



OS Awareness Manual ThreadX

OS Awareness Manual ThreadX

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

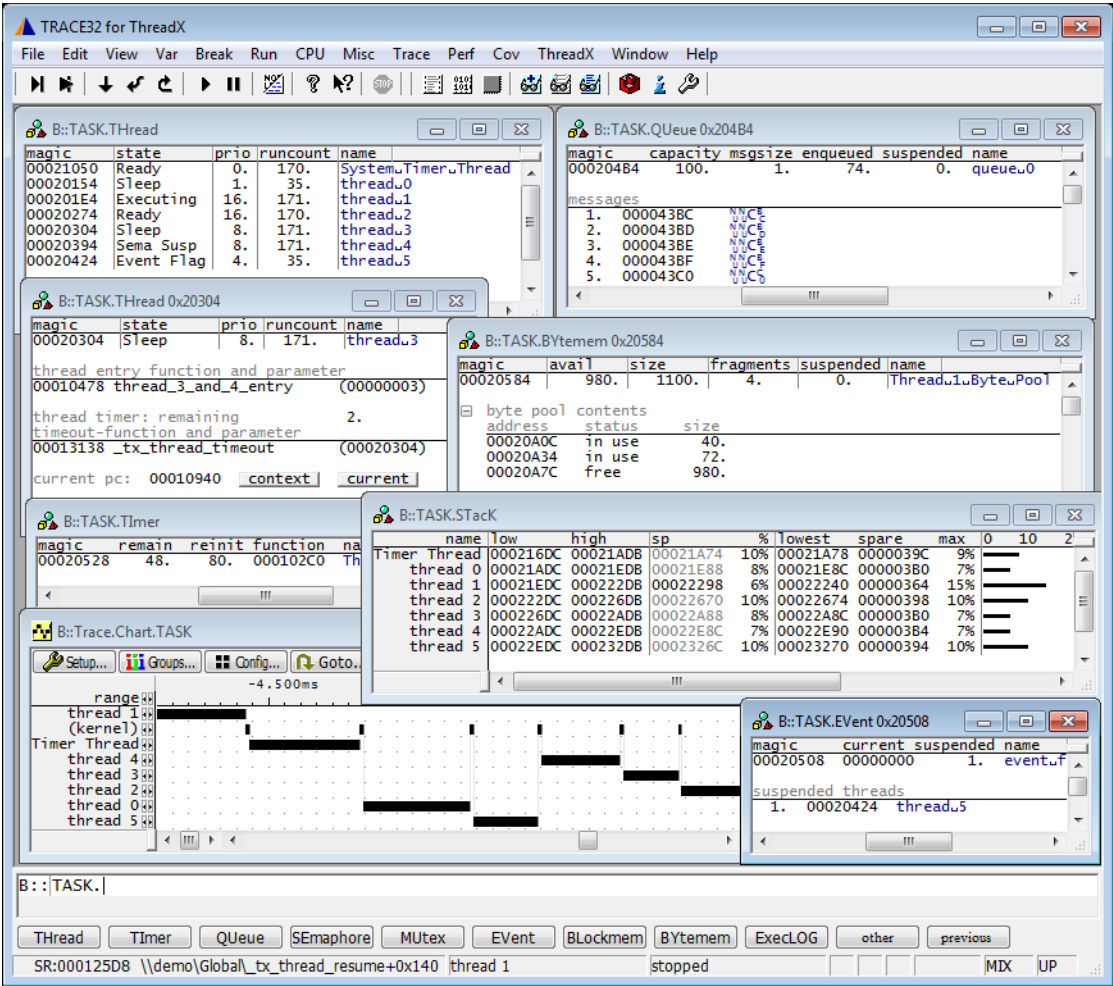
TRACE32 Documents	
OS Awareness Manuals	
OS Awareness Manual ThreadX	1
History	4
Overview	4
Terminology	5
Brief Overview of Documents for New Users	5
Supported Versions	5
Configuration	6
Quick Configuration Guide	7
Hooks & Internals in ThreadX	7
Features	8
Display of Kernel Resources	8
Task Stack Coverage	8
Task-Related Breakpoints	9
Task Context Display	10
SMP Support	11
Dynamic Task Performance Measurement	12
Task Runtime Statistics	12
Task State Analysis	13
Function Runtime Statistics	15
ThreadX specific Menu	17
ThreadX Commands	18
TASK.BLockmem	Display block memory pools 18
TASK.BYtemem	Display byte memory pools 18
TASK.EVent	Display event flags 19
TASK.ExecLOG	Display thread performance log 19
TASK.MUtex	Display mutexes 20
TASK.QUeue	Display queues 20
TASK.SEmaphore	Display semaphores 21
TASK.THread	Display threads 22
TASK.Tlmer	Display application timers 23
TASK.TRACE	Display event trace buffer 23

TASK.TRACEVM	Copy event trace buffer to LOGGER	23
ThreadX PRACTICE Functions		25
TASK.CONFIG()	OS Awareness configuration information	25
TASK.TH.MAGIC()	Magic number of thread	25
TASK.BY.MAGIC()	Magic number of byte pool	25
TASK.BL.MAGIC()	Magic number of block pool	26

History

04-Feb-21 Removing legacy command TASK.TASKState.

Overview



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

Terminology

Note the terminology: while ThreadX talks about “threads”, the OS Awareness uses the term “task”. They are used interchangeably in this context.

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 ThreadX is supported for the following versions:

- ThreadX 3.0, 4.0 and 5.x on ARC, ARM, ColdFire, MicroBlaze, MIPS32/64, Nios-II, PowerPC, SH4, StarCore, x86, XScale and XTENSA.

Configuration

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

Automatic configuration tries to locate the ThreadX 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 analyzer 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:

TASK.CONFIG threadx

See also the example “`~/demo/<processor>/kernel/threadx/threadx.cmm`”.

Quick Configuration Guide

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

1. Start the TRACE32 Debugger.
2. Load your application as normal.
3. Execute the command:

```
TASK.CONFIG ~/demo/<arch>/kernel/threadx/threadx.t32
```

See “[Configuration](#)”.

4. Execute the command:

```
MENU.ReProgram ~/demo/<arch>/kernel/threadx/threadx.men
```

See “[ThreadX Specific Menu](#)”.

5. Start your application.

Now you can access the ThreadX extensions through the menu.

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

Hooks & Internals in ThreadX

No hooks are used in the kernel.

To retrieve information on kernel objects, the OS Awareness uses the global ThreadX pointers exported by the ThreadX library, and the structures defined in the `tx_api.h` file. Be sure that your application is compiled and linked with debugging symbols switched on. The ThreadX library may be compiled without debugging symbols.

Features

The OS Awareness for ThreadX supports the following features.

Display of Kernel Resources

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

TASK.BLockmem	Block memory
TASK.BYtemem	Byte memory
TASK.EVent	Event flags
TASK.MUtex	Mutexes
TASK.QUeue	Queues
TASK.SEmaphore	Semaphores
TASK.THread	Threads
TASK.Tlmer	Timers

For a description of the commands, refer to chapter “**ThreadX 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
System Timer Thread	000058B4	00005CAF	00005C5C	8%	00005C5C	000003A8	8%		
thread 0	00007324	0000771F	000076CC	8%	000076CC	000003A8	8%		
thread 1	0000772C	00007B27	00007AF8	4%	00007ACC	000003A0	8%		
thread 2	00007B34	00007F2F	00007ECC	9%	00007ECC	00000398	9%		
thread 3	00007F3C	00008337	000082DC	8%	000082DC	000003A0	8%		
thread 4	00008344	0000873F	000086E4	8%	000086E4	000003A0	8%		
thread 5	0000874C	00008B47	00008AD4	11%	00008AD4	00000388	11%		
thread 6	00008B54	00008F4F	00008EF4	8%	00008EF4	000003A0	8%		
thread 7	00008F5C	00009357	000092FC	8%	000092FC	000003A0	8%		

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.

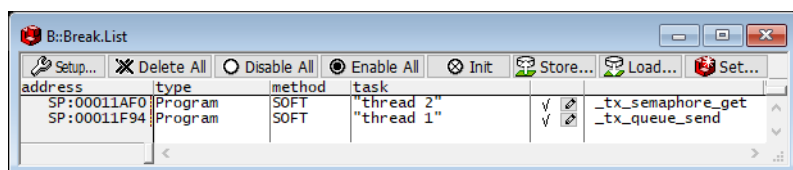
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

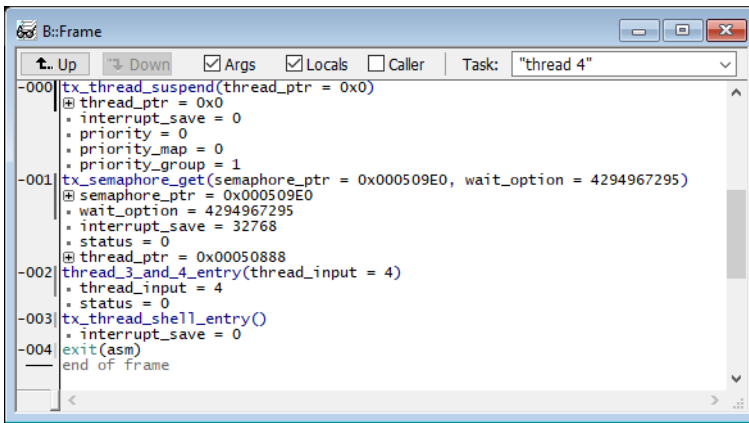
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.



```
B::Frame
Task: "thread 4"
-000| tx_thread_suspend(thread_ptr = 0x0)
    | @ thread_ptr = 0x0
    | * interrupt_save = 0
    | * priority = 0
    | * priority_map = 0
    | * priority_group = 1
-001| tx_semaphore_get(semaphore_ptr = 0x000509E0, wait_option = 4294967295)
    | @ semaphore_ptr = 0x000509E0
    | * wait_option = 4294967295
    | * interrupt_save = 32768
    | * status = 0
    | @ thread_ptr = 0x00050888
-002| thread_3_and_4_entry(thread_input = 4)
    | * thread_input = 4
    | * status = 0
-003| tx_thread_shell_entry()
    | * interrupt_save = 0
-004| exit(asm)
    | end of frame
```

SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application runs may change over time.

To support such SMP systems, the debugger allows a “system view”, where one TRACE32 PowerView GUI is used for the whole system, i.e. for all cores that are used by the SMP OS. For information about how to set up the debugger with SMP support, please refer to the [Processor Architecture Manuals](#).

All core relevant windows (e.g. [Register.view](#)) show the information of the current core. The [state line](#) of the debugger indicates the current core. You can switch the core view with the [CORE.select](#) command.

Target breaks, be they manual breaks or halting at a breakpoint, halt all cores synchronously. Similarly, a [Go](#) command starts all cores synchronously. When halting at a breakpoint, the debugger automatically switches the view to the core that hit the breakpoint.

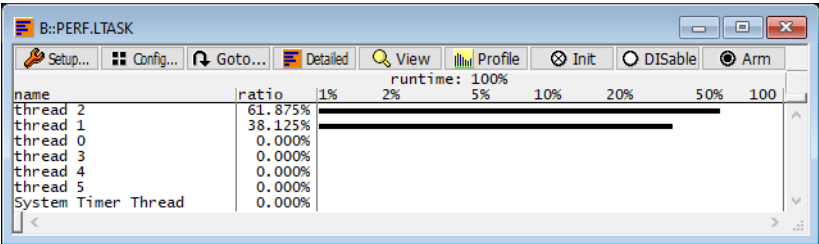
Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

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:

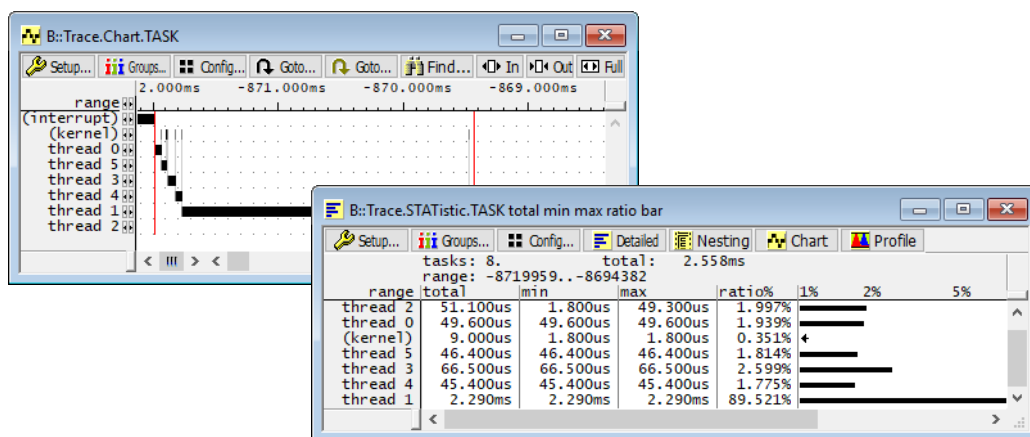
This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based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



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.

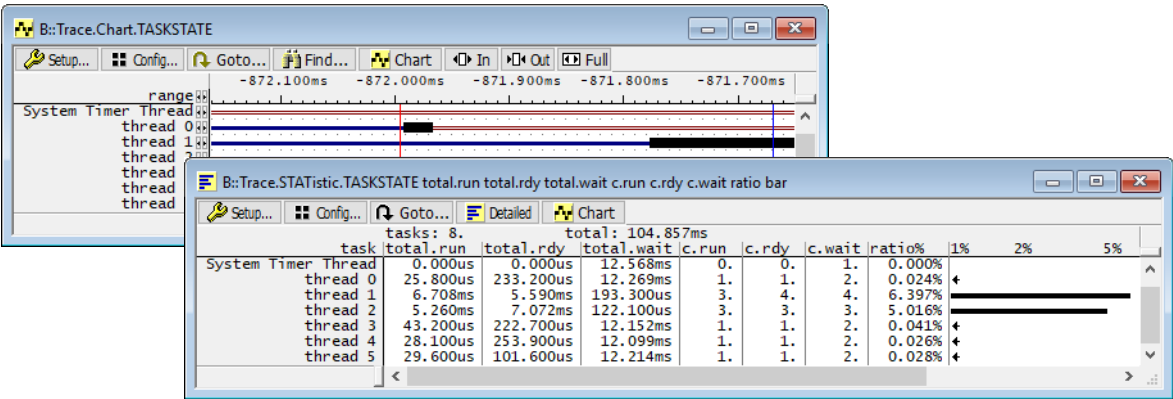
```
Break.Set Var.RANGE(TCB_array) /Write /TraceData
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

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)".

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



NOTE:

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

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

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

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

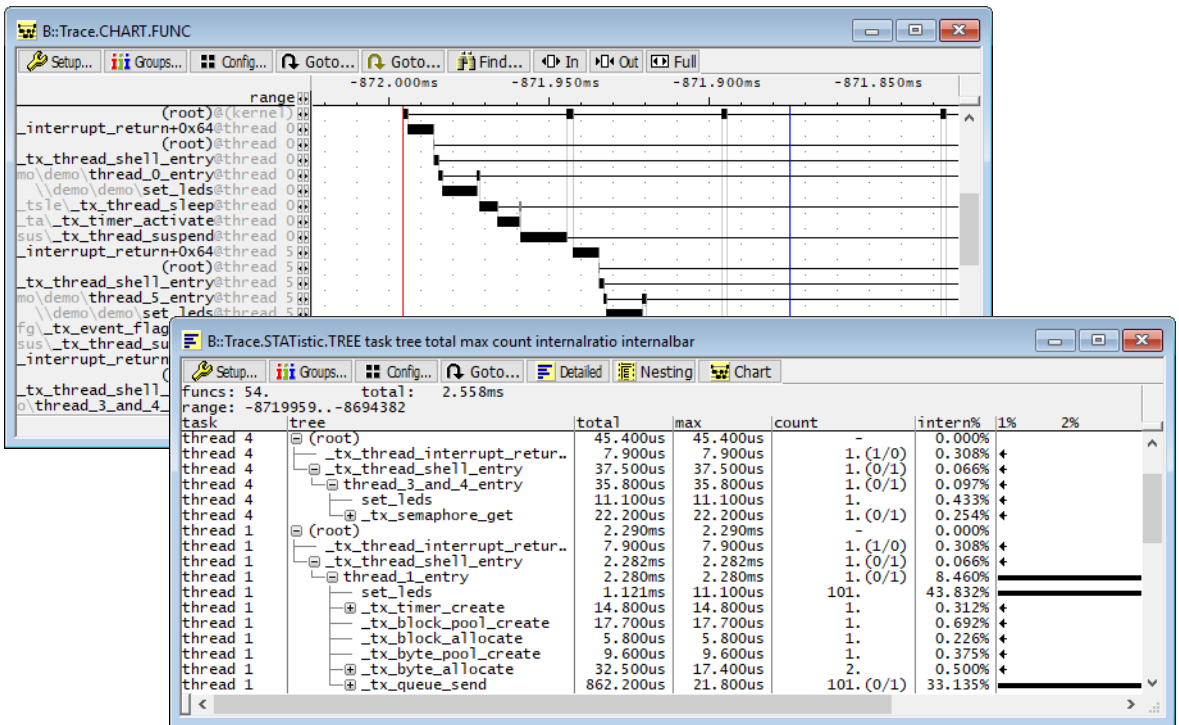
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

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

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

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

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



ThreadX specific Menu

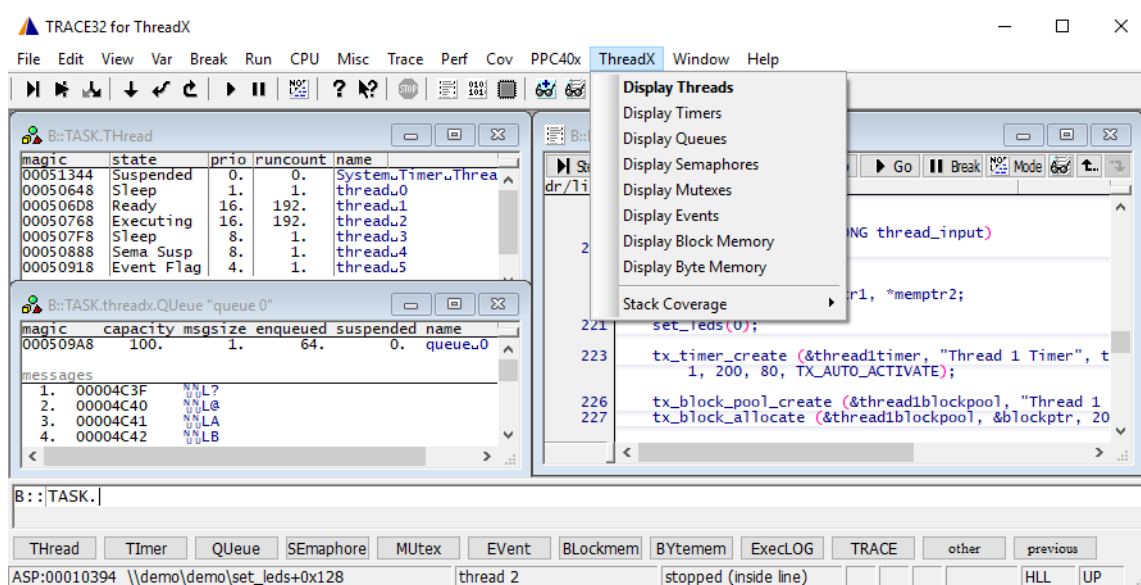
The menu file “threadx.men” contains a menu with ThreadX specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- The **Display** menu items launch the appropriate kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the ThreadX specific stack coverage, and provide an easy way to add or remove threads 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 thread switches (if any) or thread switches together with default display.
- The **Perf** menu contains additional submenus for thread runtime statistics, thread related function runtime statistics or statistics on task states.

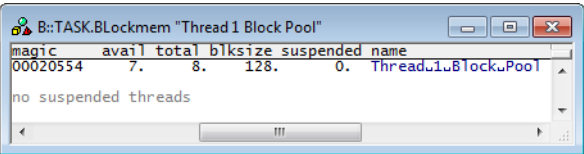


TASK.BLockmem

Display block memory pools

Format: **TASK.BLockmem** <blkpool>

Displays a table with the pools of the Block Memory. Specifying a magic number or pool name will show you detailed information about that pool.

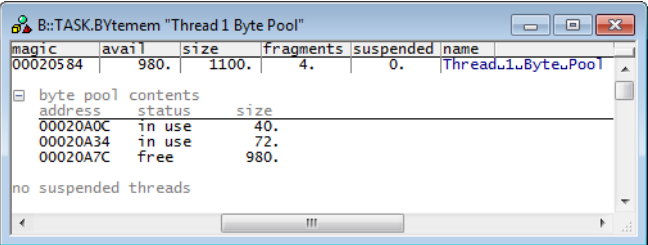


TASK.BYtemem

Display byte memory pools

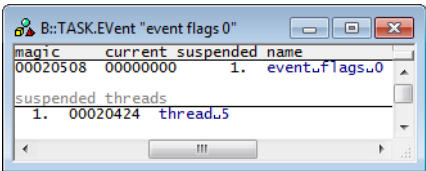
Format: **TASK.BYtemem** <bytpool>

Displays a table with the pools of the Byte Memory. Specifying a magic number or pool name will show you detailed information about that pool.



Format: **TASK.Event** <event>

Displays a table with the ThreadX event lag groups. Specifying an event flag magic number or name will show you the suspended threads of that event.



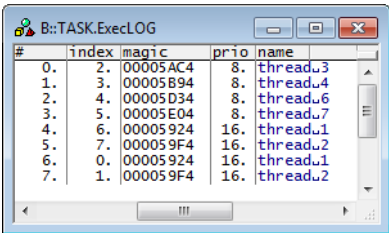
“magic” defines a unique ID which the OS Awareness uses for the event identification. The fields 'magic' and 'name' are mouse sensitive. Double-clicking on them will perform the appropriate action.

TASK.ExecLOG

Display thread performance log

Format: **TASK.ExecLOG**

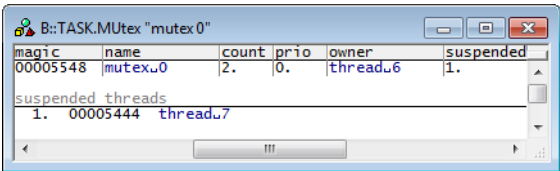
TASK.ExecLOG displays the kernel internal buffer of the thread performance information log.



ThreadX must be built with TX_THREAD_ENABLE_PERFORMANCE_INFO. See ThreadX documentation more information on this ThreadX feature.

Format: **TASK.SEmaphore** <sema>

Displays a table with the ThreadX mutexes. Specifying a mutex magic number or name will show you the suspended threads of that mutex.



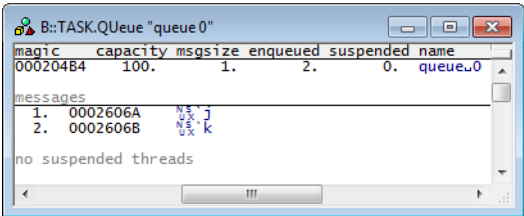
“magic” defines a unique ID which the OS Awareness uses for the mutex identification. The fields 'magic' and 'name' are mouse sensitive. Double-clicking on them will perform the appropriate action.

TASK.QUeue

Display queues

Format: **TASK.QUeue** <queue>

Displays a table with the ThreadX queues. Specifying a queue magic number or queue name will show you the messages in that queue and waiting threads.

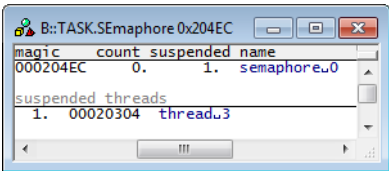


“magic” defines a unique ID which the OS Awareness uses for the queue identification. The fields 'magic' and 'name' are mouse sensitive. Double-clicking on them will perform the appropriate action.

Format:

TASK.Semaphore <sema>

Displays a table with the ThreadX semaphores. Specifying a semaphore magic number or name will show you the suspended threads of that semaphore.

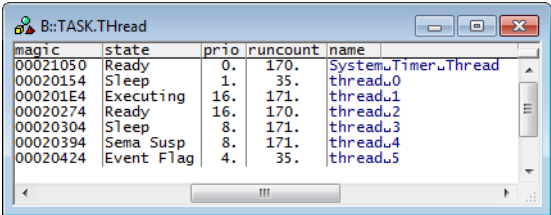


“magic” defines a unique ID which the OS Awareness uses for the semaphore identification. The fields 'magic' and 'name' are mouse sensitive. Double-clicking on them will perform the appropriate action.

Format: **TASK.THread** [<thread> [/SORTup | /SORTDOWN <sortitem>]]

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

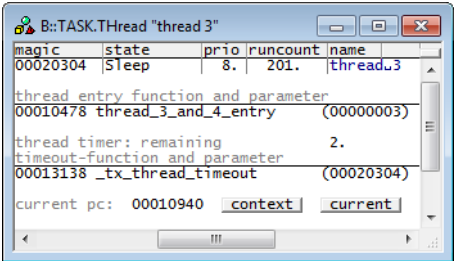
Without any arguments, a table with all created threads will be shown.



You can sort the window to the entries of a column by clicking on the column header. An initial sorting can be specified by using a comma as placeholder for “thread” and specifying the sort direction together with the sort item. Use **MAGIC**, **STATE**, **PRIO**, **RUNCOUNT** or **NAME** as sort item. Example:

TASK.THread , /SORTup NAME

To display detailed information on one thread, specify a thread name in quotes or a magic number to the command.



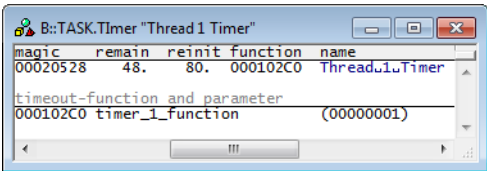
“magic” is a unique ID, used by the OS Awareness to identify a specific thread (address of the TCB).

The fields “magic”, “thread entry” and “timeout function” 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 thread. “current” resets it to the current context. See “[Thread Context Display](#)”.

Format: **TASK.Timer** <timer>

Displays a table with the ThreadX application timers. Specifying a timer magic number or timer name will show you more information on that timer.



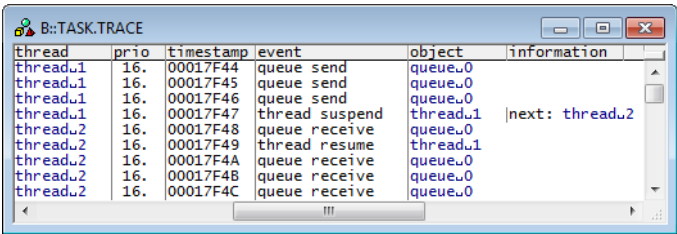
“magic” defines a unique ID which the OS Awareness uses for the timer identification. The fields “magic” “function” and “name” are mouse sensitive. Double-clicking on them will perform the appropriate action.

TASK.TRACE

Display event trace buffer

Format: **TASK.TRACE**

TASK.TRACE displays the kernel internal buffer of the event trace feature.



ThreadX must be built with TX_ENABLE_EVENT_TRACE. See ThreadX documentation more information on this ThreadX feature.

TASK.TRACEVM

Copy event trace buffer to LOGGER

Format: **TASK.TRACEVM**

TASK.TRACEVM copies the entries of the kernel internal event trace to a debugger-internal buffer in virtual memory (VM:), using the **LOGGER** structure layout.

ThreadX must be built with TX_ENABLE_EVENT_TRACE. See ThreadX documentation more information on this ThreadX feature.

Activate the LOGGER and copy the buffers with:

```
Trace.METHOD Logger
Logger.RESet
Logger.ADDRESS VM:0x1000
Logger.TimeStamp Up
Logger.TimeStamp.Rate 1000.
Logger.Init
TASK.TRACEVM
Logger.ARM
Logger.OFF
```

After this, you can use the Logger contents for [Task Runtime Statistics](#) and [Task State Analysis](#).

ThreadX PRACTICE Functions

There are special definitions for ThreadX specific PRACTICE functions.

TASK.CONFIG()

OS Awareness configuration information

Syntax:

TASK.CONFIG(magic | magicsize)

Parameter and Description:

magic	Parameter Type: String (<i>without</i> 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 (<i>without</i> quotation marks). Returns the size of the task magic number (1, 2 or 4).

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

TASK.TH.MAGIC()

Magic number of thread

Syntax:

TASK.TH.MAGIC(<thread_name>)

Returns the magic number of the given thread.

Parameter Type: String (*with* quotation marks).

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

TASK.BY.MAGIC()

Magic number of byte pool

Syntax:

TASK.BY.MAGIC(<pool_name>)

Returns the magic number of the given byte pool.

Parameter Type: String (*with* quotation marks).

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

Syntax: **TASK.BL.MAGIC(<pool_name>)**

Returns the magic number of the given block pool.

Parameter Type: [String](#) (*with* quotation marks).

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