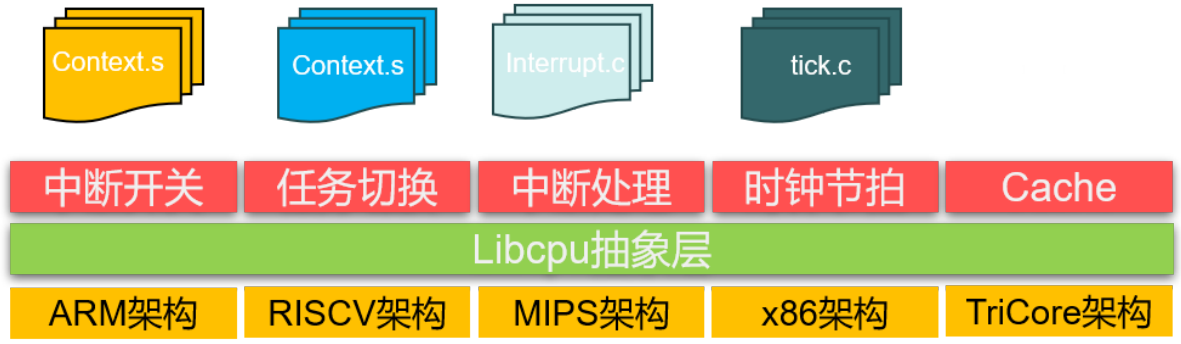
kendryte K210移植RT-Thread OS

总论

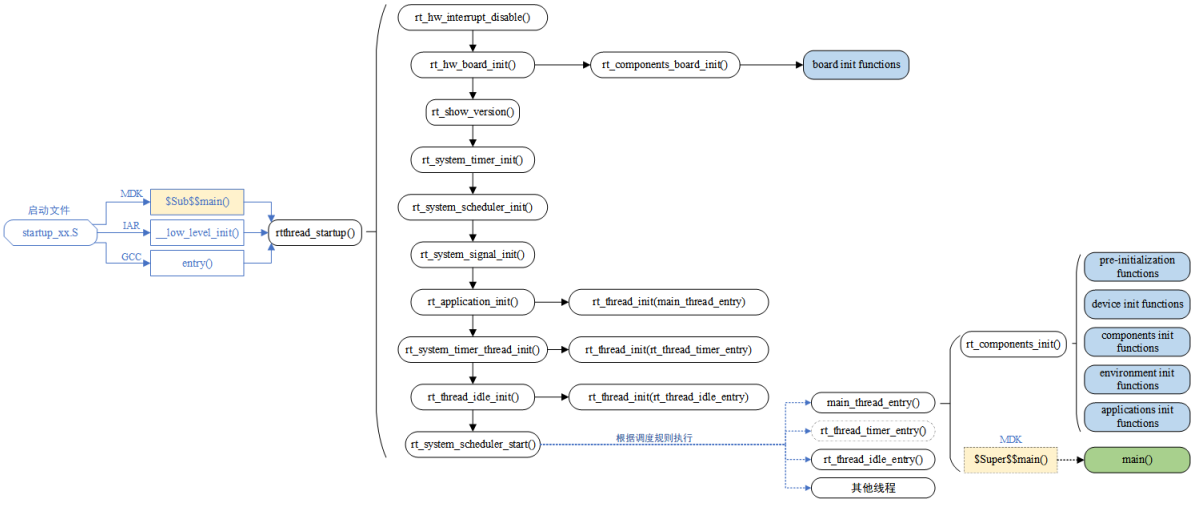
在嵌入式领域有多种不同 CPU 架构，例如 Cortex-M、ARM920T、MIPS32、RISC-V 等等。为了使 RT-Thread 能够在不同 CPU 架构的芯片上运行，RT-Thread 提供了一个 libcpu 抽象层来适配不同的 CPU 架构。libcpu 层向上对内核提供统一的接口，包括全局中断的开关，线程栈的初始化，上下文切换等。

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。[1]

对于RT-Thread OS，芯片移植工作可以归纳为以下四个方面的对接：

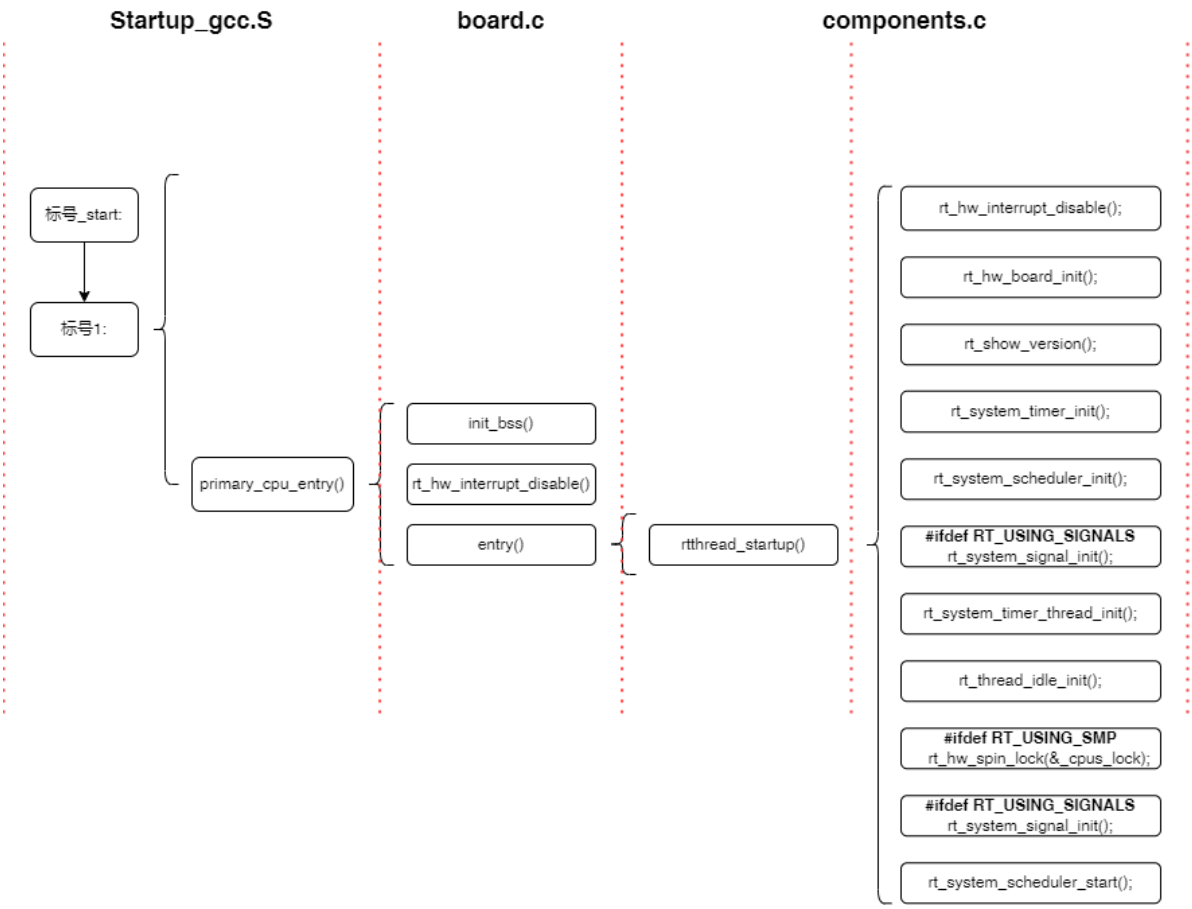


记住上面的四项内容，这部分中每项工作都包括**汇编和C语言**两部分。从程序构成角度则分为**初始化和接口调用**两部分。此外，对于RT-Thread我们最好还要初始化一组UART供调试使用（msh）。我们接着来看RT-Thread OS的启动流程，如下图Fig.1所示[1]：



我们假设当前选择使用GCC方式编译，那么进入rtthread_startup()的函数为entry()。

以K210为例，在从startup_gcc.S文件进入到entry()前，调用了init_bss()函数，对bss段的内容进行了初始化，然后关闭了中断，最后才进入到entry()，如下图所示：

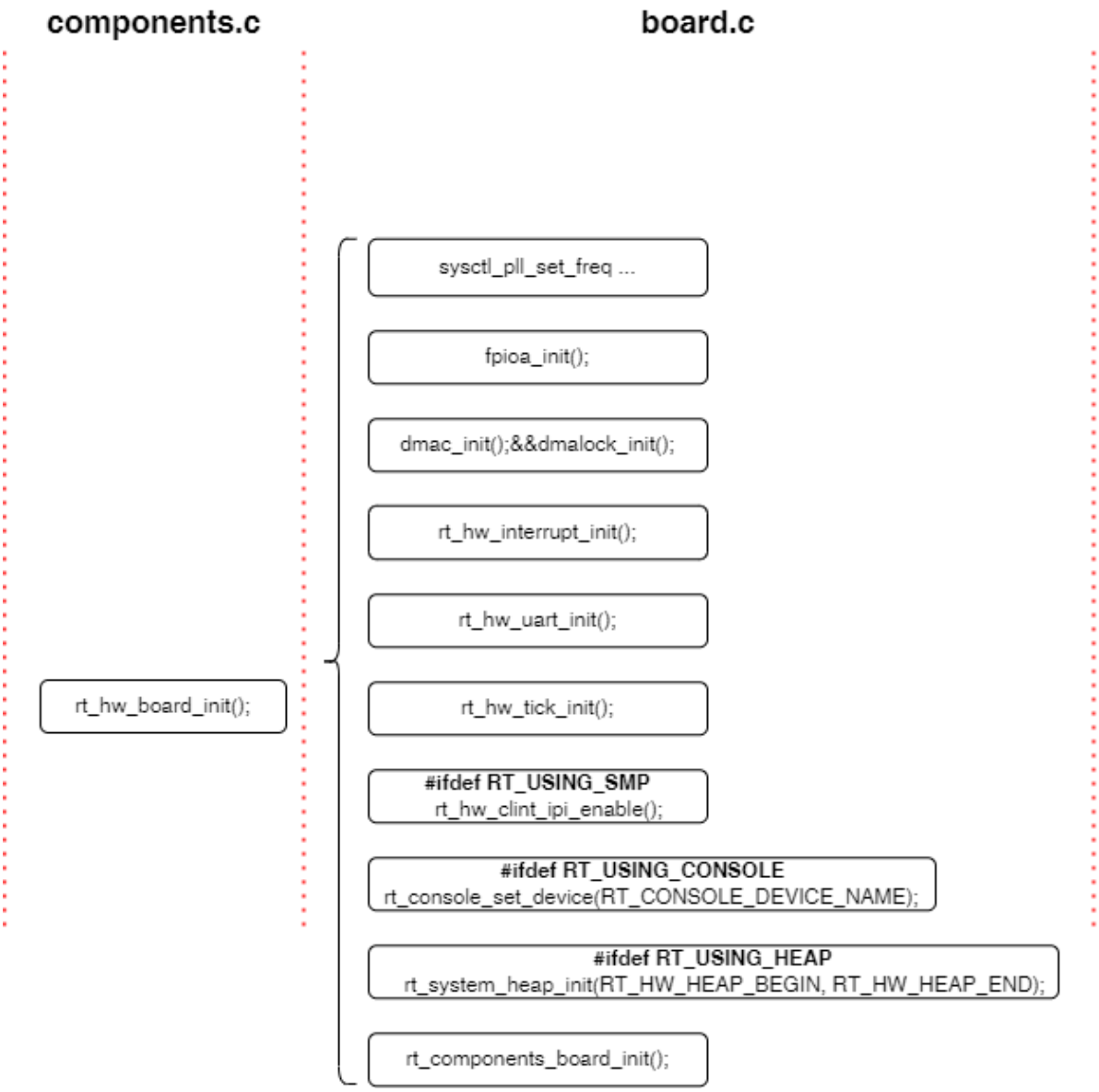



```
int rtthread_startup(void)
{
    //全局中断关闭
    rt_hw_interrupt_disable();

    //板级硬件初始化
    /* board level initialization
     * NOTE: please initialize heap inside board initialization.
     */
    rt_hw_board_init();

    //显示版本号
    /* show RT-Thread version */
    rt_show_version();

    //系统定时器初始化
    /* timer system initialization */
    rt_system_timer_init();

    //调度器初始化
    /* scheduler system initialization */
    rt_system_scheduler_init();

    //信号初始化，用于软中断
#ifdef RT_USING_SIGNALS
    /* signal system initialization */
    rt_system_signal_init();
#endif /* RT_USING_SIGNALS */

    //main线程初始化
```

```
    /* create init_thread */
    rt_application_init();

    //计时器线程初始化
    /* timer thread initialization */
    rt_system_timer_thread_init();

    //空闲线程初始化
    /* idle thread initialization */
    rt_thread_idle_init();

    //多核自旋锁初始化
#ifdef RT_USING_SMP
    rt_hw_spin_lock(&_cpus_lock);
#endif /* RT_USING_SMP */

    //开启任务调度工作
    /* start scheduler */
    rt_system_scheduler_start();

    /* never reach here */
    return 0;
}
```

上图为实际K210的初始化流程，从这张图我们可以看到，上电后芯片的起始工作为startup_XXX.s，这一部分主要是初始化相关的寄存器和堆栈空间，设置中断入口函数等工作。紧接着跳转到`primary_cpu_entry()`函数，在这部分首先初始化了bss段，然后关闭全局中断，并跳转到`entry()`，该函数中仅有一个函数`rtthread_startup()`，从这一部分开始，RT-Thread操作系统进入到正式的初始化过程。这里我们看到，官方在进入到`rtthread_startup()`前，多了一个`rt_hw_interrupt_disable()`函数，此外，对于init_bss段初始化，也建议从board.c迁移到libcpu抽象层的文件中。

进入到`rtthread_startup()`工作，我们在关闭全局中断后，首先是调用`rt_hw_board_init()`函数，我们首先了解一下这部分都进行了哪些初始化工作，该代码如下：




```
void rt_hw_board_init(void)
{
    //设置系统时钟
    sysctl_pll_set_freq(SYSCTL_PLL0, 800000000UL);
    sysctl_pll_set_freq(SYSCTL_PLL1, 400000000UL);
    sysctl_pll_set_freq(SYSCTL_PLL2, 45158400UL);
    sysctl_clock_set_threshold(SYSCTL_THRESHOLD_APB1, 2);

    //现场可编程 IO 阵列初始化，该部分可以实现允许用户将 255 个内部功能映射到芯片外围的 48 个自由 IO 上，即GPIO初始化
    /* Init FPIOA */
    fpioa_init();

    //直接内存存取控制器初始化
    /* Dmac init */
    dmac_init();
    dmalock_init();

    //初始化中断处理函数
    /* initalize interrupt */
    rt_hw_interrupt_init();
    /* initialize hardware interrupt */
```

```
    //初始化UART
    rt_hw_uart_init();

    //初始化时钟节拍
    rt_hw_tick_init();

    //初始化SMP功能
#ifdef RT_USING_SMP
    rt_hw_clint_ipi_enable();
#endif

    //控制台交互输出初始化
#ifdef RT_USING_CONSOLE
    /* set console device */
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif /* RT_USING_CONSOLE */

    //内存堆管理初始化
#ifdef RT_USING_HEAP
    rt_kprintf("heap: [0x%08x - 0x%08x]\n", (rt_ubase_t) RT_HW_HEAP_BEGIN,
(rt_ubase_t) RT_HW_HEAP_END);
    /* initialize memory system */
    rt_system_heap_init(RT_HW_HEAP_BEGIN, RT_HW_HEAP_END);
#endif

    //on-board peripherals级别功能函数自动初始化
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif
}
```

我们发现对于操作系统移植，最重要的函数为以下几个：

`rt_hw_interrupt_init()`

`rt_hw_tick_init()`

对于SMP，则涉及

`rt_hw_clint_ipi_enable()`

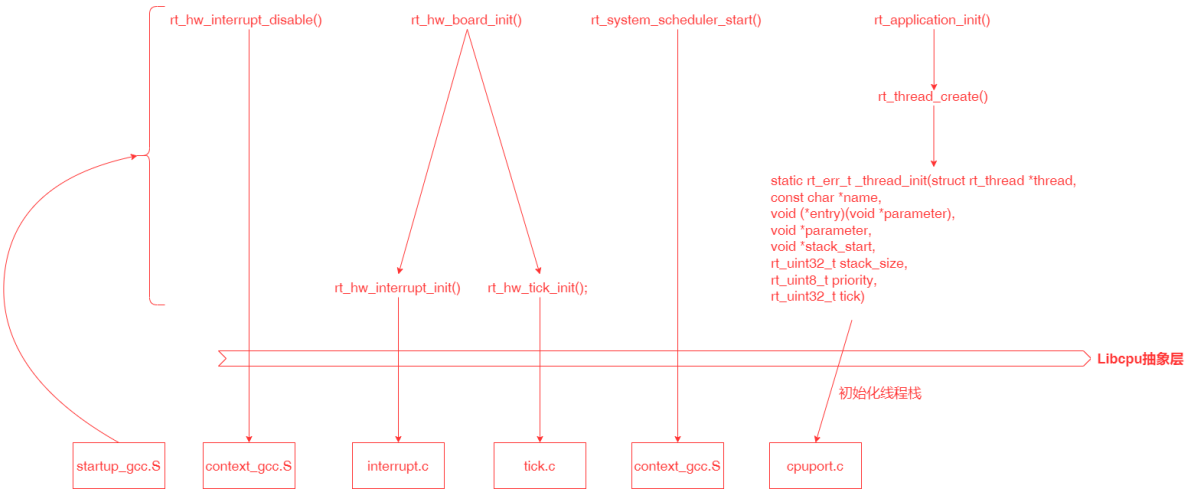
总结一下，到现在为止，我们已经涉及了以下几个移植工作了：

- startup_XXX.S启动文件
- 全局中断开关初始化和函数接口
- 中断处理初始化和函数接口
- 时钟节拍初始化和函数接口（系统时钟初始化）

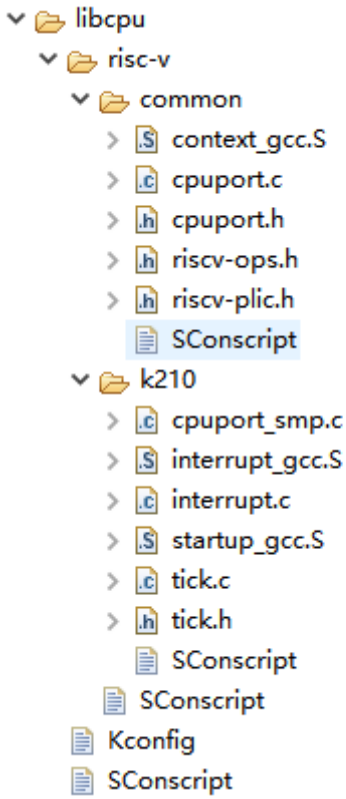
貌似到现在为止还差一个上下文切换初始化和函数接口，我们跳出`rt_hw_board_init()`函数往下看，有一个`rt_system_scheduler_start()`函数，这一部分就是在调用上下文切换功能。到此，我们再加上这一部分内容。

- 上下文切换函数调用

那么，整个操作系统就可以运行起来了。我们再用一张图显示一下移植相关的libcpu工作对我们操作系统运行起到了哪些支撑作用。



我们再看一下lipcpu抽象层中的文件列表，这里不讨论SMP：



riscv-ops.h和riscv-plic.h文件作用设置了一些常用变量，实际系统中并未使用。

参考文献

[1]. RT-Thread 官方文档-内核移植