```
BIP: 380
Layer: Applications
Title: Output Script Descriptors General Operation
Author: Pieter Wuille <pieter@wuille.net>
        Andrew Chow <andrew@achow101.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0380
Status: Draft
Type: Informational
Created: 2021-06-27
License: BSD-2-Clause
```

Abstract

Output Script Descriptors are a simple language which can be used to describe collections of output scripts. There can be many different descriptor fragments and functions. This document describes the general syntax for descriptors, descriptor checksums, and common expressions.

Copyright

Motivation

Bitcoin wallets traditionally have stored a set of keys which are later serialized and mutated to produce the output scripts that the wallet watches and the addresses it provides to users. Typically backups have consisted of solely the private keys, nowadays primarily in the form of BIP 39 mnemonics. However this backup solution is insuffient, especially since the introduction of Segregated Witness which added new output types. Given just the private keys, it is not possible for restored wallets to know which kinds of output scripts and addresses to produce. This has lead to incompatibilities between wallets when restoring a backup or exporting data for a watch only wallet.

Further complicating matters are BIP 32 derivation paths. Although BIPs 44, 49, and 84 have specified standard BIP 32 derivation paths for different output scripts and addresses, not all wallets support them nor use those derivation paths. The lack of derivation path information in these backups and exports leads to further incompatibilities between wallets.

Current solutions to these issues have not been generic and can be viewed as being layer violations. Solutions such as introducing different version bytes for extended key serialization both are a layer violation (key derivation should be separate from script type meaning) and specific only to a particular derivation path and script type.

Output Script Descriptors introduces a generic solution to these issues. Script types are specified explicitly through the use of Script Expressions. Key derivation paths are specified explicitly in Key Expressions. These allow for creating wallet backups and exports which specify the exact scripts, subscripts (redeemScript, witnessScript, etc.), and keys to produce. With the general structure specified in this BIP, new Script Expressions can be introduced as new script types are added. Lastly, the use of common terminology and existing standards allow for Output Script Descriptors to be engineer readable so that the results can be understood at a glance.

Specification

Descriptors consist of several types of expressions. The top level expression is a `SCRIPT`. This expression may be followed by `#CHECKSUM`, where `CHECKSUM` is an 8 character alphanumeric descriptor checksum. Although the checksum is optional for parsing, applications may choose to reject descriptors that do not contain a checksum.

Script Expressions

Script Expressions (denoted `SCRIPT`) are expressions which correspond directly with a Bitcoin script. These expressions are written as functions and take arguments. Such expressions have a script template which is filled with the arguments correspondingly. Expressions are written with a human readable identifier string with the arguments enclosed with parentheses. The identifier string should be alphanumeric and may include underscores.

The arguments to a script expression are defined by that expression itself. They could be a script expression, a key expression, or some other expression entirely.

Key Expressions

A common expression used as an argument to script expressions are key expressions (denoted `KEY`). These represent a public or private key and, optionally, information about the origin of that key. Key expressions can only be used as arguments to script expressions.

Key expressions consist of:

- Optionally, key origin information, consisting of:
  - An open bracket `[`
  - Exactly 8 hex characters for the fingerprint of the key where the derivation starts (see BIP 32 for details)
  - Followed by zero or more `/NUM` or `/NUMh` path elements to indicate the unhardened or hardened derivation steps between the fingerprint and the key that follows.
  - A closing bracket `]`
- Followed by the actual key, which is either:

- A hex encoded public key, which depending the script expression, may be either:
  - 66 hex character string beginning with `02` or `03` representing a compressed public key
  - 130 hex character string beginning with `04` representing an uncompressed public key
- A WIF encoded private key
- `xpub` encoded extended public key or `xprv` encoded extended private key (as defined in BIP 32)
  - Followed by zero or more `/NUM` or `/NUMh` path elements indicating BIP 32 derivation steps to be taken after the given extended key.
  - Optionally followed by a single `/*` or `/*h` final step to denote all direct unhardened or hardened children.

If the `KEY` is a BIP 32 extended key, before output scripts can be created, child keys must be derived using the derivation information that follows the extended key. When the final step is `/*` or `/*'`, an output script will be produced for every child key index. The derived key must be not be serialized as an uncompressed public key. Script Expressions may have further requirements on how derived public keys are serialized for script creation.

In the above specification, the hardened indicator `h` may be replaced with alternative hardened indicators of `H` or `'`.

Normalization of Key Expressions with Hardened Derivation When a descriptor is exported without private keys, it is necessary to do additional derivation to remove any intermediate hardened derivation steps for the exported descriptor to be useful. The exporter should derive the extended public key at the last hardened derivation step and use that extended public key as the key in the descriptor. The derivation steps that were taken to get to that key must be added to the previous key origin information. If there is no key origin information, then one must be added for the newly derived extended public key. If the final derivation is hardened, then it is not necessary to do additional derivation.

Character Set

The expressions used in descriptors must only contain characters within this character set so that the descriptor checksum will work.

The allowed characters are:

```
0123456789()[],'/*abcdefgh@:$%{}
IJKLMNOPQRSTUVWXYZ&+-.;<=>?!^_|~
ijklmnopqrstuvwxyzABCDEFGH`#"\<space>
```

Note that on the last line is a space character.

This character set is written as 3 groups of 32 characters in this specific order so that the checksum below can identify more errors. The first group are the most common "unprotected" characters (i.e. things such as hex and keypaths that do not already have their own checksums). Case errors cause an offset that is a multiple of 32 while as many alphabetic characters are in the same group while following the previous restrictions.

Checksum

Following the top level script expression is a single octothorpe (`#`) followed by the 8 character checksum. The checksum is an error correcting checksum similar to bech32.

The checksum has the following properties:

- Mistakes in a descriptor string are measured in "symbol errors". The higher the number of symbol errors, the harder it is to detect:
  - An error substituting a character from `0123456789()[],'/*abcdefgh@:$%{}` for another in that set always counts as 1 symbol error.
    - Note that hex encoded keys are covered by these characters. Extended keys (`xpub` and `xprv`) use other characters too, but also have their own checksum mechanism.
    - `SCRIPT` expression function names use other characters, but mistakes in these would generally result in an unparsable descriptor.
  - A case error always counts as 1 symbol error.
  - Any other 1 character substitution error counts as 1 or 2 symbol errors.
- Any 1 symbol error is always detected.
- Any 2 or 3 symbol error in a descriptor of up to 49154 characters is always detected.
- Any 4 symbol error in a descriptor of up to 507 characters is always detected.
- Any 5 symbol error in a descriptor of up to 77 characters is always detected.
- Is optimized to minimize the chance of a 5 symbol error in a descriptor up to 387 characters is undetected
- Random errors have a chance of 1 in 240 of being undetected.

The checksum itself uses the same character set as bech32: `qpzry9x8gf2tvdw0s3jn54khce6mua7l`

Valid descriptor strings with a checksum must pass the criteria for validity specified by the Python3 code snippet below. The function `descsum_check` must return true when its argument `s` is a descriptor consisting in the form `SCRIPT#CHECKSUM`.

```
INPUT_CHARSET = "0123456789()[],'/*abcdefgh@:$%{}IJKLMNOPQRSTUVWXYZ&+-.;<=>?!^_|~ijklmnopqrs
CHECKSUM_CHARSET = "qpzry9x8gf2tvdw0s3jn54khce6mua7l"
GENERATOR = [0xf5dee51989, 0xa9fdca3312, 0x1bab10e32d, 0x3706b1677a, 0x644d626ffd]
```

```
def descsum_polymod(symbols):
    """Internal function that computes the descriptor checksum."""
    chk = 1
    for value in symbols:
        top = chk >> 35
        chk = (chk & 0x7ffffffff) << 5 ^ value
        for i in range(5):
            chk ^= GENERATOR[i] if ((top >> i) & 1) else 0
    return chk

def descsum_expand(s):
    """Internal function that does the character to symbol expansion"""
    groups = []
    symbols = []
    for c in s:
        if not c in INPUT_CHARSET:
            return None
        v = INPUT_CHARSET.find(c)
        symbols.append(v & 31)
        groups.append(v >> 5)
        if len(groups) == 3:
            symbols.append(groups[0] * 9 + groups[1] * 3 + groups[2])
            groups = []
    if len(groups) == 1:
        symbols.append(groups[0])
    elif len(groups) == 2:
        symbols.append(groups[0] * 3 + groups[1])
    return symbols

def descsum_check(s):
    """Verify that the checksum is correct in a descriptor"""
    if s[-9] != '#':
        return False
    if not all(x in CHECKSUM_CHARSET for x in s[-8:]):
        return False
    symbols = descsum_expand(s[:-9]) + [CHECKSUM_CHARSET.find(x) for x in s[-8:]]
    return descsum_polymod(symbols) == 1
```

This implements a BCH code that has the properties described above. The entire descriptor string is first processed into an array of symbols. The symbol for each character is its position within its group. After every 3rd symbol, a 4th symbol is inserted which represents the group numbers combined together. This means that a change that only affects the position within a group, or only a group number change, will only affect a single symbol.

To construct a valid checksum given a script expression, the code below can be used:

```
def descsum_create(s):
    """Add a checksum to a descriptor without"""
    symbols = descsum_expand(s) + [0, 0, 0, 0, 0, 0, 0, 0]
    checksum = descsum_polymod(symbols) ^ 1
    return s + '#' + ''.join(CHECKSUM_CHARSET[(checksum >> (5 * (7 - i))) & 31] for i in ran
```

Test Vectors

The following tests cover the checksum and character set:

- Valid checksum: `raw(deadbeef)#89f8spxm`
- No checksum: `raw(deadbeef)`
- Missing checksum: `raw(deadbeef)#`
- Too long checksum (9 chars): `raw(deadbeef)#89f8spxmx`
- Too short checksum (7 chars): `raw(deadbeef)#89f8spx`
- Error in payload: `raw(deedbeef)#89f8spxm`
- Error in checksum: `raw(deedbeef)##9f8spxm`
- Invalid characters in payload: `raw(Ü)#00000000`

The following tests cover key expressions:

Valid expressions:

- Compressed public key: `0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d5e154050a9346ea98ce600`
- Uncompressed public key: `04a34b99f22c790c4e36b2b3c2c35a36db06226e41c692fc82b8b56ac1c540c5b`
- Public key with key origin: `[deadbeef/0h/0h/0h]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d5e`
- Public key with key origin (`'` as hardened indicator): `[deadbeef/0'/0'/0']0260b2003c386519fc9eadf2b`
- Public key with key origin (mixed hardened indicator): `[deadbeef/0'/0h/0']0260b2003c386519fc9eadf`
- WIF uncompressed private key `5KYZdUEo39z3FPrtuX2QbbwGnNP5zTd7yyr2SC1j299sBCnWjss`
- WIF compressed private key `L4rK1yDtCWekvXuE6oXD9jCYfFNV2cWRpVuPLBcCU2z8TrisoyY1`
- Extended public key: `xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRkrgZw4k`
- Extended public key with key origin: `[deadbeef/0h/1h/2h]xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54`
- Extended public key with derivation: `[deadbeef/0h/1h/2h]xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54`
- Extended public key with derivation and children: `[deadbeef/0h/1h/2h]xpub6ERApfZwUNrhLCkDtcHTcxd`
- Extended public key with hardened derivation and unhardened children: `xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRkrgZw4koxb5JaHWkY4ALHY2gr`
- Extended public key with hardened derivation and children: `xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54`
- Extended public key with key origin, hardened derivation and children: `[deadbeef/0h/1h/2]xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRkrgZw4k`
- Extended private key: `xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2YvmhqLQYMmrj4xJXXWYpDPS3xz7iAxn8L39`
- Extended private key with key origin: `[deadbeef/0h/1h/2h]xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2Yvmh`
- Extended private key with derivation: `[deadbeef/0h/1h/2h]xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2Yvmh`
- Extended private key with derivation and children: `[deadbeef/0h/1h/2h]xprvA1RpRA33e1JQ7ifknakTFpg`
- Extended private key with hardened derivation and unhardened children: `xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2YvmhqLQYMmrj4xJXXWYpDPS3xz7iAxn8L39njGVyuoseXzU6rcxFLJ`
- Extended private key with hardened derivation and children: `xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2YvmhqLQYMmrj4xJXXWYpDPS3xz7iAxn8L39njGVyuoseXzU6rcxFLJ`

- Extended private key with key origin, hardened derivation and children: `[deadbeef/0h/1h/2]xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2YvmhqLQYMmrj4xJXXWYpDPS3xz7iAxn8L39n`

Invalid expressiosn:

- Children indicator in key origin: `[deadbeef/0h/0h/0h/*]0260b2003c386519fc9eadf2b5cf124dd8eea4c4`
- Trailing slash in key origin: `[deadbeef/0h/0h/0h/]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d5`
- Too short fingerprint: `[deadbef/0h/0h/0h]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d5e15405`
- Too long fingerprint: `[deadbeeef/0h/0h/0h]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d5e1540`
- Invalid hardened indicators: `[deadbeeef/0f/0f/0f]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d`
- Invalid hardened indicators: `[deadbeeef/0H/0H/0H]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d`
- Invalid hardened indicators: `[deadbeeef/-0/-0/-0]0260b2003c386519fc9eadf2b5cf124dd8eea4c4e68d`
- Private key with derivation: `L4rK1yDtCWekvXuE6oXD9jCYfFNV2cWRpVuPLBcCU2z8TrisoyY1/0`
- Private key with derivation children: `L4rK1yDtCWekvXuE6oXD9jCYfFNV2cWRpVuPLBcCU2z8TrisoyY1/*`
- Derivation index out of range: `xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5e2DtALGdso3p` `"pkh(xpub661MyMwAqRbcFW31YEwpkMuc5THy2PSt5bDMsktWQcFF8syAmRUapSCGu8ED9W6oDMSgv6Zz8idoc4`
- Invalid derivation index: `xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5e2DtALGdso3pGz6ss` `"pkh(xpub661MyMwAqRbcFW31YEwpkMuc5THy2PSt5bDMsktWQcFF8syAmRUapSCGu8ED9W6oDMSgv6Zz8idoc4`
- Multiple key origins: `[aaaaaaaa][aaaaaaaa]xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5`
- Missing key origin start: `aaaaaaaa]xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5e2DtALGd`
- Non hex fingerprint: `[gaaaaaaa]xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5e2DtALGdso`
- Key origin with no public key: `[deadbeef]`

Backwards Compatibility

Output script descriptors are an entirely new language which is not compatible with any existing software. However many components of the expressions reuse encodings and serializations defined by previous BIPs.

Output script descriptors are designed for future extension with further fragment types and new script expressions. These will be specified in additional BIPs.

Reference Implementation

Descriptors have been implemented in Bitcoin Core since version 0.17.

Appendix A: Index of Expressions

Future BIPs may specify additional types of expressions. All available expression types are listed in this table.

Name	Denoted As	BIP
Script	`SCRIPT`	380
Key	`KEY`	380
Tree	`TREE`	386

Appendix B: Index of Script Expressions

Script expressions will be specified in additional BIPs. This Table lists all available Script expressions and the BIPs specifying them.

Expression	BIP
`pk(KEY)`	381
`pkh(KEY)`	381
`sh(SCRIPT)`	381
`wpkh(KEY)`	382
`wsh(SCRIPT)`	382
`multi(NUM, KEY, ..., KEY)`	383
`sortedmulti(NUM, KEY, ..., KEY)`	383
`combo(KEY)`	384
`raw(HEX)`	385
`addr(ADDR)`	385
`tr(KEY)`, `tr(KEY, TREE)`	386