

Architecting a bridge

Axel T. J. Rohde, November 23rd, 2022

Have you ever thought about this?

Excellence comes at the cost of specialization.

Both concepts are so intertwined, in fact, that we often confuse their meaning. It requires us to take a step back and have a look at the bigger picture to realize that specialization is sometimes the ultimate cause of a certain category of problem. A failure to recognize this will usually make those issues grow larger, because they are instinctively assigned to even more specialized experts, who have a tendency to... ignore the bigger picture! And so forth...

These are the kind of impractical thoughts one may fancy during a coffee break, before returning to the inescapable zugzwang of daily business. But, hey! Lucky me is now in the luxurious situation of an intermezzo between switching jobs, and [Hacktoberfest](#) got me started! So there's no excuses: What would be a specific instance of said category of problem? One where I can get practical about it, without getting self-absorbed in intellectual masturbation?

Well, many come to mind, but among them there's one that has been a quite frustrating annoyance in almost every project I have worked on in the last 20+ years of my professional activity:

The need ...

... can be phrased in a few words:

*Communication between
a microcontroller's firmware
and the software running on a PC*

Wait! What?!

Aren't there already plenty of out-of-the-box solutions to address this?

What about USB?

Well this is what makes it so annoying: USB is so "powerful" that it requires an awful amount of expertise to tame it.

The kind of expertise that neither of the stakeholders at each endpoint possesses.

The specialists ...

... at both ends of said need: Who are they?

- [Bare-metal firmware](#) developers, who excel at interfacing tangible electronics. In their world, every single bit and clock cycle is relevant and they brilliantly master communication protocols like I2C, SPI and UART.
- [Application software](#) developers, who excel at abstracting hardware away. In their world, [high-level languages](#) and powerful [frameworks](#) are key to stay in control of complex applications that run on equipment that's evolving at the pace of Moore's law.

Despite their differences, both specialists share an important talent: stitching modular pieces of reusable software together to avoid reinventing the wheel when facing a recurring task.

The trench ...

... between those two worlds should therefore not exist, right?

Especially when the amazingly well-architected USB standard is addressing exactly this kind of scenario, right?

Well... yes and no.

USB is indeed excellent at providing [interoperability](#), but it is strongly biased towards a particular scenario, namely consumer market gadgets like a mouse, a keyboard or a mass-storage device, that need to work on any of the different models of personal computer manufactured in the past and in the future.

The magic of USB, though, quickly fades away as soon as your device isn't expected by any operating system; regardless of it matching any of the [USB Class Codes](#).

But wait! USB still has one trick up its sleeve:

The “rope bridge” ...

... of the good old [serial port](#).

It's in fact so old and has morphed through so many reincarnations, that it requires a bit of archeology to understand why those ports are named [tty](#) in UNIX.

And thanks to USB, the good old serial port is still among us, and most operating systems will more or less gracefully deal with those “Virtual COM ports”, thus allowing engineers of exotic devices to reliably transfer streams of bytes from a microcontroller to a modern computer.

The traps ...

... ahead are difficult to foresee.

Clinging to the attractive opportunity provided by serial communication seems to invariably lead into a characteristic [anti-pattern](#). And it ends up being the same anti-pattern, regardless whether we have arrived here enchanted by the universality of USB or the potential of a TCP/IP protocol stack made possible by the wonders of [uIP](#) and the like.

It usually starts off with a deep dive into the Internets, on a quest to find the “modular pieces of reusable code” that will unburden us from having to reinvent the wheel about how to manage the actual transmission of data that we need to exchange between the firmware of a device and the software of its host.

Surprisingly, [search results](#) seem to cluster around three quite clumsy options:

- Communicate like a PC:
 - e.g.: let the firmware run a web-server to expose a proper RESTful API !
- Communicate like a microcontroller:
 - e.g.: let the PC software talk proper fieldbus from the 90's; CAN bus is so OG !

That's only two of the three, but young engineers, eager as they are to embrace complexity in all its splendor, will typically fall in love with one of those two options. *(Been there, done that back in 2007: I was so proud to get [Dean Camera's USB stack](#) to work on a tiny AVR8 microcontroller. So cute!)*

The irony is, that none of these two options actually addresses the fundamental need. Sooner or later, and regardless of the choice, firmware and software developers will need to sit down and talk about how they agree to structure the information that they intend to exchange. At this point they either do so in the PC's terms or the microcontroller's terms, so one of the sides is likely to lose out. Even if they manage to overcome the cultural barrier, those options are inevitably sub-optimal at one of both ends of the line.

That's when the fatal allure of the third option comes into action.

The one that, old and young engineers alike, fall for:

- Let's meet in the middle!
Look how simple it is, if we agree that I send you six bytes and the third one means this and that, the fourth one such and such. Let's get this done already!

This third option is surprisingly feasible, and yes, it does indeed work.

So, where's the problem?

Down the rabbit hole

By now we are already deep down the rabbit hole.

And we are stuck there, because the third option usually works *well enough* to stay around.

I'll spare you the details about why this is an anti-pattern, because I guess it suffices to scratch the surface for you to imagine the obvious implications:

- Documentation
 - It can be [done nicely](#), but it requires a huge effort, because it can't be automated, nor delegated to anybody other than the original developers.
- Interoperability
 - Ideally only your [proprietary software](#) talks to your device, because otherwise you'll need to support your user community with [drivers](#) and/or [libraries](#) for each OS and generation thereof over the product life-time.
- Maintainability:
 - Would you dare to rename a parameter or rearrange the address-space of IDs? If so, then please update all mentions in the documentation and bump the protocol's version number, please, or you'll confuse everybody.
- Scalability
 - Will your initial protocol be able to keep up with future requirements?

How did we get here?

What pulls us towards the attractor of the anti-pattern described earlier?

The proximate cause seems pretty obvious: we didn't find any "modular piece of reusable code" that would fit our needs right out-of-the-box.

And [again](#):

Why didn't we find any?

Because existing approaches focus on how to get (more or less structured) data transmitted, and the actual specifics about the data being transmitted are up to each individual application.

And again:

Why do existing approaches focus on the networking internals instead of the actual need?

Because it's what specialists do!

Can we build a better bridge?

Is it possible to design sufficiently well architected “modular pieces of reusable code” that might save future generations from falling into the traps of this and other anti-patterns?

To be honest, I wasn't so sure when I started thinking about this challenge, but some proof-of-concepts later I believe there is, and therefore go through the pain of writing this manifesto in the hopes to collect sufficient criticism to mature [this thing](#) into something that is of sufficient value to be shared as a [stable](#) open-source project. *(I'm especially eager to find out whether I'm implementing [yet another](#) piece of middleware and was just bad at googling ;-)*

Bear with me, for a seemingly unrelated digression into ...

Engineering human language

Some engineering problems are much easier to address if we modify the way we talk about them. Note how arithmetic is much easier done with arabic, than roman numerals.

My proposal is based on the notion that firmware and software developers would find a more fruitful common ground talking in terms of [object-oriented-programming](#) instead of the data-transmission mentality that will draw them into the anti-pattern described earlier.

Even firmware developers coding in C feel comfortable thinking and talking in terms of OOP. And I would dare to say that even Product Managers would have [more of a chance](#).

They could all agree on how to structure the capabilities of their device into **features** and which **properties**, **commands** and **events** each should implement, without caring about the internals of how that gets transmitted over a serial connection of whatever kind.

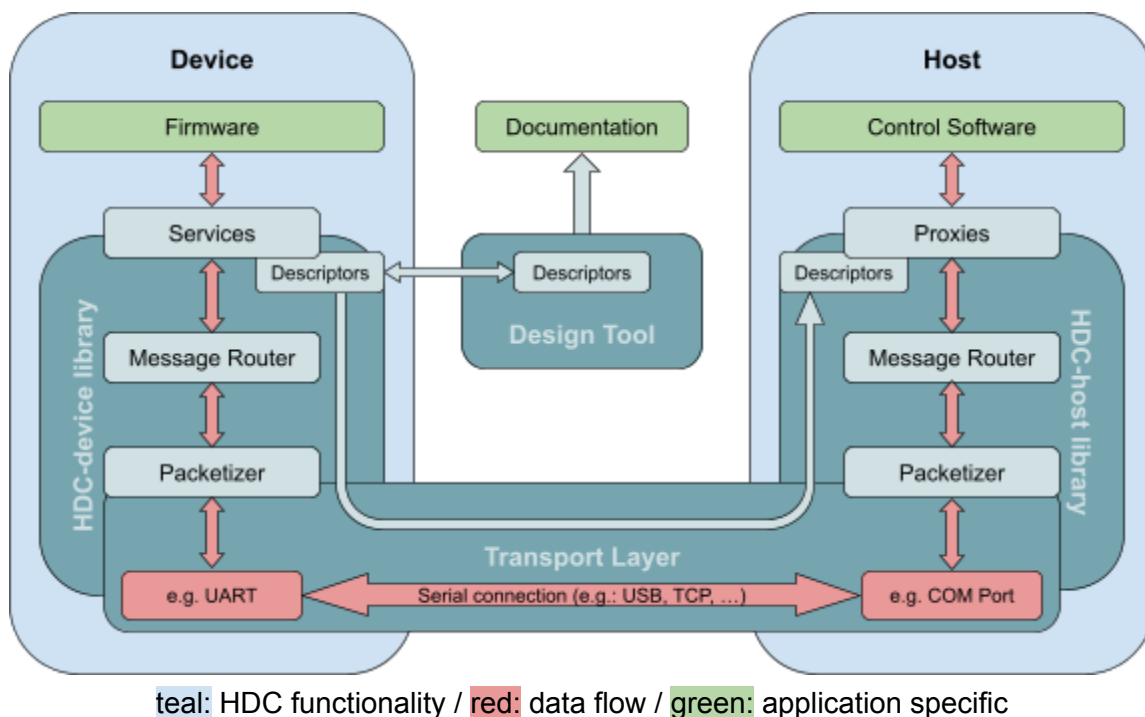
This approach is nothing new. It's simply an object-oriented API.

Fine, but how does it work?

HDC uses **descriptors** to declare said **features, properties, commands** and **events** that constitute the API of a device.

Descriptors can either be authored directly in the programming language of the device's firmware, or with the assistance of an interactive design tool. This tool can auto-generate the human-readable documentation for a device's API and also auto-generate the source-code that implements that API as exposed within the host. In some situations it is also useful that a host can generate the API proxies dynamically, even for unknown devices, by simply pulling its descriptors to instantiate the corresponding proxy API objects.

At the lowest level of abstraction, HDC takes care of how **request** and **reply messages** are exchanged, and how those are serialized and deserialized in form of **packets** for transmission over a **transport** layer. Those internals are handled by the HDC libraries and developers do not need to care about those anymore.



Feedback wanted!

The project is in pre-alpha stage and hosted on: <https://github.com/kiksotik/hdc>

There you'll find a draft of the HDC-spec, all of the source-code, including some demos and some ReadMe files.

Any kind of feedback, especially criticism, is very welcome!

You can either drop me an [email](#) or initiate a public discussion on the [issue tracker](#).