SPINTROL

使用指南

SPC1169 Flash 使用指南

版本 A/1 – 2023 年 6 月

SPINTROL

目录

1 Flash 存储器概述 7

1.1 主要特征 7

1.2 访问接口 7

1.3 写保护 9

1.4 读保护 9

1.5 配置字 9

1.6 EEPROM 模拟 9

2 Flash 操作实例 11

2.1 设定时序参数 11

2.2 XIP 读操作 11

2.3 Flash 控制器操作 11

3 实际用例讲解 12

3.1 关闭 XIP 期间产生中断 12

3.2 调用第三方函数 API 16

SPINTROL

©2023 旋智电子科技（上海）有限公司

图片列表

图 1-1：Flash 存储器访问接口 7
图 1-2：Flash 存储器逻辑结构 8
图 3-1：加载域与执行域 17

SPINTROL

表格列表

表 1-1：Flash 存储器结构 8
表 1-2：配置字描述 9

SPINTROL

©2023 旋智电子科技（上海）有限公司

版本历史

版本	日期	作者	状态	变更
A/0	2023 年 4 月 20 日	CanChai	Outdated	首次发布。
A/1	2023 年 6 月 9 日	CanChai	Released	更新章节 1.5

SPINTROL

术语或缩写

术语或缩写	描述

SPINTROL

©2023 旋智电子科技（上海）有限公司

1 Flash 存储器概述

1.1 主要特征

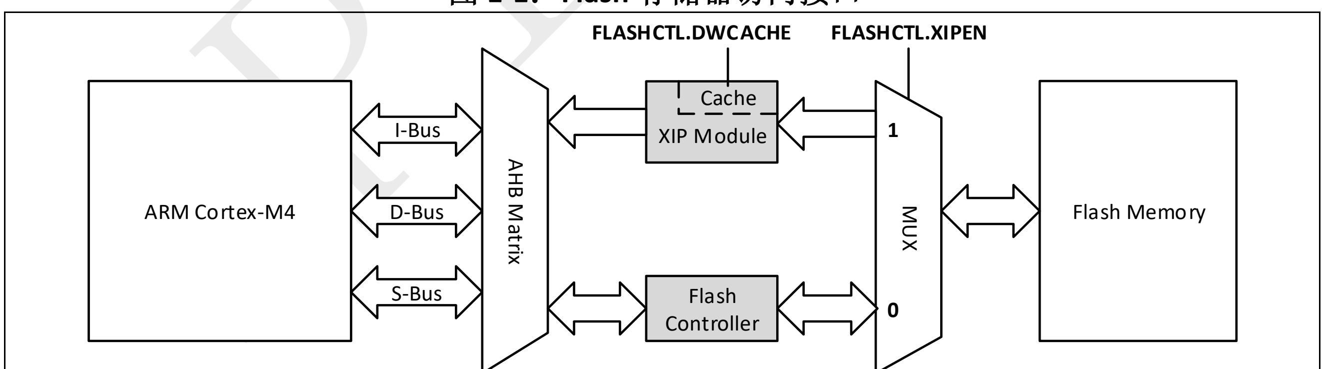
SPC1169/SPD1179/SPD1176 片内 Flash 存储器用于存储用户程序、数据等内容，其主要特点如下：

- 高达 128KB 存储空间
- 存储空间组成：
- 主存储空间：共 32 扇区，每个扇区大小为 512*64 bits
- NVR 空间：共 3 扇区，每个扇区 512*64 bits
- 单个扇区可擦除次数为：至少 100000 次@ TJ=85 oC
- 数据保存时间期限：超过 20 年@ TJ=85 oC
- ECC 保护
- 符合 AEC-Q100 一级标准

1.2 访问接口

SPC1169/SPD1179/SPD1176 存储器的访问接口有两个：XIP 模块和 Flash 控制器，如图 1-1：Flash 存储器访问接口所示。XIP 模块只能实现从 Flash 存储器中读取数据，用于 CPU 从 Flash 存储器中取代码和数据；Flash 控制器是接在 AHB 总线上的一个外设，可以实现 Flash 存储器的读、写以及擦除操作。但是，这两个接口不能同时工作，同一时刻只能有一个接口处于工作状态。Flash 存储器当前的访问接口由寄存器 FLASHCTL.XIPEN 进行控制。

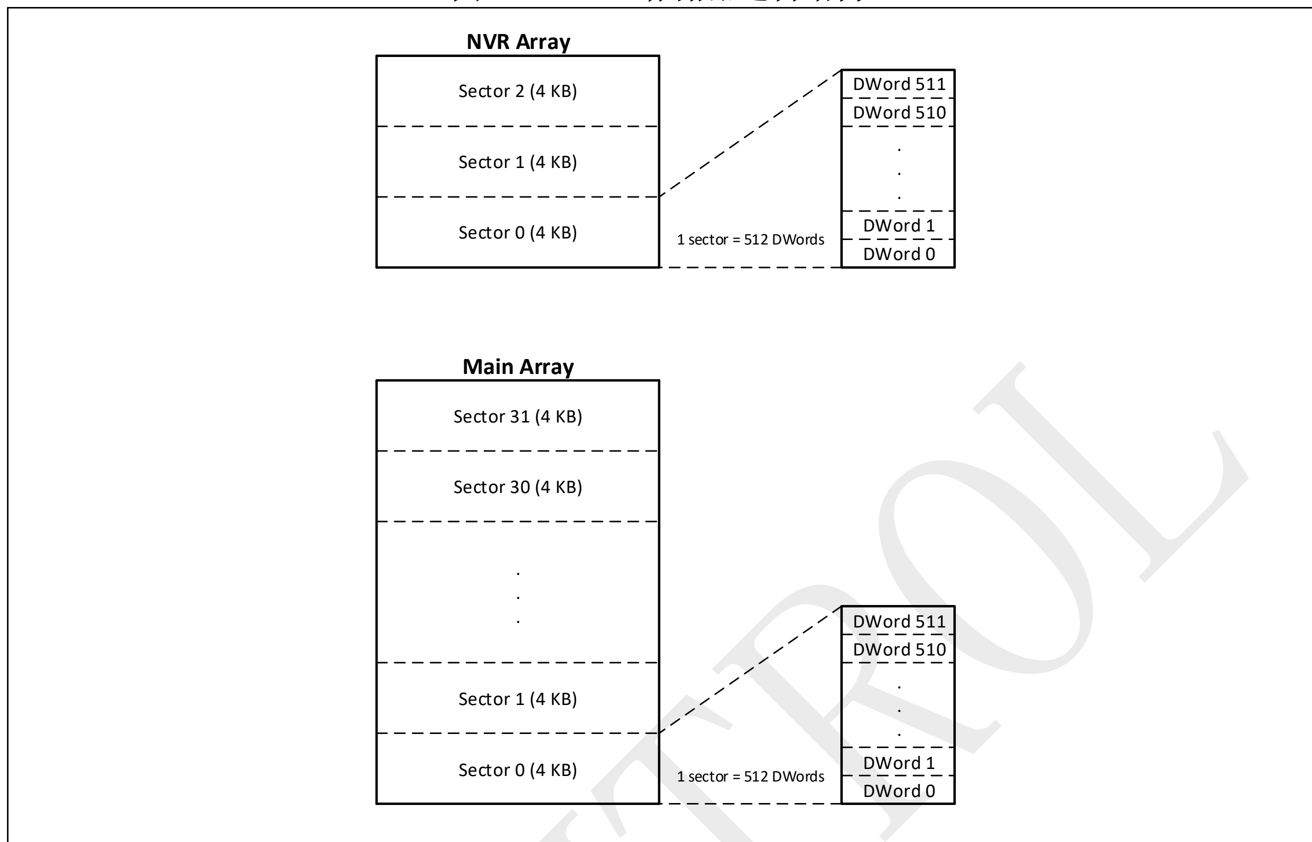
图 1-1：Flash 存储器访问接口



SPC1169/SPD1179/SPD1176 Flash 存储器的构成如图 1-2：Flash 存储器逻辑结构和表 1-1：Flash 存储器结构所示，包含两个部分：

- 主存储单元：这部分由 32 个扇区组成，每个扇区的大小为 4096 字节，共计 128KB；
- NVR 存储单元：这部分由 3 个扇区组成，每个扇区的大小为 4096 字节。

图 1-2：Flash 存储器逻辑结构



（1）对于容量为 128KB 的 Flash 存储器，主存储器单元只有 32 个扇区（Sector0~Sector31）。

表 1-1：Flash 存储器结构

存储区块	名称	起始地址	大小
主存储单元	Sector 0	0x1000 0000	4096 字节
	Sector 1	0x1000 1000	4096 字节
	Sector 2	0x1000 2000	4096 字节
	Sector 3	0x1000 3000	4096 字节
	…	…	…
	Sector 31	0x1001 F000	4096 字节
NVR 存储单元	Sector 0	0x1100 2000	4096 字节
	Sector 1	0x1100 3000	4096 字节
	Sector 2	0x1100 4000	4096 字节

©2023 旋智电子科技（上海）有限公司

1.3 写保护

Flash 主存储区可以设置保护，以防止意外写入，写保护最小粒度为扇区。如果对设置了保护的扇区执行编程或擦除操作，则操作将被忽略。

写保护通过设置 FLASHWP 寄存器对应比特位激活，重置寄存器对应比特位将关闭写保护。

1.4 读保护

SPC1169/SPD1179/SPD1176 提供了 Flash 存储器的读保护功能，该保护功能可以防止通过调试接口读取和修改 Flash 存储器，这种保护在所有设备操作模式下均有效。

读保护通过设置配置字区域 CHIP_SECURITY 字段而使能，重新复位系统加载新的配置字后即可激活。

若要禁用读取保护：擦除整个 Flash 存储区，在整个擦除过程中，读取保护仍处于启用状态。

1.5 配置字

有两个配置字可供使用，根据应用需求由最终用户进行配置。配置字位于主 Flash 存储器的末尾（最后一个 DWord），即 0x1001FFF8 处。

这些配置字的描述如表 1-2：配置字描述所示，新编程的配置字在上电复位后加载并激活。

表 1-2：配置字描述

地址	名字	描述
0x1001FFF8	CHIP_SECURITY	锁住芯片调试接口配置字 0xFFFFFFFF：芯片调试接口将不会锁住 其它：芯片调试接口将会锁住
0x1001FFFC	WDT_ENABLE	看门狗使能配置字 0xFFFFFFFF：芯片启动时关闭看门狗 其它：芯片启动时使能看门狗

1.6 EEPROM 模拟

EEPROM 是许多嵌入式应用中需要在运行时以字节或字为粒度更新的非易失性存储数据的重要组件。为了消除元件、节省 PCB 空间和降低系统成本，可以使用嵌入式 Flash 存储器代替外部 EEPROM 进行数据存储。SPC1169/SPD1179/SPD1176 实现了一种模拟软件方案，将三个 NVR 扇区模拟为 EEPROM 存储器。模拟的 EEPROM 支持最多 256 个字的数据存储，虚拟地址为 0~255。Boot ROM 提供以下接口来访问模拟的 EEPROM：

– EEPROM_Init()

此函数尝试将模拟的 EEPROM 恢复到已知的良好状态，并在访问 EEPROM 之前以及每次断电重启后，都应调用此函数。

– EEPROM_Format()

此函数擦除所有三个 NVR 扇区，并将 NVR 扇区 0 设置为有效。

- EEPROM_WriteWord()

  用户应用程序调用此函数以更新数据。

- EEPROM_ReadWord()

  此函数返回与传递的虚拟地址相对应的最新的数据。

详细的 EEPROM 信息请参考《RC-032_SPC1169_SPD1179_SPD1176 模拟 EEPROM 使用指南》。

SPINTROL

©2023 旋智电子科技（上海）有限公司

2 Flash 操作实例

2.1 设定时序参数

Flash 正常工作时需要满足一定的物理时序要求，用户可以调用 pHWLIB->FLASHC_SetTiming()函数对 Flash 时序进行设置。每当用户计划更改 HCLK 频率时，都应在频率更改之前调用 FLASHC_RelaxXIPTiming()，以宽松的时序要访问 Flash。在频率更改后，应调用 pHWLIB->FLASHC_SetTiming()函数以设置优化的 Flash 时序。

2.2 XIP 读操作

通过 XIP 模块，可以实现对 Flash 存储器的读操作。用户可以直接对 Flash 进行字节、半字或者字形式的读取。下面是一个示例代码，从 Flash 中地址 0x10000100 处读取一个字节数据。

示例代码

```
uint8_t *pu8Data = (uint8_t *)0x10000100;
uint8_t u8Byte;

u8Byte = *pu8Data;
```

2.3 Flash 控制器操作

在 Spintrol 第三代产品中，由于 Flash 驱动没有放在 ROM，当用户代码（包括 Flash 驱动）在 Flash 存储器中运行时，此时若想向 Flash 写入数据或者擦除 Flash，需要先关闭 XIP，这将导致 CPU 无法通过 XIP 取到指令，导致内核挂死。解决这个问题的办法是在 IDE 中进行配置，将 Flash 的驱动放在 RAM 中，但这些配置较为繁琐，在特殊情况下会变的更为麻烦.

为了解决这种问题，在 SPC1169/SPD1179/SPD1176 的 Flash 驱动代码都放在了 ROM 空间，并开放了可供用户使用的接口，详细接口信息请参考 SDK 库函数，在此就不再列举相关示例代码。

SPINTROL

3 实际用例讲解

3.1 关闭 XIP 期间产生中断

由于在擦除 Flash 的过程中，会关闭 XIP 通道，此时 CPU 无法通过 XIP 通道拿到存储在 Flash 中的代码指令。在实际应用中我们可能会遇到一个场景：在擦除 Flsh 的时候，发生了中断，此时如果不做特殊处理，CPU 将无法拿到存储在 Flash 中的中断向量表信息，这将会造成处理器异常。在 SPC1169 平台相关的产品的 SDK 示例代码中提供了正确处理此种情形的示例代码，名为《Flash_With_ISR_In_RAM》，以下将以 SPC1169 的 KEIL 工程示例代码细节讲解需要关注的细节。

如下代码演示在主动关闭 XIP 的情况下，如果发生 TIMER1 中断，在实际工程中将如何处理。

示例代码

```
#include <stdio.h>
#include "spc1169.h"

#if defined ( __CC_ARM )
#define             _RAM_FUNC_
#else
#define             _RAM_FUNC_        __ramfunc
#endif

#define             VectorTableLoadAddr                          0x10000000
#define             TimerPeridCount                              1 //1ms
#define             VectorTableRelocateAddr                      0x1FFFF000

uint32_t u32VectorCnt = 0;
static uint32_t u32Count = 0;

int main(void)
{
    CLOCK_InitWithRCO(CLOCK_CPU_100MHZ);

    Delay_Init();

    /*
     * Init the UART
     */
    PIN_SetChannel(PIN_GPIO10, PIN_GPIO10_UART0_TXD);
    PIN_SetChannel(PIN_GPIO11, PIN_GPIO11_UART0_RXD);
    UART_Init(UART0, 38400);

    /* Copy vector table to RAM */
    for(u32VectorCnt = 0; u32VectorCnt < 79; u32VectorCnt++)
    {
        *((volatile uint32_t *)(VectorTableRelocateAddr + u32VectorCnt * 4)) = 
\
                                    *((volatile uint32_t *)(VectorTableLoadAddr + 
u32VectorCnt * 4));
    }

    /* Redirection the Vector Address */
    SCB->VTOR = VectorTableRelocateAddr;
    printf("Exception vector table Addr is : %x\n", SCB->VTOR);
```


©2023 旋智电子科技（上海）有限公司

```
    /* Flash operation allow */
    FLASHC_WALLOW();

    /* Init Timer1 */
    TIMER_Init(TIMER1, TimerPeridCount);

    /* Open Global INT for TIMER1 */
    NVIC_EnableIRQ(TIMER1_IRQn);

    TIMER_Enable(TIMER1);

    while (1)
    {
        printf("Erase Start (Round %d)...\n",u32Count);

        /* When Erase Flash, XIP Will be Closed, In order to make CPU Fetch 
Instruction
           Normally, we Must Put Flash Driver Code into RAM in .icf file*/
        /* Disable XIP, emulate the start of Flash operation */
        FLASHC_DisableXIP();
        u32Count++;
        /* Enable XIP, emulate the end of Flash operation */
        FLASHC_EnableXIP();
        printf("Erase End...\n");
    }

}

/* If the project is IAR Project, use '__ramfunc' direction letting the func 
running
    in ram address region. But '__ramfunc' is not usefull in KEIL project, So 
we must
    use .sct file to put 'main.o' in ram region for FEIL project. */
_RAM_FUNC_ void TIMER1_IRQHandler(void)
{
    /* When Erase Flash, XIP Will be Closed, At that moment, If TIMER1 
Interrupt
      Happend, CPU will go to Fetch TIMER Driver Code,So we Must Put TIMER 
Driver
      Code into RAM in .icf file*/
    /* Clear the INT */
    TIMER_ClearInt(TIMER1);
}
```

- 中断向量表位置

首先需要考虑到，处理中断的第一要务是要让中断向量表处在 CPU 能够访问到的地址段上，若这一步没有做到，其它努力将都是徒劳。对于前文描述的情形，关闭 XIP 之后，CPU 将无法访问 Flash，所以此时需要将中断向量表从 Flash 中移到 RAM 中，以便 CPU 能够正常访问。其代码如下：

示例代码

```
    /* Copy vector table to RAM */
    for(u32VectorCnt = 0; u32VectorCnt < 79; u32VectorCnt++)
    {
        *((volatile uint32_t *)(VectorTableRelocateAddr + u32VectorCnt * 4)) = \
                              *((volatile uint32_t *)(VectorTableLoadAddr + u32VectorCnt * 4));
    }
```

之后，需要告诉 CPU 更新之后的中断向量表位置在哪里，其代码操作如下：

示例代码

```
    /* Redirection the Vector Address */
    SCB->VTOR = VectorTableRelocateAddr;
```

– 其它代码位置

在找到中断向量表入口之后，CPU 将会访问对应中断的服务函数，所以需要继续考虑此时中断服务函数以及其它 CPU 将会访问到的代码的存储位置。这些在 XIP 关闭期间 CPU 将会访问到的代码段都需要经过处理，使其处于 RAM 中，才不会引起 CPU 异常。

对于中断处理函数而言，将其放入到 RAM 的处理方式如下代码所示，且处理了 KEIL 及 IAR 两种 IDE 所携带的不同编译器的情况，在按照如下方式处理之后，中断服务函数将会被放入到 RAM 中。

示例代码

```
/* If the project is IAR Project, use '__ramfunc' direction letting the func running
    in ram address region. But '__ramfunc' is not usefull in KEIL project, So we must
    use .sct file to put 'main.o' in ram region for FEIL project. */
_RAM_FUNC_ void TIMER1_IRQHandler(void)
{
    /* When Erase Flash, XIP Will be Closed, At that moment, If TIMER1 Interrupt
      Happend, CPU will go to Fetch TIMER Driver Code,So we Must Put TIMER Driver
      Code into RAM in .icf file*/
    /* Clear the INT */
    TIMER_ClearInt(TIMER1);
}
```

从中断服务函数中可以看出，里面调用了 TIMER 的驱动库函数，且在 main 函数中，在关闭 XIP 之后，对 u32Count 变量进行了操作，所以我们需要继续处理 TIMER 驱动以及 u32Count 变量的问题。

对于 TIMER 驱动，需要使用 KEIL IDE 的 SCT 文件进行如下设置，从 SCT 文件中可以看出已经将 timer.o 文件的执行域放在了 RAM 的地址段。

©2023 旋智电子科技（上海）有限公司

示例代码

```
LR_IROM1 0x10000000 0x0000F000  {    ; load region size_region
  ER_IROM1 0x10000000 0x0000F000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x1FFFC000 0x00003000  {  ; RW data
   ; Must put all code refered in the timer interrupt function into RAM
   timer.o
   .ANY (+RW +ZI)
  }
}
```

由于临时变量都会放在 RAM 区域，所以对于 u32Count 变量无需做特殊处理，但需要注意的是，若 main 函数中在 XIP 关闭后操作的不是一个临时变量，而是某个外设的驱动，此时就需要考虑在 SCT 文件中对其进行特殊处理，将其执行域放入到 RAM 地址段。

SPINTROL

3.2 调用第三方函数 API

在实际的项目开发过程中，一个项目的不同功能块可能由不同的公司或者同一公司的不同组完成，这样的开发方式需要双方协商好可调用的 API 接口供对方使用；而硬件层面，也需要双方协商好双方代码可以使用的 Flash 地址段以及 RAM 地址段，否则会产生代码覆盖及运行时数据覆盖的问题。

在 SPC1169 平台相关的产品的 SDK 示例代码中提供了一对示例工程演示此种情形，名为《Flash_Call_User_Func》及《Flash_User_Func》，以下将以 SPC1169 的 KEIL 工程示例代码细节讲解需要关注的细节。

- 调用方代码

调用方代码主要演示如何调用第三方的函数 API 接口。

示例代码

```
#include <stdio.h>
#include <string.h>
#include "func.h"

uint32_t     SPINTestFlag, *argv_1;
PHBs         * MotorStatus, * argvPHB;
int          * argv;
int          val = 10;


int main(void)
{
    CLOCK_InitWithRCO(CLOCK_CPU_100MHZ);

    Delay_Init();

    /*
     * Init the UART
     */
    PIN_SetChannel(PIN_GPIO10, PIN_GPIO10_UART0_TXD);
    PIN_SetChannel(PIN_GPIO11, PIN_GPIO11_UART0_RXD);
    UART_Init(UART0, 38400);
    printf("Start test...\n");

    ThirdFuncLib->pTestFunc();

    argv = &val;
    SPINTestFlag = ThirdFuncLib->pUserFunc(1, 2, 3, argv, 4);
    printf("Spintrol test result : %d \n",SPINTestFlag);

    argv_1 = (uint32_t *)&val;
    argvPHB = malloc(sizeof(struct PHB));
    memset(argvPHB, 0x0, sizeof(struct PHB));

    MotorStatus = ThirdFuncLib->pGetMotorStatus(argv_1, argvPHB, argv, &argv_1,
3);
    printf("Motor test result : %d , %d\n",MotorStatus->phase,MotorStatus-
>voltage);

    while (1)
    {
```

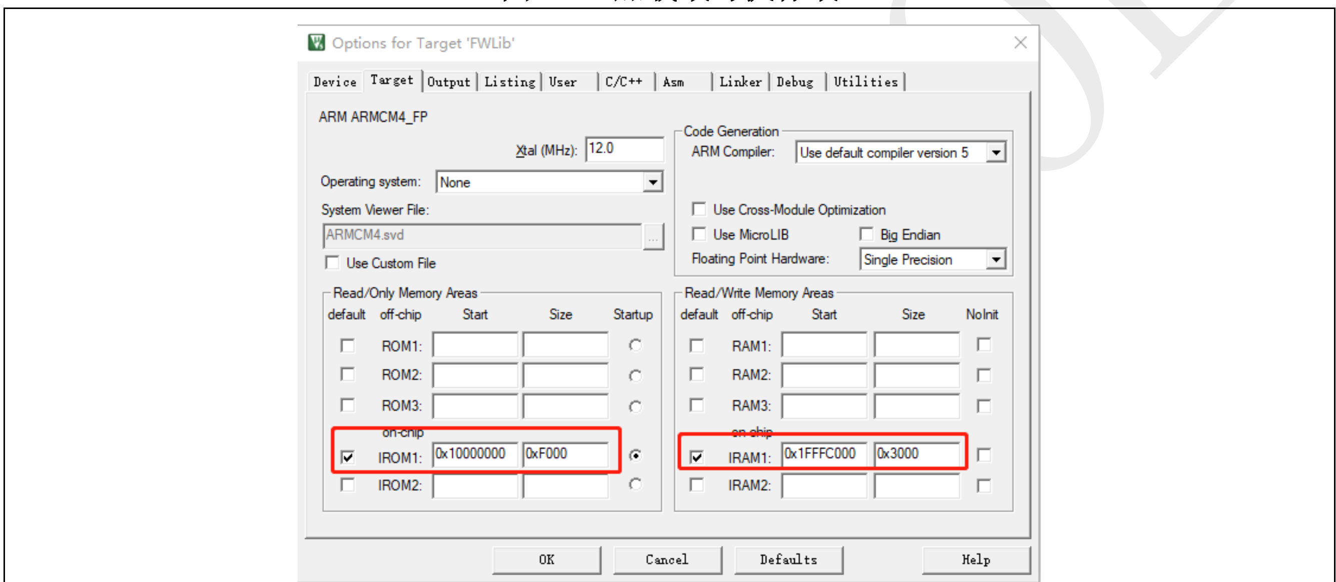

©2023 旋智电子科技（上海）有限公司

```
        }
}
```

在此示例代码中，比较重要的是需要关注 ThirdFuncLib 变量以及工程的加载域、执行域以及堆栈空间的配置。

关于调用方工程加载域及执行域的配置如图 3-1：加载域与执行域所示，从图中可以看出调用者工程的加载域和执行域为 0x10000000~0x1000F000（SPC1169 的 Flash 地址范围为：0x10000000~0x10020000），堆栈空间为 0x1FFFC000~0x1FFFF000（SPC1169 的 RAM 地址范围为：0x1FFF8000~0x20000000）。

图 3-1：加载域与执行域



关于 ThirdFuncLib 变量的细节如下代码所示，其中结构体中断的函数指针成员及 ThirdFuncLib 变量存放的位置由双方协商而定。

示例代码

```
#ifndef FUNC_LIB_H
#define FUNC_LIB_H

#ifdef __cplusplus
extern "C" {
#endif

#include "spc1169.h"

typedef struct PHB
{
  uint32_t voltage;
  uint16_t phase;
}PHBs;


typedef struct FunLibStruct
{
    uint32_t (*pUserFunc)(uint8_t a, uint8_t b, int vtor, int* c, int d);
    PHBs* (*pGetMotorStatus)(uint32_t *PHA, PHBs * PHB, int *PHC, uint32_t
**Key, uint32_t Count);
    void (*pTestFunc)(void);
}FUNCLIB;


const FUNCLIB *ThirdFuncLib = (FUNCLIB *)(0x1000F000);

#ifdef __cplusplus
}
#endif /* extern "C" */

#endif /* FUNC_LIB_H */
```

从以上的代码中可以看出，ThirdFuncLib 指针变量指向地址 0x1000F000，这个地址将在被调用方的代码中进行解释。

- **被调用方代码**

被调用方代码就是提供调用方 API 的代码，其代码如下，在如下的代码中有一个和调用方一模一样的结构体，且这个结构体存放的地址被指定为 0x1000F000，至此，就可以解释，为什么要在调用方代码中将 ThirdFuncLib 指针变量指向地址 0x1000F000。

示例代码

```
#include <stdio.h>
#include "spc1169.h"
#include "func.h"

#if defined ( __CC_ARM )
#define         _PRE_Define
#define         _FUNC_Absolute_Addr_
__attribute__((section(".ARM.__at_0x1000F000")))
```


©2023 旋智电子科技（上海）有限公司

```
#else
#define             _PRE_Define                      __root
#define             _FUNC_Absolute_Addr_             @ (0x1000F000)
#endif

/* Put the function into 0x1000F000 */
_PRE_Define const FUNCLIB ThirdFuncLib _FUNC_Absolute_Addr_ =
{
    UserFunc,
    GetMotorStatus,
    TestFunc,
};

void TestFunc(void)
{
    return;
}

uint32_t UserFunc(uint8_t a, uint8_t b, int vtor, int* c, int d)
{
    d = a + b;
    vtor += d;
    c = &d;
    return *c + vtor;
}

PHBs* GetMotorStatus(uint32_t *PHA, PHBs * PHB, int *PHC, uint32_t **Key,
uint32_t Count)
{
    PHB->voltage = *PHA + *PHC;
    *Key = &Count;
    PHB->phase = **Key;
    return PHB;
}
```

与调用方工程类似，被调用方的工程配置也需要关注加载域、执行域以及堆栈空间的配置。被调用方的这些参数的具体配置使用 KEIL IDE 的 SCT 文件进行配置，具体如下：

示例代码

```
LR_IROM1 0x1000F000 0x00001000  {    ; load region size_region
  ER_IROM1 0x1000F000 0x00001000  {  ; load address = execution address
   .ANY (+RO)
  }
  RW_IRAM1 0x1FFFF000 UNINIT 0x00001000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

从 SCT 配置文件中可以看出，被调用方的加载域与执行域为 0x1000F000~0x10010000，而堆栈空间为 0x1FFFF000~0x20000000。对比调用方的设置，可以看出，双方不存在重叠的地质区域，用户在实际使用时也需要特别注意此类设置，防止地址区域重叠。