Technical Note

# AMetal 中断说明

## AMetal

TN01010101 V0.1.0 Date:2018/09/19 工程技术笔记

| 类别 | 内容 |
| --- | --- |
| 关键词 | AMetal、中断 |
| 摘　要 | AMetal 中断说明 |

广州周立功单片机科技有限公司

修订历史

| 版本 | 日期 | 原因 |
| --- | --- | --- |
| 发布 0.1.0 | 2018/9/19 | 创建文档 |
| | | |

# 目 录

# 1. AMetal 中断优先级设置

## 1.1 问题引入

AMetal 致力于应用程序的跨平台复用，而不同芯片对中断优先级的支持（如支持的优先级个数）不尽相同，因此，AMetal 建议应用程序设计得与中断优先级无关，以此保证应用程序可以在任何平台上运行。基于此目的，在 AMetal 中，所有外设中断的默认优先级是相同的，无法嵌套，即一个中断无法打断正在运行的另一个中断。

为避免影响系统的实时性（避免某一中断产生后长时间得不到处理），这就要求任何中断的服务程序占用时间极短，从而保证任何中断都能够快速响应，而不会长时间处于等待前一个中断完成的状态。基于此，我们建议在中断服务函数中，不做任何比较耗时的操作，如 ADC 采集、$I^2C$ 读写、SPI 读写、串口发送以及运行一些复杂的算法等等。

通常情况下，只要不涉及到硬件操作，即使中断服务函数比较耗时，程序一般还能正常运行，只是可能影响系统的实时性（其他中断无法及时响应）。但在一些特殊情况下，若在一个中断服务程序中，等待另一个硬件中断产生并执行相应的中断服务程序，则可能出现卡死现象（因为 AMetal 默认所有外设中断优先级相同，不能嵌套运行，在当前中断运行时，其他中断无法运行，程序将永远处于等待状态）。例如，用户期望每隔一定时间（如 1s）使用 ADC 对某一引脚输入的模拟电压进行采样。出于程序设计的简洁性，用户代码的设计可能如 列表 1.1 所示，这里以 ZLG116 为例。

列表 1.1: ADC 定时采集引脚电压

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"


/**
 * \brief 定时器 3 用户回调函数
 */
void __tim_timing_callback(void* handle_adc)
{
    uint32_t adc_mv;      /* 采样电压 */
    uint16_t val_buf[12]; /* ADC 转换值 */
    uint32_t adc_code;    /* ADC 转换均值   */
    int      i;
    uint32_t sum;

    int adc_bits = am_adc_bits_get(handle_adc, 1); /* 获取 ADC 转换精度 */
    int adc_vref = am_adc_vref_get(handle_adc, 1); /* 获取 ADC 参考电压 */

    am_adc_read(handle_adc, 1, val_buf, 12); /* 启动 ADC 转换器，采集 12 次  */
    for (sum = 0, i = 0; i < 12; i++) { /* 均值处理 */
        sum += val_buf[i];
    }
```

```
    adc_code = sum / 12;

    adc_mv = adc_code * adc_vref / ((1UL << adc_bits) - 1); /* 计算采样电压 */

    /* 串口输出采样电压值 */
    am_kprintf("Sample : %d, Vol: %d mv\r\n", adc_code, adc_mv);
}

void am_main(void)
{
    /* 初始化定时器 3 */
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 初始化 ADC */
    am_adc_handle_t handle_adc = am_zlg116_adc_inst_init();

    /* 设置回调函数 */
    am_timer_callback_set(handle_timer3, 0, __tim_timing_callback, handle_adc);

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    while (1){
        ;
    }
}
```

该程序首先初始化定时器 3 和 ADC，再通过调用定时器用户回调设置函数 am_timer_callback_set() 将定时器 3 的用户回调函数设置为 __tim_timing_callback()，该用户回调由用户自己定义，包含了对 ADC 的一系列操作，如调用 am_adc_read() 开启 AD 转换。函数 am_timer_callback_set() 定义在 AMetal 标准层文件 am_timer.h 中，如 列表 1.2 所示。

列表 1.2: 定时器用户回调设置

```
int am_timer_callback_set (am_timer_handle_t  handle,
                           uint8_t            chan,
                           void              (*pfn_callback)(void *),
                           void               *p_arg);
```

参数 handle 是定时器标准服务操作句柄，由定时器初始化函数返回，即 handle_timer3，参数 chan 是定时器所用的通道，本例中设置为 0，参数 pfn_callback 是用户想要设置的用户回调函数的指针，这里即是 __tim_timing_callback，p_arg 是用户回调函数的参数，即 ADC 标准服务操作句柄 handle_adc，由 ADC 初始化函数返回。

该程序设置定时时间为 1s，每隔 1s 产生一次溢出中断，调用定时器中断用户回调函数 __tim_timing_callback(), 在该用户回调函数中调用函数 am_adc_read() 开启 AD 转换。实际运行可以发现，该程序不能正常运行，会卡死在定时器 3 的中断服务程序中。其原因是 ADC 采样的内部实现也是基于中断实现的，函数 am_adc_read() 的内部过程可以理解为：启动采样、等待采样完成（需要等待采样完成中断通知）、获取采样结果。示意图详见 图 1.1 。

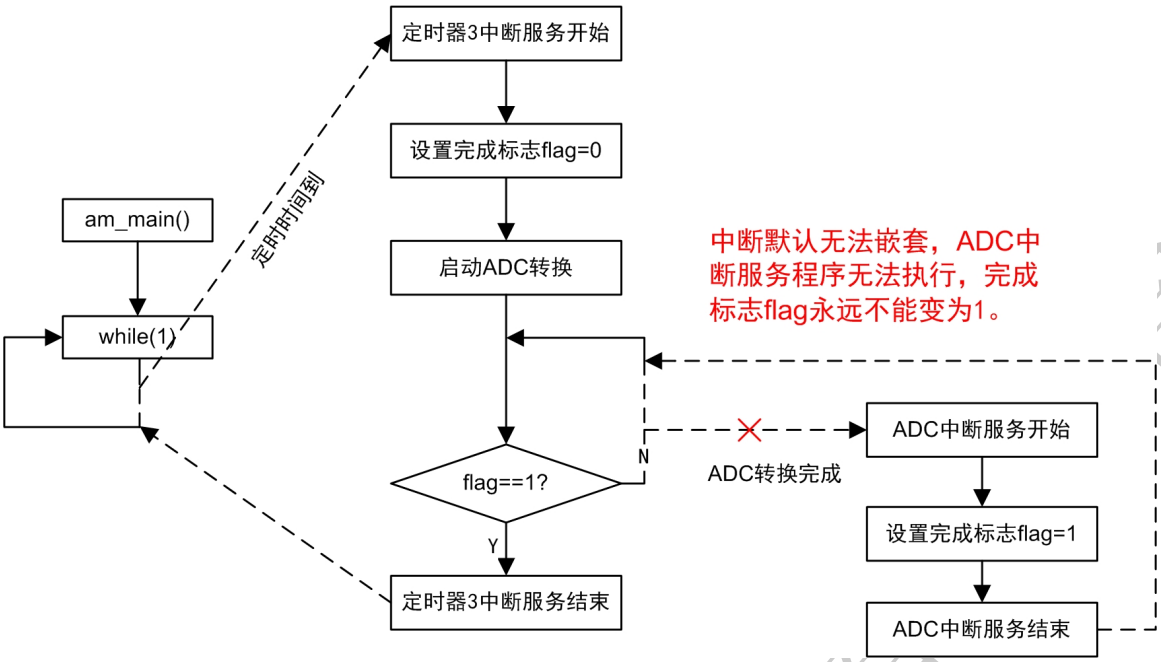


图 1.1: ADC 定时采集引脚电压过程

由于启动采样后需要等待在 ADC 采样完成中断中将完成标志 flag 设置为 1，而 am_adc_read() 的执行环境是在定时器 3 的中断服务程序中，此时，即使 ADC 中断产生，也无法运行（无法打断定时器 3 的中断服务程序）。从而导致 flag 标志无法被设置为 1，定时器 3 中断服务程序中的等待过程将永远无法结束，程序处于卡死状态。

一种简单的解决办法就是使 ADC 转换完成中断可以打断定时器 3 的中断服务程序，flag 标志即可以被设置为 1。这就要求 ADC 的中断优先级需要高于定时器 3 的中断优先级。下节将详细介绍优先级的设置方法，值得注意的是，不同芯片对优先级的支持可能不同，因此，添加了优先级设置代码的应用程序可能无法跨平台复用。

## 1.2 中断优先级设置函数

AMetal 中定义有通过 NVIC（嵌套向量中断控制器）设置中断优先级的函数，用户可在 am_arm_nvic.h 文件中找到该函数原型，支持 cortex-m0、cortex-m0+、cortex-m3、cortex-m4 四种内核。其函数原型如 列表 1.3 所示。

列表 1.3: 中断/异常优先级设置函数

```
int am_arm_nvic_priority_set(int      inum,
                             uint32_t preempt_priority,
                             uint32_t sub_priority);
```

此函数是中断/系统异常优先级设置函数，用于设置外设中断和部分系统异常的优先级。其中，参数 inum 是中断编号，preempt_priority 用于设置抢占优先级（主优先级），sub_priority 用于设置响应优先级（次优先级），函数返回 AM_OK（0）或 AM_EINVAL（22）表示操作成功或参数错误。优先级的数值越低，优先级越高。由于 cortex-m0/cortex-m0+ 内核的芯片不支持优先级分组，只有抢占优先级, 因此使用 cortex-m0/cortex-m0+ 内核芯片时对 sub_priority 的设置无效。下面，本文将对该函数的各个参数进行详细介绍。

### 1.2.1 inum

inum 是用户想要设置优先级的中断/系统异常的中断编号。每一个外设中断和系统异常都有一个唯一的中断编号，由芯片硬件本身决定。当两个中断/系统异常的抢占优先级和响应优先级都相等时，中断编号就会影响中断的响应顺序，中断编号小的被优先处理。

相同内核的 MCU，其系统异常的中断编号是固定的，如 表 1.1 、表 1.2 所示，分别是 cortex-m0/m0+ 和 cortex-m3/m4 的异常类型（包括系统异常和外设中断（IRQ））及其对应的中断编号和优先级，以及其中断编号在 AMetal 中的宏定义。系统异常中，Reset(复位)、NMI（不可屏蔽中断）、HardFault(硬件错误) 的优先级是固定的，不可配置，其余系统异常和外设中断的优先级是可配置的。固定优先级的系统异常其优先级始终高于其余可配置的系统异常和外设中断，优先级的数值越低，其优先级越高，Reset 具有最高优先级。

表 1.1: cortex-m0/m0+ 异常类型

| 异常编号 | 中断编号 | 异常类型 | 优先级 | 中断编号宏定义 |
|---|---|---|---|---|
| 1 | — | Reset | -3(最高) | — |
| 2 | -14 | NMI | -2 | INUM_NMI |
| 3 | -13 | HardFault | -1 | INUM_HARDFAULT |
| 4-10 | — | Reserved | — | — |
| 11 | -5 | SVCall | 可配置 | INUM_SVCALL |
| 12-13 | — | Reserved | — | — |
| 14 | -2 | PendSV | 可配置 | INUM_PENDSV |
| 15 | -1 | SysTick | 可配置 | INUM_SYSTICK |
| 16 and above | — | IRQ | 可配置 | — |

表 1.2: cortex-m3/m4 异常类型

| 异常编号 | 中断编号 | 异常类型 | 优先级 | 中断编号宏定义 |
|---|---|---|---|---|
| 1 | — | Reset | -3(最高) | — |
| 2 | -14 | NMI | -2 | INUM_NMI |
| 3 | -13 | HardFault | -1 | INUM_HARDFAULT |
| 4 | -12 | MemManage | 可配置 | INUM_MEMMANAGE |
| 5 | -11 | BusFault | 可配置 | INUM_BUSFAULT |
| 6 | -10 | UsageFault | 可配置 | INUM_USAGEFAULT |
| 7-10 | — | Reserved | — | — |
| 11 | -5 | SVCall | 可配置 | INUM_SVCALL |
| 12-13 | — | Reserved | — | — |
| 14 | -2 | PendSV | 可配置 | INUM_PENDSV |
| 15 | -1 | SysTick | 可配置 | INUM_SYSTICK |
| 16 and above | — | IRQ | 可配置 | — |

系统异常的中断编号都是小于 0 的，相同内核的芯片其系统异常相同，AMetal 将系统异常的中断编号都定义在了文件 am_arm_nvic.h 中。而外设中断（IRQ）的中断编号都是大于或等于 0 的，具体某个外设中断的中断编号与 MCU 硬件相关,AMetal 将其定义在 xxx_inum.h

文件中（xxx 是芯片型号，如 zlg116_inum.h），用户也可通过查阅芯片手册获得，ZLG116 外设中断编号如 列表 1.4 所示。

列表 1.4: ZLG116 外设中断编号

```
#define INUM_WWDT                 0    /**< \brief 窗口定时器中断 */
#define INUM_PVD                  1    /**< \brief 电源电压检测中断 */

#define INUM_FLASH                3    /**< \brief 闪存全局中断 */
#define INUM_RCC_CRS              4    /**< \brief RCC 和 CRS* 全局中断 */
#define INUM_EXTI0_1              5    /**< \brief EXTI 线 [1: 0] 中断 */
#define INUM_EXTI2_3              6    /**< \brief EXTI 线 [3: 2] 中断 */
#define INUM_EXTI4_15             7    /**< \brief EXTI 线 [15: 4] 中断 */

#define INUM_DMA1_1               9    /**< \brief DMA1 通道 1 全局中断 */
#define INUM_DMA1_2_3             10   /**< \brief DMA1 通道 2、3 全局中断 */
#define INUM_DMA1_4_5             11   /**< \brief DMA1 通道 4、5 全局中断 */
#define INUM_ADC_COMP             12   /**< \brief ADC 和比较器中断 (与 EXTI19、20
组合在一起) */
#define INUM_TIM1_BRK_UP_TRG_COM  13   /**< \brief TIM1 刹车、更新、触发、COM */
#define INUM_TIM1_CC              14   /**< \brief TIM1 捕捉比较中断 */
#define INUM_TIM2                 15   /**< \brief TIM2 全局中断 */
#define INUM_TIM3                 16   /**< \brief TIM3 全局中断 */

#define INUM_TIM14                19   /**< \brief TIM14 全局中断 */

#define INUM_TIM16                21   /**< \brief TIM16 全局中断 */
#define INUM_TIM17                22   /**< \brief TIM17 全局中断 */
#define INUM_I2C1                 23   /**< \brief I2C1 全局中断 */

#define INUM_SPI1                 25   /**< \brief SPI1 全局中断 */
#define INUM_SPI2                 26   /**< \brief SPI2 全局中断 */
#define INUM_UART1                27   /**< \brief UART1 全局中断 */
#define INUM_UART2                28   /**< \brief UART2 全局中断 */
#define INUM_AES                  29   /**< \brief AES 全局中断 */
#define INUM_CAN                  30   /**< \brief CAN 全局中断 */
#define INUM_USB                  31   /**< \brief USB 全局中断 */
```

### 1.2.2 preempt_priority

preempt_priority 用于设置抢占优先级（主优先级），具有高抢占优先级的中断可以在具有低抢占优先级的中断处理过程中被响应，即中断嵌套。当两个中断源的抢占优先级相同时，这两个中断将没有嵌套关系，当一个中断到来后，如果正在处理另一个中断，这个后到来的中断就要等到前一个中断处理完之后才能被处理。其可设置的值的范围与优先级位数及分组有关，设置的值越低，表示其抢占优先级越高。

不同的 MCU 其可用于设置中断/异常优先级的位数是不同的，由芯片硬件本身决定。cortex-m0 及 cortex-m0+ 内核的 MCU（如 ZLG116）其每个外设中断和可配置优先级的系统异常只有两个 bit 可用于设置优先级，cortex-m3 及 cortex-m4 内核的 MCU 每个外设中断和可配置优先级的系统异常有 8 个 bit 可用于设置优先级，但一般芯片厂商在实现时并没有实现全部 8 个位，ARM 规定至少需要实现 3 个位，从高位向低位计，即 [7:5]。用户可通过查阅芯片手册获取芯片支持的优先级位数，也可通过 NVIC 设备信息结构体（定义在 NVIC 用户配置文件 am_hwconf_arm_nvic.c 中）查看。在 ZLG116 中，其所支持的优先级位数如 列表 1.5 所示。

列表 1.5: 芯片支持的优先级位数

```
#define __NVIC_PRIORITY_BITS        2    /**< \brief 芯片支持的优先级位数 */
```

将中断优先级设置位数分成两个部分，一部分用于设置抢占优先级，一部分用于设置响应优先级，这个过程叫做优先级分组。（关于响应优先级，本文将在 1.2.3 节进行介绍）cortex-m0 和 cortex-m0+ 内核的 MCU 不支持优先级分组，只有抢占优先级。cortex-m3 和 cortex-m4 内核 MCU 的优先级分组方式如 表 1.3 所示。

表 1.3: cortex-m3/m4 中断/异常优先级分组

| 优先级组 | 抢占优先级配置位 | 响应优先级配置位 |
|---|---|---|
| 000 | [7:1] | [0] |
| 001 | [7:2] | [1:0] |
| 010 | [7:3] | [2:0] |
| 011 | [7:4] | [3:0] |
| 100 | [7:5] | [4:0] |
| 101 | [7:6] | [5:0] |
| 110 | [7] | [6:0] |
| 111 | None | [7:0] |

例如，一块以 cortex-m3 为内核的 MCU，假设其优先级位数为 4，即在 8 个 bit 中从高位向低位计，有 4 个 bit 可用于设置优先级，若其分组方式设置为 4，即二进制的 100，从 表 1.3 可看出，在 8 个位中，高三位被分配给了抢占优先级，其余分配给了响应优先级。由此可得出，其抢占优先级位数为 3，可设置的范围为 0~7，响应优先级位数为 1，可设置为 0~1。

在 AMetal 中，我们已根据不同的 MCU 对优先级分组进行了默认配置，用户若有需要，可在 NVIC 设备信息结构体（定义在 NVIC 用户配置文件 am_hwconf_arm_nvic.c 中）查看或修改。ZLG116 中优先级分组设置如 列表 1.6 所示（因为 ZLG116 不支持优先级分组，所以实际上此处的设置是无效的）。

列表 1.6: NVIC 中断优先级分组设置

```
#define __NVIC_PRIORITY_GROUP       0    /**< \brief NVIC 中断优先级分组 */
```

### 1.2.3 sub_priority

sub_priority 用于设置响应优先级（次优先级），如果两个中断的抢占优先级相同，且两个中断同时到达，则 CPU 将根据它们的响应优先级高低来决定先处理哪一个。与抢占优先级相同，其可设置的值的范围与优先级位数及分组有关，设置的值越低，表示其响应优先级越高。优先级位数由芯片本身决定，cortex-m3 及 cortex-m4 为内核的 MCU 其分组方式参照 表 1.3 所示。cortex-m0 和 cortex-m0+ 内核的 MCU 没有响应优先级，对该参数的设置无效。

## 1.3 问题解决

为了解决在问题引入部分由于中断优先级相同导致的程序卡死的问题，我们提供了以下解决方案。

### 1.3.1 设置中断优先级

列表 1.1 所示代码之所以在运行时出现卡死，是因为 ADC 中断和定时器 3 中断的优先级相等，ADC 转换完成中断在触发后得不到响应，因此，我们可以通过设置中断优先级，使 ADC 的中断优先级高于定时器 3 中断，在 ZLG116 中，其所支持的中断优先级位数是两位，且不支持优先级分组，所有位均用于设置抢占优先级，因此，其抢占优先级可设置为 0~3，数值越小，优先级越高。基于此，我们可以将定时器 3 中断的抢占优先级设置为 1，ADC 中断的抢占优先级设置为 0，使 ADC 中断的抢占优先级高于定时器 3 中断的抢占优先级。我们将 列表 1.1 所示代码作如下修改，如 列表 1.7 所示。

列表 1.7: ADC 定时采集管脚电压（中断嵌套）

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"

/**
 * \brief 定时器 3 用户回调函数
 */
void __tim_timing_callback(void* handle_adc)
{
    uint32_t adc_mv;      /* 采样电压 */
    uint16_t val_buf[12]; /* ADC 转换值 */
    uint32_t adc_code;    /* ADC 转换均值   */
    int      i;
    uint32_t sum;

    int adc_bits = am_adc_bits_get(handle_adc, 1); /* 获取 ADC 转换精度 */
    int adc_vref = am_adc_vref_get(handle_adc, 1); /* 获取 ADC 参考电压 */

    am_adc_read(handle_adc, 1, val_buf, 12); /* 启动 ADC 转换器，采集 12 次  */

    /* 均值处理 */
    for (sum = 0, i = 0; i < 12; i++) {
        sum += val_buf[i];
    }
    adc_code = sum / 12;

    adc_mv = adc_code * adc_vref / ((1UL << adc_bits) - 1); /* 计算采样电压 */

    /* 串口输出采样电压值 */
    am_kprintf("Sample : %d, Vol: %d mv\r\n", adc_code, adc_mv);
}

void am_main(void)
{
    /* 初始化定时器 3 */
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 初始化 ADC */
    am_adc_handle_t handle_adc = am_zlg116_adc_inst_init();
```

```
    am_arm_nvic_priority_set(INUM_TIM3, 1, 0);      /* 设置定时器 3 中断优先级 */
    am_arm_nvic_priority_set(INUM_ADC_COMP, 0, 0); /* 设置 ADC 中断优先级 */

    /* 设置回调函数 */
    am_timer_callback_set(handle_timer3, 0, __tim_timing_callback, handle_adc);

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    while (1){
        ;
    }
}
```

上述方法通过设置中断的优先级实现了中断嵌套，使得 ADC 中断能够打断定时器 3 中断，避免程序卡死。但实际上 AMetal 并不推荐在应用程序中涉及到中断优先级的设置，原因是不同芯片对中断优先级的支持是不同的，如果一个应用程序涉及到设置中断优先级，那么这个程序很可能就是不可跨平台复用的，AMetal 建议在设计应用程序时避免引入中断优先级。如果遇到需要中断嵌套的情况（如上述情况），我们建议使用其他方法设计程序，例如标志位查询或使用中断延迟处理机制。下面，我们将介绍上述问题基于这两种方法的解决方案。

### 1.3.2 推荐方案（避免使用中断优先级）

（1）. 查询标志位

为了避免在定时器中断中嵌套 ADC 中断，我们可以通过一个定时标志 flag_timer 来记录定时时间是否已到，flag_timer 初始化为 0，定时器 3 每隔 1s 产生一次溢出中断，在定时器中断服务函数中，我们设置标志 flag_timer=1, 表明定时时间已到，随后退出定时器中断，在整个定时器中断服务中不对 ADC 进行任何操作。在主程序的 while(1) 循环中，对定时标志 flag_timer 循环检测，当 flag_timer 的值为 0 时，不做任何操作，继续循环，若 flag_timer 已被置为 1，则表明定时时间到，程序将调用 am_adc_read() 函数实现 AD 转换。示意图详见 图 1.2 。

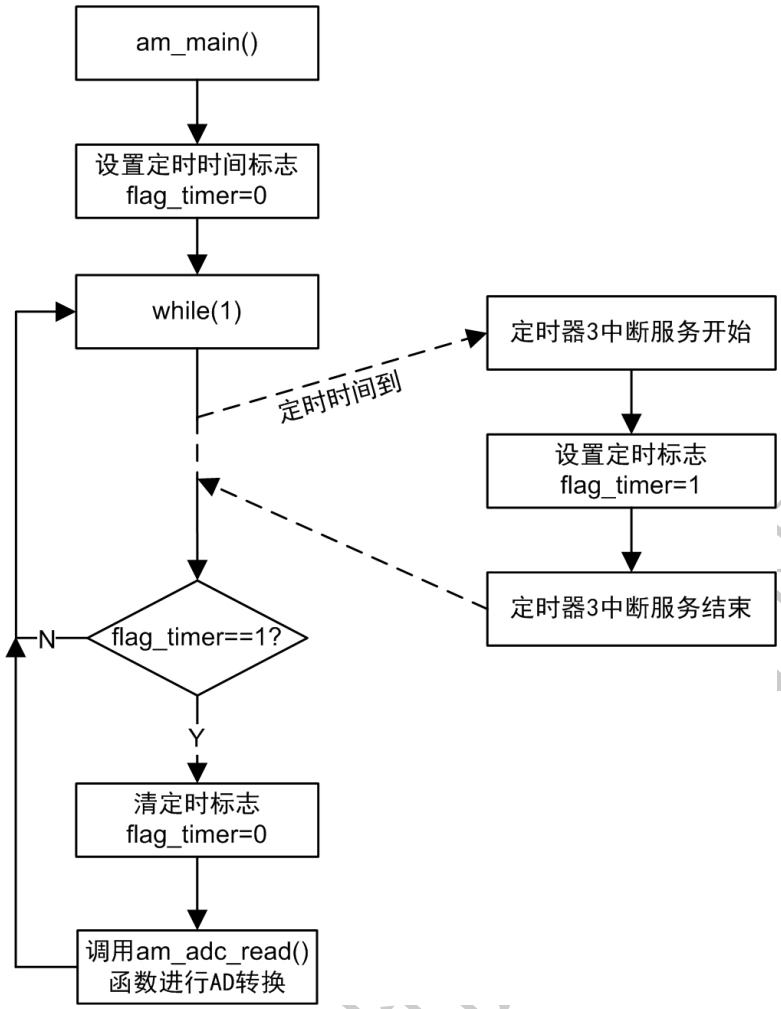


图 1.2: 标志位查询法

程序代码如 列表 1.8 所示。

列表 1.8: ADC 定时采集管脚电压（标志位查询法）

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"

/**
 * \brief 定时器 3 用户回调函数
 */
void __tim_timing_callback(void *p_flag)
{
    *(int *)p_flag = 1;/* 将定时标志置 1 */
}
```

```c
void am_main(void)
{
    /* 初始化定时器 3 */
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 初始化 ADC */
    am_adc_handle_t handle_adc = am_zlg116_adc_inst_init();

    int flag_timer = 0;   /* 设置定时标志 */

    uint32_t adc_mv;      /* 采样电压 */
    uint16_t val_buf[12]; /* ADC 转换值 */
    uint32_t adc_code;    /* ADC 转换均值   */
    int      i;
    uint32_t sum;

    int adc_bits = am_adc_bits_get(handle_adc, 1); /* 获取 ADC 转换精度 */
    int adc_vref = am_adc_vref_get(handle_adc, 1); /* 获取 ADC 参考电压 */

    /* 设置回调函数 */
    am_timer_callback_set(handle_timer3, 0, __tim_timing_callback, &flag_timer);

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    while (1) {

        /* 检测定时标志 */
        if (flag_timer == 1) {

            flag_timer = 0; /* 清定时标志 */

            /* 启动 ADC 转换器，采集 12 次  */
            am_adc_read(handle_adc, 1, val_buf, 12);

            /* 均值处理 */
            for (sum = 0, i = 0; i < 12; i++) {
                sum += val_buf[i];
            }
            adc_code = sum / 12;

            /* 计算采样电压 */
            adc_mv = adc_code * adc_vref / ((1UL << adc_bits) - 1);

            /* 串口输出采样电压值 */
            am_kprintf("Sample : %d, Vol: %d mv\r\n", adc_code, adc_mv);
        }
    }
}
```

程序通过定时标志 flag_timer 记录定时时间是否已到，用于判断是否进入 AD 转换流程，定时器 3 中断和 ADC 中断相互独立，不存在嵌套关系，当 ADC 转换完成中断被触发时，能够立即得到响应。用这种方法编写的应用程序不依赖于中断优先级，可以在不同的芯片上进行复用，但缺点也很明显，即 CPU 需要不断的对定时标志 flag_timer 进行查询，频繁的占用 CPU，效率较低。基于此，我们可以使用中断延迟处理机制来解决类似问题。

（2）. 中断延迟处理机制

在实际应用过程中，往往希望硬件中断函数尽可能快的结束，因此，往往将中断中比较耗时的工作（如 AD 转换、$I^2C$、串口等）放到中断延迟任务队列中，待 CPU 空闲的时候再

对它们进行处理，避免了后面的中断长时间得不到响应，提高了系统的实时性。另外，这种方法也避免了过多的中断嵌套，如上述 ADC 定时测量引脚输入电压，我们可以把对 ADC 的一系列操作（包括标志位设置，开启 AD 转换、等待转换完成等）封装成一个中断延迟处理任务，在定时器中断服务程序中，我们仅仅将该任务添加到中断延迟处理任务队列中, 随即退出定时器中断服务，此时 CPU 空闲，CPU 开始处理 ADC 中断延迟处理任务，进行 AD 转换。我们可以将问题引入部分的程序作如下修改，如 列表 1.9 所示。

列表 1.9: ADC 定时采集管脚电压（中断延迟处理）

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"
#include "am_isr_defer.h"
#include "am_jobq.h"

/**
 * \brief 定时器 3 用户回调函数
 */
void __tim_timing_callback(void * p_job)
{
    /* 添加中断延迟任务到队列 */
    am_isr_defer_job_add ((am_isr_defer_job_t *)p_job);
}

/**
 * \brief ADC 中断延迟任务处理函数
 */
void isr_defer_adc (void *handle_adc)
{

    uint32_t adc_mv;      /* 采样电压 */
    uint16_t val_buf[12]; /* ADC 转换值 */
    uint32_t adc_code;    /* ADC 转换均值   */
    int      i;
    uint32_t sum;

    int adc_bits = am_adc_bits_get(handle_adc, 1); /* 获取 ADC 转换精度 */
    int adc_vref = am_adc_vref_get(handle_adc, 1); /* 获取 ADC 参考电压 */

    /* 启动 ADC 转换器，采集 12 次  */
    am_adc_read(handle_adc, 1, val_buf, 12);

    /* 均值处理 */
    for (sum = 0, i = 0; i < 12; i++) {
    sum += val_buf[i];
    }
    adc_code = sum / 12;

    /* 计算采样电压 */
    adc_mv = adc_code * adc_vref / ((1UL << adc_bits) - 1);

    /* 串口输出采样电压值 */
```

```
    am_kprintf("Sample : %d, Vol: %d mv\r\n", adc_code, adc_mv);
}

void am_main(void)
{
    /* 初始化定时器 3*/
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 初始化 ADC*/
    am_adc_handle_t handle_adc = am_zlg116_adc_inst_init();

    am_isr_defer_job_t isr_defer_job_adc;  /* 定义中断延迟处理任务 */

    /* 初始化任务 */
    am_isr_defer_job_init (&isr_defer_job_adc,
                           isr_defer_adc,
                           handle_adc,
                           0);

    /* 设置回调函数 */
    am_timer_callback_set(handle_timer3,
                          0,
                          __tim_timing_callback,
                         &isr_defer_job_adc);

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    while (1){
        ;
    }
}
```

在该程序中，我们通过中断延迟处理任务类型 am_isr_defer_job_t 定义了一个中断延迟处理任务 isr_defer_job_adc，并调用初始化函数对其进行初始化，其函数原型如 列表 1.10 所示

列表 1.10: 中断延迟处理任务初始化函数

```
void am_isr_defer_job_init (am_isr_defer_job_t  *p_job,
                            am_pfnvoid_t         func,
                            void                *p_arg,
                            uint16_t             pri);
```

该函数中，参数 p_job 是要进行初始化的中断延迟处理任务（即 isr_defer_job_adc）的指针，这里我们通过取地址符号 & 获得，即 &isr_defer_job_adc。参数 func 是该任务的任务处理函数，即函数 isr_defer_adc()，包含对 ADC 的一系列操作，如调用 am_adc_read() 函数等。p_arg 是任务处理函数的参数，本例中传入的是 ADC 外设的句柄，由 ADC 初始化函数 am_zlg116_adc_inst_init() 获得。参数 pri 是该任务的优先级，决定了其在队列中的执行顺序，设置的值越低，则优先被执行，这里的优先级只与其执行顺序相关，不能进行嵌套。AMetal 已经设置好了中断延迟模块使用的优先级数目，用户可在文件 am_bsp_isr_defer_pendsv.h 中查看，如 列表 1.11 所示，表明其可使用的优先级数目为 32，即可设置的值为 0~31，本例中我们将 ADC 中断延迟处理任务的优先级设置为 0。

列表 1.11: 中断延迟模块使用的优先级数目定义

```
/**
 * \brief 定义中断延迟模块使用的优先级数目为 32
 *
 * 只需要使用该宏定义一次，job 的有效优先级即为：0 ~ 31
 *
 */
#define AM_BSP_ISR_DEFER_PRIORITY_NUM     32
```

定时器 3 开启后，每隔 1s 产生一次溢出中断，调用定时器 3 回调函数 __tim_timing_callback()，并传入中断延迟处理任务的指针 &isr_defer_job_adc。在该函数中，我们调用任务添加函数 am_isr_defer_job_add() 将初始化好的中断延迟处理任务添加到任务队列，其函数原型如 列表 1.12 所示。参数 p_job 是要添加的任务的指针，在本例中即是 &isr_defer_job_adc。

列表 1.12: 将一个任务添加到中断延迟工作队列中

```
int am_isr_defer_job_add (am_isr_defer_job_t *p_job);
```

任务添加完成后即退出定时器中断，此时若 CPU 空闲，则会按照任务的优先级，调用各个任务的任务处理函数，对中断延迟处理任务队列中的任务进行处理，在本例中，即调用任务处理函数 isr_defer_adc()，进行 ADC 的相关操作，包括调用 am_adc_read() 函数等，完成 AD 转换。示意图如 图 1.3 所示。

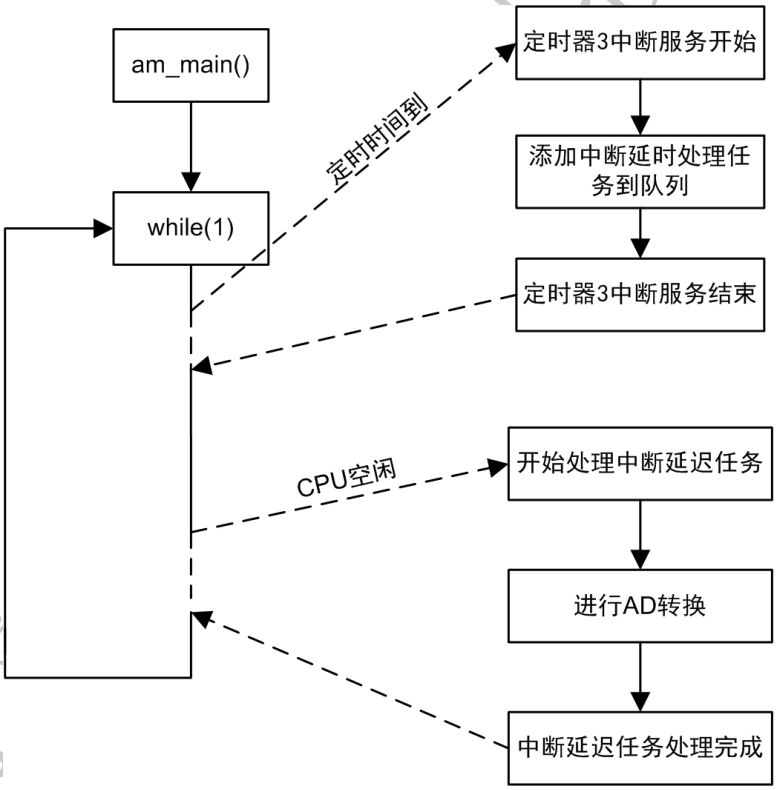


图 1.3: 中断延迟处理

这种方法使用户在设计程序时无需设置中断优先级，中断延迟模块将本应需要嵌套的中断作业放到了中断延迟处理任务队列中，当程序没有其他中断或异常运行时才进行处理。

这种机制适合用于那些需要在中断中执行耗时操作的情况（如 ADC 转换、$I^2C$ 读取、串口通信等），将这些比较耗时的作业都放入中断延迟处理任务队列中，当 CPU 空闲时再对它们进行处理，避免其他中断长时间处于等待状态，提高了系统的实时性。

# 2. AMetal 中断延迟说明

## 2.1 AMetal 中断机制简介

### 2.1.1 AMetal 中断执行流程

在 AMetal 平台上，所有外设中断都映射到同一个中断函数入口 am_exc_eint_handler（该函数定义在 am_arm_nvic.c 文件中），如 列表 2.1 所示，是 ZLG116 中断向量表，定义在启动文件中（gcc 编译器下其定义在文件 am_xxx_gcc_vectors.c 中，armcc 编译器下，其定义在 am_xxx_armcc_startup.s 中，xxx 表示芯片型号，这里以 am_zlg116_gcc_vectors.c 为例）。

列表 2.1: 中断向量表定义

```
void (* const gVectors[])(void) =
{
    (void (*)(void))((unsigned long)&_estack),
    ResetHandler,
    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
    BusFault_Handler,
    UsageFault_Handler,
    0, 0, 0, 0,
    SVC_Handler,
    0,
    0,
    PendSV_Handler,
    SysTick_Handler,

    am_exc_eint_handler,          /* 0 */
    am_exc_eint_handler,          /* 1 */
    0,                            /* 2 */
    am_exc_eint_handler,          /* 3 */
    am_exc_eint_handler,          /* 4 */
    am_exc_eint_handler,          /* 5 */
    am_exc_eint_handler,          /* 6 */
    am_exc_eint_handler,          /* 7 */
    0,                            /* 8 */
    am_exc_eint_handler,          /* 9 */
    am_exc_eint_handler,          /* 10 */
    am_exc_eint_handler,          /* 11 */
    am_exc_eint_handler,          /* 12 */
    am_exc_eint_handler,          /* 13 */
    am_exc_eint_handler,          /* 14 */
    am_exc_eint_handler,          /* 15 */
    am_exc_eint_handler,          /* 16 */
    0,                            /* 17 */
    0,                            /* 18 */
    am_exc_eint_handler,          /* 19 */
    0,                            /* 20 */
    am_exc_eint_handler,          /* 21 */
    am_exc_eint_handler,          /* 22 */
    am_exc_eint_handler,          /* 23 */
    0,                            /* 24 */
    am_exc_eint_handler,          /* 25 */
    am_exc_eint_handler,          /* 26 */
    am_exc_eint_handler,          /* 27 */
    am_exc_eint_handler,          /* 28 */
    am_exc_eint_handler,          /* 29 */
    am_exc_eint_handler,          /* 30 */
```

```
    am_exc_eint_handler          /* 31 */
}; /* gVectors */
```

在外设的初始化过程中,AMetal 会调用中断连接函数 am_int_connect() 将外设实际的中断回调函数放入 NVIC 驱动中，以 ZLG116 的定时器 3 为例，如 列表 2.2 所示。

列表 2.2: 定时器中断初始化

```
/**
 * \brief 定时器中断回调函数
 */
void __tim_irq_handler (void *p_arg)
{
    //...         //检测中断标志位

    //...         //调用相应的定时器用户回调

    //...         //清除中断标志位
}

/**
 * \brief 定时器初始化函数
 */
am_timer_handle_t am_zlg_tim_timing_init (
    am_zlg_tim_timing_dev_t             *p_dev,
    const am_zlg_tim_timing_devinfo_t  *p_devinfo)
{

    //...                   //硬件初始化

    /* 连接中断回调 */
    am_int_connect(p_dev->p_devinfo->inum, __tim_irq_handler, (void *)p_dev);

    //...                   //使能中断
    //...

}
```

函数 am_zlg_tim_timing_init() 是定时器初始化函数，通过调用中断连接函数 am_int_connect() 将某一外设中断（这里是定时器 3 中断）的回调函数 __tim_irq_handler() 放入了 NVIC 驱动中，中断回调连接函数如 列表 2.3 所示，其函数原型声明在 am_int.h 文件中。

列表 2.3: 中断回调连接函数

```
int am_int_connect(int inum, am_pfnvoid_t pfn_isr, void *p_arg);
```

inum 是中断编号，pfn_isr 是中断回调函数的指针，p_arg 是中断回调函数的入口参数，在外设进行初始化时，该函数将外设对应的中断回调放入 NVIC 驱动中。

所有外设中断触发后，程序都会首先进入中断处理函数 am_exc_eint_handler()（如 列表 2.4 所示），在这个函数中，首先获取到对应中断的中断号（本例中即定时器 3 中断号），再根据中断号从 NVIC 驱动中取出对应的中断回调，并进行调用（即 __tim_irq_handler()）。实际上，在 ZLG116 中，所有的定时器中断都被连接到了同一个定时器处理函数 __tim_irq_handler()，如 列表 2.2 所示，在这个函数中，进行中断标志位的检测和清除，并调用相应定时器的用户回调函数，用户回调函数通过函数 am_timer_callback_set() 进行设置。

列表 2.4: 中断处理函数

```
void am_exc_eint_handler (void)
{
    //...       //初始化设备信息

    //...       //获取发生中断的中断编号

    //...       //获取相应中断号的中断回调函数

    //...       //调用中断回调
}
```

AMetal 中断机制如 图 2.4 所示。

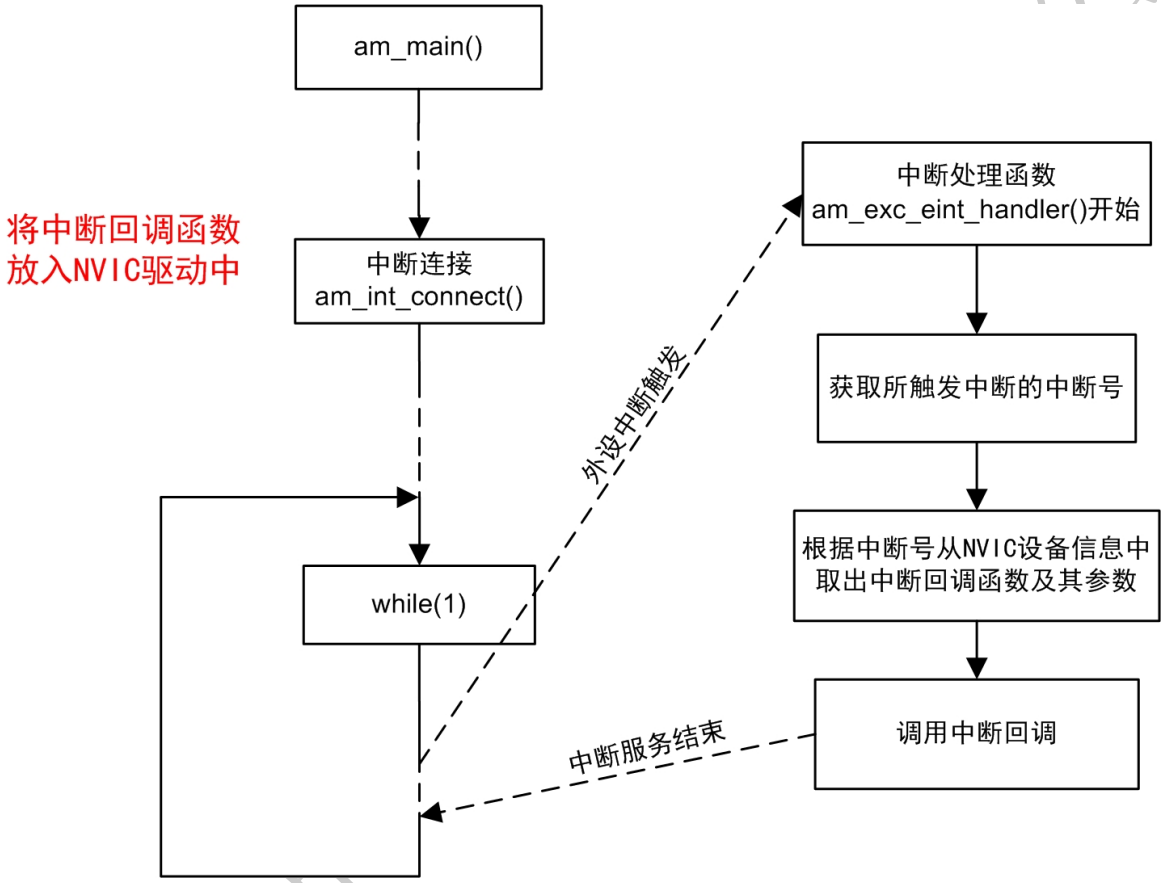


图 2.4: AMetal 中断机制示意图

### 2.1.2 实例详述

普通的中断方式无法实现参数的调用，而 AMetal 平台采用的这种中断方式实现了中断回调函数的含参数调用，实现了数据与代码的分离，大幅提高了代码的可复用性。以 ZLG 定时器中断为例，所有定时器中断连接到同一中断回调函数 __tim_irq_handler()，如 列表 2.2 所示，其参数是定时器结构体指针，根据传入的定时器结构体参数的不同，调用不同的定时器用户回调函数。中断标志位检测和清除这些可复用的语句都被放在了定时器中断处理函数 __tim_irq_handler() 中，用户只需在对应定时器的用户回调函数中实现自己的功能即可，而不必在每个定时器用户回调函数中进行中断标志位检测和清除。

以 ZLG116 定时器 3 中断为例，有如 列表 2.5 所示程序，通过定时器 3 定时 1s，在其溢出中断中翻转 LED 灯的状态。

列表 2.5: 定时器定时翻转 LED

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"

/**
 * \brief 定时器 3 用户回调函数
 */
void __tim_timing_callback(void *p_led_id)
{
    am_led_toggle(*(int*)p_led_id);/* 翻转 LED 状态 */
}

void am_main(void)
{
    int led_id = LED0;

    /* 初始化定时器 3 */
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 设置定时器 3 用户回调函数 */
    am_timer_callback_set(handle_timer3, 0, __tim_timing_callback, &led_id);

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    am_led_on(led_id);/* 点亮 LED */

    while (1) {
        ;
    }
}
```

程序开始运行后，在定时器初始化过程中，通过函数 am_int_connect() 将定时器 3 的中断回调函数 __tim_irq_handler() 放置到了 NVIC 驱动中。函数 am_timer_callback_set() 将定时器 3 的用户回调函数设置为函数 __tim_timing_callback()，定时器用户回调函数由用户自己定义，包含用户想要在定时器中断中执行的具体操作，如 LED 状态翻转，用户回调函数参数设置为指向 LED 编号的指针。

定时时间设置为 1s，当定时时间到达后，定时器 3 发生溢出中断，进入中断处理函数 am_exc_eint_handler(), 并在其中获取到相应的中断号，即定时器 3 的中断号，根据这个中断号，从 NVIC 驱动中取出其在定时器初始化过程中存放的中断回调 __tim_irq_handler() 并对其进行调用。在定时器中断回调函数中，首先检测中断标志位，若中断标志位置 1，则调用定时器 3 的用户回调函数 __tim_timing_callback()，翻转 LED 状态。用户回调执行结束后，清中断标志位，中断服务结束。示意图如 图 2.5 所示。

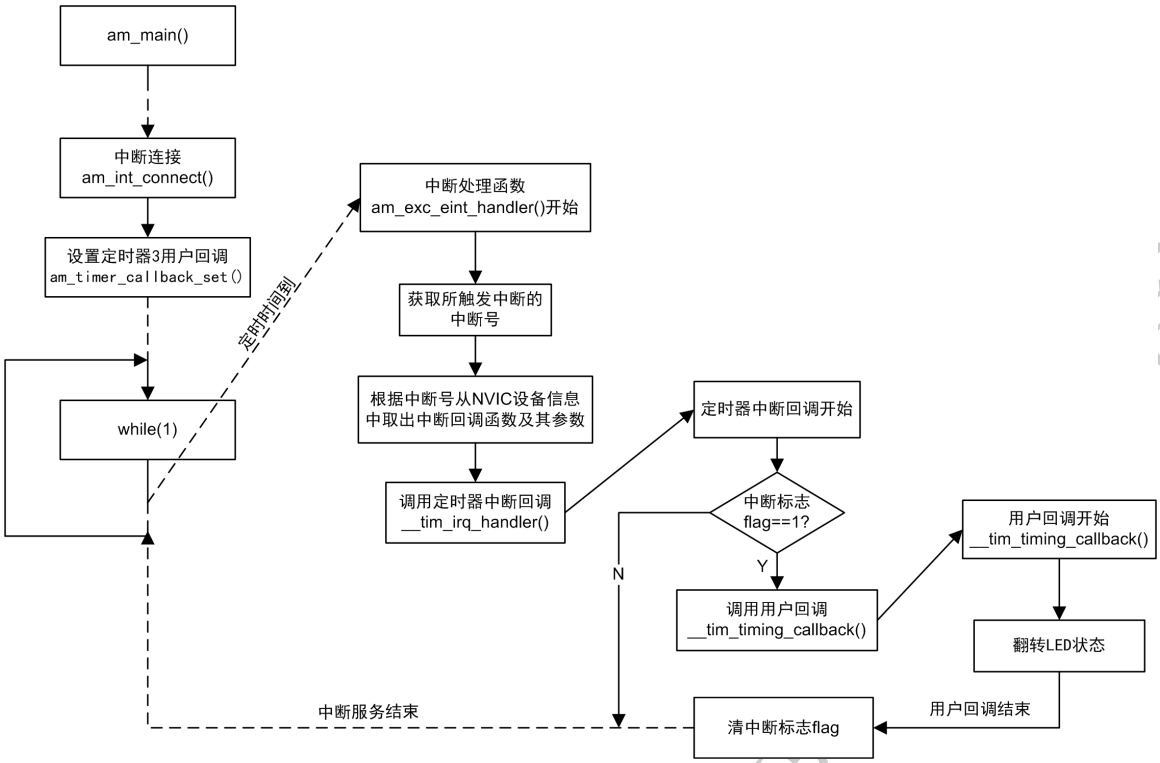


图 2.5: 定时器定时翻转 LED 示意图

## 2.2 AMetal 中断方式修改

虽然这种方式比直接调用中断服务函数（如 图 2.6 所示）多进行了一个中断处理过程，但其实际运行时间极短，经测算，当采用 cortex-m0 内核，系统时钟为 48MH, 分别采用 O0 和 O3 方式优化程序的情况下，中断触发后到中断回调函数开始执行的间隔时间仅仅被延长了约 3.88us 和 1.58us，如 图 2.7 所示。这段时间延迟对中断的实时性影响极小，但在极少数情况下，若用户对中断的实时性要求非常高，这段时间延迟超过了用户的可接受范围，可以按以下方式进行修改，这里以第一小节 ZLG116 定时翻转 LED 状态的程序为例。

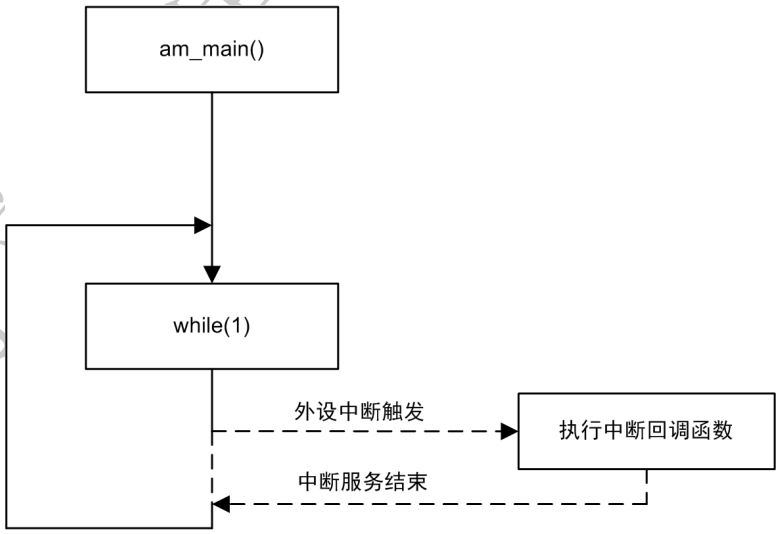


图 2.6: 普通中断机制示意图

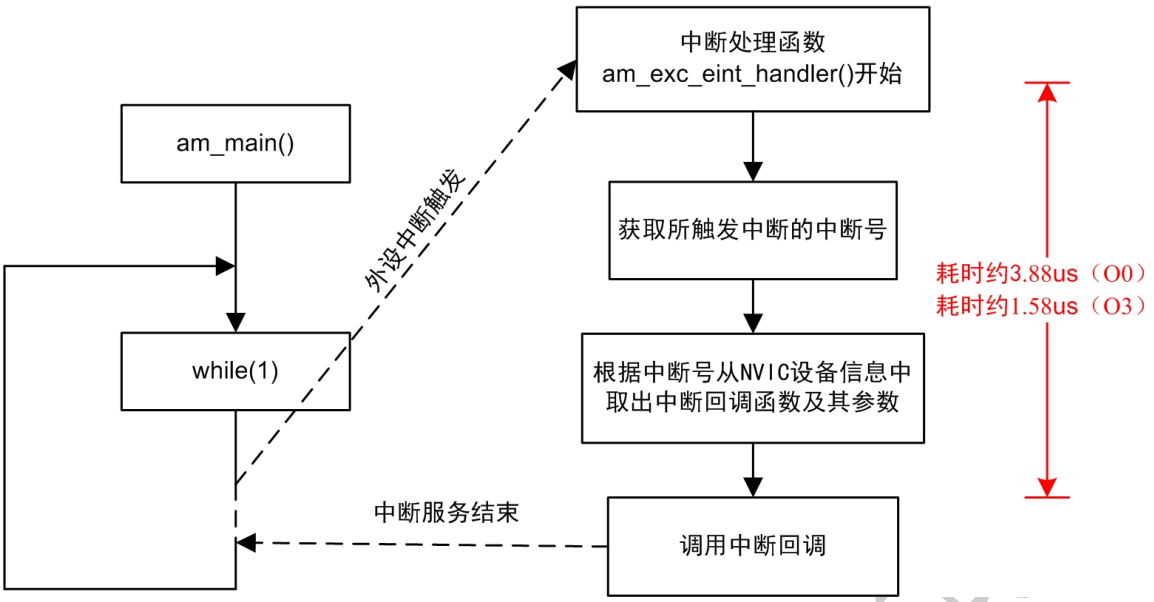


图 2.7: AMetal 中断机制示意图

### 2.2.1 定义并声明中断服务函数

根据用户需求在用户文件（如 main.c 文件）中定义中断服务函数并在启动文件 am_zlg116_gcc_vectors.c 中声明, 用 extern 关键词进行修饰，表明其定义在该文件外部，如列表 2.6 、列表 2.7 所示（若使用的是 armcc 编译器则无需进行声明）。在 AMetal 提供的驱动中，在定时器中断回调函数中，我们进行了中断标志位的检测和清除等操作。因此，若用户不使用 AMetal 提供的中断驱动，则应自己在中断服务函数中实现这些操作，此外，由于这种中断方式无法实现参数调用，因此，LED 编号无法传入，只能使用常量 LED0 替换。

列表 2.6: 定义中断服务函数

```
/**
 * \brief 定时器 3 中断服务函数
 */
void tim3_handler (void)
{

    /* 检测中断标志位 */
    if (amhw_zlg_tim_status_flg_get(ZLG116_TIM3, AMHW_ZLG_TIM_UIF) != 0 ) {

        am_led_toggle (LED0);/* 翻转 LED0 的状态 */

        /* 清除溢出标志 */
        amhw_zlg_tim_status_flg_clr(ZLG116_TIM3, AMHW_ZLG_TIM_UIF)
    }
}
```

列表 2.7: 声明中断服务函数

```
extern void tim3_handler (void);
```

### 2.2.2 修改中断向量表

gcc 编译器和 armcc 编译器下的启动文件不同，中断向量表也不同，我们分别介绍两种编译器下的修改方法。

（1）.gcc 下修改中断向量表

根据用户所使用的中断的中断号在中断向量表中找到相应位置，修改中断入口函数，使其映射到用户自己定义的中断服务函数，这里以 ZLG116 的定时器 3 中断为例，其中断号为 16，找到该中断号对应的位置，并将其位置上的函数名 am_exc_eint_handler 修改为用户自己定义的中断服务函数的函数名，即 tim3_handler，如 列表 2.8 所示。

列表 2.8: 中断向量表 (gcc)

```
void (* const gVectors[])(void) =
{
    (void (*)(void))((unsigned long)&_estack),
    ResetHandler,
    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
    BusFault_Handler,
    UsageFault_Handler,
    0, 0, 0, 0,
    SVC_Handler,
    0,
    0,
    PendSV_Handler,
    SysTick_Handler,

    am_exc_eint_handler,          /* 0 */
    am_exc_eint_handler,          /* 1 */
    0,                            /* 2 */
    am_exc_eint_handler,          /* 3 */
    am_exc_eint_handler,          /* 4 */
    am_exc_eint_handler,          /* 5 */
    am_exc_eint_handler,          /* 6 */
    am_exc_eint_handler,          /* 7 */
    0,                            /* 8 */
    am_exc_eint_handler,          /* 9 */
    am_exc_eint_handler,          /* 10 */
    am_exc_eint_handler,          /* 11 */
    am_exc_eint_handler,          /* 12 */
    am_exc_eint_handler,          /* 13 */
    am_exc_eint_handler,          /* 14 */
    am_exc_eint_handler,          /* 15 */
    tim3_handler,                 /* 16 */
    0,                            /* 17 */
    0,                            /* 18 */
    am_exc_eint_handler,          /* 19 */
    0,                            /* 20 */
    am_exc_eint_handler,          /* 21 */
    am_exc_eint_handler,          /* 22 */
    am_exc_eint_handler,          /* 23 */
    0,                            /* 24 */
    am_exc_eint_handler,          /* 25 */
    am_exc_eint_handler,          /* 26 */
    am_exc_eint_handler,          /* 27 */
    am_exc_eint_handler,          /* 28 */
    am_exc_eint_handler,          /* 29 */
    am_exc_eint_handler,          /* 30 */
```

```
    am_exc_eint_handler          /* 31 */
}; /* gVectors */
```

（2）.armcc 下修改中断向量表

在启动文件 am_zlg116_armcc_startup.s 中修改中断向量表，同样的，将对应位置的函数名修改为自己定义的中断服务函数名，如 列表 2.9 所示，将定时器 3 所对应的位置的函数名由 am_exc_eint_handler 修改为用户自己定义的中断服务函数名 tim3_handler。

列表 2.9: 中断向量表 (armcc)

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler

                ; External Interrupts
                DCD     am_exc_eint_handler       ; Window Watchdog
                DCD     am_exc_eint_handler       ; PVD through EXTI Line detect
                DCD     am_exc_eint_handler       ; RTC through EXTI Line & Tamper
                DCD     am_exc_eint_handler       ; FLASH
                DCD     am_exc_eint_handler       ; RCC & CRS
                DCD     am_exc_eint_handler       ; EXTI Line 0 and 1
                DCD     am_exc_eint_handler       ; EXTI Line 2 and 3
                DCD     am_exc_eint_handler       ; EXTI Line 4 to 15
                DCD     0                         ; Reserved
                DCD     am_exc_eint_handler       ; DMA1 Channel 1
                DCD     am_exc_eint_handler       ; DMA1 Channel 2 and Channel 3
                DCD     am_exc_eint_handler       ; DMA1 Channel 4 and Channel 5
                DCD     am_exc_eint_handler       ; ADC1 & COMP
                DCD     am_exc_eint_handler       ; TIM1 Break, Update, Trigger
                                                  ; and Commutation
                DCD     am_exc_eint_handler       ; TIM1 Capture Compare
                DCD     am_exc_eint_handler       ; TIM2
                DCD     tim3_handler              ; TIM3
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     am_exc_eint_handler       ; TIM14
                DCD     0                         ; Reserved
                DCD     am_exc_eint_handler       ; TIM16
                DCD     am_exc_eint_handler       ; TIM17
                DCD     am_exc_eint_handler       ; I2C1
                DCD     0                         ; Reserved
                DCD     am_exc_eint_handler       ; SPI1
                DCD     am_exc_eint_handler       ; SPI2
                DCD     am_exc_eint_handler       ; UART1
                DCD     am_exc_eint_handler       ; UART2
                DCD     am_exc_eint_handler       ; AES
                DCD     am_exc_eint_handler       ; CAN
                DCD     am_exc_eint_handler       ; USB
```

最后，由于不使用 AMetal 提供的中断驱动，按照以上步骤修改完毕后，我们还需要对主程序做一些调整，如 列表 2.10 所示。我们删除掉定时器中断用户回调的设置，由于在函数 am_timer_enable_us() 中，默认当用户回调为空时，无法使能定时器中断，因此，我们还需要再调用硬件层的定时器相关中断类型使能函数 amhw_zlg_tim_int_enable() 。该函数定义在 amhw_zlg_tim.h 文件中，如 列表 2.11 所示，参数 p_hw_tim 是指向定时器寄存器块的指针，本例中即是指向定时器 3 寄存器块的指针 ZLG116_TIM3（定义在芯片外设映射定义文件 zlg116_periph_map.h 中，需在 main.c 中包含该文件），参数 int_opt 是定时器相关中断类型，其值为 amhw_zlg_tim_int_t 这一枚举类型（定义在 amhw_zlg_tim.h 中），本例中其值为 AMHW_ZLG_TIM_UIE，表示允许更新中断。最后，因为这种中断方式无法将参数传入中断服务函数，所以我们删除掉 led_id 的定义，并将其替换为常量 LED0。

列表 2.10: 主程序（修改后）

```
#include "ametal.h"
#include "am_led.h"
#include "am_delay.h"
#include "am_vdebug.h"
#include "am_board.h"
#include "demo_am116_core_entries.h"
#include "demo_std_entries.h"
#include "zlg116_inum.h"
#include "am_zlg116_inst_init.h"
#include "am_arm_nvic.h"
#include "am_timer.h"
#include "am_adc.h"
#include "zlg116_periph_map.h"

void am_main(void)
{

    /* 初始化定时器 3 */
    am_timer_handle_t handle_timer3 = am_zlg116_tim3_timing_inst_init();

    /* 设置定时时间为 1s （即 1000000us） */
    am_timer_enable_us(handle_timer3, 0, 1000000);

    /* 定时器 3 中断使能 */
    amhw_zlg_tim_int_enable(ZLG116_TIM3, AMHW_ZLG_TIM_UIE);

    am_led_on(LED0);/* 点亮 LED0 */

    while (1) {
        ;
    }
}
```

列表 2.11: 定时器相关中断类型使能函数

```
void amhw_zlg_tim_int_enable (amhw_zlg_tim_t *p_hw_tim, uint8_t int_opt);
```

经过以上步骤，定时器 3 中断触发过后，CPU 就会直接调用用户自己定义的中断服务函数 tim3_handler(), 执行中断标志位的检测，并翻转 LED 状态，随后清除中断标志位。程序在中断触发过后不会首先进入中断处理函数 am_exc_eint_handler() ，减少从中断触发到开始执行中断服务函数的间隔时间。如 图 2.8 所示。

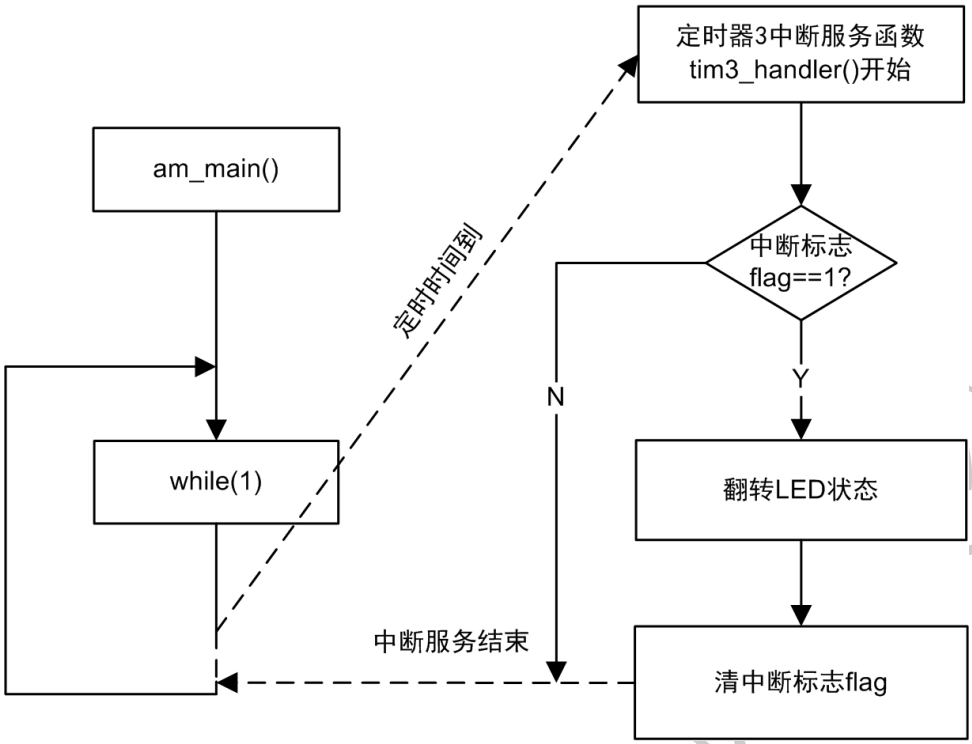


图 2.8: 修改后的中断流程

这种中断方式虽然降低了中断的时间延迟，但也导致了无法对中断回调函数进行含参数调用，无法实现数据与代码的分离，程序可复用性变差。例如本例的 LED 编号，在 AMetal 提供的中断驱动中，其作为用户回调参数，通过定时器用户回调设置函数 am_timer_callback_set() 存放进了定时器设备驱动中。中断发生后，在定时器中断回调函数 __tim_irq_handler() 中，根据传入的定时器设备，将其用户回调取出并调用。对于不同的 LED 灯，只需在主函数中改变变量 led_id 的值即可，其用户回调函数是可复用的。但是，若采用普通的中断方式，则无法实现中断函数的含参数调用，LED 编号无法传入中断服务函数，中断服务函数内的 LED 编号只能用常量代替，对于不同的 LED 灯，只能更改中断服务函数，无法实现程序复用。

# 销售与服务网络


## 广州周立功单片机科技有限公司

地址：广州市天河区龙怡路 117 号银汇大厦 16 楼
邮编：510630
电话：020-38730916 38730917 38730976 38730977
网址：www.zlgmcu.com
传真：020-38730925

### 广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：020-87578634/87569917
传真：020-87578842

### 南京周立功

地址：南京市秦淮区汉中路 27 号友谊广场 17 层 F、G 区
电话：025-68123901/68123902/68123919
传真：025-68123900

### 北京周立功

地址：北京市海淀区紫金数码园 3 号楼（东华合创大厦）8 层 0802 室
电话：010-62635033/62635573/62635884
传真：010-82164433

### 重庆周立功

地址：重庆市渝北区龙溪街道新溉大道 18 号山顶国宾城 11 幢 4-14
电话：023-68796438/68796439/68797619
传真：023-68796439

### 杭州周立功

地址：杭州市西湖区紫荆花路 2 号杭州联合大厦 A 座 4 单元 508
电话：0571-89719484/89719499/89719498
传真：0571-89719494

### 成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室
电话：028-85439836/85432683/85437446
传真：028-68796439

### 深圳周立功（一部）

地址：深圳市福田区深南中路 2072 号电子大厦 1203 室
电话：0755-82941683/82907445
传真：0755-83793285

### 深圳周立功（二部）

地址：深圳市坪山区比亚迪路大万文化广场 A 座 1705
电话：0755-83781788/83782922
传真：0755-83793285

### 武汉周立功

地址：武汉市武昌区武珞路 282 号思特大厦 807 室
电话：027-87168497/87168297/87168397
传真：027-87163755

### 上海周立功

地址：上海市黄浦区北京东路 668 号科技京城东座 12E 室
电话：021-53083451/53083452/53083453
传真：021-53083491

## 周立功厦门办

地址：厦门市思明区厦禾路 855 号英才商厦 618 室
电话：18650195588

## 周立功苏州办

地址：江苏省苏州市广济南路 258 号（百脑汇科技中心 1301 室）
电话：0512-68266786 & 18616749830

## 周立功合肥办

地址：安徽省合肥市蜀山区黄山路 665 号汇峰大厦 1607
电话：13851513746

## 周立功宁波办

地址：浙江省宁波市高新区星海南路 16 号轿辰大厦 1003
电话：0574-87228513/87229313

## 周立功天津办

地址：天津市河东区十一经路与津塘公路交口鼎泰大厦 1004 室
电话：18622359231

## 周立功山东办

地址：山东省青岛市李沧区青山路 689 号宝龙公寓 3 号楼 311
电话：13810794370

## 周立功郑州办

地址：河南郑州市中原区百花路与建设路东南角锦绣华庭 A 座 1502 室
电话：17737307206

## 周立功沈阳办

地址：沈阳市浑南新区营盘西街 17 号万达广场 A4 座 2722 室
电话：18940293816

## 香港周立功

地址：香港新界沙田火炭禾香街 9-15 力坚工业大厦 13 层
电话：(852)26568073 26568077

## 周立功长沙办

地址：湖南省长沙市岳麓区奥克斯广场国际公寓 A 栋 2309 房
电话：0731-85161853