

## Chapter 8. RTX-51 Real-Time Operating System

RTX51 is a multitasking real-time operating system for the 8051 family. RTX51 simplifies system and software design of complex and time-critical projects. RTX51 is a powerful tool to manage several jobs (tasks) on a single CPU. There are two distinct versions of **RTX51**:

**RTX51 Full** which performs both round-robin and preemptive task switching with 4 task priorities and can be operated with interrupt functions in parallel. RTX51 supports signal passing; message passing with a mailbox system and semaphores. The `os_wait` function of RTX51 can wait for the following events: interrupt; timeout; signal from task or interrupt; message from task or interrupt; semaphore.

**RTX51 Tiny** which is a subset of RTX51 Full. RTX51 Tiny easily runs on single-chip systems without off-chip memory. However, program using RTX51 Tiny can access off-chip memory. RTX51 Tiny allows round-robin task switching, supports signal passing and can be operated with interrupt functions in parallel. The `os_wait` function of RTX51 Tiny can wait for the following events: timeout; interval; signal from task or interrupt.

The rest of this section uses RTX-51 to refer to RTX-51 Full and RTX-51 Tiny. Differences between the two are stated where applicable.

### Introduction

Many microcontroller applications require simultaneous execution of multiple jobs or tasks. For such applications, a real-time operating system (RTOS) allows flexible scheduling of system resources (CPU, memory, etc.) to several tasks. RTX-51 implements a powerful RTOS that is easy to use. RTX-51 works with all 8051 derivatives.

You write and compile RTX-51 programs using standard C constructs and compiling them with C51. Only a few deviations from standard C are required in order to specify the task ID and priority. RTX-51 programs also require that you include the `RTX51.H` or `RTX51TNY.H` header file. When you select in the  $\mu$ Vision2 dialog Options for Target - Target the operating system, the linker adds the appropriate RTX-51 library file.

## Single Task Program

A standard C program starts execution with the main function. In an embedded application, main is usually coded as an endless loop and can be thought of as a single task that is executed continuously. For example:

```
int counter;

void main (void) {
    counter = 0;

    while (1) {
        counter++;
    }
}
```

/\* repeat forever \*/  
/\* increment counter \*/

## Round-Robin Task Switching

RTX51 Tiny allows a quasi-parallel, simultaneous execution of several tasks. Each task is executed for a predefined timeout period. A timeout suspends the execution of a task and causes another task to be started. The following example uses this round-robin task switching technique.

### Simple C Program using RTX51

```
#include <rtx51tiny.h> /* Definitions for RTX51 Tiny */
int counter0;
int counter1;

job0 () _task_ 0 {

    os_create_task (1); /* Mark task 1 as "ready" */

    while (1) {
        counter0++;
    }
}

job1 () _task_ 1 {
    while (1) {
        counter1++;
    }
}
```

/\* Endless loop \*/  
/\* Increment counter 0 \*/  
/\* Endless loop \*/  
/\* Increment counter 1 \*/

# 8

RTX51 starts the program with task 0 (assigned to job0). The function `os_create_task` marks task 1 (assigned to job1) as ready for execution. These two functions are simple count loops. After the timeout period has been completed, RTX51 interrupts job0 and begins execution of job1. This function even reaches the timeout and the system continues with job0.

## The os\_wait Function

The `os_wait` function provides a more efficient way to allocate the available processor time to several tasks. `os_wait` interrupts the execution of the current task and waits for the specified event. During the time in which a task waits for an event, other tasks can be executed.

## Wait for Timeout

RTX51 uses an 8051 timer in order to generate cyclic interrupts (timer ticks). The simplest event argument for `os_wait` is a timeout, where the currently executing task is interrupted for the specified number of timer ticks. The following uses timeouts for the time delay.

### Program with os\_wait Function

```
#include <rtx51tny.h>          /* Definitions for RTX51 Tiny */

int counter0;
int counter1;

job0 () _task_ 0 {
    os_create_task (1);

    while (1) {
        counter0++;           /* Increment counter 0      */
        os_wait (K_TMO, 3, 0); /* Wait 3 timer ticks  */
    }
}

job1 () _task_ 1 {
    while (1) {
        counter1++;           /* Increment counter 1      */
        os_wait (K_TMO, 5, 0); /* Wait 5 timer ticks      */
    }
}
```

This program is similar to the previous example with the exception that `job0` is interrupted with `os_wait` after `counter0` has been incremented. RTX51 waits three timer ticks until `job0` is ready for execution again. During this time, `job1` is executed. This function also calls `os_wait` with a timeout of 5 ticks. The result: `counter0` is incremented every three ticks and `counter1` is incremented every five timer ticks.

## Wait for Signal

Another event for `os_wait` is a signal. Signals are used for task coordination: if a task waits with `os_wait` until another task issues a signal. If a signal was previously sent, the task is immediately continued.

### Program with Wait for Signal.

```
#include <rtx51tiny.h>          /* Definitions for RTX51 Tiny */

int counter0;
int counter1;

job0 () _task_ 0 {
    os_create_task (1);

    while (1) {
        if (++counter0 == 0) {    /* On counter 0 overflow      */
            os_send_signal (1);  /* Send signal to task 1 */
        }
    }
}

job1 () _task_ 1 {
    while (1) {
        os_wait (K_SIG, 0, 0);   /* Wait for signal; no timeout */
        counter1++;              /* Increment counter 1         */
    }
}
```

In this example, task 1 waits for a signal from task 0 and therefore processes the overflow from counter0.

## Preemptive Task Switching

The full version of RTX51 provides preemptive task switching. This feature is not included in RTX51 Tiny. It is explained here to provide a complete overview of multitasking concepts.

In the previous example, task 1 is not immediately started after a signal has arrived, but only after a timeout occurs for task 0. If task 1 is defined with a higher priority than task 0, by means of preemptive task switching, task 1 is started immediately after the signal has arrived. The priority is specified in the task definition (priority 0 is the default value).

## RTX51 Technical Data

Description	RTX-51 Full	RTX-51 Tiny
Number of tasks	256; max. 19 tasks active	16
RAM requirements	40 .. 46 bytes DATA 20 .. 200 bytes IDATA (user stack) min. 650 bytes XDATA	7 bytes DATA 3 * <task count> IDATA
Code requirements	6KB .. 8KB	900 bytes
Hardware requirements	timer 0 or timer 1	timer 0
System clock	1000 .. 40000 cycles	1000 .. 65535 cycles
Interrupt latency	< 50 cycles	< 20 cycles
Context switch time	70 .. 100 cycles (fast task) 180 .. 700 cycles (standard task) depends on stack load	100 .. 700 cycles depends on stack load
Mailbox system	8 mailboxes with 8 integer entries each	not available
Memory pool system	up to 16 memory pools	not available
Semaphores	8 * 1 bit	not available

## Overview of RTX51 Routines

The following table lists some of the RTX-51 functions along with a brief description and execution timing (for RTX-51 Full).

Function	Description	CPU Cycles
<code>isr_rcv_message</code> †	Receive a message (call from interrupt).	71 (with message)
<code>isr_send_message</code> †	Send a message (call from interrupt).	53
<code>isr_send_signal</code>	Send a signal to a task (call from interrupt).	46
<code>os_attach_interrupt</code> †	Assign task to interrupt source.	119
<code>os_clear_signal</code>	Delete a previously sent signal.	57
<code>os_create_task</code>	Move a task to execution queue.	302
<code>os_create_pool</code> †	Define a memory pool.	644 (size 20 * 10 bytes)
<code>os_delete_task</code>	Remove a task from execution queue.	172
<code>os_detach_interrupt</code> †	Remove interrupt assignment.	96
<code>os_disable_isr</code> †	Disable 8051 hardware interrupts.	81
<code>os_enable_isr</code> †	Enable 8051 hardware interrupts.	80
<code>os_free_block</code> †	Return a block to a memory pool.	160
<code>os_get_block</code> †	Get a block from a memory pool.	148
<code>os_send_message</code> †	Send a message (call from task).	443 with task switch
<code>os_send_signal</code>	Send a signal to a task (call from tasks).	408 with task switch 316 with fast task switch 71 without task switch
<code>os_send_token</code> †	Set a semaphore (call from task).	343 with fast task switch 94 without task switch
<code>os_set_slice</code> †	Set the RTX-51 system clock time slice.	67
<code>os_wait</code>	Wait for an event.	68 for pending signal 160 for pending message

† These functions are available only in RTX-51 Full.

Additional debug and support functions in RTX-51 Full include the following:

Function	Description
<code>oi_reset_int_mask</code>	Disables interrupt sources external to RTX-51.
<code>oi_set_int_mask</code>	Enables interrupt sources external to RTX-51.
<code>os_check_mailbox</code>	Returns information about the state of a specific mailbox.
<code>os_check_mailboxes</code>	Returns information about the state of all mailboxes in the system.
<code>os_check_pool</code>	Returns information about the blocks in a memory pool.
<code>os_check_semaphore</code>	Returns information about the state of a specific semaphore.
<code>os_check_semaphores</code>	Returns information about the state of all semaphores in the system.
<code>os_check_task</code>	Returns information about a specific task.

Function	Description
<code>os_check_tasks</code>	Returns information about all tasks in the system.

## CAN Functions

The CAN functions are available only with RTX-51 Full. CAN controllers supported include the Philips 82C200 and 80C592 and the Intel 82526. More CAN controllers are in preparation.

CAN Function	Description
<code>can_bind_obj</code>	Bind an object to a task; task is started when object is received.
<code>can_def_obj</code>	Define communication objects.
<code>can_get_status</code>	Get CAN controller status.
<code>can_hw_init</code>	Initialize CAN controller hardware.
<code>can_read</code>	Directly read an object's data.
<code>can_receive</code>	Receive all unbound objects.
<code>can_request</code>	Send a remote frame for the specified object.
<code>can_send</code>	Send an object over the CAN bus.
<code>can_start</code>	Start CAN communications.
<code>can_stop</code>	Stop CAN communications.
<code>can_task_create</code>	Create the CAN communication task.
<code>can_unbind_obj</code>	Disconnect the binding between a task and an object.
<code>can_wait</code>	Wait for reception of a bound object.
<code>can_write</code>	Write new data to an object without sending it.

## TRAFFIC: RTX-51 Tiny Example Program

The TRAFFIC example is a pedestrian traffic light controller that shows the usage of multitasking RTX-51 Tiny Real-time operating system. During a user-defined time interval, the traffic light is operating. Outside this time interval, the yellow light flashes. If a pedestrian pushes the request button, the traffic light goes immediately into *walk* state. Otherwise, the traffic light works continuously.

### Traffic Light Controller Commands

The serial commands that TRAFFIC supports are listed in the following table. These commands are composed of ASCII text characters. All commands must be terminated with a carriage return.

Command	Serial Text	Description
Display	D	Display clock, start, and ending times.
Time	T <i>hh:mm:ss</i>	Set the current time in 24-hour format.
Start	S <i>hh:mm:ss</i>	Set the starting time in 24-hour format. The traffic light controller operates normally between the start and end times. Outside these times, the yellow light flashes.
End	E <i>hh:mm:ss</i>	Set the ending time in 24-hour format.



## Software

The TRAFFIC application is composed of three files that can be found in the \KEIL\C51\EXAMPLES\TRAFFIC folder.

TRAFFIC.C contains the traffic light controller program that is divided into the following tasks:

- **Task 0 init:** initializes the serial interface and starts all other tasks. Task 0 deletes itself since initialization is needed only once.
- **Task 1 command:** is the command processor for the traffic light controller. This task controls and processes serial commands received.
- **Task 2 clock:** controls the time clock.
- **Task 3 blinking:** flashes the yellow light when the clock time is outside the active time range.
- **Task 4 lights:** controls the traffic light phases while the clock time is in the active time range (between the start and end times).
- **Task 5 keyread:** reads the pedestrian push button and sends a signal to the task lights.
- **Task 6 get\_escape:** If an ESC character is encountered in the serial stream the command task gets a signal to terminate the display command.

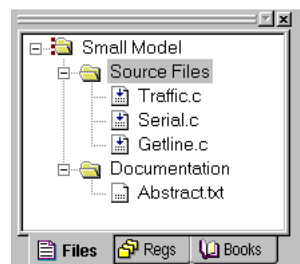
SERIAL.C implements an interrupt driven serial interface. This file contains the functions *putchar* and *getkey*. The high-level I/O functions *printf* and *getline* call these basic I/O routines. The traffic light application will also operate without using interrupt driven serial I/O, but will not perform as well.

GETLINE.C is the command line editor for characters received from the serial port. This source file is also used by the MEASURE application.

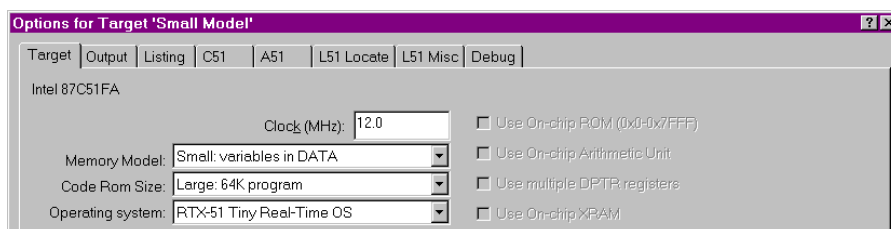
## TRAFFIC Project



Open the TRAFFIC.UV2 project file that is located in \KEIL\C51\EXAMPLES\TRAFFIC folder with  $\mu$ Vision2. The source files for the TRAFFIC project will be shown in the **Project Window – Files** page.



The RTX-51 Tiny Real-Time OS is selected under Options for Target.



Build the TRAFFIC program with **Project - Build** or the toolbar button.

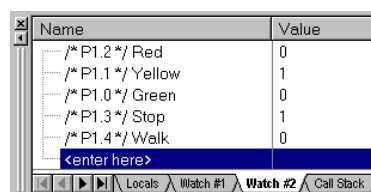


## Run the TRAFFIC Program

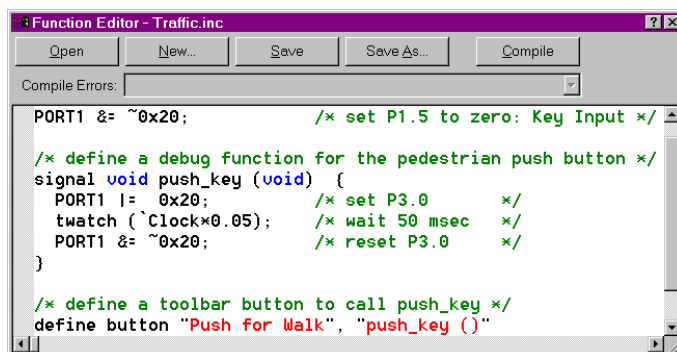
You can test TRAFFIC with the  $\mu$ Vision2 simulator.



The watch variables shown on the right allow you to view port status that drives the lights.



The **push\_key** signal function simulates the pedestrian push key that switches the light system to *walk* state. This function is called with the **Push for**



Use **Debug – Function Editor** to open TRAFFIC.INC. This file is specified

the **Push for Walk** toolbar button.

under **Options for Target – Debug – Initialization File** and defines the signal function **push\_key**, the port initialization and the toolbar button.

**Note:** the VTREG symbol *Clock* is literalized with a back quote (`), since there is a C function named *clock* in the **TRAFFIC.C** module. Refer to “Literal Symbols” on page 121 for more information.



Now run the TRAFFIC application. Enable **View – Periodic Window Update** to view the lights in the watch window during program execution.



The **Serial Window #1** displays the *printf* output and allows you to enter the traffic light controller commands described in the table above.

Set the clock time outside of the start/end time interval to flash the yellow light.

```

Serial #1
| with pedestrian self-service. Outside of this time range
| the yellow caution lamp is blinking.
+-----+-----+
| command | syntax | function |
+-----+-----+
| Display | D      | display times |
| Time    | T hh:mm:ss | set clock time |
| Start   | S hh:mm:ss | set start time |
| End     | E hh:mm:ss | set end time |
+-----+-----+

Command: d
Start Time: 07:30:00    End Time: 18:30:00
Clock Time: 12:02:44
  
```

## RTX Kernel Aware Debugging

A RTX application can be tested with the same methods and commands as standard 8051 applications. When you select an **Operating System** under **Options for Target – Target**,  $\mu$ Vision2 enables additional debugging features: a dialog lists the operating system status and with the **\_TaskRunning\_** debug function you may stop program execution when a specific task is active.

The following section exemplifies RTX debugging with the TRAFFIC example.



Stop program execution, reset the CPU and kill all breakpoints.



An RTX-51 application can be tested in the same way as standard applications. You may open source files, set break points and single step through the code. The TRAFFIC application starts with task 0 *init*.



```

C:\Keil\C51\EXAMPLES\TRAFFIC\TRAFFIC.C
/*****
/*      Task 0 'init': Initialize
*****/
void init (void) _task_ INIT {          /* program execution start
serial_init ();                        /* initialize the serial i
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 long
}

```



$\mu$ Vision2 is completely kernel aware. You may display the task status with the menu command **Peripherals – RTX Tiny Tasklist**.

TID	Task Name	State	Wait for Event	Sig	Timer	Stack
0	init	Deleted		0	0x31	0x7F
1	command	Running		0	0x31	0x7F
2	clock	Waiting	Timeout	0	0x25	0xF7
3	blinking	Deleted		0	0x31	0xF9
4	lights	Waiting	Signal & TimeOut	0	0x34	0xF9
5	keyread	Waiting	Timeout	0	0x01	0xFB
6	get_escape	Ready		0	0x31	0xFD

The dialog **RTX51 Tiny Tasklist** gives you the following information:

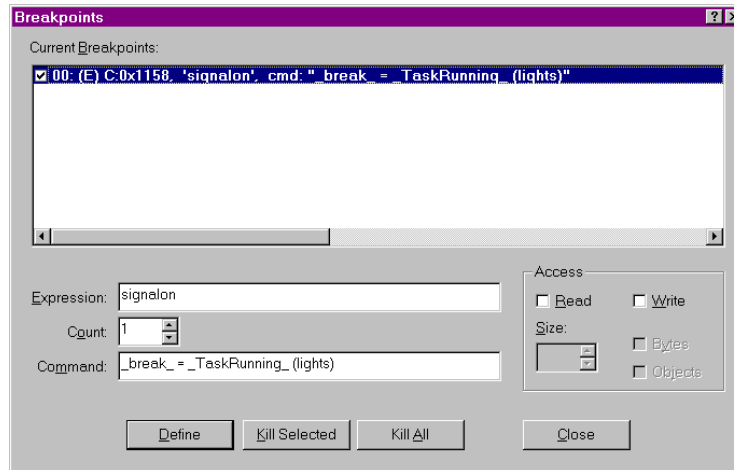
Heading	Description
<b>TID</b>	<i>task_id</i> used in the definition of the task function.
<b>Task Name</b>	name of the task function.
<b>State</b>	task state of the function; explained in detail in the next table.
<b>Wait for Event</b>	event the task is waiting for; the following events are possible (also in combination): <b>Timeout:</b> the task <b>Timer</b> is set to the duration is specified with the <i>os_wait</i> function call. After the <b>Timer</b> decrements to zero, the task goes into <b>Ready</b> state. <b>Interval:</b> the time interval specified with <i>os_wait</i> is added to the task <b>Timer</b> value. After the <b>Timer</b> decrements to zero, the task goes into <b>Ready</b> state. <b>Signal:</b> the <i>os_wait</i> function was called with <b>K_SIG</b> and the task waits for <b>Sig = 1</b> .
<b>Sig</b>	status of the <b>Signal</b> bit that is assigned to this task.
<b>Timer</b>	value of the <b>Timer</b> that is assigned to this task. The <b>Timer</b> value decrements with every RTX system timer tick. If the <b>Timer</b> becomes zero and the task is waiting for <b>Timeout</b> or <b>Interval</b> the task goes into <b>Ready</b> state.
<b>Stack</b>	value of the stack pointer (SP) that is used when this task is <b>Running</b> .

RTX-51 Tiny contains an efficient stack management that is explained in the "RTX51 Tiny" User's Guide, Chapter 5: RTX51 Tiny, Stack Management.

This manual provides detailed information about the **Stack** value.

State	Task State of a RTX51 Task Function
<b>Deleted</b>	Tasks that are not started are in the <b>Deleted</b> state.
<b>Ready</b>	Tasks that are waiting for execution are in the <b>Ready</b> state. After the currently <b>Running</b> task has finished processing, RTX starts the next task that is in the <b>Ready</b> state.
<b>Running</b>	The task currently being executed is in the <b>Running</b> state. Only one task is in the <b>Running</b> state at a time.
<b>Timeout</b>	Tasks that were interrupted by a round-robin timeout are in the <b>Timeout</b> state. This state is equivalent to <b>Ready</b> ; however, a round-robin task switch is marked due to internal operating procedures.
<b>Waiting</b>	Tasks that are waiting for an event are in the <b>Waiting</b> state. If the event occurs which the task is waiting for, this task then enters the <b>Ready</b> state.

The **Debug – Breakpoints...** dialog allows you to define breakpoints that stop the program execution only when the task specified in the ***\_TaskRunning\_*** debug function argument is **Running**. Refer to “Predefined Functions” on page 134 for a detailed description of the ***\_TaskRunning\_*** debug function.



The breakpoint at the function ***signalon*** stops execution only if ***lights*** is the current **Running** task.