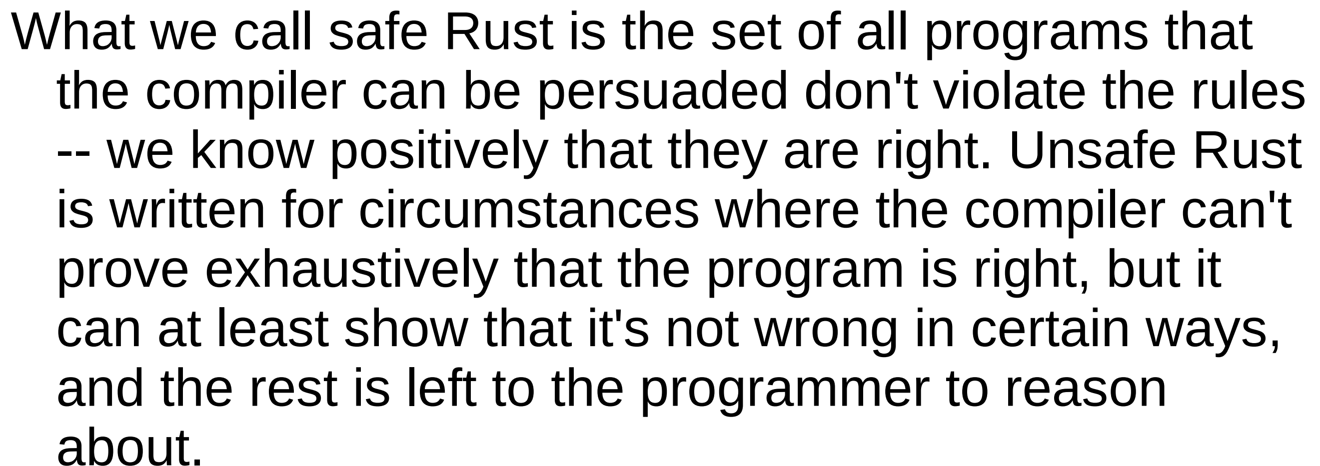
Rust for IOT

talks.edunham.net/lca2019
@qedunham
edunham on irc.mozilla.org

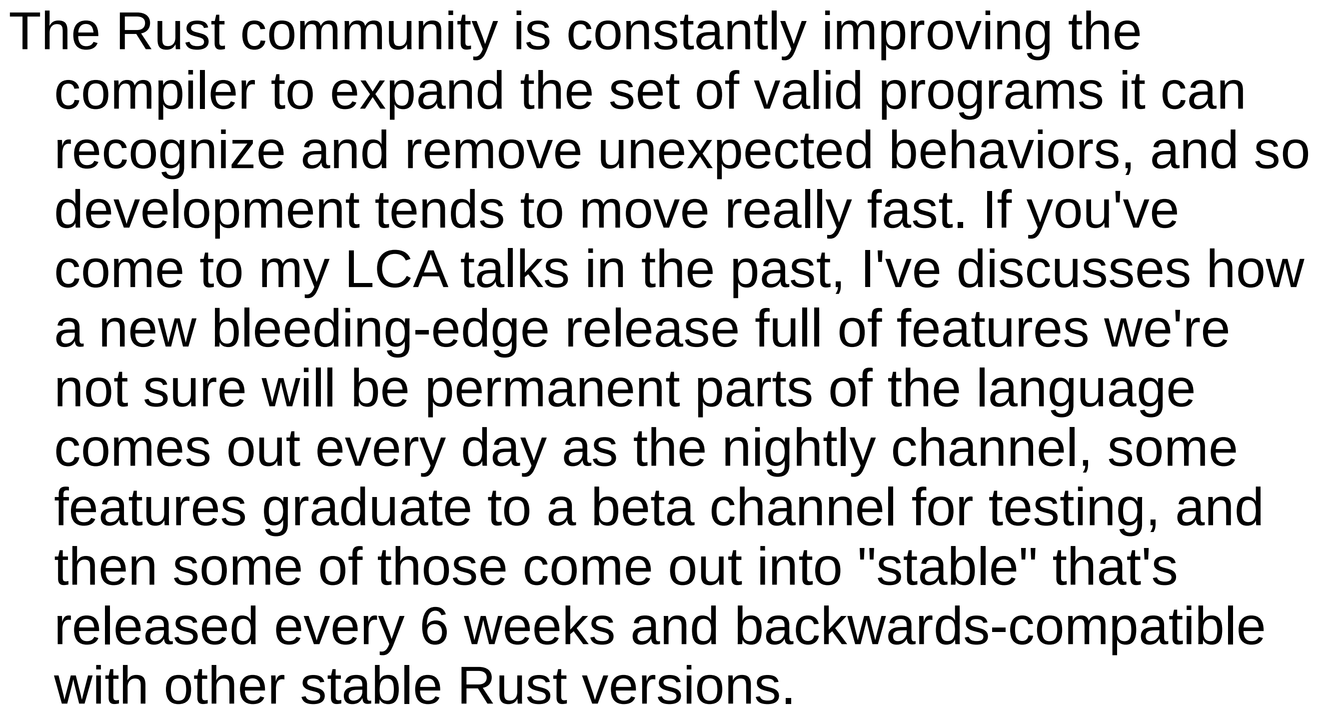
Hi! I hope you're here to talk about Rust and IOT with me. I've contributed to the Rust project in various ways for awhile, and done a variety of home automation and robotics stuff thoughout my career. You will not have heard of me from any prominent embedded Rust projects or the fantastic embedded Rust book, because those things are written by experts who know better than to try to fit any summary of the topic into 45 minutes. And that's just what I'm going to do.

When I first proposed this talk to LCA, there wasn't a central repository of all the scattered Rust-on-embedded-systems knowledge. between the proposal and today some fantastic resources have been published that I'll recommend to you as we go through. This frees me up to do a little exercise at the end that takes advantage of what makes a conference like this really special: the other humans in the room. In lieu of a regular Q&A session, at the end I'll ask everyone who wants to connect with others about their Rust project or idea to give us a quick sentence summary of what that idea is, and I have these Rust mascots for you to display to make it easy for others to find you through the conference.

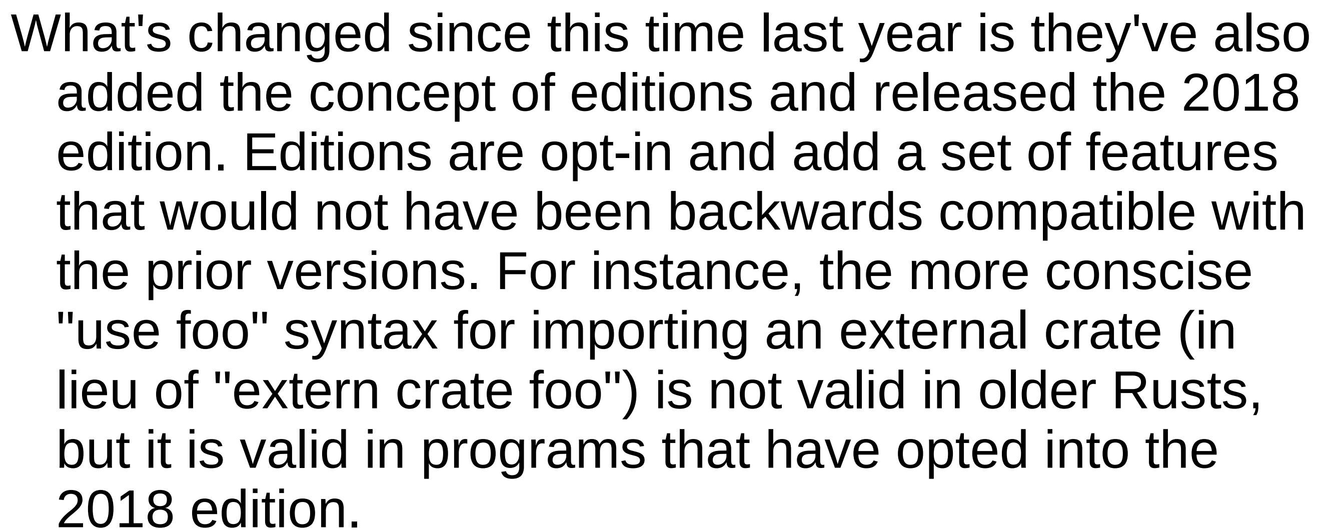
First things first, you've probably heard of the Rust language. It attacks the appparent dichotomies between human-readable versus fast on hardware, ergonomic to develop versus verifiable, by sweeping a lot of problems under the rug of the compiler and then buiding a compiler robust enough to address them. To write code free of certain classes of bug in any language, you need to follow a series of rules, and what sets Rust apart is that these rules live explicitly in the language specification and are enforced at compile time rather than just living in a book or in the programmers' memories.



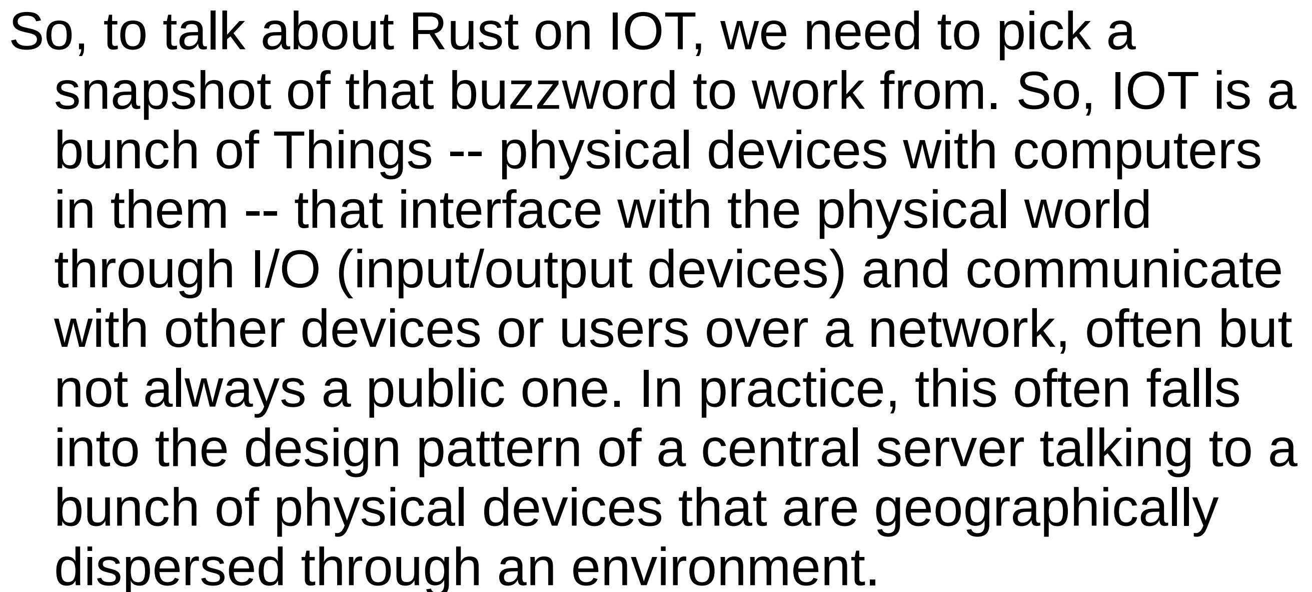
What we call safe Rust is the set of all programs that the compiler can be persuaded don't violate the rules -- we know positively that they are right. Unsafe Rust is written for circumstances where the compiler can't prove exhaustively that the program is right, but it can at least show that it's not wrong in certain ways, and the rest is left to the programmer to reason about.



The Rust community is constantly improving the compiler to expand the set of valid programs it can recognize and remove unexpected behaviors, and so development tends to move really fast. If you've come to my LCA talks in the past, I've discusses how a new bleeding-edge release full of features we're not sure will be permanent parts of the language comes out every day as the nightly channel, some features graduate to a beta channel for testing, and then some of those come out into "stable" that's released every 6 weeks and backwards-compatible with other stable Rust versions.

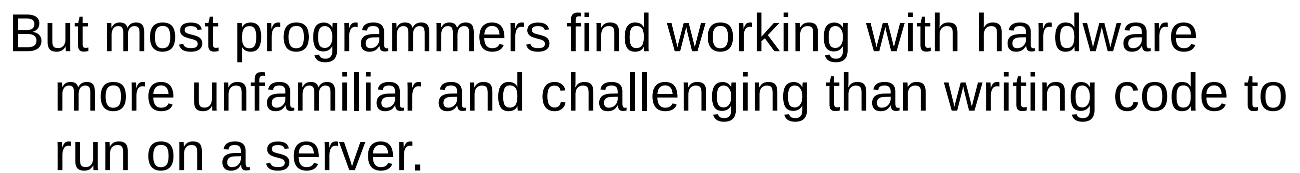


What's changed since this time last year is they've also added the concept of editions and released the 2018 edition. Editions are opt-in and add a set of features that would not have been backwards compatible with the prior versions. For instance, the more conscise "use foo" syntax for importing an external crate (in lieu of "extern crate foo") is not valid in older Rusts, but it is valid in programs that have opted into the 2018 edition.



So, to talk about Rust on IOT, we need to pick a snapshot of that buzzword to work from. So, IOT is a bunch of Things -- physical devices with computers in them -- that interface with the physical world through I/O (input/output devices) and communicate with other devices or users over a network, often but not always a public one. In practice, this often falls into the design pattern of a central server talking to a bunch of physical devices that are geographically dispersed through an environment.

So, we need software on both sides – software to run the server, and software to run the devices. Both can be written in Rust.



But most programmers find working with hardware more unfamiliar and challenging than writing code to run on a server.

That central server or database in many IOT design patterns is actually an extremely familiar use case if you've done web development before, and working on that end is a great place to start if you're learning a new programming language. There are generally more resources and mentors available to help you debug a web server problem than a hardware one.

Plus, writing code that runs on an operating system lets you avoid thinking about many hardware problems while also thinking about your code. The operating system can hide these problems from you, whereas your code has to deal with them directly if it runs without an operating system.

iot.mozilla.org/specification

github.com/mozilla-iot/wiki/wiki/Supported-Hardware

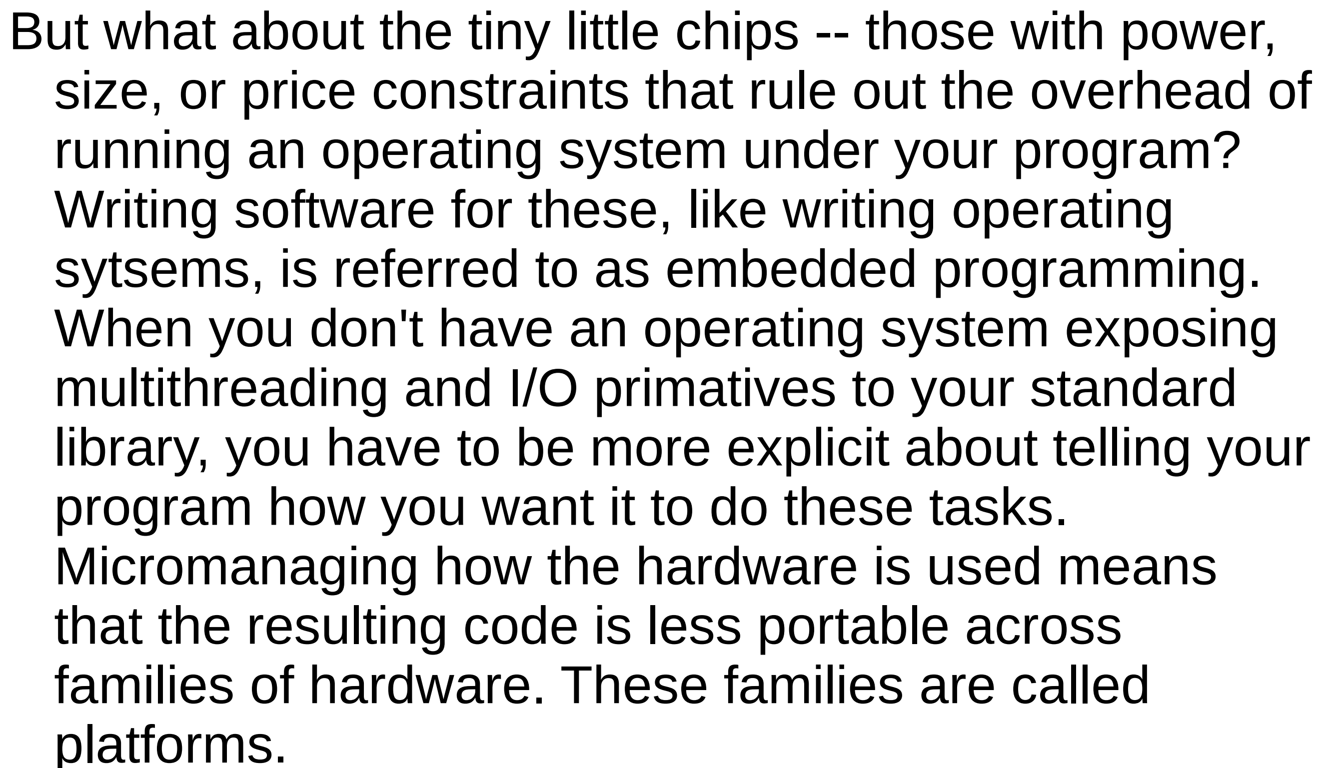
If you choose to have your devices talk to the server over a spec such as the Web Thing Standard (https://iot.mozilla.org/specification/), you can build on the work that others have done and potentially get IOT project started without having to do any embedded development at all.

The Mozilla Things project Rust example shows how to leverage the webthing crate to set up a server that, in their example, talks to a dimmable light and a humidity sensor that expose the WebThing API. If you go this route, check out the wiki (https://github.com/mozilla-iot/wiki/wiki/Supported-Hardware) for a list of supported devices that are likely to integrate well with the Web Thing server.

If you're interested in learning more about the Things project, Kathy Giori will be running a workshop in A3 from 3:50-5:30pm tomorrow.

github.com/wezm/linux-conf-au-2019-epaper-badge

- I point this out because when you run a program within an operating system, the standard library makes all kinds of useful guesses about what you want for multithreading and i/o. It’s better to avoid reimplementing these features when you can, because the standard library has been well tested over many years on a variety of platforms.
- As a rule of thumb, if a board is commonly used to run a Linux derivative, you can probably write Rust with the standard library to target it. A great example of this, hopefully here in the room right now, is Wesley Moore's epaper badge written in Rust, source at link. He runs a Linux derivative on a Raspberry Pi, which means a regular Rust program can interface with peripherals such as the epaper screen through the operating sytsem.

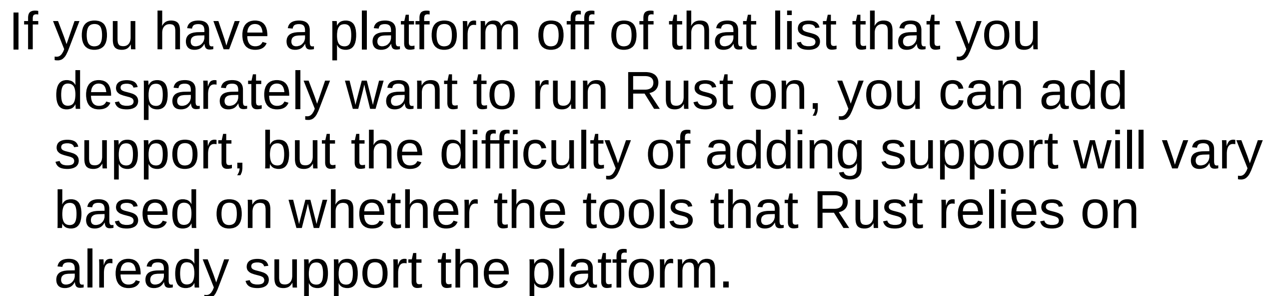


But what about the tiny little chips -- those with power, size, or price constraints that rule out the overhead of running an operating system under your program? Writing software for these, like writing operating sytsems, is referred to as embedded programming. When you don't have an operating system exposing multithreading and I/O primatives to your standard library, you have to be more explicit about telling your program how you want it to do these tasks. Micromanaging how the hardware is used means that the resulting code is less portable across families of hardware. These families are called platforms.

Fortunately, Rust already provides support for a variety of platforms out of the box.

forge.rust-lang.org/platform-support.html

- Rust categorizes its support for various platforms in tiers . Tier 1 is "guaranteed to work" Every change to Rust is tested on every tier 1 platform. Tier 2 is "guaranteed to build", in that each change is tested that code still builds, but the code is not always run for testing, often due to limited availability of appropriate hardware. Tier 3 "might work but no promises", for platforms on which we can’t always test that code will build.
- These different levels of testing correlate to whether you're likely to be the one discovering a bug that could have been tested for.



If you have a platform off of that list that you desparately want to run Rust on, you can add support, but the difficulty of adding support will vary based on whether the tools that Rust relies on already support the platform.

To understand the different processes you can use to add Rust support for a new platform, let’s quickly recap the stages that a Rust program goes through on its journey between the code that you edit and the code that the processor runs.

areweideyet.com

`cargo rustc -- --emit asm`

When you compile Rust code, it goes through several steps. First the compiler makes MIR, Rust's intermediate representation of the program. That's the level at which editor integrations hook in to warn you about errors as you write your code. Your IDE probably has Rust support; see the link.

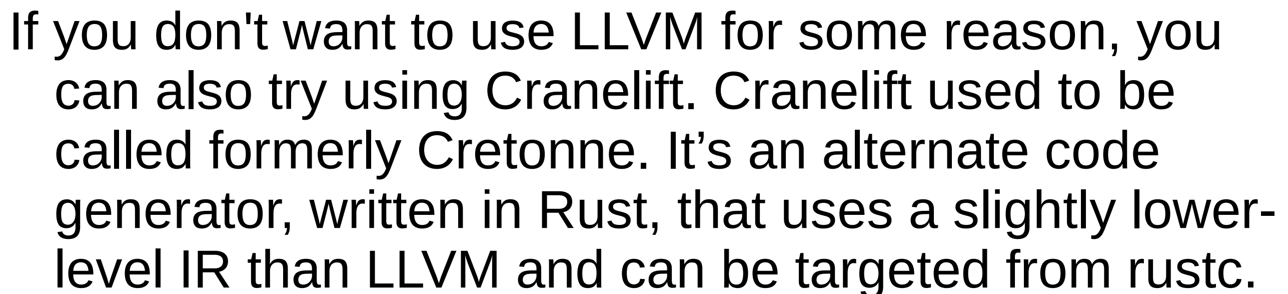
Then the MIR is usually transformed into LLVM IR, the Low-Level Virtual Machine Intermediate Representation, which goes to LLVM to be turned into assembly for the target architecture. LLVM optimizes the IR and converts it into a runnable program.

If you want to see what your code has turned into at each step of this process, you can pass `--emit asm` (or llvm-ir, or mir, or llvm-bc, etc) to Cargo, or for programs that aren't depending on other crates you can change "Run" to "Show Assembly" on play.rust-lang.org.

forge.rust-lang.org/platform-support.html

So, to figure out whether your embedded device of choice is supported by Rust, first check whether its architecture appears as a target in the platform support list. If it's not there, Google around to see whether LLVM can target it.

If you find that you’re working with a platform that has LLVM support but doesn’t have Rust support yet, contact the rust-embedded working group for guidance on how to add it to Rust!



If you don't want to use LLVM for some reason, you can also try using Cranelift. Cranelift used to be called formerly Cretonne. It’s an alternate code generator, written in Rust, that uses a slightly lower-level IR than LLVM and can be targeted from rustc.

Regardless of which code generator you prefer, If you want to add a new target to Rust, you'll have to add it to a code generator that Rust supports... and that's a lot of work.

github.com/thepowersgang/mrustc

https://github.com/emosenkis/esp-rs

Alternately, if you have a platform that you can write C for but that you can’t generate Rust code for, you can turn your Rust into C and then compile the C through the usual methods.

Eitan Mosenkis did this to get Rust on the esp8266 by compiling Rust to C with mrustc and then running that C through a toolchain capable of targeting the hardware at hand. The drawback of mrustc is that it doesn’t do all the borrow checking and verification that regular rustc does – it assumes the code that you pass into it is valid Rust. That can be fine if you run your code through normal rustc first, but it lets you introduce all kinds of errors if you don't.

- rust-embedded.github.io/book/interoperability/index.html
- https://crates.io/crates/bindgen

Finally, you might find yourself wanting to use some C code from an embedded Rust program, or some Rust code form an embedded C program.

If you find yourself in that situation, the Embedded Rust book goes into greater detail on interoperability between embedded Rust and embedded C. If you're looking to take advantage of stuff that's already in C like this, check it out.

If you're looking to integrate between C and Rust, you'll likely also want to consider using Bindgen to automatically generate FFI bindings.

doc.rust-lang.org/core/index.html

If you’re working without the standard library, where do primitive types and methods come from?

Even when opting out of using the standard library, most Rust programs will have the Core crate available. It doesn’t help out with i/o and multithreading the way std does, but it still exposes many of the types and macros that you’re used to. It’s well documented on the Rust site.

github.com/japaric/xargo

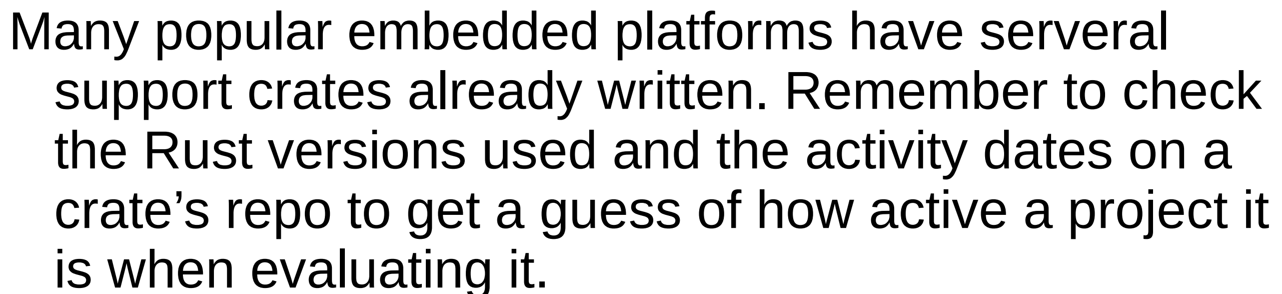
If Rust’s usual standard library won’t work for your hardware but you need some of its features that aren’t available in core, tooling is available to build and use custom versions of it.

Sometimes, you or your dependencies might want to reimplement pieces of the standard library in a way that's appropriate for the platforms you're targeting. In that case, consider using the tool xargo. If you're watching this talk after 2019, xargo's sysroot building features may already have been integrated into Cargo. Check the wiki.

github.com/rust-embedded/awesome-embedded-rust

So, after all that, you've found a device with a story for targeting it with Rust. Maybe that story is LLVM, maybe it's Cranelift, maybe it's leveraging a C toolchain.

Next, check out the awesome-embedded-rust repo (another new thing since I proposed this talk!) to identify the support crates and example code available for your platfom.



Many popular embedded platforms have serveral support crates already written. Remember to check the Rust versions used and the activity dates on a crate’s repo to get a guess of how active a project it is when evaluating it.

If you've only been doing regular Rust so far, you'll see a lot of new words popping up through these examples that might confuse you. Many of these concepts are common to embedded development in other languages, but they can seem intimidating when they all come at you at once.

So I’d like to go through a few of the first things you’ll need to be acquainted with when writing Rust to target an embedded device.

github.com/fudanchii/imtomu-rs

First, there's HAL everywhere. HAL stands for Hardware Abstraction Layer, which expresses the way that standard functions you'd normally get through the OS should be performed on a given CPU family.

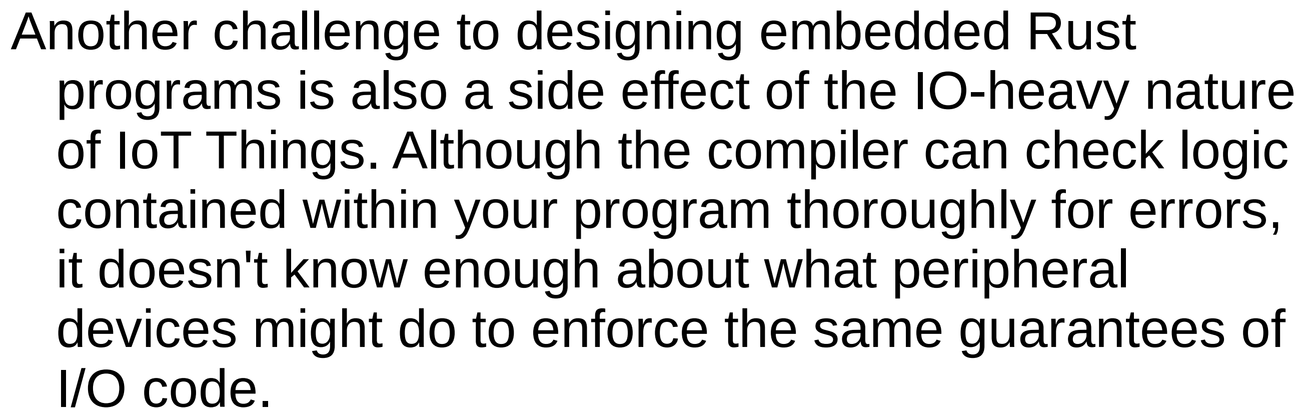
HAL crates for a given board are also a great place to look for examples on that board, since they're often written as a side effect of an expert trying to make the board do something else. For instance, the Tomu (Tom's Open Micro USB) is a device developed by members of our own LCA community, and its HAL crate includes blinking light and boot examples to get users started.

rust-embedded.github.io/book/start/semihosting.html

While an embedded board is designed to interface with inputs and outputs to its surrounding environment, it generally has little to no human-readable output.

If your program fails in some way as you're developing it, the board can maybe blink a light at you, but that's generally about it. To get useful debug information off of a program that runs on an embedded system, consider a technique known as semihosting.

Find a semihosting crate for your platform, and then you can use it to log messages from the embedded device to your computer or "host" machine's console. There's a good example of this in the Embedded Rust Book using the cortex-m-semihosting crate.



Another challenge to designing embedded Rust programs is also a side effect of the IO-heavy nature of IoT Things. Although the compiler can check logic contained within your program thoroughly for errors, it doesn't know enough about what peripheral devices might do to enforce the same guarantees of I/O code.

But, the principles behind writing memory-safe code still apply. For any thing -- in this case a peripheral -- we want either one mutable (writable) instance, or as many readable instances as we want.

rust-embedded.github.io/book/peripherals/singletons.html

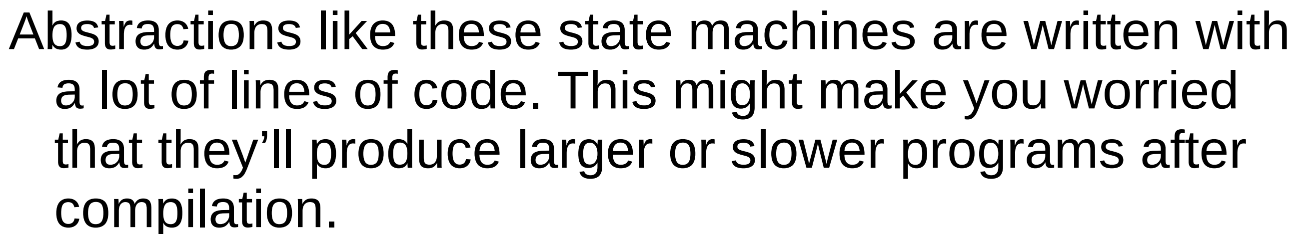
Using the singleton design pattern helps the compiler make sure that your code only instantiates each peripheral once. When you know there’s only one instance of your peripheral, the borrow checker can help you reason about whether it could ever be read from and written to simultaneously.

The alternative would be to use mutable global state to store the code that represents your peripheral, which is always unsafe because you can't make guarantees about whether any other part of the program will write it while you're trying to read it.

rust-embedded.github.io/book/static-guarantees/state-machines.html

Another way to improve the compiler's ability to reason about peripheral devices is to model them as state machines. A state machine is just a representation of which of several valid states it's permissible to transition between -- for instance if I had the states of wait to speak, get onstage, give intro, give talk, take questions, a state machine could codify the common sense that says it's not valid to go straight from waiting to speak to taking questions without giving the talk in between.

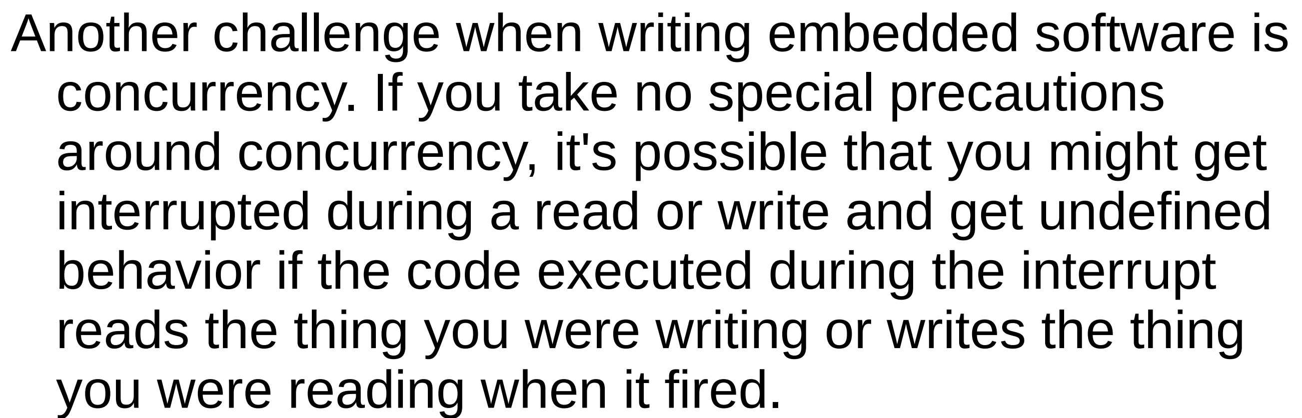
By expressing what the device is allowed to do, the compiler can check whether your code might ever ask it to do something illegal.



Abstractions like these state machines are written with a lot of lines of code. This might make you worried that they'll produce larger or slower programs after compilation.

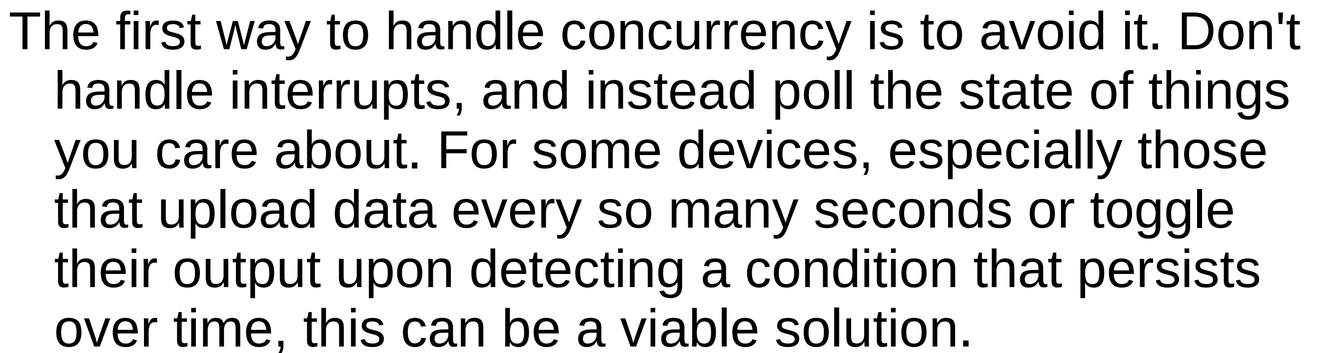
But they use a bunch of structs that don't actually hold any data to represent the states, which might seem like unnecessary bloat to your program. The great thing about the Rust compiler is that, after using the state machine to enforce borrowing rules, it can tell that the states will never be used at runtime, and they don't actually affect the size of your compiled program.

That's called zero-cost abstraction, and it's when you get the benefits of having the abstract idea of the states without any cost in the sense of an enlarged or slowed program.



Another challenge when writing embedded software is concurrency. If you take no special precautions around concurrency, it's possible that you might get interrupted during a read or write and get undefined behavior if the code executed during the interrupt reads the thing you were writing or writes the thing you were reading when it fired.

Instead, there are 3 ways you can tackle concurrency in your code.



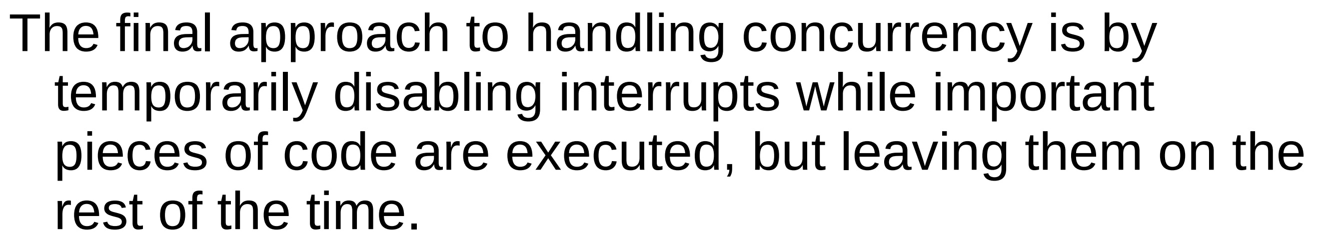
The first way to handle concurrency is to avoid it. Don't handle interrupts, and instead poll the state of things you care about. For some devices, especially those that upload data every so many seconds or toggle their output upon detecting a condition that persists over time, this can be a viable solution.

Avoiding concurrency altogether makes your programs easier to write and easier to debug, but it’s not always possible.

rust-embedded.github.io/book/concurrency/index.htm

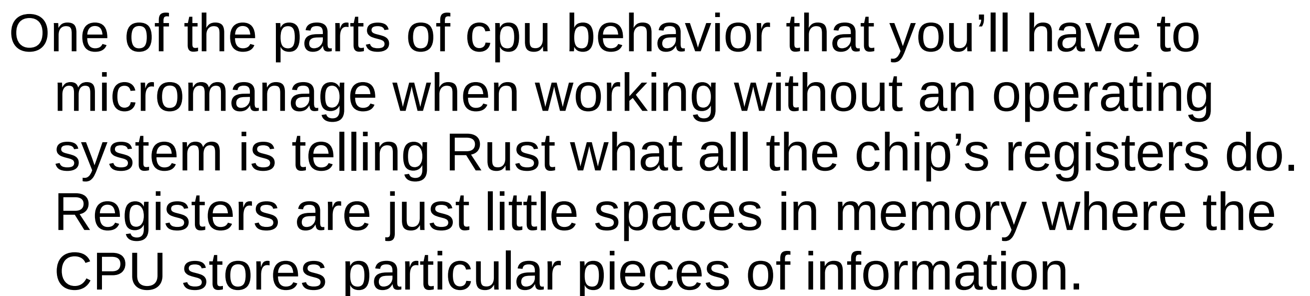
The second way to handle currency is by using atomic operations provided by the instruction set architecture of the embedded device that you're targeting. For instance, cortex-m3 offers atomic operations that will retry until success if they're interrupted.

If your board and HAL crate support atomics, they can allow you to perform important operations without the risk of getting interrupted. But what if you don’t have atomics?



The final approach to handling concurrency is by temporarily disabling interrupts while important pieces of code are executed, but leaving them on the rest of the time.

These areas of code in which interrupts are disabled are called critical sections. The syntax for critical sections will vary based on what the support crates for your architecture provide. They have the drawback that an interrupt happening to fall during a critical section may be missed, but that's less problematic than getting undefined behavior.



One of the parts of cpu behavior that you’ll have to micromanage when working without an operating system is telling Rust what all the chip’s registers do. Registers are just little spaces in memory where the CPU stores particular pieces of information.

Each has its own address. Some store specialized information that the CPU knows to look there for, and others are general-purpose and can hold whatever you tell them to.

Using the right specialized registers matters a lot, since some CPU instructions may use specific ones, and reading from the right register can tell you about the current state of the CPU and the last instructions it performed.

So one of your jobs as an embedded programmer is to tell Rust about what registers the target of your code has available.

crates.io/search?q=svd2rust

Rather than defining all the registers by hand, register definitions are generated by a tool called svd2rust, based on SVD (System View Description Format) files that are often provided by chip manufacturers.

If you search crates.io for svd2rust plus the name of the chip you're targeting, you can quickly tell whether someone else has done the work of producing the register definition for you. Otherwise, you can use the svd2rust crate to turn an SVD file provided by your chip's manufacturer into Rust code.

- github.com/redox-os/redox
- robigalia.org
- intermezzos.github.io

If you're interested in learning more about embedded Rust by reading it, the many operating systems implemented in Rust are a great place to start. The Redox OS is perhaps the best-known Rust operating system, and runs on nightly Rust (https://). Robogalia is working on improving the Rust ecosystem around the SeL4 microkernel.(https:///). Intermezzos (http:///) is a teaching operating system that comes with a book explaining itself, and is designed to help developers new to systems programming get their hands dirty.

- www.tockos.org
- Hail board
- imix board

The Tock operating system (https://www.tockos.org/) offers the Hail and imix open hardware development boards and focuses on improving the IOT ecosystem. If you want open hardware boards to customize for IOT purposes that come with working Rust code for them out of the box and an active community, Tock is an excellent resource.

- irc.mozilla.org: #rust-embedded
- Twitter: @rustembedded
- "embedonomicon"
- "debugonomicon"
- github.com/rust-embedded/wg
- rust-embedded.github.io/discovery
- github.com/rust-embedded/awesome-embedded-rust

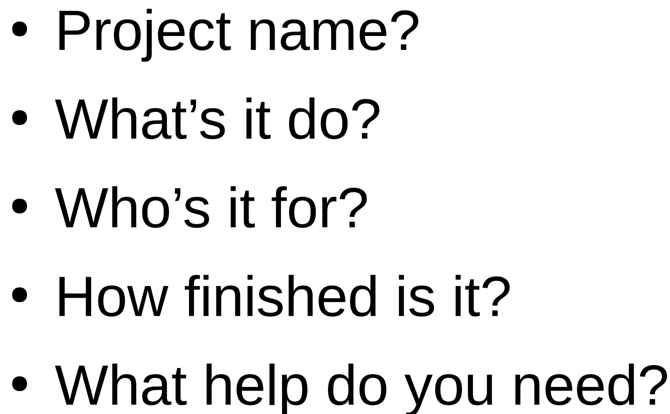
As I mentioned at the start, Rust's embedded resources have expanded dramatically even in the 10 months or so since I first proposed this talk to fill what, at the time, felt like a gap in documentation.

There's now an embedded working group, with an IRC channel and the handle @rustembedded on Twitter.

If you want a book that will teach you the basics of embedded microcontrollers while just happening to use Rust as a teaching language, consult the embedded discovery book online. For more contact info and the subteams within the Embedded working group, check out their GitHub.

For more experienced programmers, there're the *nomicon books as well.

There's the awesome-embedded-rust master list and there's a Showcase of embedded Rust projects currently under active development in the rust-embedded github org.



And one of the huge benefits of being at a conference like LCA is that, in addition to us speakers, there are hundreds of amazing attendees who are also experts in the topics we cover.

Rather than a Q&A session, I’d like to have a mini Rust BoF. I'd like to hear from those of you who have Rust IOT projects in progress or want to start them.

Come on up to the mic and tell us what your favorite Rust IOT project (or project you contribute to) does, what kind of help or feedback or adoption it's looking for, and what people should search to find its web presence!

For everybody that shares a project, I have a little Rust mascot that you can clip to your bag or your badge and wear for the rest of the conference as an indication that people interested in Rust should come and chat with you.

And everybody with questions -- these are the experts who'll have the time and inclination to help you

talks.edunham.net/lca2019
@qedunham
edunham on irc.mozilla.org