TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

NIRO ÅHMAN

TESTING OF LINUX KERNEL MODULES

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 5th of October 2016

NIRO ÅHMAN: Testing of Linux Kernel Modules
Tampere University of Technology
Master of Science thesis, 42 pages
June 2017
Master's Degree Programme in Information Technology
Major: Embedded Systems
Examiner: Prof. Timo D. Hämäläinen
Keywords: Linux, testing, device driver, Kernel module

Linux kernel's modular structure allows a developer to easily develop and implement drivers to the custom hardware usually found in embedded devices. Linux device drivers are implemented as part of the kernel which makes a defect in device driver or device easily break the whole system. Due to the nature of the embedded devices, testing of kernel drivers and related code is essential in finding possible defects before the end product is shipped to customer. This is especially important at Nokia Networks where System on a Chip (SoC) Application Specific Integrated Circuit (ASIC) chips are part of the critical communication system infrastructure.

While testing is necessary it is also time consuming and the tools available are limited due to kernel modules existing in kernel space. Usually inspecting the Linux kernel module source code manually is enough as the source code consists mostly from simple functions and parts that repeat in multiple modules.

This thesis work introduces the basics of Linux operating system with focus on the kernel modules. The work also explains the key terminology and tools available for testing Linux kernel and Linux kernel modules.

In the execution part a new testing method to test Linux kernel modules in user space is introduced. This method allows a more detailed and specified method of testing. For unit testing the kernel modules we used a method of mocking the kernel functions to work with user space test framework. This allows writing of specific tests for kernel modules to see that they behaved like expected.

It was noticed that there is a place for this method. As the framework was developed rapidly it needs improvement and refactoring of algorithms before taken into active use. Due to rapid development the framework is simple and new features should also be added for example, support for code coverage measurement.

NIRO ÅHMAN: Linuxin ytimen modulien testaus
Tampereen teknillinen yliopisto
Diplomityö, 42 sivua
Kesäkuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Timo D. Hämäläinen
Avainsanat: Linux, testaus, laiteajuri, ytimen moduli

Linux kernelin modulaarinen rakenne mahdollistaa laiteajureiden helpon kehittämisen mukautetuille laitteelle joita sulautetuissa laitteissa on. Linuxin laiteajurit liitetään osaksi kerneliä, minkä johdosta laiteajurissa tai laitteessa oleva vika helposti rikkoo koko järjestelmän. Sulautettujen laitteiden luonnon vuoksi, Linux laiteajureiden ja niihin liittyvän koodin testaaminen on erityisen tärkeää, jotta mahdolliset viat löydetään ennenkuin lopputuote lähetetään asiakkaalle. Tämä on erityisen tärkeää Nokia Networksillä jossa kehitetyt SoC ASIC piirit ovat tärkeä osa kriittistä tietoliikenne infraa.

Vaikka testaus on tärkeää se on myös aikaa vievää ja saatavilla olevat työkalut ovat rajoittuneet johtuen ytimen modulien toiminnasta ydintilassa. Yleensä Linuxin ytimen modulin lähdekoodin läpikäyminen käsin on riittäävää sillä se koostuu yleensä pienistä ja yksinkertaisista funktioista, jotka toistuvat useissa moduleissa.

Tämä diplomityö esittelee Linux käyttöjärjestelmän perusteet keskittyen ytimen moduleihin. Työ selittää myös tärkeimmän terminologian ja saatavilla olevat työkalut Linux kernelin ja sen modulien testaukseen.

Työn käytännöllisessä osuudessa esitellään uusi menetelmä, jolla ytimen moduleja voidaan testata käyttäjätilassa. Menetelmä mahdollistaa yksityiskohtaisemman tavan testata. Ytimen modulien yksikkö testaus sovitettiin käyttäjätilan testipenkkiin mallintamalla käytetyt ytimen funktiot. Tämä mahdollistaa yksityiskohtaisten testien kirjoittamisen ytimen moduleille, jotta voidaan nähdä niiden toimivat odotetusti.

Huomattiin että tälle menetelmälle on paikkansa. Koska menetelmä kehitettiin nopeasti sitä tarvitsee parantaa ja algoritmeja täytyy refaktoroida ennenkuin se voidaan ottaa aktiiviseen käyttöön. Nopean kehittämisen seurauksena menetelmä on yksinkertainen ja uusia ominaisuuksia olisi hyvä ottaa käyttöön, kuten tuki koodin kattavuudelle tutkimiselle.

PREFACE

This thesis was supposed to be a walk in the park and finished in a few months. I guess if your studies take ten years, you could expect that the thesis work will also take some time.

I would like to thank Nokia Networks for an interesting thesis topic and for providing challenges throughout the year. Practical part of the thesis work was done during the later half of 2016 while working in the Tampere SoC software organization and the thesis was written while working full time at Nokia Networks.

I would like to thank my wife Annamaija for patience with me even during the hardest times. The motivation I received was more than any man could hope for. I owe also a big thank you for all my three lovely daughters for allowing me to spend time away from them and providing me with breaks to clear my head. Huge thanks also for Petri for clearing that there is an h in through. I would also like to thank my whole SoC software team at Nokia Networks and my supervisor Petri Aalto, for welcoming me into the team and being there if I had any questions.

Also thanks to Professor Timo D. Hämäläinen for the great feedback.

Tampere, May 18, 2017

Niro Åhman

Table of Contents

List of Figures

List of Tables

List of Programs

LIST OF TERMS AND ABBREVIATIONS

C	Low level programming language popular in embedded systems.
C++	General purpose object oriented programming language.
FireWire	High speed serial bus interface standard.
Gcov	Source code analysis and statement-by-statement profiling tool.
Google Test	Google's C++ test framework.
Python	High-level programming language.
Qemu	Open source machine emulator and virtualizer.
SELinux	Security-Enhanced Linux is an open-source security addition to Linux kernel created by NSA.
Unity	Simple unit testing framework for C.
Unix	Family of multitasking, multi-user computer operating systems.
Virtualbox	Free and open-source virtual machine monitor for x86 computers.
Vmware workstation	Computer virtualization software.

ASIC	Application Specific Integrated Circuit.
CPU	Central Processing Unit.
FFF	Fake Function Framework.
FPGA	Field-Programmable Gate Array.
GDB	Gnu DeBugger.
GPL2	GNU General Public License, version 2.
IEEE	Institute of Electrical and Electronics Engineers.
IP	Intellectual Property.
IPC	InterProcess Communication.
JSON	JavaScript Object Notation.
KGDB	Kernel Gnu DeBugger.
MUT	Kernel Module Under Test.
SoC	System on a Chip.

1. INTRODUCTION

This thesis introduces Linux kernel module development in Linux system and a new method for testing Linux kernel modules. Topics are observed functionally not timely, as there is no real world timing needs for the kernel module testing and because device drivers are mainly used for device configuration.

1.1 Background

Nokia Networks develops new SoC ASIC chips for telecommunication products. As the SoC developed and the Linux device drivers controlling them are part of a critical communication infrastructure the requirements for hardware and software are high and they need to be tested thoroughly.

While the testing of Linux kernel modules is necessary it is also time consuming and the tools available are limited compared to testing tools for normal computer applications. This makes the testing methods for Linux kernel modules non-standard and complicated.

At Nokia Networks the Linux device drivers are developed in parallel to SoC ASIC design. This makes overall the development process faster but the device drivers need to be developed based only on the constantly updating specifications of the device. The kernel and modules developed can be tested on the real device only after the first engineering samples of chips are complete, usually in the last steps in the development process. For this reason the idea is to unit test code in our kernel modules and verify that the implementation works as they are supposed to.

1.2 Motivation

The main reason to research this topic was the limited amount of methods and tools available for testing the Linux kernel modules. This is causing the testing to take up a lot of resources as developers need to test their source codes mainly manually. Other reason is the overall need to test code as much as possible with the need to constantly improve our methods and techniques used for testing kernel modules. Another beneficial aspect would be the adding of automated testing.

Usually every kernel module developer has his/her own way of testing the kernel modules created with self made or tools commonly available. This creates the requirement to implement the new framework as simple as possible to allow everyone to use it.

Not being able to use user space tools like test frameworks and other tools is not optimal but it is understandable. This causes developers to develop different mechanisms to bypass the restrictions set by the Linux kernel space.

The aim of this thesis work was to find new methods for testing Linux kernel modules that would allow easier testing and a possibility to automate it. Testing of the kernel modules follows this principle and suffers from the penalties of testing support and tools for kernel space, that the user space has.

1.3 Thesis outline

Chapter 2 presents Linux operating system and Linux device driver development as literary review part of this thesis. Topics necessary of understating of this thesis are emphasized. Chapter 3 explains terminology and tools available for Linux kernel module testing. With the emphasis on unit testing during SoC ASIC development. Current methods used for Linux kernel module testing at Nokia Networks are described in Chapter 4.

In Chapter 5 a framework to test Linux kernel modules in user space is introduced as the execution part of this thesis work. The chapter also describes the reasoning why the framework was selected and the implementation requirements, architecture of the framework and describes the setup and usage of the framework. In Chapter 6 the additional kernel module testing framework is evaluated and compared against other methods currently available and future improvements are presented. Finally Chapter 7 concludes the Thesis and gives vision for future work.

2. LINUX OPERATING SYSTEM

Typically the term Linux is considered meaning the Linux kernel [16]. There exists numerous Linux distributions which have bundled different applications on top of the same core Linux kernel running in all of them. The modular structure allows Linux kernel to exist in different types of distributions from ultra lightweight to extra heavy desktop systems. Lightweight systems usually come with just the bare minimum kernel build where developers can add the support needed.

The modular structure of Linux kernel allows developers to build the kernel manually for embedded devices selecting only the needed drivers and options. For embedded devices that are scarce on resources and performance is an issue this allows the kernel image to be as small as possible and the kernel be as light as possible getting the best solution. The heavier desktop distributions like Ubuntu and Fedora come with prebuilt kernels with usually as much compiled into them as possible to support large set of peripheral devices, they also provide additional software and user interfaces on top of the base Linux kernel.

Linux is a collaborative project developed over the internet. The first version of Linux was created by Linus Torvalds in 1991 as an operating system for computers with Intel 80386 microprocessor.[16].

Currently the Linux kernel is developed as an open source project licensed with GNU General Public License, version 2 (GPL2) in Linus torvald's Github -repository [14][10]. New version of Linux kernel is released in every two-to-three months[15].

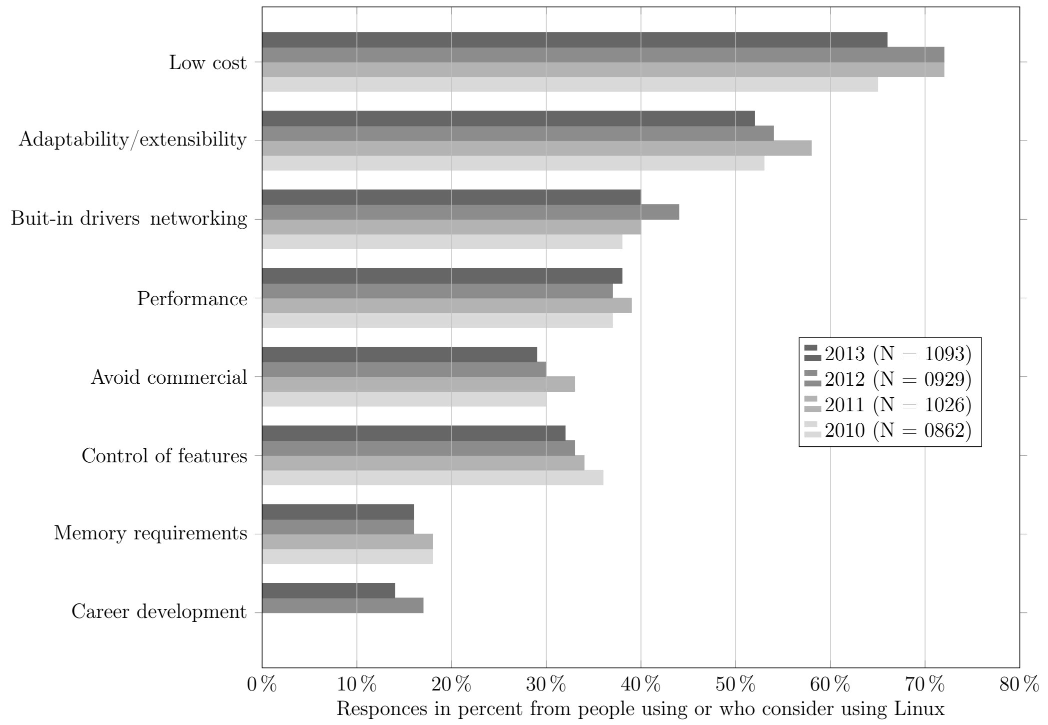

Figure 2.1 *Why are you interested in embedded Linux? [1]*

Linux is a popular operating system in embedded devices, based on the market study made by UBM technology. An online survey made in 2013 shows that embedded Linux is used or going to be used in more than half of the embedded projects. The study also shows that the need for commercial operating system is diminishing and open source systems are preferred. This can also seen in figure 2.1 that shows why developers are interested in embedded Linux. [1]

2.1 Components and architecture of the Linux system

Linux like many other Unix based systems is composed from three different parts [18]. The Linux kernel and system utilities are written in C language giving the kernel amazing portability to diverse hardware architectures [16]. As we can see in figure 2.2, C language is also the most popular language among embedded system developers.

Figure 2.3 illustrates the three components and how they relate to each other. Loadable kernel modules at the bottom are presented in more detail in Section 2.3.

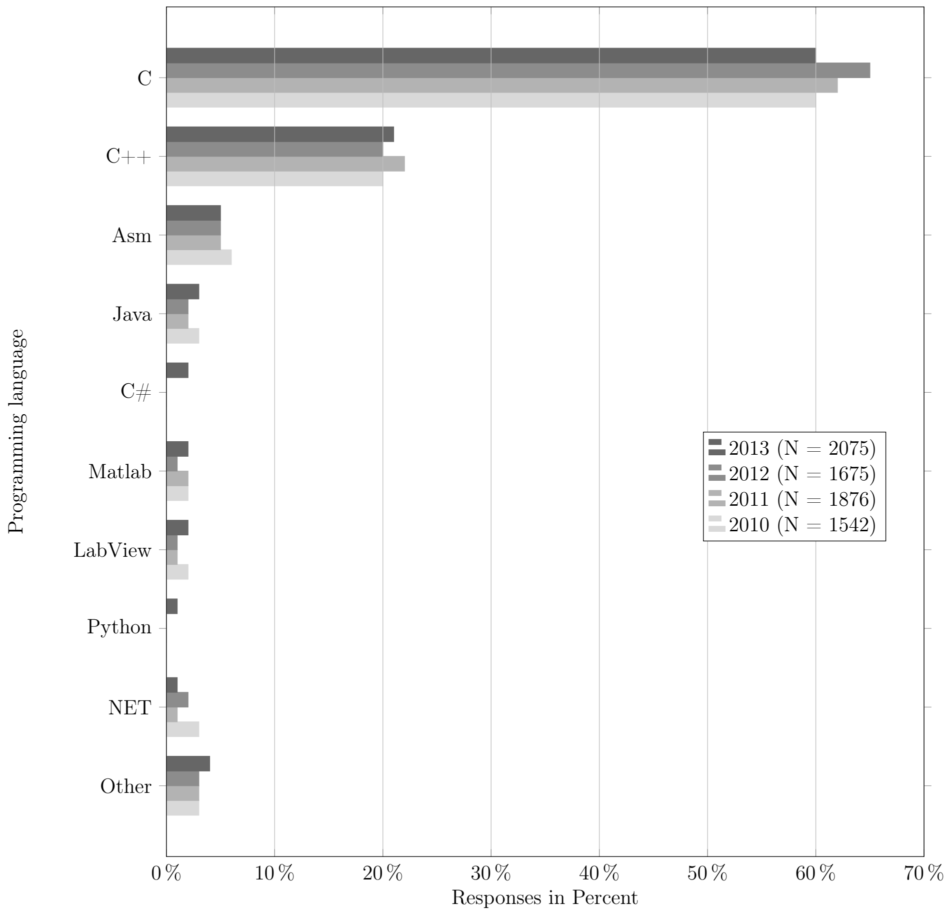

***Figure** 2.2 My current embedded project is programmed mostly in [1].*

Kernel is the core internals of Linux operating systems. The kernel is responsible for maintaining all the important abstractions and providing the basic functionality for all other parts of the system, manages hardware connected to the system and distributes system resources.[18][16]

System libraries define a standard set of functions through which an application can interact with the kernel. Most of the operating system code that does not need the full privileges of the kernel is implemented with these functions. [18]

System utilities are programs performing specialized management tasks. Some system utilities run only once configuring an aspect of the system. System utilities called daemons may run permanently handling tasks when required.[18] Programs such as a login process and shell are system utilities[16].

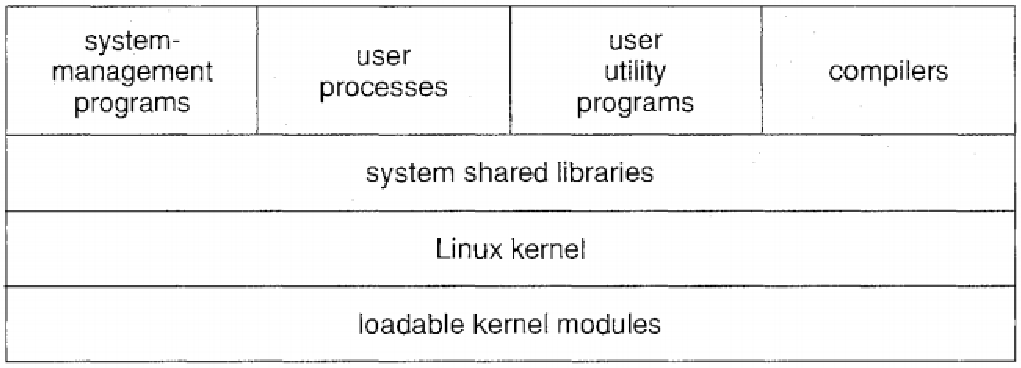

***Figure* 2.3** *Components of the Linux system [18, p.807]*

The architecture of the Linux operating system is illustrated in figure 2.4. Normal user applications in Linux systems are run in user mode in user space where only a limited subset of systems resources are available. The user space applications cannot access hardware directly or access memory outside of their region allocated by the kernel.

The applications communicate with the kernel through a system call interface that is described in Section 2.1.1. The kernel lies in the kernel space in an elevated system state called kernel mode, which includes a protected memory space and full access to hardware.[16]

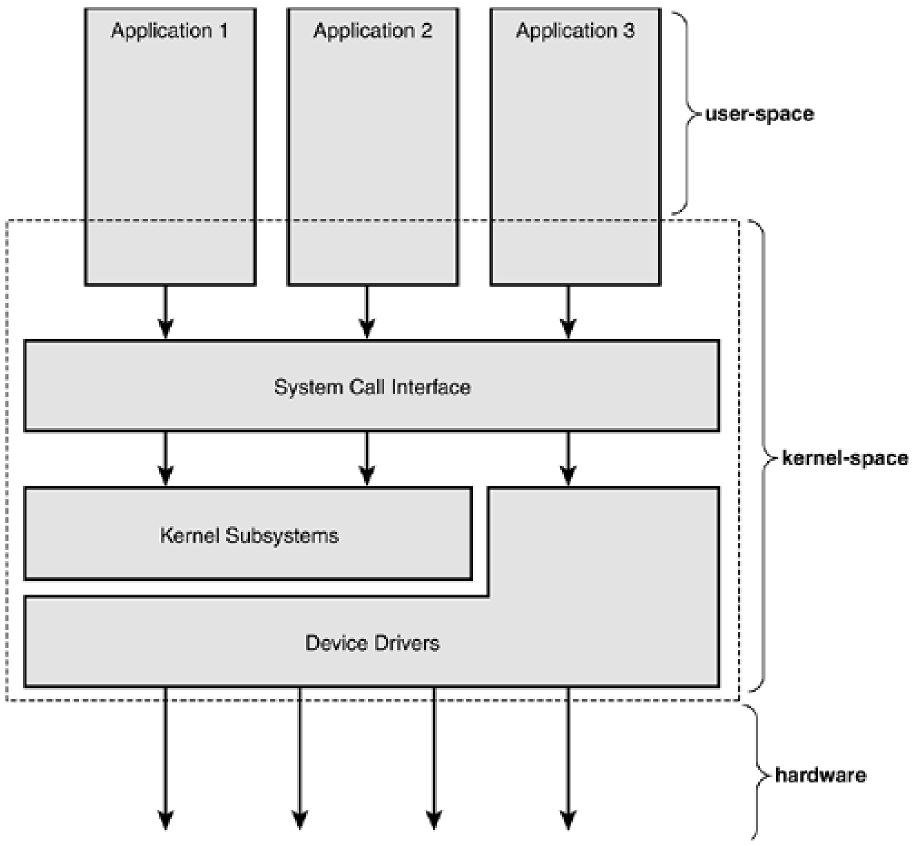

Figure *2.4 Relationship between applications, the kernel and hardware [16, p.6]*

Each processor in a Linux system can be seen of doing exactly one of the following three things [16]:

- Executing user code in a process in the user space.
- Executing on behalf of specified process in the kernel space.
- Handling an interrupt in the kernel space.

Separating the applications to run in the user space, and kernel and related system processes into kernel space gives additional security benefits. An application cannot write for example over important memory regions or load malicious code into the kernel.

2.1.1 System calls

Applications use system calls to communicate with the kernel. The application calls functions in a library that relies on to the system call interface to instruct the kernel to continue the task on behalf of the application. Some calls could provide more extensive features in them than the kernel function while others call the corresponding kernel functions directly.[16]

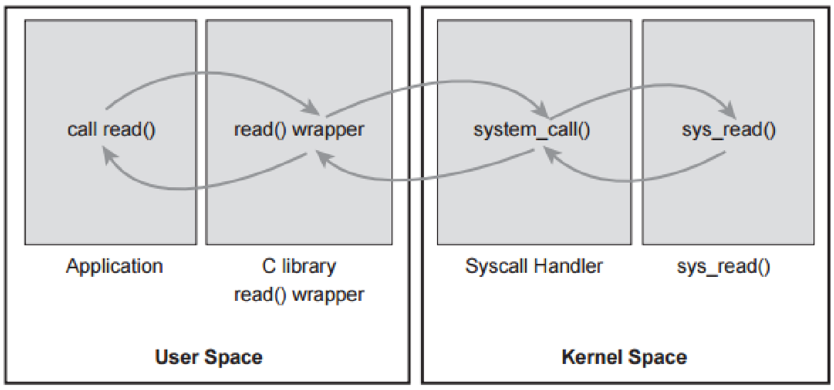

Figure 2.5 *Example of system call process [16, p.74].*

The process of using a *read* system call is described in 2.5. First the application calls *read()*-function that is presented by a C library wrapper which redirects the call into a system call for syscall handler in kernel space. Finally this executes the read call for a desired object. The return value is brought back the same way until it reaches the application.

2.1.2 Interrupt handling

The kernel also manages hardware connected to the system. All systems the Linux supports provide the concept of interrupts [16]

The basic concept is that when a device wants to communicate with the kernel, it issues an interrupt that interrupts the processor, which in turn interrupts the kernel. Every interrupt has a number that is used for identification. The kernel uses the number to execute an interrupt handler to process and respond to the interrupt. [16]

The kernel modules created for ASICs should handle interrupts from different hardware blocks and thus plugging the kernel the ability to communicate with the custom hardware.

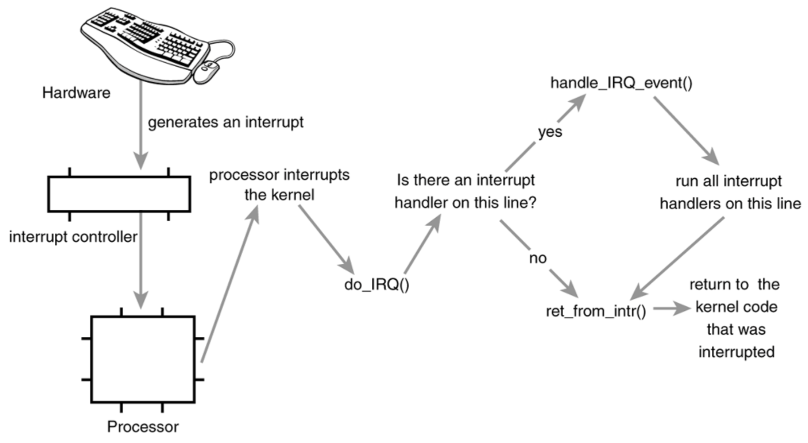
Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts the kernel
do_IRQ()
Is there an interrupt handler on this line?
yes
handle_IRQ_event()
run all interrupt handlers on this line
no
ret_from_intr()
return to the kernel code that was interrupted

Figure 2.6 *Example of an interrupt flow [16, p.223]*

A typical interrupt flow is presented in figure 2.6. The interrupt generated by hardware is routed through an interrupt controller to a processor that interrupts the kernel suspending all current operation. The kernel checks if any interrupt handlers are registered on the interrupt line in question and forwards the interrupt to that handler. Usually the device driver has a kernel module registered to handle the interrupts for the device in question. After the interrupt handler has completed or there was no interrupt handler, the kernel code is continued from where it was suspended.

2.2 Kernel subsystems

In general the kernel can be divided into monolithic and microkernels. The design of the monolithic kernel is simpler since it is are implemented entirely as a single process running in a single address space and usually exist as a single static binary. Because everything exists on the same address space, the communication within the kernel is trivial. In microkernels, the kernel functionality is split in separate processes, called servers that exist on different address spaces. The servers requiring privileged execution mode run in that mode while all the other servers run in user space. The microkernel communicates via InterProcess Communication (IPC) mechanism instead of direct communication, because the servers exist on different address spaces. Like most Unix systems, Linux has a monolithic kernel as it executes in a single address space entirely on the kernel space and is compiled into a single binary called the kernel image. However unlike other Unix systems Linux borrows many features from microkernels: modular design, kernel preemption, kernel threads and the capability to dynamically load kernel modules into kernel image. The modular structure allows the set of kernel features to be extended at runtime. This means that new features can be added or removed from the kernel while the system is running.[16][8]

Linux kernel developers use both ISO C99 and GNU C extensions for the C language making the kernel compile only with gcc. Recently Intel C compiler has also compiled the Linux kernel successfully.[16, 18]

The Unix system has several concurrent processes attending different tasks, each of these processes are requesting system resources like computing power, memory or some other resource. The kernel is executable code that handles all those requests. The role of the kernel can be split in the following five categories seen in 2.7.[8]

In addition to the subsystems presented in this chapter, there exist newer subsystems as the kernel constantly supports new features and technologies like FireWire.

2.2.1 Scheduler

Scheduling activity implements the abstraction of several processes on top of one or more Central Processing Unit (CPU) and allocating CPU time to different task withing the operating system. [8] [18] Linux uses two algorithms for scheduling processes, one shares CPU time fairly and preemtive across all CPU cores and the other is based on absolute priorities and designed for real-time tasks [18].

2.2.2 Memory management

The kernel builds up a virtual addressing space for all processes on top of the limited available actual resources. Different parts of the kernel interact with the memory-management subsystem through function calls [8]. Page sharing and copy-on-write is used by the memory management system to minimize the amount of duplicate data shared by different processes.[18]

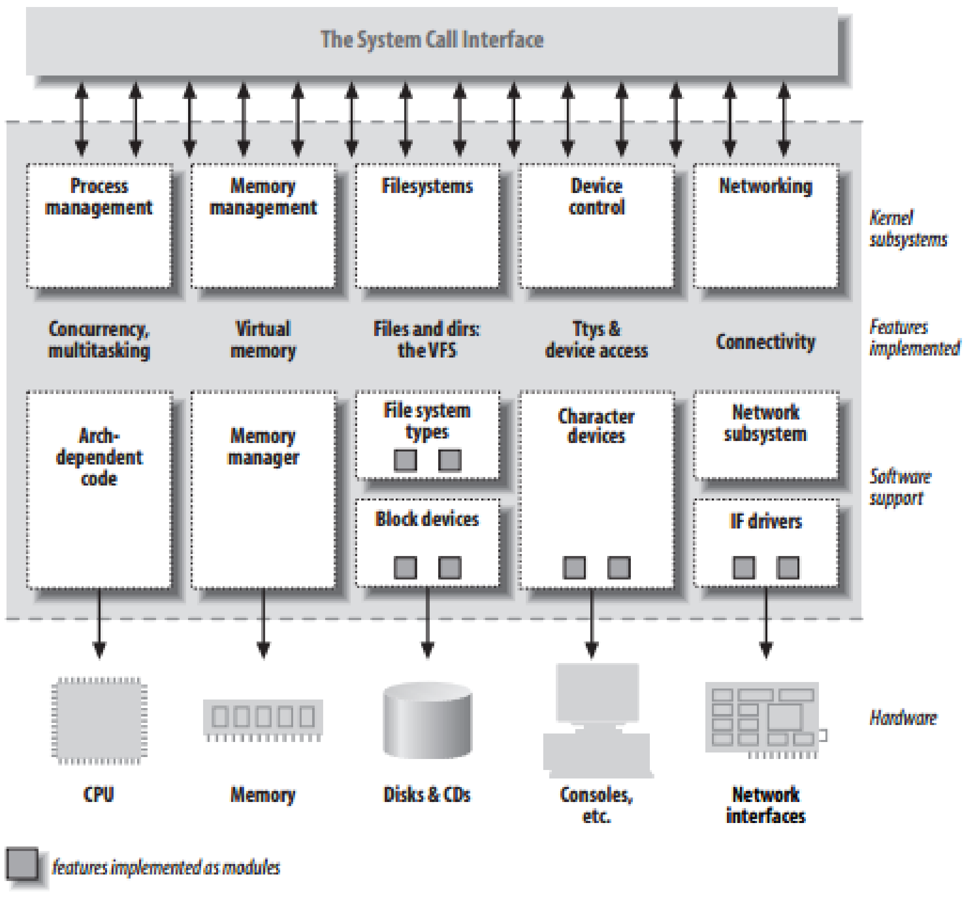

***Figure** 2.7 Split view of kernel [8, p.6]*

Linux memory management has two components. The first deals with physical memory allocations and freeing with pages or group of pages. The second component deals with memory mapped into address space of running processes, called virtual memory.

The physical memory is split in three zones listed in the table 2.1. Memory mapped to Central Processing Unit (CPU) address space is identified with ZONE_NORMAL and is used for most routine operations. [18]

Table 2.1 *Relationship of zones and physical addresses on the Intel 80x86 [18, p.821]*

Zone	**Physical memory**
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Zones are architecture specific and sized differently. Relationship of zones and physical addresses on Intel 80x86 are shown in table 2.1. Zones might be also missing on some architectures, for example if the architecture does not limit what DMA can access. The primary physical memory manager in the Linux kernel is called a page allocator and each of these zones listed above have their own allocator. The allocator is responsible for allocating and freeing all physical pages of the zone. [18]

The address space available to each process is defined by Linux virtual memory system. It manages loading of virtual memory pages created on demand from disk and swapping them back to disk if needed. A new virtual address space is created when a new program is run with *exec()* system call or when a new process is created with *fork()* system call. Virtual memory is responsible for the important task of relocating pages of memory from physical memory to disk when the memory is needed. [18]

2.2.3 Linux device drivers

Device control operations are performed by source code called device driver that is specific to the device being addressed excluding processor, memory and very few other entities. The kernel must have embedded the device driver for every peripheral present in the system.[8]

The device drivers make particular hardware device to connect to well-defined internal programming interface. The role of the device driver is to map the standardized calls that the user makes into device-specific operations that act on the real device. The modularity of Linux allows the drivers to be build separately from the rest of the kernel and plugged in when needed. Opposite to Linux kernel sources being available open source for everyone the device driver hides the details on how the device operates. Figure 2.4 shows the relationship of device drivers in the flow between applications and hardware.[8]

The Linux system distinguishes three fundamental device types listed below: character driver, block driver and a network driver. Modules usually implement one of these types and these modules are classified as a char module, a block module or a network module. Modules could be build huge, implementing different drivers. To support decomposition, the good practice is to create a different module for new functionalities. This improves scalability and extendability.[8]

Character devices A character device can be accessed as a stream of bytes usually through filesystem nodes like */dev/tty1.* The difference between a regular file and a character device is the ability to move back and forth. Most character devices are accessed sequentially making the filesystem node only a data channel between the system and the device. Character driver implements at least the open, close, read and write system calls.[8]

Block devices Block devices are accessed through filesystem nodes similar to character devices. Unlike most Unix systems Linux allows the application to read and write a block device like a character device permitting the transfer of any number of bytes at a time. Because of this the only difference between block devices and character devices in Linux systems is the kernel/driver interface which is completely different.[8]

Network interfaces The network interface is a device that is able to send and receive data in the form of packets from other hosts. Usually network interfaces are hardware devices but there exist also pure software devices like a loopback interface. Kernel's network subsystem is in charge of driving the network interface. As network interfaces are not stream-oriented devices they are not easily mapped to a filesystem node like */dev/eth0* and communication handled through read and write system calls. Instead the Unix defines a unique name for the interface like eth0 but the communication is done via functions related to packet transmission. [8]

Kernel modules are explained in more detail in Section 2.3

2.2.4 Filesystem

The filesystem concept is heavily used in Linux where almost everything can be treated as a file. The Linux system uses file abstraction resulting from kernel building a structured filesystem on top of unstructured hardware.[8]

2.2.5 Networking

Networking is managed by the system as most network operations are not specific for a single process. Incoming packets are for example asynchronous events that need to be collected, identified and dispatched to different processes.[8]

2.3 Kernel modules

The Linux kernel is modular and supports dynamic insertions and removal of code from itself during runtime, extending the feature set in the form of kernel module. A kernel module is a binary image, a loadable kernel object, that contains all the subroutines, data, entry and exit points grouped together. This enables the minimal base kernel image to be small and support adding of additional features and drivers when needed. Kernel modules are responsible for communication between the Linux kernel and attached hardware. For example, when hot plugging a new device, the necessary driver modules can be inserted into kernel, as is described in program 2.1.[8] The kernel keeps track of the modules and makes sure that their startup routine is called when the module is loaded and a cleanup routine is called before the module is unloaded.

The module support of Linux kernel has three components[18]:

Module management allows modules to be loaded into memory and to communicate with the rest of the kernel. Module management is much more than just loading the module into kernel, any references the module makes to kernel symbols or entry points must also be updated to point to correct locations in kernel's address space. Linux splits the module management in two sections to handle this. The other manages the sections of the kernel module code in kernel memory and the other handles symbols the kernel module is allowed to reference. [18]

Driver registration allows module to tell the kernel that a new driver is available. Without telling the kernel what kind of functionality the module provides it is just an isolated part of memory. [18] More information about driver registration was described in Section 2.2.3.

Conflict-resolution mechanism allows device drivers to reserve hardware resources and protect the reserved resources from other drivers accidentally using them. Linux kernel expects the module to register the hardware resources it is going to need, the decision to choose what to do if resource is busy or not available is left to the module. [18]

```
/* list modules packaged with kernel */
[root@noksu] $ ls /lib/modules/$(uname -r)/kernel/drivers/

/* add one of those modules into kernel */
[root@noksu] $ modprobe kernel_module

/* add kernel module from any folder into kernel */
[root@noksu] $ insmod kernel_module.ko

/* remove kernel module from kernel */
[root@noksu] $ rmmod kernel_module.ko
or
[root@noksu] $ modprobe -r kernel_module

/* log output can be seen from dmesg */
[root@noksu] $ dmesg

/* show modules loaded into kernel */
[root@noksu] $ lsmod
```

Program 2.1 *Example of adding and removing kernel module from kernel.*

Commands to load kernel modules is presented in program 2.1 modules packaged with the kernel can also be listed and loaded easily. Adding or removing modules from the kernel requires root privileges to prevent unauthorized user from loading their own possible hostile code into kernel. Currently also at least in Fedora 25 SELinux prevents addition of custom kernel modules even from root user but an exception to SELinux need to be made.

Ideally the kernel modules for universal devices would be in the source tree of Linux kernel for distribution with kernel to be available for anyone to use. [16] But in the case of embedded devices like in custom ASIC chips the module source code contains sensitive information. For these reasons debugging of proprietary modules, custom modifications to the original Linux kernel etc. are done in house without the support from Linux community or any external party.

```c
/*
 *  kernel_module.c
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>

static int __init hello_init(void)
{
        printk(KERN_INFO "Hello\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_INFO "Goodbye\n");
}

/* required module_init and module_exit macros
 * that are run when kernel module is loaded
 * or removed.
 */
module_init(hello_init);
module_exit(hello_exit);
```

Program 2.2 *Example of a simple Linux kernel module source file.*

Program 2.2 is the functional implementation of a simple kernel module written in C that shows how the kernel modules are structured. When the compiled kernel module is loaded into kernel it displays a log output a message "Hello" with KERN_INFO severity. When the module is unloaded it displays message "Goodbye".

3. TERMINOLOGY AND TOOLS FOR LINUX KERNEL MODULE TESTING

This chapter briefly describes the key terminology and tools commonly used for Linux kernel module testing. This work focuses only on the unit testing of the kernel modules. As the only thing needed is to test that the developed device driver source code does what it needs to do.

Testing in Linux can be split depending on whether the code we want to test runs in the kernel space or in the user space. The user space has multiple frameworks and testing tools widely available, which makes testing easy and simple. However the Linux kernel runs in kernel space where those methods are not available and testing is much more complicated.

The core parts of the Linux kernel has few specialized methods for testing different structures for example the Linux Test Project. Linux has a large community of developers and testing is generally heavily centered on the community to test each others code through design and code reviews, but that is not an option with private custom SoC kernel modules.[3]

3.1 Unit testing

Unit testing refers to testing the smallest segment of application that could be tested, for example it can be a test for a function or it can be a test that evaluates that a function, which writes values to memory has actually written them correctly. Test cases are kept simple, which return only true or false based on the result.[5] In embedded systems developing rapidly regular unit testing is critical since it substantially improves the reliability and quality of software [20].

Unit testing also helps the developer to write better code with less defects, as the coder writes smaller functions with pass or fail results that are easy to test. Unit test are usually coded after each segment of the source code is written, which allows better coverage and maintenance.

3.2 Test Frameworks

Another synonym for a test framework is an automated test environment. It provides ability to create test cases and test suites for program code and can be automated.

Unit tests are usually created while coding the part that is needed to be tested. This creates an extending test suite for the source code and one can automatically run all the tests or parts of them, for example smaller test suite that runs fast and is used often and then a larger suite to run once a day that contains more tests and runs slower. This allows noticing if something breaks on previously completed parts while implementing new features.

There are multiple test frameworks available for different programming languages and with different feature sets [4]. Google Test is one of the most common open source test framework. Google Test nowadays also includes Google Mock mocking framework.[11]

3.3 Mocks

Mock objects are one form of special test case objects that allows different kind of testing. Martin Fowler describes mocks as "Objects pre-programmed with expectations which form a specification of the calls they are expected to receive". Mocks do behavioral verification, we have described what kind of calls we expect the mock to have and we check that the program made the right calls for the mock.[9]

The implemented framework uses Fake Function Framework (FFF) as a mocking framework. Fake function framework is a simple mocking framework. The implementation is only one header file [2].

With FFF amount of calls to specific mock can be asserted and return values can be specified for return values or through reference parameter values. These values can also be set to be a sequence of different values. We can also specify different method of operation for different parameters the mock might receive. Like with many other fake frameworks, the fakes can be reset at start of each test case to

Table 3.1 *Fake Function Framework Cheat Sheet [2]*

Macro	Description	Example
FAKE_VOID_FUNC(fn [arg_types*]);	Define a fake function named fn returning void with n arguments	FAKE_VOID_FUNC(DISPLAY_output_message, const char*);
FAKE_VOID_FUNC(DISPLAY_output_message, const char*);	Define a fake function returning a value with type return_type taking n arguments	FAKE_VALUE_FUNC(int, DISPLAY_get_line_insert_index);
RESET_FAKE(fn);	Reset the state of fake function called fn	RESET_FAKE(DISPLAY_init);

keep the environment clean. Table 3.1 describes the basic syntax of FFF. Later in Section 5.3.3 FFF is used more.

For FFF to be implemented in a project, these lines need to be added. Now the calls to function DISPLAY_init() are routed to mocked version that without implementation defaults to an empty function.

3.4 Debugging and debuggers

Matloff and Salzman describe the rule that is essential for debugging: "Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true about the code actually are true. When you find that one of your assumptions is not true, you have found a clue to the location (if not the exact nature) of a bug." [17]. Debugging is not clearly defined, but it has some well known principles that are in use.

As with many of the things we focus on Linux developing and the developing tool available for it. While debugging can be done manually, using a debugging tool can assist and speed up the process a lot. Without debugging tools one needs to compile and run the program and then analyze the output that can be found in trace code and this has to be done for each bug found. Debugging tools allows developer to focus on finding the bugs without the need to focus on redoing the repetitive parts. GDB is the most common debugger available for all Linux distributions. While being a terminal program it provides many useful things for a developer. There exists also many graphical debugging tools and most of those are based on GDB and are just graphical front ends for it. With GDB we can simply run the program inside it and see the crash output. We can add breakpoints to different source code files and lines in them, this enables the process to pause on those breakpoints and we can evaluate e.g. values of variables.[17]

To debug a kernel running physically we need to have a serial connection. If we run the other computer virtually through Virtualbox or Vmware workstation we can use simulated connections. In both cases the Linux kernel running in another computer needs to have KGDB enabled so that the GDB from the host computer can connect to it. To debug kernel modules, we also need the physical address where the kernel module is loaded on the client computer so that we can set the GDB to tap on the right process. Using this method we can do normal GDB debugging described earlier to a kernel running live on client computer.

4. EXISTING TESTING METHODS OF LINUX KERNEL MODULES

The common method used for testing is manually inspecting the code which is described in more detail in Section 4.1. Section 4.2 describes an other often used method used together with manual inspection. It implements and additional kernel module, which passes interrupts to the kernel module under test.

When creating device drivers for a SoC we do not have access to the real device for testing purposes, as the physical hardware is developed at the same time and exist only as a specification. There exist multiple tools which try to fake the real hardware like software that mimics a real hardware register with software.

The kernel drivers can be tested on real processor architecture found on the hardware using Qemu, which supports multiple architectures, described in Section 4.4. Other method is to use a Field-Programmable Gate Array (FPGA) platform to create a test environment, with the possibility to program the FPGA with parts of our own hardware design. Implementing the whole ASIC design into FPGA platform is not possible because it does not fit. The FPGA environment can then be part of a larger testing environment or directly connected to developers computer.

In addition to the following methods there also exists an Intellectual Property (IP) project level automated build chains that also runs tests automatically for the projects source codes when new changes are committed into repository.

Final step used with each other method is code review presented in Section 4.5. It is used to test every line of source code going for production.

4.1 Manual code inspection

Manually inspecting the source code is a method for software testing. It means going through the code while developing it to find defects. The developer takes the role of end user and tries to test that the program is doing what it is supposed to do. It might include more specific testing phase where the programmer follows different paths the program takes in the code and looks that everything works as it should. However, even the best programmers can not notice all the defects and watching code intensively could get really tiresome after few hours lowering the success rate of finding defects. Manually testing the code also takes a a lot of time compared to automated methods available.

Inspecting the Linux kernel module source code manually is usually enough as kernel module source code consists mostly from simple functions and parts that repeat in multiple modules. That is one of the reasons why there does not exist a good tool for it in Linux kernel and the responsibility of testing is trusted heavily on the developer and the community.

Manual inspection is mainly used for preemptive code checking before the code is compiled. It can also be used after a possible defect is found to find the part of the code causing the defect. Linux kernel provides multiple log files for different parts of the system which may be useful when finding source of the defect. If the defect is causing the Linux kernel to panic and crash, a crash dump is generated if the option for it is enabled in the kernel settings. When developing kernel modules in kernel space the kernel crashes pretty often especially when the defect occurs when the module is loaded into the kernel. The crash dump provides information of what happened before the crash and it can be used to pinpoint for example the function that was executed just before the crash.

The developer can also try to pin point the location of the defect by implementing debug prints in the code to print out values of variables and generally hinting that the crash has not happened yet if the debug lines are printed.

4.2 Testing using fake module

Testing using fake module is good for testing the basic functionality of the kernel module. This method uses extra kernel module to aid with the kernel module testing. The method consists of a Google Test test framework in user space and a kernel module that is used to take commands from user space and that generates interrupts for the kernel module under test. Faked register is also used with this method to provide a common registry for the kernel modules involved to emulate the behavior of the real hardware. User space test framework is compiled into a program that is run with root privileges after the kernel modules have been loaded in the Linux kernel. Depending on the test type the user space application communicates data through device file to the tester kernel module and then waits for the result of the input data. The tester that resides in the kernel space can write to registers that the kernel module in test is watching thus faking interrupts and getting the module react as we desire. The tester module then reads the response of the kernel module from registers under test and passes the results to the user space test program that asserts the validity of the data received.

When a defect is found the GDB can be used to debug the user space test program. Kernel space code can be tested with the method described in Section 4.3. Apart from that the methods used for finding defects are usually the same as in manual inspection.

This method can also be used directly in a Linux workstation by compiling the kernel modules with the sources of kernel currently running on the system and then loading the compiled modules into kernel. But virtualization program like Virtualbox or Vmware workstation is preferred as a defect in kernel module often crashes the kernel and it is remarkably faster to reload a Virtualbox saved state than reboot the whole system.

4.3 Debugging with GDB and KGDB

GDB is used together with KGDB for debugging the running kernel and modules. This method requires a host and a virtual machine or it can be used between two computers using serial port connection. If the host computer is Windows then two Linux virtual machines are needed, one of them acts as the host and the other one as a client.

The benefit of this method is that the testing can be done for a real running kernel and modules. The KGDB is an extension to Linux kernel and needs to be added before this method is operational. This allows the GDB running in the host computer to break into the kernel running in the client computer to inspect memory, variables and see the call stack information similar to normal user space GDB debugger would. For example, to debug a kernel module running in the client computer we need to get and give the memory location where the module is loaded to the GDB running on the host machine so that it can look in the right place. [19]

4.4 Emulating with Qemu

Qemu is a generic and open source machine emulator that is able to emulate multiple different architectures. Qemu can also be used as a virtualizer by executing guest code directly on the host CPU with near native perfomance. [7]

Qemu gives one more solution to emulate the part of the actual device in developers own environment without any actual hardware, especially the architecture and CPU type. Qemu can boot the system from kernel image and does not need a boot loader which is a big benefit.

To run kernel specific to our hardware architecture in Qemu we need to cross compile the Linux kernel and modules for the specific architecture used on the device. Cross compiling means that we can for example compile a kernel image for ARM architecture on a computer that uses x86 architecture by using a cross compiler toolkit.

GDB and KGDB mentioned in Section 4.3 can be used with Qemu to gain extra benefit for debugging the kernel and kernel modules. Qemu gains the same benefits as running kernel in virtual machine but with addition of specific architecture and processor type. Qemu also boots and works a lot faster than full virtual machine OS.

4.5 Code review

Institute of Electrical and Electronics Engineers (IEEE) standard, Glossary of Software Engineering Terminology defines code review as: "A meeting at which software code is presented to project personnel, managers, users, customers or other interested parties for comments or approval."[6]. The purpose of code review is to detect defects in the code and design by evaluating the code by manually going through it [13]. Code review allows to catch those defects not found using unit tests.

In practice code review is used for each line of device driver source code. After most of the design modifications to hardware are made and the device driver passes the unit tests a group of developers go through the source code and usually the developer responsible for the coding presents the source code and design decisions made in it. This allows others to comment and notice if there is something missing or possible additions.

5. FRAMEWORK FOR TESTING KERNEL MODULES IN USER SPACE

When current methods used in Chapter 4 were researched an ability to be able to test the Linux kernel module code and step it line by line was noticed missing. It was found easier to implement it in the user space where the tools available are not so limited.

The first ideas were about importing the whole Linux kernel into user space, or editing or mocking parts of it and compile that against a test bench. That approach was found troublesome and at some parts confusing. One of the problems was that it required constant updates and monitoring as kernel is switched pretty often even in the development environment. Changes in the kernel has to be taken into account as there might be kernel modules using functions found only in the latest kernel versions. Kernel modules are also highly dependable of the Linux kernel sources and one cannot compile the Linux kernel files in user space. A mock framework had to be used to mock the calls to the Linux kernel with our own alternatives.

This lead to thinking about mocking the kernel functions found in the kernel modules being tested. The amount of time consumed would be much less if we just mock the kernel functions, and also give us faster access to get the kernel functions to behave as we want them to. It was also planned that there should be the possibility to automate the testing with the new framework.

This chapter describes an additional framework to test the Linux kernel modules. The framework was designed as a practical part of this thesis work. The reasoning why this framework was selected is presented in Section 5.1. The needs and other important criteria needed for the framework are described in Section 5.2. The design and implementation of the method is described in Section 5.3. Setup of the framework is introduced in Section 5.4. Evaluation and future improvements are pondered in Chapter 6.

5.1 Reasons why this approach was selected

The aim of the framework was to make the method as simple as possible, without the need to make any modifications to the source code of the kernel modules under test. The test cases should also be written as easily and as fast as possible. For this many developers are already using Google Test so implementing tests in that would have been an easy solution. But the idea of using Google Test was discontinued after it was thought cleaner to stay completely in C environment. The kernel module code is in C and Google Test is written in C++ so using it would bring unnecessary trouble as we would have to compile it with a C++ compiler. This is the reason Unity was chosen as the testing framework as it is also written in C. The similarity in syntax allows easier working for those already using Google Test in their work and with Unity we can have the testing framework written in the same language as Linux kernel and modules. Both frameworks allow us to implement different kind of assertions for tests and also non fatal expectations that show failure but do not break the testing.[11][12]

There did not exist before any viable solution to debug the kernel module code by creating automated test cases or debug a kernel function. This framework implements the kernel module in user space with mocked version of all kernel related functions. The source can also be debugged easily line by line without the need to run it in a virtual machine with GDB and KGDB mentioned earlier. It was also desired that we could step the kernel module code. With the implemented user space framework we can use GDB to step all of the kernel code, not just the test bench. Except of course the Linux kernel files that have been mocked out. This gives us a benefit to really see what is going on in every line of the code.

5.2 Implementation requirements

As the method is only used in house it is designed to work on the operating systems used in the company. The framework depends only on the kernel sources so it should be universal for all Linux systems and thus work with all of them.

The kernel sources used as a resource for mocking functions need to be compiled with the configuration files used. This ensures that all the necessary kernel header files that the kernel modules might need are generated.

The actual kernel module code Kernel Module Under Test (MUT) needs to be compiled correctly. The compiler will inform about errors in the compilation process and those need to be addressed first. For the mocking process to start the actual kernel module code needs to compile correctly so that we can differentiate the missing kernel functions needed in the mocked function creation phase. This is done by finding the missing kernel functions when compiling without the mocked functions.

A version of FFF and Unity are designed to remain as the versions included in the source files of the framework, meaning we are not getting the latest version automatically from their git repositories when running the setup script. This is done to prevent changes in FFF or Unity from breaking the framework as both frameworks are developing rapidly. If a newer version exists from those frameworks the files can be updated manually and then checked that those changes do not cause any issues on the framework. If everything is fine the newer versions can then be committed to replace the ones existing in the framework.

5.3 Architecture and implementation

The framework consists of two parts. The first part is a setup script written in Python that generates the actual testing environment. The kernel module or multiple modules with dependencies are described in the configuration file written in JavaScript Object Notation (JSON). The original setup script needs only be run if there is a change in the kernel modules source code involving an addition of a new kernel function. Without rerunning the script the fff mock will not be created. The second part is the test environment created by the Python script that allows one to create own tests using the mocked versions of kernel functions automatically created by the script.

The framework is implemented in a single folder that is distributed through version control system. From there the framework is easily usable with different kernel modules and projects.

```
TestFolder
├─ kmodtt_setup.py
├─ parameters.json
├─ include
│  └─ fff.h
├─ pythoncode
│  ├─ __init__.py
│  ├─ create_makefile.py
│  ├─ create_testfile.py
│  ├─ parse_includes.py
│  ├─ parse_mocks.py
│  └─ parse_parameters.py
├─ test.c
└─ vendor
   └─ Unity
```

Figure 5.1 *Directory tree of the new method.*

The file structure of the framework is described in Figure 5.1. The main Python script is in file *kmodtt_ setup.py*, the main script includes related python code found in the *pythoncode* -folder. *Include* contains the version of FFF used and *vendor* contains the version of Unity framework used. Parameters are set up in the *parameters.json* file.

5.3.1 Python setup script

The Python script aims to automate the framework initialization process by automatically creating the Unity and FFF environments and creating mocks for each kernel function found in the kernel module we want to test.

The scripts code is divided in multiple parts for clearer understanding but has only one control script called *kmodtt_ setup.py* that includes and uses the other parts.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

/* FFF is included and macros activated */
#include "fff.h"
DEFINE_FFF_GLOBALS

/* Unity macros defining test case and test runner*/
#define TEST_F(SUITE, NAME) void NAME()
#define RUN_TEST(SUITE, TESTNAME) printf(" Running %s.%s: \n", #SUITE, #TE

/* Mocked versions of Linux kernel functions are listed here*/
FAKE_VALUE_FUNC1(int, printk, const char*);

/* Function to reset function mocks between tests etc. */
void setup()
{
    RESET_FAKE(printk);
}

/* Empty test case */
TEST_F(testSuite, testCase)
{
}

/* Main function for adding test runners */
int main(void)
{
    RUN_TEST(testSuite, testCase);

    return 0;
}
```

Program 5.1 *An example of a test.c file generated by the Python script.*

Steps that the script makes when running the main Python script is described below and the usage of the script is defined in Section 5.4.

1. The script parses the parameters from *parameters.json* which are kernel folder, module folders and desired test folder path.

2. Test file called *test.c* is created in the testing folder. The test folder is cleared if it exists and created if it does not.

3. CMake configuration file *CMakeList.txt* is created based on the test file *test.c* for compilation.

4. Script runs CMake to create the compilation environment based on *CMakeList.txt*.

5. Makefile created in the previous step is then run with make -command to compile the testing environment. Note that at this point there is no kernel functions mocked and no kernel headers included so the compilation wont be successful. The compilation output is directed to a *makeoutput.txt* file.

6. Script uses the *makeoutput.txt* to parse out the needed kernel functions.

7. Script parses all kernel module source files and collects every kernel header used in them.

8. Based on the kernel headers collected in the previous step the script makes a fake kernel header structure where headers are in correct place to be included but empty. Without this structure we would have to remove the include lines from kernel modules and would not fulfill the need to keep changes to the kernel modules as minimal as possible.

9. Script parses the real kernel headers for the parameters and return value types used in the real functions and based on them creates the mocked versions with FFF syntax.

10. The *test.c* test file, shown in program 5.1, is created again, now including the mocked kernel functions and includes.

11. Script creates the *CMakeList.txt* -file again and uses CMake to create the Makefile. Finally the script compiles the test framework on last time with make-command and the framework should compile

5.3.2 The generated test framework

The test framework is generated in the test folder given to the script in the *parameters.json* -file. The structure of the test folder is described in figure 5.2 which shows the structure of the files generated by the Python script. Possible kernel module *include* files are in the *include* folder, the empty header structure is in the *include/linux/* -folder. Test cases are written in the *test.c* -file and the compilation instructions are in the *Makefile* where the compilation can be run with the *make* -command. The kernel module specific defines can be added in the *test.h* -file. Compilation recreates an executable binary file called kmodtt.

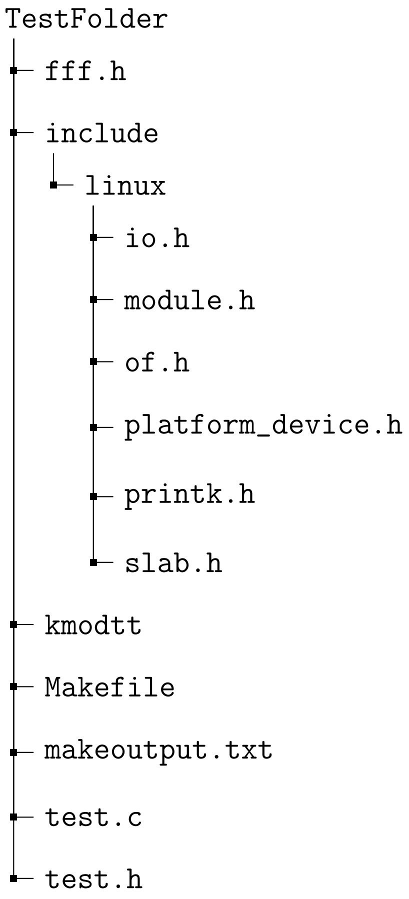

Figure 5.2 *Directory tree of the generated test folder.*

5.3.3 Mocking kernel functions

Linux kernel functions needed by the kernel module under test do not work directly as the kernel headers needed by the kernel module cannot be included into user space implementation directly. Including a Linux kernel header would cause an endless chain of includes which in the end leads to a machine specific functions. This is one of the main reasons we are mocking the kernel interface functions. Mocking also gives us benefits, as we can see the input to mocked function and control the output based on that. This gives us a wide range of testing capabilities.

The Mock framework used in this thesis is FFF. It is simple and works only by including one .h file. Mocked functions also default to ignoring the calls, so the framework compiles even thought the functionality is not yet written. The syntax for FFF can be found in table 3.1 and an example mocked function can be found in line 15 of program 5.1.

Configuring of the mocked functions e.g. return values, is left to the developer as the configurations vary between test cases.

5.3.4 Faking kernel and hardware structures

Structures that are available on the real hardware like memories and registers need also be faked as the real hardware is not available but also because we do not have access to the kernel space implementations from the test framework in user space.

Hardware registers are faked with a software register that emulates the internal structure of real registers. This is not simple as for example writing bit in a register could also trigger changes in multiple other different registers in the real device. This needs to be emulated to work also with the software registers and it can not be done with simple table-structure.

The kernel modules read interrupt data from memory and to replicate that in user space we use a structure to mimic memory. When MUT calls a kernel function to write to memory, the call is redirected to write it to our fake memory that enables bits like they would be on the device.

5.4 Setup and usage

The framework was designed to be simple and portable. The developers may have their own setup for different projects and kernels. The initialization of the framework is done with a python script that takes the kernel and module directories as parameters and outputs the test framework in to a folder given as a parameter.

The main setup script is run with the following command and for initialization there is nothing else to edit than the *settings.json* -file.

```
$ python kmodtsetup.py
```

This starts the script and it takes a while for the script to search functions from the kernel header files. After the script is complete the test framework is put in the folder described in the parameters. Dummy mocks for FFF and basic test case structure of Unity is build in test.c file and it can be seen in program 5.1.

The Python script generates the basic functionality and compiles the test framework first time into a *kmodtt* -binary which runs the test suite.

As per FFF syntax the default macros for functions needed mocking are created. It is up to user to create test cases and do customization to mocks.

A simple test case for the kernel module could be to mock the printk-function. We can, for example, test that it has been called only once and that the message was "Hello" by adding the following test case into the test.c in program 5.1.

```
 1 TEST_F(ModuleTest, printktest)
   {
 3     /* calling the function in kernel module */
       hello_init();
 5
       /* define the message we expect */
 7     char msg[] = "hello";

 9     /* assertations of what we expect happens */
       assert(printk_fake.call_count == 1);
11     assert(printk_fake.arg0_val == msg);

13     /* defines the return value of mocked version of prink-function */
       printk_fake.return_val = 0;
15 }
```

Program 5.2 *Example of a test case in test file.*

This test case is implemented as a test case called *printktest* that belongs to the test suite called *Moduletest.* These are described in line 1 of program 5.2. One test suite could have multiple test cases, which all run when executing the test binary. To run the test case described previously the following macro needs to be added into the main function of the test file.

```
RUN_TEST(ModuleTest, printktest);
```

The modified mocks and test cases can be easily copied to other implementations of this framework, which contain the same kernel functions as the copied part.

```
# compiler used
CC=gcc
# possible linker flags
LDFLAGS=
# name of the output binary
TARGET=kmodtt
# first batch of include files
INCLUDES= -include test.h -I/home/nahman/win/dippa/code/moduletest/include
# kernel specific macros are defined empty
DEFINES= -D__init= -D__exit= -D__iomem= -D__user=
# generated part based on the kernel module directories specified in param
SRCTEST = $(wildcard /home/nahman/win/dippa/code/moduletest/*.c)
SRCMODULE := $(wildcard /home/nahman/win/dippa/code/simplemodule/*.c)
INCMODULE += -I/home/nahman/win/dippa/code/simplemodule/include
# uniting variables to produce the lines to call the compiler
SOURCES += $(SRCTEST)
SOURCES += $(SRCMODULE)
INCLUDES += $(INCMODULE)
CFLAGS=  $(INCLUDES) $(DEFINES)
OBJECTS=$(SOURCES:.c=.o)
all: $(SOURCES) $(TARGET)
$(TARGET): $(OBJECTS)
        $(CC) -o $@ $^ $(LDFLAGS)
%.o: %.c
        $(CC) -c -o $@ $^ $(CFLAGS)
clean:
        rm $(OBJECTS)
```

Program 5.3 *Example of Makefile used to compile the test framework.*

After adding tests the test framework can be compiled again with the *make* - command that runs the *Makefile* seen in program 5.3.

6. EVALUATION OF THE NEW TESTING FRAMEWORK

This chapter evaluates and measures the benefits of the additional kernel module test framework described in Chapter 5.

6.1 Overall performance

The unit tests performed with the testing framework are small and there is no impact on performance on normal computers even with larger test suites. Typical time to run the test cases is measured in few seconds. The time to run the build script could take a minute depending on the amount of different kernel functions on the framework. This is caused by algorithm that currently performs the same kernel header file search for each kernel function found in the kernel module being tested.

The Biggest benefit of the test framework is that the tests can be easily automated as part of the build process and repeated multiple times each time even with different values if needed. This is a huge benefit when comparing to manual testing and checking things. With automated test framework it is possible to detect new defects caused by code change. These changes could be a new feature or a fix to some other defect. These are hard to see with manual testing.

This new framework is good for testing kernel modules with decent amount of kernel functions used but with complicated logic in them. The framework can find bugs and logic mistakes that might exist. It also has the ability to test that the input from the kernel functions is exactly as we want it to be.

6.2 Usability

Setting up the test framework is easy as it was designed for the environment used by the developers who are the key audience for the framework. The syntax of Unity was easy to understand as it is similar to Google Tests syntax, which is currently the most used in user space testing. The mocking framework FFF has also a straight forward syntax that is easy to adopt.

The biggest problem with the mocking framework was at first the implementation and mocking functions manually. This was resolved by the script that automated the search and mocking of the kernel functions.

A big disappointment was that for a larger set of modules that used different set of kernel functions the framework did not work as well out of the box and needed added support for the python parser scripts. This might be a problem also in the future as usually for larger devices multiple kernel modules are depending on each other.

The test framework runs as a user space application allowing debugging of the source code line by line using GDB. Other user space testing tools are also available if needed in the future.

6.3 Comparison against other methods and frameworks

Comparing this framework against other existing methods mentioned in Chapter 4 is difficult, because each method serves a different purpose and to work in full effect multiple methods are used in serial. The closest method to compare would be manual testing. One clear example would be testing that kernel function is called X number of times with parameters Y and Z. Manually testing this would take a long time as we would have to count manually every time the kernel functions gets called and check if the parameters are correct or not and then increase the count. The test framework would have a test case that could ask from the mock framework how many times it was called with parameters Y and Z and assert if that equals the value described in the test case.

The best method available for testing different types of modules based on code complicity, reusage and the amount of kernel functions are described in table 6.1. The framework is highly usable if there are many different kernel functions used in the kernel module and the module code is complicated and needed often in multiple projects or modules. If the module code is less complicated or the re usability is low it might be excess amount of work to implement the test cases and mock functions.

Table 6.1 *Choosing the best method for different types of kernel modules.*

Amount of calls to kernel functions	Code reused rarely		Code reused frequently	
	Simple code	**complicated code**	**Simple code**	**complicated code**
Many	manual	manual	manual	framework
Few	manual	framework	framework	framework

The framework shines when there is only few calls to kernel functions but the code is complicated. This keeps the amount of mocked functions needed and work required low but the full advantage of generated mocks can be taken into use.

6.4 Current status and future improvements

Like every other method available to test the Linux kernel modules, this framework has also cases where it excels and cases it does not fit very well. This framework is still under development and needs support periodically. It is not expected that the framework works straight out of the box for any larger set of modules. The script might also have defects with mocking different kernel functions not already used and tested currently, this means that there might be functions mocked wrong and little fixes need to be done to get the script to support those. The work to maintain this framework diminishes over time as the script used for mocking constantly improves and new functions are tested. The framework has an ability to search from different versions of the Linux kernel sources specified in the settings file. There might be changes in the upcoming versions of kernel that mix up the python script in the future. At this point it might be good to note that the Linux kernel should be backwards compatible. For example, a driver created for Linux 2.6.13 should work in Linux kernel version 4.19.2 .

As the largest part of this framework is the python script that generates the test framework that uses elsewhere developed FFF and Unity, most of the additional options that might need overhauling in the future are related to the python generation part. Included frameworks FFF and Unity evolve independently and upgrading them to newer versions must be done manually.

The Python code for the current version of the test framework was rapidly created to work as a prototype and works as just a setup script that needs to be run rarely. Optimizing at this point does not provide any benefits. Further optimizing the algorithms especially the one that finds the functions in kernel headers will make the script that generates the test framework run remarkably faster.

The Python script could have the option to only update the mocks of the test file, so that when the kernel module source code changes we can just reinitialize the framework which creates the mocks while saving our manually edited test cases and other modifications. If some more work is to be done to the framework it could be a usable tool among the others to test the Linux kernel modules.

It might also be good to add code coverage support for the test framework and scripts using Gcov. This should be easily achieved.

7. CONCLUSION

This thesis work introduced Linux operating system and kernel modules used in device drivers focusing on the testing aspects. Testing of Linux kernel and kernel modules was also compared to testing methods of normal user space programs. Current methods used for testing Linux kernel modules were reviewed and evaluated to find the needs of an additional testing framework.

The execution part in Chapter 5 introduced a new framework to test Linux kernel modules in user space. The framework allowed more detailed and specified method for testing the Linux kernel modules. The framework used mocks of the Linux kernel functions to allow it to be compiled in user space.

The new framework was evaluated in Chapter 6. It was seen that the test framework worked well for example with modules with complicated code. It also added the possibility for automated testing and possibility to customize the mocks.

For Nokia Networks this thesis work provided a usable prototype of a simple and maintainable framework for unit testing the Linux kernel modules in user space. Current methods were also reviewed and evaluated. This provided valuable information of their usefulness and comprehensiveness.

In my opinion this thesis work was successful in reaching the goals set for it in Chapter 1. This thesis work achieved to clarify to the reader what kind of testing is involved in Linux device driver development and overall information about Linux and kernel modules. When the current methods were evaluated there was found a need for a new method so this new method was implemented.

This thesis taught me a lot about testing, problem solving and Linux kernel in general. Going through multiple Linux kernel files, device drivers and functions improved my professional understanding remarkably.

BIBLIOGRAPHY

[1] Embedded market study, 2013. UBM Technology. [WWW] Referenced: 01.05.2017, Available: http://www.iuma.ulpgc.es/~nunez/UBM2013EmbeddedMarketStudyb.pdf.

[2] Fake function framework. Meekrosoft. [WWW] Referenced: 01.05.2017, Available: https://github.com/meekrosoft/fff.

[3] Linux test project. Linux Community. [WWW] Referenced: 01.05.2017, Available: https://github.com/linux-test-project/ltp/wiki/GettingStarted.

[4] List of unit testing frameworks. Wikipedia community. [WWW] Referenced: 01.05.2017, Available: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

[5] Unit testing. Agile Alliance. [WWW] Referenced: 01.05.2017, Available: https://www.agilealliance.org/glossary/unit-test/.

[6] *IEEE Std 610.12-1990.* IEEE Standard Glossary of Software Engineering Terminology, 1990.

[7] F. Bellard. Qemu wiki. [WWW] Referenced: 01.05.2017, Available: http://wiki.qemu.org/Main_Page.

[8] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers.* O'Reilly Media, 2005, 640 p.

[9] M. Fowler, *"Mocks aren't Stubs"*, 2007, [WWW] Referenced: 05.05.2017, Available: https://martinfowler.com/articles/mocksArentStubs.html.

[10] *GNU General Public Lisence, version 2*, Free Software Foundation Inc, [WWW] Referenced 04.05.2017, Available: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

[11] *Google C++ Testing Framework*, Google Inc, [WWW] Referenced: 03.05.2017, Available: https://github.com/google/googletest/blob/master/googletest/docs/Primer.md.

[12] Unity test framework. Throw the switch. [WWW] Referenced: 01.05.2017, Available: https://github.com/ThrowTheSwitch/Unity.

[13] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management.* Wiley-IEEE Computer Society Press, 2007.

[14] *Linux kernels Github repository*, L. Torvalds and Linux community, [WWW] Referenced: 04.05.2017, Available: https://github.com/torvalds/linux.

[15] *Linux-project*, Linux Foundation, Website, Available(referenced 04.05.2017): https://www.linuxfoundation.org/projects/Linux.

[16] R. Love, *Linux kernel development.* Addison Wesley, 2010, 440 p.

[17] N. Matloff and P. Salzman, *The Art of Debugging with GDB, DDD and Eclipse.* No Starch Press,US, 2008.

[18] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts.* John Wiley & Sons Inc, 2009, 982 p.

[19] J. Wessel, *Using kgdb, kdb and the kernel debugger internals*, [WWW] Referenced: 15.05.2017, Available: https://www.kernel.org/doc/htmldocs/kgdb/.

[20] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building Embedded Linux Systems.* O'Reilly Media, 2008, 391 p.