序言

本手册解释如何使用 rtx51小型实时操作系统并给出 rtx51完全版功能的概述。这个手册并不是一本详细的实时应用的入门教程并且假定你已经熟悉了 Keil C51、A51、相关工具、DOS操作系统和8051微处理器的硬件和指令体系。

建议使用下列书籍作为实时程序设计领域的入门教程。

Deitel, H.M., Operating Systems, second edition,
Addison-Wesley Publishing Company, 1990

Ripps, David, A Guide to Real-Time Programming, Englewood Cliffs, N.J,
Prentice Hall, 1988/

Allworth, S.T., Introduction to Real-Time Software Design,
Springer-Verlag Inc., New York

本用户指南包含6部分：

第 1部分： 概述、描述 rtx51实时操作系统的功能并给出RTX51 Tiny和 RTX51 Full 版本的基本特征和差异。同时包括 RTX51 Full 和 RTX51 Tiny. 的技术数据。

第 2部分：系统需求和定义、 讨论 RTX51 Tiny的开发工具和目标系统的需求，解释在 RTX51 Tiny 手册中使用的术语和任务定义的描述。

第3部分：建立 RTX51 Tiny 应用程序、 描述建立 RTX51 Tiny应用程序的步骤。

第4部分：库函数、 提供全部 RTX51 Tiny库程序的索引。

第5部分：系统调试、 描述 RTX51 Tiny的堆栈管理还包括系统调试得有关信息。

第6部分：应用程序例子、 包括几个使用 RTX51 Tiny的例子和软件开发过程的描述。此信息可以作为你的实时设计的指导原则。

符号约定

本手册使用下列约定：

范例说明

Bold:BL51采用 Bold字体，大写的文本用于可执行程序、数据文件、源文件、环境变量的名称和你在DOS命令提示窗口键入的命令。 这种文本通常表示你必须逐字地输入的命令。

例如：

CLS DIR DS51.INI

C51 A51 SET

注意你事实上不需要使用全部的大写字母来输入这些命令。

Courier 这种字体的正文通常用于表示出现在屏幕上的信息或打印在打印机上的信息。

本字体也用于出现在命令行上的论述或描述文字。

KEYS **这种字体的文字表示在键盘上实际存在的键。 例如：、“**Press Enter to Continue.”

ALT+<x> **表明一Alt组合键；Alt和 <x>键必须同时按下。**

CTRL+<x> **表明一Ctrl组合控制键；Ctrl和 <x>控制键必须同时按下。**

1

概述

RTX51是一个用于8051系列处理器多任务实时操作系统。
RTX51可以简化那些复杂而且时间要求严格的工程的软件设计工作。
有二个不同的RTX51版本可以利用：
RTX51 Full 使用四个任务优先权完成同时存在时间片轮转调度和抢先的任务切换。 RTX51工作在与中断功能相似的状态下。 信号和信息可以通过邮箱系统在任务之间互相传递。 你可以从一存储池中分配和释放内存。 你可以强迫一个任务等待中断、超时或者是从另一个任务或中断发出的信号或信息。
RTX51 Tiny 是一个 RTX51的子集，它可以很容易地在没有任何外部存储器的单片8051系统上运转。 除了下列例外，RTX51 Tiny支持许多在 RTX51中的特征。
RTX51 Tiny仅支持时间片轮转任务切换和使用信号进行任务切换。不支持抢先式的任务切换。不包括消息历程。 没有存储器池分配程序。
在本章其它部分所提到的 RTX51包括这两种变形。 Differences between the two are so stated in the text as their need becomes applicable.

入门

许多微处理器应用程序要求同时执行两个工作或任务. 对于这样的应用程序，一个实时操作系统（RTOS）允许灵活的分配系统资源（中央处理器、存储器、等等.)给各个任务。 RTX51是一个很好使用的强大的实时操作系统。 RTX51可以运行于所有的8051派生机型。
你可以使用标准 C语言编写和编译一个程序或使用 C51构造、编译他们， 仅在指定任务标识符和优先权上有一点差别。.RTX51程序也要求你载运程序中用include命令引入实施管理的头文件并使用 BL51 linker/locator进行连接和选择适当的 RTX51库文件。

1单任务程序

一个标准的 C语言程序从主函数开始执行。 在一嵌入式应用中,主函数通常是一段无限循环的代码，可以认为是一个连续执行的单独任务。
例如：

```
int counter;
void main (void) {
counter = 0;
while (1) { /* 始终重复 */
counter++; /* 计数器加1 */
}
}
```

时间片轮转程序

一中更高级的 C语言程序可以在不使用实时操作系统的情况下实现时间片轮转拟多任务系统。 在这种系统中、任务或功能被一段无限循环程序重复调用。
例如：

```
int counter;
void main (void) {
counter = 0;
while (1) { /* 始终重复 */
check_serial_io ();
```

```
process_serial_cmds (); /* 处理串行输入 */
check_kbd_io ();
process_kbd_cmds (); /* process keyboard input */
adjust_ctrlr_parms (); /* adjust the controller */
counter++; /* increment counter */
}
}
```

用 RTX51进行时间片轮转调度

rtx51也能完成时间片轮转多重任务，而且允许准并行执行多个无限循环或任务。 任务并不是并行执行的而是按时间片执行的。

可利用的中央处理器时间被分成时间片由 RTX51分配一个时间片给每个任务。 每个任务允许执行一个预先确定的时间。 然后、rtx51切换到另一准备运行的任务并且允许这个任务执行片刻。 时间片非常段、通常为几个毫秒。 因此、它表现得如同各个任务是同时地执行的。

RTX51使用一个8051硬件计时器中断作为定时程序。

产生的周期性中断用于驱动 RTX51时钟。

RTX51不需要在你的程序中拥有一个主函数。 它将自动开始执行任务 0。 如果你确实有一个主函数、你必须利用 RTX51 Tiny中的 os_create_task函数或 RTX51中的 os_start_system函数手工启动 RTX51。

下列例子显示一个只使用时间片轮转任务调度的简单的 RTX51应用程序。 在本程序里的二个任务是简单计数器回路。 rtx51开始执行函数名为 job0的任务 0。 这些功能添加了另一个叫做 job1任务。 在 job0运行一会儿以后、RTX51切换到 job1。 在 job0运行一会儿以后、RTX51转回到 job0。 这个过程将不确定地重复下去。

```
#include <rtx51tny.h>
int counter0;
int counter1;
void job0 (void) _task_ 0 {
os_create (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 */
}
}
```

RTX51事件

即使是在等待一个任务的时间片到达时，你也可以使用 os_wait函数通知 RTX51它可以让另一个任务开始执行。 这个功能中止正在运行的当前任务然后等待一指定事件的发生。 在这个时候、

任意数量的其他任务仍可以执行。

使用 RTX51的时间溢出事件

你可以用 os_wait函数等待的最简单的事件是事件溢出 RTX51时钟报时信号周期。 这类事件可被用于任务中需要延迟一段时间的地方。 这可用于查询一个开关状态的代码中。 在这样的条件下，只须每隔 50ms左右查询一次开关。下一个例子技术示范你可以在允许其他的任务执行的时候使用os_wait功能延迟任务的执行。

```
#include <rtx51tny.h>
int counter0;
int counter1;
void job0 (void) _task_ 0 {
os_create (1); /* mark task 1 as ready */
while (1) { /* loop forever */
counter0++; /* update the counter */
os_wait (K_TMO, 3); /* pause for 3 clock ticks */
}
}
void job1 (void) _task_ 1 {
while (1) { /* loop forever */
counter1++; /* update the counter */
os_wait (K_TMO, 5); /* pause for 5 clock ticks */
}
}
```

在上面的例子中、job0像前面叙述的一样启动 job1。 然后、在增加 counter0计数以后，job0呼叫 os_wait函数以暂停3个时钟报时信号。 这时、rtx51切换到下一个任务 job1。 在 job1增加counter1计数以后、它也调用 os_wait以暂停5个时钟报时信号。 现在、rtx51没有其他的任务需要执行，因此在它可以延续执行 job0之前它进入一个空循环等待3个时钟报时信号过去。

本例子的结果是 counter0每3个时钟报时周期加1，而 counter1每5个时钟报时周期加 1。

使用 RTX51的信号

你可以使用 os_wait功能暂停一个任务并等待从另一个任务发出的信号（或旗标）。 这可以用于协调两个或更多的任务。

等待一个信号会如下面所诉工作：如果一任务在等待一个信号，并且信号标志是 0，在这个信号被发送之前，这个任务将一直处于挂起状态。 如果信号标志已经是 1，当任务查询信号时、信号标志会被清除，并且继续执行任务。

以下例子说明了这种应用：

```
#include <rtx51tny.h>
int counter0;
int counter1;
void job0 (void) _task_ 0 {
os_create (1); /* mark task 1 as ready */
while (1) { /* loop forever */
if (++counter0 == 0) /* update the counter */
os_send_signal (1); /* signal task 1 */
}
}
void job1 (void) _task_ 1 {
while (1) { /* loop forever */
```

```
os_wait (K_SIG, 0, 0); /* wait for a signal */
counter1++; /* update the counter */
}
}
```

在上述例子中，job1一直处于等待状态，直到它接收到从任何其他任务发出的信号。 当它接收到一个信号时、它将使 counter1加 1然后继续等待另一个信号。 job0 将连续地增加 counter0直到它溢出到 0。 当溢出发生时、job0发送一个信号给 job1同时 RTX51标记 job1为执行状态。 在RTX51到达下一个时钟报时周期前，job1不会开始执行。

优先权和抢先机制

上述程序的缺点是当 job0发出信号时 job1并不是立即开始执行。 在一些情况下，由于时间的原因这是不受欢迎的。RTX51允许你指定任务的优先级。一个具有较高优先级的任务变成可用的时，会中断一个低优先级任务或抢在它前面执行。 这叫做优先型多任务或仅仅称之为抢先机制。

注意 RTX51 Tiny不支持抢先机制和优先等级。

你可以变更上述函数 job1的说明给它一个比 job0高的优先级。 全部任务的默认优先级均为 0。这是最低的优先级。 优先级可以可以设定为 0-3。 下面的例子说明如何定义 job1的优先级为1。

```
void job1 (void) _task_ 1 _priority_ 1 {
while (1) { /* loop forever */
os_wait (K_SIG, 0, 0); /* wait for a signal */
counter1++; /* update the counter */
}
}
```

现在、每当 job0发送一个信号给 job1时，job1将立即开始执行。

用 RTX51进行编译和连接

rtx51是完全地统一到 c51程序设计语言中的。 这使得很容易熟悉如何生成 RTX51应用程序。 上述的例子是可执行的 RTX51程序。 你不需要书写任何 8051汇编程序或函数。 你唯一需要做的是用c51编译你的 RTX51程序并把他们用 BL51 Linker/Locator连接在一起。 例如：如果你使用 RTX51 Tiny的话，你将使用以下命令行命令：

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51TINY
```

使用以下命令行编译和连接使用 RTX51。

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51
```

中断

RTX51工作在与中断功能相似的状态下。 中断函数可以与 RTX51通信并且可以发送信号或信息给RTX51任务。 RTX51 Full允许将中断指定给一个任务。

信息传送

RTX51 Full支持使用以下函数在任务间进行信息交换：SEND、 RECEIVE MESSAGE和 WAIT for MESSAGE。 信息是一个可以解释为指向存储块的指示字的 16位数值。 RTX51 Full支持可变大小

带有存储池系统的信息。

CAN通信

局域网控制器可以很容易地用 RTX51/CAN实现。局域网控制器可以很容易地用 RTX51/CAN实现。RTX51/CAN是一个统一在 RTX51 Full中的 CAN任务。 一RTX51 CAN任务实现经由 CAN网络的信息传送。 其他的CAN工作站既可以用 RTX51配置也可以不用 RTX51配置。

BITBUS通信

RTX51 Full包括支持用 Intel 8044传送信息的主和从 BITBUS任务。

事件

RTX51的等待功能支持以下事件：

- 超时(Timeout)：挂起运转的任务指定数量的时钟报时周期。
- 间隔(Interval)：（仅在 RTX51 Tiny中使用）类似于超时，但是软件定时器没有复位来产生循环的间隔（时钟所需要的）。
- 信号（Signal）：用于任务内部协调。
- 信息（Message）：（仅适用于 RTX51 Full）用于信息的交换。
- 中断(Interrupt)：（仅适用于 RTX51 Full）一个任务可以等待 8051硬件中断。
- 旗标标志(Semaphore)：（仅适用于 RTX51 Full）旗标标志用于管理共享的系统资源。

RTX51函数

下表列出全部 RTX51函数；RTX51 Tiny支持的功能用记号(*)标出。 [时间标记(Timings)由 RTX51 Full控制]

函数	说明	运行时间 (时钟周期)
os_create (*)	将一任务加入到执行队列	302
os_delete (*)	将一任务从执行队列中删除	172
os_send_signal (*)	发送一个信号给任务（在任务中调用)	408 with task switch 快速任务转换时：316 没有任务转换时：71
os_clear_signal (*)	删除一个发送的信号	57
isr_send_signal (*)	发送一个信号给任务（在中断中调用)	46
os_wait (*)	等待事件	等待信号时 68 等待消息时 160
os_attach_interrupt	指定一个任务到中断源	119
os_detach_interrupt	删除中断指定	96
os_disable_isr	使 8051硬件中断无效	81
os_enable_isr	使 8051硬件中断有效	80
os_send_message/ os_send_token	发送一条信息或设定一个旗标（从任务中调用)	任务转换时 443 快速任务转换时：343 没有任务转换时：94
isr_send_message	发送一条信息给任务（在中断中调用)	53
isr_recv_message	接收一条信息（在中断中调用}	71（使用信息时)
os_create_pool	定义一个存储池	644（大小20 *10字节)
os_get_block	从一存储池得到一个存储块	148
os_free_block	将一个存储块反还给存储池	160
os_set_slice	定义 RTX51系统时钟值	67

附加的调试和支撑函数：check_mailboxes、check_task、check_tasks、check_mail、check_pool、set_int_mask、reset_int_mask。

CAN函数（仅RTX51 Full可用}

CAN控制器支持：Philips 82c200、80c592和Intel 82526（更多 CAN控制器在准备中）。

CAN函数	说明
can_task_create	建立 CAN通信任务
can_hw_init	CAN控制器硬件初始化
can_def_obj	定义通信对象
can_start / can_stop	启动/停止 CAN通信
can_send	在 CAN总线上发送对象
can_write	再不发送的情况下将新的数据写入一个对象
can_read	直接读一个对象的数据
can_receive	接收全部不受限制的对象
can_bind_obj	将一个目标绑定到一个任务；当接收到对象时，任务开始
can_unbind_obj	解开任务和对象之间的绑定
can_wait	等待接收一个绑定对象
can_request	发送一个远程帧给指定对象
can_get_status	获得 CAN控制器的实际状态。

技术数据

文字说明	RTX51 Full	RTX51 Tiny
任务的数量	256; 最多. 19 任务处于激活状态	16
RAM需求	40 .. 46 bytes DATA 20 .. 200 bytes IDATA （用户堆栈) min. 650 bytes XDATA	7 bytes DATA 3 * <任务计数> idata
程序存储器需求	6KB .. 8KB	900 bytes
硬件要求	timer 0 or timer 1	timer 0
系统时钟	1000 .. 40000 cycles	1000 .. 65535 cycles
中断等待	< 50 cycles	< 20 cycles
切换时间	70 .. 100 cycles (快速任务) 180 .. 700 cycles (标准任务) 依靠堆栈加载	100 .. 700 cycles 依靠堆栈加载
邮箱系统	8个邮箱每个邮箱8个入口	不可用
存储器池系统	最多可到 16个存储器池	不可用
旗标	8 * 1 bit	不可用

要求和定义

以下章节描述了 RTX51 Tiny的软硬件需求并定义了在本手册中使用的术语。 RTX51 Tiny使用一些综合的系统调用，如同建立在 C51编译程序内部的使用 _task_关键字定义任务一样。（RTX51 Tiny uses a combination of system calls as well as the _task_ keyword for the task definition which is built in to the C51 compiler。）. RTX51 Tiny的任务定义和重要的特征

也在这章内描述。

开发工具需求

操作 RTX51 Tiny需要以下软件产品：

- c51编译程序
- bl51 Code Banking连接程序
- A51宏汇编程序

|库文件 RTX51TNY.LIB必须保存在 DOS环境变量 C51LIB指定的程序库路径内。 一般是目录 C51\LIB。

|包含文件rtx51tny.h必须保存在 DOS环境变量 C51INC指定的包含路径内。 一般是目录 C51\INC。

目标系统需求

RTX51 Tiny可以在没有任何外部数据存储器的单片 8051系统上运行。

但应用程序仍然可以访问外部存储器。 RTX51 Tiny可以使用C51支持的全部存储器模块。 选择记忆模型仅影响应用目标的位置。 RTX51 Tiny应用程序的系统变量和堆栈区总是被保存在 8051的内部存储器中（DATA或IDATA）。 一般来说、RTX51 Tiny应用程序工作在小模式下。

RTX51 Tiny仅支持时间片轮转任务切换。 不支持抢先式的任务切换和任务优先权。 如果你的应用程序必须使用抢先式的任务切换，你需要使用 RTX51 Full实时执行程序。

RTX51 Tiny没有按照 bank switching程序设计。 如果你的 code banking应用程序需要实时的多重任务，你需要使用 RTX51 Full实时执行程序。

中断处理

可以向中断功能一样操作。类似于其他的8051应用程序的，为了触发中断必须使能 8051硬件寄存器内的中断源。RTX51 Tiny不包括任何中断管理；因此、中断使能足够处理中断。

RTX51使用 8051的 timer 0定时器和定时器中断。停用全部中断（EA位）或定时0中断会停止 RTX51 Tiny的操作。 除了很少几条 8051指令外、定时0中断不应该关闭。

可重入功能

不允许从几个任务或中断过程调用非可再入 C语言函数。

非可再入 C51函数将它们的参数和自动变量（局部数据）保存在静态存储器内；因此、当重复调用函数时这些数据会被改写。 如果用户可以保证不会递归呼叫的话，非可再入 C语言函数可以被多个任务调用、。 通常来说，这意味着必须禁止时间片轮转任务调度而且 RTX51 TINY系统函数不会呼叫任何这样的函数。

那些仅使用寄存器作为参变量和自动变量的 C语言函数总是可再入的而且可以从不同的 RTX51 Tiny任务中没有任何限制的调用。

C51编译程序也提供可重入功能。 *参看 C51用户手册以便获得更多信息*。 可再入函数将他们的参变量和局部数据变量储存到一个可再入堆栈内并且数据是被保护的以预防多重呼叫。 然而，RTX51 Tiny不包括任何 C51可再入堆栈管理。 如果你在你的应用程序中使用可再入函数，你必须保证这些功能不呼叫任何 RTX51 TINY系统函数而且那些可再入函数不会被 RTX51 Tiny的时间片轮转任务调度所中断。 完全版本： RTX51 Full包括一个可再入函数的堆栈管理。

C51库函数

全部的可再入 C51库函数可以没有任何限制的用于全部任务。

非可再入 c51库函数与非可再入 C语言函数在应用时有着同样的限制。 *参看可重入功能以获得*

更多信息。

多数据指针和数学单元的用法

| c51编译程序允许你使用8051派生类型的多数据指针和数学单元。 因为 RTX51 Tiny不包括任何对这些硬件的管理，最好你不要与 RTX51 Tiny一起使用这些器件。 如果你可以保证在使用这些派生硬件的程序执行期间不会被时间片轮转任务中断的话，你可以使用多数据指针和数学单元。

寄存器段

RTX51 Tiny分配全部任务到寄存器段 0。 因此、全部的任务函数必须用c51的默认设置 registerbank（0）编译。 |中断函数可以使用剩余的寄存器段。 然而，需要寄存器段中的 6 个固定的字节。 这些用于 RTX51 Tiny的寄存器段可以用配置变量 INT_REGBANK定义。 *参看第 3 章, RTX51 Tiny配置以获得更多信息。*

任务定义

实时或多任务应用程序由一个或多个完成具体的操作的任务组成。 RTX51 Tiny允许最多 16个任务。 任务是使用下面的格式定义的、返回质类型和参数列表为空的 C语言函数

```
void func (void) _task_ num
```

num是一个从 0到 15的任务标识号。

例子：

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

定义函数 job0为任务号0。 这个任务所做的是增加一个计数器的计数值并重复。 你会注意到在这种方式下全部的任务是用无限循环实现的。

任务管理

你定义的各个任务可以处于许多不同状态中的任意一个位置。

RTX51 Tiny 核心保持每个任务特有的状态。 下面是一个不同状态的文字说明。

状态	文字说明
RUNNING	当前正在运行的任务处于 RUNNING状态。 同一时间只有一个任务可以运行
READY	等待运行的任务处于 READY状态。 在当前运行的任务处理完成之后，RTX51 Tiny开始下一个处于 READY状态的任务。
WAITING	等待一个事件的任务处于 WAITING状态。 如果事件发生的话，任务进入 READY状态。
DELETED	没有开始的任务处于删除状态。
TIME-OUT	被时间片轮转超时终端的任务处于 TIME - OUT状态。 这个状态与 READY状态相同。

任务切换

RTX51 Tiny能完成时间片轮转多重任务，而且允许准并行执行多个无限循环或任务。 任务并不是并行执行的而是按时间片执行的。

可利用的中央处理器时间被分成时间片由RTX51 Tiny分配一个时间片给每个任务。 每个任务允许执行一个预先确定的时间。 然后、RTX51 Tiny切换到另一准备运行的任务并且允许这个任务执行片刻。 一个时间片的持续时间可以用配置变量 TIMESHARING 定义。 *参看第3章，RTX51 Tiny配置以获得更多信息。* 即使是在等待一个任务的时间片到达时，你也可以使用 os_wait系统函数通知 RTX51它可以让另一个任务开始执行。 os_wait中止正在运行的当前任务然后等待一指定事件的发生。 在这个时候、任意数量的其他任务仍可以执行。

RTX51 Tiny将处理器分配到一个任务的过程叫做调度程序(scheduler)。 RTX51 Tiny调度程序定义哪些任务按照下面的规则运行：

如果出现以下情况，当前运行任务中断：

1. 任务调用 os_wait函数并且指定事件没有发生。
2. 任务运行时间超过定义的时间片轮转超时时间。

如果出现以下情况，则开始另一个任务：

1. 没有其他的任务运行。
2. 将要开始的任务处 READY或 TIME - OUT状态。

事件

RTX51 Tiny的 os_wait函数支持以下事件类型：

SIGNAL: 任务间通信位。 信号可以使用RTX51 TINY系统函数设定或清除。 一个任务可以等待信号被设定后继续执行。 如果一个任务调用 os_wait函数来等待一个信号并且信号没有设定，任务将一直挂起到信号设定。 然后，任务返回到READY状态而且可以开始执行。

TIMEOUT: 一个从os_wait函数开始的时间延迟。 延迟的持续时间为指定的时钟报时信号。 使用一个超时值调用 os_wait函数的任务将中止到时间延迟结束。 然后，任务返回到READY状态而且可以开始执行。

INTERVAL: 一个从os_wait函数开始的间隔延迟。 延迟的间隔为指定的时钟报时信号。与超时延迟的区别是 RTX51计时器没有复位。Therefore the event INTERVAL works with a timer which is running permantly. interval事件可以用于以相同的间隔运行的任务；一个简单例子是一个时钟。

注意：事件SIGNAL可以与事件TIMEOUT结合，此时 RTX51 Tiny将等待信号和时间周期全部发生。

建立 RTX51 Tiny应用程序

编写RTX51 Tiny程序要求你包含 rtx51tny.h头标文件在在你的C语言程序的\c51\inc\子目录下而且使用_task_函数属性声明你的任务。

RTX51 Tiny程序不需要一个C语言主函数(main)。 连接过程将包含首先执行任务0的程序代码。

RTX51 TINY配置

你可以修改在\c51\lib\子目录的 RTX51 TINY配置文件conf_tny.a51。 你可以改变在这个配置文件中的下列参数。

- 用于系统时钟报时中断的寄存器组
- 系统计时器的间隔时间
- 时间片轮转超时值

- 内部数据存储器容量
- RTX51 Tiny运行之后，释放的堆栈大小

这个文件的一部分列在下面。

```
;------------------------------------;
This file is part of the 'RTX51 tiny' Real-Time Operating System Package
;------------------------------------;
CONF_TNY.A51: This code allows configuration of the
; 'RTX51 tiny' Real Time Operating System
;
; To translate this file use A51 with the following invocation:

; A51 CONF_TNY.A51
;
; To link the modified CONF_TNY.OBJ file to your application use the following
; BL51 invocation:
;
; BL51 <your object file list>, CONF_TNY.OBJ <controls>
;
;------------------------------------;
; 'RTX51 tiny' Hardware-Timer
; ==========================
;
; With the following EQU statements the initialization of the 'RTX51 tiny'
; Hardware-Timer can be defined ('RTX51 tiny' uses the 8051 Timer 0 for
; controlling RTX51 software timers).
;
; ; define the register bank used for the timer interrupt.
INT_REGBANK EQU 1 ; default is Registerbank 1
;
; ; define Hardware-Timer Overflow in 8051 machine cycles.
INT_CLOCK EQU 10000 ; default is 10000 cycles
;
; ; define Round-Robin Timeout in Hardware-Timer Ticks.
TIMESHARING EQU 5 ; default is 5 ticks.
; ;
; ; note: Round-Robin can be disabled by using value 0.
; Note: Round-Robin Task Switching can be disabled by using '0' as
; value for the TIMESHARING equate.
;------------------------------------;
; 'RTX51 tiny' Stack Space
; =======================
;
; The following EQU statements defines the size of the internal RAM used
; for stack area and the minimum free space on the stack. A macro defines
; the code executed when the stack space is exhausted.
```

```
;
; ; define the highest RAM address used for CPU stack
RAMTOP EQU 0FFH ; default is address (256 - 1)
;
FREE_STACK EQU 20 ; default is 20 bytes free space on stack
;
STACK_ERROR MACRO
CLR EA ; disable interrupts
SJMP $ ; endless loop if stack space is exhausted
ENDM
;
;----------------------------------------
```

这个配置文件定义了许多可以修改为适合你具体的应用程序的常数。 这些在下面的表格中有具体的描述。

变量	文字说明
INT_REGBANK	指示哪些寄存器组将用于RTX51 Tiny的系统中断。
INT_CLOCK	定义系统时钟间隔。系统时钟使用这个间隔产生中断。定义的数目确定了每一中断的CPU周期数量。
TIMESHARING	定义时间片轮转任务切换的超时时间。 它的值表明了在 RTX51 Tiny切换到另一任务之前时间报时信号中断的数目。 如果这个值是 0，时间片轮转多重任务将失效。
RAMTOP	表明8051派生系列内存储器存储单元的最大尺寸。 用于 8051，这个值应设定为 7Fh。 用于 8052，这个值应设定为 0FFh。
FREE_STACK	按字节定义了自由堆栈区的大小。 当切换任务时，RTX51 Tiny检验堆栈区指定数量的有效字节。 如果堆栈区太小，RTX51 Tiny将激活 STACK_ERROR宏。 用于 FREE_STACK的缺省值是20。 允许值为0-0ffh。
STACK_ERROR	RTX51 Tiny检查到一个堆栈问题时是运行的宏。 你可以把这个宏改为你的应用程序需要完成的任何操作。

编译 RTX51 Tiny程序

RTX51 Tiny应用程序不需要专门的编译程序开关或设置。 你可以编译普通的C语言源文件一样编译你的 RTX51 Tiny源文件。

连接 RTX51 Tiny程序

RTX51 Tiny应用程序必须使用BL51 code banking linker/locator进行连接。
RTX51 TINY指令必须在所有目标文件后在命令行上规定。
参看工具手册中的 RTX51TINY指令。

优化 RTX51 Tiny程序

当建立rtx51应用程序时，下列项目应该注意。

■ 如果可能的话，禁止时间片轮转多重任务。 那些使用时间片轮转多任务处理的任务需要13字节的堆栈空间用于存储任务环境（寄存器，等等.)。 如果任务切换是由 os_wai t函数触发的则不需要环境存储器。

os_wait函数还会改善系统反应时间，因为一个等待执行的任务无须等待全部的时间片轮转超时持续时间。

■ 不要将报时信号中断速率设得太快。 将报时信号发生率设到低一些的数值，增加每秒发出报时信号的数目。 每个时钟报时中断约有100到200个CPU周期。 所以，应该把时间报时信号速率设

的足够高以使中断等待时间减到最少。

RTX51 TINY系统函数

在 \c51\lib\子目录下的 RTX51 Tiny库文件 RTX51TNY.LIB中有许多子程序。 这些子程序允许你建立和解除任务，从一个任务向另一个任务发送和接收信号，或延迟一个任务一定数量的报时信号。

这些子程序在下面的列表中简单的介绍了一下，并在后面引用的函数中详细说明。

子程序	文字说明
isr_send_signal	从一个中断发送一个信号到一个任务
os_clear_signal	删除一个发送的信号
os_create_task	移动一个任务到运行队列
os_delete_task	从运行队列中删除一个任务
os_running_task_id	返回当前运行的任务的任务标识符（task ID)
os_send_signal	从一个任务发送一个信号到另一个任务
os_wait	等待一个事件
os_wait1	等待一个事件
os_wait1 os_wait2	等待一个事件

函数调用

下面几页描述了 RTX51 TINY系统函数。 在这里，系统函数是按字母顺序排序的和说明的，每个系统函数被分成几个部分：

摘要： 简要地描述子程序的作用，列出包括它的说明和原型的包含文件，说明语法，并且描述所有的参数。

文字说明： 提供一个子程序的详细说明和它是如何使用的。

返回值：描述子程序的返回值。

参见：相关的子程序。

例子： 举例说明如何恰当的使用函数或程序段。

isr_send_signal

摘要： #include <rtx51tny.h>

char isr_send_signal (**unsigned char** *task_id*); /* **ID** of task to signal */

文字说明 isr_send_signal函数向任务 task_id发送一个信号。如果指定任务已经在等待一个信号，这个函数调用会把任务准备好用于运行。 另外，信号保存在任务的信号标志位。

isr_send_signal函数只可以从中断函数中调用。

返回值：如果成功， isr_send_signal函数的返回值为0；如果规定的任务不存在，则返回值为-1。

参见： os_clear_signal, os_send_signal, os_wait

例子： #include <rtx51tny.h>

```
void tst_isr_send_signal (void) interrupt 2
{
isr_send_signal (8); /* signal task #8 */
```

```
}
```

os_clear_signal

摘要： **#include <rtx51tny.h>**

char os_clear_signal (

unsigned char *task_i*d); /* task **ID** of signal to clear */

文字说明 **os_clear_signal函数清除用task_id表示的任务的信号标志位。**

返回值 **如果信号标志位成功地清除，** **os_clear_signal函数的返回值为0。** **如果规定的任务不存在，返回值为-1。**

参见： isr_send_signal, os_send_signal

例子：

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */
void tst_os_clear_signal (void) _task_ 8
{
os_clear_signal (5);/* clear signal flag in task 5 */
}
```

os_create_task

摘要： **#include <rtx51tny.h>**

char os_create_task (

unsigned char *task_i*d); /* **ID** of task to start */

文字说明 **| os_create_task函数启动使用用task_id表示的任务号定义的任务函数。** **|任务被标记为ready状态并且依据RTX51 Tiny规定的运行。**

如果任务成功地开始，返回值：os_create_task函数的返回值为0。 **如果任务不能被开始或如果没有使用规定任务号定义的任务，返回值为-1。**

参见： os_delete_task

例子：

```
#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");
   }
}
```

os_delete_task

摘要： **#include <rtx51tny.h>**

char os_delete_task (

unsigned char *task_i*d); /* **ID** of task to stop and delete */

文字说明 **os_delete_task函数停止用** **task_id表示的任务的信号标志位。** **指定的任务被从任务列表中删除。**

返回值 如果任务成功地停止和删除，返回值：os_create_task函数的返回值为0。 如果规定的任务不存在或没有开始，返回值为-1。

参见： os_create_task

例子：
```
#include <rtx51tny.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");
}
}
```

os_running_task_id

摘要： **#include <rtx51tny.h>**

char os_running_task_id (void);

文字说明 os_running_task_id函数判断当前执行的任务函数的任务标识符。

返回值 os_running_task_id函数返回当前执行的任务函数的任务标识符。 返回值的范围为0-15。

参见： os_create_task, os_delete_task

例子：
```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */
void tst_os_running_task (void) _task_ 3
{
unsigned char tid;
tid = os_running_task_id ();
/* tid = 3 */
}
```

os_send_signal

摘要： **#include <rtx51tny.h>**

char os_send_signal (

unsigned char *task_i*d); /* **ID** of task to signal */

文字说明 os_send_signal函数向任务 task_id发送一个信号。 如果指定任务已经在等待一个信号，这个函数调用会把任务准备好用于运行。 另外，信号保存在任务的信号标志位。

os_send_signal函数只可以从任务函数中调用。

返回值：如果成功， os_send_signal函数的返回值为0；如果规定的任务不存在，则返回值为-1。

参见： isr_send_signal, os_clear_signal, os_wait.

例子：
```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */
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 */
}
```

os_wait

摘要： **#include <rtx51tny.h>**

char os_wait (

unsigned char *event_sel*, /* events to wait for */

unsigned char *ticks*, /* timer ticks to wait */

unsigned int *dummy*); /* unused argument */

文字说明 os_wait函数停止当前任务并等待一个或几个事件，比如一个时间间隔、一个超时、或从一个任务或中断发送给另一个任务或中断的信号。 参数 event_sel确定事件或要等待的事件，并且可以综合使用下列常数：

事件常数	文字说明
K_IVL	等待一个报时信号间隔。
K_SIG	等待一个信号。
K_TMO	等待一个超时(time-out)。

上述事件可以用字符"|"进行逻辑或。 例如：K_TMO | K_SIG，规定任务等待一个超时或一个信号。参数 ticks规定等待一个间隔事件(k_ivl)或一个超时事件(k_tmo)的报时信号数目。参数 dummy是为了提供与 RTX51的兼容性，在 RTX51 Tiny中没有使用。

返回值 当一个指定的事件发生时，任务允许运行。 运行被恢复并且 os_wait函数返回一个用于识别重新启动任务的事件的识别常数。 可能的返回值是：

返回值	文字说明
SIG_EVENT	接收到一个信号。
TMO_EVENT	一个超时(time-out)已经完成或一个间隔(interval)已经期满。
NOT_OK	参数 event_sel的值无效。

参见： os_wait1, os_wait2

例子： **#include <rtx51tny.h>**

```
#include <stdio.h> /* for printf */
void tst_os_wait (void) _task_ 9
{
while (1)
{
char event;
event = os_wait (K_SIG + K_TMO, 50, 0);
switch (event)
{
default:
/* this should never happen */
break;
case TMO_EVENT: /* time-out */
/* 50 tick time-out occurred */
break;
case SIG_EVENT: /* signal recvd */
```

```
/* signal received */
break;
}
}
```


os_wait1

摘要： **#include <rtx51tny.h>**

char os_wait1 (

unsigned char *event_sel*); /* events to wait for */

文字说明 os_wait1函数停止当前任务并等待一个事件的发生。os_wait1函数是os_wait函数的一个子集并且不允许os_wait的全部事件。 event_sel参数规定要等待的事件并且只可以使用值K_SIG等待一个信号。当一个信号事件发生时，任务允许运行。 运行被恢复并且 os_wait1函数返回一个用于识别重新启动任务的事件的识别常数。 可能的返回值是：

返回值	文字说明
SIG_EVENT	接收到一个信号
NOT_OK	参数 event_sel的值无效。

参见： os_wait, os_wait2

例子： See **os_wai**t.

os_wait2

摘要： **#include <rtx51tny.h>**

char os_wait2 (

unsigned char *event_sel*, /* events to wait for */

unsigned char *ticks*); /* timer ticks to wait */

文字说明 os_wait2函数停止当前任务并等待一个或几个事件，比如一个时间间隔、一个超时、或从一个任务或中断发送给另一个任务或中断的信号。 参数 event_sel确定事件或要等待的事件，并且可以综合使用下列常数：

事件常数	文字说明
K_IVL	等待一个报时信号间隔。
K_SIG	等待一个信号。
K_TMO	等待一个超时。

上述事件可以用字符"|"进行逻辑或。 例如：K_TMO | K_SIG，规定任务等待一个超时或一个信号。参数 ticks规定等待一个间隔事件（K_IVL）或一个超时事件（K_TMO）的报时信号数目。

返回值 当一个指定的事件发生时，任务允许运行。 运行被恢复并且 os_wait2函数返回一个用于识别重新启动任务的事件的识别常数。 可能的返回值是：

返回值	文字说明
SIG_EVENT	接收到一个信号。
TMO_EVENT	一个超时(time-out)已经完成或一个间隔(interval)已经期满。
NOT_OK	参数 event_sel的值无效。

参见： os_wait, os_wait1

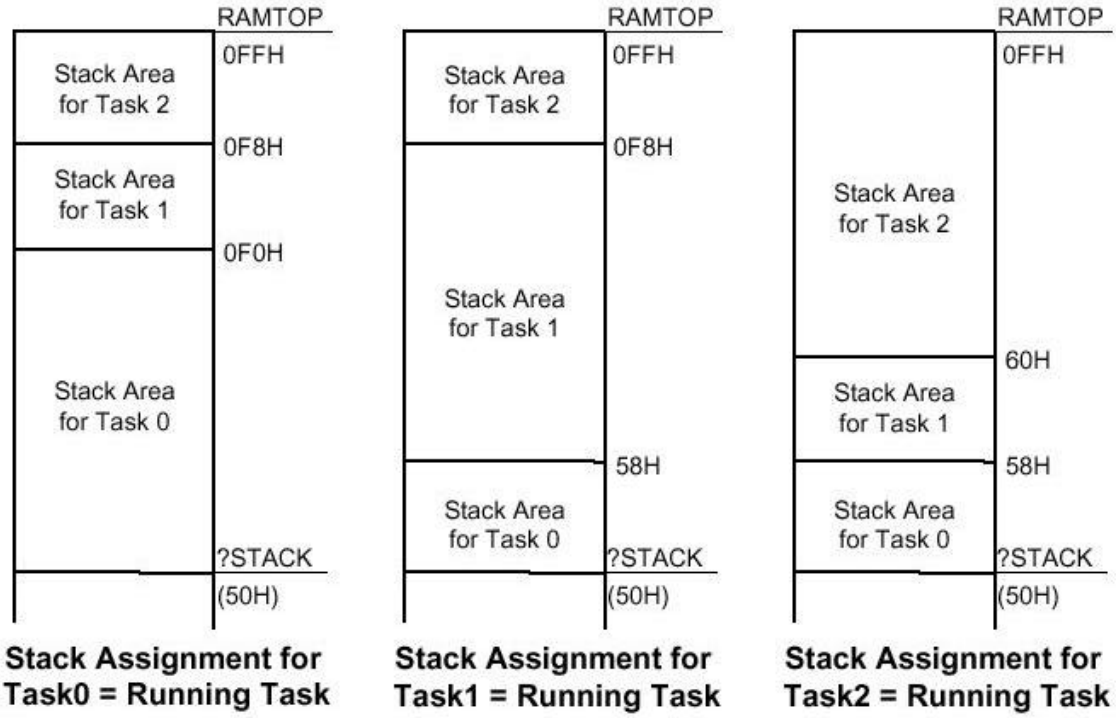
例子： See **os_wai**t..

系统调试

这一章包括有关堆栈处理和使用 dScope-51进行调试的说明。

堆栈管理

RTX51 Tiny为每个任务保留一个专用堆栈区。 由于RTX51 Tiny的设计仅使用8051的片内存储器资源，全部的堆栈在8051的内部存储器（IDATA）中管理。 为了分派最大的可利用的栈空间到当前运行的任务，用于其他未运行任务的堆栈空间被移走。 下面的图形说明单个任务的堆栈分配。



图形说明RTX51 Tiny总是将全部的闲置内存分配为运行任务的堆栈区。 用于堆栈的存储器起始于符号 ?STACK，?STACK表示 ?STACK段的起始地址。 符号 ?STACK指向内部存储器中第一个未分配的字节|。

用 dScope- 51调试

一个 RTX51 Tiny应用程序可以使用dScope-51进行测试（dScope-51源程序级调试程序）。 RTX51系统状况可以用一个调试功能显示。 以下是对使用这个调试功能的解释。

|调试功能在文件DBG_TINY.INC中定义（Windows版本的文件名是DBG_TINY.DSW），并且可以通过下面的命令调入 dScope-51内。 在定义这个调试功能之前， RTX51 Tiny应用程序必须装入。 调试功能通过按 F3键激活并显示 RTX51 Tiny状态。 另外，每任务转换都会用一个信息表示。

例子：

```
DS51 TRAFFIC
>INCLUDE DBG_TINY.INC
>G
<F3-KEY>
```

Task ID	Start	State	Wait for Event	Signal	Timer	Stack
0	0026H	DELETED		0	131	84H
1	00D1H	WAITING	SIGNAL	0	131	84H
2	0043H	WAITING	TIMEOUT	0	5	86H
3	0278H	DELETED		0	131	88H
4	02ACH	WAITING	SIGNAL & TIMEOUT	0	220	88H
5	032BH	WAITING	TIMEOUT	0	1	8AH
6	000EH	WAITING	SIGNAL	0	131	FBH

调试输出的解释：

Task ID 表明用于任务的任务号，任务是用 c51编译程序内的 _task_关键字定义的。

Start 表明任务函数的起始地址。

State 表明任务函数的状态。 状态可以是以下任意一个：

状态	文字说明
RUNNING	当前正在运行的任务处于 RUNNING状态。 同一时间只有一个任务可以运行
READY	等待运行的任务处于 READY状态。 在当前运行的任务处理完成之后，RTX51 Tiny开始下一个处于 READY状态的任务。
WAITING	等待一个事件的任务处于 WAITING状态。 如果事件发生的话，任务进入 READY状态。
.DELETED	没有开始的任务处于删除状态。
TIME-OUT	被时间片轮转超时中断的任务处于 TIME - OUT状态。 This state is equivalent to the READY STATE. 这个状态与 READY状态相同。

Wait for Event **表明当前任务正在等待哪些事件。 可以综合使用以下事件：**

事件	文字说明
TIMEOUT	在计时器数值变为0之前，任务处于等待状态。 当os_wait函数被 K_TMO或 K_IVL事件选择器调用时，显示这个事件。
SIGNAL	在信号标志位变为1之前，这个任务处于等待状态。 当os_wait函数被 K_SIG事件选择器调用时，显示这个事件。

Signal 表明信号标志位的状态：1为信号置位，0为信号复位。

Timer 表明 timeout所需要的报时信号数量。

请注意：计时器一直自由运行，并且只在 os_wait函数被参数 K_TMO调用时设定 timeou值。

Stack 表明内部存储器中局部任务堆栈的起始地址。

在本章的前面，描述了在堆栈管理区下RTX - 51任务的布局。

应用程序例子

RTX_EX1: 你的第一个RTX51程序

程序RTX_EX1示范了如何使用RTX51 Tiny的时间片轮转调度多重任务。

这程序仅由一个源文件rtx_ex1.c组成，位于\C51V4\RTX_TINY\RTX_EX1或\CDEMO\51\RTX_TINY\RTX_EX1目录。 RTX_EX1.C的内容列在下面。

```
/*----------------------------------------------------------------------------*/
/* */
/* RTX_EX1.C: The first RTX51 Program */
/* */
/*----------------------------------------------------------------------------*/
#pragma CODE DEBUG OBJECTEXTEND
#include <rtx51tny.h> /* RTX51 tiny functions & defines */
int counter0; /* counter for task 0 */
int counter1; /* counter for task 1 */
int counter2; /* counter for task 2 */
/*----------------------------------------------------------------------------*/
/* Task 0 'job0': RTX51 tiny starts execution with task 0 */
/*----------------------------------------------------------------------------*/
job0 () _task_ 0 {
os_create_task (1); /* start task 1 */
os_create_task (2); /* start task 2 */
while (1) { /* endless loop */
counter0++; /* increment counter 0 */
}
}
/*----------------------------------------------------------------------------*/
/* Task 1 'job1': RTX51 tiny starts this task with os_create_task (1) */
/*----------------------------------------------------------------------------*/
job1 () _task_ 1 {
while (1) { /* endless loop */
counter1++; /* increment counter 1 */
}
}
/*----------------------------------------------------------------------------*/
/* Task 2 'job2': RTX51 tiny starts this task with os_create_task (2) */
/*----------------------------------------------------------------------------*/
job2 () _task_ 2 {
while (1) { /* endless loop */
counter2++; /* increment counter 2 */
}
}
```

在 DOS提示符下键入下面的命令来编译和连接RTX_EX1。

```
C51 RTX_EX1.C DEBUG OBJECTEXTEND
BL51 RTX_EX1.OBJ RTX51TINY
```

编译和连接 RTX_EX1后，你可以使用 ds51测试它。 键入：

```
DS51 RTX_EX1 INIT(RTX_EX1.INI)
```

INIT（RTX_EX1.INI）指令加载一个配置 DS51屏幕的初始化文件；加载适当的输入输出驱动文件；初始化变量 counter0、counter1、和counter2的观察点；最后开始执行 RTX_EX1。

当各任务执行时，你会看见对应的计数器的数值不断增加。 计数器变量显示在屏幕顶端的监视窗口内。

键入 CTRL+C停止执行 RTX_EX1，然后在DS51命令提示符下键入

INCLUDE DBG_TINY.INC

这会装入一个允许你显示任务状态信息的包含文件。 你可能需要用 ALT+U键增加 exe窗口的大小，以便显示全部任务信息。

一旦包含文件加载后，按下 F3键即可显示这个程序中定义的三个任务的状态信息。

Task ID	Start	State	Wait for Event	Signal	Timer	Stack
0	000EH	TIMEOUT		0	217	20H
1	0023H	RUNNING		0	217	2FH
2	002EH	TIMEOUT		0	217	F0H

RTX_EX2: 一个简单的 RTX51应用程序

|程序r RTX_EX2示范了RTX51 Tiny应用程序如何使用os_wait函数和信号传递。 这程序仅由一个源文件RTX_EX2.C组成，位于\C51V4\RTX_TINY\RTX_EX2或\CDEMO\51\RTX_TINY\RTX_EX2目录。RTX_EX2.C的内容列在下面。

```
/*----------------------------------------------------------------------------*/
/* RTX_EX2.C: A RTX51 Application */
/*----------------------------------------------------------------------------*/
#pragma CODE DEBUG OBJECTEXTEND
#include <rtx51tny.h> /* RTX51 tiny functions & defines */
int counter0; /* counter for task 0 */
int counter1; /* counter for task 1 */
int counter2; /* counter for task 2 */
int counter3; /* counter for task 2 */
/*----------------------------------------------------------------------------*/
/* Task 0 'job0': RTX51 tiny starts execution with task 0 */
/*----------------------------------------------------------------------------*/
job0 () _task_ 0 {
os_create_task (1); /* start task 1 */
os_create_task (2); /* start task 2 */
os_create_task (3); /* start task 3 */
while (1) { /* endless loop */
counter0++; /* increment counter 0 */
os_wait (K_TMO, 5, 0); /* wait for timeout: 5 ticks */
}
}
/*----------------------------------------------------------------------------*/
/* Task 1 'job1': RTX51 tiny starts this task with os_create_task (1) */
/*----------------------------------------------------------------------------*/
job1 () _task_ 1 {
while (1) { /* endless loop */
counter1++; /* increment counter 1 */
os_wait (K_TMO, 10, 0); /* wait for timeout: 10 ticks */
}
```

```
}
/*****************************************************************************/
/* Task 2 'job2': RTX51 tiny starts this task with os_create_task (2) */
/*****************************************************************************/
job2 () _task_ 2 {
while (1) { /* endless loop */
counter2++; /* increment counter 2 */
if (counter2 == 0) { /* signal overflow of counter 2 */
os_send_signal (3); /* to task 3 */
}
}
}
/*****************************************************************************/
/* Task 3 'job3': RTX51 tiny starts this task with os_create_task (3) */
/*****************************************************************************/
job3 () _task_ 3 {
while (1) { /* endless loop */
os_wait (K_SIG, 0, 0); /* wait for signal */
counter3++; /* process overflow from counter 2 */
}
}
```

在 DOS提示符下键入下列命令来编译和连接 RTX_EX2。

```
C51 RTX_EX2.C DEBUG OBJECTEXTEND
BL51 RTX_EX2.OBJ RTX51TINY
```

编译和连接 RTX_EX2后，你可以使用 ds51测试它。 键入

```
DS51 RTX_EX2
```

来运行 DS51和加载 RTX_EX2。 当 DS51加载后，在 DS51命令提示符下键入下面的命令。

```
WS counter0
WS counter1
WS counter2
WS counter3
G
```

这会为四个任务的计数器变量设置观察点和开始执行 RTX_EX2。 RTX_EX2按下面的方式增加四个计数器的值：

counter0 每5个 RTX51报时信号增加一个计数值

counter1 每10个 RTX101报时信号增加一个计数值

counter2 以能达到的最快速度增加（这个任务获得大多数可用的 CPU时间)

counter3 每次 counter2溢出后加1

键入CTRL+C停止执行RTX_EX1并键入F3来显示在这个程序中定义的四个任务的状态信息。

Task ID	Start	State	Wait for Event	Signal	Timer	Stack
0	000EH	WAITING	TIMEOUT	0	5	28H
1	0032H	WAITING	TIMEOUT	0	10	2AH
2	0047H	RUNNING		0	196	2CH
3	005DH	WAITING	SIGNAL	0	196	FDH

RTX_EX2使用os_wait函数来等待事件。 各个任务等待的事件显示在上面的任务列表里。

TRAFFIC: 红绿灯控制器

前面的例子 RTX_EX1和RTX_EX2，仅仅显示了 RTX51 Tiny的基本特征。 在不使用RTX51的情况下，这些例子可以很容易地实现。 这个例子：行人红绿灯控制器，更加复杂而且如果没有像 RTX51一样的多任务实时操作系统的话，无法很容易地实现。

TRAFFIC是一个定时控制的红绿灯控制器。 在一个用户定义的时钟间隔时间内，可以操作红绿灯。超过时间间隔范围时，黄色信号灯闪烁。 如果一个行人按下请求按钮，红绿灯立即变成“walk”状态。 否则，红绿灯连续地工作。

红绿灯控制器命令

你可以通过8051的串口与红绿灯控制器通讯。 你可以使用DS51的串行窗口来测试|红绿灯控制器命令。 可用的串行命令列在下面的表格中。 这些命令由 ASCII字符组成。 所有命令必须用回车结束。

命令	串口文本	文字说明
Display	D	显示时钟，开始和结束时间。
Time	T *hh:mm:ss*	按 24小时格式设置当前时间。
Start	S *hh:mm:ss*	开始按 24小时格式设置启动时间。在开始和结束时间之间，红绿灯控制器正常地操作。超过这些时间范围时，黄色信号灯闪烁。
End	E *hh:mm:ss*	按 24小时格式设置结束时间。

软件

TRAFFIC应用程序由三个文件组成，这三个文件在 \C51V4\RTX_TINY\TRAFFIC 或 \CDEMO\51\RTX_TINY\TRAFFIC 目录中可以找到。

TRAFFIC.C包括红绿灯控制器的程序，被分成下面几个任务：

- **Task 0 Initialize**: 初始化串行接口并开始全部其他任务。 任务0删除它本身，因为初始化仅需要一次。
- **Task 1 Command**: 是红绿灯控制器的命令处理程序。 这个任务控制和处理串口命令接收。
- **Task 2 Clock**: 控制定时器。
- **Task 3 Blinking**: 当时钟时间超过有效时间范围之外时，黄色灯光闪烁。
- **Task 4 Lights**: 在时钟时间处于有效时间范围（在开始和结束时间之间）之内时，控制红绿灯。
- **Task 5 Button**: 读取行人按下按钮并发送信号给 lights任务。
- **Task 6 Quit**: 在串口字符流中，检查ESC字符。 如果发现ESC字符，这个任务终止一个前面指定的显示指令。

serial.c实现串行接口的中断驱动。 这个文件包括函数putchar和getkey。 高级的输入输出函数printf和getline调用这个基本输入输出子程序。 即使没有中断驱动串行输入/输出，红绿灯应用程序也可以操作。 但也不会执行。

GETLINE.C 是接到来自串口的字符的命令行编辑器。 这个源文件也被用于MEASURE应用程序。

TRAFFIC.C

```
/*****************************************************************************
*/
/*
*/
/* TRAFFIC.C: Traffic Light Controller using the C51 Compiler
*/
/*
```

```
*/
/*****************************************************************************
*/
code char menu[] =
"\n"
"+***** TRAFFIC LIGHT CONTROLLER using C51 and RTX-51 tiny *****+\n"
"| This program is a simple Traffic Light Controller. Between |\n"
"| start time and end time the system controls a traffic light |\n"
"| with pedestrian self-service. Outside of this time range |\n"
"| the yellow caution lamp is blinking. |\n"
"+ command -+ syntax -----+ function --------------------------+\n"
"| Display | D | display times |\n"
"| Time | T hh:mm:ss | set clock time |\n"
"| Start | S hh:mm:ss | set start time |\n"
"| End | E hh:mm:ss | set end time |\n"
"+----------+------------+-----------------------------------+\n";
#include <reg52.h> /* special function registers 8052
*/
#include <rtx51tny.h> /* RTX-51 tiny functions & defines
*/
#include <stdio.h> /* standard I/O .h-file
*/
#include <ctype.h> /* character functions
*/
#include <string.h> /* string and memory functions
*/.RTX TINY 51



extern getline (char idata *, char); /* external function: input line
*/
extern serial_init (); /* external function: init serial UART
*/
#define INIT 0 /* task number of task: init
*/
#define COMMAND 1 /* task number of task: command
*/
#define CLOCK 2 /* task number of task: clock
*/
#define BLINKING 3 /* task number of task: blinking
*/
#define LIGHTS 4 /* task number of task: signal
*/
#define KEYREAD 5 /* task number of task: keyread
*/
```

```
#define GET_ESC 6 /* task number of task: get_escape
*/
struct time { /* structure of the time record
*/
unsigned char hour; /* hour
*/
unsigned char min; /* minute
*/
unsigned char sec; /* second
*/
};
struct time ctime = { 12, 0, 0 }; /* storage for clock time values
*/
struct time start = { 7, 30, 0 }; /* storage for start time values
*/
struct time end = { 18, 30, 0 }; /* storage for end time values
*/
sbit red = P1^2; /* I/O Pin: red lamp output
*/
sbit yellow = P1^1; /* I/O Pin: yellow lamp output
*/
sbit green = P1^0; /* I/O Pin: green lamp output
*/
sbit stop = P1^3; /* I/O Pin: stop lamp output
*/
sbit walk = P1^4; /* I/O Pin: walk lamp output
*/
sbit key = P1^5; /* I/O Pin: self-service key input
*/
idata char inline[16]; /* storage for command input line
*/
/*****************************************************************************
*/
/* Task 0 'init': Initialize
*/
/*****************************************************************************
*/
init () _task_ INIT { /* program execution starts here
*/
serial_init (); /* initialize the serial interface
*/
os_create_task (CLOCK); /* start clock task
*/
os_create_task (COMMAND); /* start command task
*/
os_create_task (LIGHTS); /* start lights task
```

```
*/
os_create_task (KEYREAD); /* start keyread task
*/
os_delete_task (INIT); /* stop init task (no longer needed)
*/
}
bit display_time = 0; /* flag: signal cmd state display_time
*/
/*************************************************************************
*/
/* Task 2 'clock'
*/
/*************************************************************************
*/
clock () _task_ CLOCK {
while (1) { /* clock is an endless loop
*/
if (++ctime.sec == 60) { /* calculate the second
*/
ctime.sec = 0;
if (++ctime.min == 60) { /* calculate the minute
*/
ctime.min = 0;
if (++ctime.hour == 24) { /* calculate the hour
*/
ctime.hour = 0;
}
}
}
if (display_time) { /* if command_status == display_time
*/
os_send_signal (COMMAND); /* signal to task command: time changed
*/
}
os_wait (K_IVL, 100, 0); /* wait interval: 1 second
*/
}
}
struct time rtime; /* temporary storage for entry time
*/
/*************************************************************************
*/
/* readtime: convert line input to time values & store in rtime
*/
/*************************************************************************
*/
```

```
bit readtime (char idata *buffer) {
unsigned char args; /* number of arguments
*/
rtime.sec = 0; /* preset second
*/
args = sscanf (buffer, "%bd:%bd:%bd", /* scan input line for
*/
&rtime.hour, /* hour, minute and second
*/
&rtime.min,
&rtime.sec);
if (rtime.hour > 23 || rtime.min > 59 || /* check for valid inputs
*/
rtime.sec > 59 || args < 2 || args == EOF) {
printf ("\n*** ERROR: INVALID TIME FORMAT\n");
return (0);
}
return (1);
}
#define ESC 0x1B /* ESCAPE character code
*/
bit escape; /* flag: mark ESCAPE character entered
*/
/*****************************************************************************
*/
/* Task 6 'get_escape': check if ESC (escape character) was entered
*/
/*****************************************************************************
*/
get_escape () _task_ GET_ESC {
while (1) { /* endless loop
*/
if (_getkey () == ESC) escape = 1; /* set flag if ESC entered
*/
if (escape) { /* if escape flag send signal
*/
os_send_signal (COMMAND); /* to task 'command'
*/
}
}
}
/*****************************************************************************
*/
/* Task 1 'command': command processor */
/*****************************************************************************
*/
```

```
command () _task_ COMMAND {
unsigned char i;
printf (menu); /* display command menu
*/
while (1) { /* endless loop
*/
printf ("\nCommand: "); /* display prompt
*/
getline (&inline, sizeof (inline)); /* get command line input
*/
for (i = 0; inline[i] != 0; i++) { /* convert to uppercase
*/
inline[i] = toupper(inline[i]);
}
for (i = 0; inline[i] == ' '; i++); /* skip blanks
*/
switch (inline[i]) { /* proceed to command function
*/
case 'D': /* Display Time Command
*/
printf ("Start Time: %02bd:%02bd:%02bd "
"End Time: %02bd:%02bd:%02bd\n",
start.hour, start.min, start.sec,
end.hour, end.min, end.sec);
printf (" type ESC to abort\r");
os_create_task (GET_ESC); /* ESC check in display loop
*/
escape = 0; /* clear escape flag
*/
display_time = 1; /* set display time flag
*/
os_clear_signal (COMMAND); /* clear pending signals
*/
while (!escape) { /* while no ESC entered
*/
printf ("Clock Time: %02bd:%02bd:%02bd\r", /* display time
*/
ctime.hour, ctime.min, ctime.sec);
os_wait (K_SIG, 0, 0); /* wait for time change or ESC
*/
}
os_delete_task (GET_ESC); /* ESC check not longer needed
*/
display_time = 0; /* clear display time flag
*/
printf ("\n\n");
```

```
break;
case 'T': /* Set Time Command
*/
if (readtime (&inline[i+1])) { /* read time input and
*/
ctime.hour = rtime.hour; /* store in 'ctime'
*/
ctime.min = rtime.min;
ctime.sec = rtime.sec;
}
break;
case 'E': /* Set End Time Command
*/
if (readtime (&inline[i+1])) { /* read time input and
*/
end.hour = rtime.hour; /* store in 'end'
*/
end.min = rtime.min;
end.sec = rtime.sec;
}
break;
case 'S': /* Set Start Time Command */
if (readtime (&inline[i+1])) { /* read time input and
*/
start.hour = rtime.hour; /* store in 'start'
*/
start.min = rtime.min;
start.sec = rtime.sec;
}
break;
default: /* Error Handling
*/
printf (menu); /* display command menu
*/
break;
}
}
}
/*****************************************************************************
*/
/* signalon: check if clock time is between start and end
*/
/*****************************************************************************
*/
bit signalon () {
if (memcmp (&start, &end, sizeof (struct time)) < 0) {
```

```
if (memcmp (&start, &ctime, sizeof (struct time)) < 0 &&
memcmp (&ctime, &end, sizeof (struct time)) < 0) return (1);
}
else {
if (memcmp (&end, &ctime, sizeof (start)) > 0 &&
memcmp (&ctime, &start, sizeof (start)) > 0) return (1);
}
return (0); /* signal off, blinking on
*/
}
/*****************************************************************************
*/
/* Task 3 'blinking': runs if current time is outside start & end time
*/
/*****************************************************************************
*/
blinking () _task_ BLINKING { /* blink yellow light
*/
red = 0; /* all lights off
*/
yellow = 0;
green = 0;
stop = 0;
walk = 0;
while (1) { /* endless loop
*/
yellow = 1; /* yellow light on
*/
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
yellow = 0; /* yellow light off
*/
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
if (signalon ()) { /* if blinking time over
*/
os_create_task (LIGHTS); /* start lights
*/
os_delete_task (BLINKING); /* and stop blinking
*/
}
}
}
/*****************************************************************************
*/
/* Task 4 'lights': executes if current time is between start & end time
```

```
*/
/*****************************************************************************
*/
lights () _task_ LIGHTS { /* traffic light operation
*/
red = 1; /* red & stop lights on
*/
yellow = 0;
green = 0;
stop = 1;
walk = 0;
while (1) { /* endless loop
*/
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
if (!signalon ()) { /* if traffic signal time over
*/
os_create_task (BLINKING); /* start blinking
*/
os_delete_task (LIGHTS); /* stop lights
*/
}
yellow = 1;
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
red = 0; /* green light for cars
*/
yellow = 0;
green = 1;
os_clear_signal (LIGHTS);
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
os_wait (K_TMO + K_SIG, 250, 0); /* wait for timeout & signal
*/
yellow = 1;
green = 0;
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
red = 1; /* red light for cars
*/
yellow = 0;
os_wait (K_TMO, 30, 0); /* wait for timeout: 30 ticks
*/
stop = 0; /* green light for walkers
*/
walk = 1;
```

```
os_wait (K_TMO, 100, 0); /* wait for timeout: 100 ticks
*/
stop = 1; /* red light for walkers
*/
walk = 0;
}
}
/*****************************************************************************
*/
/* Task 5 'keyread': process key stroke from pedestrian push button
*/
/*****************************************************************************
*/
keyread () _task_ KEYREAD {
while (1) { /* endless loop
*/
if (key) { /* if key pressed
*/
os_send_signal (LIGHTS); /* send signal to task lights
*/
}
os_wait (K_TMO, 2, 0); /* wait for timeout: 2 ticks
*/
}
}
```

SERIAL.C

```
/*****************************************************************************
*/
/*
*/
/* SERIAL.C: Interrupt Controlled Serial Interface for RTX-51 tiny
*/
/*
*/
/*****************************************************************************
*/
#include <reg52.h> /* special function register 8052
*/
#include <rtx51tny.h> /* RTX-51 tiny functions & defines
*/
#define OLEN 8 /* size of serial transmission buffer
*/
unsigned char ostart; /* transmission buffer start index
*/
unsigned char oend; /* transmission buffer end index
*/
```

```
idata char outbuf[OLEN]; /* storage for transmission buffer
*/
unsigned char otask = 0xff; /* task number of output task
*/
#define ILEN 8 /* size of serial receiving buffer
*/
unsigned char istart; /* receiving buffer start index
*/
unsigned char iend; /* receiving buffer end index
*/
idata char inbuf[ILEN]; /* storage for receiving buffer
*/
unsigned char itask = 0xff; /* task number of output task
*/
#define CTRL_Q 0x11 /* Control+Q character code
*/
#define CTRL_S 0x13 /* Control+S character code
*/
bit sendfull; /* flag: marks transmit buffer full
*/
bit sendactive; /* flag: marks transmitter active
*/
bit sendstop; /* flag: marks XOFF character
*/
/*************************************************************************
*/
/* putbuf: write a character to SBUF or transmission buffer
*/
/*************************************************************************
*/
putbuf (char c) {
if (!sendfull) { /* transmit only if buffer not full
*/
if (!sendactive && !sendstop) { /* if transmitter not active:
*/
sendactive = 1; /* transfer the first character direct
*/
SBUF = c; /* to SBUF to start transmission
*/
}
else { /* otherwize:
*/
outbuf[oend++ & (OLEN-1)] = c; /* transfer char to transmission buffer
*/
if (((oend ^ ostart) & (OLEN-1)) == 0) sendfull = 1;
} /* set flag if buffer is full
```

```
*/
}
}
/*****************************************************************************
*/
/* putchar: interrupt controlled putchar function
*/
/*****************************************************************************
*/
char putchar (char c) {
if (c == '\n') { /* expand new line character:
*/
while (sendfull) { /* wait for transmission buffer empty
*/
otask = os_running_task_id (); /* set output task number
*/
os_wait (K_SIG, 0, 0); /* RTX-51 call: wait for signal
*/
otask = 0xff; /* clear output task number
*/
}
putbuf (0x0D); /* send CR before LF for <new line>
*/
}
while (sendfull) { /* wait for transmission buffer empty
*/
otask = os_running_task_id (); /* set output task number
*/
os_wait (K_SIG, 0, 0); /* RTX-51 call: wait for signal
*/
otask = 0xff; /* clear output task number
*/
}
putbuf (c); /* send character
*/
return (c); /* return character: ANSI requirement
*/
}
/*****************************************************************************
*/
/* _getkey: interrupt controlled _getkey
*/
/*****************************************************************************
*/
char _getkey (void) {
while (iend == istart) {
```

```
itask = os_running_task_id (); /* set input task number
*/
os_wait (K_SIG, 0, 0); /* RTX-51 call: wait for signal
*/
itask = 0xff; /* clear input task number
*/
}
return (inbuf[istart++ & (ILEN-1)]);
}
/*************************************************************************
*/
/* serial: serial receiver / transmitter interrupt
*/
/*************************************************************************
*/
serial () interrupt 4 using 2 { /* use registerbank 2 for interrupt
*/
unsigned char c;
bit start_trans = 0;
if (RI) { /* if receiver interrupt
*/
c = SBUF; /* read character
*/
RI = 0; /* clear interrupt request flag
*/
switch (c) { /* process character
*/
case CTRL_S:
sendstop = 1; /* if Control+S stop transmission
*/
break;
case CTRL_Q:
start_trans = sendstop; /* if Control+Q start transmission
*/
sendstop = 0;
break;
default: /* read all other characters into inbuf
*/
if (istart + ILEN != iend) {
inbuf[iend++ & (ILEN-1)] = c;
}
/* if task waiting: signal ready
*/
if (itask != 0xFF) isr_send_signal (itask);
break;
}
```

```
}
if (TI || start_trans) { /* if transmitter interrupt
*/
TI = 0; /* clear interrupt request flag
*/
if (ostart != oend) { /* if characters in buffer and
*/
if (!sendstop) { /* if not Control+S received
*/
SBUF = outbuf[ostart++ & (OLEN-1)]; /* transmit character
*/
sendfull = 0; /* clear 'sendfull' flag
*/
/* if task waiting: signal ready
*/
if (otask != 0xFF) isr_send_signal (otask);
}
}
else sendactive = 0; /* if all transmitted clear 'sendactive'
*/
}
}
/***************************************************************************
*/
/* serial_init: initialize serial interface
*/
/***************************************************************************
*/
serial_init () {
SCON = 0x50; /* mode 1: 8-bit UART, enable receiver
*/
TMOD |= 0x20; /* timer 1 mode 2: 8-Bit reload
*/
TH1 = 0xf3; /* reload value 2400 baud
*/
TR1 = 1; /* timer 1 run
*/
ES = 1; /* enable serial port interrupt
*/
}
```

GETLINE.C

```
/***************************************************************************
*/
/*
*/
/* GETLINE.C: Line Edited Character Input
```

```
*/
/*
*/
/*****************************************************************************
*/
#include <stdio.h>
#define CNTLQ 0x11
#define CNTLS 0x13
#define DEL 0x7F
#define BACKSPACE 0x08
#define CR 0x0D
#define LF 0x0A
/***************/
/* Line Editor */
/***************/
void getline (char idata *line, unsigned char n) {
unsigned char cnt = 0;
char c;
do {
if ((c = _getkey ()) == CR) c = LF; /* read character
*/
if (c == BACKSPACE || c == DEL) { /* process backspace
*/
if (cnt != 0) {
cnt--; /* decrement count
*/
line--; /* and line pointer
*/
putchar (0x08); /* echo backspace
*/
putchar (' ');
putchar (0x08);
}
}
else if (c != CNTLQ && c != CNTLS) { /* ignore Control S/Q
*/
putchar (*line = c); /* echo and store character
*/
line++; /* increment line pointer
*/
cnt++; /* and count
*/
}
} while (cnt < n - 1 && c != LF); /* check limit and line feed
*/
*line = 0; /* mark end of string
```

```
*/
}
```

编译和连接TRAFFIC

在 DOS提示符下键入下列命令来编译和连接 TRAFFIC。

```
C51 TRAFFIC.C DEBUG OBJECTEXTEND RF (TRAFFIC.REG)
C51 SERIAL.C DEBUG OBJECTEXTEND RF (TRAFFIC.REG)
C51 GETLINE.C DEBUG OBJECTEXTEND RF (TRAFFIC.REG)
BL51 @TRAFFIC.LIN
```

另外，有一个名为 TRAFFIC.BAT的批处理文件，你可以用它编译、连接、并且自动地运行DS51。

测试并调试 TRAFFIC

编译和连接 TRAFFIC后，你可以使用 ds51测试它。 键入 DS51 TRAFFIC来运行 DS51并装入DS51.INI初始化文件。 这个文件会自动地装入输入输出驱动程序、加载traffic程序、加载一用于显示任务状态的包含文件、激活红绿灯的断点、定义行人按钮函数（使用F4激活），并且启动TRAFFIC应用程序。 下面是DS51.INI的列表。

```
load ..\..\ds51\8052.iof /* load 8052 CPU driver*/
include dbg_tiny.inc /* load debug function for RTX51 Tiny */
/* define watch variables */
ws red
ws yellow
ws green
ws stop
ws walk
/* set P1.5 to zero: Key Input */
PORT1 &= ~0x20;
/* define a debug function for the pedestrian push button */
signal void button (void) {
PORT1 |= 0x20; /* set Port1.5 */
twatch (50000); /* wait 50 ms */
PORT1 &= ~0x20; /* reset Port1.5 */
}
/* define F4 key as call to button () */
set F4="button ()"
```

你可以用 GO命令执行 TRAFFIC应用程序：g

当 ds51开始执行TRAFFIC时，串口窗口会显示命令菜单并等待你键入一个命令。 用Alt+S转换到串口窗口；按下 d键和ENTER键。 这会显示红绿灯的当前时间和启动与终止时间范围。 例如：

```
Start Time: 07:30:00 End Time: 18:30:00
Clock Time: 12:00:11 type ESC to abort
```

当程序运行时，你可以监视交通信号灯的红、黄、绿灯的变化。 行人按钮使用 F4模拟。 按下F4会看到交通信号灯切换到红灯而且行走灯亮。 你可以像以前一样使用F3显示任务状态。 将显示下面的任务信息：

Task ID	Start	State	Wait for Event	Signal	Timer	Stack
0	0026H	DELETED		0	131	84H
1	00D1H	WAITING	SIGNAL	0	131	84H

2	0043H	WAITING	TIMEOUT	0	5	86H
3	0278H	DELETED		0	131	88H
4	02ACH	WAITING	SIGNAL & TIMEOUT	0	220	88H
5	032BH	WAITING	TIMEOUT	0	1	8AH
6	000EH	WAITING	SIGNAL	0	131	FBH

如果Exe窗口不够显示全部的状态文本，你可以按下ALT+R删除寄存器窗口。 你还可以增加Exe窗口的垂直尺寸。 按下ALT+E选择Exe窗口然后键入ALT+U数次来调整窗口的大小。

当你想停止 DS51时，在 DS51退出提示符下键入 EXIT。

A

B

C

D

E

F

I

K

L

M

N

O

P

R

S

T

U

Germany and Europe

KEIL ELEKTRONIK GmbH
Bretonischer Ring 15
D-85630 Grasbrunn b. München
Tel: (49) (089) 46 50 57
FAX: (49) (089) 46 81 62

Keil Software is market in the United States and Canada also under the Franklin Software, Inc.

KEIL ELEKTRONIK GmbH has representatives in the following countries: Australia, Austria, Belgium, CFR, Denmark, Finland, France, Germany, India, Ireland, Israel, Italy, Netherlands, Norway, Poland, Spain, South Africa, Sweden, Switzerland, Taiwan, United Kingdom, United States and Canada.

Contact KEIL ELEKTRONIK GmbH to obtain the name and address of the distributor nearest to you.

Printed in Germany 2-95, Document #9443-1.

RTX51 TINY

实时操作系统

用户指南 2.95