LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
RTXC Quadros

Release 02.2026

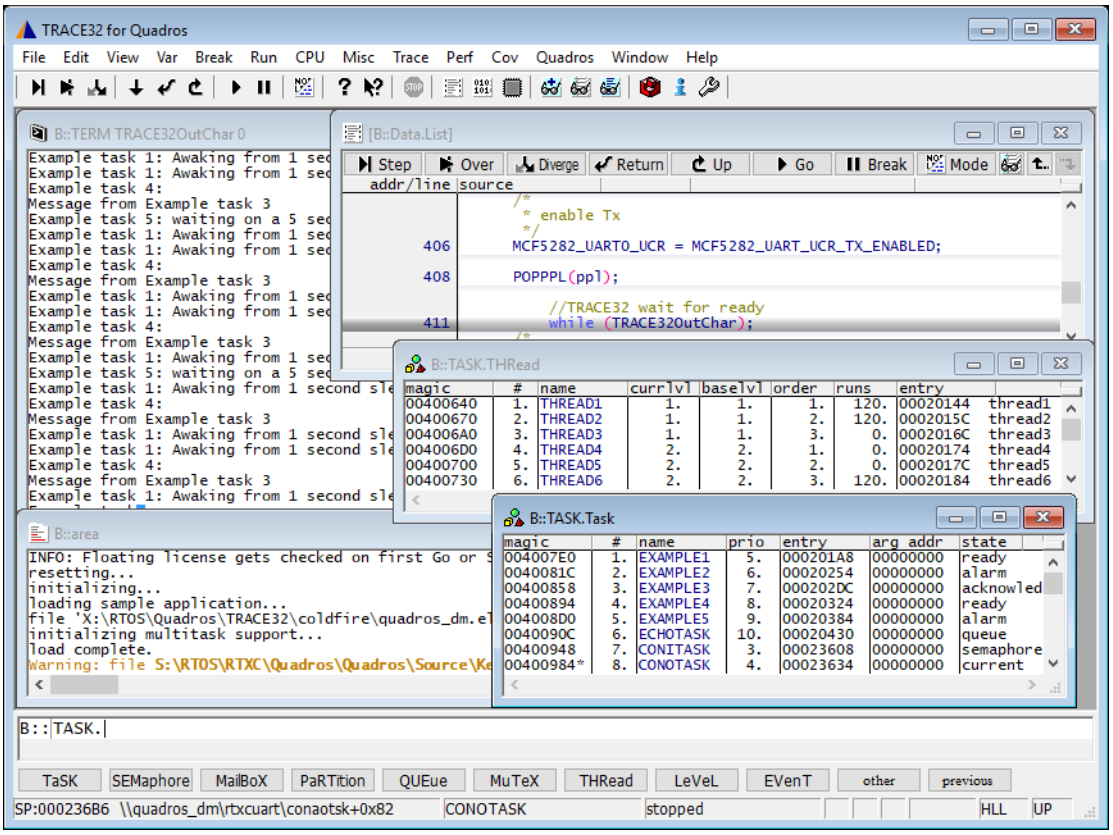
MANUAL

OS Awareness Manual RTXC Quadros

OS Awareness Manual RTXC Quadros

Version 03-Apr-2026

Overview



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

Terminology

RTXC Quadros uses the terms “threads” and “tasks”. If not otherwise specified, the TRACE32 term “task” corresponds to RTXC Quadros (multi-stack) tasks.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Tutorial"** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.
- **"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.

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**.

Supported Versions

Currently RTXC Quadros is supported for the following versions:

- RTXC Quadros (ms and ss) V1.0 on ARM, C167, ColdFire, PowerPC, StarCore and TMS320C55xx.

Configuration

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

Automatic configuration tries to locate the RTXC Quadros 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 have dual port access for the display functions (display “On The Fly”), you have to map emulation or shadow memory to the address space of all used system tables.

For system resource display, 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:	**TASK.CONFIG quadros**

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

Quick Configuration Guide

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

1. Copy the files "`quadros.t32`" and "`quadros.men`" to your project directory (from TRACE32 directory "~~/demo/<*processor*>/kernel/quadros").
2. Start the TRACE32 Debugger.
3. Load your application as normal.
4. Execute the command "`TASK.CONFIG quadros`" (See "**Configuration**").
5. Execute the command "`MENU.ReProgram quadros`" (See "**RTXC Quadros Specific Menu**").
6. Start your application.

Now you can access the RTXC Quadros extensions through the menu.

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

Hooks & Internals in RTXC Quadros

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 RTXC Quadros kernel must be compiled with debug information.

Features

The OS Awareness for RTXC Quadros supports the following features.

Terminal Emulation

The terminal emulation window can be used to communicate with the target side terminal I/O. The communication via two memory buffers requires no external interface. See the **TERM** command group for a description of the terminal emulation. On request we can provide you with the source code for the target interface routines for RTXC Quadros.

Display of Kernel Resources

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

TASK.ALaRm	Alarms
TASK.CouNTer	Counters
TASK.EVenT	Event sources
TASK.EXCeption	Exceptions
TASK.LeVeL	Levels
TASK.MailBoX	Mailboxes
TASK.MuTeX	Mutexes
TASK.PaRTition	Partitions
TASK.PIPe	Pipes
TASK.QUEue	Queues
TASK.SEMaphore	Semaphores
TASK.TaSK	Tasks
TASK.THRead	Threads

For a description of the commands, refer to chapter "**RTXC Quadros Commands**".

When working with emulation memory or shadow memory, these resources can be displayed "On The Fly", i.e. while the target application is running, without any intrusion to the application. If using this dual port memory feature, be sure that emulation memory is mapped to all places, where RTXC Quadros holds its tables.

When working only with target memory, the information will only be displayed if the target application is stopped.

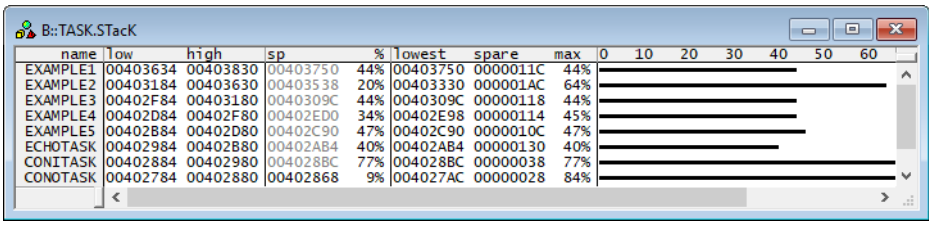
Task Stack Coverage

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-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.

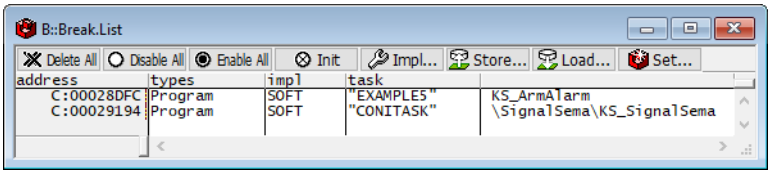
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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



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**, **List.auto**, **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.

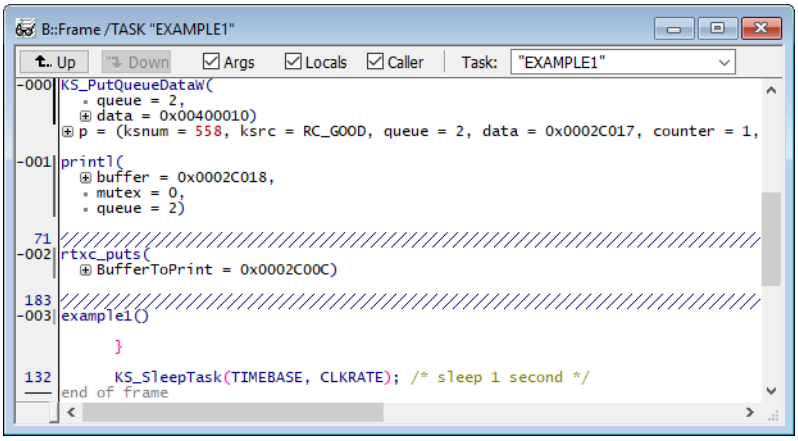
- Use a magic number, task ID, or task name for <*task*>. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.

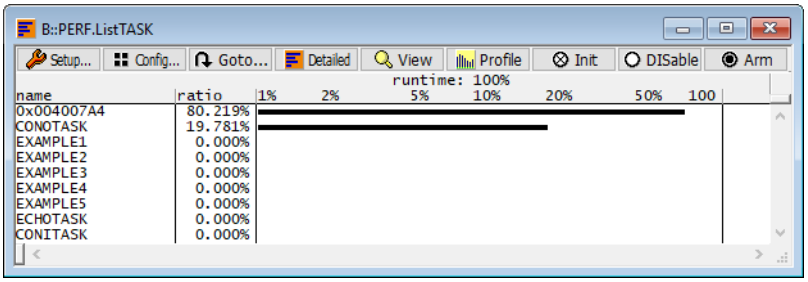


Dynamic Task Performance Measurement

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 (**TRANSlation.CONFIG.MMUSPACES ON**), then you need to set **PERF.CONFIG.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

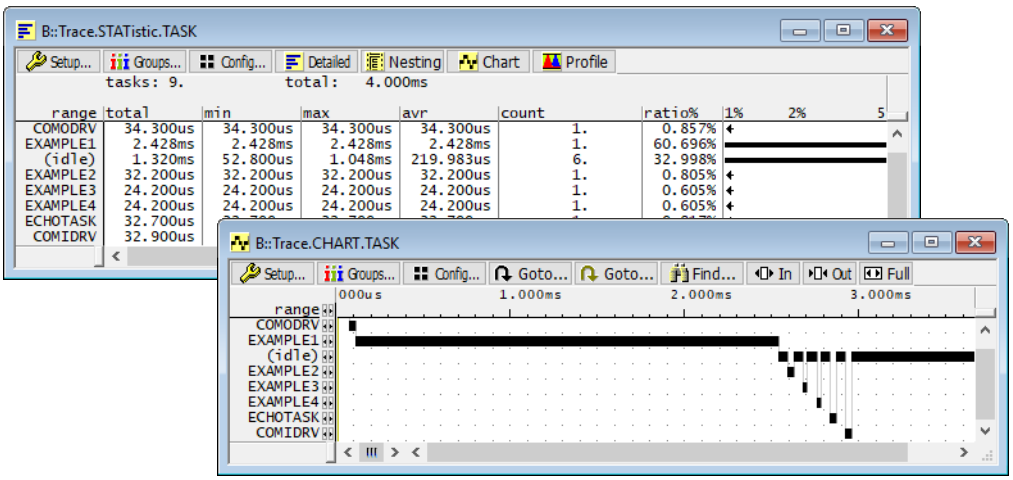
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **LOGGER**). For details, refer to **"OS-aware Tracing"** in TRACE32 Concepts, page 36 (trace32_concepts.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)".



Task State Analysis

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches and data accesses (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate a 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"** in TRACE32 Concepts, page 36 (trace32_concepts.pdf).

The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically.

This feature requires that the following data accesses are recorded:

- All accesses to the status words of all tasks
- Accesses to the current task variable (= magic address)

Adjust your trace logic to record all data write accesses, or limit the recorded data to the area where all TCBs are located (plus the current task pointer).

Example: This script assumes that the TCBs are located in an array named TCB_array and consequently limits the tracing to data write accesses on the TCBs and the task switch.

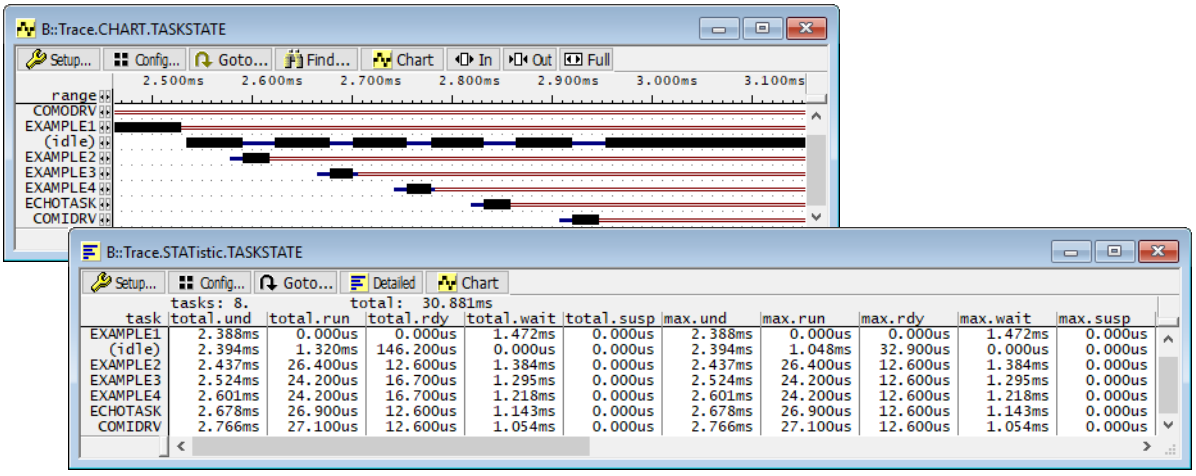
```
Break.Set Var.RANGE(TCB_array) /Write /TraceData
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state timechart

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

All kernel activities up to the task switch are added to the calling task.



Function 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"** in TRACE32 Concepts, page 36 (trace32_concepts.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
```

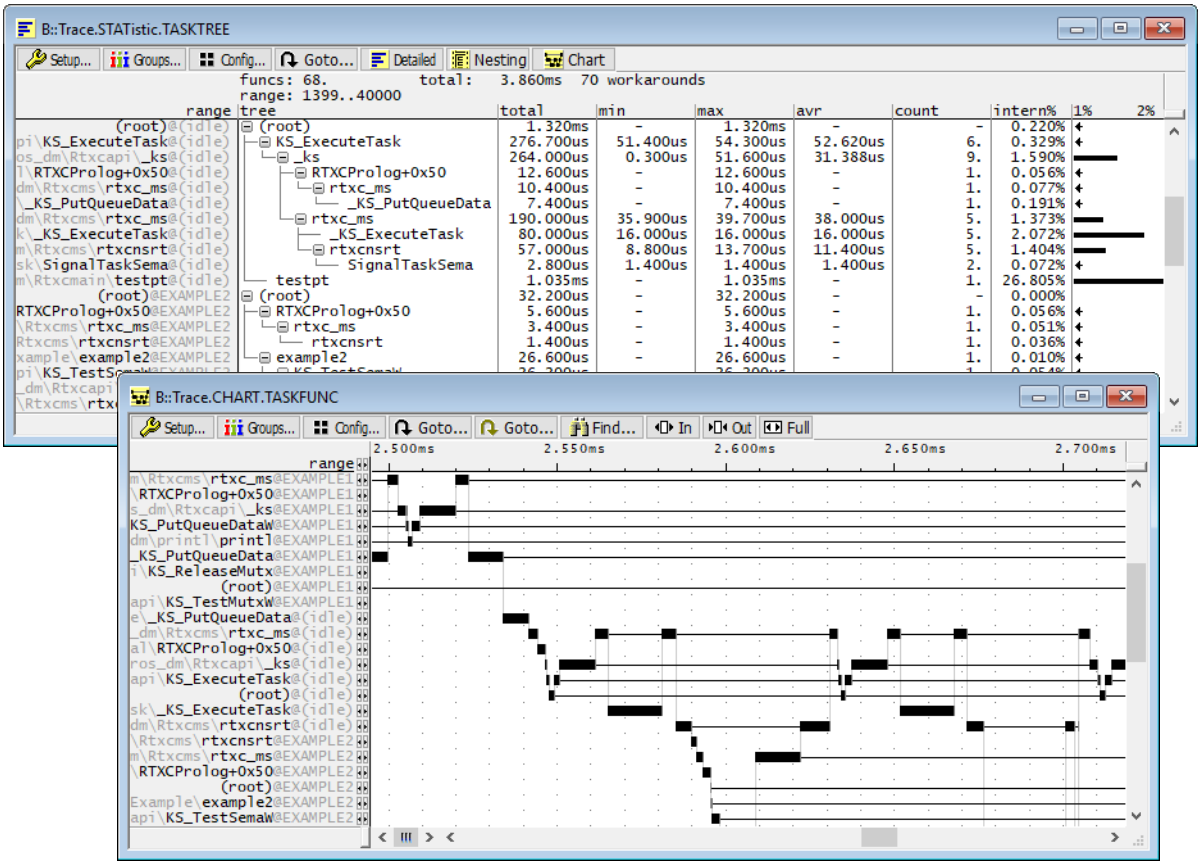
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



RTXC Quadros specific Menu

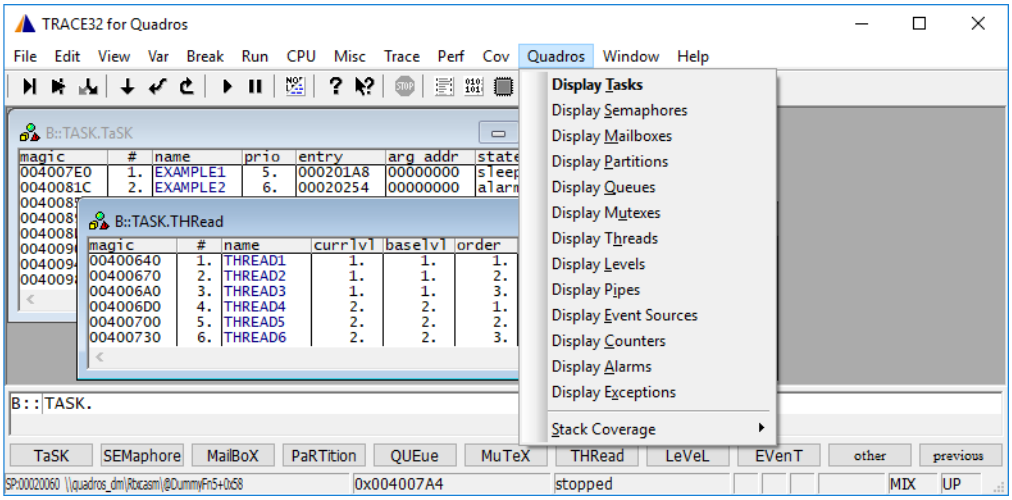
The menu file “quadros.men” contains a menu with RTXC Quadros specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the RTXC Quadros 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.

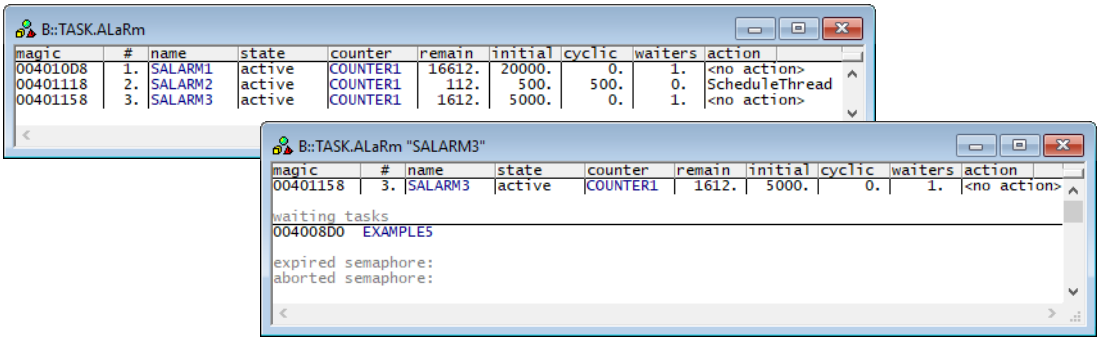


TASK.ALaRm — Display alarms

Format: **TASK.ALaRm** [*<alarm>*]

Displays the alarm table of RTXC Quadros or detailed information about one specific alarm.

Without any arguments, a table with all created alarms will be shown. Specify an alarm name, alarm ID or alarm magic number to display detailed information on that alarm.



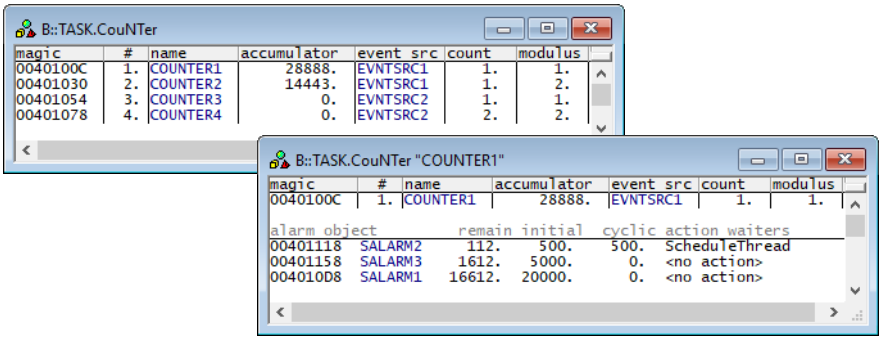
"magic" is a unique ID, used by the OS Awareness to identify a specific alarm (address of the alarm control structure).

The fields "magic" and "waiting tasks" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

Format: **TASK.CouNTer** [<*counter*>]

Displays the counter table of RTXC Quadros or detailed information about one specific counter.

Without any arguments, a table with all created counters will be shown. Specify a counter name, counter ID or counter magic number to display detailed information on that counter.



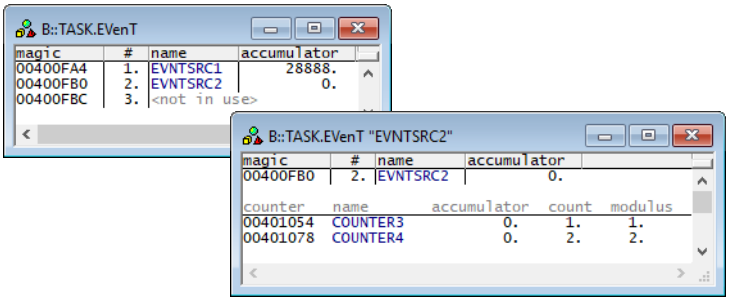
“magic” is a unique ID, used by the OS Awareness to identify a specific counter (address of the counter control structure).

The fields “magic” and “alarm” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

Format: **TASK.EVenT** [*<event>*]

Displays the event source table of RTXC Quadros or detailed information about one specific event source.

Without any arguments, a table with all created event sources will be shown. Specify a event source name, event ID or event magic number to display detailed information on that event source.



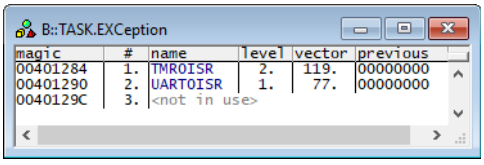
"magic" is a unique ID, used by the OS Awareness to identify a specific event source (address of the event source control structure).

The fields "magic" and "counter" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.EXCeption Display exceptions

Format: **TASK.EXCeption**

Displays a table with all created exceptions of RTXC Quadros.

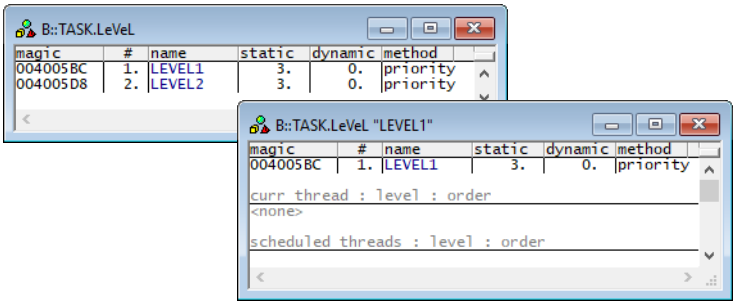


"magic" is a unique ID, used by the OS Awareness to identify a specific exception (address of the exception control structure).

Format: **TASK.LeVeL** [*<level>*]

Displays the level table of RTXC Quadros or detailed information about one specific level.

Without any arguments, a table with all created levels will be shown. Specify a level name, level ID or level magic number to display detailed information on that level.



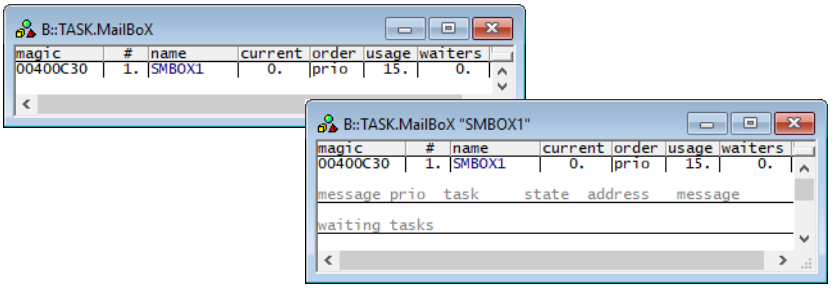
“magic” is a unique ID, used by the OS Awareness to identify a specific level (address of the level control structure).

The fields “magic” and “thread” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

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

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

Without any arguments, a table with all created mailboxes will be shown. Specify a mailbox name, mailbox ID or mailbox magic number to display detailed information on that mailbox.



"magic" is a unique ID, used by the OS Awareness to identify a specific mailbox (address of the mailbox control structure).

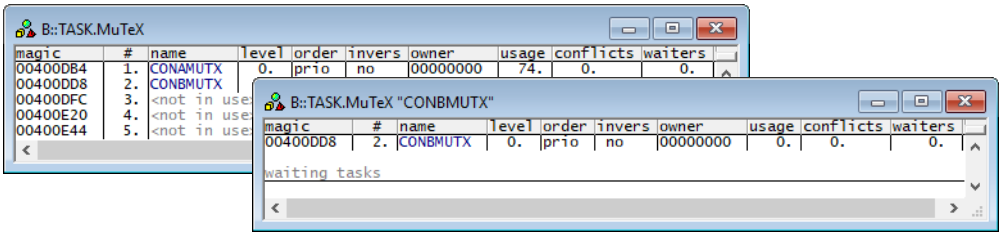
The fields "magic", "address" and "waiting tasks" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.MuTeX Display mutex

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

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

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



"magic" is a unique ID, used by the OS Awareness to identify a specific mutex (address of the mutex control structure).

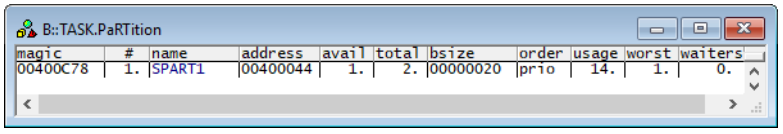
The fields “magic”, “owner” and “waiting tasks” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.PaRTition Display partitions

Format: **TASK.PaRTition** [*<partition>*]

Displays the partition table of RTXC Quadros or detailed information about one specific partition.

Without any arguments, a table with all created partitions will be shown. Specify a partition name, partition ID or partition magic number to display detailed information on that partition.



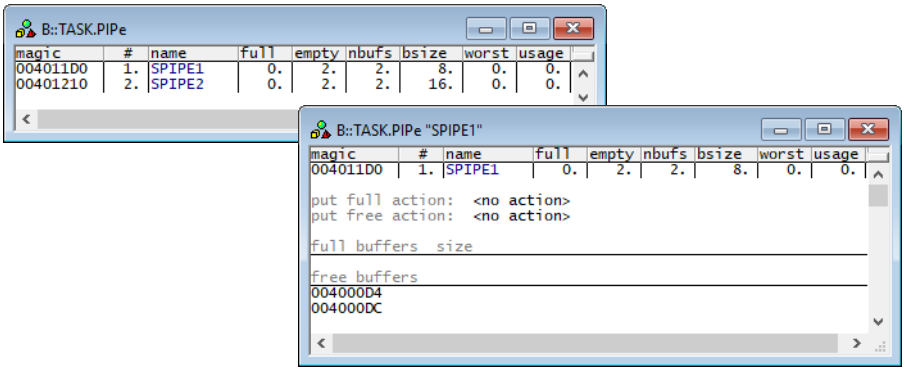
“magic” is a unique ID, used by the OS Awareness to identify a specific partition (address of the partition control structure). The fields “magic”, “address” and “waiting tasks” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.PIPe Display pipes

Format: **TASK.PIPe** [*<pipe>*]

Displays the pipe table of RTXC Quadros or detailed information about one specific pipe.

Without any arguments, a table with all created pipes will be shown. Specify a pipe name, pipe ID or pipe magic number to display detailed information on that pipe.



“magic” is a unique ID, used by the OS Awareness to identify a specific pipe (address of the pipe control structure).

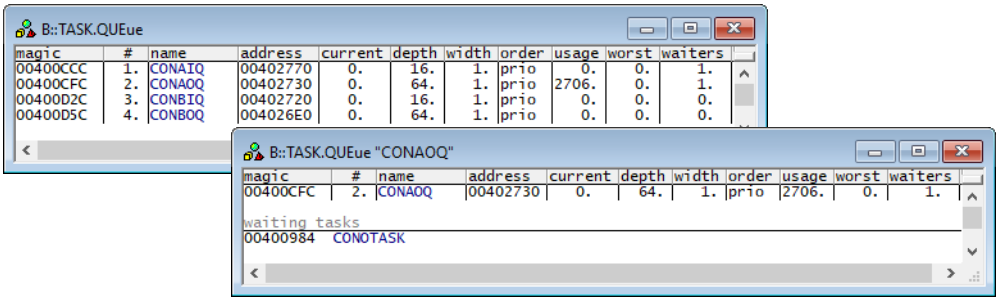
The fields “magic”, “full buffers” and “free buffers” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.QUEue Display queues

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

Displays the queue table of RTXC Quadros or detailed information about one specific queue.

Without any arguments, a table with all created queues will be shown.
Specify a queue name, queue ID or queue magic number to display detailed information on that queue.



“magic” is a unique ID, used by the OS Awareness to identify a specific queue (address of the queue control structure).

The fields “magic”, “address” and “waiting tasks” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

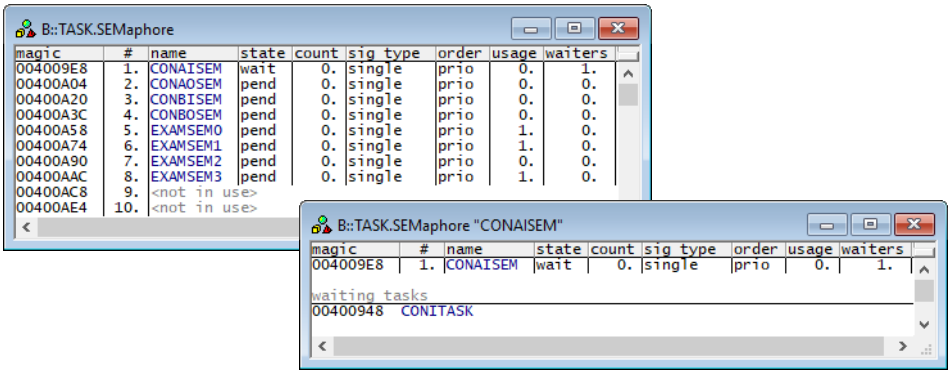
TASK.SEMaphore Display semaphores

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

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

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

Specify a semaphore name, semaphore ID or semaphore magic number to display detailed information on that semaphore.



“magic” is a unique ID, used by the OS Awareness to identify a specific semaphore (address of the semaphore control structure).

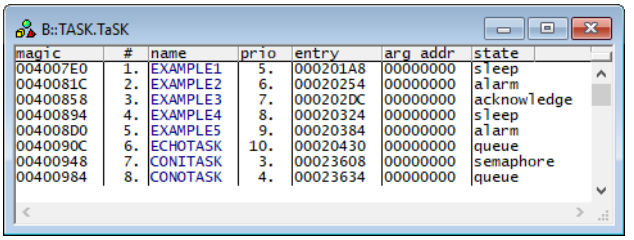
The fields “magic” and “waiting tasks” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

TASK.TaSK Display tasks

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

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

Without any arguments, a table with all created tasks will be shown.
Specify a task name, task ID or task magic number to display detailed information on that task.



“magic” is a unique ID, used by the OS Awareness to identify a specific task (address of the task control structure).

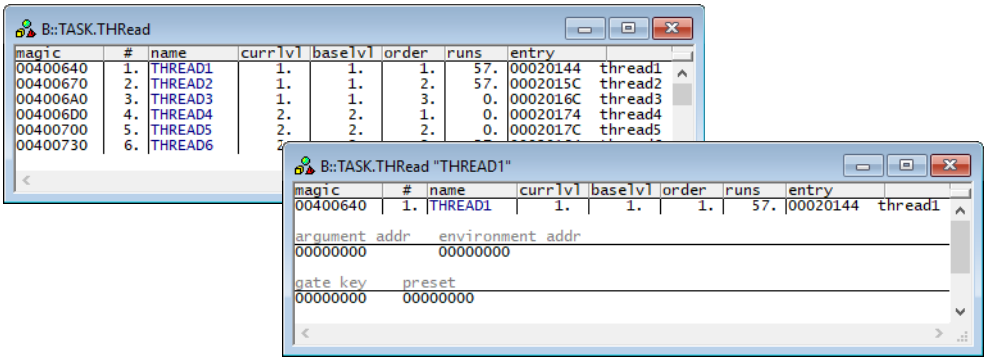
The fields “magic”, “entry” and “arg addr” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on them will show a local menu.

Pressing the “context” button (if available) changes the register context to this task. “current” resets it to the current context. See “**Task Context Display**”.

Format: **TASK.THRead** [*<thread>*]

Displays the thread table of RTXC Quadros or detailed information about one specific thread.

Without any arguments, a table with all created threads will be shown. Specify a thread name, thread ID or thread magic number to display detailed information on that thread.



“magic” is a unique ID, used by the OS Awareness to identify a specific thread (address of the thread control structure).

The fields “magic”, “entry” and “addr” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

RTXC Quadros PRACTICE Functions

There are special definitions for RTXC Quadros specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

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

Parameter and Description:

magic	**Parameter Type**: String (***without*** quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number). If the application is a multi-stack or dual-mode system, the magic address of multi-stack is returned, otherwise the magic address of single-stack is returned.
magicms	Returns the address of the magic number for the multi-stack scheduler (= address of the current running task).
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).
magicss	**Parameter Type**: String (***without*** quotation marks). Returns the address of the magic number for the single-stack scheduler (= address of the current running thread).

Return Value Type: Hex value.

TASK.VERSION() Awareness information

Syntax: **TASK.VERSION(***<item>* | **cpufamily | date | rtos)**

Parameter and Description:

<item>	**Parameter Type**: String (***without*** quotation marks). Reports awareness version information.
cpufamily	**Parameter Type**: Returns the CPU family name this awareness is for.

date	**Parameter Type**: String (***without*** quotation marks). Returns the compile date of this awareness.
rtos	**Parameter Type**: String (***without*** quotation marks). Returns the OS name this awareness is for.

Return Value Type: String.

TASK.TASK.LIST() Next task magic number in task list

Syntax: **TASK.TASK.LIST(**<*task_magic*>**)**

Returns the next task magic number in the task list. Specify zero for the first task. Returns zero if no further task is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.TASK.NAME() Task name

Syntax: **TASK.TASK.NAME(**<*task_magic*>**)**

Returns the name of the specified task.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.TASK.ID2MAGIC() Task magic number of task ID

Syntax: **TASK.TASK.ID2MAGIC(**<*task_id*>**)**

Returns the task magic number of a given task ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.THREAD.LIST() Next thread magic number in the thread list

Syntax: **TASK.THREAD.LIST(**<*task_magic*>**)**

Returns the next thread magic number in the thread list. Specify zero for the first thread. Returns zero if no further thread is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.THREAD.NAME() Name of thread

Syntax: **TASK.THREAD.NAME(**<*thread_magic*>**)**

Returns the name of the specified thread.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.THREAD.ID2MAGIC() Thread magic number of thread ID

Syntax: **TASK.THREAD.ID2MAGIC(**<*thread_id*>**)**

Returns the thread magic number of a given thread ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.SEMAPHORE.ID2MAGIC() Magic number of a given semaphore ID

Syntax: **TASK.SEMAPHORE.ID2MAGIC(**<*semaphore_id*>**)**

Returns the semaphore magic number of a given semaphore ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.SEMAPHORE.LIST() Next magic number in the semaphore list

Syntax: **TASK.SEMAPHORE.LIST(***<semaphore_magic>***)**

Returns the next semaphore magic number in the semaphore list. Specify zero for the first semaphore. Returns zero if no further semaphore is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.SEMAPHORE.NAME() Name of semaphore

Syntax: **TASK.SEMAPHORE.NAME(***<semaphore_magic>***)**

Returns the semaphore name for the specified semaphore magic number.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.SEMAPHORE.STATE() State of semaphore

Syntax: **TASK.SEMAPHORE.STATE(***<semaphore_magic>***)**

Returns the state of the semaphore.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.SEMAPHORE.COUNT() Count of semaphore

Syntax: **TASK.SEMAPHORE.COUNT(***<semaphore_magic>***)**

Returns the count of the semaphore.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

Syntax: **TASK.SEMAPHORE.WAITERS.COUNT(**<*semaphore_magic*>**)**

Returns the number of tasks waiting on this semaphore.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.SEMAPHORE.WAITERS.LIST() Next task magic number

Syntax: **TASK.SEMAPHORE.WAITERS.LIST(**<*semaphore_magic*>**,**<*task_magic*>**)**

Returns the next task magic number in the waiting list.

Parameter and Description:

<*semaphore_magic*>	**Parameter Type**: Decimal or hex or binary value.
<*task_magic*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.MUTEX.LIST() Next mutex magic number in mutex list

Syntax: **TASK.MUTEX.LIST(**<*mutex_magic*>**)**

Returns the next mutex magic number in the mutex list. Specify zero for the first mutex. Returns zero if no further mutex is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.MUTEX.NAME() Name of mutex

Syntax: **TASK.MUTEX.NAME(**<*mutex_magic*>**)**

Returns the mutex name for the specified mutex magic number.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.MUTEX.ID2MAGIC() Mutex magic number of mutex ID

Syntax: **TASK.MUTEX.ID2MAGIC(**<*mutex_id*>**)**

Returns the mutex magic number for the specified mutex ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.MUTEX.WAITERS.COUNT() Tasks waiting on mutex

Syntax: **TASK.MUTEX.WAITERS.COUNT(**<*mutex_magic*>**)**

Returns the number of tasks waiting on this mutex.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

Syntax: **TASK.MUTEX.WAITERS.LIST(***<mutex_magic>***, *<task_magic>*)**

Returns the next task magic number in the waiting list.

Parameter and Description:

<mutex_magic>	**Parameter Type**: Decimal or hex or binary value.
<task_magic>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.QUEUE.LIST() Next queue magic number in queue list

Syntax: **TASK.QUEUE.LIST(***<queue_magic>***)**

Returns the next queue magic number in the queue list. Specify zero for the first queue. Returns zero if no further queue is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.QUEUE.NAME() Name of queue

Syntax: **TASK.QUEUE.NAME(***<queue_magic>***)**

Returns the queue name for the specified queue magic number.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.QUEUE.ID2MAGIC() Queue magic number of queue ID

Syntax: **TASK.QUEUE.ID2MAGIC(***<queue_id>***)**

Returns the queue magic number for the specified queue ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.QUEUE.WAITERS.COUNT() Tasks waiting on this queue

Syntax: **TASK.QUEUE.WAITERS.COUNT(***<queue_magic>***)**

Returns the number of tasks waiting on this queue.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.QUEUE.WAITERS.LIST() Next task magic number in waiting list

Syntax: **TASK.QUEUE.WAITERS.LIST(***<queue_magic>***, *<task_magic>***)**

Returns the next task magic number in the waiting list.

Parameter and Description:

<queue_magic>	**Parameter Type**: Decimal or hex or binary value.
<task_magic>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PIPE.LIST() Next pipe magic number in pipe list

Syntax: **TASK.PIPE.LIST(**<*pipe_magic*>**)**

Returns the next pipe magic number in the pipe list. Specify zero for the first pipe. Returns zero if no further pipe is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PIPE.NAME() Name of pipe

Syntax: **TASK.PIPE.NAME(**<*pipe_magic*>**)**

Returns the pipe name for the specified pipe magic number.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.PIPE.ID2MAGIC() Magic number of pipe ID

Syntax: **TASK.PIPE.ID2MAGIC(**<*pipe_id*>**)**

Returns the magic number of a given pipe ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.