```
BIP: 173
Layer: Applications
Title: Base32 address format for native v0-16 witness outputs
Author: Pieter Wuille <pieter.wuille@gmail.com>
        Greg Maxwell <greg@xiph.org>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0173
Status: Final
Type: Informational
Created: 2017-03-20
License: BSD-2-Clause
Replaces: 142
```

Table of Contents

Introduction

Abstract

This document proposes a checksummed base32 format, "Bech32", and a standard for native segregated witness output addresses using it.

Copyright

Motivation

For most of its history, Bitcoin has relied on base58 addresses with a truncated double-SHA256 checksum. They were part of the original software and their scope was extended in BIP13 for Pay-to-script-hash (P2SH). However, both the character set and the checksum algorithm have limitations:

- Base58 needs a lot of space in QR codes, as it cannot use the *alphanumeric mode*.
- The mixed case in base58 makes it inconvenient to reliably write down, type on mobile keyboards, or read out loud.
- The double SHA256 checksum is slow and has no error-detection guarantees.
- Most of the research on error-detecting codes only applies to character-set sizes that are a prime power, which 58 is not.
- Base58 decoding is complicated and relatively slow.

Included in the Segregated Witness proposal are a new class of outputs (witness programs, see BIP141), and two instances of it ("P2WPKH" and "P2WSH", see BIP143). Their functionality is available indirectly to older clients by embedding in P2SH outputs, but for optimal efficiency and security it is best to use it directly. In this document we propose a new address format for native witness outputs (current and future versions).

This replaces BIP142, and was previously discussed here (summarized here).

Examples

All examples use public key `0279BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798`. The P2WSH examples use `key OP_CHECKSIG` as script.

- Mainnet P2WPKH: `bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4`
- Testnet P2WPKH: `tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx`

- Mainnet P2WSH: `bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3`
- Testnet P2WSH: `tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7`

Specification

We first describe the general checksummed base32[1] format called *Bech32* and then define Segregated Witness addresses using it.

Bech32

A Bech32[2] string is at most 90 characters long and consists of:

- The **human-readable part**, which is intended to convey the type of data, or anything else that is relevant to the reader. This part MUST contain 1 to 83 US-ASCII characters, with each character having a value in the range [33-126]. HRP validity may be further restricted by specific applications.
- The **separator**, which is always "1". In case "1" is allowed inside the human-readable part, the last one in the string is the separator[3].
- The **data part**, which is at least 6 characters long and only consists of alphanumeric characters excluding "1", "b", "i", and "o"[4].

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
+0	q	p	z	r	y	9	x	8
+8	g	f	2	t	v	d	w	0
+16	s	3	j	n	5	4	k	h
+24	c	e	6	m	u	a	7	l

Checksum

The last six characters of the data part form a checksum and contain no information. Valid strings MUST pass the criteria for validity specified by the Python3 code snippet below. The function `bech32_verify_checksum` must return true when its arguments are:

- `hrp` : the human-readable part as a string

- `data` : the data part as a list of integers representing the characters after conversion using the table above

```
def bech32_polymod(values):
  GEN = [0x3b6a57b2, 0x26508e6d, 0x1ea119fa, 0x3d4233dd, 0x2a1462b3]
  chk = 1
  for v in values:
    b = (chk >> 25)
    chk = (chk & 0x1ffffff) << 5 ^ v
    for i in range(5):
      chk ^= GEN[i] if ((b >> i) & 1) else 0
  return chk

def bech32_hrp_expand(s):
  return [ord(x) >> 5 for x in s] + [0] + [ord(x) & 31 for x in s]

def bech32_verify_checksum(hrp, data):
  return bech32_polymod(bech32_hrp_expand(hrp) + data) == 1
```

This implements a BCH code that guarantees detection of **any error affecting at most 4 characters** and has less than a 1 in 10^9 chance of failing to detect more errors. More details about the properties can be found in the Checksum Design appendix. The human-readable part is processed by first feeding the higher bits of each character's US-ASCII value into the checksum calculation followed by a zero and then the lower bits of each[5].

To construct a valid checksum given the human-readable part and (non-checksum) values of the data-part characters, the code below can be used:

```
def bech32_create_checksum(hrp, data):
  values = bech32_hrp_expand(hrp) + data
  polymod = bech32_polymod(values + [0,0,0,0,0,0]) ^ 1
  return [(polymod >> 5 * (5 - i)) & 31 for i in range(6)]
```

Error correction

One of the properties of these BCH codes is that they can be used for error correction. An unfortunate side effect of error correction is that it erodes error detection: correction changes invalid inputs into valid inputs, but if more than a few errors were made then the valid input may not be the correct input. Use of an incorrect but valid input can cause funds to be lost irrecoverably. Because of this, implementations SHOULD NOT implement correction beyond potentially suggesting to the user where in the string an error might be found, without suggesting the correction to make.

Uppercase/lowercase

The lowercase form is used when determining a character's value for checksum purposes.

Encoders MUST always output an all lowercase Bech32 string. If an uppercase version of the encoding result is desired, (e.g.- for presentation purposes, or QR code use), then an uppercasing procedure can be performed external to the encoding process.

Decoders MUST NOT accept strings where some characters are uppercase and some are lowercase (such strings are referred to as mixed case strings).

For presentation, lowercase is usually preferable, but inside QR codes uppercase SHOULD be used, as those permit the use of *alphanumeric mode*, which is 45% more compact than the normal *byte mode*.

Segwit address format

A segwit address[6] is a Bech32 encoding of:

- The human-readable part "bc"[7] for mainnet, and "tb"[8] for testnet.
- The data-part values:
    - 1 character (representing 5 bits of data): the witness version
    - A conversion of the 2-to-40-byte witness program (as defined by BIP141) to base32:
        - Start with the bits of the witness program, most significant bit per byte first.
        - Re-arrange those bits into groups of 5, and pad with zeroes at the end if needed.
        - Translate those bits to characters using the table above.

Decoding

Software interpreting a segwit address:

- MUST verify that the human-readable part is "bc" for mainnet and "tb" for testnet.
- MUST verify that the first decoded data value (the witness version) is between 0 and 16, inclusive.
- Convert the rest of the data to bytes:
    - Translate the values to 5 bits, most significant bit first.
    - Re-arrange those bits into groups of 8 bits. Any incomplete group at the end MUST be 4 bits or less, MUST be all zeroes, and is discarded.
    - There MUST be between 2 and 40 groups, which are interpreted as the bytes of the witness program.

Decoders SHOULD enforce known-length restrictions on witness programs. For example, BIP141 specifies *If the version byte is 0, but the witness* program is neither 20 nor 32 bytes, the script must fail.

As a result of the previous rules, addresses are always between 14 and 74 characters long, and their length modulo 8 cannot be 0, 3, or 5. Version 0 witness addresses are always 42 or 62 characters, but implementations MUST allow the use of any version.

Implementations should take special care when converting the address to a scriptPubkey, where witness version *n* is stored as *OP_n*. OP_0 is encoded as 0x00, but OP_1 through OP_16 are encoded as 0x51 though 0x60 (81 to 96 in decimal). If a bech32 address is converted to an incorrect scriptPubKey the result will likely be either unspendable or insecure.

Compatibility

Only new software will be able to use these addresses, and only for receivers with segwit-enabled new software. In all other cases, P2SH or P2PKH addresses can be used.

Rationale

1. ^ **Why use base32 at all?** The lack of mixed case makes it more efficient to read out loud or to put into QR codes. It does come with a 15% length increase, but that does not matter when copy-pasting addresses.
2. ^ **Why call it Bech32?** "Bech" contains the characters BCH (the error detection algorithm used) and sounds a bit like "base".
3. ^ **Why include a separator in addresses?** That way the human-readable part is unambiguously separated from the data part, avoiding potential collisions with other

human-readable parts that share a prefix. It also allows us to avoid having character-set restrictions on the human-readable part. The separator is *1* because using a non-alphanumeric character would complicate copy-pasting of addresses (with no double-click selection in several applications). Therefore an alphanumeric character outside the normal character set was chosen.

4. **^ Why not use an existing character set like RFC3548 or z-base-32?** The character set is chosen to minimize ambiguity according to this visual similarity data, and the ordering is chosen to minimize the number of pairs of similar characters (according to the same data) that differ in more than 1 bit. As the checksum is chosen to maximize detection capabilities for low numbers of bit errors, this choice improves its performance under some error models.
5. **^ Why are the high bits of the human-readable part processed first?** This results in the actually checksummed data being *[high] 0 [low] [data]*. This means that under the assumption that errors to the human readable part only change the low 5 bits (like changing an alphabetical character into another), errors are restricted to the *[low] [data]* part, which is at most 89 characters, and thus all error detection properties (see appendix) remain applicable.
6. **^ Why not make an address format that is generic for all scriptPubKeys?** That would lead to confusion about addresses for existing scriptPubKey types. Furthermore, if addresses that do not have a one-to-one mapping with scriptPubKeys (such as ECDH-based addresses) are ever introduced, having a fully generic old address type available would permit reinterpreting the resulting scriptPubKeys using the old address format, with lost funds as a result if bitcoins are sent to them.
7. **^ Why use 'bc' as human-readable part and not 'btc'?** 'bc' is shorter.
8. **^ Why use 'tb' as human-readable part for testnet?** It was chosen to be of the same length as the mainnet counterpart (to simplify implementations' assumptions about lengths), but still be visually distinct.

Reference implementations

- Reference encoder and decoder:
  - For C
  - For C++
  - For JavaScript
  - For Go
  - For Python

    - For Haskell
    - For Ruby
    - For Rust
- Fancy decoder that localizes errors:
    - For JavaScript (demo website)

Registered Human-readable Prefixes

SatoshiLabs maintains a full list of registered human-readable parts for other cryptocurrencies:

SLIP-0173 : Registered human-readable parts for BIP-0173

Appendices

Test vectors

The following strings are valid Bech32:

- `A12UEL5L`
- `a12uel5l`
- `an83characterlonghumanreadablepartthatcontainsthenumber1andtheexcludedcharactersbio1tt5tgs`
- `abcdef1qpzry9x8gf2tvdw0s3jn54khce6mua7lmqqqxw`
- `11qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqc8247j`
- `split1checkupstagehandshakeupstreamerranterredcaperred2y9e3w`
- `?1ezyfcl` WARNING: During conversion to US-ASCII some encoders may set unmappable characters to a valid US-ASCII character, such as '?'. For example:

```
>>> bech32_encode('\x80'.encode('ascii', 'replace').decode('ascii'), [])
'?1ezyfcl'
```

The following string are not valid Bech32 (with reason for invalidity):

- 0x20 + `1nwldj5` : HRP character out of range
- 0x7F + `1axkwrx` : HRP character out of range

- 0x80 + `1eym55h` : HRP character out of range
- `an84characterslonghumanreadablepartthatcontainsthenumber1andtheexcludedcharactersbio1569pvx` : overall max length exceeded
- `pzry9x0s0muk` : No separator character
- `1pzry9x0s0muk` : Empty HRP
- `x1b4n0q5v` : Invalid data character
- `li1dgmt3` : Too short checksum
- `de1lg7wt` + 0xFF: Invalid character in checksum
- `A1G7SGD8` : checksum calculated with uppercase form of HRP
- `10a06t8` : empty HRP
- `1qzzfhee` : empty HRP

The following list gives valid segwit addresses and the scriptPubKey that they translate to in hex.

- `BC1QW508D6QEJXTDG4Y5R3ZARVARY0C5XW7KV8F3T4` : `0014751e76e8199196d454941c45d1b3a323f1433bd6`
- `tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7` : `00201863143c14c5166804bd19203356da136c985678cd4d27a1b8c6329604903262`
- `bc1pw508d6qejxtdg4y5r3zarvary0c5xw7kw508d6qejxtdg4y5r3zarvary0c5xw7k7grplx` : `5128751e76e8199196d454941c45d1b3a323f1433bd6751e76e8199196d454941c45d1b3a323f1433bd6`
- `BC1SW50QA3JX3S` : `6002751e`
- `bc1zw508d6qejxtdg4y5r3zarvaryvg6kdaj` : `5210751e76e8199196d454941c45d1b3a323`
- `tb1qqqqqp399et2xygdj5xreqhjjvcmzhxw4aywxecjdzew6hylgvsesrxh6hy` : `0020000000c4a5cad46221b2a187905e5266362b99d5e91c6ce24d165dab93e86433`

The following list gives invalid segwit addresses and the reason for their invalidity.

- `tc1qw508d6qejxtdg4y5r3zarvary0c5xw7kg3g4ty` : Invalid human-readable part
- `bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t5` : Invalid checksum
- `BC13W508D6QEJXTDG4Y5R3ZARVARY0C5XW7KN40WF2` : Invalid witness version
- `bc1rw5uspcuh` : Invalid program length
- `bc10w508d6qejxtdg4y5r3zarvary0c5xw7kw508d6qejxtdg4y5r3zarvary0c5xw7kw5rljs90` : Invalid program length

- `BC1QR508D6QEJXTDG4Y5R3ZARVARYV98GJ9P` : Invalid program length for witness version 0 (per BIP141)
- `tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sL5k7` : Mixed case
- `bc1zw508d6qejxtdg4y5r3zarvaryvqyzf3du` : zero padding of more than 4 bits
- `tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3pjxtptv` : Non-zero padding in 8-to-5 conversion
- `bc1gmk9yu` : Empty data section

Checksum design

Design choices

BCH codes can be constructed over any prime-power alphabet and can be chosen to have a good trade-off between size and error-detection capabilities. While most work around BCH codes uses a binary alphabet, that is not a requirement. This makes them more appropriate for our use case than CRC codes. Unlike Reed-Solomon codes, they are not restricted in length to one less than the alphabet size. While they also support efficient error correction, the implementation of just error detection is very simple.

We pick 6 checksum characters as a trade-off between length of the addresses and the error-detection capabilities, as 6 characters is the lowest number sufficient for a random failure chance below 1 per billion. For the length of data we're interested in protecting (up to 71 bytes for a potential future 40-byte witness program), BCH codes can be constructed that guarantee detecting up to 4 errors.

Selected properties

Many of these codes perform badly when dealing with more errors than they are designed to detect, but not all. For that reason, we consider codes that are designed to detect only 3 errors as well as 4 errors, and analyse how well they perform in practice.

The specific code chosen here is the result of:

- Starting with an exhaustive list of 159605 BCH codes designed to detect 3 or 4 errors up to length 93, 151, 165, 341, 1023, and 1057.
- From those, requiring the detection of 4 errors up to length 71, resulting in 28825 remaining codes.
- From those, choosing the codes with the best worst-case window for 5-character errors, resulting in 310 remaining codes.

- From those, picking the code with the lowest chance for not detecting small numbers of *bit* errors.

As a naive search would require over 6.5 * 10^{19} checksum evaluations, a collision-search approach was used for analysis. The code can be found here.

Properties

The following table summarizes the chances for detection failure (as multiples of 1 in 10^9).

Window length		Number of wrong characters					
Length	**Description**	**≤4**	**5**	**6**	**7**	**8**	**≥9**
8	Longest detecting 6 errors	0			1.127	0.909	n/a
18	Longest detecting 5 errors	0		0.965	0.929	0.932	0.931
19	Worst case for 6 errors	0	0.093	0.972	0.928	0.931	
39	Length for a P2WPKH address	0	0.756	0.935	0.932	0.931	
59	Length for a P2WSH address	0	0.805	0.933	0.931		
71	Length for a 40-byte program address	0	0.830	0.934	0.931		
89	Longest detecting 4 errors	0	0.867	0.933	0.931		

This means that when 5 changed characters occur randomly distributed in the 39 characters of a P2WPKH address, there is a chance of *0.756 per billion* that it will go undetected. When those 5 changes occur randomly within a 19-character window, that chance goes down to *0.093 per billion*. As the number of errors goes up, the chance converges towards *1 in 2^{30}* = *0.931 per billion*.

Even though the chosen code performs reasonably well up to 1023 characters, other designs are preferable for lengths above 89 characters (excluding the separator).

Acknowledgements

This document is inspired by the address proposal by Rusty Russell, the base32 proposal by Mark Friedenbach, and had input from Luke Dashjr, Johnson Lau, Eric Lombrozo, Peter Todd, and various other reviewers.