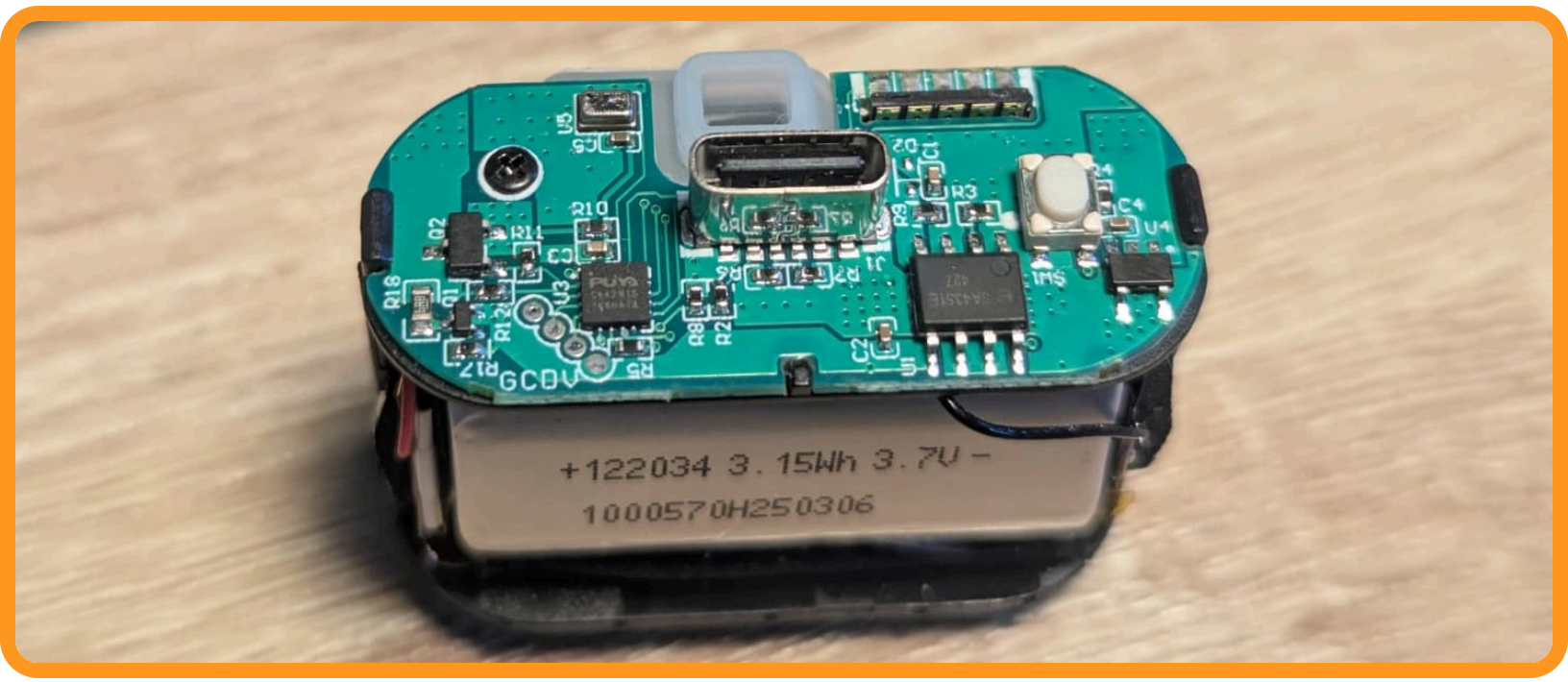
Hosting a WebSite on a Disposable Vape

2025-09-13 :: Bogdan Ionescu :: 6 min read (1267 words) :: source :: report issue

#programming #arm #tools #electronics



Preface

This article is *NOT* served from a web server running on a disposable vape. If you want to see the real deal, click here. The content is otherwise identical.

Background

For a couple of years now, I have been collecting disposable vapes from friends and family. Initially, I only salvaged the batteries for “future” projects (It’s not hoarding, I promise), but recently, disposable vapes have gotten more advanced. I wouldn’t want to be the lawyer who one day will have to argue how a device with USB C and a rechargeable battery can be classified as “disposable”. Thankfully, I don’t plan on pursuing law anytime soon.

Last year, I was tearing apart some of these fancier pacifiers for adults when I noticed something that caught my eye, instead of the expected black blob of goo hiding some ASIC (Application Specific Integrated Circuit) I see a little integrated circuit inscribed “PUYA”. I don’t blame you if this name doesn’t excite you as much it does me, most people have never heard of them. They are most well known for their

flash chips, but I first came across them after reading Jay Carlson's blog post about the cheapest flash microcontroller you can buy. They are quite capable little ARM Cortex-M0+ micros.

Over the past year I have collected quite a few of these PY32 based vapes, all of them from different models of vape from the same manufacturer. It's not my place to do free advertising for big tobacco, so I won't mention the brand I got it from, but if anyone who worked on designing them reads this, thanks for labeling the debug pins!

What are we working with

The chip is marked `PUYA C642F15`, which wasn't very helpful. I was pretty sure it was a `PY32F002A`, but after poking around with pyOCD, I noticed that the flash was 24k and we have 3k of RAM. The extra flash meant that it was more likely a `PY32F002B`, which is actually a very different chip.[1]

So here are the specs of a microcontroller so *bad*, it's basically disposable:

- 24MHz Coretex M0+
- 24KiB of Flash Storage
- 3KiB of Static RAM
- a few peripherals, none of which we will use.

You may look at those specs and think that it's not much to work with. I don't blame you, a 10y old phone can barely load google, and this is about 100x slower. I on the other hand see a *blazingly* fast web server.

Getting online

The idea of hosting a web server on a vape didn't come to me instantly. In fact, I have been playing around with them for a while, but after writing my post on semihosting, the penny dropped.

If you don't feel like reading that article, semihosting is basically syscalls for embedded ARM microcontrollers. You throw some values/pointers into some registers and call a breakpoint instruction. An attached debugger interprets the values in the registers and performs certain actions. Most people just use this to get some logs printed from the microcontroller, but they are actually bi-directional.

If you are older than me, you might remember a time before Wi-Fi and Ethernet, the dark ages, when you had to use dial-up modems to get online. You might also know

that the ghosts of those modems still linger all around us. Almost all USB serial devices actually emulate those modems: a 56k modem is just 57600 baud serial device. Data between some of these modems was transmitted using a protocol called SLIP (Serial Line Internet Protocol).[2]

This may not come as a surprise, but Linux (and with some tweaking even macOS) supports SLIP. The `slattach` utility can make any `/dev/tty*` send and receive IP packets. All we have to do is put the data down the wire in the right format and provide a virtual tty. This is actually easier than you might imagine, pyOCD can forward all semihosting though a telnet port. Then, we use `socat` to link that port to a virtual tty:

```
pyocd gdb -S -O semihost_console_type=telnet -T $(PORT) $(PYOCDFLAGS) &
socat PTY,link=$(TTY),raw,echo=0 TCP:localhost:$(PORT),nodelay &
sudo slattach -L -p slip -s 115200 $(TTY) &
sudo ip addr add 192.168.190.1 peer 192.168.190.2/24 dev sl0
sudo ip link set mtu 1500 up dev sl0
```

Ok, so we have a “modem”, but that’s hardly a web server. To actually talk TCP/IP, we need an IP stack. There are many choices, but I went with uIP because it’s pretty small, doesn’t require an RTOS, and it’s easy to port to other platforms. It also, helpfully, comes with a very minimal HTTP server example.

After porting the SLIP code to use semihosting, I had a working web server…half of the time. As with most highly optimised libraries, uIP was designed for 8 and 16-bit machines, which rarely have memory alignment requirements. On ARM however, if you dereference a `u16 *`, you better hope that address is even, or you’ll get an exception. The `uip_chksum` assumed `u16` alignment, but the script that creates the filesystem didn’t. I actually decided to modify a bit the structure of the filesystem to make it a bit more portable. This was my first time working with `perl` and I have to say, it’s quite well suited to this kind of task.

Blazingly fast

So how fast is a web server running on a disposable microcontroller. Well, initially, not very fast. Pings took ~1.5s with 50% packet loss and a simple page took over 20s to load. That’s so bad, it’s actually funny, and I kind of wanted to leave it there.

However, the problem was actually between the seat and the steering wheel the whole time. The first implementation read and wrote a single character at a time, which

had a massive overhead associated with it. I previously benchmarked semihosting on this device, and I was getting ~20KiB/s, but uIP's SLIP implementation was designed for very low memory devices, so it was serialising the data byte by byte. We have a whopping 3kiB of RAM to play with, so I added a ring buffer to cache reads from the host and feed them into the SLIP poll function. I also split writes in batches to allow for escaping.

Now this is what I call blazingly fast! Pings now take 20ms, no packet loss and a full page loads in about 160ms. This was using using almost all of the RAM, but I could also dial down the sizes of the buffer to have more than enough headroom to run other tasks. The project repo has everything set to a nice balance latency and RAM usage:

```
Memory region         Used Size  Region Size  %age Used
           FLASH:        5116 B        24 KB     20.82%
             RAM:        1380 B         3 KB     44.92%
```

For this blog however, I paid for none of the RAM, so I'll use all of the RAM.

As you may have noticed, we have just under 20kiB (80%) of storage space. That may not be enough to ship all of React, but as you can see, it's more than enough to host this entire blog post. And this is not just a static page server, you can run any server-side code you want, if you know C that is.

Just for fun, I added a json api endpoint to get the number of requests to the main page (since the last crash) and the unique ID of the microcontroller.

Resources

- Code for this project

1. While getting things together for this post, I came across this project that correctly identified these MCUs as PY32C642, which are pretty much identical to the 002B. ↩

2. Later modems used PPP (Point-to-Point Protocol) ↩