Transaction Spec for Bitcoin

Layout: Specification
Date: 2017-08-26
Activation: 1515888000
Version: 1.0

This section of the Bitcoin (BSV) specification ("spec") documents the **transaction data structure** for implementing a compatible BSV client, including transaction format, opcodes, and examples.

Transaction

A **transaction** is one of the two base primitives in the BSV system, the other being a **block**. Primitive in this context means that it is one of the data types for which the BSV spec provides built-in support. A transaction is a transfer of BSV that is broadcast to the network and collected into blocks. A transaction typically references previous transaction outputs as new transaction **inputs** and dedicates all input Bitcoincash values to new outputs. Transactions are not encrypted, so it is possible to browse and view every transaction ever collected into a block. Once transactions are buried under enough confirmations they can be considered irreversible. Transaction comprises a **signature** and **redeem script pair**, which provides flexibility in releasing outputs. A serialized transaction contains an input and an output.

Transaction requirements

A transaction that meets the criteria documented here is said to be standard. Standard transactions are accepted into the mempool and relayed by nodes on the network. This ensures that nodes have a similar looking mempool so that the system behave predictably. Standard transaction outputs nominate addresses, and the redemption of any future inputs requires a relevant signature. Transaction requirements:

- Transaction size: < 100k
- Version must be 1 or 2
- Signature script must be data push only
- Script size must be 1650 or less

NOTE: A BSV node should be able to process non-standard transactions as well. Even if a node cannot successfully relay a non-standard transaction, it should not crash if it ends up having to process one of those transactions.

Transaction Input

Inputs to a transaction include the outpoint, signature script, and sequence.

An input is a reference to an output from a previous transaction. Multiple inputs are often listed in a transaction. All of the new transaction's input values (that is, the total coin value of the previous outputs referenced by the new transaction's inputs) are added up, and the total (less any transaction fee) is completely used by the outputs of the new transaction. Previous tx is a hash of a previous transaction. Index is the specific output in the referenced transaction. scriptSig is the first half of a script (discussed in more detail later).

Transaction Output

Outputs from a transaction include the BSV amount and redeem script which is used to spend the output and sets up parameters for the signature script. Redeem scripts should not use OP_CODES.

Opcodes

The opcodes used in the pubkey scripts of standard transactions are as follows.

0x00 to 0x4e

There are arious data pushing opcodes from 0x00 to 0x4e (1--78) that must be used must be used to push signatures and public keys onto the stack.

OP_TRUE/OP_1, OP_2 through OP_16

OP_TRUE/OP_1 (0x51) and OP_2 through OP_16 (0x52--0x60) push the values 1 through 16 to the stack.

OP_CHECKSIG

OP_CHECKSIG consumes a signature and a full public key, and pushes true onto the stack if the transaction data specified by the SIGHASH flag was converted into the signature using the same ECDSA private key that generated the public key. Otherwise, it pushes false onto the stack.

OP_DUP

OP_DUP pushes a copy of the topmost stack item on to the stack.

OP_HASH160

OP_HASH160 consumes the topmost item on the stack, computes the RIPEMD160(SHA256()) hash of that item, and pushes that hash onto the stack.

OP_EQUAL

OP_EQUAL consumes the top two items on the stack, compares them, and pushes true onto the stack if they are the same, false if not.

OP_VERIFY

OP_VERIFYconsumes the topmost item on the stack. If that item is zero (false) it terminates the script in failure.

OP_EQUALVERIFY

OP_EQUALVERIFY runs OP_EQUAL and then OP_VERIFY in sequence.

OP_CHECKMULTISIG

OP_CHECKMULTISIGconsumes the value (n) at the top of the stack, consumes that many of the next stack levels (public keys), consumes the value (m) now at the top of the stack, and

consumes that many of the next values (signatures) plus one extra value.

The "one extra value" it consumes is the result of an off-by-one error in the original Bitcoin implementation. This value is not used in Bitcoincash, so signature scripts prefix the list of secp256k1 signatures with a single OP_0 (0x00).

OP_CHECKMULTISIG compares the first signature against each public key until it finds an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds an ECDSA match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result.

Because public keys are not checked again if they fail any signature comparison, signatures must be placed in the signature script using the same order as their corresponding public keys were placed in the pubkey script or redeem script.

The OP_CHECKMULTISIG verification process requires that signatures in the signature script be provided in the same order as their corresponding public keys in the pubkey script or redeem script.

OP_RETURN

OP_RETURN terminates the script in failure when executed.

Address Conversion

The hashes used in P2PKH and P2SH outputs are commonly encoded as Bitcoincash addresses. This is the procedure to encode those hashes and decode the addresses.

First, get your hash. For P2PKH, you RIPEMD-160(SHA256()) hash a ECDSA public key derived from your 256-bit ECDSA private key (random data). For P2SH, you RIPEMD-160(SHA256()) hash a redeem script serialized in the format used in raw transactions.

Taking the resulting hash:

1. Add an address version byte in front of the hash. The version bytes commonly used by Bitcoincash are:
    - 0x00 for P2PKH addresses on the main Bitcoincash network (mainnet)
    - 0x6f for P2PKH addresses on the Bitcoincash testing network (testnet)
    - 0x05 for P2SH addresses on mainnet
    - 0xc4 for P2SH addresses on testnet
2. Create a copy of the version and hash; then hash that twice with SHA256: SHA256(SHA256(version . hash))
3. Extract the first four bytes from the double-hashed copy. These are used as a checksum to ensure the base hash gets transmitted correctly.
4. Append the checksum to the version and hash, and encode it as a base58 string: BASE58(version . hash . checksum).

The code can be traced using the [base58 header file][core base58.h].

To convert addresses back into hashes, reverse the base58 encoding, extract the checksum, repeat the steps to create the checksum and compare it against the extracted checksum, and then remove the version byte.

Raw Transaction Format

Bitcoincash transactions are broadcast between peers in a serialized byte format, called raw format. It is this form of a transaction which is SHA256(SHA256()) hashed to create the TXID and, ultimately, the merkle root of a block containing the transaction---making the transaction format part of the consensus rules.

Bitcoincash Core and many other tools print and accept raw transactions encoded as hex.

A raw transaction has the following top-level format:

Bytes	Name	Data Type	Description
4	version	uint32_t	Transaction version number; currently version 1. Programs creating transactions using newer consensus rules may use higher version numbers.
Varies	tx_in count	compactSize uint	Number of inputs in this transaction.
Varies	tx_in	txIn	Transaction inputs. See description of txIn below.
Varies	tx_out count	compactSize uint	Number of outputs in this transaction.
Varies	tx_out	txOut	Transaction outputs. See description of txOut below.
4	lock_time	uint32_t	A time (Unix epoch time) or block number.

A transaction may have multiple inputs and outputs, so the txIn and txOut structures may recur within a transaction. CompactSize unsigned integers are a form of variable-length integers; they are described in CompactSize unsigned integer.

TxIn: Transaction Input

Each non-coinbase input spends an outpoint from a previous transaction.

Bytes	Name	Data Type	Description
36	previous_output	outpoint	The previous outpoint being spent. See description of outpoint below.
Varies	script bytes	compactSize uint	The number of bytes in the signature script. Maximum is 10,000 bytes.
Varies	signature script	char[]	Script that satisfies conditions in the outpoint's pubkey script. Should only contain data pushes.
4	sequence	uint32_t	Sequence number. Default is 0xffffffff.

Outpoint

The outpoint is a reference to an output from a previous transaction. Because a single transaction can include multiple outputs, the outpoint structure includes both a TXID and an output index number to refer to the specific part of a specific output.

Bytes	Name	Data Type	Description
32	hash	char[32]	The TXID of the transaction holding the output to spend. The TXID is a hash provided here in internal byte order.
4	index	uint32_t	The output index number of the specific output to spend from the transaction. The first output is 0x00000000.

TxOut: Transaction Output

Each output spends a certain number of Satoshis, placing them under control of anyone who can satisfy the provided pubkey script.

Bytes	Name	Data Type	Description
8	value	int64_t	Number of Satoshis to spend. May be zero; the sum of all outputs may not exceed the sum of Satoshis previously spent to the outpoints provided in the input section. (Exception: coinbase transactions spend the block subsidy and collected transaction fees.)
1+	pk_script bytes	compactSize uint	Number of bytes in the pubkey script. Maximum is 10,000 bytes.
Varies	pk_script	char[]	Defines the conditions which must be satisfied to spend this output.

CompactSize Unsigned Integers

The raw transaction format and several peer-to-peer network messages use a type of variable-length integer to indicate the number of bytes in a following piece of data.

The source code and this document refers to these variable length integers as compactSize. Because it's used in the transaction format, the format of compactSize unsigned integers is part of the consensus rules.

For numbers from 0 to 252, compactSize unsigned integers look like regular unsigned integers. For other numbers up to 0xffffffffffffffff, a byte is prefixed to the number to indicate its length---but otherwise the numbers look like regular unsigned integers in little-endian order. For example, the number 515 is encoded as 0xfd0302.

Value	Bytes Used	Format
>= 0 && <= 252	1	uint8_t
>= 253 && <= 0xffff	3	0xfd followed by the number as uint16_t
>= 0x10000 && <= 0xffffffff	5	0xfe followed by the number as uint32_t
>= 0x100000000 && <= 0xffffffffffffffff	9	0xff followed by the number as uint64_t

Signature Script

The purpose of the signature script (scriptSig) is to ensure that the spender is a legitimate spender, that is, evidence of private key held.

The scriptSig contains two components: a signature and a public key. The public key must match the hash given in the script of the redeemed output. The public key is used to verify the redeemers signature, which is the second component. More precisely, the second component is an ECDSA signature over a hash of a simplified version of the transaction. It, combined with the public key, proves the transaction was created by the real owner of the address in question.

Signature scripts are not signed, so anyone can modify them. This means signature scripts should only contain data and data-pushing opcodes which can't be modified without causing the pubkey script to fail. Placing non-data-pushing opcodes in the signature script currently makes a transaction non-standard, and future consensus rules may forbid such transactions altogether. (Non-data-pushing opcodes are already forbidden in signature scripts when spending a P2SH pubkey script.)

Sequence

Check lock time verify (s4)

Check sequence verify (s4)

Standard Transaction Format Examples

P2SH

23-bytes
OP_HASH160
<reedem script hash>
OP_EQUAL
Use address version=1 and hash=<reedem script hash>

P2PKH

25 bytes
OP_DUP
OP_HASH160
<public key="" hash="">
OP_EQUALVERIFY
OP_CHECKSIG
Use address version=0 and hash= [public key="" hash=""]

P2PK

35 or 67 bytes
<public key="">
OP_CHECKSIG
Use address version=0 and hash=HASH160([public key=""])

Bare multisig

<n: [0-20]="">
<pubkey 0="">
...
<pubkey n="">
<(null)>
OP_CHECKMULTISIG

NOTE: Bare multisig (which isn't wrapped into P2SH) is limited to 3-of-3.

Data carrier

Limited to one per transaction
Limited to 223 bytes
OP_RETURN
<push data="">

NOTE: Multiple pushes of data are allowed.