

# OS Awareness Manual REALOS

MANUAL

# OS Awareness Manual REALOS

---

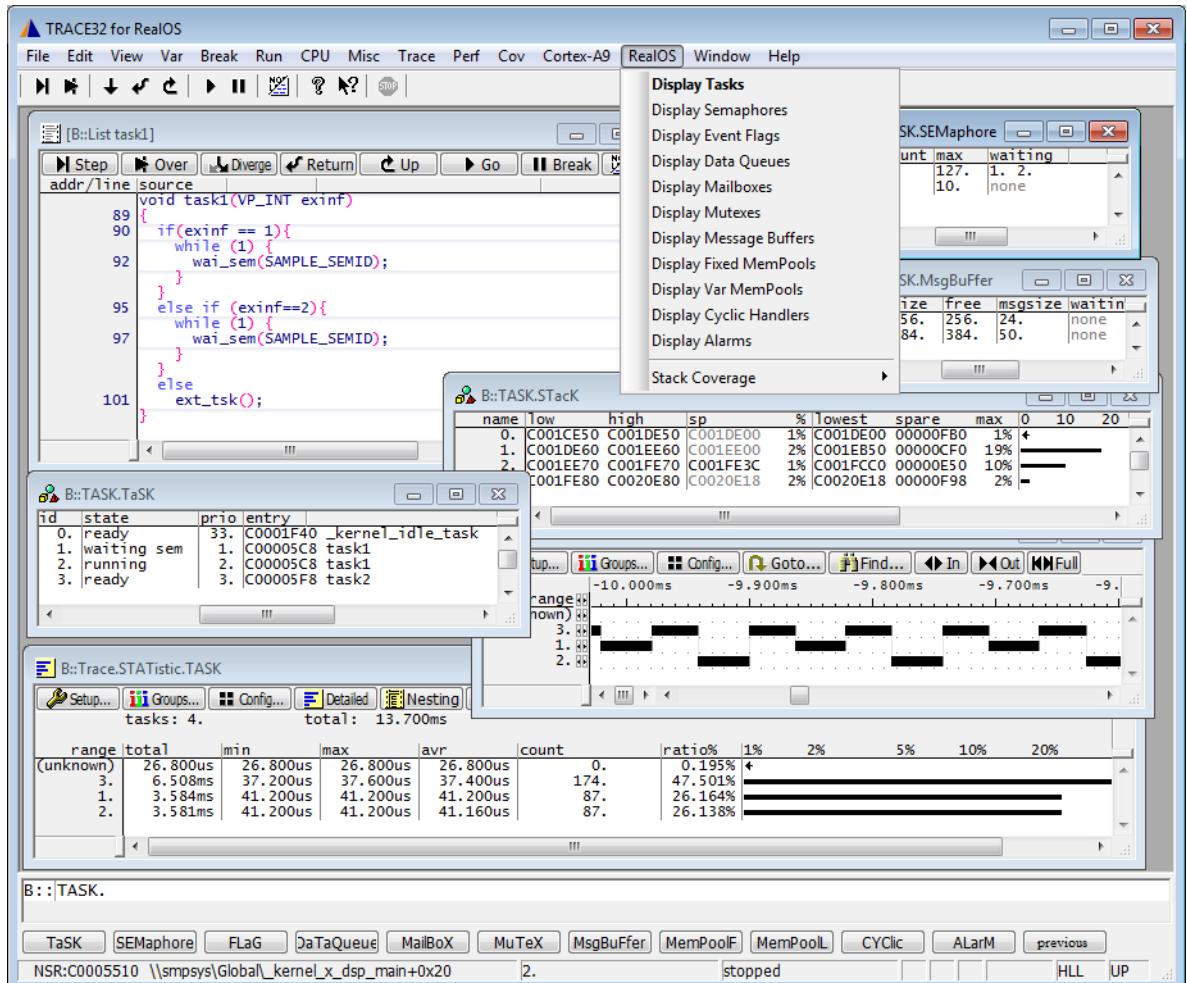
[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

<a href="#">TRACE32 Documents</a>	.....	
<a href="#">OS Awareness Manuals</a>	.....	
<a href="#">OS Awareness Manual REALOS</a>	.....	1
<a href="#">    Overview</a>	.....	3
Brief Overview of Documents for New Users		4
Supported Versions		4
<a href="#">    Configuration</a>	.....	5
Quick Configuration Guide		6
Hooks & Internals in REALOS		6
<a href="#">    Features</a>	.....	7
Display of Kernel Resources		7
Task-Related Breakpoints		8
Task Stack Coverage		9
Task Context Display		9
Dynamic Task Performance Measurement		10
Task Runtime Statistics		10
Function Runtime Statistics		12
REALOS specific Menu		13
<a href="#">    REALOS Commands</a>	.....	14
TASK.ALarM	Display alarm handlers	14
TASK.CYClic	Display cyclic handlers	14
TASK.DaTaQueue	Display data queues	15
TASK.FLaG	Display event flags	15
TASK.MailBoX	Display mailboxes	16
TASK.MemPoolF	Display fixed memory pools	16
TASK.MemPoolL	Display variable memory pools	17
TASK.MsgBuFfer	Display message buffers	17
TASK.MuTeX	Display mutexes	18
TASK.SEMaphore	Display semaphores	19
TASK.TaSK	Display tasks	19
<a href="#">    REALOS PRACTICE Functions</a>	.....	20
TASK.CONFIG()	OS Awareness configuration information	20

## Overview



The OS Awareness for REALOS contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

## Architecture-independent information:

- [“Training Basic Debugging”](#) (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 REALOS is supported for the following versions:

- REALOS v1 on ARM architecture.

# Configuration

---

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

Automatic configuration tries to locate the REALOS internals automatically. For this purpose 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. In case of ICD, you have to 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. Each of the **TASK.CONFIG** arguments can be substituted by '0', which means that this argument will be searched and configured automatically. For a fully automatic configuration omit all arguments:

Format:	<b>TASK.CONFIG realos</b>
---------	---------------------------

See also “[Hooks & Internals](#)” for details on the used symbols.

To get a quick access to the features of the OS Awareness for REALOS with your application, follow the following roadmap:

1. Start the TRACE32 Debugger.
2. Load your application as normal.
3. Execute the command  
TASK.CONFIG ~~/demo/<arch>/kernel/uc3std/realos.t32  
(See "[Configuration](#)").
4. Execute the command  
MENU.ReProgram ~~/demo/<arch>/kernel/uc3std/realos.men  
(See "[RTOS Specific Menu](#)").
5. Start your application.

Now you can access the REALOS extensions through the menu.

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

## Hooks & Internals in REALOS

---

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols and structure definitions. Ensure that access to those structures is possible every time when features of the OS Awareness are used. The REALOS kernel must be compiled with debug information.

The OS Awareness for REALOS supports the following features.

## Display of Kernel Resources

---

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

<b>TASK.TaSK</b>	Tasks
<b>TASK.SEMaphore</b>	Semaphores
<b>TASK.FLaG</b>	Event flags
<b>TASK.DaTaQueue</b>	Data queues
<b>TASK.MailBoX</b>	Mailboxes
<b>TASK.MsgBuFfer</b>	Message buffers
<b>TASK.MuTeX</b>	Mutexes
<b>TASK.MemPoolF</b>	Fixed sized memory pools
<b>TASK.MemPoolL</b>	Variable sized memory pools
<b>TASK.CYClic</b>	Cyclic handlers
<b>TASK.ALarM</b>	Alarm handlers

For a description of the commands, refer to chapter “[REALOS Commands](#)”.

If your hardware allows memory access 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.

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

# 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:

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

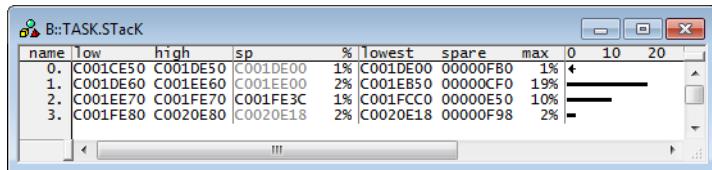
# Task Stack Coverage

For stack usage coverage of 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, 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.



## 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.

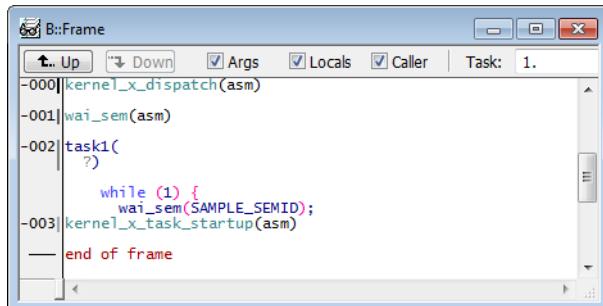
- Use a magic number, task ID, or task name for <task>. For information about the parameters, see **“What to know about the Task Parameters”** (general\_ref\_t.pdf).
- To switch back to the current context, omit all parameters.

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 screenshot shows the B::Frame window with the title 'B::Frame'. The assembly code for task1 is displayed in the main pane:

```
-000| kernel_x_dispatch(asn)
-001| wai_sem(asn)
-002| task1(
-003|     ?
-004|     while (1) {
-005|         wai_sem(SAMPLE_SEMID);
-006|     kernel_x_task_startup(asn)
-007| }
-008| end of frame
```

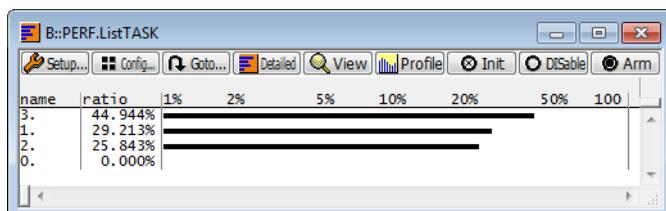
## Dynamic Task Performance Measurement

---

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:

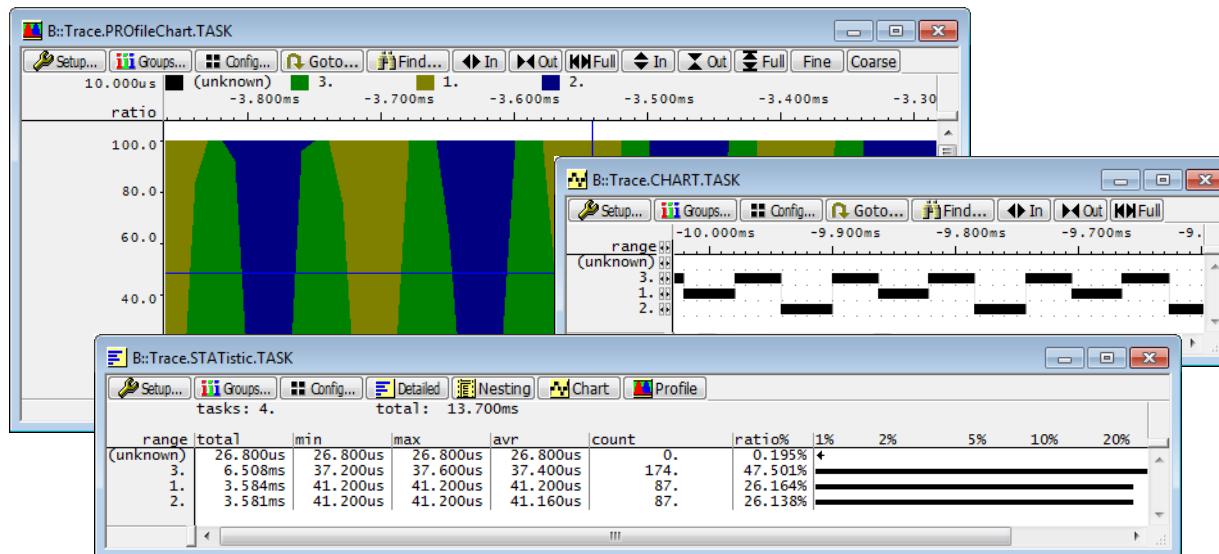
This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based 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:

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



**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:

<a href="#">Trace.ListNesting</a>	Display function nesting
<a href="#">Trace.STATistic.Func</a>	Display function runtime statistic
<a href="#">Trace.STATistic.TREE</a>	Display functions as call tree
<a href="#">Trace.STATistic.sYmbol /SplitTASK</a>	Display flat runtime analysis
<a href="#">Trace.Chart.Func</a>	Display function timechart
<a href="#">Trace.Chart.sYmbol /SplitTASK</a>	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 “realos.men” contains a menu with REALOS specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **RealOS**.

- The **Display** menu items launch the kernel resource display windows.

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.

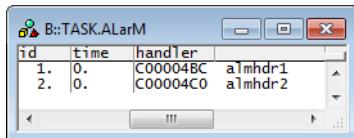
## TASK.ALarM

Display alarm handlers

Format: **TASK.ALarM**

Displays the table of installed alarm handlers or detailed information about one specific alarm handler.

Without any arguments, a table with all installed alarm handlers will be shown.  
Specify a alarm handler ID to display detailed information on that alarm handler.



<b>id</b>	<b>time</b>	<b>handler</b>
1.	0.	C000048C a1mhdr1
2.	0.	C00004C0 a1mhdr2

The fields “id” and “handler” are mouse sensitive. Double-clicking on them open appropriate windows. Right clicking on them will show local menu.

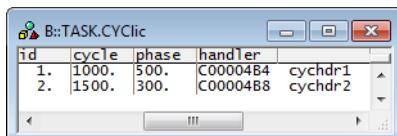
## TASK.CYClic

Display cyclic handlers

Format: **TASK.CYClic**

Displays the table of installed cyclic handlers or detailed information about one specific cyclic handler.

Without any arguments, a table with all installed cyclic handlers will be shown.  
Specify a cyclic handler ID to display detailed information on that cyclic handler.



<b>id</b>	<b>cycle</b>	<b>phase</b>	<b>handler</b>
1.	1000.	500.	C00004B4 cycchr1
2.	1500.	300.	C00004B8 cycchr2

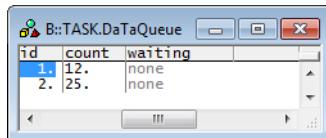
The fields “id” and “handler” are mouse sensitive. Double-clicking on them open appropriate windows. Right clicking on them will show local menu.

Format: **TASK.DaTaQueue [<queue>]**

Displays the data queue table of REALOS or detailed information about one specific data queue.

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

Specify a data queue ID to display detailed information on that data queue.



B::TASK.DaTaQueue		
id	count	waiting
1.	12.	none
2.	25.	none

The “waiting” column shows the task IDs waiting.

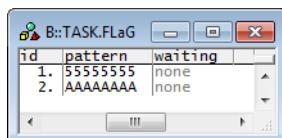
The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.FLaG [<flag>]**

Displays the event flag table of REALOS or detailed information about one specific event flag.

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

Specify a flag ID to display detailed information on that flag.



B::TASK.FLaG		
id	pattern	waiting
1.	55555555	none
2.	AAAAAAA	none

The “waiting” column shows the task IDs waiting.

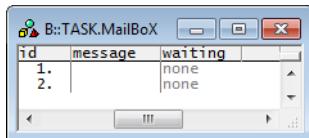
The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.MailBoX [<mailbox>]**

Displays the mailbox table of REALOS or detailed information about one specific mailbox.

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

Specify a mailbox ID to display detailed information on that mailbox.



B::TASK.MailBoX		
id	message	waiting
1.		none
2.		none

The “waiting” column shows the task IDs waiting.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

## TASK.MemPoolF

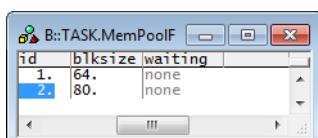
Display fixed memory pools

Format: **TASK.MemPoolF [<mempool>]**

Displays the fixed size memory pool table of REALOS or detailed information about one specific memory pool.

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

Specify a pool ID to display detailed information on that memory pool.



B::TASK.MemPoolF		
id	blksize	waiting
1.	64.	none
2.	80.	none

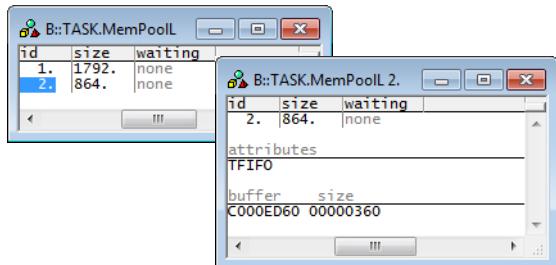
The “waiting” column shows the task IDs waiting.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.MemPoolL [<mempool>]**

Displays the variable size memory pool table of REALOS or detailed information about one specific memory pool.

Without any arguments, a table with all created memory pools will be shown.  
Specify a pool ID to display detailed information on that memory pool.



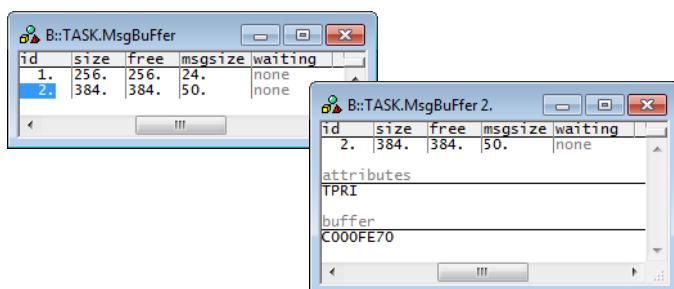
The “waiting” column shows the task IDs waiting.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.MsgBuFfer [<msgbuffer>]**

Displays the message buffer table of REALOS or detailed information about one specific message buffer.

Without any arguments, a table with all created message buffers will be shown.  
Specify a message buffer ID to display detailed information on that message buffer.



The “waiting” column shows the task IDs waiting.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

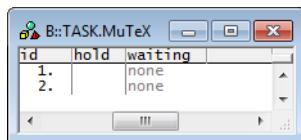
## TASK.MuTeX

### Display mutexes

Format: **TASK.MuTeX [<mutex>]**

Displays the mutex table of REALOS or detailed information about one specific mutex.

Without any arguments, a table with all created mutexes will be shown.  
Specify a mutex ID to display detailed information on that mutex.



Id	hold	waiting
1.	none	none
2.	none	none

“hold” shows the task ID that locked this mutex.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.SEMaphore** [<semaphore>]

Displays the semaphore table of REALOS or detailed information about one specific semaphore.

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

Specify a semaphore ID to display detailed information on that semaphore.

B::TASK.SEMaphore			
id	count	max	waiting
1.	0.	127.	2. 1.
2.	5.	10.	none

The “waiting” column shows the task IDs waiting.

The field “id” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.TaSK** [<task>]

Displays the task table of REALOS or detailed information about one specific task.

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

Specify a task magic or ID to display detailed information on that task.

B::TASK.TaSK			
id	state	prio	entry
0.	ready	33.	C0001F40 _kernel_Idle_task
1.	running	1.	C00005C8 task1
2.	waiting sem	2.	C00005C8 task1
3.	ready	3.	C00005F8 task2

B::TASK.TaSK 2.			
id	state	prio	entry
2.	waiting sem	2.	C00005C8 task1
attributes			
HLNG ACT			
initial prio base prio timeout			
2. 2. FFFFFFFF			
counter activation wakeup suspended			
0. 0. 0. 0.			
entry point extra info symbol			
C00005C8 00000002 task1			
stack size pointer			
C001EE70 00001000 C001FE10			
waiting object			
1.			

The fields “id” and “entry” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on them will show a local menu.

# REALOS PRACTICE Functions

---

There are special definitions for REALOS specific PRACTICE functions.

## **TASK.CONFIG()**

OS Awareness configuration information

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

**Parameter and Description:**

<b>magic</b>	<b>Parameter Type:</b> <a href="#">String</a> ( <i>without</i> quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its <a href="#">task magic number</a> ).
<b>magicsize</b>	<b>Parameter Type:</b> <a href="#">String</a> ( <i>without</i> quotation marks). Returns the size of the task magic number (1, 2 or 4).

**Return Value Type:** [Hex value](#).