Tiny Tools

Gerard J. Holzmann

MANY PROGRAMMERS like the convenience of IDEs when developing code. The best examples are Microsoft’s Visual Studio for Windows and Eclipse for Unix-like systems, which both have been around for many years. You get all the features you need to build and debug software, and a lot of other things you’ll probably never realize are also there. You can use all these features without having to know very much about what goes on behind the screen.

And there’s the rub. If you’re like me, you want to know precisely what goes on behind the screen, and you want to be able to control every bit of it. IDEs can sometimes feel as if they’re taking over every last corner of your computer, leaving you wondering how much bigger your machine would have to be to make things run a little more smoothly.

So what follows is for those of us who don’t use IDEs. It’s for the bare-metal programmers, who prefer to write code using their own screen editor and who do everything else with command-line tools. There are no real conveniences you need to give up to work this way, and you gain a better understanding of your development environment and retain full control to change, extend, or improve it whenever you find better ways to do things.

Bare-Metal Programming
Many developers who write embedded software work this way. Mission-critical flight software development for spacecraft, for instance, is typically done on Linux systems, using standard screen editors and a lot of command-line tools. The target code will ultimately execute under real-time operating systems, on custom-built hardware. Productivity, accuracy, and reliability really matter in these applications, thus creating a strong case for the bare-metal approach.

What types of questions do software developers typically ask when...
working on an application? There actually aren't that many. The two most common questions are these:

- Where is this variable declared, and where is it used?
- How is this function defined, and where is it used?

In an IDE, you can click on a name, and its definition pops up. Or, you can click on a function name and see the screen switch to its definition. That can be an unwelcome context switch as you're staring at a subtle piece of code you're trying to understand. It's rather simple to answer such queries with a few small command-line tools that you run in their own window, separate from the editor, so that you never lose context.

You can find some of the functionality you need in standalone tools that work almost like an IDE, but without the editor. A good example is the Cscope tool, which Joe Steffen at Bell Labs developed in the early '80s. Curiously, the original motivation for Cscope was that using command-line calls to scan code was too slow, so Joe built a database to speed up the resolution of standard types of queries. Today, though, computers are fast enough that this performance argument no longer applies, unless the information you need is buried so deeply in a directory hierarchy that you might do better to refactor your code first, before trying to add to it. This means you don't have to construct a database, store it, and keep it up to date before you can answer routine queries about your code.

To become a good bare-metal programmer, you must be comfortable with shell programming and at least the standard set of core Unix text-processing tools: grep, sed, and awk. I always use the Bash shell that's currently the default on Linux systems, but almost any other modern shell will do. The text-processing tools can quickly extract information from potentially large numbers of files, and present it in its most useful form.

When I have to evaluate the C source code for an application, I often use a couple of simple queries to get a first impression of the overall quality. They're hard to replicate in an IDE. For example, most coding standards for safety-critical software have the rule that all switch statements contain a default clause. This is Rule 16.4 in the most recent MISRA-C guidelines for critical code. How hard is this to check? You can fire up a serious static source code analyzer to check it, or you can type these two queries:

```bash
$ cat `find . -name ".*c" -print` | grep -c -e switch 1065
$ cat `find . -name ".*c" -print` | grep -c -e default 809
```

You'll get two numbers that should be fairly close if the rule is followed. The check is, of course, not precise because the keywords switch and default could also appear in strings or comments, but that won't likely dominate the results. I'll get back to more precise ways to check these types of things shortly.

It's just as simple to quickly check for uses of goto or continue statements that many coding standards also frown upon, the use of union data structures, compiler-dependent pragma directives, or risky calls to routines such as strcpy instead of strncpy.

It gets a little harder if you want to check for a rule that's very similar to the use of a default clause in switch statements. How, for instance, would you check if every if-then-else-if chain always ends with a final else? This is Rule 15.7 in MISRA-C, which ensures that no cases are accidently missed if the final else is omitted. In the following code fragment, such a missed case occurs when both d and c2 are false:

```c
if (c1) { s1; } else if (c2) { s2; }
```

In this case, you can’t just look for the keyword combination else if because there’s more context here that you must take into account. Static analyzers normally don’t check for these patterns either, so you’d have to come up with an alternative. It’s not hard to write such a check, using only basic command-line tools, as I show later.

Another common thing programmers need to do when developing or browsing code is print a suspicious code fragment, prefixed with a filename and line numbers. In these
cases, it helps to have a few simple extra commands in your toolset. So, let’s talk about that first.

A Survival Kit
When I move to a new system, because I’m either upgrading my desktop or setting up a temporary work environment on someone else’s machine, I first install a survival kit of tools I wrote that can make life easier. These tools are small and simple enough that they’re guaranteed to work anywhere. They’re no more than about 20 lines long, with just two exceptions.

The first exception is a tokenizer for C code of about 600 lines, which I’ll talk about shortly. The other is a small emulation of the main features of the sam screen editor that Rob Pike developed about 30 years ago at Bell Labs. The sam editor is a favorite of many former Bell Labs researchers, although curiously not of Rob himself.3 The full Unix version of sam is about 15K lines of C and isn’t installed on many systems. My survival version of the key features is about one-tenth that size and is written in Tcl/Tk, which is available on most systems.

Let’s talk about some of my other tiny tools. Programmers often want to look at a numbered listing of a code fragment. There are a few ways to do this with Unix commands. For instance, you can use `pr` in a script:

```
pr –T –n S*
```

Here’s a similar query using a starting line number:

```
pr –T –n –N S* S3
```

Why not wrap this into a single command, called `num`, so that you don’t have to remember the names of all those options? Figure 1 shows `num` as a tiny tool, applied to itself.

Another tiny tool in my kit is `line`, which prints a specific line from a file, optionally with a few surrounding lines for context. It uses the `num` script. As is not unusual, about half the script is for error handling (see Figure 2).

The first two arguments to this script are a filename followed by a line number; an optional third argument can specify the number of lines you want to see before and after that line. This lets you say, for instance,

```
line line 9 1
```

Another even smaller tool I use a lot is `any`. It quickly finds all locations of a variable name or text string in the source files in the current directory. You can write a basic version of this tool with just a single `grep` command (see Figure 3).

---

**Figure 1.** The `num` command as a tiny tool, applied to itself. It lets you look at a numbered listing of a code fragment.

```
$ num num
1 #!/bin/sh
2
3 # num [nr] [file]*
4
5 if [ -f $1 ]
6 then
7 pr –T –n $*
8 else
9 N=$1
10 shift
11 pr –T –n –N $N $*
12 fi
```

**Figure 2.** The `line` tool prints a specific line from a file, optionally with a few surrounding lines for context. It uses the `num` script from Figure 1.

```
$ line line
1 #!/bin/sh
2
3 if [ $# -lt 2 -o $# -gt 3 ]
4 then echo "usage: line file linenr [nrlines]"
5 exit 1
6 fi
7
8 if [ ! -f $1 ]
9 then echo "error: no such file: $1"
10 exit 1
11 fi
12
13 n1=$2; n2=$2
14 if [ $# -eq 3 ]
15 then n1=`expr $n1 - $3`
16 n2=`expr $n2 + $3`
17 fi
18
19 sed -n ${n1},${n2}p $1 | num $n1
```
This version of the script, however, has a flaw. It works well for longer names that are relatively unique but fails miserably when you need to find all uses of, say, a single-letter variable i or j.

A Tokenizer for C
To solve problems such as this, I use ctok, a small standalone tokenizer for C that’s remarkably useful (see Figure 4). Instead of passing the lexical tokens that it recognizes in an input stream to a parser, ctok prints this information on the standard output.

The ctok code is about 600 lines of lex input, which compiles into a small standalone executable that has become a core part of my survival kit. Two types of output that ctok produces are tagged line and ident and are of interest in solving the variable-matching problem:

A simple grep for k over the same source files would be quite unhelpful. Using the C tokenizer, you can easily build additional tiny checkers, including more accurate checkers for the two examples I started with: finding switch statements without a default clause, or if-then-else-if chains that don’t end with an else.

The last two tiny tools I discuss here are really part of the same family. I use them to quickly find the definition of functions and data structures in C source files. It’s not hard to write more-sophisticated versions of these tiny shell scripts by using the tokenizer as a front end, but these basic versions already provide most of the needed functionality.

The first tool, ff (find function), finds and prints the definition of a function, optionally restricting the search to a specific file (see Figure 6). This script uses the fact that I always write the name of a function starting on a new line, right after the function type, which is also on a line by itself. The end of the definition is always a single closing curly brace in the left margin. If you use a different format, you’ll have to either adjust the script to match it or switch to a ctok-based version that can remain independent of such formatting choices.

Similarly, the ft tool finds data type or typedef definitions. Again, the simple version I describe here (see...
Figure 7) depends on the specific way I format these definitions. If you use a different format, you’ll have to adjust these scripts to match that.

The tiny scripts from my survival kit are the tools I use the most every day. Of course, none of this is rocket science, and if these tools don’t increase my productivity, at least they make the job of writing code a little more fun.

The last two scripts I described make assumptions about a particular coding style. A nice side effect of this is that it gives me an extra reason to be consistent in using that style. A consistent style generally improves readability, and you could well make the case that it’s also the first step toward improving code quality.

References

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.