
NOTICE1: Please do not copy the contents of this page to your blog. You can share this page but please share with the original link. That is how we compliment the authors of good documents and open source projects.

NOTICE2: Please notice that low-level programming is out of trend and currently there are not many companies hiring low-level developer. It is getting harder for me to find a job. If you haven't started a professional career yet, I would like to recommend you consider other fields either carefully.

NOTICE3: If you want a quick start, go to "How to start?".

- Low-Level Programming University
 - What is it?
 - What is the Low Level
 - Theory
 - Languages
 - * Assembly
 - * C language
 - * Rust language
 - Applications
 - * Hardware and Firmware
 - * Linux kernel and device driver
 - References
 - * Other applications
 - Future of low-level programming
 - How to start?
- Translations
- Who am I?

Low-Level Programming University

What is it?

I'm inspired by google-interview-university. I'd like to share my experience and show a roadmap to becoming a low-level programmer because I have found that these skills are not as common as they once were. In addition, many students and beginners ask me how they could become low-level programmers and Linux kernel engineers.

This page cannot include every link/book/course. For example, this page introduces Arduino but there is not detailed information about Arduino and embedded systems. You should go further yourself. You have the keyword “Arduino” with which you can start. So your next step is probably googling Arduino, buying a kit, and doing something for yourself, not collecting links or free books. Please remember this page is just a roadmap for beginners.

Low-level programming is a part of computer science. Absolutely it would be much better to get education in computer science first. * Path to a free self-taught education in Computer Science!

What is the Low-Level?

I classify low-level programming as programming that is very close to the machine, using a lower level programming language like C or assembly. This is in contrast to higher-level programming, typical of user-space applications, using high level languages (e.g. Python, Java). * Wikipedia: Low-level programming language

Yes, systems programming is a very close concept to low-level programming. This page includes the hardware design and firmware development that is not included in systems programming. * Wikipedia: System programming

Finally, this page includes topics ranging from hardware components to the Linux kernel. That is a huge range of layers. A one page document can never cover the details of all the layers, so the aim of this document is to serve as a starting point for low-level programming.

Theory

There are two background theories to low-level programming: * Computer Architecture * Operating Systems

I think the best way to learn theory is by taking a course. Reading books is not bad but takes too much time and effort. You can find many good classes on online universities, for instance, Coursera.org and edx.org. Theory is theory. I don't think you need to get an A+ in the class, just understand the big picture. You'll get better and better with experience.

Let me introduce several books that I've read. They are commonly used as textbooks in universities. If there is no class with these books in your university, it's worth spending some time reading them. * Computer Architecture * Computer Architecture, Fifth Edition: A Quantitative Approach * Computer Systems: A Programmer's Perspective * Computer Organization and Design, Fourth Edition: The Hardware/Software Interface * Operating Systems * The Magic Garden Explained: The Internals of UNIX System V Release 4 an Open Systems Design * The Design of the UNIX Operating System * Operating

Systems: Internals and Design Principles by William Stallings * Recommended Courses * CS401: Operating Systems from saylor.org * General Programming Skill * Structure and Interpretation of Computer Programs * It's about how to be a good Software programmer. You need not only theory but only technique because programming is a kind of craftwork. * If you learn Lisp/Scheme, you should be able to learn any other language quickly. * I've solved about 80% exercises. It should be worth to try every single exercise. * Hardware Design * Build Your Own 8086 Microprocessor Kit * If you don't build your HW board, you don't understand what physical memory mapped device is. * Modern APs includes so many IPs. So you don't have a chance to understand how CPU core and peripheral devices are connected. * When you build your own 8086 kit, you have a chance to locate each peripheral devices on the physical memory. And you can set how the main HW components (BUS, IRQ, Clock, Power and etc) works with your own eyes. * I built the 8086 kit in my University. It was one of the most valuable courses I've ever taken. Try to build your own HW kit. It would be better if the HW is older and simpler because you should do more for yourself. * Google "8086 kit". You would be able to find some web-sites you can buy a HW scheme, parts and manuals.

There is an infinite list of good books. I don't want to say that you should read many books. Just read one book carefully. Whenever you learn a theory, implement simulation code of it. **Implementing one thing is better than knowing one hundred theories.**

Languages

Assembly

Choose one between x86 or ARM. No need to know both. It doesn't matter to know assembly language. The essential thing is understanding the internals of a CPU and computer. So you don't need to practice the assembly of the latest CPU. Select 8086 or Cortex-M.

- 8086 assembly programming with emu8086
 - basic concepts of CPU and computer architecture
 - basic concepts of C programming language
- 64bit assembly programming (translation in progress)
 - basic concepts of modern CPU and computer architecture
 - basic concepts of disassembling and debugging of C code
 - *need help for translation*
- Learning assembly for linux-x64
 - pure 64-bit assembly programming with NASM and inline assembly with GCC

-
- ARM Architecture Reference Manual, 2nd Edition
 - Complete reference on ARM programming
 - Computer Organization and Design
 - MIPS Edition
 - ARM Edition
 - RISC-V Edition
 - Academic books that explain how every component of a computer work from the ground up.
 - Explains in detail the different concepts that make up computer architecture.
 - They are not targeted at readers who wish to become proficient in a specific assembly language.
 - The MIPS and ARM edition cover the same topics but by dissecting a different architecture.
 - Both editions contain examples in the x86 world

C language

There is no shortcut. Just read the entire book and solve all the exercises.

- C Programming: A Modern Approach, 2nd Edition
- The C Programming Language 2nd Edition
- Modern C: Jens Gustedt. Modern C. Manning, 2019, 9781617295812. fahal-02383654f
 - For new standard of C
- Is Parallel Programming Hard, And, If So, What Can You Do About It?
 - raw implementation of synchronization with C
 - Essential for large scale C programming (especially for kernel programming)
- C Project Based Tutorials?
 - If you finish reading one or two C programming books, then you MUST make something.
 - Choose whatever you like.
 - First make on your own and then compare with someone else's source code. It is very important to compare your source and others. You can improve your skill only when you read the other's source and learn better methods. Books are dead and source is live.
- C and other languages based projects
 - find more interesting projects

-
- Michael Abrash's Graphics Programming Black Book, Special Edition
 - Reference on optimization using C and a bit of x86 assembly
 - Starts from the 8088 up to today
 - Special focus on low-level graphics optimization
 - Framework and plugin design in C
 - How to develop framework and plugin in C for large scale software
 - Very basic programming tips for Linux kernel source reading

If you want to be expert of C programming, visit <https://leetcode.com/>. Good luck!

Rust language

I am sure that the next language for the systems programming would be Rust. I will make a list what I did to learn Rust.

Linus Torvalds said "Unless something odd happens, it [Rust] will make it into 6.1."

- The Rust Programming Language
 - Great introduction, but lack of examples and exercises.
- Rust by Example
 - While reading "The Rust Programming Language", you can find examples and exercises here.
 - But there are not many exercises you can do something for yourself. Only some examples includes "do-this" exercises and they are very simple.
- Programming Rust, 2nd
 - Deeper introduction, but still lack of examples and exercises.
- Exercism
 - Good exercises to practice individual features of RUST.
 - I am not sure Mentors are working actively but it would be enough to compare your solution with others.
 - ★ After submitting your solution, you can see other's solutions with "Community solutions" tab (since Exercism V3).
 - ★ Many easy level exercises are for functional feature such as map/filter/any and etc.
- Easy rust

-
- A book written in easy English.
 - Youtube materials provided: <https://www.youtube.com/playlist?list=PLfllocyHVgsRwLkTAhG0E-2QxCf-ozBkk>
 - Let's get rusty
 - There are many Youtubers uploading Rust course but I enjoyed this course most.
 - He has been uploading the latest news for Rust. It's worth subscribing.
 - Rust for Linux
 - See the example sources and check how Rust got into the Linux kernel

Applications

Hardware and Firmware

If you want to be an embedded systems engineer, it would be best to start from a simple hardware kit, rather than starting with the latest ARM chipset.

- Arduino Start Kit
 - There are many series of Arduinos but “Arduino Start Kit” has the most simple processor(ATmega328P) and guide book
 - ATmega328P has an 8-bit core which is a good place to start digital circuit design and firmware development.
 - You don't need to know how to draw schematics and layouts and assemble the chips.
 - But you do need to know how to read schematics and understand how the chips are connected.
 - Firmware developers should be able to read the schematics and figure out how to send data to the target device.
 - Follow the guide book!
- 8086 manual
 - If you're a beginner to x86 architecture, 8086 is also very good guide for processor architecture and x86 assembly
- 80386 manual
 - Best guide for protected mode and paging mechanism of 80x86 processor
 - Web version: <https://pdos.csail.mit.edu/6.828/2011/readings/i386/toc.htm>

At this point, you should be good to start the latest ARM or x86 processor. * <https://www.raspberrypi.org/>
* <https://beagleboard.org/> * <https://www.arduino.cc/en/ArduinoCertified/IntelEdison>

For example, the Raspberry Pi board has a Cortex-A53 Processor that supports a 64-bit instruction set. This allows you to experience a modern processor architecture with rPi. Yes, you can buy it... but... what are you going to do with it? If you have no target project, you would be likely to throw the board into a drawer and forget it like other gadgets you may have bought before.

So, I recommend one project for you. * Making your own kernel * Good references: <https://www.reddit.com/r/osdev/>
* Learning operating system development using Linux kernel and Raspberry Pi * (description of the project) This repository contains a step-by-step guide that teaches how to create a simple operating system (OS) kernel from scratch...(skip)...Each lesson is designed in such a way that it first explains how some kernel feature is implemented in the RPi OS, and then it tries to demonstrate how the same functionality works in the Linux kernel.

I've made a toy kernel that supports 64-bit long mode, paging and very simple context switching. Making a toy kernel is good way to understand modern computer architecture and hardware control.

In fact, you have already the latest processor and the latest hardware devices. Your laptop! Your desktop! You already have all that you need in order to start! You don't need to buy anything. The qemu emulator can emulate the latest ARM processors and Intel processors. So everything you need is already on hand. There are many toy kernels and documents you can refer to. Just install qemu emulator and make a tiny kernel that just boots, turns on paging, and prints some messages.

Other toy kernels: * <https://littleosbook.github.io/> * <https://tuhdo.github.io/os01/>

Linux kernel and device driver

You don't need to make a complete operating system. Join the Linux community and participate in development.

Some resources for Linux kernel and device driver development from beginner to advanced.
* Books: Read the following in order * The Design of the Unix Operating System * The basic concepts of Unix are applied into all operating systems. * This book is a very good place to learn the core concepts of operating systems. * Linux Device Drivers * Make all examples for yourself * Linux Kernel Development * Understand the design of the Linux Kernel * Understanding the Linux Kernel * Read this book and the kernel source v2.6 at the same time
* Never start with the latest version, v2.6 is enough! * Use qemu and gdb to run the kernel source line by line * <http://stackoverflow.com/questions/11408041/how-to-debug-the-linux-kernel-with-gdb-and-qemu> * <https://github.com/gurugio/linuxdeveloptip/blob/master/qemu-gdb-kdump.md> * Use busybox to make the simplest filesystem that takes only one second to boot

* <https://github.com/gurugio/linuxdeveloptip/blob/master/minikernelwithbusybox.md> * Other resources: Free resources I recommend * Linux device driver labs * Practical guide and excellent exercises making Linux device drivers with essential kernel APIs * I think this document introduces almost all essential kernel APIs. * The Eudypytula Challenge * *Sadly, this challenge does not accept new challengers because there is no challenge anymore.* The maintainer said he/she is planning a new format. I hope it comes back ASAP. * But you can find the questions of the challenge with Google. Some people already uploaded what they did. Find the questions and try to solve them on your own, and compare your solution with others. * This is like an awesome private teacher who guides you on what to do. * If you don't know what to do, just start this. * Learning operating system development using Linux kernel and Raspberry Pi * This project is not completed yet. * I always think making a kernel similar to the Linux kernel is the best way to understand the Linux kernel. * Block layer and device driver * start from a simple block device driver example (Ramdisk) with multi-queue mode * go forward to block layer * I completed translation into English. Please send me your feedback. * md driver of Linux kernel(Korean) * how mdadm tool works and how it calls md driver * how md driver works * Lions' Commentary on UNIX 6th Edition, with Source Code

References

Check when you need something

- Free-electrons homepage
 - many slide files introducing good topics, specially ARM-linux
- Julia Evans's posting: You can be a kernel hacker!
 - guide to start kernel programming

Other applications

Yes, you might not be interested in Linux or firmware. If so, you can find other applications: * Windows systems programming & device drivers * Security * Reverse engineering

I don't have any knowledge about those applications. Please send me any information for beginners.

Kernels and drivers are not all of low-level programming. One more important application of low-level programming is the software-defined storage or distributed filesystem. Detailed descriptions of them is beyond the scope of this document but there is an excellent course where you can try a simple distributed filesystem. * Course: <https://pdos.csail.mit.edu/archive/6.824-2012/> * reference Source: <https://github.com/srned/yfs>

Future of low-level programming

I do not know the future, but I keep my eye on Rust. * <https://hacks.mozilla.org/2016/11/rust-and-the-future-of-systems-programming/>

If I could have one week free and alone, I would learn Rust. That is because Rust is the latest language with which I can develop Linux device drivers. * <https://github.com/tsgates/rust.ko>

IoT is new trend, so it's worth to check what OSs are for IoT. ARM, Samsung and some companies has their own realtime OS but sadly many of them are closed source. But Linux Foundation also has a solution: Zephyr * <https://www.zephyrproject.org/>

Typical cloud servers have many layers; for instance, host OS, KVM driver, qemu process, guest OS and service application. A container has been developed to provide light virtualization. In the near future, a new concept of OS, a so-called library OS or Unikernel, would replace the typical stack of SW for virtualization. * <http://unikernel.org/>

Big data and cloud computing require bigger and bigger storage. Some disks directly attached to server machines cannot satisfy the required capacity, stability and performance. Therefore there has been research to make huge storage systems with many storage machines connected by a high speed network. It used to be focused on making one huge storage volume. But currently they are providing many volumes dedicated for many virtual machines. * https://en.wikipedia.org/wiki/Software-defined_storage * https://en.wikipedia.org/wiki/Clustered_file_system * [https://en.wikipedia.org/wiki/Ceph_\(software\)](https://en.wikipedia.org/wiki/Ceph_(software))

How to start?

I received an email asking how to start. There are many information about books, courses and projects in this page. It is my mistake to forget to write how to start. Unfortunately, there is no King's Road to King's Landing. I will just write what I did in order. If you have already done something, please skip it. AGAIN, this is just an example that you could do in order, just in case if you do not know how to start or what to do.

- Reading OS theory books: at least "The Design of the UNIX Operating System by Maurice J. Bach"
- Learn assembly and C
 - 8086 assembly programming with emu8086
 - * It is enough if you understand the concept of assembly programming. You do not need to do something practical.
 - The C Programming Language 2nd Edition
 - * DO YOUR BEST TO solve every single exercise!
 - C Programming: A Modern Approach, 2nd Edition

-
- Do something practical with C
 - C Project Based Tutorials?: Find one or two interesting projects and make your own project.
 - leetcode.com: If you cannot find an interesting project, it would be also good to focus on data structure and algorithm.
 - Do a hardware project
 - Raspberrypi or Arduino does not matter. You need experience to control a hardware directly with only C. ONLY C!
 - I recommend to buy a ATmega128 kit and make a firmware to turn on/off LEDs, detect switch input and display message on the text LCD. Motor control program is also a very good project: for instance, the line tracer.
 - DO NOT use any library. You should make everything on your own, except program downloader.
 - Basic of the Linux kernel
 - Low-level programming is very close to the operating system. You should know inside of the OS.
 - Start with drivers
 - * Read Linux Device Drivers
 - * Linux device driver labs
 - * The Eudypytula Challenge
 - Read Linux Kernel Development to understand the internal of Linux kernel.
 - Go to the professional field
 - If you want to be a professional Linux Kernel Developer
 - * must read Understanding the Linux Kernel
 - Then try to make a toy kernel
 - Learn operating system development using Linux kernel and Raspberry Pi
 - Making your own kernel
 - Write the github link to your kernel on your resume (Don't forget to write the detailed description in commit message)
 - * Check the latest issues at <https://lwn.net/> and join it.
 - Check "Recent kernel patches" at "<https://lwn.net/Kernel/>" or direct link <https://lwn.net/Kernel/Patches>
 - Find an interesting patch to you. Try to understand the source code. Of course it would be really difficult but try. You will be closer and closer whenever you try.

-
- Build kernel and test it on your system. For example, performance test, stability test with LTP(<https://linux-test-project.github.io/>) or static code analysis tools inside of kernel.
 - Report any problem if you find any: compile warnings/errors, performance drop, kernel panic/oops or any problem
 - If it works well, report that with the spec of your system. The patch owner would write a “Reviewed-by” tag with your name.
 - Find your name in kernel git log
- Or find other topics
 - ★ There are many fields where the low-level engineer can work: security, Compiler, Firmware, robot/car and so on

Translations

Please send me the pull request if you'd like to translate this page. I'll list it here.

- Chinese (Traditional)
- Chinese (Simplified)
- Portuguese (Brazilian)
- Italian
- Czech
- Russian
- Turkish
- Persian
- Spanish
- French

Who am I?

I'm inspired by google-interview-university. I'd like to share my experience and show a roadmap to becoming a low-level programmer because I have found that these skills are not as common as they once were. In addition, many students and beginners ask me how they could become low-level programmers and Linux kernel engineers.

FYI, I have over 10 years of experience as a low-level programmer: * 80x86 Assembly programming * Hardware device with ATmel chip and firmware * C language system programming for Unix * Device driver in Linux * Linux kernel: page allocation * Linux kernel: block device driver and md module