Fetching Live Balances on a Microntroller with WatchAddr (Code Companion #3)

Author	josh
Date	2019-11-20 21:04:50

Overview

One of the biggest perks of offline wallets (like paper wallets and hardware wallets) is the ability to store private keys away from prying eyes. In previous code companions, I talked about the challenges of building uBitAddr, a custom offline keypair generator that works with BCH, BTC, LTC, and ETH.

However, you probably want to keep tabs on your sweet, sweet offline savings. Many wallets have a "watch address" feature that lets you track the balances of an address without having the private key on hand for spending. In this project, I built a custom miniature "watch address" utility that runs on a wifi-enabled microcontroller!

Watching with Wifi

Fetching Data on an ESP8266 with MicroPython

The first part of my work involved connecting a wifi-enabled microprocessor to an API that would allow me to fetch live balance data. I wanted to be able to fetch data on addresses for my "big 4", BCH, BTC, LTC, and ETH. So I chose to connect to the bitcoin.com API for BCH, and BlockCypher for the other 3. Both provide an easy to use API that does not require an API key - an added layer of complexity that isn't necessary for fetching totally public blockchain data.

Connecting to a network on this platform is surprisingly straightforward, and that makes building internet-connected projects a treat:

```
    # Connect to the wifi access point configured in auth.py
    def connect_wifi(self):

        conn = network.WLAN(network.STA_IF)
        conn.active(True)

        conn.connect(auth.SSID, auth.PASS)

        return conn.active()
```

This code initializes a WLAN connection using the `network` module built into MicroPython and connects using the pre-configured network name (SSID) and password. This data is stored in a separate file called `auth.py` - anyone cloning this project will simply copy the sample and add their own configuration.

Building a Better API

Connecting directly to the Bitcoin.com/BlockCypher APIs directly worked pretty well for just balance data, but I ran into two fairly serious issues fast.

First, connecting via HTTP isn't secure or private - a man-in-the-middle attack could steal the user's addresses. This isn't a security issue per-se, as blockchain addresses are public data and can't be used to steal funds. However, it does compromise the privacy of the user who might not want others on the network to discover their addresses and therefore wallet balances!

Second, it turns out that the buffer for TLS (HTTPS) connections on the ESP8266 are *very small*, around 5KB. Fetching balance and price data in JSON format (with a bunch of stuff I didn't need) overflowed the buffer every time! The Cryptonator price API doesn't support HTTP at all, so I needed to kill two birds with one stone and find a fix for this.

My solution was to essentially create my own API *on top* of the balance and price APIs I wanted to use. This "proxy" fetches the balance and price data and digests it down to only the very minimum data I need, the address balance and current USD value. I return this data in a comma-delimited string - no fuss and a very small data size.

```
            # Fetch the data in multiple threads to reduce IO latency
            data = {}
            t_bal = threading.Thread(target=self.fetch_bal, args=(address,
currency.upper(), data))
            t_price = threading.Thread(target=self.fetch_price, args=(address,
currency.upper(), data))

            t_bal.start()
            t_price.start()
            t_bal.join()
            t_price.join()

            bal = data["bal"]
            price = data["price"]
            usd = bal * price

            # Return the data in the form of a comma-separated list
            response = "{0:.8f},{1:.2f}".format(bal,usd)
            return response
```

There's two functions in `api/watchaddr.py` not shown here that fetch the desired data from the API endpoints, parse, and return. This main code calls those functions and formats the data in the very basic string format before returning the response.

There's also a fun optimization I added here: I'm fetching the data in two separate threads. Python doesn't have "true" multithreading due to the Global Interpreter Lock, but it still works great for IO bound operations like waiting for a network response!

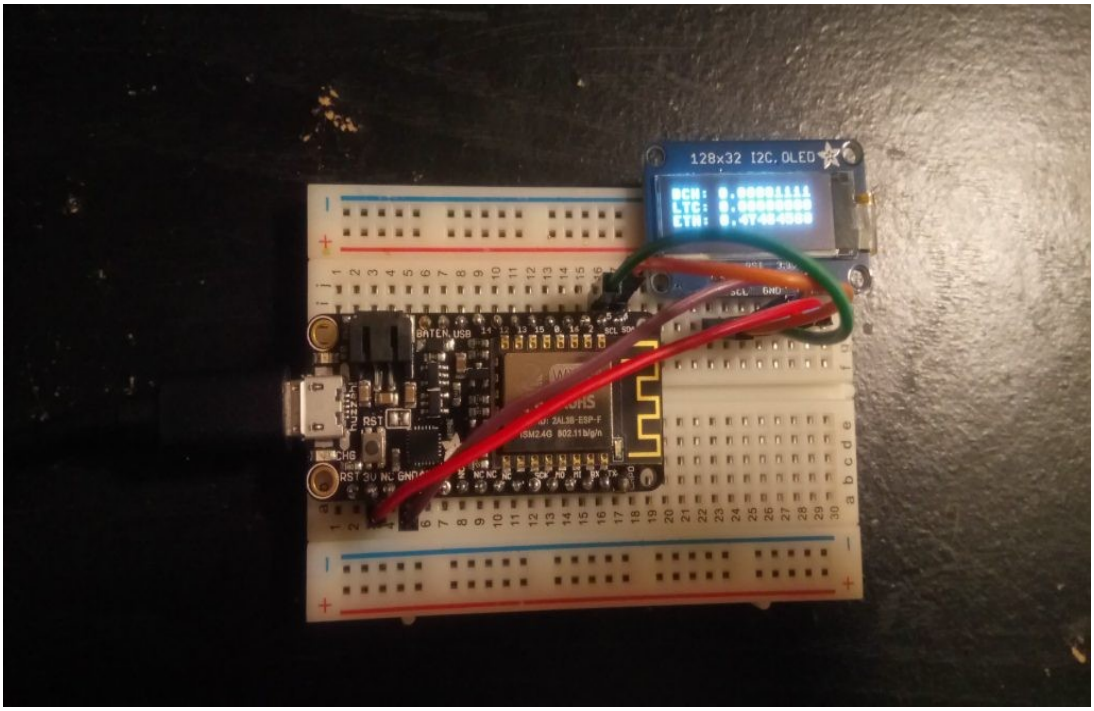
Screen Time!

Fetching the data and building a proxy is fun and all, but the ultimate goal is to run this off of a battery and not just off a USB cable connected to my laptop. So, I needed some way to display the information that wasn't writing to my laptop's console. In comes the delightfully small and powerful Adafruit OLED display.

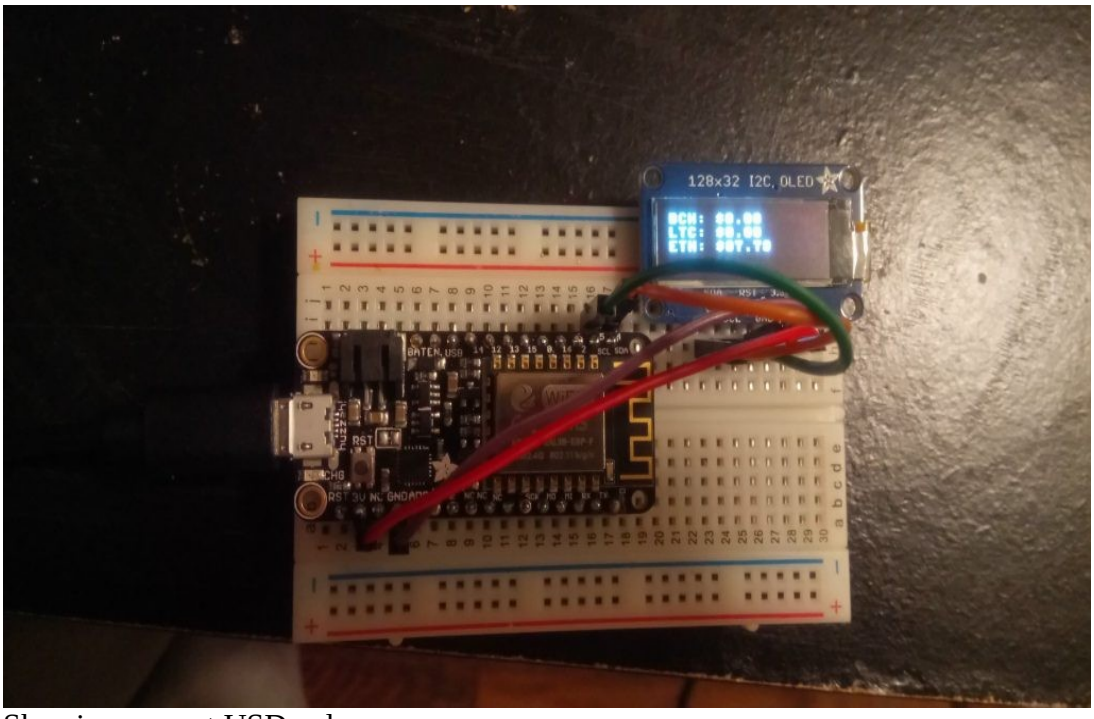
This screen is more robust than the character screen I've used in past projects - it can display pixels! Fortunately it does come with a simple API for writing text which is all we need here. This screen is wired up to communicate over the i2c serial protocol, which I find nice and simple to use. I also have an SPI version of the screen I would like to add support for as well, and it would be easy to wire up a character LCD.

```
    # Initialize the OLED screen for display
    def init_oled(self):

        i2c = machine.I2C(-1, machine.Pin(5), machine.Pin(4))
        self.oled = ssd1306.SSD1306_I2C(128, 32, i2c)
        self.oled_line = 0

    # Define a flexible display function
    # This can simply print to serial or output to a peripheral
    def output_data(self, data):
        if self.output == self.OUTPUT_DISPLAY:
            self.oled.text(data, 0, self.oled_line)
            self.oled.show()
            if self.oled_line == 30:
                self.oled_line = 0
            else:
                self.oled_line = self.oled_line + 10
        else:
            print(data)
```

These functions initialize the OLED screen and allow output of up to 3 rows of text (all that will fit on the screen).



Showing current crypto balances



Showing current USD value

Offline Addresses, Online Watching

This was another very fun project to build, and I'm proud of it! I love tinkering, and working with microcontrollers has been an exciting new avenue for learning and building interesting projects.

This utility is a nice companion for uBitAddr, so someone making offline wallets can keep tabs on their secure savings in a small, lightweight package. And as always, I think looking at this code can help folks understand more about how cryptocurrencies and software work.