LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
OSE Epsilon

MANUAL

OS Awareness Manual OSE Epsilon

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

OS Awareness Manuals ..... 🗁

OS Awareness Manual OSE Epsilon **1**

History **3**

Overview **3**

Terminology 3

Brief Overview of Documents for New Users 4

Supported Versions 4

Configuration **5**

Quick Configuration Guide 6

Hooks & Internals in OSE Epsilon 6

Features **7**

Display of Kernel Resources 7

Task Runtime Statistics 7

Task State Analysis 8

Function Runtime Statistics 9

Task Stack Coverage 10

OSE Epsilon specific Menu 12

OSE Epsilon Commands **13**

TASK.DProc Display processes 13

TASK.DQueue Display signal queue 13

OSE Epsilon PRACTICE Functions **15**

TASK.CONFIG() OS Awareness configuration information 15

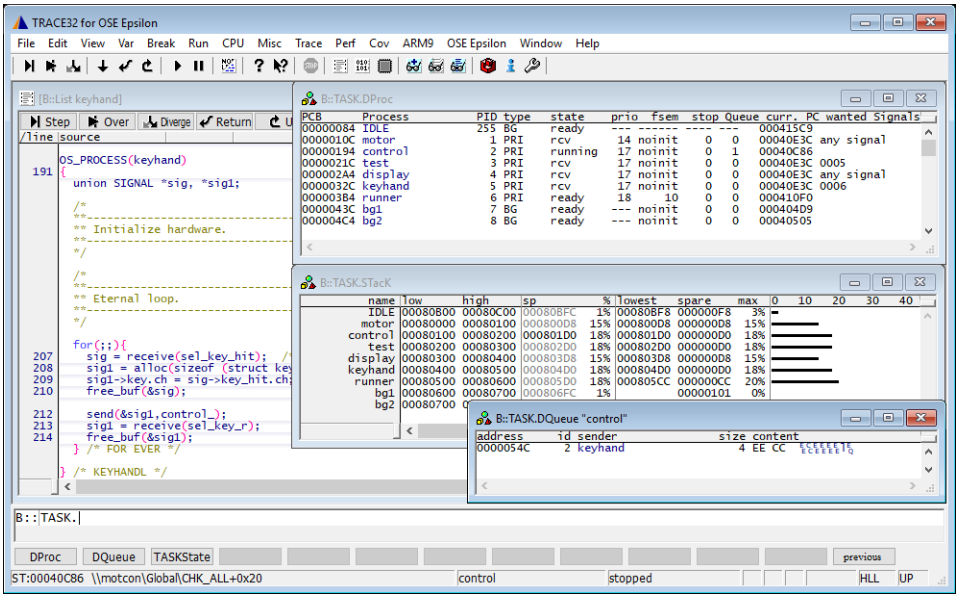
OS Awareness Manual OSE Epsilon

Version 06-Jun-2024

History

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

Overview



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

Terminology

OSE Epsilon uses the term “process”. If not otherwise specified, the TRACE32 term “task” corresponds to OSE Epsilon process.

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 **OSE Epsilon** is supported for the following versions:

- OS166 on C167 with small or large memory model.
- OSARM on ARM7.

Configuration

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

Automatic configuration tries to locate the OSE Epsilon 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 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 ose**<*proc*>

Examples:

```
TASK.CONFIG osearmE                    ; OSE Epsilon awareness for AR

TASK.CONFIG oseC16x                    ; OSE Epsilon awareness for C16x
```

See also the example “~~/demo/<*processor*>/kernel/oseepsilon/osee.cmm”

Quick Configuration Guide

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

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

Now you can access the OSE Epsilon extensions through the menu.

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

Hooks & Internals in OSE Epsilon

No hooks are used in the kernel.

For detecting the current running task, the kernel symbol "ZZ_CUR_PCB" is used.

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols. Ensure that access to those symbols is possible every time when features of the OS Awareness are used.

Features

The OS Awareness for OSE Epsilon supports the following features.

Display of Kernel Resources

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

TASK.DProc	Processes
TASK.DQueue	Signal queues

For a description of the commands, refer to chapter “OSE Epsilon 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 OSE Epsilon holds its tables.

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

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).

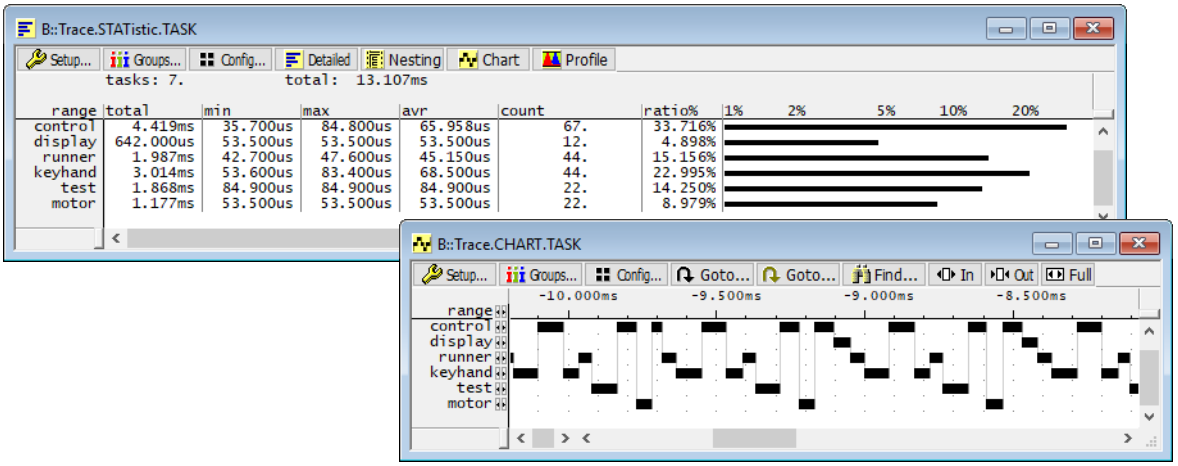
Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

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).

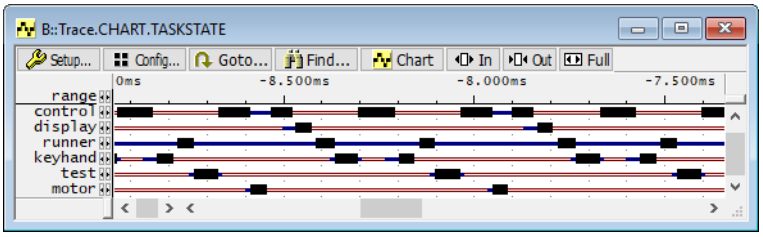
Example: This script assumes that the TCBs are located in an array named TCB_array and consequently limits the tracing to data write accesses on the TCBs and the task switch.

```
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)".



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
```

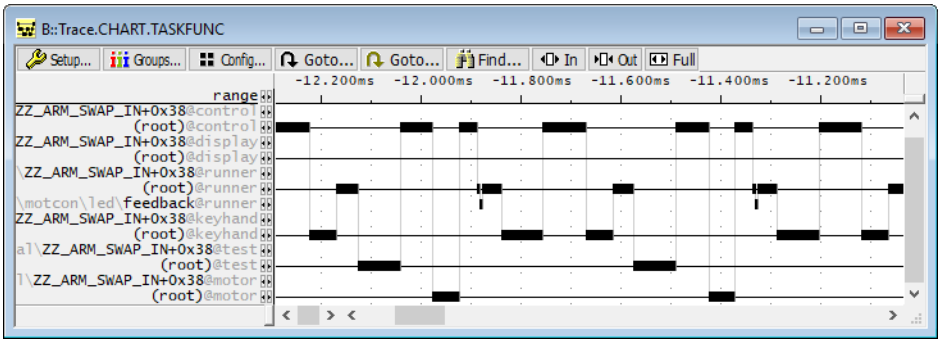
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



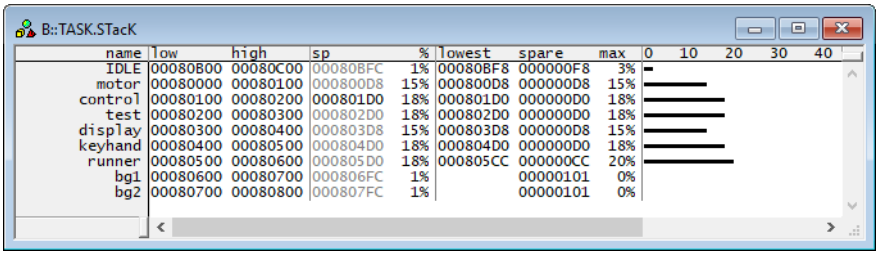
Task Stack Coverage

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.

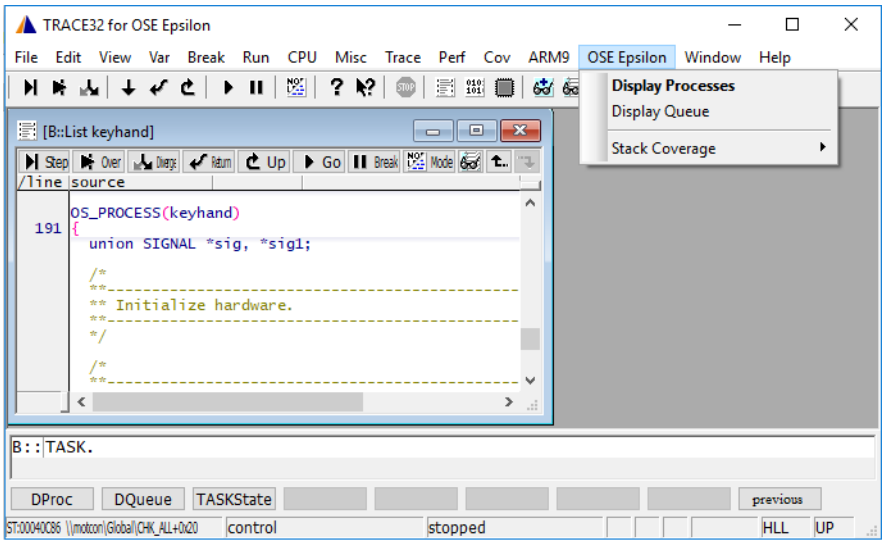
It is recommended to display only the tasks you are interested in because the evaluation of the used stack space is very time consuming and slows down the debugger display.


B::TASK.STacK
name low high sp % lowest spare max 0 10 20 30 40
IDLE 00080B00 00080C00 00080BFC 1% 00080BF8 000000F8 3%
motor 00080000 00080100 000800D8 15% 000800D8 000000D8 15%
control 00080100 00080200 000801D0 18% 000801D0 000000D0 18%
test 00080200 00080300 000802D0 18% 000802D0 000000D0 18%
display 00080300 00080400 000803D8 15% 000803D8 000000D8 15%
keyhand 00080400 00080500 000804D0 18% 000804D0 000000D0 18%
runner 00080500 00080600 000805D0 18% 000805CC 000000CC 20%
bg1 00080600 00080700 000806FC 1% 00000101 0%
bg2 00080700 00080800 000807FC 1% 00000101 0%

OSE Epsilon specific Menu

The menu file “osee.men” contains a menu with OSE Epsilon specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **OSE Epsilon**.



- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the OSE Epsilon specific stack coverage and provides an easy way to add or remove processes from the stack coverage window.

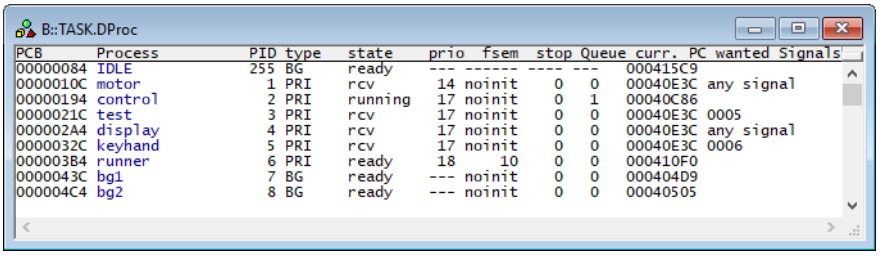
In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- 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 process runtime statistics, process related function runtime statistics or statistics on process states.

TASK.DProc — Display processes

Format: **TASK.DProc**

Displays the process table of OS166 and OSARM.



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

The **state** column shows the current state of each process.

The **in_q** column shows the number of current signals in the process queue. The **sigwait** column shows the signal on which the process is waiting.

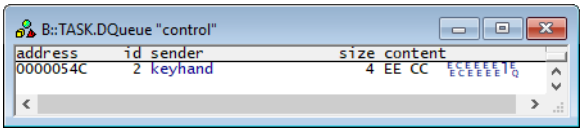
Double-clicking on the magic or on a number of the **in_q** column opens a separate **TASK.DQueue** window, showing a detailed list of the signals in queue of that process

NOTE for OSI166: While running in real time, the state '**running**' cannot be detected and is displayed as '**ready**'.

TASK.DQueue — Display signal queue

Format: **TASK.DQueue** <*process*>

Displays the signal queue table of the specified process.
Specify the process by its magic number or by its name.



Double click on the address to get the signal structure displayed.

Signal Structure Display

When double-clicking on the address of a signal inside the TASK.DQueue window, the Debugger tries to display the signal as "`union ALL_SIGNALS`" over all signal structures. The perl script sigdb.pl (available in ~~/demo/*<processor>*/kernel/oseepsilon) generates this union automatically and writes it into a file called all_sig.c. This C file must be linked into your application. In the directory, where all your signal source files are located, type: "`sigdb.pl *.sig >all_sig.c`". The perl script needs a perl interpreter installed on your host.

OSE Epsilon PRACTICE Functions

There are special definitions for OSE Epsilon 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.