

Application note

N32H7xx Series RT-Thread Nano Porting Application Notes

Introduction

The purpose of this document is to enable users to quickly become familiar with porting RT-Thread Nano to the N32H7xx series microcontrollers (MCUs).

Contents

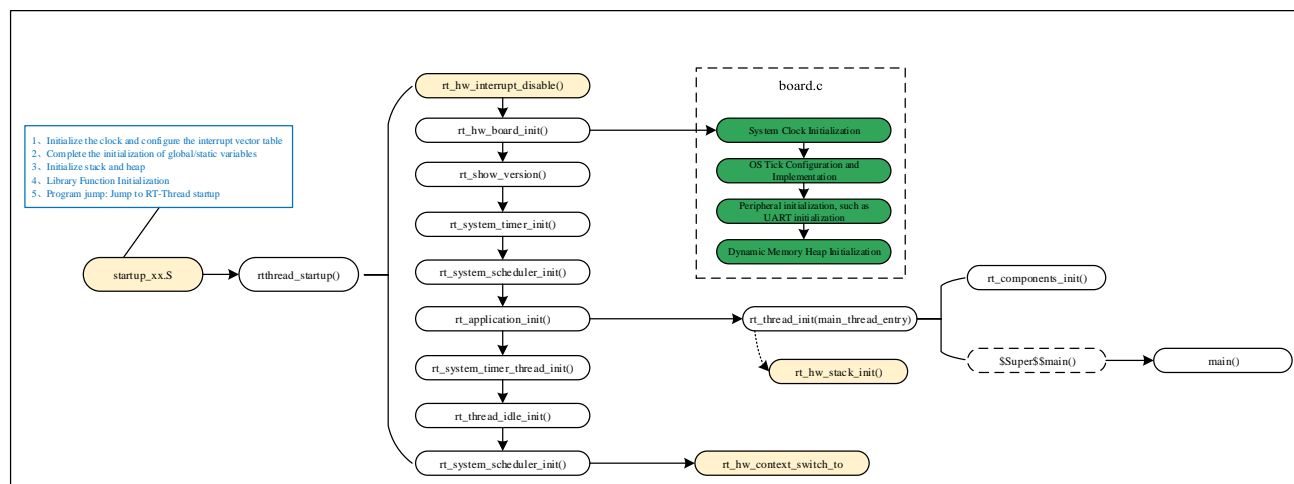
Contents.....	1
1 RT-Thread Nano Porting Principles.....	2
1.1 Startup Process	2
1.2 libcpu Porting	3
1.2.1 Startup File startup.s.....	3
1.2.2 Context Switching context_xx.S	3
1.2.3 Thread Stack Initialization cpuport.c.....	4
1.2.4 Interrupts and Exceptions Hooking interrupt.c	4
1.3 Board-Level Porting board.c	4
1.3.1 Configuring the System Clock	5
1.3.2 Implementing the OS Tick.....	5
1.3.3 Hardware Peripheral Initialization.....	6
1.3.4 Implementing a Dynamic Memory Heap	6
2 Porting RT-Thread Nano Based on Keil MDK	8
2.1 Basic Project Preparation.....	8
2.2 Nano Pack Installation.....	8
2.3 Adding RT-Thread Nano to the Project.....	9
2.4 Adapting RT-Thread Nano	11
2.4.1 Interrupts and Exception Handling	11
2.4.2 System Clock Configuration	11
2.4.3 CONSOLE Serial Port Initialization	12
2.4.4 Memory Heap Initialization.....	14
2.5 Writing First Application	15
2.6 Configuring RT-Thread Nano	16
2.7 Precautions	17
3 Porting RT-Thread Nano Based on IAR.....	18
3.1 Basic Project Preparation.....	18
3.2 Nano Pack Installation.....	18
3.3 Adding RT-Thread Nano to the Project.....	19
3.3.1 Add Nano Source File	19
3.3.2 Add Header File Path.....	20
3.4 Adapting RT-Thread Nano.....	21
3.4.1 Interrupts and Exception Handling	21
3.4.2 System Clock Configuration	21
3.4.3 CONSOLE Serial Port Initialization	22
3.4.4 Memory Heap Initialization.....	24
3.5 Writing First Application	27
3.6 Configuring RT-Thread Nano	27
4 Version history	29
5 Notice	30

1 RT-Thread Nano Porting Principles

1.1 Startup Process

The RT-Thread startup process is shown below, with the colored parts in the diagram requiring special attention from the user (yellow indicates content related to libcpu porting, and green indicates content related to board-level porting).

Figure 1-1 RT-Thread Startup Process



The unified entry point for the RT-Thread startup code is `rthread_startup()`. After the chip startup file completes the necessary tasks (such as initializing the clock, configuring the interrupt vector table, initializing the stack, etc.), the program will ultimately jump to the RT-Thread startup entry during execution. The startup process of RT-Thread is as follows:

1. Disable global interrupts and initialize system-related hardware.
2. Print system version information and initialize system kernel objects (such as timers and schedulers).
3. Initialize the user main thread (which also initializes the thread stack), and sequentially initialize various modules within the main thread.
4. Initialize the software timer thread and the idle thread.
5. Start the scheduler, have the system switch to the first thread to start running (such as the main thread), and enable global interrupts.

Figure 1-2 RT-Thread Nano Source Code Directory

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

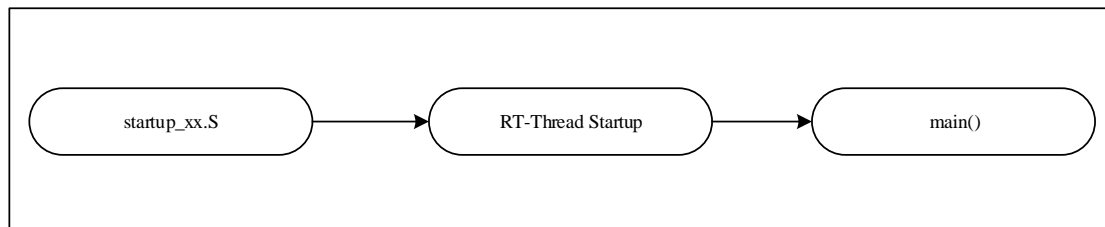
1.2 libcpu Porting

The libcpu abstraction layer of RT-Thread provides a unified CPU architecture porting interface downward. This part of the interface includes global interrupt switch functions, thread context switch functions, clock tick configuration and interrupt functions, cache, and so on. The CPU architectures supported by RT-Thread can be found in the libcpu folder in the source code.

1.2.1 Startup File startup.s

Each chip has a corresponding startup file, and the startup files vary in different development environments. The startup files are located in the chip firmware library. After the system integrates RT-Thread, the startup of RT-Thread will be executed before calling the main() function, as shown in the figure below:

Figure 1-3 RT-Thread startup call



- startup.s: mainly completes the initialization of the clock, configures the interrupt vector table; completes the initialization of global/static variables; initializes the stack; initializes library functions; and handles program jumps.
- Program jump: In KEIL MDK and IAR, the chip's startup file does not need to be modified and will automatically jump to the RT-Thread system startup function rththread_startup().

1.2.2 Context Switching context_xx.S

Context switching refers to the CPU switching from one thread to another, or between a thread and an interrupt, and so on. During a context switch, the CPU generally stops executing the current code and saves the specific location of the current program so that it can resume execution later.

Table 1-1 context_xx.S implementation function

Function to be implemented	Description
rt_base_t rt_hw_interrupt_disable(void)	Disable global interrupts
void rt_hw_interrupt_enable(rt_base_t level)	Enable global interrupts
void rt_hw_context_switch_to(rt_uint32 to)	Context switches without a source thread are called when the scheduler starts the first thread, and they are also called inside signal handlers
void rt_hw_context_switch(rt_uint32 from, rt_uint32 to)	Switching from thread to another thread, used for switching between threads
void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to)	Switching from thread to another thread, used for switching inside interrupt

Note: in Cortex-M, the PendSV interrupt handler is PendSV_Handler(), and the actual work of thread switching is

done within `PendSV_Handler()`.

1.2.3 Thread Stack Initialization `cpuport.c`

In RT-Thread, threads have independent stacks. During a thread switch, the current thread's context is stored in the stack, and when the thread is to resume execution, the context information is read from the stack to restore it.

The fault exception handling function `rt_hw_hard_fault_exception()` is executed in the `HardFault_Handler` interrupt when a hardware error occurs.

This file mainly implements the thread stack initialization function `rt_hw_stack_init()` and the hard fault exception handling function.

Table 1-2 `cpuport.c` implementation function

Function to be implemented	Description
<code>rt_hw_stack_init()</code>	Initialize thread stack
<code>rt_hw_hard_fault_exception()</code>	Exception function: System hardware error

1.2.4 Interrupts and Exceptions Hooking `interrupt.c`

On the Cortex-M core, all interrupts are handled using an interrupt vector table. When an interrupt is triggered, the processor directly determines which interrupt source it is and then jumps to the corresponding fixed location for handling, without the need to implement interrupt management manually.

1.3 Board-Level Porting `board.c`

Note: `board.c` and `rtconfig.h` are hardware/board-specific files and need to be implemented individually during porting. For the Cortex M architecture, you can refer to the existing `board.c` and `rtconfig.h` files in the `bsp` folder of the Nano source code.

Board-level porting mainly focuses on the implementation of the `rt_hw_board_init()` function. This function is in the board-level configuration file `board.c` and performs many essential tasks for system startup, including:

1. Configure the system clock.
2. Implement the OS tick.
3. Initialize peripherals: such as GPIO/UART, etc. Call them here if needed.
4. Initialize the system memory heap to enable dynamic heap memory management.
5. Board-level automatic initialization. Functions set to auto-initialize using `INIT_BOARD_EXPORT()` will be initialized here.
6. Other necessary initializations, such as MMU configuration (if needed, please call the application functions yourself in the `rt_hw_board_init` function).

Figure 1-4 board.c execution flow

```
/* board.c */
Void rt_hw_board_init(void)
{
    /* Part 1: System Initialization, System Clock Configuration, etc. */
    /* 1: System layer initialization, add this part if needed */
    /* ..... */
    /* 2: System Clock Configuration */
    SystemClock_Config(); // Config System Clock
    SystemCoreClockUpdate(); // Update system clock frequency SystemCoreClock

    /* Part 2: Configure the OS Tick frequency to implement the OS tick (and implement OS Tick increment in the
    interrupt service routine) */
    _SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND);

    /* Part 3: Initialize hardware peripherals; if needed, call them here */
    /* ..... */

    /* Part 4: System Dynamic Memory Heap Initialization */
    #if defined(RT_USING_MAIN) && defined(RT_USING_HEAP)
        rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
    #endif

    /* Part 5: Initialization using INIT_BOARD_EXPORT() */
    #ifndef RT_USING_COMPONENTS_INIT
        rt_components_board_init();
    #endif

    /* Part 6: Other Initialization */
}
```

1.3.1 Configuring the System Clock

The system clock provides the basic working clock for various hardware modules. It is configured in the `rt_hw_board_init()` function, either by calling library functions or by implementing it yourself.

Figure 1-5 Example of configuring the system clock in board.c

```
/* board.c */
void rt_hw_board_init(void)
{
    /* Part 1: System Initialization, System Clock Configuration, etc. */
    /* Clock initialization, the function name is not required, the function should be implemented by yourself,
    such as SystemClock_Config() or SystemCoreClockUpdate() */
    rt_hw_clock_init();

    /* ..... */
}
```

1.3.2 Implementing the OS Tick

The OS tick is also called clock-tick or OS tick. Any operating system needs to provide a clock tick to handle all time-related events in the system.

Implementation of clock ticks: periodic interrupts are realized through a hardware timer, and the `rt_tick_increase()` function is called in the timer interrupt to increment the global variable `rt_tick`, thereby implementing clock ticks. Generally, on Cortex-M, the internal SysTick timer is used directly for this purpose.

Figure 1-6 OS Tick Example

```
/* board.c */
void rt_hw_board_init(void)
{
    /* Part 2: Configure the OS Tick frequency to implement the OS tick (and implement OS Tick increment in the
    interrupt service routine) */
    _SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND); // Using SysTick to implement clock ticks
}

/* SysTick interrupt handler */
void SysTick_Hnadler(void)
{
    /* Enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

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

Note: When initializing the clock tick, the macro `RT_TICK_PER_SECOND` will be used. By modifying the value of this macro, you can change the duration of one clock tick in the system.

1.3.3 Hardware Peripheral Initialization

Hardware initialization, such as UART initialization (for connecting to the console), needs to manually call the UART initialization function within the `rt_hw_board_init()` function.

Figure 1-7 OS Tick Example

```
/* board.c */
Void rt_hw_board_init(void)
{
    /* Part 3: Initialize hardware peripherals; if needed, call them here */
    uart_init();
}
```

Note: If `uart_init()` or other peripheral initialization functions have already been initialized using the macro `INIT_BOARD_EXPORT()`, there is no need to explicitly call them here. You should choose one of the two initialization methods.

1.3.4 Implementing a Dynamic Memory Heap

RT-Thread Nano does not enable the dynamic memory heap feature by default. Enabling `RT_USING_HEAP` allows the use of dynamic memory features, meaning you can use `rt_malloc`, `rt_free`, and various system APIs for dynamically creating objects. The initialization of the dynamic memory heap management function is done through the function `void rt_system_heap_init(void *begin_addr, void *end_addr)`, and initializing the dynamic memory heap requires specifying the start and end addresses of the heap memory.

After enabling `RT_USING_HEAP`, the system uses an array as the heap by default. The starting and ending addresses of the heap are passed as parameters to the heap initialization function. The heap initialization function `rt_system_heap_init()` will be called in `rt_hw_board_init()`. After enabling the heap, the system defaults to using an array as the heap (the default heap is relatively small; please increase it according to the chip's RAM when actually

using it). The obtained starting and ending addresses of the heap are passed as parameters to the heap initialization function:

Figure 1-8 Dynamic Memory Heap (Array) Initialization Example

```
#define RT_HEAP_SIZE 1025
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;
}

/* 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(rt_heap_begin_get(), rt_heap_end_get()); // The starting and ending addresses of the heap
    #endif
}
```

If you don't want to use an array as a dynamic memory heap, you can reassign the size of the system HEAP. For example, you can use the end of the RAM ZI segment as the starting address of the HEAP (here you need to check if it corresponds with the linker script), and use the end address of the RAM as the end address of the HEAP. This way, all the available RAM can be used as the dynamic memory heap. The following example redefines the starting and ending addresses of the HEAP and uses them as parameters to initialize the system HEAP.

Figure 1-9 Dynamic Memory Heap (Non-Array) Initialization Example

```
#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$$RW_IRAM1$Z$$Limit;
    #define HEAP_BEGIN    ((void *)&Image$$RW_IRAM1$Z$$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 Porting RT-Thread Nano Based on Keil MDK

RT-Thread Nano is already integrated into Keil MDK and can be downloaded and added directly within the IDE. The main steps to port Nano are:

1. Prepare a basic Keil MDK project, obtain the RT-Thread Nano package, and install it.
2. Add the RT-Thread Nano source code to the basic project.
3. Adapt Nano, mainly from the aspects of interrupts, clocks, and memory, to achieve the porting.
4. Verify the porting result: write the first application code, using RT-Thread Nano to blink an LED.
5. Finally, you can configure Nano: Nano is customizable, and the system can be tailored through the configuration file `rtconfig.h`.

2.1 Basic Project Preparation

Before porting RT-Thread Nano, it is necessary to prepare a bare-metal project that can run properly.

Figure 2-1 Bare-metal project example

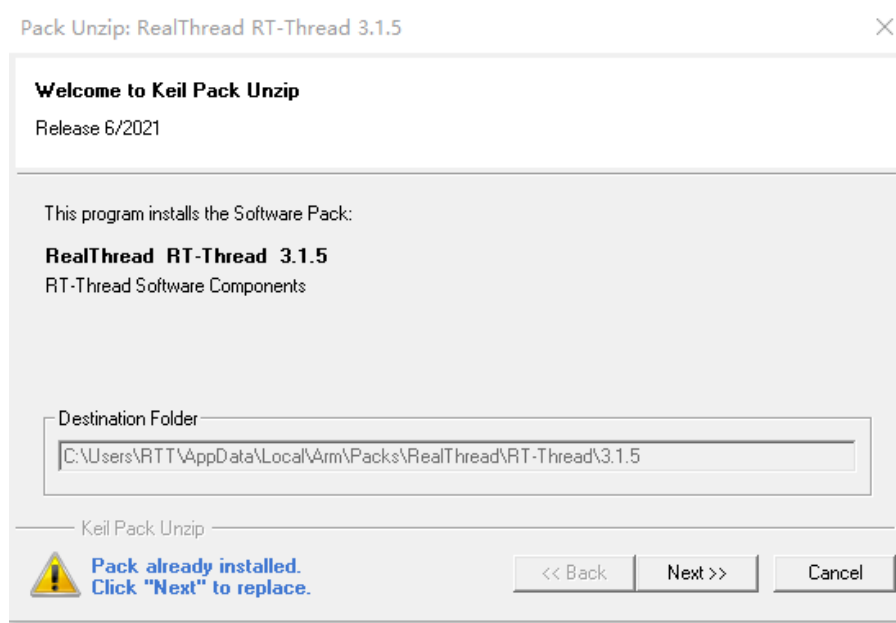
```
/**
 **\name    delay.
 **\fun     delay program.
 **\param   none
 **\return  none
 */
static void delay(void)
{
    int i = 0x1000000;
    while (i--);
}

/**
 **\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 Installation

Visit <https://www.keil.arm.com/packs/rt-thread-realthread/versions/> to download the Pack installation file. After the download is complete, double-click the file to install it:

Figure 2-2 Nano Pack Install


2.3 Adding RT-Thread Nano to the Project

Open the ready-to-run bare-metal program and add RT-Thread to the project. As shown in the figure below, click Manage Run-Time Environment.

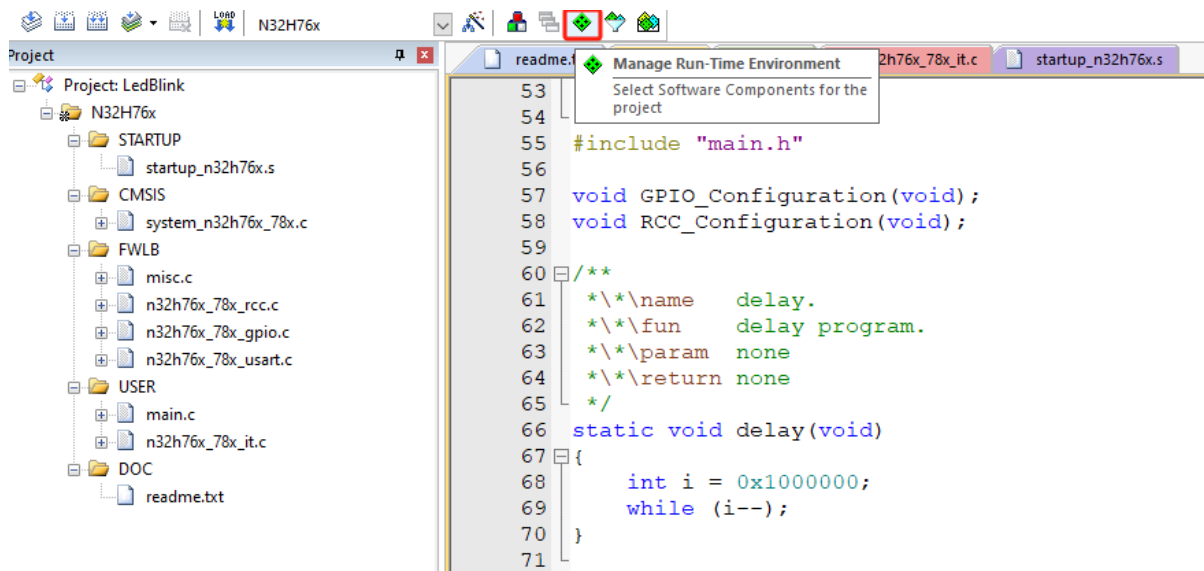
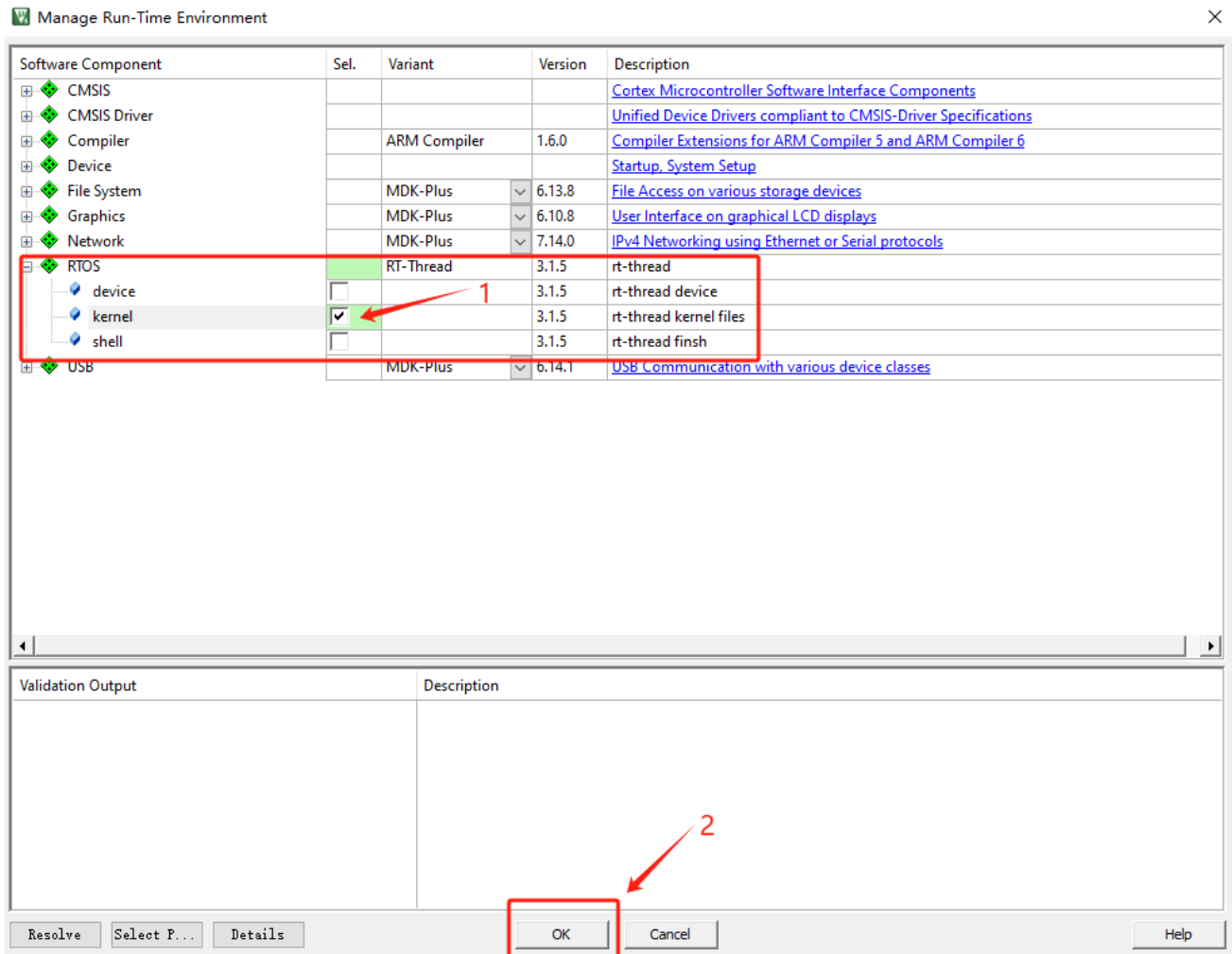
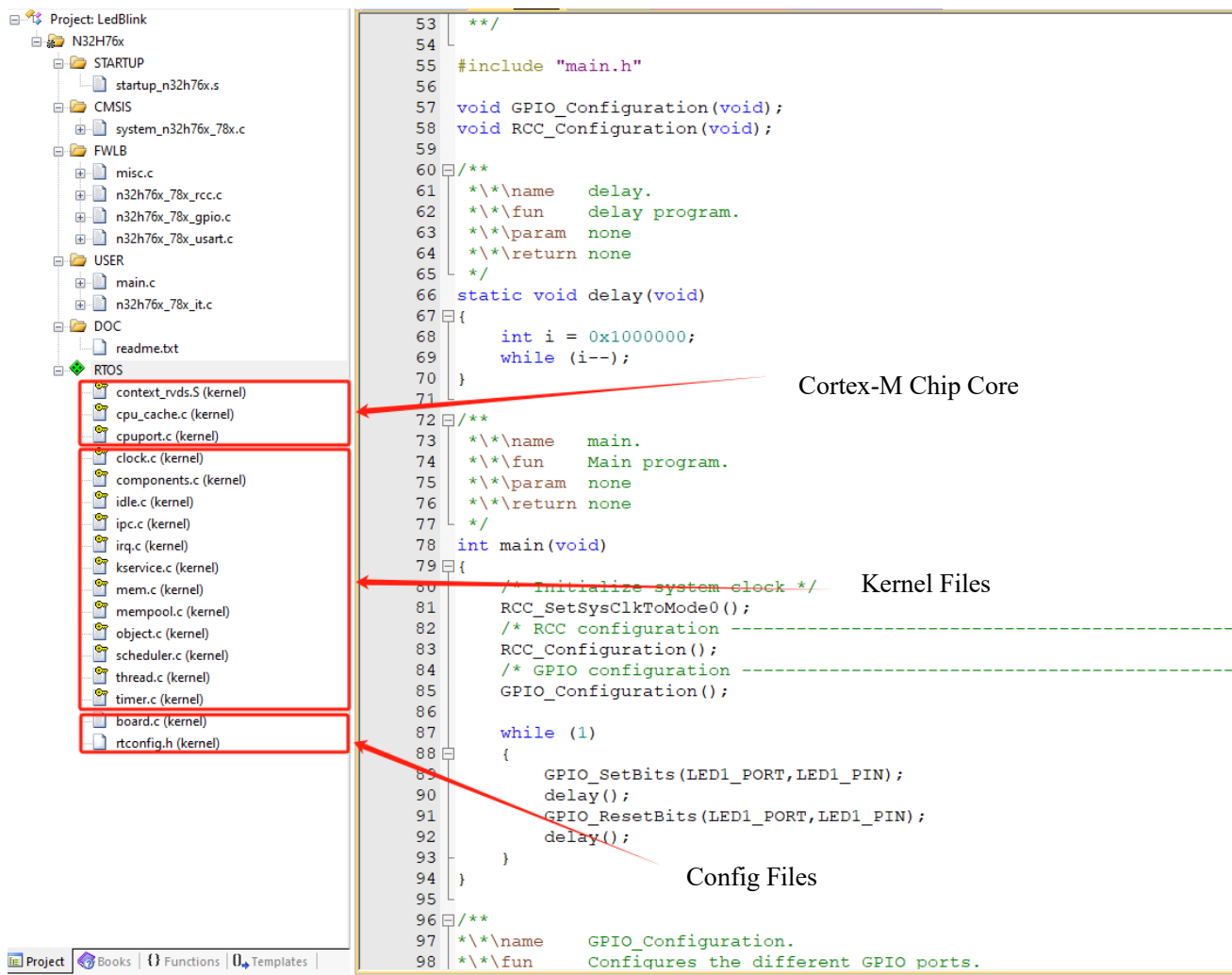
Figure 2-3 Step 1 of Adding Nano to the Project


Figure 2-4 Step 2 of Adding Nano to the Project


You can now see in Project that the RT-Thread RTOS has been added. Expanding RTOS, you can see the files that have been added to the project:

Figure 2-5 Project architecture after adding Nano


2.4 Adapting RT-Thread Nano

2.4.1 Interrupts and Exception Handling

RT-Thread will take over the exception handler `HardFault_Handler()` and the pendable service handler `PendSV_Handler()`. These two functions have already been implemented by RT-Thread, so you need to remove the `HardFault_Handler()` and `PendSV_Handler()` functions from the `n32h76x_78x_it.c` file in the project to avoid duplicate definitions during compilation. If the project compiles without any duplicate function definition errors, no modifications are needed.

2.4.2 System Clock Configuration

It is necessary to implement the system clock configuration in `board.c` (to provide working clocks for the MCU and peripherals) and the OS tick configuration (to provide a heartbeat/tick for the operating system).

As shown in the code below, system initialization and OS Tick configuration need to be done in the `board.c` file. For the Cortex-M architecture, the `rt_os_tick_callback()` function needs to be called in the `SysTick_Handler()` function.

Figure 2-6 System clock and OS tick configuration in board.c

```

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 Serial Port Initialization

Enable the console function by turning on RT_USING_CONSOLE in rtconfig.h, then implement the UART initialization in the uart_init() function in board.c and implement the console output redirection in the rt_hw_console_output() function.

Figure 2-7 Console serial port initialization in board.c

```
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);
```

Figure 2-8 Console output redirection in board.c

```
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 Memory Heap Initialization

The initialization of the system memory heap is completed in the `rt_hw_board_init()` function in `board.c`. Whether the memory heap feature is used depends on whether the macro `RT_USING_HEAP` in `rtconfig.h` is enabled. RT-Thread Nano does not enable the memory heap feature by default, which helps maintain a smaller footprint without allocating space for the memory heap.

Enabling the system heap allows the use of dynamic memory functions, such as `rt_malloc`, `rt_free`, and various system APIs for dynamically creating objects. If you need to use the system memory heap functionality, simply enable the `RT_USING_HEAP` macro in `rtconfig.h`, at which point the memory heap initialization function `rt_system_heap_init()` will be called.

Figure 2-9 Call to `rt_system_heap_init()`

```
/**
 * 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
}
```

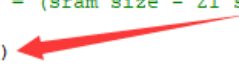
Initializing the memory heap requires two parameters: the starting address and the ending address of the heap. By default, the system uses an array as the heap and obtains the starting and ending addresses of the heap. The size of this array can be manually changed, as shown below:

Figure 2-10 Configuring array as heap size

```

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

```



Note: After enabling the heap dynamic memory feature, the default heap size is relatively small. You need to increase it when using it, otherwise memory allocation or thread creation may fail. There are two ways to modify it:

- *Method 1: You can directly modify the size of RT_HEAP_SIZE defined in the array.*
- *Method 2: Use the end of the RAM ZI section as the starting address of the HEAP, and use the end address of the RAM as the ending address of the HEAP.*

2.5 Writing First Application

After successfully porting RT-Thread Nano, you can start writing your first application code to verify the porting results. At this point, the main() function transforms into a thread of the RT-Thread operating system. You can now implement your first application in the main() function: flashing the onboard LED indicator. Here, it is directly modified based on the bare-metal LED indicator.

1. First, add the relevant RT-Thread header file <rtthread.h> at the beginning of the file.
2. Implement the LED blinking code in the main() function (that is, in the main thread).
3. Replace the delay function with the delay function provided by RT-Thread, rt_thread_mdelay(). This function will trigger system scheduling, switch to other threads, and reflect the real-time characteristics of threads.

Figure 2-11 LED Flasher Application Example

```
#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();

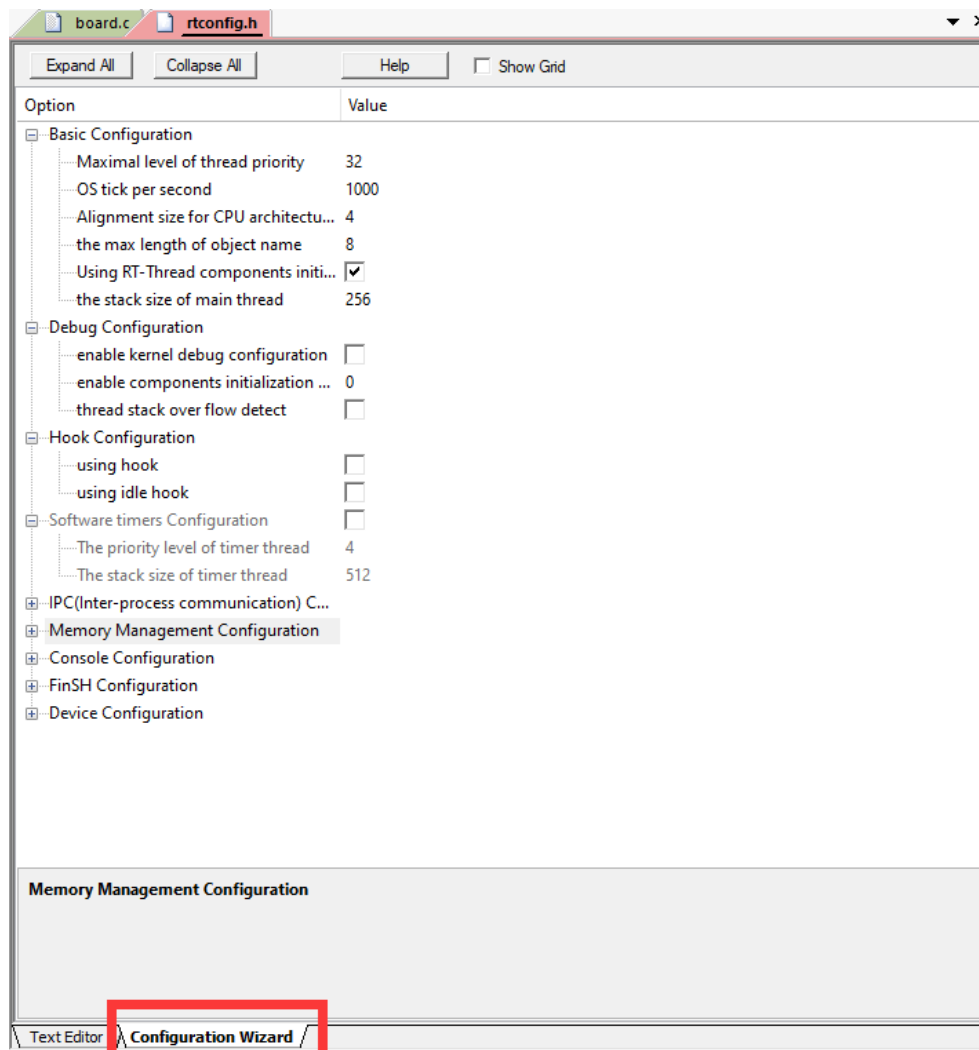
    while (1)
    {
        GPIO_SetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
        GPIO_ResetBits(LED1_PORT, LED1_PIN);
        rt_thread_mdelay(500);
    }
}
```

2.6 Configuring RT-Thread Nano

Users can configure the corresponding features according to their needs by modifying the macro definitions in the `rtconfig.h` file.

The MDK configuration wizard makes it easy to configure the project. In the 'Value' column, you can select the corresponding function and modify related values, which is equivalent to directly editing the configuration file `rtconfig.h`.

Figure 2-12 rtconfig.h configuration interface



2.7 Precautions

Do not delete the RTE folder under the MDK_ARM folder, as it contains the board.c and rtconfig.h files. If deleted, you will need to rewrite these two files according to the previous instructions.

Figure 2-13 MDK_ARM folder

Nationtech.N32H76x_78x_Library.1.0.0 > projects > n32h76x_EVAL > examples > GPIO > LedBlink > MDK-ARM >				
名称	修改日期	类型	大小	
RTE	2025/7/14 9:59	文件夹		
LedBlink.uvoptx	2025/7/14 15:22	UVOPTX 文件	13 KB	
LedBlink.uvprojx	2025/7/14 15:30	磳ision5 Project	18 KB	

3 Porting RT-Thread Nano Based on IAR

The main steps for porting RT-Thread Nano based on IAR:

1. Prepare a basic IAR project and obtain the RT-Thread Nano source code compressed package.
2. Add the RT-Thread Nano source code to the basic project and include the corresponding header file paths.
3. Adapt Nano, mainly focusing on interrupts, clocks, memory, and applications to achieve the porting.
4. Finally, configure Nano: Nano is modular, and the system can be tailored through the configuration file rtconfig.h.

3.1 Basic Project Preparation

Figure 3-1 IAR Basic Project Preparation

```

/**
 * \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 Nano Pack Installation

Visit <https://www.keil.arm.com/packs/rt-thread-realthread/versions/>, download the RT-Thread Nano source code. After the download is complete, decode it as shown in the figure below:

Figure 3-2 RT-Thread Nano source code

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 Adding RT-Thread Nano to the Project

3.3.1 Add Nano Source File

Create a new 'rtthread' folder under the prepared IAR bare-metal project, and add the following files in that folder:

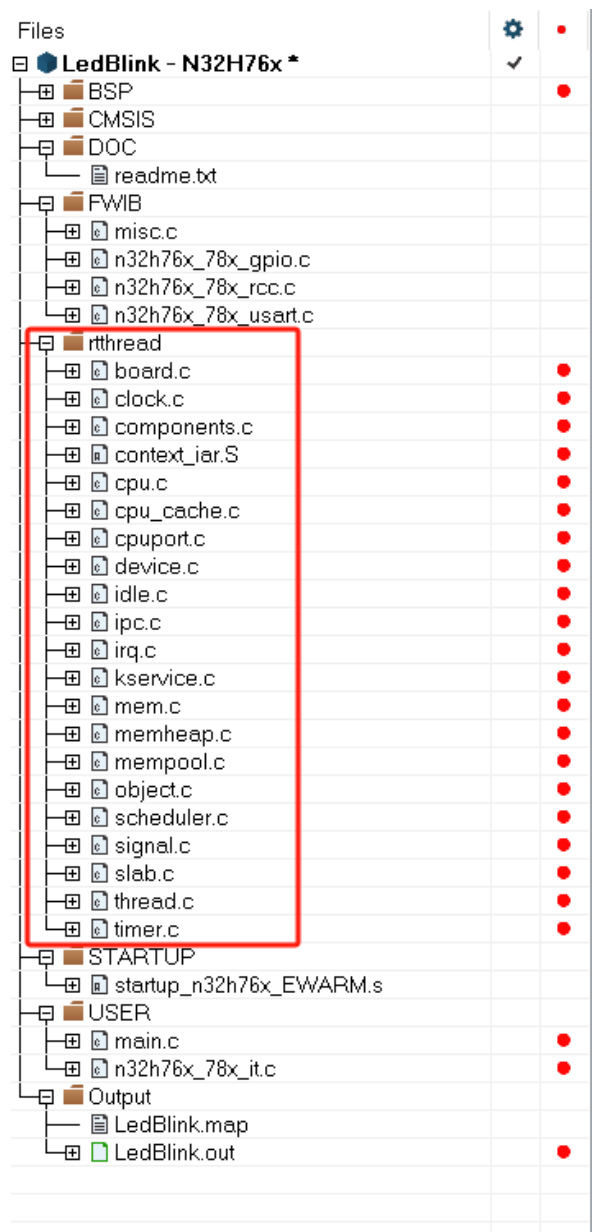
- The include, libcpu, and src folders in the Nano source code
- Configuration files: two files in the bsp folder of the source code: board.c and rtconfig.h

Figure 3-3 rtthread folder

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

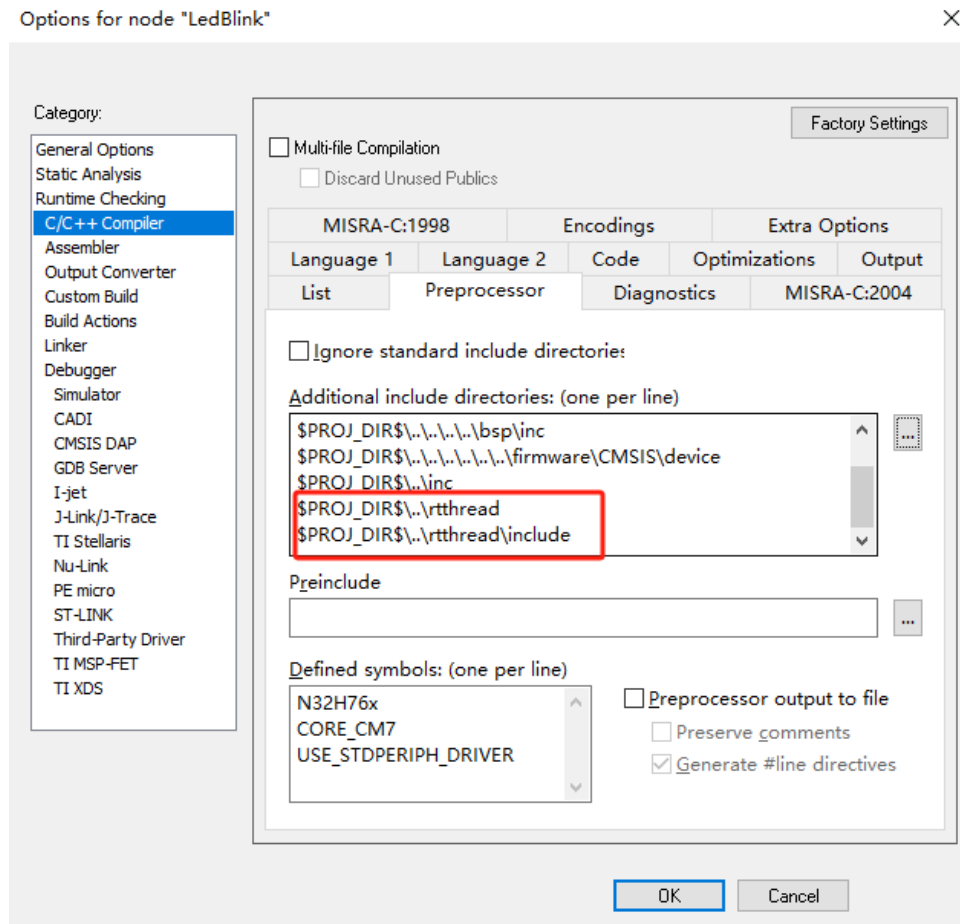
Double-click to open the IAR bare-metal project, create a new rtthread group, and add the following source code under this group:

- Add all files from the rtthread/src/ folder to the project
- Add the CPU port files and context switching files for the corresponding kernel from the rtthread/libcpu/ folder to the project: cpuport.c and context_iar.S
- Add board.c from the rtthread/ folder

Figure 3-4 RT-Thread Grouping


3.3.2 Add Header File Path

Click Project -> Options... to enter the interface shown below, and add the path to the location of the rtconfig.h header file, as well as the path to the header files in the include folder.

Figure 3-5 Add header file path


3.4 Adapting RT-Thread Nano

3.4.1 Interrupts and Exception Handling

RT-Thread will take over the exception handler `HardFault_Handler()` and the pendable service handler `PendSV_Handler()`. These two functions have already been implemented by RT-Thread, so you need to remove the `HardFault_Handler()` and `PendSV_Handler()` functions from the `n32h76x_78x_it.c` file in the project to avoid duplicate definitions during compilation. If the project compiles without any duplicate function definition errors, no modifications are needed.

3.4.2 System Clock Configuration

It is necessary to implement the system clock configuration in `board.c` (to provide working clocks for the MCU and peripherals) and the OS tick configuration (to provide a heartbeat/tick for the operating system).

As shown in the code below, system initialization and OS Tick configuration need to be done in the `board.c` file. For the Cortex-M architecture, the `rt_os_tick_callback()` function needs to be called in the `SysTick_Handler()` function.

Figure 3-6 System clock and OS tick configuration in board.c

```

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
}

```

3.4.3 CONSOLE Serial Port Initialization

Enable the console function by turning on RT_USING_CONSOLE in rtconfig.h, then implement the UART initialization in the `uart_init()` function in board.c and implement the console output redirection in the `rt_hw_console_output()` function.

Figure 3-7 Console serial port initialization in board.c

```
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);
```

Figure 3-8 Console output redirection in board.c

```
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 Memory Heap Initialization

The initialization of the system memory heap is completed in the `rt_hw_board_init()` function in `board.c`. Whether the memory heap feature is used depends on whether the macro `RT_USING_HEAP` in `rtconfig.h` is enabled. RT-Thread Nano does not enable the memory heap feature by default, which helps maintain a smaller footprint without allocating space for the memory heap.

Enabling the system heap allows the use of dynamic memory functions, such as `rt_malloc`, `rt_free`, and various system APIs for dynamically creating objects. If you need to use the system memory heap functionality, simply enable the `RT_USING_HEAP` macro in `rtconfig.h`, at which point the memory heap initialization function `rt_system_heap_init()` will be called.

Figure 3-9 Call to `rt_system_heap_init()`

```
/**
 * 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
}
```

Initializing the memory heap requires two parameters: the starting address and the ending address of the heap. By default, the system uses an array as the heap and obtains the starting and ending addresses of the heap. The size of this array can be manually changed, as shown below:

Figure 3-10 Configuring array as heap size

```

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

```

Note: After enabling the heap dynamic memory feature, the default heap size is relatively small. You need to increase it when using it, otherwise memory allocation or thread creation may fail. There are two ways to modify it:

- *Method 1: You can directly modify the size of RT_HEAP_SIZE defined in the array.*
- *Method 2: Use the end of the RAM ZI section as the starting address of the HEAP, and use the end address of the RAM as the ending address of the HEAP.*

Note: During the porting process, please pay attention to the configuration for dynamic memory allocation in kservice.c. The new version of the RT-Thread Nano kernel manages Memory Pool, Dynamic Heap, and Tiny Memory differently. In addition to enabling the corresponding macros in rtconfig.h, it is also necessary to define the macros for the respective memory management.

Figure 3-11 Memory management configuration in rtconfig.h

```

// <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>

```

Figure 3-12 Memory management selection macro in kservice.c

```

#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 Writing First Application

After successfully porting RT-Thread Nano, you can start writing your first application code to verify the porting results. At this point, the main() function transforms into a thread of the RT-Thread operating system. You can now implement your first application in the main() function: flashing the onboard LED indicator. Here, it is directly modified based on the bare-metal LED indicator.

1. First, add the relevant RT-Thread header file <rtthread.h> at the beginning of the file.
2. Implement the LED blinking code in the main() function (that is, in the main thread).
3. Replace the delay function with the delay function provided by RT-Thread, rt_thread_mdelay(). This function will trigger system scheduling, switch to other threads, and reflect the real-time characteristics of threads.

Figure 3-13 LED Flasher Application Example

```
#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 Configuring RT-Thread Nano

Users can configure the corresponding functions according to their needs by enabling or disabling the macro definitions in the rtconfig.h file. Below is a code snippet from rtconfig.h:

Figure 3-14 Code snippet of rtconfig.h

```
// <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 Version history

Date	Version	Modify
2025/12/29	V1.0.0	Initial version

5 Notice

This document is the exclusive property of NSING TECHNOLOGIES PTE. LTD. (Hereinafter referred to as NSING). This document, and the product of NSING described herein (Hereinafter referred to as the Product) are owned by NSING under the laws and treaties of Republic of Singapore and other applicable jurisdictions worldwide. The intellectual properties of the product belong to NSING Technologies Inc. and NSING Technologies Inc. does not grant any third party any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only. NSING reserves the right to make changes, corrections, enhancements, modifications, and improvements to this document at any time without notice. Please contact NSING and obtain the latest version of this document before placing orders. Although NSING has attempted to provide accurate and reliable information, NSING assumes no responsibility for the accuracy and reliability of this document. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NSING be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product. NSING Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, Insecure Usage'. Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to supporter sustain life. All Insecure Usage shall be made at user's risk. User shall indemnify NSING and hold NSING harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage Any express or implied warranty with regard to this document or the Product, including, but not limited to. The warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law. Unless otherwise explicitly permitted by NSING, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.