DOI:10.1145/3469118

George V. Neville-Neil

Article development led by acmqueue
queue.acm.org

Kode Vicious
In Praise of the Disassembler

There is much to be learned from the lower-level details of hardware.

Dear KV,

I have read enough of your columns to see that one of your frequent themes is that source code is meant to be read by people, including one's future selves, and that how that code is processed by an interpreter such as Python or a compiler is less important than making the code clear to the next reader. You seem to be saying that our tools will work out what we mean and that we should treat the interpreter or compiler as a black box that magically turns our source into running code. I feel you are ignoring an important part of understanding software, which is what happens when your compiled code executes on a machine—after all, no computer executes C, C++, or Rust directly; they are running a compiled binary. What happens when you have a bug that appears in the binary only because of a mistake by the compiler, linker, assembler, or other part of the tool chain, which must occur from time to time. What then?

Dissembling at the Back End

Dear Dissembling,

Indeed, there have been many people and many movements within the software industry over the past 50 years that have shifted developers and development further away from machine code and assembly language—and not without good reasons. The abstractions of higher-level languages over the admittedly brief history of computing have allowed the explosion of software and services that nonprogrammers take for granted every day. Those of us who often work down in the bowels of technology know that these abstractions, and the movement of developers away from the machine, come with costs.

There are many problems in software systems that cannot be properly understood without a good understanding of the lower-level—some might argue the lowest-level—details of the machines we work on, and it shocks and angers me when I try to explain such things and get blank stares from people who have been in the industry for many years. The attitude can be summed up in a line I heard back in high school when I was learning my first assembly language on what was even then an ancient machine, the DEC-10. "You'll never need to use this because in the future all languages will be high-level like Fortran and Cobol" was the refrain of the math teachers who proctored our computer lab.

What is interesting about this quote is that it is wrong in two different ways. The first is that the languages they chose, though they are still in use today, are by far not the majority languages in software development, meaning that they were not the be-all and end-all that these teachers thought they were. The second fallacy is the idea that what I learned in my first brush with assembly and machine code would be useless in my career, when nothing has been further from the truth.

While it is true that the last piece of assembler I wrote by hand (and which wound up in a commercial product) was written 30 years ago, the lessons I learned by interacting directly with the hardware—without even the cushion of C (a.k.a. assembly with for loops)—remain with me to this day and have allowed me to track down difficult bugs, as well as significant performance problems. I have written in the past about understanding algorithms and how these expressions of problem solving relate to good performance[a] but the other side of this coin is understanding how the machine on which your algorithms run actually works. What does it take to understand the nether regions of computer systems? KV's advice comes in three parts: Start small. Start small. Start small.

Start at the small end of processor and computer hardware. A modern laptop, desktop, or server system is a fantastically complex piece of equipment that for the most part grew by the accretion of features that will utterly distract anyone who is new to this area of computing. You want to start with a small system, with few features, and with a small instruction set, so that you can, hopefully, jam nearly all the details into your head at once. Symmetric

a See https://bit.ly/3io2PuF

multiprocessing, multilevel caches, speculative execution of instructions, long pipelines, and all the rest of the innovations in hardware that have been added to improve performance as we have hit the wall at the end of Moore's Law are important to learn—later. Very few people start learning piano by sitting down to play Mozart or Fats Waller, and so you should not attempt to scale the heights of a superscaler processor on day one.

A good place to start in 2021 is with a small, cheap, embedded processor, and by this I definitely do not mean the Raspberry Pi or any of that ilk. The current Pi is based on a complex ARMv8 design. Do not start there. A better place to start is with the popular Atmel AVR chips, which are available on Arduino and other boards used by hobbyists and embedded-systems designers. These processors are eight-bit—yes, you read that correctly, eight-bit—systems much like the early microcomputers of the 1980s, and they have small memories, slow processing speeds, a small number of registers, and most importantly, a small and easy-to-remember set of assembly operations (opcodes).

These constraints actually help you learn the machine without a lot of extraneous distractions. Another advantage of this architecture is that instructions take either one or two clock cycles, depending on whether they are internal or I/O operations. Having instructions with a small, known cycle time makes it easier to think about the performance of the code you're looking at. Getting experience with performance in this way is key to being able to understand the performance of larger and more complex architectures. KV cannot emphasize enough that you want to start with an assembly language that is small and easy to understand. Go look at any data book for a big Intel, ARM, or other processor and you will see what I mean. The AVR instruction set fits on a single page.

Read small programs. I have given this advice to developers of languages at all levels, from high to low, but when you are trying to learn the lowest levels of the machine, it is even more important. The canonical "Hello, World" program taught to all C programmers results in a binary file with millions of instructions when it is statically linked. A description of what happens when you execute

What does it take to understand the nether regions of computer systems?

it is the subject of an excellent talk by Brooks Davis at the 2016 Technical BSDCan Conference.[b]

The examples in most beginning programming tutorials for the Arduino—those that blink an LED—are the size of an example I would suggest. Later, you can develop a code editor that morphs into a mail reader and Web browser, which seems to be the course of all software projects over time, but for now just turn the light on and off. There are two ways to examine these small programs. The first is to look at the output of various compilers that build C or higher-level code for these embedded devices, such as the Arduino IDE, or LLVM- and GNU-based cross-compilers. You can look at the code by dumping it with an objdump program, or you can look at it in a debugger if you have one that understands the AVR instruction set. The debugger is the more interesting environment because you can both read the code and also execute it, stop it, inspect registers and memory, and the like. Both LLDB from LLVM and GDB from GNU have assembler modes, so you can even switch between a higher-level language and assembler.

Write small pieces of low-level code. That piece of assembly I wrote 30 years ago was only a couple of pages long, in part because what it did was simple (pulling audio data from a parallel port on a microcomputer) and because it was written using a powerful complex instruction set computer (CISC) assembly language (Motorola 68K). A CISC assembly language is closer to C than machine code and often has opcodes that do quite a bit on your behalf. The AVR can be considered in the family of reduced instruction set computer (RISC) processors, where each instruction is very simple, and complex operations must be built out of these. Much of the raw assembly that is still written today is along this model of a few pages of instructions.

When you are starting out you want to be able to hold the entire program in your head if at all possible. Once you are conversant with your first, simple assembly language and the machine architecture you are working with, it will be completely possible to look at a page or two of your assembly and know not only what it is supposed to do but also what the machine will do for you step by step. When you look at a high-level language, you should be able to understand what you mean it to do, but often you have no idea just how your intent will be translated into action—assembly and machine code is where the action is.

Developing these skills will take you far past blinking an LED on a hobby board. Being able to apply these same skills to larger machines, with more complex architectures, makes it possible to find all kinds of Heisenbugs,[c] optimize systems for power and performance, as well as understand the ramifications of low-level security attacks such as return-oriented programming (ROP) and buffer overflows. Without these skills, you are relegated to the cloudy upper layers of software, which is fine, until the tools or your hardware or the gods of software fail you.

KV

b See https://bit.ly/3gl3CK1

c See https://bit.ly/2TMk1j3

Related articles on queue.acm.org

Programming in Franglais
Rodney Bates
https://bit.ly/3cL5QBN

GNL Is Not Linux
Kode Vicious
https://bit.ly/3g47lNn

Coding for the Code
Friedrich Steimann and Thomas Kühne
https://bit.ly/3gh2rvj

George V. Neville-Neil (kv@acm.org) is the proprietor of Neville-Neil Consulting and co-chair of the ACM *Queue* editorial board. He works on networking and operating systems code for fun and profit, teaches courses on various programming-related subjects, and encourages your comments, quips, and code snips pertaining to his *Communications* column.

Copyright held by author.