

RTX51-Tiny 使用

GARY Studio

有问题请联系 QQ:782387489

1、RTX51-Tiny 简介:

RTX51 Tiny Version 2 (版本 2.02) 是一个实时操作系统 (RTOS), 允许你设计实现在同时一时间完成多功能或者运行多任务的应用。在嵌入式应用当中, 往往会在没有 RTOS 的条件下实现一个特定的实时程序 (在一个单循环中实现一种或多种功能, 或者运行一个或多个任务); 这样的设计往往有很多资源分配, 运行时间以及程序维护的问题需要解决, 而像 RTX51 这样的 RTOS 就可以帮你解决这些问题。

一个实时操作系统 (RTOS) 可以更灵活有效地分配系统资源, 例如 CPU 和存储器, 同时也提供任务之间的通信。RTX51 Tiny 是一个功能强大且易于使用的 RTOS, 可以运行在所有 8051 以及其衍生产品上。

RTX51 Tiny 的程序设计可以使用标准 C 语言, 并可以用 Keil 的 C51 C 编译器进行编译。利用 C51 附加的特性, 你可以很容易声明任务函数, 却不需要复杂的堆栈以及变量帧的配置。RTX51 Tiny 的程序设计仅要求你包换一个特定的头文件以及正确连接 RTX51 的库到你的程序中。

1.1 新增的特性

RTX51 Tiny 版本 2 新增了很多特性, 使得实时软件的设计更容易, 例如:

➤ 支持代码分页

RTX51 Tiny 现在支持代码分页, 代码分页的支持是一个可选项, 需要通过配置文件 CONF_TNY.A51 进行使能。和所有的代码分页应用一样, 也必须通过文件 L51_BANK.A51 来配置代码分页的硬件操作。

➤ 直接任务切换

os_switch_task 是新增加的函数, 可以从一个任务立即切换到另外一个处于就绪 (READY) 状态的任务。

➤ 任务就绪标志

新的 RTX51 Tiny 库函数 isr_set_ready 和 os_set_ready, 让你可以给一个任务设置就绪 (READY) 标志。就绪 (READY) 标志将使一个任务立即处于就绪 (READY) 状态, 等待一个时间间隔 (interval)、超时 (timeout) 或信号 (signal) 事件 (参考函数 os_wait), 任务将在下一个时机唤醒进入运行。

➤ 支持 CPU 的 Idle 模式

RTX51 Tiny 现在允许在 Idle 任务里进入 CPU 的 Idel 模式 (许多器件都支持这个模式)。

➤ 支持在定时中断中添加用户代码

你现在可以在 RTX51 Tiny 的定时器中断中添加自己的代码, 并在你自己的程序中使用和 RTX51 Tiny 相同且固定的时钟频率。这是一个可选项, 需要通过配置文件 CONF_TNY.A51 来使能。

➤ 支持调整时间间隔

如果调用 `os_wait` 时，同时混合了时间间隔和信号两种事件，那么就可以通过函数 `os_reset_interval` 来调整一个时间间隔的超时值。

此外，RTX51 Tiny 已经彻底调整结构获得了更多的灵活性，提高了性能并且减少了代码和数据的空间需求。如果符合以下的条件，RTX51 Tiny 版本 2 是可以裁剪的，代码量将显著缩减。

- (1) 禁止时间轮转的任务切换；
- (2) 使用 RTX51 Tiny 新的系统函数；
- (3) 禁止堆栈检查；

1.2 解决的问题

下面的列表是 RTX51 Tiny 版本 1.06 中发现的问题，这些问题在版本 2 中已经被修正。

(1) RTX51 Tiny 版本 1.06 中，如果 `os_wait` 期间有中断产生，那么函数 `isr_send_signal` 将会破坏就绪 (READY) 状态，这将导致任务被挂起，在中断中等待信号。

(2) RTX51 Tiny 版本 1.06 中，在 `os_wait` 函数参数中不能组合 `K_IVL` 和 `K_SIG` 这两种事件条件，因为当信号事件产生时没有办法调整定时器的时间间隔。版本 2 提供了函数 `os_reset_interval`，允许你正确调整定时器的时间间隔。

(3) RTX51 Tiny 版本 1.06 中，针对时间轮转调度的参数 `TIMESHARING` 不能设置为 1，如果在时间轮转调度发生之前有中断产生，那么这个时间轮转调度的周期将被破坏，由 1 变成了 256。

(4) RTX51 Tiny 版本 1.06 中，当用户的中断函数执行时间大于 RTX51 Tiny 的系统时钟周期时，RTX51 Tiny 的系统定时器将会发生递归调用，从而导致 `SAVEPSW` 和 `SAVEACC` 的值被重写，典型的现象就是导致系统崩溃，版本 2 已经解决这个问题。如果你的应用程序中包含了一个执行时间大于 RTX51 Tiny 的系统时钟周期，那么要将 `LONG_USR_INTR` 的选项置 1。如果你的程序中，优先级高的中断函数消耗了比较多的时间，那么也应该把 `LONG_USR_INTR` 置 1。

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

1.3 产品规格

产品参数	限制值
最大可定义的任务数	16
最大激活的任务数	16
需要的 CODE 空间	900 字节（最大）
需要的 DATA 空间	7 字节
需要的 STACK 空间	3 字节/任务
需要的 XDATA 空间	0 字节
需要的定时器	定时器 0
系统时钟的机器周期数	1,000-65,535
中断响应时间	20 机器周期（最大）
上下文切换时间	100-700 机器周期

1.4 工具需求

RTX51 Tiny 需要以下的软件工具：

- ◆ C51 编译器
- ◆ A51 宏汇编器
- ◆ BL51 或者 LX51 连接器

库文件 RTX51TNY.LIB 和 RTX51BT.LIB 必须保存在编译器的库路径中，通常就是这个文件夹：\KEIL\C51\LIB

头文件 RTX51TNY.H 也必须保存在 include 的路径中，通常是：\KEIL\C51\INC

1.5 目标系统要求

程序可能会使用到外部数据存储器，不过内核没有这个需求。RTX51 Tiny 可以运行在所有被 Keil C51 编译所支持的寄存器模式上，寄存器模式的选择仅仅影响应用程序目标的定位而已。RTX51 Tiny 的系统变量和应用程序的堆栈区总是位于 8051 的内部数据存储器里（DATA 或者 I DATA）。典型的应用，RTX51 Tiny 的应用程序往往选择 SMALL 这种内存模式。

RTX51 Tiny 采用合作式的任务调度（每一个任务都会调用一个内核函数）和时间轮转的任务调度（每一个任务在操作系统切换到下一个任务之前都运行一个固定的时间）。不支持有优先权的任务调度以及任务的优先级，如果你的应用需要使用到有优先权的任务调度那么你需要使用 RTX51 Full 实时操作系统。

1.5.1 中断

RTX51 Tiny 和你的中断程序是工作在并行的模式下，中断服务程序通过发送信号（通过调用内核函数 `isr_send_signal`）和设置就绪标志（通过调用内核函数 `isr_set_ready`）的方式和 RTX51 Tiny 的任务进行通信。

RTX51 Tiny 没有中断程序的管理能力，所以在你得 RTX51 Tiny 应用程序中要管理中断程序的使能和运行等。

RTX51 Tiny 使用了定时器 0、定时器 0 的中断和寄存器组 1。如果你的程序使用了定时器 0，将会导致 RTX51 Tiny 内核工作不正常。当然，你也可以把你的定时器 0 中断服务程序添加到 RTX51 Tiny 的定时器 0 中断服务程序之后，细节请参考硬件定时器章节。

RTX51 Tiny 假定系统中断使能控制位总是处于允许的状态（EA = 1），RTX51 Tiny 库函数根据需要来改变系统中断的使能控制，目的是为了保护内核的结构不被中断所破坏；不过 RTX51 Tiny 的控制方式比较简单，并没有对 EA 的状态进行保存或者恢复。如果你的程序在调用 RTX51 Tiny 内核函数之前禁止 EA，那么内核将会失去响应。

通常不允许在你的程序访问临界区时，短暂禁止中断；如果必须要禁止中断，那么要确保禁止的时间很短，同时在中断禁止期间不允许调用任何内核函数。

1.5.2 重入函数

C51 编译器支持重入函数，通过将参数和局部变量保存到重入栈中来实现重入，支持重入的函数可以被递归调用或者同时调用。不过 RTX51 Tiny 不包含任何对 C51 重入栈的管理，因此只能在你的应用程序中使用重入函数，要确保重入函数不调用任何 RTX51 Tiny 的内核函数同时也不被时间轮转任务调度所中断。

如果 C 函数仅仅使用了寄存器参数以及自动变量，这是天然的可重入函数，在 RTX51 Tiny 系统中调用没有任何限制。

非重入 C 函数不能被超过一个以上的任务或者中断程序调用，因为函数的参数和局部变量是保存在静态存储区的，一旦发生多个任务同时调用或者递归调用，就会导致参数和局部变量被覆盖。

如果你要从多任务中调用非重入函数，你必须要确保不会发生递归或者同时调用的情况发生。通常，这意味着你禁止了时间轮转任务调度同时非重入函数也不调用任何的内核服务。

注意：

不论你在多个任务或者中断程序中调用重入或者非重入函数，你最好是禁止时间轮转的任务调度，改用合作式任务调度。

1.5.3 C 库函数

C51 库函数中，支持重入的函数可以在任何任务中调用，没有限制；非重入 C51 库函数的限制和普通 C 的非重入函数是一样的。

1.5.4 多数据指针

Keil C51 编译器允许你使用多数据指针（许多 8051 衍生产品都有支持多数据指针），但是 RTX51 Tiny 并不支持对多数据指针的管理，因此在 RTX51 Tiny 应用上如果要使用多数据指针，那么要非常小心。

本质上，你要确保时间轮转任务调度不会发生在数据指针选择器改变的地方。

注意：

如果你想要使用多数据指针，那么你应该禁止时间轮转任务调度，改用合作式任务调度。

1.5.5 算术单元

Keil C51 编译器允许你使用算术单元（许多 8051 衍生产品都有支持算术单元），但是 RTX51 Tiny 并不支持对算术单元的管理，因此在 RTX51 Tiny 应用上如果要使用算术单元，那么要非常小心。

本质上，你要确保时间轮转任务调度不会发生在使用算术单元的地方。

注意：

如果你想要使用算术单元，那么你应该禁止时间轮转任务调度，改用合作式任务调度。

1.5.6 寄存器组

RTX51 Tiny 假定所有的任务都使用寄存器组 0，正因为这样所有的任务函数都必须采用默认的 C51 设定来编译（REGISTERBANK(0)）。

中断程序应该可以使用剩下的寄存器组，不过 RTX51 Tiny 在寄存器组中占用了 6 个固定的寄存器，RTX51 Tiny 使用哪一个寄存器组可以由配置文件来设定（CONF_TINY.A51）。

2、实时程序设计

实时程序设计必须要求能够快速响应实时事件，如果处理的事件比较少，那么在没有实时操作系统的条件下也是比较容易实现的。一旦处理的事件增加，那么程序的复杂性和不确定性就大大增加，这个时候就需要从操作系统上获益了。

2.1 单任务系统

嵌入式系统或者是标准 C 程序设计，程序都是从 `main` 函数开始。在嵌入式系统程序中 `main` 函数通常是一个死循环，也可以被看作是一个连续运行的单一任务，下面就是一个例子。

```
void main (void)
{
while (1)                /* repeat forever */
{
    do_something ();      /* execute the do_something 'task' */
}
}
```

在这个例子中，函数 `do_something` 可以被看作是一个单任务，因为这里只有一个需要执行的任务，因此不需要多任务能力或者一个多任务操作系统。

2.2 多任务程序设计

许多经典的 C 程序也可以实现一个假的多任务，在这里一个循环中调用了几个函数（或者是任务），下面就是一个例子：


```
void main (void)
{
    int counter = 0;

    while (1)                /* repeat forever */
    {
        check_serial_io ();    /* check for serial input */
        process_serial_cmds (); /* process serial input */

        check_kbd_io ();       /* check for keyboard input */
        process_kbd_cmds ();    /* process keyboard input */

        adjust_ctrlr_parms (); /* adjust the controller */

        counter++;             /* increment counter */
    }
}
```

在这个例子中，每一个函数完成一个独立的操作或者任务，这些函数（或者任务）以一定的顺序执行，一个接着一个。

一旦增加更多的任务，那么执行的顺序就变成了一个问题。在以上的例子中，一旦函数 `process_hbd_cmds` 运行了太长的时间，那么主循环就需要很长的时间才可以执行到 `check_serial_io`，串行输入的数据就很有可能丢失。当然，我们也可以在循环中插入更多的 `check_serial_io` 函数调用，但是最终这样的技术也不能很好地解决问题。

2.3 RTX51 Tiny 程序设计

如果你采用 RTX51 Tiny，那么在你的应用中就可以为每一个任务创建一个独立的任务函数，以下就是一个例子。

```
void check_serial_io_task (void) _task_ 1
{
    /* This task checks for serial I/O */
}
```



```
void process_serial_cmds_task (void) _task_ 2
{
/* This task processes serial commands */
}

void check_kbd_io_task (void) _task_ 3
{
/* This task checks for keyboard I/O */
}

void process_kbd_cmds_task (void) _task_ 4
{
/* This task processes keyboard commands */
}

void startup_task (void) _task_ 0
{
os_create_task (1);    /* Create serial_io Task */
os_create_task (2);    /* Create serial_cmds Task */
os_create_task (3);    /* Create kbd_io Task */
os_create_task (4);    /* Create kbd_cmds Task */

os_delete_task (0);    /* Delete the Startup Task */
}
```

在这个例子中，每一个函数定义为一个 RTX51 Tiny 的任务，在 RTX51 Tiny 程序设计中不需要一个 main 函数，其总是从任务 0 开始执行。一个典型应用，任务 0 只是简单产生所有其他的任务，然后将自己删除。

3、操作原理

RTX51 Tiny 使用和管理你的目标系统资源，这一章将讨论这些资源是如何被 RTX51 Tiny 使用和管理的。

依据每一个不同的项目，RTX51 Tiny 将被配置成不同的样子，这里将说明可能的处理。

3.1 定时中断

RTX51 Tiny 使用标准 8051 的定时器 0（模式 1）去产生一个周期性的中断，该中断就作为 RTX51 Tiny 的时钟，RTX51 Tiny 库函数所指定的超时和时间间隔参数都是利用 RTX51 Tiny 时钟来测量的。

RTX51 Tiny 默认的时钟中断是 10000 个机器周期，因此对于一个运行在 12MHz 时钟标准 8051 而言，内核的时钟就是 100Hz（周期 0.01 秒），计算公式就是： $12\text{MHz}/12/10000$ 。这个值可以通过配置文件 CONF_TNY.A51 来更改。

注意：

- 你可以增加自己的定时中断代码到 RTX51 Tiny 的定时中断服务程序中，更多的信息可以参考配置文件 CONF_TNY.A51
- 参考概述一章中关于中断一节的讨论，可以获得更多关于 RTX51 Tiny 如何使用中断系统的细节。

3.2 任务

RTX51 Tiny 基本上可以看作是一个任务切换器，因此要创建一个 RTX51 Tiny 程序，你就必须在一个应用程序中包含至少一个任务函数。以下的细节可以帮助你快速理解 RTX51 Tiny。

- ◆ Keil C51 编译器引进了一个新的关键字（_task_），用来在 C 语言程序中进行任务定义；
- ◆ RTX51 Tiny 严密维护每一个任务在某一个状态（运行<Running>，就绪<Ready>，等待<Waiting>，删除<Deleted>或者超时<Time-Out>）；
- ◆ 任何时候就只有一个任务处于运行状态；
- ◆ 大多数任务处于就绪<Ready>，等待<Waiting>，删除<Deleted>或者超时<Time-Out>状态；
- ◆ 如果一旦出现你所定义的所有任务都处于阻塞状态，那么就会运行一个 Idel 任务；

3.3 任务管理

每一个 RTX51 Tiny 任务总是被严密维护在一个状态上（下表将列出所有可能的任务），任务的状态将传递给任务调度器。

任务状态	描 述
RUNNING (运行)	当前正在运行的任务就是处于运行 (RUNNING) 状态, 而且所有任务中仅有一个任务处于运行状态。内核函数 <code>os_running_task_id</code> 返回的是处于运行状态的任务的任务号。
READY (就绪)	准备好运行的任务就处于就绪 (READY) 状态, 一旦处于运行状态的任务处理完毕, RTX51 Tiny 将开始运行下一个处于就绪状态的任务。如果使用函数 <code>os_set_ready</code> 和 <code>isr_set_ready</code> , 那么一个任务会立即变成就绪状态 (即使这个任务还在等待超时或者信号事件)。
WAITING (等待)	任务正在等待一个事件, 那么就处于等待 (WAITTING) 状态, 一旦事件发生任务就会切换到就绪状态, 内核函数 <code>os_wait</code> 就是用来将一个任务从等待 (WAITTING) 状态切换到就绪 (READY) 状态。
DELETED (删除)	任务绝不会再启动或者已经被删除, 那么就处于删除 (DELETED) 状态。内核函数 <code>os_delete_task</code> 将一个已经启动 (通过 <code>os_create_task</code>) 的任务切换到删除状态。
TIME-OUT (超时)	一个连续运行的任务, 被时间轮转内核调度中断后处于超时 (TIME-OUT) 状态。对内核而言, 这个任务相当于处在就绪 (READY) 状态。

3.4 事件

在实时操作系统里, 事件被用来控制程序中任务的行为。一个任务可以等待一个事件, 同时也可以给其他任务设置一个事件标志。

内核函数 `os_wait` 允许一个任务去等待一个或者多个事件。

- 一个任务可以等待一个超时事件 (Timeout), 超时事件是一个非常普通的事件, 就是一个简单的数, 代表有多少个内核时钟滴答数。当一个任务在等待超时事件, 那么其他的任务将继续运行。一旦需要等待的内核时钟滴答数已经耗尽, 那么这个等待的任务则继续运行。
- 时间间隔事件 (Interval) 是超时事件的一个变种, 两者之间的差别就在于时间间隔事件所要求的时钟滴答数和任务中上一次调用的内核函数 `os_wait` 有关。时间间隔事件通常用来产生一个规则而且是同步运行的任务 (例如: 一秒钟运行一次的任务), 而不管任务运行和 `os_wait` 之间的时间是多长。如果所设定的时钟滴答数已经耗尽 (时间是从内核函数 `os_wait` 的上一次调用开始算起), 任务将立即运行 (在没有其他任务运行的条件下)。
- 信号事件 (Signal) 是任务之间相互通信的一个简单形式, 一个任务可以等待另外一个任务给他发信号 (通过内核函数 `os_send_signal` 和 `isr_send_signal`)。
- 每一个任务都有一个就绪 (Ready) 标志, 这个标志可以被其他任务设置起来 (通过调用 `os_set_ready` 和 `isr_set_ready`)。一个正在等待超时 (Timeout)、时间间隔 (Interval) 和信号 (Signal) 事件的任务就可以通过设置其就绪标志让其启动运行。

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

每一个事件都有一个相关联的且由 RTX51 Tiny 维护的标志。下表是一个事件选择器，用来选择内核函数 `os_wait` 想要等待的事件。

事件选择	功 能 描 述
K_IVL	等待时间间隔事件
K_SIG	等待信号事件
K_TMO	等待超时事件

内核函数 `os_wait` 的返回值将说明具体发生的事件。

函数返回值	描 述
RDY_EVENT	任务的就绪标志被设置
SIG_EVENT	收到一个信号事件
TMO_EVENT	超时事件或者时间间隔事件设定的内核时钟滴答数已经耗尽

内核函数 `os_wait` 可以等待以下的组合事件：

- ◆ K_SIG | K_TMO：内核函数 `os_wait` 延迟当前任务的执行，一直到收到信号或者设定的超时时间已经耗尽。
- ◆ K_SIG | K_IVL：内核函数 `os_wait` 延迟当前任务的执行，一直到收到信号或者设定的间隔时间已经耗尽。

注意：

- ◆ 事件选择器不能将 K_IVL 和 K_TMO 这两个事件组合在一起。

3.5 任务调度器

调度器的作用就是将处理器分配给一个任务，RTX51 Tiny 的任务调度器通过以下的规则来决定哪一个任务获得运行权。

如果有以下条件发生，那么当前的任务被中断：

- (1) 当前任务调用了 `os_switch_task`，另外一个任务准备运行；
- (2) 当前任务调用了 `os_wait`，而要求的事件还没有发生；
- (3) 当前任务已经运行了太长的时间，超过了时间轮转所定义的时间片的值；

如果有以下条件发生，另一个任务开始运行：

- (1) 没有其他任务在运行；
- (2) 任务从就绪（READY）状态或者超时（TIME-OUT）状态启动运行；

3.6 时间轮转任务切换

RTX51 Tiny 可以配置成使用时间轮转的多任务系统（或者叫任务切换）。时间轮转允许准并行地执行几个任务，这些任务并不是连续运行的，而是运行一个时间片（CPU 的运行时间被分成时间片，RTX51 Tiny 分配时间片给每一个任务）。因为时间片很短，通常只有几

个毫秒，那么这些任务看起来象是在同时运行。

任务在分配给他们的时间片里一直运行（除非任务的时间片被放弃），然后 RTX51 Tiny 切换到下一个处于就绪（READY）状态的任务去运行。这个时间片参数可以通过配置文件 CONF_TNY.A51 去设置。

下面的例子是一个简单的 RTX51 Tiny 程序，采用了时间轮转的多任务机制。程序中的两个任务都是计数循环，RTX51 Tiny 从命名为 job0 的任务 0 开始运行，该任务还产生了另一个命名为 job1 的任务。在 job0 执行完自己的时间片之后，RTX51 Tiny 切换到 job1 运行。在 job1 运行完自己的时间片之后，RTX51 Tiny 又切换回 job0 运行。这个过程一直被不确定地重复着。

```
#include <rtx51tny.h>

int counter0;
int counter1;

void job0 (void) _task_ 0 {
    os_create_task (1);      /* mark task 1 as ready */
    while (1) {              /* loop forever */
        counter0++;          /* update the counter */
    }
}

void job1 (void) _task_ 1 {
    while (1) { /* loop forever */
        counter1++;          /* update the counter */
    }
}
```

注意：

比用等待来耗尽时间片的更好方式是使用内核函数 `os_wait` 和 `os_switch_task`，让 RTX51 Tiny 可以切换到另一个任务。内核函数 `os_wait` 的作用就是挂起当前任务（将该任务的状态更改为等待状态（WAITING））一直到特定的事件发生（将任务的状态更改为就绪状态（READY））。在这期间，其他的任务将会被运行。

3.7 合作式任务切换

如果你禁止了时间轮转的多任务方式，那么你必须让程序中的各个任务工作在合作的方式下。这就要求在你的应用程序中，每一个任务都要在特定的地方调用内核函数 `os_wait` 或者是 `os_switch_task`，从而使 RTX51 Tiny 可以完成任务的切换。

`os_wait` 和 `os_switch_task` 的不同是，`os_wait` 让你的任务等待一个事件，而 `os_switch_task`

则立即切换到已经处于就绪状态（READY）的任务。

3.8 Idle 任务

当没有任何任务处于就绪状态去运行的时候，RTX51 Tiny 就会执行一个 Idle 任务，其实这 Idle 就是一个死循环，就像下面这样：

```
SJMP $
```

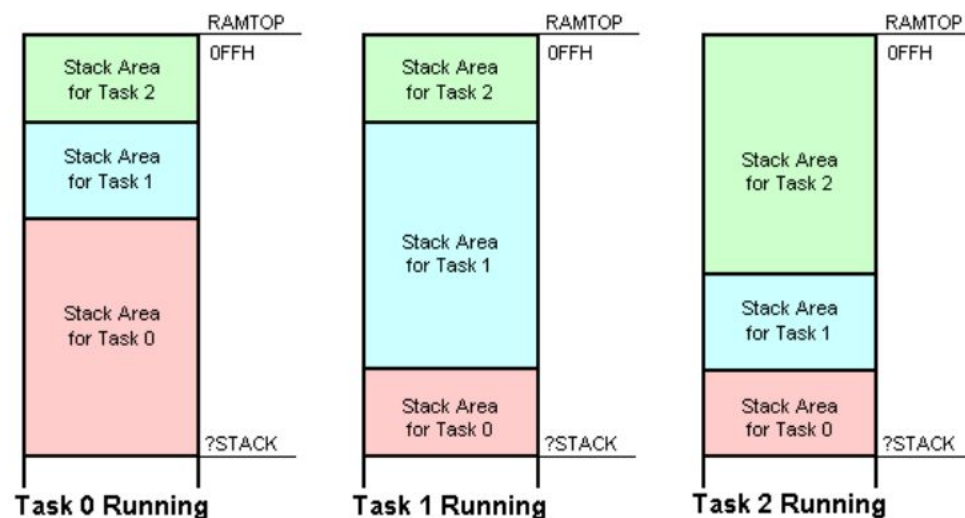
一些 8051 兼容器件提供了一个 idel 模式（停止程序的运行，一直到有中断产生），可以节省电源的损耗。在这种模式下，处理器的周边，包括中断系统是一直在工作的。

RTX51 Tiny 允许你在 Idle 任务（在没有任何任务需要运行的条件下）中激活 idel 模式。当 RTX51 Tiny 的定时器中断（或者是任何的中断）产生时，处理器恢复程序执行模式。允许给 Idle 任务添加代码，这需要设置配置文件 CONF_TNY.A51 来实现。

3.9 堆栈管理

RTX51 Tiny 可以在 8051 的内部数据存储区（IDATA）给每一个任务维护一个堆栈。当一个任务运行时，它将获得一个最大可能的堆栈空间。当任务切换发生时，堆栈重新分配，先前任务的堆栈被缩小，而当前任务的堆栈被扩大。

下图就是一个有三个独立任务的示例程序，关于任务堆栈在内部数据寄存器的空间分配。



符号?STACK 指示堆栈的起始地址。在这个例子中，位于堆栈区以下包含有全局变量，寄存器和可位寻址存储器，剩余的空间保留用来做任务堆栈。栈顶的地址是可以通过配置文件 CONF_TNY.A51 进行设置。

4、配置 RTX51 Tiny

RTX51 Tiny 可以针对每一个应用的需求进行客制化。

4.1 配置

RTX51 Tiny 必须根据你所创建的嵌入式应用进行配置。所有的配置可以在文件 CONF_TNY.A51 中找到。这个文件保存在以下的路径中：\KEIL\C51\RTXINY2\

文件 CONF_TNY.A51 中的选项允许你：

- ◆ 定义定时器中断服务程序所使用的寄存器组；
- ◆ 定义定时器中断的间隔（单位是机器周期，有一些衍生的 8051 可能不是这个单位）；
- ◆ 定义在定时中断中执行的用户代码；
- ◆ 定义时间轮转的超时值；
- ◆ 允许或者禁止时间轮转的任务切换；
- ◆ 定义你的应用程序是否包含长时间的中断服务程序；
- ◆ 定义是否采用代码分页；
- ◆ 定义 RTX51 Tiny 的堆栈顶端地址（默认是 FFH）
- ◆ 定义最小的堆栈需求；
- ◆ 定义在堆栈出错是执行的代码；
- ◆ 定义在 Idle 任务中运行的代码；

一个默认的 CONF_TNY.A51 文件已经包含在 RTX51 Tiny 的库当中，但是为了确保配置在你的应用当中生效，你必须把文件拷贝到你的项目文件目录中，并添加到你的项目当中。

为了客制化你的 RTX51 Tiny，你必须更改文件 CONF_TNY.A51 的设定；

注意：

如果你没有将配置文件 CONF_TNY.A51 包含进你的项目当中，那么库中默认的配置文件的默认选项在你的项目中可能起的是反作用。

4.1.1 硬件定时器

以下在配置文件 CONF_TNY.A51 中，由 EQU 伪指令定义的参数将决定 RTX51 Tiny 的硬件定时器是如何配置的，这个定时器将用于产生内核时钟：

- ◆ INT_REGBANK 设定 RTX51 Tiny 定时中断程序所使用的寄存器组，默认设置是 1（也就是寄存器组 1）
- ◆ INT_CLOCK 定义定时器在产生中断前的时钟周期数，这个值的范围是 1000 - 65535，小的数值将产生快速的中断，那么初始化和重装到定时器的值就是：(65536 - INT_CLOCK)。这个参数的默认值是 10000。
- ◆ LONG_USR_INTR 设定应用程序中是否有执行时间比较长的中断服务程序存在，所谓较长的（定时器中断的频率就看作是 RTX51 Tiny 内核时钟频率，也就是整个实时

内核的时钟节拍)

◆ HW_TIMER_CODE 是一个指令的宏定义，这宏将放在 RTX51 Tiny 中断服务程序的末尾，在默认的设定中，这个宏的是一条中断返回指令，如下所示：

```
HW_TIMER_CODE MACRO ; Empty Macro by default
RETI
ENDM
```

4.1.2 时间轮转

在默认设置下，时间轮转的任务调度是设置为允许的。下面的 EQU 伪指令将可以设定时间轮转的任务切换是允许还是禁止。

- ◆ TIMESHARING 设定在进行时间轮转的任务切换前需要消耗多少个 RTX51 Tiny 的内核时钟滴答数。如果这个值为 0，那么将禁止进行时间轮转的任务切换，默认的设置是 5；

注意：

整个内核时间轮转的时间片大小就取决于 INT_CLOCK 和 TIMESHARING 的设定。如标准 8051 工作在 12MHz 的晶振下，依据默认的设定（INT_CLOCK 的值为 10000，TIMESHARING 的值为 5），RTX51 Tiny 的内核时钟频率是 100Hz（12MHz/12/10000），时间轮转调度的时间片时间是：0.05 秒（5/100Hz）。

4.1.3 长中断

典型的设计中，中断服务程序（ISRs）将执行得很快，占用比较短的时间。出于某些原因，你的 ISRs 可能需要运行一个比较长的时间。如果一个高优先级的 ISR 执行时间比 RTX51 Tiny 内核时钟周期还要长，那么就有可能发生 RTX51 Tiny 定时器中断服务程序被这个执行时间比较长的用户 ISR 所中断，这就有可能导致当前的 RTX51 Tiny 定时中断程序被下一个发生的 RTX51 Tiny 中断所重入。

如果你有必要使用一个高优先权的中断，并需要较长的运行时间，那么你应该考虑尽可能减少 ISR 的指令数，或者将 RTX51 Tiny 的内核时钟频率调低，再或者使用下面的配置选项：

- ◆ LONG_USR_INTR 设定应用程序中是否有执行时间比较长的中断服务程序存在，所谓较长的运行时间是指大于内核时钟周期。如果选项的值为 1，那么 RTX51 Tiny 会增加代码去保护内核的定时器中断服务程序，避免发生重入的问题。默认的设置值是 0，要求比较快的中断服务程序。

4.1.4 代码分页

下面的设定选项允许你去选择是否在你的 RTX51 Tiny 程序中使用代码分页。

- ◆ CODE_BANKING 设定是否采用代码分页，如果值为 1，表示使用代码分页；如果

值为 0，表示不使用代码分页。默认的值是 0。

注意：

- ◆ 如果使用代码分页，那么要求使用版本在 2.12 以上的 L51_BANK.A51；
- ◆ 在我们目前的显示器项目中，都必须采用代码分页；

4.1.5 堆栈

有几个选项用来对堆栈进行配置，下面的几个 EQU 伪指令将定义内部数据存储器用于堆栈的大小以及堆栈的最小自由空间。当 CPU 堆栈不能提供足够自由栈空间时，一条指令宏定义了在这样的条件下执行的代码。

- ◆ RAMTOP 定义了栈顶地址，通常你不需要去更改这个值，除非你有位于栈顶的 IDATA 型变量；
- ◆ FREE_STACK 定义最小的堆栈需求，当发生任务切换时，RTX51 Tiny 会检测堆栈是否不能提供这个最小的需求。如果不能满足要求，那么一个 STACK_ERROR 宏会被运行。如果这个值是 0，那么将禁止进行堆栈检查。默认的设置值是 20 字节；
- ◆ STACK_ERROR 一个指令宏，在发生堆栈错误的时候（CPU 堆栈不能提供 FREE_STACK 所要求的最小堆栈空间）被运行。默认的宏是一个死循环，如下所示：

```
STACK_ERROR MACRO
CLR EA ; disable interrupts
SJMP $ ; endless loop if stack space is exhausted
ENDM
```

4.1.6 Idle 任务

当没有任何任务就绪去运行时，RTX51 Tiny 会运行一个 Idle 任务。这个 Idle 任务仅是一个简单的死循环，什么事也不做，仅仅在等待内核调度器切换到一个已经就绪的任务。下面是 EQU 伪指令，允许你将 Idle 任务更改成别的样子。

◆ CPU_IDLE 是一个指令宏，定义在 Idle 任务中执行的指令。默认的指令就是设置寄存器 PCON（许多 8051 器件都有这个位）的 Idle 模式控制位。Idle 模式通过停止程序运行来达到省电的目的，程序只有在收到中断之后才恢复运行。下面是示例：

```
CPU_IDLE MACRO
ORL PCON,#1 ; set 8051 CPU to IDLE
ENDM
```

- ◆ CPU_IDLE_CODE 设定是否会在 Idle 任务中执行 CPU_IDLE 这个宏，默认设置为 0，所以在 Idle 的任务中并不会运行这个宏。

4.2 库文件

RTX51 Tiny 包含了两个库文件：

- ◆ RTX51TINY.LIB 使用在没有代码分页的 RTX51 Tiny 程序设计中；
- ◆ RTX51BT.LIB 使用在有代码分页的 RTX51 Tiny 程序设计中；

位于目录\KEIL\C51\RTXTINY2\SOURCECODE\下的工程项目文件 RTXTINY2.PRJ，可以用来建立这两个库。

注意：

- ◆ 在你的应用程序中如果没有明确包含这两个库文件，在你使用µVision IDE 或者命令行进行连接操作时，也会进行自动连接。
- ◆ 当你建立 RTX51 Tiny 库文件时，默认的配置文件 CONF_TNY.A51 被包含到库当中。如果你不在项目中明确包含配置文件 CONF_TNY.A51，那么库的一个默认样式就会被包含进来。而后续如果对配置文件进行修改就会被保存到库当中，这就很有可能对你的应用造成不良影响。

4.3 优化

有几方面的事情可以帮助你优化 RTX51 Tiny 的程序设计：

- ◆ 如果可能，禁止时间轮转的任务调度，启用这个调度机制会占用 13 字节的堆栈空间，用于保存任务的上下文和所有的寄存器。如果任务切换是调用 RTX51 Tiny 的内核函数来实现（如：os_wai 或者 os_switch_task），那么就不需要这个堆栈空间。
- ◆ 使用内核函数 os_wait 来取代时间轮转的超时任务切换，可以提高系统的响应时间和任务的响应时间。
- ◆ 避免将内核时钟频率调节得太高，因为内核的定时中断服务程序需要占用 100 - 200 个机器周期，太高的内核时钟频率，会使得留给任务运行的时间太少。调节合适的内核时钟频率，让内核定时中断服务程序的影响减到最小。
- ◆ 为了减少 RTX51 Tiny 对内存的占用，任务号的分配应该从 0 开始，按顺序增加。

5、使用 RTX51 Tiny

要使用 RTX51 Tiny，你必须能够成功创建 RTX51 程序并进行编译和连接。最典型的过程，你必须完成以下的三项步骤。

- ◆ 编辑 RTX51 程序；
- ◆ 对程序进行编译和连接；
- ◆ 对程序进行测试和调试；

5.1 编写程序

在采用 RTX51 Tiny 进行程序设计时，你必须使用关键字 `_task_` 来定义 RTX51 Tiny 任务，要使用 RTX51 Tiny 内核函数，还必须包含头文件 RTX51TNY.H。

5.1.1 包含文件

RTX51 Tiny 仅需要包含一个头文件：RTX51TNY.H，所有的内核函数和常量定义都在这个头文件当中。在你的源文件中使用如下的格式：

```
#include <rtx51tny.h>
```

5.1.2 编程指导

在创建 RTX51 Tiny 程序时，有一些规则需要去遵循：

- ◆ 确认包含头文件：RTX51TNY.H
- ◆ 不要创建 main 函数，RTX51 Tiny 有自己的 main 函数；
- ◆ 程序至少包含一个任务函数；
- ◆ RTX51 Tiny 程序要求打开中断（EA = 1）。如果在临界区禁止了中断，需要特别小心去处理。参考在概述中有关中断的一节，包含了 RTX51 Tiny 如何使用中断系统的细节；
- ◆ 程序必须包含至少一个内核函数的调用（例如：os_wait），否则连接器不会包含 RTX51 Tiny 的库；
- ◆ Task0 是程序运行的第一个函数，你必须用内核函数 os_create_task 从 task0 创建所有其他的任务；
- ◆ 任务函数不能有 exit 或者 return 语句。任务函数必须采用一个 while(1) 这样的死循环或者是类似的结构。调用内核函数 os_delete_task 可以暂停一个任务的运行；
- ◆ 必须在 µVision IDE 或者命令行进行中指定使用 RTX51 Tiny；

5.1.3 定义任务

实时或者多任务应用通常由一个或者多个执行特定操作的任务组成。RTX51 Tiny 最多支持 16 个任务。

所谓的任务其实就是一个简单 C 函数，具有 void 型的参数列表和 void 型的返回值，而

且还需要采用关键字 `_task_` 来表明函数的属性，下面是一个例子：

```
void func (void) _task_ task_id
```

在这里：

`func` 就是任务函数的名字；
`task_id` 就是任务 ID，其范围是从 0 到 15

在下面的例子中，定义了 `job0` 函数作为任务 0，这个任务的功能就是重复对变量 `counter0` 进行加 1 操作。

```
void job0 (void) _task_ 0 {  
    while (1) {  
        counter0++;        /* increment counter */  
    }  
}
```

注意：

- ◆ 所有任务必须包含一个死循环，不允许有任何 `return` 语句；
- ◆ 任务没有返回值，它们必须采用 `void` 返回类型；
- ◆ 不能给任务传递参数，必须是一个 `void` 型的参数列表；
- ◆ 每一个任务必须采用唯一且不能重复的 ID 号；
- ◆ 为了减小 RTX51 Tiny 对内存的需求，所有任务号必须从 0 开始，依次增加；

5.2 编译和连接

有两种基本的方法编译和连接你的 RTX51 Tiny 应用程序：

- ◆ 采用 `µVision IDE`；
- ◆ 采用命令行；

5.2.1 命令行工具

RTX51 Tiny 已经完全集成到 C51 编程语言当中，这使得创建 RTX51 Tiny 应用程序是很容易的，仅仅需要采用 C 语言编程，而不需要使用汇编语言来编程。

从命令行编译你的 RTX51 Tiny 应用程序...

像通常的应用一样调用编译器，而不需要特别的参数，以下是示例：

```
C51 RTXPROG.C DEBUG OBJECTEXTEND
```

所产生的目标文件 `RTXPROG.OBJ` 除了包含 C 语言代码以外，也包含了你所定义的 RTX51 Tiny 任务。

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

从命令行连接你的 RTX51 Tiny 应用程序...

- ◆ 需要在命令行中设定参数 RTX51TINY
- ◆ 如果你更改了配置，那么你需要在目标文件列表中包含：RTX_CONF.OBJ

例如：

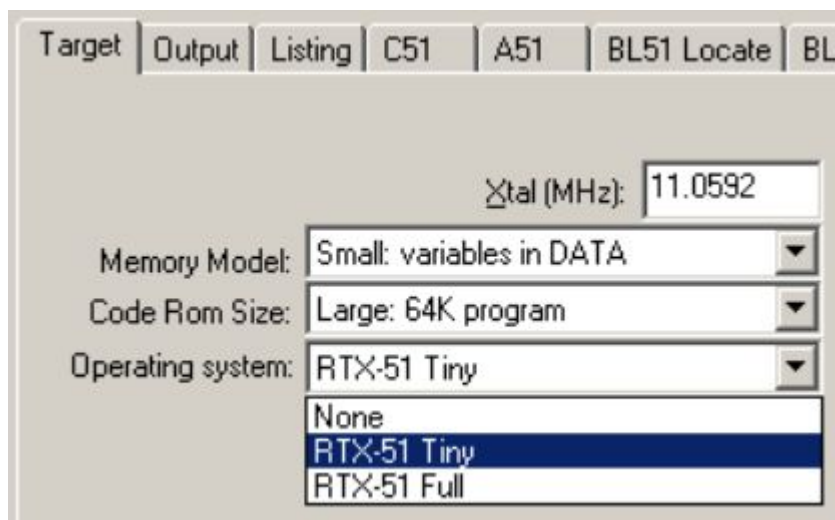
```
BL51 RTXPROG.OBJ, RTX_CONF.OBJ RTX51TINY
```

命令行参数 RTX51TINY，将指示连接器在连接 RTXPROG.OBJ 和 RTX_CONF.OBJ 的同时，将 RTX51 Tiny 的库也连接进来。这样一个 RTX51 Tiny 的程序就被创建（C 文件包含了任务的定义）。

注意：

- ◆ 不要在你的 RTX51 Tiny 程序中创建 main 函数，仅需要任务函数就可以了。mian 函数已经包含在 RTX51 Tiny 的库当中，用于启动操作系统和调用任务 0。如果在你的应用程序中包含了 mian 函数，那么连接器会产生一个 mian 符号重复定义的错误；
- ◆ 必须要在你的程序中至少创建一个任务函数；
- ◆ 程序必须至少调用一个 RTX51 Tiny 的内核函数（例如 os_wait 和 os_create_task），这样连接器才会把 RTX51 Tiny 的库连接进来；

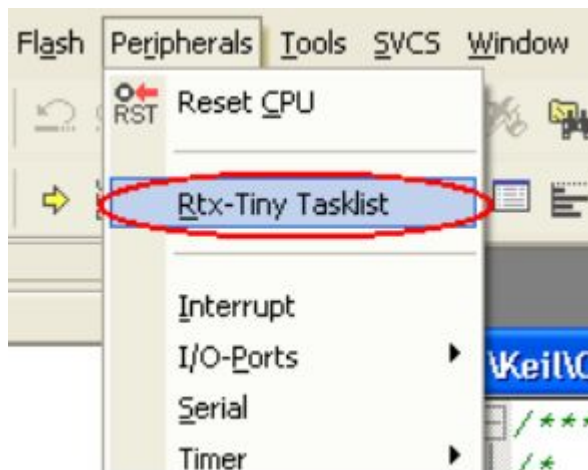
5.2.2 uVision IDE



采用 uVision 创建 RTX51 Tiny 程序...

- ◆ 打开 [Options for Target] 对话框（通过 Project 菜单选择 Options for Target 选项）；
- ◆ 选择 Target 标签；
- ◆ 在 Operating System 的下拉列表中选择 RTX-51 Tiny；

5.3 调试



μVision 仿真器允许你运行和测试 RTX51 Tiny 应用程序；加载 RTX51 Tiny 应用程序和其他的程序没有什么两样；在调试的时候也不需要特别的命令或者选项。

一个[kernel-aware]对话框将显示 RTX51 Tiny 内核以及在你程序中的所有任务的全部参数。在 Peripherals 菜单中选择 RTX51 Tiny Tasklist 项目，就可以显示这个对话框。

TID	Task Name	State	Wait for Event	Sig	Timer	Stack
0	init	Deleted		0	0xA4	0x7F
1	command	Waiting	Signal	0	0xA4	0x7F
2	clock	Waiting	Timeout	0	0x54	0x81
3	blinking	Deleted		0	0xA4	0x83
4	lights	Waiting	Signal & TimeOut	0	0xB8	0x83
5	keyread	Waiting	Timeout	0	0x02	0x85
6	get_escape	Waiting	Signal	0	0xA4	0xFB

- ◆ TID 是任务 ID，这个任务 ID 是在任务定义时所设定的；
- ◆ Task Name 是任务函数的函数名；
- ◆ State 是任务的当前状态；
- ◆ Wait for Event 说明该任务正在等待的事件是什么；
- ◆ Sig 指示该任务的信号标志的状态（用 1 表示设置）；
- ◆ Timer 是一个定时器，表示这个任务在到达超时还剩下多少个内核时钟。只有在该任务等待一个超时或者时间间隔时间时才有意义，否则就是一个自由运行的定时器，没有任何意义；
- ◆ Stack 表示该任务的任务堆栈起始地址；

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

6、例子

RTX51 Tiny 包含了几个示例项目，用以帮助你开始设计 RTX51 Tiny 程序。

这些例子都保存在一个独立的文件夹中：\KEIL\C51\RTXINY2\EXAMPLES\，而且还包含了 μ Vision 的项目文件，可以帮助你尽快创建和运行程序。

在 KEIL C51 工具安装时，将自动安装这些示例程序：

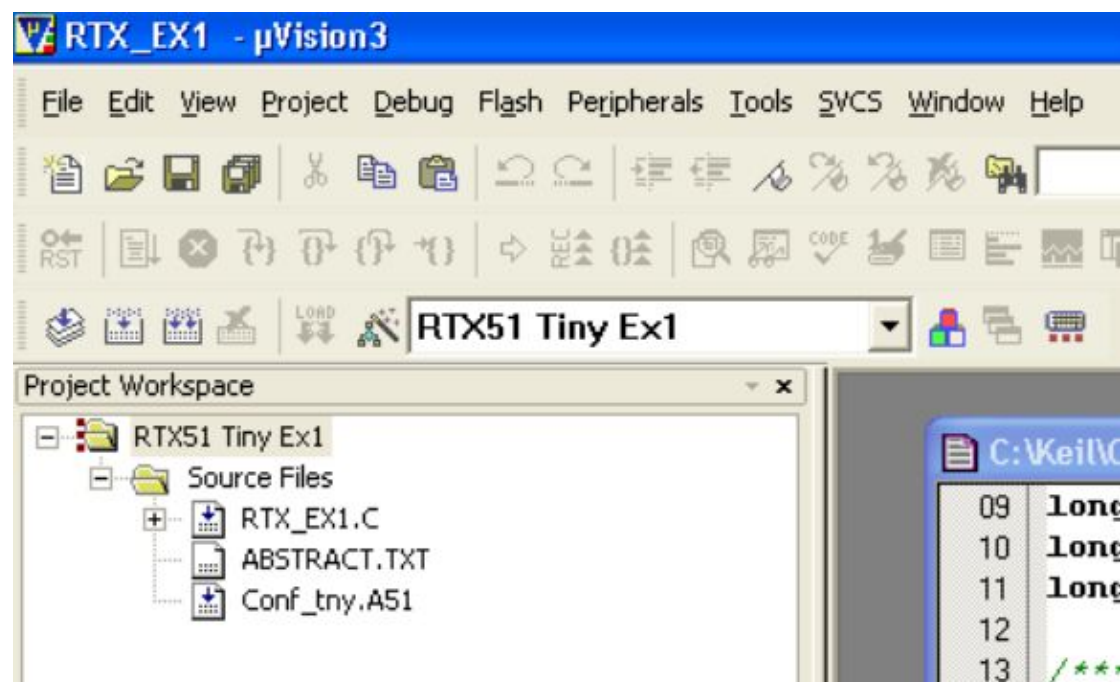
- ◆ RTX_EX1
这个项目演示采用 RTX51 Tiny 内核的时间轮转多任务调度机制；
- ◆ RTX_EX2
这个项目显示内核函数 os_wait 的功能以及信号的传递；
- ◆ TRAFFIC
这个项目演示如何采用 RTX51 Tiny 实现一个行人交通灯控制；

6.1 例子 RTX_EX1

RTX_EX1 是一个针对 μ Vision 的 RTX51 Tiny 演示项目，该项目仅包含了一个源文件 RTX_EX1.C 用于演示时间轮转的多任务调度。这个项目的所有文件都包含在以下的文件夹中：\KEIL\C51\RTXTINY2\EXAMPLES\EX1\

使用 μ Vision 来创建 RTX_EX1...

- (1) 启动 μ Vision IDE 并打开项目 RTX_EX1.UV2



- (2) 打开 Project 菜单并选择 [ Build Target], 在这里 Targe 就是 RTX_EX1;

使用命令行来创建 RTX_EX1...

- (1) 编译源文件: RTX_EX1.C
- (2) 连接目标文件: RTX_EX1.OBJ


一旦程序 RTX_EX1 被编译和连接, 那么你就可以利用 μ Vision Debugger 来进行测试。

6.2 例子 RTX_EX2

RTX_EX2 是一个针对 μ Vision 的 RTX51 Tiny 演示项目, 演示内核函数 os_wait 的功能以及信号是如何传递的。

RTX_EX2 项目仅包含了一个源文件 RTX_EX2.C。这个项目的文件都包含在以下的文件夹中: \KEIL\C51\RTXTINY2\EXAMPLES\EX2\

使用 μ Vision 来创建 RTX_EX2...

- (1) 启动 μ Vision IDE 并打开项目 RTX_EX1.UV2;
- (2) 通过 Project 菜单中的 [ Build Target] 选项, 对 RTX_EX2 进行编译和连接;

使用命令行来创建 RTX_EX2...

- (1) 编译源文件: RTX_EX2.C
- (2) 连接目标文件: RTX_EX2.OBJ

一旦程序 RTX_EX2 被编译和连接, 那么你就可以利用 μ Vision Debugger 来进行测试。

6.3 例子 Traffic

TRAFFIC 示例程序是一个行人交通灯控制器, 要比示例程序 RTX_EX1 和 RTX_EX2 复杂得多。如果不借助于像 RTX51 Tiny 这样的实时多任务操作系统, 那么是比较难实现的。

TRAFFIC 是一个由时间驱动的红绿灯控制器, 在一个由用户设定的时间段里, 如果有行人按下请求按钮, 红绿灯会立即转成 “walk” 状态, 否则就继续运行; 不在这个设定的时间里, 黄灯将会闪烁;

红绿灯控制器的命令

你可以通过 8051 的串行口和红绿灯控制器进行通信; 也可以利用调试器的 “Serial Window” 窗口来测试红绿灯控制器的命令。

下表列出所有通过串口发送的命令, 这些命令由 ASCII 字符组成, 所有命令只有接收到回车符之后才会发送。

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

命令	串行文本	功能描述
显示	D	显示时钟，开始和结束的时间
时间	T hh:mm:ss	设置当前时间，24 小时格式
开始	S hh:mm:ss	设置开始时间（24 小时格式）。红绿灯控制器通常是在开始和结束时间之间起作用，如果不在这段时间里，黄灯会闪烁。）
结束	E hh:mm:ss	设置结束时间，24 小时格式

源文件

TRAFFIC 应用程序由三个源文件构成，这些文件保存在如下的文件夹当中：

\\KEIL\\C51\\RTXTINY2\\EXAMPLES\\TRAFFIC\\

(1) TRAFFIC.C 包含了红绿灯控制程序，并分成了如下的任务：

- ◆ 任务 0 (Task0) 初始化：初始化串行接口和所有任务，任务 0 执行完之后将删除自己，因为任务 0 只需要执行一次；
- ◆ 任务 1 (Task1) 命令处理：红绿灯控制器的命令处理器。这个任务处理和控制串行口命令的接收；
- ◆ 任务 2 (Task2) 时钟：时钟控制和处理；
- ◆ 任务 3 (Task3) 闪烁：当时钟的时间不在有效的范围时，黄灯闪烁；
- ◆ 任务 4 (Task4) 灯控：当时钟的时间落在有效的范围时（在开始时间和结束时间之间），分阶段对红绿灯进行控制；
- ◆ 任务 5 (Task5) 按钮：判断行人是否有按下按钮，如果有就发送信号给灯控任务；
- ◆ 任务 6 (Task6) 退出：检查串行口是否有收到 ESC 字符，如果有收到，那么就终止一个先前启动的显示命令（通过串行口输入的命令）；

(2) SERIAL.C 采用中断方式来驱动串行口，这个文件包含了函数 putchar 和 getkey。高阶的 I/O 函数 printf 和 getline 其实调用的就是这些基本的 I/O 函数。这个红绿灯控制的应用程序也可以采用非中断方式来驱动串行口，不过性能就不如中断方式了。

(3) GETLINE.C 是一个命令编辑器，用于从串行口接收字符，这个源文件也用在 MEASURE 的项目中。

使用μVision 来创建 TRAFFIC...

(1) 启动μVision IDE 并打开项目 TRAFFIC.UV2

(2) 通过 Project 菜单中的[ Build Target]选项，对 TRAFFIC 进行编译和连接；

一旦程序 TRAFFIC 被编译和连接，那么你就可以利用μVision Debugger 来进行测试。

7、库函数参考

下面的内容将描述 RTX51 Tiny 内核函数，这些函数按字目的顺序排列。每一个函数都按如下的几个部分进行描述：

摘要 (Summary)	简要描述函数的结果、列出包含该函数声明和原型的头文件、语法说明和每一个参数的描述。
描述 (Description)	提供该函数的详细描述以及函数如何使用。
返回值 (Return Value)	描述该函数的返回值。
相关函数 (See Also)	相关联的内核函数的名字。
示例 (Example)	给出一个正确使用该函数的程序片段。

注意：

- ◆ 如果函数是以 `os_` 开头的，那么可以在任务中调用，但是不能在中断函数中调用；
- ◆ 如果函数是以 `isr_` 开始的，那么可以在中断函数里调用，但是不能在任务中调用；

7.1 isr_send_signal

摘要 (Summary)	<pre>#include <rtx51tny.h> char isr_send_signal(unsigned char task_id); /* 发送信号给一个任务 */</pre>
描述 (Description)	<p>函数 <code>isr_send_signal</code> 发送一个信号给任务（这个任务用 <code>task_id</code> 来标明）。如果该指定的任务正在等待一个信号，那么这个函数就是让这个任务进入就绪状态准备去运行，并没有启动一个任务的能力。换言之，信号是保存在任务的信号标志中。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；◆ 这个函数可以在任何中断函数中被调用；
返回值 (Return Value)	如果函数 <code>isr_send_signal</code> 返回 0，那么调用成功；如果返回 -1，说明所指定的任务并不存在。
相关函数 (See Also)	<code>os_clear_signal</code> , <code>os_send_signal</code> , <code>os_wait</code>
示例 (Example)	<pre>#include <rtx51tny.h> void tst_isr_send_signal (void) interrupt 2 { isr_send_signal (8); /* 给任务 8 发送信号 */ }</pre>

7.2 isr_set_ready

摘要 (Summary)	<pre>#include <rtx51tiny.h> char isr_set_ready(unsigned char task_id); /* 任务进入就绪状态 */</pre>
描述 (Description)	<p>函数 <code>isr_set_ready</code> 使一个指定的任务（用 <code>task_id</code> 来标明）进入就绪状态。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；◆ 这个函数可以在任何中断函数中被调用；
返回值 (Return Value)	没有返回值
示例 (Example)	<pre>#include <rtx51tiny.h> void tst_isr_set_ready (void) interrupt 2 { isr_set_ready (1); /* 给任务 1 设置就绪标志*/ }</pre>

7.3 os_clear_signal

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_clear_signal(unsigned char task_id); /* 清除指定任务的信号标志 */</pre>
描述 (Description)	<p>函数 os_clear_signal 清除一个指定任务（这个任务用 task_id 来标明）的信号标志。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	如果函数 os_clear_signal 返回 0，那么指定任务的信号标志被成功清除；如果返回-1，说明所指定的任务并不存在。
相关函数 (See Also)	isr_send_signal, os_send_signal, os_wait
示例 (Example)	<pre>#include <rtx51tiny.h> void tst_os_clear_signal (void) _task_ 8 { . . . os_clear_signal (5); /* 清除任务 5 的信号标志 */ . . . }</pre>

7.4 os_create_task

摘要 (Summary)	<pre>#include <rtx51tny.h> char os_create_task(unsigned char task_id); /* 指定的任务被启动 */</pre>
描述 (Description)	<p>函数 <code>os_create_task</code> 用于启动指定任务（用 <code>task_id</code> 来标明）。该任务被标记成就绪状态，将在下一个可获得的时机时运行。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	<p>如果函数 <code>os_create_task</code> 返回 0，那么指定任务被成功启动；如果返回 -1，说明任务没有被启动。如果一个任务没有被启动，那么这个任务可能已经在运行或者没有采用 <code>task_id</code> 进行定义。</p>
相关函数 (See Also)	<code>os_delete_task</code>
示例 (Example)	<pre>include <rtx51tny.h> #include <stdio.h> /* for printf */ void new_task (void) _task_ 2 { .. } void tst_os_create_task (void) _task_ 0 { . . if (os_create_task (2)) { printf ("Couldn't start task 2\n"); } . . }</pre>

7.5 os_delete_task

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_delete_task(unsigned char task_id); /* 指定的任务被删除 */</pre>
描述 (Description)	<p>函数 <code>os_delete_task</code> 用于删除指定任务（用 <code>task_id</code> 来标明）。该任务从任务队列中移除。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	<p>如果函数 <code>os_delete_task</code> 返回 0，那么指定任务被成功停止和删除；如果返回-1，说明该任务不存在或者该任务没有被启动。</p>
相关函数 (See Also)	<code>os_create_task</code>
示例 (Example)	<pre>#include <rtx51tiny.h> #include <stdio.h> /* for printf */ void tst_os_delete_task (void) _task_ 0 { . . . if (os_delete_task (2)) { printf ("Couldn't stop task 2\n"); } . . . }</pre>

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

7.6 os_reset_interval

摘要 (Summary)	<pre>#include <rtx51tny.h> void os_reset_interval(unsigned char ticks); /* 内核时钟数 */</pre>
描述 (Description)	<p>如果函数 <code>os_wait</code> 同时等待 <code>K_IVL</code> 和 <code>K_SIG</code> 事件，那么函数 <code>os_reset_interval</code> 用来校正定时器的问题。在该条件下，如果一个信号 (<code>K_SIG</code>) 事件让函数 <code>os_wait</code> 退出等待，而时间间隔没有被调整的话，那么接下来函数 <code>os_wait</code> 等待一个时间间隔事件，可能就不能按要求准确延时了。</p> <p>函数 <code>os_reset_interval</code> 允许你去复位计时器，以致接下来函数 <code>os_wait</code> 的调用可以达到预期的目的。</p>
返回值 (Return Value)	没有返回值
示例 (Example)	<pre>#include <rtx51tny.h> void task_func (void) _task_4 { . switch (os_wait2 (K_SIG K_IVL, 100)) { case TMO_EVENT: /* Timeout occurred */ /* os_reset_interval not required */ break; case SIG_EVENT: /* Signal received */ /* os_reset_interval required */ os_reset_interval (100); /* do something with the signal */ break; } . }</pre>

7.7 os_running_task_id

摘要 (Summary)	<pre>#include <rtx51tny.h> char os_running_task_id(void);</pre>
描述 (Description)	<p>函数 os_running_task_id 返回当前正在运行的任务的 ID 号是多少。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	<p>函数 os_running_task_id 返回当前正在运行的任务的 ID 号，这个值的范围是：0 - 15</p>
示例 (Example)	<pre>#include <rtx51tny.h> void tst_os_running_task (void) _task_3 { unsigned char tid; tid = os_running_task_id (); /* tid = 3 */ }</pre>

7.8 os_send_signal

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_send_signal(unsigned char task_id); /* 给指定任务发送信号 */</pre>
描述 (Description)	<p>函数 <code>os_send_signal</code> 给一个指定任务（用 <code>task_id</code> 来标明）发送信号。如果该任务已经在等待信号，那么这个函数就让这个指定任务进入就绪状态准备运行，当然这个函数并没有启动一个任务的功能。如果所指定的任务还没有在等待信号，那么这个信号也保存在该指定任务的信号标志中。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	如果函数 <code>os_send_signal</code> 返回 0，那么信号被成功发送；如果返回-1，说明该任务不存。
相关函数 (See Also)	<code>isr_send_signal</code> , <code>os_clear_signal</code> , <code>os_wait</code>
示例 (Example)	<pre>#include <rtx51tiny.h> void signal_func (void) _task_ 2 { . . . os_send_signal (8); /* signal task #8 */ . . . } void tst_os_send_signal (void) _task_ 8 { . . . os_send_signal (2); /* signal task #2 */ . . . }</pre>

7.9 os_set_ready

摘要 (Summary)	<pre>#include <rtx51tny.h> char os_set_ready(unsigned char task_id); /* 给指定任务设定就绪标志 */</pre>
描述 (Description)	<p>函数 os_set_ready 设置一个指定任务（用 task_id 来标明）进入就绪状态。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	没有返回值
示例 (Example)	<pre>#include <rtx51tny.h> void ready_func (void) _task_ 2 { . . . os_set_ready (1); /* 给任务 1 设置就绪标志*/ . . . }</pre>

7.10 os_switch_task

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_switch_task(void);</pre>
描述 (Description)	<p>函数 os_switch_task 让一个任务暂停，同时让另外一个任务运行。如果调用函数 os_switch_task 的任务是唯一一个就绪任务，那么这个任务就会马上重新运行。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；
返回值 (Return Value)	没有返回值
示例 (Example)	<pre>#include <rtx51tiny.h> #include <stdio.h> /* for printf */ void long_job (void) _task_ 1 { float f1, f2; f1 = 0.0; while (1) { f2 = log (f1); f1 += 0.0001; os_switch_task (); // 运行其他任务 } }</pre>

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

7.11 os_wait

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_wait (unsigned char event_sel, /* 选择要等待的事件 */ unsigned char ticks, /* 需要等待的内核时钟周期数 */ unsigned int dummy); /* 没有使用的参数 */</pre>	
描述 (Description)	<p>函数 os_wait 可以暂停当前任务并等待一个或者几个事件，例如：时间间隔、超时或者从另外一个任务或中断发出的信号。参数 event_sel 用来指定要等待的事件，也可以是事件的组合，下表是这些事件的一个列表：</p>	
	事 件	描 述
	K_IVL	等待一个时间间隔事件，单位是内核时钟周期
	K_SIG	等待一个信号事件
	K_TMO	等待一个超时事件，单位是内核时钟周期
	<p>这些事件可以是一个逻辑或的关系，用符号 ‘ ’ 来表示。例如 K_TMO K_SIG，表示一个任务等待一个超时事件或者信号事件。</p> <p>参数 ticks 设定了时间间隔事件（K_IVL）和超时事件（K_TMO）的时间，单位是内核时钟周期。</p> <p>参数 dummy 是为了和 RTX51 Full 相兼容，在 RTX51 Tiny 是没有用的。</p> <p>注意：</p> <ul style="list-style-type: none">◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统；◆ 关于 K_IVL、K_SIG 和 K_TMO，参考操作原理一章中事件一节，可以有更多的细节；	
返回值 (Return Value)	<p>当一个指定的事件发生时，任务进入就绪状态。当任务重新开始运行时，下表就是由函数 os_wait 返回的值，这些值表明是什么事件触发任务重新开始运行。</p>	
	返 回 值	描 述
	RDY_EVENT	任务的就绪标志被函数 os_set_ready 或 isr_set_ready 所设置
	SIG_EVENT	接收到一个信号
	TMO_EVENT	超时事件或者时间间隔事件
	NOT_OK	参数 event_sel 的值是非法的
相关函数 (See Also)	isr_send_signal, isr_set_ready, os_clear_signal, os_reset_interval, os_send_sigantl, os_set_ready, os_wait1, os_wait2	
示例 (Example)	<pre>#include <rtx51tiny.h> #include <stdio.h> /* for printf */</pre>	

RTX51-Tiny 使用

REV 1.0 — 14 July 2014

Operating System Manual

```
void tst_os_wait (void) _task_ 9
{
    while (1)
    {
        char event;
        event = os_wait (K_SIG + K_TMO, 50, 0);
        switch (event)
        {
            default: /* 这个条件绝不会发生 */
                break;
            case TMO_EVENT: /* 超时事件 */
                /* 50 个内核时钟周期的超时条件 */
                break;
            case SIG_EVENT: /* 接收到信号 */
                break;
        }
    }
}
```

7.12 os_wait1

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_wait1(unsigned char event_sel); /* 指定等待的事件 */</pre>	
描述 (Description)	<p>函数 os_wait1 暂停当前任务，等待一个事件发生。函数 os_wait1 是函数 os_wait 的一个子集，并不能支持所有的事件。参数 event_sel 仅仅只能设定为信号事件 (K_SIG)。</p> <p>注意：</p> <ul style="list-style-type: none"> ◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统； ◆ 关于 K_IVL、K_SIG 和 K_TMO，参考操作原理一章中事件一节，可以有更多的细节； 	
返回值 (Return Value)	<p>当一个指定的事件发生时，任务进入就绪状态。当任务重新开始运行时，下表就是由函数 os_wait1 返回的值，这些值表明是什么事件触发任务重新开始运行。</p>	
	返回值	描述
	RDY_EVENT	任务的就绪标志被函数 os_set_ready 或 isr_set_ready 所设置
	SIG_EVENT	接收到一个信号
	NOT_OK	参数 event_sel 的值是非法的
示例 (Example)	请参考函数 os_wait 的示例部分	

7.13 os_wait2

摘要 (Summary)	<pre>#include <rtx51tiny.h> char os_wait2 (unsigned char event_sel, /* 选择要等待的事件 */ unsigned char ticks); /* 需要等待的内核时钟周期数 */</pre>	
描述 (Description)	<p>函数 os_wait2 可以暂停当前任务并等待一个或者几个事件，例如：时间间隔、超时或者从另外一个任务或中断发出的信号。参数 event_sel 用来指定要等待的事件，也可以是事件的组合，下表是这些事件的一个列表：</p>	
	事 件	描 述
	K_IVL	等待一个时间间隔事件，单位是内核时钟周期
	K_SIG	等待一个信号事件
	K_TMO	等待一个超时事件，单位是内核时钟周期
	<p>这些事件可以是一个逻辑或的关系，用符号 ‘ ’ 来表示。例如 K_TMO K_SIG，表示一个任务等待一个超时事件或者信号事件。</p> <p>参数 ticks 设定了时间间隔事件（K_IVL）和超时事件（K_TMO）的时间，单位是内核时钟周期。</p> <p>注意：</p> <ul style="list-style-type: none"> ◆ 这个函数是 RTX51 Tiny 实时操作系统的一部分，PK51 专业开发工具集中包含有这个操作系统； ◆ 关于 K_IVL、K_SIG 和 K_TMO，参考操作原理一章中事件一节，可以有更多的细节； 	
返回值 (Return Value)	<p>当一个指定的事件发生时，任务进入就绪状态。当任务重新开始运行时，下表就是由函数 os_wait2 返回的值，这些值表明是什么事件触发任务重新开始运行。</p>	
	返 回 值	描 述
	RDY_EVENT	任务的就绪标志被函数 os_set_ready 或 isr_set_ready 所设置
	SIG_EVENT	接收到一个信号
	TMO_EVENT	超时事件或者时间间隔事件
	NOT_OK	参数 event_sel 的值是非法的
示例 (Example)	请参考函数 os_wait 的示例部分	

7.14 `os_wait`, `os_wait1` 和 `os_wait2` 之间的差异

`os_wait` 有三个输入参数（在 RTX51 Tiny 操作系统中，第 3 个参数是没有用的），而 `os_wait1` 只有一个参数，只是对信号事件进行处理。

和 `os_wait` 相比较，`os_wait2` 具有完全一样的功能，差别只是少了一个参数，建议在 RTX51 Tiny 中使用 `os_wait2` 而不是 `os_wait`，这样程序量会比较小，因为只需要传递了两个参数。