Fourmilab Blockchain Tools

by John Walker

Version 1.0
September 2021

Build 754 — 2021-09-14 19:36 UTC

Contents

Chapter 1

Introduction

This collection of programs and utilities provides a set of tools for advanced users, explorers, and researchers of the Bitcoin and Ethereum blockchains. Some of the tools are self-contained, while others require access to a system (either local or remote) which runs a “full node” using the Bitcoin Core software and maintains a complete copy of the up-to-date Bitcoin blockchain. In order to use the Address Watcher, the node must maintain a transaction index, which is enabled by setting “`txindex=1`” in its `bitcoin.conf` file.

Some utilities (for example, the Bitcoin and Ethereum address generator and paper wallet tools) do not require access to a Bitcoin node and others may be able to be used on nodes which have “pruned” the blockchain to include only more recent blocks.

⟨ *Project Title* 1a ⟩ ≡

```
    Blockchain Tools◇
```

Fragment never referenced.

⟨ *Project Version* 1b ⟩ ≡

```
    1.0◇
```

Fragment referenced in 145c.

⟨ *Project File Name* 1c ⟩ ≡

```
    blockchain_tools◇
```

Fragment referenced in 135a, 144b, 145c.

1.1 Build information

The following, updated automatically by the `Makefile` when the program is built, defines the build number (incremented for each “`make build`”) and the UTC date and time of the build.

⟨ *Build number* 1d ⟩ ≡

```
    754◇
```

Fragment referenced in 145ab.

⟨ *Build date and time* 2a ⟩ ≡

```
    2021-09-14 19:36◇
```

Fragment referenced in 145ab.

1.2 Configuration

Include the configuration from `configuration.w`.

Default level of verbosity.

⟨ *Verbosity level* 2b ⟩ ≡

```
    0◇
```

Fragment referenced in 102, 117, 128a.

Interval to poll the blockchain in seconds.

⟨ *Blockchain poll interval* 2c ⟩ ≡

```
    300◇
```

Fragment referenced in 102, 117, 128a.

The following definitions configure the defaults for RPC access to a host running `bitcoind`. All of these may be overridden by command line options.

⟨ *RPC query method* 2d ⟩ ≡

```
    rpc◇
```

Fragment referenced in 139b.

⟨ *RPC host* 2e ⟩ ≡

```
    localhost◇
```

Fragment referenced in 139b.

⟨ *RPC port* 2f ⟩ ≡

```
    8332◇
```

Fragment referenced in 139b, 140a.

⟨ *Bitcoin CLI path* 2g ⟩ ≡

```
    bitcoin-cli◇
```

Fragment referenced in 139b.

⟨ *RPC user* 2h ⟩ ≡

```
    myuser◇
```

Fragment referenced in 139b.

⟨ *RPC password* 2i ⟩ ≡

```
    ◇
```

Fragment referenced in 139b.

These definitions specify defaults for `address_watch`.

⟨ *AW block start* 3a ⟩ ≡

```
-1◇
```

Fragment referenced in 102.

⟨ *AW block end* 3b ⟩ ≡

```
-1◇
```

Fragment referenced in 102.

⟨ *AW block file* 3c ⟩ ≡

```
◇
```

Fragment referenced in 102.

⟨ *AW watch file* 3d ⟩ ≡

```
◇
```

Fragment referenced in 102.

⟨ *AW log file* 3e ⟩ ≡

```
◇
```

Fragment referenced in 102, 117.

⟨ *AW monitor wallet* 3f ⟩ ≡

```
FALSE◇
```

Fragment referenced in 102.

⟨ *AW wallet purge interval* 3g ⟩ ≡

```
3600◇
```

Fragment referenced in 114a.

Defaults for `confirmation_watch`.

⟨ *CW watch confirmations* 3h ⟩ ≡

```
FALSE◇
```

Fragment referenced in 117.

⟨ *CW deem confirmed* 3i ⟩ ≡

```
6◇
```

Fragment referenced in 117, 128a.

If you want the `bitcoin_address` generator to be able to query the HotBits radioactive random number generator, enter your API key and the query URL below.

⟨ *HotBits query URL* 4a ⟩ ≡

```
https://www.fourmilab.ch/cgi-bin/Hotbits.api?nbytes=[NBYTES]&fmt=hex&apikey=[APIKEY]◇
```

Fragment referenced in 25.

⟨ *HotBits API key* 4b ⟩ ≡

```
Pseudorandom◇
```

Fragment referenced in 25.

1.3 Host System Properties

These path names to the Perl and Python interpreters are embedded in programs in the respective languages so they may be invoked directly from the command line. If these are incorrect, you can still run the programs by explicitly calling the correct interpreter. Due to incompatibilities, many systems have both Python versions 2 and 3 installed. If this is the case, be sure you specify the path to Python version 3 or greater below.

⟨ *Perl directory* 4c ⟩ ≡

```
/usr/bin/perl◇
```

Fragment referenced in 145a.

⟨ *Python directory* 4d ⟩ ≡

```
/usr/bin/python3◇
```

Fragment referenced in 145b.

Chapter 2

User Guide

2.1 Overview

Fourmilab Blockchain Tools provide a variety of utilities for users, experimenters, and researchers working with blockchain-based cryptocurrencies such as Bitcoin and Ethereum. These are divided into two main categories.

2.1.1 Bitcoin and Ethereum Address Tools

These programs assist in generating, analysing, archiving, protecting, and monitoring addresses on the Bitcoin and Ethereum blockchains. They do not require you run a local node or maintain a copy of the blockchain, and all security-related functions may be performed on an "air-gapped" machine with no connection to the Internet or any other computer.

- Blockchain Address Generator creates address and private key pairs for both the Bitcoin and Ethereum blockchains, supporting a variety of random generators, address types, and output formats.
- Multiple Key Manager allows you to split the secret keys associated with addresses into n multiple parts, from which any $k \leq n$ can be used to reconstruct the original key, allowing a variety of secure custodial strategies.
- Paper Wallet Utilities includes a Paper Wallet Generator which transforms a list of addresses and private keys generated by the Blockchain Address Generator or parts of keys produced by the Multiple Key Manager into a HTML file which may be printed for off-line "cold storage", and a Cold Storage Wallet Validator that provides independent verification of the correctness of off-line copies of addresses and keys.
- Cold Storage Monitor connects to free blockchain query services to allow periodic monitoring of a list of cold storage addresses to detect unauthorised transactions which may indicate they have been compromised.

2.1.2 Bitcoin Blockchain Analysis Tools

This collection of tools allows various kinds of monitoring and analysis of the Bitcoin blockchain. They do not support Ethereum. These programs are intended for advanced, technically-oriented users who run their own full Bitcoin Core node on a local computer. Note that anybody can run a Bitcoin node as long as they have a computer with the modest CPU and memory capacity required, plus the very large (and inexorably growing) file storage capacity to archive the entire Bitcoin blockchain. You can run a Bitcoin node without being a "miner", nor need you expose your computer to external accesses from other nodes unless you so wish.

These tools are all read-only monitoring and analysis utilities. They do not generate transactions of any kind, nor do they require unlocked access to the node owner's wallet.

- Address Watch monitors the Bitcoin blockchain and reports any transactions which reference addresses on a "watch list", either deposits to the address or spending of funds from it. The program may also be used to watch activity on the blockchain, reporting statistics on blocks as they are mined and published.
- Confirmation Watch examines blocks as they are mined and reports confirmations for a transaction as they arrive.
- Transaction Fee Watch analyses the transaction fees paid to include transactions in blocks and the reward to miners and produces real-time statistics and log files which may be used to analyse transaction fees over time.

2.2 Blockchain Address Generator

The Blockchain Address Generator, with program name `blockchain_address`, is a stand-alone tool for generating addresses and private keys for both the Bitcoin and Ethereum blockchains. This program does not require access to a Bitcoin node and may be run on an "air gapped" machine without access to the Internet. This permits generating keys and addresses for offline cold storage of funds (for example, in paper wallets kept in secure locations) without the risk of having private keys compromised by spyware installed on the generating machine.

The Address Generator may be run from the command line (including being launched by another program) or interactively, where the user enters commands from the keyboard. The commands used in both modes of operation are identical.

2.2.1 Architecture

The address generator is not a single-purpose utility, but rather more of a toolkit which can be used in a variety of ways to meet your requirements. The program is implemented as a "stack machine", somewhat like the FORTH or PostScript languages. Its stack stores "seeds", which are 256-bit integers represented as 64 hexadecimal digits, "`0`" to "`F`" (when specifying seeds in hexadecimal, upper or lower case letters may be used interchangeably). Specifications on the command line are not options in the usual sense, but rather commands that perform operations on the stack. When in interactive mode, the same commands may be entered from the keyboard, without the leading "`-`", and perform identically.

Here are some sample commands which illustrate operations you can perform.

```
blockchain_address -urandom -btc
```

Obtain a seed from the system's fast (non-blocking) entropy source and generate a Bitcoin key/address pair from it, printing the results on the console.

```
blockchain_address -repeat 10 -pseudo -format CSV -eth
```

Generate 10 seeds using the program's built-in Mersenne Twister pseudorandom generator (seeded with entropy from the system's fast entropy source), then create Ethereum key/address pairs for each and write as a Comma-Separated Value (CSV) file intended, for example, as offline "paper wallet" cold storage.

```
blockchain_address -repeat 16 -hbapik MyApiKey -hotbits -shuffle -repeat 1 -xor -test -btc
```

Request 16 seeds from Fourmilab's HotBits radioactive random number generator (requires Internet connection), shuffle the bytes among the 16 seeds, exclusive-or the two top seeds together, perform a randomness test on the result using Fourmilab's random sequence tester, then use the seed to generate a Bitcoin key/address pair.

2.2.2 Commands

`-aes`
: Encrypt the second item on the stack with the Advanced Encryption Standard, 256 bit key size version, with the key on the top of the stack. The stack data are encrypted in two 128 bit AES blocks in cipher-block chaining mode and the encrypted result is placed on the top of the stack.

`-bindump` ***filename***
: Write the entire stack in binary to the named `filename`. A dump to file may be reloaded onto the stack with the `-binfile` command.

`-binfile` ***filename***
: Read successive 64 byte blocks from the binary file *filename* and place them on the stack, pushing down the stack with each block.

`-btc`
: Use the seed on the top of the stack, which is removed after the command completes, to generate a Bitcoin private key and public address, which are displayed on the console in all of the various formats available. If the `-format` command has been to select CSV output, CSV records are generated using the specified format options. If a `-repeat` value has been set, that number of stack items will be used to generate multiple key/address pairs.

`-clear`
: Remove all items from the stack.

`-drop`
: Remove the top item from the stack.

`-dump`
: Dump the entire stack in hexadecimal to the console or to a file if `-outfile` has been set. A dump to file may be reloaded onto the stack with the `-hexfile` command.

`-dup`
: Duplicate the top item on the stack and push on the stack.

`-eth`
: Generate an Ethereum private key and public address from the seed at the top of the stack, which is removed. The key and address are displayed on the console in human-readable form. If the `-format` command has been to select CSV output, CSV records are generated using the specified format options. If a `-repeat` value has been set, that number of stack items will be used to generate multiple key/address pairs.

`-format` ***fmt***
: Set the format to be used for key/address pairs generated by the `-btc` and `-eth` commands. If the first three letters of *fmt* are "`CSV`" (case-sensitive), a Comma-Separated Value file is generated. Letters following "`CSV`" select options, which vary depending upon the type of address being generated. For Bitcoin addresses, the following options are available.

`q`	Use uncompressed private key
`u`	Use uncompressed public address
`l`	Legacy ("`1`") public address
`c`	Compatible ("`3`") public address
`s`	Segwit "`bc1`" public address

For Ethereum addresses, options are:

`n`	No checksum on public address
`p`	Include full public key

For either kind of address, the letter "`k`" indicates that a subsequent key generation command will not remove the keys it processes from the stack. This permits generating the same keys in different formats. The letter "`b`" on either address type causes the private key to be omitted from CSV format output, replaced by a null string. This allows generation of address lists containing only public addresses that may be used with utilities such as `cold_comfort` and `address_watch` without risking compromise of the private keys.

`-hbapik` ***APIkey***

When requesting true random data from Fourmilab's HotBits radioactive random number generator, use the *APIkey* to permit access to the generator. If you don't have an API key (they are free), you may request pseudorandom data based upon a radioactively-generated seed by specifying an API key of "`Pseudorandom`".

`-help`

Print a summary of these commands.

`-hexfile` ***filename***

Load one or more seeds from the named *filename*, which contains data in hexadecimal format. White space in the file (including line breaks) is ignored, and each successive sequence of 64 hexadecimal digits is pushed onto the stack as a 256 bit seed. The `-hexfile` command can load keys dumped to a file with the `-outfile` and `-dump` commands back onto the stack.

`-hotbits`

Retrieve one or more 256 bit seeds from Fourmilab's HotBits radioactive random number generator, using the API key specified by the `-hbapik` command. If the `-repeat` command has specified multiple keys, that number of keys will be retrieved from HotBits and pushed on the stack.

`-inter`

Enter interactive mode. The user is prompted for commands, which are entered exactly as on the command line, except without the leading hyphen on the command name. To exit interactive mode and return to processing commands from the command line, enter "`end`", "`exit`", "`quit`", or the end of file character.

`-minigen`

Generate a Bitcoin mini private key, display the generated key, and push the full seed for the key on the stack. Mini private keys were introduced to allow encoding a Bitcoin private key on physical coins, bills, or other objects which lack the space for a full private key, which can be up to 52 characters long. A mini key is just 30 characters, but can represent only a subset of possible Bitcoin addresses and is consequently less secure—they should be used only when absolutely necessary. Due to the nature of mini keys, the generation process is different that used by the `-btc` command. The `-minigen` command internally generates the seed for the key by mixing the system's fast entropy generator and this program's internal pseudorandom generator seeded by the system fast entropy generator. After finding a suitable key, it pushes the seed on the stack and displays the corresponding key. You may then use the `-btc` command to generate the corresponding public Bitcoin address in whichever format(s) you wish. If the `-format` is set to "`CSV`", an address file is generated which is compatible with the `btc` command, but with the addition of a fifth field in every record containing the mini key. You may use the `-repeat` command to generate multiple keys and the "`k`" option on the `-format` to keep the seeds on the stack.

`-minikey` *mini_private_key*

Validate and decode the specified mini private key (see above) and, if it is properly formatted, place the seed it encodes on the stack. You may then use the `-btc` command to generate other forms of private keys or public addresses from the seed. Both legacy 22 character and the present standard 30 character mini keys may be specified.

`-mnemonic`

Generate a Bitcoin Improvement Proposal 39 (BIP39) mnemonic phrase from the seed on the top of the stack. The seed remains on the stack.

`-not`
: Invert the bits of the seed on the top of the stack.

`-outfile` ***filename***

Output from subsequent `-btc`, `-eth`, and `dump` commands will be written to *filename* instead of standard output. Specifying a *filename* of "`-`" restores output to standard output. Each key generation command overwrites any previous output in *filename*; it is not concatenated. Note that a file written by `-dump` may be loaded back on the stack with the `-hexfile` command.

`-over`

Duplicate the second item on the stack and push it on the top of the stack.

`-p`

Print the top item on the stack on the console.

`-phrase` ***words...***

Push a key defined by a Bitcoin Improvement Proposal 39 (BIP39) mnemonic phrase on the stack. On the command line, the phrase should be enclosed in quotes.

`-pseudo`

Push one or more seeds generated by the internal Mersenne Twister pseudorandom generator onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. The pseudorandom generator is itself seeded by entropy supplied by the system's fast entropy source (`/dev/urandom` on most Unix-like systems).

`-pseudoseed`

Use the number of stack items set by `-repeat` to seed the pseudorandom generator. You may specify up to 78 stack items, representing 624 32-bit seed values. Any more are not used and left on the stack. Any previous generator and seed is deleted. This is normally used only for regression testing where repeatable pseudorandom data are required.

`-random`

Push one or more seeds read from the system's strong entropy source (`/dev/random` on most Unix-like systems) onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. Reading data from a strong source faster than the system can collect hardware entropy may result in delays: the program will wait as long as necessary to obtain the requested number of bytes.

`-repeat` ***n***

Commands which generate and consume seeds will create and use *n* seeds instead of the default of 1. To restore the default, specify `repeat 1`.

`-roll` ***n***

Rotate the top *n* stack items, moving item *n* to the top of the stack and pushing other items down.

`-rot`

Rotate the top three stack items. Item three becomes the top of the stack and the other items are pushed down.

`-rrot`

Reverse rotate the top three stack items. The seed on the top of the stack becomes the third item and the two items below it move up, with the second becoming the top.

`-seed` ***hex_data***

The 256 bit seed, specified as 64 hexadecimal digits, is pushed onto the stack. The seed may be preceded by "`0x`", but this is not required.

`-sha2`

The seed on the top of the stack is replaced by the hash (digest) generated by the Secure Hash Algorithm 2 (SHA-2), 256 bit version (SHA2-256). If `-repeat` has been set greater than one, the specified number of seeds will be removed from the stack and concatenated, top down, the digest computed, and placed back on the stack.

`-sha3`

The seed on the top of the stack is replaced by the hash (digest) generated by the Secure Hash Algorithm 3 (SHA-3), 256 bit version (SHA3-256). If `-repeat` has been set greater than one, the specified number of seeds will be removed from the stack and concatenated, top down, the digest computed, and placed back on the stack.

`-shuffle`

Shuffle bytes of items on the stack using pseudorandom values generated as for the `-pseudo` command. Shuffling bytes can mitigate the risk of interception of seeds generated remotely and transmitted across the Internet. (Secure `https:` connections are used for all such requests, but you never know....) The number of items shuffled is set by `-repeat`.

`-swap`

Exchange the top two items on the stack.

`-test`

Use the Fourmilab `ent` random sequence tester to evaluate the apparent randomness of the top items on the stack. The number of items tested may be set with `-repeat`. You must have `ent` installed on your system to use this command. Randomness is evaluated at the bit stream level.

`-testall`

Use the Fourmilab `ent` random sequence tester to evaluate the apparent randomness of the entire contents of the stack. You must have `ent` installed on your system to use this command. Randomness is evaluated at the bit stream level.

`-testmode` ***n***

Set developer test modes to the bit-coded value *n*, which is the sum of the mode bits to enable. These are intended for development and regression testing and should not be enabled for production use, leaving the setting at the default of 0. The 1 bit makes the `-minigen` produce deterministic output from a fixed `-pseudoseed`. The 2 bit causes `blockchain_address` to list all of the Perl library modules it has used during its execution.

`-type` ***Any text***

Display the text on the console. This is often used in command files to inform the user what's going on.

`-urandom`

Push one or more seeds read from the system's fast entropy source (`/dev/urandom` on most Unix-like systems) onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. The fast generator has no limitation on generation rate, so you may request any amount of data without possibility of delay.

`-wif` ***private_key***

Push the seed represented by the Bitcoin Wallet Import Format (WIF) key onto the stack.

`-xor`

Perform a bitwise exclusive or of the top two items on the stack and push the result on the stack.

`-zero`

Push an all zero seed on the stack.

2.3 Multiple Key Manager

The Multiple Key Manager (`multi_key`) splits the private keys used to access funds stored in Bitcoin or Ethereum addresses into multiple independent parts, allowing them to be distributed among a number of custodians or storage locations. The original keys may subsequently be reconstructed from a minimum specified number of parts. Each secret key is split into n parts ($n \geq 2$), of which any $k, 2 \leq k \leq n$ are sufficient to reconstruct the entire original key, but from which the key cannot be computed from fewer than k parts. In the discussion below, we refer to n as the number of `parts` and k as the number `needed`. The splitting and reconstruction of keys is performed using the Shamir Secret Sharing technique.

The ability to split secret keys into parts allows implementing a wide variety of custodial arrangements. For example, a company treasury's cold storage vault might have secret keys split five ways, with copies entrusted to the chief executive officer, chief financial officer, an inside director, an outside director, and one kept in a safe at the office of the company's legal firm. If the parts were generated so that any three would re-generate the secret keys, then at at least three people would have to approve access to the funds stored in the vault, which reduces the likelihood of their misappropriation. The existence of more parts than required guards against loss or theft of one of the parts: should that happen, three of the remaining copies can be used to withdraw the funds and transfer them to new accounts protected by new multi-part keys.

To create multiple keys, start with a comma-separated value (CSV) file in the format created by `blockchain_address` with "`format CSV`" selected. Let's call this file `keyfile.csv`. Now, to split the keys in this file into five parts, any three of which are sufficient to reconstruct the original keys, use the command:

```
multi_key -parts 5 -needed 3 keyfile.csv
```

This will generate five split key files named `keyfile-1.csv`, `keyfile-2.csv`, … `keyfile-5.csv`. These are the files which are distributed to the five custodians. After verifying independently that the parts can be successfully reconstructed (you can't be too careful!), the original `keyfile.csv` is destroyed, leaving no copy of the complete keys. (All of this should, of course, be done on an "air gapped" machine not connected to any network or external device which might compromise the complete keys while they exist.)

When access to the keys is required, any three of the five parts should be provided by their holders and combined with a command like:

```
multi_key -join keyfile-4.csv keyfile-1.csv keyfile-2.csv
```

Again, you can use any three parts and specify them in any order. This will create a file named `keyfile-merged.csv` containing the original keys in the same format as was created by `blockchain_address`. You can then use this file with any of the other utilities in this collection or use one or more of the secret keys to "sweep" the funds into a new address. To maximise security, once a set of keys has been recombined, funds should be removed from all and those not used transferred to new cold storage addresses, broken into parts as you wish. In many cases, it makes sense to split individual keys rather than a collection of many so you need only join the ones you immediately intend to use.

Once the parts have been generated on the air-gapped machine, they are usually written to offline paper storage (using the `paper_wallet` program, for example, which works with split key files as well as complete key files) or archival media such as write-once optical discs, perhaps with several identical redundant copies per part. Their custodians should store the copies of their parts in multiple secure, private locations to protect against mishaps that might destroy all copies of their part.

The ability to create multiple parts allows flexibility in their distribution. You might, for example, entrust two parts to the company CEO, who would only need one part from another officer or director to access the vault, while requiring three people other than the CEO to access it.

Although primarily intended to split blockchain secret keys into parts, `multi_key` may be used to protect and control access to any kind of secret which can be expressed as 1024 or fewer text characters: for example, passwords on root signing certificates, decryption keys for private client information, or the formula for fizzy soft drinks.

2.3.1 Command line options

`-help`
: Print how to call information.

`-join`
: Reconstruct the original private keys from the parts included in the files specified on the command line. You must supply at least the `-needed` number of parts when they were created (if you specify more, the extras are ignored). The output is written to a file with the specified `-name` or, if none is given, that of the first part with its number replaced with "`-merged`". The file will be will be in the comma-separated value (CSV) format in which `blockchain_address` writes addresses and keys it generates and is used by other programs in this collection.

`-name` ***name***
: When splitting keys, the individual part files will be named "*name*-*n*`.csv`", where *n* is the part number. If no `-name` is specified, the name of the first key file supplied will be used.

`-needed` ***k***
: When reconstructing the original keys, at least *k* parts (default 3) must be specified. This option is ignored when joining the parts.

`-parts` ***n***
: Keys will be split into *n* parts (default 3). This option is ignored when joining parts.

`-prime` ***p***
: Use the prime number *p* when splitting parts. This should only be specified if you're a super expert who has read the code, understands the algorithm, and knows what you're doing, otherwise you're likely to mess things up. The default is 257.

2.4 Paper Wallet Utilities

The safest way to store cryptocurrency assets not needed for transactions in the near term is in "cold storage": kept offline either on a secure (and redundant) digital medium or, safest of all, paper (again, replicated and stored in multiple secure locations). A cold storage wallet consists simply of a list of one or more pairs of blockchain public addresses and private keys. Funds are sent to the public address and the corresponding private key is never used until the funds are needed and they are "swept" into an online wallet by entering the public key.

The `blockchain_address` program makes it easy to generate address and key pairs for offline cold storage, encoding them as comma-separated value (CSV) files which can easily be read by programs. For storage on paper, a more legible human-oriented format is preferable, which the utilities in this chapter aid in creating and verifying.

2.4.1 Paper Wallet Generator

The `paper_wallet` program reads a list of Bitcoin or Ethereum public address and private key pairs, generated by `blockchain_address` in comma-separated value (CSV) format, and creates an HTML file which can be loaded into a browser and then printed locally to create paper cold storage wallets. In the interest of security, this process, as with generation of the CSV file, should be done on a machine with no connection to the Internet ("air gappped"), and copies of the files deleted from its storage before the machine is connected to a public network.

2.4.1.1 Creating a paper wallet

Assume you've created a cold storage wallet with twenty Ethereum addresses using the `blockchain_address` program, for example with the command:

```
blockchain_address -repeat 20 -urandom -outfile coldstore.csv -format CSV -eth
```

This should be done on the same air-gapped machine on which you'll now create the paper wallet. Be careful to generate the `coldstore.csv` file in a location you'll erase before connecting the machine to a public network. If you wish to keep a machine-readable cold storage wallet, copy the `coldstore.csv` file to multiple removable media (for example, flash storage devices [perhaps encrypted], writeable compact discs, etc.) Be aware that no digital storage medium has unlimited data retention life, and even if the data are physically present, it may be difficult to near-impossible to find a drive which can read it in the not-so-distant future. By contrast, we have millennia of experience with ink on paper, and if protected from physical damage, a printed cold storage wallet will remain legible for centuries.

Now let's create a paper wallet. Using the `coldstore.csv` file we've just generated and the default parameters, this can be done with:

```
paper_wallet coldstore.csv >coldstore.html
```

You can now load the `coldstore.html` file into a Web browser with a `file:coldstore.html` URL, use print preview to verify it is properly formatted, then print as many copies as you require for safe storage to a local printer.

2.4.1.2 Command line options

`-date` ***text***
: The specified *text* will be used as the date in the printed wallet. Any text may be used: enclose it in quotes if it contains spaces or special characters interpreted by the shell. If no `-date` is specified, the current date is used, in ISO-8601 `YYYY-MM-DD` format.

`-font` ***fname***
: Use HTML/CSS font name *fname* to display addresses and keys. The default is `monospace`.

`-help`
: Print a summary of the command line options.

`-offset` ***n***
: The integer *n* will be added to the address numbers (first CSV field) in the input file. If you've generated a number of cold storage wallets with the same numbers and wish to distinguish them in the printed versions, this allows doing so.

`-perpage` ***n***
: Addresses will be printed *n* per page. The default is 10 addresses per page. The number which will fit on a page depends upon your paper size, font selection, and margins used when printing—experiment with print preview to choose suitable settings.

`-prefix` ***text***
: Use *text* as a prefix for address numbers from the CSV file (optionally adjusted by the `-offset` option). This allows further distinguishing addresses in the printed document.

`-separator` ***text***
: Display addresses and private keys as groups of four letters and number separated by the sequence *text*, which may be an HTML text entity such as "`&ndash;`".

`-size` ***sspec***
: Use HTML/CSS font size *sspec* to display addresses and keys. The default is `medium`.

`-title` ***text***
: Use the specified `text` as the title for the cold storage wallet. If no title is specified, "Bitcoin Wallet" or "Ethereum Wallet" will be used, depending upon the type of address in the CSV file.

`-weight` ***wgt***
: Use HTML/CSS font weight *wgt* to display addresses and keys. The default is `normal`.

2.4.2 Cold Storage Wallet Validator

When placing funds in offline cold storage wallets, an abundance of caution is the prudent approach. By their very nature, once funds are sent to the public address of a cold storage wallet, that address is never used again, nor is its private key ever used at all until the time comes, perhaps years or decades later, to "sweep" the funds from cold storage back into an online wallet. Consequently, if, for whatever reason, there should be an error in which the private key in the offline wallet does not correspond to the public address to which the funds were sent, those funds will be irretrievably lost, with no hope whatsoever of recovery. Entering the private key into a machine connected to the Internet in order to verify it would defeat the entire purpose of a cold storage wallet: that its private keys, once generated on an air-gapped machine, are never used prior to returning the funds from cold storage.

While the circumstances in which a bad address/key pair might be generated and stored may seem remote, the consequences of this happening, whether due to software or hardware errors, incorrect operation of the utilities used to generate them, or malice, are so dire that a completely independent way to verify their correctness is valuable.

The `validate_wallet` program performs this validation on cold storage wallets, either in the CSV format generated by `blockchain_address` or the printable HTML produced by `paper_wallet`. Further verification that the printed output from the HTML corresponds to the file which was printed will require manual inspection or scanning and subsequent verification. The `validate_wallet` program is a "clean room" re-implementation of the blockchain address generation process used by `blockchain_address` to create cold storage wallets. It is written in a completely different programming language (Python version 3 as opposed to Perl), and uses the Python cryptographic libraries instead of Perl's. While it is possible that errors in lower-level system libraries shared by both programming languages might corrupt the results, this is much less likely than an error in the primary code or the language-specific libraries they use.

2.5 Cold Storage Monitor

For safety, cryptocurrency balances which are not needed for active transactions are often kept in "cold storage", either off-line in redundant digital media not accessible over a network or printed on paper (for example, produced with the `paper_wallet` program) kept in multiple separate locations. Once sent to these cold storage addresses, there should be no further transactions whatsoever referencing them until they are "swept" back into an active account for use.

But under the principle of *doveryay, no proveryay* (trust, but verify), a prudent custodian should monitor cold storage addresses to confirm they remain intact and have not been plundered by any means. (It's usually an inside job, but you never know.) One option is to run a "hot monitor" that constantly watches transactions on the blockchain such as the `address_watch` utility included here, but that requires you to operate a full Bitcoin node and does not, at present, support monitoring of Ethereum addresses.

The `cold_comfort` utility provides a less intensive form of monitoring which works for both Bitcoin and Ethereum cold storage addresses, does not require access to a local node, but instead uses free query services that return the current balance for addresses. You can run this job periodically (once a week is generally sufficient) with a list of your cold storage addresses, producing a report of any discrepancies between their expected balances and those returned by the query.

Multiple query servers are supported for both Bitcoin and Ethereum addresses, which may be selected by command line options, and automatic recovery from transient errors while querying servers is provided.

2.5.1 Watching cold storage addresses

The list of cold storage addresses to be watched is specified in a CSV file in the same format produced by `blockchain_address` and read by `paper_wallet`, plus an extra field giving the expected balance in the cold storage address. For example, an Ethereum address in which a balance of 10.25 Ether has been deposited might be specified as:

```
1,"0x1F77Ea4C2d49fB89a72A5F690fc80deFbb712021","",10.25
```

The private key field is not used by the `cold_comfort` program and should, in the interest of security, be replaced by a blank field as has been done here. There is no reason to expose the private keys of cold storage addresses on a machine intended only to monitor them. You can use the "`b`" and "`k`" options on a `-format CSV` command to generate a copy of the addresses without the private keys. To query all addresses specified in a file named `coldstore.csv` and report the current and expected balances, noting any discrepancies, use:

```
cold_comfort -verbose coldstore.csv
```

If you don't specify `-verbose`, only addresses whose balance differs from that specified in the CSV file will be reported.

2.5.2 Command line options

The `cold_comfort` program is configured by the following command line options.

`-btcsource` ***sitename***

Specify the site queried to obtain the balance of Bitcoin addresses. The sites supported are:

- `blockchain.info`
- `blockcypher.com`
- `btc.com`

You must specify the site name exactly as given above.

`-dust` ***n***

Some miscreants use the blockchain as a means of "spamming" users, generally to promote some shady, scammy scheme. They do this by sending tiny amounts of currency to a large number of accounts, whose holders they hope will be curious and investigate the transaction that sent them, in which the spam message is embedded, usually as bogus addresses. You might think getting paid to receive spam is kind of cool, but the amounts sent are smaller than the transaction cost it would take to spend or aggregate them with other balances. This is an irritation to cold storage managers, who may find their inactive accounts occasionally receiving these tiny payments, which in blockchain argot are called "dust". This option sets the threshold *n* (default 0.001) below which reported balances in excess of that expected will be ignored and not considered discrepancies. If `-verbose` is specified, they will be flagged in the report as "Dust".

`-ethsource` ***sitename***

Specify the site queried to obtain the balance of Ethereum addresses. The sites supported are:

- `blockchain.com`
- `etherscan.io`
- `ethplorer.io`

You must specify the site name exactly as given above.

`-help`

Print a summary of the command line options.

`-loop`

Loop forever querying addresses. After each pass through all the addresses, a pause of `-waitloop` seconds will occur.

`-retry` ***n***

If a query fails, retry it *n* times before abandoning the request and reporting the failure (default 3).

`-shuffle`
: Shuffle the order in which addresses are queried before each pass checking them. This may (or may not) make it less obvious they represent a single cold storage vault.

`-verbose`
: Report all addresses, even if an address's current balance is the same as expected. Transient query failures and retries are also reported.

`-waitconst` ***n***
: Wait *n* seconds (default 17) between queries for address balances. This avoids overloading the sites providing this free service and getting banned for abusing them.

`-waitloop` ***n***
: When using the `-loop` option, pause for *n* seconds (default 3600) after completing queries for all the addresses in the list before commencing the next pass.

`-waitrand` ***n***
: Add a random number between 0 and *n* seconds (default 20) to the constant set by `waitconst` between individual queries. This further reduces the load on the query sites and makes it less obvious they're coming from an automated process.

2.6 Address Watch

The `address_watch` program monitors the Bitcoin blockchain, watching for transactions which involve one or more watched Bitcoin addresses, specified on the command line, in a file listing addresses to watch, or from the addresses in a Bitcoin Core wallet. Address Watch can be used by those who keep Bitcoin reserves in "cold storage", on paper or offline devices for security, alerting them if one of these addresses is used in a transaction, indicating its security has been compromised. The program can also display statistics of blocks added to the blockchain and write a log that can be used for analysis of the blockchain's behaviour. This program requires access to a Bitcoin node with a full copy of the blockchain, configured with transaction indexing ("`txindex=1`").

2.6.1 Command line options

Address Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specify how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-bfile` ***filename***
: Specifies a file used to save the most recent block examined by the program. When the program starts, it begins scanning at the next block. As each block is processed, the block file is updated so a subsequent run of the program will start at the next block.

`-end` ***n***
: Stop scanning and exit after processing block *n*. If no `-end` is specified, `address_watch` will continue scanning for newly-published blocks at the specified `-poll` interval.

`-help`
: Print a summary of the command line options.

`-lfile` ***filename***
: For each transaction involving a watched address, append an entry to a log file containing fields in Comma Separated Value (CSV) format as described in "Watched address log file" below.

`-poll` ***time***
: After reaching the current end of the blockchain, check for newly-published blocks after the specified

time in seconds. If *time* is set to zero, `address_watch` will exit after scanning the last block.

`-sfile` ***filename***
: As each block is processed, append an entry describing it to the statistics file *filename*. Records are written in Comma Separated Value (CSV) format as described in "Block statistics log file" below.

`-start` ***n***
: Start scanning the blockchain at block *n*. If no `-start` is specified, scanning will begin with the next block after that specified in the `-bfile` file or with the next block published.

`-stats`
: For each block processed, print statistics about its content on the console. The statistics are the same as written to a file by the `-sfile` option, but formatted in a primate-readable format.

`-type` ***Any text***
: Print the text on the console.

`-verbose`
: Print detailed information about the contents of blocks. The more times you specify `-verbose`, the more output you'll get.

`-wallet`
: Include addresses in the Bitcoin Core wallet with unspent balances in those watched for transactions. Since every spend transaction in Bitcoin Core completely spends the source address and places unspent funds in a new change address, the option will automatically track these newly-generated addresses as they appear and are used. The list of wallet addresses is updated before scanning each new block that arrives.

`-watch` [***label,***] ***address***
: Add the specified Bitcoin *address* to the watch list. You can specify a label before the address, separated by a comma, for example: `"Money Bin,1ScroogeYebEqDTbdjk36WzLxjCZTkNe3w"`.

`-wfile` ***filename***
: Add addresses read from the specified *filename* in Comma Separated Value (CSV) format to the watch list. Each line in the file specified an address as: *Label*`,`*Bitcoin address*`,`*Private key*`,`*Balance*. The *Label* is an optional human-readable name for the address, and the *Private key* and *Balance* fields are not used by this program.

2.6.2 Log file formats

The `address_watch` program can write two log files, both in Comma Separated Value format, with fields as follows. New items are appended to an existing log file.

2.6.3 Watched address log file

The `-lfile` option enables logging of transactions involving watched addresses. Each log item is as follows.

1. Address label from wallet
2. Bitcoin address
3. Value (negative if spent, positive if received)
4. Date and time (ISO 8601 format)
5. Block number
6. Transaction ID
7. Block hash

2.6.4 Block statistics log file

The `-sfile` option logs statistics for blocks as they are added to the blockchain, with records containing the following fields.

1. Block number
2. Date and time (Unix `time()` format)
3. Number of transactions in block
4. Smallest transaction (bytes)
5. Largest transaction (bytes)
6. Mean transaction size (bytes)
7. Transaction size standard deviation
8. Total size of transactions (bytes)
9. Smallest transaction value (BTC)
10. Largest transaction value (BTC)
11. Mean transaction value (BTC)
12. Transaction value standard deviation
13. Total transaction value (BTC)
14. Total miner reward for block (including transaction fees)
15. Base miner reward for block (less transaction fees)

2.7 Confirmation Watch

When a Bitcoin transaction is posted to the network, it first is placed in the "mempool" by nodes which receive it. Miner nodes choose transactions from the mempool, usually based upon the transaction fee per byte they offer, validate them against their local copy of the entire Bitcoin blockchain and, if and when they find a hash for a candidate block that meets the present difficulty requirement, publish the block to the blockchain and notify other nodes of its publication. Other nodes indepedently validate the transactions it contains and add their confirmations to the transaction, which are recorded on the blockchain. By convention, a transaction is deemed fully confirmed once six or more independent confirmations for it are recorded on the blockchain. Most Bitcoin wallet programs will not spend funds received (even "change" from funds in your own wallet which have been partially spent) until at least six confirmations are received for its transfer to your wallet.

The `confirmation_watch` utility monitors a transaction on the blockchain and reports confirmations as they arrive. It can be used to monitor pending transactions and report when a specified number of confirmations are received. Depending upon the configuration, you can run `confirmation_watch` with the following command lines.

`confirmation_watch` ***transaction_id block_hash***

This form of command may always be used, regardless of configuration. It specifies the hexadecimal transaction ID and hash of the block which contains it. Both of these can be found in the console output and log file generated by `address_watch`.

`confirmation_watch` ***transaction_id***

If your Bitcoin Core node has been configured with with "`txindex=1`", which maintains an index of transactions, you can specify just the *transaction_id*, with the block hash found from the transaction index.

`confirmation_watch` ***address/label***

If you have specified a log file maintained by `address_watch` on the command line with the `-lfile` option, you may specify just the Bitcoin public address to which the transaction pertains or the label you have assigned to in the the Bitcoin Core wallet. The most recent transaction involving that address will be retrieved from the log file and monitored for confirmations.

2.7.1 Command line options

Confirmation Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specify how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-confirmed` ***n***
Specifies the number of confirmations which must be received before a transaction is deemed confirmed. If a transaction is being monitored by the `-watch` option, `confirmation_watch` will exit after this number of confirmations arrive.

`-help`
Print a summary of how to call and command line options.

`-lfile` ***filename***
Use the log file written by the `address_watch` program to locate transactions for a Bitcoin address specified either by its public address or a label given to it in the Bitcoin Core wallet. If this option is not specified, transactions must be identified by their transaction ID.

`-testmode`
Instead of taking the transaction to be watched from the command line or indirectly from the `address_watch` log file, choose a transaction from the most recently mined block and watch its confirmations. This allows developers to test the program on a representative transaction without the need to to submit one or manually find one in a block dump.

`-type` ***Any text***
Print the text on the console.

`-verbose` ***n***
Print detailed information about transactions and confirmations. The more times you specify `-verbose`, the more information you'll see.

`-watch`
Poll for new confirmations every `-poll` seconds until the `-confirmed` number have arrived.

2.8 Transaction Fee Watch

Bitcoin transactions submitted for inclusion in the blockchan are accompanied by a transaction fee paid to the miner who includes the transaction in a block published to the blockchain. Transactions can be selected by miners at their discretion, but in most cases will be chosen to maximise the reward for including them in a block, which usually means those which offer the highest transaction fee per byte (or, more precisely, "virtual byte") of the transaction. Whenever a block is added to the blockchain, Bitcoin Core computes statistics of the fees for transactions within it. In addition, Bitcoin Core computes an "estimated smart fee" as a suggestion to those submitting transactions at the current time.

The `fee_watch` program monitors the blockchain and reports the fee statistics for each block published and fee recommendations from Bitcoin Core, optionally writing both of these to a log file for analysis by other programs. The program is configured by the following command line options.

2.8.1 Command line options

Fee Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specify how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-confirmed` ***n***

Specifies the number of confirmations which must be received before a transaction is deemed confirmed. This is used when requesting an estimate of the current transaction fee with the Bitcoin Core API call `estimatesmartfee` to indicate the priority of the transaction. The default, 6, corresponds to standard priority for this call.

`-ffile` ***filename***

Write a log file of fee information collected by `fee_watch`. The log is written in Comma Separated Value (CSV) format, and contains two kinds of records, distinguished by a digit in the first field. See "Log file format" below for details.

`-help`

Print a summary of how to call and command line options.

`-poll` ***time***

Query and report transction fee estimates and statistics every *time* seconds, by default 300 seconds (five minutes).

`-quiet`

Suppress console output for periodic transaction fee polls. Use this option when writing a log file with the `-ffile` option if you don't want to also see information as it is collected.

`-type` ***Any text***

Print the text on the console.

`-verbose` ***n***

Print detailed information about operations. The more times you specify `-verbose`, the more information you'll see.

2.8.2 Log file format

When the `-ffile` option is specified, `fee_watch` writes a log file recording the transaction fee information it collects. This file is written in Comma Separated Value (CSV) format, and consists of two type of records, as follows.

2.8.2.1 Estimated fee record

These records report the estimated fee, according to the Bitcoin Core `estimatesmartfee` API call, at the indicated time. The estimated transaction fee in the record is expressed in BTC per virtual kilobyte of transaction size, where virtual transaction size is as defined in Bitcoin Improvement Proposal 141 section "Transaction size calculations". One record of this type is generated for every `-poll` interval.

1. Record type, `1`
2. Date and time (Unix `time()` format)
3. Date and time (ISO 8601 format)
4. Estimated transaction fee, BTC per virtual kilobyte

2.8.2.2 Block fee statistics record

If any blocks have been added to the blockchain since the last `-poll` interval, a record will be written, reporting fee statistics for transactions in the block. Note that the time in these records is the time the block was added to the blockchain, not the time of the `fee_watch` poll. The values reported in these records are those returned by the `getblockstats` API call for the block, with fees reported in units of satoshis (BTC 0.00000001) per virtual byte of transaction, where virtual bytes are as defined for the Estimated fee record above.

1. Record type, `2`
2. Block date and time (Unix `time()` format)
3. Block date and time (ISO 8601 format)
4. Block number
5. Minimum fee rate
6. Mean (average) fee rate
7. Maximum fee rate
8. 10th percentile fee rate
9. 25th percentile fee rate
10. 50th percentile fee rate
11. 75th percentile fee rate
12. 90th percentile fee rate

2.9 RPC API configuration

The `address_watch`, `confirmation_watch`, and `fee_watch` programs all require access to the Application Programming Interface (API) provided by a Bitcoin Core node. Access to this interface can be via three mechanisms:

`local` Access to a Bitcoin Core node running on the same machine via the `bitcoin-cli` command line program.

`rpc` Access to a Bitcoin Core node via its Remote Procedure Call (RPC) interface. The node may either be on the same machine or on a different machine configured to accept requests from the host submitting them.

`ssh` Access a remote Bitcoin Core node by submitting commands to its `bitcoin-cli` utility via the Secure Shell (SSH) facility. The client and node machine must be configured to permit password-less access via public key authentication.

The following options, common to all of these programs, allow you to confgure access to the API. These options may be set on the command line or via a configuration file common to all of the programs.

`-clipath` ***path***
Specify the *path* used to invoke the `bitcoin-cli` program on the node machine. This option is used for the `local` and `ssh` access methods. Note that on an SSH login, the user's terminal login scripts are not executed, so you may have to specify an explicit path even if `bitcoin-cli` is in a directory included in the `PATH` declared by those scripts.

`-host` ***hostname***
Specifies the host (machine network name) on which Bitcoin Core is running. If this is the same computer, use `localhost`, otherwise specify the local machine name, fully qualified domain name, or IP address of the machine.

`-method` ***which***
Sets the method used to access the API. Use `local` if accessing a Bitcoin Core node on the same machine, or `ssh` to access a Bitcoin Core node on another machine. The `rpc` option selects direct access via the RPC interface on the same or a different host. RPC access is the most efficient and should be used if available.

`-rpccpass` ***password***
Set the password for access via the `rpc` method. This password is configured in the `bitcoin.conf` file via the `rpcpassword` statement. If the *password* specified is the null string (`""`), the user will be prompted to enter the password from the console, which is far more secure than specifying it on the command line.

`-port` ***number***

Sets the port used to communicate with the Bitcoin Core node when the `rpc` method is selected. The default is 8332.

`-user` ***userid***

Sets the User ID (login name) for access to a Bitcoin Core node on another machine via the `ssh` method.

2.10 Installation

Fourmilab Blockchain Tools are written in the Perl and Python programming languages, which are pre-installed on most modern versions of Unix-like operating systems such as Linux, FreeBSD, and Macintosh OS X, and available for many other systems. Consequently, you can run any of the pre-built versions of the tools, all of which have file types of "`.pl`" or "`.py` by simply invoking them with the `perl` or `python3` commands. The programs use a number of modules, some of which are "core" or "standard"—included as part of current language distributions, and others which may have to be installed either from the operating system's software library or the Comprehensive Perl Archive Network and its search engine, MetaCPAN ot with the `pip3` utility for Python. If a module is available from your operating system's distribution library, that's generally the best way to install it, since it will be automatically updated by the system's software update mechanism.

2.10.1 Required Perl modules

Here is a list of all Perl modules used by the programs. Not all programs use all modules: if you're only interested in some of the programs, you need only install those they require. Modules marked as "*core*" will be pre-installed on most modern versions of Perl.

- `Bitcoin::BIP39`
- `Bitcoin::Crypto::Key::Private`
- `Bitcoin::Crypto::Key::Public`
- `Crypt::CBC`
- `Crypt::Digest::Keccak256`
- `Crypt::OpenSSL::AES`
- `Crypt::Random::Seed`
- `Crypt::SSSS`
- `Data::Dumper` *core*
- `Digest::SHA` *core*
- `Digest::SHA3`
- `Getopt::Long` *core*
- `JSON` *core*
- `List::Util` *core*
- `LWP::Protocol::https`
- `LWP::Simple`
- `LWP`
- `MIME::Base64` *core*
- `Math::Random::MT`
- `POSIX` *core*
- `Statistics::Descriptive`
- `Term::ReadKey`
- `Text::CSV`

2.10.2 Required Python modules

To avoid commonality in language and libraries in the interest of avoiding single points of failure when validating the correctness of generated wallets, the `validate_wallet` program is written in the Python language (version 3 or greater), and requires the following modules be installed on systems that run it. Modules marked "*standard*" are part of Python's standard libraries should be installed on most systems that support the language. If you don't run `validate_wallet`, you needn't bother installing these modules.

- `base58`
- `binascii` *standard*
- `coincurve`
- `cryptos`
- `fileinput` *standard*
- `re` *standard*
- `sha3`
- `sys` *standard*

2.10.3 Building from original source code

This software, including all programs, support files, utilities, and documentation was developed using the Literate Programming methodology, where the goal is that programs should be as readable to humans as they are by computers. The package is written using the **nuweb** literate programming system, which is language-agnostic: it can be used to develop software in any programming language, including multiple languages in a single project, as is the case for this one. The **nuweb** tools are free software written in portable C, with source code downloadable from the link above.

Programs in **nuweb** are called "Web files", which have nothing whatsoever to do with the World-Wide Web (which it predates), having a file type of "`.w`". All of the other files in the distribution are generated automatically from the master Web. If you wish to modify one or more of the programs, it's best to modify the master code in the Web file and re-generate the programs from it. All of the building and maintenance operations are performed by a `Makefile` which is, itself, generated from the Web. If you edit any of the files associated with this program, be sure to use a text editor which supports the Unicode-compatible UTF-8 character set: otherwise some special characters may be turned into gibberish.

Documentation is generated automatically in the LaTeX document preparation language, with the final PDF documents produced with XeTeX, a version of TeX extended to support the full Unicode character set. These utilities can be installed from the distribution archives of most Unix-like systems.

2.10.4 Configuration parameters

When you build from source code, a number of build-time configuration parameters are incorporated from the Web file `configuration.w`. Please see the documentation for that file in the source code listing, in the Introduction chapter, section "Configuration". Most of the configuration parameters set defaults which can be overridden by command-line options, so usually setting them is a convenience to avoid having to specify the options you prefer, not a necessity.

2.11 License and Disclaimer of Warranty and Liability

This product (software, documents, and data files) is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License (legal text). You are free to copy and redistribute this material in any medium or format, and to remix, transform, and build upon the material for any purpose, including commercially. You must give credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon this material, you must distribute your contributions under the same license as the original.

This product is provided with no warranty, either expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose, regarding these materials and is made available available solely on an “as-is” basis.

In no event shall John Walker be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of distribution or use of these materials. The sole and exclusive liability of John Walker, regardless of the form of action, shall not exceed the compensation received by the author for the product.

John Walker reserves the right to revise and improve this products as he sees fit. This publication describes the state of this product at the time of its publication, and may not reflect the product at all times in the future.

In particular, no claims are made for, or warranty of, the correctness of results produced by these programs, or security of them, and no liability shall result from their use or misuse. Before sending funds to any cryptocurrency address, it is *essential* to verify that you possess the correct private key to retrieve them, and that this key is stored securely in a manner that protects it from loss, theft, or destruction. Because the correctness and security of any computer system depends upon not just the applications running on it, but the language and system libraries they use, the underlying operating system, the hardware on which it runs, and the personnel and procedures which operate it, it is entirely the responsibility of the user to independently verify the correctness of any results it produces and to satisfy themselves of their security for the intended application.

Chapter 3

Blockchain Address Generator

This program generates Bitcoin and Ethereum public address and private key pairs from a variety of sources of random and pseudorandom data, including Fourmilab's HotBits radioactive random number generator. The program is implemented as a stack machine where command line "options" are actually commands and arguments that allow specification, generation, and manipulation of random and pseudorandom data, generation of Bitcoin and Ethereum private keys and public addresses from them, and their output in a variety of formats.

3.1 Main program

3.1.1 Program plumbing

```
"perl/blockchain_address.pl" 25≡
     ⟨ Explanatory header for Perl files 145a ⟩

         ⟨ Perl language modes 144a ⟩

         #   Configured HotBits access
         my $HotBits_API_key = "⟨ HotBits API key 4b ⟩";
         my $HotBits_Query = "⟨ HotBits query URL 4a ⟩";

         use Bitcoin::Crypto::Key::Private;
         use Bitcoin::Crypto::Key::Public;
         use Bitcoin::BIP39 qw(entropy_to_bip39_mnemonic bip39_mnemonic_to_entropy);
         use Digest::SHA qw(sha256 sha256_hex);
         use Digest::SHA3 qw(sha3_256_hex);
         use Crypt::CBC;
         use Crypt::Digest::Keccak256 qw(keccak256_hex);
         use Crypt::Random::Seed;
         use MIME::Base64;
         use LWP::Simple;
         use Getopt::Long qw(GetOptionsFromArray);
         use Data::Dumper;
     ◇
```

File defined by 25, 27, 28.

3.1.2 Process command line options

If project- or program-level configuration files are present, process them, then process command line options.

"perl/blockchain_address.pl" 27≡

```
my $opt_Format = "";        # Format for generated keys

my $repeat = 1;             # Repeat command this number of times
my $outputFile = "-";       # Output file for keys
my $testMode = 0;           # Bit-coded test modes
my @seeds;                 # Stack of seeds

my %options = (
    "aesenc"    =>  \&arg_aesenc,
    "aesdec"    =>  \&arg_aesdec,
    "bindump=s" =>  \&arg_bindump,
    "binfile=s" =>  \&arg_binfile,
    "btc"       =>  \&arg_btc,
    "clear"     =>  \&arg_clear,
    "drop"      =>  \&arg_drop,
    "dump"      =>  \&arg_dump,
    "dup"       =>  \&arg_dup,
    "eth"       =>  \&arg_eth,
    "format=s"  =>  \$opt_Format,
    "hbapik=s"  =>  \$HotBits_API_key,
    "help"      =>  \&showHelp,
    "hexfile=s" =>  \&arg_hexfile,
    "hotbits"   =>  \&arg_hotbits,
    "inter"     =>  \&arg_inter,
    "minigen"   =>  \&arg_minigen,
    "minikey=s" =>  \&arg_minikey,
    "mnemonic"  =>  \&arg_mnemonic,
    "not"       =>  \&arg_not,
    "outfile=s" =>  \&arg_outfile,
    "over"      =>  \&arg_over,
    "p"         =>  \&arg_printtop,
    "phrase=s"  =>  \&arg_phrase,
    "pick=i"    =>  \&arg_pick,
    "pseudo"    =>  \&arg_pseudo,
    "pseudoseed"=>  \&arg_pseudoseed,
    "random"    =>  \&arg_random,
    "repeat=i"  =>  \$repeat,
    "roll=i"    =>  \&arg_roll,
    "rot"       =>  \&arg_rot,
    "rrot"      =>  \&arg_rrot,
    "seed=s"    =>  \&arg_seed,
    "sha2"      =>  \&arg_sha2,
    "sha3"      =>  \&arg_sha3,
    "shuffle"   =>  \&arg_shuffle,
    "swap"      =>  \&arg_swap,
    "test"      =>  \&arg_test,
    "testall"   =>  \&arg_testall,
    "testmode=i" => \$testMode,
    "type=s"    =>  sub { print("$_[1]\n"); },
    "urandom"   =>  \&arg_urandom,
    "wif=s"     =>  \&arg_wif,
    "xor"       =>  \&arg_xor,
    "zero"      =>  \&arg_zero
);

processConfiguration();

GetOptions(
           %options
          ) ||
    die("Invalid command line option");

if ($testMode & 4) {
    print("Modules used\n  ",
```

Include local and utility functions we employ.

`"perl/blockchain_address.pl"` 28≡

```
        #   Shared utility functions
        ⟨ readHexfile: Read hexadecimal data from a file 141 ⟩
        ⟨ Pseudorandom number generator 142a, ... ⟩
        ⟨ Command and option processing 133, ... ⟩

        #   Local functions
        ⟨ Command line argument handlers 29 ⟩
        ⟨ hexToBytes: Convert hexadecimal string to binary 57a ⟩
        ⟨ bytesToHex: Convert binary string to hexadecimal 57b ⟩
        ⟨ genBtcAddress: Generate Bitcoin address from one hexadecimal seed 47c, ... ⟩
        ⟨ editBtcAddress: Edit Bitcoin private key and public address 48c, ... ⟩
        ⟨ findMiniKey: Find a Bitcoin mini key 58 ⟩
        ⟨ genEthAddress: Generate Ethereum address from one hexadecimal seed 52a, ... ⟩
        ⟨ editEthAddress: Edit Ethereum private key and public address 53a, ... ⟩
        ⟨ computeEthChecksum: Add checksum to Ethereum address 55 ⟩
        ⟨ BIP39encode: Encode seed as BIP39 mnemonic phrase 56a ⟩
        ⟨ shuffleBytes: Shuffle bytes 143 ⟩
        ⟨ showHelp: Show Bitcoin address help information 59, ... ⟩
        ⟨ stackCheck: Check for stack underflow 56b ⟩
    ◇
```

File defined by 25, 27, 28.

3.2 Local functions

3.2.1 Command line argument handlers

⟨ *Command line argument handlers* 29 ⟩ ≡

```
        ⟨ arg_aesenc: Encrypt second item with top of stack key 30a ⟩
        ⟨ arg_aesdec: Decrypt second item with top of stack key 30b ⟩
        ⟨ arg_bindump: Dump seeds from stack to binary file 31a ⟩
        ⟨ arg_binfile: Push seeds from binary file on stack 31b ⟩
        ⟨ arg_btc: Generate Bitcoin key/address from top of stack 32a ⟩
        ⟨ arg_clear: Clear stack 32b ⟩
        ⟨ arg_drop: Drop the top item from the stack 32c ⟩
        ⟨ arg_dump: Dump the stack 33a ⟩
        ⟨ arg_dup: Duplicate the top item from the stack 33b ⟩
        ⟨ arg_eth: Generate Ethereum key/address from top of stack 34a ⟩
        ⟨ arg_hexfile: Push seeds from hexfile on stack 34b ⟩
        ⟨ arg_hotbits: Request seed(s) from HotBits 35a ⟩
        ⟨ arg_minigen: Find Bitcoin mini private key 35b ⟩
        ⟨ arg_minikey: Decode Bitcoin mini private key 36 ⟩
        ⟨ arg_mnemonic: Generate mnemonic phrase from stack top 37a ⟩
        ⟨ arg_not: Invert bits in top of stack item 37b ⟩
        ⟨ arg_outfile: Redirect generated address output to file 37c ⟩
        ⟨ arg_over: Duplicate the second item from the stack 38a ⟩
        ⟨ arg_pick: Duplicate the nth item from the stack 39a ⟩
        ⟨ arg_pseudo: Generate pseudorandom seed and push on stack 39b ⟩
        ⟨ arg_pseudoseed: Set pseudorandom generator seed 40a ⟩
        ⟨ arg_phrase: Specify seed as BIP39 phrase 38b ⟩
        ⟨ arg_printtop: Print top of stack 38c ⟩
        ⟨ arg_random: Request seed(s) from strong generator 40b ⟩
        ⟨ arg_roll: Rotate item n to top of stack 41a ⟩
        ⟨ arg_rot: Rotate three stack items 41b ⟩
        ⟨ arg_rrot: Reverse rotate three stack items 41c ⟩
        ⟨ arg_seed: Push seed on stack 42a ⟩
        ⟨ arg_sha2: Replace top of stack with SHA2-256 hash 42b ⟩
        ⟨ arg_sha3: Replace top of stack with SHA3-256 hash 42c ⟩
        ⟨ arg_shuffle: Shuffle bytes on stack 43a ⟩
        ⟨ arg_swap: Swap the two top items on the stack 43b ⟩
        ⟨ arg_test: Test stack items for randomness 44a ⟩
        ⟨ arg_testall: Test entire stack contents for randomness 44b ⟩
        ⟨ arg_urandom: Request seed(s) from fast generator 45a ⟩
        ⟨ arg_wif: Load seed from Wallet Input Format (WIF) private key 45b ⟩
        ⟨ arg_xor: Exclusive-or top two stack items 46a ⟩
        ⟨ arg_zero: Push all zeroes on the stack 46b ⟩
```

◇

Fragment referenced in 28.

3.2.1.1 arg_aesenc — -aesenc: Encrypt second item with top of stack key

⟨ *arg_aesenc: Encrypt second item with top of stack key* 30a ⟩ ≡

```
        sub arg_aesenc {
            stackCheck(2);
            my $key = hexToBytes(pop(@seeds));
            my $plaintext = hexToBytes(pop(@seeds));
            my $crypt = Crypt::CBC->new(
                            -chain_mode => "cfb",
                            -pbkdf  => "none",
                            -header => "none",
                            -key    => $key,
                            -iv     => substr(sha256($key), 0, 16),
                            -cipher => "Crypt::OpenSSL::AES");
            push(@seeds, bytesToHex($crypt->encrypt($plaintext)));
        }
```

◇

Fragment referenced in 29.
Defines: `arg_aesenc` 27.
Uses: `bytesToHex` 57b, `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.2 arg_aesdec — -aesdec: Decrypt second item with top of stack key

⟨ *arg_aesdec: Decrypt second item with top of stack key* 30b ⟩ ≡

```
        sub arg_aesdec {
            stackCheck(2);
            my $key = hexToBytes(pop(@seeds));
            my $codetext = hexToBytes(pop(@seeds));
            my $crypt = Crypt::CBC->new(
                            -chain_mode => "cfb",
                            -pbkdf  => "none",
                            -header => "none",
                            -key    => $key,
                            -iv     => substr(sha256($key), 0, 16),
                            -cipher => "Crypt::OpenSSL::AES");
            push(@seeds, bytesToHex($crypt->decrypt($codetext)));
        }
```

◇

Fragment referenced in 29.
Defines: `arg_aesdec` 27.
Uses: `bytesToHex` 57b, `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.3 `arg_bindump` — `-bindump`: Dump seeds from stack to binary file

⟨ *arg_bindump: Dump seeds from stack to binary file* 31a ⟩ ≡

```
sub arg_bindump {
    my ($name, $value) = @_;

    if (open(BO, ">$value")) {
        foreach my $seed (@seeds) {
            print(BO hexToBytes($seed));
        }
        close(BO);
    } else {
        print("Cannot create file $value.\n");
    }
}
```

◇

Fragment referenced in 29.
Defines: `arg_binfile` 27, 31b.
Uses: `hexToBytes` 57a.

3.2.1.4 `arg_binfile` — `-binfile`: Push seeds from binary file on stack

⟨ *arg_binfile: Push seeds from binary file on stack* 31b ⟩ ≡

```
sub arg_binfile {
    my ($name, $value) = @_;

    open(BI, "<$value") || die("Cannot open $value");
    my $dat;
    while (read(BI, $dat, 32) == 32) {
        push(@seeds, bytesToHex($dat));
    }
    close(BI);
}
```

◇

Fragment referenced in 29.
Defines: `arg_binfile` 27.
Uses: `bytesToHex` 57b.

3.2.1.5 arg_btc — -btc: Generate Bitcoin key/address

⟨ *arg_btc: Generate Bitcoin key/address from top of stack* 32a ⟩ ≡

```
sub arg_btc {
    my ($name, $value) = @_;
    stackCheck($repeat);

    my $keyn = 1;
    my $keep = ($opt_Format =~ m/k/);
    my @kept;
    ⟨ Open output file 47a ⟩
    ⟨ Begin command repeat 46c ⟩
        my $seed = pop(@seeds);
        if ($keep) {
            push(@kept, $seed);
        }
        my ($priv, $pub) = genBtcAddress($seed, $opt_Format, 1);
        print(editBtcAddress($priv, $pub, $opt_Format, $keyn++, ""));
    ⟨ End command repeat 46d ⟩
    ⟨ Close output file 47b ⟩
    if ($keep) {
        ⟨ Begin command repeat 46c ⟩
            push(@seeds, pop(@kept));
        ⟨ End command repeat 46d ⟩
    }
}
```

◇

Fragment referenced in 29.
Defines: `arg_btc` 27.
Uses: `editBtcAddress` 48c, `genBtcAddress` 47c, `stackCheck` 56b.

3.2.1.6 arg_clear — -clear: Clear stack

⟨ *arg_clear: Clear stack* 32b ⟩ ≡

```
sub arg_clear {
    @seeds = ();
}
```

◇

Fragment referenced in 29.
Defines: `arg_clear` 27.

3.2.1.7 arg_drop — -drop: Drop the top item from the stack

⟨ *arg_drop: Drop the top item from the stack* 32c ⟩ ≡

```
sub arg_drop {
    stackCheck(1);
    pop(@seeds);
}
```

◇

Fragment referenced in 29.
Defines: `arg_drop` 27.
Uses: `stackCheck` 56b.

3.2.1.8 arg_dump — -dump: Dump the stack

⟨ *arg_dump: Dump the stack* 33a ⟩ ≡

```
        sub arg_dump {
            ⟨ Open output file 47a ⟩
            if ($outputFile eq "-") {
                print("  ", join("\n  ", reverse(@seeds)), "\n");
            } else {
                print(join("\n", @seeds), "\n");
            }
            ⟨ Close output file 47b ⟩
        }
```

◇

Fragment referenced in 29.
Defines: `arg_dump` 27.

3.2.1.9 arg_dup — -dup: Duplicate the top item from the stack

⟨ *arg_dup: Duplicate the top item from the stack* 33b ⟩ ≡

```
        sub arg_dup {
            stackCheck(1);
            push(@seeds, $seeds[$#seeds]);
        }
```

◇

Fragment referenced in 29.
Defines: `arg_dup` 27.
Uses: `stackCheck` 56b.

3.2.1.10 arg_eth — -eth: Generate Ethereum key/address

⟨ *arg_eth: Generate Ethereum key/address from top of stack* 34a ⟩ ≡

```
        sub arg_eth {
            my ($name, $value) = @_;
            stackCheck($repeat);

            my $keyn = 1;
            my $keep = ($opt_Format =~ m/k/);
            my @kept;
            ⟨ Open output file 47a ⟩
            ⟨ Begin command repeat 46c ⟩
                my $seed = pop(@seeds);
                if ($keep) {
                    push(@kept, $seed);
                }
                my ($priv, $pub) = genEthAddress($seed, $opt_Format, 1);
                print(editEthAddress($priv, $pub, $opt_Format, $keyn++));
            ⟨ End command repeat 46d ⟩
            ⟨ Close output file 47b ⟩
            if ($keep) {
                ⟨ Begin command repeat 46c ⟩
                    push(@seeds, pop(@kept));
                ⟨ End command repeat 46d ⟩
            }
        }
    ◇
```

Fragment referenced in 29.
Defines: `arg_eth` 27.
Uses: `editEthAddress` 53a, `genEthAddress` 52a, `stackCheck` 56b.

3.2.1.11 arg_hexfile — -hexfile: Push seeds from hexfile on stack

⟨ *arg_hexfile: Push seeds from hexfile on stack* 34b ⟩ ≡

```
        sub arg_hexfile {
            my ($name, $value) = @_;

            my $hf = readHexfile($value);
            while ($hf =~ s/^([\dA-F]{64})//i) {
                push(@seeds, uc($1));
            }
        }
    ◇
```

Fragment referenced in 29.
Defines: `arg_hexfile` 27.
Uses: `readHexfile` 141.

3.2.1.12 `arg_hotbits` — `-hotbits`: Request seed(s) from HotBits

⟨ *arg_hotbits: Request seed(s) from HotBits* 35a ⟩ ≡

```
    sub arg_hotbits {
        my $n = 32 * $repeat;
        my $hbq = $HotBits_Query;
        $hbq =~ s/\[NBYTES\]/$n/;
        $hbq =~ s/\[APIKEY\]/$HotBits_API_key/;
        my $hbr = get($hbq);
        $hbr =~ m:<pre>(.*?\w+.*?)</pre>:s || die("Cannot parse HotBits reply: $hbr");
        my $hf = $1;
        $hf =~ s/\W//g;
        while ($hf =~ s/^([\dA-F]{64})//i) {
            push(@seeds, $1);
        }
    }
```

◇

Fragment referenced in 29.
Defines: `arg_hotbits` 27.

3.2.1.13 `arg_minigen` — `-minigen`: Find Bitcoin mini private key

Find a Bitcoin mini key, push the seed to which it corresponds onto the stack, and output the mini key. This command responds to the `-repeat` setting to create multiple mini keys.

⟨ *arg_minigen: Find Bitcoin mini private key* 35b ⟩ ≡

```
    sub arg_minigen {
        my $keyn = 1;
        my $keep = ($opt_Format =~ m/k/);
        my @kept;
        ⟨ Open output file 47a ⟩
        ⟨ Begin command repeat 46c ⟩
            my ($minikey, $privkey) = findMiniKey();
# my ($minikey, $privkey) = ("SoEGNB7gNfVnnjUFK4YYmuPgJD2t7M", "A71E1257AEF63B24BBB37D294C7E09096F921622D6D926
            if ($keep) {
                push(@kept, $privkey);
            }
            my ($priv, $pub) = genBtcAddress($privkey, $opt_Format, 1);
            print(editBtcAddress($priv, $pub, $opt_Format, $keyn++, $minikey));
        ⟨ End command repeat 46d ⟩
        ⟨ Close output file 47b ⟩
        if ($keep) {
            ⟨ Begin command repeat 46c ⟩
                push(@seeds, pop(@kept));
            ⟨ End command repeat 46d ⟩
        }
    }
```

◇

Fragment referenced in 29.
Defines: `arg_minigen` 27.
Uses: `editBtcAddress` 48c, `genBtcAddress` 47c.

3.2.1.14 arg_minikey — -minikey *key*: Decode seed from mini private key

Decode a mini private key and push the result on the stack. Mini private keys are used in some physical forms of Bitcoin and applications where a compact representation of a private key is desirable. They are 30 character sequences (22 characters for early Casascius Series 1 coins, but discouraged due to its limited security) consisting of the letter "S" followed by characters from the Bitcoin base58 character set. A valid mini key will, when the character "?" is appended, have an SHA2-256 hash whose first byte is zero. The private key is obtained simply by taking the SHA2-256 hash of the entire mini key itself, including the "S". There is no way to practically generate a mini-key from an arbitrary key—mini keys are discovered by a randomised search for strings which pass the validity hash test. We accept both 22- and 30-character mini keys.

⟨ *arg_minikey: Decode Bitcoin mini private key* 36 ⟩ ≡

```
sub arg_minikey {
    my ($name, $value) = @_;

    my $goof = "";
    if ($value =~ m/^S/) {
        if (($value =~ m/^S([\w]{29})$/) ||
            ($value =~ m/^S([\w]{21})$/)) {
            my $adr = $1;
            if ($adr =~ m/^[1-9A-HJ-NP-Za-km-z]+$/) {
                if (sha256_hex("$value?") =~ m/^00/) {
                    push(@seeds, uc(sha256_hex($value)));
                } else {
                    $goof = "bad checksum";
                }
            } else {
                $goof = "invalid character in key";
            }
        } else {
            $goof = "incorrect length";
        }
    } else {
        $goof = "does not start with \"S\"";
    }
    if ($goof) {
        print("Invalid mini key: $goof.\n");
    }
}
```

◇

Fragment referenced in 29.
Defines: arg_minikey 27.

3.2.1.15 arg_mnemonic — -mnemonic: Generate BIP39 mnemonic phrase from stack top

⟨ *arg_mnemonic: Generate mnemonic phrase from stack top* 37a ⟩ ≡

```
sub arg_mnemonic {
    stackCheck(1);
    print(BIP39encode("  ", pop(@seeds), 64));
}
```

◇

Fragment referenced in 29.
Defines: `arg_mnemonic` Never used.
Uses: `BIP39encode` 56a, `stackCheck` 56b.

3.2.1.16 arg_not — -not: Invert bits in top of stack item

⟨ *arg_not: Invert bits in top of stack item* 37b ⟩ ≡

```
sub arg_not {
    stackCheck(1);
    my $b = hexToBytes(pop(@seeds));
    my $bi = "";
    for (my $i = 0; $i < bytes::length($b); $i++) {
        $bi .= sprintf("%02X", ord(bytes::substr($b, $i, 1)) ^ 0xFF);
    }
    push(@seeds, $bi);
}
```

◇

Fragment referenced in 29.
Defines: `arg_not` 27.
Uses: `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.17 arg_outfile — -outfile *filename* Redirect generated address output to file

⟨ *arg_outfile: Redirect generated address output to file* 37c ⟩ ≡

```
sub arg_outfile {
    my ($name, $value) = @_;

    $outputFile = $value;
}
```

◇

Fragment referenced in 29.
Defines: `arg_outfile` 27.

3.2.1.18 arg_over — -over: Duplicate the second item from the stack

⟨ *arg_over: Duplicate the second item from the stack* 38a ⟩ ≡

```
        sub arg_over {
            stackCheck(2);
            push(@seeds, $seeds[$#seeds - 1]);
        }
```

◇

Fragment referenced in 29.
Defines: arg_over 27.
Uses: stackCheck 56b.

3.2.1.19 arg_phrase — -phrase: Specify seed as BIP39 phrase

⟨ *arg_phrase: Specify seed as BIP39 phrase* 38b ⟩ ≡

```
        sub arg_phrase {
            my ($name, $value) = @_;

            my $seed = bip39_mnemonic_to_entropy(
                mnemonic => $value,
                encoding => "hex");
            push(@seeds, uc($seed));
        }
```

◇

Fragment referenced in 29.
Defines: arg_phrase 27.

⟨ *arg_printtop: Print top of stack* 38c ⟩ ≡

```
        sub arg_printtop {
            if (scalar(@seeds) > 0) {
                print("  $seeds[-1]\n");
            } else {
                print("Stack empty.\n");
            }
        }
```

◇

Fragment referenced in 29.
Defines: arg_printtop 27.

3.2.1.20 arg_pick — -pick *n*: Duplicate the *n*th item from the stack

⟨ *arg_pick: Duplicate the nth item from the stack* 39a ⟩ ≡

```
        sub arg_pick {
            my ($name, $value) = @_;

            stackCheck($value + 1);
            push(@seeds, $seeds[$#seeds - $value]);
        }
```

◇

Fragment referenced in 29.
Defines: `arg_pick` 27.
Uses: `stackCheck` 56b.

3.2.1.21 arg_pseudo — -pseudo: Generate pseudorandom seed and push on stack

⟨ *arg_pseudo: Generate pseudorandom seed and push on stack* 39b ⟩ ≡

```
        sub arg_pseudo {
            randInit();
            ⟨ Begin command repeat 46c ⟩
                my $s = "";
                for (my $i = 0; $i < 32; $i++) {
                    $s .= sprintf("%02X", randNext(256));
                }
                push(@seeds, $s);
            ⟨ End command repeat 46d ⟩
        }
```

◇

Fragment referenced in 29.
Defines: `arg_pseudo` 27.
Uses: `randInit` 142a, `randNext` 142b.

3.2.1.22 arg_pseudoseed — -pseudoseed: Set pseudorandom generator seed

⟨ *arg_pseudoseed: Set pseudorandom generator seed* 40a ⟩ ≡

```
    sub arg_pseudoseed {
        my $orepeat = $repeat;
        if ($repeat > 78) {
            $repeat = 78;
        }
        stackCheck($repeat);

        my $allbytes = "";
        ⟨ Begin command repeat 46c ⟩
            $allbytes .= hexToBytes(pop(@seeds));
        ⟨ End command repeat 46d ⟩

        my @pseed = unpack("L4", $allbytes);
        $randGen = Math::Random::MT->new(@pseed);
        $repeat = $orepeat;
    }
```

◇

Fragment referenced in 29.
Defines: `arg_pseudoseed` 27.
Uses: `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.23 arg_random — -random: Request seed(s) from strong generator

⟨ *arg_random: Request seed(s) from strong generator* 40b ⟩ ≡

```
    sub arg_random {
        my $n = 32 * $repeat;
        my $rgen = new Crypt::Random::Seed;
        if (defined($rgen)) {
            my $rbytes = $rgen->random_bytes($n);
            while ($rbytes =~ s/^(.{32})//s) {
                my $hn = $1;
                push(@seeds, bytesToHex($hn));
            }
        } else {
            print("No strong random generator available.");
        }
    }
```

◇

Fragment referenced in 29.
Defines: `arg_random` 27.
Uses: `bytesToHex` 57b.

3.2.1.24 `arg_roll` — `-roll` *n*: Rotate item *n* to top of stack

⟨ *arg_roll: Rotate item n to top of stack* 41a ⟩ ≡

```
sub arg_roll {
    my ($name, $value) = @_;

    stackCheck($value + 1);
    my $itemn = splice(@seeds, -($value + 1), 1);
    push(@seeds, $itemn);
}
```

◇

Fragment referenced in 29.
Defines: `arg_roll` 27.
Uses: `stackCheck` 56b.

3.2.1.25 `arg_rot` — `-rot`: Rotate three stack items

⟨ *arg_rot: Rotate three stack items* 41b ⟩ ≡

```
sub arg_rot {
    stackCheck(3);
    my $item3 = splice(@seeds, -3, 1);
    push(@seeds, $item3);
}
```

◇

Fragment referenced in 29.
Defines: `arg_rot` 27.
Uses: `stackCheck` 56b.

3.2.1.26 `arg_rrot` — `-rrot`: Reverse rotate three stack items

⟨ *arg_rrot: Reverse rotate three stack items* 41c ⟩ ≡

```
sub arg_rrot {
    stackCheck(3);
    my $item1 = pop(@seeds);
    splice(@seeds, 2, 0, $item1);
}
```

◇

Fragment referenced in 29.
Defines: `arg_rrot` 27.
Uses: `stackCheck` 56b.

3.2.1.27 arg_seed — -seed ***hex*****: Push seed on stack**

⟨ *arg_seed: Push seed on stack* 42a ⟩ ≡

```
sub arg_seed {
    my ($name, $value) = @_;

    $value =~ s/^0x//i;
    if ($value !~ m/^[\dA-F]{64}/i) {
        die("Invalid seed.  Must be 64 hexadecimal digits");
    }
    push(@seeds, uc($value));
}
```

◇

Fragment referenced in 29.
Defines: `arg_seed` 27.

3.2.1.28 arg_sha2 — -sha2: Replace top of stack with its SHA2-256 hash

⟨ *arg_sha2: Replace top of stack with SHA2-256 hash* 42b ⟩ ≡

```
sub arg_sha2 {
    stackCheck($repeat);

    my $bytes = "";
    ⟨ Begin command repeat 46c ⟩
        $bytes .= hexToBytes(pop(@seeds));
    ⟨ End command repeat 46d ⟩
    my $sha2_256 = uc(sha256_hex($bytes));
    push(@seeds, $sha2_256);
}
```

◇

Fragment referenced in 29.
Defines: `arg_sha2` 27.
Uses: `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.29 arg_sha3 — -sha3: Replace top of stack with its SHA3-256 hash

⟨ *arg_sha3: Replace top of stack with SHA3-256 hash* 42c ⟩ ≡

```
sub arg_sha3 {
    stackCheck($repeat);

    my $bytes = "";
    ⟨ Begin command repeat 46c ⟩
        $bytes .= hexToBytes(pop(@seeds));
    ⟨ End command repeat 46d ⟩
    my $sha3_256 = uc(sha3_256_hex($bytes));
    push(@seeds, $sha3_256);
}
```

◇

Fragment referenced in 29.
Defines: `arg_sha3` 27.
Uses: `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.30 `arg_shuffle` — `-shuffle`: Shuffle bytes on stack

Shuffle the bytes of items on the stack. Why would you want to do this? Suppose, for example, you've obtained entropy from a source on the Internet and, despite retrieving it using `https:`, are worried about the data being intercepted along the way or archived by the site that generated it. You can allay that risk, in part, by generating a much larger quantity of data than you need, shuffling the bytes using a different seed generated locally, then using a key from the shuffled bytes. If the sample from which you select your actual key is sufficiently large, guessing which bytes were chosen is intractable. The number of items shuffled may be set with `-repeat`.

⟨ *arg_shuffle: Shuffle bytes on stack* 43a ⟩ ≡

```
    sub arg_shuffle {
        stackCheck($repeat);

        my $allbytes = "";
        ⟨ Begin command repeat 46c ⟩
            $allbytes .= hexToBytes(pop(@seeds));
        ⟨ End command repeat 46d ⟩
        my $sbytes = bytesToHex(shuffleBytes($allbytes));
        while ($sbytes =~ s/^(\w{64})//) {
            push(@seeds, $1);
        }
    }
```

◇

Fragment referenced in 29.
Defines: `arg_shuffle` 27.
Uses: `bytesToHex` 57b, `hexToBytes` 57a, `shuffleBytes` 143, `stackCheck` 56b.

3.2.1.31 `arg_swap` — `-swap`: Swap the two top items on the stack

⟨ *arg_swap: Swap the two top items on the stack* 43b ⟩ ≡

```
    sub arg_swap {
        my ($name, $value) = @_;

        stackCheck(2);
        my $itemn = splice(@seeds, -2, 1);
        push(@seeds, $itemn);
    }
```

◇

Fragment referenced in 29.
Defines: `arg_swap` 27.
Uses: `stackCheck` 56b.

3.2.1.32 `arg_test` — `-test`: Test stack items for randomness

Take the seed(s) on the top of the stack and feed to `ent` to perform an analysis of its randomness. The number of seeds tested may be set with `-repeat`. Note that when interpreting these results, the brevity of the data may cause them to appear less than random compared to a larger sample. We perform the randomness tests on a bit-level basis, as byte-level tests are useless on small samples.

⟨ *arg_test: Test stack items for randomness* 44a ⟩ ≡

```
        sub arg_test {
            stackCheck($repeat);

            my $catseed = join("", @seeds[0 .. ($repeat - 1)]);
            my $r = "Randomness analysis:\n";
            my $ent_analysis = `echo $catseed | xxd -r -p - | ent -b`;
            $ent_analysis =~ s/\n\n/\n/gs;
            $ent_analysis =~ s/^/    /mg;
            $ent_analysis =~ s/(of this|would exceed)/  $1/gs;
            print("$ent_analysis");
        }
```

◇

Fragment referenced in 29.
Defines: `arg_test` 27.
Uses: `stackCheck` 56b.

3.2.1.33 `arg_testall` — `-testall`: Test randomness of entire stack

Tests the entire contents of the stack for randomness with `ent`. The stack items are concatenated, from top to bottom, and the resulting bit stream tested. This can be used to evaluate random sources used to generate multiple keys.

⟨ *arg_testall: Test entire stack contents for randomness* 44b ⟩ ≡

```
        sub arg_testall {
            stackCheck(1);
            my $catseed = join("", @seeds);
            my $r = "Randomness analysis:\n";
            my $ent_analysis = `echo $catseed | xxd -r -p - | ent -b`;
            $ent_analysis =~ s/\n\n/\n/gs;
            $ent_analysis =~ s/^/    /mg;
            $ent_analysis =~ s/(of this|would exceed)/  $1/gs;
            print("$ent_analysis");
        }
```

◇

Fragment referenced in 29.
Defines: `arg_testall` 27.
Uses: `stackCheck` 56b.

3.2.1.34 arg_urandom — -urandom: Request seed(s) from fast generator

⟨ *arg_urandom: Request seed(s) from fast generator* 45a ⟩ ≡

```
sub arg_urandom {
    my $n = 32 * $repeat;
    my $rgen = Crypt::Random::Seed->new(NonBlocking => 1);
    if (defined($rgen)) {
        my $rbytes = $rgen->random_bytes($n);
        while ($rbytes =~ s/^(.{32})//s) {
            my $hn = $1;
            push(@seeds, bytesToHex($hn));
        }
    } else {
        print("No fast random generator available.");
    }
}
```

◇

Fragment referenced in 29.
Defines: `arg_urandom` 27.
Uses: `bytesToHex` 57b.

3.2.1.35 arg_wif — -wif *key*: Push seed extracted from Wallet Input Format (WIF) private key on stack

Extract the seed from the private key argument supplied in Wallet Import Format (WIF) and push the seed on the stack.

⟨ *arg_wif: Load seed from Wallet Input Format (WIF) private key* 45b ⟩ ≡

```
sub arg_wif {
    my ($name, $value) = @_;

    my $priv = Bitcoin::Crypto::Key::Private->from_wif($value);
    my $seed = $priv->to_hex();
    push(@seeds, uc($seed));
}
```

◇

Fragment referenced in 29.
Defines: `arg_wif` 27.

3.2.1.36 arg_xor — -xor: Exclusive-or top two stack items

Exclusive-or the two top items on the stack, removing them and pushing the result.

⟨ *arg_xor: Exclusive-or top two stack items* 46a ⟩ ≡

```
sub arg_xor {
    stackCheck(2);
    my $ol = hexToBytes(pop(@seeds));
    my $or = hexToBytes(pop(@seeds));
    if (bytes::length($ol) != bytes::length($or)) {
        print("-xor: arguments are different lengths.\n");
        exit(1);
    }
    my $rbytes;
    for (my $i = 0; $i < bytes::length($ol); $i++) {
        $rbytes .= chr(ord(bytes::substr($ol, $i, 1))) ^
                   chr(ord(bytes::substr($or, $i, 1)));
    }
    push(@seeds, bytesToHex($rbytes));
}
```

◇

Fragment referenced in 29.
Defines: `arg_xor` 27.
Uses: `bytesToHex` 57b, `hexToBytes` 57a, `stackCheck` 56b.

3.2.1.37 `arg_zero` — `-zero`: Push all zeroes on the stack

Push a value of all zero bits on the stack. This is shortcut for explicitly specifying such a value with `-seed`.

⟨ *arg_zero: Push all zeroes on the stack* 46b ⟩ ≡

```
sub arg_zero {
    ⟨ Begin command repeat 46c ⟩
        push(@seeds, "00" x 32);
    ⟨ End command repeat 46d ⟩
 }
```

◇

Fragment referenced in 29.
Defines: `arg_zero` 27.

3.2.1.38 Repeat command if `-repeat` specified

These macros are wrapped around sequences of code which should be executed the number of times specified by the `-repeat` command.

⟨ *Begin command repeat* 46c ⟩ ≡

```
for (my $rpt = 0; $rpt < $repeat; $rpt++) {
```

◇

Fragment referenced in 32a, 34a, 35b, 39b, 40a, 42bc, 43a, 46b.

⟨ *End command repeat* 46d ⟩ ≡

```
}
```

◇

Fragment referenced in 32a, 34a, 35b, 39b, 40a, 42bc, 43a, 46b.

3.2.1.39 Open output file

These macros enclose code whose output should be sent to the console or written to the file named by a previous `-outfile` command.

⟨ *Open output file* 47a ⟩ ≡

```
        if ($outputFile ne "-") {
            open(OFILE, ">$outputFile") || die("Cannot create file $outputFile");
            select OFILE;
        }
```

◇

Fragment referenced in 32a, 33a, 34a, 35b.

⟨ *Close output file* 47b ⟩ ≡

```
        if ($outputFile ne "-") {
            select STDOUT;
            close(OFILE);
        }
```

◇

Fragment referenced in 32a, 33a, 34a, 35b.

3.2.2 `genBtcAddress` — Generate address from one hexadecimal seed

A Bitcoin address and private key pair are generated from the argument, which specifies the 256 bit random seed as 64 hexadecimal digits. The private key and public address objects are returned in a list.

⟨ *genBtcAddress: Generate Bitcoin address from one hexadecimal seed* 47c ⟩ ≡

```
        sub genBtcAddress {
            my ($seed, $mode, $n) = @_;

            if ($seed !~ m/^[\dA-F]{64}/i) {
                die("Invalid seed.  Must be 64 hexadecimal digits");
            }
```

◇

Fragment defined by 47cd, 48ab.
Fragment referenced in 28.
Defines: `genBtcAddress` 32a, 35b.

Generate the private key from the hexadecimal seed.

⟨ *genBtcAddress: Generate Bitcoin address from one hexadecimal seed* 47d ⟩ ≡

```
            my $priv = Bitcoin::Crypto::Key::Private->from_hex($seed);
```

◇

Fragment defined by 47cd, 48ab.
Fragment referenced in 28.

Verify that we can decode the seed from the private key.

⟨ *genBtcAddress: Generate Bitcoin address from one hexadecimal seed* 48a ⟩ ≡

```
        my $dhex = uc($priv->to_hex());
        if ($dhex ne $seed) {
            die("Verify failed: Decoded " . $priv->to_hex() . "\n" .
                "               Encoded $seed");
        }
```

◇

Fragment defined by 47cd, 48ab.
Fragment referenced in 28.

Generate the public Bitcoin address from the private key. Note that if you're storing the private key, you needn't store the public address with it, since you can always re-generate it in any form you wish from the private key.

⟨ *genBtcAddress: Generate Bitcoin address from one hexadecimal seed* 48b ⟩ ≡

```
        my $pub = $priv->get_public_key();

        return ($priv, $pub);
    }
```

◇

Fragment defined by 47cd, 48ab.
Fragment referenced in 28.

3.2.3 `editBtcAddress` — Edit private key and public address

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 48c ⟩ ≡

```
    sub editBtcAddress {
        my ($priv, $pub, $mode, $n, $minikey) = @_;
```

◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.
Defines: `editBtcAddress` 32a, 35b.

Extract the seed from the private key in hexadecimal and encode it in base64.

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 48d ⟩ ≡

```
        my $phex = uc($priv->to_hex());
        my $pb64 = encode_base64($priv->to_bytes());
        chomp($pb64);
```

◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.

Generate compressed and uncompressed private keys, both encoded in WIF (Wallet Import Format). This is how private keys are usually stored in an off-line or paper wallet.

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 49a ⟩ ≡

```
            $priv->set_compressed(TRUE);
            my $WIFc = $priv->to_wif();

            $priv->set_compressed(FALSE);
            my $WIFu = $priv->to_wif();
```

◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.

Generate the public Bitcoin address from the private key. Note that if you're storing the private key, you needn't store the public address with it, since you can always re-generate it in any form you wish from the private key. We generate all forms of public addresses, compressed and uncompressed.

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 49b ⟩ ≡

```
            $pub->set_compressed(TRUE);
            my $pub_legacy = $pub->get_legacy_address();
            my $pub_compat = $pub->get_compat_address();
            my $pub_segwit = $pub->get_segwit_address();
            my $pub_hex = uc($pub->to_hex());

            $pub->set_compressed(FALSE);
            my $pub_legacy_u = $pub->get_legacy_address();
            my $pub_compat_u = $pub->get_compat_address();
            my $pub_segwit_u = $pub->get_segwit_address();
            my $pub_hex_u = uc($pub->to_hex());
```

◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.

Compose the output representation of the private key and public address. The format is specified by `$mode`, which can be "`CSV`*t*", where "*t*" is one or more of:

- `b` Exclude private key from CSV file
- `q` Use uncompressed private key
- `u` Use uncompressed public address
- `l` Legacy ("`1`") public address
- `c` Compatible ("`3`") public address
- `s` Segwit "`bc1`" public address

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 50 ⟩ ≡

```
            my $r = "";

            if ($mode =~ m/^CSV(\w*)$/) {
                my $CSVmodes = $1;

                #   Comma-separated value file

                my $privK = $WIFc;
                if ($CSVmodes =~ m/b/i) {       # b     Exclude private key
                    $privK = "";
                }
                if ($CSVmodes =~ m/q/i) {       # q     Uncompressed private key
                    $privK = $WIFu;
                }
                my $comp = $CSVmodes !~ m/u/i;  # u     Uncompressed public address
                my $pubK = $comp ? $pub_legacy : $pub_legacy_u;
                if ($CSVmodes =~ m/c/i) {       # c     Compatible ("3") public address
                    $pubK = $comp ? $pub_compat : $pub_compat_u;
                } elsif ($CSVmodes =~ m/s/i) {  # s     Segwit "bc1" public address
                    $pubK = $comp ? $pub_segwit : $pub_segwit_u;
                }

                my $mk = "";
                #   If generated from mini key, append it to CSV record
                if ($minikey && ($CSVmodes !~ m/b/i)) {
                    $mk = ",,\"$minikey\"";
                }
                $r = "$n,\"$pubK\",\"$privK\"$mk\n";

            } else {
```
◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.

If `$mode` is anything else, primate-readable output is generated. This includes all formats of the private key and public address, from which the user may choose whichever they prefer.

⟨ *editBtcAddress: Edit Bitcoin private key and public address* 51 ⟩ ≡

```
            #   Human-readable output

            #   Display private key seed in hexadecimal

            $r .= "Private key:\n";
            $r .= "    Hexadecimal:      $phex\n";
            $r .= "    Base64:           $pb64\n";
            $r .= BIP39encode("    BIP39:            ", $phex, 64);

            #   Display private key in both compressed and
            #   uncompressed  formats.

            $r .= "    WIF compressed:   $WIFc\n";
            $r .= "    WIF uncompressed: $WIFu\n";
            $r .= "    Minikey:          $minikey\n" if $minikey;

            #   Display public Bitcoin address in various formats

            $r .= "\nPublic Bitcoin address:\n" .
                  "  Compressed:\n" .
                  "    Legacy:  $pub_legacy\n" .
                  "    Compat:  $pub_compat\n" .
                  "    Segwit:  $pub_segwit\n" .
                  "    Hex:     $pub_hex\n" .
                  "  Uncompressed:\n" .
                  "    Legacy:  $pub_legacy_u\n" .
                  "    Compat:  $pub_compat_u\n" .
                  "    Segwit:  $pub_segwit_u\n" .
                  "    Hex:     $pub_hex_u\n";

            return $r;
        }
    }
```

◇

Fragment defined by 48cd, 49ab, 50, 51.
Fragment referenced in 28.
Uses: `BIP39encode` 56a.

3.2.4 `genEthAddress` — Generate Ethereum address from one hexadecimal seed

An Ethereum address and private key pair are generated from the argument, which specifies the 256 bit random seed as 64 hexadecimal digits. The private key and public address objects are returned in a list. Note that we use the `Bitcoin::Crypto::Key` package here to generate the public and private keys from the seed. This is not an error—Bitcoin and Ethereum use identical elliptic curve generator points and algorithms, so we can simply use the Bitcoin code as-is and then proceed to the different subsequent encoding employed by Ethereum.

⟨ *genEthAddress: Generate Ethereum address from one hexadecimal seed* 52a ⟩ ≡

```
        sub genEthAddress {
            my ($seed, $mode, $n) = @_;

            if ($seed !~ m/^[\dA-F]{64}/i) {
                die("Invalid seed.  Must be 64 hexadecimal digits");
            }
```

◇

Fragment defined by 52abcd.
Fragment referenced in 28.
Defines: `genEthAddress` 34a.

Generate the private key from the hexadecimal seed.

⟨ *genEthAddress: Generate Ethereum address from one hexadecimal seed* 52b ⟩ ≡

```
            my $priv = Bitcoin::Crypto::Key::Private->from_hex($seed);
```

◇

Fragment defined by 52abcd.
Fragment referenced in 28.

Verify that we can decode the seed from the private key.

⟨ *genEthAddress: Generate Ethereum address from one hexadecimal seed* 52c ⟩ ≡

```
            my $dhex = uc($priv->to_hex());
            if ($dhex ne $seed) {
                die("Verify failed: Decoded " . $priv->to_hex() . "\n" .
                    "                Encoded $seed");
            }
```

◇

Fragment defined by 52abcd.
Fragment referenced in 28.

Generate the public Ethereum address from the private key.

⟨ *genEthAddress: Generate Ethereum address from one hexadecimal seed* 52d ⟩ ≡

```
            my $pub = $priv->get_public_key();

            return ($priv, $pub);
        }
```

◇

Fragment defined by 52abcd.
Fragment referenced in 28.

3.2.5 editEthAddress — Edit Ethereum private key and public address

⟨ *editEthAddress: Edit Ethereum private key and public address* 53a ⟩ ≡

```
        sub editEthAddress {
            my ($priv, $pub, $mode, $n) = @_;
```

◇

Fragment defined by 53abc, 54ab.
Fragment referenced in 28.
Defines: `editEthAddress` 34a.

Extract the seed from the private key in hexadecimal and encode it in base64.

⟨ *editEthAddress: Edit Ethereum private key and public address* 53b ⟩ ≡

```
            my $phex = "0x" . $priv->to_hex();
```

◇

Fragment defined by 53abc, 54ab.
Fragment referenced in 28.

⟨ *editEthAddress: Edit Ethereum private key and public address* 53c ⟩ ≡

```
            $pub->set_compressed(FALSE);
            my $pub_hex_u = $pub->to_hex();
            my $pub_addr = "0x" .
                substr(keccak256_hex(hexToBytes(substr($pub_hex_u, 2))), -40);
            my $pub_addrc = computeEthChecksum($pub_addr);
```

◇

Fragment defined by 53abc, 54ab.
Fragment referenced in 28.
Uses: `computeEthChecksum` 55, `hexToBytes` 57a.

Compose the output representation of the private key and public address. The format is specified by `$mode`, which can be “`CSV`em t”, where “`t`” is one or more of:

- `b` Exclude private key from CSV file
- `n` No checksum on public address
- `p` Include full public key

⟨ *editEthAddress: Edit Ethereum private key and public address* 54a ⟩ ≡

```
            my $r = "";

            if ($mode =~ m/^CSV(\w*)$/) {
                my $CSVmodes = $1;

                my $dpub_addr = ($CSVmodes =~ m/n/i) ? $pub_addr : $pub_addrc;
                my $pkey = ($CSVmodes =~ m/p/i) ? ",\"$pub_hex_u\"" : "";
                if ($CSVmodes =~ m/b/i) {       # b     Exclude private key
                    $phex = "";
                }

                $r = "$n,\"$dpub_addr\",\"$phex\"$pkey\n";

            } else {
```

◇

Fragment defined by 53abc, 54ab.
Fragment referenced in 28.

If `$mode` is anything other than `CSV`, primate-readable output is generated. This includes all formats of the private key and public address, from which the user may choose whichever they prefer.

⟨ *editEthAddress: Edit Ethereum private key and public address* 54b ⟩ ≡

```
                #   Human-readable output

                #   Display private key seed in hexadecimal

                $r .= "Private key:\n";
                $r .= "  Hexadecimal: $phex\n";

                #   Display public Ethereum address

                $r .= "\nPublic Ethereum address:\n" .
                      "  Address:     $pub_addr\n" .
                      "  Checksum:    $pub_addrc\n" .
                      "  Public key:  $pub_hex_u\n";

                return $r;
            }
        }
```

◇

Fragment defined by 53abc, 54ab.
Fragment referenced in 28.

3.2.6 `computeEthChecksum`: Add checksum to Ethereum address

Ethereum addresses have an optional, most curious, checksum mechanism. Originally, Ethereum addresses were just hexadecimal addresses extracted from a hash of the public key as described above in `genEthAddress()` and `editEthAddress()`. A single character error in entering or transcribing such an address, as long as it remained a valid 40 digit hexadecimal number, would result in sending funds to “etherspace”—lost forever without any hope of recovery, since finding a private key which maps to the incorrect address is intractable.

Bitcoin addresses contain a checksum which catches, with a very high probability, such errors. To remedy the shortcoming in Ethereum addresses, in 2016 a proposed standard was published, "EIP-55: Mixed-case checksum address encoding", which prescribed the following upward-compatible scheme.

The computed hexadecimal address, with lower case letters for digits "`a`" through "`f`", is used to compute a Keccak256 digest (the same hash algorithm used in computing the public address) of the address (its hexadecimal text representation, not the binary value). Next, scan the 40 character public hexadecimal address, ignoring all digits from 0 to 9. For each letter, check the hexadecimal digit at the corresponding position in the hash (obviously, only the first 40 characters of the hash will be used). If the digit is between 8 and F, the letter in the address is converted from lower to upper case.

Clients unaware of checksums will ignore the case of the hexadecimal digits. Checksum-aware clients will, when presented with an address containing mixed case characters, recompute the checksum and, if it doesn't match, report the error. Note that an address which contains only digits from 0 to 9 or, when checksummed, happens to come out all capitals or all lower case, will evade the checksum test. Still, it's better than nothing.

⟨ *computeEthChecksum: Add checksum to Ethereum address* 55 ⟩ ≡

```
    sub computeEthChecksum {
        my ($eaddr) = @_;

        my $eal = lc($eaddr);
        #   Strip leading hex specification, if present
        $eal =~ s/^0x//;
        my $eahash = keccak256_hex($eal);
        for (my $i = 0; $i < length($eal); $i++) {
            my $ch = substr($eal, $i, 1);
            if ($ch =~ m/[a-f]/) {
                if (substr($eahash, $i, 1) =~ m/[89a-f]/) {
                    substr($eal, $i, 1) = uc($ch);
                }
            }
        }
        return "0x$eal";
    }
```

◇

Fragment referenced in 28.
Defines: `computeEthChecksum` 53c.

3.2.7 `BIP39encode` — Encode seed as BIP39 mnemonic phrase

This function encodes a 256 bit seed as a sequence of words according to Bitcoin Improvement Proposal 39 (BIP39), using the English word list from the reference implementation. The words are arranged in multiple lines of maximum length `$maxlen` using the specified `$prefix` on the first line. The Perl `Bitcoin::BIP39` package we use supports word lists for several other languages, but BIP39 states "it is **strongly discouraged** to use non-English wordlists for generating the mnemonic sentences" (emphasis in the original).

⟨ *BIP39encode: Encode seed as BIP39 mnemonic phrase* 56a ⟩ ≡

```
sub BIP39encode {
    my ($prefix, $seed, $maxlen) = @_;

    my $s = $prefix;
    my $cont = " " x length($prefix);
    my $r = "";
    my $bip39 = entropy_to_bip39_mnemonic( entropy_hex => $seed );
    my @b39 = split(/\s+/, $bip39);
    foreach my $word (@b39) {
        if ((length($s) + length($word)) > $maxlen) {
            $s =~ s/\s+$//;
            $r .= "$s\n";
            $s = $cont;
        }
        $s .= "$word ";
    }
    $s =~ s/\s+$//;
    $r .= "$s\n";
    return $r;
}
```

◇

Fragment referenced in 28.
Defines: `BIP39encode` 37a, 51.

3.2.8 `stackCheck` — Check for stack underflow

⟨ *stackCheck: Check for stack underflow* 56b ⟩ ≡

```
sub stackCheck {
    my ($required) = @_;

    if ($required > scalar(@seeds)) {
        print("Stack underflow: $required item(s) needed, only " .
            scalar(@seeds) . " present.\n");
        exit(1) if !$interactive;
        die("Stack underflow");
    }
}
```

◇

Fragment referenced in 28.
Defines: `stackCheck` 30ab, 32ac, 33b, 34a, 37ab, 38a, 39a, 40a, 41abc, 42bc, 43ab, 44ab, 46a.

3.2.9 `hexToBytes` — Convert hexadecimal string to binary

⟨ *hexToBytes: Convert hexadecimal string to binary* 57a ⟩ ≡

```
        sub hexToBytes {
            my ($hex) = @_;

            my $bytes;
            while ($hex =~ s/^([\dA-F]{2})//i) {
                $bytes .= chr(hex($1));
            }
            return $bytes;
        }
    ◇
```

Fragment referenced in 28.
Defines: `hexToBytes` 30ab, 31a, 37b, 40a, 42bc, 43a, 46a, 53c.

3.2.10 `bytesToHex` — Convert binary string to hexadecimal

⟨ *bytesToHex: Convert binary string to hexadecimal* 57b ⟩ ≡

```
        sub bytesToHex {
            my ($bytes) = @_;

            my $hex;
            for (my $i = 0; $i < bytes::length($bytes); $i++) {
                $hex .= sprintf("%02X", ord(bytes::substr($bytes, $i, 1)));
            }
            return $hex;
        }
    ◇
```

Fragment referenced in 28.
Defines: `bytesToHex` 30ab, 31b, 40b, 43a, 45a, 46a.

3.2.11 `findMiniKey` — Find a Bitcoin mini key

Search for a Bitcoin mini key that passes the checksum test, then return the mini key and corresponding full hexadecimal private key. Since there is no way to encode an arbitrary seed into a mini key, we simply search the name space with pseudorandomly generated strings until we find one whose hash is correct, then generate the seed from it.

⟨*findMiniKey: Find a Bitcoin mini key* 58⟩ ≡

```
        sub findMiniKey {
            my $b58r = "23456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz";
            my $b58rl = length($b58r);

            randInit();
            my $rgen;
            my $rbuf = "";
            if (!($testMode & 1)) {
                $rgen = Crypt::Random::Seed->new(NonBlocking => 1);
            }

            my $mk;
            do {
                $mk = "S";
                for my $i (1 .. 29) {
                    my $ri;
                    if ($rgen) {
                        do {
                            $ri = randNext(64);
                            if (defined($rgen)) {
                                if ($rbuf eq "") {
                                    $rbuf = $rgen->random_bytes(64);
                                }
                                $rbuf =~ s/^(.)//s;
                                my $rch = $1;
                                $ri ^= ord($rch) & 63;
                            }
                        } while ($ri >= $b58rl);
                    } else {
                        $ri = randNext($b58rl);
                    }
                    $mk .= substr($b58r, $ri, 1);
                }
            } while (sha256_hex("$mk?") !~ m/^00/);

            return ($mk, uc(sha256_hex($mk)));
        }
```

◇

Fragment referenced in 28.
Uses: `randInit` 142a, `randNext` 142b.

3.2.12 showHelp — Show help information

⟨ *showHelp: Show Bitcoin address help information* 59 ⟩ ≡

```
    sub showHelp {
        my $help = <<"    EOD";
perl blockchain_address.pl [ command... ]
  Commands and arguments:
    -aesenc             Encrypt second item on stack with top of stack key
    -aesdec             Decrypt second item on stack with top of stack key
    -bindump filename   Dump seeds from stack to binary file
    -binfile filename   Load seed(s) from binary file
    -btc                Generate Bitcoin public address/private key from stack seed
    -clear              Clear stack
    -drop               Drop top item on stack
    -dup                Duplicate top item on stack
    -eth                Generate Ethereum address/private key from stack seed
    -format f           Select CSV key output mode: CSVx, where x is
                            Bitcoin:
                              b   Suppress private key
                              c   Compatible public address ("3...")
                              k   Keep addresses on stack
                              l   Legacy public address ("1...")
                              q   Uncompressed private key
                              s   Segwit public address ("bc1...")
                              u   Uncompressed public address
                            Ethereum:
                              b   Suppress private key
                              k   Keep addresses on stack
                              n   No checksum on public address
                              p   Include full public key
    -hbapik hbapikey    Specify HotBits API key
    -help               Print this message
    -hexfile filename   Load hexadecimal seed(s) from filename
    -hotbits            Get seed(s) from HotBits, place on stack
```

◇

Fragment defined by 59, 60.
Fragment referenced in 28.

⟨ *showHelp: Show Bitcoin address help information* 60 ⟩ ≡

```
    -inter              Process interactive commands
    -minigen            Generate Bitcoin mini private key
    -minikey key        Decode Bitcoin mini private key
    -mnemonic           Generate BIP39 mnemonic phrase from stack top
    -not                Invert stack top
    -outfile filename   Redirect key generation output to file or - for console
    -over               Duplicate second item on stack to top
    -p                  Print top item on stack
    -phrase words...    Specify seed as BIP39 menemonic phrase
    -pick n             Duplicate the nth item on the stack to top
    -pseudo             Generate pseudorandom seed and push on stack
    -pseudoseed         Set seed for pseudorandom generator
    -random             Obtain a seed from system strong generator, push on stack
    -repeat n           Repeat following commands n times
    -roll n             Rotate item n to top of stack
    -rot                Rotate the top three stack items
    -rrot               Reverse rotate top three stack items
    -seed hex           Push the hexadecimal seed on top of stack
    -sha2               Replace top of stack with its SHA2-256 digest
    -sha3               Replace top of stack with its SHA3-256 digest
    -shuffle            Shuffle bytes on stack
    -swap               Swap the two top items on the stack
    -test               Test stack items for randomness
    -testall            Test entire stack contents for randomness
    -testmode n         Select test modes (0 for production)
    -type Any text      Display text argument on standard output
    -urandom            Obtain a seed from system fast generator, push on stack
    -wif                Push seed extracted from Wallet Input Format private key
    -xor                Bitwise exclusive-or top two stack items
    -zero               Push zeroes on stack
EOD
        $help =~ s/^    //gm;
        print($help);
        if (!$interactive) {
            exit(0);
        }
    }
```

◇

Fragment defined by 59, 60.
Fragment referenced in 28.

Chapter 4

Multiple Key Manager

The `multi_key` program implements Shamir Secret Sharing for blockchain private keys, allowing them to be distributed among multiple custodians or storage locations, then reconstructed from a minimum specified number of parts. Each secret key is split into n parts ($n \geq 2$), of which any $k, 2 \leq k \leq n$ are sufficient to reconstruct the entire original key, but from which the key cannot be computed from fewer than k parts. In the implementation below, we refer to n as the number of `parts` and k as the number `needed`. The heavy lifting is done by the Perl library module `Crypt::SSSS`.

The blockchain addresses and private keys are specified in a Comma-Separated Value (CSV) file in the format produced by `blockchain_address` and used by other utilities in the collection. Both Bitcoin and Ethereum addresses may be used.

4.1 Program plumbing

"`perl/multi_key.pl`" 61≡

```
    ⟨ Explanatory header for Perl files 145a ⟩

        ⟨ Perl language modes 144a ⟩

        use Crypt::SSSS;
        use Digest::SHA qw(sha256 sha256_hex);
        use List::Util qw(shuffle);
        use Text::CSV qw(csv);
        use POSIX qw(log10);
        use Getopt::Long;
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.2 Settings and option processing

"perl/multi_key.pl" 62≡

```
    my $basename = "";                  # Base name for generated files
    my $join = FALSE;                   # Join parts into complete keys
    my $prime = 257;                    # Prime used to set security
    my $parts = 3;                      # Number of shared keys to generate
    my $needed = 3;                     # Shared keys to reassemble address

    GetOptions(
        "help"      =>  \&showHelp,
        "join"      =>  \$join,
        "name=s"    =>  \$basename,
        "needed=i"  =>  \$needed,
        "parts=i"   =>  \$parts,
        "prime=i"   =>  \$prime
    ) || die("Command line option error");

    my $csv = Text::CSV->new({ binary => 1 }) ||
        die("Cannot use CSV: " . Text::CSV->error_diag());

    if ($basename eq "") {
        if ((scalar(@ARGV) > 0) && ($ARGV[0] ne "")) {
            $basename = $ARGV[0];
            $basename =~ s/\.\w*$//;
        }
        if ($basename eq "") {
            $basename = $join? "joined_keys-1" : "shared_keys";
        }
    }
```
◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.3 First pass

On the first pass, read the records from the input file(s) and save them in the `@records` array. We use the `Text::CSV` parser for the standard first three fields (label, address, and private key), then save any extra material which follows them to be re-appended when the output file is written. This allows preserving extra information, such as balances, when keys are split and reassembled.

"perl/multi_key.pl" 63a≡

```
        my @records;
        my $naddrs = 0;
        while (my $l = <>) {
            chomp($l);
            $l =~ s/^\s+//;
            $l =~ s/\s+$//;
            if (($l ne "") && ($l !~ m/^#/)) {
                my $extra;
                if (($l !~ m/\s*\-1,/) && ($l =~ s/^([^,]+,[^,]+,[^,]+)(,.*)$/$1/)) {
                    $extra = $2;
                }
                if ($csv->parse($l)) {
                    $naddrs++;
                    my @fields = $csv->fields;
                    if ($extra) {
                        $fields[3] = $extra;
                    }
                    push(@records, \@fields);
               }
            }
        }
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

After loading the records, if this is a join operation, invoke the `joinParts()` function to perform it.

"perl/multi_key.pl" 63b≡

```
        if ($join) {
            exit(joinParts());
        }
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: `joinParts` 67b.

4.4 Split private keys into parts

Each private key in the input file will be encoded into the specified number of `parts`, and written to separate CSV output files which, if no `-name` is specified, bear the base name of the first input file with a hyphen and part number appended.

4.4.1 Create part output files

Start by creating files for each of the split key parts. These are CSV files like those used for addresses and keys, except the key field is replaced by the encoded part for that key. The files have headers with fields:

`-1,`*parts*`,`*needed*`,`*prime*`,`*partno*

where *parts* is the number of parts, of which *needed* are required to reconstruct the key, *prime* is the prime number used in the encoding, and *partno* is the number of this part, from 1 to *parts*.

"perl/multi_key.pl" 64a≡

```
        my $fnd = int(log10($parts)) + 1;
        my @files;
        for (my $f = 1; $f <= $parts; $f++) {
            my $fnx = sprintf("%s-%0${fnd}d.csv", $basename, $f);
            open($files[$f], ">$fnx") || die("Cannot create $fnx");
            $files[$f]->printf("-1,$parts,$needed,$prime,$f\n");
        }
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.4.2 Process key records

Process records, replacing the original private key with the shared key part in each file.

"perl/multi_key.pl" 64b≡

```
        my $fail = 0;

        for (my $r = 0; $r < scalar(@records); $r++) {
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.4.2.1 Encode and checksum the private key

To permit validation of the private key after it is reconstructed from parts, encode it by prefixing it with its length, then compute and append the double SHA256 checksum to the end. It is this encoded key which is actually split into parts.

"perl/multi_key.pl" 64c≡

```
            my $privkey = chr(32 + length($records[$r]->[2])) . $records[$r]->[2];
            $privkey .= compCheck($privkey);

            my $shares = ssss_distribute(
                message =>  $privkey,
                k       =>  $needed,
                n       =>  $parts,
                p       =>  $prime
            );
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: compCheck 73a.

4.4.2.2 Write the split parts to part files

Assemble the part item, prefixing it with the type sentinel, part number, and delimiter, and computing and appending the checksum of these. We save a copy of the parts in @hexpart, which we'll use to confirm the ability to reconstruct the key from the parts in the safety check below. A record is added to the part file, consisting of the fields from the original key record but with the private key replaced by the encoded part. The part items are saved in the @hexpart array for our subsequent reconstruction quality check.

"perl/multi_key.pl" 65≡

```
            my @hexpart;
            for (my $f = 1; $f <= $parts; $f++) {
                my $hexcheck = sprintf("S%0${fnd}d-%s", $f,
                    unpack("H*", $shares->{$f}->binary));
                $hexcheck .= compCheck($hexcheck);
                push(@hexpart, $hexcheck);
                my $extra = $records[$r]->[3] ? $records[$r]->[3] : "";
                $files[$f]->printf("%s,\"%s\",\"%s\"%s\n", $records[$r]->[0],
                    $records[$r]->[1], $hexcheck, $extra);
            }
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: `compCheck` 73a.

4.4.2.3 Reconstruction quality check

Now that we've generated the parts for this address and written them to the parts files, using copies of the parts squirreled away in the `@hexpart` array, re-assemble them in random order as many different ways as there are parts. This verifies that when the time comes we'll actually be able to reconstruct the original keys from the parts files.

"perl/multi_key.pl" 66≡

```
            for (my $l = 0; $l < $parts; $l++) {

                #   Shuffle order of parts before reconstruction
                @hexpart = shuffle(@hexpart);

                #   Perform reconstruction of key from groups of shuffled parts

                for (my $p = 0; $p <= ($parts - $needed); $p++) {
                    my $rkey = { };
                    for (my $q = $p; $q < ($p + $needed); $q++) {
                        my ($pstat, $pno, $hxp) =  parsePart($hexpart[$q]);
                        if ($pstat < 0) {
                            die("Cannot parse hex part $q " .
                                "$hexpart[$q]: ($pstat, $pno, $hxp)\n");
                        }
                        #   Unpack the hex part payload to bytes and save in parts hash
                        $rkey->{$pno} = pack('C*',  map({ hex($_) } ($hxp =~ /(..)/g)));
                    }
                    my $rpk = ssss_reconstruct(p => $prime, shares => $rkey);
                    my ($kstat, $privad) = parseKey($rpk);
                    if (!$kstat) {
                        die("Bad checksum in reconstructed record: $rpk\n  $privad");
                    }
                    if ($records[$r]->[2] ne $privad) {
                        $fail++;
                        printf(STDERR "** Reconstruction failure on key %d, " .
                            "parts %d through %d:\n   Exp: (%s)\n   Rcv: (%s)\n",
                            $r, ($p + 1), ($p + $needed), $records[$r]->[2], $privad);
                    }
                }
            }
        }
```
◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: parseKey 72b, parsePart 72a.

4.4.3 Close part files

When all keys have been split and written to the part files, we're done. Close the part files and exit. If an error occurred in the split or test reconstruction process, delete the part files to avoid a tragedy which might occur were they kept and later used to try to reconstruct keys. The exit status indicates whether the parts were successfully generated (0) or an error occurred (1).

"perl/multi_key.pl" 67a≡

```
    for (my $f = 1; $f <= $parts; $f++) {
        close($files[$f]);
    }

    #   If errors were detected, delete part files to avoid tragedy
    if ($fail > 0) {
        print(STDERR "Failures to reconstruct keys from parts: $fail.\n" .
                     "  Deleting part files.\n");
        for (my $f = 1; $f <= $parts; $f++) {
            unlink(sprintf("%s-%0${fnd}d.csv", $basename, $f));
        }
    }

    exit($fail > 0);
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.5 Join parts into complete keys

When multi_key is invoked with the -join option, the input is expected to be the concatenation of the *needed* number of split key files produced by an earlier run. These may be specified as multiple file names on the command line, and may be in any order. If more split parts are supplied than needed, a warning is issued and the extras ignored.

"perl/multi_key.pl" 67b≡

```
    sub joinParts {
        my $warn = 0;
        my $error = 0;

        my ($restParts, $restNeeded, $restPrime, $restPart);
        my %partsSeen;
        my @addresses;
        my %parts;
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Defines: joinParts 63b.

4.5.1 Read split parts, validate, and save

Process all of the records read from the input file, representing the parts to be reconstructed. Each is checked and saved as an object in a hash with a key of the public address to which it corresponds.

"perl/multi_key.pl" 67c≡

```
        for (my $r = 0; $r < scalar(@records); $r++) {
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.5.1.1 Process part definition record

If this is a part definition record, identified by a value of −1 in the *label* field, validate it and save for subsequent processing.

"perl/multi_key.pl" 68≡

```
                #   Test for part definition record and process
                if ($records[$r]->[0] eq "-1") {
                    #   Check for inconsistency among parts and save
                    #   the part generation parameters.
                    if ($restParts && ($restParts != $records[$r]->[1])) {
                        printf("Warning: Record definition for part %d " .
                               "part count %d inconsistent " .
                               "with previous parts (%d).\n",
                               $records[$r]->[4], $records[$r]->[1], $restParts);
                        $warn++;
                    } else {
                        $restParts = $records[$r]->[1];
                    }

                    if ($restNeeded && ($restNeeded != $records[$r]->[2])) {
                        printf("Warning: Record definition for part %d " .
                               "parts needed %d inconsistent " .
                               "with previous parts (%d).\n",
                               $records[$r]->[4], $records[$r]->[2], $restNeeded);
                        $warn++;
                    } else {
                        $restNeeded = $records[$r]->[2];
                    }

                    if ($restPrime && ($restPrime != $records[$r]->[3])) {
                        printf("Warning: Record definition for part %d " .
                               "parts needed %d inconsistent " .
                               "with previous parts (%d).\n",
                               $records[$r]->[4], $records[$r]->[3], $restPrime);
                         $warn++;
                   } else {
                        $restPrime = $records[$r]->[3];
                    }

                    #   Warn if this is a duplicate specification of this part
                    $restPart = $records[$r]->[4];
                    if ($partsSeen{$restPart}) {
                        printf("Warning: Duplicate specification for part %d.\n", $restPart);
                        $warn++;
                    } else {
                        $partsSeen{$restPart} = TRUE;
                    }
                } else {
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.5.1.2 Process private key part record

Otherwise, this is a record specifying a part of the private key for an address. Parse it, validate the checksum in the part item, verify the part number corresponds to that expected from the previous header record, and save in a hash keyed by the public address pointing to an array indexed by part number.

"perl/multi_key.pl" 69≡

```
                my ($label, $pubaddr, $partkey, $extra) = ($records[$r]->[0],
                    $records[$r]->[1], $records[$r]->[2], $records[$r]->[3]);
                my ($pstat, $pno, $pvalue) = parsePart($partkey);

                if (!defined($extra)) {
                    $extra = "";
                }

                if ($pstat < 0) {
                    if ($pstat == -1) {
                        printf("Error: cannot parse part %d key: %s\n", $restPart,
                            $partkey);
                    } else {
                        printf("Error: bad checksum in part %d key: %s\n", $restPart,
                            $partkey);
                    }
                    $error++;
                } else {
                    if ($pno != $restPart) {
                        printf("Warning: part number (%d) for address %s " .
                            "differs from part number (%d) in header record.\n",
                            $pno, $pubaddr, $restPart);
                            $warn++;
                    }
                    my $ap = {
                        label   =>  $label,
                        partkey =>  $pvalue,
                        extra   =>  $extra
                    };
                    if (!$parts{$pubaddr}) {
                        $parts{$pubaddr} = [ ];
                        push(@addresses, $pubaddr);
                    }
                    $parts{$pubaddr}->[$restPart] = $ap;
                }
            }
        }
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: `parsePart` 72a.

4.5.2 Part validity and completeness checks

Now that all of the parts specifications have been loaded, check that the *needed* number of parts have been supplied (error if too few, warning if too many, with any extras ignored), and that all parts have been specified for all public addresses.

"perl/multi_key.pl" 70a≡

```
#   Verify correct number of parts specified
my $nps = scalar(keys(%partsSeen));
if ($nps < $restNeeded) {
    printf("Error: fewer parts specified (%s) than needed (%s).\n",
        $nps, $restNeeded);
    $error++;
} elsif ($nps > $restNeeded) {
    printf("Warning: more parts specified (%s) than needed (%s).\n",
        $nps, $restNeeded);
    $warn++;
}

#   Verify that all parts are specified for all addresses
foreach my $a (@addresses) {
    foreach my $pt (keys(%partsSeen)) {
        if (!($parts{$a}->[$pt])) {
            print("Error: part $pt missing for address $a.\n");
            $error++;
        }
    }
}
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.5.3 Create output key file

The output file containing the addresses and reassembled private keys is given a name constructed from the base name of the first part (after deleting its part number suffix) and appending "`-merged.csv`". The file is created and a comment written to it identifying the parts from which it was assembled.

"perl/multi_key.pl" 70b≡

```
$basename =~ s/\-\d+$//;
$basename .= "-merged.csv";
open(FO, ">$basename") ||
    die("Cannot create $basename");
my $title = "# Private keys assembled from parts ";
foreach my $pn (sort { $a <=> $b } (keys(%partsSeen))) {
    $title .= "$pn, ";
}
$title =~ s/, $/\n/;
print(FO $title);
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

4.5.4 Reconstruct, validate, and output private keys

We're finally ready to assemble the pieces into the private keys. We iterate using the `@addresses` array to preserve the order of the keys in the first part file. As each private key is reconstructed, its internal checksum, appended when the original key was split, is verified and any errors reported. A record is appended to the output file with the reassembled private key.

"perl/multi_key.pl" 71≡

```
        foreach my $a (@addresses) {
            my $rkey = { };
            my $lbl;
            my $rpts = 0;
            foreach my $pt (keys(%partsSeen)) {
                #   Unpack the hex part payload to bytes and save in parts hash
                if (defined($parts{$a}->[$pt])) {
                    my $hxp = $parts{$a}->[$pt]->{partkey};
                    $lbl = $parts{$a}->[$pt]->{label};
                    $rkey->{$pt} = pack('C*',  map({ hex($_) } ($hxp =~ /(..)/g)));
                    $rpts++;
                    #   If more parts were specified than needed, stop
                    #   after we've processed the number required.
                    if ($rpts >= $restNeeded) {
                        last;
                    }
                }
            }
            my $rpk = ssss_reconstruct(p => $prime, shares => $rkey);
            my ($kstat, $privad) = parseKey($rpk);
            if (!$kstat) {
                print("Bad checksum inreconstructed key for $a: $rpk\n  $privad");
                $error++;
            }
            #   We arbitrarily use the extra information from the last
            #   part (all parts should have identical extra information).
            my $ext = $parts{$a}->[-1]->{extra};
            printf(FO "%s,\"%s\",\"%s\"%s\n", $lbl, $a, $privad, $ext);
        }
        close(FO);

        return ($error > 0) ? 2 : (($warn > 0) ? 1 : 0);
    }
```

◇

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Uses: parseKey 72b.

4.6 Local functions

4.6.1 parsePart — Parse and validate part record

Parse a part record into components and validate. Returns (*status*, *partNumber*, *partValue*).

"perl/multi_key.pl" 72a≡

```
        sub parsePart {
            my ($part) = @_;

            $part =~ m/^S(\d+)\-(\w+?)(\w{8})$/ || return (-1, "", "");
            my ($partNumber, $partValue, $checksum) = ($1, $2, $3);
            $partNumber =~ s/^0//g;
            my $rcheck = compCheck(substr($part, 0, -8));
            if ($rcheck ne $checksum) {
                return (-2, $checksum, $rcheck);
            }
            return (TRUE, $partNumber, $partValue);
        }
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Defines: parsePart 66, 69.
Uses: compCheck 73a.

4.6.2 parseKey — Parse encoded key record

Parse an encoded key record, extracting the length, key, and checksum, then validate the checksum. This used to validate key records after they are reassembled from parts. A list is returned with a status indicating validity of the checksum and the extracted private key.

"perl/multi_key.pl" 72b≡

```
        sub parseKey {
            my ($rpk) = @_;

            my $rlen = ord(substr($rpk, 0, 1)) - 32;
            my $privad = substr($rpk, 1, $rlen);
            my $cksum = substr($rpk, $rlen + 1, 8);

            my $kcheck = compCheck(substr($rpk, 0, $rlen + 1));
            if ($kcheck ne $cksum) {
                return (FALSE, "$cksum != $kcheck");
            }
            return (TRUE, $privad);
        }
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Defines: parseKey 66, 71.
Uses: compCheck 73a.

4.6.3 compCheck — Compute checksum on string

The checksum is a Bitcoin-address-like double SHA256 hash expressed in hexadecimal and trimmed to just the first 8 hexadecimal digits (4 bytes).

"perl/multi_key.pl" 73a≡

```
        sub compCheck {
            my ($s) = @_;

            return substr(sha256_hex(sha256($s)), 0, 8);
        }
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.
Defines: compCheck 64c, 65, 72ab.

4.6.4 showHelp — Show help information

"perl/multi_key.pl" 73b≡

```
        sub showHelp {
            my $help = <<"EOD";
    perl multi_key.pl [ option... ] file...
      Commands and arguments:
        -help               Print this message
        -join               Join parts and reconstruct keys
        -name filename      Specify name of part or joined key files
        -needed k           Set k parts required to reconstruct keys
        -parts n            Split keys into n parts, of which k are needed
        -prime p            Use p as prime number to encode (super-experts only!)
    EOD
            print($help);
            exit(0);
        }
    ◇
```

File defined by 61, 62, 63ab, 64abc, 65, 66, 67abc, 68, 69, 70ab, 71, 72ab, 73ab.

Chapter 5

Paper Wallet Utilities

These utilities, `paper_wallet` and `validate_wallet`, create and validate cold storage paper wallets, starting with Bitcoin or Ethereum addresses in the CSV format generated by `blockchain_address` or parts of multiple part keys created by `multi_key`. Paper wallets are created by expressing them as an HTML file, which may be loaded into a browser and printed.

5.1 Paper Wallet Generator

Read a list of addresses and private keys generated by `blockchain_address` and output HTML which prints them in a format suitable for offline cold storage. You'll usually print multiple copies and store them in redundant secure locations. This program can also produce printable documents from parts of multiple key wallets generated by `multi_key`.

`"perl/paper_wallet.pl"` 74≡

```
⟨ Explanatory header for Perl files 145a ⟩

    ⟨ Perl language modes 144a ⟩

    use Text::CSV qw(csv);
    use POSIX qw(strftime);
    use Getopt::Long;
#use Data::Dumper;
◇
```

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.1 Modes and option processing

"perl/paper_wallet.pl" 75≡

```
my $date = "";                  #   Date override for address page
my $fontname = "monospace";     #   Font name for addresses
my $fontsize = "medium";        #   Font size for addresses
my $fontweight = "normal";      #   Font weight for addresses
my $offset = 0;                 #   Add to address numbers in the input file
my $perpage = 10;               #   Print this number of addresses per page
my $prefix = "";                #   Prefix the address numbers with this string
my $separator = "";             #   Separator for address groups
my $title = "";                 #   Title for page

GetOptions(
    "date=s"        =>  \$date,
    "font=s"        =>  \$fontname,
    "help"          =>  \&showHelp,
    "offset=i"      =>  \$offset,
    "perpage=i"     =>  \$perpage,
    "prefix=s"      =>  \$prefix,
    "separator=s"   =>  \$separator,
    "size=s"        =>  \$fontsize,
    "title=s"       =>  \$title,
    "weight=s"      =>  \$fontweight
) || die("Command line option error");

my $csv = Text::CSV->new({ binary => 1 }) ||
    die("Cannot use CSV: " . Text::CSV->error_diag());

#   If no date specified, use current UTC date
if ($date eq "") {
    $date = strftime("%F", gmtime(time()));
}
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.2 First pass: read address records

In the first pass, read the records, determine which kind of blockchain they represent, and save them in an array for processing on the second pass. This allows us to know how many pages we're going to generate if the output is paginated.

"perl/paper_wallet.pl" 76≡

```
my $started = FALSE;
my $inpage = 0;
my $multipart = FALSE;
my ($partsn, $partsk, $partsthis);

my @records;
my ($naddrs, $npages) = (0, 0);
while (my $l = <>) {
    chomp($l);
    $l =~ s/^\s+//;
    $l =~ s/\s+$//;
    if (($l ne "") && ($l !~ m/^#/)) {
        if ($csv->parse($l)) {
            $naddrs++;
            my @fields = $csv->fields;
            if (!$started) {
                if ($fields[0] eq "-1") {
                    $multipart = TRUE;
                    ($partsn, $partsk, $partsthis) = ($fields[1],
                        $fields[2], $fields[4]);
                    $naddrs--;
                    next;
                }
                $started = TRUE;
                if ($title eq "") {
                    $title = (($fields[1] =~ m/^0x/g) ?
                        "Ethereum" : "Bitcoin") . " Wallet";
                }
                $npages++;
            } elsif (($perpage > 0) && ($inpage >= $perpage)) {
                $inpage = 0;
                $npages++;
            }
            if (($offset != 0) && ($fields[0] =~ m/^\-?\d+$/)) {
                $fields[0] += $offset;
            }
            $fields[0] = $prefix . $fields[0];
            push(@records, \@fields);
            $inpage++;
        }
    }
}
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.3 Second pass: generate HTML output

In the second pass we process the records from the first pass and generate the HTML output.

"perl/paper_wallet.pl" 77a≡

```
$inpage = 0;
$started = FALSE;
my $pageno = 1;
my $pagetop = TRUE;
for (my $i = 0; $i < scalar(@records); $i++) {
    if (!$started) {
        HTMLstart($date, $title);
        pageHeader($pageno);
        $started = TRUE;
    } elsif (($perpage > 0) && ($inpage >= $perpage)) {
        pageFooter($pageno, $npages);
        $inpage = 0;
        $pagetop = TRUE;
        $pageno++;
        pageHeader($pageno);
    }
    HTMLrec($pagetop, $records[$i]->[0], $records[$i]->[1], $records[$i]->[2]);
    $pagetop = FALSE;
    $inpage++;
}

pageFooter($pageno, $npages);
HTMLend();
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4 Utility functions

5.1.4.1 addrFormat — Format address with separators

Format a public address or private key. If a nonblank separator has been set, insert them between groups of four characters in the string to make it more primate-readable.

"perl/paper_wallet.pl" 77b≡

```
sub addrFormat {
    my ($addr) = @_;

    if ($separator) {
        my $a = "";
        while ($addr =~ s/^(.+?)(....)$//) {
            $a = "<span class=\"s\"></span>$2$a";
            $addr = $1;
        }
        $addr = "$addr$a";
    }
    return $addr;
}
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.2 pageHeader — Generate page header

Generate the HTML for the page heading.

"perl/paper_wallet.pl" 78a≡

```
    sub pageHeader {
        my ($pageno) = @_;

        my ($multihead, $multistyle) = ("", "");
        if ($multipart) {
            $multihead = "\n        <h4>Part $partsthis of $partsn ($partsk needed)";
            $multistyle = " class=\"multi\"";
        }
        my $headerdiv = ($pageno == 1) ? "firstpage" : "subsequentpage";
        print <<"EOD";
    <div class="$headerdiv">
        <h1>$title</h1>
        <h3$multistyle>$date</h3>$multihead
    </div>
EOD
    }
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.3 pageFooter — Generate page footer

Generate the HTML for the page footer.

"perl/paper_wallet.pl" 78b≡

```
    sub pageFooter {
        my ($pageno, $ofpages) = @_;

        print <<"EOD";

    <div class="pagefooter">
        <p>Page $pageno of $ofpages</p>
    </div>
EOD
    }
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.4 HTMLstart — Generate HTML file prologue

Generate the prologue at the start of the HTML file. With the exception of the title and date, this is entirely canned and identical for every file we generate. The bulk of the header is the Cascading Style Sheet (CSS), which we define later and include here.

"perl/paper_wallet.pl" 79≡

```
    sub HTMLstart {
        my ($date, $title) = @_;
        print <<"EOD";
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>$title</title>
<style>
⟨ Style sheet for paper wallets 81b, ... ⟩
</style>
</head>

<body class="standard">

EOD
    }
◇
```

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.5 `HTMLrec` — Output one address record to HTML file

Emit the record for one blockchain address / private key pair. Note that our `validate_wallet` program is sensitively dependent upon the format of these records; if you change them indiscriminately, you're likely to break that program.

"perl/paper_wallet.pl" 80a≡

```
    sub HTMLrec {
        my ($pagetop, $n, $pub, $priv) = @_;

        $pub = addrFormat($pub);
        $priv = addrFormat($priv);
        print <<"EOD";

    <table class="addr">
        <tr class="pub">
            <th class="num" rowspan="2">
                $n
            </th>
            <td class="type">Pub:</td>
            <td class="pub">
                $pub
            </td>
        </tr>
        <tr class="priv">
            <td class="type">Priv:</td>
            <td class="priv">
                $priv
            </td>
        </tr>
    </table>
EOD
    }
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.6 `HTMLend` — Generate HTML file epilogue

Emit the end of the HTML file.

"perl/paper_wallet.pl" 80b≡

```
    sub HTMLend {
        print <<"EOD";

</body>
</html>
EOD
    }
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.4.7 showHelp — Show help information

"perl/paper_wallet.pl" 81a≡

```
        sub showHelp {
            my $help = <<"EOD";
    perl paper_wallet.pl [ option... ] file...
      Commands and arguments:
        -date text          Use text as generation date
        -font fname         Display addresses and keys in CSS font fname
        -help               Print this message
        -offset n           Add n to the address numbers in the input file
        -perpage n          Print n addresses per page
        -prefix text        Prefix address labels with text
        -separator text     Show addresses as 4 character groups with separator text
        -size fsize         Display addresses with CSS font size fsize
        -title text         Use text as page title
        -weight wgt         Show addresses with CSS font weight wgt
    EOD
            print($help);
            exit(0);
        }
```

◇

File defined by 74, 75, 76, 77ab, 78ab, 79, 80ab, 81a.

5.1.5 Style sheet

This CSS style sheet is embedded in the HTML file we generate. It is embedded in the interest of its being entirely self-contained and thus more easily transferred from, for example, an air-gapped computer on which it is generated and another machine with a direct connection to a printer.

5.1.5.1 Page-level formatting

⟨ *Style sheet for paper wallets* 81b ⟩ ≡

```
        body.standard {
            background-color: #FFFFFF;
            color: #000000;
        }

        div.pagefooter {
            text-align: center;
        }

        div.subsequentpage {
            page-break-before: always;
        }
```

◇

Fragment defined by 81b, 82, 83.
Fragment referenced in 79.

5.1.5.2 Table of addresses and keys

⟨ *Style sheet for paper wallets* 82 ⟩ ≡

```
 table.addr {
     border-collapse: collapse;
     margin-bottom: 1.5ex;
     margin-left: auto;
     margin-right: auto;
 }

 th.num {
     border-bottom: 1px solid black;
     border-right: 1px solid black;
     font-size: 20pt;
     font-weight: bold;
     padding-left: 0.25em;
     padding-right: 0.5em;
     text-align: right;
     width: 12mm;
}

 tr.priv {
     border-bottom: 1px solid black;
     border-left: 1px solid black;
     border-right: 1px solid black;
 }

 tr.pub {
     border-left: 1px solid black;
     border-right: 1px solid black;
     border-top: 1px solid black;
 }

 td.type {
     font-weight: bold;
     padding-left: 6px;
 }

 td.priv, td.pub {
     font-family: "$fontname";
     font-size: $fontsize;
     font-weight: $fontweight;
     padding-left: 6px;
     padding-right: 6px;
     width: 180mm;
 }
```

◇

Fragment defined by 81b, 82, 83.
Fragment referenced in 79.

5.1.5.3 Formatting of items

⟨ *Style sheet for paper wallets* 83 ⟩ ≡

```
    h1 {
        margin-bottom: 0px;
        text-align: center;
    }

    h3 {
        margin-top: 0px;
        text-align: center;
    }

    h3.multi {
        margin-bottom: 0px;
    }

    h4 {
        margin-top: 0px;
        text-align: center;
    }

    span.s:after {
        font-family: sans-serif;
        content: "";
    }
◇
```

Fragment defined by 81b, 82, 83.
Fragment referenced in 79.

5.2 Cold Storage Wallet Validator

This program, completely independent of the Perl blockchain utilities that generate them, verifies that the private keys in a cold storage wallet file correspond to the public addresses generated from them. It avoids the tragedy when, for whatever cause, funds are sent to a public address for which the corresponding private key is not known. It can validate either a CSV wallet generated by `blockchain_address`, or a printable HTML file created from it with `paper_wallet`. This program cannot directly validate multi-part keys created by `multi_key`, it may be used to validate their reconstruction after the parts are joined.

To avoid any commonality with the wallet generation code, it is written in a different programming language, Python, and uses that language's libraries. This program requires Python version 3 or above.

5.2.1 Bitcoin key and address functions

5.2.1.1 Bitcoin library modules

The following Python library modules are used to manipulate Bitcoin public addresses amd private keys.

⟨ *Bitcoin library modules* 84a ⟩ ≡

```
import base58
import binascii
from cryptos import Bitcoin
◇
```

Fragment referenced in 90.

5.2.1.2 Extract private seed from WIF private address

Private keys in cold storage wallets are stored in Wallet Input Format, which consists of the 256-bit seed and a checksum, encoded in Base 58 format. This function decodes the private key, extracts the binary key, and returns it as a hexadecimal string.

⟨ *Extract private seed from WIF private address* 84b ⟩ ≡

```
def getPrivateKey(WIF):
    first_encode = base58.b58decode(WIF)
    private_key_full = binascii.hexlify(first_encode)
    private_key = private_key_full[2:-8]
    return private_key
◇
```

Fragment referenced in 90.

5.2.1.3 Generate public address from WIF private key

Convert a Bitcoin private key in Wallet Import Format (WIF) to the corresponding public address, in legacy “1” compressed format.

⟨ *Generate public address from WIF private key* 84c ⟩ ≡

```
def WIF_to_Bitcoin_address(WIF):
    c = Bitcoin(testnet=False)
    pk = getPrivateKey(WIF)
    BTCaddr = c.privtoaddr(pk)
    return BTCaddr
◇
```

Fragment referenced in 90.

5.2.2 Ethereum key and address functions

5.2.2.1 Ethereum library modules

The following Python library modules are used to manipulate Ethereum public addresses and private keys.

⟨ *Ethereum library modules* 84d ⟩ ≡

```
from coincurve import PublicKey
from sha3 import keccak_256
◇
```

Fragment referenced in 90.

5.2.2.2 Add “checksum” to Ethereum public address

Add the “checksum” to a public Ethereum address by computing its hash and setting hexadecimal letter digits to upper case based upon the magnitude of the byte of the hash. Note that re-generating the public address with checksum and comparing against the address in the file guarantees that the checksum in the file’s public address is correct.

⟨ *Add “checksum” to Ethereum public address* 85a ⟩ ≡

```
def checksumETHaddr(address):
    haddr = address.hex()
    formatted_address = ""
    addressHash = keccak_256(str(haddr).encode("UTF-8")).digest()[:40].hex()
    for i in range(40):
        if (int(addressHash[i], 16) >= 8 and int(haddr[i], 16) >= 10):
            formatted_address += str(haddr[i]).upper()
        else:
            formatted_address += str(haddr[i])
    return formatted_address
◇
```

Fragment referenced in 90.

5.2.2.3 Generate checksummed Ethereum address from private key

Generate a checksummed Ethereum address from a hexadecimal private key.

⟨ *Generate checksummed Ethereum address from private key* 85b ⟩ ≡

```
def Key_to_Ethereum_address(privkey_hex):
    private_key = bytes.fromhex(privkey_hex)
    public_key = PublicKey.from_valid_secret(private_key).format(compressed=False)[1:]
    public_addr_b = keccak_256(public_key).digest()[-20:]
    public_addr = checksumETHaddr(public_addr_b)
    return public_addr
◇
```

Fragment referenced in 90.

5.2.3 Input parsing utilities

5.2.3.1 Remove separators from address

In the interest of readability, `paper_wallet` allows inserting separators between groups of characters in the long public address and private key strings. This function recognises these sequences and removes them, allowing the raw addresses to be processed.

⟨ *Remove separators from address* 85c ⟩ ≡

```
sep = re.compile('<span class="s"></span>')

def removeSep(addr):
    return sep.sub("", addr, 0)
◇
```

Fragment referenced in 90.

5.2.3.2 Get next address, key pair

The `nextAddr()` function returns the next quadruple of label, public address, private key, and currency type from the wallet file. It reads either machine-readable CSV wallets or the HTML files generated from them by `paper_wallet`, which are intended to be printed to make offline cold storage wallets. This allows verifying the correctness of both formats of wallets, guarding against corruption creating HTML from the CSV master (or corruption after they are created).

Errors are dignosed internally, with error messages giving the line number in the file. At end of file, a triple consisting of three blank strings is returned. This function is agnostic as to currency type and address variants. It's up to the caller to recognise what the record represents.

⟨ *Get next address, key pair* 86 ⟩ ≡

```
l = ""
lineno = 0
html = False
comment = re.compile(r'\s*#')
csv = re.compile(r'(\w+),"(\w+)","(\w+)"')
ifile = fileinput.input()
goodRec = 0
badRec = 0

def nextAddr():
    global l, lineno, html, badRec
    EXnum = EXpub = EXpriv = False
    Hstate = 0

    while True:
        l = ifile.readline()
        if not l:
            break
        lineno += 1
        l = l.rstrip()
        if l.find("<!DOCTYPE html>") >= 0:
            html = True
        else:
```
◇

Fragment defined by 86, 87, 88a.
Fragment referenced in 90.

For HTML input, we look for the table item which precedes the fields and set a flag which causes the next line to be saved as the value for that field.

⟨ *Get next address, key pair* 87 ⟩ ≡

```
                if html:
                    if l.find('th class="num"') >= 0:
                        if Hstate == 0:
                            EXnum = True
                        else:
                            print("%d: HTML format error: %s" % (lineno, l))
                            badRec += 1
                            Hstate = 0
                    elif l.find('td class="pub"') >= 0:
                        if Hstate == 1:
                            EXpub = True
                        else:
                            print("%d: HTML format error: %s" % (lineno, l))
                            badRec += 1
                            Hstate = 0
                    elif l.find('td class="priv"') >= 0:
                        if Hstate == 2:
                            EXpriv = True
                        else:
                            print("%d: HTML format error: %s" % (lineno, l))
                            badRec += 1
                            Hstate = 0
                    else:
                        #   If one of the field flags has been set on the
                        #   previous line, save this one as the value of
                        #   that field.
                        if EXnum:
                            Hnum = l.lstrip().rstrip()
                            EXnum = False
                            Hstate = 1
                        elif EXpub:
                            Hpub = removeSep(l.lstrip().rstrip())
                            EXpub = False
                            Hstate = 2
                        elif EXpriv:
                            #   If this is the private key, we've now seen
                            #   all of the fields for this record.  Return
                            #   the fields to the caller.  We leave things
                            #   set to start scanning for the next record
                            #   when we're next called.
                            Hpriv = removeSep(l.lstrip().rstrip())
                            EXpriv = False
                            return (Hnum, Hpub, Hpriv, currencyID(Hpub, Hpriv))
```

◇

Fragment defined by 86, 87, 88a.
Fragment referenced in 90.

If the input is CSV, parse the fields and validate they are correct for one of the currencies we support.

⟨ *Get next address, key pair* 88a ⟩ ≡

```
            else:
                if not ((l == "") or comment.match(l)):
                    #   This is not a comment.  Try parsing as a CSV record
                    r = csv.match(l)
                    if r:
                        return (r.group(1), r.group(2), r.group(3),
                                currencyID(r.group(2), r.group(3)))
                    print("%d.  Cannot parse CSV record: %s" % (lineno, l))
                    badRec += 1
    return ("", "", "", "")
◇
```

Fragment defined by 86, 87, 88a.
Fragment referenced in 90.

5.2.3.3 Identify currency from address format

We allow users to validate cold storage wallets for either Bitcoin or Ethereum without requiring them to identify the currency the wallet uses. This is done by recognising the distinctive format used by the public address and private key for the respective currencies. Any of the multitude of currencies which share the Ethereum format may be verified by the Ethereum code. The regular expressions used to test address formats also serve to reject addresses which contain invalid characters or are improperly formatted.

⟨ *Identify currency from address format* 88b ⟩ ≡

```
ETHpub = re.compile(r"0x([\da-fA-F]+)")
ETHpriv = re.compile(r"0x([\da-fA-F]+)")
BTCpub = re.compile(r"(1[1-9A-HJ-NP-Za-km-z]+)")
BTCpriv = re.compile(r"([KL][1-9A-HJ-NP-Za-km-z]+)")

def currencyID(pub, priv):
    curr = "?"
    if (ETHpub.match(pub)) and (ETHpriv.match(priv)):
        curr = "ETH"
    elif (BTCpub.match(pub)) and (BTCpriv.match(priv)):
        curr = "BTC"
    return curr
◇
```

Fragment referenced in 90.

5.2.4 Validate addresses in file

This is the main processing loop of `validate_wallet`. It reads records from the input stream, parses them into number, public address, and private key, and verifies that the address can be re-generated from the key.

⟨ *Validate addresses in file* 89 ⟩ ≡

```
currency = ""

fileinput.input()
while True:
    (label, pubaddrW, privkey, rcurr) = nextAddr()
    if label == "": break
    if rcurr == "?":
        print("%d.  Record represents no known currency: %s" %
              (lineno, l))
        badRec += 1
    else:
        if (currency != "") and (currency != rcurr):
            print("%d.  Currency (%s) differs from that of previous record (%s)." %
                  (lineno, rcurr, currency))
        currency = rcurr

        pubaddr = ""
        if currency == "BTC":
            pubaddr = WIF_to_Bitcoin_address(privkey)
        elif currency == "ETH":
            pubaddr = "0x" + Key_to_Ethereum_address(privkey[2:])

    if pubaddrW != pubaddr:
        print("%d.  Mismatch on address %s.\n    Computed: %s\n    Wallet:   %s" %
              (lineno, label, pubaddr, pubaddrW))
        badRec += 1
    else:
        goodRec += 1
◇
```

Fragment referenced in 90.

5.2.5 Executable program

Here we put the pieces together into the complete program. Python does not permit forward references to functions, so we take care to declare all functions before they are referenced, independent of the logical order in which they are presented in the previous sections.

```
"python/validate_wallet.py" 90≡
     ⟨ Explanatory header for Python files 145b ⟩

     import fileinput
     import re
     import sys

     #   - - -  Bitcoin  - - -
     ⟨ Bitcoin library modules 84a ⟩
     ⟨ Extract private seed from WIF private address 84b ⟩
     ⟨ Generate public address from WIF private key 84c ⟩

     #   - - -  Ethereum  - - -
     ⟨ Ethereum library modules 84d ⟩
     ⟨ Add "checksum" to Ethereum public address 85a ⟩
     ⟨ Generate checksummed Ethereum address from private key 85b ⟩

     #   - - -  Utility functions  - - -
     ⟨ Remove separators from address 85c ⟩
     ⟨ Get next address, key pair 86, ... ⟩
     ⟨ Identify currency from address format 88b ⟩

     ⟨ Validate addresses in file 89 ⟩

     print("Addresses: %d good, %d bad." % (goodRec, badRec))
     sys.exit(0 if ((goodRec > 0) and (badRec == 0)) else 1)
     ◇
```

Chapter 6

Cold Storage Monitor

The `cold_comfort` program monitors a list of Bitcoin or Ethereum addresses, queries their current balance from free servers, compares it with the expected balance, and reports any discrepancies. This can be employed by users of offline "cold storage" to detect any unauthorised transactions referencing them.

6.1 Program plumbing

"perl/cold_comfort.pl" 91a≡

```
    ⟨ Explanatory header for Perl files 145a ⟩

        ⟨ Perl language modes 144a ⟩
    ◇
```

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.2 Required library modules

"perl/cold_comfort.pl" 91b≡

```

        use Crypt::Random::Seed;
        use Getopt::Long;
        use JSON;
        use LWP::Simple;
        use List::Util qw(shuffle);
        use Math::Random::MT;
        use Text::CSV qw(csv);
    ◇
```

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.3 Definitions and mode settings

"perl/cold_comfort.pl" 92a≡

```
        use constant SATOSHI => 0.00000001;
        use constant ERRFLAG => " ****";

        my $APIretry = 3;                   # Maximum attempts to make API query
        my $ignoreZero = FALSE;             # Ignore zero balance addresses
        my $dust = 0.001;                   # Don't report balance increases less than this
        my $loop = FALSE;                   # Loop forever checking addresses
        my $shuffle = FALSE;                # Shuffle order of address queries
        my $verbose = FALSE;                # Show all queries, not just alerts
        my $waitconst = 17;                 # Constant wait between queries, seconds
        my $waitloop = 3600;                # Wait between series of queries
        my $waitrand = 20;                  # Random wait between queries, seconds
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.4 Data sources for address balance queries

The following sites can be queried for the balance of Bitcoin and Ethereum addresses, respectively. The user can select the site used with the `-btcsource` and `-ethsource` command line options. These sites tend to come and go, so we provide three alternatives for each. Note that adding a new site involves more than just adding an entry to one of these tables: you must write a function which composes a query, sends it to the server, and parses the result it returns.

"perl/cold_comfort.pl" 92b≡

```
        #   Bitcoin data sources
        my $srcBTC = "blockchain.info";
        my %btcSource = (
            "blockchain.info"   =>  \&s_b_blockchain,
            "blockcypher.com"   =>  \&s_b_blockcypher,
            "btc.com"           =>  \&s_b_btc
        );

        #   Ethereum data sources
        my $srcETH = "blockchain.com";
        my %ethSource = (
            "blockchain.com"    =>  \&s_e_blockchain,
            "etherscan.io"      =>  \&s_e_etherscan,
            "ethplorer.io"      =>  \&s_e_ethplorer
        );
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Uses: `s_b_blockchain` 98b, `s_b_blockcypher` 98a, `s_b_btc` 99a, `s_e_blockchain` 99b, `s_e_etherscan` 100a, `s_e_ethplorer` 100b.

6.5 Initialisation and command line option processing

"perl/cold_comfort.pl" 93≡

```
        randInit();

        GetOptions(
            "btcsource=s"   =>  \$srcBTC,
            "dust=f"        =>  \$dust,
            "ethsource=s"   =>  \$srcETH,
            "help"          =>  \&showHelp,
            "loop"          =>  \$loop,
            "retry=i",      =>  \$APIretry,
            "shuffle"       =>  \$shuffle,
            "verbose"       =>  \$verbose,
            "waitconst=f"   =>  \$waitconst,
            "waitloop=f"    =>  \$waitloop,
            "waitrand=f"    =>  \$waitrand,
            "zero"          =>  \$ignoreZero
        ) || die("Command line option error");

        #   Validate address query source specifications

        if (!defined($btcSource{$srcBTC})) {
            print("Unknown Bitcoin query source.\n");
            exit(2);
        }

        if (!defined($ethSource{$srcETH})) {
            print("Unknown Ethereum query source.\n");
            exit(2);
        }
    ◇
```

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Uses: `randInit` 142a.

6.6 First pass: Read list of addresses to be monitored

Read the list of addresses from the input stream. The addresses to be watched are specified in CSV format with the following fields.

1. Label
2. Public address
3. Private key (ignored if specified)
4. Expected balance

These are stored in an array of arrays, with an additional item, initialised to zero, added to the end which is used to keep track of the number of retries for queries that failed.

"perl/cold_comfort.pl" 94a≡

```
my $csv = Text::CSV->new({ binary => 1 }) ||
    die("Cannot use CSV: " . Text::CSV->error_diag());

my $adrs = [ ];
while (my $l = <>) {
    chomp($l);
    $l =~ s/^\s+//;
    $l =~ s/\s+$//;
    if (($l ne "") && ($l !~ m/^#/)) {
        if ($csv->parse($l)) {
            push(@$adrs, [ $csv->fields, 0 ]);
       }
    }
}
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.7 Second pass: Query addresses and report discrepancies

If `-shuffle` is specified, we shuffle the order in which addresses are queried. This may make it more difficult for query services to identify our requests as representing a fixed collection of cold storage addresses.

"perl/cold_comfort.pl" 94b≡

```
my ($balErrs, $APIerrs);

do {
    if ($shuffle) {
        @$adrs = shuffle(@$adrs);
    }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

Query the balance of the addresses and compare against the expected balance, reporting any discrepancies.

"perl/cold_comfort.pl" 95a≡

```
            ($balErrs, $APIerrs) = (0, 0);
            for (my $i = 0; $i < scalar(@$adrs); $i++) {
                my ($label, $bcaddr, $balance, $tries) =
                    ($adrs->[$i][0], $adrs->[$i][1], $adrs->[$i][3], $adrs->[$i][4]);
                if ((!$ignoreZero) || ($balance > 0)) {
                    $balance += 0;
                    my $warn = "";
                    my $cbal;
                    my $cbalf;
                    if ($bcaddr =~ m/^0x/i) {
                        $cbal = $ethSource{$srcETH}($bcaddr);
                    } else {
                        $cbal = $btcSource{$srcBTC}($bcaddr);
                    }
                    if (defined($cbal)) {
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

We compare the balance reported by the query with the expected balance using a slightly complicated set of rules. Due to floating point round-off and rounding in values reported by servers, we ignore any discrepancy less than one `SATOSHI` (10^{-8}). If the reported balance is less than expected by more than this threshold, we treat it as an error. If the reported balance is greater, we compare it with the `-dust` setting. Cryptocurrency blockchains, particularly Bitcoin at this writing, are afflicted by spammers who send nugatory funds to addresses with significant balances to promote a variety of scams. These small deposits are referred to as "dust", in that the transaction cost to spend or transfer them exceeds their value. But they can cause discrepancies in the balance comparison. We ignore these balance increases up to the `-dust` threshold. If you're getting dust reports and confirm that's what they are, just update the balance in your cold storage database to include the dust it has collected.

"perl/cold_comfort.pl" 95b≡

```
                        my $bdiff = $cbal - $balance;
                        $cbalf = sprintf("%16.8f", $cbal);
                        if ($bdiff < -(SATOSHI)) {
                             $warn = ERRFLAG;
                             $balErrs++;
                        } elsif ($bdiff > SATOSHI) {
                             $warn = ($cbal < ($balance + $dust)) ? " Dust" : ERRFLAG;
                        }
                    } else {
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

If the API query for the address balance fails, increment the number of queries made for it. If we've made fewer than the number of tries set by `-retry`, re-queue the query at the end of the address list for next try. If the number of tries has been exhausted, this is flagged as a hard fail and abandoned.

"perl/cold_comfort.pl" 96≡

```
                    $cbalf = " " x 16;
                    $tries++;
                    if ($tries < $APIretry) {
                        if ($verbose) {
                            $warn = " API fail, try $tries/$APIretry";
                        }
                        push(@$adrs, [ $label, $bcaddr, $adrs->[$i][2], $balance, $tries ]);
                    } else {
                        $warn = " API failure";
                        $APIerrs++;
                    }
                }
                if ($verbose || ($warn ne "")) {
                    printf("%-12s  %-42s  %16.8f  %s%s\n",
                           $label, $bcaddr, $balance, $cbalf, $warn);
                }
                if ($i < (scalar(@$adrs) - 1)) {
                    sleep($waitconst + randNext($waitrand));
                }
            }
        }
        if ($loop && ($waitloop > 0)) {
            sleep($waitloop);
        }
    } while ($loop);

    exit(($balErrs > 0) ? 1 : (($APIerrs > 0) ? 2 : 0));
```
◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Uses: randNext 142b.

6.8 Local functions

6.8.1 `showHelp`: Show how to call information

"perl/cold_comfort.pl" 97a≡

```
    sub showHelp {
        my $btcsites = join(", ", sort(keys(%btcSource)));
        my $ethsites = join(", ", sort(keys(%ethSource)));
        my $help = <<"EOD";
perl cold_comfort.pl [ options... ] address_file...
  Options:
    -btcsource site     Site to query for Bitcoin balances: $btcsites
    -dust n             Ignore "dust" sent to address less than n units
    -ethsource site     Site to query for Ethereum balances: $ethsites
    -help               Print this message
    -loop               Loop forever polling addresses
    -retry n            Try failed API query requests n times
    -shuffle            Shuffle order in which addresses queried
    -verbose            Show all polls, regardless of error status
    -waitconst n        Wait constant n seconds between queries
    -waitloop n         Wait n seconds between re-polls in -loop
    -waitrand n         Wait random time 0 to n seconds between address polls
    -zero               Ignore addresses with zero expected balance
EOD
        print($help);
        exit(0);
    }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.9 Utility functions

"perl/cold_comfort.pl" 97b≡

⟨ *Pseudorandom number generator* 142a, ... ⟩

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.

6.10 Address query source handlers

These functions query different services to obtain the balance for a specified Bitcoin or Ethereum address. The argument is the address and the value returned is the balance as a floating point value of currency units or `undef` if the query fails.

6.10.1 Bitcoin

6.10.1.1 blockcypher.com

"perl/cold_comfort.pl" 98a≡

```
sub s_b_blockcypher {
    my ($address) = @_;

    my $balance;
    my $reply = get("https://api.blockcypher.com/v1/btc/main/addrs/$address/balance");
    if (defined($reply)) {
        my $r = decode_json($reply);
        $balance = $r->{balance} * SATOSHI;
    }

    return $balance;
}
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_b_blockcypher` 92b.

6.10.1.2 blockchain.info

"perl/cold_comfort.pl" 98b≡

```
sub s_b_blockchain {
    my ($address) = @_;

    my $balance = get("https://blockchain.info/q/addressbalance/$address");

    if (defined($balance)) {
        $balance *= SATOSHI;
    }

    return $balance;
}
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_b_blockchain` 92b.

6.10.1.3 btc.com

"perl/cold_comfort.pl" 99a≡

```
        sub s_b_btc {
            my ($address) = @_;

            my $balance;

            my $request = LWP::UserAgent->new();
            $request->agent("cold_comfort");
            my $response = $request->get("https://btc.com/btc/search?q=$address");
            if ($response->is_success) {
                my $reply = $response->content;
                if ($reply =~ m:Balance</div><div\s+class="ant-col\s+ant-col-24\s+text-c">([\d\.\,]+)\s+BTC</div>
                    $balance = $1;
                    $balance =~ s:[,<>/b]::g;
                    $balance = $balance + 0;
                }
            }

            return $balance;
        }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_b_btc` 92b.

6.10.2 Ethereum

6.10.2.1 blockchain.com

"perl/cold_comfort.pl" 99b≡

```
        sub s_e_blockchain {
            my ($address) = @_;

            my $balance;
            my $reply = get("https://www.blockchain.com/eth/address/$address");
            if (defined($reply)) {
                if ($reply =~ m/The\s+current\s+value\s+of\s+this\s+address\s+is\s+([\d\.,]+)\s+ETH/) {
                    $balance = $1;
                    $balance =~ s/,//g;
                    $balance = $balance + 0;
                }
            }

            return $balance;
        }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_e_blockchain` 92b.

6.10.2.2 etherscan.io

"perl/cold_comfort.pl" 100a≡

```
        sub s_e_etherscan {
            my ($address) = @_;

            my $balance;
            my $request = LWP::UserAgent->new();
            $request->agent("cold_comfort");
            my $response = $request->get("https://etherscan.io/address/$address");
            if ($response->is_success) {
                my $reply = $response->content;
                if ($reply =~ m:<div\s+class="col\-md\-8">([\d\.,<>/b]+)\s+Ether</div>:) {
                    $balance = $1;
                    $balance =~ s:[,<>/b]::g;
                    $balance = $balance + 0;
                }
            }

            return $balance;
        }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_e_etherscan` 92b.

6.10.2.3 ethplorer.io

"perl/cold_comfort.pl" 100b≡

```
        sub s_e_ethplorer {
            my ($address) = @_;

            my $balance;
            my $reply = get("https://api.ethplorer.io/getAddressInfo/$address?apiKey=freekey");
            if (defined($reply)) {
                my $r = decode_json($reply);
                if ($r->{address} eq lc($address)) {
                    $balance = $r->{ETH}->{balance};
                }
            }

            return $balance;
        }
```

◇

File defined by 91ab, 92ab, 93, 94ab, 95ab, 96, 97ab, 98ab, 99ab, 100ab.
Defines: `s_e_ethplorer` 92b.

Chapter 7

Bitcoin Address Watcher

This program monitors the Bitcoin blockchain and, whenever new blocks are added, scans them for transactions involving addresses on a watch list, which may be specified on the command line, from a file, or from the user's wallet. For every transaction inolving that address, whether as input or output, a message on standard output and an optional permanent log entry is generated showing:

1. Label (if any) from the watch list file or wallet
2. Bitcoin address
3. Value of transaction in BTC
4. Date and time
5. Block number (height)
6. Transaction ID
7. Block hash

7.1 Main program

We start with the usual start-of-program definitions and declaring and processing the command-line options.

```
"perl/address_watch.pl" 102≡
     ⟨Explanatory header for Perl files 145a⟩

         ⟨Perl language modes 144a⟩

         use LWP;
         use JSON;
         use Text::CSV qw(csv);
         use Getopt::Long qw(GetOptionsFromArray);
         use POSIX qw(strftime);
         use Term::ReadKey;
         use Statistics::Descriptive;

         use Data::Dumper;

         my $block_start = ⟨AW block start 3a⟩;          # Starting block
         my $block_end = ⟨AW block end 3b⟩;               # End block
         my $block_file = "⟨AW block file 3c⟩";          # Incremental scanning block file
         my @watch_addrs;                                # Addresses to watch
         my $watch_file = "⟨AW watch file 3d⟩";           # File containing watch addresses
         my $log_file = "⟨AW log file 3e⟩";              # Log file
         my $verbose = ⟨Verbosity level 2b⟩;            # Verbose output ?
         my $poll_time = ⟨Blockchain poll interval 2c⟩;  # Poll interval in seconds
         my $last_block_time = -1;                        # Time of last block
         my $b_interval_smoothed = -1;                    # Smoothed inter-block interval, seconds
         my $b_interval_smoothing = 0.2;                  # Interval smoothing factor
         my $stats = FALSE;                               # Show statistics of blocks ?
         my $statlog = "";                                # Block statistics log file
         my $wallet = ⟨AW monitor wallet 3f⟩;             # Monitor unspent funds in wallet ?
         ⟨RPC configuration variables 139b⟩

         my %options = (
             ⟨RPC command line options 139c⟩
             "bfile=s"       => \$block_file,
             "end=i"         => \$block_end,
             "help"          => \&showHelp,
             "lfile=s"       => \$log_file,
             "poll=i"        => \$poll_time,
             "sfile=s"       => \$statlog,
             "start=i"       => \$block_start,
             "stats"         => \$stats,
             "type=s"        =>  sub { print("$_[1]\n"); },
             "verbose+"      => \$verbose,
             "wallet"        => \$wallet,
             "watch=s"       => \@watch_addrs,
             "wfile=s"       => \$watch_file
         );

         processConfiguration();

         GetOptions(
             %options
         ) || die("Command line option error");

         my $statc = $stats || ($statlog ne "");
     ◇
```

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: processConfiguration 135a.

7.1.1 Build list of watched addresses

Next, build the list of Bitcoin addresses we'll be watching. These may be specified on the command line with the `-watch` option, or read from a (single) file specified by the `-wfile` option. In addition, addresses in the user's wallet with an unspent balance can be automatically monitored by specifying the `-wallet` option. When watching wallet addresses, we re-fetch the list for every poll of the blockchain to accommodate any changes due to transactions since the previous poll.

When reading the list of addresses from a `-wfile` CSV file, we ignore blank lines, comments which begin with "`#`", and Ethereum addresses which begin with "`0x`".

`"perl/address_watch.pl"` 103≡

```
        my %adrh;

        #   Add watch addresses specified on the command line
        foreach my $a (@watch_addrs) {
            my ($label, $balance) = ("", "");

            if ($a =~ s/^(\w+),//) {
                $label = $1;
            }
            if ($a =~ s/,[\d\.]+$//) {
                $balance = $1;
            }
            $adrh{$a} = [ $label, $balance ];
        }
        undef(@watch_addrs);

        #   Add watch addresses specified in a -wfile
        if ($watch_file ne "") {
            my $csv = Text::CSV->new({ binary => 1 }) ||
                die("Cannot use CSV: " . Text::CSV->error_diag());
            open(WF, "<$watch_file") || die("Cannot open $watch_file");
            while (my $l = <WF>) {
                chomp($l);
                $l =~ s/^\s+//;
                $l =~ s/\s+$//;
                if (($l ne "") && ($l !~ m/^#/)) {
                    if ($csv->parse($l)) {
                        my @fields = $csv->fields;
                        if ($fields[1] !~ m/^0x/i) {
                            $adrh{$fields[1]} = [ $fields[0], $fields[3] ];
                        }
                    }
                }
            }
            close(WF);
        }

        if (scalar(keys(%adrh)) == 0) {
            print(STDERR "No watch addresses specified.\n");
            exit(2);
        }
```
◇

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.

7.1.2 Prompt for RPC password

If the "`rpc`" query method was selected and no password was specified, ask the user for it from standard input.

`"perl/address_watch.pl"` 104≡

```
        if (($RPCmethod eq "rpc") &&
            ((!defined($RPCpass)) || ($RPCpass eq ""))) {
            $RPCpass = getPassword("Bitcoin RPC password: ");
        }
    ◇
```

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: `$RPCmethod` 139b, `$RPCpass` 139b, `getPassword` 135b.

7.1.3 Determine range of blocks to scan

Determine the start and ending blocks to scan. This depends in a non-trivial but convenient way on the **`-start`**, **`-end`**, and **`-bfile`** options. If **`-bfile`** is specified, the start block will be read from it. Otherwise, the start block will be that specified by **`-start`** or, if -1 (the default), the most recent block. If a negative start block is specified, the scan will start at that number of blocks before the most recent block.

If no end block is specified, the most recent block is used. This means that if we've already scanned the most recent block, it will not be re-scanned.

"perl/address_watch.pl" 105≡

```
#   If a block file is present, read start block from it
if ($block_file ne "") {
    open(BF, "<$block_file") || die("Cannot open block file $block_file");
    my $l = <BF>;
    close(BF);
    chomp($l);
    if ($l =~ m/(\d+)/) {
        $block_start = $1 + 1;
    } else {
        print(STDERR "Invalid value in block file.\n");
        exit(2);
    }
}

#   If no end block specified, use most recent block

if ($block_end < 0) {
    $block_end = sendRPCcommand([ "getblockcount" ]);
}

#   If negative start block specified, start that number
#   before the last block.

if ($block_start < 0) {
    $block_start = $block_end + $block_start
}

if (($block_start > $block_end) && ($poll_time == 0)) {
    print("No blocks to scan.\n");
    exit(0);
}
```
◇

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: sendRPCcommand 136a.

7.1.4 Retrieve and scan blocks

Having determined the range to blocks to scan, proceed to scan them and accumulate references to addresses we're watching within them. Before entering the scanning loop, we perform an initial scan of the wallet for addresses with unspent balances. This avoids missing any address whose balance changed between the time we started the program and the first block we receive after starting.

"perl/address_watch.pl" 106≡

```
updateWalletAddresses();

do {
    my $myaddrs = [];

    if ($block_start <= $block_end) {
        print("Scanning blocks $block_start to $block_end.\n") if $verbose;
    }

    my $scanned = 0;
    for (my $j = $block_start; $j <= $block_end; $j++) {
        if ($wallet && ($j == $block_start)) {
            updateWalletAddresses();
        }
        print("  Scanning block $j.\n") if $verbose;
        my $mine = scanBlock($j, $verbose);
        if (scalar(@$mine) > 0) {
            push(@$myaddrs, $mine);
        }
        $scanned++;
    }
```

◇

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: `scanBlock` 109a, `updateWalletAddresses` 114a.

7.1.5 Display references to watched addresses

We've finished scanning all specified blocks. Output references we've found in them to addresses we're watching.

"perl/address_watch.pl" 107a≡

```
my $nref = (scalar(@$myaddrs) == 0) ? 0 : scalar(@{$myaddrs->[0]});
if (scalar($nref) > 0) {
    printf("%d reference%s to watched addresses:\n",
        $nref, ($nref > 1) ? "s" : "");
    for (my $i = 0; $i < $nref; $i++) {
        my ($b_height, $b_hash, $b_time, $t_txid, $a_addr, $t_value) =
           ($myaddrs->[0]->[$i]->[0], $myaddrs->[0]->[$i]->[1],
            $myaddrs->[0]->[$i]->[2], $myaddrs->[0]->[$i]->[3],
            $myaddrs->[0]->[$i]->[4], $myaddrs->[0]->[$i]->[5]);
        my $utime = etime($b_time);

        my $logItem = sprintf("%-12s  %36s  %11.8f  %19s  %8d  %64s  %64s\n",
            $adrh{$a_addr}->[0], $a_addr, $t_value, $utime, $b_height, $t_txid, $b_hash);
        print($logItem);
        if ($log_file ne "") {
            open(LF, ">>$log_file")|| die("Cannot open log file $log_file");
            printf(LF "\"%s\",%s,%.8f,%s,%d,%s,%s\n",
                $adrh{$a_addr}->[0], $a_addr, $t_value, $utime,
                $b_height, $t_txid, $b_hash);
            close(LF);
        }
   }
}
```

◇

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: etime 132.

7.1.6 Save last block scanned for next run

If a block file is specified, save the last block scanned so we can resume with the next block on a subsequent run.

"perl/address_watch.pl" 107b≡

```
if (($block_file ne "") && ($scanned > 0)) {
    open(BF, ">$block_file") || die("Cannot open block file for update");
    print(BF "$block_end\n");
    close(BF);
    print("Updated block file to last block $block_end.\n") if $verbose;
}
```

◇

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.

7.1.7 If polling, wait and resume scan

If we're polling, sleep for the specified polling interval and resume the scan with the next block after the one we've just examined.

"perl/address_watch.pl" 108a≡

```
            if ($poll_time > 0) {
                $block_start = $block_end + 1;
                sleep($poll_time);
                print("Resuming scan after $poll_time seconds at " .
                    etime(time()) . ".\n") if $verbose;
                $block_end = sendRPCcommand([ "getblockcount" ]);
            }
        } while ($poll_time > 0);
    ◇
```

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.
Uses: `etime` 132, `sendRPCcommand` 136a.

7.1.8 Local and utility functions

Import local (program-specific) functions defined below and utlity functions common to multiple programs.

"perl/address_watch.pl" 108b≡

```
        #   Local functions
        ⟨ scanBlock: Scan a block by index on the blockchain 109a, ... ⟩
        ⟨ updateWalletAddresses: Watch unspent wallet addresses 114a, ... ⟩
        ⟨ showHelp: Show address watch help information 115 ⟩

        #   Utility functions
        ⟨ etime: Edit time to ISO 8601 132 ⟩
        ⟨ Command and option processing 133, ... ⟩
        ⟨ sendRPCcommand: Send a Bitcoin RPC/JSON command 136a ⟩
        ⟨ getPassword: Prompt user to enter password 135b ⟩
        ⟨ blockReward: Compute reward for mining block 140b ⟩
    ◇
```

File defined by 102, 103, 104, 105, 106, 107ab, 108ab.

7.2 Local functions

7.2.1 `scanBlock` — Scan a block by index on the blockchain

The transactions in the block are scanned for references to addresses on the watch list. Any found are returned as an array of arrays, with each containing:

0. Block number
1. Block hash
2. Time of block
3. Transaction ID
4. Address referenced
5. Value from transaction

From this, the caller can recover the details from the transaction and see what's going on.

⟨ *scanBlock: Scan a block by index on the blockchain* 109a ⟩ ≡

```
sub scanBlock {
    my ($height, $verbose) = @_;

    my @hits;
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.
Defines: `scanBlock` 106.

Get hash for block from its number (height), then fetch the block from the blockchain.

⟨ *scanBlock: Scan a block by index on the blockchain* 109b ⟩ ≡

```
    my $bh = sendRPCcommand([ "getblockhash", "$height" ]);
    print("    Block hash $bh\n") if $verbose;
    my $blk = sendRPCcommand([ "getblock",  $bh, "2" ]);
    my $r = decode_json($blk);
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.
Uses: `sendRPCcommand` 136a.

Scan the block for references to addresses we're watching. We start by extracting the block-level information.

⟨ *scanBlock: Scan a block by index on the blockchain* 109c ⟩ ≡

```
    my $b_height = $r->{height};        # Block height (index)
    my $b_hash = $r->{hash};            # Block hash
    my $b_time = $r->{time};            # Block time
    my $b_nTx = $r->{nTx};              # Transactions in block

    print("    Block $b_height " . gmtime($b_time) .
          " Transactions $b_nTx\n") if $verbose >= 1;

    my ($stat_value, $stat_size);
    my $stat_reward = 0;
    if ($statc) {
        $stat_value = Statistics::Descriptive::Sparse->new();
        $stat_size = Statistics::Descriptive::Sparse->new();
    }
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.

Now loop over the transactions in the block, saving any which cite one of the addresses we're watching. After we've scanned them all, return any references to watched addresses.

⟨ *scanBlock: Scan a block by index on the blockchain* 110 ⟩ ≡

```
            my %vincache;

            for (my $t = 0; $t < $b_nTx; $t++) {
                #   Transaction ID
                my $t_txid = $r->{tx}->[$t]->{txid};
                if ($statc) {
                    $stat_size->add_data($r->{tx}->[$t]->{vsize});
                }
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.

The source of funds for the transaction is specified by one or more "`vin`" items. These do not directly specify the address, but rather give the transaction in which the funds may be found and the "`vout`" item within it that contains the address(es). To check for references to our addresses, we must look up each of these transactions, which requires that Bitcoin Core be configured with "`txindex=1`", causing it to build and maintain a transaction index. If this index is absent, we cannot monitor input addresses.

Because looking up transactions and decoding them from JSON is costly, we cache transactions we query in `%vincache` and serve previously-retrieved and decoded objects from the cache. This dramatically speeds up processing queries for many blocks.

One additional wrinkle is that an input transaction may have its source be the "coinbase": newly-created Bitcoin paid to miners as incentive for publishing blocks. These transactions have no previous transaction and hence no addresses in their "`vout`" section. Since there are no addresses to check, we needn't examine such transactions further.

⟨ *scanBlock: Scan a block by index on the blockchain* 111 ⟩ ≡

```
                my $t_nvin = scalar(@{$r->{tx}->[$t]->{vin}});
                my $t_nvout = scalar(@{$r->{tx}->[$t]->{vout}});

                print("  $t.  $t_txid  In: $t_nvin  Out: $t_nvout\n") if $verbose >= 2;

                for (my $v = 0; $v < $t_nvin; $v++) {
                    if (defined($r->{tx}->[$t]->{vin}->[$v]->{txid}) &&
                        defined($r->{tx}->[$t]->{vin}->[$v]->{vout})) {
                        my ($vintx, $vinn) = ($r->{tx}->[$t]->{vin}->[$v]->{txid},
                            $r->{tx}->[$t]->{vin}->[$v]->{vout});
                        my $vi;
                        if (!defined($vi = $vincache{$vintx})) {
                            my $vitx = sendRPCcommand([ "getrawtransaction",  $vintx, "true" ]);
                            $vi = decode_json($vitx);
                            $vincache{$vintx} = $vi;
                        }
                        if (defined($vi->{vout}->[$vinn]->{scriptPubKey}->{addresses})) {
                            #   This is not a "coinbase" transaction.  Scan source addresses
                            my $vi_naddr = scalar(@{$vi->{vout}->[$vinn]->{scriptPubKey}->{addresses}});
                            #   Loop over addresses in vout item
                            for (my $a = 0; $a < $vi_naddr; $a++) {
                                my $a_addr = $vi->{vout}->[$vinn]->{scriptPubKey}->{addresses}->[$a];
                                my $t_value = $vi->{vout}->[$vinn]->{value};
                                if (!defined($t_value)) {
                                    $t_value = 0;
                                }
                                my $flag = $adrh{$a_addr};
                                if ($verbose >= 3) {
                                    my $pflag = $flag ? " *****" : "";
                                    print("      In  $v.$a.  $a_addr$pflag\n");
                                }
                                if ($flag) {
                                    #   This is one of the addresses we're watching: add to the hit list
                                    push(@hits, [ $b_height, $b_hash, $b_time, $t_txid, $a_addr, -$t_value ]);
                                }
                            }
                        }
                    }
                }
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.
Uses: `sendRPCcommand` 136a.

The "`vout`" items in the transaction specify the addresses (or scripts) to which the funds are to be sent. These are more straightforward to process than "`vin`" items, as they contain the actual address and do not require us to look up a transaction to find it.

⟨ *scanBlock: Scan a block by index on the blockchain* 112 ⟩ ≡

```
            #   Loop over vout items
            for (my $v = 0; $v < $t_nvout; $v++) {
                if (defined($r->{tx}->[$t]->{vout}->[$v]->{scriptPubKey}) &&
                    defined($r->{tx}->[$t]->{vout}->[$v]->{scriptPubKey}->{addresses})) {
                    my $v_naddr = scalar(@{$r->{tx}->[$t]->{vout}->[$v]->{scriptPubKey}->{addresses}});
                    #   Loop over addresses in vout item
                    for (my $a = 0; $a < $v_naddr; $a++) {
                        my $a_addr = $r->{tx}->[$t]->{vout}->[$v]->{scriptPubKey}->{addresses}->[$a];
                        my $t_value = $r->{tx}->[$t]->{vout}->[$v]->{value};
                        if (!defined($t_value)) {
                            $t_value = 0;
                        }
                        if ($t == 0) {
                            $stat_reward += $t_value;
                        }
                        my $flag = $adrh{$a_addr};
                        if ($verbose >= 3) {
                            my $pflag = $flag ? " *****" : "";
                            print("      Out $v.$a.  $a_addr$pflag\n");
                        }
                        if ($flag) {
                            #   This is one of the addresses we're watching: add to the hit list
                            push(@hits, [ $b_height, $b_hash, $b_time, $t_txid, $a_addr, $t_value ]);
                        }
                        if ($statc && ($t_value > 0)) {
                            $stat_value->add_data($t_value);
                        }
                    }
                }
            }
        }
        if ($statc) {
            ⟨ Show statistics for block 113 ⟩
        }
        $last_block_time = $b_time;
        return \@hits;
    }
```

◇

Fragment defined by 109abc, 110, 111, 112.
Fragment referenced in 108b.

7.2.1.1 Show statistics for block

If we're collecting statistics for blocks, output them in either/or primate readable form on standard output and a record appended to the log file specified by the `-sfile` option. Statistics include:

- Block number ("height")
- Date and time
- Number of transactions
- Transaction size: minimum, maximum, mean, standard deviation, and total
- Value: minimum, maximum, mean, standard deviation, and total

⟨ *Show statistics for block* 113 ⟩ ≡

```
    if ($stats) {
        print("  Block $b_height  " . etime($b_time) . " $b_nTx transactions\n");
        my $brw = blockReward($b_height);
        printf("    Reward %.2f (mining block %.2f, transaction fees %.2f)\n",
               $stat_reward, $brw, $stat_reward - $brw);
        printf("    Size: min %d  max %d  mean %.2f  SD %.2f  Total %d\n",
            $stat_size->min(), $stat_size->max(), $stat_size->mean(),
            $stat_size->standard_deviation(), $stat_size->sum());
        printf("    Value: min %.8f  max %.8g  mean %.8g  SD %.8g  Total %.8g\n",
            $stat_value->min(), $stat_value->max(), $stat_value->mean(),
            $stat_value->standard_deviation(), $stat_value->sum());
        if ($last_block_time > 0) {
            my $b_interval = $b_time - $last_block_time;
            if ($b_interval_smoothed >= 0) {
                $b_interval_smoothed = $b_interval_smoothed +
                    ($b_interval_smoothing *
                        ($b_interval - $b_interval_smoothed));
            } else {
                $b_interval_smoothed = $b_interval;
            }
            printf("    Time since last block: %.2f minutes, smoothed %.2f.\n",
                $b_interval / 60, $b_interval_smoothed / 60);
        }

    }
    if ($statlog) {
        open(SL, ">>$statlog");
            printf(SL "%12d,%d,%d,%d,%d,%.2f,%.2f,%d,%.8f,%.8g,%.8g,%.8g,%.8g,%.8g,%.8g\n",
                $b_height, $b_time, $b_nTx,
                $stat_size->min(), $stat_size->max(), $stat_size->mean(),
                $stat_size->standard_deviation(), $stat_size->sum(),
                $stat_value->min(), $stat_value->max(), $stat_value->mean(),
                $stat_value->standard_deviation(), $stat_value->sum(),
                $stat_reward, blockReward($b_height));
        close(SL);
    }
```
◇

Fragment referenced in 112.
Uses: `blockReward` 140b, `etime` 132.

7.2.2 `updateWalletAddresses` — Watch unspent wallet addresses

When the `-wallet` option is specified, every time we begin a poll for new blocks on the blockchain, we obtain the current list of addresses within the wallet which have a nonzero balance. These are automatically added to the watch list, so we'll monitor them without the need for the user to manually watch them. Since any spend transaction will result in a wallet address disappearing and a new change address replacing it, wallet addresses are dynamic, and this keeps the monitor up to date. On every scan, addresses previously added from the wallet are removed, so on each scan the list is current as of the time it began.

We start by removing any expired wallet addresses from the watch list. When a wallet address is added to the watch list or updated if already present, we append the time to the record for the address. When an address disappears from the wallet, this may mean that its balance has been zeroed out due to being spent and replaced by a new address with the change from the transaction. We still want to monitor the original address, however, in order to log the transaction in which the funds from it were spent. Thus, we only purge an address from the wallet watch list after the time specified by “AW wallet purge interval” has expired. This should be set to a time greater than the longest time expected between a transaction’s being sent to the mempool and the first confirmation arriving on the blockchain.

⟨ *updateWalletAddresses: Watch unspent wallet addresses* 114a ⟩ ≡

```
        sub updateWalletAddresses {
            my $now = time();

            foreach my $adr (keys(%adrh)) {
                 if (($adrh{$adr}->[1] =~ m/^W/) &&
                     (($now - $adrh{$adr}->[2]) > ⟨ AW wallet purge interval 3g ⟩)) {
                     printf("Purged wallet address $adr, age %d seconds.\n",
                        $now - $adrh{$adr}->[2]); # if ($verbose >= 2);
                     delete($adrh{$adr});
                 }
            }
```

◇

Fragment defined by 114ab.
Fragment referenced in 108b.
Defines: `updateWalletAddresses` 106.

Retrieve addresses with an unspent balance from the wallet and add them to the watch list. Any addresses already on the list will have their time of last presence updated to reset the expiration purge time.

⟨ *updateWalletAddresses: Watch unspent wallet addresses* 114b ⟩ ≡

```
            #   Retrieve unspent addresses from wallet and add to watch hash
            my $uw = sendRPCcommand([ "listunspent" ]);
            if (defined($uw)) {
                my $w = decode_json($uw);
                for (my $i = 0; $i < scalar(@$w); $i++) {
                    my $addr = $w->[$i]->{address};
                    my $balance = $w->[$i]->{amount};
                    my $label = $w->[$i]->{label};
                    if (!defined($label)) {
                        $label = "Wallet" . ($i + 1);
                    }
                    print("Watching wallet $label,$addr,W$balance,$now\n") if ($verbose >= 2);
                    $adrh{$addr} = [ $label, "W$balance", $now ];
                }
            }
        }
```

◇

Fragment defined by 114ab.
Fragment referenced in 108b.
Uses: `sendRPCcommand` 136a.

7.2.3 showHelp — Show help information

⟨ *showHelp: Show address watch help information* 115 ⟩ ≡

```
        sub showHelp {
            my $help = <<"    EOD";
    perl address_watch.pl [ option... ] address_file
      Commands and arguments:
        -bfile filename     Set file to save last block scanned
        -end n              Last block to scan
        -help               Print this message
        -lfile filename     Set log file
        -poll n             Poll for new block every n seconds, 0 = never
        -sfile filename     Write block statistics to named file
        -start n            First block to scan
        -stats              Generate block statistics
        -type Any text      Display text argument on standard output
        -verbose            Print debug information, more for every -verbose
        -wallet             Scan wallet for addresses to watch
        -wfile filename     CSV file of addresses to watch
      ⟨ RPC options help information 140a ⟩
    EOD
            $help =~ s/^    //gm;
            print($help);
            exit(0);
        }
    ◇
```

Fragment referenced in 108b.

Chapter 8

Bitcoin Confirmation Watcher

This utility queries the status of a transaction and reports changes in the number of confirmations it has received. It can monitor a recent transaction and report new confirmations as they arrive, exiting when a specified number of confirmations (default 6) have been received. The transaction can be specified by its transaction ID and the hash of the block containing it. If the server running Bitcoin Core is configured with "`txindex=1`", the block hash need not be specified.

`confirmation_watch` *transaction_id block_hash*

If you are running `address_watch` on the same machine and have configured it to write a log file, you can specify either the Bitcoin address or the label you've assigned to it, with the transaction ID and block hash retrieved from the log. If, for some screwball reason, a label is the same as a transaction ID, the label takes precedence; we only look for a transaction ID if the specification does not match a label.

`confirmation_watch` *address/label*

8.1 Program plumbing

```
"perl/confirmation_watch.pl" 116≡
    ⟨ Explanatory header for Perl files 145a ⟩

        ⟨ Perl language modes 144a ⟩

        ⟨ RPC configuration variables 139b ⟩

        use LWP;
        use JSON;
        use Text::CSV qw(csv);
        use Getopt::Long qw(GetOptionsFromArray);
        use POSIX qw(strftime);
        use Term::ReadKey;

        use Data::Dumper;
    ◇
```

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

8.2 Command line option processing

"perl/confirmation_watch.pl" 117≡

```
    my $log_file = "⟨AW log file 3e⟩";               # Log file from Address Watch
    my $watch = ⟨CW watch confirmations 3h⟩;          # Watch for confirmations ?
    my $poll_time = ⟨Blockchain poll interval 2c⟩;  # Poll time for watch check
    my $testmode = FALSE;                            # Test on recent blockchain transaction
    my $verbose = ⟨Verbosity level 2b⟩;             # Verbose output ?
    my $confirmed = ⟨CW deem confirmed 3i⟩;          # Number of confirmations required

    my %options = (
        ⟨RPC command line options 139c⟩
        "confirmed=i"   => \$confirmed,
        "help"          => \&showHelp,
        "lfile=s"       => \$log_file,
        "poll=i"        => \$poll_time,
        "testmode"      => \$testmode,
        "type=s"        => sub { print("$_[1]\n"); },
        "verbose+"      => \$verbose,
        "watch"         => \$watch
    );

    processConfiguration();

    GetOptions(
        %options
    ) || die("Command line option error");
◇
```

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.
Uses: `processConfiguration` 135a.

8.3 For test mode, choose transaction from recent block

When `-testmode` is specified, rather than taking the transaction ID and block hash from the command, line we scan recent blocks, starting with the newest, looking for the non-codebase transaction with the fewest inputs and outputs. We'll usually find one with the mininum of one input and output early in the first block we scan. We then use that as the transaction to watch, allowing easy testing without the need to submit an actual transaction or manually dig out a transaction from a block dump. Because we know the hash of the block in which we found the transaction, this works even when the node is configured without a transaction index.

"perl/confirmation_watch.pl" 118a≡

```
        my ($txID, $blockHash);

        if ($testmode) {
            my ($nvin, $nvout, $vtotal, $vaddr);
            my $lastBlock = sendRPCcommand([ "getblockcount" ]);
            while (TRUE) {
                print("Searching block $lastBlock\n") if $verbose >= 2;
                ($txID, $blockHash, $nvin, $nvout, $vtotal, $vaddr) = getRecentTransaction($lastBlock);
                if (defined($txID)) {
                    print("Testing with transaction: $txID\n  Block: $lastBlock\n");
                    print("  Hash:  $blockHash\n") if $blockHash;
                    print("  Sending BTC $vtotal to ");
                    if ($nvout <= 2) {
                        print("$vaddr\n");
                    } else {
                        print("$nvout addresses\n");
                    }
                    last;
                }
                $lastBlock--;
            }
        } else {
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.
Uses: sendRPCcommand 136a.

8.4 Look up address or label in address_watch log

If an address is specified, try looking up in the Address Watch log to find the transaction ID and block hash. We accept either the Bitcoin address or the label the user assigned to it. If a single argument if specified, we have a kludgelet to decide whether it's a label or a transaction ID: if the length is less than 48 characters or it contains a character which isn't a valid hexadecimal digit, we deem it a label, otherwise it's interpreted as a transaction ID.

"perl/confirmation_watch.pl" 118b≡

```
            if (scalar(@ARGV) == 1) {
                my $addr = $ARGV[0];
                if ((length($addr) < 48) || ($addr =~ m/[^\da-f]/i)) {
                    if ($log_file eq "") {
                        print("Cannot look up address or label unless log file (-lfile) specified.\n");
                        exit(2);
                    }
                    my $found = FALSE;
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

If the address has not yet appeared in the address_watch log, continue to poll until it shows up. This allows starting the confirmation watch on a labeled address as soon as (or for that matter before) a transaction is entered, without waiting for its first confirmation to appear on the blockchain.

"perl/confirmation_watch.pl" 119≡

```
    do {
        open(LI, "<$log_file") || die("Cannot open log file $log_file");
        my ($txid, $blockhash);
        while (my $l = <LI>) {
            if (($l =~ m/^"[^"]*",$addr,\S+,\S+\s+\S+,\S+,(\S+),(\S+)/) ||
                ($l =~ m/^"$addr",\S+,\S+,\S+\s+\S+,\S+,(\S+),(\S+)/)
               ) {
                ($txid, $blockhash) = ($1, $2);
                $found = TRUE;
            }
        }
        close(LI);
        if ($watch && (!$found)) {
            print("No transaction for this address found in address_watch log.\n" .
                  "Waiting $poll_time seconds before next check.\n")
                if $verbose;
            sleep($poll_time);
        }
        if ($found) {
            @ARGV = ( $txid, $blockhash );
        }
    } while ($watch && (!$found));
    if (!$found) {
        print("Bitcoin address not found in Address Watch log file.\n");
        exit(1);
    }
} else {
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

If the user has specified just a transaction ID with no block hash, we can only process the request if the Bitcoin node to which we're talking is configured to maintain a transaction index. We verify whether this is the case and reject the request if it isn't. It would be possible to handle the case where a transaction ID has been assigned but the transaction hasn't been mined from the mempool onto the blockchain, waiting for it to show up in a block the same way we do polling for a label or address to appear in the `address_watch` log, but this isn't presently implemented.

"perl/confirmation_watch.pl" 120a≡

```
                    $ARGV[1] = "";
                    if (!(hasTXindex())) {
                        print("No transaction index (txindex=1) on Bitcoin node.\n");
                        print("You must supply the block hash for the transaction.\n");
                        exit(1);
                    }
                }
            } else {
                if (scalar(@ARGV) < 2) {
                    print("usage: confirmation_watch transaction_id block_hash\n");
                    exit(0);
                }
            }

            $txID = $ARGV[0];
            $blockHash = $ARGV[1];
        }
        @ARGV = ( );
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

8.5 Prompt for RPC password

If the "rpc" query method was selected and no password was specified, ask the user for it from standard input.

"perl/confirmation_watch.pl" 120b≡

```
        if (($RPCmethod eq "rpc") && ($RPCpass eq "")) {
            $RPCpass = getPassword("Bitcoin RPC password: ");
        }
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.
Uses: $RPCmethod 139b, $RPCpass 139b, getPassword 135b.

8.6 Retrieve confirmations for transaction

Now retrieve the confirmations for the transaction. If -watch is specified, continue to watch until we've received the -confirm number of confirmations, at which point we exit. On confirmations after the first, we date the confirmation to the time of the last block mined before we received it.

"perl/confirmation_watch.pl" 121≡

```
my $l_confirmations = -1;

do {
    my $query = [ "getrawtransaction", $txID, "true" ];
    if ($blockHash ne "") {
        push(@$query, $blockHash);
    }
    my $txj = sendRPCcommand($query);
    my $tx = decode_json($txj);

    print(Data::Dumper->Dump([$tx], [ qw(Transaction) ])) if $verbose >= 3;

    my $t_confirmations = $tx->{confirmations};
    my $t_time = $tx->{time};

    if ((!$watch) || ($t_confirmations != $l_confirmations)) {
        $l_confirmations = $t_confirmations;
        #   If confirmation count is greater than 1, set time to
        #   that of the most recent block.
        if ($t_confirmations > 1) {
            my $lastBlock = sendRPCcommand([ "getblockcount" ]);
            my $lbStat = sendRPCcommand([ "getblockstats", $lastBlock, '[ "time" ]' ]);
            if ($lbStat) {
                my $lbT = decode_json($lbStat);
                $t_time = $lbT->{time};
            }
        }

        #   Show date and time and number of confirmations
        print(etime($t_time) . "  Confirmations: $t_confirmations\n");

        if (($verbose >= 2) && ($t_confirmations == 1)) {
            #   Number of "vout" items in transaction
            my $t_nvout = scalar(@{$tx->{vout}});
            #   Loop over vout items
            for (my $v = 0; $v < $t_nvout; $v++) {
                if (defined($tx->{vout}->[$v]->{scriptPubKey}) &&
                    defined($tx->{vout}->[$v]->{scriptPubKey}->{addresses})) {
                    my $v_naddr = scalar(@{$tx->{vout}->[$v]->{scriptPubKey}->{addresses}});
                    my $v_value = $tx->{vout}->[$v]->{value};
                    #   Loop over addresses in vout item
                    for (my $a = 0; $a < $v_naddr; $a++) {
                        #   Show destination addresses and amounts
                        my $a_addr = $tx->{vout}->[$v]->{scriptPubKey}->{addresses}->[$a];
                        printf("  => %-42s  %12.8f\n", $a_addr, $v_value);
                    }
                }
            }
        }
    }
```

◇

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.
Uses: etime 132, sendRPCcommand 136a.

8.7 Test for confirmation and wait until next poll

If we've received the specified number of confirmations, exit. If `-watch` is specified, wait until the next poll time and check for new confirmations.

"perl/confirmation_watch.pl" 122a≡

```
            if ($watch && ($l_confirmations < $confirmed)) {
               sleep($poll_time);
            }
        } while ($watch && ($l_confirmations < $confirmed));
◇
```

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

8.8 Utility functions

Define our local functions.

"perl/confirmation_watch.pl" 122b≡

```
        ⟨ getRecentTransaction: Choose recent transaction for test mode 123, ... ⟩
        ⟨ hasTXindex: Test Bitcoin node transaction index support 126a ⟩
        ⟨ showHelp: Show confirmation watch help information 126b ⟩
◇
```

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

Import utility functions we share with other programs.

"perl/confirmation_watch.pl" 122c≡

```
        ⟨ etime: Edit time to ISO 8601 132 ⟩
        ⟨ Command and option processing 133, ... ⟩
        ⟨ sendRPCcommand: Send a Bitcoin RPC/JSON command 136a ⟩
        ⟨ getPassword: Prompt user to enter password 135b ⟩
◇
```

File defined by 116, 117, 118ab, 119, 120ab, 121, 122abc.

8.8.1 `getRecentTransaction` — Choose recent transaction for test mode

When `-testmode` is selected, rather than monitoring a transaction specified on the command line, we pick a transaction from the most recent block (which is guaranteed, therefore, to have just a single confirmation) and monitor it instead. This allows testing without making a transaction or the need to configure the node with `txindex=1` and/or go through a hideously complicated process of searching for a transaction and block hash pair to monitor. We pick a non-coinbase transaction from the block with the fewest inputs and outputs, usually quickly finding one with just one of each. Should we fail for some screwball reason (a miner publishes a block with no transactions just to collect the reward), we return undefined to inform the caller to try again with an older block. The transaction ID and block hash of the selected transaction is returned, along with the number of inputs and outputs, the total value in BTC, and the transaction's output addresses.

Start by retrieving the block specified by the argument.

⟨ *getRecentTransaction: Choose recent transaction for test mode* 123 ⟩ ≡

```
    sub getRecentTransaction {
        my ($blockNo) = @_;

        my $bh = sendRPCcommand([ "getblockhash", "$blockNo" ]);
        print("    Block hash $bh\n") if $verbose;
        my $blk = sendRPCcommand([ "getblock",  $bh, "2" ]);
        my $r = decode_json($blk);

        my $b_hash = $r->{hash};            # Block hash
        my $b_nTx = $r->{nTx};              # Transactions in block

        print("    Block $blockNo " . gmtime($r->{time}) .
              " Transactions $b_nTx\n") if $verbose >= 2;

        my ($vinmin, $voutmin) = (1e20, 1e20);
        my $strans = -1;
```
◇

Fragment defined by 123, 124, 125.
Fragment referenced in 122b.
Uses: `sendRPCcommand` 136a.

Iterate over the transactions in the block, keeping track of the transaction with the minimum number of both inputs and outputs. Coinbase transactions (rewards to miners) are ignored. If we find a transaction with just one input and output, we aren't going to see anything better, we quit immediately at that point choose it.

⟨ *getRecentTransaction: Choose recent transaction for test mode* 124 ⟩ ≡

```
            for (my $t = 0; $t < $b_nTx; $t++) {
                #   Transaction ID
                my $t_txid = $r->{tx}->[$t]->{txid};

                my $t_nvin = scalar(@{$r->{tx}->[$t]->{vin}});
                my $t_nvout = scalar(@{$r->{tx}->[$t]->{vout}});

                if ($verbose >= 3) {
                    my $nvinc = (($t_nvin == 1) &&
                                 (defined($r->{tx}->[$t]->{vin}->[0]->{coinbase}))) ? "coinbase" : $t_nvin;
                    print("  $t.  $t_txid  In: $nvinc  Out: $t_nvout\n");
                }

                #   Ignore coinbase transactions
                if (!(($t_nvin == 1) &&
                      (defined($r->{tx}->[$t]->{vin}->[0]->{coinbase})))) {
                    #   Not coinbase
                    if (($t_nvin == 1) && ($t_nvout == 1)) {
                        #   Found a (1, 1): shortcut escape from search
                        $strans = $t;
                        last;
                    }
                    if (($t_nvin <= $vinmin) && ($t_nvout <= $voutmin)) {
                        $strans = $t;
                        $vinmin = $t_nvin;
                        $voutmin = $t_nvout;
                    }
                }
            }
```

◇

Fragment defined by 123, 124, 125.
Fragment referenced in 122b.

If we found a suitable transaction, compute its value and return. In order to return the value of the transaction in BTC, walk through the `vout` items and sum the values in each. If no suitable transaction was found, return `undef` for all values.

⟨ *getRecentTransaction: Choose recent transaction for test mode* 125 ⟩ ≡

```
        if ($strans >= 0) {
            my $v_total = 0;
            my $v_addr = "";
            #   Loop over vout items to collect addresses and values
            for (my $v = 0; $v < scalar(@{$r->{tx}->[$strans]->{vout}}); $v++) {
                if (defined($r->{tx}->[$strans]->{vout}->[$v]->{scriptPubKey}) &&
                    defined($r->{tx}->[$strans]->{vout}->[$v]->{scriptPubKey}->{addresses})) {
                    my $v_naddr = scalar(@{$r->{tx}->[$strans]->{vout}->[$v]->{scriptPubKey}->{addresses}});
                    my $v_value = $r->{tx}->[$strans]->{vout}->[$v]->{value};
                    #   Loop over addresses in vout item
                    for (my $a = 0; $a < $v_naddr; $a++) {
                        #   Show destination addresses and amounts
                        my $a_addr = $r->{tx}->[$strans]->{vout}->[$v]->{scriptPubKey}->{addresses}->[$a];
                        printf("$strans.$v  To %-42s Value %12.8f\n",
                            $a_addr, $v_value) if $verbose >= 2;
                        $v_total += $v_value;
                        $v_addr .= ($v_addr ? ", " : "") . $a_addr;
                    }
                }
            }
            return ($r->{tx}->[$strans]->{txid}, $r->{hash},
                    scalar(@{$r->{tx}->[$strans]->{vin}}),
                    scalar(@{$r->{tx}->[$strans]->{vout}}), $v_total, $v_addr);
        }
        return (undef) x 6;
    }
```

◇

Fragment defined by 123, 124, 125.
Fragment referenced in 122b.

8.8.2 hasTXindex — Test if Bitcoin node has transaction index

By default, the Bitcoin Core node software maintains only an index for transactions in the user's wallet and unconfirmed "active" transactions. This prevents looking up other transactions by just their transaction ID—the hash of the block containing them must be supplied. The operator of the node can enable an index of all transactions by setting `txindex=1` in the node configuration file. This function tests whether the node to which it is talking maintains a transaction index, and allows callers to adjust their behaviour accordingly.

⟨ *hasTXindex: Test Bitcoin node transaction index support* 126a ⟩ ≡

```
sub hasTXindex {
    my $txindex = FALSE;

    my $ixqs = sendRPCcommand([ "getindexinfo", "txindex" ]);
    if ($ixqs) {
        my $ixq = decode_json($ixqs);
        if ($ixq) {
            $txindex = $ixq->{txindex}->{synced};
        }
    }
    return $txindex;
}
```

◇

Fragment referenced in 122b.
Uses: `sendRPCcommand` 136a.

8.8.3 `showHelp` — Show help information

⟨ *showHelp: Show confirmation watch help information* 126b ⟩ ≡

```
    sub showHelp {
        my $help = <<"    EOD";
perl confirmation_watch.pl [ option... ] transaction\_id/address/label [ block\_hash ]
  Commands and arguments:
    -confirmed n        Confirmations to deem transaction confirmed
    -help               Print this message
    -lfile filename     Log file from address_watch for looking up labels
    -poll n             Poll for new block every n seconds, 0 = never
    -testmode           Test with a randomly chosen recent transaction
    -type Any text      Display text argument on standard output
    -verbose            Print debug information, more for every -verbose
    -watch              Poll waiting for -confirmed confirmations
  ⟨ RPC options help information 140a ⟩
EOD
        $help =~ s/^    //gm;
        print($help);
        exit(0);
    }
```

◇

Fragment referenced in 122b.

Chapter 9

Bitcoin Transaction Fee Watcher

This utility collects data which may be used to plot, analyse, and predict the evolution of Bitcoin transaction fees over time. Data are collected at a specified polling interval and may be displayed on standard output and/or written to a log file in comma-separated value (CSV) format. Both Bitcoin Core's estimated fees and actual fee data from blocks added to the blockchain are reported. No analysis is done—that's up to programs which read and process the log.

9.1 Program plumbing

"perl/fee_watch.pl" 127≡

```
⟨ Explanatory header for Perl files 145a ⟩

    ⟨ Perl language modes 144a ⟩

    ⟨ RPC configuration variables 139b ⟩

    use LWP;
    use JSON;
    use Text::CSV qw(csv);
    use Getopt::Long qw(GetOptionsFromArray);
    use POSIX qw(strftime);
    use Term::ReadKey;

    use Data::Dumper;
◇
```

File defined by 127, 128ab, 129, 130ab, 131a.

9.2 Command line option processing

"perl/fee_watch.pl" 128a≡

```
        my $conf_target = ⟨ CW deem confirmed 3i ⟩;          # Confirmation target in blocks
        my $fee_file = "";                                   # Fee watch log file
        my $poll_time = ⟨ Blockchain poll interval 2c ⟩;    # Poll time for watch check
        my $quiet = FALSE;                                   # Suppress console output
        my $verbose = ⟨ Verbosity level 2b ⟩;              # Verbose output ?

        my %options = (
            ⟨ RPC command line options 139c ⟩
            "confirmed=i"   => \$conf_target,
            "ffile=s"       => \$fee_file,
            "help"          => \&showHelp,
            "poll=i"        => \$poll_time,
            "quiet"         => \$quiet,
            "type=s"        =>  sub { print("$_[1]\n"); },
            "verbose+"      => \$verbose
        );

        processConfiguration();

        GetOptions(
            %options
        ) || die("Command line option error");
    ◇
```

File defined by 127, 128ab, 129, 130ab, 131a.
Uses: `processConfiguration` 135a.

9.3 Prompt for RPC password

If the "`rpc`" query method was selected and no password was specified, ask the user for it from standard input.

"perl/fee_watch.pl" 128b≡

```
        if (($RPCmethod eq "rpc") && ($RPCpass eq "")) {
            $RPCpass = getPassword("Bitcoin RPC password: ");
        }
    ◇
```

File defined by 127, 128ab, 129, 130ab, 131a.
Uses: `$RPCmethod` 139b, `$RPCpass` 139b, `getPassword` 135b.

9.4 Poll fees at the specified interval

We poll for the current fees at each specified interval. This occurs at an even multiple of the interval, not at intervals based upon when the program started. For example, if you set the interval to 10 minutes, polls will be at the top of the hour, 10, 20,... etc. minutes thereafter. In each poll, begin by making an `estimatesmartfee` query, which provides the estimate which the Bitcoin Core wallet recommends for transactions it submits. If logging is enabled, this is logged as a type 1 record.

"perl/fee_watch.pl" 129≡

```
my $block_start = -1;               # Last block processed
my $lastfee = -1;                   # Last estimated fee

while (TRUE) {
    my $t = time();
    my $wait = $poll_time - ($t % $poll_time);
    print("Waiting $wait seconds until next poll.\n") if $verbose;
    sleep($wait);
    $t = time();

    my $efj = sendRPCcommand([ "estimatesmartfee", $conf_target ]);
    my $ef = decode_json($efj);
    print(Data::Dumper->Dump([$ef], [ qw(estimatesmartfee) ])) if $verbose >= 2;

    my $estimatedFee = $ef->{feerate};

    if (!$quiet) {
        my $feediff = "";
        if ($lastfee >= 0) {
            if ($lastfee != $estimatedFee) {
                $feediff = sprintf("  %+.8f  %+.2f%%",
                    $estimatedFee - $lastfee,
                    100 * (($estimatedFee - $lastfee) / $lastfee));
            }
        }
        $lastfee = $estimatedFee;
        printf("%s  Estimated fee %10.8f%s\n", etime($t), $estimatedFee, $feediff);
    }

    if ($fee_file ne "") {
        open(FO, ">>$fee_file");
        print(FO "1,$t," . etime($t) . ",$estimatedFee\n");
    }
```

◇

File defined by 127, 128ab, 129, 130ab, 131a.
Uses: etime 132, sendRPCcommand 136a.

Now query block-level statistics for all blocks which have arrived since the last poll. These are obtained with getblockstats, which provides the minimum, maximum. mean, and median fees paid by transactions in the block, as well as a histogram of fees at the 10, 25, 50, 75, and 90 percentile levels. If logging is enabled, these are logged as a type 2 record.

"perl/fee_watch.pl" 130a≡

```
        my $block_end = sendRPCcommand([ "getblockcount" ]);
        if ($block_start < 0) {
            $block_start = $block_end;
        }

        for (my $j = $block_start; $j <= $block_end; $j++) {
            my $bsj = sendRPCcommand([ "getblockstats", $j ]);
            my $bs = decode_json($bsj);
            print(Data::Dumper->Dump([$bs], [ qw(getblockstats) ])) if $verbose >= 2;
            my $btime = $bs->{time};
            my ($feerate_min, $feerate_mean, $feerate_max) =
                ($bs->{minfeerate}, $bs->{avgfeerate}, $bs->{maxfeerate});
            my @feerate_percentiles = @{$bs->{feerate_percentiles}};

            if (!$quiet) {
                printf("  Block %d  %s\n    Fee rate min %d, mean %d, max %d\n",
                    $j, etime($btime),
                    $feerate_min, $feerate_mean, $feerate_max);
                printf("    Fee percentiles: " .
                    "10%% $feerate_percentiles[0]  25%% $feerate_percentiles[1]  " .
                    "50%% $feerate_percentiles[2]  75%% $feerate_percentiles[3]  " .
                    "90%% $feerate_percentiles[4]\n");
            }
            if ($fee_file ne "") {
                print(FO "2,$btime," . etime($btime) . ",$j," .
                    "$feerate_min,$feerate_mean,$feerate_max," .
                    "$feerate_percentiles[0],$feerate_percentiles[1]," .
                    "$feerate_percentiles[2],$feerate_percentiles[3]," .
                    "$feerate_percentiles[4]\n");
            }
        }
        if ($fee_file ne "") {
            close(FO);
        }
        $block_start = $block_end + 1;
    }
```

◇

File defined by 127, 128ab, 129, 130ab, 131a.
Uses: etime 132, sendRPCcommand 136a.

9.5 Local functions

Define our local functions.

"perl/fee_watch.pl" 130b≡

⟨ *showHelp: Show fee watch help information* 131b ⟩

◇

File defined by 127, 128ab, 129, 130ab, 131a.

9.5.1 Utility functions

Import utility functions we share with other programs.

"perl/fee_watch.pl" 131a≡

```
    ⟨ etime: Edit time to ISO 8601 132 ⟩
    ⟨ Command and option processing 133, ... ⟩
    ⟨ sendRPCcommand: Send a Bitcoin RPC/JSON command 136a ⟩
    ⟨ getPassword: Prompt user to enter password 135b ⟩
◇
```

File defined by 127, 128ab, 129, 130ab, 131a.

9.5.2 showHelp — Show help information

⟨ *showHelp: Show fee watch help information* 131b ⟩ ≡

```
    sub showHelp {
        my $help = <<"    EOD";
perl fee_watch.pl [ option... ]
  Commands and arguments:
    -confirmed n        Confirmations to deem transaction confirmed
    -ffile filename     Log file for fee statistics
    -help               Print this message
    -poll n             Poll for new block every n seconds, 0 = never
    -quiet              Suppress console output
    -type Any text      Display text argument on standard output
    -verbose            Print debug information, more for every -verbose
  ⟨ RPC options help information 140a ⟩
EOD
        $help =~ s/^    //gm;
        print($help);
        exit(0);
    }
◇
```

Fragment referenced in 130b.

Chapter 10

Utility Functions

10.1 `etime` — Edit time to ISO 8601

⟨ *etime: Edit time to ISO 8601* 132 ⟩ ≡

```
sub etime {
    my ($t) = @_;

    return strftime("%F %T", gmtime($t));
}
```

◇

Fragment referenced in 108b, 122c, 131a.
Defines: `etime` 107a, 108a, 113, 121, 129, 130a.

10.2 Command and option processing

These functions provide an integrated way to handle option specifications and commands whether provided as command-line options, from a configuration file, or entered interactively by the user. All of these functions are driven by a hash defining commands and actions in the same form as used by `Getopt::Long`. Any option not defined in the hash will be ignored if in a configuration file (allowing the same file to be used by multiple programs with only some options in common) or reported as an error message if entered interactively.

10.2.1 processCommand — Parse and process command

⟨ *Command and option processing* 133 ⟩ ≡

```
sub processCommand {
    my ($command, $interactive) = @_;

    my ($verb, $noun) = ("", "");
    $command =~ s/\s+$//;
    #   Ignore blank lines and comments
    if (($command ne "") && ($command !~ m/^\s*#/)) {
        $command =~ m/^\s*(\w+)(?:\s+(\S.*?))?\s*$/ ||
            die("Unable to parse command \"$command\"\n");
        ($verb, $noun) = ($1, $2);
        my $inop = TRUE;
        foreach my $op (keys(%options)) {
            $op =~ s/(?:\+|=\w+)$//;
            if ($op eq $verb) {
                $inop = FALSE;
                last;
            }
        }
        if ($inop) {
            if ($interactive) {
                return ("", "") if ($verb =~ m/^(?:en|ex|qu)/);
                print("Unknown command/option \"$verb\".\n");
                return ("?", "");
            } else {
                return ("", "");
            }
        }
        $noun = "" if (!defined($noun));
        my @optarr = ( "-$verb" );
        if ($noun ne "") {
            push(@optarr, $noun);
        }
        if (!GetOptionsFromArray(\@optarr, %options)) {
            if ($interactive) {
                print("Error in command \"$command\".\n");
            }
        }
    }
    return ($verb, $noun);
}
```

◇

Fragment defined by 133, 134ab, 135a.
Fragment referenced in 28, 108b, 122c, 131a.
Defines: `processCommand` 134ab.

10.2.2 arg_inter — Process interactive commands

A utility may process interactive commands from the user by processing the `-inter` option and calling this handler. It prompts the user for commands and arguments and executes them interactively. Interactive mode is exited by any of the commands “`end`”, “`exit`”, or “`quit`”, all of which may be abbreviated to two characters.

⟨ *Command and option processing* 134a ⟩ ≡

```
        my $interactive = FALSE;

        sub arg_inter {
            $interactive = TRUE;
            while (TRUE) {
                print("> ");
                my $l = <> || last;
                chomp($l);
                if ($l !~ m/^\s*$/) {
                    my ($v, $n) = (" ", "");
                    eval {
                        ($v, $n) = processCommand($l, TRUE);
                    };
                    last if ($v eq "");
                }
            }
            $interactive = FALSE;
        }
```

◇

Fragment defined by 133, 134ab, 135a.
Fragment referenced in 28, 108b, 122c, 131a.
Defines: `arg_inter` 27.
Uses: `processCommand` 133.

10.2.3 `processCommandFile` — Process commands from file

Read and execute commands from the file named by the argument. Errors are ignored, allowing a general configuration file to be used for multiple programs, not all of which support options it declares.

⟨ *Command and option processing* 134b ⟩ ≡

```
        sub processCommandFile {
            my ($fname) = @_;

            open(CI, "<$fname") ||
                die("Cannot open command file $fname");

            while (my $l = <CI>) {
                chomp($l);
                my ($v, $n) = processCommand($l, FALSE);
            }
            close(CI);
        }
```

◇

Fragment defined by 133, 134ab, 135a.
Fragment referenced in 28, 108b, 122c, 131a.
Defines: `processCommandFile` 135a.
Uses: `processCommand` 133.

10.2.4 `processConfiguration` — Process program configuration

We first look for a project-wide configuration file and, if present, process it. Then, we look for a configuration file for a specific program, which will be named *program_name*`.conf`; if present, options it sets will override those in the project configuration file. Options in both files can be overridden by those on the command line, which are processed after both configuration files.

⟨ *Command and option processing* 135a ⟩ ≡

```
    sub processConfiguration {
        if (-f "⟨ Project File Name 1c ⟩.conf") {
            processCommandFile("⟨ Project File Name 1c ⟩.conf");
        }
        my $progName = "file name";
        $progName =~ m|^(?:[^/]*/)?(\w+)\.\w+$| ||
            die("Cannot extract program name from $progName");
        $progName = $1;
        if (-f "$progName.conf") {
            processCommandFile("$progName.conf");
        }
    }
```

◇

Fragment defined by 133, 134ab, 135a.
Fragment referenced in 28, 108b, 122c, 131a.
Defines: `processConfiguration` 27, 102, 117, 128a.
Uses: `processCommandFile` 134b.

10.3 `getPassword` — Prompt user to enter password

The user is prompted to enter a password by the message argument, which is output on standard error (in case standard output has been redirected), then the user's input is accepted with echo disabled in the interest of security.

⟨ *getPassword: Prompt user to enter password* 135b ⟩ ≡

```
    sub getPassword {
        my ($prompt) = @_;

        ReadMode("noecho");
        print(STDERR $prompt);
        my $pw = <STDIN>;
        chomp($pw);
        ReadMode("original");
        return $pw;
    }
```

◇

Fragment referenced in 108b, 122c, 131a.
Defines: `getPassword` 104, 120b, 128b.

10.4 `sendRPCcommand` — Send a Bitcoin RPC/JSON command

This function sends a command to the Bitcoin RPC/JSON API and returns its result, or an undefined value in case of error. Its argument is a reference of an array of arguments in precisely the form they would be submitted on the `bitcoin-cli` command line. The query is sent by the means specified by the `$RPCmethod` variable, as follows.

`local` Request via command line `bitcoin-cli` on the local machine, which is a Bitcoin node.

- **`ssh`** Submit request via ssh to `bitcoin-cli` installed on Bitcoin node running on host `$RPChost` at path name `$RPCcli`.
- **`rpc`** Make a direct RPC call to the Bitcoin daemon running on `$RPChost` at port `$RPCport`, logging in as `$RPCuser` with password `$RPCpass`. The Bitcoin daemon on that host must be configured to accept requests from the submitting host and user.

⟨*sendRPCcommand: Send a Bitcoin RPC/JSON command* 136a⟩ ≡

```
sub sendRPCcommand {
    my ($args) = @_;

    my $result;

    if ($RPCmethod eq "local") {
        ⟨Request via bitcoin-cli on the local machine 136b⟩

    } elsif ($RPCmethod eq "ssh") {
        ⟨Request via bitcoin-cli on a remote machine via ssh 137a⟩

     } elsif ($RPCmethod eq "rpc") {
        ⟨Request via direct RPC call 137b, ... ⟩

   } else {
        print(STDERR "Unknown -method configured: \"$RPCmethod\".\n");
        exit(1);
    }

    if (defined($result)) {
        chomp($result);
    }

    return $result;
}
```

◇

Fragment referenced in 108b, 122c, 131a.
Defines: `sendRPCcommand` 105, 108a, 109b, 111, 114b, 118a, 121, 123, 126a, 129, 130a.
Uses: `$RPCmethod` 139b.

10.4.1 Request via `bitcoin-cli` on the local machine

⟨*Request via bitcoin-cli on the local machine* 136b⟩ ≡

```
map({ s/^(\[.*?\])$/'$1'/ } @$args);
my $cmd = join(" ", @$args);
$result = `$RPCcli $cmd 2>&1`;
```

◇

Fragment referenced in 136a.
Uses: `$RPCcli` 139b.

10.4.2 Request via `bitcoin-cli` on a remote machine via `ssh`

⟨ *Request via bitcoin-cli on a remote machine via ssh* 137a ⟩ ≡

```
        map({ s/^(\[.*?\])$/'$1'/ } @$args);
        my $cmd = join(" ", @$args);
        $cmd =~ s/"/\\"/g;
        $result = `ssh $RPChost $RPCcli \"$cmd\" 2>&1`;
```

◇

Fragment referenced in 136a.
Uses: `$RPCcli` 139b, `$RPChost` 139b.

10.4.3 Request via direct RPC call

We first extract the query type (or "method"), which is the first item in the argument list.

⟨ *Request via direct RPC call* 137b ⟩ ≡

```
        my $method = shift(@$args);
```

◇

Fragment defined by 137bc, 138ab, 139a.
Fragment referenced in 136a.

Next, we translate the arguments in the remainder of the list into JSON-encoded values. This isn't as simple as it might seem, since numbers and the reserved words "`true`", "`false`", and "`null`" must not be quoted, while strings must be quoted. We accept string arguments either pre-quoted or as bare strings, to which quotes are added and embedded quotes escaped.

⟨ *Request via direct RPC call* 137c ⟩ ≡

```
        for (my $i = 0; $i < scalar(@$args); $i++) {
            if ($args->[$i] !~ m/^(?:true|false|null|[\d\.]+|".*"|\[.*?\])$/) {
                my $s = $args->[$i];
                $s =~ s/"/\\"/g;
                $args->[$i] = "\"$s\"";
            }
        }
```

◇

Fragment defined by 137bc, 138ab, 139a.
Fragment referenced in 136a.

Assemble the query to be sent to the server. This is encoded as a JSON object containing the method and array of parameters.

⟨*Request via direct RPC call* 138a⟩ ≡

```
    my $params = join(",\n        ", @$args);
    my $request = LWP::UserAgent->new();
    $request->agent("trans_watch");
    #   Specify requester's credentials, including user and password
    $request->credentials("$RPChost:$RPCport", "jsonrpc",
        $RPCuser, $RPCpass);
    #   Compose JSON query to be sent via POST
    my $query = <<"                EOD";
{
"jsonrpc": "1.0",
"id": "trans_watch",
"method": "$method",
"params": [
$params
]
}
EOD
```

◇

Fragment defined by 137bc, 138ab, 139a.
Fragment referenced in 136a.
Uses: `$RPChost` 139b, `$RPCpass` 139b, `$RPCport` 139b, `$RPCuser` 139b.

Build HTTP request. Since we're sending a pure text string via POST rather than a set of key, value pairs, we have to roll our own `HTTP::Request`.

⟨*Request via direct RPC call* 138b⟩ ≡

```
    my $rq = HTTP::Request->new("POST",
                                "http://$RPChost:$RPCport/",
                                [ "Content-Type" => "text/plain" ],
                                $query);
    my $reply = $request->request($rq);
```

◇

Fragment defined by 137bc, 138ab, 139a.
Fragment referenced in 136a.
Uses: `$RPCport` 139b.

If the request succeeded (result code 200), extract the content. Note that unlike the result returned by `bitcoin-cli`, this is wrapped in an outer "`result`" object, from which we must extract the actual content. We further check the error status within the reply, returning `undef` if it is non-null.

⟨*Request via direct RPC call* 139a⟩ ≡

```
        if ($reply->{_rc} == 200) {
            $result = $reply->{_content};
            $result =~ s/^\{"result":(.*?)(,"error":[^\{]+\})$/$1/ ||
                die("Cannot extract RPC result");
            my $errstat = $2;
            if ($errstat !~ m/"error"\s*:\s*null/) {
                $result = undef;
            }
        }
```

◇

Fragment defined by 137bc, 138ab, 139a.
Fragment referenced in 136a.

10.4.4 RPC configuration

Define the variables specifying the RPC configuration.

⟨*RPC configuration variables* 139b⟩ ≡

```
        my $RPCmethod = "⟨ RPC query method 2d ⟩"; # RPC query method: "local", "ssh", "rpc"
        my $RPChost = "⟨ RPC host 2e ⟩";             # Host where bitcoind runs
        my $RPCport = ⟨ RPC port 2f ⟩;               # bitcoind RPC query port (standard 8332)
        my $RPCcli = "⟨ Bitcoin CLI path 2g ⟩";    # Path to run bitcoin-cli
        my $RPCuser = "⟨ RPC user 2h ⟩";             # RPC user name
        my $RPCpass = "⟨ RPC password 2i ⟩";        # RPC password
```

◇

Fragment referenced in 102, 116, 127.
Defines: `$RPCcli` 136b, 137a, 139c, `$RPChost` 137a, 138a, 139c, `$RPCmethod` 104, 120b, 128b, 136a, 139c, `$RPCpass` 104, 120b, 128b, 138a, 139c, `$RPCport` 138ab, 139c, `$RPCuser` 138a, 139c.

Define the command-line options to set the RPC configuration variables.

⟨*RPC command line options* 139c⟩ ≡

```
        "clipath=s"     => \$RPCcli,
        "host=s"        => \$RPChost,
        "method=s"      => \$RPCmethod,
        "rpcpass=s"     => \$RPCpass,
        "port=i"        => \$RPCport,
        "user=s"        => \$RPCuser,
```

◇

Fragment referenced in 102, 117, 128a.
Uses: `$RPCcli` 139b, `$RPChost` 139b, `$RPCmethod` 139b, `$RPCpass` 139b, `$RPCport` 139b, `$RPCuser` 139b.

Define the `-help` output for the RPC configuration options.

⟨ *RPC options help information* 140a ⟩ ≡

```
    Bitcoin API access configuration options:
      -clipath path        Path name to execute bitcoin-cli command line utility
      -host hostname       Host (name or IP address) where Bitcoin Core runs
      -method which        Query method: local, rpc, ssh
      -rpcpass "text"      Bitcoin RPC API password
      -port n              Port for RPC API requests (default ⟨ RPC port 2f ⟩)
      -user userid         User name for requests via ssh◇
```

Fragment referenced in 115, 126b, 131b.

10.5 `blockReward` — Compute reward for mining block

When a miner solves a hash and publishes a block, they receive a reward composed of a fee for the block plus all of the fees for transactions packed into the block. The block reward in Bitcoin, R_b, is computed on a scale which declines with the block number b according to:

$$R_b = \frac{50}{2^{\lfloor (b+1)/210000 \rfloor}}$$

⟨ *blockReward: Compute reward for mining block* 140b ⟩ ≡

```
        sub blockReward {
            my ($b) = @_;

            return 50 / (2 ** int(($b + 1) / 210000));
        }
    ◇
```

Fragment referenced in 108b.
Defines: `blockReward` 113.

10.6 `readHexfile` — Read hexadecimal data from a file

Read a “hexfile” containing hexadecimal data. We ignore everything until we find a line with at least 32 characters of valid hexadecimal data and nothing else. We then read successive lines containing nothing but valid hexadecimal data and white space until encountering a line which doesn’t pass this test or end of file. Returns the hexadecimal data stream with no embedded white space.

⟨ *readHexfile: Read hexadecimal data from a file* 141 ⟩ ≡

```
sub readHexfile {
    my ($fname) = @_;

    my $data = "";
    my $ignore = TRUE;
    my $hex = qr/[\dA-Fa-f]/;

    open(FI, "<$fname") || die("Cannot open $fname");
    while (my $l = <FI>) {
        chomp($l);
        $l =~ s/\s+//g;
        my $isHex = $l =~ m/^$hex+$/;
        if ($ignore) {
            if ($isHex && (length($l) >= 32)) {
                $ignore = FALSE;
            }
        }
        if (!$ignore) {
            if ($isHex) {
                $data .= $l;
            } else {
                last;
            }
        }
    }
    close(FI);
    if (length($data) & 1) {
        die("Number of hexadecimal digits is odd");
    }
    return $data;
}
```
◇

Fragment referenced in 28.
Defines: `readHexfile` 34b.

10.7 Pseudorandom number generator

We use the Mersenne Twister algorithm as a pseudorandom number generator. It is implemented in the Perl module `Math::Random::MT`, which we import and use to create and initialise our generator, `$randGen`.

10.7.1 `randInit` — Initialise pseudorandom generator

Any code which requires the random generator should call `randInit()` before requesting any data. If the generator has not been initialised, a 2496 byte random seed is obtained from non-blocking `Crypt::Random::Seed` and used to initialise a new generator. If the generator has been previously initialised, the call is ignored, so there's no need for application code to check whether a call to `randInit()` is needed.

⟨ *Pseudorandom number generator* 142a ⟩ ≡

```
        use Math::Random::MT;

        my $randGen;                    # Pseudorandom number generator

        sub randInit {
            if (!defined($randGen)) {
                my (@seed, $rbuf);

                my $rgen = Crypt::Random::Seed->new(NonBlocking => 1);
                $rbuf = $rgen->random_bytes(624 * 4);
                @seed = unpack("L4", $rbuf);
                $randGen = Math::Random::MT->new(@seed);
            }
        }
```

◇

Fragment defined by 142ab.
Fragment referenced in 28, 97b.
Defines: `randInit` 39b, 58, 93, 143.

10.7.2 `randNext` — Get next value from pseudorandom generator

The next pseudorandom value is returned by `randNext(`n`)`, where n specifies the range of values returned, in the half-open interval $[0, n)$, that is, $0 \leq r < n$, where r is the random variate returned. Thus, to return the value of a pseudorandom byte, use `randNext(256)`.

⟨ *Pseudorandom number generator* 142b ⟩ ≡

```
        sub randNext {
            my ($n) = @_;

            return $randGen->rand($n);
        }
```

◇

Fragment defined by 142ab.
Fragment referenced in 28, 97b.
Defines: `randNext` 39b, 58, 96, 143.

10.8 `shuffleBytes` — Shuffle bytes in string

Shuffle the bytes in a string using the Fisher-Yates shuffle algorithm. The pseudorandom values for the shuffle are obtained from the Mersenne Twister generator `randNext()`.

⟨ *shuffleBytes: Shuffle bytes* 143 ⟩ ≡

```
sub shuffleBytes {
    my ($in) = @_;

    randInit();
    my @bytes = unpack("C*", $in);

    my $n = scalar(@bytes);
    for (my $i = $n - 1; $i >= 1; $i--) {
        my $j = randNext($i + 1);
        my $temp = $bytes[$j];
        $bytes[$j] = $bytes[$i];
        $bytes[$i] = $temp;
    }

    return pack("C*", @bytes);
}
```

◇

Fragment referenced in 28.
Defines: `shuffleBytes` 43a.
Uses: `randInit` 142a, `randNext` 142b.

Chapter 11

Meta and Miscellaneous

This is a collection of items which are about building the programs and tools used in the process.

11.1 Perl language modes

⟨ *Perl language modes* 144a ⟩ ≡

```
require 5;
use strict;
use warnings;
use utf8;

use constant FALSE => 0;
use constant TRUE => 1;
```

◇

Fragment referenced in 25, 61, 74, 91a, 102, 116, 127, 150.

11.2 Explanatory header for shell-like files

⟨ *Explanatory header for shell-like files* 144b ⟩ ≡

```
#   NOTE: This program was automatically generated by the Nuweb
#   literate programming tool.  It is not intended to be modified
#   directly.  If you wish to modify the code or use it in another
#   project, you should start with the master, which is kept in the
#   file ⟨ Project File Name 1c ⟩.w in the public GitHub repository:
#       https://github.com/Fourmilab/⟨ Project File Name 1c ⟩.git
#   and is documented in the file ⟨ Project File Name 1c ⟩.pdf in the root directory
#   of that repository.
```

◇

Fragment referenced in 145abc.

11.3 Explanatory header for Perl files

This header comment appears at the top of all Perl files generated from this web. It explains where to go for the master source code.

⟨ *Explanatory header for Perl files* 145a ⟩ ≡

```
#! ⟨ Perl directory 4c ⟩

⟨ Explanatory header for shell-like files 144b ⟩
    #
    #   Build ⟨ Build number 1d ⟩  ⟨ Build date and time 2a ⟩
◇
```

Fragment referenced in 25, 61, 74, 91a, 102, 116, 127, 150.

11.4 Explanatory header for Python files

This header comment appears at the top of all Python files generated from this web. It explains where to go for the master source code.

⟨ *Explanatory header for Python files* 145b ⟩ ≡

```
#! ⟨ Python directory 4d ⟩

⟨ Explanatory header for shell-like files 144b ⟩
    #
    #   Build ⟨ Build number 1d ⟩  ⟨ Build date and time 2a ⟩
◇
```

Fragment referenced in 90.

11.5 Makefile

The `Makefile` is used to control all build and maintenance operations. Due to a regrettable episode in the ancient history of Unix, the distinction between hardware tab characters and other white space is significant. Nuweb always uses space characters, which would break `make`, so the `Makefile` incorporates a little trick: after performing a `make build` from the web, if this file has been expanded to `Makefile.mkf` and it is newer than the current `Makefile`, it is processed with `sed` and `unexpand` to restore the tabs as required.

`"Makefile.mkf"` 145c≡

```

⟨ Explanatory header for shell-like files 144b ⟩

PROJECT = ⟨ Project File Name 1c ⟩
VERSION = ⟨ Project Version 1b ⟩

#       Path names for build utilities

NUWEB = nuweb
LATEX = xelatex
PDFVIEW = evince
GNUFIND = find

duh:
        @echo "What'll it be, mate?  build view peek guide geek lint stats clean bl"
◇
```

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.1 Build program files

Rebuild all changed files from the master Nuweb `.w` files. Here is where we perform the dirty trick to convert spaces to tabs so a newly-generated `Makefile` will work.

"Makefile.mkf" 146a≡

```
    build:
            perl tools/build/update_build.pl
            $(NUWEB) -t $(PROJECT).w
            chmod 755 perl/*.pl
            chmod 755 python/*.py
            @if [ \( ! -f Makefile \) -o \( Makefile.mkf -nt Makefile \) ] ; then \
                    echo Makefile.mkf is newer than Makefile ; \
                    sed "s/ \*$$//" Makefile.mkf | unexpand >Makefile ; \
            fi
    ◇
```

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.2 Generate and view PDF document

The `view` target re-generates the master document containing all documentation and code, while `peek` simply views the most-recently-generated document (without checking if it is current). We delete the LaTeX intermediate files so an error in an earlier run which might, for example, have corrupted the table of contents, does not wreck this one.

"Makefile.mkf" 146b≡

```
    pdf:
            rm -f $(PROJECT).log $(PROJECT).toc $(PROJECT).out $(PROJECT).aux
            $(NUWEB) -o -r $(PROJECT).w
            $(LATEX) $(PROJECT).tex
            # We have to re-run Nuweb to incorporate the updated TOC
            $(NUWEB) -o -r $(PROJECT).w
            $(LATEX) $(PROJECT).tex

    view:
            make pdf
            $(PDFVIEW) $(PROJECT).pdf

    peek:
            $(PDFVIEW) $(PROJECT).pdf
    ◇
```

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.3 Generate and view User Guide PDF document

Build the composite document for the program, then process it with `sed` filters which extract the User Guide as a separate LaTeX document, compile it into a PDF, and view it.

"Makefile.mkf" 147a≡

```
gpdf:
        rm -f $(PROJECT)_user_guide.log $(PROJECT)_user_guide.toc \
              $(PROJECT)_user_guide.out $(PROJECT)_user_guide.aux
        $(NUWEB) -o -r $(PROJECT).w
        sed -e '/^\\expunge{begin}{userguide}$$/,/^\\expunge{end}{userguide}$$/d' \
            $(PROJECT).tex | \
            sed -e 's/\\impunge{userguide}//' >$(PROJECT)_user_guide.tex
        $(LATEX) $(PROJECT)_user_guide.tex
        $(LATEX) $(PROJECT)_user_guide.tex

gview:
        make gpdf
        $(PDFVIEW) $(PROJECT)_user_guide.pdf

geek:
        $(PDFVIEW) $(PROJECT)_user_guide.pdf
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.4 Syntax check all Perl programs

All Perl programs in the directory tree are checked with `perl -c`. This requires the GNU `find` utility, which supports the "`-quit`" action that allows us to stop after the first error it detects.

"Makefile.mkf" 147b≡

```
lint:
        @# Uses GNU find extension to quit on first error
        $(GNUFIND) perl tools -type f -name \*.pl -print \
                \( -exec perl -c {} \; -o -quit \)
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.5 Build and syntax check Perl programs

The "`bl`" target is a convenience which causes an error in the build to avoid running the subsequent lint.

"Makefile.mkf" 147c≡

```
bl:
        make --no-print-directory build
        make --no-print-directory lint
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.6 Show statistics of the project

"How's it coming along?" Compute and print statistics about the project at the present time.

"Makefile.mkf" 148a≡

```
stats:
        @echo Build `grep "Build number" build.w | sed 's/[^0-9]//g'` \
                `grep "Build date and time " build.w | \
                sed 's/[^:0-9 \-]//g' | sed 's/^ *//'`
        @echo Web: `wc -l *.w`
        @echo Lines: `find . -type f \( -wholename ./perl/\*.pl \
                -o -wholename ./python/\*.py \) -exec cat {} \; | wc -l`
        @if [ -f $(PROJECT).log ] ; then \
                echo -n "Pages: " ; \
                tail -5 $(PROJECT).log | grep pages | sed 's/[^0-9]//g' ; \
        fi
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.7 Copy development files into distribution directories

"Makefile.mkf" 148b≡

```
dist:
        make bl
        make pdf
        make gpdf
        cp -pv perl/*.pl python/*.py bin
        cp -pv $(PROJECT).pdf $(PROJECT)_user_guide.pdf doc
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.8 Build release archive from distribution directories

"Makefile.mkf" 148c≡

```
release:
        rm -rf $(PROJECT)-$(VERSION)
        tar cfv release_temp.tar \
            *.w Makefile bin doc \
            figures perl/.keep python/.keep tools
        mkdir $(PROJECT)-$(VERSION)
        ( cd $(PROJECT)-$(VERSION) ; tar xfv ../release_temp.tar )
        rm release_temp.tar
        tar cfvz $(PROJECT)-$(VERSION).tar.gz $(PROJECT)-$(VERSION)
        rm -rf $(PROJECT)-$(VERSION)
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.9 Clean up intermediate files from earlier builds

Delete intermediate files from the build process, or all files generated from the web.

"Makefile.mkf" 149a≡

```
clean:
        rm -f nw[0-9]*[0-9] rm *.aux *.log *.out *.pdf *.tex *.toc \
            perl/*.pl python/*.py *.gz bin/*.p[ly] doc/*.pdf
        rm -rf test/test_output

squeaky:
        make clean
        rm -f Makefile.mkf
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.5.10 Regression testing

The main regression test for the stand-alone blockchain utilities resides in the `test` subdirectory but may be run from any directory. It writes its temporary output in a `test/test_output` directory, which it cleans up every time it runs. Tests are run on programs from the `bin` subdirectory where the `dist` target installs them. To avoid confusion, we make sure the most recently built version of one program is the same as the one in that directory and warn if it looks like we're testing an old version.

"Makefile.mkf" 149b≡

```
regress:
        @if cmp -s perl/blockchain_address.pl bin/blockchain_address.pl; [ $$? -ne 0 ] ; \
        then \
            echo "Did you forget to make dist?" ; \
        fi
        test/test.sh

```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

The `regress_update` target updates the reference output which is tested against regression runs from the results of the most recent run. This should only be done when a change is made to the regression test which creates a known difference in the output.

"Makefile.mkf" 149c≡

```
regress_update:
        cp -p test/test_output/test_log.txt test/test_log_expected.txt
```

◇

File defined by 145c, 146ab, 147abc, 148abc, 149abc.

11.6 Build number and date maintenance

This Perl program is run by the `Makefile` every time a "`make build`" is run. It increments the build number and places the current UTC date and time in the `build.w` file which is included here to implement build number consistency checking.

"tools/build/update_build.pl" 150≡

```
⟨ Explanatory header for Perl files 145a ⟩

    ⟨ Perl language modes 144a ⟩

    use POSIX qw(strftime);

    my $bfile = "build.w";              # Build file name

    #   Read current file into string

    open(FI, "<$bfile") || die("Cannot open $bfile");
    my $btext = do {
        local $/ = undef;
        <FI>;
    };
    close(FI);

    #   Update build number and date

    my $date = strftime("%F %H:%M", gmtime(time()));

    $btext =~ m/\@d\s+Build\s+number\s+\@\{(\d+)\@/s;
    my $buildno = $1;
    $buildno++;

    #   Substitute build number and date into file

    $btext =~ s/(\@d\s+Build\s+number\s+\@\{)\d+/$1$buildno/s ||
        die("Cannot substitute build number");
    $btext =~ s/(\@d Build date and time \@\{)[^\@]+/$1$date/s ||
        die("Cannot substitute date");

    #   Write out the updated file

    open(FO, ">$bfile") || die("Cannot open $bfile for writing");
    print(FO $btext);
    close(FO);

    print("Build $buildno $date\n");
◇
```

11.7 Git configuration

The project's source code is managed with Git. This `.gitignore` file excludes all files generated automatically from this master document from version control.

".gitignore" 151≡

```
.gitignore
Makefile
Makefile.mkf
*.aux
*.log
*.out
/*.pdf
*.tex
*.toc
bin/*.pl
bin/*.py
run
test/test_output
tools
*.pl
*.py
*.gz
◇
```

Indices

Three indices are created automatically: an index of file names, an index of macro names, and an index of user-specified identifiers. An index entry includes the name of the entry, where it was defined, and where it was referenced.

12.1 Files

12.2 Macros

12.3 Identifiers

Sections which define identifiers are underlined.

Appendix A

Abbreviations used in this document

⟨ *AW* 157a ⟩ ≡

```
{\tt address\_watch}◇
```

Fragment never referenced.

⟨ *BA* 157b ⟩ ≡

```
{\tt blockchain\_address}◇
```

Fragment never referenced.

⟨ *CC* 157c ⟩ ≡

```
{\tt cold\_comfort}◇
```

Fragment never referenced.

⟨ *CW* 157d ⟩ ≡

```
{\tt confirmation\_watch}◇
```

Fragment never referenced.

⟨ *FW* 157e ⟩ ≡

```
{\tt fee\_watch}◇
```

Fragment never referenced.

⟨ *MK* 157f ⟩ ≡

```
{\tt multi\_key}◇
```

Fragment never referenced.

⟨ *PW* 157g ⟩ ≡

```
{\tt paper\_wallet}◇
```

Fragment never referenced.

⟨ VW 158 ⟩ ≡

```
    {\tt validate\_wallet}◇
```

Fragment never referenced.

Appendix B

Development Log

2021 April 5

Added a `-zero` option to `build_watch_list` to include zero-balance accounts in the watch list. By default, they are excluded.

Added a `-wallet` option in `address_watch` which scans the wallet for "`listunspent`" and adds the addresses with an unspent balance to the watch list. This is re-fetched on each periodic scan of new blocks so that the address list is always current whenever we look at new blocks.

Due to Perl syntactic Hell when attempting to mix explicit arrays and references to arrays, `address_watch` was only reporting the first hit on a watched address in a block. I rewrote the whole mess to use only references, added all the requisite arrows and explicit dereferences and now it appears to work OK.

Fixed some messiness with `-verbose` handling in `address_watch`. It makes more sense now and the output is easier to read.

Added the ability in `confirmation_watch` to specify the RPC password from the keyboard with no echo or piped from standard input. This is handled by a new function, `getPassword(`*prompt*>`)`, which we can use in other cases where passwords are required.

If no `-wfile` was specified to `address_watch` but explicit addresses were specified with `-watch`, an error would be reported. Fixed so there's an error only if no addresses are specified by either mechanism.

2021 April 6

Added support for unlocking and locking wallets in `address_watch` when the user has locked the wallet and `-wallet` is specified. The password for the wallet is read from standard input with echo disabled or may be specified with a command line option which is, of course, in a multi-user environment, hideously insecure.

Rewrote `sendRPCcommand()` to accept its arguments as a reference to a list instead of a string it parses. The original scheme didn't play nice with quoted arguments which contain spaces, as happens when specifying pass phrases for the RPC API and wallets. Other than passing a list instead of a string, nothing has changed.

2021 April 7

Updated `confirmation_watch` to use the new list argument `sendRPCcommand()` function. There is just one call on this function in the entire program.

Integrated all of the programs into a Nuweb Literate Programming web named `bitcoin_tools.w`. This allows eliminating duplication across the various programs and easier maintenance, as well as much improved documentation.

Split the global configuration parameters, which set the default for all settings, into a separate `configuration.w` Nuweb file. These are the settings used when their corresponding command-line option is not specified.

2021 April 8

Completed the transition to Nuweb, breaking up over-long sequences of code and adding documentation where appropriate.

2021 April 9

Brought the project under Git configuration control. The `.gitignore` file excludes everything generated from the `.w` files by Nuweb, including itself.

Added statistics collection and output to `address_watch`. The `-stats` option collects statistics on each block examined and reports the block number, date and time, number of transactions, total transaction size, and for both the transaction size and value of transactions, the minimum, maximum, mean, standard deviation, and total. The `-sfile` option collects the same information and writes it as a record to the file specified. In the statistics file, the date and time is written as a Unix `time()` value.

Re-enabled inclusion of the build number, date, and time in generated files. Since these files are excluded from the Git repository by `.gitignore`, they will not cause unnecessary update transactions.

Began implementation of the stack-oriented version of `blockchain_address`. This will increase the flexibility of generation of addresses by this program. Commands allow fetching seeds from the command line, HotBits, `/dev/random`, or `/dev/urandom`.

2021 April 10

Implemented BIP39 encoding and decoding in `blockchain_address`. The BIP39-encoded phrase is output along with other formats by the `-key` command, and a seed specified by a BIP39 phrase supplied as a string argument may be pushed onto the stack with the `-phrase` command.

Added stack underflow checking to all commands that require arguments from the stack. The `stackCheck(`n`)` function checks if n or more arguments are on the stack and errors if fewer are present.

Implemented a proper `hexToBytes()` function to convert one of our strings of hexadecimal digits into a string of bytes composed of pairs of digits. Naturally, there's a `bytesToHex()` to go the other way.

Implemented an `-xor` command to exclusive-or the two top items on the stack and replace them with the result.

Added a full suite of FORTH-like stack commands: `-drop`, `-dup`, `-over`, `-pick` n, `-roll` n, `-rot`, and `-rrot`.

Added a `-type` *message* command to output to standard output.

2021 April 11

Added code to `confirmation_watch` to dump the transaction if the `-verbose` level is two or above.

If a previous generation of the PDF document failed due to a syntax error in a title appearing in the table of contents, it could torpedo a subsequent run due to the error having been transcribed to the `.toc` document. I added a command to the `view` target in the `Makefile` to delete all of the intermediate files from the last run to avoid this happening.

2021 April 12

Added a `-wif` option to `blockchain_address` to extract the seed from a private key in Wallet Import Format (WIF) and push it on the stack. The key may be in either compressed or uncompressed format.

2021 April 16

Now, it's back to `address_watch` to implement watching of inputs to transactions which come from addresses we're watching. For the application of monitoring cold storage against unauthorised accesses, this is a critical functionality. Doing this requires being able to look up transactions by their transaction ID, and to do this on a Bitcoin Core node means enabling the `txindex=1` mode on the server, which causes it to build an index from transaction index to the block that contains the transaction and, in turn, enable the `getrawtransaction` API call to return a transaction purely from its ID, without needing to know in which block it appears.

Enabling this requires a complete re-scan of the archived blocks back to the Genesis block. At the time I did this yesterday, the complete blockchain was 360 Gb, and performing a complete re-scan and re-index, which you do by starting Bitcoin with:

```
    bitcoin-qt -rescan -reindex
```

This starts a process which ran for around ten and a half hours, using around two CPU cores for the first nine hours, then seven cores for the last hour. When it was done, the transaction index, `indexes/txindex` was 31 Gb and we were able to retrieve transactions by ID.

Now let's start digging into the inputs side of a transaction. Using JSON dumps from a test transaction, we see a "`vin`" section which contains an array of inputs to the transaction. Each of these contains a "`txid`" field with the transaction which last modified the input and a "`vout`" field which identifies which of the output addresses in that transaction is being spent as an input to this one. To learn more about the inputs, we must look up their transaction IDs.

Looking up each input transaction, then, we examine the item in its "`vout`" array specified as the source, and can finally extract the address(es) and value associated with it. (I don't know what it means for more than one address to be present in the "`addresses`" field of a "`vout`".)

With this analysis in place, I added analysis of inbound transactions to `address_watch`. For the cold storage monitoring application, this is arguably the most important, since is somebody has obtained the private key of a cold storage vault address, the first indication will be a transfer to another address with the cold storage as the inbound funds of the thief's transaction. The real world blockchain has a large number of references to the same transactions as inputs within the block, so I implemented a cache mechanism to that we'll only ever look up a transaction once per scan of a block.

2021 April 17

Added "logic" to `confirmation_watch` which allows, when specifying a single argument, it to be either a label looked up in the `address_watch -lfile` log or a transaction ID, which may be specified without the block hash in which it resides if Bitcoin Core has been built with `txindex=1`. This is done with a hideous kludge which considers anything of 48 or fewer characters or which contains a character that is not a hexadecimal digit to be a label. If two arguments are specified, they continue to be interpreted as a transaction index and block hash.

Added a command to the `build` target of the `Makefile` to mark all the Perl programs executable when they are re-generated.

2021 April 18

If a source of funds in a transaction was a "coinbase" transaction: newly-created Bitcoin paid to miners in compensation for publishing a block, that transaction contains no addresses in its "`vout`" section, which caused `scanBlock()` in `address_watch` to fail with a reference an undefined variable. I added code to detect absence of an `address` in source transactions and skip scanning them for matches to one of our watched addresses (since newly-created funds can't possibly have come from a watched address).

2021 April 19

Added a `-help` option to all of the programs. Those which share the common RPC options use a common definition of the options imported into the help text they print.

Added analysis of reward fees paid to miners to address_watch. For each block, the value items in the "vout" section of the first transaction (which is always the "coinbase" reward to the miner who published the block) are summed, giving the total reward. The portion of that which is the current standard reward for a new block, computed by the new common utility function blockReward(), is output and subtracted to yield the portion of the reward due to transaction fees. This information is included in the status shown on standard ouput by the -stats option and in the file written by the -sfile option.

Converted the output of the -sfile command to proper CSV which may be imported directly into a spreadsheet or database.

2021 April 20

Implemented a configuration file and optional interactive command facility. This is driven from the same option array we use to process command line options. The configuration facility is included in each program and uses its %option declaration as a hidden parameter. The function processCommand() parses and executes a command defined in the %options hash, ignoring blank lines and "#" comments. If an undefined command is submitted, a warning is given if running in interactive mode, but ignored otherwise. This allows using a generic configuration file which specifies options that only some of the programs implement.

A ready-to-use arg_inter() argument processor is provided, which can be invoked by a command line option (usually -inter) to interact with the user. Commands and arguments are read and processed until interactive mode is exited with any of "end", "exit", "quit", or end of file.

The processConfiguration() function provides configuration file support. Called before the program processes its command line options, it looks for .conf configuration files named for the project and the specific program and, if present, processes them in order. Program configuration, if present, overrides that for the overall project, and both may be overridden by command line options.

Added a -swap command to blockchain_address, which I had somehow overlooked on the last pass.

Added a -binfile command to blockchain_address, which reads as many sequences of 32 bytes as exist in the file and pushes them, as hexadecimal seed values, onto the stack. In the process, I found and fixed a bug in bytesToHex() that caused it to be sensitive to end of line characters in the byte stream and updated the -random and -urandom commands to use that function rather than their own built-in code.

Made the -type command universal in all programs. This allows it to be used in configuration files for all programs.

Added a -test command to blockchain_address to run a bit-level randomness analysis of the seed on the top of the stack. The top of the stack is unchanged. Removed the built-in randomess test from the key command.

2021 April 21

Implemented a -pseudo command in blockchain_address which places one or more (it respects repeat) pseudorandom seeds generated by a Mersenne Twister generator itself seeded from /dev/urandom on the stack.

Added -zero and -not commands for stack manipulation.

Added a -shuffle command, which shuffles all of the bytes on the stack. All stack items are concatenated together, shuffled as a single byte stream, then divided back into 32 byte seeds and pushed back onto the stack.

2021 April 23

Added a -testall command to blockchain_address which tests the entire contents of the stack for randomness with ent.

2021 April 24

Tested address_watch with an encrypted wallet. It turns out that you only need to unlock the wallet with its encryption key for functions which require private keys, such as sending funds or (duh!) retrieving private keys. Consequently, since the only query we make of the wallet is listunspent, which does not return private keys, there is no need to worry about unlocking and re-locking the wallet for our requests. I commented out the unlock/lock code, keeping it around just in case it should come in handy for some further project which requires unlocked access to the wallet.

2021 April 25

Added a -clear command to blockchain_address to clear the stack. Added a -p command to print the top of the stack, or report "Stack empty."

Added a "bl" target to the Makefile to do a build and then lint, aborting the make if an error occurs in the build.

Added code to force hexadecimal letter digits to upper case when loading seeds with -seed or hexfile in blockchain_address.

Our strategy in address_watch of purging addresses from the wallet containing unspent funds and retrieving a new list before each scan of a block had the consequence of failing to see the transaction which spent the funds in a wallet address. This happens because when using funds from a wallet address, its balance is zeroed out and replaced with a new address containing the change (if any). These changes happen in the wallet when the transaction is broadcast to the mempool, but don't appear in the blockchain until the first confirmation for the transaction arrives. When that happens, however, the sending address will no longer appear in the list returned by listunspent since it has been zeroed out when the transaction was broadcast.

To avoid this happening and, more critically, keep a watch for rogue "looting transactions" where a private key has been compromised and a miscreant enters a transaction to transfer the entire balance of a wallet address to a third party, every time we scan wallet addresses we now save the time of the scan and continue to watch a wallet address after its balance goes to zero for an interval specified by configuration parameter "AW wallet purge interval" which is set by default to 3600 seconds (one hour). As long as transactions do not sit in the mempool longer than this before being confirmed, we'll not miss the spend transaction when it arrives on the blockchain.

Added code to the -seed command in blockchain_address to allow an optional "0x" hexadecimal specifier before the seed.

Added initial support for generation of Ethereum addresses, including the whacky upper and lower case hexadecimal letter checksums used by these addresses. This appears to be working correctly, but is pretty rough in user interface. I'll clean this up once I've done some more functionality testing.

2021 April 26

Added CSV output support for Ethereum address generation, including options to use non-checksummed public addresses and include the complete public key in the CSV file. Fixed some CSV generation bugs common to BTC and ETH.

Improved recovery from errors in blockchain_address interactive mode. Stack underflow no longer bounces you out to the command line.

Added statistics for time since last block, including an exponentially smoothed moving average of time between recent blocks.

2021 April 28

Added code which causes specifying the null string as an RPC password to prompt the user to enter the password from the console. Changed the option to specify the RPC password to -rpcpass.

2021 May 1

Updated the code in `confirmation_watch` which looks up wallet labels and addresses in the `address_watch` log file to parse the new CSV format of that file. Changed the logic to search the log file for all references to the specified address and use the transaction ID and block hash of the most recently added transaction involving that address, not the oldest. Completed major sections of the User Guide for `address_watch` and `confirmation_watch`.

2021 May 2

Removed the median transaction fee from the type 2 block fee statistics output and logged by `fee_watch`. This is not the median fee rate, as I misunderstood it to be, but the median *fee* for transactions in the block, which is useless for our statistical analysis purposes.

Added User Guide section for `fee_watch`, including documenting its log file format.

2021 May 3

Moved all of the Perl files generated from the web to a new `perl` subdirectory and adjusted output file commands and the `Makefile` accordingly.

Created a new `run` subdirectory, which will not be included in the distribution, containing symbolic links to the programs in the `perl` directory and all of the configuration, log, and status files needed to run them locally. This will make it convenient to test and run in production locally without contaminating the development and distribution files with any local configuration information. The `run` directory is excluded from the Git repository.

Renamed the project and all derivative files `blockchain_tools`. This required a little wizardry in the intermediate steps to get the `Makefile` to build the right thing, but once changed all is well.

Renaming the Perl destination directory broke the code which checks for and loads a program-specific configuration file. Fixed to ignore a directory prefix before the program name.

2021 August 14

Added an `-outfile` command to `blockchain_address` to redirect the output of key generation commands to the specified file.

2021 August 15

Added the ability to extract the User Guide from the integrated document for the program. This is controlled by declarations within the LaTeX code in the web file. Code which should not be included in the User Guide is bracketed by statements:

```
    \expunge{begin}{userguide}
    \expunge{end}{userguide}
```

and code which is to be included only in the User Guide is declared with:

```
    \impunge{userguide}{TeX code}
```

A new `guide` target in the `Makefile` generates the complete document, then runs `sed` filters over it to remove everything marked to be expunged and expand the material which appears only in the User Guide, compiles the document, and displays the resulting PDF. The `geek` target just views this PDF.

These targets are intended for the development cycle. For release, a general target to build distribution files will be added and used.

2021 August 16

Renamed the paper wallet generation program `paper_wallet` and the cold storage wallet validator to `validate_wallet` to be consistent with the other programs in the package.

Added the ability to specify the font family, size, and weight to be used to display addresses and keys in `paper_wallet`.

Added a `-separator` option to `paper_wallet` to insert user-defined separators between groups of four characters in paper wallet addresses and keys. The `validate_wallet` program now ignores separators in addresses it validates.

2021 August 23

Added `*.py` files to the list of those not included in the repository via `.gitignore`.

Integrated the `cold_comfort` program into the main web.

Updated the "`make stats`" target to count lines of Python programs as well as Perl.

2021 August 31

The `-wfile` parser in `address_watch` was befuddled by blank lines and comments which we permit in other programs. I added code to ignore them and also to ignore any Ethereum addresses which might be present in a composite cold storage listing. In addition, I corrected the parser to take the balance from the fourth field, after the blank private key field we added for compatibililty with other programs.

2021 September 1

In `cold_comfort`, the expression "`-SATOSHI`", where "`SATOSHI`" is a Perl `use constant` declaration, caused a warning on Juno, which has an older version of Perl (5.16 as opposed to 5.30 on Hayek and Ragnar). I rewrote the expression as "`-(SATOSHI)`" and the warning went away.

2021 September 2

In `cold_comfort`, changed the handling of API failures so that the message appears in the warning/error field of the report instead of the current balance. This triggers display of the address item when `-verbose` is not set.

2021 September 3

In `blockchain_address`, added an option to the `-format` command, "`k`", which causes the `-btc` and `-eth` commands to preserve the keys on the stack from which they generate addresses. This allows generating key/address pairs multiple times in various formats using the same source keys. This can be cancelled by entering a new `-format` without the option.

Added a "`b`" (Bowdlerise) option to the `-format` command in `blockchain_address` which causes the private key to be excluded from `CSV` format output, allowing it to be used with utilities such as `cold_comfort` and `address_watch` without compromising security.

2021 September 4

Replaced our own function to shuffle the order of items in an array with the `List::Util shuffle()` function in `cold_comfort` and `multi_key`. The shuffling of bytes on the stack performed by the `blockchain_address`'s `-shuffle` command continues to be done by a custom function which uses the higher-quality Mersenne Twister pseudorandom generator in the interest of security.

Integrated `multi_key` into the main web.

2021 September 5

Replaced hard-coded in-line accesses to `/dev/random` and `/dev/urandom` with calls to the Perl module `Crypt::Random::Seed`, which provides access to these facilities on systems that support them and alternatives on others which do not. If no strong generator is available, the `-random` command in `blockchain_address` will report an error and do nothing.

Added `-help` option support to `multi_key` and `paper_wallet`.

2021 September 6

Made the `-dump` command in `blockchain_address` write its output to a file if `-outfile` has diverted it from the console. The output is written in such a format and order that `-hexfile` will reload it onto the stack.

Modified the `-help` command in `blockchain_address` so it doesn't exit if invoked in interactive (`-inter`) mode.

Added a `-bindump` command to `blockchain_address` which writes the entire contents of the stack in binary to a specified file. Such files can be loaded back on the stack with the `-binfile` command.

Replaced all `\ref{LBL}` references with `\hyperref[LBL]{}` wrapped around the referenced text.

2021 September 7

Fixed page numbering in the PDF, where the last two pages of the table of contents had Arabic rather than Roman numbers.

Modified handling of `-inter` mode in `blockchain_address` so a blank line does not terminate interactive mode, but is simply ignored.

Added a "Project Version" declaration in the Introduction and used it as the version number in the document and distribution archive.

Began the process of building distributions for release in the `Makefile`. Added a `dist` target which copies the generated Perl and Python files to a `bin` directory, where there are pre-created symbolic links to invoke the programs without the file type extension, and to copy the PDF documentation to a `doc` directory. A new `release` target builds a Gzipped tar archive containing the web files, the `Makefile`, and the `bin`, `doc`, `figures`, and `tools` directories. I have yet to see if this is sufficient to rebuild everything from bare metal.

2021 September 8

Added README.md, LICENSE.md, and CONTRIBUTING.md to the main archive in preparation for publication on GitHub.

2021 September 9

Renamed the `blockchain_address` command `-sha256` to `-sha2` to make room for support of SHA3.

Modified the `-sha2` command in `blockchain_address` to respond to the `-repeat` setting. It now computes the 256 bit digest of the concatenation of the specified number of items, top to bottom being concatenated left to right, and replaces them with the digest.

Added a `-sha3` command which computes SHA-3 digests in the same manner as `-sha2`.

Rationalised the handling of AES encryption and decryption in `blockchain_address`. The main problem in providing a useful encryption and decryption facility is the requirement that all of the objects upon which `blockchain_address` operates be 256 bit quantities. Typical use of AES encryption in, for example, cipher-block-chaining mode, is not length-preserving: the encrypted text is longer than the plaintext, with a header containing the (usually random) initial vector used to encrypt the it. This complete message is required, then, to decrypt the shorter plaintext. But since things we might want to encrypt are 256 bits and we can only pass quantities of that length along our pipelines, there's a problem.

But consider what this might be used for in our application: almost always encrypting private keys so that they may be stored separately from the encryption key in the interest of security. (Of course, for most such applications, splitting keys into parts with `multi_key` is a better solution.) The main rationale for a random initial vector is to avoid known plaintext attacks. But the private keys that people will be encryption will be near-maximal entropy random or pseudorandom bit strings (unless the user is doing something stupid which, of course, with a toolbox utility like this, they are entitled to do). Consequently, the additional security provided by a random initial vector is not as important as for potentially low entropy text. To maintain a secure form of encryption and not increase message length, we synthesise an initial vector from the encryption key, forming its 256 bit SHA2 digest, then using the first 16 bytes of this as the initial vector. Used with the cipher feedback mode and no header, the encrypted data remain 256 bits long. To decrypt it with the key, we rebuild the initial vector from the key, then pass it to decryption.

Encryption is now done by an `-aesenc` command with decryption performed by `aesdec`, both of which take the encryption key from the top of the stack and the plaintext or codetext from the second item and place the result back on the stack.

Building the initial vector from the key may not have the crystalline purity of a purely random initial vector, but it not only preserves message length by allowing us to dispense with a header, it also has the salutary effect of making the codetext produced by encrypting data with a key deterministic, which allows it to be easily checked in regression tests.

Modified the `-test` command in `blockchain_address` to respect the `-repeat` setting, allowing any number of items to be tested. The `-testall` command is thus no longer strictly necessary, but left in as a convenience so you don't have to fiddle with the repeat setting when loading keys from a binary or hexadecimal file, for example.

Modified `blockchain_address`'s -tt shuffle command to respect the `-repeat` setting instead of blindly shuffling the entire contents of the stack. This also provides the useful capability to shuffle just the bytes of the top item on the stack.

Fixed the flaw that caused `blockchain_address` to blow out to the command line when something was entered that `processCommand()` could not parse.

Added a `-pseudoseed` command to `blockchain_address` to facilitate regression testing. It seeds the pseudorandom number generator with up 78 seeds with the number set by `-repeat` (additional values are ignored and left on the stack), representing the maximum of 624 32-bit state values used by Mersenne Twister.

2021 September 10

Added a regression test in a new subdirectory `test/test.sh`. This tests all of the stand-alone blockchain utilities: `blockchain_address`, `cold_comfort`, `multi_key`, `paper_wallet`, `validate_wallet`. The Bitcoin node monitoring utilities are not tested, as configuring access to a full Bitcoin node is more complicated to arrange. The test may be run from a new `Makefile` target, "`regress`". When the test is run, it compares the output to a reference file, `test/test_log_expected.txt`. If discrepancies are found, the `diff` is shown and an exit status of 1 is returned, which will cause `make` to report the failure. If a change to the test or the environment in which it runs requires the reference output to be updated, the make target `regress_update` will copy the most recent test run output to the reference master.

Added decoding of Bitcoin mini private keys to `blockchain_address` with the `-minikey` command. Both the standard 30 character format and legacy 22 character form used by Series 1 Casascius physical Bitcoin tokens are accepted.

Added generation of Bitcoin mini private keys to `blockchain_address` with the `-minigen` command. Due to the screwball way mini keys are "found", this command combines generation of seeds with creation of keys and does not take seeds off the stack. Only the current standard 30 character mini keys are produced; the (even more) insecure legacy 22 character format is not supported.

Added tests of `-minikey` to the regression test.

To aid in regression testing where we want to test things which are usually non-deterministic, I added a `-testmode` command to `blockchain_address`. This sets a bit-coded value, initially zero, to select test modes. The only bit currently used is 1, which causes `-minigen` to not mix input from the local fast entropy source into the key, and thus behave deterministically and repeatably.

Added some "big hack attack" code to the Makefile `regress` target to check for the all-too-frequent omission of running "`make dist`" before running the regression test, and thus using an older version of the code. We just check one of the Perl programs, counting on the discrepancy in build number being the same in all generated code.

2021 September 11

Added a `-testmode 4` bit to print all of the modules loaded by `blockchain_address` by the time it finally exits.

Added support for multi-part keys in `paper_wallet`. If the key file has an initial record with the special label "`-1`", the multi-part parameters are parsed and used to label the printed pages for that part with a heading of "Part *i* of *n* (*k* needed)".

Added tests for creation of paper wallets from multi-part keys to the regression test.

2021 September 12

Extended the `sendRPCcommand()` function used by all the utilities that interact with a Bitcoin node to accept RPC arguments which are JSON lists. These are tricky and messy to handle, as each of our access methods: `local`, `rpc`, and `ssh` require different quoting of the argument to make it through correctly to the JSON request received by the API.

Fixed `confirmation_watch` so that when it displays the time a confirmation was received, it shows the time of the most recent block at the time the confirmation arrived. Previously, it always showed the time of the block containing the transaction when it first appeared on the blockchain.

Logged on to github.com.

Created a new `blockchain_tools` repository with access URLs:

HTTPS `https://github.com/Fourmilab/blockchain_tools.git`
SSH `git@github.com:Fourmilab/blockchain_tools.git`

Linked the local repository to the GitHub archive:

```
git remote add origin git@github.com:Fourmilab/blockchain_tools.git
```

Confirmed that my local "`git sync`" command works with the remote repository.

The documents in the repository root now work properly.

2021 September 13

Implemented the `-testmode` option in `confirmation_watch` to allow easy testing without the need to submit a transaction or find one in a recent block. When set, `confirmation_watch` scans the most recent block (and more, if necessary) to find a transaction with a single confirmation and uses its transaction ID and block hash instead of requiring they be specified from the command line or the `address_watch` log file. This is useless for normal user cases, but it makes testing much simpler.

Appendix C

To do

Accumulate value in to address as well as value out in `address_watch` and report in statistics.

Options which request passwords prompt the user interactively if given a blank argument.