OS Awareness Manual
ARTX-166

MANUAL

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .... 🗁

OS Awareness Manuals .... 🗁

OS Awareness Manual ARTX-166 1

Overview 3

Brief Overview of Documents for New Users 4

Supported Versions 4

Configuration 5

Quick Configuration Guide 5

Hooks & Internals in ARTX-166 6

Features 7

Display of Kernel Resources 7

Task Stack Coverage 7

Task-Related Breakpoints 8

Dynamic Task Performance Measurement 9

Task Runtime Statistics 9

Task State Analysis 10

Function Runtime Statistics 11

ARTX-166 Specific Menu 13

ARTX-166 Commands 14

TASK.Task Display tasks 14

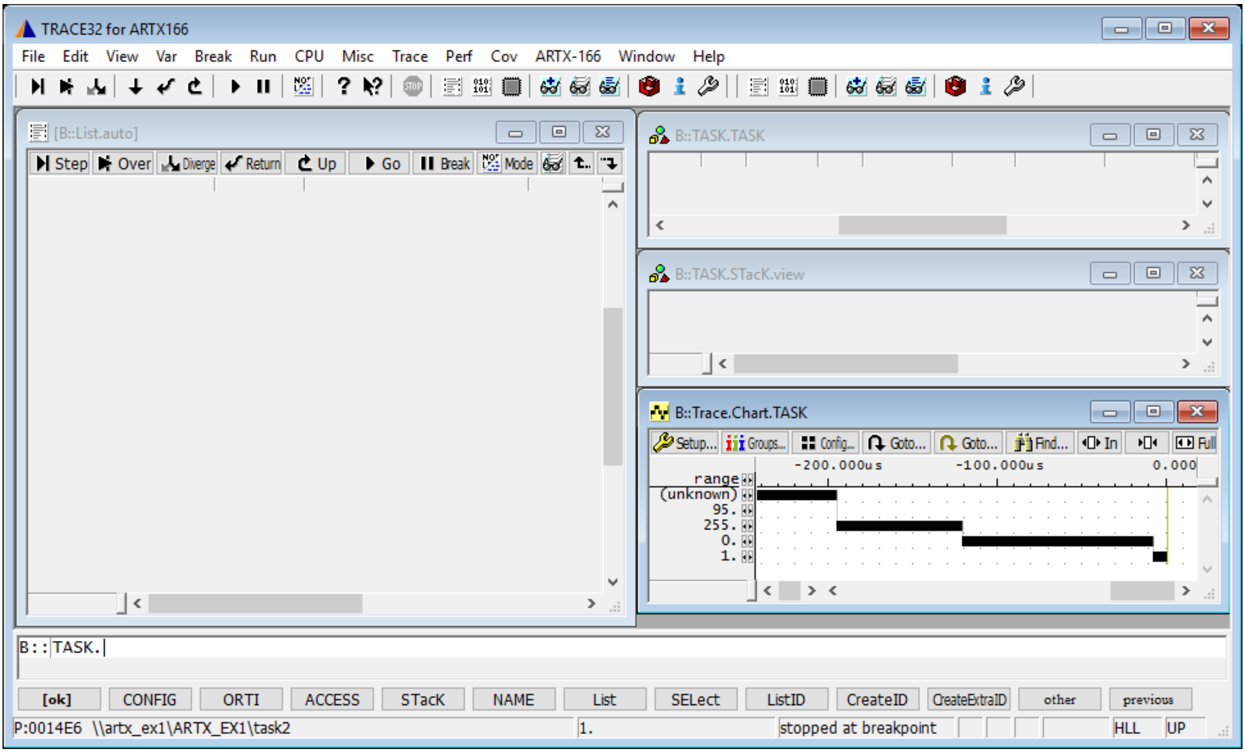
ARTX-166 PRACTICE Functions 15

TASK.CONFIG() OS Awareness configuration information 15

OS Awareness Manual ARTX-166

Version 06-Jun-2024

Overview


TRACE32 for ARTX166
File Edit View Var Break Run CPU Misc Trace Perf Cov ARTX-166 Window Help
[B::List.auto]
Step Over Diverge Return Up Go Break Mode
B::TASK.TASK
B::TASK.STacK.view
B::Trace.Chart.TASK
Setup... Groups... Config... Goto... Goto... Find... In Full
-200.000us -100.000us 0.000
range
(unknown)
95.
255.
0.
1.
B::TASK.
[ok] CONFIG ORTI ACCESS STacK NAME List SELect ListID CreateID CreateExtraID other previous
P:0014E6 \\artx_ex1\ARTX_EX1\task2 1. stopped at breakpoint HLL UP

The OS Awareness for ARTX-166 contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

Brief Overview of Documents for New Users

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 ARTX-166 is supported for the following versions:

- ARTX-166 V3.x on all C166/C167 derivatives

Configuration

The **TASK.CONFIG** command loads an extension definition file called “artx166.t32” (directory “~~/demo/*<processor>*/kernel/artx166”). It contains all necessary extensions.

Automatic configuration tries to locate the ARTX-166 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:        TASK.CONFIG  artx166
```

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 ARTX-166 with your application, follow the following roadmap:

1. Copy the files “`artx166.t32`” and “`artx166.men`” to your project directory
   (from TRACE32 directory “~~/demo/<*processor*>/kernel/rtx”).

2. Start the TRACE32 Debugger.

3. Load your application as normal.

4. Execute the command “`TASK.CONFIG artx166`”
   (See “**Configuration**”).

5. Execute the command “`MENU.ReProgram artx166`”
   (See “**ARTX-166 Specific Menu**”).

6. Start your application.

Now you can access the ARTX-166 extensions through the menu.

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

Hooks & Internals in ARTX-166

No hooks are used in the kernel.

For detecting the current running task, the kernel symbol “`os_runtask`” is used.

For retrieving the kernel data and 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.

Be sure that your application is compiled and linked with debugging symbols switched on.

Features

The OS Awareness for ARTX-166 supports the following features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following ARTX-166 components can be displayed:

TASK.Task	Tasks

For a description of the commands, refer to chapter “**ARTX-166 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.

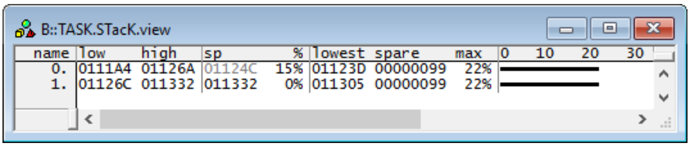
Task Stack Coverage

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.



```
 name |low     high   |sp           % |lowest  spare       max |0     10     20     30
   0. |0111A4  01126A |01124C     15% |01123D  00000099    22% |
   1. |01126C  011332 |011332      0% |011305  00000099    22% |
```

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.

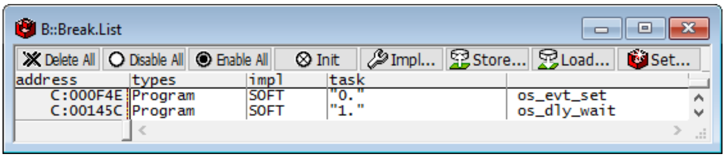
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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



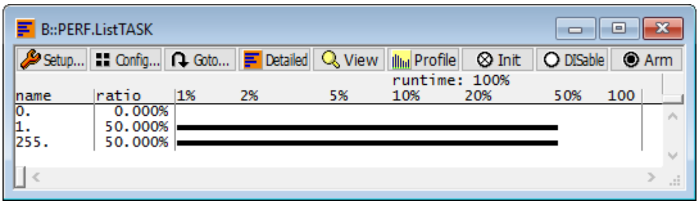
```
address       types          impl     task
    C:000F4E  Program        SOFT     "0."                      os_evt_set
    C:00145C  Program        SOFT     "1."                      os_dly_wait
```

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).



```
                                   runtime: 100%
name      |ratio    |1%     2%      5%    10%      20%         50%    100
0.        |   0.000%|
1.        |  50.000%|
255.      |  50.000%|
```

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).

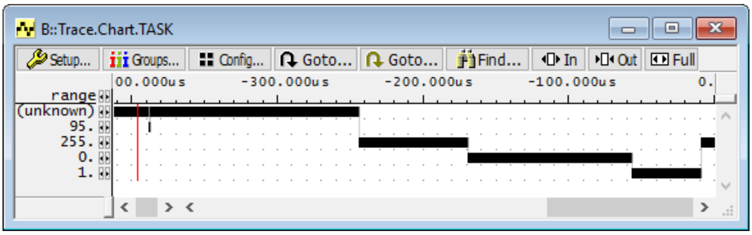
Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

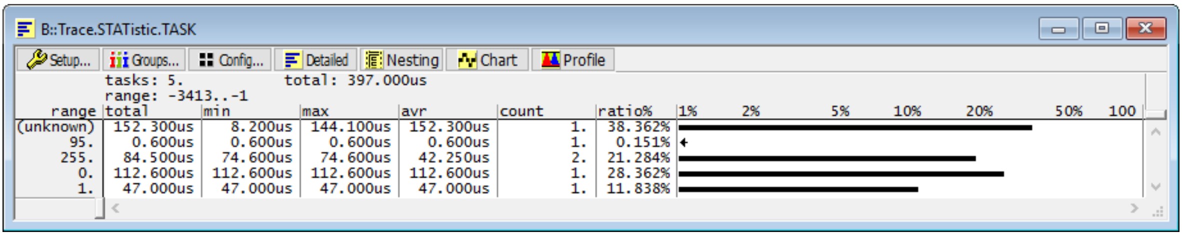
To evaluate the contents of the trace buffer, use these commands:

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)”.





```
            tasks: 5.           total: 397.000us
            range: -3413..-1
      range total      min        max        avr        count        ratio%  1%   2%    5%   10%   20%    50%  100
(unknown)  152.300us    8.200us  144.100us  152.300us           1.  38.362%
      95.    0.600us    0.600us    0.600us    0.600us           1.   0.151%
     255.   84.500us   74.600us   74.600us   42.250us           2.  21.284%
       0.  112.600us  112.600us  112.600us  112.600us           1.  28.362%
       1.   47.000us   47.000us   47.000us   47.000us           1.  11.838%
```

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”** (glossary.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)”.

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”** (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
```

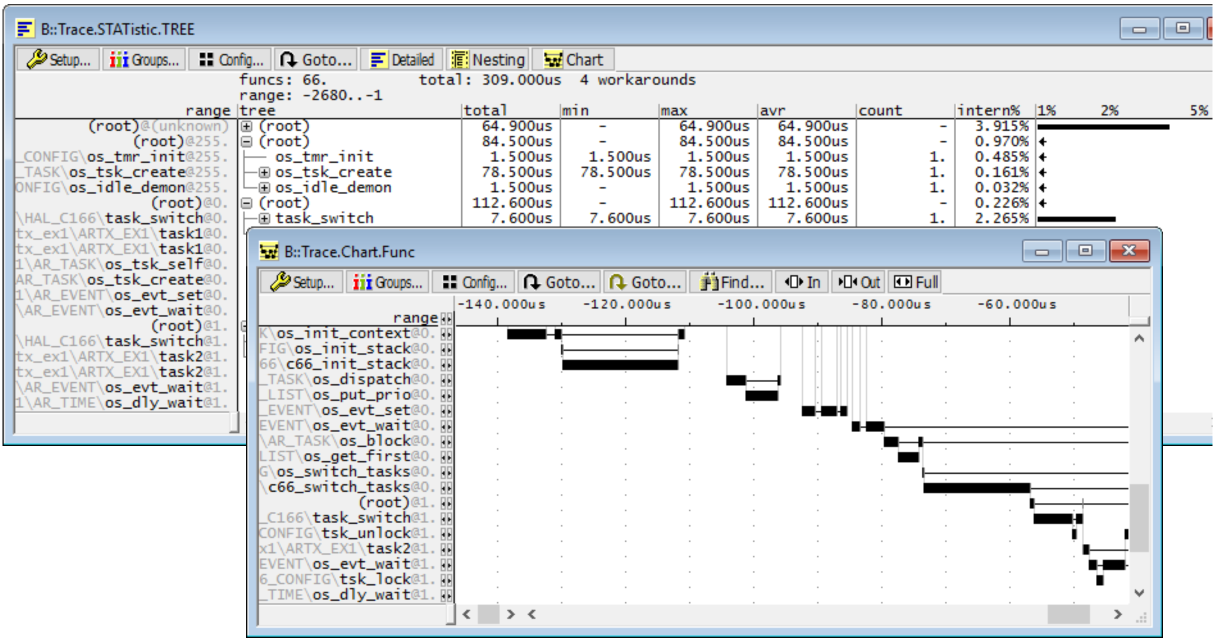
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



```
                     funcs: 66.        total: 309.000us   4 workarounds
                     range: -2680..-1
               range tree                total      min        max        avr        count      intern% 1%    2%    5%
  (root)@(unknown) ⊞ (root)               64.900us         -   64.900us   64.900us          -   3.915%
       (root)@255. ⊟ (root)               84.500us         -   84.500us   84.500us          -   0.970%
_CONFIG\os_tmr_init@255.  ├── os_tmr_init  1.500us   1.500us    1.500us    1.500us          1.  0.485%
_TASK\os_tsk_create@255.  ├⊞ os_tsk_create 78.500us  78.500us   78.500us   78.500us         1.  0.161%
ONFIG\os_idle_demon@255.  └⊞ os_idle_demon 1.500us         -    1.500us    1.500us          1.  0.032%
         (root)@0.   ⊟ (root)              112.600us         -  112.600us  112.600us          -   0.226%
\HAL_C166\task_switch@0.  ├⊞ task_switch   7.600us   7.600us    7.600us    7.600us          1.  2.265%
tx_ex1\ARTX_EX1\task1@0.
tx_ex1\ARTX_EX1\task1@0.
1\AR_TASK\os_tsk_self@0.
AR_TASK\os_tsk_create@0.
1\AR_EVENT\os_evt_set@0.
\AR_EVENT\os_evt_wait@0.
            (root)@1.
\HAL_C166\task_switch@1.
tx_ex1\ARTX_EX1\task2@1.
tx_ex1\ARTX_EX1\task2@1.
\AR_EVENT\os_evt_wait@1.
1\AR_TIME\os_dly_wait@1.
```

```
                  range
K\os_init_context@0.
FIG\os_init_stack@0.
66\c66_init_stack@0.
_TASK\os_dispatch@0.
_LIST\os_put_prio@0.
_EVENT\os_evt_set@0.
EVENT\os_evt_wait@0.
\AR_TASK\os_block@0.
LIST\os_get_first@0.
G\os_switch_tasks@0.
\c66_switch_tasks@0.
            (root)@1.
_C166\task_switch@1.
CONFIG\tsk_unlock@1.
x1\ARTX_EX1\task2@1.
EVENT\os_evt_wait@1.
6_CONFIG\tsk_lock@1.
_TIME\os_dly_wait@1.
```

ARTX-166 Specific Menu

The menu file “rtx.men” contains a menu with ARTX-166 specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **ARTX-166**.

- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the ARTX-166 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:

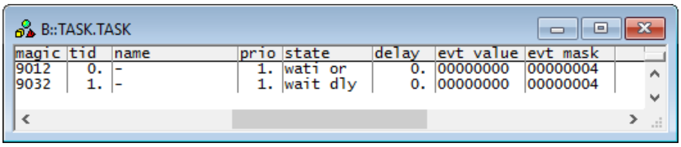
- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with the default display.
- The **Perf** menu contains additional submenus for task runtime statistics and statistics on task states.

TASK.Task

Display tasks

Format: **TASK.Task**

Displays the task table of ARTX-166.


B::TASK.TASK

magic	tid	name	prio	state	delay	evt value	evt mask
9012	0.	-	1.	wati or	0.	00000000	00000004
9032	1.	-	1.	wait dly	0.	00000000	00000004

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

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

ARTX-166 PRACTICE Functions

There are special definitions for ARTX-166 specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

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

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).
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.