LAUTERBACH
DEVELOPMENT TOOLS

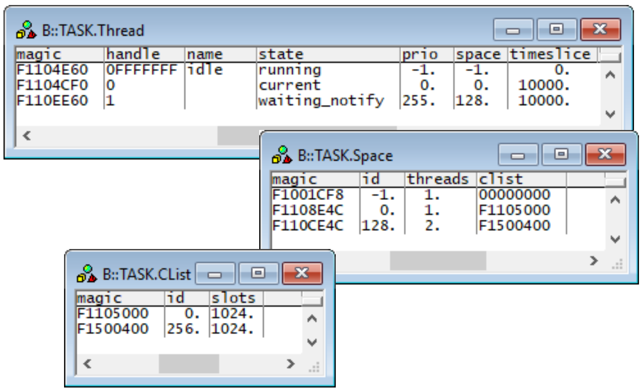
OS Awareness Manual OKL4

OS Awareness Manual OKL4

OS Awareness Manual OKL4

Version 06-Jun-2024

Overview



```
magic     handle    name  state           prio  space  timeslice
F1104E60  0FFFFFFF  idle  running          -1.   -1.        0.
F1104CF0  0               current           0.    0.    10000.
F110EE60  1               waiting_notify  255.  128.    10000.
```

```
magic     id    threads  clist
F1001CF8   -1.  1.       00000000
F1108E4C    0.  1.       F1105000
F110CE4C  128.  2.       F1500400
```

```
magic     id    slots
F1105000    0.  1024.
F1500400  256.  1024.
```

The OS Awareness for OKL4 contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

Terminology

OKL4 uses the term “threads”. If not otherwise specified, the TRACE32 term “task” corresponds to OKL4 threads.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Training Basic Debugging”** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **“T32Start”** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **“General Commands”** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **“OS Awareness Manuals”** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

Supported Versions

Currently OKL4 is supported for the following versions:

- OKL4 V3 on ARM

Configuration

The **TASK.CONFIG** command loads an extension definition file called “okl4.t32” (directory “~~/demo/<*processor*>/kernel/okl4”). It contains all necessary extensions.

Automatic configuration tries to locate the OKL4 internals automatically. For this purpose all symbol tables must be loaded and accessible at any time the OS Awareness is used.

If you want to display the OS objects “On The Fly” while the target is running, you need to have access to memory while the target is running. In case of ICD, you have to enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent).

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 okl4**

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

Quick Configuration Guide

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

1. Carefully read the PRACTICE demo start-up script (~~/demo/<*processor*>/kernel/okl4/okl4.cmm).
2. Make a copy of the PRACTICE script “okl4.cmm”. Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the kernel resources and use the trace functions (if available).

Now you can access the OKL4 extensions through the menu.

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

Hooks & Internals in OKL4

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.

Both, the cells and the kernel need to be built with debugging symbols switched on. You need to rebuild the kernel from the SDK tree. Configure the application to be linked with this specific kernel. Preserve the kernel’s and the cells binary files to be loaded into the debugger as symbol files.

Features

The OS Awareness for OKL4 supports the following features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following OKL4 components can be displayed:

TASK.Thread	Threads
TASK.Space	Address spaces
TASK.CList	Capability lists

For a description of the commands, refer to chapter "**OKL4 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-Related Breakpoints

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

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

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

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

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

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

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

Break.CONFIG.UseContextID ON	Enables the comparison to the whole Context ID register.
Break.CONFIG.MatchASID ON	Enables the comparison to the ASID part only.
TASK.List.tasks	If **TASK.List.tasks** provides a trace ID (**traceid** column), the debugger will use this ID for comparison. Without the trace ID, it uses the magic number (**magic** column) for comparison.

When single stepping, the debugger halts at the next instruction, regardless of which task hits this breakpoint. When debugging shared code, stepping over an OS function may cause a task switch and coming back to the same place - but with a different task. If you want to restrict debugging to the current task, you can set up the debugger with **SETUP.StepWithinTask ON** to use task-related breakpoints for single stepping. In this case, single stepping will always stay within the current task. Other tasks using the same code will not be halted on these breakpoints.

If you want to halt program execution as soon as a specific task is scheduled to run by the OS, you can use the **Break.SetTask** command.

Task Context Display

You can switch the whole viewing context to a task that is currently not being executed. This means that all register and stack-related information displayed, e.g. in **Register**, **Data.List**, **Frame** etc. windows, will refer to this task. Be aware that this is only for displaying information. When you continue debugging the application (**Step** or **Go**), the debugger will switch back to the current context.

To display a specific task context, use the command:

Frame.TASK [*<task>*]	Display task context.

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

To display the call stack of a specific task, use the following command:

Frame /Task *<task>*	Display call stack of a task.

If you'd like to see the application code where the task was preempted, then take these steps:

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

MMU Support

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

Scanning System and Processes

To scan the address translation of a specific address space, use the command **TASK.MMU.SCAN "<space>"**. This command scans the MMU translation of the specified address space.

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

Dynamic Task Performance Measurement

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

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

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

Task Runtime Statistics

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

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

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

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

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

Function Runtime Statistics

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

All function-related statistic and time chart evaluations can be used with task-specific information. The function timings will be calculated dependent on the task that called this function. To do this, in addition to the function entries and exits, the task switches must be recorded.

To do a selective recording on task-related function runtimes based on the data accesses, use the following command:

```
; Enable flow trace and accesses to the magic location
Break.Set TASK.CONFIG(magic) /TraceData
```

To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

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

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

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

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

OKL4 Specific Menu

The menu file "okl4.men" contains a menu with OKL4 specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **OKL4**.

- The **Display** menu items launch the kernel resource display windows.

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

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

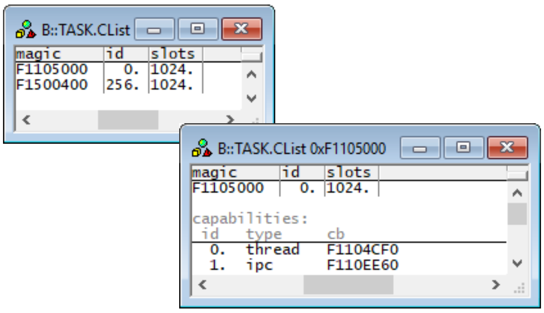
TASK.CList

Display capability lists

Format: **TASK.Clist** [<*clist*>]

Displays the capability list table of OKL4 or detailed information about one specific capability list.

Without any arguments, a table with all created capability lists will be shown.
Specify a capability list ID or magic number to display detailed information on that capability list.



```
magic     id    slots
F1105000    0.  1024.
F1500400  256.  1024.
```

```
magic     id    slots
F1105000    0.  1024.

capabilities:
 id   type      cb
  0.  thread    F1104CF0
  1.  ipc       F110EE60
```

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

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

TASK.MMU.SCAN

Scan MMU address space

Format: **TASK.MMU.SCAN** [<*space*>]

Scans the target MMU of this address space, and sets the Debugger MMU appropriately, to cover the physical to logical address translation of this specific address space.

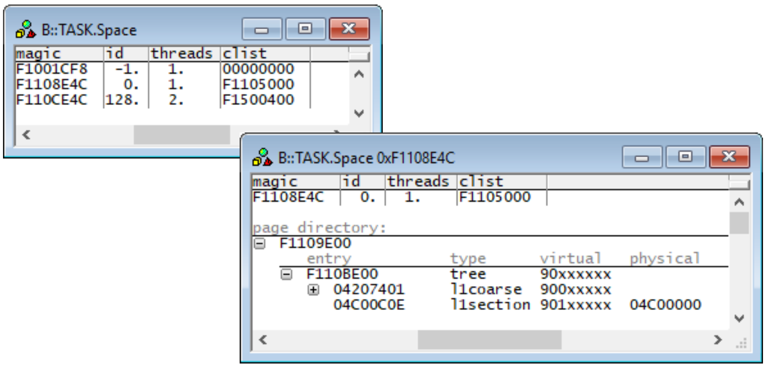
The command walks through all page tables which are defined for this address spaces and prepares the Debugger MMU to hold the physical to logical address translation. This is needed to provide full HLL support.

<*space*>	Specify a space magic or ID.

Format: **TASK.Space** [<*space*>]

Displays the address space table of OKL4 or detailed information about one specific address space.

Without any arguments, a table with all created address spaces will be shown.
Specify a space ID or magic number to display detailed information on that address space.



```
magic    |id  |threads|clist
F1001CF8 | -1.|  1.   |00000000
F1108E4C |  0.|  1.   |F1105000
F110CE4C |128.|  2.   |F1500400
```

```
magic    |id  |threads|clist
F1108E4C |  0.|  1.   |F1105000

page directory:
⊟ F1109E00
      entry           type      virtual   physical
  ⊟ F110BE00          tree      90xxxxxx
    ⊞ 04207401        l1coarse  900xxxxx
      04C00C0E        l1section 901xxxxx  04C00000
```

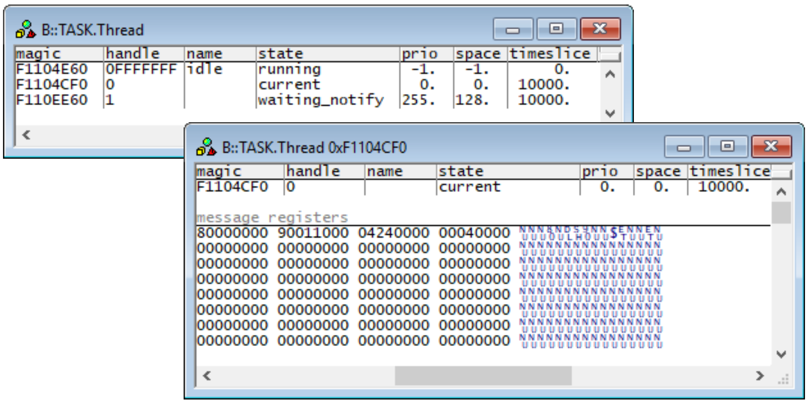
“magic” is a unique ID, used by the OS Awareness to identify a specific address space (address of the space structure).

The fields “magic”, “id” and “clist” are mouse sensitive. Double-clicking on them opens appropriate windows.

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

Displays the thread table of OKL4 or detailed information about one specific thread.

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



```
magic     handle    name  state           prio  space timeslice
F1104E60  0FFFFFFF  idle  running          -1.   -1.       0.
F1104CF0  0               current           0.    0.   10000.
F110EE60  1               waiting_notify  255.  128.   10000.
```

```
magic     handle  name  state        prio  space timeslice
F1104CF0  0             current        0.    0.   10000.

message registers
80000000 90011000 04240000 00040000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

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

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

OKL4 PRACTICE Functions

There are special definitions for OKL4 specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

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

Parameter and Description:

magic	**Parameter Type**: String (***without*** quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number).
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.