**Run Mode Debugging with TRACE32 as GDB Front-end**

使用gdb协议调试。

The TRACE32 software is licensed by a USB dongle.

Requires a gdbserver/gdbstub running on the target.

The TRACE32 GDB Front-end works in so-called **Run Mode Debugging**: At a breakpoint only the selected process is stopped, while the kernel and all other processes continue to run.

**Stop Mode Debugging via JTAG**

Stop Mode Debugging requires a TRACE32 JTAG debugger hardware.

If debugging is performed via the JTAG interface, TRACE32 works in so-called **Stop Mode Debugging**. At a breakpoint the CPU and thus the whole target system is stopped.

T32server 在trace安装目录 demo/arm/etc/t32server/

Gdbserver 在andriod源码 /external/valgrind/corerind/m\_gdbserver/

分析linux kernel相关 ：

B::task.config C:\T32USB\demo\arm64\kernel\linux\linux-3.x\linux3.t32

B::menu.REPROGRAM\_C:\T32USB\demo\arm64\kernel\linux\linux-3.x\linux.men

Task.tesk

Task.check

Sys.config //配置dap寄存器地址

sYmbol.List.MAP //列出加载在debugger中的符号表

Help.Filter.ADD rtoslinux //help文档过滤条件 rtoslinux

加载kernel image或符号表。

Vmlinux中的地址为虚拟地址，在MMU en的情况下，不需偏移；在MMU off的情况下，虚拟地址需要减去物理地址的起始地址。

例如，虚拟地址其实地址0xc0000000，物理地址起始地址0x80000000，通过trace32加载kernel image应该使用如下指令：

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

/nosymbol 不加载符号表

/nocode 不加载image文件

**sYmbol.CLEANUP** 用来清除kernel中的空变量。

Trace32原文：

At this stage, the target MMU is still disabled, so you need to download the kernel code using physical addresses. However, the kernel Elf file “vmlinux”, which is usually used for the kernel, contains virtual addresses. Thus you need to subtract the virtual address base from the physical address base.

if the kernel has the virtual start address 0xC0000000 (which is a typical value) and should be

downloaded on the RAM starting at the physical address 0x80000000, the command to download the kernel would be

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

MMU enable： \_\_enable\_mmu

Trace32 查看地址转换：

Mmu.dump pagetabel 0xffffff8008080000--0xffffff8008090000l (ogical\_addr\_range)

HW MMU table

OS/Target’s translation tabel

Debugger translation tabel

**MMU.PageTable.dump** [<*address>* |*<range*>]

**MMU.PageTable.List** [<*address>* |*<range*>]

**MMU.PageTable.SCAN** [<*address>* |*<range*>]

**dump** Show the current processor/target MMU table.

**List** Show the current debugger MMU table.

**SCAN** Scan the current processor/target MMU table into the debugger MMU table.

保存现场，使用simulater分析。

**What is needed to revise a Linux trace with a TRACE32 instruction set**

**simulator?**

Example for ARM926

SAVING IN TRACE32 DEBUGGER:

*; save trace recording*

*Trace.SAVE trace.ad*

*; save whole RAM address range*

*Data.SAVE.Binary image.bin ASD:<address\_range>*

*; save registers*

*STORE regs.cmm register*

*; save important CP15 registers*

*Data.SAVE.Binary p15.bin C15:0x0--0xfff /Long*

*LOADING IN TRACE32 INSTRUCTION SET SIMULATOR:*

*SYStem.RESet*

*SYStem.CPU ARM926*

*SYStem.Option MMUspaces ON ; enable space ID*

*System.Up*

*; load linux image*

*Data.LOAD.Binary image.bin ASD:address\_range*

*; load registers*

*do regs.cmm*

*Data.LOAD.Binary cp15.bin c15:0*

*; load symbol information*

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

*; specifiy MMU table format*

*MMU.FORMAT LINUX swapper\_pg\_dir <kernel-range> <RAM*

*phys. address>*

*TRANSlation.COMMON 0xc000000--0xfffffff*

*TRANSlation.TableWalk ON*

*TRANSlation.ON*

*; configure Linux awareness*

*TASK.CONFIG linux*

*Menu.ReProgram linux*

*Help.Filter.ADD rtoslinux*

*; load trace recording*

*Trace.LOAD trace.ad*

Linux kernel中一个thread就是一个process。

**To the Linux kernel, there is no concept of a thread.** Linux implements all threads as standard processes.

User application，一个process，可以拥有多个thread。

虚拟内存分为两部分：Kernel space，user space。 Kernel运行在特权模式，可以访问整个范围虚拟空间地址。User space processor，只能访问部分虚拟空间地址。

（这里插两句对虚拟空间和物理空间的理解。虚拟空间的范围和cpu的地址总线相关；物理空间的范围，是DDR器件决定的。）



每一个process都有自己的page table。(kernel也是一个process)。Cpu中没有存储所有process的page table。所以，trace32可以直接看到当前cpu（TTBR0/TTBR1 for ARM）存储了page table的process的 虚拟地址转换物理地址的关系(table walk)。

Linux Kernel控制所有process的MMU切换，kernel的一个结构体变量保存所有process的page table translation。如果，需要查看其他process的table walk。找到这个结构体变量，就可以了。

在linux awareness，TRACE32 commands ***TRANSlation.ScanID*** and

***TRANSlation.ListID***.用来显示running task的page table descriptors。

Space ID (memory space identifier)

Kernel threads space ID都是0。User process 的 space ID是process ID的低16bit。子进程的space ID和父进程相同。

**SYStem.Option MMUSPACES ON**

Stop mode：

**Advantages:**

**• bootstrap, interrupt or post mortem debugging is possible**

**• no software restrictions (like memory protection, ...) apply to the debugger**

**• the full MMU table and code of all processes alive can be made visible**

**• only JTAG is required, no special communication interface as RS232 or Ethernet is needed**

**Disadvantages:**

**• halts the complete CPU, not only the desired process**

**• synchronization and communications to peripherals usually get lost**

**• debug HW and a debug interface on the target are needed**

**Advantages:**

**• halts only the desired process**

**• synchronization and communications to peripherals usually continue**

**• no debugger hardware and no JTAG interface are needed**

**Disadvantages**:

**• no bootstrap, interrupt or post mortem debugging is possible**

**• all software restrictions apply to the debugger too (memory protection, ...)**

**• only the current MMU and code of this scheduled process is visible**

**• actions from GDB change the state of the target (e.g page faults are triggered)**

**• one RS232 or Ethernet interface of the target is blocked**

Trace32 log文件，可以记录包含鼠标操作在内的所有命令。

**LOG.OPEN** *<file>* Create and open a file for the commands to be logged. The default

extension for LOG-files is (**.log**)

**LOG.CLOSE** Close the LOG-file

Store命令，可以保存已使用trace32的所有配置，用来恢复系统设定。

**STOre** *<file>* **SYStem** Create a batch to restore the SYStem settings

**ClipSTOre SYStem** Provide the commands to restore the SYStem settings in the cliptext

History 命令显示，trace32命令行中使用过的命令。

**HISTory.type** Display the command history

Reset命令用来reset trace32全部或者部分功能。

RESet ; reset debugger completely

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

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

AREA.view 显示errors and warnings.

**AREA.view** [<*area*>]

TERM命令用来在trace32中打开一个串口终端

TERM.RESet ; reset old and set new definitions

TERM.METHOD COM com1 115200. 8 NONE 1STOP NONE

; for com10 use \\.\com10

TERM.SIZE 80 1000 ; define capacity of the TERM window

TERM.SCROLL ON ; scrolling follows to the TERM cursor

TERM.Mode VT100 ; or ASCII (default), STRING, HEX ...

WINPOS 50% 0% 50% 100% term\_win ; define next window position and size

TERM.view ; open the TERM window

SCREEN.ALways ; TERM window always updated

也可以使用demo中的脚步： 入参 com port和baud rate。

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

**Displaying the Source Code**

加载符合表后，trace32默认从符号表中找源代码路径。如果，trace32运行的机器不是编译机器，源代码是找不到的。

需要在data.load时，使用**/STRIPPART** and **/SourcePATH**来指定源码。

For example, if you have compiled your kernel on a Linux machine in the directory /home/user/linuxkernel/

linux-3.4 and you are running TRACE32 on a Windows machine where you have the kernel

source files tree under C:\Debugging\Linux\Sources\linux-3.4, you can load the kernel symbols

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

or Data.LOAD.Elf vmlinux /NoCODE /STRIPPART 4. /SOURCEPATH C:\Debugging\Linux\Sources

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

sYmbol.SourcePATH C:\Debugging\Linux\Sources

还可以使用如下命令，替换源码路径

sYmbol.SourcePATH.Translate "/home/user/linux-kernel" "C:\Linux\Linux\Sources\"

具体参考trace32 help文档《training\_rtos\_linux.pdf》 page 26.

Trace32下载kernel image，参考trace32 help文档《training\_rtos\_linux.pdf》 page 28.（包括文件系统image ramdisk.image.gz）

**Set up the Linux Awareness**

; load the awareness for Linux-3.x on ARM

TASK.CONFIG ~~/demo/arm/kernel/linux/linux-3.x/linux3.t32

; load Linux menu:

MENU.ReProgram ~~/demo/arm/kernel/linux/linux-3.x/linux.men

**Mark the Kernel Address Space**

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

GROUP.Create "kernel" 0x0000:0xC0000000--0xFFFFFFFF /RED

**TASK.DMESG**

command to display the kernel log buffer:

**Verifying Image and Symbols**

**TASK.CHECK**

Linux\_banner 字符串变量保存编译的信息。和“cat /proc/version”得到的信息应该时间戳一样。

**Build-in device drivers debug.**

所有的build-in device drivers都注册在initcall table. 老版本linux \_\_initcall\_start，新版本Linux：\_\_initcall0\_start, \_\_initcall1\_start ……

在 **sYmbol.Browse** window 中右键-> other-> view it as a fixed table。

在fixed table中， 右键->format->pointer symbol。

**Trapping Segmentation Violation**

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

Depending on the kernel version, if the mapping of the page fails, the kernel jumps to the label “bad\_area” or “\_\_do\_user\_fault”/”\_\_do\_kernel\_fault”. To trap segmentation violation, we set a breakpoint in our example inside the function “do\_page\_fault”

before “\_\_do\_user\_fault” is called and look to the structure “regs” which contains the complete register set at the location where the fault occurred.

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



**Kernel Module**

Kernel mode在kernel运行时load，link到kernel中。Debug module初始化过程，可以使用kernel awareness，linux 目录中module debugging->debug module in init。也可以

DO ~~/demo/arm/kernel/linux/linux-3.x/mod\_debug.cmm demomod(module name)

the TRACE32 Linux awareness 还可以自动加载module的符号表，linux 目录中symbol autoloader -> set loader script, the autoloader script 在trace32 安装目录(“~~/demo/<arch>/kernel/linux/<linux\_version>/autoload.cmm”).

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

TASK.sYmbol.DELeteMod "demomod" ; erase obsolete module symbols

**Processes**

对一个process，从一开始运行debug。在kernel目录中，process debugging->debug process on.配置需要debug的process名称和需要停止的function名。也可以 DO ~~/demo/arm/kernel/linux/linux-3.x/app\_debug.cmm hello

如果是debug一个已经开始运行的process。

Task.symbol.load 或者 linux目录 process debugging->load symbols. 或者 per mouse click in the TASK.DTask or TASK.Process window. 来加载符号表。

**Symbol Cleanup after Process Termination** sYmbol.Delete \\hello ; get rid of invalid symbols

**Process Watch System**

Linux 目录，process debugging->watch processes. 添加process到 task.watch窗口。添加后，可以跟踪指定process的状态，自动添加符号表，自动更新MMU table。

**Threads**

Threads在linux中共享process的同一块内存。一个process可以有多个thread。Kernel也是一个process，特权模式下的process。

**Libraries**

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

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

**Linux specific Windows and Features**

**Display of System Resources**

**TASK.PS** displays the process table similar to the output of the “ps” shell command. **TASK.DTask** give you more detailed information.

**TASK.Process** displays the processes with their threads.

**Task Related Single Stepping**

单步调试多个task共享的代码，可能遇到，单步后，切换到其他task的情况。可使用SETUP.StepWithinTask ON 解决。

**Task Context Display**

显示一个不在运行态的task相关的memory，registers。显示一个运行态的task的frame。

List /TASK "mytask"

Register /TASK "mytask"

Var.Frame /TASK "mytask"

这里显示的信息是相应的task的信息。并不是当前的registers。

也可以在task.dtask窗口中，右键对应的task，switch context来切换不同task的环境信息。

**Linux Trace**

具体参考trace32 help文档《training\_rtos\_linux.pdf》 page 55.

Reference documents:

Trace32 help document:Training\_rtos\_linux.pdf rtos\_linux\_top.pdf rtos\_linux\_run.pdf

[www.wowotech.net/armv8a\_arch/turn-on-mmu.html](http://www.wowotech.net/armv8a_arch/turn-on-mmu.html)