OS Awareness Manual OSEK/ORTI

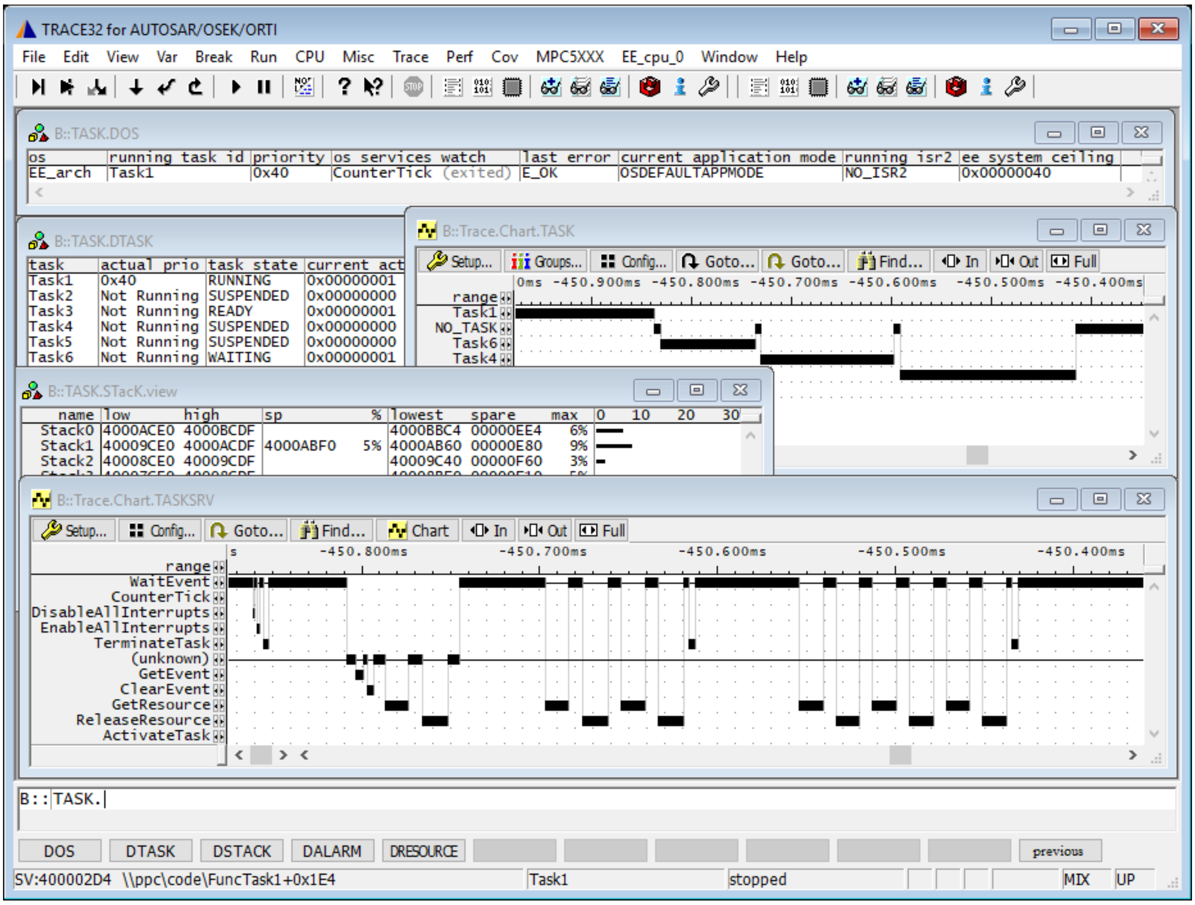
Release 02.2022

OS Awareness Manual OSEK/ORTI

OS Awareness Manual OSEK/ORTI

Version 09-Mar-2022

Overview


TRACE32 for AUTOSAR/OSEK/ORTI
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX EE_cpu_0 Window Help
B::TASK.DOS
B::TASK.DTASK
B::Trace.Chart.TASK
Setup... Groups... Config... Goto... Goto... Find... In Out Full
B::TASK.STacK.view
B::Trace.Chart.TASKSRV
Setup... Config... Goto... Find... Chart In Out Full
DOS DTASK DSTACK DALARM DRESOURCE previous

```
os       running task id priority os services watch     last error current application mode running isr2 ee system ceiling
EE_arch  Task1           0x40     CounterTick (exited)  E_OK       OSDEFAULTAPPMODE         NO_ISR2      0x00000040
```

```
task   actual prio task state current act
Task1  0x40        RUNNING    0x00000001
Task2  Not Running SUSPENDED  0x00000000
Task3  Not Running READY      0x00000001
Task4  Not Running SUSPENDED  0x00000000
Task5  Not Running SUSPENDED  0x00000000
Task6  Not Running WAITING    0x00000001
```

```
  name  low      high     sp        % lowest   spare    max
Stack0  4000ACE0 4000BCDF             4000BBC4 00000EE4  6%
Stack1  40009CE0 4000ACDF 4000ABF0 5% 4000AB60 00000E80  9%
Stack2  40008CE0 40009CDF             40009C40 00000F60  3%
```

```
0ms -450.900ms -450.800ms -450.700ms -450.600ms -450.500ms -450.400ms
range
Task1
NO_TASK
Task6
Task4
```

```
s -450.800ms -450.700ms -450.600ms -450.500ms -450.400ms
range
WaitEvent
CounterTick
DisableAllInterrupts
EnableAllInterrupts
TerminateTask
(unknown)
GetEvent
ClearEvent
GetResource
ReleaseResource
ActivateTask
```

```
B::TASK.
```

```
SV:400002D4 \\ppc\code\FuncTask1+0x1E4   Task1   stopped   MIX UP
```

The OSEK Run Time Interface (ORTI) is a specification, which enables debuggers to become OS-aware, without knowing the OS itself. Most AUTOSAR/OSEK system builders are able to extract all necessary information of the OS component into a text file, called “ORTI file”.

The TRACE32 Debugger can load such an ORTI file and adds some special extensions to itself. This manual describes the additional features, such as additional commands and statistic evaluations.

Note that only those extensions are available, which are specified in the ORTI file. I.e. there may be only a subset of the described features available, depending on your ORTI file.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Training - Debugger Basics”** (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 ORTI is supported for the following versions:

- ORTI V2.0 (no official version)
- ORTI V2.1, V2.2
- ORTI V2.3 (inofficial SMP and OTM extensions)

Configuration

The following command is used to load the ORTI file and to extend the capabilities of the debugger according to this file:

TASK.ORTI *<orti_file>*	Load ORTI file.

The AUTOSAR/OSEK system builder generates an ORTI file, when generating the OS (usually with the extension ".ort" or ".orti"). Specify this file to the TASK.ORTI command.

The ORTI file refers to global symbols exported by the OS. Ensure that those symbols are loaded and accessible while executing **TASK.ORTI** and while using the ORTI extensions. Usually the compiled application contains these symbols.

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).

See also the example "~~/demo/kernel/orti/orti.cmm".

Quick Configuration Guide

To get a quick access to the features of the ORTI debugger with your application, follow this roadmap:

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

```
TASK.ORTI <orti_file>
```

   See "**Configuration**".

4. Start your application.

Now you can access the ORTI extensions through the TRACE32 menu.

In case of any problems, please carefully read the previous configuration chapters and check the **requirements**.

Hooks and Internals in ORTI

There are some requirements to do a successful debugging and tracing with ORTI files. In case of problems, please check carefully these items.

Requirements for Debugging

Please note these requirements for debugging:

- The module to be debugged must be compiled and linked with debug information.
  If possible, compile the OS with debug information, too.
- It is recommended (but not mandatory) to switch off compiler optimization for easier debugging.
  But be aware that this affects the run time behavior.
- The AUTOSAR/OSEK system builder must create an ORTI file.
  This file is system specific - it must be rebuilt as soon as the application layout is changed.
- The "`RUNNINGTASK`" attribute of the "`OS`" object should evaluate to a single address location

  OK: `RUNNINGTASK = "OS_taskCurrent";`
  Not OK: `RUNNINGTASK = "OS_taskCurrent->runPrio";`

  Single address is needed by the debugger to get the current running task by reading a single memory location. E.g. used to display the current task in the status line, provide task-specific breakpoints, calculate dynamic task performance, detect task switches in the real time trace, and various other things.

  Contact the OS vendor or Lauterbach if this requirement is not met.
- The "`RUNNINGTASK`" attribute of the "`OS`" object should be declared as "`ENUM`"

  OK:
  ```
  ENUM ["NO_TASK" = 0xff, "InitTask" = 0, "Cyclic" = 1, "Loop" = 2]
           RUNNINGTASK, "Running task";
  ```
  Not OK: `CTYPE RUNNINGTASK, "Running task";`

  ENUM is used by the debugger to know all the possible tasks in the system.

  Contact the OS vendor or Lauterbach if this requirement is not met.

Requirements for Tracing

Tracing with ORTI requires that the on-chip trace generation logic can generated task information. For details refer to **"OS-aware Tracing"** (glossary.pdf).

Additional information on task switch packets: Either the OS writes the current task ID into the ownership trace register, or a PreTaskHook must be created and registered to write the task ID. Example for a PreTaskHook for PowerPCs with diab compiler:

```
asm volatile void send_OTM (TaskType value)
{
%reg value
!
    mtpid0 value
    isync
}

void PreTaskHook (void)
{
    TaskType taskid;
    GetTaskID(&taskid);
    send_OTM(taskid);
}
```

For recording various objects (tasks, ISRs, services) and/or ids that do not fit in a single ownership trace message, use a complex OTM protocol. Ask Lauterbach for the specification of the “ORTI OTM Proposal”.

Debug Features

The OS Awareness for OSEK/ORTI supports the following debug features.

Display of Kernel Resources

The ORTI awareness defines new PRACTICE commands to display various kernel resources. The Information available depends on the objects defined in the ORTI file. The commands for displaying those information has the form “TASK.D*<object>*”. For a detailed description please refer to the chapter “**ORTI 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

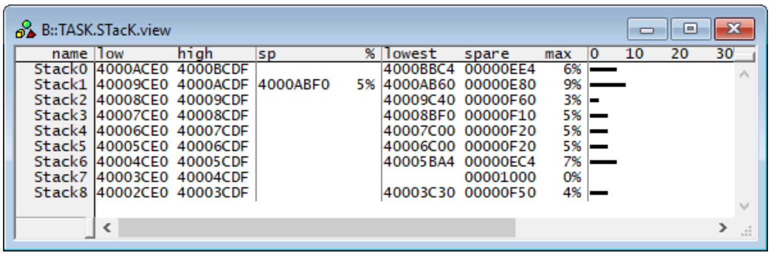
For stack usage coverage of OSEK stacks, you can use the **TASK.STacK** command. This command will set up a window with all defined OSEK stacks.

This feature is only available, if the ORTI file presents the stack characteristics of each task as defined in the ORTI specification.

To use the calculation of the maximum stack usage, flag memory must be mapped to the stack areas, when working with the emulation memory. When working with the target memory, the debugger uses the stack pattern given in the ORTI file. If the ORTI file does not provide a stack pattern, it must be defined with the command **TASK.STacK.PATtern** (default value is zero).

To add/remove one stack to/from the stack coverage, you can call the **TASK.STacK.ADD** rsp. **TASK.STacK.ReMove** commands and select from the list window.

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



```
  name  low       high      sp            % lowest    spare     max  0   10   20   30
Stack0  4000ACE0  4000BCDF                  4000BBC4  00000EE4   6%
Stack1  40009CE0  4000ACDF  4000ABF0     5% 4000AB60  00000E80   9%
Stack2  40008CE0  40009CDF                  40009C40  00000F60   3%
Stack3  40007CE0  40008CDF                  40008BF0  00000F10   5%
Stack4  40006CE0  40007CDF                  40007C00  00000F20   5%
Stack5  40005CE0  40006CDF                  40006C00  00000F20   5%
Stack6  40004CE0  40005CDF                  40005BA4  00000EC4   7%
Stack7  40003CE0  40004CDF                            00001000   0%
Stack8  40002CE0  40003CDF                  40003C30  00000F50   4%
```

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.

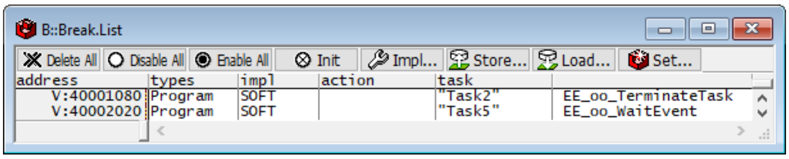
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering 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     action     task
  V:40001080 Program   SOFT                "Task2"     EE_oo_TerminateTask
  V:40002020 Program   SOFT                "Task5"     EE_oo_WaitEvent
```

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.

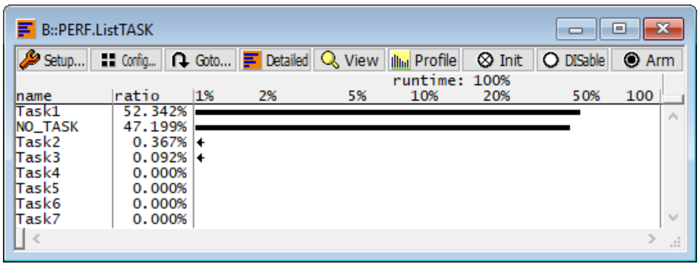
NOTE:	The evaluation of the task context is only available, if the loaded ORTI file provides information about the contexts.

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
Task1       52.342%
NO_TASK     47.199%
Task2        0.367%
Task3        0.092%
Task4        0.000%
Task5        0.000%
Task6        0.000%
Task7        0.000%
```

OSEK/ORTI specific Menu

When loading the ORTI file with the command **TASK.ORTI**, the debugger automatically extends the menus, depending on the contents of the ORTI file.

You will find a new menu, named to the OSEK OS name.

- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu (if available) starts and resets the OSEK stack coverage and provides an easy way to add or remove stacks from the stack coverage window.

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

- The **Trace**, **List** menu is extended.
  - "Task Switches" shows a trace list window with only task switches (if any)
  - "Default and Tasks" shows switches together with the default display.
- The **Perf** menu contains additional submenus
  - "Task Runtime" enables and shows the **task runtime analysis**
  - "Task Function Runtime" enables and shows the **function runtime statistics** based on tasks
  - "Task State" enables shows the **task state analysis**
  - "CPU Load" enables and shows the **CPU load analysis**

Trace Features

The OS Awareness for OSEK/ORTI supports the following trace features.

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)”.

For further requirements for tracing, see also “**Hooks & Internals**”.

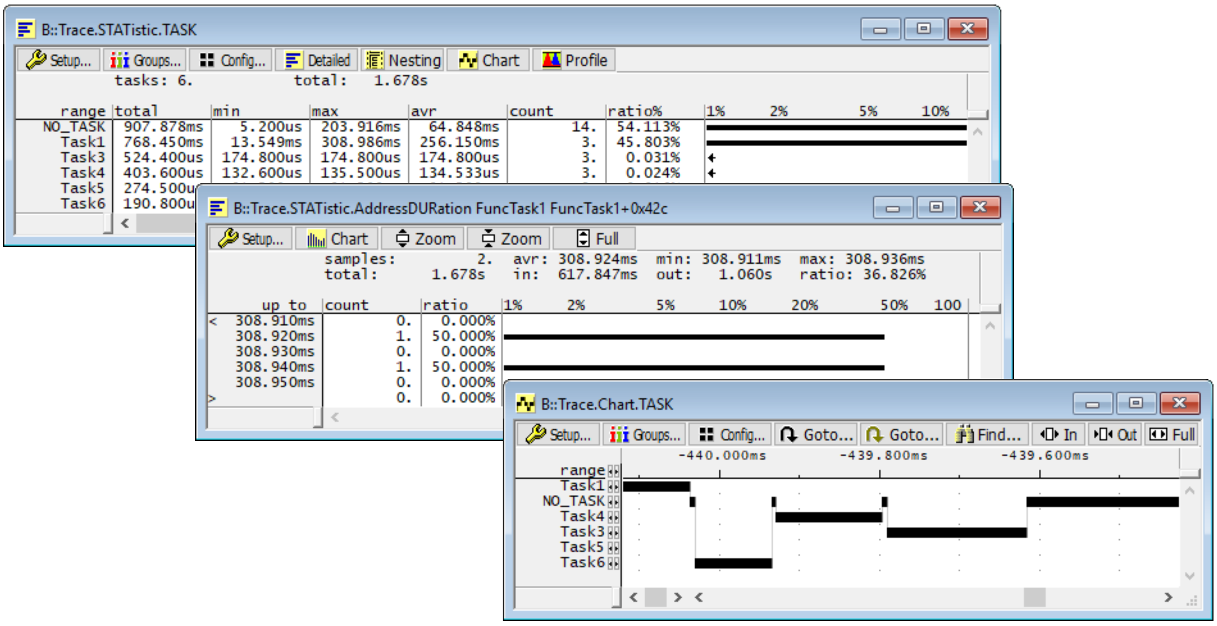
Hints on evaluating the function run times:

To display the time periods of calling timed tasks or timed runnables, use the body function of the tasks or runnables with **Trace.STATistic.AddressDIStance**. E.g:

```
Trace.STATistic.AddressDistance Func_Task10ms
```

To display the gross runtimes of a function use **Trace.STATistic.AddressDURation** with the start and exit address of the function. Please note that sYmbol.EXIT() won't work for task body functions that end with TerminateTask(), as this function may be called by other tasks in between. Instead, set the second argument to the calling line of TerminateTask(). Also, the evaluation of recursive functions are not possible with this command. Examples:

```
Trace.STATistic.AddressDURation myRunnable sYmbol.EXIT(myRunnable)
Trace.STATistic.AddressDURation myTaskFunc myTaskFunc\15
```



```
B::Trace.STATistic.TASK
Setup... Groups... Config... Detailed Nesting Chart Profile
             tasks: 6.             total:    1.678s

    range |total     |min       |max       |avr       |count     |ratio%    |1%     2%       5%     10%
  NO_TASK | 907.878ms|   5.200us|203.916ms |  64.848ms|       14.|  54.113% |
    Task1 | 768.450ms|  13.549ms|308.986ms |256.150ms |        3.|  45.803% |
    Task3 | 524.400us| 174.800us|174.800us |174.800us |        3.|   0.031% |
    Task4 | 403.600us| 132.600us|135.500us |134.533us |        3.|   0.024% |
    Task5 | 274.500u
    Task6 | 190.800u
```

```
B::Trace.STATistic.AddressDURation FuncTask1 FuncTask1+0x42c
Setup... Chart Zoom Zoom Full
             samples:         2.   avr: 308.924ms   min: 308.911ms   max: 308.936ms
             total:       1.678s   in:  617.847ms   out:    1.060s   ratio: 36.826%

       up to |count       |ratio  |1%     2%       5%     10%     20%      50%   100
<  308.910ms |          0.|  0.000%|
   308.920ms |          1.| 50.000%|
   308.930ms |          0.|  0.000%|
   308.940ms |          1.| 50.000%|
   308.950ms |          0.|  0.000%|
>            |          0.|  0.000%|
```

```
B::Trace.Chart.TASK
Setup... Groups... Config... Goto... Goto... Find... In Out Full
                 -440.000ms          -439.800ms          -439.600ms
    range
    Task1
  NO_TASK
    Task4
    Task3
    Task5
    Task6
```

Task State Analysis

NOTE: This feature is **only** available, if your debugger equipment is able to trace memory data accesses (flow trace is not sufficient).

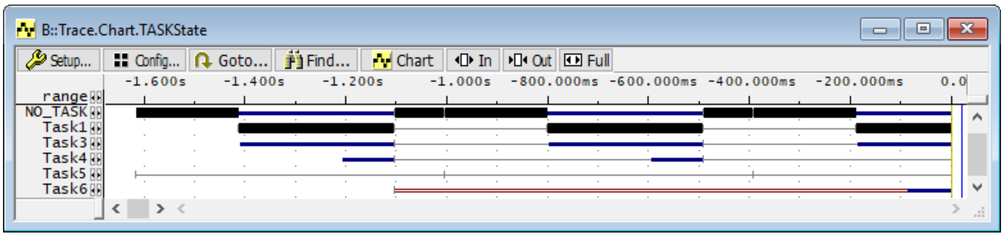
The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically. This feature is implemented by evaluating all accesses to the status words of all tasks. Additionally the accesses to the current task pointer (=magic) are evaluated.

A complete recording of all data accesses is necessary for this feature.

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state time chart

All kernel activities up to the task switch are added to the calling task. The start of the recording time, when the calculation doesn't know, which task is running, is calculated as "(root)".



Service Runtime Statistics

The time spent in an OSEK service routine can be evaluated statistically and displayed graphically. To do this, a memory location, that marks service routine entries and exits, must be recorded (that means, recording all services).

To do a selective recording on OSEK service routines with flow traces (ICD, e.g. ETM and NEXUS trace), based on the data accesses, use the following PRACTICE command:

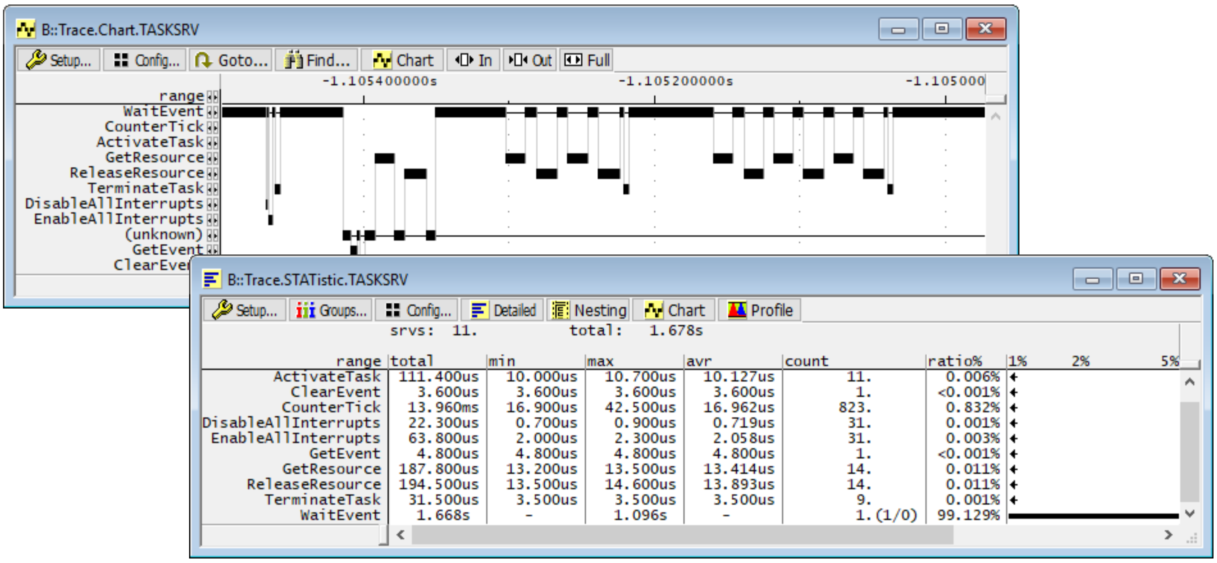
```
; Enable tracing only on the magic location
Break.Set TASK.CONFIG(magic_service) /TraceEnable
```

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

Trace.List List.TASK	Display service nesting
Trace.STATistic.TASKSRV	Display service runtime statistic
Trace.Chart.TASKSRV	Display service time chart
Trace.PROfileSTATistic.TASKSRV	Display service runtime within fixed time intervals statistically
Trace.PROfileChart.TASKSRV	Display service runtime within fixed time intervals as colored graph

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

For further requirements for tracing, see also “**Hooks & Internals**”.



```
srvs:  11.          total:   1.678s
```

range	total	min	max	avr	count	ratio%
ActivateTask	111.400us	10.000us	10.700us	10.127us	11.	0.006%
ClearEvent	3.600us	3.600us	3.600us	3.600us	1.	<0.001%
CounterTick	13.960ms	16.900us	42.500us	16.962us	823.	0.832%
DisableAllInterrupts	22.300us	0.700us	0.900us	0.719us	31.	0.001%
EnableAllInterrupts	63.800us	2.000us	2.300us	2.058us	31.	0.003%
GetEvent	4.800us	4.800us	4.800us	4.800us	1.	<0.001%
GetResource	187.800us	13.200us	13.500us	13.414us	14.	0.011%
ReleaseResource	194.500us	13.500us	14.600us	13.893us	14.	0.011%
TerminateTask	31.500us	3.500us	3.500us	3.500us	9.	0.001%
WaitEvent	1.668s	-	1.096s	-	1.(1/0)	99.129%

ISR2 Runtime Statistics

The time spent in an OSEK interrupt service routine (ISR2) can be evaluated statistically and displayed graphically. To do this, a memory location, that marks ISR2 routine entries and exits, must be recorded (that means, recording all interrupt service routines).

To do a selective recording on OSEK interrupt service routines with flow traces (ICD, e.g. ETM and NEXUS trace), based on the data accesses, use the following PRACTICE command:

```
; Enable tracing only on the magic location
Break.Set TASK.CONFIG(magic_isr2) /TraceEnable
```

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

Trace.List List.TASK	Display ISR2 entries and exits
Trace.STATistic.TASKINTR	Display ISR2 runtime statistic
Trace.Chart.TASKINTR	Display ISR2 time chart
Trace.PROfileSTATistic.TASKINTR	Display ISR2 runtime within fixed time intervals statistically
Trace.PROfileChart.TASKINTR	Display ISR2 runtime within fixed time intervals as colored graph
Trace.STATistic.TASKVSINTR	Display ISR2 runtime statistics against task runtimes

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

For further requirements for tracing, see also "**Hooks & Internals**".

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:

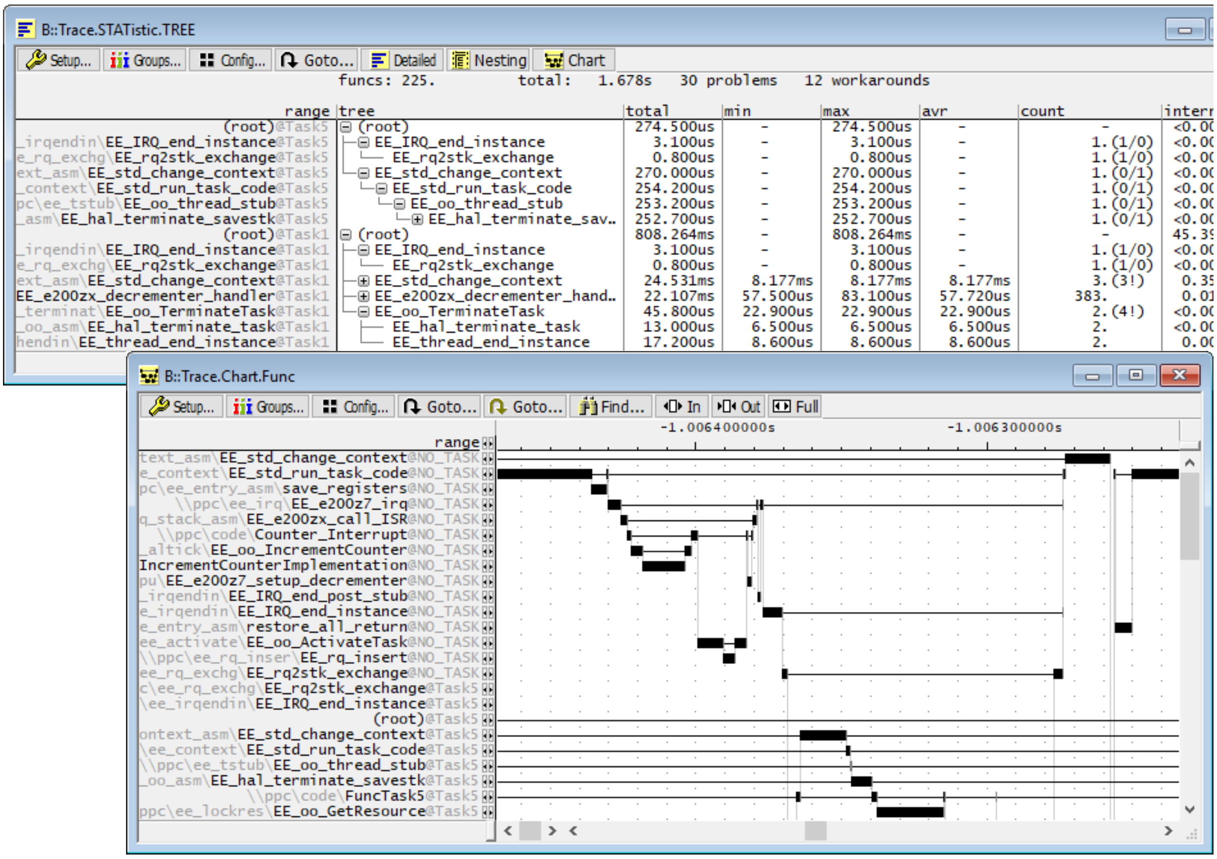
```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

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)".

For further requirements for tracing, see also “**Hooks & Internals**”.



```
                    funcs: 225.      total:   1.678s   30 problems   12 workarounds

                          range  tree                             total       min         max         avr         count         interr
                  (root)@Task5   ⊟ (root)                         274.500us   -           274.500us   -                 -       <0.00
_irqendin\EE_IRQ_end_instance@Task5  ├⊟ EE_IRQ_end_instance       3.100us   -             3.100us   -           1.(1/0)       <0.00
e_rq_exchg\EE_rq2stk_exchange@Task5  │ └── EE_rq2stk_exchange       0.800us   -             0.800us   -           1.(1/0)       <0.00
ext_asm\EE_std_change_context@Task5  └⊟ EE_std_change_context     270.000us   -           270.000us   -           1.(0/1)       <0.00
_context\EE_std_run_task_code@Task5    └⊟ EE_std_run_task_code    254.200us   -           254.200us   -           1.(0/1)       <0.00
pc\ee_tstub\EE_oo_thread_stub@Task5      └⊟ EE_oo_thread_stub     253.200us   -           253.200us   -           1.(0/1)       <0.00
_asm\EE_hal_terminate_savestk@Task5        └⊞ EE_hal_terminate_sav..  252.700us   -       252.700us   -           1.(0/1)       <0.00
                  (root)@Task1   ⊟ (root)                         808.264ms   -           808.264ms   -                 -       45.3
_irqendin\EE_IRQ_end_instance@Task1  ├⊟ EE_IRQ_end_instance       3.100us   -             3.100us   -           1.(1/0)       <0.00
e_rq_exchg\EE_rq2stk_exchange@Task1  │ └── EE_rq2stk_exchange       0.800us   -             0.800us   -           1.(1/0)       <0.00
ext_asm\EE_std_change_context@Task1  ├⊞ EE_std_change_context      24.531ms    8.177ms     8.177ms    8.177ms    3.(3!)         0.35
EE_e200zx_decrementer_handler@Task1  ├⊞ EE_e200zx_decrementer_hand..  22.107ms  57.500us   83.100us   57.720us  383.             0.01
_terminat\EE_oo_TerminateTask@Task1  └⊟ EE_oo_TerminateTask        45.800us   22.900us    22.900us   22.900us   2.(4!)        <0.00
_oo_asm\EE_hal_terminate_task@Task1    ├── EE_hal_terminate_task    13.000us    6.500us     6.500us    6.500us   2.            <0.00
hendin\EE_thread_end_instance@Task1    └── EE_thread_end_instance   17.200us    8.600us     8.600us    8.600us   2.             0.00
```

```
                                               range     -1.006400000s                 -1.006300000s
text_asm\EE_std_change_context@NO_TASK
e_context\EE_std_run_task_code@NO_TASK
pc\ee_entry_asm\save_registers@NO_TASK
  \\ppc\ee_irq\EE_e200z7_irq@NO_TASK
q_stack_asm\EE_e200zx_call_ISR@NO_TASK
  \\ppc\code\Counter_Interrupt@NO_TASK
_altick\EE_oo_IncrementCounter@NO_TASK
IncrementCounterImplementation@NO_TASK
pu\EE_e200z7_setup_decrementer@NO_TASK
_irqendin\EE_IRQ_end_post_stub@NO_TASK
e_irqendin\EE_IRQ_end_instance@NO_TASK
e_entry_asm\restore_all_return@NO_TASK
ee_activate\EE_oo_ActivateTask@NO_TASK
\\ppc\ee_rq_inser\EE_rq_insert@NO_TASK
ee_rq_exchg\EE_rq2stk_exchange@NO_TASK
c\ee_rq_exchg\EE_rq2stk_exchange@Task5
\ee_irqendin\EE_IRQ_end_instance@Task5
                          (root)@Task5
ontext_asm\EE_std_change_context@Task5
\ee_context\EE_std_run_task_code@Task5
\\ppc\ee_tstub\EE_oo_thread_stub@Task5
_oo_asm\EE_hal_terminate_savestk@Task5
               \\ppc\code\FuncTask5@Task5
ppc\ee_lockres\EE_oo_GetResource@Task5
```

CPU Load Analysis

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 CPU load. The CPU load is calculated by comparing the time spent in all tasks against the time spent in the idle task. The measurement is done by using the **GROUP** command to group all idle tasks and calculating the time spent in all other tasks.

Example: Two idle tasks named “IdleTask1” and “IdleTask2”:

```
; Create a group called "idle" with the idle tasks
GROUP.CreateTASK "idle" "IdleTask1"
GROUP.CreateTASK "idle" "IdleTask2"

; Unmark “idle” and mark all others in red
GROUP.COLOR "idle"  NONE
GROUP.COLOR "other" RED

; Merge idle tasks and other tasks
GROUP.MERGE "idle"
GROUP.MERGE "other"
```

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

Trace.STATistic.TASK	Display CPU load statistic evaluation
Trace.PROfileChart.TASK	Display CPU load as colored graph

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

When CPU load analysis is no longer needed, or if a detailed **Task Runtime Statistic** is needed, disable the grouping of the tasks with:

```
;comments
GROUP.SEParate "idle"
GROUP.SEParate "other"
```

TASK.D<object>

Display OSEK objects

Format: **TASK.D***<object>*

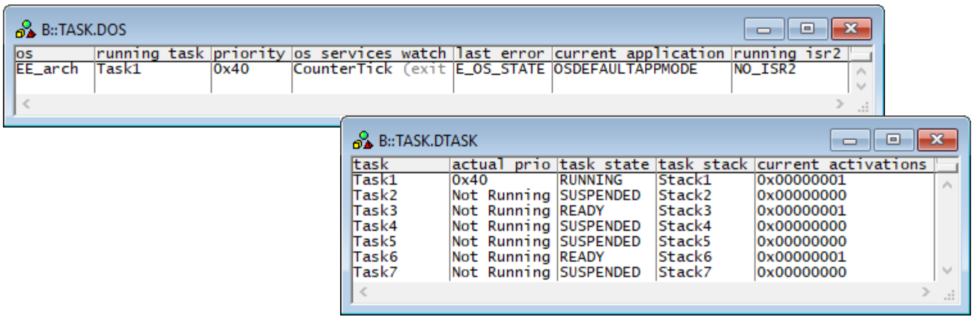
Each object type, defined in the ORTI file, gets its own command to display those objects.

<object>	Extend the command with the object type name.

Usually ORTI defines, among others, an “OS” object type and a “TASK” object type. Display the OS information and the task table, using the commands:

```
TASK.DOS
TASK.DTASK
```

Some of the table entries may contain links. Double click on them to follow them.


B::TASK.DOS

os	running task	priority	os services watch	last error	current application	running isr2
EE_arch	Task1	0x40	CounterTick (exit	E_OS_STATE	OSDEFAULTAPPMODE	NO_ISR2

B::TASK.DTASK

task	actual prio	task state	task stack	current activations
Task1	0x40	RUNNING	Stack1	0x00000001
Task2	Not Running	SUSPENDED	Stack2	0x00000000
Task3	Not Running	READY	Stack3	0x00000001
Task4	Not Running	SUSPENDED	Stack4	0x00000000
Task5	Not Running	SUSPENDED	Stack5	0x00000000
Task6	Not Running	READY	Stack6	0x00000001
Task7	Not Running	SUSPENDED	Stack7	0x00000000

ORTI PRACTICE Functions

There are special definitions for OSEK/ORTI specific PRACTICE functions.

TASK.CONFIG()

OS Awareness configuration information

Syntax 1:	**TASK.CONFIG(***<keyword>***)** (Single core instance)
Syntax 2:	**TASK.CONFIG(***<keyword>***[***<logical_core>***])**
Syntax 3:	**TASK.CONFIG(***<keyword>***:***<logical_core>***)**
<keyword>:	**magic** | **magicsize** | **magic_service** | **magic_isr2**
<logical_ core>:	**0**…*n*

Returns information about the OS Awareness configuration depending on the *<keywords>* described below.

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).
magic_service	**Parameter Type**: String (***without*** quotation marks). Returns the address of the service trace variable.
magic_isr2	**Parameter Type**: String (***without*** quotation marks). Returns the address of the isr2 trace variable.
<logical_core>	**Parameter Type**: • Decimal (***without*** the trailing dot in this particular case) or • Hex value (the preceding **0x** is mandatory in this particular case) In SMP systems, each of these keywords can have an optional core number (starting with zero) in square brackets (**[]**), or separated by a colon (**:**). See examples below.

Return Value Type: Hex value.

Example 1: This script returns the address of the variable that holds the current task in a single core instance.

```
PRINT TASK.CONFIG(magic)
```

Example 2: These script lines show how to return the address of the variable that holds the current task of the 2nd, 13th, and 19th core.

```
;parameter type decimal:
;omit the trailing dot in this particular case
PRINT TASK.CONFIG(magic[1])     ;2nd logical core
PRINT TASK.CONFIG(magic[12]     ;13th logical core

// or

PRINT TASK.CONFIG(magic:1)      ;2nd logical core
PRINT TASK.CONFIG(magic:12)     ;13th logical core
```

```
;parameter type hex:
;you must prefix the value with 0x in this particular case
PRINT TASK.CONFIG(magic[0x1])   ;2nd logical core
PRINT TASK.CONFIG(magic[0x12]   ;19th logical core

// or

PRINT TASK.CONFIG(magic:0x1)    ;2nd logical core
PRINT TASK.CONFIG(magic:0x12)   ;19th logical core
```

TASK.ORTI.ADDRESS() — Address of ORTI attribute

Syntax: **TASK.ORTI.ADDRESS (***<object>***.***<attribute>***)**

Returns the address of the specified ORTI attribute.

Parameter Type: String.

Return Value Type: Address.

Example:

```
PRINT TASK.ORTI.ADDRESS(os.runningtask)
```

TASK.ORTI.RANGE() — Address range of ORTI attributes

Syntax: **TASK.ORTI.RANGE(***<object>***.***<attribute>*[|…]**)**

Returns the address range occupied by the ORTI attributes. You can specify more than one attribute by concatenating them with a pipe “|” character (see example 2 below).

Parameter Type: String.

Return Value Type: Address range.

Example 1:

```
PRINT TASK.ORTI.RANGE(task.state)
```

Example 2:

```
PRINT TASK.ORTI.RANGE(task.state|os.runningtask)
```