
移植 RT-THREAD NANO 到 RISC-V

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 ©2019



WWW.RT-THREAD.ORG

Wednesday 27th November, 2019

目录

目录	i
1 准备工作	1
1.1 下载 Nano 源码	1
1.2 基础工程准备	1
2 添加 RT-Thread Nano 到工程	2
2.1 添加 Nano 源文件	2
2.2 添加头文件路径	4
3 适配 RT-Thread Nano	5
3.1 修改 start.S	5
3.2 中断与异常处理	6
3.3 系统时钟配置	7
3.4 内存堆初始化	8
4 编写第一个应用	9
5 配置 RT-Thread Nano	10

本文介绍了如何移植 RT-Thread Nano 到 RISC-V 架构，以 Eclipse GCC 环境为例，基于一个 GD32V103 MCU 的基础工程作为示例进行讲解。

移植 Nano 的主要步骤：

1. 准备一个基础的 Eclipse 工程，并获取 RT-Thread Nano 源码压缩包。
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。

1 准备工作

- 下载 RT-Thread Nano 发布版本代码。
- 准备一份基础的裸机源码工程，如 LED 指示灯闪烁示例代码。

1.1 下载 Nano 源码

点击[此处](#) 下载 RT-Thread Nano 源码。

1.2 基础工程准备

在移植 RT-Thread Nano 之前，我们需要准备一个能正常运行的裸机工程。作为示例，本文使用的是基于 GD32V103 的一个 LED 闪烁程序。程序的主要截图如下：

```
46 int main(void)
47 {
48     gd_eval_led_init(LED1);
49
50     while(1)
51     {
52         gd_eval_led_on(LED1);
53         delay_1ms(1000);
54
55         gd_eval_led_off(LED1);
56         delay_1ms(1000);
57     }
58 }
59
```

图 1：裸机示例代码

在我们的例程中主要做了系统初始化与 LED 闪烁功能，编译下载程序后，就可以看到开发板上的 LED 在闪烁了。读者可以根据自己的需要使用的芯片，完成一个类似的裸机工程。

2 添加 RT-Thread Nano 到工程

2.1 添加 Nano 源文件

在准备好的 Eclipse 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。注意 libcpu 仅保留与该芯片架构相关的文件，如示例中使用的是 `bunblebee` 与 `common`。
- 配置文件：示例代码 `rtthread-nano/bsp` 中的两个文件：`board.c` 与 `rtconfig.h`。

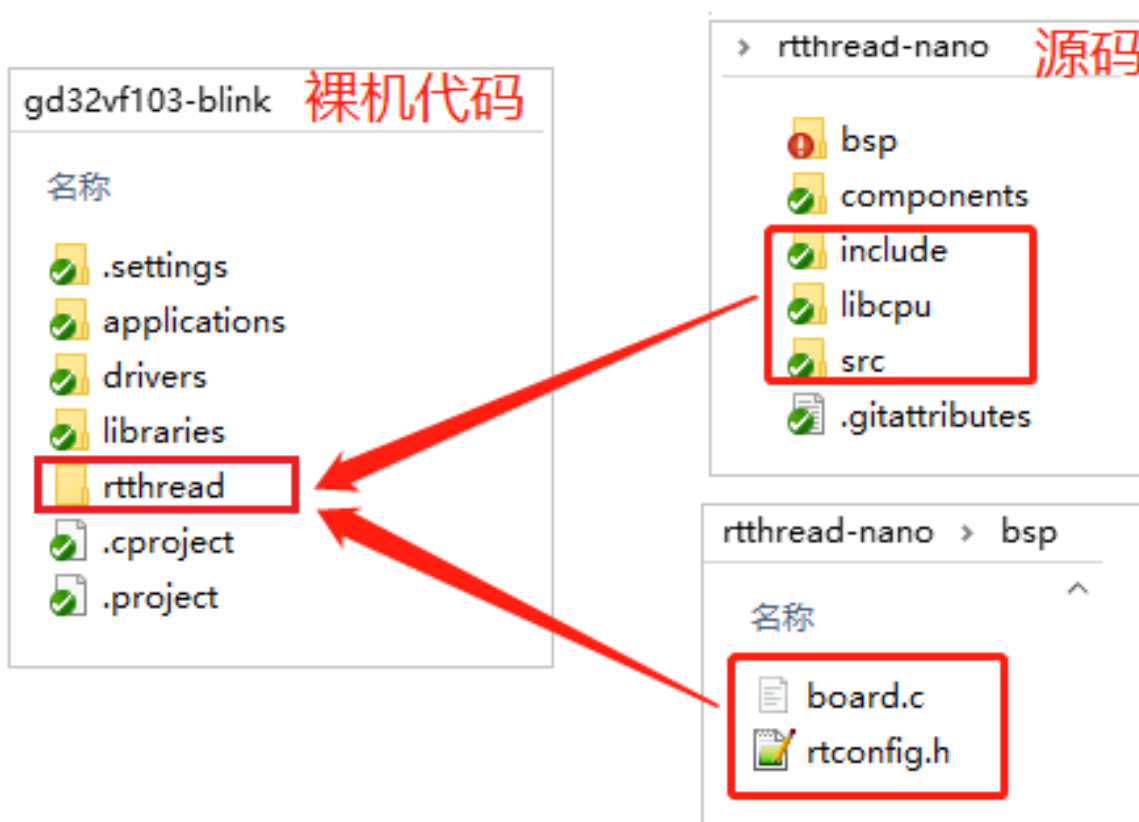


图 2: 给裸机工程添加 Nano 源码以及必要的配置文件

重新打开 eclipse 工作空间，导入工程，rtthread 已经加载到工程中：

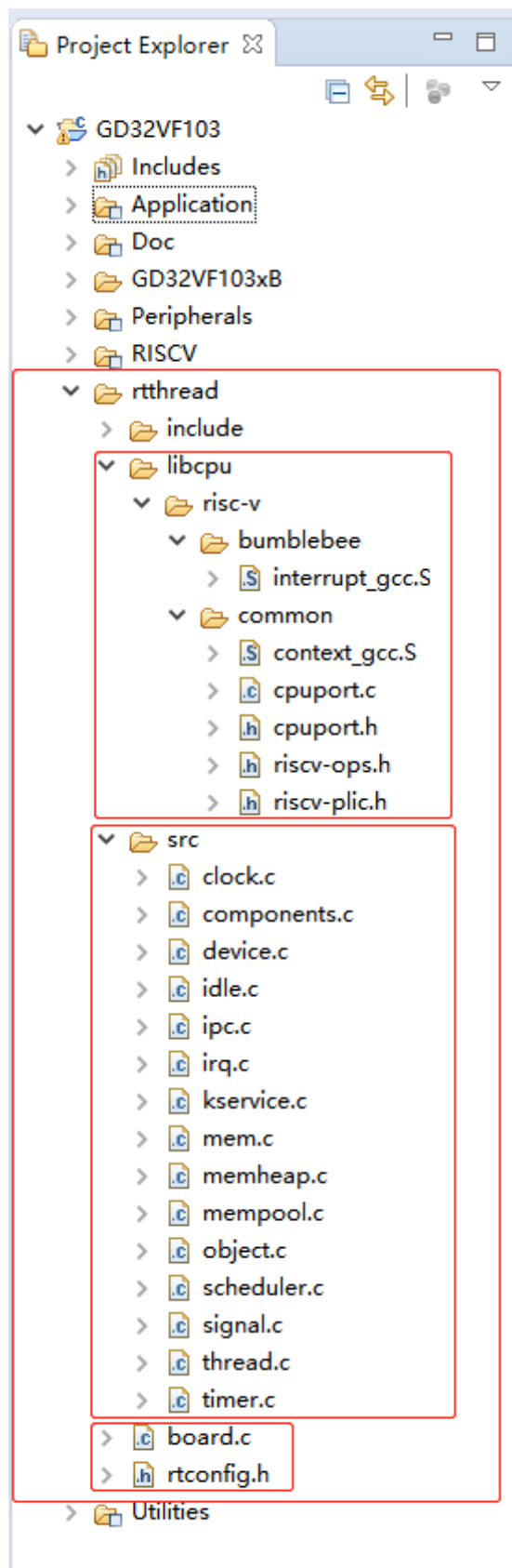


图 3: eclipse 工程

Cortex-M 芯片内核移植代码:

```
context_gcc.s  
cpuport.c
```

Kernel 文件包括:

```
clock.c  
components.c  
device.c  
idle.c  
ipc.c  
irq.c  
kservice.c  
mem.c  
object.c  
scheduler.c  
thread.c  
timer.c
```

板级配置代码及配置文件:

```
board.c  
rtconfig.h
```

2.2 添加头文件路径

右击工程，点击 **properties** 进入下图所示界面，点击 **C/C++ Build -> settings**，分别添加汇编与 C 的头文件路径：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。然后点击 **C/C++ General -> Path and Symbols**，添加相应的头文件，最后点击应用即可。

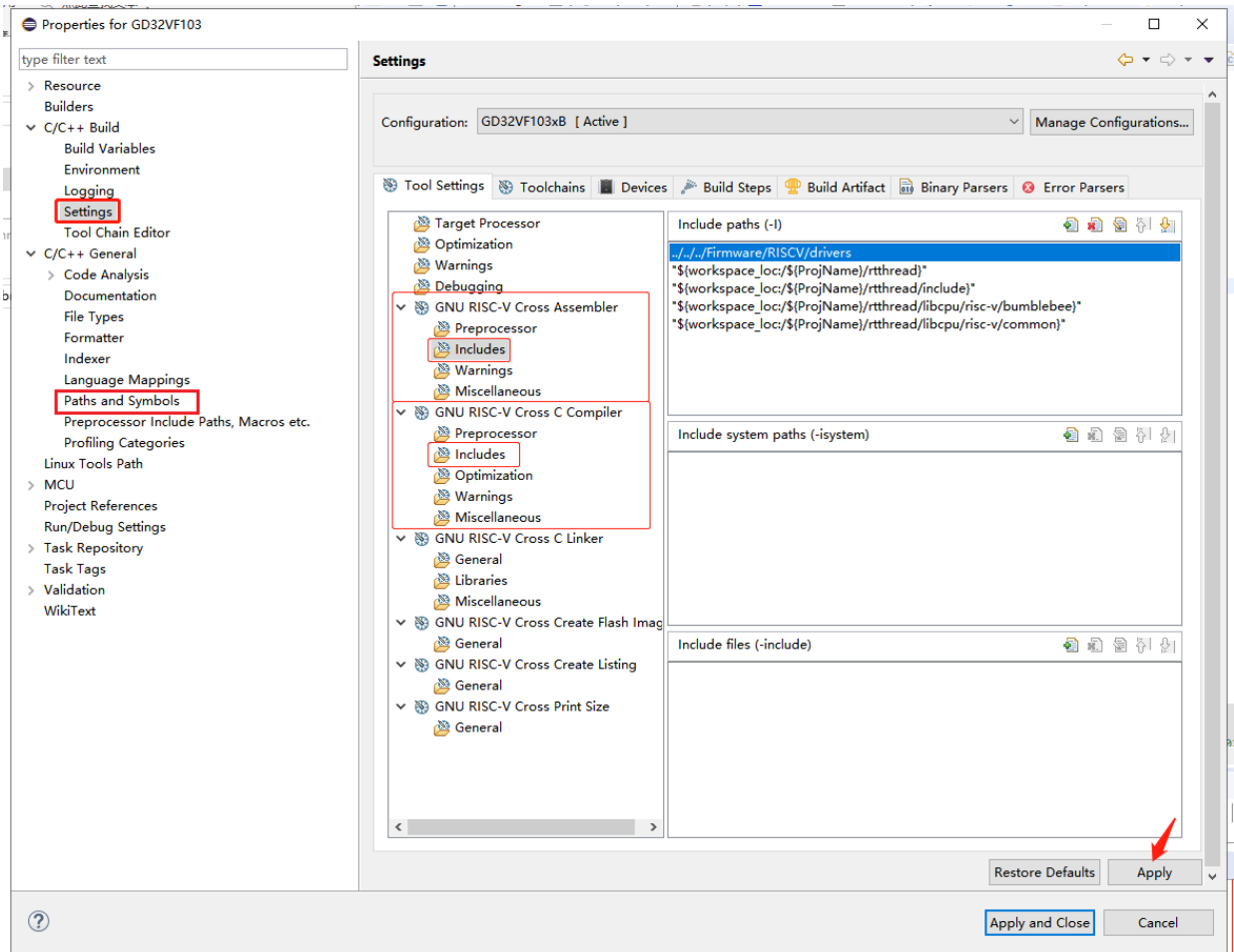


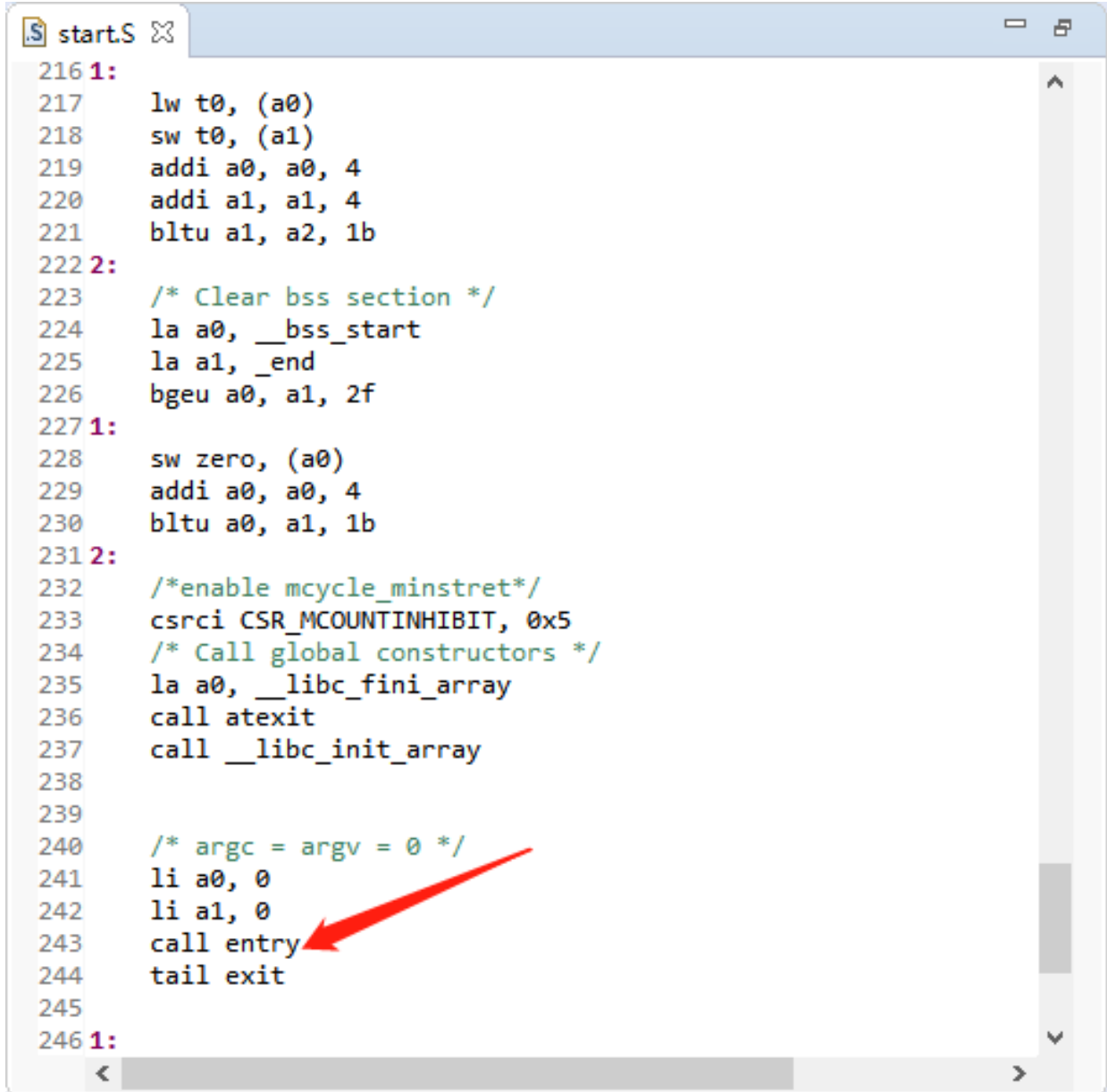
图 4: 添加头文件路径

3 适配 RT-Thread Nano

3.1 修改 start.S

修改启动文件，实现 RT-Thread 的启动：由于 RT-Thread Nano 在 GCC 环境下的启动是由 entry() 函数调用了启动函数 rt_thread_startup()，所以需要修改启动文件 start.S，使其在启动时先跳转至 entry() 函数执行，而不是跳转至 main()，这样就实现了 RT-Thread 的启动。

```
/* RT-Thread 在 GCC 下的启动方式 */
int entry(void)
{
    rtthread_startup();
    return 0;
}
```



```
start.S
216 1:
217     lw t0, (a0)
218     sw t0, (a1)
219     addi a0, a0, 4
220     addi a1, a1, 4
221     bltu a1, a2, 1b
222 2:
223     /* Clear bss section */
224     la a0, __bss_start
225     la a1, __end
226     bgeu a0, a1, 2f
227 1:
228     sw zero, (a0)
229     addi a0, a0, 4
230     bltu a0, a1, 1b
231 2:
232     /*enable mcycle_minstret*/
233     csrwi CSR_MCOUNTINHIBIT, 0x5
234     /* Call global constructors */
235     la a0, __libc_fini_array
236     call atexit
237     call __libc_init_array
238
239
240     /* argc = argv = 0 */
241     li a0, 0
242     li a1, 0
243     call entry
244     tail exit
245
246 1:
```

图 5: 修改 *start.S*

3.2 中断与异常处理

RT-Thread 提供中断管理方法，当系统没有实现类似中断向量表的功能，物理中断要和用户的中断服务例程相关联，就需要使用中断管理接口对中断进行管理，这样当发生中断时就可以触发相应的中断，执行中断服务例程。

本例程中的 gd32f103 芯片在启动文件中提供了中断向量表，用户可直接使用中断向量提供的函数实现对应 IRQ。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。

3.3 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）。

配置示例如下图所示，`riscv_clock_init()` 配置了系统时钟，`ostick_config()` 配置了 OS Tick。

```

66 void rt_hw_board_init()
67 {
68     /* system clock Configuration */
69     riscv_clock_init();
70
71     /* OS Tick Configuration */
72     ostick_config(TMR_FREQ / RT_TICK_PER_SECOND);
73
74     /* Call components board initial (use INIT_BOARD_EXPORT()) */
75 #ifdef RT_USING_COMPONENTS_INIT
76     rt_components_board_init();
77 #endif
78
79 #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
80     rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
81 #endif
82 }
83

```

图 6: 系统时钟与 OS Tick 配置

`riscv_clock_init()` 配置了系统时钟，示例如下：

```

39
40 static void ostick_config(rt_uint32_t ticks) {
41     /* set value */
42     *(rt_uint64_t *) (TMR_CTRL_ADDR + TMR_MTIMECMP) = ticks;
43     /* enable interrupt */
44     eclic_irq_enable(CLIC_INT_TMR, 0, 0);
45     /* clear value */
46     *(rt_uint64_t *) (TMR_CTRL_ADDR + TMR_MTIME) = 0;
47 }
48

```

图 7: OS Tick 初始化配置

`ostick_config()` 配置了 OS Tick，示例如下，此处 OS Tick 使用硬件定时器实现，需要用户在 board.c 中实现该硬件定时器的中断服务例程 `eclic_mtip_handler()`，调用 RT-Thread 提供的 `rt_tick_increase()`。

```

84 /* This is the timer interrupt service routine. */
85 void eclic_mtip_handler(void) {
86     /* clear value */
87     *(rt_uint64_t *) (TMR_CTRL_ADDR + TMR_MTIME) = 0;
88
89     /* enter interrupt */
90     rt_interrupt_enter();
91     /* tick increase */
92     rt_tick_increase();
93
94     /* leave interrupt */
95     rt_interrupt_leave();
96 }
97

```

图 8: OS Tick 的实现: 硬件定时器 ISR 实现

由于 `eclic_mtip_handler()` 中断服务例程由用户在 `board.c` 中重新实现, 做了系统 OS Tick, 所以需要将自定义的 `eclic_mtip_handler` 删除, 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 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 `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, 如下所示:

```

66 void rt_hw_board_init()
67 {
68     /* system clock Configuration */
69     riscv_clock_init();
70
71     /* OS Tick Configuration */
72     ostick_config(TMR_FREQ / RT_TICK_PER_SECOND);
73
74     /* Call components board initial (use INIT_BOARD_EXPORT()) */
75 #ifdef RT_USING_COMPONENTS_INIT
76     rt_components_board_init();
77 #endif
78
79 #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
80     rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
81 #endif
82 }

```

图 9: 系统 heap 初始化

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:

```
#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE]; // heap default size: 4K(1024 * 4)
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;
}
#endif
```

图 10: 默认 heap 的实现

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：

- 可以直接修改数组中定义的 RT_HEAP_SIZE 的大小，至少大于各个动态申请内存大小之和，但要小于芯片 RAM 总大小。
- 也可以参考《RT-Thread Nano 移植原理》——实现动态内存堆 章节进行修改，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，这是 heap 能设置的最大值的方法。

4 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。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：初始化 LED 引脚、在循环中点亮 / 熄灭 LED。
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。

```

39
40 #include <rtthread.h>
41
42 int main(void)
43 {
44     gd_eval_led_init(LED1);
45
46     while(1)
47     {
48         gd_eval_led_on(LED1);
49         rt_thread_mdelay(500);
50
51         gd_eval_led_off(LED1);
52         rt_thread_mdelay(500);
53     }
54 }
55

```

图 11: RT-Thread 第一个应用

编译程序之后下载到芯片就可以看到基于 RT-Thread 的程序运行起来了，LED 正常闪烁。

注意事项：当添加 RT-Thread 之后，裸机中的 `main()` 函数会自动变成 RT-Thread 系统中 main 线程的入口函数。由于线程不能一直独占 CPU，所以此时在 `main()` 中使用 `while(1)` 时，需要有让出 CPU 的动作，比如使用 `rt_thread_mdelay()` 系列的函数让出 CPU。

与裸机 LED 闪烁应用代码的不同：

1). 延时函数不同：RT-Thread 提供的 `rt_thread_mdelay()` 函数可以引起操作系统进行调度，当调用该函数进行延时，本线程将不占用 CPU，调度器切换到系统的其他线程开始运行。而裸机的 `delay` 函数是一直占用 CPU 运行的。

2). 初始化系统时钟的位置不同：移植好 RT-Thread Nano 之后，不需要再在 `main()` 中做相应的系统配置（如 `hal` 初始化、时钟初始化等），这是因为 RT-Thread 在系统启动时，已经做好了系统时钟初始化等的配置，这在上一小节“系统时钟配置”中有讲解。

5 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：

```

...

// <h>IPC(Inter-process communication) Configuration
// <c1>Using Semaphore
// <i>Using Semaphore
#define RT_USING_SEMAPHORE

```

```

// </c>
// <cl>Using Mutex
// <i>Using Mutex
// #define RT_USING_MUTEX // 打开此宏则使能互斥量的使用
// </c>
// <cl>Using Event
// <i>Using Event
// #define RT_USING_EVENT // 打开此宏则使能事件集的使用
// </c>
// <cl>Using MailBox
// <i>Using MailBox
// #define RT_USING_MAILBOX // 打开此宏则使能邮箱的使用
// </c>
// <cl>Using Message Queue
// <i>Using Message Queue
// #define RT_USING_MESSAGEQUEUE // 打开此宏则使能消息队列的使用
// </c>
// </h>

// <h>Memory Management Configuration
// <cl>Using Memory Pool Management
// <i>Using Memory Pool Management
// #define RT_USING_MEMPOOL // 打开此宏则使能内存池的使用

...

```

RT-Thread Nano 默认未开启宏 `RT_USING_HEAP`，故只支持静态方式创建任务及信号量。若要通过动态方式创建对象则需要要在 `rtconfig.h` 文件里开启 `RT_USING_HEAP` 宏定义。完整配置详见[《RT-Thread Nano 配置》](#)。