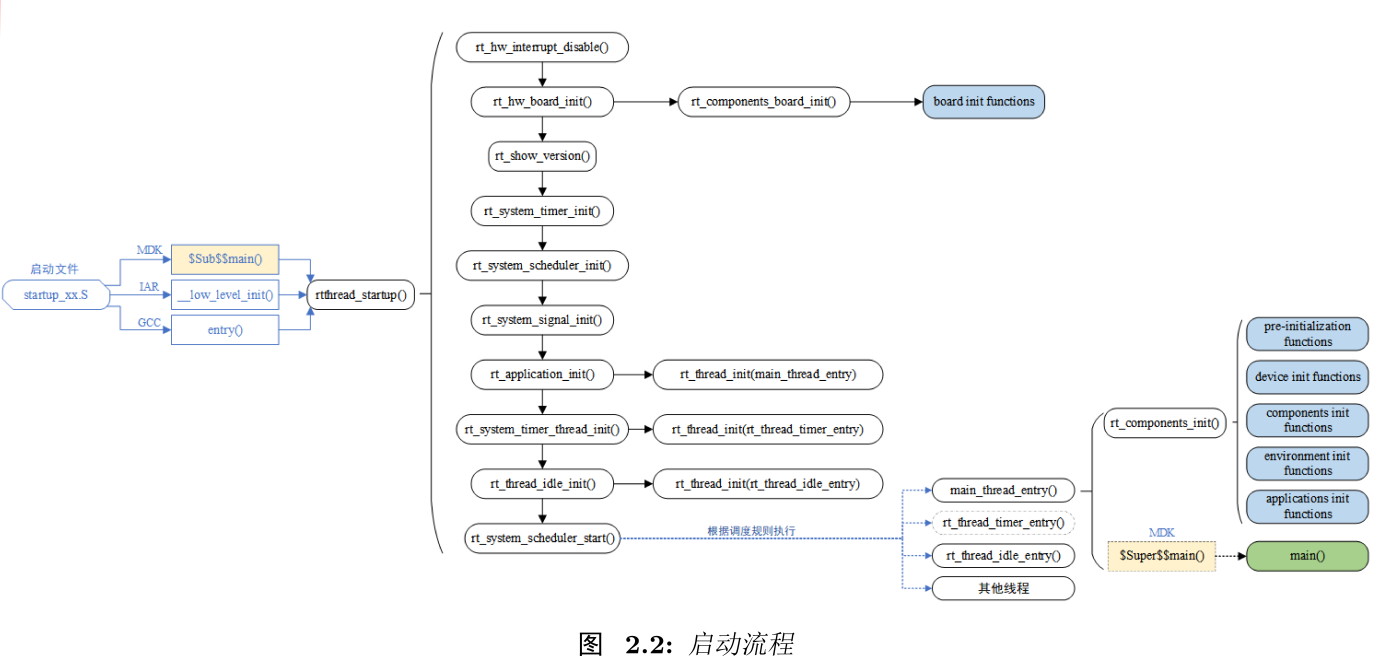
**RT操作系统启动详细分析**



RTT4/port/start.S文件是MIPS32 所需的典型独立启动代码，汇编语言配置总线、堆栈、存储器、高速缓存、翻译后备缓存区TLB，跳转到rtthread\_startup（）函数启动内核。

RTT4/src/components.c文件中rtthread\_startup（）函数，调用以下函数启动内核：

# 第一章 rt\_hw\_interrupt\_disable

RTT4/port/irq.c文件中rt\_hw\_interrupt\_disable（）函数关闭中断。

# 第二章rt\_hw\_board\_init

RTT4/port/bsp\_start.c文件中rt\_hw\_board\_init（）函数板级初始化。安装异常向量(数据/指令必须使用K1地址)，初始化中断向量表，获取存储空间大小并初始化RT系统堆栈，初始化控制台、操纵系统定时器、板子上的其他组件、取消中断屏蔽。

## 2.1 mips\_init\_isr\_table

RTT4/port/irq.c文件中mips\_init\_isr\_table（）初始化中断向量表。（循环调用，初始化每个中断的处理程序与参数）。

## 2.2 rt\_system\_heap\_init

RTT4/port/mem.c文件中rt\_system\_heap\_init（）初始化RT系统堆栈。（对齐地址、堆开始、堆结束）。用rt\_sem\_init初始化一个名为“heap”的信号量。

### 2.2.1 rt\_sem\_init

RTT4/src/ipc.c文件中rt\_sem\_init（）初始化一个信号量。见附录。

## 2.3 console\_init

ls1x-drv/console/console.c文件中console\_init（）初始化控制台。设置波特率并打开串口。

### 2.3.1 ls1x\_uart\_init

ls1x-drv/console/console.c文件中ls1x\_uart\_init（）设置控制台波特率。

### 2.3.2 ls1x\_uart\_open

ls1x-drv/console/console.c文件中ls1x\_uart\_open（）打开串口。

## 2.4 Clock\_initialize

RTT4/port/tick.c文件中Clock\_initialize（）初始化操作系统定时器.计算步长。mips\_mask\_interrupt（）禁用中断，LS1x\_CPU\_FREQUENCY（）获取龙芯CPU频率用以设置定时器步长，mips\_set\_timer（）设定定时器（跳转到汇编），rt\_hw\_interrupt\_install（）安装时钟中断向量，mips\_unmask\_interrupt（）使能中断，mips\_enable\_dc（）启用计数比较。

### 2.4.1 rt\_hw\_interrupt\_install

RTT4/port/irq.c文件中rt\_hw\_interrupt\_install（）安装时钟中断向量。其中断处理程序为Clock\_isr（）。

RTT4/port/irq.c文件中Clock\_isr（）首先使Clock\_driver\_ticks时间值加一，再用mips\_set\_timer（）复位定时器，最后执行rt\_tick\_increase（）增加RT操作系统时间值。

RTT4/src/clock.c文件中rt\_tick\_increase（）函数全局时间值rt\_tick加一，调用rt\_thread\_self（）获取当前线程，检查当前时间片时间值，如果超时rt\_schedule（）启用调度器切换线程。最后调用rt\_timer\_check（）该函数将检查定时器列表，如果发生超时事件，将调用相应的超时函数。

## 2.5 rt\_components\_board\_init

RTT4/src/components.c文件中第83行rt\_components\_board\_init（）此板级初始化为空。

## 2.6 mips\_unmask\_interrupt

RTT4/port/mips/mips.c文件中mips\_unmask\_interrupt（）取消中断屏蔽。

# 第三章 rt\_show\_version

RTT4/src/kservice.c文件中rt\_show\_version（）打印版本信息。

# 第四章 rt\_system\_timer\_init

RTT4/src/timer.c文件中rt\_system\_timer\_init（）初始化系统定时器。用循环测定出硬件定时器列表rt\_timer\_list中定时器的个数，并挨个初始化（此处只有一个硬件定时器）

## 4.1 rt\_list\_init

RTT4/include/rtsevervice.h文件中rt\_list\_init（）初始化列表中某个节点，前后节点都为1。

# 第五章 rt\_system\_scheduler\_init

RTT4/src/scheduler.c文件中rt\_system\_scheduler\_init（）初始化调度器系统。遍历所有优先级初始化线程优先级列表，初始化已就绪的优先组为0，初始化已经失效的线程。

## 5.1 rt\_list\_init

RTT4/include/rtsevervice.h文件中rt\_list\_init（）初始化列表中某个节点，前后节点都为1。

# 第六章 rt\_application\_init

RTT4/src/components.c文件中rt\_application\_init（）创建主线程。堆栈大小为1024\*8，优先级为10，时间片为20个tick。

## 6.1 rt\_thread\_create

RTT4/src/thread.c文件中rt\_thread\_create（）创建主线程，创建一个线程对象并分配内存和堆栈。见附录

## 6.2 rt\_thread\_startup

RTT4/src/thread.c文件中rt\_thread\_startup（）开启主线程，启动一个线程并将其放入系统就绪队列（此时调度程序未启动，并不会执行主线程），详细信息见附录。

## 6.3 main\_thread\_entry线程

RTT4/src/components.c文件中main\_thread\_entry（）为主线程入口函数

### 6.3.1 rt\_components\_init

RTT4/src/components.c文件中rt\_components\_init（）初始化RT-Thread组件，其写法比较奇特，用函数指针调用，故该函数调用十分不明显。详细过程如下：

#### 6.3.1.1 dfs\_init

RTT4/components/dfs/src/dfs.c文件中dfs\_init（）初始化设备文件系统，具体操作如下：清除文件系统操作表与文件系统表等，创建设备文件系统锁，设置当前工作目录。

rt\_mutex\_init（）创建名为"fslock"的文件锁，本质上是一个互斥量。

#### 6.3.1.2 libc\_system\_init

RTT4/components/libc/compilers/newlib/libc.c文件中libc\_system\_init（）啥都没执行。

#### 6.3.1.3 clock\_time\_system\_init

RTT4/components/libc/time/clock\_time.c文件中clock\_time\_system\_init（）时钟时间系统初始化。具体操作如下：用函数rt\_device\_find（）找到实时时钟设备，读取实时秒数，并获取tick值。最后通过计算得出时间值（微秒与秒）。

RTT4/src/device.c文件中rt\_device\_find（）按指定名称查找设备驱动，遍历information->object\_list结构体链表中的所有节点，找到节点名称为指定名称的结构体（对象）。

#### 6.3.1.4 finsh\_system\_init

RTT4/components/finish/shell.c文件中finsh\_system\_init（）初始化操作系统外壳。具体操作如下：初始化系统功能与系统变量，创建一个外壳线程并启动它(线程名字FINSH\_THREAD\_NAME、线程入口finsh\_thread\_entry、堆栈大小1024\*8、优先级20、时间片10)，初始化一个名叫shrx的信号量，设置提示方式。

##### 6.3.1.4.1 finsh\_thread\_entry线程

RTT4/components/finish/shell.c文件中finsh\_thread\_entry（）该线程是操作系统外壳线程，永远不会失效。该线程的作用就是接收CMD命令行，然后执行相关命令。具体操作如下：将控制台设备作为操作系统外壳设备，进入循环（始终不会结束的循环）周期性释放系统资源，如果外部线程占用资源将挂起此线程，否则从控制台读取输入的命令，解码后执行相关操作（方向键关乎历史命令，退格键擦除命令，回车换行键执行等）。

### 6.3.2 main

main.c文件中main（）即为主程序。期间执行了一些列初始化操作。

# 第七章 rt\_system\_timer\_thread\_init

空

# 第八章 rt\_thread\_idle\_init

RTT4/src/idle.c文件中rt\_thread\_idle\_init（）创建空闲线程。优先级为31，时间片为32个tick。

## 8.1 rt\_thread\_init

RTT4/src/thread.c文件中 rt\_thread\_init（）创建空闲线程，（静态）。支持多线程编程，即创建多个空线程，但此时只有一个。空闲线程名称为“tidle+数字”，入口为rt\_thread\_idle\_entry，堆栈大小为sizeof(rt\_thread\_stack[i])，优先级为31（已经特别低了，最低为32），时间片为32。

## 8.2 rt\_thread\_startup

RTT4/src/thread.c文件中rt\_thread\_startup（）开启空线程。

## 8.3 rt\_thread\_idle\_entry线程

RTT4/src/idle.c文件中rt\_thread\_idle\_entry（）为空线程入口函数。此线程套用while（1）循环，属于系统线程，所以不会被剔除掉。用for循环遍历钩子函数列表idle\_hook\_list[i]（最多四个钩子函数），如果不为空就执行。最后调用rt\_thread\_idle\_excute（）清除已经执行完毕的线程。

### 8.3.1 idle\_hook\_list[i]

\*\*\*\*\*\*\*\*\*\*\*\*\*idle\_hook\_list[i]（）如果钩子函数列表非空，将执行该函数。

### 8.3.2 rt\_thread\_idle\_excute

RTT4/src/idle.c文件中rt\_thread\_idle\_excute（）在空闲时清除线程。调用\_has\_defunct\_thread()函数判断是否存在已经结束的线程，如果存在就一直循环卸载。调用rt\_hw\_interrupt\_disable()禁止中断，再次检查是否存在已经运行结束的线程，如果有就调用rt\_list\_entry（）获取列表tlist中下一个运行结束的线程。调用rt\_list\_remove（）移除已经运行完的线程，锁定调度程序并调用线程清理thread->cleanup(thread)，调用rt\_object\_is\_systemobject（）函数确保不是系统对象后，rt\_object\_detach（）分离此对象，解开调度器锁，调用rt\_hw\_interrupt\_enable（）使能中断。调用RT\_KERNEL\_FREE（）函数释放该线程占用的堆栈空间，调用函数rt\_object\_delete（）删除线程对象。

# 第九章 rt\_system\_scheduler\_start

RTT4/src/scheduler.c文件中rt\_system\_scheduler\_start（）启动调度程序。调用\_get\_highest\_priority\_thread（）选择一个具有最高优先级的线程，然后切换到它。切换过程如下：先调用函数rt\_schedule\_remove\_thread（）移除就序列表中下一个将要执行的线程，然后将下一个将要执行的线程状态更改为运行状态，最后调用函数rt\_hw\_context\_switch\_to（）加载堆栈完成线程切换。

# 第十章 抽象表达

暂且分为四大类线程，如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 线程 | 堆栈大小 | 优先级 | 时间片 | 运行 |
| main\_thread\_entry 主线程 | 1024\*8 | 10 | 20 | 单次 |
| finsh\_thread\_entry 控制台线程 | 1024\*8 | 20 | 10 | while |
| rt\_thread\_idle\_entry 空闲线程 | sizeof(rt\_thread\_stack[i]) | 31 | 32 | while |
| 用户创建线程 | 任意 | 0-32 | 任意 | 任意 |

线程状态：创建、就绪、运行、阻塞、死亡。

该代码中：线程列表、就绪线程列表、死亡线程列表……

**重中之重：**线程和中断不同，不是如果存在比当前线程优先级高的线程就绪就立即切换，而是一定要等到当前线程让出资源（主动延时让出或者时间片超时被迫让出）而启用调度器时才会发生线程切换。

线程运行过程详解：上述表格前三个线程，首先执行主线程一旦主线程延时进入阻塞状态让出资源将立即执行控制台线程，控制台线程延时进入阻塞状态让出资源后主线程继续执行。主线程执行完毕启用调度器，优先执行控制台线程，该线程让出资源后（目前主线程执行完毕处于死亡状态，用户并未创建线程）所以会执行优先级最低的空闲线程，该空闲线程会清理执行完毕的主线程，然后再执行控制台线程……（有了用户创建线程也同上）

# 附加1 创建信号量 RTT4/src/ipc.c文件中rt\_sem\_init（信号量对象，信号量名字，信号量初始值，信号量标志）

调用rt\_object\_init（）初始化一个对象，调用rt\_ipc\_object\_init（）初始化一个进程间通信（InterProcess Communication）IPC对象，设置初始值，设置信号量标志。

1. rt\_object\_init（要初始化的指定对象，对象类型，对象名称），通过指定对象类型解读出对象信息，禁用线程调度。从该类型对象对应的对象列表开始遍历，查找对象。新建一个对象指针，让该指针指向刚刚查找到的对象。启用线程调度，设置对象类型、对象名称、RT\_OBJECT\_HOOK\_CALL（rt\_object\_attach\_hook, (object)）设置回调函数。中断禁止，将对象插入信息对象列表，中断使能。

1.1 rt\_list\_entry（节点入口，结构体类型，结构中列表的名称）获取此条目的结构。

1.2 rt\_list\_insert\_after（列表，新建节点），在列表后插入一个节点。将对象插入信息对象列表。

2. rt\_ipc\_object\_init（）此资源上挂起的线程均为空。

# 附加2 启用调度器 RTT4/src/scheduler.c文件中rt\_schedule（）

它将选择一个具有最高优先级的线程，然后切换到它。①禁止中断，确保调度器没有被锁定，确保有某个线程处于就绪状态，获取已经就绪线程的最高优先级highest\_ready\_priority与下一个将要被执行的线程to\_thread。如果当前线程处于运行状态，做进一步判断：当前线程优先级小于最高就绪线程优先级则下一个将要被执行的线程还是当前线程，当先线程优先级等于最高优先级且没有挂起下一个将要被执行的线程还是当先线程，如果当前线程是被挂起的就取消挂起。②如果下一个要被执行的线程是当前线程，从就序列表删除当前线程改变线程运行状态，否则执行调度。更新正在运行的线程优先级信息，rt\_schedule\_insert\_thread(from\_thread)将当前线程插入就序列表，rt\_schedule\_remove\_thread(to\_thread)将下一个要被执行的线程从就序列表中删除。\_rt\_scheduler\_stack\_check (to\_thread)检测线程是否能正常调用。③最后根据标志置位rt\_interrupt\_nest选择线程切换两种方式rt\_hw\_context\_switch（）和rt\_hw\_context\_switch\_interrupt（）之一。将当前线程运行情况保存到线程栈中，从线程栈加载下一个将要执行的线程情况完成切换。最后使能中断。

1. to\_thread = \_get\_highest\_priority\_thread(&highest\_ready\_priority)从参数rt\_thread\_ready\_priority\_group解码出已经就绪的最高优先级highest\_ready\_priority，调用rt\_list\_entry（）从列表rt\_thread\_priority\_table[highest\_ready\_priority]获取就绪的最高优先级线程。

2. rt\_schedule\_insert\_thread(from\_thread)禁止中断，当前线程不插入只改变状态。线程状态改为就绪，改变已就绪线程优先级状态rt\_thread\_ready\_priority\_group，rt\_list\_insert\_before（）将当前线程thread->tlist插入对应优先级的就序列表rt\_thread\_priority\_table[thread->current\_priority]。

3. rt\_schedule\_remove\_thread(to\_thread)禁止中断，调用rt\_list\_remove（）从就序列表移除线程，调用rt\_list\_isempty（）检测该优先级线程列表是否为空，如果是空就更改rt\_thread\_ready\_priority\_group。使能中断。

# 附加3 创建线程 RTT4/src/thread.c文件中rt\_thread\_create（）

调用rt\_object\_allocate（）分配一个线程对象，RT\_KERNEL\_MALLOC（）分配堆栈，分配失败后调用rt\_object\_delete（）删除线程对象，最终调用函数\_rt\_thread\_init（）初始化线程。

1. rt\_object\_allocate（）从对象系统中分配一个对象。调用rt\_object\_get\_information（）从对象类型中解读出对象信息，调用RT\_KERNEL\_MALLOC（）分配一个对象，rt\_memset（）清除存储器残留数据，设置对象类型、标志、名字，用RT\_OBJECT\_HOOK\_CALL（）调用钩子函数。禁止中断，调用rt\_list\_insert\_after（）将新建对象插入对象列表，使能中断。

2. RT\_KERNEL\_MALLOC（）分配对象直接调用rt\_malloc（）。更新要分配空间的大小size以对齐（不能太大也不能太小需要修正），调用rt\_sem\_take（）获取内存信号量，最后分配存储器空间。

3. rt\_object\_delete（）用RT\_OBJECT\_HOOK\_CALL（）调用钩子函数，重置对象类型，禁止中断，将对象从列表中移除，使能中断，调用RT\_KERNEL\_FREE（）释放该对象占用的存储器空间。

4. \_rt\_thread\_init（）初始化线程，rt\_list\_init(&(thread->tlist))初始化线程列表，设置线程入口、线程参数、堆栈地址与堆栈大小。调用rt\_memset（）初始化线程堆栈，调用rt\_hw\_stack\_init（）设置线程堆栈指针[入口、参数、堆栈地址、出口]。设置线程优先级、时间片、错误信息、线程状态、初始化清除函数并清理用户数据，调用rt\_timer\_init（）初始化定时器。最后用RT\_OBJECT\_HOOK\_CALL(rt\_thread\_inited\_hook, (thread))调用钩子函数。

# 附加4 开启线程 RTT4/src/thread.c文件中rt\_thread\_startup（）

设置线程优先级与掩码，该线程默认状态设置为挂起。调用rt\_thread\_resume(thread)将该线程放入系统就绪队列。调用rt\_thread\_self（）检测当前是否有线程正在运行，如果没有的话，调用rt\_schedule（）启用调度器。