



Kernel source code

Objective: Get familiar with the kernel source code.

After this lab, you will be able to

- Explore the sources in search for files, function headers or other kinds of information...
- Browse the kernel sources with tools like cscope and LXR.

Setup

Go to the `/home/<user>/felabs/linux/sources` directory.

Download and extract the latest 2.6.30 kernel sources.

Get familiar with the sources

As a Linux kernel user, you will very often need to find which file implements a given function. So, it is useful to be familiar with exploring the kernel sources.

1. Find the Linux logo image in the sources
2. Find who the maintainer of the 3C505 network driver is.
3. Find the declaration of the `platform_device_register()` function.

Use a kernel source indexing tool

Now that you know how to do things in a manual way, let's use more automated tools.

Try LXR (Linux Cross Reference) at <http://lxr.free-electrons.com> and choose the Linux version closest to yours.

If you don't have Internet access, you can use cscope or Kscope instead.

As in the previous section, use this tool to find where the `platform_device_register()` is declared, implemented and even used.

You may look for all files with `logo` in their name.

Of course, if your kernel has a significant amount of custom code, or if you are not always connected to the Internet, you can run LXR on your own computer.







Kernel - Native compiling

Objective: compile a recent kernel for your GNU/Linux PC

After this lab, you will be able to

- Rebuild the Linux kernel running on your GNU/Linux PC
- Remove some kernel features you don't need.
- Update the bootloader configuration
- Update the initramfs filesystem your distribution boots on.

Setup

Stay in the `/home/<user>/felabs/linux/sources` directory.

Download and apply the 2.6.30 patches to your 2.6.29 kernel sources.

Why recompiling your kernel?

Ordinary users should never need to recompile their kernel. However, as a new kernel developer, you can have good reasons for doing this:

- If your hardware requires a new driver that is only available in a recent kernel.
- But also to test recent kernel releases and features, and then be able to report bugs (appreciated contributions).
- To develop drivers that are architecture independent. A typical example is USB device drivers, which are much easier to develop and test on a PC workstation.

Kernel configuration

Identify the version of the kernel currently running on your workstation.

Copy the corresponding file to the `.config` file in the root directory of the kernel sources. Now, update this configuration according to the 2.6.30 sources:

```
make oldconfig
```

Answer the questions corresponding to the new parameters in the new kernel sources. See what the new features are. If you are not sure whether you need a particular feature or not, accept the default settings (highlighted in upper case). If you are sure you don't need a feature, answer `n`.

Keep a copy of this initial configuration.

As explained in the lectures, add a specific suffix to the version name of the kernel you will compile.

Simplifying the configuration

Compiling a Linux kernel for a GNU/Linux distribution can take up to several hours, according to the speed of your workstation.

Edit the current configuration and remove features and drivers that

This will allow you do make the difference between multiple kernels you will build from the same sources, and with kernel binaries supplied by others.

This is because distributions usually support most features and devices that Linux can support.



you are sure you won't need on your system, such as:

- Drivers for other PC manufacturers
- Drivers for hardware you don't have and for protocols you don't need: amateur radio, isdn, floppy disks, filesystems for other Unix filesystems...

Unselecting all these features can take a significant amount of time, but will also save a lot of compiling time. Exploring the kernel configuration interface is also a nice opportunity to learn about kernel features.

If you have any doubt about a particular feature, don't disable it. Today's distributions rely on advanced kernel features, and your distro may use one that you may not be aware of.

Compile your kernel

Caution: in some distros, you may have specific tools to recompile and update a kernel in a way that is consistent with the distribution rules, in particular by generating new kernel packages that can be installed and managed like official ones.

In our labs, we recommend to use Ubuntu Linux (based on Debian), but we are not trying to teach a Ubuntu or Debian course. Therefore, the instructions that follow try to be generic and should work with most GNU/Linux distributions.

Compile your kernel, making sure you're running a sufficient number of parallel jobs.

Install your new kernel and its modules

To copy your kernel to /boot:

```
sudo make install
```

To copy the newly compiled kernel modules to where the modprobe command will look for them (/lib/modules/<version>):

```
sudo make modules_install
```

Update your distribution initramfs

Each distribution can have its own way of booting the system. They all boot on an initramfs containing special scripts and kernel modules needed to access the full root filesystem where the distribution is installed.

You could start from the `initrd` image of the current kernel version, uncompress and extract this compressed cpio archive, and update the modules it contains by hand.

At least in Ubuntu, you could run the `mkinitramfs` utility to do this automatically:

```
sudo mkinitramfs -o /boot/initrd.img-<version> <version>
```

If you're using another distribution, you should find a similar utility.

Update your bootloader configuration

Edit the `/boot/grub/menu.lst` file which is the configuration file for the GRUB bootloader. Add new lines describing the kernel



version that you have just compiled.

Testing your new kernel

Now, you're ready to test your new kernel. Just reboot, and in the GRUB boot menu, choose the version that you've just compiled.

If everything works as expected, you can even edit GRUB's configuration file to make the new version the one that is booted by default.

Otherwise, show your trouble to your instructor. He will be happy to help you to understand your trouble and fix your kernel configuration.





Kernel – Module development environment

Objective: Setup an NFS based kernel module development environment.

After this lab, you will be able to

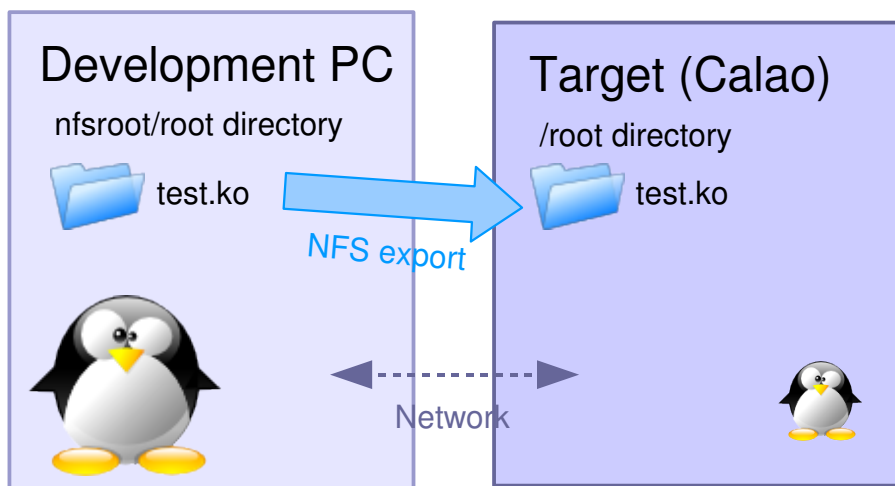
- Boot an ARM kernel using an NFS root filesystem, which is somewhere on your development workstation.
- Compile and test standalone kernel modules, which code is outside of the main Linux sources.
- Develop a simple kernel module.

Lab implementation

While developing a kernel module, the developer wants to change the source code, compile and test the new kernel module very frequently. While writing and compiling the kernel module is done the development workstation, the test of the kernel module usually has to be done on the target, since it might interact with hardware specific to the target.

However, flashing the root filesystem on the target for every test is time-consuming and would use the flash chip needlessly.

Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed through the network by the target, using NFS.



Setup

Go to the `/home/<user>/felabs/linux/modules` directory.

Get the sources from the latest 2.6.29.x release and place them in the current directory.

Cross-compiling toolchain setup

Download the `arm-unknown-linux-uclibc-gnueabi-4.3.2.tar.bz2` toolchain from <http://free-electrons.com/labs/tools/>

NFS root filesystems are particularly useful to compile modules on your host, and make them directly visible on the target. You no longer have to update the root filesystem by hand and transfer it to the target (requiring a shutdown and reboot).



and extract the archive in the / directory:

Now add the path of your cross-compiler to the Unix PATH:
`export PATH=/usr/local/xtools/arm-unknown-linux-
uclibcgnueabi/bin/:$PATH`

Kernel configuration

Configure this kernel with the ready-made configuration for the Calao USB-A9263 board. If needed, add the configuration options that enable booting on a root filesystem.

Compile your kernel and generate the uImage kernel image suitable for U-Boot.

Setting up the NFS server

Install the NFS server by installing the `nfs-kernel-server` package. Once installed, edit the `/etc/exports` file as root to add the following lines, assuming that the IP address of your board will be `192.168.0.100`:

```
/home/<user>/felabs/linux/modules/nfsroot  
192.168.0.100(rw,no_root_squash,no_subtree_check)
```

```
/home/<user>/felabs/linux/character/nfsroot  
192.168.0.100(rw,no_root_squash,no_subtree_check)
```

```
/home/<user>/felabs/linux/debugging/nfsroot  
192.168.0.100(rw,no_root_squash,no_subtree_check)
```

The path and the options must be on the same line!

Then, restart the NFS server:

```
sudo /etc/init.d/nfs-kernel-server restart
```

Setting up serial communication with the board

Plug the Calao board on your computer. Start Minicom on `/dev/ttyUSB1`, which is the port the CALAO board's console is connected to.

You should now see the U-Boot prompt:

```
USB-A9263>
```

You may need to reset the board (using the tiny reset button close to the USB host connectors).

You can now use U-Boot. Run the `help` command to see the available commands.

Setting up Ethernet communication

The kernel image will be transferred to the board using the TFTP protocol, which works on top of an Ethernet connection.

To start with, install a TFTP server on your development workstation:

```
sudo apt-get install tftpd-hpa
```

Set `RUN_DAEMON="yes"` in the `/etc/default/tftpd-hpa` file.

The U-boot bootloader needs the kernel zImage file to be encapsulated in a special container. You can have the kernel Makefile generate this container for you in a uImage file:
`make uImage`



Now restart your TFTP server:

```
sudo /etc/init.d/tftpd-hpa restart
```

Copy your uImage file to the `/var/lib/tftpboot` directory.

Plug the USB Ethernet adapter provided by the instructor and plug it to the Calao board through the Ethernet cable. A new network interface, probably `eth1` or `eth2`, should appear on your Linux system. With the Network Manager tasklet on your desktop, configure this network interface to use a static IP address, like `192.168.0.1` (of course, make sure that this address belongs to a separate network segment from the one of the main company network).

Now, configure the network on the board in U-Boot

```
setenv ipaddr 192.168.0.100
setenv serverip 192.168.0.1
```

To make these settings permanent, save the environment:

```
saveenv
```

You can then test the TFTP connection. First, put a small text file in the directory exported through TFTP on your development workstation. Then, from U-Boot, do:

```
tftp 0x21000000 textfile.txt
```

This should download the file `textfile.txt` from your development workstation into the board's memory at location `0x21000000` (this location is part of the board DRAM). You can verify that the download was successful by dumping the contents of the memory:

```
md 0x21000000
```

Boot the system

First, boot the board to the U-Boot prompt.

Before booting the kernel, we need to tell it that the root filesystem should be mounted over NFS, by setting some kernel parameters.

First, in Minicom, type `[Ctrl] [a] [w]` to enable line wrapping. This will allow you to type long lines like the one that follows.

Use the following U-Boot command to do so (in just 1 line):

```
setenv bootargs root=/dev/nfs ip=192.168.0.100 nfsroot=192.168.0.1:
/home/<user>/felabs/linux/modules/nfsroot
```

Of course, you need to adapt the IP addresses to your exact network setup. Save the environment variables (with `saveenv`).

Now, download the kernel image through tftp:

```
tftp 0x21000000 uImage
```

Now, boot your kernel:

```
bootm 0x21000000
```

If everything goes right, you should reach a shell prompt. Otherwise, check your setup or ask your instructor for clarifications.





Kernel – Writing modules

Objective: create a simple kernel module

After this lab, you will be able to

- Write a kernel module with several capabilities, including module parameters.
- Access kernel internals from your module.
- Setup the environment to compile it
- Create a kernel patch

Setup

Stay inside the `/home/<user>/felabs/linux/modules` directory.

Boot your board again, as you did in the previous lab.

Writing a module

Go to the `nfsroot/root` directory. All the files you generate there will also be visible from the target. That's great to load modules!

Create a `hello_version.c` file implementing a module which displays this kind of message when loaded:

```
Hello Master. You are currently using Linux <version>.
```

... and displays a goodbye message when unloaded.

Caution: you must use a kernel variable or function to get version information, and not just the value of a C macro. Otherwise, you will only get the version of the kernel you used to build the module.

Building your module

The current directory contains a `Makefile` file, which lets you build modules outside a kernel source tree.

Compile your module.

Testing your module

Load your new module file. Check that it works as expected. Until this, unload it, modify its code, compile and load it again as many times as needed.

Run a command to check that your module is on the list of loaded modules. Now, try to get the list of loaded modules with only the `cat` command.

Adding a parameter to your module

Add a `who` parameter to your module. Your module will say “Hello `<who>`” instead of “Hello Master”.

Compile and test your module by checking that it takes the `who` parameter into account when you load it.

You may just start with a module that displays a hello message, and add version information later.

Suggestion: you can use kernel sources from the previous labs to look for files which contain `version` in their name, and see what they do.

Actually, you don't need complete kernel sources to build a module. A build directory is enough.



Adding time information

Improve your module, so that when you unload it, it tells you how many seconds elapsed since you loaded it.

You can use the `do_gettimeofday()` function to achieve this.

Adding the `hello_version` sources to the kernel sources

Add your module sources to the `drivers/misc/` directory in your kernel sources. Of course, also modify kernel configuration and building files accordingly, so that you can select your module in `make xconfig` and have it compiled by the `make` command.

Configure your kernel with the config file corresponding to your running kernel. Now check that the configuration interface shows your new driver and lets you configure it as a module.

Run the `make` command and make sure that the code of your new driver is getting compiled. Then, install your kernel module using `make modules_install`. Beware, the modules should be installed in the root filesystem of the target, not in the root filesystem of your development workstation!

Create a kernel patch

You can be proud of your new module! To be able to share it with others, create a patch which adds your new files to the mainstream kernel.

Test that your patch file is compatible with the `patch` command by applying it to unmodified kernel sources.

Good job!

You may search for other drivers in the kernel sources using the `do_gettimeofday()` function. Looking for other examples always helps!

You may also wait for compiling to be over and look for a new `hello_version.ko` file, but this may take much more time.



Kernel - Output-only character driver

Objective: understanding the basics of managing a /dev entry

After this lab, you will be able to

- Write a simple character driver
- Have the kernel allocate a free major number for you, and create a device file accordingly
- Write simple `file_operations` functions for a device, including `ioctl` controls.
- Copy data from user memory space to kernel memory space and vice-versa.
- You will practice kernel standard error codes a little bit too.

As in the *Module development environment* lab, we will use a the Calao board booted from NFS.

Setup

Go to the `/home/<user>/felabs/linux/character` directory. Compile a recent Linux 2.6.X kernel for the Calao board.

If you choose a kernel `< 2.6.31`, apply the patch available in the `data/` directory. It will add support in the network driver for *netconsole*, a mechanism that allows to get console messages over the network. This will be useful since we won't have the serial messages anymore!

Then configure your kernel with:

- root over NFS support
- loadable module support
- netconsole support (in Device Drivers → Network device support → Network console logging support)

and of course without the serial port driver (in Device Drivers → Character Drivers → Serial drivers → AT91 / AT32 on-chip serial port support), and reflash the kernel.

You also need to adapt the kernel parameters (`bootargs` environment variable) so that:

- It loads the root filesystem over NFS from `/home/<user>/felabs/linux/character/nfsroot`.
- It configures netconsole properly. To do, add the following kernel argument `netconsole=4444@192.168.0.TARGET/eth0,5555@192.168.0.HOST/`

As we are going to re-implement the serial driver of the Calao board, we won't be able to use the serial console to communicate with the board. We will instead use an SSH connection. Therefore, the Dropbear SSH server is already installed on your target filesystem. To connect to the Calao board, use:

```
ssh root@IPADDRESS
```

The root password is empty, just press Enter.

As in the previous lab, look at the lab directory for files that you can reuse.



To get the console output, we will use netcat on the host to listen for UDP packets on port 5555:

```
netcat -u -l -p 5555
```

Now go to the `nfsroot/root` directory, which will contain the kernel module source code.

You have a ready-made `Makefile` to compile your module, and a template `serial.c` module which you will fill with your own code. Modify the `Makefile` so that it points to your kernel sources, the ones you compiled in the previous lab.

Major number registration

Start the system on the Calao board. Find an available character device major number on your virtual system.

Modify the `serial.c` file to register this major number in your new driver. Compile your module, load it, and check that it effectively registered the major number you chose.

Simplistic character driver

Now, add code to register a character driver with your major number, and the empty file operation functions which already exist in `serial.c`. Also create the corresponding `/dev/serial` device file.

In the module initialization function, remap the I/O memory region starting at address (`AT91_BASE_SYS + AT91_DBGU`), for a size of `SZ_512` (512 bytes). The `AT91_*` constants are already defined in Linux kernel headers.

Then, implement the write operation by writing character by character to the serial port using the following steps:

1. Wait until the `ATMEL_US_TXRDY` bit gets set in the `ATMEL_US_CSR` register (`ATMEL_US_CSR` is an offset in the I/O memory region previously remapped). You can busy-wait for this condition to happen. In the busy-wait loop, you can call the `cpu_relax()` kernel function to relax the CPU during the wait.
2. Write the character code to the `ATMEL_US_THR` register.

Once done, compile and load your module. Test that your write function works properly by using:

```
echo "test" > /dev/serial
```

The `"test"` string should appear on the remote side (i.e in the Minicom connected to `/dev/ttyUSB1`).

You'll quickly discover that newlines do not work properly. To fix this, when the userspace application sends `"\n"`, you must send `"\r\n"` to the serial port.

ioctl operation

We would like to maintain a counter of the number of characters written through the serial port. So we need to implement two `ioctl()` operations:



- `SERIAL_GET_COUNTER`, which will return in a variable passed by address the current value of the counter;
- `SERIAL_RESET_COUNTER`, which as its name says, will reset the counter to zero.

Two already-compiled test applications are already available in the `nfsroot/root/` directory, with their source code. They assume that `SERIAL_GET_COUNTER` is `ioctl` operation 0 and that `SERIAL_RESET_COUNTER` is `ioctl` operation 1.



Kernel - Sleeping and handling interrupts

Objective: learn how to register and implement a simple interrupt handler, and how to put a process to sleep and wake it up at a later point

During this lab, you will

- Register an interrupt handler for the serial controller of the Calao board
- See how Linux handles shared interrupt lines
- Implement the `read()` operation of the serial port driver to put the process to sleep when no data is available
- Implement the interrupt handler to wake-up the sleeping process waiting for received characters
- Handle communication between the interrupt handler and the `read()` operation.

Setup

This lab is a continuation of the *Output-only character driver* lab, so we'll re-use the code in `/home/<user>/felabs/linux/character`. Your Calao board should boot over NFS and mount `/home/<user>/felabs/linux/character/nfsroot/` as the root filesystem.

Register the handler

First, declare an interrupt handler function. Then, in the module initialization function, register this handler to IRQ number `AT91_ID_SYS`. Note that this IRQ is shared, so the appropriate flags must be passed at registration time.

Then, in the interrupt handler, just print a message and return `IRQ_NONE` (to tell the kernel that we haven't handled the interrupt).

Compile and load your module. Look at the kernel logs: they are full of our message indicating that interrupts are occurring, even if we are not receiving from the serial port! It shows you that interrupt handlers on shared IRQ lines are all called every time an interrupt occurs.

Enable and filter the interrupts

In fact, at the moment, reception and interrupts are not enabled at the level of the serial port controller. So in the initialization function of the module:

- Write `ATMEL_US_RSTSTA | ATMEL_US_RSTRX` to the `ATMEL_US_CR` register;
- Write `ATMEL_US_TXEN | ATMEL_US_RXEN` to the `ATMEL_US_CR` register;
- Write `ATMEL_US_RXRDY` to the `ATMEL_US_IER` register (IER stands for Interrupt Enable Register).

Now, in our interrupt handler we want to filter out the interrupts that come from the serial controller. To do so, read the value of the `ATMEL_US_CSR` register and the value of the `ATMEL_US_IMR` register.



If the result of a *binary and* operation between these two values is different from zero, then it means that the interrupt is coming from our serial controller.

If the interrupt comes from our serial port controller, print a message and return `IRQ_HANDLED`. If the interrupt doesn't come from our serial port controller, just return `IRQ_NONE` without printing a message.

Compile and load your driver. Have a look at the kernel messages. You should no longer be flooded with interrupt messages.

Start `minicom` on `/dev/ttyUSB1`. Press one character (nothing will appear since the target system is not echoing back what we're typing). Then, in the kernel log, you should see the message of our interrupt handler. If not, check your code once again and ask your instructor for clarification!

Read the received characters

You can read the received characters by reading the `ATMEL_US_RHR` register using `readl()`. It must be done in code that loops until the `ATMEL_US_RXRDY` bit of the `ATMEL_US_CSR` register goes back to zero. This method of operation allows to read several characters in a single interrupt.

For each received character, print a message containing the character.

Compile and load your driver. From `minicom` on `/dev/ttyUSB1` on the host, send characters to the target. The kernel messages on the target should properly tell you which characters are being received.

Sleeping, waking up and communication

Now, we would like to implement the `read()` operation of our driver so that a userspace application reading from our device can receive the characters from the serial port.

First, we need a communication mechanism between the interrupt handler and the `read()` operation. We will implement a very simple circular buffer. So let's declare a global buffer in our driver:

```
#define SERIAL_BUFSIZE 16
static char serial_buf[SERIAL_BUFSIZE];
```

Two integers that will contain the next location in the circular buffer that we can write to, and the next location we can read from:

```
static int serial_buf_rd, serial_buf_wr;
```

In the interrupt handler, store the received character at location `serial_buf_wr` in the circular buffer, and increment the value of `serial_buf_wr`. If this value goes greater than `SERIAL_BUFSIZE`, reset it to zero.

In the `read()` operation, if the `serial_buf_rd` value is different from the `serial_buf_wr` value, it means that one character can be read from the circular buffer. So, read this character, store it in the userspace buffer, update the `serial_buf_rd` variable, and return to userspace (we will only read one character at a time, even if the userspace application requested more than one).

Now, what happens in our `read()` function if no character is



available for reading (i.e, if `serial_buf_wr` is equal to `serial_buf_rd`)? We should put the process to sleep!

To do so, declare a global wait queue in our driver, named for example `serial_wait`. In the `read()` function, use `wait_event_interruptible()` to wait until `serial_buf_wr` is different from `serial_buf_rd`. And in the interrupt handler, after storing the received characters in the circular buffer, use `wake_up()` to wake up all processes waiting on the wait queue.

Compile and load your driver. Run `cat /dev/serial` on the target, and then in Minicom on the development workstation side, print some characters. They should appear on the remote side if everything works correctly!





Kernel - Locking

Objective: practice with basic locking primitives

During this lab, you will

- Practice with locking primitives to implement exclusive access to the device.

Setup

Stay in the `/home/<user>/felabs/linux/character` directory.

You need to have completed the previous two labs to perform this one.

Boot your board with the same NFS environment as before, and load your serial module.

Adding appropriate locking

We have two shared resources in our driver:

- The buffer that allows to transfer the read data from the interrupt handler to the `read()` operation.
- The device itself. It might not be a good idea to mess with the device registers at the same time and in two different contexts.

Therefore, your job is to add a spinlock to the driver, and use it in the appropriate locations to prevent concurrent accesses to the shared buffer and to the device.

Please note that you don't have to prevent two processes from writing at the same time: this can happen and is a valid behavior. However, if two processes write data at the same time to the serial port, the serial controller should not get confused.





Kernel – Kernel crash and kernel debugging

Objective: Analyze a kernel crash, and perform kernel debugging with a hardware debugger.

Kernel crash analysis

Setup

Go to the `/home/<user>/felabs/linux/debugging` directory.

Get 2.6.29 kernel sources for the Calao board. The kernel must be compiled with:

- Support for root filesystem over NFS support
- The `CONFIG_DEBUG_INFO` configuration option, (Kernel Hacking section) which makes it possible to see source code in the disassembled kernel

The `nfsroot/` directory is the root filesystem.

Compile the `drvbroken` module provided in `nfsroot/root/drvbroken`, after modifying the Makefile so that `KDIR` properly points to your kernel source tree.

Run the target system on the Calao board, and load the `drvbroken` kernel module. See it crashing in a nice way.

Analyzing the crash message

Analyze the crash message carefully. Knowing that on ARM, the `pc` register contains the location of the instruction being executed, find in which function does the crash happens, and what the function call stack is.

Using LXR (for example <http://lxr.free-electrons.com>) or the kernel source code, have a look at the definition of this function. This, with a careful review of the driver source code should probably be enough to help you understand and fix the issue.

Further analysis of the problem

If the function source code is not enough, then you can look at the disassembled version of the function, either using:

```
arm-linux-objdump -S linux-2.6.29/vmlinux >
vmlinux.disasm
```

or:

```
arm-linux-gdb linux-2.6.29/vmlinux
(gdb) disassemble function_name
```

Then find at which exact instruction the crash occurs. The offset is provided by the crash output, as well as a dump of the code around the crashing instruction.

Of course, analyzing the disassembled version of the function requires some assembly skills on the architecture you are working on.



Kernel debugging with JTAG

Setup

Download OpenOCD 0.1.0 from OpenOCD website at <http://openocd.berlios.de/web/>. Install the `libftdi-dev` package. `libftdi` is a library needed to drive the JTAG through the FTDI2232 chip, visible on Calao board between the main USB connector and the Atmel CPU.

Then, configure and compile OpenOCD as follows:

```
./configure --enable-ft2232_libftdi
make
```

You don't necessarily need to install OpenOCD system-wide, we'll use it directly from its compilation directory.

Your lab directory contains an `openocd.cfg` configuration file for your board. Have a look at this file. The commands are described in the OpenOCD documentation at <http://openocd.berlios.de/doc/index.html>.

Usage

Then, run OpenOCD:

```
sudo ~/openocd-0.1.0/src/openocd -f openocd.cfg
```

OpenOCD must be started with root privileges in order to access the USB device. When started, OpenOCD should say (amongst other messages):

```
Warn : no gdb port specified, using default port 3333
```

So we will use port 3333 to connect to the GDB port of OpenOCD. In another terminal, start GDB:

```
arm-linux-gdb
```

Then, in GDB, load the kernel ELF image, which contains all the debugging symbols:

```
(gdb) symbol-file <path>/linux-2.6.29/vmlinux
```

And connect to the GDB port of OpenOCD:

```
(gdb) target remote localhost:3333
```

GDB can now be used as usual, except that it allows to do kernel debugging. To test it, set a breakpoint on the kernel function `sys_sync()` (using `b sys_sync`) and then continue the execution (using `c`). On your Calao board, run the `sync` command, which will trigger a call to the `sys_sync()` kernel function, that implements the `sync` system call. The breakpoint should be hit, and you get access to the GDB prompt at the beginning of the `sys_sync()` function.

Exploring kernel internals

We can now explore the kernel internals. For example, let's see the contents of the `task_struct` structure. On ARM, the bottom of the kernel stack of each process contains a `thread_info` structure, which points to `task_struct`. So, by accessing the stack pointer register, we can access the `task_struct` structure of the currently



running process.

First, let's print the value of the stack pointer register:

```
p $sp
```

Then, let's compute the bottom the stack, knowing that the stack size is 8k bytes:

```
p ((unsigned long) $sp & ~(8191))
```

Now, convert this address to a `struct thread_info` pointer:

```
p (struct thread_info *) ((unsigned long)$sp & ~(8191))
```

And finally, access the `task` member of this structure to get the `task_struct` pointer:

```
p ((struct thread_info *) ((unsigned long)$sp & ~(8191)))->task
```

We can then show the PID and the name of the current process:

```
p ((struct thread_info *) ((unsigned long)$sp & ~(8191)))->task->pid
```

```
p ((struct thread_info *) ((unsigned long)$sp & ~(8191)))->task->comm
```