RT-Thread 编程指南

——*对象设计及线程设计*

2010.12.26

www.rt-thread.org

主要内容

- RT-Thread简介
- RT-Thread 的对象设计
- RT-Thread 的线程设计

RT-Thread 简介

- RT-Thread，本土化的开源实时操作系统：
  - 成立于2006年，完全本土化开源社区开发、维护
- 稳定性：
  - 30 x 24小时无差错运行
- 精巧、高效、稳定的实时内核，同时也是一款经过商业产品考验的实时内核
- 开放源代码，商业应用许可宽松，可免费地在商业产品中使用

RT-Thread 简介

- 实时核心—调度器
  - 基于优先级的全抢占式线程调度器：
    - 只要有高优先级线程就绪，立刻切换到高优先级线程执行；
    - 调度算法采用bitmap的方式，计算最高优先级线程时间恒定。
  - 最大支持256/32/8级线程优先级
  - 无限线程数，相同优先级线程采用时间片可设置的Round-robin算法。

RT-Thread 简介

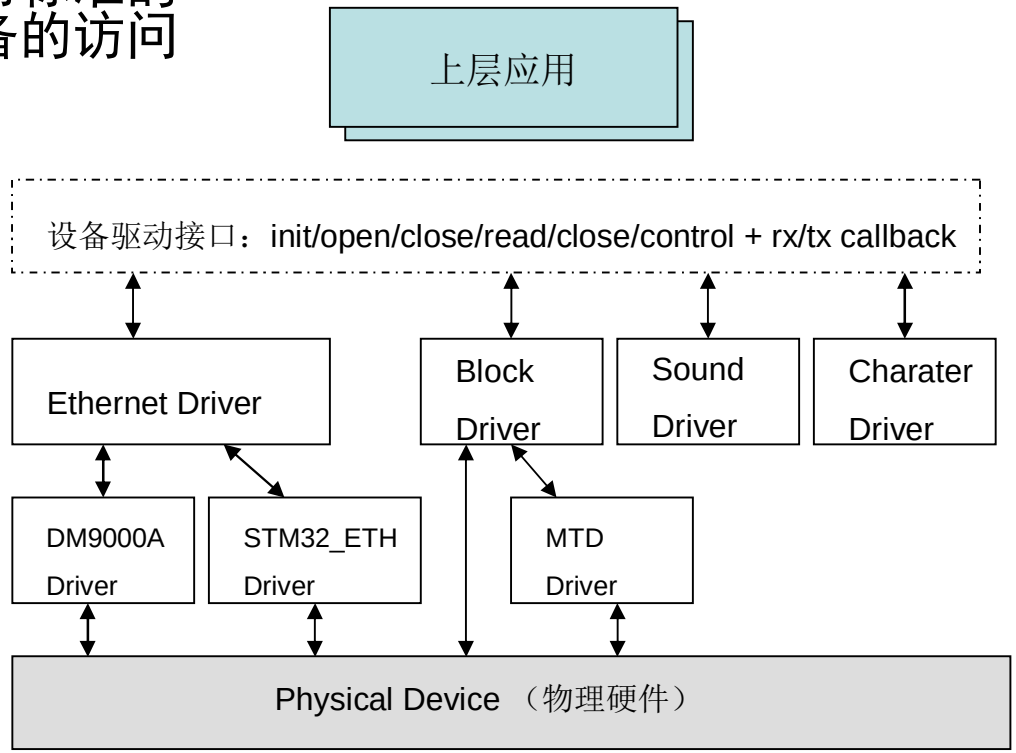
- 实时核心—同步、通信
  - 丰富的线程间同步机制：
    - 信号量、用于防止优先级翻转的互斥量、用于多事件触发的事件集；
  - 丰富的线程间通信机制：
    - 带缓存功能的邮箱通信和消息队列通信方式；
  - 发送、释放动作可安全的在中断服务例程中使用。
  - 当资源不可用时，线程可按照FIFO方式或优先级排队方式挂起在等待队列上。

RT-Thread 简介

- 实时核心—内存管理
  - 内存管理模块：
    - 带线程挂起功能的静态内存池：
      - 每个内存块大小固定；
      - 申请、释放内存块行为实时而高效。
      - 当内存池为空时，线程可选择是否休眠，当有内存块可用时，自动唤醒等待线程。
    - 多线程环境安全的动态堆内存管理。
  - 基本系统亦可脱离内存分配器而独立运行。

RT-Thread 简介

- 实时核心—设备驱动
  - 基于名字的对象化设备模型：
    - 上层应用只需查找相应设备名获得设备句柄即可采用标准的设备接口进行硬件设备的访问操作；
    - 通过这套设备模型，可以做到应用与底层设备的无关性。
    - 当前支持： 字符设备，块设备、网络设备、声音设备等。



RT-Thread 简介

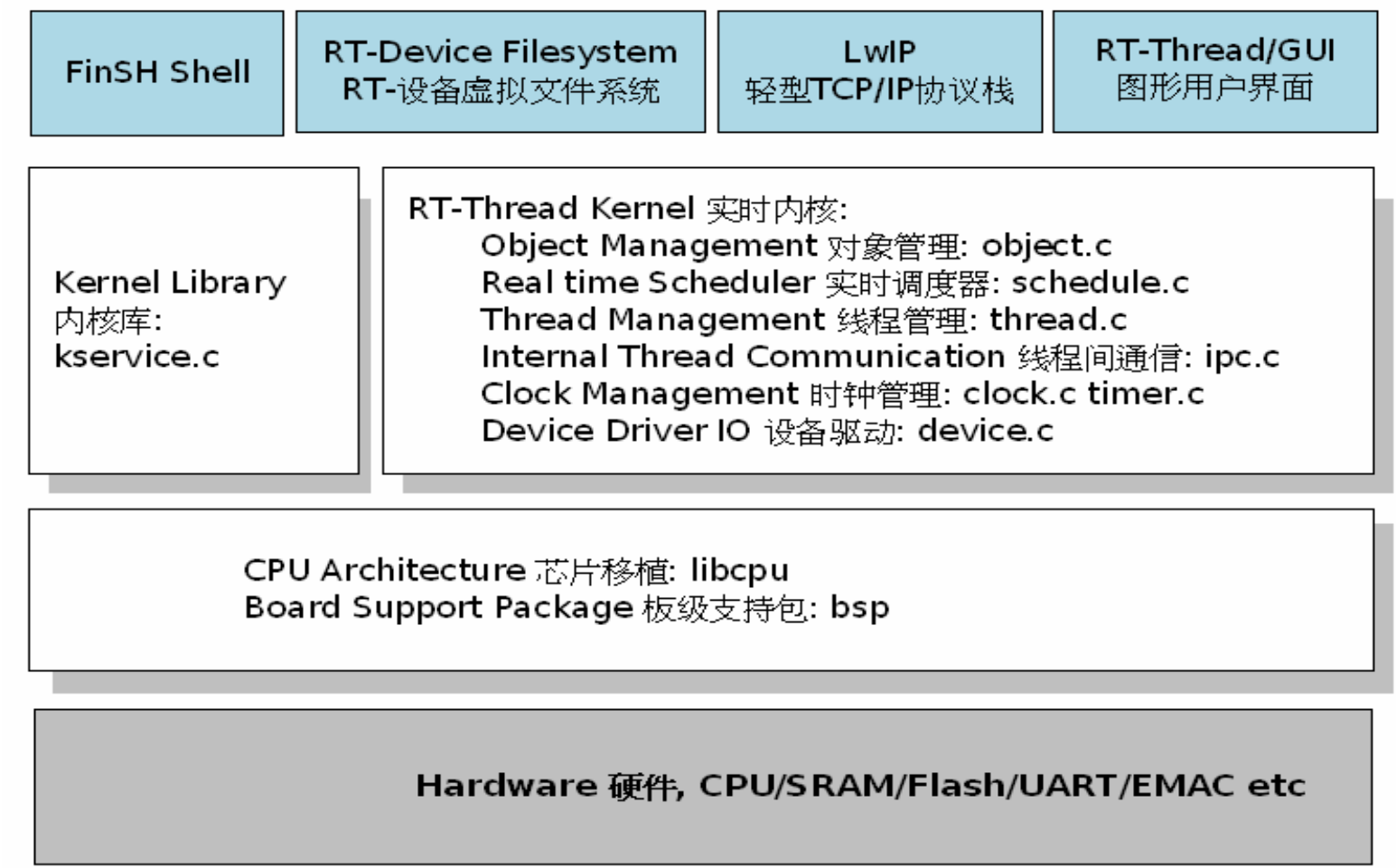
- 实时核心—基本指标
  - 基本内核配置资源占用情况：
    - 13344字节 ROM，1800字节 RAM
  - 简化版资源占用情况：
    - 2.5K ROM，1K RAM
  - 线程上下文切换时间（在72MHz的STM32上采用逻辑分析仪测得的结果）：
    - 挂起操作引起线程上下文切换：4.25 μs
    - 信号量引起线程上下文切换：7.25 μs
    - 邮箱引起线程上下文切换：8.63 μs

RT-Thread 简介

- 丰富外围组件
  - finsh shell
  - 设备虚拟文件系统
  - TCP/IP协议栈（LwIP TCP/IPv4）
  - RT-Thread/GUI

RT-Thread 简介

- RT-Thread结构



主要内容

- RT-Thread简介
- RT-Thread 的对象设计
- RT-Thread 的线程设计

RT-Thread 内核对象模型

- C语言的对象化模型

  面向对象的特征：

  - 封装 — 隐藏内部实现
  - 继承 — 复用现有代码
  - 多态 — 改写对象行为

RT-Thread 内核对象模型

- 封装
  - 命名习惯

    名词+动词
  - static 修辞

    对于某些方法，仅局限在对象内部使用，采用static修辞把作用范围局限在一个文件的内部
  - 对象的创建/析构
    - rt_sem_init / rt_sem_detach
    - rt_sem_create / rt_sem_delete

RT-Thread 内核对象模型

- 继承

  - 继承性是子类自动共享父类之间数据和方法的机制
  - RT-Thread支持单继承
  - 关键词：**结构体**、**指针**、**强制类型转换**

RT-Thread 内核对象模型

- 继承实现示例

```
/* 父类 */
  struct parent_class
  {
      int a, b;
      char *str;
  };
/* 继承于父类的子类 */
  struct child_class
  {
      struct parent_class p;
      int a, b;
  };
```

```
/* 操作示例函数*/
  void func()
  {
      struct child_class obj, *obj_ptr; /* 子类对象及指针 */
      struct parent_class *parent_ptr; /* 父类指针 */
      obj_ptr = &obj;
      /* 取父指针 */
      parent_ptr = (struct parent_class*) &obj;
      /* 可通过转换过类型的父类指针访问相应的属性 */
      parent_ptr->a = 1;
      parent_ptr->b = 5;
      /* 子类属性的操作 */
      obj_ptr->a = 10;
      obj_ptr->b = 100;
  }
```

RT-Thread 内核对象模型

– 多态

- 对象根据所接收的消息而做出动作。同一消息为不同的对象接受时可产生完全不同的行为（方法）

- 关键词：**函数指针**

RT-Thread 内核对象模型

- 多态 实现示例

```
#include <stdio.h>
#include <stdlib.h>

struct parent_class
{
    int a;
    int b;

    void (*vfunc)(struct parent_class* self, int parameter);
};
#define PARENT(obj)   (struct parent_class*)(obj)

struct child_class
{
    struct parent_class parent;
    int a;
};
#define CHILD(obj)   (struct child_class*)(obj)
```

RT-Thread 内核对象模型

```
/* 父类虚拟方法 */
static void _parent_vfunc(struct parent_class* self, int parameter)
{
    if (self != NULL)
    {
        self->a = self->b + parameter;
        printf("parent vfunction, a = %d\n", self->a);
    }
}

/* 父类方法 */
void parent_vfunc(struct parent_class* self, int parameter)
{
    if (self != NULL)
    {
        self->vfunc(self, parameter);
    }
}
```

RT-Thread 内核对象模型

```
/* 初始化parent对象 */
void parent_init(struct parent_class* self)
{
    if (self != NULL)
    {
        self->a = 0;
        self->b = 0;
        self->vfunc = _parent_vfunc;
    }
}

/* 输出parent中的a成员数值 */
void parent_a(struct parent_class* self)
{
    if (self != NULL)
    {
        printf("a = %d\n", self->a);
    }
}
```

RT-Thread 内核对象模型

```
/* child重载的虚方法 */
void child_vfunc(struct parent_class* self, int parameter)
{
    struct child_class* child;

    child = (struct child_class*)self;
    child->a = parameter;

    printf("child vfunction, a = %d\n", child->a);
}

/* 初始化child对象 */
void child_init(struct child_class* self)
{
    struct parent_class* parent;

    /* 初始化继承的父类对象 */
    parent = (struct parent_class*)self;
    parent_init(parent);

    /* 初始化其余对象 */
    self->a = 100;
    /* 重载父类方法 */
    parent->vfunc = child_vfunc;
}
```

RT-Thread 内核对象模型

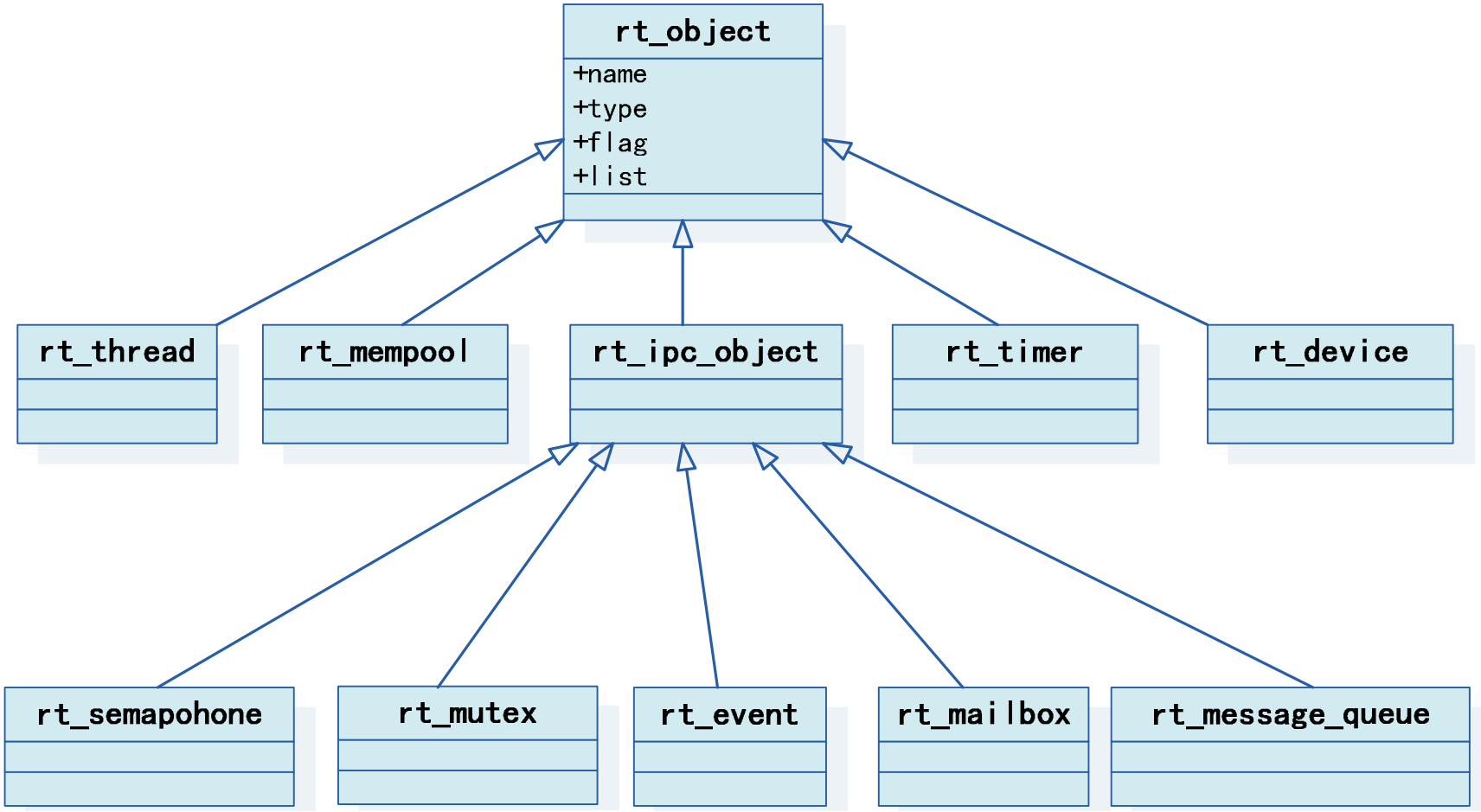
```
int main(int argc, char** argv)
{
    struct parent_class parent;
    struct child_class child;

    /* 初始化parent对象 */
    parent_init(&parent);
    /* 调用相应的方法 */
    parent_vfunc(&parent, 10);
    parent_a(&parent);

    /* 初始化child对象 */
    child_init(&child);
    /* 调用相应的方法 */
    parent_vfunc(PARENT(&child), 10);
    parent_a(PARENT(&child));
}
```

RT-Thread 内核对象模型

- RT-Thread中各类内核对象的派生和继承关系



RT-Thread 内核对象模型

- RT-Thread内核对象模型设计方法的优点

  - 提高了系统的可重用性和扩展性，增加新的对象类别很容易，只需要继承通用对象的属性再加少量扩展即可。

  - 提供统一的对象操作方式，简化了各种具体对象的操作，提高了系统的可靠性。

RT-Thread 内核对象模型

- 内核对象接口
  - 初始化系统对象

    ```
    void rt_system_object_init(void);
    ```

  - 初始化对象

    ```
    void rt_object_init(struct rt_object* object, enum
        rt_object_class_type type, const char* name);
    ```

  - 脱离对象

    ```
    void rt_object_detach(rt_object_t object);
    ```

  - 分配对象

    ```
    rt_object_t rt_object_allocate(enum rt_object_class type type, const
        char* name);
    ```

  - 删除对象

    ```
    void rt_object_delete(rt_object_t object);
    ```

  - 辨别对象

    ```
    rt_err_t rt_object_is_systemobject(rt_object_t object);
    ```

主要内容

- RT-Thread简介
- RT-Thread 的对象设计
- RT-Thread 的线程设计

RT-Thread 线程调度与管理

- 线程
  - 线程是RT-Thread中最基本的调度单位，它描述了一个任务执行的上下文关系，也描述了这个任务所处的优先等级。
  - RT-Thread中提供的线程调度器是基于全抢占式优先级的调度

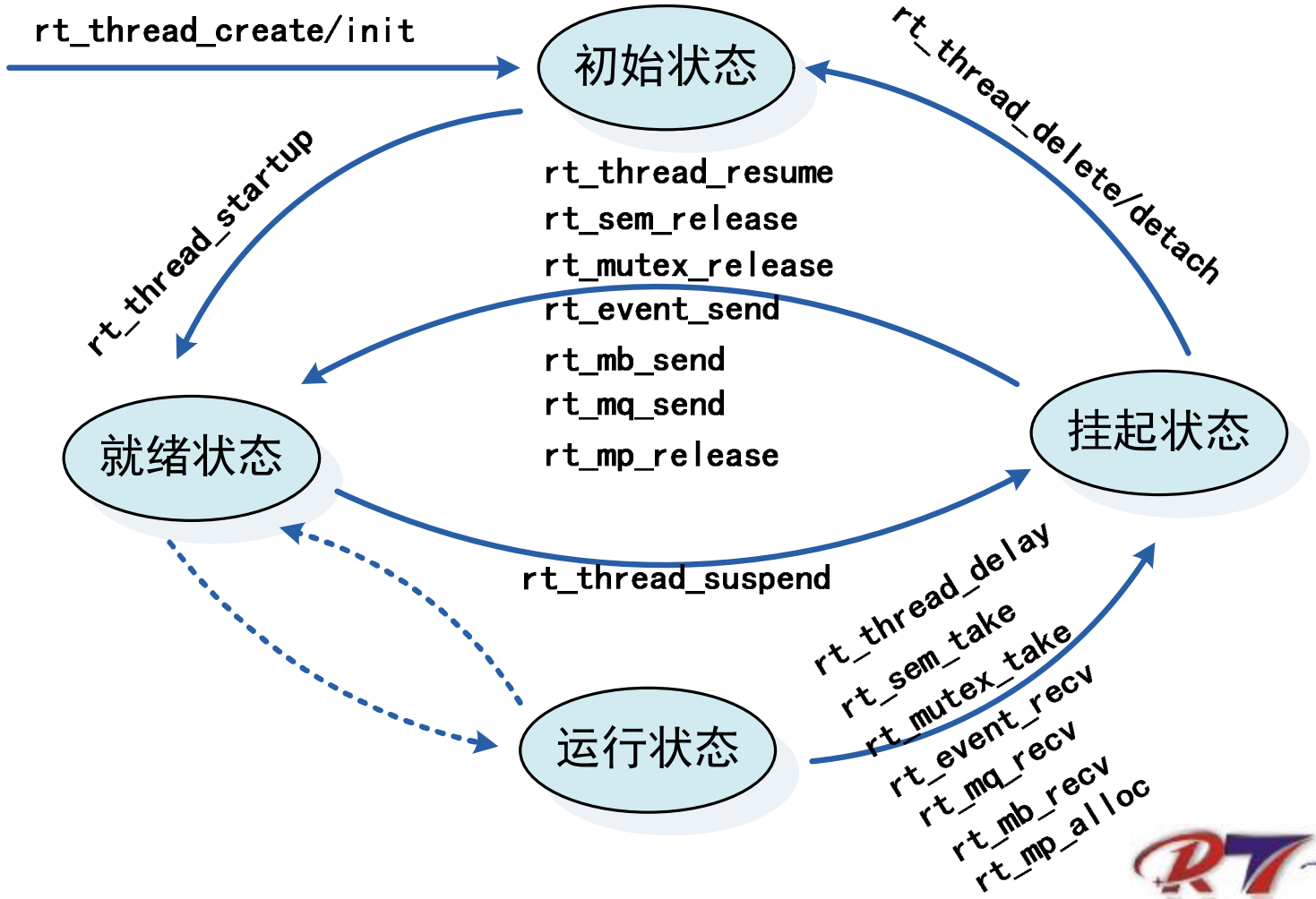
RT-Thread 线程调度与管理

- RT-Thread线程状态

状态	描述
RT_THREAD_INIT/CLOSE	线程初始状态。当线程刚开始创建还没开始运行时就处于这个状态；当线程运行结束时也处于这个状态。在这个状态下，线程不参与调度
RT_THREAD_SUSPEND	挂起态。线程此时被挂起：它可能因为资源不可用而等待挂起； 或主动延时一段时间而被挂起。在这个状态下，线程不参与调度
RT_THREAD_READY	就绪态。线程正在运行；或当前线程运行完让出处理机后，操作系统寻找最高优先级的就绪态线程运行
RT_THREAD_RUNNING	运行态。线程当前正在运行，在单核系统中，只有`rt_thread_self()`函数返回的线程处于这个状态；在多核系统中则不受这个限制。

RT-Thread 线程调度与管理

- 线程状态转换关系



RT-Thread 线程调度与管理

- 空闲线程 （idle）

  - 空闲线程是系统线程中一个比较特殊的线程，它具备最低的优先级，当系统中无其他线程可运行时，调度器将调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起

  - 在RT-Thread实时操作系统中空闲线程提供了钩子函数，可以让系统在空闲的时候执行一些特定的任务，例如系统运行指示灯闪烁，电源管理等。在允许钩子函数之外，RT-Thread把真正的线程删除动作也放到了空闲线程中（在删除线程时，仅改变线程的状态为关闭状态不再参与系统调度）。

RT-Thread 线程调度与管理

- 调度器接口
  - 调度器初始化

    ```
    void rt_system_scheduler_init(void);
    ```

  - 启动调度器

    ```
    void rt_system_scheduler_start(void);
    ```

  - 执行调度

    ```
    void rt_schedule(void);
    ```

  - 设置调度器钩子

    ```
    void rt_scheduler_sethook(void (*hook)(struct rt_thread* from,
        struct rt_thread* to));
    ```

RT-Thread 线程调度与管理

- 线程相关接口
  - 线程创建

```
rt_thread_t rt_thread_create(const char* name,
        void (*entry)(void* parameter), void* parameter,
        rt_uint32_t stack_size,
        rt_uint8_t priority, rt_uint32_t tick);
```

  - 线程删除

```
rt_err_t rt_thread_delete(rt_thread_t thread)
```

RT-Thread 线程调度与管理

- 线程初始化

```
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);
```

- 线程脱离

```
rt_err_t rt_thread_detach (rt_thread_t thread)；
```

RT-Thread 线程调度与管理

- 线程启动

```
rt_err_t rt_thread_startup(rt_thread_t thread) ;
```

- 当前线程

```
rt_thread_t rt_thread_self(void);
```

- 线程让出处理机

```
rt_err_t rt_thread_yield();
```

- 线程睡眠

```
rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);
```

- 线程挂起

```
rt_err_t rt_thread_suspend (rt_thread_t thread);
```

- 线程恢复

```
rt_err_t rt_thread_resume (rt_thread_t thread);
```

RT~Thread

RT-Thread 线程调度与管理

- 线程控制

  rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg);

  其中cmd参数指示出控制命令，当前支持的命令包括：

  - RT_THREAD_CTRL_CHANGE_PRIORITY - 动态更改线程的优先级
  - RT_THREAD_CTRL_STARTUP - 开始运行一个线程，等同于 rt_thread_startup() 函数调用。
  - RT_THREAD_CTRL_CLOSE - 关闭一个线程，等同于

    rt_thread_delete() 函数调用。arg参数必须是一个线程控制块指针。

- 初始化空闲线程

  void rt_thread_idle_init(void);

- 设置空闲线程钩子

  void rt_thread_idle_set_hook(void (*hook)(void));

RT-Thread 线程调度与管理

- 线程设计要点
  - 考虑上下文环境
    - 工作内容与工作内容间是否有重叠的部分，是否能够合并到一起进行处理，或者单独划分开进行处理。
  - 线程的状态跃迁
    - 线程不活跃的时候必须让出处理机，设计线程的时候就明确知道哪些点需要让线程从就绪态跃迁到阻塞态 。
  - 线程运行时间长度
    - 线程所关心的一种事件或多种事件触发状态下，线程由阻塞态跃迁为就绪态进行设定的工作，再从就绪态跃迁为阻塞态所需要的时间。

谢　谢

www.rt-thread.org

http://rt-thread.googlecode.com/