Hex Encoding, Version Prefixes, and Keccak (uBitAddr Code Companion Update)

Author	josh
Date	2019-10-02 19:01:40

Overview

In a previous tutorial, I shared the first published version of my uBitAddr project. This software and hardware project is a totally open source, open hardware, DIY offline address generator. It allows the user to generate a private key and the associated address for long-term storage, in something like a paper or metal wallet.

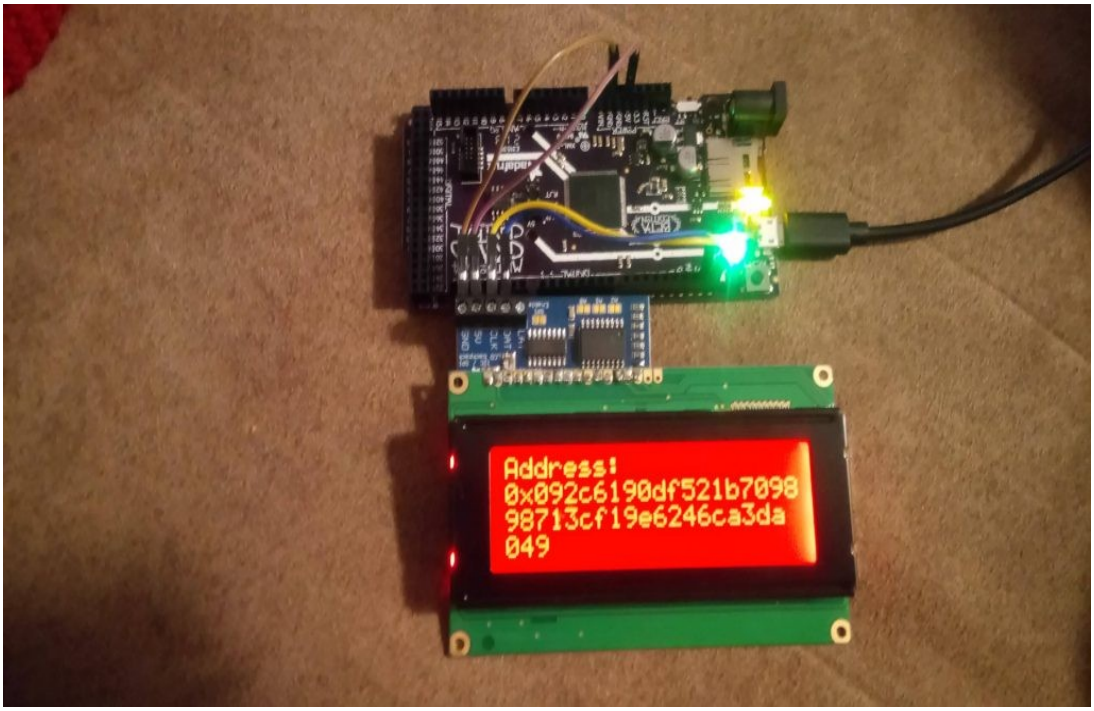
Since developing the original version, I've added several new features to the project and faced some interesting challenges along the way. Let's take a look at some of the new features and complications I had to tackle.

Expanding Currency Support to my "Big 4"

Why add new currencies?

Personally, I am not a maximalist when it comes to Bitcoin or any other cryptocurrency. There are several I find interesting to study and use, and for my own personal preference those currencies are Bitcoin Cash, Ethereum, Litecoin, and Bitcoin.

The original version supported standard Bitcoin addresses (base58check encoded, no segwit support) and by extension supported Bitcoin Cash. However, BCH has long moved to a different address format, and Litecoin and Ethereum require an address derivation scheme that's a bit different. I decided to take this project to it's full potential (for my interests, at least) and add module and API support for all those currencies and formats.



Generating an Ethereum keypair totally offline!

Adding BCH cashaddr support

Thankfully, adding Bitcoin Cash support was fairly straightforward. The Trezor crypto libraries I use for cryptographic primitives/encoding already support cashaddr. I created a separate "address_from_pubkey" function in the module `__init__.c` code that derives the address from the pubkey and uses Trezor's cashaddr encoding:

```
// Add the version specifier
unsigned char raw_address_nocheck[RAW_ADDRESS_NOCHECK_LENGTH];
raw_address_nocheck[0] = CASHADDR_P2PKH_BITS | CASHADDR_RIPEMD160_BITS;
memcpy(raw_address_nocheck + 1, round_2, RIPEMD160_DIGEST_LENGTH);

// Cashaddr  encode
cash_addr_encode((char*) address, "bitcoincash", raw_address_nocheck,
RAW_ADDRESS_NOCHECK_LENGTH);
```

There's a slightly different version specifier that has to be used with cashaddr, and that's really it. Trezor's `cash_addr_encode` takes care of the special cashaddr checksum, so there was no need to compute that manually like I do in the base58check code.

For the API, I simply added an optional flag in the constructor to allow cashaddr encoding instead of the legacy base58check:

```
uba = uBitAddr(output=uBitAddr.OUTPUT_DISPLAY, bch=True)
```

Litecoin Support

Litecoin support was again, fairly straightforward. The address derivation process is largely the same, but LTC uses different *version specifiers* for the WIF encoded key and the base58check address. Instead of using `unsigned char BTC_ADDR_PREFIX = 0x0;` for the address, LTC uses `unsigned char LTC_ADDR_PREFIX = 0x30;`. Likewise, the private key prefix is slightly different: `unsigned char LTC_WIF_PREFIX = 0xB0;` vs. Bitcoin's `unsigned char BTC_WIF_PREFIX = 0x80;`. Otherwise, the derivation process is the same.

On the API side, at this point I added the ability to specify the desired currency as a optional argument when constructing the `uBitAddr` object:

```
uba = uBitAddr(output=uBitAddr.OUTPUT_DISPLAY, currency=uBitAddr.LTC)
```

Ethereum Support

Adding support for Ethereum was the most challenging. First, Ethereum requires a different hashing algorithm for deriving the address from the public key. It uses the *Keccak* version of SHA3, which outputs a 256 bit hash. However, Keccak is different than the final version of SHA3 accepted as the NIST standard. Fortunately, once again, Trezor has us covered. Their hardware wallet supports Ethereum and therefore has Keccak primitives in the crypto code. Another difference is that the address derivation scheme removes the `04` byte from the front of the pubkey (the byte that indicates the key is uncompressed) before the single round of Keccak hashing:

```
      // First, hash the public key without the 04 uncompressed pubkey
indicator byte at the front
      unsigned char round_1[SHA3_256_DIGEST_LENGTH];

      keccak_256(pubkey + 1, ETH_PUBKEY_LENGTH, round_1);

      unsigned char raw_address[RAW_ETH_ADDRESS_LENGTH];
      memcpy(raw_address, round_1 + 12, RIPEMD160_DIGEST_LENGTH);
```

Next, ETH addresses use hex encoding rather than base58 or cashaddr. This would seem to be the simplest form of encoding, but once again the world of microcontroller programming threw me for a loop! Typically, one can format output data as hex in C using `sprintf(%02x)`. However, I was not able to compile the CircuitPython module with that function call included.

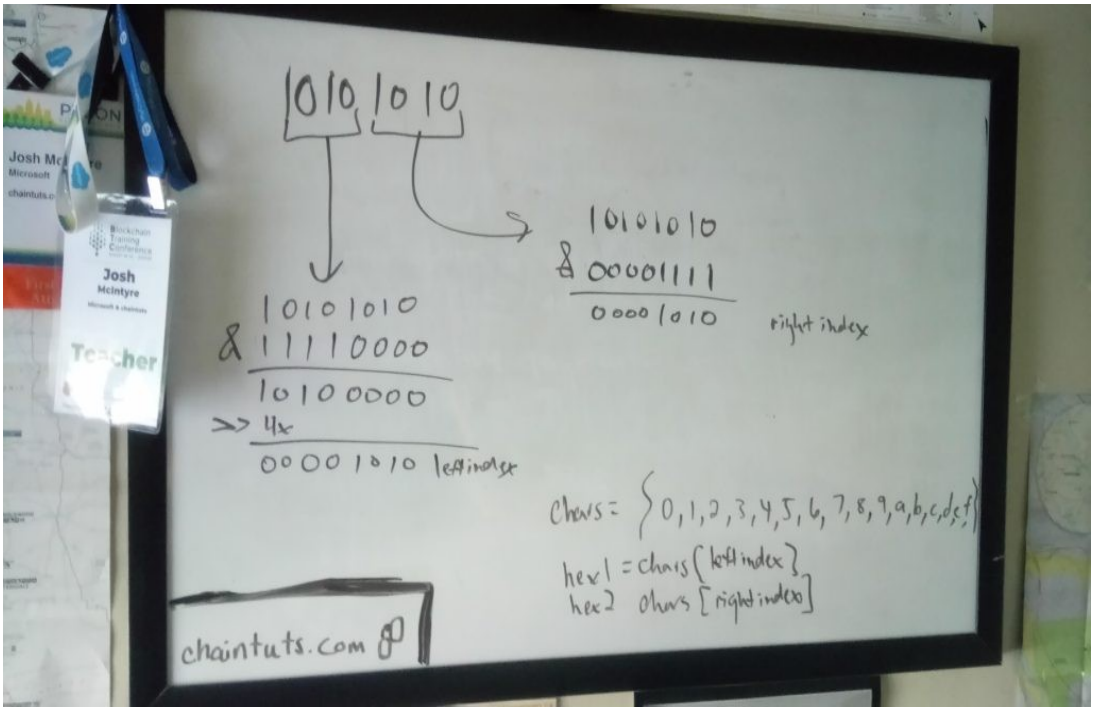
So, I searched StackOverflow for solutions. Most of the sample code didn't make immediate sense, but a user mentioned "using bit masking and shifting" and I was able to work out the solution on the whiteboard. I added a function that does hex encoding by masking off 4 bits at a time (a "nibble"), and for the leftmost nibble, using a 4 bit right shift. This gives a number that can be used as an index on the set of hex characters, hence giving us the two characters we need for encoding a single byte as hex.

```
// Convert a byte to hex format and write directly to the buffer
// This is a substitute for sprintf on a microcontroller platform
void byte_to_hex(unsigned char byte, unsigned char* buffer)
{
```

```
      char hex_chars[16] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'a', 'b', 'c', 'd', 'e', 'f' };
      unsigned char left_mask = 0xF0;
      unsigned char right_mask = 0xF;

      // First, calculate the character for the first nibble (4 bits)
      // Mask off the last 4 bits, then right shift 4 bytes
      // This will be the index used to get the right character from hex_chars
      unsigned char left_index = byte & left_mask;
      left_index = left_index >> 4;
      *buffer = hex_chars[left_index];

      unsigned char right_index = byte & right_mask;
      *(buffer + 1) = hex_chars[right_index];
}
```



Whiteboarding up a homebrew hex encoding function

Homebrew Offline Address Generation, Achieved!

The end result of this work is, in my opinion, a pretty sweet little project. A Reddit user called this project "cypherpunk" and I have to agree - there's nothing more nerdy and paranoid than writing *your own code* for address generation. This is a very fun, challenging, and constantly evolving project.

Some final thoughts are, of course on security, This code is experimental, so play with it and use it at your own risk. I am a security-conscious engineer but I am NOT a cryptography expert. As well, there are pitfalls to dealing with raw keypairs compared to mnemonic seeds. As long as they're generated with sufficient entropy they are secure, but there are pitfalls when it comes to

writing down and storing the keys as well as sweeping the funds properly for future spending. Always do your research and ask questions before dealing with serious money!