# OS Awareness Manual OS21

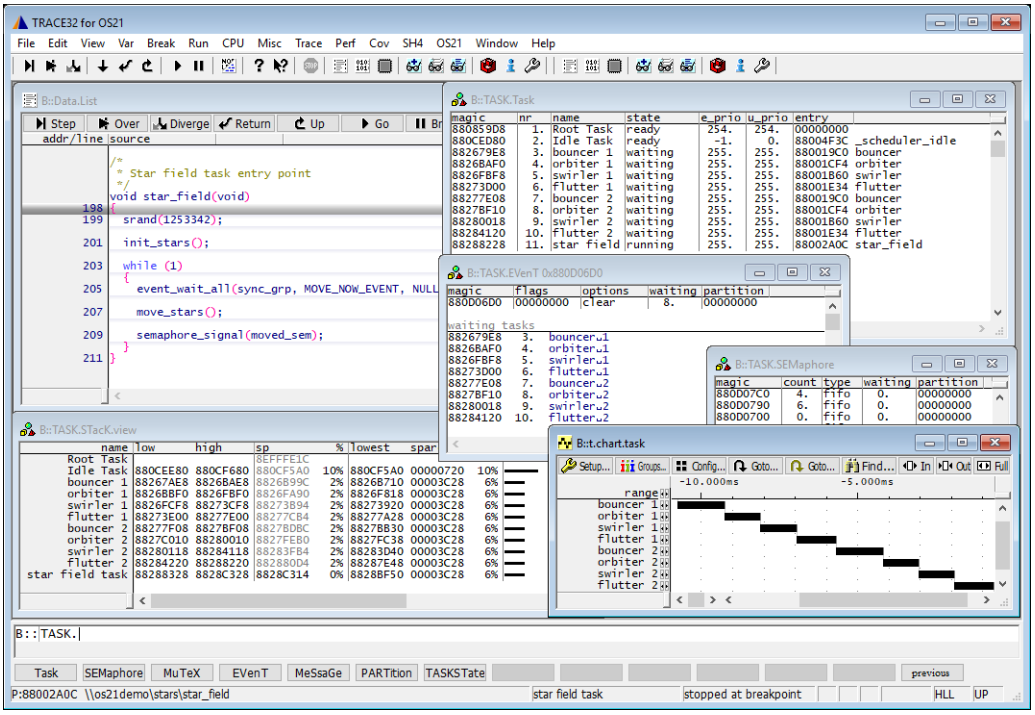
# OS Awareness Manual OS21

**Version 18-Nov-2020**

## 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 OS21 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:**

- **"Debugger Basics - Training"** (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 OS21 is supported for the following versions:

- OS21 V2.x and V3.x on ST40 and ARM

The **TASK.CONFIG** command loads an extension definition file called “os21.t32” (directory “~~/demo/<*processor*>/kernel/os21”). It contains all necessary extensions.

Automatic configuration tries to locate the OS21 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). In case of a ICE or FIRE, 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 os21** |
|---|---|

See **Hooks & Internals** for details on the used symbols.

See also the example “~~/demo/<*processor*>/kernel/os21/os21.cmm”.

# Quick Configuration Guide

To get a quick access to the features of the OS Awareness for OS21 with your application, follow the following roadmap:

1. Copy the files "`os21.t32`" and "`os21.men`" to your project directory

   (from TRACE32 directory "~~/demo/<*processor*>/kernel/os21").

2. Start the TRACE32 Debugger.

3. Load your application as normal.

4. Execute the command "`TASK.CONFIG os21`"

   (See "**Configuration**").

5. Execute the command "`MENU.ReProgram os21`"

   (See "**OS21 Specific Menu**").

6. Start your application.

Now you can access the OS21 extensions through the menu.

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

# Hooks & Internals in OS21

No hooks are used in the kernel.

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

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols and structure definitions. Ensure that the kernel is compiled with debug information and that access to the kernel structures is possible every time when features of the OS Awareness are used.

The kernel (and thus the OS Awareness) provides a list semaphores, mutexes and Events ONLY, if the debug checks for this objects are switched on. To include these debug checks, edit the "`src/os21/makest40.inc`" file and define the constants"`CONF_DEBUG_CHECK_SEM`", "`CONF_DEBUG_CHECK_MTX`" and "`CONF_DEBUG_CHECK_EVT`".

# Features

The OS Awareness for OS21 supports the following features.

## Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following OS21 components can be displayed:

| | |
|---|---|
| **TASK.Task** | Tasks |
| **TASK.SEMaphore** | Semaphores |
| **TASK.MuTeX** | Mutexes |
| **TASK.EVenT** | Event groups |
| **TASK.MeSsaGe** | Message queues |
| **TASK.PARTition** | Partitions |

For a description of the commands, refer to chapter "**OS21 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.

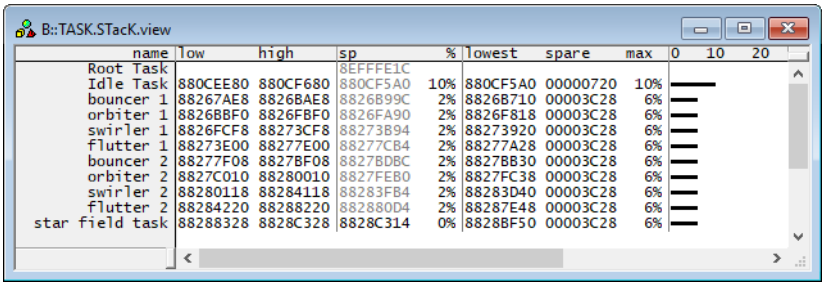
## Task Stack Coverage

For stack usage coverage of OS21 tasks, you can use the **TASK.STacK** command. Without any parameter, this command will set up a window with all active OS21 tasks. If you specify only a magic number as parameter, the stack area of this task will be automatically calculated.

To use the calculation of the maximum stack usage, flag memory must be mapped to the task stack areas, when working with the emulation memory. When working with the target memory, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is 0x12345678).

To add/remove one task to/from the task stack coverage, you can either call the **TASK.STacK.ADD** rsp. **TASK.STacK.ReMove** commands with the task magic number as parameter, or omit the parameter and select from the task list 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:

| | |
|---|---|
| **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:

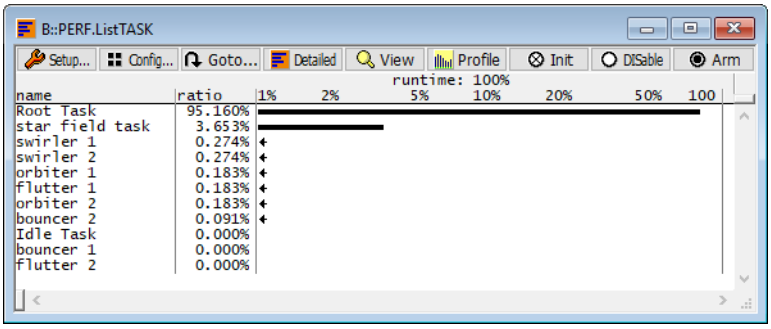
1. Open the **Frame /Caller /Task** *<task>* window.
2. Double-click the line showing the OS service call.

# Dynamic Task Performance Measurement

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

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

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




# Task Runtime Statistics

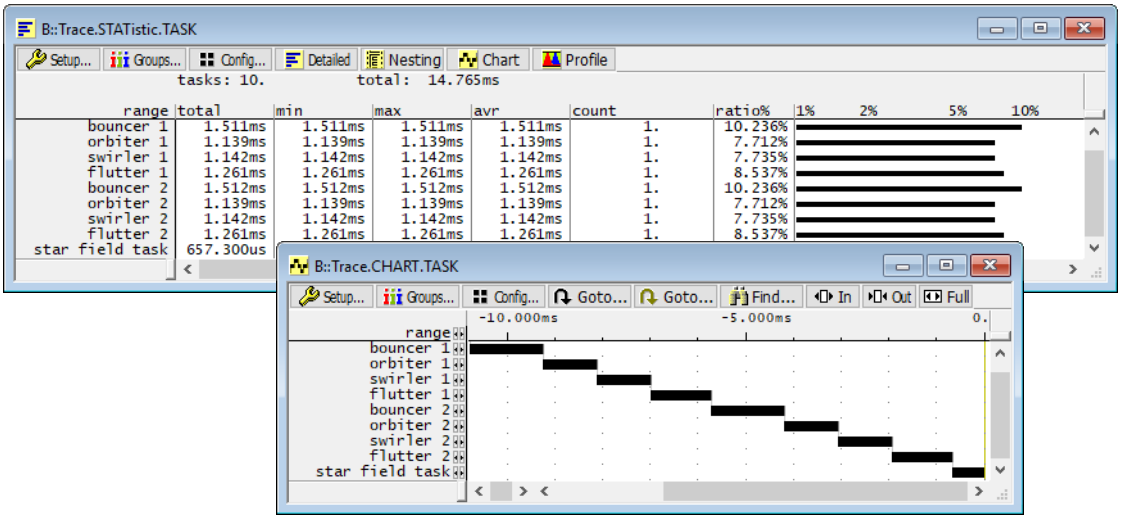
| | |
|---|---|
| **NOTE:** | This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding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

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

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

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

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



| | |
|---|---|
| **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)".

| 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)".

# OS21 specific Menu

The menu file “os21.men” contains a menu with OS21 specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **OS21**.

- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the OS21 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:

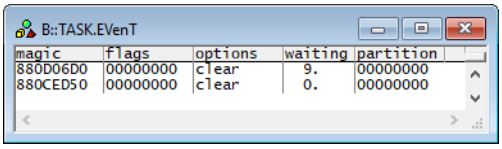
- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with the default display.
- The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime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

## TASK.EVenT — Display event groups

Format: **TASK.EVenT** [<*event group*>]

Displays the event group table of OS21 or detailed information about one specific event group.

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




"magic" is the ID of the event group, used by OS21 and the OS Awareness to identify a specific event group (address of the event group structure).

"waiting" specifies the number of tasks waiting for events of this event group.

The fields "magic" and "waiting tasks" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

**NOTE:** An event group table is **only** available, if the kernel is compiled with debug checks for event groups. To switch these check on, define "`CONF_DEBUG_CHECK_EVT`" in the kernel make file "`sh-superh-elf/src/os21/makest40.inc`". If these checks are not included, you must give an event group ID as argument.

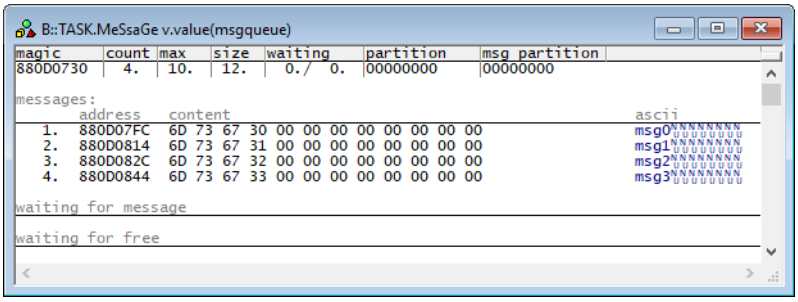
**Hint:** When using variables that hold the event group ID (initialized with "`event_group_create`"), use the function "`v.value(myevtgrp)`" to retrieve the event group ID. E.g.:

```
TASK.MuTeX v.value(sync_grp)
```

Format: **TASK.MeSsaGe** <*message_queue*>

Displays detailed information about one specific message queue.

Specify a message queue magic number to display detailed information on that message queue.




“magic” is the ID of the message queue, used by OS21 and the OS Awareness to identify a specific message queue (address of the message queue structure).

“count” specifies the number of messages stored in the queue.

“max” specifies the maximum number of messages that can be stored in the queue.

“size” specifies the size of the messages in bytes.

“waiting” specifies the number of tasks waiting for getting a message from the queue and waiting to place a message into the queue.

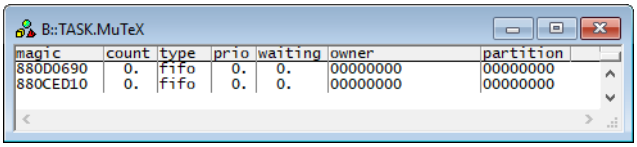
The fields “magic” and “address” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

**Hint:** When using variables that hold the message queue ID (initialized with “`message_create_queue`”), use the function “`v.value(mymsgq)`” to retrieve the message queue ID. E.g.:
`TASK.MeSsaGe v.value(msgqueue)`

Format: **TASK.MuTeX** [<*mutex*>]

Displays the mutex table of OS21 or detailed information about one specific mutex.

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




"magic" is the ID of the mutex, used by OS21 and the OS Awareness to identify a specific mutex (address of the mutex structure).
"waiting" specifies the number of tasks waiting for this mutex.
"owner" specifies the mutex owning task.

The fields "magic", "owner" and "waiting tasks" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

**NOTE:** A mutex table is **only** available, if the kernel is compiled with debug checks for mutexes. To switch these check on, define "`CONF_DEBUG_CHECK_MTX`" in the kernel make file "`sh-superh-elf/src/os21/makest40.inc`". If these checks are not included, you must give a mutex ID as argument.

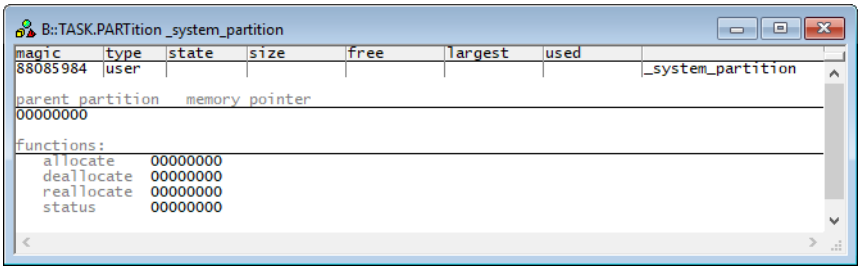
**Hint:** When using variables that hold the mutex ID (set with "`mutex_create_xxx`"), use the function "`v.value(mymutex)`" to retrieve the mutex ID. E.g.:

```
TASK.MuTeX v.value(_mem_mutex)
```

Format: **TASK.PARTition** <*partition*>

Displays detailed information about one specific partition.

Specify a partition magic number to display detailed information on that partition.




"magic" is the ID of the partition, used by OS21 and the OS Awareness to identify a specific partition (address of the partition structure).

The fields "magic" and "functions" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

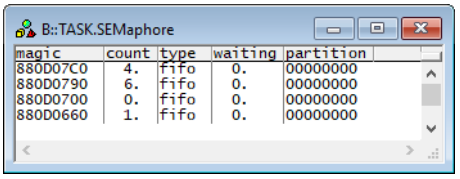
**Hint:** When using variables that hold the partition ID (initialized with "`partition_create_xxx`"), use the function "`v.value(mypart)`" to retrieve the partition ID. E.g.:
`TASK.PARTition v.value(mypartition)`

## TASK.SEMaphore Display semaphores

Format: **TASK.SEMaphore** [<*semaphore*>]

Displays the semaphore table of OS21 or detailed information about one specific semaphore.

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




"magic" is the ID of the semaphore, used by OS21 and the OS Awareness to identify a specific semaphore (address of the semaphore structure).

“waiting” specifies the number of tasks waiting for this semaphore.

The fields “magic” and “waiting tasks” are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

**NOTE:** A semaphore table is **only** available, if the kernel is compiled with debug checks for semaphores. To switch these check on, define “`CONF_DEBUG_CHECK_SEM`” in the kernel make file “`sh-superh-elf/src/os21/makest40.inc`”. If these checks are not included, you must give a semaphore ID as argument.

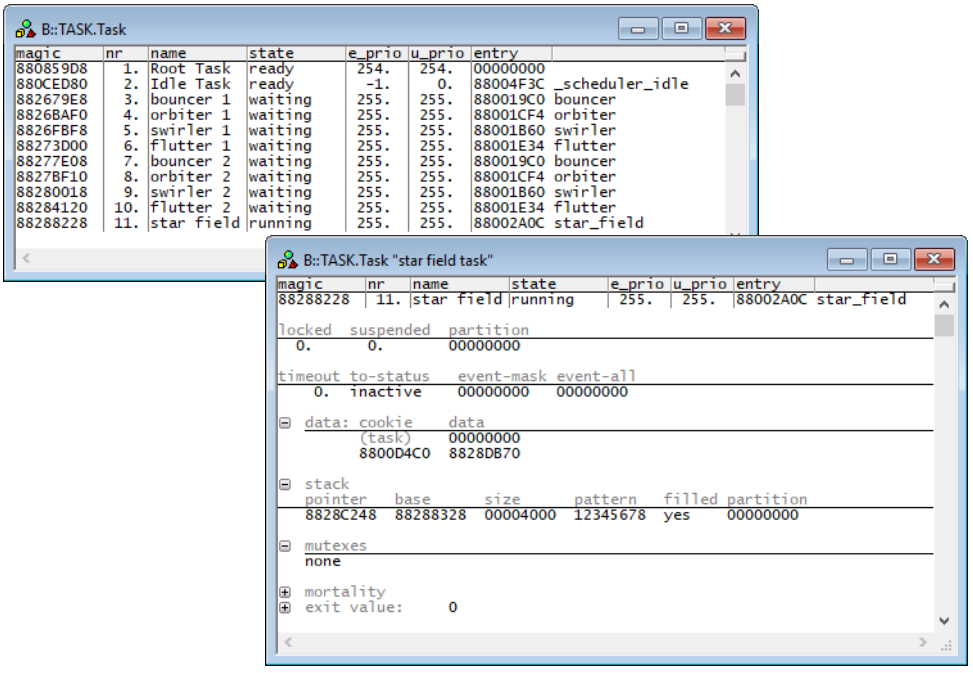
**Hint:** When using variables that hold the semaphore ID (initialized with “`semaphore_create_xxx`”), use the function “`v.value(mysema)`” to retrieve the semaphore ID. E.g.:
`TASK.SEMaphore v.value(moved_sem)`

## TASK.Task — Display tasks

| Format: | **TASK.Task** [<*task*>] |
|---|---|

Displays the task table of OS21 or detailed information about one specific task.

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




“magic” is the ID of the task, used by OS21 and the OS Awareness to identify a specific task (address of the task structure).
“nr” specifies a task number.
“e_prio” and “u_prio” specify the effective priority and user priority (given at task creation).

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

Pressing the “context” button (if available) changes the register context to this task. “current” resets it to the current context. See “**Task Context Display**”.

## TASK.TASKState — Mark task state words

Format: **TASK.TASKState**

This command sets Alpha breakpoints on all tasks status words.

The statistic evaluation of task states (see **Task State Analysis**) requires recording of the accesses to the task state words. By setting Alpha breakpoints to these words and selectively recording Alpha’s, you can do a selective recording of task state transitions.

Because setting the Alpha breakpoints by hand is very hard to do, this utility command automatically sets the Alpha’s to the status words of all tasks currently created. It does NOT set breakpoints to tasks that terminated or haven’t yet been created.

# OS21 PRACTICE Functions

There are special definitions for OS21 specific PRACTICE functions.

## TASK.CONFIG() OS Awareness configuration information

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

**Parameter and Description**:

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

**Return Value Type**: Hex value.

# Frequently-Asked Questions

No information available