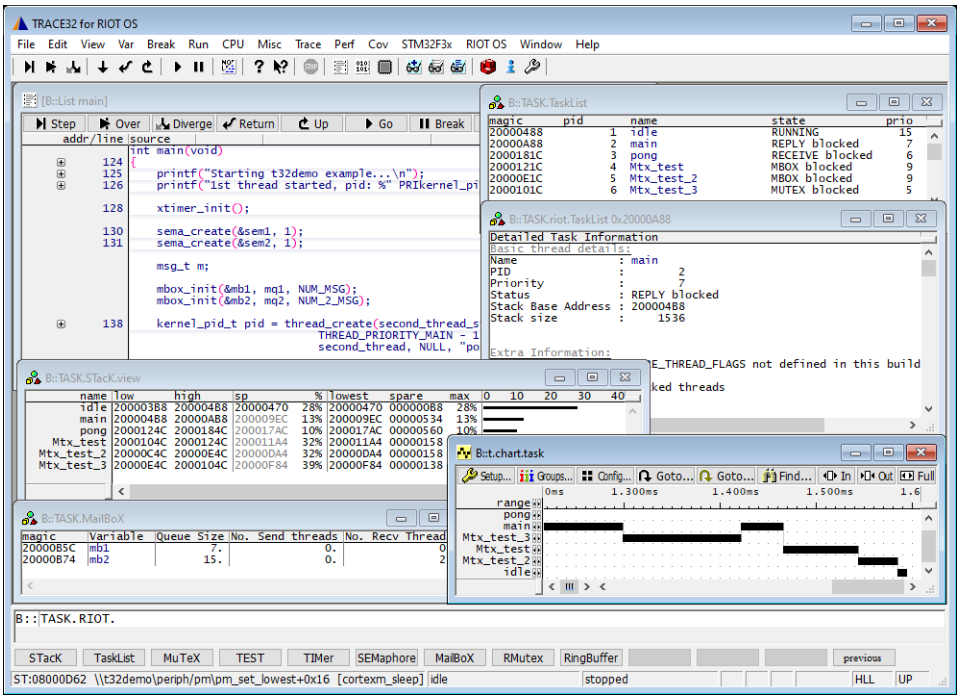
# OS Awareness Manual RIOT

**Version 18-Nov-2020**

# History

| | |
|---|---|
| 28-Aug-18 | The title of the manual was changed from “RTOS Debugger for *<x>*” to “OS Awareness Manual *<x>*”. |

# Overview




The OS Awareness for RIOT OS (RIOT) contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistical evaluations.

# Terminology

The terms *task* and *thread* are used interchangeably throughout this manual. RIOT does not support a true threaded concept such as POSIX threads, but each task is very lightweight and is often referred to as a thread by the RIOT documentation.

# Brief Overview of Documents for New Users

**Architecture-independent information:**

- **"Debugger Basics - Training"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.

**Architecture-specific information:**

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:

  - Choose **Help** menu > **Processor Architecture Manual**.

- **"OS Awareness Manuals"** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

# Supported Versions

Currently RIOT is supported for the following versions:

- 32 bit ARM cores, including Cortex-M.

# Restrictions

RIOT is supplied in full source code for users to modify to fit their requirements. The awareness has been built and tested against an unmodified version of RIOT. Please see **"Hooks & Internals in RIOT"**, page 7 for more information.

Currently, the awareness supports a maximum of 32 mailboxes, 32 mutexes, 32 ring buffers, 32 recursive mutexes, 32 semaphores, and 32 timers. If your system requires support for more than this, please contact your local Lauterbach representative.

Some of the more complex analysis features require data trace.

- This is an option on ARM9 and ARM11 systems and will be listed as ETM (Embedded Trace Macrocell) or off-chip trace. Some devices may only offer on-chip trace or ETB (Embedded Trace Buffer) and this is seldom sufficient for these types of analysis. More information about this subject can be found in **"ARM-ETM Trace"** (trace_arm_etm.pdf).
- Many Cortex-M based systems have an option for off-chip trace, although not all provide enough information to perform complex analyses. More information about this can be found in **"Cortex-M Trace Training"** (training_cortexm_etm.pdf).

# Configuration

The **TASK.CONFIG** command loads an extension definition file called “riot.t32” (directory “~~/demo/*<arch>*/kernel/riot”). It contains all necessary extensions.

The OS Awareness for RIOT will try to automatically locate all of the required internal information by itself and, as such, no manual configuration is necessary or possible. In order to achieve this, all symbol tables must be loaded and accessible at any time the OS Awareness is used.

If you want to display the OS objects “On The Fly” while the target is running, you need to have access to memory while the target is running. Enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent).

For system resource display and trace functionality, you can do an automatic configuration of the OS Awareness. For this purpose it is necessary that all system internal symbols are loaded and accessible at any time, the OS Awareness is used. To configure the awareness, use the command:

| Format: | **TASK.CONFIG riot** |
|---|---|

See also “**Hooks & Internals**” for details on the used symbols.

# Quick Configuration Guide

Example scripts are provided in ~~/demo/*<arch>*/kernel/riot/boards/*<board>*. It is recommended to take one of these as a starting point and modify it to suit your target and setup.

If you already have a setup/configuration script which configures the target and loads the application code and/or symbols, you can add the following lines to your script after the symbols have been loaded:

```
TASK.CONFIG ~~/demo/<arch>/kernel/riot/riot.t32
MENU.ReProgram ~~/demo/<arch>/kernel/riot/riot.men
```

These lines will automatically configure the awareness and add a custom menu that provides access to many of the features.

**To get a quick access to the features of the OS Awareness for RIOT with your application, follow this roadmap:**

1. Start the TRACE32 Debugger.
2. Load your application as normal.
3. Execute the command **TASK.CONFIG** ~~/demo/<*arch*>/kernel/riot/riot.t32
   (See "**Configuration**").
4. Execute the command **MENU.ReProgram** ~~/demo/<*arch*>/kernel/riot/riot.men
   (See "**RIOT Specific Menu**").
5. Start your application.

Now you can access the RIOT extensions through the menu.

In case of any problems, please carefully read the previous **Configuration** chapter.

## Hooks & Internals in RIOT

No hooks are used in the kernel.

To retrieve information on kernel objects, the OS Awareness uses the global RIOT variables and structures. Be sure that your application is compiled and linked with debugging symbols switched on.

Many of the features of RIOT are defined at build time. If these features were not included in the build, TRACE32 will not display them.

# Features

The OS Awareness for RIOT supports the following features.

## Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following RIOT components can be displayed:

| | |
|---|---|
| **TASK.MailBoX** | Mailboxes |
| **TASK.MuTeX** | Mutexes |
| **TASK.RingBuffer** | Ring buffers |
| **TASK.RMutex** | Recursive mutexes |
| **TASK.SEMaphore** | Semaphores |
| **TASK.TaskList** | Tasks |
| **TASK.TIMer** | Timers |

For a description of the commands, refer to **"RIOT Commands"**, page 16.

If your hardware allows accessing the memory while the target is running, these resources can be displayed "On The Fly", i.e. while the application is running, without any intrusion to the application. Be aware that a screen update may occur midway through a scheduling operation which may cause display inconsistencies.

Without this capability, the information will only be displayed if the target application is stopped.

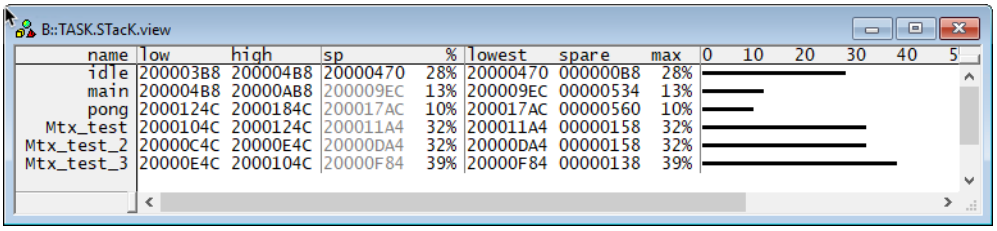
## Task Stack Coverage

For stack usage coverage of the tasks, you can use the **TASK.STacK** command. Without any parameter, this command will open a window displaying with all active tasks. If you specify only a task magic number as parameter, the stack area of this task will be automatically calculated.

To use the calculation of the maximum stack usage, flag memory must be mapped to the task stack areas when working with the emulation memory. When working with the target memory, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is zero).

To add/remove one task to/from the task stack coverage, you can either call the **TASK.STacK.ADD** or **TASK.STacK.ReMove** commands with the task magic number as the parameter, or omit the parameter and select the task from the **TASK.STacK.*** window.

It is recommended to display only the tasks you are interested in because the evaluation of the used stack space is very time consuming and slows down the debugger display.




RIOT pre-fills each address in a thread's stack with a value that is equal to the address the value is being written to. To get a correct indication of the amount of stack used, the command **TASK.STacK.RESet** must be used. Failure to do this will show all stacks at 100% usage.

## Task-Related Breakpoints

Any breakpoint set in the debugger can be restricted to fire only if a specific task hits that breakpoint. This is especially useful when debugging code which is shared between several tasks. To set a task-related breakpoint, use the command:

| | |
|---|---|
| **Break.Set** *<address>*\|*<range>* [*/<option>*] **/TASK** *<task>* | Set task-related breakpoint. |

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- For a general description of the **Break.Set** command, please see its documentation.

By default, the task-related breakpoint will be implemented by a conditional breakpoint inside the debugger. This means that the target will *always* halt at that breakpoint, but the debugger immediately resumes execution if the current running task is not equal to the specified task.

| | |
|---|---|
| **NOTE:** | Task-related breakpoints impact the real-time behavior of the application. |

On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering the tasks.

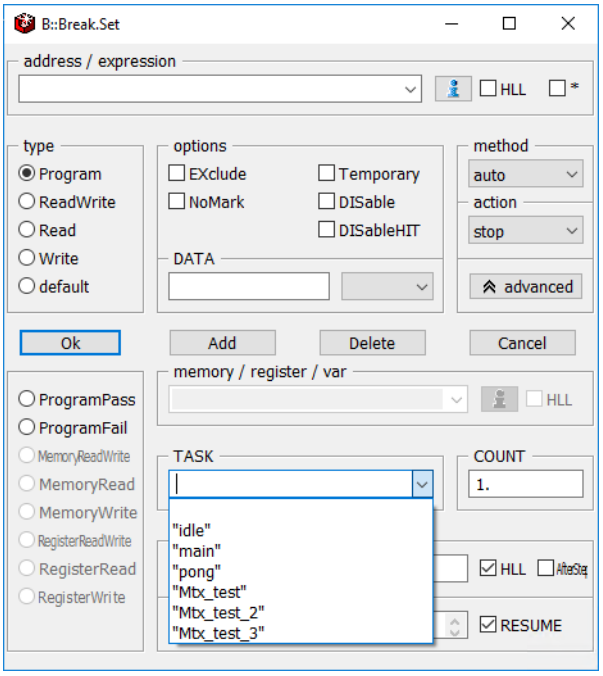
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

| | |
|---|---|
| **Break.CONFIG.UseContextID ON** | Enables the comparison to the whole Context ID register. |
| **Break.CONFIG.MatchASID ON** | Enables the comparison to the ASID part only. |
| **TASK.List.tasks** | If **TASK.List.tasks** provides a trace ID (**traceid** column), the debugger will use this ID for comparison. Without the trace ID, it uses the magic number (**magic** column) for comparison. |

When single stepping, the debugger halts at the next instruction, regardless of which task hits this breakpoint. When debugging shared code, stepping over an OS function may cause a task switch and coming back to the same place - but with a different task. If you want to restrict debugging to the current task, you can set up the debugger with **SETUP.StepWithinTask ON** to use task-related breakpoints for single stepping. In this case, single stepping will always stay within the current task. Other tasks using the same code will not be halted on these breakpoints.

If you want to halt program execution as soon as a specific task is scheduled to run by the OS, you can use the **Break.SetTask** command.

The **Break.Set** window adds a drop-down list of tasks to aid setting task-aware breakpoints from the user interface. An example can be seen below.




## Task Context Display

You can switch the whole viewing context to a task that is currently not being executed. This means that all register and stack-related information displayed, e.g. in **Register**, **Data.List**, **Frame** etc. windows, will refer to this task. Be aware that this is only for displaying information. When you continue debugging the application (**Step** or **Go**), the debugger will switch back to the current context.

To display a specific task context, use the command:

**Frame.TASK** [*<task>*] Display task context.

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

To display the call stack of a specific task, use the following command:

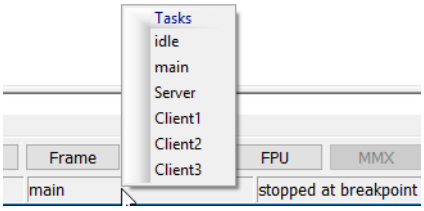
**Frame /Task** *<task>* Display call stack of a task.

If you'd like to see the application code where the task was preempted, then take these steps:

1. Open the **Frame /Caller /Task** *<task>* window.
2. Double-click the line showing the OS service call.

The **TASK.TaskList** *<thread_magic>* window contains a button ("context") to execute this command with the displayed task, and to switch back to the current context ("current").

The current task is also shown on the TRACE32 state line. Right-clicking this will open a pop-up menu listing all tasks. Selecting one from here will also change the context to the selected task.




## Dynamic Task Performance Measurement

The debugger can execute a dynamic performance measurement by evaluating the current running task in changing time intervals. Start the measurement with the commands **PERF.Mode TASK** and **PERF.Arm**, and view the contents with **PERF.ListTASK**. The evaluation is done by reading the 'magic' location (= current running task) in memory. This memory read may be non-intrusive or intrusive, depending on the **PERF.METHOD** used.

If **PERF** collects the PC for function profiling of processes in MMU-based operating systems (**SYStem.Option MMUSPACES ON**), then you need to set **PERF.MMUSPACES**, too.

For a general description of the **PERF** command group, refer to **"General Commands Reference Guide P"** (general_ref_p.pdf).

## Task Runtime Statistics

| | |
|---|---|
| **NOTE:** | This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **Logger**). For details, refer to **"OS-aware Tracing"** (glossary.pdf). |

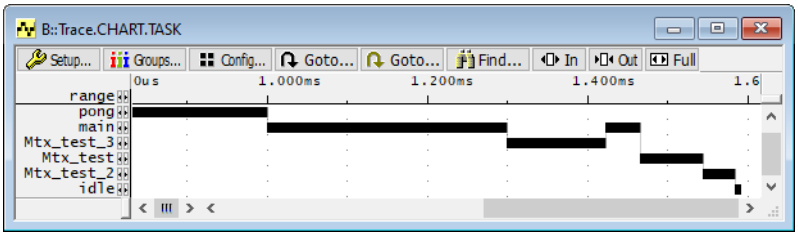
Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

To evaluate the contents of the trace buffer, use these commands:

| | |
|---|---|
| **Trace.List List.TASK DEFault** | Display trace buffer and task switches |
| **Trace.STATistic.TASK** | Display task runtime statistic evaluation |
| **Trace.Chart.TASK** | Display task runtime timechart |
| **Trace.PROfileSTATistic.TASK** | Display task runtime within fixed time intervals statistically |
| **Trace.PROfileChart.TASK** | Display task runtime within fixed time intervals as colored graph |
| **Trace.FindAll Address TASK.CONFIG(magic)** | Display all data access records to the "magic" location |
| **Trace.FindAll CYcle owner OR CYcle context** | Display all context ID records |

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".

All kernel activities are added to the calling task.



| NOTE: | This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **Logger**). For details, refer to **"OS-aware Tracing"** (glossary.pdf). |
|---|---|

All function-related statistic and time chart evaluations can be used with task-specific information. The function timings will be calculated dependent on the task that called this function. To do this, in addition to the function entries and exits, the task switches must be recorded.

To do a selective recording on task-related function runtimes based on the data accesses, use the following command:

```
; Enable flow trace and accesses to the magic location
Break.Set TASK.CONFIG(magic) /TraceData
```

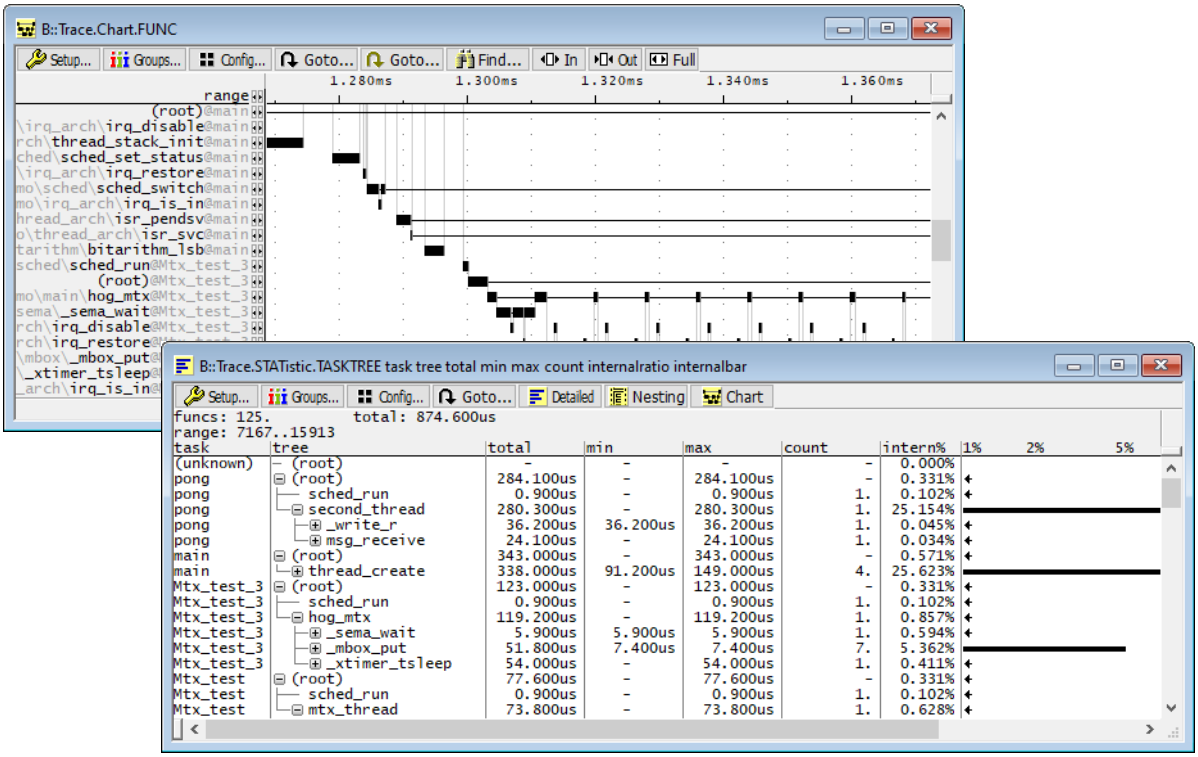
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

To evaluate the contents of the trace buffer, use these commands:

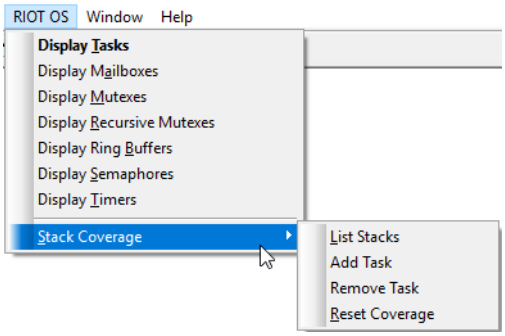
| | |
|---|---|
| **Trace.ListNesting** | Display function nesting |
| **Trace.STATistic.Func** | Display function runtime statistic |
| **Trace.STATistic.TREE** | Display functions as call tree |
| **Trace.STATistic.sYmbol /SplitTASK** | Display flat runtime analysis |
| **Trace.Chart.Func** | Display function timechart |
| **Trace.Chart.sYmbol /SplitTASK** | Display flat runtime timechart |

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".



The menu file “riot.men” contains a set of additional menus with RIOT specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **RIOT OS**, which looks like the image below.




- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the RIOT specific stack coverage and provides an easy way to add or remove tasks from the stack coverage window.

In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

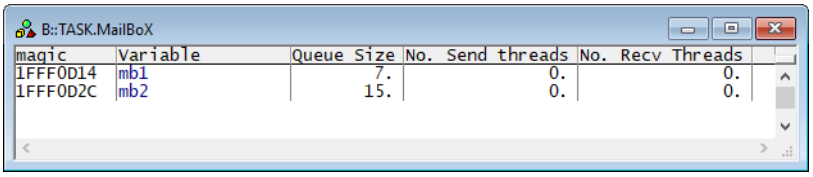
- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with the default display.
- The **Perf** menu contains additional submenus for task runtime statistics and statistics on task states.

## TASK.MailBoX — Display mailboxes

| Format: | **TASK.MailBoX** [*<mailbox_magic>*] |
|---|---|

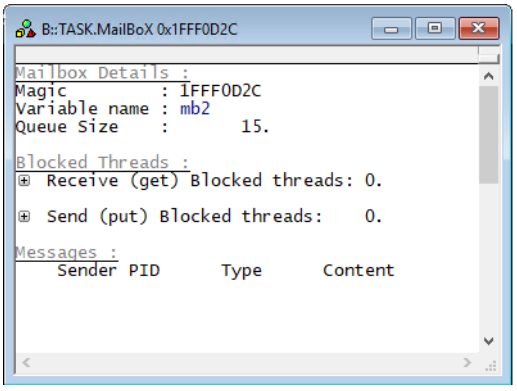
Displays a list of all system mailboxes or detailed information about one specific mailbox.

Without any arguments, a table with all created mailboxes will be shown.

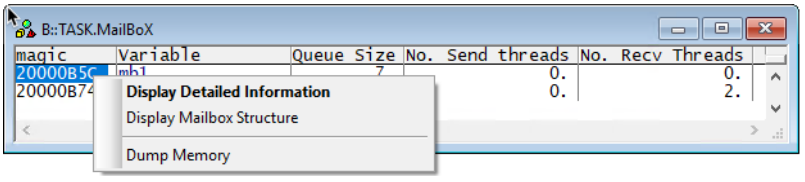



| | |
|---|---|
| *<mailbox_magic>* | Specify a mailbox magic number to display detailed information on that mailbox.<br>“magic” is a unique ID, used by the OS Awareness to identify a specific mailbox (address of the mbox_t structure). |

The field “magic” is mouse sensitive, double-clicking it opens an appropriate window showing more detail.



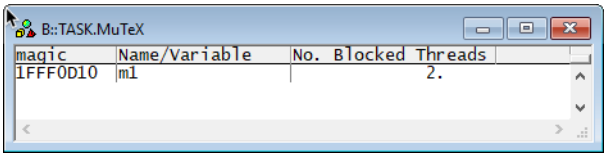

Right-clicking it will show a local menu.



| Format: | **TASK.MuTeX** [*<mutex_magic>*] |
|---|---|

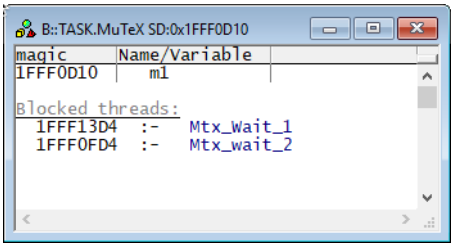
Displays a list of all mutexes or detailed information about one specific mutex.

Without any arguments, a table with all mutexes will be shown.

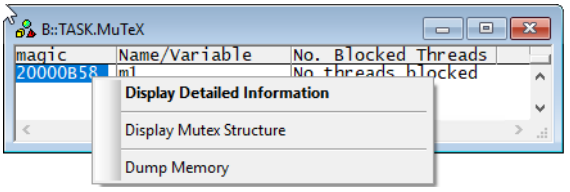



| | |
|---|---|
| *<mutex_magic>* | Specify a mutex magic number to display detailed information on that mutex.<br>“magic” is a unique ID, used by the OS Awareness to identify a specific mutex (address of the mutex_t structure). |

The field “magic” is mouse sensitive, double-clicking it opens an appropriate window.

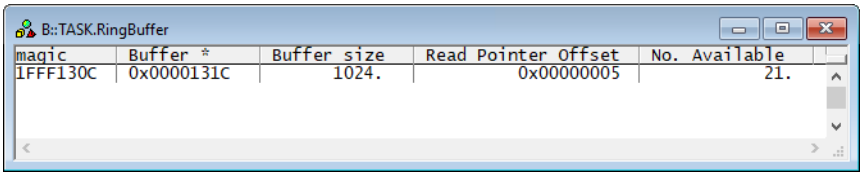



Right-clicking it will show a local menu.

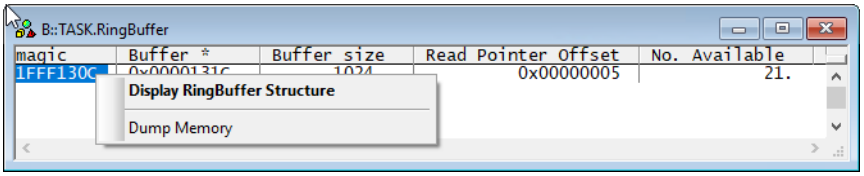


Format: **TASK.RingBuffer**

Shows a list of all ring buffers in the system.



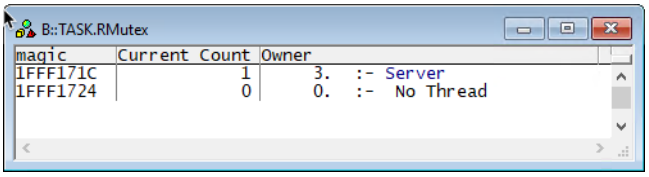

Right-clicking it will show a local menu.



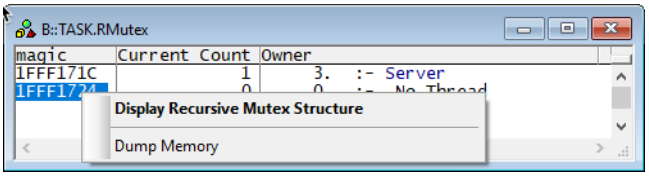

## TASK.RMutex Display recursive mutexes

Format: **TASK.RMutex**

Displays a list of all recursive mutexes.



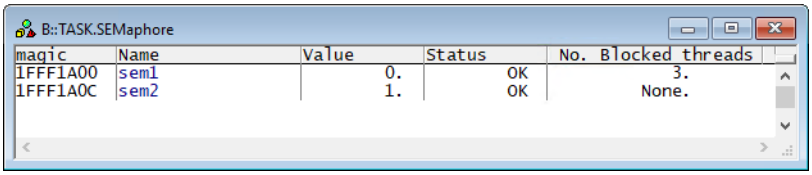

Right-clicking the magic of a recursive mutex will open a context specific menu.



| Format: | **TASK.SEMaphore** [*<semaphore_magic>*] |
|---|---|

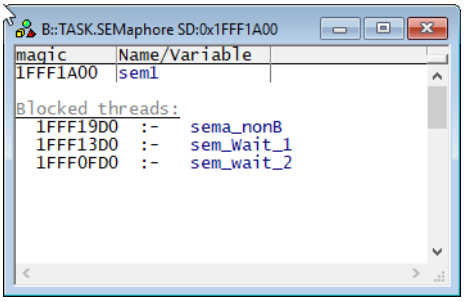
Displays a list of all semaphores or detailed information about one specific semaphore.

Without any arguments, a table with all semaphores will be shown.

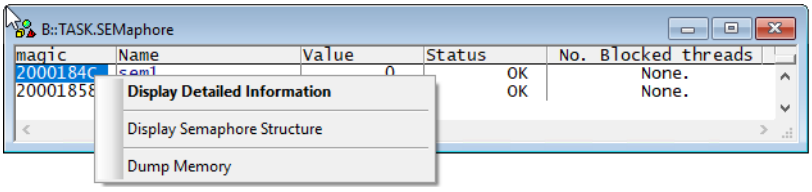



| | |
|---|---|
| *<semaphore_magic>* | Specify a semaphore magic number to display detailed information on that semaphore.<br>“magic” is a unique ID, used by the OS Awareness to identify a specific task (address of a sema_t structure). |

The field “magic” is mouse sensitive, double-clicking it opens an appropriate window showing more information about a single semaphore.

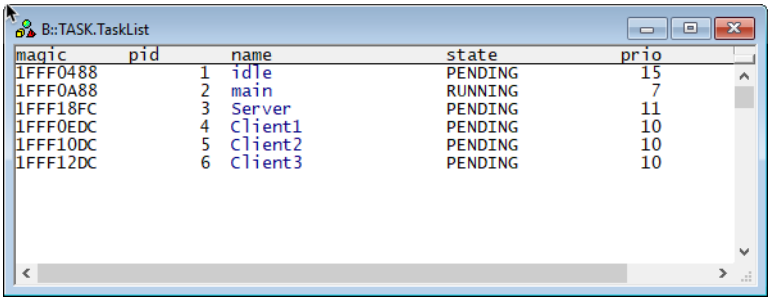



Right-clicking it will show a local menu.



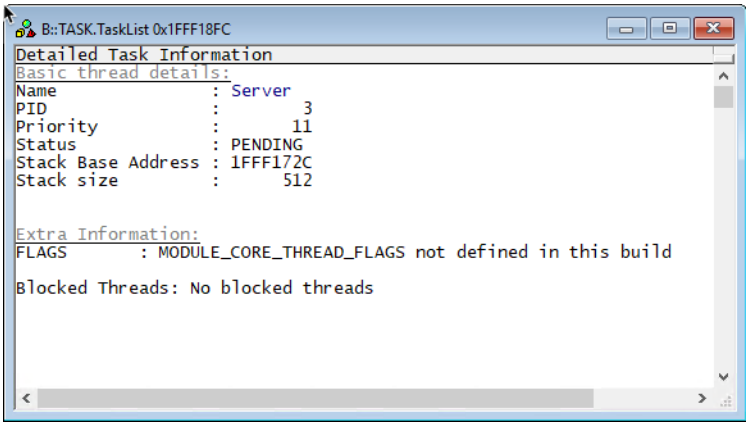
| Format: | **TASK.TaskList** [*<thread_magic>*] |
|---|---|

Displays a list of threads or detailed information about one specific thread.

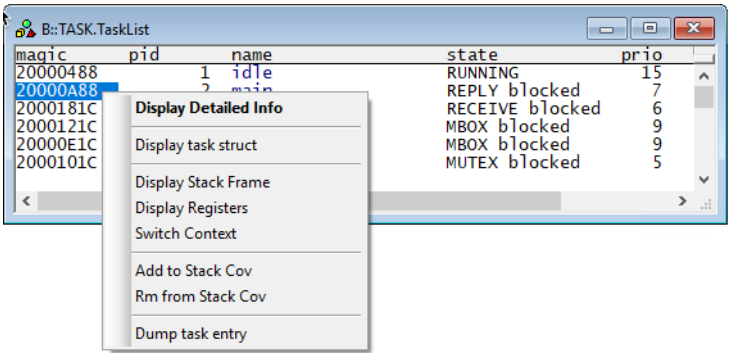



| | |
|---|---|
| *<thread_magic>* | Specify a thread magic number to display detailed information on that thread.<br>"magic" is a unique ID, used by the OS Awareness to identify a specific timer (address of the thread_t structure). |

The "magic" field is mouse sensitive, double-clicking it opens an appropriate window.

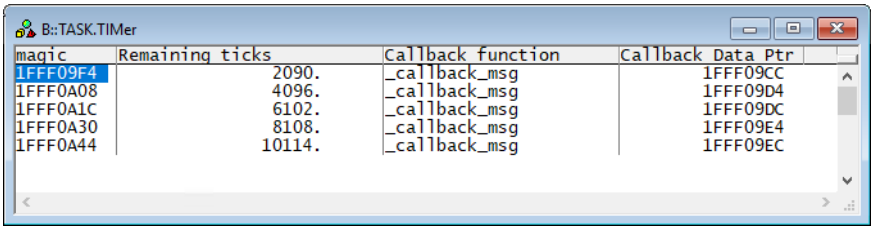



Right-clicking it will show a local menu.

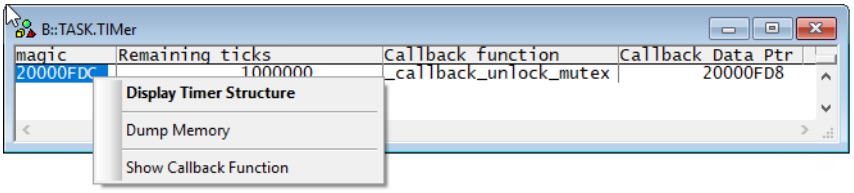


Format: **TASK.TIMer**

Displays a list of all unexpired timers.




Right-clicking the magic of a timer will open a context specific menu.



There are special definitions for RIOT specific PRACTICE functions.

## TASK.CONFIG() OS Awareness configuration information

Syntax: **TASK.CONFIG(magic | magicsize | tcb)**

**Parameter and Description**:

| | |
|---|---|
| **magic** | **Parameter Type**: String (***without*** quotation marks).<br>Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number). |
| **magicsize** | **Parameter Type**: String (***without*** quotation marks).<br>Returns the size of the task magic number (1, 2 or 4). |
| **tcb** | **Parameter Type**: String (***without*** quotation marks).<br>Returns the name of the TCB structure. |

**Return Value Type**: Hex value.

**Why does a thread not show as semaphore blocked?**

RIOT implements semaphores as a layer on top of a mutex. When a thread is blocked on a semaphore, in reality it is blocked on the underlying mutex. The status bit in the TCB shows mutex blocked. This is the correct behavior of RIOT OS.

**Why is a thread listed as mutex blocked when it isn't?**

When a thread acquires a mutex, RIOT sets the status bit in the TCB as mutex blocked, even though the thread is not blocked.

If a thread makes a non-blocking attempt to lock a mutex (for example: `mutex_trylock()`), it is temporarily marked as mutex blocked by RIOT until the next kernel call from that thread.

Both of these are normal behavior.