



[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

TRACE32 Documents	
RTOS Debugger	
RTOS Debugger for Linux - Stop Mode	1
Debugging Modes for Embedded Linux	3
Run Mode Debugging with TRACE32 as GDB Front-end	3
Stop Mode Debugging via JTAG	4
Integrated Run & Stop Mode Debuggings via JTAG	4
Supported Debugging Modes sorted by Architectures	5
Overview	6
Configuration	7
Manual Configuration	7
Automatic Configuration	7
Quick Configuration Guide	8
Hooks & Internals in Linux	8
Features	9
Display of Kernel Resources	9
Task Related Breakpoints	9
Task Context Display	10
MMU Support	11
Space IDs	11
MMU Declaration	12
Scanning System and Processes	13
Symbol Autoloader	14
SMP Support	15
Dynamic Task Performance Measurement	15
Task Runtime Statistics	16
Task State Analysis	18
Function Runtime Statistics	19
Linux Specific Menu	20
Basics of HW-based Debugging	23
Debugging Linux Kernel and User Processes	23
Linux Kernel	24
Downloading the Kernel	24

Debugging the Kernel Startup	25
Debugging the Kernel	25
User Processes	26
Debugging the Process	26
Debugging into Shared Libraries	28
Debugging Linux Threads	29
On Demand Paging	29
Kernel Modules	36
Trapping Segmentation Violation	39
Linux Commands	40
TASK.DTask	Display tasks 40
TASK.PS	Display 'ps' output 41
TASK.MODule	Display kernel modules 41
TASK.FS	Display file system internals 42
TASK.FS.Types	Display file system types 42
TASK.FS.Mount	Display mount points 42
TASK.FS.MountDevs	Display mounted devices 42
TASK.FS.PROC	Display /proc file system 42
TASK.TASKState	Mark task state words 43
TASK.MMU.SCAN	Scan process MMU space 43
TASK.sYmbol	Process/Module symbol management 44
TASK.sYmbol.LOAD	Load process symbols & MMU 44
TASK.sYmbol.DELeTe	Unload process symbols & MMU 45
TASK.sYmbol.LOADMod	Load module symbols & MMU 45
TASK.sYmbol.DELeTeMod	Unload module symbols & MMU 46
TASK.sYmbol.LOADLib	Load library symbols 47
TASK.sYmbol.DELeTeLib	Unload library symbols 47
TASK.sYmbol.Option	Set symbol management options 48
TASK.Watch	Watch processes 50
TASK.Watch.View	Show watched processes 51
TASK.Watch.ADD	Add process to watch list 53
TASK.Watch.DELeTe	Remove process from watch list 53
TASK.Watch.DISable	Disable watch system 53
TASK.Watch.DISableBP	Disable process creation breakpoints 54
TASK.Watch.ENABLE	Enable watch system 54
TASK.Watch.ENABLEBP	Enable process creation breakpoints 55
TASK.Watch.Option	Set watch system options 55
TASK.Option	Set awareness options 56
Linux PRACTICE Functions	57
Error Messages	59
Appendix	60
Appendix A: insmod patch for Linux 2.4	60

Debugging Modes for Embedded Linux

TRACE32 provides 3 modes for debugging embedded Linux: Run Mode Debugging, Stop Mode Debugging and Integrated Run & Stop Mode Debugging.

Run Mode Debugging with TRACE32 as GDB Front-end

Pure software debugger, no TRACE32 hardware required.

The TRACE32 software is licensed by a USB dongle.

Requires debug agent running as Linux process on the target. Available solutions:

- Single-process debugging via standard **gdbserver** as debug agent.
- Multi-process debugging via **t32server** as debug agent.

The t32server allows to attach a separate gdbserver to several processes thus providing multi-process debugging. The t32server is provided by LAUTERBACH.

TRACE32 GUI communicates with gdbserver / t32server on target via RS232, ethernet or BLUETOOTH.

The TRACE32 GDB Front-end works in so-called **Run Mode Debugging**: At a breakpoint only the selected process is stopped, while the kernel and all other processes continue to run.

Stop Mode Debugging via JTAG

Stop Mode Debugging requires a TRACE32 JTAG debugger hardware.

If debugging is performed via the JTAG interface, TRACE32 works in so-called **Stop Mode Debugging**. At a breakpoint the CPU and thus the whole target system is stopped.

The main advantages of Stop Mode Debugging are:

- Since the only requirement for Stop Mode Debugging is a functioning JTAG interface, debugging can start at the reset vector.
- Since TRACE32 provides both Linux and MMU support, debugging of the kernel and beyond process boundaries is possible.

Integrated Run & Stop Mode Debuggings via JTAG

Integrated Run & Stop Mode Debugging requires a TRACE32 JTAG debugger hardware.

If debugging is performed via the JTAG interface, TRACE32 can be configured:

- to allow Stop Mode Debugging via JTAG.
- to allow Run Mode Debugging via the t32server running as debug agent on the target.
- to allow, if required, to switch between the two debugging modes.

TRACE32 communicates with the t32server:

- via ethernet for the **MIPS** and **PowerPC** architecture.
- via DCC (Debug Communications Channel) for the **ARM** architecture. Since the ARM-JTAG interface provides a DCC, Integrated Run & Stop Mode Debugging can function with JTAG as the only communication interface.

Supported Debugging Modes sorted by Architectures

Architecture	GDB Front-end	Stop Mode Debugging	Integrated Run & Stop Mode Debugging
ARM	Yes, refer to "TRACE32 GDB Front-end for ARM" (monitor_gdb_arm.f m).	Yes, read this manual.	Yes, via DCC, refer to "RTOS Debugger for Linux - Run Mode" (rtos_linux_run.pdf).
ColdFire	—	Yes, read this manual.	—
I386	Yes	—	—
MIPS	Yes, refer to "TRACE32 GDB Front-end for MIPS" (monitor_gdb_mips.f m).	Yes, read this manual.	Yes, via ethernet, refer to "RTOS Debugger for Linux - Run Mode" (rtos_linux_run.pdf).
PowerPC	Yes, refer to "TRACE32 GDB Front-end for PPC" (monitor_gdb_ppc.f m).	Yes, read this manual.	YES. via ethernet, refer to "RTOS Debugger for Linux - Run Mode" (rtos_linux_run.pdf).
SH4	—	Yes, read this manual.	—
XScale	Yes	Yes, read this manual.	—

B::TASK.DTask							
magic	command	state	uid	pid	ttt	flags	nice
01072000	swapper	running	0.	0.	0	00000100	20.
010FE000	init	sleeping	0.	1.	0	00000100	0.
010FC000	keventd	sleeping	0.	2.	0	00000040	0.
010F8000	ksoftirqd_CPU0	sleeping	0.	3.	0	00000040	19.
010F6000	kswapd	sleeping	0.	4.	0	00000840	0.
010F4000	bdfush	sleeping	0.	5.	0	00000040	0.
010F2000	kupdated	sleeping	0.	6.	0	00000040	0.
010C6000	sh	running	0.	14.	S0	00000000	0.
011D2000	hello	current	0.	17.	S0	00000000	0.

B::TASK.PS pid ppid uid tty flags time nice count state cmd										
pid	ppid	uid	tty	flags	time	nice	count	state	command	
0.	0.	0.	-	0100	00001A3C	20.	-100.	running	[swapper]	
1.	0.	0.	-	0100	00000000	0.	9.	sleeping	init	
2.	1.	0.	-	0040	00000000	0.	9.	sleeping	[keventd]	
3.	0.	0.	-	0040	00000000	19.	1.	sleeping	[ksoftirqd_CPU0]	
4.	0.	0.	-	0840	00000000	0.	9.	sleeping	[kswapd]	
5.	0.	0.	-	0040	00000000	0.	7.	sleeping	[bdfush]	
6.	0.	0.	-	0040	00000000	0.	7.	sleeping	[kupdated]	
14.	1.	0.	ttys0	0000	00000000	0.	4.	running	/bin/sh	
17.	14.	0.	ttys0	0000	00000000	0.	5.	running	hello	

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

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.

For general informations about the In-Circuit Debugger refer to the ["ICD Debugger User's Guide"](#) (debugger_user.pdf), ["ICE User's Guide"](#) (ice_user.pdf) or ["ICE User's Guide"](#) (ice_user.pdf). All general commands are described in ["IDE Reference Guide"](#) (ide_ref.pdf) and ["General Commands and Functions"](#).

Currently Linux is supported for the following versions:

- Linux kernel versions 2.4 and 2.6 on ARM, ColdFire, MIPS, PowerPC, SH4, and XScale

Configuration

The **TASK.CONFIG** command loads an extension definition file called "linux.t32" (directory "demo/<processor>/kernel/linux"). 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 RTOS debugger is used.

If a system symbol is not available or if another address should be used for a specific system variable then the corresponding argument must be set manually with the appropriate address. In this case, use the manual configuration, which can require some additional arguments.

If you want do use the display functions "On The Fly", i.e. displaying the OS objects, while the target is running, you need to have access to memory while running. In case of a ICE or FIRE, you have to map emulation or shadow memory to the address space of all used system tables. In case of ICD, you have to enable **SYSystem.MemAccess** or **SYSystem.CpuAccess** (CPU dependent).

Manual Configuration

Manual configuration is not recommended for the Linux RTOS debugger. If, for some reason, you want to use different symbols than those used by the automatic configuration, call LAUTERBACH for support.

See **Hooks & Internals** for details.

Automatic Configuration

For system resource display and trace functionality, you can do an automatic configuration of the RTOS debugger. For this purpose it is necessary, that all system internal symbols are loaded and accessible at any time, the RTOS debugger 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 full automatic configuration omit all arguments:

Format:	TASK.CONFIG linux
---------	--------------------------

If a system symbol is not available, or if another address should be used for a specific system variable, then the corresponding argument must be set manually with the appropriate address (see **Manual Configuration**).

Note that the default Linux kernel configuration 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".

To access all features of the RTOS debugger you should follow the following roadmap:

1. Read carefully the demo startup script (demo/<processor>/kernel/linux/linux.cmm).
2. Make a copy of the “linux.cmm” batchfile. 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 read carefully the previous Configuration chapters.

Hooks & Internals in Linux

No hooks are used in the kernel.

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

The RTOS debugger for Linux supports the following features.

Display of Kernel Resources

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

- | | |
|-------------------------|--------------------|
| - tasks | TASK.DTask |
| - “ps” outputs | TASK.PS |
| - kernel modules | TASK.MODULE |
| - file system internals | TASK.FS |

For a detailed description of each command refer to the chapter “**Linux Commands**”.

If your hardware allows accessing the memory, 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:

Format: **Break.Set** <address>|<range> [</option>] **/TASK** <task>

Use a task magic, id or name for <task>.

This task related breakpoint is implemented by a conditional breakpoint inside the debugger. I.e., 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.

Please note, that this feature impacts the real-time behavior of the application.

For a general description of the **Break.Set** command please see it's documentation.

Task Context Display

You are able to switch the whole viewing context to a currently not executing task. This means, that all register and stack related information (such as [Register](#), [Data.List](#), [Var.Frame](#) etc.) will be shown according to this task. Be aware, that this is only for displaying information. When continuing debugging the application (step or go), the debugger will switch back to the current context.

For displaying a specific task context, use the command:

Format: **Register.TASK** [*<task>*]

Specify a task magic or a task name (in double quotes) as parameter.
To switch back to the current context, omit all parameters.

For displaying the call stack of a specific task, you can use the following command:

Format: **Var.Frame /Task** *<task>*

If you'd like to see the application code, where the task was preempted, execute the "Var.Frame /Caller /Task *<task>*" window. Double click on the line showing the OS service call.

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

Space IDs

Processes of Linux may reside virtually on the same address. To distinguish those addresses, the Debugger uses an additional space id, that specifies, to which virtual memory space the address refers. The command **SYStem.Option MMUSPACES ON** enables the additional space id. For all processes using the kernel address space and for the kernel itself, the space id is zero. For processes using their own address space, the space id equals to the process id. Threads of a particular process share the memory space of the invoking process. Thus, the space id of threads equals to the space id of the parent process.

You may scan the whole system for space ids, using the command **MMU.SCANID**. Use **MMU.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
```

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

MMU.FORMAT declares the basic format of the MMU tables:

Format:	MMU. FORMAT <i><format></i> <i><base address></i> <i><kernel translation></i>
---------	--

Use as *<format>*:

LINUX	standard format
LINUX26	PowerPC with physical table pointers (some 2.6 variants)
LINUXEXT	PowerPC with 64bit table entries
LINUX32	MIPS32
LINUXBIG	MIPS32 with 64bit table entries
LINUX64	MIPS64 with 64bit table pointers and entries

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

<kernel translation> specifies the virtual 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.

When declaring the MMU layout, you should also create the kernel translation manually with **MMU.Create**.

The kernel code, which resides in the kernel space, can be accessed by any process, regardless of the current space id. Use the command **MMU.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 **MMU.ON**.

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

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

Please see also the sample scripts in the demo directory.

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

The command **MMU.SCAN only** scans the contents of the current processor MMU settings. Use the command **MMU.SCANALL** to go through all space ids and scan their MMU settings. Note, that on some systems, this may take a long time. In this case you may scan single processes.

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

MMU.List shows the debugger's address translation table for all scanned space ids.

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

See also chapter "**Debugging Linux Kernel and User Processes**"

The RTOS Debugger for Linux contains an “Autoloader”, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding Linux components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader, the debugger invokes the according command. The command is usually a call to a PRACTICE script, that handles loading the symbol file to the appropriate addresses.

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

The autoloader can be configured to react only on processes, kernel modules, (all) libraries, or libraries of the current process (see also **TASK.sYmbol.Option AutoLoad**). It is recommended to set only those components, that you are interested in, because this decreases the time of the autoloader checks highly.

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

Format:	sYmbol.AutoLoad.CHECKLINUX “<action>”
<action>	action to take for symbol load, e.g. “do autoloader”

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

The point of time, at which the component information is retrieved from the target, can be set:

Format:	sYmbol.AutoLoad.CHECK [ON OFF]
---------	---

A single “sYmbol.AutoLoad.CHECK” command manually triggers the refreshing of the target’s information.

If “sYmbol.AutoLoad.CHECK ON” is set, the debugger automatically reads the information on every go/halt or step cycle. This significantly slows down the debugger’s speed. If “sYmbol.AutoLoad.CHECK OFF” is set, no automatic read will be done, you have to manually trigger the information read when necessary.

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

The RTOS Debugger for Symbian supports symmetric multiprocessing (SMP).

An SMP system consists of several equal CPU cores. The operating system distributes the ready threads to all available cores, so that several threads are running in parallel. This means, that an application may run on any available core. Moreover, the core, at which the application runs, may change in time.

To support such SMP systems, the debugger allows a “system view”, where one PowerView UI is used for the whole system, i.e. for all cores that are used by the SMP OS. Please see the target guides how to set up the debugger with SMP support

All core relevant windows (e.g. [Register](#)) show the information of the actual core. The status line of the debugger indicates the actual core. You can switch the core view with the [CORE](#) command.

Target breaks, be it manual breaks or halting at a breakpoint, halt all cores synchronously. Similar, a “Go” 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; i.e. the breakpoint will always hit, regardless which core addresses it.

In SMP systems, the [TASK.TASK](#) 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 may execute a dynamic performance measurement by evaluating the current running task in fixed time intervals. Start the measurement with the command [PERF.Mode TASK](#), and view the contents with [PERF.List](#). 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.

For a general description of the [PERF](#) command refer to “[General Commands Reference Guide P](#)” (general_ref_p.pdf).

NOTE: This feature is **only** available, if your debugger equipment is able to trace task switches (see below).

Out of the recordings done by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically. To do this, the debugger must be able to detect the task switches out of the trace, i.e. the task switches need to be recorded.

Usually, there's a variable in the system that holds the current running task. By recording the accesses to this memory location (aka "magic" location), the debugger detects a task switch by a change of this variable. Please note, that the debugger equipment must be able to trace memory data accesses in this case, program flow trace is not sufficient.

If a hardware trace solution is available, that provide a so called "context id" (e.g. ARM11 ETM), and if this context id is served by the operating system, it is sufficient to enable the recording of the context id (owner cycle); no data trace is needed.

To do a selective recording on task switches with state analyzers (ICE and FIRE), use the following PRACTICE commands:

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

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

To do a selective recording on task switches with flow traces (ICD, e.g. ETM and NEXUS trace), based on the data accesses, use the following PRACTICE command:

```
; Enable tracing only on the magic location
Break.Set task.config(magic) /TraceEnable
```

To do a selective recording on task switches with flow traces, based on the context id, use the following PRACTICE command:

```
; Enable tracing of the context id (e.g. 32bit)
ETM.ContextID 32
```


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.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 “(root)”.

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

Some Linux distributions/versions support the context id. In most cases, the context id is only served with the address space id of the process. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide performance information of threads.

To allow a detailed performance analysis on Linux threads, the context id must contain the Linux pid of the thread (each thread gets its own pid - despite the name). Set the lower 8 bit of the context id register with the Linux own ASID, and set the upper 24 bit with the pid of the thread, i.e. “(pid << 8) | ASID”.

The Linux awareness needs to be informed about the changed format of the context id:

TASK.Option THRCTX ON

Example for an ARM926:

Include an assembler routine to set the context id in arch/arm/mm/proc-arm926.S:

```
ENTRY(cpu_arm926_set_context_id)
    mcr    p15, 0, r0, c13, c0, 1    @ set context id register
    mov    pc, lr
```

Additionally, extend the `context_switch()` function in `kernel/sched.c` (additional lines are in bold):

```
unsigned int ctxid;
void cpu_arm926_set_context_id(unsigned int);

arch_enter_lazy_cpu_mode();

cpu_arm926_set_context_id(0);

if (!mm) {
    next->active_mm = oldmm;
    atomic_inc(&oldmm->mm_count);
    enter_lazy_tlb(oldmm, next);
    ctxid = next->pid << 8;
} else {
#ifdef CONFIG_CPU_HAS_ASID
    ctxid = (mm->context.id & (unsigned int) 0xFF) | (next->pid << 8);
#else
    ctxid = next->pid << 8;
#endif
    switch_mm(oldmm, mm, next);
}

cpu_arm926_set_context_id(ctxid);
```

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). E.g. the state analyzers of ICE or FIRE are capable of doing so. 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. The breakpoints to the task status words are set by the **TASK.TASKState** command.

“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 debugger equipment is able to trace memory data accesses (program flow trace is not sufficient).

All function related statistic and timechart 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, additionally to the function entries and exits, the task switches must be recorded.

To do a selective recording on task related function runtimes with state analyzers (ICE and FIRE), use the following PRACTICE commands:

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

; Mark the function entries/exits with Alpha/Beta breakpoints
Break.SetFunc

; Program the Analyzer to record function entries/exits and task switches
Analyzer.ReProgram
(
    Sample.Enable if AlphaBreak||BetaBreak
    Mark.A        if AlphaBreak
    Mark.B        if BetaBreak
)
```

To do a selective recording on task related function runtimes with flow traces (ICD, e.g. ETM and NEXUS trace), use the following PRACTICE command:

```
; Enable tracing only on the magic location
Break.Set task.config(magic) /TraceData
```

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

Trace.List List.TASK FUNC	Display function nesting
Trace.STATistic.TASKFunc	Display function runtime statistic
Trace.STATistic.TASKTREE	Display functions as call tree
Trace.Chart.TASKFunc	Display function timechart

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

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

Linux Specific Menu

The file "linux.men" contains an alternate menu with Linux specific topics. Load this menu with the **MENU.ReProgram** command.

You will find a new pull-down menu called "**Linux**".

The “**Display**” topics 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 resp. 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 Process on main...**” allows you to start debugging a process on its main() function. Select this prior to starting the process. Specify the name of the process you want to debug. Then start the process in your Linux terminal. The debugger will load the symbols and halt at main(). 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.
- “**Scan Process MMU Pages...**” scans the MMU pages of the specified process.
“**Scan All MMU Tables**” performs a scan over all target side kernel and process MMU pages.
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 resp. 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 its 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”.
- “**Scan All MMU Tables**” performs a scan over all target side kernel and process MMU pages.
See also chapter “[Scanning System and Processes](#)”.

“**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 resp. 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](#).
- “**Scan Process MMU Pages...**” scans the MMU pages of the specified process. Specify the name of the process that uses the library you want to debug.
“**Scan All MMU Tables**” performs a scan over all target side kernel and process MMU pages.
See also chapter “[Scanning System and Processes](#)”.

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.
- **“Set Loader Script...”** allows you to specify the script that is called when a symbol file load is required. You may also set the automatic autoloader check.
- Use **“Set Components Checked...”** to specify, which Linux components should be managed by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.

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

The **“Trace”** pull-down menu is extended. In the **“List”** submenu, you can choose for an trace list window showing only task switches (if any) or task switches together with default display.

The **“Perf”** menu contains additional submenus for task runtime statistics, task related function runtime statistics or statistics on task states, if a trace is available. See also chapter **“Task Runtime Statistics”**.

This chapter moved into the [“Training RTOS Debugger for Linux - Stop Mode”](#) in , page 1 (training_rtos_linux.pdf).

We strongly recommend to read this training manual.

Debugging Linux Kernel and User Processes

Linux runs on virtual address spaces. The kernel uses a static address translation, usually starting from virtual address 0xc0000000 mapped to the physical start address of the RAM. Each user process gets its own user address space when loaded, usually starting from virtual 0x0, mapped to any physical RAM area, that is currently free. Due to this address translations, debugging the Linux kernel and the user processes requires some settings to the Debugger.

To distinguish those different memory mappings, TRACE32 uses “space ids”, defining individual address translations for each id. The kernel itself is attached to the space id zero. Each process, that has an own memory space, gets a space id, that is equal to its process id. Linux threads get the space id of the parent process.

See also chapter [“MMU Support”](#).

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 start the linux kernel from Flash, or if you download the kernel via Ethernet, do this as you are doing it without debugging.

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 work around.

Note, that at every time the Linux awareness is used, it needs the kernel symbols. Please see the chapters above, 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 an 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 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](#)”).

If `MMU.TableWalk` is OFF, you need to scan the MMU translation of this process:

```
MMU.TaskPageTable.SCAN 12.:0 ; scan MMU of process id 12.
```

See also chapter “[Scanning System and Processes](#)”.

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:

If you want to debug your process right from the beginning (at “main()”), you have to load the symbols *before* starting the process. This is a tricky thing, because you have to know the process id, which is assigned first at the process startup. Set a breakpoint into the process start handler of Linux, when the process is already loaded but not yet started. The function `set_binfmt()` may serve as a good point. When the breakpoint is hit, check if the process is already loaded. If so, extract the process id, and load the symbols. Don’t scan the process MMU now, because it’s not yet set up. Linux loads the code first, if it is accessed by the CPU. So you’re not able to set a software breakpoint yet into the process, because it will be overwritten by the swapper, when it loads actually the code. Instead, set an on-chip breakpoint to the `main()` routine of the process. As soon as the process is started, the code will be loaded and the breakpoint will be hit. Now you’re able to scan the process MMU and set software breakpoints. See the script “`app_debug.cmm`” in the demo directory, how to do this.

The “Linux” menu contains this procedure in a menu item: “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.

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:

Start your process and open a [TASK.DTask](#) window. Double click on the "magic" of the process using the library, and expand the "code files" tree (if available). A list will appear, that shows the loaded libraries and the corresponding load addresses. 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, 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"
```

Linux Threads are implemented as tasks, that share the same virtual memory. The Linux RTOS Debugger 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 it's 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 of a process, 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 4 KByte continuous memory region, 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” entrypoint, 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. Use MMU.SCANALL to rescan the MMU and the newly loaded page. A simple work around for functions is to set the breakpoint at the function entry plus 4 (e.g. "main+4"). Then the application will halt *after* the page was loaded.

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.

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. A subsequent `MMU.SCANALL` scans the translation table of the whole process. 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. A subsequent `MMU.SCANALL` scans the translation table of the whole process. 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 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. If MMU.TableWalk is OFF, you need to rescan the MMU translation with "MMU.SCANALL". See also chapter "[Scanning System and Processes](#)".

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 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:

In Linux kernel version 2.6, the kernel contains all section information, if the kernel is configured with `CONFIG_KALLSYMS=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.

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

If the section information is available, 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 ".ko" !

&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.ko /nocode /noclear /gnu \
/reloc .text at &text /reloc .rodata at &rodata \
/reloc .data at &data /reloc .bss at &bss
```

TASK.sYmbol.LOADMod will use the standard sections 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`” in 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 “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.

Format: TASK.DTask [<task>]

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

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

B::TASK.DTask							
magic	command	state	uid	pid	tty	flags	nice
01072000	swapper	running	0.	0.	0	00000100	20.
010FE000	init	sleeping	0.	1.	0	00000100	0.
010FC000	keventd	sleeping	0.	2.	0	00000040	0.
010F8000	ksoftirqd_CPU0	sleeping	0.	3.	0	00000040	19.
010F6000	kswapd	sleeping	0.	4.	0	00000840	0.
010F4000	bdflush	sleeping	0.	5.	0	00000040	0.
010F2000	kupdated	sleeping	0.	6.	0	00000040	0.
010C6000	sh	running	0.	14.	S0	00000000	0.
011D2000	hello	current	0.	17.	S0	00000000	0.

B::TASK.DTask "sh"							
magic	command	state	uid	pid	tty	flags	nice
010C6000	sh	running	0.	14.	S0	00000000	0.
gid	sigpending	vm size	ttb	tty name	path		
0.	00000000	00000000	-	ttyS0	/bin/sh		
+' flags							
-" parent		youngest child	younger sibling	older sibling			
init		hello	-	keventd			
-" arguments							
/bin/sh							
-" environment							
TERM=linux							
PATH=/bin:/usr/bin:/etc:/sbin:/usr/sbin							
+' open files							
-" code addr/size		data addr/size	stack start/end				
011F0040 / 0000BB00		011FBB44 / 000015E0	011FFF98 / 011FEFA9				
+' times							

“magic” is an unique id, used by the RTOS Debugger 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 of a process, declare the threading method manually with the TASK.Option Threading command.

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**

Displays the process table of Linux.

The display is similar to the output of the “ps” shell command.

B::TASK.PS pid ppid uid tty flags time nice count stat cmd										
pid	ppid	uid	tty	flags	time	nice	count	state	command	
0.	0.	0.	-	0100	00001A3C	20.	-100.	running	[swapper]	
1.	0.	0.	-	0100	00000000	0.	9.	sleeping	init	
2.	1.	0.	-	0040	00000000	0.	9.	sleeping	[keventd]	
3.	0.	0.	-	0040	00000000	19.	1.	sleeping	[ksoftirqd_CPU0]	
4.	0.	0.	-	0840	00000000	0.	9.	sleeping	[kswapd]	
5.	0.	0.	-	0040	00000000	0.	7.	sleeping	[bdflush]	
6.	0.	0.	-	0040	00000000	0.	7.	sleeping	[kupdated]	
14.	1.	0.	ttys0	0000	00000000	0.	4.	running	/bin/sh	
17.	14.	0.	ttys0	0000	00000000	0.	5.	running	hello	

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”.

B::TASK.MODUle						
magic	name	size	code addr	data addr	used by	
C685D000	pcmcia_core	37984.	C685D060	C686597C	0.	(unused)
C017B284		0.	C017B2E4	C0177F90	1.	(unused)

“magic” is an unique id, used by the RTOS Debugger to identify a module (address of the module struct).

“code addr” and “data addr” specify the address of the .text segment resp. 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.

Format: **TASK.FS.<Types | Mount | MountDevs | PROC>**

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

TASK.FS.Types

Display file system types

Format: **TASK.FS.Types**

This command displays the all file system types that are currently registered in the Linux kernel.

TASK.FS.Mount

Display mount points

Format: **TASK.FS.Mount**

This command displays the current mount points.

TASK.FS.MountDevs

Display mounted devices

Format: **TASK.FS.MountDevs**

This command displays all currently mounted devices (i.e. super blocks).

TASK.FS.PROC

Display /proc file system

Format: **TASK.FS.PROC**

This command displays the contents of the “/proc” file system (procfs), even if it is not mounted.

Format: TASK.TASKState

This command sets Alpha breakpoints on all tasks status words.

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

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

TASK.MMU.SCAN

Scan process MMU space

Format: TASK.MMU.SCAN [<process>]
--

If a process magic, id or name is specified, this command scans the target MMU of the space id, specified by the given process, and sets the Debugger MMU appropriately, to cover the physical to logical address conversion of this specific process.

If no argument is specified, the command scans all current processes.

Space IDs must be enabled ([SYStem.Option MMUSPACES ON](#)) to successfully execute this command.

The command walks through all page tables which are defined for the memory spaces of the process and prepares the Debugger MMU to hold the physical to logical address conversion of this process. This is needed to provide full HLL support. If a process was loaded dynamically, you must set the Debugger MMU to this process, otherwise the Debugger won't know, where the physical image of the process is placed.

See also [MMU Support](#).

Example:

```
; scan the memory space of the process "hello"
TASK.MMU.SCAN "hello"
```

(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 & MMU
TASK.sYmbol.DELeTe	Unload Process Symbols & MMU
TASK.sYmbol.LOADMod	Load Module Symbols & MMU
TASK.sYmbol.DELeTeMod	Unload Module Symbols & MMU
TASK.sYmbol.LOADLib	Load Library Symbols
TASK.sYmbol.DELeTeLib	Unload Library Symbols
TASK.sYmbol.Option	Set symbol management options

See also

- TASK.sYmbol.DELeTe
- TASK.sYmbol.DELeTeMod
- TASK.sYmbol.LOAD
- TASK.sYmbol.LOADMod

TASK.sYmbol.LOAD

Load process symbols & MMU

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. that show up in the TASK.DTask window).

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

magic__	command_	state__	uid__	pid__	tty	flags__	nice__
80044000	hello	current	0.	24.	0	00000000	0.

the command:

TASK.sYmbol.LOAD "hello"

will execute internally the commands:

```
TASK.MMU.SCAN 24.  
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.

See also

■ [TASK.sYmbol](#)

TASK.sYmbol.DELeTe

Unload process symbols & MMU

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

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

When debugging of a process is finished, or if the process exited, you should remove loaded process symbols and MMU entries. Otherwise the remained entries may disturb further debugging. This command deletes the symbols of the specified process and deletes it's MMU entries.

Example:

when deleting the above loaded symbols with the command:

```
TASK.sYmbol.DELeTe "hello"
```

the debugger will execute internally the commands:

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

See also

■ [TASK.sYmbol](#)

TASK.sYmbol.LOADMod

Load module symbols & 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. that show up in the `TASK.MODUle` window).

Example:

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

See also chapter “[Debugging Kernel Modules](#)”

See also

■ [TASK.sYmbol](#)

TASK.sYmbol.DEleteMod

Unload module symbols & 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 remained entries may disturb further debugging. This command deletes the symbols of the specified module and deletes it's MMU entries.

Example:

```
TASK.sYmbol.DEleteMod "pcmcia_core"
```

See also chapter “[Debugging Kernel Modules](#)”

See also

■ [TASK.sYmbol](#)

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. 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](#)"

Format: **TASK.sYmbol.DELeTeLib** <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".

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

Example:

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

See also chapter "[Debugging Into Shared Libraries](#)"

Format:	TASK.sYmbol.Option <option>
<option>	LOADCMD <command> LOADMCMD <command> LOADLCMD <command> MMUSCAN <ON OFF> AutoLoad <option>

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
%v	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"
```


LOADCMD:

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 LOADCMD "D.LOAD.Elf %s 0x%x:0x%x /nocode /noclear"  
TASK.sYmbol.Option LOADMCMD "do myloadscript %s 0x%x 0x%x"
```

MMUSCAN:

This option controls, if the symbol loading mechanisms of TASK.sYmbol scan the MMU page tables of the loaded components, too. When using MMU.TableWalk, then switch this off.

AutoLoad:

This option controls, which components are checked and managed by the AutoLoader:

Process	check processes
Library	check all libraries of all processes
Module	check kernel modules
CurrLib	check only libraries of current process
ALL	check processes, libraries and kernel modules
NoProcess	don't check processes
NoLibrary	don't check libraries
NoModule	don't check modules
NONE	check nothing.

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

Example:

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

(Not available for all processors!)

The TASK.Watch command group build 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 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

See also

- | | | | |
|-------------------------------------|---------------------------------------|--------------------------------------|--|
| ■ TASK.Watch.ADD | ■ TASK.Watch.DELeTe | ■ TASK.Watch.DISable | ■ TASK.Watch.DISableBP |
| ■ TASK.Watch.ENABLE | ■ TASK.Watch.ENABLEBP | ■ TASK.Watch.View | |

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

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

NOTE: This feature may affect the realtime behavior of the target application!
(Please see below for details.)

The argument takes a process name for the initial process to watch.

B::TASK.Watch.View "ping"				
process	spaceid	state	entry	
ping	6.	loaded	main	
visadm	--	no process	--	
filesys	1.	no symbols	--	

process	Specifies the name of the process to watch for.
spaceid	Specifies the current space id (= process id) of the watched process. If greyed, the debugger is currently not able to determine the space id of the process (e.g. the target is running).
state	Indicates the current watch state of the process. If greyed, the debugger is currently not able to determine the watch state. no process: the debugger couldn't find the process in the actual Linux process list. no symbols: the debugger found the process and loaded it's MMU settings, but couldn't load the symbols of the process (most likely because the according symbol files were missing). loaded: the debugger found the process, loaded it's MMU settings and loaded it's symbols
entry	Specifies the process entry point, which is <code>main()</code> . If greyed, 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 <code>TASK.Watch.DISableBP</code>)

The watch system for processes is able to automatically load and unload the symbols of a process and it's 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 it's entry point.

Add processes to the watch list with the command **TASK.Watch.ADD**.
Remove processes from the watch list with the command **TASK.Watch.DELeTe**.
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 greyed (inactive).

If the target is halted (either by running on a breakpoint, or by halting it manually), the watch system starts it's 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 it's MMU entries and loads the appropriate symbol files. In fact, it executes **TASK.sYmbol.LOAD** for the new process (see there).

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 it's 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. 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. 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 those breakpoints are available when using this feature.

NOTE: Upon every process creation, the target application is halted for a short time and resumed after searching for the watched processes. **This impacts the realtime behavior of your target.**

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

See also

■ [TASK.Watch](#)

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

Specify the process name (in quotes) or magic to add this process to the watched processes list.

Please see [TASK.Watch.View](#) for details.

See also

■ [TASK.Watch](#)

TASK.Watch.DELeTe

Remove process from watch list

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

Specify the process name (in quotes) or magic to remove this process from the watched processes list.

Please see [TASK.Watch.View](#) for details.

See also

■ [TASK.Watch](#)

TASK.Watch.DISable

Disable watch system

Format: **TASK.Watch.DISable**

This command 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 greyed.

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

See also

■ [TASK.Watch](#)

Format:	TASK.Watch.DISableBP
---------	-----------------------------

This command prevents the debugger from setting on-chip breakpoints for the detection of process creation. After executing this command, the target will run in realtime. 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.

See also

■ [TASK.Watch](#)

TASK.Watch.ENABLE

Enable watch system

Format:	TASK.Watch.ENABLE
---------	--------------------------

This command 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.

See also

■ [TASK.Watch](#)

Format:

TASK.Watch.ENable

This command enables the previously disabled on-chip breakpoints for detection of process creation.

Please see [TASK.Watch.View](#) for details.

See also

- [TASK.Watch](#)

TASK.Watch.Option

Set watch system options

Format:

TASK.Watch.Option *<option>*

<option>:

BreakFuncC *<function>*

Set various options to the watch system.

- BreakFuncC

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

```

Format:          TASK.Option <option>

<option>:        Threading <threading> [ ON | OFF ]
                  MEMMAP <mem_map>
                  NameMode [ comm | TaskName | ARG0 ]
                  THRCTX [ ON | OFF ]

```

Set various options to the awareness.

Threading	<p>Set the Threading type used by Linux.</p> <p>TGROUP: threads are organized by the thread_group list.</p>
MEMMAP	<p>For ColdFire only:</p> <p>Set the MMU memory mapping table (default is symbol “mem_map”).</p> <p>If no mapping table is used within the MMU, specify “0”.</p>
NameMode	<p>Set the mode how the task names are evaluated.</p> <p>comm: use the “comm” field in the task structure (default).</p> <p>TaskName: use the name evaluated by TASK.NAME and comm (allows renaming of the tasks with TASK.NAME.Set).</p> <p>ARG0: use the arg[0] statement of the process call.</p>
THRCTX	<p>Set the context id type, that is recorded with the real-time trace (e.g. ETM).</p> <p>If set to on, the context id in the trace contains thread switch detection.</p> <p>See Task Runtime Statistics.</p>

See also chapter “[Debugging Threads](#)”.

Linux PRACTICE Functions

There are special definitions for Linux specific PRACTICE functions.

TASK.CONFIG(<item>)	Reports configuration parameters
TASK.CONFIG(magic)	Returns the address for the magic number
TASK.CONFIG(magicsize)	Returns the size of the magic number (1,2 or 4)
TASK.CURRENT(spaceid)	Returns the current space id.
TASK.ERROR.CODE()	Checks for Awareness errors and returns the error code (0 = no error)
TASK.ERROR.HELP()	Checks for Awareness errors and returns the error help id.
TASK.LIB.ADDRESS (<i><libraryname></i> , <i><processmagic></i>)	Returns the load address of the library, loaded by the specified process
TASK.LIB.CODESIZE (<i><libraryname></i> , <i><processmagic></i>)	Returns the code size of the library, loaded by the specified process
TASK.MOD.CODEADDR(<modulename>)	Returns the code start address of the module
TASK.MOD.DATAADDR(<modulename>)	Returns the data start address of the module
TASK.MOD.MAGIC(<modulename>)	Returns the “magic” value of the module
TASK.MOD.NAME(<modulemagic>)	Returns the name of the given module magic
TASK.MOD.SECTION (<i><sectionname></i> , <i><modulemagic></i>)	Returns the address of the section of the specified module
TASK.PROC.CODEADDR(<processname>)	Returns the code start address of the process
TASK.PROC.CODESIZE(<processname>)	Returns the code size of the process

TASK.PROC.DATAADDR (<i><processname></i>)	Returns the data start address of the process
TASK.PROC.DATASIZE (<i><processname></i>)	Returns the data size of the process
TASK.PROC.MAGIC (<i><processname></i>)	Returns the “magic” value of the process
TASK.PROC.NAME (<i><processmagic></i>)	Returns the name of the specified process
TASK.PROC.PSID (<i><processname></i>)	Returns the process id of the specified process
TASK.PROC.SPACEID (<i><processname></i>)	Returns the space id of the specified process
TASK.PROC.VMAEND (<i><processname></i> , <i><address></i>)	Returns the end address of the given process virtual memory area holding the given address
TASK.PROC.VMASTART (<i><processname></i> , <i><address></i>)	Returns the start address of the given process virtual memory area holding the given address

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 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;
}

```