New features in Linux 2.6
Training lab book

Thomas Petazzoni
Free Electrons
http://free-electrons.com

About this document

This document is part of an embedded Linux training from Free Electrons.

You will find the whole training materials (slides and lab book)
on http://free-electrons.com/training/linux26.

Lab data can be found on http://free-electrons.com/labs/embedded_linux.tar.bz2.

Copying this document

Document updates and translations available on http://free-electrons.com/docs/linux26-features.

Corrections, suggestions, contributions and translations are welcome!

Training setup

See the training labs on http://free-electrons.com/training/drivers for setup instructions, which are shared with these practical labs.

Lab 1 – Use the new *debugfs* filesystem

Objective: Set up and use in a simple kernel module the new *debugfs* facility

After this lab, you will be able to

- Configure and compile the Linux kernel with debugfs support
- Configure userspace to access debugfs
- Create, compile and use a kernel module that exports debugging information to userspace using *debugfs*

Setup

Go to the `/mnt/labs/linux26/lab1` directory.

Make sure you have at least 600 MB of free disk space.

Getting, configuring and compiling the kernel

Create a `linux/` directory and download the 2.6.24 kernel inside this directory using `ketchup`.

Modify the main kernel Makefile to do cross-compilation on ARM : set `ARCH` to `arm` and `CROSS_COMPILE` to `arm-linux-`. Then, configure the kernel for the Realview machine type, using

```
make realview_defconfig
```

Edit the generated configuration to :

- enable support for the debug filesystem, in the *Kernel Hacking* section of the kernel configuration tool
- enable support for root filesystem over NFS (in Networking, Kernel level autoconfiguration, and in File systems, NFS client support and NFS root filesystem support)

Finally, compile your kernel !

NFS environment setup

Setup your system so that the `nfsroot/` directory is exported by NFS. You can follow the instructions of lab 4 of « Embedded Linux kernel and driver development » training book, available of Free Electrons' website.

You should now be able to run Qemu using the provided `run_qemu` script, and see a shell start after the kernel boot. Make sure that the *debugfs* support is properly compiled by looking at the `/proc/filesystems` file.

Setup the *debugfs* filesystem

To access the informations exported through *debugfs* by the kernel, this virtual filesystem needs to be mounted. In the target filesystem, create a `/debug` directory, and make sure the *debugfs* filesystem gets mounted during boot by modifying the `/etc/init.d/rcS` script.

Use *debugfs*

The `/src` directory of the target filesystem contains a kernel module

skeleton, with two variables: `myvar` and `str`. The goal of the lab is to export them to userspace using *debugfs*. Documentation for *debugfs* is available at:

http://www.free-electrons.com/kerneldoc/latest/DocBook/filesystems/

Fill the `debugfstest_init()` and `debugfstest_exit()` functions to properly export and unexport these variables to userspace, in a new *debugfs* directory named `test`.

Lab 2 – Creating an userspace driver

Objective: Experiment the new UIO framework of the Linux kernel to implement a simple userspace driver

After this lab, you will be able to

- Configure and compile the Linux kernel with UIO support
- Create the small kernel module needed for UIO
- Create an userspace application that interacts with the hardware through UIO

Setup

Go to the `/mnt/labs/linux26/lab2` directory.

Make sure you have at least 600 MB of free disk space.

Getting, configuring and compiling the kernel

Create a `linux/` directory and download the 2.6.24 kernel inside this directory using `ketchup`. Apply the `data/uio-on-arm.patch` patch to the kernel tree.

Patching the kernel is needed because the main ARM Kconfig file does not use the default drivers/Kconfig file. So every time a new driver type is added to the kernel, the ARM Kconfig file must be updated. Work is ongoing to fix that in the next version of the kernel.

Modify the main kernel Makefile to do cross-compilation on ARM: set `ARCH` to `arm` and `CROSS_COMPILE` to `arm-linux-`. Then, configure the kernel for the Realview machine type, using

```
make realview_defconfig
```

Edit the generated configuration to:

- disable support for the PL011 serial driver
- enable support for UIO
- enable support for root filesystem over NFS (in Networking, Kernel level autoconfiguration, and in File systems, NFS client support and NFS root filesystem support)

Finally, compile your kernel!

NFS environment setup

Setup your system so that the `nfsroot/` directory is exported by NFS. You can follow the instructions of lab 4 of « Embedded Linux kernel and driver development » training book, available of Free Electrons' website.

You should now be able to run Qemu using the provided `run_qemu` script, and see a shell start after the kernel boot.

Basic UIO driver

The goal of the lab is to write a simple UIO driver for the PL011 UART, which will allow to send and receive characters through the serial port.

UIO drivers are made of two parts: a small kernel module that registers the driver and contains a basic interrupt handler, and an userspace part, which is the core of the driver. Documentation about UIO is available in the DocBook format in the kernel `Documentation/`

directory and a compiled version of it is available online at http://www.free-electrons.com/kerneldoc/latest/DocBook/uio-howto.

The `nfsroot/src/` directory contains a skeleton for the kernel module and the userspace driver. Your first task is to get the character emission working by :

- filling the `pl011_probe()` and `pl011_remove()` functions so that they properly register and unregister the device to the UIO framework
- filling the `main()` function of the userspace driver to open the UIO device, map its register region to memory using `mmap()`, and write some characters to the correct register so that they get written to the serial port

The following hints will help you in filling the functions :

- In the pl011_probe() function, the physical memory address of the serial port is available in `dev->res.start`, and the end of the register region is in `dev->res.end`.
- Start without any interrupt handler.
- Sending bytes through the serial ports is done by writing to the `UART01x_DR` register, for which a macro is available in the userspace driver skeleton. The macro contains the offset of the register in the register region.
- You will have to create the `/dev/uio0` device node with the correct major and minor numbers. They can be found in `/sys/class/uio/`.
- The 2.6.24 kernel contains the kernel part of an UIO driver, in the file `drivers/uio/uio_cif.c`. Do not hesitate to use it as an example !

To test your driver, you will have to first compile it on the host using the provided `Makefile`, which compiles both the kernel and user parts. Then, on the target, you need to insert the kernel module into the running kernel :

```
insmod /src/pl011-uio.ko
```

Once the kernel module is correctly inserted, you can run the userspace driver :

```
/src/user
```

Note that Qemu is configured with the `-serial stdio` option which means that the first serial port input and output take place through the terminal that was used to run Qemu.

Adding support for reception

To get reception from the serial port working, we will use the interrupt handling facilities provided by UIO. By registering a simple interrupt handler through UIO, your userspace driver will be able to block using the `read()` system call until an interrupt occurs.

By following the documentation of UIO, write and register an interrupt handler in your kernel module that :

- reads from the `UART011_MIS` register, and if the value is zero, returns with `IRQ_NONE`. A zero value means that no interrupt

To get your `pl011_remove()` function working, you'll have to use the `amba_set_drvdata()` and `amba_get_drvdata()` function. They allow to associate an opaque pointer, such as your pointer to the `struct uio_info` to an `amba_device` structure. That way, you'll be able to get back your `struct uio_info` pointer in `pl011_remove()`.

occurred for this device : the interrupt is probably coming for another device sharing the same IRQ line.

- writes 0 to the `UART011_IMSC` register to disable the interrupt and returns `IRQ_HANDLED`. It allows to disable the interrupts until it is handled by userspace, and to notify the Linux kernel that the interrupt was correctly handled.

In the `pl011_probe()` function, in addition to the registration in UIO of the interrupt handler, we need to properly initialize the device to get interrupts working :

- write `UART011_IFLS_TX4_8` to the `UART011_IFLS` register
- write `UART01x_CR_UARTEN | UART011_CR_RXE | UART011_CR_TXE` to the `UART011_CR` register
- write `UART011_RXIM` to the `UART011_IMSC` register

On the userspace side, we can now wait for interrupts by simply reading a four bytes integer from the UIO device. The integer contains the number of interrupts received up to now. It allows to detect if interrupts were missed or not. When an interrupt has been received, we can read the received character from the same register we used to transmit characters. To get the next interrupts, we also need to re-enable interrupts by writing `UART011_RXIM` to the `UART011_IMSC` register.

Now, when pressing keys in the console that runs Qemu, an interrupt should be generated and your userspace driver should be properly notified and able to receive the characters.