应用指南

CM32M4xxR 系列芯片
中断应用指南

V1.0

目录

一、概述.....- 1 -
二、中断向量表.....- 1 -
三、ECLIC 初始化.....- 4 -
四、中断服务函数.....- 5 -
五、中断配置.....- 9 -
六、历史版本..... - 10 -

一、概述

CM32M4xxR 系列芯片采用芯来 N308 处理器（RISC-V 内核），在中断的管理及处理上与基于 ARM Cortex-M4 处理器存在一定差异。因此，对于前期使用 Cortex-M4 中断系统的开发者而言，刚切换到 N308 可能会对中断系统的配置比较陌生。本文档旨在帮助开发者快速熟悉 CM32M4xxR 中断系统的特点及使用方式，并会对 N308 和 Cortex-M4 的中断系统差异进行简要说明。

需要指出的是以下内容均基于 CM32M4xxR 的 SDK，并将直接引用 SDK 中的文件名、函数名等信息。另外，虽然 Cortex-M4 是处理器型号而非 MCU 型号，但由于 MCU 的中断系统主要取决于其处理器的中断特性，因此为避免叙述累赘，以下可能会直接以 Cortex-M4 代指基于 Cortex-M4 处理器的 MCU，请注意区分。

二、中断向量表

CM32M4xxR 的向量表（.vtable/vtable_ilm）定义在启动文件 startup_cm32m4xxr.S 中，共 105 个中断向量，包括 19 个内部中断向量和 86 个外部中断向量；与中断向量表对应的中断编号（IRQn）在 CM32M4xxR.h 文件中定义，由枚举实现，从数值从 0 开始，且内部中断和外部中断连续编号。这里内部中断是指供处理器内核使用的中断，而外部中断指供处理器外的片上外设使用的中断。

在 19 个内部中断中，CM32M4xxR 只使用了软件中断和 TIMER 中断两个中断，分别对应中断编号 3 和 7；在 86 个外部中断中，除了对应中断号 61、68、80、81 和 97 的 5 个中断保留外，其他均为有效中断，可以根据相应的外设来配置使用。

中断向量表在存储器中的位置由链接控制脚本 gcc_cm32m4xxr_xx.ld 确定，一般位于逻辑地址 0x00000000 处，也即代码区域的起始位置。在内核启动初期，需将中断向量表的地址传递至 mtvt 寄存器中，以下是在启动文件中对中断向量表的地址配置；

```
/*
 * Intialize ECLIC vector interrupt
 * base address mtvt to vector_base
 */
la t0, vector_base
csrw CSR_MTVT, t0
```

处理器在响应中断时会根据 mtvt 寄存器中存储的向量表基地址来查找中断向量，当前在程序运行过程中可以通过修改 mtvt 来重定位向量表，需要注意的是**在重定位向量表时要满足 mtvt 的 512 字节对齐要求（最大中断数为 105）**。附 MTVT 对齐要求列表：

最大中断个数	MTVT 对齐方式
0-16	64 bytes
17-32	128 bytes
33-64	256 bytes
65-128	512 bytes
129-256	1024 bytes
257-512	2048 bytes

上面介绍的向量表和中断编号的定义、向量表存储位置及重定位特性与 Cortex-M4 类似，可以类比 Cortex-M4 及 CMSIS 代码来理解。以下对 N308 和 Cortex-M4 关于中断向量表差异比较大的一些点进行说明，分别是复位流程、异常处理和非向量中断。

（1）复位流程

在复位后及处理器开始执行程序前，Cortex-M4 会从向量表读出头两个字，分别是主栈指针的初始值和复位向量，并分别赋给 MSP 和 PC，接着执行复位处理函数并逐步引导至用户程序。

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
```

Cortex-M4 内核中断向量表定义

同样 CM32M4xxR 在复位后会将直接从初始位置 0 地址运行，但向量表第一个字为执行指令，一般情况下此处会放置一条跳转指令，并跳转至真正的启动程序，然后由启动程序逐步引导至用户程序。

```
.globl vector_base
vector_base:
```

```
    j _start     /* Jump to _start when reset for ILM/FlashXIP mode.*/
    .align LOG_REGBYTES     /* Need to align 4 byte for RV32, 8 Byte for RV64 */
    DECLARE_INT_HANDLER       default_intexc_handler   /* 1: Reserved */
    DECLARE_INT_HANDLER       default_intexc_handler    /* 2: Reserved */
    DECLARE_INT_HANDLER       eclic_msip_handler     /* 3: Machine software interrupt */
```

CM32M4xxR 中断向量表定义

这里二者的区别主要有两点：一是向量表头一（两）个字的内容的含义不同，这点显而易见；二是复位后处理器对向量表头一（两）个字的处理方式不同，Cortex-M4 从向量表第二个字中取出的是复位处理函数的地址，赋给 PC 然后取该地址的指令执行，而 CM32M4xxR 从向量表第一个字取出的就是一条指令，即在此之前 PC 的值已经被设为 0。

CM32M4xxR 的启动由_start 函数负责，该函数定义在 startup_cm32m4xxr.S 文件中，且为了使处理器在复位后跳转至_start 执行，在向量表的第一个字中放置了 j _start 指令。

因此通常我们在 Cortex-M4 内核上进行应用层的 Bootloader 开发时，通常将采用如下的应用层跳转实现：

```
  /* Jump to user application */
  JumpAddress = *(__IO uint32_t*) (APPLICATION_ADDRESS + 4);//
  JumpToApplication = (pFunction) JumpAddress;//
  /* Initialize user application's Stack Pointer */
  __set_MSP(*(__IO uint32_t*) APPLICATION_ADDRESS); //
  JumpToApplication(); //
```

Cortex-M4 App 跳转示例

而在 CM32M4xxR 下，可直接跳转至应用代码的初始地址，并将随后通过 j 指令跳转至应用层的_start 方法，Bootloader 的跳转示例如下：

```
void JumpApp(void)
{
    pFunction JumpToApplication = (pFunction) APPLICATION_ADDRESS;
    JumpToApplication();
}
```

CM32M4xxR 应用跳转示例

（2）异常处理

Cortex-M4 将异常、NMI 和中断统一进行管理（按 ARM 的说法，中断也是一种异常）：三者编号是连续的，向量表中包含三者的向量，且除了优先级固定的几个

异常（复位、NMI 和 HardFault）外，其他异常和中断的优先级都可以在统一的可编程范围（0~255）内进行配置。

在 N308 中异常、NMI 和中断两者分开并独立编号，向量表中只包含中断向量，而不包含异常和 NMI 向量，且 N308 的异常和 NMI 的优先级是不可配的，其优先级始终高于中断。在发生异常、NMI 时，分别由 mtvec 和 mnvec 寄存器指定入口地址（NMI 也可以与异常共用一个处理入口），且与中断处理入口地址相互独立。

CM32M4xxR 的启动文件中设置异常和 NMI 共享处理入口 exc_entry，其实现位于在 intexc_cm32m4xxr.S 文件中，并提供了默认的异常处理实现。用户可通过 `Exception_Register_EXC(uint32_t EXCn, unsigned long exc_handler)` 接口为不同的异常注册响应的响应函数。当然用户也可重载 exc_entry 并自定义异常的响应入口；但注意 exc_entry 入口地址需保证 64 字节对齐；

（3）非向量中断

在中断方面 N308 区别于 Cortex-M4 的另外一点就是前者支持向量中断和非向量中断两种中断处理模式，而后者只支持向量处理模式。CM32M4xxR 每个中断源均可以设置成向量或者非向量中断方式。其特点在于：如果被配置成为向量处理模式，则该中断被处理器内核响应后，处理器直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址。而在非向量处理模式中，所有中断在被响应时都会先进入一个公共的处理函数中（由 mtvt2 寄存器指定入口地址），且在所有挂起中断处理完成后从该函数中退出。公共处理函数在这里的作用主要是负责保存和恢复上下文，并实现中断的咬尾。需要补充的是 **N308 向量中断模式不支持中断咬尾，另外无论是向量中断还是非向量中断，最终均会跳转至前述的中断向量表中相应的中断向量处执行中断服务程序。**

CM32M4xxR 的非向量处理的公共入口是 irq_entry，定义在 intexc_cm32m4xxr.S 文件中。另外，由于向量处理模式下对中断服务函数编写有一些特殊的要求（详见下文“四、中断服务函数”），因此建议在刚开始使用 CM32M4xxR 的中断时优先使用非向量中断（复位后默认的模式），待后续对这两种中断处理模式的使用方式更加明确后再视需求使用向量处理模式。

三、ECLIC 初始化

ECLIC 是 N308 的中断控制器，相当于 Cortex-M4 的 NVIC，在 CM32M4xxR 的

启动过程中会对 ECLIC 进行一些基本的初始化工作，目前进行的初始化工作包括设置**中断阈值**和**中断级别的有效位数**等两项。中断阈值表示中断目标的阈值级别，由 ECLIC 的 mth 寄存器配置，只有高于 mth 寄存器中的值的中断才能够被送入处理器内核进行处理，功能上类似于 Cortex-M4 的 BASEPRI 寄存器。中断级别的有效位数关系到中断优先级的划分。ECLIC 将每个中断的优先级分为两部分：中断级别（level）和中断优先级（priority），这两者的关系类似于 Cortex-M4 中的分组优先级和子优先级的关系，前者决定是否可以形成中断嵌套，后者决定在多个相同 level 的中断同时挂起的情况下先响应哪个中断，数值越大表示相应的级别或优先级越高。每个中断源均可以通过 clicintctl[i]寄存器设置 level 和 priority 的值，对于 CM32M4xxR，该寄存器中 level 域和 priority 域的总有效位数为 4 位，其中 level 域的位数由 cliccfg 寄存器的 nlbits 域指定，剩下的 4 – nlbits 位即为 priority 域的位数。

ECLIC 的初始化由 ECLIC_Init 函数负责，目前仅是将中断阈值设为 0，即阈值不起作用，并将中断级别的有效位数设为 4，即 clicintctl[i]寄存器中的有效位全部作为 level 域。该函数定义在 system_cm32m4xxr.c 文件中，并在启动过程中被调用，调用关系为：_start → __libc_init_array → _init → **ECLIC_Init** → ECLIC_SetMth/ECLIC_SetCfgNlbits，其中 ECLIC_SetMth 和 ECLIC_SetCfgNlbits 分别设置阈值和中断级别的有效位数，且均为芯来提供的标准接口，在 core_feature_eclic.h 中定义，在功能上 ECLIC_SetCfgNlbits/ECLIC_GetCfgNlbits 对应于 CMSIS 中的 NVIC_SetPriorityGrouping/NVIC_GetPriorityGrouping。用户可在系统启动后重新对上述阈值、中断级别有效位进行自定义配置；

四、中断服务函数

Cortex-M4 在响应中断时硬件会自动保存上下文，因此在编写中断服务函数时无需考虑压栈和出栈，可以像普通函数一样来实现。而 N308 没有硬件压栈机制，因此需要软件压栈和出栈，在非向量和向量两种中断处理模式下，软件需要做的工作及对中断服务函数的要求有些不同。

1、非向量中断模式

由于非向量处理模式时处理器在跳到中断服务程序之前需要先执行一段共有的软件代码进行上下文的保存，因此，从中断源拉高到处理器开始执行中断服务程序中的第一条指令，需要经历中断跳转、CSR 寄存器备份、通用寄存器上下文备份等

多个时钟周期的开销。

由于 RISC-V 内核中断处理时硬件不会主动的完成寄存器的压栈备份和恢复，因此在中断响应时，需要通过软件完成。目前该部分已在 irq_entry 中进行了实现，因此用户在开发中断服务函数时无需考虑；

```
.weak irq_entry
/* This label will be set to MTVT2 register */
irq_entry:
    /* Save the caller saving registers (context) */
    SAVE_CONTEXT
    /* Save the necessary CSR registers */
    SAVE_CSR_CONTEXT
    /* This special CSR read/write operation, which is actually
     * claim the CLIC to find its pending highest ID, if the ID
     * is not 0, then automatically enable the mstatus.MIE, and
     * jump to its vector-entry-label, and update the link register
     */
    csrrw ra, CSR_JALMNXTI, ra
    /* Critical section with interrupts disabled */
    DISABLE_MIE
    /* Restore the necessary CSR registers */
    RESTORE_CSR_CONTEXT
    /* Restore the caller saving registers (context) */
    RESTORE_CONTEXT

    /* Return to regular code */
    mret
```

对于非向量处理模式的中断而言，由于在跳入和退出中断服务程序之前，处理器要进行上下文的保存和恢复，因此在中断响应处理函数中可以实现对其他函数的调用，并且支持中断的嵌套及咬尾特性（进行“中断咬尾”能够节省显著的时间（节省一次背靠背的保存上下文和恢复上下文））。

以下通过一个简单的示例展示如何进行非向量中断配置：本示例采用 TIM3 实现一个 0.5 秒的周期性超时中断，用于控制 LED 进行闪烁。

1.配置 TIM3 的定时周期，计数器分频为 10KHz，超时计数为 5000，合计 0.5 秒；

```
TIM_TimeBaseInitType TIM_TimeBaseStructure = {0};
/* Time base configuration */
TIM_TimeBaseStructure.Period = 5000 - 1;     //0.5s 超时
```

```
TIM_TimeBaseStructure.Prescaler = 7200 - 1;    //分频 10KHz
TIM_TimeBaseStructure.ClkDiv = 0;
TIM_TimeBaseStructure.CntMode = TIM_CNT_MODE_UP;
TIM_InitTimeBase(TIM3, &TIM_TimeBaseStructure);
/* TIM3 enable update irq */
TIM_ConfigInt(TIM3, TIM_INT_UPDATE, ENABLE);
/* TIM3 enable counter */
TIM_Enable(TIM3, ENABLE);
```

2.配置 TIM3 超时中断配置为非向量中断模式，抢占级别为 0，优先级为 1；

```
/* ECLIC Interrupt Config */
ECLIC_SetLevelIRQ(TIM3_IRQn, 0);
ECLIC_SetPriorityIRQ(TIM3_IRQn, 1);
ECLIC_SetTrigIRQ(TIM3_IRQn, ECLIC_LEVEL_TRIGGER);
ECLIC_SetShvIRQ(TIM3_IRQn, ECLIC_NON_VECTOR_INTERRUPT);
ECLIC_EnableIRQ(TIM3_IRQn);
```

3.定义 TIM3 的超时中断处理函数

```
void TIM3_IRQHandler() {
  if (TIM_GetIntStatus(TIM3, TIM_INT_UPDATE) != RESET) {
    TIM_ClrIntPendingBit(TIM3, TIM_INT_UPDATE);
    LedBlink(LED1_PORT, LED1_PIN);
  }
}
```

非向量中断模式下，中断开发与 Cortex-M4 内核系列芯片基本相似，用户可参照以上流程完成中断系统的配置；

2、向量中断模式

向量处理模式时处理器会直接跳到中断服务程序，并没有进行上下文的保存，因此，中断响应延迟非常短，从中断源拉高到处理器开始执行中断服务程序中的第一条指令，基本上只需要硬件进行查表和跳转的时间开销。

在向量处理模式下，由于是直接跳转到中断服务函数执行而没有进行上下文保存，所以理论上不能在中断服务函数中调用子函数，如果调用子函数则需要在中断服务函数前加__attribute__ （（interrupt））修饰，这样编译器会自动插入保存上下文的代码。这种情况下虽然保证了功能的正确性，但是由于保存上下文造成的开销，又会事实上还是增大中断的响应延迟（与非向量模式相当）并且造成代码尺寸（Code Size）的膨胀。因此，在实践中，如果使用向量处理模式，那么不推荐在向量处理模式的中断服务程序函数中调用其他的子函数。

不过需要补充说明的是**在向量处理模式下__attribute__ （（interrupt））关键**

字是必须加上的，因为只有加上该关键字编译器才会生成正确中断退出指令 mret，否则只会生成普通的退出指令 ret，后者是无法正常退出中断的。

以下通过一个简单的示例展示如何进行向量中断配置：本示例采用 TIM3 实现一个 0.5 秒的周期性超时中断，用于控制 LED 进行闪烁。

1.配置 TIM3 的定时周期，计数器分频为 10KHz，超时计数为 5000，合计 0.5 秒；

```
TIM_TimeBaseInitType TIM_TimeBaseStructure = {0};
  /* Time base configuration */
  TIM_TimeBaseStructure.Period = 5000 - 1;     //0.5s 超时
  TIM_TimeBaseStructure.Prescaler = 7200 - 1;    //分频 10KHz
  TIM_TimeBaseStructure.ClkDiv = 0;
  TIM_TimeBaseStructure.CntMode = TIM_CNT_MODE_UP;
  TIM_InitTimeBase(TIM3, &TIM_TimeBaseStructure);
  /* TIM3 enable update irq */
  TIM_ConfigInt(TIM3, TIM_INT_UPDATE, ENABLE);
  /* TIM3 enable counter */
  TIM_Enable(TIM3, ENABLE);
```

2.配置 TIM3 超时中断配置为非向量中断模式，抢占级别为 0，优先级为 2；

```
/* ECLIC Interrupt Config */
  ECLIC_SetLevelIRQ(TIM3_IRQn, 0);
  ECLIC_SetPriorityIRQ(TIM3_IRQn, 2);
  ECLIC_SetTrigIRQ(TIM3_IRQn, ECLIC_LEVEL_TRIGGER);
  ECLIC_SetShvIRQ(TIM3_IRQn, ECLIC_VECTOR_INTERRUPT);
  ECLIC_EnableIRQ(TIM3_IRQn);
```

3.定义 TIM3 的超时中断处理函数，添加__INTERRUPT 修饰为中断处理函数，并支持子函数调用；

```
__INTERRUPT void TIM3_IRQHandler() {
  if (TIM_GetIntStatus(TIM3, TIM_INT_UPDATE) != RESET) {
    TIM_ClrIntPendingBit(TIM3, TIM_INT_UPDATE);


    LedBlink(LED2_PORT, LED2_PIN);
  }
}
```

向量中断模式时，由于在跳入中断服务程序之前，处理器并没有进行任何特殊的处理，且由于处理器内核在响应中断后，mstatus 寄存器中的 MIE 域将会被硬件自动更新成为 0（即全局中断关闭）。因此向量中断模式下默认不支持中断嵌套。但如果用户需要使用向量模式下中断嵌套服务，可在中断服务函数的开头和结束加入必要的 CSR 寄存器特殊处理，接口如下 SAVE_IRQ_CSR_CONTEXT() 和

RESTORE_IRQ_CSR_CONTEXT();

因此上述示例支持嵌套的中断服务函数可修改为：

```
__INTERRUPT void TIM3_IRQHandler() {
  // Save necessary CSRs into variables for vector interrupt nesting
  SAVE_IRQ_CSR_CONTEXT();
  if (TIM_GetIntStatus(TIM3, TIM_INT_UPDATE) != RESET) {
    TIM_ClrIntPendingBit(TIM3, TIM_INT_UPDATE);
    LedBlink(LED2_PORT, LED2_PIN);
  }
  //Restore necessary CSRs from variables for vector interrupt nesting
  RESTORE_IRQ_CSR_CONTEXT();
}
```

五、中断配置

在使用中断时，除了配置前述的中断阈值和中断级别的有效位数之外，针对具体的中断还需对中断处理模式，中断优先级，中断触发方式和中断使能等进行配置。

关于两种中断处理模式的区别前面已简要说明，ECLIC 每一个中断源均可被配置为向量模式或非向量模式，默认使用非向量处理模式，且这里再次建议优先使用非向量处理模式。对于该配置，有一组标准接口 ECLIC_SetShvIRQ/ECLIC_GetShvIRQ 可以使用。

中断优先级配置有 ECLIC_SetLevelIRQ/ECLIC_GetLevelIRQ 和 ECLIC_SetPriorityIRQ/ ECLIC_GetPriorityIRQ 两组标准接口可以使用，前者对应中断级别，后者对应中断优先级，两组接口在功能上对应于 CMSIS 中的 NVIC_SetPriority/NVIC_GetPriority。

在中断触发方式上，ECLIC 中的每个中断源均支持电平和边沿（上升沿或下降沿）触发方式，可以通过 ECLIC_SetTrigIRQ/ECLIC_GetTrigIRQ 这组接口进行设置。需要注意的是，**这里所说的电平或边沿属性并非从外部芯片引脚输入的中断信号的属性，而是指片上外设送入到 ECLIC 的信号属性，其与具体外设有关。**

对于中断使能，每个中断均可单独使能和禁止，另外还可设置全局使能和禁止。单独的中断使能/禁止接口为 ECLIC_EnableIRQ/ECLIC_DisableIRQ，功能上对应于 CMSIS 的 NVIC_SetEnableIRQ/NVIC_GetEnableIRQ，全局使能/禁止接口为

__enable_irq/__disable_irq，与 CMSIS 的接口兼容。

上述接口中除了__enable_irq/__disable_irq 是定义在 core_feature_base.h 中之外，其他接口均定义在 ECLIC 相关的文件 core_feature_eclic.h 中。除此之外，该文件中还定义了 ECLIC_SetPendingIRQ、ECLIC_GetPendingIRQ 和 ECLIC_ClearPendingIRQ 等其他常用接口，分别用于设置、查询和清除中断的挂起状态，且功能上分别对应于 CMSIS 的 NVIC_SetPendingIRQ、NVIC_GetPendingIRQ 和 NVIC_ClearPendingIRQ。其他中断接口及特性可以直接查阅 core_feature_eclic.h 文件。

以上中断配置可以看作外设配置的一部分与外设一同配置。

六、历史版本

版本	日期	修改内容
V1.0	20211028	新建