



OS Awareness Manual PikeOS

Release 02.2025

TRACE32 Online Help

TRACE32 Directory

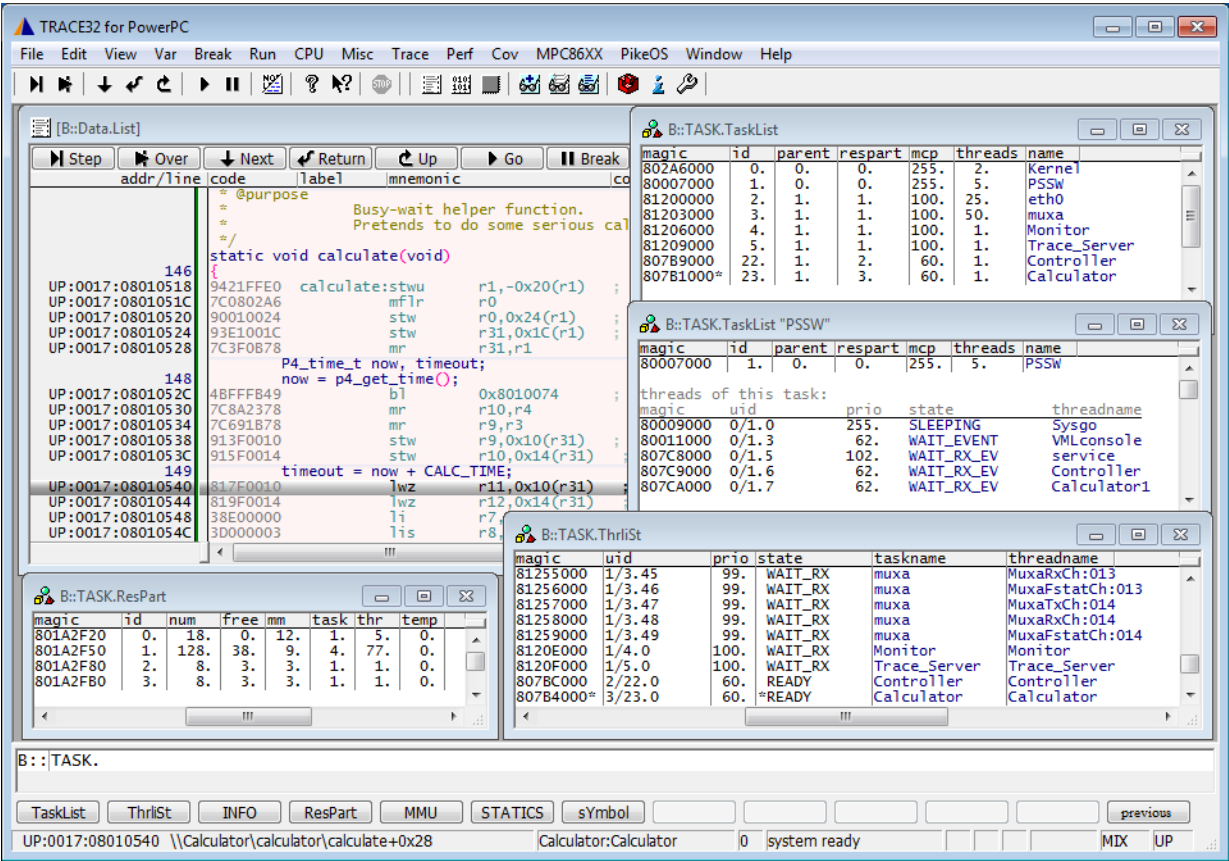
TRACE32 Index

TRACE32 Documents	
OS Awareness Manuals	
OS Awareness Manual PikeOS	1
Overview	5
Terminology	5
Brief Overview of Documents for New Users	6
Supported Versions	6
Configuration	7
Quick Configuration Guide	7
Hooks and Internals in PikeOS	8
Features	9
Display of Kernel Resources	9
Task Stack Coverage	9
Task-Related Breakpoints	10
Task Context Display	12
MMU Support	14
Space IDs	14
MMU Declaration	14
Scanning System and Processes	17
Symbol Autoloader	17
SMP Support	19
POSIX Personality	19
APEX Personality	20
Linux Personality	20
Dynamic Task Performance Measurement	20
Task Runtime Statistics	21
Function Runtime Statistics	22
PikeOS specific Menu	23
Debugging PikeOS Components	24
PikeOS Kernel	24
Downloading the PikeOS Image	24
Debugging the Kernel Startup	25
Debugging the Kernel	25
System Extensions	25

System Extensions in V3.x		25
System Extensions in V4.x		26
User Tasks		26
Debugging the Task		27
Start Debugging a Task from main		27
Debugging PikeOS Threads		28
POSIX		28
Configuring POSIX Awareness		28
Multiple POSIX Personalities with multiple awareness		29
APEX		30
Configuring APEX Awareness		30
Multiple APEX Personalities with multiple awareness		31
Multiple APEX Personalities with combined awareness		31
ELinOS		33
Linux Kernel		34
Linux Kernel Modules		34
Linux Processes		35
Linux Libraries		35
PikeOS Commands		36
EXTension.AXInfo	Display APEX information	36
EXTension.AXProcess	Display APEX processes	36
EXTension.ELModule	Display ELinOS modules	37
EXTension.ELProcess	Display ELinOS processes	37
EXTension.ELThread	Display ELinOS threads	37
EXTension.PXThread	Display POSIX threads	38
TASK.DrvList	Display system information	39
TASK.INFO	Display system information	40
TASK.Option	Set awareness options	40
TASK.ResPart	Display resource partitions	41
TASK.TaskAdspace	Display task address space	41
TASK.TaskFile.ADD	Map file name to task name	41
TASK.TaskFile.view	Display file name to task name mapping	42
TASK.TaskList	Display 'PikeOS' tasks	43
TASK.ThrliSt	Display threads	44
PikeOS PRACTICE Functions		45
TASK.CONFIG()	OS Awareness configuration information	45
TASK.TASK.MAGIC()	magic number of task	45
TASK.TASK.ID()	ID of task	45
TASK.TASK.NAME()	Name of task	46
TASK.TASK.ID2NAME()	Convert task ID to name	46
TASK.TASKNAME2ID()	Convert task name to ID	46
TASK.TASKFILE()	Symbol file name of task	46
EXT.AXPROCESS.THREAD()	PikeOS thread of APEX process	47

EXT.AXPROCESS.THREAD2()	PikeOS thread of APEX process	47
EXT.ELINOS.SPACEID()	Space ID of ELinOS personality	47
EXT.ELPROCESS.NAME()	Name of ELinOS process	47
EXT.ELLIBRARY.ADDRESS()	Load address of ELinOS library	48
EXT.ELLIBRARY.SPACEID()	Space ID of ELinOS library	48
EXT.ELLIBRARY.NAME()	Name of ELinOS library	48
EXT.ELMODULE.MAGIC()	Module magic number of ELinOS module	48
EXT.ELMODULE.NAME()	Name of ELinOS module	49
EXT.ELMODULE.SECADDR()	Section address of ELinOS module	49

Overview



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

Terminology

The PikeOS kernel uses the term “task” for a collection of threads in an address space, and “thread” for an execution unit (a PikeOS “task” refers to a “process” in PSSW). If not otherwise specified, the TRACE32 term “task” corresponds to PikeOS “threads”, while a PikeOS “task” corresponds to a “space ID” in TRACE32. Please consider carefully the different meanings of “task” in both environments.

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 PikeOS is supported for the following versions:

- PikeOS 1.4, 2.2, 3.1 on PowerPC
These versions are supported with a previous PikeOS awareness not covered by this document. Contact Lauterbach for more information.
- PikeOS 3.3, 3.4 and 3.5 on ARM, PowerPC and x86
- PikeOS 4.0, 4.1 and 4.2 on ARM, PowerPC and x86
- PikeOS 5.0 and 5.1 on ARM and PowerPC

Configuration

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

Automatic configuration tries to locate the PikeOS internals automatically. For this purpose, the kernel symbols must be loaded and accessible at any time the OS Awareness is used (see also “[Hooks & Internals](#)”).

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**.

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:

```
TASK.CONFIG ~/demo/<arch>/kernel/pikeos/<version>/pikeos.t32
```

(Note: “`~~`” refers to the TRACE32 installation directory)

Note that the symbols of the PikeOS kernel must be loaded into the debugger. See [Hooks & Internals](#) for details on the used symbols. See also the examples in the demo directories “`~/demo/<arch>/kernel/pikeos`”.

Quick Configuration Guide

To fully configure the OS Awareness for PikeOS, please use one of the demo startup scripts as template. Find the templates in the directory `~/demo/<arch>/kernel/pikeos`.

“pikeos.cmm” shows a small setup for a simple static PikeOS system, based on the “hello” demo project.

“pikeos-ipc.cmm” shows a setup for a dynamic PikeOS system, based on the “inter-partition-communication” demo project.

“pikeos-linux.cmm” shows a setup for a PikeOS system with ELinOS partition.

Follow this roadmap:

1. Carefully read the demo start-up scripts (`~/demo/<arch>/kernel/pikeos`).
2. Make a copy of the appropriate script. 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).

Now you can access the PikeOS extensions through the menu.

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

Hooks and Internals in PikeOS

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols of the PSP kernel. This means that every time, when features of the OS Awareness are used, the symbols of the kernel must be available and accessible.

PikeOS version 3:

You can find the kernel symbol files in the directory `$(PIKEOS_PSP_DIR)/$(PIKEOS_PSP)/objects` of your PikeOS installation. Use the kernel file that your integration project refers to in “project.xml.conf”. If e.g. “Ukernel” equaled “smp-tracesys”, then use the file “psp-kernel-smp-tracesys”. Load the symbols with the command:

```
Data.LOAD.Elf <path_to_psp>/objects/psp-kernel-smp-tracesys /NoCODE /NoClear
```

PikeOS version 4 and 5:

You can find the kernel symbol files in the PikeOS installation directory, subdirectory `target/<arch>/<proc>/object/bsp/<board>`. Check the BSP directory in the “project.mk” file of your integration project, settings “PIKEOS_TARGET_FILES” and “PIKEOS_DTS_DIR”. Use the kernel file that your integration project refers to in “project.rbx”. See the “psp” attribute of the “romimage” element. If e.g. “psp” refers to “kernel-nodebug-tracesys-smp.bin”, then load the according ELF file with the command:

```
Data.LOAD.Elf <path_to_board>/kernel-nodebug-tracesys-smp.elf /NoCODE /NoClear
```

Please also look at the demo startup script `pikeos.cmm`, how to load the kernel symbols and the symbols of your application.

Features

The OS Awareness for PikeOS supports the following features.

Display of Kernel Resources

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

TASK.INFO	System information
TASK.ResPart	Resource partitions
TASK.TaskList	Tasks
TASK.ThrliSt	Threads

For a description of the commands, refer to chapter “[PikeOS 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.

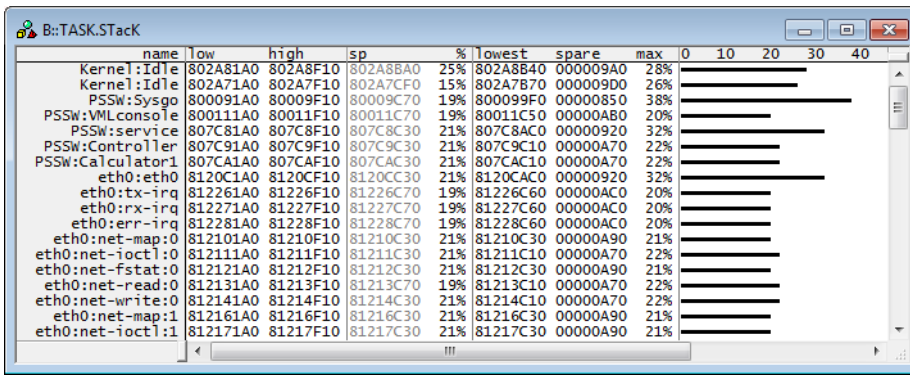
Task Stack Coverage

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.



NOTE:

PikeOS does *not* save the user space stack address and range. The TASK.STack.view command by default *only* shows the system stack range for the PikeOS threads.

Usually the user stack range is specified with the P4_STACK macro to the p4_thread_arg() call. If you want to cover this stack range, you have to initialize the stack with a predefined pattern (usually zero) and add the stack range manually using the TASK.STack.Add command.

PikeOS does *not* initialize the system stack with a predefined pattern. It is up to the system integrator or application programmer to ensure stack initialization for a proper stack coverage analysis.

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.

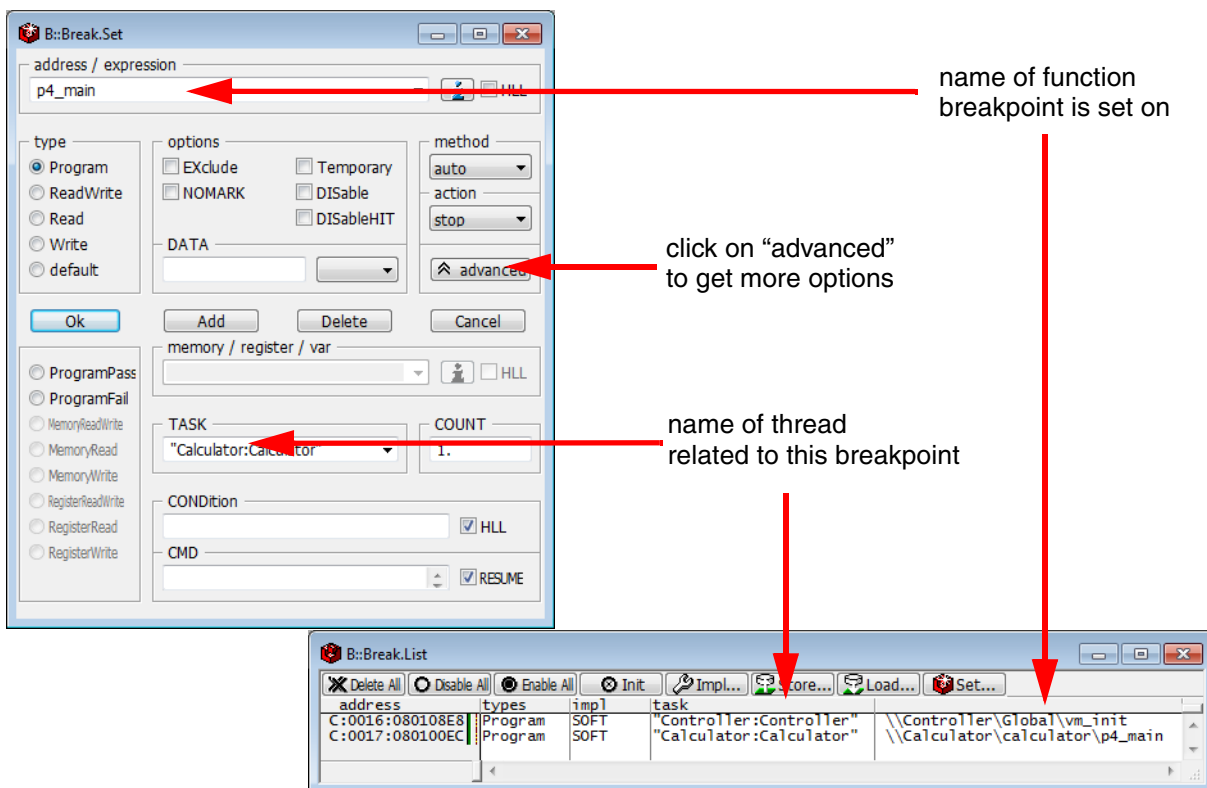
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

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.

Example for a task-related breakpoint, equivalent to the **Break.Set** <function> /TASK <task> 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.

```

B::Frame /Task "Controller:Controller"
Up Down [x] Args [x] Locals [x] Caller Task: "Controller:Controller"
-000|| p4_ipc(asm)
-001|| vm_call_svc(asm)
-002|| vm_cprintf(asm)
-003|| p4_main()
    inport = ((id = 0), (id = 0), (id = 0))
    outport = ((id = 1), (id = 0), (id = 0))
    portconfig = (max_nb_messages = 32, max_msg_size = 256, direction = VM
    * i = 3
    * msgsize = 4
    * counter = 0x60000000
    * task_configured = (1, 0, 0)
    * rc = VM_E_OK
    * sleep_time = 3000000000
    * fd = (i = (size = 0, pos = 0, type = VM_FILE_T_DIR, oflags = 0, vflags =
    * taskid = (23, 0, 0)
    * portname = "ContrIn2"
    * my_uid = 2108416
    * my_taskid = 22
    * msgbuf = (23, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    }
    }
    }
    /* Show the number of calculations performed by each rem
    vm_cprintf("Calculations performed:");
-004|| p4_entry()
    p4_main();
    end of frame

```

Call stack frame of a thread, showing the calling line and local variables.

To provide full debugging possibilities, the Debugger has to know, how virtual addresses are translated to physical addresses and vice versa. All **MMU** and **TRANSlation** commands refer to this necessity.

Space IDs

Different PikeOS Tasks may use identical virtual addresses. To distinguish those addresses, the debugger uses an additional identifier, the so-called space ID (memory space ID) that specifies to which virtual memory space the address refers. The command **SYStem.Option.MMUSPACES ON** enables the use of the space ID. For all PikeOS Tasks using the kernel address space, the space ID is zero. For Tasks using their own address space, the space ID corresponds to the task ID. Threads of a particular task use the memory space of the invoking task. Consequently threads have the same space ID as the task they belong to.

You may scan the whole system for space IDs using the command **TRANSlation.ScanID**. Use **TRANSlation.ListID** to get a list of all recognized space IDs.

The function **task.taskid("<task>")** returns the ID for a given PikeOS Task.

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
PIKEOS.E500	PIKEOS specific format for PowerPC e500 core with 128-bit PTEs (formerly named PIKEOSE5). Works for PikeOS 4.1 and older. For e500 cores with PikeOS 4.2 and newer use E500MC format.*/
PIKEOS.E500MC	PIKEOS specific format for PowerPC e500mc core (PPC64 only) with 32-bit PTEs. Can also be used with PikeOS 4.2 and newer on PPC32 e500 cores.*/
PIKEOS.E500MC4G	PIKEOS specific format for PowerPC e500mc core addressing 4GB of memory. Has no common address range with 32-bit PTEs.*/
PIKEOS.E5500	PIKEOS specific format for PowerPC e5500 core with 64-bit PTEs
PIKEOS.OEA	PIKEOS specific format for PowerPC core (formerly named PIKEOS) for the OEA architecture */
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> is currently unused. Specify zero.

<logical_kernel_address_range>

<logical_kernel_address_range> specifies the virtual to physical address translation of the kernel address range.

<physical_kernel_address>

<physical_kernel_address> specifies the physical start address of the kernel.

The kernel code, which resides in the kernel space, can be accessed by any PikeOS Task, regardless of the current space ID. Use the command **TRANSlation.COMMON** to define the complete address range that is addressed by the kernel as commonly used area.

Enable the debugger's table walk with **TRANSlation.TableWalk ON**, and switch on the debugger's MMU translation with **TRANSlation.ON**.

Example: RAM at physical address 0x0, a typical MMU declaration looks like:

```
MMU.FORMAT PIKEOS 0 0x80000000--0xbfffffff 0x0
TRANSlation.COMMON 0x80000000--0xffffffff ; common area
TRANSlation.TableWalk ON
TRANSlation.ON
```

Please see also the sample scripts in the ~/demo directory.

To access the different process spaces correctly, the debugger needs to know the address translation of every virtual address it uses. You can either scan the MMU tables and place a copy of them into the debugger's address translation table, or you can use a table walk, where the debugger walks through the MMU tables each time it accesses a virtual address.

The command **MMU.SCAN** *only* scans the contents of the current processor MMU settings. Use the command **MMU.SCAN ALL** to go through all space IDs and scan their MMU settings. Note that on some systems, this may take a long time. In this case you may scan a single PikeOS Task (see below).

To scan the address translation of a specific PikeOS Task, use the command **MMU.SCAN TaskPageTable <task_id>**. This command scans the space ID of the specified task. To scan the kernel space, use:

```
MMU.SCAN TaskPageTable 0.
```

TRANSlation.List shows the address translation table for all scanned space IDs.

If you set **TRANSlation.TableWalk ON**, the debugger tries first to look up the address translation in its own table (**TRANSlation.List**). If this fails, it walks through the target MMU tables to find the translation for a specific address. This feature eliminates the need of scanning the MMU each time it changes, but walking through the tables for each address may result in a very slow reading of the target. The address translations found with the table walk are only temporarily valid (i.e. not stored in **TRANSlation.List**), and are invalidated at each **Go** or **Step**.

See also chapter “**Debugging PikeOS Kernel and User Processes**”.

Symbol Autoloader

The OS Awareness for PikeOS contains a “Symbol Autoloader”, which automatically loads symbol files corresponding to executed tasks or libraries. The autoloader maintains a list of address ranges, corresponding to PikeOS components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader (e.g. via **List**), the debugger invokes the command necessary to load the corresponding symbols to the appropriate addresses (including relocation). This is usually done via a PRACTICE script.

In order to load symbol files, the debugger needs to be aware of the currently loaded components. This information is available in the kernel data structures and can be interpreted by the debugger. The command **sYmbol.AutoLOAD.CHECK** defines, *when* these kernel data structures are read by the debugger (only on demand or after each program execution).

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

The loaded components can change over time, when processes are started and stopped and libraries are loaded or unloaded. The command **sYmbol.AutoLOAD.CHECK** configures the strategy, when to “check” the kernel data structures for changes in order to keep the debugger's information regarding the components up-to-date.

Without parameters, the **sYmbol.AutoLOAD.CHECK** command *immediately* updates the component information by reading the kernel data structures. This information includes the component name, the load address and the space ID and is used to fill the autoloader list (shown via **sYmbol.AutoLOAD.List**).

With **sYmbol.AutoLOAD.CHECK ON**, the debugger *automatically* reads the component information *each time the target stops executing* (even after assembly steps), having to assume that the component information might have changed. This significantly slows down the debugger which is inconvenient and often superfluous, e.g. when stepping through code that does not load or unload components.

With the parameter **ONGO**, the debugger checks for changed component info like with **ON**, but *not when performing single steps*.

With **sYmbol.AutoLOAD.CHECK OFF**, no automatic read is performed. In this case, the update has to be triggered manually when considered necessary by the user.

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

When configuring the OS Awareness for PikeOS, set up the symbol autoloader with the following command:

sYmbol.AutoLOAD.CHECKPIKEOS "<action>"

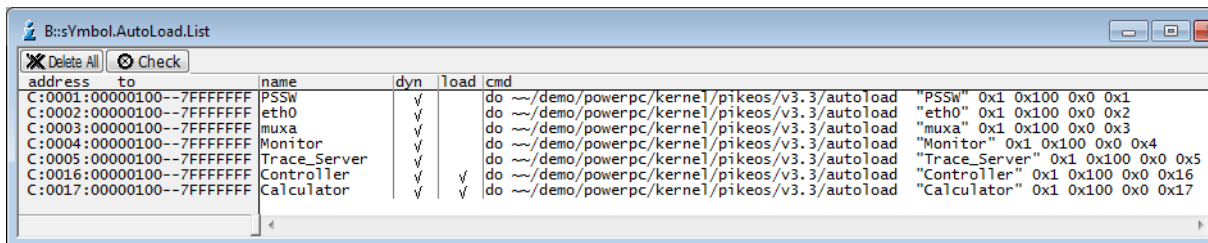
<action> to take for symbol load, e.g. "DO autoload.cmm "

The command **sYmbol.AutoLOAD.CHECKPIKEOS** is used to define which action is to be taken, for loading the symbols corresponding to a specific address. The action defined is invoked with specific parameters (see below). With PikeOS, the pre-defined action is to call the script `~/demo/<arch>/kernel/pikeos/<version>/autoload.cmm`.

NOTE: The action parameter needs to be written with quotation marks (for the parser it is a string).

Note that *defining* this action, does not cause its execution. The action is executed on demand, i.e. when the address is actually accessed by the debugger e.g. in the **List** or **Trace.List** window. In this case the autoloader executes the <action> appending parameters indicating the name of the component, its type (process, library), the load address and space ID.

For checking the currently active components use the command **sYmbol.AutoLOAD.List**. Together with the component name, it shows details like the load address, the space ID, and the command that will be executed to load the corresponding object files with symbol information. Only components shown in this list are handled by the autoloader.



address	to	name	dyn	load	cmd
C:0001:00000100--7FFFFFFF		PSSW	✓		do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "PSSW" 0x1 0x100 0x0 0x1
C:0002:00000100--7FFFFFFF		eth0	✓		do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "eth0" 0x1 0x100 0x0 0x2
C:0003:00000100--7FFFFFFF		muxa	✓		do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "muxa" 0x1 0x100 0x0 0x3
C:0004:00000100--7FFFFFFF		Monitor	✓		do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "Monitor" 0x1 0x100 0x0 0x4
C:0005:00000100--7FFFFFFF		Trace_Server	✓		do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "Trace_Server" 0x1 0x100 0x0 0x5
C:0016:00000100--7FFFFFFF		Controller	✓	✓	do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "Controller" 0x1 0x100 0x0 0x16
C:0017:00000100--7FFFFFFF		Calculator	✓	✓	do ~/demo/powerpc/kernel/pikeos/v3.3/autoload "Calculator" 0x1 0x100 0x0 0x17

NOTE:

The GNU compiler generates different code if an application is built with debug info (option “-g”), even if the optimization level is the same. Ensure that you *always* use the debug version on *both* sides, the target where you start the application, and the debugger where you load the symbol file.

In PikeOS, task names are often different from their (symbol) file names. For the symbol autoloader to automatically find the appropriate symbol file for a task, you can map task names to file names with the command **TASK.TaskFile.ADD**.

SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

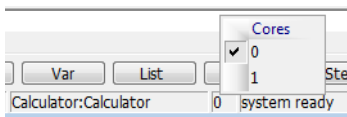
An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application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



POSIX Personality

The OS Awareness for PikeOS contains an additional awareness for a POSIX Personality. For a detailed description see the chapter “Debugging PikeOS Components”, “**POSIX**”.

APEX Personality

The OS Awareness for PikeOS contains an additional awareness for an APEX Personality. For a detailed description see the chapter “Debugging PikeOS Components”, “[APEX](#)”.

Linux Personality

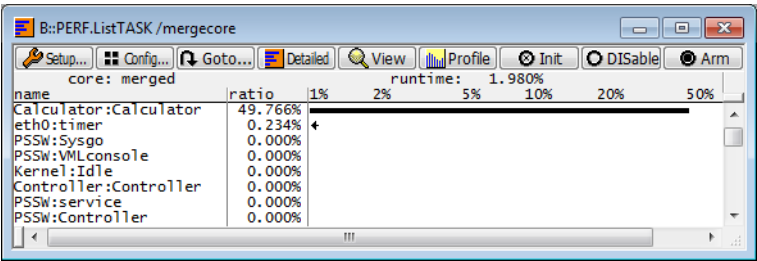
The OS Awareness for PikeOS has a built-in detection and an additional awareness for a Linux Personality based on ELinOS. For a detailed description see the chapter “Debugging PikeOS Components”, “[ELinOS](#)”.

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).



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)”.

On ARM architectures, PikeOS serves the ContextID register. In PikeOS version up to 3.4, only the address space ID (ASID) of the process is written. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide performance information of threads. In PikeOS versions since 3.5, the kernel writes the ASID and the thread ID into the ContextID register. This is sufficient for thread runtime analysis as well as function runtime analysis.

If your PikeOS version does *not* write the thread ID into the ContextID register, inform the PikeOS awareness about this by switching off the THRCTX option:

TASK.Option THRCTX OFF ; for PikeOS versions up to 3.4

All kernel activities up to the thread switch are added to the calling thread.

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:

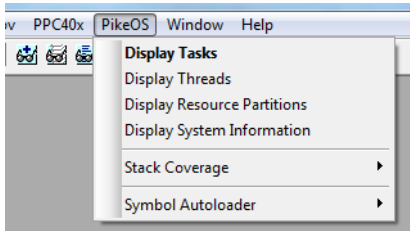
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat runtime timechart

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

PikeOS specific Menu

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

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



- The **Display** menu items launch the kernel resource display windows. See chapter “[Display of Kernel Resources](#)”.
- The **Stack Coverage** submenu starts and resets the PikeOS specific [stack coverage](#) and provides an easy way to add or remove tasks from the [TASK.Stack.view](#) window.
- Use the **Symbol Autoloader** submenu to configure the symbol autoloader. See also chapter “[Symbol Autoloader](#)”.
 - **List Components** opens a [sYmbol.AutoLOAD.List](#) window showing all components currently active in the autoloader.
 - **Check Now!** performs a [sYmbol.AutoLOAD.CHECK](#) and reloads the autoloader list.
 - **Set Loader Script** allows you to specify the script that is called when a symbol file load is required. You may also set the automatic autoloader check.

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 thread switches (if any) or thread switches together with default display.
- The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime statistics or statistics on task states, if a trace is available. See also chapter “[Task Runtime Statistics](#)”.

Debugging PikeOS Components

PikeOS runs on virtual address spaces. The kernel uses a static address translation. Each user task gets its own user address space when loaded, mapped to any physical RAM area that is currently free. Due to this address translations, debugging the PikeOS kernel and the user tasks requires some settings to the debugger.

To distinguish those different memory mappings, TRACE32 uses space IDs, defining individual address translations for each ID. The kernel itself is attached to the space ID zero. Each PikeOS Task that has its own memory space gets a space ID that corresponds to its task ID. Threads get the space ID of the task they belong to.

See also chapter “[MMU Support](#)”.

PikeOS Kernel

The PikeOS system builder generates an image that contains the startup code, the kernel and any given application. The file format depends on the system settings, usually it is in ELF or binary format.

Additionally, the PikeOS Awareness needs the symbols of the kernel. Please see section “[Hooks & Internals](#)” how to find the symbol files of the kernel.

Downloading the PikeOS Image

If you start the PikeOS image from Flash, or if you download the image using a bootloader, do this as you are doing it without debugging.

If you want to download the PikeOS image using the debugger, you have to watch about the file format. If the generated image is in ELF format, simply download this to the target. If the image is in binary format, you have to tell the debugger at which address to download it. Please also see the example scripts.

Examples:

```
Data.Load.Elf ipc-mpc8641hpcn-elf           ; downloading ELF image
Data.Load.Binary ipc-mpc8641hpcn 0x00100000 ; downloading binary
```

To create the image in ELF format, configure the “PikeOS boot strategy” in the integration project to “ELF”.

When downloading the kernel via the debugger, remember to set startup parameters that the kernel requires, before booting the kernel. Usually the bootloader passes these parameters to the image.

Debugging the Kernel Startup

The kernel image starts with MMU switched off, i.e. the processor operates on physical addresses. However, all symbols of the kernel file are virtual addresses. If you want to debug this (tiny) startup sequence, you have to load the kernel symbols and relocate them to physical address.

As soon as the processor MMU is switched on, you have to reload the symbol to its virtual addresses. See the next chapter on how to debug the kernel in the virtual address space.

Debugging the Kernel

For debugging the kernel itself, and for using the PikeOS awareness, you have to load the virtual addressed symbols of the kernel into the debugger. The kernel symbol file contains all addresses in virtual format, so it's enough to simply load the file, e.g.:

```
Data.Load.Elf psp-kernel-smp-tracesys /NoCODE
```

See also “[Hooks & Internals](#)” how to find the correct kernel symbol file for your image.

System Extensions

“System Extensions” are functional extensions linked to the PikeOS base software (PSSW), which runs as a separate task with ID 1. Debugging system extensions depends on the used PikeOS version.

System Extensions in V3.x

In PikeOS V3.x, system extensions are dynamically loaded and linked to the PSSW when booting. Unfortunately, there is no way to determine the load address from the started PikeOS image. To get the start address, you have to examine the startup log.

In the integrator project, open the <project.rbx> file and add

```
<tag key="UK_LOG_LEVEL" value="5" />
```

to enable debug logging of the PSSW. Please note that this file is generated each time you modify your integrator project, so check this setting after each project modification.

When PSSW starts, it prints out a lot of debug info. Watch for the loading of the system extension. For example:

```
<SSW DEBUG INFO> System Extension "simple-pp" resides at virtual base 0x46A260
```

Use this address to load the symbols of the system extension. The demo directory contains a script “loadse.cmm” that helps loading the symbol file of a system extension. Call this script with the load address mentioned in the debug log.

```
DO ~/demo/<arch>/kernel/pikeos/<version>/loadse.cmm \  
    <system_extension> <load_address>
```

E.g.:

```
DO ~/demo/powerpc/kernel/pikeos/v3.x/loadse.cmm \  
    myworkspace/simple-pp.unstripped 0x46A260
```

System Extensions in V4.x

In PikeOS V4.x, system extensions are statically linked to the PSSW when building a fusion project. To debug system extensions, simply load the symbols of the newly built PSSW.

If you want to debug startup routines of the system extensions (e.g. the install routine), load the symbols of the PSSW before it starts (to space ID 1) and set an onchip breakpoint onto the routine. E.g.:

```
Data.LOAD.Elf myworkspace/myfusionproject/mypssw1-normal.unstripped \  
    1:0 /NoCODE /NoClear /name PSSW  
Go simple_pp_install /Onchip
```

After PikeOS booted, the symbol autoloader may be used to load the PSSW symbols and to debug the system extension routines:

```
sYmbol.SourcePATH myworkspace/myfusionproject  
TASK.TaskFile.ADD "PSSW" "mypssw1-normal"  
sYmbol.AutoLOAD.CHECK  
sYmbol.AutoLOAD.TOUCH "PSSW"
```

User Tasks

Each user task in PikeOS gets its own virtual memory space. To distinguish the different memory spaces, the debugger assigns a “space ID”, which correlates to the task ID. Using this space ID, it is possible to address a unique memory location, even if several PikeOS use the same virtual address.

Note that at every time the PikeOS awareness is used, it needs the kernel symbols. Please see the chapters above on how to load them. Hence, load all task symbols with the option /NoClear to preserve the kernel symbols.

Ensure that you load the symbol file containing debug information, i.e. the “unstripped” version.

Debugging the Task

To correlate the symbols of a user task with the virtual addresses of this task, it is necessary to load the symbols into this space ID.

Manually Load Task Symbols:

For example, if you've got a task called "hello" with the task ID 12 (the dot specifies a decimal number!):

```
Data.LOAD Elf hello.unstripped 12.:0 /NoCODE /NoClear
```

The task ID of a task may also be calculated by using the PRACTICE function **TASK.TASKID()** (see chapter "[PikeOS PRACTICE Functions](#)").

Using the Symbol Autoloader:

If the symbol autoloader is configured (see chapter "[Symbol Autoloader](#)"), the symbols will be automatically loaded when accessing an address inside the process. You can also force the loading of the symbols of a process with

```
sYmbol.AutoLOAD.CHECK  
sYmbol.AutoLOAD.TOUCH "hello"
```

In PikeOS, task names are often different from their (symbol) file names. For the symbol autoloader to automatically find the appropriate symbol file for a task, you can map task names to file names with the command **TASK.TaskFile.ADD**.

Using the Menus:

Select "Display Tasks", right click on the "magic" of a process, and select "Load task symbols".

Start Debugging a Task from main

If you want to debug your task right from the beginning at "p4_main()", you have to load the symbols *before* starting the process. This is a tricky thing because you have to know the space ID, which is assigned first at the task startup.

The script "app_debug.cmm" in the ~/demo directory automates this step. Call the script with the task name *first*, *then* start the Task within PikeOS. The script waits for the task to be started, loads the symbols and halts the task at p4_main(). If the symbol file name is different to the task name, you can add the symbol file name as a second parameter. Examples:

```
; wait for "hello" to be started and load symbol file:  
DO ~/demo/<arch>/kernel/pikeos/<version>/app_debug hello  
  
; wait for "hello" and load "filehello" as symbol file:  
DO ~/demo/<arch>/kernel/pikeos/<version>/app_debug hello filehello
```

Debugging PikeOS Threads

PikeOS threads share the same virtual memory of the parent task. The OS Awareness for PikeOS assigns one space ID for all threads that belong to a specific task. It is sufficient, to load the debug information of this task only once (onto its space ID) to debug all threads of this task. See chapter “[Debugging the Task](#)” for loading the tasks symbols.

The **TASK.ThrliSt** window shows which thread is currently running (marked with a star).

POSIX

The OS Awareness for PikeOS contains an additional awareness for a POSIX Personality. It allows to view POSIX threads and their states.

Configuring POSIX Awareness

To configure the POSIX awareness:

- Load the symbols of the PikeOS task that contains the POSIX personality.
- Load the POSIX awareness specifying the task ID.

Example:

```
; Load the symbols of the PikeOS task with the POSIX personality,  
; e.g. "scheduling"  
sYmbol.AutoLOAD.TOUCH "scheduling"  
  
; read the task ID of the POSIX personality  
&posixid=task.taskid("scheduling")  
  
; load the POSIX awareness, e.g. for ARM architecture  
EXTension.LOAD ~/demo/arm/kernel/pikeos/posix/posix.t32 &posixid  
Menu.ReProgram ~/demo/arm/kernel/pikeos/posix/posix.men
```

The above sequence is put into a helper script `~/demo/<arch>/kernel/pikeos/<version>/load_posix.cmm`. Specify the PikeOS task name as parameter to the script.

Example: Load POSIX awareness for PikeOS task named “scheduling”:

```
DO ~/demo/arm/kernel/pikeos/v5.x/load_posix.cmm scheduling
```

It is necessary to compile the POSIX application with debug information enabled for the POSIX code. In your application's Makefile, add "POSIX_CFLAGS += -g -O0". "-g" is required for the debug information, "-O0" is recommended to disable optimization and easier debugging.

If the "POSIX_DEBUG" option is set to "true" in your project configuration, the "debug" libraries of POSIX are used which usually also contain a GDB agent. This agent may set a "breakpoint interrupt" at startup of a POSIX personality that can cause a shutdown of the personality. As TRACE32 doesn't need this GDB breakpoint, patch it to do nothing, right at the entry point. E.g. in ARM environments, you can use:

```
Data.Assemble gdb_breakpoint bx r14
```

After loading the POSIX awareness, you've got additional features for the POSIX personality:

EXTension.PXThread

Show all POSIX threads of the POSIX personality

An additional "POSIX" menu provides easy access to the features.

Debugging POSIX threads works the same way as debugging standard PikeOS applications.

Multiple POSIX Personalities with multiple awareness

If you have multiple POSIX Personalities simply use the loading script for several personalities.

Example: Load POSIX awareness for PikeOS tasks named "posix" and "scheduling":

```
DO ~/demo/arm/kernel/pikeos/v5.x/load_posix.cmm posix
DO ~/demo/arm/kernel/pikeos/v5.x/load_posix.cmm scheduling
```

You will get an own menu for each of the personalities, named like the task name. These menus provide easy access to the individual personalities.

Access the individual commands with an infix to the EXTension command:

EXTen-

sion.<name>.PXThread

Show all POSIX thread of the POSIX personality <name>

The OS Awareness for PikeOS contains an additional awareness for an APEX Personality. It allows to view APEX processes and their states.

Configuring APEX Awareness

To configure the APEX awareness:

- Load the symbols of the PikeOS task that contains the APEX personality.
- Load the symbol file of the used APEX OS library to the space ID of the PikeOS task containing the APEX personality (same as task ID). The used APEX OS can be read out of your project4.rbx file, e.g. "apex_os_devel". The symbols must be renamed to "apex_os".
- Load the APEX awareness specifying the task ID.

Example:

```
; Load the symbols of the PikeOS task with the APEX personality,
; e.g. "apex"
sYmbol.AutoLOAD.TOUCH "apex"

; read the task ID of the APEX personality
&apexid=task.taskid("apex")

; load the symbols of the used APEX OS library
Data.LOAD.Elf apex_os_devel.unstripped &apexid:0 /name apex_os \
    /NoCODE /NoClear

; load the APEX awareness, e.g. with PikeOS v5 on ARM architecture
EXTension.LOAD ~/demo/arm/kernel/pikeos/v5.x/apex/apex.t32 &apexid
Menu.ReProgram ~/demo/arm/kernel/pikeos/v5.x/apex/apex.men
```

The above sequence is put into a helper script `~/demo/<arch>/kernel/pikeos/<version>/load_apex.cmm`. Specify the PikeOS task name and kernel type as parameter to the script.

Example: Load APEX awareness for PikeOS task named "apex" with kernel type "devel":

```
DO ~/demo/arm/kernel/pikeos/v5.x/load_apex.cmm apex devel
```

After loading the APEX awareness, you've got additional features for the APEX personality:

EXTension.AXProcess

Show all APEX processes of the APEX personality

An additional "APEX" menu provides easy access to the features.

Debugging APEX processes works the same way as debugging standard PikeOS applications.

Multiple APEX Personalities with multiple awareness

If you have multiple APEX Personalities simply use the loading script for several personalities.

Example: Load APEX awareness for PikeOS tasks named “apex” and “suspend” with kernel type “devel”:

```
DO ~/demo/arm/kernel/pikeos/v5.x/load_apex.cmm apex devel
DO ~/demo/arm/kernel/pikeos/v5.x/load_apex.cmm suspend devel
```

You will get an own menu for each of the personalities, named like the task name. These menus provide easy access to the individual personalities.

Access the individual commands with an infix to the EXTension command:

EXTension.<name>.AXProcess

Show all APEX processes of the APEX personality <name>

Multiple APEX Personalities with combined awareness

If several APEX personalities use the same APEX OS library, one awareness can serve for multiple APEX personalities. To configure the APEX awareness for multiple personalities:

- Load the symbols of the PikeOS tasks that contain the APEX personalities
- Load the symbol file of the used APEX OS library to *all* space IDs of the PikeOS tasks containing the APEX personalities (same as task IDs). The used APEX OS can be read out of your project4.rbx file, e.g. “apex_os_devel”. The symbols must be renamed to “apex_os”.
- Load the APEX awareness specifying the task IDs of the APEX personalities.

Example:

```
; Load the symbols of the PikeOS task with the APEX personality,  
; e.g. "apex" and "suspend"  
sYmbol.AutoLOAD.TOUCH "apex"  
sYmbol.AutoLOAD.TOUCH "suspend"  
  
; read the task IDs of the APEX personality  
&apexid1=task.taskid("apex")  
&apexid2=task.taskid("suspend")  
  
; load the symbols of the used APEX OS library to all used IDs  
Data.LOAD.Elf apex_os_devel.unstripped &apexid1:0 /name apex_os \  
/NoCODE /NoClear  
Data.LOAD.Elf apex_os_devel.unstripped &apexid2:0 /name apex_os \  
/NoCODE /NoClear  
  
; load the APEX awareness, e.g. with PikeOS v5 on ARM architecture  
EXTension.LOAD ~/demo/arm/kernel/pikeos/v5.x/apex/apex.t32 \  
    &apexid1 &apexid2  
Menu.ReProgram ~/demo/arm/kernel/pikeos/v5.x/apex/apex.men
```

An additional “APEX” menu provides easy access to the features.

The OS Awareness for PikeOS contains an automatic detection and an additional awareness for an ElinOS Linux Personality. It enables debugging the Linux kernel, kernel modules, user processes and libraries.

To configure the ElinOS awareness, load the symbols of the PikeOS task that contains the ElinOS personality. After this, load the ElinOS awareness files:

```
; Load the symbols of the PikeOS task with the ElinOS personality
; named "P4Linux"

sYmbol.AutoLOAD.TOUCH "P4Linux"

; read the task ID of the ElinOS personality

&elinosid=task.taskid ("P4Linux")

; and load the ElinOS awareness, e.g. for ARM architecture

EXTension.LOAD ~/demo/arm/kernel/pikeos/elinos/elinos.t32 &elinosid
Menu.ReProgram ~/demo/arm/kernel/pikeos/elinos/elinos.men

; re-check the symbol autoloader to recognize Linux processes

sYmbol.AutoLOAD.CHECK
```

The ElinOS awareness can only work on one ElinOS personality at a time. If you have several ElinOS personalities and want to work on a different personality, reload the ElinOS awareness with the appropriate task ID.

After loading the ElinOS awareness, you've got additional features for the ElinOS personality:

Debugging the [Linux kernel](#)

Debugging [Linux kernel modules](#)

Debugging [Linux processes](#)

Debugging [Linux libraries](#)

EXTension.ELProcess Show processes of the ElinOS personality

EXTension.ELThread Show threads of the ElinOS personality

EXTension.ELModule Show kernel modules of the ElinOS personality

The Linux kernel itself runs as a PikeOS task named “P4Linux”. In order to debug the kernel (or contents of it like built-in drivers), you need to load the symbols of the Linux kernel. Ensure that you built the kernel with full debug info enabled (CONFIG_DEBUG_INFO=y). The symbol file then is named “vmlinux” and resides in the linux directory of your ELinOS project. Use the symbol autoloader to load this file as symbol file for the P4Linux task.

After loading the symbols of the “P4Linux” task, you can debug the Linux kernel as any other PikeOS task.

Linux Kernel Modules

To debug already loaded kernel modules, open an [EXTension.ELModule](#) window and right-click on the “magic” of the module. Select “Load module symbols” from the local menu to load the symbol file of the selected module. After this, you can debug and access the functions and variables of the modules with the symbol browser.

Alternatively to the local menu, you can load the symbols of a kernel module by running the “el_autoload.cmm” script with the module name and parameter “3”.
E.g. for a module named “mymodule” on ARM architecture:

```
DO ~/demo/arm/kernel/pikeos/elinos/el_autoload.cmm "mymodule" 3
```

If you want to debug the initialization routine of a kernel module, select the menu item “ELinOS” -> “Debug Module on init...” *before* you load the kernel module in Linux. Specify the module name in the upcoming dialog, then load the kernel module in Linux. The debugger then automatically halts at the module init routine.

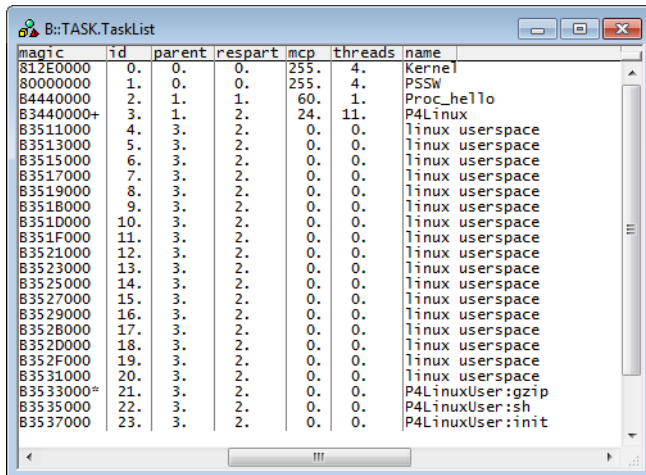
Alternatively to the menu, you can initiate the catching of the module’s init routine by running the “el_mod_debug.cmm” script with the module name as parameter.

E.g. for a module named “mymodule” on ARM architecture:

```
DO ~/demo/arm/kernel/pikeos/elinos/el_mod_debug.cmm mymodule
```

Linux Processes

Linux processes run as a PikeOS task. PikeOS reserves several tasks for Linux processes (“linux userspace”). As soon as a process runs, it gets one of these task slots. The **TASK.TaskList** window then shows the Linux processes as PikeOS task named “P4LinuxUser:<name>”.



magic	id	parent	respart	mcp	threads	name
812E0000	0.	0.	0.	255.	4.	Kernel
80000000	1.	0.	0.	255.	4.	PSSW
84440000	2.	1.	1.	60.	1.	Proc_hello
83440000+	3.	1.	2.	24.	11.	P4Linux
B3511000	4.	3.	2.	0.	0.	linux userspace
B3513000	5.	3.	2.	0.	0.	linux userspace
B3515000	6.	3.	2.	0.	0.	linux userspace
B3517000	7.	3.	2.	0.	0.	linux userspace
B3519000	8.	3.	2.	0.	0.	linux userspace
B351B000	9.	3.	2.	0.	0.	linux userspace
B351D000	10.	3.	2.	0.	0.	linux userspace
B351F000	11.	3.	2.	0.	0.	linux userspace
B3521000	12.	3.	2.	0.	0.	linux userspace
B3523000	13.	3.	2.	0.	0.	linux userspace
B3525000	14.	3.	2.	0.	0.	linux userspace
B3527000	15.	3.	2.	0.	0.	linux userspace
B3529000	16.	3.	2.	0.	0.	linux userspace
B352B000	17.	3.	2.	0.	0.	linux userspace
B352D000	18.	3.	2.	0.	0.	linux userspace
B352F000	19.	3.	2.	0.	0.	linux userspace
B3531000	20.	3.	2.	0.	0.	linux userspace
B3533000*	21.	3.	2.	0.	0.	P4LinuxUser:gzip
B3535000	22.	3.	2.	0.	0.	P4LinuxUser:sh
B3537000	23.	3.	2.	0.	0.	P4LinuxUser:init

Debugging a Linux process then works as simple as debugging a PikeOS task. Just load the symbols of the process (using the symbol autoloader) and start debugging. You can also use the local menus of **EXTension.ELProcess** or **EXTension.ELThread** (right click on the “magic”) to load the symbols of an ELinOS process.

If you want to debug the Linux process right from the beginning at “main()”, select the menu item “ELinOS” -> “Debug Process on enit...” *before* you start the process within Linux. Specify the process name in the upcoming dialog, then start the process in Linux. The debugger then automatically halts at the process at main().

Alternatively to the menu, you can initiate the catching of the process’ main routine by running the “el_app_debug.cmm” script with the process name as parameter.

E.g. for a process named “myprocess” on ARM architecture:

```
DO ~/demo/arm/kernel/pikeos/elinos/el_app_debug.cmm myprocess
```

Linux Libraries

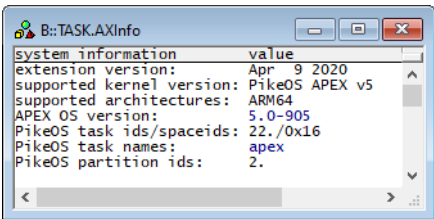
To debug user libraries, open an **EXTension.ELProcess** window and double-click on the “magic” of the process that contains the library to debug. Expand the “code files” tree. Right-click on the library to debug and select “Load library symbols” from the local menu to load the symbol file of the selected library. After this, you can debug and access the functions and variables of the library with the symbol browser.

EXTension.AXInfo

Display APEX information

Format: **EXTension.AXInfo**

Displays information about the APEX awareness and the used APEX personality.



The APEX awareness must be loaded and configured for this command. See [APEX](#).

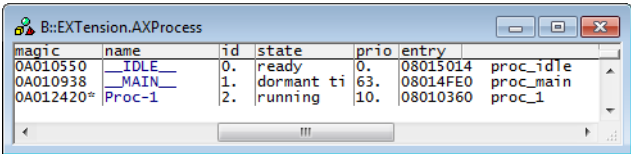
EXTension.AXProcess

Display APEX processes

Format: **EXTension.AXProcess** <process> <space_id>

Displays a table of all processes created in the APEX personality.

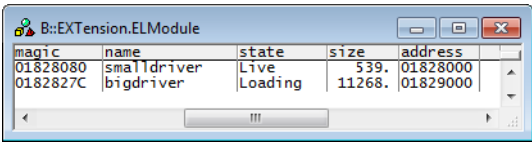
Without any arguments, a table with all created processes will be shown.
Specify a process magic number, ID or name to display detailed information on that process.
When multiple personalities are configured, the <space_id> may specify the space ID (= PikeOS task ID) of the requested process.



The APEX awareness must be loaded and configured for this command. See [APEX](#).

Format:EXTension.ELModule

Displays a table of all kernel modules created in the ELinOS personality.



The field “magic” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

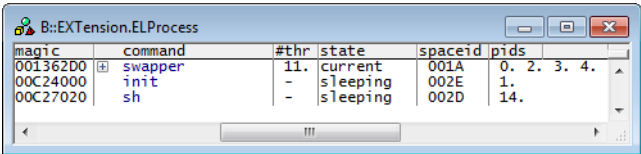
The ELinOS awareness must be loaded and configured for this command. See [ELinOS](#).

EXTension.ELProcess

Display ELinOS processes

Format:EXTension.ELProcess

Displays a table of all processes created in the ELinOS personality.



The field “magic” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

The ELinOS awareness must be loaded and configured for this command. See [ELinOS](#).

EXTension.ELThread

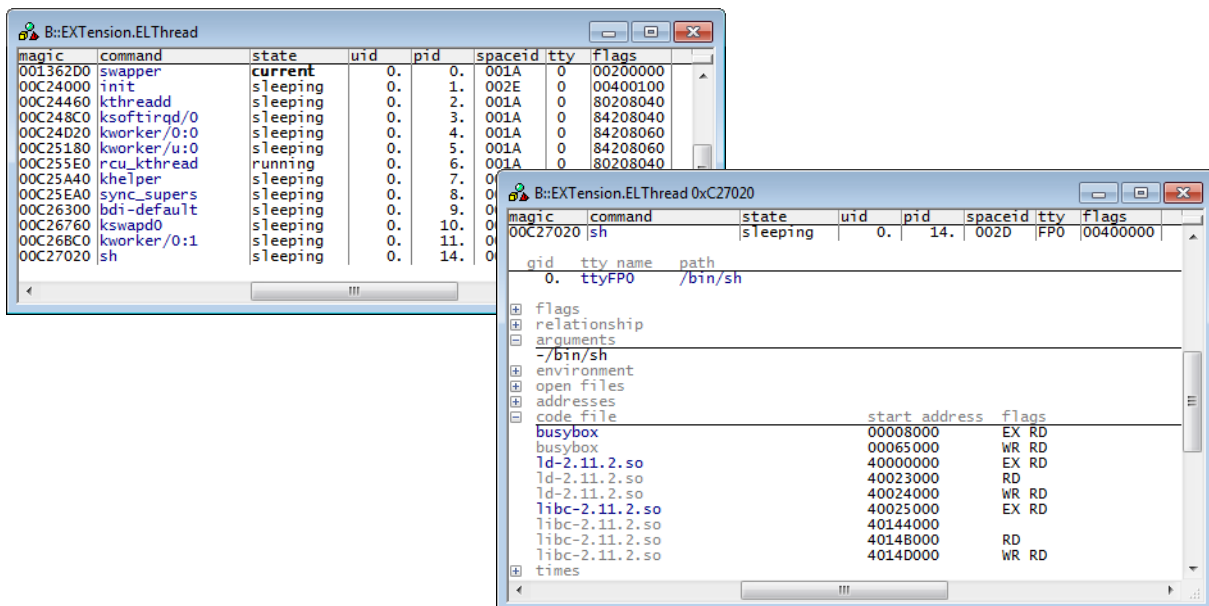
Display ELinOS threads

Format:EXTension.ELThread [<thread>]

Displays a table of all processes and threads created in the ELinOS personality, or detailed information about one specific process or thread.

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

Specify a thread magic number, ID or name to display detailed information on that thread.



The field “magic” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

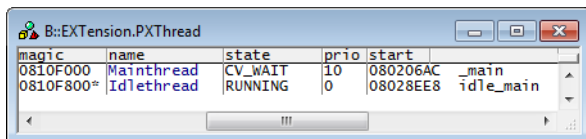
The ELinOS awareness must be loaded and configured for this command. See [ELinOS](#).

EXTension.PXThread

Display POSIX threads

Format: **EXTension.PXThread**

Displays a table of all threads created in the POSIX personality.

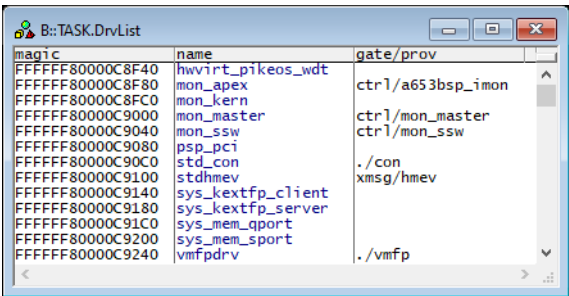


The POSIX awareness must be loaded and configured for this command. See [POSIX](#).

Format:

TASK.DrvList

Displays a table of kernel drivers loaded in PikeOS.

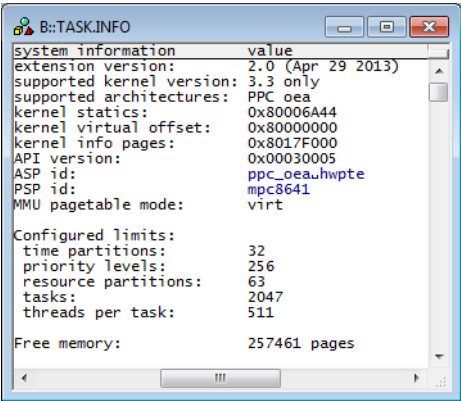


The screenshot shows a window titled "B::TASK.DrvList" with a table of kernel drivers. The table has three columns: "magic", "name", and "gate/prov". The data is as follows:

magic	name	gate/prov
FFFFFFFF80000C8F40	hwvirt_pikeos_wdt	
FFFFFFFF80000C8F80	mon_apex	ctrl/a653bsp_imon
FFFFFFFF80000C8FC0	mon_kern	
FFFFFFFF80000C9000	mon_master	ctrl/mon_master
FFFFFFFF80000C9040	mon_ssw	ctrl/mon_ssw
FFFFFFFF80000C9080	psp_pc1	
FFFFFFFF80000C90C0	std_con	./con
FFFFFFFF80000C9100	stdhmev	xmsg/hmev
FFFFFFFF80000C9140	sys_kextfp_client	
FFFFFFFF80000C9180	sys_kextfp_server	
FFFFFFFF80000C91C0	sys_mem_qport	
FFFFFFFF80000C9200	sys_mem_sport	
FFFFFFFF80000C9240	vmfpdrv	./vmfp

Format: **TASK.INFO**

Displays information about the awareness and the used PikeOS kernel.



TASK.Option

Set awareness options

Format: **TASK.Option** *<option>*

<option>: **THRCTX** [ON | OFF]

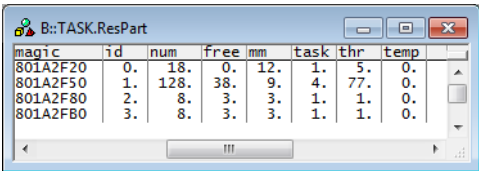
Sets various options to the awareness.

THRCTX

Set the context ID type that is recorded with the real-time trace (e.g. ETM). If set to on, the context ID in the trace contains thread ID. If set to off, the context ID only contains the ASID.
See [Task Runtime Statistics](#).

Format: **TASK.ResPart**

Displays a table of resource partitions defined in PikeOS.



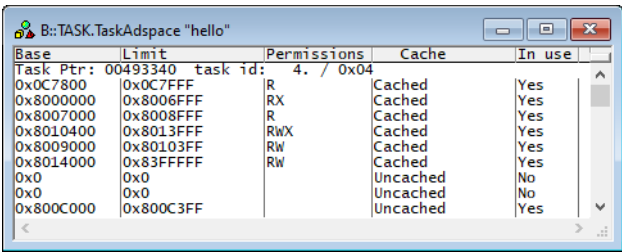
TASK.TaskAdspace

Display task address space

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

Only available on PikeOS for MPU.

Displays the address table of one task.



TASK.TaskFile.ADD

Map file name to task name

Format: **TASK.TaskFile.ADD** "<task_name>" "<file>"

Maps a file name to a task name.

In PikeOS, task names are often different from their (symbol) file names. For the symbol autoloader to automatically find the appropriate symbol file for a task, use this command to map task names to file names.

Format:	TASK.TaskFile.view
---------	--------------------

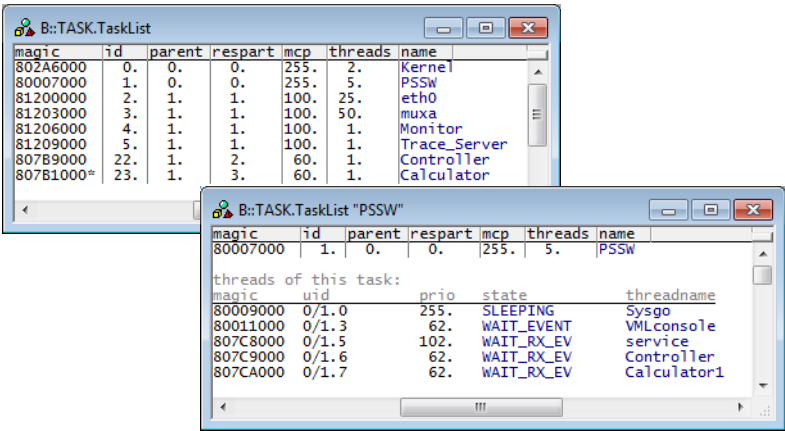
Show task name to file name mappings. See [TASK.TaskFile.ADD](#).

Format:

TASK.TaskList [<task>]

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

Without any arguments, a table with all created tasks will be shown.
Specify a task magic number, ID or name 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 task struct).

The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

Format: **TASK.ThrliSt** [*<thread>*]

Displays the thread table of PikeOS or detailed information about one specific thread.

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

Specify a thread magic number, ID or name to display detailed information on that thread.

[illegible]

“magic” is a unique ID used by the OS Awareness to identify a specific thread (address of the thread struct).

The field “magic” is mouse sensitive. Double-clicking on it opens an appropriate window. Right clicking on it will show a local menu.

There are special definitions for PikeOS specific PRACTICE functions.

TASK.CONFIG()

OS Awareness configuration information

Syntax:

TASK.CONFIG(magic | magicsize)

Parameter and Description:

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

Return Value Type: Hex value.

TASK.TASK.MAGIC()

magic number of task

Syntax:

TASK.TASK.MAGIC("<task_name>")

Returns the magic number of the given task.

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

Return Value Type: Hex value.

TASK.TASK.ID()

ID of task

Syntax:

TASK.TASK.ID(<task_magic>)

Returns the ID of the given task.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

Syntax: **TASK.TASK.NAME(<task_magic>)**

Returns the name of the given task.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [String](#).

TASK.TASK.ID2NAME()

Convert task ID to name

Syntax: **TASK.TASK.ID2NAME(<task_ID>)**

Returns the name of the given task ID.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [String](#).

TASK.TASKNAME2ID()

Convert task name to ID

Syntax: **TASK.TASKNAME2ID("<task_name>")**

Returns the ID of the given task.

Parameter Type: [String](#) (*with quotation marks*).

Return Value Type: [Hex value](#).

TASK.TASKFILE()

Symbol file name of task

Syntax: **TASK.TASKFILE("<task_name>")**

Returns the symbol file name of the given task, defined with [TASK.TaskFile.ADD](#).

Parameter Type: [String](#) (*with quotation marks*).

Return Value Type: [String](#).

Syntax: **EXT.AXPROCESS.THREAD(<process_magic>)**

Returns the magic of the PikeOS thread of the APEX process based on the specified process magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [Hex value](#)..

EXT.AXPROCESS.THREAD2()

PikeOS thread of APEX process

Syntax: **EXT.AXPROCESS.THREAD2(<process_magic>,<space_id>)**

Returns the magic of the PikeOS thread of the APEX process based on the specified process magic number and space ID of APEX personality.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [Hex value](#)..

EXT.ELINOS.SPACEID()

Space ID of ELinOS personality

Syntax: **EXT.ELINOS.SPACEID()**

Returns the space ID of the ELinOS personality.

Return Value Type: [Hex value](#).

EXT.ELPROCESS.NAME()

Name of ELinOS process

Syntax: **EXT.ELPROCESS.NAME(<process_magic>)**

Returns the name of the ELinOS process based on the specified process magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [String](#).

Syntax: **EXT.ELLIBRARY.ADDRESS(<library_magic>)**

Returns the load address of the ELinOS library based on the specified library magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [Hex value](#).

Syntax: **EXT.ELLIBRARY.SPACEID(<library_magic>)**

Returns the space ID for the specified ELinOS library magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [Hex value](#).

Syntax: **EXT.ELLIBRARY.NAME(<library_magic>)**

Returns the library name for the specified ELinOS library magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [String](#).

Syntax: **EXT.ELMODULE.MAGIC("<module_name>")**

Returns the module magic number for the specified ELinOS module name.

Parameter Type: [String](#) (*with* quotation marks).

Return Value Type: [Hex value](#).

Syntax:

EXT.ELMODULE.NAME(<module_magic>)

Returns the name of the ELinOS module based on the specified module magic number.

Parameter Type: [Decimal](#) or [hex](#) or [binary value](#).

Return Value Type: [String](#).

EXT.ELMODULE.SECADDR()

Section address of ELinOS module

Syntax:

EXT.ELMODULE.SECADDR(<module_magic>,<index>)

Returns the section address of the ELinOS module based on the specified module magic number and the indexed section number.

Parameter and Description:

<module_magic>	Parameter Type: Decimal or hex or binary value .
<index>	Parameter Type: Decimal or hex or binary value .

Return Value Type: [Hex value](#).

•