Training Linux Debugging

Release 09.2023

Training Linux Debugging

Training Linux Debugging

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**

- **Linux Trace**

- **Troubleshooting**

Please note that Linux debugging for Intel x86/x64 is covered by a different training document, refer to **"Training Linux Debugging for Intel® x86/x64"** (training_rtos_linux_x86.pdf).

Documentation Updates

The latest version of this document is available for download from:
www.lauterbach.com/pdf/training_rtos_linux.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 32bit kernel, the split between kernel and user space is at a certain barrier which is configurable in the kernel. Three options are available:

- 3G/1G user/kernel i.e. the kernel space starts at `0xC0000000` (`CONFIG_VMSPLIT_3G`)

- 2G/2G user/kernel i.e. the kernel space starts at `0x80000000` (`CONFIG_VMSPLIT_2G`)

- 1G/3G user/kernel i.e. the kernel space starts at `0x40000000` (`CONFIG_VMSPLIT_1G`)
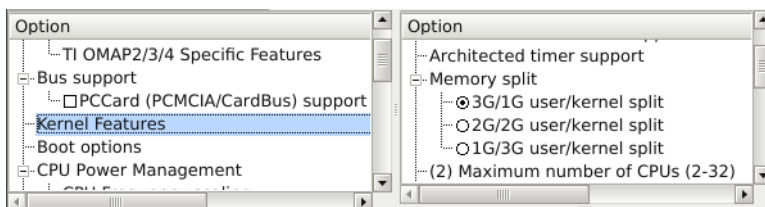
For Arm 32 bit processors, the kernel additionally reserves the 16 MB below the start of the kernel for kernel modules. The kernel space starts for instance from `0xBF000000` if a 3G/1G user/kernel split is used.

For 64 bit kernels, user and kernel space are separated by a reserved virtual address range.

Debugger Memory Access

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 generally pointed by a special register (e.g. TTBR0/TTBR1 for Arm). 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 command **TASK.List.SPACES**. In order to access this information, the debugger needs to have access to the kernel data at any time. The Linux debugging script has thus to inform the debugger about the logical to physical address translation used by the kernel.

> **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 by TRACE32 Linux Menu"**, page 36 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
```
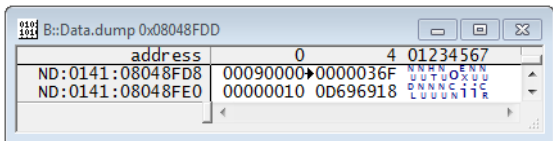

virtual address of current process 0x141
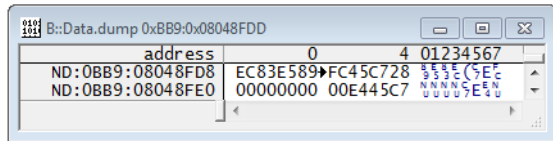
virtual address of specified process 0xBB9
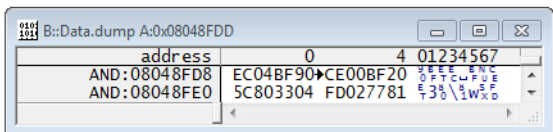
access to physical address A:0x8048FDD
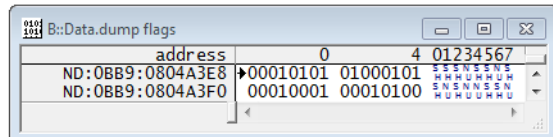
Symbol "flags" with process 0xBB9

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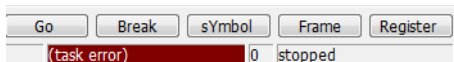
You can ignore the "task error" message as long as the kernel has not yet booted. In case you still get this error after the kernel boot, then you probably have a wrong configuration or a problem with the kernel debug symbols.

On Demand Paging

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 processor 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 and creates a translation in the current page table.

The following screen shots show an example of on-demand paging on an Arm Cortex-A processor: the code of the process consists of multiple memory pages. The program counter is at the end of the page (address `0x18FF8`). The next page (at address `0x19000`) is not yet loaded into physical memory. Therefore the debugger cannot read the memory at these addresses and shows question marks.
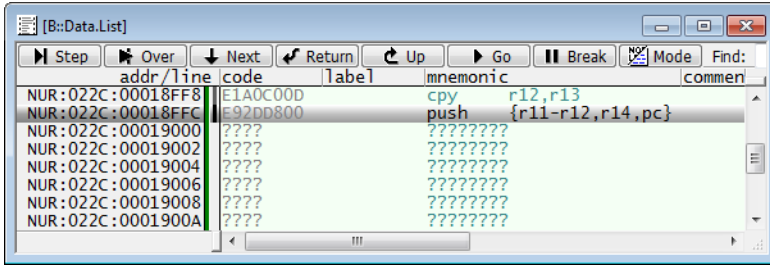
After a single step, a `PABORT` exception takes place (address `0xFFFF000C`).
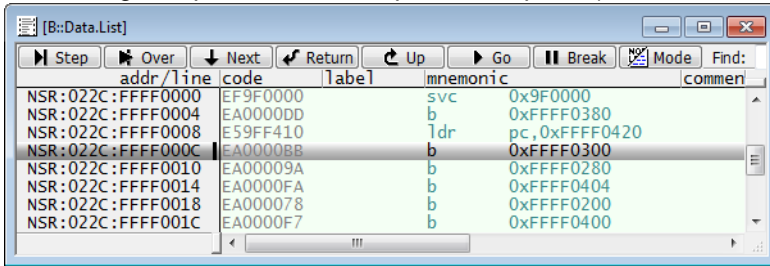
The kernel handler for pre-fetch abort is called then, which loads and maps the page.
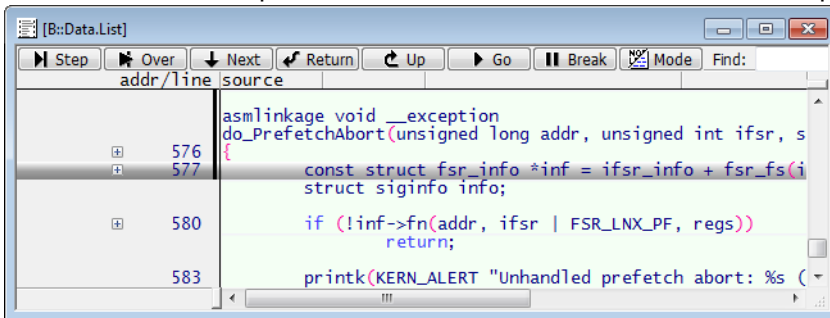
Finally the execution returns to the address which caused the exception (`0x19000`).
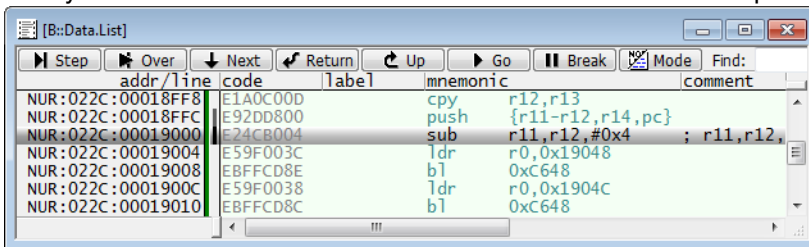
Please also refer to **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf) for more information about on-demand paging.

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

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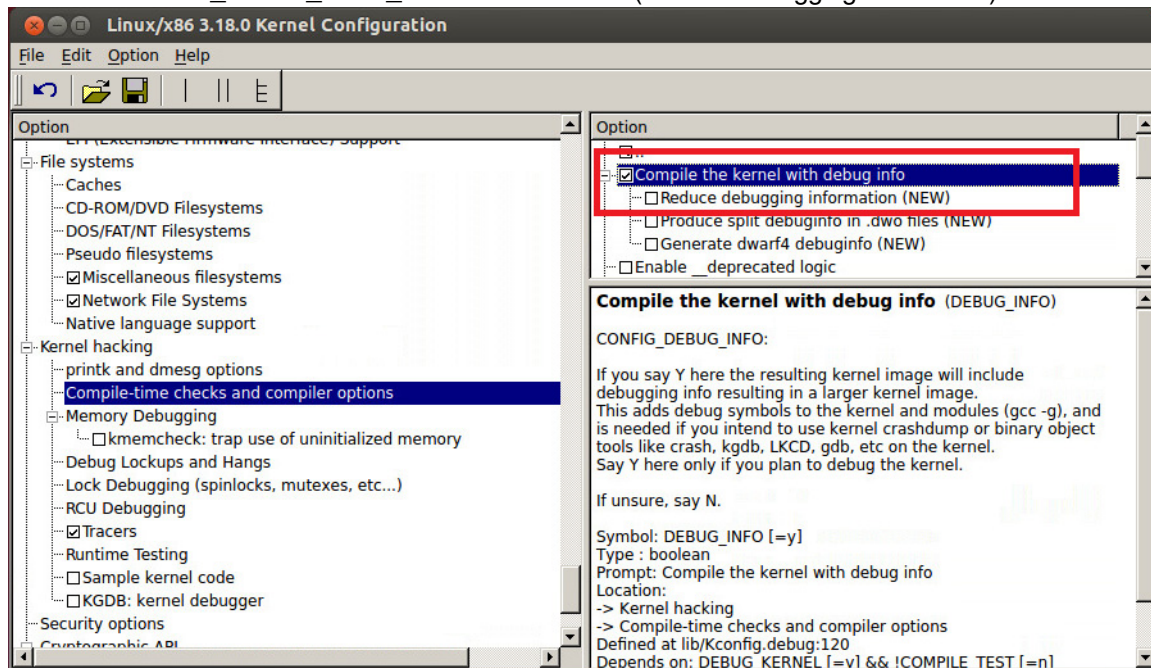
Moreover the option "Produce split debug info in .dwo files" (`CONFIG_DEBUG_INFO_SPLIT`) has to be disabled.

```
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.

Disable KPTI a.k.a. KAISER (Arm 64 bit only)

If the option `CONFIG_UNMAP_KERNEL_AT_EL0` is enabled, the kernel is unmapped when running in user space and mapped back in on exception entry via a trampoline page in the vector table. Thus, when the execution is stopped in user space, the debugger won't have access to the kernel data. Please disable this option in the kernel configuration if possible. Otherwise, you can add `"pti=0"` to the kernel boot parameters or set the variable `__kpti_forced` to a negative value e.g.

```
IF sYmbol.EXIST(__kpti_forced)
   Var.set __kpti_forced = -1
```

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.
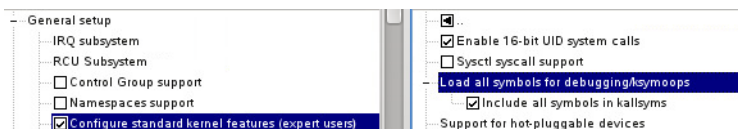
Extracting the Kernel Configuration

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/<arch>/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
```

Debugger Setup

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 your **Processor Architecture Manual**.

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**:800080000).

```
SYStem.Option.MMUSPACES ON      ; enable space IDs
```

It is recommended to add the command at the start of the Linux debugging script after resetting the debugger environment and before establishing the debug connection.

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

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.

Architecture Specific Options

Additional settings are needed depending on the used target processor. Please refer to your **Processor Architecture Manual** and to the Linux debugging demo scripts for more information.

The following settings are e.g. needed for Armv7 Cortex-A processors:

```
SYStem.Option.DACR ON          ; give debugger global write permissions
TrOnchip.Set DABORT OFF        ; used by Linux for data page misses!
TrOnchip.Set PABORT OFF        ; used by Linux for program page misses!
TrOnchip.Set UNDEF  OFF        ; might be used by Linux for FPU detection
```

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 Kernel Symbols

Kernel symbols are very important when debugging a Linux system. Without kernel symbols, no Linux aware debugging is possible. You need to load the kernel symbols even if you only debug user applications and do not debug the kernel code.

The kernel debug information is included in the file `vmlinux`. This file has to be compiled with debugging information enabled as already explained. You can load the kernel debug symbols using the following command:

```
Data.LOAD.Elf vmlinux /NoCODE          ; load only kernel debug symbols
```

The option **/NOCODE** should be used to only load the symbols without kernel code.

The symbols of the `vmlinux` file contain empty structure definitions (forward declarations in the source files). These may confuse the Linux awareness. To remove those structure definitions, execute a **sYmbol.CLEANUP** right after loading the symbols into the debugger.

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:
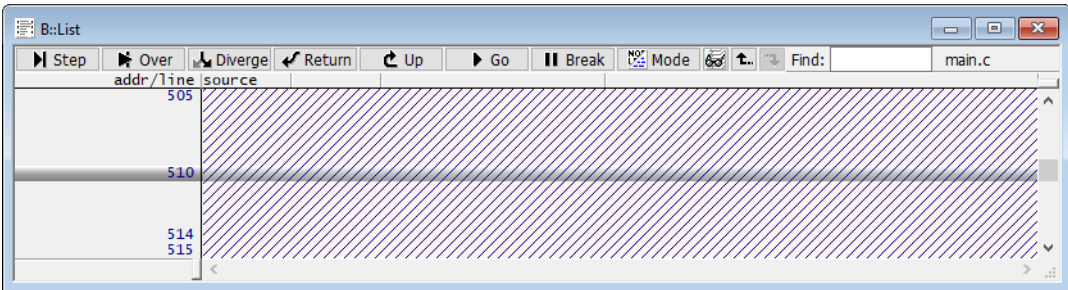
The easiest way to inform the debugger about the path of the source file is to do a right mouse click in the hatched area then select **Resolve Path**. A file search dialog will appear.
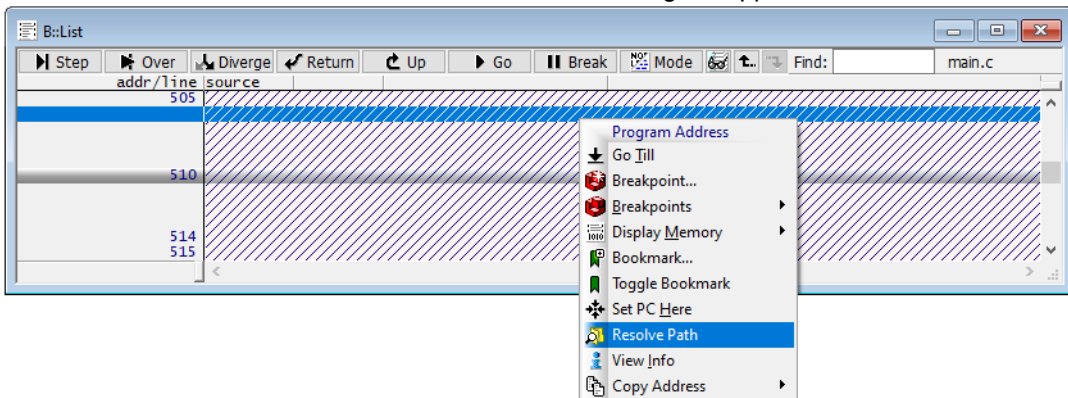
You just need then to browse to the source code file. The result of **Resolve Path** is a source path translation which will be used to locate all kernel source code files. This means that you have to resolve the path of a single source code file and all other kernel sources will be automatically found by TRACE32.

You can see the result of **Resolve Path** using the command **sYmbol.SourcePATH.List**.
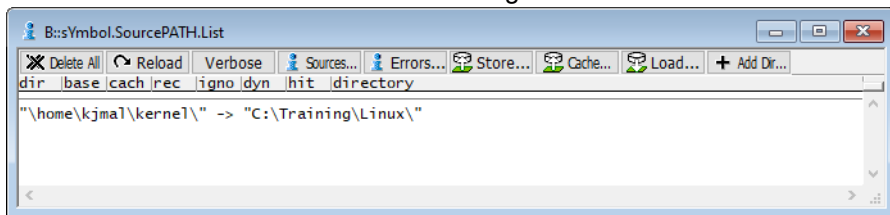
Using the button **Store...**, the resulting **sYmbol.SourcePATH.Translate** command can be saved in a PRACTICE script.

```
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 an SD card to the RAM. However, you can also use the debugger to download the kernel to the target memory over JTAG. In this case you 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:<physical_start>-<logical_start> /NosYmbol
```

Since the `vmlinux` file is mapped to logical addresses, it has to be loaded with an offset which is equal the kernel physical start minus the kernel logical start. If we have for instance a 32 bit kernel starting at the logical address `0xC0000000` and that should be downloaded to the memory at `0x10000000`, we should the use the following command:

```
Data.LOAD.Elf vmlinux A:0x10000000-0xC0000000 /NosYmbol
```

Some architectures (e.g. SH and MIPS) use the same virtual and physical start address for the kernel. In this case you can simply download the kernel code and load the symbols using:

```
Data.LOAD.Elf vmlinux
```

> **When shifting the kernel image from virtual to physical start address (e.g 0x10000000-0xC0000000) you have to only load the kernel code without debug symbols using the option /NoSymbol. Otherwise, the kernel debug symbols will be loaded on shifted addresses. The debug symbols should be then loaded separately.**

Sometime, it is also necessary to restrict the download to a RAM area range due to wrong information by some gcc versions in the Elf file:

```
Data.LOAD.Elf vmlinux 0x10000000-0xC0000000 0x10000000--0x1FFFFFFF
                                             /NosYmbol
```

Downloading the Kernel Code at the Kernel Entry

You can set an on-chip breakpoint at the kernel entry point 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 already been prepared by the boot-loader. For example:

```
Go 0x10008000 /Onchip
WAIT !STATE.RUN()

Data.LOAD.Elf vmlinux A:0x10000000-0xC0000000 /NosYmbol
```

If the kernel image is compressed, the breakpoint at the kernel entry is hit twice: the first time to execute the decompression routine and the second the start the kernel. You need to take this in consideration in your script e.g. using the **/COUNT** option:

```
Go 0x10008000 /Onchip /COUNT 2
WAIT !STATE.RUN()

Data.LOAD.Elf vmlinux A:0x10000000-0xC0000000 /NosYmbol
```

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. The program counter should be set to kernel entry point. Other registers should be set depending on the target architecture. For the Arm 32 bit Cortex-A architecture, the register R2 should point to a device tree blob.

```
; Reset all registers:
Register.RESet
; set PC on start address of image:
Register.Set PC 0x80008000
; Set R2 to the start of the device tree blob
Register.Set R2 0x82000000
; Download the device tree blob
Data.LOAD.Binary omap4-panda.dtb 0x82000000 /NoClear
```

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 0x81600000 /NoClear /NosYmbol
```

Please note that the **/NoClear** option should be used here, otherwise already loaded debug symbols would be cleared.

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 LINUX swapper_pg_dir 0xc0000000--0xc1ffffff 0x80000000
```

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`.

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
```

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 Arm Cortex-A kernel, the common range starts at `0xbf000000`, `0x7f000000` or `0x3f000000` respectively for 3G/1G, 2G/2G, 1G/3G user/kernel split and ends at `0xffffffff` (since kernel modules are mapped in the 16 MB below the start of the kernel).

```
; possible common ranges for Arm 32 bit kernels:
TRANSlation.COMMON 0xbf000000--0xffffffff ;3G/1G user/kernel split
TRANSlation.COMMON 0x7f000000--0xffffffff ;2G/2G user/kernel split
TRANSlation.COMMON 0x3f000000--0xffffffff ;1G/3G user/kernel split
```

The common range for a 32 bit kernel starts otherwise at `0xc0000000`, `0x80000000` or `0x40000000` respectively for 3G/1G, 2G/2G, 1G/3G user/kernel split and ends at `0xffffffff`.

```
; possible common ranges for non Arm 32 bit kernels:
TRANSlation.COMMON 0xc0000000--0xffffffff ;3G/1G user/kernel split
TRANSlation.COMMON 0x80000000--0xffffffff ;2G/2G user/kernel split
TRANSlation.COMMON 0x40000000--0xffffffff ;1G/3G user/kernel split
```

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
```

Enable The Address Translation

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.

Lauterbach provides two scripts (for Arm 32 and 64 bit Cortex-A processors) that try to detect the debugger address translation relevant settings and print them to the AREA window. These scripts can be found in `~~/demo/arm/kernel/linux/board/generic-template`

You need then to copy the detected setting from the AREA window into your Linux debugging script.
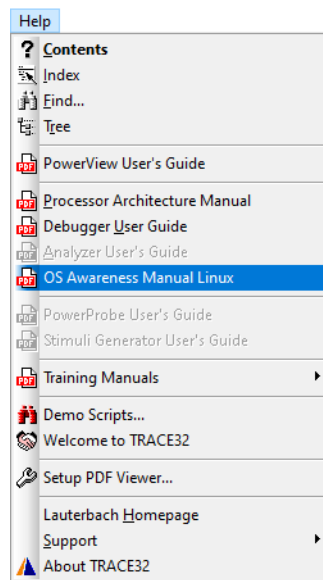
Set up the Linux Awareness

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/<arch>/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/<arch>/kernel/linux/awareness/`

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

The Linux menu file includes many useful menu items developed for the TRACE32 PowerView user interface to ease Linux debugging.
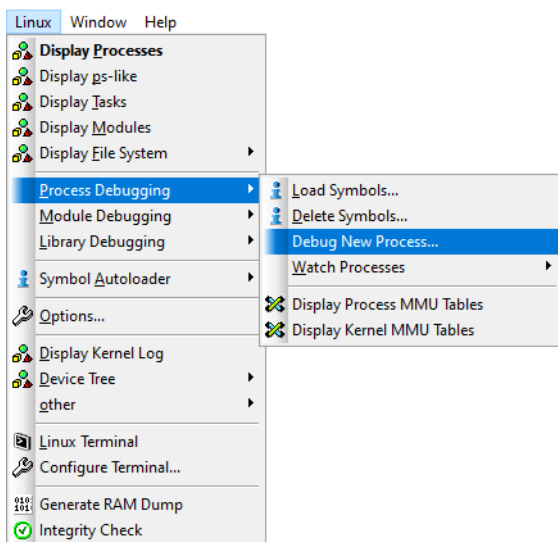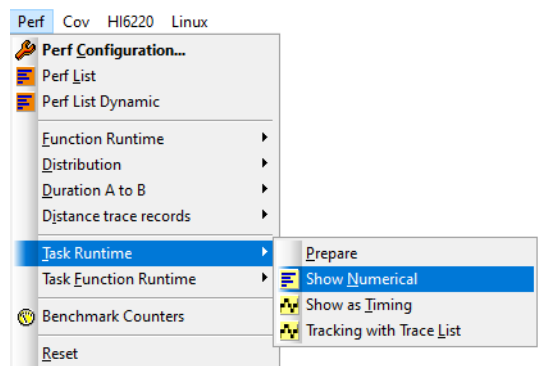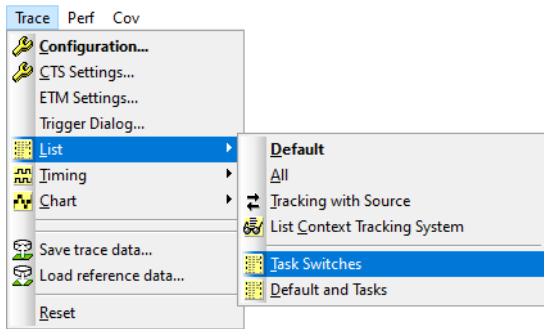
The Linux awareness and Linux menu are based on scripts available under:
`~~/demo/<arch>/kernel/linux/awareness`.

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"**

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"      0x0000:0xC0000000--0xFFFFFFFF /RED
```

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/arm/kernel/linux/disable_watchdogs.cmm
```

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

Setup for SMP Linux

At kernel start, only the first core of an SMP system is generally accessible. If the debugger tries to access the other cores, there is often a fatal error. When debugging an SMP Linux from the start, the user has thus to connect only to the first core, wait until all core are activated by the kernel then re-connect to the whole SMP system. The **CORE.ASSIGN** command can be used here to assign a set of cores to the SMP system. This command can only be used in "system down" mode. Example:

```
SYStem.CPU IMX6QUAD           ; quad core CPU
CORE.ASSIGN 1.                ; only assign the first core
<...>                         ; further CPU specific settings
SYStem.Up                     ; reset the target and connect to first core

<...>                         ; load kernel symbols, set up the address
                              ; translation and load the Linux awareness

Go                            ; resume the execution, the kernel will boot
WAIT 10.s                     ; wait until other cores are activated by the
                              ; kernel e.g. 10.s

SYStem.Down                   ; detach from the target (core assignment
                              ; can only be changed in "down" mode)
CORE.ASSIGN 1. 2. 3. 4.       ; assign all four core
SYStem.Mode Attach            ; re-attach to the target
; the OS awareness needs to be re-loaded since it is disabled by the
; the SYStem.Down command
TASK.CONFIG ~~/demo/arm/kernel/linux/awareness/linux.t32
```

You may use, instead of a wait time (10 seconds in the example above), a breakpoint on a kernel function that is executed just after all cores have been enabled by the kernel. The kernel function `smp_cpus_done` can be used here for instance:

```
; the script smp.cmm will be executed when the breakpoint is hit and the
; breakpoint will be disabled
Break.Set smp_cpus_done /CMD "DO smp.cmm" /DISableHIT
Go
```

```
; script smp.cmm
SYStem.Down
CORE.ASSIGN 1. 2. 3. 4.
SYStem.Mode Attach
TASK.CONFIG ~~/demo/arm/kernel/linux/awareness/linux.t32
```

NOTE:	Special care needs to be taken when debugging the boot of an SMP kernel. If you change the core assignment too early, thus before the cores are enabled by the kernel, you will most probably get a debug access error. If you however stop the program execution with only one core assigned after the kernel has enabled other cores, the kernel will panic.

Example Linux Setup-Scripts

You can find demo startup scripts for different target boards in the TRACE32 installation directory under `~~/demo/<arch>/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 sets up Linux aware debugging for a kernel running on an OMAP4430 processor with two Cortex-A9 cores. In this example the kernel is already running on both Cortex-A9 cores. The RAM is located at 0x80000000.

```
REset
WinCLEAR

SYStem.CPU OMAP4430
SYStem.Option.DACR ON          ; give Debugger global write permissions
TrOnchip.Set DABORT OFF        ; used by Linux for page miss!
TrOnchip.Set PABORT OFF        ; used by Linux for page miss!
TrOnchip.Set UNDEF OFF         ; may be used by Linux for FPU detection
SYStem.Option.MMUSPACES ON ; enable space IDs to virtual addresses

SYStem.Mode 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

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

; Set up the debugger address translation
MMU.FORMAT LINUXSWAP3 swapper_pg_dir 0xC0000000--0xDFFFFFFF 0x80000000
TRANSLATION.COMMON 0xBF000000--0xFFFFFFFF
TRANSLATION.TableWalk ON
TRANSlation.ON

; Load Linux awareness and Linux menu
PRINT "initializing multi task support..."
TASK.CONFIG ~~/demo/arm/kernel/linux/awareness/linux.t32
MENU.ReProgram ~~/demo/arm/kernel/linux/awareness/linux.men

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

ENDDO
```

The second example shows a Linux script for the Zynq UltraScale+ APU which has four Cortex-A53 cores. We reset here the target and connect to the first core using the **SYStem.Up** command. We resume then the program execution till the entry point of the kernel and download the kernel binary to the memory.

```
REset
WinCLEAR

SYStem.CPU ZYNQ-ULTRASCALE+-APU
SYStem.Option.MMUSPACES ON
CORE.ASSIGN 1.
SYStem.Up

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

SETUP.IMASKASM ON            ; lock interrupts while single stepping

Break.Set 0x80000 /Onchip  ; set a breakpoint at the kernel entry
Go
TERM.TRIGGER "ZynqMP>"
WAIT TERM.TRIGGERED(D:0)
TERM.OUT "bootm 0x20000000" 10.
WAIT !STATE.RUN()
Break.Delete

; Load the Linux kernel
Data.LOAD.Elf vmlinux A:0x80000-0xFFFFFF8008080000 /NoSymbol

; Load the kernel symbols
Data.LOAD.Elf vmlinux /NoCODE

; set up the source path translation
sYmbol.SourcePATH.Translate "\home\kjmal\kernel\" "C:\Training\Linux\"

; set up the debugger address translation
MMU.FORMAT LINUXSWAP3 swapper_pg_dir _text--(_end-1) 0x80000
TRANSLATION.COMMON 0xf000000000000000--0xffffffffffffffff
TRANSLATION.TableWalk ON
TRANSlation.ON

Break.Set smp_cpus_done /CMD "DO smp.cmm" /DISableHIT
Go

; Initialize Linux awareness
TASK.CONFIG ~~/demo/arm/kernel/linux/awareness/linux.t32
; loads Linux menu:
MENU.RePProgram ~~/demo/arm/kernel/linux/awareness/linux.men

ENDDO
```

Debugging 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.
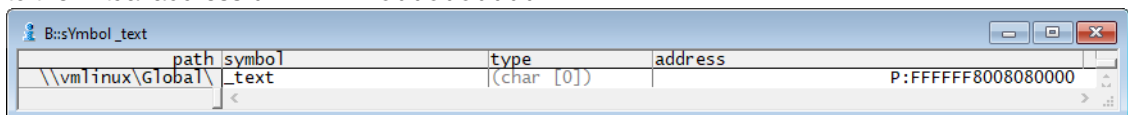
The Kernel

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. In the following screen shot, the program counter points to the kernel entry point on a Armv8 Cortex-A processor where the RAM starts at address `0x0`.
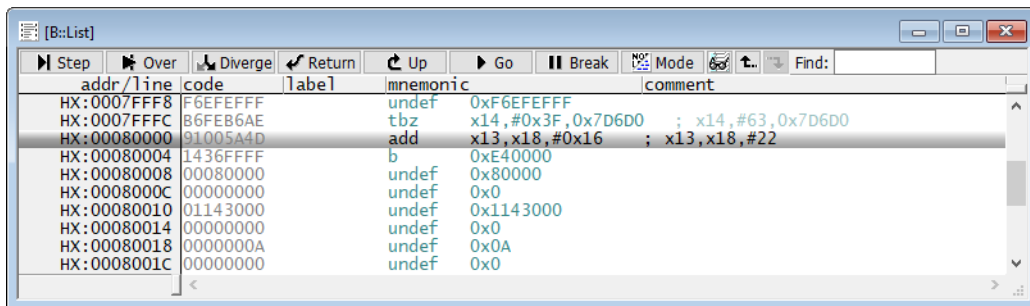
The corresponding debug symbol is the label `_text` which is mapped in this example by the `vmlinux` file to the virtual address `0xFFFFFF8008080000`.
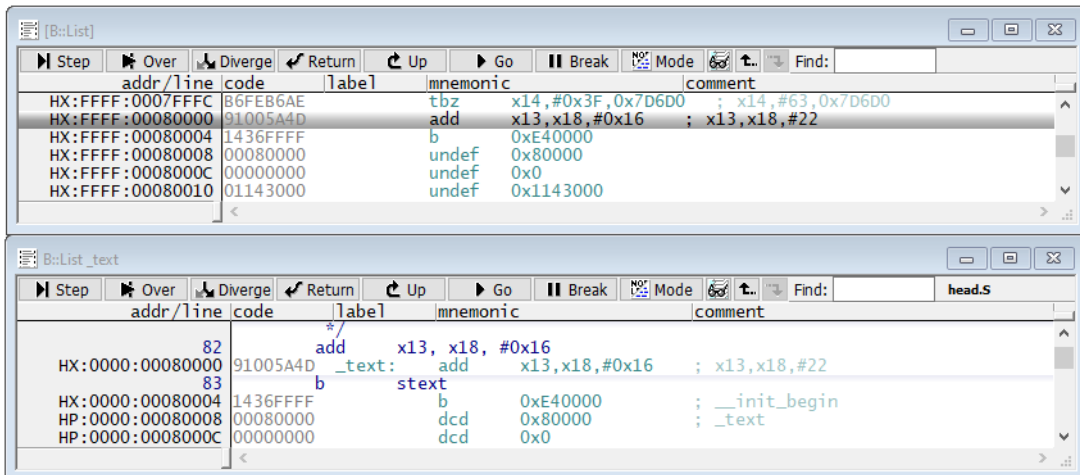

To debug the kernel startup code, we have thus to load the `vmlinux` file with an offset which is equal to the physical kernel start address minus the kernel virtual address.

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

This corresponds in our example to `0x80000-0xFFFFFF8008080000`:

```
Data.LOAD.Elf vmlinux 0x80000-0xFFFFFF8008080000 /NoCODE
```

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.

```
[B::List]                                                              ☐ ▣ ✕
 ▶I Step   ▶ Over  ⅃ Diverge  ✔ Return   ↻ Up    ▶ Go   II Break  Mode  ⌖ t..  Find:
       addr/line code       label        mnemonic              comment
 HX:FFFF:0007FFFC B6FEB6AE             tbz      x14,#0x3F,0x7D6D0  ; x14,#63,0x7D6D0
 HX:FFFF:00080000 91005A4D             add      x13,x18,#0x16      ; x13,x18,#22
 HX:FFFF:00080004 1436FFFF             b        0xE40000
 HX:FFFF:00080008 00080000             undef    0x80000
 HX:FFFF:0008000C 00000000             undef    0x0
 HX:FFFF:00080010 01143000             undef    0x1143000
```

```
B::List_text                                                          ☐ ▣ ✕
 ▶I Step   ▶ Over  ⅃ Diverge  ✔ Return   ↻ Up    ▶ Go   II Break  Mode  ⌖ t..  Find:          head.S
       addr/line code       label        mnemonic              comment
                */
              82          add      x13, x18, #0x16
 HX:0000:00080000 91005A4D _text:     add      x13,x18,#0x16      ; x13,x18,#22
              83          b        stext
 HX:0000:00080004 1436FFFF          b        0xE40000           ; __init_begin
 HP:0000:00080008 00080000          dcd      0x80000            ; _text
 HP:0000:0008000C 00000000          dcd      0x0
```

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 an alternative, you may extend the common range and keep the address extension enabled. You have however to undo this change when the kernel switches to virtual address space.

```
TRANSlation.COMMON 0x0--0xffffffffffffffff
```

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. In order to use on-chip breakpoints during kernel boot, it may be necessary to edit the kernel configuration and re-compile the kernel. For the Arm architecture, you have to disable the following kernel configuration:

```
# CONFIG_HAVE_HW_BREAKPOINT is not set
```

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

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

Kernel Boot

After enabling the MMU, the kernel switches to virtual address space. The kernel symbols have to be loaded without any offset.

```
Data.LOAD.Elf vmlinux /NoCODE           ; load the kernel symbols
```

As explained previously, you need to set up the debugger address translation and load the Linux awareness. As the whole kernel code is already loaded into the memory, you can use software breakpoints on kernel functions which are safer in this case since on-chip breakpoints may be deleted by the kernel during boot.

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 to have a `uImage` loaded by the boot-loader (e.g. from a memory card) and a `vmlinux` file on the host which is not from the same build as the `uImage` file. 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:
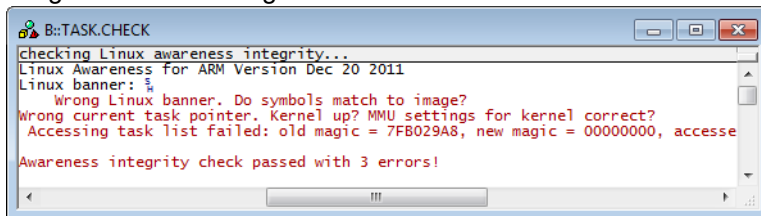

Please note that **TASK.CHECK** command only does a basic check based on the `linux_banner` string. In some cases, this basic check cannot detect that there is a mismatch between the kernel code and the loaded kernel debug symbols. Please refer to **"Troubleshooting"**, page 55 for more details.

SMP Debugging

An SMP kernel starts booting only using the first core. During boot, the kernel enables the other cores. When the debugger tries to access cores that have not yet been enabled by the kernel, a fatal errors could occur. You need in this case to assign only the first core for debugging. As soon as the other cores have been enabled by the kernel, the core assignment has to be changed. If you stop the program execution while the debugger is attached to only a part of the running cores, the kernel will most probably panic.

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.
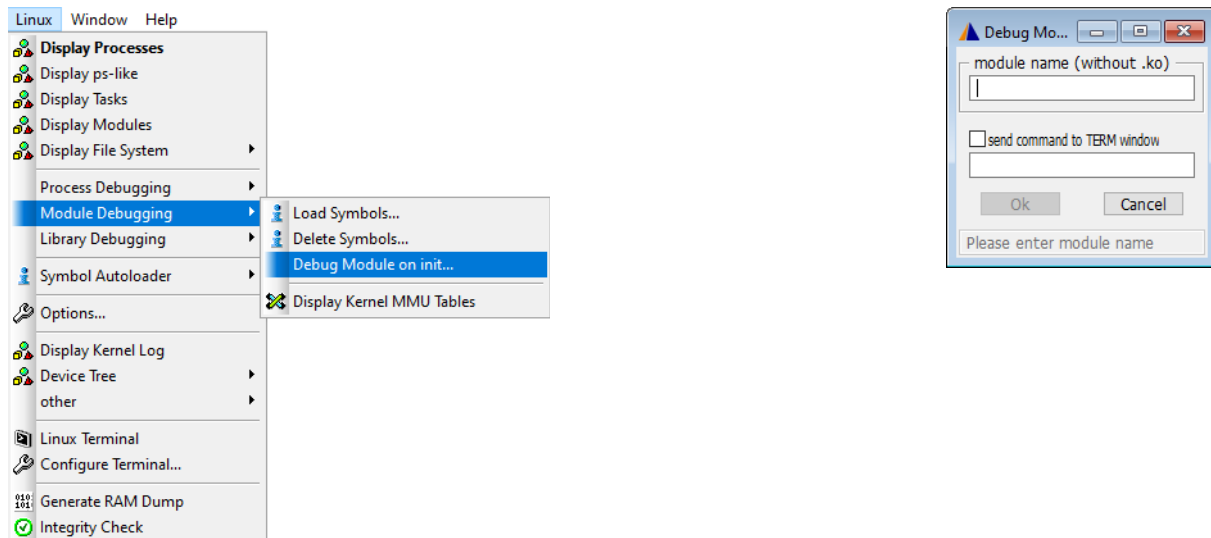
The **"Debug Module on init..."** menu point is based on the script `mod_debug.cmm` available in the path of the Linux awareness. The script sets a breakpoint at a kernel function that is executed when a new kernel module is loaded. As soon as the breakpoint is hit, the TRACE32 Symbol Autoloader will 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/arm/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/arm/kernel/linux/awareness/mod_debug.cmm mymod
```

The script additionally accepts the following arguments:

- `/term "<command>"`: send the command *<command>* to the TRACE32 terminal window in order to load the module e.g. `/term "insmod mymod.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.

> **NOTE:** **Remember that the kernel modules are part of the kernel address range and should be covered by TRANSlation.COMMON.**

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 "demomod"          ; 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 "demomod"        ; erase obsolete module symbols
```

Processes

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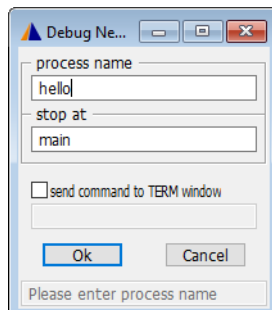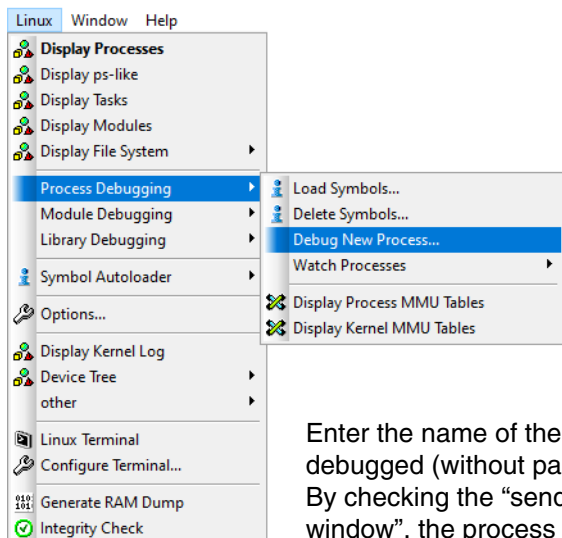
Enter the name of the process to be debugged (without parameters).
By checking the "send command to TERM window", the process will be started from the TERM window.

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/arm/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/arm/kernel/linux/awareness/app_debug.cmm hello
```

The script additionally accepts the following arguments:

- `/term "<command>"`: send the command *<command>* to the TRACE32 terminal window in order to start the process e.g. `/term "/home/user/t32/hello"`

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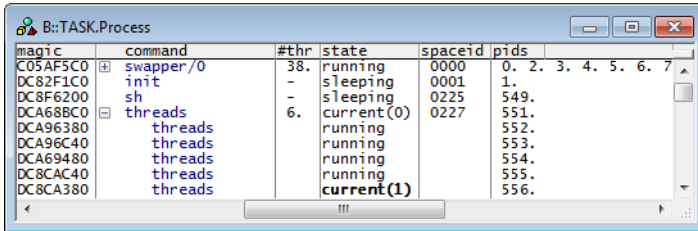
After the process exists, its debug symbols have to be deleted using the menu **Linux > Process Debugging > Delete Symbols...** or the command **TASK.sYmbol.Delete**

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

Further features of the TRACE32 Linux awareness are shown in chapter "Linux Specific Windows", page 45.

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 they are loaded when loading the process 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 Linux menu **Library Debugging > Load Symbols...**
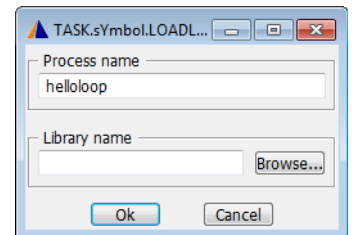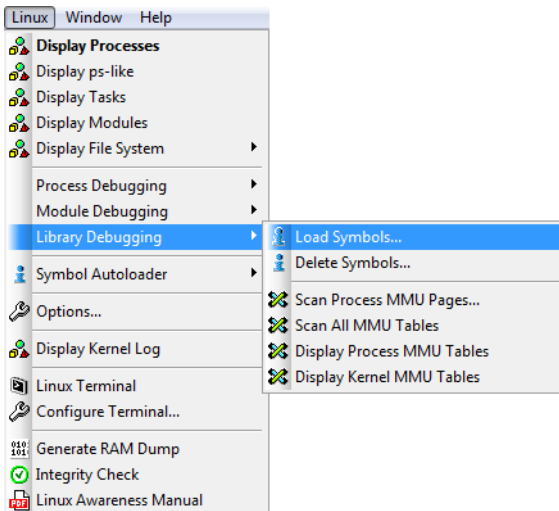
This menu point is based on the TRACE32 command **TASK.sYmbol.LOADLib**.

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

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"
```

If task related breakpoints are not supported by the core, the debugger will always stop on the breakpoint address and resume executing if the current task is not the specified one.

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 single stepping 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. Internally, the Linux awareness will retrieve the register values of the selected tasks from the kernel data structures.

```
List     /TASK "mytask"
Register /TASK "mytask"
Frame    /TASK "mytask"
```
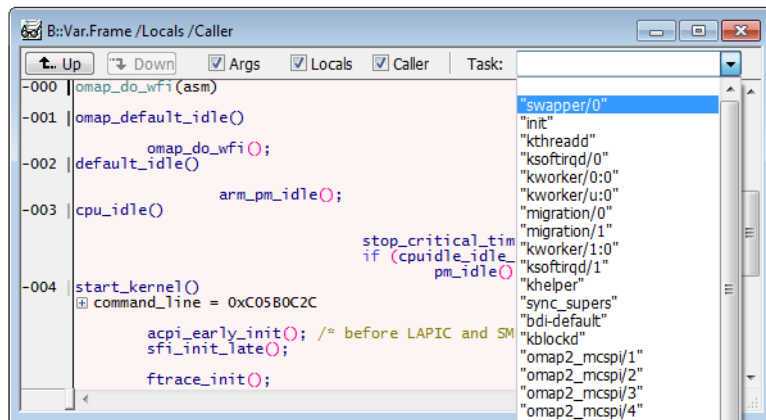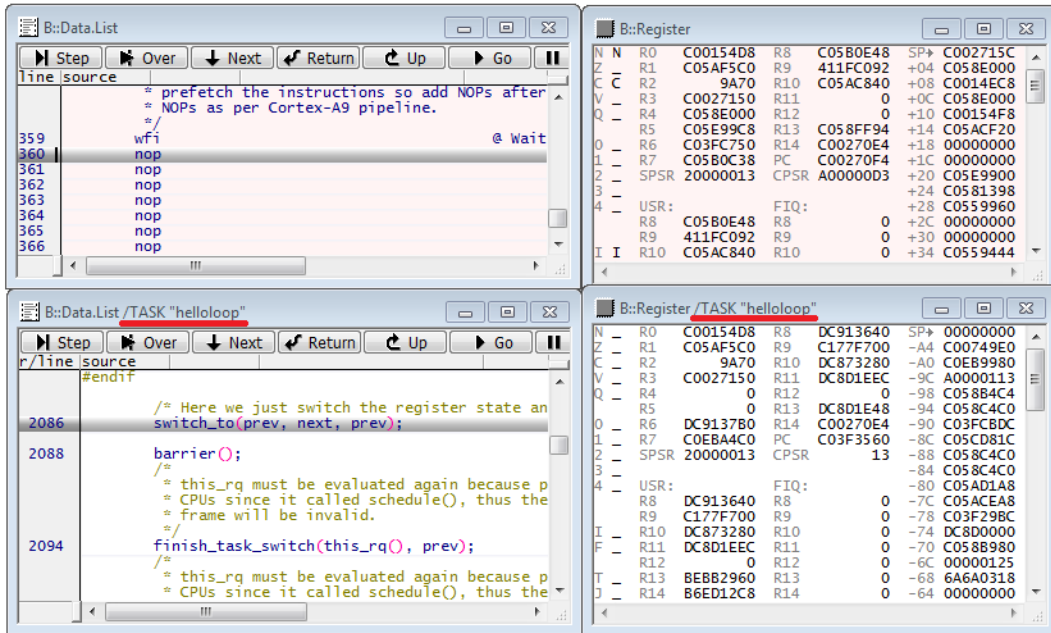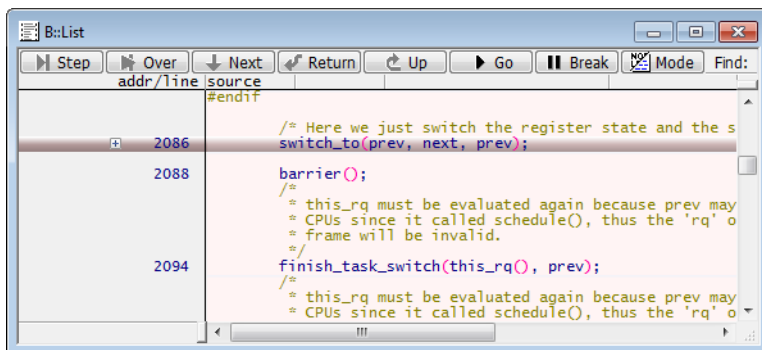
You can additionally "virtually" switch the context also from the **TASK.DTask** window by popup menu-item **"Switch Context"**:
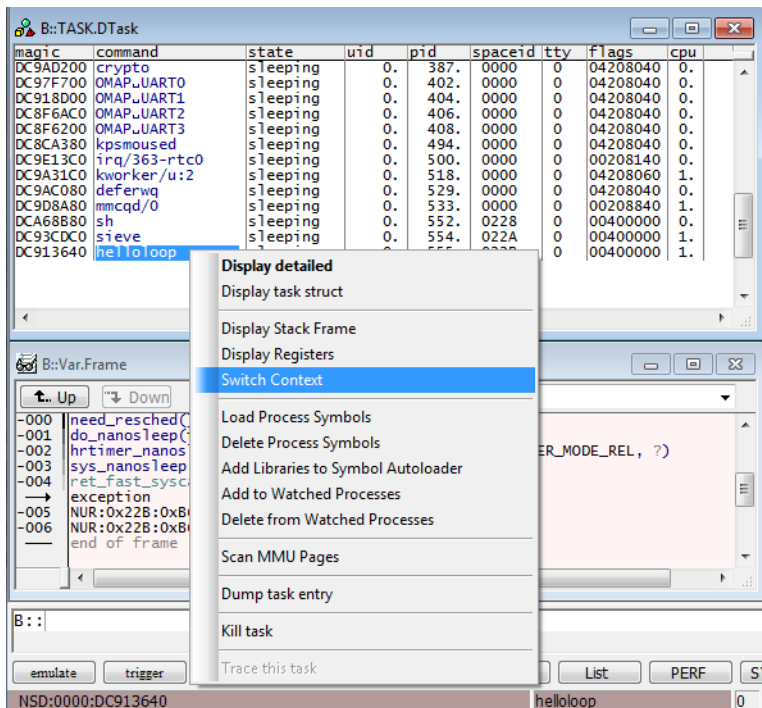
Switch to the helloloop task.

It's not the current PC from the target ("main", process "helloloop") but the PC where the task "helloloop" will be continued!

Care for the grey buttons.
After the context switch you get the full wanted info. But it's not the current processor state.

There is a pseudo PC bar (light red) showing the PC where process "helloloop" will be continued.

Linux Specific Windows

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.

Displaying the Task List

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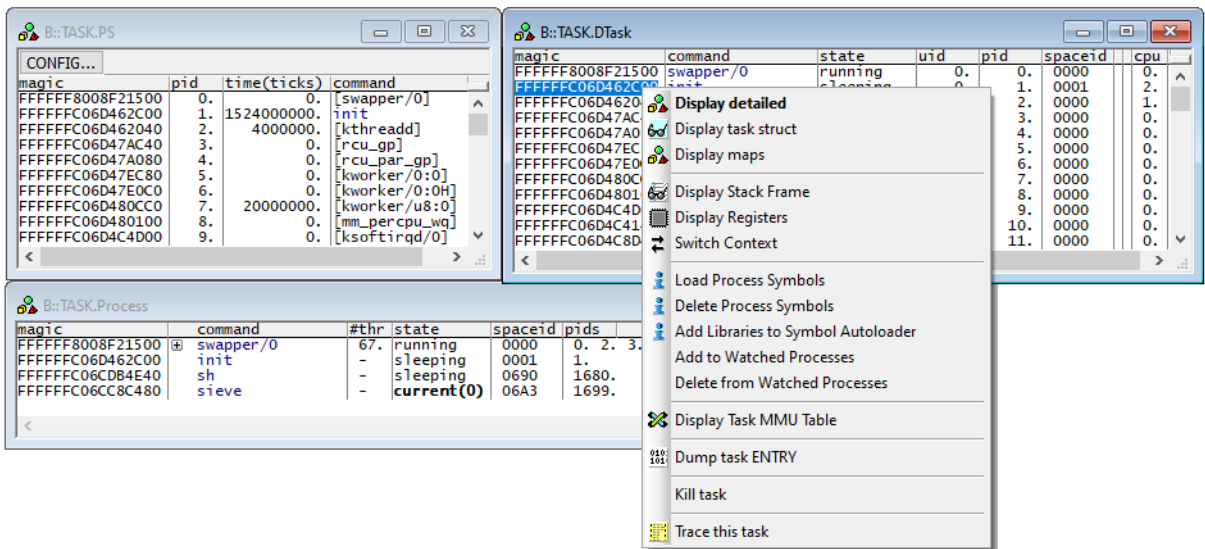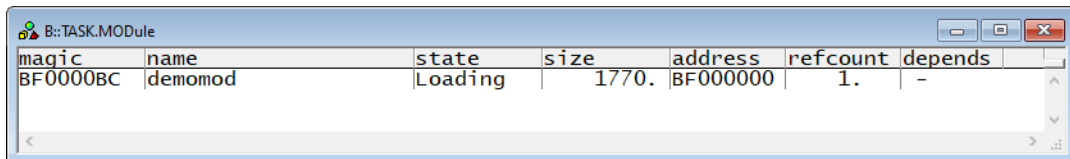
By doing a right mouse click on the task magic in these three views, you get a pull-down menu with the following options for the selected task:

- **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.
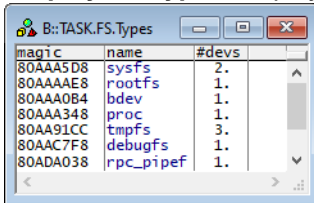

By doing a right mouse click on the module's magic, you get a pull down menu with the following options:

- **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.
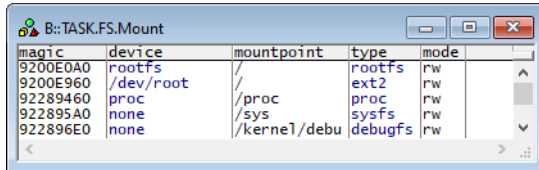
File System Information

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.
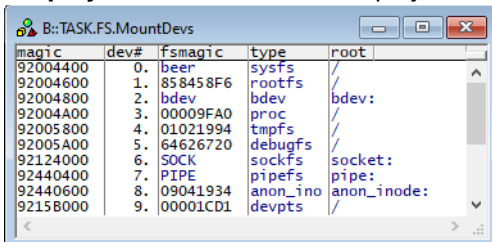

- **Display Mount Points**: display the current mount points.
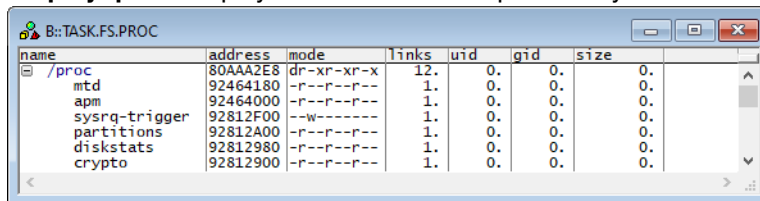
- **Display Mounted Devices**: display all currently mounted devices (i.e.super blocks).
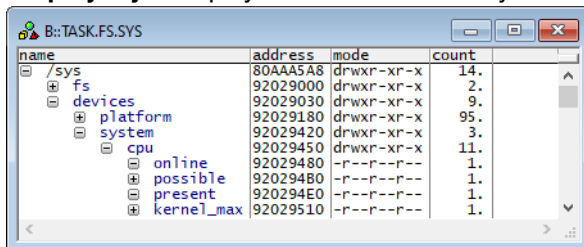
- **Display /proc**: display the content of the /proc file system.
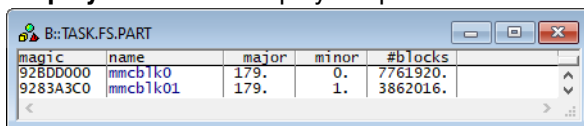
- **Display /sys**: display the content of the /sys file system.

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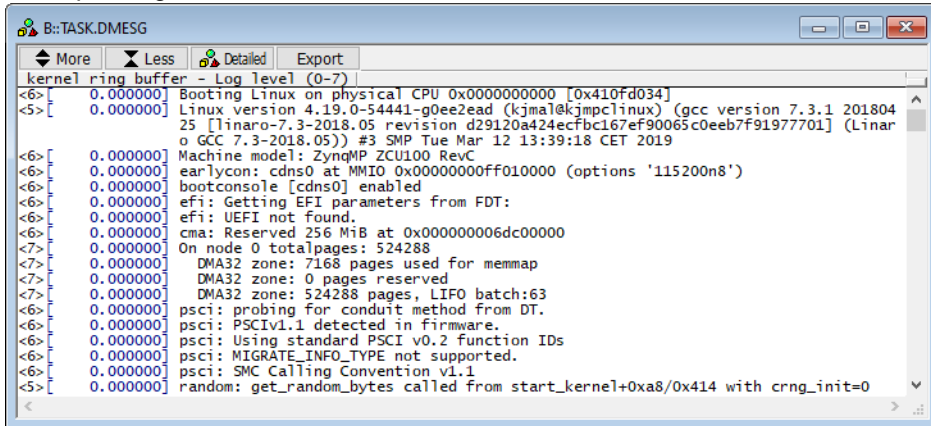
The **TASK.DMESG** window includes the following buttons:

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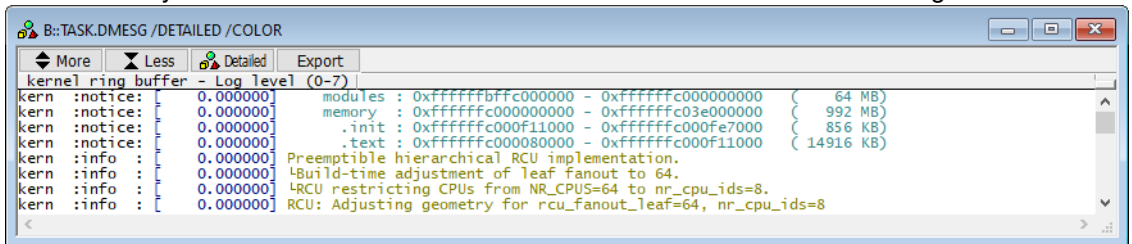
- **Export**: open a dialog for exporting the kernel log to an external file. The dialog allows to select the file format (ASCII or XHTML) and the log levels and facilities that should be included in the exported file. The dialog is based on the script `dmesg.cmm` available in the path of the Linux awareness.
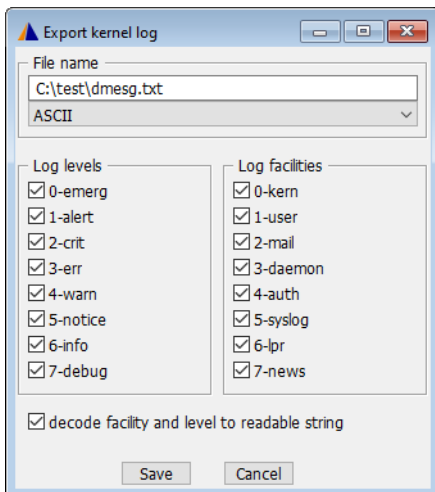
Device Tree

You can display the device tree as blob (tree view) or source by selecting the menu **Linux > Device Tree** then **Display Device Tree** or **Display Source**. The corresponding Linux awareness commands are **TASK.DTB** and **TASK.DTS**. The menu additionally offers the possibility to extract the device tree as blob or source file.

```
B::TASK.DTB
Device Tree Blob
DTB version: 17

Tree structure:
/
    compatible          = "hisilicon,hi6220-hikey" "hisil
    interrupt-parent    = <0x00000001>
    #address-cells      = <0x00000002>
    #size-cells         = <0x00000002>
    model               = "HiKey Development Board"
    psci
    cpus
        #address-cells      = <0x00000002>
        #size-cells         = <0x00000000>
        cpu-map
```

```
B::TASK.DTS
/dts-v1/;

/ {
    compatible = "hisilicon,hi6220-hikey", "hisilicon,hi6220";
    interrupt-parent = <0x1>;
    #address-cells = <0x2>;
    #size-cells = <0x2>;
    model = "HiKey Development Board";

    psci {
        compatible = "arm,psci-0.2";
        method = "smc";
    };

    cpus {
```
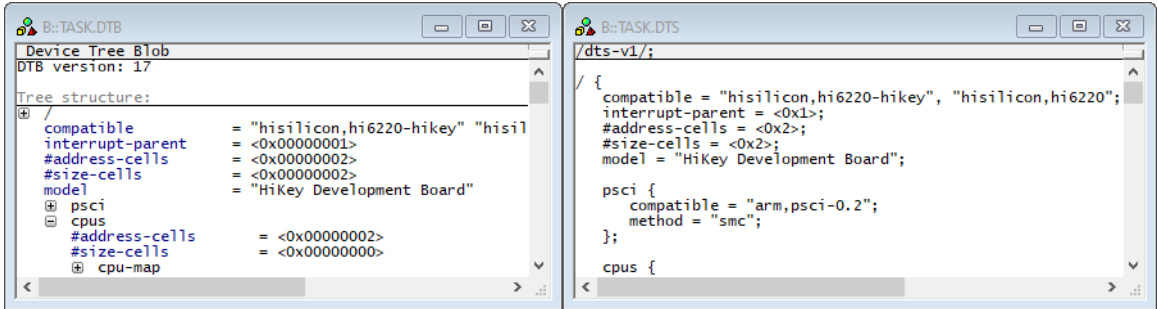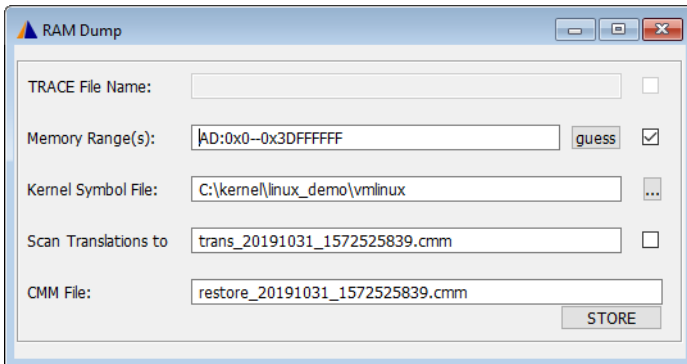
RAM Dump Generation

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`.

```
RAM Dump

TRACE File Name:        [                              ]  ☐

Memory Range(s):        [AD:0x0--0x3DFFFFFF    ]  guess   ☑

Kernel Symbol File:     [C:\kernel\linux_demo\vmlinux  ]  ...

Scan Translations to    [trans_20191031_1572525839.cmm ]  ☐

CMM File:               [restore_20191031_1572525839.cmm]

                                              STORE
```

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.

Linux Trace

Please note that this chapter does not contain information about general trace configuration and usage. It only points to the needed settings and conditions to achieve task aware trace for a target Linux system.

Overview

When tracing a system with virtual memory management where a single virtual address can correspond to different physical addresses, the trace tool (which gets from the on-chip trace module only the virtual address of the executed code) needs to know the current memory space for every trace time. The trace tools needs thus to get the task switching information in the trace. This information is only necessary for task run-time analysis.
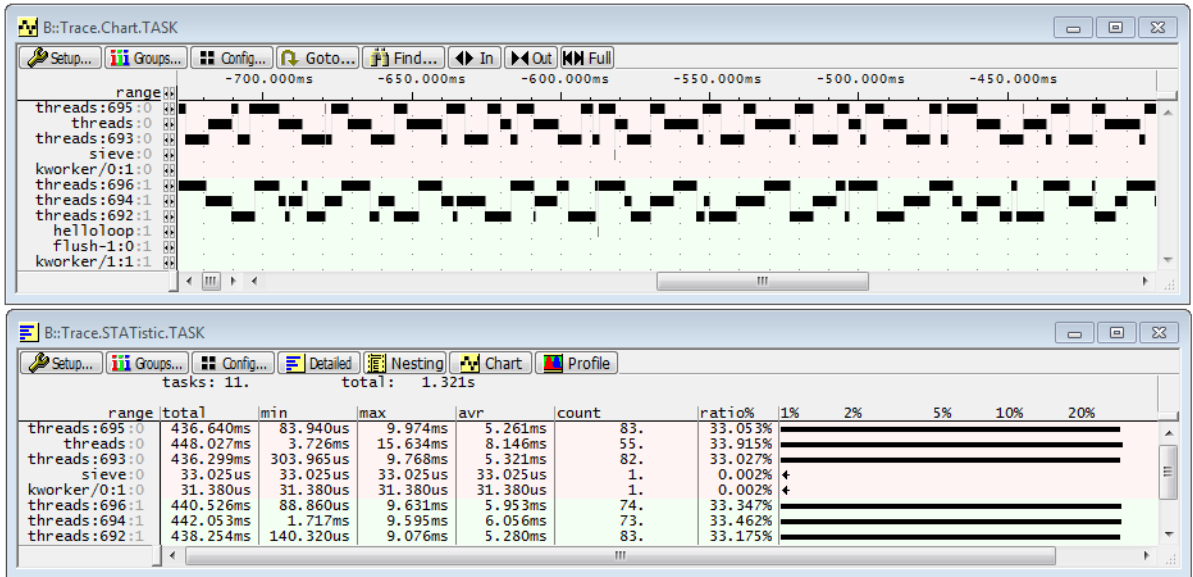
If a data trace is available, the debugger can trace the write accesses to the memory location which contains information about the current Linux task for each core using **TraceData** or **TraceEnable** breakpoints. This addresses is delivered for the current core by the **TASK.CONFIG(magic)** Linux Awareness function.

```
; Example: exporting task switches via data trace and all instruction
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

If however no data trace is available, the debugger needs then a different trace mechanism to be aware of the context switches in the kernel. This is e.g. possible with the ETM Context ID trace for the Arm architecture or the Ownership Trace Messages for the Nexus trace.

Context ID Trace for Arm Cortex-A

Most Arm Cortex-A processors do not have a data trace support. For such processors, the debugger uses the Context ID trace messages for task aware trace. The CONTEXTIDR register have to be written by the kernel on every task switch.

Kernel versions older than 3.6 need to be patched to support tracing all Linux tasks.

For kernels 3.6 and newer, you just need to enable **Kernel hacking > Write the current PID to the CONTEXTIDR register** (`CONFIG_PID_IN_CONTEXTIDR`) in the kernel configuration:
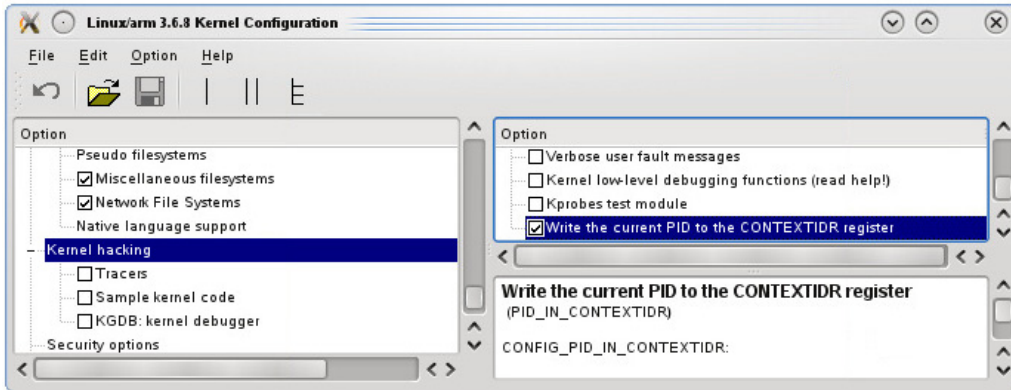
To enable Context ID trace **ETM.ContextID 32** has additionally to be set in TRACE32.

OTM Trace for PowerArchitecture based QorIQ Processors

PowerArchitecture based QorIQ processors have limited data trace capabilities that cannot be used to trace multiple addresses at the same time. Data trace can thus only be used for single processor systems to trace task switches as write accesses to a single address have to be traced in this case.

For SMP systems, Ownership Trace Messages have to be used instead. In fact, the Linux kernel of the NXP SDK writes on a task switch an identifier of the new task in the **PID** register (SPR48):

```
mtspr    SPRN_PID,r3
```

Depending in the processor, the transmitted PID value is in this case up to 14 bit. As an alternative, the kernel can be patched to use the **NPIDR** register (SPR517) that provides a 32bit value.

The command **NEXUS.OTM PID0** (or **NEXUS.OTM NPIDR** in case NPIDR is used) has then to be used to enable the generation of Ownership Trace Messages on write accesses to this register. Please refer for more information to **"QorIQ Debugger and NEXUS Trace"** (debugger_ppcqoriq.pdf).

Using the LOGGER for Task Switch Trace

Some processors do not have any on-chip trace support like the PowerPC P2020. We will write in this example for an SMP Linux running on the P2020 a kernel module that uses the kernel tracepoints and registers a probe function "`logger_sched_switch`" on the kernel scheduler tracer. This probe function is called on every task switch and writes the trace data in special TRACE32 format to a buffer. After stopping the debugger reads the buffer and displays the task switches with timing information

Kernel module logger.ko.

```
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/tracepoint.h>
#include <trace/events/sched.h>
#include <linux/smp.h>

#define T32_LOGGER_SIZE 1024

volatile int T32_TraceEnable = 0; // will be set by the debugger
volatile int T32_Magic[2] = {0, 0}; // will be set by the debugger

typedef struct {
    unsigned long tshigh;/* high part of timestamp and cycle info */
    unsigned long tslow; /* low part of timestamp */
    unsigned long address;
    unsigned long data;
} T32_loggerData;

struct {
    T32_loggerData * ptr; /* pointer to trace data */
    long size; /* size of trace buffer */
    volatile long index;  /* current write pointer */
    long tindex; /* index of trigger record */
    long iflags; /* incoming flags, Bit 0: ARM, Bit 8: Stack Mode */
    long oflags; /* out. flags, Bit 0: Overflow, Bit 8: Trigger, Bit 9: Break */
    long reserved1;
    long reserved2;
    T32_loggerData buffer[T32_LOGGER_SIZE];
} T32_LoggerStruct;

static void T32_LoggerInit(void)
{
    T32_LoggerStruct.ptr = T32_LoggerStruct.buffer;
    T32_LoggerStruct.size = T32_LOGGER_SIZE;
}

unsigned long GetTBL(void)
{
    unsigned long tb;
    asm volatile ("mftb %0": "=r" (tb));
    return tb;
}
```

```
void T32_LoggerDataFunc(int cycle, void* addr, unsigned long data, int core)
{
    int             index;
    if (!(T32_LoggerStruct.iflags & 0x01))
        return;

    if (T32_LoggerStruct.index >= T32_LoggerStruct.size) {
        if (T32_LoggerStruct.iflags & 0x100)
            return;
        T32_LoggerStruct.oflags |= 0x01;
        T32_LoggerStruct.index = 0;
    }

    index = T32_LoggerStruct.index++;
    T32_LoggerStruct.ptr[index].tslow = GetTBL();
    T32_LoggerStruct.ptr[index].tshigh = (cycle << 24) | (core << 16);
    T32_LoggerStruct.ptr[index].address = (unsigned long) addr;
    T32_LoggerStruct.ptr[index].data = data;
    T32_LoggerStruct.index = index + 1;
}

void T32_LoggerTrigger(void)
{
    if (T32_LoggerStruct.oflags & 0x100)
        return;
    T32_LoggerStruct.tindex = T32_LoggerStruct.index;
    T32_LoggerStruct.oflags |= 0x100;
}

static void logger_sched_switch(void *ignore, struct task_struct *prev,
                                struct task_struct *next)
{
    int cpu = smp_processor_id();
    if (!T32_TraceEnable)
        return;

    T32_LoggerDataFunc(0x34, (void *)T32_Magic[cpu], (unsigned long)next, cpu);
}

static int __init logger_init(void)
{
    T32_LoggerInit();
    return register_trace_sched_switch(logger_sched_switch, NULL);
}
static void __exit logger_exit(void)
{
    unregister_trace_sched_switch(logger_sched_switch, NULL);
}

module_init(logger_init)
module_exit(logger_exit)

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Khaled Jmal");
MODULE_DESCRIPTION("Logger trace module");
```

The following scripts then insert the module, enable and configure the trace:

```
IF !STATE.RUN()
  Go
TERM.OUT "insmod logger.ko" 10. ; insert the logger module
WAIT 1.s
Break
TASK.SYMBOL.LOADMod "logger" ; load the module symbols
MMU.SCAN ALL
LOGGER.ADDRESS T32_LoggerStruct
LOGGER.Mode E ON
Trace.METHOD LOGGER ; select logger as default trace method
LOGGER.TimeStamp UP
LOGGER.Init
CORE.SELECT 1
&magic=task.config(magic)
Var T32_Magic[1]=&magic
CORE.SELECT 0
&magic=task.config(magic)
Var T32_Magic[0]=&magic
LOGGER
Var T32_TraceEnable=1 ; enable the trace
```

We can then display the exact task switches on both cores using the **Trace.Chart.TASK** command:
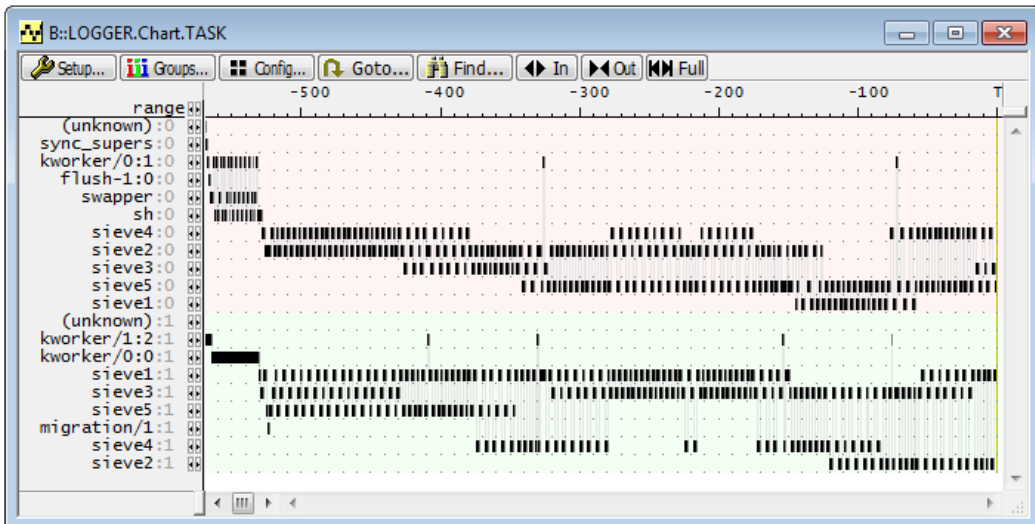
Troubleshooting

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.