LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
OSE Delta

Release 02.2025

MANUAL

OS Awareness Manual OSE Delta

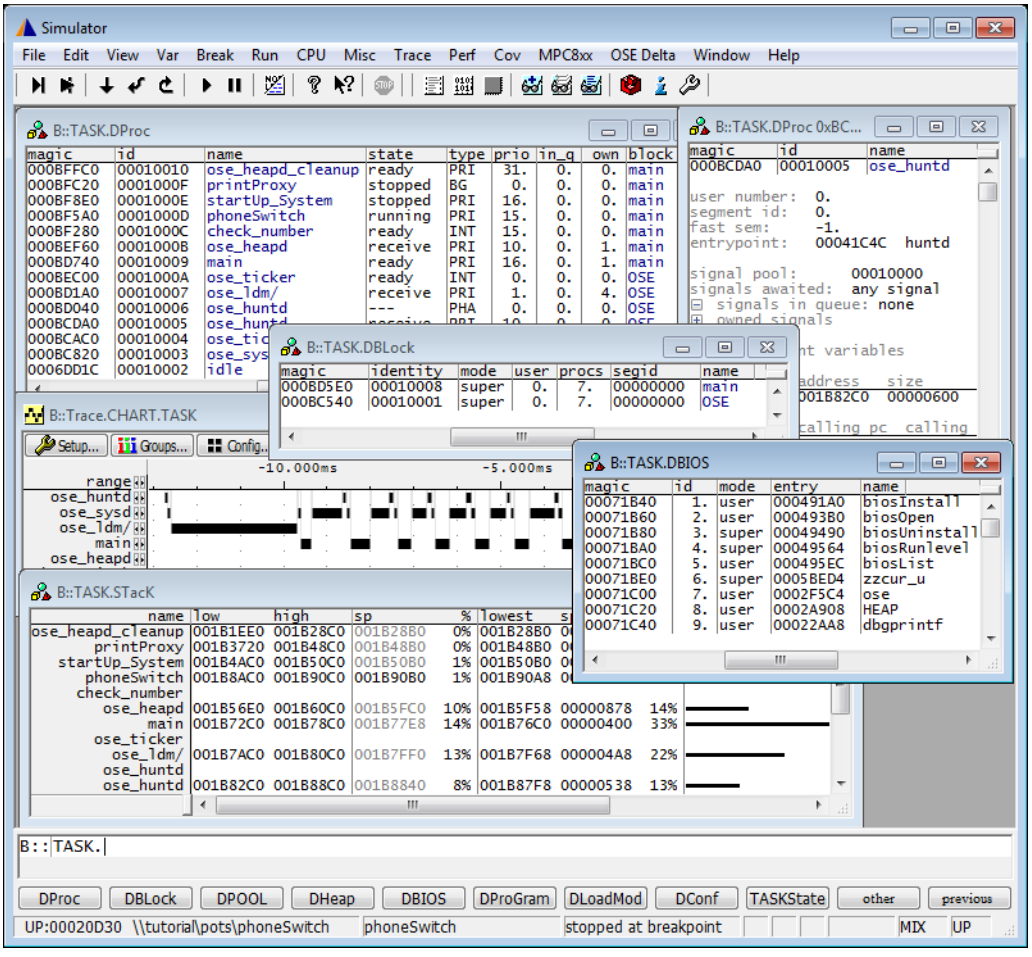
OS Awareness Manual OSE Delta

Version 13-Feb-2025

History

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

Overview



The OS Awareness for OSE Delta (aka “OSE”) contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

Terminology

OSE Delta uses the term “process”, while TRACE32 uses the term “task”. Both are used interchangeably throughout this manual.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Debugger Tutorial”** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **“General Commands”** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.
- **“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.

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**.
- **“T32Start”** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.

Supported Versions

Currently OSE Delta is supported and tested for the following versions:

- OSE Delta (aka “OSE”) for ARM, CPU32, MIPS, PowerPC;
- OSE Versions 3.0 to 5.x

Configuration

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

Automatic configuration tries to locate the OSE Delta 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, too.

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**.

Manual Configuration

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

Format:	**TASK.CONFIG osed** <*magic_address*> **0** <*ose_kernel_struct*>

<*magic_address*> Specifies a memory location that contains the current running task. This address can be found at the location zzosvarp+0x2c. Use “`data.long(zzosvarp+2c)`” as magic address.

<*ose_kernel_struct*> Specify the additional argument with the symbol for the OSE Delta kernel structure pointer. This is normally “`zzosvarp`”.

Example for manual configuration:

```
TASK.CONFIG osed data.long(zzosvarp+2c) 0 zzosvarp
```

See **Hooks & Internals** for details on <*ose_kernel_structure*>.

See also the example "~~/demo/<*processor*>/kernel/osedelta/osed.cmm".

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 osed**

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 (see **Manual Configuration**).

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

See also the example "~~/demo/<*processor*>/kernel/osedelta/osed.cmm"

Quick Configuration Guide

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

1. Copy the files "`osed.t32`" and "`osed.men`" to your project directory (from TRACE32 directory "~~/demo/<*processor*>/kernel/osedelta").
2. Start the TRACE32 Debugger.
3. Load your application as normal.
4. Execute the command:

   ```
   TASK.CONFIG osed
   ```

   See "**Automatic Configuration**".
5. Execute the command:

   ```
   MENU.ReProgram osed
   ```

   See "**OSE Delta Specific Menu**".
6. Start your application.

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

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

No hooks are used in the kernel.

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

Be sure that your application is compiled and linked with debugging symbols switched on.

Features

The OS Awareness for OSE Delta supports the following features.

Terminal Emulation for dbgprintf

The terminal emulation window can be used as an output window for the “dbgprintf” function of OSE. The communication via two memory buffers requires no external interface. See the **TERM** command group for a description of the terminal emulation. Find an example dbgprintf implementation for the Terminal Emulation in “~~/demo/<*processor*>/kernel/osedelta/t32printf.c”.

Display of Kernel Resources

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

TASK.DBIOS	BIOS modules
TASK.DBLock	Blocks
TASK.DConf	Configuration
TASK.DLoadMod	Load modules
TASK.DPOOL	Pools
TASK.DProc	Processes
TASK.DProGram	Loaded programs

For a description of the commands, refer to chapter “**OSE Delta Commands**”.

If your hardware allows memory access while the target is running, these resources can be displayed “On The Fly”, i.e. while the application is running, without any intrusion to the application.

Without this capability, the information will only be displayed if the target application is stopped.

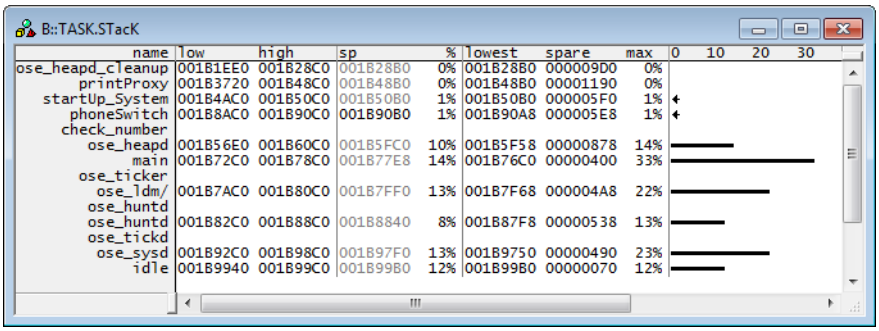
Task Stack Coverage

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

To use the calculation of the maximum stack usage, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is zero).

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

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



Task-Related Breakpoints

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

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

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- For a general description of the **Break.Set** command, please see its documentation.

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

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

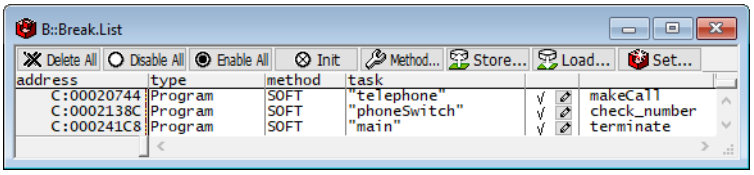
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

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

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**, **List.auto**, **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.

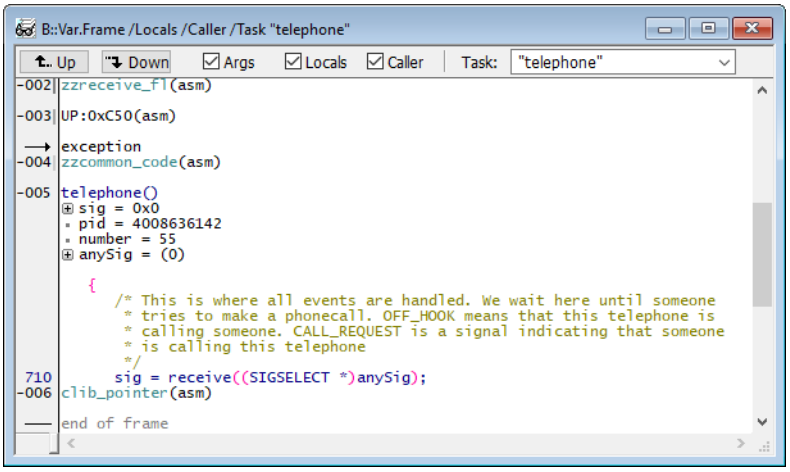
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.



MMU Support

This chapter only applies to OSE versions since 5.0.

OSE uses virtual memory management for load modules. In order to provide full debugging capabilities on load modules, the Debugger needs to know how virtual addresses are translated to physical addresses and vice versa. All MMU commands refer to this necessity.

MMU Declaration

To access the virtual and physical addresses correctly, the debugger needs to know the format of the MMU tables in the target.

The following command is used to declare the basic format of MMU tables:

MMU.FORMAT *<format>* [*<base_address>* [*<logical_kernel_address_range>* *<physical_kernel_address>*]]	Define MMU table structure

<format> Options for ARM:

<format>	Description
STD	Standard format defined by the CPU
TINY	MMU format using a tiny page size of only 1024 bytes

<format> Options for PowerPC:

<format>	Description
OSE	OSE format for load modules
STD	Standard format defined by the CPU

<format> Options for RISC-V:

<format>	Description
STD	Automatic detection of the page table format from the SATP register.
SV32	32-bit page table format (for SV32 targets only)
SV32X4	Stage 2 (G-stage) 32-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables.
SV39	39-bit page table format (for SV64 targets only)
SV39X4	Stage 2 (G-stage) 39-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables.
SV48	48-bit page table format (for SV64 targets only)
SV48X4	Stage 2 (G-stage) 48-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables.
SV57	57-bit page table format (for SV64 targets only)
SV57X4	Stage 2 (G-stage) 57-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables.

<format> Options for x86:

<format>	Description
EPT	Extended page table format (type autodetected)
EPT4L	Extended page table format (4-level page table)
EPT5L	Extended page table format (5-level page table)
P32	32-bit format with 2 page table levels
PAE	Format with 3 page table levels
PAE64	64-bit format with 4 page table levels
PAE64L5	64-bit format with 5 page table levels
STD	Automatic detection of the page table format used by the CPU

<base_address>

<base_address> specifies the base address of the kernel translation table. This address can generally be found at the label "`hal_global_page_table`".

<logical_kernel_address_range>

<logical_kernel_address_range> specifies the virtual to physical address translation of the kernel address range. Currently not necessary.

<physical_kernel_address>

<physical_kernel_address> specifies the physical start address of the kernel. Currently not necessary.

The kernel code and linked-in processes reside in a physical address space. See your OSE configuration file (usually rtose5.conf) for the krn/log_mem entry with mapping type "SASE". Create a debugger MMU entry with **TRANSlation.Create** that corresponds to this entry.

E.g. if your configuration file specifies:

```
krn/log_mem/RAM_SASE=base:0 size:32M mapping_type:SASE
```

then create the following debugger MMU entry with the virtual address range as first parameter and the equally mapped physical base address as second parameter:

```
TRANSlation.Create 0x0--0x01ffffff 0x0
```

Load modules get their own virtual address area when loaded. The debugger can use its own table walk to resolve the virtual addresses to physical addresses. Enable the table walk with **tRANSlation.TableWalk ON**.

Use **TRANSlation.ON** to enable the debugger's address translation.

A complete MMU declaration may look like this:

```
MMU.FORMAT OSE hal_global_page_table
TRANSlation.Create 0x0--0x01ffffff 0x0
TRANSlation.TableWalk ON
TRANSlation.ON
```

If the MMU format is not available, or for examination of the translation, you can scan the address translation of a load module after loading them. Use the command **TASK.MMU.SCAN** to read the translation tables.

TRANSlation.List shows the setup of the MMU declaration and scanned translations.

SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application runs may change over time.

To support such SMP systems, the debugger allows a “system view”, where one TRACE32 PowerView GUI is used for the whole system, i.e. for all cores that are used by the SMP OS. For information about how to set up the debugger with SMP support, please refer to the **Processor Architecture Manuals**.

All core relevant windows (e.g. **Register.view**) show the information of the current core. The status bar of the debugger indicates the current core. You can switch the core view with the **CORE.select** command.

Target breaks, be they manual breaks or halting at a breakpoint, halt all cores synchronously. Similarly, a **Go** command starts all cores synchronously. When halting at a breakpoint, the debugger automatically switches the view to the core that hit the breakpoint.

Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

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.CONFIG.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

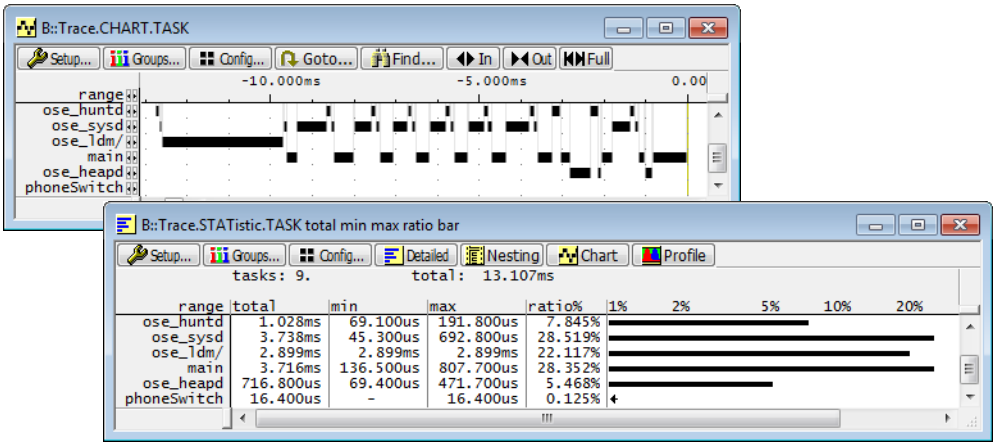
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **LOGGER**). For details, refer to **"OS-aware Tracing"** in TRACE32 Concepts, page 36 (trace32_concepts.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"** in TRACE32 Concepts, page 36 (trace32_concepts.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"** in TRACE32 Concepts, page 36 (trace32_concepts.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:

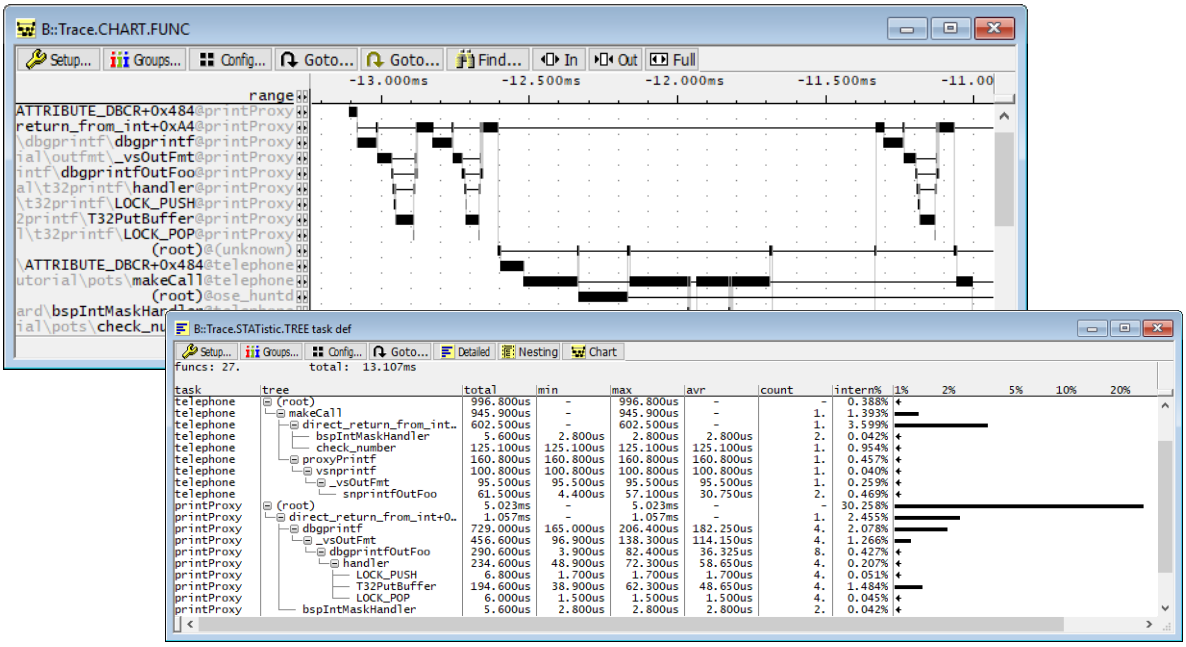
```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

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)".



OSE Delta specific Menu

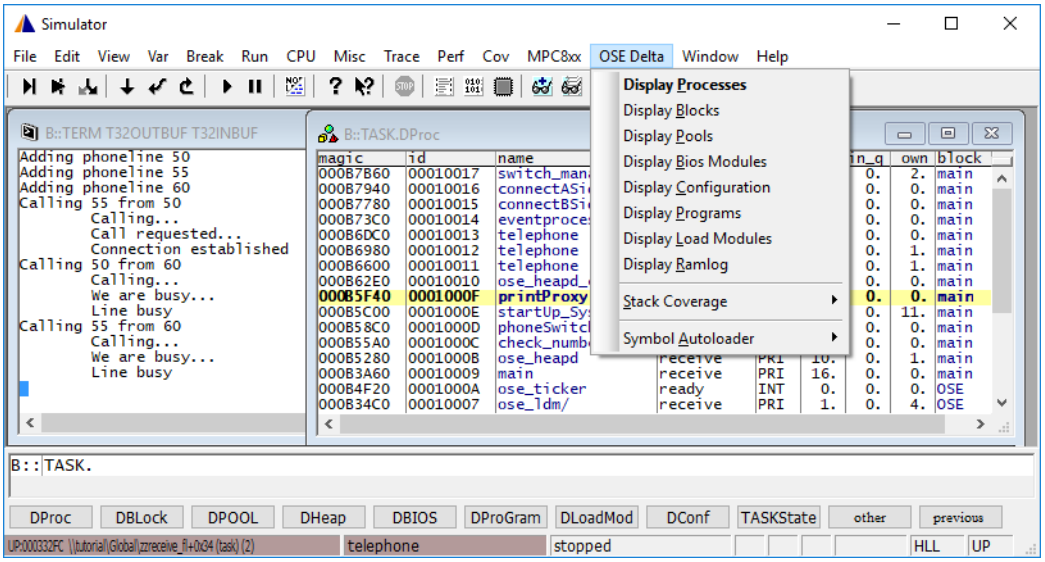
The menu file "osed.men" contains a menu with OSE Delta specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- The **Display** menu items launch the appropriate kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the OSE Delta 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 default display.
- The **Perf** menu contains additional submenus for task runtime statistics and statistics on task states.



Debugging OSE Load Modules

OSE can load and link additional software parts, so-called “load modules”. Debugging of these load modules is possible with the help of the OS Awareness for OSE.

The load modules are handled differently between the OSE versions.

Load Modules in OSE 4

If the load module is already loaded in the OSE4 system, you need to load the symbols into the debugger and relocate them to the actually linked address. Use

TASK.SYMbol.LOAD *<loadmodule>*

or the “load” button in

TASK.DProGram

to load the symbols.

Alternatively, you can load the symbols manually and relocate them with **TASK.SYMbol.RELOC**.
Example:

```
Data.LOAD.Elf tutorial_lm.elf /NoCODE /NoClear /cygdrive
TASK.SYMbol.RELOC "tutorial_lm.elf"
```

If you want to debug a load module from the beginning, you have to load the symbols into the debugger *after* loading the module into your system, but *before* starting it.

Follow this roadmap:

1. Ensure that your system is running and the symbols for the monolith (only) are loaded.
2. Ensure that the OS Awareness for OSE is configured
3. Transfer your load module onto the target (if not already there
4. Use the OSE shell to load and link the module into the system:
   `pgload tutorial_lm.elf`
   Notice the “handle”.
5. In TRACE32, execute this command to stop the program run.

   ```
   Break
   ```

6. Ensure with **TASK.DProGram** that the load module is listed there.
7. Load the symbols of the module as mentioned above.
8. Check with “`sYmbol main`”, which main function the module uses.
9. Set a breakpoint onto the main function:

   ```
   Break.Set \\tutorial_lm\osemain\main
   ```

10. Continue the program run with:

    ```
    Go
    ```

11. Use the OSE shell to start the load module, using the handle given above:
    `pgstart 1`

The debugger will halt the application at “main()”. You can now continue debugging the start of the load module.

Load Modules in OSE 5

For load modules in OSE5, you need to define the MMU layout to the debugger or scan the MMU pages of the load module area. See chapter **MMU Support** for this.

When a load module is already loaded in the OSE5 system, load the symbols into the debugger and relocate them to the actually linked address. You can use the **Symbol Autoloader** to manage the loading of symbol files or explicitly load symbols with **TASK.SYMbol.LOAD**. The “load” button in **TASK.DProGram** triggers this command, too.

Alternatively, you can load the symbols manually and relocate them. Use the “`/RELOCTYPE`” option to load the symbols of a load module. The symbol file name must be the same as the file name of the load module.
Example:

```
Data.LOAD.Elf pingpong_debug.elf /NoCODE /NoClear /RELOCTYPE 1
```

You can also use the relocation information of the **task.lm.relocinfo()** function to relocate the sections in a script.
Example:

```
&reloc=task.lm.relocinfo("pingpong")
Data.LOAD.Elf pingpong_debug.elf /NoCODE /NoClear &reloc
```

If you want to debug a load module from the beginning, you have to load the symbols into the debugger *after* loading the module into your system, but *before* starting it.

Follow this roadmap:

1. Ensure that your system is running and the symbols for the monolith (only) are loaded.
2. Ensure that the OS Awareness for OSE is configured.
3. Transfer your load module onto the target (if not already there).
4. Use the OSE shell to load and link the module into the system:
   `pm_install pingpong /ram/pingpong_debug.elf`
   `pm_create pingpong`
   Notice the “handle”.
5. In TRACE32, execute this command to stop the program run:

   ```
   Break
   ```

6. Scan the load module MMU pages if necessary.
7. Ensure with **TASK.DProGram** that the load module is listed there.
8. Load the symbols of the module as mentioned above.
9. Check with “`sYmbol.name main`”, which main function the module uses.
10. Set a breakpoint onto the main function, e.g.:

    ```
    Break.Set \\pingpong_debug\pingpoing_main\main
    ```

11. Continue the program run with:

```
Go
```

12. Use the OSE shell to start the load module, using the handle given above, e.g.:
`pm_start 1`

The debugger will halt the application at “main()”. You can now continue debugging the start of the load module.

Symbol Autoloader

The OS Awareness for OSE Delta contains an autoloader, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding OSE Delta load modules and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader, the debugger invokes the appropriate command. The command is usually a call to a PRACTICE script that loads the symbol file to the appropriate addresses.

The command **sYmbol.AutoLOAD.List** shows a list of all known address ranges/components and their symbol load commands.

The autoloader must be enables with the **TASK.SYMbol.Option AutoLoad** command.

The autoloader reads the target’s tables for the load modules and fills the autoloader list with the modules found on the target. All necessary information, such as load addresses, are retrieved from kernel-internal information.

sYmbol.AutoLOAD.CHECKCoMmanD "*<action>*"

<action>	Action to take for symbol load, e.g. "`DO autoload.cmm` "

If an address is accessed that is covered by the autoloader list, the autoloader calls *<action>* and appends the load addresses and the space ID of the component to the action. Usually, *<action>* is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example script “autoload.cmm” in the ~~/demo directory.

The point in time when the component information is retrieved from the target can be set:

sYmbol.AutoLOAD.CHECK [**ON** | **OFF** | **ONGO**]

(no argument)	A single **sYmbol.AutoLOAD.CHECK** command refreshes the information about the target.
ON	The debugger automatically reads the information on every go/halt or step cycle. This significantly slows down the debugger’s speed when single stepping.

ONGO The debugger automatically reads the information on every go/halt cycle, but not when single stepping.

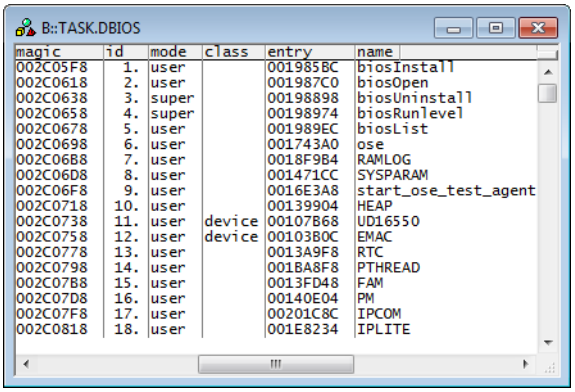
OFF no automatic update of the autoloader table will be done, you have to manually trigger the information read when necessary. To accomplish that, execute the **sYmbol.AutoLOAD.CHECK** command without arguments.

NOTE:	The autoloader covers only components that are already started. Components that are not in the current module table are not covered.

TASK.DBIOS — Display bios modules

Format: **TASK.DBIOS**

Displays a table of all installed bios modules of OSE Delta.



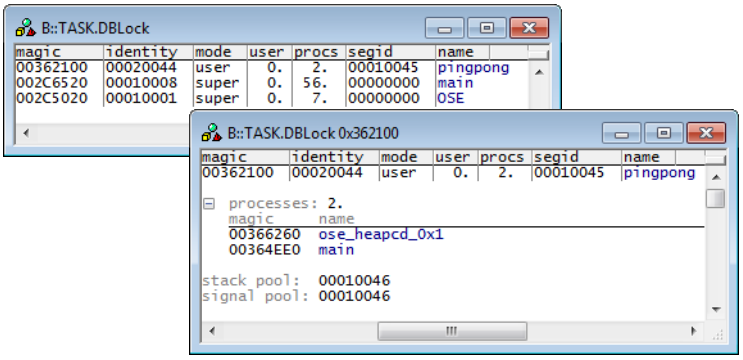
TASK.DBLock — Display blocks

Format: **TASK.DBlock** [<*block*>]

Displays the block table of OSE Delta or detailed information about one specific block.

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

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



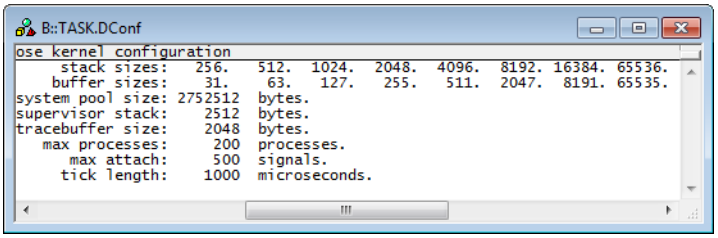
"magic" is a unique ID, used by the OS Awareness to identify a specific block (address of the BCB).

The field “magic” and pool IDs are mouse sensitive, double clicking on them opens appropriate windows.

TASK.DConf — Display kernel configuration

Format: **TASK.DConf**

Displays the OSE Delta kernel configuration.



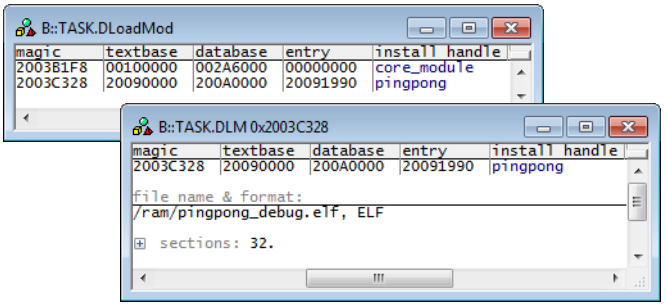
TASK.DLoadMod — Display load modules

Format: **TASK.DLoadMod** [<*loadmodule*>]

Only available on OSE5 systems. For OSE4 systems, use TASK.DProGram.

Displays a table with all loaded load modules of OSE Delta or detailed information about one specific load module.

Without any arguments, a table with all loaded load modules will be shown.
Specify the name or magic number of a load module to display detailed information on that load module.

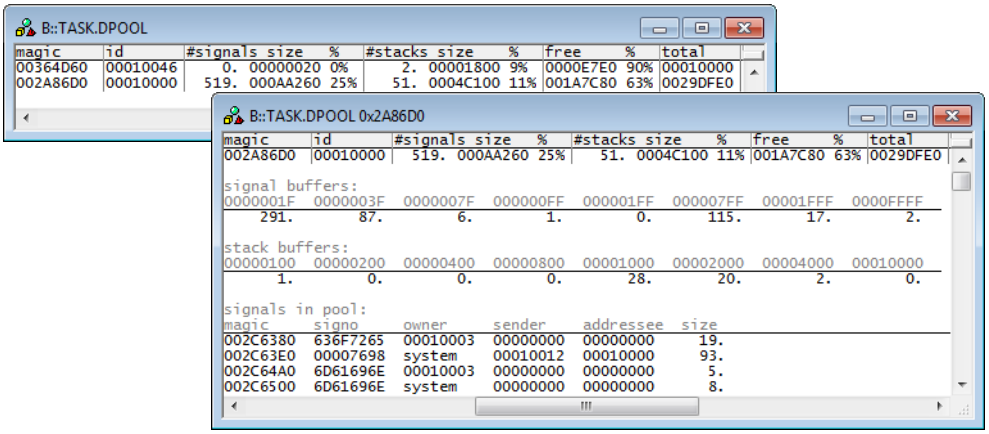


“magic” is a unique ID, used by the OS Awareness to identify a specific load module.

The fields “magic”, “textbase” and “entry" are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.DPOOL** [*<pool>*]

Displays the pool table of OSE Delta or detailed information about one specific pool.

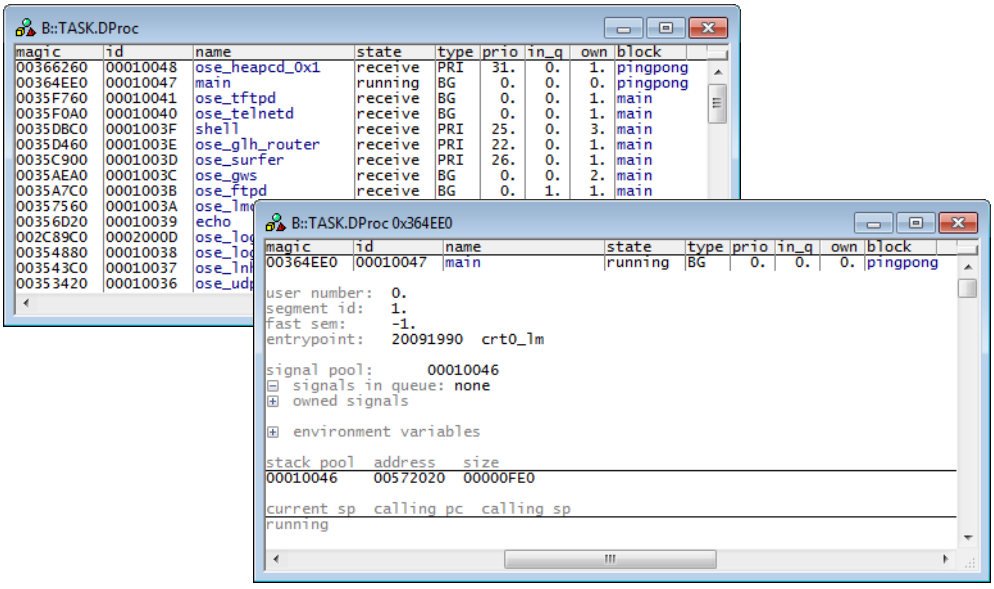


<pool> Without any arguments, a table with all created pools will be shown. Specify a pool magic number or ID to display detailed information on that pool.
"magic" is a unique ID, used by the OS Awareness to identify a specific pool (address of the pool control block).

The fields "magic" and "id" are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.DProc** [*<process>*]

Displays the process table of OSE Delta or detailed information about one specific process.

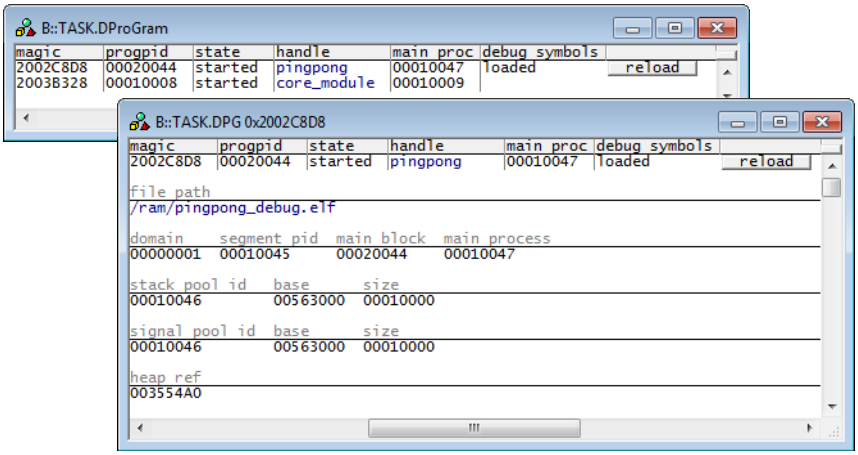


<process> Without any arguments, a table with all created processes will be shown. Specify a process name or magic number to display detailed information on that process, including its signal queue.
“magic” is a unique ID, used by the OS Awareness to identify a specific process (address of the PCB).

The fields “magic”, “name”, “pid”, “entrypoint” and “sender” are mouse sensitive, double clicking them opens appropriate windows.

Format: **TASK.DProGram** [<*program*>]

Displays a table with all loaded programs of OSE Delta or detailed information about one specific program.



<program> Without any arguments, a table with all loaded programs will be shown. Specify the name or magic number of a program to display detailed information on that program.
"magic" is a unique ID, used by the OS Awareness to identify a specific load module.

The fields "magic" and "name" are mouse sensitive, double clicking on them opens appropriate windows.

"debug symbols" shows if the appropriate debugging symbols are loaded into the debugger, and if relocation is needed.

The button on the end of the line shows either **load**, **relocate** or **reload**, depending on the state of the debugging symbols.

- **load** tries to load and relocate the debugging symbols, by executing **TASK.SYMbol.LOAD**.
- **relocate** appears, if symbols are loaded, but do not seem to fit to the addresses of the load module. It then tries to relocate the symbols automatically, by executing **TASK.SYMbol.RELOC**.
- **reload** first removes the symbols of the load module, second reloads them with **TASK.SYMbol.RELOAD**. Use this feature, if you changed your load module and reloaded it into the OSE system, to get the new symbols.

Only available on OSE5 systems.

Format: **TASK.MMU.SCAN** [*<address>* [*<size>*]]

Scans the MMU tables of OSE Delta for load modules.

Load modules are subject to a virtual memory management. In order to handle this correctly, the debugger needs to know the correct virtual to physical address translation for the load modules.

Without any argument, the debugger scans the whole address translation, which covers kernel, built-in processes and all load modules. This can take a very long time. To reduce the time, specify the start address and the size of your virtual address range (size defaults to 128MB if omitted). This range is defined in the OSE configuration file as “SAS” mapping type. E.g. if your configuration file specifies:

```
krn/log_mem/RAM=base:512M size:32M mapping_type:SAS
```

then scan this virtual address range after loading your modules with the virtual base address as first parameter and the size as second parameter:

```
TASK.MMU.SCAN 0x20000000 0x02000000
```

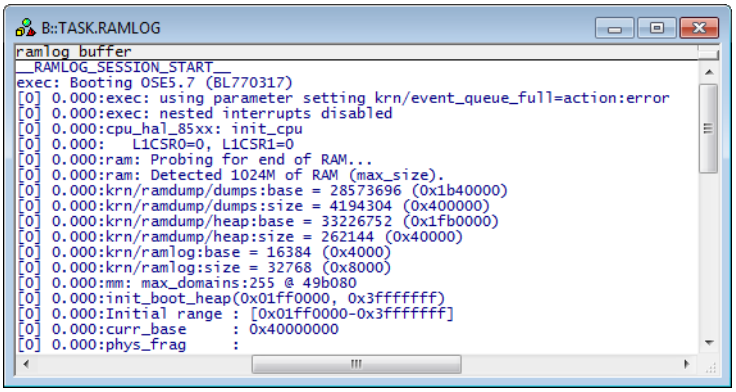
Check **TRANSlation.List** for the result.

See also chapter “**MMU Support**”.

TASK.RAMLOG Display ramlog

Format: **TASK.RAMLOG**

Displays the messages in the RAMLOG area of OSE.



Format: **TASK.SYMbol.**<*sub_cmd*>

Specify the name in quotes, or the magic of the program, as listed in **TASK.DProGram**.

LOAD <*program*> Tries to load the symbols of the load module associated with the specified program.
The debugger searches for a file with the name of the load module and loads the symbols of this file. After this, the debugger tries to relocate the symbols to the addresses, where the load module is located. For a comfortable loading of symbols, configure the **Symbol Autoloader**.

RELOAD <*program*> First removes the symbols of the specified program, before loading and relocating them again.

RELOC <*program*> Tries to relocate the symbols of a program to the address, where the associated lode module is located.

Option <*option*> <*value*> The <*option*> **LOADCMD** specifies, how the symbol file of a module is to be loaded. This command is only used, if the Symbol Autoloader is not configured.

The default load command is

```
Data.LOAD.ELF %s.%s /NoCODE /NoClear
```

where the first %s will be replaced by the program name, and the second %s by the file extension. You may change this, e.g. if you want to add additional load options:

```
TASK.SYMbol.Option LOADCMD "d.load.elf %s.%s /NoCODE /NoClear /gnu"
```

Option <*option*> <*value*> (cont.) The <*option*> **AutoLoad** [**ON** | **OFF**] enables or disables the **Symbol Autoloader**. When enabling, ensure that the Symbol Autoloader is configured correctly.

PRACTICE Functions

There are special definitions for OSE Delta specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

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

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).
kernel	**Parameter Type**: String (***without*** quotation marks). Returns the address of the kernel state variable.

Return Value Type: Hex value.

TASK.PG.ADDR() Segment address of program

Only OSE 4.x!

Syntax: **TASK.PG.ADDR(***<program>***, [0|1])**

Returns the segment address of the specified program.

Parameter and Description:

<program>	**Parameter Type**: String (***without*** quotation marks).
**0	1**	**Parameter Type**: Decimal or hex or binary value. 0 is program segment (P:) and 1 is data segment (D:).

Return Value Type: Hex value.

TASK.PG.RELOC() Relocation address for program

Only OSE 4.x!

Syntax: **TASK.PG.RELOC(**<*program*>**,** [**0**|**1**]**)**

Returns the relocation address for the specified program.

Parameter and Description:

<*program*>	**Parameter Type**: String (***without*** quotation marks).
0 | **1**	**Parameter Type**: Decimal or hex or binary value. 0 is program segment (P:) and 1 is data segment (D:).

Return Value Type: Hex value.

TASK.LM.LIST() Next magic in load module list

Syntax: **TASK.LM.LIST(**<*lmmagic*>**)**

Returns the next “magic” in the load module list. Specify zero for the first process. Returns zero if no further load module is available.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.LM.HANDLE() Install handle of load module

Syntax: **TASK.LM.HANDLE(**<*lmmagic*>**)**

Returns the “install handle” of the load module.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.LM.FILENAME() File name for load module

Syntax: **TASK.LM.FILENAME("**<*loadmodule*>**")**

Returns the file name for the specified load module.

Parameter Type: String (*with* quotation marks).

Return Value Type: String.

TASK.LM.RELOCINFO() Relocation information for load module

Only OSE 5.x!

Syntax: **TASK.LM.RELOCINFO("**<*loadmodule*>**")**

Returns the relocation information for the specified load module, to be used with **Data.LOAD.Elf**.

Parameter Type: String (*with* quotation marks).

Return Value Type: String.

TASK.LM.RELOCITER() Relocation information of index section

Only OSE 5.x!

Syntax: **TASK.LM.RELOCITER("**<*loadmodule*>**",** <*index*>**)**

Returns the relocation information of the <*index*> section for the specified load module, to be used with **Data.LOAD.Elf** (relocinfo as iteration).

Parameter and Description:

<*loadmodule*>	**Parameter Type**: String (*with* quotation marks).
<*index*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: String.

•