



RISC-V 平台移植 RTOS

ARM 上移植实时操作系统大家可能比较熟悉,且例程较多,对于 RISC-V 内核的 MCU,可能相对比较陌生。下面结合 WCH 的 CH32V103 和 CH32V307 两款芯片来详细说下针对 RISC-V 平台,移植实时操作系统的注意点。

在移植前,有必要对 RISC-V 的一些基本知识点有一定的了解,相关知识可以参考下面几个文档,这里不展开讨论,仅结合 V103 和 V307 针对用到的 CSR 寄存器、特权模式、汇编指令等仅作简述。

- | | |
|---|---|
|  riscv-spec-v2.1 .pdf |  riscv-privilege d.pdf |
| 1.RISC-V SPEC: | 2.RISC-V 特权手册: |
|  RISC-V-Reader-C hinese-v2p1.pdf | |
| 3.翻译版手册: | |

之所以选择 V103 和 V307 两个芯片主要是其极具代表性:首先,直观上其外设的使用方法和我们之前熟悉的 F103, F107 等是兼容的,这样降低了我们使用和移植时的难度,基于 WCH 提供的外设库,我们以前上层的代码甚至于不用修改可直接使用。其次, V103 是 WCH RISC-V 青稞内核家族中的 V3A 内核, V307 为 V4F 内核, V3 内核支持 RV32IMAC 指令集,即除支持 RISC-V 基本的 32 位整数指令集外,还支持硬件乘除法,原子指令,压缩指令。V4F 在 V3A 的基础上增加了单精度硬件浮点,并且其性能也比 V3A 高。除此所有的 V4 内核还支持自定义压缩指令-XW 扩展,包括以下指令 c.lbu/c.lhu/c.sb/c.sh/c.lbusp/c.lhusp/c.sbsp/c.shsp。

除上述之外,虽然两者的中断控制器(PFIC)相较于现行的 PLIC 均不同,均不是统一入口,而是采用中断向量表寻址的方式,均可配置 4 条免表中断(VTF),但是 V3A 的中断向量表处存放是一条指令,而 V4F 的向量表既可以存放跳转指令,也可以存放中断处理函数的地址。两者均支持中断嵌套和硬件压栈,区别在于 V3A 最大嵌套两级, V4F 最大可达八级,同时 V3A 的硬件压栈深度两级, V4F 的硬件压栈深度为三级(更多详细内容参考手册中断章节)。

这里需要注意的是一般情况下,移植 RTOS 的时需要关闭硬件压栈,因为在切换任务时,我们希望自己控制出栈入栈的内容。但是针对 V4F 核, CSR 0x804 中有增加控制位(bit5)GIHWSTKNEN(全局中断和硬件压栈关闭使能),可以进中断时置位该位,关闭中断和硬件压栈,我们手动保存当前线程栈,恢复新线程栈,中断 mret 返回后,硬件自动清除该位,恢复中断和硬件压栈使能。这样即可保证 RTOS 下,硬件压栈可正常使用,保证 RTOS 下的中断响应速度。

RISC-V 寄存器如下图 1 所示,其中 x0-x31 为整形寄存器, f0-f31 为浮点寄存器(V3 没

有浮点寄存器)。所有带 caller 的寄存器，当发生中断时需要保存，值得注意的是，WCH 的硬

| 寄存器 | 调用名字 | 用途 | 存储者 |
|---------|----------|---------------------------|--------|
| x0 | zero | 常数0 | N.A. |
| x1 | ra | 返回地址 | Caller |
| x2 | sp | 栈指针 | Callee |
| x3 | gp | 全局指针 | / |
| x4 | tp | 线程指针 | / |
| x5-x7 | t0-t2 | 临时存储 | Caller |
| x8 | s0/fp | 保存寄存器/帧指针 (配合栈指针界定一个函数的栈) | Callee |
| x9 | s1 | 保存寄存器 | Callee |
| x10-x11 | a0-a1 | 函数参数/返回值 | Caller |
| x12-x17 | a2-a7 | 函数参数 | Caller |
| x18-x27 | s2-s11 | 保存寄存器 | Callee |
| x28-x31 | t3-t6 | 临时存储 | Caller |
| f0-f7 | ft0-ft7 | 浮点临时存储 | Caller |
| f8-f9 | fs0-fs1 | 浮点保存寄存器 | Callee |
| f10-f11 | fa0-fa1 | 浮点函数参数/返回值 | Caller |
| f12-f17 | fa2-fa7 | 浮点函数参数 | Caller |
| f18-f27 | fs2-fs11 | 浮点保存寄存器 | Callee |
| f28-f31 | ft8-ft11 | 浮点临时存储 | Caller |

图 1 RISC-V 寄存器

件压栈保存的寄存器仅仅保存整数的 16 个 caller saved 寄存器。正常一个中断函数的寄存器保存我们不用关心，编译器会帮我们做的很好。但是当我们从一个汇编入口进中断函数的时候这些过程就不得不由我们自己来实现。寄存器中几个相对特殊的 x0 恒为 0，x1 是返回地址寄存器 ra，函数调用时用来存放返回地址，x2 为堆栈指针 sp，x3 为 gp 全局指针，用来寻址全局变量。

除了上述的寄存器，移植还要关心的是几个 CSR 寄存器 mstatus，mepc。正常情况下大部分 CSR 只能在机器模式下操作（WCH 的 v3 和 v4 内核支持机器模式和用户模式）。mstatus 中，MIE 为中断使能，当进中断时 MPIE 更新为 MIE，返回时 MIE 更新为 MPIE。MPP 用于保存进中断之前的特权模式，如果我们设置其为 MPP=0b11，那么将一直处于机器模式，其 mret 返回后还是处于机器模式。mepc 是机器模式下异常程序指针，其只会在发生异常时被更新（中断也是一类异常），进异常时我们可以从另外两个 CSR 寄存器 mcause 来看引起异常原因通过 mtval 查看引起异常时的值。当从异常返回时 mepc 的值被更新给 pc。我们正是通过进中断修改 mepc 来实现任务的切换的，后面会详细说明这个过程。

实时操作系统大家应该不陌生，常见的 uCOS，FreeRTOS，RT-Thread，LiteOS_M，TencentOS_Tiny 等等，其基本的思路都是一样的，需要一个定时器，为系统提供滴答时钟。还需要一个切换上下文的环境，例如 ARM 中常见的 PendSV 中断。V3/V4 内核自带一个 SysTick 64 位的定时器，可以为系统提供时钟，且提供软件中断可以为任务切换提供环境（其实任务切换并不一定要放在某个中断中，只要做好原任务的保存，新任务的加载即可）。

从一个 RISC-V 裸机例程移植一个 RTOS，个人觉得需要弄清楚以下几个问题，那么让一

个实时内核在 MCU 上运行起来应该不难:

(1) RISC-V 中断保存哪些寄存器

前文说过, RISC-V 内核进中断需要保存 caller saved(顾名思义, 调用者需要保存)的寄存器。当不开启硬件浮点时, 编译器会把 16 个寄存器在中断函数开始时存入堆栈, 中断返回前恢复, 如下图 2 和图 3 所示。我们内核支持硬件压栈, 硬件保存和恢复的也正是这 16 个寄存器。使用硬件压栈时需要使能硬件功能, 即硬件压栈使能(不同芯片配置位置不同, 详见手册中断章节), 同时需要通知编译器不自动生成图 2 和图 3 中的软件出入栈的代码, 即在 MRS 声明中断函数时由 `__attribute__((interrupt("WCH-Interrupt-fast")))` 方式定义编译器不自动添加软件出入栈代码, 由 `__attribute__((interrupt()))` 方式定义编译器添加软件出入栈的代码。

```
00001038 <SysTick_Handler>:
1038: 7139          addi sp,sp,-64
103a: de06          sw ra,60(sp)
103c: dc16          sw t0,56(sp)
103e: dala          sw t1,52(sp)
1040: d81e          sw t2,48(sp)
1042: d62a          sw a0,44(sp)
1044: d42e          sw a1,40(sp)
1046: d232          sw a2,36(sp)
1048: d036          sw a3,32(sp)
104a: ce3a          sw a4,28(sp)
104c: cc3e          sw a5,24(sp)
104e: ca42          sw a6,20(sp)
1050: c846          sw a7,16(sp)
1052: c672          sw t3,12(sp)
1054: c476          sw t4,8(sp)
1056: c27a          sw t5,4(sp)
1058: c07e          sw t6,0(sp)
105a: 828a          mv t0,sp
105c: 8201a103     lw sp,-2016(gp) # 20000030 <k_irq_stk_top>
1060: c016          sw t0,0(sp)
1062: e000f7b7     lui a5,0xe000f
```

图 2 整形寄存器入栈

```
109a: 8b1ff0ef     jal ra,94a <tos_knl_irq_leave>
109e: 4102         lw sp,0(sp)
10a0: 50f2         lw ra,60(sp)
10a2: 52e2         lw t0,56(sp)
10a4: 5352         lw t1,52(sp)
10a6: 53c2         lw t2,48(sp)
10a8: 5532         lw a0,44(sp)
10aa: 55a2         lw a1,40(sp)
10ac: 5612         lw a2,36(sp)
10ae: 5682         lw a3,32(sp)
10b0: 4772         lw a4,28(sp)
10b2: 47e2         lw a5,24(sp)
10b4: 4852         lw a6,20(sp)
10b6: 48c2         lw a7,16(sp)
10b8: 4e32         lw t3,12(sp)
10ba: 4ea2         lw t4,8(sp)
10bc: 4f12         lw t5,4(sp)
10be: 4f82         lw t6,0(sp)
10c0: 6121         addi sp,sp,64
10c2: 30200073     mret
```

图 3 整形寄存器出栈

当开启硬件压栈并且编译器中声明使用硬件压栈后, 中断函数汇编代码如下图 4 所示。可见进入中断后直接执行的中断代码, 形如图 2 和图 3 中的 16 个寄存器的入栈和出栈由硬件在中断开始和结束时自动完成。同时也可以看出整个中断函数可以减少 34 条指令。

```
00001038 <SysTick_Handler>:
1038: 828a          mv t0,sp
103a: 8201a103     lw sp,-2016(gp) # 20000030 <k_irq_stk_top>
103e: c016          sw t0,0(sp)
1040: e000f7b7     lui a5,0xe000f
1044: 0007a023     sw zero,0(a5) # e000f000 <_eusrstack+0xc000a000>
1048: 00078223     sb zero,4(a5)
104c: 000782a3     sb zero,5(a5)
1050: 00078323     sb zero,6(a5)
1054: 000783a3     sb zero,7(a5)
1058: 00078423     sb zero,8(a5)
105c: 000784a3     sb zero,9(a5)
1060: 00078523     sb zero,10(a5)
1064: 000785a3     sb zero,11(a5)
1068: 4705         li a4,1
106a: c398          sw a4,0(a5)
106c: 957ff0ef     jal ra,9c2 <tos_knl_is_running>
1070: c511         beqz a0,107c <SysTick_Handler+0x44>
1072: 8bbff0ef     jal ra,92c <tos_knl_irq_enter>
1076: 35d5         jal f5a <tos_tick_handler>
1078: 8d3ff0ef     jal ra,94a <tos_knl_irq_leave>
107c: 4102         lw sp,0(sp)
107e: 30200073     mret
```

图 4 开启硬件压栈后中断函数汇编代码

由此也可知道前文说的一般中断切换上下文时不开启硬件压栈的原因: 开启后中断返回时硬件会复写 16 个 caller saved 寄存器。

当开启硬件浮点时，除了上述 16 个整形还会增加 20 个浮点寄存器，如下图 5 所示。由此也可以看出，硬件压栈只对整形的寄存器生效。

```

00001038 <SysTick_Handler>:
1038: 715d          addi    sp,sp,-80
103a: e682          fsw    ft0,76(sp)
103c: e486          fsw    ft1,72(sp)
103e: e28a          fsw    ft2,68(sp)
1040: e08e          fsw    ft3,64(sp)
1042: fe12          fsw    ft4,60(sp)
1044: fc16          fsw    ft5,56(sp)
1046: fa1a          fsw    ft6,52(sp)
1048: f81e          fsw    ft7,48(sp)
104a: f62a          fsw    fa0,44(sp)
104c: f42e          fsw    fa1,40(sp)
104e: f232          fsw    fa2,36(sp)
1050: f036          fsw    fa3,32(sp)
1052: ee3a          fsw    fa4,28(sp)
1054: ec3e          fsw    fa5,24(sp)
1056: ea42          fsw    fa6,20(sp)
1058: e846          fsw    fa7,16(sp)
105a: e672          fsw    ft8,12(sp)
105c: e476          fsw    ft9,8(sp)
105e: e27a          fsw    ft10,4(sp)
1060: e07e          fsw    ft11,0(sp)
1062: 828a          mv     t0,sp
1064: 8201a103     lw     sp,-2016(gp) # 2000
1068: c016          sw     t0,0(sp)
106a: e000f7b7     lui   a5,0xe000f

10a2: 8a9ff0ef     jal   ra,94a <tos_knl_irq_leave>
10a6: 4102          lw     sp,0(sp)
10a8: 6036          flw    ft0,76(sp)
10aa: 60a6          flw    ft1,72(sp)
10ac: 6116          flw    ft2,68(sp)
10ae: 6186          flw    ft3,64(sp)
10b0: 7272          flw    ft4,60(sp)
10b2: 72e2          flw    ft5,56(sp)
10b4: 7352          flw    ft6,52(sp)
10b6: 73c2          flw    ft7,48(sp)
10b8: 7532          flw    fa0,44(sp)
10ba: 75a2          flw    fa1,40(sp)
10bc: 7612          flw    fa2,36(sp)
10be: 7682          flw    fa3,32(sp)
10c0: 6772          flw    fa4,28(sp)
10c2: 67e2          flw    fa5,24(sp)
10c4: 6852          flw    fa6,20(sp)
10c6: 68c2          flw    fa7,16(sp)
10c8: 6a32          flw    ft8,12(sp)
10ca: 6ea2          flw    ft9,8(sp)
10cc: 6f12          flw    ft10,4(sp)
10ce: 6f82          flw    ft11,0(sp)
10d0: 6161          addi   sp,sp,80
10d2: 30200073     mret

```

图 5 浮点寄存器出入栈

(2) 任务栈保存哪些内容

图 1 中列举了所有的寄存器，当需要切换任务时刻的寄存器值，除 x0 恒为 0，其他的寄存器无法预知其值，切换时均需要保存（gp 寄存器编译好后，固定不变，理论上可以不操作，为保持一致性和完整性，一并保存），如果使用浮点，还应该包括浮点寄存器。每个 RTOS 均会定义一个和上下文保存相关的结构体，以 rt-thread 为例，可以看到如下图 6 的数据结构定义。

```

/* float register */
#ifdef ARCH_RISCV_FPU
rv_floatreg_t f0; /* f0 */
rv_floatreg_t f1; /* f1 */
rv_floatreg_t f2; /* f2 */
rv_floatreg_t f3; /* f3 */
rv_floatreg_t f4; /* f4 */
rv_floatreg_t f5; /* f5 */
rv_floatreg_t f6; /* f6 */
rv_floatreg_t f7; /* f7 */
rv_floatreg_t f8; /* f8 */
rv_floatreg_t f9; /* f9 */
rv_floatreg_t f10; /* f10 */
rv_floatreg_t f11; /* f11 */
rv_floatreg_t f12; /* f12 */
rv_floatreg_t f13; /* f13 */
rv_floatreg_t f14; /* f14 */
rv_floatreg_t f15; /* f15 */
rv_floatreg_t f16; /* f16 */
rv_floatreg_t f17; /* f17 */
rv_floatreg_t f18; /* f18 */
rv_floatreg_t f19; /* f19 */
rv_floatreg_t f20; /* f20 */
rv_floatreg_t f21; /* f21 */
rv_floatreg_t f22; /* f22 */
rv_floatreg_t f23; /* f23 */
rv_floatreg_t f24; /* f24 */
rv_floatreg_t f25; /* f25 */
rv_floatreg_t f26; /* f26 */
rv_floatreg_t f27; /* f27 */
rv_floatreg_t f28; /* f28 */
rv_floatreg_t f29; /* f29 */
rv_floatreg_t f30; /* f30 */
rv_floatreg_t f31; /* f31 */
#endif

/* float register */
#ifdef ARCH_RISCV_FPU
};

```

图 6 上下文保存结构体

可以看到除了通用寄存器外，还有两个前文提到的成员 mepc、mstatus，其中 mstatus 中含有中断的使能控制位，而 mepc 为机器模式下异常程序指针寄存器，其值会在执行 mret 后更新给 pc，我们正式通过设置该寄存器的值来控制程序运行的切换。

当我们新建一个线程，初始化线程时，会为其开辟一个线程栈（程序中通常设置一个数

组), 即对上述结构体做初始化, 在 `rt-thread` 中的代码如下图 7 所示。

```
static rt_err_t _rt_thread_init(struct rt_thread *thread,
                               const char *name,
                               void (*entry)(void *parameter),
                               void *parameter,
                               void *stack_start,
                               rt_uint32_t stack_size,
                               rt_uint8_t priority,
                               rt_uint32_t tick)
{
    /* init thread list */
    rt_list_init(&(thread->tlist));

    thread->entry = (void *)entry;
    thread->parameter = parameter;

    /* stack init */
    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /* init thread stack */
    rt_memset(thread->stack_addr, '#', thread->stack_size);
#ifdef ARCH_CPU_STACK_GROWS_UPWARD
    thread->sp = (void *)rt_hw_stack_init(thread->entry, thread->parameter,
                                          (void *)((char *)thread->stack_addr),
                                          (void *)_rt_thread_exit);
#else
    thread->sp = (void *)rt_hw_stack_init(thread->entry, thread->parameter,
                                          (rt_uint8_t *)((char *)thread->stack_addr + thread->stack_size - sizeof(rt_ubase_t)),
                                          (void *)_rt_thread_exit);
#endif /* ARCH_CPU_STACK_GROWS_UPWARD */
}

rt_uint8_t *rt_hw_stack_init(void *tentry,
                             void *parameter,
                             rt_uint8_t *stack_addr,
                             void *texit)
{
    struct rt_hw_stack_frame *frame;
    rt_uint8_t *stk;
    int i;

    stk = stack_addr + sizeof(rt_ubase_t);
    stk = (rt_uint8_t *)RT_ALIGN_DOWN((rt_ubase_t)stk, REGBYTES);
    stk -= sizeof(struct rt_hw_stack_frame);

    frame = (struct rt_hw_stack_frame *)stk;

    for (i = 0; i < sizeof(struct rt_hw_stack_frame) / sizeof(rt_ubase_t); i++)
    {
        ((rt_ubase_t *)frame)[i] = 0xdeadbeef;
    }

    frame->ra = (rt_ubase_t)texit;
    frame->a0 = (rt_ubase_t)parameter;
    frame->epc = (rt_ubase_t)tentry;

    /* force to machine mode(MPP=11) and set MPIE to 1 and FS=11 */
    frame->mstatus = 0x00007880;
    return stk;
}
```

图 7 线程堆栈初始化

由程序可知, 堆栈初始化在线程初始化中被调用, 线程初始化程序中首先将整个堆栈空间设成“#”, 然后根据堆栈的增长方向设置不同参数, 以红框中的向下增长为例, 将线程的入口位置, 线程可以带一个参数, 返回地址, 堆栈顶部地址。从堆栈初始化程序 `*rt_hw_stack_init` 中可以看出, 其先将堆栈顶部地址对齐, 然后向下偏移一个 `rt_hw_stack_frame` 结构体的大小, 用于存储图 6 中需要存储的寄存器, 并对该部分空间进行了初始化。其中把线程的入口地址给了 `mepc`, 线程输入参数给 `a0`, `mstatus` 初始值 (MPP、MPIE、FS、MIE), 即强制机器模式, 使能浮点, MPIE 为 1, MIE 为 0。如果不带硬件浮点, 可将该值设置为 `0x1880`。另外设置 `ra` 为线程的返回地址, 一般情况下一个线程我们希望一直运行的, 当需要返回时说明该线程不再需要运行, 所以返回地址一般是一段将该线程从线

程列表中删除并切换至下一个线程的一段程序，即图 7 红框的中调用的函数 `_rt_thread_exit`。

初始化线程时会定义一个 `rt_thread` 结构的全局变量，线程的操作即依靠该结构体。其内部内容如下图 8 所示，其内部可以看到一个 `sp` 成员，初始化好的堆栈指针即传给该成员。

```
struct rt_thread
{
    /* rt object */
    char name[RT_NAME_MAX];          /*< the name of thread */
    rt_uint8_t type;                 /*< type of object */
    rt_uint8_t flags;                /*< thread's flags */

#ifdef RT_USING_MODULE
    void *module_id;                 /*< id of application modul
#endif

    rt_list_t list;                  /*< the object list */
    rt_list_t tlist;                 /*< the thread list */

    /* stack point and entry */
    void *sp;                         /*< stack point */
    void *entry;                       /*< entry */
    void *parameter;                  /*< parameter */
    void *stack_addr;                 /*< stack address */
    rt_uint32_t stack_size;           /*< stack size */

    /* error code */
    rt_err_t error;                   /*< error code */

    rt_uint8_t stat;                  /*< thread status */

#ifdef RT_USING_SMP
    rt_uint8_t bind_cpu;              /*< thread is bind to cpu */
    rt_uint8_t oncpu;                 /*< process on cpu */

    rt_uint16_t scheduler_lock_nest;  /*< scheduler lock count */
    rt_uint16_t cpus_lock_nest;      /*< cpus lock count */
    rt_uint16_t critical_lock_nest;  /*< critical lock count */
#endif /*RT_USING_SMP*/

    rt_uint8_t current_priority;      /*< current priority */
    rt_uint8_t init_priority;        /*< initialized priority */
#ifdef RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t number;
    rt_uint8_t high_mask;
#endif
    rt_uint32_t number_mask;

#ifdef defined(RT_USING_EVENT)
    /* thread event */
    rt_uint32_t event_set;
    rt_uint8_t event_info;
#endif

#ifdef defined(RT_USING_SIGNALS)
    /* the pending signals */
    rt_sigset_t sig_pending;         /*< the mask bits of signal */
    rt_sigset_t sig_mask;
#endif

#ifdef RT_USING_SMP
    void *sig_ret;                    /*< the return stack pointer from
#endif
    rt_sighandler_t *sig_vectors;    /*< vectors of signal handler */
    void *si_list;                    /*< the signal infor list */
#endif

    rt_ubase_t init_tick;             /*< thread's initialized tick */
    rt_ubase_t remaining_tick;       /*< remaining tick */
#ifdef RT_USING_CPU_USAGE
    rt_uint64_t duration_tick;        /*< cpu usage tick */
#endif
    struct rt_timer thread_timer;     /*< built-in thread timer */
    void (*cleanup)(struct rt_thread *tid); /*< cleanup function when thread
    /* light weight process if present */
#ifdef RT_USING_LWP
    void *lwp;
#endif

    rt_ubase_t user_data;             /*< private user data beyond this t
};
```

图 8 `rt_thread` 结构体详情

综上所述可以看出有每个线程一个 `rt_thread` 结构体，由 `rt_thread->sp` 可获得该线程的堆栈位置，堆栈的栈顶的 `sizeof(rt_hw_stack_frame)` 空间存放了该线程运行需要的 CPU 寄存器值，剩余空间用于该线程运行时变量的出入栈。

以上的内容在其他 RTOS 中也能看到，例如上下文保存结构体 `rt_hw_stack_frame` 在华为鸿蒙 LiteOS_M 中有 `TaskContext`，TencentOS_Tiny 中有 `cpu_context_t`，而线程管理的结构体 `rt_thread`，LiteOS_M 中 `LosTaskCB`，TencentOS_Tiny 中有 `k_task_st` 等。

(3) 切换至第一个任务

`rt-thread` 和其他 RTOS 有点区别的是其 `gcc` 下的入口函数定义为 `entry`，而 `main` 函数则可以被初始化为线程之一，图 9 为 `rt-thread` 的详细的启动流程。`rt-thread` 定义一个 `rt_thread` 类型的全局指针 `rt_current_thread`，用于实时获取当前运行的线程。从图 9 可知，硬件启动后进 `rtthread_startup`，其开始进行了必要的初始化，如系统滴答定时器、堆、串口、调度器、定时器、`main` 线程、空闲 `idle` 线程等等，最后执行了 `rt_system_scheduler_start` 后转交调度器执行。其内容如下图 10 所示。

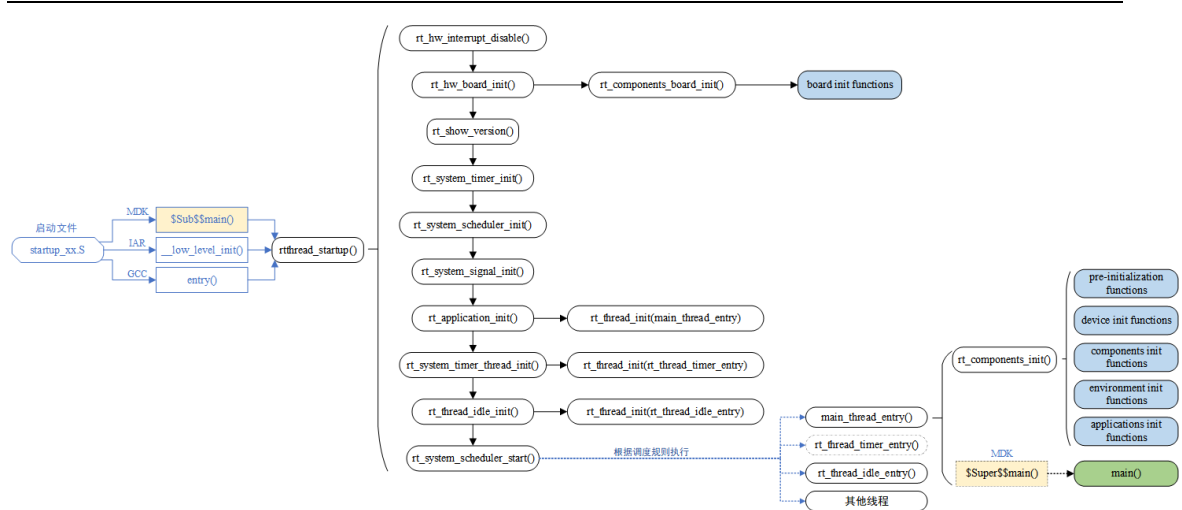


图 9 rt_thread 启动流程

```

void rt_system_scheduler_start(void)
{
    register struct rt_thread *to_thread;
    register rt_ubase_t highest_ready_priority;

    #if RT_THREAD_PRIORITY_MAX > 32
        register rt_ubase_t number;

        number = __rt_ffs(rt_thread_ready_priority_group) - 1;
        highest_ready_priority = (number << 3) + __rt_ffs(rt_thread_ready_table[number]) - 1;
    #else
        highest_ready_priority = __rt_ffs(rt_thread_ready_priority_group) - 1;
    #endif
    /* 找到优先级最高的就绪组的优先级 */

    /* get switch to thread */
    to_thread = rt_list_entry(rt_thread_priority_table[highest_ready_priority].next,
                             struct rt_thread,
                             tlist); /* 从就绪列表中获取最高优先级的线程任务 */
    rt_current_thread = to_thread; /* 并赋给全局指针rt_current_thread */

    /* switch to new thread */
    rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp); /* 转而执行该任务，也是第一个任务 */

    /* never come back */
}

```

图 10 调度器启动

由图 10 可知，其会查找优先级较高的就绪组优先级，并根据该优先级查找就绪链表，获取优先级较高的任务并得到控制块 to_thread，然后调用 rt_hw_context_switch_to 切换至该任务。其是一段汇编实现的代码，传入的参数为该任务的 sp 指针。由前文可知，我们可以根据 to_thread->sp 得到该任务的堆栈位置，该堆栈的顶部空间存储了执行该任务时 cpu 寄存器的值，由此也可推测该部分汇编代码主要完成的就是从 sp 处恢复 cpu 寄存器值，并转而执行该任务。其代码如下图 11 所示，详见注释。

```

rt_hw_context_switch_to:
    la t0, _eusrstack
    addi t0, t0, -512
    csrw mscratch, t0 //裸机下的堆栈区域保存后留给以后中断函数使用

    LOAD sp, (a0) //从传入的参数得到该任务的sp

    LOAD a0, 0 * REGBYTES(sp)
    csrw mepc, a0 //从堆栈恢复mepc, 其在任务初始化的时候初始为任务的入口地址
    LOAD x1, 1 * REGBYTES(sp) //恢复ra, 其在任务初始化时初始为任务的返回函数地址, _rt_thread_exit

    li a0, 0x7800 //无硬件浮点设置为0x1800, 详见mstatus寄存器定义
    csrcs mstatus, a0
    LOAD a0, 2 * REGBYTES(sp)
    csrcs mstatus, a0 //设置mstatus (机器模式、浮点、MPIE位)

    LOAD x4, 4 * REGBYTES(sp) //恢复整形寄存器
    LOAD x5, 5 * REGBYTES(sp)
    LOAD x6, 6 * REGBYTES(sp)
    LOAD x7, 7 * REGBYTES(sp)
    LOAD x8, 8 * REGBYTES(sp)
    LOAD x9, 9 * REGBYTES(sp)
    LOAD x10, 10 * REGBYTES(sp) //恢复a0, 其初始化为任务函数的输入参数
    LOAD x11, 11 * REGBYTES(sp)
    LOAD x12, 12 * REGBYTES(sp)
    LOAD x13, 13 * REGBYTES(sp)
    LOAD x14, 14 * REGBYTES(sp)
    LOAD x15, 15 * REGBYTES(sp)
    LOAD x16, 16 * REGBYTES(sp)
    LOAD x17, 17 * REGBYTES(sp)
    LOAD x18, 18 * REGBYTES(sp)
    LOAD x19, 19 * REGBYTES(sp)
    LOAD x20, 20 * REGBYTES(sp)
    LOAD x21, 21 * REGBYTES(sp)
    LOAD x22, 22 * REGBYTES(sp)
    LOAD x23, 23 * REGBYTES(sp)
    LOAD x24, 24 * REGBYTES(sp)
    LOAD x25, 25 * REGBYTES(sp)
    LOAD x26, 26 * REGBYTES(sp)
    LOAD x27, 27 * REGBYTES(sp)
    LOAD x28, 28 * REGBYTES(sp)
    LOAD x29, 29 * REGBYTES(sp)
    LOAD x30, 30 * REGBYTES(sp)
    LOAD x31, 31 * REGBYTES(sp)
    addi sp, sp, 32 * REGBYTES //sp偏移

#ifdef ARCH_RISCV_FPU
7 FLOAD f0, 0 * FREGBYTES(sp) //如果有硬件浮点, 恢复浮点寄存器
3 FLOAD f1, 1 * FREGBYTES(sp)
3 FLOAD f2, 2 * FREGBYTES(sp)
3 FLOAD f3, 3 * FREGBYTES(sp)
1 FLOAD f4, 4 * FREGBYTES(sp)
2 FLOAD f5, 5 * FREGBYTES(sp)
3 FLOAD f6, 6 * FREGBYTES(sp)
1 FLOAD f7, 7 * FREGBYTES(sp)
5 FLOAD f8, 8 * FREGBYTES(sp)
5 FLOAD f9, 9 * FREGBYTES(sp)
7 FLOAD f10, 10 * FREGBYTES(sp)
3 FLOAD f11, 11 * FREGBYTES(sp)
3 FLOAD f12, 12 * FREGBYTES(sp)
3 FLOAD f13, 13 * FREGBYTES(sp)
2 FLOAD f14, 14 * FREGBYTES(sp)
2 FLOAD f15, 15 * FREGBYTES(sp)
3 FLOAD f16, 16 * FREGBYTES(sp)
1 FLOAD f17, 17 * FREGBYTES(sp)
5 FLOAD f18, 18 * FREGBYTES(sp)
5 FLOAD f19, 19 * FREGBYTES(sp)
7 FLOAD f20, 20 * FREGBYTES(sp)
3 FLOAD f21, 21 * FREGBYTES(sp)
3 FLOAD f22, 22 * FREGBYTES(sp)
3 FLOAD f23, 23 * FREGBYTES(sp)
1 FLOAD f24, 24 * FREGBYTES(sp)
3 FLOAD f25, 25 * FREGBYTES(sp)
3 FLOAD f26, 26 * FREGBYTES(sp)
1 FLOAD f27, 27 * FREGBYTES(sp)
5 FLOAD f28, 28 * FREGBYTES(sp)
5 FLOAD f29, 29 * FREGBYTES(sp)
7 FLOAD f30, 30 * FREGBYTES(sp)
3 FLOAD f31, 31 * FREGBYTES(sp)
3 addi sp, sp, 32 * FREGBYTES //sp偏移
#endif
L mret //mret返回后转向mepc指向的位置执行

```

图 11 切换至第一个任务

由上注释分析可知，`rt_hw_context_switch_to` 通过传入的 `sp`，恢复 `cpu` 寄存器，其中 `mepc` 寄存器任务初始化时设置为任务的入口地址，`ra` 寄存器设置为返回地址，其指向公用函数 `rt_thread_exit`。当 `mret` 返回后，`pc` 更新为 `mepc` 值，即转向执行任务函数，若其不是一个持续执行的 `[while(1)]` 函数，那么其返回至 `_rt_thread_exit`，删除该任务，并切换至其他任务。由此也可理解图 10 中，最后一句注释“never come back”的含义了，一旦开始执行任务，`pc` 值被改变，再不会回到调用的地方。

其他 RTOS 中也有和此汇编函数类似的汇编实现，例如 `liteOS_M` 中的 `HalStartToRun`，`TencentOS_Tiny` 中的 `port_sched_start` 等。

(4) 任务之间的切换

了解了如何切换至第一个任务，如何实现不同任务之间的切换呢。在这之前想必对“任务优先级”，“时间片轮转”等概念有一定的了解。`rt-thread` 正常运行也需要个定时器为其提供时钟，且任务初始化时设置了任务优先级。调度器在就绪的任务列表中寻找优先级较高的任务切换执行，当优先级相同时，调度器会按照设置的时间片大小来轮流调度线程，用时间片来约束任务的单次执行时长。不管因优先级还是时间片耗尽，从当前任务切换至新任务时均需要保存当前任务的上下文至当前任务的堆栈区，获取新任务的堆栈，并从新任务堆栈区恢复上下文，切换并执行。

在 ARM 中，系统的滴答时钟由内核定时器 `Systick` 提供，并且在 `pendSV` 中进行任务切换。类比 RISC-V 我们内核提供了一个 64bit `Systick` 定时器，同时也有软中断 `SW_handler`（其实整个切换也不一定在某个中断中切换，只要做好上下文保存即可），需要切换时，置位其相应的 `pend` 位，即可触发进中断，实现切换。`rt-thread` 中用了三个全局变量，用于中断切换上下文 `rt_interrupt_from_thread`、`rt_interrupt_to_thread`、`rt_thread_switch_interrupt_flag`，前两个分别用来存储“from”线程 `sp` 指针和“to”线程的 `sp` 指针，当需要切换时，`flag` 被函数 `rt_hw_context_switch_interrupt` 置位，并触发进软中断如图 12 所示。在中断中实现“from”到“to”线程的切换，并将 `flag` 清零。

```
void rt_hw_context_switch_interrupt(rt_ubase_t from, rt_ubase_t to)
{
    if (rt_thread_switch_interrupt_flag == 0)
        rt_interrupt_from_thread = from;

    rt_interrupt_to_thread = to;
    rt_thread_switch_interrupt_flag = 1;
    /* switch just in sw_handler */
    sw_setpend();
}
```

图 12 中断切换上下文

`SW` 中断函数同样是一段汇编实现的代码，类容如下图 13 所示，其中重点代码已经给出注释，注意查看。从注释可以看出，其相较于启动第一个任务多了开头的就任务保存的过程。值得注意的是 V307 支持进中断后将硬件压栈临时关闭，这样在上下文切换的时候可以

手动恢复我们想要的寄存器值，而当中断返回后，硬件压栈自动打开，不影响其他外设中断使用硬件压栈。而对于没有此功能的 V103 来说，移植操作系统时不能打开硬件压栈。

```
.global SW_handler
.align 2
SW_handler:
#ifdef ARCH_RISCV_FPU
    addi    sp, sp, -32 * FREGBYTES
    FSTORE f0, 0 * FREGBYTES(sp) //如果支持硬件浮点，保存浮点寄存器
    FSTORE f1, 1 * FREGBYTES(sp)
    FSTORE f2, 2 * FREGBYTES(sp)
    FSTORE f3, 3 * FREGBYTES(sp)
    FSTORE f4, 4 * FREGBYTES(sp)
    FSTORE f5, 5 * FREGBYTES(sp)
    FSTORE f6, 6 * FREGBYTES(sp)
    FSTORE f7, 7 * FREGBYTES(sp)
    FSTORE f8, 8 * FREGBYTES(sp)
    FSTORE f9, 9 * FREGBYTES(sp)
    FSTORE f10, 10 * FREGBYTES(sp)
    FSTORE f11, 11 * FREGBYTES(sp)
    FSTORE f12, 12 * FREGBYTES(sp)
    FSTORE f13, 13 * FREGBYTES(sp)
    FSTORE f14, 14 * FREGBYTES(sp)
    FSTORE f15, 15 * FREGBYTES(sp)
    FSTORE f16, 16 * FREGBYTES(sp)
    FSTORE f17, 17 * FREGBYTES(sp)
    FSTORE f18, 18 * FREGBYTES(sp)
    FSTORE f19, 19 * FREGBYTES(sp)
    FSTORE f20, 20 * FREGBYTES(sp)
    FSTORE f21, 21 * FREGBYTES(sp)
    FSTORE f22, 22 * FREGBYTES(sp)
    FSTORE f23, 23 * FREGBYTES(sp)
    FSTORE f24, 24 * FREGBYTES(sp)
    FSTORE f25, 25 * FREGBYTES(sp)
    FSTORE f26, 26 * FREGBYTES(sp)
    FSTORE f27, 27 * FREGBYTES(sp)
    FSTORE f28, 28 * FREGBYTES(sp)
    FSTORE f29, 29 * FREGBYTES(sp)
    FSTORE f30, 30 * FREGBYTES(sp)
    FSTORE f31, 31 * FREGBYTES(sp)
#endif

    addi sp, sp, -32 * REGBYTES
    STORE x5, 5 * REGBYTES(sp) //存储当前t0至堆栈
    li    t0, 0x80
    STORE t0, 2 * REGBYTES(sp) //存储MPPIE

    li    t0, 0x20 //对于v307,可以置位bit5临时关闭硬件压栈,任务切换时我们手动恢复寄存器
    csrs 0x804, t0 //mret返回后,硬件自动清除该位,使能硬件压栈,即通用中断可以在操作系统
    //下使用硬件压栈
    STORE x1, 1 * REGBYTES(sp) //存储当前任务整形寄存器
    STORE x4, 4 * REGBYTES(sp)
    STORE x6, 6 * REGBYTES(sp)
    STORE x7, 7 * REGBYTES(sp)
    STORE x8, 8 * REGBYTES(sp)
    STORE x9, 9 * REGBYTES(sp)
    STORE x10, 10 * REGBYTES(sp)
    STORE x11, 11 * REGBYTES(sp)
    STORE x12, 12 * REGBYTES(sp)
    STORE x13, 13 * REGBYTES(sp)
    STORE x14, 14 * REGBYTES(sp)
    STORE x15, 15 * REGBYTES(sp)
    STORE x16, 16 * REGBYTES(sp)
    STORE x17, 17 * REGBYTES(sp)
    STORE x18, 18 * REGBYTES(sp)
    STORE x19, 19 * REGBYTES(sp)
    STORE x20, 20 * REGBYTES(sp)
    STORE x21, 21 * REGBYTES(sp)
    STORE x22, 22 * REGBYTES(sp)
    STORE x23, 23 * REGBYTES(sp)
    STORE x24, 24 * REGBYTES(sp)
    STORE x25, 25 * REGBYTES(sp)
    STORE x26, 26 * REGBYTES(sp)
    STORE x27, 27 * REGBYTES(sp)
    STORE x28, 28 * REGBYTES(sp)
    STORE x29, 29 * REGBYTES(sp)
    STORE x30, 30 * REGBYTES(sp)
    STORE x31, 31 * REGBYTES(sp)
```

```

csrrw sp, mscratch, sp //此处交换了sp和mscratch值, mscratch该值在启动第一个任务前存入
call rt_interrupt_enter //使用新sp值调用中断入口c函数

jal sw_clearpend //调用清除软中断, 此处还可以调用用户软中断函数

call rt_interrupt_leave //使用新sp值调用中断出口c函数
csrrw sp, mscratch, sp //切回任务栈sp

la s0, rt_thread_switch_interrupt_flag //判断该标志位是否为1, 如果为1表明确实需要切换任务
lw s2, 0(s0) //否则跳转到标号1处执行, 1处为恢复寄存器操作, 即又
beqz s2, 1f //把当前任务的寄存器恢复, 恢复当前任务的执行, 该标志位
sw zero, 0(s0) //当需要切换时由外部的c函数置位, 此处清零

csrr a0, mepc
STORE a0, 0 * REGBYTES(sp) //需要切换时, 继续存储当前任务的mepc值

la s0, rt_interrupt_from_thread //获取rt_interrupt_from_thread变量地址, 该变量存放的是当前线程的sp存放地址
LOAD s1, 0(s0) //获取该地址
STORE sp, 0(s1) //把当前sp保存至该地址

la s0, rt_interrupt_to_thread //获取rt_interrupt_to_thread变量地址, 该变量存放要切换的线程的sp存放地址
LOAD s1, 0(s0) //获取新线程sp地址
LOAD sp, 0(s1) //从该地址获取新线程的sp值, 赋给sp寄存器, 即cpu的sp寄存器更新为新任务的sp值

LOAD a0, 0 * REGBYTES(sp) //基于新线程的sp, 恢复其mepc寄存器值
csrw mepc, a0

1: LOAD x1, 1 * REGBYTES(sp)

li t0, 0x7800
csrs mstatus, t0
LOAD t0, 2*REGBYTES(sp)
csrs mstatus, t0 //恢复mstatus寄存器值并保持机器模式

```

```

LOAD x4, 4 * REGBYTES(sp) //恢复整形寄存器值
LOAD x5, 5 * REGBYTES(sp)
LOAD x6, 6 * REGBYTES(sp)
LOAD x7, 7 * REGBYTES(sp)
LOAD x8, 8 * REGBYTES(sp)
LOAD x9, 9 * REGBYTES(sp)
LOAD x10, 10 * REGBYTES(sp)
LOAD x11, 11 * REGBYTES(sp)
LOAD x12, 12 * REGBYTES(sp)
LOAD x13, 13 * REGBYTES(sp)
LOAD x14, 14 * REGBYTES(sp)
LOAD x15, 15 * REGBYTES(sp)
LOAD x16, 16 * REGBYTES(sp)
LOAD x17, 17 * REGBYTES(sp)
LOAD x18, 18 * REGBYTES(sp)
LOAD x19, 19 * REGBYTES(sp)
LOAD x20, 20 * REGBYTES(sp)
LOAD x21, 21 * REGBYTES(sp)
LOAD x22, 22 * REGBYTES(sp)
LOAD x23, 23 * REGBYTES(sp)
LOAD x24, 24 * REGBYTES(sp)
LOAD x25, 25 * REGBYTES(sp)
LOAD x26, 26 * REGBYTES(sp)
LOAD x27, 27 * REGBYTES(sp)
LOAD x28, 28 * REGBYTES(sp)
LOAD x29, 29 * REGBYTES(sp)
LOAD x30, 30 * REGBYTES(sp)
LOAD x31, 31 * REGBYTES(sp)
addi sp, sp, 32 * REGBYTES

```

```

#ifdef ARCH_RISCV_FPU
    FLOAD    f0, 0 * FREGBYTES(sp) //有硬件浮点，恢复浮点寄存器值
    FLOAD    f1, 1 * FREGBYTES(sp)
    FLOAD    f2, 2 * FREGBYTES(sp)
    FLOAD    f3, 3 * FREGBYTES(sp)
    FLOAD    f4, 4 * FREGBYTES(sp)
    FLOAD    f5, 5 * FREGBYTES(sp)
    FLOAD    f6, 6 * FREGBYTES(sp)
    FLOAD    f7, 7 * FREGBYTES(sp)
    FLOAD    f8, 8 * FREGBYTES(sp)
    FLOAD    f9, 9 * FREGBYTES(sp)
    FLOAD    f10, 10 * FREGBYTES(sp)
    FLOAD    f11, 11 * FREGBYTES(sp)
    FLOAD    f12, 12 * FREGBYTES(sp)
    FLOAD    f13, 13 * FREGBYTES(sp)
    FLOAD    f14, 14 * FREGBYTES(sp)
    FLOAD    f15, 15 * FREGBYTES(sp)
    FLOAD    f16, 16 * FREGBYTES(sp)
    FLOAD    f17, 17 * FREGBYTES(sp)
    FLOAD    f18, 18 * FREGBYTES(sp)
    FLOAD    f19, 19 * FREGBYTES(sp)
    FLOAD    f20, 20 * FREGBYTES(sp)
    FLOAD    f21, 21 * FREGBYTES(sp)
    FLOAD    f22, 22 * FREGBYTES(sp)
    FLOAD    f23, 23 * FREGBYTES(sp)
    FLOAD    f24, 24 * FREGBYTES(sp)
    FLOAD    f25, 25 * FREGBYTES(sp)
    FLOAD    f26, 26 * FREGBYTES(sp)
    FLOAD    f27, 27 * FREGBYTES(sp)
    FLOAD    f28, 28 * FREGBYTES(sp)
    FLOAD    f29, 29 * FREGBYTES(sp)
    FLOAD    f30, 30 * FREGBYTES(sp)
    FLOAD    f31, 31 * FREGBYTES(sp)
    addi    sp, sp, 32 * FREGBYTES
#endif
mret//返回后pc值更新为mepc值，该值为此次新任务上一次被进中断切走后的下一条指令地址
//即续接上一次切换位置继续执行。v307再这之后也将重新使能硬件压栈，以保证新任务
//执行期间的通用中断仍然可以使用硬件压栈。

```

图 13 软中断切换上下文过程

对于其他的 RTOS 也是大同小异，例如鸿蒙 LiteOS_M 中有个 `g_losTask` 全局变量，其是个结构体，内部为两个任务控制块类型的指针，分别为 `*runTask` 指向当前运行的任务，`*newTask` 指向要切换的新任务，腾讯的 `TencentOS_Tiny` 中亦有两个任务控制块类型的指针 `*k_curr_task` 指向当前任务，`*k_next_task` 指向下一个要切换任务。这和 `rt-thread` 中的“from”和“to”是为异曲同工之处。通过管理操作这些变量，实现任务到任务的切换。

以上四个点弄清楚后，把一个实时内核成功移植至 RISC-V 平台应该不难。