

# OS Awareness Manual XOS

MANUAL

[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

[TRACE32 Documents](#) .....



[OS Awareness Manuals](#) .....



[OS Awareness Manual XOS](#) .....

1

[History](#) .....

3

[Overview](#) .....

4

    Terminology

4

    Brief Overview of Documents for New Users

4

    Supported Versions

4

[Configuration](#) .....

5

    Automatic Configuration

5

    Quick Configuration Guide

6

    Hooks & Internals in XOS

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

    Function Runtime Statistics

10

    XOS specific Menu

12

[XOS Commands](#) .....

13

    TASK.Thread

        Display threads

13

    TASK.TIMER

        Display timers

13

[XOS PRACTICE Functions](#) .....

14

    TASK.CONFIG()

        OS Awareness configuration information

14

## History

---

28-Aug-18      The title of the manual was changed from “RTOS Debugger for <x>” to “OS Awareness Manual <x>”.

# Overview

---

The OS Awareness for XOS 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 XOS 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 XOS is supported for the following versions:

- Version 1.10 on XTENSA.

# Configuration

---

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

Automatic configuration tries to locate the XOS 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).

## Automatic Configuration

---

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 xos.t32**

Automatic configuration

See also the example “`~/demo/<arch>/kernel/xos/xos.cmm`”.

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

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

Now you can access the XOS extensions through the menu.

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

## Hooks & Internals in XOS

---

No hooks are used in the kernel.

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.

The OS Awareness for XOS supports the following features.

## Display of Kernel Resources

---

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

<b>TASK.Thread</b>	Threads
<b>TASK.TIMER</b>	Timers

For a description of the commands, refer to chapter “[XOS 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.

# Task-Related Breakpoints

Any breakpoint set in the debugger can be restricted to fire only if a specific task hits that breakpoint. This is especially useful when debugging code which is shared between several tasks. To set a task-related breakpoint, use the command:

**Break.Set** <address>|<range> [/<option>] /TASK <task>      Set task-related breakpoint.

- Use a magic number, task ID, or task name for <task>. For information about the parameters, see [“What to know about the Task Parameters”](#) (general\_ref\_t.pdf).
- For a general description of the **Break.Set** command, please see its documentation.

By default, the task-related breakpoint will be implemented by a conditional breakpoint inside the debugger. This means that the target will *always* halt at that breakpoint, but the debugger immediately resumes execution if the current running task is not equal to the specified task.

**NOTE:**

Task-related breakpoints impact the real-time behavior of the application.

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

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

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

When single stepping, the debugger halts at the next instruction, regardless of which task hits this breakpoint. When debugging shared code, stepping over an OS function may cause a task switch and coming back to the same place - but with a different task. If you want to restrict debugging to the current task, you can set up the debugger with **SETUP.StepWithinTask** ON to use task-related breakpoints for single stepping. In this case, single stepping will always stay within the current task. Other tasks using the same code will not be halted on these breakpoints.

If you want to halt program execution as soon as a specific task is scheduled to run by the OS, you can use the **Break.SetTask** command.

# Dynamic Task Performance Measurement

---

The debugger can execute a dynamic performance measurement by evaluating the current running task in changing time intervals. Start the measurement with the commands **PERF.Mode TASK** and **PERF.Arm**, and view the contents with **PERF.ListTASK**. The evaluation is done by reading the ‘magic’ location (= current running task) in memory. This memory read may be non-intrusive or intrusive, depending on the **PERF.METHOD** used.

If **PERF** collects the PC for function profiling of processes in MMU-based operating systems (**SYStem.Option.MMUSPACES ON**), then you need to set **PERF.MMUSPACES**, too.

For a general description of the **PERF** command group, refer to “[General Commands Reference Guide P](#)” (general\_ref\_p.pdf).

## Task Runtime Statistics

---

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

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

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

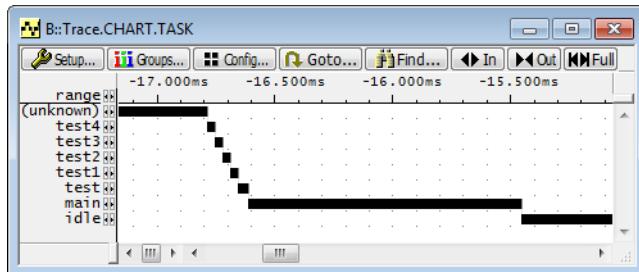
<b>Trace.List List.TASK DEFault</b>	Display trace buffer and task switches
<b>Trace.STATistic.TASK</b>	Display task runtime statistic evaluation
<b>Trace.Chart.TASK</b>	Display task runtime timechart
<b>Trace.PROfileSTATistic.TASK</b>	Display task runtime within fixed time intervals statistically
<b>Trace.PROfileChart.TASK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

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

B::Trace.STATistic.TASK

tasks: 8. total: 19.997ms

range	total	min	max	avr	count	ratio%	1%	2%
(unknown)	3.212ms	3.212ms	3.212ms	3.212ms	0.	16.062%		
test4	32.500us	32.500us	32.500us	32.500us	1.	0.162%	♦	
test3	32.500us	32.500us	32.500us	32.500us	1.	0.162%	♦	
test2	32.500us	32.500us	32.500us	32.500us	1.	0.162%	♦	
test1	32.500us	32.500us	32.500us	32.500us	1.	0.162%	♦	
test	44.300us	44.300us	44.300us	44.300us	1.	0.221%	♦	
main	1.196ms	1.158ms	1.158ms	597.850us	2.	5.979%		
idle	15.415ms	15.415ms	15.415ms	15.415ms	1.	77.086%		



## 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:

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

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

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

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

- The **Display** menu items launch the appropriate kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the XOS 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 the default display.
- The **Perf** menu contains additional submenus for thread runtime statistics, thread related function runtime statistics or statistics on task states.

## TASK.Thread

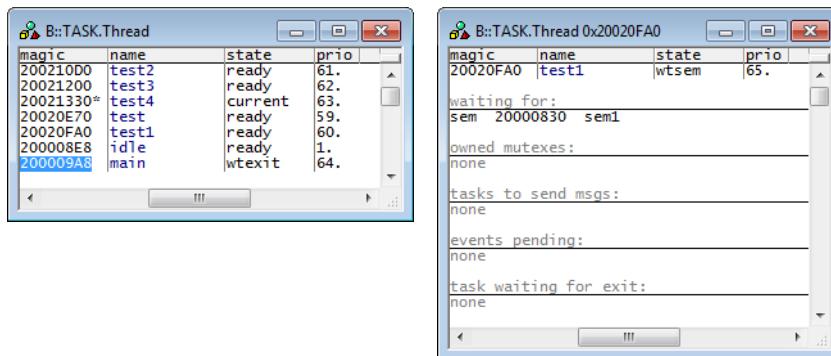
Display threads

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

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

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

Specify a thread name or magic number to display detailed information on that thread.



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

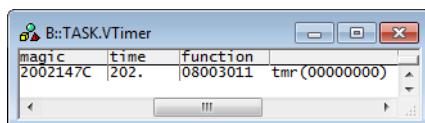
The field “magic” and other fields are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on them will show a local menu.

## TASK.TIMER

Display timers

Format: **TASK.TIMER**

Displays a list of all created virtual timers.



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

# XOS PRACTICE Functions

---

There are special definitions for XOS specific PRACTICE functions.

## **TASK.CONFIG()**

OS Awareness configuration information

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

**Parameter and Description:**

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

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