OS Awareness Manual Atomthreads

MANUAL

OS Awareness Manual Atomthreads

OS Awareness Manual Atomthreads

Version 06-Jun-2024

Overview

The OS Awareness for Atomthreads 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. Atomthreads 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 Atomthreads documentation.

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 Atomthreads is supported for the following versions:

- Atomthreads on 32 bit ARM, including Cortex-M.

Restrictions

Atomthreads 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 Atomthreads. Please see **"Hooks & Internals in Atomthreads"**, page 6 for more information.

Atomthreads, in its un-modified form, does not provide task IDs or task names. In these cases, always use the task magic number for any task. In any listing where task names would be shown, the task entry function is used instead.

Thread information may be held in a number of disparate locations throughout the system. Currently, the awareness supports a maximum of 32 mutexes, 32 message queues and 32 semaphores. 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. More information about this can be found in **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf).
- Cortex-A based systems do not provide data trace but the context ID feature may be used. This will require a modification to the Atomthreads kernel to cause a trace packet to be generated on each task switch. More details can be found in **"Appendix A: Context ID on Cortex-A Systems"**, page 20.

Configuration

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

The OS Awareness for Atomthreads 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. 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. To configure the OS Awareness, use the command:

Format:	**TASK.CONFIG atomthreads**

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

Quick Configuration Guide

Example scripts are provided in ~~/demo/*<arch>*/kernel/atomthreads/boards. 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/atomthreads/atomthreads.t32
MENU.ReProgram ~~/demo/<arch>/kernel/atomthreads/atomthreads.men
```

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

Hooks & Internals in Atomthreads

No hooks are used in the kernel.

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

To use the awareness's stack checking features, please ensure that you define **STACK_CHECK=TRUE** when building your Atomthreads application.

Features

The OS Awareness for Atomthreads supports the following features.

Display of Kernel Resources

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

TASK.TaskList	Tasks
TASK.SEMaphore	Semaphores
TASK.MuTeX	Mutexes
TASK.Queue	Message Queues
TASK.TIMer	Timers

For a description of the commands, refer to chapter “**Atomthreads PRACTICE Functions**”.

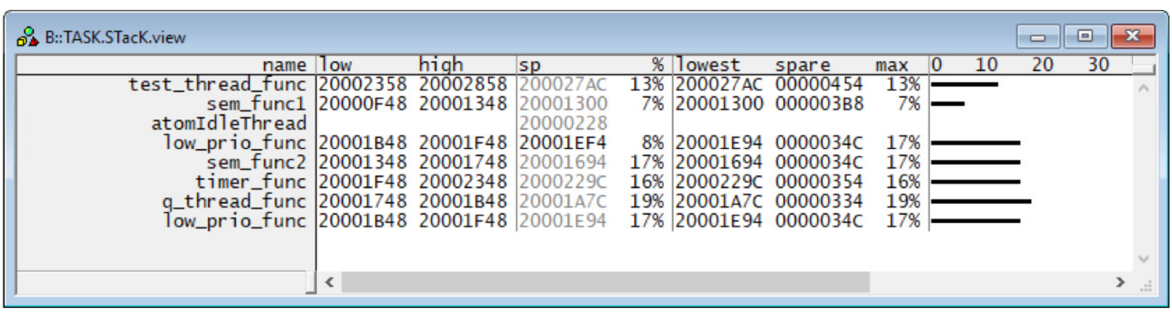
If your hardware allows memory access while the target is running, these resources can be displayed “On The Fly”, i.e. while the application is running, without any intrusion to the application. Be aware that a screen update may occur midway through a scheduling operation which may cause display inconsistencies.

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

Task Stack Coverage

For stack usage coverage of Atomthreads Tasks, you can use the **TASK.STacK** command. Without any parameter, this command will set up a window with all active tasks. If you specify only a task magic number as parameter, the stack area 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 0x5A). The stack display will look like the image below.


B::TASK.STacK.view

name	low	high	sp	%	lowest	spare	max
test_thread_func	20002358	20002858	200027AC	13%	200027AC	00000454	13%
sem_func1	20000F48	20001348	20001300	7%	20001300	000003B8	7%
atomIdleThread			20000228				
low_prio_func	20001B48	20001F48	20001EF4	8%	20001E94	0000034C	17%
sem_func2	20001348	20001748	20001694	17%	20001694	0000034C	17%
timer_func	20001F48	20002348	2000229C	16%	2000229C	00000354	16%
q_thread_func	20001748	20001B48	20001A7C	19%	20001A7C	00000334	19%
low_prio_func	20001B48	20001F48	20001E94	17%	20001E94	0000034C	17%

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 parameter, or omit the parameter and select from the task list 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.

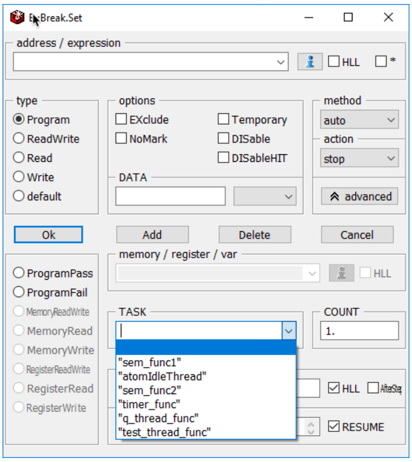
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

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.

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:

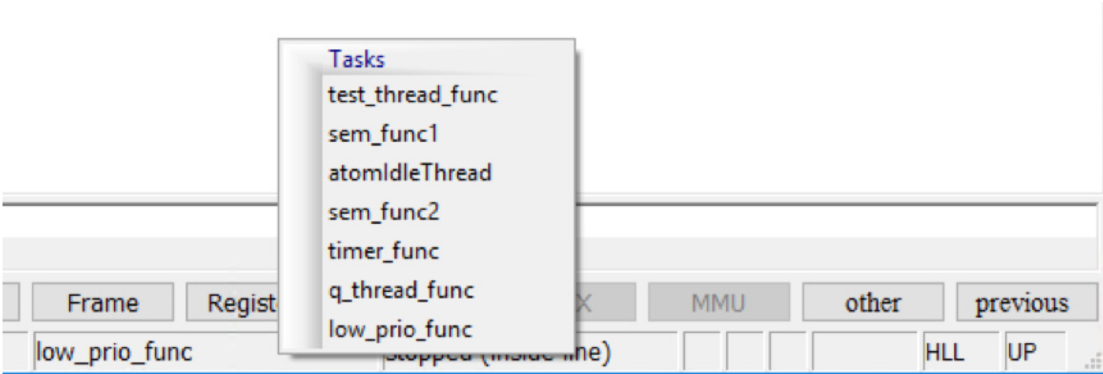
Frame /Task *<task>*	Display call stack of a task.

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 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.

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:

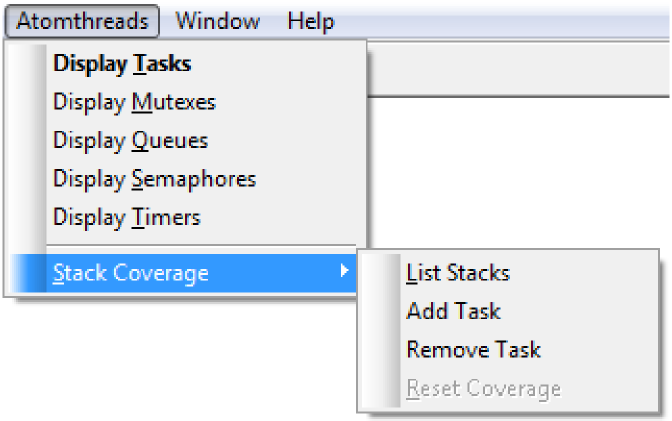
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat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

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

Atomthreads Specific Menu

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

You will find a new menu called **Atomthreads** which looks like the image below.



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

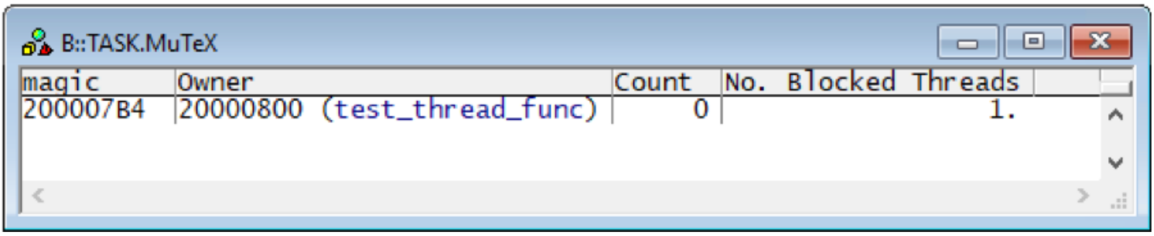
- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with the default display.
- The **Perf** menu contains additional submenus for task runtime statistics and statistics on task states.

TASK.MuTeX — Display mutexes

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

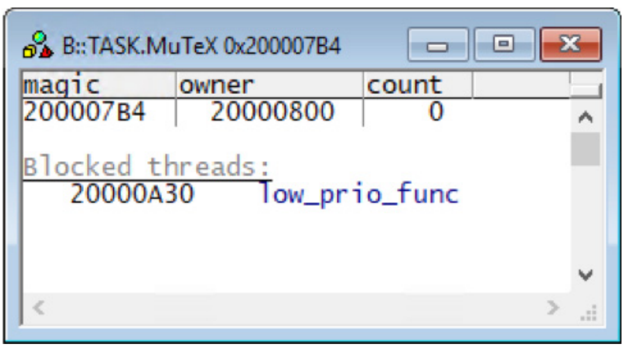
Displays the mutex table of Atomthreads or detailed information about one specific mutex.

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

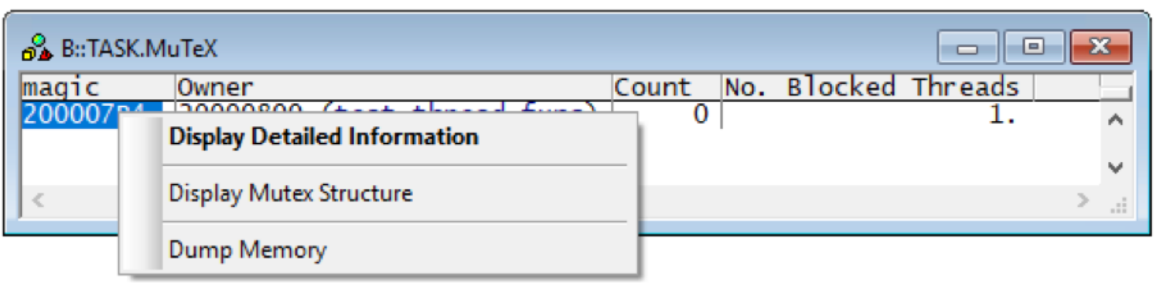


<mutex_magic>	Specify a mutex magic number of a mutex to display detailed information on that mutex. “magic” is a unique ID, used by the OS Awareness to identify a specific semaphore (address of the ATOM_MUTEX structure).

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



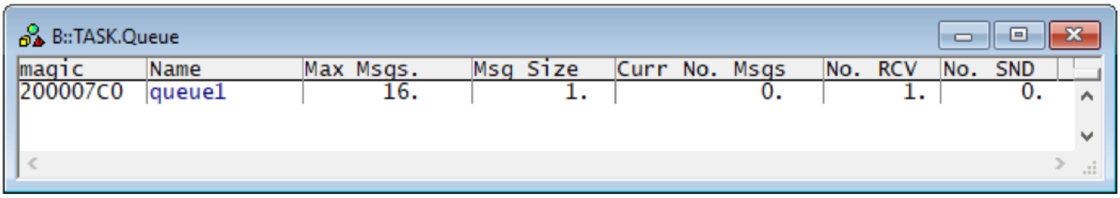
Right-clicking it will show a local menu.



Format:	**TASK.Queue** [*<message_queue_magic>*]

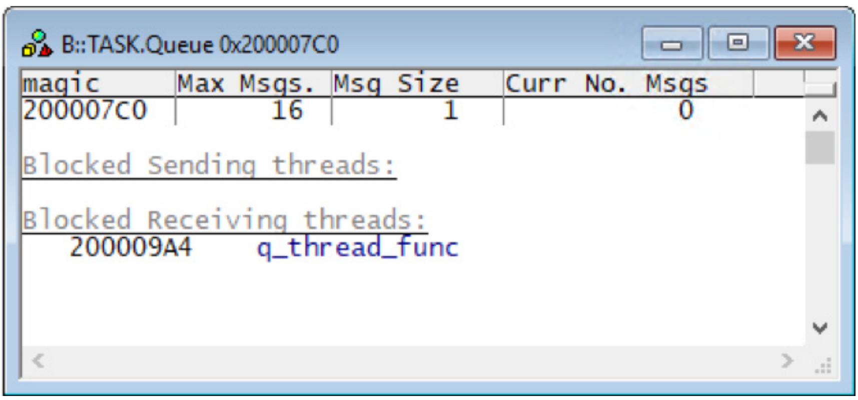
Displays the message queue table of Atomthreads or detailed information about one specific message queue.

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

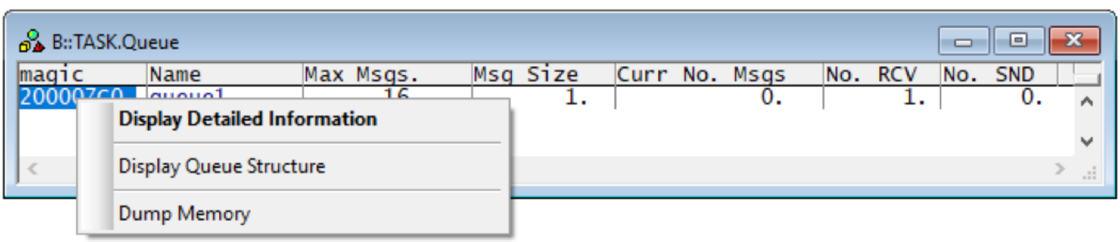


<message_queue_magic>	Specify a message queue magic number to display detailed information on that message queue. “magic” is a unique ID, used by the OS Awareness to identify a specific message queue (address of the ATOM_QUEUE structure).

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



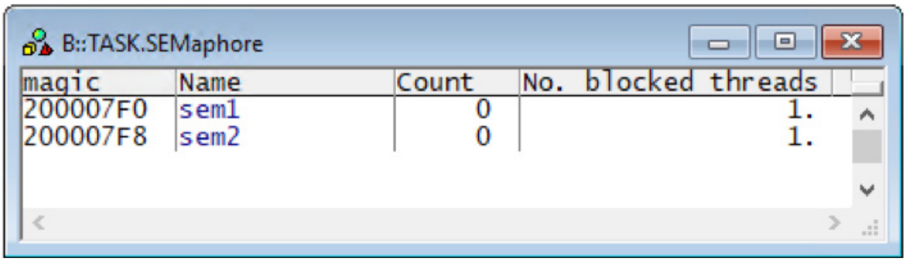
Right-clicking it will show a local menu.



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

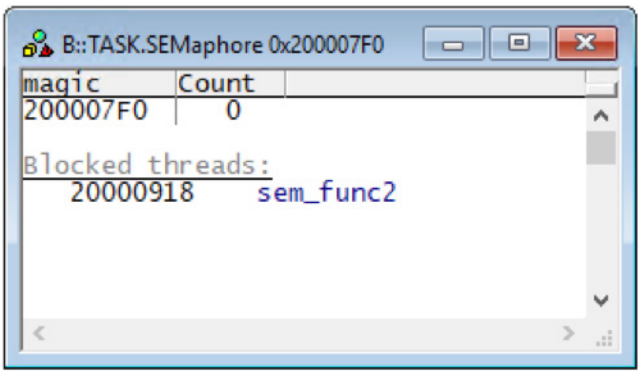
Displays the semaphore table of Atomthreads or detailed information about one specific semaphore

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

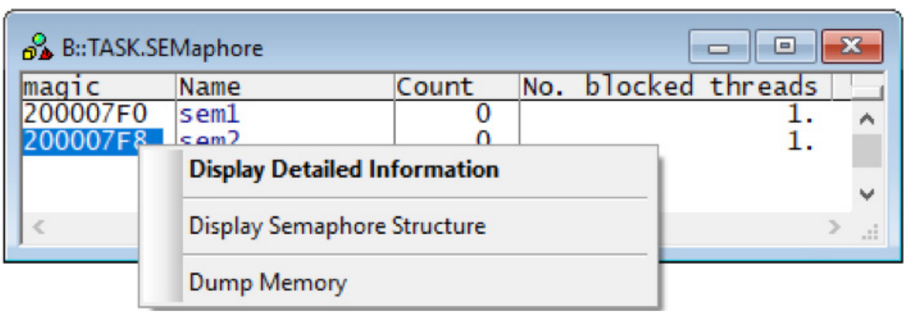


<semaphore_magic>	Specify a semaphore magic number to display detailed information on that semaphore. “magic” is a unique ID, used by the OS Awareness to identify a specific semaphore (address of the ATOM_SEM structure).

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



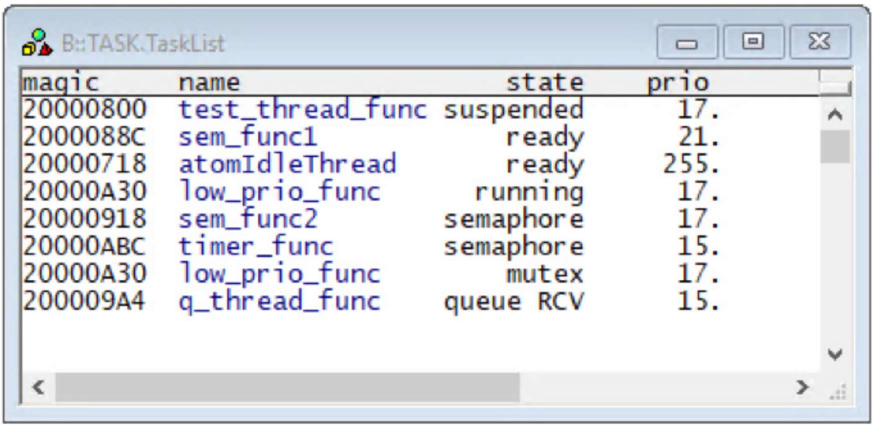
Right-clicking it will show a local menu.



Format:	**TASK.TaskList** [<*task_magic*>]

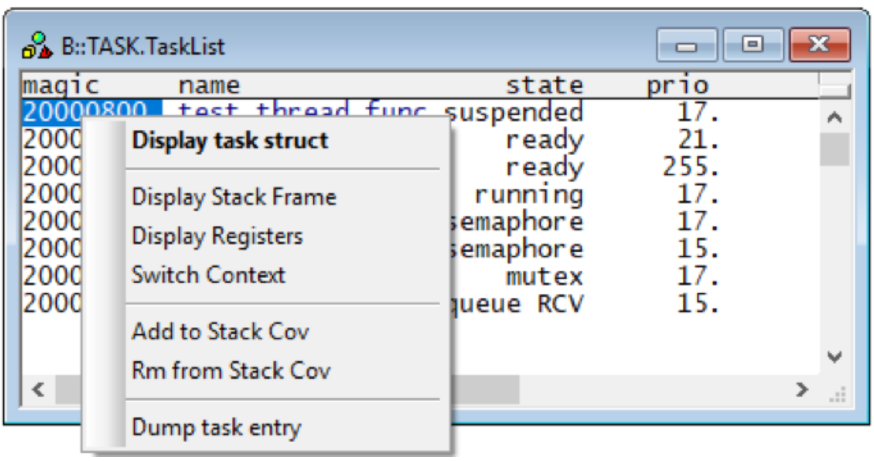
Displays the task table of Atomthreads or detailed information about one specific task.

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



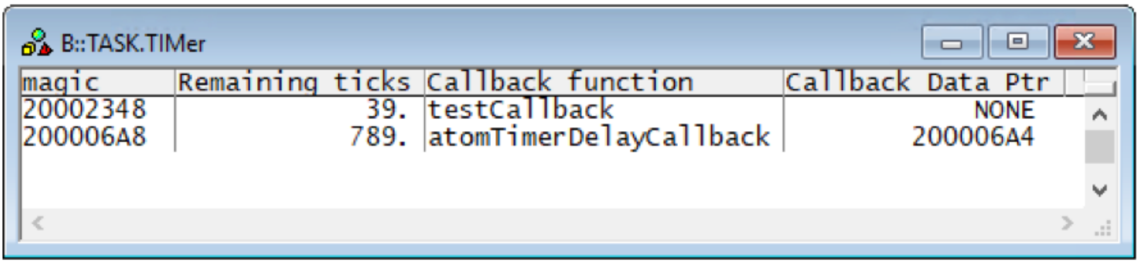
<*task_magic*>	Specify a task magic number to display detailed information on that task. "magic" is a unique ID, used by the OS Awareness to identify a specific task (address of the TCB).

The field "magic" is mouse sensitive, double-clicking it opens an appropriate window. Right-clicking it will show a local menu.



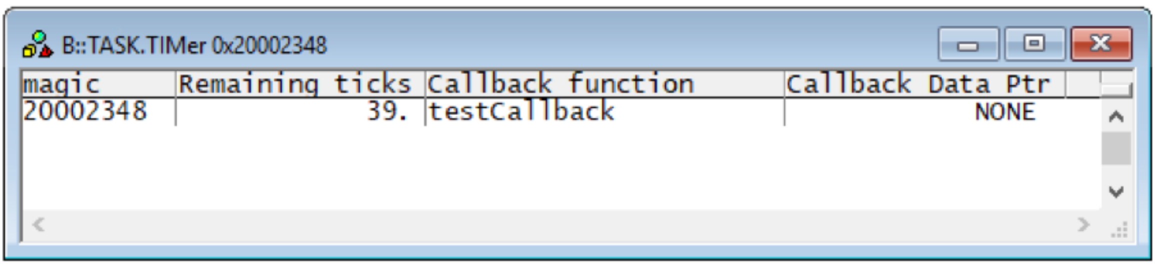
Format:	**TASK.TIMer** [*<timer_magic>*]

Displays a list of system timers or detailed information about one specific timer.

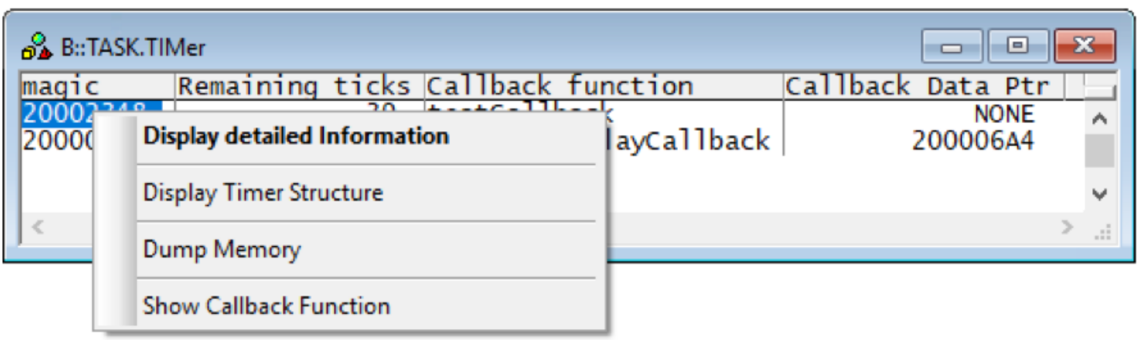


<timer_magic>	Specify a timer magic number to display detailed information on that timer. “magic” is a unique ID, used by the OS Awareness to identify a specific timer (address of the ATOM_TIMER structure).

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



Right-clicking it will show a local menu.



Atomthreads PRACTICE Functions

There are special definitions for Atomthreads 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). 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).
tcb	**Parameter Type**: String (***without*** quotation marks). Returns the name of the TCB structure.

Return Value Type: Hex value.

TASK.STRUCT() OS structure names

Syntax: **TASK.STRUCT(***<item>***)**

Reports OS structure names.

Parameter Type: String (***without*** quotation marks).

Return Value Type: String.

Appendix A: Context ID on Cortex-A Systems

For any kind of task analysis, TRACE32 needs to be able to trace task switches. This requires data trace, which is not available on Cortex-A based systems. They do provide a context ID register which can be updated by the RTOS when it makes a task switch. TRACE32 can use these generated packets to perform task switch based analyses. To take advantage of this, the RTOS must be modified to generate these packets on each task switch. A possible modification for Atomthreads is shown below. Modify the file `ports/armv7a/atomport.c`.

```
void archContextSwitch(ATOM_TCB *old_tcb, ATOM_TCB *new_tcb)
{
uint32_t tmp = 0x0, lr = 0x0;
pt_regs_t *old_regs = (pt_regs_t *)((uint32_t)old_tcb->sp_save_ptr
- sizeof(pt_regs_t));
pt_regs_t *new_regs = (pt_regs_t *)((uint32_t)new_tcb->sp_save_ptr
- sizeof(pt_regs_t));
asm volatile (" mov %0, lr\n\t" :"=r"(lr):);

/* TRACE32:
 * Added to provide context switches for trace based debugging
 */
asm volatile( " mcr p15, 0, %0, c13, c0, 1" :: "r" (new_tcb));

if (archSetJump(old_regs, &tmp)) {
old_regs->lr = lr;
archLongJump(new_regs);
}
}m
```