



SolarA²

常见问题 FAQ

文档版本 01

发布日期 2024-12-20

版权所有 © 海思技术有限公司2024。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思和其他海思商标均为海思技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

海思技术有限公司

地址：上海市青浦区虹桥港路2号101室 邮编：201721

网址：<https://www.hisilicon.com/cn/>

客户服务邮箱：support@hisilicon.com



前言

概述

本文档主要介绍SolarA²解决方案中的常见问题和解决办法。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
SolarA ²	1.1.0

读者对象

本文档（本指南）主要适用于以下工程师：



- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
危险	表示如不可避免则将会导致死亡或严重伤害的具有高等级风险的危害。
警告	表示如不可避免则可能导致死亡或严重伤害的具有中等级风险的危害。
注意	表示如不可避免则可能导致轻微或中度伤害的具有低等级风险的危害。



符号	说明
 须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

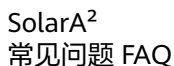
修订记录

修订日期	版本	修订说明
2024-08-23	00B01	第1次临时版本发布。
2024-12-20	01	第1次正式版本发布。



目录

前言.....	i
1 模拟模块使用类.....	1
1.1 ADC.....	1
1.1.1 ADC 采样转换完成的标志寄存器，查询后需要进行置位恢复为 0.....	1
1.1.1.1 问题描述.....	1
1.1.1.2 解决办法.....	1
1.2 GPIO.....	1
1.2.1 GPIO 初始化同组不同 PIN 注册中断问题.....	1
1.2.1.1 问题描述.....	2
1.2.1.2 解决办法.....	2
2 sample 使用类.....	3
2.1 调试打印.....	3
2.1.1 调用打印接口，无法正常打印.....	3
2.1.1.1 问题描述.....	3
2.1.1.2 解决办法.....	3
3 其他注意事项.....	4
3.1 寄存器读写不符合预期、读写后出现未定义行为.....	4
3.1.1 IP 寄存器读写前必须 CRG 使能.....	4
3.1.1.1 问题描述.....	4
3.1.1.2 解决办法.....	4
3.1.2 错误的寄存器访问操作.....	4
3.1.2.1 问题描述.....	4
3.1.2.2 解决办法.....	5
3.2 对寄存器读取后进行移位操作，不符合预期.....	6
3.2.1 问题描述.....	7
3.2.2 解决办法.....	7
3.3 全局变量定义使用，赋值不生效.....	7
3.3.1 问题描述.....	7
3.3.2 解决办法.....	7
3.4 JTAG 管脚相关复选功能配置不生效.....	8
3.4.1 问题描述.....	8
3.4.2 解决办法.....	8



3.5 串口首次烧录打印出现乱码.....	8
3.5.1 问题描述.....	8
3.5.2 解决办法.....	8
3.6 硬件配置不当引起 I2C 通信失败.....	8
3.6.1 问题描述.....	8
3.6.2 解决办法.....	9
3.7 栈空间分配不当导致栈溢出踩内存异常卡死.....	9
3.7.1 问题描述.....	9
3.7.2 解决办法.....	9
3.8 请求低功耗模式后执行其他代码，存在未知异常行为.....	11
3.8.1 问题描述.....	11
3.8.2 解决办法.....	12
3.9 APT 在设置边界比较点存在延迟一个周期发波的问题.....	12
3.9.1 问题描述.....	12
3.9.2 解决办法.....	12
3.10 在中断内关闭中断，下次开启中断时，导致中断进入次数异常（以 APT 为例）.....	12
3.10.1 问题描述.....	12
3.10.2 解决办法.....	14
3.11 APT 四种典型死区配置波形.....	15
3.11.1 问题描述.....	15



插图目录

图 3-1 stackAnalyze 栈分析工具示意图..... 10

图 3-2 imageAnalyze 镜像分析工具示意图..... 10

图 3-3 手动修改 LDS 文件配置示意图..... 11

图 3-4 IDE 配置 LDS 栈空间大小示意图..... 11

图 3-5 apt 定时中断示例代码..... 13

图 3-6 enable apt 定时中断示例代码..... 13

图 3-7 标志位示例代码..... 14

图 3-8 APT 中断流程图..... 14

图 3-9 A_HIGH_B_LOW..... 15

图 3-10 A_LOW_B_HIGH..... 16

图 3-11 A_HIGH_B_HIGH..... 16

图 3-12 A_LOW_B_LOW..... 17



表格目录

表 3-1 8/16bit 读写需要注意的寄存器列表..... 5



1 模拟模块使用类

1.1 ADC

1.1.1 ADC 采样转换完成的标志寄存器，查询后需要进行置位恢复为0

1.1.1.1 问题描述

直接读取ADC某个SOC的采样完成标志寄存器，直接读寄存器后没有恢复为默认值。

1.1.1.2 解决办法

- 解决办法1：使用HAL接口。
使用HAL接口HAL_ADC_CheckSocFinish()时，查询后会返回采样状态，内部已经实现了置位操作。
HAL_ADC_CheckSocFinish返回值：
 - BASE_STATUS_ERROR：表示SOC没有采样完成。
 - BASE_STATUS_OK：表示SOC采样完成。
- 解决办法2：使用DCL接口。
使用DCL接口时，需要两个接口配合完成，先使用DCL_ADC_GetConvState()接口获取状态，再使用DCL_ADC_ResetConvState()进行置位。
DCL_ADC_GetConvState返回值：
 - 返回值等于0：表示SOC没有采样完成。
 - 返回值不等于0：表示SOC采样完成。

1.2 GPIO

1.2.1 GPIO 初始化同组不同 PIN 注册中断问题



1.2.1.1 问题描述

同一组GPIO的不同pin分别注册中断回调函数时，通过对每个pin命名一个handle进行操作，注册中断回调的时候对每个pin重命名的handle进行注册，出现中断只响应同组中的一个pin问题，同组的其余pin的回调函数不响应。

例如：

```
g_led1.baseAddress = GPIO0;
g_led1.pins = GPIO_PIN_0;
HAL_GPIO_Init(&g_led1);
HAL_GPIO_SetDirection(&g_led1, g_led1.pins, GPIO_OUTPUT_MODE);
HAL_GPIO_SetValue(&g_led1, g_led1.pins, GPIO_LOW_LEVEL);
HAL_GPIO_SetIrqType(&g_led1, g_led1.pins, GPIO_INT_TYPE_RISE_EDGE);
HAL_GPIO_RegisterCallBack(&g_led1, GPIO_PIN_0, GPIO_CallBackFunc);
IRQ_Register(IRQ_GPIO0, HAL_GPIO_IrqHandler, &g_led1);
IRQ_SetPriority(IRQ_GPIO0, 1);
IRQ_EnableN(IRQ_GPIO0);

g_led2.baseAddress = GPIO0;
g_led2.pins = GPIO_PIN_1;
HAL_GPIO_Init(&g_led2);
HAL_GPIO_SetDirection(&g_led2, g_led2.pins, GPIO_OUTPUT_MODE);
HAL_GPIO_SetValue(&g_led2, g_led2.pins, GPIO_LOW_LEVEL);
HAL_GPIO_SetIrqType(&g_led2, g_led2.pins, GPIO_INT_TYPE_RISE_EDGE);

HAL_GPIO_RegisterCallBack(&g_led2, GPIO_PIN_1, GPIO_CallBackFunc);
IRQ_Register(IRQ_GPIO0, HAL_GPIO_IrqHandler, &g_led2);
IRQ_SetPriority(IRQ_GPIO0, 1);
IRQ_EnableN(IRQ_GPIO0);
```

1.2.1.2 解决办法

合并同组GPIO下不同pin的handle名称为一个名称。

由于中断注册的时候每个中断号只会注册第一个回调函数，如果采用上述方式进行同组GPIO不同pin的初始化，会在注册回调函数的时候导致第二个pin的回调函数失效，正确的解法如下，合并同组不同pin的handle名称。

```
g_led.baseAddress = GPIO0;
g_led.pins = GPIO_PIN_0 | GPIO_PIN_1;
HAL_GPIO_Init(&g_led);
HAL_GPIO_SetDirection(&g_led, g_led.pins, GPIO_OUTPUT_MODE);
HAL_GPIO_SetValue(&g_led, g_led.pins, GPIO_LOW_LEVEL);
HAL_GPIO_SetIrqType(&g_led, g_led.pins, GPIO_INT_TYPE_RISE_EDGE);

HAL_GPIO_RegisterCallBack(&g_led, GPIO_PIN_0, GPIO_CallBackFunc);
HAL_GPIO_RegisterCallBack(&g_led, GPIO_PIN_1, GPIO_CallBackFunc);
IRQ_Register(IRQ_GPIO0, HAL_GPIO_IrqHandler, &g_led);
IRQ_SetPriority(IRQ_GPIO0, 1);
IRQ_EnableN(IRQ_GPIO0);
```



2 sample 使用类

2.1 调试打印

2.1.1 调用打印接口，无法正常打印

2.1.1.1 问题描述

DBG_PRINTF_USE宏定义定义为DBG_USE_UART_PRINTF后，DBG_PRINTF()默认使用UART进行打印，此时直接调用打印接口DBG_PRINTF()，没有打印输出。

2.1.1.2 解决办法

默认打印是通过UART来实现，所以在使用打印DBG_PRINTF前请确保已经用HAL_UART_Init完成初始化串口。

此外，其他需要注意的内容如下：

- 如果需要使能打印，请包含头文件“debug.h”。
- 需要将feature.h中的DBG_PRINTF_USE宏定义定义为DBG_USE_UART_PRINTF，否则没有串口打印输出。
- 为了让UART可以正常输出，请确保UART在IOConifg()调用后，再进行初始化。否则UART会因为未初始化而不能正常执行。
- 支持UART打印，SDK默认为UART0输出打印。若需要修改，在feature.h修改DBG_PRINTF_UART_PORT宏定义，来选择其他串口。



3 其他注意事项

3.1 寄存器读写不符合预期、读写后出现未定义行为

本章主要描述寄存器读写的限制，寄存器读写访问不同于普通的RAM，必须使用32bit的形式来进行读写，否则有可能因编译器不同或者编译选项不同出现异常行为，包括写不成功或者总线访问出错的现象。

3.1.1 IP 寄存器读写前必须 CRG 使能

3.1.1.1 问题描述

直接对某个模块的寄存器进行写操作，然后再读取，出现写操作的数值和读取的数值不一致。

3.1.1.2 解决办法

访问IP寄存器前，必须通过HAL_CRG_IpEnableSet将该IP时钟使能，否则会出现读写不成功的情况，以UART为例，以下为使能UART0时钟的操作。

```
static void UART0_Init(void)
{
    HAL_CRG_IpEnableSet(UART0_BASE, IP_CLK_ENABLE);
    ....
}
```

以上代码为UART0模块的时钟使能。

3.1.2 错误的寄存器访问操作

3.1.2.1 问题描述

按8/16bit读写寄存器，可能导致程序行为不可知。

出错示例：

```
/* I2C寄存器定义 */
typedef union {
```



```
unsigned int reg;
struct {
    unsigned int i2c_enable      : 1;
    unsigned int reserved0      : 7;
    unsigned int sda_hold_duration : 16;
    unsigned int reserved1      : 8;
} BIT;
} I2C_GLB_REG;

typedef struct {
    I2C_GLB_REG      I2C_GLB;          /**< Offset Address: 0x0000. */
    ...
} volatile I2C_RegStruct;

/* 错误示例代码 */
I2C_RegStruct *i2cReg = address; /* address是I2C寄存器基地址 */
unsigned int duration = i2cReg->I2C_GLB.BIT.sda_hold_duration; /* 直接对地址按16bits进行读操作 */
```

如上的寄存器操作中，sda_hold_duration在寄存器中占16bits，如果直接对读操作，会造成程序异常。

3.1.2.2 解决办法

先将32bit的寄存器读取到变量中，再对变量按位读取，避免对寄存器直接进行8/16bit读写操作。除了示例中的寄存器外，用户在读写8/16bit时，需要注意的寄存器列表如表3-1所示。

正确代码写法：

```
I2C_RegStruct *i2cReg = address; /* address是I2C寄存器基地址 */
I2C_GLB_REG glb;
unsigned int duration;
glb.reg = i2cReg->I2C_GLB.reg; /* 将寄存器读到32位的结构体变量中 */
duration = glb.BIT.sda_hold_duration; /* 对32位的结构体变量再进行读操作 */
```

表 3-1 8/16bit 读写需要注意的寄存器列表

模块名	寄存器名
I2C	I2C_GLB
	I2C_DEV_ADDR
	I2C_DATA_BUF
	I2C_PATTERN_DATA1
	I2C_PATTERN_DATA2
DAC	DAC_CTRL
CAN	IF1_DATA_A1
	IF1_DATA_A2
	IF1_DATA_B1
	IF1_DATA_B2



模块名	寄存器名
	IF2_DATAA1
	IF2_DATAA2
	IF2_DATAB1
	IF2_DATAB2
FLASH	EFLASH_CAPACITY_1
	BUF_CLEAR
PMC	LOWPOWER_STATUS
SPI	SPICR0
APT	EM_WD_CNT
	EM_WD_STS
	EM_VCAP_STS1
	EM_TCAP_VAL
ADC	ADC_INT1_CTRL
	ADC_INT2_CTRL
	ADC_INT3_CTRL
	ADC_INT4_CTRL
	ADC_PPBO_PPBI_DLY
	ADC_PPBI_PPBO_DLY
CMM	CMVER
CFD	CFDVER
QDM	QDMVER
SYSCTRL	SC_RST_CNT0
	SC_RST_CNT1
	APT_POE_FILTER
	APT_EVTIO_FILTER
	APT_EVTMP_FILTER

3.2 对寄存器读取后进行移位操作，不符合预期

寄存器读写访问不同于普通的RAM，必须使用32bit的形式来进行操作，否则有可能因编译器不同或者编译选项不同出现异常行为。



3.2.1 问题描述

对寄存器地址读出数据后赋值给局部变量，再对局部变量进行移位操作，结果不符合预期。产生这个结果的原因是，局部变量如果没有在其他地方使用，编译器有可能把局部变量优化掉，变成对寄存器直接进行移位操作，导致异常。

3.2.2 解决办法

在对寄存器地址操作时，为了避免被工具链优化，在对寄存器地址添加关键字“volatile”，阻止编译器对寄存器的读、写、移位等操作进行不符合预期的优化。

3.3 全局变量定义使用，赋值不生效

3.3.1 问题描述

全局变量使用，在中断中给该全局变量赋值，在中断外再读取此全局变量，中断中赋值操作不生效。

错误使用示例：

```
static unsigned int g_flag; /* 定义全局变量 */
.....
void InterruptCallBack(void)
{
    g_flag = 1;             /* 中断中修改标志位 */
}

int main (void)
{
    .....
    while ( g_flag == 1 ) { /* 中断外读取全局变量 */
        .....
    }
}
```

3.3.2 解决办法

如果全局变量在中断中进行更新，而且其他位置进行了读操作，请确保用volatile来修饰该全局变量。否则读该全局变量的处理可能被优化，未按预期循环读取。

正确示例：

```
static volatile unsigned int g_flag; /* 定义全局变量，添加了volatile进行修饰 */
.....
void InterruptCallBack(void)
{
    g_flag = 1;             /* 中断中修改标志位 */
}

int main (void)
{
    .....
    while ( g_flag == 1 ) { /* 中断外读取全局变量 */
        .....
    }
}
```



3.4 JTAG 管脚相关复选功能配置不生效

3.4.1 问题描述

例如：GPIO1_3/GPT1_PWM/CAPM2_SRC0/UART1_RXD/I2C0_SCL与JTAG_TDO为同一管脚的不同复用功能，当管脚功能复选为非JTAG_TDO的其他功能时无法正常工作，如：UART1_RXD接收不到数据，GPIO1_3翻转不生效，GPT1_PWM无法产生PWM。

复用关系如下：

GPIO1_3/JTAG_TDO/GPT1_PWM/CAPM2_SRC/UART1_RXD/I2C0_SCL 复用
IOCMG14寄存器。

3.4.2 解决办法

由于系统启动时，boot引脚电平决定当前系统处于正常启动或升级模式，而试制阶段考虑调试升级功能，boot管脚外围电路采用上拉设计使系统在启动时处于升级模式，则单板上电复位后，JTAG相关管脚强制为JTAG功能，且无法通过软件切换为其他复用功能。从而导致JTAG端口复用功能配置失效，需要更改boot管脚外围电路为下拉设计使系统状态寄存器处于正常启动模式或通过写系统升级清除标识寄存器，然后再配置端口复用。写系统升级清除标识寄存器的代码如下。与JTAG管脚相关的复选功能均存在以上问题，考虑到量产安全和JTAG调试功能，因此建议此解决办法只适用于试制和实验室调试阶段。

```
SYSCTRL0->SC_SYS_STAT.BIT.update_mode_clear = 1;
```

3.5 串口首次烧录打印出现乱码

3.5.1 问题描述

通过IDE串口烧录文件后，如果程序运行之初就有大量打印，可能出现乱码。通过按复位键硬复位才恢复正常。

问题的原因在于：串口在PC串口打开前已经发送了大量数据，导致串口FIFO满，使得pc串口软件打开后，接收数据时因为数据丢失，导致bit错位。

3.5.2 解决办法

单击复位键重新执行程序。

3.6 硬件配置不当引起 I2C 通信失败

3.6.1 问题描述

在软件正确配置I2C模块初始化后，通过I2C通信管脚与对接设备进行通信，可能出现无数据信号发出或通信过程中时钟紊乱，导致通信失败。



3.6.2 解决办法

- 通信过程中无数据信号发出：
 - I/O复用功能需配置为I2C通信管脚。在使用I2C管脚进行通信时，需将I/O复用关系配置为I2C_SCL和I2C_SDA功能，并且通信需使用此两个管脚进行通信；
 - I2C_SCL管脚和I2C_SDA管脚需连接外部上拉电阻。由于I2C通信管脚是开漏输出，无电平拉高能力，在I2C通信时I2C_SCL管脚和I2C_SDA管脚需要连接外部上拉电阻。
- 通信过程中时钟紊乱现象：

I2C通信时不仅需要连接通信两端的SCL和SDA线，还需连接GND管脚，保证通信两端设备共地。

3.7 栈空间分配不当导致栈溢出踩内存异常卡死

3.7.1 问题描述

栈空间分配不合理导致软件运行过程中异常卡死或者概率性异常卡死，程序无法正常运行。需合理分配栈空间大小。

3.7.2 解决办法

栈空间分配理论分析方法：

针对栈溢出场景，需结合应用分析可能需要的最大栈空间，然后进行合理的栈空间分配。最大栈空间分配方法可以参照如下估算公式：

$$\text{MaxStackSize} = \text{mainfunc}(\text{stackSize}) + \text{redundance}(\text{stackSize}) + \text{intHandler}(\text{stackSize}) * n + \sum_{i=1}^n \text{MaxCallbackfunc}(\text{stackSize})_i$$

MaxStackSize：最大估算栈开销。

mainfunc(stackSize)：main函数最大栈开销，用栈分析工具查找pmp_init展开后查询main函数中中断使能后的函数最大栈开销。

redundance(stackSize)：冗余栈开销，防止栈溢出。

n：最大中断嵌套层数，与使用中断优先级数量相等；n=0时表示无中断。

intHandler(stackSize)：中断栈空间开销，栈分析工具中查找intHandler的栈空间开销。

MaxCallback(stackSize)_i：取第i级多个中断回调函数中最大栈空间开销，栈分析工具中分别查找对应callback函数的栈空间开销，同级取最大值。

估算示例：以SDK sample中的pmsm_sensorless_2shunt_foc代码为例进行分析，结合IDE自带的栈分析工具和镜像分析工具分析结果如图3-1和图3-2所示。

图 3-1 stackAnalyze 栈分析工具示意图

函数名称	深度	最大开销(Byte)	本地开销(Byte)
pmp_init	13	320	0
Chip_Init	10	320	32
main	12	256	16
MotorCarrierProcessCallback	5	208	16
CheckPotentiometerValueCallba...	6	192	16
MotorSysErrCallback	6	192	16
MotorStartStopKeyCallback	10	144	16
NmiEntry	2	112	112
TrapEntry	2	112	112
MotorStatemachineCallBack	3	96	48

图 3-2 imageAnalyze 镜像分析工具示意图

区域	起始地址	结束地址	容量	可用	已用
RAM_CODE	0x000000002000000	0x02000000	0 B	0 B	0 B
RAM_RESERVE_DATA	0x0000000004000000	0x04000000	0 B	0 B	0 B
RAM_DATA	0x0000000004000000	0x04002800	10.00KB	5.93KB	4.07KB
RAM_STACK	0x0000000004002800	0x04004000	6.00KB	5.00KB	1.00KB
FLASH_MAGIC	0x0000000003000000	0x03000004	4 B	0 B	4 B
FLASH_CODE	0x0000000003000004	0x03028000	160.00KB	131.88KB	28.12KB

1. 程序有5个中断，3个优先级，即n=3；
2. 通过栈开销工具分析可知：
mainfunc(stackSize)=256Byte；
redundance(stackSize)=200Byte，用户自定义；
intHandle(stackSize)=164Byte。

分析3类优先级回调函数最大栈开销分别为：

- 中断优先级7：208Byte（MotorCarrierProcessCallback）。
- 中断优先级6：192Byte（MotorSysErrCallback）。
- 中断优先级1：192Byte、144Byte、96Byte；
（CheckPotentiometerValueCallback、MotorStartStopKeyCallback、MotorStatemachineCallBack），取最大值192Byte。
 $\sum(\text{MaxCallbackfunc}(\text{stackSize})_i) = 208 + 192 + 192 = 592\text{Byte}$ 。
代入公式计算可知：

MaxStackSize=1540Byte；可基于以上估算结果进行栈空间内存分配。

- 通过镜像分析工具可知，RAM空间使用了4.07KB，剩余5.93KB，栈空间溢出到内存数据的空间还比较大，踩内存导致数据跳变的机率较小，栈空间可调范围较大。

栈空间配置调整方法：

- 手动修改LDS文件配置方法，如图3-3所示。

图 3-3 手动修改 LDS 文件配置示意图

```
RAM_START = RAM_RESERVE_DATA_START + RAM_RESERVE_DATA_SIZE + RAM_DIAGNOSE_BUF_SIZE;
RAM_SIZE  = 0x5000 - RAM_CODE_SIZE - RAM_RESERVE_DATA_SIZE - RAM_DIAGNOSE_BUF_SIZE - STACK_SRAM_BOUND_SIZE;
RAM_END   = SRAM_END;

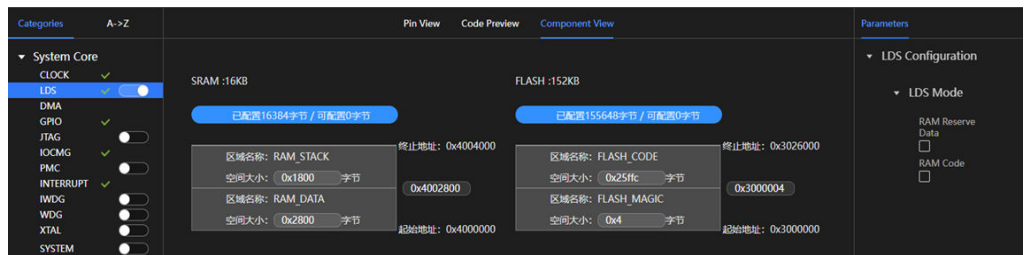
STACK_SRAM_BOUND_START = RAM_START + RAM_SIZE;

STACK_START = STACK_SRAM_BOUND_START + STACK_SRAM_BOUND_SIZE;

NMI_STACK_SIZE = 1024;
STACK_SIZE      = 0x3000 - NMI_STACK_SIZE;
INIT_STACK_SIZE = 1024;
```

- IDE工具配置LDS空间大小方法，如图3-4所示。

图 3-4 IDE 配置 LDS 栈空间大小示意图



须知

手动修改LDS文件时，RAM+STACK空间大小总和不要超过芯片最大的SRAM区间大小，具体SRAM区间大小请查看芯片对应的数据手册中定义的大小，STACK空间最小需保障函数正常运行和中断正常压栈。

3.8 请求低功耗模式后执行其他代码，存在未知异常行为

3.8.1 问题描述

调用DCL_PMC_EnterDeepSleep()或调用DCL_PMC_EnterShutDown()接口后，芯片继续执行后续代码或中断代码，存在未知异常行为。



3.8.2 解决办法

1. 调用HAL_PMC_EnterDeepSleepMode()或HAL_PMC_EnterShutdownMode()接口代替DCL层接口；
2. 在调用DCL_PMC_EnterDeepSleep()或DCL_PMC_EnterShutDown()接口前关闭所有中断，接口后执行while(1)等待芯片切换功耗模式完成。

3.9 APT 在设置边界比较点存在延迟一个周期发波的问题

3.9.1 问题描述

配置：计数方式不限（up down/up/down），APT的加载模式为缓存加载，加载点为零点。

期望动作：当用户在t(n)周期设置占空比指令为100%时，会配置比较点C = 0，D = 0；期望在t(n+1)会输出100% 的PWM波。

现实情况：芯片会延迟一个周期发波。

原因：芯片底层逻辑是在第0拍加载比较点，在第1拍生效。当用户在t(n)周期设置比较点C = 0，D = 0时，将在t(n+1)周期的计数值0点加载C点，但当C点加载完成后，APT实际计数值为1，此时APT在t(n+1)周期无法响应C点的动作，可以响应D点动作，将t(n+2)时刻响应C点动作。所以APT会在t(n+1)周期保持t(n)周期的动作。

3.9.2 解决办法

如果需要无延迟输出100% 占空比，建议配置C = 1，D = 0。

3.10 在中断内关闭中断，下次开启中断时，导致中断进入次数异常（以 APT 为例）

3.10.1 问题描述

apt的定时中断，如果在apt定时中断回调里disable掉apt的定时中断，延时一段时间再stop apt，然后再enable apt的定时中断，此时apt处于stop状态，但还是会进入apt的定时中断。

过程如下：

1. apt的定时中断，如果在apt定时中断回调里disable掉apt的定时中断；

图 3-5 apt 定时中断示例代码

```
void APT0TimerCallback(void *aptHandle)
{
    BASE_FUNC_UNUSED(aptHandle);
    a++;

    IRQ_DisableN(IRQ_APT0_TMR);
    /* USER CODE BEGIN APT0_TIMER_INTERRUPT */
    /* USER CODE END APT0_TIMER_INTERRUPT */
}

/* 建议用户定义全局变量、结构体、宏定义或函数声明等 */
/* USER CODE END 1 */

int main(void)
{
    /* USER CODE BEGIN 2 */
    /* 建议用户放置初始化代码或启动代码等 */
    /* USER CODE END 2 */
    SystemInit();
    /* USER CODE BEGIN 3 */
    HAL_APT_StartModule(RUN_APT0);
    BASE_FUNC_DELAY_S(1);
    /* USER CODE END 3 */
}
```

启动apt之后，进入定时中断会关闭定时中断

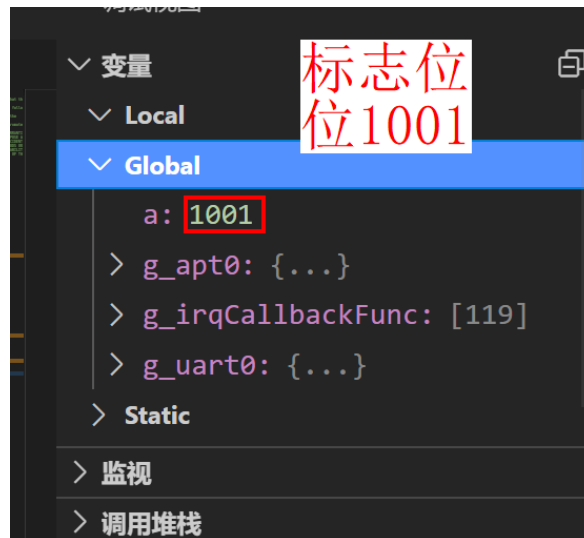
2. 延时一段时间再stop apt，然后再enable apt的定时中断；

图 3-6 enable apt 定时中断示例代码

```
HAL_APT_StopModule(RUN_APT0);
HAL_APT_PWMDeInit(&g_apt0);
a = 1000;
DBG_PRINTF("have stop\r\n");
// HAL_APT_RegisterCallBack(&g_apt0, APT_TIMER_INTERRUPT, APT0TimerCallback);
IRQ_EnableN(IRQ_APT0_TMR);
/* 建议用户放置初始配置代码 */
/* USER CODE END 3 */
while (1) {
    /* USER CODE BEGIN 4 */
    /* 建议用户放置周期性执行代码 */
    /* USER CODE END 4 */
}
```

3. 此时apt处于stop状态，但还是会进入apt的定时中断。

图 3-7 标志位示例代码



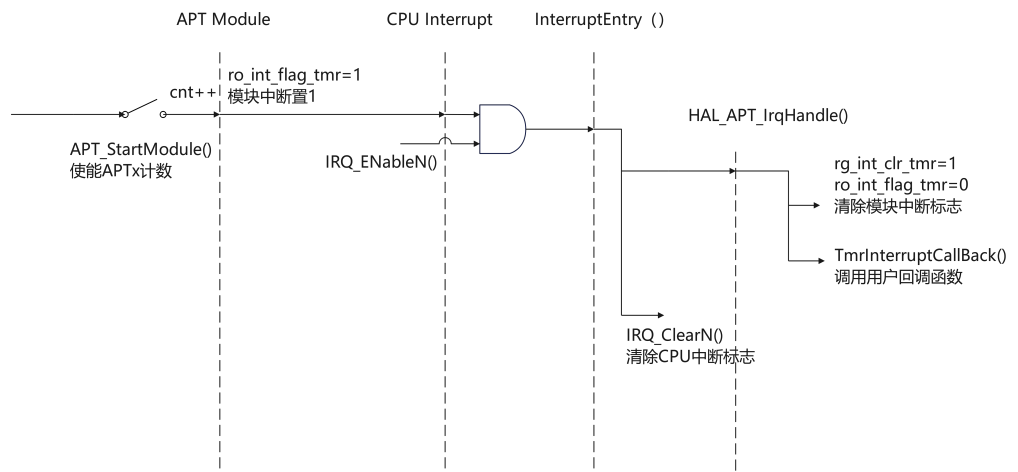
3.10.2 解决办法

- 原因说明

APT中断流程图如图3-8所示。

在关闭中断时，只将CPU的中断关闭了，但是模块仍在计数，当满足触发中断的条件时仍会产生APT_x->INIT_TMR_FLAG标志位，当再次使能中断IRQ_EnableN时，会立即回调中断函数，导致产生了与预期不相符的中断进入次数。

图 3-8 APT 中断流程图



- 解决办法

仅对中断关闭和开启（禁能和使能）操作，以APT0为例：

- 关闭中断：使用关闭CPU中断的接口：IRQ_DisableN(IRQ_APT0_TMR)；
- 开启中断：
 - 在开启中断前，先清除模块的中断标志位：APT0->INT_TMR_FLAG.BIT.rg_int_clr_tmr = 0x1；

- 使用开启CPU中断接口，使能中断：IRQ_EnableN(IRQ_APT0_TMR)。
- 对中断模块关闭和开启（禁能和使能）操作：
 - 关闭模块：使用关闭模块的接口：HAL_APT_StopModule(RUN_APT0)；
 - 开启模块：使用开启模块的接口：HAL_APT_StartModule(RUN_APT0)。

3.11 APT 四种典型死区配置波形

3.11.1 问题描述

APT模块提供了四种典型的死区波形配置形状：A_HIGH_B_LOW、A_LOW_B_HIGH、A_HIGH_B_HIGH、A_LOW_B_LOW。基本形状如下：

图 3-9 A_HIGH_B_LOW

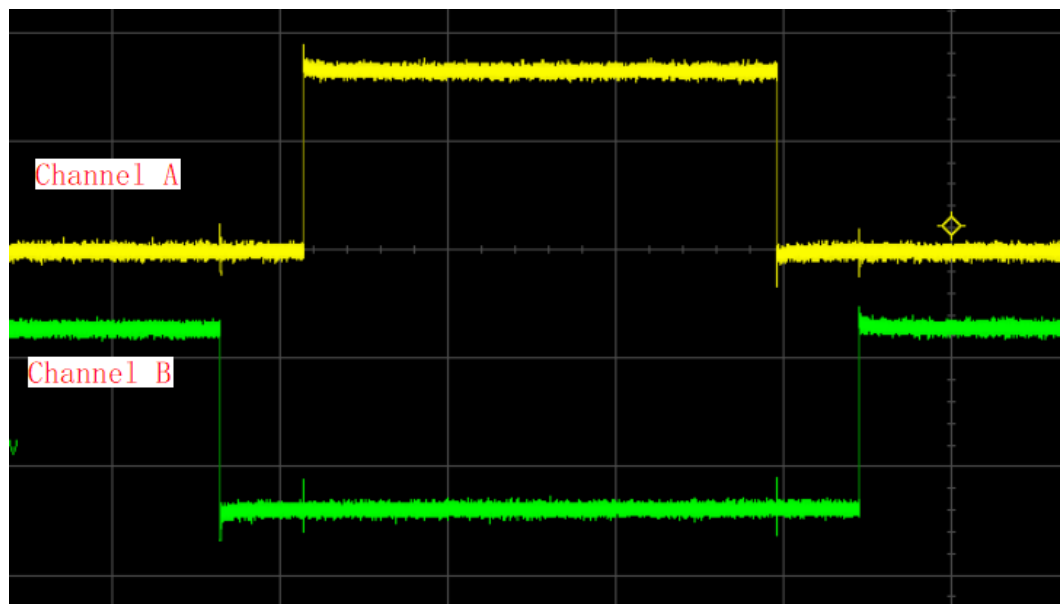




图 3-10 A_LOW_B_HIGH

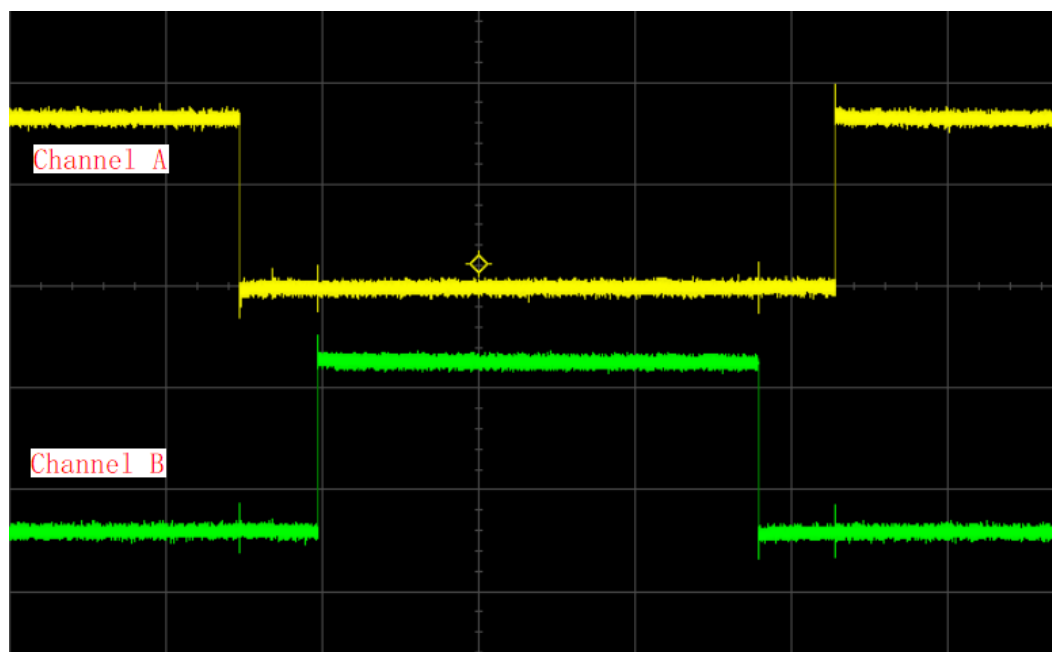


图 3-11 A_HIGH_B_HIGH

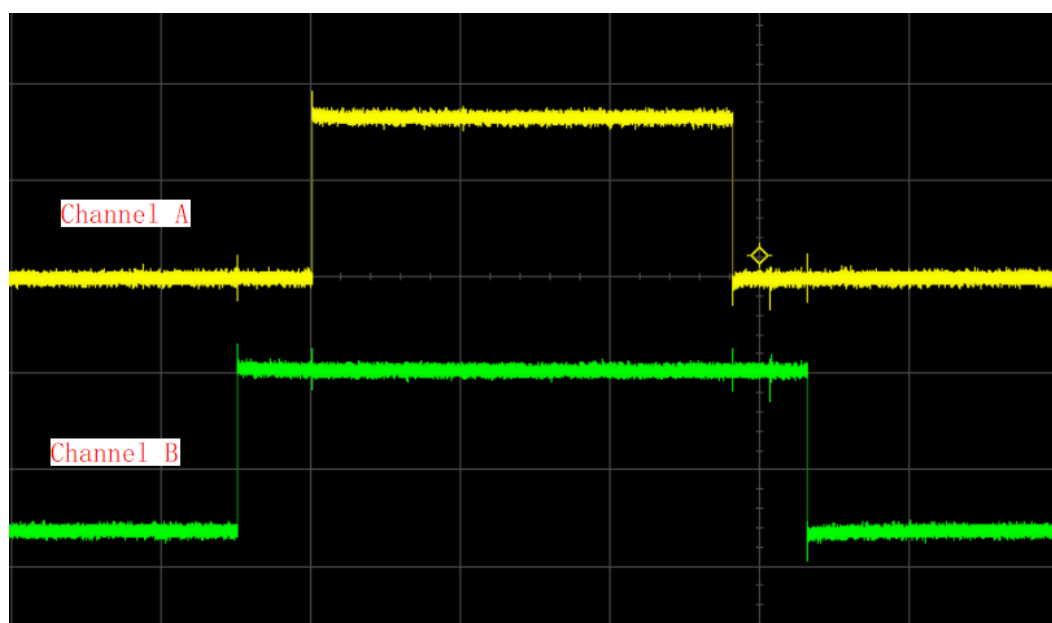




图 3-12 A_LOW_B_LOW

