芯动力——硬件加速设计方法

第七章 基于平头哥E902处理器的SoC设计

——(6) RT-Thread Nano移植

邸志雄@西南交通大学

zxdi@home.swjtu.edu.cn

slides与源代码网址 http://www.dizhixiong.cn/class5/

RT-Thread介绍

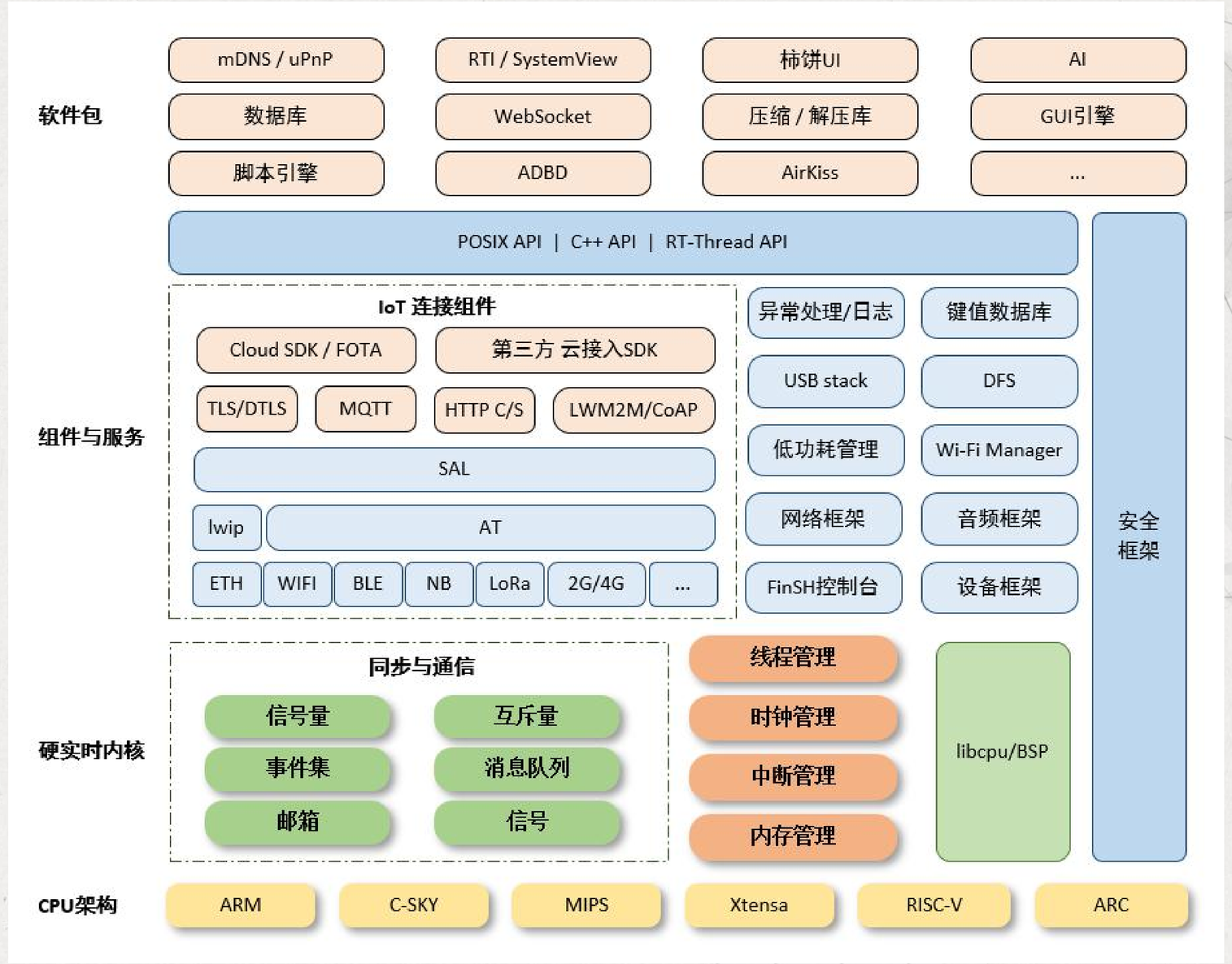
简介

Real Time-Thread值

RT-Thread介绍随着物联网的兴起，它正演变成一个功能强大、组件丰富的物联网操作系统，而不仅仅是一个实时内核。

RT-Thread Nano

- 从图中可以看到，RT-Thread是以操作系统内核（可以是 RTOS、Linux 等）为基础，包括如文件系统、图形库等较为完整的中间件组件，具备低功耗、安全、通信协议支持和云端连接能力的软件平台。这也是RT-Thread与其它RTOS如FreeRTOS、uC/OS的主要区别。除此之外，还可以看到RT-Thread支持ARM、MIPS、RISC-V等多种主流的CPU架构。



RT-Thread Nano

- 为RT-Thread Nano是一款可裁剪的、抢占式实时多任务的RTOS。其内存资源占用极小，功能包括任务处理、软件定时器、信号量、邮箱和实时调度等相对完整的实时操作系统特性。下图是 RT-Thread Nano 的软件框图，包含支持的 CPU 架构与内核源码，还有可拆卸的 FinSH 组件。

RT-Thread Nano

RT-Thread Nano特性

- 1.、代码简单。与完整版的RT-Thread不同，RT-Thread Nano只是一个纯净的实时内核，去除了完整版特有的 device 框架和组件，无需配置工具，直接将源码添加至工程即可。
- 2、 移植简单。内核源代码、CPU支持和板级支持文件分层设计，使 Nano 的移植过程变得极为简单。事实上，添加 Nano 源码到工程，就已完成 90% 的移植工作。
- 3、使用简单。

易裁剪：Nano 的配置文件为 rtconfig.h，可以在文件中开关宏定义以开启或关闭某些功能。

易添加 FinSH 组件：只需要对接两个必要的函数即可完成 FinSH 组件的移植。

为什么要使用RTOS

BareMetal裸机开发（不使用RTOS）

- 所有的操作都是在一个无限的大循环while(1)里面实现，随着系统变得复杂编程变得困难，开发效率低。
- 功能复杂的情况下，实时性较差或较难保证。
- 生态较差，许多高级软件组件依赖于RTOS来实现。例如乐鑫、TI等提供的WIFi SoC的SDK都只支持操作系统开发。

为什么要使用RTOS

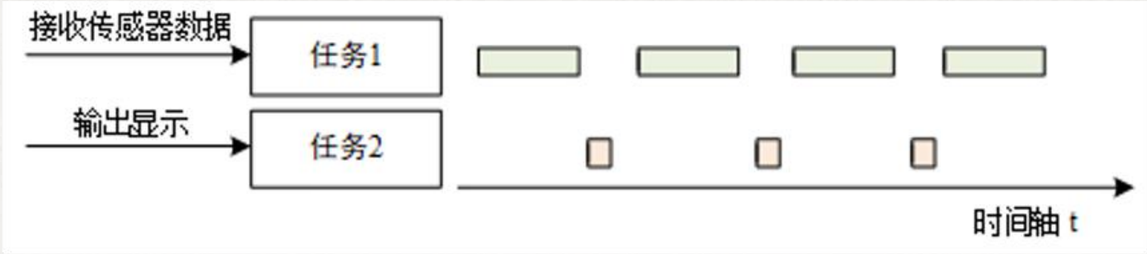
RTOS开发

- 只需要为新任务创建一个新的线程即可，模块化程度较好，开发效率较高。
- 通过合理设置线程的优先级等操作，做好全局的统筹规划，可以保证较好的实时性。
- 生态较好，如RT-Thread提供了很多组件和开发包，降低了开发难度。

线程间的切换会造成系统额外的资源开销

RTOS中线程是什么？

- 在实现一个复杂的应用时，我们通常会将其分解成多个小的子任务，如下图所示，一个子任务不间断地读取传感器数据，并将数据写到共享内存中，另外一个子任务周期性的从共享内存中读取数据，并将传感器数据输出到显示屏上。



- 在 RT-Thread 中，与上述子任务对应的程序实体就是线程，线程是实现任务的载体，它是最基本的调度单位，它描述了一个任务执行的运行环境，也描述了这个任务所处的优先等级等。当线程运行时，它会认为自己是以独占 CPU 的方式在运行，线程执行时的运行环境称为上下文，具体来说就是各个变量和环境，包括所有的寄存器变量、堆栈、内存信息等。

RTOS中线程是什么？

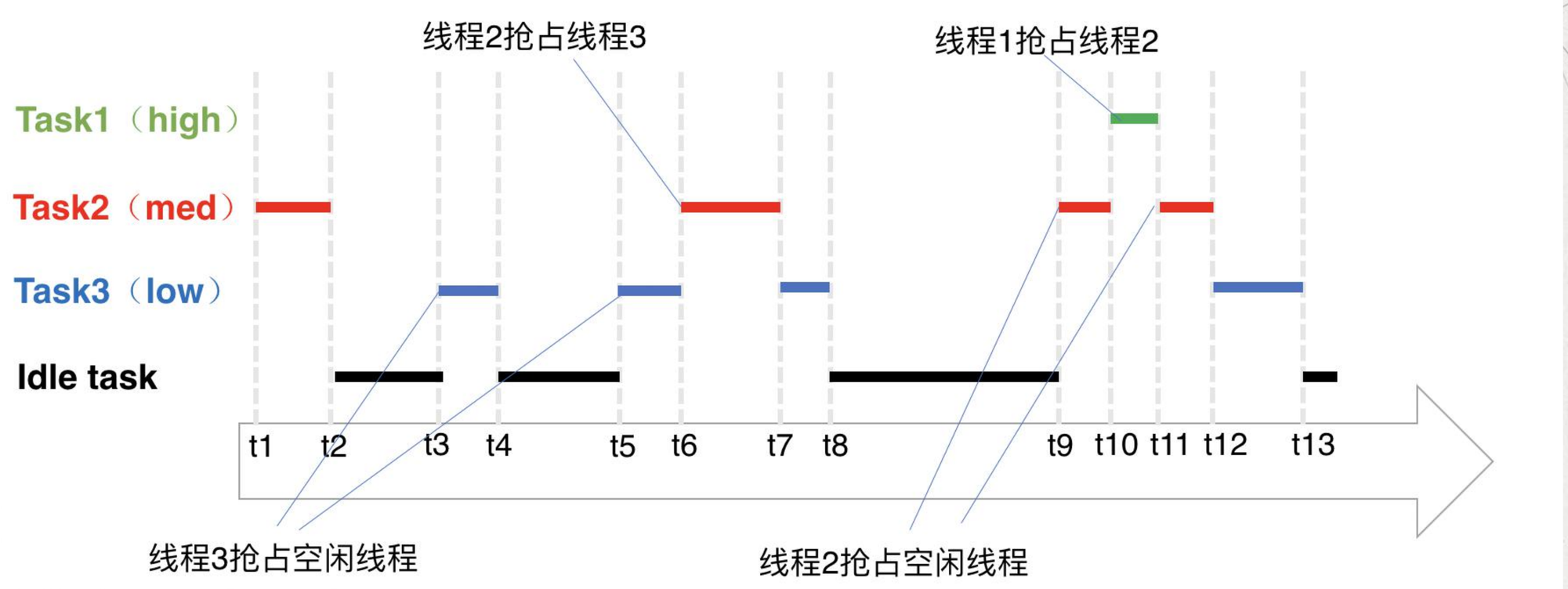
RTOS多线程是否意味着CPU同时处理多个任务？

RTOS可以让许多个任务同时运行，但这并不意味着处理器在同一时刻真地执行了多个任务。事实上，一个处理器核在某一时刻只能运行一个程序，由于每次对一个任务的执行时间很短、任务与任务之间通过任务调度器进行非常快速地切换（调度器根据优先级决定此刻该执行的任务），给人造成多个任务在一个时刻同时运行的错觉。

这里的任务调度器也就是线程调度器。

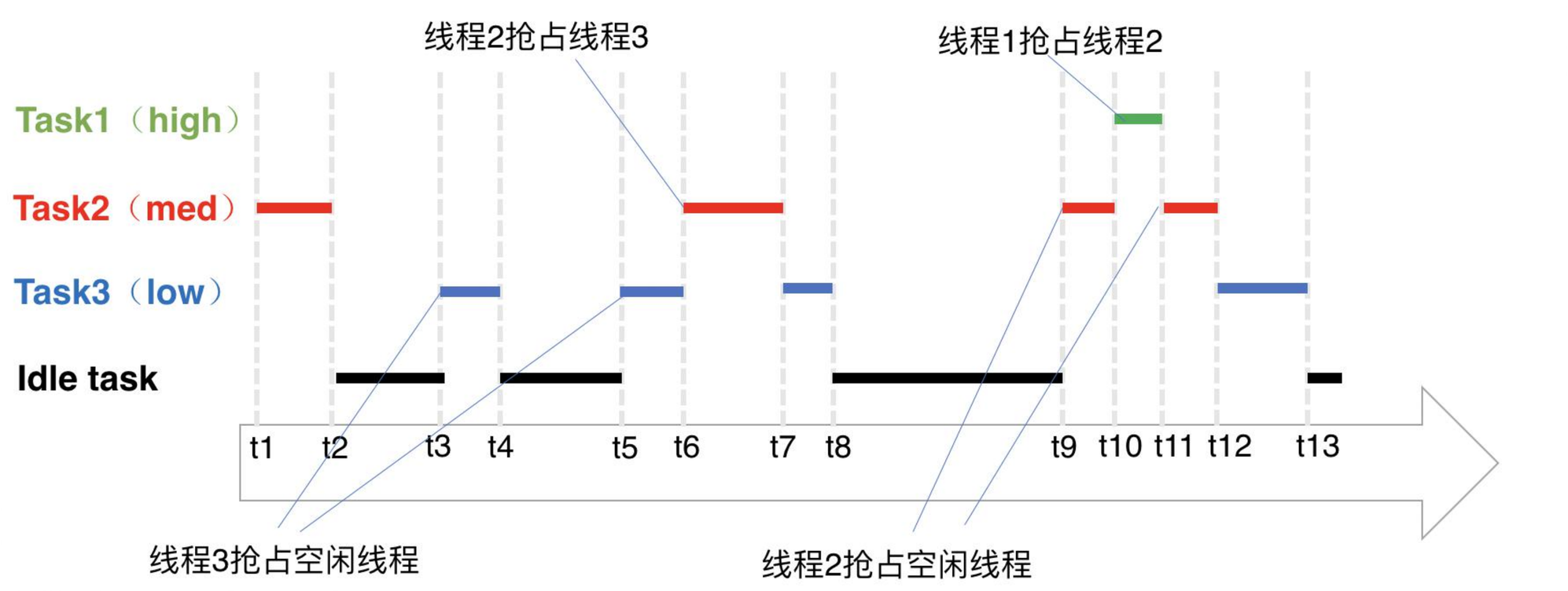
RTT线程调度器是如何工作的?

- 任何操作系统都需要提供一个时钟节拍，可以看做是系统心跳，通过定时器中断来实现。在每个时钟节拍里，RTT线程调度器会从线程就绪列表中查找出最高优先级的线程执行，保证高优先级的线程立刻得到CPU的使用权。



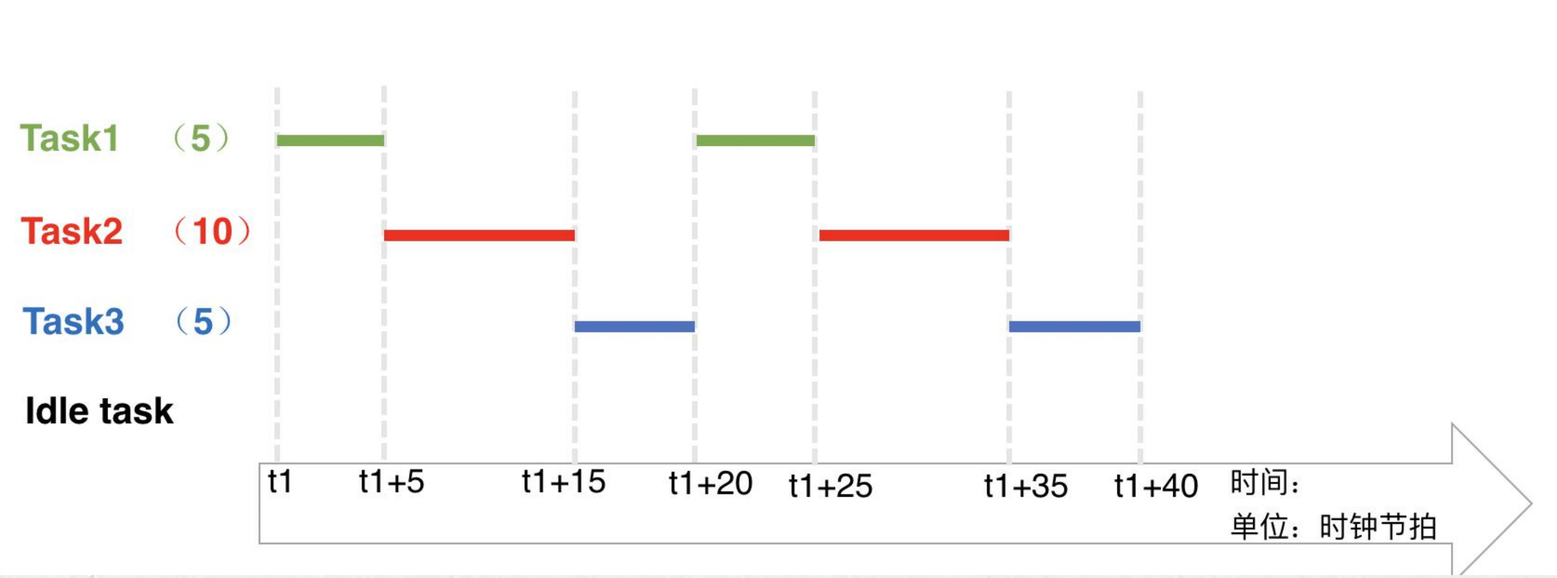
RTT线程调度器是如何工作的?

- 如图共有用户创建的三个线程和空闲线程，空闲线程是系统创建的最低优先级的线程，用于释放系统资源，线程状态永远为就绪态。可以看出，只有在线程就绪（等待执行）的情况下，才可以抢占低优先级的线程。



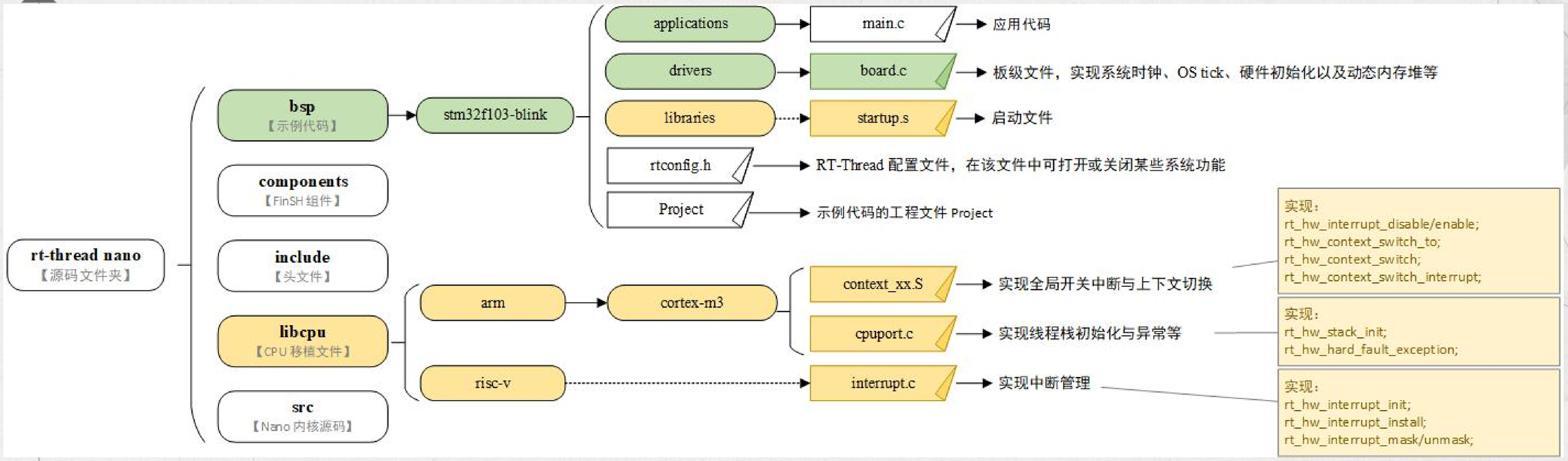
当线程就绪列表中不同线程优先级相同如何处理?

- 每个线程在初始化时都被赋予时间片这个参数，其单位为时钟节拍的个数。线程调度器在遇到相同优先级的就绪线程时采用时间片轮转调度方式进行调度。



- 假设三个线程优先级相同并始终处于就绪状态，并且时间片分别设置为了5、10、5。则调度器会依次在三个线程中切换执行，并且每次三个任务分别执行5、10、5个时钟节拍。

RT-Thread Nano移植目录结构



- RT-Thread内核源文件位于include和src文件夹下，直接添加进wujian100 SDK即可，无需任何修改。
- RT-Thread的移植主要包含板级移植和CPU架构的移植，板级移植是为了适配不同的板卡以调整使用外设资源，CPU架构移植是为了适配不同的处理器架构。
- 其中黄色部分是CPU架构移植部分，绿色的是板级移植部分，需要修改的主要就是这些部分

一、CPU架构移植 ①Libcpu层实现

- RT-Thread提供了一个libcpu抽象层屏蔽CPU架构的差异，libcpu层向上对内核提供了统一的接口。移植CPU架构，其实就是为libcpu层API编写不同的实现代码，以实现不同架构CPU的适配。由于这部分内容涉及寄存器的操作，因此需要用汇编语言来完成。

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

libcpu层相关的API

可以看出，libcpu层主要实现全局中断开关和线程上下文切换的功能。

一、CPU架构移植 ①Libcpu层实现

libcpu层是如何屏蔽CPU架构的差异，实现统一的接口？

- 内核在调用相应的功能时，只需要关注相应的接口，而不需要知道底层的实现，以实现屏蔽不同CPU架构的差异，用户则需根据不同的处理器构架完成底层部分。可以发现，libcpu层是RT-Thread可移植性高的一个原因。

```
.global rt_hw_interrupt_disable
.type rt_hw_interrupt_disable, %function
rt_hw_interrupt_disable:
    MRS     r0, PRIMASK
    CPSID   I
    BX      LR
```

ARM Cortex-M4架构上的实现

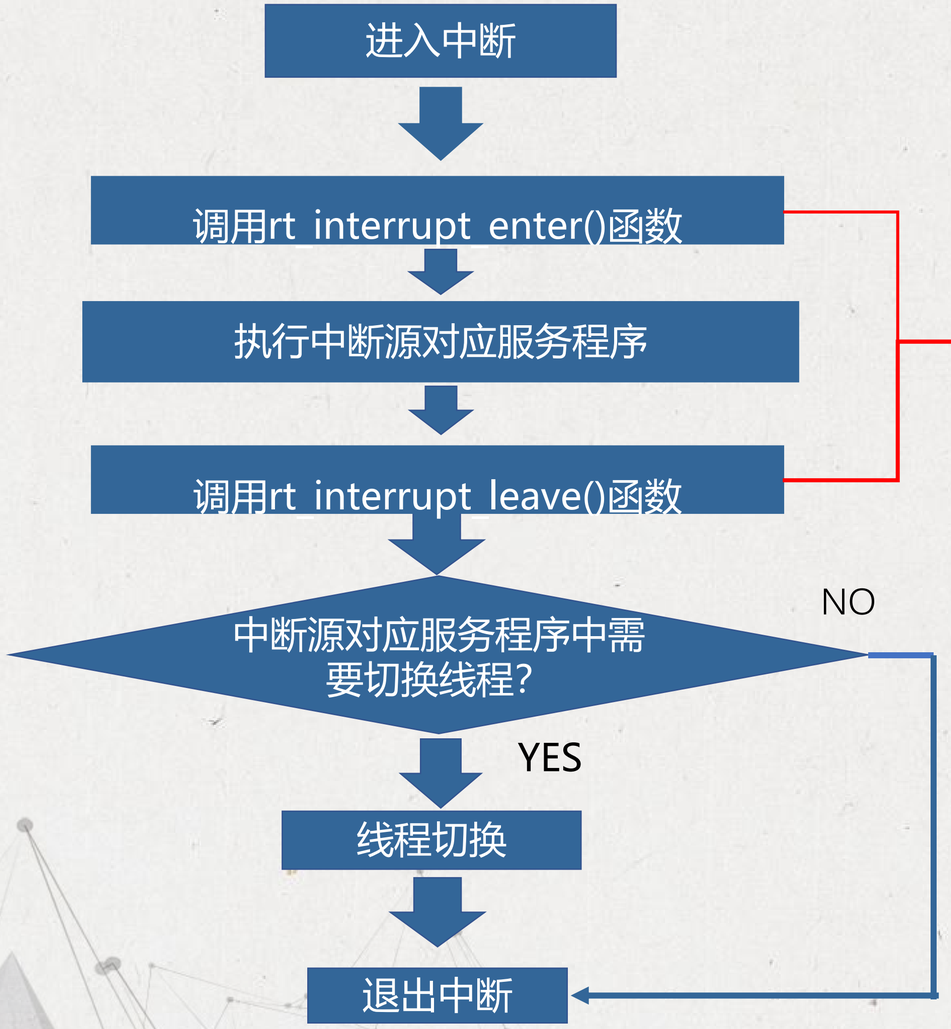
```
    .align  2
    .globl rt_hw_interrupt_disable
rt_hw_interrupt_disable:
    csrrci a0, mstatus, 8
    ret
```

RISC-V RV32EC架构上的实现

输libcpu层最简单的关闭中断函数

RT-Thread源码中libcpu层的程序

一、CPU架构移植 ②中断服务程序的修改



主要完成改变rt_interrupt_nest变量即修改中断标志位，反应当前是否处于中断状态

在线程环境下进行切换和在中断环境进行切换是存在差异的。

线程环境下，如果调用线程切换函数，那么可以马上进行上下文切换；

而在中断环境下为了保证中断的实时性，需要等待中断处理函数完成之后才能进行切换。

二、修改启动文件

什么是启动文件？wujian100启动文件完成了哪些工作？

- 启动文件是由汇编编写，是系统上电复位后第一个执行的程序，是在进入main用户程序之前完成的操作。每款芯片都有对应的启动文件，启动文件是由芯片厂商提供的。

```
Reset_Handler:
.option push
.option norelax
    la      gp, __global_pointer$
.option pop
    la      a0, Default_Handler
    ori     a0, a0, 3
    csrw    mtvec, a0

    la      a0, __Vectors
    csrw    mtvt, a0
```

配置异常服务程序入口和中断向量表

```
    /* Load data section */
    la      a0, __erodata
    la      a1, __data_start__
    la      a2, __data_end__
    bgeu    a1, a2, 2f
1:
    lw      t0, (a0)
    sw      t0, (a1)
    addi    a0, a0, 4
    addi    a1, a1, 4
    bltu    a1, a2, 1b
2:

    /* Clear bss section */
    la      a0, __bss_start__
    la      a1, __bss_end__
    bgeu    a0, a1, 2f
1:
    sw      zero, (a0)
    addi    a0, a0, 4
    bltu    a0, a1, 1b
```

将data段数据由ROM加载到RAM，将RAM中bss段数据赋值为0

```
2:

#ifndef __NO_SYSTEM_INIT
    jal     SystemInit
#endif

#ifndef __NO_BOARD_INIT
    jal     board_init
#endif

    jal     main
```

搭建中断向量表
将异常服务程序、中断向量表地址分别写入mtvec与mtvt寄存器

↓

完成数据段的搬移和赋值

↓

执行SystemInit系统初始化函数

↓

执行board_init板级初始化函数

↓

进入main用户应用程序

wujian100启动文件程序及工作流程

二、修改启动文件

什么是启动文件？wujian100启动文件完成了哪些工作？

- data段内存区域存放程序中初始化过的全局变量，这段区域被存放在ROM里，需要搬运至RAM中指定位置。
- bss段内存区域用来存放程序中未初始化的全局变量，需要将其所在RAM区域清零处理。

```
Reset_Handler:
.option push
.option norelax
    la      gp, __global_pointer$
.option pop
    la      a0, Default_Handler
    ori     a0, a0, 3
    csrw    mtvec, a0

    la      a0, __Vectors
    csrw    mtvt, a0
```

配置异常服务程序入口和中断向量表

```
    /* Load data section */
    la      a0, __erodata
    la      a1, __data_start__
    la      a2, __data_end__
    bgeu    a1, a2, 2f
1:
    lw      t0, (a0)
    sw      t0, (a1)
    addi    a0, a0, 4
    addi    a1, a1, 4
    bltu    a1, a2, 1b
2:

    /* Clear bss section */
    la      a0, __bss_start__
    la      a1, __bss_end__
    bgeu    a0, a1, 2f
1:
    sw      zero, (a0)
    addi    a0, a0, 4
    bltu    a0, a1, 1b
```

将data段数据由ROM加载到RAM，将RAM中bss段数据赋值为0

```
2:

#ifndef __NO_SYSTEM_INIT
    jal     SystemInit
#endif

#ifndef __NO_BOARD_INIT
    jal     board_init
#endif

    jal     main
```

搭建中断向量表
将异常服务程序、中断向量表地址分别写入mtvec与mtvt寄存器

↓

完成数据段的搬移和赋值

↓

执行SystemInit系统初始化函数

↓

执行board_init板级初始化函数

↓

进入main用户应用程序

wujian100启动文件程序及工作流程

二、修改启动文件

什么是启动文件？wujian100启动文件完成了哪些工作？

- SystemInit函数完成CLIC中断管理控制器的配置，如清除中断等待位等操作。
- Board_init函数主要完成串口的配置，从而可以在main应用程序中直接使用printf()函数。

```
Reset_Handler:
.option push
.option norelax
    la      gp, __global_pointer$
.option pop
    la      a0, Default_Handler
    ori     a0, a0, 3
    csrw    mtvec, a0

    la      a0, __Vectors
    csrw    mtvt, a0
```

配置异常服务程序入口和中断向量表

```
    /* Load data section */
    la      a0, __erodata
    la      a1, __data_start__
    la      a2, __data_end__
    bgeu    a1, a2, 2f
1:
    lw      t0, (a0)
    sw      t0, (a1)
    addi    a0, a0, 4
    addi    a1, a1, 4
    bltu    a1, a2, 1b
2:

    /* Clear bss section */
    la      a0, __bss_start__
    la      a1, __bss_end__
    bgeu    a0, a1, 2f
1:
    sw      zero, (a0)
    addi    a0, a0, 4
    bltu    a0, a1, 1b
```

将data段数据由ROM加载到RAM，将RAM中bss段数据赋值为0

```
2:

#ifndef __NO_SYSTEM_INIT
    jal     SystemInit
#endif

#ifndef __NO_BOARD_INIT
    jal     board_init
#endif

    jal     main
```

搭建中断向量表
将异常服务程序、中断向量表地址分别写入mtvec与mtvt寄存器

↓

完成数据段的搬移和赋值

↓

执行SystemInit系统初始化函数

↓

执行board_init板级初始化函数

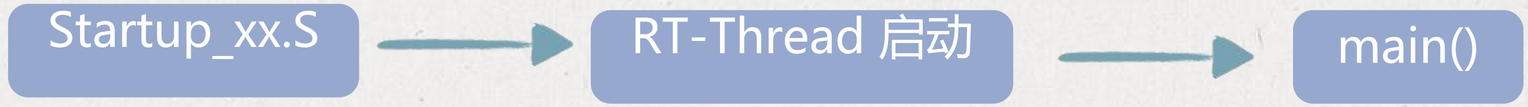
↓

进入main用户应用程序

wujian100启动文件程序及工作流程

二、修改启动文件

在使用RT-Thread 时，需要将 RT-Thread 的启动放在调用 main() 函数之前，如下图所示




```
#ifndef __NO_SYSTEM_INIT
    jal     SystemInit
#endif

#ifndef __NO_BOARD_INIT
    jal     board_init
#endif

    jal     main
```

启动代码修改前

```
#ifndef __NO_SYSTEM_INIT
    jal     SystemInit
#endif

#ifndef __NO_BOARD_INIT
    jal     board_init
#endif

    jal     entry
```

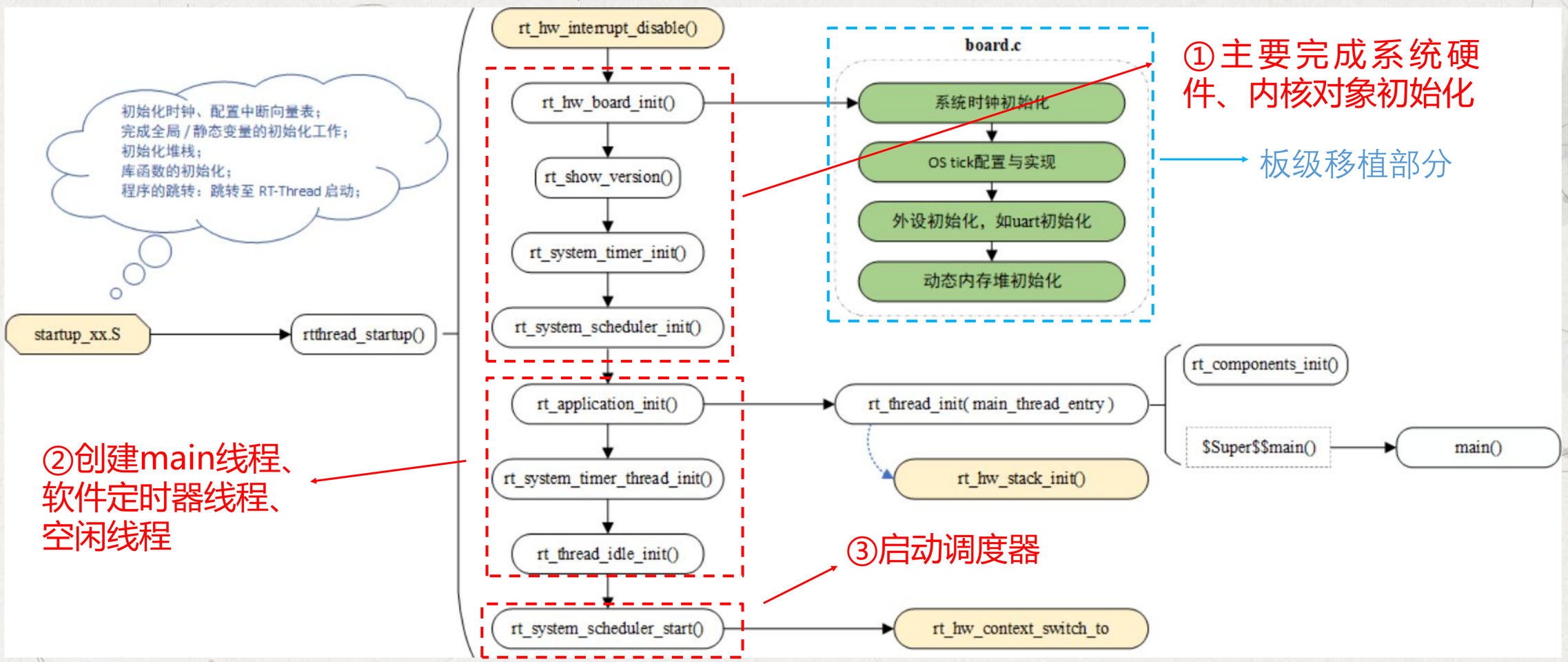
启动代码修改后

```
int entry(void)
{
    rtthread_startup();
    return 0;
}
```

- 通过修改启动文件使其跳转到entry函数，RT-Thread在entry函数中实现了RT-Thread的启动。
- 注意：RT-Thread将main函数当做线程看待，会在系统启动后调用。

三、板级移植

RT-Thread启动流程



- 可以看出，RT-Thread在完成一系列的初始化后，会创建main线程和idle空闲线程，根据系统配置选择是否创建软件定时器线程。随后启动调度器，系统切换到第一个线程开始运行（如 main 线程）。

三、板级移植

- 板级移植主要是针对 rt_hw_board_init() 函数内容的实现，函数中需要我们实现最基础的系统时钟配置、OS节拍实现，可以根据需要实现外设如GPIO/UART的初始化、系统内存堆栈的初始化等。

配置硬件定时器实现周期性中断

```
void rt_hw_board_init()
{
    uint32_t value;
    timer_handle_t timer1_handle=_SysTick_Config( RT_TICK_PER_SECOND);
    printf("board int success\n");

#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
}
```

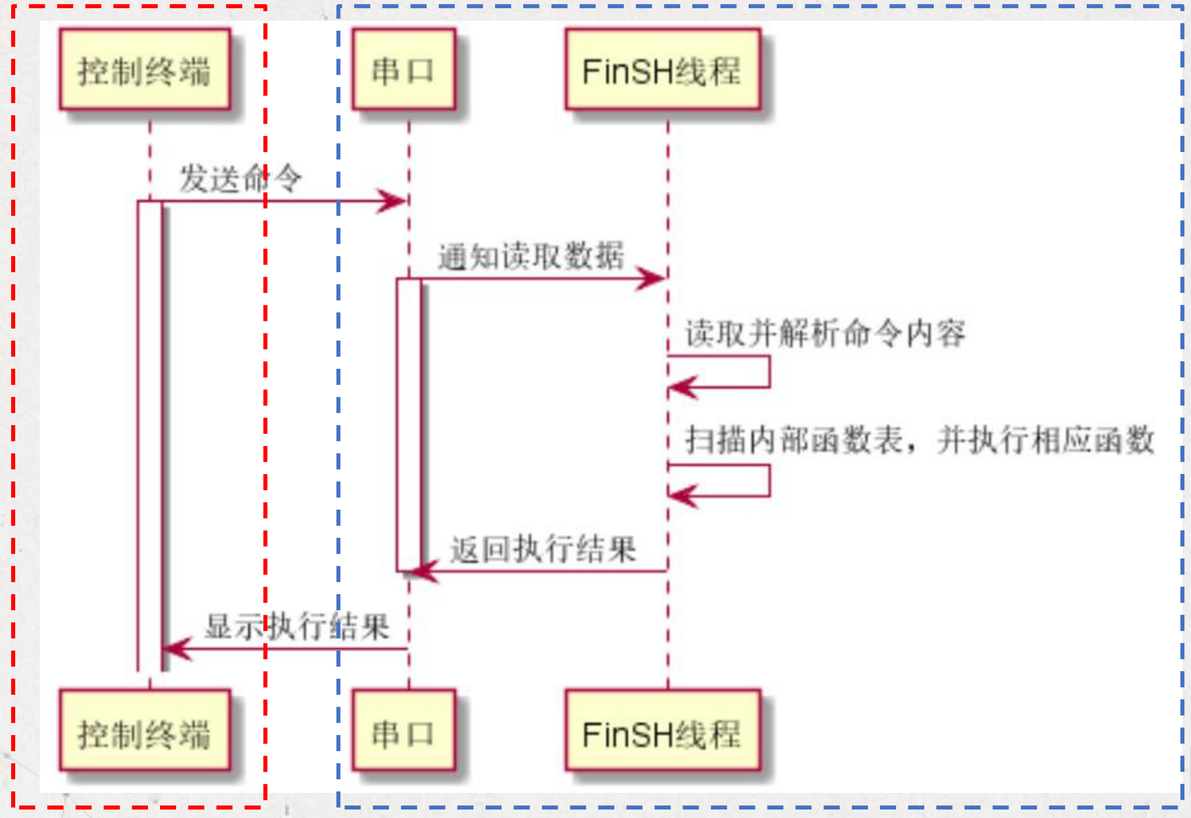
```
void SysTick_Handler(void)
{
    /* enter interrupt */
   rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}

static void timer_event_cb_fun(int32_t idx, timer_event_e event)
{
//  printf("%d int event\n",idx);
    SysTick_Handler();

}
```

定时器中断中实现OS节拍

四、 移植FinSH组件

- Finsh组件是RT-Thread的命令行组件，提供一套用户在命令行调用的操作接口，主要用于调试或查看系统信息。这个组件可以使用串口、以太网、USB等与PC机进行通信，这里我们使用串口实现该组件通信。

PC端

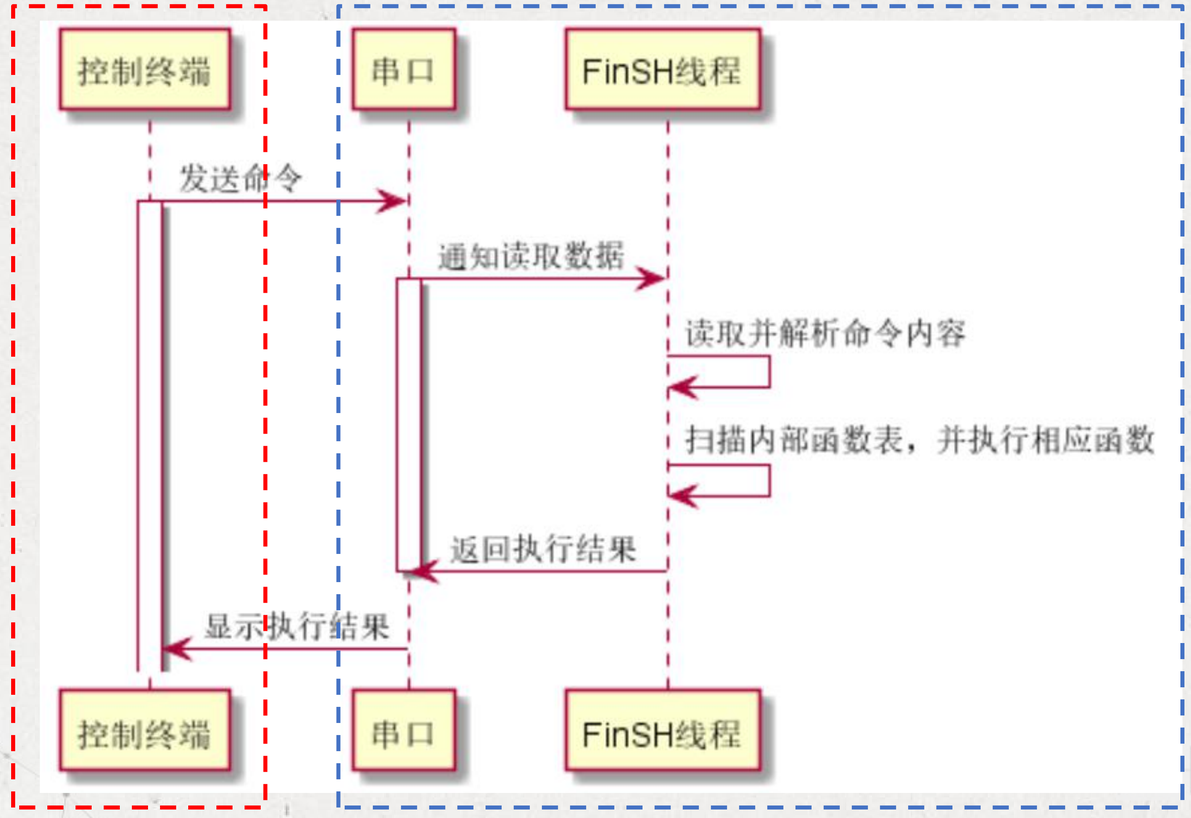


Wujian100端
(需要实现的部分

四、 移植FinSH组件

- 用户在控制终端输入命令，控制终端通过串口将命令传给设备里的 FinSH，FinSH 会读取设备输入命令，解析并自动扫描内部函数表，寻找对应函数名，执行函数后输出回应，回应通过原路返回，将结果显示在控制终端上。

PC端



Wujian100端
(需要实现的部分

四、 移植FinSH组件

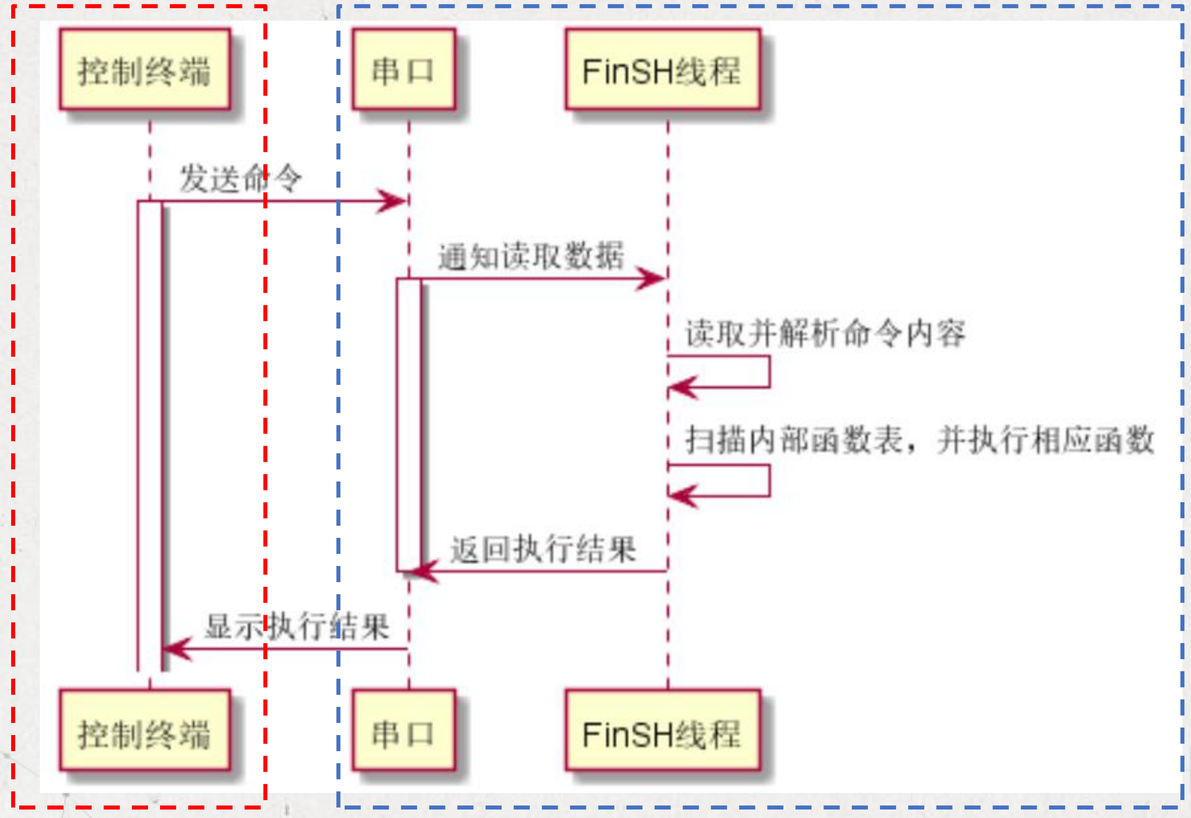
移植Finsh组件的步骤

添加FinSH源码至工程

实现串口接收和发送函数与FinSH组件的对接

修改ld链接脚本

在rtconfig.h 配置文件中使能FinSH组件



PC端

Wujian100端
(需要实现的部分

四、 移植FinSH组件

①实现控制台终端打印

```
RT_WEAK void rt_hw_console_output(const char *str)
{
    int i,size;
     size=rt_strlen (str);
    for (i = 0; i < size; i++)
    {
        if (*(str + i) == '\n')
        {
            csi_usart_putchar(console_handle, '\r');
        }
        csi_usart_putchar(console_handle, *(str+i));
    }
}
RTM_EXPORT(rt_hw_console_output);
```

实现 rt_hw_console_output()函数

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.3 build Oct 23 2019
 2006 - 2018 Copyright by rt-thread team
Hello RT-Thread
Hello RT-Thread
Hello RT-Thread
```

完成该函数后，可以实现控制台字符输出

- 对接**rt_hw_console_output()** 函数，注意：RT-Thread 系统中已有的打印均以 \n 结尾，而并非 \r\n，所以在字符输出时，需要在输出 \n 之前输出 \r，完成回车与换行，否则系统打印出来的信息将只有换行。

四、 移植FinSH组件

②实现命令输入

- 实现命令输入需要将串口对接rt_hw_console_getchar()：获取一个字符，即在该函数中实现uart 获取字符，可以使用中断方式或查询方式获取（注意不要死等，在未获取到字符时，需要让出 CPU。

```
char rt_hw_console_getchar(void)
{
    if(buff_addr==0)
    {
        buff_i=0;
        rt_thread_mdelay(10);
    }
    else if(buff_i<=buff_addr)
    {
        buff_i++;
        return buff[buff_i-1];

    }
    else
    {
        buff_addr=0;
        rt_thread_mdelay(10);
    }
}
```

查询方式实现

rt_hw_console_getchar()函数

四、 移植FinSH组件

③修改链接脚本

- 进行了上面两步操作程序仍不能正常使用 FinSH 组件，是因为 RT-Thread 组件的初始化函数都放在特殊的代码段中，默认的编译配置可能不会被链接器显式调用的函数。 需要修改链接文件保证链接时保持特定代码段不被链接器优化，因此需要在链接脚本的.text 代码段中添加一段代码。

```
/* section information for finsh shell */
. = ALIGN(4);
__fsymtab_start = .;
KEEP(*(FSymTab))
__fsymtab_end = .;
. = ALIGN(4);
__vsymtab_start = .;
KEEP(*(VSymTab))
__vsymtab_end = .;
. = ALIGN(4);

/* section information for initial. */
. = ALIGN(4);
__rt_init_start = .;
KEEP(*(SORT(.rti_fn*)))
__rt_init_end = .;
. = ALIGN(4);

/* section information for modules */
. = ALIGN(4);
__rtmsymtab_start = .;
KEEP(*(RTMSymTab))
__rtmsymtab_end = .;
```

在board/wujian100_open_evb/gcc_csky.ld链接脚本
.text段中添加以上代码

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.3 build Oct 23 2019
 2006 - 2018 Copyright by rt-thread team
msh >
msh >help
RT-Thread shell commands:
version list_thread list_sem list_event list_timer help ps free
msh >
msh >ps
thread pri  status      sp     stack size max used left tick  error
------ ---  ------- ---------- ----------  ------  ---------- ---
tshell   5  ready   0x00000040 0x00001000    06%   0x00000001 000
tidle   31  ready   0x00000040 0x00000100    25%   0x00000020 000
main    10  ready   0x00000078 0x00000400    18%   0x00000013 000
msh >
msh >
```

再次编译下载，确认初始化 FinSH 组件成功
输入命令后，FinSH组件会执行对应操作

FinSH组件命令

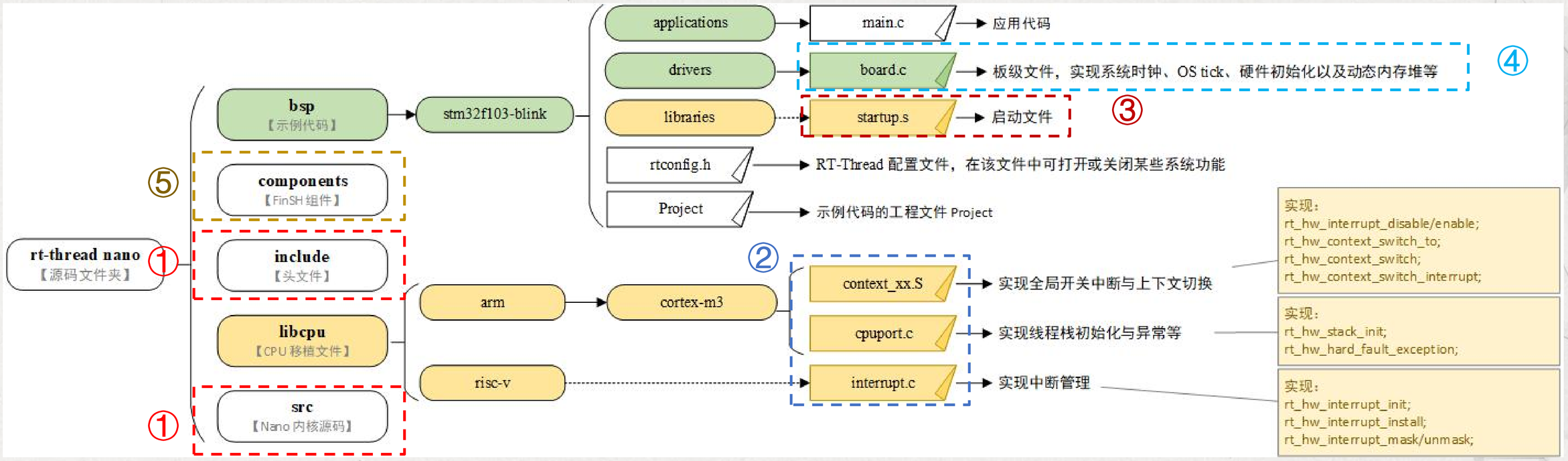
- FinSH命令行组件在移植成功后，可以输入一系列的命令来执行对应的操作。输入help命令后可以打印查看当前系统支持的所有命令，默认支持命令的数量与用到的RT-Thread组件有关。

```
RT-Thread shell commands:
version          - show RT-Thread version information
list_thread      - list thread
list_sem         - list semaphore in system
list_event       - list event in system
list_mutex       - list mutex in system
list_mailbox     - list mail box in system
list_msgqueue    - list message queue in system
list_timer       - list timer in system
list_device      - list device in system
exit             - return to RT-Thread shell mode.
help             - RT-Thread shell help.
ps               - List threads in the system.
time             - Execute command with time.
free             - Show the memory usage in the system.
```

所有显示 RT-Thread 内核状态信息的命令

- 除了FinSH内置命令外，FinSH 还提供了多个宏接口来导出自定义命令，导出的命令可以直接在 FinSH 中执行。

总结



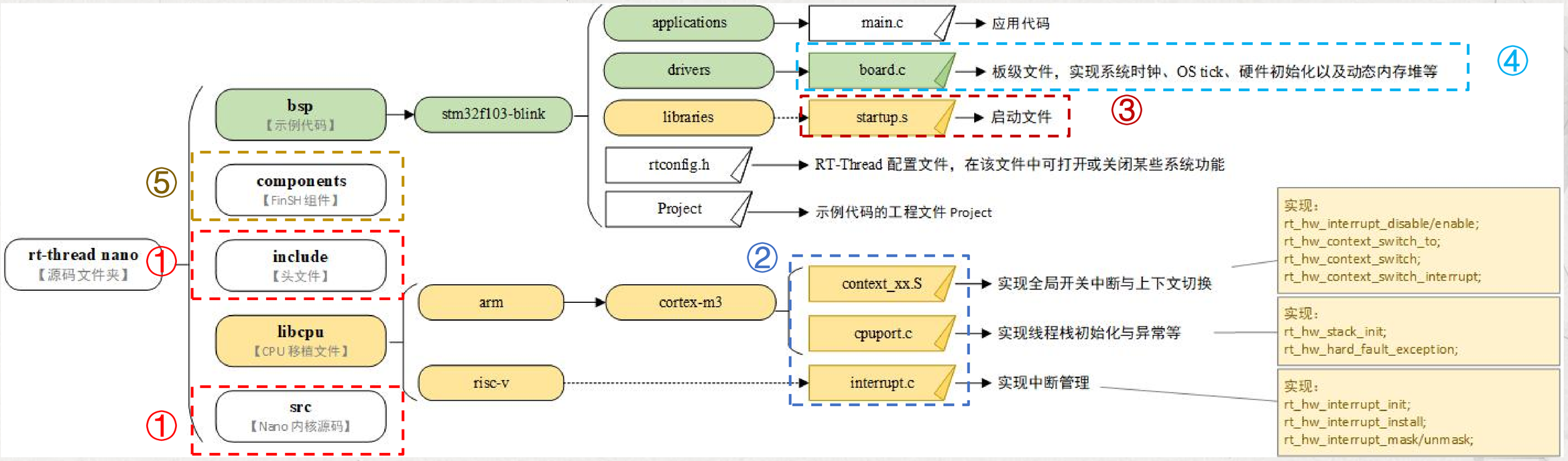
移植需要完成的操作：

①添加**RT-Thread**源码。

②**CPU**架构的移植：汇编实现**libcpu**层、修改中断服务程序

③修改启动文件，实现**RT-Thread**启动

总结



④板级移植：实现OS节拍并在rt_config.h中配置每秒的节拍个数

⑤添加**FinSH**组件：添加**FinSH**源码、实现串口相关函数的对接、修改链接脚本、修改**rtconfig.h**配置文件使能FinSH组件等