AVR32743: AVR32 AP7 Linux Kernel Module Application Example

32-bit AVR®
Microcontrollers

Application Note

Features

- **Linux kernel module basics**
  - **Initialization and cleanup**
  - **Module parameters**
  - **Build system integration**
- **Platform driver example**
- **SPI driver example**

1 Introduction

This application note explains how to write a custom loadable driver module for the Linux® kernel. Common mechanisms like module parameters, initialization and cleanup are explained in detail. The Linux driver model is introduced, and a short introduction on how to write drivers for two common bus types, the platform and spi busses, is given. This application note also explains how to build a custom module using the Linux kernel's internal build system.

Rev. 32098A-AVR-07/08

2 Kernel module basics

This section explains some of the basics about loadable Linux kernel modules: How to do initialization and cleanup, how to pass parameters and how to build and execute the module. A simple kernel module example, “simple”, is provided to demonstrate how this works.

When trying out the examples, it is recommended to increase the kernel console log level to at least 7. Otherwise, some of the output from the example modules may not be shown on the console, although it will still be shown by the “dmesg” command. The following command will set the console log level appropriately:

```
echo 7 4 1 7 > /proc/sys/kernel/printk
```

The first number indicates the console log level; setting it to 8 will show debug messages on the console as well.

2.1 A simple example

To build the “simple” module, unpack the sources that came with this application note and enter the “simple” subdirectory. In there, you’ll find three files:

- simple.c. This is the source code for the module.
- Kbuild: This is used by the kernel build system (kbuild) to determine how the module is to be built.
- Makefile: Simple makefile that utilizes kbuild to build the module.

Everything is set up to work out of the box, except for one thing: The Makefile wrapper needs to know where the AVR®32 Linux kernel sources are located. After editing the Makefile and updating the KDIR variable if necessary, simply run “make” to build the module.

After the module has been successfully built, copy it onto the target system and load it using the “insmod” utility:

```
insmod simple.ko
```

If nothing appears to be happening, the kernel log level may be set too low. Try running “dmesg” and look at the last few lines of the output:

```
Hello World!
The number is: 0
```

To unload the module, use the “rmmod” utility:

```
rmmod simple
```

2.2 Module initialization and cleanup

Every kernel module should include at least two functions: The initialization function and the cleanup function. In many cases, these two functions are the only ones that are called directly by the kernel; everything else is called as a result of registering the driver with one or more subsystems.

The names of the initialization and cleanup functions don’t matter, but the usual convention is to use a name on the form *module-name*_init for the initialization function and *module-name*_exit for the cleanup function. To specify which functions are to be used as init- and cleanup functions, use the macros **module_init()** and **module_exit()** respectively, like this:

```
int __init my_module_init(void)
{
    /* Initialization code goes here */
    return 0;
}
module_init(my_module_init);

void __exit my_module_exit(void)
{
    /* Cleanup code goes here */
}
module_exit(my_module_exit);
```

In the example above, the function my_module_init() will be called after the module has been loaded into memory, while the my_module_exit() function will be called right before the module is unloaded.

If the driver is built into the kernel instead of as a loadable module, no modifications are necessary. The module_init() macro will ensure that the initialization function is called at some point during the boot process, while the module_exit() macro will not do anything since the driver will never be unloaded.

2.2.1 Reducing the run-time memory footprint

The previous example used two special preprocessor macros to annotate the init- and cleanup functions: __init and __exit. __init means that the function is only used for initialization, and can be discarded once the driver has been successfully initialized. __exit means that the function can be discarded if the driver will never be unloaded (i.e. if it is compiled into the kernel instead of as a loadable module.)

Properly annotating functions using the __init and __exit macros may help reduce the amount of kernel memory consumed by the driver after it has been fully loaded and initialized. But be careful not to annotate any functions that may be called while the module is fully operational.

2.2.2 Return values

The vast majority of kernel functions use a common return value convention, returning 0 for success and a negative error code for failure. The module initialization functions must also follow this convention, so if anything goes wrong during initialization, the function should return a negative value, preferably a meaningful one.

For example, if the user specified an invalid module parameter, the module init function should return **–EINVAL**, which means “Invalid Argument”. Other meaningful error codes may be found in include/asm-generic/errno-base.h and include/asm-generic/errno.h.

2.3 Module parameters

Module parameters are a straightforward mechanism for allowing user-specific settings. When loading the module, the user may override certain defaults in the module through the command line.

The “simple” example above takes a single module parameter, “number”, which can be overridden on the insmod command line, like this:

```
insmod simple.ko number=42
```

This results in the following being printed to the console:

```
Hello World!
The number is: 42
```

Module parameters are basically just regular global or static variables which are subjected to a bit of preprocessor magic. The following three lines of code from the "simple" module is all it takes to support module parameters:

```
static int number;
module_param(number, int, S_IRUGO);
MODULE_PARM_DESC(number, "Number to be printed on startup");
```

If the "number" parameter is overridden on the command line, the variable "number", will be initialized to the specified value before the module initialization function is called. If the parameter is not specified on the command line, it will retain its initial value, 0.

The second parameter to module_param() specifies the type of the parameter, which can be one of *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *charp* (pointer to character array), *bool* or *invbool* (same as *bool*, but the inverse of the specified value is stored). The module_param() macro is defined in "include/linux/moduleparam.h".

The third parameter to module_param() specifies the file mode of the parameter as it appears under /sys/module. S_IRUGO means that the file should be readable for everyone, but not writeable for anyone. This is usually a good choice since writable module parameters need special care to be taken when implementing the module.

If the driver is built into the kernel, module parameters can still be specified on the kernel command line using the form "*module-name*.*parameter-name*=*value*". For example, if the "simple" module is built-in, the "number" parameter can be set to 42 by passing "simple.number=42" on the kernel command line.

2.3.1 Other forms of module metadata

It is considered good practice to include some additional information about the module. The following macros can be used for this purpose:

- **MODULE_DESCRIPTION**: A short description of the purpose of the module.
- **MODULE_AUTHOR**: Information about who developed the module. Usually, this is specified on the form "Full Name <e-mail address>", but the e-mail address can be omitted.
- **MODULE_LICENSE**: Which license the module is distributed under.

The license types recognized by the kernel can be found in "include/linux/module.h". Note that loading a module that has a license which is not GPL-compatible (i.e. "Properietary" or a license type the kernel doesn't recognize) will "taint" the kernel. This means that if the kernel crashes, the register dump will show that the kernel is tainted, which is a hint to other developers that the crash may be caused by code they don't have access to. Many developers will ignore bug reports from tainted kernels because of this. Note that even if a proprietary module is unloaded, the kernel will stay tainted until the next reboot.

2.4 Building a custom module

The easiest way to build a custom module is to use the kernel's build system. This ensures that the correct compiler flags and configuration options are being used.

To build the module using the kernel's build system, first add a file named "Kbuild" to the directory containing the module's source code containing one or more lines on the form

```
obj-m            += my_module.o
```

This will instruct the kernel's build system to take a single file named "my_module.c" and turn it into a module named "my_module.ko". To build more than one module in a single directory, simply add more such lines to the Kbuild file.

Building a single module from several source files can be done by adding a few lines on the following form:

```
obj-m            += my_module.o
my_module-y      += my_module-1.o my_module-2.o
```

This will instruct the kernel's build system to compile the files "my_module-1.c" and "my_module-2.c" into separate object files, and then link them together to produce a module named "my_module.ko".

For more information about the kernel's build system, please consult the documentation in "Documentation/kbuild".

When an appropriate Kbuild file is in place, the modules can be built by running

```
make ARCH=avr32 CROSS_COMPILE=avr32-linux- –C $KERNEL_SRC_DIR \
           M=`pwd` modules
```

Where $KERNEL_SRC_DIR should be replaced by the full path to a Linux kernel source tree configured for AVR32. The examples distributed with this application note include a wrapper Makefile which allows the modules to be built by simply running "make".

3 The Linux driver model

The Linux Device model is built around the concept of *busses*, *devices* and *drivers*. All devices in the system are connected to a bus of some kind. The bus does not have to be a real one; busses primarily exist to gather similar devices together and coordinate initialization, shutdown and power management.

When a device in the system is found to match a driver, they are bound together. The specifics about how to match devices and drivers are bus-specific. The PCI bus, for example, compares the PCI Device ID of each device against a table of supported PCI IDs provided by the driver. The platform bus, on the other hand, simply compares the name of each device against the name of each driver; if they are the same, the device matches the driver.

Binding a device to a driver involves calling the driver's *probe()* function passing a pointer to the device as a parameter. From this point on, it's the responsibility of the driver to get the device properly initialized and register it with any appropriate subsystems.

Devices that can be hot-plugged must be un-bound from the driver when they are removed from the system. This involves calling the driver's *remove()* function passing a pointer to the device as a parameter. This also happens if the driver is a dynamically loadable module and the module is unloaded.

All device driver callbacks, including *probe()* and *remove()*, must follow the return value convention described in section 2.2.2.

4 Writing a platform driver

The "platform" example distributed with this application note demonstrates how devices are bound and un-bound to drivers on the platform bus. The "platform_driver" module is the driver part and does not take any parameters. To load it, simply run the following command.

```
insmod platform_driver.ko
```

It will not show any output to the console unless the "platform_test" module has already been loaded.

The "platform_test" module can be used to test the "platform_driver" module. It takes two parameters. The first one, "nr_devices", specifies how many platform devices to be added to the system; by default, one platform device is registered. The other one, "value", specifies a value to be passed to the driver as platform-specific data.

When both the "platform_driver" and "platform_test" modules have been loaded, the following is printed to the console:

```
Registering device "platform_example.0"...
platform_example platform_example.0: probe() called, value: 0
```

Unloading the "platform_test" module results in the following console output:

```
Removing device platform_example.0...
platform_example platform_example.0: remove() called
```

The order in which the modules are loaded and unloaded does not matter. When both are loaded, the devices are bound to the driver. When one of them goes away, all the devices are unbound.

4.1 Platform driver structure

A platform driver is usually built around a single instance of struct platform_driver. In the platform driver example, it is initialized as follows.

```
static struct platform_driver platform_example_driver = {
    .probe      = platform_example_probe,
    .remove     = __devexit_p(platform_example_remove),
    .suspend    = platform_example_suspend,
    .resume     = platform_example_resume,
    .driver     = {
          .name  = "platform_example",
    },
};
```

After this structure has been registered by calling platform_driver_register(), usually from the module initialization function, any devices named "platform_example" will be bound to the driver, resulting in platform_example_probe() to be called with the matching device as a parameter. If any matching devices show up later, they will be bound to the driver in the same way.

When the driver is unregistered by calling platform_driver_unregister(), usually from the module exit function, all the devices that have previously been bound to the driver are unbound, resulting in platform_example_remove() to be called with the device to be unbound as a parameter. If a device bound to the driver goes away, it will be unbound in the same way.

When the power management subsystem in Linux determines that a device is to be suspended, for example if the system as a whole is being suspended, platform_example_suspend() is called with the device as a parameter. When the device is to be resumed, platform_example_resume() is called with the device as a parameter.

Finally, the name of the driver is specified in the “driver” structure around which the platform_driver structure is wrapped. This name determines which devices can be bound to the driver.

4.1.1 Reducing the run-time memory footprint

In some cases, the hotplug functionality of the driver core is not considered useful. This is typically the case with drivers for on-chip hardware which can never actually be hot-plugged, so keeping all the initialization and cleanup code in memory after all the devices have been initialized is just a waste of precious memory.

Such drivers can be written in a way so that all the initialization code is discarded after the module has been initialized by using the “__init” annotation on the probe() callback. Since keeping references to discarded memory in the platform_driver structure is potentially dangerous, the probe() hook should be removed from the platform_driver instance and an alternative way of registering the driver should be used: platform_driver_probe(). This function takes the address of the probe() function as the second parameter and will bind all devices present at the time the driver was registered to the driver. Any matching devices that show up later will not be bound to the driver.

The “platform” example can be modified to consume slightly less memory by defining the platform device instance like this.

```
static struct platform_driver platform_example_driver = {
    .remove       = __exit_p(platform_example_remove),
    .suspend      = platform_example_suspend,
    .resume       = platform_example_resume,
    .driver       = {
           .name  = "platform_example",
    },
};
```

Then, the module initialization function must be modified to use the alternative way of registering the driver.

```
static int __init platform_example_init(void)
{
    return platform_driver_probe(&platform_example_driver,
                 platform_example_probe);
}
```

Now, the platform_example_probe() function may be safely annotated as __init, and platform_example_remove() as __exit. If the driver is built into the kernel, platform_example_remove() will be discarded as well, but if it is built as a module, it will be kept around to do any necessary cleanup when the module is unloaded.

5 Writing a SPI driver

The “spi” example distributed with this application note demonstrates how devices are bound and un-bound to drivers on the spi bus. The “spi_example” module does not take any parameters. To load it, simply run the following command.

```
insmod spi_example.ko
```

It doesn’t show any output on the console unless the board code added one or more matching SPI devices.

To add a SPI device, the board code must contain a structure like the following:

```
static struct spi_board_info spi1_board_info[] __initdata = {
        {
                .modalias       = "spi_example",
                .max_speed_hz   = 1000000,
                .chip_select    = 0,
                .mode           = SPI_MODE_3,
                .platform_data  = &spi_example_data,
        }
};
```

This structure must be passed to at32_add_device_spi() when adding the SPI master device.

```
at32_add_device_spi(1, spi1_board_info,
                    ARRAY_SIZE(spi1_board_info));
```

The platform_data field is optional, but for the example above to work, the platform data for the driver must be defined, for example like this.

```
static struct spi_example_data spi_example_data = {
        .value = 42,
};
```

When the board information above has been added to the kernel, something similar to the following should be printed on the console when the spi_example driver is loader:

```
spi_example spi1.0: probe() called, value: 42
spi_example spi1.0: sending 55 56 57 58...
spi_example spi1.0: received 00 00 00 00
```

If the MISO and MOSI lines are connected together, the values received should be the same as the values that were sent. Unloading the “spi_example” module results in the following console output:

```
spi_example spi1.0: remove() called
```

5.1 SPI driver structure

The SPI driver structure is by design very similar to the platform driver structure described in section 4.1. The main differences are that the driver instance must be defined as a struct spi_driver, it must be registered by calling spi_register_driver() and unregistered by calling spi_unregister_driver().

Also, the SPI driver framework doesn't have anything similar to the platform_driver_probe() function, so the memory saving techniques described in section 4.1.1 can not be applied to SPI drivers.

5.2 Adding SPI devices

As a rule, all SPI devices in the system are added by the board initialization code. The SPI subsystem does provide functions for adding SPI devices at run time, but these functions assume that all platform-specific setup such as port multiplexing has already been taken care of, so in general, it's best to avoid this interface. On most boards, the SPI devices are fixed, so the board initialization code is the most natural place to specify which devices are present and how they are connected.

For each SPI master device that has any devices connected to it, the board code must define an array with board-specific information about each slave device, like in the example given at the beginning of section 5. The meaning of each field is described in include/linux/spi/spi.h. The most common ones are:

- **modalias**. This must match the name of the driver (as specified in the struct spi_driver initializer.)
- **max_speed_hz**. The highest SCK rate in Hz supported by the device.
- **chip_select**. A number identifying the chip select line that the device is connected to. For example, 0 means the device is connected to NPCS0 on the SPI controller.
- **mode**. Specifies the clock polarity, clock phase and other parameters to be used when communicating with the device. For most devices, one of the combinations defined by **SPI_MODE0**..**SPI_MODE3** will do.
- **platform_data**. A pointer that can be used to pass platform-specific information to the device driver.

The board information array must be passed as the second parameter to at32_add_device_spi(). This function will also add a platform device for the master controller driver (atmel®_spi) and setting up the port multiplexer so that the chip selects of all the devices specified in the board information array will work.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
Avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature