Ronetix Application Note AN008: Debugging the ARM Linux Kernel and Applications

1. Introduction

The increasing popularity of high-speed 32-bit ARM based microcontrollers allowed Linux to enter the world of embedded devices. That is why the need of debugging its kernel and applications became essential. Since Linux is a true multi-process operating system, it utilizes a Memory Management Unit (MMU) to give each process a separate memory space. The MMU is also responsible for the protection of each process's memory space against each other.The switching among different processes complicates the debugging, so here I will show you how to debug Linux kernel and applications without interfere processes not been debuggers.

What I use

For the purpose of this application note I will use ARM-ELF and ARM-LINUX GNU toolchains and the Ronetix PM9261. The Ronetix board comes with Linux installed.

2. Debugging the kernel

2.1 Kernel building

2.1.1 Installing the GNU toolchain

```
wget https://download.ronetix.at/toolchains/arm/ronetix-gnutools-arm-elf-4.1.1-linux.tar.bz2
cd /
tar xvfj ronetix-gnutools-arm-elf-4.1.1-linux.tar.bz2
export PATH=/usr/cross/arm-elf/bin:$PATH
export ARCH=arm
export CROSS_COMPILE=arm-elf-
```

2.1.2 Installing and compiling the Linux kernel

```
cd
mkdir ronetix
cd ronetix
wget https://download.ronetix.at/boards/linux/kernel/2.6.23.14/linux-2.6.23.14.tar.bz2
wget https://download.ronetix.at/boards/linux/kernel/2.6.23.14/2.6.23-at91.patch
wget https://download.ronetix.at/boards/linux/kernel/2.6.23.14/linux-2.6.23.14-ronetix-08-04-29.2136.patch
tar xvfj linux-2.6.23.tar.bz2
cd linux-2.6.23.14
patch -p1 < ../2.6.23-at91.patch
patch -p1 < ../linux-2.6.23-ronetix.patch
make pm9261_defconfig
make menuconfig
```

– go to "Kernel hacking" and enable the "Kernel debugging" option

```
make clean
make modules
```

– generate vmImage:

```
mkdir image
cd image
arm-elf-objcopy -O binary -S ../vmlinux vmlinux.bin
gzip -f9 vmlinux.bin
mkimage -A arm -O linux -T kernel -C gzip -a 0x20008000 -e 0x20008000 -n "PM9261 Linux Kernel On
ly Image" -d vmlinux.bin.gz vmImage
rm vmlinux.bin.gz
cd ..
```

The binary file for programming is: image/vmImage

2.1.3 Installing the Kernel modules

```
cd
cd ronetix
wget https://download.ronetix.at/boards/linux/rootfs/2.6.23.14-pm9261/rootfs-pm926x-openzaurus-2
0081103.tar.bz2
su -
tar xvfj rootfs-pm926x-openzaurus.tar.bz2
cd linux-2.6.23.14
make INSTALL_MOD_PATH=../openzaurus-gpe-install-ronetix modules_install
```

2.1.4 Generating the ROOTFS image

```
cd
cd ronetix
mkfs.jffs2 -v -n -d openzaurus-gpe-install-ronetix --little-endian --pad --eraseblock=131072 --p
agesize=2048 -o rootfs.jffs2
The binary file for programming is: rootfs.jffs2
```

```
2.2 Programming of U-BOOT, Kernel and ROOTFS
```

Copy the just compiled file images vmImage and rootfs.jffs2 as well as the bootloader (https://download.ronetix.at/boards/linux/bootloader/u-boot-2008.10/u-boot-pm9261.bin) to your TFTP or FTP root directory and correct the paths in the PEEDI's configuration file.
Open a serial or telnet console to PEEDI:

```
telnet 192.168.3.100 ; assuming the IP address of PEEDI is 192.168.3.100

peedi> flash set 0 ; select the first flash profile (U-BOOT)
peedi> flash erase
peedi> flash program

peedi> flash set 1 ; select the second flash profile (Kernel)
peedi> flash program

peedi> flash set 2 ; select the ROOTFS flash profile
peedi> flash erase
peedi> flash program
```

Instead of the above commands, all images can be programmed at once using a script:

```
peedi> run $prog
```

2.3 Kernel debugging

The board should be programmed with the u-boot, the binary version of the vmlinux and the rootfs.
Start gdb/insight:

```
arm-elf-insight vmlinux
```

In the gdb console window:

```
(gdb) target remote 192.168.3.100:2000
(gdb) set $pc = 0x10000040 ; assuming the u-boot is flashed at 0x10000000
(gdb) c
```

The board is running: first starts the u-boot, then the Linux kernel. If you have a serial console you can see that the Linux is working. Now you can halt the kernel and set breakpoints where you want.

If you want to have a breakpoint before the kernel is running, it is not a good idea to use the halt command because we may halt in a user process, so it is better to set a break point somewhere inside the kernel, for example we can set a break point in the **start_kernel()** function of the **main.c**. The right address can be found using the **nm** utility.

```
(gdb) target remote 192.168.3.100:2000
(gdb) set $pc = 0x10000040 ; assuming the u-boot is flashed at 0x10000000
(gdb) hbreak start_kernel ; hardware breakpoint is used because the MMU is still not active
(gdb) c
```

The board is running (u-boot, then the kernel) and will stop at the start_kernel function.

```
(gdb) delete ; delete the hardware breakpoint
```

Now you can use the software breakpoints. We can put break and watch points anywhere we want the target to break and debug it and start it again.
If some break is hit the target will stop and gdb will show the corresponding source file. You can step-in step-over function calls, add/remove breaks, watch variables, examine target's memory and so on.

3. Debugging applications

Debugging Linux applications is similar to debugging the kernel with some characteristics.
Here is the "Hello world" like application we will debug:

```
#include <stdio.h>

int main()
{
    printf( "Halting the target...\r\n" );
    asm( ".long 0xDFFFDFFF" ); // halt the target and let us put breaks
    printf( "Entering eternal loop...\r\n" );
```

```
    while ( 1 )
    {
        sleep( 1 );
        printf( "tick\r\n" );
    }
    return 0;
}
```

Where 0xDFFFDFFF is the value specified for the CORE_BREAK_PATTERN parameter in the PEEDI target configuration file and break add -1 command must be added in the INIT section, to set the ARM debug registers to break on the pattern.

I will compile it using the -g option which will include the debug information that gdb needs:

```
arm-linux-gcc -g –O0 main.c -o main.elf
```

Now we have to start the Linux on the target and wait till it displays the Linux login prompt. Login and start the application.

It will show a single line saying “Halting the target…” and will do at is says. This exactly is the point of the break pattern on the second line of the main() function. I will put some more light here:

Usually the Linux kernel is executed from higher addresses than user app and its address space does not overlap any other process address space. In other words there is single and only one virtual address space that uses those memory address ranges and it belongs to the kernel. This allows us to use hardware break and watch point and guarantees us that only the kernel will hit them.

Unlike the kernel, user applications use same virtual addresses (which are translated to different physical ones by the MMU). This means that if we set a hardware break or watch points any user process may hit it. That is why when debugging user applications only software break points must be used. This way they are dedicated to the process been debugged and it is guaranteed no other process will hit them. As a consequence, asynchronous stop of target must be avoided, because there is no guarantee that the CPU will stop when executing the debugged process. This means that the halt PEEDI command, CTRL-C in gdb or the stop button of insight must not be used. Instead software break points may be set where the debugged process have to be stopped. Here I need to mention that you have to set the CORE_BREAKMODE parameter in the target configuration file of PEEDI to SOFT.

Now you understand why I used hardware breakpoint to halt the kernel, but software breakpoint pattern in the application source to stop the user process.

So after the process is halted (and the whole target) I will see the current PC value, increment it by 4 (2 in case THUMB code is debugged) and set it back to the PC. This is done only to skip the bkpt instruction:

```
peedi> info target
CORE0 -> ARM9 - stopped by breakpoint (ARM9)
PC=0x000083B8, CPSR=0x60000010
peedi> set pc 0x83BC
```

Here we can start gdb on the host:

```
arm-elf-insight main.elf
```

Then connect to PEEDI:

```
(gdb) target remote 192.168.1.100:2000
```

Now we will just make a single step for gdb to refresh the target state:

```
(gdb) si
```

From now on we can debug as we debugged the kernel, i.e. set some software break points and start the process:

```
(gdb) continue
```

After it hits a break we can step-by-step, step-in, step-over a function calls, examine the memory and so on.

4. Conclusion

Debugging Linux kernel and applications may look hard at first sight, but it gets easy once you have tried it.

In this application note I have showed debugging Linux kernel and applications running on ARM targets.

Share on LinkedinShare on FacebookShare on Reddit on TwitterShare Via EmailShare on Whatsapp

Share