OS Awareness Manual Linux

OS Awareness Manual Linux

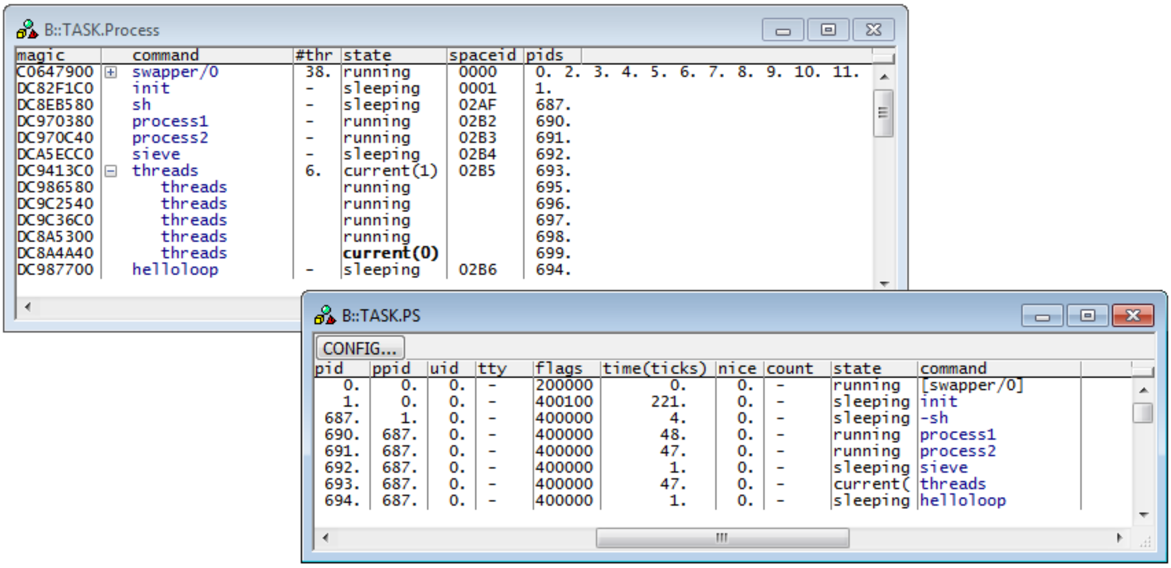
OS Awareness Manual Linux

Version 06-Jun-2024

History

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

Overview



```
B::TASK.Process
magic       command        #thr state       spaceid pids
C0647900 ⊞ swapper/0        38. running     0000    0. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
DC82F1C0    init            -   sleeping    0001    1.
DC8EB580    sh              -   sleeping    02AF    687.
DC970380    process1        -   running     02B2    690.
DC970C40    process2        -   running     02B3    691.
DCA5ECC0    sieve           -   sleeping    02B4    692.
DC9413C0 ⊟ threads          6.  current(1)  02B5    693.
DC986580       threads          running             695.
DC9C2540       threads          running             696.
DC9C36C0       threads          running             697.
DC8A5300       threads          running             698.
DC8A4A40       threads          current(0)          699.
DC987700    helloloop       -   sleeping    02B6    694.
```

```
B::TASK.PS
CONFIG...
pid   ppid  uid  tty flags  time(ticks) nice count state    command
  0.    0.   0.  -   200000          0.   0.  -    running  [swapper/0]
  1.    0.   0.  -   400100        221.   0.  -    sleeping init
687.    1.   0.  -   400000          4.   0.  -    sleeping -sh
690.  687.   0.  -   400000         48.   0.  -    running  process1
691.  687.   0.  -   400000         47.   0.  -    running  process2
692.  687.   0.  -   400000          1.   0.  -    sleeping sieve
693.  687.   0.  -   400000         47.   0.  -    current( threads
694.  687.   0.  -   400000          1.   0.  -    sleeping helloloop
```

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

This document describes Stop Mode debugging for Linux. If you are interested by Linux Run Mode debugging or Integrated Run & Stop Mode debugging, please refer to **“Run Mode Debugging Manual Linux”** (rtos_linux_run.pdf).

Terminology

Linux uses the terms “processes” and “tasks”. If not otherwise specified, the TRACE32 term “task” corresponds to Linux tasks, which may be executing processes or POSIX 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.

- **Linux Debugging Reference Card** (support.lauterbach.com/downloads/reference-cards)

Please also refer to the TRACE32 Linux debugging training manuals:

- **"Training Linux Debugging"** (training_rtos_linux.pdf)
- **"Training Linux Debugging for Intel® x86/x64"** (training_rtos_linux_x86.pdf)

Supported Versions

Currently Linux is supported for the following versions:

- Linux kernel versions 2.4, 2.6, 3.x, 4.x and 5.x on Andes, ARC, ARM, ARM64, Beyond, ColdFire, MIPS, MIPS64, PowerPC, PowerPC64, RISC-V SH4, XScale, x86 and x64

Configuration

The **TASK.CONFIG** command loads an extension definition file. For Linux-2.x, this file is called Linux2.t32 (directory “~~/demo/<*processor*>/kernel/linux/linux-2.x/”). For linux-3.x and newer, the extension definition file is called linux.t32 (directory “~~/demo/<*processor*>/kernel/linux/awareness/”). It contains all necessary extensions.

Automatic configuration tries to locate the Linux 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. This is not supported by all processor architectures. Please also to your **Processor Architecture Manual** for more information.

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 linux.t32**

Note that the default Linux kernel configuration generally does not include any debug information. Please change your configuration to generate kernel debug information.

See **Hooks & Internals** for details. See also the example “~~/demo/<*processor*>/kernel/linux/linux.cmm”.

Quick Configuration Guide

To access all features of the OS awareness you should follow the following road map:

1. Carefully read the PRACTICE demo start-up script (~~/demo/<*processor*>/kernel/linux/linux.cmm).
2. Make a copy of the PRACTICE script file “linux.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).

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

Hooks & Internals in Linux

No hooks are used in the kernel.

For retrieving the kernel data 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. This requires that the whole Linux kernel is compiled with debug symbols switched on, and that the symbols of the “vmlinux” file are loaded.

If you control the compile stage by hand, just switch on debug symbols by adding the option “-g” to gcc. In most kernel configuration scripts, you have an option **“Kernel Hacking” > “Compile kernel with debug info”** that enables debug symbols to the kernel.

Features

The OS awareness for Linux supports the following features.

Display of Kernel Resources

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

TASK.DTask	Tasks
TASK.Process	Processes and threads
TASK.PS	“ps” outputs
TASK.MODule	Kernel modules
TASK.FS	File system internals
TASK.DMESG	Kernel log buffer
TASK.DTB	Device tree blob
TASK.DTS	Device tree source
TASK.NET	Network devices
TASK.VMAINFO	Vmalloc info

For a detailed description of each command, refer to chapter “**Linux 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 and TRANSlation commands refer to this necessity.

Because of the “On Demand Paging” mechanism of the Linux kernel, when single stepping the code, the instruction pointer could jump to a not yet loaded page. The debugger will not be able to display the assembly code and could not single step the current instruction. See “**On Demand Paging**” for details and workaround.

Space IDs

Under Linux different processes may use identical virtual addresses. To distinguish between those addresses, the debugger uses an additional identifier, the so-called space ID (memory space ID) that specifies, which virtual memory space an address refers to. The command **SYStem.Option.MMUSPACES ON** enables the use of the space ID. The space ID is zero for all processes using the kernel address space (kernel tasks) and for the kernel code itself. For processes using their own address space, the space ID equals the lower 16bits of the process ID. Threads of a particular process use the memory space of the invoking parent process. Consequently threads have the same space ID as the parent process.

You may scan all the kernel’s data structures for space IDs, using the command **TRANSlation.ScanID**. Use **TRANSlation.ListID** to get a list of all recognized space IDs.

The function **task.proc.spaceid(“<process>”)** returns the space ID for a given process. If the space ID is not equal to zero, load the symbols of a process to this space ID:

```
LOCAL &spaceid
&spaceid=task.proc.spaceid("myProcess")
Data.LOAD myProcess &spaceid:0 /NoCODE /NoClear
```

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 Andes:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<format> Options for ARC:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<format> Options for ARM:

<format>	Description
LINUX	Standard format used by Linux
LINUXSWAP	Linux <= 2.6.37 with configured swap space (CONFIG_SWAP)
LINUXSWAP3	Linux >= 2.6.38 with configured swap space (CONFIG_SWAP)
STD	Standard format defined by the CPU
TINY	MMU format using a tiny page size of only 1024 bytes

<format> Options for BEYOND:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<format> Options for MicroBlaze:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<format> Options for MIPS:

<format>	Description
LINUX32	Linux 32-bit, page size 4kB
LINUX32P16	Linux 32-bit, page size 16kB
LINUX32P16R2	Linux 32-bit, page size 16kB, used on MIPS32 R2 or R6 (internally identical to format LINUX32P16R41) (CONFIG_CPU_MIPSR2 or CONFIG_CPU_MIPSR6 on Linux4.1 or later)
LINUX32P16R2	Deprecated: internally identical to format LINUX32P16R41
LINUX32R4K	Linux 32-bit, page size 4kB, like LINUX32 but different page flags
LINUX32RIXI	Linux 32-bit with RI/XI bits (CONFIG_USE_RI_XI_PAGE_BITS)
LINUX64	Linux 64-bit with 64-bit PTEs, page size 4kB. Separate page table for high address range can be specified with optional extra parameter *<base_address_highrange>* (For details, see **MMU.FORMAT** in debugger_mips.pdf).
LINUX64HTLB	Linux 64-bit with 64-bit PTEs, page size 4kB for huge TLB. Uses separate sub table for addresses > 0xFFFFFFFFC0000000.
LINUX64HTLBP16	Linux 64-bit like LINUX64HTLB but pag esize 16kB.
LINUX64P16	Linux 64-bit with 64-bit PTEs, page size 16kB. Depth 3 levels.
LINUX64P64	Linux 64-bit with 64-bit PTEs, page size 64kB. Depth 3 levels.
LINUX64P64LT	Linux 64-bit with 64-bit PTEs, page size 64kB. Depth 2 levels with large level 1 table (used for BROADCOM(R) XLP SDK 3.7.10 and alike)
LINUX64RIXI	Linux 64-bit with 64-bit PTEs with RI/XI bits, page size 4kB (CONFIG_USE_RI_XI_PAGE_BITS). Separate page table for high address range can be specified with optional extra parameter *<base_address_highrange>* (For details, see **MMU.FORMAT** in debugger_mips.pdf).
LINUXBIG	Linux 32-bit with 64-bit PTEs on MIPS32 (CONFIG_64BIT_PHYS_ADDR && CONFIG_CPU_MIPS32)
LINUXBIG64	Linux 32-bit with 64-bit PTEs on MIPS64 (CONFIG_64BIT_PHYS_ADDR && !CONFIG_CPU_MIPS32)

<format> Options for Motorola 68000:

<format>	Description
LINUX	Standard format used by Linux

<format> Options for NIOS:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<format> Options for PowerPC:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)
LINUX64_E6	Use LINUX64_E6 for e6500 core devices
LINUXE5	Linux with 64-bit PTEs, e500 core (CONFIG_PTE_64BIT && CONFIG_FSL_BOOKE). Covers 32-bit virtual address range.
LINUXEXT	Linux with 64-bit PTEs, no e500 core (CONFIG_PTE_64BIT && !CONFIG_FSL_BOOKE). Covers 32-bit virtual address range.
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 SH4:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)
LINUXEXTP64	Linux with extended TLBs (3 page table levels, 64-bit PTEs) (CONFIG_X2TLB && CONFIG_PAGE_SIZE_64KB)

<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)
LINUX64	PAE64 derivative with different level 1 translation table entries for addresses >0xFFFF800000000000
P32	32-bit format with 2 page table levels
PAE	Format with 3 page table levels (CONFIG_X86_PAE)
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

<format> Options for XTENSA:

<format>	Description
LINUX	Standard format used by Linux
LINUX26	Linux format with physical table pointers (some 2.6 variants)

<base_address>

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

<logical_kernel_address_range>

<logical_kernel_address_range> specifies the logical to physical address translation of the kernel address range. This spans continuously usually over the complete physical address range. Typically the virtual address range of the kernel starts at `0xC0000000` for a 32bit CPU.

<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 process, 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. Please note that the address range, where kernel modules are held, must be part of this area.

And don't forget to switch on the debugger's MMU translation with **TRANSlation.ON**.

Example: Having 32MB RAM at physical address 0x20000000, a typical MMU declaration looks like:

```
  MMU.FORMAT LINUX swapper_pg_dir 0xC0000000--0xC1FFFFFF 0x20000000
  TRANSlation.COMMON 0xC0000000--0xFFFFFFFF
  TRANSlation.ON
```

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

ARM/ARM64:

For ARM/ARM64 a detection mechanism is available in
~~/demo/arm/kernel/linux/board/generic-template/detect_translation.cmm
The script prints the mandatory parameters into the AREA window.

ColdFire:

The MMU format on ColdFire uses `kernel_pg_dir` as base address, and an additional parameter for user space memory maps. If `CONFIG_NEED_MULTIPLE_NODES` is *not* set, specify `mem_map`, otherwise specify zero. E.g., when the kernel translation ranges from `0xC0000000--0xC3FFFFFF` to physical address zero:

```
; CONFIG_NEED_MULTIPLE_NODES is not set
MMU.FORMAT LINUX kernel_pg_dir mem_map 0xc0000000--0xc3ffffff 0x0
; CONFIG_NEED_MULTIPLE_NODES=y
MMU.FORMAT LINUX kernel_pg_dir 0 0xc0000000--0xc3ffffff 0x0
```

x64

The MMU format on x64 uses `init_level4_pgt` or `init_top_pgt` as base address. Example:

```
  IF sYmbol.EXIST(init_level4_pgt)
     &base_address="init_level4_pgt"
  ELSE
     &base_address="init_top_pgt"

  MMU.FORMAT STD &base_address 0xffffffff80000000--0xffffffff9fffffff 0x0
```

Debugger Table Walk

To access the different process spaces correctly, the debugger needs to know the address translation of every virtual address it uses. If the debugger table walk is enabled (**TRANSlation.TableWalk ON**), the debugger walks through the MMU page tables each time it accesses a virtual address.

The debugger can also hold a local translation list. Translations can be added to this list either manually using the command **TRANSlation.Create** or by scanning the MMU page tables using **MMU.SCAN**. Scanning the MMU page tables is however not recommended since the scanned translation can get outdated after resuming the program execution.

Please note that the debugger local translation list (**TRANSlation.List**) has always the highest priority in the debugger translation process: the debugger tries first to look up the address translation in it’s own table (**TRANSlation.List**). If this fails, it walks through the target MMU tables to find the translation for a specific address.

Symbol Autoloader

The OS awareness for Linux contains a so-called autoloader, which automatically loads symbol files corresponding to executed processes, modules or libraries. The autoloader maintains a list of address ranges, corresponding to Linux 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**] Update autoloader table

The loaded components can change over time, when processes are started and stopped and kernel modules or libraries 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.

The command **TASK.sYmbol.Option AutoLoad** configures which types of components the autoloader shall consider:

- Processes,
- Kernel modules,
- All libraries
- Libraries of the current process, or
- Libraries of a specific process

It is recommended to restrict the components to the minimal set of interest (rather than all components), because it makes the autoloader checks much faster. By default, only processes are checked by the autoloader.

The command **sYmbol.AutoLOAD.CHECKLINUX** is used in the context of Linux to define which action is to be taken, for loading the symbols corresponding to a specific address.

sYmbol.AutoLOAD.CHECKLINUX *<action>*	Configure autoloader for Linux debugging

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

The action defined is invoked with Linux specific parameters (see below).

NOTE:	The action parameter needs to be written with quotation marks (for the parser it is a string) and (currently) requires a `space` before the closing quotation mark.

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, kernel module), the load address and space ID. A typical call is shown below. Please see the default script `~~/demo/<arch>/kernel/linux/awareness/autoload.cmm` for details.

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.

The autoloader is automatically set when the awareness is loaded with the **TASK.CONFIG** command (this is only valid for awareness newer than November 2012). It is thus not needed to include the command **sYmbol.AutoLOAD.CHECKLINUX** in your PRACTICE script.

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 state line 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.

In SMP systems, the **TASK.DTask** command contains an additional column that shows at which core the task is running, or was running the last time.

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)”.

Process / thread switch support for ARM using context ID register:

Most Arm Cortex-A processors do not have a data trace support. For such processors, the debugger uses the Context ID trace messages for task aware trace. The CONTEXTIDR register have to be written by the kernel on every task switch. This is enabled in the Linux kernel for Arm 32 and 64 bit processors with `CONFIG_PID_IN_CONTEXTIDR`.

Task State Analysis

NOTE: This feature is *only* available, if your debugger equipment is able to trace memory data accesses (flow trace is not sufficient). The scripts mentioned herein are based on state analyzers.

The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically. This feature is implemented by recording all accesses to the status words of all tasks. Additionally the accesses to the current task pointer (=magic) are traced.

“running” means the currently running task. “ready” defines runnable (R) tasks. The task states sleeping (S), disk sleep (D) and paging (W) are counted as “suspended”. “waiting” defines zombie (Z) and stopped (T) processes.

To do a selective recording on task states, the following PRACTICE commands can be used:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Mark all task state words with Alpha breakpoints
TASK.TASKState

; Program the Analyzer to record task state transitions
Analyzer.ReProgram
(
     Sample.Enable if AlphaBreak&&Write
)
```

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state time chart

All kernel activities up to the task switch are added to the calling task. The start of the recording time, when the calculation doesn't know, which task is running, is calculated as "(root)".

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)”.

Linux Specific Menu

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

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

- The **Display** menu items launch the kernel resource display windows. See chapter “**Display of Kernel Resources**”.
- **Process Debugging** refers to actions related to process based debugging.
  See also chapter “**Debugging the Process**”.
  - Use **Load Symbols…** and **Delete Symbols…** to load rsp. delete the symbols of a specific process. You may select a symbol file on the host with the “Browse” button. See also **TASK.sYmbol**.
  - **Debug New Process…** allows you to start debugging a process on a selected function (per default the main() function). Select this prior to starting the process. Specify the name of the process you want to debug and eventually the function that you want to stop at. Then start the process in your Linux terminal. The debugger will load the symbols and halt at the selected function. See also the demo script “app_debug.cmm”.
  - **Watch Processes** opens a process watch window or adds or removes processes from the process watch window. Specify a process name. See **TASK.Watch** for details.
  - **Display Process MMU Tables** executes the command **MMU.List TaskPageTable** for the selected process.
  - **Display Kernel MMU Tables** executes the command **MMU.List KernelPageTable**.
  - See also chapter “**Scanning System and Processes**”.
- **Module Debugging** refers to actions related to kernel module based debugging.
  See also chapter “**Kernel Modules**”.
  - Use **Load Symbols…** and **Delete Symbols…** to load rsp. delete the symbols of a specific kernel module. You may select a symbol file on the host with the “Browse” button. See also **TASK.sYmbol**.
  - **Debug Module on init…** allows you to start debugging a kernel module on it’s init function. Select this prior to inserting the module. Specify the name of the module you want to debug. Then insert the module in your Linux terminal. The debugger will load the symbols and halt at the init function (if available). See also the demo script “mod_debug.cmm”.
  - **Display Kernel MMU Tables** executes the command **MMU.List KernelPageTable**.
- **Library Debugging** refers to actions related to library based debugging.
  See also chapter “**Debugging into Shared Libraries**”.
  - Use **Load Symbols…** and **Delete Symbols…** to load rsp. delete the symbols of a specific library. Please specify the library name and the process name that uses this library. You may select a symbol file on the host with the “Browse” button. See also **TASK.sYmbol**.
  - **Display Process MMU Tables** executes the command **MMU.List TaskPageTable** for the selected process.

  - **Display Kernel MMU Tables** executes the command **MMU.List KernelPageTable**.

- Use the **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.

  - **Components** open a dialog to select the components that should be checked by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.

  - **Config** opens the autoloader configuration window.

  - **Set Target Root Path** sets the target the target root path on the host using the command **TASK.sYmbol.Option ROOTPATH**.

- **Options** open the dialog for setting different Linux awareness options. Please refer to the documentation of the commands **TASK.Option** and **TASK.sYmbol.Option** for more information.

- **Linux Terminal** opens a terminal window that can be configured prior to opening with **Configure Terminal**.

- **Generate RAM Dump** starts a RAM dump generation tool

- **Integrity Check** executes the command **TASK.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 task switches (if any) or task switches together with the default display.

- The **Perf** menu contains additional sub-menus 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 Linux Kernel and User Processes

This chapter describes the needed settings for debugging the Linux kernel, kernel modules as well as user space processes including threads and shared libraries.

You can find more information about debugging the Linux components in the Linux Debugging training manuals. Please refer to the following documents for more information:

- **“Training Linux Debugging”** (training_rtos_linux.pdf)

- **“Training Linux Debugging for Intel® x86/x64”** (training_rtos_linux_x86.pdf)

Linux Kernel

The Linux make process can generate different outputs (e.g. zipped, non-zipped, with or without debug info). For downloading the Linux kernel, you may choose whatever format you prefer. However, the Linux awareness needs several kernel symbols, i.e. you have to compile your kernel with debug information and preserve the resulting kernel file “vmlinux”. This file is in ELF format, and all other kernel images are derived from this file.

Downloading the Kernel

If you want to download the kernel image using the debugger, you have to specify, to which address to download it. The Linux kernel image is usually located at the physical start address of the RAM (sometimes the vector table is skipped, check label _stext in the system map).

When downloading a binary image, specify the start address, where to load. E.g., if the physical address starts at 0xA0000000:

```
  Data.LOAD.Binary vmlinux.bin 0xA0000000 /NosYmbol
```

When downloading the ELF image, you have to relocate the virtual addresses of the file to the physical addresses of the RAM. E.g., if the kernel starts virtually at 0xC0000000 (default), and the RAM starts physically at 0xA0000000:

```
Data.LOAD.Elf vmlinux 0xA0000000-0xC0000000 /NosYmbol
```

When downloading the kernel via the debugger, remember to set startup options that the kernel may require, before booting the kernel.

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 vmlinux file are virtual addresses. If you want to debug this (tiny) startup sequence, you have to load and relocate the symbols as mentioned above.

- Downloading the kernel via debugger:

  Just omit the /NosYmbol option, when downloading the kernel:

  (assuming physical address 0xA0000000 and virtual address 0xC0000000)

  ```
  Data.LOAD.Elf vmlinux 0xA0000000-0xC0000000
  ```

  After downloading, set your PC to the physical start address, and you're ready to debug.

- Downloading the kernel via Ethernet:

  Just load the symbols into the debugger *before* it is downloaded by the boot monitor:

  (assuming physical address 0xA0000000 and virtual address 0xC0000000)

  ```
  Data.LOAD.Elf vmlinux 0xA0000000-0xC0000000 /NoCODE
  ```

  Then, set an on-chip(!) breakpoint to the physical start address of the kernel (software breakpoints won't work, as they would be overwritten by the kernel download):

  ```
  Break.Set 0xA0000000 /Onchip
  ```

  Now let the boot monitor download and start the Linux image. It will halt on the start address, ready to debug. Delete the breakpoint when hit.

As soon as the processor MMU is switched on, you have to reload the symbol table to it's 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 Linux awareness, you have to load the virtual addressed symbols of the kernel into the debugger. The vmlinux ELF image contains all addresses in virtual format, so it's enough to simply load the file:

```
Data.LOAD.Elf vmlinux /NoCODE
```

You have to inform the debugger about the kernel address translations. See chapter **"MMU Declaration"** how to set up the MMU format to the debugger.

The kernel address translation covers the kernel code and data (sections of vmlinux) and is mapped in continuous pages to the physical RAM space.

User Processes

Each user process in Linux gets its own virtual memory space, each usually starting at address zero. To distinguish the different memory spaces, the debugger assigns a “space ID”, which is equal to the process ID. Using this space ID, it is possible to address a unique memory location, even if several processes use the same virtual address.

Linux uses the “on demand paging” mechanism to load the code and data of processes and shared libraries. Debugging those pages is not trivial, see “**On Demand Paging**” for details and workaround.

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

Debugging the Process

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

Please watch out for demand paging (see chapter “**On Demand Paging**”).

Manually Load Process Symbols:

For example, if you’ve got a a process called “hello” with the process ID **`12.`** (the dot specifies a decimal number!):

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

The space ID of a process may also be calculated by using the PRACTICE function `task.proc.spaceid()` (see chapter “**Linux PRACTICE Functions**”).

Automatically Load Process Symbols:

If a process name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command

```
  TASK.sYmbol.LOAD "hello"                 ; load symbols and scan MMU
```

This command loads the symbols of “hello” and scans the MMU of the process “hello”. See **TASK.sYmbol.LOAD** for more information.

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"
```

Debugging a Process From Scratch, Using a Script:

The script app_debug.cmm available in the path of the Linux awareness can be used to debug a process from the start.

The **Linux** menu a menu item which is based on this script: **Linux** -> **Process Debugging** -> **Debug Process on main**. See also chapter “**Linux Specific Menu**”.

When finished debugging with a process, or if restarting the process, you have to delete the symbols and restart the application debugging. Delete the symbols with:

```
sYmbol.Delete \\hello
```

If the autoloader is configured:

```
sYmbol.AutoLoad.CLEAR "hello"
```

Debugging a Process From Scratch, with Automatic Detection:

The **TASK.Watch** command group implements the above script as an automatic handler and keeps track of a process launch and the availability of the process symbols. See **TASK.Watch.View** for details.

Debugging Processes with the same name:

Please note that the provided awareness scripts don’t support debugging multiple processes which have the same name. When trying e.g. to debug a new process from main from the Linux menu or using the script app_debug.cmm and there is already a process with the same name running, you will get an error message saying that the process is already running. Moreover, when loading process symbols using the autoloader, the loader script autoload.cmm delete all symbols loaded on the same name. Thus, we recommend to use different names for processes.

Debugging into Shared Libraries

If the process uses shared libraries, Linux loads them into the address space of the process. The process itself contains no symbols of the libraries. If you want to debug those libraries, you have to load the corresponding symbols into the debugger.

Dynamically loaded libraries will be first linked, when they're called the first time. I.e. when stepping into a library function the first time, you may step into the dynamic loader instead of your library. To prevent this, set the environment variable "LD_BIND_NOW=1" on your target. This instructs the loader, to link all functions at load time instead of at run time.

Please watch out for demand paging (see chapter "**On Demand Paging**").

Manually Load Library Symbols:

1. Start your process and open a **TASK.DTask** window.
2. Double-click the magic value of the process that uses the library.
3. Expand the "code files" tree (if available).

   A list will appear that shows the loaded libraries and the corresponding load addresses.
4. Load the symbols to this address and into the space ID of the process.

   E.g. if the process has the space ID 12., the library is called "lib.so" and it is loaded on address 0xff8000, then use the command:

   ```
   Data.LOAD.Elf lib.so 12.:0xff8000 /NoCODE /NoCLEAR
   ```

   Of course, this library must be compiled with debugging information.

Automatically Load Library Symbols:

If a library name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command

```
TASK.sYmbol.LOADLib "hello" "libc.so"     ; load symbols and scan MMU
```

This command loads the symbols of the library "libc.so", used by the process "hello", and scans the MMU of the process "hello". See **TASK.sYmbol.LOADLib** for more information.

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 library. You can also force the loading of the symbols of a library with

```
sYmbol.AutoLoad.CHECK
sYmbol.AutoLoad.TOUCH "libc.so"
```

Debugging Linux Threads

Linux Threads are implemented as tasks that share the same virtual memory. The OS awareness for Linux assigns one space ID for all threads that belong to a specific process. It is sufficient, to load the debug information of this process only once (onto its space ID) to debug all threads of this process. See chapter "**Debugging the Process**" for loading the process' symbols.

There are several different mechanisms how threads are managed inside the Linux kernel. The Linux awareness tries to detect them automatically, but this may fail on some systems. If the **TASK.DTask** window doesn't show all threads correctly, declare the threading method manually with the **TASK.Option Threading** command.

The **TASK.DTask** window shows which thread is currently running ("current").

On Demand Paging

When a process is started, Linux doesn't load any code or data of this process. Instead, it uses the "on demand paging" mechanism. This means, Linux loads memory pages first, when they are accessed. As long as they aren't accessed by the CPU, they're not present in the system.

A "memory page" is a continuous memory region (e.g. 4 KByte), with a dedicated virtual and physical address range. The MMU handles the whole (user space) memory in such pages.

When starting a process, Linux just sets up it's task structures and loads the characteristics of the process' code and data sections from the process' file (size and addresses of the sections), but not the sections themselves. Then the kernel jumps to the "main" routine of the process. The first instruction fetch will then cause a code page fault, because the code is not yet present. The page fault handler then loads the actual code page (4 KByte) that contains the code of the "main" entry point, from the file. Note that only one page is loaded. If the program jumps to a location outside this page, or steps over a page boundary, another code page fault happens. While running, more and more pages will be loaded. (While on desktop systems it is common that pages are also discarded, on embedded systems this is usually not the case.) If a process terminates, all pages of this process are removed.

The same page loading mechanism applies to data and stack addresses. Variables are first visible to the system, after the CPU accessed them (by reading or writing the address and thus urging a page load). The stack grows page wise, as it is used.

When debugging those paged processes, you have to take care about this paging.

- The process' code and data is first visible to the debugger, after the pages were loaded.
- You cannot set a software breakpoint onto a function that is located in a page which is not yet loaded. The code for this function simply not yet exists, and thus cannot be patched with the breakpoint instruction. In such cases, use on-chip breakpoints instead.
- The CPU handles on-chip breakpoints *before* code page faults. If the CPU jumps onto an on-chip breakpoint, and the appropriate page is not yet loaded, the debugger will halt before the page is loaded. You'll see the program counter on a location with no actual code (usually the debugger shows "???" then). The same may happen, if you single step over a page boundary. In such cases, set an on-chip(!) breakpoint onto the next instruction and let the system "Go". The page fault handler will then load the page, the processor will execute the first instruction and halt on the next breakpoint. A simple workaround for functions is to set the breakpoint at the function entry

plus 4 (e.g. "main+4") for 32 bit fixed opcode length. Then the application will halt *after* the page was loaded. An another solution is to load the code to the debugger virtual memory and enable the option **TRANSlation.SHADOW ON**. If the code is not yet available, the debugger reads it from the loaded image in the debugger virtual memory. In this case, the code will be displayed in grey. The debugger could then calculate the next instruction when executing a step over for example and will automatically use an on-chip breakpoint. Moreover, if the option **TASK.sYmbol.Option AutoLOAD VM** is set, the autoloader will load the code for process to the debugger virtual memory an will enable **TRANSlation.SHADOW**. However this will only work for processes but not for libraries.

For already loaded processes, the demo directory contains a script called “app_page_load.cmm” that forces loading all code and data pages of the process by patching the application’s code. Please see the comments inside this script for details.

The on demand paging is a basic design feature of Linux that cannot be switched off.

However, there are two (different) ways to force Linux to load the appropriate pages in advance, before they’re actually used. This eases debugging in those pages. The solutions need modifications (patches) to the kernel. You may use one of both, or both patched together, depending on your needs.

Please note that beginning from Linux version 4.3.x, it is necessary to disable the kernel configuration `CONFIG_CPU_SW_DOMAIN_PAN` otherwise the proposed patches will not work correctly.

1. Forcing the load of pages used by processes (without libraries)

The following kernel modification accesses (reads) each code and data page, right before the process’ main entry point is started. This forces a page load to all pages in advance. As soon as the process entry point is called, all pages are present in the system. You may then set software breakpoints anywhere in the process’ code, and view the data area at any time.

A note to the stack: The stack space of a Linux process is not limited. If the stack grows, while the application is running, the stack pages will be allocated. Thus, there's no chance to calculate the stack space in advance. The following patch loads only the first stack page (4 kB).

Libraries used by the process are still subject to the demand paging.

Please note that this patch changes the behavior of the system.

This loading in advance will only work, if there is enough (physical) memory free to load the whole process code and data pages. If this is not the case, pages will be discarded again!

There is a variable called "`t32_force_process_page_load`" initialized to zero. Only if this variable is non-zero, the loading of the pages is performed. To enable the page loading, set

```
Data.Set t32_force_process_page_load 1
```

The function "`load_elf_binary()`" in fs/binfmt_elf.c calls "`start_thread()`". `start_thread()` is either defined as function in arch/mips/kernel/process.c, or as a macro in include/asm-<*arch*>/processor.h.

At the beginning of start_thread, add a call to load all pages; e.g.:

```
void start_thread() {
     /**** TRACE32 patch ****/

      void t32_load_all_process_pages (void);

                                    /* prototype */

      t32_load_all_process_pages();

                                    /* load all pages before start */

      …
```

then, somewhere in process.c, add the following code snippet:

```
/**** TRACE32 patch to force loading of all pages ***/
/**** of the new thread in advance                ***/

volatile char t32_force_process_page_load = 0;

                                 /* to be set by TRACE32 */

void t32_process_page_load_done (void)
{

                                 /* dummy function to inform TRACE32 */

}

void t32_load_all_process_pages (void)
{
     unsigned long  page, end;
     volatile char dummy;

     if (!t32_force_process_page_load)
          return;

     if (!current->mm)
          return;

      /* load code pages */
      page = current->mm->start_code & 0xfffff000;
      end  = (current->mm->end_code-1) & 0xfffff000;
      if (page)
           for (; page <= end; page += 0x1000)
                dummy = *((char*) page);

                                 /* force page load */
```

```
         /* load data pages */
         page = current->mm->start_data & 0xfffff000;
         end  = (current->mm->brk-1) & 0xfffff000;
         if (page)
              for (; page <= end; page += 0x1000)
                   dummy = *((char*) page);

                                    /* force page load */

         /* load stack page */
         page = current->mm->start_stack & 0xfffff000;
         dummy = *((char*) page);

                                    /* force page load */

         t32_process_page_load_done();
    }
```

Now, if "`t32_force_process_page_load`" is set to one, all pages of the process are loaded. When debugging a process from scratch, set a breakpoint to "`t32_process_page_load_done()`". The MMU Scan at that stage then scans all code and data pages that the process might use. After that, you should be able to debug your process without the demand paging troubles.

2. Forcing the load of all ELF pages (of processes and static libraries)

The following kernel modification accesses (reads) each page, right after Linux built a virtual memory map for it (actually, after reading the ELF memory map of the file). This forces a page load to all pages of processes and libraries that are read with the Linux kernel ELF loader. As soon as the process entry point is called, all pages are present in the system. You may then set software breakpoints anywhere in the process' or libraries's code, and view the data area at any time. (Please note that this does not cover dynamic libraries that are loaded with ld.so.)

The pages of uninitialized data (.bss segment) and the stack are not subject to the ELF loader, thus these pages are NOT forced to load, using this patch.

Please note that this patch changes the behavior of the system.

This loading in advance will only work, if there is enough (physical) memory free to load the whole process' and libraries code and data pages. If this is not the case, pages will be discarded again!

There is a variable called "`t32_force_elf_page_load`" initialized to zero. Only if this variable is non-zero, the loading of the pages is performed. To enable the page loading, set

```
Data.Set t32_force_elf_page_load 1
```

Modify the function “`elf_map()`” in fs/binfmt_elf.c. Add right before return (but after up_write()!):

```
…
up_write(&current->mm->mmap_sem);
{
     /**** TRACE32 patch ****/
     void t32_load_elf_page (unsigned long addr, unsigned long len);
     t32_load_elf_page(map_addr,
               eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr));
}
return(map_addr);
…
```

then, somewhere in binfmt_elf.c, add the following code snippet:

```
/**** TRACE32 patch to force load of mapped page in advance ****/

volatile char t32_force_elf_page_load = 0;

                                                  /* to be set by TRACE32
                                                  */

void t32_load_elf_page (unsigned long addr, unsigned long len)
{
     volatile char dummy;
     unsigned long page, end;

     if (!t32_force_elf_page_load)
          return;

     if (!addr || !len)
          return;

     /* load all mapped pages */

     page = addr & 0xfffff000;
     end  = (addr + len-1) & 0xfffff000;

     for (; page <= end; page += 0x1000)
          dummy = *((char*) page);

                                                  /* force page load */

}
```

Now, if "`t32_force_elf_page_load`" is set to one, all pages of all processes and static libraries are loaded in advance. After that, you should be able to debug your process without the demand paging troubles.

Kernel Modules

Kernel modules are dynamically loaded and linked by the kernel into the kernel space. If you want to debug kernel modules, you have to load the symbols of the kernel module into the debugger, and to relocate the code and data address information.

All information about a module is stored in the module's header that is created dynamically while the module loads. In order to get access to all module headers after a new module was loaded, the debugger needs to get informed about the newly created module's address translation.

The handling of kernel modules in Linux is completely different in Kernel versions 2.4 and 2.6.

Kernel Modules in Linux Kernel Version 2.4

The kernel does not hold any information about the module's loaded sections. To successfully load the symbols of a kernel modules, you may 1) manually load and relocate the sections or 2) let the debugger guess the section addresses or 3) patch the modutils to provide the necessary information.

1. **Manually Load Symbols (2.4)**

   If you're using insmod of the standard modutils (not busybox), then load your kernel module and generate relocation information with the option "-m":

   ```
   /sbin/insmod -m mymodul
   ```

   The output of this command will tell you where Linux has relocated the sections of the module.

   Now, load the symbols of the modules into the debugger, using the sections and addresses given by insmod. Give the option "`/reloc`" to the load command for each section:

   ```
   Data.LOAD.Elf mymodul.o /gnu /nocode /noclear /reloc .text at 0x…
        /reloc .data at 0x…  /reloc .bss at 0x… …
   ```

2. **Debugger’s Address Guessing (2.4)**

While it is easy to compute the address of the .text section of a kernel module, calculating the others is very hard and not safe. The debugger makes some wild guesses to get the address of the .data section. It may give correct results, but it may also be wrong.

Open the **TASK.MODule** window. It will give you a correct code address, and a guessed data address. Use these addresses to load the module’s symbols.In PRACTICE script files, you may use the functions **TASK.MOD.CODEADDR()** and **TASK.MOD.DATAADDR()** to retrieve the load addresses:

```
&code=task.mod.codeaddr("mymodule");
&data=task.mod.dataaddr("mymodule");
Data.LOAD.Elf mymodul.o /gnu /nocode /noclear \
     /reloc .text at &code /reloc .rodata after .text \
     /reloc .data at &data /reloc .bss after .data
```

TASK.sYmbol.LOADMod will use the same addresses to automatically load the symbols.

3. **Patching modutils (2.4)**

The Linux awareness contains a special detection of section’s addresses that needs a patch to insmod. When using this patch, all section information will be available in the module header, and all sections can be loaded correctly. **Appendix A** shows the necessary patch.

Now use the function task.mod.section() to get the addresses of all sections and relocate the symbols accordingly. Example PRACTICE script file for a module called “mymod”:

```
local &modulename &modulemagic &text &rodata &data &bss

&modulename="mymod"      ; without ".o" !

&modulemagic=task.mod.magic("&modulename")

&text=task.mod.section(".text",&modulemagic)
&rodata=task.mod.section(".rodata",&modulemagic)
&data=task.mod.section(".data",&modulemagic)
&bss=task.mod.section(".bss",&modulemagic)

Data.LOAD.Elf &modulename.o /nocode /noclear /gnu \
/reloc .text at &text /reloc .rodata at &rodata \
/reloc .data at &data /reloc .bss at &bss
```

If the section information is available, **TASK.sYmbol.LOADMod** will use the standard sections to automatically load the symbols.

Kernel Modules in Linux Kernel Version 2.6 and Newer

In Linux kernel version 2.6 and newer, the kernel contains all section information, if the kernel is configured with `CONFIG_KALLSYMS=y` and `CONFIG_SYSFS=y`. When configuring the kernel, set the option “General Setup”->”Configure standard kernel features”->”Load all symbols” to yes. You may say no to all sub-options. Also, set the option “File systems”->”Pseudo filesystems”->”sysfs file system support” to yes.

Without setting KALLSYMS or SYSFS, no section information is available, and debugging kernel modules is not possible.

If the section information is available, the debugger can read out this information to get the addresses of all sections and relocate the symbols accordingly. The option /reloctype of the **Data.LOAD.Elf**Data.LOAD.Elf command instructs the debugger to use the relocation information on the target. Internally to the Linux awareness, kernel modules have the type 3, so specify this as reloctype. Please note that access to the kernel variables must be possible whenever executing this command.

Example command for a module called “mymod”:

```
Data.LOAD.Elf mymod.ko /NoCODE /NoClear /reloctype 3
```

TASK.sYmbol.LOADMod will try to automatically load the symbols.

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 kernel module. You can also force the loading of the symbols of a kernel module with

```
sYmbol.AutoLoad.CHECK
sYmbol.AutoLoad.TOUCH "mymod"
```

Debugging the kernel module’s init routine:

To debug the kernel module’s init routine, you need to break into the kernel, right when the module is loaded. Either use the script “mod_debug.cmm” from the demo directory, or use the “Linux” menu item: “Linux” -> “Module Debugging” -> “Debug Module on init…”. See also chapter “**Linux Specific Menu**”.

Trapping Segmentation Violation

“Segmentation Violation” happens, if the code tries to access a memory location that cannot be mapped in an appropriate way. E.g. if a process tries to write to a read-only area, or if the kernel tries to read from a non-existent address. A segmentation violation is detected inside the kernel routine “do_page_fault()” (in arch/<*processor*>/mm/fault.c), if the mapping of page fails. If so, it (usually) jumps to a local label called “bad_area:”.

To trap segmentation violations, set a breakpoint onto the label “bad_area”. Some compilers don’t expose local labels; in this case, search the appropriate line in “do_page_fault()” and set the breakpoint manually, or modify the code to call a dummy function after “bad_area:” and set the breakpoint onto this dummy function.

Use the command **Var.Local** to display the local variables of “do_page_fault()”. This function is called with three parameters:

- “address” contains the memory address that caused the fault;
- “write” specifies, if it was a write (true) or read (false) access;
- “regs” is a structure containing the complete register set at the location, where the fault occurred.

When halted at “bad_area”, you may load the temporary register set of TRACE32 with these values. See the example script “segv.cmm” in the ~~/demo directory.

Use **Data.List**, **Var.Local** etc. then to analyze the fault.

As soon as debugging is continued (e.g. “Step”, “Go”, …), the original register settings at “bad_area” are restored.

TASK.CHECK Check awareness integrity

Format:	**TASK.CHECK**

This command does some integrity checks and displays the result of these.

No error should occur here.

TASK.DMESG Display the kernel ring buffer

Format:	**TASK.DMESG** [**/**<*option*>]
<*option*>:	**Level** <*log_level*> **Facility** <*log_facility*> **DETAILED** **COLOR**

Display the kernel messages in a window similar to the dmesg Linux command.

Level Restrict the displayed log messages to the specified log levels. This option can be used multiple times. Log levels names or numbers can be used with this options. Level names: EMERG, ALERT, CRIT, ERR, WARN, NOTICE, INFO, DEBUG.

Facility Restrict the displayed log messages to the specified log facilities. This option can be used multiple times. Log facility names or numbers can be used with this options. Facility names: KERN, USER, MAIL, DAEMON, AUTH, SYLOG, LPR, NEWS.

DETAILED Display the log level and facility as readable strings.

COLOR Enable the coloring of the log messages according to the log level and facility.

TASK.DTask

Display tasks

Format:	**TASK.DTask** [<*task_magic*>] [**/**<*option*>]
<*option*>:	**SORT** [<*item*>] **SORTUP** [<*item*>] **SORTDOWN** [<*item*>]

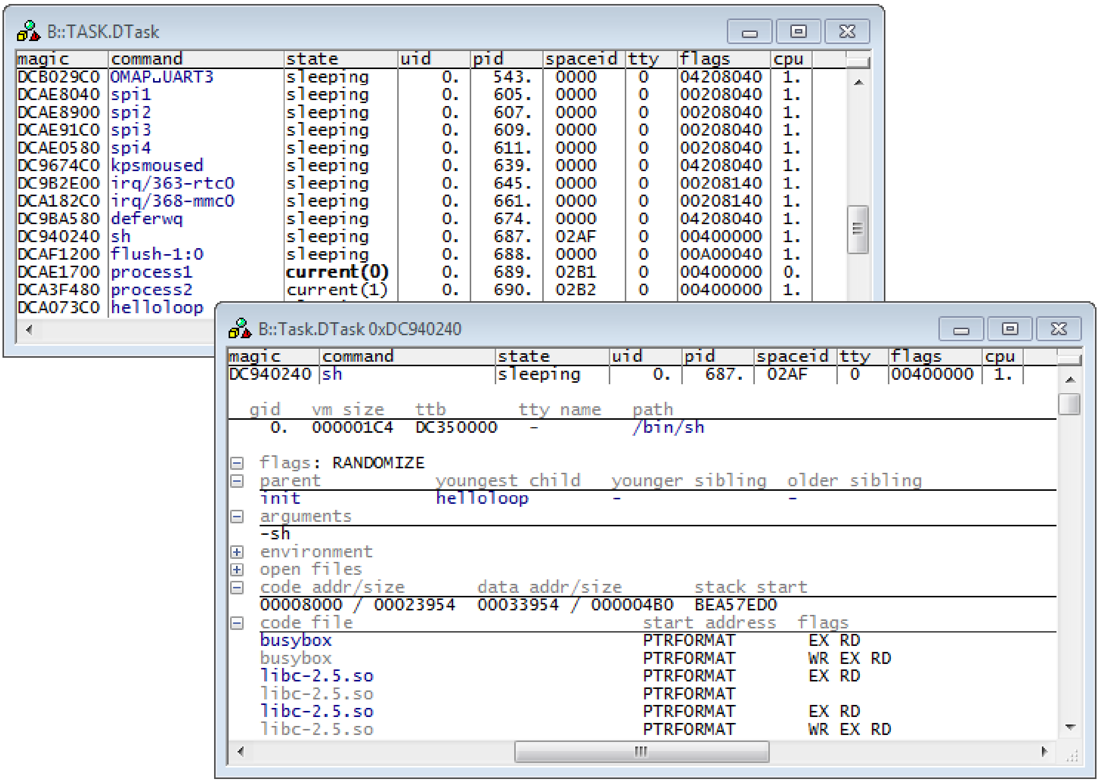
Displays the task table of Linux or detailed information about one specific task.
“Tasks” are activated processes.

SORT [<*item*>]
Sort up the TASK.DTask lines based on the given item. Item is a TASK.DTask column and can be MAGIC, CMD, STATE, UID, SPACEID or CPU. MAGIC is used if no sort item is specified.

SORTUP [<*item*>]
Same as SORT.

SORTDOWN [<*item*>]
Same as SORT, the lines are however sorted down.

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



B::TASK.DTask

```
magic    command        state       uid    pid   spaceid tty flags    cpu
DCB029C0 OMAP␣UART3     sleeping     0.   543.   0000    0  04208040 1.
DCAE8040 spi1           sleeping     0.   605.   0000    0  00208040 1.
DCAE8900 spi2           sleeping     0.   607.   0000    0  00208040 1.
DCAE91C0 spi3           sleeping     0.   609.   0000    0  00208040 1.
DCAE0580 spi4           sleeping     0.   611.   0000    0  00208040 1.
DC9674C0 kpsmoused      sleeping     0.   639.   0000    0  04208040 1.
DC9B2E00 irq/363-rtc0   sleeping     0.   645.   0000    0  00208140 1.
DCA182C0 irq/368-mmc0   sleeping     0.   661.   0000    0  00208140 1.
DC9BA580 deferwq        sleeping     0.   674.   0000    0  04208040 1.
DC940240 sh             sleeping     0.   687.   02AF    0  00400000 1.
DCAF1200 flush-1:0      sleeping     0.   688.   0000    0  00A00040 1.
DCAE1700 process1       current(0)   0.   689.   02B1    0  00400000 0.
DCA3F480 process2       current(1)   0.   690.   02B2    0  00400000 1.
DCA073C0 helloloop
```

B::Task.DTask 0xDC940240

```
magic    command        state       uid    pid   spaceid tty flags    cpu
DC940240 sh             sleeping     0.   687.   02AF    0  00400000 1.

  gid    vm size   ttb        tty name   path
   0.    000001C4  DC350000   -          /bin/sh

⊟ flags: RANDOMIZE
⊟ parent            youngest child   younger sibling   older sibling
  init              helloloop        -                 -
⊟ arguments
  -sh
⊞ environment
⊞ open files
⊟ code addr/size        data addr/size        stack start
  00008000 / 00023954   00033954 / 000004B0   BEA57ED0
⊟ code file                                  start address   flags
  busybox                                    PTRFORMAT        EX RD
  busybox                                    PTRFORMAT        WR EX RD
  libc-2.5.so                                PTRFORMAT        EX RD
  libc-2.5.so                                PTRFORMAT
  libc-2.5.so                                PTRFORMAT        EX RD
  libc-2.5.so                                PTRFORMAT        WR EX RD
```

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

“magic” is a unique ID, used by the OS awareness to identify a specific task (address of the 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.

A “task” in Linux maps to a non-threaded process and to each thread of a threaded process.

There are several different mechanisms how threads are managed inside the Linux kernel. The Linux awareness tries to detect them automatically, but this may fail on some systems. If the **TASK.DTask** window doesn’t show all threads correctly, declare the threading method manually with the **TASK.Option Threading** command.

TASK.DTB — Display the device tree blob

Format:	**TASK.DTB**

Display the device tree blob as a tree view.

TASK.DTS — Display the device tree source

Format:	**TASK.DTS**

Display the source code of the device tree.

TASK.NET — Display network devices

Format:	**TASK.NET**

Display network devices.

TASK.FS Display file system internals

Format:	**TASK.FS.**<*option*>
<*option*>:	**Types** [<*type_magic*>] | **Mount** | **MountDevs** [<*device_magic*>] | **PROC** | **PART**

This command displays internal data structures of the used file systems. See the appropriate command description for details.

Types [<*type_magic*>]	Display file system types. Without any arguments, this command displays all file system types that are currently registered in the Linux kernel. Specify a file system type magic to open a detailed view.
Mount	Display the current mount points.
MountDevs [<*device_magic*>]	Display mounted devices. Without any arguments, this command displays all currently mounted devices (i.e. super blocks). Specify a mounted device magic to open a detailed view.
PROC	Display /proc file system. PROC displays the contents of the “/proc” file system (procfs), even if it is not mounted.
SYS	Display /sys file system. SYS displays the contents of the “/sys” file system (sysfs), even if it is not mounted.
PART	Display the partition table.

TASK.MAPS Display process maps

Format:	**TASK.MAPS** <*process*>

Display the mapped memory regions and their access permissions similar to the /proc/<*pid*>/maps file.

TASK.MMU.SCAN Scan process MMU space

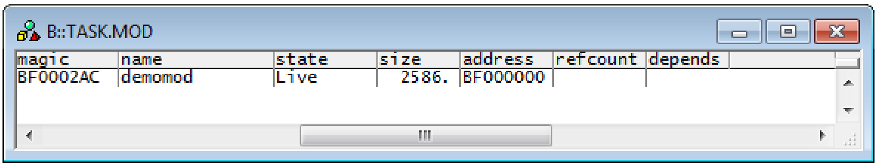
Format:	**TASK.MMU.SCAN** [<*process*>] (deprecated)

This command is deprecated. Please use **MMU.SCAN** instead.

TASK.MODule Display kernel modules

Format: **TASK.MODule**

Displays a table with all loaded kernel modules of Linux. The display is similar to the output of “lsmod”.



```
magic     name           state     size     address   refcount depends
BF0002AC  demomod        Live        2586.  BF000000
```

“magic” is a unique ID, used by the OS awareness to identify a module (address of the module struct).
“code addr” and “data addr” specify the address of the .text segment rsp. the .data segment.
The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

TASK.Option Set awareness options

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

<option>: **Threading** *<threading>* [**ON** | **OFF**]
MEMMAP *<mem_map>*
NameMode [**comm** | **TaskName** | **ARG0** | **ARG0COMM**]
THRCTX [**ON** | **OFF**]
SIDCACHE [**ON** | **OFF**]
CORES *<assignment>*
AutoLOAD ProcName [**TaskName** | **INODE**]

Sets various options to the awareness.

Threading Set the Threading type used by Linux.
TGROUP: threads are organized by the thread_group list (default). Set this to OFF if you encounter doubled thread entries.
See also chapter “**Debugging Threads**”.

MEMMAP deprecated

NameMode

Set the mode how the task names are evaluated.
comm: use the “comm” field in the task structure (default).
TaskName: use the name evaluated by TASK.NAME and “comm” (allows renaming of the tasks with TASK.NAME.Set).
ARG0: use the arg[0] statement of the process call.
ARG0COMM: use arg[0] as process name and “comm” as thread name (suitable for Android)

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 switch detection. See **Task Runtime Statistics**.

SIDCACHE

The Linux awareness uses a caching mechanism for the current space ID. In some rare conditions, this caching may fail. When setting this option to OFF, the caching is switched off; this is safer but slows down debugging (esp. single stepping).

CORES

This command can be used for SMP systems if the Linux kernel is using a different core assignment that the physical one e.g.
TASK.Option CORES 4 2 1 3

AutoLOAD

ProcName
Autoloader will use the name of the Task (selected by TASK.Option NameMode) per default to find the symbol file. With mode **INODE** the Autoloader will use the name of the real file. The latter mode is useful when symlinks (e.g. busybox) or prctl(PR_SET_NAME,...) is used to rename the process.

TASK.Process Display processes

Format: **TASK.Process** [<*process*>] [**/Open**]

Display processes with their threads. Threads are grouped to their belonging process.

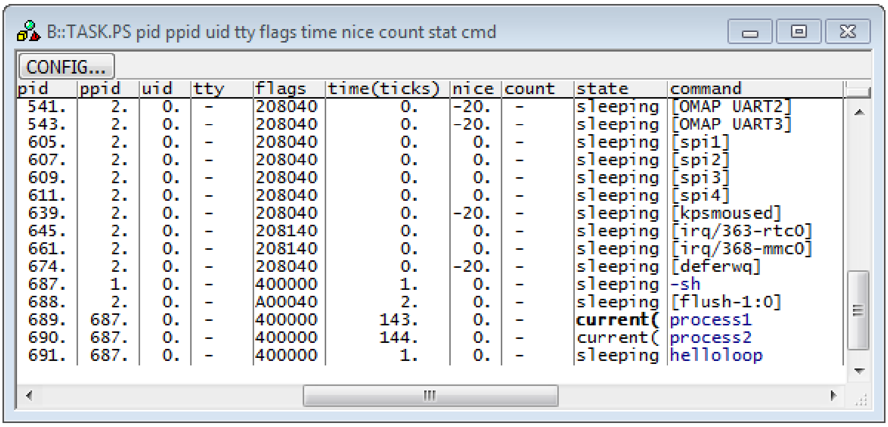
Without any arguments, this command displays a table with all created processes
Click on the plus sign in front of the process name to see the threads (if any).

Specify a process magic or name to see the threads of this process.

The option **/Open** expands all thread trees automatically. This option may be useful when printing the output of the window into a file (see WinPrint).

Format:	**TASK.PS** *<items>*
<items>:	**pid** | **ppid** | **uid** | **sid** | **pgid** | **cmd** | **pri** | **flags** | **tty** | **time** | **stat** | **nice** | **stackp** | **tmout** | **alarm** | **pending** | **blocked** | **vsz** | **rss** | **start** | **majflt** | **minflt** | **trs** | **drs** | **rss** | **count** | **nswap** | **ttb** | **vctxsw** | **nvctxsw**

Displays the process table of Linux.
The display is similar to the output of the “ps” shell command.



```
pid   ppid  uid  tty   flags   time(ticks) nice count  state    command
 541.    2.   0.  -    208040        0.    -20.   -    sleeping [OMAP UART2]
 543.    2.   0.  -    208040        0.    -20.   -    sleeping [OMAP UART3]
 605.    2.   0.  -    208040        0.      0.   -    sleeping [spi1]
 607.    2.   0.  -    208040        0.      0.   -    sleeping [spi2]
 609.    2.   0.  -    208040        0.      0.   -    sleeping [spi3]
 611.    2.   0.  -    208040        0.      0.   -    sleeping [spi4]
 639.    2.   0.  -    208040        0.    -20.   -    sleeping [kpsmoused]
 645.    2.   0.  -    208140        0.      0.   -    sleeping [irq/363-rtc0]
 661.    2.   0.  -    208140        0.      0.   -    sleeping [irq/368-mmc0]
 674.    2.   0.  -    208040        0.    -20.   -    sleeping [deferwq]
 687.    1.   0.  -    400000        1.      0.   -    sleeping -sh
 688.    2.   0.  -    A00040        2.      0.   -    sleeping [flush-1:0]
 689.  687.   0.  -    400000      143.      0.   -    current( process1
 690.  687.   0.  -    400000      144.      0.   -    current( process2
 691.  687.   0.  -    400000        1.      0.   -    sleeping helloloop
```

(Not available for all processors!)

The **TASK.sYmbol** command group helps to load and unload symbols and MMU settings of a given process or kernel module. In particular the commands are:

TASK.sYmbol.LOAD	Load process symbols and MMU
TASK.sYmbol.DELete	Unload process symbols and MMU
TASK.sYmbol.LOADMod	Load module symbols and MMU
TASK.sYmbol.DELeteMod	Unload module symbols and MMU
TASK.sYmbol.LOADLib	Load library symbols
TASK.sYmbol.DELeteLib	Unload library symbols
TASK.sYmbol.Option	Set symbol management options

TASK.sYmbol.DELete Unload process symbols and MMU

Format: **TASK.sYmbol.DELete** *<process>*

When debugging of a process is finished, or if the process exited, you should remove loaded process symbols and MMU entries. Otherwise the remaining entries may interfere with further debugging. This command deletes the symbols of the specified process and deletes its MMU entries.

<process>	Specify the process name or path (in quotes) or magic to unload the symbols of this process.

Example: When deleting the above loaded symbols with the command:

```
TASK.sYmbol.DELete "hello"
```

the debugger will internally execute the commands:

```
TRANSlation.Delete 24.:0--0xffffffff
sYmbol.Delete \\hello
```

TASK.sYmbol.DELeteLib — Unload library symbols

Format: **TASK.sYmbol.DELeteLib** *<process> <library>*

When debugging of a library is finished, or if the library is removed from the kernel, you should remove loaded library symbols. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified library.

<process> Specify the process to which the desired library belongs (name in quotes or magic).

<library> Specify the library name in quotes. The library name **must** match the name as shown in **TASK.DTask** *<process>*, ”code files”.

Example:

```
TASK.sYmbol.DELeteLib "hello" "libc-2.2.1.so"
```

See also chapter “**Debugging Into Shared Libraries**”.

TASK.sYmbol.DELeteMod — Unload module symbols and MMU

Format: **TASK.sYmbol.DELeteMod** *<module>*

Specify the module name (in quotes) or magic to unload the symbols of this kernel module.

When debugging of a module is finished, or if the module is removed from the kernel, you should remove loaded module symbols and MMU entries. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified module and deletes its MMU entries.

Example:

```
TASK.sYmbol.DELeteMod "pcmcia_core"
```

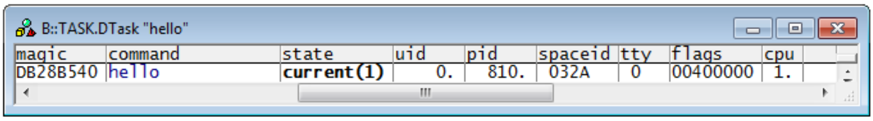
See also chapter “**Debugging Kernel Modules**”.

Format: **TASK.sYmbol.LOAD** <*process*>

Specify the process name or path (in quotes) or magic to load the symbols of this process.

In order to debug a user process, the debugger needs the symbols of this process, and the process specific MMU settings (see chapter “Debugging User Processes”).
This command retrieves the appropriate space ID, loads the symbol file of an existing process and reads its MMU entries. Note that this command works only with processes that are already loaded in Linux (i.e. processes that show up in the **TASK.DTask** window).

Example: If the **TASK.DTask** window shows the entry:



magic	command	state	uid	pid	spaceid	tty	flags	cpu
DB28B540	hello	current(1)	0.	810.	032A	0	00400000	1.

```
TASK.sYmbol.LOAD "hello"
```

If the symbol autoloader is enabled for processes (**TASK.sYmbol.Option AutoLoad Process**), the following commands will be internally executed:

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

Otherwise, the following command will be internally executed:

```
Data.LOAD.Elf hello 24.:0 /GNU /NoCODE /NoClear
```

If the symbol file is not within the current directory, specify the path to the ELF file. E.g.:

```
TASK.sYmbol.LOAD "C:\mypath\hello"
```

Loads the ELF file “C:\mypath\hello” of the process “hello”. Note that the process name must equal to the filename of the ELF file.

TASK.sYmbol.LOADLib Load library symbols

Format: **TASK.sYmbol.LOADLib** <*process*> <*library*>

As first parameter, specify the process to which the desired library belongs (name in quotes or magic). Specify the library name in quotes as second parameter. The library name **must** match the name as shown in **TASK.DTask** <*process*>, ”code files”.

In order to debug a library, the debugger needs the symbols of this library, relocated to the correct addresses where Linux linked this library. This command retrieves the appropriate load addresses and loads the .so symbol file of an existing library. Note that this command works only with libraries that are already loaded in Linux (i.e. libraries that show up in the **TASK.DTask** <*process*> window).

Example:

```
TASK.sYmbol.LOADLib "hello" "libc-2.2.1.so"
```

See also chapter “**Debugging Into Shared Libraries**”.

TASK.sYmbol.LOADMod Load module symbols and MMU

Format: **TASK.sYmbol.LOADMod** <*module*>

Specify the module name (in quotes) or magic to load the symbols of this module.

In order to debug a kernel module, the debugger needs the symbols of this module, and the module specific MMU settings. This command retrieves the appropriate load addresses, loads the .o/.ko symbol file of an existing module and reads its MMU entries. Note that this command works only with modules that are already loaded in Linux (i.e. modules that show up in the **TASK.MODule** window).

Example:

```
TASK.sYmbol.LOADMod "pcmcia_core"
```

See also chapter “**Debugging Kernel Modules**”.

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

<option>:
LOADCMD *<command>*
LOADMCMD *<command>*
LOADLCMD *<command>*
MMUSCAN [**ON** | **OFF**]
AutoLoad *<option>*
ROOTPATH *<path>*
PROCRANGE *<start> <size>*
LIBFLAGS *<action> <flags>*

Set a specific option to the symbol management.

LOADCMD:
This setting is only active, if the symbol autoloader for processes is off.
TASK.sYmbol.LOAD uses a default load command to load the symbol file of the process. This loading command can be customized using this option with the command enclosed in quotes. Two parameters are passed to the command in a fixed order:

%s	name of the process
%x	space ID of the process

Examples:

```
TASK.sYmbol.Option LOADCMD "Data.LOAD.Elf %s 0x%x:0 /NoCODE /NoClear"

TASK.sYmbol.Option LOADCMD "DO myloadscript %s 0x%x"
```

LOADMCMD:
This setting is only active, if the symbol autoloader for kernel modules is off.
TASK.sYmbol.LOADMod uses a default load command to load the symbol file of the module. This loading command can be customized using this option with the command enclosed in quotes. Three parameters are passed to the command in a fixed order:

Examples:

%s	name of the module
%x	start (=code) address of the module
%x	data address of the module (if applicable)

```
TASK.sYmbol.Option LOADMCMD "Data.LOAD.Elf %s /NoCODE /NoClear /GCC3
     /RELOC .text AT 0x%x /RELOC .data AT 0x%x /RELOC .bss AFTER .data"

TASK.sYmbol.Option LOADMCMD "do myloadmscript %s 0x%x 0x%x"
```

LOADLCMD:
This setting is only active, if the symbol autoloader for libraries is off.
TASK.sYmbol.LOADLib uses a default load command to load the symbol file of the library. This loading command can be customized using this option with the command enclosed in quotes. Three parameters are passed to the command in a fixed order:

%s	name of the library
%x	space ID of the library
%x	load address of the library

Examples:

```
TASK.sYmbol.Option LOADLCMD "D.LOAD.Elf %s 0x%x:0x%x /NoCODE /NoClear"

TASK.sYmbol.Option LOADMCMD "DO myloadlscript %s 0x%x 0x%x"
```

AutoLoad:
This option controls, which components are checked and managed by the **symbol autoloader**:

Process	check processes
Library	check all libraries of all processes
Module	check kernel modules
CurrLib	check only libraries of current process
ProcLib *<process>*	check libraries of specified process
ALL	check processes, libraries and kernel modules
NoProcess	don’t check processes
NoLibrary	don’t check libraries
NoModule	don’t check modules
VM	load process code to debugger virtual memory
NOVM	don’t load process code to debugger virtual memory
NONE	check nothing.

The options are set ***additionally***, not removing previous settings.

The default is “Process”, i.e. the only processes are checked by the symbol autoloader (if configured).

Example:

```
; check processes and kernel modules
TASK.sYmbol.Option AutoLoad Process
TASK.sYmbol.Option AutoLoad Module
```

MMUSCAN:

This option controls, if the symbol loading mechanisms of TASK.sYmbol scan the MMU page tables of the loaded components, too. Default is OFF for use with **TRANSlation.TableWalk**. Set this to ON if you do not use the table walk.

ROOTPATH:

If this option is set, the symbol autoloader tries to find the symbol files in the directory, starting with the path given with this option, and appending the path that the file uses on the target.
This is essentially useful, if the root path of your target maps to a specific path on your development machine. The symbol autoloader will then find all files automatically in the given tree.

Example, e.g. your root path on the target ("/") maps to /nfsroot/device/root on your host:

```
TASK.sYmbol.Option ROOTPATH "/nfsroot/device/root"
```

PROCRANGE:

The symbol autoloader uses a dummy address range for a process, if it cannot determine the correct range from the task structure. This is especially necessary when loading the symbols before starting the process. In some rare cases, the built-in dummy area does not fit to the actual Linux system. This option then specifies the start address and size of the dummy process range.
Use this option *only* if you are told to do so.

LIBFLAGS:

Set the shared object flags that control the loading of the debug symbols. Possible actions are:

LOAD set the flags that must be present so that the debug symbols of the library will be loaded.

NOLOAD set the flags that will prevent the debug symbols of the library from being loaded.

Please refer to the kernel source file `include/linux/mm.h` for more information about the possible flags. LOAD is set by default to 0x00000004 (VM_EXEC). NOLOAD is set by default to 0x00101002 (VM_EXECUTABLE|VM_ACCOUNT|VM_WRITE).

Please do not use this option if unsure.

Format:	**TASK.VMAINFO**

Displays vmalloc info.

TASK.Watch — Watch processes

The **TASK.Watch** command group builds a watch system that watches your Linux target for specified processes. It loads and unloads process symbols automatically. Additionally it covers process creation and may stop watched processes at their entry points.

In particular the **TASK.Watch** commands are:

TASK.Watch.View	Activate watch system and show watched processes
TASK.Watch.ADD	Add process to watch list
TASK.Watch.DELete	Remove process from watch list
TASK.Watch.DISable	Disable watch system
TASK.Watch.ENable	Enable watch system
TASK.Watch.DISableBP	Disable process creation breakpoints
TASK.Watch.ENableBP	Enable process creation breakpoints
TASK.Watch.Option	Set watch system options

TASK.Watch.ADD — Add process to watch list

Format:	**TASK.Watch.ADD** *<process>*

Adds a process to the watch list.

<process>	Specify the process name (in quotes) or magic.

Please see **TASK.Watch.View** for details.

TASK.Watch.DELete — Remove process from watch list

Format:	**TASK.Watch.DELete** *<process>*

Removes a process from the watch list.

Please see **TASK.Watch.View** for details.

TASK.Watch.DISable Disable watch system

Format: **TASK.Watch.DISable**

Disables the complete watch system. The watched processes list is no longer checked against the target and is not updated. You'll see the **TASK.Watch.View** window grayed out.

This feature is useful if you want to keep process symbols in the debugger, even if the process terminated.

TASK.Watch.DISableBP Disable process creation breakpoints

Format: **TASK.Watch.DISableBP**

Prevents the debugger from setting on-chip breakpoints for the detection of process creation. After executing this command, the target will run in real-time. However, the watch system can no longer detect process creation. Automatic loading of process symbols will still work.

This feature is useful if you'd like to use the on-chip breakpoints for other purposes.

Please see **TASK.Watch.View** for details.

TASK.Watch.ENable Enable watch system

Format: **TASK.Watch.ENable**

Enables the previously disabled watch system. It enables the automatic loading of process symbols as well as the detection of process creation.

Please see **TASK.Watch.View** for details.

TASK.Watch.ENableBP — Enable process creation breakpoints

Format:	**TASK.Watch.ENable**

Enables the previously disabled on-chip breakpoints for detection of process creation.

Please see **TASK.Watch.View** for details.

TASK.Watch.Option — Set watch system options

Format:	**TASK.Watch.Option** *<option>*
<option>:	**BreakFunC** *<function>*

Set various options to the watch system.

BreakFunC

Set the breakpoint location for process creation detection.
The default value is “set_binfmt”.
Example: `TASK.Watch.Option BreakFunC "set_binfmt"`

BreakOptC

Set the option that is used to set the breakpoint for process creation detection.
The default value is “/Onchip”
Example: `TASK.Watch.Option BreakOptC "/Soft"`

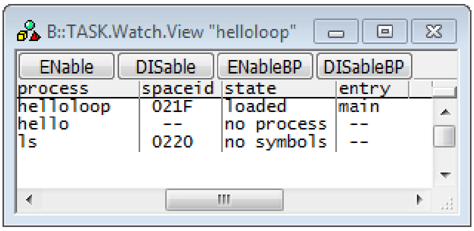
Please see **TASK.Watch.View** for details.

TASK.Watch.View   Show watched processes

Format: **TASK.Watch.View** [*<process>*]

Activates the watch system for processes and shows a table of the watched processes.

NOTE: **This feature may affect the real-time behavior of the target application!** Please see below for details.



B::TASK.Watch.View "helloloop"

```
process    spaceid state       entry
helloloop   021F   loaded      main
hello        --    no process   --
ls          0220   no symbols   --
```

<process> Specify a process name for the initial process to be watched.

Description of Columns in the TASK.Watch.View Window

process	The name of the process to be watched.
spaceid	The current space ID (= process ID) of the watched process. If grayed, the debugger is currently not able to determine the space ID of the process (e.g. the target is running).
state	The current watch state of the process. If grayed, the debugger is currently not able to determine the watch state. **no process**: The debugger couldn’t find the process in the current Linux process list. **no symbols**: The debugger found the process and loaded the MMU settings of the process but couldn’t load the symbols of the process (most likely because the corresponding symbol files were missing). **loaded**: The debugger found the process and loaded the process’s MMU settings and symbols.
entry	The process entry point, which is `main()`. If grayed, the debugger is currently not able to detect the entry point or is unable to set the process entry breakpoint (e.g. because it is disabled with **TASK.Watch.DISableBP**).

The watch system for processes is able to automatically load and unload the symbols of a process and its MMU settings, depending on their state in the target. Additionally, the watch system can detect the creation of a process and halts the process at its entry point.

TASK.Watch.ADD	Add processes to the watch list.
TASK.Watch.DELete	Remove processes from the watch list.

The watch system for processes is active as long as the **TASK.Watch.View** window is open or iconized. As soon as this window is closed, the watch system will be deactivated.

Automatic Loading and Unloading of Process Symbols

In order to detect the current processes, the debugger must have full access to the target, i.e. the target application must be stopped (with one exception, see below for creation of processes). As long as the target runs in real time, the watch system is not able to get the current process list, and the display will be grayed out (inactive).

If the target is halted (either by hitting a breakpoint, or by halting it manually), the watch system starts its work. For each of the processes in the watch list, it determines the state of this process in the target.

If a process is active on the target, which was previously not found there, the watch system scans its MMU entries and loads the appropriate symbol files. In fact, it executes **TASK.sYmbol.LOAD** for the new process.

If a watched process was previously loaded but is no longer found on the Linux process list, the watch system unloads the symbols and removes the MMU settings from the debugger MMU table. The watch system executes **TASK.sYmbol.DELete** for this process.

If the process was previously loaded and is now found with another space ID (e.g. if the process terminated and started again), the watch system first removes the process symbols and reloads them to the appropriate space ID.

You can disable the loading / unloading of process symbols with the command **TASK.Watch.DISable**.

Detection of Process Creation

To halt a process at its main entry point, the watch system can detect the process creation and set the appropriate breakpoints.

To detect the process creation, the watch system sets an on-chip breakpoint on a kernel function that is called upon creation of processes. Every time the breakpoint is hit, the debugger checks if a watched process is started. If not, it simply resumes the target application. If the debugger detects the start of a newly created (and watched) process, it sets an on-chip breakpoint onto the main entry point of the process (`main()`) and resumes the target application. A short while after this, the main breakpoint will hit and halt the target at the entry point of the process. The process is now ready to be debugged.

NOTE:	This feature uses one permanent on-chip breakpoint and one temporary on-chip breakpoint when a process is created. Please ensure that at least those two on-chip breakpoints are available when using this feature. Upon every process creation, the target application is halted for a short time and resumed after searching for the watched processes. **This impacts the real-time behavior of your target.**

If you don't want the watch system to set breakpoints, you can disable them with the command **TASK.Watch.DISableBP**. Of course, detection of process creation won't work then.

Linux PRACTICE Functions

There are special definitions for Linux specific PRACTICE functions.

TASK.ARCHITECTURE() Target architecture

Syntax: **TASK.ARCHITECTURE()**

Returns the target architecture of the Linux awareness.

Return Value Type: String.

TASK.CONFIG() OS awareness configuration information

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

Parameter and Description:

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

Return Value Type: Hex value.

TASK.CURRENT() Magic or space ID of current task

Syntax: **TASK.CURRENT(process | spaceid)**

Parameter and Description:

process	**Parameter Type**: String (***without*** quotation marks).
spaceid	**Parameter Type**: String (***without*** quotation marks).

Return Value Type: Hex value.

TASK.ERROR.CODE() Awareness error code

Syntax: **TASK.ERROR.CODE()**

Checks for awareness errors and returns the bit wise OR of the following error codes.

Return Value Type: Hex value.

Return Value and Description:

0	No error.
1	Failed to detect kernel symbols.
2	Failed to detect kernel structures.
4	Failed to detect kernel structure members.
8	Pointer size does not fit.

TASK.ERROR.HELP() Awareness error help ID

Syntax: **TASK.ERROR.HELP()**

Checks for awareness errors and returns the error help ID.

Return Value Type: String.

TASK.LIB.ADDRESS() Library load address

Syntax: **TASK.LIB.ADDRESS("***<library_name>***", ***<process_magic>***)**

Returns the load address of the library, loaded by the specified process.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.LIB.CODESIZE() Library code size

Syntax: **TASK.LIB.CODESIZE("***<library_name>***",** *<process_magic>***)**

Returns the code size of the library, loaded by the specified process.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.LIB.PATH() Library target path and name

Syntax: **TASK.LIB.PATH("***<library_name>***",** *<process_magic>***)**

Returns the path and file name of the library on the target, loaded by the specified process.

Parameter and Description:

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

Return Value Type: String.

TASK.MOD.CODEADDR() Code start address of module

Syntax: **TASK.MOD.CODEADDR("***<module_name>***")**

Returns the code start address of the module.

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

Return Value Type: Hex value.

TASK.MOD.DATAADDR() Data start of module

Syntax: **TASK.MOD.DATAADDR("***<module_name>***")**

Returns the data start of the module.

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

Return Value Type: Hex value.

TASK.MOD.SIZE() Size of module

Syntax: **TASK.MOD.SIZE("***<module_name>***")**

Returns the size of the module.

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

Return Value Type: Hex value.

TASK.MOD.MAGIC() Magic value of module

Syntax: **TASK.MOD.MAGIC("***<module_name>***")**

Returns the “magic” value of the module.

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

Return Value Type: Hex value.

TASK.MOD.MCB() Structure address of module

Syntax: **TASK.MOD.MCB(***<module_magic>***)**

Returns the module’s structure address.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.MOD.NAME() Name of module magic

Syntax: **TASK.MOD.NAME(***<module_magic>***)**

Returns the name of the given module magic.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.MOD.SECTION() Address of a specified module's section

Syntax: **TASK.MOD.SECTION("***<section_name>***",***<module_magic>***)**

Returns the address of the section of the specified kernel module.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.MOD.SECNAME() Name of a module section with a given number

Syntax: **TASK.MOD.SECNAME(***<module_magic>***,***<section_number>***)**

Returns the name of the section specified by the iteration number.

Parameter and Description:

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

Return Value Type: String.

TASK.MOD.SECADDR() Address of a module section with a given number

Syntax: **TASK.MOD.SECADDR(***<module_magic>***,***<section_number>***)**

Returns the address of the section specified by the iteration number.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.OS.VERSION() Version of the used Linux OS

Syntax: **TASK.OS.VERSION()**

Returns the version of the used Linux OS.

Return Value Type: Hex value.

TASK.PROC.CODEADDR() Code start address of process

Syntax: **TASK.PROC.CODEADDR("***<process_name>***")**

Returns the code start address of the process.

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

Return Value Type: Hex value.

TASK.PROC.CODESIZE() Code size of process

Syntax: **TASK.PROC.CODESIZE("***<process_name>***")**

Returns the code size of the process.

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

Return Value Type: Hex value.

TASK.PROC.DATAADDR() Data start address of process

Syntax: **TASK.PROC.DATAADDR("***<process_name>***")**

Returns the data start address of the process.

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

Return Value Type: Hex value.

TASK.PROC.DATASIZE() Data size of process

Syntax: **TASK.PROC.DATASIZE("***<process_name>***")**

Returns the data size of the process.

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

Return Value Type: Hex value.

TASK.PROC.FileName() Filename of process

Syntax: **TASK.PROC.FileName(***<task_magic>***)**

Returns the name of the main application file.

Parameter Type: Hex value.

Return Value Type: String.

TASK.PROC.LIST() List of processes

Syntax: **TASK.PROC.LIST(*<magic_value>*)**

Returns the next magic in the process list. Returns zero if no further process is available.

Parameter Type: Decimal or hex or binary value.

Parameter and Description:

0	Specify zero for the first process. E.g. `PRINT TASK.PROC.LIST(0)       ; e.g. prints 1234`

<magic_value>	Get the magic value of the process in list following the parameter. E.g.
	`PRINT TASK.PROC.LIST(0x1234) ; e.g. prints 2456`
	`PRINT TASK.PROC.LIST(0x2456) ; e.g. prints 0`

Return Value Type: Hex value.

Example:

```
&magicvalue=TASK.PROC.LIST(0)
WHILE &magicvalue!=0
(
  PRINT &magicvalue
  &magicvalue=TASK.PROC.LIST(&magicvalue)
)
```

TASK.PROC.MAGIC() Magic value of process

Syntax: **TASK.PROC.MAGIC("*<process_name>*")**

Returns the “magic” value of the process.

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

Return Value Type: Hex value.

TASK.PROC.MAGIC2SID() Space ID of process

Syntax: **TASK.PROC.MAGIC2SID(***<process_magic>***)**

Returns the space ID of the specified process.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PROC.NAME() Name of process

Syntax: **TASK.PROC.NAME(***<process_magic>***)**

Returns the name of the specified process.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.PROC.NAME2TRACEID() Trace ID of process

Syntax: **TASK.PROC.NAME2TRACEID("***<process_name>***")**

Returns the trace ID of the specified process.

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

Return Value Type: Hex value.

TASK.PROC.PATH() Path and file name of executable on target

Syntax: **TASK.PROC.PATH(***<process_magic>***)**

Returns the path and file name of the executable on the target.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.PROC.PSID() Process ID

Syntax: **TASK.PROC.PSID("***<process_name>***")**

Returns the process ID of the specified process.

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

Return Value Type: Hex value.

TASK.PROC.SID2MAGIC() Magic value of process

Syntax: **TASK.PROC.SID2MAGIC(***<space_id>***)**

Returns the “magic” value of the process that has the given space ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PROC.SPACEID() Space ID of process

Syntax: **TASK.PROC.SPACEID("***<process_name>***")**

Returns the space ID of the specified process.

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

Return Value Type: Hex value.

TASK.PROC.TCB() Control structure address of task

Syntax: **TASK.PROC.TCB(***<process_magic>***)**

Returns the task’s control structure address.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PROC.TRACEID() Trace ID of process

Syntax: **TASK.PROC.TRACEID(***<process_magic>***)**

Returns the trace ID of the specified process.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.PROC.VMAEND() End address of a process virtual memory area

Syntax: **TASK.PROC.VMAEND("***<process_name>***",** *<address>***)**

Returns the end address of the given process virtual memory area holding the given address.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.PROC.VMASTART() Start address of a process virtual memory area

Syntax: **TASK.PROC.VMASTART("***<process_name>***",** *<address>***)**

Returns the start address of the given process virtual memory area holding the given address.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.VERSION.BUILD() Build number of Linux awareness

Syntax: **TASK.VERSION.BUILD()**

Returns the build number of the Linux awareness.

Return Value Type: String.

TASK.VERSION.DATE() Build date of Linux awareness

Syntax: **TASK.VERSION.DATE()**

Returns the build date of the Linux awareness.

Return Value Type: String.

Error Messages

While using the Linux awareness, error messages may occur. This chapter explains the meanings of the messages, and what could cause the error.

No error.

No error was detected by the awareness.

Failed to detect kernel symbols.

The awareness couldn’t find the necessary kernel symbols.

Are the symbols of the Linux kernel loaded?
Are the kernel symbols still accessible?
Maybe a missing “/NoClear” option when loading symbols of other components?

Failed to detect kernel structures.

The awareness couldn’t find the HLL information of the kernel.

Is the kernel compiled with debug information?

Failed to detect kernel structure members.

The awareness couldn’t find the HLL structure member information of the kernel structures.

Is the kernel fully compiled with debug information?
Try to execute “`sYmbol.CLEANUP`”.

Unknown Error Id.

An error was detected, but the error code couldn’t be resolved.

Probably a bug in the Linux awareness. Please execute “`PRINT task.error.code()`” and report it to LAUTERBACH.

Appendix

Appendix A: insmod patch for Linux 2.4

This patch provides section information in the kernel module headers, to ease symbol loading for kernel modules in Linux 2.4.
The patch applies to two function in the file modutils/obj/obj_reloc.c

Change the function obj_load_size():

```
unsigned long
obj_load_size (struct obj_file *f)
{
     unsigned long dot = 0;
     struct obj_section *sec;

     /* TRACE32: next lines inserted */

     unsigned long strsize = 0;

     /* calculate space for section names in front of sections */
     for (sec = f->load_order; sec ; sec = sec->load_next)
     {
          strsize += 4 + strlen(sec->name)+1;   /* address plus zero
          terminated name */
          if (strsize & 3) strsize += 4-(strsize&3); /* align to 32bit */
     }
     strsize += 8;   /* start and end marker */

     /* preserve first (struct module) section */
     sec = f->load_order;
     {
          ElfW(Addr) align;

          align = sec->header.sh_addralign;
          if (align && (dot & (align - 1)))
          dot = (dot | (align - 1)) + 1;

          sec->header.sh_addr = dot;
          dot += sec->header.sh_size;
     }
     sec = sec->load_next;
     /* add section name size */
     if (dot & 3) dot += 4-(dot&3);
     dot += strsize;

     /* TRACE32: end insert */

     /* Finalize the positions of the sections relative to one */
     /* another.*/

     /* TRACE32: line changed: */
     /*for (sec = f->load_order; sec ; sec = sec->load_next)*/
     for (; sec ; sec = sec->load_next)
     {
          ElfW(Addr) align;

          align = sec->header.sh_addralign;
          if (align && (dot & (align - 1)))
          dot = (dot | (align - 1)) + 1;

          sec->header.sh_addr = dot;
          dot += sec->header.sh_size;
     }

     return dot;
}
```

Change the function obj_create_image():

```
int
obj_create_image (struct obj_file *f, char *image)
{
     struct obj_section *sec;
     ElfW(Addr) base = f->baseaddr;

     /* TRACE32: next lines inserted */

     struct obj_section *sec2;

     /* preserve first (struct module) section */
     sec = f->load_order;
     {
          char *secimg = image;

          if (sec->contents != 0)
          {
               secimg = image + (sec->header.sh_addr - base);

               /* Note that we allocated data for NOBITS sections */
               /* earlier. */
               memcpy(secimg, sec->contents, sec->header.sh_size);
          }

          /* create section names in front of sections */
          secimg += (sec->header.sh_size);
          if ((int)secimg & 3) secimg += 4-((int)secimg&3);
          strncpy (secimg, "T32S", 4);  /* start marker */
          secimg += 4;
          for (sec2 = f->load_order; sec2 ; sec2 = sec2->load_next)
          {
               *((unsigned long*) secimg) = sec2->header.sh_addr;
               secimg += 4;
               strcpy (secimg, sec2->name);
               secimg += strlen (sec2->name) + 1;
               if ((int)secimg & 3) secimg += 4-((int)secimg&3);
          }
          strncpy (secimg, "T32E", 4);  /* end marker */
     }

     sec = sec->load_next;

/* TRACE32: end insert */
```

```
        /* TRACE32: line changed: */
        /* for (sec = f->load_order; sec ; sec = sec->load_next)*/
        for (; sec ; sec = sec->load_next)
        {
                char *secimg;

                if (sec->contents == 0)
                        continue;

                secimg = image + (sec->header.sh_addr - base);

                /* Note that we allocated data for NOBITS sections */
                /* earlier. */
                memcpy(secimg, sec->contents, sec->header.sh_size);
        }

        return 1;
}
```

FAQ

Please refer to https://support.lauterbach.com/kb.