

LINUX & Android Debugging

Frank,Xing

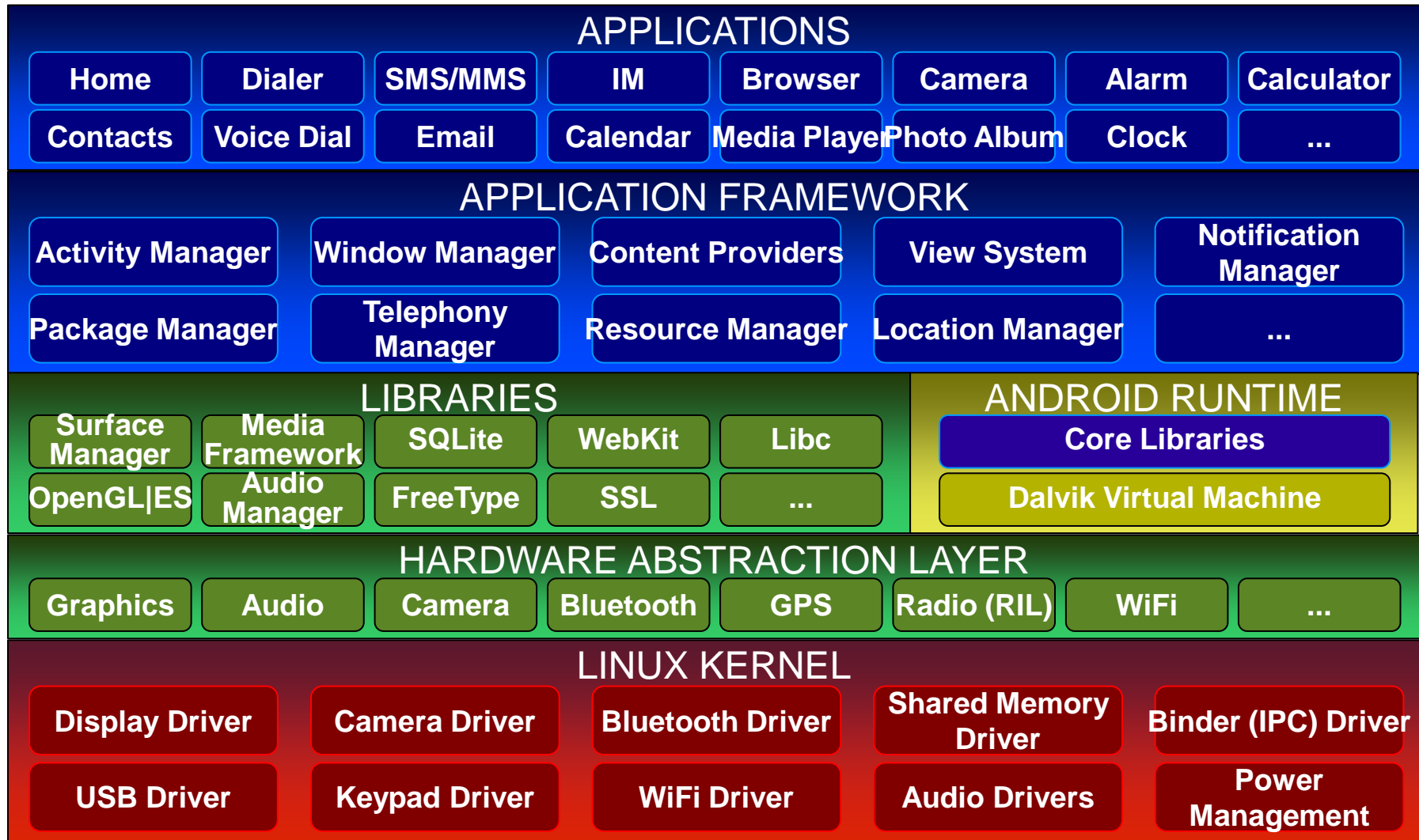
➤ Platform and Debug Overview

◆ Linux Debugging

- ◆ Configuration Linux-Aware Debugging
- ◆ Stop-Mode-Debugging
- ◆ Stop & Run-Mode-Debugging

◆ VM-Dalvik Debugging

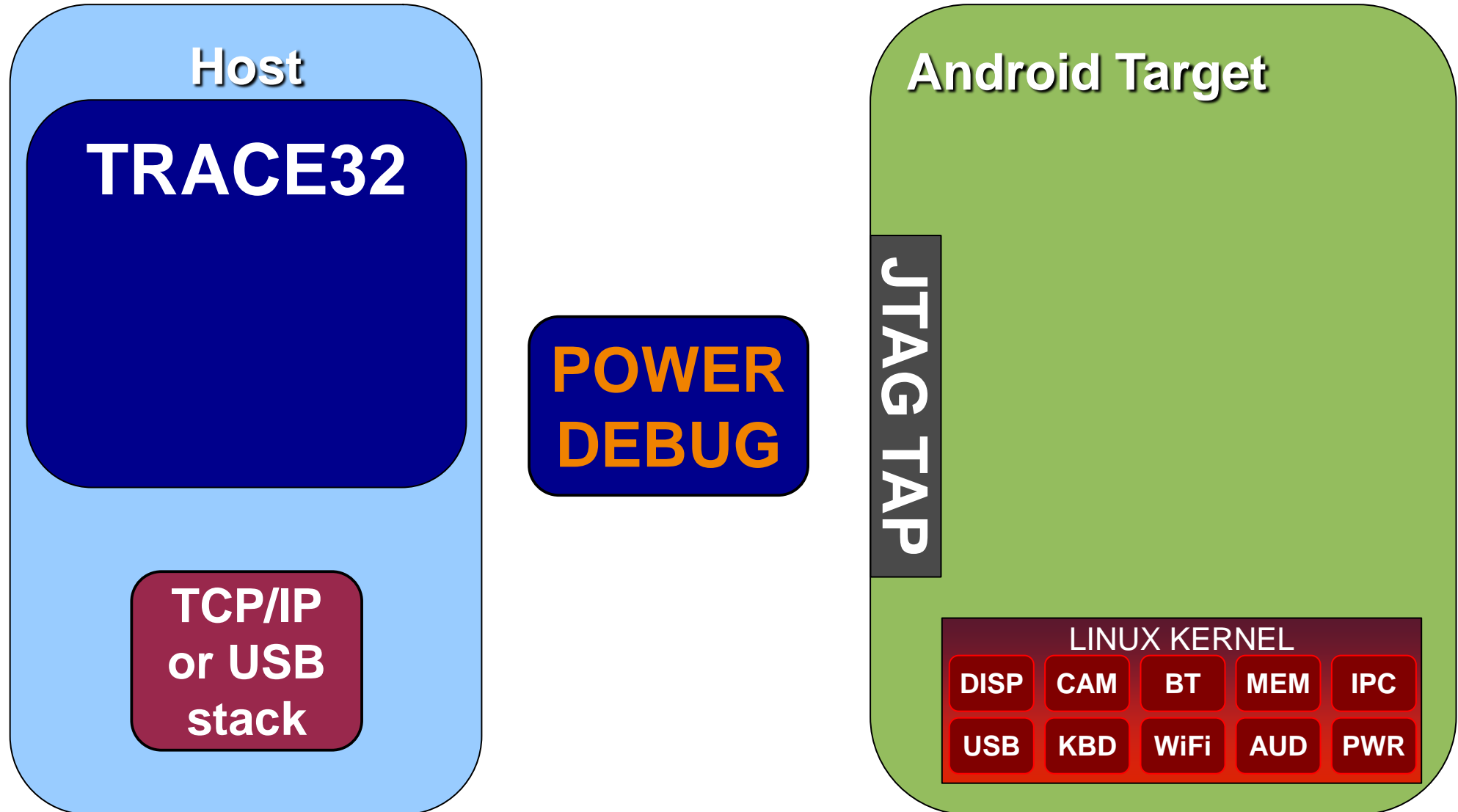
Android Architecture



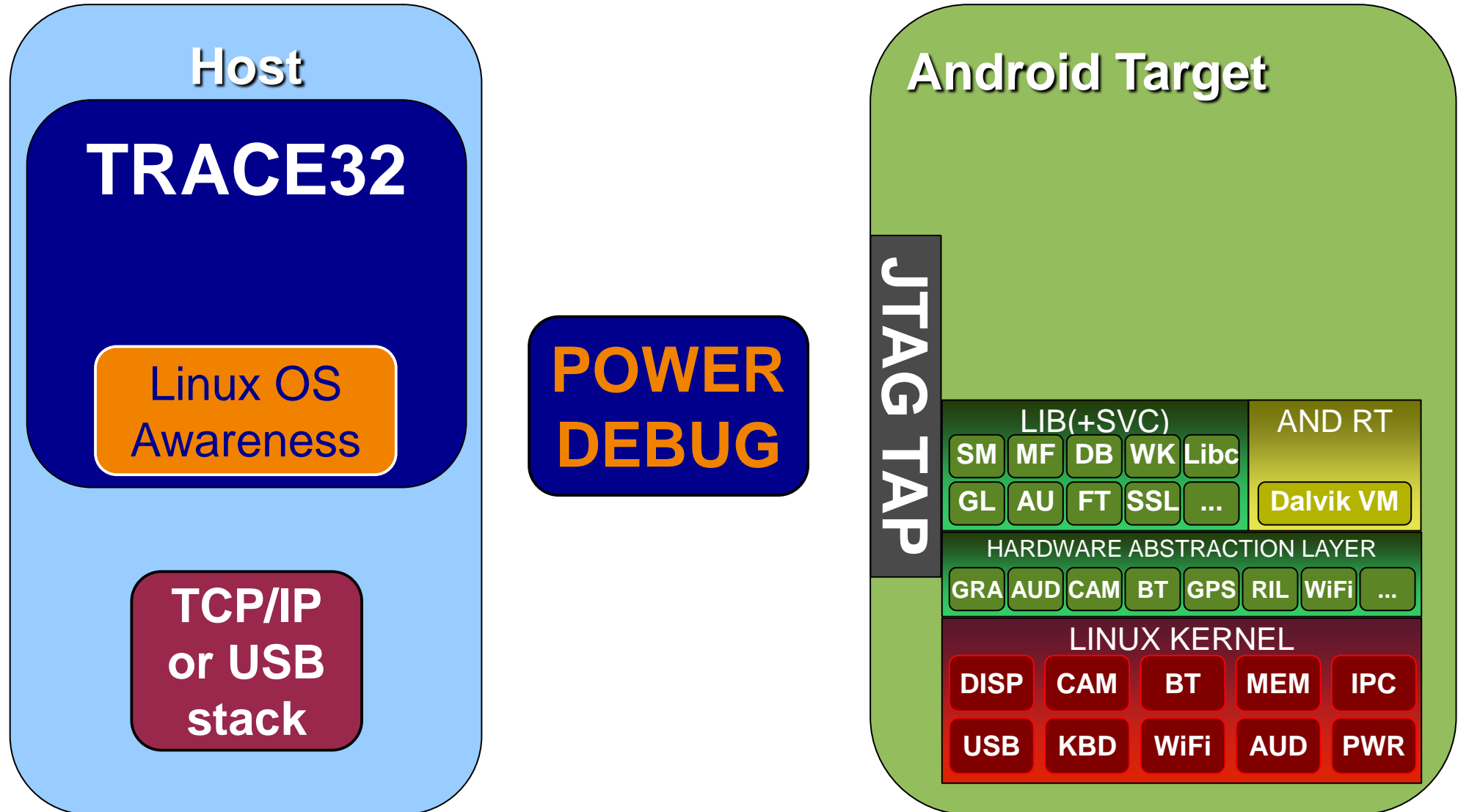
Debug Abstraction Layers

- Native Machine Code
 - needs Embedded Debugger HW+SW for specific machine (JTAG/SWD, Register Layout, MMU Types, Peripherals, ...)
 - for Assembly and HLL Level Debugging (C/C++)
- Operating System Tasks and Native Applications
 - needs Linux 'Operating System Awareness'
 - for Task-Aware Debugging of Services and Native Applications
- Virtual Machine Applications
 - needs Dalvik 'Virtual Machine Awareness'
 - for HLL Level Debugging of VM Code (Java) Applications

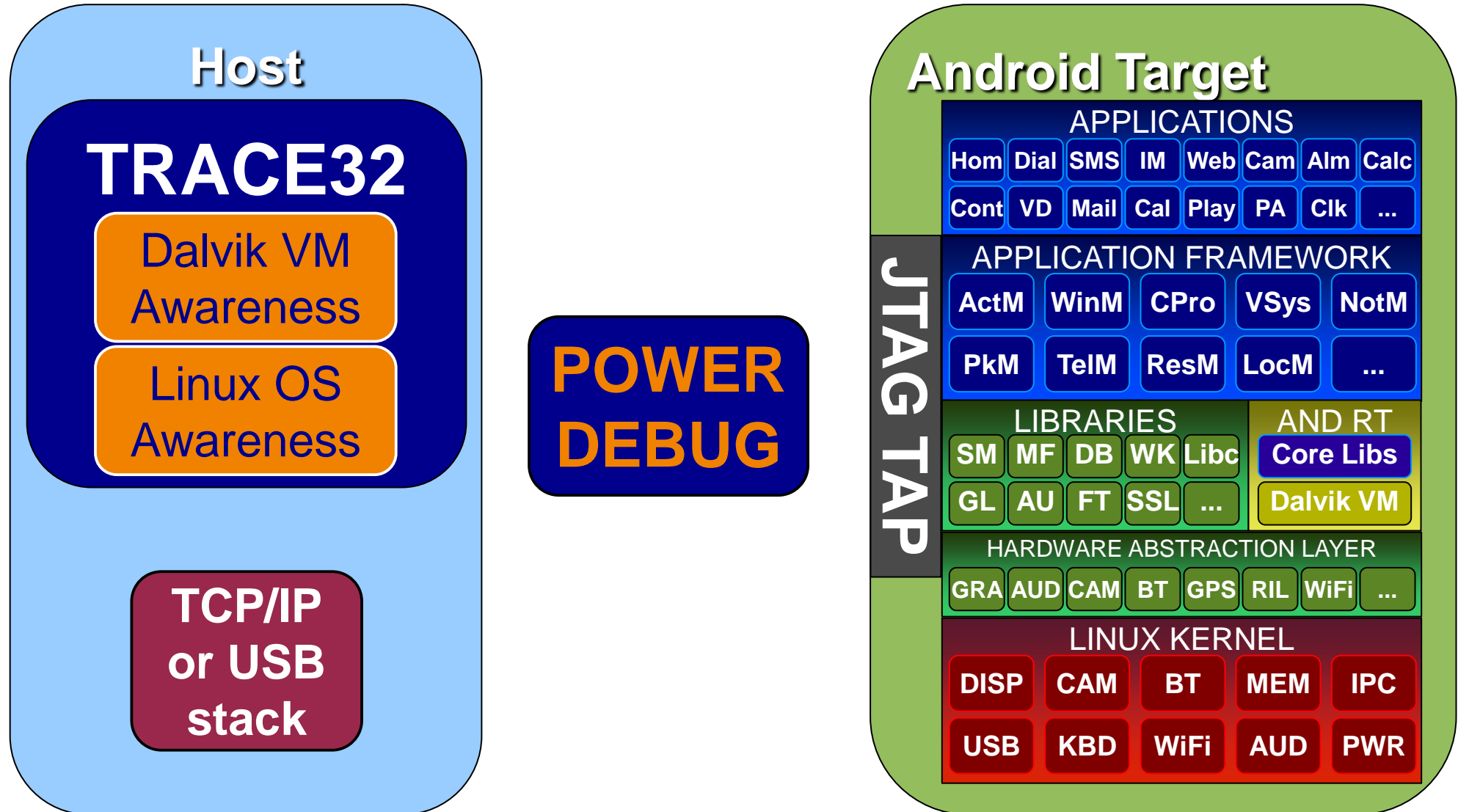
Stop-Mode Debugging – Machine Level



Stop-Mode Debugging – OS Awareness



Stop-Mode Debugging – OS + VM Awareness



LINUX & Android Debugging

Frank,Xing

- ◆ Platform and Debug Overview
 - Linux Debugging
 - Configuration Linux-Aware Debugging
 - ◆ Stop-Mode-Debugging
 - ◆ Stop & Run-Mode-Debugging
- ◆ VM-Dalvik Debugging

Customers components for Linux-Debugging

- Requirements for the customer SW
 - The **application AND the Linux kernel** need to be **compiled with** appropriate settings to have full **debug information** included.
 - For the kernel it means you need to activate “**kernel hacking**” → “**Compile kernel with debug info**” before compilation!
 - The more, you **MUST** load the target with the same version of application. This means: **DON'T** load an application/kernel compiled without debug information. It has always different code contents than the one compiled with debug information.
 - **INSTEAD** if you need a stripped version, copy "vmlinux" to "vmlinux.elf" and **USE** (e.g.) command "arm-none-linux-gnueabi-strip vmlinux" to get a kernel file without debug symbols.

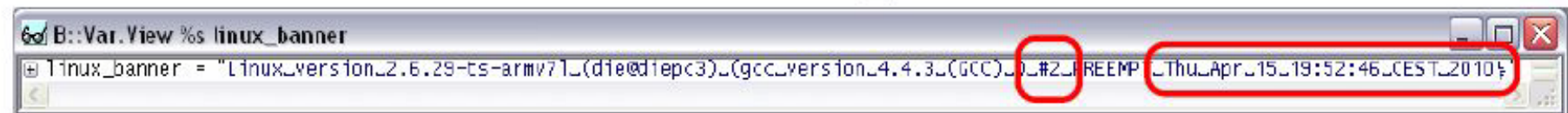
Checking version on Target

- Linux version running on the target:
 - In the target shell, use “cat /proc/version”:



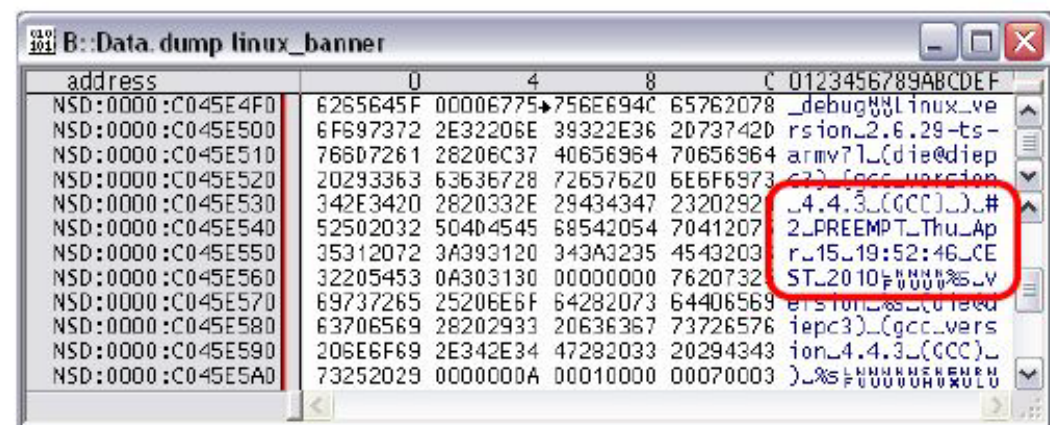
```
B::TERM
root@beagleboard:~# cat /proc/version
Linux version 2.6.29-ts-armv7l (die@diepc3) (gcc version 4.4.3 (GCC)) #2 PREEMPT
Thu Apr 15 19:52:46 CEST 2010
```

- In TRACE32 use “Var.View %s linux_banner”



```
B::Var.View %s linux_banner
Linux_banner = "Linux_version_2.6.29-ts-armv7l(die@diepc3)(gcc_version_4.4.3_(GCC))_#2_PREEMPT_Thu_Apr_15_19:52:46_CEST_2010"
```

- Or use “Data.dump linux_banner”



address	0	4	8	C 0123456789ABCDEF
NSD:0000:C045E4F0	6265645F	00006775	756E694C	65762078 _debugLinux_ve
NSD:0000:C045E500	6F697372	2E32206E	39322E36	20737420 rsion_2.6.29-ts-
NSD:0000:C045E510	76607261	28206C37	40656964	70656964 armv7l(die@diep
NSD:0000:C045E520	20293363	63636728	72657620	6E6F6973 32) gcc version
NSD:0000:C045E530	342E3420	2820332E	29434347	2320292 4.4.3_(GCC))_#
NSD:0000:C045E540	52502032	50404545	68542054	7041207 2_PREEMPT_Thu_Ap
NSD:0000:C045E550	35312072	3A393120	343A3235	4543203 r_15_19:52:46_CE
NSD:0000:C045E560	32205453	0A303130	00000000	7620732 ST_2010_#0000%sv
NSD:0000:C045E570	69737265	25206E6F	64282073	64406569 erston%sv
NSD:0000:C045E580	63706569	28202933	20636367	73726576 iepc3)(gcc_vers
NSD:0000:C045E590	206E6F69	2E342E34	47282033	20294343 ion_4.4.3_(GCC)_
NSD:0000:C045E5A0	73252029	0000000A	00010000	00070003)_%s#0000000000

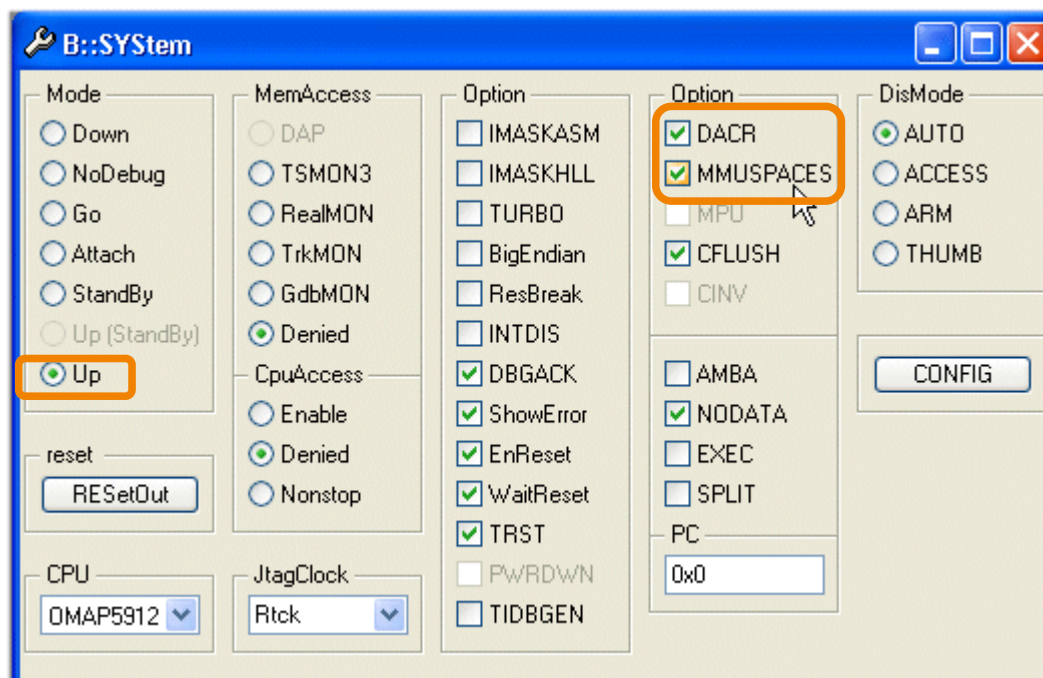
Debugger-Setup for Linux-Debugging

- Debugger-Setup before SYStem.Up (e.g. by script linux.cmm)
 - Select the **CPU type**, **JTAG-Clock** and appropriate **SYStem-Options** for your target (refer to the manual "debugger_arm.pdf" for details).
 - **Linux-Specific** debugger preparations:
 - SYStem.Option DACR ON (global debugger write-permission)
 - SYStem.Option MMUSPACES ON/OFF (for correct symbol handling ON after Kernel-Start-up is finished)
 - TrOnchip.Set PABORT OFF (allow Linux program page misses)
 - TrOnchip.Set DABORT OFF (allow Linux data page misses)
 - TrOnchip.Set UNDEF OFF (allow Linux the FPU detection)

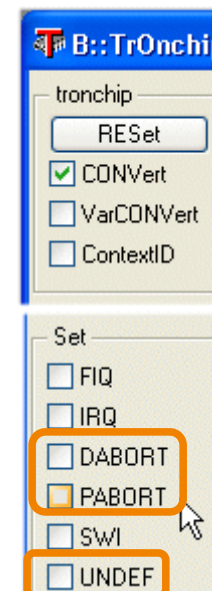
Debugger-Setup for Linux-Debugging

- Debugger-Settings before SYStem.Up:

Debugger SYStem control window



On-chip Trigger control window



Target-Setup for Linux-Debugging

- Target-Setup after SYStem.Up (e.g. by linux.cmm)

After **SYStem.Up** the communication via JTAG is established.

- **Let the boot monitor run** (e.g. Uboot from target flash) **to initialize the target HW** . To debug the kernel-start-up you need to stop the boot- monitor in time (breakpoint- or time-triggered).

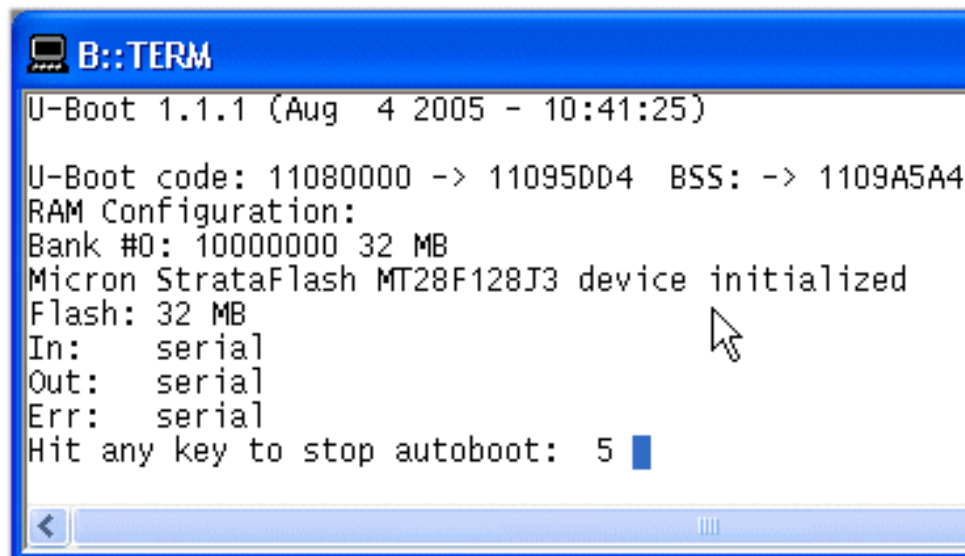
OR

- **Use debugger scripts to initialize the target HW.**
You can start a script e.g. by "DO init-target.cmm" within "linux.cmm".
It needs to do the same things as the boot-monitor usually does.

Target-Setup for Linux-Debugging

- Target-Setup by running the boot-monitor

This is the easy way to initialize your target HW. A very common boot-monitor program is the bootloader Uboot.



```
B::TERM
U-Boot 1.1.1 (Aug  4 2005 - 10:41:25)

U-Boot code: 11080000 -> 11095DD4  BSS: -> 1109A5A4
RAM Configuration:
Bank #0: 10000000 32 MB
Micron StrataFlash MT28F128J3 device initialized
Flash: 32 MB
In:  serial
Out: serial
Err: serial
Hit any key to stop autoboot:  5
```

- **To debug the Kernel-Start-up** you need to stop the boot-monitor in time. It can be done time-triggered during a countdown (e.g. Uboot parameter "bootdelay") or breakpoint-triggered within the delay-loop for example.

LINUX & Android Debugging

Frank,Xing

- ◆ Platform and Debug Overview
 - Linux Debugging
 - ◆ Configuration Linux-Aware Debugging
 - Stop-Mode-Debugging
 - ◆ Stop & Run-Mode-Debugging
- ◆ VM-Dalvik Debugging

LINUX Debugging

➤ Stop-Mode-Debugging

- Debugging different Linux parts
 - Kernel-Start-up
 - Kernel (inclusive static device drivers)
 - Kernel-Modules (inclusive dynamic device drivers)
 - Processes
 - Libraries
 - Threads

Debugging the Kernel-Start-up

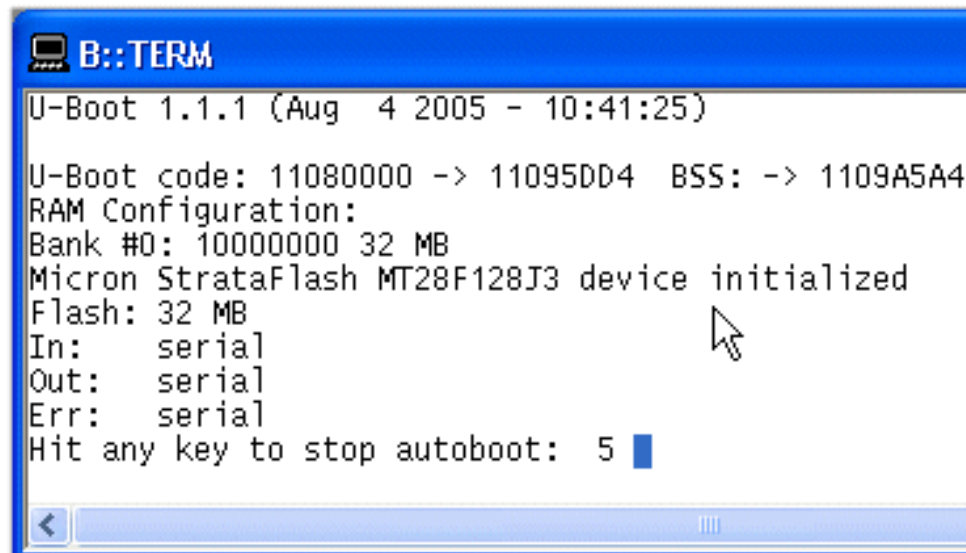
- **Prior to start debugging the Kernel-Start-up:**
 - The target HW needs to be initialized (e.g. by the bootloader Uboot).
 - The Linux-Kernel has to be loaded to the physical RAM address.
 - Also the **debug symbols of vmlinux need to be loaded temporarily to the physical addresses**.
(Reason: As long as the Kernel-Start-up isn't finished the MMU makes no active address translation and so the symbols of the Kernel-Start-up are still based on physical addresses.)

Data.LOAD.Elf vmlinux 0x10008000-0xC0008000 /NOCODE

This can be done in one step when loading the binary part of vmlinux also via the debugger. Then option /NOCODE is omitted.

Debugging the Kernel-Start-up

- Stop the bootloader just before it starts Linux
 - During the countdown shown by the bootloader on the terminal window.
(The duration is defined for example by Uboot parameter "bootdelay".)



```
B::TERM
U-Boot 1.1.1 (Aug 4 2005 - 10:41:25)

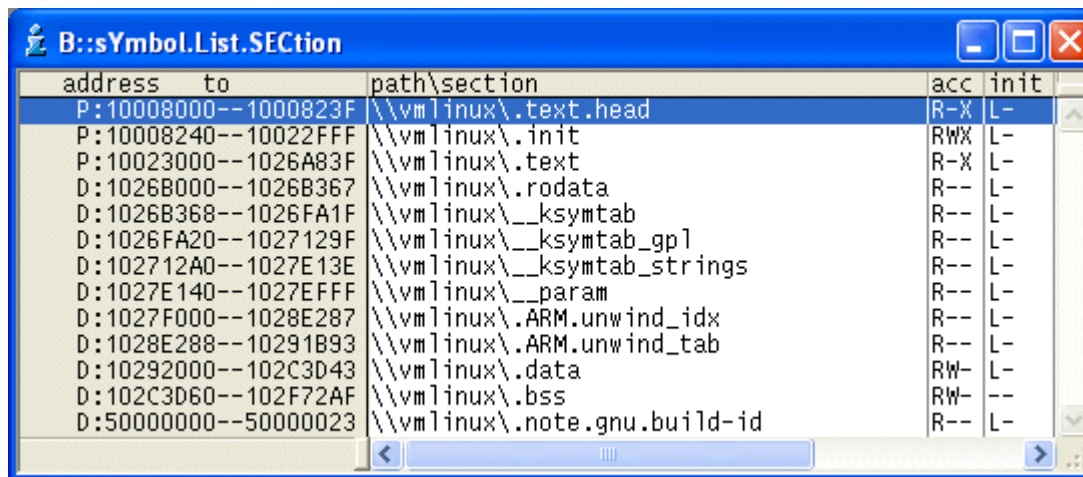
U-Boot code: 11080000 -> 11095DD4 BSS: -> 1109A5A4
RAM Configuration:
Bank #0: 10000000 32 MB
Micron StrataFlash MT28F128J3 device initialized
Flash: 32 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 5
```

OR

- By a breakpoint at phys. kernel start-address "stext" (label "__init_begin")
(It needs to be an Onchip-BP if loading the kernel image vmlinux via
bootloader.)

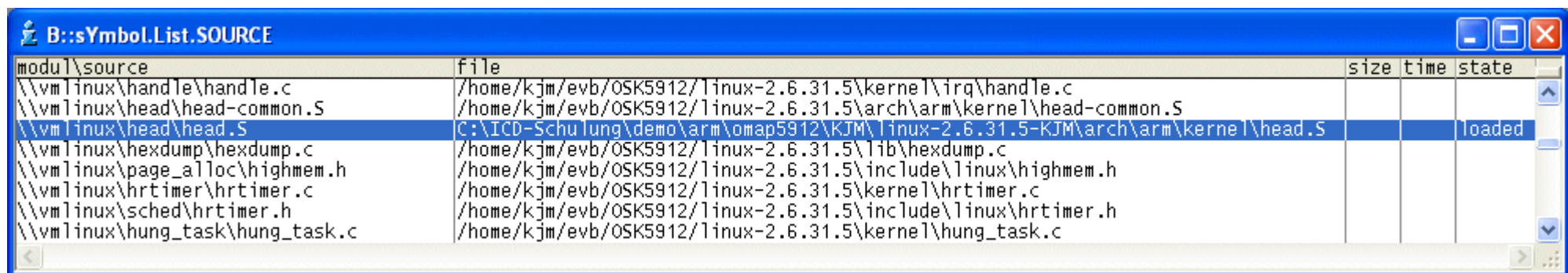
Debugging the Kernel-Start-up

- **sYmbol.List.SEction**; shows address ".text.head" where the kernel starts.
- You get a display of the program code by a double-click on the specific line. This causes the debugger to load the source file and display the right lines.



The screenshot shows a window titled "B::sYmbol.List.SEction" with a table of memory sections. The first row is selected, showing the ".text.head" section at address P:10008000--1000823F.

address	to	path\section	acc	init
P:10008000--1000823F		\\vmlinux\\.text.head	R-X	L-
P:10008240--10022FFF		\\vmlinux\\.init	RWX	L-
P:10023000--1026A83F		\\vmlinux\\.text	R-X	L-
D:1026B000--1026B367		\\vmlinux\\.rodata	R--	L-
D:1026B368--1026FA1F		\\vmlinux_ksymtab	R--	L-
D:1026FA20--1027129F		\\vmlinux_ksymtab_gpl	R--	L-
D:102712A0--1027E13E		\\vmlinux_ksymtab_strings	R--	L-
D:1027E140--1027EFFF		\\vmlinux_param	R--	L-
D:1027F000--1028E287		\\vmlinux\\.ARM.unwind_idx	R--	L-
D:1028E288--10291B93		\\vmlinux\\.ARM.unwind_tab	R--	L-
D:10292000--102C3D43		\\vmlinux\\.data	RW-	L-
D:102C3D60--102F72AF		\\vmlinux\\.bss	RW-	--
D:50000000--50000023		\\vmlinux\\.note.gnu.build-id	R--	L-



The screenshot shows a window titled "B::sYmbol.List.SOURCE" with a table of source files. The third row is selected, showing the file "C:\ICD-Schulung\demo\arm\omap5912\KJM\linux-2.6.31.5-KJM\arch\arm\kernel\head.S".

modul\source	file	size	time	state
\\vmlinux\handle\handle.c	/home/kjm/evb/OSK5912/linux-2.6.31.5\kernel\irq\handle.c			
\\vmlinux\head\head-common.S	/home/kjm/evb/OSK5912/linux-2.6.31.5\arch\arm\kernel\head-common.S			
\\vmlinux\head\head.S	C:\ICD-Schulung\demo\arm\omap5912\KJM\linux-2.6.31.5-KJM\arch\arm\kernel\head.S			loaded
\\vmlinux\hexdump\hexdump.c	/home/kjm/evb/OSK5912/linux-2.6.31.5\lib\hexdump.c			
\\vmlinux\page_alloc\highmem.h	/home/kjm/evb/OSK5912/linux-2.6.31.5\include\linux\highmem.h			
\\vmlinux\hrtimer\hrtimer.c	/home/kjm/evb/OSK5912/linux-2.6.31.5\kernel\hrtimer.c			
\\vmlinux\sched\hrtimer.h	/home/kjm/evb/OSK5912/linux-2.6.31.5\include\linux\hrtimer.h			
\\vmlinux\hung_task\hung_task.c	/home/kjm/evb/OSK5912/linux-2.6.31.5\kernel\hung_task.c			

Debugging the Kernel-Start-up

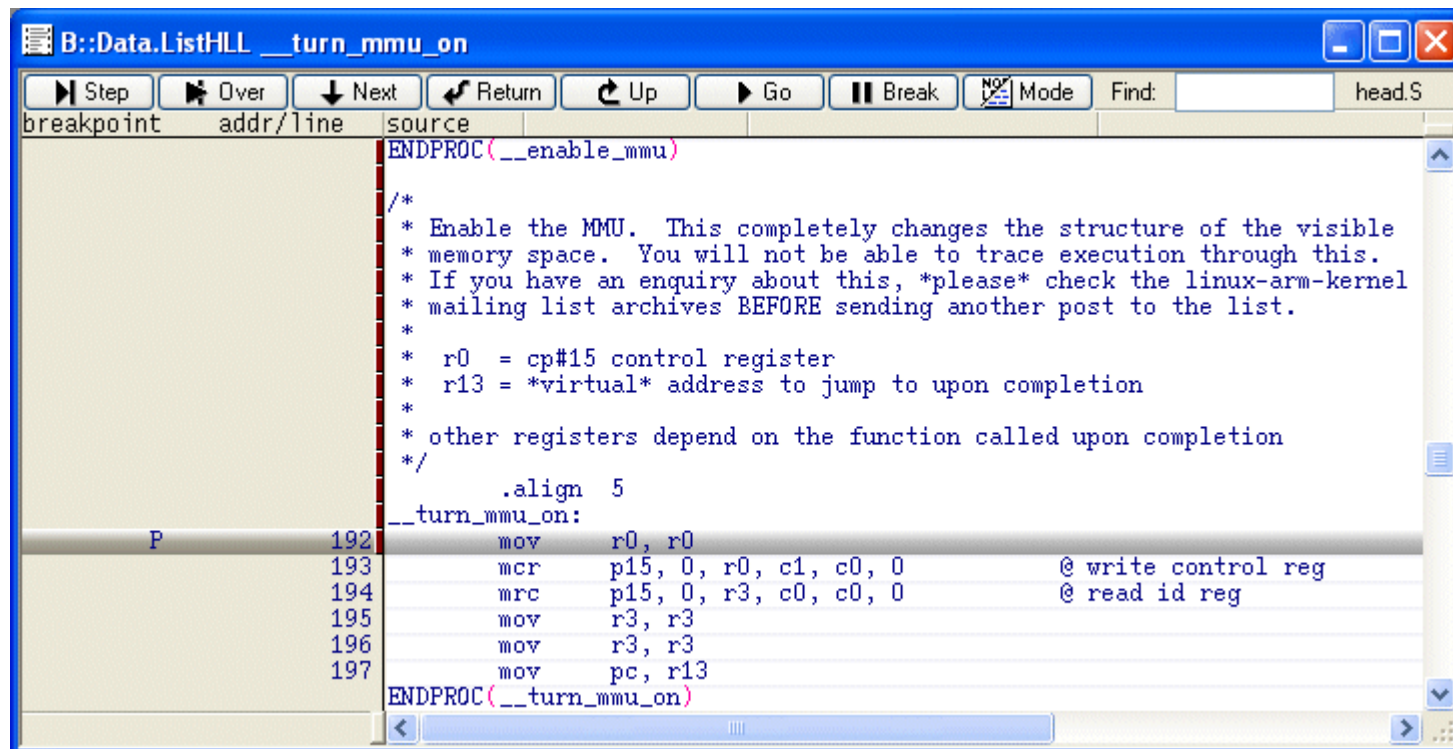
- Break.Set stext /Onchip can stop the Kernel-Start at "_init_begin"

addr/line	code	label	mnemonic	comment
SR:10007FF8	00000000		andeq r0,r0,r0	
SR:10007FFC	00000000		andeq r0,r0,r0	
			...	
			* numbers for r1.	
			* We're trying to keep crap to a minimum; DO NOT add any machine specific	
			* crap here - that's what the boot loader (or in extreme, well justified	
			* circumstances, zImage) is for.	
			*/	
			.section ".text.head", "ax"	
			ENTRY(stext)	
79			msr cpsr_c, #PSR_F_BIT PSR_I_BIT SVC_MODE @ ensure svc mode	
SR:10008000	E321F0D3	__init_begin:	msr cpsr_c, #0xD3	
				@ and irqs disabled
81			mrc p15, 0, r9, c0, c0	@ get processor id
SR:10008004	EE109F10		mrc p15,0x0,r9,c0,c0,0x0; p15,0,r9,c0,c0,0 (id)	
82			bl __lookup_processor_type	@ r5=procinfo r9=cpuuid
SR:10008008	EB000055		bl 0x10008164	; __lookup_processor_type
83			movs r10, r5	@ invalid processor (r5=0)?
SR:1000800C	E1B0A005		movs r10,r5	
84			beq __error_p	@ yes, error 'p'
SR:10008010	0A000051		beq 0x1000815C	; __error
85			bl __lookup_machine_type	@ r5=machinfo
SR:10008014	EB00006A		bl 0x100081C4	; __lookup_machine_type
86			movs r8, r5	@ invalid machine (r5=0)?

address	types	impl	name
R:10008000	Program	ONCHIP	__init_begin
R:10008060	Program	ONCHIP	__turn_mmu_on


Debugging the Kernel-Start-up

- After "__enable_mmu" the Kernel-Start-up is finished with "__turn_mmu_on".
By the last instruction ... (see next page)



The screenshot shows a window titled "B::Data.ListHLL __turn_mmu_on". It contains a list of assembly instructions with their addresses and source code. The instructions are as follows:

breakpoint	addr/line	source
		ENDPROC(__enable_mmu)
		/*
		* Enable the MMU. This completely changes the structure of the visible
		* memory space. You will not be able to trace execution through this.
		* If you have an enquiry about this, *please* check the linux-arm-kernel
		* mailing list archives BEFORE sending another post to the list.
		*
		* r0 = cp#15 control register
		* r13 = *virtual* address to jump to upon completion
		* other registers depend on the function called upon completion
		*/
		.align 5
		__turn_mmu_on:
P	192	mov r0, r0
	193	mcr p15, 0, r0, c1, c0, 0 @ write control reg
	194	mrc p15, 0, r3, c0, c0, 0 @ read id reg
	195	mov r3, r3
	196	mov r3, r3
	197	mov pc, r13
		ENDPROC(__turn_mmu_on)

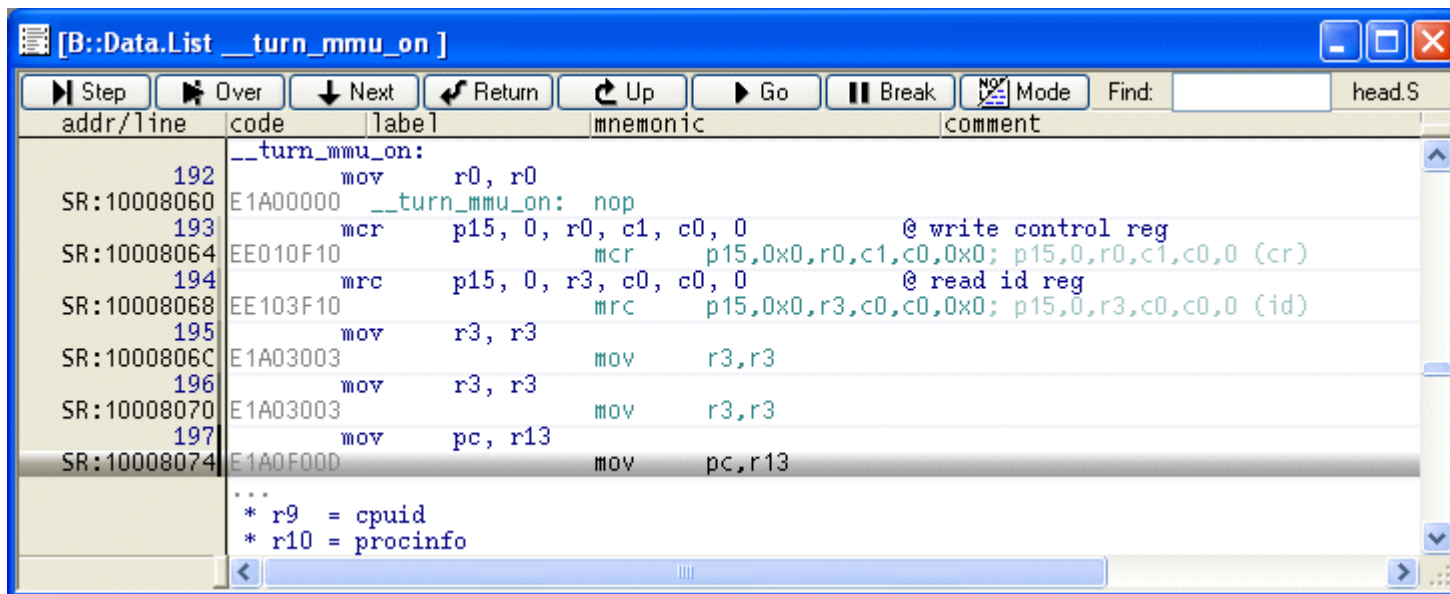


The screenshot shows a window titled "B::Break.List". It contains a table with columns: address, types, impl, and name. The table has two rows of data:

address	types	impl	name
R:10008000	Program	ONCHIP	__init_begin
R:10008060	Program	ONCHIP	__turn_mmu_on

Debugging the Kernel-Start-up

- Linux switches to virtual addresses by activating the MMU-Translation.



The screenshot shows a debugger window titled "[B::Data.List __turn_mmu_on]". The window contains a table of assembly instructions with columns for address/line, code, label, mnemonic, and comment. The instructions are as follows:

addr/line	code	label	mnemonic	comment
		__turn_mmu_on:		
192			mov r0, r0	
SR:10008060	E1A00000	__turn_mmu_on:	nop	
193			mcr p15, 0, r0, c1, c0, 0	@ write control reg
SR:10008064	EE010F10		mcr p15,0x0,r0,c1,c0,0x0; p15,0,r0,c1,c0,0 (cr)	
194			mrc p15, 0, r3, c0, c0, 0	@ read id reg
SR:10008068	EE103F10		mrc p15,0x0,r3,c0,c0,0x0; p15,0,r3,c0,c0,0 (id)	
195			mov r3, r3	
SR:1000806C	E1A03003		mov r3,r3	
196			mov r3, r3	
SR:10008070	E1A03003		mov r3,r3	
197			mov pc, r13	
SR:10008074	E1A0F00D		mov pc,r13	
			...	
			* r9 = cpuid	
			* r10 = procinfo	

- Now it is time to **load the symbols of vmlinux to the virtual addresses**. At latest when you use symbolic expressions or want to get RTOS information.

Debugging the Kernel-Start-up

- Background info

[B::Data.List]

addr/line	code	label	mnemonic	comment
SR:C0008060	E1A00000		nop	
SR:C0008064	EE010F10		mcr	p15,0x0,r0,c1,c0,0x0; p15,0,r0,c1,c0,0 (cr)
SR:C0008068	EE103F10		mrc	p15,0x0,r3,c0,c0,0x0; p15,0,r3,c0,c0,0 (id)
SR:C000806C	E1A03003		mov	r3,r3
SR:C0008070	E1A03003		mov	r3,r3
SR:C0008074	E1A0F00D		mov	pc,r3 ← our former jump code
SR:C0008078	E59F4064		ldr	r4,0xC00080E4

B::Data.List

addr/line	code	label	mnemonic	comment
SR:C0008110	C0293FF8		strgtb	r3,[r9],-r8
SR:C0008114	E24F302C		sub	r3,pc,#0x2C ← the jump destination
SR:C0008118	E8B300F0		ldmia	r3!,{r4-r7}
SR:C000811C	E1540005		cmp	r4,r5
SR:C0008120	11550006		cmpne	r5,r6
SR:C0008124	1494B004		ldrne	r11,[r4],#0x4
SR:C0008128	1485B004		strne	r11,[r5],#0x4

B::MMU.List PageTable

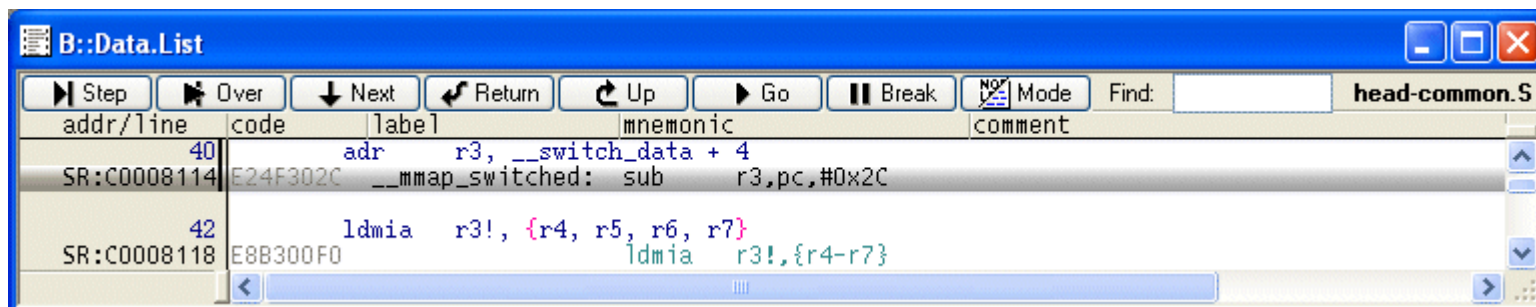
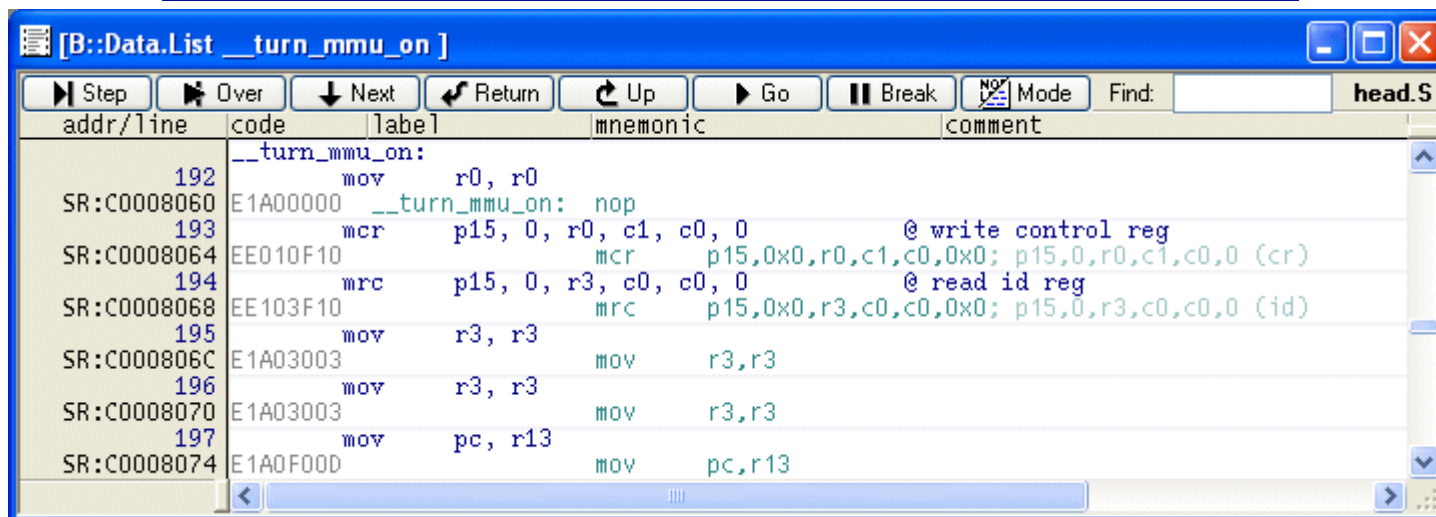
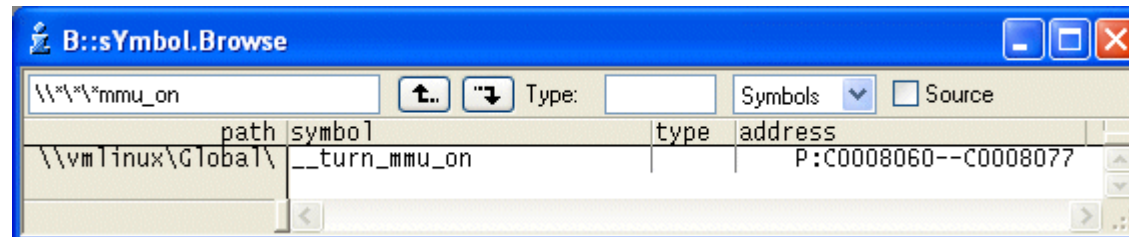
address	physical	d	size	permissions	glb	shr
C:00000000--0FFFFFFF						
C:10000000--100FFFFFFF	A:10000000--100FFFFFFF	00	00100000	P:readwrite U:readwrite	yes	no
C:10100000--BFFFFFFF						
C:C0000000--C02FFFFF	A:10000000--102FFFFFFF	0 ← reason why it works	00100000			no
C:C0300000--FFFFFFF						

Debugging the Linux-Kernel

- Before loading the debug symbols of the kernel to the virtual addresses you need to activate the debugger MMUSPACES and the MMU related debugger Translation.
- Changing to **SYStem.MMUSPACES ON** requires to have NO symbols loaded!
sYmbol.RESet ; delete all loaded debug symbols
- **Data.LOAD.Elf vmlinux /NOCODE** ; the code has been loaded before
- Now that **symbol "swapper_pg_dir"** is available at its virtual address you can apply the necessary **TRANSlation commands**.
- Since the kernel is mapped in a continuous address range all symbols are immediately usable.

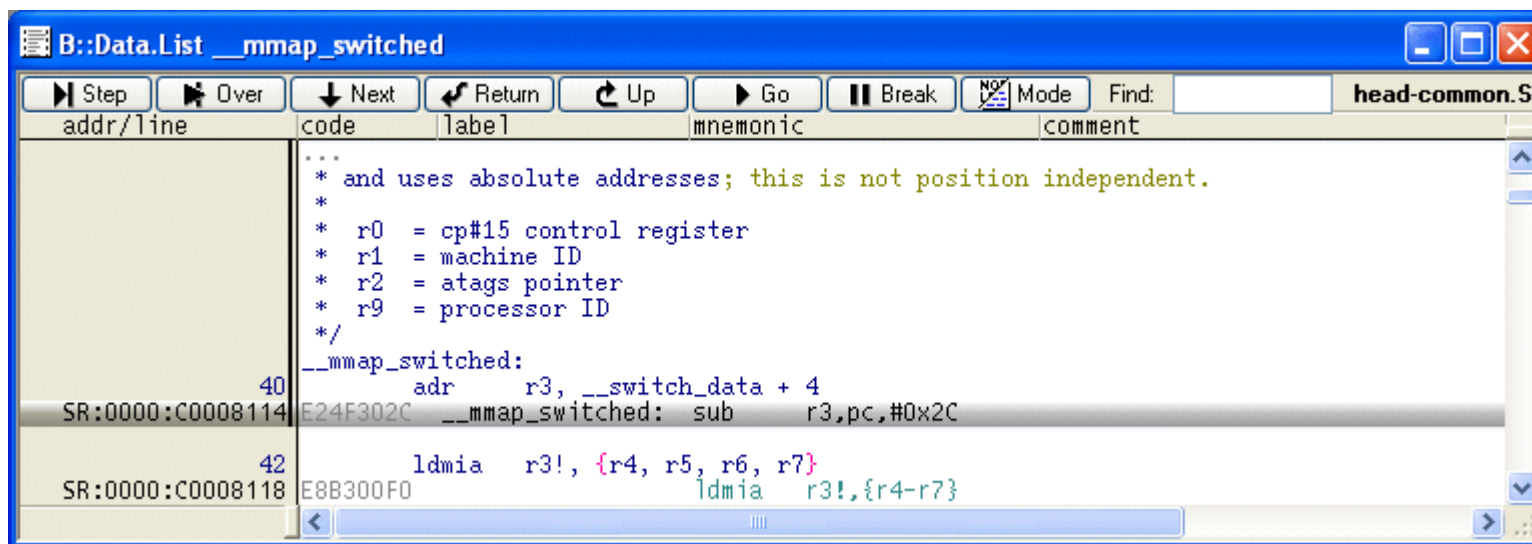
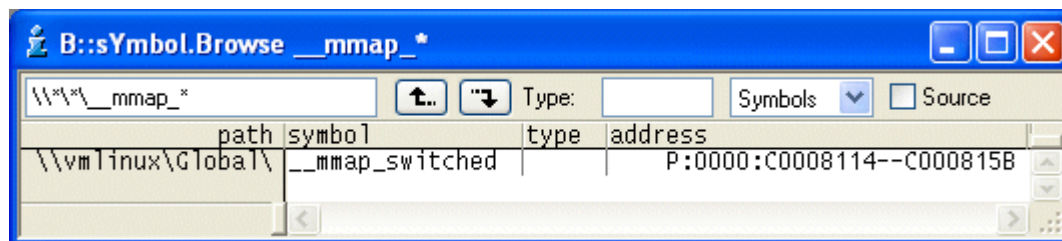
Debugging the Linux-Kernel

- Background info (virtual symbols are loaded but MMUSPACES are still OFF)



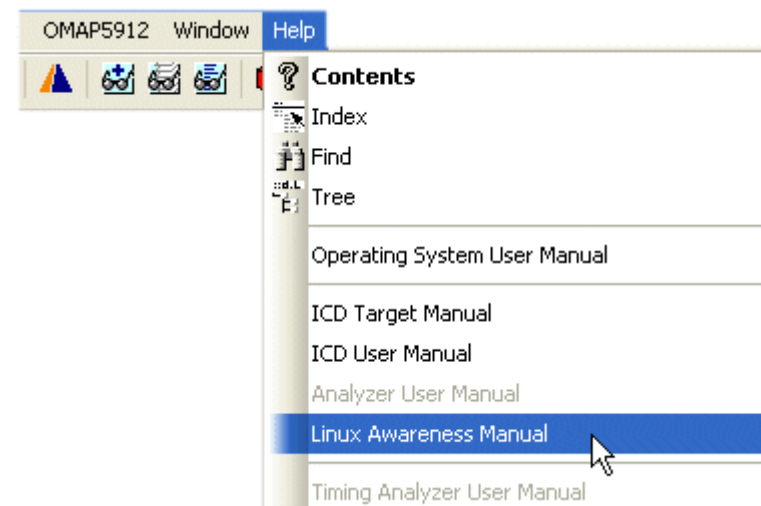
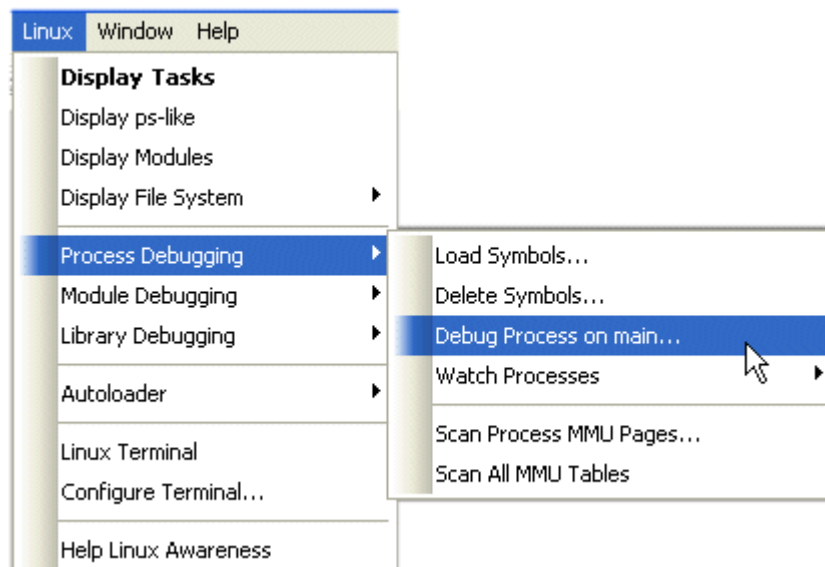
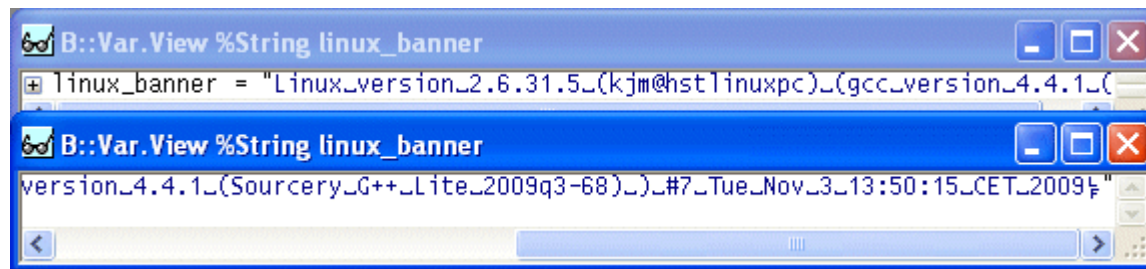
Debugging the Linux-Kernel

- The windows show now the debuggers **SpaceIDs** after command **SYStem.Option MMUSPACES ON**. The **kernel SpaceID** is always **“0000:”** plus a “memory-class”. “SR:” means Supervisor 32-Bit ARM access. The SpaceID is needed to handle the symbols for different processes on the same virtual address.

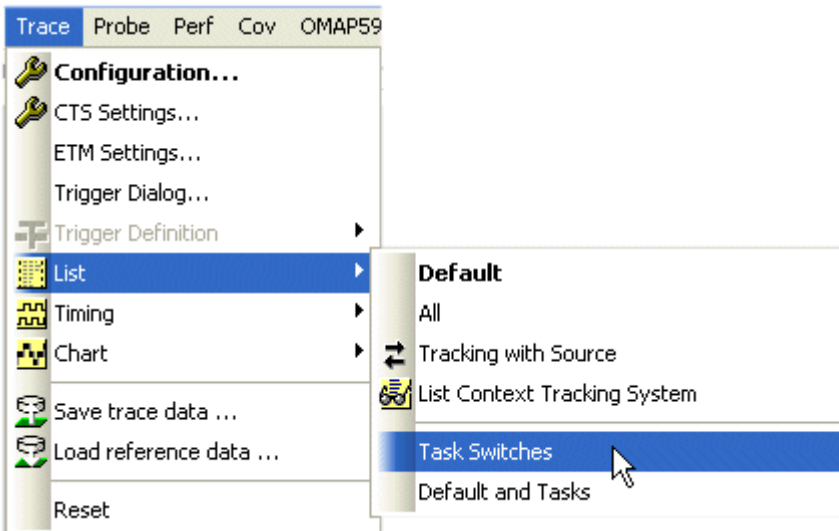
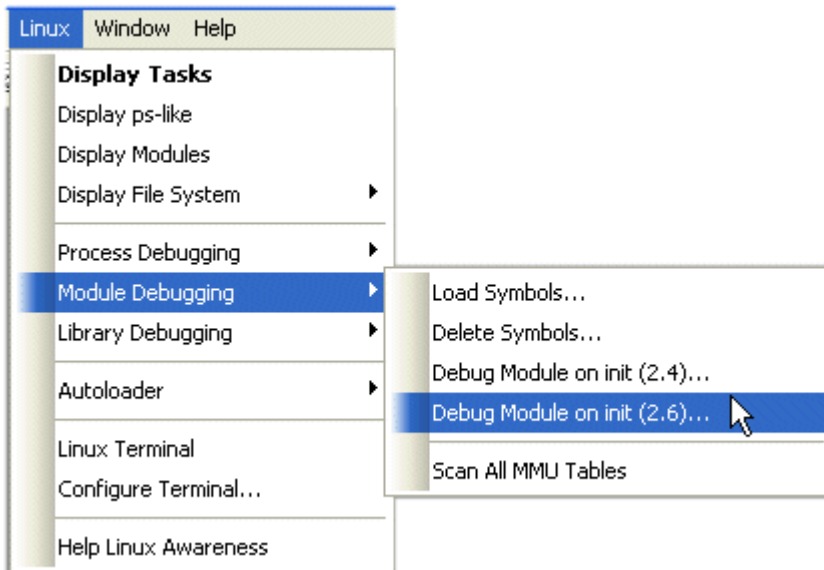


Debugging the Linux-Kernel

- After all preparations for Linux-Debugging and the kernel is up you are able to use the TRACE32 RTOS-Awareness and TRACE32 Linux-Menu.

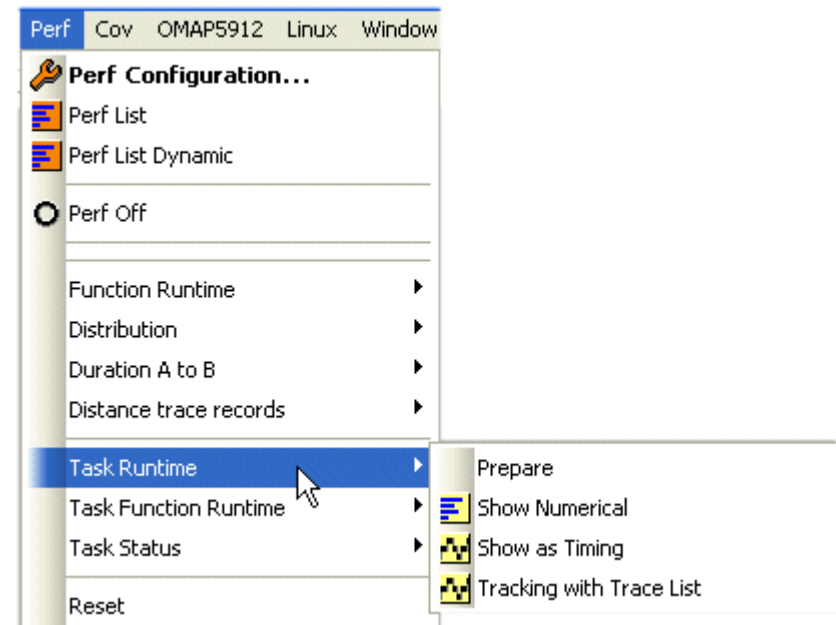


Debugging the Linux-Kernel



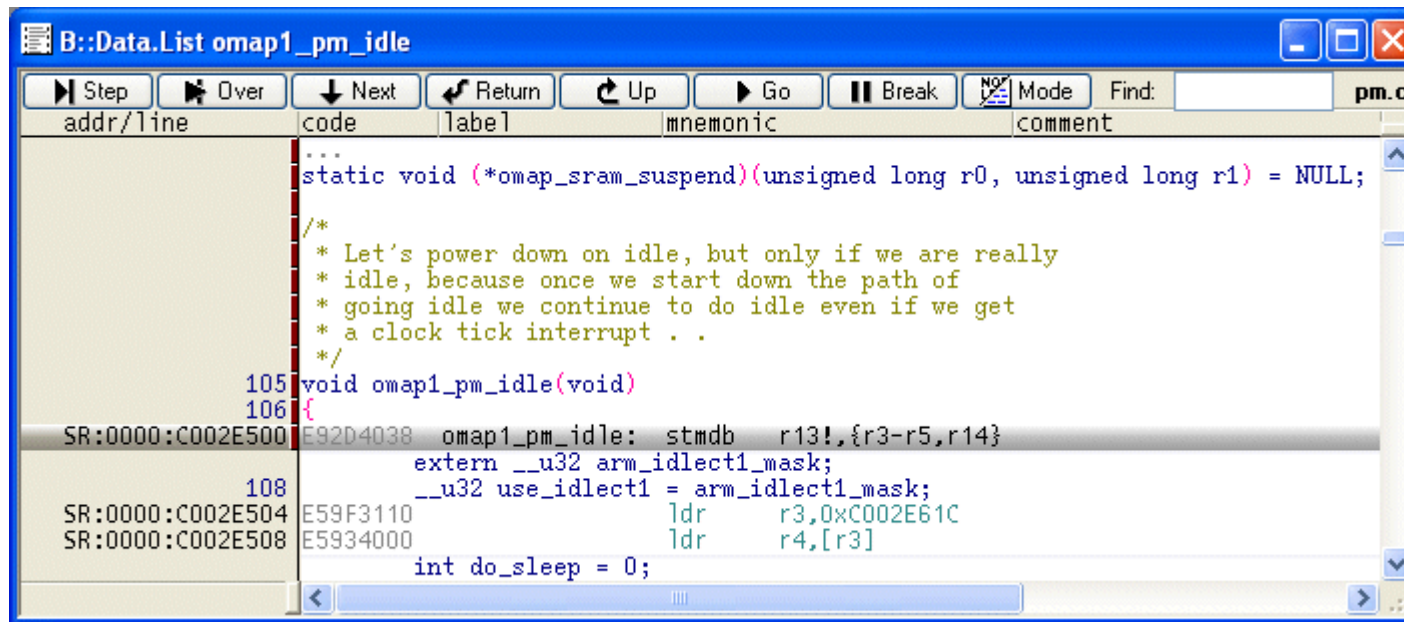
Several TRACE32 Linux-Menu items will ease Linux-Debugging for you.

Features related to tracing are not subject to this documentation. Refer to "Tracing Linux".



Debugging the Linux-Kernel

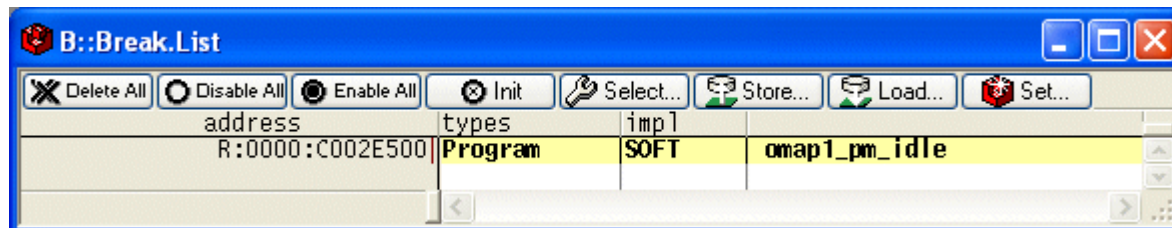
- Example: Break on the kernel-loop for the power-management of the board.



The screenshot shows the 'B::Data.List omap1_pm_idle' window. It displays the source code for the 'omap1_pm_idle' function, which is a static void function. The code includes comments about power management and a loop for sleeping. The assembly code is also visible, showing instructions like 'stmdb r13!, {r3-r5, r14}', 'ldr r3, 0xC002E61C', and 'ldr r4, [r3]'. The window has a toolbar with buttons for Step, Over, Next, Return, Up, Go, Break, and Mode. The 'Find' field is empty, and the 'pm.c' file is selected.

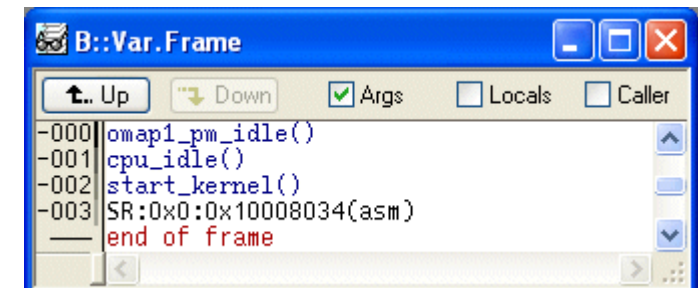
```
...
static void (*omap_sram_suspend)(unsigned long r0, unsigned long r1) = NULL;

/*
 * Let's power down on idle, but only if we are really
 * idle, because once we start down the path of
 * going idle we continue to do idle even if we get
 * a clock tick interrupt . .
 */
105 void omap1_pm_idle(void)
106 {
SR:0000:C002E500 E92D4038 omap1_pm_idle: stmdb r13!, {r3-r5, r14}
                extern __u32 arm_idlect1_mask;
                __u32 use_idlect1 = arm_idlect1_mask;
108 SR:0000:C002E504 E59F3110                ldr r3, 0xC002E61C
SR:0000:C002E508 E5934000                ldr r4, [r3]
                int do_sleep = 0;
```



The screenshot shows the 'B::Break.List' window. It displays a table of break points. The first break point is at address 'R:0000:C002E500', with type 'Program', impl 'SOFT', and name 'omap1_pm_idle'. The window has a toolbar with buttons for Delete All, Disable All, Enable All, Init, Select..., Store..., Load..., and Set....

address	types	impl	
R:0000:C002E500	Program	SOFT	omap1_pm_idle



The screenshot shows the 'B::Var.Frame' window. It displays the call stack for the 'omap1_pm_idle' function. The stack frames are listed from bottom to top: 'end of frame', 'SR:0x0:0x10008034(asm)', 'start_kernel()', 'cpu_idle()', and 'omap1_pm_idle()'. The window has a toolbar with buttons for Up, Down, Args, Locals, and Caller.

```
-000| omap1_pm_idle()
-001| cpu_idle()
-002| start_kernel()
-003| SR:0x0:0x10008034(asm)
    | end of frame
```

Debugging the Linux-Kernel

- Specific Linux related information / displays.

B::TASK.DTask

magic	command	state	uid	pid	spaceid
C0295678	swapper	current	192.	0.	0000
C1C20000	init	sleeping	256.	1.	0001
C1C20340	kthreadd	sleeping	451.	2.	0000
C1C20680	ksoftirqd/0	sleeping	511.	3.	0000
C1C209C0	watchdog/0	sleeping	316.	4.	0000
C1C20D00	events/0	sleeping	256.	5.	0000
C1C21040	khelper	sleeping	451.	6.	0000
C1C21A00	async/mgr	sleeping	316.	9.	0000
C1C21380	kblockd/0	sleeping	451.	75.	0000
C1C216C0	khungtaskd	sleeping	511.	109.	0000
C1C22700	pdflush	sleeping	511.	110.	0000
C1C23400	pdflush	sleeping	451.	111.	0000
C1C23740	kswapd0	sleeping	511.	112.	0000
C14F5040	aio/0	sleeping	451.	159.	0000
C14F4D00	mtdblockd	sleeping	256.	810.	0000
C14F5D40	pccardd	sleeping	256.	825.	0000
C14F56C0	sh	sleeping	511.	844.	034C

B::TASK.DTask

magic	command	state	uid	pid	spaceid
C0295678	swapper	current	192.	0.	0000
C1C20000	init				
C1C20340	kthreadd				
C1C20680	ksoftirqd/0				
C1C209C0	watchdog/0				
C1C20D00	events/0				
C1C21040	khelper				
C1C21A00	async/mgr				
C1C21380	kblockd/0				
C1C216C0	khungtaskd				
C1C22700	pdflush				
C1C23400	pdflush				
C1C23740	kswapd0				
C14F5040	aio/0				
C14F4D00	mtdblockd				
C14F5D40	pccardd				
C14F56C0	sh				

Display detailed

- Display task struct
- Display Stack Frame
- Display Registers
- Switch Context
- Load Process Symbols
- Delete Process Symbols
- Add to Watched Processes
- Delete from Watched Processes
- Scan MMU Pages
- Dump task entry
- Kill task

B::Var.View %m %s (struct task_struct*)0xC0295678

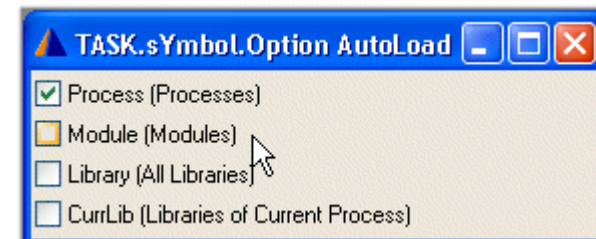
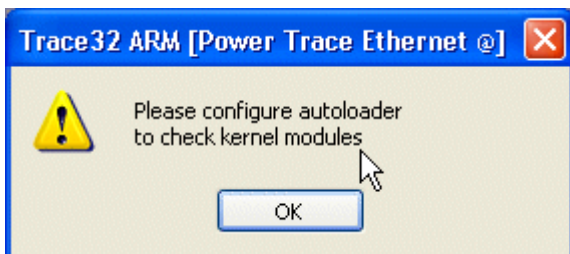
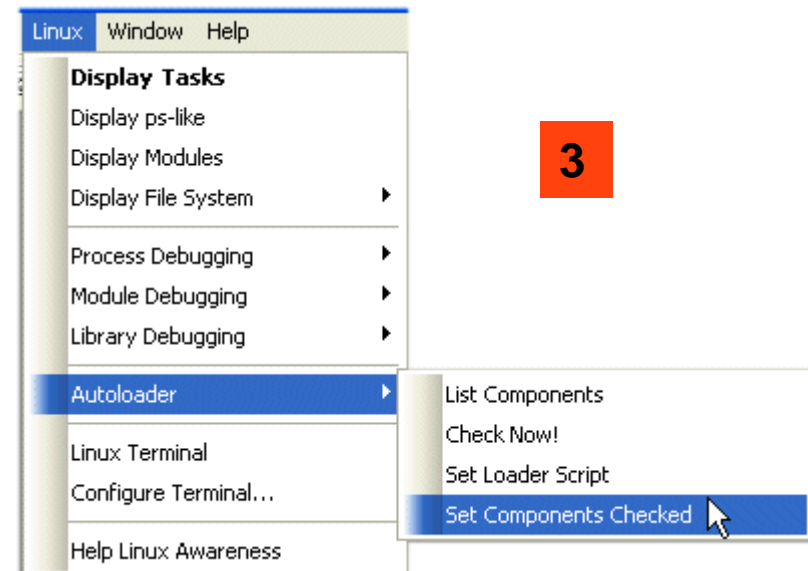
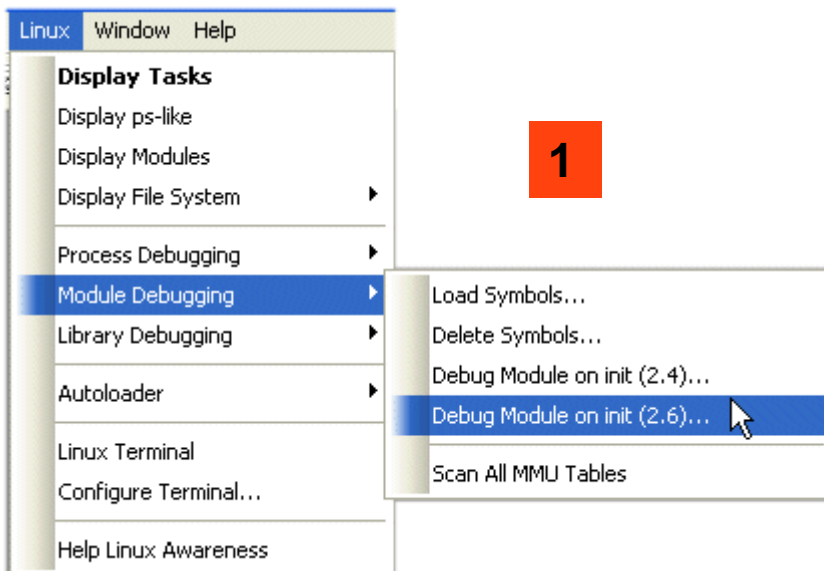
cpu_timers = ((next = 0xC0295838, prev = 0xC0295838), (next
real_cred = 0xC029CB14,
cred = 0xC029CB14,
cred_guard_mutex = (count = (counter = 1), wait_lock = (raw
comm = 'swapper',
link_count = 0,
total_link_count = 0,
sysvsem = (undo_list = 0x0),
last_switch_count = 0,
thread = (address = 0, trap_no = 0, error_code = 0, debug =
fs = 0xC02A695C,

B::Task.DTask 0xC0295678

magic	command	state	uid	pid	spaceid
C0295678	swapper	current	192.	0.	0000
gid	vm size	tty	name	path	
192.	-	-	-	-	
flags:	SYNCWRITE				
parent	youngest child	younger sibling	olde		
-	kthreadd	-	-		
arguments					

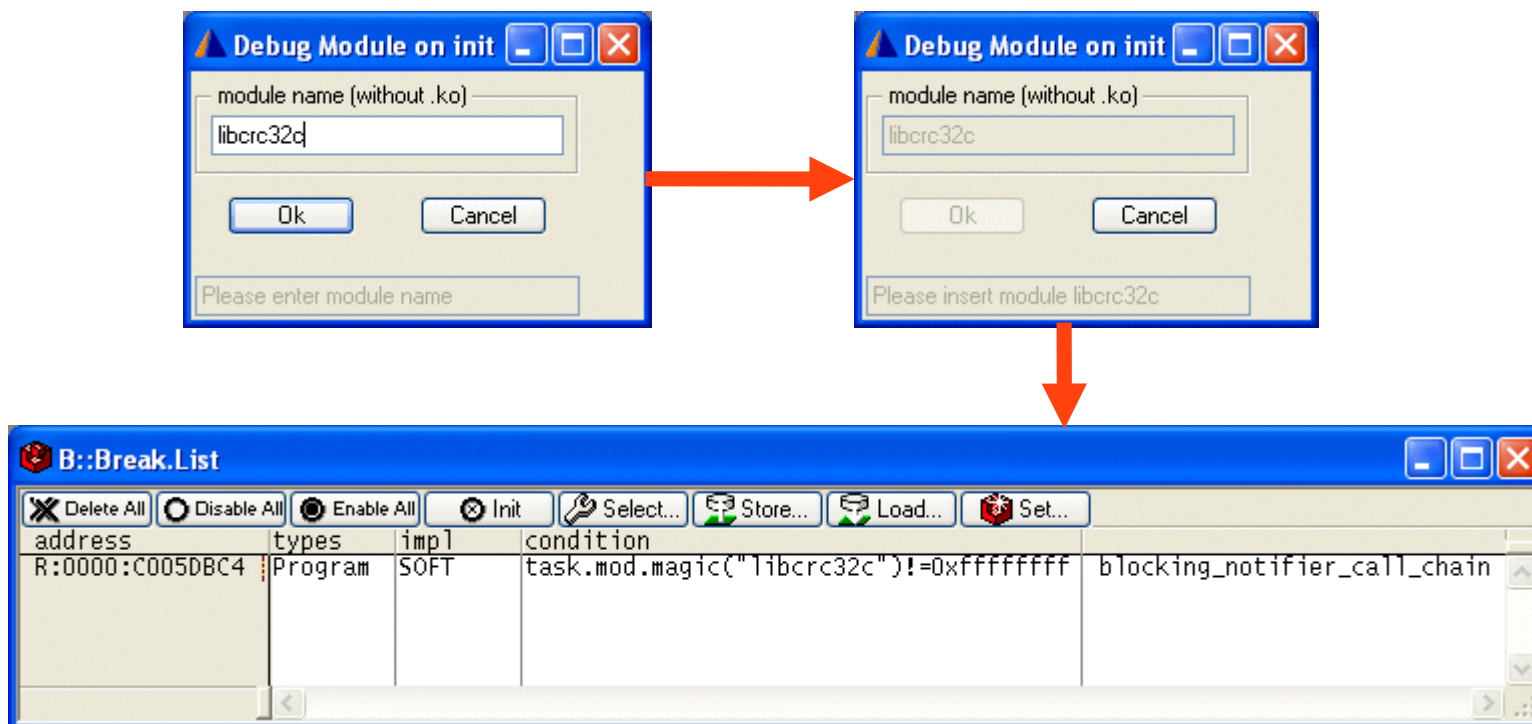
Debugging Linux-Kernel-Modules

- To stop on the start of a new Kernel-Module you can use the provided menu item "Debug Module on init".



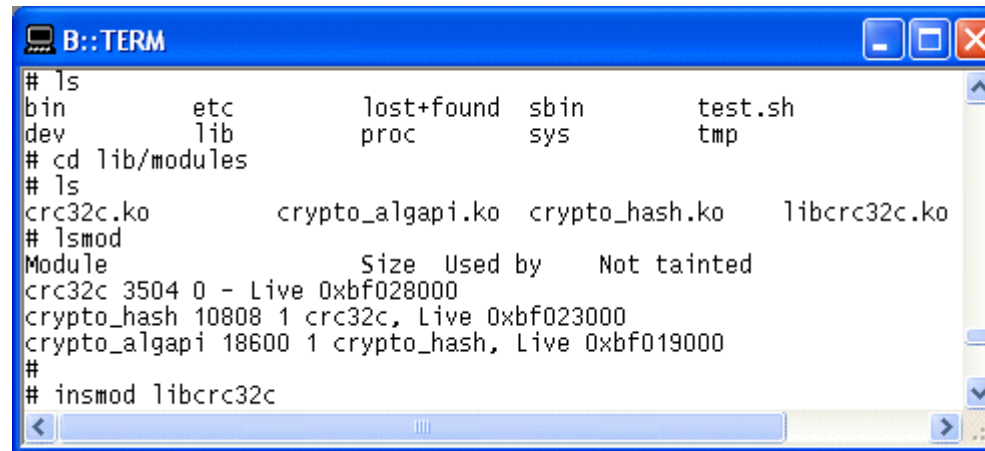
Debugging Linux-Kernel-Moduls

- The menu “Debug Module on init (2.x)” is used to stop on the begin of Kernel-Modules. It uses the commands of script “**mod_debug.cmm**” which can also be used directly with the name of the module as parameter.
- **DO mod_debug.cmm libcrc32c** ; without kernel name-extension “.ko”



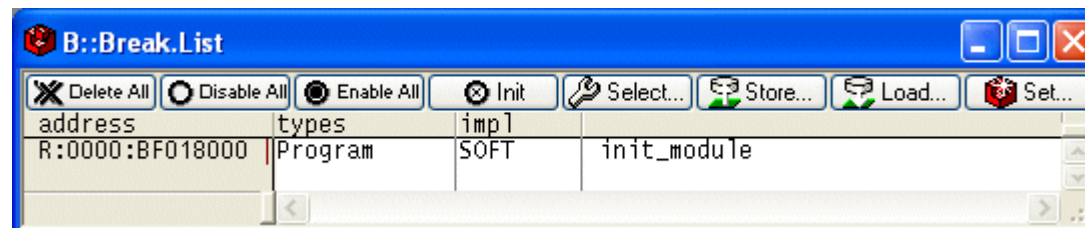
Debugging Linux-Kernel-Moduls

- Currently already 3 Kernel-Modules are loaded (state “Live”) on Linux and the next will be inserted now via the TERM window (CPU has to be running).



```
# ls
bin          etc          lost+found  sbin         test.sh
dev          lib          proc        sys          tmp
# cd lib/modules
# ls
crc32c.ko      crypto_algapi.ko  crypto_hash.ko  libcrc32c.ko
# lsmod
Module                  Size Used by    Not tainted
crc32c 3504 0 - Live 0xbf028000
crypto_hash 10808 1 crc32c, Live 0xbf023000
crypto_algapi 18600 1 crypto_hash, Live 0xbf019000
#
# insmod libcrc32c
```

- After the breakpoint at “**blocking_notifier_call_chain**” was reached and the module name is the same as used with Linux command “insmod” a **new breakpoint is set to “init_module**” - the begin of the new Kernel-Module.



Debugging Linux-Kernel-Modules

- The CPU stopped on the breakpoint at “init_module” (“__init”) the begin of the observed Kernel-Module “libcrc32c.ko”.

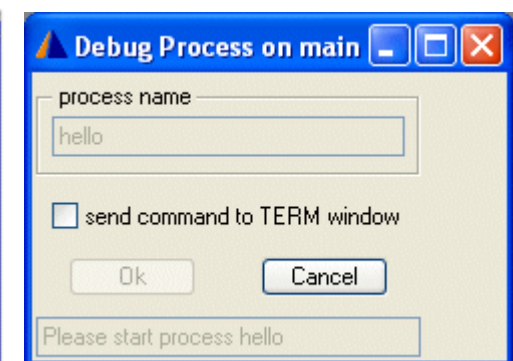
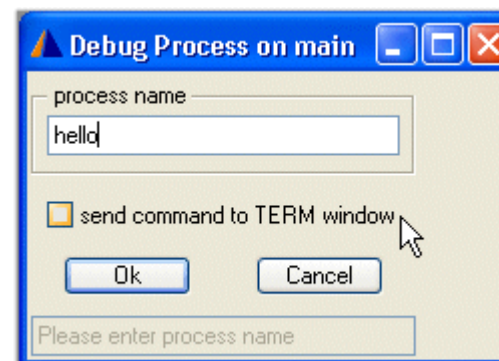
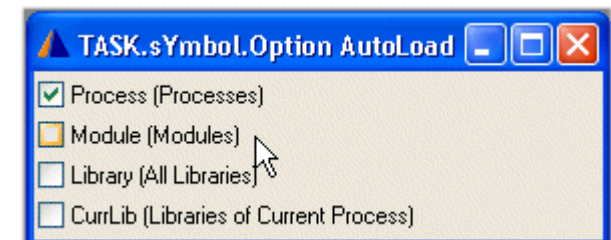
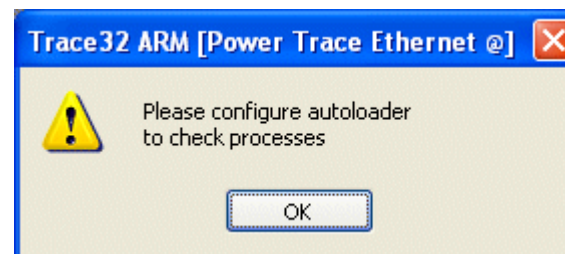
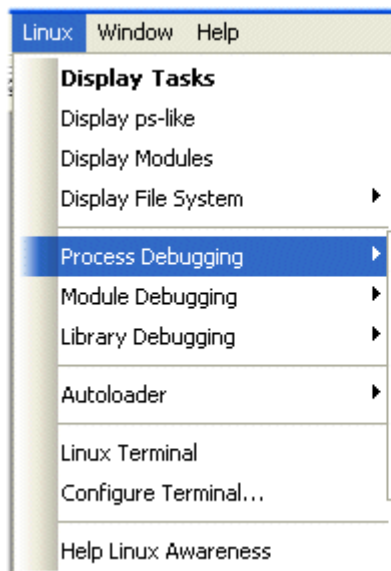
```
[B::Data.List libcrc32c_mod_init]
Step Over Next Return Up Go Break Mode Find: libcrc32c.c
addr/line code label mnemonic comment
SR:0000:BF036FFE 64 tfm = crypto_alloc_shash("crc32c", 0, 0);
EXPORT_SYMBOL(crc32c);
63 static int __init libcrc32c_mod_init(void)
SR:0000:BF037000 63 {
E3A01000 init_module: mov r1,#0x0
EXPORT_SYMBOL(crc32c);
63 static int __init libcrc32c_mod_init(void)
SR:0000:BF037004 63 {
E92D4038 stmdb r13!,{r3-r5,r14}
64 tfm = crypto_alloc_shash("crc32c", 0, 0);
```

- TASK.MODULE** lists now all 4 Kernel-Modules. The “address” below belongs to structure variable “module_core” not to function “init_module”! (Offset 0x3000.)

magic	name	state	size	address	refcount	depends
BF0347C4	libcrc32c	Loading	2424.	BF034000	1.	-
BF028C58	crc32c	Live	3504.	BF028000	0.	-
BF0258E0	crypto_hash	Live	10808.	BF023000	2.	libcrc32c crc32c
BF01D750	crypto_algapi	Live	18600.	BF019000	1.	crypto_hash

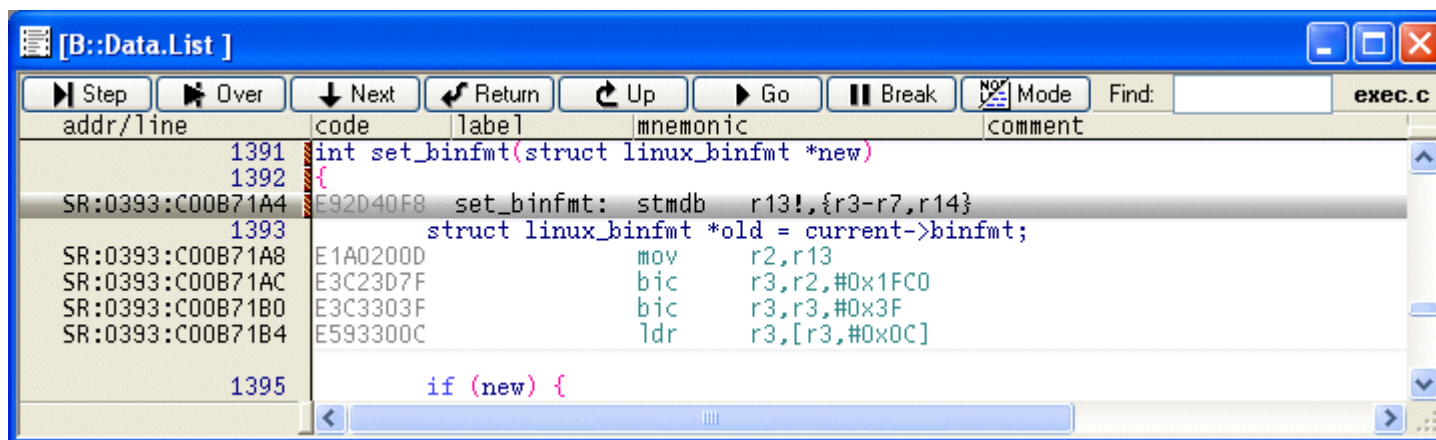
Debugging Linux-Processes (= Android Native Applications)

- If the process is not started yet you can use the TRACE32 Linux-Menu to stop on the function “main” of the process. If the function name where you want to stop the CPU is different then use the commands of script “**app_debug.cmm**” which is equal to using the menu item “**Debug Process on main ...**”.
- **DO app_debug.cmm hello** will stop on function “main” of process “hello”.



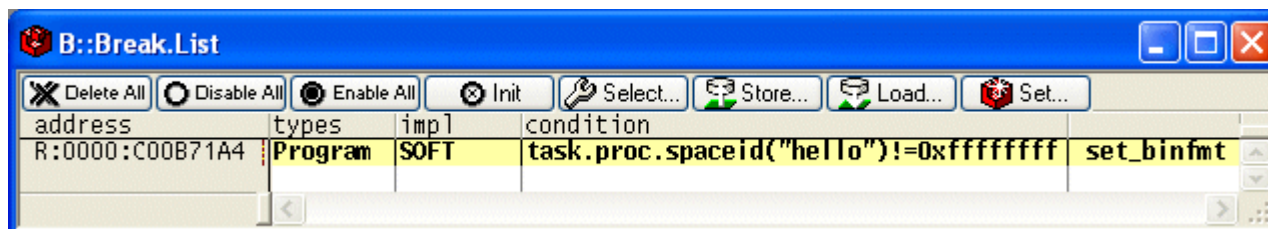
Debugging Linux-Processes

- Script “app_debug.cmm” uses a software **breakpoint on kernel-function** “**set_binfmt**” to check if the new process has the name observed (“hello”).



addr/line	code	label	mnemonic	comment
1391	int set_binfmt(struct linux_binfmt *new)			
1392	{			
SR:0393:C00B71A4	E92D40F8	set_binfmt:	stmdb r13!,{r3-r7,r14}	
1393				struct linux_binfmt *old = current->binfmt;
SR:0393:C00B71A8	E1A0200D		mov r2,r13	
SR:0393:C00B71AC	E3C23D7F		bic r3,r2,#0x1FC0	
SR:0393:C00B71B0	E3C3303F		bic r3,r3,#0x3F	
SR:0393:C00B71B4	E593300C		ldr r3,[r3,#0x0C]	
1395				if (new) {

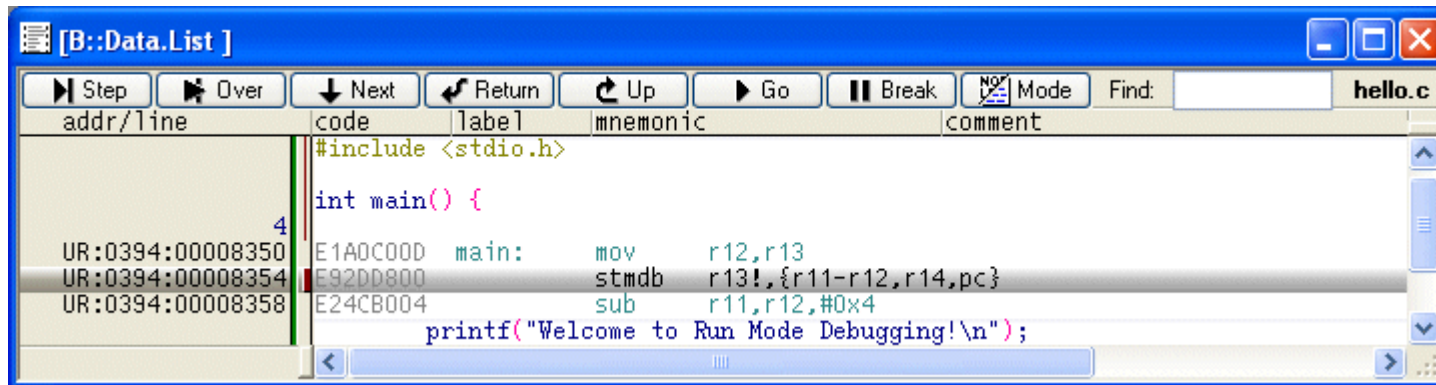
- If this breakpoint is reached and the new process the observed then a new breakpoint is set to function “**main+0x4**”. The offset is used to get sure that **Demand-Paging** was finished and memory can be displayed. It needs to be an **Onchip-Breakpoint** since Linux assigned no memory for the process yet.



address	types	impl	condition	
R:0000:C00B71A4	Program	SOFT	task.proc.spaceid("hello")!=0xffffffff	set_binfmt

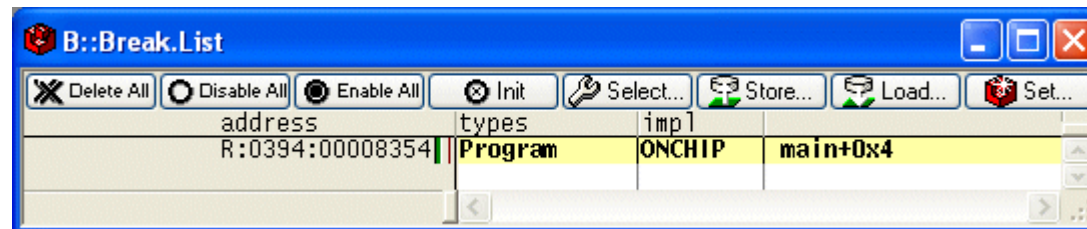
Debugging Linux-Processes

- Function “main+0x4” was reached.



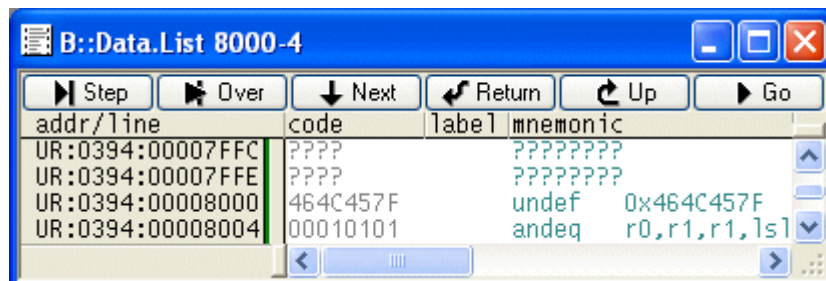
The screenshot shows the B::Data.List window with the following content:

addr/line	code	label	mnemonic	comment
	#include <stdio.h>			
	int main() {			
UR:0394:00008350	E1A0C00D	main:	mov r12,r13	
UR:0394:00008354	E92DD800		stmdb r13!,{r11-r12,r14,pc}	
UR:0394:00008358	E24CB004		sub r11,r12,#0x4	
	printf("Welcome to Run Mode Debugging!\n");			

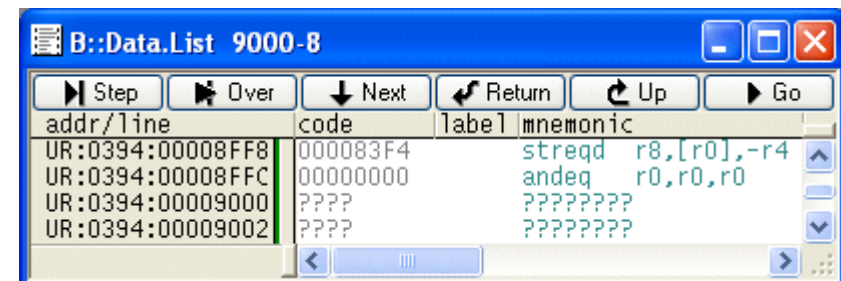


address	types	impl	
R:0394:00008354	Program	ONCHIP	main+0x4

- Only a 4 KB page is loaded for the process.



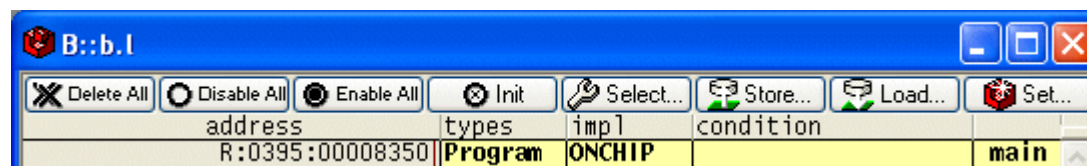
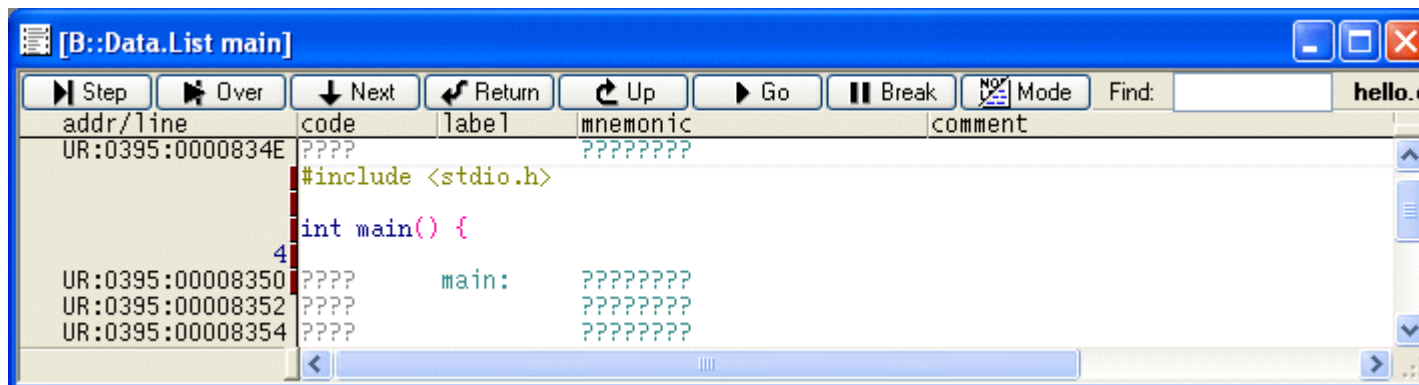
addr/line	code	label	mnemonic
UR:0394:00007FFC	????		????????
UR:0394:00007FFE	????		????????
UR:0394:00008000	464C457F	undef	0x464C457F
UR:0394:00008004	00010101	andeq	r0,r1,r1,ls1



addr/line	code	label	mnemonic
UR:0394:00008FF8	000083F4		streqd r8,[r0],-r4
UR:0394:00008FFC	00000000		andeq r0,r0,r0
UR:0394:00009000	????		????????
UR:0394:00009002	????		????????

Debugging Linux-Processes

- Background info: If you stop on main without offset 0x4 the debugger can load the symbols related to the MMU page-table entries for the process “hello”.
- But Linux doesn't load the 4 KB page before the CPU tries to fetch code on the address of “main”. This leads to a Page-Fault causing an exception handler to load the code page. But the Onchip-Breakpoint on “main” also stopped the exception handler. So no code page will be loaded due to the breakpoint.
- An additional single-step will allow loading the page and stop with PC on “main+0x4”. Same as the recommended breakpoint with offset did.



Debugging Linux-Processes

- Specific Process related information / displays. (spaceid = pid → decimal)

The image shows a workflow for debugging a Linux process using Lauterbach development tools. It consists of three main windows:

- B::TASK.DTask**: A table listing tasks. The task with magic C14F5A00 and command 'hello' is selected.
- B::Var.View %m %s (struct task_struct)*0x...**: A window showing the internal structure of the selected task, with 'comm = "hello"' highlighted.
- B::Task.DTask 0xC14F5A00**: A detailed view of the task, showing its state, parent, arguments, environment, and loaded libraries.

Red arrows indicate the workflow: from the 'Display task struct' menu item in the first window to the 'Var.View' window, and from the 'Kill task' menu item to the detailed task view window.

B::TASK.DTask

magic	command	state	uid	pid	spaceid	tty	flags
C14F49C0	pccardd	sleeping	0.	825.	0000	0	80200040
C14F56C0	sh	sleeping	0.	844.	034C	0	00400000
C14F5A00	hello	current	0.	916.	0394	0	00400000

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::Var.View %m %s (struct task_struct)*0x...

```
+ cred_guard_mutex = (count = (counter = 1), wait  
+ comm = "hello",  
+ link_count = 0,  
+ total_link_count = 1,  
+ sysvsem = (undo_list = 0x0),  
+ last_switch_count = 0,  
+ thread = (address = 0, trap_no = 0, error_code  
+ fs = 0xC15AB600,
```

B::Task.DTask 0xC14F5A00

magic	command	state	uid	pid	spaceid	tty	flags
C14F5A00	hello	current	0.	916.	0394	0	00400000

gid	vm size	tth	tty name	path
0.	0000015C	C1CD0000	-	/bin/hello

flags

parent youngest child younger sibling older sibling

sh

arguments

environment

open files

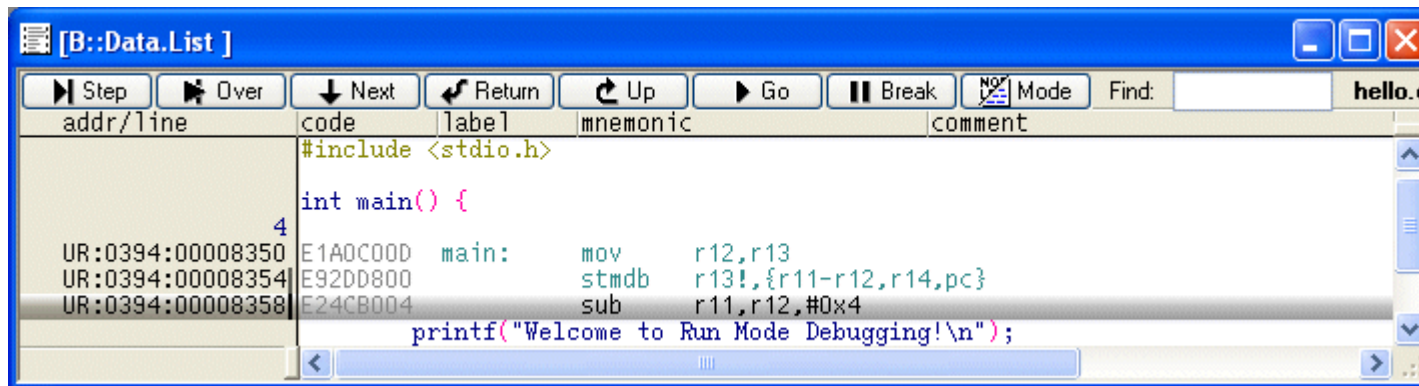
console

code addr/size	data addr/size	stack start
00008000 / 00000424	00010424 / 00000118	BEDCFEC0

code file	start address	flags
hello	00008000	EX RD
hello	00010000	WR EX RD
ld-2.5.so	40000000	EX RD
ld-2.5.so	40023000	WR EX RD
libc-2.5.so	40025000	EX RD

Debugging Linux-Processes

- Switching the context between Linux-Processes (before switching to swapper).

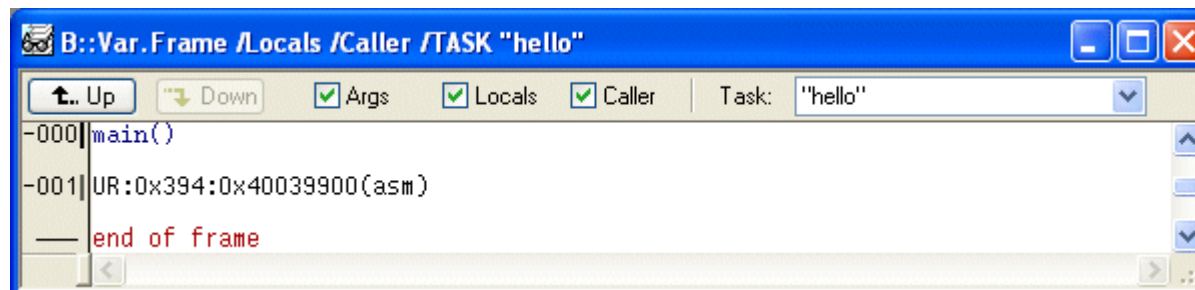


The [B::Data.List] window displays the assembly code for a program named 'hello.c'. The code is as follows:

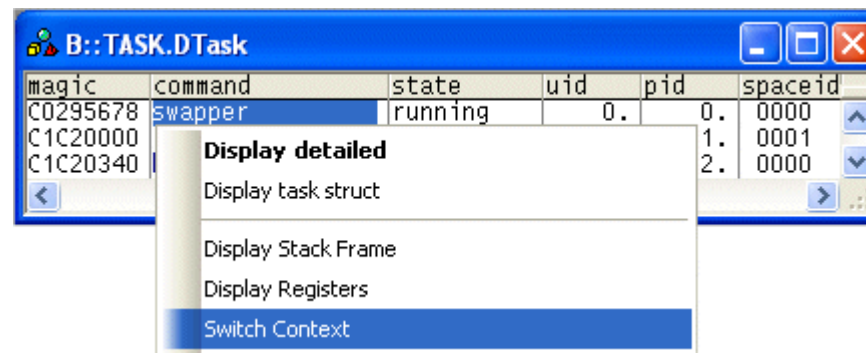
```
#include <stdio.h>

int main() {
    E1A0C00D main:  mov     r12,r13
    E92DD800      stmdb   r13!,{r11-r12,r14,pc}
    E24CB004      sub     r11,r12,#0x4
    printf("Welcome to Run Mode Debugging!\n");
}
```

The assembly code is shown in a table with columns: addr/line, code, label, mnemonic, and comment. The address range shown is from UR:0394:00008350 to UR:0394:00008358.



The B::Var.Frame /Locals /Caller /TASK "hello" window shows the stack frame for the main() function. The frame is displayed in a table with columns: addr/line, code, label, mnemonic, and comment. The frame is shown in a table with columns: addr/line, code, label, mnemonic, and comment. The frame is shown in a table with columns: addr/line, code, label, mnemonic, and comment.



The B::TASK.DTask window shows a list of tasks. The tasks are listed in a table with columns: magic, command, state, uid, pid, and spaceid. The tasks are shown in a table with columns: magic, command, state, uid, pid, and spaceid.

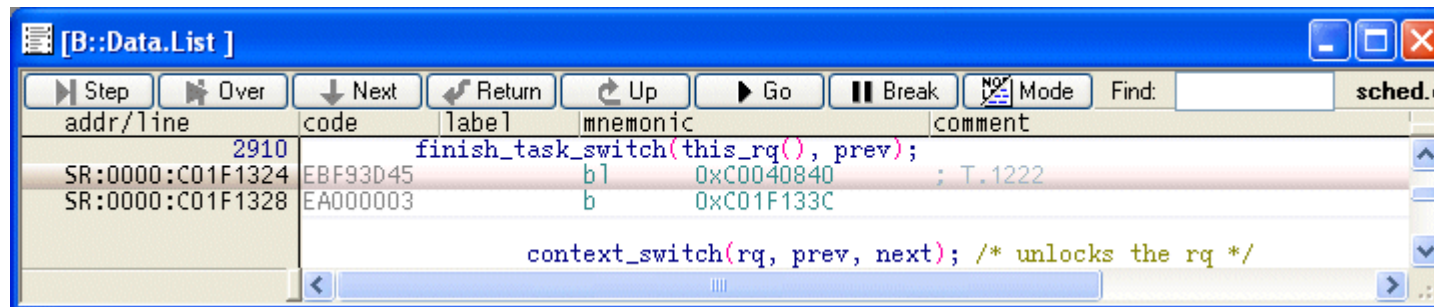
magic	command	state	uid	pid	spaceid
C0295678	swapper	running	0.	0.	0000
C1C20000				1.	0001
C1C20340				2.	0000

A context switch menu is displayed over the table, with the following options:

- Display detailed
- Display task struct
- Display Stack Frame
- Display Registers
- Switch Context

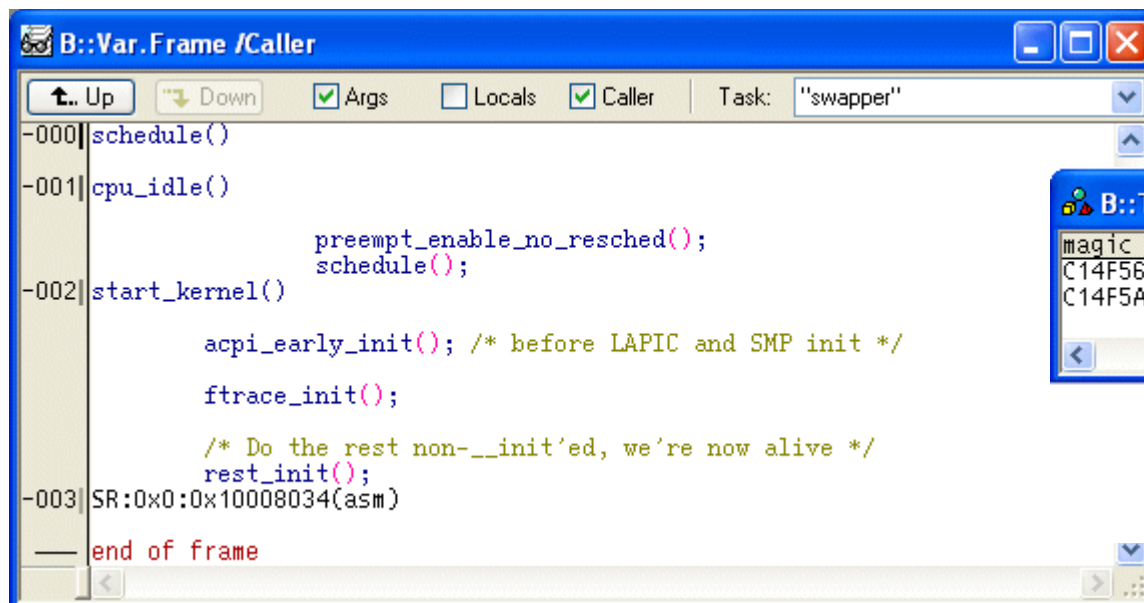
Debugging Linux-Processes

- Switching the context between Linux-Processes (option: switch-back to hello).
- Statusbar and TASK.List window can also be used to switch process.



[B::Data.List]

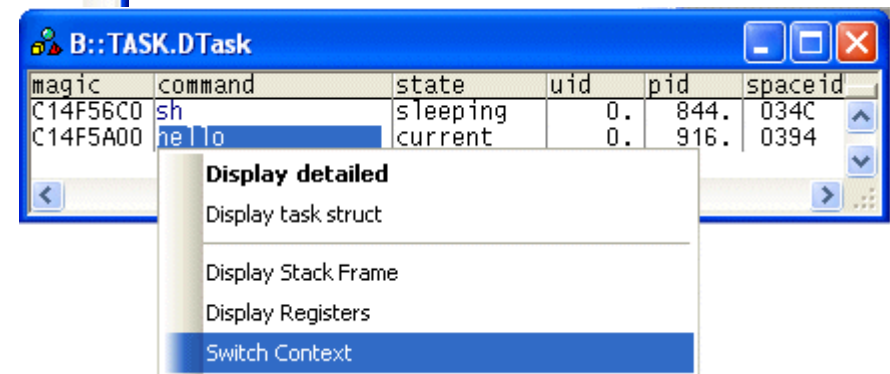
addr/line	code	label	mnemonic	comment
2910				finish_task_switch(this_rq(), prev);
SR:0000:C01F1324	EBF93D45	b1	0xC0040840	; T.1222
SR:0000:C01F1328	EA000003	b	0xC01F133C	
				context_switch(rq, prev, next); /* unlocks the rq */



B::Var.Frame /Caller

Task: "swapper"

```
-000| schedule()  
-001| cpu_idle()  
      preempt_enable_no_resched();  
      schedule();  
-002| start_kernel()  
      acpi_early_init(); /* before LAPIC and SMP init */  
      ftrace_init();  
      /* Do the rest non-__init'ed, we're now alive */  
      rest_init();  
-003| SR:0x0:0x10008034(asm)  
      end of frame
```



B::TASK.DTask

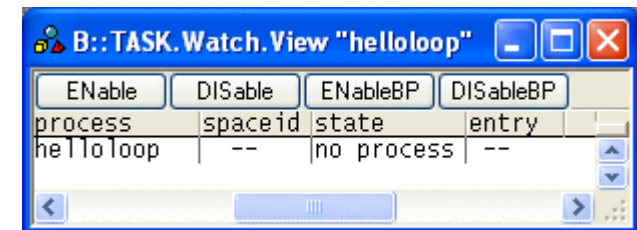
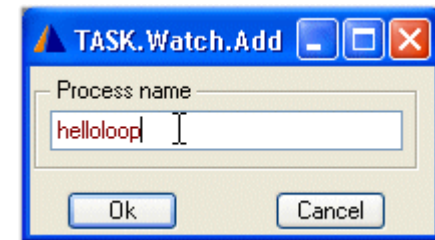
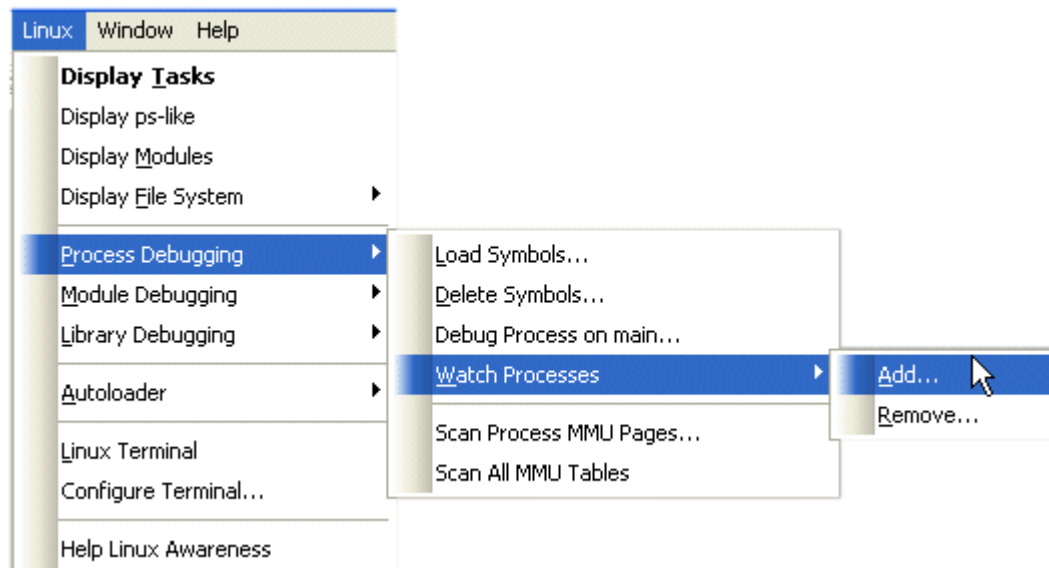
magic	command	state	uid	pid	spaceid
C14F56C0	sh	sleeping	0.	844.	034C
C14F5A00	hello	current	0.	916.	0394

Display detailed

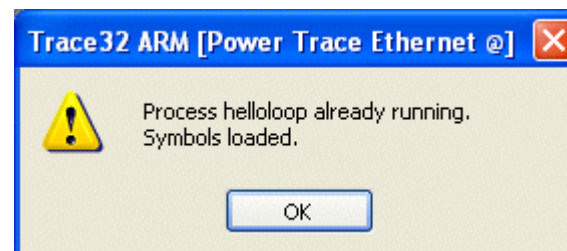
- Display task struct
- Display Stack Frame
- Display Registers
- Switch Context

Debugging Linux-Processes

- The “Watch” feature can check in parallel for more than one specified process.

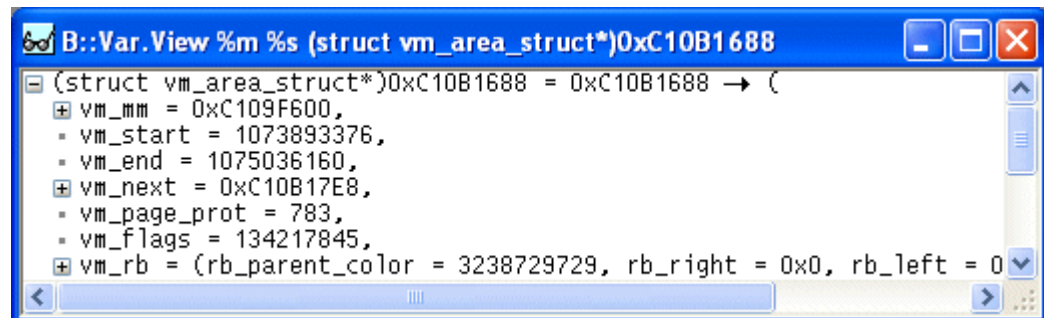
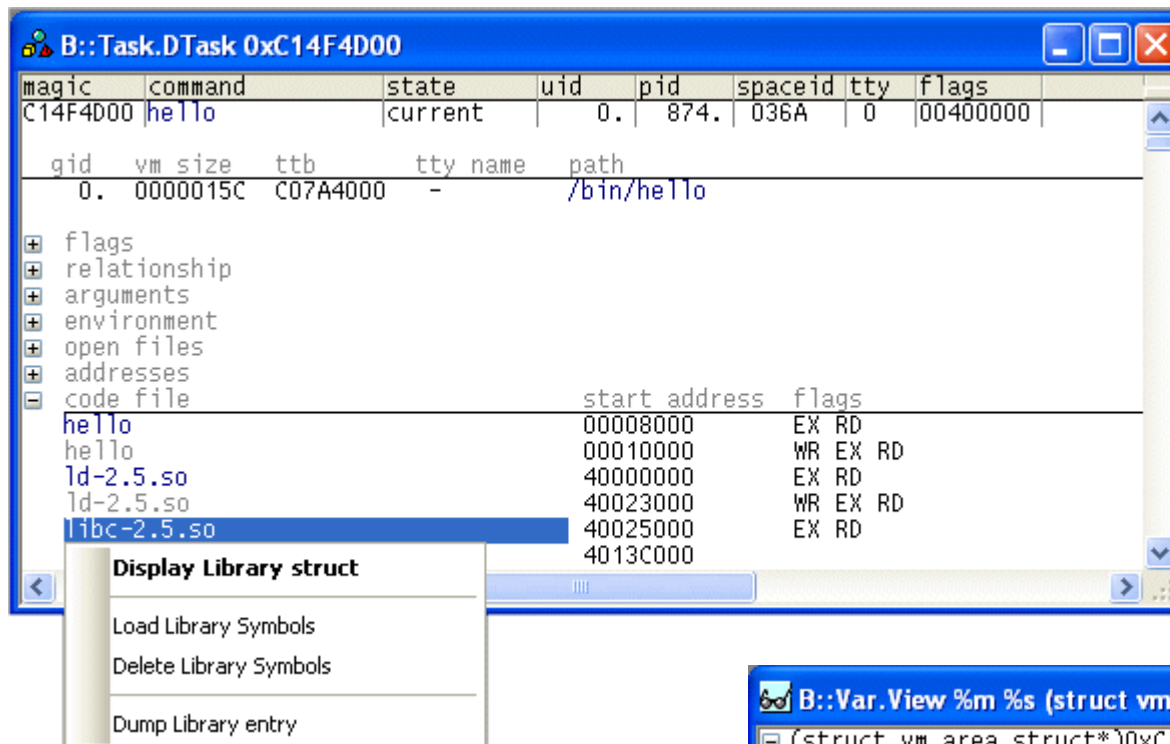


- Trying to use the “Debug Process on main” a second time on an already running process will fail, so you get a warning:



Debugging Linux-Libraries

- Libraries are loaded and linked dynamically to processes. Thus, they run in the virtual address space of the process and have dynamic addresses.



Debugging Linux-Threads

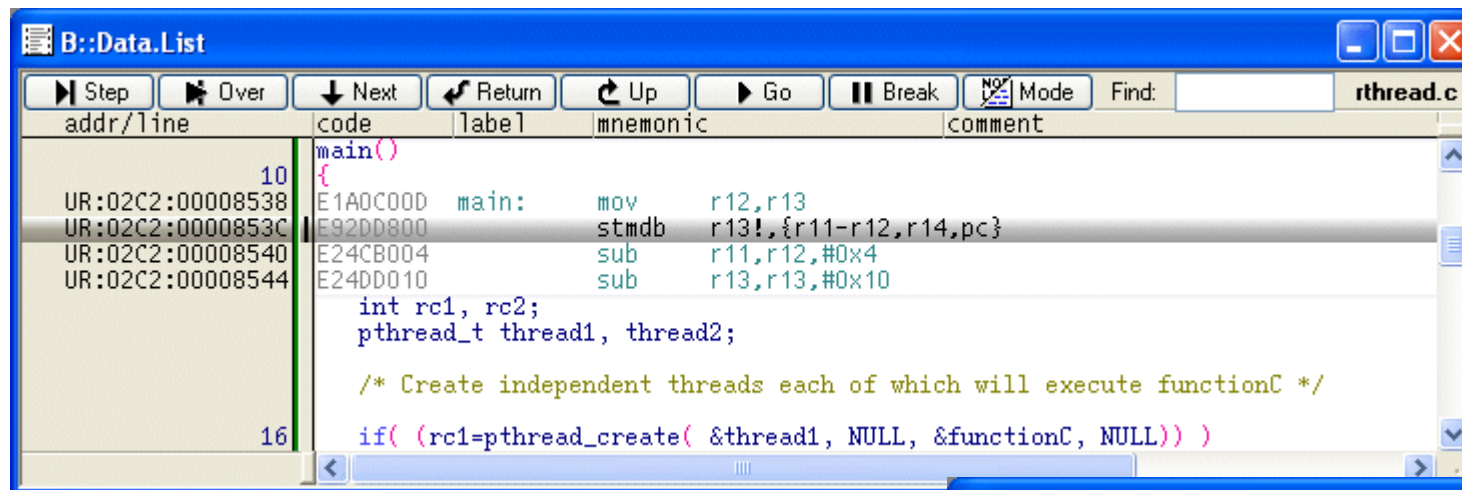
- Threads are Linux tasks that share the same virtual memory space and get the same spaceid by TRACE32 as the creating process. Thus all have the same symbols at once when loading the process symbols.



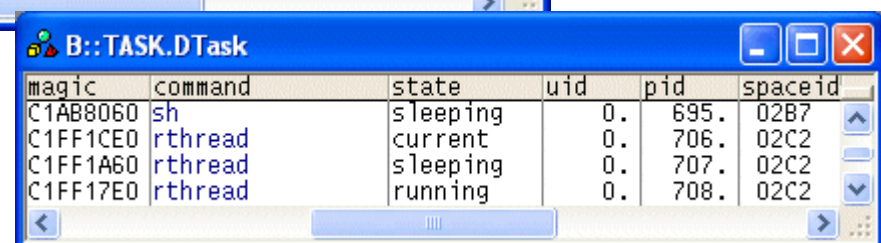
```
B::TERM
#
# ./rthread
```



magic	command	state	uid	pid	spaceid
C1AB8060	sh	sleeping	0.	695.	02B7
C1FF1CE0	rthread	current	0.	706.	02C2



```
B::Data.List
Step Over Next Return Up Go Break Mode Find: rthread.c
addr/line code label mnemonic comment
10 main()
UR:02C2:00008538 E1A0C00D main: mov r12,r13
UR:02C2:0000853C E92DD800 stmdb r13!,{r11-r12,r14,pc}
UR:02C2:00008540 E24CB004 sub r11,r12,#0x4
UR:02C2:00008544 E24DD010 sub r13,r13,#0x10
int rc1, rc2;
pthread_t thread1, thread2;
/* Create independent threads each of which will execute functionC */
16 if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
```



magic	command	state	uid	pid	spaceid
C1AB8060	sh	sleeping	0.	695.	02B7
C1FF1CE0	rthread	current	0.	706.	02C2
C1FF1A60	rthread	sleeping	0.	707.	02C2
C1FF17E0	rthread	running	0.	708.	02C2

LINUX & Android Debugging

Frank,Xing

- ◆ Platform and Debug Overview

- Linux Debugging

- ◆ Configuration Linux-Aware Debugging
- ◆ Stop-Mode-Debugging

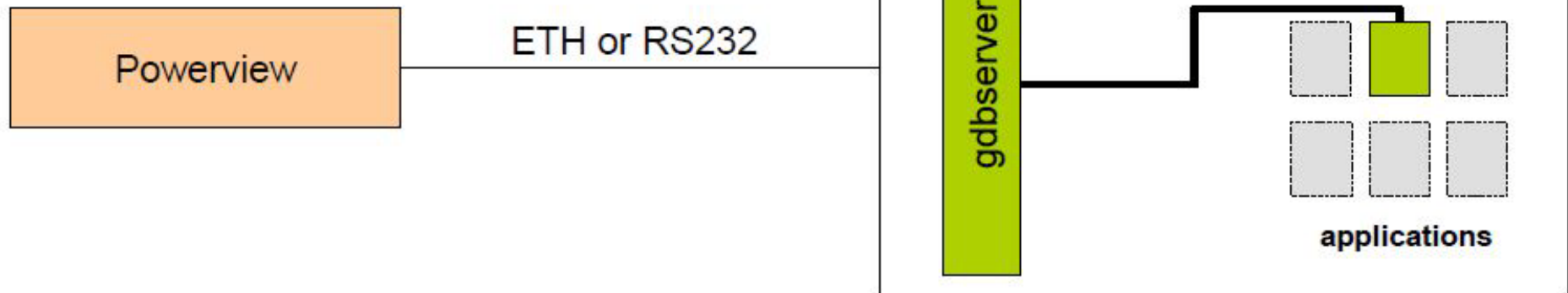
- Stop & Run-Mode-Debugging

- ◆ VM-Dalvik Debugging

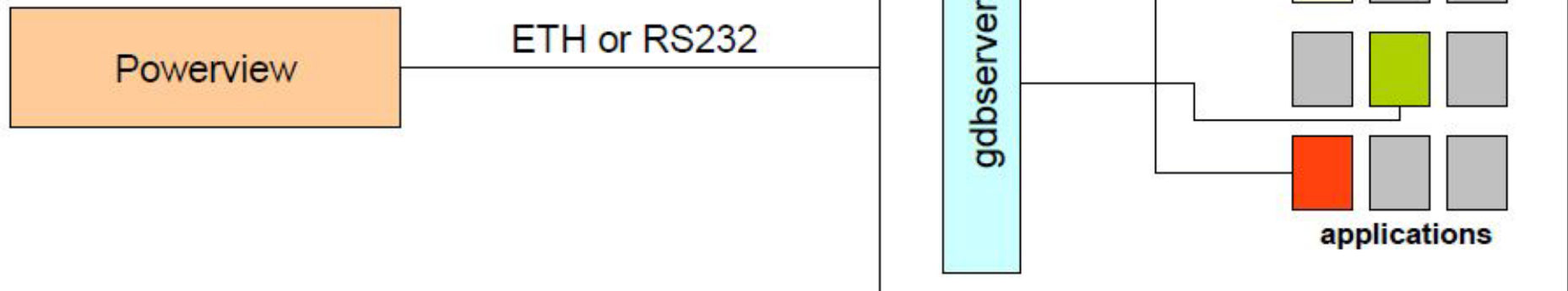
LINUX Debugging

- **Run-Mode-Debugging**
- **Stop & Run-Mode Debugging**

gdbserver <ip>:<port> <process>
gdbserver <ip>:<port> --attach <pid>



gdbserver -multi <ip>:<port>



Example for Trace32 Setup Script

```
SYStem.CPU CortexA8
TERm.METHOD COM COM1 115200. 8 NONE 1STOP NONE
TERm
TERm.OUT "gdbserver -multi :2345" 10.
SYStem.PORT 10.1.99.15:2345
SYStem.GDBLIBPATH C:\Linux\RunMode\targetlibs
SYStem.Option gdbEXTENDED ON
SYStem.Option gdbNONSTOP ON
SYStem.Option MMUSPACES ON
WAIT 1.s ; to be sure that the gdbserver has been started
SYStem.Mode Attach

TASK.List
TASK.RUN /bin/sieve

ENDO
```

LINUX Debugging

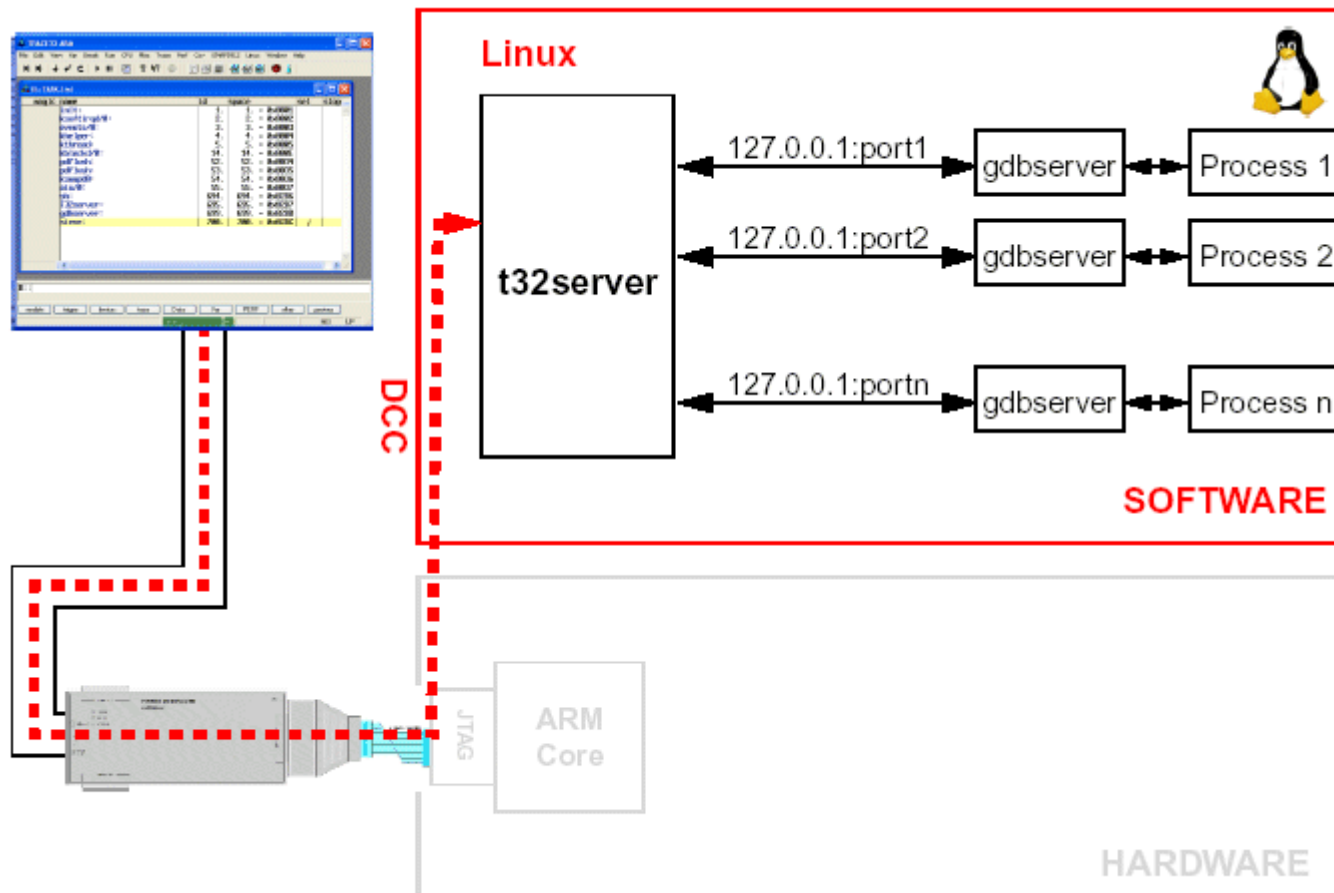
- **Run-Mode-Debugging**
 - **Stop & Run-Mode Debugging**

Run & Stop Mode Debugging on Linux-Processes

- Opposite to “**Stop Mode Debugging**” on all earlier pages, now “**Run Mode Debugging**” will be explained. The main difference is that you can stop a process but the CPU executes still all other processes.
- To debug not only a single process via a standard gdbserver as agent you can use the “t32server” which can serve multiple gdbservers.
- If TRACE32 is just used as a GDB-Front-End SW on the host you can choose between RS232, TCP/IP connection for Run Mode Debugging.
- TRACE32 can not only be used for Run Mode or Stop Mode Debugging. All of their advantages can be used together by “**Integrated Run & Stop Mode Debugging**”. The Stop Mode Debugging feature always requires a TRACE32 debug HW connected via JTAG to the target. For Integrated Run & Stop Mode Debugging the CPUs DCC communication interface to the TRACE32 “t32server” via JTAG connection is required.

Run & Stop Mode Debugging on Linux-Processes

- For Run Mode Debugging the RS232 or ethernet interface is sufficient.
- For Integrated Run & Stop Mode Debugging the DCC connection is used. The t32server process is started on the target via a terminal window.

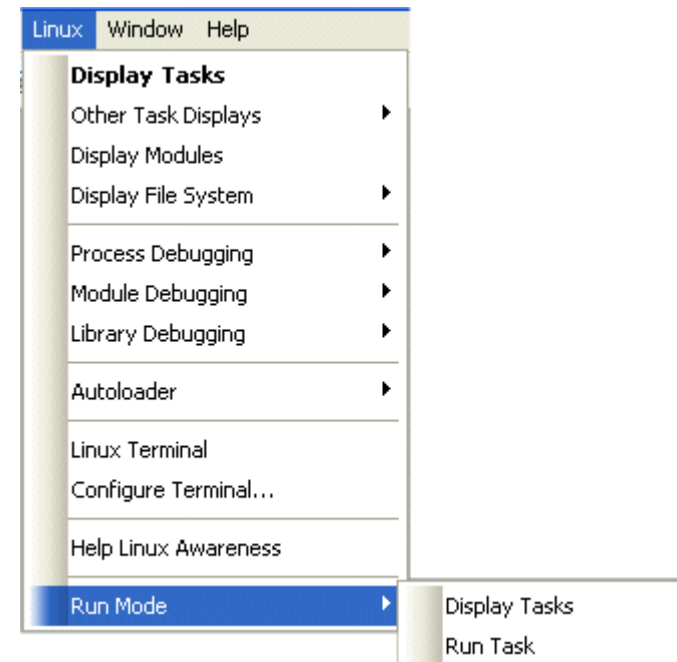
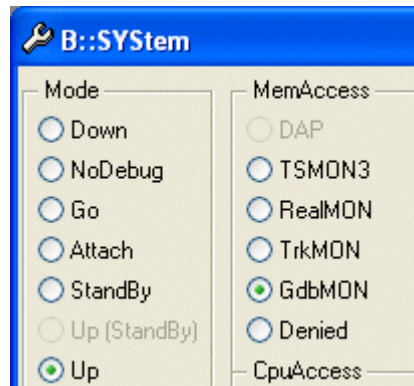


Run & Stop Mode Debugging on Linux-Processes

- Starting the t32server process on the target via the terminal window:
 - **./t32server** ; Communication via DCC
(You find details about DCC in your “ARM Technical Reference Manual”.
The gdbserver has to be in the Linux file system (FS) at “/bin”.)
 - **./t32server <host_ip>:<port_nr>** ; For GDB-Front-End via TCP/IP
(The IP address 127.0.0.1 has to be configured, because the t32server communicates with separate gdbservers via TCP/IP.)
- For details related to pure TRACE32 GDB-Front-End Debugging look at manual “monitor_gdb_arm.pdf”. (Almost a subset of the following.)
- The gdbserver has to be version 6.5 or higher. Check this by command
gdbserver --version ; no feedback → too old
(For version 6.6 you should use the gdbserver from the TRACE32 DVD.
The run length encoding is deactivated in this version.)
- The following pages explain “Integrated Run & Stop Mode Debugging” enabling you to switch from Stop Mode to Run Mode and back.

Run & Stop Mode Debugging on Linux-Processes

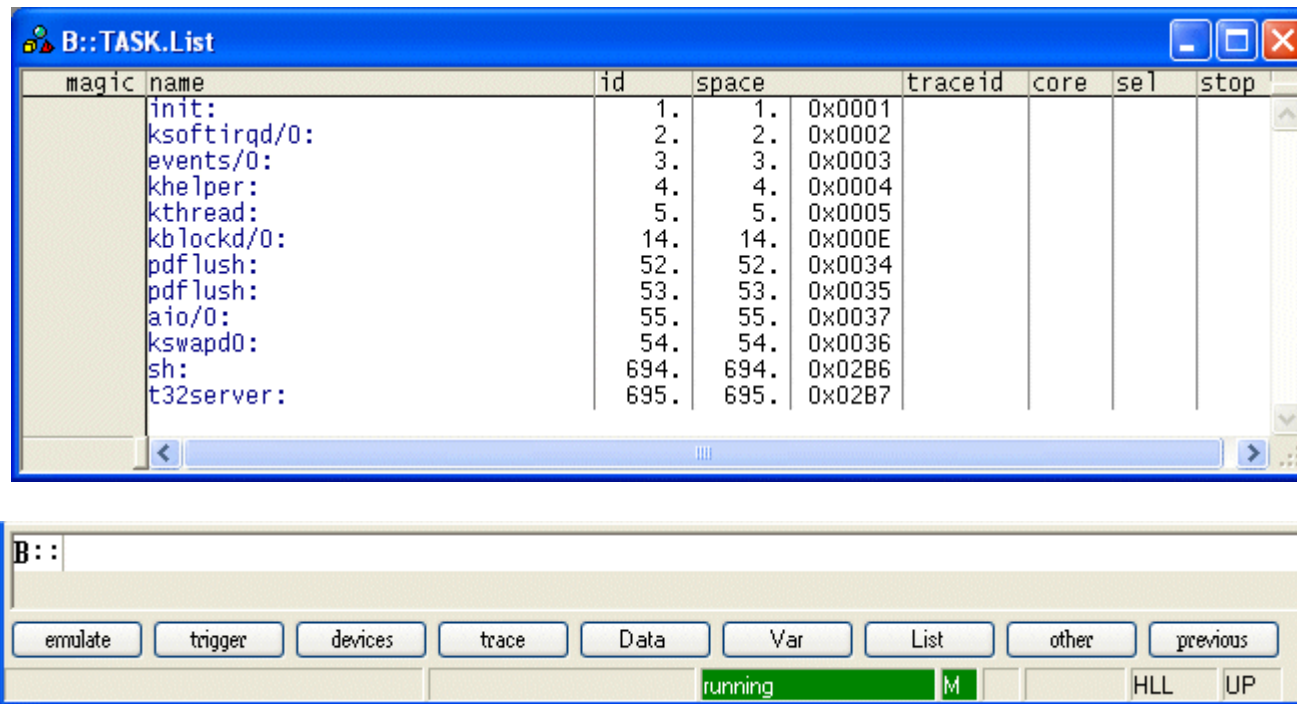
- After the setup for Stop Mode Debugging according to the previous chapters you can now switch to Run Mode Debugging.
 - **SYStem.MemAccess GdbMON** ; Use the DCC interface to t32server



- **Run Mode specific TRACE32 Menu**

Run & Stop Mode Debugging on Linux-Processes

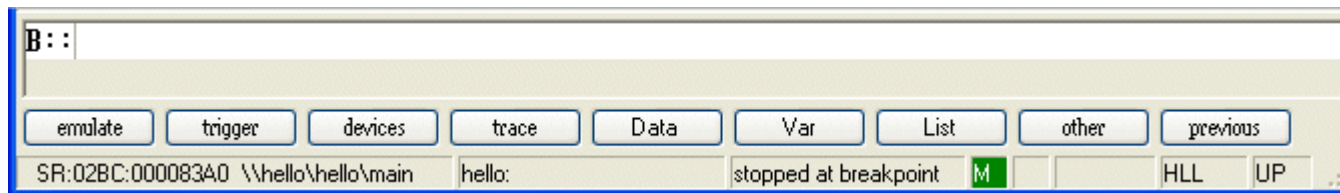
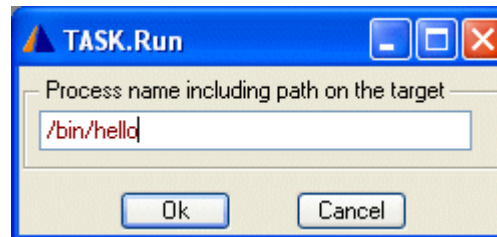
- **Go.MONitor** ; Switch to Run Mode Debugging when CPU was stopped



- TASK.List shows all started processes. No process is stopped currently by the debugger (see column “stop”) and the CPU is running in real-time (“running”, green background).
- The monitor is running for Run Mode Debugging (“M”, green background).

Run & Stop Mode Debugging on Linux-Processes

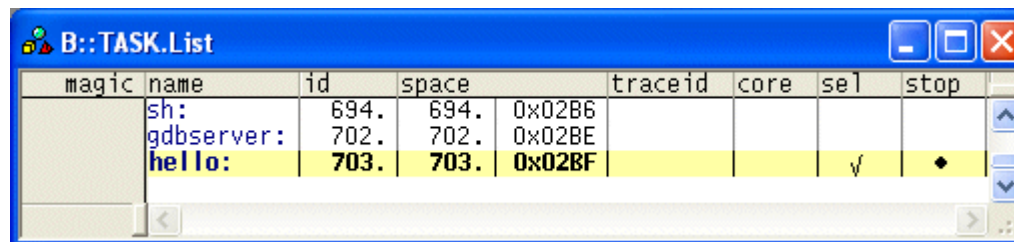
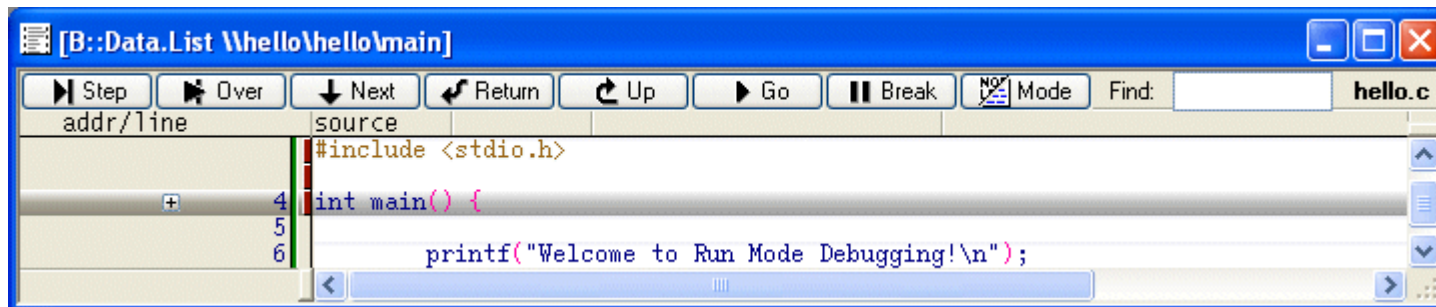
- **TASK.RUN /bin/hello** ; Start process “hello” via TRACE32
TRACE32



- The TRACE32 status line is showing:
 - The address where the process stopped currently
 - The name of the currently displayed process
 - The process status or cause for stopping
 - The Run Mode is active (“M”, green background: the monitor and all the processes are running if they are not marked as stopped (TASK.List))

Run & Stop Mode Debugging on Linux-Processes

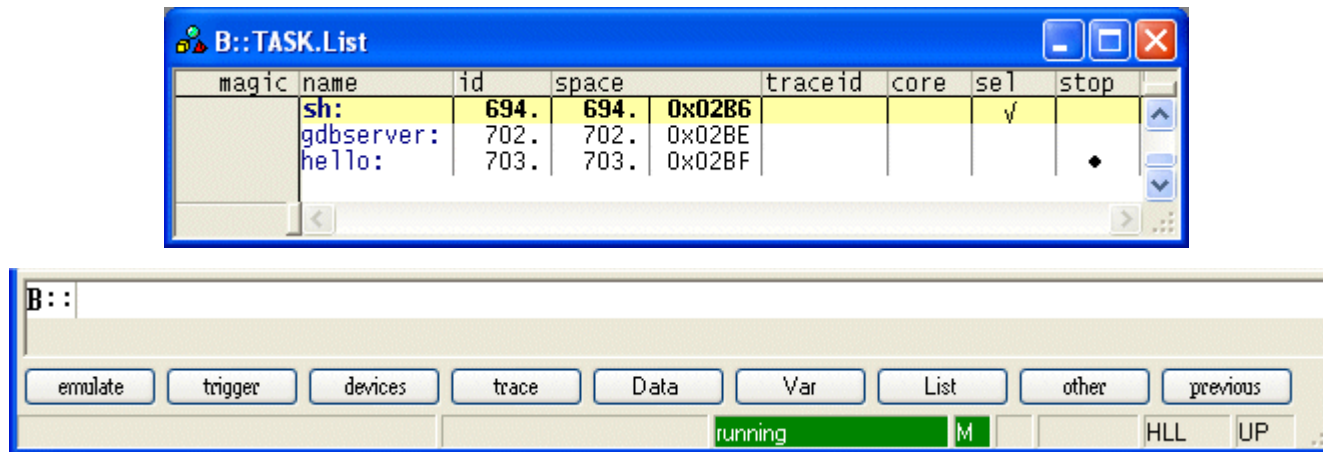
- Only process “hello” was stopped at function “main” and its symbols were loaded. The column “stop” shows that process “sh” and the gdbserver for process “hello” are still running.



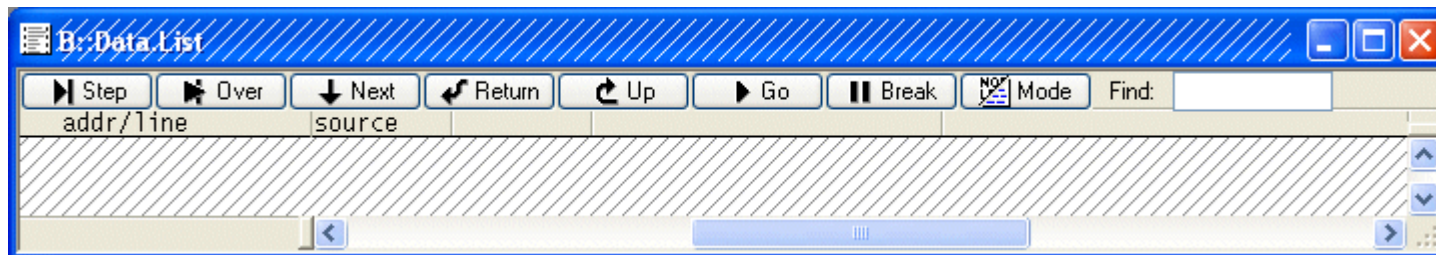
- The focus is selected to be on process “hello” (column “sel”).

Run & Stop Mode Debugging on Linux-Processes

- The check-sign in column “sel” set by a double-click decides which gdbserver is attached to control a process. So the focus has moved to a other process.

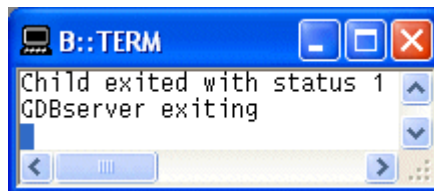


- But process “sh” is running so no info is available currently.

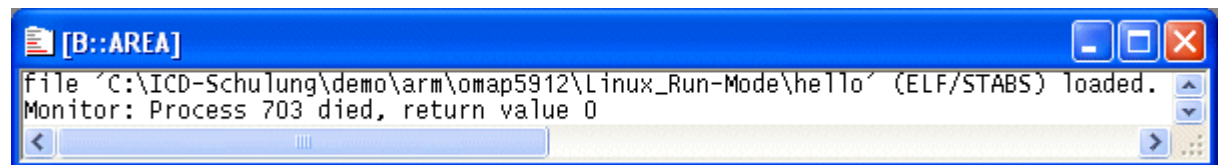


Run & Stop Mode Debugging on Linux-Processes

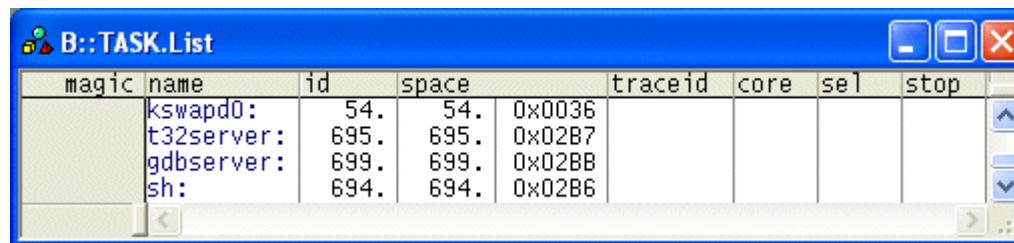
- If the execution of process “hello” is continued till its end you will get messages that the process died and the related gdbserver was exited.
- The t32server is responsible for more than one specific gdbserver. So the t32server is not terminated.



```
B::TERM
Child exited with status 1
GDBserver exiting
```



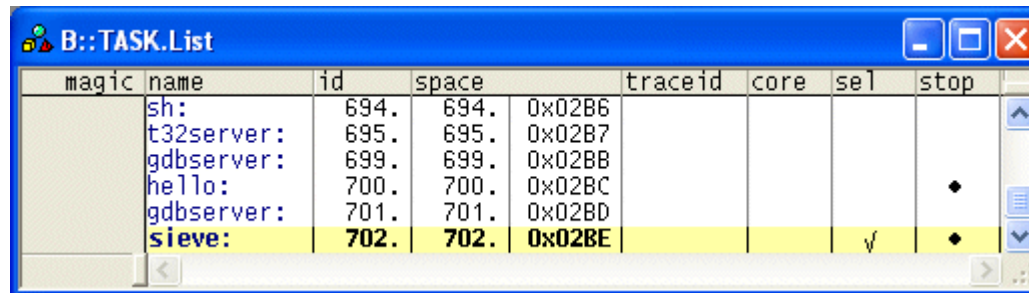
```
[B::AREA]
file 'C:\ICD-Schulung\demo\arm\omap5912\Linux_Run-Mode\hello' (ELF/STABS) loaded.
Monitor: Process 703 died, return value 0
```



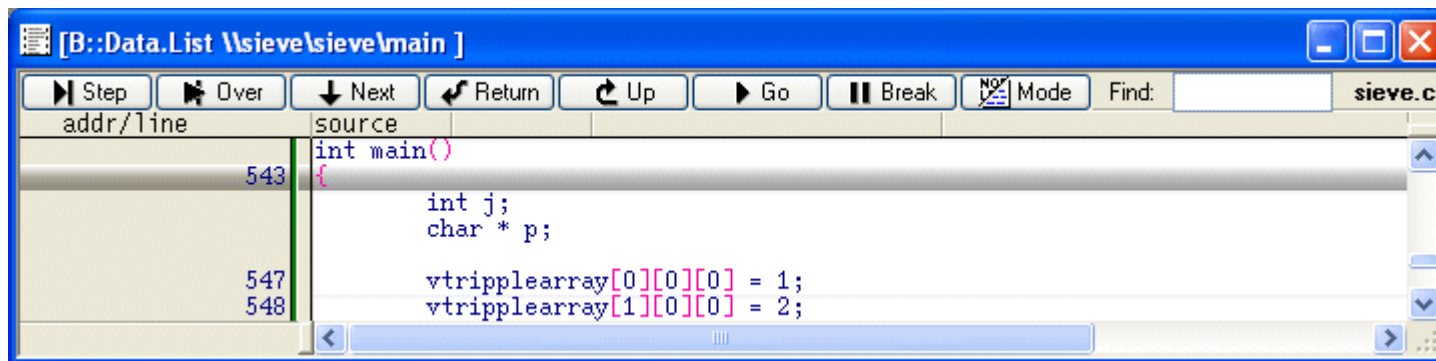
magic	name	id	space	traceid	core	sel	stop
	kswapd0:	54.	54.	0x0036			
	t32server:	695.	695.	0x02B7			
	gdbserver:	699.	699.	0x02B8			
	sh:	694.	694.	0x02B6			

Run & Stop Mode Debugging on Linux-Processes

- Run Mode Debugging of several processes at same time. If two processes are stopped you see infos for both.



magic	name	id	space	traceid	core	sel	stop
	sh:	694.	694.	0x02B6			
	t32server:	695.	695.	0x02B7			
	gdbserver:	699.	699.	0x02BB			
	hello:	700.	700.	0x02BC			♦
	gdbserver:	701.	701.	0x02BD			
	sieve:	702.	702.	0x02BE		✓	♦

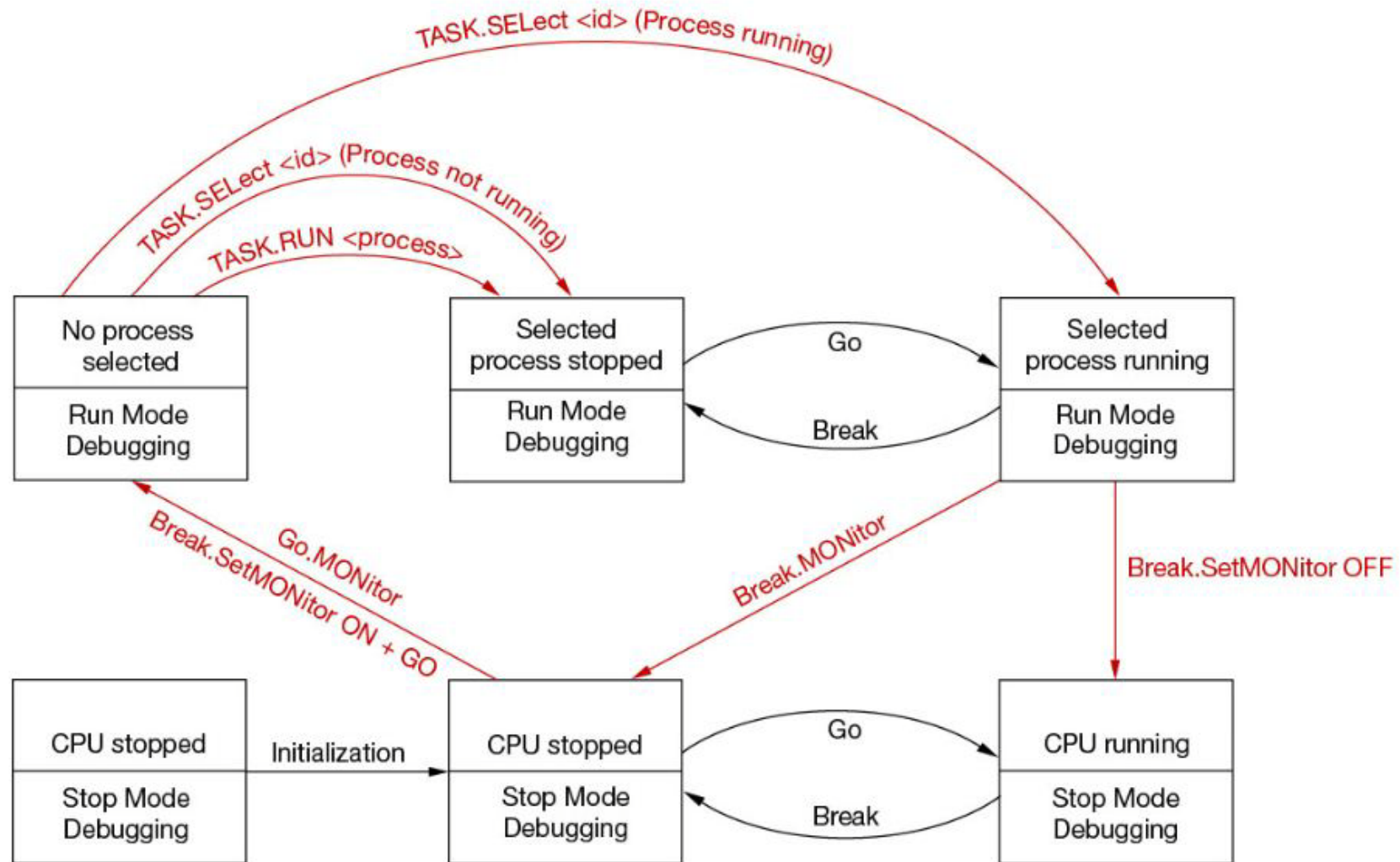


```
[B::Data.List \\sieve\\sieve\\main]
Step Over Next Return Up Go Break Mode Find: sieve.c
addr/line source
543 int main()
{
    int j;
    char * p;
547 vtripplearray[0][0][0] = 1;
548 vtripplearray[1][0][0] = 2;
```



```
[B::Data.List \\hello\\hello\\main]
Step Over Next Return Up Go Break Mode Find:
addr/line source
4 #include <stdio.h>
5 int main() {
6     printf("Welcome to Run Mode Debugging!\\n");
```


Run & Stop Mode Switching



LINUX & Android Debugging

Frank,Xing

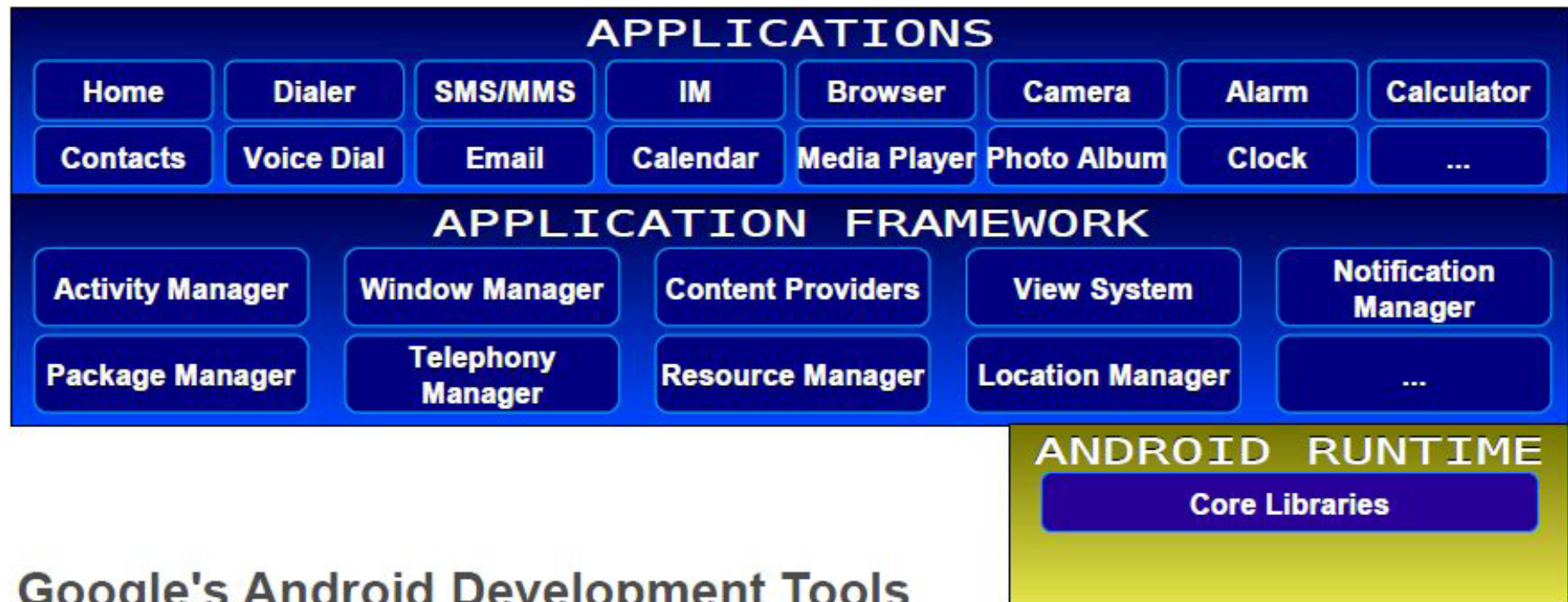
◆ Platform and Debug Overview

➤ Linux Debugging

- ◆ Configuration Linux-Aware Debugging
- ◆ Stop-Mode-Debugging
- ◆ Stop & Run-Mode-Debugging

➤ VM-Dalvik Debugging

Assisted VM Application Debugging



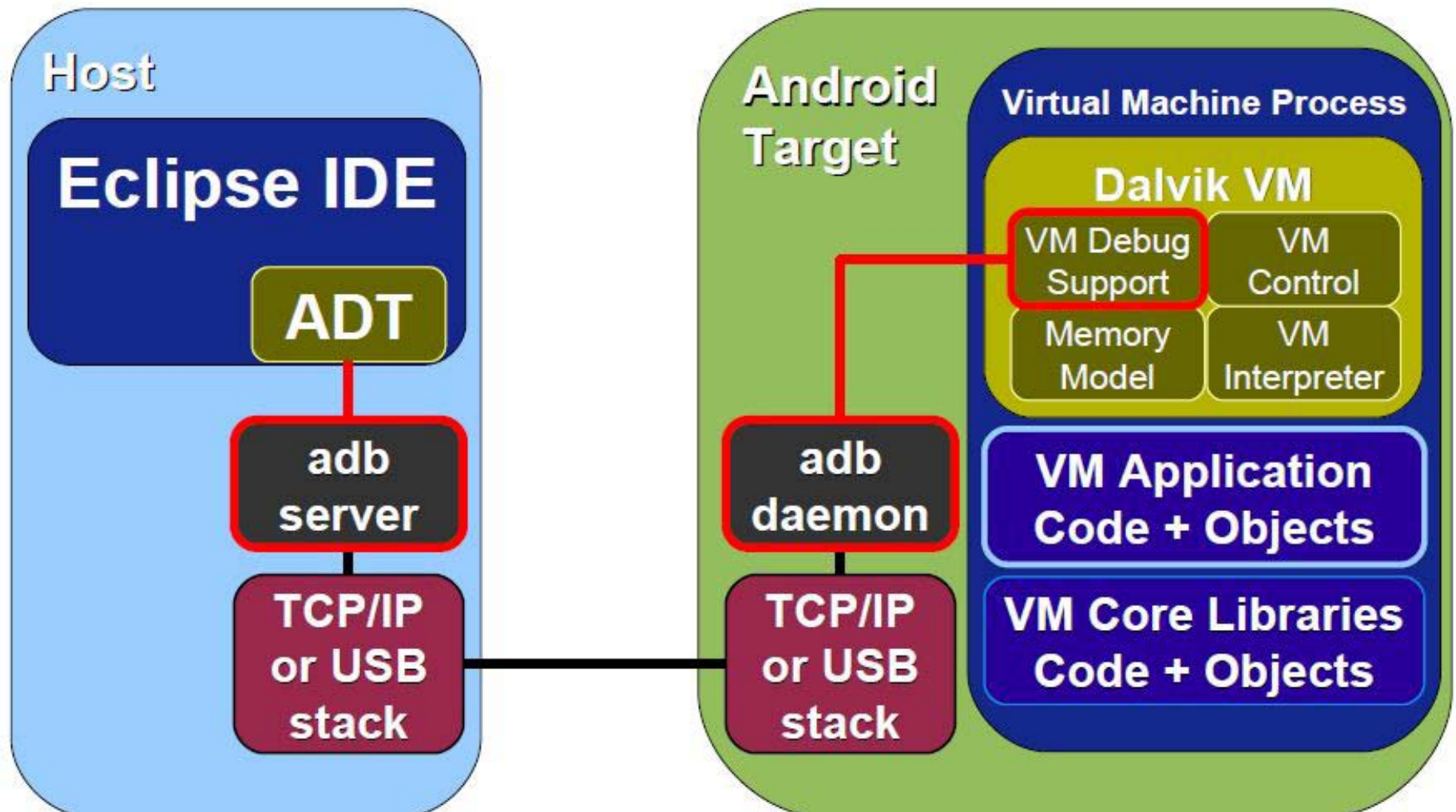
Google's Android Development Tools

can be used to debug Dalvik VM Application components:

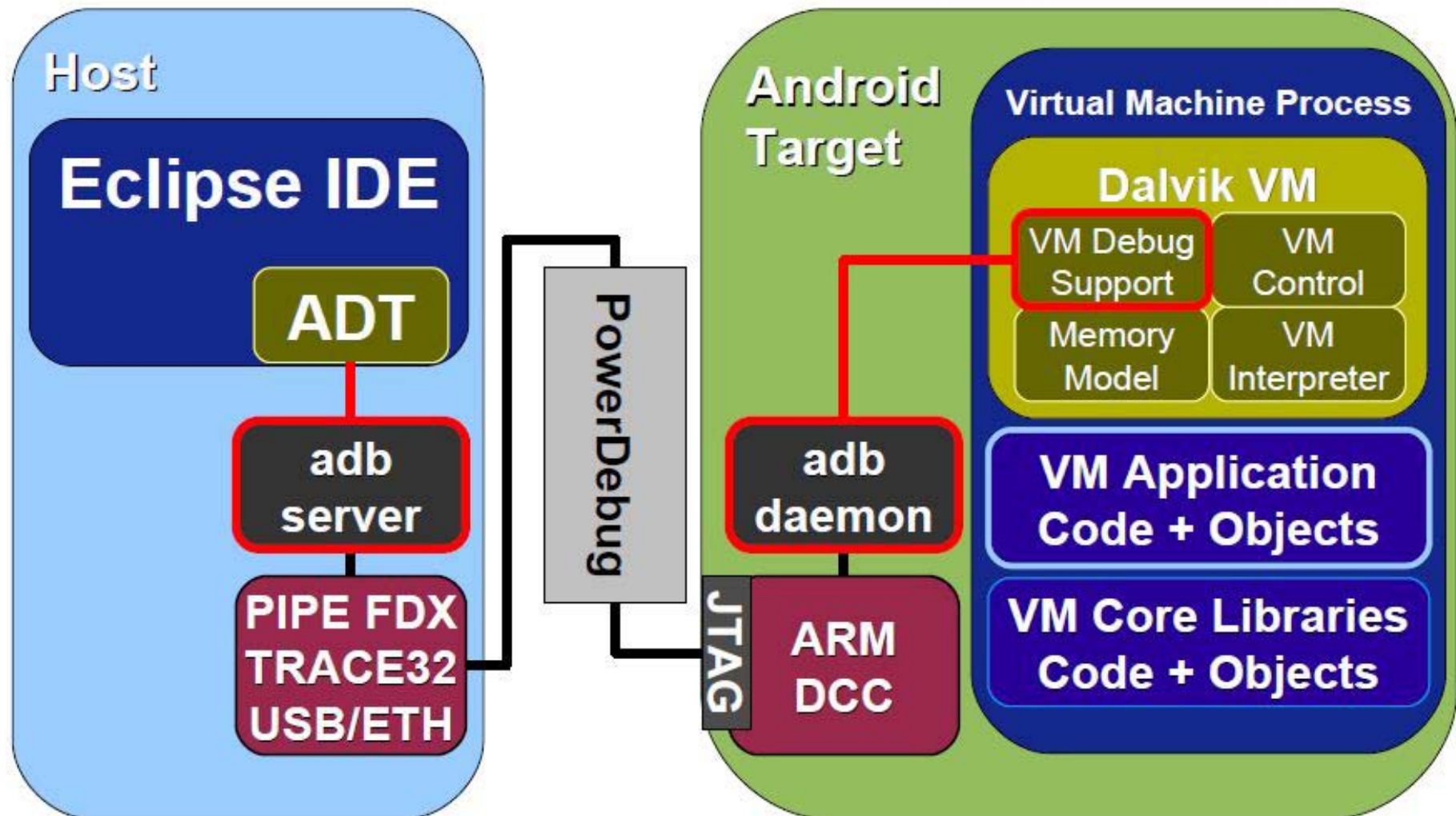
Applications, Application Framework, Core Libraries

...but the system must already be "live"!

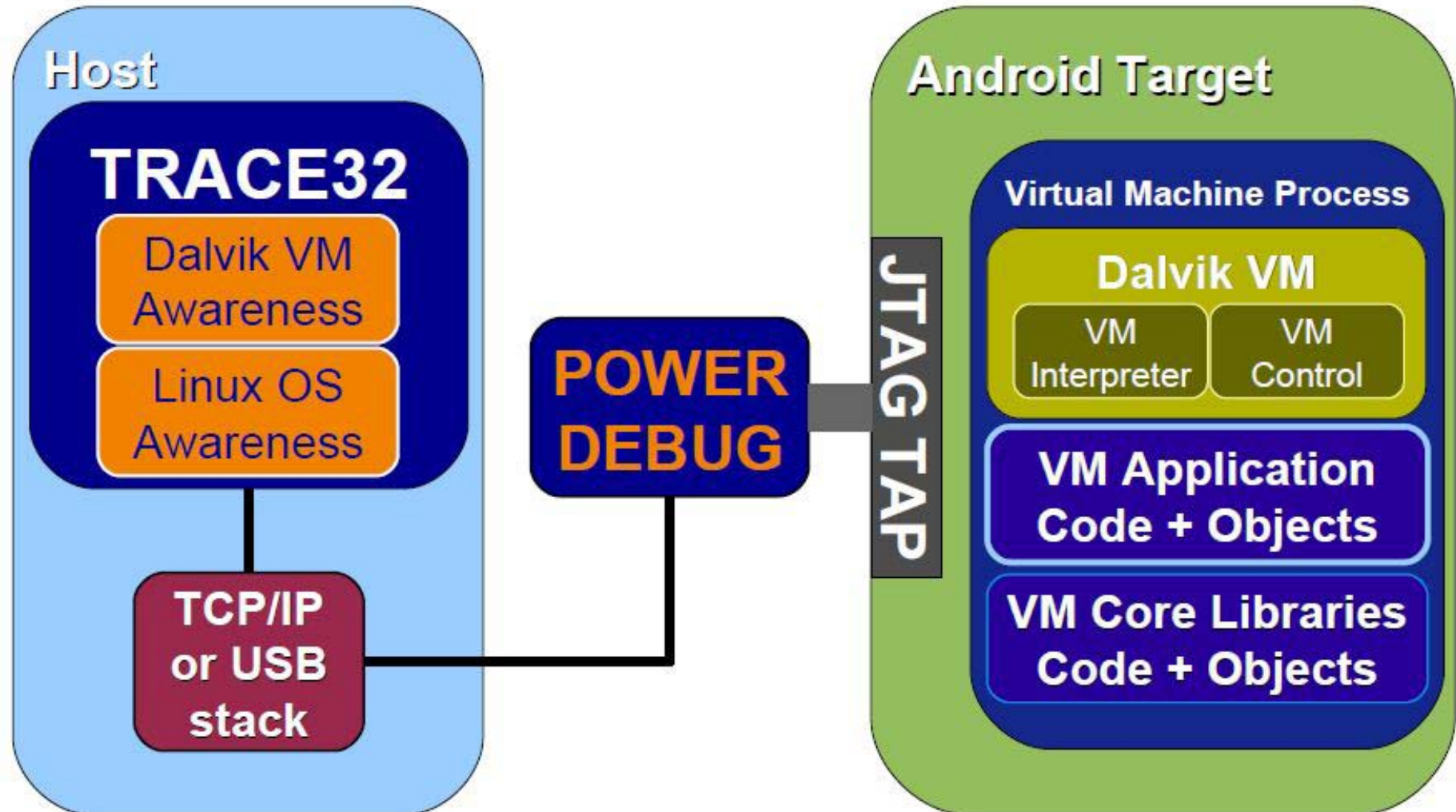
Assisted VM Application Debugging



Assisted VM Application Debugging – Trace32



VM Application Debugging – Stop Mode?



Trace32 VM Application Debugging – Status?

Current Research & Development to enable debugging in **stop-mode**, via **JTAG**, without VM assistance, for **Dalvik VM**

- Applications
- Application Framework
- Core Libraries

Available Today:

- Dalvik VM Extension load + debug
- Dalvik VM process list
- Dalvik VM process stack view

Trace32 VM Application Debugging – Status?

B::EXTension.VMList						
magic	pid	proc/task name	method name	class descriptor	class source	lvl
C7F7C960	0786	Binder_Thread_#				16
C2C09900	07A1	r.MountListener				20
C7E0A640	0765	com.android.inputmethod.latin	wait	Ljava/lang/Object;		11
C7E0A960	0766	HeapWorker				12
C7ED4640	076F	Signal_Catcher				11
C7ED4960	0770	JDWP				9
C7ED4C80	0771	Binder_Thread_#				16
C7ED5900	0775	Binder_Thread_#				16
C7E0AC80	0767	com.android.phone	wait	Ljava/lang/Object;		11
C7E0AFA0	0768	HeapWorker				12
C7E0B2C0	0769	Signal_Catcher				11
C7E0B5E0	076A	JDWP				9
C7ED55E0	0774	Binder_Thread_#				16
C7ED5C20	0776	Binder_Thread_#				16

B::EXT.VMView 0xC7E0AC80			
method name	class descriptor	class source	frameptr
wait	Ljava/lang/Object;		4104BDEC
next	Landroid/os/MessageQueue;	MessageQueue.java	4104BE10
loop	Landroid/os/Looper;	Looper.java	4104BE74
main	Landroid/app/ActivityThread;	ActivityThread.java	4104BEA4
(n/a)	(n/a)	(n/a)	4104BED0
invokeNative	Ljava/lang/reflect/Method;	Method.java	4104BEE4
invoke	Ljava/lang/reflect/Method;	Method.java	4104BF18
run	Lcom/android/internal/os/ZygoteInit\$Metho	ZygoteInit.java	4104BF58
main	Lcom/android/internal/os/ZygoteInit;	ZygoteInit.java	4104BF8C
(n/a)	(n/a)	(n/a)	4104BFC0
main	Ldalvik/system/NativeStart;	NativeStart.java	4104BFD4
(n/a)	(n/a)	(n/a)	4104BFEC

Trace32 VM Application Debugging – Status

Current development focus:

- Virtual Machine Source Code lookup

Current Platform and Solution:

- ARM
- Opening platform

(<http://www.lauterbach.com/vmandroid.html>)



Thank You!

Questions?