



SolarA²

从 ARM-M 向 RISC-V 移植开发指南

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

本文档描述如何将MCU工程从ARM-M芯片移植到RISC-V芯片上进行开发，同时提供移植参考示例，本文中示例以ARM-M0核进行举例，ARM-M其他内核移植方法一致。

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

本文档主要适用于升级的操作人员。

- 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
<div>须知</div>	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。
<div>说明</div>	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 概述.....	1
2 移植准备.....	2
2.1 开发环境.....	2
2.2 软件开发包.....	3
2.2.1 开发包目录结构.....	3
2.2.2 ARM-M 与 RISC-V 移植对比.....	4
3 软件移植.....	5
3.1 移植流程.....	5
3.2 中断与异常.....	7
3.2.1 实现方式.....	7
3.2.1.1 ARM-M 实现方式.....	7
3.2.1.2 STM32 实现 ARM-M 方式中断.....	8
3.2.1.3 RISC-V 实现方式.....	8
3.2.2 对外接口.....	9
3.2.2.1 ARM-M 对外接口.....	9
3.2.2.2 RISC-V 对外接口.....	10
3.3 系统时钟.....	10
3.3.1 实现方式.....	10
3.3.1.1 ARM-M 实现方式.....	10
3.3.1.2 RISC-V 实现方式.....	11
3.3.2 对外接口.....	11
3.3.2.1 ARM-M 对外接口.....	12
3.3.2.2 RISC-V 对外接口.....	12
3.4 外设模块.....	12
4 编译工具链移植.....	13
4.1 RISC-V32 编译工具链实现方式.....	13
4.2 ARMCC 迁移 RISC-V32 编译工具链.....	14
4.2.1 ARMCC 实现方式.....	14
4.2.2 ARMCC 与 RISC-V32 差异对比.....	15
4.3 IAR 迁移 RISC-V32 编译工具链.....	15
4.3.1 IAR 实现方式.....	15



4.3.2 IAR 与 RISC-V32 差异对比.....	16
4.4 TI CL2000 迁移 RISC-V32 编译工具链.....	17
4.4.1 TI CL2000 实现方式.....	17
4.4.2 TI CL2000 与 RISC-V32 差异对比.....	17
5 移植实例参考.....	19
5.1 中断与异常移植实例.....	19
5.1.1 STM32G0 系列实例.....	19
5.1.2 RISC-V 实例.....	20
5.2 系统时钟移植实例.....	22
5.2.1 STM32G0 系列实例.....	22
5.2.2 RISC-V 实例.....	23
6 参考资料.....	26
A 缩略语.....	27



插图目录

图 2-1 系统构架图.....	3
图 3-1 移植流程图.....	6
图 4-1 RISC-V32 编译工具链.....	13
图 4-2 ARMCC 编译工具链.....	14
图 4-3 IAR 编译工具链.....	16
图 4-4 TI CL2000 编译工具链.....	17



表格目录

表 2-1 开发环境信息.....	2
表 2-2 软件开发包目录结构.....	3
表 2-3 架构对比.....	4
表 2-4 移植对比表.....	4
表 3-1 ARM-M 内核中断对外接口.....	9
表 3-2 RISC-V 对外中断接口.....	10
表 3-3 ARM-M 内核定时器对外接口.....	12
表 3-4 RISC-V 内核定时器对外接口.....	12
表 4-1 工具链组件包内容.....	14
表 4-2 编译工具链对比.....	15
表 4-3 编译器对比.....	16
表 4-4 编译工具链对比.....	17
表 A-1 缩略语.....	27



1 概述

随着RISC-V架构的逐渐成熟和应用，越来越多的MCU工程开始考虑从ARM-M芯片移植到RISC-V芯片。ARM-M和RISC-V芯片的架构不同，需要对硬件平台进行适配，包括处理器、外设接口、存储器等。在移植前，需要对硬件平台进行重新设计和调整，以确保软件能够正确运行。在整个移植过程中，软件是一个非常重要的环节，本文将从软件方面介绍MCU工程如何从ARM-M芯片移植到RISC-V芯片。

首先，选择合适的编译器是非常重要的。RISC-V架构的编译器有很多种，例如GCC、Clang等。在选择编译器时，需要考虑到编译器的稳定性、支持的指令集、编译速度等因素。同时，还需要注意编译器的版本，以确保编译出的代码能够在RISC-V芯片上正确运行。

其次，软件开发环境也需要进行适配。ARM-M和RISC-V芯片的软件开发环境不同，包括驱动程序和应用程序等。驱动程序包括芯片的启动代码、中断处理、时钟管理等。应用程序包括各种功能模块、算法等。在移植过程中，需要将ARM-M芯片上的应用程序移植到RISC-V芯片上，并进行相应的修改和适配。需要注意RISC-V芯片的指令集与ARM-M芯片可能存在差异，如果代码中涉及指令集，需要在软件中进行相应的调整。

最后，调试工具也是移植过程中需要考虑的因素之一。ARM-M和RISC-V芯片的调试接口不同，因此需要使用不同的调试工具。在移植过程中，需要对调试工具进行重新配置和调整，以确保软件能够正确调试。

总之，MCU工程从ARM-M芯片移植到RISC-V芯片需要考虑多个方面，包括软件开发平台、编译器、调试工具等。在移植过程中，需要对这些方面进行重新设计和调整，以确保软件能够正确运行。



2 移植准备

本节介绍移植前需要准备的开发环境和软件开发包，利用配套的IDE和SDK可以帮助用户屏蔽芯片架构差异和硬件外设差异，专注于应用程序移植适配，并确保用户代码在RISC-V芯片上可以正确编辑、编译、调试到最终运行。

2.1 开发环境

移植环境部署所需工具：HiSpark-Studio工具，安装与使用方法请参见《HiSparkStudio 使用指南》。

HiSpark-Studio是面向智能设备开发者提供的一站式集成开发环境，支持代码编辑、编译、烧录、调试和变量监控等功能，支持C/C++语言，支持Windows10系统，具有以下特点：

- 支持代码查找、代码高亮、代码自动补齐、代码输入提示、代码检查等，开发者可以轻松、高效编码。
- 支持多种类型开发板。
- 支持单步调试能力和查看内存、变量、调用栈、寄存器、汇编等调试信息。

开发环境信息如[表2-1](#)所示。

表 2-1 开发环境信息

软件工具	说明
HiSpark-Studio	支持代码编辑、编译、烧录、调试、变量监控功能。
Variable Trace	变量监控组件。

说明

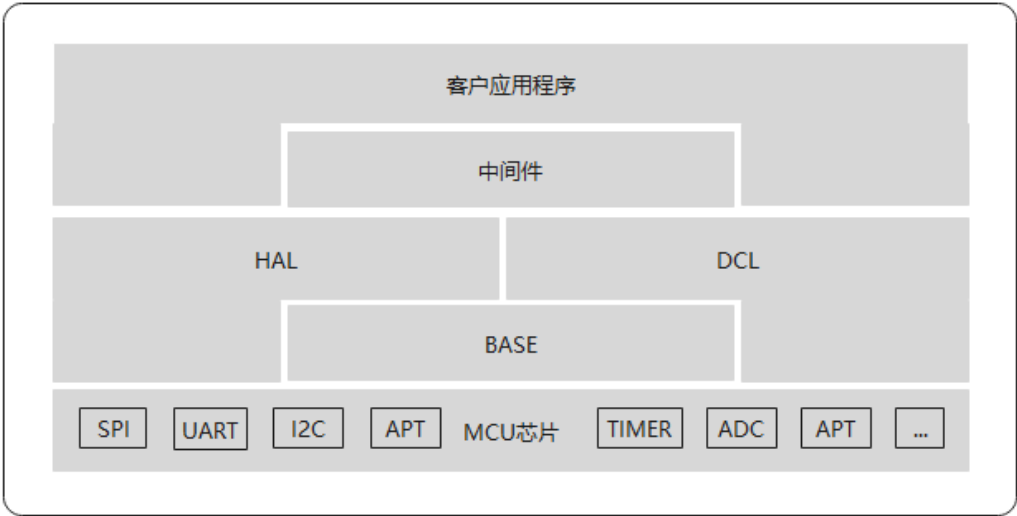
编译工具链编译选项说明，请见《RISCV32 GCC工具链使用指南》中“编译选项”章节。



2.2 软件开发包

SolarA² SDK是一种易于移植的软件开发包，可以帮助开发人员快速地开发出适用于不同平台的嵌入式应用程序，系统框架如图2-1所示。SolarA²采用软件与底层硬件解耦的设计，用户不需要关心ARM-M和RISC-V的架构差异，软件包会为中断和系统时钟提供对外统一的接口。对于硬件外设如UART/SPI/I2C等，抽离出HAL和DCL驱动代码，这种屏蔽外设差异的设计，可以帮助开发人员快速地开发出适用于不同平台的嵌入式应用程序，从而提高开发效率和应用程序的可靠性。

图 2-1 系统构架图



2.2.1 开发包目录结构

软件开发包的目录结构如表2-2所示，提供了从芯片启动代码、驱动程序到应用sample的完整移植参考资源。用户工程从ARM-M移植到RISC-V，软件主要待适配的内容包括：系统启动文件、中断异常、系统时钟、外设驱动等。RISC-V的系统启动文件、中断异常、系统时钟在SolarA²的实现在chip目录下，外设驱动的实现在drivers目录下。

表 2-2 软件开发包目录结构

文件夹名	描述
application	应用sample目录，存放各级sample和客户主程序入口。
board	板级特性目录，存放板级的软件特性。
build	编译构建目录，存放编译相关的脚本和配置信息。
chip	支持芯片目录，存放当前SDK支持的芯片系列的代码。
drivers	驱动目录，存放当前SDK支持的IP驱动。
generatecode	IDE自动生成代码相关目录。



文件夹名	描述
middleware	中间件目录，存放当前SDK版本支持的中间件。
tools	工具目录，存放SDK版本使用的工具。

2.2.2 ARM-M 与 RISC-V 移植对比

通用微控制器软件接口标准（Cortex Microcontroller Software Interface Standard，CMSIS）是Arm Cortex微控制器的抽象层组件，独立于芯片供应商。CMSIS定义了通用接口并确保设备支持一致性，ARM-M的芯片厂商使用CMSIS软件接口简化了处理器和外围设备的重用。在SolarA²软件包中，对RISC-V处理器的使用同样简化出一套对外的接口。以CMSIS中Cortex-M0核为例，[表2-4](#)介绍了处理器对外抽象功能在ARM-M和RISC-V的文件对照列表。

表 2-3 架构对比

对比项	RISC-V	ARM-M0
架构	LinxCore 131（HiMiDeerV200）。	Armv6-M。
指令集	RV32IMCF+自定义指令集。	Thumb or Thumb-2 subset。
流水线	3级流水线。	3级流水线。
总线接口	AHB访问ITCM和DTCM。	AMBA 3 AHB-Lite。
数据排布	小端。	小端（默认）。
系统时钟	systick(64 bits)。	systick(24 bits)。
中断	96个外设中断，16级优先级。	32个外设和终端，4级优先级。
调试接口	JTAG/SWD。	JTAG/SWD。

表 2-4 移植对比表

说明	Arm Cortex M0	RISC-V
中断接口	CMSIS\Include\core_cm0.h	base\inc\interrupt.h chip\interrupt_ip.h
CPU系统控制配置	CMSIS\Include\core_cm0.h	chip\sysctrl.h
SysTick配置	CMSIS\Include\core_cm0.h	chip\systick.h

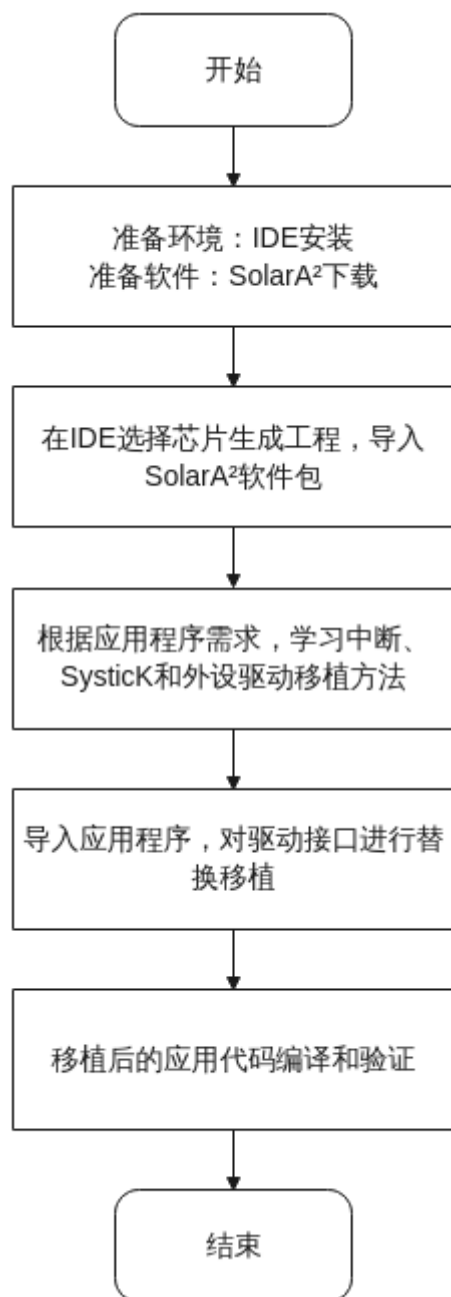


3 软件移植

3.1 移植流程

在完成移植准备工作后，按照[图3-1](#)进行软件移植和移植后的代码验证。

图 3-1 移植流程图



本节将介绍代码从ARM-M移植到RISC-V的操作流程，包括选择芯片生成工程、学习移植CPU相关内容、使用驱动库进行外设移植、导入和适配应用代码、编译和验证移植工程等要点，具体步骤如下。

步骤1 移植准备。

安装HiSpark-Studio IDE保证RISC-V的编译、调试和验证环境正确。下载SolarA²软件包，提供了对外的移植接口，并用于后续建立工程。

步骤2 建立工程。

选择对应的RISC-V芯片，IDE会从SolarA²中导入芯片对应的工程包，生成启动引导文件和内存空间分配文件等配置文件，编译和调试环境完成配置。



步骤3 移植代码。

掌握RISC-V架构的中断和SysTick的使用方法和外设模块的移植。建立的工程已经导入所有的适配文件，用户只需要按照驱动库的使用方法进行移植。

步骤4 导入程序。

导入应用层代码，这个过程需要对应用代码的结构和功能有一定的了解，需要在框架中进行适当的修改和调试，如果代码中涉及到指令集的编码，需要按照RISC-V标准手册进行适配。

步骤5 编译验证。

在完成移植工作后，我们需要对移植工程进行编译和验证。这个过程需要使用RISC-V架构的编译器和调试工具，同时也需要对移植工程的功能进行全面的测试和验证。

---结束

在移植过程中，我们需要对RISC-V架构和ARM-M架构有一定的了解，同时也需要对移植工具和调试工具有一定的掌握。移植工作需要耐心和细心，同时也需要对移植工程的稳定性和可靠性进行全面的测试和验证。

3.2 中断与异常

3.2.1 实现方式

本节将阐述Arm Cortex-M内核和RISC-V的中断及异常实现方式的区别。

3.2.1.1 ARM-M 实现方式

Arm Cortex-M内核中包含了NVIC (Nested vectored interrupt controller)，即嵌套向量中断控制器，通过NVIC来管理中断和异常。其主要包括：中断优先级设置、中断使能、清中断等操作。以下以Cortex-M0内核为例，其他Cortex-M内核实现方式大同小异。Cortex-M0内核中断实现在core_cm0.h文件中。

1. Cortex-M0内核NVIC中断寄存器结构体

```
typedef struct
{
    __IOM uint32_t ISER[1U];          /*!< Offset: 0x000 (R/W) Interrupt Set Enable Register */
    uint32_t RESERVED0[31U];
    __IOM uint32_t ICER[1U];          /*!< Offset: 0x080 (R/W) Interrupt Clear Enable Register */
    uint32_t RESERVED1[31U];
    __IOM uint32_t ISPR[1U];          /*!< Offset: 0x100 (R/W) Interrupt Set Pending Register */
    uint32_t RESERVED2[31U];
    __IOM uint32_t ICPR[1U];          /*!< Offset: 0x180 (R/W) Interrupt Clear Pending Register */
    uint32_t RESERVED3[31U];
    uint32_t RESERVED4[64U];
    __IOM uint32_t IP[8U];             /*!< Offset: 0x300 (R/W) Interrupt Priority Register */
} NVIC_Type;
```

📖 说明

一般配置中断时只使用ISER、ICER、IP三个寄存器，ISER用于使能中断，ICER用于清除中断，IP用于设置中断优先级。



2. Cortex-M0内核NVIC函数接口

ARM-M内核实现中断配置，通过以下接口来进行实现。接口中实现了对NVIC_Type寄存器的访问操作，实现对中断优先级、中断使能等操作。用户根据自己需要调用以下接口来实现中断的管理操作。

```
#define NVIC_SetPriorityGrouping    __NVIC_SetPriorityGrouping
#define NVIC_GetPriorityGrouping    __NVIC_GetPriorityGrouping
#define NVIC_EnableIRQ             __NVIC_EnableIRQ
#define NVIC_GetEnableIRQ          __NVIC_GetEnableIRQ
#define NVIC_DisableIRQ            __NVIC_DisableIRQ
#define NVIC_GetPendingIRQ         __NVIC_GetPendingIRQ
#define NVIC_SetPendingIRQ         __NVIC_SetPendingIRQ
#define NVIC_ClearPendingIRQ       __NVIC_ClearPendingIRQ
/*#define NVIC_GetActive            __NVIC_GetActive        not available for Cortex-M0
*/
#define NVIC_SetPriority            __NVIC_SetPriority
#define NVIC_GetPriority            __NVIC_GetPriority
#define NVIC_SystemReset           __NVIC_SystemReset
```

说明

NVIC_SetPriority、NVIC_EnableIRQ、NVIC_DisableIRQ一般常用于分别设置中断优先级、开启中断、关闭中断（清中断）接口。

3.2.1.2 STM32 实现 ARM-M 方式中断

本节简单阐述STM32G071RB的HAL库实现ARM-M方式中断，以及STM32中断调用流程。

中断配置流程：

步骤1 调用HAL_NVIC_SetPriority设置中断优先级。

HAL_NVIC_SetPriority调用ARM-M内核的NVIC_SetPriority接口来设置中断优先级。

步骤2 调用HAL_NVIC_EnableIRQ开启对应中断。

HAL_NVIC_EnableIRQ调用ARM-M内核的NVIC_EnableIRQ接口开启对应中断。

----结束

中断调用流程：

步骤1 汇编启动文件startup_stm32g071rbtx.s中响应CPU中断。

步骤2 调用对应中断号的xxxx_IRQHandler处理函数。

步骤3 xxxx_IRQHandler函数中调用用户所对应中断处理函数HAL_xxxx_IRQHandler。

----结束

3.2.1.3 RISC-V 实现方式

RISC-V中断实现是通过interrupt.h中的接口实现中断优先级、清除中断、中断使能等操作。接口具体定义在drivers/base/inc/interrupt.h。

中断配置流程：

步骤1 调用IRQ_Register注册中断回调函数到中断向量表。



- 步骤2** 调用IRQ_SetPriority修改中断优先级，如果使用默认中断优先级，则不需要调用该函数。
- 步骤3** 调用IRQ_EnableN使能某个中断。
- 结束
- 中断调用流程：
- 步骤1** 汇编启动文件startup.s中CPU响应中断。
- 步骤2** 通过call InterruptEntry指令进入中断入口函数InterruptEntry。
- 步骤3** InterruptEntry函数中会调用中断号所对应的中断处理函数。
- 步骤4** 响应中断处理函数后，执行IRQ_ClearN函数清除对应中断。
- 结束

3.2.2 对外接口

3.2.2.1 ARM-M 对外接口

Cortex-M0内核中断对外接口如表3-1所示，其他Cortex-M内核实现方式大同小异。Cortex-M0内核中断实现在core_cm0.h文件中。

表 3-1 ARM-M 内核中断对外接口

ARM-M内核中断对外接口	接口描述
NVIC_SetPriorityGrouping	设置中断优先级分组
NVIC_GetPriorityGrouping	获取中断优先级分组
NVIC_EnableIRQ	在NVIC中断控制器中开启特定中断（常用）
NVIC_GetEnableIRQ	获取已使能中断号
NVIC_DisableIRQ	在NVIC中断控制器中关闭特定中断（常用）
NVIC_GetPendingIRQ	获取挂起中断编号
NVIC_SetPendingIRQ	设置中断挂起位
NVIC_ClearPendingIRQ	清除中断挂起位
NVIC_SetPriority	设置特定中断优先级（常用）
NVIC_GetPriority	获取特定中断优先级
NVIC_SystemReset	系统复位

说明

表3-1中所对应的中断类相关接口具体实现请参考core_cm0.h文件。



3.2.2.2 RISC-V 对外接口

RISC-V中断对应接口如表3-2所示，相关RISC-V中断接口具体定义在drivers/base/inc/interrupt.h。

表 3-2 RISC-V 对外中断接口

RISC-V中断接口	接口描述
IRQ_SetPriority	设置中断优先级
IRQ_GetPriority	获取中断优先级
IRQ_Enable	使能全局中断
IRQ_Disable	关闭全局中断
IRQ_EnableN	使能某个中断
IRQ_DisableN	关闭某个中断
IRQ_Init	中断初始化
IRQ_Register	注册中断回调函数
IRQ_Unregister	复位中断回调函数为空
IRQ_ClearAll	清除所有中断使能
InterruptEntry	中断入口函数

说明

中断类相关接口使用方法请参考《SolarA² 驱动程序说明》中的中断类接口相关章节。

3.3 系统时钟

3.3.1 实现方式

3.3.1.1 ARM-M 实现方式

1. 系统时钟结构体SysTick_Type定义如下：

```
typedef struct
{
    __IOM uint32_t CTRL;           /*!< Offset: 0x000 (R/W) SysTick Control and Status Register */
    __IOM uint32_t LOAD;           /*!< Offset: 0x004 (R/W) SysTick Reload Value Register */
    __IOM uint32_t VAL;            /*!< Offset: 0x008 (R/W) SysTick Current Value Register */
    __IOM uint32_t CALIB;          /*!< Offset: 0x00C (R/ ) SysTick Calibration Register */
} SysTick_Type;
```



2. 系统时钟配置流程如下：

步骤1 通过ARM-M内核定时器SysTick_Config(uint32_t ticks)配置接口对系统时钟进行初始化配置。

步骤2 对系统时钟中断号SysTick_IRQn进行中断优先级设置。

----结束

3. 系统时钟使用流程如下（以STM32G0芯片为例）：

步骤1 调用HAL_InitTick()接口对系统时钟进行初始化，包括系统时钟配置初始化和系统时钟中断号初始化。

步骤2 通过在系统时钟中断处理函数SysTick_Handler()中调用HAL_IncTick()函数实现对系统时钟的定时计数。

步骤3 使用HAL_GetTick()函数获取当前时刻系统时钟的时间大小uwTick。

----结束

3.3.1.2 RISC-V 实现方式

1. 系统时钟句柄结构体TIMER_RegStruct定义如下：

```
typedef struct {  
    unsigned int    timer_load;    /**< Initial count value register. Offset address:  
    0x00000000U. */  
    unsigned int    timer_value;    /**< Current count value register. Offset address:  
    0x00000004U. */  
    TIMER_CONTROL_Reg    TIMERx_CONTROL; /**< Timer control register. Offset address:  
    0x00000008U. */  
    unsigned int    timer_intclr;    /**< Interrupt clear register. Offset address: 0x0000000CU.  
    */  
    TIMER_RIS_Reg    TIMERx_RIS;    /**< Raw interrupt register. Offset address:  
    0x00000010U. */  
    TIMER_MIS_Reg    TIMERx_MIS;    /**< Masked interrupt register. Offset address:  
    0x00000014U. */  
    unsigned int    timerbgload;    /**< Initial count value register. Offset address:  
    0x00000018U. */  
    TIMER_CONTROLB_Reg    TIMERx_CONTROLB; /**< Timerx control register B. Offset address:  
    0x0000001CU. */  
} volatile TIMER_RegStruct;
```

2. 系统时钟配置流程

通过调用SYSTICK_Init()接口对系统时钟进行初始化，其中包括时钟基地址、定时器预加载值load值/bgload值、定时器模式、分频大小、中断使能等配置，最后调用HAL_TIMER_Start()接口使能时钟寄存器。

3. 系统时钟使用流程

在完成上述系统时钟初始化配置后，可以直接调用DCL_SYSTICK_GetTick()接口来获取系统时钟的当前时刻值。

3.3.2 对外接口



3.3.2.1 ARM-M 对外接口

表 3-3 ARM-M 内核定时器对外接口

ARM-M内核定时器对外接口	接口描述
<code>__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)</code>	系统定时器配置接口，参数ticks表示两次定时中断之间的时间间隔。
<code>HAL_GetTick()</code>	获取系统时钟的当前时刻值。

3.3.2.2 RISC-V 对外接口

表 3-4 RISC-V 内核定时器对外接口

RISC-V内核定时器对外接口	接口描述
<code>void SYSTICK_Init(void)</code>	系统定时器的初始化配置接口。
<code>unsigned int DCL_SYSTICK_GetTick()</code>	获取系统时钟的当前时刻值。

3.4 外设模块

外设模块的移植，SolarA²提供的HAL接口帮助用户快速移植。

📖 说明

外设模块接口的使用方法和使用用例请参考《SolarA² 驱动程序说明》。

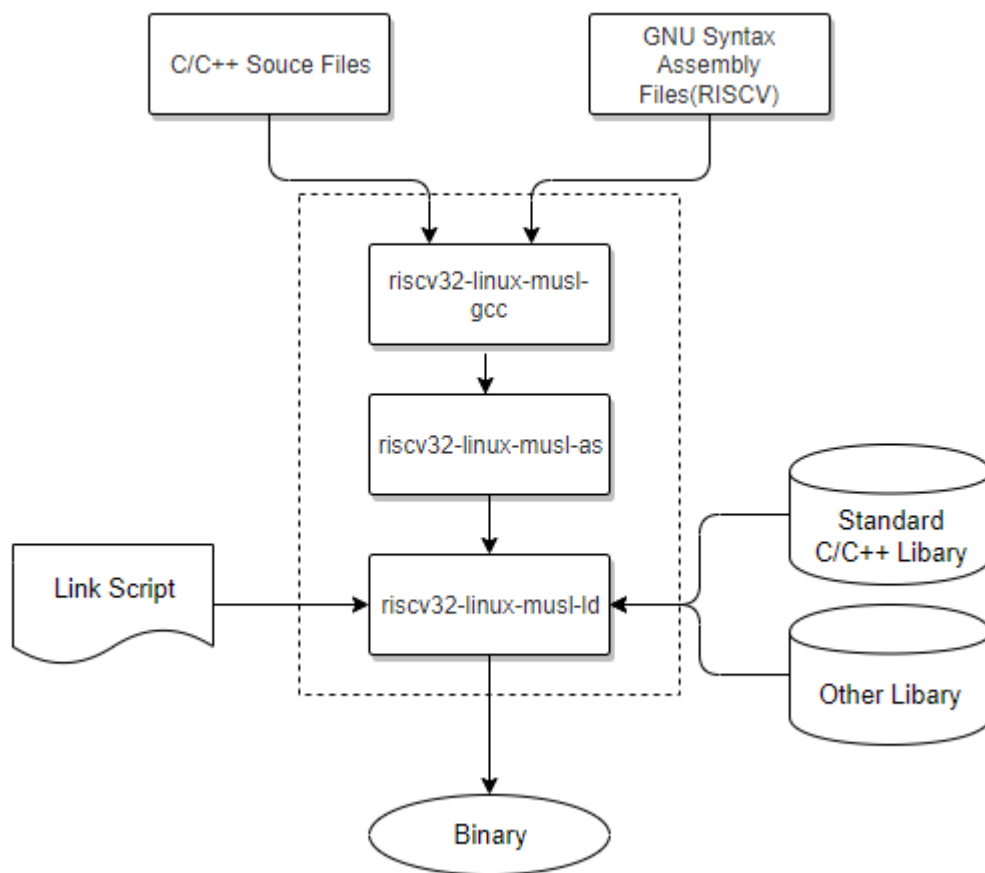


4 编译工具链移植

4.1 RISC-V32 编译工具链实现方式

本节将阐述RISC-V32编译工具链的工作原理和实现方式。RISC-V32编译工具链分为编译器、汇编器和链接器三大部分。其工作原理与实现方式如图4-1所示。

图 4-1 RISC-V32 编译工具链



组件包如表4-1所示。



表 4-1 工具链组件包内容

名称	描述
cc_riscv32_musl.tar.gz	运行在Linux环境，支持自定义指令。
cc_riscv32_musl_fp.tar.gz	运行在Linux环境，支持硬浮点。
cc_riscv32_musl_win.tar.gz	Windows工具链，与Linux工具链源码相同。
cc_riscv32_musl_fp_win.tar.gz	Windows工具链，支持硬浮点，与Linux工具链源码相同。
cc_riscv32_win_env.tar.gz	Windows工具链所依赖的动态库。

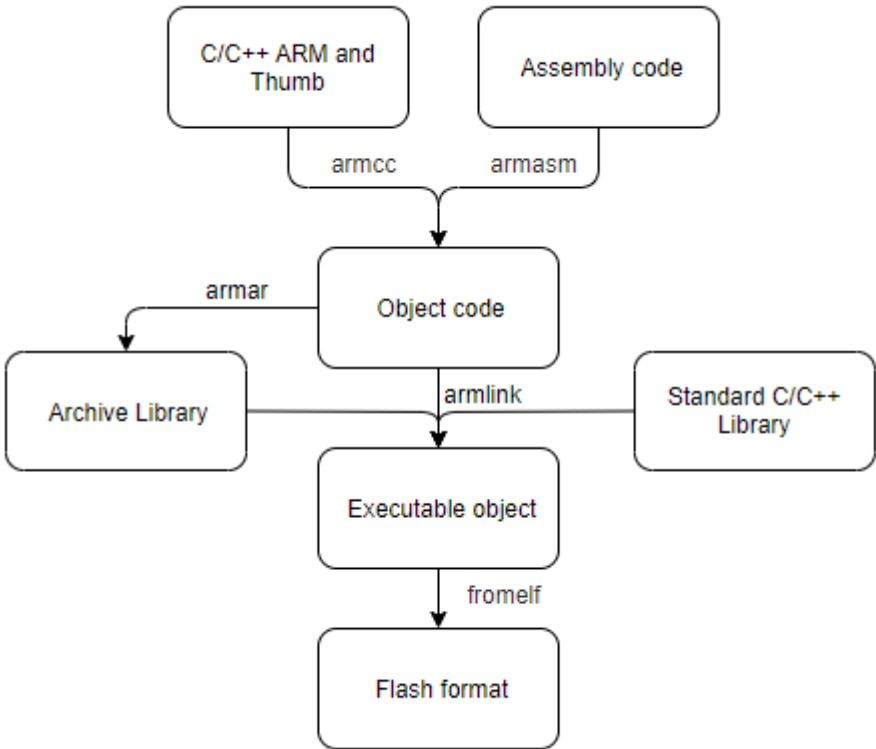
RISCV32编译工具链的详细介绍请参照《RISCV32 GCC工具链使用指南》文档。

4.2 ARMCC 迁移 RISCV32 编译工具链

4.2.1 ARMCC 实现方式

本节将阐述ARMCC编译工具链的工作原理和实现方式。

图 4-2 ARMCC 编译工具链





4.2.2 ARMCC 与 RISC-V32 差异对比

本节将阐述ARMCC编译工具链和RISC-V32编译工具链的区别。主要差异对比如表4-2所示，详细区别请参照《ARMCC迁移RISC-V32编译工具链指南》文档。

表 4-2 编译工具链对比

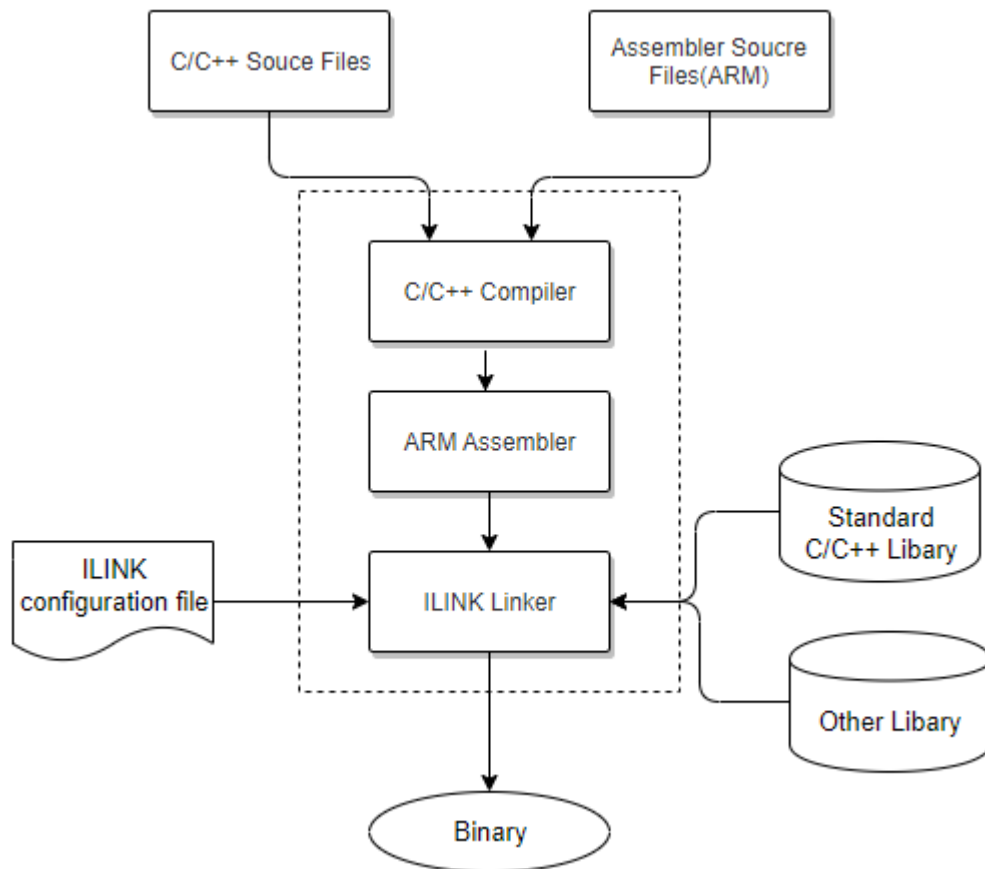
工具	ARMCC	RISC-V32编译工具链
Compiler	KEIL5 ARM C/C++ Compiler(ARMCC) 集成版本 ARM Compiler 5.06 update 7 (build 960) 支持标准C90/C99/C++03/C++11	riscv32-linux-musl-gcc for C riscv32-linux-musl-g++ for C++ 基于开源软件GCC-7.3.0构建 支持标准C89/C90/C++03/C++11/C++14
Assembler	KEIL5 ARM Assembler(armasm) 汇编语言对应ARM指令集要求，汇编器的伪指令满足armasm风格。	riscv32-linux-musl-as 汇编语言对应RISC-V指令集要求，汇编器的伪指令满足GNU风格，具体差异可见下文描述。
Linker	KEIL5 ARM Linker(armlink) 链接脚本是arm自定义的语法。	riscv32-linux-musl-ld 链接脚本是满足GNU Linker script定义的语法要求。
Archiver	KEIL5 ARM Archiver(armar)	riscv32-linux-musl-ar
Image Conversion utility	KEIL5 ARM Image Converter(fromelf)	riscv32-linux-musl-objcopy

4.3 IAR 迁移 RISC-V32 编译工具链

4.3.1 IAR 实现方式

本节将阐述IAR编译工具链的工作原理和实现方式。

图 4-3 IAR 编译工具链



4.3.2 IAR 与 RISC-V 差异对比

本节将阐述 IAR 编译工具链和 RISC-V 编译工具链的区别。主要差异对比如表 4-3 所示，详细区别请参照《IAR 迁移 RISC-V 编译工具链指南》文档。

表 4-3 编译器对比

工具	IAR	RISC-V 编译工具链
Compiler	IAR C/C++ Compiler(iccarm) 集成版本 IAR Embedded wokbench IDE V9.30.1.x 支持标准 C89/C18	riscv32-linux-musl-gcc 基于开源软件 GCC-7.3.0 构建 支持标准 C89/C11
Assembler	IAR ARM Assembler(iasmarm) 汇编语言对应 ARM 指令集要求，汇编器的伪指令满足 IAR 风格。	riscv32-linux-musl-as 汇编语言对应 RISC-V 指令集要求，汇编器的伪指令满足 GNU 风格，具体差异可见下文描述。
Linker	IAR ILINK Linker(ilinkarm) 链接脚本是 IAR 自定义的语法。	riscv32-linux-musl-ld 链接脚本是满足 GNU Linker script 定义的语法要求。

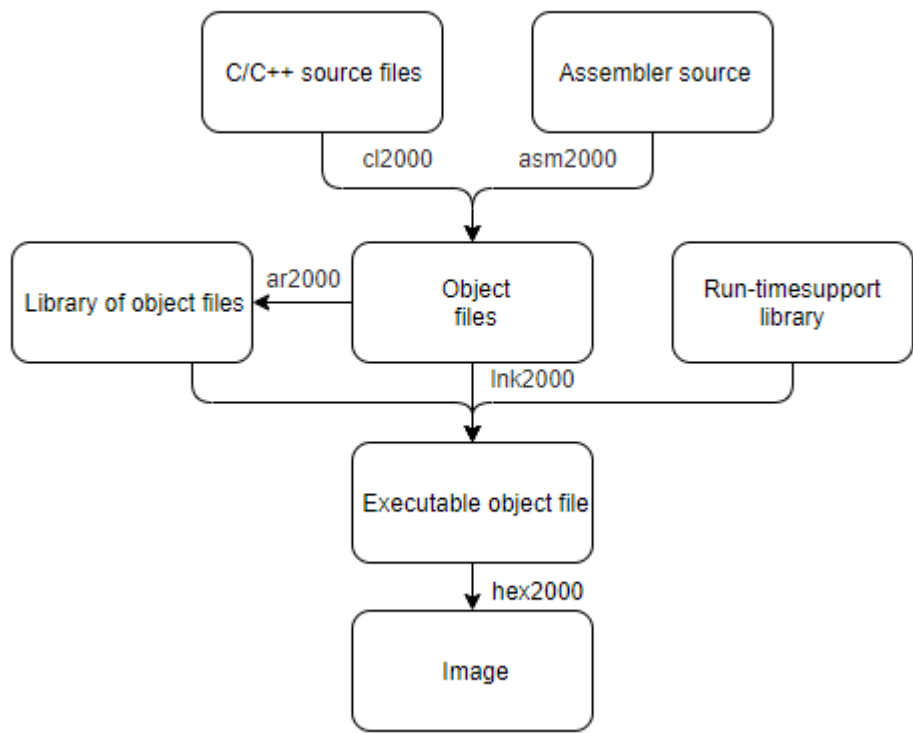


4.4 TI CL2000 迁移 RISC-V32 编译工具链

4.4.1 TI CL2000 实现方式

本节将阐述TI CL2000编译工具链的工作原理和实现方式。

图 4-4 TI CL2000 编译工具链



4.4.2 TI CL2000 与 RISC-V32 差异对比

本节将阐述TI cl2000编译工具链和RISC-V32编译工具链的区别。主要差异对比如表4-4所示，详细区别请参照《TI cl2000迁移RISC-V32编译工具链指南》文档。

表 4-4 编译工具链对比

工具	TI CL2000	RISC-V32编译工具链
Compiler	cl2000 集成版本 TMS320C2000 C/C++ Parser v22.6.0.LTS 支持标准C89/C99/C11/C++03	riscv32-linux-musl-gcc 基于开源软件GCC-7.3.0构建 支持标准C89/C11/C++03/C++11/C++14



工具	TI CL2000	RISCV32编译工具链
Assembler	cl2000(asm2000) 汇编语言对应C28x汇编语言指令要求。	riscv32-linux-musl-as 汇编语言对应RISCV指令集要求，汇编器的伪指令满足GNU风格，具体差异可见下文描述。
Linker	cl2000(lnk2000) 满足链接命令文件语法要求。	riscv32-linux-musl-ld 链接脚本是满足GNU Linker script定义的语法要求。
Archiver	ar2000 将多个单独的文件合并为一个存档文件。	riscv32-linux-musl-ar 用于创建、修改和提取静态库文件。
C++ Demangler	dem2000 C++ 名称还原器是一种调试辅助工具，其将检测到的每个已改编的名称转换为其在 C++ 源代码中找到的原始名称。	riscv32-linux-musl-c++filt 将C++符号进行解码，将其转换为易于阅读的形式。
Disassembler	dis2000 接受目标文件或可执行文件作为输入，并将反汇编的目标代码写入标准输出或指定文件。	riscv32-linux-musl-objdump 查看目标程序中的段信息和调试信息，也可以用来对目标程序进行反汇编。
Hex Converter	hex2000 可以将可执行目标文件转换为适合输入到EPROM 编程器的格式。	riscv32-linux-musl-objcopy 可以对最后生成的程序文件进行一定的编辑、转换。
Name Utility	nm2000 输出目标文件定义和引用的符号。	riscv32-linux-musl-nm 列出目标文件中的符号。
Strip Utility	strip2000 从目标文件中删除符号表和调试信息。	riscv32-linux-musl-strip 去除目标文件中的一些符号表等信息。



5 移植实例参考

5.1 中断与异常移植实例

为了方便用户快速从ARM-M内核的NVIC中断移植到RISC-V中断，下面以看门狗为例进行中断介绍。对于ARM-M内核中断，将以STM32G0系列的WWDG为例介绍如何迁移到RISC-V。

5.1.1 STM32G0 系列实例

中断配置

```
//此函数用户自己实现，HAL_WWDG_Init会自动调用此接口
void HAL_WWDG_MspInit(WWDG_HandleTypeDef* hwwdg)
{
    __HAL_RCC_WWDG_CLK_ENABLE();//开启WWDG时钟
    HAL_NVIC_SetPriority(WWDG_IRQn, 0, 0); //设置WWDG中断优先级
    HAL_NVIC_EnableIRQ(WWDG_IRQn);//开启WWDG中断
}
//HAL_NVIC_SetPriority实现
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority)
{
    .....
    NVIC_SetPriority(IRQn, PreemptPriority); //调用ARM内核接口来设置中断优先级
}
//HAL_NVIC_EnableIRQ实现
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn)
{
    .....
    NVIC_EnableIRQ(IRQn);//调用ARM内核接口来设置中断使能
}
```

中断回调函数和中断处理函数相关配置

```
//用户自定义实现WWDG回调函数
void WWDG_WakeupCallback(WWDG_HandleTypeDef* hwwdg)
{
    HAL_WWDG_Refresh(&WWDG_Handler,0x7F);//更新窗口看门狗值
}
//WWDG中断服务函数，用户自己实现
void WWDG_IRQHandler(void)
{
}
```



```
    HAL_WWDG_IRQHandler(&WWDG_Handler); //调用WWDG中断处理函数
}
//WWDG用户初始化操作，里面进行WWDG注册中断回调函数
WWDG_HandleTypeDef WWDG_Handler;
void WWDG_Init(uint8_t tr,uint8_t wr,uint32_t fprer)
{
    .....
    HAL_WWDG_Init(&WWDG_Handler);    //初始化WWDG

    HAL_WWDG_RegisterCallback(&WWDG_Handler,HAL_WWDG_EWI_CB_ID,WWDG_WakeupCallback); //注册WWDG中断回调函数
    HAL_WWDG_Start_IT(&WWDG_Handler); //开启窗口看门狗，并开启WWDG中断
}
```

中断调用流程

步骤1 WWDG产生中断，汇编启动文件startup_stm32g071rbtx.s中响应CPU中断，执行WWDG_IRQHandler。

```
.....
.word SysTick_Handler
.word WWDG_IRQHandler          /* Window WatchDog          */
.word PVD_IRQHandler           /* PVD through EXTI Line detect */
.word RTC_TAMP_IRQHandler      /* RTC through the EXTI line    */
.word FLASH_IRQHandler         /* FLASH                        */
.....
```

步骤2 调用HAL_WWDG_IRQHandler函数。

步骤3 HAL_WWDG_IRQHandler调用用户自定义的回调函数WWDG_WakeupCallback。

----结束

5.1.2 RISC-V 实例

中断配置

```
IRQ_Register(IRQ_WWDG, HAL_WWDG_IrqHandler, &g_wwdg); //中断注册
IRQ_SetPriority(IRQ_WWDG, 1); //设置中断优先级
IRQ_EnableN(IRQ_WWDG); //中断使能
//中断注册实现
unsigned int IRQ_Register(unsigned int irqNum, IRQ_PROC_FUNC func, void *arg)
{
    INTERRUPT_ASSERT_PARAM(func != NULL);
    INTERRUPT_PARAM_CHECK_WITH_RET(irqNum < IRQ_MAX, IRQ_ERRNO_NUM_INVALID);
    if (g_irqCallbackFunc[irqNum].pfnHandler != IRQ_DummyHandler) {
        return IRQ_ERRNO_ALREADY_CREATED;
    }
    IRQ_SetCallBack(irqNum, func, arg);
    return BASE_STATUS_OK;
}
//中断使能实现
unsigned int IRQ_EnableN(unsigned int irqNum)
{
    unsigned int irqOrder;
    unsigned int locienVal;
    #if defined(USER_MODE_ENABLE) && (USER_MODE_ENABLE == 1)
        unsigned int priv = IRQ_GetCpuPrivilege();
    #endif
    INTERRUPT_PARAM_CHECK_WITH_RET((irqNum >= IRQ_VECTOR_CNT && irqNum <
    IRQ_MAX), IRQ_ERRNO_NUM_INVALID);
```



```
/* The interrupt enable bits that can be controlled in the mie register (32 bits), up to 32
   can be controlled, and each bit corresponds to an interrupt enable */
RISCV_PRIV_MODE_SWITCH(priv);
if (irqNum < IRQ_MIE_TOTAL_CNT) {
    irqOrder = 1U << irqNum;
    SET_CSR(mie, irqOrder);
} else if (irqNum < IRQ_LOCIEN1_OFFSET) {
    irqOrder = irqNum - IRQ_MIE_TOTAL_CNT;
    locienVal = READ_CUSTOM_CSR(LOCIE0);
    locienVal |= (1U << irqOrder);
    WRITE_CUSTOM_CSR_VAL(LOCIE0, locienVal);
} else if (irqNum < IRQ_LOCIEN2_OFFSET) {
    irqOrder = irqNum - IRQ_LOCIEN1_OFFSET;
    locienVal = READ_CUSTOM_CSR(LOCIE1);
    locienVal |= (1U << irqOrder);
    WRITE_CUSTOM_CSR_VAL(LOCIE1, locienVal);
} else {
    irqOrder = irqNum - IRQ_LOCIEN2_OFFSET;
    locienVal = READ_CUSTOM_CSR(LOCIE2);
    locienVal |= (1U << irqOrder);
    WRITE_CUSTOM_CSR_VAL(LOCIE2, locienVal);
}
RISCV_PRIV_MODE_SWITCH(priv);
return BASE_STATUS_OK;
}

//中断优先级实现
unsigned int IRQ_SetPriority(unsigned int irqNum, unsigned int priority)
{
    INTERRUPT_PARAM_CHECK_WITH_RET((irqNum >= IRQ_VECTOR_CNT && irqNum <
    IRQ_MAX), IRQ_ERRNO_NUM_INVALID);
    INTERRUPT_PARAM_CHECK_WITH_RET((priority >= IRQ_PRIO_LOWEST && priority <=
    IRQ_PRIO_HIGHEST), \
    IRQ_ERRNO_PRIORITY_INVALID);
    /* The locipri register is specifically used to configure the priority of the
       external non-standard interrupts of the CPU, so the number of internal
       standard interrupts should be subtracted */
    IRQ_SetLocalPriority(irqNum - IRQ_VECTOR_CNT, priority);
    return BASE_STATUS_OK;
}
```

中断回调函数和中断处理函数相关配置

```
WWDG_Handle g_wwdg;
//用户自定义的中断回调函数
void WWDGCallbackFunction(void *handle)
{
    WWDG_Handle *wwdgHandle = (WWDG_Handle *)handle;
    HAL_WWDG_Refresh(wwdgHandle);
}
//用户进行初始化相关操作，HAL_WWDG_RegisterCallback接口注册用户中断回调函数
void WWDG_Init(void)
{
    .....
    HAL_WWDG_Init(&g_wwdg);
    HAL_WWDG_RegisterCallback(&g_wwdg, WWDGCallbackFunction); //注册回调函数
    .....
}
```

中断回调流程

步骤1 汇编启动文件startup.s中CPU响应WWDG中断。



步骤2 通过call InterruptEntry指令进入中断入口函数InterruptEntry。

步骤3 InterruptEntry函数中会调用WWDG中断号所对应的中断处理函数。

```
void InterruptEntry(unsigned int irqNum)
{
    g_irqCallbackFunc[irqNum].pfnHandler(g_irqCallbackFunc[irqNum].param);
    IRQ_ClearN(irqNum); //清除WWDG中断
}
```

步骤4 由于IRQ_Register接口把HAL_WWDG_IrqHandler中断处理函数和中断号IRQ_WWDG进行注册，所以最终执行到HAL_WWDG_IrqHandler函数。

步骤5 HAL_WWDG_IrqHandler中会执行用户WWDGCallbackFunction回调函数。

----结束

5.2 系统时钟移植实例

5.2.1 STM32G0 系列实例

系统时钟配置

```
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /*Configure the SysTick to have interrupt in 1ms time basis*/
    if (HAL_SYSTICK_Config(SystemCoreClock / (1000U / uwTickFreq)) > 0U) // 以中断时间间隔
    1ms进行系统时钟的初始化配置
    {
        return HAL_ERROR;
    }

    /* Configure the SysTick IRQ priority */
    if (TickPriority < (1UL << __NVIC_PRIO_BITS))
    {
        HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U); // 设置系统时钟中断号优先级
        uwTickPrio = TickPriority;
    }
    else
    {
        return HAL_ERROR;
    }

    /* Return function status */
    return HAL_OK;
}
```

系统时钟延时实例

以系统时钟进行延时为例进行描述：

```
__weak void HAL_Delay(uint32_t Delay)
{
    uint32_t tickstart = HAL_GetTick(); // 获取系统时钟当前时刻大小
    uint32_t wait = Delay;

    /* Add a freq to guarantee minimum wait */
    if (wait < HAL_MAX_DELAY)
    {

```



```
    wait += (uint32_t)(uwTickFreq);
}

while((HAL_GetTick() - tickstart) < wait) // while空循环实现延时
{
}
}
```

其中获取系统时钟当前时刻函数HAL_GetTick()具体实现如下：

```
__weak uint32_t HAL_GetTick(void)
{
    return uwTick; // 直接返回全局变量时刻值uwTick
}
```

而系统时钟是通过每次在系统时钟中断处理函数SysTick_Handler中累加全局时刻值uwTick来实现记录当前时刻大小的，具体实现如下：

```
void SysTick_Handler(void)
{
    /* USER CODE BEGIN SysTick_IRQn 0 */

    /* USER CODE END SysTick_IRQn 0 */
    HAL_IncTick();
    /* USER CODE BEGIN SysTick_IRQn 1 */

    /* USER CODE END SysTick_IRQn 1 */
}

__weak void HAL_IncTick(void)
{
    uwTick += uwTickFreq;
}
```

5.2.2 RISC-V 实例

系统时钟配置

在RISC-V内核的系统时钟实现中，直接使用预先定义的定时器TIMER类型的SYSTICK变量进行系统时钟基地址初始化，其他属性也和定时器结构体成员初始化一致，具体实现如下：

```
void SYSTICK_Init()
{
    /* Choose the config to support GetTick and Delay */
    g_systickHandle.baseAddress = SYSTICK;
#ifdef NOS_TASK_SUPPORT
    /* Change the period load to the user defined usecond */
    g_systickHandle.load = (HAL_CRG_GetIpFreq(SYSTICK_BASE) / CRG_FREQ_1MHz) *
CFG_SYSTICK_TICKINTERVAL_US;
    g_systickHandle.bgLoad = (HAL_CRG_GetIpFreq(SYSTICK_BASE) / CRG_FREQ_1MHz) *
CFG_SYSTICK_TICKINTERVAL_US;
#else
    g_systickHandle.load = SYSTICK_MAX_VALUE;
    g_systickHandle.bgLoad = SYSTICK_MAX_VALUE;
#endif
    g_systickHandle.mode = TIMER_MODE_RUN_PERIODIC;
    g_systickHandle.prescaler = TIMERPRESCALER_NO_DIV;
    g_systickHandle.size = TIMER_SIZE_32BIT;
    /* Don't Support IRQ because only needs to read the value of systick */
}
```



```
g_systickHandle.interruptEn = BASE_CFG_DISABLE;
HAL_TIMER_Init(&g_systickHandle);
#ifdef NOS_TASK_SUPPORT
/* Support IRQ to upload the totalCycle and detect the timeout lists */
SYSTICK_IRQ_Enable();
#endif
HAL_TIMER_Start(&g_systickHandle);
}
```

系统时钟延时实例

下面给出了使用三种不同时间单位进行延时的函数实现：

```
/**
 * @brief Delay number of us.
 * @param us The number of us to delay.
 * @retval None.
 */
void BASE_FUNC_DelayUs(unsigned int us) // 以微秒为单位进行延时us微秒
{
    unsigned int preTick = DCL_SYSTICK_GetTick();
    unsigned int tickInUs = (SYSTICK_GetCRGHZ() / CRG_FREQ_1MHz) * us; // 延时tickUs微秒
    unsigned int curTick;
    unsigned int delta;

    /* Wait until the delta is greater than tickInUs */
    do {
        curTick = DCL_SYSTICK_GetTick();
        delta = (curTick >= preTick) ? curTick - preTick : SYSTICK_MAX_VALUE - preTick + curTick
+ 1;
    } while (delta < tickInUs);
}

/**
 * @brief Delay number of ms.
 * @param ms The number of ms to delay.
 * @retval None.
 */
void BASE_FUNC_DelayMs(unsigned int ms) // 以毫秒为单位进行延时ms毫秒
{
    for (unsigned int i = 0; i < ms; ++i) {
        BASE_FUNC_DelayUs(BASE_DEFINE_DELAY_US_IN_MS);
    }
}

/**
 * @brief Delay number of seconds.
 * @param seconds The number of seconds to delay.
 * @retval None.
 */
void BASE_FUNC_DelaySeconds(unsigned int seconds) // 以秒为单位进行延时seconds秒
{
    for (unsigned int i = 0; i < seconds; ++i) {
        BASE_FUNC_DelayMs(BASE_DEFINE_DELAY_MS_IN_SEC);
    }
}
```

其中，获取系统时钟当前时刻值DCL_SYSTICK_GetTick()函数实现如下：

```
unsigned int DCL_SYSTICK_GetTick(void)
{
```




```
#ifndef NOS_TASK_SUPPORT
    /* Return the load value(period) and the counter value, make the returned counter in count
    up mode */
    return DCL_GetCpuCycle();
#else
    /* Invert the counter value, make the returned counter in count up mode */
    return ~SYSTICK->timer_value;    // 通过直接读取SYSTICK寄存器的timer_value值来作为当前时
    刻值大小
#endif
}
```



6 参考资料

- 《HiSparkStudio 使用指南》
- 《SolarA² 驱动程序说明》
- 《RISC-V32 GCC工具链使用指南》
- 《ARMCC迁移RISC-V32编译工具链指南》
- 《IAR迁移RISC-V32编译工具链指南》
- 《TI cl2000迁移RISC-V32编译工具链指南》
- RISC-V指令集非特权标准: <https://github.com/riscv/riscv-isa-manual/releases/download/archive/riscv-spec-v2.2.pdf>
- RISC-V指令集特权标准: <https://github.com/riscv/riscv-isa-manual/releases/download/archive/riscv-privileged-v1.10.pdf>
- RISC-V调试标准: https://github.com/riscv/riscv-debug-spec/releases/download/task_group_vote/riscv-debug-draft.pdf



A 缩略语

表 A-1 缩略语

缩略语	英文	中文
CMSIS	Cortex Microcontroller Software Interface Standard	微控制器软件接口标准
DCL	Direct Configuration Layer	直接配置层
GCC	GNU Compiler Collection	GNU编译器集
HAL	Hardware Abstraction Layer	硬件抽象层
IDE	Integrated Development Environment	集成开发环境
MCU	Microcontroller Unit	微控制器单元
NVIC	Nested vectored interrupt controller	嵌套向量中断控制器
WWDG	Windowed Watch Dog	窗口看门狗