Application note

Debugging Linux kernel and applications

Version 1.0

RONETIX
embedded development tools

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 debugees.

What I use

For the purpose of this application note I will use ARM-ELF and ARM-LINUX GNU toolchains and the KaRo Triton starter-kit 2 equipped with a Triton LP module. The Triton board comes with installed Linux on it.

2 Setting PEEDI

Apart all the common things that must be set in the target configuration file, there are two XScale specific parameters that must be set. The first is the address of the debug handler. The debug handler is a 2KB debug monitor like program that is downloaded to the CPU's mini instruction cache at a defined virtual address. This address is chosen so that no user instruction code is overwritten which guarantees no instruction fetches will be made from the CPU while executing user code. There is another limitation of the debug handler address – the fact that PEEDI uses a branch instruction to override the CPU reset vector, which allows has branch range of +/- 32MB and exception vectors may reside on either 0x0000_0000 or 0xFFFF_0000. So finally the debug address may have values from 0x0000_0000 to 0x01FF_FC00 and from 0xFE00_0000 to 0xFFFF_FC00 aligned to a 1KB(0x400) boundary and not overlapping any user code. I personally choose the address at 0xFFFFF800 which is the last 2KB of the CPU memory space.

After downloading the debug handler PEEDI needs to override the CPU's reset vector to point the debug handler's entry. Because the mini instruction cache is 32 bytes long PEEDI cannot override only the reset vector but will have to override all eight vectors. That is why PEEDI must be aware of the applications vectors all the time. Here is the second XScale specific thing to set – tell PEEDI how to manage exception vectors. There are two options – to set constant values for the vectors or tell PEEDI to "refresh" vectors each time a debug event occurs. Every vector may have its own behavior set. So if we choose constant vector(s) we have to set the corresponding CORE_VECTOR_XXX parameter to a value which represents a valid ARM instruction that resides in the user's code corresponding vector. For example 0xE59FF018 stands for the "LDR pc, [pc, #18]" instruction, which is very common for an exception vector. My personal choice is the second option - tell PEEDI to get the vectors from the targets memory every time the CPU enters the debug handler. To do that I will set all CORE_VECTOR_XXX parameters to AUTO. This technique works fine even the vectors are filled by PEEDI itself during application code download. There is a situation when PEEDI cannot handle vectors automatically – when they are set by the application at runtime. To assist PEEDI in that horrible moment you have to set a breakpoint in the user code just after the vectors are set and before interrupts are enabled. You can do this in three ways:

1. Set 32 bit write access watch point at the last modified by the user code vector.
2. Set hardware breakpoint to a point of the code where the vectors have been set but not yet enabled.
3. In the source code, add a software break "*asm("bkpt 0");*", where the vectors have been set but not yet enabled.

After the target has stopped you can start it again immediately. The whole process can be automated easily in the INIT section of the core like this:

```
[INIT_XSCALE]
break add watch 0xffff001C w 32     ; set watchpoint on FIQ vector
go                                  ; start target
wait 30000 stop                     ; wait to break
go                                  ; start again with updated vectors
```

There is one more thing we need to take care of before starting actual debugging – make sure no user code will destroy the debug handler. Especially the Linux port

provided with the Triton board destroys the handler in to places during boot, to prevent this:

- Replace "**MCR p15, 0, rd, c7, c5, 1**" with "**MCR p15, 0, rd, c7, c5**" anywhere in the sources
- Disable **CONFIG_XSCALE_CACHE_ERRATA** (Workaround for XScale cache errata) option when building Linux kernel.

Now we are ready to plug the cables and verify what we have done so far. If everything is correct after you connect PEEDI to the Triton board you should see first booting RedBoot, after that Linux and shortly the Linux login prompt. Now you can type some letters to see the Linux is working normally, then you can issue the PEEDI CLI **halt** command to stop the target, try to type some more letters – nothing should appear. Now issue **go** command and the letters you have previously type should appear. So we have control over the target!

3 Debugging the kernel

To debug the kernel we will need an ELF image of it containing debug information i.e. compiled with the GCC **–g** option and this image must be FLASHed on the target. First we need to do is to stop the target while executing kernel code. It is not a good idea to use the **halt** command because we may halt in user process, so it is better to set a break point somewhere inside the kernel or even remove the last to lines from the INIT section I have showed you previously. Now if we restart the target it will break after the vectors are set by the kernel boot code. Here we can start gdb/insight on the host loading the kernel ELF:

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

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
```

You will see that that the execution has stopped in the __trap_init() function which sets the exception vectors, it is showed in assembler code because it is implemented in the entry-armv.S assembler source file. To debug some C source, we can for example set a break point in the start_kernel() function of the main.c file where the time_init() function is called and then start the target:

```
(gdb) continue
```

After s second you will see target stopped and gdb showing the main.c source file with highlighted line where time_init() function gets called.

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.

4 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( "bkpt 0" );        // halt the target and let us put breaks
      printf( "Entering eternal loop...\r\n" );
      while ( 1 )
      {
            sleep( 1 );
            printf( "tick\r\n" );
      }
      return 0;
}
```

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

```
arm-linux-gcc -g 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 as user root and password root and these are default for the Triton-LP port provided from KaRo.

The kernel creates a RAM file system where we will download the application’s ELF file we will debug (of course a TFTP server must be running on the host):

```
root@triton1:~# tftp -g -r main.elf -l /tmp/main.elf 192.168.1.1
```

Linux may complain about permissions, so let’s calm it down:

```
root@triton1:~# chmod 777 /tmp/main.elf
```

Now we are ready to launch our application:

```
root@triton1:~# /tmp/main.elf
```

It will show a single line saying "Halting the target..." and will do at is says. This exactly is the point of the “bkpt” assembler instruction 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 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 mean 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 watch point to halt the kernel, but software breakpoint instruction 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 -> XScale - stopped by breakpoint (XSCALE)
         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.

5 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 XScale targets, but this applies to all ARM targets if you ignore the XScale specifics. For example the *asm ("bkpt 0")* must be replaced with *asm (".long BREAK_PATTERN")*, where BREAK_PATTERN is the value specified for the CORE_BREAK_PATTERN parameter in the PEEDI target configuration file.