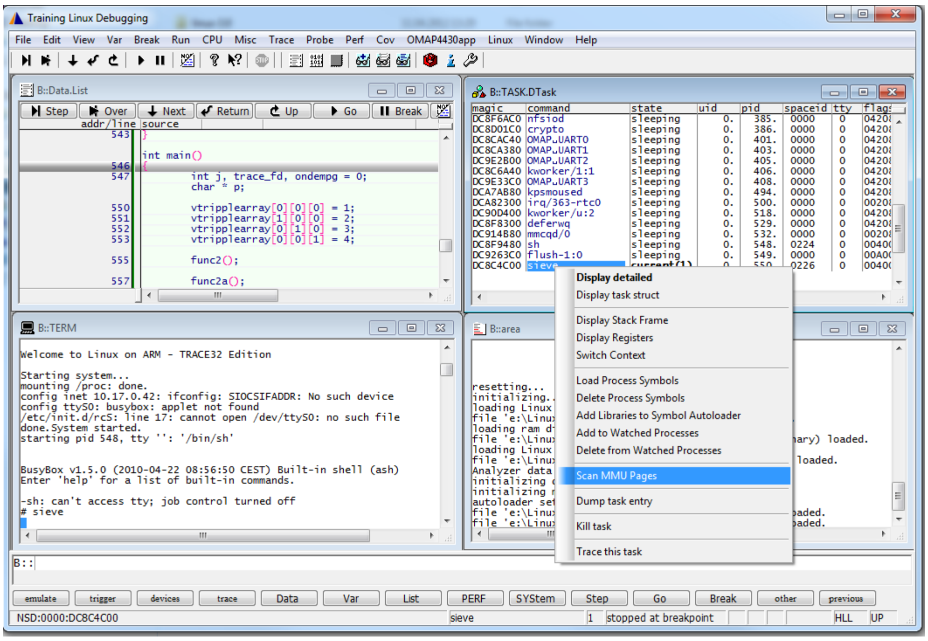
Training Linux Debugging

Training Linux Debugging

Version August, 23 2012


Training Linux Debugging
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov OMAP4430app Linux Window Help
B::Data.List
Step Over Next Return Up Go Break
B::TASK.DTask
B::TERM
B::area
Display detailed
Display task struct
Display Stack Frame
Display Registers
Switch Context
Load Process Symbols
Delete Process Symbols
Add Libraries to Symbol Autoloader
Add to Watched Processes
Delete from Watched Processes
Scan MMU Pages
Dump task entry
Kill task
Trace this task
B::
emulate trigger devices trace Data Var List PERF SYStem Step Go Break other previous
NSD:0000:DC8C4C00 sieve 1 stopped at breakpoint HLL UP

```
     addr/line source
           543 }

               int main()
           546 {
           547         int j, trace_fd, ondempg = 0;
                       char * p;

           550         vtripplearray[0][0][0] = 1;
           551         vtripplearray[1][0][0] = 2;
           552         vtripplearray[0][1][0] = 3;
           553         vtripplearray[0][0][1] = 4;

           555         func2();

           557         func2a();
```

```
magic    command       state     uid   pid    spaceid tty flags
DC8F6AC0 nfsiod        sleeping   0.   385.   0000    0   0420
DC8D01C0 crypto        sleeping   0.   386.   0000    0   0420
DC8CAC40 OMAP␣UART0    sleeping   0.   401.   0000    0   0420
DC8CA380 OMAP␣UART1    sleeping   0.   403.   0000    0   0420
DC9E2B00 OMAP␣UART2    sleeping   0.   405.   0000    0   0420
DC8C6A40 kworker/1:1   sleeping   0.   406.   0000    0   0420
DC9E33C0 OMAP␣UART3    sleeping   0.   408.   0000    0   0420
DCA7AB80 kpsmoused     sleeping   0.   494.   0000    0   0420
DCA82300 irq/363-rtc0  sleeping   0.   500.   0000    0   0020
DC90D400 kworker/u:2   sleeping   0.   518.   0000    0   0420
DC8F8300 deferwq       sleeping   0.   529.   0000    0   0420
DC914B80 mmcqd/0       sleeping   0.   532.   0000    0   0020
DC8F9480 sh            sleeping   0.   548.   0224    0   0040
DC9263C0 flush-1:0     sleeping   0.   549.   0000    0   00A0
DC8C4C00 sieve         current(1) 0.   550.   0226    0   0040
```

```
Welcome to Linux on ARM - TRACE32 Edition

Starting system...
mounting /proc: done.
config inet 10.17.0.42: ifconfig: SIOCSIFADDR: No such device
config ttyS0: busybox: applet not found
/etc/init.d/rcS: line 17: cannot open /dev/ttyS0: no such file
done.System started.
starting pid 548, tty '': '/bin/sh'


BusyBox v1.5.0 (2010-04-22 08:56:50 CEST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

-sh: can't access tty; job control turned off
# sieve
```

```
resetting...
initializing..
loading Linux
file 'e:\Linux
loading ram d
file 'e:\Linux                                   ary) loaded.
loading Linux
file 'e:\Linux                                   loaded.
Analyzer data
initializing
initializing
autoloader se
file 'e:\Linux                                   aded.
file 'e:\Linux                                   aded.
```

Prolog

This training will have the main subjects:

- **Basic terms on embedded Linux**
- **Building the kernel, a kernel module and a simple demo application**
- **Setting up a script for Linux-aware debugging**
- **Debugging Linux components by TRACE32 Linux menu**

Further TRACE32 documents related to Linux Debugging:

This is a quick-tutorial. It will **not** introduce you to all features of the Linux Awareness. For a complete description, refer to the **”RTOS Debugger for Linux - Stop Mode”** (rtos_linux_stop.pdf) and to the **”RTOS Debugger for Linux - Run Mode”** (rtos_linux_run.pdf).

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**

1.) Linux Components

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

1. Bootloader(s)
2. The Linux Kernel
3. Kernel Modules
4. Processes and Threads
5. Libraries (Shared Objects)

Moreover, we can talk about two different spaces of executed code: kernel space with privileged rights which includes the kernel and kernel modules and user space with limited rights which includes processes, threads and libraries.

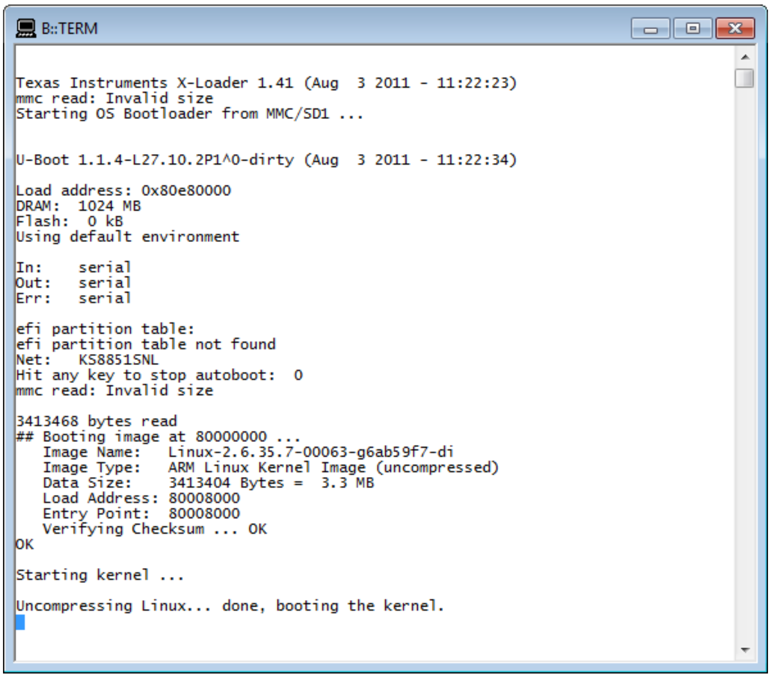
a) Bootloader

The bootloader is executed at the reset vector or is loaded by the Boot ROM code. It is not a part of Linux but a necessary pre-step that makes loading and starting the kernel possible. Its basic functionality consists of initializing the processor and the memory, loading the operating system from somewhere (e.g. over the network or out of Flash) and preparing/extracting the loaded image. Finally the bootloader passes important parameters (boot information) to the kernel image.

A typical example of a bootloader is U-Boot which is available for a wide range of embedded architectures and is not limited to the use for Linux only.

Some systems require more than one bootloader. E.g. on the PandaBoard, the Boot ROM code loads a first “light weighted” bootloader called X-Loader from the memory card. The X-Loader initializes enough memory for U-Boot which is then loaded form the memory card. The U-Boot makes further initialization and loads the kernel.

Linux boot process on the PandaBoard:



```
Texas Instruments X-Loader 1.41 (Aug  3 2011 - 11:22:23)
mmc read: Invalid size
Starting OS Bootloader from MMC/SD1 ...


U-Boot 1.1.4-L27.10.2P1^0-dirty (Aug  3 2011 - 11:22:34)

Load address: 0x80e80000
DRAM:  1024 MB
Flash:  0 kB
Using default environment

In:    serial
Out:   serial
Err:   serial

efi partition table:
efi partition table not found
Net:   KS8851SNL
Hit any key to stop autoboot:  0
mmc read: Invalid size

3413468 bytes read
## Booting image at 80000000 ...
   Image Name:   Linux-2.6.35.7-00063-g6ab59f7-di
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3413404 Bytes =  3.3 MB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

b) 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. ELF) called “vmlinux”. You can also find the kernel in compressed binary format (zImage/uImage). You will see later in this training how to configure and compile the Linux kernel.

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.

c) 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 “insmod” and “rmmod”. Typically kernel modules contain code for device drivers, ISRs, file systems etc. Kernel modules run at kernel level with kernel privilege (supervisor).

d) 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.

e) 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 context and 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.

2.) The Linux Awareness

The debugger need to be “aware” of the operating system used on the target. In TRACE32, this is handled by a separated file than can be loaded additionally to the debugger software. A set of additional commands, options and displays will be then available and simplify the debugging of the operating system. **The Linux Awareness is based on the file “linux.t32” which is available on the installation DVD or on request!**

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

⚠	**Linux aware debugging is not possible if the kernel symbol information is not available or if doesn’t fit the executed kernel code.**

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.

3.) 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.

3.1.) MMU Support

The debugger needs to know how the MMU of the processor works to be able to translate the addresses by itself correctly at each state of the target.

a) MMU Tables

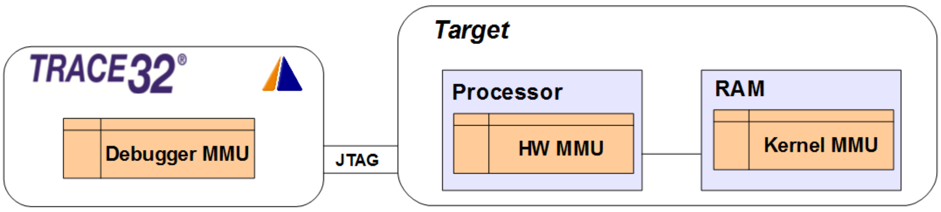
Linux operates completely in virtual memory space: all functions, variables, pointers etc. work with virtual addresses. Also, the symbols are bound to virtual addresses. However, the virtual address space doesn't cover all physical memory. **To gain access to the whole target, the debugger also needs to work with physical addresses.**

When accessing memory via the JTAG interface there can be hardware limitations. Some processor architectures (e.g. ARM) use virtual addresses, allowing easy access of current virtual addresses. If the user wants to access virtual addresses, that are currently not mapped in the hardware MMU (e.g. when accessing variables of another process), the debugger needs to execute a special handling to access this memory out of the MMU. Some other processor architectures (e.g. most PowerPC's) use always physical addresses when using JTAG. If you want to debug with virtual addresses (that's what you normally do), then the debugger must convert them first to physical addresses.

All this leads to the requirement, that the debugger needs to know the complete MMU translation tables. If this is the case, the debugger (and the user) can use any virtual or physical address, whatever is appropriate. With the use of the translation tables, the debugger then internally calculates the correct accessing method to the target.

The processor's MMU only holds the translation table for the current operation (e.g. for the current process plus kernel pages). Linux manages the switching of the MMU for different processes, so Linux holds several translation tables. The debugger needs to access or get a copy of all those tables.

Thus, we have **three** different MMU tables:



- **Hardware:** Usually held in the processor, this translation table holds all information necessary for the current operation (e.g. the current process). It will be extended at page faults and could completely change at a process switch. The hardware MMU usually consists of registers and/or dedicated memory areas.
- **Kernel:** The operating system holds a set of translation tables, that are necessary for the complete system run. The hardware MMU tables are usually (but not necessarily) a subset of these tables. Each process plus the kernel have an own translation table. The tables are stored in the target's memory.
- **Debugger:** The debugger needs al least a translation for kernel addresses. This should be created by the user when configuring TRACE32 for Linux Debugging. Moreover, the debugger can scan the kernel MMU tables and held a local copy of them. However, this is not needed if TableWalk is enabled. The debugger MMU translation tables are maintained and used only inside the debugger software, and don't affect the target at all. You can see the debugger MMU translation using the **TRANSlation.List** command.

It is very important, that those three MMU tables are not mixed up.

b) Space IDs

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

⚠	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. This command will fail, if any symbols are loaded, because symbols are bound to space ids.

After this command, a virtual address looks like "`001E:10001244`", which means virtual address 0x10001244 with space id 0x1E (process id = pid = 46.).

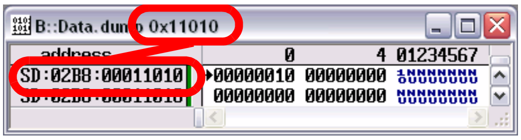
You can now access the complete memory:

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

Data.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 already have to specify, to which space id they should belong. See chapter “Debugging Linux Components” for details.

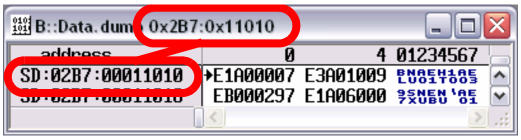
Because the symbols already contain the information of the space id, you don’t have to specify it manually.

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



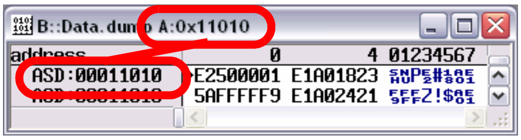
```
B::Data.dump 0x11010
address                 0        4 01234567
SD:02B8:00011010   00000010 00000000
                   00000000 00000000
```

virtual address of current process 0x02B8



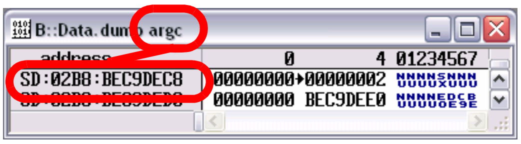
```
B::Data.dump 0x2B7:0x11010
address                 0        4 01234567
SD:02B7:00011010   E1A00007 E3A01009
                   EB000297 E1A06000
```

virtual address of specified process 0x02B7



```
B::Data.dump A:0x11010
address                 0        4 01234567
ASD:00011010       E2500001 E1A01823
                   5AFFFFF9 E1A02421
```

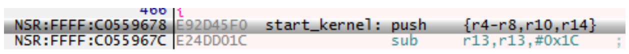
access to physical address A:0x11010



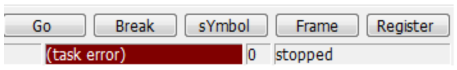
```
B::Data.dump argc
address                 0        4 01234567
SD:02B8:BEC9DEC8   00000000 00000002
                   00000000 BEC9DEE0
```

virt. addr., symbol “argc” belongs to pid 720.

If the Linux awareness is enabled, the debugger tries to get the current space id by accessing the kernel internal structures. If this fails e.g. because of wrong symbol information, an access error or because the kernel structures has not been yet initialized (e.g. if you stop at the start of the kernel), the debugger set the current space id to 0xFFFF and shows the message “task error” in the status line.



```
              466 {
NSR:FFFF:C0559678 E92D45F0  start_kernel: push   {r4-r8,r10,r14}
NSR:FFFF:C055967C E24DD01C                sub    r13,r13,#0x1C
```



```
(task error)  0  stopped
```

You can ignore the “task error” message as long as the kernel has not yet booted.

3.2.) Virtual Memory Management for Linux components

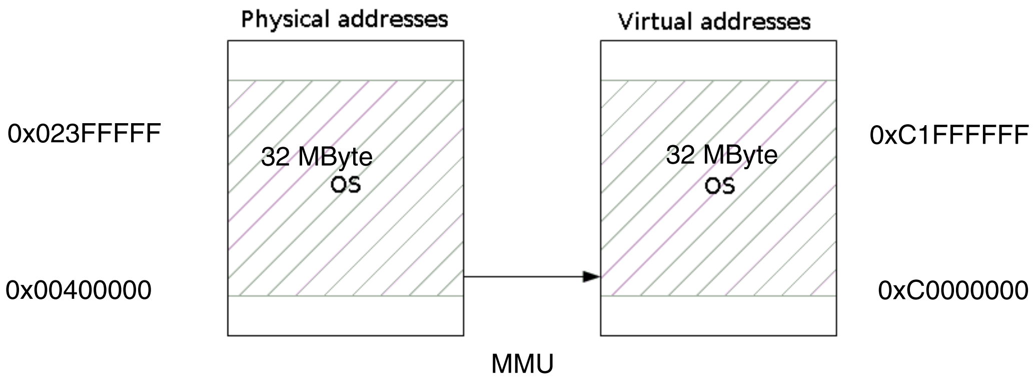
Linux uses several techniques for virtual memory management (VMM). The actual VMM used depends on the component running.

a) Bootloader

When coming out of reset, the processor runs with disabled MMU. Thus, the bootloader starts with no VMM used. Usually **the complete bootloader runs in physical address space**, never using any virtual addressing. The MMU may be used (on some processors e.g. some MPCs it can't be switched off), but without any address translation and page fault handling.

b) Kernel

The virtual start address of the kernel depends on the used architecture and is sometimes configurable. It is typically 0xC0000000 or 0x80000000. The end of the virtual range of the kernel is usually the virtual start address + the physical size of the RAM. The kernel is mapped in one continuous block from the virtual address range to the physical address range. Except the startup of the kernel, where the MMU tables are initialized, the kernel runs with enabled MMU.



Example: If the target has 32 MByte RAM at physical address 0x004000000, and the Linux' virtual kernel starts at 0xC0000000, then it will end at virtual address 0xC1FFFFFF.

c) Kernel Modules

Kernel modules run completely in virtual address space. The address space of a kernel module is unique, it is always accessible using it's virtual addresses.

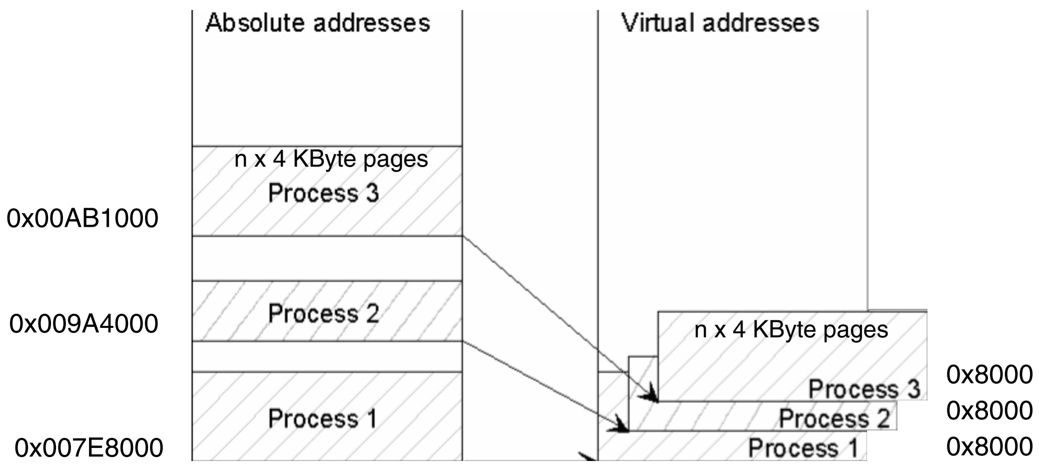
The address space of a kernel module is mapped dynamically page-by-page, when the module is loaded. **The kernel module is not mapped in one block to the physical memory but spread to free 4 KByte pages.** Of course, in virtual addressing mode, you don't realize this spreading.

The virtual address range of a kernel module usually lies above or below the virtual address range of the kernel. For the ARM architecture for example, the kernel modules are linked between the virtual end address of the user space and the kernel start address.

d) Processes and Threads

The virtual address range for processes depends heavily on the used architecture. For ARM it is defined in arch/arm/include/asm/memory.h and is usually form 0x00000000 to 0xBFFFFFFF. The virtual memory lets the process allocate and manage memory as if it alone owned all the memory of the system. Each process has one or more memory pages with a typical size of 4KBytes on 32bit architectures.

No virtual page will be mapped to physical memory before an access to the processes virtual address occurs. Only those pages, that are really used, are mapped to physical pages.



The virtual addresses of processes are **not unique** in opposite to the used physical address ranges. So it depends always on the individual process to unambiguously map it to the correct physical memory. Each process “sees” only it’s own memory and has no access to the memory of other processes (memory protection).

Threads share the virtual memory abstraction. All threads having the same parent process share the same virtual address space. The MMU translation is valid for all threads of one process. If the demand paging mechanism loads a page for a specific thread, it becomes automatically available for all threads of this process.

e) Libraries (Shared Objects)

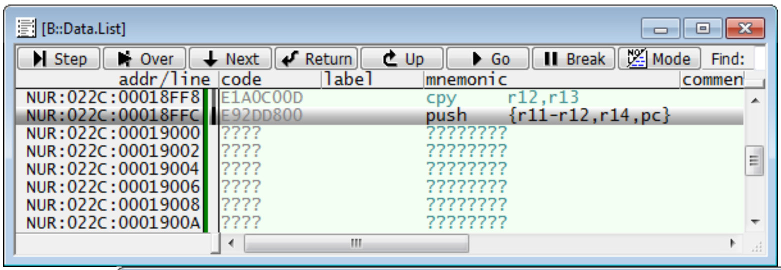
Linux loads and links libraries dynamically at run-time, when they are needed. They run in the context and virtual address space of the process, that uses them. See “Processes” for further details of the virtual memory management used. Watch out for demand paging as well!

3.3.) Demand Paging

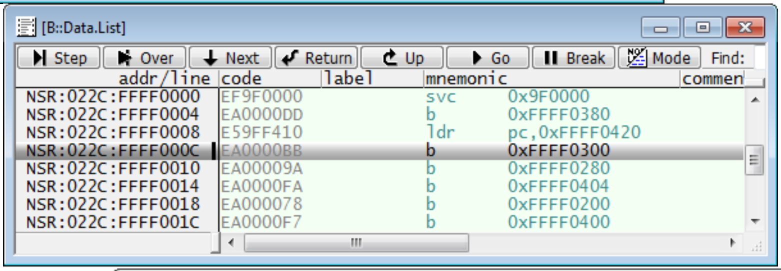
Linux is designed for heavy MMU usage with demand paging. Demand paging means, that code and data pages of processes (and libraries) are loaded first, if they are accessed. If the process tries to access a memory page, that is not yet loaded, it creates a page fault. The page fault handler then loads the appropriate page. Only the actual used pages are really loaded.

However, this demand paging disturbs the debugger a lot. As a work around, two patches are available to avoid demand paging while debugging. One is placed in the ELF loader, the other one is placed in the process handler. Please see the Linux Awareness Manual for details.

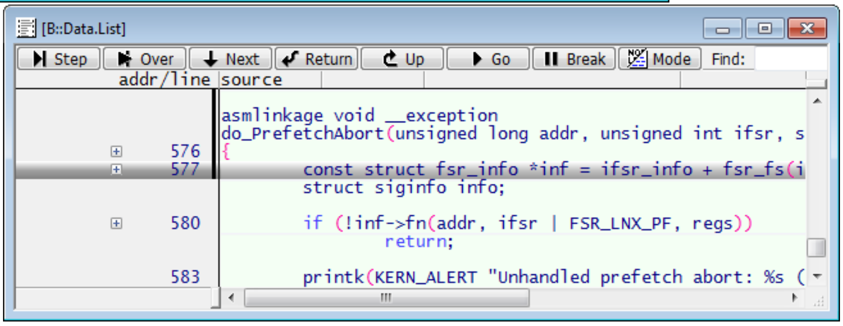
The following screen shots show an example of demand paging: the code of a process consists of more that one memory page. The program counter is at the end of the page (address 0x18FF8). The next page beginning at address 0x19000 is not yet loaded thus the debugger cannot read the memory at these addresses and shows question marks. After a single step, a PABORT exception takes place (address 0xFFFF000C) and then the kernel handler for prefecth abort (do_PrefetchAbort) is called which loads and maps the page and finally returns to the address which caused the exception (0x19000).



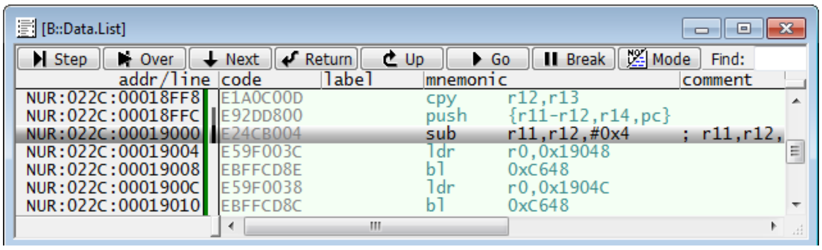
```
          addr/line code      label      mnemonic                     commen
NUR:022C:00018FF8 E1A0C00D               cpy      r12,r13
NUR:022C:00018FFC E92DD800               push     {r11-r12,r14,pc}
NUR:022C:00019000 ????                   ????????
NUR:022C:00019002 ????                   ????????
NUR:022C:00019004 ????                   ????????
NUR:022C:00019006 ????                   ????????
NUR:022C:00019008 ????                   ????????
NUR:022C:0001900A ????                   ????????
```



```
          addr/line code      label      mnemonic                     commen
NSR:022C:FFFF0000 EF9F0000               svc      0x9F0000
NSR:022C:FFFF0004 EA0000DD               b        0xFFFF0380
NSR:022C:FFFF0008 E59FF410               ldr      pc,0xFFFF0420
NSR:022C:FFFF000C EA0000BB               b        0xFFFF0300
NSR:022C:FFFF0010 EA00009A               b        0xFFFF0280
NSR:022C:FFFF0014 EA0000FA               b        0xFFFF0404
NSR:022C:FFFF0018 EA000078               b        0xFFFF0200
NSR:022C:FFFF001C EA0000F7               b        0xFFFF0400
```



```
    addr/line source

              asmlinkage void __exception
              do_PrefetchAbort(unsigned long addr, unsigned int ifsr, s
          576 {
          577         const struct fsr_info *inf = ifsr_info + fsr_fs(i
                      struct siginfo info;

          580         if (!inf->fn(addr, ifsr | FSR_LNX_PF, regs))
                              return;

          583         printk(KERN_ALERT "Unhandled prefetch abort: %s (
```



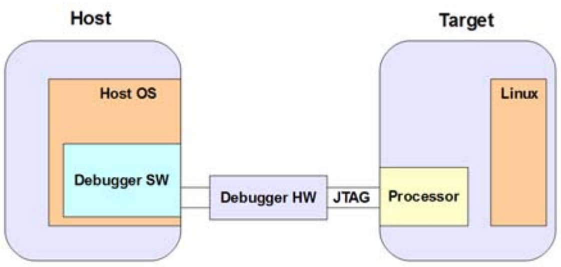
```
          addr/line code      label      mnemonic                     comment
NUR:022C:00018FF8 E1A0C00D               cpy      r12,r13
NUR:022C:00018FFC E92DD800               push     {r11-r12,r14,pc}
NUR:022C:00019000 E24CB004               sub      r11,r12,#0x4        ; r11,r12,
NUR:022C:00019004 E59F003C               ldr      r0,0x19048
NUR:022C:00019008 EBFFCD8E               bl       0xC648
NUR:022C:0001900C E59F0038               ldr      r0,0x1904C
NUR:022C:00019010 EBFFCD8C               bl       0xC648
```

4.) Run-Mode vs. Stop-Mode Debugging

There are two principle alternatives of debugging a Linux target: hardware based and software based. This chapter gives a small introduction about the differences to help you understand the operation theory of the hardware based debugger TRACE32 in conjunction with a Linux target.

a.) Hardware Based Debuggers

TRACE32 is a hardware based debugger, i.e. it uses special hardware to access target, processor and memory (e.g. by using the JTAG interface). No active target software is required, there are no software requirements at all at the target. 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 are visible physically, even if they are virtually not mapped into the MMU. If the debugger knows the address translation of all processes, you gain access to any process data at any time.

The demand paging of Linux implies, that pages of the application may be physically not present in the system, as long as they are not accessed. No access of such pages is possible (including software breakpoints), as long as they are not loaded.

Advantages:

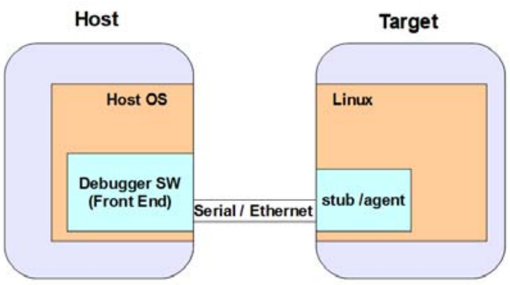
- **bootstrap, interrupt or post mortem debugging is possible**
- **no software restrictions (like memory protection, ...) apply to the debugger**
- **the full MMU table and code of all processes alive can be made visible**
- **only JTAG is required, no special communication interface as RS232 or Ethernet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

Disadvantages:

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

b) 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 at least the interrupts 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 and other processes.

Advantages:

- **halts only the desired process**
- **synchronization and communications to peripherals usually continue**
- **no debugger hardware an 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**

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

5.) Building the kernel, a kernel module and a simple demo application

Before going forward with writing Linux TRACE32 scripts and debugging the different Linux components, we will show how to build the kernel, a kernel module and a demo application for our demo board.

Building the kernel

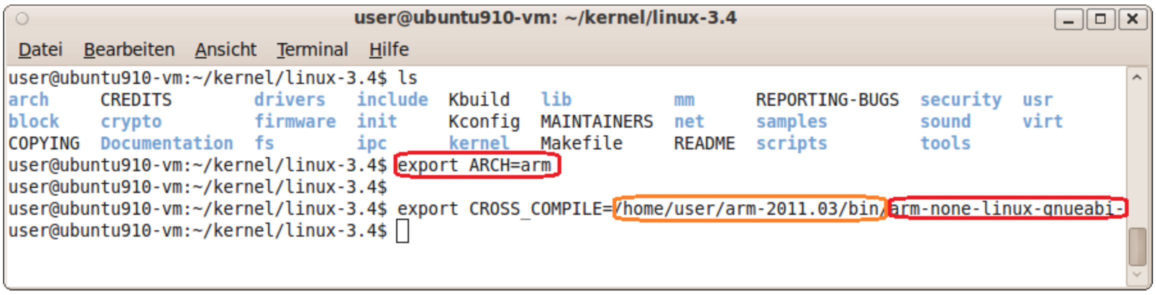
You can get the vanilla version of the kernel from ftp:ftp.kernel.org/pub/linux/kernel. We will download the kernel version 3.4. The kernel source code is always available in a complete tarball and an incremental patch and is distributed in both GNU zip (gzip) and bzip2 format. After downloading the kernel we will uncompress it using the following command:

```
$ tar xvjf linux-3.4.tar.bz2
```

or

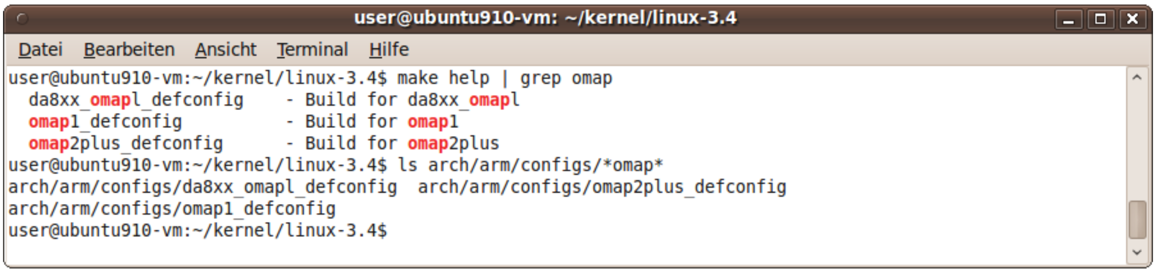
```
$ tar xvzf linux-3.4.tar.gz
```

We need to set then two environment variables used by the kernel makefiles. The variable ARCH should contain the used architecture (e.g. “arm” or *powerpc”). The other variable, CROSS_COMPILE, specifies the prefix used for all executables used during compilation (gcc and related bin-utils).


user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

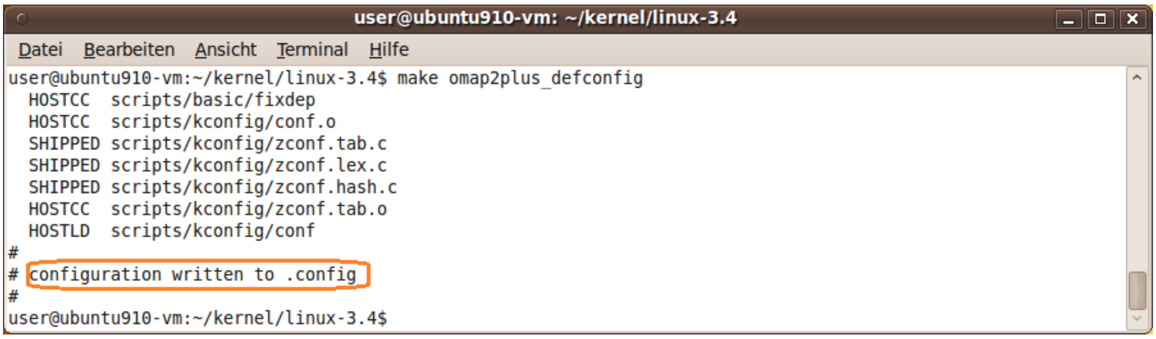
```
user@ubuntu910-vm:~/kernel/linux-3.4$ ls
arch     CREDITS        drivers   include  Kbuild   lib         mm      REPORTING-BUGS  security  usr
block    crypto         firmware  init     Kconfig  MAINTAINERS net     samples         sound     virt
COPYING  Documentation  fs        ipc      kernel   Makefile    README  scripts         tools
user@ubuntu910-vm:~/kernel/linux-3.4$ export ARCH=arm
user@ubuntu910-vm:~/kernel/linux-3.4$
user@ubuntu910-vm:~/kernel/linux-3.4$ export CROSS_COMPILE=/home/user/arm-2011.03/bin/arm-none-linux-gnueabi-
user@ubuntu910-vm:~/kernel/linux-3.4$
```

We need then to configure the kernel for out target platform. With “make help” we can get a list of all available platforms of our architecture. Each platform has a config file under arch/$ARCH/configs. For our Pandaboard, we use the config file omap2plus_defconfig.


user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

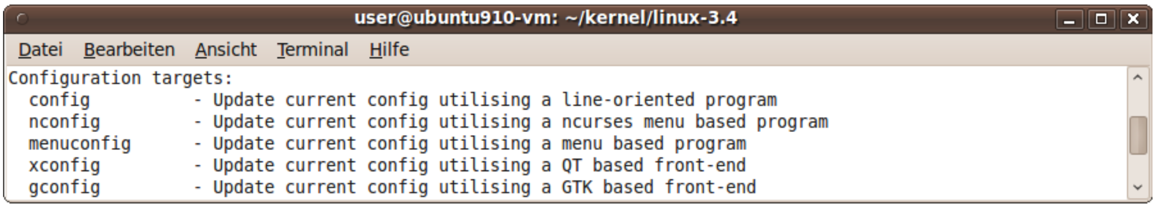
```
user@ubuntu910-vm:~/kernel/linux-3.4$ make help | grep omap
  da8xx_omapl_defconfig    - Build for da8xx_omapl
  omap1_defconfig          - Build for omap1
  omap2plus_defconfig      - Build for omap2plus
user@ubuntu910-vm:~/kernel/linux-3.4$ ls arch/arm/configs/*omap*
arch/arm/configs/da8xx_omapl_defconfig  arch/arm/configs/omap2plus_defconfig
arch/arm/configs/omap1_defconfig
user@ubuntu910-vm:~/kernel/linux-3.4$
```

With “make omap2plus_defconfig” we create then a configuration file (.config) for our target.

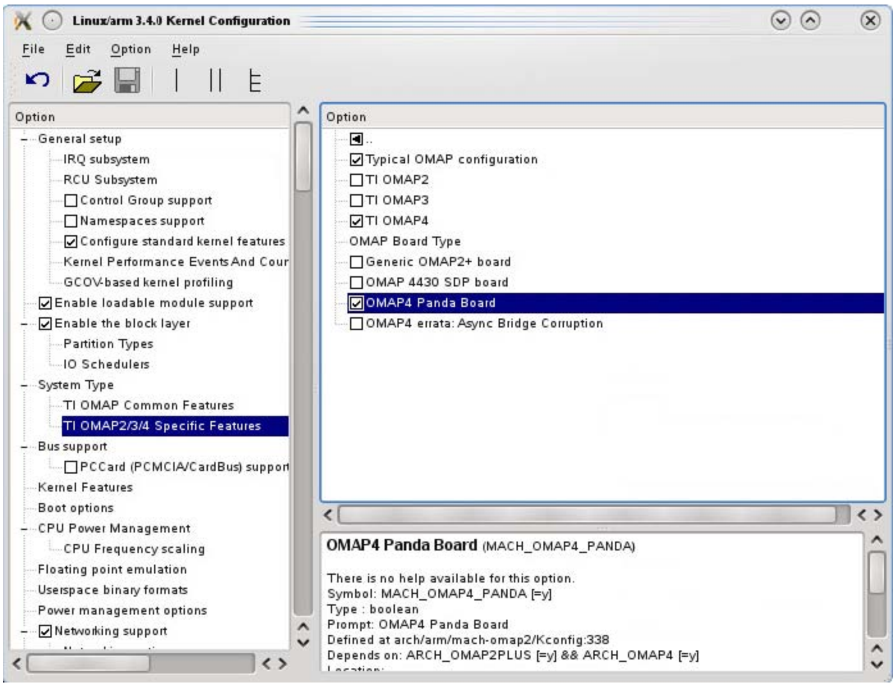

user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

```
user@ubuntu910-vm:~/kernel/linux-3.4$ make omap2plus_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
user@ubuntu910-vm:~/kernel/linux-3.4$
```

To edit this configuration, different interface and dialogs are available. We will use the (Qt) base configuration tool xconfig.


user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

```
Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based program
  menuconfig      - Update current config utilising a menu based program
  xconfig         - Update current config utilising a QT based front-end
  gconfig         - Update current config utilising a GTK based front-end
```

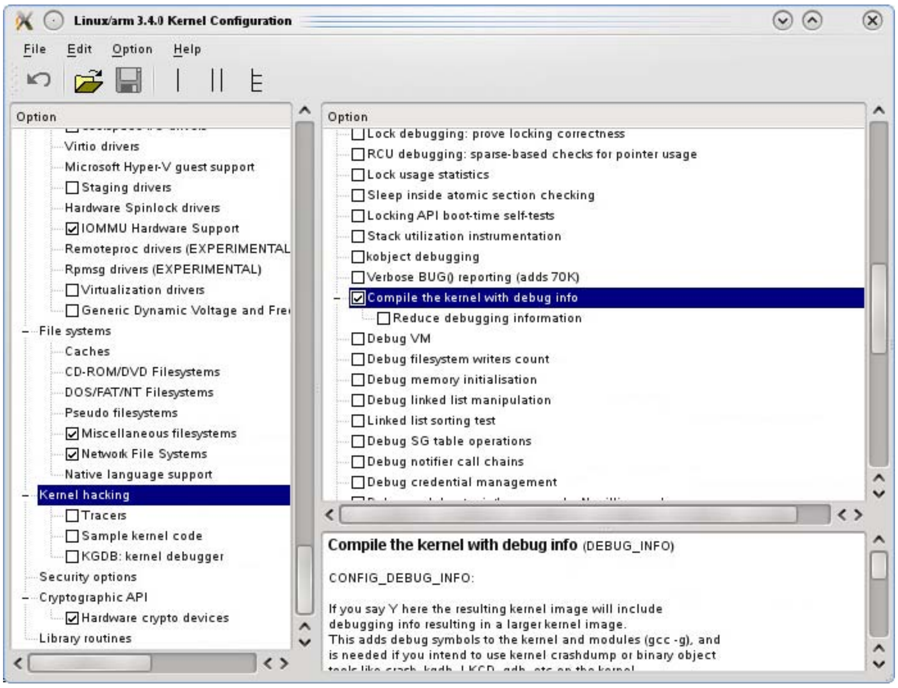


```
- General setup
    IRQ subsystem
    RCU Subsystem
    [ ] Control Group support
    [ ] Namespaces support
    [✓] Configure standard kernel features
    Kernel Performance Events And Cour
    GCOV-based kernel profiling
  [✓] Enable loadable module support
- [✓] Enable the block layer
    Partition Types
    IO Schedulers
- System Type
    TI OMAP Common Features
    TI OMAP2/3/4 Specific Features
- Bus support
    [ ] PCCard (PCMCIA/CardBus) suppor
  Kernel Features
  Boot options
- CPU Power Management
    CPU Frequency scaling
  Floating point emulation
  Userspace binary formats
  Power management options
- [✓] Networking support
```

```
..
[✓] Typical OMAP configuration
[ ] TI OMAP2
[ ] TI OMAP3
[✓] TI OMAP4
OMAP Board Type
[ ] Generic OMAP2+ board
[ ] OMAP 4430 SDP board
[✓] OMAP4 Panda Board
[ ] OMAP4 errata: Async Bridge Corruption
```

```
OMAP4 Panda Board (MACH_OMAP4_PANDA)

There is no help available for this option.
Symbol: MACH_OMAP4_PANDA [=y]
Type : boolean
Prompt: OMAP4 Panda Board
Defined at arch/arm/mach-omap2/Kconfig:338
Depends on: ARCH_OMAP2PLUS [=y] && ARCH_OMAP4 [=y]
```

The most important option that we have to enable here is “Compile the kernel with debug info” in “Kernel hacking”. If this option is not enabled, the kernel will be compiled without symbol information and Linux aware debugging will be not possible.



```
Option
  Virtio drivers
  Microsoft Hyper-V guest support
  [ ] Staging drivers
  Hardware Spinlock drivers
  [✓] IOMMU Hardware Support
  Remoteproc drivers (EXPERIMENTAL
  Rpmsg drivers (EXPERIMENTAL)
  [ ] Virtualization drivers
  [ ] Generic Dynamic Voltage and Fre
- File systems
  Caches
  CD-ROM/DVD Filesystems
  DOS/FAT/NT Filesystems
  Pseudo filesystems
  [✓] Miscellaneous filesystems
  [✓] Network File Systems
  Native language support
- Kernel hacking
  [ ] Tracers
  [ ] Sample kernel code
  [ ] KGDB: kernel debugger
  Security options
- Cryptographic API
  [✓] Hardware crypto devices
  Library routines
```

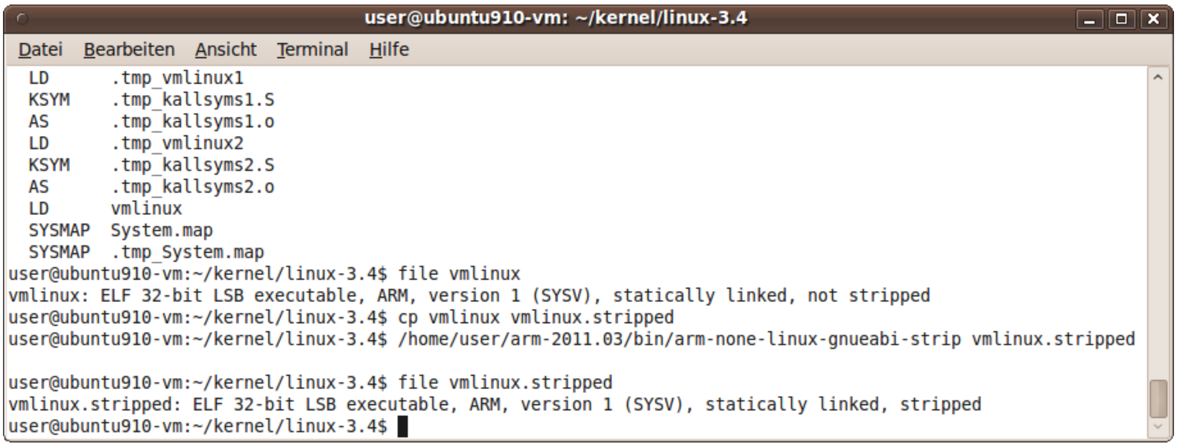
```
Option
[ ] Lock debugging: prove locking correctness
[ ] RCU debugging: sparse-based checks for pointer usage
[ ] Lock usage statistics
[ ] Sleep inside atomic section checking
[ ] Locking API boot-time self-tests
[ ] Stack utilization instrumentation
[ ] kobject debugging
[ ] Verbose BUG() reporting (adds 70K)
[✓] Compile the kernel with debug info
    [ ] Reduce debugging information
[ ] Debug VM
[ ] Debug filesystem writers count
[ ] Debug memory initialisation
[ ] Debug linked list manipulation
[ ] Linked list sorting test
[ ] Debug SG table operations
[ ] Debug notifier call chains
[ ] Debug credential management
```

Compile the kernel with debug info (DEBUG_INFO)

CONFIG_DEBUG_INFO:

If you say Y here the resulting kernel image will include
debugging info resulting in a larger kernel image.
This adds debug symbols to the kernel and modules (gcc -g), and
is needed if you intend to use kernel crashdump or binary object

Now we are ready to compile the kernel. We will compile the bare kernel by calling “make vmlinux”. After the compile process finishes, we will get a statically linked Elf file.

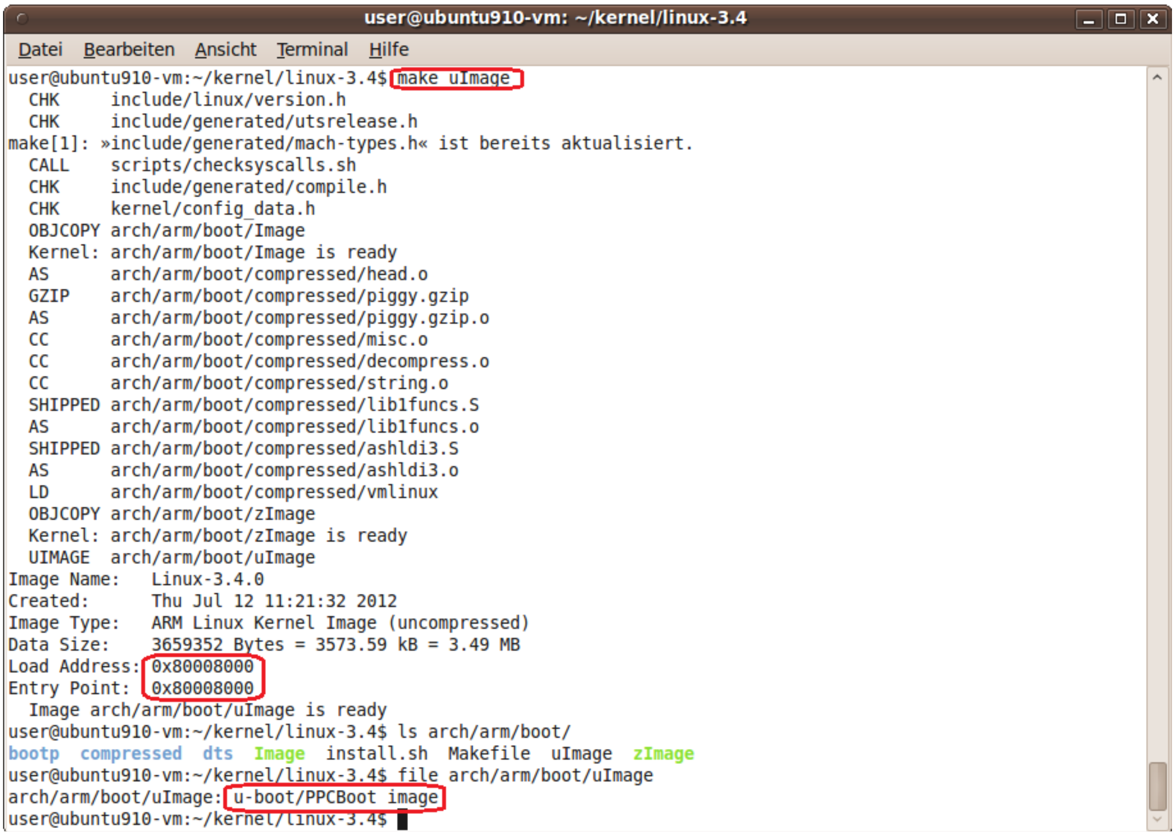


```
  LD      .tmp_vmlinux1
  KSYM    .tmp_kallsyms1.S
  AS      .tmp_kallsyms1.o
  LD      .tmp_vmlinux2
  KSYM    .tmp_kallsyms2.S
  AS      .tmp_kallsyms2.o
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
user@ubuntu910-vm:~/kernel/linux-3.4$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked, not stripped
user@ubuntu910-vm:~/kernel/linux-3.4$ cp vmlinux vmlinux.stripped
user@ubuntu910-vm:~/kernel/linux-3.4$ /home/user/arm-2011.03/bin/arm-none-linux-gnueabi-strip vmlinux.stripped

user@ubuntu910-vm:~/kernel/linux-3.4$ file vmlinux.stripped
vmlinux.stripped: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked, stripped
user@ubuntu910-vm:~/kernel/linux-3.4$ █
```

If you also need the stripped version of the kernel, you can use the stip command. **Do not compile the kernel twice, with and without debugging information enabled!**

You can also compile the U-Boot wrapped image of the kernel (uImage) with “make uImage”. **If you use both files uImage and vmlinux (e.g. dowloading uImage with the bootloader and loading the kernel symbols from vmlinux) you need to take care that both files come from the same compiling process.**

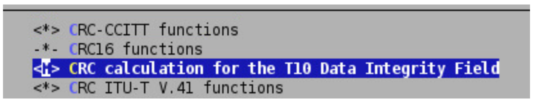
You will find the uImage under arch/arm/boot.


user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

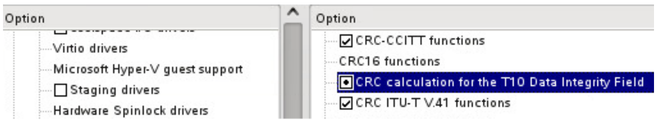
```
user@ubuntu910-vm:~/kernel/linux-3.4$ make uImage
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: »include/generated/mach-types.h« ist bereits aktualisiert.
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
  CHK     kernel/config_data.h
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  CC      arch/arm/boot/compressed/string.o
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  SHIPPED arch/arm/boot/compressed/ashldi3.S
  AS      arch/arm/boot/compressed/ashldi3.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.4.0
Created:      Thu Jul 12 11:21:32 2012
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3659352 Bytes = 3573.59 kB = 3.49 MB
Load Address: 0x80008000
Entry Point:  0x80008000
  Image arch/arm/boot/uImage is ready
user@ubuntu910-vm:~/kernel/linux-3.4$ ls arch/arm/boot/
bootp  compressed  dts  Image  install.sh  Makefile  uImage  zImage
user@ubuntu910-vm:~/kernel/linux-3.4$ file arch/arm/boot/uImage
arch/arm/boot/uImage: u-boot/PPCBoot image
user@ubuntu910-vm:~/kernel/linux-3.4$
```

Building a kernel module

If the kernel module is in kernel source tree then you can select to compile it as module in the kernel configuration dialog (e.g. using xconfig or meuconfig:).



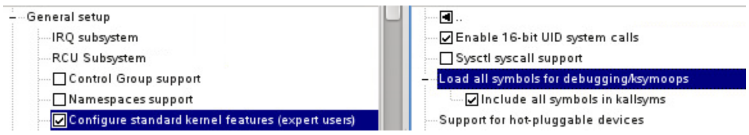
```
<*> CRC-CCITT functions
-*- CRC16 functions
<M> CRC calculation for the T10 Data Integrity Field
<*> CRC ITU-T V.41 functions
```



Virtio drivers
Microsoft Hyper-V guest support
☐ Staging drivers
Hardware Spinlock drivers

☑ CRC-CCITT functions
CRC16 functions
▣ CRC calculation for the T10 Data Integrity Field
☑ CRC ITU-T V.41 functions

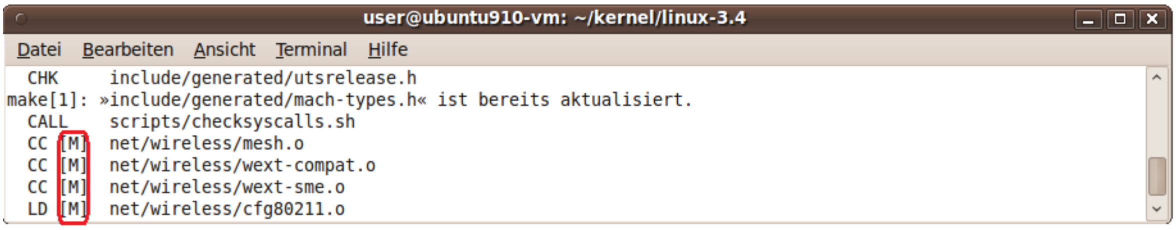
	The kernel contains all section information if it has been configured with CONFIG_KALLSYMS=y. When configuring the kernel, set the option "General Setup"-> "Configure standard kernel features" -> "Load all symbols" to yes. Without KALLSYMS, no section information is available and debugging kernel modules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



General setup
IRQ subsystem
RCU Subsystem
☐ Control Group support
☐ Namespaces support
☑ Configure standard kernel features (expert users)

..
☑ Enable 16-bit UID system calls
☐ Sysctl syscall support
Load all symbols for debugging/ksymoops
☑ Include all symbols in kallsyms
Support for hot-pluggable devices

You can the compile the modules by calling “make modules”.


user@ubuntu910-vm: ~/kernel/linux-3.4
Datei Bearbeiten Ansicht Terminal Hilfe

```
  CHK     include/generated/utsrelease.h
make[1]: »include/generated/mach-types.h« ist bereits aktualisiert.
  CALL    scripts/checksyscalls.sh
  CC [M]  net/wireless/mesh.o
  CC [M]  net/wireless/wext-compat.o
  CC [M]  net/wireless/wext-sme.o
  LD [M]  net/wireless/cfg80211.o
```

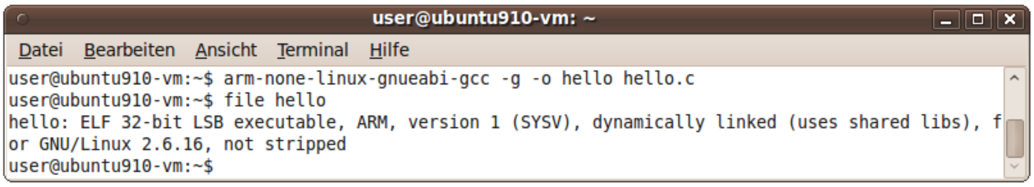
If the module “lives” outside the kernel source tree, you need to create a Makefile where you instruct make how to find the kernel source files.

```
obj-m += demomod.o
EXTRA_CFLAGS:= -g
all:
»	make -C /home/user/linux-3.4 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-  M=/home/user/mod/procfs modules
clean:
»	make -C /home/user/linux-3.4 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-  M=/home/user/mod/procfs clean
```

Building a user application

When compiling an application, you need to use the “-g” compiler option to enable debug information:

```
$ arm-none-linux-gnueabi-gcc -g -o hello hello.c
```



```
user@ubuntu910-vm:~$ arm-none-linux-gnueabi-gcc -g -o hello hello.c
user@ubuntu910-vm:~$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), f
or GNU/Linux 2.6.16, not stripped
user@ubuntu910-vm:~$
```

Setting up a script for Linux-aware debugging

This chapter will tell about the **typical steps** how to prepare a Linux target and the TRACE32 debugger **for convenient Linux-Debugging**. The used target for the explanation is a Pandaboard with a OMAP4430 (dual Cortex-A9).

The following pages will show the setup by now step by step:

1. **Capture of commands for a script**
2. **Linux Setup-Steps and -Commands**
3. **Example Linux Setup-Script file**
4. **Linux Debugging Dialog**

1.) Capture of commands for a script

It can be an advantage to record the commands and wanted settings used inside the TRACE32 graphical user interface (GUI) right from the start. So you should open first a LOG-file:

LOG.OPEN *<file>*	Create and open a file for the commands to be logged. The default extension for LOG-files is (`.log`)
LOG.CLOSE	Close the LOG-file

The advantage of this log file is it records all commands TRACE32 provides till you close it. Even menu selections and mouse-clicks. Whereas the ClipSTORE and STORE <file> commands save only specified parameters (e.g SYStem settings) to be reactivated by the automatically generated script later.

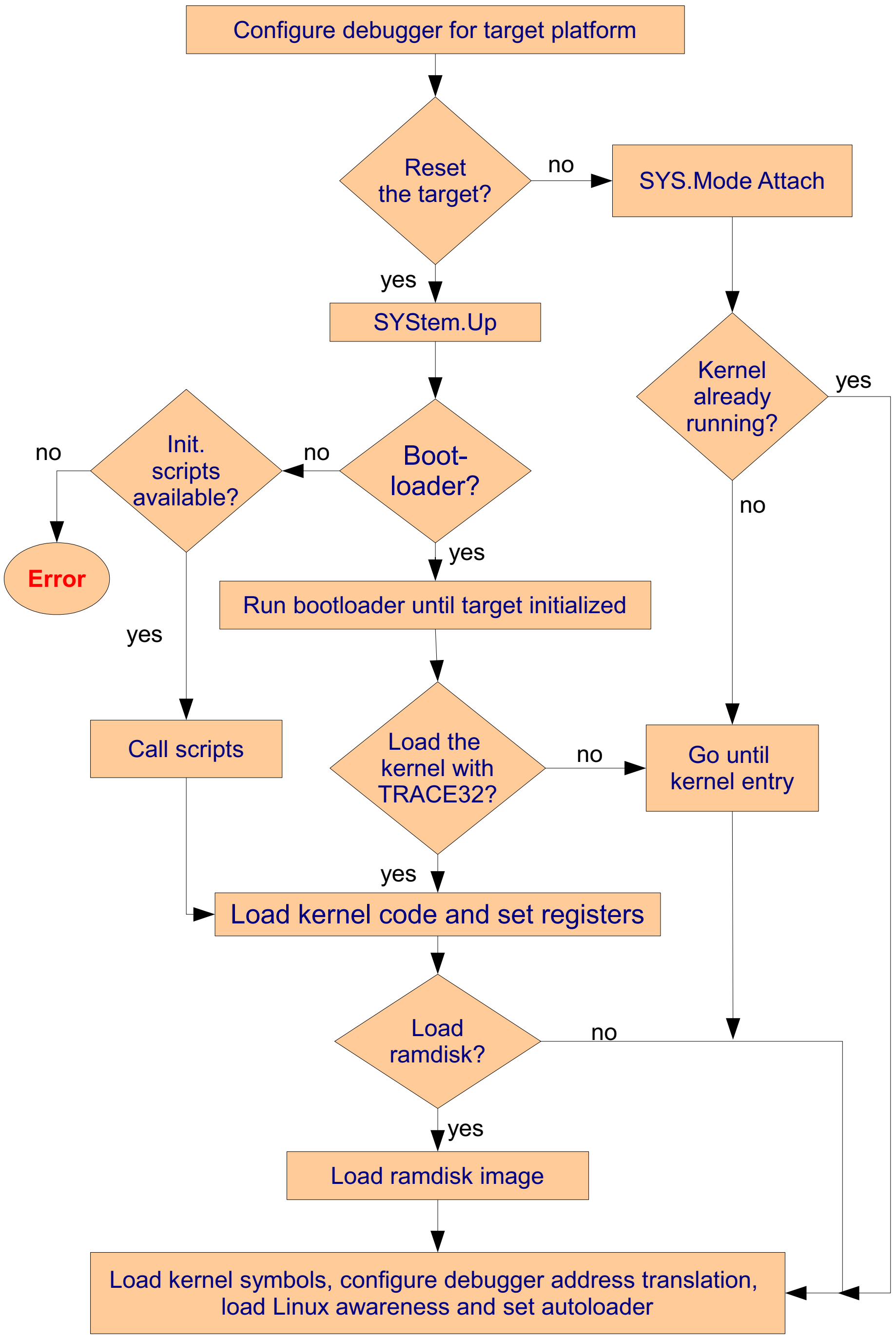
STOre *<file>* **SYStem**	Create a batch to restore the SYStem settings
ClipSTOre SYStem	Provide the commands to restore the SYStem settings in the cliptext

The HISTory command only samples the commands from the command line. But it offers a quick access to the commands used already. Use the cursor-up key or mouse to select commands form the HISTory list.

HISTory.type	Display the command history

2.) Linux Setup-Steps and -Commands

For writing a correct startup script, different steps are needed depending on how Linux is loaded and started. The commonly used steps are summarized in the following diagram:



Configure debugger for target platform
Reset the target?
no
SYS.Mode Attach
yes
SYStem.Up
Kernel already running?
yes
no
Init. scripts available?
no
Boot-loader?
no
Error
yes
Run bootloader until target initialized
yes
Call scripts
Load the kernel with TRACE32?
no
Go until kernel entry
yes
Load kernel code and set registers
Load ramdisk?
no
yes
Load ramdisk image
Load kernel symbols, configure debugger address translation, load Linux awareness and set autoloader

So, typically, a Linux script consists of

- Debugger Reset.
- Chip-related debugger setup.
- Target setup (if needed).
- Download the kernel code and setting kernel start parameters (if needed)
- Download the filesystem (if needed).
- Loading kernel symbols
- Setup debugger MMU translation.
- Setup Linux awareness.
- Setup the symbol autoloader

The numbering scheme goes as follows: Numbers are for sequential steps. Letters are for alternate and conditional steps.

1) 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. So use command **RESet** for a complete restart or the specific resets for each debugger function.

1.a) Complete Reset

```
RESet                           ; reset debugger completely
```

1.b) Individual Resets

```
SYStem.Down                     ; stops debugger target connection
TASK.RESet                      ; reset Linux awareness
Break.Delete                    ; remove any left breakpoints
MAP.RESet                       ; reset debugger’s memory mapping
TRANSlation.RESet               ; reset debugger’s MMU translation
sYmbol.RESet                    ; reset debugger’s symbols table
SYStem.RESet                    ; reset debugger’s system settings
```

The above sequence are the **minimal** statements you should use for resetting the system.

2) Debugger Setup

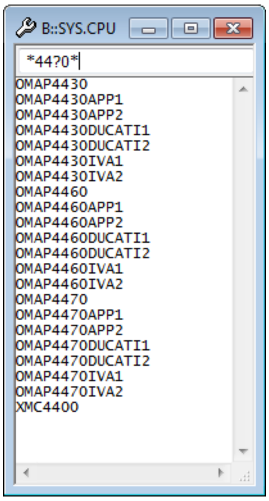
First you need to setup the debugger to be able to connect to the target platform. This includes e.g. selecting the appropriated CPU, setting the JTAG clock and selecting target-specific debugger options. Moreover, some additional options related to Linux debugging have to be enabled like address extension. Finally, you need to connect to the target using **SYStem.Up** or **SYStem.Mode Attach**.

2.a) Debugger setup for the OMAP4430 target

This tutorial shows the needed command sequence for an OMAP4430 from Texas Instruments. These steps will be similar for other target platforms. If you have a different target platform or architecture you may need other debugger settings.

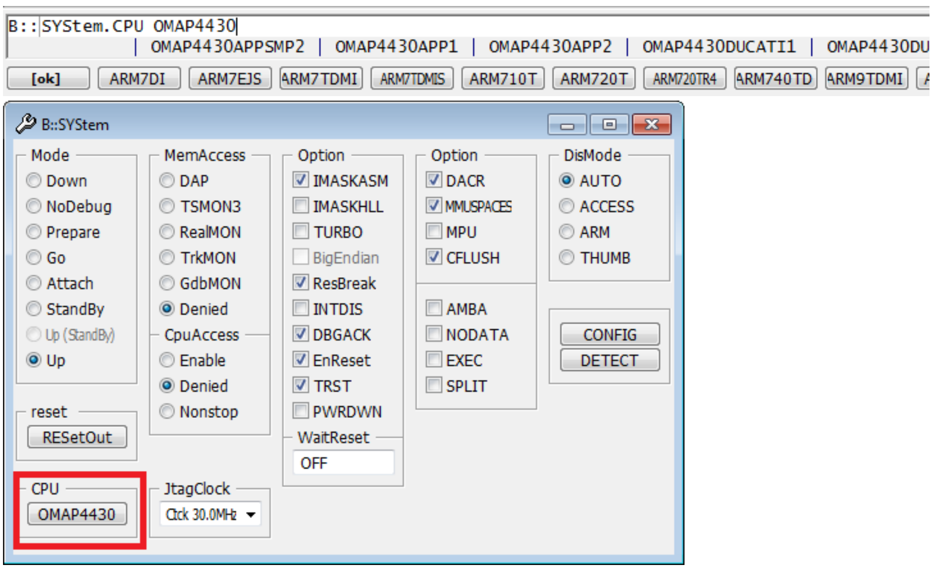
CPU selection

You need to select a CPU to debug your target.



B::SYS.CPU

```
*44?0*
OMAP4430
OMAP4430APP1
OMAP4430APP2
OMAP4430DUCATI1
OMAP4430DUCATI2
OMAP4430IVA1
OMAP4430IVA2
OMAP4460
OMAP4460APP1
OMAP4460APP2
OMAP4460DUCATI1
OMAP4460DUCATI2
OMAP4460IVA1
OMAP4460IVA2
OMAP4470
OMAP4470APP1
OMAP4470APP2
OMAP4470DUCATI1
OMAP4470DUCATI2
OMAP4470IVA1
OMAP4470IVA2
XMC4400
```



```
B::SYStem.CPU OMAP4430
  | OMAP4430APPSMP2 | OMAP4430APP1 | OMAP4430APP2 | OMAP4430DUCATI1 | OMAP4430DU
[ok] ARM7DI ARM7EJS ARM7TDMI ARM7TDMIS ARM710T ARM720T ARM720TR4 ARM740TD ARM9TDMI
```

B::SYStem

- Mode: ( ) Down ( ) NoDebug ( ) Prepare ( ) Go ( ) Attach ( ) StandBy ( ) Up (StandBy) [✓ Up]
- MemAccess: ( ) DAP ( ) TSMON3 ( ) RealMON ( ) TrkMON ( ) GdbMON [✓ Denied]
- CpuAccess: ( ) Enable [✓ Denied] ( ) Nonstop
- Option: [✓ IMASKASM] [ ] IMASKHLL [ ] TURBO [ ] BigEndian [✓ ResBreak] [ ] INTDIS [✓ DBGACK] [✓ EnReset] [✓ TRST] [ ] PWRDWN
- WaitReset: OFF
- Option: [✓ DACR] [✓ MMUSPACES] [ ] MPU [✓ CFLUSH] [ ] AMBA [ ] NODATA [ ] EXEC [ ] SPLIT
- DisMode: [✓ AUTO] ( ) ACCESS ( ) ARM ( ) THUMB
- reset: RESetOut
- CPU: OMAP4430
- JtagClock: Ctck 30.0MHz

You can use the search field in the **SYStem.CPU** window to find your CPU name. Alternatively, you can use the command line to write the CPU name partially to be completed by pressing the tabulator key on the keyboard. This way also the amount of displayed CPUs is reduced temporarily.:

```
SYStem.CPU OMAP4430             ; OMAP4430 target to be debugged
```

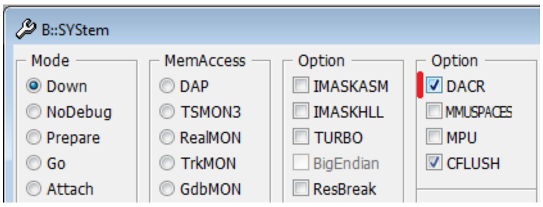
JTAG Clock

Using the command **SYStem.JtagClock** you can select an appropriated JTAG clock for you target. This could be necessary if the default clock set by the debugger is for example too high. You can see the selected JTAG clock in the **SYStem** window. For our Pandaboard we use the clock CTCK 30MHz.

```
SYStem.JtagClock CTCK 30MHz     ;
```

DACR option

If you have an ARM target, you need to set the DACR Option to ON to grant write access to all memory domains for the debugger. Because the target MMU usually write protects some memory areas (e.g. code pages). But the debugger needs write access for example to set software breakpoints..


B::SYStem

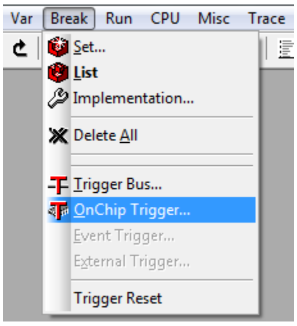
Mode	MemAccess	Option	Option
● Down	○ DAP	☐ IMASKASM	☑ DACR
○ NoDebug	○ TSMON3	☐ IMASKHLL	☐ MMUSPACES
○ Prepare	○ RealMON	☐ TURBO	☐ MPU
○ Go	○ TrkMON	☐ BigEndian	☑ CFLUSH
○ Attach	○ GdbMON	☐ ResBreak	

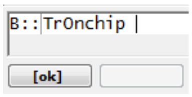
```
    SYStem.Option DACR ON            ; give Debugger global write permissions
```

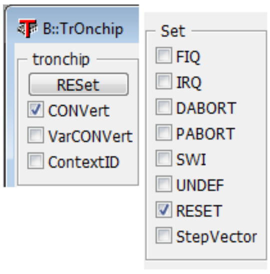
	Different targets may need some hardware related settings. Please check the specific Processor Architecture Manual ("debugger_<architecture>.pdf") for possible options.

Onchip Trigger

You can configure the debugger to halt or not on exceptions. For ARM cores, TRACE32 setup the target to halts on most of the exception routines. Linux uses some of these exceptions, so the debugger's detection has to be switched off. Debugging MIPS cores for example doesn't need this.




B::TrOnchip


B::TrOnchip

tronchip

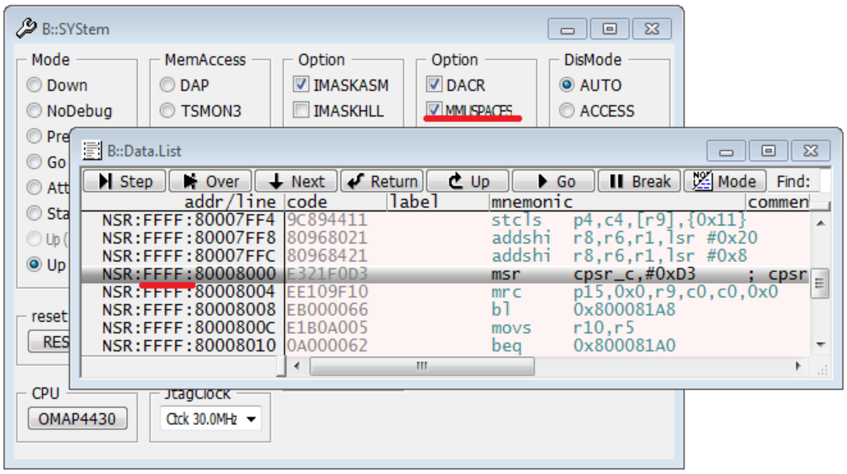
☑ CONVert
☐ VarCONVert
☐ ContextID

Set
☐ FIQ
☐ IRQ
☐ DABORT
☐ PABORT
☐ SWI
☐ UNDEF
☑ RESET
☐ StepVector

```
    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
```

Address extension

Switch on the debugger's virtual address extension to use space ids (needed to differ e.g. several "main" functions of different processes). The addresses in the Data.List and Data.dump windows will be extended with a space id (e.g **FFFF**:800080000).



Option: [✓ IMASKASM] IMASKHLL
Option: [✓ DACR] [✓ MMUSPACES]
DisMode: [✓ AUTO] ACCESS

```
       addr/line code     label       mnemonic                          commen
NSR:FFFF:80007FF4 9C894411            stcls   p4,c4,[r9],{0x11}
NSR:FFFF:80007FF8 80968021            addshi  r8,r6,r1,lsr #0x20
NSR:FFFF:80007FFC 80968421            addshi  r8,r6,r1,lsr #0x8
NSR:FFFF:80008000 E321F0D3            msr     cpsr_c,#0xD3          ; cpsr
NSR:FFFF:80008004 EE109F10            mrc     p15,0x0,r9,c0,c0,0x0
NSR:FFFF:80008008 EB000066            bl      0x800081A8
NSR:FFFF:8000800C E1B0A005            movs    r10,r5
NSR:FFFF:80008010 0A000062            beq     0x800081A0
```

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

Remark: Older documentation and TRACE32-SW uses “SYStem.Option MMU ON” instead of “MMUSPACES”. Please use only the new naming.

The “SYStem.Option MMUSPACES” can only be changed if NO debug symbols are loaded!

Connect to the target

The command **SYStem.Up** resets the target (if supported by the JTAG interface) and enters debug mode.

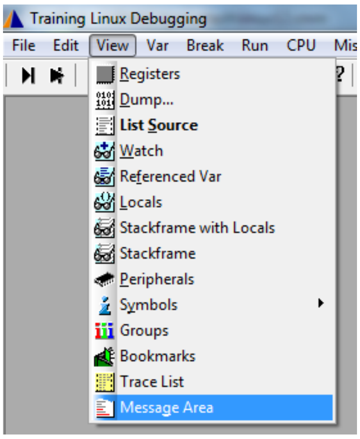
```
SYStem.Up                       ; activates the JTAG communication
```

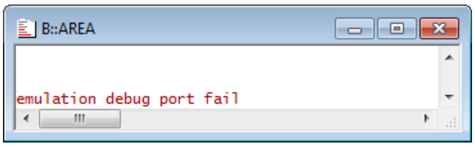
The command **SYStem.Mode Attach** attaches to the target without resetting the cores.

```
SYStem.Mode Attach              ; attach to the cores
```

At this stage the Linux awareness has not yet been enable, so if a problem occurs then it is related to the debugger settings or to a problem on the target. Please check in this case the AREA window for errors and warnings.

AREA.view [<*area*>]





```
emulation debug port fail
```

2.b) Examples of specific settings for other processors

PPC: Some of the PowerPC processors have an adjustable IO base address. If this base address can not be determined from the processor by the debugger itself, you need to tell the debugger it's address. You may either set it manually, or let the debugger try to get the reset configuration word.

```
SYStem.Option BASE AUTO          ; get reset io base address
```

On some PowerPC architectures (especially PowerQUICC II and III), a JTAG memory access to an area, that is not covered by a chip select, may cause a processor deadlock. Only a complete reset recovers this state. To prevent the debugger from accessing those memory areas, explicitly prohibit the access.

```
MAP.DenyAccess 0x01000000--0xefffffff      ; deny access to non-existent
                                           ; memory
```

MIPS (e.g. AU1200)**:** Also for MIPS cores the debugger needs extra access permissions to get write access to all memory areas.

```
SYStem.Option UnProtect ON       ; give Debugger global write permissions
```

Some CPUs have a special mode to transfer data faster via the debug interface. You can use this opportunity to speed up the download of the Linux kernel or the RAM-Disk to the target. This option usually needs to be undone after the transfer to enable normal debugging.

```
SYStem.Option TURBO ON           ; activate special CPU JTAG speed

SYStem.Option TURBO OFF          ; disable TURBO option for safety
```

While single stepping, external interrupts may occur. On some architectures, this leads with the next single step into the interrupt handler. This effect disturbs normally 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       ; lock interrupts while assembler single stepping
SETUP.IMASKHLL OFF      ; allow interrupts while HLL single stepping
```

If a 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.

3) Target Setup

3.1) Basic Board Setup

Of course the actual target setup depends on your individual target and should arrange to have the necessary knowledge about your target characteristics. The most important target initialization is to bring the processor in a well-defined state and initialize the board to get access to the RAM. There are two ways to initialize your target:

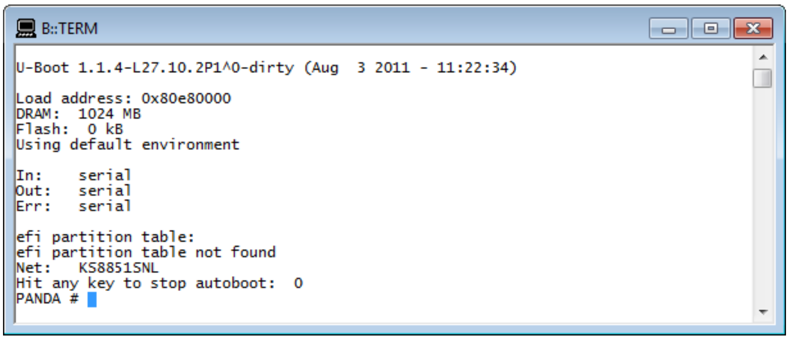
- Target initialization by a bootloader
- Target initialization by the debugger

3.1.a) Target initialization by a bootloader

Let the bootloader run as long as it needs to initialize the hardware and stop it before it begins loading the kernel.

```
Go                           ; start the bootloader as long as necessary
WAIT 5.s                     ; to initialize the target
Break                        ; stop before loading the kernel
```

Usually the bootloader prints a countdown message in the serial terminal window after target initialization is finished.



```
U-Boot 1.1.4-L27.10.2P1^0-dirty (Aug  3 2011 - 11:22:34)

Load address: 0x80e80000
DRAM:  1024 MB
Flash:  0 kB
Using default environment

In:    serial
Out:   serial
Err:   serial

efi partition table:
efi partition table not found
Net:   KS8851SNL
Hit any key to stop autoboot:  0
PANDA # 
```

You can use the terminal window "**TERM**" provided by TRACE32.

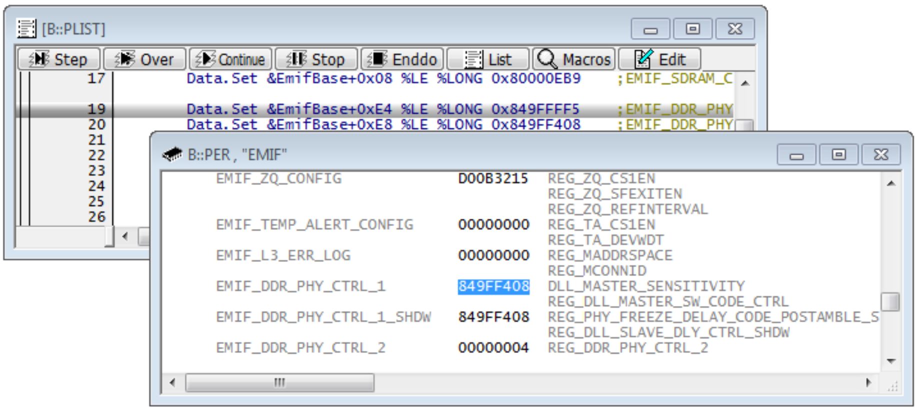
```
TERM.RESet                            ; reset old and set new definitions
TERM.METHOD COM com4 115200. 8 NONE 1STOP NONE
                                      ; for com10 use \\.\com10
TERM.SIZE 80 1000                     ; define capacity of the TERM window
TERM.SCROLL ON                        ; scrolling follows to the TERM cursor
TERM.Mode VT100                       ; or ASCII (default), STRING, HEX ...
WINPOS 50% 0%  50% 100% term_win      ; define next window position and size
TERM.view                             ; open the TERM window
SCREEN.ALways                         ; TERM window always updated
```

3.1.b) Target initialization by the debugger

You can set all processor **peripheral registers** explicitly by the debugger to the configuration required by the board. For sure this should finally be done by a script. It's a better style to call the scripts for own topics (e.g. board_setup.cmm) within your main setup script for Linux-Debugging (e.g. linux.cmm):

```
; main setup script linux.cmm
; ...
 DO board_setup.cmm                       ; run the special setup by a call
```

Inside the script for the target initialization the parameters are set by command Data.Set or PER.Set:



```
Data.Set &EmifBase+0x08 %LE %LONG 0x80000EB9    ;EMIF_SDRAM_C

Data.Set &EmifBase+0xE4 %LE %LONG 0x849FFFF5    ;EMIF_DDR_PHY
Data.Set &EmifBase+0xE8 %LE %LONG 0x849FF408    ;EMIF_DDR_PHY
```

```
EMIF_ZQ_CONFIG               D00B3215  REG_ZQ_CS1EN
                                       REG_ZQ_SFEXITEN
                                       REG_ZQ_REFINTERVAL
EMIF_TEMP_ALERT_CONFIG       00000000  REG_TA_CS1EN
                                       REG_TA_DEVWDT
EMIF_L3_ERR_LOG              00000000  REG_MADDRSPACE
                                       REG_MCONNID
EMIF_DDR_PHY_CTRL_1          849FF408  DLL_MASTER_SENSITIVITY
                                       REG_DLL_MASTER_SW_CODE_CTRL
EMIF_DDR_PHY_CTRL_1_SHDW     849FF408  REG_PHY_FREEZE_DELAY_CODE_POSTAMBLE_S
                                       REG_DLL_SLAVE_DLY_CTRL_SHDW
EMIF_DDR_PHY_CTRL_2          00000004  REG_DDR_PHY_CTRL_2
```

3.2) Additional Setup

Additional setup steps might be needed for individual requirements. E.g. the setup of the ARM ETM trace ports could be managed by the script "init_etm.cmm".

```
DO init_etm                               ; setup of the ARM ETM trace ports
```

4) Download the kernel

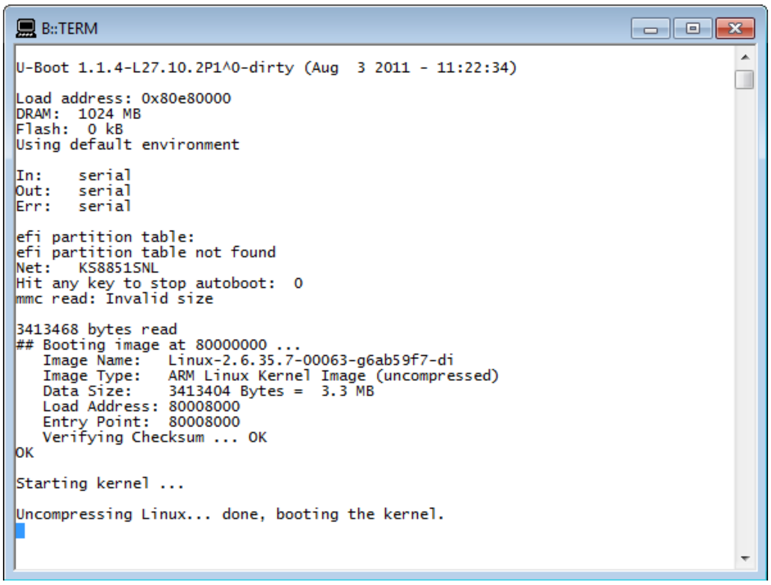
Loading and booting the Linux kernel can also be done by two different procedures:

a) Downloading and booting the kernel by the bootloader

b) Downloading and booting the kernel by the debugger

4.a) Downloading and booting the kernel by the bootloader

The can let the bootloader load the kernel into the RAM, decompress it (if compressed) and start it from there. In this case you need to configure the bootloader using its command line. The kernel will be typically loaded from flash memory, a memory card or over the network. The bootloader will also take car of setting the kernel start parameters.



```
U-Boot 1.1.4-L27.10.2P1^0-dirty (Aug  3 2011 - 11:22:34)

Load address: 0x80e80000
DRAM:  1024 MB
Flash:  0 kB
Using default environment

In:    serial
Out:   serial
Err:   serial

efi partition table:
efi partition table not found
Net:   KS8851SNL
Hit any key to stop autoboot:  0
mmc read: Invalid size

3413468 bytes read
## Booting image at 80000000 ...
   Image Name:   Linux-2.6.35.7-00063-g6ab59f7-di
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3413404 Bytes =  3.3 MB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

You can set an onchip breakpoint at the entry point of the kernel to debug the kernel startup.

4.b) Downloading and booting the kernel by the debugger

You can use the debugger to download and start Linux. In this case, you have to set the correct initial values for the processor registers.

At this stage, the target MMU is still disabled, so you need to download the kernel code using physical addresses. However, the kernel ELf “vmlinux”, which is usually used for the kernel, contains virtual addresses. Thus you need to subtract the virtual address base from the physical address base. For example, if the kernel has the virtual start address 0xC0000000 (which is a typical value) and should be downloaded on the RAM starting at the physical address 0x80000000, the command to download the kernel would be:

```
Data.LOAD.Elf vmlinux 0x80000000-0xC0000000 /NosYmbol
```

Please note that "`0x80000000-0xC0000000`" is not an address range. Both addresses are separated here by a single minus sign.

In our example here, if you simply try to download the kernel using the command

```
Data.LOAD.Elf vmlinux /NosYmbol
```

The debugger will try do download the kernel code using the virtual addresses as physical ones which is wrong and could result on a fatal error on the target.

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 0x80000000-0xC0000000) you have to only load the kernel code without symbols using the option /NoSymbol. Otherwise, the kernel symbols will be loaded on wrong addresses. The symbols are 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 0x80000000-0xC0000000 0x80000000--0x8FFFFFFF
                                                  /NosYmbol
```

When downloading the kernel using the debugger, you need also to take care of setting the program counter at the entry point of the kernel and the kernel start parameters in the general purpose registers. For the ARM architecture, the kernel get two start parameters. The first one, which should be written in R1, is a machine specific code. The second one should be written in R2 and is a pointer to a structure in the memory which contains information about the RAM start and size, the boot command line... You need also to setup this structure. Other architectures have different kernel start parameters.

```
Register.RESet
Register.Set PC 0x80008000     ; set PC on start address of image
Register.Set R1 2791.          ; Set machine type in R1 (2791.)
Register.Set R2 0x80000100
```

The code for the machine type can be found in the list “mach-types” inside the controller specific tool folder.

5.) Download the filesystem

You can configure the bootloader to load the filesystem (e.g. from flash memory or over the network) or configure the kernel to mount a network filesystem (nfs). Alternatively you can download a ramdisk image to the RAM for example:

```
Data.LOAD.Binary ramdisk.image.gz 0x81600000 /NoClear /NosYmbol
```

6.) Loading 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 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.

7.) S etup debugger MMU translation

The debugger needs to have some information about the format of the MMU tables used by the kernel and the kernel address translation. This is configured using the command **MMU.FORMAT**

```
MMU.FORMAT LINUX swapper_pg_dir 0xc0000000--0xc1ffffff 0x80000000
```

The first argument of this command is the format of the MMU tables which could be e.g. “LINUX” or “LINUXSWAP”. Please check rtos_linux_stop.pdf for actual format specifier. The second argument is a kernel symbol for the kernel page table and is usually called `swapper_pg_dir`. The third parameter is the virtual to physical kernel address mapping.

⚠	**If you get the error message “invalid combination” after the MMU.FORMAT command, check if you have enabled the MMUSPACES.**

Moreover, you need to set the common address range with the command **TRANSlation.COMMON**. This is actually the common address range for all processes and is everything above the process address range e.g.

```
TRANSlation.COMMON 0xbf000000--0xffffffff ;for kernel modules below the
                                          ;kernel
TRANSlation.COMMON 0xc0000000--0xffffffff ;for kernel modules above the
                                          ;kernel
```

Finally you need to enable the MMU table walk with **TRANSlation.TableWalk ON** and enable the debugger address translation with the command **TRANSlation.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, the debugger tries to access the kernel MMU tables to get the corresponding physical address. In this case, scanning the MMU is no longer necessary.

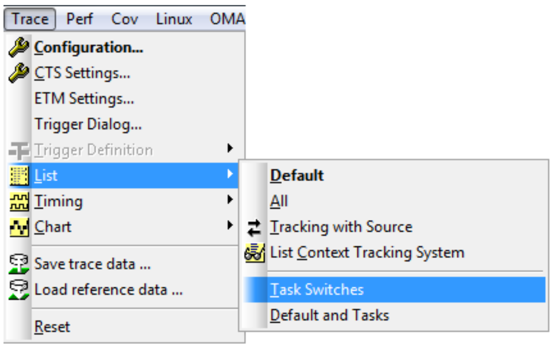
8.) Setup Linux awareness

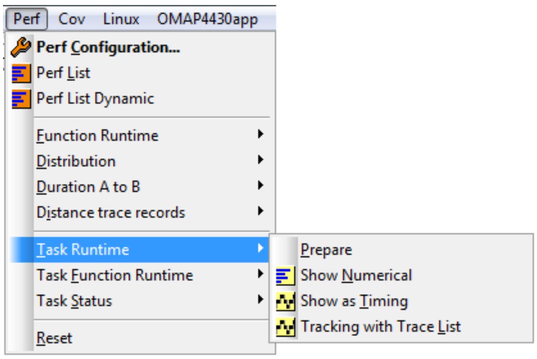
The TRACE32 Linux awareness based on file linux.t32 (updates only on CD or on request) should be loaded now. Load the awareness and its menu and add the Linux awareness manual to the help filter.

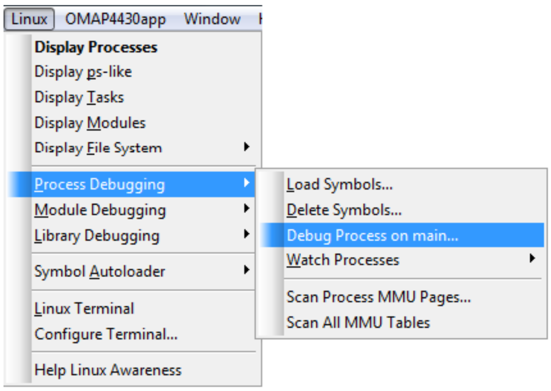
```
TASK.CONFIG ~~/demo/<arch>/kernel/linux/linux.t32 ; load the awareness
MENU.ReProgram ~~/demo/<arch>/kernel/linux/linux.men ; load Linux menu
HELP.FILTER.Add rtoslinux ; add linux awareness manual to help filter
```

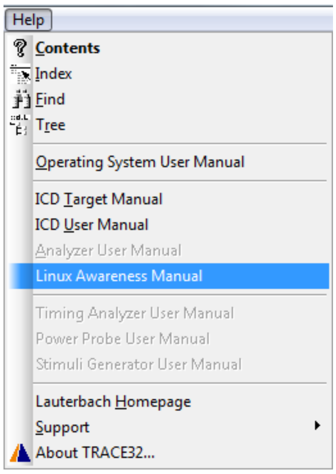
The files are located in you TRACE32 installation directory under “\demo\<arch>\kernel\linux\“ where <arch> if e.g. arm for the ARM architecture. The Linux menu file includes many useful menu items developed for the TRACE32-GUI to ease Linux-Debugging.

All menus are based on the script "linux.men". We will have a look on this script later to see what is done by TRACE32 if you use them.









9.) Setup symbol autoloader

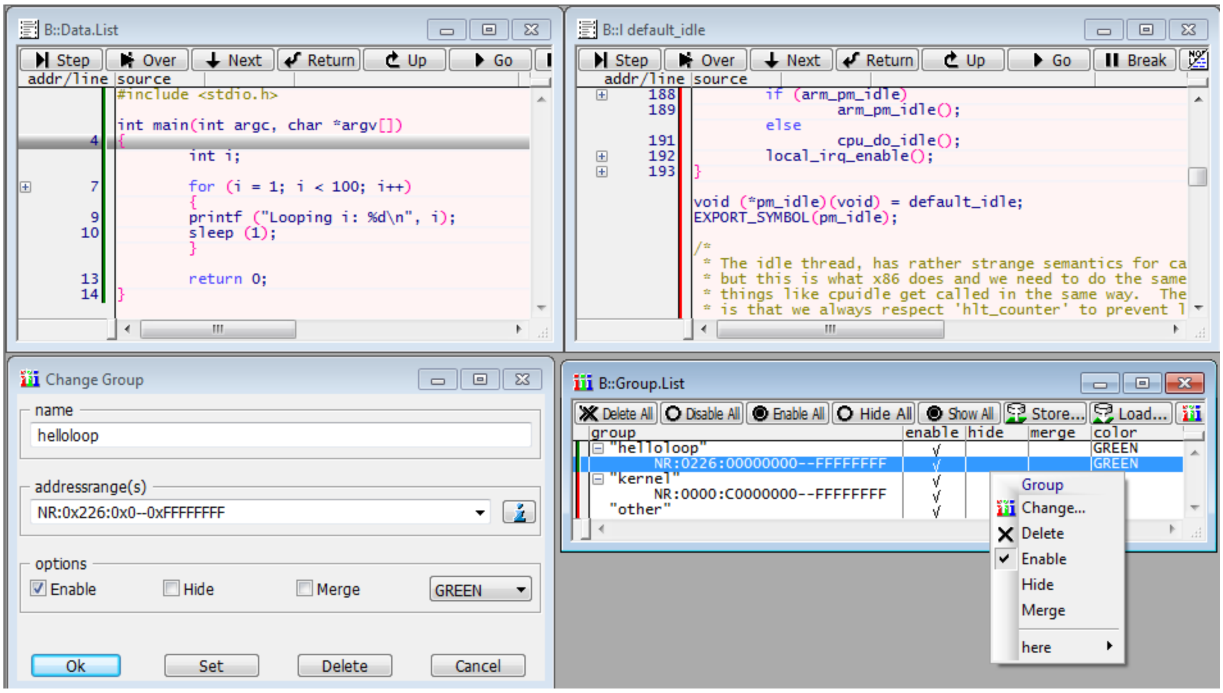
You can specify a script that will be called when a process, kernel module or library is loaded the first time, to load the corresponding symbols. This script, "autoload.cmm" is provided by Lauterbach and is located under "\demo\<arch>\kernel\linux\". You can use the following command to setup the autoloader script. **Don't erase the blank behind "autoload ".**

```
sYmbol.AutoLoad.CHECKLINUX "do ~~/demo/<arch>/kernel/linux/autoload "
```

10.) Mark the Kernel Address Space

For better visibility, you can mark the kernel address space to be displayed with a red bar.
Application addresses can only be marked if the process has started, the process MMU table is valid and the process id (pid) / TRACE32 space id has been assigned!

```
GROUP.Create "kernel"     0x0000:0xC0000000--0xFFFFFFFF /RED
GROUP.Create "helloloop"  0x02BF:0x00000000--0xFFFFFFFF /GREEN
```



```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;

        for (i = 1; i < 100; i++)
        {
        printf ("Looping i: %d\n", i);
        sleep (1);
        }

        return 0;
}
```

```
                if (arm_pm_idle)
                        arm_pm_idle();
                else
                        cpu_do_idle();
                local_irq_enable();
}

void (*pm_idle)(void) = default_idle;
EXPORT_SYMBOL(pm_idle);

/*
 * The idle thread, has rather strange semantics for ca
 * but this is what x86 does and we need to do the same
 * things like cpuidle get called in the same way.  The
 * is that we always respect 'hlt_counter' to prevent l
```

name: helloloop

addressrange(s): NR:0x226:0x0--0xFFFFFFFF

options: [✓ Enable] Hide Merge GREEN

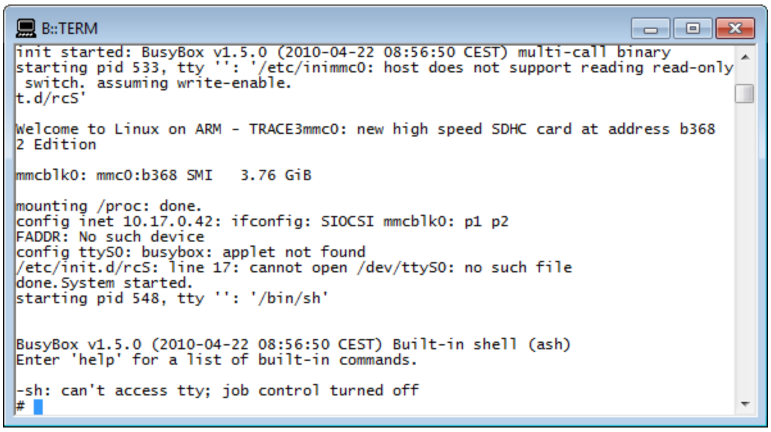
```
group                                enable hide  merge  color
⊟ "helloloop"                          √                 GREEN
     NR:0226:00000000--FFFFFFFF        √                 GREEN
⊟ "kernel"                             √
     NR:0000:C0000000--FFFFFFFF        √
  "other"                              √
```

11.) Boot Linux

Everything is set up now. If the kernel is not already running and if you are not interested in debugging the kernel boot, you can let Linux run as long as it needs to boot completely (e.g. 10 seconds).

```
Go
WAIT 10.s
Break
```

If the kernel boots correctly, you should get a prompt shell in the serial terminal window:



```
init started: BusyBox v1.5.0 (2010-04-22 08:56:50 CEST) multi-call binary
starting pid 533, tty '': '/etc/inimmc0: host does not support reading read-only
 switch. assuming write-enable.
t.d/rcS'

Welcome to Linux on ARM - TRACE3mmc0: new high speed SDHC card at address b368
2 Edition

mmcblk0: mmc0:b368 SMI   3.76 GiB

mounting /proc: done.
config inet 10.17.0.42: ifconfig: SIOCSI mmcblk0: p1 p2
FADDR: No such device
config ttyS0: busybox: applet not found
/etc/init.d/rcS: line 17: cannot open /dev/ttyS0: no such file
done.System started.
starting pid 548, tty '': '/bin/sh'


BusyBox v1.5.0 (2010-04-22 08:56:50 CEST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

-sh: can't access tty; job control turned off
#
```

3.) Example Linux setup-script

You can find here an example Linux script for the Pandaboard. We connect here to the target with SYStem.Up, we use the bootloader to initialize the target, we download the kernel and ramdisk image with the debugger and set the initial values for the CPU registers.

```
 RESet

 ; setup of ICD
 PRINT “initializing...”
 SYStem.CPU OMAP4430
 SYStem.JtagClock CTCK 30MHz
 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          ; my be used by Linux for FPU detection
 SYStem.Option MMUSPACES ON      ; enable space ids to virtual addresses
 SYStem.Up

 ; Open a serial terminal window
 TERM.METHOD COM COM1 115200. 8 NONE 1STOP NONE
 TERM.MODE VT100
 TERM.SCROLL ON
 TERM.SIZE 80. 1000
 WINPOS 76.0 25.0 80. 24. 0. 0. TermWin
 SCREEN.ALWAYS
 TERM

 SETUP.IMASKASM ON          ; lock interrupts while single stepping

 ; Let the boot monitor setup the board
    Go
    PRINT "target setup..."
    WAIT 1.s
    Break

; Load the Linux kernel
 Data.LOAD.Elf vmlinux 0x80008000-0xc0008000 /GNU /NoSymbol /NoREG

 Register.RESet

 ; Set PC on start address of image
 Register.Set PC 0x80008000

 ; Set machine type in R1; see arch/arm/tools/mach-types
 Register.Set R1 2791. ; omap4_panda

 DO atag_list.cmm ; call script to set R2 and init. the atag structure
```

continued on next page.

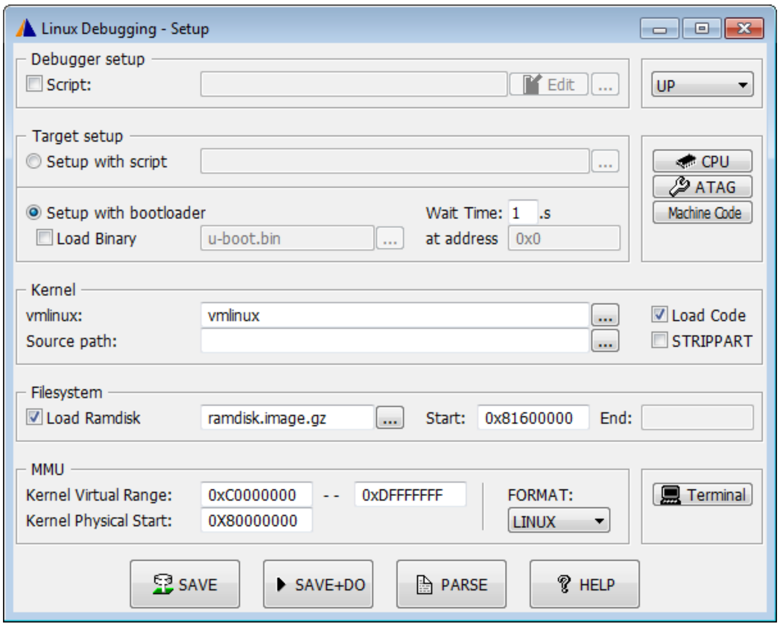
continued:

```
 ; Loading RAM disk
 Data.LOAD.Binary ramdisk.image.gz 0x81600000 /NoClear /NoSymbol

 ; Load the Linux kernel symbols into the debugger
 Data.LOAD.Elf vmlinux /GNU /NoCODE /STRIPPART 3.

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

 PRINT "initializing debugger MMU..."
 MMU.FORMAT LINUX swapper_pg_dir 0xc0000000--0xdfffffff 0x80000000
 TRANSLATION.COMMON 0xbf000000--0ffffffff
 TRANSLATION.TableWalk ON
 TRANSlation.ON

; Initialize Linux Awareness
 PRINT "initializing multi task support..."
 TASK.CONFIG ~~/demo/arm/kernel/linux/linux      ; loads Linux awareness
 MENU.ReProgram ~~/demo/arm/kernel/linux/linux   ; loads Linux menu
 HELP.FILTER.Add rtoslinux  ; add linux awareness manual to help filter

 sYmbol.Autoload.CHECKLINUX "do ~~/demo/arm/kernel/linux/autoload.cmm "

 Go
 WAIT 5.s
 Break
```

4.) Linux Debugging Dialog

The Linux script can be setup using the Linux Debugging dialog. This dialog can be opened by calling the script linux_dialog.cmm that can be fount under demo\<arch>\kenel\linux e.g.

```
DO ~~\demo\arm\kernel\linux\linux_dialog.cmm
```



Debugger setup
☐ Script:
UP

Target setup
○ Setup with script
● Setup with bootloader — Wait Time: 1 .s
☐ Load Binary: u-boot.bin — at address 0x0

Kernel
vmlinux: vmlinux — ☑ Load Code
Source path: — ☐ STRIPPART

Filesystem
☑ Load Ramdisk: ramdisk.image.gz — Start: 0x81600000 — End:

MMU
Kernel Virtual Range: 0xC0000000 - - 0xDFFFFFFF
Kernel Physical Start: 0X80000000
FORMAT: LINUX

Debugging Linux components by TRACE32 Linux menu

This chapter will show how to debug the different Linux components explained in the previous chapters. First you will see the easy handling by the special included Linux menu. Then we will have a look on the script "linux.men" which builds the Linux menu to explain how it works. It will give a deeper understanding what happens when you use items of this menu. This might help to understand what is necessary to do if the results are not as expected.

Displaying the source code

Before going through debugging the different Linux components, we will show first an issue which is common to all these components and which is **displaying the source code for loaded symbols**. If you are not running TRACE32 on the host where you compiled your object files (e.g. bootloader, kernel..) the debugger, which uses per default the compile path to find the source files, will not find these files. Thus, the debugger needs to be set up to be able to find the source code. Two options are available in the **Data.LOAD** command for this purpose: **/STRIPPART** and **/SourcePATH.** With the option **/STRIPPART** you can remove parts of the path stored in the object file. With the option **/SourcePATH** you can specify a basis directory for the source files. The debugger will take the rest of the compile path (without the stripped part) and will append it to the specified source path. The source path can also be set using the command **sYmbol.SourcePATH**.

For example, if you compiled our kernel on a Linux machine in the directory `/home/user/linux-kernel/linux-3.4` and you are running TRACE32 on a Windows machine where you have the kernel source files tree under `C:\Debugging\Linux\Sources\linux-3.4`, you can for example load the kernel symbols with

```
Data.LOAD.Elf vmlinux /NoCODE /STRIPPART "linux-kernel"
                                /SourcePATH C:\Debugging\Linux\Sources
```

or

```
Data.LOAD.Elf vmlinux /NoCODE /STRIPPART 4.
                               /SourcePATH  C:\Debugging\Linux\Sources
```

or

```
Data.LOAD.Elf vmlinux /NoCODE /STRIPPART 4.
sYmbol.SourcePATH C:\Debugging\Linux\Sources
```

In our case we get the same result if we use the /STRIPPART option with the parameter “linux-kernel” or 4. (“/”+”home/”+”user/”+”linux-kernel/”).

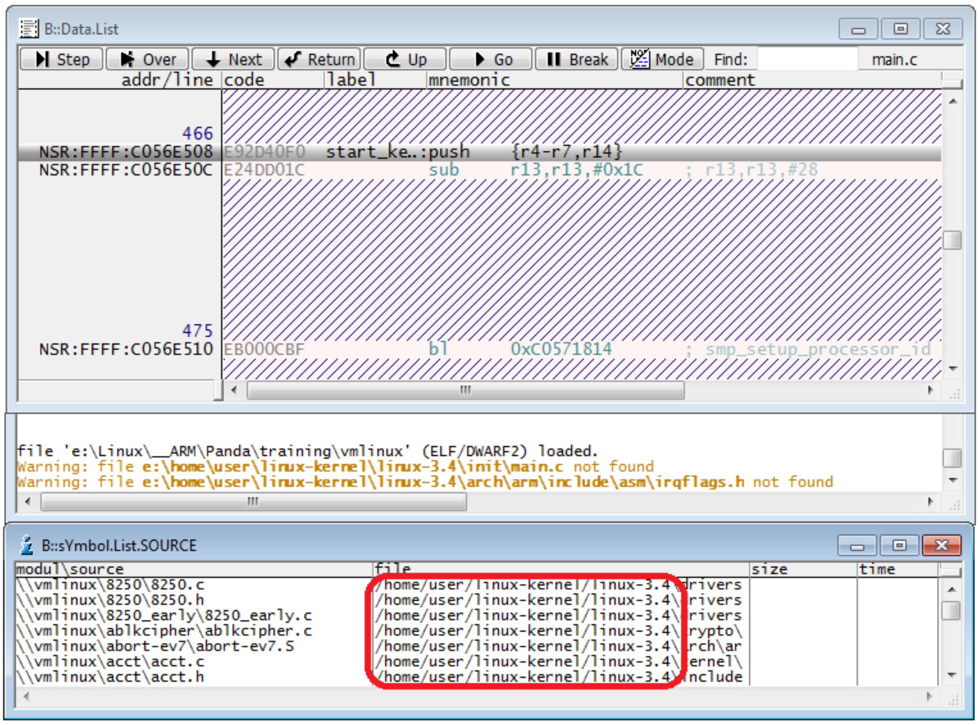
To look for the source file init/main.c the debugger will here use the path

`C:\Debugging\Linux\Sources\ + ~~/home/user/linux-kernel~~/linux-3.4/init/`

thus

`C:\Debugging\Linux\Sources\linux-3.4\init`

1. Data.LOAD.Elf vmlinux /NoCODE

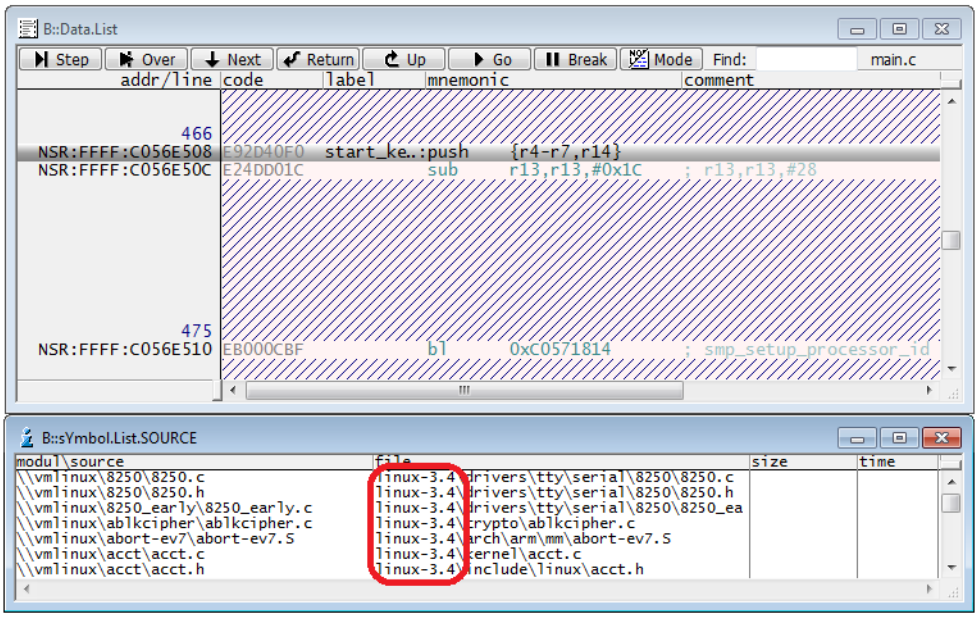


```
addr/line         code      label           mnemonic              comment
              466
NSR:FFFF:C056E508 E92D40F0  start_ke..:push  {r4-r7,r14}
NSR:FFFF:C056E50C E24DD01C             sub   r13,r13,#0x1C        ; r13,r13,#28
              475
NSR:FFFF:C056E510 EB000CBF             bl    0xC0571814           ; smp_setup_processor_id
```

```
file 'e:\Linux\__ARM\Panda\training\vmlinux' (ELF/DWARF2) loaded.
Warning: file e:\home\user\linux-kernel\linux-3.4\init\main.c not found
Warning: file e:\home\user\linux-kernel\linux-3.4\arch\arm\include\asm\irqflags.h not found
```

```
modul\source                          file                                   size    time
\\vmlinux\8250\8250.c                 /home/user/linux-kernel/linux-3.4\drivers
\\vmlinux\8250\8250.h                 /home/user/linux-kernel/linux-3.4\rivers
\\vmlinux\8250_early\8250_early.c     /home/user/linux-kernel/linux-3.4\rivers
\\vmlinux\ablkcipher\ablkcipher.c     /home/user/linux-kernel/linux-3.4\rypto\
\\vmlinux\abort-ev7\abort-ev7.S       /home/user/linux-kernel/linux-3.4\rch\ar
\\vmlinux\acct\acct.c                 /home/user/linux-kernel/linux-3.4\ernel\
\\vmlinux\acct\acct.h                 /home/user/linux-kernel/linux-3.4\nclude
```

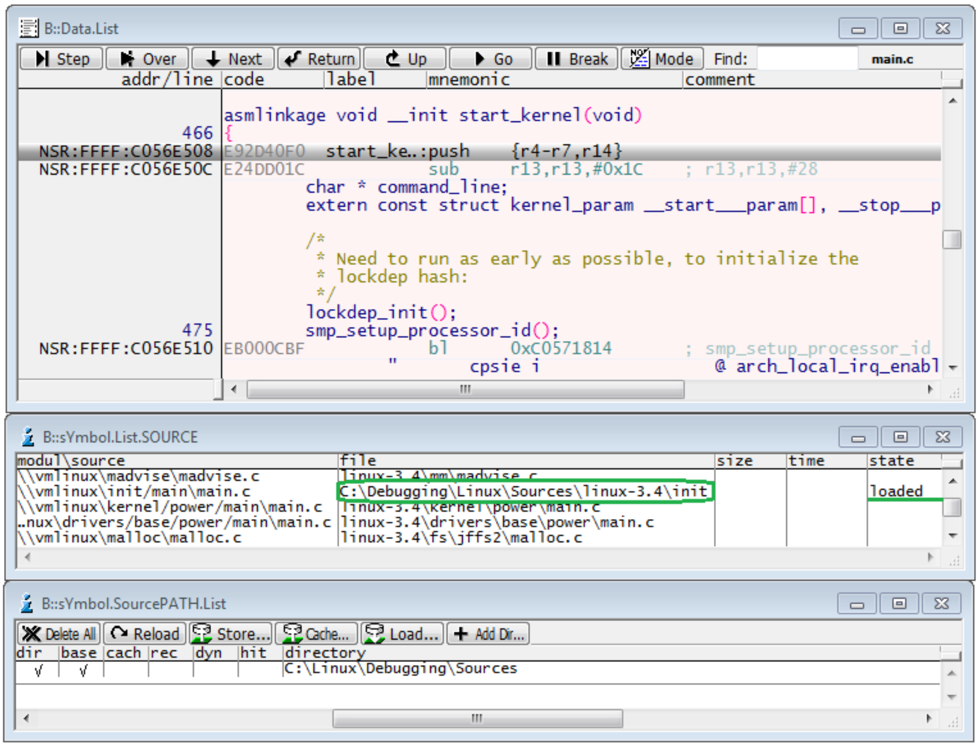
2. Data.LOAD.Elf vmlinux /NoCODE /STRIPPART 4.



```
addr/line         code      label           mnemonic              comment
              466
NSR:FFFF:C056E508 E92D40F0  start_ke..:push  {r4-r7,r14}
NSR:FFFF:C056E50C E24DD01C             sub   r13,r13,#0x1C        ; r13,r13,#28
              475
NSR:FFFF:C056E510 EB000CBF             bl    0xC0571814           ; smp_setup_processor_id
```

```
modul\source                          file                                   size    time
\\vmlinux\8250\8250.c                 linux-3.4\drivers\tty\serial\8250\8250.c
\\vmlinux\8250\8250.h                 linux-3.4\drivers\tty\serial\8250\8250.h
\\vmlinux\8250_early\8250_early.c     linux-3.4\drivers\tty\serial\8250\8250_ea
\\vmlinux\ablkcipher\ablkcipher.c     linux-3.4\crypto\ablkcipher.c
\\vmlinux\abort-ev7\abort-ev7.S       linux-3.4\arch\arm\mm\abort-ev7.S
\\vmlinux\acct\acct.c                 linux-3.4\kernel\acct.c
\\vmlinux\acct\acct.h                 linux-3.4\include\linux\acct.h
```

3. sYmbol.SourcePATH C:\Debugging\Linux\Sources


B::Data.List

Step | Over | Next | Return | Up | Go | Break | Mode | Find: | main.c

```
          addr/line code      label      mnemonic           comment
                    asmlinkage void __init start_kernel(void)
                466 {
 NSR:FFFF:C056E508 E92D40F0  start_ke..:push    {r4-r7,r14}
 NSR:FFFF:C056E50C E24DD01C             sub     r13,r13,#0x1C     ; r13,r13,#28
                              char * command_line;
                              extern const struct kernel_param __start___param[], __stop___p

                              /*
                               * Need to run as early as possible, to initialize the
                               * lockdep hash:
                               */
                              lockdep_init();
                475           smp_setup_processor_id();
 NSR:FFFF:C056E510 EB000CBF             bl      0xC0571814        ; smp_setup_processor_id
                                 "         cpsie i                @ arch_local_irq_enabl
```

B::sYmbol.List.SOURCE

```
modul\source                      file                                     size   time   state
\\vmlinux\madvise\madvise.c       linux-3.4\mm\madvise.c
\\vmlinux\init/main\main.c        C:\Debugging\Linux\Sources\linux-3.4\init              loaded
\\vmlinux\kernel/power/main\main.c linux-3.4\kernel\power\main.c
..nux\drivers/base/power/main\main.c linux-3.4\drivers\base\power\main.c
\\vmlinux\malloc\malloc.c         linux-3.4\fs\jffs2\malloc.c
```

B::sYmbol.SourcePATH.List

Delete All | Reload | Store... | Cache... | Load... | Add Dir...

```
dir  base cach rec  dyn  hit  directory
 √    √                       C:\Linux\Debugging\Sources
```

For more information about displaying the source core, please refer to the ICD manual (training_icd.pdf).

1.) 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. If you want to debug different components at once, you have to aggregate the commands for the components.

When using several symbol files at once (e.g. kernel, processes and libraries), it is convenient to use the **GROUP** command to mark each component with an own color. Find some example settings below.

The **"RTOS Debugger for Linux - Stop Mode"** (rtos_linux_stop.pdf) gives additional detailed instructions.

a) Bootloader

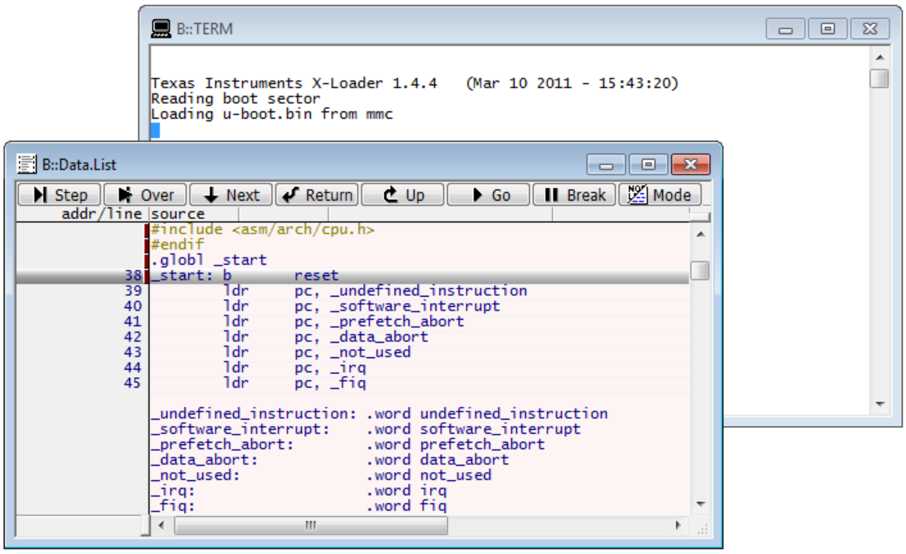
Debugging the bootloader is an easy task. The bootloader is usually stored in Flash and runs in physical address space. It is built as one complete linked image, containing everything you need. Of course you need to build an image containing debug symbols for debugging on High-Level-Language (HLL). Convert this image into a file format suitable for burning into the flash and burn it.

Right after initializing your debugger, load the symbols (only) into the debugger. If you are debugging inside a Flash memory range, remember to map onchip breakpoints for the range or use onchip breakpoints there. A simple debugger startup with the boot loader in Flash could look like this:

```
MAP.BOnchip 0xff000000--0xffffffff    ; use on-chip breakpoints in Flash

SYStem.Up                             ; initialize debugger and reset CPU

Data.LOAD.Elf u-boot /NoCODE          ; load symbols of boot loader
```

Moreover, on an SMP system the bootloader runs only on the first core. The other cores generally wait until they are started by the kernel.

On our Pandaboard, after **SYStem.Up**, the first core is stopped at the Boot ROM which tries to load from the SD card the first bootloader (X-Loader) into the SRAM. The X-Loader initializes then the SDRAM and loads u-boot from the SD card there. The following screen shot shows the first instruction of the u-boot:



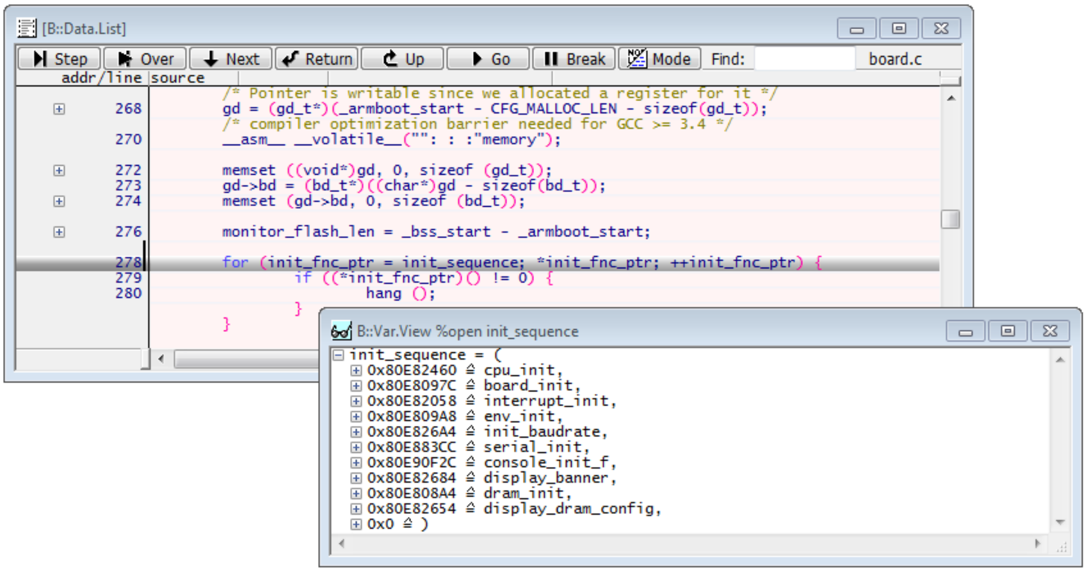
```
Texas Instruments X-Loader 1.4.4   (Mar 10 2011 - 15:43:20)
Reading boot sector
Loading u-boot.bin from mmc
```

```
#include <asm/arch/cpu.h>
#endif
.globl _start
_start: b       reset
        ldr     pc, _undefined_instruction
        ldr     pc, _software_interrupt
        ldr     pc, _prefetch_abort
        ldr     pc, _data_abort
        ldr     pc, _not_used
        ldr     pc, _irq
        ldr     pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:        .word prefetch_abort
_data_abort:            .word data_abort
_not_used:              .word not_used
_irq:                   .word irq
_fiq:                   .word fiq
```

The script we use here to debug u-boot is pretty simple:

```
SYStem.CPU OMAP4430
SYStem.Up
Data.LOAD.Elf u-boot /NoCODE /STRIPPART 3.
Go _start /ONCHIP
```

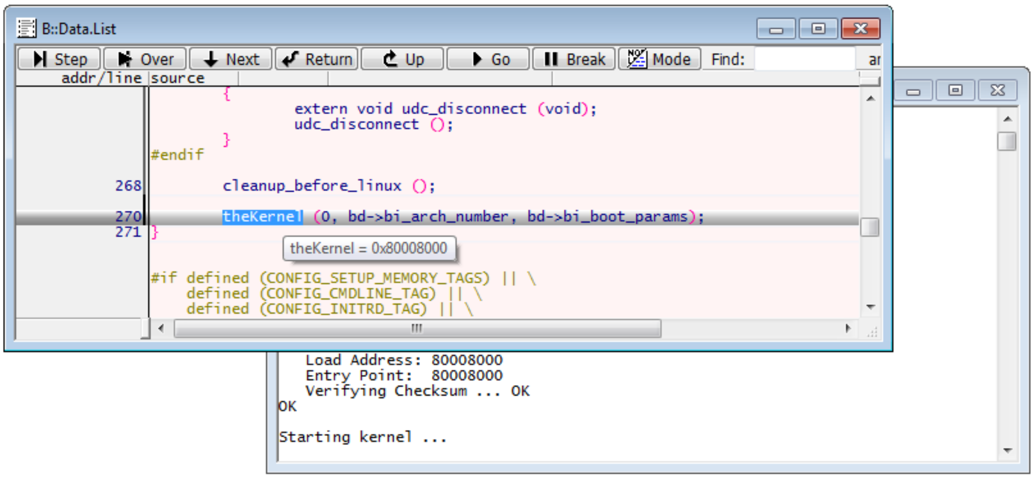
You can e.g. debug the bootloader to check the hardware initialization



```
        /* Pointer is writable since we allocated a register for it */
        gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
        /* compiler optimization barrier needed for GCC >= 3.4 */
        __asm__ __volatile__("": : :"memory");

        memset ((void*)gd, 0, sizeof (gd_t));
        gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
        memset (gd->bd, 0, sizeof (bd_t));

        monitor_flash_len = _bss_start - _armboot_start;

        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
                if ((*init_fnc_ptr)() != 0) {
                        hang ();
                }
        }
```

```
init_sequence = (
  0x80E82460 ≙ cpu_init,
  0x80E8097C ≙ board_init,
  0x80E82058 ≙ interrupt_init,
  0x80E809A8 ≙ env_init,
  0x80E826A4 ≙ init_baudrate,
  0x80E883CC ≙ serial_init,
  0x80E90F2C ≙ console_init_f,
  0x80E82684 ≙ display_banner,
  0x80E808A4 ≙ dram_init,
  0x80E82654 ≙ display_dram_config,
  0x0 ≙ )
```

After initializing the target and loading the kernel, the bootloader jumps to the kernel start



```
        {
                extern void udc_disconnect (void);
                udc_disconnect ();
        }
#endif

        cleanup_before_linux ();

        theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
}

#if defined (CONFIG_SETUP_MEMORY_TAGS) || \
    defined (CONFIG_CMDLINE_TAG) || \
    defined (CONFIG_INITRD_TAG) || \
```

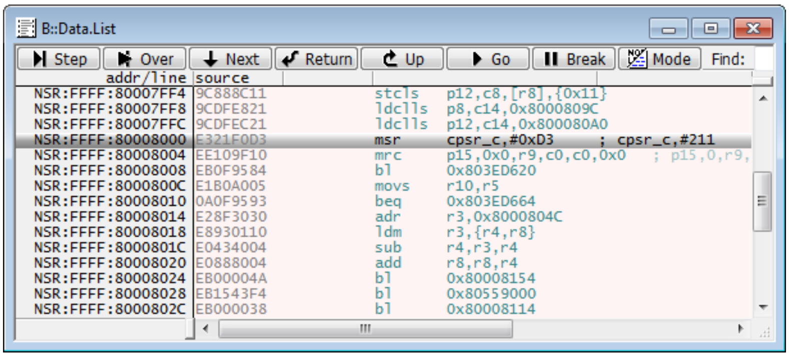
```
    Load Address: 80008000
    Entry Point:  80008000
    Verifying Checksum ... OK
OK

Starting kernel ...
```

It is not unusual that the bootloader relocates its code. In this case, you need to take care that the symbols are loaded on the correct addresses otherwise breakpoints could fail: onchip breakpoints will not hit and software breakpoints will hit but you will see a “bkpt #0x0” instruction and no source code.

Kernel startup

The bootloader jumps into the kernel startup routine. It starts at physical address space, does some initialization and sets up the MMU. **Finally the kernel startup switches into virtual address space** and does the rest of the initialization.

The complete kernel startup is part of the vmlinux image. As a such, it's symbols are bound to virtual addresses. So if you load the kernel symbols without specifying any address, you will see no symbol information:



```
          addr/line source
  NSR:FFFF:80007FF4 9C888C11        stcls   p12,c8,[r8],{0x11}
  NSR:FFFF:80007FF8 9CDFE821        ldclls  p8,c14,0x8000809C
  NSR:FFFF:80007FFC 9CDFEC21        ldclls  p12,c14,0x800080A0
  NSR:FFFF:80008000 E321F0D3        msr     cpsr_c,#0xD3      ; cpsr_c,#211
  NSR:FFFF:80008004 EE109F10        mrc     p15,0x0,r9,c0,c0,0x0   ; p15,0,r9,
  NSR:FFFF:80008008 EB0F9584        bl      0x803ED620
  NSR:FFFF:8000800C E1B0A005        movs    r10,r5
  NSR:FFFF:80008010 0A0F9593        beq     0x803ED664
  NSR:FFFF:80008014 E28F3030        adr     r3,0x8000804C
  NSR:FFFF:80008018 E8930110        ldm     r3,{r4,r8}
  NSR:FFFF:8000801C E0434004        sub     r4,r3,r4
  NSR:FFFF:80008020 E0888004        add     r8,r8,r4
  NSR:FFFF:80008024 EB00004A        bl      0x80008154
  NSR:FFFF:80008028 EB1543F4        bl      0x80559000
  NSR:FFFF:8000802C EB000038        bl      0x80008114
```

To debug the tiny startup sequence running in physical address range, you need to relocate the virtual symbols to the physical addresses:

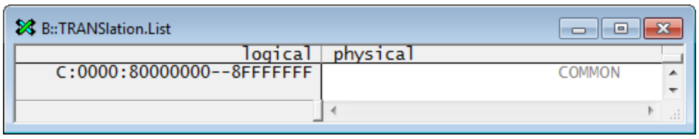
Data.LOAD.Elf vmlinux *<phys. start addr>-<virtual start addr>* **/NoCODE**

E.g., if the kernel runs in virtual address 0xC0000000, and the physical memory starts at 0x80000000, you have to shift the addresses by 0x80000000-0xC0000000:

```
Data.LOAD.Elf vmlinux 0x80000000-0xC0000000 /NoCODE
```

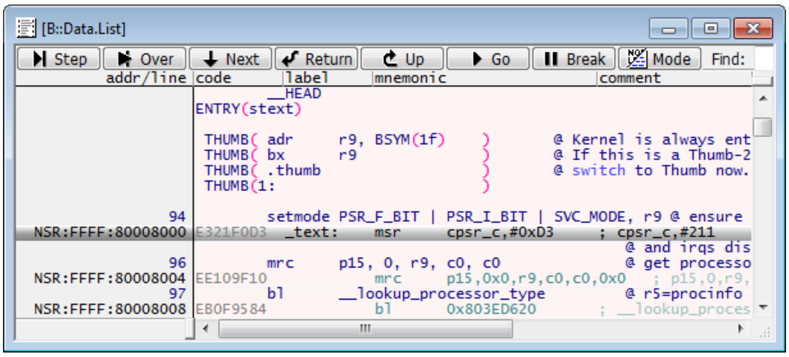
If the address extension has already been enabled (**SYStem.Option MMUSPACES ON**), then you need to declare the startup of the kernel as common area in the debugger translation table to be able to see the debug information. However, you have to undo this after the MMU is enabled.

```
TRANSlation.COMMON 0x80000000--0x8fffffff
```



```
                 logical  physical
 C:0000:80000000--8FFFFFFF                    COMMON
```

Now, you can see the symbol information:



```
         addr/line code         label      mnemonic                         comment
                                  __HEAD
                   ENTRY(stext)

                    THUMB( adr      r9, BSYM(1f)     )       @ Kernel is always ent
                    THUMB( bx       r9               )       @ If this is a Thumb-2
                    THUMB( .thumb                    )       @ switch to Thumb now.
                    THUMB(1:                         )

                94          setmode PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9 @ ensure
 NSR:FFFF:80008000 E321F0D3  _text:     msr      cpsr_c,#0xD3       ; cpsr_c,#211
                                                                  @ and irqs dis
                96          mrc      p15, 0, r9, c0, c0           @ get processo
 NSR:FFFF:80008004 EE109F10             mrc      p15,0x0,r9,c0,c0,0x0   ; p15,0,r9,
                97          bl       __lookup_processor_type      @ r5=procinfo
 NSR:FFFF:80008008 EB0F9584             bl       0x803ED620         ; __lookup_proces
```

Please note, that as long as the debugger MMU is not set up, it is recommended to use onchip breakpoints:

```
Break.Set rest_init /Onchip
```

Kernel Boot

After enabling the MMU, the kernel startup code switches to virtual addreses. From now on, the Linux kernel runs completely in virtual address space. The symbols are all bound to these virtual addresses, so simply load the Linux symbols as they are. As already noted in a previous chapter, some structure information needs to be fixed by a “clean-up”

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

sYmbol.CLEANUP                          ; get rid of useless symbols
```

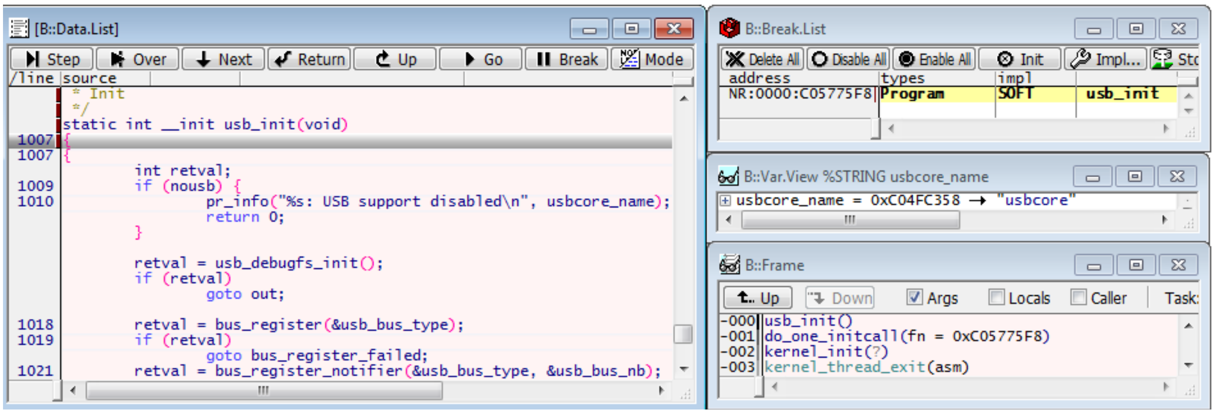
As already explained, the debugger needs to know the MMU translation map for the kernel, the address of the kernel translation table and the MMU format used by the kernel. By declaring the kernel range as COMMON, the loaded symbols will be displayed for all space ids.

```
MMU.FORMAT Linux swapper_pg_dir 0xC0000000--0xC1FFFFFF 0x00000000
TRANSlation.COMMON 0xC0000000--0xFFFFFFFF
TRANSlation.ON
```

For your convenience, you may mark the complete kernel address space with a red bar:

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

You can use here software breakpoints since all the kernel code is already loaded and has a translation map.



```
[B::Data.List]
/line source
 * Init
 */
static int __init usb_init(void)
{
{
        int retval;
        if (nousb) {
                pr_info("%s: USB support disabled\n", usbcore_name);
                return 0;
        }

        retval = usb_debugfs_init();
        if (retval)
                goto out;

        retval = bus_register(&usb_bus_type);
        if (retval)
                goto bus_register_failed;
        retval = bus_register_notifier(&usb_bus_type, &usb_bus_nb);
```

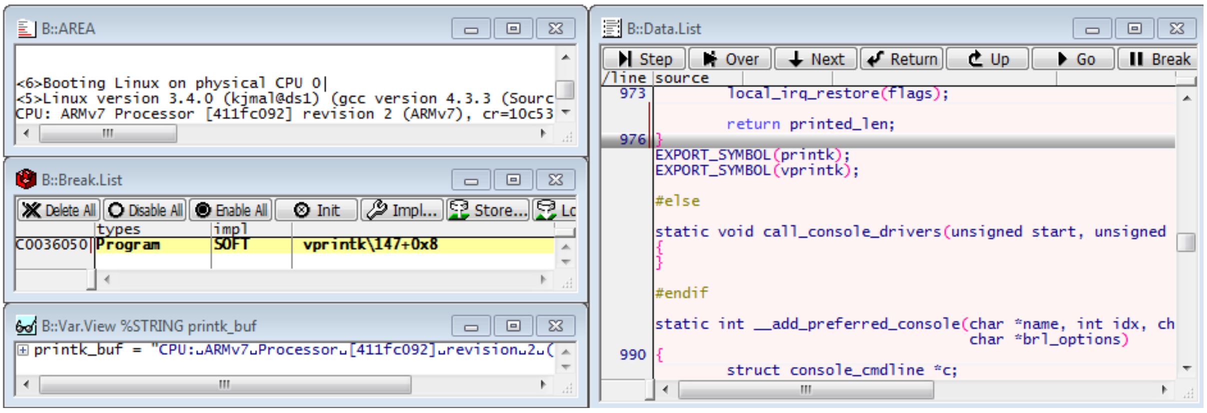
```
B::Break.List
address              types    impl
NR:0000:C05775F8     Program  SOFT   usb_init
```

```
B::Var.View %STRING usbcore_name
usbcore_name = 0xC04FC358 → "usbcore"
```

```
B::Frame
-000 usb_init()
-001 do_one_initcall(fn = 0xC05775F8)
-002 kernel_init(?)
-003 kernel_thread_exit(asm)
```

If your kernel does not boot correctly, you can look for the last message in the terminal window and debug the corresponding function. If you have no terminal output because the error happens before the serial console has been initialized or because there is a problem in the serial driver, you can set a breakpoint at the end of the `vprintk` function and watch the `printk_buf` buffer. You can for example print the buffer to the AREA window using the following script.

```
Break.Delete
Break.Set SYMBOL.EXIT(vprintk)
Go
AREA.CLEAR
ON.PBREAK GOSUB
(
  Var.PRINT %STRING printk_buf
  SCREEN
  Go
)
STOP
```



```
B::AREA
<6>Booting Linux on physical CPU 0|
<5>Linux version 3.4.0 (kjmal@ds1) (gcc version 4.3.3 (Sourc
CPU: ARMv7 Processor [411fc092] revision 2 (ARMv7), cr=10c53
```

```
B::Break.List
          types    impl
C0036050  Program  SOFT   vprintk\147+0x8
```

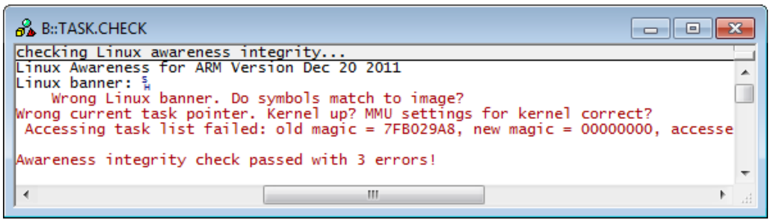
```
B::Var.View %STRING printk_buf
printk_buf = "CPU:␣ARMv7␣Processor␣[411fc092]␣revision␣2␣(
```

```
B::Data.List
/line source
        local_irq_restore(flags);

        return printed_len;
}
EXPORT_SYMBOL(printk);
EXPORT_SYMBOL(vprintk);

#else

static void call_console_drivers(unsigned start, unsigned
{
}

#endif

static int __add_preferred_console(char *name, int idx, ch
                                   char *brl_options)
{
        struct console_cmdline *c;
```

Verifying image and symbols

It is very important that the kernel running on the target is from the **very same build** than the symbol file loaded into the debugger. A typical error is to have a uImage loaded by the bootloader (e.g. from a memory card) and a vmlinux and both files are not from the same build. This can lead to very strange results.

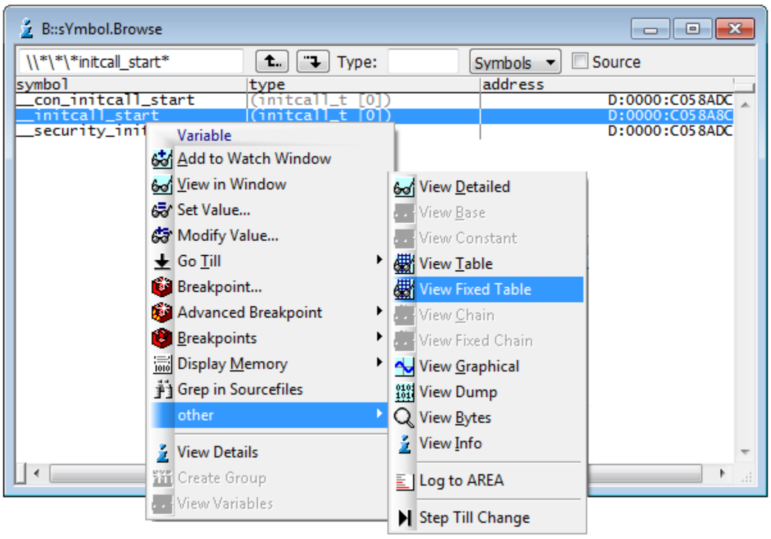
You can check if the kernel code matches the loaded symbols using the **TASK.CHECK** command. First let the kernel boot, stop the target and then execute **TASK.CHECK**. When the symbols does not match the kernel code, you will get an error message in this window:



```
checking Linux awareness integrity...
Linux Awareness for ARM Version Dec 20 2011
Linux banner: 
    Wrong Linux banner. Do symbols match to image?
Wrong current task pointer. Kernel up? MMU settings for kernel correct?
 Accessing task list failed: old magic = 7FB029A8, new magic = 00000000, accesse

Awareness integrity check passed with 3 errors!
```

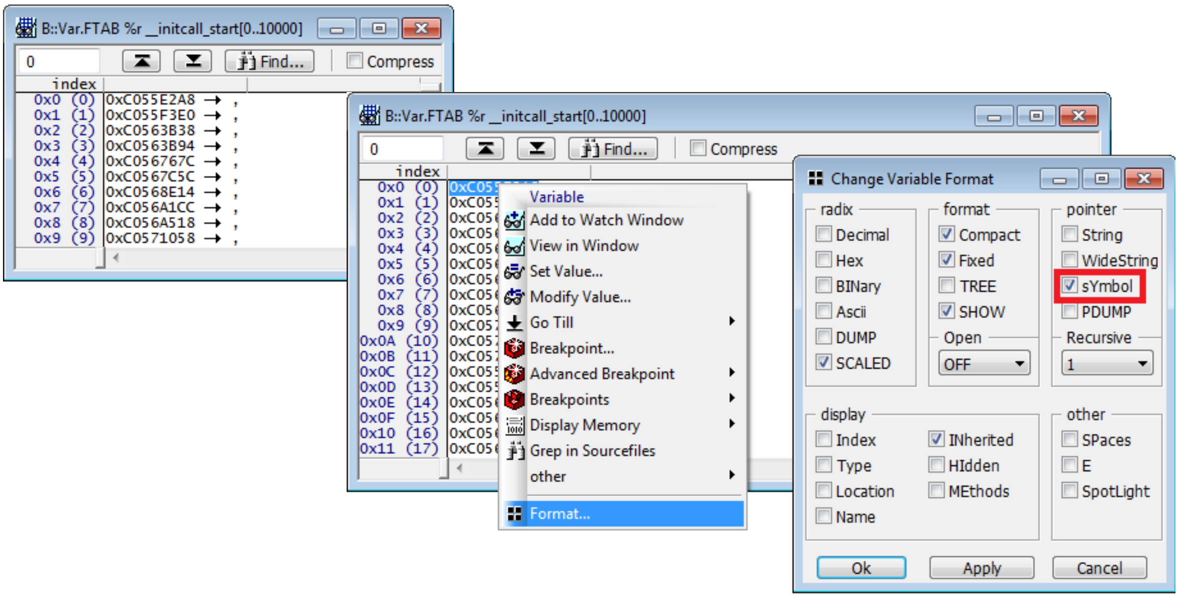
Example: debugging built-in device drivers

Most of the built-in device drivers are registered in the initcall table. You can search for the __initcall_start label in the **sYmbol.Browse** window and view it as a fixed table:



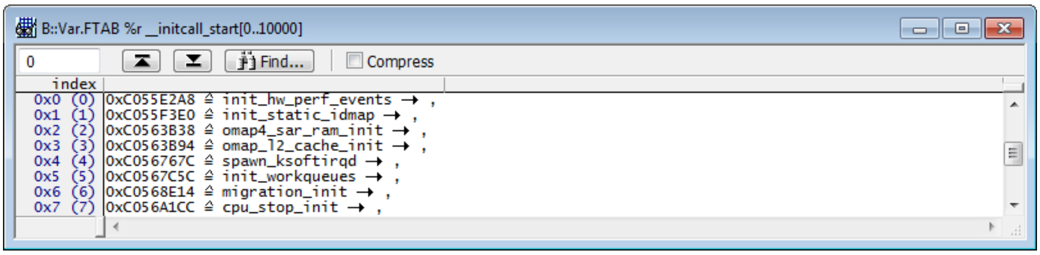
```
symbol               type              address
__con_initcall_start (initcall_t [0])  D:0000:C058ADC
__initcall_start     (initcall_t [0])  D:0000:C058A8C
__security_init                        D:0000:C058ADC
```

You will get a table with the start addresses of the device drivers init functions you can display the names of the functions by checking the sYmbol item in “pointer” category:



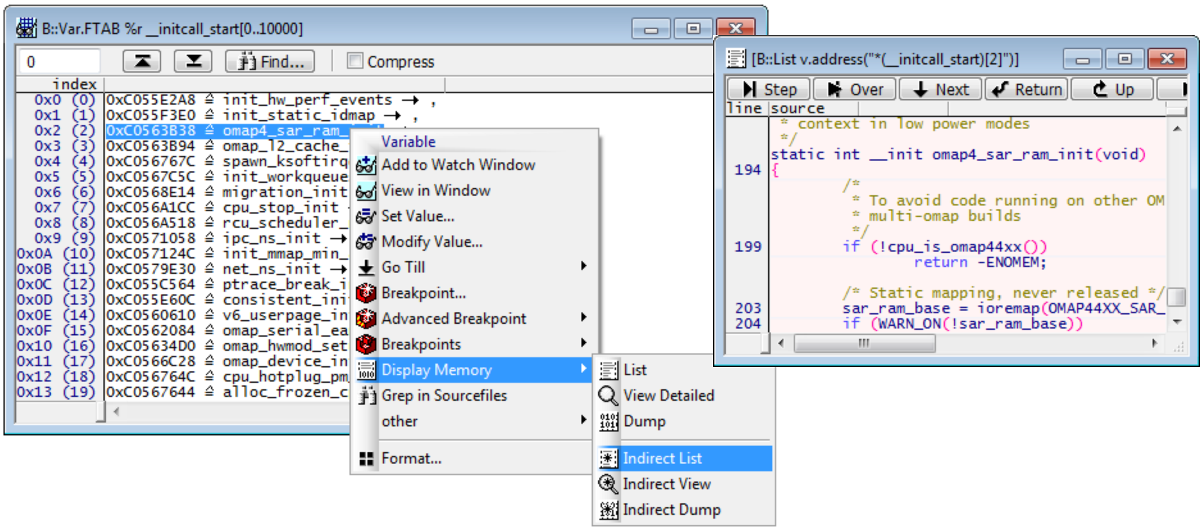
```
index
0x0  (0)  0xC055E2A8 → ,
0x1  (1)  0xC055F3E0 → ,
0x2  (2)  0xC0563B38 → ,
0x3  (3)  0xC0563B94 → ,
0x4  (4)  0xC056767C → ,
0x5  (5)  0xC0567C5C → ,
0x6  (6)  0xC0568E14 → ,
0x7  (7)  0xC056A1CC → ,
0x8  (8)  0xC056A518 → ,
0x9  (9)  0xC0571058 → ,
```

radix: Decimal [ ], Hex [ ], BINary [ ], Ascii [ ], DUMP [ ], SCALED [✓]
format: Compact [✓], Fixed [✓], TREE [ ], SHOW [✓]; Open: OFF
pointer: String [ ], WideString [ ], sYmbol [✓], PDUMP [ ]; Recursive: 1
display: Index [ ], Type [ ], Location [ ], Name [ ], INherited [✓], HIdden [ ], MEthods [ ]
other: SPaces [ ], E [ ], SpotLight [ ]



```
index
0x0  (0)  0xC055E2A8 ≙ init_hw_perf_events → ,
0x1  (1)  0xC055F3E0 ≙ init_static_idmap → ,
0x2  (2)  0xC0563B38 ≙ omap4_sar_ram_init → ,
0x3  (3)  0xC0563B94 ≙ omap_l2_cache_init → ,
0x4  (4)  0xC056767C ≙ spawn_ksoftirqd → ,
0x5  (5)  0xC0567C5C ≙ init_workqueues → ,
0x6  (6)  0xC0568E14 ≙ migration_init → ,
0x7  (7)  0xC056A1CC ≙ cpu_stop_init → ,
```

By selecting “Display Memory” -> “Indirect List”, you can display the source code of a specific function in the list:



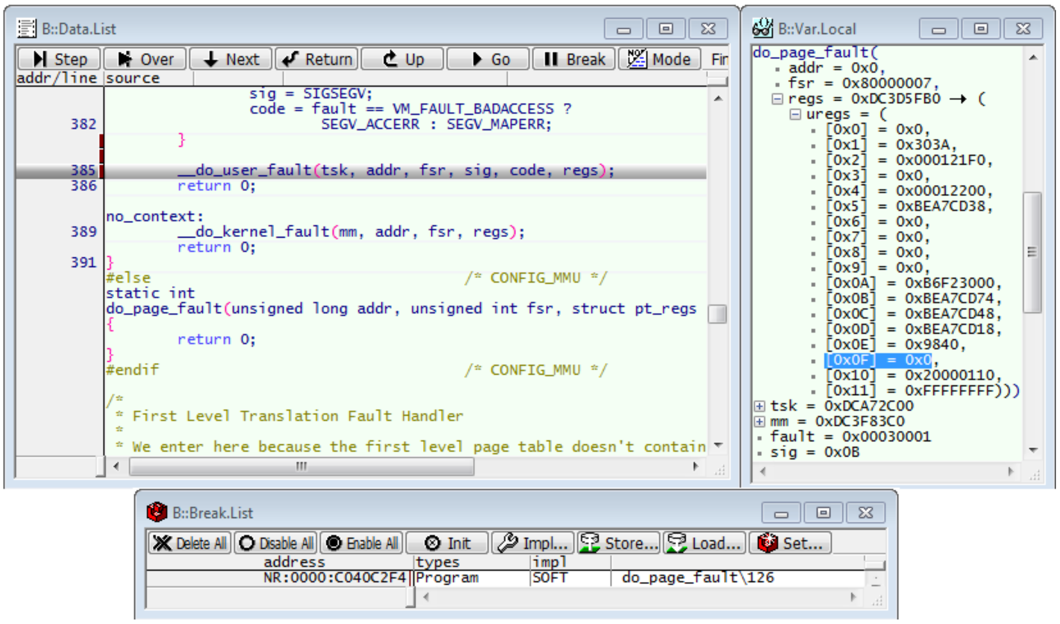
```
index
0x0  (0)  0xC055E2A8 ≙ init_hw_perf_events → ,
0x1  (1)  0xC055F3E0 ≙ init_static_idmap → ,
0x2  (2)  0xC0563B38 ≙ omap4_sar_ram_
0x3  (3)  0xC0563B94 ≙ omap_l2_cache_
0x4  (4)  0xC056767C ≙ spawn_ksoftirq
0x5  (5)  0xC0567C5C ≙ init_workqueue
0x6  (6)  0xC0568E14 ≙ migration_init
0x7  (7)  0xC056A1CC ≙ cpu_stop_init
0x8  (8)  0xC056A518 ≙ rcu_scheduler_
0x9  (9)  0xC0571058 ≙ ipc_ns_init →
0x0A (10) 0xC057124C ≙ init_mmap_min_
0x0B (11) 0xC0579E30 ≙ net_ns_init →
0x0C (12) 0xC055C564 ≙ ptrace_break_i
0x0D (13) 0xC055E60C ≙ consistent_ini
0x0E (14) 0xC0560610 ≙ v6_userpage_in
0x0F (15) 0xC0562084 ≙ omap_serial_ea
0x10 (16) 0xC05634D0 ≙ omap_hwmod_set
0x11 (17) 0xC0566C28 ≙ omap_device_in
0x12 (18) 0xC056764C ≙ cpu_hotplug_pm
0x13 (19) 0xC0567644 ≙ alloc_frozen_c
```

```
line source
      * context in low power modes
      */
     static int __init omap4_sar_ram_init(void)
     {
             /*
              * To avoid code running on other OM
              * multi-omap builds
              */
             if (!cpu_is_omap44xx())
                     return -ENOMEM;

             /* Static mapping, never released */
             sar_ram_base = ioremap(OMAP44XX_SAR_
             if (WARN_ON(!sar_ram_base))
```

Segmentation violation happens if the code tries to access a memory location that cannot be mapped in an appropriate way. E.g. if a process tries to write to a read-only area or if the kernel tries to read from an non existent address. A segmentation violation is de detected inside the kernel routine “do_page_fault”.

Depending on the kernel version, if the mapping of the page fails, the kernel jumps to the label “`bad_area`” or “`__do_user_fault`”/”`__do_kernel_fault`”.

To trap segmentation violation, we set a breakpoint in our example onto the label “`do_page_fault`” before “`__do_user_fault`” is called and look to the structure “regs” which contains the complete register set at the location where the fault occurred.

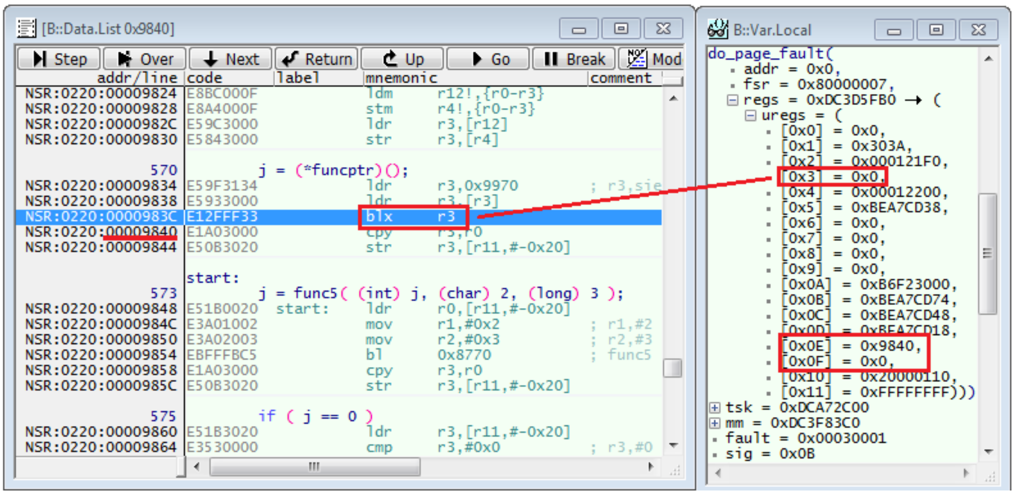


```
addr/line source
                 sig = SIGSEGV;
                 code = fault == VM_FAULT_BADACCESS ?
                        SEGV_ACCERR : SEGV_MAPERR;
          }

          __do_user_fault(tsk, addr, fsr, sig, code, regs);
          return 0;

no_context:
          __do_kernel_fault(mm, addr, fsr, regs);
          return 0;
}
#else                                    /* CONFIG_MMU */
static int
do_page_fault(unsigned long addr, unsigned int fsr, struct pt_regs
{
          return 0;
}
#endif                                   /* CONFIG_MMU */

/*
 * First Level Translation Fault Handler
 *
 * We enter here because the first level page table doesn't contain
```

```
do_page_fault(
  addr = 0x0,
  fsr = 0x80000007,
  regs = 0xDC3D5FB0 → (
    uregs = (
      [0x0] = 0x0,
      [0x1] = 0x303A,
      [0x2] = 0x000121F0,
      [0x3] = 0x0,
      [0x4] = 0x00012200,
      [0x5] = 0xBEA7CD38,
      [0x6] = 0x0,
      [0x7] = 0x0,
      [0x8] = 0x0,
      [0x9] = 0x0,
      [0x0A] = 0xB6F23000,
      [0x0B] = 0xBEA7CD74,
      [0x0C] = 0xBEA7CD48,
      [0x0D] = 0xBEA7CD18,
      [0x0E] = 0x9840,
      [0x0F] = 0x0,
      [0x10] = 0x20000110,
      [0x11] = 0xFFFFFFFF))
  tsk = 0xDCA72C00
  mm = 0xDC3F83C0
  fault = 0x00030001
  sig = 0x0B
```

```
      address            types     impl
      NR:0000:C040C2F4   Program   SOFT     do_page_fault\126
```

The register with the index 15 is the program counter which caused the segmentation violation and the register with the index 14 contains the return address. In out case the segmentation violation is caused by a branch to the address 0x0 which is caused by a zero function pointer.

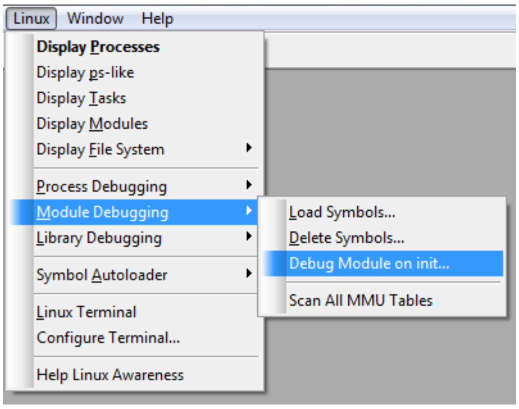


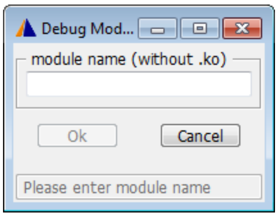
```
      addr/line code     label   mnemonic                    comment
NSR:0220:00009824 E8BC000F       ldm      r12!,{r0-r3}
NSR:0220:00009828 E8A4000F       stm      r4!,{r0-r3}
NSR:0220:0000982C E59C3000       ldr      r3,[r12]
NSR:0220:00009830 E5843000       str      r3,[r4]

                       j = (*funcptr)();
NSR:0220:00009834 E59F3134       ldr      r3,0x9970          ; r3,sig
NSR:0220:00009838 E5933000       ldr      r3,[r3]
NSR:0220:0000983C E12FFF33       blx      r3
NSR:0220:00009840 E1A03000       cpy      r3,r0
NSR:0220:00009844 E50B3020       str      r3,[r11,#-0x20]

                  start:
                       j = func5( (int) j, (char) 2, (long) 3 );
NSR:0220:00009848 E51B0020 start: ldr     r0,[r11,#-0x20]
NSR:0220:0000984C E3A01002       mov      r1,#0x2            ; r1,#2
NSR:0220:00009850 E3A02003       mov      r2,#0x3            ; r2,#3
NSR:0220:00009854 EBFFFBC5       bl       0x8770             ; func5
NSR:0220:00009858 E1A03000       cpy      r3,r0
NSR:0220:0000985C E50B3020       str      r3,[r11,#-0x20]

                       if ( j == 0 )
NSR:0220:00009860 E51B3020       ldr      r3,[r11,#-0x20]
NSR:0220:00009864 E3530000       cmp      r3,#0x0            ; r3,#0
```

```
do_page_fault(
  addr = 0x0,
  fsr = 0x80000007,
  regs = 0xDC3D5FB0 → (
    uregs = (
      [0x0] = 0x0,
      [0x1] = 0x303A,
      [0x2] = 0x000121F0,
      [0x3] = 0x0,
      [0x4] = 0x00012200,
      [0x5] = 0xBEA7CD38,
      [0x6] = 0x0,
      [0x7] = 0x0,
      [0x8] = 0x0,
      [0x9] = 0x0,
      [0x0A] = 0xB6F23000,
      [0x0B] = 0xBEA7CD74,
      [0x0C] = 0xBEA7CD48,
      [0x0D] = 0xBEA7CD18,
      [0x0E] = 0x9840,
      [0x0F] = 0x0,
      [0x10] = 0x20000110,
      [0x11] = 0xFFFFFFFF))
  tsk = 0xDCA72C00
  mm = 0xDC3F83C0
  fault = 0x00030001
  sig = 0x0B
```

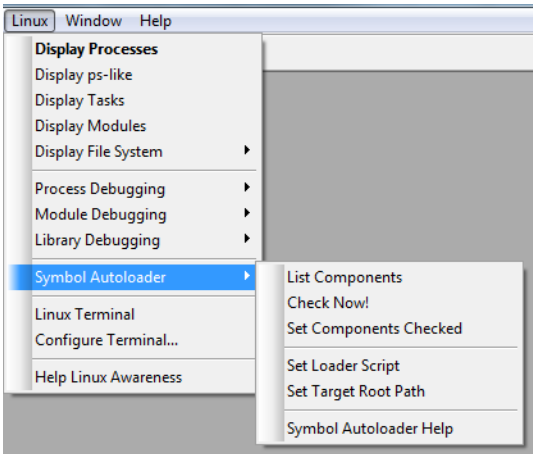
c) Kernel Modules

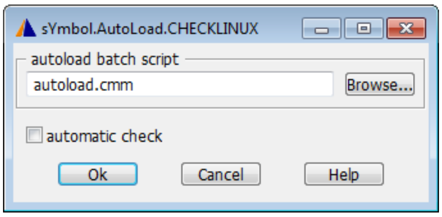
Kernel modules are loaded and linked into the kernel at run-time. To ease the debugging of kernel modules the enhanced Linux menu offers item “Debug Module on init...“ which waits until a module is loaded, and loads the symbols if the init function belongs to the wanted module name

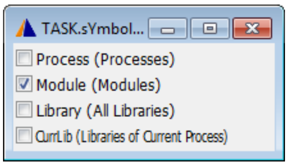




This **“Module Debugging” menu depends on the “autoloader“** function of the TRACE32 Linux Awareness based on the script “\demo\arm\kernel\linux\autoload.cmm”..



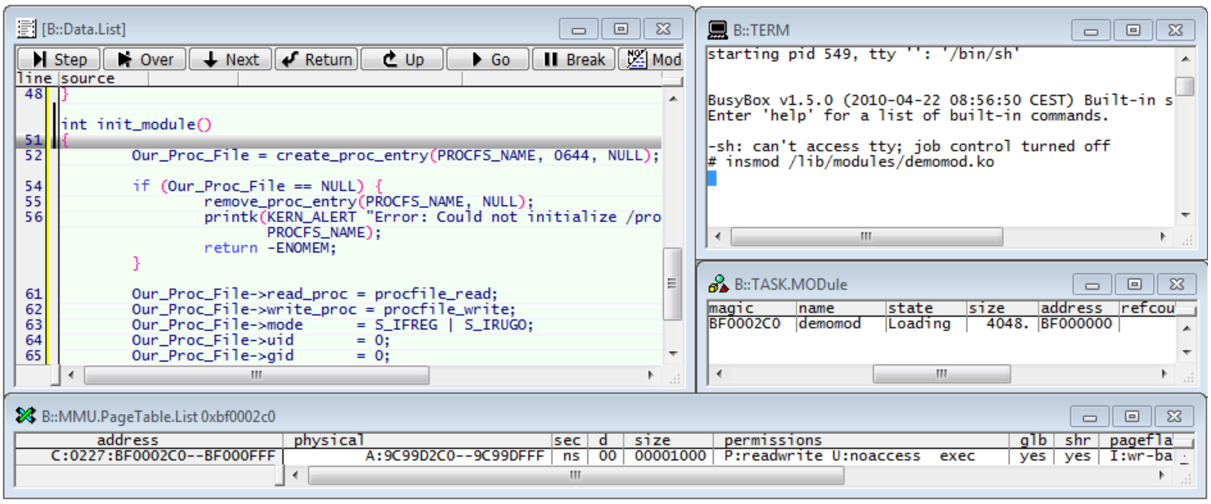




Remember, that **the kernel modules are part of the kernel address range and should be covered by TRANSlation.COMMON**.

Please note, that the section addresses (and with them the possibility to debug kernel modules) for Linux kernel version 2.4 are only available under certain circumstances! See the Linux Awareness Manual, how to set up these prerequisites, and how to debug the initialization routine of a kernel module.

After loading for example the kernel module demomod by calling “insmod demomod.ko” entered into a terminal window you will see the code and source of “demomod” at the address of the modules init function.



```
line source
}

int init_module()
{
        Our_Proc_File = create_proc_entry(PROCFS_NAME, 0644, NULL);

        if (Our_Proc_File == NULL) {
                remove_proc_entry(PROCFS_NAME, NULL);
                printk(KERN_ALERT "Error: Could not initialize /pro
                        PROCFS_NAME);
                return -ENOMEM;
        }

        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->write_proc = procfile_write;
        Our_Proc_File->mode       = S_IFREG | S_IRUGO;
        Our_Proc_File->uid        = 0;
        Our_Proc_File->gid        = 0;
```

```
starting pid 549, tty '': '/bin/sh'


BusyBox v1.5.0 (2010-04-22 08:56:50 CEST) Built-in s
Enter 'help' for a list of built-in commands.

-sh: can't access tty; job control turned off
# insmod /lib/modules/demomod.ko
```

magic	name	state	size	address	refcou
BF0002C0	demomod	Loading	4048.	BF000000	

address	physical	sec	d	size	permissions	glb	shr	pagefla
C:0227:BF0002C0--BF000FFF	A:9C99D2C0--9C99DFFF	ns	00	00001000	P:readwrite U:noaccess exec	yes	yes	I:wr-ba

If you remove a kernel module from the kernel (with “rmmod”), you **should** remove the symbols of the module from the debugger.

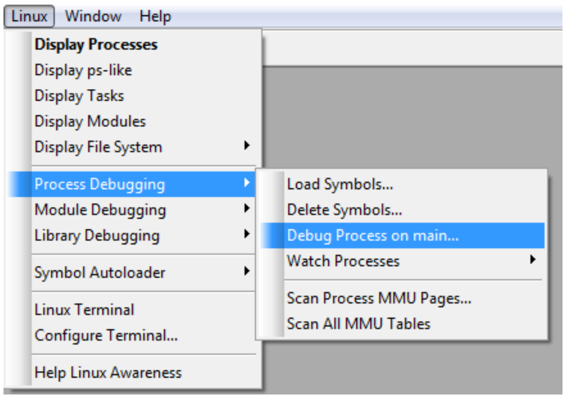
```
TASK.sYmbol.DELeteMod “demomod”        ; erase obsolete module symbols
```

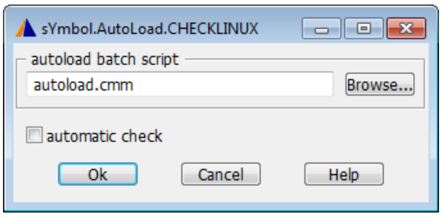
The demo directory contains a script file called “mod_debug.cmm” which has the same functionality as the menu item “Debug Module on init...“:

```
DO ~~\demo\arm\kernel\linux\mod_debug.cmm demomod
```

Debugging a Process not started yet

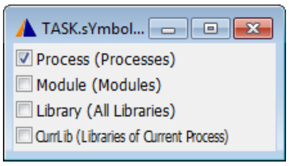
You can configure the debugger to debug a process from its start. **The Linux menu** provides a comfortable way to debug processes from main. The process will be stopped right after the instruction at main is executed. (Due to the page fault handling the needed page could not available until the CPU wants to execute the instruction at main.)





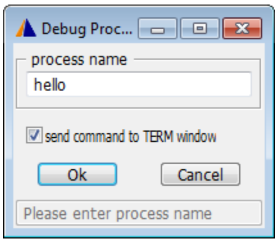
autoload batch script: autoload.cmm

automatic check



- [✓] Process (Processes)
- [ ] Module (Modules)
- [ ] Library (All Libraries)
- [ ] CurrLib (Libraries of Current Process)

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



process name: hello

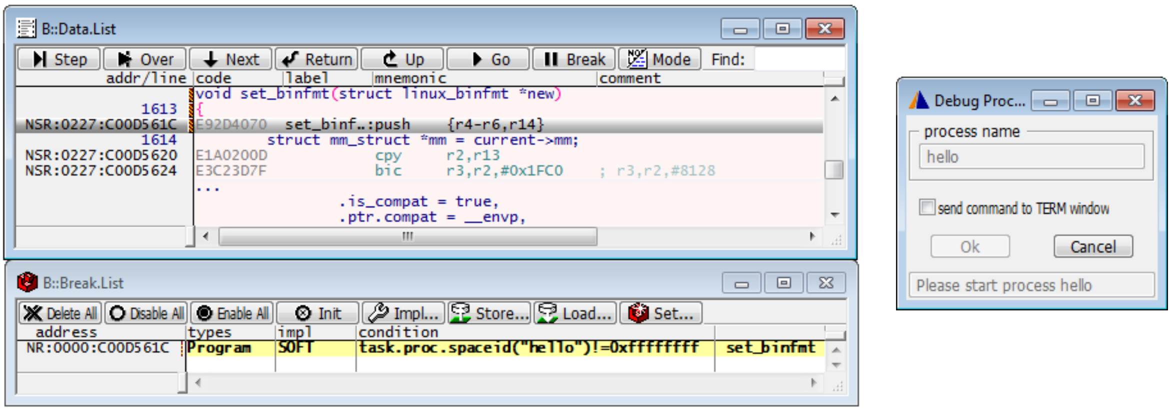
[✓] send command to TERM window

Please enter process name

You can also use the script app_debug.cmm available under demo\<arch>\kernel\linux to debug processes on main e.g.

```
DO ~~\demo\arm\kernel\linux\app_debug.cmm hello
```

Now after this preparation you can wait till the application reaches main of the given process. Internally a conditional software breakpoint is set on kernel function “set_binfmt”. When this breakpoint is hit, TRACE32 checks if the specified process is already loaded. If so the debugger extracts the space id and loads the symbols. At this stage, the code of the process has not yet been loaded so we can’t set software breakpoints on the process’ code. We use instead here an onchip breakpoint to the first instruction after main() (e.g. main+4).



```
addr/line          code      label          mnemonic              comment
                   void set_binfmt(struct linux_binfmt *new)
1613               {
NSR:0227:C00D561C  E92D4070  set_binf..:push    {r4-r6,r14}
1614                  struct mm_struct *mm = current->mm;
NSR:0227:C00D5620  E1A0200D             cpy     r2,r13
NSR:0227:C00D5624  E3C23D7F             bic     r3,r2,#0x1FC0        ; r3,r2,#8128
                   ...
                                .is_compat = true,
                                .ptr.compat = __envp,
```

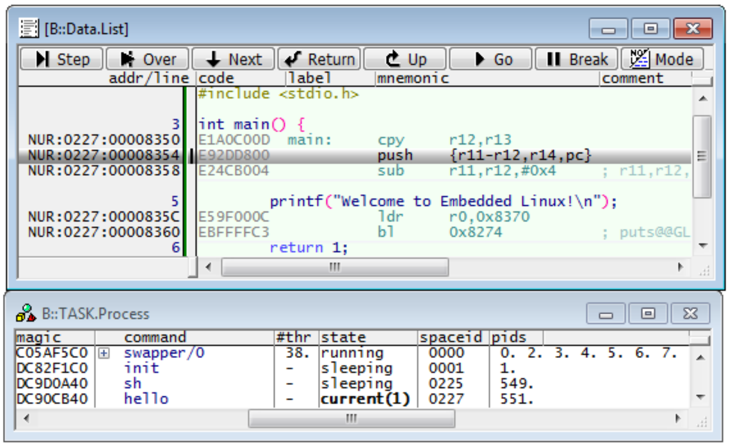
```
address            types    impl  condition
NR:0000:C00D561C   Program  SOFT  task.proc.spaceid("hello")!=0xffffffff   set_binfmt
```

process name: hello

☐ send command to TERM window

Please start process hello

As soon as the process is started, the code will be loaded and executed and the breakpoint will be hit. Now, we are able to scan the process' MMU and to set software breakpoints.



```
addr/line          code      label   mnemonic              comment
                   #include <stdio.h>

3                  int main() {
NUR:0227:00008350  E1A0C00D  main:   cpy    r12,r13
NUR:0227:00008354  E92DD800          push   {r11-r12,r14,pc}
NUR:0227:00008358  E24CB004          sub    r11,r12,#0x4     ; r11,r12,

5                          printf("Welcome to Embedded Linux!\n");
NUR:0227:0000835C  E59F000C          ldr    r0,0x8370
NUR:0227:00008360  EBFFFFC3          bl     0x8274           ; puts@@GL
6                          return 1;
```

```
magic     command      #thr  state       spaceid  pids
C05AF5C0  swapper/0    38.   running     0000     0. 2. 3. 4. 5. 6. 7.
DC82F1C0  init         -     sleeping    0001     1.
DC9D0A40  sh           -     sleeping    0225     549.
DC90CB40  hello        -     current(1)  0227     551.
```

The supporting dialog of "Debug Process on main" is closed (if used: app_debug.cmm is finished) and all breakpoints used for this aim are erased.

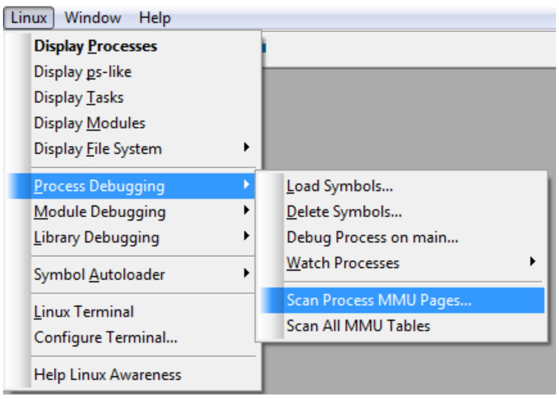
For your convenience, you may define groups of processes and they'll be marked by colored bars:

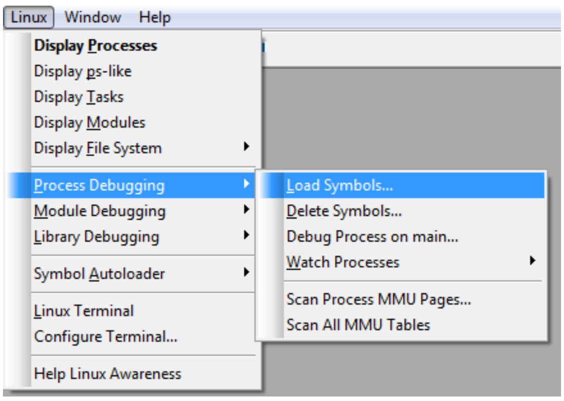
```
GROUP.Create "myprocgrp" 0x2C0:0x0--0x7fffffff /GREEN
```

You can continue now debugging your process.

Debugging a Process started already

If the process is already running it is **not possible to work with** the comfortable menu item **“Debug Process on main”**! Because the feature works with 2 breakpoint on addresses that will not be used again: “set_binfmt” from kernel and “main+0x4 at early start of your process. Actually, the process MMU tables already exist and the code of the process is accessible. We just need to load the symbols.

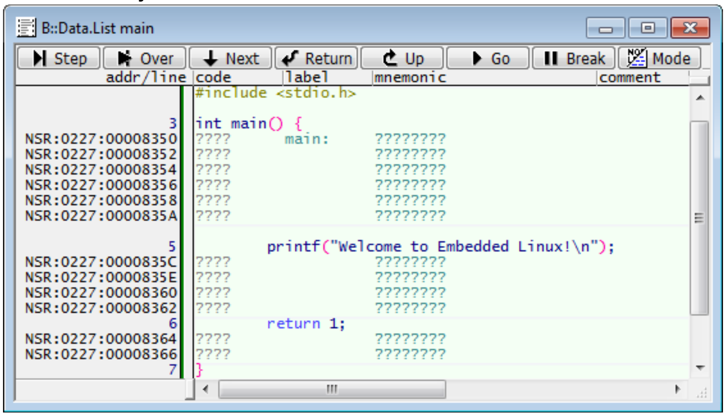




If you activated the **TRANSlation.TableWalk** and initialized the symbol autoloader you can just stop anywhere and the debugger will be automatically updated to the new target state.

Symbol cleanup after process termination

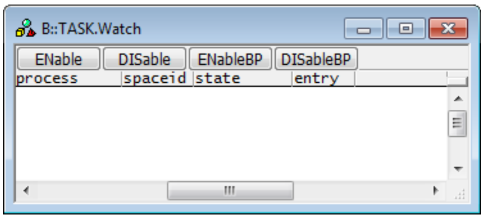
After the process “hello” exits its symbols are no more valid:

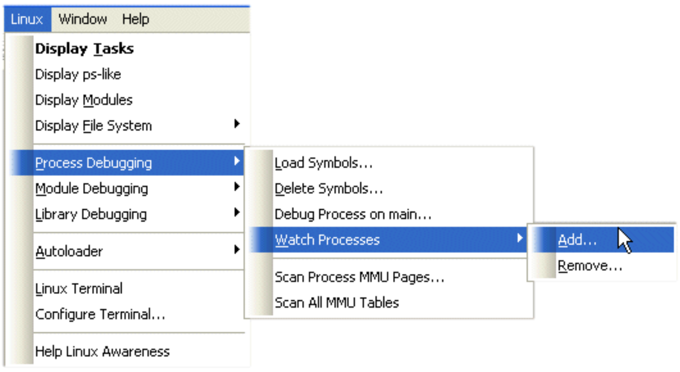


```
        addr/line |code       |label     |mnemonic                  |comment
                  |#include <stdio.h>
                 3|int main() {
NSR:0227:00008350 |????        main:      ????????
NSR:0227:00008352 |????                   ????????
NSR:0227:00008354 |????                   ????????
NSR:0227:00008356 |????                   ????????
NSR:0227:00008358 |????                   ????????
NSR:0227:0000835A |????                   ????????
                 5|        printf("Welcome to Embedded Linux!\n");
NSR:0227:0000835C |????                   ????????
NSR:0227:0000835E |????                   ????????
NSR:0227:00008360 |????                   ????????
NSR:0227:00008362 |????                   ????????
                 6|        return 1;
NSR:0227:00008364 |????                   ????????
NSR:0227:00008366 |????                   ????????
                 7|}
```

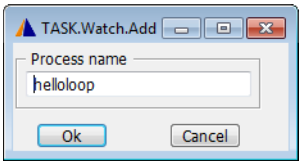
```
sYmbol.Delete \\hello                        ; get rid of invalid symbols
```

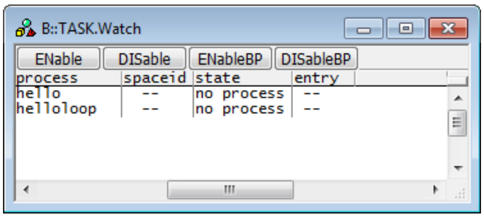
There is a “process watch system” available, that watches for the creation and termination of specified processes. This watch system then loads and deletes symbols automatically and keeps the MMU in sync. See “`TASK.Watch`” in the Linux Awareness Manual for details.





Use the watch system and the menu to add and remove processes to be observed and synchronized to.



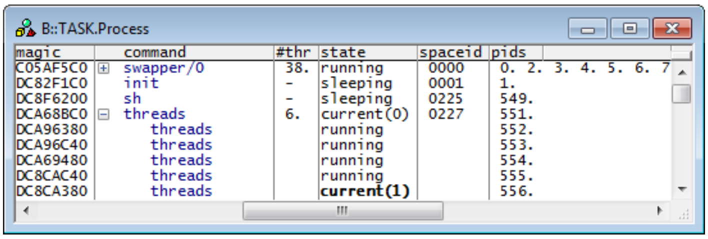


```
process   spaceid state       entry
hello     --      no process  --
helloloop --      no process  --
```

Now that you know how to debug Linux processes the biggest part is done. But there are many Linux related TRACE32 windows offering Linux specific information you might be looking for. So **further exiting features of the TRACE32 Linux Awareness are shown in chapter “Linux specific Windows and Features”.**

e) 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. See chapter “Processes” how to load and handle these symbols.

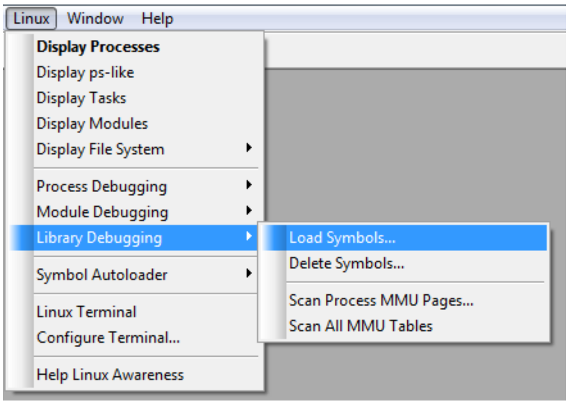


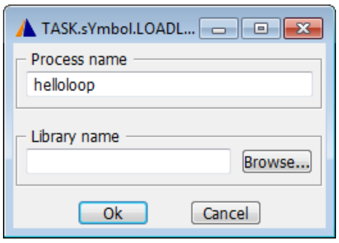
```
magic      command       #thr state       spaceid pids
C05AF5C0 ⊞ swapper/0     38.  running     0000    0. 2. 3. 4. 5. 6. 7
DC82F1C0   init          -    sleeping    0001    1.
DC8F6200   sh            -    sleeping    0225    549.
DCA68BC0 ⊟ threads       6.   current(0)  0227    551.
DCA96380     threads          running             552.
DCA96C40     threads          running             553.
DCA69480     threads          running             554.
DC8CAC40     threads          running             555.
DC8CA380     threads          current(1)          556.
```

f) 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 you can use the menu “Library Debugging”:

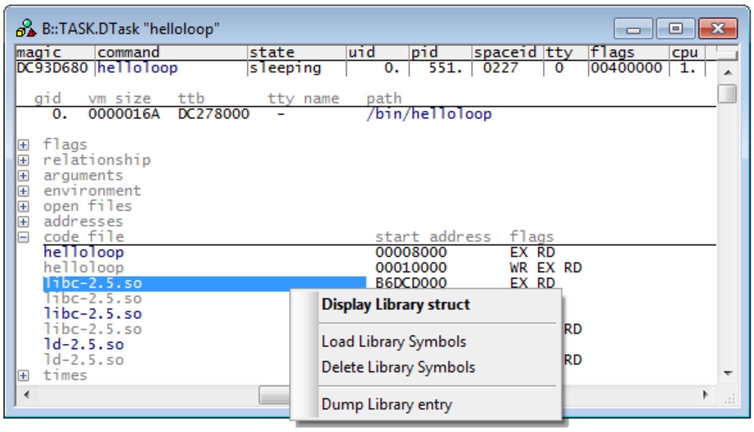
Or you display first the task list by command “Task.DTask” and then continue with double or right-clicks:





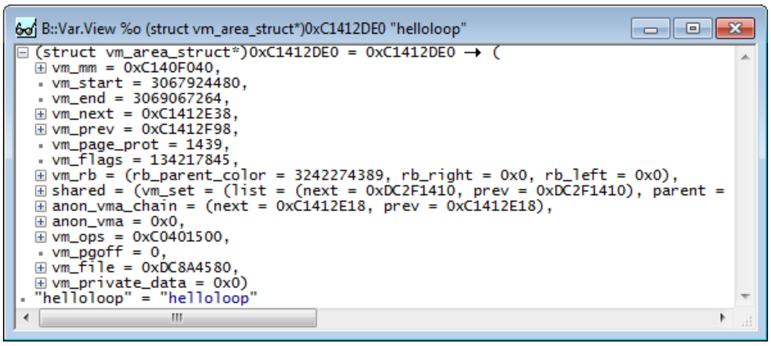
Process name: helloloop

Library name:



```
magic    |command        |state     |uid  |pid   |spaceid|tty |flags    |cpu
DC93D680 |helloloop      |sleeping  |   0.|  551.| 0227  | 0  |00400000 | 1.

  gid   vm size   ttb        tty name   path
    0.  0000016A  DC278000    -         /bin/helloloop

⊞ flags
⊞ relationship
⊞ arguments
⊞ environment
⊞ open files
⊞ addresses
⊟ code file                                    start address  flags
   helloloop                                   00008000       EX RD
   helloloop                                   00010000       WR EX RD
   libc-2.5.so                                 B6DCD000       EX RD
   libc-2.5.so
   libc-2.5.so
   libc-2.5.so                                                         RD
   ld-2.5.so
   ld-2.5.so                                                           RD
⊞ times
```

You can also show the internal kernel structure for a selected library by selecting “display library struct”.


B::Var.View %o (struct vm_area_struct*)0xC1412DE0 "helloloop"

```
⊟ (struct vm_area_struct*)0xC1412DE0 = 0xC1412DE0 → (
  ⊞ vm_mm = 0xC140F040,
  ▪ vm_start = 3067924480,
  ▪ vm_end = 3069067264,
  ⊞ vm_next = 0xC1412E38,
  ⊞ vm_prev = 0xC1412F98,
  ▪ vm_page_prot = 1439,
  ▪ vm_flags = 134217845,
  ⊞ vm_rb = (rb_parent_color = 3242274389, rb_right = 0x0, rb_left = 0x0),
  ⊞ shared = (vm_set = (list = (next = 0xDC2F1410, prev = 0xDC2F1410), parent =
  ⊞ anon_vma_chain = (next = 0xC1412E18, prev = 0xC1412E18),
  ⊞ anon_vma = 0x0,
  ⊞ vm_ops = 0xC0401500,
  ▪ vm_pgoff = 0,
  ⊞ vm_file = 0xDC8A4580,
  ⊞ vm_private_data = 0x0)
▪ "helloloop" = "helloloop"
```

In case you want to load or delete the symbols of the library and wish to update the MMU automatically by a script you have to use the following commands:

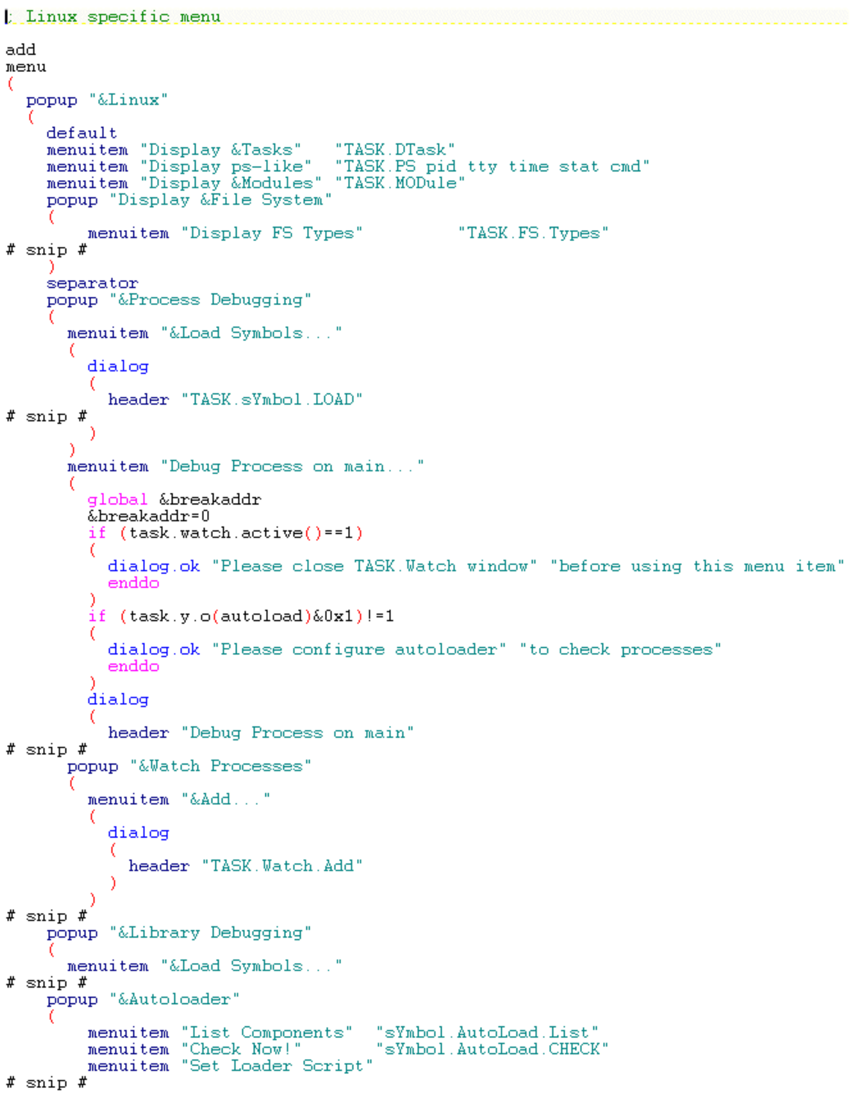
```
TASK.sYmbol.LOADLib “helloloop” “ld-2.2.5.so”   ; load library symbols

TASK.sYmbol.DELeteLib ”\\ld-2.2.5.so”           ; erase library symbols
```

Please remember to configure the “autoloader” for libraries!

2.) The Linux menu file

This topic is dedicated to persons interested in how the menus of the Linux Awareness work. Related to the previous explanations and examples you will recognize several sections and actions of the menu-building file "\demo\arm\kernel\linux\linux.men" now.

By this knowledge you should be able to use command sequences of "linux.men" in you own scripts.
But remember the features "Debug Process on main" and "Debug Module on init..." are available already as scripts: "app_debug.cmm" and "mod_debug.cmm" at "\demo\arm\kernel\linux\".



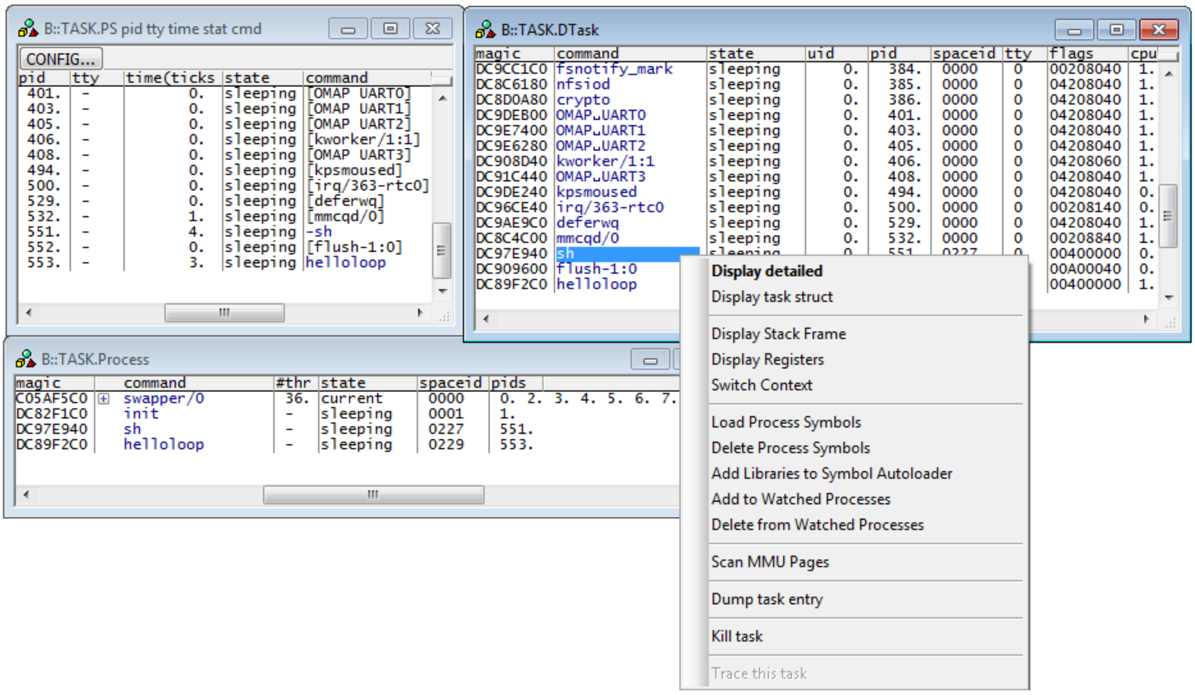
```
; Linux specific menu

add
menu
(
  popup "&Linux"
  (
    default
    menuitem "Display &Tasks"   "TASK.DTask"
    menuitem "Display ps-like"  "TASK.PS pid tty time stat cmd"
    menuitem "Display &Modules" "TASK.MODule"
    popup "Display &File System"
    (
        menuitem "Display FS Types"         "TASK.FS.Types"
# snip #
    )
    separator
    popup "&Process Debugging"
    (
      menuitem "&Load Symbols..."
      (
        dialog
        (
          header "TASK.sYmbol.LOAD"
# snip #
        )
      )
      menuitem "Debug Process on main..."
      (
        global &breakaddr
        &breakaddr=0
        if (task.watch.active()==1)
        (
          dialog.ok "Please close TASK.Watch window" "before using this menu item"
          enddo
        )
        if (task.y.o(autoload)&0x1)!=1
        (
          dialog.ok "Please configure autoloader" "to check processes"
          enddo
        )
        dialog
        (
          header "Debug Process on main"
# snip #
      popup "&Watch Processes"
      (
        menuitem "&Add..."
        (
          dialog
          (
            header "TASK.Watch.Add"
          )
        )
# snip #
    popup "&Library Debugging"
    (
      menuitem "&Load Symbols..."
# snip #
    popup "&Autoloader"
    (
        menuitem "List Components"   "sYmbol.AutoLoad.List"
        menuitem "Check Now!"        "sYmbol.AutoLoad.CHECK"
        menuitem "Set Loader Script"
# snip #
```

3.) Linux specific Windows and Features

Display of system resources

You can display the list of running tasks, kernel modules...

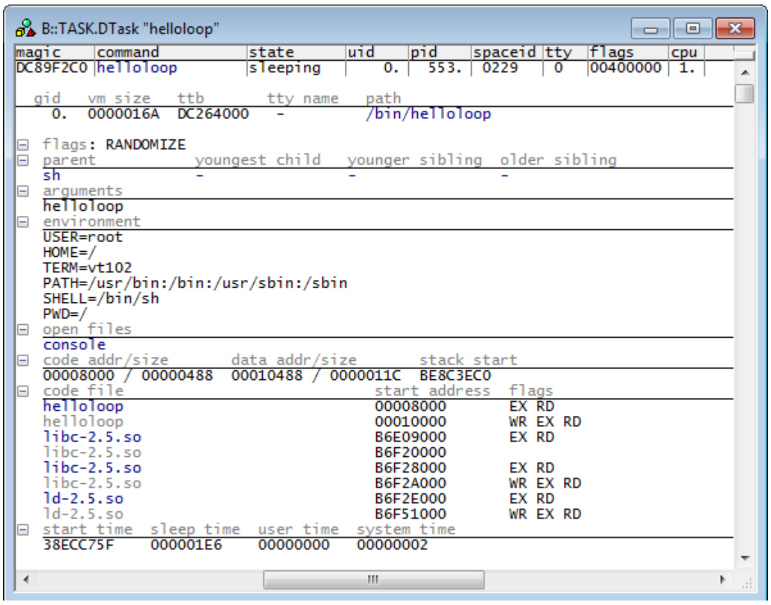
Several windows are available to display e.g. the process list. TASK.PS displays the process table similar to the output of the “ps” shell command. TASK.DTask give you more detailed information. TASK.Process displays the processes with their threads. You can open all there window from the Linux menu or from the command line.



```
B::TASK.PS pid tty time stat cmd
CONFIG...
pid   tty  time(ticks state    command
 401.  -           0. sleeping [OMAP UART0]
 403.  -           0. sleeping [OMAP UART1]
 405.  -           0. sleeping [OMAP UART2]
 406.  -           0. sleeping [kworker/1:1]
 408.  -           0. sleeping [OMAP UART3]
 494.  -           0. sleeping [kpsmoused]
 500.  -           0. sleeping [irq/363-rtc0]
 529.  -           0. sleeping [deferwq]
 532.  -           1. sleeping [mmcqd/0]
 551.  -           4. sleeping -sh
 552.  -           0. sleeping [flush-1:0]
 553.  -           3. sleeping helloloop
```

```
B::TASK.DTask
magic    command        state    uid   pid   spaceid tty flags    cpu
DC9CC1C0 fsnotify_mark  sleeping   0.  384.  0000    0   00208040 1.
DC8C6180 nfsiod         sleeping   0.  385.  0000    0   04208040 1.
DC8D0A80 crypto         sleeping   0.  386.  0000    0   04208040 1.
DC9DEB00 OMAP␣UART0     sleeping   0.  401.  0000    0   04208040 1.
DC9E7400 OMAP␣UART1     sleeping   0.  403.  0000    0   04208040 1.
DC9E6280 OMAP␣UART2     sleeping   0.  405.  0000    0   04208040 1.
DC908D40 kworker/1:1    sleeping   0.  406.  0000    0   04208060 1.
DC91C440 OMAP␣UART3     sleeping   0.  408.  0000    0   04208040 1.
DC9DE240 kpsmoused      sleeping   0.  494.  0000    0   04208040 0.
DC96CE40 irq/363-rtc0   sleeping   0.  500.  0000    0   00208140 0.
DC9AE9C0 deferwq        sleeping   0.  529.  0000    0   04208040 1.
DC8C4C00 mmcqd/0        sleeping   0.  532.  0000    0   00208840 1.
DC97E940 sh             sleeping   0.  551.  0227    0   00400000 0.
DC909600 flush-1:0                                       00A00040 0.
DC89F2C0 helloloop                                       00400000 1.
```

```
B::TASK.Process
magic      command     #thr state    spaceid pids
C05AF5C0 ⊞ swapper/0   36.  current  0000    0. 2. 3. 4. 5. 6. 7.
DC82F1C0   init        -    sleeping 0001    1.
DC97E940   sh          -    sleeping 0227    551.
DC89F2C0   helloloop   -    sleeping 0229    553.
```



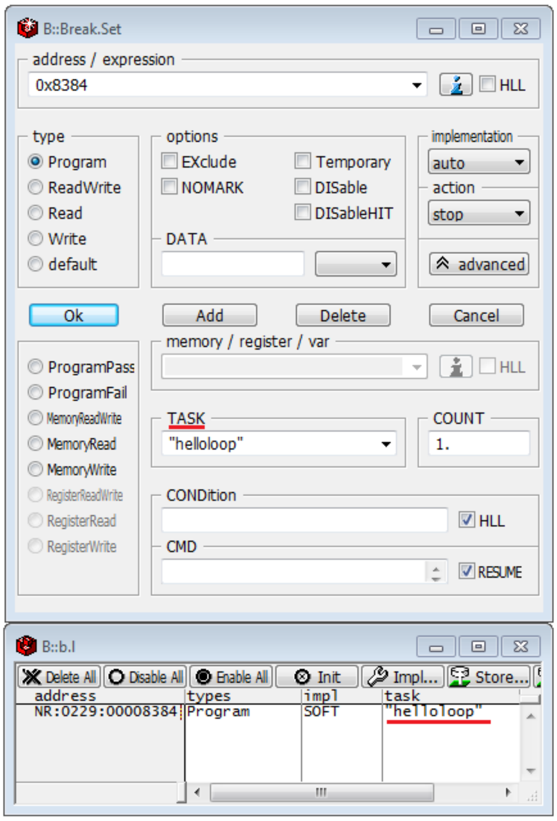
```
B::TASK.DTask "helloloop"
magic    command    state    uid   pid   spaceid tty flags    cpu
DC89F2C0 helloloop  sleeping   0.  553.  0229    0   00400000 1.

  gid   vm size   ttb       tty name   path
   0.   0000016A  DC264000  -          /bin/helloloop

⊟ flags: RANDOMIZE
⊟ parent            youngest child   younger sibling   older sibling
  sh                -                -                 -
⊟ arguments
  helloloop
⊟ environment
  USER=root
  HOME=/
  TERM=vt102
  PATH=/usr/bin:/bin:/usr/sbin:/sbin
  SHELL=/bin/sh
  PWD=/
⊟ open files
  console
⊟ code addr/size        data addr/size        stack start
  00008000 / 00000488   00010488 / 0000011C   BE8C3EC0
⊟ code file                          start address   flags
  helloloop                          00008000        EX RD
  helloloop                          00010000        WR EX RD
  libc-2.5.so                        B6E09000        EX RD
  libc-2.5.so                        B6F20000
  libc-2.5.so                        B6F28000        EX RD
  libc-2.5.so                        B6F2A000        WR EX RD
  ld-2.5.so                          B6F2E000        EX RD
  ld-2.5.so                          B6F51000        WR EX RD
⊟ start time   sleep time   user time   system time
  38ECC75F     000001E6     00000000    00000002
```

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 the specified one.



address / expression: 0x8384

type: [✓ Program]

implementation: auto

action: stop

TASK: "helloloop"

COUNT: 1.

CONDition: [✓ HLL]

CMD: [✓ RESUME]

```
address           types     impl   task
NR:0229:00008384  Program   SOFT   "helloloop"
```

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:

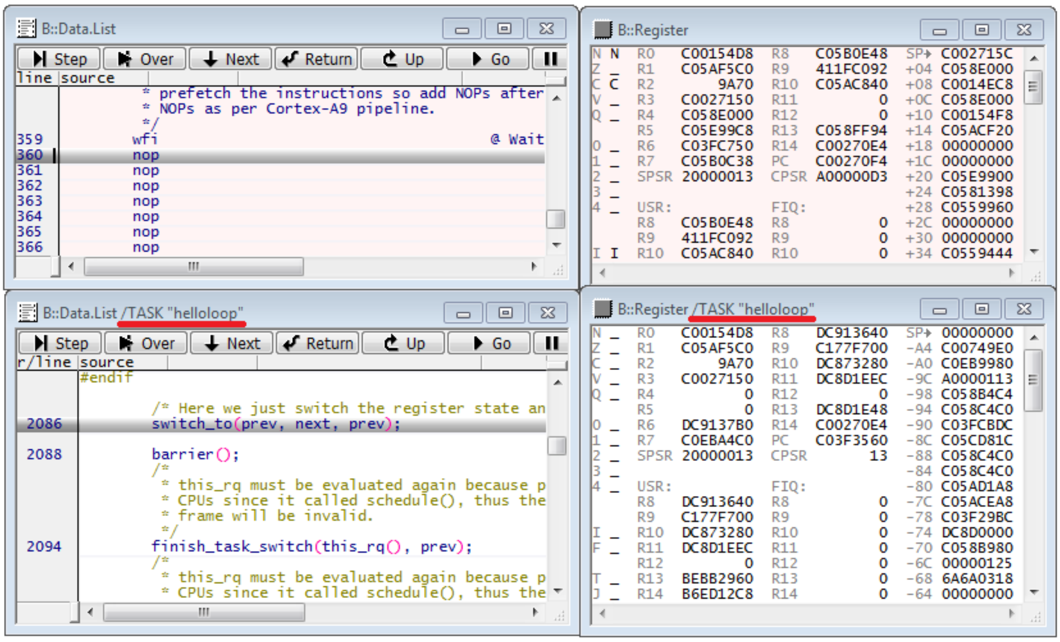
```
SETUP.StepWithinTask ON
```

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.

```
Data.List /TASK “mytask”
Register /TASK “mytask”
Var.Frame /TASK “mytask”
```

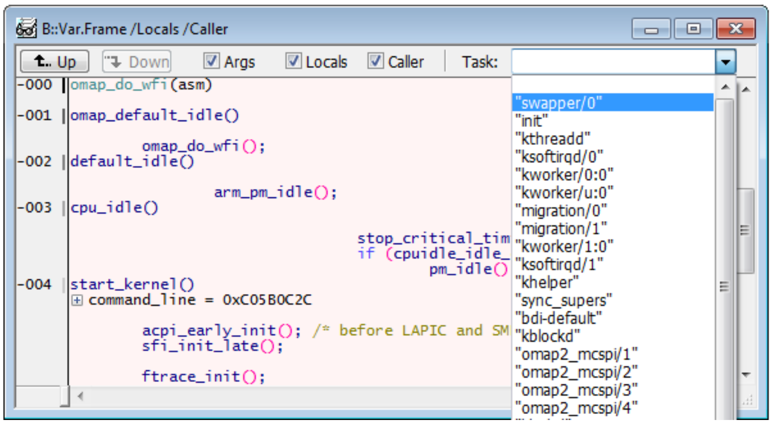


```
line source
                * prefetch the instructions so add NOPs after
                * NOPs as per Cortex-A9 pipeline.
                */
359            wfi                                       @ Wait
360            nop
361            nop
362            nop
363            nop
364            nop
365            nop
366            nop
```

```
N N  R0    C00154D8  R8    C05B0E48  SP> C002715C
Z _  R1    C05AF5C0  R9    411FC092  +04 C058E000
C C  R2        9A70  R10   C05AC840  +08 C0014EC8
V _  R3    C0027150  R11          0  +0C C058E000
Q _  R4    C058E000  R12          0  +10 C00154F8
     R5    C05E99C8  R13   C058FF94  +14 C05ACF20
0 _  R6    C03FC750  R14   C00270E4  +18 00000000
1 _  R7    C05B0C38  PC    C00270F4  +1C 00000000
2 _  SPSR  20000013  CPSR  A00000D3  +20 C05E9900
3 _                                  +24 C0581398
4 _  USR:            FIQ:            +28 C0559960
     R8    C05B0E48  R8           0  +2C 00000000
     R9    411FC092  R9           0  +30 00000000
I I  R10   C05AC840  R10          0  +34 C0559444
```

```
r/line source
       #endif

                /* Here we just switch the register state an
 2086           switch_to(prev, next, prev);

 2088           barrier();
                /*
                 * this_rq must be evaluated again because p
                 * CPUs since it called schedule(), thus the
                 * frame will be invalid.
                 */
 2094           finish_task_switch(this_rq(), prev);
                /*
                 * this_rq must be evaluated again because p
                 * CPUs since it called schedule(), thus the
```

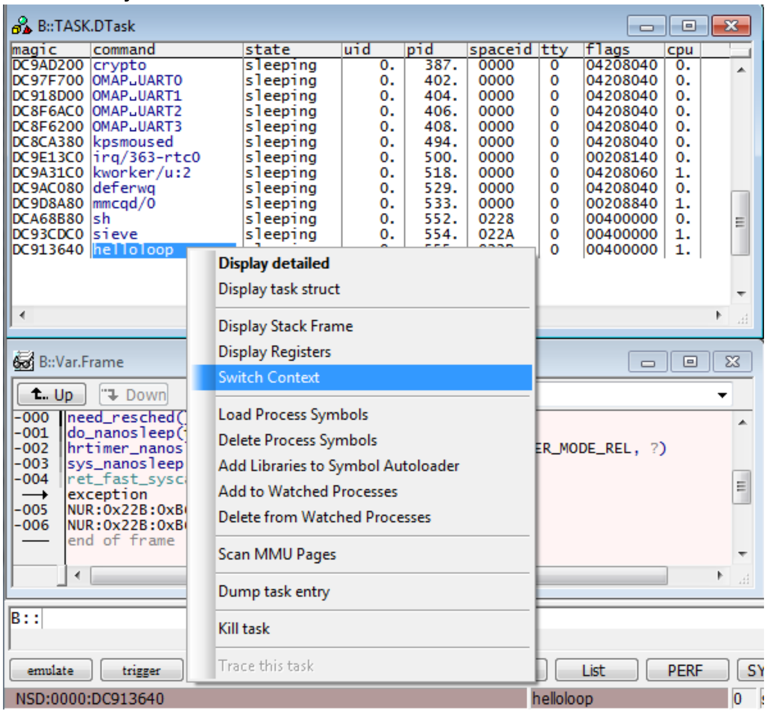
```
N _  R0    C00154D8  R8    DC913640  SP> 00000000
Z _  R1    C05AF5C0  R9    C177F700  -A4 C00749E0
C _  R2        9A70  R10   DC873280  -A0 C0EB9980
V _  R3    C0027150  R11   DC8D1EEC  -9C A0000113
Q _  R4           0  R12          0  -98 C058B4C4
     R5           0  R13   DC8D1E48  -94 C058C4C0
0 _  R6    DC9137B0  R14   C00270E4  -90 C03FCBDC
1 _  R7    C0EBA4C0  PC    C03F3560  -8C C05CD81C
2 _  SPSR  20000013  CPSR        13  -88 C058C4C0
3 _                                  -84 C058C4C0
4 _  USR:            FIQ:            -80 C05AD1A8
     R8    DC913640  R8           0  -7C C05ACEA8
     R9    C177F700  R9           0  -78 C03F29BC
I _  R10   DC873280  R10          0  -74 DC8D0000
F _  R11   DC8D1EEC  R11          0  -70 C058B980
     R12          0  R12          0  -6C 00000125
T _  R13   BEBB2960  R13          0  -68 6A6A0318
J _  R14   B6ED12C8  R14          0  -64 00000000
```



```
-000 |omap_do_wfi(asm)

-001 |omap_default_idle()

             omap_do_wfi();
-002 |default_idle()

                     arm_pm_idle();
-003 |cpu_idle()

                                   stop_critical_tim
                                   if (cpuidle_idle_
                                           pm_idle()
-004 |start_kernel()
     ⊞ command_line = 0xC05B0C2C

             acpi_early_init(); /* before LAPIC and SM
             sfi_init_late();

             ftrace_init();
```

```
"swapper/0"
"init"
"kthreadd"
"ksoftirqd/0"
"kworker/0:0"
"kworker/u:0"
"migration/0"
"migration/1"
"kworker/1:0"
"ksoftirqd/1"
"khelper"
"sync_supers"
"bdi-default"
"kblockd"
"omap2_mcspi/1"
"omap2_mcspi/2"
"omap2_mcspi/3"
"omap2_mcspi/4"
```

Moreover, you can switch the context also from the “DTask” window by popup menu-item “Switch Context”:



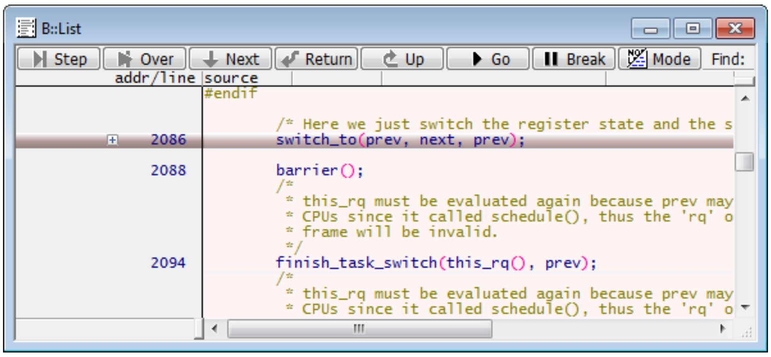
```
magic    command       state     uid  pid   spaceid tty flags    cpu
DC9AD200 crypto        sleeping   0.  387.  0000    0   04208040 0.
DC97F700 OMAP-UART0    sleeping   0.  402.  0000    0   04208040 0.
DC918D00 OMAP-UART1    sleeping   0.  404.  0000    0   04208040 0.
DC8F6AC0 OMAP-UART2    sleeping   0.  406.  0000    0   04208040 0.
DC8F6200 OMAP-UART3    sleeping   0.  408.  0000    0   04208040 0.
DC8CA380 kpsmoused     sleeping   0.  494.  0000    0   04208040 0.
DC9E13C0 irq/363-rtc0  sleeping   0.  500.  0000    0   00208140 0.
DC9A31C0 kworker/u:2   sleeping   0.  518.  0000    0   04208060 1.
DC9AC080 deferwq       sleeping   0.  529.  0000    0   04208040 0.
DC9D8A80 mmcqd/0       sleeping   0.  533.  0000    0   00208840 1.
DCA68B80 sh            sleeping   0.  552.  0228    0   00400000 0.
DC93CDC0 sieve         sleeping   0.  554.  022A    0   00400000 1.
DC913640 helloloop                                  0   00400000 1.
```

```
-000  need_resched(
-001  do_nanosleep(
-002  hrtimer_nanos
-003  sys_nanosleep
-004  ret_fast_sysc
 →    exception
-005  NUR:0x22B:0xB
-006  NUR:0x22B:0xB
 ——   end of frame
                                ER_MODE_REL, ?)
```

```
B::
NSD:0000:DC913640                    helloloop        0
```

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!



```
addr/line source
          #endif

                  /* Here we just switch the register state and the s
     2086         switch_to(prev, next, prev);

     2088         barrier();
                  /*
                   * this_rq must be evaluated again because prev may
                   * CPUs since it called schedule(), thus the 'rq' o
                   * frame will be invalid.
                   */
     2094         finish_task_switch(this_rq(), prev);
                  /*
                   * this_rq must be evaluated again because prev may
                   * CPUs since it called schedule(), thus the 'rq' o
```

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.

Epilog

Thank you for reading and working through this training manual. You should now be able to debug Linux targets. This tutorial was written with as much care as possible for accuracy and comprehensibility.

However, Linux is a very complex system with different and changing internal structures. Please forgive any unclear or even incorrect content. If you find any mistakes, or if you have any comments/ideas to improve this document, feel free to mail your proposals to the Linux support team at Lauterbach. Your feedback as user of our tools is the most valuable input for us.

The LAUTERBACH Team.