

# Table of Contents

[1 Introduction](#20_2____Static_keys)

[1.2 Acknowledgements](#1_2___Acknowledgements)

[1.5 What Modules are in my Kernel?](#On_Arch_Linux)

[2 Headers](#2__Using_the_X_Window_System__It)

[3 Examples](#1______sudo_pacman__S_linux_head)

[4.2 Hello and Goodbye](#the_standard_kernel_build_mechan)

[4.3 The \_\_init and \_\_exit Macros](#3)

[4.4 Licensing and Module Documentation](#15)

[4.5 Passing Command Line Arguments to a Module](#4_5___Passing_Command_Line_Argum)

[4.6 Modules Spanning Multiple Files](#66)

[4.7 Building modules for a precompiled kernel](#6_______include_____Specifically)

[5 Preliminaries](#If_you_do_not_desire_to_actually)

[5.3 User Space vs Kernel Space](#that_do_the_real_work____system)

[5.4 Name Space](#Recall_the_discussion_about_libr)

[5.6 Device Drivers](#The_kernel_has_its_own_space_of)

[6 Character Device drivers](#6_Character_Device_drivers)

[6.2 The file structure](#6_2___The_file_structure)

[6.4 Unregistering A Device](#6_4___Unregistering_A_Device)

[6.6 Writing Modules for Multiple Kernel Versions](#156)

[7.1 The proc\_ops Structure](#31_________________offset____len)

[7.2 Read and Write a /proc File](#7_2___Read_and_Write_a__proc_Fil)

[7.3 Manage /proc file with standard filesystem](#82)

[7.4 Manage /proc file with seq\_file](#60)

[8 sysfs: Interacting with your module](#93_______endif)

[9 Talking To Device Files](#Set_the_value_of_myvariable_and)

[10 System Calls](#72___________return_0)

[11 Blocking Processes and threads](#286)

[11.2 Completions](#11_2___Completions)

[12 Synchronization](#84______MODULE_LICENSE__GPL)

[12.2 Spinlocks](#24________________pr_info__mutex)

[12.3 Read and write locks](#which_disable_both_hardware_and)

[12.4 Atomic operations](#34)

[13 Replacing Print Macros](#71______module_init_example_atom)

[13.2 Flashing keyboard LEDs](#70)

[14 GPIO](#thus_making_it_possible_to_trace)

[14.3 DHT11 sensor](#14_3___DHT11_sensor)

[15 Scheduling Tasks](#2______sudo_insmod_dht11_ko)

[15.2 Work queues](#for_compatibility__For_further_i)

[16.2 Detecting button presses](#handler__Usually_there_is_a_cert)

[16.3 Bottom Half](#198___________gpio_free_buttons)

[16.4 Threaded IRQ](#193)

[17 Virtual Input Device Driver](#Note_that_passing_NULL_to_both_h)

[18 Standardizing the interfaces: The Device Model](#102)

[19 Device Tree](#92___________return_0)

[19.3 Example: Device Tree Module](#__Device_specific_data__Custom_p)

[19.4 Device Tree Source Example](#103_________________compatible)

[19.5 Testing Device Tree Modules](#1_________Example_device_tree_fr)

[20 Optimizations](#__of_property_read_bool____Check)

[21 Common Pitfalls](#1______cat__dev_key_state)

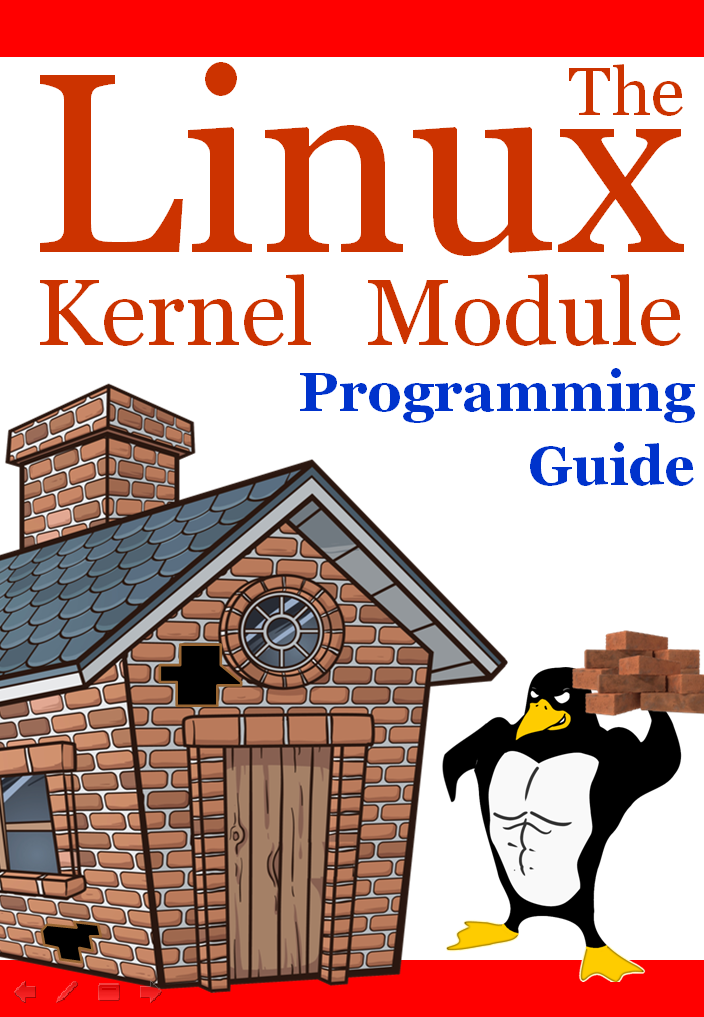
[2](#1______journalctl___since__1_hou)



# The Linux Kernel Module Programming Guide

Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, Jim Huang

September 4, 2025



Peter Jay Salzman, Michael Burian,

Ori Pomerantz, Bob Mottram,

Jim Huang

Contents

[1 Introduction](#20_2____Static_keys) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

[1.1 Authorship](#20_2____Static_keys) . . . . . . . . . . . . . . . . . . . . . . . . . . 4

[1.2 Acknowledgements](#1_2___Acknowledgements) . . . . . . . . . . . . . . . . . . . . . . 5

[1.3 What Is A Kernel Module?](#1_2___Acknowledgements) . . . . . . . . . . . . . . . . . 5

[1.4 Kernel module package](#1_2___Acknowledgements) . . . . . . . . . . . . . . . . . . . 5

[1.5 What Modules are in my Kernel?](#On_Arch_Linux) . . . . . . . . . . . . . . 6

[1.6 Is there a need to download and compile the kernel?](#On_Arch_Linux) . . . 6

[1.7 Before We Begin](#On_Arch_Linux) . . . . . . . . . . . . . . . . . . . . . . . 6

[2 Headers](#2__Using_the_X_Window_System__It) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

[3 Examples](#1______sudo_pacman__S_linux_head) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

[4 Hello World](#1______sudo_pacman__S_linux_head) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

[4.1 The Simplest Module](#1______sudo_pacman__S_linux_head) . . . . . . . . . . . . . . . . . . . . 8

[4.2 Hello and Goodbye](#the_standard_kernel_build_mechan) . . . . . . . . . . . . . . . . . . . . . . 13

[4.3 The \_\_init and \_\_exit Macros](#3) . . . . . . . . . . . . . . 14

[4.4 Licensing and Module Documentation](#15) . . . . . . . . . . . 15

[4.5 Passing Command Line Arguments to a Module](#4_5___Passing_Command_Line_Argum) . . . . . 16

[4.6 Modules Spanning Multiple Files](#66) . . . . . . . . . . . . . . 18

[4.7 Building modules for a precompiled kernel](#6_______include_____Specifically) . . . . . . . . . 19

[5 Preliminaries](#If_you_do_not_desire_to_actually) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

[5.1 How modules begin and end](#If_you_do_not_desire_to_actually) . . . . . . . . . . . . . . . . . 22

[5.2 Functions available to modules](#If_you_do_not_desire_to_actually) . . . . . . . . . . . . . . . 22

[5.3 User Space vs Kernel Space](#that_do_the_real_work____system) . . . . . . . . . . . . . . . . . 23

[5.4 Name Space](#Recall_the_discussion_about_libr) . . . . . . . . . . . . . . . . . . . . . . . . . . 24

[5.5 Code space](#Recall_the_discussion_about_libr) . . . . . . . . . . . . . . . . . . . . . . . . . . 24

[5.6 Device Drivers](#The_kernel_has_its_own_space_of) . . . . . . . . . . . . . . . . . . . . . . . . 25

[6 Character Device drivers](#6_Character_Device_drivers) . . . . . . . . . . . . . . . . . . . . . . . 27

[6.1 The file\_operations Structure](#6_Character_Device_drivers) . . . . . . . . . . . . . . . . 27

[6.2 The file structure](#6_2___The_file_structure) . . . . . . . . . . . . . . . . . . . . . . . 29

2

[6.3 Registering A Device](#6_2___The_file_structure) . . . . . . . . . . . . . . . . . . . . . 29

[6.4 Unregistering A Device](#6_4___Unregistering_A_Device) . . . . . . . . . . . . . . . . . . . 31

[6.5 chardev.c](#6_4___Unregistering_A_Device) . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

[6.6 Writing Modules for Multiple Kernel Versions](#156) . . . . . . . 35

[7 The /proc Filesystem](#156) . . . . . . . . . . . . . . . . . . . . . . . . 35

[7.1 The proc\_ops Structure](#31_________________offset____len) . . . . . . . . . . . . . . . . . . . 37

[7.2 Read and Write a /proc File](#7_2___Read_and_Write_a__proc_Fil) . . . . . . . . . . . . . . . . 38

[7.3 Manage /proc file with standard filesystem](#82) . . . . . . . . 40

[7.4 Manage /proc file with seq\_file](#60) . . . . . . . . . . . . . . . 42

[8 sysfs: Interacting with your module](#93_______endif) . . . . . . . . . . . . . . . . . 45

[9 Talking To Device Files](#Set_the_value_of_myvariable_and) . . . . . . . . . . . . . . . . . . . . . . . 48

[10 System Calls](#72___________return_0) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

[11 Blocking Processes and threads](#286) . . . . . . . . . . . . . . . . . . . 69

[11.1 Sleep](#286) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

[11.2 Completions](#11_2___Completions) . . . . . . . . . . . . . . . . . . . . . . . . . 76

[12 Synchronization](#84______MODULE_LICENSE__GPL) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

[12.1 Mutex](#84______MODULE_LICENSE__GPL) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

[12.2 Spinlocks](#24________________pr_info__mutex) . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

[12.3 Read and write locks](#which_disable_both_hardware_and) . . . . . . . . . . . . . . . . . . . . . 82

[12.4 Atomic operations](#34) . . . . . . . . . . . . . . . . . . . . . . 83

[13 Replacing Print Macros](#71______module_init_example_atom) . . . . . . . . . . . . . . . . . . . . . . . 85

[13.1 Replacement](#71______module_init_example_atom) . . . . . . . . . . . . . . . . . . . . . . . . . 85

[13.2 Flashing keyboard LEDs](#70) . . . . . . . . . . . . . . . . . . . 87

[14 GPIO](#thus_making_it_possible_to_trace) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

[14.1 GPIO](#thus_making_it_possible_to_trace) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

[14.2 Control the LED’s on/off state](#thus_making_it_possible_to_trace) . . . . . . . . . . . . . . . 90

[14.3 DHT11 sensor](#14_3___DHT11_sensor) . . . . . . . . . . . . . . . . . . . . . . . . 95

[15 Scheduling Tasks](#2______sudo_insmod_dht11_ko) . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

[15.1 Tasklets](#2______sudo_insmod_dht11_ko) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

[15.2 Work queues](#for_compatibility__For_further_i) . . . . . . . . . . . . . . . . . . . . . . . . . 102

[16 Interrupt Handlers](#for_compatibility__For_further_i) . . . . . . . . . . . . . . . . . . . . . . . . . . 102

[16.1 Interrupt Handlers](#for_compatibility__For_further_i) . . . . . . . . . . . . . . . . . . . . . . 102

[16.2 Detecting button presses](#handler__Usually_there_is_a_cert) . . . . . . . . . . . . . . . . . . . 104

[16.3 Bottom Half](#198___________gpio_free_buttons) . . . . . . . . . . . . . . . . . . . . . . . . . 108

[16.4 Threaded IRQ](#193) . . . . . . . . . . . . . . . . . . . . . . . . 112

[17 Virtual Input Device Driver](#Note_that_passing_NULL_to_both_h) . . . . . . . . . . . . . . . . . . . . . 117

[18 Standardizing the interfaces: The Device Model](#102) . . . . . . . . . . 129

[19 Device Tree](#92___________return_0) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

[19.1 Introduction to Device Tree](#92___________return_0) . . . . . . . . . . . . . . . . . 131

[19.2 Device Tree and Kernel Modules](#92___________return_0) . . . . . . . . . . . . . . 131

[19.3 Example: Device Tree Module](#__Device_specific_data__Custom_p) . . . . . . . . . . . . . . . 132

[19.4 Device Tree Source Example](#103_________________compatible) . . . . . . . . . . . . . . . . 134

[19.5 Testing Device Tree Modules](#1_________Example_device_tree_fr) . . . . . . . . . . . . . . . . 135

[19.6 Common Device Tree Functions](#1_________Example_device_tree_fr) . . . . . . . . . . . . . . 135

[20 Optimizations](#__of_property_read_bool____Check) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

[20.1 Likely and Unlikely conditions](#__of_property_read_bool____Check) . . . . . . . . . . . . . . . 136

[20.2 Static keys](#__of_property_read_bool____Check) . . . . . . . . . . . . . . . . . . . . . . . . . . 136

[21 Common Pitfalls](#1______cat__dev_key_state) . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

[21.1 Using standard libraries](#1______cat__dev_key_state) . . . . . . . . . . . . . . . . . . . 141

[21.2 Disabling interrupts](#1______cat__dev_key_state) . . . . . . . . . . . . . . . . . . . . . 141

[22 Where To Go From Here?](#1______cat__dev_key_state) . . . . . . . . . . . . . . . . . . . . . . 141

1 Introduction

The Linux Kernel Module Programming Guide is a free book; you may repro-

duce or modify it under the terms of the [Open Software License,](https://opensource.org/licenses/OSL-3.0) version 3.0.

This book is distributed in the hope that it would be useful, but without

any warranty, without even the implied warranty of merchantability or fitness for a particular purpose.

The author encourages wide distribution of this book for personal or com-

mercial use, provided the above copyright notice remains intact and the method

adheres to the provisions of the [Open Software License.](https://opensource.org/licenses/OSL-3.0) In summary, you may copy and distribute this book free of charge or for a profit. No explicit permis-sion is required from the author for reproduction of this book in any medium, physical or electronic.

Derivative works and translations of this document must be placed un-

der the Open Software License, and the original copyright notice must remain intact. If you have contributed new material to this book, you must make the material and source code available for your revisions. Please make revi-sions and updates available directly to the document maintainer, Jim Huang . This will allow for the merging of updates and provide consistent revisions to the Linux community.

If you publish or distribute this book commercially, donations, royalties, or

printed copies are greatly appreciated by the author and the [Linux Documen-](https://tldp.org/)

[tation Project](https://tldp.org/) (LDP). Contributing in this way shows your support for free software and the LDP. If you have questions or comments, please contact the address above.

1.1 Authorship

The Linux Kernel Module Programming Guide was initially authored by Ori Pomerantz for Linux v2.2. As the Linux kernel evolved, Ori’s availability to maintain the document diminished. Consequently, Peter Jay Salzman assumed the role of maintainer and updated the guide for Linux v2.4. Similar constraints arose for Peter when tracking developments in Linux v2.6, leading to Michael Burian joining as a co-maintainer to bring the guide up to speed with Linux v2.6. Bob Mottram contributed to the guide by updating examples for Linux v3.8 and later. Jim Huang then undertook the task of updating the guide for recent Linux versions (v5.0 and beyond), along with revising the LaTeX document. The guide continues to be maintained for compatibility with modern kernels (v6.x series) while ensuring examples work with older LTS kernels.

1.2 Acknowledgements

The following people have contributed corrections or good suggestions:

Amit Dhingra, Andrew Kreimer, Andrew Lin, Andy Shevchenko, Arush Sharma, Aykhan Hagverdili, Benno Bielmeier, Bob Lee, Brad Baker, Che-Chia Chang, Cheng-Shian Yeh, Cheng-Yang Chou, Chih-En Lin, Chih-Hsuan Yang, Chih-Yu Chen, Ching-Hua (Vivian) Lin, Chin Yik Ming, Chung-Han Tsai, cvvletter, Cyril Brulebois, Daniele Paolo Scarpazza, David Porter, demonsome, Dimo Velev, Ekang Monyet, Ethan Chan, Francois Audeon, Gilad Reti, Hao.Dong, heartofrain, Horst Schirmeier, Hsin-Hsiang Peng, Hung-Jen Pao, Ignacio Martin, I-Hsin Cheng, Integral, Iûnn Kiàn-îng, Jian-Xing Wu, Jimmy Ma, Johan Calle, keytouch, Kohei Otsuka, Kuan-Wei Chiu, manbing, Marconi Jiang, mengxinayan, Meng-Zong Tsai, Peter Lin, Roman Lakeev, Sam Erickson, Shao-Tse Hung, Shih-Sheng Yang, Stacy Prowell, Steven Lung, Tristan Lelong, Tse-Wei Lin, Tucker Polomik, Tyler Fanelli, VxTeemo, Wei-Hsin Yeh, Wei-Lun Tsai, Xatierlike Lee, Yan-Jie Chan, Yen-Yu Chen, Yin-Chiuan Chen, Yi-Wei Lin, Yo-Jung Lin, Yu-Chun Lin, Yu-Hsiang Tseng, YYGO.

1.3 What Is A Kernel Module?

Involvement in the development of Linux kernel modules requires a foundation in the C programming language and a track record of creating conventional programs intended for process execution. This pursuit delves into a domain where an unregulated pointer, if disregarded, may potentially trigger the total elimination of an entire filesystem, resulting in a scenario that necessitates a complete system reboot.

A Linux kernel module is precisely defined as a code segment capable of

dynamic loading and unloading within the kernel as needed. These modules enhance kernel capabilities without necessitating a system reboot. A notable example is seen in the device driver module, which facilitates kernel interaction with hardware components linked to the system. In the absence of modules, the prevailing approach leans toward monolithic kernels, requiring direct integration of new functionalities into the kernel image. This approach leads to larger ker-nels and necessitates kernel rebuilding and subsequent system rebooting when new functionalities are desired.

1.4 Kernel module package

Linux distributions provide the commands modprobe, insmod and depmod within a package.

On Ubuntu/Debian GNU/Linux:

1 sudo apt-get install build-essential kmod

On Arch Linux:

1 sudo pacman -S gcc kmod

1.5 What Modules are in my Kernel?

To discover what modules are already loaded within your current kernel, use the command lsmod.

1 lsmod

Modules are stored within the file /proc/modules, so you can also see them

with:

1 cat /proc/modules

This can be a long list, and you might prefer to search for something partic-

ular. To search for the fat module:

1 lsmod | grep fat

1.6 Is there a need to download and compile the kernel?

To effectively follow this guide, there is no obligatory requirement for performing such actions. Nonetheless, a prudent approach involves executing the examples within a test distribution on a virtual machine, thus mitigating any potential risk of disrupting the system.

1.7 Before We Begin

Before delving into code, certain matters require attention. Variances exist among individuals’ systems, and distinct personal approaches are evident. The achievement of successful compilation and loading of the inaugural “hello world” program may, at times, present challenges. It is reassuring to note that over-coming the initial obstacle on the first attempt paves the way for subsequent endeavors to proceed seamlessly.

1. Modversioning. A module compiled for one kernel will not load if a differ-

ent kernel is booted, unless CONFIG\_MODVERSIONS is enabled in the kernel. Module versioning will be discussed later in this guide. Until module versioning is covered, the examples in this guide may not work correctly if running a kernel with modversioning turned on. However, most stock Linux distribution kernels come with modversioning enabled. If difficul-ties arise when loading the modules due to versioning errors, consider compiling a kernel with modversioning turned off.

2. Using the X Window System. It is highly recommended to extract, com-

pile, and load all the examples discussed in this guide from a console. Working on these tasks within the X Window System is discouraged. Modules cannot directly print to the screen like printf() can, but they can log information and warnings to the kernel’s log ring buffer. This output is not automatically displayed on any console or terminal. To view kernel module messages, you must use dmesg to read the kernel log ring buffer, or check the systemd journal with journalctl -k for kernel

messages. Refer to [4](#1______sudo_pacman__S_linux_head) for more information. The terminal or environment from which you load the module does not affect where the output goes—it always goes to the kernel log.

3. SecureBoot. Numerous modern computers arrive pre-configured with UEFI

SecureBoot enabled—an essential security standard ensuring booting ex-clusively through trusted software endorsed by the original equipment manufacturer. Certain Linux distributions even ship with the default Linux kernel configured to support SecureBoot. In these cases, the kernel module necessitates a signed security key.

Failing that, an attempt to insert your first “hello world” module would result in the message: “ERROR: could not insert module”. If this message “Lockdown: insmod: unsigned module loading is restricted; see man kernel lockdown.7 ” appears in the dmesg output, the simplest approach involves disabling UEFI SecureBoot from the boot menu of your PC or laptop, allowing the successful insertion of the “hello world” module. Naturally, an alternative involves undergoing intricate procedures such as generating keys, system key installation, and module signing to achieve functionality. However, this intricate process is less appropriate for beginners. If inter-

ested, more detailed steps for [SecureBoot](https://wiki.debian.org/SecureBoot) can be explored and followed.

2 Headers

Before building anything, it is necessary to install the header files for the kernel.

On Ubuntu/Debian GNU/Linux:

1 sudo apt-get update

2 apt-cache search linux-headers-`uname -r`

The following command provides information about the available kernel

header files. Then, for example:

1 sudo apt-get install linux-headers-`uname -r`

On Arch Linux:

1 sudo pacman -S linux-headers

On Fedora:

1 sudo dnf install kernel-devel kernel-headers

3 Examples

All the examples from this document are available within the examples subdi-rectory.

Should compile errors occur, it may be due to a more recent kernel version

being in use, or there might be a need to install the corresponding kernel header files.

4 Hello World

4.1 The Simplest Module

Most individuals beginning their programming journey typically start with some variant of a hello world example. It is unclear what the outcomes are for those who deviate from this tradition, but it seems prudent to adhere to it. The learning process will begin with a series of hello world programs that illustrate various fundamental aspects of writing a kernel module.

Presented next is the simplest possible module.

Make a test directory:

1 mkdir -p ~/develop/kernel/hello-1 2 cd ~/develop/kernel/hello-1

Paste this into your favorite editor and save it as hello-1.c:

1 /\*

2 \* hello-1.c - The simplest kernel module. 3 \*/

4 #include /\* Needed by all modules \*/ 5 #include /\* Needed for pr\_info() \*/ 6

7 int init\_module(void)

8 {

9 pr\_info("Hello world 1.\n");

10

11 /\* A nonzero return means init\_module failed; module can't be loaded. \*/

12 return 0;

13 }

14

15 void cleanup\_module(void)

16 {

17 pr\_info("Goodbye world 1.\n");

18 }

19

20 MODULE\_LICENSE("GPL");

Now you will need a Makefile. If you copy and paste this, change the

indentation to use tabs, not spaces.

1 obj-m += hello-1.o

2

3 PWD := $(CURDIR)

4

5 all:

6 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules 7

8 clean:

9 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

In Makefile, $(CURDIR) can be set to the absolute pathname of the current

working directory (after all-C options are processed, if any). See more about

CURDIR in [GNU make manual.](https://www.gnu.org/software/make/manual/make.html)

And finally, just run make directly.

1 make

If there is no PWD := $(CURDIR) statement in the Makefile, then it may not

compile correctly with sudo make. This is because some environment variables are specified by the security policy and cannot be inherited. The default security policy is sudoers. In the sudoers security policy, env\_reset is enabled by default, which restricts environment variables. Specifically, path variables are not retained from the user environment; they are set to default values (for

more information, see: [sudoers manual).](https://www.sudo.ws/docs/man/sudoers.man/) You can see the environment variable settings by:

$ sudo -s

# sudo -V

Here is a simple Makefile as an example to demonstrate the problem men-

tioned above.

1 all:

2 echo $(PWD)

Then, we can use the-p flag to print out the environment variable values

from the Makefile.

$ make -p | grep PWD

PWD = /home/ubuntu/temp

OLDPWD = /home/ubuntu

echo $(PWD)

The PWD variable will not be inherited with sudo.

$ sudo make -p | grep PWD

echo $(PWD)

However, there are three ways to solve this problem.

1. You can use the-E flag to temporarily preserve them.

1 $ sudo -E make -p | grep PWD 2 PWD = /home/ubuntu/temp 3 OLDPWD = /home/ubuntu 4 echo $(PWD)

2. You can disable env\_reset by editing /etc/sudoers as root using visudo.

1 ## sudoers file.

2 ##

3 ...

4 Defaults env\_reset

5 ## Change env\_reset to !env\_reset in previous line to keep all

, → environment variables

Then execute env and sudo env individually.

1 # disable the env\_reset 2 echo "user:" > non-env\_reset.log; env >>

,→ non-env\_reset.log

3 echo "root:" >> non-env\_reset.log; sudo env >>

,→ non-env\_reset.log

4 # enable the env\_reset 5 echo "user:" > env\_reset.log; env >> env\_reset.log 6 echo "root:" >> env\_reset.log; sudo env >>

,→ env\_reset.log

You can view and compare these logs to find differences between env\_reset and !env\_reset.

3. You can preserve environment variables by appending them to env\_keep

in /etc/sudoers.

1 Defaults env\_keep += "PWD"

After applying the above change, you can check the environment variable settings by:

$ sudo -s

# sudo -V

If all goes smoothly you should then find that you have a compiled hello-1.ko

module. You can find info on it with the command:

1 modinfo hello-1.ko

At this point the command:

1 lsmod | grep hello

should return nothing. You can try loading your new module with:

1 sudo insmod hello-1.ko

The dash character will get converted to an underscore, so when you again

try:

1 lsmod | grep hello

You should now see your loaded module. It can be removed again with:

1 sudo rmmod hello\_1

Notice that the dash was replaced by an underscore. To see the module’s

output messages, use dmesg to view the kernel log ring buffer:

1 sudo dmesg | tail -10

You should see messages like “Hello world 1.” and “Goodbye world 1.” from

your module. Alternatively, you can check the systemd journal for kernel mes-sages:

1 journalctl --since "1 hour ago" | grep kernel

You now know the basics of creating, compiling, installing and removing

modules. Now for more of a description of how this module works.

Kernel modules must have at least two functions: a "start" (initialization)

function called init\_module() which is called when the module is insmoded into the kernel, and an "end" (cleanup) function called cleanup\_module() which is called just before it is removed from the kernel. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start

and end functions of a module, and you will learn how to do this in Section [4.2.](#the_standard_kernel_build_mechan) In fact, the new method is the preferred method. However, many people still use init\_module() and cleanup\_module() for their start and end functions.

Typically, init\_module() either registers a handler for something with the

kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The cleanup\_module() function is supposed to undo whatever init\_module() did, so the module can be unloaded safely.

Lastly, every kernel module needs to include . We needed

to include only for the macro expansion for the pr\_alert()

log level, which you’ll learn about in Section [2.](#1______journalctl___since__1_hou)

1. A point about coding style. Another thing that may not be immediately

obvious to anyone getting started with kernel programming is that inden-tation within your code should use tabs and not spaces. It is one of the coding conventions of the kernel. You may not like it, but you will need to get used to it if you ever submit a patch upstream.

2. Introducing print macros. In the beginning there was printk, usually fol-

lowed by a priority such as KERN\_INFO or KERN\_DEBUG. More recently this can also be expressed in abbreviated form using a set of print macros, such as pr\_info and pr\_debug. This just saves some mindless key-

board bashing and looks a bit neater. They can be found within [in-](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/printk.h)

[clude/linux/printk.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/printk.h) Take time to read through the available priority macros.

Important: These functions write to the kernel log ring buffer, not di-rectly to any terminal or console. To view the output from your kernel modules, you must use dmesg or journalctl -k.

3. About Compiling. Kernel modules need to be compiled a bit differently

from regular userspace apps. Former kernel versions required us to care much about these settings, which are usually stored in Makefiles. Al-though hierarchically organized, many redundant settings accumulated in sublevel Makefiles and made them large and rather difficult to maintain. Fortunately, there is a new way of doing these things, called kbuild, and the build process for external loadable modules is now fully integrated into

the standard kernel build mechanism. To learn more on how to compile modules which are not part of the official kernel (such as all the examples

you will find in this guide), see file [Documentation/kbuild/modules.rst.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/kbuild/modules.rst)

Additional details about Makefiles for kernel modules are available in [Doc-](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/kbuild/makefiles.rst)

[umentation/kbuild/makefiles.rst.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/kbuild/makefiles.rst) Be sure to read this and the related files before starting to hack Makefiles. It will probably save you lots of work.

Here is another exercise for the reader. See that comment above

the return statement in init\_module()? Change the return value to something negative, recompile and load the module

again. What happens?

4.2 Hello and Goodbye

In early kernel versions you had to use the init\_module and cleanup\_module functions, as in the first hello world example, but these days you can name those anything you want by using the module\_init and module\_exit macros. These

macros are defined in [include/linux/module.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/module.h) The only requirement is that your init and cleanup functions must be defined before calling those macros, otherwise you will get compilation errors. Here is an example of this technique:

1 /\*

2 \* hello-2.c - Demonstrating the module\_init() and module\_exit() macros. 3 \* This is preferred over using init\_module() and cleanup\_module(). 4 \*/

5 #include /\* Needed for the macros \*/ 6 #include /\* Needed by all modules \*/ 7 #include /\* Needed for pr\_info() \*/ 8

9 static int \_\_init hello\_2\_init(void)

10 {

11 pr\_info("Hello, world 2\n");

12 return 0;

13 }

14

15 static void \_\_exit hello\_2\_exit(void) 16 {

17 pr\_info("Goodbye, world 2\n");

18 }

19

20 module\_init(hello\_2\_init);

21 module\_exit(hello\_2\_exit);

22

23 MODULE\_LICENSE("GPL");

So now we have two real kernel modules under our belt. Adding another

module is as simple as this:

1 obj-m += hello-1.o

2 obj-m += hello-2.o

3

4 PWD := $(CURDIR)

5

6 all:

7 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules 8

9 clean:

10 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Now have a look at [drivers/char/Makefile](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/char/Makefile) for a real world example. As you

can see, some things got hardwired into the kernel (obj-y) but where have all those obj-m gone? Those familiar with shell scripts will easily be able to spot them. For those who are not, the obj-$(CONFIG\_FOO) entries you see everywhere expand into obj-y or obj-m, depending on whether the CONFIG\_FOO variable has been set to y or m. While we are at it, those were exactly the kind of variables that you have set in the .config file in the top-level directory of the Linux kernel source tree, the last time you ran make menuconfig or something similar.

4.3 The \_\_init and \_\_exit Macros

The \_\_init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules. If you think about when the init function is invoked, this makes perfect sense.

There is also an \_\_initdata which works similarly to \_\_init but for init

variables rather than functions.

The \_\_exit macro causes the omission of the function when the module

is built into the kernel, and like \_\_init, has no effect for loadable modules. Again, if you consider when the cleanup function runs, this makes complete sense; built-in drivers do not need a cleanup function, while loadable modules do.

These macros are defined in [include/linux/init.h](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/init.h) and serve to free up kernel

memory. When you boot your kernel and see something like Freeing unused kernel memory: 236k freed, this is precisely what the kernel is freeing.

1 /\*

2 \* hello-3.c - Illustrating the \_\_init, \_\_initdata and \_\_exit macros. 3 \*/

4 #include /\* Needed for the macros \*/ 5 #include /\* Needed by all modules \*/ 6 #include /\* Needed for pr\_info() \*/ 7

8 static int hello3\_data \_\_initdata = 3; 9

10 static int \_\_init hello\_3\_init(void) 11 {

12 pr\_info("Hello, world %d\n", hello3\_data); 13 return 0;

14 }

15

16 static void \_\_exit hello\_3\_exit(void) 17 {

18 pr\_info("Goodbye, world 3\n");

19 }

20

21 module\_init(hello\_3\_init);

22 module\_exit(hello\_3\_exit);

23

24 MODULE\_LICENSE("GPL");

4.4 Licensing and Module Documentation

Honestly, who loads or even cares about proprietary modules? If you do then you might have seen something like this:

$ sudo insmod xxxxxx.ko

loading out-of-tree module taints kernel.

module license 'unspecified' taints kernel.

You can use a few macros to indicate the license for your module. Some ex-

amples are "GPL", "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MIT/GPL", "Dual MPL/GPL" and "Proprietary". They are defined

within [include/linux/module.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/module.h)

To reference what license you are using, a macro is available called MODULE\_LICENSE.

This and a few other macros describing the module are illustrated in the below example.

1 /\*

2 \* hello-4.c - Demonstrates module documentation. 3 \*/

4 #include /\* Needed for the macros \*/ 5 #include /\* Needed by all modules \*/ 6 #include /\* Needed for pr\_info() \*/ 7

8 MODULE\_LICENSE("GPL");

9 MODULE\_AUTHOR("LKMPG");

10 MODULE\_DESCRIPTION("A sample driver"); 11

12 static int \_\_init init\_hello\_4(void) 13 {

14 pr\_info("Hello, world 4\n");

15 return 0;

16 }

17

18 static void \_\_exit cleanup\_hello\_4(void) 19 {

20 pr\_info("Goodbye, world 4\n");

21 }

22

23 module\_init(init\_hello\_4);

24 module\_exit(cleanup\_hello\_4);

4.5 Passing Command Line Arguments to a Module

Modules can take command line arguments, but not with the argc/argv you might be used to.

To allow arguments to be passed to your module, declare the variables that

will take the values of the command line arguments as global and then use the

module\_param() macro, (defined in [include/linux/moduleparam.h)](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/moduleparam.h) to set the mechanism up. At runtime, insmod will fill the variables with any command line arguments that are given, like insmod mymodule.ko myvariable=5. The vari-able declarations and macros should be placed at the beginning of the module for clarity. The example code should clear up my admittedly lousy explanation.

The module\_param() macro takes 3 arguments: the name of the variable,

its type and permissions for the corresponding file in sysfs. Integer types can be signed as usual or unsigned. If you would like to use arrays of integers or strings, see module\_param\_array() and module\_param\_string().

1 int myint = 3;

2 module\_param(myint, int, 0);

Arrays are supported too, but things are a bit different now than they were

in the olden days. To keep track of the number of parameters you need to pass a pointer to a count variable as third parameter. At your option, you could also ignore the count and pass NULL instead. We show both possibilities here:

1 int myintarray[2];

2 module\_param\_array(myintarray, int, NULL, 0); /\* not interested in count \*/ 3

4 short myshortarray[4];

5 int count;

6 module\_param\_array(myshortarray, short, &count, 0); /\* put count into "count"

, → variable \*/

A good use for this is to have the module variable’s default values set, like

a port or IO address. If the variables contain the default values, then perform autodetection (explained elsewhere). Otherwise, keep the current value. This will be made clear later on.

Lastly, there is a macro function, MODULE\_PARM\_DESC(), that is used to

document arguments that the module can take. It takes two parameters: a variable name and a free form string describing that variable.

1 /\*

2 \* hello-5.c - Demonstrates command line argument passing to a module. 3 \*/

4 #include

5 #include /\* for ARRAY\_SIZE() \*/ 6 #include

7 #include

8 #include

9 #include

10

11 MODULE\_LICENSE("GPL");

12

13 static short int myshort = 1;

14 static int myint = 420;

15 static long int mylong = 9999;

16 static char \*mystring = "blah";

17 static int myintarray[2] = { 420, 420 }; 18 static int arr\_argc = 0;

19

20 /\* module\_param(foo, int, 0000)

21 \* The first param is the parameter's name. 22 \* The second param is its data type. 23 \* The final argument is the permissions bits, 24 \* for exposing parameters in sysfs (if non-zero) at a later stage. 25 \*/

26 module\_param(myshort, short, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP); 27 MODULE\_PARM\_DESC(myshort, "A short integer"); 28 module\_param(myint, int, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH); 29 MODULE\_PARM\_DESC(myint, "An integer"); 30 module\_param(mylong, long, S\_IRUSR); 31 MODULE\_PARM\_DESC(mylong, "A long integer"); 32 module\_param(mystring, charp, 0000); 33 MODULE\_PARM\_DESC(mystring, "A character string"); 34

35 /\* module\_param\_array(name, type, num, perm); 36 \* The first param is the parameter's (in this case the array's) name. 37 \* The second param is the data type of the elements of the array. 38 \* The third argument is a pointer to the variable that will store the number

39 \* of elements of the array initialized by the user at module loading time.

40 \* The fourth argument is the permission bits. 41 \*/

42 module\_param\_array(myintarray, int, &arr\_argc, 0000); 43 MODULE\_PARM\_DESC(myintarray, "An array of integers"); 44

45 static int \_\_init hello\_5\_init(void) 46 {

47 int i;

48

49 pr\_info("Hello, world 5\n=============\n"); 50 pr\_info("myshort is a short integer: %hd\n", myshort); 51 pr\_info("myint is an integer: %d\n", myint); 52 pr\_info("mylong is a long integer: %ld\n", mylong); 53 pr\_info("mystring is a string: %s\n", mystring); 54

55 for (i = 0; i < ARRAY\_SIZE(myintarray); i++) 56 pr\_info("myintarray[%d] = %d\n", i, myintarray[i]); 57

58 pr\_info("got %d arguments for myintarray.\n", arr\_argc); 59 return 0;

60 }

61

62 static void \_\_exit hello\_5\_exit(void) 63 {

64 pr\_info("Goodbye, world 5\n");

65 }

66

67 module\_init(hello\_5\_init);

68 module\_exit(hello\_5\_exit);

It is recommended to experiment with the following code:

$ sudo insmod hello-5.ko mystring="bebop" myintarray=-1 $ sudo dmesg -t | tail -7

myshort is a short integer: 1

myint is an integer: 420

mylong is a long integer: 9999

mystring is a string: bebop

myintarray[0] = -1

myintarray[1] = 420

got 1 arguments for myintarray.

$ sudo rmmod hello-5

$ sudo dmesg -t | tail -1

Goodbye, world 5

$ sudo insmod hello-5.ko mystring="supercalifragilisticexpialidocious" myintarray=-1,-1 $ sudo dmesg -t | tail -7

myshort is a short integer: 1

myint is an integer: 420

mylong is a long integer: 9999

mystring is a string: supercalifragilisticexpialidocious myintarray[0] = -1

myintarray[1] = -1

got 2 arguments for myintarray.

$ sudo rmmod hello-5

$ sudo dmesg -t | tail -1

Goodbye, world 5

$ sudo insmod hello-5.ko mylong=hello

insmod: ERROR: could not insert module hello-5.ko: Invalid parameters

4.6 Modules Spanning Multiple Files

Sometimes it makes sense to divide a kernel module between several source files.

Here is an example of such a kernel module.

1 /\*

2 \* start.c - Illustration of multi filed modules 3 \*/

4

5 #include /\* We are doing kernel work \*/

6 #include /\* Specifically, a module \*/ 7

8 int init\_module(void)

9 {

10 pr\_info("Hello, world - this is the kernel speaking\n"); 11 return 0;

12 }

13

14 MODULE\_LICENSE("GPL");

The next file:

1 /\*

2 \* stop.c - Illustration of multi filed modules 3 \*/

4

5 #include /\* We are doing kernel work \*/ 6 #include /\* Specifically, a module \*/ 7

8 void cleanup\_module(void)

9 {

10 pr\_info("Short is the life of a kernel module\n"); 11 }

12

13 MODULE\_LICENSE("GPL");

And finally, the makefile:

1 obj-m += hello-1.o

2 obj-m += hello-2.o

3 obj-m += hello-3.o

4 obj-m += hello-4.o

5 obj-m += hello-5.o

6 obj-m += startstop.o

7 startstop-objs := start.o stop.o

8

9 PWD := $(CURDIR)

10

11 all:

12 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules 13

14 clean:

15 $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

This is the complete makefile for all the examples we have seen so far. The

first five lines are nothing special, but for the last example we will need two lines. First we invent an object name for our combined module, second we tell make what object files are part of that module.

4.7 Building modules for a precompiled kernel

Obviously, we strongly suggest you to recompile your kernel, so that you can enable a number of useful debugging features, such as forced module unloading

(MODULE\_FORCE\_UNLOAD): when this option is enabled, you can force the kernel to unload a module even when it believes it is unsafe, via a sudo rmmod -f module command. This option can save you a lot of time and a number of reboots dur-ing the development of a module. If you do not want to recompile your kernel then you should consider running the examples within a test distribution on a virtual machine. If you mess anything up then you can easily reboot or restore the virtual machine (VM).

There are a number of cases in which you may want to load your module

into a precompiled running kernel, such as the ones shipped with common Linux distributions, or a kernel you have compiled in the past. In certain circumstances you could require to compile and insert a module into a running kernel which you are not allowed to recompile, or on a machine that you prefer not to reboot. If you can’t think of a case that will force you to use modules for a precompiled kernel you might want to skip this and treat the rest of this chapter as a big footnote.

Now, if you just install a kernel source tree, use it to compile your kernel

module and you try to insert your module into the kernel, in most cases you would obtain an error as follows:

insmod: ERROR: could not insert module poet.ko: Invalid module format

Less cryptic information is logged to the systemd journal:

kernel: poet: disagrees about version of symbol module\_layout

In other words, your kernel refuses to accept your module because version

strings (more precisely, version magic, see [include/linux/vermagic.h)](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/vermagic.h) do not match. Incidentally, version magic strings are stored in the module object in the form of a static string, starting with vermagic:. Version data are inserted in your module when it is linked against the kernel/module.o file. To inspect version magics and other strings stored in a given module, issue the command modinfo module.ko:

$ modinfo hello-4.ko

description: A sample driver author: LKMPG

license: GPL

srcversion: B2AA7FBFCC2C39AED665382 depends:

retpoline: Y

name: hello\_4

vermagic: 5.4.0-70-generic SMP mod\_unload modversions

To overcome this problem we could resort to the--force-vermagic op-

tion, but this solution is potentially unsafe, and unquestionably unacceptable in production modules. Consequently, we want to compile our module in an environment which was identical to the one in which our precompiled kernel was built. How to do this, is the subject of the remainder of this chapter.

First of all, make sure that a kernel source tree is available, having exactly

the same version as your current kernel. Then, find the configuration file which was used to compile your precompiled kernel. Usually, this is available in your current boot directory, under a name like config-5.14.x. You may just want to copy it to your kernel source tree: cp /boot/config-`uname -r` .config.

Let’s focus again on the previous error message: a closer look at the version

magic strings suggests that, even with two configuration files which are exactly the same, a slight difference in the version magic could be possible, and it is sufficient to prevent insertion of the module into the kernel. That slight difference, namely the custom string which appears in the module’s version magic and not in the kernel’s one, is due to a modification with respect to the original, in the makefile that some distributions include. Then, examine your Makefile, and make sure that the specified version information matches exactly the one used for your current kernel. For example, your makefile could start as follows:

VERSION = 5

PATCHLEVEL = 14

SUBLEVEL = 0

EXTRAVERSION = -rc2

In this case, you need to restore the value of symbol EXTRAVERSION

to-rc2. We suggest keeping a backup copy of the makefile used to compile your kernel available in /lib/modules/5.14.0-rc2/build. A simple command as follows should suffice.

1 cp /lib/modules/`uname -r`/build/Makefile linux-`uname -r`

Here linux-`uname -r` is the Linux kernel source you are attempting to build.

Now, please run make to update configuration and version headers and ob-

jects:

$ make

SYNC include/config/auto.conf.cmd HOSTCC scripts/basic/fixdep

HOSTCC scripts/kconfig/conf.o

HOSTCC scripts/kconfig/confdata.o

HOSTCC scripts/kconfig/expr.o

LEX scripts/kconfig/lexer.lex.c YACC scripts/kconfig/parser.tab.[ch] HOSTCC scripts/kconfig/preprocess.o

HOSTCC scripts/kconfig/symbol.o

HOSTCC scripts/kconfig/util.o

HOSTCC scripts/kconfig/lexer.lex.o

HOSTCC scripts/kconfig/parser.tab.o

HOSTLD scripts/kconfig/conf

If you do not desire to actually compile the kernel, you can interrupt the

build process (CTRL-C) just after the SPLIT line, because at that time, the files you need are ready. Now you can turn back to the directory of your module and compile it: It will be built exactly according to your current kernel settings, and it will load into it without any errors.

5 Preliminaries

5.1 How modules begin and end

A typical program starts with a main() function, executes a series of instruc-tions, and terminates after completing these instructions. Kernel modules, however, follow a different pattern. A module always begins with either the init\_module function or a function designated by the module\_init call. This function acts as the module’s entry point, informing the kernel of the module’s functionalities and preparing the kernel to utilize the module’s functions when necessary. After performing these tasks, the entry function returns, and the module remains inactive until the kernel requires its code.

All modules conclude by invoking either cleanup\_module or a function spec-

ified through the module\_exit call. This serves as the module’s exit function, reversing the actions of the entry function by unregistering the previously reg-istered functionalities.

It is mandatory for every module to have both an entry and an exit function.

While there are multiple methods to define these functions, the terms “entry function” and “exit function” are generally used. However, they may occasionally be referred to as init\_module and cleanup\_module, which are understood to mean the same.

5.2 Functions available to modules

Programmers use functions they do not define all the time. A prime example of this is printf(). You use these library functions which are provided by the standard C library, libc. The definitions for these functions do not actually enter your program until the linking stage, which ensures that the code (for printf() for example) is available, and fixes the call instruction to point to that code.

Kernel modules are different here, too. In the hello world example, you

might have noticed that we used a function, pr\_info() but did not include a standard I/O library. That is because modules are object files whose symbols get resolved upon running insmod or modprobe. The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you’re curious about what symbols have been exported by your kernel, take a look at /proc/kallsyms.

One point to keep in mind is the difference between library functions and

system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions

that do the real work — system calls. System calls run in kernel mode on the user’s behalf and are provided by the kernel itself. The library function printf() may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call write(), which then sends the data to standard output.

Would you like to see what system calls are made by printf()? It is easy!

Compile the following program:

1 #include

2

3 int main(void)

4 {

5 printf("hello");

6 return 0;

7 }

with gcc -Wall -o hello hello.c. Run the executable with strace ./hello.

Are you impressed? Every line you see corresponds to a system call. [strace](https://strace.io/) is a handy program that gives you details about what system calls a program is making, including which call is made, what its arguments are and what it returns. It is an invaluable tool for figuring out things like what files a pro-gram is trying to access. Towards the end, you will see a line which looks like write(1, "hello", 5hello). There it is. The face behind the printf() mask. You may not be familiar with write, since most people use library func-tions for file I/O (like fopen, fputs, fclose). If that is the case, try looking at man 2 write. The 2nd man section is devoted to system calls (like kill() and read()). The 3rd man section is devoted to library calls, which you would probably be more familiar with (like cosh() and random()).

You can even write modules to replace the kernel’s system calls, which we

will do shortly. Crackers often make use of this sort of thing for backdoors or trojans, but you can write your own modules to do more benign things, like have the kernel log a message whenever someone attempts to delete a file on your system.

5.3 User Space vs Kernel Space

The kernel primarily manages access to resources, be it a video card, hard drive, or memory. Programs frequently vie for the same resources. For instance, as a document is saved, updatedb might commence updating the locate database. Sessions in editors like vim and processes like updatedb can simultaneously utilize the hard drive. The kernel’s role is to maintain order, ensuring that users do not access resources indiscriminately.

To manage this, CPUs operate in different modes, each offering varying levels

of system control. The Intel 80386 architecture, for example, featured four such modes, known as rings. Unix, however, utilizes only two of these rings: the highest ring (ring 0, also known as “supervisor mode”, where all actions are permissible) and the lowest ring, referred to as “user mode”.

Recall the discussion about library functions vs system calls. Typically, you

use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function’s behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transferred back to user mode.

5.4 Name Space

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you are writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples’ global variables; some of the variable names can clash. When a program has lots of global variables which aren’t meaningful enough to be distinguished, you get namespace pollution. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols.

When writing kernel code, even the smallest module will be linked against

the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you do not want to declare everything as static, another option is to declare a symbol table and register it with the kernel. We will get to this later.

The file /proc/kallsyms holds all the symbols that the kernel knows about

and which are therefore accessible to your modules since they share the kernel’s codespace.

5.5 Code space

Memory management is a very complicated subject and the majority of O’Reilly’s

[Understanding The Linux Kernel](https://www.oreilly.com/library/view/understanding-the-linux/0596005652/) exclusively covers memory management! We are not setting out to be experts on memory management, but we do need to know a couple of facts to even begin worrying about writing real modules.

If you have not thought about what a segfault really means, you may be

surprised to hear that pointers do not actually point to memory locations. Not real ones, anyway. When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things which a computer scientist would know about. This memory begins with 0x00000000 and extends up to whatever it needs to be. Since the memory space for any two processes does not overlap, every process that can access a memory address, say 0xbffff978, would be ac-cessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a pro-cess like our Hello, World program cannot access the space of another process, although there are ways which we will talk about later.

The kernel has its own space of memory as well. Since a module is code

which can be dynamically inserted and removed in the kernel (as opposed to a semi-autonomous object), it shares the kernel’s codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data because of an off-by-one error, then you’re trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful.

It should be noted that the aforementioned discussion applies to any oper-

ating system utilizing a monolithic kernel. This concept differs slightly from “building all your modules into the kernel” , although the underlying principle is similar. In contrast, there are microkernels, where modules are allocated their

own code space. Two notable examples of microkernels include the [GNU Hurd](https://www.gnu.org/software/hurd/)

and the [Zircon kernel](https://fuchsia.dev/fuchsia-src/concepts/kernel) of Google’s Fuchsia.

5.6 Device Drivers

One class of module is the device driver, which provides functionality for hard-ware like a serial port. On Unix, each piece of hardware is represented by a file located in /dev named a device file which provides the means to communicate with the hardware. The device driver provides the communication on behalf of a user program. So the es1370.ko sound card device driver might connect the /dev/sound device file to the Ensoniq ES1370 sound card. A userspace program like mp3blaster can use /dev/sound without ever knowing what kind of sound card is installed.

Let’s look at some device files. Here are device files which represent the first

three partitions on the primary SCSI storage devices:

$ ls -l /dev/sda[1-3]

brw-rw---- 1 root disk 8, 1 Apr 9 2025 /dev/sda1 brw-rw---- 1 root disk 8, 2 Apr 9 2025 /dev/sda2 brw-rw---- 1 root disk 8, 3 Apr 9 2025 /dev/sda3

Notice the column of numbers separated by a comma. The first number is

called the device’s major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 8, because they’re all controlled by the same driver.

The minor number is used by the driver to distinguish between the various

hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.

Devices are divided into two types: character devices and block devices. The

difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it is faster to read or write sectors which are close

to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don’t need this type of buffering, and they don’t operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of ls -l. If it is ‘b’ then it is a block device, and if it is ‘c’ then it is a character device. The devices you see above are block devices. Here are some character devices (the serial ports):

crw-rw---- 1 root dial 4, 64 Feb 18 23:34 /dev/ttyS0 crw-r----- 1 root dial 4, 65 Nov 17 10:26 /dev/ttyS1 crw-rw---- 1 root dial 4, 66 Jul 5 2000 /dev/ttyS2 crw-rw---- 1 root dial 4, 67 Jul 5 2000 /dev/ttyS3

If you want to see which major numbers have been assigned, you can look

at [Documentation/admin-guide/devices.txt.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/devices.txt)

When the system was installed, all of those device files were created by the

mknod command. To create a new char device named coffee with major/minor number 12 and 2, simply do mknod /dev/coffee c 12 2. You do not have to put your device files into /dev, but it is done by convention. Linus put his device files in /dev, and so should you. However, when creating a device file for testing purposes, it is probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you’re done writing the device driver.

A few final points, although implicit in the previous discussion, are worth

stating explicitly for clarity. When a device file is accessed, the kernel utilizes the file’s major number to identify the appropriate driver for handling the access. This indicates that the kernel does not necessarily rely on or need to be aware of the minor number. It is the driver that concerns itself with the minor number, using it to differentiate between various pieces of hardware.

It is important to note that when referring to “hardware”, the term is used

in a slightly more abstract sense than just a physical PCI card that can be held in hand. Consider the following two device files:

$ ls -l /dev/sda /dev/sdb

brw-rw---- 1 root disk 8, 0 Jan 3 09:02 /dev/sda

brw-rw---- 1 root disk 8, 16 Jan 3 09:02 /dev/sdb

By now you can look at these two device files and know instantly that they

are block devices and are handled by same driver (block major 8). Sometimes two device files with the same major but different minor number can actually represent the same piece of physical hardware. So just be aware that the word “hardware” in our discussion can mean something very abstract.

6 Character Device drivers

6.1 The file\_operations Structure

The file\_operations structure is defined in [include/linux/fs.h,](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/fs.h) and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some func-tion defined by the driver to handle a requested operation.

For example, every character driver needs to define a function that reads from

the device. The file\_operations structure holds the address of the module’s function that performs that operation. Here is what the definition looks like for kernel 5.4 and later versions:

1 struct file\_operations {

2 struct module \*owner;

3 loff\_t (\*llseek) (struct file \*, loff\_t, int); 4 ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*); 5 ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*); 6 ssize\_t (\*read\_iter) (struct kiocb \*, struct iov\_iter \*); 7 ssize\_t (\*write\_iter) (struct kiocb \*, struct iov\_iter \*); 8 int (\*iopoll)(struct kiocb \*kiocb, bool spin); 9 int (\*iterate) (struct file \*, struct dir\_context \*);

10 int (\*iterate\_shared) (struct file \*, struct dir\_context \*); 11 \_\_poll\_t (\*poll) (struct file \*, struct poll\_table\_struct \*); 12 long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long); 13 long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long); 14 int (\*mmap) (struct file \*, struct vm\_area\_struct \*); 15 unsigned long mmap\_supported\_flags; 16 int (\*open) (struct inode \*, struct file \*); 17 int (\*flush) (struct file \*, fl\_owner\_t id); 18 int (\*release) (struct inode \*, struct file \*); 19 int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync); 20 int (\*fasync) (int, struct file \*, int); 21 int (\*lock) (struct file \*, int, struct file\_lock \*); 22 ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*,

, → int);

23 unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned

,→ long, unsigned long, unsigned long);

24 int (\*check\_flags)(int);

25 int (\*flock) (struct file \*, int, struct file\_lock \*); 26 ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*,

, → size\_t, unsigned int);

27 ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*,

,→ size\_t, unsigned int);

28 int (\*setlease)(struct file \*, long, struct file\_lock \*\*, void \*\*); 29 long (\*fallocate)(struct file \*file, int mode, loff\_t offset, 30 loff\_t len);

31 void (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f); 32 ssize\_t (\*copy\_file\_range)(struct file \*, loff\_t, struct file \*, 33 loff\_t, size\_t, unsigned int); 34 loff\_t (\*remap\_file\_range)(struct file \*file\_in, loff\_t pos\_in, 35 struct file \*file\_out, loff\_t pos\_out, 36 loff\_t len, unsigned int remap\_flags); 37 int (\*fadvise)(struct file \*, loff\_t, loff\_t, int);

38 } \_\_randomize\_layout;

Some operations are not implemented by a driver. For example, a driver

that handles a video card will not need to read from a directory structure. The corresponding entries in the file\_operations structure should be set to NULL.

[1](#38__________randomize_layout)

There is a gcc extension that makes assigning to this structure more conve-

nient. You will see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

1 struct file\_operations fops = {

2 read: device\_read,

3 write: device\_write,

4 open: device\_open,

5 release: device\_release

6 };

However, there is also a C99 way of assigning to elements of a structure,

[designated initializers,](https://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html) and this is definitely preferred over using the GNU ex-tension. You should use this syntax in case someone wants to port your driver. It will help with compatibility:

1 struct file\_operations fops = {

2 .read = device\_read,

3 .write = device\_write,

4 .open = device\_open,

5 .release = device\_release 6 };

The meaning is clear, and you should be aware that any member of the

structure which you do not explicitly assign will be initialized to NULL by gcc.

An instance of struct file\_operations containing pointers to functions

that are used to implement read, write, open, . . . system calls is commonly named fops.

Since Linux v3.14, the read, write and seek operations are guaranteed for

thread-safe by using the f\_pos specific lock, which makes the file position update to become the mutual exclusion. So, we can safely implement those operations without unnecessary locking.

Additionally, since Linux v5.6, the proc\_ops structure was introduced to re-

place the use of the file\_operations structure when registering proc handlers.

See more information in the [7.1](#31_________________offset____len) section.

1 As of Linux kernel 6.12, several member fields have been added, removed, or had their

prototypes changed. For example, additions include fop\_flags, splice\_eof, and uring\_cmd; removals include iterate and sendpage; and the prototype for iopoll was modified.

6.2 The file structure

Each device is represented in the kernel by a file structure, which is defined in

[include/linux/fs.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/fs.h) Be aware that a file is a kernel level structure and never appears in a user space program. It is not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open ‘file’, not a file on a disk, which is represented by a structure named inode.

An instance of struct file is commonly named filp. You’ll also see it referred

to as a struct file object. Resist the temptation.

Go ahead and look at the definition of file. Most of the entries you see, like

struct dentry, are not used by device drivers, and you can ignore them. This is because drivers do not fill file directly; they only use structures contained in file which are created elsewhere.

6.3 Registering A Device

As discussed earlier, char devices are accessed through device files, usually lo-cated in /dev. This is by convention. When writing a driver, it is OK to put the device file in your current directory. Just make sure you place it in /dev for a production driver. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it is operating on, just in case the driver handles more than one device.

Adding a driver to your system means registering it with the kernel. This

is synonymous with assigning it a major number during the module’s initial-

ization. You do this by using the register\_chrdev function, defined by [in-](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/fs.h)

[clude/linux/fs.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/fs.h)

1 int register\_chrdev(unsigned int major, const char \*name, struct

, → file\_operations \*fops);

Where unsigned int major is the major number you want to request,

const char \*name is the name of the device as it will appear in /proc/devices and struct file\_operations \*fops is a pointer to the file\_operations ta-ble for your driver. A negative return value means the registration failed. Note that we didn’t pass the minor number to register\_chrdev. That is because the kernel doesn’t care about the minor number; only our driver uses it.

Now the question is, how do you get a major number without hijacking one

that’s already in use? The easiest way would be to look through [Documentation/admin-](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/devices.txt)

[guide/devices.txt](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/devices.txt) and pick an unused one. That is a bad way of doing things because you will never be sure if the number you picked will be assigned later. The answer is that you can ask the kernel to assign you a dynamic major num-ber.

If you pass a major number of 0 to register\_chrdev, the return value will

be the dynamically allocated major number. The downside is that you can not

make a device file in advance, since you do not know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in /proc/devices, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is that we can have our driver make the device file using the device\_create function after a successful registration and device\_destroy during the call to cleanup\_module.

However, register\_chrdev() would occupy a range of minor numbers as-

sociated with the given major. The recommended way to reduce waste for char device registration is using cdev interface.

The newer interface completes the char device registration in two distinct

steps. First, we should register a range of device numbers, which can be com-pleted with register\_chrdev\_region or alloc\_chrdev\_region.

1 int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name); 2 int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, const

, → char \*name);

The choice between two different functions depends on whether you know

the major numbers for your device. Using register\_chrdev\_region if you know the device major number and alloc\_chrdev\_region if you would like to allocate a dynamically-allocated major number.

Second, we should initialize the data structure struct cdev for our char

device and associate it with the device numbers. To initialize the struct cdev, we can achieve by the similar sequence of the following codes.

1 struct cdev \*my\_dev = cdev\_alloc(); 2 my\_cdev->ops = &my\_fops;

However, the common usage pattern will embed the struct cdev within a

device-specific structure of your own. In this case, we’ll need cdev\_init for the initialization.

1 void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops);

Once we finish the initialization, we can add the char device to the system

by using the cdev\_add.

1 int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count);

To find an example using the interface, you can see ioctl.c described in

section [9.](#Set_the_value_of_myvariable_and)

6.4 Unregistering A Device

We can not allow the kernel module to be rmmod’ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we are lucky, no other code was loaded there, and we’ll get an ugly error message. If we are unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can not be very positive.

Normally, when you do not want to allow something, you return an error

code (a negative number) from the function which is supposed to do it. With cleanup\_module that’s impossible because it is a void function. However, there is a counter which keeps track of how many processes are using your module. You can see what its value is by looking at the 3rd field with the command cat /proc/modules or lsmod. If this number isn’t zero, rmmod will fail. Note that you do not have to check the counter within cleanup\_module because the check will be performed for you by the system call sys\_delete\_module,

defined in [include/linux/syscalls.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/syscalls.h) You should not use this counter directly,

but there are functions defined in [include/linux/module.h](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/module.h) which let you display this counter:

• module\_refcount(THIS\_MODULE): Return the value of reference count of

current module.

Note: The use of try\_module\_get(THIS\_MODULE) and module\_put(THIS\_MODULE)

within a module’s own code is considered unsafe and should be avoided. The kernel automatically manages the reference count when file operations are in progress, so manual reference counting is unnecessary and can lead to race conditions. For a deeper understanding of when and how to properly use mod-

ule reference counting, see [https://stackoverflow.com/questions/1741415/](https://stackoverflow.com/questions/1741415/linux-kernel-modules-when-to-use-try-module-get-module-put)

[linux-kernel-modules-when-to-use-try-module-get-module-put.](https://stackoverflow.com/questions/1741415/linux-kernel-modules-when-to-use-try-module-get-module-put)

6.5 chardev.c

The next code sample creates a char driver named chardev. You can verify it has been registered by checking:

1 cat /proc/devices

This will show the device’s major number. To actually use the device, you

need to read from /dev/chardev (or open the file with a program) and the driver will put the number of times the device file has been read from into the file. We do not support writing to the file (like echo "hi" > /dev/chardev), but catch these attempts and tell the user that the operation is not supported. Do not worry if you do not see what we do with the data we read into the buffer;

we do not do much with it. We simply read in the data and print a message acknowledging that we received it.

In a multi-threaded environment, without any protection, concurrent access

to the same memory may lead to race conditions and will not preserve per-formance. In the kernel module, this problem may happen due to multiple instances accessing the shared resources. Therefore, a solution is to enforce exclusive access. We use atomic Compare-And-Swap (CAS) to maintain the states, CDEV\_NOT\_USED and CDEV\_EXCLUSIVE\_OPEN, to determine whether the file is currently opened by someone or not. CAS compares the contents of a memory location with the expected value and, only if they are the same, modifies the contents of that memory location to the desired value. See more

concurrency details in the [12](#84______MODULE_LICENSE__GPL) section.

1 /\*

2 \* chardev.c: Creates a read-only char device that says how many times 3 \* you have read from the dev file 4 \*/

5

6 #include

7 #include

8 #include

9 #include

10 #include

11 #include

12 #include /\* for sprintf() \*/ 13 #include

14 #include

15 #include

16 #include /\* for get\_user and put\_user \*/ 17 #include

18

19 #include

20

21 /\* Prototypes - this would normally go in a .h file \*/ 22 static int device\_open(struct inode \*, struct file \*); 23 static int device\_release(struct inode \*, struct file \*); 24 static ssize\_t device\_read(struct file \*, char \_\_user \*, size\_t, loff\_t \*); 25 static ssize\_t device\_write(struct file \*, const char \_\_user \*, size\_t, 26 loff\_t \*); 27

28 #define DEVICE\_NAME "chardev" /\* Dev name as it appears in /proc/devices \*/ 29 #define BUF\_LEN 80 /\* Max length of the message from the device \*/ 30

31 /\* Global variables are declared as static, so are global within the file. \*/

32

33 static int major; /\* major number assigned to our device driver \*/ 34

35 enum {

36 CDEV\_NOT\_USED,

37 CDEV\_EXCLUSIVE\_OPEN,

38 };

39

40 /\* Is device open? Used to prevent multiple access to device \*/ 41 static atomic\_t already\_open = ATOMIC\_INIT(CDEV\_NOT\_USED);

42

43 static char msg[BUF\_LEN + 1]; /\* The msg the device will give when asked \*/ 44

45 static struct class \*cls;

46

47 static struct file\_operations chardev\_fops = { 48 .read = device\_read,

49 .write = device\_write,

50 .open = device\_open,

51 .release = device\_release,

52 };

53

54 static int \_\_init chardev\_init(void) 55 {

56 major = register\_chrdev(0, DEVICE\_NAME, &chardev\_fops); 57

58 if (major < 0) {

59 pr\_alert("Registering char device failed with %d\n", major); 60 return major;

61 }

62

63 pr\_info("I was assigned major number %d.\n", major); 64

65 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 66 cls = class\_create(DEVICE\_NAME); 67 #else

68 cls = class\_create(THIS\_MODULE, DEVICE\_NAME); 69 #endif

70 device\_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE\_NAME); 71

72 pr\_info("Device created on /dev/%s\n", DEVICE\_NAME); 73

74 return 0;

75 }

76

77 static void \_\_exit chardev\_exit(void) 78 {

79 device\_destroy(cls, MKDEV(major, 0)); 80 class\_destroy(cls);

81

82 /\* Unregister the device \*/

83 unregister\_chrdev(major, DEVICE\_NAME); 84 }

85

86 /\* Methods \*/

87

88 /\* Called when a process tries to open the device file, like 89 \* "sudo cat /dev/chardev"

90 \*/

91 static int device\_open(struct inode \*inode, struct file \*file) 92 {

93 static int counter = 0;

94

95 if (atomic\_cmpxchg(&already\_open, CDEV\_NOT\_USED, CDEV\_EXCLUSIVE\_OPEN)) 96 return-EBUSY;

97

98 sprintf(msg, "I already told you %d times Hello world!\n", counter++); 155 }

99

100 return 0;

101 }

102

103 /\* Called when a process closes the device file. \*/ 104 static int device\_release(struct inode \*inode, struct file \*file) 105 {

106 /\* We're now ready for our next caller \*/ 107 atomic\_set(&already\_open, CDEV\_NOT\_USED); 108

109 return 0;

110 }

111

112 /\* Called when a process, which already opened the dev file, attempts to 113 \* read from it.

114 \*/

115 static ssize\_t device\_read(struct file \*filp, /\* see include/linux/fs.h \*/ 116 char \_\_user \*buffer, /\* buffer to fill with data \*/ 117 size\_t length, /\* length of the buffer \*/ 118 loff\_t \*offset) 119 {

120 /\* Number of bytes actually written to the buffer \*/ 121 int bytes\_read = 0;

122 const char \*msg\_ptr = msg;

123

124 if (!\*(msg\_ptr + \*offset)) { /\* we are at the end of message \*/ 125 \*offset = 0; /\* reset the offset \*/ 126 return 0; /\* signify end of file \*/ 127 }

128

129 msg\_ptr += \*offset;

130

131 /\* Actually put the data into the buffer \*/ 132 while (length && \*msg\_ptr) {

133 /\* The buffer is in the user data segment, not the kernel 134 \* segment so "\*" assignment won't work. We have to use 135 \* put\_user which copies data from the kernel data segment to 136 \* the user data segment. 137 \*/

138 put\_user(\*(msg\_ptr++), buffer++); 139 length--;

140 bytes\_read++;

141 }

142

143 \*offset += bytes\_read;

144

145 /\* Most read functions return the number of bytes put into the buffer. \*/

146 return bytes\_read;

147 }

148

149 /\* Called when a process writes to dev file: echo "hi" | sudo tee /dev/chardev

, → \*/

150 static ssize\_t device\_write(struct file \*filp, const char \_\_user \*buff, 151 size\_t len, loff\_t \*off) 152 {

153 pr\_alert("Sorry, this operation is not supported.\n"); 154 return-EINVAL;

156

157 module\_init(chardev\_init);

158 module\_exit(chardev\_exit);

159

160 MODULE\_LICENSE("GPL");

6.6 Writing Modules for Multiple Kernel Versions

The system calls, which are the major interface the kernel shows to the processes, generally stay the same across versions. A new system call may be added, but usually the old ones will behave exactly like they used to. This is necessary for backward compatibility – a new kernel version is not supposed to break regular processes. In most cases, the device files will also remain the same. On the other hand, the internal interfaces within the kernel can and do change between versions.

There are differences between different kernel versions, and if you want to

support multiple kernel versions, you will find yourself having to code condi-tional compilation directives. The way to do this is to compare the macro LINUX\_VERSION\_CODE to the macro KERNEL\_VERSION. In version a.b.c of the kernel, the value of this macro would be 16 8 2 a + 2b + c.

7 The /proc Filesystem

In Linux, there is an additional mechanism for the kernel and kernel modules to send information to processes — the /proc filesystem. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as /proc/modules which provides the list of modules and /proc/meminfo which gathers memory usage statistics.

The method to use the proc filesystem is very similar to the one used with

device drivers — a structure is created with all the information needed for the /proc file, including pointers to any handler functions (in our case there is only one, the one called when somebody attempts to read from the /proc file). Then, init\_module registers the structure with the kernel and cleanup\_module unregisters it.

Normal filesystems are located on a disk, rather than just in memory (which

is where /proc is), and in that case the index-node (inode for short) number is a pointer to a disk location where the file’s inode is located. The inode contains information about the file, for example the file’s permissions, together with a pointer to the disk location or locations where the file’s data can be found.

Because we do not get called when the file is opened or closed, there is

nowhere for us to put try\_module\_get and module\_put in this module, and if the file is opened and then the module is removed, there is no way to avoid the consequences. The kernel’s automatic reference counting for file operations

helps prevent module removal while files are in use, but /proc files require careful handling due to their different lifecycle.

Here is a simple example showing how to use a /proc file. This is the

HelloWorld for the /proc filesystem. There are three parts: create the file /proc/helloworld in the function init\_module, return a value (and a buffer) when the file /proc/helloworld is read in the callback function procfile\_read, and delete the file /proc/helloworld in the function cleanup\_module.

The /proc/helloworld is created when the module is loaded with the func-

tion proc\_create. The return value is a pointer to struct proc\_dir\_entry, and it will be used to configure the file /proc/helloworld (for example, the owner of this file). A null return value means that the creation has failed.

Every time the file /proc/helloworld is read, the function procfile\_read

is called. Two parameters of this function are very important: the buffer (the second parameter) and the offset (the fourth one). The content of the buffer will be returned to the application which read it (for example the cat command). The offset is the current position in the file. If the return value of the function is not null, then this function is called again. So be careful with this function, if it never returns zero, the read function is called endlessly.

$ cat /proc/helloworld

HelloWorld!

1 /\*

2 \* procfs1.c

3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10

11 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 6, 0) 12 #define HAVE\_PROC\_OPS

13 #endif

14

15 #define procfs\_name "helloworld"

16

17 static struct proc\_dir\_entry \*our\_proc\_file; 18

19 static ssize\_t procfile\_read(struct file \*file\_pointer, char \_\_user \*buffer, 20 size\_t buffer\_length, loff\_t \*offset) 21 {

22 char s[13] = "HelloWorld!\n";

23 int len = sizeof(s);

24 ssize\_t ret = len;

25

26 if (\*offset >= len || copy\_to\_user(buffer, s, len)) { 27 pr\_info("copy\_to\_user failed\n"); 28 ret = 0;

29 } else {

30 pr\_info("procfile read %s\n",

, → file\_pointer->f\_path.dentry->d\_name.name);

31 \*offset += len;

32 }

33

34 return ret;

35 }

36

37 #ifdef HAVE\_PROC\_OPS

38 static const struct proc\_ops proc\_file\_fops = { 39 .proc\_read = procfile\_read,

40 };

41 #else

42 static const struct file\_operations proc\_file\_fops = { 43 .read = procfile\_read,

44 };

45 #endif

46

47 static int \_\_init procfs1\_init(void) 48 {

49 our\_proc\_file = proc\_create(procfs\_name, 0644, NULL, &proc\_file\_fops); 50 if (NULL == our\_proc\_file) {

51 pr\_alert("Error:Could not initialize /proc/%s\n", procfs\_name); 52 return-ENOMEM;

53 }

54

55 pr\_info("/proc/%s created\n", procfs\_name); 56 return 0;

57 }

58

59 static void \_\_exit procfs1\_exit(void) 60 {

61 proc\_remove(our\_proc\_file);

62 pr\_info("/proc/%s removed\n", procfs\_name); 63 }

64

65 module\_init(procfs1\_init);

66 module\_exit(procfs1\_exit);

67

68 MODULE\_LICENSE("GPL");

7.1 The proc\_ops Structure

The proc\_ops structure is defined in [include/linux/proc\_fs.h](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/proc_fs.h) in Linux v5.6+. In older kernels, it used file\_operations for custom hooks in /proc filesystem, but it contains some members that are unnecessary in VFS, and every time VFS expands file\_operations set, /proc code comes bloated. On the other hand, not only the space, but also some operations were saved by this structure to improve its performance. For example, the file which never disappears in /proc can set the proc\_flag as PROC\_ENTRY\_PERMANENT to save 2 atomic ops, 1 allocation, 1 free in per open/read/close sequence.

7.2 Read and Write a /proc File

We have seen a very simple example for a /proc file where we only read the file /proc/helloworld. It is also possible to write in a /proc file. It works the same way as read, a function is called when the /proc file is written. But there is a little difference with read, data comes from user, so you have to import data from user space to kernel space (with copy\_from\_user or get\_user)

The reason for copy\_from\_user or get\_user is that Linux memory (on Intel

architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes.

The only memory segment accessible to a process is its own, so when writing

regular programs to run as processes, there is no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The put\_user and get\_user macros allow you to access that memory. These functions handle only one character, you can handle several characters with copy\_to\_user and copy\_from\_user. As the buffer (in read or write function) is in kernel space, for write function you need to import data because it comes from user space, but not for the read function because data is already in kernel space.

1 /\*

2 \* procfs2.c - create a "file" in /proc 3 \*/

4

5 #include /\* We're doing kernel work \*/ 6 #include /\* Specifically, a module \*/ 7 #include /\* Necessary because we use the proc fs \*/ 8 #include /\* for copy\_from\_user \*/ 9 #include

10

11 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 6, 0) 12 #define HAVE\_PROC\_OPS

13 #endif

14

15 #define PROCFS\_MAX\_SIZE 1024

16 #define PROCFS\_NAME "buffer1k"

17

18 /\* This structure hold information about the /proc file \*/ 19 static struct proc\_dir\_entry \*our\_proc\_file; 20

21 /\* The buffer used to store character for this module \*/ 22 static char procfs\_buffer[PROCFS\_MAX\_SIZE]; 23

24 /\* The size of the buffer \*/

25 static unsigned long procfs\_buffer\_size = 0;

26

27 /\* This function is called then the /proc file is read \*/ 28 static ssize\_t procfile\_read(struct file \*file\_pointer, char \_\_user \*buffer, 29 size\_t buffer\_length, loff\_t \*offset) 30 {

31 char s[13] = "HelloWorld!\n";

32 int len = sizeof(s);

33 ssize\_t ret = len;

34

35 if (\*offset >= len || copy\_to\_user(buffer, s, len)) { 36 pr\_info("copy\_to\_user failed\n"); 37 ret = 0;

38 } else {

39 pr\_info("procfile read %s\n",

,→ file\_pointer->f\_path.dentry->d\_name.name);

40 \*offset += len;

41 }

42

43 return ret;

44 }

45

46 /\* This function is called with the /proc file is written. \*/ 47 static ssize\_t procfile\_write(struct file \*file, const char \_\_user \*buff, 48 size\_t len, loff\_t \*off) 49 {

50 procfs\_buffer\_size = len;

51 if (procfs\_buffer\_size >= PROCFS\_MAX\_SIZE) 52 procfs\_buffer\_size = PROCFS\_MAX\_SIZE - 1; 53

54 if (copy\_from\_user(procfs\_buffer, buff, procfs\_buffer\_size)) 55 return-EFAULT;

56

57 procfs\_buffer[procfs\_buffer\_size] = '\0';

58 \*off += procfs\_buffer\_size;

59 pr\_info("procfile write %s\n", procfs\_buffer); 60

61 return procfs\_buffer\_size;

62 }

63

64 #ifdef HAVE\_PROC\_OPS

65 static const struct proc\_ops proc\_file\_fops = { 66 .proc\_read = procfile\_read,

67 .proc\_write = procfile\_write,

68 };

69 #else

70 static const struct file\_operations proc\_file\_fops = { 71 .read = procfile\_read,

72 .write = procfile\_write,

73 };

74 #endif

75

76 static int \_\_init procfs2\_init(void) 77 {

78 our\_proc\_file = proc\_create(PROCFS\_NAME, 0644, NULL, &proc\_file\_fops);

79 if (NULL == our\_proc\_file) {

80 pr\_alert("Error:Could not initialize /proc/%s\n", PROCFS\_NAME); 81 return-ENOMEM;

82 }

83

84 pr\_info("/proc/%s created\n", PROCFS\_NAME); 85 return 0;

86 }

87

88 static void \_\_exit procfs2\_exit(void) 89 {

90 proc\_remove(our\_proc\_file);

91 pr\_info("/proc/%s removed\n", PROCFS\_NAME); 92 }

93

94 module\_init(procfs2\_init);

95 module\_exit(procfs2\_exit);

96

97 MODULE\_LICENSE("GPL");

7.3 Manage /proc file with standard filesystem

We have seen how to read and write a /proc file with the /proc interface. But it is also possible to manage /proc file with inodes. The main concern is to use advanced functions, like permissions.

In Linux, there is a standard mechanism for filesystem registration. Since

every filesystem has to have its own functions to handle inode and file op-erations, there is a special structure to hold pointers to all those functions, struct inode\_operations, which includes a pointer to struct proc\_ops.

The difference between file and inode operations is that file operations deal

with the file itself whereas inode operations deal with ways of referencing the file, such as creating links to it.

In /proc, whenever we register a new file, we’re allowed to specify which

struct inode\_operations will be used to access to it. This is the mechanism we use, a struct inode\_operations which includes a pointer to a struct proc\_ops which includes pointers to our procfs\_read and procfs\_write functions.

Another interesting point here is the module\_permission function. This

function is called whenever a process tries to do something with the /proc file, and it can decide whether to allow access or not. Right now it is only based on the operation and the uid of the current user (as available in current, a pointer to a structure which includes information on the currently running process), but it could be based on anything we like, such as what other processes are doing with the same file, the time of day, or the last input we received.

It is important to note that the standard roles of read and write are reversed

in the kernel. Read functions are used for output, whereas write functions are used for input. The reason for that is that read and write refer to the user’s point of view — if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

1 /\*

2 \* procfs3.c

3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 10, 0) 12 #include

13 #endif

14

15 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 6, 0) 16 #define HAVE\_PROC\_OPS

17 #endif

18

19 #define PROCFS\_MAX\_SIZE 2048UL

20 #define PROCFS\_ENTRY\_FILENAME "buffer2k" 21

22 static struct proc\_dir\_entry \*our\_proc\_file; 23 static char procfs\_buffer[PROCFS\_MAX\_SIZE]; 24 static unsigned long procfs\_buffer\_size = 0; 25

26 static ssize\_t procfs\_read(struct file \*filp, char \_\_user \*buffer, 27 size\_t length, loff\_t \*offset) 28 {

29 if (\*offset || procfs\_buffer\_size == 0) { 30 pr\_debug("procfs\_read: END\n"); 31 \*offset = 0;

32 return 0;

33 }

34 procfs\_buffer\_size = min(procfs\_buffer\_size, length); 35 if (copy\_to\_user(buffer, procfs\_buffer, procfs\_buffer\_size)) 36 return-EFAULT;

37 \*offset += procfs\_buffer\_size;

38

39 pr\_debug("procfs\_read: read %lu bytes\n", procfs\_buffer\_size); 40 return procfs\_buffer\_size;

41 }

42 static ssize\_t procfs\_write(struct file \*file, const char \_\_user \*buffer, 43 size\_t len, loff\_t \*off) 44 {

45 procfs\_buffer\_size = min(PROCFS\_MAX\_SIZE, len); 46 if (copy\_from\_user(procfs\_buffer, buffer, procfs\_buffer\_size)) 47 return-EFAULT;

48 \*off += procfs\_buffer\_size;

49

50 pr\_debug("procfs\_write: write %lu bytes\n", procfs\_buffer\_size); 51 return procfs\_buffer\_size;

52 }

53 static int procfs\_open(struct inode \*inode, struct file \*file) 54 {

55 return 0;

56 }

57 static int procfs\_close(struct inode \*inode, struct file \*file) 58 {

59 return 0;

60 }

61

62 #ifdef HAVE\_PROC\_OPS

63 static struct proc\_ops file\_ops\_4\_our\_proc\_file = { 64 .proc\_read = procfs\_read,

65 .proc\_write = procfs\_write,

66 .proc\_open = procfs\_open,

67 .proc\_release = procfs\_close,

68 };

69 #else

70 static const struct file\_operations file\_ops\_4\_our\_proc\_file = { 71 .read = procfs\_read,

72 .write = procfs\_write,

73 .open = procfs\_open,

74 .release = procfs\_close,

75 };

76 #endif

77

78 static int \_\_init procfs3\_init(void) 79 {

80 our\_proc\_file = proc\_create(PROCFS\_ENTRY\_FILENAME, 0644, NULL, 81 &file\_ops\_4\_our\_proc\_file); 82 if (our\_proc\_file == NULL) {

83 pr\_debug("Error: Could not initialize /proc/%s\n",

84 PROCFS\_ENTRY\_FILENAME);

85 return-ENOMEM;

86 }

87 proc\_set\_size(our\_proc\_file, 80); 88 proc\_set\_user(our\_proc\_file, GLOBAL\_ROOT\_UID, GLOBAL\_ROOT\_GID);

89

90 pr\_debug("/proc/%s created\n", PROCFS\_ENTRY\_FILENAME); 91 return 0;

92 }

93

94 static void \_\_exit procfs3\_exit(void) 95 {

96 remove\_proc\_entry(PROCFS\_ENTRY\_FILENAME, NULL); 97 pr\_debug("/proc/%s removed\n", PROCFS\_ENTRY\_FILENAME); 98 }

99

100 module\_init(procfs3\_init);

101 module\_exit(procfs3\_exit);

102

103 MODULE\_LICENSE("GPL");

Still hungry for procfs examples? Well, first of all keep in mind, there are

rumors around, claiming that procfs is on its way out, consider using sysfs in-stead. Consider using this mechanism, in case you want to document something kernel related yourself.

7.4 Manage /proc file with seq\_file

As we have seen, writing a /proc file may be quite “complex”. So to help people writing /proc file, there is an API named seq\_file that helps formatting a

/proc file for output. It is based on sequence, which is composed of 3 functions: start(), next(), and stop(). The seq\_file API starts a sequence when a user reads the /proc file.

A sequence begins with the call of the function start(). If the return is a

non NULL value, the function next() is called; otherwise, the stop() function is called directly. This function is an iterator, the goal is to go through all the data. Each time next() is called, the function show() is also called. It writes data values in the buffer read by the user. The function next() is called until it returns NULL. The sequence ends when next() returns NULL, then the function stop() is called.

BE CAREFUL: when a sequence is finished, another one starts. That means

that at the end of function stop(), the function start() is called again. This loop finishes when the function start() returns NULL. You can see a scheme of

this in the Figure [1.](#start___treatment)

The seq\_file provides basic functions for proc\_ops, such as seq\_read,

seq\_lseek, and some others. But nothing to write in the /proc file. Of course, you can still use the same way as in the previous example.

1 /\*

2 \* procfs4.c - create a "file" in /proc 3 \* This program uses the seq\_file library to manage the /proc file. 4 \*/

5

6 #include /\* We are doing kernel work \*/ 7 #include /\* Specifically, a module \*/ 8 #include /\* Necessary because we use proc fs \*/ 9 #include /\* for seq\_file \*/

10 #include

11

12 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 6, 0) 13 #define HAVE\_PROC\_OPS

14 #endif

15

16 #define PROC\_NAME "iter"

17

18 /\* This function is called at the beginning of a sequence. 19 \* ie, when:

20 \* - the /proc file is read (first time) 21 \* - after the function stop (end of sequence) 22 \*/

23 static void \*my\_seq\_start(struct seq\_file \*s, loff\_t \*pos) 24 {

25 static unsigned long counter = 0;

26

27 /\* beginning a new sequence? \*/

28 if (\*pos == 0) {

29 /\* yes => return a non null value to begin the sequence \*/ 30 return &counter;

31 }

32

33 /\* no => it is the end of the sequence, return end to stop reading \*/ 34 \*pos = 0;

35 return NULL;

36 }

37

38 /\* This function is called after the beginning of a sequence. 39 \* It is called until the return is NULL (this ends the sequence).

40 \*/

41 static void \*my\_seq\_next(struct seq\_file \*s, void \*v, loff\_t \*pos) 42 {

43 unsigned long \*tmp\_v = (unsigned long \*)v; 44 (\*tmp\_v)++;

45 (\*pos)++;

46 return NULL;

47 }

48

49 /\* This function is called at the end of a sequence. \*/ 50 static void my\_seq\_stop(struct seq\_file \*s, void \*v) 51 {

52 /\* nothing to do, we use a static value in start() \*/ 53 }

54

55 /\* This function is called for each "step" of a sequence. \*/ 56 static int my\_seq\_show(struct seq\_file \*s, void \*v) 57 {

58 loff\_t \*spos = (loff\_t \*)v;

59

60 seq\_printf(s, "%Ld\n", \*spos);

61 return 0;

62 }

63

64 /\* This structure gather "function" to manage the sequence \*/ 65 static struct seq\_operations my\_seq\_ops = { 66 .start = my\_seq\_start,

67 .next = my\_seq\_next,

68 .stop = my\_seq\_stop,

69 .show = my\_seq\_show,

70 };

71

72 /\* This function is called when the /proc file is open. \*/ 73 static int my\_open(struct inode \*inode, struct file \*file) 74 {

75 return seq\_open(file, &my\_seq\_ops);

76 };

77

78 /\* This structure gather "function" that manage the /proc file \*/

79 #ifdef HAVE\_PROC\_OPS

80 static const struct proc\_ops my\_file\_ops = { 81 .proc\_open = my\_open,

82 .proc\_read = seq\_read,

83 .proc\_lseek = seq\_lseek,

84 .proc\_release = seq\_release,

85 };

86 #else

87 static const struct file\_operations my\_file\_ops = { 88 .open = my\_open,

89 .read = seq\_read,

90 .llseek = seq\_lseek,

91 .release = seq\_release,

92 };

93 #endif

94

95 static int \_\_init procfs4\_init(void) 96 {

97 struct proc\_dir\_entry \*entry;

98

99 entry = proc\_create(PROC\_NAME, 0, NULL, &my\_file\_ops);

100 if (entry == NULL) {

101 pr\_debug("Error: Could not initialize /proc/%s\n", PROC\_NAME); 102 return-ENOMEM;

103 }

104

105 return 0;

106 }

107

108 static void \_\_exit procfs4\_exit(void) 109 {

110 remove\_proc\_entry(PROC\_NAME, NULL); 111 pr\_debug("/proc/%s removed\n", PROC\_NAME); 112 }

113

114 module\_init(procfs4\_init);

115 module\_exit(procfs4\_exit);

116

117 MODULE\_LICENSE("GPL");

If you want more information, you can read this web page:

• <https://lwn.net/Articles/22355/>

• <https://kernelnewbies.org/Documents/SeqFileHowTo>

You can also read the code of [fs/seq\_file.c](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/seq_file.c) in the Linux kernel.

8 sysfs: Interacting with your module

sysfs allows you to interact with the running kernel from userspace by reading or setting variables inside of modules. This can be useful for debugging purposes, or just as an interface for applications or scripts. You can find sysfs directories and files under the /sys directory on your system.

1 ls -l /sys

Attributes can be exported for kobjects in the form of regular files in the

filesystem. Sysfs forwards file I/O operations to methods defined for the at-tributes, providing a means to read and write kernel attributes.

A simple attribute definition:

1 struct attribute {

2 char \*name;

3 struct module \*owner;

4 umode\_t mode;

5 };

6

7 int sysfs\_create\_file(struct kobject \* kobj, const struct attribute \* attr); 8 void sysfs\_remove\_file(struct kobject \* kobj, const struct attribute \* attr);

For example, the driver model defines struct device\_attribute like:

1 struct device\_attribute {

2 struct attribute attr;

3 ssize\_t (\*show)(struct device \*dev, struct device\_attribute \*attr, 4 char \*buf);

5 ssize\_t (\*store)(struct device \*dev, struct device\_attribute \*attr, 6 const char \*buf, size\_t count); 7 };

8

9 int device\_create\_file(struct device \*, const struct device\_attribute \*);

10 void device\_remove\_file(struct device \*, const struct device\_attribute \*);

To read or write attributes, the show() or store() method must be speci-

fied when declaring the attribute. For the common cases [include/linux/sysfs.h](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/sysfs.h) provides convenience macros (\_\_ATTR, \_\_ATTR\_RO, \_\_ATTR\_WO, etc.) to make defining attributes easier as well as making code more concise and readable.

An example of a hello world module which includes the creation of a variable

accessible via sysfs is given below.

1 /\*

2 \* hello-sysfs.c sysfs example

3 \*/

4 #include

5 #include

6 #include

7 #include

8 #include

9 #include

10

11 static struct kobject \*mymodule;

12

13 /\* the variable you want to be able to change \*/ 14 static int myvariable = 0;

15

16 static ssize\_t myvariable\_show(struct kobject \*kobj, 17 struct kobj\_attribute \*attr, char \*buf) 18 {

19 return sprintf(buf, "%d\n", myvariable);

20 }

21

22 static ssize\_t myvariable\_store(struct kobject \*kobj, 23 struct kobj\_attribute \*attr, const char \*buf, 24 size\_t count) 25 {

26 sscanf(buf, "%d", &myvariable);

27 return count;

28 }

29

30 static struct kobj\_attribute myvariable\_attribute = 31 \_\_ATTR(myvariable, 0660, myvariable\_show, myvariable\_store);

32

33 static int \_\_init mymodule\_init(void) 34 {

35 int error = 0;

36

37 pr\_info("mymodule: initialized\n"); 38

39 mymodule = kobject\_create\_and\_add("mymodule", kernel\_kobj); 40 if (!mymodule)

41 return-ENOMEM;

42

43 error = sysfs\_create\_file(mymodule, &myvariable\_attribute.attr);

44 if (error) {

45 kobject\_put(mymodule);

46 pr\_info("failed to create the myvariable file "

47 "in /sys/kernel/mymodule\n");

48 }

49

50 return error;

51 }

52

53 static void \_\_exit mymodule\_exit(void) 54 {

55 pr\_info("mymodule: Exit success\n"); 56 kobject\_put(mymodule);

57 }

58

59 module\_init(mymodule\_init);

60 module\_exit(mymodule\_exit);

61

62 MODULE\_LICENSE("GPL");

Make and install the module:

1 make

2 sudo insmod hello-sysfs.ko

Check that it exists:

1 lsmod | grep hello\_sysfs

What is the current value of myvariable ?

1 cat /sys/kernel/mymodule/myvariable

Set the value of myvariable and check that it changed.

1 echo "32" | sudo tee /sys/kernel/mymodule/myvariable 2 cat /sys/kernel/mymodule/myvariable

Finally, remove the test module:

1 sudo rmmod hello\_sysfs

In the above case, we use a simple kobject to create a directory under

sysfs, and communicate with its attributes. Since Linux v2.6.0, the kobject structure made its appearance. It was initially meant as a simple way of uni-fying kernel code which manages reference counted objects. After a bit of mission creep, it is now the glue that holds much of the device model and its sysfs interface together. For more information about kobject and sysfs,

see [Documentation/driver-api/driver-model/driver.rst](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/driver-api/driver-model/driver.rst) and [https://lwn.net/](https://lwn.net/Articles/51437/)

[Articles/51437/.](https://lwn.net/Articles/51437/)

9 Talking To Device Files

Device files are supposed to represent physical devices. Most physical devices are used for output as well as input, so there has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. In the following example, this is implemented by device\_write.

This is not always enough. Imagine you had a serial port connected to a

modem (even if you have an internal modem, it is still implemented from the CPU’s perspective as a serial port connected to a modem, so you don’t have to tax your imagination too hard). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to configure the rate at which data is sent and received.

The answer in Unix is to use a special function called ioctl (short for Input

Output ConTroL). Every device can have its own ioctl commands, which can be read ioctl’s (to send information from a process to the kernel), write ioctl’s (to return information to a process), both or neither. Notice here the roles of read and write are reversed again, so in ioctl’s read is to send information to the kernel and write is to receive information from the kernel.

The ioctl function is called with three parameters: the file descriptor of the

appropriate device file, the ioctl number, and a parameter, which is of type long so you can use a cast to use it to pass anything. You will not be able to pass a

structure this way, but you will be able to pass a pointer to the structure. Here is an example:

1 /\*

2 \* ioctl.c

3 \*/

4 #include

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11 #include

12

13 struct ioctl\_arg {

14 unsigned int val;

15 };

16

17 /\* Documentation/userspace-api/ioctl/ioctl-number.rst \*/ 18 #define IOC\_MAGIC '\x66'

19

20 #define IOCTL\_VALSET \_IOW(IOC\_MAGIC, 0, struct ioctl\_arg) 21 #define IOCTL\_VALGET \_IOR(IOC\_MAGIC, 1, struct ioctl\_arg) 22 #define IOCTL\_VALGET\_NUM \_IOR(IOC\_MAGIC, 2, int) 23 #define IOCTL\_VALSET\_NUM \_IOW(IOC\_MAGIC, 3, int) 24

25 #define IOCTL\_VAL\_MAXNR 3

26 #define DRIVER\_NAME "ioctltest"

27

28 static unsigned int test\_ioctl\_major = 0; 29 static unsigned int num\_of\_dev = 1; 30 static struct cdev test\_ioctl\_cdev; 31 static int ioctl\_num = 0;

32

33 struct test\_ioctl\_data {

34 unsigned char val;

35 rwlock\_t lock;

36 };

37

38 static long test\_ioctl\_ioctl(struct file \*filp, unsigned int cmd, 39 unsigned long arg) 40 {

41 struct test\_ioctl\_data \*ioctl\_data = filp->private\_data; 42 int retval = 0;

43 unsigned char val;

44 struct ioctl\_arg data;

45 memset(&data, 0, sizeof(data));

46

47 switch (cmd) {

48 case IOCTL\_VALSET:

49 if (copy\_from\_user(&data, (int \_\_user \*)arg, sizeof(data))) { 50 retval = -EFAULT;

51 goto done;

52 }

53

54 pr\_alert("IOCTL set val:%x .\n", data.val);

55 write\_lock(&ioctl\_data->lock);

56 ioctl\_data->val = data.val;

57 write\_unlock(&ioctl\_data->lock);

58 break;

59

60 case IOCTL\_VALGET:

61 read\_lock(&ioctl\_data->lock);

62 val = ioctl\_data->val;

63 read\_unlock(&ioctl\_data->lock);

64 data.val = val;

65

66 if (copy\_to\_user((int \_\_user \*)arg, &data, sizeof(data))) { 67 retval = -EFAULT;

68 goto done;

69 }

70

71 break;

72

73 case IOCTL\_VALGET\_NUM:

74 retval = \_\_put\_user(ioctl\_num, (int \_\_user \*)arg);

75 break;

76

77 case IOCTL\_VALSET\_NUM:

78 ioctl\_num = arg;

79 break;

80

81 default:

82 retval = -ENOTTY;

83 }

84

85 done:

86 return retval;

87 }

88

89 static ssize\_t test\_ioctl\_read(struct file \*filp, char \_\_user \*buf, 90 size\_t count, loff\_t \*f\_pos) 91 {

92 struct test\_ioctl\_data \*ioctl\_data = filp->private\_data; 93 unsigned char val;

94 int retval;

95 int i = 0;

96

97 read\_lock(&ioctl\_data->lock);

98 val = ioctl\_data->val;

99 read\_unlock(&ioctl\_data->lock); 100

101 for (; i < count; i++) {

102 if (copy\_to\_user(&buf[i], &val, 1)) { 103 retval = -EFAULT;

104 goto out;

105 }

106 }

107

108 retval = count;

109 out:

110 return retval;

111 }

112

113 static int test\_ioctl\_close(struct inode \*inode, struct file \*filp) 114 {

115 pr\_alert("%s call.\n", \_\_func\_\_); 116

117 if (filp->private\_data) {

118 kfree(filp->private\_data); 119 filp->private\_data = NULL; 120 }

121

122 return 0;

123 }

124

125 static int test\_ioctl\_open(struct inode \*inode, struct file \*filp) 126 {

127 struct test\_ioctl\_data \*ioctl\_data; 128

129 pr\_alert("%s call.\n", \_\_func\_\_); 130 ioctl\_data = kmalloc(sizeof(struct test\_ioctl\_data), GFP\_KERNEL); 131

132 if (ioctl\_data == NULL)

133 return-ENOMEM;

134

135 rwlock\_init(&ioctl\_data->lock); 136 ioctl\_data->val = 0xFF;

137 filp->private\_data = ioctl\_data; 138

139 return 0;

140 }

141

142 static struct file\_operations fops = { 143 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 144 .owner = THIS\_MODULE,

145 #endif

146 .open = test\_ioctl\_open,

147 .release = test\_ioctl\_close,

148 .read = test\_ioctl\_read,

149 .unlocked\_ioctl = test\_ioctl\_ioctl, 150 };

151

152 static int \_\_init ioctl\_init(void) 153 {

154 dev\_t dev;

155 int ret;

156

157 ret = alloc\_chrdev\_region(&dev, 0, num\_of\_dev, DRIVER\_NAME);

158

159 if (ret)

160 return ret;

161

162 test\_ioctl\_major = MAJOR(dev); 163 cdev\_init(&test\_ioctl\_cdev, &fops); 164 ret = cdev\_add(&test\_ioctl\_cdev, dev, num\_of\_dev); 165

166 if (ret) {

167 unregister\_chrdev\_region(dev, num\_of\_dev); 168 return ret;

169 }

170

171 pr\_alert("%s driver(major: %d) installed.\n", DRIVER\_NAME, 172 test\_ioctl\_major);

173 return 0;

174 }

175

176 static void \_\_exit ioctl\_exit(void) 177 {

178 dev\_t dev = MKDEV(test\_ioctl\_major, 0); 179

180 cdev\_del(&test\_ioctl\_cdev);

181 unregister\_chrdev\_region(dev, num\_of\_dev); 182 pr\_alert("%s driver removed.\n", DRIVER\_NAME); 183 }

184

185 module\_init(ioctl\_init);

186 module\_exit(ioctl\_exit);

187

188 MODULE\_LICENSE("GPL");

189 MODULE\_DESCRIPTION("This is test\_ioctl module");

You can see there is an argument called cmd in test\_ioctl\_ioctl() func-

tion. It is the ioctl number. The ioctl number encodes the major device number, the type of the ioctl, the command, and the type of the parameter. This ioctl number is usually created by a macro call (\_IO, \_IOR, \_IOW or \_IOWR — depend-ing on the type) in a header file. This header file should then be included both by the programs which will use ioctl (so they can generate the appropriate ioctl’s) and by the kernel module (so it can understand it). In the example below, the header file is chardev.h and the program which uses it is userspace\_ioctl.c.

If you want to use ioctls in your own kernel modules, it is best to receive an

official ioctl assignment, so if you accidentally get somebody else’s ioctls, or if they get yours, you’ll know something is wrong. For more information, consult

the kernel source tree at [Documentation/userspace-api/ioctl/ioctl-number.rst.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/userspace-api/ioctl/ioctl-number.rst)

Also, we need to be careful that concurrent access to the shared resources

will lead to the race condition. The solution is using atomic Compare-And-Swap

(CAS), which we mentioned at [6.5](#6_4___Unregistering_A_Device) section, to enforce the exclusive access.

1 /\*

2 \* chardev2.c - Create an input/output character device 3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11 #include /\* Specifically, a module \*/ 12 #include

13 #include

14 #include /\* for get\_user and put\_user \*/ 15 #include

16

17 #include

18

19 #include "chardev.h"

20 #define DEVICE\_NAME "char\_dev"

21 #define BUF\_LEN 80

22

23 enum {

24 CDEV\_NOT\_USED,

25 CDEV\_EXCLUSIVE\_OPEN,

26 };

27

28 /\* Is the device open right now? Used to prevent concurrent access into 29 \* the same device

30 \*/

31 static atomic\_t already\_open = ATOMIC\_INIT(CDEV\_NOT\_USED); 32

33 /\* The message the device will give when asked \*/ 34 static char message[BUF\_LEN + 1]; 35

36 static struct class \*cls;

37

38 /\* This is called whenever a process attempts to open the device file \*/

39 static int device\_open(struct inode \*inode, struct file \*file) 40 {

41 pr\_info("device\_open(%p)\n", file); 42

43 return 0;

44 }

45

46 static int device\_release(struct inode \*inode, struct file \*file) 47 {

48 pr\_info("device\_release(%p,%p)\n", inode, file); 49

50 return 0;

51 }

52

53 /\* This function is called whenever a process which has already opened the

54 \* device file attempts to read from it. 55 \*/

56 static ssize\_t device\_read(struct file \*file, /\* see include/linux/fs.h \*/ 57 char \_\_user \*buffer, /\* buffer to be filled \*/ 58 size\_t length, /\* length of the buffer \*/ 59 loff\_t \*offset) 60 {

61 /\* Number of bytes actually written to the buffer \*/ 62 int bytes\_read = 0;

63 /\* How far did the process reading the message get? Useful if the message

64 \* is larger than the size of the buffer we get to fill in device\_read.

65 \*/

66 const char \*message\_ptr = message;

67

68 if (!\*(message\_ptr + \*offset)) { /\* we are at the end of message \*/ 69 \*offset = 0; /\* reset the offset \*/ 70 return 0; /\* signify end of file \*/ 71 }

72

73 message\_ptr += \*offset;

74

75 /\* Actually put the data into the buffer \*/ 76 while (length && \*message\_ptr) { 77 /\* Because the buffer is in the user data segment, not the kernel

78 \* data segment, assignment would not work. Instead, we have to 79 \* use put\_user which copies data from the kernel data segment to

80 \* the user data segment.

81 \*/

82 put\_user(\*(message\_ptr++), buffer++);

83 length--;

84 bytes\_read++;

85 }

86

87 pr\_info("Read %d bytes, %ld left\n", bytes\_read, length); 88

89 \*offset += bytes\_read;

90

91 /\* Read functions are supposed to return the number of bytes actually

92 \* inserted into the buffer.

93 \*/

94 return bytes\_read;

95 }

96

97 /\* called when somebody tries to write into our device file. \*/ 98 static ssize\_t device\_write(struct file \*file, const char \_\_user \*buffer, 99 size\_t length, loff\_t \*offset)

100 {

101 int i;

102

103 pr\_info("device\_write(%p,%p,%ld)", file, buffer, length); 104

105 for (i = 0; i < length && i < BUF\_LEN; i++) 106 get\_user(message[i], buffer + i); 107

108 /\* Again, return the number of input characters used. \*/ 109 return i;

110 }

111

112 /\* This function is called whenever a process tries to do an ioctl on our

113 \* device file. We get two extra parameters (additional to the inode and file

114 \* structures, which all device functions get): the number of the ioctl

called

115 \* and the parameter given to the ioctl function. 116 \*

117 \* If the ioctl is write or read/write (meaning output is returned to the

118 \* calling process), the ioctl call returns the output of this function.

119 \*/

120 static long

121 device\_ioctl(struct file \*file, /\* ditto \*/ 122 unsigned int ioctl\_num, /\* number and param for ioctl \*/ 123 unsigned long ioctl\_param) 124 {

125 int i;

126 long ret = 0;

127

128 /\* We don't want to talk to two processes at the same time. \*/ 129 if (atomic\_cmpxchg(&already\_open, CDEV\_NOT\_USED, CDEV\_EXCLUSIVE\_OPEN)) 130 return-EBUSY;

131

132 /\* Switch according to the ioctl called \*/ 133 switch (ioctl\_num) {

134 case IOCTL\_SET\_MSG: {

135 /\* Receive a pointer to a message (in user space) and set that to

136 \* be the device's message. Get the parameter given to ioctl by

137 \* the process.

138 \*/

139 char \_\_user \*tmp = (char \_\_user \*)ioctl\_param; 140 char ch;

141

142 /\* Find the length of the message \*/ 143 get\_user(ch, tmp);

144 for (i = 0; ch && i < BUF\_LEN; i++, tmp++) 145 get\_user(ch, tmp);

146

147 device\_write(file, (char \_\_user \*)ioctl\_param, i, NULL); 148 break;

149 }

150 case IOCTL\_GET\_MSG: {

151 loff\_t offset = 0;

152

153 /\* Give the current message to the calling process - the parameter

154 \* we got is a pointer, fill it. 155 \*/

156 i = device\_read(file, (char \_\_user \*)ioctl\_param, 99, &offset); 157

158 /\* Put a zero at the end of the buffer, so it will be properly

159 \* terminated.

160 \*/

161 put\_user('\0', (char \_\_user \*)ioctl\_param + i); 162 break;

163 }

164 case IOCTL\_GET\_NTH\_BYTE:

165 /\* This ioctl is both input (ioctl\_param) and output (the return

166 \* value of this function). 167 \*/

168 ret = (long)message[ioctl\_param]; 169 break;

170 }

171

172 /\* We're now ready for our next caller \*/ 173 atomic\_set(&already\_open, CDEV\_NOT\_USED); 174

175 return ret;

176 }

177

178 /\* Module Declarations \*/

179

180 /\* This structure will hold the functions to be called when a process does

181 \* something to the device we created. Since a pointer to this structure

182 \* is kept in the devices table, it can't be local to init\_module. NULL is

183 \* for unimplemented functions.

184 \*/

185 static struct file\_operations fops = { 186 .read = device\_read,

187 .write = device\_write,

188 .unlocked\_ioctl = device\_ioctl, 189 .open = device\_open,

190 .release = device\_release, /\* a.k.a. close \*/ 191 };

192

193 /\* Initialize the module - Register the character device \*/ 194 static int \_\_init chardev2\_init(void) 195 {

196 /\* Register the character device (at least try) \*/ 197 int ret\_val = register\_chrdev(MAJOR\_NUM, DEVICE\_NAME, &fops); 198

199 /\* Negative values signify an error \*/ 200 if (ret\_val < 0) {

201 pr\_alert("%s failed with %d\n", 202 "Sorry, registering the character device ", ret\_val); 203 return ret\_val;

204 }

205

206 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 207 cls = class\_create(DEVICE\_FILE\_NAME); 208 #else

209 cls = class\_create(THIS\_MODULE, DEVICE\_FILE\_NAME); 210 #endif

211 device\_create(cls, NULL, MKDEV(MAJOR\_NUM, 0), NULL, DEVICE\_FILE\_NAME);

212

213 pr\_info("Device created on /dev/%s\n", DEVICE\_FILE\_NAME); 214

215 return 0;

216 }

217

218 /\* Cleanup - unregister the appropriate file from /proc \*/ 219 static void \_\_exit chardev2\_exit(void) 220 {

221 device\_destroy(cls, MKDEV(MAJOR\_NUM, 0)); 222 class\_destroy(cls);

223

224 /\* Unregister the device \*/

225 unregister\_chrdev(MAJOR\_NUM, DEVICE\_NAME); 226 }

227

228 module\_init(chardev2\_init);

229 module\_exit(chardev2\_exit);

230

231 MODULE\_LICENSE("GPL");

1 /\*

2 \* chardev.h - the header file with the ioctl definitions. 3 \*

4 \* The declarations here have to be in a header file, because they need

5 \* to be known both to the kernel module (in chardev2.c) and the process

6 \* calling ioctl() (in userspace\_ioctl.c). 7 \*/

8

9 #ifndef CHARDEV\_H

10 #define CHARDEV\_H

11

12 #include

13

14 /\* The major device number. We can not rely on dynamic registration 15 \* any more, because ioctls need to know it. 16 \*/

17 #define MAJOR\_NUM 100

18

19 /\* Set the message of the device driver \*/ 20 #define IOCTL\_SET\_MSG \_IOW(MAJOR\_NUM, 0, char \*) 21 /\* \_IOW means that we are creating an ioctl command number for passing

22 \* information from a user process to the kernel module. 23 \*

24 \* The first arguments, MAJOR\_NUM, is the major device number we are using.

25 \*

26 \* The second argument is the number of the command (there could be several

27 \* with different meanings).

28 \*

29 \* The third argument is the type we want to get from the process to the

30 \* kernel.

31 \*/

32

33 /\* Get the message of the device driver \*/ 34 #define IOCTL\_GET\_MSG \_IOR(MAJOR\_NUM, 1, char \*) 35 /\* This IOCTL is used for output, to get the message of the device driver.

36 \* However, we still need the buffer to place the message in to be input,

37 \* as it is allocated by the process. 38 \*/

39

40 /\* Get the n'th byte of the message \*/ 41 #define IOCTL\_GET\_NTH\_BYTE \_IOWR(MAJOR\_NUM, 2, int) 42 /\* The IOCTL is used for both input and output. It receives from the user

43 \* a number, n, and returns message[n]. 44 \*/

45

46 /\* The name of the device file \*/ 47 #define DEVICE\_FILE\_NAME "char\_dev" 48 #define DEVICE\_PATH "/dev/char\_dev" 49

50 #endif

1 /\* userspace\_ioctl.c - the process to use ioctl's to control the kernel

, → module

2 \*

3 \* Until now we could have used cat for input and output. But now 4 \* we need to do ioctl's, which require writing our own process. 5 \*/

6

7 /\* device specifics, such as ioctl numbers and the 8 \* major device file. \*/

9 #include "../chardev.h"

10

11 #include /\* standard I/O \*/ 12 #include /\* open \*/

13 #include /\* close \*/

14 #include /\* exit \*/

15 #include /\* ioctl \*/ 16

17 /\* Functions for the ioctl calls \*/ 18

19 int ioctl\_set\_msg(int file\_desc, char \*message) 20 {

21 int ret\_val;

22

23 ret\_val = ioctl(file\_desc, IOCTL\_SET\_MSG, message); 24

25 if (ret\_val < 0) {

26 printf("ioctl\_set\_msg failed:%d\n", ret\_val);

27 }

28

29 return ret\_val;

30 }

31

32 int ioctl\_get\_msg(int file\_desc)

33 {

34 int ret\_val;

35 char message[100] = { 0 };

36

37 /\* Warning - this is dangerous because we don't tell 38 \* the kernel how far it's allowed to write, so it 39 \* might overflow the buffer. In a real production 40 \* program, we would have used two ioctls - one to tell 41 \* the kernel the buffer length and another to give 42 \* it the buffer to fill

43 \*/

44 ret\_val = ioctl(file\_desc, IOCTL\_GET\_MSG, message); 45

46 if (ret\_val < 0) {

47 printf("ioctl\_get\_msg failed:%d\n", ret\_val);

48 }

49 printf("get\_msg message:%s", message); 50

51 return ret\_val;

52 }

53

54 int ioctl\_get\_nth\_byte(int file\_desc) 55 {

56 int i, c;

57

58 printf("get\_nth\_byte message:");

59

60 i = 0;

61 do {

62 c = ioctl(file\_desc, IOCTL\_GET\_NTH\_BYTE, i++);

63

64 if (c < 0) {

65 printf("\nioctl\_get\_nth\_byte failed at the %d'th byte:\n", i);

66 return c;

67 }

68

69 putchar(c);

70 } while (c != 0);

71

72 return 0;

73 }

74

75 /\* Main - Call the ioctl functions \*/ 76 int main(void)

77 {

78 int file\_desc, ret\_val;

79 char \*msg = "Message passed by ioctl\n";

80

81 file\_desc = open(DEVICE\_PATH, O\_RDWR); 82 if (file\_desc < 0) {

83 printf("Can't open device file: %s, error:%d\n", DEVICE\_PATH, 84 file\_desc);

85 exit(EXIT\_FAILURE);

86 }

87

88 ret\_val = ioctl\_set\_msg(file\_desc, msg); 89 if (ret\_val)

90 goto error;

91 ret\_val = ioctl\_get\_nth\_byte(file\_desc); 92 if (ret\_val)

93 goto error;

94 ret\_val = ioctl\_get\_msg(file\_desc);

95 if (ret\_val)

96 goto error;

97

98 close(file\_desc);

99 return 0;

100 error:

101 close(file\_desc);

102 exit(EXIT\_FAILURE);

103 }

10 System Calls

So far, the only thing we’ve done was to use well defined kernel mechanisms to register /proc files and device handlers. This is fine if you want to do something the kernel programmers thought you’d want, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you are mostly on your own.

Notice that this example has been unavailable since Linux v6.9. Specifically,

after this [commit,](https://github.com/torvalds/linux/commit/1e3ad78334a69b36e107232e337f9d693dcc9df2#diff-4a16bf89a09b4f49669a30d54540f0b936ea0224dc6ee9edfa7700deb16c3e11R52) due to the system call table changing the implementation from an indirect function call table to a switch statement for security issues,

such as Branch History Injection (BHI) attack. See more information [here.](https://bugs.launchpad.net/ubuntu/+source/linux/+bug/2060909)

Should one choose not to use a virtual machine, kernel programming can

become risky. For example, while writing the code below, the open() system call was inadvertently disrupted. This resulted in an inability to open any files, run programs, or shut down the system, necessitating a restart of the virtual machine. Fortunately, no critical files were lost in this instance. However, if such modifications were made on a live, mission-critical system, the consequences could be severe. To mitigate the risk of file loss, even in a test environment, it

is advised to execute sync right before using insmod and rmmod.

Forget about /proc files, forget about device files. They are just minor

details. Minutiae in the vast expanse of the universe. The real process to kernel communication mechanism, the one used by all processes, is system calls. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism used. If you want to change the behaviour of the kernel in interesting ways, this is the place to do it. By the way, if you want to see which system calls a program uses, run strace .

In general, a process is not supposed to be able to access the kernel. It can

not access kernel memory and it can’t call kernel functions. The hardware of the CPU enforces this (that is the reason why it is called “protected mode” or “page protection”).

System calls are an exception to this general rule. What happens is that the

process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel — and therefore you’re allowed to do whatever you want.

The location in the kernel a process can jump to is called system\_call.

The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (sys\_call\_table) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it is at the source file arch/$(architecture)/kernel/entry.S, after the line ENTRY(system\_call).

So, if we want to change the way a certain system call works, what we need

to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at sys\_call\_table to point to our function. Because we might be removed later and we don’t want to leave the system in an unstable state, it’s important for cleanup\_module to restore the table to its original state.

To modify the content of sys\_call\_table, we need to consider the control

register. A control register is a processor register that changes or controls the general behavior of the CPU. For x86 architecture, the cr0 register has various control flags that modify the basic operation of the processor. The WP flag in cr0 stands for write protection. Once the WP flag is set, the processor disallows fur-ther write attempts to the read-only sections. Therefore, we must disable the WP flag before modifying sys\_call\_table. Since Linux v5.3, the write\_cr0 function cannot be used because of the sensitive cr0 bits pinned by the security issue, the attacker may write into CPU control registers to disable CPU protec-tions like write protection. As a result, we have to provide the custom assembly routine to bypass it.

However, sys\_call\_table symbol is unexported to prevent misuse. But

there have few ways to get the symbol, manual symbol lookup and kallsyms\_lookup\_name. Here we use both depend on the kernel version.

Because of the control-flow integrity, which is a technique to prevent the redi-

rect execution code from the attacker, for making sure that the indirect calls go to the expected addresses and the return addresses are not changed. Since Linux v5.7, the kernel patched the series of control-flow enforcement (CET) for x86, and some configurations of GCC, like GCC versions 9 and 10 in Ubuntu Linux, will add with CET (the-fcf-protection option) in the kernel by de-fault. Using that GCC to compile the kernel with retpoline off may result in CET being enabled in the kernel. You can use the following command to check out the-fcf-protection option is enabled or not:

$ gcc -v -Q -O2 --help=target | grep protection

Using built-in specs.

COLLECT\_GCC=gcc

COLLECT\_LTO\_WRAPPER=/usr/lib/gcc/x86\_64-linux-gnu/9/lto-wrapper ...

gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)

COLLECT\_GCC\_OPTIONS='-v' '-Q' '-O2' '--help=target' '-mtune=generic' '-march=x86-64'

/usr/lib/gcc/x86\_64-linux-gnu/9/cc1 -v ... -fcf-protection ... GNU C17 (Ubuntu 9.3.0-17ubuntu1~20.04) version 9.3.0 (x86\_64-linux-gnu)

...

But CET should not be enabled in the kernel, it may break the Kprobes and bpf. Consequently, CET is disabled since v5.11. To guarantee the manual symbol lookup worked, we only use up to v5.4.

Unfortunately, since Linux v5.7 kallsyms\_lookup\_name is also unexported,

it needs certain trick to get the address of kallsyms\_lookup\_name. If CONFIG\_KPROBES is enabled, we can facilitate the retrieval of function addresses by means of Kprobes to dynamically break into the specific kernel routine. Kprobes inserts a breakpoint at the entry of function by replacing the first bytes of the probed instruction. When a CPU hits the breakpoint, registers are stored, and the control will pass to Kprobes. It passes the addresses of the saved registers and the Kprobe struct to the handler you defined, then executes it. Kprobes can be registered by symbol name or address. Within the symbol name, the address will be handled by the kernel.

Otherwise, specify the address of sys\_call\_table from /proc/kallsyms

and /boot/System.map into sym parameter. Following is the sample usage for /proc/kallsyms:

$ sudo grep sys\_call\_table /proc/kallsyms

ffffffff82000280 R x32\_sys\_call\_table

ffffffff820013a0 R sys\_call\_table

ffffffff820023e0 R ia32\_sys\_call\_table

$ sudo insmod syscall-steal.ko sym=0xffffffff820013a0

Using the address from /boot/System.map, be careful about KASLR (Ker-

nel Address Space Layout Randomization). KASLR may randomize the address of kernel code and data at every boot time, such as the static address listed in /boot/System.map will offset by some entropy. The purpose of KASLR is to protect the kernel space from the attacker. Without KASLR, the attacker may find the target address in the fixed address easily. Then the attacker can use return-oriented programming to insert some malicious codes to execute or receive the target data by a tampered pointer. KASLR mitigates these kinds of attacks because the attacker cannot immediately know the target address, but a brute-force attack can still work. If the address of a symbol in /proc/kallsyms is different from the address in /boot/System.map, KASLR is enabled with the kernel, which your system running on.

$ grep GRUB\_CMDLINE\_LINUX\_DEFAULT /etc/default/grub

GRUB\_CMDLINE\_LINUX\_DEFAULT="quiet splash"

$ sudo grep sys\_call\_table /boot/System.map-$(uname -r) ffffffff82000300 R sys\_call\_table

$ sudo grep sys\_call\_table /proc/kallsyms

ffffffff820013a0 R sys\_call\_table

# Reboot

$ sudo grep sys\_call\_table /boot/System.map-$(uname -r) ffffffff82000300 R sys\_call\_table

$ sudo grep sys\_call\_table /proc/kallsyms

ffffffff86400300 R sys\_call\_table

If KASLR is enabled, we have to take care of the address from /proc/kallsyms each time we reboot the machine. In order to use the address from /boot/System.map, make sure that KASLR is disabled. You can add the nokaslr for disabling KASLR in next booting time:

$ grep GRUB\_CMDLINE\_LINUX\_DEFAULT /etc/default/grub

GRUB\_CMDLINE\_LINUX\_DEFAULT="quiet splash"

$ sudo perl -i -pe 'm/quiet/ and s//quiet nokaslr/' /etc/default/grub $ grep quiet /etc/default/grub

GRUB\_CMDLINE\_LINUX\_DEFAULT="quiet nokaslr splash"

$ sudo update-grub

For more information, check out the following:

• [Cook: Security things in Linux v5.3](https://lwn.net/Articles/804849/)

• [Unexporting the system call table](https://lwn.net/Articles/12211/)

• [Control-flow integrity for the kernel](https://lwn.net/Articles/810077/)

• [Unexporting kallsyms\_lookup\_name()](https://lwn.net/Articles/813350/)

• [Kernel Probes (Kprobes)](https://www.kernel.org/doc/Documentation/kprobes.txt)

• [Kernel address space layout randomization](https://lwn.net/Articles/569635/)

The source code here is an example of such a kernel module. We want to

“spy” on a certain user, and to pr\_info() a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called our\_sys\_openat. This function checks the uid (user’s id) of the current process, and if it is equal to the uid we spy on, it calls pr\_info() to display the name of the file to be opened. Then, either way, it calls the original openat() function with the same parameters, to actually open the file.

The init\_module function replaces the appropriate location in sys\_call\_table

and keeps the original pointer in a variable. The cleanup\_module function uses that variable to restore everything back to normal. This approach is danger-ous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A’s openat system call will be A\_openat and B’s will be B\_openat. Now, when A is inserted into the kernel, the system call is replaced with A\_openat, which will call the original sys\_openat when it is done. Next, B is inserted into the kernel, which replaces the system call with B\_openat, which will call what it thinks is the original system call, A\_openat, when it’s done.

Now, if B is removed first, everything will be well — it will simply restore

the system call to A\_openat, which calls the original. However, if A is removed and then B is removed, the system will crash. A’s removal will restore the system call to the original, sys\_openat, cutting B out of the loop. Then, when B is removed, it will restore the system call to what it thinks is the original, A\_openat, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won’t change the system call when it is removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to B\_openat so that it is no longer pointing to A\_openat, so it will not restore it to sys\_openat before it is removed from memory. Unfortunately, B\_openat will still try to call A\_openat which is no longer there, so that even without removing B the system would crash.

For x86 architecture, the system call table cannot be used to invoke a system

call after commit [1e3ad78](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e3ad78334a69b36e107232e337f9d693dcc9df2) since v6.9. This commit has been backported to long term stable kernels, like v5.15.154+, v6.1.85+, v6.6.26+ and v6.8.5+, see this

[answer](https://stackoverflow.com/a/78607015) for more details. In this case, thanks to Kprobes, a hook can be used instead on the system call entry to intercept the system call.

Note that all the related problems make syscall stealing unfeasible for pro-

duction use. In order to keep people from doing potentially harmful things sys\_call\_table is no longer exported. This means, if you want to do some-thing more than a mere dry run of this example, you will have to patch your current kernel in order to have sys\_call\_table exported.

1 /\*

2 \* syscall-steal.c

3 \*

4 \* System call "stealing" sample. 5 \*

6 \* Disables page protection at a processor level by changing the 16th bit

7 \* in the cr0 register (could be Intel specific). 8 \*/

9

10 #include

11 #include

12 #include

13 #include /\* which will have params \*/ 14 #include /\* The list of system calls \*/ 15 #include /\* For current\_uid() \*/ 16 #include /\* For \_\_kuid\_val() \*/ 17 #include

18

19 /\* For the current (process) structure, we need this to know who the 20 \* current user is.

21 \*/

22 #include

23 #include

24

25 /\* The way we access "sys\_call\_table" varies as kernel internal changes. 26 \* - Prior to v5.4 : manual symbol lookup 27 \* - v5.5 to v5.6 : use kallsyms\_lookup\_name() 28 \* - v5.7+ : Kprobes or specific kernel module parameter 29 \*/

30

31 /\* The in-kernel calls to the ksys\_close() syscall were removed in Linux

, → v5.11+.

32 \*/

33 #if (LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 7, 0)) 34

35 #if defined(CONFIG\_KPROBES)

36 #define HAVE\_KPROBES 1

37 #if defined(CONFIG\_X86\_64)

38 /\* If you have tried to use the syscall table to intercept syscalls and it

39 \* doesn't work, you can try to use Kprobes to intercept syscalls. 40 \* Set USE\_KPROBES\_PRE\_HANDLER\_BEFORE\_SYSCALL to 1 to register a pre-handler

41 \* before the syscall.

42 \*/

43 #define USE\_KPROBES\_PRE\_HANDLER\_BEFORE\_SYSCALL 0 44 #endif

45 #include

46 #else

47 #define HAVE\_PARAM 1

48 #include /\* For sprint\_symbol \*/ 49 /\* The address of the sys\_call\_table, which can be obtained with looking up

50 \* "/boot/System.map" or "/proc/kallsyms". When the kernel version is v5.7+,

51 \* without CONFIG\_KPROBES, you can input the parameter or the module will look

52 \* up all the memory.

53 \*/

54 static unsigned long sym = 0;

55 module\_param(sym, ulong, 0644);

56 #endif /\* CONFIG\_KPROBES \*/

57

58 #else

59

60 #if LINUX\_VERSION\_CODE <= KERNEL\_VERSION(5, 4, 0) 61 #define HAVE\_KSYS\_CLOSE 1

62 #include /\* For ksys\_close() \*/ 63 #else

64 #include /\* For kallsyms\_lookup\_name \*/ 65 #endif

66

67 #endif /\* Version >= v5.7 \*/

68

69 /\* UID we want to spy on - will be filled from the command line. \*/ 70 static uid\_t uid = -1;

71 module\_param(uid, int, 0644);

72

73 #if USE\_KPROBES\_PRE\_HANDLER\_BEFORE\_SYSCALL 74

75 /\* syscall\_sym is the symbol name of the syscall to spy on. The default is

76 \* "\_\_x64\_sys\_openat", which can be changed by the module parameter. You can

77 \* look up the symbol name of a syscall in /proc/kallsyms. 78 \*/

79 static char \*syscall\_sym = "\_\_x64\_sys\_openat"; 80 module\_param(syscall\_sym, charp, 0644); 81

82 static int sys\_call\_kprobe\_pre\_handler(struct kprobe \*p, struct pt\_regs \*regs) 83 {

84 if (\_\_kuid\_val(current\_uid()) != uid) { 85 return 0;

86 }

87

88 pr\_info("%s called by %d\n", syscall\_sym, uid); 89 return 0;

90 }

91

92 static struct kprobe syscall\_kprobe = { 93 .symbol\_name = "\_\_x64\_sys\_openat", 94 .pre\_handler = sys\_call\_kprobe\_pre\_handler, 95 };

96

97 #else

98

99 static unsigned long \*\*sys\_call\_table\_stolen;

100

101 /\* A pointer to the original system call. The reason we keep this, rather

102 \* than call the original function (sys\_openat), is because somebody else

103 \* might have replaced the system call before us. Note that this is not 104 \* 100% safe, because if another module replaced sys\_openat before us, 105 \* then when we are inserted, we will call the function in that module -106 \* and it might be removed before we are. 107 \*

108 \* Another reason for this is that we can not get sys\_openat. 109 \* It is a static variable, so it is not exported. 110 \*/

111 #ifdef CONFIG\_ARCH\_HAS\_SYSCALL\_WRAPPER 112 static asmlinkage long (\*original\_call)(const struct pt\_regs \*); 113 #else

114 static asmlinkage long (\*original\_call)(int, const char \_\_user \*, int,

, → umode\_t);

115 #endif

116

117 /\* The function we will replace sys\_openat (the function called when you 118 \* call the open system call) with. To find the exact prototype, with 119 \* the number and type of arguments, we find the original function first

120 \* (it is at fs/open.c).

121 \*

122 \* In theory, this means that we are tied to the current version of the

123 \* kernel. In practice, the system calls almost never change (it would

124 \* wreck havoc and require programs to be recompiled, since the system

125 \* calls are the interface between the kernel and the processes). 126 \*/

127 #ifdef CONFIG\_ARCH\_HAS\_SYSCALL\_WRAPPER 128 static asmlinkage long our\_sys\_openat(const struct pt\_regs \*regs) 129 #else

130 static asmlinkage long our\_sys\_openat(int dfd, const char \_\_user \*filename, 131 int flags, umode\_t mode) 132 #endif

133 {

134 int i = 0;

135 char ch;

136

137 if (\_\_kuid\_val(current\_uid()) != uid) 138 goto orig\_call;

139

140 /\* Report the file, if relevant \*/ 141 pr\_info("Opened file by %d: ", uid); 142 do {

143 #ifdef CONFIG\_ARCH\_HAS\_SYSCALL\_WRAPPER 144 get\_user(ch, (char \_\_user \*)regs->si + i); 145 #else

146 get\_user(ch, (char \_\_user \*)filename + i); 147 #endif

148 i++;

149 pr\_info("%c", ch);

150 } while (ch != 0);

151 pr\_info("\n");

152

153 orig\_call:

154 /\* Call the original sys\_openat - otherwise, we lose the ability to

155 \* open files.

156 \*/

157 #ifdef CONFIG\_ARCH\_HAS\_SYSCALL\_WRAPPER 158 return original\_call(regs);

159 #else

160 return original\_call(dfd, filename, flags, mode); 161 #endif

162 }

163

164 static unsigned long \*\*acquire\_sys\_call\_table(void) 165 {

166 #ifdef HAVE\_KSYS\_CLOSE

167 unsigned long int offset = PAGE\_OFFSET; 168 unsigned long \*\*sct;

169

170 while (offset < ULLONG\_MAX) { 171 sct = (unsigned long \*\*)offset; 172

173 if (sct[\_\_NR\_close] == (unsigned long \*)ksys\_close) 174 return sct;

175

176 offset += sizeof(void \*); 177 }

178

179 return NULL;

180 #endif

181

182 #ifdef HAVE\_PARAM

183 const char sct\_name[15] = "sys\_call\_table"; 184 char symbol[40] = { 0 };

185

186 if (sym == 0) {

187 pr\_alert("For Linux v5.7+, Kprobes is the preferable way to get " 188 "symbol.\n");

189 pr\_info("If Kprobes is absent, you have to specify the address of " 190 "sys\_call\_table symbol\n"); 191 pr\_info("by /boot/System.map or /proc/kallsyms, which contains all the

, → "

192 "symbol addresses, into sym parameter.\n"); 193 return NULL;

194 }

195 sprint\_symbol(symbol, sym);

196 if (!strncmp(sct\_name, symbol, sizeof(sct\_name) - 1)) 197 return (unsigned long \*\*)sym; 198

199 return NULL;

200 #endif

201

202 #ifdef HAVE\_KPROBES

203 unsigned long (\*kallsyms\_lookup\_name)(const char \*name); 204 struct kprobe kp = {

205 .symbol\_name = "kallsyms\_lookup\_name", 206 };

207

208 if (register\_kprobe(&kp) < 0) 209 return NULL;

210 kallsyms\_lookup\_name = (unsigned long (\*)(const char \*name))kp.addr; 211 unregister\_kprobe(&kp);

212 #endif

213

214 return (unsigned long \*\*)kallsyms\_lookup\_name("sys\_call\_table"); 215 }

216

217 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 3, 0) 218 static inline void \_\_write\_cr0(unsigned long cr0) 219 {

220 asm volatile("mov %0,%%cr0" : "+r"(cr0) : : "memory"); 221 }

222 #else

223 #define \_\_write\_cr0 write\_cr0

224 #endif

225

226 static void enable\_write\_protection(void) 227 {

228 unsigned long cr0 = read\_cr0(); 229 set\_bit(16, &cr0);

230 \_\_write\_cr0(cr0);

231 }

232

233 static void disable\_write\_protection(void) 234 {

235 unsigned long cr0 = read\_cr0(); 236 clear\_bit(16, &cr0);

237 \_\_write\_cr0(cr0);

238 }

239 #endif

240

241 static int \_\_init syscall\_steal\_start(void) 242 {

243 #if USE\_KPROBES\_PRE\_HANDLER\_BEFORE\_SYSCALL 244 int err;

245 /\* use symbol name from the module parameter \*/ 246 syscall\_kprobe.symbol\_name = syscall\_sym; 247 err = register\_kprobe(&syscall\_kprobe); 248 if (err) {

249 pr\_err("register\_kprobe() on %s failed: %d\n", syscall\_sym, err); 250 pr\_err("Please check the symbol name from 'syscall\_sym'

, → parameter.\n");

251 return err;

252 }

253 #else

254 if (!(sys\_call\_table\_stolen = acquire\_sys\_call\_table())) 255 return-1;

256

257 disable\_write\_protection();

258

259 /\* keep track of the original open function \*/ 260 original\_call = (void \*)sys\_call\_table\_stolen[\_\_NR\_openat]; 261

262 /\* use our openat function instead \*/ 263 sys\_call\_table\_stolen[\_\_NR\_openat] = (unsigned long \*)our\_sys\_openat; 264

265 enable\_write\_protection();

266 #endif

267

268 pr\_info("Spying on UID:%d\n", uid); 269 return 0;

270 }

271

272 static void \_\_exit syscall\_steal\_end(void) 273 {

274 #if USE\_KPROBES\_PRE\_HANDLER\_BEFORE\_SYSCALL 275 unregister\_kprobe(&syscall\_kprobe); 276 #else

277 if (!sys\_call\_table\_stolen)

278 return;

279

280 /\* Return the system call back to normal \*/ 281 if (sys\_call\_table\_stolen[\_\_NR\_openat] != (unsigned long \*)our\_sys\_openat)

, → {

282 pr\_alert("Somebody else also played with the "); 283 pr\_alert("open system call\n"); 284 pr\_alert("The system may be left in "); 285 pr\_alert("an unstable state.\n");

286 }

287

288 disable\_write\_protection();

289 sys\_call\_table\_stolen[\_\_NR\_openat] = (unsigned long \*)original\_call; 290 enable\_write\_protection();

291 #endif

292

293 msleep(2000);

294 }

295

296 module\_init(syscall\_steal\_start); 297 module\_exit(syscall\_steal\_end);

298

299 MODULE\_LICENSE("GPL");

11 Blocking Processes and threads

11.1 Sleep

What do you do when somebody asks you for something you can not do right away? If you are a human being and you are bothered by a human being, the only thing you can say is: "Not right now, I’m busy. Go away! ". But if you are a kernel module and you are bothered by a process, you have another possibility. You can put the process to sleep until you can service it. After all, processes are being put to sleep by the kernel and woken up all the time (that is the way multiple processes appear to run on the same time on a single CPU).

This kernel module is an example of this. The file (called /proc/sleep) can

only be opened by a single process at a time. If the file is already open, the kernel module calls wait\_event\_interruptible. The easiest way to keep a file open is to open it with:

1 tail -f

This function changes the status of the task (a task is the kernel data struc-

ture which holds information about a process and the system call it is in, if any) to TASK\_INTERRUPTIBLE, which means that the task will not run until it is woken up somehow, and adds it to WaitQ, the queue of tasks waiting to access the file. Then, the function calls the scheduler to context switch to a different process, one which has some use for the CPU.

When a process is done with the file, it closes it, and module\_close is called.

That function wakes up all the processes in the queue (there’s no mechanism to only wake up one of them). It then returns and the process which just closed the file can continue to run. In time, the scheduler decides that that process has had enough and gives control of the CPU to another process. Eventually, one of the processes which was in the queue will be given control of the CPU by the sched-uler. It starts at the point right after the call to wait\_event\_interruptible.

# This means that the process is still in kernel mode - as far as the process is

concerned, it issued the open system call and the system call has not returned yet. The process does not know somebody else used the CPU for most of the time between the moment it issued the call and the moment it returned.

It can then proceed to set a global variable to tell all the other processes

that the file is still open and go on with its life. When the other processes get a piece of the CPU, they’ll see that global variable and go back to sleep.

So we will use tail -f to keep the file open in the background, and attempt

to access it with another background process. This way, we don’t need to switch to another terminal window or virtual terminal to run the second process. As soon as the first background process is killed with kill %1 , the second is woken up, is able to access the file and finally terminates.

To make our life more interesting, module\_close does not have a monopoly

on waking up the processes which wait to access the file. A signal, such as Ctrl +c (SIGINT) can also wake up a process. This is because we used wait\_event\_interruptible. We could have used wait\_event instead, but that would have resulted in extremely angry users whose Ctrl+c’s are ignored.

In that case, we want to return with-EINTR immediately. This is important

so users can, for example, kill the process before it receives the file.

There is one more point to remember. Some times processes don’t want to

sleep, they want either to get what they want immediately, or to be told it cannot be done. Such processes use the O\_NONBLOCK flag when opening the file. The kernel is supposed to respond by returning with the error code-EAGAIN from operations which would otherwise block, such as opening the file in this example. The program cat\_nonblock, available in the examples/other directory, can be used to open a file with O\_NONBLOCK.

$ sudo insmod sleep.ko

$ cat\_nonblock /proc/sleep

Last input:

$ tail -f /proc/sleep &

Last input:

Last input:

Last input:

Last input:

Last input:

Last input:

Last input:

tail: /proc/sleep: file truncated

[1] 6540

$ cat\_nonblock /proc/sleep

Open would block

$ kill %1

[1]+ Terminated tail -f /proc/sleep $ cat\_nonblock /proc/sleep

Last input:

$

1 /\*

2 \* sleep.c - create a /proc file, and if several processes try to open it

3 \* at the same time, put all but one to sleep. 4 \*/

5

6 #include

7 #include

8 #include /\* for sprintf() \*/ 9 #include /\* Specifically, a module \*/

10 #include

11 #include /\* Necessary because we use proc fs \*/ 12 #include

13 #include /\* for get\_user and put\_user \*/ 14 #include

15 #include /\* For putting processes to sleep and 16 waking them up \*/ 17

18 #include

19 #include

20

21 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 6, 0) 22 #define HAVE\_PROC\_OPS

23 #endif

24

25 /\* Here we keep the last message received, to prove that we can process our

26 \* input.

27 \*/

28 #define MESSAGE\_LENGTH 80

29 static char message[MESSAGE\_LENGTH]; 30

31 static struct proc\_dir\_entry \*our\_proc\_file; 32 #define PROC\_ENTRY\_FILENAME "sleep" 33

34 /\* Since we use the file operations struct, we can't use the special proc

35 \* output provisions - we have to use a standard read function, which is this

36 \* function.

37 \*/

38 static ssize\_t module\_output(struct file \*file, /\* see include/linux/fs.h \*/ 39 char \_\_user \*buf, /\* The buffer to put data to 40 (in the user segment) \*/ 41 size\_t len, /\* The length of the buffer \*/ 42 loff\_t \*offset) 43 {

44 static int finished = 0;

45 int i;

46 char output\_msg[MESSAGE\_LENGTH + 30];

47

48 /\* Return 0 to signify end of file - that we have nothing more to say

49 \* at this point.

50 \*/

51 if (finished) {

52 finished = 0;

53 return 0;

54 }

55

56 sprintf(output\_msg, "Last input:%s\n", message); 57 for (i = 0; i < len && output\_msg[i]; i++) 58 put\_user(output\_msg[i], buf + i); 59

60 finished = 1;

61 return i; /\* Return the number of bytes "read" \*/ 62 }

63

64 /\* This function receives input from the user when the user writes to the

65 \* /proc file.

66 \*/

67 static ssize\_t module\_input(struct file \*file, /\* The file itself \*/ 68 const char \_\_user \*buf, /\* The buffer with input

,→ \*/

69 size\_t length, /\* The buffer's length \*/

70 loff\_t \*offset) /\* offset to file - ignore \*/

71 {

72 int i;

73

74 /\* Put the input into message, where module\_output will later be able

75 \* to use it.

76 \*/

77 for (i = 0; i < MESSAGE\_LENGTH - 1 && i < length; i++) 78 get\_user(message[i], buf + i); 79 /\* we want a standard, zero terminated string \*/ 80 message[i] = '\0';

81

82 /\* We need to return the number of input characters used \*/ 83 return i;

84 }

85

86 /\* 1 if the file is currently open by somebody \*/ 87 static atomic\_t already\_open = ATOMIC\_INIT(0); 88

89 /\* Queue of processes who want our file \*/ 90 static DECLARE\_WAIT\_QUEUE\_HEAD(waitq); 91

92 /\* Called when the /proc file is opened \*/ 93 static int module\_open(struct inode \*inode, struct file \*file) 94 {

95 /\* Try to get without blocking \*/ 96 if (!atomic\_cmpxchg(&already\_open, 0, 1)) { 97 /\* Success without blocking, allow the access \*/ 98 return 0;

99 }

100 /\* If the file's flags include O\_NONBLOCK, it means the process does not

101 \* want to wait for the file. In this case, because the file is already

, → open,

102 \* we should fail with -EAGAIN, meaning "you will have to try again",

103 \* instead of blocking a process which would rather stay awake. 104 \*/

105 if (file->f\_flags & O\_NONBLOCK) 106 return-EAGAIN;

107

108 while (atomic\_cmpxchg(&already\_open, 0, 1)) { 109 int i, is\_sig = 0;

110

111 /\* This function puts the current process, including any system 112 \* calls, such as us, to sleep. Execution will be resumed right 113 \* after the function call, either because somebody called 114 \* wake\_up(&waitq) (only module\_close does that, when the file 115 \* is closed) or when a signal, such as Ctrl-C, is sent 116 \* to the process

117 \*/

118 wait\_event\_interruptible(waitq, !atomic\_read(&already\_open)); 119

120 /\* If we woke up because we got a signal we're not blocking, 121 \* return -EINTR (fail the system call). This allows processes 122 \* to be killed or stopped. 123 \*/

124 for (i = 0; i < \_NSIG\_WORDS && !is\_sig; i++) 125 is\_sig = current->pending.signal.sig[i] &

, → ~current->blocked.sig[i];

126

127 if (is\_sig) {

128 /\* Return -EINTR if we got a signal \*/ 129 return-EINTR;

130 }

131 }

132

133 return 0; /\* Allow the access \*/ 134 }

135

136 /\* Called when the /proc file is closed \*/ 137 static int module\_close(struct inode \*inode, struct file \*file) 138 {

139 /\* Set already\_open to zero, so one of the processes in the waitq will

140 \* be able to set already\_open back to one and to open the file. All 141 \* the other processes will be called when already\_open is back to one,

142 \* so they'll go back to sleep. 143 \*/

144 atomic\_set(&already\_open, 0); 145

146 /\* Wake up all the processes in waitq, so if anybody is waiting for the

147 \* file, they can have it.

148 \*/

149 wake\_up(&waitq);

150

151 return 0; /\* success \*/

152 }

153

154 /\* Structures to register as the /proc file, with pointers to all the relevant

155 \* functions.

156 \*/

157

158 /\* File operations for our /proc file. This is where we place pointers to all

159 \* the functions called when somebody tries to do something to our file. NULL

160 \* means we don't want to deal with something. 161 \*/

162 #ifdef HAVE\_PROC\_OPS

163 static const struct proc\_ops file\_ops\_4\_our\_proc\_file = { 164 .proc\_read = module\_output, /\* "read" from the file \*/ 165 .proc\_write = module\_input, /\* "write" to the file \*/ 166 .proc\_open = module\_open, /\* called when the /proc file is opened \*/ 167 .proc\_release = module\_close, /\* called when it's closed \*/ 168 .proc\_lseek = noop\_llseek, /\* return file->f\_pos \*/ 169 };

170 #else

171 static const struct file\_operations file\_ops\_4\_our\_proc\_file = { 172 .read = module\_output,

173 .write = module\_input,

174 .open = module\_open,

175 .release = module\_close,

176 .llseek = noop\_llseek,

177 };

178 #endif

179

180 /\* Initialize the module - register the /proc file \*/ 181 static int \_\_init sleep\_init(void) 182 {

183 our\_proc\_file =

184 proc\_create(PROC\_ENTRY\_FILENAME, 0644, NULL,

,→ &file\_ops\_4\_our\_proc\_file);

185 if (our\_proc\_file == NULL) {

186 pr\_debug("Error: Could not initialize /proc/%s\n",

, → PROC\_ENTRY\_FILENAME);

187 return-ENOMEM;

188 }

189 proc\_set\_size(our\_proc\_file, 80); 190 proc\_set\_user(our\_proc\_file, GLOBAL\_ROOT\_UID, GLOBAL\_ROOT\_GID); 191

192 pr\_info("/proc/%s created\n", PROC\_ENTRY\_FILENAME); 193

194 return 0;

195 }

196

197 /\* Cleanup - unregister our file from /proc. This could get dangerous if

198 \* there are still processes waiting in waitq, because they are inside our

199 \* open function, which will get unloaded. I'll explain how to avoid removal

200 \* of a kernel module in such a case in chapter 10. 201 \*/

202 static void \_\_exit sleep\_exit(void) 203 {

204 remove\_proc\_entry(PROC\_ENTRY\_FILENAME, NULL); 205 pr\_debug("/proc/%s removed\n", PROC\_ENTRY\_FILENAME); 206 }

207

208 module\_init(sleep\_init);

209 module\_exit(sleep\_exit);

210

211 MODULE\_LICENSE("GPL");

1 /\*

2 \* cat\_nonblock.c - open a file and display its contents, but exit rather

, → than

3 \* wait for input.

4 \*/

5 #include /\* for errno \*/ 6 #include /\* for open \*/ 7 #include /\* standard I/O \*/ 8 #include /\* for exit \*/

9 #include /\* for read \*/

10

11 #define MAX\_BYTES 1024 \* 4

12

13 int main(int argc, char \*argv[])

14 {

15 int fd; /\* The file descriptor for the file to read \*/ 16 size\_t bytes; /\* The number of bytes read \*/ 17 char buffer[MAX\_BYTES]; /\* The buffer for the bytes \*/ 18

19 /\* Usage \*/

20 if (argc != 2) {

21 printf("Usage: %s \n", argv[0]); 22 puts("Reads the content of a file, but doesn't wait for input");

23 exit(EXIT\_FAILURE);

24 }

25

26 /\* Open the file for reading in non blocking mode \*/ 27 fd = open(argv[1], O\_RDONLY | O\_NONBLOCK); 28

29 /\* If open failed \*/

30 if (fd == -1) {

31 puts(errno == EAGAIN ? "Open would block" : "Open failed"); 32 exit(EXIT\_FAILURE);

33 }

34

35 /\* Read the file and output its contents \*/ 36 do {

37 /\* Read characters from the file \*/

38 bytes = read(fd, buffer, MAX\_BYTES);

39

40 /\* If there's an error, report it and die \*/

41 if (bytes == -1) {

42 if (errno == EAGAIN)

43 puts("Normally I'd block, but you told me not to");

44 else

45 puts("Another read error");

46 exit(EXIT\_FAILURE);

47 }

48

49 /\* Print the characters \*/

50 if (bytes > 0) {

51 for (int i = 0; i < bytes; i++)

52 putchar(buffer[i]);

53 }

54

55 /\* While there are no errors and the file isn't over \*/ 56 } while (bytes > 0);

57

58 close(fd);

59 return 0;

60 }

11.2 Completions

Sometimes one thing should happen before another within a module having multiple threads. Rather than using /bin/sleep commands, the kernel has another way to do this which allows timeouts or interrupts to also happen.

Completions as code synchronization mechanism have three main parts, ini-

tialization of struct completion synchronization object, the waiting or barrier part through wait\_for\_completion(), and the signalling side through a call to complete().

In the subsequent example, two threads are initiated: crank and flywheel.

It is imperative that the crank thread starts before the flywheel thread. A com-pletion state is established for each of these threads, with a distinct completion defined for both the crank and flywheel threads. At the exit point of each thread the respective completion state is updated, and wait\_for\_completion is used by the flywheel thread to ensure that it does not begin prematurely. The crank thread uses the complete\_all() function to update the completion, which lets the flywheel thread continue.

So even though flywheel\_thread is started first you should notice when

you load this module and run dmesg, that turning the crank always happens first because the flywheel thread waits for the crank thread to complete.

There are other variations of the wait\_for\_completion function, which

include timeouts or being interrupted, but this basic mechanism is enough for many common situations without adding a lot of complexity.

1 /\*

2 \* completions.c

3 \*/

4 #include

5 #include /\* for IS\_ERR() \*/ 6 #include

7 #include

8 #include

9 #include

10 #include

11

12 static struct completion crank\_comp; 13 static struct completion flywheel\_comp; 14

15 static int machine\_crank\_thread(void \*arg) 16 {

17 pr\_info("Turn the crank\n");

18

19 complete\_all(&crank\_comp);

20 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 17, 0) 21 kthread\_complete\_and\_exit(&crank\_comp, 0); 22 #else

23 complete\_and\_exit(&crank\_comp, 0);

24 #endif

25 }

26

27 static int machine\_flywheel\_spinup\_thread(void \*arg) 28 {

29 wait\_for\_completion(&crank\_comp);

30

31 pr\_info("Flywheel spins up\n");

32

33 complete\_all(&flywheel\_comp); 34 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5, 17, 0) 35 kthread\_complete\_and\_exit(&flywheel\_comp, 0);

36 #else

37 complete\_and\_exit(&flywheel\_comp, 0); 38 #endif

39 }

40

41 static int \_\_init completions\_init(void) 42 {

43 struct task\_struct \*crank\_thread;

44 struct task\_struct \*flywheel\_thread;

45

46 pr\_info("completions example\n");

47

48 init\_completion(&crank\_comp);

49 init\_completion(&flywheel\_comp);

50

51 crank\_thread = kthread\_create(machine\_crank\_thread, NULL, "KThread

,→ Crank");

52 if (IS\_ERR(crank\_thread))

53 goto ERROR\_THREAD\_1;

54

55 flywheel\_thread = kthread\_create(machine\_flywheel\_spinup\_thread, NULL,

56 "KThread Flywheel");

57 if (IS\_ERR(flywheel\_thread))

58 goto ERROR\_THREAD\_2;

59

60 wake\_up\_process(flywheel\_thread);

61 wake\_up\_process(crank\_thread);

62

63 return 0;

64

65 ERROR\_THREAD\_2:

66 kthread\_stop(crank\_thread);

67 ERROR\_THREAD\_1:

68

69 return-1;

70 }

71

72 static void \_\_exit completions\_exit(void) 73 {

74 wait\_for\_completion(&crank\_comp);

75 wait\_for\_completion(&flywheel\_comp); 76

77 pr\_info("completions exit\n");

78 }

79

80 module\_init(completions\_init);

81 module\_exit(completions\_exit);

82

83 MODULE\_DESCRIPTION("Completions example");

84 MODULE\_LICENSE("GPL");

12 Synchronization

If processes running on different CPUs or in different threads try to access the same memory, then it is possible that strange things can happen or your system can lock up. To avoid this, various types of mutual exclusion kernel functions are available. These indicate if a section of code is "locked" or "unlocked" so that simultaneous attempts to run it can not happen.

12.1 Mutex

You can use kernel mutexes (mutual exclusions) in much the same manner that you might deploy them in userland. This may be all that is needed to avoid collisions in most cases.

Mutexes in the Linux kernel enforce strict ownership: only the task that

successfully acquired the mutex can release (or unlock) it. Attempting to release a mutex held by another task or releasing an unheld mutex multiple times by the same task typically leads to errors or undefined behavior. If a task tries to lock a mutex it already holds, it may be blocked or sleep, where the task waits for itself to release the lock.

Before use, a mutex must be initialized through specific APIs (such as

mutex\_init or by using the DEFINE\_MUTEX macro for compile-time initializa-tion). And it is prohibited to directly modify the internal structure of a mutex using a memory manipulation function like memset.

1 /\*

2 \* example\_mutex.c

3 \*/

4 #include

5 #include

6 #include

7

8 static DEFINE\_MUTEX(mymutex);

9

10 static int \_\_init example\_mutex\_init(void) 11 {

12 int ret;

13

14 pr\_info("example\_mutex init\n");

15

16 ret = mutex\_trylock(&mymutex);

17 if (ret != 0) {

18 pr\_info("mutex is locked\n");

19

20 if (mutex\_is\_locked(&mymutex) == 0)

21 pr\_info("The mutex failed to lock!\n");

22

23 mutex\_unlock(&mymutex);

24 pr\_info("mutex is unlocked\n");

25 } else

26 pr\_info("Failed to lock\n");

27

28 return 0;

29 }

30

31 static void \_\_exit example\_mutex\_exit(void) 32 {

33 pr\_info("example\_mutex exit\n");

34 }

35

36 module\_init(example\_mutex\_init);

37 module\_exit(example\_mutex\_exit);

38

39 MODULE\_DESCRIPTION("Mutex example"); 40 MODULE\_LICENSE("GPL");

The various suffixes appended to mutex functions in the Linux kernel pri-

marily dictate how a task waiting to acquire a lock will behave, particularly concerning its interruptibility.

When a task calls mutex\_lock(), and if the mutex is currently unavailable,

the task enters a sleep state until it can successfully obtain the lock. During this period, the task cannot be interrupted. In contrast, functions with the \_interruptible suffix, such as mutex\_lock\_interruptible(), behave simi-larly to mutex\_lock() but allow the waiting process to be interrupted by sig-nals. If a task receives a signal (like a termination signal) while waiting for the lock, it will exit the waiting state and return an error code (-EINTR). This is useful for applications that need to handle external events even while waiting for a lock.

Beyond these fundamental locking behaviors, other mutex functions offer

specialized capabilities. Functions like mutex\_lock\_nested and mutex\_lock\_interruptible\_nested() incorporate the \_\_nested() functionality, providing support for nested locking. This prior locking mechanism aids in managing lock acquisition and preventing deadlocks, often employing a subclass parameter for more precise deadlock de-tection. The latter variant combines nested locking with the ability for the wait-ing process to be interrupted by signals. Another function is mutex\_trylock(), which attempts to acquire the mutex without blocking. It returns 1 if the lock is successfully acquired and 0 if the mutex is already held by another task.

Despite the fact that mutex\_trylock does not sleep, it is still generally not

safe for use in interrupt context because its implementation isn’t atomic. If an interrupt occurs between checking the lock’s availability and its acquisition, this can lead to race conditions and potential data corruption.

12.2 Spinlocks

As the name suggests, spinlocks lock up the CPU that the code is running on, taking 100% of its resources. Because of this you should only use the spinlock mechanism around code which is likely to take no more than a few milliseconds

to run and so will not noticeably slow anything down from the user’s point of view.

The example here is "irq safe" in that if interrupts happen during the

lock then they will not be forgotten and will activate when the unlock happens, using the flags variable to retain their state.

1 /\*

2 \* example\_spinlock.c

3 \*/

4 #include

5 #include

6 #include

7 #include

8

9 static DEFINE\_SPINLOCK(sl\_static);

10 static spinlock\_t sl\_dynamic;

11

12 static void example\_spinlock\_static(void) 13 {

14 unsigned long flags;

15

16 spin\_lock\_irqsave(&sl\_static, flags); 17 pr\_info("Locked static spinlock\n"); 18

19 /\* Do something or other safely. Because this uses 100% CPU time, this 20 \* code should take no more than a few milliseconds to run. 21 \*/

22

23 spin\_unlock\_irqrestore(&sl\_static, flags); 24 pr\_info("Unlocked static spinlock\n"); 25 }

26

27 static void example\_spinlock\_dynamic(void) 28 {

29 unsigned long flags;

30

31 spin\_lock\_init(&sl\_dynamic);

32 spin\_lock\_irqsave(&sl\_dynamic, flags); 33 pr\_info("Locked dynamic spinlock\n"); 34

35 /\* Do something or other safely. Because this uses 100% CPU time, this 36 \* code should take no more than a few milliseconds to run. 37 \*/

38

39 spin\_unlock\_irqrestore(&sl\_dynamic, flags); 40 pr\_info("Unlocked dynamic spinlock\n"); 41 }

42

43 static int \_\_init example\_spinlock\_init(void) 44 {

45 pr\_info("example spinlock started\n"); 46

47 example\_spinlock\_static();

48 example\_spinlock\_dynamic();

49

50 return 0;

51 }

52

53 static void \_\_exit example\_spinlock\_exit(void) 54 {

55 pr\_info("example spinlock exit\n"); 56 }

57

58 module\_init(example\_spinlock\_init); 59 module\_exit(example\_spinlock\_exit); 60

61 MODULE\_DESCRIPTION("Spinlock example"); 62 MODULE\_LICENSE("GPL");

Taking 100% of a CPU’s resources comes with greater responsibility. Situ-

ations where the kernel code monopolizes a CPU are called atomic contexts. Holding a spinlock is one of those situations. Sleeping in atomic contexts may leave the system hanging, as the occupied CPU devotes 100% of its resources doing nothing but sleeping. In some worse cases the system may crash. Thus, sleeping in atomic contexts is considered a bug in the kernel. They are some-times called “sleep-in-atomic-context” in some materials.

Note that sleeping here is not limited to calling the sleep functions explicitly.

If subsequent function calls eventually invoke a function that sleeps, it is also considered sleeping. Thus, it is important to pay attention to functions being used in atomic context. There’s no documentation recording all such functions, but code comments may help. Sometimes you may find comments in kernel source code stating that a function “may sleep”, “might sleep”, or more explicitly “the caller should not hold a spinlock”. Those comments are hints that a function may implicitly sleep and must not be called in atomic contexts.

Now, let’s differentiate between a few types of spinlock functions in the

Linux kernel: spin\_lock(), spin\_lock\_irq(), spin\_lock\_irqsave(), and spin\_lock\_bh().

spin\_lock() does not allow the CPU to sleep while waiting for the lock,

which makes it suitable for most use cases where the critical section is short. However, this is problematic for real-time Linux because spinlocks in this con-figuration behave as sleeping locks. This can prevent other tasks from running and cause the system to become unresponsive. To address this in real-time Linux environments, a raw\_spin\_lock() is used, which behaves similarly to a spin\_lock() but without causing the system to sleep.

On the other hand, spin\_lock\_irq() disables interrupts while holding the

lock, but it does not save the interrupt state. This means that if an interrupt occurs while the lock is held, the interrupt state could be lost. In contrast, spin\_lock\_irqsave() disables interrupts and also saves the interrupt state, ensuring that interrupts are restored to their previous state when the lock is released. This makes spin\_lock\_irqsave() a safer option in scenarios where preserving the interrupt state is crucial.

Next, spin\_lock\_bh() disables softirqs (software interrupts) but allows

hardware interrupts to continue. Unlike spin\_lock\_irq() and spin\_lock\_irqsave(),

which disable both hardware and software interrupts, spin\_lock\_bh() is useful when hardware interrupts need to remain active.

For more information about spinlock usage and lock types, see the following

resources:

• [Lesson 1: Spin locks](https://www.kernel.org/doc/Documentation/locking/spinlocks.txt)

• [Lock types and their rules](https://docs.kernel.org/locking/locktypes.html)

12.3 Read and write locks

Read and write locks are specialised kinds of spinlocks so that you can exclu-sively read from something or write to something. Like the earlier spinlocks example, the one below shows an "irq safe" situation in which if other functions were triggered from irqs which might also read and write to whatever you are concerned with then they would not disrupt the logic. As before it is a good idea to keep anything done within the lock as short as possible so that it does not hang up the system and cause users to start revolting against the tyranny of your module.

1 /\*

2 \* example\_rwlock.c

3 \*/

4 #include

5 #include

6 #include

7

8 static DEFINE\_RWLOCK(myrwlock);

9

10 static void example\_read\_lock(void) 11 {

12 unsigned long flags;

13

14 read\_lock\_irqsave(&myrwlock, flags); 15 pr\_info("Read Locked\n");

16

17 /\* Read from something \*/

18

19 read\_unlock\_irqrestore(&myrwlock, flags); 20 pr\_info("Read Unlocked\n");

21 }

22

23 static void example\_write\_lock(void) 24 {

25 unsigned long flags;

26

27 write\_lock\_irqsave(&myrwlock, flags); 28 pr\_info("Write Locked\n");

29

30 /\* Write to something \*/

31

32 write\_unlock\_irqrestore(&myrwlock, flags); 33 pr\_info("Write Unlocked\n");

34 }

35

36 static int \_\_init example\_rwlock\_init(void) 37 {

38 pr\_info("example\_rwlock started\n"); 39

40 example\_read\_lock();

41 example\_write\_lock();

42

43 return 0;

44 }

45

46 static void \_\_exit example\_rwlock\_exit(void) 47 {

48 pr\_info("example\_rwlock exit\n");

49 }

50

51 module\_init(example\_rwlock\_init); 52 module\_exit(example\_rwlock\_exit); 53

54 MODULE\_DESCRIPTION("Read/Write locks example"); 55 MODULE\_LICENSE("GPL");

Of course, if you know for sure that there are no functions triggered by

irqs which could possibly interfere with your logic then you can use the simpler read\_lock(&myrwlock) and read\_unlock(&myrwlock) or the corresponding write functions.

12.4 Atomic operations

If you are doing simple arithmetic: adding, subtracting or bitwise operations, then there is another way in the multi-CPU and multi-hyperthreaded world to stop other parts of the system from messing with your mojo. By using atomic operations you can be confident that your addition, subtraction or bit flip did actually happen and was not overwritten by some other shenanigans. An example is shown below.

1 /\*

2 \* example\_atomic.c

3 \*/

4 #include

5 #include

6 #include

7 #include

8

9 #define BYTE\_TO\_BINARY\_PATTERN "%c%c%c%c%c%c%c%c"

10 #define BYTE\_TO\_BINARY(byte)

, → \

11 ((byte & 0x80) ? '1' : '0'), ((byte & 0x40) ? '1' : '0'),

, → \

12 ((byte & 0x20) ? '1' : '0'), ((byte & 0x10) ? '1' : '0'),

, → \

13 ((byte & 0x08) ? '1' : '0'), ((byte & 0x04) ? '1' : '0'),

, → \

14 ((byte & 0x02) ? '1' : '0'), ((byte & 0x01) ? '1' : '0') 15

16 static void atomic\_add\_subtract(void) 17 {

18 atomic\_t debbie;

19 atomic\_t chris = ATOMIC\_INIT(50);

20

21 atomic\_set(&debbie, 45);

22

23 /\* subtract one \*/

24 atomic\_dec(&debbie);

25

26 atomic\_add(7, &debbie);

27

28 /\* add one \*/

29 atomic\_inc(&debbie);

30

31 pr\_info("chris: %d, debbie: %d\n", atomic\_read(&chris), 32 atomic\_read(&debbie)); 33 }

34

35 static void atomic\_bitwise(void)

36 {

37 unsigned long word = 0;

38

39 pr\_info("Bits 0: " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(word)); 40 set\_bit(3, &word);

41 set\_bit(5, &word);

42 pr\_info("Bits 1: " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(word)); 43 clear\_bit(5, &word);

44 pr\_info("Bits 2: " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(word)); 45 change\_bit(3, &word);

46

47 pr\_info("Bits 3: " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(word)); 48 if (test\_and\_set\_bit(3, &word)) 49 pr\_info("wrong\n");

50 pr\_info("Bits 4: " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(word)); 51

52 word = 255;

53 pr\_info("Bits 5: " BYTE\_TO\_BINARY\_PATTERN "\n", BYTE\_TO\_BINARY(word)); 54 }

55

56 static int \_\_init example\_atomic\_init(void) 57 {

58 pr\_info("example\_atomic started\n"); 59

60 atomic\_add\_subtract();

61 atomic\_bitwise();

62

63 return 0;

64 }

65

66 static void \_\_exit example\_atomic\_exit(void) 67 {

68 pr\_info("example\_atomic exit\n");

69 }

70

71 module\_init(example\_atomic\_init); 72 module\_exit(example\_atomic\_exit); 73

74 MODULE\_DESCRIPTION("Atomic operations example"); 75 MODULE\_LICENSE("GPL");

Before the C11 standard adopted the built-in atomic types, the kernel al-

ready provided a small set of atomic types by using a bunch of tricky architecture-specific codes. Implementing the atomic types by C11 atomics may allow the kernel to throw away the architecture-specific codes and make the kernel code be more friendly to the people who understand the standard. But there are some problems, such as the memory model of the kernel doesn’t match the model formed by the C11 atomics. For further details, see:

• [kernel documentation of atomic types](https://www.kernel.org/doc/Documentation/atomic_t.txt)

• [Time to move to C11 atomics?](https://lwn.net/Articles/691128/)

• [Atomic usage patterns in the kernel](https://lwn.net/Articles/698315/)

13 Replacing Print Macros

13.1 Replacement

In Section [1.7,](#On_Arch_Linux) it was noted that the X Window System and kernel module programming are not conducive to integration. This remains valid during the development of kernel modules. However, in practical scenarios, the necessity emerges to relay messages to the tty (teletype) originating the module load command.

The term “tty” originates from teletype, which initially referred to a com-

bined keyboard-printer for Unix system communication. Today, it signifies a text stream abstraction employed by Unix programs, encompassing physical terminals, xterms in X displays, and network connections like SSH.

To achieve this, the “current” pointer is leveraged to access the active task’s

tty structure. Within this structure lies a pointer to a string write function, facilitating the string’s transmission to the tty.

1 /\*

2 \* print\_string.c - Send output to the tty we're running on, regardless if 3 \* it is through X11, telnet, etc. We do this by printing the string to the

4 \* tty associated with the current task. 5 \*/

6 #include

7 #include

8 #include

9 #include /\* For current \*/

10 #include /\* For the tty declarations \*/ 11

12 static void print\_string(char \*str)

13 {

14 /\* The tty for the current task \*/ 15 struct tty\_struct \*my\_tty = get\_current\_tty();

16

17 /\* If my\_tty is NULL, the current task has no tty you can print to (i.e.,

18 \* if it is a daemon). If so, there is nothing we can do. 19 \*/

20 if (my\_tty) {

21 const struct tty\_operations \*ttyops = my\_tty->driver->ops; 22 /\* my\_tty->driver is a struct which holds the tty's functions, 23 \* one of which (write) is used to write strings to the tty. 24 \* It can be used to take a string either from the user's or 25 \* kernel's memory segment. 26 \*

27 \* The function's 1st parameter is the tty to write to, because the

28 \* same function would normally be used for all tty's of a certain

29 \* type.

30 \* The 2nd parameter is a pointer to a string.

31 \* The 3rd parameter is the length of the string.

32 \*

33 \* As you will see below, sometimes it's necessary to use 34 \* preprocessor stuff to create code that works for different 35 \* kernel versions. The (naive) approach we've taken here does not

36 \* scale well. The right way to deal with this is described in 37 \* section 2 of

38 \* linux/Documentation/SubmittingPatches

39 \*/

40 (ttyops->write)(my\_tty, /\* The tty itself \*/

41 str, /\* String \*/

42 strlen(str)); /\* Length \*/

43

44 /\* ttys were originally hardware devices, which (usually) strictly

45 \* followed the ASCII standard. In ASCII, to move to a new line you

46 \* need two characters, a carriage return and a line feed. On Unix,

47 \* the ASCII line feed is used for both purposes - so we can not

48 \* just use \n, because it would not have a carriage return and the

49 \* next line will start at the column right after the line feed.

50 \*

51 \* This is why text files are different between Unix and MS Windows.

52 \* In CP/M and derivatives, like MS-DOS and MS Windows, the ASCII

53 \* standard was strictly adhered to, and therefore a newline requires

54 \* both a LF and a CR.

55 \*/

56 (ttyops->write)(my\_tty, "\015\012", 2);

57 }

58 }

59

60 static int \_\_init print\_string\_init(void) 61 {

62 print\_string("The module has been inserted. Hello world!"); 63 return 0;

64 }

65

66 static void \_\_exit print\_string\_exit(void) 67 {

68 print\_string("The module has been removed. Farewell world!"); 69 }

70

71 module\_init(print\_string\_init);

72 module\_exit(print\_string\_exit);

73

74 MODULE\_LICENSE("GPL");

13.2 Flashing keyboard LEDs

In certain conditions, you may desire a simpler and more direct way to commu-nicate to the external world. Flashing keyboard LEDs can be such a solution: It is an immediate way to attract attention or to display a status condition. Keyboard LEDs are present on every hardware, they are always visible, they do not need any setup, and their use is rather simple and non-intrusive, compared to writing to a tty or a file.

From v4.14 to v4.15, the timer API made a series of changes to improve

memory safety. A buffer overflow in the area of a timer\_list structure may be able to overwrite the function and data fields, providing the attacker with a way to use return-oriented programming (ROP) to call arbitrary functions within the kernel. Also, the function prototype of the callback, containing a unsigned long argument, will prevent work from any type checking. Further-more, the function prototype with unsigned long argument may be an obstacle to the forward-edge protection of control-flow integrity. Thus, it is better to use a unique prototype to separate from the cluster that takes an unsigned long argument. The timer callback should be passed a pointer to the timer\_list structure rather than an unsigned long argument. Then, it wraps all the in-formation the callback needs, including the timer\_list structure, into a larger structure, and it can use the container\_of macro instead of the unsigned long

value. For more information, see: [Improving the kernel timers API.](https://lwn.net/Articles/735887/)

Before Linux v4.14, setup\_timer was used to initialize the timer and the

timer\_list structure looked like:

1 struct timer\_list {

2 unsigned long expires;

3 void (\*function)(unsigned long); 4 unsigned long data;

5 u32 flags;

6 /\* ... \*/

7 };

8

9 void setup\_timer(struct timer\_list \*timer, void (\*callback)(unsigned long),

10 unsigned long data);

Since Linux v4.14, timer\_setup is adopted and the kernel step by step

converting to timer\_setup from setup\_timer. One of the reasons why the API was changed is that it needed to coexist with the old version of the interface. Moreover, the timer\_setup was implemented by setup\_timer at first.

1 void timer\_setup(struct timer\_list \*timer, 2 void (\*callback)(struct timer\_list \*), unsigned int flags);

The setup\_timer was then removed since v4.15. As a result, the timer\_list

structure had changed to the following.

1 struct timer\_list {

2 unsigned long expires;

3 void (\*function)(struct timer\_list \*); 4 u32 flags;

5 /\* ... \*/

6 };

The following source code illustrates a minimal kernel module which, when

loaded, starts blinking the keyboard LEDs until it is unloaded.

1 /\*

2 \* kbleds.c - Blink keyboard leds until the module is unloaded. 3 \*/

4

5 #include

6 #include /\* For KDSETLED \*/ 7 #include

8 #include /\* For tty\_struct \*/ 9 #include /\* For MAX\_NR\_CONSOLES \*/

10 #include /\* for fg\_console \*/ 11 #include /\* For vc\_cons \*/ 12

13 MODULE\_DESCRIPTION("Example module illustrating the use of Keyboard LEDs."); 14

15 static struct timer\_list my\_timer; 16 static struct tty\_driver \*my\_driver; 17 static unsigned long kbledstatus = 0; 18

19 #define BLINK\_DELAY HZ / 5

20 #define ALL\_LEDS\_ON 0x07

21 #define RESTORE\_LEDS 0xFF

22

23 /\* Function my\_timer\_func blinks the keyboard LEDs periodically by invoking

24 \* command KDSETLED of ioctl() on the keyboard driver. To learn more on

, → virtual

25 \* terminal ioctl operations, please see file: 26 \* drivers/tty/vt/vt\_ioctl.c, function vt\_ioctl(). 27 \*

28 \* The argument to KDSETLED is alternatively set to 7 (thus causing the led

29 \* mode to be set to LED\_SHOW\_IOCTL, and all the leds are lit) and to 0xFF 30 \* (any value above 7 switches back the led mode to LED\_SHOW\_FLAGS, thus 31 \* the LEDs reflect the actual keyboard status). To learn more on this, 32 \* please see file: drivers/tty/vt/keyboard.c, function setledstate(). 33 \*/

34 static void my\_timer\_func(struct timer\_list \*unused) 35 {

36 struct tty\_struct \*t = vc\_cons[fg\_console].d->port.tty;

37

38 if (kbledstatus == ALL\_LEDS\_ON)

39 kbledstatus = RESTORE\_LEDS;

40 else

41 kbledstatus = ALL\_LEDS\_ON;

42

43 (my\_driver->ops->ioctl)(t, KDSETLED, kbledstatus); 44

45 my\_timer.expires = jiffies + BLINK\_DELAY; 46 add\_timer(&my\_timer);

47 }

48

49 static int \_\_init kbleds\_init(void) 50 {

51 int i;

52

53 pr\_info("kbleds: loading\n"); 54 pr\_info("kbleds: fgconsole is %x\n", fg\_console);

55 for (i = 0; i < MAX\_NR\_CONSOLES; i++) {

56 if (!vc\_cons[i].d)

57 break;

58 pr\_info("poet\_atkm: console[%i/%i] #%i, tty %p\n", i, MAX\_NR\_CONSOLES, 59 vc\_cons[i].d->vc\_num, (void \*)vc\_cons[i].d->port.tty); 60 }

61 pr\_info("kbleds: finished scanning consoles\n"); 62

63 my\_driver = vc\_cons[fg\_console].d->port.tty->driver; 64 pr\_info("kbleds: tty driver name %s\n", my\_driver->driver\_name); 65

66 /\* Set up the LED blink timer the first time. \*/ 67 timer\_setup(&my\_timer, my\_timer\_func, 0); 68 my\_timer.expires = jiffies + BLINK\_DELAY; 69 add\_timer(&my\_timer);

70

71 return 0;

72 }

73

74 static void \_\_exit kbleds\_cleanup(void) 75 {

76 pr\_info("kbleds: unloading...\n");

77 del\_timer(&my\_timer);

78 (my\_driver->ops->ioctl)(vc\_cons[fg\_console].d->port.tty, KDSETLED, 79 RESTORE\_LEDS); 80 }

81

82 module\_init(kbleds\_init);

83 module\_exit(kbleds\_cleanup);

84

85 MODULE\_LICENSE("GPL");

If none of the examples in this chapter fit your debugging needs, there might

yet be some other tricks to try. Ever wondered what CONFIG\_LL\_DEBUG in make menuconfig is good for? If you activate that you get low level access to the serial port. While this might not sound very powerful by itself, you can

patch [kernel/printk.c](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/printk.c) or any other essential syscall to print ASCII characters,

thus making it possible to trace virtually everything what your code does over a serial line. If you find yourself porting the kernel to some new and former unsupported architecture, this is usually amongst the first things that should be implemented. Logging over a netconsole might also be worth a try.

While you have seen lots of stuff that can be used to aid debugging here,

there are some things to be aware of. Debugging is almost always intrusive. Adding debug code can change the situation enough to make the bug seem to disappear. Thus, you should keep debug code to a minimum and make sure it does not show up in production code.

14 GPIO

14.1 GPIO

General Purpose Input/Output (GPIO) appears on the development board as pins. It acts as a bridge for communication between the development board and external devices. You can think of it like a switch: users can turn it on or off (Input), and the development board can also turn it on or off (Output).

To implement a GPIO device driver, you use the gpio\_request() function

to enable a specific GPIO pin. After successfully enabling it, you can check that the pin is being used by looking at /sys/kernel/debug/gpio.

1 cat /sys/kernel/debug/gpio

There are other ways to register GPIOs. For example, you can use gpio\_request\_one()

to register a GPIO while setting its direction (input or output) and initial state at the same time. You can also use gpio\_request\_array() to register multiple GPIOs at once. However, note that gpio\_request\_array() has been removed since Linux v6.10.

When using GPIO, you must set it as either output with gpio\_direction\_output()

or input with gpio\_direction\_input().

• when the GPIO is set as output, you can use gpio\_set\_value() to choose

to set it to high voltage or low voltage.

• when the GPIO is set as input, you can use gpio\_get\_value() to read

whether the voltage is high or low.

14.2 Control the LED’s on/off state

In Section [9,](#Set_the_value_of_myvariable_and) we learned how to communicate with device files. Therefore, we will further use device files to control the LED on and off.

In the implementation, a pull-down resistor is used. The anode of the LED

is connected to GPIO4, and the cathode is connected to GND. For more details

about the Raspberry Pi pin assignments, refer to [Raspberry Pi Pinout.](https://pinout.xyz/) The

materials used include a Raspberry Pi 5, an LED, jumper wires, and a 220Ω resistor.

1 /\*

2 \* led.c - Using GPIO to control the LED on/off 3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11 #include

12 #include

13 #include

14 #include

15 #include

16

17 #include

18

19 #define DEVICE\_NAME "gpio\_led"

20 #define DEVICE\_CNT 1

21 #define BUF\_LEN 2

22

23 static char control\_signal[BUF\_LEN]; 24 static unsigned long device\_buffer\_size = 0; 25

26 struct LED\_dev {

27 dev\_t dev\_num;

28 int major\_num, minor\_num;

29 struct cdev cdev;

30 struct class \*cls;

31 struct device \*dev;

32 };

33

34 static struct LED\_dev led\_device; 35

36 /\* Define GPIOs for LEDs.

37 \* TODO: According to the requirements, search /sys/kernel/debug/gpio to 38 \* find the corresponding GPIO location. 39 \*/

40 static struct gpio leds[] = { { 4, GPIOF\_OUT\_INIT\_LOW, "LED 1" } }; 41

42 /\* This is called whenever a process attempts to open the device file \*/ 43 static int device\_open(struct inode \*inode, struct file \*file) 44 {

45 return 0;

46 }

47

48 static int device\_release(struct inode \*inode, struct file \*file) 49 {

50 return 0;

51 }

52

53 static ssize\_t device\_write(struct file \*file, const char \_\_user \*buffer, 54 size\_t length, loff\_t \*offset)

55 {

56 device\_buffer\_size = min(BUF\_LEN, length); 57

58 if (copy\_from\_user(control\_signal, buffer, device\_buffer\_size)) { 59 return-EFAULT;

60 }

61

62 /\* Determine the received signal to decide the LED on/off state. \*/ 63 switch (control\_signal[0]) {

64 case '0':

65 gpio\_set\_value(leds[0].gpio, 0);

66 pr\_info("LED OFF");

67 break;

68 case '1':

69 gpio\_set\_value(leds[0].gpio, 1);

70 pr\_info("LED ON");

71 break;

72 default:

73 pr\_warn("Invalid value!\n");

74 break;

75 }

76

77 \*offset += device\_buffer\_size;

78

79 /\* Again, return the number of input characters used. \*/ 80 return device\_buffer\_size;

81 }

82

83 static struct file\_operations fops = { 84 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 85 .owner = THIS\_MODULE,

86 #endif

87 .write = device\_write,

88 .open = device\_open,

89 .release = device\_release,

90 };

91

92 /\* Initialize the module - Register the character device \*/ 93 static int \_\_init led\_init(void)

94 {

95 int ret = 0;

96

97 /\* Determine whether dynamic allocation of the device number is needed. \*/

98 if (led\_device.major\_num) {

99 led\_device.dev\_num = MKDEV(led\_device.major\_num,

,→ led\_device.minor\_num);

100 ret =

101 register\_chrdev\_region(led\_device.dev\_num, DEVICE\_CNT,

, → DEVICE\_NAME);

102 } else {

103 ret = alloc\_chrdev\_region(&led\_device.dev\_num, 0, DEVICE\_CNT, 104 DEVICE\_NAME); 105 }

106

107 /\* Negative values signify an error \*/ 108 if (ret < 0) {

109 pr\_alert("Failed to register character device, error: %d\n", ret); 110 return ret;

111 }

112

113 pr\_info("Major = %d, Minor = %d\n", MAJOR(led\_device.dev\_num), 114 MINOR(led\_device.dev\_num)); 115

116 /\* Prevents module unloading while operations are in use \*/ 117 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 118 led\_device.cdev.owner = THIS\_MODULE; 119 #endif

120

121 cdev\_init(&led\_device.cdev, &fops); 122 ret = cdev\_add(&led\_device.cdev, led\_device.dev\_num, 1); 123 if (ret) {

124 pr\_err("Failed to add the device to the system\n"); 125 goto fail1;

126 }

127

128 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 129 led\_device.cls = class\_create(DEVICE\_NAME); 130 #else

131 led\_device.cls = class\_create(THIS\_MODULE, DEVICE\_NAME); 132 #endif

133 if (IS\_ERR(led\_device.cls)) { 134 pr\_err("Failed to create class for device\n"); 135 ret = PTR\_ERR(led\_device.cls); 136 goto fail2;

137 }

138

139 led\_device.dev = device\_create(led\_device.cls, NULL, led\_device.dev\_num,

140 NULL, DEVICE\_NAME); 141 if (IS\_ERR(led\_device.dev)) { 142 pr\_err("Failed to create the device file\n"); 143 ret = PTR\_ERR(led\_device.dev); 144 goto fail3;

145 }

146

147 pr\_info("Device created on /dev/%s\n", DEVICE\_NAME); 148

149 ret = gpio\_request(leds[0].gpio, leds[0].label); 150

151 if (ret) {

152 pr\_err("Unable to request GPIOs for LEDs: %d\n", ret); 153 goto fail4;

154 }

155

156 ret = gpio\_direction\_output(leds[0].gpio, leds[0].flags); 157

158 if (ret) {

159 pr\_err("Failed to set GPIO %d direction\n", leds[0].gpio); 160 goto fail5;

161 }

162

163 return 0;

164

165 fail5:

166 gpio\_free(leds[0].gpio);

167

168 fail4:

169 device\_destroy(led\_device.cls, led\_device.dev\_num); 170

171 fail3:

172 class\_destroy(led\_device.cls); 173

174 fail2:

175 cdev\_del(&led\_device.cdev);

176

177 fail1:

178 unregister\_chrdev\_region(led\_device.dev\_num, DEVICE\_CNT);

179

180 return ret;

181 }

182

183 static void \_\_exit led\_exit(void) 184 {

185 gpio\_set\_value(leds[0].gpio, 0); 186 gpio\_free(leds[0].gpio);

187

188 device\_destroy(led\_device.cls, led\_device.dev\_num); 189 class\_destroy(led\_device.cls); 190 cdev\_del(&led\_device.cdev);

191 unregister\_chrdev\_region(led\_device.dev\_num, DEVICE\_CNT);

192 }

193

194 module\_init(led\_init);

195 module\_exit(led\_exit);

196

197 MODULE\_LICENSE("GPL");

Make and install the module:

1 make

2 sudo insmod led.ko

Switch on the LED:

1 echo "1" | sudo tee /dev/gpio\_led

Switch off the LED:

1 echo "0" | sudo tee /dev/gpio\_led

Finally, remove the module:

1 sudo rmmod led

14.3 DHT11 sensor

The DHT11 sensor is a well-known entry-level sensor commonly used to measure humidity and temperature. In this subsection, we will use GPIO to communicate through a single data line. The DHT11 communication protocol can be referred

to in the [datasheet.](https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf?srsltid=AfmBOoppls-QTd864640bVtbK90sWBsFzJ_7SgjOD2EpwuLLGUSTyYnv)

In the implementation, the data pin of the DHT11 sensor is connected to

GPIO4 on the Raspberry Pi. The sensor’s VCC and GND pins are connected to 3.3V and GND, respectively. For more details about the Raspberry Pi pin

assignments, refer to [Raspberry Pi Pinout.](https://pinout.xyz/) The materials used include a Rasp-berry Pi 5, a DHT11 sensor, and jumper wires.

1 /\*

2 \* dht11.c - Using GPIO to read temperature and humidity from DHT11 sensor.

3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11 #include

12 #include

13 #include

14 #include

15 #include

16

17 #include

18

19 #define GPIO\_PIN\_4 575

20 #define DEVICE\_NAME "dht11"

21 #define DEVICE\_CNT 1

22

23 static char msg[64];

24

25 struct dht11\_dev {

26 dev\_t dev\_num;

27 int major\_num, minor\_num;

28 struct cdev cdev;

29 struct class \*cls;

30 struct device \*dev;

31 };

32

33 static struct dht11\_dev dht11\_device; 34

35 /\* Define GPIOs for LEDs.

36 \* TODO: According to the requirements, search /sys/kernel/debug/gpio to 37 \* find the corresponding GPIO location. 38 \*/

39 static struct gpio dht11[] = { { GPIO\_PIN\_4, GPIOF\_OUT\_INIT\_HIGH, "Signal" }

, → };

40

41 static int dht11\_read\_data(void)

42 {

43 int timeout;

44 uint8\_t sensor\_data[5] = { 0 };

45 uint8\_t i, j;

46

47 gpio\_set\_value(dht11[0].gpio, 0);

48 mdelay(20);

49 gpio\_set\_value(dht11[0].gpio, 1);

50 udelay(30);

51 gpio\_direction\_input(dht11[0].gpio); 52 udelay(2);

53

54 timeout = 300;

55 while (gpio\_get\_value(dht11[0].gpio) && timeout--) 56 udelay(1);

57

58 if (timeout == -1)

59 return-ETIMEDOUT;

60

61 timeout = 300;

62 while (!gpio\_get\_value(dht11[0].gpio) && timeout--) 63 udelay(1);

64

65 if (timeout == -1)

66 return-ETIMEDOUT;

67

68 timeout = 300;

69 while (gpio\_get\_value(dht11[0].gpio) && timeout--) 70 udelay(1);

71

72 if (timeout == -1)

73 return-ETIMEDOUT;

74

75 for (j = 0; j < 5; j++) {

76 uint8\_t byte = 0;

77 for (i = 0; i < 8; i++) {

78 timeout = 300;

79 while (gpio\_get\_value(dht11[0].gpio) && timeout--)

80 udelay(1);

81

82 if (timeout == -1)

83 return-ETIMEDOUT;

84

85 timeout = 300;

86 while (!gpio\_get\_value(dht11[0].gpio) && timeout--)

87 udelay(1);

88

89 if (timeout == -1)

90 return-ETIMEDOUT;

91

92 udelay(50);

93 byte <<= 1;

94 if (gpio\_get\_value(dht11[0].gpio))

95 byte |= 0x01;

96 }

97 sensor\_data[j] = byte;

98 }

99

100 if (sensor\_data[4] != (uint8\_t)(sensor\_data[0] + sensor\_data[1] + 101 sensor\_data[2] + sensor\_data[3])) 102 return-EIO;

103

104 gpio\_direction\_output(dht11[0].gpio, 1); 105 sprintf(msg, "Humidity: %d%%\nTemperature: %d deg C\n", sensor\_data[0], 106 sensor\_data[2]);

107

108 return 0;

109 }

110

111 static int device\_open(struct inode \*inode, struct file \*file) 112 {

113 int ret, retry;

114

115 for (retry = 0; retry < 5; ++retry) { 116 ret = dht11\_read\_data();

117 if (ret == 0)

118 return 0;

119 msleep(10);

120 }

121 gpio\_direction\_output(dht11[0].gpio, 1); 122

123 return ret;

124 }

125

126 static int device\_release(struct inode \*inode, struct file \*file) 127 {

128 return 0;

129 }

130

131 static ssize\_t device\_read(struct file \*filp, char \_\_user \*buffer, 132 size\_t length, loff\_t \*offset) 133 {

134 int msg\_len = strlen(msg);

135

136 if (\*offset >= msg\_len)

137 return 0;

138

139 size\_t remain = msg\_len - \*offset; 140 size\_t bytes\_read = min(length, remain); 141

142 if (copy\_to\_user(buffer, msg + \*offset, bytes\_read)) 143 return-EFAULT;

144

145 \*offset += bytes\_read;

146

147 return bytes\_read;

148 }

149

150 static struct file\_operations fops = { 151 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 152 .owner = THIS\_MODULE,

153 #endif

154 .open = device\_open,

155 .release = device\_release,

156 .read = device\_read 157 };

158

159 /\* Initialize the module - Register the character device \*/ 160 static int \_\_init dht11\_init(void) 161 {

162 int ret = 0;

163

164 /\* Determine whether dynamic allocation of the device number is needed. \*/

165 if (dht11\_device.major\_num) { 166 dht11\_device.dev\_num =

167 MKDEV(dht11\_device.major\_num, dht11\_device.minor\_num); 168 ret = register\_chrdev\_region(dht11\_device.dev\_num, DEVICE\_CNT, 169 DEVICE\_NAME); 170 } else {

171 ret = alloc\_chrdev\_region(&dht11\_device.dev\_num, 0, DEVICE\_CNT, 172 DEVICE\_NAME); 173 }

174

175 /\* Negative values signify an error \*/ 176 if (ret < 0) {

177 pr\_alert("Failed to register character device, error: %d\n", ret); 178 return ret;

179 }

180

181 pr\_info("Major = %d, Minor = %d\n", MAJOR(dht11\_device.dev\_num), 182 MINOR(dht11\_device.dev\_num)); 183

184 /\* Prevents module unloading while operations are in use \*/ 185 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 186 dht11\_device.cdev.owner = THIS\_MODULE; 187 #endif

188

189 cdev\_init(&dht11\_device.cdev, &fops); 190 ret = cdev\_add(&dht11\_device.cdev, dht11\_device.dev\_num, 1); 191 if (ret) {

192 pr\_err("Failed to add the device to the system\n"); 193 goto fail1;

194 }

195

196 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 197 dht11\_device.cls = class\_create(DEVICE\_NAME); 198 #else

199 dht11\_device.cls = class\_create(THIS\_MODULE, DEVICE\_NAME); 200 #endif

201 if (IS\_ERR(dht11\_device.cls)) { 202 pr\_err("Failed to create class for device\n"); 203 ret = PTR\_ERR(dht11\_device.cls); 204 goto fail2;

205 }

206

207 dht11\_device.dev = device\_create(dht11\_device.cls, NULL, 208 dht11\_device.dev\_num, NULL, DEVICE\_NAME);

209 if (IS\_ERR(dht11\_device.dev)) { 210 pr\_err("Failed to create the device file\n"); 211 ret = PTR\_ERR(dht11\_device.dev); 212 goto fail3;

213 }

214

215 pr\_info("Device created on /dev/%s\n", DEVICE\_NAME); 216

217 ret = gpio\_request(dht11[0].gpio, dht11[0].label); 218

219 if (ret) {

220 pr\_err("Unable to request GPIOs for dht11: %d\n", ret); 221 goto fail4;

222 }

223

224 ret = gpio\_direction\_output(dht11[0].gpio, 1); 225

226 if (ret) {

227 pr\_err("Failed to set GPIO %d direction\n", dht11[0].gpio); 228 goto fail5;

229 }

230

231 return 0;

232

233 fail5:

234 gpio\_free(dht11[0].gpio);

235

236 fail4:

237 device\_destroy(dht11\_device.cls, dht11\_device.dev\_num); 238

239 fail3:

240 class\_destroy(dht11\_device.cls); 241

242 fail2:

243 cdev\_del(&dht11\_device.cdev); 244

245 fail1:

246 unregister\_chrdev\_region(dht11\_device.dev\_num, DEVICE\_CNT);

247

248 return ret;

249 }

250

251 static void \_\_exit dht11\_exit(void) 252 {

253 gpio\_set\_value(dht11[0].gpio, 0); 254 gpio\_free(dht11[0].gpio);

255

256 device\_destroy(dht11\_device.cls, dht11\_device.dev\_num); 257 class\_destroy(dht11\_device.cls); 258 cdev\_del(&dht11\_device.cdev); 259 unregister\_chrdev\_region(dht11\_device.dev\_num, DEVICE\_CNT);

260 }

261

262 module\_init(dht11\_init);

263 module\_exit(dht11\_exit);

264

265 MODULE\_LICENSE("GPL");

Make and install the module:

1 make

2 sudo insmod dht11.ko

Check the Output of the DHT11 Sensor:

1 sudo cat /dev/dht11

Expected Output:

$ sudo cat /dev/dht11

Humidity: 61%

Temperature: 30°C

Finally, remove the module:

1 sudo rmmod dht11

15 Scheduling Tasks

There are two main ways of running tasks: tasklets and work queues. Tasklets are a quick and easy way of scheduling a single function to be run. For example, when triggered from an interrupt, whereas work queues are more complicated but also better suited to running multiple things in a sequence.

It is possible that in future tasklets may be replaced by threaded IRQs. How-

ever, discussion about that has been ongoing since 2007 [(Eliminating tasklets](https://lwn.net/Articles/239633)

and [The end of tasklets),](https://lwn.net/Articles/960041/) so expecting immediate changes would be unwise. See

the section [16.1](#for_compatibility__For_further_i) for alternatives that avoid the tasklet debate.

15.1 Tasklets

Here is an example tasklet module. The tasklet\_fn function runs for a few sec-onds. In the meantime, execution of the example\_tasklet\_init function may continue to the exit point, depending on whether it is interrupted by softirq.

1 /\*

2 \* example\_tasklet.c

3 \*/

4 #include

5 #include

6 #include

7 #include

8

9 /\* Macro DECLARE\_TASKLET\_OLD exists for compatibility.

10 \* See https://lwn.net/Articles/830964/ 11 \*/

12 #ifndef DECLARE\_TASKLET\_OLD

13 #define DECLARE\_TASKLET\_OLD(arg1, arg2) DECLARE\_TASKLET(arg1, arg2, 0L) 14 #endif

15

16 static void tasklet\_fn(unsigned long data) 17 {

18 pr\_info("Example tasklet starts\n"); 19 mdelay(5000);

20 pr\_info("Example tasklet ends\n");

21 }

22

23 static DECLARE\_TASKLET\_OLD(mytask, tasklet\_fn); 24

25 static int \_\_init example\_tasklet\_init(void) 26 {

27 pr\_info("tasklet example init\n");

28 tasklet\_schedule(&mytask);

29 mdelay(200);

30 pr\_info("Example tasklet init continues...\n"); 31 return 0;

32 }

33

34 static void \_\_exit example\_tasklet\_exit(void) 35 {

36 pr\_info("tasklet example exit\n");

37 tasklet\_kill(&mytask);

38 }

39

40 module\_init(example\_tasklet\_init); 41 module\_exit(example\_tasklet\_exit); 42

43 MODULE\_DESCRIPTION("Tasklet example"); 44 MODULE\_LICENSE("GPL");

So with this example loaded dmesg should show:

tasklet example init

Example tasklet starts

Example tasklet init continues...

Example tasklet ends

Although tasklet is easy to use, it comes with several drawbacks, and developers have been discussing their removal from the Linux kernel. The tasklet callback runs in atomic context, inside a software interrupt, meaning that it cannot sleep or access user-space data, so not all work can be done in a tasklet handler. Also, the kernel only allows one instance of any given tasklet to be running at any given time; multiple different tasklet callbacks can run in parallel.

In recent kernels, tasklets can be replaced by workqueues, timers, or threaded

interrupts. [2](#15_1) While the removal of tasklets remains a longer-term goal, the current kernel contains more than a hundred uses of tasklets. Now developers are proceeding with the API changes and the macro DECLARE\_TASKLET\_OLD exists

2 The goal of threaded interrupts is to push more of the work to separate threads, so that

the minimum needed for acknowledging an interrupt is reduced, and therefore the time spent handling the interrupt (where it can’t handle any other interrupts at the same time) is reduced.

See [https://lwn.net/Articles/302043/.](https://lwn.net/Articles/302043/)

for compatibility. For further information, see [https://lwn.net/Articles/](https://lwn.net/Articles/830964/)

[830964/.](https://lwn.net/Articles/830964/)

15.2 Work queues

To add a task to the scheduler we can use a workqueue. The kernel then uses the Completely Fair Scheduler (CFS) to execute work within the queue.

1 /\*

2 \* sched.c

3 \*/

4 #include

5 #include

6 #include

7

8 static struct workqueue\_struct \*queue = NULL; 9 static struct work\_struct work;

10

11 static void work\_handler(struct work\_struct \*data) 12 {

13 pr\_info("work handler function.\n"); 14 }

15

16 static int \_\_init sched\_init(void) 17 {

18 queue = alloc\_workqueue("HELLOWORLD", WQ\_UNBOUND, 1); 19 if (!queue) {

20 pr\_err("Failed to allocate workqueue\n");

21 return-ENOMEM;

22 }

23 INIT\_WORK(&work, work\_handler);

24 queue\_work(queue, &work);

25 return 0;

26 }

27

28 static void \_\_exit sched\_exit(void) 29 {

30 flush\_workqueue(queue);

31 destroy\_workqueue(queue);

32 }

33

34 module\_init(sched\_init);

35 module\_exit(sched\_exit);

36

37 MODULE\_LICENSE("GPL");

38 MODULE\_DESCRIPTION("Workqueue example");

16 Interrupt Handlers

16.1 Interrupt Handlers

Except for the last chapter, everything we did in the kernel so far we have done as a response to a process asking for it, either by dealing with a special

file, sending an ioctl(), or issuing a system call. But the job of the kernel is not just to respond to process requests. Another job, which is every bit as important, is to speak to the hardware connected to the machine.

There are two types of interaction between the CPU and the rest of the com-

puter’s hardware. The first type is when the CPU gives orders to the hardware, the other is when the hardware needs to tell the CPU something. The second, called interrupts, is much harder to implement because it has to be dealt with when convenient for the hardware, not the CPU. Hardware devices typically have a very small amount of RAM, and if you do not read their information when available, it is lost.

Under Linux, hardware interrupts are called IRQs (Interrupt ReQuests).

There are two types of IRQs, short and long. A short IRQ is one which is expected to take a very short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur (but not interrupts from the same device). If at all possible, it is better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it is doing (unless

it is processing a more important interrupt, in which case it will deal with this one only when the more important one is done), saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state. Linux kernel solves the problem by splitting interrupt handling into two parts. The first part executes right away and masks the interrupt line. Hardware interrupts must be handled quickly, and that is why we need the second part to handle the heavy work deferred from an interrupt handler. Historically, BH (Linux naming for Bottom Halves ) statistically book-keeps the deferred functions. Softirq and its higher level abstraction, Tasklet, replace BH since Linux 2.3.

The way to implement this is to call request\_irq() to get your interrupt

handler called when the relevant IRQ is received.

In practice IRQ handling can be a bit more complex. Hardware is often

designed in a way that chains two interrupt controllers, so that all the IRQs from interrupt controller B are cascaded to a certain IRQ from interrupt controller A. Of course, that requires that the kernel finds out which IRQ it really was afterwards and that adds overhead. Other architectures offer some special, very low overhead, so called "fast IRQ" or FIQs. To take advantage of them requires handlers to be written in assembly language, so they do not really fit into the kernel. They can be made to work similar to the others, but after that procedure, they are no longer any faster than "common" IRQs. SMP enabled kernels running on systems with more than one processor need to solve another truckload of problems. It is not enough to know if a certain IRQs has happened, it’s also important to know what CPU(s) it was for. People still interested in more details, might want to refer to "APIC" now.

This function receives the IRQ number, the name of the function, flags, a

name for /proc/interrupts and a parameter to be passed to the interrupt

handler. Usually there is a certain number of IRQs available. How many IRQs there are is hardware-dependent.

The flags can be used to specify behaviors of the IRQ. For example, use

IRQF\_SHARED to indicate you are willing to share the IRQ with other interrupt handlers (usually because a number of hardware devices sit on the same IRQ); use the IRQF\_ONESHOT to indicate that the IRQ is not reenabled after the handler finished. It should be noted that in some materials, you may encounter another set of IRQ flags named with the SA prefix. For example, the SA\_SHIRQ and the SA\_INTERRUPT. Those are the IRQ flags in the older kernels. They have been removed completely. Today only the IRQF flags are in use. This function will only succeed if there is not already a handler on this IRQ, or if you are both willing to share.

16.2 Detecting button presses

Many popular single board computers, such as Raspberry Pi or Beagleboards, have a bunch of GPIO pins. Attaching buttons to those and then having a button press do something is a classic case in which you might need to use interrupts, so that instead of having the CPU waste time and battery power polling for a change in input state, it is better for the input to trigger the CPU to then run a particular handling function.

Here is an example where buttons are connected to GPIO numbers 17 and

18 and an LED is connected to GPIO 4. You can change those numbers to whatever is appropriate for your board.

1 /\*

2 \* intrpt.c - Handling GPIO with interrupts 3 \*

4 \* Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de) 5 \* from:

6 \* https://github.com/wendlers/rpi-kmod-samples 7 \*

8 \* Press one button to turn on a LED and another to turn it off. 9 \*/

10

11 #include

12 #include

13 #include /\* for ARRAY\_SIZE() \*/ 14 #include

15 #include

16 #include

17

18 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 10, 0) 19 #define NO\_GPIO\_REQUEST\_ARRAY

20 #endif

21

22 static int button\_irqs[] = { -1, -1 }; 23

24 /\* Define GPIOs for LEDs.

25 \* TODO: Change the numbers for the GPIO on your board. 26 \*/

27 static struct gpio leds[] = { { 4, GPIOF\_OUT\_INIT\_LOW, "LED 1" } }; 28

29 /\* Define GPIOs for BUTTONS

30 \* TODO: Change the numbers for the GPIO on your board. 31 \*/

32 static struct gpio buttons[] = { { 17, GPIOF\_IN, "LED 1 ON BUTTON" }, 33 { 18, GPIOF\_IN, "LED 1 OFF BUTTON" } }; 34

35 /\* interrupt function triggered when a button is pressed. \*/ 36 static irqreturn\_t button\_isr(int irq, void \*data) 37 {

38 /\* first button \*/

39 if (irq == button\_irqs[0] && !gpio\_get\_value(leds[0].gpio)) 40 gpio\_set\_value(leds[0].gpio, 1); 41 /\* second button \*/

42 else if (irq == button\_irqs[1] && gpio\_get\_value(leds[0].gpio)) 43 gpio\_set\_value(leds[0].gpio, 0); 44

45 return IRQ\_HANDLED;

46 }

47

48 static int \_\_init intrpt\_init(void) 49 {

50 int ret = 0;

51

52 pr\_info("%s\n", \_\_func\_\_);

53

54 /\* register LED gpios \*/

55 #ifdef NO\_GPIO\_REQUEST\_ARRAY

56 ret = gpio\_request(leds[0].gpio, leds[0].label); 57 #else

58 ret = gpio\_request\_array(leds, ARRAY\_SIZE(leds)); 59 #endif

60

61 if (ret) {

62 pr\_err("Unable to request GPIOs for LEDs: %d\n", ret); 63 return ret;

64 }

65

66 /\* register BUTTON gpios \*/

67 #ifdef NO\_GPIO\_REQUEST\_ARRAY

68 ret = gpio\_request(buttons[0].gpio, buttons[0].label); 69

70 if (ret) {

71 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 72 goto fail1;

73 }

74

75 ret = gpio\_request(buttons[1].gpio, buttons[1].label); 76

77 if (ret) {

78 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 79 goto fail2;

80 }

81 #else

82 ret = gpio\_request\_array(buttons, ARRAY\_SIZE(buttons)); 83

84 if (ret) {

85 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 86 goto fail1;

87 }

88 #endif

89

90 pr\_info("Current button1 value: %d\n", gpio\_get\_value(buttons[0].gpio)); 91

92 ret = gpio\_to\_irq(buttons[0].gpio);

93

94 if (ret < 0) {

95 pr\_err("Unable to request IRQ: %d\n", ret); 96 #ifdef NO\_GPIO\_REQUEST\_ARRAY

97 goto fail3;

98 #else

99 goto fail2;

100 #endif

101 }

102

103 button\_irqs[0] = ret;

104

105 pr\_info("Successfully requested BUTTON1 IRQ # %d\n", button\_irqs[0]); 106

107 ret = request\_irq(button\_irqs[0], button\_isr, 108 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, 109 "gpiomod#button1", NULL); 110

111 if (ret) {

112 pr\_err("Unable to request IRQ: %d\n", ret); 113 #ifdef NO\_GPIO\_REQUEST\_ARRAY

114 goto fail3;

115 #else

116 goto fail2;

117 #endif

118 }

119

120 ret = gpio\_to\_irq(buttons[1].gpio); 121

122 if (ret < 0) {

123 pr\_err("Unable to request IRQ: %d\n", ret); 124 #ifdef NO\_GPIO\_REQUEST\_ARRAY

125 goto fail3;

126 #else

127 goto fail2;

128 #endif

129 }

130

131 button\_irqs[1] = ret;

132

133 pr\_info("Successfully requested BUTTON2 IRQ # %d\n", button\_irqs[1]); 134

135 ret = request\_irq(button\_irqs[1], button\_isr, 136 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, 137 "gpiomod#button2", NULL); 138

139 if (ret) {

140 pr\_err("Unable to request IRQ: %d\n", ret); 141 #ifdef NO\_GPIO\_REQUEST\_ARRAY

142 goto fail4;

143 #else

144 goto fail3;

145 #endif

146 }

147

148 return 0;

149

150 /\* cleanup what has been setup so far \*/ 151 #ifdef NO\_GPIO\_REQUEST\_ARRAY

152 fail4:

153 free\_irq(button\_irqs[0], NULL); 154

155 fail3:

156 gpio\_free(buttons[1].gpio);

157

158 fail2:

159 gpio\_free(buttons[0].gpio);

160

161 fail1:

162 gpio\_free(leds[0].gpio);

163 #else

164 fail3:

165 free\_irq(button\_irqs[0], NULL); 166

167 fail2:

168 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons));

169

170 fail1:

171 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 172 #endif

173

174 return ret;

175 }

176

177 static void \_\_exit intrpt\_exit(void) 178 {

179 pr\_info("%s\n", \_\_func\_\_);

180

181 /\* free irqs \*/

182 free\_irq(button\_irqs[0], NULL); 183 free\_irq(button\_irqs[1], NULL); 184

185 /\* turn all LEDs off \*/

186 #ifdef NO\_GPIO\_REQUEST\_ARRAY

187 gpio\_set\_value(leds[0].gpio, 0); 188 #else

189 int i;

190 for (i = 0; i < ARRAY\_SIZE(leds); i++) 191 gpio\_set\_value(leds[i].gpio, 0); 192 #endif

193

194 /\* unregister \*/

195 #ifdef NO\_GPIO\_REQUEST\_ARRAY

196 gpio\_free(leds[0].gpio);

197 gpio\_free(buttons[0].gpio);

198 gpio\_free(buttons[1].gpio);

199 #else

200 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 201 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons)); 202 #endif

203 }

204

205 module\_init(intrpt\_init);

206 module\_exit(intrpt\_exit);

207

208 MODULE\_LICENSE("GPL");

209 MODULE\_DESCRIPTION("Handle some GPIO interrupts");

16.3 Bottom Half

Suppose you want to do a bunch of stuff inside of an interrupt routine. A common way to avoid blocking the interrupt for a significant duration is to defer the time-consuming part to a workqueue. This pushes the bulk of the work off into the scheduler. This approach helps speed up the interrupt handling process itself, allowing the system to respond to the next hardware interrupt more quickly.

Kernel developers generally discourage using tasklets due to their design

limitations, such as memory management issues and unpredictable latencies. Instead, they recommend more robust mechanisms like workqueues or softirqs. To address tasklet shortcomings, Linux contributors introduced the BH workqueue, activated with the WQ\_BH flag. This workqueue retains critical features, such as execution in atomic (softirq) context on the same CPU and the inability to sleep.

The example below extends the previous code to include an additional task

executed in process context when an interrupt is triggered.

1 /\*

2 \* bottomhalf.c - Top and bottom half interrupt handling 3 \*

4 \* Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de) 5 \* from:

6 \* https://github.com/wendlers/rpi-kmod-samples 7 \*

8 \* Press one button to turn on an LED and another to turn it off 9 \*/

10

11 #include

12 #include

13 #include

14 #include

15 #include

16 #include

17 #include

18 #include

19

20 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 10, 0) 21 #define NO\_GPIO\_REQUEST\_ARRAY

22 #endif

23

24 static int button\_irqs[] = { -1, -1 }; 25

26 /\* Define GPIOs for LEDs.

27 \* TODO: Change the numbers for the GPIO on your board. 28 \*/

29 static struct gpio leds[] = { { 4, GPIOF\_OUT\_INIT\_LOW, "LED 1" } }; 30

31 /\* Define GPIOs for BUTTONS

32 \* TODO: Change the numbers for the GPIO on your board. 33 \*/

34 static struct gpio buttons[] = {

35 { 17, GPIOF\_IN, "LED 1 ON BUTTON" },

36 { 18, GPIOF\_IN, "LED 1 OFF BUTTON" },

37 };

38

39 /\* Workqueue function containing some non-trivial amount of processing \*/

40 static void bottomhalf\_work\_fn(struct work\_struct \*work) 41 {

42 pr\_info("Bottom half workqueue starts\n"); 43 /\* do something which takes a while \*/ 44 msleep(500);

45

46 pr\_info("Bottom half workqueue ends\n"); 47 }

48

49 static DECLARE\_WORK(bottomhalf\_work, bottomhalf\_work\_fn); 50

51 /\* interrupt function triggered when a button is pressed \*/ 52 static irqreturn\_t button\_isr(int irq, void \*data) 53 {

54 /\* Do something quickly right now \*/ 55 if (irq == button\_irqs[0] && !gpio\_get\_value(leds[0].gpio)) 56 gpio\_set\_value(leds[0].gpio, 1); 57 else if (irq == button\_irqs[1] && gpio\_get\_value(leds[0].gpio)) 58 gpio\_set\_value(leds[0].gpio, 0); 59

60 /\* Do the rest at leisure via the scheduler \*/ 61 schedule\_work(&bottomhalf\_work); 62 return IRQ\_HANDLED;

63 }

64

65 static int \_\_init bottomhalf\_init(void) 66 {

67 int ret = 0;

68

69 pr\_info("%s\n", \_\_func\_\_);

70

71 /\* register LED gpios \*/

72 #ifdef NO\_GPIO\_REQUEST\_ARRAY

73 ret = gpio\_request(leds[0].gpio, leds[0].label); 74 #else

75 ret = gpio\_request\_array(leds, ARRAY\_SIZE(leds)); 76 #endif

77

78 if (ret) {

79 pr\_err("Unable to request GPIOs for LEDs: %d\n", ret); 80 return ret;

81 }

82

83 /\* register BUTTON gpios \*/

84 #ifdef NO\_GPIO\_REQUEST\_ARRAY

85 ret = gpio\_request(buttons[0].gpio, buttons[0].label); 86

87 if (ret) {

88 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 89 goto fail1;

90 }

91

92 ret = gpio\_request(buttons[1].gpio, buttons[1].label); 93

94 if (ret) {

95 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 96 goto fail2;

97 }

98 #else

99 ret = gpio\_request\_array(buttons, ARRAY\_SIZE(buttons));

100

101 if (ret) {

102 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 103 goto fail1;

104 }

105 #endif

106

107 pr\_info("Current button1 value: %d\n", gpio\_get\_value(buttons[0].gpio)); 108

109 ret = gpio\_to\_irq(buttons[0].gpio); 110

111 if (ret < 0) {

112 pr\_err("Unable to request IRQ: %d\n", ret); 113 #ifdef NO\_GPIO\_REQUEST\_ARRAY

114 goto fail3;

115 #else

116 goto fail2;

117 #endif

118 }

119

120 button\_irqs[0] = ret;

121

122 pr\_info("Successfully requested BUTTON1 IRQ # %d\n", button\_irqs[0]); 123

124 ret = request\_irq(button\_irqs[0], button\_isr, 125 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, 126 "gpiomod#button1", NULL); 127

128 if (ret) {

129 pr\_err("Unable to request IRQ: %d\n", ret); 130 #ifdef NO\_GPIO\_REQUEST\_ARRAY

131 goto fail3;

132 #else

133 goto fail2;

134 #endif

135 }

136

137 ret = gpio\_to\_irq(buttons[1].gpio); 138

139 if (ret < 0) {

140 pr\_err("Unable to request IRQ: %d\n", ret); 141 #ifdef NO\_GPIO\_REQUEST\_ARRAY

142 goto fail3;

143 #else

144 goto fail2;

145 #endif

146 }

147

148 button\_irqs[1] = ret;

149

150 pr\_info("Successfully requested BUTTON2 IRQ # %d\n", button\_irqs[1]); 151

152 ret = request\_irq(button\_irqs[1], button\_isr, 153 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, 154 "gpiomod#button2", NULL); 155

156 if (ret) {

157 pr\_err("Unable to request IRQ: %d\n", ret); 158 #ifdef NO\_GPIO\_REQUEST\_ARRAY

159 goto fail4;

160 #else

161 goto fail3;

162 #endif

163 }

164

165 return 0;

166

167 /\* cleanup what has been setup so far \*/ 168 #ifdef NO\_GPIO\_REQUEST\_ARRAY

169 fail4:

170 free\_irq(button\_irqs[0], NULL); 171

172 fail3:

173 gpio\_free(buttons[1].gpio);

174

175 fail2:

176 gpio\_free(buttons[0].gpio);

177

178 fail1:

179 gpio\_free(leds[0].gpio);

180 #else

181 fail3:

182 free\_irq(button\_irqs[0], NULL); 183

184 fail2:

185 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons)); 186

187 fail1:

188 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 189 #endif

190

191 return ret;

192 }

193

194 static void \_\_exit bottomhalf\_exit(void) 195 {

196 pr\_info("%s\n", \_\_func\_\_);

197

198 /\* free irqs \*/

199 free\_irq(button\_irqs[0], NULL); 200 free\_irq(button\_irqs[1], NULL); 201

202 /\* turn all LEDs off \*/

203 #ifdef NO\_GPIO\_REQUEST\_ARRAY

204 gpio\_set\_value(leds[0].gpio, 0); 205 #else

206 int i;

207 for (i = 0; i < ARRAY\_SIZE(leds); i++) 208 gpio\_set\_value(leds[i].gpio, 0); 209 #endif

210

211 /\* unregister \*/

212 #ifdef NO\_GPIO\_REQUEST\_ARRAY

213 gpio\_free(leds[0].gpio);

214 gpio\_free(buttons[0].gpio);

215 gpio\_free(buttons[1].gpio);

216 #else

217 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 218 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons)); 219 #endif

220 }

221

222 module\_init(bottomhalf\_init);

223 module\_exit(bottomhalf\_exit);

224

225 MODULE\_LICENSE("GPL");

226 MODULE\_DESCRIPTION("Interrupt with top and bottom half");

16.4 Threaded IRQ

Threaded IRQ is a mechanism to organize both top-half and bottom-half of an IRQ at once. A threaded IRQ splits the one handler in request\_irq() into two: one for the top-half, the other for the bottom-half. The request\_threaded\_irq() is the function for using threaded IRQs. Two handlers are registered at once in the request\_threaded\_irq().

Those two handlers run in different context. The top-half handler runs in in-

terrupt context. It’s the equivalence of the handler passed to the request\_irq(). The bottom-half handler on the other hand runs in its own thread. This thread is created on registration of a threaded IRQ. Its sole purpose is to run this bottom-half handler. This is where a threaded IRQ is “threaded”. If IRQ\_WAKE\_THREAD is returned by the top-half handler, that bottom-half serving thread will wake up. The thread then runs the bottom-half handler.

Here is an example of how to do the same thing as before, with top and

bottom halves, but using threads.

1 /\*

2 \* bh\_thread.c - Top and bottom half interrupt handling 3 \*

4 \* Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de) 5 \* from:

6 \* https://github.com/wendlers/rpi-kmod-samples 7 \*

8 \* Press one button to turn on a LED and another to turn it off 9 \*/

10

11 #include

12 #include

13 #include

14 #include

15 #include

16 #include

17

18 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 10, 0) 19 #define NO\_GPIO\_REQUEST\_ARRAY

20 #endif

21

22 static int button\_irqs[] = { -1, -1 }; 23

24 /\* Define GPIOs for LEDs.

25 \* FIXME: Change the numbers for the GPIO on your board. 26 \*/

27 static struct gpio leds[] = { { 4, GPIOF\_OUT\_INIT\_LOW, "LED 1" } }; 28

29 /\* Define GPIOs for BUTTONS

30 \* FIXME: Change the numbers for the GPIO on your board. 31 \*/

32 static struct gpio buttons[] = {

33 { 17, GPIOF\_IN, "LED 1 ON BUTTON" },

34 { 18, GPIOF\_IN, "LED 1 OFF BUTTON" },

35 };

36

37 /\* This happens immediately, when the IRQ is triggered \*/ 38 static irqreturn\_t button\_top\_half(int irq, void \*ident) 39 {

40 return IRQ\_WAKE\_THREAD;

41 }

42

43 /\* This can happen at leisure, freeing up IRQs for other high priority task \*/

44 static irqreturn\_t button\_bottom\_half(int irq, void \*ident) 45 {

46 pr\_info("Bottom half task starts\n"); 47 mdelay(500); /\* do something which takes a while \*/ 48 pr\_info("Bottom half task ends\n"); 49 return IRQ\_HANDLED;

50 }

51

52 static int \_\_init bottomhalf\_init(void) 53 {

54 int ret = 0;

55

56 pr\_info("%s\n", \_\_func\_\_);

57

58 /\* register LED gpios \*/

59 #ifdef NO\_GPIO\_REQUEST\_ARRAY

60 ret = gpio\_request(leds[0].gpio, leds[0].label); 61 #else

62 ret = gpio\_request\_array(leds, ARRAY\_SIZE(leds)); 63 #endif

64

65 if (ret) {

66 pr\_err("Unable to request GPIOs for LEDs: %d\n", ret); 67 return ret;

68 }

69

70 /\* register BUTTON gpios \*/

71 #ifdef NO\_GPIO\_REQUEST\_ARRAY

72 ret = gpio\_request(buttons[0].gpio, buttons[0].label); 73

74 if (ret) {

75 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 76 goto fail1;

77 }

78

79 ret = gpio\_request(buttons[1].gpio, buttons[1].label); 80

81 if (ret) {

82 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 83 goto fail2;

84 }

85 #else

86 ret = gpio\_request\_array(buttons, ARRAY\_SIZE(buttons)); 87

88 if (ret) {

89 pr\_err("Unable to request GPIOs for BUTTONs: %d\n", ret); 90 goto fail1;

91 }

92 #endif

93

94 pr\_info("Current button1 value: %d\n", gpio\_get\_value(buttons[0].gpio)); 95

96 ret = gpio\_to\_irq(buttons[0].gpio);

97

98 if (ret < 0) {

99 pr\_err("Unable to request IRQ: %d\n", ret); 100 #ifdef NO\_GPIO\_REQUEST\_ARRAY

101 goto fail3;

102 #else

103 goto fail2;

104 #endif

105 }

106

107 button\_irqs[0] = ret;

108

109 pr\_info("Successfully requested BUTTON1 IRQ # %d\n", button\_irqs[0]); 110

111 ret = request\_threaded\_irq(button\_irqs[0], button\_top\_half, 112 button\_bottom\_half, 113 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, 114 "gpiomod#button1", &buttons[0]); 115

116 if (ret) {

117 pr\_err("Unable to request IRQ: %d\n", ret); 118 #ifdef NO\_GPIO\_REQUEST\_ARRAY

119 goto fail3;

120 #else

121 goto fail2;

122 #endif

123 }

124

125 ret = gpio\_to\_irq(buttons[1].gpio); 126

127 if (ret < 0) {

128 pr\_err("Unable to request IRQ: %d\n", ret); 129 #ifdef NO\_GPIO\_REQUEST\_ARRAY

130 goto fail3;

131 #else

132 goto fail2;

133 #endif

134 }

135

136 button\_irqs[1] = ret;

137

138 pr\_info("Successfully requested BUTTON2 IRQ # %d\n", button\_irqs[1]); 139

140 ret = request\_threaded\_irq(button\_irqs[1], button\_top\_half, 141 button\_bottom\_half, 142 IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING,

143 "gpiomod#button2", &buttons[1]); 144

145 if (ret) {

146 pr\_err("Unable to request IRQ: %d\n", ret); 147 #ifdef NO\_GPIO\_REQUEST\_ARRAY

148 goto fail4;

149 #else

150 goto fail3;

151 #endif

152 }

153

154 return 0;

155

156 /\* cleanup what has been setup so far \*/ 157 #ifdef NO\_GPIO\_REQUEST\_ARRAY

158 fail4:

159 free\_irq(button\_irqs[0], &buttons[0]); 160

161 fail3:

162 gpio\_free(buttons[1].gpio);

163

164 fail2:

165 gpio\_free(buttons[0].gpio);

166

167 fail1:

168 gpio\_free(leds[0].gpio);

169 #else

170 fail3:

171 free\_irq(button\_irqs[0], &buttons[0]); 172

173 fail2:

174 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons)); 175

176 fail1:

177 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 178 #endif

179

180 return ret;

181 }

182

183 static void \_\_exit bottomhalf\_exit(void) 184 {

185 pr\_info("%s\n", \_\_func\_\_);

186

187 /\* free irqs \*/

188 free\_irq(button\_irqs[0], &buttons[0]); 189 free\_irq(button\_irqs[1], &buttons[1]); 190

191 /\* turn all LEDs off \*/

192 #ifdef NO\_GPIO\_REQUEST\_ARRAY

193 gpio\_set\_value(leds[0].gpio, 0); 194 #else

195 int i;

196 for (i = 0; i < ARRAY\_SIZE(leds); i++) 197 gpio\_set\_value(leds[i].gpio, 0); 198 #endif

199

200 /\* unregister \*/

201 #ifdef NO\_GPIO\_REQUEST\_ARRAY

202 gpio\_free(leds[0].gpio);

203 gpio\_free(buttons[0].gpio);

204 gpio\_free(buttons[1].gpio);

205 #else

206 gpio\_free\_array(leds, ARRAY\_SIZE(leds)); 207 gpio\_free\_array(buttons, ARRAY\_SIZE(buttons)); 208 #endif

209 }

210

211 module\_init(bottomhalf\_init);

212 module\_exit(bottomhalf\_exit);

213

214 MODULE\_LICENSE("GPL");

215 MODULE\_DESCRIPTION("Interrupt with top and bottom half");

A threaded IRQ is registered using request\_threaded\_irq(). This function

only takes one additional parameter than the request\_irq() – the bottom-half handling function that runs in its own thread. In this example it is the button\_bottom\_half(). Usage of other parameters are the same as request\_irq().

Presence of both handlers is not mandatory. If either of them is not needed,

pass the NULL instead. A NULL top-half handler implies that no action is taken except to wake up the bottom-half serving thread, which runs the bottom-half handler. Similarly, a NULL bottom-half handler effectively acts as if request\_irq() were used. In fact, this is how request\_irq() is implemented.

# Note that passing NULL to both handlers is considered an error and will make

registration fail.

17 Virtual Input Device Driver

The input device driver is a module that provides a way to communicate with the interaction device via the event. For example, the keyboard can send the press or release event to tell the kernel what we want to do. The input device driver will allocate a new input structure with input\_allocate\_device() and sets up input bitfields, device id, version, etc. After that, registers it by calling input\_register\_device().

Here is an example, vinput, It is an API to allow easy development of virtual

input drivers. The driver needs to export a vinput\_device() that contains the virtual device name and vinput\_ops structure that describes:

• the init function: init()

• the input event injection function: send()

• the readback function: read()

Then using vinput\_register\_device() and vinput\_unregister\_device()

will add a new device to the list of support virtual input devices.

1 int init(struct vinput \*);

This function is passed a struct vinput already initialized with an allo-

cated struct input\_dev. The init() function is responsible for initializing the capabilities of the input device and register it.

1 int send(struct vinput \*, char \*, int);

This function will receive a user string to interpret and inject the event using

the input\_report\_XXXX or input\_event call. The string is already copied from user.

1 int read(struct vinput \*, char \*, int);

This function is used for debugging and should fill the buffer parameter with

the last event sent in the virtual input device format. The buffer will then be copied to user.

vinput devices are created and destroyed using sysfs. And, event injection

is done through a /dev node. The device name will be used by the userland to export a new virtual input device.

The class\_attribute structure is similar to other attribute types we talked

about in section [8:](#93_______endif)

1 struct class\_attribute {

2 struct attribute attr;

3 ssize\_t (\*show)(struct class \*class, struct class\_attribute \*attr, 4 char \*buf);

5 ssize\_t (\*store)(struct class \*class, struct class\_attribute \*attr, 6 const char \*buf, size\_t count); 7 };

In vinput.c, the macro CLASS\_ATTR\_WO(export/unexport) defined in [in-](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/device.h)

[clude/linux/device.h](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/device.h) (in this case, device.h is included in [include/linux/input.h)](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/input.h) will generate the class\_attribute structures which are named class\_attr\_export/unexport. Then, put them into vinput\_class\_attrs array and the macro ATTRIBUTE\_GROUPS(vinput\_class) will generate the struct attribute\_group vinput\_class\_group that should be assigned in vinput\_class. Finally, call class\_register(&vinput\_class) to create attributes in sysfs.

To create a vinputX sysfs entry and /dev node.

1 echo "vkbd" | sudo tee /sys/class/vinput/export

To unexport the device, just echo its id in unexport:

1 echo "0" | sudo tee /sys/class/vinput/unexport

1 /\*

2 \* vinput.h

3 \*/

4

5 #ifndef VINPUT\_H

6 #define VINPUT\_H

7

8 #include

9 #include

10

11 #define VINPUT\_MAX\_LEN 128

12 #define MAX\_VINPUT 32

13 #define VINPUT\_MINORS MAX\_VINPUT

14

15 #define dev\_to\_vinput(dev) container\_of(dev, struct vinput, dev) 16

17 struct vinput\_device;

18

19 struct vinput {

20 long id;

21 long devno;

22 long last\_entry;

23 spinlock\_t lock;

24

25 void \*priv\_data;

26

27 struct device dev;

28 struct list\_head list;

29 struct input\_dev \*input;

30 struct vinput\_device \*type;

31 };

32

33 struct vinput\_ops {

34 int (\*init)(struct vinput \*);

35 int (\*kill)(struct vinput \*);

36 int (\*send)(struct vinput \*, char \*, int);

37 int (\*read)(struct vinput \*, char \*, int);

38 };

39

40 struct vinput\_device {

41 char name[16];

42 struct list\_head list;

43 struct vinput\_ops \*ops;

44 };

45

46 int vinput\_register(struct vinput\_device \*dev); 47 void vinput\_unregister(struct vinput\_device \*dev); 48

49 #endif

1 /\*

2 \* vinput.c

3 \*/

4

5 #include

6 #include

7 #include

8 #include

9 #include

10 #include

11

12 #include

13

14 #include "vinput.h"

15

16 #define DRIVER\_NAME "vinput"

17

18 #define dev\_to\_vinput(dev) container\_of(dev, struct vinput, dev) 19

20 static DECLARE\_BITMAP(vinput\_ids, VINPUT\_MINORS); 21

22 static LIST\_HEAD(vinput\_devices); 23 static LIST\_HEAD(vinput\_vdevices); 24

25 static int vinput\_dev;

26 static struct spinlock vinput\_lock; 27 static struct class vinput\_class; 28

29 /\* Search the name of vinput device in the vinput\_devices linked list,

30 \* which added at vinput\_register(). 31 \*/

32 static struct vinput\_device \*vinput\_get\_device\_by\_type(const char \*type) 33 {

34 int found = 0;

35 struct vinput\_device \*vinput;

36 struct list\_head \*curr;

37

38 spin\_lock(&vinput\_lock);

39 list\_for\_each (curr, &vinput\_devices) { 40 vinput = list\_entry(curr, struct vinput\_device, list); 41 if (vinput && strncmp(type, vinput->name, strlen(vinput->name)) == 0)

,→ {

42 found = 1;

43 break;

44 }

45 }

46 spin\_unlock(&vinput\_lock);

47

48 if (found)

49 return vinput;

50 return ERR\_PTR(-ENODEV);

51 }

52

53 /\* Search the id of virtual device in the vinput\_vdevices linked list, 54 \* which added at vinput\_alloc\_vdevice(). 55 \*/

56 static struct vinput \*vinput\_get\_vdevice\_by\_id(long id) 57 {

58 struct vinput \*vinput = NULL;

59 struct list\_head \*curr;

60

61 spin\_lock(&vinput\_lock);

62 list\_for\_each (curr, &vinput\_vdevices) { 63 vinput = list\_entry(curr, struct vinput, list); 64 if (vinput && vinput->id == id) 65 break;

66 }

67 spin\_unlock(&vinput\_lock);

68

69 if (vinput && vinput->id == id)

70 return vinput;

71 return ERR\_PTR(-ENODEV);

72 }

73

74 static int vinput\_open(struct inode \*inode, struct file \*file) 75 {

76 int err = 0;

77 struct vinput \*vinput = NULL;

78

79 vinput = vinput\_get\_vdevice\_by\_id(iminor(inode)); 80

81 if (IS\_ERR(vinput))

82 err = PTR\_ERR(vinput);

83 else

84 file->private\_data = vinput;

85

86 return err;

87 }

88

89 static int vinput\_release(struct inode \*inode, struct file \*file) 145 static void vinput\_unregister\_vdevice(struct vinput \*vinput) 146 {

90 {

91 return 0;

92 }

93

94 static ssize\_t vinput\_read(struct file \*file, char \_\_user \*buffer, size\_t

, → count,

95 loff\_t \*offset)

96 {

97 int len;

98 char buff[VINPUT\_MAX\_LEN + 1]; 99 struct vinput \*vinput = file->private\_data; 100

101 len = vinput->type->ops->read(vinput, buff, count); 102

103 if (\*offset > len)

104 count = 0;

105 else if (count + \*offset > VINPUT\_MAX\_LEN) 106 count = len - \*offset;

107

108 if (raw\_copy\_to\_user(buffer, buff + \*offset, count)) 109 return-EFAULT;

110

111 \*offset += count;

112

113 return count;

114 }

115

116 static ssize\_t vinput\_write(struct file \*file, const char \_\_user \*buffer, 117 size\_t count, loff\_t \*offset) 118 {

119 char buff[VINPUT\_MAX\_LEN + 1]; 120 struct vinput \*vinput = file->private\_data; 121

122 memset(buff, 0, sizeof(char) \* (VINPUT\_MAX\_LEN + 1)); 123

124 if (count > VINPUT\_MAX\_LEN) { 125 dev\_warn(&vinput->dev, "Too long. %d bytes allowed\n",

, → VINPUT\_MAX\_LEN);

126 return-EINVAL;

127 }

128

129 if (raw\_copy\_from\_user(buff, buffer, count)) 130 return-EFAULT;

131

132 return vinput->type->ops->send(vinput, buff, count); 133 }

134

135 static const struct file\_operations vinput\_fops = { 136 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 137 .owner = THIS\_MODULE,

138 #endif

139 .open = vinput\_open,

140 .release = vinput\_release,

141 .read = vinput\_read,

142 .write = vinput\_write,

143 };

144

147 input\_unregister\_device(vinput->input); 148 if (vinput->type->ops->kill)

149 vinput->type->ops->kill(vinput); 150 }

151

152 static void vinput\_destroy\_vdevice(struct vinput \*vinput) 153 {

154 /\* Remove from the list first \*/ 155 spin\_lock(&vinput\_lock);

156 list\_del(&vinput->list);

157 clear\_bit(vinput->id, vinput\_ids); 158 spin\_unlock(&vinput\_lock);

159

160 kfree(vinput);

161 }

162

163 static void vinput\_release\_dev(struct device \*dev) 164 {

165 struct vinput \*vinput = dev\_to\_vinput(dev); 166 int id = vinput->id;

167

168 vinput\_destroy\_vdevice(vinput); 169

170 pr\_debug("released vinput%d.\n", id); 171 }

172

173 static struct vinput \*vinput\_alloc\_vdevice(void) 174 {

175 int err;

176 struct vinput \*vinput = kzalloc(sizeof(struct vinput), GFP\_KERNEL); 177

178 if (!vinput) {

179 pr\_err("vinput: Cannot allocate vinput input device\n"); 180 return ERR\_PTR(-ENOMEM);

181 }

182

183 spin\_lock\_init(&vinput->lock); 184

185 spin\_lock(&vinput\_lock);

186 vinput->id = find\_first\_zero\_bit(vinput\_ids, VINPUT\_MINORS); 187 if (vinput->id >= VINPUT\_MINORS) { 188 err = -ENOBUFS;

189 goto fail\_id;

190 }

191 set\_bit(vinput->id, vinput\_ids); 192 list\_add(&vinput->list, &vinput\_vdevices); 193 spin\_unlock(&vinput\_lock);

194

195 /\* allocate the input device \*/ 196 vinput->input = input\_allocate\_device(); 197 if (vinput->input == NULL) {

198 pr\_err("vinput: Cannot allocate vinput input device\n"); 199 err = -ENOMEM;

200 goto fail\_input\_dev;

201 }

202

203 /\* initialize device \*/

204 vinput->dev.class = &vinput\_class; 205 vinput->dev.release = vinput\_release\_dev; 206 vinput->dev.devt = MKDEV(vinput\_dev, vinput->id); 207 dev\_set\_name(&vinput->dev, DRIVER\_NAME "%lu", vinput->id); 208

209 return vinput;

210

211 fail\_input\_dev:

212 spin\_lock(&vinput\_lock);

213 list\_del(&vinput->list);

214 fail\_id:

215 spin\_unlock(&vinput\_lock);

216 kfree(vinput);

217

218 return ERR\_PTR(err);

219 }

220

221 static int vinput\_register\_vdevice(struct vinput \*vinput) 222 {

223 int err = 0;

224

225 /\* register the input device \*/ 226 vinput->input->name = vinput->type->name; 227 vinput->input->phys = "vinput"; 228 vinput->input->dev.parent = &vinput->dev; 229

230 vinput->input->id.bustype = BUS\_VIRTUAL; 231 vinput->input->id.product = 0x0000; 232 vinput->input->id.vendor = 0x0000; 233 vinput->input->id.version = 0x0000; 234

235 err = vinput->type->ops->init(vinput); 236

237 if (err == 0)

238 dev\_info(&vinput->dev, "Registered virtual input %s %ld\n", 239 vinput->type->name, vinput->id); 240

241 return err;

242 }

243

244 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 245 static ssize\_t export\_store(const struct class \*class, 246 const struct class\_attribute \*attr, 247 #else

248 static ssize\_t export\_store(struct class \*class, struct class\_attribute \*attr, 249 #endif

250 const char \*buf, size\_t len) 251 {

252 int err;

253 struct vinput \*vinput;

254 struct vinput\_device \*device; 255

256 device = vinput\_get\_device\_by\_type(buf); 257 if (IS\_ERR(device)) {

258 pr\_info("vinput: This virtual device isn't registered\n"); 259 err = PTR\_ERR(device);

260 goto fail;

261 }

262

263 vinput = vinput\_alloc\_vdevice(); 264 if (IS\_ERR(vinput)) {

265 err = PTR\_ERR(vinput);

266 goto fail;

267 }

268

269 vinput->type = device;

270 err = device\_register(&vinput->dev); 271 if (err < 0)

272 goto fail\_register;

273

274 err = vinput\_register\_vdevice(vinput); 275 if (err < 0)

276 goto fail\_register\_vinput; 277

278 return len;

279

280 fail\_register\_vinput:

281 input\_free\_device(vinput->input); 282 device\_unregister(&vinput->dev); 283 /\* avoid calling vinput\_destroy\_vdevice() twice \*/ 284 return err;

285 fail\_register:

286 input\_free\_device(vinput->input); 287 vinput\_destroy\_vdevice(vinput); 288 fail:

289 return err;

290 }

291 /\* This macro generates class\_attr\_export structure and export\_store() \*/

292 static CLASS\_ATTR\_WO(export);

293

294 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 295 static ssize\_t unexport\_store(const struct class \*class, 296 const struct class\_attribute \*attr, 297 #else

298 static ssize\_t unexport\_store(struct class \*class, struct class\_attribute

, → \*attr,

299 #endif

300 const char \*buf, size\_t len) 301 {

302 int err;

303 unsigned long id;

304 struct vinput \*vinput;

305

306 err = kstrtol(buf, 10, &id);

307 if (err) {

308 err = -EINVAL;

309 goto failed;

310 }

311

312 vinput = vinput\_get\_vdevice\_by\_id(id); 313 if (IS\_ERR(vinput)) {

314 pr\_err("vinput: No such vinput device %ld\n", id); 315 err = PTR\_ERR(vinput);

316 goto failed;

317 }

318

319 vinput\_unregister\_vdevice(vinput); 320 device\_unregister(&vinput->dev); 321

322 return len;

323 failed:

324 return err;

325 }

326 /\* This macro generates class\_attr\_unexport structure and unexport\_store() \*/

327 static CLASS\_ATTR\_WO(unexport);

328

329 static struct attribute \*vinput\_class\_attrs[] = { 330 &class\_attr\_export.attr,

331 &class\_attr\_unexport.attr,

332 NULL,

333 };

334

335 /\* This macro generates vinput\_class\_groups structure \*/ 336 ATTRIBUTE\_GROUPS(vinput\_class);

337

338 static struct class vinput\_class = { 339 .name = "vinput",

340 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 341 .owner = THIS\_MODULE,

342 #endif

343 .class\_groups = vinput\_class\_groups, 344 };

345

346 int vinput\_register(struct vinput\_device \*dev) 347 {

348 spin\_lock(&vinput\_lock);

349 list\_add(&dev->list, &vinput\_devices); 350 spin\_unlock(&vinput\_lock);

351

352 pr\_info("vinput: registered new virtual input device '%s'\n", dev->name); 353

354 return 0;

355 }

356 EXPORT\_SYMBOL(vinput\_register);

357

358 void vinput\_unregister(struct vinput\_device \*dev) 359 {

360 struct list\_head \*curr, \*next; 361

362 /\* Remove from the list first \*/ 363 spin\_lock(&vinput\_lock);

364 list\_del(&dev->list);

365 spin\_unlock(&vinput\_lock);

366

367 /\* unregister all devices of this type \*/ 368 list\_for\_each\_safe (curr, next, &vinput\_vdevices) { 369 struct vinput \*vinput = list\_entry(curr, struct vinput, list); 370 if (vinput && vinput->type == dev) { 371 vinput\_unregister\_vdevice(vinput);

372 device\_unregister(&vinput->dev); 373 }

374 }

375

376 pr\_info("vinput: unregistered virtual input device '%s'\n", dev->name); 377 }

378 EXPORT\_SYMBOL(vinput\_unregister); 379

380 static int \_\_init vinput\_init(void) 381 {

382 int err = 0;

383

384 pr\_info("vinput: Loading virtual input driver\n"); 385

386 vinput\_dev = register\_chrdev(0, DRIVER\_NAME, &vinput\_fops); 387 if (vinput\_dev < 0) {

388 pr\_err("vinput: Unable to allocate char dev region\n"); 389 err = vinput\_dev;

390 goto failed\_alloc;

391 }

392

393 spin\_lock\_init(&vinput\_lock); 394

395 err = class\_register(&vinput\_class); 396 if (err < 0) {

397 pr\_err("vinput: Unable to register vinput class\n"); 398 goto failed\_class;

399 }

400

401 return 0;

402 failed\_class:

403 unregister\_chrdev(vinput\_dev, DRIVER\_NAME); 404 failed\_alloc:

405 return err;

406 }

407

408 static void \_\_exit vinput\_end(void) 409 {

410 pr\_info("vinput: Unloading virtual input driver\n"); 411

412 unregister\_chrdev(vinput\_dev, DRIVER\_NAME); 413 class\_unregister(&vinput\_class); 414 }

415

416 module\_init(vinput\_init);

417 module\_exit(vinput\_end);

418

419 MODULE\_LICENSE("GPL");

420 MODULE\_DESCRIPTION("Emulate input events");

Here the virtual keyboard is one of example to use vinput. It supports

all KEY\_MAX keycodes. The injection format is the KEY\_CODE such as defined

in [include/linux/input.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/input.h) A positive value means KEY\_PRESS while a negative value is a KEY\_RELEASE. The keyboard supports repetition when the key stays pressed for too long. The following demonstrates how simulation work.

Simulate a key press on "g" (KEY\_G = 34):

1 echo "+34" | sudo tee /dev/vinput0

Simulate a key release on "g" (KEY\_G = 34):

1 echo "-34" | sudo tee /dev/vinput0

1 /\*

2 \* vkbd.c

3 \*/

4

5 #include

6 #include

7 #include

8 #include

9

10 #include "vinput.h"

11

12 #define VINPUT\_KBD "vkbd"

13 #define VINPUT\_RELEASE 0

14 #define VINPUT\_PRESS 1

15

16 static unsigned short vkeymap[KEY\_MAX]; 17

18 static int vinput\_vkbd\_init(struct vinput \*vinput) 19 {

20 int i;

21

22 /\* Set up the input bitfield \*/ 23 vinput->input->evbit[0] = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_REP); 24 vinput->input->keycodesize = sizeof(unsigned short);

25 vinput->input->keycodemax = KEY\_MAX; 26 vinput->input->keycode = vkeymap; 27

28 for (i = 0; i < KEY\_MAX; i++)

29 set\_bit(vkeymap[i], vinput->input->keybit);

30

31 /\* vinput will help us allocate new input device structure via 32 \* input\_allocate\_device(). So, we can register it straightforwardly.

33 \*/

34 return input\_register\_device(vinput->input); 35 }

36

37 static int vinput\_vkbd\_read(struct vinput \*vinput, char \*buff, int len) 38 {

39 spin\_lock(&vinput->lock);

40 len = snprintf(buff, len, "%+ld\n", vinput->last\_entry); 41 spin\_unlock(&vinput->lock);

42

43 return len;

44 }

45

46 static int vinput\_vkbd\_send(struct vinput \*vinput, char \*buff, int len) 47 {

48 int ret;

49 long key = 0;

50 short type = VINPUT\_PRESS;

51

52 /\* Determine which event was received (press or release) 53 \* and store the state.

54 \*/

55 if (buff[0] == '+')

56 ret = kstrtol(buff + 1, 10, &key);

57 else

58 ret = kstrtol(buff, 10, &key);

59 if (ret)

60 dev\_err(&vinput->dev, "error during kstrtol: -%d\n", ret); 61 spin\_lock(&vinput->lock);

62 vinput->last\_entry = key;

63 spin\_unlock(&vinput->lock);

64

65 if (key < 0) {

66 type = VINPUT\_RELEASE;

67 key = -key;

68 }

69

70 dev\_info(&vinput->dev, "Event %s code %ld\n", 71 (type == VINPUT\_RELEASE) ? "VINPUT\_RELEASE" : "VINPUT\_PRESS",

, → key);

72

73 /\* Report the state received to input subsystem. \*/ 74 input\_report\_key(vinput->input, key, type); 75 /\* Tell input subsystem that it finished the report. \*/ 76 input\_sync(vinput->input);

77

78 return len;

79 }

80

81 static struct vinput\_ops vkbd\_ops = { 82 .init = vinput\_vkbd\_init,

83 .send = vinput\_vkbd\_send,

84 .read = vinput\_vkbd\_read,

85 };

86

87 static struct vinput\_device vkbd\_dev = { 88 .name = VINPUT\_KBD,

89 .ops = &vkbd\_ops,

90 };

91

92 static int \_\_init vkbd\_init(void) 93 {

94 int i;

95

96 for (i = 0; i < KEY\_MAX; i++)

97 vkeymap[i] = i;

98 return vinput\_register(&vkbd\_dev);

99 }

100

101 static void \_\_exit vkbd\_end(void)

102 {

103 vinput\_unregister(&vkbd\_dev); 104 }

105

106 module\_init(vkbd\_init);

107 module\_exit(vkbd\_end);

108

109 MODULE\_LICENSE("GPL");

110 MODULE\_DESCRIPTION("Emulate keyboard input events through /dev/vinput");

18 Standardizing the interfaces: The Device Model

Up to this point we have seen all kinds of modules doing all kinds of things, but there was no consistency in their interfaces with the rest of the kernel. To impose some consistency such that there is at minimum a standardized way to start, suspend and resume a device model was added. An example is shown below, and you can use this as a template to add your own suspend, resume or other interface functions.

1 /\*

2 \* devicemodel.c

3 \*/

4 #include

5 #include

6 #include 7 #include

8

9 struct devicemodel\_data {

10 char \*greeting;

11 int number;

12 };

13

14 static int devicemodel\_probe(struct platform\_device \*dev) 15 {

16 struct devicemodel\_data \*pd =

17 (struct devicemodel\_data \*)(dev->dev.platform\_data);

18

19 pr\_info("devicemodel probe\n"); 20 pr\_info("devicemodel greeting: %s; %d\n", pd->greeting, pd->number);

21

22 /\* Your device initialization code \*/ 23

24 return 0;

25 }

26

27 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 11, 0) 28 static void devicemodel\_remove(struct platform\_device \*dev) 29 {

30 pr\_info("devicemodel example removed\n"); 31 /\* Your device removal code \*/ 32 }

33 #else

34 static int devicemodel\_remove(struct platform\_device \*dev)

35 {

36 pr\_info("devicemodel example removed\n"); 37 /\* Your device removal code \*/ 38 return 0;

39 }

40 #endif

41

42 static int devicemodel\_suspend(struct device \*dev) 43 {

44 pr\_info("devicemodel example suspend\n"); 45

46 /\* Your device suspend code \*/

47

48 return 0;

49 }

50

51 static int devicemodel\_resume(struct device \*dev) 52 {

53 pr\_info("devicemodel example resume\n"); 54

55 /\* Your device resume code \*/

56

57 return 0;

58 }

59

60 static const struct dev\_pm\_ops devicemodel\_pm\_ops = { 61 .suspend = devicemodel\_suspend, 62 .resume = devicemodel\_resume, 63 .poweroff = devicemodel\_suspend, 64 .freeze = devicemodel\_suspend, 65 .thaw = devicemodel\_resume,

66 .restore = devicemodel\_resume,

67 };

68

69 static struct platform\_driver devicemodel\_driver = { 70 .driver =

71 {

72 .name = "devicemodel\_example",

73 .pm = &devicemodel\_pm\_ops,

74 },

75 .probe = devicemodel\_probe,

76 .remove = devicemodel\_remove,

77 };

78

79 static int \_\_init devicemodel\_init(void) 80 {

81 int ret;

82

83 pr\_info("devicemodel init\n");

84

85 ret = platform\_driver\_register(&devicemodel\_driver);

86

87 if (ret) {

88 pr\_err("Unable to register driver\n");

89 return ret;

90 }

91

92 return 0;

93 }

94

95 static void \_\_exit devicemodel\_exit(void) 96 {

97 pr\_info("devicemodel exit\n"); 98 platform\_driver\_unregister(&devicemodel\_driver);

99 }

100

101 module\_init(devicemodel\_init);

102 module\_exit(devicemodel\_exit);

103

104 MODULE\_LICENSE("GPL");

105 MODULE\_DESCRIPTION("Linux Device Model example");

19 Device Tree

19.1 Introduction to Device Tree

Device Tree is a data structure that describes hardware components in a system, particularly in embedded systems and ARM-based platforms. Instead of hard-coding hardware details in the kernel source, Device Tree provides a separate, human-readable description that the kernel can parse at boot time. This sep-aration allows the same kernel binary to support multiple hardware platforms, making development and maintenance significantly easier.

Device Tree files (with .dts extension for source files and .dtb for compiled

binary files) use a hierarchical structure similar to a filesystem to represent the hardware topology. Each hardware component is represented as a node with properties that describe its characteristics, such as memory addresses, interrupt numbers, and device-specific parameters.

19.2 Device Tree and Kernel Modules

While Device Tree is primarily used during kernel initialization, kernel modules can also interact with Device Tree nodes through the platform device framework. When the kernel parses the Device Tree at boot, it creates platform devices for nodes that have compatible strings. Kernel modules can then register platform drivers that match these compatible strings, allowing them to be automatically probed when the corresponding hardware is detected.

The key concepts for Device Tree interaction in kernel modules include:

• Compatible strings: Unique identifiers that match Device Tree nodes

to their drivers

• Property reading: Functions to extract configuration data from Device

Tree nodes

• Platform driver framework: Infrastructure for binding drivers to de-

vices described in Device Tree

• Device-specific data: Custom properties that can be defined for specific

hardware

19.3 Example: Device Tree Module

The following example demonstrates how a kernel module can interact with Device Tree nodes. This module registers a platform driver that matches specific compatible strings and extracts properties from the matched Device Tree nodes.

1 /\* devicetree.c - Demonstrates device tree interaction with kernel modules \*/

2

3 #include

4 #include

5 #include

6 #include

7 #include

8 #include 9 #include

10

11 #define DRIVER\_NAME "lkmpg\_devicetree" 12

13 /\* Structure to hold device-specific data \*/ 14 struct dt\_device\_data {

15 const char \*label;

16 u32 reg\_value;

17 u32 custom\_value;

18 bool has\_clock;

19 };

20

21 /\* Probe function - called when device tree node matches \*/ 22 static int dt\_probe(struct platform\_device \*pdev) 23 {

24 struct device \*dev = &pdev->dev;

25 struct device\_node \*np = dev->of\_node;

26 struct dt\_device\_data \*data;

27 const char \*string\_prop;

28 u32 value;

29 int ret;

30

31 pr\_info("%s: Device tree probe called for %s\n", DRIVER\_NAME, 32 np->full\_name);

33

34 /\* Allocate memory for device data \*/ 35 data = devm\_kzalloc(dev, sizeof(\*data), GFP\_KERNEL); 36 if (!data)

37 return-ENOMEM;

38

39 /\* Read a string property \*/

40 ret = of\_property\_read\_string(np, "label", &string\_prop); 41 if (ret == 0) {

42 data->label = string\_prop; 43 pr\_info("%s: Found label property: %s\n", DRIVER\_NAME, data->label);

44 } else {

45 data->label = "unnamed";

46 pr\_info("%s: No label property found, using default\n", DRIVER\_NAME);

47 }

48

49 /\* Read a u32 property \*/

50 ret = of\_property\_read\_u32(np, "reg", &value); 51 if (ret == 0) {

52 data->reg\_value = value;

53 pr\_info("%s: Found reg property: 0x%x\n", DRIVER\_NAME,

,→ data->reg\_value);

54 }

55

56 /\* Read a custom u32 property \*/ 57 ret = of\_property\_read\_u32(np, "lkmpg,custom-value", &value);

58 if (ret == 0) {

59 data->custom\_value = value; 60 pr\_info("%s: Found custom-value property: %u\n", DRIVER\_NAME,

61 data->custom\_value);

62 } else {

63 data->custom\_value = 42; /\* Default value \*/ 64 pr\_info("%s: No custom-value found, using default: %u\n", DRIVER\_NAME,

65 data->custom\_value);

66 }

67

68 /\* Check for presence of a property \*/ 69 data->has\_clock = of\_property\_read\_bool(np, "lkmpg,has-clock"); 70 pr\_info("%s: has-clock property: %s\n", DRIVER\_NAME, 71 data->has\_clock ? "present" : "absent"); 72

73 /\* Store device data for later use \*/ 74 platform\_set\_drvdata(pdev, data); 75

76 pr\_info("%s: Device probe successful\n", DRIVER\_NAME); 77 return 0;

78 }

79

80 /\* Remove function - called when device is removed \*/ 81 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 11, 0) 82 static void dt\_remove(struct platform\_device \*pdev) 83 {

84 struct dt\_device\_data \*data = platform\_get\_drvdata(pdev); 85

86 pr\_info("%s: Removing device %s\n", DRIVER\_NAME, data->label); 87 /\* Cleanup is handled automatically by devm\_\* functions \*/ 88 }

89 #else

90 static int dt\_remove(struct platform\_device \*pdev) 91 {

92 struct dt\_device\_data \*data = platform\_get\_drvdata(pdev); 93

94 pr\_info("%s: Removing device %s\n", DRIVER\_NAME, data->label); 95 /\* Cleanup is handled automatically by devm\_\* functions \*/ 96 return 0;

97 }

98 #endif

99

100 /\* Device tree match table - defines compatible strings this driver supports

, → \*/

101 static const struct of\_device\_id dt\_match\_table[] = { 102 {

103 .compatible = "lkmpg,example-device", 104 },

105 {

106 .compatible = "lkmpg,another-device", 107 },

108 {} /\* Sentinel \*/

109 };

110 MODULE\_DEVICE\_TABLE(of, dt\_match\_table); 111

112 /\* Platform driver structure \*/

113 static struct platform\_driver dt\_driver = { 114 .probe = dt\_probe,

115 .remove = dt\_remove,

116 .driver = {

117 .name = DRIVER\_NAME,

118 .of\_match\_table = dt\_match\_table, 119 },

120 };

121

122 /\* Module initialization \*/

123 static int \_\_init dt\_init(void)

124 {

125 int ret;

126

127 pr\_info("%s: Initializing device tree example module\n", DRIVER\_NAME); 128

129 /\* Register the platform driver \*/ 130 ret = platform\_driver\_register(&dt\_driver); 131 if (ret) {

132 pr\_err("%s: Failed to register platform driver\n", DRIVER\_NAME); 133 return ret;

134 }

135

136 pr\_info("%s: Module loaded successfully\n", DRIVER\_NAME); 137 return 0;

138 }

139

140 /\* Module cleanup \*/

141 static void \_\_exit dt\_exit(void)

142 {

143 pr\_info("%s: Cleaning up device tree example module\n", DRIVER\_NAME); 144 platform\_driver\_unregister(&dt\_driver); 145 }

146

147 module\_init(dt\_init);

148 module\_exit(dt\_exit);

149

150 MODULE\_LICENSE("GPL");

151 MODULE\_DESCRIPTION("Device tree interaction example for LKMPG");

19.4 Device Tree Source Example

To use the above module, you would need a Device Tree entry like this:

1 /\* Example device tree fragment \*/ 2 lkmpg\_device@0 {

3 compatible = "lkmpg,example-device"; 4 reg = <0x40000000 0x1000>;

5 label = "LKMPG Test Device";

6 lkmpg,custom-value = <100>;

7 lkmpg,has-clock;

8 };

The properties in this Device Tree node would be read by the module’s

probe function when the device is matched. The compatible property is used to match the device with the driver, while other properties provide device-specific configuration.

19.5 Testing Device Tree Modules

Testing Device Tree modules can be done in several ways:

1. Using Device Tree overlays: On systems that support it (like Rasp-

berry Pi), you can load Device Tree overlays at runtime to add new devices without rebooting.

2. Modifying the main Device Tree: Add your device nodes to the sys-

tem’s main Device Tree source file and recompile it.

3. Using QEMU: For development and testing, QEMU can emulate systems

with custom Device Trees, allowing you to test your modules without physical hardware.

To check if your device was properly detected, you can examine the sysfs

filesystem:

1 # List all platform devices 2 ls /sys/bus/platform/devices/ 3

4 # Check device tree nodes

5 ls /proc/device-tree/

19.6 Common Device Tree Functions

Here are some commonly used Device Tree functions in kernel modules:

• of\_property\_read\_string()- Read a string property

• of\_property\_read\_u32()- Read a 32-bit integer property

• of\_property\_read\_bool()- Check if a boolean property exists

• of\_find\_property()- Find a property by name

• of\_get\_property()- Get a property’s raw value

• of\_match\_device()- Match a device against a match table

• of\_parse\_phandle()- Parse a phandle reference to another node

These functions provide a robust interface for extracting configuration data

from Device Tree nodes, allowing modules to be highly configurable without code changes.

20 Optimizations

20.1 Likely and Unlikely conditions

Sometimes you might want your code to run as quickly as possible, especially if it is handling an interrupt or doing something which might cause noticeable latency. If your code contains boolean conditions and if you know that the conditions are almost always likely to evaluate as either true or false, then you can allow the compiler to optimize for this using the likely and unlikely macros. For example, when allocating memory you are almost always expecting this to succeed.

1 bvl = bvec\_alloc(gfp\_mask, nr\_iovecs, &idx); 2 if (unlikely(!bvl)) {

3 mempool\_free(bio, bio\_pool);

4 bio = NULL;

5 goto out;

6 }

When the unlikely macro is used, the compiler alters its machine instruc-

tion output, so that it continues along the false branch and only jumps if the condition is true. That avoids flushing the processor pipeline. The opposite happens if you use the likely macro.

20.2 Static keys

Static keys allow us to enable or disable kernel code paths based on the runtime state of a key. Their APIs have been available since 2010 (most architectures are already supported) and use self-modifying code to eliminate the overhead of cache and branch prediction. The most typical use case of static keys is for performance-sensitive kernel code, such as tracepoints, context switching, networking, etc. These hot paths of the kernel often contain branches and can be optimized easily using this technique. Before we can use static keys in the kernel, we need to make sure that gcc supports asm goto inline assembly, and the following kernel configurations are set:

1 CONFIG\_JUMP\_LABEL=y

2 CONFIG\_HAVE\_ARCH\_JUMP\_LABEL=y

3 CONFIG\_HAVE\_ARCH\_JUMP\_LABEL\_RELATIVE=y

To declare a static key, we need to define a global variable using the DEFINE\_STATIC\_KEY\_FALSE

or DEFINE\_STATIC\_KEY\_TRUE macro defined in [include/linux/jump\_label.h.](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/jump_label.h) This macro initializes the key with the given initial value, which is either false or true, respectively. For example, to declare a static key with an initial value of false, we can use the following code:

1 DEFINE\_STATIC\_KEY\_FALSE(fkey);

Once the static key has been declared, we need to add branching code to the

module that uses the static key. For example, the code includes a fastpath, where a no-op instruction will be generated at compile time as the key is initialized to false and the branch is unlikely to be taken.

1 pr\_info("fastpath 1\n");

2 if (static\_branch\_unlikely(&fkey)) 3 pr\_alert("do unlikely thing\n"); 4 pr\_info("fastpath 2\n");

If the key is enabled at runtime by calling static\_branch\_enable(&fkey),

the fastpath will be patched with an unconditional jump instruction to the slowpath code pr\_alert, so the branch will always be taken until the key is disabled again.

The following kernel module derived from chardev.c, demonstrates how the

static key works.

1 /\*

2 \* static\_key.c

3 \*/

4

5 #include

6 #include

7 #include

8 #include /\* for sprintf() \*/ 9 #include

10 #include

11 #include

12 #include /\* for get\_user and put\_user \*/ 13 #include /\* for static key macros \*/ 14 #include

15

16 #include

17

18 static int device\_open(struct inode \*inode, struct file \*file); 19 static int device\_release(struct inode \*inode, struct file \*file);

20 static ssize\_t device\_read(struct file \*file, char \_\_user \*buf, size\_t count, 21 loff\_t \*ppos); 22 static ssize\_t device\_write(struct file \*file, const char \_\_user \*buf, 23 size\_t count, loff\_t \*ppos); 24

25 #define DEVICE\_NAME "key\_state"

26 #define BUF\_LEN 10

27

28 static int major;

29

30 enum {

31 CDEV\_NOT\_USED,

32 CDEV\_EXCLUSIVE\_OPEN,

33 };

34

35 static atomic\_t already\_open = ATOMIC\_INIT(CDEV\_NOT\_USED); 36

37 static char msg[BUF\_LEN + 1];

38

39 static struct class \*cls;

40

41 static DEFINE\_STATIC\_KEY\_FALSE(fkey); 42

43 static struct file\_operations chardev\_fops = { 44 #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(6, 4, 0) 45 .owner = THIS\_MODULE,

46 #endif

47 .open = device\_open,

48 .release = device\_release,

49 .read = device\_read,

50 .write = device\_write,

51 };

52

53 static int \_\_init chardev\_init(void) 54 {

55 major = register\_chrdev(0, DEVICE\_NAME, &chardev\_fops); 56 if (major < 0) {

57 pr\_alert("Registering char device failed with %d\n", major); 58 return major;

59 }

60

61 pr\_info("I was assigned major number %d\n", major); 62

63 #if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(6, 4, 0) 64 cls = class\_create(DEVICE\_NAME); 65 #else

66 cls = class\_create(THIS\_MODULE, DEVICE\_NAME); 67 #endif

68

69 device\_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE\_NAME); 70

71 pr\_info("Device created on /dev/%s\n", DEVICE\_NAME); 72

73 return 0;

74 }

75

76 static void \_\_exit chardev\_exit(void)

77 {

78 device\_destroy(cls, MKDEV(major, 0)); 79 class\_destroy(cls);

80

81 /\* Unregister the device \*/

82 unregister\_chrdev(major, DEVICE\_NAME); 83 }

84

85 /\* Methods \*/

86

87 /\*\*

88 \* Called when a process tried to open the device file, like 89 \* cat /dev/key\_state

90 \*/

91 static int device\_open(struct inode \*inode, struct file \*file) 92 {

93 if (atomic\_cmpxchg(&already\_open, CDEV\_NOT\_USED, CDEV\_EXCLUSIVE\_OPEN)) 94 return-EBUSY;

95

96 sprintf(msg, static\_key\_enabled(&fkey) ? "enabled\n" : "disabled\n"); 97

98 pr\_info("fastpath 1\n");

99 if (static\_branch\_unlikely(&fkey)) 100 pr\_alert("do unlikely thing\n"); 101 pr\_info("fastpath 2\n");

102

103 return 0;

104 }

105

106 /\*\*

107 \* Called when a process closes the device file 108 \*/

109 static int device\_release(struct inode \*inode, struct file \*file) 110 {

111 /\* We are now ready for our next caller. \*/ 112 atomic\_set(&already\_open, CDEV\_NOT\_USED); 113

114 return 0;

115 }

116

117 /\*\*

118 \* Called when a process, which already opened the dev file, attempts to 119 \* read from it.

120 \*/

121 static ssize\_t device\_read(struct file \*filp, /\* see include/linux/fs.h \*/ 122 char \_\_user \*buffer, /\* buffer to fill with data \*/ 123 size\_t length, /\* length of the buffer \*/ 124 loff\_t \*offset) 125 {

126 /\* Number of the bytes actually written to the buffer \*/ 127 int bytes\_read = 0;

128 const char \*msg\_ptr = msg;

129

130 if (!\*(msg\_ptr + \*offset)) { /\* We are at the end of the message \*/ 131 \*offset = 0; /\* reset the offset \*/ 132 return 0; /\* signify end of file \*/ 133 }

134

135 msg\_ptr += \*offset;

136

137 /\* Actually put the data into the buffer \*/ 138 while (length && \*msg\_ptr) {

139 /\*\*

140 \* The buffer is in the user data segment, not the kernel 141 \* segment so "\*" assignment won't work. We have to use 142 \* put\_user which copies data from the kernel data segment to 143 \* the user data segment. 144 \*/

145 put\_user(\*(msg\_ptr++), buffer++); 146 length--;

147 bytes\_read++;

148 }

149

150 \*offset += bytes\_read;

151

152 /\* Most read functions return the number of bytes put into the buffer. \*/

153 return bytes\_read;

154 }

155

156 /\* Called when a process writes to dev file; echo "enable" > /dev/key\_state \*/

157 static ssize\_t device\_write(struct file \*filp, const char \_\_user \*buffer, 158 size\_t length, loff\_t \*offset) 159 {

160 char command[10];

161

162 if (length > 10) {

163 pr\_err("command exceeded 10 char\n"); 164 return-EINVAL;

165 }

166

167 if (copy\_from\_user(command, buffer, length)) 168 return-EFAULT;

169

170 if (strncmp(command, "enable", strlen("enable")) == 0) 171 static\_branch\_enable(&fkey); 172 else if (strncmp(command, "disable", strlen("disable")) == 0) 173 static\_branch\_disable(&fkey); 174 else {

175 pr\_err("Invalid command: %s\n", command); 176 return-EINVAL;

177 }

178

179 /\* Again, return the number of input characters used. \*/ 180 return length;

181 }

182

183 module\_init(chardev\_init);

184 module\_exit(chardev\_exit);

185

186 MODULE\_LICENSE("GPL");

To check the state of the static key, we can use the /dev/key\_state interface.

1 cat /dev/key\_state

This will display the current state of the key, which is disabled by default. To change the state of the static key, we can perform a write operation on

the file:

1 echo enable > /dev/key\_state

This will enable the static key, causing the code path to switch from the

fastpath to the slowpath.

In some cases, the key is enabled or disabled at initialization and never

changed, we can declare a static key as read-only, which means that it can only be toggled in the module init function. To declare a read-only static key, we can use the DEFINE\_STATIC\_KEY\_FALSE\_RO or DEFINE\_STATIC\_KEY\_TRUE\_RO macro instead. Attempts to change the key at runtime will result in a page fault. For

more information, see [Static keys](https://www.kernel.org/doc/Documentation/static-keys.txt)

21 Common Pitfalls

21.1 Using standard libraries

You can not do that. In a kernel module, you can only use kernel functions which are the functions you can see in /proc/kallsyms.

21.2 Disabling interrupts

You might need to do this for a short time and that is OK, but if you do not enable them afterwards, your system will be stuck and you will have to power it off.

22 Where To Go From Here?

For those deeply interested in kernel programming, [kernelnewbies.org](https://kernelnewbies.org) and the

[Documentation](https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation) subdirectory within the kernel source code are highly recom-mended. Although the latter may not always be straightforward, it serves as a valuable initial step for further exploration. Echoing Linus Torvalds’ perspec-tive, the most effective method to understand the kernel is through personal examination of the source code.

Contributions to this guide are welcome, especially if there are any signif-

icant inaccuracies identified. To contribute or report an issue, please initiate

an issue at [https://github.com/sysprog21/lkmpg.](https://github.com/sysprog21/lkmpg) Pull requests are greatly appreciated.

Happy hacking!

start() treatment

return is NULL?

No

next() treatment

No Yes

return is NULL?

Yes

stop() treatment

Figure 1: How seq\_file works