

应用笔记

N32H7xx系列RT-Thread Nano移植应用笔记

简介

此文档的目的在于让使用者能够快速熟悉在 N32H7xx 系列微控制器（MCU）上移植 RT-Thread Nano。

目录

目录	2
1 RT-Thread Nano 移植原理	4
1.1 启动流程	4
1.2 libcpu 移植	5
1.2.1 启动文件 startup.s	5
1.2.2 上下文切换 context_xx.S	5
1.2.3 线程栈初始化 cpuport.c	5
1.2.4 中断与异常挂接 interrupt.c	6
1.3 板级移植 board.c	6
1.3.1 配置系统时钟	7
1.3.2 实现 OS 节拍	7
1.3.3 硬件外设初始化	8
1.3.4 实现动态内存堆	8
2 基于 Keil MDK 移植 RT-Thread Nano	10
2.1 基础工程准备	10
2.2 Nano Pack 安装	10
2.3 添加 RT-Thread Nano 到工程	11
2.4 适配 RT-Thread Nano	13
2.4.1 中断与异常处理	13
2.4.2 系统时钟配置	13
2.4.3 CONSOLE 串口初始化	14
2.4.4 内存堆初始化	16
2.5 编写第一个应用	17
2.6 配置 RT-Thread Nano	18
2.7 注意事项	19
3 基于 IAR 移植 RT-Thread Nano	20
3.1 基础工程准备	20
3.2 下载 RT-Thread Nano 源码	20
3.3 添加 RT-Thread Nano 到工程	21
3.3.1 添加 Nano 源文件	21
3.3.2 添加头文件路径	22
3.4 适配 RT-Thread Nano	23
3.4.1 中断与异常处理	23
3.4.2 系统时钟配置	23
3.4.3 CONSOLE 串口初始化	24
3.4.4 内存堆初始化	26
3.5 编写第一个应用	29
3.6 配置 RT-Thread Nano	29

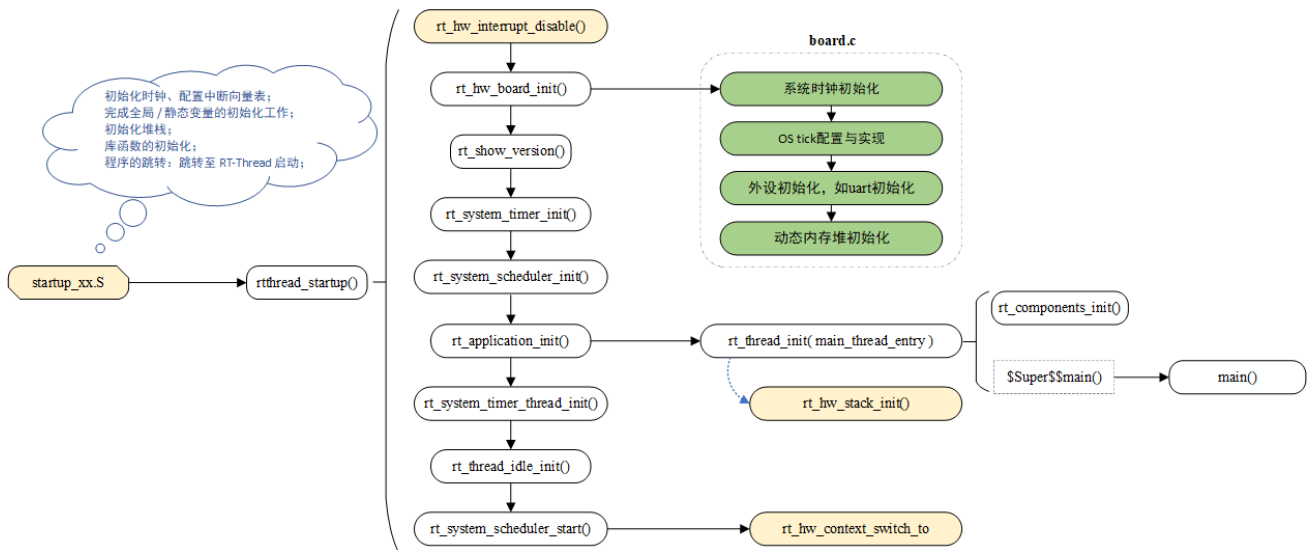
4 版本历史	31
5 声明	32

1 RT-Thread Nano 移植原理

1.1 启动流程

RT-Thread 启动流程如下所示，在图中标出颜色的部分需要用户特别注意（黄色表示 libcpu 移植相关的内容，绿色部分表示板级移植相关的内容）。

图 1-1 RT-Thread 启动流程



RT-Thread 启动代码统一入口为 `rththread_startup()`，芯片启动文件在完成必要工作（如初始化时钟、配置中断向量表、初始化堆栈等）后，最终会在程序跳转时，跳转至 RT-Thread 的启动入口中。RT-Thread 的启动流程如下：

1. 全局关中断，初始化与系统相关的硬件。
2. 打印系统版本信息，初始化系统内核对象（如定时器、调度器）。
3. 初始化用户 `main` 线程（同时会初始化线程栈），在 `main` 线程中对各类模块依次进行初始化。
4. 初始化软件定时器线程、初始化空闲线程。
5. 启动调度器，系统切换到第一个线程开始运行（如 `main` 线程），并打开全局中断。

图 1-2 RT-Thread Nano 源码目录

名称	修改日期	类型	大小
include	2025/6/25 15:22	文件夹	
libcpu	2025/6/25 15:22	文件夹	
src	2025/6/25 15:22	文件夹	
board.c	2024/9/20 16:19	C 文件	2 KB
rtconfig.h	2024/9/20 16:19	H 文件	4 KB

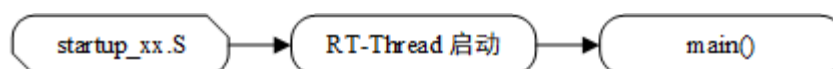
1.2 libcpu 移植

RT-Thread 的 libcpu 抽象层向下提供了一套统一的 CPU 架构移植接口，这部分接口包含了全局中断开关函数、线程上下文切换函数、时钟节拍的配置和中断函数、Cache 等等内容，RT-Thread 支持的 cpu 架构在源码的 libcpu 文件夹下。

1.2.1 启动文件 startup.s

每款芯片都有相对应的启动文件，在不同开发环境下启动文件也不相同，启动文件位于芯片固件库中。当系统加入 RT-Thread 之后，会将 RT-Thread 的启动放在调用 main() 函数之前，如下图所示：

图 1-3 RT-Thread 启动调用



- startup.s: 主要完成初始化时钟、配置中断向量表；完成全局 / 静态变量的初始化工作；初始化堆栈；库函数的初始化；程序的跳转等内容。
- 程序跳转：芯片在 KEIL MDK 与 IAR 下的启动文件不用做修改，会自动转到 RT-Thread 系统启动函数 rththread_startup()。

1.2.2 上下文切换 context_xx.S

上下文切换表示 CPU 从一个线程切换到另一个线程、或者线程与中断之间的切换等。在上下文切换过程中，CPU 一般会停止处理当前运行的代码，并保存当前程序运行的具体位置以便之后继续运行。

表 1-1 context_xx.S 实现函数

需实现的函数	描述
rt_base_t rt_hw_interrupt_disable(void)	关闭全局中断
void rt_hw_interrupt_enable(rt_base_t level)	打开全局中断
void rt_hw_context_switch_to(rt_uint32 to)	没有来源线程的上下文切换，在调度器启动第一个线程的时候调用，以及在 signal 里面会调用
void rt_hw_context_switch(rt_uint32 from, rt_uint32 to)	从 from 线程切换到 to 线程，用于线程和线程之间的切换
void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to)	从 from 线程切换到 to 线程，用于中断里面进行切换的时候使用

注意：在 Cortex-M 中，PendSV 中断处理函数是 PendSV_Handler()，线程切换的实际工作在 PendSV_Handler() 里完成。

1.2.3 线程栈初始化 cpuport.c

在 RT-Thread 中，线程具有独立的栈，当进行线程切换时，会将当前线程的上下文存在栈中，当线程要恢复运行时，再从栈中读取上下文信息，进行恢复。

故障异常处理函数 rt_hw_hard_fault_exception()，在发生硬件错误时，执行 HardFault_Handler 中断，会执

行该函数。

该文件中主要实现线程栈的初始化 `rt_hw_stack_init()` 与 `hard fault` 异常处理函数。

表 1-2 `cpuport.c` 实现函数

需实现的函数	描述
<code>rt_hw_stack_init()</code>	实现线程栈的初始化
<code>rt_hw_hard_fault_exception()</code>	异常函数：系统硬件错误

1.2.4 中断与异常挂接 `interrupt.c`

在 Cortex-M 内核上，所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理，不需要再自行实现中断管理。

1.3 板级移植 `board.c`

注意： `board.c`、`rtconfig.h` 是与硬件 / 板级相关的文件，在移植时需自行实现。Cortex M 架构可参考 Nano 源码 `bsp` 文件夹中已有的 `board.c`、`rtconfig.h`。

板级移植主要是针对 `rt_hw_board_init()` 函数内容的实现，该函数在板级配置文件 `board.c` 中，函数中做了许多系统启动必要的工作，其中包含：

1. 配置系统时钟。
2. 实现 OS 节拍。
3. 初始化外设：如 GPIO/UART 等等，若需要请在此处调用。
4. 初始化系统内存堆，实现动态堆内存管理。
5. 板级自动初始化，使用 `INIT_BOARD_EXPORT()` 自动初始化的函数会在此处被初始化。
6. 其他必要的初始化，如 MMU 配置（需要时请自行在 `rt_hw_board_init` 函数中调用应用函数实现）。

图 1-4 board.c 执行流程

```
/* board.c */
void rt_hw_board_init(void)
{
    /* 第一部分：系统初始化、系统时钟配置等 */
    /* 1：系统层初始化，若需要则增加此部分 */
    /* ... */
    /* 2：系统时钟配置 */
    SystemClock_Config();      // 配置系统时钟
    SystemCoreClockUpdate();   // 更新系统时钟频率 SystemCoreClock

    /* 第二部分：配置 OS Tick 的频率，实现 OS 节拍（并在中断服务例程中实现 OS Tick 递增） */
    _SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND);

    /* 第三部分：初始化硬件外设，若有需要，则放在此处调用 */

    /* 第四部分：系统动态内存堆初始化 */
#ifdef RT_USING_USER_MAIN && defined(RT_USING_HEAP)
    rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif

    /* 第五部分：使用 INIT_BOARD_EXPORT() 进行的初始化 */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

    /* 第六部分：其他初始化 */
}
```

1.3.1 配置系统时钟

系统时钟是给各个硬件模块提供工作时钟的基础，在 `rt_hw_board_init()` 函数中完成，可以调用库函数实现配置，也可以自行实现。

图 1-5 board.c 配置系统时钟示例

```
/* board.c */
void rt_hw_board_init()
{
    /* 第一部分：系统初始化、系统时钟配置等 */
    rt_hw_clock_init()      // 时钟初始化，函数名不做要求，函数自行实现，如 SystemClock_Config()、SystemCoreClockUpdate()
    ...
}
```

1.3.2 实现 OS 节拍

OS 节拍也叫时钟节拍或 OS tick。任何操作系统都需要提供一个时钟节拍，以供系统处理所有和时间有关的事件。

时钟节拍的实现：通过硬件 timer 实现周期性中断，在定时器中断中调用 `rt_tick_increase()` 函数实现全局变量 `rt_tick` 自加，从而实现时钟节拍。一般地，在 Cortex M 上直接使用内部的滴答定时器 SysTick 实现。

图 1-6 OS 节拍示例

```
/* board.c */
void rt_hw_board_init()
{
    ...
    /* 第二部分：配置 OS Tick 的频率，实现 OS 节拍（并在中断服务例程中实现 OS Tick 递增） */
    _SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND); // 使用 SysTick 实现时钟节拍
    ...
}

/* systick 中断服务例程 */
void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}
```

注意：在初始化时钟节拍的时候，会用到宏 `RT_TICK_PER_SECOND`。通过修改该宏的值，可以修改系统中一个时钟节拍的时间长度。

1.3.3 硬件外设初始化

硬件初始化，如 UART 初始化等（对接控制台），需要在 `rt_hw_board_init()` 函数中手动调用 UART 初始化函数。

图 1-7 硬件外设初始化示例

```
/* board.c */
void rt_hw_board_init(void)
{
    ....
    /* 第三部分：初始化硬件外设，若有需要，则放在此处调用 */
    uart_init();
    ....
}
```

注意：`uart_init()` 或者其他的外设初始化函数，若已经使用了宏 `INIT_BOARD_EXPORT()` 进行初始化，则不需要在此进行显式调用。两种初始化方法选择一种即可。

1.3.4 实现动态内存堆

RT-Thread Nano 默认不开启动态内存堆功能，开启 `RT_USING_HEAP` 将可以使用动态内存功能，即可以使用 `rt_malloc`、`rt_free` 以及各种系统动态创建对象的 API。动态内存堆管理功能的初始化是通过 `void rt_system_heap_init(void *begin_addr, void *end_addr)` 函数完成的，动态内存堆的初始化需要指定堆内存的起始地址和结束地址。

开启 RT_USING_HEAP 后，系统默认使用数组作为 heap，heap 的起始地址与结束地址作为参数传入 heap 初始化函数，heap 初始化函数 rt_system_heap_init() 将在 rt_hw_board_init() 中被调用。开启 heap 后，系统中默认使用数组作为 heap（heap 默认较小，实际使用时请根据芯片 RAM 情况改大），获得的 heap 的起始地址与结束地址，作为参数传入 heap 初始化函数：

图 1-8 动态内存堆（数组）初始化示例

```
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE];
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}

void rt_hw_board_init(void)
{
    ....
    #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
        rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get()); //传入 heap 的起始地址与结束地址
    #endif
    ....
}
```

如果不想使用数组作为动态内存堆，则可以重新指定系统 HEAP 的大小，例如使用 RAM ZI 段结尾处作为 HEAP 的起始地址（这里需检查与链接脚本是否对应），使用 RAM 的结尾地址作为 HEAP 的结尾地址，这样可以将空余 RAM 全部作为动态内存 heap 使用。如下示例重新定义了 HEAP 的起始地址与结尾地址，并作为初始化参数进行系统 HEAP 初始化。

图 1-9 动态内存堆（非数组）初始化示例

```
#define N32_SRAM1_START    (0x20000000)
/* End address = 0x20000000(base address) + 20K(RAM Size) */
#define N32_SRAM1_END      (N32_SRAM1_START + 20 * 1024)

#if defined(__CC_ARM) || defined(__CLANG_ARM)
// RW_IRAM1 must correspond to the runtime domain name in the linker script
extern int Image$$RW_IRAM1$$ZI$$Limit;
#define HEAP_BEGIN ((void *)&Image$$RW_IRAM1$$ZI$$Limit)
#endif

#define HEAP_END          (N32_SRAM1_END)

/* board.c */
void rt_hw_board_init(void)
{
    /* Part 4: System Dynamic Memory Heap Initialization */
    #if defined(RT_USING_MAIN) && defined(RT_USING_HEAP)
        rt_system_heap_init((void *)HEAP_BEGIN, (void *)HEAP_END); // The starting and ending addresses of the heap
    #endif
}
```

2 基于 Keil MDK 移植 RT-Thread Nano

RT-Thread Nano 已集成在 Keil MDK 中，可以直接在 IDE 中进行下载添加。移植 Nano 的主要步骤：

1. 准备一个基础的 keil MDK 工程，并获取 RT-Thread Nano pack 安装包并进行安装。
2. 在基础工程中添加 RT-Thread Nano 源码。
3. 适配 Nano，主要从 中断、时钟、内存这几个方面进行适配，实现移植。
4. 验证移植结果：编写第一个应用代码，基于 RT-Thread Nano 闪烁 LED。
5. 最后可对 Nano 进行配置：Nano 是可裁剪的，通过配置文件 rtconfig.h 实现对系统的裁剪。

2.1 基础工程准备

在移植 RT-Thread Nano 之前，需要准备一个能正常运行的裸机工程。

图 2-1 裸机工程示例

```
/**
 **\name    delay.
 **\fun     delay program.
 **\param   none
 **\return  none
 */
static void delay(void)
{
    int i = 0x1000000;
    while (i--);
}

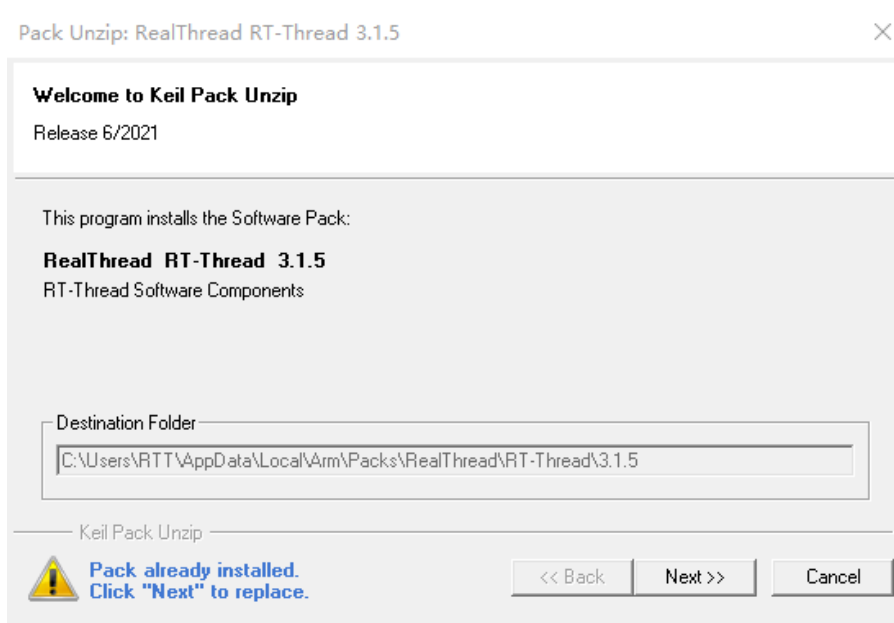
/**
 **\name    main.
 **\fun     Main program.
 **\param   none
 **\return  none
 */
int main(void)
{
    /* Initialize system clock */
    RCC_SetSysClkToMode0();
    /* RCC configuration -----*/
    RCC_Configuration();
    /* GPIO configuration -----*/
    GPIO_Configuration();

    while (1)
    {
        GPIO_SetBits(LED1_PORT, LED1_PIN);
        delay();
        GPIO_ResetBits(LED1_PORT, LED1_PIN);
        delay();
    }
}
```

2.2 Nano Pack 安装

访问 <https://www.keil.arm.com/packs/rt-thread-realthread/versions/> 网址下载 Pack 安装文件，下载结束后双击文件进行安装：

图 2-2 Nano Pack 安装



2.3 添加 RT-Thread Nano 到工程

打开已经准备好的可以运行的裸机程序，将 RT-Thread 添加到工程。如下图，点击 Manage Run-Time Environment。

图 2-3 Nano 添加到工程步骤一

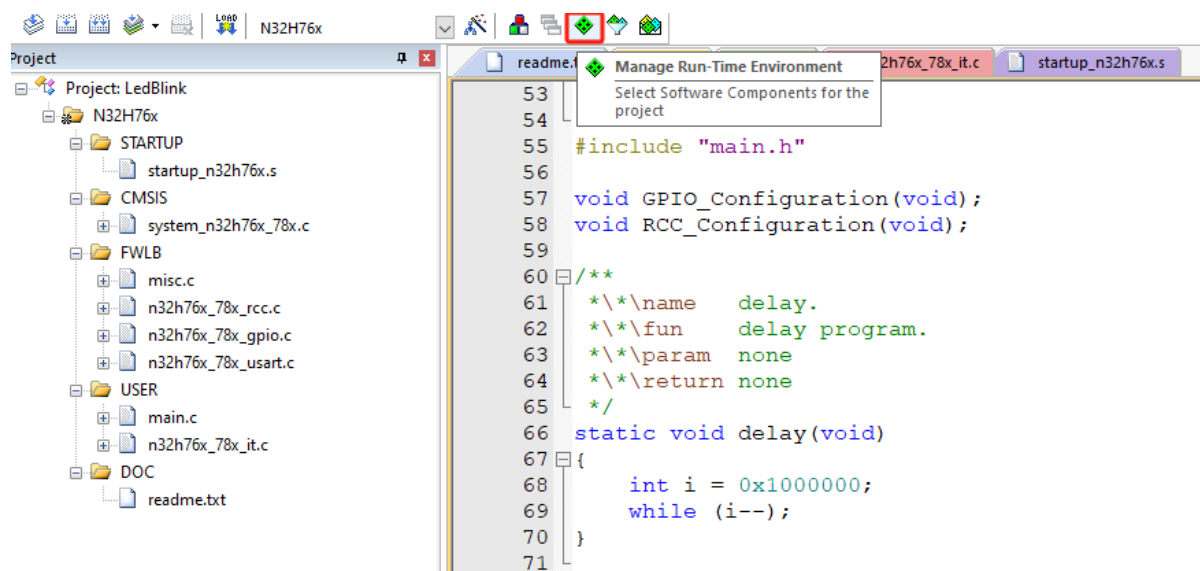
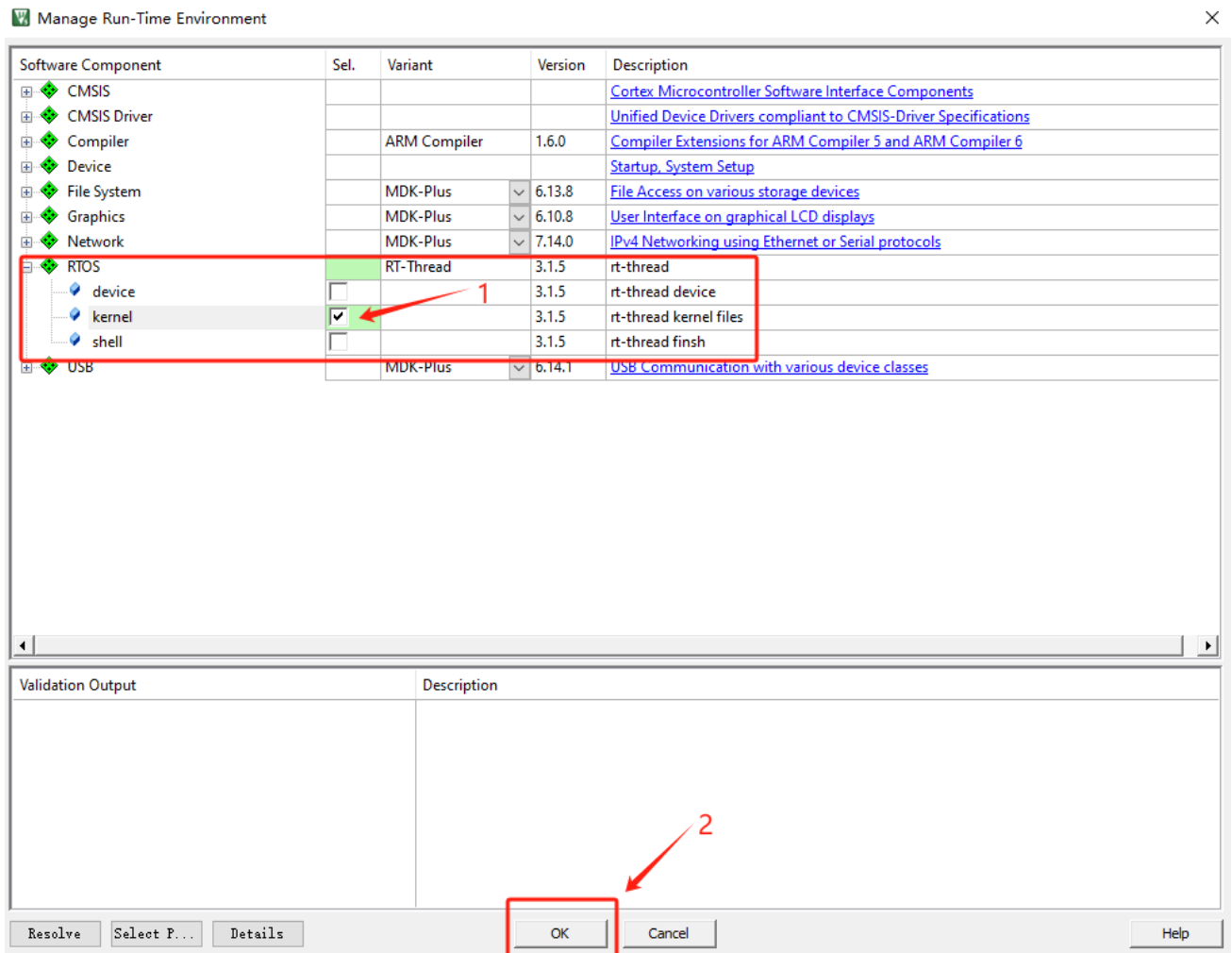
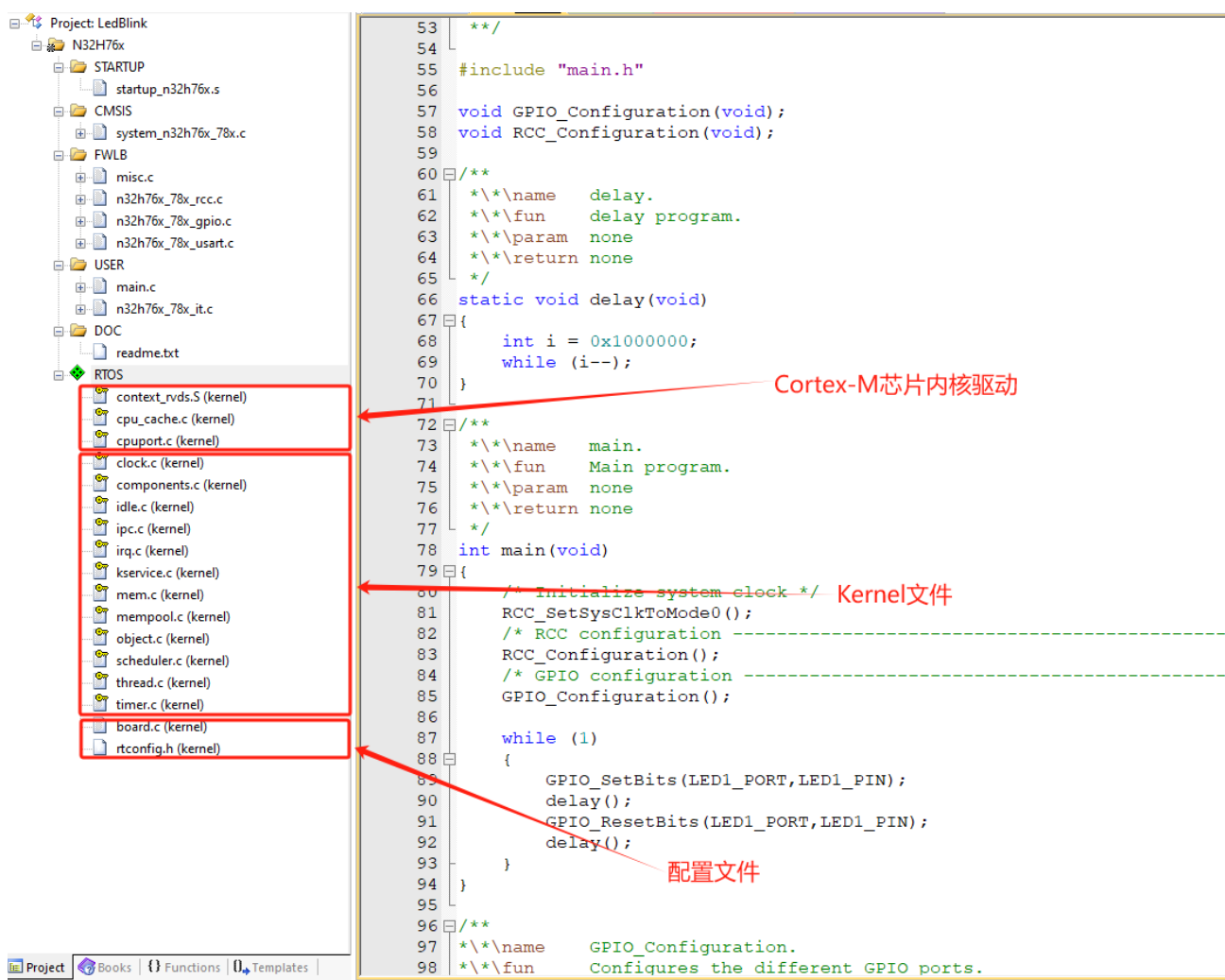


图 2-4 Nano 添加到工程步骤二



现在可以在 Project 看到 RT-Thread RTOS 已经添加进来了，展开 RTOS，可以看到添加到工程的文件：

图 2-5 Nano 添加到工程后工程架构



2.4 适配 RT-Thread Nano

2.4.1 中断与异常处理

RT-Thread 会接管异常处理函数 `HardFault_Handler()` 和悬挂处理函数 `PendSV_Handler()`，这两个函数已由 RT-Thread 实现，所以需要删除工程里 `n32h76x_78x_it.c` 文件中的 `HardFault_Handler()` 和 `PendSV_Handler()` 这两个函数，避免在编译时产生重复定义。如果此时对工程进行编译，没有出现函数重复定义的错误，则不用做修改。

2.4.2 系统时钟配置

需要在 `board.c` 中实现 系统时钟配置（为 MCU、外设提供工作时钟）与 `os tick` 的配置（为操作系统提供心跳 / 节拍）。

如下代码所示，需要在 `board.c` 文件中系统初始化和 OS Tick 的配置，cortex-M 架构需在 `SysTick_Handler()` 函数调用 `rt_os_tick_callback()` 函数。

图 2-6 board.c 中系统时钟与 os tick 配置

```
void rt_os_tick_callback(void)
{
    rt_interrupt_enter();

    rt_tick_increase();

    rt_interrupt_leave();
}

/**
 * \name    SysTick_Handler.
 * \fun     This function handles SysTick exception.
 * \param   none
 * \return  none
 */
void SysTick_Handler(void)
{
    rt_os_tick_callback();
}

/**
 * This function will initial your board.
 */
void rt_hw_board_init(void)
{
    RCC_ClocksTypeDef RCC_Clocks = { 0 };
    /*
     * TODO 1: OS Tick Configuration
     * Enable the hardware timer and call the rt_os_tick_callback function
     * periodically with the frequency RT_TICK_PER_SECOND.
     */

    /* 1: Initialize system clock */
    RCC_SetSysClkToMode0();
    /* frequencies of different on chip clocks */
    RCC_GetClocksFreqValue(&RCC_Clocks);

    /* 2: OS Tick update */
    SysTick_Config(RCC_Clocks.M7ClkFreq / RT_TICK_PER_SECOND);

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#ifdef defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
    rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif
}
```

2.4.3 CONSOLE 串口初始化

在 rtconfig.h 中打开 RT_USING_CONSOLE 使能控制台功能，然后在 board.c 中的 uart_init() 函数实现串口初始化和 rt_hw_console_output() 函数实现控制台输出重定向。

图 2-7 board.c 中控制台串口初始化

```
static int uart_init(void)
{
    /* TODO 2: Enable the hardware uart and config baudrate */
    GPIO_InitType GPIO_InitStructure;
    USART_InitType USART_InitStructure;

    DEBUG_USART_GPIO_APBxClkCmd( DEBUG_USART_GPIO_CLK, ENABLE);

    DEBUG_USART_APBxClkCmd(DEBUG_USART_CLK, ENABLE);

    GPIO_InitStruct(&GPIO_InitStructure);
    GPIO_InitStructure.Pin          = DEBUG_USART_TX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode    = GPIO_MODE_AF_PP;
    GPIO_InitStructure.GPIO_Pull    = GPIO_NO_PULL;
    GPIO_InitStructure.GPIO_Alternate = DEBUG_USART_TX_GPIO_AF;
    GPIO_InitPeripheral( DEBUG_USART_TX_GPIO_PORT, &GPIO_InitStructure );

    GPIO_InitStructure.Pin          = DEBUG_USART_RX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode    = GPIO_MODE_AF_PP;
    GPIO_InitStructure.GPIO_Pull    = GPIO_NO_PULL;
    GPIO_InitStructure.GPIO_Alternate = DEBUG_USART_RX_GPIO_AF;
    GPIO_InitPeripheral( DEBUG_USART_RX_GPIO_PORT, &GPIO_InitStructure );

    USART_StructInit(&USART_InitStructure);
    USART_InitStructure.BaudRate    = DEBUG_USART_BAUDRATE;
    USART_InitStructure.WordLength  = USART_WL_8B;
    USART_InitStructure.StopBits    = USART_STPB_1;
    USART_InitStructure.Parity      = USART_PE_NO ;
    USART_InitStructure.HardwareFlowControl = USART_HFCTRL_NONE;
    USART_InitStructure.Mode        = USART_MODE_RX | USART_MODE_TX;
    USART_Init( LOG_USARTx, &USART_InitStructure );

    USART_Enable( LOG_USARTx, ENABLE );
    return 0;
}
INIT_BOARD_EXPORT(uart_init);
```

图 2-8 board.c 中控制台输出重定向

```
void rt_hw_console_output(const char *str)
{
    /* TODO 3: Output the string 'str' through the uart */
    uint8_t char_r = '\r';
    /* Enter the critical */
    rt_enter_critical();

    /* Until the end of the string */
    while (*str != '\0')
    {
        if (*str == '\n')
        {
            /* Loop until the end of transmission */
            while (USART_GetFlagStatus(LOG_USARTx, USART_FLAG_TXC) == RESET)
            {
            }
            USART_SendData(LOG_USARTx, char_r);
        }

        /* Loop until the end of transmission */
        while (USART_GetFlagStatus(LOG_USARTx, USART_FLAG_TXC) == RESET)
        {
        }
        USART_SendData(LOG_USARTx, *(uint8_t *)str);

        str++;
    }

    /* Exit the critical */
    rt_exit_critical();
}
```

2.4.4 内存堆初始化

系统内存堆的初始化在 board.c 中的 rt_hw_board_init() 函数中完成, 内存堆功能是否使用取决于 rtconfig.h 中宏 RT_USING_HEAP 是否开启, RT-Thread Nano 默认不开启内存堆功能, 这样可以保持一个较小的体积, 不用为内存堆开辟空间。

开启系统 heap 将可以使用动态内存功能, 如使用 rt_malloc、rt_free 以及各种系统动态创建对象的 API。若需要使用系统内存堆功能, 则打开 rtconfig.h 中 RT_USING_HEAP 宏定义即可, 此时内存堆初始化函数 rt_system_heap_init() 将被调用。

图 2-9 rt_system_heap_init()调用

```
/**
 * This function will initial your board.
 */
void rt_hw_board_init(void)
{
    RCC_ClocksTypeDef RCC_Clocks = { 0 };
    /*
     * TODO 1: OS Tick Configuration
     * Enable the hardware timer and call the rt_os_tick_callback function
     * periodically with the frequency RT_TICK_PER_SECOND.
     */

    /* 1: Initialize system clock */
    RCC_SetSysClkToMode0();
    /* frequencies of different on chip clocks */
    RCC_GetClocksFreqValue(&RCC_Clocks);

    /* 2: OS Tick update */
    SysTick_Config(RCC_Clocks.M7ClkFreq / RT_TICK_PER_SECOND);

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

    #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
        rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
    #endif
}
```

初始化内存堆需要堆的起始地址与结束地址这两个参数, 系统中默认使用数组作为 heap, 并获取了 heap 的起始地址与结束地址, 该数组大小可手动更改, 如下所示:

图 2-10 数组作为 heap 大小配置

```

14 #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
15 /*
16  * Please modify RT_HEAP_SIZE if you enable RT_USING_HEAP
17  * the RT_HEAP_SIZE max value = (sram size - ZI size), 1024 means 1024 bytes
18  */
19 #define RT_HEAP_SIZE (15*1024)
20 static rt_uint8_t rt_heap[RT_HEAP_SIZE];
21
22 RT_WEAK void *rt_heap_begin_get(void)
23 {
24     return rt_heap;
25 }
26
27 RT_WEAK void *rt_heap_end_get(void)
28 {
29     return rt_heap + RT_HEAP_SIZE;
30 }
31 #endif

```

注意：开启 heap 动态内存功能后，heap 默认值较小，在使用的时候需要改大，否则可能会有申请内存失败或者创建线程失败的情况，修改方法有以下两种：

- 方法一：可以直接修改数组中定义的 RT_HEAP_SIZE 的大小
- 方法二：使用 RAM ZI 段结尾处作为 HEAP 的起始地址，使用 RAM 的结尾地址作为 HEAP 的结尾地址

2.5 编写第一个应用

移植好 RT-Thread Nano 之后，则可以开始编写第一个应用代码验证移植结果。此时 main() 函数就转变成 RT-Thread 操作系统的一个线程，现在可以在 main() 函数中实现第一个应用：板载 LED 指示灯闪烁，这里直接基于裸机 LED 指示灯进行修改。

1. 首先在文件首部增加 RT-Thread 的相关头文件 <rtthread.h>
2. 在 main() 函数中（也就是在 main 线程中）实现 LED 闪烁代码
3. 将延时函数替换为 RT-Thread 提供的延时函数 rt_thread_mdelay()。该函数会引起系统调度，切换到其他线程运行，体现了线程实时性的特点

图 2-11 LED 闪烁应用示例

```
#include "main.h"
#include "n32h76x_78x_it.h"
#include <rtthread.h>

void GPIO_Configuration(void);
void RCC_Configuration(void);

/**
 * \name      main.
 * \fun       Main program.
 * \param     none
 * \return    none
 */
int main(void)
{
    /* RCC configuration ----- */
    RCC_Configuration();
    /* GPIO configuration ----- */
    GPIO_Configuration();

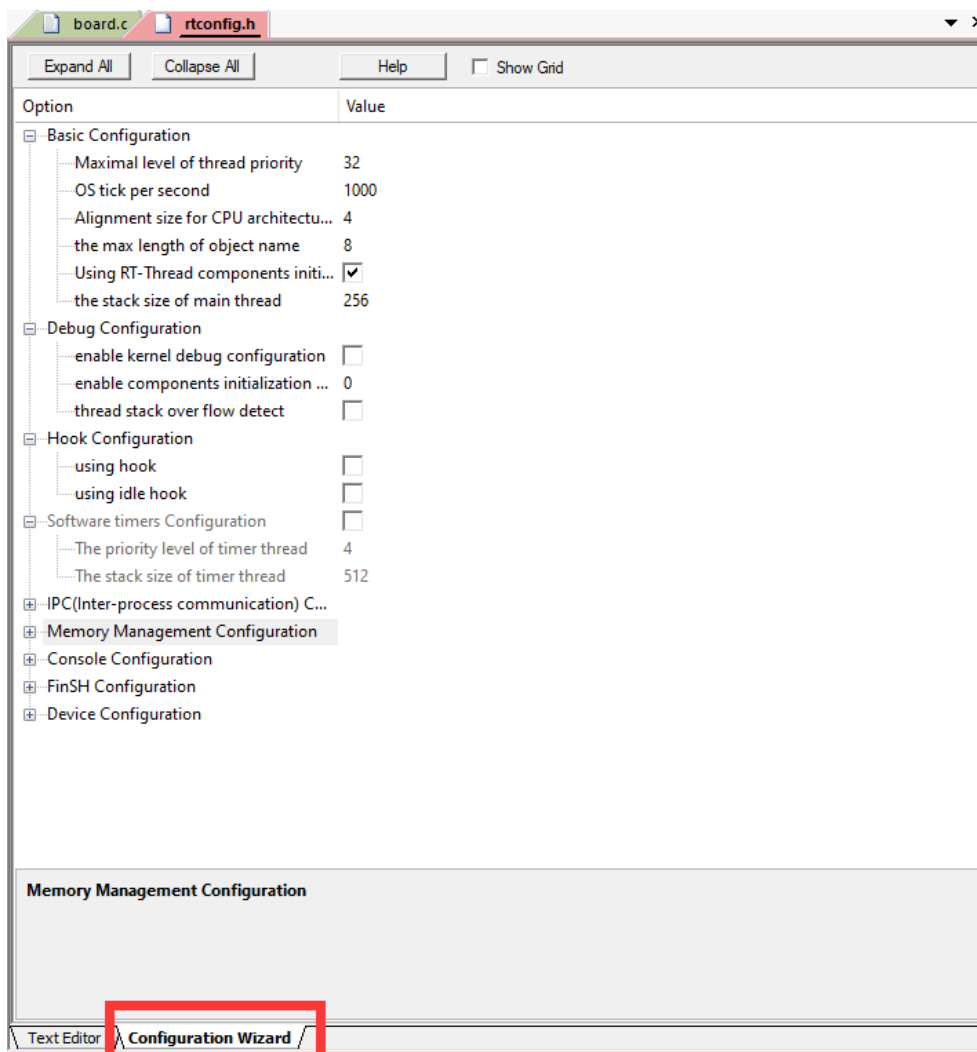
    while (1)
    {
        GPIO_SetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
        GPIO_ResetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
    }
}
```

2.6 配置 RT-Thread Nano

用户可以根据自己的需要通过修改 `rtconfig.h` 文件里面的宏定义配置相应功能。

MDK 的配置向导 `configuration Wizard` 可以很方便的对工程进行配置，`Value` 一栏可以选中对应功能及修改相关值，等同于直接修改配置文件 `rtconfig.h`。

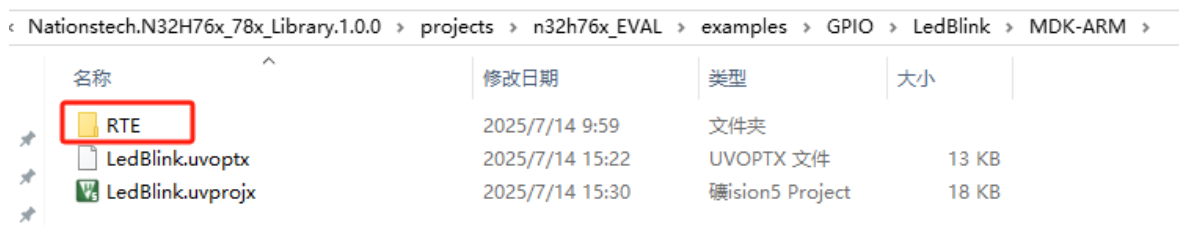
图 2-12 rtconfig.h 配置界面



2.7 注意事项

切勿删除 MDK_ARM 文件夹下的 RTE 文件夹，里面包含 board.c 和 rtconfig.h 文件，若删除需要根据前面描述重新编写这两个文件。

图 2-13 MDK_ARM 文件夹



3 基于 IAR 移植 RT-Thread Nano

基于 IAR 移植 RT-Thread Nano 的主要步骤:

1. 准备一个基础的 IAR 工程, 并获取 RT-Thread Nano 压缩包源码
2. 在基础工程中添加 RT-Thread Nano 源码, 添加相应头文件路径
3. 适配 Nano, 主要从 中断、时钟、内存、应用 这几个方面进行适配, 实现移植
4. 最后可对 Nano 进行配置: Nano 是可裁剪的, 通过配置文件 rtconfig.h 实现对系统的裁剪

3.1 基础工程准备

图 3-1 IAR 基础工程准备

```

/**
 * \name Delay.
 * \fun Delay program.
 * \param none
 * \return none
 */
static void Delay(void)
{
    int i = 0x1000000;
    while (i--);
}

/**
 * \name main.
 * \fun Main program.
 * \param none
 * \return none
 */
int main(void)
{
    /* Initialize system clock */
    RCC_SetSysClkToMode0();
    /* RCC configuration ----- */
    RCC_Configuration();
    /* GPIO configuration ----- */
    GPIO_Configuration();

    while (1)
    {
        GPIO_SetBits(LED1_PORT, LED1_PIN);
        Delay();
        GPIO_ResetBits(LED1_PORT, LED1_PIN);
        Delay();
    }
}

```

3.2 下载 RT-Thread Nano 源码

访问 <https://www.rt-thread.org/download.html#download-rt-thread-nano>, 下载 RT-Thread Nano 源码, 下载完成后解码如下图所示:

图 3-2 Nano 源码

名称	修改日期	类型	大小
bsp	2024/9/20 16:19	文件夹	
components	2024/9/20 16:19	文件夹	
docs	2024/9/20 16:19	文件夹	
include	2024/9/20 16:19	文件夹	
libcpu	2024/9/20 16:19	文件夹	
src	2024/9/20 16:19	文件夹	
.gitattributes	2024/9/20 16:19	GITATTRIBUTES ...	1 KB
.gitignore	2024/9/20 16:19	GITIGNORE 文件	1 KB
.travis.yml	2024/9/20 16:19	YML 文件	8 KB
AUTHORS	2024/9/20 16:19	文件	1 KB
ChangeLog.md	2024/9/20 16:19	MD 文件	134 KB
LICENSE	2024/9/20 16:19	文件	12 KB
README.md	2024/9/20 16:19	MD 文件	5 KB
README_zh.md	2024/9/20 16:19	MD 文件	6 KB

3.3 添加 RT-Thread Nano 到工程

3.3.1 添加 Nano 源文件

在准备好的 IAR 裸机工程下面新建 rtthread 文件夹，并在该文件中添加以下文件：

- Nano 源码中的 include、libcpu、src 文件夹
- 配置文件：源代码 bsp 文件夹中的两个文件：board.c 与 rtconfig.h

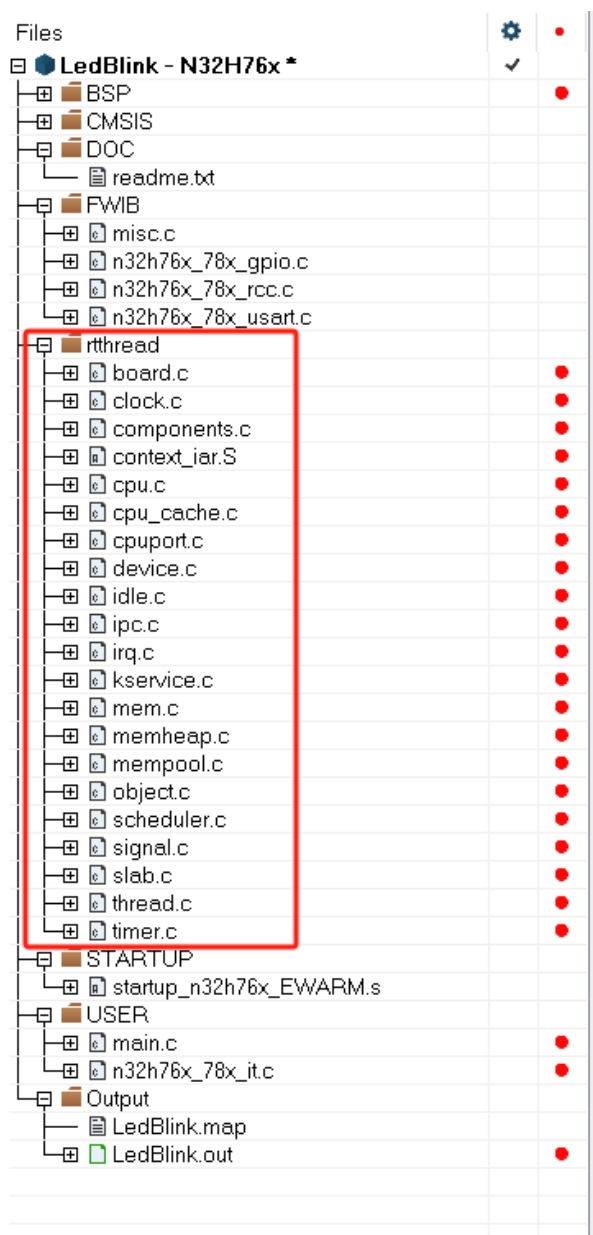
图 3-3 rtthread 文件夹

名称	修改日期	类型	大小
include	2025/6/25 15:22	文件夹	
libcpu	2025/6/25 15:22	文件夹	
src	2025/6/25 15:22	文件夹	
board.c	2024/9/20 16:19	C 文件	2 KB
rtconfig.h	2024/9/20 16:19	H 文件	4 KB

双击打开 IAR 裸机工程，新建 rtthread 分组，并在该分组下添加以下源码：

- 添加工程下 rtthread/src/ 文件夹中所有文件到工程
- 添加工程下 rtthread/libcpu/ 文件夹中相应内核的 CPU 移植文件及上下文切换文件：cpuport.c 以及 context_iar.S
- 添加 rtthread/ 文件夹下的 board.c

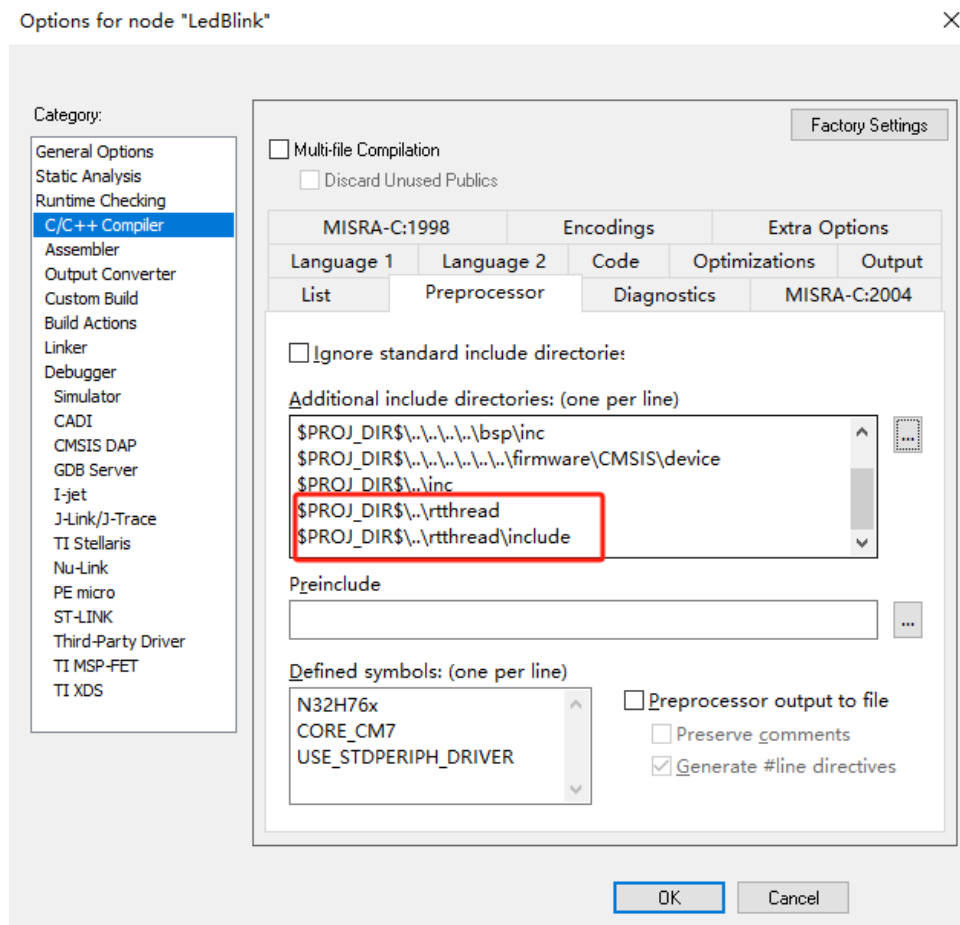
图 3-4 rtthread 分组



3.3.2 添加头文件路径

点击 Project -> Options... 进入下图所示界面，添加 rtconfig.h 头文件所在位置的路径，添加 include 文件夹下的头文件路径。

图 3-5 头文件路径添加



3.4 适配 RT-Thread Nano

3.4.1 中断与异常处理

RT-Thread 会接管异常处理函数 `HardFault_Handler()` 和悬挂处理函数 `PendSV_Handler()`，这两个函数已由 RT-Thread 实现，所以需要删除工程里 `n32h76x_78x_it.c` 文件中的 `HardFault_Handler()` 和 `PendSV_Handler()` 这两个函数，避免在编译时产生重复定义。如果此时对工程进行编译，没有出现函数重复定义的错误，则不用做修改。

3.4.2 系统时钟配置

需要在 `board.c` 中实现 系统时钟配置（为 MCU、外设提供工作时钟）与 `os tick` 的配置（为操作系统提供心跳 / 节拍）。

如下代码所示，需要在 `board.c` 文件中系统初始化和 OS Tick 的配置，cortex-M 架构需在 `SysTick_Handler()` 函数调用 `rt_os_tick_callback()` 函数。

图 3-6 board.c 中系统时钟与 os tick 配置

```

void rt_os_tick_callback(void)
{
    rt_interrupt_enter();

    rt_tick_increase();

    rt_interrupt_leave();
}

/**
 * \name SysTick_Handler.
 * \fun This function handles SysTick exception.
 * \param none
 * \return none
 */
void SysTick_Handler(void)
{
    rt_os_tick_callback();
}

/**
 * This function will initial your board.
 */
void rt_hw_board_init(void)
{
    RCC_ClocksTypeDef RCC_Clocks = { 0 };
    /*
     * TODO 1: OS Tick Configuration
     * Enable the hardware timer and call the rt_os_tick_callback function
     * periodically with the frequency RT_TICK_PER_SECOND.
     */

    /* 1: Initialize system clock */
    RCC_SetSysClkToMode0();
    /* frequencies of different on chip clocks */
    RCC_GetClocksFreqValue(&RCC_Clocks);

    /* 2: OS Tick update */
    SysTick_Config(RCC_Clocks.M7ClkFreq / RT_TICK_PER_SECOND);

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
    rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif
}

```

3.4.3 CONSOLE 串口初始化

在 rtconfig.h 中打开 RT_USING_CONSOLE 使能控制台功能，然后在 board.c 中的 uart_init() 函数实现串口初始化和 rt_hw_console_output() 函数实现控制台输出重定向。

图 3-7 board.c 中控制台串口初始化

```
static int uart_init(void)
{
    /* TODO 2: Enable the hardware uart and config baudrate */
    GPIO_InitType GPIO_InitStructure;
    USART_InitType USART_InitStructure;

    DEBUG_USART_GPIO_APBxClkCmd( DEBUG_USART_GPIO_CLK, ENABLE);

    DEBUG_USART_APBxClkCmd(DEBUG_USART_CLK, ENABLE);

    GPIO_InitStruct(&GPIO_InitStructure);
    GPIO_InitStructure.Pin = DEBUG_USART_TX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.GPIO_Pull = GPIO_NO_PULL;
    GPIO_InitStructure.GPIO_Alternate = DEBUG_USART_TX_GPIO_AF;
    GPIO_InitPeripheral( DEBUG_USART_TX_GPIO_PORT, &GPIO_InitStructure );

    GPIO_InitStructure.Pin = DEBUG_USART_RX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.GPIO_Pull = GPIO_NO_PULL;
    GPIO_InitStructure.GPIO_Alternate = DEBUG_USART_RX_GPIO_AF;
    GPIO_InitPeripheral( DEBUG_USART_RX_GPIO_PORT, &GPIO_InitStructure );

    USART_StructInit(&USART_InitStructure);
    USART_InitStructure.BaudRate = DEBUG_USART_BAUDRATE;
    USART_InitStructure.WordLength = USART_WL_8B;
    USART_InitStructure.StopBits = USART_STPB_1;
    USART_InitStructure.Parity = USART_PE_NO ;
    USART_InitStructure.HardwareFlowControl = USART_HFCTRL_NONE;
    USART_InitStructure.Mode = USART_MODE_RX | USART_MODE_TX;
    USART_Init( LOG_USARTx, &USART_InitStructure );

    USART_Enable( LOG_USARTx, ENABLE );
    return 0;
}
INIT_BOARD_EXPORT(uart_init);
```

图 3-8 board.c 中控制台输出重定向

```
void rt_hw_console_output(const char *str)
{
    /* TODO 3: Output the string 'str' through the uart */
    uint8_t char_r = '\r';
    /* Enter the critical */
    rt_enter_critical();

    /* Until the end of the string */
    while (*str != '\0')
    {
        if (*str == '\n')
        {
            /* Loop until the end of transmission */
            while (USART_GetFlagStatus(LOG_USARTx, USART_FLAG_TXC) == RESET)
            {
            }
            USART_SendData(LOG_USARTx, char_r);
        }

        /* Loop until the end of transmission */
        while (USART_GetFlagStatus(LOG_USARTx, USART_FLAG_TXC) == RESET)
        {
        }
        USART_SendData(LOG_USARTx, *(uint8_t *)str);

        str++;
    }

    /* Exit the critical */
    rt_exit_critical();
}
```

3.4.4 内存堆初始化

系统内存堆的初始化在 board.c 中的 rt_hw_board_init() 函数中完成, 内存堆功能是否使用取决于 rtconfig.h 中宏 RT_USING_HEAP 是否开启, RT-Thread Nano 默认不开启内存堆功能, 这样可以保持一个较小的体积, 不用为内存堆开辟空间。

开启系统 heap 将可以使用动态内存功能, 如使用 rt_malloc、rt_free 以及各种系统动态创建对象的 API。若需要使用系统内存堆功能, 则打开 rtconfig.h 中 RT_USING_HEAP 宏定义即可, 此时内存堆初始化函数 rt_system_heap_init() 将被调用。

图 3-9 rt_system_heap_init()调用

```
/**
 * This function will initial your board.
 */
void rt_hw_board_init(void)
{
    RCC_ClocksTypeDef RCC_Clocks = { 0 };
    /*
     * TODO 1: OS Tick Configuration
     * Enable the hardware timer and call the rt_os_tick_callback function
     * periodically with the frequency RT_TICK_PER_SECOND.
     */

    /* 1: Initialize system clock */
    RCC_SetSysClkToMode0();
    /* frequencies of different on chip clocks */
    RCC_GetClocksFreqValue(&RCC_Clocks);

    /* 2: OS Tick update */
    SysTick_Config(RCC_Clocks.M7ClkFreq / RT_TICK_PER_SECOND);

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

    #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
        rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
    #endif
}
```

初始化内存堆需要堆的起始地址与结束地址这两个参数, 系统中默认使用数组作为 heap, 并获取了 heap 的起始地址与结束地址, 该数组大小可手动更改, 如下所示:

图 3-10 数组作为 heap 大小配置

```

14 #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
15 /*
16  * Please modify RT_HEAP_SIZE if you enable RT_USING_HEAP
17  * the RT_HEAP_SIZE max value = (sram size - ZI size), 1024 means 1024 bytes
18  */
19 #define RT_HEAP_SIZE (15*1024)
20 static rt_uint8_t rt_heap[RT_HEAP_SIZE];
21
22 RT_WEAK void *rt_heap_begin_get(void)
23 {
24     return rt_heap;
25 }
26
27 RT_WEAK void *rt_heap_end_get(void)
28 {
29     return rt_heap + RT_HEAP_SIZE;
30 }
31 #endif

```

注意：开启 heap 动态内存功能后，heap 默认值较小，在使用的时候需要改大，否则可能会有申请内存失败或者创建线程失败的情况，修改方法有以下两种：

- 方法一：可以直接修改数组中定义的 RT_HEAP_SIZE 的大小
- 方法二：使用 RAM ZI 段结尾处作为 HEAP 的起始地址，使用 RAM 的结尾地址作为 HEAP 的结尾地址

注意：移植过程中请注意 kservice.c 中的关于动态内存分配配置，新版的 RT-Thread Nano 内核对于 Memory Pool、Dynamic Heap、Tiny Memory 的管理方式有所不同，除了在 rtconfig.h 中打开相应宏之外，还需要定义对应内存管理的宏。

图 3-11 rtconfig.h 中内存管理配置

```

// <h>Memory Management Configuration
// <cl>Memory Pool Management
// <i>Memory Pool Management
// #define RT_USING_MEMPOOL
// </c>
// <cl>Dynamic Heap Management(Algorithm: small memory )
// <i>Dynamic Heap Management
#define RT_USING_HEAP
#define RT_USING_SMALL_MEM
#define RT_USING_SMALL_MEM_AS_HEAP
// </c>
// <cl>using tiny size of memory
// <i>using tiny size of memory
// #define RT_USING_TINY_SIZE
// </c>
// </h>

```

图 3-12 kservice.c 中内存管理选择宏

```
#if defined(RT_USING_SMALL_MEM_AS_HEAP)
static rt_smem_t system_heap;
rt_inline void _smem_info(rt_size_t *total,
    rt_size_t *used, rt_size_t *max_used)
{
    if (total)
        *total = system_heap->total;
    if (used)
        *used = system_heap->used;
    if (max_used)
        *max_used = system_heap->max;
}
#define _MEM_INIT(_name, _start, _size) \
    system_heap = rt_smem_init(_name, _start, _size)
#define _MEM_MALLOC(_size) \
    rt_smem_alloc(system_heap, _size)
#define _MEM_REALLOC(_ptr, _newsize) \
    rt_smem_realloc(system_heap, _ptr, _newsize)
#define _MEM_FREE(_ptr) \
    rt_smem_free(_ptr)
#define _MEM_INFO(_total, _used, _max) \
    _smem_info(_total, _used, _max)
#elif defined(RT_USING_MEMHEAP_AS_HEAP)
static struct rt_memheap system_heap;
void *_memheap_alloc(struct rt_memheap *heap, rt_size_t size);
void _memheap_free(void *rmem);
void *_memheap_realloc(struct rt_memheap *heap, void *rmem, rt_size_t newsize);
#define _MEM_INIT(_name, _start, _size) \
    rt_memheap_init(&system_heap, _name, _start, _size)
#define _MEM_MALLOC(_size) \
    _memheap_alloc(&system_heap, _size)
#define _MEM_REALLOC(_ptr, _newsize) \
    _memheap_realloc(&system_heap, _ptr, _newsize)
#define _MEM_FREE(_ptr) \
    _memheap_free(_ptr)
#define _MEM_INFO(_total, _used, _max) \
    rt_memheap_info(&system_heap, _total, _used, _max)
#elif defined(RT_USING_SLAB_AS_HEAP)
static rt_slab_t system_heap;
rt_inline void _slab_info(rt_size_t *total,
    rt_size_t *used, rt_size_t *max_used)
{
    if (total)
        *total = system_heap->total;
    if (used)
        *used = system_heap->used;
    if (max_used)
        *max_used = system_heap->max;
}
#define _MEM_INIT(_name, _start, _size) \
    system_heap = rt_slab_init(_name, _start, _size)
#define _MEM_MALLOC(_size) \
    rt_slab_alloc(system_heap, _size)
#define _MEM_REALLOC(_ptr, _newsize) \
    rt_slab_realloc(system_heap, _ptr, _newsize)
#define _MEM_FREE(_ptr) \
    rt_slab_free(system_heap, _ptr)
#define _MEM_INFO _slab_info
#else
#define _MEM_INIT(...)
#define _MEM_MALLOC(...) RT_NULL
#define _MEM_REALLOC(...) RT_NULL
#define _MEM_FREE(...)
#define _MEM_INFO(...)
#endif
```

3.5 编写第一个应用

移植好 RT-Thread Nano 之后，则可以开始编写第一个应用代码验证移植结果。此时 `main()` 函数就转变成 RT-Thread 操作系统的一个线程，现在可以在 `main()` 函数中实现第一个应用：板载 LED 指示灯闪烁，这里直接基于裸机 LED 指示灯进行修改。

1. 首先在文件首部增加 RT-Thread 的相关头文件 `<rtthread.h>`
2. 在 `main()` 函数中（也就是在 `main` 线程中）实现 LED 闪烁代码
3. 将延时函数替换为 RT-Thread 提供的延时函数 `rt_thread_mdelay()`。该函数会引起系统调度，切换到其他线程运行，体现了线程实时性的特点

图 3-13 LED 闪烁应用示例

```
#include "main.h"
#include "n32h76x_78x_it.h"
#include <rtthread.h>

void GPIO_Configuration(void);
void RCC_Configuration(void);

/**
 * \name    main.
 * \fun     Main program.
 * \param   none
 * \return  none
 */
int main(void)
{
    /* RCC configuration ----- */
    RCC_Configuration();
    /* GPIO configuration ----- */
    GPIO_Configuration();

    while (1)
    {
        GPIO_SetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
        GPIO_ResetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
    }
}
```

3.6 配置 RT-Thread Nano

用户可以根据自己的需要通过打开或关闭 `rtconfig.h` 文件里面的宏定义，配置相应功能，如下是 `rtconfig.h` 的代码片段：

图 3-14 rtconfig.h 的代码片段

```
// <e>Software timers Configuration
// <i> Enables user timers
#define RT_USING_TIMER_SOFT      0
#if RT_USING_TIMER_SOFT == 0
    #undef RT_USING_TIMER_SOFT
#endif
// <o>The priority level of timer thread <0-31>
// <i>Default: 4
#define RT_TIMER_THREAD_PRIO      4
// <o>The stack size of timer thread <0-8192>
// <i>Default: 512
#define RT_TIMER_THREAD_STACK_SIZE 512
// </e>

// <h>IPC(Inter-process communication) Configuration
// <cl>Using Semaphore
// <i>Using Semaphore
#define RT_USING_SEMAPHORE
// </c>
// <cl>Using Mutex
// <i>Using Mutex
// #define RT_USING_MUTEX
// </c>
// <cl>Using Event
// <i>Using Event
// #define RT_USING_EVENT
// </c>
// <cl>Using MailBox
// <i>Using MailBox
// #define RT_USING_SIGNALS
// </c>
// <cl>Using Signals
// <i>Using Signals
#define RT_USING_MAILBOX
// </c>
// <cl>Using Message Queue
// <i>Using Message Queue
// #define RT_USING_MESSAGEQUEUE
// </c>
// </h>
```

4 版本历史

日期	版本	修改点
2025-06-25	V0.0.1	初始版本

5 声明

国民技术股份有限公司（下称“国民技术”）对此文档拥有专属产权。依据中华人民共和国的法律、条约以及世界其他法域相适用的管辖，此文档及其中描述的国民技术产品（下称“产品”）为公司所有。

国民技术在此并未授予专利权、著作权、商标权或其他任何知识产权许可。所提到或引用的第三方名称或品牌（如有）仅用作区别之目的。

国民技术保留随时变更、订正、增强、修改和改良此文档的权利，恕不另行通知。请使用者在下单购买前联系国民技术获取此文档的最新版本。

国民技术竭力提供准确可信的资讯，但即便如此，并不推定国民技术对此文档准确性和可靠性承担责任。

使用此文档信息以及生成产品时，使用者应当进行合理的设计、编程并测试其功能性和安全性，国民技术不对任何因使用此文档或本产品而产生的任何直接、间接、意外、特殊、惩罚性或衍生性损害结果承担责任。

国民技术对于产品在系统或设备中的应用效果没有任何故意或保证，如有任何应用在其发生操作不当或故障情况下，有可能致使人员伤亡、人身伤害或严重财产损失，则此类应用被视为“不安全使用”。

不安全使用包括但不限于：外科手术设备、原子能控制仪器、飞机或宇宙飞船仪器、所有类型的安全装置以及其他旨在支持或维持生命的应用。

所有不安全使用的风险应由使用人承担，同时使用人应使国民技术免于因为这类不安全使用而导致被诉、支付费用、发生损害或承担责任时的赔偿。

对于此文档和产品的任何明示、默示之保证，包括但不限于适销性、特定用途适用性和不侵权的保证责任，国民技术可在法律允许范围内进行免责。

未经明确许可，任何人不得以任何理由对此文档的全部或部分进行使用、复制、修改、抄录和传播。