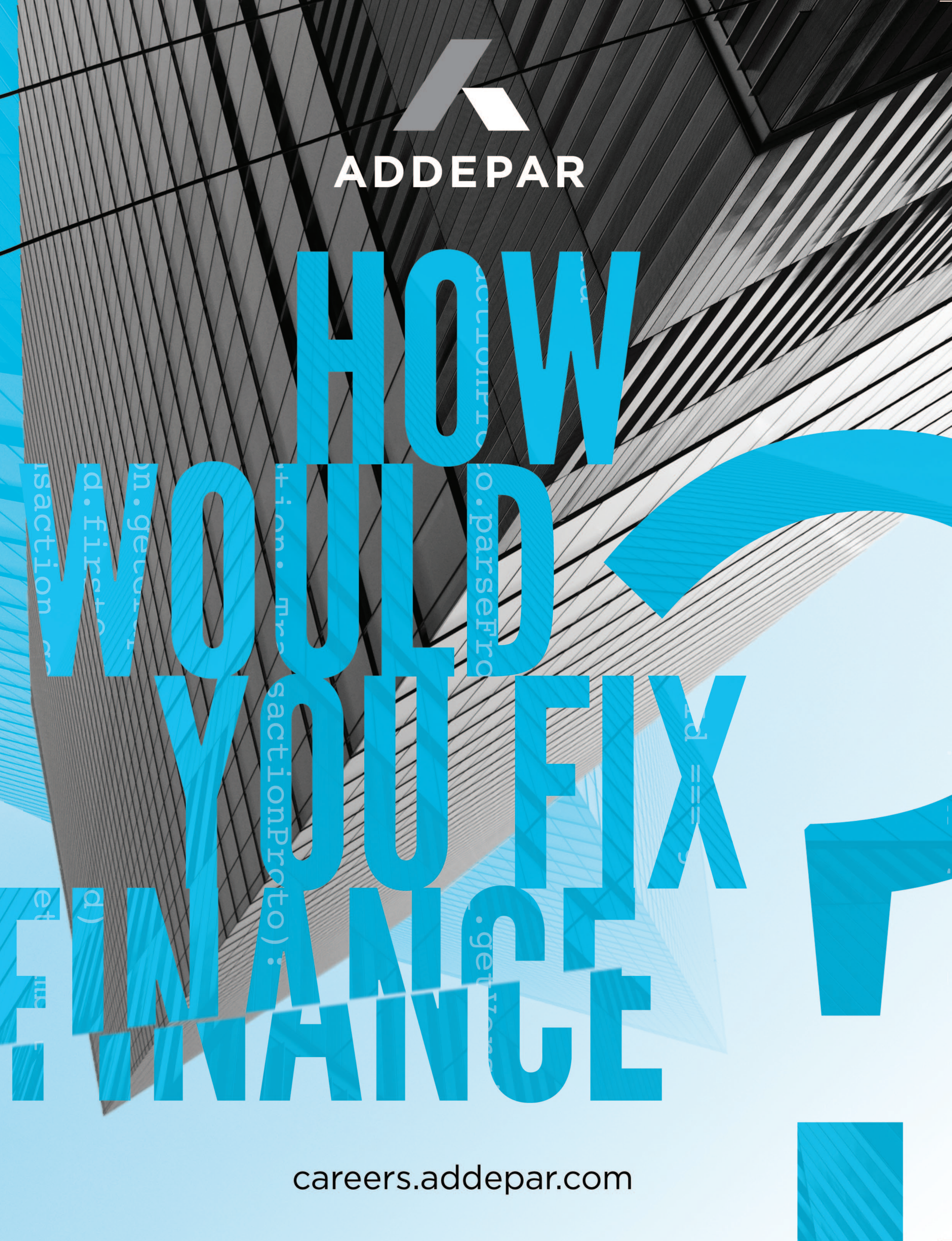
DANGER
DO NOT MESS WITH THIS MACHINE. IT HAS NO PITY.
THIS DEVICE CREATES POWERFUL ELECTRIC AND MAGNETIC FIELDS, BLINDINGLY BRIGHT LIGHT, OZONE AND OTHER TOXIC GASSES, LIGHTNING, X-RAYS, AND EXCESSIVE HEAT. ENJOY.
HOUDINI
CORONA

HACKERMONTHLY

Issue 29 October 2012


ADDEPAR
HOW
WOULD
YOU FIX
FINANCE
?
careers.addepar.com

MEET MANDRILL

By MailChimp

Mandrill is a new way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

MANDRILL.COM

Curator
Lim Cheng Soon

Contributors
Rob Flickenger
Peter Seibel
Vsevolod Dyomkin
Daniel Tenner
Dan Shipper
Tal Raviv
Jeff Preshing
Alex Young
Rob Pike
Mark Shroyer
David Woods

Proofreaders
Emily Griffin
Sigmarie Soto

Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

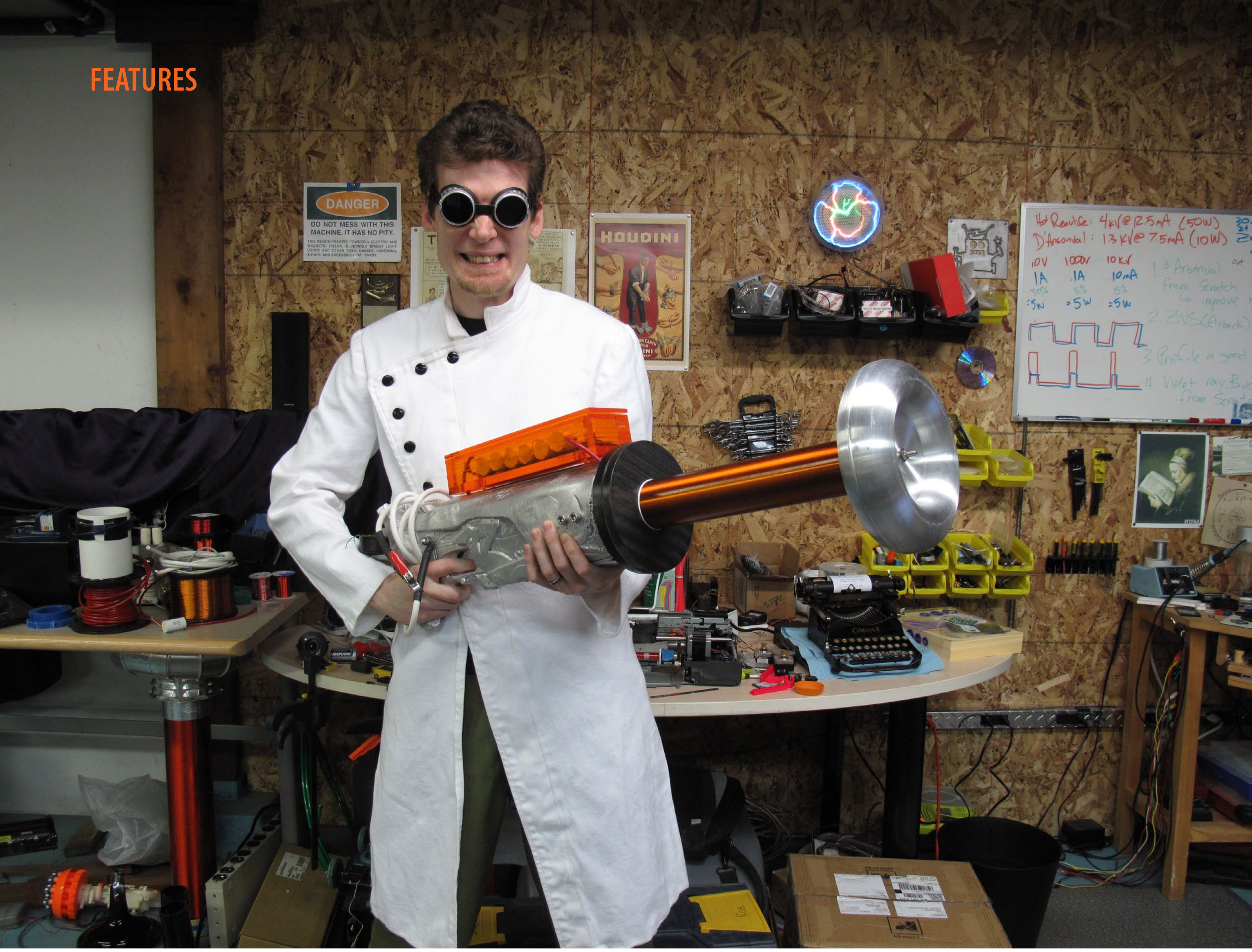
Cover Photo: Rob Flickenger

Contents

Photo by: Lily Huang

Photo by: Rob Flickenger

For links to Hacker News dicussions, visit *hackermonthly.com/issue-29*



The Tesla Gun

By ROB FLICKENGER

The year was 1889. The War of the Currents was well underway. At stake: the future of electrical power distribution on planet Earth. With the financial backing of George Westinghouse, Tesla's AC polyphase system competed for market dominance with Edison's established (but less efficient) DC system, in one of the ugliest and most epic tales of technological competition of the modern age.

More than a hundred years after the dust settled, Matt Fraction and Steven Sanders published The Five Fists of Science: a rollicking graphical retelling of what really happened at the turn of the last century. (Get yourself a copy [hn.my/ffos] and read it immediately, unless you're allergic to AWESOME.) On the right is the cover of this fantastic tale of electrical fury.

See that dapper fellow in front? That's a young Mr. Tesla. See what he's packin'?

Yep. Tesla Guns. Akimbo.

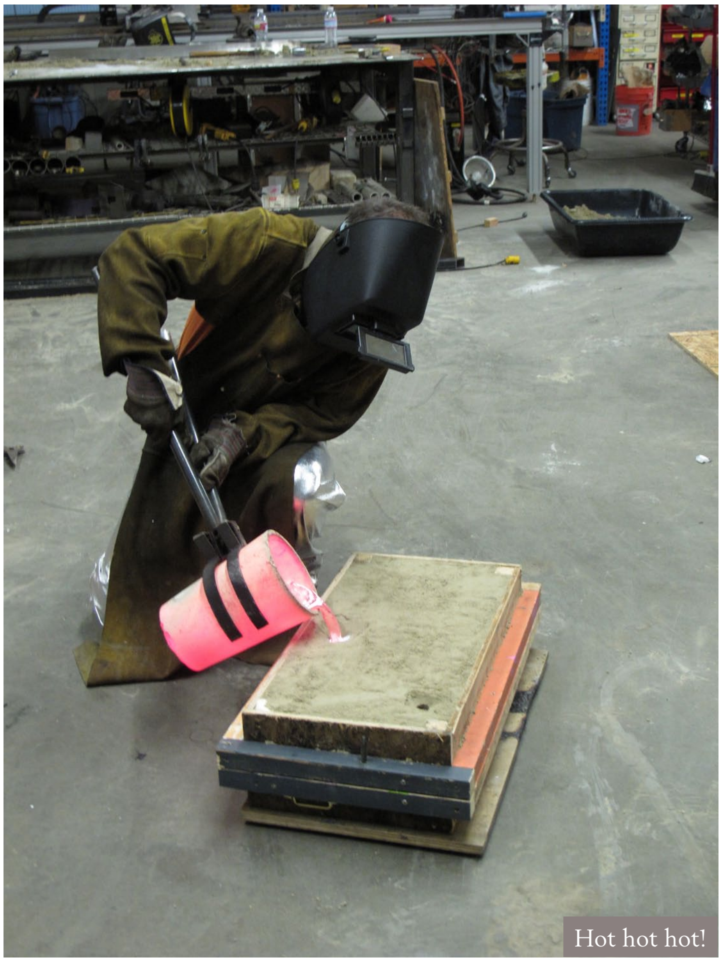
As I read this fantastic story, gentle reader, certain irrevocable processes were set in motion. The result is my answer to The Problem of Increasing Human Energy: The Tesla Gun. For reals.

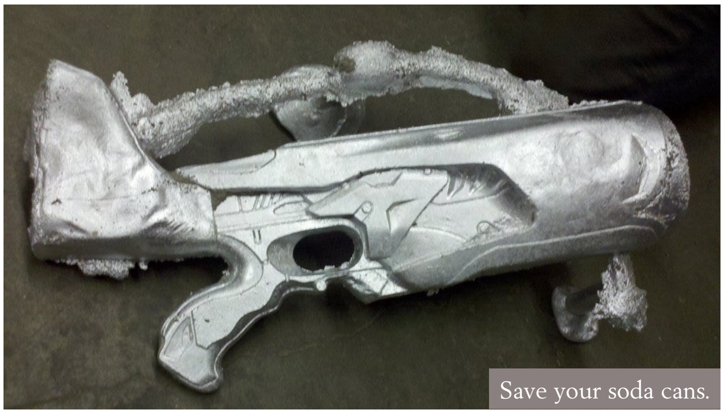
The Tesla Gun is a hand-held, battery powered lightning machine. It is a spark gap Tesla coil powered by an 18V drill battery. You pull the trigger, and lightning comes out the front.

It is functionally inferior to that of Tesla's design in the Five Fists in a few important respects. Notably, it is a bit longer and heavier than Tesla's own. It also cannot (yet) create an ion wind strong enough to cushion the user when leaping from a four story building.

On the other hand, my design is an improvement in two important respects: 1) It is battery powered, and 2) It actually exists.

I've given a few talks about how this project came to be, and it's a bit of a long story. I could not possibly have built it without the help and expertise of Seattle's many hackerspaces. Take a look at the basic components, and you'll see what I mean.


Hot hot hot!

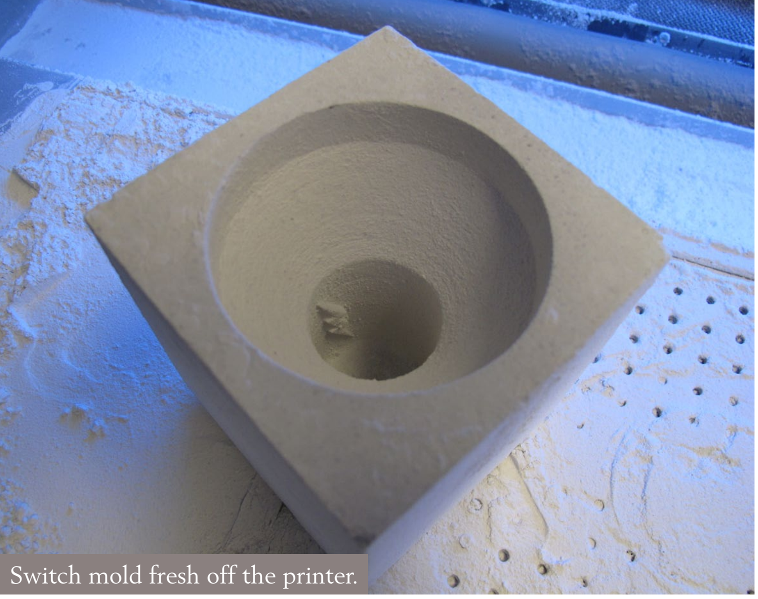

Save your soda cans.

The Housing

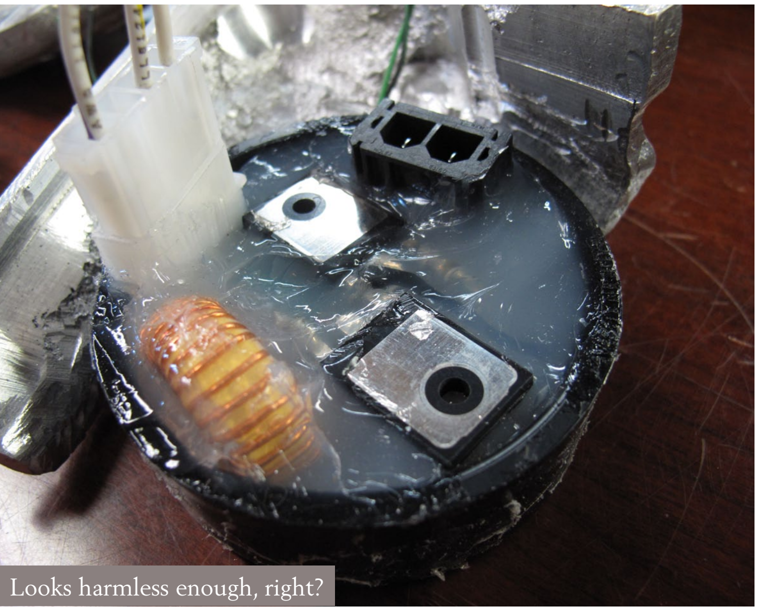
The housing is made from a nerf gun cast in aluminum. I had never made a metal casting before, so I went to the expert: Rusty from Hazard Factory. With his expert metal working skills and my limited ability to gather scrap aluminum, follow directions, and stay the hell out of the way, we had a pretty good aluminum housing in a couple of evenings.

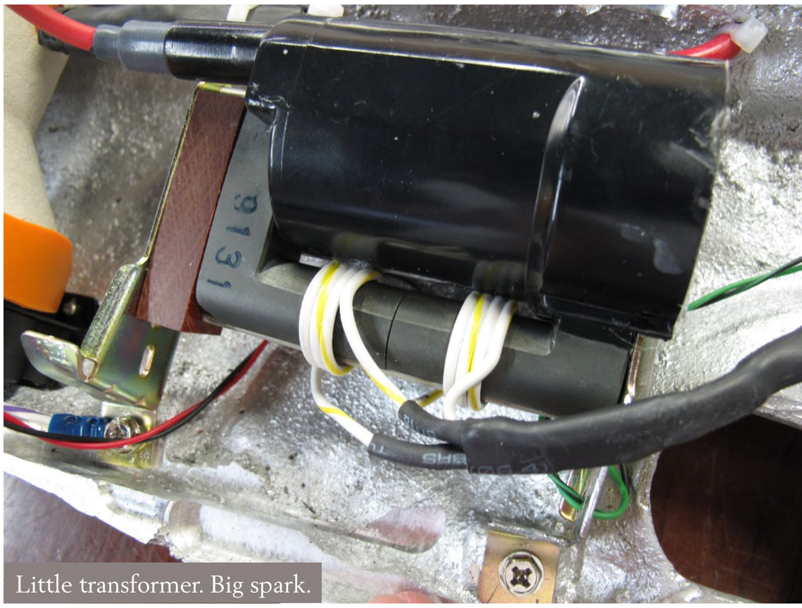
Sand casts inevitably have a few rough edges. Since I needed both halves of the housing to fit together perfectly, the next stop was Hackerbot Labs to put in some time on the Fadal 3-axis mill.

The milling process took a couple of days, but in the end I was able to remove a lot of the bulk of the interior aluminum, and the two halves lined up perfectly. With the housing finished, I set off on the next engineering challenge.


Switch mold fresh off the printer.

Radio Shack does not carry this switch.


Looks harmless enough, right?


Little transformer. Big spark.

The HV Switch

The heart of any spark gap Tesla coil is the high voltage switch. It needs to be able to withstand repeated switching events of many thousands of volts at an instantaneous current of a couple of thousand amperes, generating more than a little bit of heat along the way. This meant finding a material that was a good electrical insulator that was tough enough to withstand high temperatures. With the help of the fine folks at Metrix Create:Space, I decided to make my switch housing out of porcelain.

The first step required the use of a 3D powder printer. This kind of printer is perfect for printing molds for slip casting.

Once the mold was printed, I made a couple of castings using porcelain slip. After air drying for a couple of days, I fired them in the kiln at Metrix, let them cool for another day, and…Ta-da! A custom-sized HV switch housing, complete with little lightning bolts.

Then it was just a matter of inserting a couple of tungsten welding electrodes, and I had a fully functional high power switch. The shape was chosen to fit inside the aluminum housing while still providing room for a cooling turbine fan: a CPU cooler reclaimed from a discarded 1U server. This draws hot ions out of the switch, making for bigger and more rapid lightning.

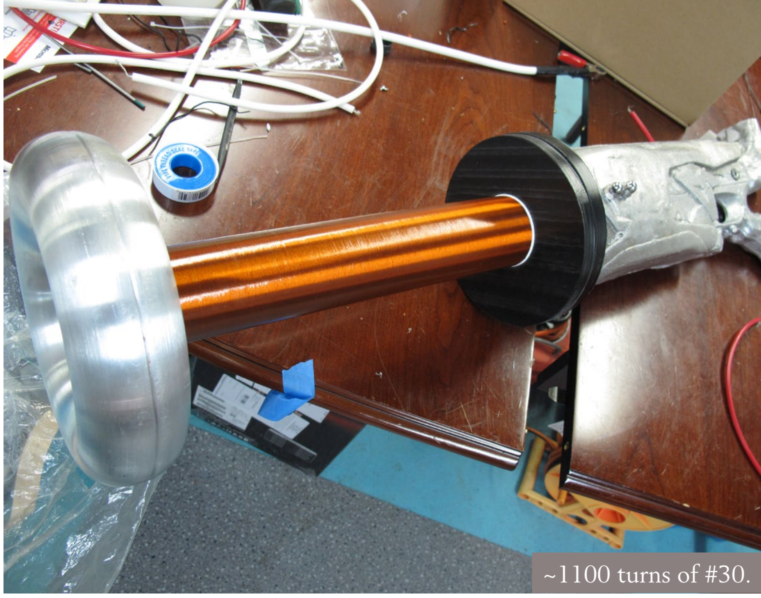
The Power Supply

Power is provided by an 18V lithium ion drill battery. That powers a ZVS driver circuit which drives a flyback transformer, stepping up that 18V to around 20,000V. This stage is affectionately known as the HOCKEY PUCK OF DOOM.

The circuit is small enough that it fits neatly in a 2.5" PVC plumbing end cap. It is potted with household-grade silicone (yes, Home Depot was an important supplier for this component). The output goes to a center tapped coil wrapped around the ferrite core of a flyback transformer salvaged from a TV.

That leads us to…

Stand well clear.


~1100 turns of #30.

HV wire. Red means DANGER.

Really, officer, it's just a movie prop! It couldn't possibly be as dangerous as it looks.

The Capacitor Bank

No, I didn't roll my own capacitors for this project. But I did make a nifty laser cut housing for them. Also, bleeder resistors are important for preventing unexpected surprises. Like waking up dead after touching this crazy toy.

The caps are 942C20P15K-F by Cornell Dubilier (the cap of choice when your current absolutely, positively needs to get there ON TIME). Since the housing is made of highly conductive aluminum, electrical connections are made with 40kV high voltage wire.

The Coils

All of that circuitry strobes the primary coil, protected by a couple of chunks of black HDPE (also milled on the Fadal).

The HDPE sandwich makes a great electrical insulator, helping to prevent arcs between the primary and secondary coils. The bottom of the secondary is also wound with PTFE tape (another great insulator, commonly found at Home Depot). The coil form is a piece of 2.5" ABS pipe wrapped in 30 gauge enameled wire, then sprayed with polyurethane finish (can you tell that the Home Depot is just a few minutes drive from my lair?).

The top load is an aluminum toroid purchased from Information Unlimited. Put it all together and there you have it: instant lightning at your trigger-happy fingertips.

Of course, the devil is in the details. How do you tune this beast? What about eddy currents in the housing? What do you use for an earth ground? Why is it so LOUD? How do you not die while operating it?

I'm afraid that this article has already gone on far too long. I'll explain a bit about those topics in future ones. Until then, stay safe and make AWESOME. ■

Rob Flickenger is a life-long hacker, tech writer, and aspiring mad scientist. His other inventions include a 15kJ coin shrinker and a camera array for capturing 3D photos of Tesla coil sparks.

Reprinted with permission of the original author.
First appeared in *hn.my/tesla* (hackerfriendly.com)

Lisp Hackers: Peter Seibel

Interviewed by VSEVOLOD DYOMKIN

With his Practical Common Lisp, Peter Seibel has helped more people (including me) discover and become users of Lisp as probably no one else has in the last decade. Dan Weinreb, one of the founders of Symbolics and later Chief Architect at ITA Software, a successful Lisp startup sold to Google for around $1B in 2011, wrote that their method of building a Lisp team was by hiring good developers and giving them PCL for two weeks, after which they could successfully integrate under the mentorship or their senior Lisp people.

A few years after PCL Peter went on to write another fantastic programming book Coders at Work.

Aside from being a writer, he was and remains a polyglot programmer, interested in various aspects of our trade, about which he blogs occasionally. His code, presented in PCL, laid the foundation for a wide-spread CL-FAD library, which deals with filenames and directories (as the name implies). More recently he created a Lisp documentation browser, Manifest. Before Lisp, Peter had worked a lot on Weblogic Java application server.

Tell us something interesting about yourself.
I'm a second generation Lisp programmer. My dad discovered Lisp when he was working at Merck in the 80s and ended up doing a big project to simulate a chemical plant in Lisp, taking over from some folks who had already been trying for quite a while using Fortran, and saving the day. Later he went to Bolt Beranek and Newman where he did more Lisp. So I grew up hearing about how great Lisp was and even getting to play around with some graphics programs on a Symbolics Lisp Machine.

I was also a childhood shareholder in Symbolics. I had a little money from some savings account that we had to close when we moved, so my parents decided I should try investing. I bought Symbolics because my parents just had. Never saw that money again. As a result, for most of my life I thought my parents were these naive, clueless investors. Later I discovered that around that time they had also invested in Microsoft which, needless to say, they did okay with.

Oh, and something I learned recently: not only was Donald Knuth one of the subjects in my book Coders at Work, but he has read the whole thing himself and liked it. That makes me happy.

What's your job? Tell us about your organization.
A few months ago I started working part-time at Etsy. Etsy is a giant online marketplace for people selling handmade and vintage items and also craft supplies. I'm in the data group where we try to find clever ways to use data to improve the website and the rest of the business.

Do you use Lisp at work? If yes, how you've made it happen? If not, why?
I always have a SLIME session going in Emacs for quick computations, and sometimes I prototype things in Lisp or write code to experiment with different ideas. However, these days I'm as likely to do those things in Python, because I can show my co-workers a sketch written in Python and expect them to understand it. I'm not sure I could do that with Lisp. But it makes me sad how slow CPython is compared to a native-compiling CL like SBCL. Usually that doesn't matter but it is annoying sometimes, mostly because Python has no real excuse. The rest of my work is in some unholy mishmash of Scala, Ruby, Javascript, and PHP.

What brought you to Lisp? What holds you?
As I mentioned, I grew up hearing from my dad about this great language. I actually spent a lot of my early career trying to understand why Lisp wasn't used more and exploring other languages pretty deeply to see how they were like and unlike Lisp. I played around with Lisp off and on until finally in 2003 I quit the startup I had been at for three years (which wasn't going anywhere) with a plan to take a year off and really learn Common Lisp. Instead I ended up taking two years off and writing Practical Common Lisp.

At this point I use it for things when it makes sense to do so, because I know it pretty well and most of my other language chops are kind of rusty. Though I'm sure my CL chops are rusty, too, compared to when I had just finished PCL.

Did you ever develop a theory why Lisp isn't used more?
Not one that is useful in the sense of helping it to be used more today. Mostly it seems to me to be the result of a series of historical accidents. You could argue that Lisp was too powerful too early and then got disrupted, in the Innovator's Dilemma sense, by various Worse is Better languages, running on systems that eventually became dominant for perhaps unrelated reasons.

Every Lisper should read The UNIX-HATERS Handbook to better understand the relation between the Lisp and Unix cultures. Lisp is the older culture, and back when the UNIX-HATERS Handbook was written, Unix machines were flaky and underpowered. They were held in the same contempt by Lisp geeks as Windows NT machines would be held by Unix geeks a few decades later. But for a variety of reasons people kept working on Unix and it got better.

And then it was in a better position than the Lisp culture to influence the way personal computing developed once micro computers arrived. While it would be a while before PCs were powerful enough to run a Unix-like OS, early on C was around to be adopted by PC programmers (including at Microsoft) once micros got powerful enough to not have to program everything in assembly. And from there, making things more Unix-like seemed like a good goal. Of course it would have been entirely possible to write a Lisp for even the earliest PCs that probably would have been as performant as the earliest Lisps running on IBM 704s and PDP-1s. My dad, back from his Lisp course at Symbolics, wrote a Lisp in BASIC on our original IBM PC. But by that point Lispers' idea of Lisp was what ran on powerful Lisp machines, not something that could have run on a PDP-1.

The AI boom and bust played its role as well. After the bust, Lisp's reputation was so tainted by its failure to deliver on the over-promises of the Lisp/AI companies that even many AI researchers disassociated themselves from it. And throughout the '90s various languages adopted some of Lisp's dynamic features, so folks who gravitated to that style of programming had somewhere else to go. Then when the web sprang into prominence, those languages were well positioned to become the glue of the Internet.

That all said, I'm heartened that Lisp continues to not only be used but to attract new programmers. I don't know if there will ever be a big Lisp revival that brings Lisp back into the mainstream. But even if there were, I'm pretty sure that there would be plenty of old-school Lispers who'd still be dissatisfied with how the revival turned out.

What's the most exciting use of Lisp you had?

I'm pretty proud of the tool chain I've built over the years while writing my two books and editing the magazine I tried to start, Code Quarterly. When I first started working on Practical Common Lisp I had some Perl scripts that I used to convert an ad-hoc light-weight text markup language into HTML. But after a little while of that I realized both that Jamie Zawinski was right about regexps and that of course I should be using Lisp if I was writing a book called Practical Common Lisp.

So I implemented a proper parser for a mostly-plain-text language that I uncreatively call Markup and backends that could generate HTML and PDF using cl-typesetting. When I was done writing and Apress wanted me to turn in Word files, I wrote an RTF backend so I could generate RTF files with all the Apress styles applied correctly. An Apress project manager later exclaimed over how "clean" the Word files I had turned had been. For editing Code Quarterly I continued to use Markup and wrote a prose diff tool that is pretty smart about when chunks of text get moved and edited a little bit.

What you dislike the most about Lisp?

I don't know if "dislike" is the right term because the alternative has its own drawbacks. But I do sometimes miss the security of refactoring with more static checks. For instance, when I programmed in Java, there was nothing better than the feeling of knowing a method was private and, therefore, I didn't have to look anywhere but in the one file where the method lived to see everywhere it could possibly be used. And in Common Lisp the possibilities for action at a distance are even worse than in some other dynamic languages because of the loose relation between symbols and the things they name. In practice that's not actually a huge problem and some implementations provide package locks and so on, but it always makes me feel a bit uneasy to know that if I :use a package and then DEFUN a function with the name of an inherited symbol, I've changed some code I really didn't mean to.

From time to time I imagine a language that lets you write constraints on your code in the language yourself — kind of like macros but instead of extending the syntax your compiler understands, they would allow you to extend the set of things you could say about your code that the compiler would then understand. So you could say things like, "this function can only be called from other functions in this file" but also anything else about the static structure of your code. I'm not sure exactly what the API for saying those things would look like, but I can imagine it being pretty useful, especially in larger projects with lots of programmers. You could establish certain rules about the overall structure of the system and have the compiler enforce them for you. But then if you want to do a big refactoring you could comment out various rules and move code around just like in a fully dynamic language. That's just a crazy idea; anyone who's crazy in the same way should feel free to take it and run with it and see if they get anywhere.

Among software projects you've participated in, what's your favorite?

Probably my favorite software I ever wrote was a genetic algorithm I wrote in the two weeks before I started at Weblogic in 1998, in order to build up my Java chops. It played Go and eventually got to the point where it could beat a random player on a 5x5 board pretty much 100% of the time. One of these days I need to rewrite that system in Common Lisp and see if I can work up to a full-size board and tougher opponents than random. (During evolution the critters played against each other to get a Red Queen effect — I just played them against a random player to see how they were doing.)

Describe your workflow, give some productivity tips to fellow programmers.

I'm not sure I'm so productive I should be giving anybody tips. When I'm writing new code I tend to work bottom up, building little bits that I can be confident in and then combining. This is obviously easy to do in a pretty informal way in Common Lisp. In other languages unit tests can be useful if you're writing a bigger system, though I'm often working on things for myself that are small enough I can get away with testing less formally. (I'm hopeful that something like Light Table will allow the ease of informal testing with the assurances of stricter testing — I'd love to have a development environment that keeps track of what tests go with what production code, shows them together, and runs the appropriate tests automatically when I change the code.)

When I'm trying to understand someone else's code I tend to find the best way is to refactor or even rewrite it. I start by just formatting it to be the way I like. Then I start changing names that seem unclear or poorly chosen. And then I start mucking with the structure. There's nothing I like better than discovering a big chunk of dead code I can delete and not have to worry about understanding. Usually when I'm done with that I not only have a piece of code that I think is much better but I also can understand the original. That actually happened recently when I took Edi Weitz's Hunchentoot web server and started stripping it down to create Toot (a basic web server) and Whistle (a more user friendly server built on top of Toot). In that case I also discarded the need for backward compatibility which allowed me to throw out lots of code. In that case I wasn't going for a "better" piece of code so much as one that met my specific needs better.

If you had all the time in the world for a Lisp project, what would it be?

I should really get back to hacking on Toot and Whistle. I tried to structure things so that all the Hunchentoot functionality could be put back in a layer built on top of Toot — perhaps I should do that just to test whether my theory was right. On the other hand, I went down this path because the whole Hunchentoot API was too hard for me to understand. So maybe I should be getting Toot and Whistle stable and well-documented enough that someone else can take on the task of providing a Hunchentoot compatibility layer.

I'd also like to play around with my Go playing critters, reimplementing them in Lisp where I could take advantage of having a to-machine-code compiler available at run time.

PCL was the book that opened the world of Lisp to me. I've also greatly enjoyed Coders at Work. So I'm looking forward for the next book you'd like to write. What would it be?

My current theory is that I'm going to write a book about statistics for programmers. Whenever I've tried to learn about statistics (which I've had to do, in earnest, for my new job), I find an impedance mismatch between the way I think and the way statisticians like to explain stuff. But I think if I was writing for programmers, then there are ways I could explain statistics that would be very clear to them at least. And I think there are lots of programmers who'd like to understand statistics better and may have had difficulties similar to mine. ■

Peter Seibel is a programmer and author of Practical Common Lisp and Coders At Work.

Vsevolod Dyomkin is a Lisp programmer from Kyiv, Ukraine. He works on Grammarly's core grammatical engine and overall architecture. He also teaches Operating Systems in Kyiv Politechnic.

Photo by: Lily Huang

How to Hack the Beliefs That Are Holding You Back

By DANIEL TENNER

We all have beliefs that are holding us back. Sometimes we're aware of them, sometimes not.

One entrepreneur I know, who shall remain nameless, admitted (after quite a lot of wine) that he has a block around sending invoices. He was perhaps exaggerating when he said that before he could send an invoice he had to down a bottle of wine and get drunk so he could hit the send button, but even so, it was clear that he had a serious block around asking people to pay him.

As an entrepreneur, that's obviously a deadly flaw. In terms of "holding you back," struggling to ask people for money for work that you've done is like wearing blocks of cement as boots. It won't just slow you down; it will probably stop you dead in your tracks.

I have — or used to have — similar blocks. Generally, many geeks early in their entrepreneurial career tend to have a general dislike of things like marketing and sales. These are things that, in my opinion, often are rooted not only in fear of an unknown activity, but also in beliefs about money. For example, I used to believe (subconsciously) that money was bad. I would spend money as quickly as (or more quickly than) I earned it. If your first thought when you're given £10,000 is how to spend it (rather than how it adds to your wealth), you probably have a similar belief that money is something to be gotten rid of, to push away. That's not a belief that's conducive to making money and becoming comfortably well off because you have to have a saving, wealth-building mindset for that.

Another would-be entrepreneur I spoke to recently was afraid to quit his job. He hated the work passionately. His wife supported his decision to quit, and he was fairly confident that he'd find something else (he had previously been a successful freelance developer). Yet, he couldn't bring himself to actually quit because he couldn't quite make the leap to believe in himself, even though he knew he should. Despite the evidence and arguments being stacked in favor of quitting, he felt he couldn't.

Now, perhaps the beliefs holding you back are of a different nature, but even if the "money thing" or the "quitting thing" doesn't apply to you, don't disregard this article. Chances are there are other beliefs rooted deep inside you that are holding you back, even if they have nothing to do with money.

So, if you're aware of such a belief and want to "fix" it, what can you do to hack your brain?

Having gone through the process, I am sharing a handful of techniques I've found that really help in a tangible way.

1 Self-affirmations

This feels really cheesy and weird when you start doing it, but it's probably the most effective on the list. Many of the beliefs that we might want to get rid of manifest themselves as "internal monologue." They're things that your subconscious is telling your conscious throughout the day.

“The main thing holding you back from achieving what you want is often yourself. These tools give you a means to fix that.”

For example, some people have an internal monologue that constantly repeats “you’re a failure” to them. By repeating it over and over again, the message becomes true. Some people precondition themselves to fail. They draw the failure to them by accepting this message over and over during the day.

Self-affirmations hack around this by overriding the negative message with a positive one. The way that it’s worked for me is:

1. Craft a brief, positive message (phrase it in positive terms) that overrides the internal message that’s bothering you. For example, if “you’re a failure” is the message that’s bothering you, a positive override might be “I will succeed in many things that make a difference.” It doesn’t need to be exactly true, but it needs to be something you can stand by and believe in, however briefly.

2. Write this message on a post-it note or a piece of cardboard, and stick it on your mirror — the one that you dress yourself in front of every morning.

3. Every morning (and as many times during the day as you can), stand in front of your mirror and, looking yourself straight in the eyes, repeat, loudly, with all the confidence you can muster in your voice, “I will succeed in many things that make a difference” (or whatever the affirmation is). Repeat it 10 times. Repeat it 50 times. However many times you can.

Three things will happen from this. First, you will feel very silly. That’s ok, don’t worry about it. It won’t pass (you’ll still feel silly the 20th time you do this), but it really doesn’t matter. Secondly, you’ll feel a good buzz. I haven’t quite figured out why that happens. I guess it’s a sense that you’re taking things into your own hands, taking action. That feels good.

Most importantly, over time (surprisingly quickly), the internal message in your head will change. As it changes, you will feel the need for the affirmations lessen. Obviously, if the message you’re overriding is deeply ingrained, it will take longer, but for me, typically, I haven’t needed to do this for more than a few weeks before the new message had sunk in.

This is an extremely effective method. You can also do variants of this, like recording a video or audio for yourself, or writing it out by hand fifty times, but in my experience, speaking to yourself while looking into your own eyes is brutally effective.

2 Brainwashing yourself

When you read stuff and you don’t take notes, you’re effectively just brainwashing yourself. Most people read whatever comes their way or whatever they feel like without really considering selection, but you can choose what you brainwash yourself with.

If you know that you have, for example, a problem with pushing away money, then there are books that repeat the opposite message over and over again. If you spend a few weeks reading a bunch of those books, chances are you'll come out the other end with an altered outlook. In my experience, it doesn't stick as much as self-affirmation, so if you do this you'll probably want to find a steady source of relevant books so you can keep re-brainwashing yourself until it really sticks.

You don't have to stick to books. Videos, podcasts, blogs, or even meetups can achieve the same thing. The key is to keep exposing yourself to information that contradicts the belief you're trying to get rid of.

Of course, you can use this in conjunction with self-affirmation to enhance the effect.

3 Who you hang out with

Another strong influence on your internal message is, sadly, who you hang out with. People have certain expectations and perceptions of you, and it's very hard to shake them off if they are one of the sources of the negative messages you're struggling with.

Obviously, if your parents or your friends constantly tell you you're a failure, that's going to work just as well as positive self-affirmations in convincing you that you are indeed a failure. If they expect you to fail, and you spend a lot of time with them, you will probably fail.

This is a tricky one, since these sources of negative influence are often not deliberate. Your parents or friends probably don't want you to fail, and if confronted, they'll almost certainly agree to change their ways — but they won't. Changing habits is very, very hard, and if people have got into the habit of perceiving you in a certain way, the change of perception has to come from you.

Sadly, I think the only thing that can be done in this case is to spend less time with people who project their negative perceptions on you, at least until you've properly dealt with the negative message so that it's no longer holding you back. But even then, be aware that exposing yourself to that external, repeated message again could bring it back.

4 Digging to the root

Finally, one last technique which also helps, especially when combined with all the others, is to truly examine your beliefs, and figure out where they come from, how they grew in you over time, what role they've played in your life, etc.

Now, I'm fully aware that our memory of these sorts of things is often very hazy, and most likely the "explanation" or "history" that you come up with will be, in many ways, a fabrication. But despite that, this somehow still works.

For example, through this type of introspection, I realized that my lack of interest in accumulating money was something that had been with me since childhood. It was something that had been encouraged by my parents, and that was one of the components of why I'm generally a "happy person." Through this insight, I also realized that one of the reasons why I found it hard to bring myself to care about money was that I associated caring about money, and accumulating it, with unhappiness. The belief there was not so much that "money is bad," but that "people who care about making money are unhappy, sharks, obsessive people who live empty lives."

Once I discovered this reasoning in my subconscious, I was able to target it directly with self-affirmations like "I want to make more money so that I can do more good," which replace the link between money and unhappiness with one between money and the capacity to do good.

Disclaimer

These techniques may not work at all for you, or you may think that they're hocus pocus. However, they worked for me and have helped me. I've discussed them with enough people to come to the conclusion that many people don't know or haven't thought about these types of tools, and most people are not using them. Some of these techniques (e.g. self-affirmations) are standard tools that therapists use to help people, so there's some validation for these things working in a wide range of cases.

The main thing holding you back from achieving what you want is often yourself. These tools give you a means to fix that. If they don't work for you, you won't have lost anything, except perhaps for the terrible experience of feeling mildly silly while talking to yourself in front of a mirror.

If they do work, then you can gain a lot. Specifically, you can give yourself the ability to achieve what you want in life. That's pretty valuable, I reckon.

Good luck with it all! ■

Daniel Tenner is the founder of Woobius and GrantTree. Known as "swombat" on Hacker News and Twitter, he is now producing *swombat.com*, a daily updated resource for people who like to read startup articles.

Reprinted with permission of the original author.
First appeared in *hn.my/belief* (swombat.com)

B2B Is Unsexy, and I Know It

By DAN SHIPPER

When I tell people I do B2B software I get some very interesting reactions.

"Why do B2B? It's so unsexy."

And that's true. B2B is unsexy in that I don't build things that my college friends want to use. But that doesn't mean it's unsatisfying or somehow inherently less valuable than a social/consumer product. In fact, I'd argue that the opposite is true. Spending every day making someone's life easier is awesome, especially when that someone actually wants to pay you for it.

So here are a few reasons why I do B2B:

Nobody ever went out of business making a profit

If you truly solve a business's problems they'll want to pay you for it. If you solve a consumer's problems, in many cases, they need to be dragged kicking and screaming to open their wallets. Writing B2B software makes it easier to make money from day one. That means that it's much more likely to generate a sustainable revenue stream than a social product that requires massive scale.

You don't need to win the lottery to succeed

The kind of scale required to generate a real return from a social product is pretty staggering. And certainly skill, experience and an understanding of social dynamics plays a large part in a company's ability to reach scale with a social product. But as far as I can tell, luck also plays a large part in creating something viral and sticky enough to succeed.

When we built WhereMyFriends.Be we had some idea that it would be a cool product, but the real reason it blew up probably had little to do with our incredible entrepreneurial foresight. We got lucky enough to hit on a small product that resonated with people, and a Mashable writer happened to like the sound of it.

We've had about 50,000 signups so far, but other than that we have very little to show for it except a sizeable hosting bill.

B2B requires no voodoo or midnight incantations

Chris Dixon and others have commented that B2B entrepreneurs seem to be much more likely to string together successful companies than other types of founders. I think that's because there's a lot less voodoo involved in creating a successful B2B software business than a social one.

Like everything else, it's hard as hell. But it's a problem that you can get your arms around and pin down. If you only need 10, 100 or 1000 customers to generate a small profit, it makes things a lot easier than needing 1 million.

"Are you making something that solves a problem for a business?"

"How do you sell it to them in a scalable way?"

"Who's making the buying decision on this problem within the organizations we're trying to target? Is it the same person who's experiencing pain?"

"How long does the sales process take?"

Those are some questions you get to ask yourself when you're building software for businesses. When you're building a social product, it's a little less clear how to proceed. Most people I know end up building their product and hoping to get covered in Techcrunch or Mashable so they can go viral.

As my dad would say: hope is not a plan.

The biggest opportunities probably aren't in social anymore

There are only so many different types of location-based, photo-sharing apps that can be built. Certainly, the unprecedented amount of data being generated by social products brings with it huge opportunities for future businesses, but the vanilla "share more easily with your friends" social model seems to be rather played out.

None of this is to say that building social products is inherently a bad idea or that social products aren't valuable. It's just a small explanation for why, as a college-age entrepreneur, I've chosen to go down a different route. ■

Dan Shipper is a student, blogger and entrepreneur. Dan has been programming for 10 years, and he's currently working on Firefly and Airtime for Email.

Being a Developer Makes You Valuable. Learning How to Market Makes You Dangerous

By TAL RAVIV

I LOVE ENGINEERING, AND not just because I'm a nerd.

The best part of engineering isn't the technical details or the particular science behind it, rather, it's the opportunity to solve an unfairly hard problem in a way no one has before. The harder the problem the more exciting it is. As a chemical-turned-software engineer, I can say the thrill is the same.

In business and marketing there's a word for that kind of person: "hustler" — or, in the software startup space, "growth hacker."

As much as engineers like to joke about our counterparts in sales and marketing, the most successful salespeople and marketers think like engineers. They do enormous amounts of research, are systematic and methodical, apply known facts and patterns, and make approximations when necessary. They measure results objectively, and they iterate. (They are admittedly rare, and it's those who don't fit this description that earn derision.)

I got an email from a student who reached out via our "breaking every rule" page. The developer, Wasswa Samuel, in his final year of computer science in Uganda, is clearly very passionate and full of energy to work on something awesome.

He described his previous entrepreneurial experience:

I started a small startup which unfortunately has refused to take off. I am guessing the idea wasn't all that awesome or it will pick up after a year, whatever. I have left the site around but am not actively working on it.

I checked out Wasswa's site. The dude's got energy, skills, appreciates good UX, and there's definitely a business there. Maybe all that's missing is some hustling.

I proposed to Wasswa that his Ugandan deals site could go from being a technical project, to a marketing project of his. It could be a chance to experiment and learn about all the different kinds of online and offline marketing and solve the "taking off" problem.

After exchanging some links for getting started, Wasswa sent me this:

Thanks for all this great content. Am loving it. I never knew there was all this amazing stuff.

That's when I realized: it's not just that developers don't see themselves as potentially amazing marketers. They might not even realize how deep and interesting of a field marketing is.

And developers who can also hack their way to growth…those guys are dangerous.

Becoming Dangerous

If you don't work closely with amazing marketers, it's hard to know where to start or what the scope of the field is. (Like learning to code, but backwards.)

The most important thing to know is: trust me, if you are smart enough to build stuff, you can crack this. To paraphrase Paul Graham's premise in founding YCombinator, "It's easier to teach an engineer business than it is to teach a business person engineering."

I bet you didn't learn coding from reading a curriculum or a list of links. You found a starting point and let your curiosity take you from there. So, here are some starting places to whet your appetite, starting with two dangerous engineers.

Patrick McKenzie's systematic, hard-working approach to letting Google do your marketing for free: *hn.my/gmark*. This is an amazing interview by Gabriel Weinberg, probably the case study for this article himself.

Gabe is working on an incredible traction book [tractionbook.com] compiling all of his interviews of other developers and non-developers and how they acquired their first 1k, 10k, 100k users (or dollars). He asks the questions you'd wish you could ask his guests.

“As much as engineers like to joke about our counterparts in sales and marketing, the most successful salespeople and marketers think like engineers.”

Get on the Mixergy list serve [mixergy.com]. Not only do they have the best subject lines your inbox has ever seen, but Andrew approaches every interview just like Gabe: he's not there to do a talk show interview. He's there to extract the specific tactics and figure out what these hustlers do at each challenge.

As you go through these resources, beyond listening to what they're saying, observe what they're doing; how Neville and Andrew and Gabe got their audiences (in three very different ways), how often do they post, how people seem to find them, how active they are in the comments, the calls to action, tone…an infinite amount of calculated (and uncalculated) actions that make them good at building audiences.

Engineers know the importance of benchmarks and “maximum theoretical” success. Fortunately, people like Rob Fitz will even share their notes [hn.my/fitz] with you so you can see what goes on behind the scenes and make concrete assumptions. Even early startups, like this one for personal funding [hn.my/gtstats], are sharing their metrics like they never have before.

There are stories of non-digital pure hustle. [hn.my/phustle]

Or pure digital. [hn.my/dhustle]

Both are highly recommended stories. The second link from Rand Fishkin's talk to Hackers and Founders is a long video. I used to see these as an hour lost. I now see them as an hour of free tuition for a topic that will probably help me more than any one hour I spent in college.

Paul, Toan, and I wrote a guide on how to get to your first 1,000 customers [hn.my/first1000] for StartupPlays. Unlike the above resources it costs money but that was the deal we made in exchange for distribution. StartupPlays, however, is an extremely valuable resource (especially Dan Martell's play [hn.my/danplay]) for a comparatively tiny price.

Don't forget Quora. There's some great stuff on growth hacking. [quora.com/growth-hacks]

Like engineering, the key is not to know everything, but rather to know where to look when you need to. Developers are in the best position to succeed; they have the hard skills and everything else is learnable. ■

Tal is the Co-Founder at Ecquire. He has constructed mobile hardware at the MIT Media Lab, designed medical imaging software for the Penn School of Medicine, developed computer simulations of biofuel processes, and created mobile applications for BlackBerry, iPhone, and Android. Tal holds a Guinness Record for the World's Largest Ball of Tape.

Reprinted with permission of the original author.
First appeared in *hn.my/danger* (talsraviv.com)

An Introduction to Lock-Free Programming

By JEFF PRESHING

Lock-free programming is a challenge, not just because of the complexity of the task itself, but because of how difficult it can be to penetrate the subject in the first place.

I was fortunate in that my first introduction to lock-free (also known as lockless) programming was Bruce Dawson's excellent and comprehensive white paper, Lockless Programming Considerations. [hn.my/lockless] And like many, I've had the occasion to put Bruce's advice into practice while developing and debugging lock-free code on platforms such as the Xbox 360.

Since then, a lot of good material has been written, ranging from abstract theory and proofs of correctness to practical examples and hardware details. I'll leave a list of references in the footnotes. At times, the information in one source may appear orthogonal to other sources. For instance, some material assumes sequential consistency, and thus sidesteps the memory ordering issues that typically plague lock-free C/C++ code. The new C++11 atomic library standard throws another wrench into the works, challenging the way many of us express lock-free algorithms.

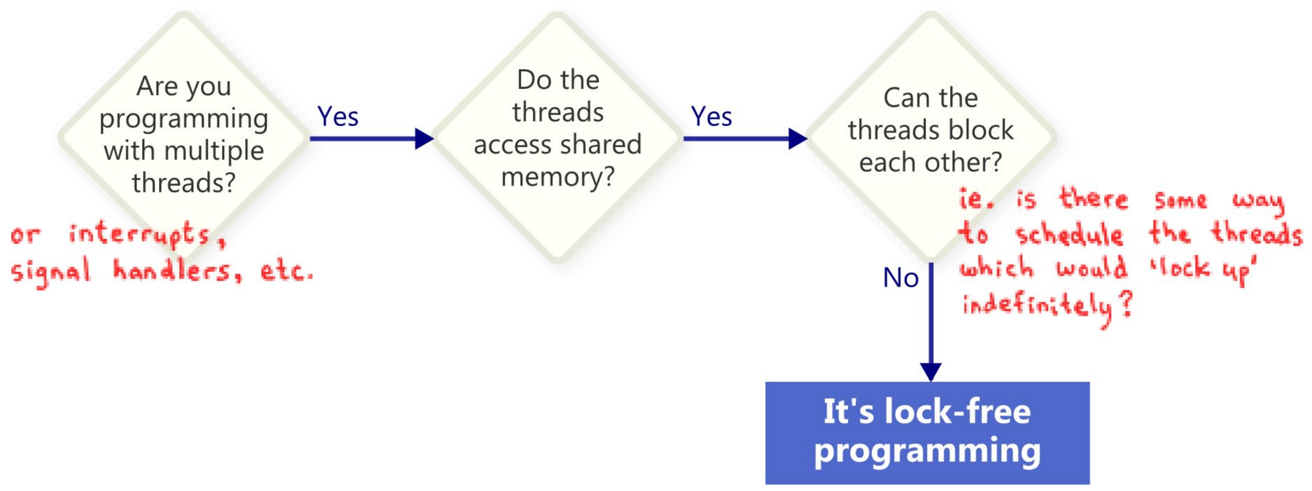
In this article, I'd like to re-introduce lock-free programming, first by defining it and then by distilling most of the information down to a few key concepts. I'll show how those concepts relate to one another using flowcharts, and then we'll dip our toes into the details a little bit. At a minimum, any programmer who dives into lock-free programming should already understand how to write correct multithreaded code using mutexes and other high-level synchronization objects such as semaphores and events.

What Is It?

People often describe lock-free programming as programming without mutexes, which are also referred to as locks. That's true, but it's only part of the story. The generally accepted definition, based on academic literature, is a bit broader. At its essence, lock-free is a property used to describe some code, without saying too much about how that code was actually written.

Basically, if some part of your program satisfies the following conditions, then that part can rightfully be considered lock-free. Conversely, if a given part of your code doesn't satisfy these conditions, then that part is not lock-free.

In this sense, the lock in lock-free does not refer directly to mutexes, but rather to the possibility of "locking up" the entire application in some way, whether it's deadlock, livelock, or even due to hypothetical thread scheduling decisions made by your worst enemy. That last point sounds funny, but it's key. Shared mutexes are ruled out trivially because as


Are you programming with multiple threads?
or interrupts, signal handlers, etc.
Yes
Do the threads access shared memory?
Yes
Can the threads block each other?
ie. is there some way to schedule the threads which would 'lock up' indefinitely?
No
It's lock-free programming

soon as one thread obtains the mutex, your worst enemy could simply never schedule that thread again. Of course, real operating systems don't work that way — we're merely defining terms.

Here's a simple example of an operation that contains no mutexes but is still not lock-free. Initially, X = 0. As an exercise for the reader, consider how two threads could be scheduled in a way that neither thread exits the loop.

```
while (X == 0)
{
    X = 1 - X;
}
```

Nobody expects a large application to be entirely lock-free. Typically, we identify a specific set of lock-free operations out of the whole codebase. For example, in a lock-free queue, there might be a handful of lock-free operations such as `push`, `pop`, perhaps `isEmpty`, and so on.

Herlihy & Shavit, authors of The Art of Multiprocessor Programming [hn.my/multipro], tend to express such operations as class methods and offer the following succinct definition of lock-free: "In an infinite execution, infinitely often some method call finishes." In other words, as long as the program is able to keep calling those lock-free operations, the number of completed calls keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations.

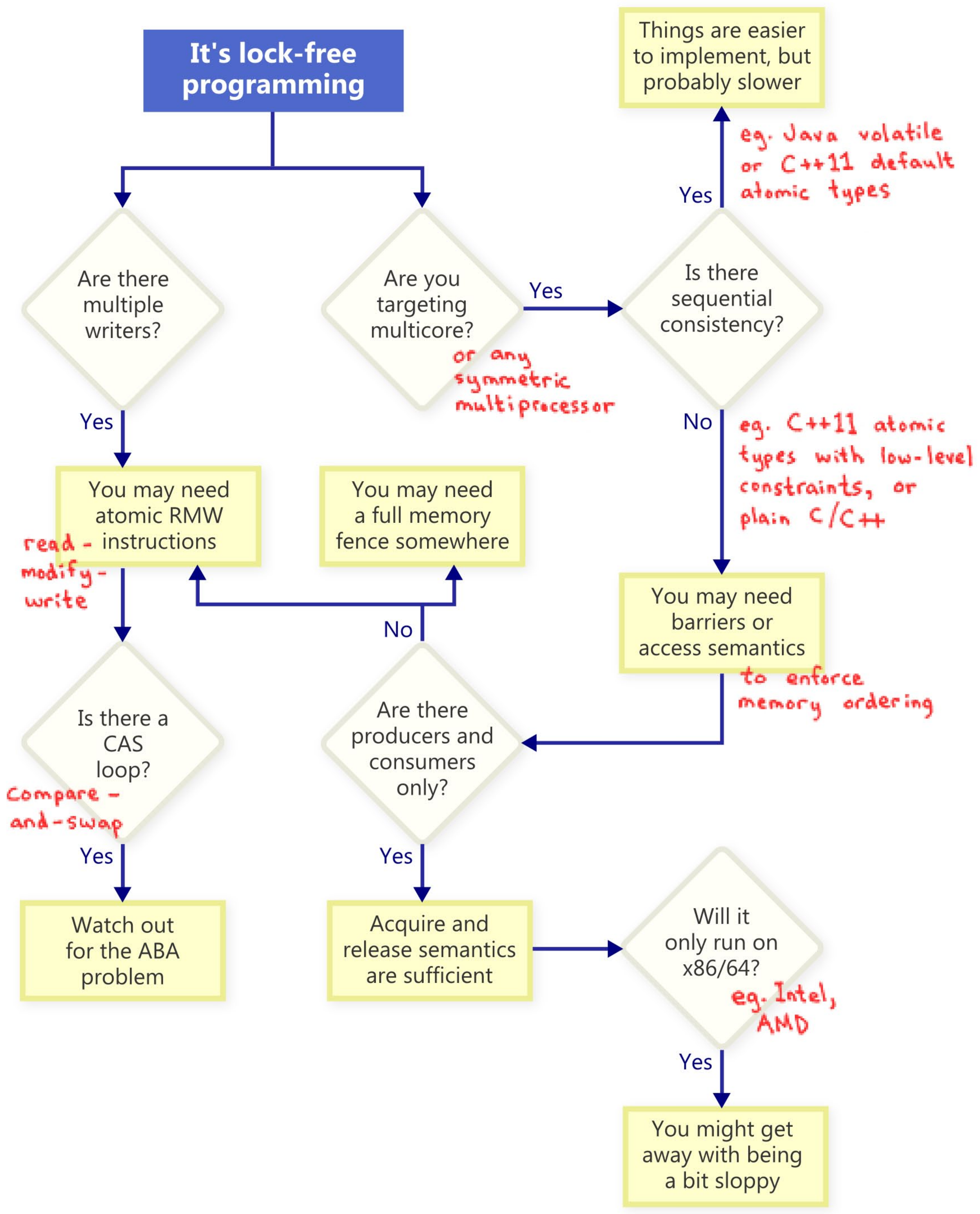
One important consequence of lock-free programming is that if you suspend a single thread, it will never prevent other threads from making progress, as a group, through their own lock-free operations. This hints at the value of lock-free programming when writing interrupt handlers and real-time systems, where certain tasks must complete within a certain time limit, no matter what state the rest of the program is in.

A final precision: Operations that are designed to block do not disqualify the algorithm. For example, a queue's pop operation may intentionally block when the queue is empty. The remaining codepaths can still be considered lock-free.

Lock-Free Programming Techniques

It turns out that when you attempt to satisfy the non-blocking condition of lock-free programming, a whole family of techniques falls out: atomic operations, memory barriers, and avoiding the ABA problem, to name a few. This is where things quickly become diabolical.

So how do these techniques relate to one another? To illustrate, I've put together the following flowchart. I'll elaborate on each one next.



It's lock-free programming
Are there multiple writers?
Yes
You may need atomic RMW instructions
read - modify - write
Is there a CAS loop?
compare - and - swap
Yes
Watch out for the ABA problem
Are you targeting multicore?
or any symmetric multiprocessor
Yes
Is there sequential consistency?
Yes
eg. Java volatile or C++11 default atomic types
Things are easier to implement, but probably slower
No
eg. C++11 atomic types with low-level constraints, or plain C/C++
You may need barriers or access semantics
to enforce memory ordering
Are there producers and consumers only?
No
You may need a full memory fence somewhere
Yes
Acquire and release semantics are sufficient
Will it only run on x86/64?
eg. Intel, AMD
Yes
You might get away with being a bit sloppy

Atomic Read-Modify-Write Operations

Atomic operations manipulate memory in a way that appears indivisible: No thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic.

Read-modify-write (RMW) operations go a step further, allowing you to perform more complex transactions atomically. They're especially useful when a lock-free algorithm must support multiple writers because when multiple threads attempt an RMW on the same address, they'll effectively line up in a row and execute those operations one at a time. I've already touched upon RMW operations in this blog, such as when implementing a lightweight mutex, a recursive mutex, and a lightweight logging system.

Examples of RMW operations include `_InterlockedIncrement` on Win32, `OSAtomicAdd32` on iOS, and `std::atomic<int>::fetch_add` in C++11. Be aware that the C++11 atomic standard does not guarantee that the implementation will be lock-free on every platform, so it's best to know the capabilities of your platform and toolchain. You can call `std::atomic<>::is_lock_free` to make sure.

Different CPU families support RMW in different ways. Processors such as PowerPC and ARM expose load-link/store-conditional instructions, which effectively allow you to implement your own RMW primitive at a low level, though this is not often done. The common RMW operations are usually sufficient.

As illustrated by the flowchart, atomic RMWs are a necessary part of lock-free programming even on single-processor systems. Without atomicity, a thread could be interrupted halfway through the transaction, possibly leading to an inconsistent state.

Compare-And-Swap Loops

Perhaps the most often-discussed RMW operation is compare-and-swap (CAS). On Win32, CAS is provided via a family of intrinsics such as `_InterlockedCompareExchange`. Often, programmers perform compare-and-swap in a loop to repeatedly attempt a transaction. This pattern typically involves copying a shared variable to a local variable, performing some speculative work, and attempting to publish the changes using CAS:

```
void LockFreeQueue::push(Node* newHead)
{
    for (;;)
    {
        // Copy a shared variable (m_Head) to a
        // local.
        Node* oldHead = m_Head;

        // Do some speculative work, not yet
        // visible to other threads.
        newHead->next = oldHead;

        // Next, attempt to publish our changes to
        // the shared variable.
        // If the shared variable hasn't changed,
        // the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head,
newHead, oldHead) == oldHead)
            return;
    }
}
```

Such loops still qualify as lock-free because if the test fails for one thread, it means it must have succeeded for another. Some architectures, however, offer a weaker variant of CAS where that's not necessarily true. When implementing a CAS loop, special care must be taken to avoid the ABA problem.

Sequential Consistency

Sequential consistency means that all threads agree on the order in which memory operations occurred, and that order is consistent with the order of operations in the program source code. Under sequential consistency, it's impossible to experience memory reordering shenanigans like the one I demonstrated in a previous post.

A simple (but obviously impractical) way to achieve sequential consistency is to disable compiler optimizations and force all your threads to run on a single processor. A processor never sees its own memory effects out of order, even when threads are pre-empted and scheduled at arbitrary times.

Some programming languages offer sequential consistency even for optimized code running in a multiprocessor environment. In C++11, you can declare all shared variables as C++11 atomic types with default memory ordering constraints. In Java, you can mark all shared variables as `volatile`. Here's the example from my previous post, rewritten in C++11 style:

```
std::atomic<int> X(0), Y(0);
int r1, r2;

void thread1()
{
    X.store(1);
    r1 = Y.load();
}

void thread2()
{
    Y.store(1);
    r2 = X.load();
}
```

Because the C++11 atomic types guarantee sequential consistency, the outcome r1 = r2 = 0 is impossible. To achieve this, the compiler outputs additional instructions behind the scenes — typically memory fences and/or RMW operations. Those additional instructions may make the implementation less efficient compared to one where the programmer has dealt with memory ordering directly.

Memory Ordering

As the flowchart suggests, any time you do lock-free programming for multicore (or any symmetric multiprocessor), and your environment does not guarantee sequential consistency, you must consider how to prevent memory reordering.

On today's architectures, the tools to enforce correct memory ordering generally fall into three categories, which prevent both compiler reordering and processor reordering:

- A lightweight sync or fence instruction, which I'll talk about in future posts.
- A full memory fence instruction, which I've demonstrated previously.
- Memory operations that provide acquire or release semantics.

Acquire semantics prevent memory reordering of operations which follow it in program order, and release semantics prevent memory reordering of operations preceding it. These semantics are particularly suitable in cases when there's a producer/consumer relationship, where one thread publishes some information and the other reads it.

Different Processors Have Different Memory Models

Different CPU families have different habits when it comes to memory reordering. The rules are documented by each CPU vendor and followed strictly by the hardware. For instance, PowerPC and ARM processors can change the order of memory stores relative to the instructions themselves, but normally, the x86/64 family of processors from Intel and AMD do not. We say the former processors have a more relaxed memory model.

There's a temptation to abstract away such platform-specific details, especially with C++11 offering us a standard way to write portable lock-free code. But currently, I think most lock-free programmers have at least some appreciation of platform differences. If there's one key difference to remember, it's that at the x86/64 instruction level, every load from memory comes with acquire semantics, and every store to memory provides release semantics — at least for non-SSE instructions and non-write-combined memory. As a result, it's been common in the past to write lock-free code which works on x86/64 but fails on other processors.

If you're interested in the hardware details of how and why processors perform memory reordering, I'd recommend Appendix C of Is Parallel Programming Hard [hn.my/perf]. In any case, keep in mind that memory reordering can also occur due to compiler reordering of instructions.

In this article, I haven't said much about the practical side of lock-free programming, such as: When do we do it? How much do we really need? I also haven't mentioned the importance of validating your lock-free algorithms. Nonetheless, I hope that for some readers, this introduction has provided a basic familiarity of lock-free concepts so you can proceed into the additional reading without feeling too bewildered. ■

Jeff Preshing is a video game developer in Montreal, Canada. He thinks lock-free programming will always play a role in software development, making it worth trying to stop messing up. His favorite muppet is Fozzie.

Backbone.js: Hacker's Guide

By ALEX YOUNG

There's no denying the popularity and impact that Backbone.js [backbonejs.org] by Jeremy Ashkenas and DocumentCloud has made. Although the documentation and examples are excellent, I thought it would be interesting to review the code on a more technical level. Hopefully this will give readers a deeper understanding of Backbone, and as the MVC series progresses these code reviews should prove useful in accurately comparing the many competing frameworks.

Follow me on a guided tour through Backbone's source to really learn how it works and what it provides.

Namespace and Conflict Management

Like most client-side projects, Backbone.js wraps everything in an immediately invoked function expression:

```
(function(){
  // Backbone.js
}).call(this);
```

Several things happen during this configuration stage. A `Backbone` "namespace" is created, and multiple versions of Backbone on the same page are supported through the `noConflict` mode:

```
var root = this;
var previousBackbone = root.Backbone;

Backbone.noConflict = function() {
  root.Backbone = previousBackbone;
  return this;
};
```

Multiple versions of Backbone can be used on the same page by calling `noConflict` like this:

```
var Backbone19 = Backbone.noConflict();
// Backbone19 refers to the most recently loaded
// version, and `window.Backbone` will be
// restored to the previously loaded version
```

This initial configuration code also supports CommonJS modules so Backbone can be used in Node projects:

```
var Backbone;
if (typeof exports !== 'undefined') {
  Backbone = exports;
} else {
  Backbone = root.Backbone = {};
}
```

The existence of Underscore.js [underscorejs.org] (also by DocumentCloud) and a jQuery-like library is checked as well.

Server Support

During configuration, Backbone sets a variable to denote if extended HTTP methods are supported by the server. Another setting controls if the server understands the correct MIME type for JSON:

```
Backbone.emulateHTTP = false;
Backbone.emulateJSON = false;
```

The Backbone.sync method that uses these values is actually an integral part of Backbone.js. A jQuery-like `ajax` method is assumed, so HTTP parameters are organized based on jQuery's API. Searching through the code for calls to the `sync` method shows it's used whenever a model is saved, fetched, or deleted (destroyed).

What if jQuery's `ajax` API isn't appropriate for your project? Well, it seems like the `sync` method is the right place to override for changing how models are persisted, and this is confirmed by Backbone's documentation:

> *The sync function may be overriden globally as `Backbone.sync`, or at a finer-grained level, by adding a `sync` function to a Backbone collection or to an individual model.*

There's no fancy plugin API for adding a persistence layer — simply override `Backbone.sync` with the same function signature:

```
Backbone.sync = function(method, model, options)
{
};
```

The default `methodMap` is useful for working out what the `method` argument does:

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'delete': 'DELETE',
  'read':   'GET'
};
```

Events

Backbone has a built-in module for handling events. It's a simple object with the following methods:

- `on: function(events, callback, context) ,` aliased to bind
- `off: function(events, callback, context) {,` aliased to unbind
- `trigger: function(events) {`

Each of these methods returns `this`, so it's a chainable object. The comments recommend using Underscore.js to add `Backbone.Events` to any object:

```
//     var object = {};
//     _.extend(object, Backbone.Events);
//     object.on('expand', function(){
// alert('expanded'); });
//     object.trigger('expand');
```

This won't overwrite the existing object; it appends the methods instead. That means it's easy to add event support to other objects in your project.

Model

Backbone.Model is where things start to get serious. Models use a constructor function that sets up various internal properties for managing things like attributes and whether or not the model has been saved yet. Underscore.js is used to add the methods from `Backbone.Events`, and then the public model API is defined. This contains most of the frequently used Backbone methods.

Notice that `Backbone.Model` is actually quite transparent: there aren't any private methods defined inside the constructor.

The `set` method supports two different signatures, making it easy to support a single attribute or multiple attributes:

```
// Handle both `"key", value` and `{key: value}`
// -style arguments.
if (_.isObject(key) || key == null) {
  attrs = key;
  options = value;
} else {
  attrs = {};
  attrs[key] = value;
}
```

The `save` method does something similar. Notice how the authors ensure an object is always set for `options`:

```
options || (options = {});
```

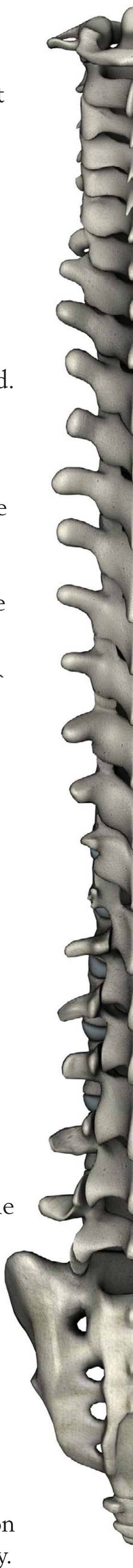
In terms of expressing the programmer's intent, this seems better than `options = options || {}`.

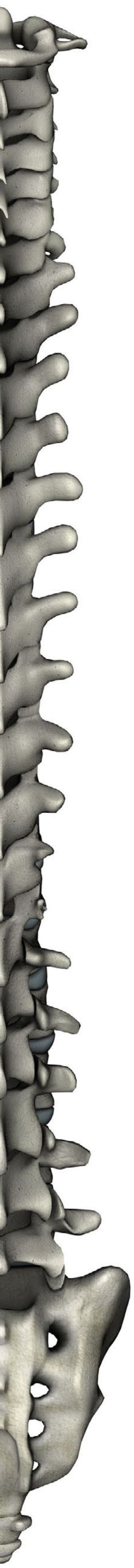
The `set` method triggers validations and prevents the method from progressing if a validation fails:

```
if (!this._validate(attrs, options)) return
false;
```

Next each attribute is iterated over. If the attribute has changed, according to Underscore's `isEqual` method, then the change is recorded. Once the list of changes has been built, the `change` method is called.

The `change` method calls `trigger` for each change. This allows for changes to any attribute to be listened on specifically, allowing the UI to be updated appropriately. For example, let's say I had a `blogPost` model instance:





```
blogPost.on('change:title', function() {
  // Update the HTML for the page title
});

blogPost.set('title', 'All Work and No Play
Makes Blank a Blank Blank');
```

Other methods also trigger `change` events: `unset`, `clear`, and `fetch`. Since we don't always care if these cause a change event, a `silent` option is supported that will be passed from these methods to `set`. It's actually quite interesting how each of these methods is implemented by reusing `set`:

```
// Clear all attributes on the model, firing
//`"change"` unless you choose to silence it.
clear: function(options) {
  options = _.extend({}, options, {unset:
true});
  return this.set(_.clone(this.attributes),
options);
},
```

The `fetch` method will trigger a sync operation that will retrieve the latest values from the server (or suitable persistence layer if it's been overridden).

The `save` method ensures only valid attributes and models are persisted, and calls `set` if required:

```
if (options.wait) {
  if (!this._validate(attrs, options)) return
false;
  current = _.clone(this.attributes);
}

// Regular saves `set` attributes before
// persisting to the server.
var silentOptions = _.extend({}, options,
{silent: true});
if (attrs && !this.set(attrs, options.wait ?
silentOptions : options)) {
  return false;
}

// Do not persist invalid models.
if (!attrs && !this.isValid()) return false;
```

The `sync` method is called to persist the changes to the server. `isNew` is used to determine if the model should be created or updated. The `isNew` state is determined by whether an `id` attribute exists or not. This could be easily overridden if a given persistence layer works a different way. Notice that Backbone internally references this attribute as `this.id` and doesn't map it to the value set with `idAttribute` in `isNew`.

A `parse` placeholder method is called whenever models are fetched, or saved. There are examples of people using this to parse other data formats like XML.

Conclusion

After looking at the Backbone.js setup and model code, we've already learned quite a lot:

- Any persistence scheme can be supported by overriding the `sync` method.
- Models are event-based.
- `change` events can drive the UI whenever models change.
- Models know when to create or update objects.
- Reusing Backbone's models, events, and Underscore methods is useful for organizing project architecture.

Although the Backbone models don't have a plugin layer, the authors have kept the design open and allowed for just the right hooks to support lots of HTTP services and data types outside the built-in RESTful JSON-oriented design.

Backbone relies heavily on Underscore.js, which means applications built with it can build on both of these libraries to create (potentially) well-designed and reusable code. ■

Alex Young is a software engineer based in London, England. He founded Helicoid as a limited company in 2006. Alex has built 5 commercial Ruby on Rails web applications for Helicoid. Each web app he build has a mobile interface, API, and some even have iPhone and Mac clients.

Reprinted with permission of the original author.
First appeared in *hn.my/bbone* (dailyjs.com)

Less is Exponentially More

By ROB PIKE

HERE IS THE text of the talk I gave at the Go SF meeting in June, 2012.

This is a personal talk. I do not speak for anyone else on the Go team here, although I want to acknowledge right up front that the team is what made and continues to make Go happen. I'd also like to thank the Go SF organizers for giving me the opportunity to talk to you.

I was asked a few weeks ago, "What was the biggest surprise you encountered rolling out Go?" I knew the answer instantly: Although we expected C++ programmers to see Go as an alternative, instead most Go programmers come from languages like Python and Ruby. Very few come from C++.

We — Ken, Robert, and myself — were C++ programmers when we designed a new language to solve the problems that we thought needed to be solved for the kind of software we wrote. It seems almost paradoxical that other C++ programmers don't seem to care.

I'd like to talk today about what prompted us to create Go, and why the result should not have surprised us like this. I promise this will be more about Go than about C++, and that if you don't know C++ you'll be able to follow along.

The answer can be summarized like this: do you think less is more, or less is less?

Here is a metaphor, in the form of a true story. Bell Labs centers were originally assigned 3-digit numbers: 111 for Physics Research, 127 for Computing Sciences Research, and so on. In the early 1980s a memo came around announcing that as our understanding of research had grown, it had become necessary to add another digit so we could better characterize our work. So our center became 1127. Ron Hardin joked, half-seriously, that if we really understood our world better, we could drop a digit and go down from 127 to just 27. Of course management didn't get the joke, nor were they expected to, but I think there's wisdom in it. Less can be more. The better you understand, the pithier you can be.

Keep that idea in mind.

Back around September 2007, I was doing some minor but central work on an enormous Google C++ program, one you've all interacted with, and my compilations were taking about 45 minutes on our huge distributed compile cluster. An announcement came around that there was going to be a talk presented by a couple of Google employees serving on the C++ standards committee. They were going to tell us what was coming in C++0x, as it was called at the time. (It's now known as C++11).

In the span of an hour at that talk we heard something like 35 new features that were being planned. In fact there were many more, but only 35 were described in the talk. Some of the features were minor, of course, but the ones in the talk were at least significant enough to call out. Some were very subtle and hard to understand, like rvalue references, while others are especially C++-like, such as variadic templates, and some others are just crazy, like user-defined literals.

At this point I asked myself a question: did the C++ committee really believe that what was wrong with C++ was that it didn't have enough features? Surely, in a variant of Ron Hardin's joke, it would be a greater achievement to simplify the language rather than to add to it. Of course, that's ridiculous, but keep the idea in mind.

Just a few months before that C++ talk I had given a talk myself, which you can see on YouTube [hn.my/toy], about a toy concurrent language I had built way back in the 1980s. That language was called Newsqueak and of course it is a precursor to Go.

I gave that talk because there were ideas in Newsqueak that I missed in my work at Google, and I had been thinking about them again. I was convinced they would make it easier to write server code, and Google could really benefit from that.

I actually tried and failed to find a way to bring the ideas to C++. It was too difficult to couple the concurrent operations with C++'s control structures, and in turn that made it too hard to see the real advantages. Plus, C++ just made it all seem too cumbersome, although I admit I was never truly facile in the language. So I abandoned the idea.

But the C++0x talk got me thinking again. One thing that really bothered me — and I think Ken and Robert as well — was the new C++ memory model with atomic types. It just felt wrong to put such a microscopically-defined set of details into an already over-burdened type system. It also seemed short-sighted, since it's likely that hardware will change significantly in the next decade, and it would be unwise to couple the language too tightly to today's hardware.

We returned to our offices after the talk. I started another compilation, turned my chair around to face Robert, and started asking pointed questions. Before the compilation was done, we'd roped Ken in and had decided to do something. We did not want to be writing in C++ forever, and we — me especially — wanted to have concurrency at our fingertips when writing Google code. We also wanted to address the problem of "programming in the large" head on. More on that later.

We wrote on the white board a bunch of stuff that we wanted, desiderata if you will. We thought big, ignoring detailed syntax and semantics and focusing on the big picture.

I still have a fascinating mail thread from that week. Here are a couple of excerpts:

Robert: Starting point: C, fix some obvious flaws, remove crud, add a few missing features.

Rob: Name: "Go." You can invent reasons for this name, but it has nice properties. It's short, easy to type. Tools: goc, gol, goa. If there's an interactive debugger/interpreter it could just be called "go." The suffix is .go.

Robert: Empty interfaces: interface {}. These are implemented by all interfaces, and thus this could take the place of void.*

We didn't figure it all out right away. For instance, it took us over a year to figure out arrays and slices. But a significant amount of the flavor of the language emerged in that first couple of days.

Notice that Robert said C was the starting point, not C++. I'm not certain but I believe he meant C proper, especially because Ken was there. But it's also true that, in the end, we didn't really start from C. We built from scratch, borrowing only minor things like operators and brace brackets and a few common keywords. (And of course we also borrowed ideas from other languages we knew.) In any case, I see now that we reacted to C++ by going back down to basics, breaking it all down and starting over. We weren't trying to design a better C++, or even a better C. It was to be a better language overall for the kind of software we cared about.

In the end of course it came out quite different from either C or C++. More different even than many realize. I made a list of significant simplifications in Go over C and C++:

- Regular syntax (don't need a symbol table to parse)
- Garbage collection (only)
- No header files
- Explicit dependencies
- No circular dependencies
- Constants are just numbers
- Int and int32 are distinct types
- Letter case sets visibility
- Methods for any type (no classes)
- No subtype inheritance (no subclasses)
- Package-level initialization and well-defined order of initialization
- Files compiled together in a package
- Package-level globals presented in any order
- No arithmetic conversions (constants help)
- Interfaces are implicit (no "implements" declaration)
- Embedding (no promotion to superclass)

- Methods are declared as functions (no special location)
- Methods are just functions
- Interfaces are just methods (no data)
- Methods match by name only (not by type)
- No constructors or destructors
- Postincrement and postdecrement are statements, not expressions
- No preincrement or predecrement
- Assignment is not an expression
- Evaluation order defined in assignment, function call (no "sequence point")
- No pointer arithmetic
- Memory is always zeroed
- Legal to take address of local variable
- No "this" in methods
- Segmented stacks
- No const or other type annotations
- No templates
- No exceptions
- Built-in string, slice, map
- Array bounds checking

And yet, with that long list of simplifications and missing pieces, Go is, I believe, more expressive than C or C++. Less can be more.

But you can't take out everything. You need building blocks such as an idea about how types behave, and syntax that works well in practice, and some ineffable thing that makes libraries interoperate well.

We also added some things that were not in C or C++, like slices and maps, composite literals, expressions at the top level of the file (which is a huge thing that mostly goes unremarked), reflection, garbage collection, and so on. Concurrency, too, naturally.

One thing that is conspicuously absent is of course a type hierarchy. Allow me to be rude about that for a minute.

Early in the rollout of Go I was told by someone that he could not imagine working in a language without generic types. As I have reported elsewhere, I found that an odd remark.

To be fair he was probably saying in his own way that he really liked what the STL does for him in C++. For the purpose of argument, though, let's take his claim at face value.

What it says is that he finds writing containers like lists of ints and maps of strings an unbearable burden. I find that an odd claim. I spend very little of my programming time struggling with those issues, even in languages without generic types.

But more important, what it says is that types are the way to lift that burden. Types. Not polymorphic functions or language primitives or helpers of other kinds, but types.

That's the detail that sticks with me.

Programmers who come to Go from C++ and Java miss the idea of programming with types, particularly inheritance and subclassing and all that. Perhaps I'm a philistine about types but I've never found that model particularly expressive.

My late friend Alain Fournier once told me that he considered the lowest form of academic work to be taxonomy. And you know what? Type hierarchies are just taxonomy. You need to decide what piece goes in what box, every type's parent, whether A inherits from B or B from A. Is a sortable array an array that sorts or a sorter represented by an array? If you believe that types address all design issues you must make that decision.

I believe that's a preposterous way to think about programming. What matters isn't the ancestor relations between things but what they can do for you.

That, of course, is where interfaces come into Go. But they're part of a bigger picture, the true Go philosophy.

If C++ and Java are about type hierarchies and the taxonomy of types, Go is about composition.

Doug McIlroy, the eventual inventor of Unix pipes, wrote in 1964 (!):

> *We should have some ways of coupling programs like garden hose — screw in another segment when it becomes necessary to massage data in another way. This is the way of IO.*

That is the way of Go also. Go takes that idea and pushes it very far. It is a language of composition and coupling.

The obvious example is the way interfaces give us the composition of components. It doesn't matter what that thing is, if it implements method M, I can just drop it in here.

Another important example is how concurrency gives us the composition of independently executing computations.

And there's even an unusual (and very simple) form of type composition: embedding.

These compositional techniques are what give Go its flavor, which is profoundly different from the flavor of C++ or Java programs.

There's an unrelated aspect of Go's design I'd like to touch upon: Go was designed to help write big programs, written and maintained by big teams.

There's this idea about "programming in the large" and somehow C++ and Java own that domain. I believe that's just a historical accident, or perhaps an industrial accident. But the widely held belief is that it has something to do with object-oriented design.

I don't buy that at all. Big software needs methodology to be sure, but not nearly as much as it needs strong dependency management and clean interface abstraction and superb documentation tools, none of which is served well by C++ (although Java does noticeably better).

We don't know yet, because not enough software has been written in Go, but I'm confident Go will turn out to be a superb language for programming in the large. Time will tell.

Now, to come back to the surprising question that opened my talk:

Why does Go, a language designed from the ground up for what C++ is used for, not attract more C++ programmers?

Jokes aside, I think it's because Go and C++ are profoundly different philosophically.

C++ is about having it all there at your fingertips. I found this quote on a C++11 FAQ:

"The range of abstractions that C++ can express elegantly, flexibly, and at zero costs compared to hand-crafted specialized code has greatly increased." That way of thinking just isn't the way Go operates. Zero cost isn't a goal, at least not zero CPU cost. Go's claim is that minimizing programmer effort is a more important consideration.

Go isn't all-encompassing. You don't get everything built in. You don't have precise control of every nuance of execution. For instance, you don't have RAII. Instead you get a garbage collector. You don't even get a memory-freeing function.

What you're given is a set of powerful but easy to understand, easy to use building blocks from which you can assemble — compose — a solution to your problem. It might not end up quite as fast or as sophisticated or as ideologically motivated as the solution you'd write in some of those other languages, but it'll almost certainly be easier to write, easier to read, easier to understand, easier to maintain, and maybe safer.

To put it another way, oversimplifying of course:

Python and Ruby programmers come to Go because they don't have to surrender much expressiveness, but gain performance and get to play with concurrency.

C++ programmers don't come to Go because they have fought hard to gain exquisite control of their programming domain, and don't want to surrender any of it. To them, software isn't just about getting the job done, it's about doing it a certain way.

The issue, then, is that Go's success would contradict their world view.

And we should have realized that from the beginning. People who are excited about C++11's new features are not going to care about a language that has so much less. Even if, in the end, it offers so much more. ■

Rob Pike is a Distinguished Engineer at Google, Inc. He works on distributed systems, data mining, programming languages, and software development tools. Most recently he has been a co-designer and developer of the Go programming language.

Both true and false: a Zen moment with C

By MARK SHROYER

I RAN INTO A really fun bug at work yesterday, where I discovered that my C program was branching down logically inconsistent code paths. After drinking another cup of coffee and firing up GDB I realized that somehow a boolean variable in my code was simultaneously testing as both true and not true.

While I cannot reproduce the actual source code here, the effect was that code like

```
bool p;

/* ... */

if ( p )
    puts("p is true");

if ( ! p )
    puts("p is false");
```

would produce the output:

```
p is true
p is false
```

So what's going on here?

Well it turns out that the authors of the C language specification (and the people who went on to implement compilers for it) were serious about the concept of undefined behavior. In particular, the result of attempting to use an uninitialized variable is undefined.

And in this case that's exactly what happened: I had failed to properly initialize some memory. Easy; bug fixed. But what I think is interesting are the reasons this code failed in precisely the way it did. In order to investigate that, we need to get specific.

On 64-bit Linux (Ubuntu 12.04), compiling the following program:

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, char *argv[])
{
    volatile bool p;

    if ( p )
        puts("p is true");
    else
        puts("p is not true");

    if ( ! p )
        puts("p is false");
    else
        puts("p is not false");

    return 0;
}
```

with GCC 4.6.3, using the command line:

```
$ gcc bool1.c -g0 -O0 -fno-dwarf2-cfi-asm
-masm=intel -S -o bool1.asm
```

produces this (truncated) assembly language:

```
        .file   "bool1.c"
        .intel_syntax noprefix
        .section        .rodata
.LC0:
        .string "p is true"
.LC1:
        .string "p is not true"
.LC2:
        .string "p is false"
.LC3:
        .string "p is not false"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        push    rbp
.LCFI0:
        mov     rbp, rsp
.LCFI1:
        sub     rsp, 32
.LCFI2:
        mov     DWORD PTR [rbp-20], edi
        mov     QWORD PTR [rbp-32], rsi
        movzx   eax, BYTE PTR [rbp-1]
        test    al, al
        je      .L2
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        jmp     .L3
.L2:
        mov     edi, OFFSET FLAT:.LC1
        call    puts
.L3:
        movzx   eax, BYTE PTR [rbp-1]
        xor     eax, 1
        test    al, al
        je      .L4
        mov     edi, OFFSET FLAT:.LC2
        call    puts
        jmp     .L5
.L4:
        mov     edi, OFFSET FLAT:.LC3
        call    puts
.L5:
        mov     eax, 0
        leave
.LCFI3:
        ret
```

To perform the test `if ( p )` here, first the stack variable is loaded into a 32-bit register with `movzx eax, BYTE PTR [rbp-1]`, and then we use the instruction `test al, al` which sets the zero flag (ZF) if the lower eight bits of this value are zero. Next we execute the conditional jump `je .L2`, which jumps to print "p is not true" if ZF was set; otherwise we don't jump, and "p is true" gets printed instead.

Next let's examine the second test, `if ( ! p )`, at label `.L3`. This starts out the same by loading the boolean variable into register `eax`, but notice how the negation is handled. Rather than reorder the jumps or use `jne` instead of `je`, the compiler explicitly negates the boolean by performing a bitwise exclusive-or: `xor eax, 1`.

Normally this would be fine — a `bool` variable is only supposed to contain a value of zero or one, in which case its value can be negated by XOR with 1. When you cast to a `bool` at runtime, the compiler generates code to ensure only one or zero gets stored. For instance, the cast in this program:

```
#include <stdbool.h>

volatile char c = 0xff;
volatile bool p;

int main(int argc, char* argv[])
{
    p = (bool)c;
    return 0;
}
```

is implemented as the following four instructions:

```
        movzx   eax, BYTE PTR c[rip]
        test    al, al
        setne   al
        mov     BYTE PTR p[rip], al
```

wherein `setne` sets the register `al` to exactly 1 if the `char` contained any nonzero value, before saving the register's 8-bit value to the boolean variable.

But the compiler affords us no such protection if we accidentally use an uninitialized value as a boolean. It doesn't have to because it's not the compiler's responsibility; the result of using an uninitialized stack variable is undefined. And so if we somehow wind up with a value of e.g. `0x60` stored at the address of a `bool` variable (as I saw during my troubleshooting yesterday), both the variable and its negation (via exclusive or with 1) will be nonzero and therefore test as true.

Interestingly, enabling optimization (-O2) in GCC causes the compiler to factor out the XOR and instead reorder the jumps, meaning this program actually behaves more robustly under compiler optimization (for certain definitions of "robust" anyway):

```
        .file   "bool1.c"
        .intel_syntax noprefix
        .section        .rodata.str1.1,"aMS",@
progbits,1
.LC0:
        .string "p is true"
.LC1:
        .string "p is not true"
.LC2:
        .string "p is false"
.LC3:
        .string "p is not false"
        .section        .text.startup,"ax",@
progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
main:
.LFB22:
        sub     rsp, 24
.LCFI0:
        movzx   eax, BYTE PTR [rsp+15]
        test    al, al
        je      .L2
        mov     edi, OFFSET FLAT:.LC0
        call    puts
.L3:
        movzx   eax, BYTE PTR [rsp+15]
        test    al, al
        je      .L7
        mov     edi, OFFSET FLAT:.LC3
        call    puts
.L5:
        xor     eax, eax
        add     rsp, 24
.LCFI1:
        ret
```

And of course when we compare `char`, `int`, or other multiple-bit values for truthiness, the compiler makes no such assumption that the value can be logically negated by bitwise XOR; instead it uses `jne` in place of the `je` instruction. (Maybe someone with more knowledge of the compiler can say why GCC with `-O0` uses an `xor` at all when testing the negation of a `bool`.) ■

Mark Shroyer is a software engineer in Miami who writes low-level code for embedded Linux devices. In his free time he also enjoys geeking out with high-level programming languages. You can find him online at *markshroyer.com*

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

I Am A Statistician And I Buy Lottery Tickets

By DAVID WOODS

When my friends hear me say that I'm buying a lottery ticket for a big draw, I often get the comment, "But aren't you a statistician?" The implication is that only people who are ignorant of probability would play the lottery. I've also heard the belief that the lottery is a tax on poor people. I have a different view, that buying lottery tickets is perfectly rational for me.

There are a number of different lotteries here in Melbourne, Australia, but let's consider the draw for this Tuesday, the "Super 7's Oz Lotto." This game draws seven balls from 45, and the big "first division" prize is for getting all seven correct. There are six other consolation divisions with much smaller prizes for getting a smaller number of balls correct. The first division prize for this week is $70 million, which is quite a bit bigger than usual.

The odds against winning the big prize are 45,379,619:1. That sounds like a long shot by any measure, and this is the number that people usually quote when they tell you how crazy you are. However, those are the odds for a single combination of balls — one line. Usually, you'd buy a ticket with multiple lines. For a standard 12-line ticket, the odds shorten to 3,781,635:1, still not very likely, but definitely much improved. I actually got excited by the big draw this week and bought a 36-line ticket, for odds of 1,260,545:1, just over "one in a million" odds.

The standard argument of people who think that lotto is for suckers is based on expected return. They are taught at school that a rational investment is one with an expected value greater than the price paid. I have some problems with this argument, but I've never really calculated it before, so let us consider the numbers. Each line on the ticket costs $1.20, so a 12-line ticket costs $14.40. The way we calculate expected return is to multiply the payout times the probability:

Expected return per line = $70,000,000 / 45,379,620 = $1.54

Hang on, each line costs us $1.20 and has an expected value of $1.54… that sounds like a good investment! In fact, it could be tempting to buy every single combination of numbers to guarantee a win, with a cost of $54,455,545 and a profit of $15,544,455.

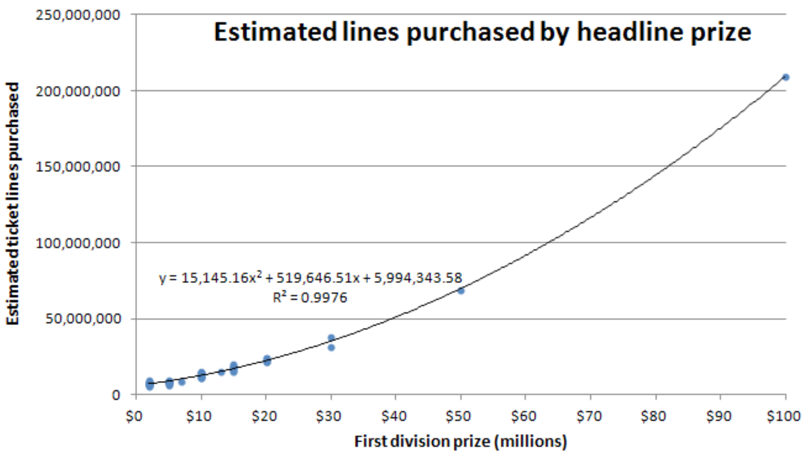
Unfortunately, there is another factor to consider. We only get the full prize if we win it alone. If another person also has the winning combination, we get half, only $35,000,000. If three or more people win, then we get a correspondingly smaller fraction of the prize. To calculate the true expected value of our ticket we need to estimate the probability distribution of the number of winners. This requires us to know the number of tickets sold. It turns out we can get an estimate of this, but we have to do some work.

The table below shows the results from the $50,000,000 draw last week. Note that the no one won the first division prize. So it jackpots to this week.

Division	Prize per winner	Division prize pool	Winners	Odds	Estimated lines sold
1	-	-	-	45,379,619:1	-
2	$34,346.25	$721,271	21	3,241,400:1	68,069,430
3	$4,759.50	$1,484,964	312	180,077:1	56,184,291
4	$366.45	$763,681	2,084	29,601:1	61,690,234
5	$49.05	$890,600	18,157	3,429:1	62,279,498
6	$24.00	$10,185,408	424,392	153:1	65,226,403
7	$14.55	$11,398,484	783,401	86:1	68,123,866

The odds allow us to calculate the probability of winning each division. Since we know the number of lines that actually won each division, we can estimate the number of lines sold. This is obviously probabilistic, so the estimates are different, but the estimate for the seventh division should be the most accurate.

We would expect the number of lines sold to be related to the advertised first division prize. This is the headline prize that is featured on TV and posters around town. Last week's advertised prize was $50,000,000. We can search to find the above table for past draws, and we can go back through the lottery's twitter feed to find the advertised first division prize. The graph below shows the headline prize vs. estimated lines sold (using the division seven winners) for the past 52 draws. Also included on the graph is the biggest draw ever, $100,000,000 on 30 June 2009.



We can use the built-in functionality in Excel to fit a quadratic curve to the data. With an R-Square value of 99.76%, the curve seems to fit the data well. By plugging $70,000,000 into this equation, we can estimate that approximately 116,580,883 lines will be sold in the upcoming draw. We can now use this to calculate the probability of winning the first division alone or having one, two, or more other winners.

The following formula gives us the probability of having X winners, where the number of lines sold is N, and the probability per line is p.

$$P(X) = {}^{N}C_{X} * p^{X} * (1-p)^{N-X}$$

By plugging in the values of N = 116,580,883 and p = 1 / 45,379,620, we can calculate the probability for all values of X. The table below shows this for X between 0 and 11.

Winners	Probability	Cumulative	Share of prize	Prob, given a win	Expected return
0	7.66%	7.66%	$0		
1	19.68%	27.34%	$70,000,000	21.31%	$14,920,082
2	25.28%	52.62%	$35,000,000	27.38%	$9,582,475
3	21.65%	74.27%	$23,333,333	23.45%	$5,470,559
4	13.90%	88.18%	$17,500,000	15.06%	$2,635,114
5	7.14%	95.32%	$14,000,000	7.74%	$1,083,143
6	3.06%	98.38%	$11,666,667	3.31%	$386,474
7	1.12%	99.50%	$10,000,000	1.22%	$121,574
8	0.36%	99.86%	$8,750,000	0.39%	$34,161
9	0.10%	99.97%	$7,777,778	0.11%	$8,668
10	0.03%	99.99%	$7,000,000	0.03%	$2,004
11	0.01%	100.00%	$6,363,636	0.01%	$425

The fifth and sixth columns are what we are interested in. We are trying to determine the expected returns, given that we have won. The probabilities are calculated using the following formula:

$$P(X \mid X > 0) = P(X) / (1 - P(0))$$

To find the expected return, if we win first division, we then sum the values in the right-hand column to give $34,244,780. Multiplying this by p gives an expected value of each line of $0.75. So, rationally, we shouldn't invest in tickets for this lottery, since 0.75 < 1.20.

However, there is another factor to consider. We have calculated the expected return for winning first division; however, we also have a chance of winning the other divisions. We need to calculate how much we can expect to win for each of the other divisions.

116,580,883 lines at $1.20 per line gives a total amount paid of $139,897,060. The game rules state that at least 55% of this total must be used in the prize pool, and analysis of results from the past year confirms this figure. This gives a prize pool of $76,943,383. The game odds page also gives the proportion of the prize pool that is allocated to each division. Using the odds, we can then determine the expected number of winners and the payout per winning line. The table below shows the expected value of a line for each division:

Division	Proportion of pool	Division prize pool	Odds	Expected winners	Prize per winner	Expected return
2	1.70%	$1,308,038	3,241,400:1	36	$36,368.52	$0.0112
3	3.50%	$2,693,018	180,077:1	683	$3,940.86	$0.0231
4	1.80%	$1,384,981	29,601:1	3,938	$351.67	$0.0119
5	2.10%	$1,615,811	3,429:1	37,926	$42.60	$0.0139
6	24.00%	$18,466,412	153:1	758,527	$24.35	$0.1584
7	26.90%	$20,697,770	86:1	1,340,640	$15.44	$0.1775

Adding up the values in the expected return column gives an expected return of $0.40 for divisions two to seven. Added to the expected return from the first division, it gives an expected value per line of $0.75 + $0.40 = $1.15. This is still less than the $1.20 we paid, but not by too much.

However, even if the expected value of a ticket was positive, it would still be a terrible investment. The expected value argument only really works in the long term — if I was investing in millions of lines of tickets or millions of different draws, then in the long run I would expect to make money. This is how casinos work: very small positive expected returns multiplied by millions of transactions. For an individual though, the probability of winning is essentially zero. We don't get to perform millions of transactions, so we are almost certainly going to lose our "investment."

So why do I still buy lottery tickets? Definitely not for the expected monetary return on investment. I think of it as a discretionary entertainment spend. I get literally hours of enjoyment from fantasizing what I'd do if I won. I happily spend $25 for two hours of entertainment at the movies, and I don't judge the value of that experience based on its expected return. For me, a lottery ticket for the occasional big draw has just as much entertainment value, or more, than the many other things that I spend money on to entertain myself.

The decision of whether to buy a lottery ticket shouldn't be based on the probability of winning or the expected return of a ticket, but on the entertainment value that comes from imagining a different life. If that entertainment value compares favorably with other activities with a similar price, then go for it. Plus, it has the added bonus that you might actually win; one-in-a-million events happen every day. Someone eventually wins the big prize, and you have to be in to win. ■

David is an analytics consultant in Melbourne, Australia. He helps companies with things like predictive modelling, optimization, and forecasting.

stripe

Accept payments online.

MEMSET®
HOSTING

Rent your IT infrastructure from Memset and discover the incredible benefits of cloud computing.

MINISERVER™
CLOUD COMPUTE

From £0.015p/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

MEMSTORE™
CLOUD STORAGE

£0.07p/GByte/month or less
99.999999% object durability
99.995% availability guarantee
RESTful API, FTP/SFTP and CDN Service

SCAN THE CODE FOR MORE INFORMATION

MEMSET®
HOSTING

CarbonNeutral® hosting

ISO 9001: Quality
ISO 14001: Environmental
ISO 27001: Security

Find out more about us at
www.memset.com

or chat to our sales team on
0800 634 9270.