Kernel – Serial controller device driver programming

Objective: Develop a serial device driver for the AT91SAM9263 CPU from scratch.

Warning

In this lab, we are going to re-implement a driver that already exists in the Linux kernel tree. Since the driver already exists, you could just copy the code, compile it, and get it to work in a few minutes. However, the purpose of this lab is to re-create this driver from scratch, taking the time to understand all the code and all the steps. So please play the game, and follow our adventure of creating a serial driver from scratch!

Setup

Go to the `/home/<user>/felabs/linux/character` directory. It contains the root filesystem that you will mount over NFS to work on this lab. Re-use the setup instructions of the lab on Character Device Drivers to get a kernel without the serial port driver and with Network Console support.

Basic module

The serial core cannot be compiled inside the kernel without an in-tree kernel driver. Therefore, for this lab, we will work directly inside the kernel source tree and not using an external module.

To do so:

- Create a basic module in `drivers/serial/fedrv.c` with just the init and cleanup functions ;
- Add a new configuration option in `drivers/serial/Kconfig`. Don't forget to select `SERIAL_CORE` in this option ;
- Update `drivers/serial/Makefile` to make sure your driver gets compiled when your new option is selected

Compile your new driver as a module, and after the kernel compilation, run:

```
make INSTALL_MOD_PATH=/path/to/nfsroot modules_install
```

To install the modules (`serial_core` and your driver) into the root filesystem. Then try to load/unload your module on the target using `modprobe`. If you're successful, we can now start working on the driver itself.

Register the UART driver

Instantiate an `uart_driver` structure with the following values:

- `owner`, `THIS_MODULE`
- `driver_name`, “`fedrv`” or any other string
- `dev_name`, “`ttyS`”
- `major`, `TTY_MAJOR` (this is the usual major for TTY devices)

- `minor`, 64 (this is the usual minor for TTY serial devices, see `Documentation/devices.txt` in the kernel source tree)
- `nr`, 1 (we will support only one port)

In the init function, register the UART driver with `uart_register_driver()` and in the cleanup function, unregister it with `uart_unregister_driver()`.

Integration in the device model

To get notifications of the UART devices that exist on our board, we will integrate our driver in the device model.

To do, so, first instantiate a platform_driver structure, with pointers to the `probe()` and `remove()` methods (they can be left empty at the moment). The driver name must be "`atmel_usart`" to match the `device definitions in arch/arm/mach-at91/`.

You should mark the probe function with `__devinit` and the remove function with `__devexit`. The remove operation should be declared as follows:

```
.remove = __devexit_p(fedrv_remove)
```

So that if the driver is statically compiled, the `fedrv_remove()` function is not compiled in and the `.remove` pointer is `NULL`.

Then, in the init and cleanup functions of the module, register and unregister the platform driver using `platform_driver_register()` and `platform_driver_unregister()`.

Finally, you need to make a small modification to the kernel. Currently, the "`atmel_usart`" platform devices are only added if the Atmel serial port driver is compiled in. However, since we disabled this driver (because we are re-implementing it), we must modify a little the board code. So, in `arch/arm/mach-at91/at91sam9263_devices.c`, replace:

```
#if defined(CONFIG_SERIAL_ATMEL)
```

by

```
#if 1
```

Then, recompile your kernel, re-flash it, and test your new module. You should see your `probe()` function being called (after adding a simple `printk()` in it). And in `/sys/devices/platform/`, you should see the device `atmel_usart.0`. This directory contains a symbolic link driver to the `atmel_usart` driver. If you follow this symbolic link, you should discover that the `atmel_usart` driver is implemented by the `fedrv` module. Congratulations!

Registering the port

Now, it's time to implement the `probe()` and `remove()` functions. Before that, we need a few definitions:

- Declare a global `uart_port` structure, that will be used to contain the informations about the single port we will manage;
- Declare an empty `uart_ops` structure.

Then, in the `probe()` operation:

- Make sure pdev->id is 0 (we only want to handle the first serial port). If it's not zero, bail out with -ENODEV
- Initialize the fields of the uart_port structures
  - ->ops should point to the uart_ops structure
  - ->dev should point to the struct device embedded in the platform device structure. So &pdev->dev should work
  - ->line should be the serial port number, i.e 0 or pdev->id
- Register the port with uart_add_one_port()
- Associate the port pointer to the platform device structure using platform_set_drvdata(). This will make it easy to find the port structure from the platform device structure in the remove() operation.

In the remove() method:

- Get the port structure from the platform device structure using platform_get_drvdata()
- Unregister the port with uart_remove_one_port().

Now, when testing your driver, in /sys/devices/platform/atmel_usart.0/, you should have a tty directory, which itself contains a ttyS0 directory. Similarly, if you go in /sys/class/tty/ttyS0, you should see that the ttyS0 device is handled by atmel_usart.0. Good!

Polled mode transmission

To keep our driver simple, we will implement a very simple polled-mode transmission model.

In the probe() operation, let's define a few more things in the port structure:

- ->fifosize, to 1 (this is hardware-dependent)
- ->iotype should be UPIO_MEM because we are accessing the hardware through memory-mapped registers
- ->flags should be UPF_BOOT_AUTOCONF so that the config_port() operation gets called to do the configuration
- ->mapbase should be pdev->resource[0].start, this is the address of the memory-mapped registers
- ->membase should be set to data->regs, where data is the device-specific platform data associated to the device. In our case, it's a atmel_uart_data structure, available through pdev->dev.platform_data. In the case of the first serial port data->regs is non-zero and contains the virtual address at which the registers have been remapped. For the other serial ports, we would have to ioremap() them.

Then, we need to create stubs for a fairly large number of operations. Even if we don't implement anything inside these operations for the moment, the serial_core layer requires these operations to exist:

- tx_empty()

- `start_tx()`
- `stop_tx()`
- `stop_rx()`
- `type()`
- `startup()`
- `shutdown()`
- `set_mctrl()`
- `set_termios()`
- `release_port()`
- `request_port()`
- `config_port()`

First, let's implement what's related to setting and getting the serial port type:

- In the `config_port()` operation, if `flags & UART_CONFIG_TYPE` is true, then set `port->type = PORT_ATMEL`. There is a global list of serial port types, and we are re-using the existing definition.
- In the `type()` operation, if `port->type` is `PORT_ATMEL` return a string like "`ATMEL_SERIAL`", otherwise return `NULL`.

Now, for the transmission itself, we will first implement `tx_empty()`. In this function, read the register `ATMEL_US_CSR` from the hardware (note: the virtual base address of the registers is in `port->membase`). If bit `ATMEL_US_TXEMPTY` is set, it means that the port is ready to transmit, therefore return `TIOCSER_TEMT`, otherwise return 0.

Then, the `start_tx()` function will do the transmission itself. Iterate until the transmission buffer is empty (use `uart_circ_empty()`) and do:

- call an auxiliary function that prints one character
- update the tail pointer of the transmission buffer
- increment `port->icount.tx`

The auxiliary function should wait until bit `ATMEL_US_TXRDY` gets set in the `ATMEL_US_CSR` register, and then send the character through the `ATMEL_US_THR` register.

Then, compile and load your driver. You should now be able to do echo "foo" > /dev/ttyS0.

Implementing reception

The last part to make our driver usable is to implement reception.

We first need to modify the `probe()` method to set `port->irq` to `pdev->resource[1].start` so that we fetch the IRQ number from the board-specific platform device definition.

Then, in the `startup()` operation, do the following steps:

- Disable all interrupts in the serial controller by writing `~0UL` to the `ATMEL_US_IDR` register

- Register the IRQ channel `port->irq` to an interrupt handler `fedrv_interrupt()`. Pass `port` as the `dev_id` so that we get a pointer to the port in the interrupt handler. Make it a shared interrupt.
- Reset the serial controller by writing `ATMEL_US_RSTSTA | ATMEL_US_RSTRX` to the `ATMEL_US_CR` register
- Enable transmission and reception by writing `ATMEL_US_TXEN | ATMEL_US_RXEN` to the `ATMEL_US_CR` register
- Enable interrupts on reception by writing `ATMEL_US_RXRDY` to the `ATMEL_US_IER` register

Similarly, in the `shutdown()` operation, do:

- Disable all interrupts by writing `~0UL` to the `ATMEL_US_IDR` register
- Free the IRQ channel using `free_irq()`.

Then, in the interrupt handler, do the following:

- Read the `ATMEL_US_CSR` register to get the controller status and perform the logical and of this value with the enabled interrupts by reading the `ATMEL_US_IMR` register. If the resulting value is 0, then the interrupt was not for us, return `IRQ_NONE`.
- If the result value has bit `ATMEL_US_RXRDY` set, call an auxiliary function `fedrv_rx_chars()` to receive the characters.

Finally, we have to implement the `fedrv_rx_chars()` function. This function should read the `ATMEL_US_CSR` register, and while `ATMEL_US_RXRDY` is set in this register, loop to read characters the following way:

- Read one character from the `ATMEL_US_RHR` register
- Increment `port->icount.rx`
- Call `uart_insert_char()` with the value of the status register, overrun to `ATMEL_US_OVRE`, and the flag set to `TTY_NORMAL` (we don't handle break characters, frame or parity errors, etc. for the moment)

Once all characters have been received, we must tell the upper layer to push these characters, using `tty_flip_buffer_push()`.

Now, if you do cat `/dev/ttyS0`, you should be able to receive characters from the serial port. By default, a `ttyS` is opened in the so-called “canonical” mode, so the characters are sent to the reading process only after entering a newline character.

You can also try to run the program that will display the login prompt and then a shell:

```
/sbin/getty -L ttyS0 115200 vt100
```

Go back to your Minicom, you should be able to login normally, but using your own serial driver!

Improvements

Of course, our serial driver needs several improvements:

- Real implementation of `set_termios()` and `set_mctrl()`
- Usage of interrupts for transmission
- Console support for early messages
- Support of several serial ports
- Handle parity and frame errors properly
- Support break characters and SysRq
- Use of DMA for transmission and reception
- etc.