OS Awareness Manual MicroC/OS-II

Release 09.2023

OS Awareness Manual MicroC/OS-II

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

OS Awareness Manuals ....

OS Awareness Manual MicroC/OS-II **1**

History **4**

Overview **5**

Brief Overview of Documents for New Users 6

Supported Versions 6

Configuration **7**

Manual Configuration 7

Automatic Configuration 9

Quick Configuration Guide 9

Hooks & Internals in µC/OS-II 9

Features **11**

Display of Kernel Resources 11

Task Stack Coverage 11

Task-Related Breakpoints 12

Task Context Display 13

Dynamic Task Performance Measurement 13

Task Runtime Statistics 14

Task State Analysis 15

Function Runtime Statistics 16

µC/OS-II specific Menu 17

µC/OS-II Commands **18**

TASK.Event Display events 18

TASK.Flag Display flags 18

TASK.Memory Display memory partitions 19

TASK.PARtition Display space partitions 19

TASK.PROCess Display MPU processes 20

TASK.Task Display tasks 20

TASK.TImer Display timers 21

µC/OS-II PRACTICE Functions **22**

TASK.CONFIG() OS Awareness configuration information 22

TASK.PAR.AVAIL() Space partitions 22

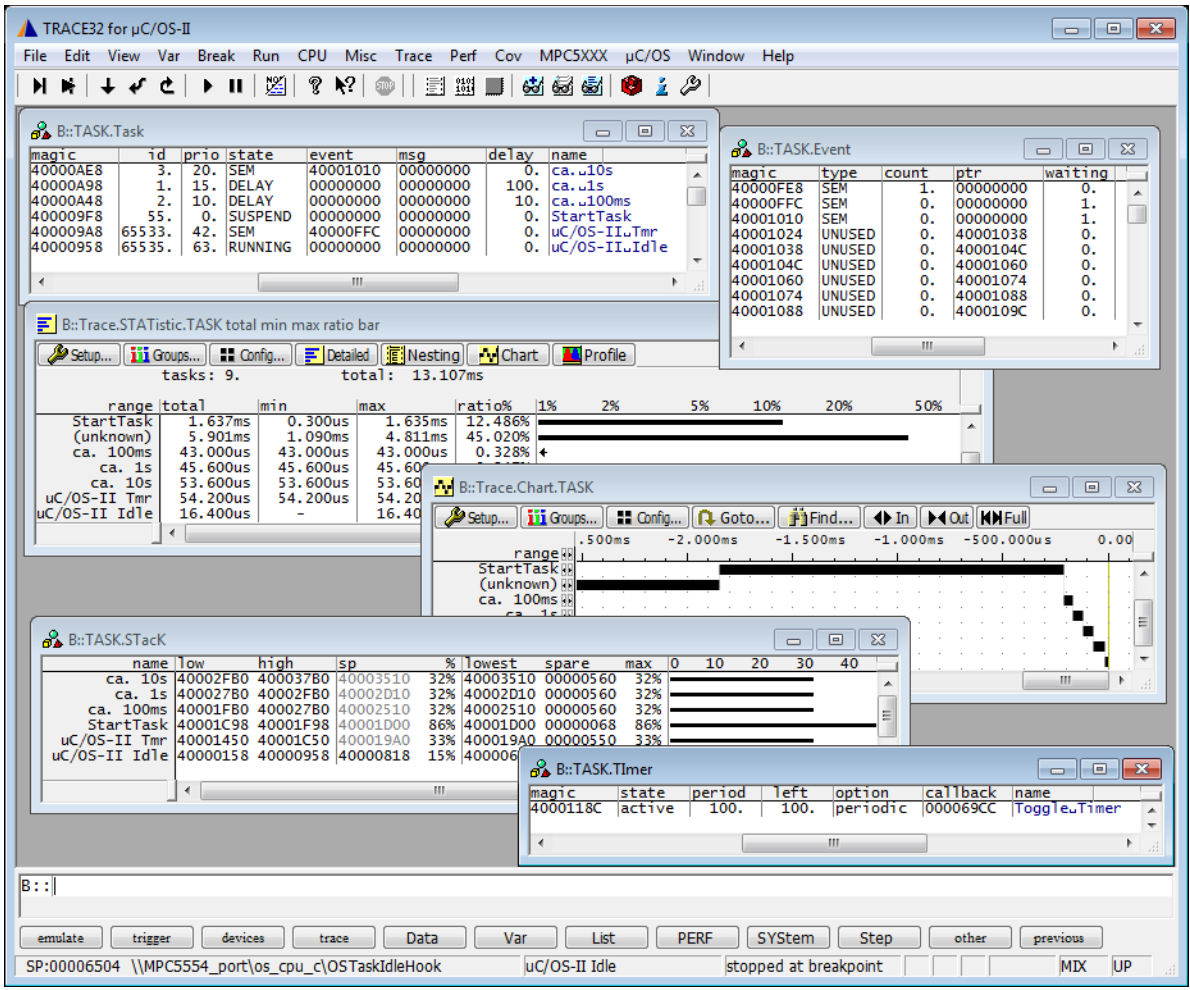
TASK.PROC.AVAIL() MPU processes 22
TASK.STRUCT() Symbol type name of the TCB structure 23

Version 09-Oct-2023

History

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

Overview



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

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 μC/OS-II is supported for the following versions:

- μC/OS-II V2.5 to 2.93 on Andes, ARC, ARM, Blackfin, C16x, C2xxx, C6xxx, ColdFire, HC08/HC12, MIPS, NiosII, PowerPC, RISC-V, TriCore, V850 and Xtensa.
- μC/OS-II V2.92 with partitioning on PowerPC
- μC/OS-II wV2.92 with MPU and certification on ARM

Configuration

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

Automatic configuration tries to locate the µC/OS-II internals automatically. For this purpose all symbol tables must be loaded and accessible at any time the OS Awareness is used.

If a system symbol is not available or if another address should be used for a specific system variable, then the corresponding argument must be set manually with the appropriate address. This can be done by manual configuration which can require some additional arguments.

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).

Manual Configuration

Manual configuration for the µC/OS OS Awareness can be used to explicitly define some memory locations. It is recommended to use automatic configuration.

Format:	**TASK.CONFIG ucos** *<magic_address> <args>*

<magic_address>	Specifies a memory location that contains the current running task. This address can be found at “`OSTCBCur`”. Specify 0 to automatically search for this symbol.
<args>	*<task_name_indirection> <task_name_offset>* See below for details.

Since µC/OS-II version 2.60, it supports task names. If you’d like to have task names, use the OS internal mechanisms for this purpose. The OS Awareness for µC/OS-II detects those task names automatically.

For versions before 2.60, you can implement task names in a way, that is supported by the OS Awareness for µC/OS:

`OS_TASK_CREATE_EXT_EN` must be defined to enable task names, and `OSTCBExtPtr` must point to the TCB extension of the task.

There are two possibilities to configure task names:

1. The TCB Extension contains the name itself.

   Specify “1” for the task name indirection. Specify additionally the offset, where the first character of the task name can be found in the TCB extension. E.g.

```
OSTCBExtPtr points to     OSTCBExtPtr points to     OSTCBExtPtr points
struct                    struct                    directly to name

struct {                  struct {                  char name[] =
     INT16U someval;           char   name[8];      ''Task1'';
     char   name[8];           INT16U someval;
     INT16U someval2;          INT16U someval2;
}                         }

task.config 0 1 2         task.config 0 1 0         task.config 0 1 0
```

2. The TCB Extension contains a pointer to the task name.

   Specify “2” for the task name indirection. Specify additionally the offset, where the pointer to the task name can be found in the TCB extension. E.g.

```
OSTCBExtPtr points to struct    OSTCBExtPtr points to struct

struct {                        struct {
     INT16U someval;                 char*  name;
     char*  name;                    INT16U someval;
     INT16U someval2;                INT16U someval2;
}                               }

task.config 0 2 2               task.config 0 2 0
```

   Specifying “0” to both naming arguments means, that no task name is evaluated.

NOTE: There is one **exception** on this. If the naming arguments are “0”, or if they are committed, a special case is searched automatically: If the TCB Extension structure is named `TASK_USER_DATA`, and if it contains (not points to) the task name in a member variable called `TaskName`, then this is automatically found and configured. If, for any reason, this automatic evaluation leads to wrong displays, you can either configure it manually as described above, or disable it by “`task.config 0 0 1`”.

Automatic Configuration

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 ucos**

Task names are automatically found, if the OS internal mechanisms are used (since version 2.60), or if the TCB Extension structure is named `TASK_USER_DATA`, and if it contains (not points to) the task name in a member variable called `TaskName`.

If a system symbol is not available, or if another value should be used for a specific system variable, then the corresponding argument must be set manually with the appropriate value (see **Manual Configuration**).

Quick Configuration Guide

To access all features of the OS Awareness you should follow the following roadmap:

1. Run the PRACTICE demo script (~~/demo/*<processor>*/kernel/ucos/ucos.cmm). Start the demo with "`do ucos`" and "`go`". The result should be a list of tasks, which continuously change their state.
2. Make a copy of the PRACTICE script file "ucos.cmm". Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the kernel resources and use the trace functions (if available).

Hooks & Internals in µC/OS-II

No hooks are used in the kernel.

To retrieve information on kernel objects, the OS Awareness uses the global µC/OS Variables and the structures defined in the ucos-ii.h file. Be sure that your application is compiled and linked with debugging symbols switched on.

Note for 68HC08 COSMIC compilers:

The compiler does not export symbol information on typedefs to unnamed structures. You have to change them (in the ucos-ii.h file) to become named structures:

Original ucos-ii.h	Change to:
`typedef struct {`	`typedef struct os_event {`
`…`	`…`
`} OS_EVENT;`	`} OS_EVENT;`

Features

The OS Awareness for µC/OS-II supports the following features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following µC/OS-II components can be displayed:

TASK.Task	Tasks
TASK.Event	Intertask Communication
TASK.Flag	Event Flags
TASK.TImer	Timers
TASK.Memory	Memory Partitions
TASK.PARtition	Space Partitions
TASK.PROCess	MPU Processes

For a description of the commands, refer to chapter “**µC/OS-II 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 Stack Coverage is only available, if you enabled `OS_TASK_CREATE_EXT_EN`, and if you created your tasks with `OSTaskCreateExt()`. To ensure proper stack calculation, specify `OS_TASK_OPT_STK_CLR` as an create option.

Note for C166 using Tasking Compiler:
The version 1.0 of the Tasking C166 port (author: K. Wannenmacher) lacks the updating of the OSTCBStkPtr variable. This causes, that the “current stack pointer” is displayed wrong.

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.

The **TASK.TASK <task>** window contains a button ("context") to execute this command with the displayed task, and to switch back to the current context ("current").

Not available for C166!
The version 1.0 of the Tasking C166 port (author: K. Wannenmacher) lacks the updating of the OSTCBStkPtr variable. This disables the usage of this feature, as we are not able to find the context of the task.

Dynamic Task Performance Measurement

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

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

For a general description of the **PERF** command group, refer to **“General Commands Reference Guide P”** (general_ref_p.pdf).

Task Runtime Statistics

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

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

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

Command	Description
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)”.

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 added to the calling task.

©1989-2023 Lauterbach

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)".

µC/OS-II specific Menu

The menu file “ucos.men” contains a menu with µC/OS-II specific menu items. Load this menu with the **MENU.ReProgram** command.

NOTE:	Load *first* the application symbols, *then* the µC/OS-II specific menu. The loading of the menu evaluates the existence of some µC/OS-II objects and creates the menu accordingly.

You will find a new menu called **µC/OS**.

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

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

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

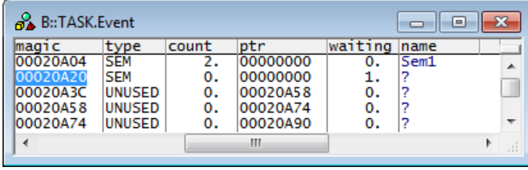
TASK.Event

Display events

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

Displays the event table of μC/OS-II or detailed information about one specific event. The event table holds all intertask communication mechanisms.

Without any arguments, a table with all created events will be shown. Specify a event magic number to display detailed information on that event.



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

The fields "magic", "ptr" and several fields in the detailed window are mouse sensitive, double clicking on them opens appropriate windows.

TASK.Flag

Display flags

Format: **TASK.Flag** <*flag*>

Displays the flag table of μC/OS-II or detailed information about one specific flag

Without any arguments, a table with all created flags will be shown. Specify a flag magic number to display detailed information on that flag.

"magic" is a unique ID, used by the OS Awareness to identify a specific flag (address of the OS_FLAG_GRP structure). The field "magic", and the task fields in the detailed window are mouse sensitive, double clicking on them opens appropriate windows.

TASK.Memory

Display memory partitions

Format:	**TASK.Memory**

Displays the table of all created memory partitions of µC/OS-II.

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

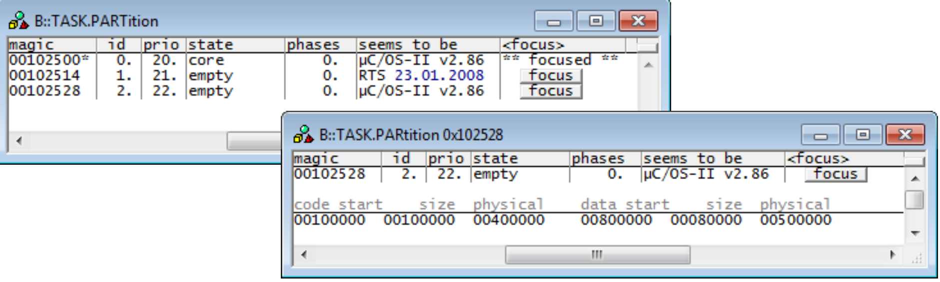
The field “address” is mouse sensitive, double clicking on it opens the appropriate window.

TASK.PARtition

Display space partitions

Format:	**TASK.PARtition**

Displays the table of all created space partitions of µC/OS-II.



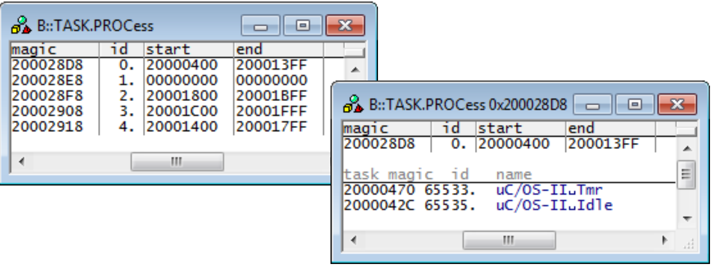
“magic” is a unique ID, used by the OS Awareness to identify a specific space partition.

TASK.PROCess

Display MPU processes

Format:	**TASK.PROCess**

Displays the table of all created MPU processes of µC/OS-II.



"magic" is a unique ID, used by the OS Awareness to identify a specific process.

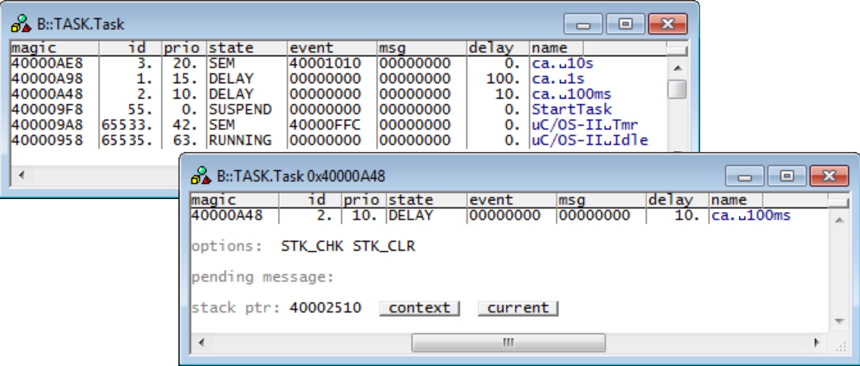
TASK.Task

Display tasks

Format:	**TASK.Task** <*task*>

Displays the task table of µC/OS-II or detailed information about one specific task.

Without any arguments, a table with all created tasks will be shown.
Specify a task magic number to display detailed information on that task.



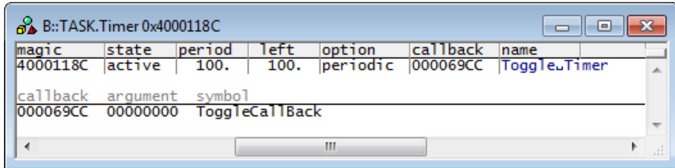
"magic" is a unique ID, used by the OS Awareness to identify a specific task (address of the TCB).

The fields "magic", "event", "msg" and "stack ptr" are mouse sensitive, double clicking on them opens appropriate windows. "magic" has a context sensitive menu, too.

Pressing the "context" button changes the register context to this task. "current" resets it to the current context. See "**Task Context Display**".

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

Displays the timer table of µC/OS (since 2.80) or detailed information about one specific timer.



Without any arguments, a table with all created timers will be shown. Specify a timer magic number to display detailed information on that timer.

“magic” is a unique ID, used by the OS Awareness to identify a specific timer (address of the OS_TMR structure). The fields “magic”, and “callback” are mouse sensitive, double clicking on them opens appropriate windows.

µC/OS-II PRACTICE Functions

There are special definitions for µC/OS 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.

TASK.PAR.AVAIL() Space partitions

Syntax: **TASK.PAR.AVAIL()**

Returns 1 if space partitions are configured.

Return Value Type: Hex value.

TASK.PROC.AVAIL() MPU processes

Syntax: **TASK.PROC.AVAIL()**

Returns 1 if MPU processes are configured.

Return Value Type: Hex value.

Syntax: **TASK.STRUCT(tcb)**

Returns the symbol type name of the TCB structure.

Return Value Type: Hex value.