Hardware Interrupts

Description: In this fourth installment of Steve's "How Computers Work" series, Steve explains the operation of "hardware interrupts" which, by instantly interrupting the normal flow of instructions, allow computers to attend to the needs of the hardware that interacts with the outside world while they are in the middle of doing other things.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-241.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-241-lq.mp3

Leo Laporte: This is Security Now! with Steve Gibson, Episode 241 for March 25, 2010: Hardware Interrupts.

It's time for Security Now!, the show that covers all the important things about keeping yourself safe online, protecting your privacy, avoiding spyware and viruses. And lately it's also been a great show for how computers work. And that's thanks to this guy, Steve Gibson, one of the pioneers of technology. Guy's been around a long time. He wrote SpinRite, the world's finest hard drive maintenance utility; just knows the ins and outs of all this stuff. Even likes to write assembly language code for the PDP-8. Hi, Steve.

Steve Gibson: Hey, Leo. And remember, I'm old.

Leo: And he's old.

Steve: I'm old. In fact, it's funny, the guy that blogged that, where he said "Steve Gibson is an old man..."

Leo: Old and weird, I think he said you were.

Steve: Well, no, what was the - there was a term later on in that posting.

Leo: But it was all positive.

Steve: Kooky, kooky.

Leo: Kooky. That's it.

Steve: "Kooky" was the word he used. So I thought, well, I haven't heard that for a while, not about myself but just in general. But he did send back some email, because I'm sure he and other friends listened to the podcast and said, "Hey, they're talking about you." Well, remember that he was CNET China, I think was where he was. And he explained, he said, "In Chinese culture, old is revered."

Leo: Yes.

Steve: It's not the same as old, and oh, look, you're old and wrinkly on the California beach. It's very different in China. So I said, "Oh, okay, I like that."

Leo: Not typically so in technology. But I think that he had a lot of good points to make about how knowing about this, how it's been and all that stuff, is very valuable, very useful.

Steve: Well, it works for us here.

Leo: And today we're going to cover something along those lines. We're going to talk about - we're going to continue building a computer from first principles; right?

Steve: Yes. And I continue getting great feedback from people who are really enjoying this. I mean, we're still covering lots of good weekly security news. But my goal is, in this computer series, is one of demystification. Rather than people just sort of, like, oh, well, you know, I turn it on, and it lights up, it's like, okay. No excuses, nothing hidden, let's look at exactly how this stuff works. And if we take it in the right sequence and build from one week to the next on what we've got before, I'm, as I promised at the beginning of this, I am sure people can end up feeling, wait a minute, that's it? That's all there is? And it's like, yes, this really is...

Leo: That's how simple it is.

Steve: It's really simple. It just does it very fast so you get complex results. But the actual little things it's doing are completely understandable.

Leo: We're going to get to that in just a little bit. But first let's see what's going on in the world of security and so forth.

Steve: Interesting little story with Firefox. I noticed since we've last spoken a week ago that Firefox, the current version train that everyone should be on is the 3.6 train. And it

went recently to 3.6.2. Now, I got a call from my tech support guy, Greg. He and I were chatting, like, maybe a week ago. And he said, "Hey, have you upgraded to 3.6?" I said yeah. And he said, "My scrolling broke." And he's using - he's using the mouse which I've spoken of and recommended several times, the Logitech Anywhere MX mouse with that inertia scrolling where you just sort of spin the wheel, and you get to scroll? And when you combine that with KatMouse, which is the little free software which automatically sends the scrolling to whatever window you're hovering over, without you having to click in the window in order to activate that window, to make it the current window, is just - it's so wonderful.

And what he was complaining of was that 3.6, he'd upgraded from 3.5 to 3.6, and that broke. And so I thought, oh, well, no, it's working for me. And so he went back to 3.5, and then it worked. And then later last week I tried upgrading one of my laptops from 3.5 to 3.6, and it broke. So I had duplicated Greg's experience. Anyway, when I saw that Firefox - so I left myself at 3.5 on my main machine. And when I saw that Firefox had gone to 3.6.2, I thought, well, I'll just - I wonder what they fixed?

Well, they fixed a heap overflow corruption problem that was a security concern, which was the main impetus for this. But they also talked about multiple stability fixes. So I went into the detailed changes at 3.6.2. And in there was a Bug No. 540510 that said, "Scrolling messages forwarded back from a plug-in are dropped instead of processed." It's like, oh, please, could this be it? So I was able to, prior to upgrading, I was able to duplicate the problem. And I noted, for example, that when, if I happened to be scrolling, the mouse was hovering over an object on the page like a Flash object or something that was fancy, the page wouldn't scroll. If I deliberately moved the mouse out of that, then it would scroll again. So it was exactly what the symptomology was, which was scrolling messages were getting lost. I upgraded to 3.6.2, and that problem was fixed. So anyway...

Leo: Huh. Isn't that interesting.

Steve: Yeah.

Leo: Now, that's from the heap overflow, or...

Steve: No, no. They called it a "regression problem." So it was something - it was a bug they introduced inadvertently at some point along the way.

Leo: Ah. What is a regression error? What does that mean?

Steve: Well, the idea is that, while you're fixing things, you change something that sort of brings back a problem that you had fixed previously. And so regression testing...

Leo: Oh, you regress to a previous bad state.

Steve: Exactly. You regress one aspect. Something that was fixed at one point comes back again.

Leo: Right.

Steve: And so one of the things that, like, major companies like Microsoft will do is they'll deliberately create a forward-moving test suite that tests to make sure that everything that they had fixed before stays fixed. And that's what's called "regression testing," to make sure that you haven't reintroduced a problem that you'd fixed before. So, I mean, it's one of the reasons this stuff is sort of slow to get done is that they're not just jumping in and going, oh, there's the problem, and change an ampersand to a circumflex or something, like oh, we had the wrong character that we typed there. They also want to make sure that they test - they add to their test suite a check to make sure that this behavior never comes back for any reason. So...

Leo: Right. Isn't that, though, doesn't that kind of come from poor programming practice, if stuff is so tightly coupled that a change in one place changes things, I mean, I know it's unavoidable because we're maintaining lots of code written by lots of different people.

Steve: Yeah.

Leo: But ideally in a program you shouldn't have these kinds of interactions.

Steve: [Laughing] Ideally, Leo...

Leo: Am I showing my navet? Yeah.

Steve: Ideally, this podcast would not exist.

Leo: Okay. I see what you're saying.

Steve: Ideally we wouldn't need to be talking about this.

Leo: However...

Steve: Yeah.

Leo: We live in the real world.

Steve: And in fact, down in my little errata section, I want to - I'm going to talk a little bit about sort of is it time to rethink all of this. I mean, this is just getting so ridiculous.

Leo: Oh, okay. Okay.

Steve: So also I'm sure you're aware, and I just wanted to cover it briefly in our podcast, the updates with Google's trials and tribulations relative to China and what they have decided to do recently. Since we last talked, Google did follow through with their statement from January 12th that they were going to stop censoring the results from their Google.cn search engine. And the way they did it - and maybe you can clarify this for me because I was confused by this, I know you've been following it more closely than I have - they did it by redirecting the Google.cn traffic to Google.com.hk.

Leo: Right.

Steve: That is, their Hong Kong domain. Since then, China has blocked that. But in the Google blog they said, "Figuring out how to make good on our promise to stop censoring search on Google.cn has been hard. We want as many people in the world as possible to have access to our services, including users in mainland China. Yet the Chinese government has been crystal clear throughout our discussions that self-censorship is a nonnegotiable, legal requirement. We believe this new approach to providing uncensored search in simplified Chinese from Google.com.hk is a sensible solution to the challenges we've faced. It's entirely legal and will meaningfully increase access to information for people in China."

Leo: Unless it's blocked by the Chinese government.

Steve: And they said, "We very much hope the Chinese government respects our decision, though we are well aware that it could at any time block access to our services."

Leo: Right.

Steve: So do you understand what they're talking about? How them simply bouncing people to .com.hk instead of .cn was legal, or wasn't breaking the spirit of what the Chinese government wanted them to do?

Leo: I think their position is that you're going to a website outside of mainland China. Oddly enough, Hong Kong is owned by mainland China. So this is why I'm not sure - let's say - that's an area I don't get. But let's say they moved it to the U.S., that you got redirected to Google.com. Well, the Chinese government can't assert what Google.com does. So all that remains to the Chinese government is to block sites outside of China.

Steve: Ah, okay, as they do with their - a big firewall.

Leo: As they do with Twitter and Facebook and many Google properties, including Blogger and...

Steve: Oh, and YouTube.

Leo: Yeah, and YouTube. So but the thing that puzzles me is Hong Kong is in fact - maybe it's in a different legal status. It is a, I mean, it's run differently than mainland China. But it is in fact owned by, I mean, it reverted to the Chinese government. I guess they have two different systems, and maybe the laws are somewhat different, and Hong Kong has some autonomy. Obviously that's the case. So they're just saying, look, we're going to continue to operate normally. If the Chinese government doesn't want the Chinese people to see it, they're going to have to filter, not us. And I think that's the right thing to do, by the way.

Steve: Yeah. There is an interesting page that I wanted to point our users to, any users who are interested about this. It's Google.com/prc - obviously as in People's Republic of China - /report.html. Again, that's www.google.com/prc/report.html, which Google is now maintaining day by day, showing, of all the different sorts of content relating to Google, which of that content that the Chinese are currently blocking to people inside of China. And so right now it shows three days' worth of different stuff. For example, YouTube has got red X's through it for all three days.

Leo: Right, right.

Steve: But lots of things are still open, and some are sort of, like blogs, I think, had like a yellow wrench on it. So those are, like, partially blocked.

Leo: The Chinese government asserted that they could selectively filter Google search results. And I don't know how they would do that. But they said they can do that. They do it right now with CNN. If you're watching CNN in China, it goes along perfectly normally. And if they do - when I was there the Uighurs were revolting in the remote regions of China.

Steve: Those darn Uighurs.

Leo: Those darn - and so when CNN did a story on it, it just would go black. You'd be watching CNN, it'd go black for a minute or two and come back. They just chop it out.

Steve: They're doing three things. They're doing DNS games, so they're able to essentially change DNS results on the fly. They also have IP-based blocking, so they can block out whole network ranges from access inside China. And they are doing deep packet inspection. They've got per-packet filters that are looking for keywords and phrases in packets. And if one of those filters trips, it propagates the IP, the remote IP, out through their network so that that IP is blocked for some length of time, on the order

of 30 minutes, so that it just sort of quickly catches any attempt to move content that they deem inappropriate. And so you'll get "connection dropped" responses inside of your browser all the time, where it just - there was a connection, and then it just got broken, just summarily broken.

Leo: Hmm. There you go.

Steve: Some interesting changes, and we'll come back to China in a second. Russia has decided that they're going to tighten down, and I think this sounds like a good thing because Russian domains are such a problem. They're going to require for the first time some proof of identity from people who register .ru, that is, in the Russian top-level domain, domain names. Computerworld had a nice story reporting that individuals will soon be, and I think even now are being, required to provide a passport, and businesses must provide their legal registration papers. Actually I think currently none is required. And historically none has been required. But they are, they have announced they're going to start doing that soon. China made a similar announcement, that I'll talk about in more detail in a second, about a month ago.

But what the security watchers are seeing is that people are simply just moving their bad stuff. China, or Russia has been notorious for being a source of scamming and spamming and all kinds of mischief in .ru domains. The problem is that, as Russia tightens down on this, there are still plenty of other top-level domains. And specifically Vietnam, which is .vn, and Indonesia, which is .id, they're beginning to become more popular havens for all the kind of stuff that we really wish the Internet was not hosting for us.

Another interesting little bit of news: On behalf of ICANN, the University of Chicago did a study, audited the domain registrar records for a huge number of domain names and determined that more than three quarters of Internet domains which are registered have incomplete, invalid, or completely false names and information. Just, you know, more than three quarters. And in fact 22 percent of website owners they found were impossible to trace.

Now, I look at this as like, okay, well, that's too bad. Except that this notion of registering a domain anonymously is a long-standing tradition on the Internet. There was this notion that one of the things you could get with the Internet was anonymity and privacy. And if you were in a situation where, for whatever reason, you felt you had a message that you wanted to make available publicly, anonymity and privacy were some things that the Internet brought along. The problem, of course, is with that being - it's a double-edged sword. And so then that brings with it security and safety problems.

Leo: Right. It's an interesting tension between privacy and anonymity, and then the fact that people, when they're forced to use their real names, act nicer. They don't send spam.

Steve: Yes.

Leo: I don't know what the answer is. It's a tough one.

Steve: Well, in fact China a month ago decided that they're going to get really serious

about this. From now on, and this policy is now in place, China will require a face-to-face meeting with anyone wanting to register a domain, during which a photograph will be taken of the registrant, and the other personal details confirmed. After this policy went in place, Chinese state-owned network operators have so far, just in the last couple weeks, closed 130,000 sites that did not have official, confirmed records. And China has said that any existing sites, any existing domains that do not have officialized records by the end of September will be closed. So they're really trying to clamp down. They say that they want to deal with the problem of pornography within China, and that this is their approach for it. And then...

Leo: They always use pornography as their excuse, but really it's dissent. We know.

Steve: Yes, it's a lot of political dissent, exactly. And then even more chilling, the head of Chinese IT Ministry said that they are continuing to research what they call a "real name system" for the Internet, which would require users to register, Chinese users to register their real identities before using public online message boards and so forth.

Leo: Yeah.

Steve: So basically stomping out anonymity within - to the degree that they have the control to do so. So that's a huge, a huge...

Leo: And that underscores the problem, where, yeah, anonymity can be, is valuable in some cases, for dissent, for whistle-blowing. And of course a repressive government doesn't want anonymity, even though that causes problems on the Internet.

Steve: Yeah. So there's so many little, a myriad of other problems, for example, that are happening all the time, continuously. I scroll through them, just these endless lists every week. And I think, okay, well, these are all so obscure that I'm not going to go into them because it'll annoy most of our population of listeners because it doesn't affect them.

But I just, as I was looking through this list this morning, just sort of shaking my head that - and it refers back to what you were saying earlier, Leo, about problems popping back up that we'd stomped out before, this whole regression issue. And I just sort of thought, you know, I hope somewhere someone is - it's really got to be a big someone, like an Apple or a Microsoft or maybe some Linux guy, are standing back and saying, you know, we're losing this race.

And, arguably, we are losing this race. The number of problems that are newly found, the AV companies who are looking for newly created viruses, I mean, they're in the tens and hundreds of thousands. And we're seeing the problem of the signatures not being updated quickly enough and just this constant flux of exploitation. And you have to think, okay, wait a minute. The problem, I mean, we understand where this came from because all of us have been around long enough to have seen machines that were initially not on the Internet, but even then had problems with floppies getting infected. So it's not like the Internet was the cause of this.

But it's the architecture, the fundamental design of our machines are not secure. I mean,

the fundamental architecture, the design, evolved from a time when there was absolutely no, and I mean no, concern about security. Which is difficult for probably our younger listeners to even imagine. But there was, once upon a time, no concern for security. It just wasn't - it wasn't on the map at all. And it began, of course, in the mainframe era, where you started to have multiuser systems where they said, okay, well, we need some sort of authentication at a terminal for someone who needs access to records because we don't want to leave our record systems open to everyone. And we need some audit trails and some accountability. So that sort of, that notion of some concern for security began to happen.

And then of course the Internet sort of grew organically from an experiment in, gee, could this notion of autonomous packet routing work on, be a scalable solution so that we're able to connect things? And I remember when I first began hearing about this notion of a global network. It's like, okay, well, that's ridiculous. You're not going to have that. Well, whoops. We do. And then there was the problem with the chicken-egg, where it was like, well, who's going to put their computers on the Internet because there is really nothing there, or who's going to bother to put content on because there's no users. Remember that, that whole discussion?

Leo: Yeah, yeah.

Steve: Again, it just sort of organically happened. It just - it did happen. And then Microsoft got caught flatfooted because they were going to go off and do MSN, the Microsoft Network, to compete with the Source and CompuServe and the other - and AOL - the other sort of dialup BBSes. And then, when the 'Net happened, Microsoft said, oh, gee, I guess we're not all going to be using big modem pools and having people dial in all the time. And they didn't have a networkable operating system, really. So they just sort of stuck Windows on the Internet. And I came along and realized that they'd also stuck everybody's C drive on the Internet.

Leo: Yes.

Steve: It's like, whoops. Gee, we hadn't really thought about that being a problem. And so of course I created ShieldsUP! in order to alert people to the fact and make it easy to check, back in those days, whether your machine was visible on the 'Net or not, and to what degree. So now, of course, malware and viruses has what it would dream of having, which is universal connectivity so that it can get up to mischief.

But all the way through this, there's never been a reset. There's never been an opportunity or a moment or, I mean, any place where people said, okay, wait a minute. This is not going to get fixed until we get serious about completely rethinking the way our systems work. And we've watched Microsoft on this show over the last five years moving toward more secure things, more secure policies, more secure practices, introducing technologies which thwart this. But it's no really - it's not that different than the perennial spy versus spy, good guys versus bad guys problem where we're trying to stay a step ahead, and there always seems to be a rich flux of new problems that the good guys are introducing inadvertently into systems that bad guys can exploit.

And what we need, and I just sort of hit this frustration this morning, looking at all the problems and thinking about it, it's like, we need a reset. I mean, we need a - if we had a fundamental rethink of the way our systems operate such that we would have

containment. I mean, there's, like, little scratches around the fringe of this notion. We've talked about sandboxing and virtual machines and freezing the state of a machine so that when you reboot it - so that it doesn't - changes you make are not persistent. There are various ways of doing that.

And we need a fundamentally robust structure, rather than a fundamentally vulnerable structure. We have today a fundamentally vulnerable structure. And if there's one thing our listeners know from sort of, through osmosis, picking up the philosophy of security that you and I talk about endlessly every week, Leo, it's this notion of the weak link. I mean, in order for security to be - in our current model, every single piece of every link in the chain has to be perfect. One mistake in one link of the chain creates a vulnerability. So with our current model, where everything has to be correct, meaning we're at a huge - we the white hats are at a huge disadvantage because the systems are ungodly complex. I mean, just phenomenally complicated. And pieces are coming from every different direction.

Apple has traditionally had a little bit of an advantage because they produced the hardware. They sort of controlled a much more homogeneous platform, where Windows was always a much more heterogeneous sort of environment. Arguably that's changing a little bit over time. But anyway, I just, I look at this, and I think, this is not something we're going to win. If we continue down this road, we're not going to win. And I don't - you can't - we keep patching things. We keep putting layers of fixes.

I was hearing the other day that there are now ways around address space layout randomization, and now ways around DEP, data execution prevention. Those new technologies that were meant to, like, solve the problems, all they did, they were more patches, they were more, oh, look, let's scramble this up and make it a little more difficult. Well, all it does is it, sure, for a while it makes it more difficult, until the bad guys sit down and scratch their heads and say, oh, this is just another challenge for us, one that we can beat. And anyway, it's just - it's so clear that we're not winning. We're not. And I don't see how it can change.

There was an article I read which I ended up not adding to the top-of-the-show stories about a huge problem somewhere in Canada now, in Calgary, a medical facility that found a trojan inside of one of the machines that was a remote-controlled trojan, and the machine had personal, private scan results and medical records. They were forced to send out 4,700 pieces of email to their patients saying it may be that your records have been - got out of our control and are now available to bad guys on the Internet. And there was a quote later in this article where one of the executives - and this is not the first time they'd had problems. I mean, one of the executives said, yeah, every so often those guys find a way into our, you know, through our firewall.

Leo: Yeah, every once in a while.

Steve: And I'm thinking, no, they don't. Every so often one of your users clicks on a link in email or is browsing around somewhere they shouldn't be and does something. I mean, and here I'm not blaming the user. I'm just saying that we're requiring too much from our users. We're requiring an unreasonable amount of discipline because the architecture, the fundamental structure of these systems started out with no concern for security, and then it just became an arm's race, the bad guys versus the good guys, where we are fundamentally disadvantaged because to be secure we have to be perfect. And there's just too many opportunities to make a mistake. We only have to make one mistake out of a bazillion, and that creates an opportunity.

Leo: So you're saying it's essentially always going to be a lopsided battle. There is no way to achieve ultimate victory. And as long as there's...

Steve: Not with our current architecture.

Leo: And as long as there's incentive for the bad guys to crack, they're going to crack and eventually succeed.

Steve: Well, and incentives only growing over time.

Leo: Oh, absolutely. That's what we've seen change, really, is the massive incentive to do it.

Steve: Yes. Now Mom and Dad are doing their banking online because the banks don't want to have tellers staffed behind the windows. And, I mean, we saw the tremendous success of ATMs demonstrated that even the customers don't want to deal with the tellers. They just want their banking transactions done. So it's arguably a win-win unless you get bad software in your machine that watches you log on. And even on the fly, in seconds, software is now able to recognize what bank you're on, what protocol you're on, intercept that, grab your credentials, and move your money somewhere else. And then in the page that is returned to you, show you that your balance is still what you think it is, even though the page would have come back showing zero.

And this is happening to people. And it's just wrong. But I just - here we are, we look at this week after week after week, and talk about it, and nothing has changed. When you proposed the podcast to me five years ago in Vancouver, I thought, well, I don't know if we have enough to talk about.

Leo: Ha ha. Little did we know. And it's only going to get worse. Now, you're not the first person to propose an Internet reboot. In fact I know at Stanford there's a working group working on the next-generation Internet. And really they're thinking we start from scratch.

Steve: Well, it's not the Internet, though, Leo. It's our machines.

Leo: Well, it's protocols, it's everything, isn't it.

Steve: Yeah.

Leo: It's everything. And that's why it'll never happen, because there's too much legacy. And you can't break everything. You just cannot break- and there's no way to do this gradually. I guess there is. I don't know.

Steve: Well, we've been - no one, I don't see anyone who did, who took a timeout. Which is why I think maybe Microsoft, somewhere, in some corner, certainly they must, I mean, they're the one we keep blaming for all the problems. They're scurrying around, getting the blame for the defects that are arguably theirs except I defend them, yes, but I understand how hard this is. So maybe somewhere they've got some secret weapon project where they're trying to figure out how not to break compatibility with everything we've done, yet fundamentally change the way these systems work so that we don't have this weakest link principle. As long as there's a chain of interdependent things that all have to be perfect, such that one problem breaks everything, we're screwed, frankly.

Leo: It's true.

Steve: There's no other way to put it.

Leo: So if you Google "FIND GENI" - the word "FIND," which is an acronym for Future Internet Network Design, and "GENI," which is an acronym for Global Environment for Network Innovations - this is a four-year project which is, by the way, only three out of the four years have been completed, funding the BBN, the folks who invented the Internet, to create a clean slate approach to the Internet's underlying architecture. However, we haven't heard a thing about it since it was started in 2007, so I'm not sanguine that there's any progress being made.

Steve: Well, and the Internet is part of the problem, but it's certainly not...

Leo: Well, it says, "A new Internet could ultimately mean replacing networking equipment and rewriting software on computers, at a cost of billions of dollars." And I just don't think it's going to happen. "But clean-slate advocates say current piecemeal efforts to address security and other problems only create inefficiencies and open the network to," as you just said, "more risk."

Steve: Yeah.

Leo: So we do have to do something, but I don't know if we can.

Steve: Okay. Had to get that off my chest.

Leo: Good for you. No, I've been thinking along the same lines lately. I'm starting to wonder if we're not facing a global meltdown, frankly, of our IT infrastructure. Because the bad guys seem to be winning.

Steve: Well, it is full employment guarantee for anybody in the security field. I mean, it's - but it shouldn't be. It's just so much resource goes into this. And it's frustrating that the bad guys are able to have so much fun at our expense. But they are.

I did have, I always try to find a different take on SpinRite's helping people. And so when

I saw the subject line "SpinRite Helps Destroy Data," I thought, uh, what? And it said "(This is actually good thing.)" A listener of ours, Nate Woods, said, "Hi, Steve and Leo. I've been listening to your Security Now! podcast for a little over four months now." Oh, Nate, you've got some back listening.

Leo: Four months. Wow.

Steve: We've got four years behind that one. Anyway, he says, "And I'm really enjoying the computer basic principles series. I recently purchased SpinRite for maintenance on the drives in my ReadyNAS NV+ and found..."

Leo: I use that. That's a great NAS.

Steve: Oh, cool, "...and found a different use for it, as well. My brother recently dropped off an old computer he was getting rid of. He said, if I didn't want it, to make sure that the drive was cleaned and then get rid of it. I had no need for it, so used a Boot-Nuke to attempt to write ones and zeroes to the drive..." That's Darek's Boot and Nuke, which is a neat little program which you're able to boot, and it just does various types of drive wiping. He said, "...then write only zeroes to the whole drive. But the drive had bad sectors, preventing Boot and Nuke from even running. So I set SpinRite on Level 5 and then ran Boot and Nuke again. Which, after using SpinRite to clean up the drive's low-level sector problems, ran perfectly. It turned out all bad sectors were near the very beginning of the drive, where the OS and most programs were. I wasn't sure if anyone had ever used your program to help destroy data before, and I thought you'd be interested. Thanks for all both of you do. Nate Woods, Streamwood, Illinois."

Leo: Had you ever heard of such a thing?

Steve: No, but, well, something similar was done. We sold a lot of SpinRite when people were wanting to upgrade their FAT16 to FAT32 because remember that Microsoft, this was probably, what, 98 at some point, or Windows 95, Windows 98, somewhere they were - Microsoft was trying to migrate people from FAT16 to FAT32, or people wanting to migrate themselves to get smaller cluster sizes, to be able to run on larger drives and so forth. And Microsoft's own conversion utility would not function if there were any bad sectors on the drive. It would just stop and just say sorry, cannot convert you. And so the news spread around the PC community that, oh, get a copy of SpinRite, run it on the drive to fix your sectors, then you can run Microsoft converter to convert your drives from FAT16 to FAT32. So that's sort of similar to this.

Leo: All right. Are you ready to take a look now at hardware interrupts?

Steve: We're going to move forward, yes. I'm going to do a quick little review of where we've come. And then, building on everything we've done so far, talk about how computers have become very deft at doing many things at once.

Leo: Well, in fact, if you think about it, if you're a computer and you need to communicate to the outside world, you've got to find a way to do it while you're busy doing other stuff. We see, it's funny, we see it all the time where a computer gets tied up. On the Mac you call it "beachballing," where it's just - it's busy doing something, and it won't pay any attention to you.

Steve: Actually I'm going to talk about that, too. I've got it here in my notes. There's a simple thing that some Windows programmers don't quite get right involving something called the "Windows message loop" that is part of what we're going to cover today.

Leo: Great. All right, we're talking with Steve Gibson, the security guru, the head honcho at GRC.com. And we're going to continue on in Steve's series on building, kind of building a computer up from scratch by solving kind of the fundamental problems that you have to solve to make the computer work.

Steve: Yeah. I don't - you know that famous cartoon with a professor on the blackboard who's trying to balance his equation, and gets to a certain point, and he can't figure out how to make it go, and he says, "And then a miracle happens." And...

Leo: [Laughing] We're not going to have any miracles here.

Steve: Yes. I don't want at any point to just sort of wave my hand and say, oh, and just take my word for it; or, oh, that just kind of, you know, happens. There is nothing that requires we do that. We've got intelligent listeners who have been paying attention. The fact is, mysterious as the inner guts of computers are, they're just not that complicated. By choice it's where I live, programming in assembly language, because I like dealing with the truth. I mean, with the actual raw capability of the machine. And it's easy to sort of say, oh, that's just sort of beyond me. But the fact is it's not beyond any of us.

So this is the fourth in our series of sort of laying down a foundation of understanding what the exact operation, the true functioning of the computers that we're all using now in our daily lives. In the first installment we looked at the whole concept of having a programmable machine where we had just a block of memory that was addressable, word by word, and something called a program counter, which counted through the steps of the program, advancing word by word; and that these words that were read were broken down into fields of bits where one group of bits was the so-called "opcode," the operation code, which instructed how the computer would interpret the rest of the bits in the word, which for example might have been another address in memory telling, for example, the computer to add the contents of that memory address into an accumulator, or subtract them, or invert them, or rotate them, or a number of different things, maybe send them out to a peripheral device or read something from a peripheral device into the accumulator, so you had I/O, input/output instructions. And that's really, at that level, that's all that's going on.

Then in the second installment we introduced the notion of indirection, which is a very powerful concept, the concept of a pointer where, instead of the instruction directly specifying an address from which you did some transaction, the instruction specified an address which contained the address with which you did the transaction. In other words, it was a pointer to the actual target of the operation, a very powerful concept, which we

then advance to the next stage with our third installment, which was our discussion of having multiple registers and a stack.

The multiple registers gave the programmer more flexibility. He wasn't spending all of his time loading and storing things to and from memory using a single accumulator. If he had multiple accumulators, multiple registers, then it was possible to sort of have little temporary scratchpads. He could keep things in various registers as part of the algorithm he was doing.

And then we introduce the notion of a stack, which was the focus of the last episode in this series, as a hugely important and hugely powerful concept, the idea being that this was sort of a sequence-oriented storage concept, storage facility, where you could say, place a value on the stack, and then place another value on the stack, and then recover a value from the stack, and recover the prior value from the stack. The idea being that, as long as you put things on a stack and removed them in the reverse order, you're able to not worry about the immediate history of the stack as long as you have some discipline about always "popping," as is the jargon, popping something from the stack in the reverse order that you pushed some things on the stack. You were able to restore the contents that you had used the stack as a temporary storage to contain.

Well, that's an incredibly powerful idea because it allows us then, for example, to call to a subroutine, to jump to a subroutine to perform some work for us. And the subroutine knows that it's going to use, for example, a certain number, a certain subset, maybe, of the registers that's in this computer hardware. But it knows that, when we call the subroutine, we don't expect it to mess up the work we're doing. We just want some service from it. So it is able to use the stack to save the contents of the registers that it might be changing. And then, just before it returns to us, it restores them, it pops them, pops these values off the stack back into the registers so that, when it comes back to us, it's very much like nothing happened. I mean, we got a little work done. But if we were in the middle of doing things, then we were able to continue, trusting that subroutine to clean up after itself. And that's sort of the key concept.

So, moving forward, we have a computer which is able to do work. If we wrote some software, it could compute pi for as long as we allowed it to work on computing pi, or do polynomials, or do roots and cubes and sort of pure math computational stuff. Well, that's useful, except that computers really came into their own when they began interacting with the physical world, with so-called peripherals. We wanted them to interact with a teletype where we can type things in. We want them to be able to interact with storage facilities, where they're able to not only store things in their main core memory, but sort of have more like an archival storage, to magnetic tape or to disk drives. So that comes back to this notion of input/output devices, or input/output instructions, where we're able to say, take the contents of an accumulator and send that out to a device.

Well, in order to do that, since the computer can typically run vastly faster than the physical devices that it's using for input and output, it needs some way of pacing itself so that it's able to provide data to a device as the device is able to accept it. And it's similarly, going in the other direction, it's able to wait around for new data to arrive from a device. If we take the case, for example, of a keyboard, somehow the computer needs to know when we press a key and then to read the value of the key we pressed and accept it, store it somewhere, and then somehow be notified when we press another key.

Well, the original computers did this in a very awkward way, but it was all that they really had. There was an instruction, part of the input/output instructions, which would allow them to sense the readiness of a device to receive data or the availability of data

from a device. And so the computer would essentially, it would execute an instruction to read the - say that it was trying to read data from the keyboard. It would execute an instruction which would allow it to sense whether new data was available. And, if so, it would branch in the program to a series of instructions that would read the data and then reset that little sensor so that it could then start waiting for the sensor to again say, oh, we've got new data available, in which case it would read that and reset the sensor and so forth.

So essentially, the computer would loop, like in a tight little loop, just reading the status, checking to see if that status said there was new data available, and if not it would go back and read it again. Now, the problem is, the computer is completely occupied while that's happening. That is, while it's reading data, waiting for us to press keys one after the other, it can't get anything else done because it's spending all of its time sitting there just waiting for data to become available.

So some early programmer said, well, we could get some work done if we sort of, like, went away and did some work, and then checked to see if data was available and, if not, go back to kind of to what we were doing again. And so the idea would be, they would take responsibility for polling the status of the keyboard every so often. Now, that was clever, and it would work, in theory. But it required a great deal of discipline from the programmers who were wanting to get sort of work done on the side while being ready to accept data from the outside because, if they got too busy, for example, just doing computational work, then the computer would seem unresponsive to the user. Or maybe two keys would get pressed one after the other, and the computer would have missed the first one. It would only see the second one when it came back, if it didn't come back often enough. So people writing code like that had to make sure that they didn't spend - they never, didn't ever spend two months' time not checking to see if something new was available.

So the programmers complained to the hardware guys and said, okay, look, this is crazy. We can write code to keep things moving. But it's really hard to do that. There's got to be a better way. Well, what was invented was the concept of a hardware interrupt, which is an incredibly powerful and, as always when I start talking about "incredibly powerful," you know that what's going to follow is "and dangerous," I mean, with power comes responsibility. Like we were talking four weeks ago when we talked about pointers, and I said pointers are incredibly powerful; but, oh, boy, can they be trouble if you're not very careful with them.

Similarly, hardware interrupts are very powerful, but I would be - I would not be surprised to learn that, for example, Toyota has had a problem with that kind of power, because it's the way computers become asynchronous. If you didn't have any interruptions, if all you were doing was running code like computing pi, then the computer is entirely - what the computer does is entirely deterministic. You start it in the morning, and it starts working on pi. And eight hours later, if you were to stop it, then it's done a certain amount of work.

If the next morning you started it at the same time, and if you checked on it in exactly the same length of time later in the second day, it is doing exactly the same thing. It's in exactly the same place. It's been - every single thing would have been predetermined based on its starting state. The entire future was determined by the way it started because, with simple software, it's entirely deterministic.

Well, that immediately goes out the window as soon as you start interacting with the real world. What hardware interrupts allow is they allow an electrical signal of some sort, an electrical signal representing some sort of physical event, like a key being pressed on the

keyboard, to interrupt at any time the normal flow of instructions. So now, with a hardware interrupt system in this computer that we were just talking about, the main code, the main work that's being done, never has to check to see if a character is available on the keyboard. It doesn't have to explicitly check.

Instead, before it gets busy doing whatever it's going to do, it sets things up so that hardware, a hardware event that occurs can interrupt it anywhere it is, literally in between instructions. So what the hardware in the computer does is, as it's stepping its program counter, word by word, through memory, reading each instruction in turn and doing that, some additional logic is added to the hardware which, just as it finished executing an instruction, there's a hardware check to see if the interrupt signal is active. If not, it just proceeds as it normally would have to read the next instruction in turn and execute that. And again, after that instruction it - it takes no time to do this in hardware. So there's no overhead in testing this hardware interrupt line between every instruction. It happens at light-speed in the logic. The computer either moves forward; or, if the hardware interrupt is active, it suspends, essentially doesn't execute the next instruction it would have in sequence. Instead it does something different.

Now, what it does in detail is a function of sort of the generation of the hardware. For example, the PDP-8, my favorite machine to use as sort of an example of in the beginning, did have hardware interrupts. And in fact I used them in the programs I wrote for blinking the lights and playing games and things. What the PDP-8 did was, when a hardware interrupt occurred, wherever it was executing in main memory, it forced that next - the location of that next instruction it would have executed to instead be stored in location zero in main memory. And it changed the program counter to a one. So what that did was, suddenly the instruction at program location one got executed, and the computer followed the trail from there with the exact location where it had been interrupted stored in memory location zero. Of course, that allowed - storing it in zero allowed the computer to get back to, to return eventually to exactly where it was.

So this was a huge innovation. Suddenly you could set things up so that, for example, when a key is pressed, no matter what you're doing at the time, suddenly you are where you were, it's stored in location zero, and you start executing at location one. And the formal name for that location one is an interrupt service routine, or an ISR, as it's abbreviated. Interrupt service routine, meaning that that routine, that code is going to service this interruption that the computer has just experienced.

So what does it have to do? Well, we have no idea now where we were. We don't know what we were doing when we got interrupted. So now what we've introduced is nondeterministic computing, where real-time events occurring in any time change the flow of code through the computer. Well, if we want to, for example, read the data from a keyboard and store it in a buffer somewhere, we have to make sure that, almost like a physician that promises to do no harm, we have to make sure we don't change anything. That is, the computer was in the middle of work of some kind. But now we've got to use the computer's own resources to read the keyboard and figure out where we were in the buffer and store what we read in the next byte of the buffer. And so we - but then we need to return to what we were doing when we were so rudely interrupted with - but leave the machine in exactly the same condition as we found it.

So this is where the stack, brilliantly, comes in because, remember, it's this beautiful sort of flexible scratch pad, which as long as we pull things back from it in the reverse order we stored them, we get them all back. And what's cool is that the program we interrupt can be using the stack, too. We just need to make sure we put everything back the way it was. So our interrupt service routine knows that it's going to use some of the registers in the computer which were probably in use when it got - when this interrupt service

routine got invoked. So it pushes the values stored in those registers onto the stack in a certain careful sequence and says, okay, I have saved things. Then it reads the data from the keyboard, figures out where it needs to store it in the buffer, does so. And then it needs to clean up after itself. So what is to say is, it needs to restore the state of the machine to exactly what it was when it got interrupted. When it does...

Leo: Now, Jimmy has an interesting question in the chatroom.

Steve: Uh-huh?

Leo: Is this recursive? In other words, what happens when you interrupt an interrupt?

Steve: Ah, well, that's a very good point. Now, in this PDP-8 model, there is a problem, which is that, when the interrupt occurred, remember where we were got stored in location zero. So imagine that, while we were doing this work in this interrupt service routine, another interrupt were to occur. Well, what would happen is where we were would get stored in location zero, and we'd start executing at location one, like we said. Well, that's horrifying because we had already stored in location zero where we were when the first interrupt occurred. Now another one has happened, while we were still in our interrupt service routine, which wiped out the record of the first one.

Well, naturally there was a way, there are several mechanisms that come into play here. The first is that the hardware disables interrupts as part of its servicing of the interrupt. So even in the very earliest machine, like the PDP-8, they understood that there was a danger with the architecture that they had. So the act of the computer doing this interruption, storing where it was at location zero and then executing from location one, the act of it doing that disables further interrupts. So what that does is it prevents exactly the problem that I just stated of an interrupt occurring while we're still in the process of servicing an existing interrupt. So that prevents there being a problem with recursion.

Now, what happened as we moved forward in architectures is things naturally got more complicated. It was recognized, for example, that there were some peripherals that were high-speed, like a tape drive or a disk drive, which were generating data or requiring data at a much greater rate than, for example, a teletype or a keyboard. And so the notion of priorities of interrupt, interrupt priority came into being, where an interrupt would be serviced only if the interrupt - if any interrupt was in the process of being serviced, if the new interrupt coming in was a higher priority than the interrupt that we were in the middle of working on. So that's confusing unless you listen to it a few times.

But what that meant was that you could have a low priority interrupt assigned to the keyboard, and a higher priority interrupt assigned to the disk drive or the tape. And by agreement of the architecture and the programmers, a higher priority interrupt could interrupt the work being done by a lower priority interrupt. And that required a fancier architecture. For example, rather than having just a single location, like location zero, you might have a block of locations, call it 100, 101, 102, 103, 104, 105, that is, a location for each priority of interrupt. So that would allow - that meant that different priorities of interrupts stored their interrupted data in different locations to prevent a collision.

So again, mechanisms were created that allowed that. But the concept here, the main concept is that we've gone from a system where the starting state determines the entire future of the computer - which would be the case, for example, of us computing pi, where we're just following instructions, one after the other, ad infinitum - to a very different system where we're now able to respond in, essentially, in real-time to physical events happening in the real world.

And thanks to this ability for the flow of our instructions to be interrupted at any time, suddenly other code - completely unrelated, perhaps, to what we were doing - is being executed. It, that other code, it promises that when it's done it will put the machine exactly back to the way it was. Then it returns - because the location where we were has been stored in memory, it treats that like an indirect pointer, remember, so it's not the location but the contents of the location. So it does, for example, an indirect jump through location zero, where the location that we were executing has been stored, which actually takes us back to where we were. The program that was running has no idea that that all just happened.

If it were really fancy, it could sense that time had been lost. And in fact that's the way some of the anti-hacking technology that has existed, like anti-copy protection or copy defeating or anti-debugging. Because things like debuggers will be breaking the normal flow of execution, time is lost. And so it's technically possible for that software to sense that, wait a minute, some time has been lost between this instruction and the next one. But in terms of sort of its own sense of the machine and the contents of the registers and what's going on, it would have no knowledge that what had just gone on had happened. It was completely separate from it.

It could look around the computer and see evidence of things. For example, it might be keeping an eye on the buffer which is being filled. The fact that it's being filled is sort of magical to it. Every so often it looks at the input buffer, and it sees characters, new characters are appearing. And there's, like, a buffer pointer that says, here's how full the buffer is. And that's changing magically, sort of by itself. It's actually being done by an interrupt service routine reading those characters from the keyboard. But that activity no longer needs to be monitored that closely. The actual main program can just kind of keep an eye on it lazily and decide if there's enough characters yet for it to, like, get them all at once and go maybe move them somewhere else or do whatever it's doing.

So it's an incredibly powerful advance in the technology of the way computers work, which has been responsible for taking us sort of to the next level of allowing computers to interact with us in a very rich way.

Leo: It's amazing how this all pieces together.

Steve: Yeah, and not that tough.

Leo: No, it's not at all.

Steve: I mean, if our listeners listened carefully, they now understand what hardware interrupts are. And it's sort of, like, duh, I mean, that's how they ought to work, and that's how they do. That's all there is to it.

Leo: Well, I think the thing I like about this series is that it just - it's almost that it has to be this way. It's like, well, these are the problems you have to solve. And one by one you solve the problems. And it just kind of inevitably almost is this way. I guess there are other ways you could solve it. But this is such an easy, straightforward, logical way to do it. It's like there's a certain inevitability to it, is I guess what I'm saying.

Steve: Right, right. I think it follows logically. One concept follows logically from the next. I mean, some of these innovations were brilliant. The notion of a stack, the idea that you could have a stack pointer where you'd sort of give it a region of memory for it to worry about, and you could just kind of give it things. And then as long as you took them back in the reverse order, it would remove them from the stack in the reverse order that it had pushed them on the stack, creating this incredibly flexible facility for having, like, temporary storage in the computer. I mean, that was a brilliant addition.

Leo: I love the stack.

Steve: Yeah.

Leo: I just love it.

Steve: I did mention that I would talk a little bit about Windows when we were talking about applications freezing.

Leo: Right. The message event, or the event...

Steve: Yeah, the message loop, it's called.

Leo: Loop, right.

Steve: One of the first things a Windows programmer learns, and this may be less true now than it was, things like Visual Basic and some of the more recent languages sort of obscure the reality of what's going on inside of Windows. But the original concept with Windows was that programs would - that Windows itself, the Windows environment, before it was even an operating system, when it ran on top of DOS, it would hand programs tokens that were called "messages." It would hand them messages, which was just a value from zero to something which represented an event. And so it was also called the event loop sometimes, or the message loop.

And the idea was that the code that the programmer, a Windows programmer would write would ask Windows for the next event. And in doing so, it was turning control back to Windows. That is, it would say, give me the next event. Well, if there wasn't a next event that affected that window, then the software would just sort of wait. It would wait for another event to occur. And that was how you could have multiple windows on the screen that sort of seemed to all be alive at once, all active at once. That is, you could

grab it and move it, or you could type in one, you could do different things. In fact, only one was receiving these Windows events. And so as the application processed these events, displaying text or moving the window around or resizing itself, whatever the event told it to do, the window was animated.

Well, one of the things that could happen is some applications take time to do something. Maybe they're copying a file to the hard drive, or they're doing some encryption, for example, some serious number crunching that's going to take many seconds. Well, if a Windows programmer didn't really understand the way to program Windows, they might, upon receiving, like, an event saying do the work from a button, because buttons, when you press buttons or you select menu items, those just generate messages. Everything in Windows is a message. So the programmer might just go off and do that work, like start doing some big encryption project.

The problem is, while he's doing that, he's no longer asking Windows for any new messages. So when Windows sends a message saying, hey, someone's trying to drag you to a different location, that window won't move because - not because anything's broken, not because anything's hung, it's just that the programmer of that Windows application didn't consider that something else might still be going on while he's busy doing work. It's very much like this polling problem we just talked about where, if the program - if we didn't have interrupt system, and the program wasn't going and checking back often enough to see if a new character had been typed, it could miss some.

So the proper way to write a Windows application is with something called "multiple threads." And in fact the next topic we're going to cover, in two weeks, after next week's Q&A, I call it the "multiverse" - multicore, multiprocessing, multitasking, multithreading. It's multi everything, what is all of that multiness at all the various levels that it can occur. We now have enough of an understanding of how computers function to tackle that.

But the idea, briefly, of multiple threads, which we're going to cover in detail in two weeks, is that it's possible to sort of split off another execution ability from sort of within your program, so that you could still - you sort of split it so that you could still be asking Windows for any new messages while at the same time you are inside the same program, you're able to be doing the math or the long, time-consuming copy operation or whatever it was you were doing. And if programs are written that way, then they do not freeze when they're just busy doing something. If you don't write your program that way, even if there's nothing wrong with the program, it freezes. And people are so used to programs not freezing, that is, staying responsive to the user interface, that they immediately think something's broken.

And in fact this was such a problem that Microsoft added technology to sense whether a program was not responding to its message loop. And that's where we've seen, I don't know when it popped in, like maybe it was Windows 98, where Windows itself will put up "not responding" in the message bar, I mean, in the bar at the top of the window, as if we didn't know that it wasn't responding. It's like, yes, we know it's not responding because we're clicking on buttons, and nothing's happening. Again, nothing necessarily is broken. It's just that the program wasn't written with enough flexibility in mind.

Leo: Very interesting stuff. It's very helpful to understand the underlying principles so that when your programs go south, you know what's going on. It's not a mystery.

Steve: And again, there are reasons for all of this.

Leo: Steve is the greatest. You could find more of Steve Gibson, in fact, 241 episodes of Security Now! at GRC.com, including 16KB versions for the bandwidth-impaired and full transcripts for those who like to read along as they listen. GRC.com. We have the shows also at TWiT.tv/sn. And

in the next few weeks, probably next week, video of this show will be available on iTunes and on YouTube, as well.

Steve: Oh, yay, cool.

Leo: Yeah. We're slowly - we're doing about a show a week. We don't want to overload the system. But...

Steve: Well, and you have to have meetings, you know, Leo.

Leo: Oh, god. Before the show began, Steve and I - I was talk- because Steve has always said he regretted getting his company so big. At the most, how many employees did you have?

Steve: We had 23.

Leo: Oh, man.

Steve: And that was about 20 too many.

Leo: Well, what you've got now is three. You right-sized. I remember one of the pieces of advice you gave me when I was first starting the business is, don't get too big, and don't have too many meetings.

Steve: Yeah, I once discovered, years later, a GrandView outline. Remember GrandView? That was a really good outliner.

Leo: Yeah, yeah.

Steve: And it was an outline that I had written for a meeting that we had about our meetings. And I thought, my god, even our meetings were having meetings.

Leo: That's bad.

Steve: So, yeah.

Leo: That's bad, yeah. You have to have meetings as you get employees because...

Steve: Got to organize.

Leo: ...it's the only way you can communicate. You've got to do it. But it does - it's a time sink.

Steve: Yeah.

Leo: And I'd rather be on the air.

Steve: We'd rather have you on the air.

Leo: Well, it's just like you. You'd rather be programming.

Steve: I would.

Leo: Doing what you do best.

Steve: I really would.

Leo: So it's hard to have - but at the same time, if you want to have control of what you do, you have to have a business. You have to run your own business. This is difficult.

Steve: Yeah.

Leo: I need interrupts and polling instead of meetings. Actually that's what I get. You watch, what, as soon as we wrap the show up, I'll be interrupted. And I'll put them on the stack.

Steve Gibson is at GRC.com. Don't forget, too, that's where the SpinRite program is, everybody's favorite hard drive maintenance and recovery utility. Somebody just asked in the chatroom, what happens if the power goes out, as it did for him, when SpinRite's running? Is that bad?

Steve: SpinRite does everything it can about making sure that it is able to be safe and survive through a power outage. It's been tested for - back when SpinRite was being reviewed actively by reviewers, they'd pull the plug out several times and plug it back in and go, hey, nothing broke. It all works. It's like, yup, I spent a lot of time making it be safe.

Leo: Yup. So there you go. GRC.com. Also, don't forget, there's that great DNS Benchmark. There's Wizmo. There's so many great free programs there, too. GRC.com. Steve, we will see you next week. We'll have a Q&A episode. If you've got questions you'd like to ask Steve, if this has raised any questions in your mind or any security questions, go to GRC.com/feedback. There's a form there for you to ask your question. And maybe we'll use your question next week.

Steve: Okay, my friend. I'll talk to you then.

Leo: Thank you, Steve.

Steve: Bye bye.