昆明学院学报　2018，40（3）：57～64　ISSN 1674－5639　CN 53－1211/G4
Journal of Kunming University

MQXLite 嵌入式操作系统的任务调度分析研究

石　栋，文　瑾
（昆明学院 信息技术学院，云南 昆明 650214）

摘要：针对目前嵌入式实时操作系统理论研究尚缺少对操作系统具体实现方法的探讨．因此，分析了MQXLite 中就绪队列和等待队列的成员结构，归纳出 MQXLite 实现任务调度的函数集群，并对 MQXLite 调度入口函数的功能进行剖析，同时研究了内核调度函数 sched_internal 的工作流程图，以及 MQXLite 操作系统中任务的调度方式．然后以苏州大学开发的 MQXLite 工程框架为例，进一步阐述其任务调度的具体实现方法．

关键词：嵌入式操作系统；MQXLite；任务；调度；同步
中图分类号：TP391　**文献标识码**：A　**文章编号**：1674－5639（2018）03－0057－08
DOI：10.14091/j.cnki.kmxyxb.2018.03.011

Research on Task Scheduling of MQXLite Embedded Operating System

SHI Dong，WEN Jin
（College of Information Technology，Kunming University，Kunming，Yunnan，China　650214）

Abstract：Due to the lack of theory research on the practical method of embedded real-time operating systems，the member structure of the ready queue and waiting queue in MQXLite was analyzed. We summarized the functions of MQXLite implements task scheduling，analyzed the function of the MQXLite scheduling entry function，meanwhile，studied the working flow chart of the kernel scheduling function sched_internal and the scheduling of the tasks in the MQXLite operating system. Finally，the MQXLite project framework developed by Suzhou University was taken as an example to further state the task scheduling and synchronization of embedded real-time operating system.

Key words：embedded operating system；MQXLite；task；scheduling；synchronization

MQXLite 是 NXP 公司在其发布的嵌入式实时操作系统 MQX 上精简出来的一款免费轻量级嵌入式实时操作系统[1]．相较于 MQX 而言，MQLite 具有体积小，支持应用低于 4 KB 的 RAM 空间运行，内核精简等特点．

文献［3～4］对嵌入式操作系统任务调度进行了以下几方面的讨论：1）分析嵌入式操作系统任务调度架构；2）分析嵌入式操作系统任务调度采用的方法；3）具体讨论一种嵌入式操作系统任务调度的改进．由此可知，目前尚缺少对嵌入式操作系统任务调度具体实现方法的相关研究．因此，本文以 MQXLite 轻量级嵌入式实时操作系统为例，并结合苏州大学嵌入式实验室开发的 MQXLite 工程框架，阐述其任务调度的具体实现，旨在为研究人员和开发者提供参考．

1　MQXLite 的任务和队列结构

1.1　任务

任务是完成一定功能的函数．但和普通函数不同的是一个任务包含 3 个基本要素：任务函数、任务堆栈、任务描述符[5]．操作系统中为每一个任务定义了单独的任务堆栈来保存该任务的相关信息，如上下文、变量等．而任务描述符是创建任务时系统记录任务相关信息的 TD_ STRUCT 结构体类型链表（如表 1 所示，表中结构体成员为部分），系统调用任务就是通过该链表实现．

收稿日期：2017－10－11
作者简介：石栋（1972—），男，云南昭通人，讲师，主要从事物联网系统开发研究.

表 1 TD_STRUCT 结构体成员

成员变量		说明
struct td_struct *	TD_NEXT	下一任务描述符地址
struct td_struct *	TD_PREV	上一任务描述符指针
_mqx_uint	STATE	当前任务状态
_task_id	TASK_ID	当前任务的唯一编号
pointer	STACK_BASE	当前任务堆栈指针（初始值）
pointer	STACK_PTR	当前任务堆栈指针（当前值）
pointer	STACK_LIMIT	当前任务堆栈大小

1.2 MQXLite 任务队列

任务队列是 MQXLite 中实现任务管理的一种数据结构. 任务队列中的一个重要成员是任务描述符，任务描述符保存了任务指向前一个任务和后一个任务的指针. 当任务被创建后，先加入到任务列表队列，随后被添加到相应优先级的任务就绪队列中，而能够获得运行的任务就是系统当前优先级最高就绪任务. 也就是说，MQXLite 从任务的创建到调度和同步都离不开队列，下面对就绪队列和等待队列进行剖析.

1.2.1 就绪任务队列结构

MQXLite 在 mqx_init.c 文件中声明了一个 READY_Q_STRUCT 类型的结构体数组用于保存优先级就绪队列的头节点：

READY_Q_STRUCT mqx_static_ready_queue[MQX_READY_QUEUE_ITEMS]，其中的 READY_Q_STRUCT 是 MQXLite 定义的结构体，其成员变量如表 2 所示.

表 2 READY_Q_STRUCT 结构体成员

成员变量		说明
TD_STRUCT_PTR	HEAD_READY_Q	本队列头节点地址
TD_STRUCT_PTR	TAIL_READY_Q	本队列尾节点地址
struct ready_q_struct	* NEXT_Q	下一优先级就绪队列头节点地址
uint_16	ENABLE_SR	本队列的硬件优先级
uint_16	PRIORITY	本队列的软件优先级

初始化 MQXLite 时，_mqxlite_init( ) 函数调用 _psp_init_readyqs( ) 函数进行就绪队列的初始化工作. _psp_init_readyqs( ) 函数执行步骤如下.

1）给内核数据区的 READY_Q_LIST 字段赋空值：

```
kernel_data -> READY_Q_LIST = (READY_Q_STRUCT_PTR) NULL;
```

2）计算就绪队列个数，即需要的头节点数. 目的是为相同优先级任务创建一个就绪队列：

```
priority_levels = kernel_data -> LOWEST_TASK_PRIORITY + 2;
```

代码中的“kernel_data -> LOWEST_TASK_PRIORITY”等于任务模板中的最低优先级值（最大数字）.

3）获取就绪队列头节点数组首地址，准备就绪队列的初始化：

```
q_ptr = &mqx_static_ready_queue[0];
```

4）初始化就绪队列的头节点：

```
n = priority_levels;
while (n--) {
q_ptr -> HEAD_READY_Q = (TD_STRUCT_PTR) q_ptr;
q_ptr -> TAIL_READY_Q = (TD_STRUCT_PTR) q_ptr;
q_ptr -> PRIORITY = (uint_16) n;
……
```

```
q_ptr -> NEXT_Q = kernel_data - > READY_Q_LIST;
kernel_data -> READY_Q_LIST = q_ptr + + ;
}
```

每一个头节点的 HEAD_READY_Q 字段和 TAIL_ READY_Q 字段均指向头节点自身，NEXT_Q 字段指向下一优先级就绪队列头节点地址．但最低优先级头节点的 NEXT_Q 字段为空，表明当前节点为整个头节点的末尾．起防止越界的作用．

5）将最高优先级队列头节点指针保存到内核数据区的 CURRENT_READY_Q 字段：

kernel_data -> CURRENT_READY_Q = kernel_data -> READY_Q_LIST.

就绪任务队列是 MQXLite 调度机制中最重要的数据结构之一．MQXLite 在初始化时会从最高优先级 0 到最低优先级 $N+1$ 分别创建一个就绪任务队列头节点，共 $N+2$ 个不同优先级的就绪任务头节点，用于链接各优先级的就绪任务．这里 N 是指任务模板列表中定义的任务优先级最大值 N（优先级最低）．在这些头节点中，优先级为 N 的头节点对应的是系统创建的空闲任务，空闲任务在系统初始化时创建．$N+1$ 优先级的任务头节点是一个空节点，作用是防止越界．就绪队列为双向链表，当队列为空时，头节点中的 HEAD_READY_Q 成员和 TAIL_READY_Q 成员均指向本头节点地址．$N+2$ 个就绪任务队列头节点之间也以链表形式连接在一起．

1.2.2　等待队列结构

等待队列是 MQXLite 对任务队列管理的另一个典型应用．在 MQXLite 中，当任务进入等待事件位置位、等待消息、等待延时等状态时，都会从就绪队列转移到相应的等待队列中．为了有效管理这些进入等待状态的任务，MQXLite 专门定义了一个结构体类型 QUEUE_STRUCT，该结构体的成员变量如表 3 所示．

表 3　QUEUE_STRUCT 成员

成员	说明
struct queue_element_struct_PTR_NEXT	队列中下一个元素指针，无则指向自身
struct queue_element_struct_PTR_PREV	队列中上一个元素指针，无则指向自身
uint_16　SIZE	队列中元素的数目
uint_16　MAX	队列元素的最大数目，为 0 则无限制

由表 3 可以看出，等待队列是一个双向链表．为方便管理这些链表，MQXLite 还定义了一组变量管理各种等待队列的头节点（表 4），除了延时等待队列头节点是定义在内核数据区，其他头节点均定义在相应功能的结构体中．

表 4　队列头节点

结构体类型	成员	说明
LWEVENT_STRUCT	WAITING_TASKS	轻量级事件等待队列头节点
LWMSGQ_STRUCT	WAITING_WRITERS	轻量级消息等待写队列头节点
LWMSGQ_STRUCT	WAITING_READERS	轻量级消息等待读队列头节点
KERNEL_DATA_STRUCT	TIMEOUT_QUEUE	延时等待队列头节点
MUTEX_STRUCT	WAITING_TASKS	互斥等待队列头节点

2　MQXLite 任务调度剖析

任务调度是指操作系统在多任务环境下依照一定规则实现任务的切换工作．而调度器是嵌入式操作系统中具体实现任务调度的服务函数或函数集合．MQXLite 的调度器是一些跟调度有关的函数组成的集合．这些函数的核心是 sched_internal 函数和 switch_task 函数，sched_start_internal，_sched_run_internal，_int_kernel_isr，_sched_check_scheduler_internal，_sched_execute_scheduler_inter-

nal,_task_block 是不同情况下执行调度的入口函数. 根据不同的具体情况，使用特定的函数实现 MQXLite 的调度，通过这种机制组成了 MQXLite 的调度器，如图 1 所示. MQXLite 中的调度函数都用汇编语言编写. 下面逐一分析各调度函数的功能.

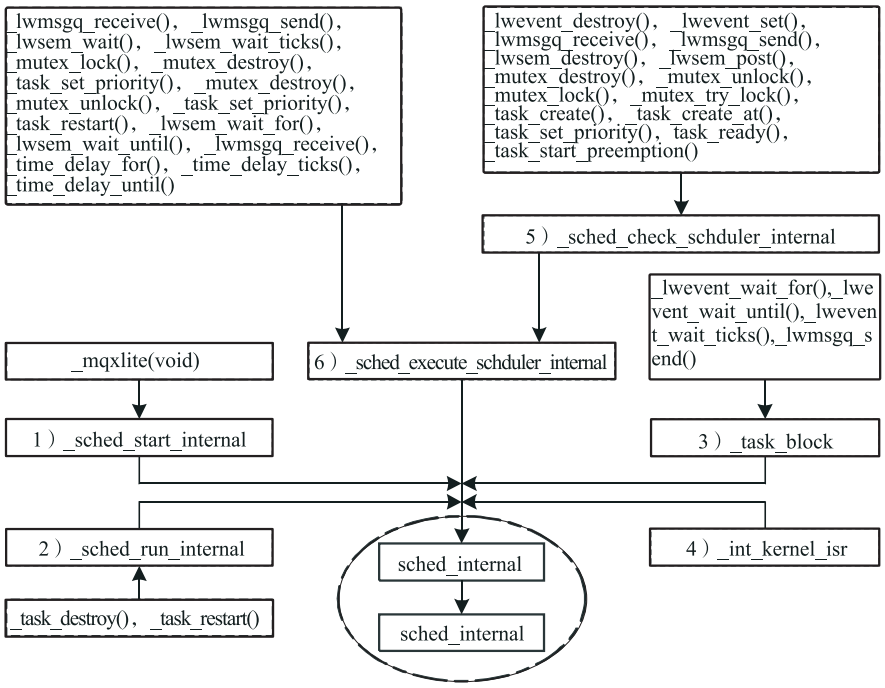

图1 MQXLite调度器

2.1 调度入口函数剖析

1）_sched_start_internal 函数用于 MQXLite 系统启动时，属于第 1 次启动调度系统无需返回的情况，由_mqxlite( ) 函数调用进入调度.

GET_KERNEL_DATA r0//r0←内核数据区首地址

b sched_internal //跳转到 sched_internal 处执行调度

2）_sched_run_internal 函数程序代码与_sched_start_internal 函数完全相同，主要用于任务销毁时或者重新开始一个任务的情况下.

3）_task_block 函数将当前任务从就绪队列中移除并置为 block 状态，随后调用_sched_internal 在就绪队列中找出新的最高优先级的任务进入运行态.

4）_int_kernel_isr 是内核中断服务例程. 在 MQXLite 中，除不可屏蔽中断 NMI、硬件错误、SVC 和 PendSV 中断依然由硬件机制管理以外，其余内核与外设中断均是由_int_kernel_isr 调用相应中断服务例程，每次产生中断进入_int_kernel_isr 前由硬件机制执行保存上下文操作. 中断处理完成后从用户中断服务例程返回_int_kernel_isr，_int_kernel_isr 都要检查是否有比进入中断前运行任务更高优先级的任务存在，如不存在更高优先级任务则返回原任务继续执行；如果存在比进入中断前优先级更高的任务，_int_kernel_isr 执行 b sched_internal 开始调度并不再返回. _int_kernel_isr 函数中与调度相关代码如下：

……

//如果在当前 readyq 的头部有不同的 TD，那么需要运行调度程序：

ldr r1，[r3，#KD_CURRENT_READY_Q]

ldr r1，[r1] //r1←最高优先级就绪任务指针

cmp r1，r2 //比较最高优先级就绪任务指针与当前激活态任务指针

beq_isr_return_end //没有更高优先级就绪任务，返回原任务继续执行

……

b sched_internal //执行调度

_sched_start_internal，_sched_run_internal，_

task_block，_int_kernel_isr 这 4 个函数都通过执行“b sched_internal”实现任务调度，其共同点是当前任务已执行完毕，无需保存当前任务现场，不会返回.

5）_sched_check_schduler_internal 通过调用_sched_execute_schduler_internal 函数实现调度，_sched_check_schduler_internal 的功能有两个：(a) 检查当前是否处于用户中断服务例程中，MQXLite 不允许在用户中断服务例程中进行调度；(b) 判断当前任务是否还是最高优先级就绪任务. 如果当前处于用户中断服务例程中，调度就不会立刻执行，而是执行 bx lr 返回用户中断服务例程继续执行. 当用户中断服务例程执行完毕以后，返回到内核中断服务例程_int_kernel_isr 后由_int_kernel_isr判断执行调度；如果当前任务并没有处于用户中断服务例程中且有更高优先级任务出现，就执行 b _sched_execute_schduler_internal，开始执行保存现场、关闭总中断操作，随后开始执行调度操作.

6）_sched_execute_schduler_internal 函数的功能是保存当前运行任务的上下文并关闭总中断. 随后执行 sched_internal 开始调度任务. 其运行场景是当前任务需要返回，也就是属于周期性任务或资源驱动任务.

```
STORE_ALL_REGISTERS//保存上下文
cpsid i //关闭中断
GET_KERNEL_DATA r0
ldr r3，［r0，#KD_ACTIVE_PTR］//获取激活态任务栈地址
str r1，［r3，#TD_STACK_PTR］//保存 SP 寄存器内容到任务栈
```

在 MQXLite 中，_sched_execute_schduler_internal 的执行场景有两种：一种是无法判断当前任务是否是用户中断服务例程，就首先执行_sched_check_scheduler_internal 判断当前状态，符合要求再进入_sched_execute_schduler_internal 函数执行；另一种是确定当前没有处于用户中断服务例程中，就直接执行_sched_execute_schduler_internal 进行调度.

2.2　调度内核 sched_internal 和 switch_task 剖析

sched_internal 和 switch_task 是 MQXLite 调度器的核心，MQXLite 所有任务的同步都是通过 sched_internal 和 switch_task 实现的.

当 MQXLite 通过入口函数进入 sched_internal 开始调度，sched_internal 首先在就绪队列中查找是否有新的就绪任务，如果找到新的就绪任务，就进入 switch_task，开始任务的切换，如果没有找到，则选择下一优先级就绪队列查找，周而复始，直到找到新的最高优先级就绪任务. MQXLite 根据用户配置的最低优先级 N 设置了 $N+2$ 个就绪队列“0 ~ $N+1$”，其中优先级为 $N+1$ 的队列是一个空队列，其 NEXT_Q 字段为 0，当查找过程中发现 NEXT_Q 字段为零，表明当前已查找到队尾，没有就绪任务需要运行，系统会进入休眠状态等待中断或事件唤醒. 当被唤醒以后，重新从最高优先级的队列开始新的查找. 由于优先级为 N 的队列始终有空闲任务等待运行，实际不会查找到队尾. 当找到新的就绪任务后，MQXLite 开始执行 switch_task. 详细流程见图 2.

当启动调度开始切换任务时，需要恢复一系列相应的寄存器，MQXLite 在创建任务时会调用_psp_build_stack_frame 函数来创建任务堆栈，同时设置了任务描述符的 STACK_BASE，STACK_LIMIT 和 STACK_PTR 这 3 个成员，并且填充了其中的 PSP_STACK_START_STRUCT 区域，而该区域保存了任务运行时的寄存器信息，通过这个函数，填充了本应自动操作的自动堆栈部分以及手动堆栈部分. 这样当任务被调度时，就从我们手工设置的值进行恢复，而当执行任务切换的时候，程序从任务堆栈中将 r4-r11 弹出堆栈，再从寄存器 control 中判断当前使用的是 MSP（中断返回）还是 PSP（任务返回），如果是从中断返回就将 R0 的栈地址赋给 PSP，此时 PSP 便指向了 R0，接着还原 PRIMASK 寄存器，将 PC 值出栈，激发内核将 R0 ~ R3，R12，LR，PC，xPSR 这 8 个寄存器弹出，这样之后 PC 等于（template_ptr - > TASK_ADDRESS）| 1，这里 TASK_ADDRESS 是任务函数，R0 是任务函数的参数 create_parameter，返回地址为_task_exit_function_internal（任务退出代码），此时便完成了第 1 次启动调度时的任务切换操作. 如果是运行中的正常任务切换，那么会调用 rest_of_stack_frame_restore，继续手动完成剩余 xPSR，PC，LR，R12，R3-R0 和 PRIMASK 的还原工作，最后当前任务（最高优先级任务）开始执行，这样就完成了任务的切换.

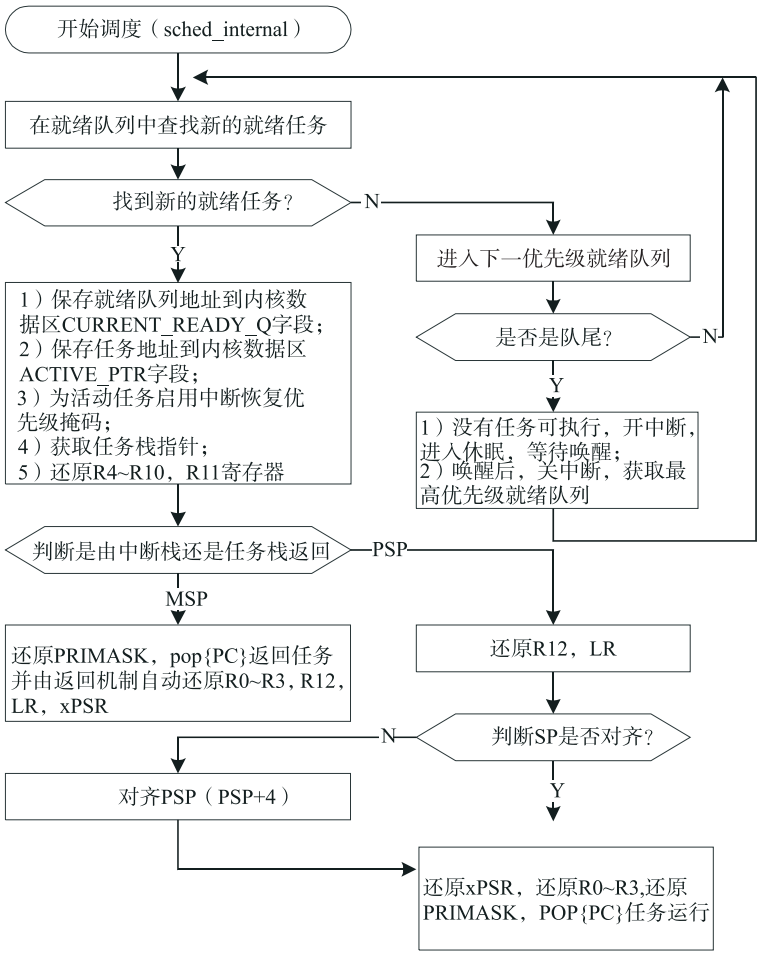

图2 调度流程图

3 MQXLite 任务延时同步分析

MQXLite 默认在启动时已初始化延时等待队列. 任务调用延时函数时，延时函数会将任务从就绪队列中转移到延时等待队列中，随后进行调度，让新的就绪任务执行. 下面以_time_delay_tick（）函数为例进行说明.

_time_delay_tick（）函数执行时，首先获取当前任务的任务描述符：

td_ptr = kernel_data -> ACTIVE_PTR;

随后将延时长度记录到当前任务描述符的 TIMEOUT 字段：

PSP _ ADD _ TICKS _ TO _ TICK _ STRUCT (&kernel_data -> TIME，time_in_ticks，&td_ptr -> TIMEOUT)；

最后调用_time_delay_internal（）函数：

_time_delay_internal（td_ptr）；

_time_delay_internal（）完成的工作有：

1）将当前任务描述符中的 TIMEOUT 字段与延时等待队列中的歌任务描述符中的 TIMEOUT 字段进行比较，找到当前任务描述符在延时等待队列中的位置.

2）将当前任务描述符从就绪队列中删除.

_QUEUE_UNLINK（td_ptr）；

3）将当前任务描述符插入到延时等待队列中并给其 STATE 字段置位.

_QUEUE_INSERT（&kernel_data -> TIMEOUT_QUEUE，tdprev_ptr，td_ptr）；

td_ptr -> STATE | = IS_ON_TIMEOUT_Q；

4）执行调度.

_sched_execute_scheduler_internal（）；

这样，当前任务就会从激活态被转移到阻塞态，新的最高优先级的任务就会被调度执行. 由于 MQXLite 操作系统有一个时钟嘀嗒 SysTick，当 SysTick 中断产生以后 MQXLite 就调用内核中断服务例程_int_kernel_isr 进行处理，_int_kernel_isr 最

后调用 SysTick 中断服务例程_time_notify_kernel（）进行处理．下面是_time_notify_kernel（）的处理过程．

（a）刷新当前系统时间：

PSP_INC_TICKS（&kernel_data - >TIME）；

（b）正常化系统时钟嘀嗒：

PSP_NORMALIZE_TICKS（&kernel_data - >TIME）；

（c）检查延时等待队列，如存在需要唤醒任务，就将该任务转移到就绪队列等待调度：

_TASK_READY（td_ptr，kernel_data）；

当_time_notify_kernel（）执行完毕返回，内核中断服务例程_int_kernel_isr，判断到就绪队列发生变化，就执行 b sched_internal 开始调度．这样进入延时的任务就重新开始执行．

4　MQXLite 任务调度实例分析

下面以苏州大学开发的 MQXLite 工程框架为例，探讨任务调度的具体实现．MQXLite 工程框架中，安排了 3 个任务：task_main（），task_light（）和 task_uart（），其任务优先级分别为7，9，8. 在 main（）函数调用_mqxlite（）进行初始化时，_mqxlite（）会将具有 MQX_AUTO_START_TASK 属性的 task_ main（）任务放入就绪队列中，在_mqxlite（）完成所有工作后，调用_sched_start_internal（）函数开始执行调度，不再返回．

进入 sched_internal 后，调度器会在优先级为 7 的就绪队列中查找到 task_main（）任务，于是 task_main（）得以执行．在 task_main（）任务中，通过_task_create_at（）函数开始创建task_light（）任务和 task_uart（）任务．_task_create_at（）函数创建 task_light（）任务后，会立即调用_CHECK_RUN_SCHEDULER 宏，其负责检查系统是否处于中断状态中，如果处于中断状态，则暂停调度；如不在中断状态，则检查是否有比当前就绪队列优先级更高的非空就绪队列存在．由于 task_light 任务优先级低于 task_main 任务，则 task_ main 任务继续执行，再次由_task_create_at（）函数创建 task_uart 任务并再次执行_CHECK_RUN_SCHEDULER（）进行调度检查．由于 task_uart 任务优先级也小于 task_main 任务，task_main 任务再次继续执行．随后 task_main 任务安装 UART0 的用户中断服务例程，使能 UART0 中断，直至最后调用_task_block（）函数将 task_main 任务从就绪队列中移除，并置为 block 状态，执行 b sched_internal，至此，操作系统开始执行调度．sched_internal 从最高优先级队列开始寻找存在的最高优先级就绪任务．此时会使 task_uart 任务运行．

task_uart 任务中调用_lwevent_wait_ticks（）函数，该函数最终会调用_time_delay_internal 函数将 task_uart 任务放到延时等待队列，并调用_sched_execute_scheduler_internal（）开始新的调度．

此时，task_main 任务已被阻塞，task_uart 任务被放到延时等待队列，就绪队列里只有 task_light 任务，因此 task_light 任务得以运行．当 task_light 任务运行到_time_delay_ticks（）函数时，该函数将 task_light 任务按延时的时钟滴答数插入到延时等待队列，并调用_sched_execute_scheduler_internal（）开始新一轮的调度．由于 task_uart 任务和 task_light 任务都处于延时等待队列中，调度器最终会调用空闲任务运行．

如果没有中断，MQXLite 将在空闲任务中一直运行下去．但该工程框架例程中有 SysTick 和 UART0 两个中断可用．假设 UART0 一直没有接收中断产生，则 task_uart 任务就一直不会运行．但系统每隔 5 ms 产生一个 SysTick 中断，SysTick 的中断服务例程_time_notify_kernel（）会被内核中断服务例程_int_kernel_isr 调用，_time_notify_kernel（）检查延时等待队列中的 task_light 任务的延时时间是否完成．如果完成，_time_notify_kernel（）就将该任务从延时等待队列转移到就绪队列中，当_time_notify_kernel（）执行完毕返回到_int_kernel_isr 中时，_int_ kernel_isr 执行“b sched_internal”调度器使得 task_light 任务得以继续运行，小灯状态改变，随后 task_light 继续调用_time_delay_ticks（）函数，把 task_ light 任务再次放到延时等待队列，并调用_sched_ execute_scheduler_internal（）开始新一轮的调度，如果此时 UART0 还没有接受中断产生，就继续进入空闲任务，直到 SysTick 中断再次产生，task_light 任务再次运行，周而复始．

如果过程中产生了 UART0 的接收中断，则_int_

kernel_isr 会调用 UART0 的中断服务例程 UART0_IRQHandler()运行，当 UART0_IRQHandler()运行到_lwevent_set()函数时，该函数会将轻量级事件 lwevent_group1 的 Event_UART0_ReData 事件位写 1，并将等待该事件位的 task_uart 任务从延时等待队列中移到就绪队列，随后立即调用_sched_check_schduler_internal 开始调度检查，当其发现当前处于用户中断服务例程，就返回中断服务例程 UART0_IRQHandler()继续执行．返回到_int_kernel_isr 以后，_int_kernel_isr 开始检查是否存在比当前运行任务（task_light 任务或空闲任务）更高优先级的任务存在．由于 task_uart 任务优先级高于 task_light 任务和空闲任务的优先级，_int_kernel_isr 立即执行“b sched_internal”调度器调用 task_uart 任务运行．当 task_uart 再次执行到lwevent_wait_ticks()函数后，就再次进入延时等待队列等待．此过程也周而复始．

5 结语

在物联网系统应用开发中，相比无操作系统的应用开发，带嵌入式操作系统的编程可以使得开发者只需关注各个分解任务的实现，而不用关心任务间的逻辑关系．因此，本文借助 MQXLite 嵌入式操作系统，分析了任务调度和同步的实现方法，并阐述了其实现机制．此外，对于实际的物联网底层开发而言，深入理解嵌入式操作系统任务调度和同步的方法，可以使物联网底层开发达到事半功倍的效果．

［参考文献］

［1］FREESCALE. Freescale MQX Lite RTOS reference manual［EB/OL］．［2017－06－12］．http://www.freescale.com/mqx.

［2］陆伟，张龙妹．嵌入式操作系统混合任务调度技术与策略研究［J］．计算机工程与应用，2015，51（15）：6－11.

［3］董艳铃，常扬．Vxworks 嵌入式实时操作系统任务调度方法研究［J］．计算机与网络，2014，40（10）：55－58.

［4］邵志勇，张学东，马丁．μC/OS-Ⅱ实时操作系统任务调度的改进［J］．鞍山科技大学学报，2003（5）：355－359.

［5］王宜怀，朱仕浪，姚望舒．嵌入式实时操作系统 MQX 应用开发技术：ARM Cortex-M 微处理器［M］．北京：电子工业出版社，2014.

（上接第 20 页）

［13］徐金杰，武忠．基于 AHP 和 DEMATEL 法的技术创新网络知识转移研究：以江苏省风电产业技术创新联盟为例［J］．情报杂志，2012（9）：121－125.

［14］祝明亮．烟草调制期间微生物研究进展［J］．微生物学通报，2008，35（8）：1278－1281.

［15］姚恒．烘烤对烟叶微生物种群的影响［J］．安徽农业科学，2010，38（29）：16166－16168.

［16］张成省，王海滨，李更新，等．仓储片烟霉变的影响因素分析［J］．中国烟草科学，2011，32（3）：80－83.

［17］孔凡玉，林建胜，张成省，等．储烟霉变机理与防霉技术研究进展［J］．中国烟草学报，2009，15（5）：78－81.

［18］瞿英，路亚静，刘紫玉，等．基于 AHP-DEMATEL 法的权重计算方法研究［J］．数学的实践与认识，2016，46（7）：38－46.