit WO */(1 << 12) | \
          /*SIEIE      1bit RW */(1 << 11) | \
          /*SOFIE      1bit RW */(1 << 10) | \
          /*PDCHKDP    1bit RW */(1 << 9) | \
          /*USB_TEST     1bit RW */(1 << 6) | \
          /*VBUS       1bit RW */(1 << 5) | \
          /*CID       1bit RW */(1 << 4) | \
          /*TIM1       1bit RW */(1 << 3) | \
          /*USB_NRST     1bit RW */(1 << 2) | \
          /*LOW_SPEED    1bit RW */(0 << 1) | \
```

```
            /*PHY_ON      1bit RW */(1 << 0)
#define usb0_io_con0_init                    \
            /*IO_PU_MODE   1bit RW */(0 << 14) | \
            /*SR         1bit RW */(0 << 13) | \
            /*IO_MODE     1bit RW */(0 << 12) | \
            /*DMDIEH      1bit RW */(1 << 11) | \
            /*DPDIEH      1bit RW */(1 << 10) | \
            /*DMDIE      1bit RW */(1 << 9) | \
            /*DPDIE      1bit RW */(1 << 8) | \
            /*DMPD       1bit RW */(1 << 7) | \
            /*DPPD       1bit RW */(0 << 6) | \
            /*DMPU       1bit RW */(0 << 5) | \
            /*DPPU       1bit RW */(1 << 4) | \
            /*DMIE       1bit RW */(0 << 3) | \
            /*DPIE       1bit RW */(0 << 2) | \
            /*DMOUT      1bit RW */(0 << 1) | \
            /*DPOUT      1bit RW */(0 << 0)
#define LS0_DEV  ( JL_USB_SIE->DEVCTL & BIT(5) )
#define FS0_DEV  ( JL_USB_SIE->DEVCTL & BIT(6) )
#define DMA_ADR0  0x100000
void out_flag(int flag)
{
  JL_PORTA->DIR &= ~(BIT(5)|BIT(6)|BIT(7)|BIT(8));
  SFR(JL_PORTA->OUT, 5, 4, flag);
}
void usb_clk_int(void)
{
  //pll initial
  SFR(JL_PLL0->CON3, 0, 5, 0b11111);//oe: 4p5 | 3p5 | 2p5 | 1p5 | 1p0
  SFR(JL_CLOCK->CLK_CON2, 8, 2, 0);//0:std_48m 1:osc_clk 2:lsb_clk 3:1'b1
}
//-------------------------------------------//
// main
//-------------------------------------------//
void usb_slv (void)
{
  int i;
  u8 volatile *ep0_adr0;
  u8 volatile *ep1_tadr0, *ep1_radr0;
   u8 volatile *ep2_tadr0, *ep2_radr0;
  u8 volatile *ep3_tadr0, *ep3_radr0;
  u8 volatile *ep4_tadr0, *ep4_radr0;
  u8 dev_req[8] = {0x80, 0x06, 0x00, 0x01, 0x00, 0x00, 0x12, 0x00};
  JL_TIMER2->CNT = 0;
  out_flag(0);
  enable_int();
  INTALL_HWI(28,sof0_isr,0);
  INTALL_HWI(29,sie0_isr,0);
  usb_clk_int();
  JL_USB->CON0 = usb0_con0_init;
  JL_USB_IO->CON0 = usb0_io_con0_init;
  JL_USB_SIE->INTRTX1E = 0x7;
  JL_USB_SIE->INTRRX1E = 0x7;
  while(!(JL_USB_SIE->POWER & BIT(3)));
  out_flag(1);
  while(JL_USB_SIE->POWER & BIT(3));
  out_flag(2);
//-----------------------------------------------------------//
```

```
// RxSetUP
//-------------------------------------------------------------//
#if EP0
//setup
  ep0_adr0 = (u8 *)DMA_ADR0;
  JL_USB->EP0_ADR = DMA_ADR0;
  while(!(JL_USB_EP0->CSR0 & BIT(0)));
  JL_USB_EP0->CSR0 |= BIT(6);
  for(int i=0; i<8; i++)
  {
   if(dev_req[i] == *(ep0_adr0++))
    JL_TIMER0->PRD = 0x69;
   else
    JL_TIMER0->PRD = 0x404;
  }
  out_flag(3);
//data
  ep0_adr0 = (u8 *)DMA_ADR0;
  JL_USB->EP0_ADR = DMA_ADR0;
  for(int i=0; i<8; i++)
  {
    *(ep0_adr0++) = i;
  }
  JL_USB->EP0_CNT = 0x08;
  JL_USB_EP0->CSR0 |= BIT(3) | BIT(1);
  while(JL_USB_EP0->CSR0 & BIT(1));
  out_flag(4);
//state
  delay(30);
#endif
//-------------------------------------------------------------//
// EP1
//-------------------------------------------------------------//
#if EP1
  JL_USB_SIE->FADDR = 1;
  JL_USB_EP1->TXMAXP = 0x80;
  JL_USB_EP1->RXMAXP = 0x80;
//tx
  JL_USB_EP1->TXCSR2 |= BIT(6);//iso
  ep1_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP1_TADR = DMA_ADR0;
  for(i=0; i<1023; i++)
  {
    *(ep1_tadr0++) = 0x58;
  }
  delay(100);
  JL_USB->EP1_CNT = 1023;
  JL_USB_EP1->TXCSR1 = BIT(0);//tx:load paket to fifo
  while(JL_USB_EP1->TXCSR1 & BIT(0));
  out_flag(5);
//rx:bulk
  ep1_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP1_RADR = DMA_ADR0;
  while(!(JL_USB_EP1->RXCSR1 & BIT(0)));
  JL_USB_EP1->RXCSR1 = 0;
  ep1_radr0 = (u8 *)(DMA_ADR0 + 64);
  JL_USB->EP1_RADR = DMA_ADR0;
  while(!(JL_USB_EP1->RXCSR1 & BIT(0)));
```

```
  JL_USB_EP1->RXCSR1 = 0;
  ep1_radr0 = (u8 *)DMA_ADR0;
  for(i=0; i<64*2; i++)
  {
    if(*(ep1_radr0++) == 0xaa)
     JL_TIMER0->PRD = 0xe1;
    else
     //JL_TIMER0->PRD = 0x404;
     JL_TIMER0->PRD = *(ep1_radr0++);
  }
  out_flag(6);
#endif
//-------------------------------------------------------------//
// EP2
//-------------------------------------------------------------//
#if EP2
  JL_USB_SIE->FADDR = 2;
  JL_USB_EP2->TXMAXP = 0x80;
  JL_USB_EP2->RXMAXP = 0x80;
//rx:
  ep2_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP2_RADR = DMA_ADR0;
  while(!(JL_USB_EP2->RXCSR1 & BIT(0)));
  JL_USB_EP2->RXCSR1 = 0;
  ep2_radr0 = (u8 *)DMA_ADR0;
  for(i=0; i<64; i++)
  {
    if(*(ep2_radr0++) == 0xaa)
     JL_TIMER0->PRD = 0xe2;
    else
     JL_TIMER0->PRD = 0x404;
  }
  out_flag(7);
//tx
  ep2_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP2_TADR = DMA_ADR0;
  for(i=0; i<64; i++)
  {
    *(ep2_tadr0++) = 0x58;
  }
  delay(100);
  JL_USB->EP2_CNT = 64;
  JL_USB_EP2->TXCSR1 = BIT(0);//tx:load paket to fifo
  while(JL_USB_EP2->TXCSR1 & BIT(0));
  out_flag(8);
#endif
//-------------------------------------------------------------//
// EP3
//-------------------------------------------------------------//
#if EP3
  JL_USB_SIE->FADDR = 3;
  JL_USB_EP3->TXMAXP = 0x80;
  JL_USB_EP3->RXMAXP = 0x80;
//rx:
  ep3_radr0 = (u8 *)DMA_ADR0;
  JL_USB->EP3_RADR = DMA_ADR0;
  while(!(JL_USB_EP3->RXCSR1 & BIT(0)));
  JL_USB_EP3->RXCSR1 = 0;
```

```
  ep3_radr0 = (u8 *)DMA_ADR0;
  for(i=0; i<64; i++)
  {
    if(*(ep3_radr0++) == 0xaa)
     JL_TIMER0->PRD = 0xe3;
    else
     JL_TIMER0->PRD = 0x404;
  }
  out_flag(9);
//tx
  ep3_tadr0 = (u8 *)DMA_ADR0;
  JL_USB->EP3_TADR = DMA_ADR0;
  for(i=0; i<64; i++)
  {
    *(ep3_tadr0++) = 0x58;
  }
  delay(100);
  JL_USB->EP3_CNT = 64;
   JL_USB_EP3->TXCSR1 = BIT(0);//tx:load paket to fifo
  while(JL_USB_EP3->TXCSR1 & BIT(0));
  out_flag(10);
#endif
}
```