AN0083
入门指南

AT32 MCU Cortex M4内核入门指南

前言

本应用入门指南主要介绍了 AT32 M4 内核系统架构，并针对 M4 内核自带的位带、硬件浮点运算单元和滴答时钟中断功能进行基础讲解和案例解析。

支持型号列表：

支持型号	AT32F4 系列 AT32M4 系列

目录

表目录

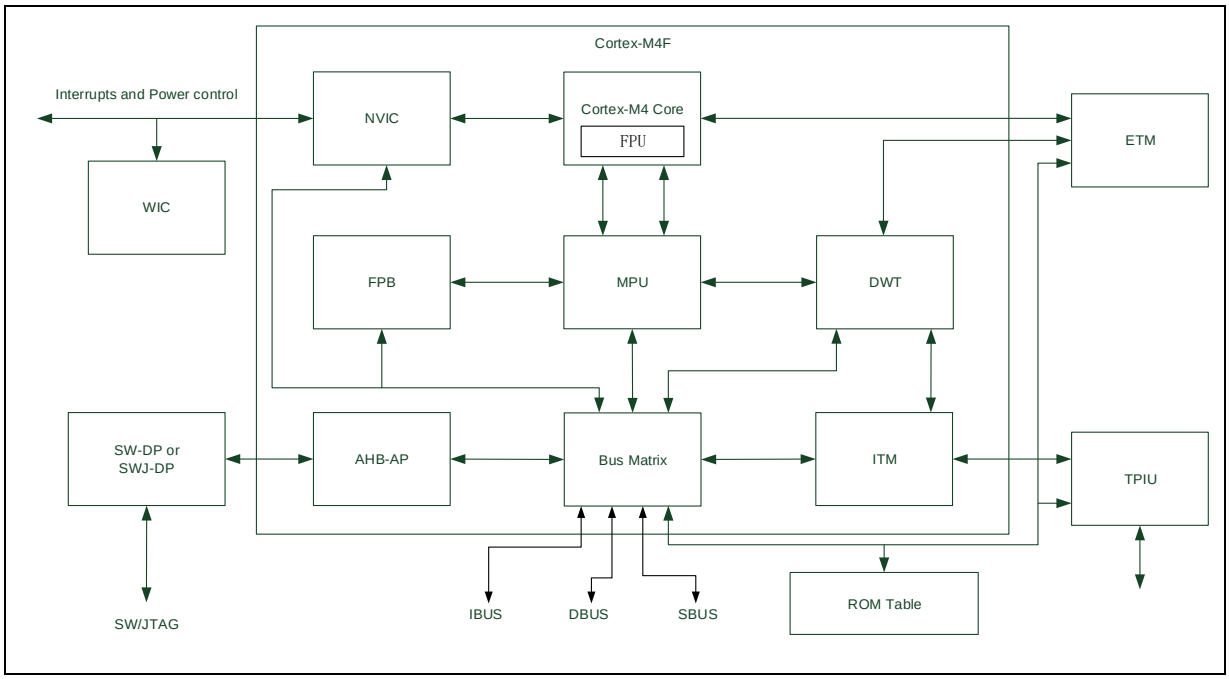
图目录

1 AT32 内核架构概述

AT32F4 系列产品是基于 CortexTM-M4F 处理器架构，该处理器是一款低功耗处理器，具有低门数，低中断延迟和低成本调试的特点。支持包括 DSP 指令集与浮点运算功能，特别适合用于深度嵌入式应用程序需要快速中断响应功能。CortexTM-M4F 处理器是基于 ARMv7-M 架构，既支持 Thumb 指令集也支持 DSP 指令集。

下图为 CortexTM-M4F 处理器的内部框图，请参阅《ARM®Cortex-M4 技术参考手册》了解关于 CortexTM-M4F 更详尽信息。

图 1. AT32 Cortex™-M4F 内部框图



本文主要就 M4 内核自带的位带、硬件浮点运算单元和滴答时钟中断功能进行基础讲解。

2 案例 位带操作

2.1 功能简介

利用位带操作，可以使用普通的加载/存储操作来对单一比特进行读写访问。在 Cortex™-M4F 中提供了两个位带区：SRAM 最低 1M 字节空间和外设区间的最低 1M 字节空间。这两个区中的地址除了可以像普通存储器一样访问外，还可以通过它们各自的位带别名区来快捷访问这两个区中任意地址的任意比特位，位带别名区将位带区每个比特膨胀成一个 32 位的字。当你访问位带别名区的一个地址时，等同于直接访问位带区的一个比特位。

图 2.位带区与位带别名区的膨胀关系图 A

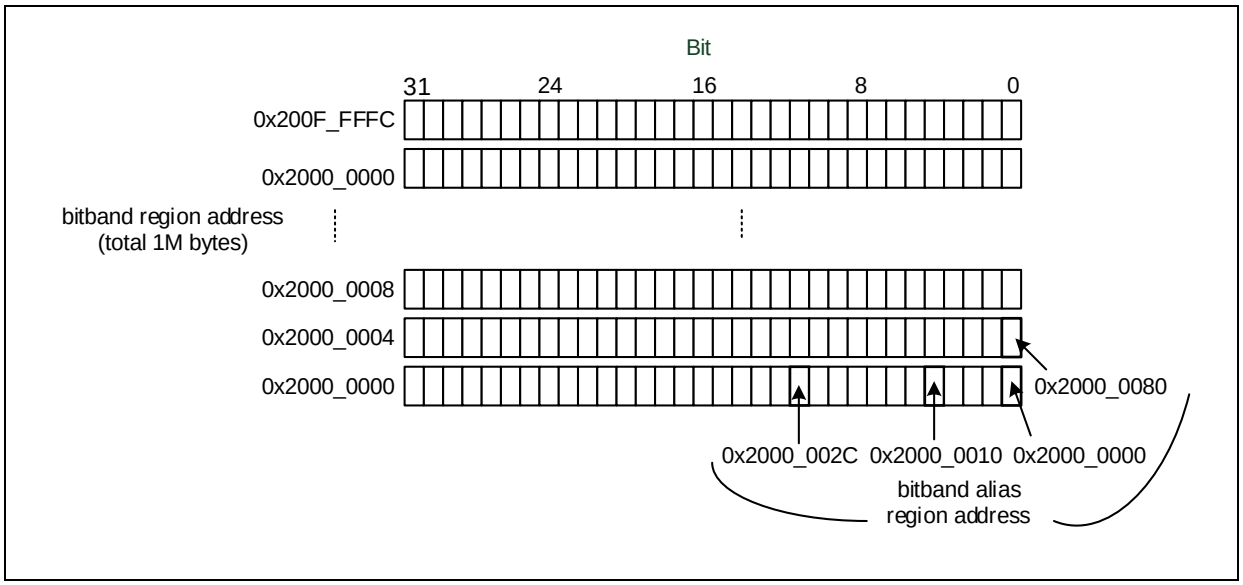
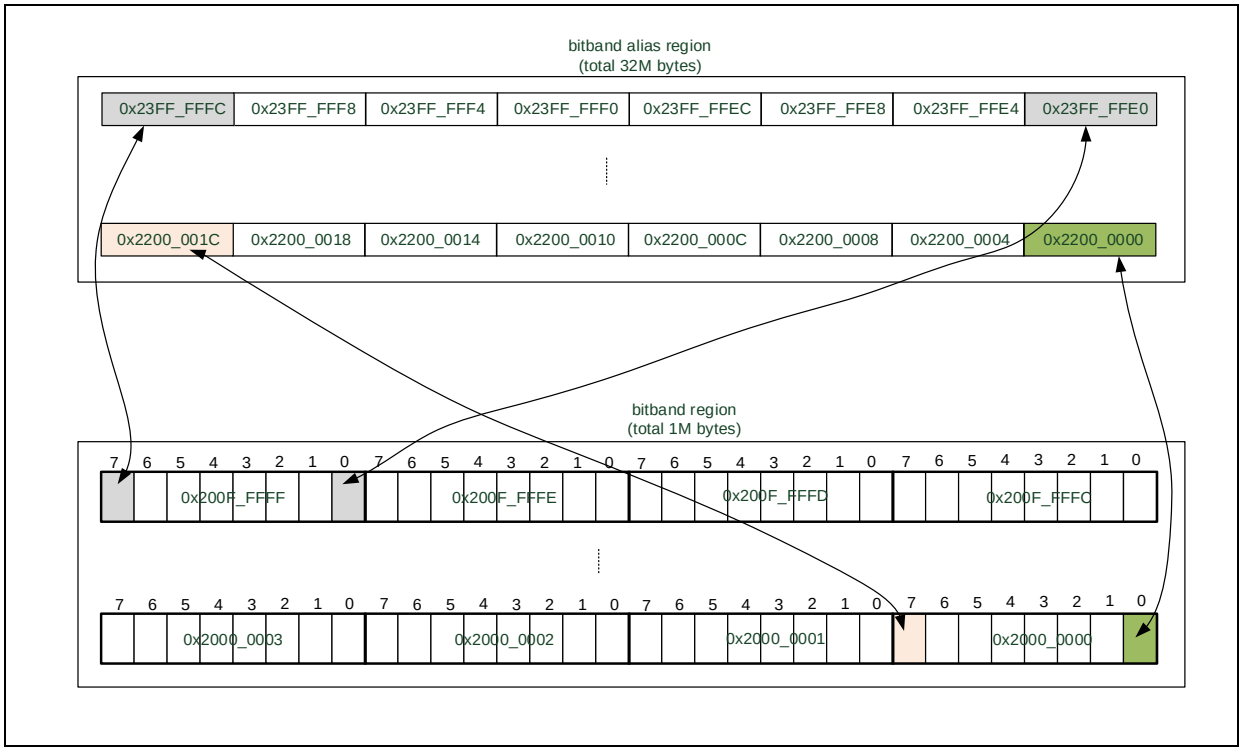


图 3.位带区与位带别名区的膨胀关系图 B



位带区：支持位带操作的地址区

位带别名区：对别名区地址的访问最终作用到位带区的访问上

在位带区中，每个比特都映射到别名地址区的一个字（这是只有 LSB 有效的字）。当一个位带别名区地址被访问时，会先把该地址变换成位带区地址。对于读操作，读取位带区地址中的一个字，再把需要的位右移到 LSB，并把 LSB 返回。对于写操作，把需要写的位左移到对应的位序号处，然后执行一个比特级的“读-改-写”过程。

支持位带操作的两个内存区的地址范围为：

SRAM 区中的最低 1M 字节：0x2000_0000~0x200F_FFFF

外设区间的最低 1M 字节： 0x4000_0000~0x400F_FFFF

对于 SRAM 位带区的某个比特，如果所在字节地址为 A，位序号为 n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr = 0x2200_0000+ (A-0x2000_0000)*32+n*4

对于外设区间位带区的某个比特，如果所在字节地址为 A，位序号为 n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr = 0x4200_0000+ (A-0x4000_0000)*32+n*4

对于 SRAM 区中，位带区与位带别名区的映射如下表所示：

表 1. SRAM 区中的位带地址映射

位带区	等效别名区地址
0x2000_0000.0	0x2200_0000.0
0x2000_0000.1	0x2200_0004.0
0x2000_0000.2	0x2200_0008.0
…	…
0x2000_0000.31	0x2200_007C.0
0x2000_0004.0	0x2200_0080.0
0x2000_0004.1	0x2200_0084.0
0x2000_0004.2	0x2200_0088.0
…	…
0x200F_FFFC.31	0x23FF_FFFC.0

对于外设区中，位带区与位带别名区的映射如下表所示：

表 2.外设区中的位带地址映射

位带区	等效别名区地址
0x4000_0000.0	0x4200_0000.0
0x4000_0000.1	0x4200_0004.0
0x4000_0000.2	0x4200_0008.0
…	…
0x4000_0000.31	0x4200_007C.0

位带区	等效别名区地址
0x4000_0004.0	0x4200_0080.0
0x4000_0004.1	0x4200_0084.0
0x4000_0004.2	0x4200_0088.0
…	…
0x400F_FFFC.31	0x43FF_FFFC.0

位带操作的优越性最容易想到的是通过 GPIO 的管脚来单独控制每盏 LED 的点亮与熄灭。另一方面，也对操作串行接口提供很大的方便。总之，位带操作对于硬件 I/O 密集型的底层程序最有用处。

位带操作还能简化跳转的判断。当跳转依据是某个位时，以前必须这样做：

读取整个寄存器

屏蔽不需要的位

比较并跳转

现在只需要：

从位带别名区读取该位的状态

比较并跳转

使代码更简洁，这只是位带操作优越性的初步体现，位带操作还有一个重要的好处是在多任务以及多任务环境中，将以前的读-改-写需要的三条指令，做成了一个硬件级别支持的原子操作，消除了以前读-改-写可能被中断，导致出现紊乱的情况。

2.2 注意事项

1) 因各系列的外设 IP 地址排布的不同，AT32F421xx 与 AT32F425xx 系列的 GPIO 外设基地址不在位带映射地址范围内。

2.3 资源准备

1) 硬件环境:

   对应产品型号的 AT-START BOARD

2) 软件环境

   project\at_start_f4xx\examples\cortex_m4\bit_band

2.4 软件设计

1) 配置流程

   SRAM 位带操作

   - 定义全局变量 variables = 0xA5A5A5A5，
   - 对 variables　bit0 的位带地址写 0
   - 检查 variables 是否修改为 0xA5A5A5A4，如果是则表示操作成功
   - 对 variables　bit0 的位带地址写 1
   - 检查 variables 是否修改为 0xA5A5A5A5，如果是则表示操作成功
   - 对 variables　bit16 的位带地址写 0
   - 检查 variables 是否修改为 0xA5A4A5A5，如果是则表示操作成功

- 对 variables bit16 的位带地址写 1
- 检查 variables 是否修改为 0xA5A5A5A5，如果是则表示操作成功
- 对 variables bit31 的位带地址写 0
- 检查 variables 是否修改为 0x25A5A5A5，如果是则表示操作成功
- 对 variables bit31 的位带地址写 1
- 检查 variables 是否修改为 0xA5A5A5A5，如果是则表示操作成功

外设位带操作

- 对 LED2 对应 GPIO ODT 寄存器 bit 位的位带地址写 0，
- 对 LED2 对应 GPIO ODT 寄存器 bit 位的位带地址写 1
- 循环执行上述操作，实现 LED toggle 功能

2) 代码介绍

main 函数代码描述

```
int main(void)
{
  /* 系统时钟配置 */
  system_clock_config();
  /* LED 延时函数等初始化 */
  at32_board_init();
  /* 初始化变量 */
  variables = 0xA5A5A5A5;
  /* 获取变量地址 */
  variables_addr = (uint32_t)&variables;

  /* 变量 variables bit0 位带写 0 并检查操作结果 */
  VARIABLES_RESET_BIT(variables_addr, 0);
  if((variables != 0xA5A5A5A4) || (VARIABLES_GET_BIT(variables_addr, 0) != 0))
  {
    result_error();
  }

  /* 变量 variables bit0 位带写 1 并检查操作结果 */
  VARIABLES_SET_BIT(variables_addr, 0);
  if((variables != 0xA5A5A5A5) || (VARIABLES_GET_BIT(variables_addr, 0) != 1))
  {
    result_error();
  }

  /* 变量 variables bit16 位带写 0 并检查操作结果 */
  VARIABLES_RESET_BIT(variables_addr, 16);
  if((variables != 0xA5A4A5A5) || (VARIABLES_GET_BIT(variables_addr, 16) != 0))
  {
```

```
    result_error();
  }

  /* 变量 variables bit16 位带写 1 并检查操作结果 */
  VARIABLES_SET_BIT(variables_addr, 16);
  if((variables != 0xA5A5A5A5) || (VARIABLES_GET_BIT(variables_addr, 16) != 1))
  {
    result_error();
  }

  /* 变量 variables bit31 位带写 0 并检查操作结果 */
  VARIABLES_RESET_BIT(variables_addr, 31);
  if((variables != 0x25A5A5A5) || (VARIABLES_GET_BIT(variables_addr, 31) != 0))
  {
    result_error();
  }

  /* 变量 variables bit31 位带写 1 并检查操作结果 */
  VARIABLES_SET_BIT(variables_addr, 31);
  if((variables != 0xA5A5A5A5) || (VARIABLES_GET_BIT(variables_addr, 31) != 1))
  {
    result_error();
  }


  for(;;)
  {
    /* 外设地址的位带操作，实现 LED2 toggle */
    PERIPHERAL_RESET_BIT((uint32_t)&LED2_GPIO->odt, 13);
    delay_ms(500);
    PERIPHERAL_SET_BIT((uint32_t)&LED2_GPIO->odt, 13);
    delay_ms(500);
  }
}
```

宏定义内容描述

```
/* 定义 SRAM 及其位带地址 */
#define RAM_BASE                0x20000000
#define RAM_BITBAND_BASE        0x22000000

/* 定义 SRAM 位带写 0 函数 */
#define VARIABLES_RESET_BIT(variables_addr, bit_number)      \
          (*(uint32_t *)(RAM_BITBAND_BASE + ((variables_addr - RAM_BASE) * 32) + ((bit_number)
* 4)) = 0)
```

```
/* 定义 SRAM 位带写 1 函数 */
#define VARIABLES_SET_BIT(variables_addr, bit_number)          \
          (*(uint32_t *)(RAM_BITBAND_BASE + ((variables_addr - RAM_BASE) * 32) + ((bit_number)
* 4)) = 1)

/* 定义 SRAM 位带值获取函数 */
#define VARIABLES_GET_BIT(variables_addr, bit_number)          \
          (*(uint32_t *)(RAM_BITBAND_BASE + ((variables_addr - RAM_BASE) * 32) + ((bit_number)
* 4)))

/* 定义外设及其位带地址 */
#define PERIPHERAL_BASE                0x40000000
#define PERIPHERAL_BITBAND_BASE        0x42000000

/* 定义外设位带写 0 函数 */
#define PERIPHERAL_RESET_BIT(peripheral_addr, bit_number)      \
          (*(uint32_t *)(PERIPHERAL_BITBAND_BASE + ((peripheral_addr - PERIPHERAL_BASE) *
32) + ((bit_number) * 4)) = 0)

/* 定义外设位带写 1 函数 */
#define PERIPHERAL_SET_BIT(peripheral_addr, bit_number)        \
          (*(uint32_t *)(PERIPHERAL_BITBAND_BASE + ((peripheral_addr - PERIPHERAL_BASE) *
32) + ((bit_number) * 4)) = 1)
```

2.5 实验效果

- SRAM 位带操作：如果不满足预期，LED4 翻转。
- 外设位带操作：如果满足预期，LED2 翻转。

3 案例 硬件浮点运算单元

3.1 功能简介

FPU 即浮点运算单元（Float Point Unit）。浮点运算，对于定点 CPU（没有 FPU 的 CPU）来说必须要按照 IEEE-754 标准的算法来完成运算，是相当耗费时间的。而对于有 FPU 的 CPU 来说，浮点运算则只是几条指令的事情，速度相当快。

AT32F4 属于 Cortex M4F 架构，带有 32 位单精度硬件 FPU，支持浮点指令集，相对于 Cortex M0 和 Cortex M3 等，高出数十倍甚至上百倍的运算性能

3.2 注意事项

1) 由各系列应用方向及成本的综合考虑，AT32F415xx、AT32F421xx 和 AT32F425xx 系列不支持硬件浮点运算单元。

3.3 资源准备

1) 硬件环境:

   对应产品型号的 AT-START BOARD

2) 软件环境

   project\at_start_f4xx\examples\cortex_m4\fpu

3.4 软件设计

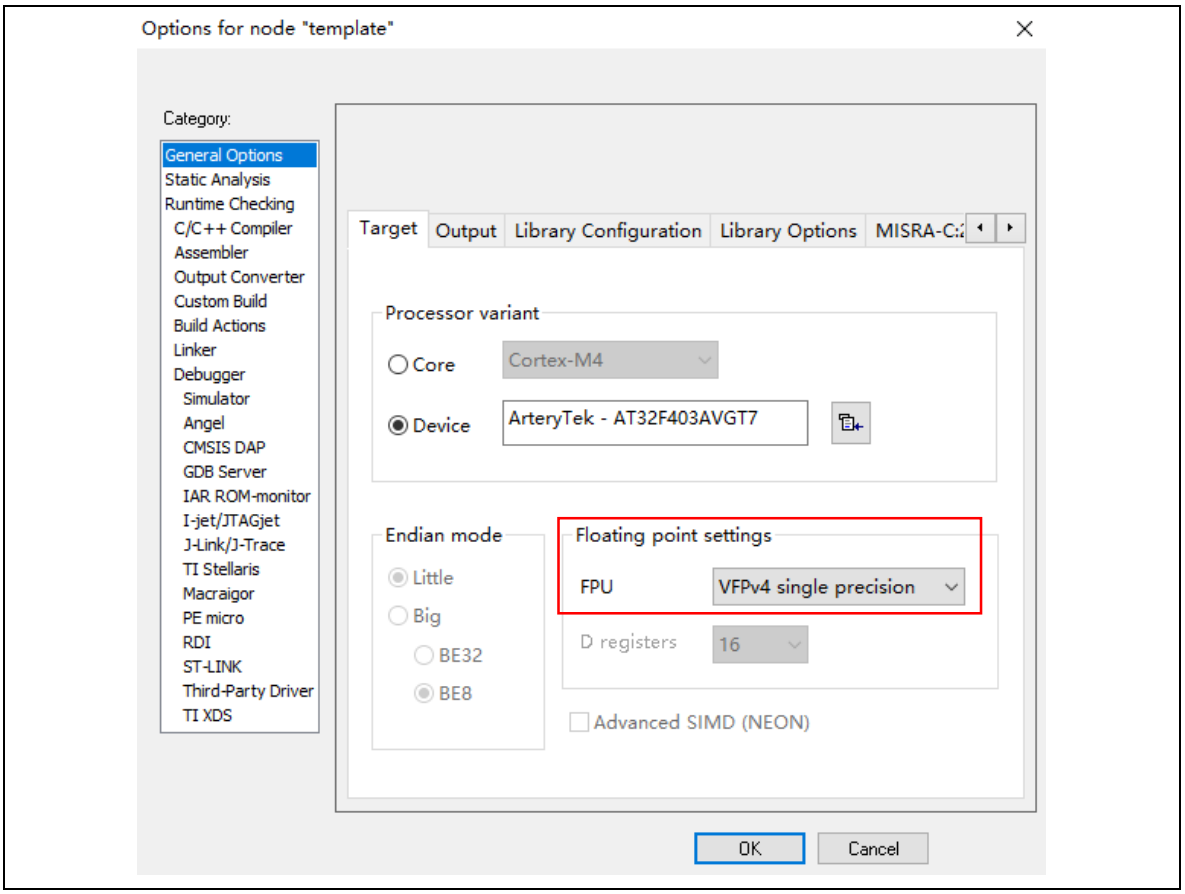
1) 配置流程

   FPU 功能的开启必须要编译器和代码都开启才可以。若只开启编译器 FPU，程序会进入 hardfault；若只开启代码中 FPU，编译器不会编译出 FPU 的代码指令。

   ■ 编译器上开启 FPU 功能

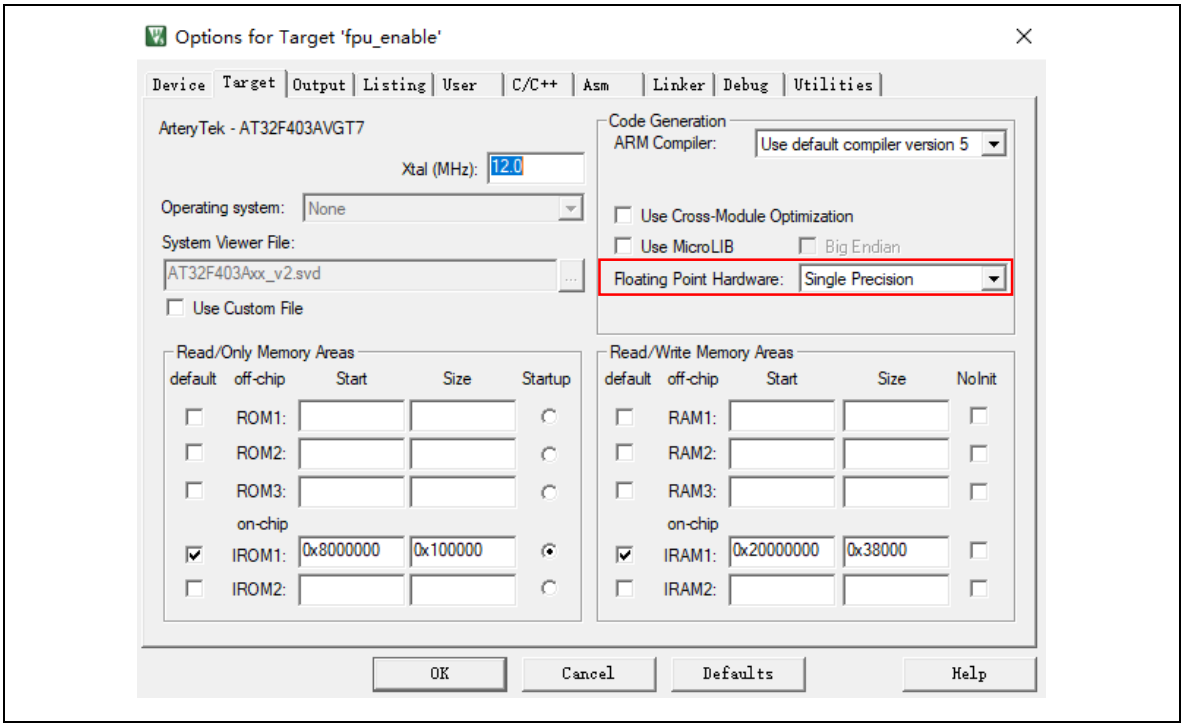
   IAR 开启 FPU 方式如下图

图 4. IAR 开启 FPU 方式



MDK 开启 FPU 方式如下图

图 5. MDK 开启 FPU 方式



■ 代码中开启 FPU 功能

在 system_at32f4xx.c 文件中 void SystemInit (void)函数确保有如下粗斜体代码

```
void SystemInit (void)
{
#if defined (__FPU_USED) && (__FPU_USED == 1U)
  SCB->CPACR |= ((3U << 10U * 2U) |           /* set cp10 full access */
                 (3U << 11U * 2U)  );         /* set cp11 full access */
#endif

  /* reset the crm clock configuration to the default reset state(for debug purpose) */
  /* set hicken bit */
  CRM->ctrl_bit.hicken = TRUE;
  …
}
```

- 执行 Julia 算法函数

比较开启和不开启 FPU 功能的 Julia 运算速度。

2) 代码介绍

main 函数代码描述

```
int main(void)
{
  /* 系统时钟配置 */
  system_clock_config();
  /* LED 延时函数等初始化 */
  at32_board_init();
  for(;;)
  {
    /* LED4 翻转 */
    at32_LED_toggle(LED4);
    /* 执行 julia 算法函数 */
    generate_julia_fpu(SCREEN_X_SIZE, SCREEN_Y_SIZE, SCREEN_X_SIZE / 2,
    SCREEN_Y_SIZE / 2, ZOOM, buffer);
  }
}
```

3.5 实验效果

- 编译器上开启 FPU 功能，观察 LED4 翻转速度。
- 编译器上关闭 FPU 功能，观察 LED4 翻转速度。
- 对比以上两种情形 LED4 翻转速度区别

4 案例 系统滴答时钟中断

4.1 功能简介

系统嘀嗒定时器是一个 24 位递减计数器，递减至零可自动重载计数初值。可产生周期性异常，用作嵌入式操作系统的多任务调度计数器，或对于无嵌入式操作系统，可用于调用需周期性执行的任务。系统嘀嗒定时器校准值固定值 9000，当系统嘀嗒时钟设定为 9MHz，产生 1ms 时间基准。

4.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\cortex_m4\systick_interrupt

4.3 软件设计

1) 配置流程

- 配置 systick 时钟源
- 配置 systick 重载值并开启 systick 中断
- 在 void SysTick_Handler(void) 函数中添加应用代码

2) 代码介绍

main 函数代码描述

```
int main(void)
{
  /* 系统时钟配置 */
  system_clock_config();

  /* 配置 systick 时钟源
     SYSTICK_CLOCK_SOURCE_AHBCLK_NODIV 表示 systick 时钟来源于 AHB 不分频
     SYSTICK_CLOCK_SOURCE_AHBCLK_DIV8  表示 systick 时钟来源于 AHB 8 分频 */
  systick_clock_source_config(SYSTICK_CLOCK_SOURCE_AHBCLK_NODIV);

  /* 配置 systick 重载值并开启 systick 中断 */
  SysTick_Config(MS_TICK);

  /* 初始化 LED2 */
  at32_led_init(LED2);

  for(;;)
  {
  }
}
```

4.4 实验效果

- 本应配置的是 1 ms systick 中断，每进 200 次 systick 中断 LED2 翻转一次，因此应该观察到的现象是 LED2 以 200 ms 一次的频率进行翻转。

5 文档版本历史

表 3. 文档版本历史

日期	版本	变更
2021.5.20	2.0.0	最初版本
2022.5.13	2.0.1	新增支持AT32F415、AT32F421和AT32F425系列，并在部分案例章节增加注意事项

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。