



深圳市英蓓特信息技术有限公司



# RTX实时操作系统内核演示

RealView微处理器开发工具

## 概述

### 目的

通过一个实例来演示RTX实时内核的工作机理，并学会怎样基于它来编写应用程序。以下会列出一些RTX的重要优势：

- 容易使用
- 具有多功能性
- 占用空间小
- 它的配置和调试支持微处理器开发工具(MDK)的集成

### 运行环境

RealView MDK v3.15 或更高版本

在讨论中所用软件的例子是基于Keil MCBSTM32评估板的。此文档中的图片是在仿真器下运行例程的截图，用户也可把这些例子下载到目标板中，以便运行在实际的硬件环境中。

注意：这个示例是在MDK的评估版上开发。在运行评估版时，它会报告一个警告信息：工程的最大代码限制为16KB个字节。

注意：这个实例需要用户有一定的MDK知识基础。在MCBSTM32E环境下的实例开发也为用户初次使用MDK提供了一个好的起点。

### 调试



文中既有示范，又有对用户的解释  
这个符号表示控制按钮。

### 开始建立

这个过程要在第一次运行示例之前进行。



复制文件夹  
../Keil/ARM/Boards/Keil/MCBSTM32/RTX\_Traffic  
进入新示例文件夹中  
../Keil/ARM/Boards/Keil/MCBSTM32/Demo

## 工程设置

打开并清除MDK



Project > Close Project

打开示例工程



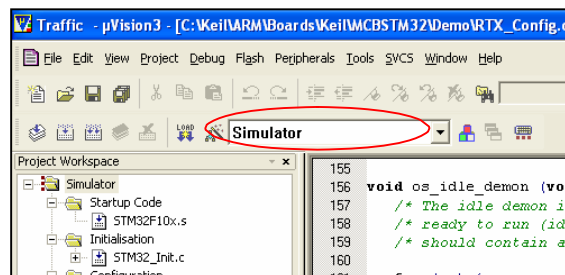
Project > Open Project...

导入../Keil/ARM/Boards/Keil/MCBSTM32/Demo/Traffic.Uv2

工程目标



选择工程Simulator



在RTX\_Traffic例子中，仿真器需要提前配置，用来协助RTX工作和模拟STM32微处理器运行代码。

### 目标选择

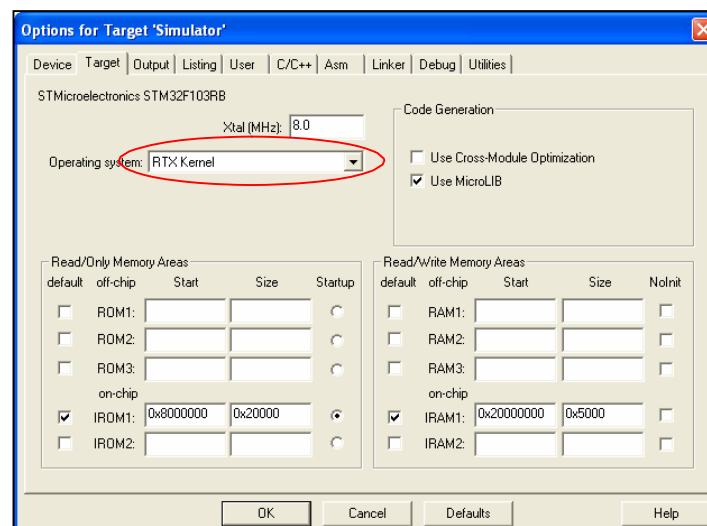
μVision工程必须包含应用程序用到的操作系统信息，从而使得它的源代码能够连接到RTOS库。

可以通过“Option for Target”对话框选择。



单击‘Options for Target’按钮

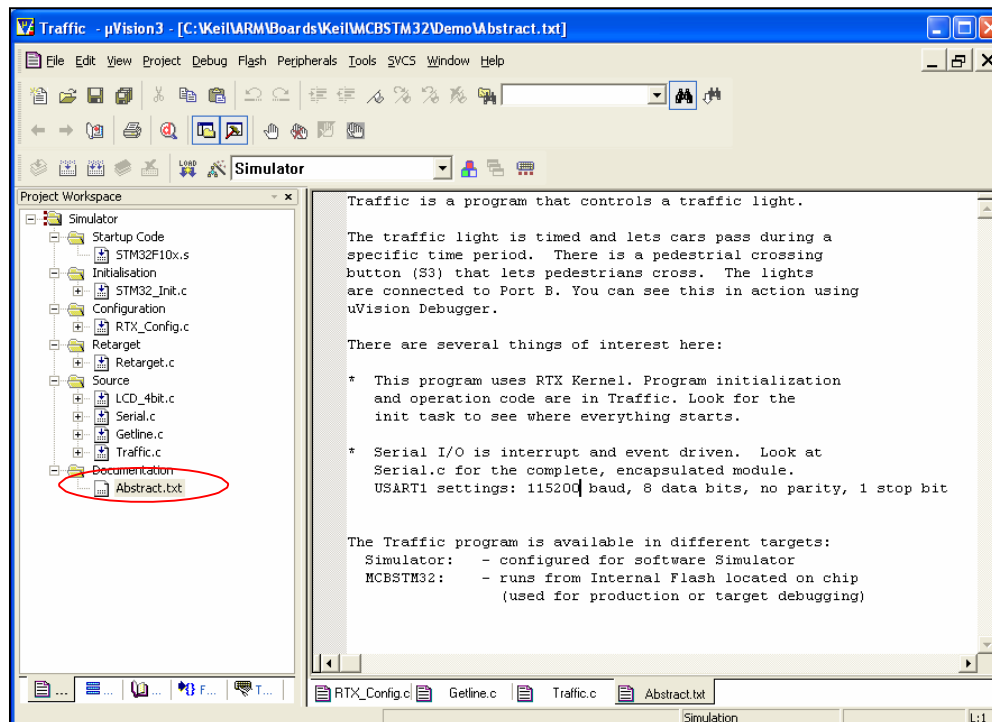
点击‘Target’选项卡



## RTX\_Traffic 示例

### RTX\_Traffic 描述

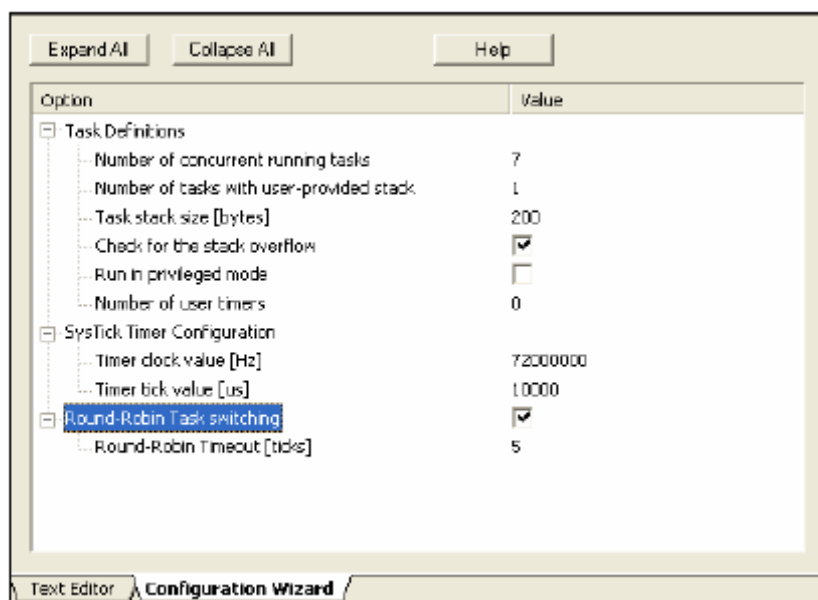
文件Abstract.txt描述了RTX\_Traffic示例的基本功能。



### RTX\_Traffic 源文件

以下源文件中包含的代码用来配置和使用RTX功能。

RTX\_Config.c 是RTX的一部分，它允许使用者去配置所需参数，例如设置同时发生的任务的最大数量，每一个任务分配的堆栈的大小，系统时钟的间隔，调度的类型等等。所有的这些选项都能通过MDK的配置向导选择





## 深圳市英蓓特信息技术有限公司

Traffic.c 包含应用程序的代码。这些代码是基于任务的，它们能被RTX调度Step 4

以下源文件对于裸机上和实时操作系统上的应用程序都是必需的。

STM32F10x.s 是微控制器的启动代码，包括异常向量表处理和复位处理。

STM32\_Init.c 是微控制器的初始化代码，这些代码用来配置内存和外设的。

Retarget.c 包含了重写 C 语言代码库的函数。这个文件包含了类似于 fputc()之类函数的自定义实现，因此 printf()函数调用直接输出到串行通信端口（例如 UART）。

LCD\_4bit.c 和 Serial.c 提供了控制字符 LCD 和 UART 的代码。

Getline.c 包含了串口接收到信息的解码函数。



用 5 分钟时间熟悉一下这些代码

RTX\_Traffic 操作

RTX\_Traffic 是一个真实嵌入式系统的写照。

在这种系统中，一个单一的控制单元可以并行地完成一系列的操作，且每种操作都需要对外部信号做出及时地响应。

Traffic.c 中的代码被分成很多同时执行的任务。每种任务执行一种操作。可以通过申明方式来判断该函数到底是一个普通函数还是一个任务：

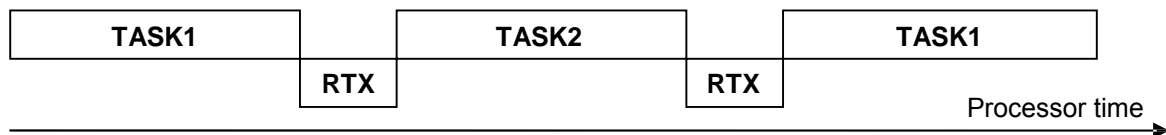
```
void init (void) _task { ... }
```

实时系统中的通用接口给人的印象是所有的任务都同时在运行，但是实际上处理器只能在某一个特定的时间点内执行某一个任务的指令，这是通过RTX内核完成的。

每一个时钟周期，系统时钟产生一个中断，并把控制信息传给 RTX。

RTX 使用时钟周期给不同的任务分配时间片。

时钟周期定义在 RTX\_Config.c 文件中，在 RTX\_Traffic 操作中一个时钟周期是 10 微秒。



因为 RTX 在每秒种内分配给每个任务若干个时间片，所以每个任务都能及时的响应外部事件。这种模式下，系统看起来好像是同时做若干操作。

## RTX\_Traffic Tasks

RTX\_Traffic 最多可以并行地执行 7 个任务。



在 Traffic.c 找到每个任务的代码，并且与下面的描述对应。

```
void clock (void) task { }
```

clock()是由 RTX 每秒钟触发的。它更新一个时钟变量并返回时钟变量的值。因此使用 RTX，用户不需要配置时钟来每秒钟产生一个中断，也不需要修改中断服务程序来处理它。

```
void blinking (void) task { }  
void lights (void) task { }  
void lcd (void) task { }
```

blinking()实现“紧急交通灯”。lights()实现“工作交通灯”。lcd()实 MCBSTM32E 字符显示屏的驱动程序。

这 3 个任务有相同的需要：周期性地更新一个输出设备。为了使系统高效地运行，外围设备仅仅在必要时才更新。这种机制由 RTX 的 os\_dly\_wait()函数轻松实现。os\_dly\_wait()在置位、清除交通灯指令和更新 LCD 状态的指令之间插入了一个固定的延时。

```
void get_escape (void) task { }  
void keyread (void) task { }
```

get\_escape()是 RTX 周期性的触发的，用来检查是否从串口接受了一个 ESC 命令。类似地，keyread()是由 RTX 触发的，用来周期性地检测用户按键。

外围设备使用 RTX 轮询非中断方式管理。这种方式由 2 个优点：

- 1 它有助于减少中断处理程序，因此使系统对外部事件的响应更加快速。
  - 2 外围设备轮探的频率是很容易配置的。
- 这种方式下，系统会对外部信号按需求尽快响应，而不会浪费处理器时间。

```
void command (void) task { }
```

command()通过串口与用户进行通信。实质上，这个任务类似于 keyread()，因为它周期性的查探串口是否有新的命令并解码命令。二者主要的区别就是 command()不通过使用 os\_dly\_wait ()函数进入睡眠状态，而是试图尽可能频繁地运行。这种机制使得它在系统中响应消息最积极。

创建 RTX\_Traffic 并启动仿真器



通过 Build Target' 按钮创建工程。

这个例子工程创建时没有错误和警报。



使用 'Start/Stop Debug Session' 按钮启动仿真器

**任务初始化和调度**

任务初始化

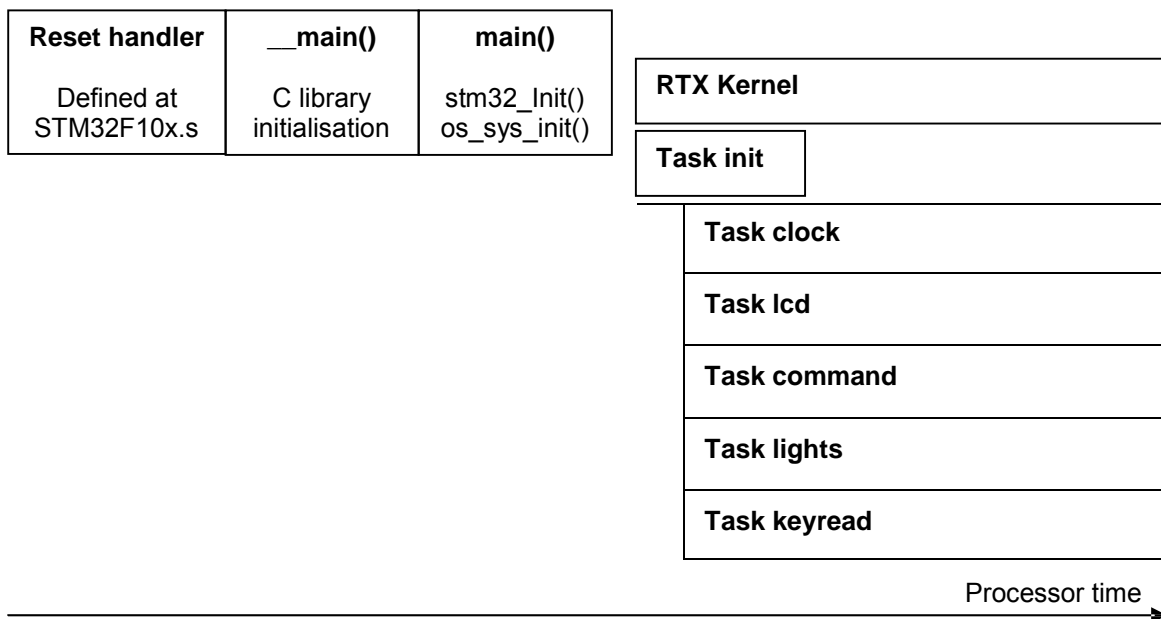
Traffic.c 在任务中是结构化的，因此 main()函数仅仅用来运行初始化代码并启动 RTX.

```
int main (void)    {
    stm32_init ();
    os_sys_init (init); }
```

启动后，RTX 启动任务 init ()。这个任务启动所有其他的任务，然后停止。

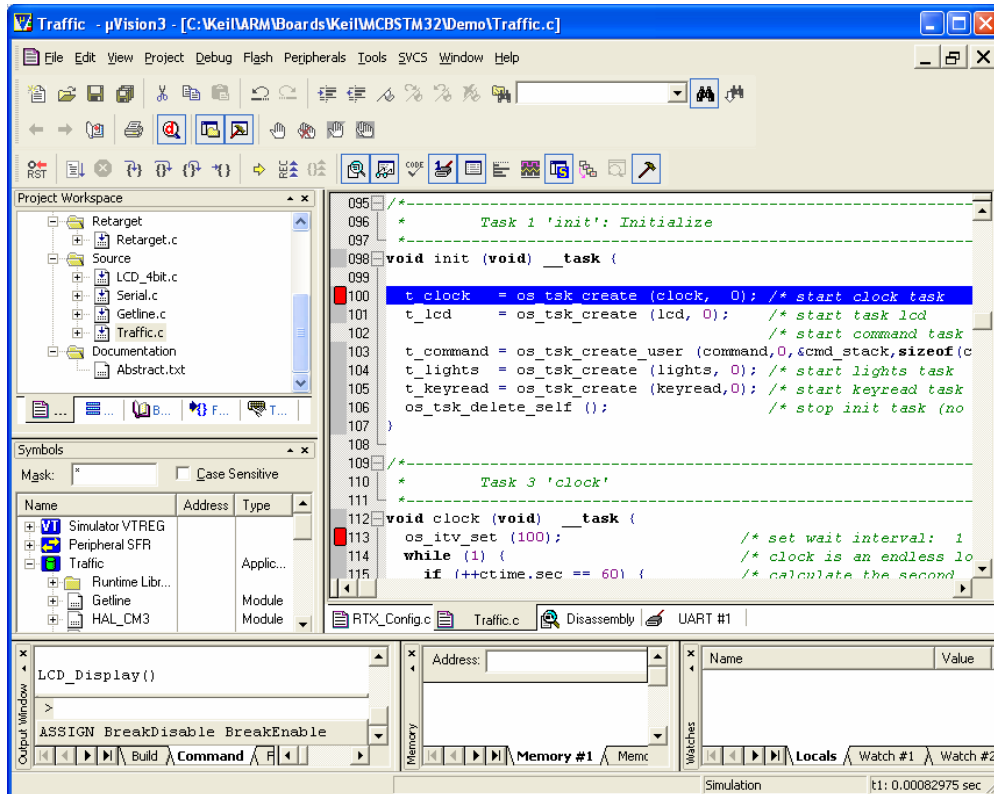
```
void init (void) __task
{
    t_clock = os_tsk_create (clock, 0);
    t_lcd = os_tsk_create (lcd, 0);
    t_command = os_tsk_create_user (command,0,&cmd_stack,sizeof(cmd_stack));
    t_lights = os_tsk_create (lights, 0);
    t_keyread = os_tsk_create (keyread,0);
    os_tsk_delete_self ();
}
```

下面的图标表明了不同的函数是如何及时地执行的。





双击任务的第一行的左边来设置断点。对任务 tasks init, clock, lcd, command, lights and keyread 都做同样的操作。







运行程序，直到仿真器在断点处暂停

在不同断点处，随着上一个任务的结束，装载下一个任务。顺序为: init -> clock -> lcd -> command -> lights -> keyread。

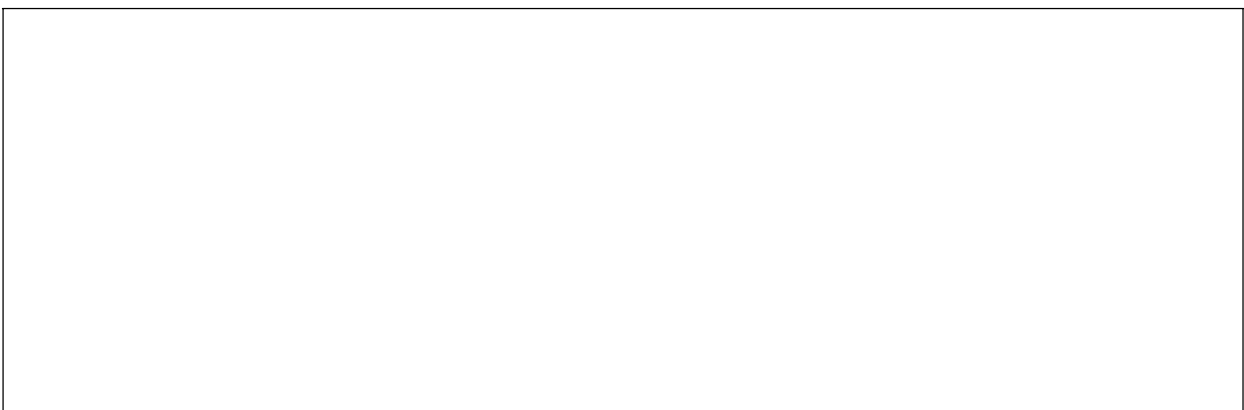
退出断点后，使用while (1)或for(;;)结构，使得每个任务都进入死循环。

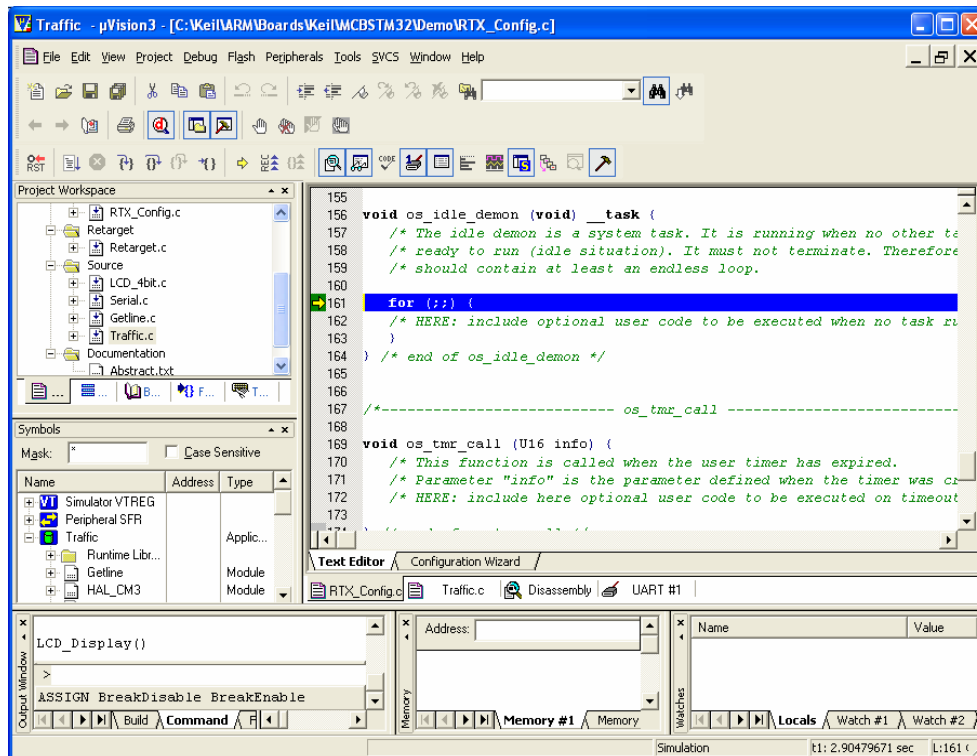
```
void my_task (void) __task      {  
  
    // Instructions that set up the task  
  
    while (1) {  
  
        // Instructions that are executed all over again  
        // Optional RTX call to request "wake-up time"  
  
    }  
  
}
```



停止程序

一旦函数os\_idle\_demon()内部运行的任务停止，则只有默认的任务作为RTX运行，其他用户任务都处于睡眠状态。




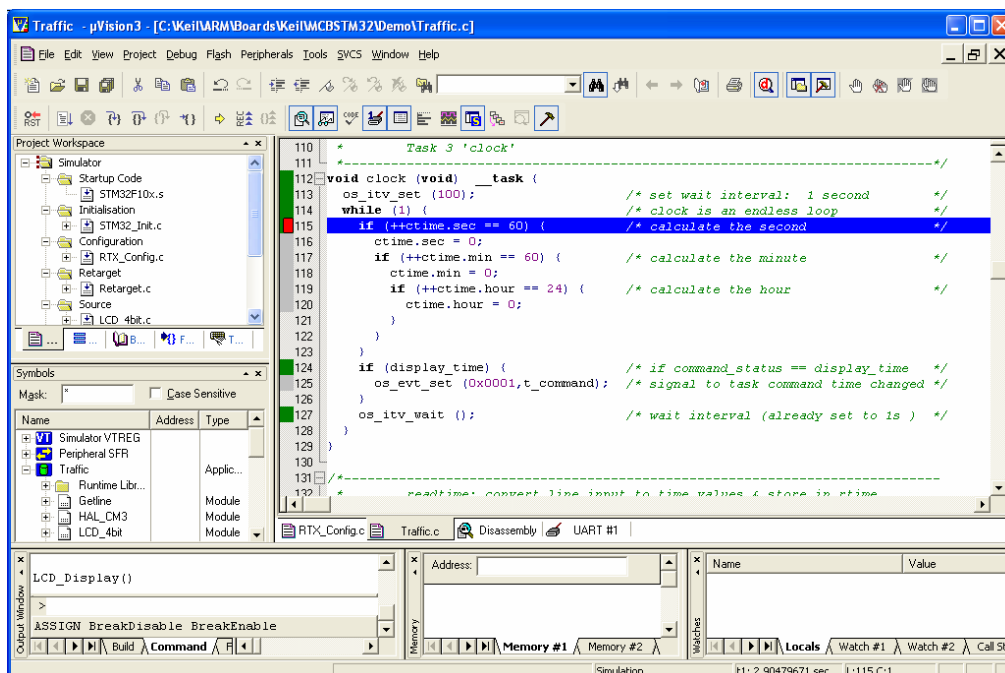


### 任务调度

这部分阐述了 RTX 如何提前唤醒不同任务。

 取消所有断点。

 在任务 tasks clock() 和 keyread()中的while(1)之后设置断点。





运行程序直到断点处暂停

可以观测到在单个任务clock()内，任务keyread()执行了20次。原因是clock()每100个系统周期运行一次。

```
Traffic.c: line 113      os_itv_set (100);          /* set wait interval: 1 second */
```

...while keyread() runs every 5 system ticks.

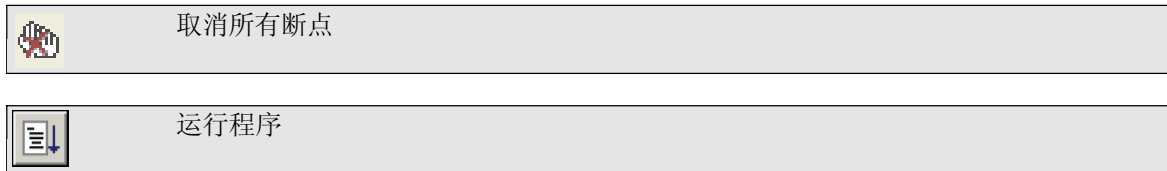
```
Traffic.c: line 314    os_dly_wait (5);          /* wait for timeout: 5 ticks */
```



取消所有断点

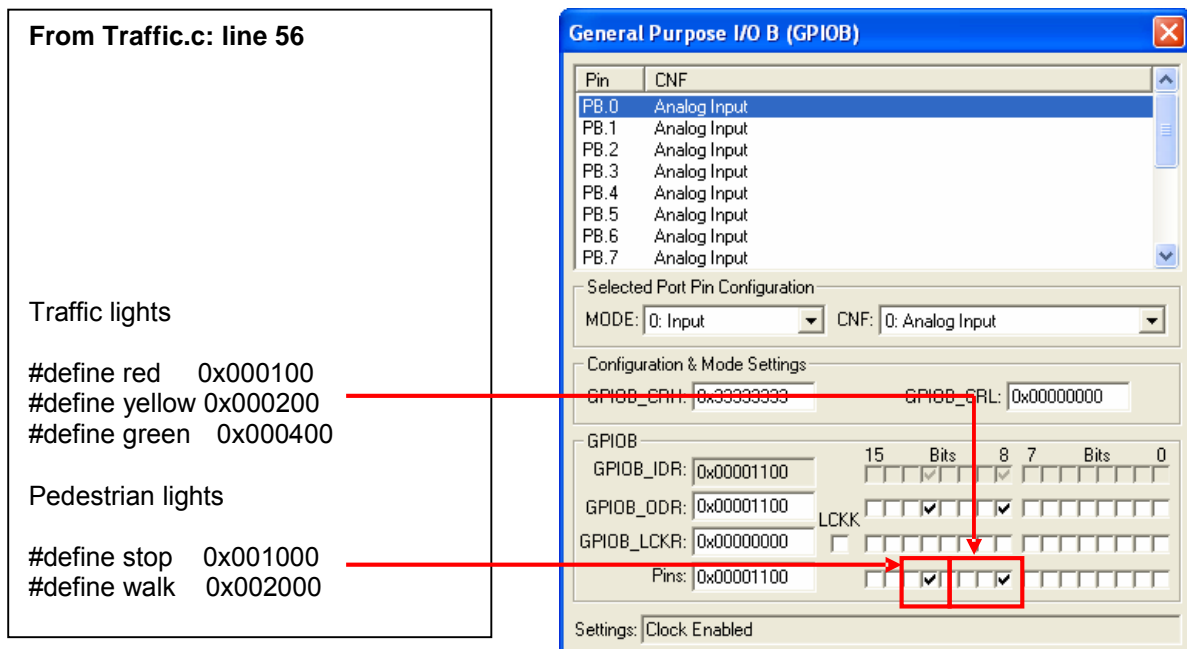
## 用µVision Debugger分析代码

这一部分阐述RTX如何集成到µVision Debugger中。



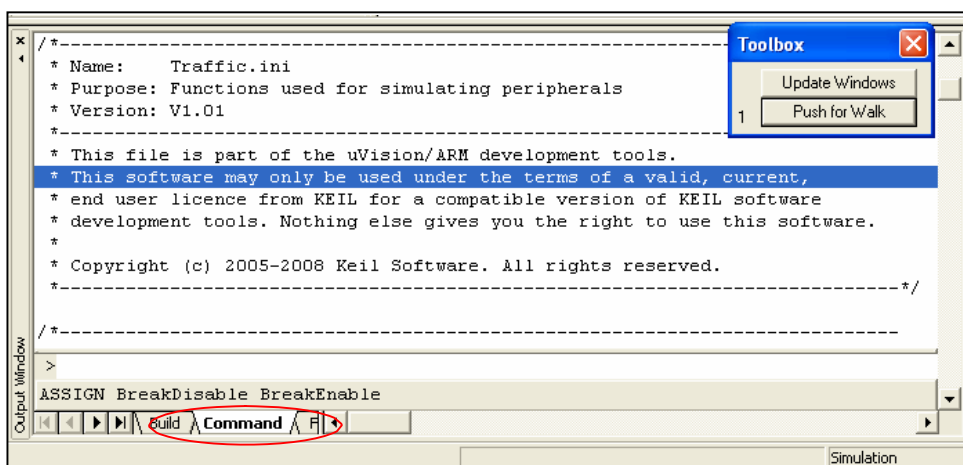
外设窗口

外设窗口可以用来监视GPIO的微控器状态，微控器连接到输出，如同交通灯来控制车辆和行人一样。



工具箱

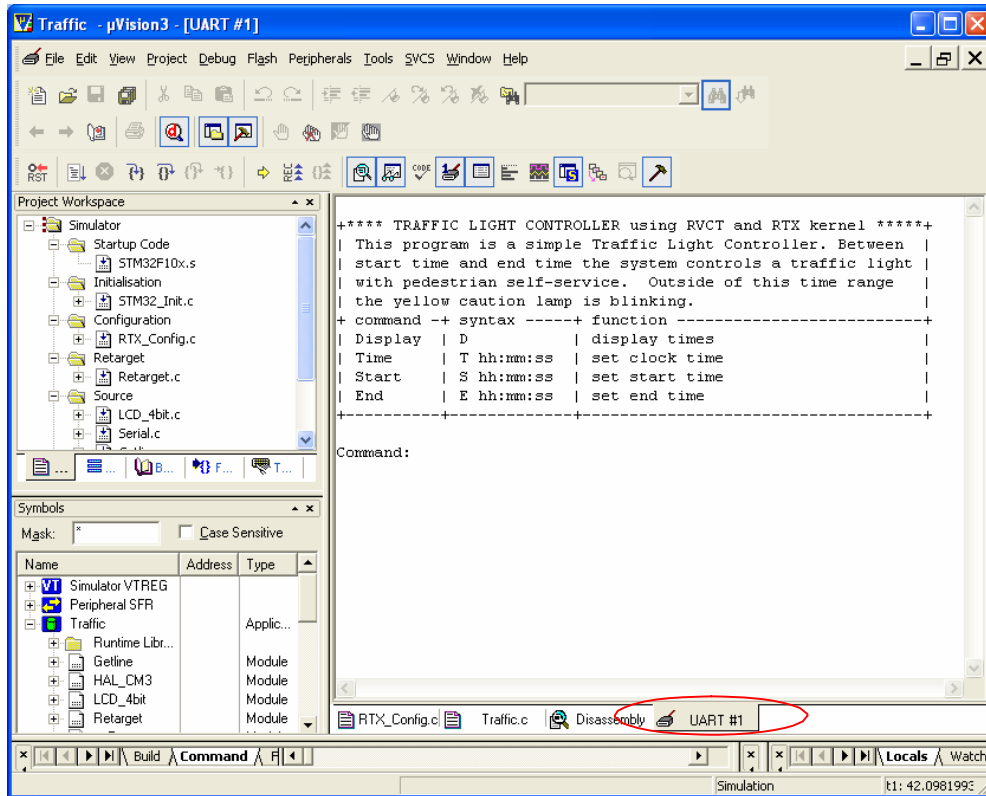
文件Traffic.ini中包含了调试脚本，用来打开仿真器的工具箱。



使用工具箱中“Push for Walk”按钮模拟行人过马路。在外设窗口中观测交通灯的作用。

## UART 窗口

点击 UART #1 标签与任务command()进行通信。



## RTX 内核分析窗口

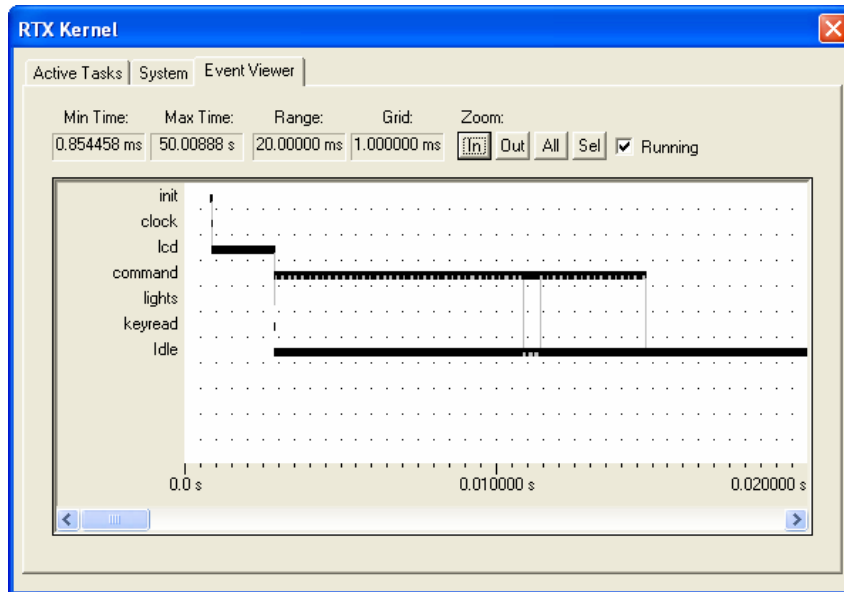


Active Tasks窗口中显示了任务的状态和使用情况。

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
2	clock	1	WAIT_ITV	25			40%
3	lcd	1	WAIT_DLY	190			40%
4	command	1	WAIT_OR		0x0000	0x0100	19%
5	lights	1	WAIT_DLY	125			40%
6	keyread	1	WAIT_DLY	5			40%
255	os_idle_demon	0	RUNNING				0%

System窗口中显示RTX的配置和处于活动状态任务的数量。

Event Viewer窗口中显示任务执行的时间段。例如：在任务开始执行的时候，通过窗口可以观测到不同的任务在什么时候被装载。



使用 $\mu$ Vision Debugger提供的所有资源来分析RTX\_Traffic的执行。

您已经浏览完了RTX实时内核演示。

如果您想做一下练习，可以试试以下内容。

更改时钟周期为5,000us。

对程序的执行有什么影响？

出现什么问题？

需要在代码里面改动哪些来处理这个问题？

在启动代码里创建一个任务，每隔2s触发一次GPIO。

在Event Viewer窗口中观测到什么结果？

将ULINK2连接到MCBSTM32E板上，在实际的硬件环境中运行例程。