Offline Address Generation with uBitAddr (Code Companion #1)

Author	josh
Date	2019-08-13 14:22:45

This is a new series I'm introducing called "Code Companion". These articles and videos highlight code projects of mine that are related to Bitcoin & cryptocurrencies. These projects can be helpful in understanding technical blockchain concepts. Explore and enjoy!

Overview

One of the most secure ways to store Bitcoin and other cryptocurrencies for the long term is to use was is called *cold storage*. This means storing the private keys that secure your funds in some manner that doesn't allow access by a networked computer. A popular method of implementing cold storage is to use a technology that long predates computing and the Bitcoin network - *paper*. A paper wallet is simply a Bitcoin address and it's private key written down on paper, usually in an exportable format such as *WIF (Wallet Import Format).*

An even more secure way to approach offline storage is to *generate* the private key and address offline. Generating and storing keys offline protects the owner from malware that might snoop the keys and send them off to a thief.

My project *uBitAddr* (pronounced MicroBitAddr) allows the generation of private keys and addresses completely offline using Adafruit M4 microprocessors wired up to an LCD screen or mini thermal receipt printer.

uBitAddr running on the Adafruit ItsyBitsy M4 with a character LCD & backpack as the output

uBitAddr running on the Adafruit Grand Central M4 with a mini thermal receipt printer as the output

How uBitAddr Works

Custom CircuitPython Module

The first big challenge in this project was getting access to the required cryptographic primitives on a microcontroller platform. Generating a basic Bitcoin address requires *elliptic curve cryptography* (secp256k1 specifically), and two different *hashing algorithms* (SHA-256 and RIPEMD160). Good implementations of these algorithms are widely available on desktop PC's and mobile phones for a variety of programming languages, but aren't readily available for microprocessors like this.

I needed to get access to good cryptographic implementations to make this work! Since I wanted to do the bulk of coding in CircuitPython, I could have found pure-Python implementations of these algorithms. However, I felt this was resource inefficient and wouldn't work on all but the most robust boards. Instead, I opted to write custom extension for CircuitPython itself using this guide from Dave Astels and some help from ladyada and Dan on Github. - thanks everyone!

For my project, I needed to use the M4 line of microcontrollers from Adafruit (using the Atmel Samd51 processor). I needed to use a processor at least this powerful to fit the compiled CircuitPython distribution, as the M0's are simply too small to support these cryptographic algorithms. Fortunately, my goal of supporting a few different controllers and not just one big powerful one still worked out - the M4 line features a bunch of different shapes and sizes, from the ItsyBitsy to the Grand Central!

For cryptographic primitives, I ended up porting over code from the popular open-source hardware wallet Trezor. The Trezor crypto libraries had everything I needed to do efficient elliptic curve crypto and hashing, and even base58 encoding for the final address and WIF private key. This code is designed to be efficient and lightweight for embedded platforms. A big thanks to Trezor's contributors for this code!

In terms of the my implementation, the most important code in this extension is features in `src/module/shared-module/bitaddr/__init__.c`. This code follows several steps to generate an address and exportable private key from start to finish:

1. Take entropy from the Python side (could be any good source, that's up to the module user)
2. Hash the entropy and generate the raw private key
3. Calculate the uncompressed public key from the private key
4. Use the Bitcoin base58 check algorithm to hash and encode the address from the public key
5. Use the WIF algorithm to encode an exportable private key

As of this writing, that function looks like this. I try to keep projects moving forward so this may change slightly in the future:

```
 // Define functions that implement the Python API
void shared_modules_bitaddr_get_address_privkey(unsigned char* address,
unsigned char* privkey, const char* entropy_privkey, const char*
entropy_ecdsa)
{
      // Init the random32 for rand.h and ecdsa.h functions
      // The random function is only needed for curve_to_jacobian - needs a
random k value
      // It will only be called once for address generation, so we'll use true
entropy
      // To "seed" random32's PRNG without causing problems
      unsigned char seed_entropy[SHA256_DIGEST_LENGTH];
      sha256_Raw((uint8_t*) entropy_ecdsa, strlen(entropy_ecdsa), (uint8_t*)
seed_entropy);
      init_random32(seed_entropy);


      // Generate the private key from some entropy
      // Then generate the public key from the private key
      unsigned char privkey_raw[SHA256_DIGEST_LENGTH];
      privkey_from_entropy(entropy_privkey, privkey_raw);

      unsigned char pubkey[PUBKEY_65_LENGTH];
      pubkey_from_privkey(privkey_raw, pubkey);

      // Generate the address from the public key
      // This address will use the legacy base58check encoding valid
      // in both BTC and BCH
      address_from_pubkey(pubkey, address);

      // Convert the private key to WIF format for export
      privkey_wif_from_raw(privkey_raw, privkey);
```

```
}
```

The next major component of this is creating a binding to the Python side in [src/module/shared-bindings/bitaddr/__init__.c](). This is a bit more complicated but the general idea is to take two sources of *entropy* (randomness) from the module user for the private key and ECDSA k value. I did this to allow flexibility - the module user can use a built in CRNG, accelerometer, diceware, etc. to generate randomness as they choose. Right now the Python code I wrote uses the convenient CRNG built in to all M4 boards.

Next, the C function `shared_modules_bitaddr_get_address_privkey` from `shared-module/__init__.c` is called. The return values (passed in to the function as pointers) are placed in a Python tuple and returned to the caller on the Python side.

```
//| .. function:: get_address
//|
//|   Returns a Bitcoin or Bitcoin Cash Legacy Address
//|
const size_t ADDRESS_STR_LENGTH = 40;
const size_t PRIVKEY_STR_LENGTH = 70;

STATIC mp_obj_t bitaddr_get_address(mp_obj_t entropy_privkey, mp_obj_t
entropy_ecdsa) {

      // Convert entropy args needed for secure address generation
      const char* entropy_privkey_char = mp_obj_str_get_str(entropy_privkey);
      const char* entropy_ecdsa_char = mp_obj_str_get_str(entropy_ecdsa);

      // Create an address cstring long enough to fit any Bitcoin address
      unsigned char address[ADDRESS_STR_LENGTH];
      unsigned char privkey[PRIVKEY_STR_LENGTH];
      shared_modules_bitaddr_get_address_privkey(address, privkey,
entropy_privkey_char, entropy_ecdsa_char);

      // make the return value
      mp_obj_tuple_t *addr_key= MP_OBJ_TO_PTR(mp_obj_new_tuple(2, NULL));
      addr_key -> items[0] = mp_obj_new_str((char*) address,
ADDRESS_STR_LENGTH);
      addr_key -> items[1] = mp_obj_new_str((char*) privkey,
PRIVKEY_STR_LENGTH);

      return addr_key;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_2(bitaddr_get_address_obj,
bitaddr_get_address);
```

Finally, I built this for at least a few of the boards I wanted to support. I tested this code on an ItsyBitsy M4 and Grand Central M4, but it should also work on the Metro M4 or other releases. They're all the same processor and seem to all have built in cryptographic number generation. So cool!

The Core CircuitPython Code

Thanks to this module, generating an address on the Python side ends up super simple! I generate a new address and private key on demand by specifying some entropy from the built in CRNG and store the results from the returned tuple:

```
    # Get entropy based on the desired source
    def get_entropy_str(self):

        if self.entropy_source == self.ENTROPY_CRNG:
            return str(os.urandom(32))
        else:
            raise Exception("No sufficient entropy source specified")

    # Generate address and private key
    def generate_address_privkey(self):

        address, privkey = bitaddr.get_address(self.get_entropy_str(),
self.get_entropy_str())

        return (address, privkey)
```

In the future it may be possible to get entropy from another source - like an accelerometer wired to the board, or a keypad input for diceware key generation.

Wiring up Outputs

The final step was to offer a few options for viewing the generated address and private key. I have an option for serial import, so it's of course possible to send your address and private key directly to a computer. However, since the whole purpose of this project is offline generation and storage, it's obvious I needed some offline output options! I opted for character LCD display and thermal printing.

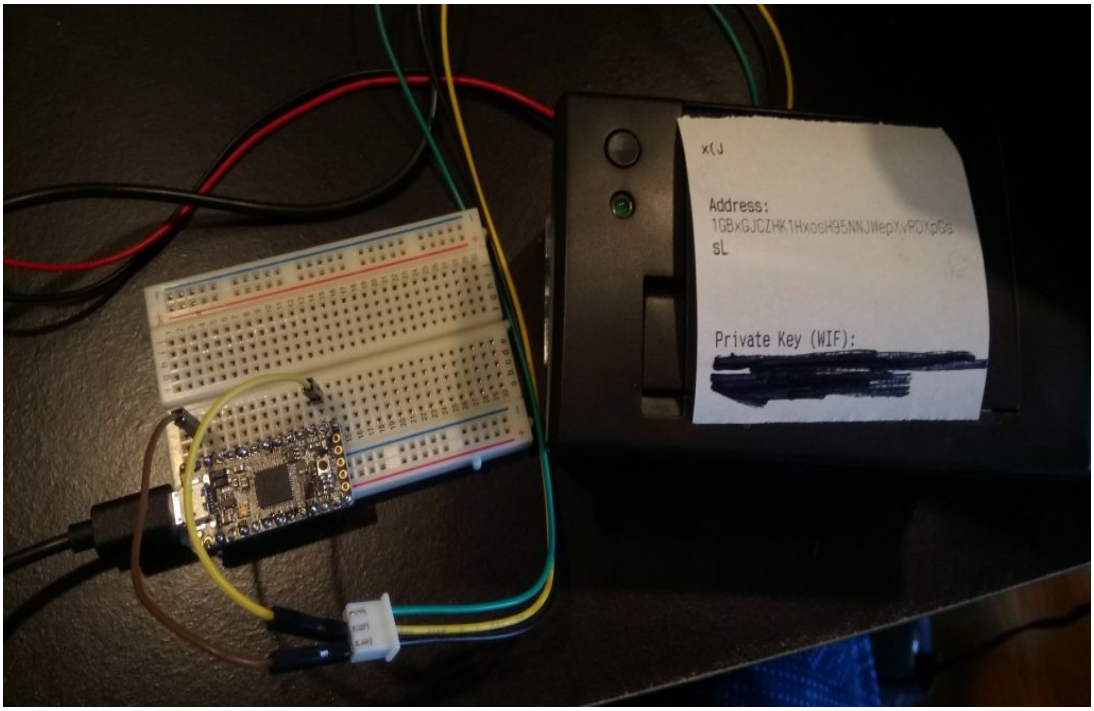
The first option is wiring up a thermal receipt printer. This uses an Adafruit mini printer that costs about $60, and it's so cool to play with. To use this peripheral, I had to wire it to an external power source that came with the kit. The printer was then wired to *ground* and *serial tx* on the processor.

```
    # Print a paper wallet with the thermal receipt printer
    def print_address_privkey(self, address, privkey, print_privkey=True):

        # Intialize the printer
        uart = busio.UART(board.TX, board.RX, baudrate=19200)
        ThermalPrinter = adafruit_thermal_printer.get_printer_class(2.69)
        printer = ThermalPrinter(uart)

        printer.bold = True

        # Warm up and wait so we get the best print quality
        printer.warm_up()
        time.sleep(2)

        # Print the address information
        printer.feed(3)
        printer.print("Address:")
```

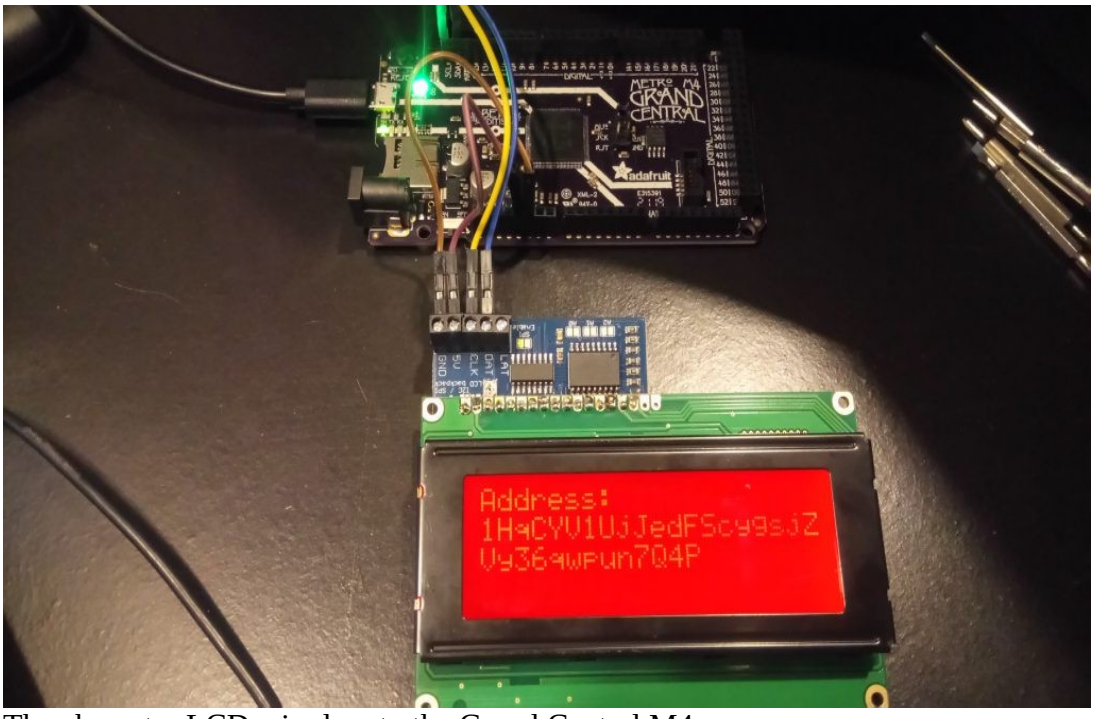
```
        printer.print(address)

        if print_privkey:
            printer.feed(3)
            printer.print("Private Key (WIF):")
            printer.print(privkey)

        printer.feed(3)
```

The next option (slightly more complex, actually) is the character LCD display. For this, I got to solder for the first time since high school! Thankfully my friend and engineer Nathan Schomer taught me well back in the day. I soldered the I2C backpack to the character LCD, and wired it to *scl* and *sda* on the board for data. Power requires either *5v or USB* depending on the board, and a *ground.*

```
    # Prepare the data for display on the character screen
    def prep_data(self, data, colmax):

        prepped_data = ""
        for i in range(0, len(data)):
            if i != 0 and i % colmax == 0:
                prepped_data = prepped_data + "\n"

            if data[i] in self.BASE58_ALPHABET:
                prepped_data = prepped_data + data[i]

        return prepped_data

    # Display the address or private key on a character LCD
    def display_address_privkey(self, address, privkey):

        # Initialize the board
        i2c = busio.I2C(board.SCL, board.SDA)
        cols = 20
        rows = 4
        lcd = character_lcd.Character_LCD_I2C(i2c, cols, rows)
        lcd.backlight = True

        # Prep the address and display, wait N seconds,
        # then display the private key
        while True:
            lcd.clear()
            address = self.prep_data(address, cols)
            lcd.message = "Address:\n" + address

            time.sleep(self.DISPLAY_INTERVAL)

            lcd.clear()
            privkey = self.prep_data(privkey, cols)
            lcd.message = "Private Key (WIF):\n" + privkey

            time.sleep(self.DISPLAY_INTERVAL)
```



The thermal printer wired up to the ItsyBitsy M4



The character LCD wired up to the Grand Central M4.

Easy Offline Address Generation with uBitAddr

I was extremely excited to get this project working from start to finish, as it was certainly a challenge. Now that it's done, it's easy to generate addresses completely offline and keep them

secure. All I have to do is boot up a board with an output, and make sure the key and address are safely written down and backed up.

This project has plenty of interesting potential features as well - as I improve on it I can add things like different entropy sources, persistent storage of keys/addresses on the board or a micro SD, different outputs, newer address formats, and more.

I encourage you all to go out and try building things like these as well! Try running your own uBitAddr setup if you have some knowledge of microcontrollers, or get started with your own address generation software built on a platform you're familiar with. If you've never used an Arduino or CircuitPython controller before, try making something fun - they are more accessible than you think. Happy tinkering!