

TRACE32: the most complete tool for embedded linux debugging



- Maurizio Menegotto
Lauterbach Italy

A Linux system is composed of several software components very different from each other. Free or cheap debuggers are generally used for one of these components, but not for others, and require the user to wade through different debugging techniques not homogeneous.

The aim of the presentation is to illustrate how a professional system Lauterbach TRACE32™ enable debugging of each linux component, from uboot to the kernel, modules and dynamic libraries, from processes to threads: a total view of the system, with a single debugger, in the same debugging session.

Agenda



Maurizio Menegotto
Lauterbach Italy

Seminar and live demo

- Linux debugging: problems & solution
- Debugging all linux components
- Stop-mode & run-mode debugging
- Trace, performance, profiling
- TRACE32 PowerTools
- Q&A

Agenda

Seminar and live demo

- **Linux debugging: problems & solution**
- Debugging all linux components
- Stop-mode & run-mode debugging
- Trace, performance, profiling
- TRACE32 PowerTools
- Q&A



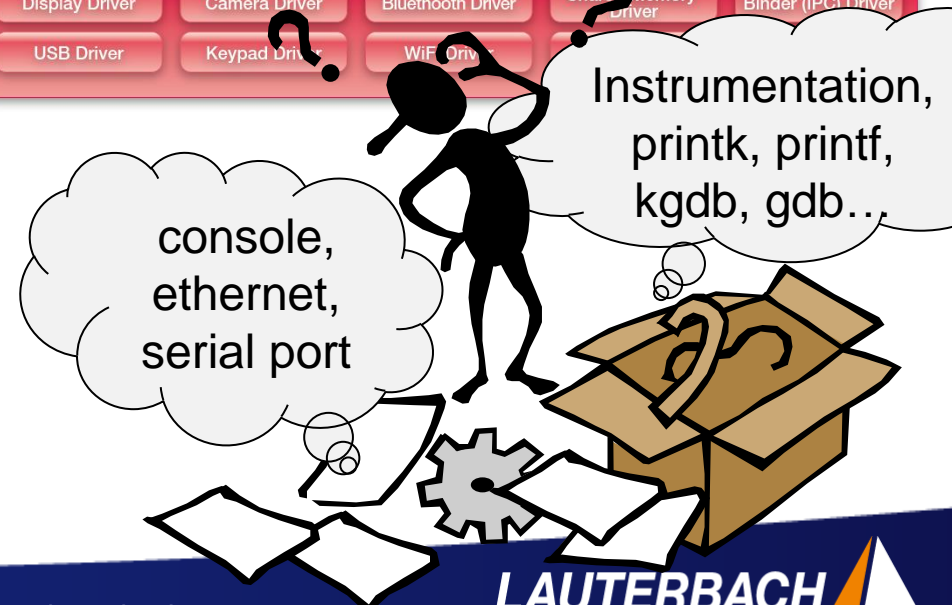
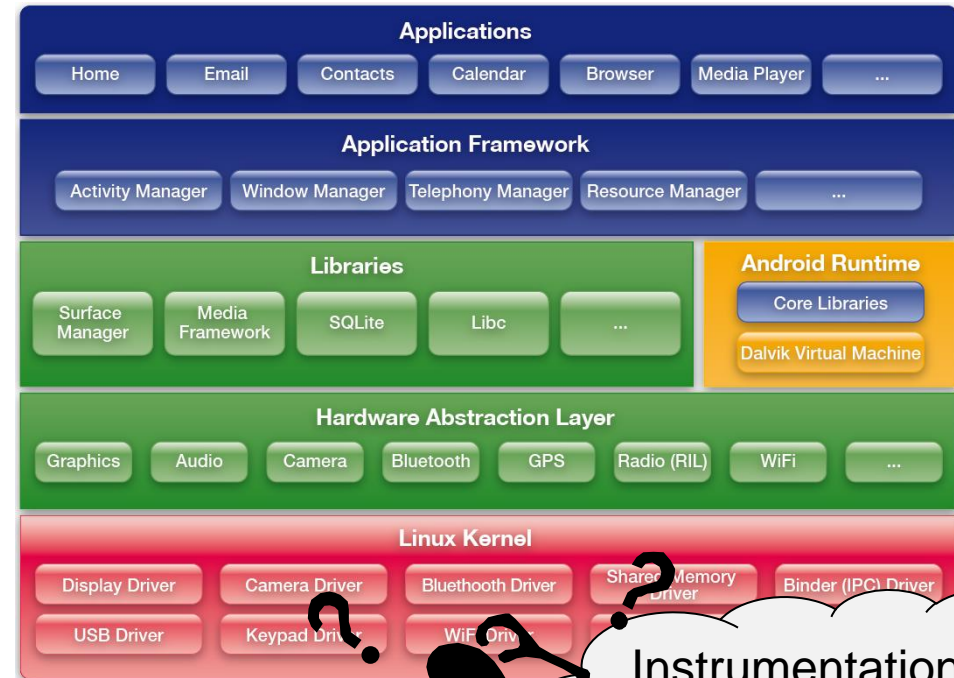
Linux debugging, the problems

In modern embedded systems, more and more frequently developers use operating systems, Linux is one of the most open source kernel used.

An embedded system based on Linux poses several problems from the point of view of debugging, as it consists of many different elements, and has advanced features that complicates the live of the debugger, such as on-demand paging, and dynamic MMU management.

The debugger free or cheap are generally used for one of these components, but not for others, and require the user to wade through different debugging techniques and frequent recompilation.

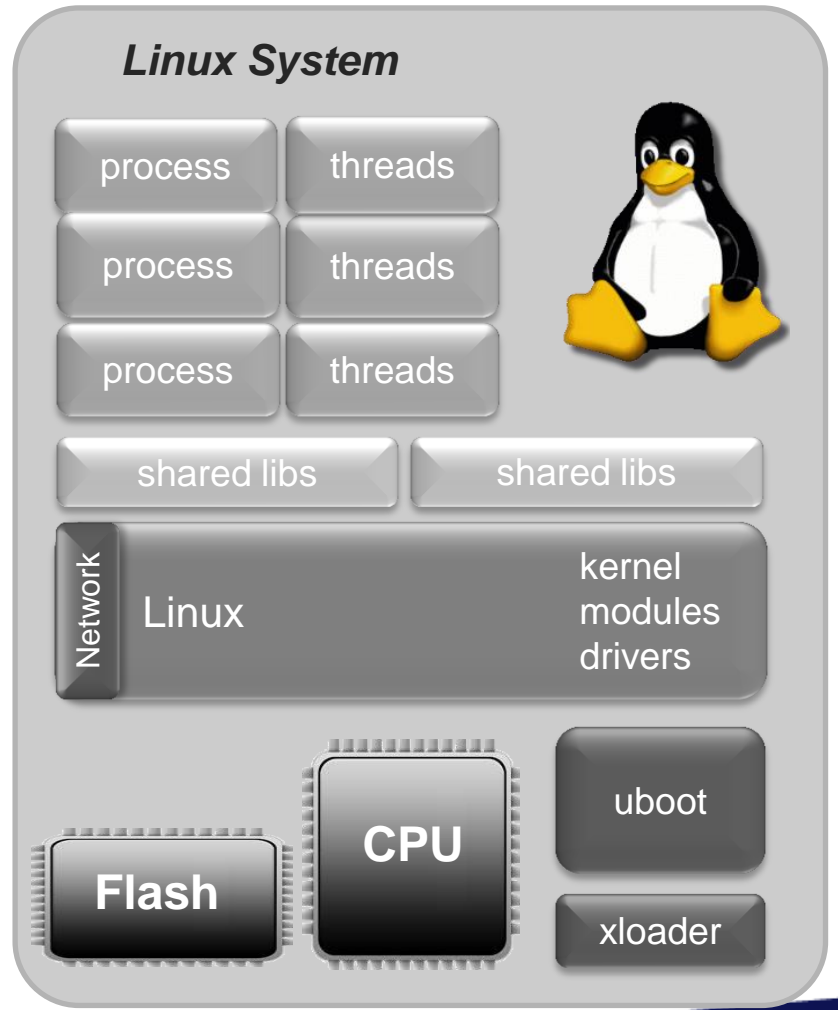
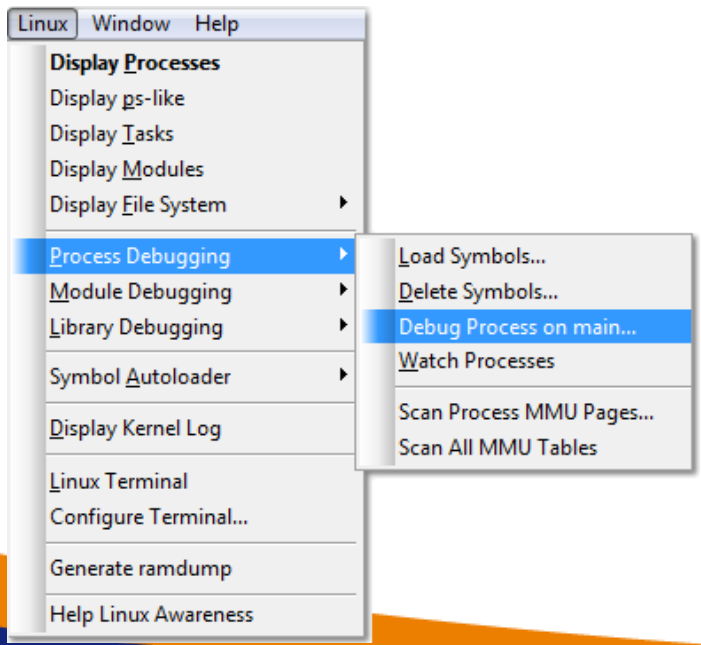
Linux system components



Linux debugging, a unique solution

A professional system Lauterbach TRACE32 enable debugging of each linux component, from uboot to the kernel, modules and dynamic libraries, from processes to threads.

TRACE32 PowerView gives an immediate and complete view of the entire system with a single debugger, in the same debugging session, both in stop-mode and run-mode.



Linux debugging, what you need:

To debug a Linux system with TRACE32 you need:

Your computer



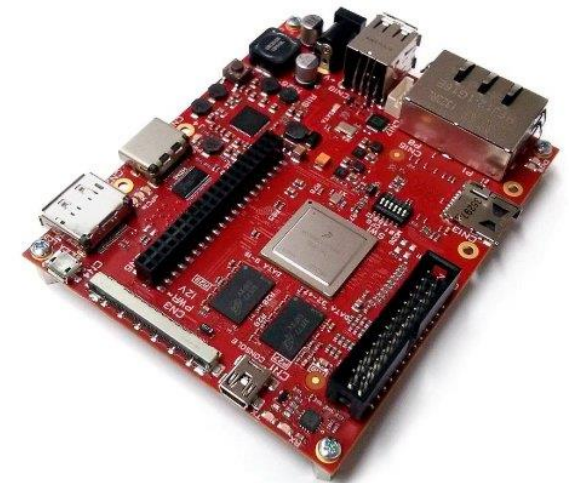
- PC Linux or Windows
- Your linux application
- TRACE32 PowerView SW

Your TRACE32



- A PowerDebug HW + JTAG debug cable for your chip.
- A Cortex™-A9 in this example

Your Target

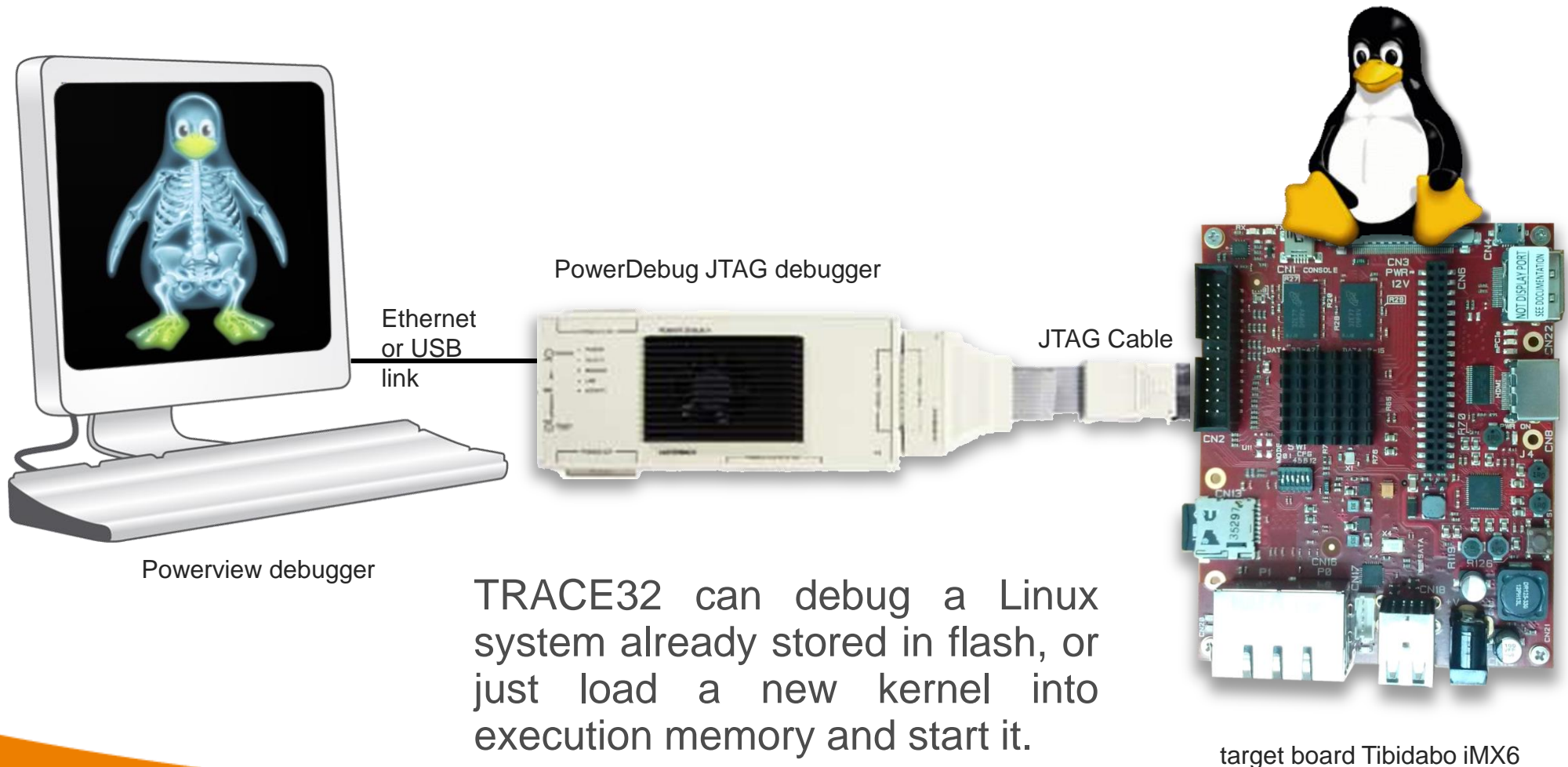


- Target with a JTAG port
- An Architech Tibidabo Board based on Freescale iMX6 Quad

➔ *Note: application and kernel must be compiled with debug symbols!*

Linux debugging, connection

The only physical connection to the target, required for debugging, is the JTAG port. TRACE32 has full target control since power-on reset.

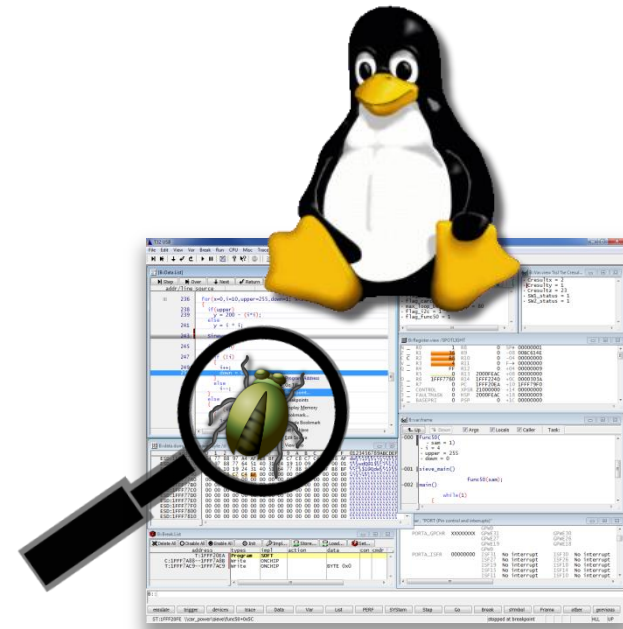


TRACE32 can debug a Linux system already stored in flash, or just load a new kernel into execution memory and start it.

Agenda

Seminar and live demo

- Linux debugging: problems & solution
- **Debugging all linux components**
- Stop-mode & run-mode debugging
- Trace, performance, profiling
- TRACE32 PowerTools
- Q&A



Debugging linux components: booting

Enabling the debugging session TRACE32 can take control of the cpu since the boot and show the program in memory, usually stopped at entry point

The image displays three windows from the TRACE32 debugger:

- [B::Data.ListASM]**: Shows assembly code with columns for address/line, code, label, mnemonic, and comment. The entry point `start_armboot` is highlighted.
- B::Data.List**: Shows the corresponding C source code for `start_armboot`, including comments about memory allocation and compiler optimization barriers.
- B::SYMBOL.LIST.TREE**: Shows a tree view of loaded symbols, including `\start`, `\qong`, `\arm1136/interrupts`, `\mx31/interrupts`, `\udivsi3`, and `\board`.

Blue arrows indicate the flow of information: one arrow points from the assembly window to the text "You can load symbols to debug the bootloader (Xloader, Uboot, ...)", and another arrow points from the symbol table window to the source code window.

You can load symbols to debug the bootloader (Xloader, Uboot, ...)

And proceed to source-level debugging of everything is running before Linux

Debugging linux components: kernel start

Typically a bootloader initializes the hardware and configure it to run the operating system.

The Linux kernel image (ulmage) is loaded into RAM by the bootloader (uboot) or even by the debugger itself.

By booting the kernel, you can continue debugging from the entry point of Linux:

Terminal emulator integrated in TRACE32

```

B::term
U-Boot 2013.10 (Mar 03 2014 - 19:44:09)

CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
Reset cause: WDOG
Board: Tibidabo
DRAM:  2 GiB
MMC:   FSL_SDHC: 0
SF: Detected N25Q1024 with page size 256 Bytes, era
No panel detected: default to HDMI
unsupported panel HDMI
In:    serial
Out:   serial
Err:   serial
Net:   Phy not found
PHY reset timed out
FEC [PRIME]
88E6123 Initialized on FEC
Hit any key to stop autoboot:  0
TIBIDABO U-Boot >
    
```

addr/line	code	label	mnemonic	comment
ZSR:10007FF0	7FBFEFFB		svclc	0xBFEFFB
ZSR:10007FF4	FEF7FDCD		cdp2	p13,0x0F,c15,c7,c13,0x6
ZSR:10007FF8	E7FBED77		undef	0xE7FBED77
ZSR:10007FFC	FFFF9D7F		undef	0xFFFF9D7F
ZSR:10008000	E1A00000		nop	
ZSR:10008004	E1A00000		nop	
ZSR:10008008	E1A00000		nop	
ZSR:1000800C	E1A00000		nop	
ZSR:10008010	E1A00000		nop	
ZSR:10008014	E1A00000		nop	
ZSR:10008018	E1A00000		nop	
ZSR:1000801C	E1A00000		nop	
ZSR:10008020	EA000002		b	0x10008030
ZSR:10008024	016F2818		clzeq	r2,r8

```

B::wr.term
Hit any key to stop autoboot:  0
TIBIDABO U-Boot >
TIBIDABO U-Boot > run boot_from_nor
SF: Detected N25Q1024 with page size 256 Bytes, erase size 4
SF: 4587520 bytes @ 0xd0000 Read: OK
## Booting kernel from Legacy Image at 12000000 ...
Image Name:   Linux-3.0.35-4.1.0+yocto+gbdde70
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3853092 Bytes = 3.7 MiB
Load Address: 10008000
Entry Point:  10008000
Verifying Checksum ... OK
Loading Kernel Image ... OK

Starting kernel ...
    
```



Debugging linux components: kernel symbols

If you boot Linux and stop the run with a BREAK, you can see the program running at address 0x8xxxxxxx

addr/line	code	label	mnemonic	comment
ZSR:80008624	E12FFF1E		bx	r14
ZSR:80008628	E12FFF1E		bx	r14
ZSR:8000862C	E92D40F0		push	{r4-r7,r14}
ZSR:80008630	E24DD014		sub	r13,r13,#0x14 ; r13,r13,#20
ZSR:80008634	EBFFFFFA		bl	0x80008624
ZSR:80008638	F10C0080		cpsid	i
ZSR:8000863C	E3073A20		movw	r3,#0x7A20
ZSR:80008640	E3483072		movt	r3,#0x8072
ZSR:80008644	E3A04001		mov	r4,#0x1 ; r4,#1
ZSR:80008648	E3095DC0		movw	r5,#0x9DC0
ZSR:8000864C	E5C34004		strb	r4,[r3,#0x4]
ZSR:80008650	EB002FE2		bl	0x800145E0
ZSR:80008654	EB08D24A		bl	0x8023CF84
ZSR:80008658	E1A01004		cpy	r1,r4

You can load kernel symbols (elf/dwarf file vmlinux) and debug it from start_kernel()

address	name
P:8003A484--8003A5F7	\init/main
P:80008224--8000823B	+ set_reset_devices
P:8000823C--80008253	+ debug_kernel
P:80008254--8000826B	+ quiet_kernel
P:8000826C--8000829B	+ init_setup
P:8000829C--800082CB	+ rdinit_setup
P:800082CC--800082F7	+ loglevel
P:800082F8--800083A3	+ do_early_param
P:800083A4--800085B3	+ unknown_bootoption
P:800085B4--800085E7	+ parse_early_options
P:800085E8--80008623	+ parse_early_param
P:80008624--80008627	+ smp_setup_processor_id
P:80008628--8000862B	+ thread_info_cache_init
P:8000862C--80008917	+ start_kernel
P:80008918--80008A5B	+ kernel_init
D:8002DD00--8002DD0B	__setup_rdinit_setup
D:8002DD0C--8002DD17	__setup_init_setup
D:8002DD18--8002DD23	__setup_loglevel
D:8002DD24--8002DD2F	__setup_quiet_kernel
D:8002DD30--8002DD3B	__setup_debug_kernel
D:8002DD3C--8002DD47	__setup_set_reset_devices
D:8002EA18--8002EA2B	kthreadd_done
D:800375E8--800375EF	__setup_str_rdinit_setup
D:800375F0--800375F5	__setup_str_init_setup

```

456 asmlinkage void __init start_kernel(void)
457 {
ZSR:8000862C E92D40F0 start_kernel:  push  {r4-r7,r14}
ZSR:80008630 E24DD014                sub   r13,r13,#0x14 ; r13,r13
char * command_line;
extern const struct kernel_param __start__param[],

461 smp_setup_processor_id();
ZSR:80008634 EBFFFFFA                bl    0x80008624 ; smp_set
                                cpsie i @ arch_local
                                :
                                : "memory", "cc");
    
```



Debugging linux components: kernel debugging

With the symbols (vmlinux) you can do source-level debugging of kernel: you can set breakpoints, run in step, see functions, registers, static variables and local variables in the stack-frame ...

[B::Data.List] Step Over Next Return Up Go Break

```

addr/line source
128 /*
129  * This is our default idle handler. We need to
130  * interrupts here to ensure we don't miss a wak
131  */
128 static void default_idle(void)
129 {
130     if (hlt_counter)
131         cpu_relax();
132     else {
133         local_irq_disable();
134         if (!need_resched()) {
135             timer_dyn_reprogram();
136             arch_idle();
137         }
138     }
139     local_irq_enable();
140 }
    
```

B::Frame.view /Locals /Caller

```

-000| default_idle()
      |   . flag = 1
      |   . nr = 1
-001| arch_local_save_flags(inline)
-001| cpu_idle()
      |   . flag = 1
      |   . nr = 1
      |   } else {
      |       stop_criti
      |       pm_idle();
-002| start_kernel()
      |   ⊕ command_line = 0x8002F82C
      |   ⊕ command_line = 0x8002F82C
      |
      |   acpi_early_init(); /* before LAPIC
      |   sfi_init_late();
      |
      |   ftrace_init();
      |
      |   /* Do the rest non-__init'ed, we'r
      |   rest_init();
-003| ZSR:0x4001:0x10008040(asm)
    
```

B::Var.View %string (\\vmlinux\init\main\start_kernel\command_line)

```

(\\vmlinux\init\main\start_kernel\command_line) = 0x8002F82C → "console=ttyxc1,115200_vmalloc=400M_ubi.mtd=1_root=ubi0:rootfs
    
```



Debugging linux components: multicore debugging

The kernel is initializing with only core 0 active. In kernel_init() function is called smp_init() which activates the secondary cores 1, 2, 3.

```

[B::List kernel_init]
Step Over Diverge Return Up Go Break Mode
addr/line source
777 static int __init kernel_init(void * unused)
778 {
    /*
    * Wait until kthreadd is all set-up.
    */
782 wait_for_completion(&kthreadd_done);
    /*
    * Now the scheduler is fully set up and can
    * gfp_allowed_mask = __GFP_BITS_MASK;
785
    /*
    * init can allocate pages on any node
    */
    set_mems_allowed(node_states[N_HIGH_MEMORY]);
    /*
    * init can run on any cpu.
    */
794 set_cpus_allowed_ptr(current, cpu_all_mask);
796 cad_pid = task_pid(current);
798 smp_prepare_cpus(setup_max_cpus);
    do_pre_smp_initcalls();
    lockup_detector_init();
803 smp_init();
804 sched_init_smp();
  
```

The system becomes now multicore, so you must configure TRACE32 to handle 4 cores simultaneously in SMP mode (Symmetric Multi Processing):

CORE.ASSIGN 1 2 3 4

CORE			CoreNumber
core	chip	corename	
1.	1.		4.
Chip: IMX63			
Core: CORTEXA9MPCORE			

Debugging linux components: multicore debugging

After the execution of `smp_init()` and configuration of TRACE32 in multicore mode you can open new views on other cores, which will be running starting from `secondary_start_kernel()` function.

The image displays three overlapping windows from the TRACE32 debugger, illustrating the state of a multicore system during kernel initialization.

- Primary Core View (Left):** Shows the `kernel_init` function. The `smp_init();` call at line 803 is circled in red. Red arrows point from this call to the secondary core views.
- Secondary Core 1 View (Bottom Right):** Shows the `secondary_start_kernel` function. The `__cpuinit secondary_start_kernel(void)` function is highlighted in blue. The code shows the initialization of the memory management context for the processor.
- Secondary Core 2 View (Middle Right):** Shows the same `secondary_start_kernel` function, with the `__cpuinit secondary_start_kernel(void)` function highlighted in blue.
- Secondary Core 3 View (Top Right):** Shows the same `secondary_start_kernel` function, with the `__cpuinit secondary_start_kernel(void)` function highlighted in blue.

The secondary core views show the following code snippets:

```

/* idle thread stack, but a set of temporary page tables
*/
asmlinkage void __cpuinit secondary_start_kernel(void)
{
    struct mm_struct *mm = &init_mm;
    unsigned int cpu = smp_processor_id();

    /* All kernel threads share the same mm context; gra
    * reference and switch to it.
    */
    atomic_inc(&mm->mm_count);
    current->active_mm = mm;
    cpumask_set_cpu(cpu, mm->cpumask);
    cpu_switch_mm(mm->pgd, mm);
    enter_lazy_tlb(mm, current);
    local_flush_tlb_all();

    printk("CPU%u: Booted secondary processor\n", cpu);

    cpu_init();
    preempt_disable();
  }

```

Debugging linux components: multicore debugging

In multicore mode SMP, all cores are handled simultaneously for commands as GO/BREAK/STEP and Breakpoints. For each window, you can have a specific view/color by core, or you can select the default core view in the Cores menu.

TRACE32 ARM: debug Linux secondary boot (step 2/2)

File Edit View Var Break Run CPU Misc Trace Perf Cov IMX61 Linux Window Help

[B::List /core 0]

```
static void __init do_basic_setup(void)
{
    cpuset_init_smp();
    usermodehelper_init();
    init_tmpfs();
    driver_init();
    init_irq_proc();
    do_ctors();
    do_initcalls();
}
```

[B::var.frame /core 2]

```
current_thread_info()
debug_smp_processor_id()
    this_cpu = -2141932348
    __func__ = (100, 101, 98, 117, 103, 98)
    sp = 3590782928
    nbits = 4
    __func__ = (100, 101, 98, 117, 103, 98)
test_bit(inline)
cpu_idle()
    flag = 1
    nr = 0
#ifdef CONFIG_HOTPLUG_CPU
```

[B::list /core 1]

```
test_bit(inline)
test_ti
need_res
cpu_idle
    flag = 1
    nr = 1
ZSR:0xFF
end of f
```

[B::list /core 1]

```
static inline int test_bit(int nr, const volatile unsigned long *addr)
{
    return 1UL & (addr[BIT_WORD(nr)] >> (nr & (BITS_PER_LONG-1)));
}
#endif /* _ASM_GENERIC_BITOPS_NON_ATOMIC_H_ */
```

[B::data.listmix /core 3]

```
@nr: bit number to test
@addr: Address to start counting from
static inline int test_bit(int nr, const volat
{
    return 1UL & (addr[BIT_WORD(nr)]
F: 80040B9C E5953000 ldr r3, [r5]
F: 80040BA0 E3500000 cmp r0, #0x0
F: 80040BA4 E280201F add r2, r0, #0x1F
F: 80040BA8 A1A02000 cpyge r2, r0
F: 80040BAC E200001F and r0, r0, #0x1F
```

T32 DEMO

Debugging linux components: linux aware debugging

Kernel debugging can be done with a JTAG debugger also not specific to Linux. The entire kernel block can be considered as a single program (very big).

HOWEVER this is not enough to debug an entire Linux system

- ❑ How can you make the debugging of "dynamic objects" as the processes and their threads, libraries and kernel modules?
- You must consider the memory management (MMU) of the CPU and the kernel
- The debugger must be aware of the running operating system. Must give a view of the resources of Linux and have specific commands for their debug.

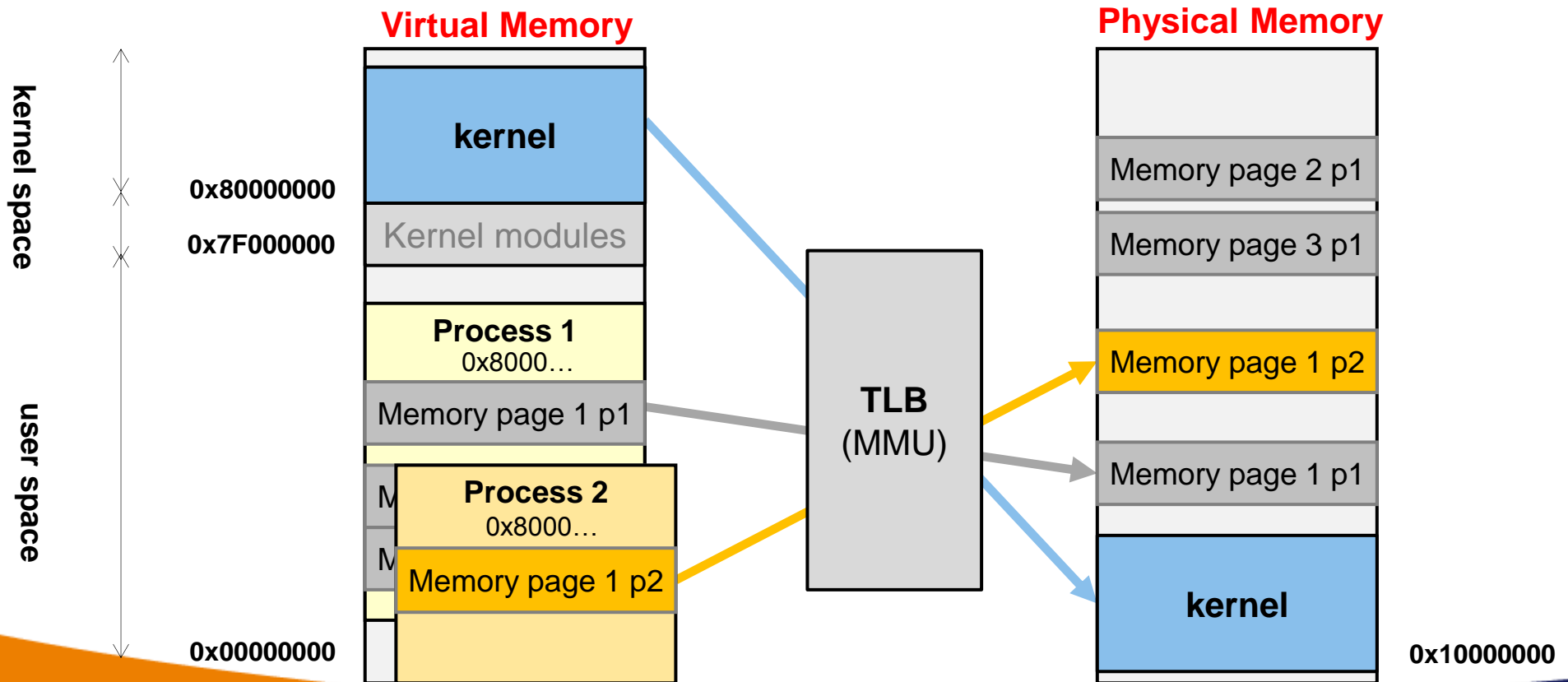
All of this is managed by the extension

TRACE32 Linux Awareness

(linux.t32)

Debugging linux components: Memory Management (1)

- ❑ The different components of the system are physically loaded in memory at absolute addresses, but execute at virtual address (logical)
- ❑ Kernel has a fixed Virtual-Physical translation
- ❑ Processes have dynamics Virtual-Physical translations



Debugging linux components: Memory Management (2)

- ❑ During debugging, the user uses logical addresses to access programs, data, symbols loaded into virtual memory *
- ❑ The core and the debugger can only access the active memory pages (TLB)
- ❑ TRACE32 can access the entire memory even at absolute addresses
- If a virtual memory area is not accessible, then TRACE32 computes the logical-physical translation, and make an access to physical address.

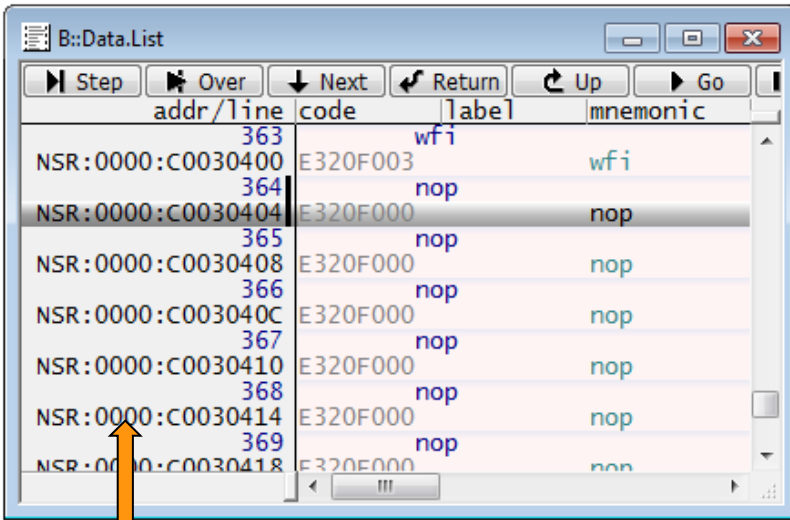
In this way TRACE32 allows the user to access at any time and debug in any area of memory using simple virtual addresses (logical)

* virtual addresses (logical) correspond to the program symbols

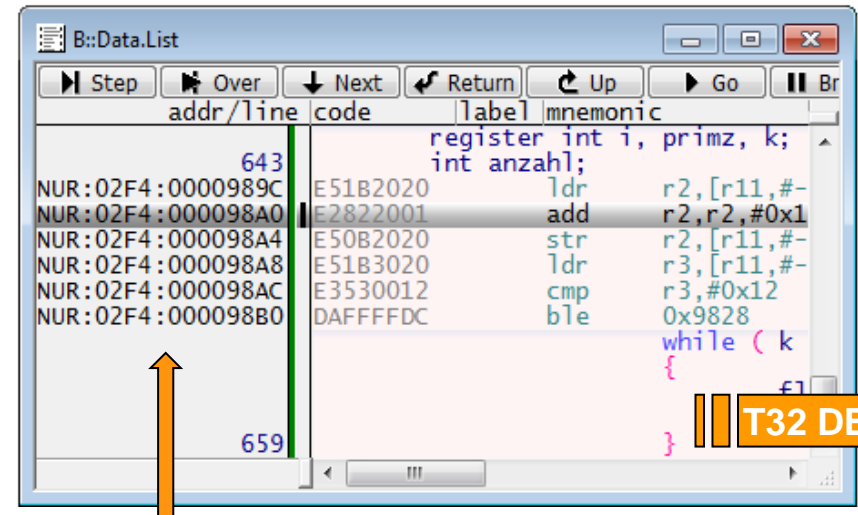
Debugging linux components: Address Extension

How to distinguish between kernel and process and between different processes?

- ❑ In Linux, the **space-id** of a process is the **PID** of his main thread
- ❑ The kernel and all its threads have by convention space-id = zero
- TRACE32 uses the identifier space-id to distinguish between the different processes by extending the address space.
- The Address Extension is enabled by the command **SYStem.Option MMUSPACES ON**



Space-id = **0x0000** : kernel thread

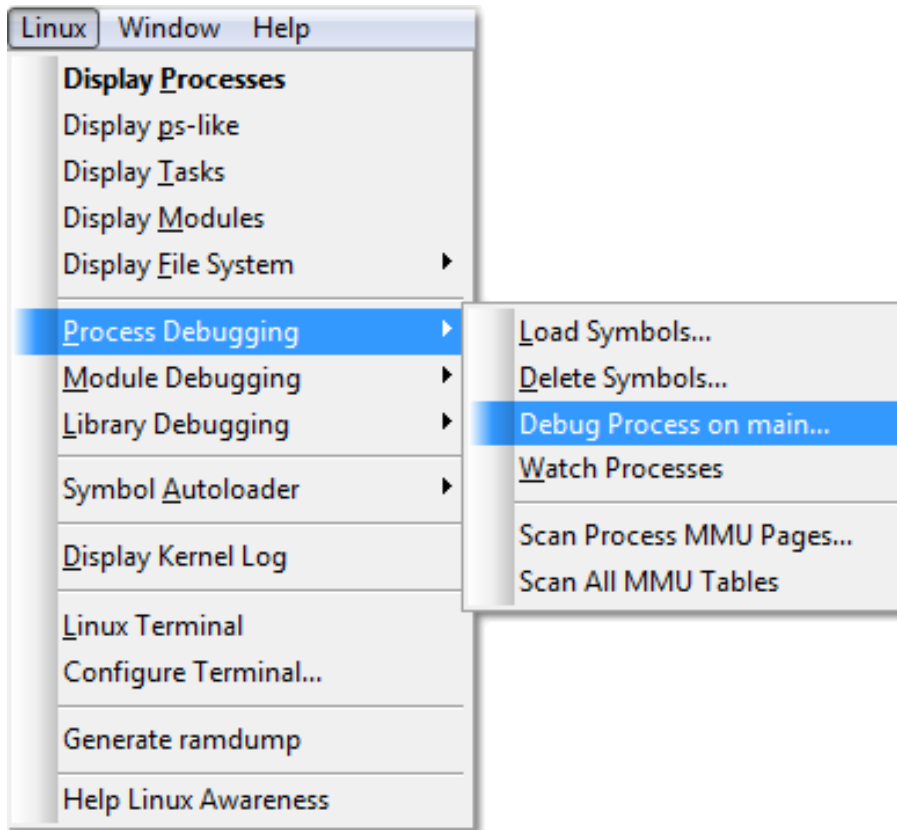


Space-id = **0x02F4** : user process PID 0x02F4



Debugging linux components: linux menu

Thanks to Memory Management and Address Extension TRACE32 allows the access and debugging of any part of a Linux system.



TRACE32 Linux Awareness menu

- Debugging the kernel
- Debugging kernel modules
- Debugging processes/threads
- Debugging libraries
- Automatically loading and unloading symbols for kernel modules, processes and libraries
- Display kernel information (file systems, kernel log, device tree...)

Debugging linux components: kernel module

Debug Module on init

magic	name	state	size	address
BF0042AC	demomod	Loading	3958.	BF004000

```

[B::List]
Step Over Next Return Up
addr/line source
+ 51 int init_module()
+ 52 {
+   Our_Proc_File = create_
+
+ 54     if (Our_Proc_File == NU
+ 55         remove_proc_ent
+ 56         printk(KERN_ALE
+           PROCFS_N
+ 58         return -ENOMEM;
+   }
+
+ 61     Our_Proc_File->read_pro
+ 62     Our_Proc_File->write_pr
    
```

Debug Mod...

module name (without .ko)

demomod

Ok Cancel

Please enter module name **2**

- Linux Window Help
- Display Processes
 - Display ps-like
 - Display Tasks
 - Display Modules
 - Display File System
- Process Debugging
- Module Debugging** **1**
 - Load Symbols...
 - Delete Symbols...
 - Debug Module on init...**
 - Scan All MMU Tables
- Library Debugging
- Symbol Autoloader
- Display Kernel Log
- Linux Terminal
- Configure Terminal...
- Generate ramdump
- Help Linux Awareness

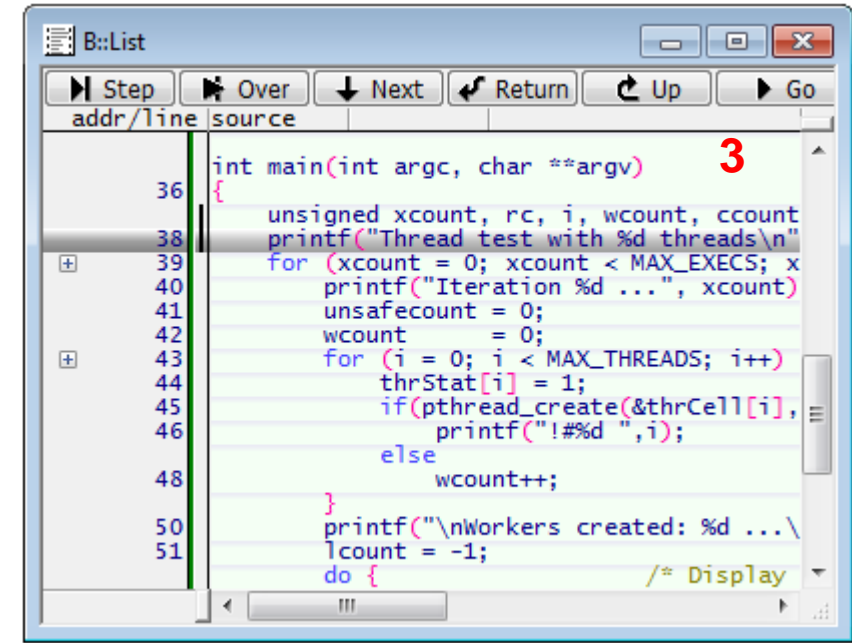
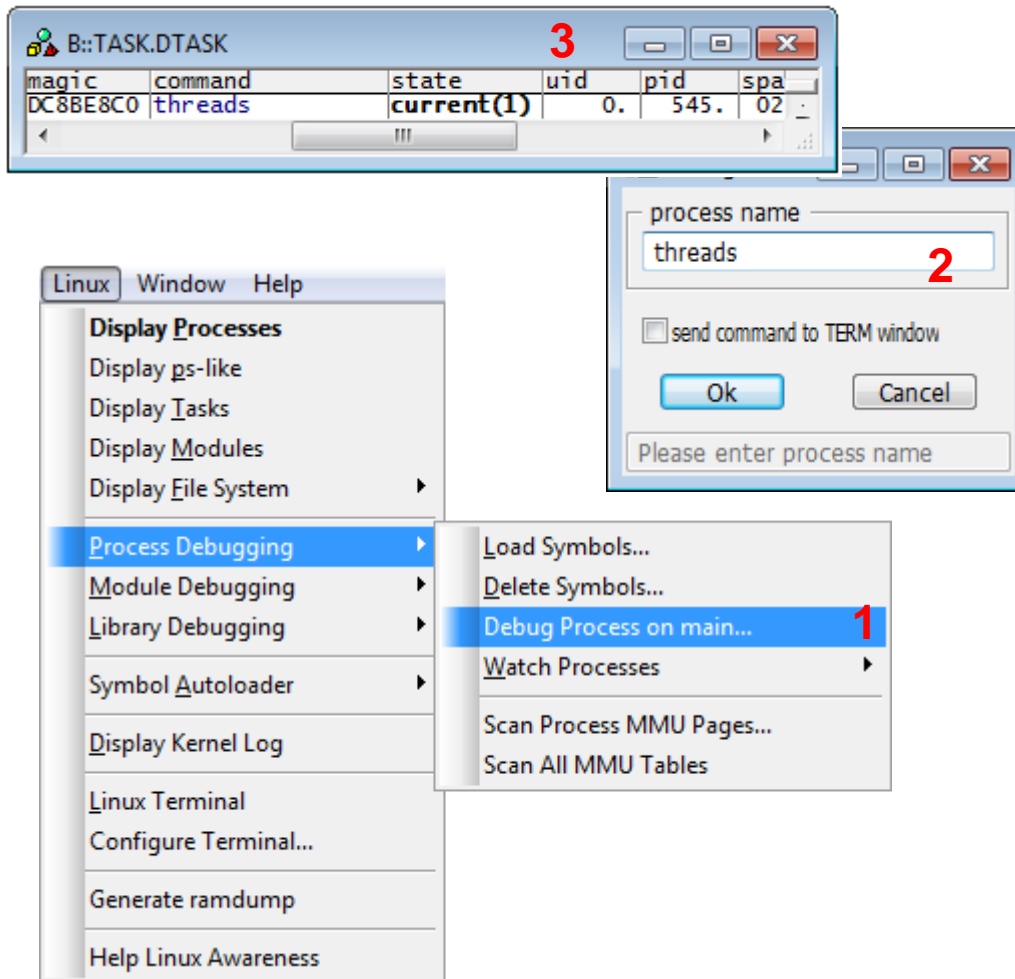
```

B::TERM.view
#
#
# insmod /lib/modules/demomod.ko 3
    
```



Debugging linux components: process (1)

Debug Process on main



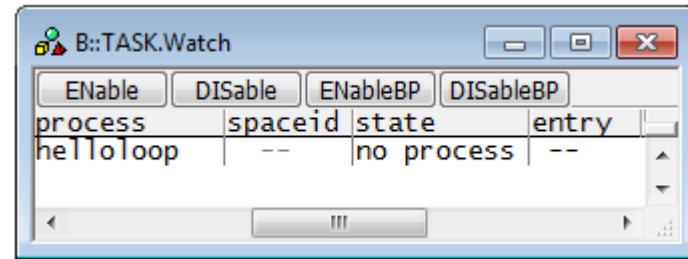
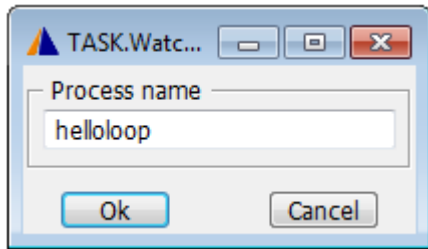
As soon as the process starts the debugger will detect it, will automatically load his symbols and will stop at the main() entry

Debugging linux components: process (2)

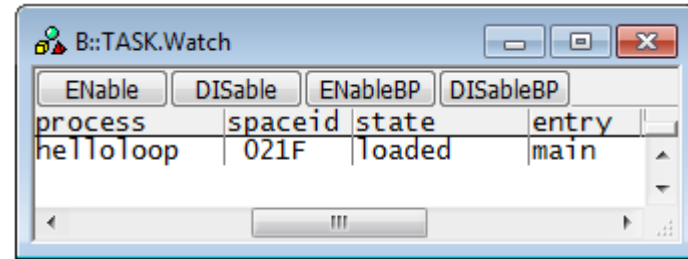
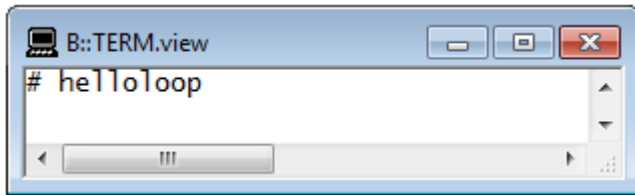
Process Debugging – Watch Processes (autoloader)

Adding a process to the "watch" list, his symbols will be automatically loaded when it start, and deleted when it exit

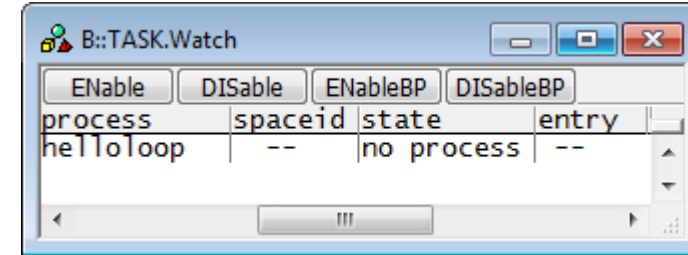
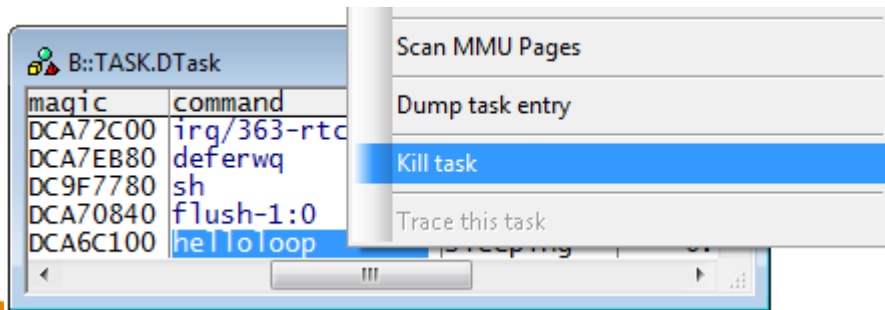
add *helloworld* to watch list



Start *helloworld*



symbols loaded



symbols deleted

Debugging linux components: process (3)

TRACE32 can show detailed informations of any process or thread

B::TASK.DTask "helloloop"

magic	command	state	uid	pid	sp
DC289540	helloloop	sleeping	0.	749.	0

qid	vm size	ttb	tty name	path
0.	0000016A	DC108000	-	/bin/hellooop

flags: RANDOMIZE
 parent: youngest child, younger sibling
 sh
 arguments: helloloop
 environment:
 USER=root
 HOME=/
 TERM=vt102
 PATH=/usr/bin:/bin:/usr/sbin:/sbin
 SHELL=/bin/sh
 PWD=/

open files
 code addr/size data addr/size stack st
 00008000 / 00000488 00010488 / 0000011C BE8D7EC0
 code file start address
 helloloop 00008000
 helloloop 00010000
 libc-2.5.so B6E26000
 libc-2.5.so B6F3D000
 libc-2.5.so B6F45000
 libc-2.5.so B6F47000
 ld-2.5.so B6F4B000
 ld-2.5.so B6F6E000

start time	sleep time	user time	system time
02D939BF	00000102	00000000	00000000

B::TASK.Process

magic	command	#thr	state	spaceid	pids
C0791CC8	swapper/0	47.	current	0000	0. 2.
DC0373C0	init	-	sleeping	0001	1.
DC127500	sh	-	sleeping	02E5	741.
DC2881C0	sieve	-	sleeping	02EC	748.
DC289540	helloloop	-	sleeping	02ED	749.

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

Display task specific information

Debugging linux components: process (4)

TRACE32 can show the execution context (program, data, registers) of any process or thread even if suspended, or if you are in break elsewhere

B::Frame /TASK "helloloop"

Task: "helloloop"

- 000 need_resched()
- 001 do_nanosleep(t = 0xDC73FF28, ?)
- 002 hrtimer_nanosleep(?, rmtmp = 0xBE8D7D30, mode = HRTIMER_MODE_REL, ?)
- 003 sys_nanosleep(?, rmtmp = 0xBE8D7D30)
- 004 ret_fast_syscall(asm)
- exception
- 005 NUR:0x2ED:0xB6EB678C (asm)
- 006 NUR:0x2ED:0xB6EB6538 (asm)
- end of frame

[B::List /TASK "helloloop"]

addr/line	code	label	mnemonic
NSR:02ED:C0541F6C	E5941004		ldr r1,[r4,#0x4]
NSR:02ED:C0541F70	E1A00004		cpy r0,r4
NSR:02ED:C0541F74	E5972004		ldr r2,[r7,#0x4]
NSR:02ED:C0541F78	EB000753		bl 0xC0543CCC
NSR:02ED:C0541F7C	E1A01000		cpy r1,r0

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

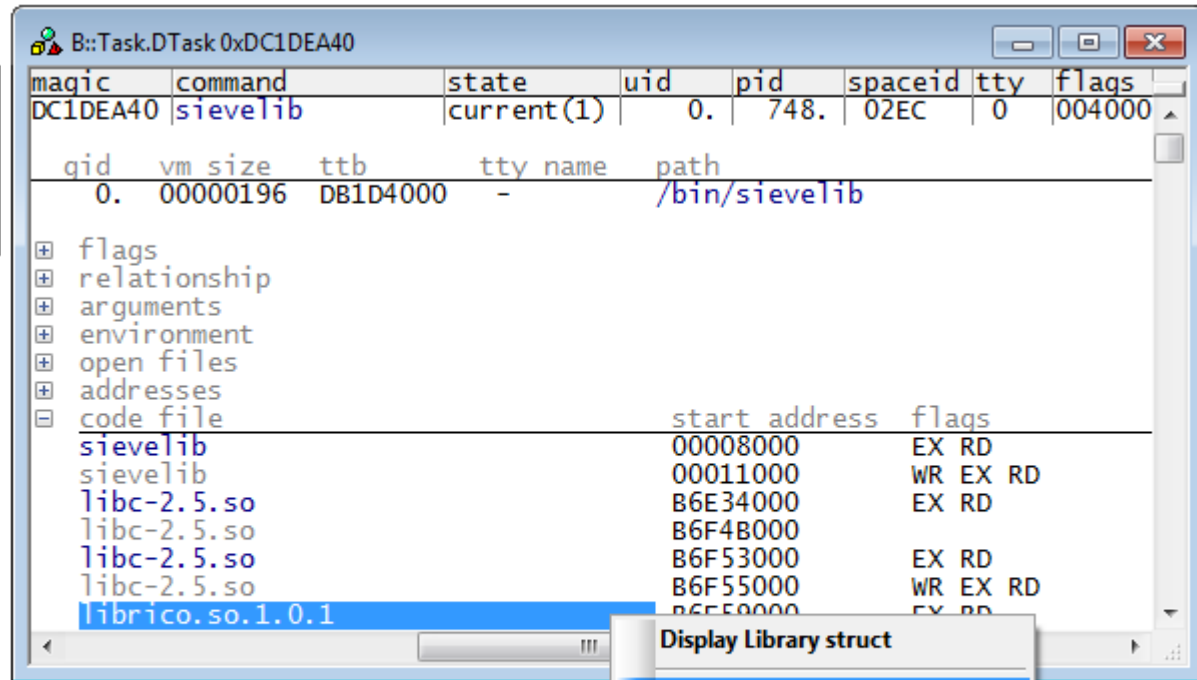
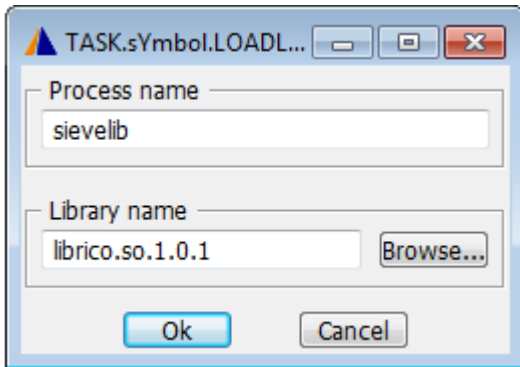
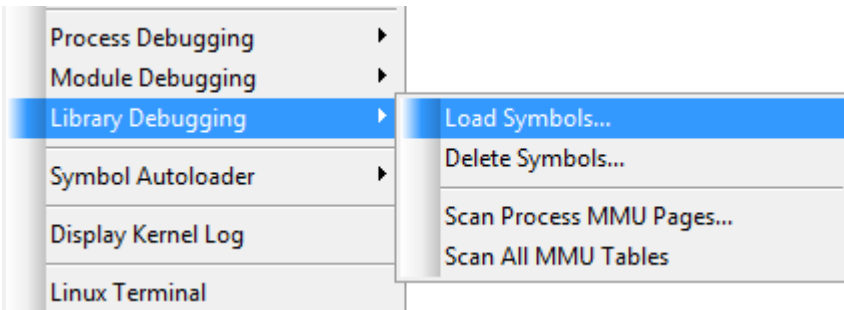
B::Register /TASK "helloloop"

N	Z	C	V	Q	0	1	2	Register	Value	Register	Value	Register	Value
-	-	-	-	-	-	-	-	R0	C0014E3C	R8	1	SP	00000000
-	-	-	-	-	-	-	-	R1	8AAD	R9	DB5C5240	-BC	C0098C2C
-	-	-	-	-	-	-	-	R2	0	R10	C078FD88	-B8	A0000113
-	-	-	-	-	-	-	-	R3	C00320C0	R11	DC73FEF4	-B4	C1124B80
-	-	-	-	-	-	-	-	R4	DC289540	R12	0	-B0	C1124BD0
-	-	-	-	-	-	-	-	R5	0	R13	DC73FE38	-AC	DC73E000
-	-	-	-	-	-	-	-	R6	C1126C80	R14	C00303F4	-A8	C3A95AC4
-	-	-	-	-	-	-	-	R7	DC078AC0	PC	C0541F7C	-A4	C0543520
-	-	-	-	-	-	-	-	SPSR	20000113	CPSR	13	-A0	DC73FF28



Debugging linux components: library

TRACE32 can load symbols of any library used by a process and enable its debugging



TASK.sYmbol.LOADLIB <process_name> <lib_name>

Debugging linux components: file system informations

TRACE32 can show detailed information about the file systems

- Display **P**rocesses
 - Display **g**s-like
 - Display **T**asks
 - Display **M**odules
 - Display **F**ile System
 - Display FS Types
 - Display Mount Points
 - Display Mounted Devices
 - Display /proc
 - Display /sys
- Process Debugging
- Module Debugging
- Library Debugging
- Symbol Autoloader
- Linux Terminal
- Configure Terminal...
- Help Linux Awareness

B::TASK.FS.Type...

magic	name	#dev
C07EAFE4	sysfs	1.
C07EFA80	rootfs	1.
C07EA234	bdev	1.
C07EAA50	proc	1.
C07E791C	tmpfs	1.
C07FC37C	devtmpfs	1.
C07EA5C8	binfmt_mi	0.
C07F2984	debugfs	1.
C080E200	sockfs	1.
C07E94C0	pipefs	1.
C07EA478	anon_inod	1.
C0814738	rpc_pipef	0.
C07EB090	devpts	1.
C07EB2B8	ext3	
C07EC420	ext2	
C07EC4CC	ext4	
C07EF6D4	cramfs	
C07EF780	ramfs	
C07EFB5C	vfat	
C07EFBC0	msdos	
C07EFE20	nfs	
C07EFD8C	nfs4	
C07F24D0	jffs2	
C07F2DE8	mqueue	
C07FEE50	mtd_inode	
C080E0E0	oprofilef	
C07F27B0	ubifs	

B::TASK.FS.PROC

name	address	mode	links
/proc	C07EA9D0	dr-xr-xr-x	12.
mtd	DB16AEC0	-r--r--r--	1.
sysrq-trigger	DB038200	--w-----	1.
partitions	DB038380	-r--r--r--	1.
diskstats	DB038440	-r--r--r--	1.
crypto	DB0383C0	-r--r--r--	1.
key-users	DB038A40	-r--r--r--	1.
kpageflags	DC7F1240	-r-----	1.
kpagecount	DC7F1300	-r-----	1.
kmsg	DC7F34C0	-r-----	1.
softirqs	DC7F13C0	-r--r--r--	1.
version	DC7F3540	-r--r--r--	1.
uptime	DC7F35C0	-r--r--r--	1.

B::TASK.FS.SYS

name	address	mode	count
/sys	C07EAF94	dr-xr-xr-x	12.
firmware	DC02B860	drwxr-xr-x	1.
kernel	DC09DCC0	drwxr-xr-x	13.
profiling	DC09DB80	-rw-r--r--	1.
kexec_loaded	DC09DB30	-r--r--r--	1.
fscaps	DC09DC70	-r--r--r--	1.
notes	DC09D9A0	-r--r--r--	1.
rcu_expedited	DC09D9F0	-rw-r--r--	1.
mm	DC7F0130	drwxr-xr-x	1.
module	DC1F84A0	drwxr-xr-x	41.
dev	DC02BA40	drwxr-xr-x	3.
fs	DC02B590	drwxr-xr-x	2.

B::TASK.FS.MountDevs

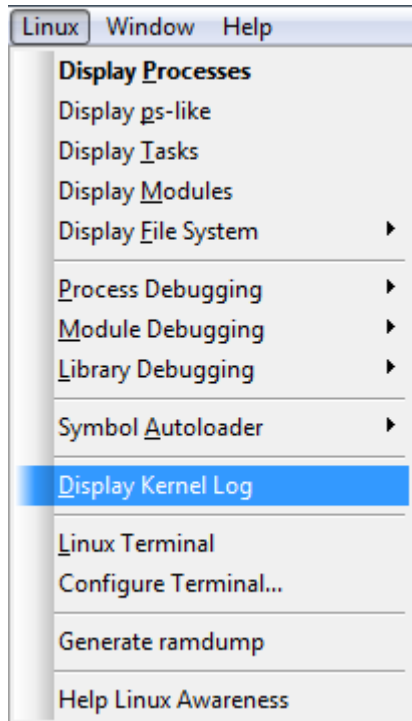
magic	dev#	fsmagic	type	root
DC027000	0.	beer	sysfs	/
DC02C800	1.	858458F6	rootfs	/
DC02C000	2.	bdev	bdev	bdev:
DC03E800	3.	00009FA0	proc	/
DC098800	4.	01021994	tmpfs	/
DC098000	5.	01021994	devtmpfs	/
DC0CA000	6.	64626720	debugfs	/
DC0D0000	7.	SOCK	sockfs	socket:
DC23A000	8.	PIPE	pipefs	pipe:
DC386800	9.	09041934	anon_ino	anon_inod
DC7FC000	10.	00001CD1	devpts	/
DB0B1000	11.	19800202	mqueue	/
DB5C6800	1048.	0000EF53	ext2	/

B::TASK.FS.Mount

magic	device	mountpoint	type
DC028B00	rootfs	/	rootfs
DC0D3CC0	/dev/root	/	ext2
DC028740	devtmpfs	/dev	devtmpf
DC0D3C00	/proc	/proc	proc

Debugging linux components: kernel log

TRACE32 can show the Kernel Log by accessing directly in memory the kernel ring buffer



```

B::TASK.DMESG
kernel ring buffer
Booting Linux on physical CPU 0x0
Linux version 3.8.2 (kjmal@kjmlinuxpc) (gcc version 4.5.2 (Sourcery G++ Lite 2011.03-41) ) #3 SMP F
CPU: ARMv7 Processor [411fc092] revision 2 (ARMv7), cr=10c53c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: OMAP4 Panda board
Memory policy: ECC disabled, Data cache writealloc
On node 0 totalpages: 118272
free_area_init_node: node 0, pgdat c0814ac0, node_mem_map c0d73000
  Normal zone: 926 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 117346 pages, LIFO batch:31
OMAP4430 ES2.1
PERCPU: Embedded 9 pages/cpu @c111b000 s13184 r8192 d15488 u36864
pcpu-alloc: s13184 r8192 d15488 u36864 alloc=9*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 117346
Kernel command line: root=/dev/ram rw mem=463M console=ttyO2,115200n8 initrd=0x81600000,4M
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
__ex_table already sorted, skipping sort
Memory: 462MB = 462MB total
Memory: 451004k/451004k available, 23108k reserved, 0K highmem
Virtual kernel memory layout:
  vector   : 0xffff0000 - 0xffff1000   (   4 kB)
  fixmap   : 0xffff0000 - 0xffffe000   ( 896 kB)
  vmalloc  : 0xdd000000 - 0xff000000   ( 544 MB)
  lowmem   : 0xc0000000 - 0xdcf00000   ( 463 MB)
  pkmap    : 0xbf000000 - 0xc0000000   (   2 MB)

```

Debugging linux components: device tree

TRACE32 can load and display the Device Tree Blob (if used)

```

B::TASK.DTB
Device Tree Blob
DTB version: 17
Memory reserve at 0x9D000000, size = 0x3000000

Tree structure:
+ /
  #address-cells      = <0x00000001>
  #size-cells         = <0x00000001>
  compatible          = "ti,omap4-panda" "ti,omap4430" "ti,omap4"
  interrupt-parent    = <0x00000001>
  model               = "TI OMAP4 PandaBoard"
  + chosen
  bootargs            = "console=ttyO2,115200n8 root=/dev/mmcblk0p2 rw rootwait"
  + aliases
  + memory
  device_type         = "memory"
  reg                 = <0x80000000 0x40000000>
  + cpus
  + cpu@0
  + cpu@1
  + soc
  + ocp
  compatible          = "ti,omap4-l3-noc" "simple-bus"
  #address-cells      = <0x00000001>
  #size-cells         = <0x00000001>
  ranges              =
  
```

TRACE32 debugging linux

User process in debug

Program counter

Symbols List

User process registers, stack

User process variables

Message area

Space-id & address

The screenshot displays the TRACE32 Linux debugging environment. At the top, the 'Linux' menu is open, showing options like 'Process Debugging', 'Module Debugging', and 'Library Debugging'. The main window shows the source code of a 'main()' function with a 'while' loop. Below the code, the 'B::Symbol.list.tree' window shows a list of symbols including 'vmlinux', 'sieve', and various functions. The 'B::Register.view / SPOTLIGHT' window shows register values for R0 through R7 and CPSR. The 'B::Var.Watch' window shows variables like 'sieve_big_loop', 'sieve_loop', and 'ast'. The 'B::TASK.Process' window shows a list of processes, with 'sieve' highlighted. The 'B::TASK.DTask 0xC7E3B580' window shows details for the 'sieve' process. The 'B::list.mix panic' window shows a kernel panic message: 'NORET_TYPE void panic(const char * fmt, ...)'.

Linux debug menu

Terminal emulator (console)

Process list

Process infos

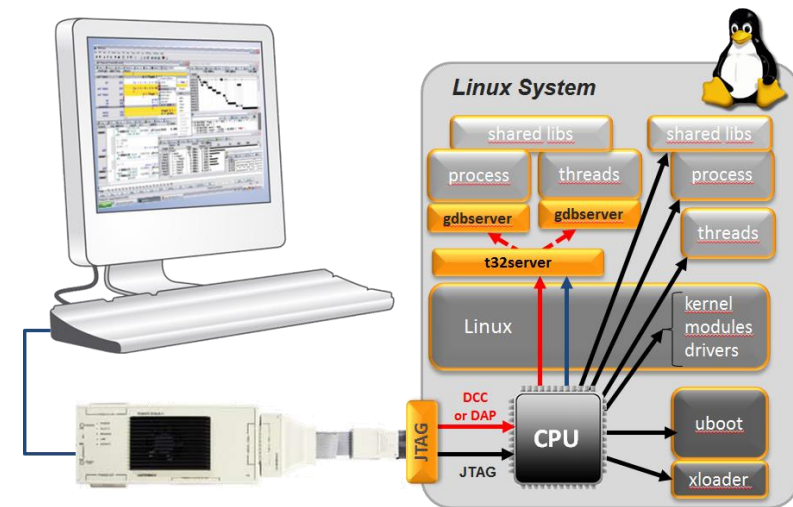
Kernel space code

Current process

Agenda

Seminar and live demo

- Linux debugging: problems & solution
- Debugging all linux components
- Stop-mode & run-mode debugging**
- Trace, performance, profiling
- TRACE32 PowerTools
- Q&A



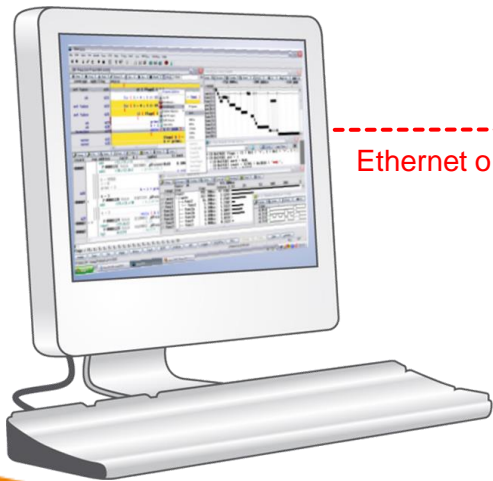
What we mean by «stop-mode debugging» ?

- Debugging via JTAG
- The "break" acts on the CPU and stops the entire linux system, including kernel, drivers, processes
- The debugger has access to all components of a linux system
- You do not need to run any sw monitor or agent or to modify the kernel: the debugger accesses directly memory and cpu registers



What we mean by «run-mode debugging» ?

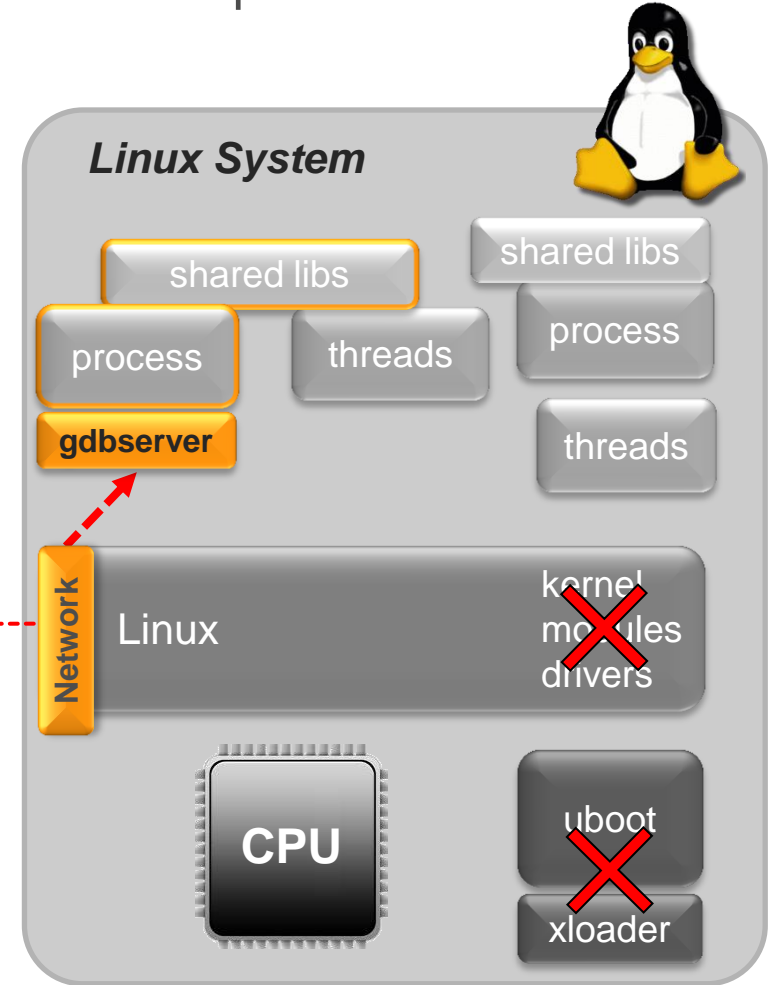
- Debugging via a communication channel: serial, ethernet
- The "break" only affects the process in debug. All other components of the linux system still running
- The debugger can only access the process stopped in debug mode
- **gdbserver** must be running in the linux system to perform the debug task: it's a target agent



TRACE32 Powerview

Ethernet or Serial

TRACE32 Powerview can also work as a front-end debugger for gdbserver



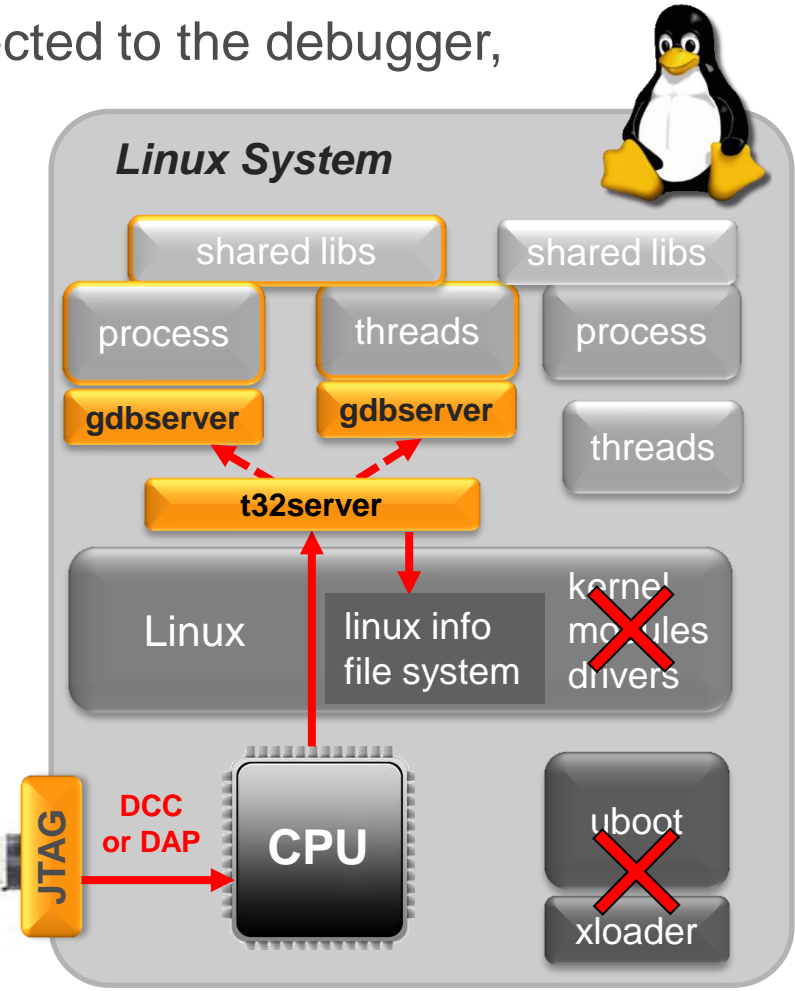
Advanced «run-mode debugging»

- Run-mode debugging through the communication channel JTAG DCC/DAP. Do not need ethernet or serial link, do not need drivers, JTAG only
- In the target execute our **t32server** agent, connected to the debugger, capable to start multiple **gdbserver** sessions
- The debugger can start/stop processes and access to some linux resources (eg. Filesystem)
- The debugger has simultaneous access to all processes being debugged



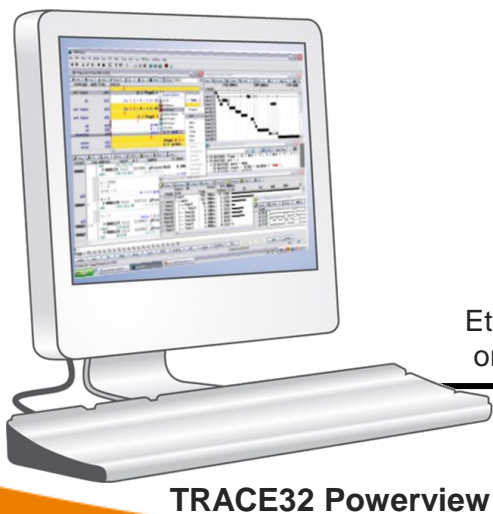
TRACE32 Powerview

Ethernet
or USB



Integrated stop-mode & run-mode debugging

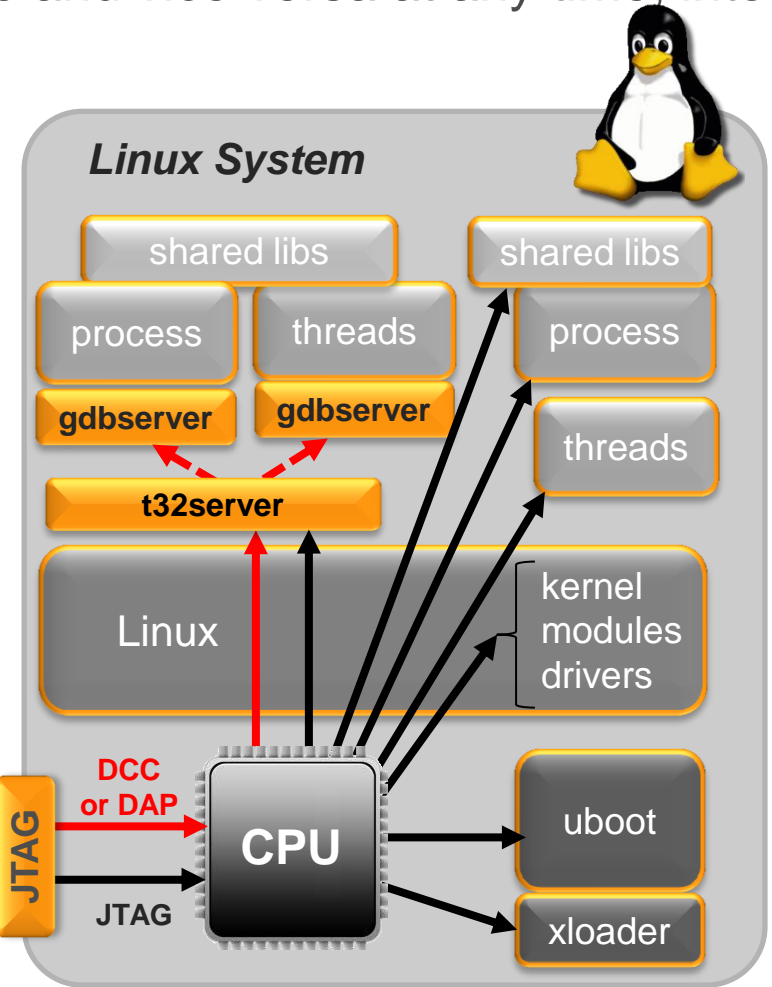
- Integration of stop-mode debugging and run-mode debugging via JTAG DCC/DAP
- TRACE32 can switch from stop-mode to run-mode and vice versa at any time, into the same debug session
- It combines the best of both debug modes, allowing users to choose the best approach to quickly solve any bug



Ethernet or USB



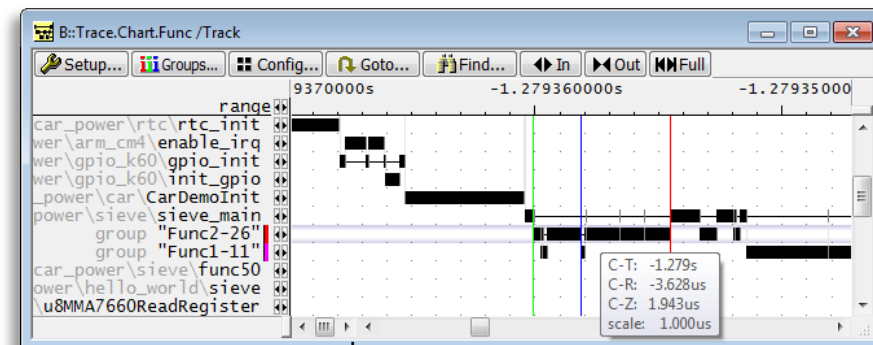
→ Run-mode debug
→ Stop-mode debug



Agenda

Seminar and live demo

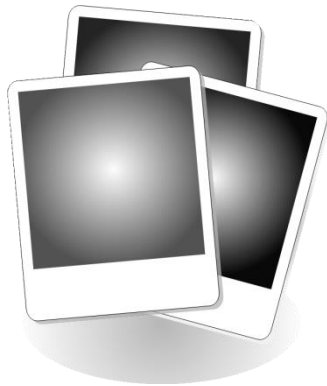
- Linux debugging: problems & solution
- Debugging all linux components
- Stop-mode & run-mode debugging
- **Trace, performance, profiling**
- TRACE32 PowerTools
- Q&A



What is the Trace?

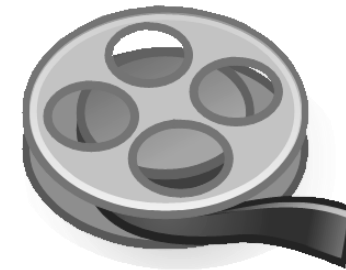
With the term "**trace**" we mean a system for recording the sequence of instructions executed and data read/written by a CPU, without having to stop it.

Debugging



Taking Pictures

Real-Time Tracing



Recording a Video

Trace ARM, ETM & ETB

Most of the chips have a **trace-port** through which the flow trace is transmitted outside: **off-chip trace**

In the ARM/Cortex™ cores the off-chip trace port is called ETM (Embedded Trace Macrocell). It is a parallel or serial port at high speed. Lauterbach has made several trace-probes to support different types of trace ports.

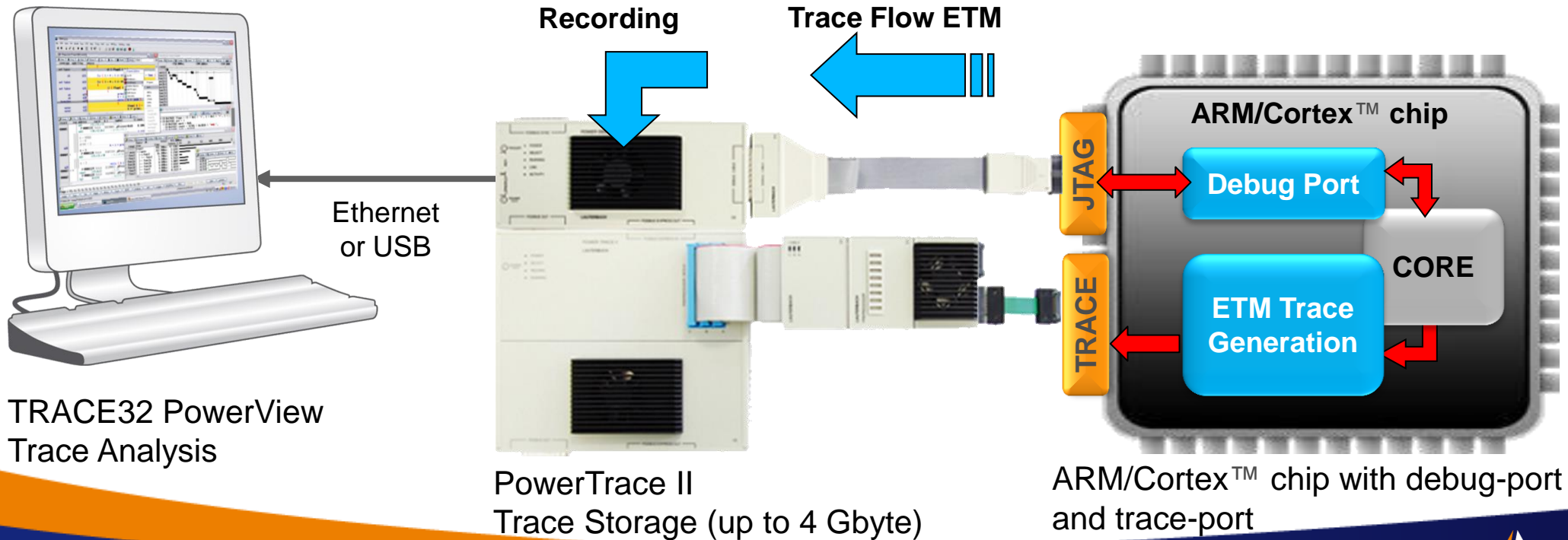


In the case where the trace port is not available, it is possible to use a trace buffer internal to the chip called ETB (Embedded Trace Buffer). It is a **on-chip trace**, typically limited to a few KByte, for which, additional hardware is not needed.

Off-chip trace ETM: recording

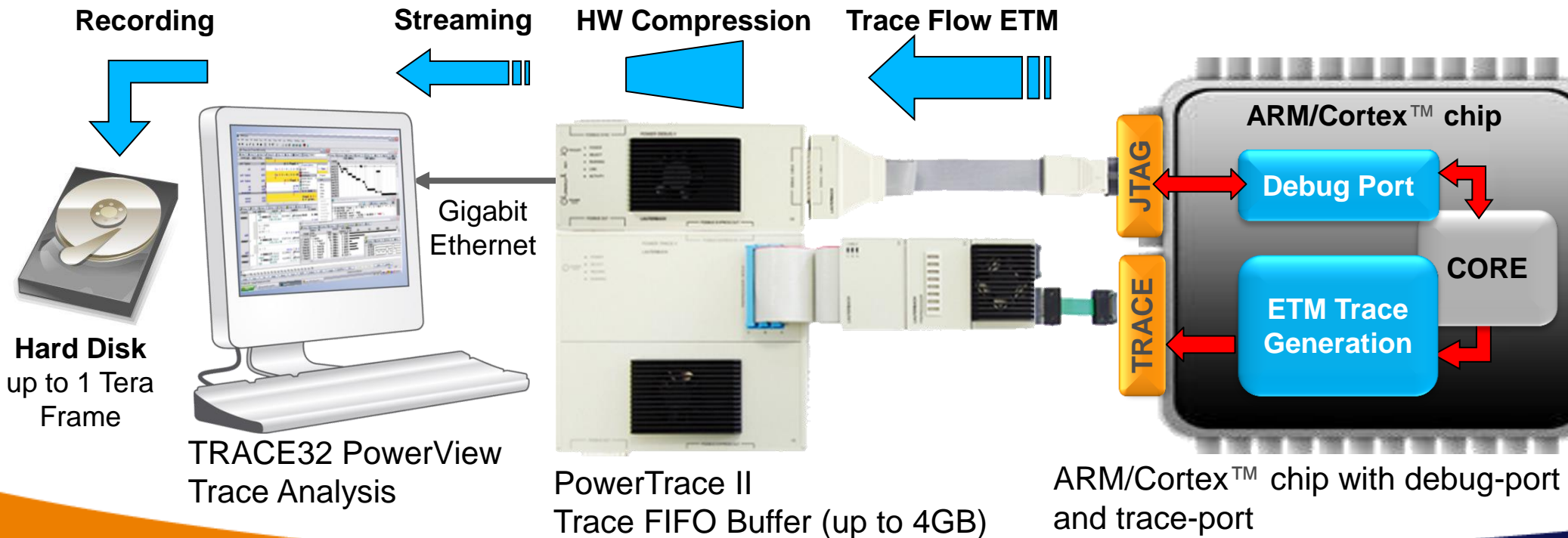
While the core is running, the trace port transmits program flow and data information in a compressed way. The method has no special restrictions:

- Few pins are required
- Allows very high speeds
- Allows trigger, filtering, data trace



Off-chip trace ETM: streaming

Normally, the trace is recorded inside PowerTrace which has a storage from 512MB up to 4GB. The recording time can be extended indefinitely using TRACE STREAMING. In this way, the trace-flow is compressed by PowerTrace II and transferred via gigabit ethernet to the host-PC, where it is registered.

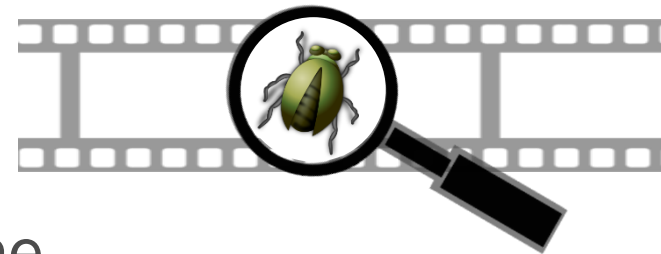


The Real-Time Trace is used for:

1) Trace-based Debugging

Fast debugging without stopping CPU

Finding bugs that only appear in real-time



2) Optimization with time measures

Analyze code performance

Analyze effect of external events



3) Qualification

Demonstrate compliance with real-time requirements

Check the code coverage



Real-time trace with Linux

- ❑ The transmission of the ETM trace is a hardware feature of the chip, it is non-intrusive and requires no modification to Linux
- ❑ The ETM trace transmits the logical addresses of program execution. But in Linux processes are all running at the same logical addresses

How to distinguish them?

- Is necessary to capture with trace also the identifier of process switches (space-id) in order to associate the recorded code to the proper linux system component

Trace with Linux: task switch

The identifier of the task switch can be easily captured by tracing writes to the variable "current process" or by tracing the "contextID" which is a ARM core register that Linux kernel updates at every task switch.

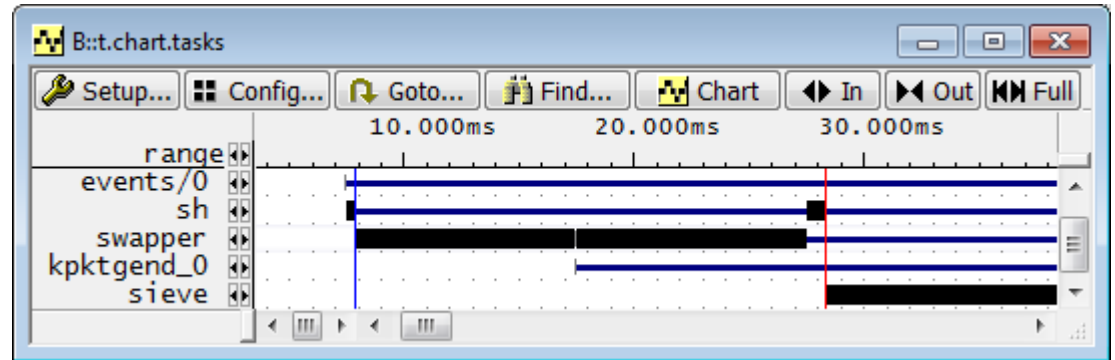
The value written (task_struct *) identifies the process and allows the immediate association of the recorded code to the proper component of the Linux system.

The screenshot displays the TRACE32 debugger interface. On the left, a window titled "B::TASK.Process" lists various processes with their magic numbers and commands. The main window shows a trace of assembly code with columns for record, run, address, cycle, data, symbol, and time. A red circle highlights the instruction "wr-long C7E3B580" in the data column, which corresponds to the "sieve" process listed in the left pane. A red arrow points from this instruction to the "sieve" entry in the process list.

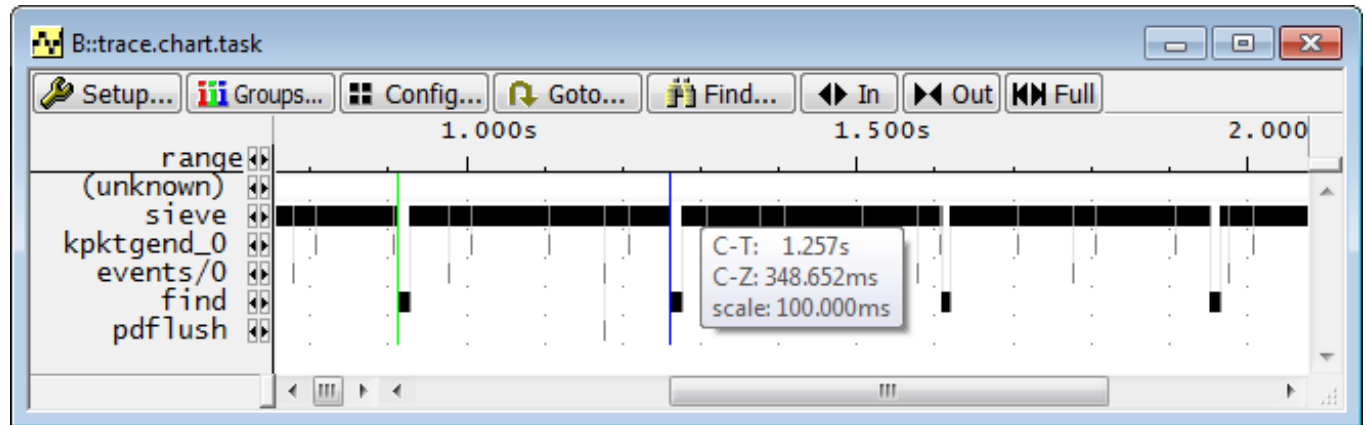
record	run	address	cycle	data	symbol	ti.back	t
+0000263428		cmp r12,#0x0 ; r12,#0		R:0000:C02B4D08 ptrace	..ux\kernel\sched\schedule+0x448	0.018us	
		#ifdef CONFIG_CODETEST		ct_thread_exit(prev);			
	3736	#endif /* CONFIG_CODETEST */		rq->nr_switches++;			
	3737	strd r0,[r3,#0x8]		rq->curr = next;			
		str r6,[r3,#0x20] ; next,[r3,#32]		TASK = C7E3B580 sieve	---		
+0000263429		D:0000:C03A0280 wr-long C7E3B580		../sched/per_cpu__runqueues+0x20		0.003us	
+0000263439		R:06AC:C02B4D10 ptrace		..ux\kernel\sched\schedule+0x450		0.003us	

Trace with Linux: task profiling

Task State
runtime chart



Task Scheduling
runtime chart



Task Timing
statistic

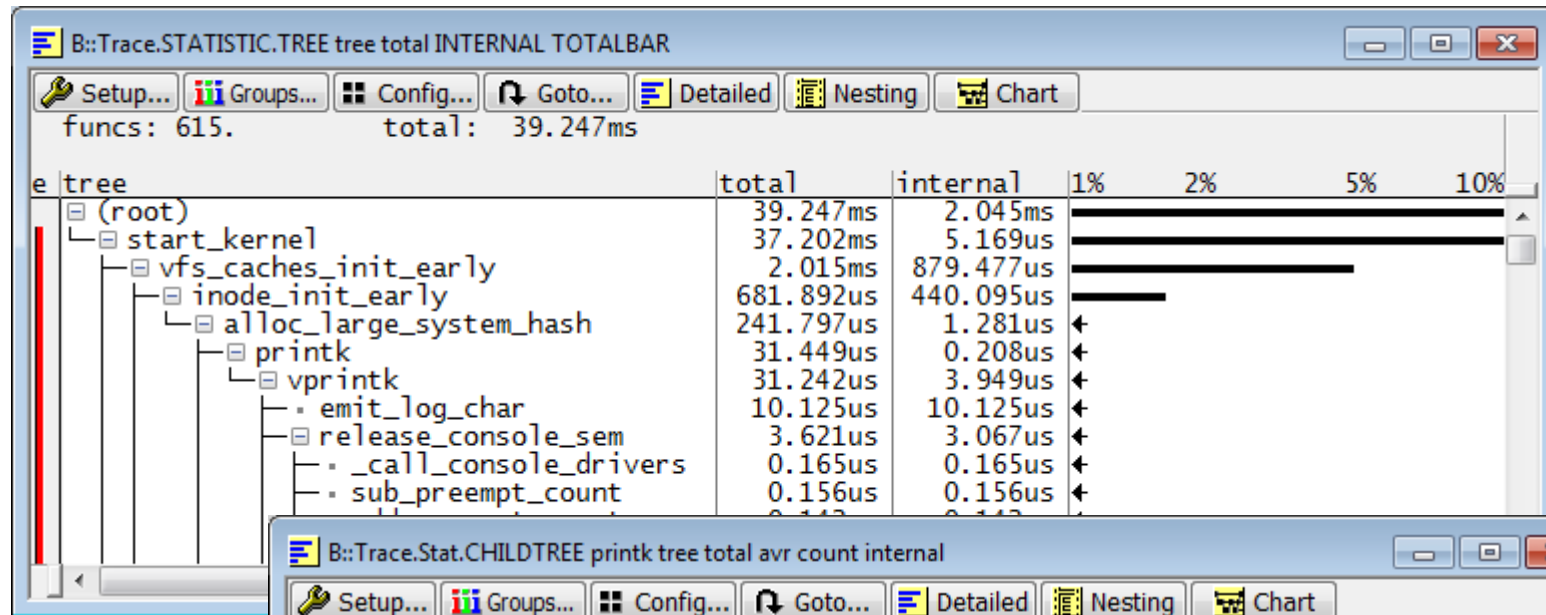
B::Trace.stat.task time avr count ratio bar

tasks: 5. total: 2.454s

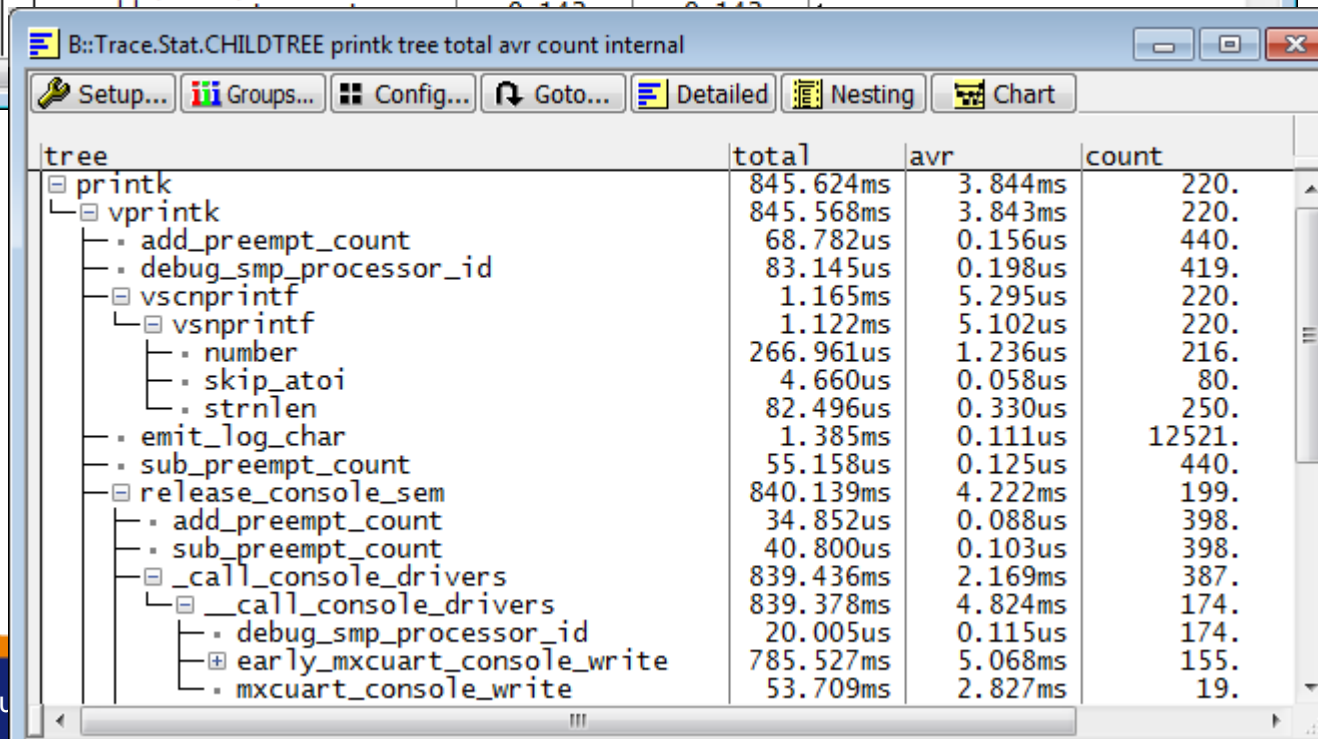
range	total	avr	count	ratio%	1%	2%	5%	10%	20%
(unknown)	2.906ms	2.906ms	0.	0.118%	←				
events/0	799.495us	53.300us	15.	0.032%	←				
find	115.105ms	12.789ms	9.	4.690%					
sieve	2.335s	51.885ms	45.	95.148%					
kpktgend_0	251.745us	10.489us	24.	0.010%	←				

Trace with Linux: code profiling

Statistic Tree Analysis



Function tree profiling of function «printk»



Trace with Linux: code coverage (1)

Code coverage
by object files

address	tree	coverage	executed	0%	50%	100%
none						
P:0000:C003C130--C003C16F	[-] \plat-mxc/irq	not exec	93.750%	[Progress bar]		
P:0000:C003C130--C003C143	[-] \plat-mxc/io	ok	100.000%	[Progress bar]		
P:0000:C003C144--C003C16F	[-] __mxc_iounmap	ok	100.000%	[Progress bar]		
P:0000:C003C170--C003C433	[-] __mxc_ioremap	not exec	90.909%	[Progress bar]		
P:0000:C003C434--C003C5AF	[-] \plat-mxc/time	partial	74.011%	[Progress bar]		
none	[-] \spba	partial	27.368%	[Progress bar]		
P:0000:C003DF8C--C003E0BF	[-] \sdma	partial	75.324%	[Progress bar]		
P:0000:C003DF8C--C003DFF3	[-] \iapiLow	partial	42.307%	[Progress bar]		
P:0000:C003DFF4--C003E00B	[-] iapi_AttachCal..	partial	42.307%	[Progress bar]		
P:0000:C003E00C--C003E033	[-] iapi_DetachCal..	ok	100.000%	[Progress bar]		
P:0000:C003E034--C003E06B	[-] iapi_ChangeCal..	ok	100.000%	[Progress bar]		
P:0000:C003E06C--C003E0BF	[-] iapi_lowSynchC..	partial	92.857%	[Progress bar]		
P:0000:C003E0C0--C003E6A3	[-] iapi_SetBuffer..	partial	85.714%	[Progress bar]		
P:0000:C003E6A4--C003EEA3	[-] \iapiLowMcu	partial	17.241%	[Progress bar]		
P:0000:C003EEA4--C0040D73	[-] \iapiMiddle	partial	9.179%	[Progress bar]		
P:0000:C003EEA4--C003EEA3	[-] \iapiHigh	partial	13.387%	[Progress bar]		
	[-] \iapiMiddleMcu	never	-	[Progress bar]		

Code coverage
by functions

tree	coverage	executed	0%	50%	100	taken	nottaken	bytes
[-] \initramfs	partial	50.189%	[Progress bar]			59.	67.	10576.
[-] read_into	partial	55.813%	[Progress bar]			0.	1.	172.
[-] do_start	partial	90.909%	[Progress bar]			0.	0.	44.
[-] write_buffer	partial	83.333%	[Progress bar]			0.	0.	96.
[-] flush_window	partial	75.342%	[Progress bar]			4.	1.	292.
[-] retain_initrd_param	never	0.000%	[Progress bar]			0.	0.	48.
[-] malloc	ok	100.000%	[Progress bar]			0.	0.	24.
[-] clean_path	partial	45.833%	[Progress bar]			1.	0.	96.
[-] do_symlink	never	0.000%	[Progress bar]			0.	0.	184.
[-] maybe_link	partial	11.538%	[Progress bar]			2.	0.	312.
[-] inflate_codes	partial	74.545%	[Progress bar]			5.	14.	1100.
[-] free	ok	100.000%	[Progress bar]			0.	0.	20.
[-] huft_free	ok	100.000%	[Progress bar]			0.	0.	52.
[-] huft_build	partial	87.335%	[Progress bar]			12.	10.	1516.
[-] inflate_fixed	partial	75.000%	[Progress bar]			5.	2.	400.
[-] do_name	partial	58.119%	[Progress bar]			2.	4.	468.

Trace with Linux: code coverage (2)

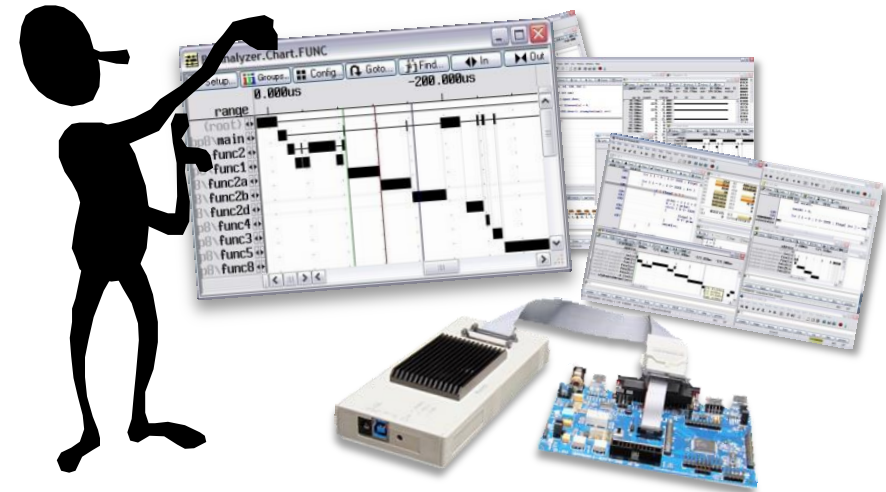
Code coverage
source code
level

coverage	addr/line	source
ok	653	static void __init do_initcalls(void)
ok	655	for (call= __initcall_start; call < __initcall_end; call++)
only exec	661	if (initcall_debug) {
never	662	printk("Calling initcall 0x%p", *call);
never	666	t0 = ktime_get();
ok	669	result = (*call)();
only exec	671	if (initcall_debug) {
never	672	delta = ktime_sub(t1, t0);
never	675	printk("initcall 0x%p", *call);
never	678	printk(" returned %d.\n", result);

The trace is an important choice

The choice of trace method depends mainly on the CPU being used and its resources.

The results obtained depend on the quality of the trace tool.



The Trace is...

➔ The tool that allows you to "see" what really happens during the execution of your application.

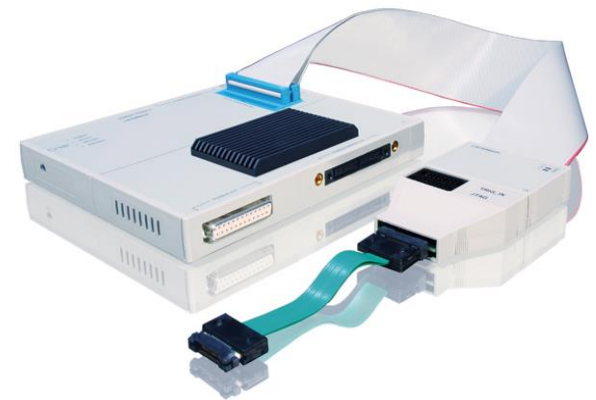
The Trace should be considered as...

➔ The instrument to reduce development time and the best guarantee to quickly find and resolve bugs

Agenda

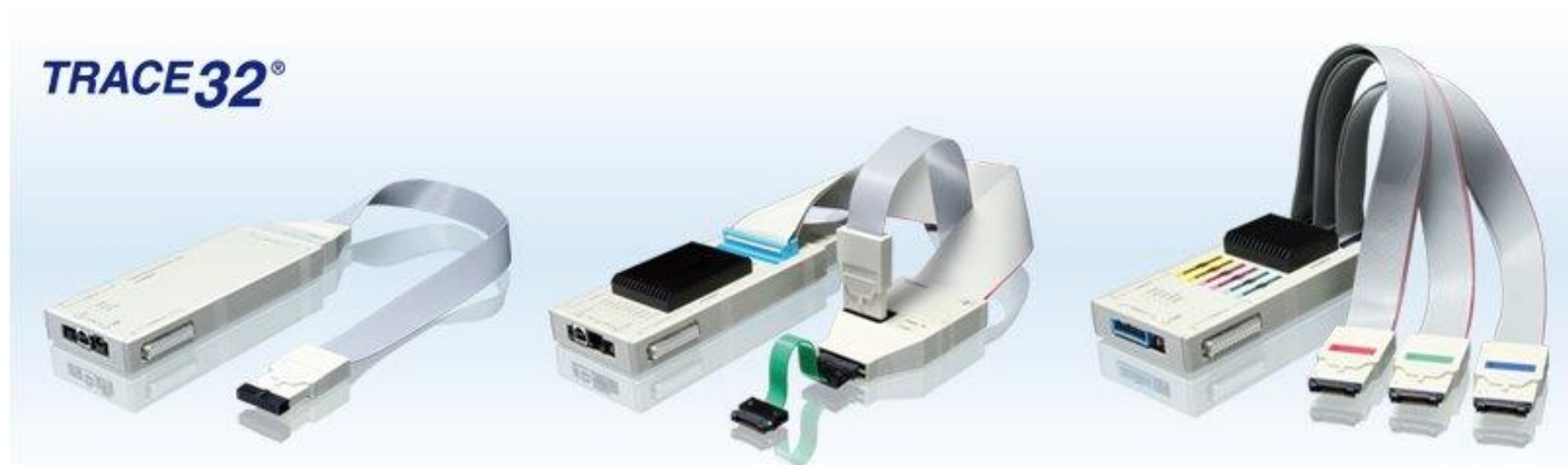
Seminar and live demo

- Linux debugging: problems & solution
- Debugging all linux components
- Stop-mode & run-mode debugging
- Trace, performance, profiling
- **TRACE32 PowerTools**
- Q&A



Lauterbach PowerTools

Lauterbach is the world leader for debug and trace tools, with over 30 years of experience. **TRACE32 PowerTools** are the most advanced hw/sw debugger available today. It is a universal and modular hardware system that support debug-port and trace-port of many different cpu and architectures.



PowerDebug
(debug)

PowerTrace
(debug+trace)

PowerIntegrator
(debug+trace+logic analyzer)

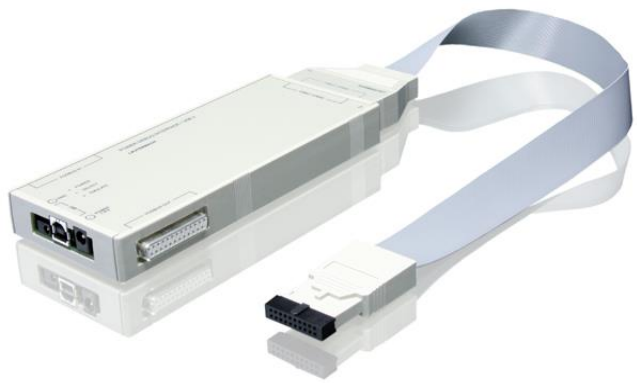
In Circuit Debuggers

A debugging system based on modular **PowerDebug** units connected to debug cables specific to different architectures and **debug ports**



Debug Cables

- Support for all CPUs
- Support for each debug-port
- Active probes at high speed
- Compatible with all PowerDebug units



PowerDebug USB-3

- Entry level system
- Link USB2/USB3



PowerDebug ETH

- Standard System
- Link USB + Eth 10/100 mbps
- Upgradable to PowerTrace



PowerDebug II

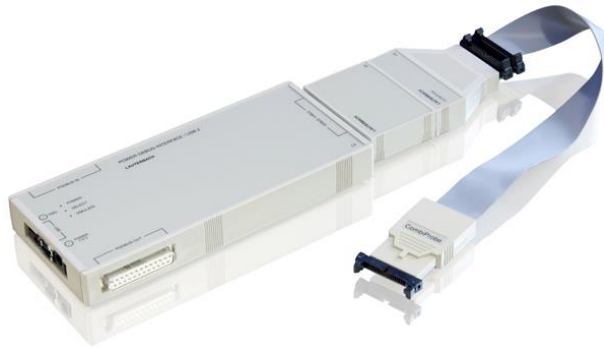
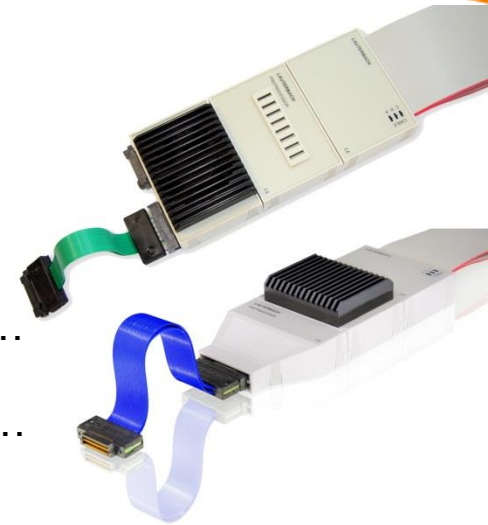
- New generation system
- Link USB + Eth 10/100/1000 mbps
- Upgradable to PowerTrace II

In Circuit Trace

A debug+trace system based on modular **PowerTrace** units that connect debug cables and trace probes specific for different architectures and different **trace-port**

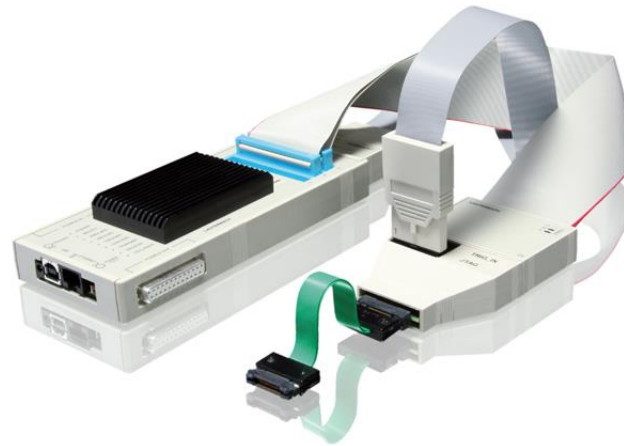
Trace Probes Autofocus

- Parallel trace ETM/NEXUS, ...
- Serial Trace HSTP Aurora, ...



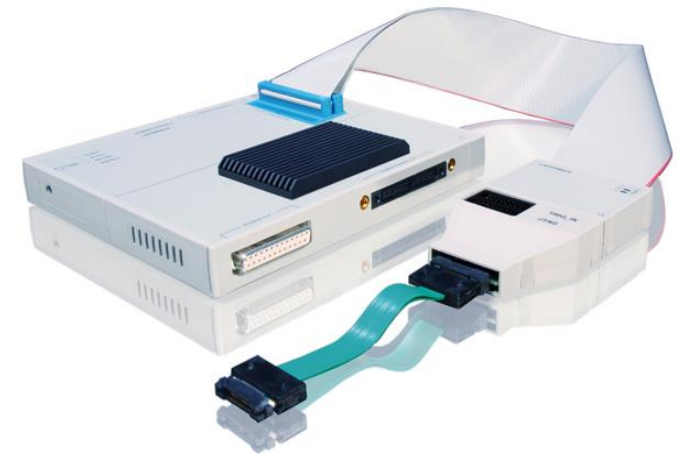
PowerDebug Combiprobe

- Low-cost system
- 128MB trace storage
- 200 Mhz max trace clock
- 1-4 bit trace port



PowerTrace

- First generation system
- 512MB trace storage
- Up to 350 Mhz trace clock



PowerTrace II

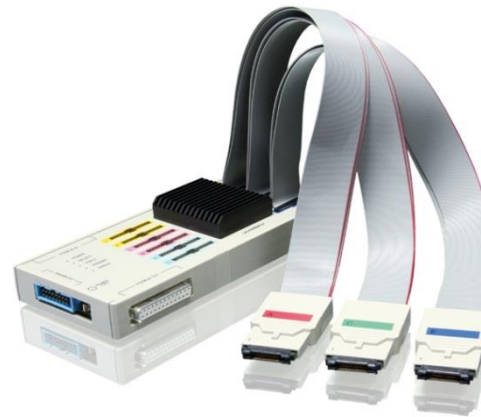
- New generation system
- 1/2/4 GB trace storage
- > GHz trace clock (HSTP)
- Trace Streaming

Logic Analyzers

Any PowerDebug and PowerTrace can be greatly enhanced with the addition of a integrated logic/protocol analyzer: **PowerIntegrator**.

A PowerIntegrator can be used for:

- **I/O timing & trigger**
- **Protocol analyzer**
CAN, FlexRay, LIN, SPI, USB, I2C, Jtag, Seriale, PCI, DigRF, ...
- **Data logger**
- **Energy test**
- **Bus-trace**
for cpu without trace port



PowerIntegrator

- 512 K-Sample
- Max 204 channels
- Max 500Mhz

Probes

- Digital and analog
- For protocols
- For memory bus

PowerIntegrator II

- Max 256000 K-Sample
- Max 102 channels
- Max 500 Mhz
- Stimuli Generator

TRACE32 PowerView for linux/QT

TRACE32 PowerView is available for Windows, MacOS-X, and Linux and Workstations. Is now available a new version of PowerView GUI for linux QT. Both the new QT version and old Motif version are available on TRACE32 software DVD.

Old Motif Gui

New QT GUI

The image displays two versions of the TRACE32 PowerView GUI. The top window, titled 'TRACE32 PowerView for TriCore', is the 'Old Motif Gui' with a grey interface. The bottom window, also titled 'TRACE32 PowerView for TriCore', is the 'New QT GUI' with a blue interface.

The New QT GUI shows the following components:

- Source Code:** A window titled 'B::List.auto' showing assembly code for a 'while' loop.


```

      P: 70001422 1000F00B add d1,d0,d15 ; d1.i.primz
      694
      P: 70001426 DA3C j16 0x7000143A
      {
      696 flags[ k ] = FALSE;
      P: 70001428 F7000091 movh.a a15,#0x7000
      P: 7000142C 1030FFD9 lea a15,[a15]0x70
      {
      696 flags[ k ] = FALSE;
      P: 70001430 F600F101 addsc.a a15,a15,d1,#0x0 ; a15,a15,k,#0
      
```
- Register View:** A window titled 'B::Register.view /sl' showing register values.


```

      D9 0 A9 0
      D10 0 A10 70008FE8
      D11 0 A11 700013E6
      D12 0 A12 0
      D13 0 A13 0
      D14 0 A14 0
      D15 3 A15 70000070
      PSW 0982 PC 70001426
      PCPI 001D013E ISP 700023F0
      FCX 000D013D ICR 0
      LCX 000D0100 BIV 70000000
      BTV 70001D00
      
```
- Trace Statistics:** A window titled 'B::Trace.STAT.TREE' showing a tree view of function calls.

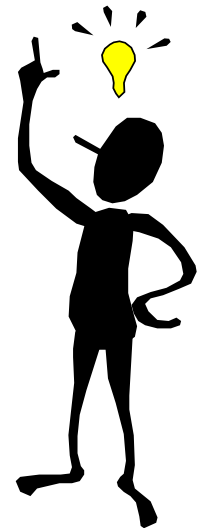
range	tree	total	min	max	avr	count	intern%	1%	2%
(root)	(root)	735.100us	-	735.100us	735.100us	1.	0.000%		
(root)	_START	735.100us	-	735.100us	735.100us	1.	0.027%		
(root)	_init_sp	734.900us	-	734.900us	734.900us	1.	0.136%		
(root)	_start	733.900us	-	733.900us	733.900us	1.	29.152%		
(root)	_c_init	25.200us	25.200us	25.200us	25.200us	1.	0.081%		
(root)	_main	494.400us	-	494.400us	494.400us	1.	0.013%		
(root)	_main	494.300us	-	494.300us	494.300us	1.	10.474%		
(root)	_func2	20.800us	20.800us	20.800us	20.800us	1.	2.203%		
(root)	_func1	4.600us	2.300us	2.300us	2.300us	2.	0.625%		
(root)	_func2a	14.800us	14.800us	14.800us	14.800us	1.	1.686%		
(root)	_func1c	2.400us	2.400us	2.400us	2.400us	1.	0.326%		
(root)	_func2b	14.000us	14.000us	14.000us	14.000us	1.	1.501%		
- Trace Chart:** A window titled 'B::Trace.Chart.symb' showing a Gantt chart of function calls.
- Command List:** A window titled 'B::Trace.Chart.symb' showing a list of commands: RESET, Init, Snapshot, List, AutoArm, AutoInit, SelfArm.

The Old Motif Gui shows a similar interface but with a grey color scheme and a different window layout. It includes a menu bar (File, Edit, View, Var, Break, Run, CPU, Misc, Trace, Perf, Cov, TriCore, Window, Help) and a toolbar with various icons. The source code window is titled 'B::t' and shows assembly code. The register view window is titled 'B::Register.view /sl' and shows register values. The trace statistics window is titled 'B::Trace.STAT.TREE' and shows a tree view of function calls. The trace chart window is titled 'B::Trace.Chart.symb' and shows a Gantt chart of function calls.

Agenda

Seminar and live demo

- Linux debugging: problems & solution
- Debugging all linux components
- Stop-mode & run-mode debugging
- Trace, performance, profiling
- TRACE32 PowerTools
- **Q&A**



To learn more:

Flyers

- ✓ Debug & Trace for ARM
- ✓ Product Overview
- ✓ Linux Flyer *Advanced Debugging and Tracing tools for ARM architectures and Linux kernels*

Web

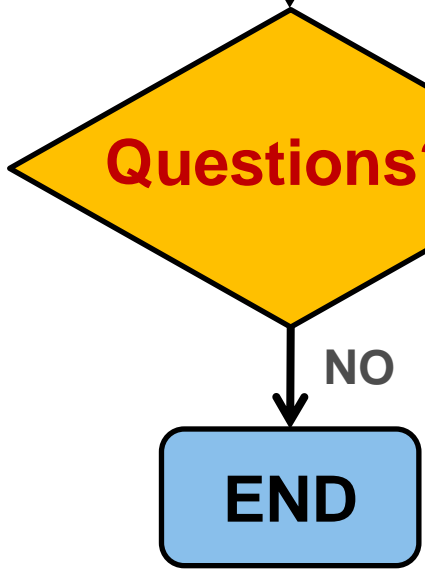
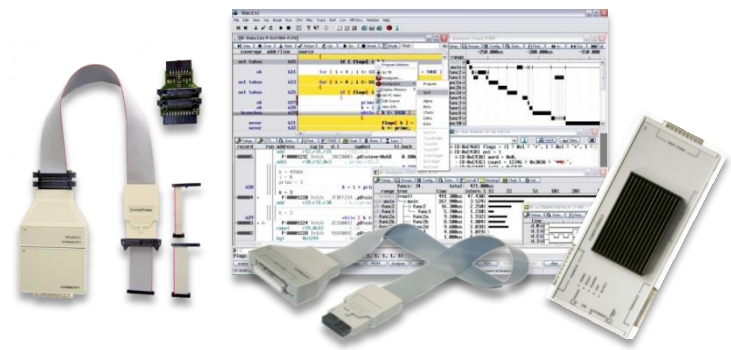
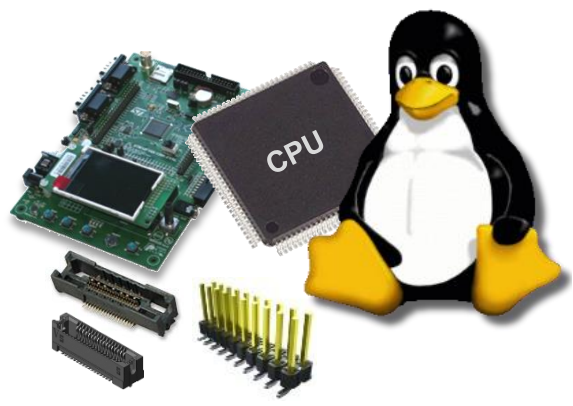
- ✓ Linux Training (training manual) www.lauterbach.com/pdf/training_rtos_linux.pdf
- ✓ RTOS Debugger for Linux (manual) www.lauterbach.com/doc/rtoslinux.pdf
- ✓ TRACE32 Startup Script (repository) www.lauterbach.com/scripts.html
- ✓ Linux Debugging Reference Card www.lauterbach.com/linux_card1_web.pdf



Q&A...

Linux? Kernel?
Target? CPU?
Connector?

Debug? Trace?
PowerDebug?
JTAG?



Answers...



Thank you for participating to the seminar: **the most complete tool for embedded linux debugging**



Maurizio Menegotto
maurizio.menegotto@lauterbach.it

Contact for Italy:

Lauterbach SRL
tel: 02-45490282
web: www.lauterbach.it
email: info_it@lauterbach.it

International contact:

Lauterbach GmbH
web: www.lauterbach.com
email: info@lauterbach.com