```
BIP: 39
Layer: Applications
Title: Mnemonic code for generating deterministic keys
Author: Marek Palatinus <slush@satoshilabs.com>
        Pavol Rusnak <stick@satoshilabs.com>
        Aaron Voisine <voisine@gmail.com>
        Sean Bowe <ewillbefull@gmail.com>
Comments-Summary: Unanimously Discourage for implementation
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0039
Status: Proposed
Type: Standards Track
Created: 2013-09-10
```

Abstract

This BIP describes the implementation of a mnemonic code or mnemonic sentence -- a group of easy to remember words -- for the generation of deterministic wallets.

It consists of two parts: generating the mnemonic and converting it into a binary seed. This seed can be later used to generate deterministic wallets using BIP-0032 or similar methods.

Motivation

A mnemonic code or sentence is superior for human interaction compared to the handling of raw binary or hexadecimal representations of a wallet seed. The sentence could be written on paper or spoken over the telephone.

This guide is meant to be a way to transport computer-generated randomness with a human-readable transcription. It's not a way to process user-created sentences (also known as brainwallets) into a wallet seed.

Generating the mnemonic

The mnemonic must encode entropy in a multiple of 32 bits. With more entropy security is improved but the sentence length increases. We refer to the initial entropy length as ENT. The allowed size of ENT is 128-256 bits.

First, an initial entropy of ENT bits is generated. A checksum is generated by taking the first `ENT / 32` bits of its SHA256 hash. This checksum is appended to the end of the initial entropy. Next, these concatenated bits are split into groups of 11 bits, each encoding a number from 0-2047, serving as an index into a wordlist. Finally, we convert these numbers into words and use the joined words as a mnemonic sentence.

The following table describes the relation between the initial entropy length (ENT), the checksum length (CS), and the length of the generated mnemonic sentence (MS) in words.

```
CS = ENT / 32
MS = (ENT + CS) / 11
```

ENT	CS	ENT+CS	MS
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

Wordlist

An ideal wordlist has the following characteristics:

a) smart selection of words

```
  - the wordlist is created in such a way that it's enough to type the first four
letters to unambiguously identify the word
```

b) similar words avoided

```
  - word pairs like "build" and "built", "woman" and "women", or "quick" and "quickly"
not only make remembering the sentence difficult but are also more error
prone and more difficult to guess
```

c) sorted wordlists

```
  - the wordlist is sorted which allows for more efficient lookup of the code words
(i.e. implementations can use binary search instead of linear search)
- this also allows trie (a prefix tree) to be used, e.g. for better compression
```

The wordlist can contain native characters, but they must be encoded in UTF-8 using Normalization Form Compatibility Decomposition (NFKD).

From mnemonic to seed

A user may decide to protect their mnemonic with a passphrase. If a passphrase is not present, an empty string "" is used instead.

To create a binary seed from the mnemonic, we use the PBKDF2 function with a mnemonic sentence (in UTF-8 NFKD) used as the password and the string "mnemonic" + passphrase (again in UTF-8 NFKD) used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits (= 64 bytes).

This seed can be later used to generate deterministic wallets using BIP-0032 or similar methods.

The conversion of the mnemonic sentence to a binary seed is completely independent from generating the sentence. This results in a rather simple code; there are no constraints on sentence structure and clients are free to implement their own wordlists or even whole sentence generators, allowing for flexibility in wordlists for typo detection or other purposes.

Although using a mnemonic not generated by the algorithm described in "Generating the mnemonic" section is possible, this is not advised and software must compute a checksum for the mnemonic sentence using a wordlist and issue a warning if it is invalid.

The described method also provides plausible deniability, because every passphrase generates a valid seed (and thus a deterministic wallet) but only the correct one will make the desired wallet available.

Wordlists

Since the vast majority of BIP39 wallets supports only the English wordlist, it is **strongly discouraged** to use non-English wordlists for generating the mnemonic sentences.

If you still feel your application really needs to use a localized wordlist, use one of the following instead of inventing your own.

- Wordlists

Test vectors

The test vectors include input entropy, mnemonic and seed. The passphrase "TREZOR" is used for all vectors.

https://github.com/trezor/python-mnemonic/blob/master/vectors.json

Also see https://github.com/bip32JP/bip32JP.github.io/blob/master/test_JP_BIP39.json

(Japanese wordlist test with heavily normalized symbols as passphrase)

Reference Implementation

Reference implementation including wordlists is available from

http://github.com/trezor/python-mnemonic

Other Implementations

Go:

- https://github.com/tyler-smith/go-bip39

Python:

- https://github.com/meherett/python-hdwallet

Elixir:

- https://github.com/aerosol/mnemo

Objective-C:

- https://github.com/nybex/NYMnemonic

Haskell:

- https://github.com/haskoin/haskoin

.NET (Standard):

- https://www.nuget.org/packages/dotnetstandard-bip39/

.NET C# (PCL):

- https://github.com/Thashiznets/BIP39.NET

.NET C# (PCL):

- https://github.com/NicolasDorier/NBitcoin

JavaScript:

- https://github.com/bitpay/bitcore/tree/master/packages/bitcore-mnemonic
- https://github.com/bitcoinjs/bip39 (used by blockchain.info)
- https://github.com/dashhive/DashPhrase.js
- https://github.com/hujiulong/web-bip39

Java:

- https://github.com/bitcoinj/bitcoinj/blob/master/core/src/main/java/org/bitcoinj/crypto/MnemonicCode.java

Ruby:

- https://github.com/sreekanthgs/bip_mnemonic

Rust:

- https://github.com/maciejhirsz/tiny-bip39/
- https://github.com/koushiro/bip0039-rs

Smalltalk:

- https://github.com/eMaringolo/pharo-bip39mnemonic

Swift:

- https://github.com/CikeQiu/CKMnemonic
- https://github.com/yuzushioh/WalletKit
- https://github.com/pengpengliu/BIP39
- https://github.com/matter-labs/web3swift/blob/develop/Sources/web3swift/KeystoreManager/BIP39.swift

- https://github.com/zcash-hackworks/MnemonicSwift
- https://github.com/ShenghaiWang/BIP39
- https://github.com/anquii/BIP39

C++:

- https://github.com/libbitcoin/libbitcoin-system/blob/master/include/bitcoin/system/wallet/mnemonic.hpp

C (with Python/Java/Javascript bindings):

- https://github.com/ElementsProject/libwally-core

Python:

- https://github.com/scgbckbone/btc-hd-wallet

Dart:

- https://github.com/dart-bitcoin/bip39