LAUTERBACH
DEVELOPMENT TOOLS

Training Linux Debugging for Intel® x86/x64

Release 09.2023

MANUAL

Training Linux Debugging for Intel® x86/x64

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Training ..... 🗁

Training Intel® x86/x64 ..... 🗁

Training Linux Debugging for Intel® x86/x64 ..... **1**

Introduction **5**

Documentation Updates 5

Related Documents and Tutorials 5

Basic Terms on Embedded Linux **6**

Linux Components 6

The Kernel 6

Kernel Modules 7

Processes and Threads 7

Libraries (Shared Objects) 7

The Linux Awareness 7

Virtual Memory Management in Linux 9

Virtual Address Map of Linux 9

Debugger Memory Access 10

On Demand Paging 13

Run-Mode vs. Stop-Mode Debugging 14

Hardware Based Debuggers 14

Software Based Debuggers 15

Kernel Configuration 16

Setting up a Script for Linux-Aware Debugging **18**

Linux Setup-Steps and -Commands 18

Set up the Debugger Address Translation 24

Mark the Kernel Address Space 27

Example Linux Setup-Scripts 28

Debugging the Linux Components **31**

The Kernel 31

Kernel Startup 31

Kernel Boot 33

Verifying Image and Symbols 34

Kernel Modules 35

Processes 37

Threads 39

Libraries 39
Task Related Breakpoints 41
Task Related Single Stepping 41
Task Context Display 42

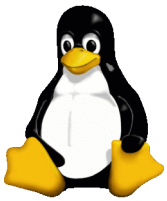
Linux specific Windows 43
Display of System Resources 43
Kernel Module List 44
File System Information 45
Kernel Log Buffer 46
RAM Dump Generation 47

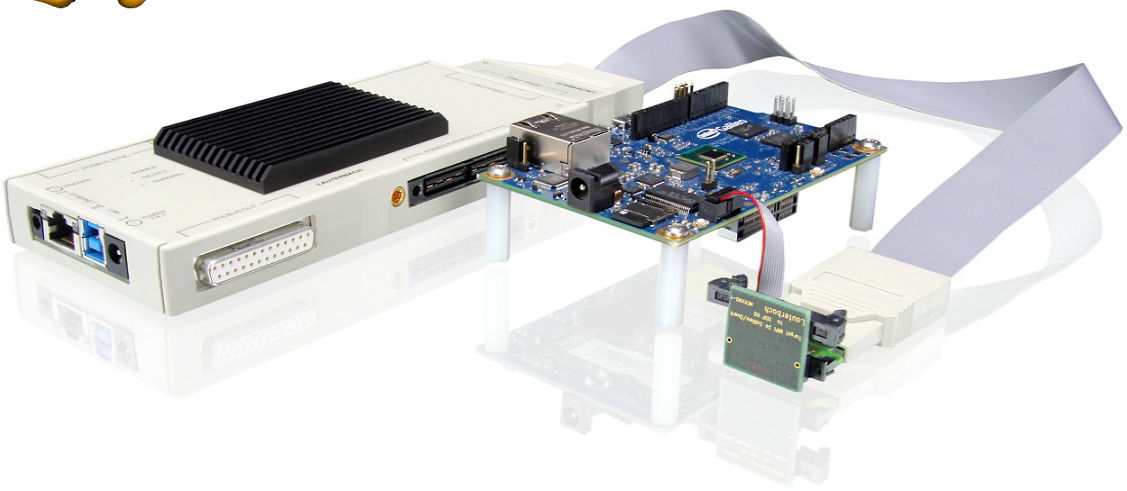
Troubleshooting 48

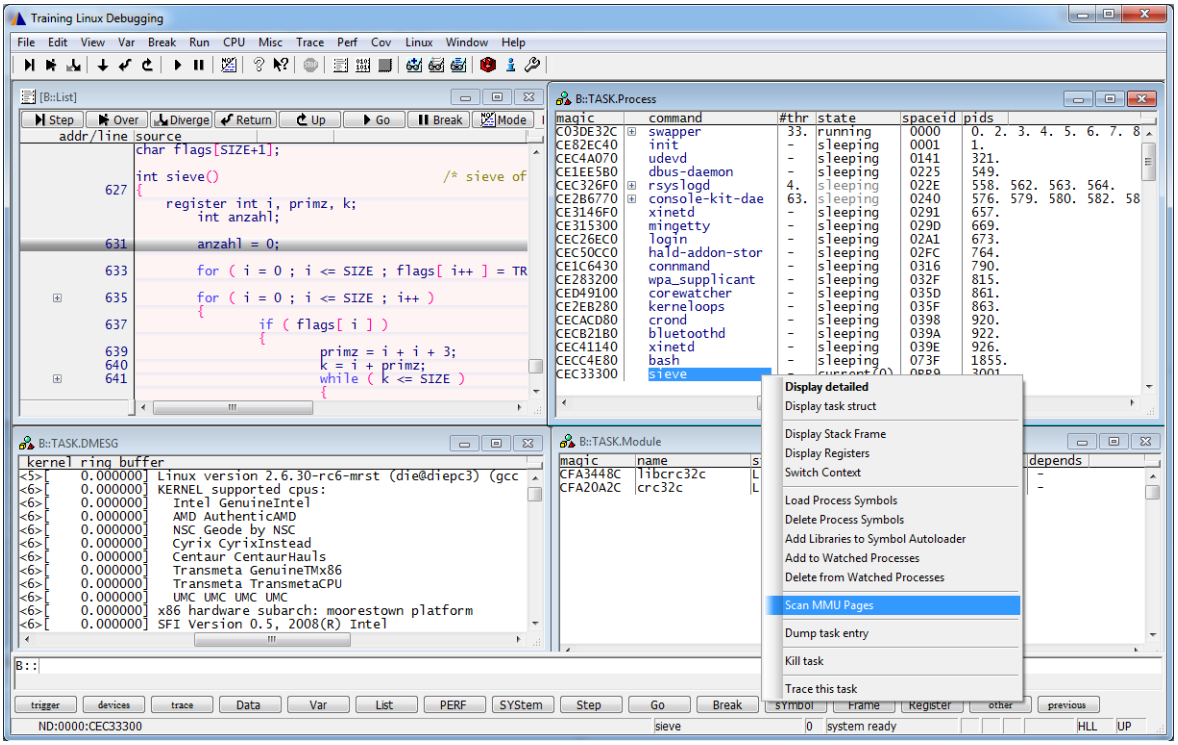
FAQ 49

Training Linux Debugging for Intel® x86/x64

Version 10-Oct-2023







Introduction

This training will have the main subjects:

- **Basic terms on embedded Linux**
- **Kernel configuration**
- **Setting up a script for Linux-aware debugging**
- **Debugging Linux components by TRACE32 Linux menu**
- **Troubleshooting**

Please note that this document only covers Linux debugging on Intel x86/x64. Please refer to **"Training Linux Debugging"** (training_rtos_linux.pdf) if you are using a different processor architecture.

Documentation Updates

The latest version of this document is available for download from:

www.lauterbach.com/pdf/training_rtos_linux_x86.pdf

Related Documents and Tutorials

- For a complete description of the Linux awareness commands, refer to the **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf).
- For information about Linux run mode debugging, please refer to **"Run Mode Debugging Manual Linux"** (rtos_linux_run.pdf) and **"TRACE32 as GDB Front-End"** (frontend_gdb.pdf).
- The **Linux Debugging Reference Card** includes an overview of frequently used TRACE32 commands for debugging targets running Linux.
- For a short video tutorial about Linux debugging, visit: **support.lauterbach.com/kb/articles/trace32-linux-debugging**

Basic Terms on Embedded Linux

This part describes essential basics and terms related to Linux and Linux-Debugging.

1. **Linux Components**
2. **The Linux Awareness**
3. **Virtual Memory Management in Linux**
4. **Run-Mode vs. Stop-Mode Debugging**

Linux Components

From the point of view of a debugger, a Linux system consists of the following components:

- The Linux kernel
- Kernel modules
- Processes and threads
- Libraries (shared objects)

Moreover, we can talk about two different spaces of executed code:

- Kernel space with privileged rights which includes the kernel
- User space with limited rights which includes processes, threads and libraries.

The kernel debug symbols (vmlinux) should be loaded in TRACE32 by the user. The debug symbols of kernel modules, processes and libraries are automatically loaded on-demand by the TRACE32 Symbol Autoloader. Please refer to the rest of this training, as well as to **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information.

The Kernel

The Linux kernel is the most important part in a Linux system. It runs in privileged kernel space and takes care of hardware initialization, device drivers, process scheduling, interrupts, memory management... The Linux kernel is generally contained in a statically linked executable in one of the object files supported by Linux (e.g. "vmlinux"). You can also find the kernel in compressed binary format (zImage/uImage). You will see later in this training how to configure the Linux kernel for Linux-aware debugging.

Kernel threads:

It is often useful for the kernel to perform operations in the background. The kernel accomplishes this via kernel threads. Kernel threads exist solely in kernel space. The significant difference between kernel threads and processes is that kernel threads operate in kernel space and do not have their own address space.

Kernel Modules

Kernel modules (*.ko) are software packages that are loaded and linked dynamically to the kernel at run time. They can be loaded and unloaded from the kernel within a user shell by the commands modeprobe/insmod and rmmod. Typically kernel modules contain code for device drivers, file systems, etc. Kernel modules run at kernel level with kernel privileges (supervisor).

Processes and Threads

A process is an application in the midst of execution. It also includes, additionally to executed code, a set of resources such as open files, pending signals, a memory address space with one or more memory mappings...

Linux processes are encapsulated by memory protection. Each process has its own virtual memory which can only be accessed by this process and the kernel. Processes run in user space with limited privileges.

A process could have one or more threads of execution. Each thread includes a unique program counter, process stack and set of process registers. To the Linux kernel, there is no concept of a thread. Linux implements all threads as standard processes. For Linux, a thread is a processes that shares certain resources with other processes.

Libraries (Shared Objects)

Libraries (shared objects, *.so) are commonly used software packages loaded and used by processes and linked to them at run-time. Libraries run in the memory space of the process that loaded them having the same limited privilege as the owning process. Same as processes, also libraries are always loaded and executed as a file through a file system.

The Linux Awareness

Debugging an operating system like Linux requires special support from the debugger. We say that the debugger needs to be **"aware"** of the operating system. Since TRACE32 supports a wide range of target operating systems, this special support is not statically linked in the debugger software but can be dynamically loaded as an extension depending on which operating system is used. Additional commands, options and displays will be then available and simplify the debugging of the operating system.The set of files providing these operating system debugging capabilities is called here **"awareness"**.

To be able to read the task list or to allow process or module debugging, the Linux awareness accesses the kernel's internal structures using the kernel symbols. **Thus the kernel symbols must always be available otherwise Linux aware debugging will not be possible**. The file vmlinux has to be compiled with debugging information enabled as will be shown later.

The Linux awareness files can be found in the TRACE32 installation directory under
`~~/demo/<arch>/kernel/linux/`

The Linux awareness can be loaded using the command **TASK.CONFIG** or **EXTension.LOAD**.

You can check the version of the loaded Linux awareness in the **VERSION.SOFTWARE** window. This information will only be shown if the Linux awareness is already loaded.

Virtual Memory Management in Linux

Before actually going into the details on how to debug a Linux system with TRACE32, we need to look at the helping features of TRACE32 that make Linux debugging possible.

Virtual Address Map of Linux

We start by discussing the virtual address map used by a running Linux system. Basically the memory is split into two sections: one section is reserved for the kernel and the second one for the user applications. The kernel runs in supervisor/privileged mode and has full access to the whole system while user processes run in user/non-privileged mode. The kernel has full visibility of the whole virtual address map, while the user processes have only a partial visibility. It's the task of the kernel to maintain the virtual address map and also the virtual to physical address translations for each user process.

The kernel space is exclusively used by the kernel, this means that a kernel logical/virtual address can have, at a given time, one single virtual-to-physical address mapping. On the other hand, the user space is shared by all running processes. Thus a virtual address in the user space can have different mappings depending on the process to which this address belongs.

The kernel space includes the kernel logical address range which is mapped to a continuous block in the physical memory. The kernel logical addresses and their associated physical addresses differ only by a constant offset. We denote this kernel logical to physical address translation as "***kernel default translation***". The rest of the kernel space includes the kernel virtual addresses which do not have necessarily the same mapping as the kernel default translation. This includes for instance kernel modules and memory allocated with vmalloc.

For a 32 bit Linux, the logical start address of the kernel is fixed by the kernel `CONFIG_PAGE_OFFSET` macro which is per default `0xC0000000` and the end address is the value of the `high_memory` variable minus one.

The virtual memory map for a 64 bit Linux kernel is described in the kernel documentation under `Documentation/x86/x86_64/mm.txt`.

Per default (i.e. with disabled debugger address translation) the debugger accesses the memory virtually (through the core). This way, it is only possible to access memory pages which are currently mapped in the translation look-aside buffers (TLB).

Alternatively, you can set up the debugger to access the memory physically. This way, the debugger will have access to all the existing physical memory. However, Linux operates completely in virtual memory space: all functions, variables, pointers etc. work with virtual addresses. Also, the symbols are bound to virtual addresses. Hence, if the user tries to read the value of a variable for instance, the debugger has to find the virtual to physical address translation for this variable and access it using its physical address.

The debugger can hold a local translation list. Translations can be added to this list manually using the **TRANSlation.Create** command. This local translation list can be viewed using the **TRANSlation.List** command. If the accessed virtual address has a translation in the local translation list then this translation is used, otherwise if the translation **"table walk"** is enabled (**TRANSlation.TableWalk ON**) then the debugger will read the target MMU page table(s) to find the virtual to physical address translation. We call this process "***debugger table walk***".

NOTE:	The debugger local translation list has the highest priority in the debugger translation process.

In contrast to the CPU address translation, if the virtual to physical address mapping is not found in the page table when performing a debugger table walk, no page fault is generated. It is then not possible for the debugger to access this address. A debugger memory access doesn't modify the MMU page tables.

Without further settings, the debugger can only access the current page table pointed by the CR3 register. However, each process as well as the kernel, has its own page table. Hence, by walking only through the current page table, it is not possible to find the virtual to physical address mapping of a process which is not the current executing one and as follows it is not possible to access the memory of such a process.

But since the Linux kernel manages the switching of the MMU for all processes, kernel structures hold the pointers for the translation pages tables for every process. The debugger just needs to get this information from the kernel data structures to be able to access the memory for any running task in the system. It is the task of the Linux awareness to get the page table descriptors for all running tasks on the system. You can display these descriptors by execution the TRACE32 commands **TRANSlation.ScanID** and **TRANSlation.ListID**.

To be able to access the kernel logical range at any time, the debugger needs to know the kernel logical to physical address translation.

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

If you enter commands with a virtual address without the TRACE32 space ID, the debugger will access the virtual address space of the current running task.

The following command enables the use of space IDs in TRACE32:

SYStem.Option.MMUSPACES ON

SYStem.Option.MMUSPACES ON doesn't switch on the processor MMU. It just **extends the addresses with space IDs.**

After enabling the address extension with the memory space IDs, a virtual address looks like "`001E:10001244`", which means virtual address `0x10001244` with space ID `0x1E` (pid = 30.).

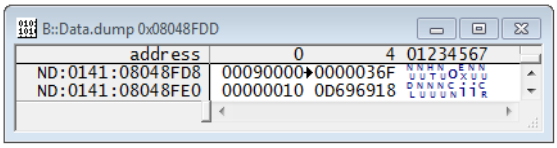
You can now access the complete memory:

```
Data.dump 0x10002480          ; Will show the memory at virtual address
                              ; 0x10002480 of the current running task

List 0x2F:0x10003000          ; Will show a code window at the address
                              ; 0x10003000 of the process having the space
                              ; id 0x2F

Data.dump A:0x10002000        ; Will show the memory at physical address
                              ; 0x10002000
```

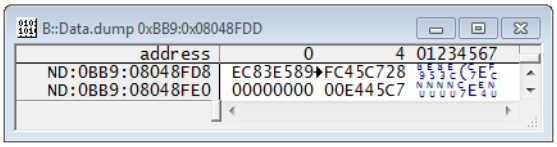
Symbols are always bound to a specific space ID. When loading the symbols, you need to specify, to which space ID they should belong. If you load the symbols without specifying the space ID, they will be bound to space ID zero (i.e. the kernel's space ID). See chapter **"Debugging the Linux Components"**, page 31 for details.

Because the symbols already contain the information of the space ID, you don't have to specify it manually.

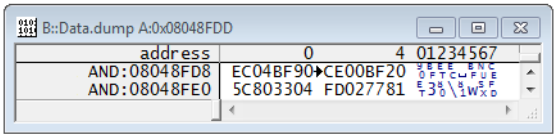
```
Data.dump myVariable          ; Will show the memory at the virtual
                              ; address of "myVariable" with the space ID
                              ; of the process holding this variable
```



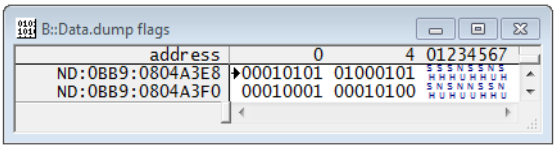
virtual address of current process 0x141



virtual address of specified process 0xBB9



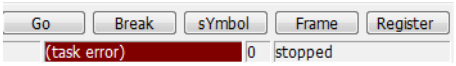
access to physical address A:0x8048FDD



Symbol "flags" with process 0xBB9

NOTE:	Address extension with the memory space IDs is per default disabled in TRACE32. The command **SYStem.Option.MMUSPACES ON** has thus to be included at the start of the Linux debugging script.

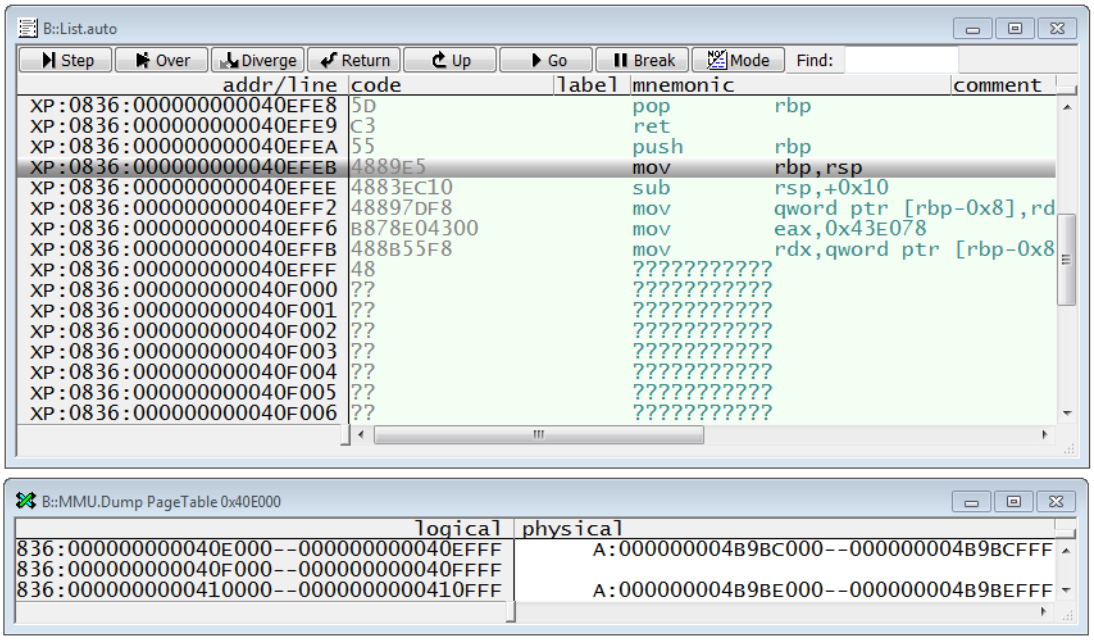
If the Linux awareness is enabled, the debugger tries to get the space ID of the current process by accessing the kernel's internal data structures. If this fails e.g. because of wrong symbol information, an access error, or simply because the kernel's data structures have not been yet initialized (in case you stop the target early in the kernel boot process), the debugger sets the current space ID to `0xFFFF` and shows the message "task error" in the status line.



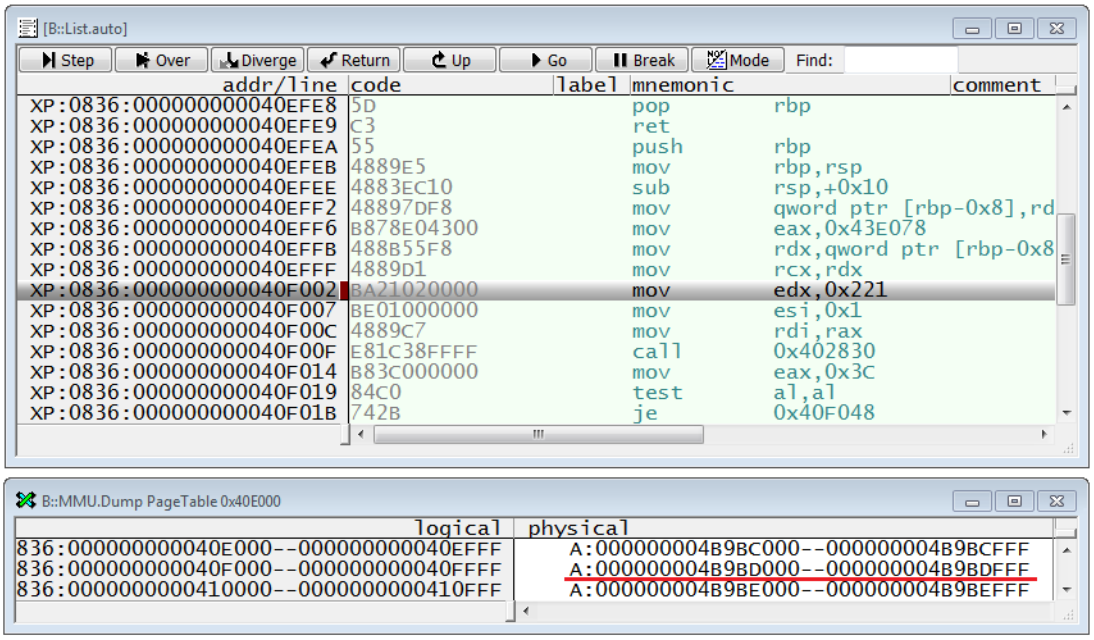
You can ignore the "task error" message as long as the kernel has not yet booted. In case you still get this error after the kernel boot, then you probably have a wrong configuration or a problem with the kernel debug symbols.

Linux is designed for heavy MMU usage with on-demand paging. On-demand paging means that code and data pages are loaded when they are first accessed. If the process tries to access a memory page that is not yet loaded, it creates a page fault. The page fault handler then loads the appropriate page.

The following screen shots show an example of on-demand paging. The instruction pointer is near the page boundary at the address `0x40EFFB`. The next memory page beginning at `0x40F000` cannot be accessed by the debugger since it doesn't have a mapping in the MMU page table of the current process.



We set an on-chip breakpoint somewhere in the next memory page and resume the execution. A page fault then occurs and the memory page is loaded and gets a mapping in the current page table.



Run-Mode vs. Stop-Mode Debugging

There are two main alternatives for debugging a Linux target: hardware based (stop mode) and software based (run mode). This chapter gives a small introduction regarding the differences between stop and run mode debugging which are both supported by TRACE32.

Hardware Based Debuggers

A hardware-based debugger uses special hardware to access target, processor and memory (e.g. by using the JTAG interface). No software components are required on the target for debugging. This allows debugging of bootstraps (right from the reset vector), interrupts, and any other software. Even if the target application runs into a complete system crash, you are still able to access the memory contents (post mortem debugging).

A breakpoint is handled by hardware, too. If it is reached, the whole target system (i.e. the processor) is stopped. Neither the kernel, nor other processes will continue. When resuming the target, it continues at the exact state, as it was halted at the breakpoint. This is very handy to debug interrupts or communications. However, keep in mind that also “keep alive” routines may be stopped (e.g. watchdog handlers).

The debugger is able to access the memory physically over the complete address range, without any restrictions. All software parts residing in physical memory are visible, even if they are not currently mapped by the TLBs. If the debugger knows the address translation of all processes, you gain access to any process data at any time.

The “on demand paging” mechanism used by Linux implies that pages of the application may be physically not present in the memory. The debugger cannot access such pages (including software breakpoints), as long as they are not loaded.

Advantages:

- **bootstrap, interrupt or post mortem debugging is possible**
- **no software restrictions (like memory protection, ...) apply to the debugger**
- **the full MMU table and code of all processes alive can be made visible**
- **only JTAG is required, no special communication interface as RS232 or Ethernet is needed**

Disadvantages:

- **halts the complete CPU, not only the desired process**
- **synchronization and communications to peripherals usually get lost**
- **debug hardware and a debug interface on the target are needed**

Software based debuggers, e.g. GDB, usually use a standard interface to the target, e.g. serial line or Ethernet. There is a small program code on the target (called “stub” or “agent”) that waits for debugging requests on the desired interface line and executes the appropriate actions. Of course, this part of the software must run, in order for the debugger to work correctly. This implies that the target must be up and running, and the driver for the interface line must be working. Hence, no bootstrap, interrupt or post mortem debugging is possible.

When using such a debugger to debug a process, a breakpoint halts only the desired process. The kernel and all other processes in the target continue to run. This may be helpful, if e.g. protocol stacks need to continue while debugging, but hinders the debugging of inter-process communication.

Because the debugging execution engine is part of the target program, all software restrictions apply to the debugger, too. In the case of a gdbserver for example, which is a user application, the debugger can only access the resources of the currently debugged processes. In this case, it is not possible to access the kernel or other processes.

Advantages:

- **halts only the desired process**
- **synchronization and communications to peripherals usually continue**
- **no debugger hardware and no JTAG interface are needed**

Disadvantages:

- **no bootstrap, interrupt or post mortem debugging is possible**
- **all software restrictions apply to the debugger too (memory protection, ...)**
- **only the current MMU and code of this scheduled process is visible**
- **actions from GDB change the state of the target (e.g page faults are triggered)**
- **one RS232 or Ethernet interface of the target is blocked**

The GDB Remote Serial Protocol (RSP) is used by some emulators/simulators (e.g. QEMU) as a debug protocol. In this case, the debug stub is part of the emulator itself. We talk this in this case about stop mode debugging.

Software based debugging is less robust and has many limitations in comparison to hardware based debugging. Thus, it is recommended to use JTAG based debugging if possible.

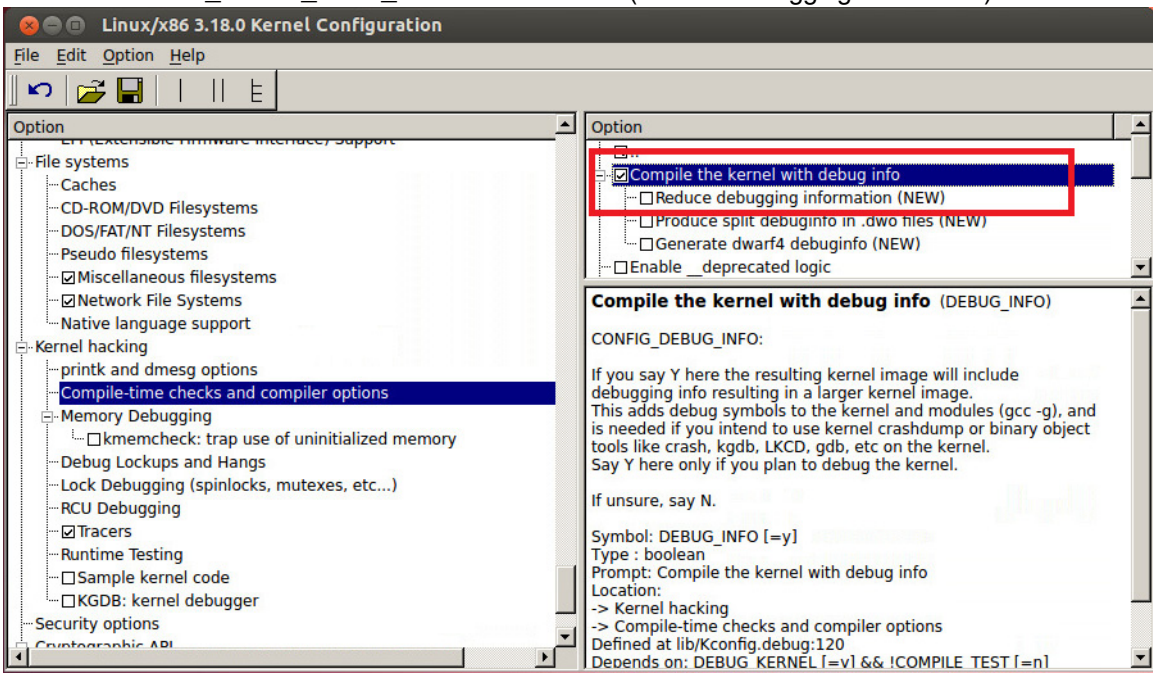
Run mode debugging is not covered by this training, for more information please refer to **“Run Mode Debugging Manual Linux”** (rtos_linux_run.pdf) and **“TRACE32 as GDB Front-End”** (frontend_gdb.pdf).

Kernel Configuration

Before going forward with writing Linux TRACE32 scripts and debugging the different Linux components, we will show the important kernel configurations that have influence on Linux debugging.

Compile The Kernel With Debug Info

To be able to do Linux aware debugging, the vmlinux file must be compiled with debug info enabled. Thus, you need to ensure that `CONFIG_DEBUG_INFO` is enabled in the kernel configuration. Please also make sure that `CONFIG_DEBUG_INFO_REDUCED` is **not** set (Reduce debugging information).



Moreover the option “Produce split debug info in .dwo files” (`CONFIG_DEBUG_INFO_SPLIT`) has to be disabled.

```
CONFIG_DEBUG_INFO=y
# CONFIG_DEBUG_INFO_REDUCED is not set
# CONFIG_DEBUG_INFO_SPLIT is not set
```

Disable Randomization

For some processor architectures, the Linux kernel offers a security feature which allows to randomize the virtual address at which the kernel image is loaded (`CONFIG_RANDOMIZE_BASE`). This option has to be disabled in the kernel configuration, otherwise the debug symbol addresses loaded from the `vmlinux` file do not match anymore the kernel code/data. As an alternative to disabling this option, you can add `"nokaslr"` to the kernel boot parameters.

Disable Lockup and Hang Detection

The Linux kernel provides the possibility to detect soft lockups and hung tasks by acting as a watchdog. This can be enabled under **Kernel hacking > Debug Lockups and Hangs**. The corresponding kernel configuration options are `CONFIG_SOFTLOCKUP_DETECTOR` and `CONFIG_DETECT_HUNG_TASK`.

If the program execution is stopped for a certain period of time, the soft lockup and hang detection could trigger a kernel panic. It is thus recommended to disable this detection in the kernel configuration.

CPU Power Management

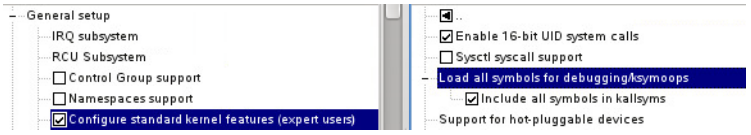
The Linux kernel CPU power management could cause for some processor architectures that single cores are not accessible by the debugger when in power saving state. CPU power management can be disabled in the Linux kernel configuration by disabling the options `CONFIG_CPU_IDLE` and `CONFIG_CPU_FREQ`.

Idle states can also be disabled for single cores from the shell by writing to the file `/sys/devices/system/cpu/cpu<x>/cpuidle/state<x>/disable`. Alternatively, you may remove the idle-states property from the device tree if available.

On some Linux distributions, power management can be disabled using specific kernel command line parameters (e.g. `"jtag=on"` or `"nohlt"`). Please refer to the documentation of the kernel command line parameters of your Linux distribution for more information.

Kernel Modules Related Configurations

The kernel contains all section information if it has been configured with `CONFIG_KALLSYMS=y`. When configuring the kernel, set the option **"General Setup"-> "Configure standard kernel features" -> "Load all symbols"** to yes. Without `KALLSYMS`, no section information is available and debugging kernel modules is not possible.



Extracting the Kernel Configuration

The Linux awareness includes a script (`getconfig.cmm`) that can be used in order to extract the kernel configuration file from a running Linux kernel. You just need to stop the program execution and call the script e.g.:

```
Break
DO ~~/demo/arm/kernel/linux/getconfig.cmm
```

The script will extract a config.gz file from the kernel. Please note that this script only works if `IKCONFIG_PROC` (enable access to .config through /proc/config.gz) is enabled in the kernel configuration.

Setting up a Script for Linux-Aware Debugging

This chapter will introduce the **typical steps** to prepare the TRACE32 debugger **for convenient Linux-Debugging**. Sample Linux debugging setup script files are presented at the end of this chapter.

Linux Setup-Steps and -Commands

To be able to do Linux aware debugging, some configuration needs to be done in TRACE32. The minimal setup includes the following steps:

- Connect to the target platform
- Load the Linux kernel symbols
- Set up the debugger address translation
- Load the Linux awareness and the Linux menu

These are the only needed configuration steps if you want to attach to a running Linux kernel. In case you want to debug the kernel boot, then you additionally need to make sure to stop the execution before the kernel start.

Moreover, it is possible to download the kernel image to the RAM using the debugger. We will discuss in this chapter which setup is needed in this case.

You can find Linux demo scripts in the TRACE32 installation directory under `~~/demo/x86/kernel/linux/board` and `~~/demo/x64/kernel/linux/board`.

Debugger Reset for Linux Debugging

Especially if you restart debugging during a debug session you are not sure about the state the debugger was in. It is thus recommended to use the command **RESet** in order to reset the debugger settings. .

```
RESet                           ; reset debugger completely
```

The RESet command doesn't reset the target but only the debugger environment.

Moreover, it is also good to clear all debugger windows before connecting to the target using the **WinCLEAR** command.

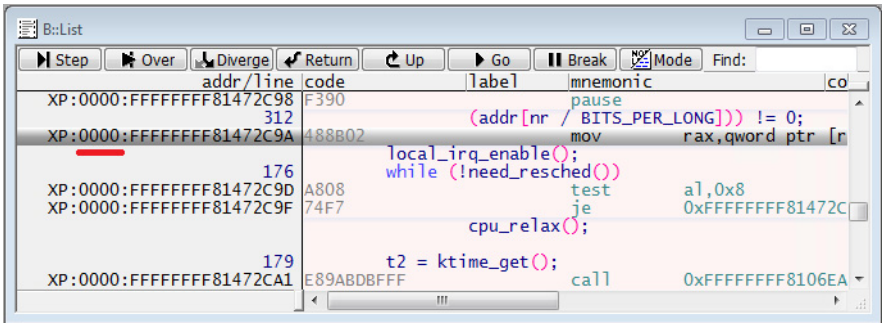
```
WinCLEAR                        ; clear all debugger windows
```

You need to set up the debugger to be able to connect to the target platform. The needed setup highly depends on the used target platform. Start-up scripts for different target platforms are available in the TRACE32 demo directory. You can use the TRACE32 menu **"File"** -> **"Search for Scripts.."** to find suitable demo scripts for your target board. Please also refer to **"Intel® x86/x64 Debugger"** (debugger_x86.pdf).

Additional settings related to OS-aware debugging are needed. These settings are presented below.

Address Extension

Switch on the debugger's virtual address extension to use space IDs. The addresses in the **List** and **Data.dump** windows will be extended with a space ID (e.g **0000**:FFFFFFF81472C9A).




```
SYStem.Option.MMUSPACES ON     ; enable space IDs to virtual addresses
```

Remark: Older documentation and TRACE32 software uses **SYStem.Option.MMU ON** instead of **SYStem.Option.MMUSPACES ON**. Please use only the new naming.

The **SYStem.Option.MMUSPACES** should be enabled at the beginning of the script before loading any debug symbols.

Set Single Step Behavior

While single stepping, external interrupts may occur. On some architectures, this leads with the next single step into the interrupt handler. This effect normally disturbs during debugging. The following sequence masks external interrupts while executing assembler single steps. Keep interrupts enabled during HLL single steps to allow paging while stepping through source code.

```
SETUP.IMASKASM ON       ; suppress interrupts during assembler stepping
SETUP.IMASKHLL OFF      ; allow interrupts while HLL single stepping
```

If an assembler single step causes a page fault, the single step will jump into the page fault handler, regardless of the above setting. The debugger will restore the interrupt mask to the value before the single step. So it might be wrong at this state and cause an unpredictable behavior of the target.

Open a Terminal Window

You can open a serial terminal window in TRACE32 using the **TERM** command:

```
TERM.RESet                          ; reset old TERM settings
TERM.METHOD COM com1 115200. 8 NONE 1STOP NONE
                                    ; for com10 use \\.\com10
TERM.SIZE 80. 1000.                 ; define size of the TERM window
TERM.SCROLL ON                      ; enable scrolling
TERM.Mode VT100
TERM.view                           ; open the TERM window
SCREEN.ALways                       ; TERM window always updated
```

You can also use the `term.cmm` script available in the TRACE32 installation under `~~/demo/etc/terminal/serial` which takes two arguments: the COM port and the baud rate e.g.

```
DO ~~/demo/etc/terminal/serial/term.cmm COM1 115200.
```

TRACE32 allows to send data to the terminal window from a script file using the command **TERM.Out**:

```
TERM.OUT "bootm 0x20000000" 10. ; 10. is the ascii code of LF
```

Moreover, TRACE32 allows to set a trigger for the occurrence of a specific string in the terminal window using the command **TERM.TRIGGER**. The PRACTICE function **TERM.TRIGGERED(**<*channel*>**)** returns then if the trigger has occurred.

```
; wait until the string "login" appears in the terminal window
TERM.TRIGGER "login:"
WAIT TERM.TRIGGERED(D:0)
```

Load the Kernel Symbols

You can load the kernel symbols using the **Data.LOAD.Elf** command. Without any further options, this command loads the symbols and download the code to the target. In order to only load the kernel symbols into the debugger without downloading the code, you need to use the **/NOCODE** option.

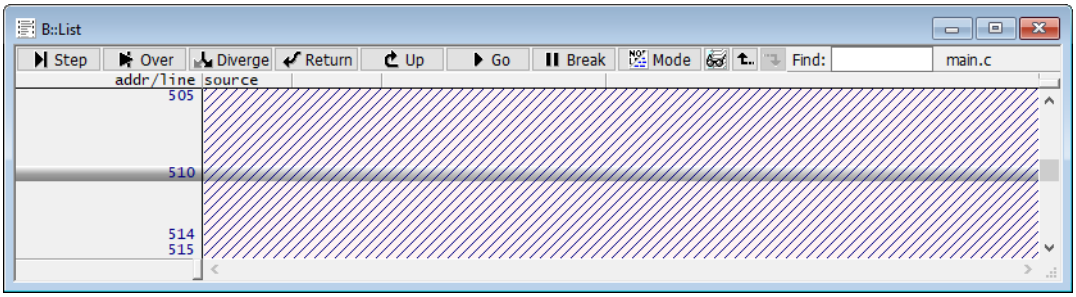
```
Data.LOAD.Elf vmlinux /NOCODE
```

For some older GNU compilers, you also need to use the /GNU option:

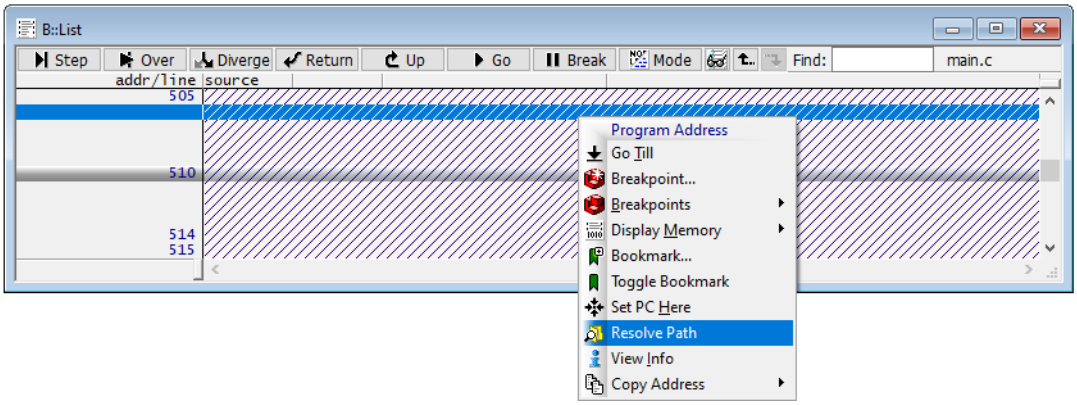
```
Data.LOAD.Elf vmlinux /NOCODE /GNU
```

Displaying the Source Code

If you are not running TRACE32 on the host where you compiled your kernel, the debugger, which uses per default the compile path to find the source files, will not find these files. The **List** window will display in this case hatches instead of the source code:

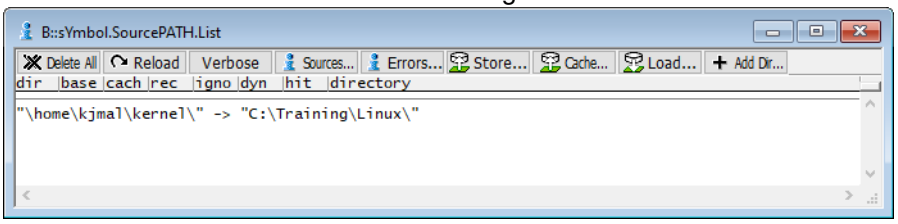


The easiest way to inform the debugger about the path of the source file is to do a right mouse click in the hatched area then select **Resolve Path**. A file search dialog will appear.



You just need then to browse to the source code file. The result of **Resolve Path** is a source path translation which will be used to locate all kernel source code files. This means that you have to resolve the path of a single source code file and all other kernel sources will be automatically found by TRACE32.

You can see the result of **Resolve Path** using the command **sYmbol.SourcePATH.List**.



Using the button **Store...**, the resulting **sYmbol.SourcePATH.Translate** command can be saved in a PRACTICE script.

```
sYmbol.SourcePATH.Translate  "\home\kjmal\kernel\" "C:\Training\Linux\"
```

Download the Kernel

It is normally the task of the boot-loader to load the kernel e.g. from the hard drive to the RAM. However, you can also use the debugger to download the kernel to the target memory over JTAG. In this case you just need to omit the **/NoCODE** option in the **Data.LOAD.Elf** command. We use here the memory class `A:` (absolute addressing) to download the code on the physical memory:

```
Data.LOAD.Elf vmlinux A:0
```

This command will load the kernel symbols and download the kernel at the physical address `0x0`.

To be able to start the kernel, you can either set up the registers and the kernel boot parameters with the debugger or download the kernel when the instruction pointer is at the kernel entry point (at this time, everything has already been set up by the boot-loader).

Downloading the Kernel Code at the Kernel Entry

You can set an on-chip breakpoint at the kernel entry point which is usually at the address `0x01000000` and let the system run. When you stop at the breakpoint, you can then download the kernel to the target memory. In this case, no further settings are needed since everything has been prepared by the boot-loader:

```
Go 0x01000000 /Onchip
WAIT !STATE.RUN()

Data.LOAD.Elf vmlinux A:0
```

Then you can simply continue the execution:

```
Go                              ; let the kernel boot
```

Downloading the Kernel after the Boot-loader Target Initialization

You can stop the boot-loader just after the target initialization and download the kernel. This way, you need to set the values of several registers and to set up the kernel boot parameters manually. Moreover, you need to enable the protected mode and the 64bit mode for the 64bit kernel.

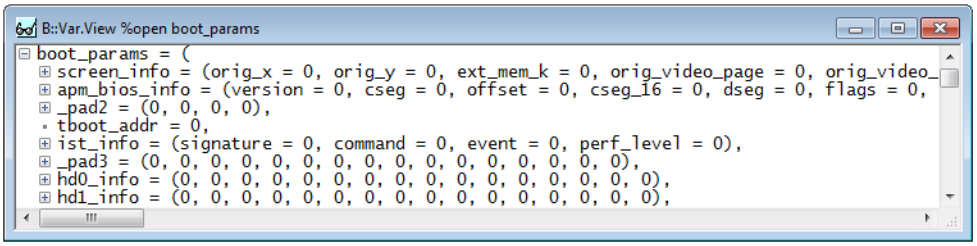
Setting the CPU Registers

The instruction pointer should be set to value defined by `CONFIG_PHYSICAL_START` and the stack pointer to a valid address e.g. `0x00010000`.

```
Register.REset                  ; reset all registers
Register.Set ESP 0x00010000     ; initialize stack pointer
Register.Set EIP 0x01000000     ; set IP to start of vmlinux
```

Setting the Kernel Boot Parameters

The kernel boot parameters are located in a structure of type (`struct boot_params`) pointed by the register `ESI`. You can access this structure after the kernel has booted using the `boot_params` symbol.



In the following example, we first set the `boot_params` area to zero and then set the parameters `alt_mem_k`, `hrd.type_pf_loader`, `hdr.ramdisk_image` and `hdr.ramdisk_size` (since we use a ramdisk as a file system) as well as the boot command line `ptr hdr.cmd_line_ptr`. The offsets of the structure elements are hard coded.

```
&bpb=0x20000                ; base address of boot parameters
Register.Set ESI &bpb       ; set ESI to point to struct boot_params
Data.Set (&bpb+0x0000)++0x0fff 0x0   ; empty boot params area
Data.Set &bpb+0x01e0 %Long 0x3fc00   ; alt_mem_k=(256-1)*1024kB=256-1MB
Data.Set &bpb+0x0210 %Byte 0x80      ; hdr.tpye_of_loader = U-Boot
Data.Set &bpb+0x0218 %Long 0x02000000  ;  hdr.ramdisk_image
Data.Set &bpb+0x021c %Long 0x00800000  ;  hdr.ramdisk_size
Data.Set &bpb+0x0228 %Long &bpb+0x1000 ; cmd_line_ptr
Data.Set &bpb+0x1000 "console=ttyS1,115200 console=ttyUSB0 "
Data.Set , "initrd=0x02000000,0x800000 root=/dev/ram "
Data.Set , "mem=240M slram=appdisk,0x0F000000,+0x1000000 "
Data.Set , 0
```

Set up the Protected Mode

The Linux kernel runs in protected mode. If you stop the bootloader before the protected mode has been enabled then you need to prepare the registers and descriptor tables manually for the protected mode. You can use for this the `setup_protected_mode.cmm` script available in the TRACE32 demo directory.

Set up the 64 Bit Mode

In case you are using a 64 bit kernel, you also need to set the CPU in 64 bit mode. A script is also available for this purpose in the TRACE32 demo directory under
~~/demo/x64/kernel/linux/board/setup_64bit_mode.cmm

Download the File System

In case you are using a ramdisk image as file system, you can download this image to the target memory using the **Data.LOAD.Binary** command:

```
Data.LOAD.Binary ramdisk.image.gz A:0x02000000 /NoClear
```

You need to use the /NoClear option here, otherwise the already loaded kernel symbols will be cleared. We also use here the “A:” memory class to force downloading the data to the physical memory. We use the `0x02000000` address since this is what has been specified in the kernel boot parameters (“`initrd=0x02000000`”).

Set up the Debugger Address Translation

The following settings have to be done by the Linux-aware debugging script in order to give the debugger access to the whole system including kernel, kernel modules and user space applications.

Kernel Page Table and Default Translation

The debugger needs to have access, at any time, to the kernel page table which contains translations for mapped address ranges owned by the kernel. Moreover, the kernel may use one of different formats to store translations in the kernel page table. The Linux-aware debugging script has thus to inform the debugger about the format and the logical address of the kernel page table as well as the logical to physical address translation for kernel addresses.

All these settings can be done using the command **MMU.FORMAT** e.g

```
MMU.FORMAT STD swapper_pg_dir 0xc0000000--0xcfffffff 0x0
```

The first argument of this command is the format of the kernel page table. Please check **“OS Awareness Manual Linux”** (rtos_linux_stop.pdf) for actual format specifier.

The second argument is a kernel symbol pointing to the start of the kernel page table and is usually called `swapper_pg_dir` for a 32bit kernel and `init_level4_pgt` or `init_top_pgt` for a 64bit kernel.

The third argument is the kernel logical to physical address translation called *kernel translation* or *default translation*. This range should at least include the whole kernel page table. You can generally use the kernel `_text` label as start of this range and the label `_end` minus 1 as its end.

```
MMU.FORMAT LINUX swapper_pg_dir _text--(_end-1) 0x80000000
```

The last argument is the physical address that corresponds to the used logical range start. You can get this address using the command **MMU.List PageTable** with the logical address as argument e.g.

```
MMU.List PageTable _text
```

Example setup for x64:

```
LOCAL &base_addr
IF sYmbol.EXIST(init_level4_pgt)
  &base_addr="init_level4_pgt"
ELSE
  &base_addr="init_top_pgt"
MMU.FORMAT LINUX64 &base_addr _text--(_end-1) 0x01000000
```

Direct and Kernel Text Mappings

For x64, you additionally need to create static translations for the direct mapping and the kernel text mapping. Please refer to `Documentation/x86/x86_64/mm.txt` for more information about these address mappings.

```
TRANSlation.Create 0xffff880000000000--0xffffc7ffffffffff 0x0
TRANSlation.Create 0xffffffff80000000--0xffffffff9fffffff 0x0
```

COMMON Range

With enabled space IDs, debug symbols as well as address translation are specific to one space ID. In user space, the **List** window displays for instance only the debug symbols of the current process. Moreover, in order to do the virtual to physical translation for an address with a given space ID, the debugger accesses the page tables corresponding to that space ID. User space application may be however executing in kernel space on behalf of the kernel. This means that it is usual to have the program counter pointing to a kernel address, e.g. a kernel function, with a user process space ID. The debugger has to display in kernel space the kernel symbols and use the kernel page tables independently of the space ID. The command **TRANSlation.COMMON** informs the debugger about common address range for all processes, i.e. everything above the process address range including kernel and kernel modules.

For a 32 bit kernel, the common range starts at `CONFIG_PAGE_OFFSET` (e.g. `0xC0000000`) and ends at `0xFFFFFFFF`.

```
TRANSlation.COMMON 0xc0000000--0xffffffff
```

The following common range can always be used for 64 bit kernels as user space is always below the address `0xf000000000000000`.

```
; common range for 64 bit kernels:
TRANSlation.COMMON 0xf000000000000000--0xffffffffffffffff
```

The debugger address translation and MMU table walk have to be enabled respectively using the commands **TRANSlation.ON** and **TRANSlation.TableWalk ON**.

```
TRANSlation.TableWalk ON
TRANSlation.ON
```

If the table walk is enabled, when accessing a virtual address which has no mapping in the debugger local address translation list (**TRANSlation.List**), the debugger tries to access the MMU page tables to get the corresponding physical address and then accesses the memory physically.

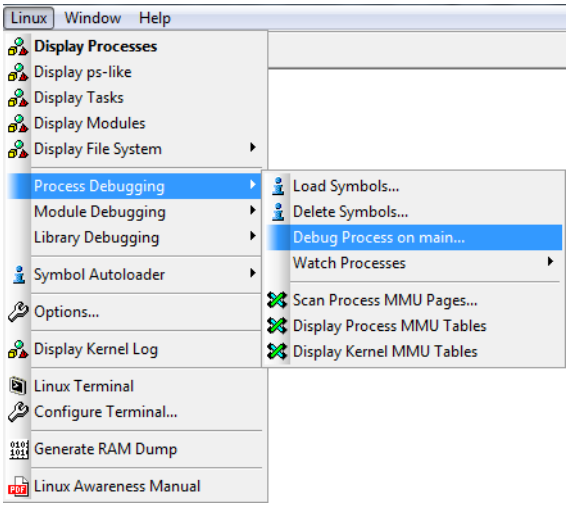
Set up the Linux Awareness

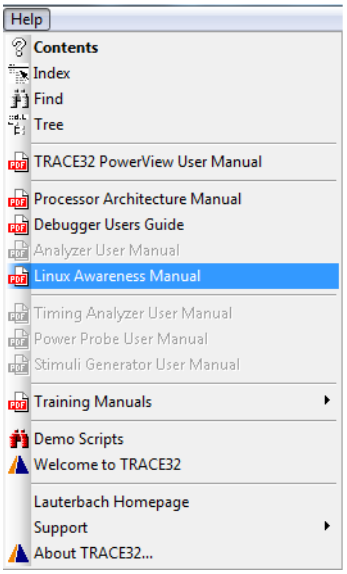
We need to load now the Linux awareness and Linux menu in TRACE32.

- For kernel versions 2.x, the Linux awareness is based on the file linux2.t32 located under *~~/demo/<x86|x64>/kernel/linux/linux-2.x/*
- The Linux awareness for kernel versions 3.x and newer is based on the file linux.t32 located under *~~/demo/<x86|x64>/kernel/linux/awareness/*

```
; load the awareness on x64
TASK.CONFIG ~~/demo/x64/kernel/linux/awareness/linux.t32
; load Linux menu:
MENU.ReProgram ~~/demo/x64/kernel/linux/awareness/linux.men
```

The Linux menu file includes many useful menu items developed for the TRACE32-GUI to ease Linux Debugging.





The Linux awareness and Linux menu are based on scripts available under:
~~/demo/<x86|x64>/kernel/linux/awareness.

These scripts are called by the Linux awareness and the Linux menu. You should thus always load the awareness from the TRACE32 installation directory to avoid compatibility problems between the Linux awareness and the mentioned scripts. If you load the Linux awareness outside the TRACE32 installation, you will get the warning **"please use awareness files from TRACE32 installation directory"**.

Disable Watchdogs and Lockup Detection

The Linux kernel includes mechanisms to detect lockups and hangs. These mechanisms could interfere with the debug functionality. Lauterbach provides within the Linux awareness a script to disable watchdogs and lockup detection by writing to specific kernel variables. This script can be found in the TRACE32 demo directory under <arch>/kernel/linux. Since the script accesses kernel variables, you should call it after the MMU has been enabled e.g. after stopping at start_kernel:

```
Go start_kernel /Onchip
WAIT !STATE.RUN()
DO ~~/demo/x64/kernel/linux/disable_watchdogs.cmm
```

Please contact the Lauterbach support in case you don't find this script in your TRACE32 installation.

Mark the Kernel Address Space

For better visibility, you can mark the kernel address space to be displayed with a red bar.!

```
GROUP.Create "kernel" 0xC0000000--0xFFFFFFFF /RED ; 32 bit kernel
```

Example Linux Setup-Scripts

You can find demo startup scripts for different target boards in the TRACE32 installation directory under ~~/demo/x86/kernel/linux/board and ~~/demo/x64/kernel/linux/board. You can also search for the newest scripts in the Lauterbach home page under the following link:

https://www.lauterbach.com/frames.html?scripts.html

The first example script set up Linux aware debugging for a 32 bit kernel running on the Intel Galileo board. In this example the kernel is already running on the target.

```
REset
WinCLEAR

SYStem.CPU QUARK
SYStem.Option.MMUSPACES ON ; enable space IDs to virtual addresses
SYStem.Attach
SETUP.IMASKASM ON          ; lock interrupts while single stepping

; Open a serial terminal window
DO ~~/demo/etc/terminal/serial/term.cmm COM1 115200.

; Open a Code Window -- we like to see something
WINPOS 0. 0. 75. 20.
List
SCREEN

; Load the Linux kernel symbols
Data.LOAD.Elf vmlinux /NOCODE

MMU.FORMAT PAE swapper_pg_dir 0xC0000000--0xCFFFFFFF 0x0
TRANSLATION.COMMON 0xC0000000--0xFFFFFFFF
TRANSLATION.TableWalk ON
TRANSlation.ON

; Initialize Linux Awareness
PRINT "initializing multi task support..."
; loads Linux awareness:
TASK.CONFIG ~~/demo/x86/kernel/linux/linux-3.x/linux3.t32
; loads Linux menu:
MENU.ReProgram ~~/demo/x86/kernel/linux/linux-3.x/linux.men

; Group kernel area to be displayed with red bar
GROUP.Create "kernel" 0xC0000000--0xFFFFFFFF /RED


ENDDO
```

The second example script set up Linux aware debugging for a 64 bit kernel running on the Crown Beach Board. We connect to the target using the **SYStem.Up** command which reset the cores. We let then boot-loader initialize the target hardware. The boot-loader is stopped before the Linux kernel is loaded. We continue then the setup using the debugger.The kernel as well as the ramdisk image are downloaded to the target memory over JTAG. The script also set the initial values for the CPU registers and the kernel boot parameters.

```
RESet

; setup of ICD
PRINT "initializing..."
SYStem.CPU ATOMZ5XX
SYStem.JtagClock 20MHz
SYStem.Option.MMUSPACES ON     ; enable space IDs to virtual addresses
SYStem.Up

; Open a serial terminal window
DO ~~/demo/etc/terminal/serial/term.cmm COM1 115200.

SETUP.IMASKASM ON          ; lock interrupts while single stepping

 ; Let the boot monitor set up the board
    Go
    PRINT "target setup..."
    WAIT 10.s
    Break

; Load the Linux kernel code and symbols
Data.LOAD.Elf vmlinux A:0 /GNU

; Loading RAM disk
Data.LOAD.Binary ramdisk.image.gz A:0x02000000 /NoClear /NoSymbol

; Initialize CPU protected mode. 0x10000 is the GDT base address
DO setup_protected_mode.cmm 0x10000

; Set PC on physical start address of the kernel
Register.Set EIP 0x01000000
; Initialize stack pointer
Register.Set ESP 0x00010000

; Setup boot_params in a separate script (please refer to 4.b)
DO setup_boot_params.cmm

; Open a Code Window -- we like to see something
WINPOS 0. 0. 75. 20.
List
SCREEN
```

continued on next page.

continued:

```
PRINT "initializing debugger MMU..."
LOCAL &base_addr
IF sYmbol.EXIST(init_level4_pgt)
  &base_addr="init_level4_pgt"
ELSE
  &base_addr="init_top_pgt"
MMU.FORMAT LINUX64 &base_addr 0xffffffff80000000--0xffffffff9fffffff 0x0

TRANSLATION.Create 0xffff880000000000--0xffffc7ffffffffff 0x0
TRANSLATION.Create 0xffffffff80000000--0xffffffff9fffffff 0x0
TRANSLATION.COMMON 0xffff880000000000--0xffffffffffffffff
TRANSLATION.TableWalk ON
TRANSlation.ON

; Initialize Linux Awareness
PRINT "initializing multi task support..."
; loads Linux awareness:
TASK.CONFIG ~~/demo/x64/kernel/linux/linux-3.x/linux3.t32
; loads Linux menu:
MENU.ReProgram ~~/demo/x64/kernel/linux/linux-3.x/linux.men

; Group kernel area to be displayed with red bar
GROUP.Create "kernel" 0xffffffff80000000--0xffffffffffffffff /RED

; set CPU in 64bit mode (see IA-32 manual, Vol 3 Ch 9.8.5), specify GDTB
; and page directory
DO ../setup_64bit_mode.cmm

Go x86_64_start_kernel
WAIT !STATE.RUN()

SYStem.Option.C0Hold ON ; prohibit power down

PRINT "booting Linux..."
Go

ENDDO
```

Debugging the Linux Components

Each of the components used to build a Linux system needs a different handling for debugging. This chapter describes in detail, how to set up the debugger for the individual components.

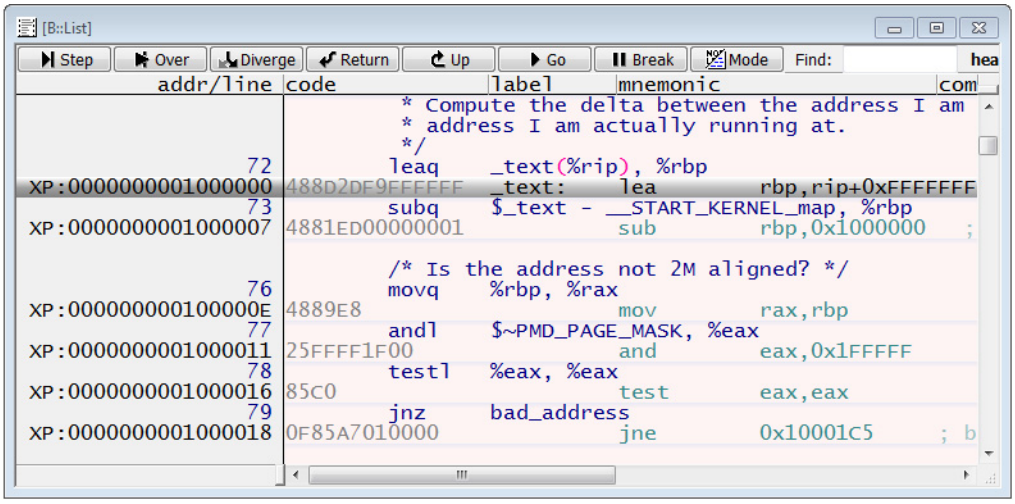
"OS Awareness Manual Linux" (rtos_linux_stop.pdf) gives additional detailed instructions.

The Kernel

We differentiate here between the part of the kernel boot running with disabled MMU, that we call kernel startup, and the rest of the kernel.

Kernel Startup

The Linux kernel starts executing with disabled MMU, i.e. at physical address space. The debug symbols of the kernel startup are however mapped to virtual addresses. The boot-loader jumps into the kernel startup routine (`phys_startup_32` / `phys_startup_64`) generally located at the address `0x01000000`. It starts at physical address space, does some initialization and set up the MMU. **Finally the kernel startup switches into logical address space**.



To be able to see the debug symbols for the kernel startup, the kernel should be loaded with an offset. The offset is needed here since the kernel runs on physical addresses. The kernel symbols are however linked to logical addresses.

Data.LOAD.Elf vmlinux *<physical_start_addr>-<logical_start_addr>* **/NoCODE**

Please note that a single minus sign "-" is used here which means that we subtract the logical start address from the physical start address.

Specifying an offset is only needed to debug the kernel startup in HLL. As soon as the kernel jumps to logical addresses after enabling the target MMU, the kernel symbols should be loaded without any offset.

Loading the kernel symbols with an offset is only needed if you want to debug the kernel startup code which runs with disabled MMU.

If the address extension with the memory space IDs is enabled, the kernel symbols will be mapped to the space ID `0x0000`. The current task is however at this time unknown, so the current space ID is `0xFFFF`. Consequently, the **List** window will not display the debug symbols. In order to see the debug symbols corresponding to the kernel startup code, you have additionally to disable the address extension.

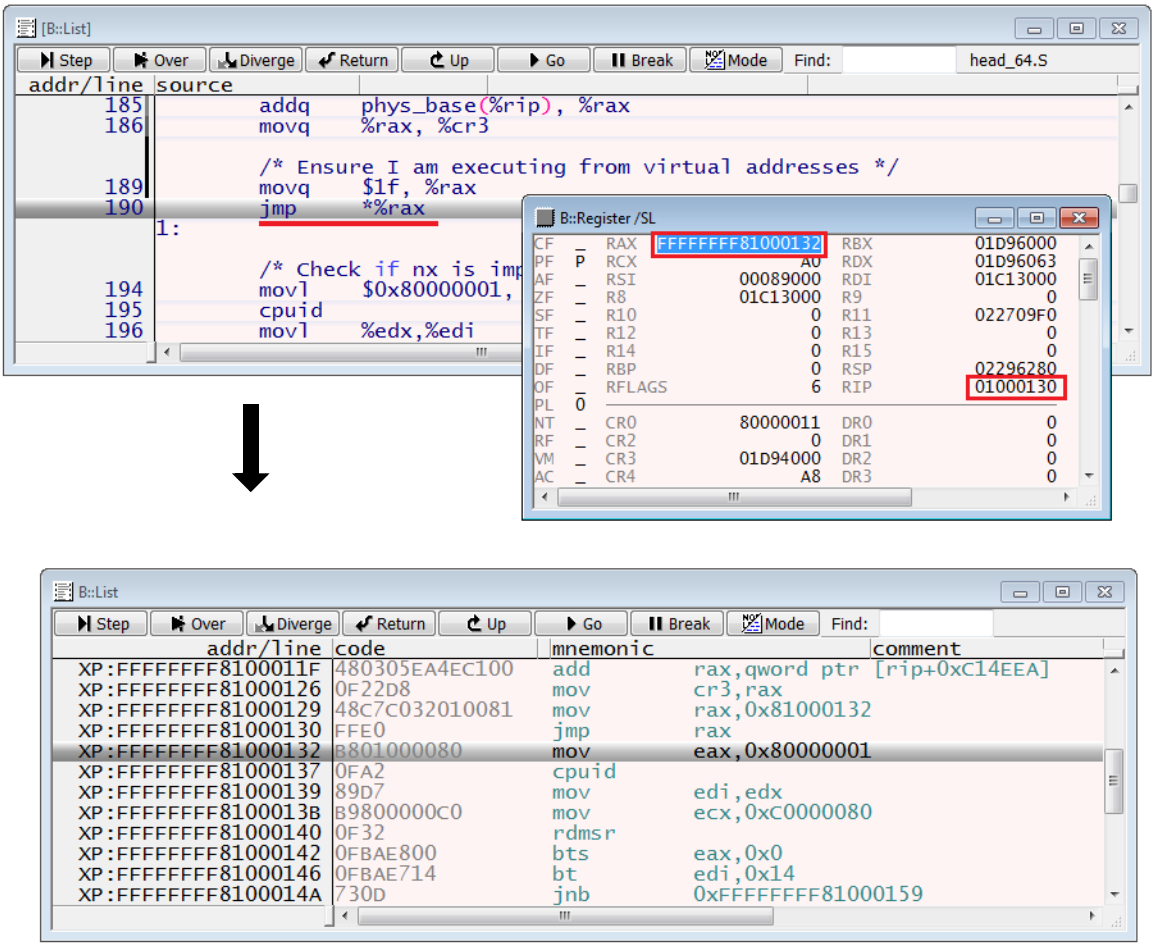
```
SYStem.Option.MMUSPACES OFF
```

As long as the debugger MMU has not been enabled, you have to use on-chip breakpoints on kernel functions. Please note however, that the kernel may reset on-chip breakpoints when booting.

Alternatively, you can first set an on-chip breakpoint at `86_x64_start_kernel` then you can use software breakpoint on the rest of the kernel boot.

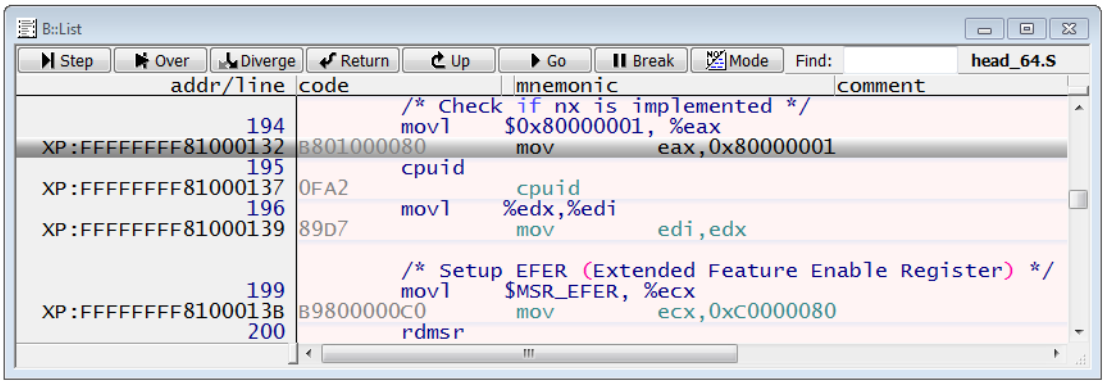
```
Go x86_x64_start_kernel /Onchip
WAIT !STATE.RUN()
Break.Set usb_init /SOFT
```

After enabling the target MMU, the kernel startup code will jump to logical addresses:



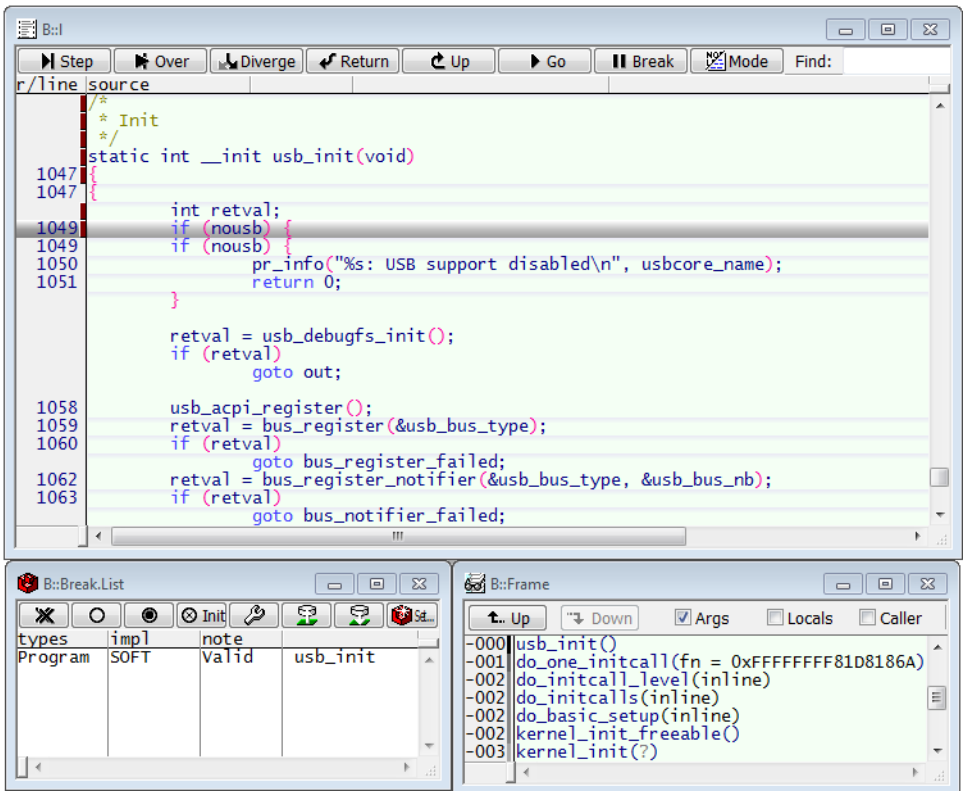
From now on, the Linux kernel runs completely in logical address space. The symbols are all bound to these logical addresses, so simply load the Linux symbols without any offset:

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



Now you need to set up the debugger address translation and load the Linux awareness as described in the previous chapter.

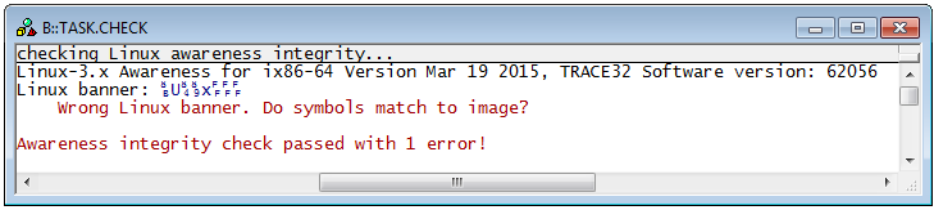
You can use now software breakpoints in the kernel range since all the kernel code is accessible.



Verifying Image and Symbols

It is very important that the kernel running on the target is from the **very same build** as the symbol file loaded into the debugger. A typical error is that the loaded `vmlinux` file doesn't match the executed kernel on the target. This can lead to different errors.

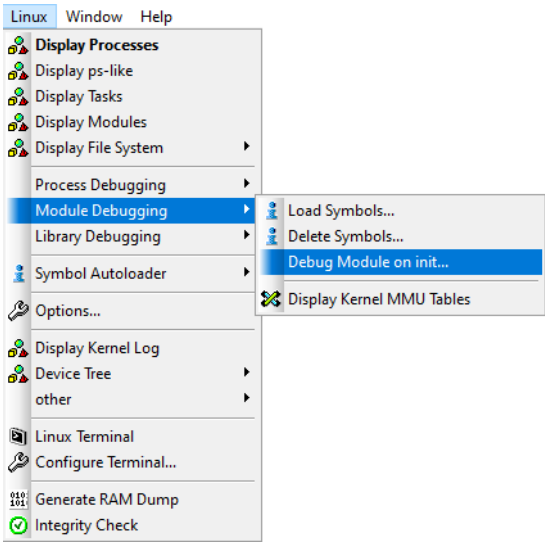
You can check if the kernel code matches the loaded symbols using the **TASK.CHECK** command. First let the kernel boot, stop the target and then execute **TASK.CHECK**. When the symbols does not match the kernel code, you will get an error message in this window:

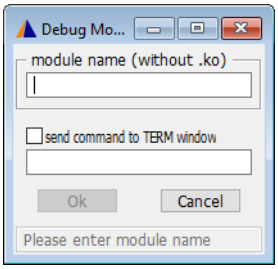


Please note that **TASK.CHECK** command only does a basic check based on the `linux_banner` string. In some cases, this basic check cannot detect that there is a mismatch between the kernel code and the loaded kernel debug symbols. Please refer to **"Troubleshooting"**, page 48 for more details.

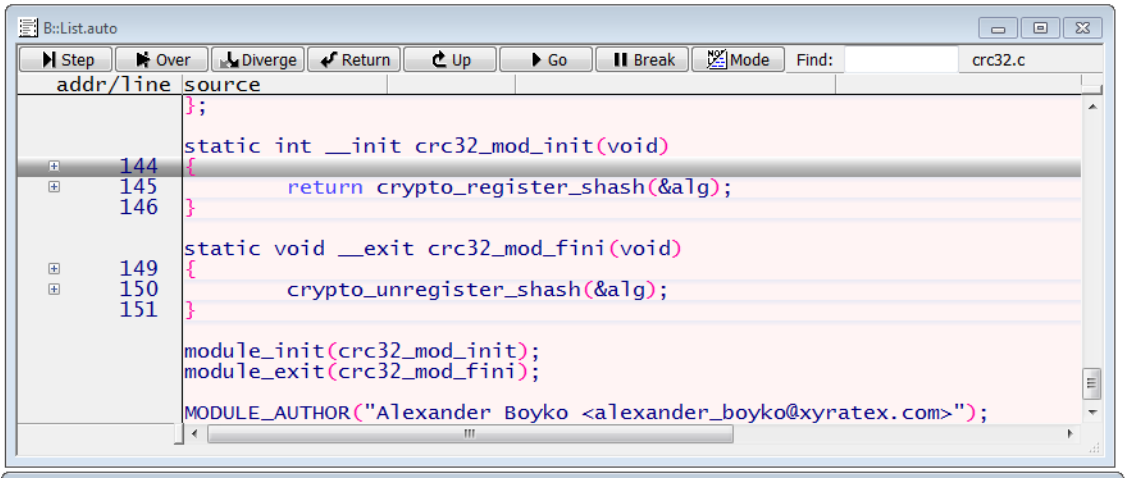
Kernel Modules

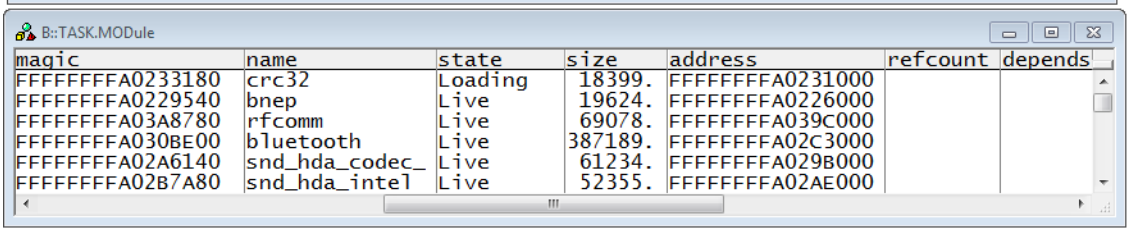
Kernel modules are loaded and linked into the kernel at run-time. To ease the debugging of kernel modules, the enhanced Linux menu offers the item **"Debug Module on init..."**. After selecting this menu point, a small dialog will pop-up where you can specify the name of the kernel module to be debugger (without extension). Optionally, you can instruct the dialog to send a specific command to the TRACE32 terminal window in order to load the kernel module.





The **"Debug Module on init..."** menu point is based on the script `mod_debug.cmm` available in the path of the Linux awareness. The script sets a breakpoint at a kernel function that is executed when a new kernel module is loaded. As soon as the breakpoint is hit, the TRACE32 Symbol Autoloader will load the kernel module symbols and relocate each section based on the information delivered by the Linux awareness. Finally, an on-chip breakpoint is set on the module `init` function and the execution is resumed.





If the Symbol Autoloader cannot find the module's ko file, a file browser will pop-up. If you want the debugger to automatically find your kernel module, you need to add its path to the TRACE32 search paths using the command **sYmbol.SourcePATH.SetDir**. Alternatively, you can define a ROOTPATH using the command **TASK.sYmbol.Option ROOTPATH**. Please refer to **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information about this command.

The script `mod_debug.cmm` can also be called from the TRACE32 command line or from a different script. By using the `/dialog` argument, the script will open the same dialog displayed after selecting the menu point **"Debug Module on init..."**:

```
DO ~~/demo/x86/kernel/linux/awareness/mod_debug.cmm /dialog
```

You can also specify instead the name of the module to be debugged (without extension) as first argument:

```
DO ~~/demo/x86/kernel/linux/awareness/mod_debug.cmm crc32
```

The script additionally accepts the following arguments:

- `/term "<cmd>"`: send the command *<cmd>* to the TRACE32 terminal window in order to load the module e.g. `/term "insmod crc32.ko"`
- `/timeout <timeout>`: exit the script with an error message in case any of the breakpoints set by the script is not reached within the given timeout e.g. `/timeout 5.s`
- `/stopat <label>`: set the on-chip breakpoint at *<label>* instead of the module's `init` function.

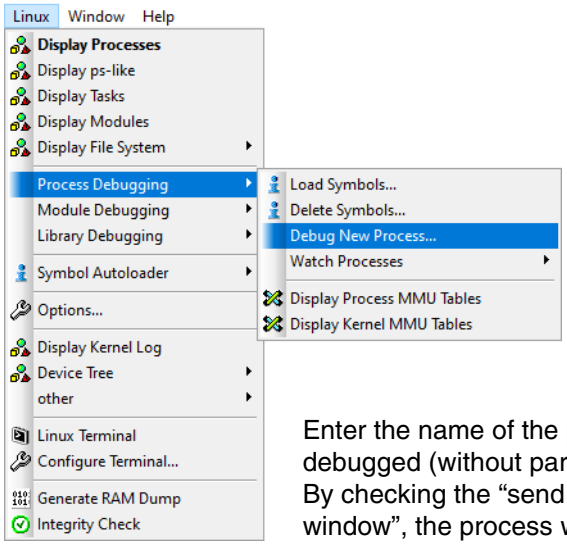
You can also load the debug symbols of already loaded modules by selecting the TRACE32 menu **Linux > Module Debugging > Load Symbols...** or using the command **TASK.sYmbol.LOADMod**

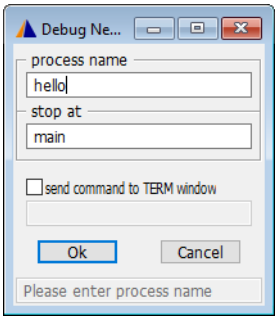
```
TASK.sYmbol.LOADMod "crc32"           ; load module symbols
```

If you remove a kernel module from the kernel, you should also remove its debug symbols in TRACE32 PowerView using the menu **Linux > Module Debugging > Delete Symbols...** or the command **TASK.sYmbol.DELeteMod**:

```
TASK.sYmbol.DELeteMod "crc32"         ; erase obsolete module symbols
```

The Linux menu provides a comfortable way to debug processes from its start. You just need to select the menu **Linux > Process Debugging > Debug New Process...** then enter the name of the process. The process will per default be stopped at its `main` function. You can also specify a different process function under "stop at". Optionally, you can instruct the dialog to send a specific command to the TRACE32 terminal window in order to start the process.





Enter the name of the process to be debugged (without parameters).
By checking the "send command to TERM window", the process will be started from the TERM window.

The menu point **Debug New Process...** is based on the script `app_debug.cmm` available in the path of the Linux awareness. The script sets a breakpoint at a kernel function that is executed when a new process is started. As soon as the breakpoint is hit, the TRACE32 Symbol Autoloader will load the process symbols and set a task specific on-chip breakpoint at the `main` function of the given process function. Then the execution is resumed.

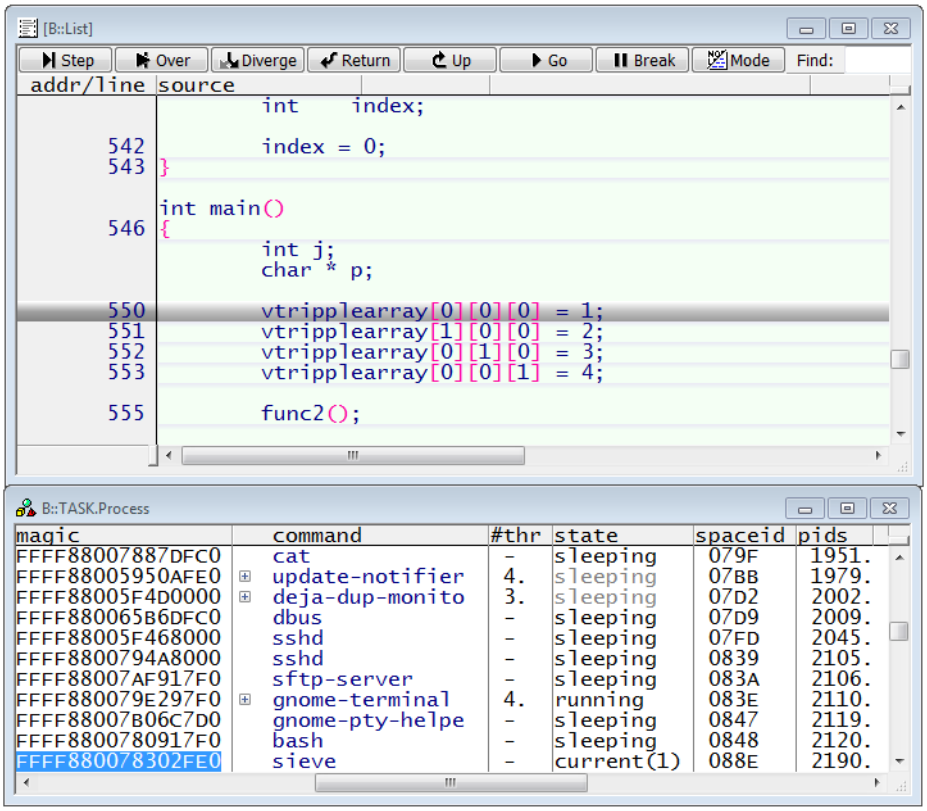
If the Symbol Autoloader cannot find the process' Elf file, a file browser will pop-up. If you want the debugger to automatically find your process' Elf file, you need to add its path to the TRACE32 search paths using the command **sYmbol.SourcePATH.SetDir**. Alternatively, you can define a ROOTPATH using the command **TASK.sYmbol.Option ROOTPATH**. Please refer to **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information about this command.

The script `app_debug.cmm` can also be called from the TRACE32 command line or from a different script. By using the `/dialog` argument, the script will open the same dialog displayed after selecting the menu point **"Debug New Process..."**:

```
DO ~~/demo/x64/kernel/linux/awareness/app_debug.cmm /dialog
```

You can also specify instead the name of the process to be debugged as first argument:

```
DO ~~/demo/x64/kernel/linux/awareness/app_debug.cmm sieve
```



The script additionally accepts the following arguments:

- `/term "<cmd>"`: send the command `<cmd>` to the TRACE32 terminal window in order to start the process e.g. `/term "/home/user/t32/sieve"`
- `/timeout <timeout>`: exit the script with an error message in case any of the breakpoints set by the script is not reached within the given timeout e.g. `/timeout 5.s`
- `/stopat <label>`: set the on-chip breakpoint at `<label>` instead of the process' `main` function.

You can also load the debug symbols of an already running process using the menu **Linux > Process Debugging > Load Symbols...** or the command **TASK.sYmbol.LOAD**

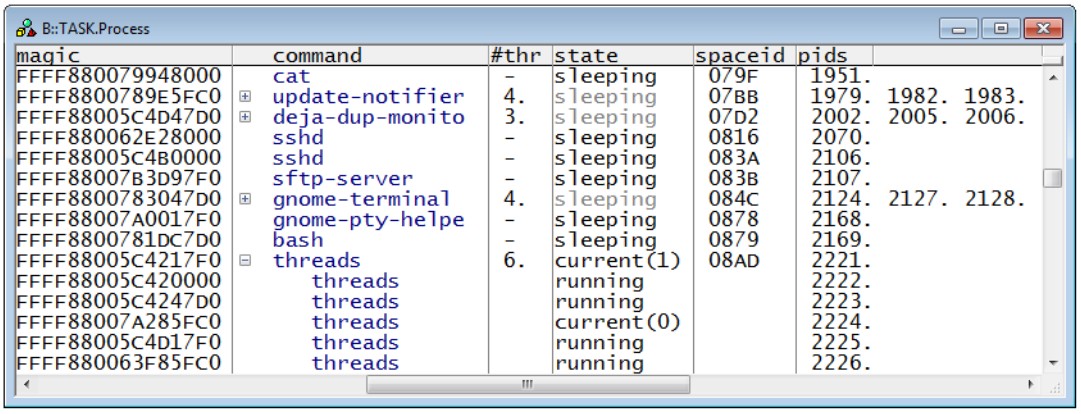
```
TASK.sYmbol.LOAD "sieve"                ; load process symbols
```

After the process exists, its debug symbols have to be deleted using the menu **Linux > Process Debugging > Delete Symbols...** or the command **TASK.sYmbol.Delete**

```
TASK.sYmbol.Delete "sieve"              ; delete process symbols
```

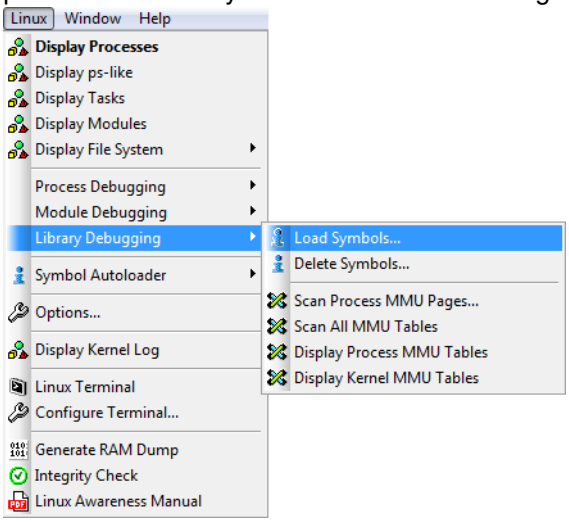
Threads

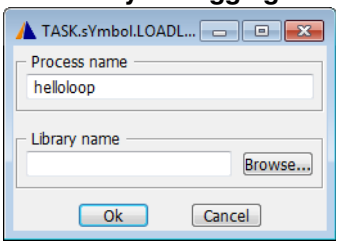
Threads are Linux tasks that share the same virtual memory space. The Linux awareness assignes the space ID of the creating process to all threads of this process. Because symbols are bound to a specific space ID, they are automatically valid for all threads of the same process. There is no special handling for threads. See chapter “Processes” how to load and handle these symbols.



Libraries

Libraries are loaded and linked dynamically to processes. Thus, they run in the virtual address space of the process and have dynamic addresses. To debug libraries, you can use the menu **“Library Debugging”**:

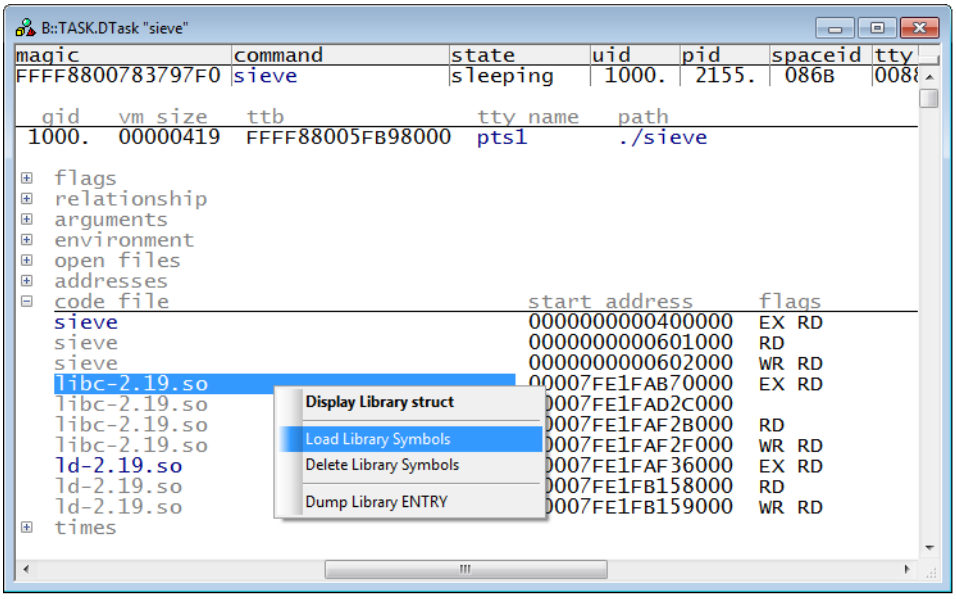




This menu point is based on the TRACE32 command **TASK.sYmbol.LOADLib**.

```
TASK.sYmbol.LOADLib "helloloop" "ld-2.2.5.so"   ; load library symbols
```

You can also display first the task list using the command **TASK.DTask** and then continue with double or right-clicks:



The debug symbols of the library will be automatically loaded by the TRACE32 Symbol Autoloader and relocated according to the information delivered by the Linux awareness. If the Symbol Autoloader cannot find the library's Elf file, a file browser will pop-up. If you want the debugger to automatically find your library's Elf file, you need to add its path to the TRACE32 search paths using the command **sYmbol.SourcePATH.SetDir**. Alternatively, you can define a ROOTPATH using the command **TASK.sYmbol.Option ROOTPATH**. Please refer to **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information about this command.

The library's debug symbols can be deleted using the menu point **Library Debugging > Delete Symbols...** or the command **TASK.sYmbol.DELeteLib**.

```
TASK.sYmbol.DELeteLib "helloloop" "ld-2.2.5.so" ; delete library symbols
```

You can also set up the Linux awareness in order to load all shared libraries of the current process or a given process. Examples:

Load all shared libraries for the current process:

```
TASK.sYmbol.Option AutoLOAD CURRLIB
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH
```

Add the libraries of process "hello" to the Symbol Autoloader, the debug symbols for each library will be loaded when the library's address range is accessed by any TRACE32 window:

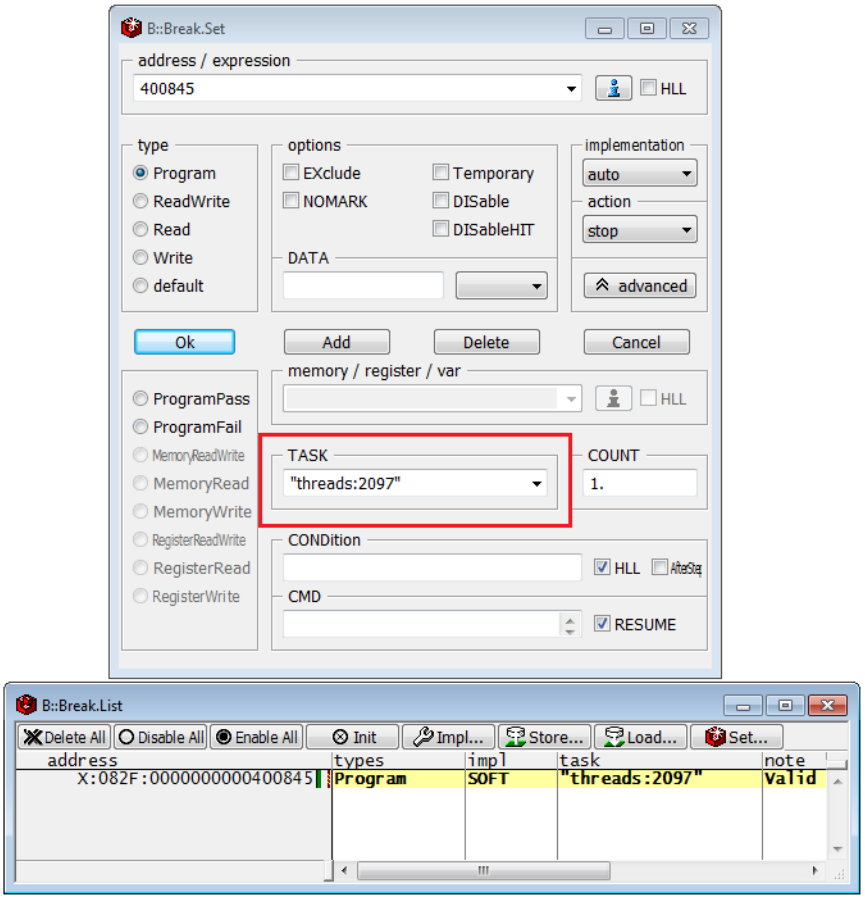
```
TASK.sYmbol.Option AutoLOAD ProcLib "hello"
sYmbol.AutoLOAD.CHECK
```

Task Related Breakpoints

You can set conditional breakpoints on shared code halting only if hit by a specified task

```
Break.Set myfunction /TASK "mytask"
```

When the breakpoint is hit, the debugger will check if the current task is the specified one. If it is not the case, the execution will be resumed.



Task Related Single Stepping

If you debug shared code with HLL single step, which is based on breakpoints, a different task could hit the step-breakpoint. You can avoid this by using the following command:

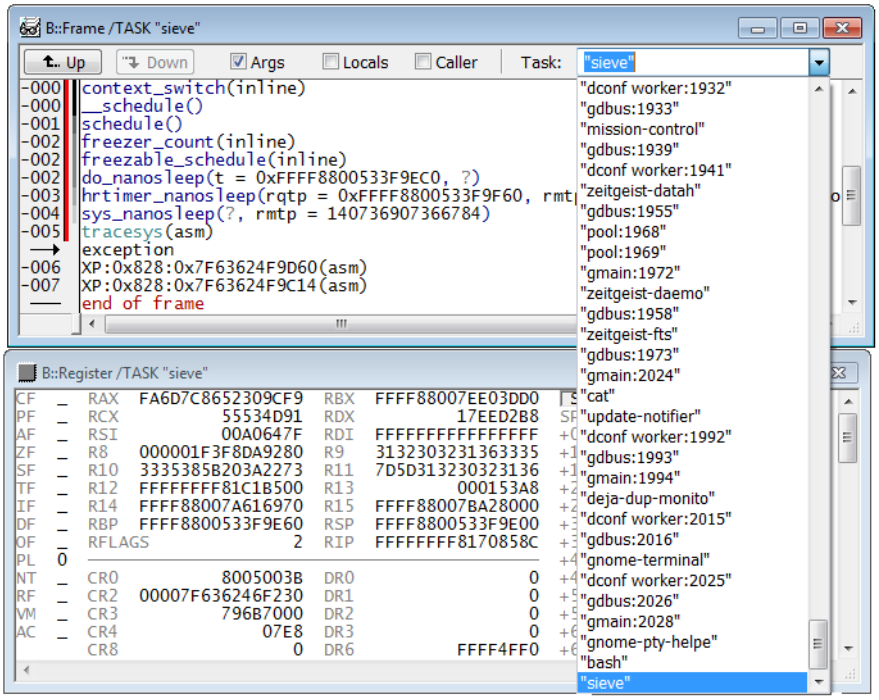
```
SETUP.StepWithinTask ON
```

Conditional breakpoints on the current task will be then used for step into / step over and you will not “leave” the task that you want to debug.

Task Context Display

You can display the memory or the registers of a task which is not currently executing. Moreover, you can display the stack frame of any running task on the system.

```
List /TASK "mytask"
Register /TASK "mytask"
Frame /TASK "mytask"
```

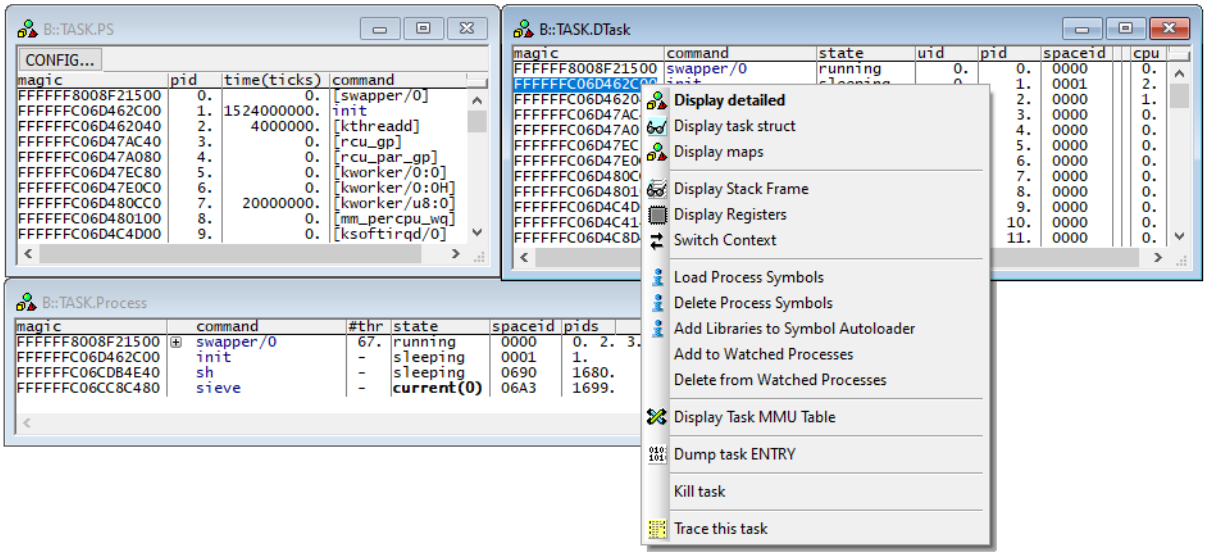


You can additionally “virtually” switch the context also from the **TASK.DTask** window by popup menu-item **“Switch Context”**.

The Linux awareness offers different commands to display kernel resources as the task list or the kernel module list. Most of these views can be opened from the Linux menu.

Display of System Resources

The Linux awareness offers three different views for displaying tasks using the commands **TASK.Process**, **TASK.PS** and **TASK.DTask**. Please refer to the documentation of these commands in **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information. These views can be opened from the Linux menu by selecting respectively **Display Processes**, **Display ps-like** and **Display Tasks**.



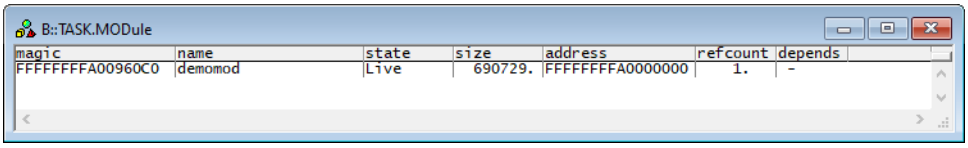
By doing a right mouse click on the task magic in these three views, you get a pull-down menu with the following options for the selected task:

- **Display detailed**: display additional information about the selected task (as the process arguments, environment variables or open files) by calling the command **TASK.DTask** with process magic as argument.
- **Display task struct**: display the kernel task structure for the selected task.
- **Display maps**: display the mapped memory regions for the selected task using the command **TASK.MAPS** similar to the Linux command `cat /proc/<pid>/maps`.
- **Display Stack Frame**: display the stack frame for the selected task. If the task is not currently executing, the Linux awareness retrieves the context information from the kernel structures.
- **Display Registers**: display the registers of the selected task. If the task is not currently executing, the Linux awareness retrieves the context information from the kernel structures.
- **Switch Context**: virtually switch the context to the selected task.

- **Load Process Symbols**: load the debug symbols of the selected process by calling the **TASK.sYmbol.LOAD** command.
- **Delete Process Symbols**: delete the debug symbols of the selected process by calling the **TASK.sYmbol.Delete** command.
- **Add Libraries to Symbol Autoloader**: update the autoloader table with the libraries of the current process. The debug symbols of these libraries will be automatically loaded as soon as their addresses are accessed by the debugger.
- **Add to Watched Processes**: add process to the process watch list. Refer to **TASK.Watch** for more information.
- **Delete from Watched Processes**: remove process from the process watch list. Refer to **TASK.Watch** for more information.
- **Display Task MMU Table**: display the task page table by calling the command **MMU.List TaskPageTable** with the process magic as argument.
- **Dump task ENTRY**: open a **Data.dump** window on the task entry point.
- **Kill Task**: write a pending kill signal to the task control structure which will cause the task to be killed after resuming the program execution.
- **Trace This Task**: do a selective trace on the code of the selected task.

Kernel Module List

You can display the list of loaded kernel modules by selecting the menu **Linux > Display Modules** which will call the **TASK.MODule** command.



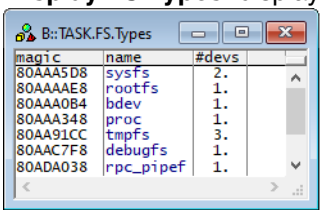
By doing a right mouse click on the module's magic, you get a pull down menu with the following options:

- **Display module struct**: display the module's kernel structure.
- **Load Module Symbols**: load the debug symbols of the selected kernel module
- **Delete Module Symbols**: delete the debug symbols of the selected kernel module
- **Dump module ENTRY**: dump the memory at the module entry.

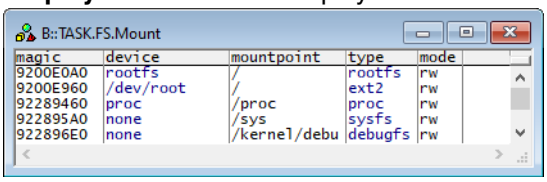
File System Information

The Linux awareness offers different view for displaying file system information. You can open these views from the menu **Linux > Display File System**:

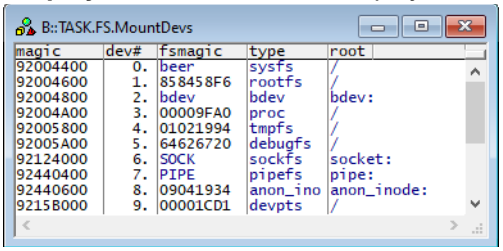
- **Display FS Types**: display all file system types that are currently registered in the Linux kernel.



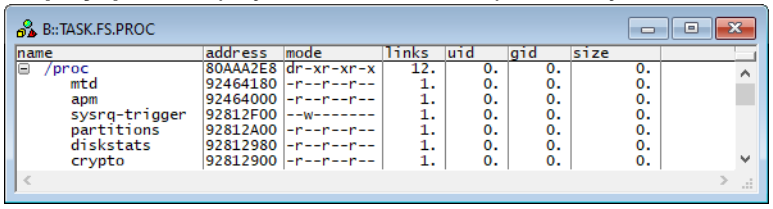
- **Display Mount Points**: display the current mount points.



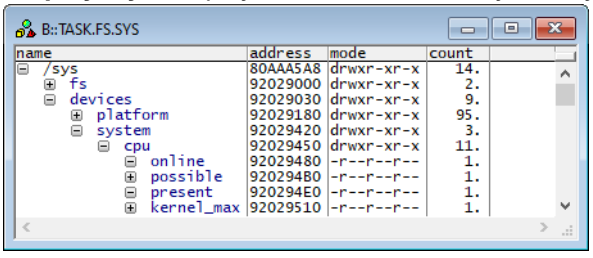
- **Display Mounted Devices**: display all currently mounted devices (i.e.super blocks).



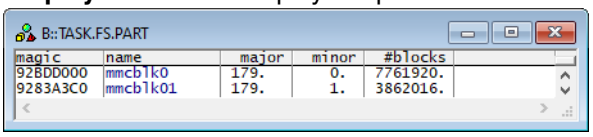
- **Display /proc**: display the content of the /proc file system.



- **Display /sys**: display the content of the /sys file system.



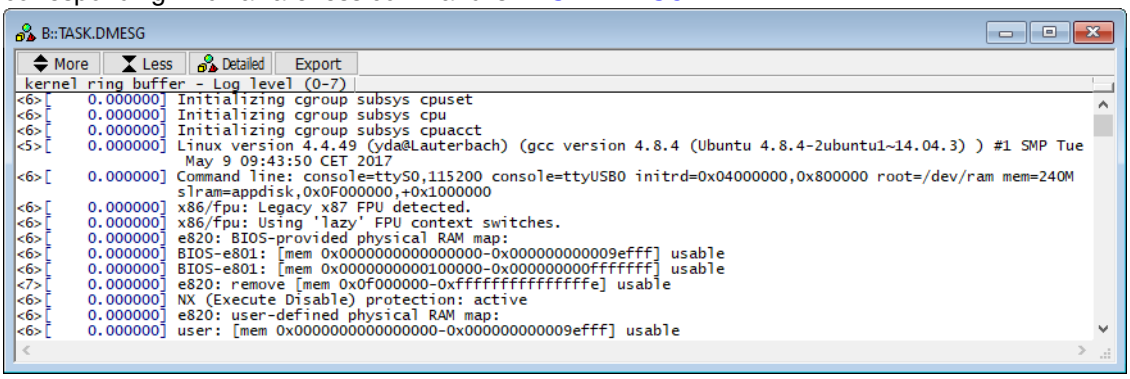
- **Display Partitions**: display the partition table.



Please refer to the documentation of the **TASK.FS** command for more information.

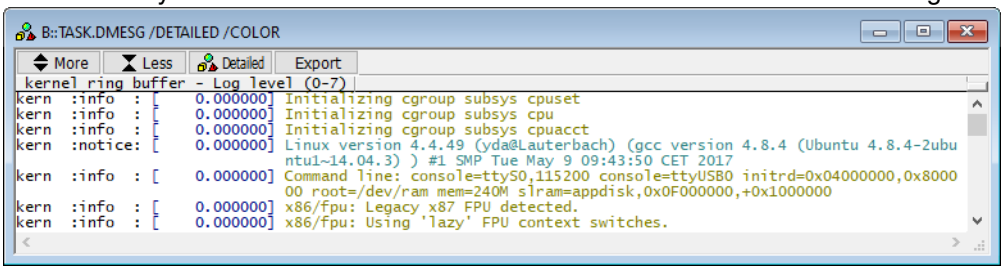
Kernel Log Buffer

By selecting the menu **Linux > Display Kernel Log** you can display the content of the kernel log buffer. The corresponding Linux awareness command is **TASK.DMESG**.

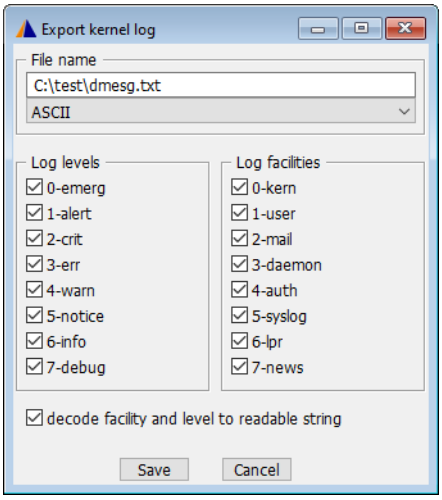


The **TASK.DMESG** window includes the following buttons:

- **More**: show more log levels.
- **Less**: show less log levels.
- **Detailed**: open the **TASK.DMESG /COLOR /DETAILED** window which will display the log level and the facility in a human readable format and use a different color for each log level.

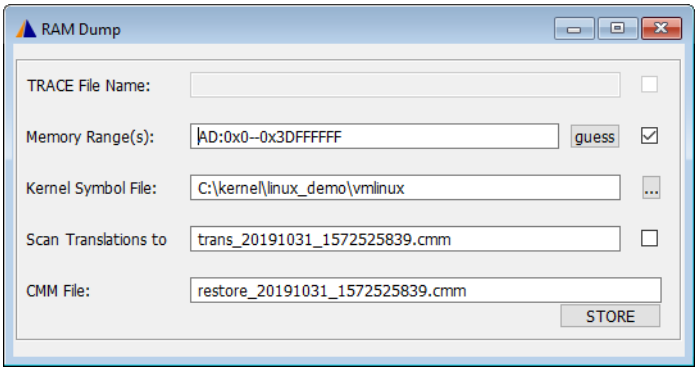


- **Export**: open a dialog for exporting the kernel log to an external file. The dialog allows to select the file format (ASCII or XHTML) and the log levels and facilities that should be included in the exported file. The dialog is based on the script `dmesg.cmm` available in the path of the Linux awareness.



RAM Dump Generation

The Linux awareness offers a dialog to generate a snap shot of the current system state for a later analysis using the TRACE32 instruction set simulator. This dialog can be opened from the menu **Linux > Generate RAM Dump** and is based on the script `ramdump.cmm` available in the TRACE32 demo directory under `~~/demo/<arch>/kernel/linux`.



After pushing the STORE button, the dialog will save the RAM contents as well as important register values and will generate a `restore_<...>.cmm` script that can be used to restore the system state on the TRACE32 instruction set simulator.

Most of the errors in Linux aware debugging are due to a wrong symbol information or to an incorrect setup of the debugger address translation.

The loaded `vmlinux` file must match the kernel binary executed on the target. To verify if this is the case, you can perform the following steps:

- Load the `vmlinux` file to the debugger virtual memory (VM:) using the following command.

```
Data.LOAD.Elf vmlinux AVM:0
```

- Display the Linux banner string from the debugger VM or print it to the area window:

```
Data AVM:linux_banner
PRINT Data.STRING(AVM:linux_banner)
```

- Compare the Linux banner string with the output of the Linux command `cat /proc/version`. Both strings must be identical including the timestamps.

Moreover, you need to make sure that the kernel was configured with `CONFIG_DEBUG_INFO` enabled and with `CONFIG_DEBUG_INFO_REDUCED` **not** set.

The next point to check in case you are having trouble is if the debugger address translation is correctly set. Problems due to an incorrect setup of the debugger address translation especially show up when debugging kernel modules or debugging in the user-space. You need to check the following:

- Is the MMU Format set with the **MMU.FORMAT** command correct?
- Is the kernel logical address translation correct? To check this translation, you can use the command **MMU.List.PageTable** address with the kernel logical start address as parameter when the kernel has already booted e.g.

```
MMU.List PageTable 0xC0000000
```

If you are still having trouble, please select the TRAC32 menu **Help > Support > Systeminfo...**, store your system information to a file and send this file together with your setup scripts as well as the content of the **TASK.TEST** window to support@lauterbach.com.

FAQ

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