Bitcoin Developer Reference

Find technical details and API documentation.

The Developer Reference aims to provide technical details and API information to help you start building Bitcoin-based applications, but it is not a specification. To make the best use of this documentation, you may want to install the current version of Bitcoin Core, either from source or from a pre-compiled executable.

Questions about Bitcoin development are best asked in one of the Bitcoin development communities. Errors or suggestions related to documentation on Bitcoin.org can be submitted as an issue or posted to the bitcoin-documentation mailing list.

In the following documentation, some strings have been shortened or wrapped: "[…]" indicates extra data was removed, and lines ending in a single backslash "\" are continued below. If you hover your mouse over a paragraph, cross-reference links will be shown in blue. If you hover over a cross-reference link, a brief definition of the term will be displayed in a tooltip.

Not A Specification

The Bitcoin.org Developer Documentation describes how Bitcoin works to help educate new Bitcoin developers, but it is not a specification—and it never will be.

Bitcoin security depends on consensus. Should your program diverge from consensus, its security is weakened or destroyed. The cause of the divergence doesn't matter: it could be a bug in your program, it could be an error in this documentation which you implemented as described, or it could be you do everything right but other software on the network behaves unexpectedly. The specific cause will not matter to the users of your software whose wealth is lost.

The only correct specification of consensus behavior is the actual behavior of programs on the network which maintain consensus. As that behavior is subject to arbitrary inputs in a large variety of unique environments, it cannot ever be fully documented here or anywhere else.

However, the Bitcoin Core developers are working on making their consensus code portable so other implementations can use it. Bitcoin Core 0.10.0 will provide `libbitcoinconsensus`, a first attempt at exporting some consensus code. Future versions of Bitcoin Core will likely provide consensus code that is more complete, more portable, and more consistent in diverse environments.

In addition, we also warn you that this documentation has not been extensively reviewed by Bitcoin experts and so likely contains numerous errors. At the bottom of the menu on the left, you will find links that allow you to report an issue or to edit the documentation on GitHub. Please use those links if you find any errors or important missing information.

Block Chain

The following subsections briefly document core block details.

Block Headers

Block headers are serialized in the 80-byte format described below and then hashed as part of Bitcoin's proof-of-work algorithm, making the serialized header format part of the consensus rules.

Bytes	Name	Data Type	Description
4	version	uint32_t	The block version number indicates which set of block validation rules to follow. See the list of block versions below.
	previous		

32	block header hash	char[32]	A SHA256(SHA256()) hash in internal byte order of the previous block’s header. This ensures no previous block can be changed without also changing this block’s header.
32	merkle root hash	char[32]	A SHA256(SHA256()) hash in internal byte order. The merkle root is derived from the hashes of all transactions included in this block, ensuring that none of those transactions can be modified without modifying the header. See the merkle trees section below.
4	time	uint32_t	The block time is a Unix epoch time when the miner started hashing the header (according to the miner). Must be greater than or equal to the median time of the previous 11 blocks. Full nodes will not accept blocks with headers more than two hours in the future according to their clock.
4	nBits	uint32_t	An encoded version of the target threshold this block’s header hash must be less than or equal to. See the nBits format described below.
4	nonce	uint32_t	An arbitrary number miners change to modify the header hash in order to produce a hash below the target threshold. If all 32-bit values are tested, the time can be updated or the coinbase transaction can be changed and the merkle root updated.

The hashes are in internal byte order; the other values are all in little-endian order.

An example header in hex:

```
02000000 ........................... Block version: 2

b6ff0b1b1680a2862a30ca44d346d9e8
910d334beb48ca0c0000000000000000 ... Hash of previous block's header
9d10aa52ee949386ca9385695f04ede2
70dda20810decd12bc9b048aaab31471 ... Merkle root

24d95a54 ........................... Unix time: 1415239972
30c31b18 ........................... Target: 0x1bc330 * 256**(0x18-3)
fe9f0864 ........................... Nonce
```

Block Versions

- **Version 1** was introduced in the genesis block (January 2009).
- **Version 2** was introduced in Bitcoin Core 0.7.0 (September 2012) as a soft fork. As described in BIP34, valid version 2 blocks require a block height parameter in the coinbase. Also described in BIP34 are rules for rejecting certain blocks; based on those rules, Bitcoin Core 0.7.0 and later versions began to reject version 2 blocks without the block height in coinbase at block height 224,412 (March 2013) and began to reject new version 1 blocks three weeks later at block height 227,930.
- **Version 3** blocks will be introduced when sufficient numbers of miners switch to using Bitcoin Core 0.10.0 and other versions that create version 3 blocks. As described in draft BIP66, this soft fork change requires strict DER encoding for all ECDSA signatures used in transactions appearing in version 3 or later blocks. Transactions that do not use strict DER encoding have been non-standard since Bitcoin Core 0.8.0.
- **Version 4** blocks will likely be introduced in the near-future as specified in draft BIP62. Possible changes include:
  - Reject version 4 blocks that include any version 2 transactions that don’t adhere to any of the version 2 transaction rules. These rules are not yet described in this documentation; see BIP62 for details.
  - A soft fork rollout of version 4 blocks identical to the rollout used for version 3 blocks (described briefly in BIP62 and in more detail in BIP34).

Merkle Trees

For an overview of merkle trees, see the block chain guide.

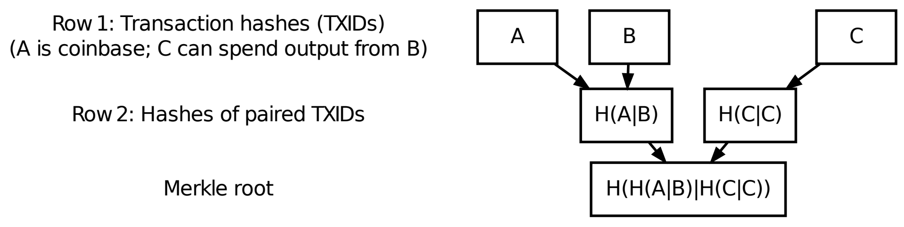
The merkle root is constructed using all the TXIDs of transactions in this block, but first the TXIDs are placed in order as required by the consensus rules:

- The coinbase transaction's TXID is always placed first.
- Any input within this block can spend an output which also appears in this block (assuming the spend is otherwise valid). However, the TXID corresponding to the output must be placed at some point before the TXID corresponding to the input. This ensures that any program parsing block chain transactions linearly will encounter each output before it is used as an input.

If a block only has a coinbase transaction, the coinbase TXID is used as the merkle root hash.

If a block only has a coinbase transaction and one other transaction, the TXIDs of those two transactions are placed in order, concatenated as 64 raw bytes, and then SHA256(SHA256()) hashed together to form the merkle root.

If a block has three or more transactions, intermediate merkle tree rows are formed. The TXIDs are placed in order and paired, starting with the coinbase transaction's TXID. Each pair is concatenated together as 64 raw bytes and SHA256(SHA256()) hashed to form a second row of hashes. If there are an odd (non-even) number of TXIDs, the last TXID is concatenated with a copy of itself and hashed. If there are more than two hashes in the second row, the process is repeated to create a third row (and, if necessary, repeated further to create additional rows). Once a row is obtained with only two hashes, those hashes are concatenated and hashed to produce the merkle root.

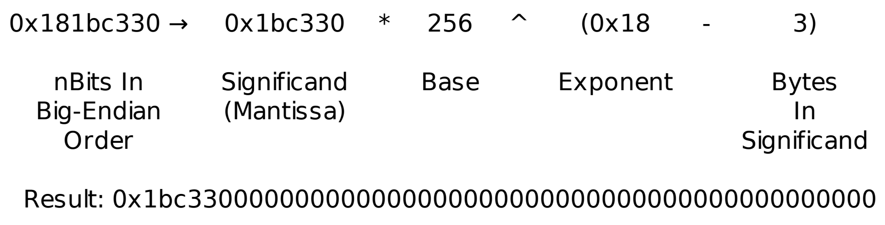

Row 1: Transaction hashes (TXIDs)
(A is coinbase; C can spend output from B)
Row 2: Hashes of paired TXIDs
Merkle root
A
B
C
H(A|B)
H(C|C)
H(H(A|B)|H(C|C))

Example Merkle Tree Construction [Hash function H() = SHA256(SHA256())]

TXIDs and intermediate hashes are always in internal byte order when they're concatenated, and the resulting merkle root is also in internal byte order when it's placed in the block header.

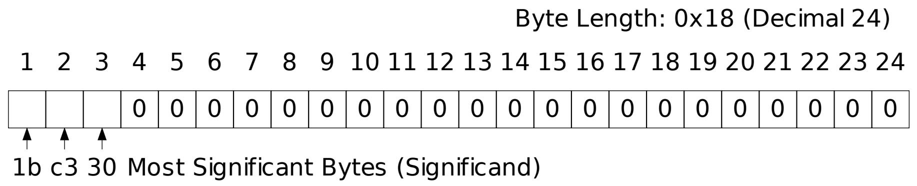
Target nBits

The target threshold is a 256-bit unsigned integer which a header hash must be equal to or below in order for that header to be a valid part of the block chain. However, the header field *nBits* provides only 32 bits of space, so the target number uses a less precise format called "compact" which works like a base-256 version of scientific notation:


0x181bc330 → 0x1bc330 * 256 ^ (0x18 - 3)
nBits In Big-Endian Order
Significand (Mantissa)
Base
Exponent
Bytes In Significand
Result: 0x1bc330000000000000000000000000000000000000000000

Converting nBits Into A Target Threshold

As a base-256 number, nBits can be quickly parsed as bytes the same way you might parse a decimal number in base-10 scientific notation:



Quickly Converting nBits 0x181bc330 Into The Target Threshold 0x1bc330000000000000000000000000000000000000000000

Although the target threshold should be an unsigned integer, the original nBits implementation inherits properties from a signed data class, allowing the target threshold to be negative if the high bit of the significand is set. This is useless—the header hash is treated as an unsigned number, so it can never be equal to or lower than a negative target threshold. Bitcoin Core deals with this in two ways:

- When parsing nBits, Bitcoin Core converts a negative target threshold into a target of zero, which the header hash can equal (in theory, at least).
- When creating a value for nBits, Bitcoin Core checks to see if it will produce an nBits which will be interpreted as negative; if so, it divides the significand by 256 and increases the exponent by 1 to produce the same number with a different encoding.

Some examples taken from the Bitcoin Core test cases:

nBits	Target	Notes
0x01003456	0x00	
0x01123456	0x12	
0x02008000	0x80	
0x05009234	0x92340000	
0x04923456	-0x12345600	High bit set (0x80 in 0x92).
0x04123456	0x12345600	Inverse of above; no high bit.

Difficulty 1, the minimum allowed difficulty, is represented on mainnet and the current testnet by the nBits value 0x1d00ffff. Regtest mode uses a different difficulty 1 value—0x207fffff, the highest possible value below uint32_max which can be encoded; this allows near-instant building of blocks in regtest mode.

Serialized Blocks

Under current consensus rules, a block is not valid unless its serialized size is less than or equal to 1 MB. All fields described below are counted towards the serialized size.

Bytes	Name	Data Type	Description
80	block header	block_header	The block header in the format described in the block header section.
Varies	txn_count	compactSize uint	The total number of transactions in this block, including the coinbase transaction.
Varies	txns	raw transaction	Every transaction in this block, one after another, in raw transaction format. Transactions must appear in the data stream in the same order their TXIDs appeared in the first row of the merkle tree. See the merkle tree section for details.

The first transaction in a block must be a coinbase transaction which should collect and spend any transaction fees paid by transactions included in this block.

All blocks with a block height less than 6,930,000 are entitled to receive a block subsidy of newly created bitcoin value, which also should be spent in the coinbase transaction. (The block subsidy started at 50 bitcoins and is being halved every 210,000 blocks—approximately once every four years. As of November 2014, it's 25 bitcoins.)

Together, the transaction fees and block subsidy are called the block reward. A coinbase transaction is invalid if it tries to spend more value than is available from the block reward.

Transactions

The following subsections briefly document core transaction details.

OP Codes

The op codes used in the pubkey scripts of standard transactions are:

- Various data pushing op codes from 0x00 to 0x4e (1–78). These aren't typically shown in examples, but they must be used to push signatures and public keys onto the stack. See the link below this list for a description.
- `OP_TRUE`/`OP_1` (0x51) and `OP_2` through `OP_16` (0x52–0x60), which push the values 1 through 16 to the stack.
- `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` pushes a copy of the topmost stack item on to the stack.
- `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` 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` consumes the topmost item on the stack. If that item is zero (false) it terminates the script in failure.
- `OP_EQUALVERIFY` runs `OP_EQUAL` and then `OP_VERIFY` in sequence.
- `OP_CHECKMULTISIG` consumes 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 Bitcoin Core implementation. This value is not used, 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. See the `OP_CHECKMULTISIG` warning below for more details.
- `OP_RETURN` terminates the script in failure when executed.

A complete list of OP codes can be found on the Bitcoin Wiki Script Page, with an authoritative list in the `opcodetype` enum of the Bitcoin Core script header file

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

⚠ `OP_CHECKMULTISIG` **warning:** The multisig verification process described above 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. For example, the following combined signature and pubkey script will produce the stack and comparisons shown:

```
OP_0 <A sig> <B sig> OP_2 <A pubkey> <B pubkey> <C pubkey> OP_3

Sig Stack       Pubkey Stack  (Actually a single stack)
---------       ------------
B sig           C pubkey
A sig           B pubkey
OP_0            A pubkey

1. B sig compared to C pubkey (no match)
2. B sig compared to B pubkey (match #1)
3. A sig compared to A pubkey (match #2)

Success: two matches found
```

But reversing the order of the signatures with everything else the same will fail, as shown below:

```
OP_0 <B sig> <A sig> OP_2 <A pubkey> <B pubkey> <C pubkey> OP_3

Sig Stack       Pubkey Stack  (Actually a single stack)
---------       ------------
A sig           C pubkey
B sig           B pubkey
OP_0            A pubkey

1. A sig compared to C pubkey (no match)
2. A sig compared to B pubkey (no match)

Failure, aborted: two signature matches required but none found so
                  far, and there's only one pubkey remaining
```

Address Conversion

The hashes used in P2PKH and P2SH outputs are commonly encoded as Bitcoin 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 (described in a following sub-section). Taking the resulting hash:

1. Add an address version byte in front of the hash. The version bytes commonly used by Bitcoin are:
   - 0x00 for P2PKH addresses on the main Bitcoin network (mainnet)
   - 0x6f for P2PKH addresses on the Bitcoin 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)`

Bitcoin's base58 encoding, called Base58Check may not match other implementations. Tier Nolan provided the following example encoding algorithm to the Bitcoin Wiki Base58Check encoding page under the Creative Commons Attribution 3.0 license:

```
code_string = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz"
x = convert_bytes_to_big_integer(hash_result)

output_string = ""

while(x > 0)
   {
       (x, remainder) = divide(x, 58)
       output_string.append(code_string[remainder])
   }

repeat(number_of_leading_zero_bytes_in_hash)
   {
   output_string.append(code_string[0]);
   }

output_string.reverse();
```

Bitcoin's own code can be traced using the base58 header file.

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

Bitcoin 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.

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

As of Bitcoin Core 0.9.3 (October 2014), all transactions use the version 1 format described below. (Note: transactions in the block chain are allowed to list a higher version number to permit soft forks, but they are treated as version 1 transactions by current software.)

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. See the locktime parsing rules.

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 the CompactSize section.

TxIn: A Transaction Input (Non-Coinbase)

Each non-coinbase input spends an outpoint from a previous transaction. (Coinbase inputs are described separately after the example section below.)

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[]	A script-language script which satisfies the conditions placed in the outpoint's pubkey script. Should only contain data pushes; see the signature script modification warning.
4	sequence	uint32_t	Sequence number; see sequence number. Default for Bitcoin Core and almost all other programs is 0xffffffff.

Outpoint: The Specific Part Of A Specific Output

Because a single transaction can include multiple outputs, the outpoint structure includes both a TXID and an output index number to refer to 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: A 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.

Example

The sample raw transaction itemized below is the one created in the Simple Raw Transaction section of the Developer Examples. It spends a previous pay-to-pubkey output by paying to a new pay-to-pubkey-hash (P2PKH) output.

```
01000000 .................................. Version
```

```
01 ......................................... Number of inputs
|
| 7b1eabe0209b1fe794124575ef807057
| c77ada2138ae4fa8d6c4de0398a14f3f ......... Outpoint TXID
| 00000000 ................................. Outpoint index number
|
| 49 ....................................... Bytes in sig. script: 73
| | 48 ..................................... Push 72 bytes as data
| | | 30450221008949f0cb400094ad2b5eb3
| | | 99d59d01c14d73d8fe6e96df1a7150de
| | | b388ab8935022079656090d7f6bac4c9
| | | a94e0aad311a4268e082a725f8aeae05
| | | 73fb12ff866a5f01 ..................... Secp256k1 signature
|
| ffffffff ................................. Sequence number: UINT32_MAX

01 ......................................... Number of outputs
| f0ca052a01000000 ......................... Satoshis (49.99990000 BTC)
|
| 19 ....................................... Bytes in pubkey script: 25
| | 76 ..................................... OP_DUP
| | a9 ..................................... OP_HASH160
| | 14 ..................................... Push 20 bytes as data
| | | cbc20a7664f2f69e5355aa427045bc15
| | | e7c6c772 ............................. PubKey hash
| | 88 ..................................... OP_EQUALVERIFY
| | ac ..................................... OP_CHECKSIG

00000000 ................................... locktime: 0 (a block height)
```

Coinbase Input: The Input Of The First Transaction In A Block

The first transaction in a block, called the coinbase transaction, must have exactly one input, called a coinbase. The coinbase input currently has the following format.

Bytes	Name	Data Type	Description
32	hash (null)	char[32]	A 32-byte null, as a coinbase has no previous outpoint.
4	index (UINT32_MAX)	uint32_t	0xffffffff, as a coinbase has no previous outpoint.
Varies	script bytes	compactSize uint	The number of bytes in the coinbase script, up to a maximum of 100 bytes.
Varies (4)	height	script	The block height of this block as required by BIP34. Uses script language: starts with a data-pushing op code that indicates how many bytes to push to the stack followed by the block height as a little-endian unsigned integer. This script must be as short as possible, otherwise it may be rejected. The data-pushing op code will be 0x03 and the total size four bytes until block 16,777,216 about 300 years from now.
Varies	coinbase script	*None*	The coinbase field: Arbitrary data not exceeding 100 bytes minus the (4) height bytes. Miners commonly place an extra nonce in this field to update the block header merkle root during hashing.
4	sequence	uint32_t	Sequence number; see sequence number.

Most (but not all) blocks prior to block height 227,836 used block version 1 which did not require the height parameter to be prefixed to the coinbase script. The block height parameter is now required.

Although the coinbase script is arbitrary data, if it includes the bytes used by any signature-checking operations such as `OP_CHECKSIG`, those signature checks will be counted as signature operations (sigops) towards the block's sigop limit. To avoid this, you can prefix all data with the appropriate push operation.

An itemized coinbase transaction:

```
01000000 .............................. Version

01 .................................... Number of inputs
| 00000000000000000000000000000000
| 00000000000000000000000000000000 ...  Previous outpoint TXID
| ffffffff ............................ Previous outpoint index
|
| 29 .................................. Bytes in coinbase
| |
| | 03 ................................ Bytes in height
| | | 4e0105 .......................... Height: 328014
| |
| | 062f503253482f0472d35454085fffed
| | f2400000f90f54696d65202620486561
| | 6c74682021 ........................ Arbitrary data
| 00000000 ............................ Sequence

01 .................................... Output count
| 2c37449500000000 .................... Satoshis (25.04275756 BTC)
| 1976a914a09be8040cbf399926aeb1f4
| 70c37d1341f3b46588ac ................ P2PKH script
| 00000000 ............................ Locktime
```

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.

Bitcoin Core code and this document refers to these variable length integers as compactSize. Many other documents refer to them as var_int or varInt, but this risks conflation with other variable-length integer encodings—such as the CVarInt class used in Bitcoin Core for serializing data to disk. 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.

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

For example, the number 515 is encoded as 0xfd0302.

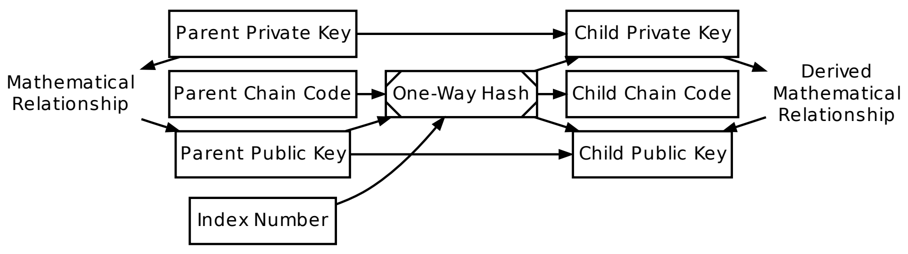
Wallets

Deterministic Wallet Formats

Type 1: Single Chain Wallets

Type 1 deterministic wallets are the simpler of the two, which can create a single series of keys from a single seed. A primary weakness is that if the seed is leaked, all funds are compromised, and wallet sharing is extremely limited.

Type 2: Hierarchical Deterministic (HD) Wallets



Normal Hierarchical Deterministic (HD) Key Derivation (BIP32)

For an overview of HD wallets, please see the developer guide section. For details, please see BIP32.

P2P Network

This section describes the Bitcoin P2P network protocol (but it is not a specification). It does not describe the discontinued direct IP-to-IP payment protocol, the BIP70 payment protocol, the GetBlockTemplate mining protocol, or any network protocol never implemented in an official version of Bitcoin Core.

All peer-to-peer communication occurs entirely over TCP.

Note: unless their description says otherwise, all multi-byte integers mentioned in this section are transmitted in little-endian order.

Constants And Defaults

The following constants and defaults are taken from Bitcoin Core's chainparams.cpp source code file.

Network	Default Port	Start String	Max nBits
Mainnet	8333	0xf9beb4d9	0x1d00ffff
Testnet	18333	0x0b110907	0x1d00ffff
Regtest	18444	0xfabfb5da	0x207fffff

Note: the testnet start string and nBits above are for testnet3; the original testnet used a different string and higher (less difficult) nBits.

Command line parameters can change what port a node listens on (see `-help`). Start strings are hardcoded constants that appear at the start of all messages sent on the Bitcoin network; they may also appear in data files such as Bitcoin Core's block database. The nBits displayed above are in big-endian order; they're sent over the network in little-endian order.

Bitcoin Core's chainparams.cpp also includes other constants useful to programs, such as the hash of the genesis blocks for the different networks as well as the alert keys for mainnet and testnet.

Protocol Versions

The table below lists some notable versions of the P2P network protocol, with the most recent versions listed first. (If you know of a protocol version that implemented a major change but which is not listed here, please open an issue.)

Version	Implementation	Major Changes
70002	Bitcoin Core 0.9.0 (Mar 2014)	• Send multiple `inv` messages in response to a `mempool` message if necessary BIP61: • Added `reject` message
70001	Bitcoin Core 0.8.0 (Feb 2013)	• Added `notfound` message. BIP37: • Added `filterload` message. • Added `filteradd` message. • Added `filterclear` message. • Added `merkleblock` message. • Added relay field to `version` message • Added `MSG_FILTERED_BLOCK` inventory type to `getdata` message.
60002	Bitcoin Core 0.7.0 (Sep 2012)	BIP35: • Added `mempool` message. • Extended `getdata` message to allow download of memory pool transactions
60001	Bitcoin Core 0.6.1 (May 2012)	BIP31: • Added nonce field to `ping` message • Added `pong` message
60000	Bitcoin Core 0.6.0 (Mar 2012)	BIP14: • Separated protocol version from Bitcoin Core version
31800	Bitcoin Core 0.3.18 (Dec 2010)	• Added `getheaders` message and `headers` message.
31402	Bitcoin Core 0.3.15 (Oct 2010)	• Added time field to `addr` message.
311	Bitcoin Core 0.3.11 (Aug 2010)	• Added `alert` message.
209	Bitcoin Core 0.2.9 (May 2010)	• Added checksum field to message headers.
106	Bitcoin Core 0.1.6 (Oct 2009)	• Added receive IP address fields to `version` message.

Message Headers

All messages in the network protocol use the same container format, which provides a required multi-field header and an optional payload. The header format is:

Bytes	Name	Data Type	Description
4	start string	char[4]	Magic bytes indicating the originating network; used to seek to next message when stream state is unknown.
	command		ASCII string which identifies what message type is contained in the payload. Followed by nulls

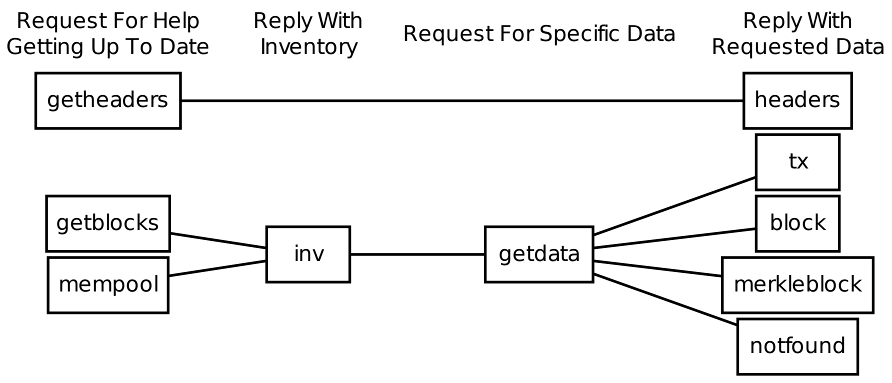
12	name	char[12]	(0x00) to pad out byte count; for example: `version\0\0\0\0\0`.
4	payload size	uint32_t	Number of bytes in payload. The current maximum number of bytes (`MAX_SIZE`) allowed in the payload by Bitcoin Core is 32 MiB—messages with a payload size larger than this will be dropped or rejected.
4	checksum	char[4]	*Added in protocol version 209.* First 4 bytes of SHA256(SHA256(payload)) in internal byte order. If payload is empty, as in `verack` and `getaddr` messages, the checksum is always 0x5df6e0e2 (SHA256(SHA256(<empty string>))).

The following example is an annotated hex dump of a mainnet message header from a `verack` message which has no payload.

```
f9beb4d9 ................... Start string: Mainnet
76657261636b000000000000 ... Command name: verack + null padding
00000000 ................... Byte count: 0
5df6e0e2 ................... Checksum: SHA256(SHA256(<empty>))
```

Data Messages

The following network messages all request or provide data related to transactions and blocks.



Overview Of P2P Protocol Data Request And Reply Messages

Many of the data messages use inventories as unique identifiers for transactions and blocks. Inventories have a simple 36-byte structure:

Bytes	Name	Data Type	Description
4	type identifier	uint32_t	The type of object which was hashed. See list of type identifiers below.
32	hash	char[32]	SHA256(SHA256()) hash of the object in internal byte order.

The currently-available type identifiers are:

Type Identifier	Name	Description
1	`MSG_TX`	The hash is a TXID.
2	`MSG_BLOCK`	The hash is of a block header.

3	`MSG_FILTERED_BLOCK`	The hash is of a block header; identical to `MSG_BLOCK`. When used in a `getdata` message, this indicates the response should be a `merkleblock` message rather than a `block` message (but this only works if a bloom filter was previously configured). **Only for use in `getdata` messages.**

Type identifier zero and type identifiers greater than three are reserved for future implementations. Bitcoin Core ignores all inventories with one of these unknown types.

Block

The `block` message transmits a single serialized block in the format described in the serialized blocks section. See that section for an example hexdump. It can be sent for two different reasons:

1. **GetData Response:** Nodes will always send it in response to a `getdata` message that requests the block with an inventory type of `MSG_BLOCK` (provided the node has that block available for relay).
2. **Unsolicited:** Some miners will send unsolicited `block` messages broadcasting their newly-mined blocks to all of their peers. Many mining pools do the same thing, although some may be misconfigured to send the block from multiple nodes, possibly sending the same block to some peers more than once.

GetBlocks

The `getblocks` message requests an `inv` message that provides block header hashes starting from a particular point in the block chain. It allows a peer which has been disconnected or started for the first time to get the data it needs to request the blocks it hasn't seen.

Peers which have been disconnected may have stale blocks in their locally-stored block chain, so the `getblocks` message allows the requesting peer to provide the receiving peer with multiple header hashes at various heights on their local chain. This allows the receiving peer to find, within that list, the last header hash they had in common and reply with all subsequent header hashes.

Note: the receiving peer itself may respond with an `inv` message containing header hashes of stale blocks. It is up to the requesting peer to poll all of its peers to find the best block chain.

If the receiving peer does not find a common header hash within the list, it will assume the last common block was the genesis block (block zero), so it will reply with in `inv` message containing header hashes starting with block one (the first block after the genesis block).

Bytes	Name	Data Type	Description
4	version	uint32_t	The protocol version number; the same as sent in the `version` message.
Varies	hash count	compactSize uint	The number of header hashes provided not including the stop hash. There is no limit except that the byte size of the entire message must be below the `MAX_SIZE` limit; typically from 1 to 200 hashes are sent.
Varies	block header hashes	char[32]	One or more block header hashes (32 bytes each) in internal byte order. Hashes should be provided in reverse order of block height, so highest-height hashes are listed first and lowest-height hashes are listed last.
32	stop hash	char[32]	The header hash of the last header hash being requested; set to all zeroes to request an `inv` message with all subsequent header hashes (a maximum of 500 will be sent as a reply to this message; if you need more than 500, you will need to send another `getblocks` message with a higher-height header hash as the first entry in block header hash field).

The following annotated hexdump shows a `getblocks` message. (The message header has been omitted.)

```
71110100 ........................... Protocol version: 70001
02 ................................. Hash count: 2

d39f608a7775b537729884d4e6633bb2
105e55a16a14d31b0000000000000000 ... Hash #1

5c3e6403d40837110a2e8afb602b1c01
714bda7ce23bea0a0000000000000000 ... Hash #2

00000000000000000000000000000000
00000000000000000000000000000000 ... Stop hash
```

GetData

The `getdata` message requests one or more data objects from another node. The objects are requested by an inventory, which the requesting node typically previously received by way of an `inv` message.

The response to a `getdata` message can be a `tx` message, `block` message, `merkleblock` message, or `notfound` message.

This message cannot be used to request arbitrary data, such as historic transactions no longer in the memory pool or relay set. Full nodes may not even be able to provide older blocks if they've pruned old transactions from their block database. For this reason, the `getdata` message should usually only be used to request data from a node which previously advertised it had that data by sending an `inv` message.

The format and maximum size limitations of the `getdata` message are identical to the `inv` message; only the message header differs.

GetHeaders

Added in protocol version 31800.

The `getheaders` message requests a `headers` message that provides block headers starting from a particular point in the block chain. It allows a peer which has been disconnected or started for the first time to get the headers it hasn't seen yet.

The `getheaders` message is nearly identical to the `getblocks` message, with one minor difference: the `inv` reply to the `getblocks` message will include no more than 500 block header hashes; the `headers` reply to the `getheaders` message will include as many as 2,000 block headers.

Headers

Added in protocol version 31800.

The `headers` message sends one or more block headers to a node which previously requested certain headers with a `getheaders` message.

Bytes	Name	Data Type	Description
Varies	count	compactSize uint	Number of block headers up to a maximum of 2,000. Note: headers-first sync assumes the sending node will send the maximum number of headers whenever possible.
Varies	headers	block_header	Block headers: each 80-byte block header is in the format described in the block headers section with an additional 0x00 suffixed. This 0x00 is called the transaction count, but because the headers message doesn't include any transactions, the transaction count is always zero.

The following annotated hexdump shows a `headers` message. (The message header has been omitted.)

```
01 ................................. Header count: 1

02000000 ........................... Block version: 2
b6ff0b1b1680a2862a30ca44d346d9e8
910d334beb48ca0c0000000000000000 ... Hash of previous block's header
9d10aa52ee949386ca9385695f04ede2
70dda20810decd12bc9b048aaab31471 ... Merkle root
24d95a54 ........................... Unix time: 1415239972
30c31b18 ........................... Target (bits)
fe9f0864 ........................... Nonce

00 ................................. Transaction count (0x00)
```

Inv

The `inv` message (inventory message) transmits one or more inventories of objects known to the transmitting peer. It can be sent unsolicited to announce new transactions or blocks, or it can be sent in reply to a `getblocks` message or `mempool` message.

The receiving peer can compare the inventories from an `inv` message against the inventories it has already seen, and then use a follow-up message to request unseen objects.

Bytes	Name	Data Type	Description
Varies	count	compactSize uint	The number of inventory entries.
Varies	inventory	inventory	One or more inventory entries up to a maximum of 50,000 entries.

The following annotated hexdump shows an `inv` message with two inventory entries. (The message header has been omitted.)

```
02 ................................. Count: 2

01000000 ........................... Type: MSG_TX
de55ffd709ac1f5dc509a0925d0b1fc4
42ca034f224732e429081da1b621f55a ... Hash (TXID)

01000000 ........................... Type: MSG_TX
91d36d997037e08018262978766f24b8
a055aaf1d872e94ae85e9817b2c68dc7 ... Hash (TXID)
```

MemPool

Added in protocol version 60002.

The `mempool` message requests the TXIDs of transactions that the receiving node has verified as valid but which have not yet appeared in a block. That is, transactions which are in the receiving node's memory pool. The response to the `mempool` message is one or more `inv` messages containing the TXIDs in the usual inventory format.

Sending the `mempool` message is mostly useful when a program first connects to the network. Full nodes can use it to quickly gather most or all of the unconfirmed transactions available on the network; this is especially useful for miners trying to gather transactions for their transaction fees. SPV clients can set a filter before sending a `mempool` to only receive transactions that match that filter; this allows a recently-started client to get most or all unconfirmed transactions related to its wallet.

The `inv` response to the `mempool` message is, at best, one node's view of the network—not a complete list of unconfirmed transactions on the network. Here are some additional reasons the list might not be complete:

- Before Bitcoin Core 0.9.0, the response to the `mempool` message was only one `inv` message. An `inv` message is limited to 50,000 inventories, so a node with a memory pool larger than 50,000 entries would not send everything. Later versions of Bitcoin Core send as many `inv` messages as needed to reference its complete memory pool.

- The `mempool` message is not currently fully compatible with the `filterload` message's `BLOOM_UPDATE_ALL` and `BLOOM_UPDATE_P2PUBKEY_ONLY` flags. Mempool transactions are not sorted like in-block transactions, so a transaction (tx2) spending an output can appear before the transaction (tx1) containing that output, which means the automatic filter update mechanism won't operate until the second-appearing transaction (tx1) is seen—missing the first-appearing transaction (tx2). It has been proposed in Bitcoin Core issue #2381 that the transactions should be sorted before being processed by the filter.

There is no payload in a `mempool` message. See the message header section for an example of a message without a payload.

MerkleBlock

Added in protocol version 70001 as described by BIP37.

The `merkleblock` message is a reply to a `getdata` message which requested a block using the inventory type `MSG_MERKLEBLOCK`. It is only part of the reply: if any matching transactions are found, they will be sent separately as `tx` messages.

If a filter has been previously set with the `filterload` message, the `merkleblock` message will contain the TXIDs of any transactions in the requested block that matched the filter, as well as any parts of the block's merkle tree necessary to connect those transactions to the block header's merkle root. The message also contains a complete copy of the block header to allow the client to hash it and confirm its proof of work.

Bytes	Name	Data Type	Description
80	block header	block_header	The block header in the format described in the block header section.
4	transaction count	uint32_t	The number of transactions in the block (including ones that don't match the filter).
Varies	hash count	compactSize uint	The number of hashes in the following field.
Varies	hashes	char[32]	One or more hashes of both transactions and merkle nodes in internal byte order. Each hash is 32 bits.
Varies	flag byte count	compactSize uint	The number of flag bytes in the following field.
Varies	flags	byte[]	A sequence of bits packed eight in a byte with the least significant bit first. May be padded to the nearest byte boundary but must not contain any more bits than that. Used to assign the hashes to particular nodes in the merkle tree as described below.

The annotated hexdump below shows a `merkleblock` message which corresponds to the examples below. (The message header has been omitted.)

```
01000000 ........................... Block version: 1
82bb869cf3a793432a66e826e05a6fc3
7469f8efb7421dc88067010000000000 ... Hash of previous block's header
7f16c5962e8bd963659c793ce370d95f
093bc7e367117b3c30c1f8fdd0d97287 ... Merkle root
76381b4d ........................... Time: 1293629558
4c86041b ........................... nBits: 0x04864c * 256**(0x1b-3)
554b8529 ........................... Nonce

07000000 ........................... Transaction count: 7
04 ................................. Hash count: 4

3612262624047ee87660be1a707519a4
43b1c1ce3d248cbfc6c15870f6c5daa2 ... Hash #1
019f5b01d4195ecbc9398fbf3c3b1fa9
```

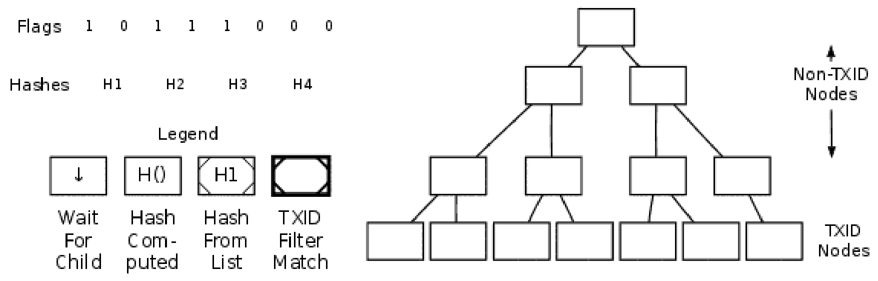
```
bb3183301d7a1fb3bd174fcfa40a2b65 ... Hash #2
41ed70551dd7e841883ab8f0b16bf041
76b7d1480e4f0af9f3d4c3595768d068 ... Hash #3
20d2a7bc994987302e5b1ac80fc425fe
25f8b63169ea78e68fbaaefa59379bbf ... Hash #4

01 ................................ Flag bytes: 1
1d ................................ Flags: 1 0 1 1 1 0 0 0
```

Note: when fully decoded, the above `merkleblock` message provided the TXID for a single transaction that matched the filter. In the network traffic dump this output was taken from, the full transaction belonging to that TXID was sent immediately after the `merkleblock` message as a `tx` message.

Parsing A MerkleBlock Message

As seen in the annotated hexdump above, the `merkleblock` message provides three special data types: a transaction count, a list of hashes, and a list of one-bit flags.

You can use the transaction count to construct an empty merkle tree. We'll call each entry in the tree a node; on the bottom are TXID nodes—the hashes for these nodes are TXIDs; the remaining nodes (including the merkle root) are non-TXID nodes—they may actually have the same hash as a TXID, but we treat them differently.



Keep the hashes and flags in the order they appear in the `merkleblock` message. When we say "next flag" or "next hash", we mean the next flag or hash on the list, even if it's the first one we've used so far.

Start with the merkle root node and the first flag. The table below describes how to evaluate a flag based on whether the node being processed is a TXID node or a non-TXID node. Once you apply a flag to a node, never apply another flag to that same node or reuse that same flag again.

Flag	TXID Node	Non-TXID Node
0	Use the next hash as this node's TXID, but this transaction didn't match the filter.	Use the next hash as this node's hash. Don't process any descendant nodes.
1	Use the next hash as this node's TXID, and mark this transaction as matching the filter.	The hash needs to be computed. Process the left child node to get its hash; process the right child node to get its hash; then concatenate the two hashes as 64 raw bytes and hash them to get this node's hash.

Any time you begin processing a node for the first time, evaluate the next flag. Never use a flag at any other time.

When processing a child node, you may need to process its children (the grandchildren of the original node) or further-descended nodes before returning to the parent node. This is expected—keep processing depth first until you reach a TXID node or a non-TXID node with a flag of 0.

After you process a TXID node or a non-TXID node with a flag of 0, stop processing flags and begin to ascend the tree. As you ascend, compute the hash of any nodes for which you now have both child hashes or for which you now have the sole child hash. See the merkle tree section for hashing instructions. If you reach a node where only the left hash is known, descend into its right child (if present) and further descendants as necessary.

However, if you find a node whose left and right children both have the same hash, fail. This is related to CVE-2012-2459.

Continue descending and ascending until you have enough information to obtain the hash of the merkle root node. If you run out of flags or hashes before that condition is reached, fail. Then perform the following checks (order doesn't matter):

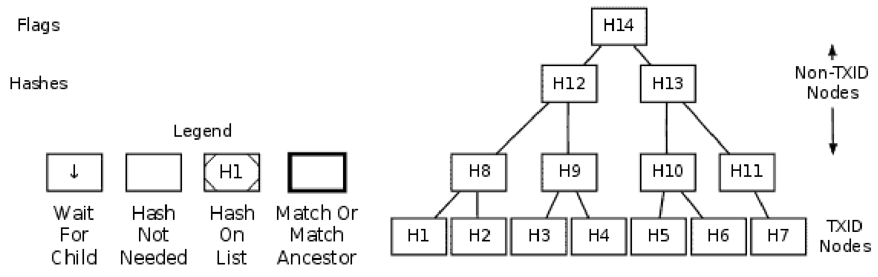
- Fail if there are unused hashes in the hashes list.
- Fail if there are unused flag bits—except for the minimum number of bits necessary to pad up to the next full byte.
- Fail if the hash of the merkle root node is not identical to the merkle root in the block header.
- Fail if the block header is invalid. Remember to ensure that the hash of the header is less than or equal to the target threshold encoded by the nBits header field. Your program should also, of course, attempt to ensure the header belongs to the best block chain and that the user knows how many confirmations this block has.

For a detailed example of parsing a `merkleblock` message, please see the corresponding merkle block examples section.

Creating A MerkleBlock Message

It's easier to understand how to create a `merkleblock` message after you understand how to parse an already-created message, so we recommend you read the parsing section above first.

Create a complete merkle tree with TXIDs on the bottom row and all the other hashes calculated up to the merkle root on the top row. For each transaction that matches the filter, track its TXID node and all of its ancestor nodes.



Start processing the tree with the merkle root node. The table below describes how to process both TXID nodes and non-TXID nodes based on whether the node is a match, a match ancestor, or neither a match nor a match ancestor.

	TXID Node	Non-TXID Node
Neither Match Nor Match Ancestor	Append a 0 to the flag list; append this node's TXID to the hash list.	Append a 0 to the flag list; append this node's hash to the hash list. Do not descend into its child nodes.
Match Or Match Ancestor	Append a 1 to the flag list; append this node's TXID to the hash list.	Append a 1 to the flag list; process the left child node. Then, if the node has a right child, process the right child. Do not append a hash to the hash list for this node.

Any time you begin processing a node for the first time, a flag should be appended to the flag list. Never put a flag on the list at any other time, except when processing is complete to pad out the flag list to a byte boundary.

When processing a child node, you may need to process its children (the grandchildren of the original node) or further-descended nodes before returning to the parent node. This is expected—keep processing depth first until you reach a TXID node or a node which is neither a TXID nor a match ancestor.

After you process a TXID node or a node which is neither a TXID nor a match ancestor, stop processing and begin to ascend the tree until you find a node with a right child you haven't processed yet. Descend into that right child and process it.

After you fully process the merkle root node according to the instructions in the table above, processing is complete. Pad your flag list to a byte boundary and construct the `merkleblock` message using the template near the beginning of this subsection.

NotFound

Added in protocol version 70001.

The `notfound` message is a reply to a `getdata` message which requested an object the receiving node does not have available for relay. (Nodes are not expected to relay historic transactions which are no longer in the memory pool or relay set. Nodes may also have pruned spent transactions from older blocks, making them unable to send those blocks.)

The format and maximum size limitations of the `notfound` message are identical to the `inv` message; only the message header differs.

Tx

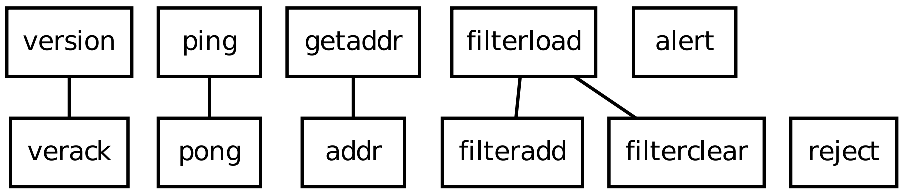
The `tx` message transmits a single transaction in the raw transaction format. It can be sent in a variety of situations;

- **Transaction Response:** Bitcoin Core and BitcoinJ will send it in response to a `getdata` message that requests the transaction with an inventory type of `MSG_TX`.
- **MerkleBlock Response:** Bitcoin Core will send it in response to a `getdata` message that requests a merkle block with an inventory type of `MSG_MERKLEBLOCK`. (This is in addition to sending a `merkleblock` message.) Each `tx` message in this case provides a matched transaction from that block.
- **Unsolicited:** BitcoinJ will send a `tx` message unsolicited for transactions it originates.

For an example hexdump of the raw transaction format, see the raw transaction section.

Control Messages

The following network messages all help control the connection between two peers or allow them to advise each other about the rest of the network.



Overview Of P2P Protocol Control And Advisory Messages

Note that almost none of the control messages are authenticated in any way, meaning they can contain incorrect or intentionally harmful information. In addition, this section does not yet cover P2P protocol operation over the Tor network; if you would like to contribute information about Tor, please open an issue.

Addr

The `addr` (IP address) message relays connection information for peers on the network. Each peer which wants to accept incoming connections creates an `addr` message providing its connection information and then sends that message to its peers unsolicited. Some of its peers send that information to their peers (also unsolicited), some of which further distribute it, allowing decentralized peer discovery for any program already on the network.

An `addr` message may also be sent in response to a `getaddr` message.

Bytes	Name	Data Type	Description
Varies	IP address count	compactSize uint	The number of IP address entries up to a maximum of 1,000.
Varies	IP addresses	network IP address	IP address entries. See the table below for the format of a Bitcoin network IP address.

Each encapsulated network IP address currently uses the following structure:

Bytes	Name	Data Type	Description
4	time	uint32	*Added in protocol version 31402.* A time in Unix epoch time format. Nodes advertising their own IP address set this to the current time. Nodes advertising IP addresses they've connected to set this to the last time they connected to that node. Other nodes just relaying the IP address should not change the time. Nodes can use the time field to avoid relaying old `addr` messages. Malicious nodes may change times or even set them in the future.
8	services	uint64_t	The services the node advertised in its `version` message.
16	IP address	char	IPv6 address in **big endian byte order**. IPv4 addresses can be provided as IPv4-mapped IPv6 addresses
2	port	uint16_t	Port number in **big endian byte order**. Note that Bitcoin Core will only connect to nodes with non-standard port numbers as a last resort for finding peers. This is to prevent anyone from trying to use the network to disrupt non-Bitcoin services that run on other ports.

The following annotated hexdump shows part of an `addr` message. (The message header has been omitted and the actual IP address has been replaced with a RFC5737 reserved IP address.)

```
fde803 ............................ Address count: 1000

d91f4854 .......................... Epoch time: 1414012889
0100000000000000 .................. Service bits: 01 (network node)
00000000000000000000ffffc0000233 ... IP Address: ::ffff:192.0.2.51
208d .............................. Port: 8333

[...] ............................. (999 more addresses omitted)
```

Alert

Added in protocol version 311.

The `alert` message warns nodes of problems that may affect them or the rest of the network. Each `alert` message is signed using a key controlled by respected community members, mostly Bitcoin Core developers.

To ensure all nodes can validate and forward `alert` messages, encapsulation is used. Developers create an alert using the data structure appropriate for the versions of the software they want to notify; then they serialize that data and sign it. The serialized data and its signature make up the outer `alert` message—allowing nodes which don't understand the data structure to validate the signature and relay the alert to nodes which do understand it. The nodes which actually need the message can decode the serialized data to access the inner `alert` message.

The outer `alert` message has four fields:

Bytes	Name	Data Type	Description
Variable	alert bytes	compactSize uint	The number of bytes in following alert field.
Variable	alert	uchar	The serialized alert. See below for a description of the current alert format.
Variable	signature bytes	compactSize uint	The number of bytes in the following signature field.
Variable	signature	uchar	A DER-encoded ECDSA (secp256k1) signature of the alert signed with the developer's alert key.

Although designed to be easily upgraded, the format of the inner serialized alert has not changed since the `alert` message was first introduced in protocol version 311.

Bytes	Name	Data Type	Description
4	version	int32_t	Alert format version. Version 1 from protocol version 311 through at least protocol version 70002.
8	relayUntil	int64_t	The time beyond which nodes should stop relaying this alert. Unix epoch time format.
8	expiration	int64_t	The time beyond which this alert is no longer in effect and should be ignored. Unix epoch time format.
4	ID	int32_t	A unique ID number for this alert.
4	cancel	int32_t	All alerts with an ID number less than or equal to this number should be canceled: deleted and not accepted in the future.
Varies	setCancel count	compactSize uint	The number of IDs in the following setCancel field. May be zero.
Varies	setCancel	int32_t	Alert IDs which should be canceled. Each alert ID is a separate int32_t number.
4	minVer	int32_t	This alert only applies to protocol versions greater than or equal to this version. Nodes running other protocol versions should still relay it.
4	maxVer	int32_t	This alert only applies to protocol versions less than or equal to this version. Nodes running other protocol versions should still relay it.
Varies	user_agent count	compactSize uint	The number of user agent strings in the following setUser_agent field. May be zero.
Varies	setUser_agent	compactSize uint + string	If this field is empty, it has no effect on the alert. If there is at least one entry is this field, this alert only applies to programs with a user agent that exactly matches one of the strings in this field. Each entry in this field is a compactSize uint followed by a string—the uint indicates how many bytes are in the following string. This field was originally called setSubVer; since BIP14, it applies to user agent strings as defined in the `version` message.
4	priority	int32_t	Relative priority compared to other alerts.
Varies	comment bytes	compactSize uint	The number of bytes in the following comment field. May be zero.

Varies	comment	string	A comment on the alert that is not displayed.
Varies	statusBar bytes	compactSize uint	The number of bytes in the following statusBar field. May be zero.
Varies	statusBar	string	The alert message that is displayed to the user.
Varies	reserved bytes	compactSize uint	The number of bytes in the following reserved field. May be zero.
Varies	reserved	string	Reserved for future use. Originally called RPC Error.

The annotated hexdump below shows an `alert` message. (The message header has been omitted.)

```
73 ................................ Bytes in encapsulated alert: 115
01000000 .......................... Version: 1
3766404f00000000 .................. RelayUntil: 1329620535
b305434f00000000 .................. Expiration: 1330917376

f2030000 .......................... ID: 1010
f1030000 .......................... Cancel: 1009
00 ................................ setCancel count: 0

10270000 .......................... MinVer: 10000
48ee0000 .......................... MaxVer: 61000
00 ................................ setUser_agent bytes: 0
64000000 .......................... Priority: 100

00 ................................ Bytes In Comment String: 0
46 ................................ Bytes in StatusBar String: 70
53656520626974636f696e2e6f72672f
666562323020696620796f7520686176
652074726f75626c6520636f6e6e6563
74696e67206166746572203230204665
627275617279 ...................... Status Bar String: "See [...]"
00 ................................ Bytes In Reserved String: 0

47 ................................ Bytes in signature: 71
30450221008389df45f0703f39ec8c1c
c42c13810ffcae14995bb648340219e3
53b63b53eb022009ec65e1c1aaeec1fd
334c6b684bde2b3f573060d5b70c3a46
723326e4e8a4f1 .................... Signature
```

Alert key compromise: Bitcoin Core's source code defines a particular set of alert parameters that can be used to notify users that the alert signing key has been compromised and that they should upgrade to get a new alert public key. Once a signed alert containing those parameters has been received, no other alerts can cancel or override it. See the `ProcessAlert()` function in the Bitcoin Core alert.cpp source code for the parameters of this message.

FilterAdd

Added in protocol version 70001 as described by BIP37.

The `filteradd` message tells the receiving peer to add a single element to a previously-set bloom filter, such as a new public key. The element is sent directly to the receiving peer; the peer then uses the parameters set in the `filterload` message to add the element to the bloom filter.

Because the element is sent directly to the receiving peer, there is no obfuscation of the element and none of the plausible-deniability privacy provided by the bloom filter. Clients that want to maintain greater privacy should recalculate the bloom filter themselves and send a new `filterload` message with the recalculated bloom filter.

Bytes	Name	Data Type	Description

Varies	element bytes	compactSize uint	The number of bytes in the following element field.
Varies	element	uint8_t[]	The element to add to the current filter. Maximum of 520 bytes, which is the maximum size of an element which can be pushed onto the stack in a pubkey or signature script. Elements must be sent in the byte order they would use when appearing in a raw transaction; for example, hashes should be sent in internal byte order.

Note: a `filteradd` message will not be accepted unless a filter was previously set with the `filterload` message.

The annotated hexdump below shows a `filteradd` message adding a TXID. (The message header has been omitted.) This TXID appears in the same block used for the example hexdump in the `merkleblock` message; if that `merkleblock` message is re-sent after sending this `filteradd` message, six hashes are returned instead of four.

```
20 ................................. Element bytes: 32
fdacf9b3eb077412e7a968d2e4f11b9a
9dee312d666187ed77ee7d26af16cb0b ... Element (A TXID)
```

FilterClear

Added in protocol version 70001 as described by BIP37.

The `filterclear` message tells the receiving peer to remove a previously-set bloom filter. This also undoes the effect of setting the relay field in the `version` message to 0, allowing unfiltered access to `inv` messages announcing new transactions.

Bitcoin Core does not require a `filterclear` message before a replacement filter is loaded with `filterload`. It also doesn't require a `filterload` message before a `filterclear` message.

There is no payload in a `filterclear` message. See the message header section for an example of a message without a payload.

FilterLoad

Added in protocol version 70001 as described by BIP37.

The `filterload` message tells the receiving peer to filter all relayed transactions and requested merkle blocks through the provided filter. This allows clients to receive transactions relevant to their wallet plus a configurable rate of false positive transactions which can provide plausible-deniability privacy.

Bytes	Name	Data Type	Description
Varies	nFilterBytes	uint8_t[]	Number of bytes in the following filter bit field.
Varies	filter	uint8_t[]	A bit field of arbitrary byte-aligned size. The maximum size is 36,000 bytes.
4	nHashFuncs	uint32_t	The number of hash functions to use in this filter. The maximum value allowed in this field is 50.
4	nTweak	uint32_t	An arbitrary value to add to the seed value in the hash function used by the bloom filter.
1	nFlags	uint8_t	A set of flags that control how outpoints corresponding to a matched pubkey script are added to the filter. See the table in the Updating A Bloom Filter subsection below.

The annotated hexdump below shows a `filterload` message. (The message header has been omitted.) For an example of how this payload was created, see the filterload example.

```
02 ......... Filter bytes: 2
b50f ....... Filter: 1010 1101 1111 0000
0b000000 ... nHashFuncs: 11
00000000 ... nTweak: 0/none
00 ......... nFlags: BLOOM_UPDATE_NONE
```

Initializing A Bloom Filter

Filters have two core parameters: the size of the bit field and the number of hash functions to run against each data element. The following formulas from BIP37 will allow you to automatically select appropriate values based on the number of elements you plan to insert into the filter (*n*) and the false positive rate (*p*) you desire to maintain plausible deniability.

- Size of the bit field in bytes (*nFilterBytes*), up to a maximum of 36,000: `(-1 / log(2)**2 * n * log(p)) / 8`
- Hash functions to use (*nHashFuncs*), up to a maximum of 50: `nFilterBytes * 8 / n * log(2)`

Note that the filter matches parts of transactions (transaction elements), so the false positive rate is relative to the number of elements checked—not the number of transactions checked. Each normal transaction has a minimum of four matchable elements (described in the comparison subsection below), so a filter with a false-positive rate of 1 percent will match about 4 percent of all transactions at a minimum.

According to BIP37, the formulas and limits described above provide support for bloom filters containing 20,000 items with a false positive rate of less than 0.1 percent or 10,000 items with a false positive rate of less than 0.0001 percent.

Once the size of the bit field is known, the bit field should be initialized as all zeroes.

Populating A Bloom Filter

The bloom filter is populated using between 1 and 50 unique hash functions (the number specified per filter by the *nHashFuncs* field). Instead of using up to 50 different hash function implementations, a single implementation is used with a unique seed value for each function.

The seed is `nHashNum * 0xfba4c795 + nTweak` as a *uint32_t*, where the values are:

- **nHashNum** is the sequence number for this hash function, starting at 0 for the first hash iteration and increasing up to the value of the *nHashFuncs* field (minus one) for the last hash iteration.
- **0xfba4c795** is a constant optimized to create large differences in the seed for different values of *nHashNum*.
- **nTweak** is a per-filter constant set by the client to require the use of an arbitrary set of hash functions.

If the seed resulting from the formula above is larger than four bytes, it must be truncated to its four most significant bytes (for example, `0x8967452301 & 0xffffffff → 0x67452301`).

The actual hash function implementation used is the 32-bit Murmur3 hash function.

⚠ **Warning:** the Murmur3 hash function has separate 32-bit and 64-bit versions that produce different results for the same input. Only the 32-bit Murmur3 version is used with Bitcoin bloom filters.

The data to be hashed can be any transaction element which the bloom filter can match. See the next subsection for the list of transaction elements checked against the filter. The largest element which can be matched is a script data push of 520 bytes, so the data should never exceed 520 bytes.

The example below from Bitcoin Core bloom.cpp combines all the steps above to create the hash function template. The seed is the first parameter; the data to be hashed is the second parameter. The result is a uint32_t modulo the size of the bit field in bits.

```
MurmurHash3(nHashNum * 0xFBA4C795 + nTweak, vDataToHash) % (vData.size() * 8)
```

Each data element to be added to the filter is hashed by *nHashFuncs* number of hash functions. Each time a hash function is run, the result will be the index number (*nIndex*) of a bit in the bit field. That bit must be set to 1. For example if the filter bit field was `00000000` and the result is 5, the revised filter bit field is `00000100` (the first bit is bit 0).

It is expected that sometimes the same index number will be returned more than once when populating the bit field; this does not affect the algorithm—after a bit is set to 1, it is never changed back to 0.

After all data elements have been added to the filter, each set of eight bits is converted into a little-endian byte. These bytes are the value of the *filter* field.

Comparing Transaction Elements To A Bloom Filter

To compare an arbitrary data element against the bloom filter, it is hashed using the same parameters used to create the bloom filter. Specifically, it is hashed *nHashFuncs* times, each time using the same *nTweak* provided in the filter, and the resulting output is modulo the size of the bit field provided in the *filter* field. After each hash is performed, the filter is checked to see if the bit at that indexed location is set. For example if the result of a hash is `5` and the filter is `01001110`, the bit is considered set.

If the result of every hash points to a set bit, the filter matches. If any of the results points to an unset bit, the filter does not match.

The following transaction elements are compared against bloom filters. All elements will be hashed in the byte order used in blocks (for example, TXIDs will be in internal byte order).

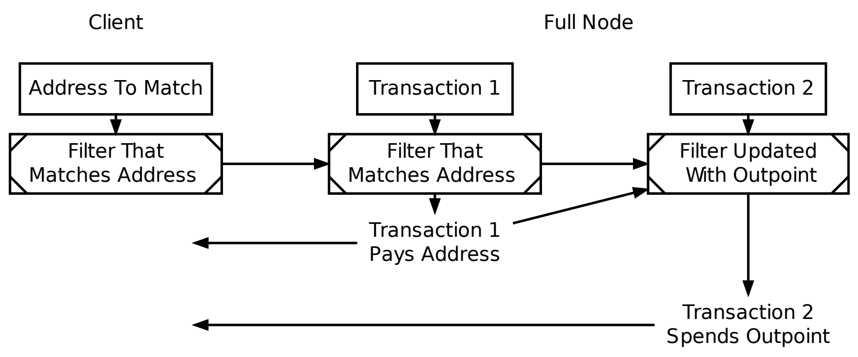
- **TXIDs:** the transaction's SHA256(SHA256()) hash.
- **Outpoints:** each 36-byte outpoint used this transaction's input section is individually compared to the filter.
- **Signature Script Data:** each element pushed onto the stack by a data-pushing op code in a signature script from this transaction is individually compared to the filter. This includes data elements present in P2SH redeem scripts when they are being spent.
- **PubKey Script Data:** each element pushed onto the the stack by a data-pushing op code in any pubkey script from this transaction is individually compared to the filter. (If a pubkey script element matches the filter, the filter will be immediately updated if the `BLOOM_UPDATE_ALL` flag was set; if the pubkey script is in the P2PKH format and matches the filter, the filter will be immediately updated if the `BLOOM_UPDATE_P2PUBKEY_ONLY` flag was set. See the subsection below for details.)

The following annotated hexdump of a transaction is from the raw transaction format section; the elements which would be checked by the filter are emphasized in bold. Note that this transaction's TXID (**`01000000017b1eab[...]`**) would also be checked, and that the outpoint TXID and index number below would be checked as a single 36-byte element.

```
01000000 .................................. Version

01 ........................................ Number of inputs
|
| 7b1eabe0209b1fe794124575ef807057
| c77ada2138ae4fa8d6c4de0398a14f3f ......... Outpoint TXID
| 00000000 ................................. Outpoint index number
|
| 49 ....................................... Bytes in sig. script: 73
| | 48 ..................................... Push 72 bytes as data
| | | 30450221008949f0cb400094ad2b5eb3
| | | 99d59d01c14d73d8fe6e96df1a7150de
| | | b388ab8935022079656090d7f6bac4c9
| | | a94e0aad311a4268e082a725f8aeae05
| | | 73fb12ff866a5f01 ..................... Secp256k1 signature
|
| ffffffff ................................. Sequence number: UINT32_MAX

01 ........................................ Number of outputs
| f0ca052a01000000 ......................... Satoshis (49.99990000 BTC)
|
| 19 ....................................... Bytes in pubkey script: 25
| | 76 ..................................... OP_DUP
| | a9 ..................................... OP_HASH160
| | 14 ..................................... Push 20 bytes as data
| | | cbc20a7664f2f69e5355aa427045bc15
| | | e7c6c772 ............................. PubKey hash
| | 88 ..................................... OP_EQUALVERIFY
| | ac ..................................... OP_CHECKSIG

00000000 .................................. locktime: 0 (a block height)
```

Updating A Bloom Filter

Clients will often want to track inputs that spend outputs (outpoints) relevant to their wallet, so the filterload field *nFlags* can be set to allow the filtering node to update the filter when a match is found. When the filtering node sees a pubkey script that pays a pubkey, address, or other data element matching the filter, the filtering node immediately updates the filter with the outpoint corresponding to that pubkey script.


Client
Full Node
Address To Match
Filter That Matches Address
Transaction 1
Filter That Matches Address
Transaction 2
Filter Updated With Outpoint
Transaction 1 Pays Address
Transaction 2 Spends Outpoint

Automatically Updating Bloom Filters To Track Relevant Transactions

If an input later spends that outpoint, the filter will match it, allowing the filtering node to tell the client that one of its transaction outputs has been spent.

The *nFlags* field has three allowed values:

Value	Name	Description
0	BLOOM_UPDATE_NONE	The filtering node should not update the filter.
1	BLOOM_UPDATE_ALL	If the filter matches any data element in a pubkey script, the corresponding outpoint is added to the filter.
2	BLOOM_UPDATE_P2PUBKEY_ONLY	If the filter matches any data element in a pubkey script and that script is either a P2PKH or non-P2SH pay-to-multisig script, the corresponding outpoint is added to the filter.

In addition, because the filter size stays the same even though additional elements are being added to it, the false positive rate increases. Each false positive can result in another element being added to the filter, creating a feedback loop that can (after a certain point) make the filter useless. For this reason, clients using automatic filter updates need to monitor the actual false positive rate and send a new filter when the rate gets too high.

GetAddr

The `getaddr` message requests an `addr` message from the receiving node, preferably one with lots of IP addresses of other receiving nodes. The transmitting node can use those IP addresses to quickly update its database of available nodes rather than waiting for unsolicited `addr` messages to arrive over time.

There is no payload in a `getaddr` message. See the message header section for an example of a message without a payload.

Ping

The `ping` message helps confirm that the receiving peer is still connected. If a TCP/IP error is encountered when sending the `ping` message (such as a connection timeout), the transmitting node can assume that the receiving node is disconnected. The response to a `ping` message is the `pong` message.

Before protocol version 60000, the `ping` message had no payload. As of protocol version 60001 and all later versions, the message includes a single field, the nonce.

Bytes	Name	Data Type	Description
8	nonce	uint64_t	*Added in protocol version 60000 as described by BIP31.* Random nonce assigned to this `ping` message. The responding `pong` message will include this nonce to identify the `ping` message to which it is replying.

The annotated hexdump below shows a `ping` message. (The message header has been omitted.)

```
0094102111e2af4d ... Nonce
```

Pong

Added in protocol version 60001 as described by BIP31.

The `pong` message replies to a `ping` message, proving to the pinging node that the ponging node is still alive. Bitcoin Core will, by default, disconnect from any clients which have not responded to a `ping` message within 20 minutes.

To allow nodes to keep track of latency, the `pong` message sends back the same nonce received in the `ping` message it is replying to.

The format of the `pong` message is identical to the `ping` message; only the message header differs.

Reject

Added in protocol version 70002 as described by BIP61.

The `reject` message informs the receiving node that one of its previous messages has been rejected.

Bytes	Name	Data Type	Description
Varies	message bytes	compactSize uint	The number of bytes in the following message field.
Varies	message	string	The type of message rejected as ASCII text *without null padding*. For example: “tx”, “block”, or “version”.
1	code	char	The reject message code. See the table below.
Varies	reason bytes	compactSize uint	The number of bytes in the following reason field. May be 0x00 if a text reason isn’t provided.
Varies	reason	string	The reason for the rejection in ASCII text. This should not be displayed to the user; it is only for debugging purposes.
Varies	extra data	*varies*	Optional additional data provided with the rejection. For example, most rejections of `tx` messages or `block` messages include the hash of the rejected transaction or block header. See the code table below.

The following table lists message reject codes. Codes are tied to the type of message they reply to; for example there is a 0x10 reject code for transactions and a 0x10 reject code for blocks.

Code	In Reply	Extra	Extra	Description

	To	Bytes	Type	
0x01	*any message*	0	N/A	Message could not be decoded. Be careful of `reject` message feedback loops where two peers each don’t understand each other’s `reject` messages and so keep sending them back and forth forever.
0x10	`block` message	32	char[32]	Block is invalid for some reason (invalid proof-of-work, invalid signature, etc). Extra data is the rejected block’s header hash.
0x10	`tx` message	32	char[32]	Transaction is invalid for some reason (invalid signature, output value greater than input, etc.). Extra data is the rejected transaction’s TXID.
0x11	`block` message	32	char[32]	The block uses a version that is no longer supported. Extra data is the rejected block’s header hash.
0x11	`version` message	0	N/A	Connecting node is using a protocol version that the rejecting node considers obsolete and unsupported.
0x12	`tx` message	32	char[32]	Duplicate input spend (double spend): the rejected transaction spends the same input as a previously-received transaction. Extra data is the rejected transaction’s TXID.
0x12	`version` message	0	N/A	More than one `version` message received in this connection.
0x40	`tx` message	32	char[32]	The transaction will not be mined or relayed because the rejecting node considers it non-standard—a transaction type or version unknown by the server. Extra data is the rejected transaction’s TXID.
0x41	`tx` message	32	char[32]	One or more output amounts are below the dust threshold. Extra data is the rejected transaction’s TXID.
0x42	`tx` message		char[32]	The transaction did not have a large enough fee or priority to be relayed or mined. Extra data is the rejected transaction’s TXID.
0x43	`block` message	32	char[32]	The block belongs to a block chain which is not the same block chain as provided by a compiled-in checkpoint. Extra data is the rejected block’s header hash.

The annotated hexdump below shows a `reject` message. (The message header has been omitted.)

```
02 ................................. Number of bytes in message: 2
7478 ............................... Type of message rejected: tx
12 ................................. Reject code: 0x12 (duplicate)
15 ................................. Number of bytes in reason: 21
6261642d74786e732d696e707574732d
7370656e74 ......................... Reason: bad-txns-inputs-spent
394715fcab51093be7bfca5a31005972
947baf86a31017939575fb2354222821 ... TXID
```

VerAck

The `verack` message acknowledges a previously-received `version` message, informing the connecting node that it can begin to send other messages. The `verack` message has no payload; for an example of a message with no payload, see the message headers section.

Version

The `version` message provides information about the transmitting node to the receiving node at the beginning of a connection. Until both peers have exchanged `version` messages, no other messages will be accepted.

If a `version` message is accepted, the receiving node should send a `verack` message—but no node should send a `verack` message before initializing its half of the connection by first sending a `version` message.

Bytes	Name	Data Type	Description
4	version	int32_t	The highest protocol version understood by the transmitting node. See the protocol version section.
8	services	uint64_t	The services supported by the transmitting node encoded as a bitfield. See the list of service codes below.
8	timestamp	int64_t	The current Unix epoch time according to the transmitting node's clock. Because nodes will reject blocks with timestamps more than two hours in the future, this field can help other nodes to determine that their clock is wrong.
8	addr_recv services	uint64_t	*Added in protocol version 106.* The services supported by the receiving node as perceived by the transmitting node. Same format as the 'services' field above. Bitcoin Core will attempt to provide accurate information. BitcoinJ will, by default, always send 0.
16	addr_recv IP address	char	*Added in protocol version 106.* The IPv6 address of the receiving node as perceived by the transmitting node in **big endian byte order**. IPv4 addresses can be provided as IPv4-mapped IPv6 addresses. Bitcoin Core will attempt to provide accurate information. BitcoinJ will, by default, always return ::ffff:127.0.0.1
2	addr_recv port	uint16_t	*Added in protocol version 106.* The port number of the receiving node as perceived by the transmitting node in **big endian byte order**.
8	addr_trans services	uint64_t	The services supported by the transmitting node. Should be identical to the 'services' field above.
16	addr_trans IP address	char	The IPv6 address of the transmitting node in **big endian byte order**. IPv4 addresses can be provided as IPv4-mapped IPv6 addresses. Set to ::ffff:127.0.0.1 if unknown.
2	addr_trans port	uint16_t	The port number of the transmitting node in **big endian byte order**.
8	nonce	uint64_t	A random nonce which can help a node detect a connection to itself. If the nonce is 0, the nonce field is ignored. If the nonce is anything else, a node should terminate the connection on receipt of a `version` message with a nonce it previously sent.
Varies	user_agent bytes	compactSize uint	Number of bytes in following user_agent field. If 0x00, no user agent field is sent.
Varies	user_agent	string	*Renamed in protocol version 60000.* User agent as defined by BIP14. Previously called subVer.
4	start_height	int32_t	The height of the transmitting node's best block chain or, in the case of an SPV client, best block header chain.

1	relay	bool	*Added in protocol version 70001 as described by BIP37.* Transaction relay flag. If 0x00, no `inv` messages or `tx` messages announcing new transactions should be sent to this client until it sends a `filterload` message or `filterclear` message. If 0x01, this node wants `inv` messages and `tx` messages announcing new transactions.

The following service identifiers have been assigned.

Value	Name	Description
0x00	*Unnamed*	This node is not a full node. It may not be able to provide any data except for the transactions it originates.
0x01	NODE_NETWORK	This is a full node and can be asked for full blocks. It should implement all protocol features available in its self-reported protocol version.

The following annotated hexdump shows a `version` message. (The message header has been omitted and the actual IP addresses have been replaced with RFC5737 reserved IP addresses.)

```
72110100 ........................... Protocol version: 70002
0100000000000000 ................... Services: NODE_NETWORK
bc8f5e5400000000 ................... Epoch time: 1415483324

0100000000000000 ................... Receiving node's services
00000000000000000000ffffc61b6409 ... Receiving node's IPv6 address
208d ............................... Receiving node's port number

0100000000000000 ................... Transmitting node's services
00000000000000000000ffffcb0071c0 ... Transmitting node's IPv6 address
208d ............................... Transmitting node's port number

128035cbc97953f8 ................... Nonce

0f ................................. Bytes in user agent string: 15
2f5361746f7368693a302e392e332f ..... User agent: /Satoshi:0.9.2.1/

cf050500 ........................... Start height: 329167
01 ................................. Relay flag: true
```

Bitcoin Core APIs

Hash Byte Order

Bitcoin Core RPCs accept and return hashes in the reverse of their normal byte order. For example, the Unix `sha256sum` command would display the SHA256(SHA256()) hash of mainnet block 300,000's header as the following string:

```
5472ac8b1187bfcf91d6d218bbda1eb2405d7c55f1f8cc820000000000000000
```

The string above is also how the hash appears in the previous-header-hash part of block 300,001's header:

```
020000005472ac8b1187bfcf91d6d218bbda1eb2405d7c55f1f8cc82000\
0000000000000ab0aaa377ca3f49b1545e2ae6b0667a08f42e72d8c24ae\
237140e28f14f3bb7c6bcc6d536c890019edd83ccf
```

However Bitcoin RPCs use the reverse byte order for hashes, so if you want to get information about block 300,000 using the `getblock` RPC, you need to reverse the byte order:

```
> bitcoin-cli getblock \
  000000000000000082ccf8f1557c5d40b21edabb18d2d691cfbf87118bac7254
```

(Note: hex representation uses two characters to display each byte of data, which is why the reversed string looks somewhat mangled.)

The rational for the reversal is unknown, but it likely stems from Bitcoin's use of hash digests (which are byte arrays in C++) as integers for the purpose of determining whether the hash is below the network target. Whatever the reason for reversing header hashes, the reversal also extends to other hashes used in RPCs, such as TXIDs and merkle roots.

Off-site documentation such as the Bitcoin Wiki tends to use the terms big endian and little endian as shown in the table below, but they aren't always consistent. Worse, these two different ways of representing a hash digest can confuse anyone who looks at the Bitcoin Core source code and finds a so-called "big endian" value being stored in a little-endian data type.

As header hashes and TXIDs are widely used as global identifiers in other Bitcoin software, this reversal of hashes has become the standard way to refer to certain objects. The table below should make clear where each byte order is used.

Data	Internal Byte Order ("Big Endian")	RPC Byte Order ("Little Endian")
Example: SHA256(SHA256(0x00))	Hash: 1406…539a	Hash: 9a53…0614
Header Hashes: SHA256(SHA256(block header))	Used when constructing block headers	Used by RPCs such as `getblock`; widely used in block explorers
Merkle Roots: SHA256(SHA256(TXIDs and merkle rows))	Used when constructing block headers	Returned by RPCs such as `getblock`
TXIDs: SHA256(SHA256(transaction))	Used in transaction inputs	Used by RPCs such as `gettransaction` and transaction data parts of `getblock`; widely used in wallet programs
P2PKH Hashes: RIPEMD160(SHA256(pubkey))	Used in both addresses and pubkey scripts	**N/A:** RPCs use addresses which use internal byte order
P2SH Hashes: RIPEMD160(SHA256(redeem script))	Used in both addresses and pubkey scripts	**N/A:** RPCs use addresses which use internal byte order

Note: RPCs which return raw results, such as `getrawtransaction` or the raw mode of `getblock`, always display hashes as they appear in blocks (internal byte order).

The code below may help you check byte order by generating hashes from raw hex.

```
#!/usr/bin/env python

from sys import byteorder
from hashlib import sha256

## You can put in $data an 80-byte block header to get its header hash,
## or a raw transaction to get its txid
data = "00".decode("hex")
hash = sha256(sha256(data).digest()).digest()

print "Warning: this code only tested on a little-endian x86_64 arch"
print
print "System byte order:", byteorder
print "Internal-Byte-Order Hash: ", hash.encode('hex_codec')
print "RPC-Byte-Order Hash:      ", hash[::-1].encode('hex_codec')
```

Remote Procedure Calls (RPCs)

Bitcoin Core provides a large number of Remote Procedure Calls (RPCs) using a HTTP JSON-RPC version 1.0 interface. Any program can access the RPCs using JSON-RPC, but Bitcoin Core also provides the `bitcoin-cli` command to wrap the JSON-RPC access for Command Line Interface (CLI) users. Most of the RPC examples in this documentation use `bitcoin-cli` for simplicity, so this subsection describes raw JSON-RPC interface and how the command-line interface translates it.

In order to start `bitcoind`, you will need to set a password for JSON-RPC in the `bitcoin.conf` file. See the Examples Page for details. JSON-RPC starts on port 8332 for mainnet and 18332 for testnet and regtest. By default, `bitcoind` doesn't use a JSON-RPC user, but you can set one (see `bitcoind --help`).

The interface is not intended for public access and is only accessible from localhost by default.

RPCs are made using the standard JSON-RPC 1.0 syntax, which sends several standard arguments:

Name	Type	Presence	Description
RPC	object	Required (exactly 1)	An object containing the standard RPC arguments
→ `jsonrpc`	number (real)	Optional (0 or 1)	The version of JSON-RPC used. Bitcoin Core currently ignores this, as it only supports version 1.0. Default is `1.0`
→ `id`	string	Required (exactly 1)	An arbitrary string that will be returned when the response is sent. May be set to an empty string (“”)
→ `method`	string	Required (exactly 1)	The RPC, such as `getbestblockhash`. See the RPC section for a list of available commands
→ `params`	array	Required (exactly 1)	An array containing parameters for the RPC. May be an empty array if allowed by the particular RPC
→ → Parameter	*any*	Optional (0 or more)	A parameter. May be any JSON type allowed by the particular RPC

In table above and in other tables describing JSON-RPC input and output, we use the following formatting

- “→” to indicate an argument that is the child of a JSON array or JSON object. For example, “→ → Parameter” above means Parameter is the child of the `params` array which itself is a child of the RPC array.
- “Plain Text” names (like “RPC” above) are unnamed in the actual JSON-RPC
- `literal` names (like `id` above) are the strings that appear in the actual JSON-RPC
- Type (specifics) are the general JSON-RPC type and the specific Bitcoin Core type
- Required/Optional describe whether a field must be returned within its containing array or object. (So an optional object may still have required children.)

For example, here is the JSON-RPC requesting the hash of the latest block on the local best block chain:

```
{
    "jsonrpc": "1.0",
    "id": "bitcoin.org developer documentation",
    "method": "getbestblockhash",
    "params": []
}
```

We can send that to a local Bitcoin Core running on testnet using cURL with the following command:

```
curl --user ':your_password' --data-binary '''
  {
      "jsonrpc": "1.0",
      "id":"bitcoin.org developer documentation",
      "method": "getbestblockhash",
      "params": []
  }''' \
  -H 'content-type: text/plain;' http://127.0.0.1:18332/
```

The output will be sent using the standard JSON-RPC 1.0 format. For example (whitespace added):

```
{
    "result": "00000000bd68bfdf381efd5fff17c723d2bb645bcbb215a6e333d4204888e951",
    "error": null,
    "id": "bitcoin.org developer documentation"
}
```

The standard JSON-RPC 1.0 result format is described below:

Name	Type	Presence	Description
Result	object	Required (exactly 1)	An object describing the results
→ `result`	*any*	Required (exactly 1)	The results as any JSON data type. If an error occured, set to `null`
→ `error`	null/object	Required (exactly 1)	If no error occurred, set to `null`. If an error occured, an object describing the error
→ → `code`	number (int)	Required (exactly 1)	The error code as set by the returning function and defined in Bitcoin Core's rpcprotocol.h
→ → `message`	string	Required (exactly 1)	An attempt to describe the problem in human-readable text. May be an empty string (""). Bitcoin Core often returns help text with embedded newline strings ("\n"); `bitcoin-cli` can expand these to actual newlines
→ `id`	string	Required (exactly 1)	The arbitrary string passed in when the RPC was called

For an example of the error output, here's the result after passing an invalid address to the `sendtoaddress` RPC (whitespace added):

```
{
    "result": null,
    "error": {
        "code": -5,
        "message": "Invalid Bitcoin address"
    },
    "id": "bitcoin.org developer documentation"
}
```

The `bitcoin-cli` command can save command line users a lot of typing compared to using cURL or another HTTP-sending command. For example, to get the block hash we got before, we would use the following command:

```
bitcoin-cli getbestblockhash
```

For non-error output, `bitcoin-cli` will only display the value of the `result` field, and if it's a string, `bitcoin-cli` will remove its JSON quotation marks. For example, the result for the command above:

```
00000000bd68bfdf381efd5fff17c723d2bb645bcbb215a6e333d4204888e951
```

For errors, `bitcoin-cli` will display only the `error` object. For example, the result of the invalid address command above as formatted by `bitcoin-cli`:

```
error: {"code":-5,"message":"Invalid Bitcoin address"}
```

Because `bitcoin-cli` abstracts away the parts of JSON-RPC we would need to repeatedly describe in each RPC description below, we describe the Bitcoin Core RPCs using `bitcoin-cli`. However, using an actual programming interface to the full JSON-RPC will serve you much better for automated tasks.

⚠ **Warning:** if you write programs using the JSON-RPC interface, you must ensure they handle high-precision real numbers correctly. See the Proper Money Handling Bitcoin Wiki article for details and example code.

Quick Reference

Block Chain RPCs

- GetBestBlockHash: returns the header hash of the most recent block on the best block chain. *Newin 0.9.0*
- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlockChainInfo: provides information about the current state of the block chain. *Newin 0.9.2*, **Updated in 0.10.0**
- GetBlockCount: returns the number of blocks in the local best block chain.
- GetBlockHash: returns the header hash of a block at the given height in the local best block chain.
- GetChainTips: returns information about the highest-height block (tip) of each local block chain. **New in 0.10.0**
- GetDifficulty: returns the proof-of-work difficulty as a multiple of the minimum difficulty.
- GetMemPoolInfo: returns information about the node's current transaction memory pool. **New in 0.10.0**
- GetRawMemPool: returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object.
- GetTxOut: returns details about a transaction output. Only unspent transaction outputs (UTXOs) are guaranteed to be available.
- GetTxOutSetInfo: returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.
- VerifyChain: verifies each entry in the local block chain database.

Control RPCs

- GetInfo: prints various information about the node and the network. **Updated in 0.10.0**, **Deprecated**
- Help: lists all available public RPC commands, or gets help for the specified RPC. Commands which are unavailable will not be listed, such as wallet RPCs if wallet support is disabled.
- Stop: safely shuts down the Bitcoin Core server.

Generating RPCs

- GetGenerate: returns true if the node is set to generate blocks using its CPU.

- SetGenerate: enables or disables hashing to attempt to find the next block.

Mining RPCs

- GetBlockTemplate: gets a block template or proposal for use with mining software.
- GetMiningInfo: returns various mining-related information. **Updated in master**
- GetNetworkHashPS: returns the estimated current or historical network hashes per second based on the last *n* blocks.
- PrioritiseTransaction: adds virtual priority or fee to a transaction, allowing it to be accepted into blocks mined by this node (or miners which use this node) with a lower priority or fee. (It can also remove virtual priority or fee, requiring the transaction have a higher priority or fee to be accepted into a locally-mined block.) **New in 0.10.0**
- SubmitBlock: accepts a block, verifies it is a valid addition to the block chain, and broadcasts it to the network. Extra parameters are ignored by Bitcoin Core but may be used by mining pools or other programs.

Network RPCs

- AddNode: attempts to add or remove a node from the addnode list, or to try a connection to a node once.
- GetAddedNodeInfo: returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the `addnode` RPC will have their information displayed.
- GetConnectionCount: returns the number of connections to other nodes.
- GetNetTotals: returns information about network traffic, including bytes in, bytes out, and the current time.
- GetNetworkInfo: returns information about the node’s connection to the network. *New in 0.9.2*, **Updated in 0.10.0**
- GetPeerInfo: returns data about each connected network node. **Updated in 0.10.0**
- Ping: sends a P2P ping message to all connected nodes to measure ping time. Results are provided by the `getpeerinfo` RPC pingtime and pingwait fields as decimal seconds. The P2P `ping` message is handled in a queue with all other commands, so it measures processing backlog, not just network ping.

Raw Transaction RPCs

- CreateRawTransaction: creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
- DecodeRawTransaction: decodes a serialized transaction hex string into a JSON object describing the transaction.
- DecodeScript: decodes a hex-encoded P2SH redeem script.
- GetRawTransaction: gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings.
- SendRawTransaction: validates a transaction and broadcasts it to the peer-to-peer network.
- SignRawTransaction: signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.

Utility RPCs

- CreateMultiSig: creates a P2SH multi-signature address.
- EstimateFee: estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks. **New in 0.10.0**
- EstimatePriority: estimates the priority that a transaction needs in order to be included within a certain number of blocks as a free high-priority transaction. **New in 0.10.0**
- ValidateAddress: returns information about the given Bitcoin address.
- VerifyMessage: verifies a signed message.

Wallet RPCs

Note: the wallet RPCs are only available if Bitcoin Core was built with wallet support, which is the default.

- AddMultiSigAddress: adds a P2SH multisig address to the wallet.
- BackupWallet: safely copies `wallet.dat` to the specified file, which can be a directory or a path with filename.
- DumpPrivKey: returns the wallet-import-format (WIP) private key corresponding to an address. (But does not remove it from the wallet.)
- DumpWallet: creates or overwrites a file with all wallet keys in a human-readable format.
- EncryptWallet: encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.
- GetAccountAddress: returns the current Bitcoin address for receiving payments to this account. If the account doesn’t exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address.
- GetAccount: returns the name of the account associated with the given address.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.
- GetBalance: gets the balance in decimal bitcoins across all accounts or for a particular account.
- GetNewAddress: returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.
- GetRawChangeAddress: returns a new Bitcoin address for receiving change. This is for use with raw transactions, not normal use.
- GetReceivedByAccount: returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetReceivedByAddress: returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetTransaction: gets detailed information about an in-wallet transaction. **Updated in 0.10.0**
- GetUnconfirmedBalance: returns the wallet’s total unconfirmed balance.
- GetWalletInfo: provides information about the wallet. *New in 0.9.2*
- ImportAddress: adds an address or pubkey script to the wallet without the associated private key, allowing you to watch for transactions affecting that address or pubkey script without being able to spend any of its outputs. **New in 0.10.0**
- ImportPrivKey: adds a private key to your wallet. The key should be formatted in the wallet import format created by the `dumpprivkey` RPC.
- ImportWallet: imports private keys from a file in wallet dump file format (see the `dumpwallet` RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.
- KeyPoolRefill: fills the cache of unused pre-generated keys (the keypool).
- ListAccounts: lists accounts and their balances. **Updated in 0.10.0**
- ListAddressGroupings: lists groups of addresses that may have had their common ownership made public by common use as inputs in the same transaction or from being used as change from a previous transaction.
- ListLockUnspent: returns a list of temporarily unspendable (locked) outputs.
- ListReceivedByAccount: lists the total number of bitcoins received by each account. **Updated in 0.10.0**
- ListReceivedByAddress: lists the total number of bitcoins received by each address. **Updated in 0.10.0**
- ListSinceBlock: gets all transactions affecting the wallet which have occurred since a particular block, plus the header hash of a block at a particular depth. **Updated in 0.10.0**
- ListTransactions: returns the most recent transactions that affect the wallet. **Updated in 0.10.0**
- ListUnspent: returns an array of unspent transaction outputs belonging to this wallet. **Updated in 0.10.0**
- LockUnspent: temporarily locks or unlocks specified transaction outputs. A locked transaction output will not be chosen by automatic

coin selection when spending bitcoins. Locks are stored in memory only, so nodes start with zero locked outputs and the locked output list is always cleared when a node stops or fails.

- Move: moves a specified amount from one account in your wallet to another using an off-block-chain transaction.
- SendFrom: spends an amount from a local account to a bitcoin address.
- SendMany: creates and broadcasts a transaction which sends outputs to multiple addresses.
- SendToAddress: spends an amount to a given address.
- SetAccount: puts the specified address in the given account.
- SetTxFee: sets the transaction fee per kilobyte paid by transactions created by this wallet.
- SignMessage: signs a message with the private key of an address.
- WalletLock: removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call `walletpassphrase` again before being able to call any methods which require the wallet to be unlocked.
- WalletPassphrase: stores the wallet decryption key in memory for the indicated number of seconds. Issuing the `walletpassphrase` command while the wallet is already unlocked will set a new unlock time that overrides the old one.
- WalletPassphraseChange: changes the wallet passphrase from 'old passphrase' to 'new passphrase'.

Removed RPCs

- GetHashesPerSec: was removed in Bitcoin Core master (unreleased). It returned a recent hashes per second performance measurement when the node was generating blocks.
- GetWork: was removed in Bitcoin Core 0.10.0.

RPCs

Warning: the block chain and memory pool can include arbitrary data which several of the commands below will return in hex format. If you convert this data to another format in an executable context, it could be used in an exploit. For example, displaying a pubkey script as ASCII text in a webpage could add arbitrary Javascript to that page and create a cross-site scripting (XSS) exploit. To avoid problems, please treat block chain and memory pool data as an arbitrary input from an untrusted source.

AddMultiSigAddress

Requires wallet support.

The `addmultisigaddress` RPC adds a P2SH multisig address to the wallet.

Parameter #1—the number of signatures required

Name	Type	Presence	Description
Required	number (int)	Required (exactly 1)	The minimum (*m*) number of signatures required to spend this m-of-n multisig script

Parameter #2—the full public keys, or addresses for known public keys

Name	Type	Presence	Description
Keys Or Addresses	array	Required (exactly 1)	An array of strings with each string being a public key or address
→ Key Or Address	string	Required (1 or more)	A public key against which signatures will be checked. Alternatively, this may be a P2PKH address belonging to the wallet—the corresponding public key will be substituted. There must be at least as many keys as specified by the Required parameter, and there may be more keys

Parameter #3—the account name

Name	Type	Presence	Description
Account	string	Optional (0 or 1)	The account name in which the address should be stored. Default is the default account, "" (an empty string)

Result—a P2SH address printed and stored in the wallet

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	The P2SH multisig address. The address will also be added to the wallet, and outputs paying that address will be tracked by the wallet

Example from Bitcoin Core 0.10.0

Adding a 2-of-3 P2SH multisig address to the "test account" by mixing two P2PKH addresses and one full public key:

```
bitcoin-cli -testnet addmultisigaddress \
  2 \
  '''
    [
      "mjbLRSidW1MY8oubvs4SMEnHNFXxCcoehQ",
      "02ecd2d250a76d204011de6bc365a56033b9b3a149f679bc17205555d3c2b2854f",
      "mt17cV37fBqZsnMmrHnGCm9pM28R1kQdMG"
    ]
  ''' \
  'test account'
```

Result:

```
2MyVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq
```

(New P2SH multisig address also stored in wallet.)

See also

- CreateMultiSig: creates a P2SH multi-signature address.
- DecodeScript: decodes a hex-encoded P2SH redeem script.
- Pay-To-Script-Hash (P2SH)

AddNode

The `addnode` RPC attempts to add or remove a node from the addnode list, or to try a connection to a node once.

Parameter #1—hostname/IP address and port of node to add or remove

Name	Type	Presence	Description
Node	string	Required (exactly 1)	The node to add as a string in the form of `<IP address>:<port>`. The IP address may be a hostname resolvable through DNS, an IPv4 address, an IPv4-as-IPv6 address, or an IPv6 address

Parameter #2—whether to add or remove the node, or to try only once to connect

Name	Type	Presence	Description

Command	string	Required (exactly 1)	What to do with the IP address above. Options are: • `add` to add a node to the addnode list. This will not connect immediately if the outgoing connection slots are full • `remove` to remove a node from the list. If currently connected, this will disconnect immediately • `onetry` to immediately attempt connection to the node even if the outgoing connection slots are full; this will only attempt the connection once

Result—`null` plus error on failed remove

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always JSON `null` whether the node was added, removed, tried-and-connected, or tried-and-not-connected. The JSON-RPC error field will be set only if you try removing a node that is not on the addnodes list

Example from Bitcoin Core 0.10.0

Try connecting to the following node.

```
bitcoin-cli -testnet addnode 192.0.2.113:18333 onetry
```

Result (no output from `bitcoin-cli` because result is set to `null`).

See also

- GetAddedNodeInfo: returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the `addnode` RPC will have their information displayed.

BackupWallet

Requires wallet support.

The `backupwallet` RPC safely copies `wallet.dat` to the specified file, which can be a directory or a path with filename.

Parameter #1—destination directory or filename

Name	Type	Presence	Description
Destination	string	Required (exactly 1)	A filename or directory name. If a filename, it will be created or overwritten. If a directory name, the file `wallet.dat` will be created or overwritten within that directory

Result—`null` or error

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always `null` whether success or failure. The JSON-RPC error and message fields will be set if a failure occurred

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet backupwallet /tmp/backup.dat
```

See also

- DumpWallet: creates or overwrites a file with all wallet keys in a human-readable format.

- ImportWallet: imports private keys from a file in wallet dump file format (see the `dumpwallet` RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.

CreateMultiSig

The `createmultisig` RPC creates a P2SH multi-signature address.

Parameter #1—the number of signatures required

Name	Type	Presence	Description
Required	number (int)	Required (exactly 1)	The minimum (*m*) number of signatures required to spend this m-of-n multisig script

Parameter #2—the full public keys, or addresses for known public keys

Name	Type	Presence	Description
Keys Or Addresses	array	Required (exactly 1)	An array of strings with each string being a public key or address
→ Key Or Address	string	Required (1 or more)	A public key against which signatures will be checked. If wallet support is enabled, this may be a P2PKH address belonging to the wallet—the corresponding public key will be substituted. There must be at least as many keys as specified by the Required parameter, and there may be more keys

Result—P2SH address and hex-encoded redeem script

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the multisig address
→ `address`	string (base58)	Required (exactly 1)	The P2SH address for this multisig redeem script
→ `redeemScript`	string (hex)	Required (exactly 1)	The multisig redeem script encoded as hex

Example from Bitcoin Core 0.10.0

Creating a 2-of-3 P2SH multisig address by mixing two P2PKH addresses and one full public key:

```
bitcoin-cli -testnet createmultisig 2 '''
  [
    "mjbLRSidW1MY8oubvs4SMEnHNFXxCcoehQ",
    "02ecd2d250a76d204011de6bc365a56033b9b3a149f679bc17205555d3c2b2854f",
    "mt17cV37fBqZsnMmrHnGCm9pM28R1kQdMG"
  ]
'''
```

Result:

```
{
  "address" : "2MyVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq",
  "redeemScript" : "522103ede722780d27b05f0b1169efc90fa15a601a32fc6c3295114500c586831b6aaf2102ecd2d250a76d204011de6bc365a56033b9b3
}
```

See also

- AddMultiSigAddress: adds a P2SH multisig address to the wallet.
- DecodeScript: decodes a hex-encoded P2SH redeem script.
- Pay-To-Script-Hash (P2SH)

CreateRawTransaction

The `createrawtransaction` RPC creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.

Parameter #1—references to previous outputs

Name	Type	Presence	Description
Outpoints	array	Required (exactly 1)	An array of objects, each one being an unspent outpoint
→ Outpoint	object	Required (1 or more)	An object describing a particular unspent outpoint
→ → `txid`	string (hex)	Required (exactly 1)	The TXID of the outpoint encoded as hex in RPC byte order
→ → `vout`	number (int)	Required (exactly 1)	The output index number (vout) of the outpoint; the first output in a transaction is index `0`

Parameter #2—P2PKH or P2SH addresses and amounts

Name	Type	Presence	Description
Outputs	object	Required (exactly 1)	The addresses and amounts to pay
→ Address/Amount	string : number (bitcoins)	Required (1 or more)	A key/value pair with the address to pay as a string (key) and the amount to pay that address (value) in bitcoins

Result—the unsigned raw transaction in hex

Name	Type	Presence	Description
`result`	string	Required (Exactly 1)	The resulting unsigned raw transaction in serialized transaction format encoded as hex. If the transaction couldn't be generated, this will be set to JSON `null` and the JSON-RPC error field may contain an error message

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet createrawtransaction '''
  [
    {
      "txid": "1eb590cd06127f78bf38ab4140c4cdce56ad9eb8886999eb898ddf4d3b28a91d",
      "vout" : 0
    }
  ]''' '{ "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe": 0.13 }'
```

Result (wrapped):

```
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f12\
06cd90b51e0000000000ffffffff01405dc600000000001976a9140dfc8bafc8\
419853b34d5e072ad37d1a5159f58488ac00000000
```

See also

- DecodeRawTransaction: decodes a serialized transaction hex string into a JSON object describing the transaction.
- SignRawTransaction: signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.
- SendRawTransaction: validates a transaction and broadcasts it to the peer-to-peer network.
- Serialized Transaction Format

DecodeRawTransaction

The `decoderawtransaction` RPC decodes a serialized transaction hex string into a JSON object describing the transaction.

Parameter #1—serialized transaction in hex

Name	Type	Presence	Description
Serialized Transaction	string (hex)	Required (exactly 1)	The transaction to decode in serialized transaction format

Result—the decoded transaction

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the decoded transaction, or JSON `null` if the transaction could not be decoded
→ `txid`	string (hex)	Required (exactly 1)	The transaction's TXID encoded as hex in RPC byte order
→ `version`	number (int)	Required (exactly 1)	The transaction format version number
→ `locktime`	number (int)	Required (exactly 1)	The transaction's locktime: either a Unix epoch date or block height; see the Locktime parsing rules
→ `vin`	array	Required (exactly 1)	An array of objects with each object being an input vector (vin) for this transaction. Input objects will have the same order within the array as they have in the transaction, so the first input listed will be input 0
→ → Input	object	Required (1 or more)	An object describing one of this transaction's inputs. May be a regular input or a coinbase
→ → → `txid`	string	Optional (0 or 1)	The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction
→ → →	number	Optional	The output index number (vout) of the outpoint being spent. The first output in a

`vout`	(int)	(0 or 1)	transaction has an index of `0`. Not present if this is a coinbase transaction
→ → → `scriptSig`	object	Optional (0 or 1)	An object describing the signature script of this input. Not present if this is a coinbase transaction
→ → → → `asm`	string	Required (exactly 1)	The signature script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The signature script encoded as hex
→ → → `coinbase`	string (hex)	Optional (0 or 1)	The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction
→ → → `sequence`	number (int)	Required (exactly 1)	The input sequence number
→ `vout`	array	Required (exactly 1)	An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0
→ → Output	object	Required (1 or more)	An object describing one of this transaction's outputs
→ → → `value`	number (bitcoins)	Required (exactly 1)	The number of bitcoins paid to this output. May be `0`
→ → → `n`	number (int)	Required (exactly 1)	The output index number of this output within this transaction
→ → → `scriptPubKey`	object	Required (exactly 1)	An object describing the pubkey script
→ → → → `asm`	string	Required (exactly 1)	The pubkey script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The pubkey script encoded as hex
→ → → → `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for `nulldata` or `nonstandard` script types (see the `type` key below)
→ → → → `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script • `pubkeyhash` for a P2PKH script • `scripthash` for a P2SH script • `multisig` for a bare multisig script • `nulldata` for nulldata scripts • `nonstandard` for unknown scripts
→ → → → `addresses`	string : array	Optional (0 or 1)	The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for `nulldata` or `nonstandard` script types
→ → → → →		Required	

Address	string	(1 or more)	A P2PKH or P2SH address

Example from Bitcoin Core 0.10.0

Decode a signed one-input, three-output transaction:

```
bitcoin-cli -testnet decoderawtransaction 0100000001268a9ad7bfb2\
1d3c086f0ff28f73a064964aa069ebb69a9e437da85c7e55c7d7000000006b48\
3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560f\
eb95de63b902206f23a0919471eaa1e45a0982ed288d374397d30dff541b2dd4\
5a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b\
3118f3db16cbbcf8f326ffffffff0350ac6002000000001976a91456847befbd\
2360df0e35b4e3b77bae48585ae06888ac80969800000000001976a9142b1495\
0b8d31620c6cc923c5408a701b1ec0a02088ac002d3101000000001976a9140d\
fc8bafc8419853b34d5e072ad37d1a5159f58488ac00000000
```

Result:

```
{
    "txid" : "ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "d7c7557e5ca87d439e9ab6eb69a04a9664a0738ff20f6f083c1db2bfd79a8a26",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed2
                "hex" : "483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982e
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.39890000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "moQR7i8XM4rSGoNwEsw3h4YEuduuP6mxw7"
                ]
            }
        },
        {
            "value" : 0.10000000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN"
                ]
            }
        },
        {
            "value" : 0.20000000,
            "n" : 2,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 0dfc8bafc8419853b34d5e072ad37d1a5159f584 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe"
                ]
            }
        }
    ]
}
```

See also

- CreateRawTransaction: creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
- SignRawTransaction: signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.
- SendRawTransaction: validates a transaction and broadcasts it to the peer-to-peer network.

DecodeScript

The `decodescript` RPC decodes a hex-encoded P2SH redeem script.

Parameter #1—a hex-encoded redeem script

Name	Type	Presence	Description
Redeem Script	string (hex)	Required (exactly 1)	The redeem script to decode as a hex-encoded serialized script

Result—the decoded script

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the decoded script, or JSON `null` if the script could not be decoded
→ `asm`	string	Required (exactly 1)	The redeem script in decoded form with non-data-pushing op codes listed. May be empty
→ `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script inside P2SH • `pubkeyhash` for a P2PKH script inside P2SH • `multisig` for a multisig script inside P2SH • `nonstandard` for unknown scripts
→ `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK or P2PKH within P2SH. It may be greater than 1 for P2SH multisig. This value will not be returned for `nonstandard` script types (see the `type` key above)
→ `addresses`	array	Optional (0 or 1)	A P2PKH addresses used in this script, or the computed P2PKH addresses of any pubkeys in this script. This array will not be returned for `nonstandard` script types
→ → Address	string	Required (1 or more)	A P2PKH address
→ `p2sh`	string (hex)	Required (exactly 1)	The P2SH address of this redeem script

Example from Bitcoin Core 0.10.0

A 2-of-3 P2SH multisig pubkey script:

```
bitcoin-cli -testnet decodescript 522103ede722780d27b05f0b1169ef\
c90fa15a601a32fc6c3295114500c586831b6aaf2102ecd2d250a76d204011de\
6bc365a56033b9b3a149f679bc17205555d3c2b2854f21022d609d2f0d359e5b\
c0e5d0ea20ff9f5d3396cb5b1906aa9c56a0e7b5edc0c5d553ae
```

Result:

```
{
    "asm" : "2 03ede722780d27b05f0b1169efc90fa15a601a32fc6c3295114500c586831b6aaf 02ecd2d250a76d204011de6bc365a56033b9b3a149f679bc
    "reqSigs" : 2,
    "type" : "multisig",
    "addresses" : [
        "mjbLRSidW1MY8oubvs4SMEnHNFXxCcoehQ",
        "mo1vzGwCzWqteip29vGWWW6MsEBREuzW94",
        "mt17cV37fBqZsnMmrHnGCm9pM28R1kQdMG"
    ],
    "p2sh" : "2MyVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq"
}
```

See also

- CreateMultiSig: creates a P2SH multi-signature address.
- Pay-To-Script-Hash (P2SH)

DumpPrivKey

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `dumpprivkey` RPC returns the wallet-import-format (WIP) private key corresponding to an address. (But does not remove it from the wallet.)

Parameter #1—the address corresponding to the private key to get

Name	Type	Presence	Description
P2PKH Address	string (base58)	Required (exactly 1)	The P2PKH address corresponding to the private key you want returned. Must be the address corresponding to a private key in this wallet

Result—the private key

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	The private key encoded as base58check using wallet import format

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet dumpprivkey moQR7i8XM4rSGoNwEsw3h4YEuduuP6mxw7
```

Result:

```
cTVNtBK7mBi2yc9syEnwbiUpnpGJKohDWzXMeF4tGKAQ7wvomr95
```

See also

- ImportPrivKey: adds a private key to your wallet. The key should be formatted in the wallet import format created by the `dumpprivkey` RPC.
- DumpWallet: creates or overwrites a file with all wallet keys in a human-readable format.

DumpWallet

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `dumpwallet` RPC creates or overwrites a file with all wallet keys in a human-readable format.

Parameter #1—a filename

Name	Type	Presence	Description
Filename	string	Required (exactly 1)	The file in which the wallet dump will be placed. May be prefaced by an absolute file path. An existing file with that name will be overwritten

Result—`null` or error

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always `null` whether success or failure. The JSON-RPC error and message fields will be set if a failure occurred

Example from Bitcoin Core 0.10.0

Create a wallet dump and then print its first 10 lines.

```
bitcoin-cli -testnet dumpwallet /tmp/dump.txt
head /tmp/dump.txt
```

Results (only showing the first 10 lines):

```
# Wallet dump created by Bitcoin v0.9.1.0-g026a939-beta (Tue, 8 Apr 2014 12:04:06 +0200)
# * Created on 2014-04-29T20:46:09Z
# * Best block at time of backup was 227221 (0000000026ede4c10594af8087748507fb06dcd30b8f4f48b9cc463cabc9d767),
#   mined on 2014-04-29T21:15:07Z

cTtefiUaLfXuyBXJBBywSdg8soTEkBNh9yTi1KgoHxUYxt1xZ2aA 2014-02-05T15:44:03Z label=test1 # addr=mnUbTmdAFD5EAg3348Ejmonub7JcWtrMck
cQNY9v93Gyt8KmwygFR59bDhVs3aRDkuT8pKaCBpop82TZ8ND1tH 2014-02-05T16:58:41Z reserve=1 # addr=mp4MmhTp3au21HPRz5waf6YohGumuNnsqT
cNTEPzZH9mjquFFADXe5S3BweNiHLUKD6PvEKEsHApqjX4ZddeU6 2014-02-05T16:58:41Z reserve=1 # addr=n3pdvsxveMBkktjsGJixfSbxacRUwJ9jQW
cTVNtBK7mBi2yc9syEnwbiUpnpGJKohDWzXMeF4tGKAQ7wvomr95 2014-02-05T16:58:41Z change=1 # addr=moQR7i8XM4rSGoNwEsw3h4YEuduuP6mxw7
cNCD679B4xi17jb4XeLpbRbZCbYUugptD7dCtUTfSU4KPuK2DyKT 2014-02-05T16:58:41Z reserve=1 # addr=mq8fzjxxVbAKxUGPwaSSo3C4WaUxdzfw3C
```

See also

- BackupWallet: safely copies `wallet.dat` to the specified file, which can be a directory or a path with filename.
- ImportWallet: imports private keys from a file in wallet dump file format (see the `dumpwallet` RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.

EncryptWallet

Requires wallet support.

The `encryptwallet` RPC encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.

⚠ **Warning:** if using this RPC on the command line, remember that your shell probably saves your command lines (including the value of the passphrase parameter). In addition, there is no RPC to completely disable encryption. If you want to return to an unencrypted wallet, you must create a new wallet and restore your data from a backup made with the `dumpwallet` RPC.

Parameter #1—a passphrase

Name	Type	Presence	Description
Passphrase	string	Required (exactly 1)	The passphrase to use for the encrypted wallet. Must be at least one character

Result—a notice (with program shutdown)

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	A notice that the server is stopping and that you need to make a new backup. The wallet is now encrypted

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet encryptwallet "test"
```

Result:

```
wallet encrypted; Bitcoin server stopping, restart to run with encrypted
wallet. The keypool has been flushed, you need to make a new backup.
```

See also

- WalletPassphrase: stores the wallet decryption key in memory for the indicated number of seconds. Issuing the `walletpassphrase` command while the wallet is already unlocked will set a new unlock time that overrides the old one.
- WalletLock: removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call `walletpassphrase` again before being able to call any methods which require the wallet to be unlocked.
- WalletPassphraseChange: changes the wallet passphrase from 'old passphrase' to 'new passphrase'.

EstimateFee

Added in Bitcoin Core 0.10.0.

The `estimatefee` RPC estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks.

Parameter #1—how many blocks the transaction may wait before being included

Name	Type	Presence	Description
Blocks	number (int)	Required (exactly 1)	The maximum number of blocks a transaction should have to wait before it is predicted to be included in a block

Result—the fee the transaction needs to pay per kilobyte

Name	Type	Presence	Description
`result`	number (bitcoins)	Required (exactly 1)	The estimated fee the transaction should pay in order to be included within the specified number of blocks. If the node doesn't have enough information to make an estimate, the value `-1` will be returned

Examples from Bitcoin Core 0.10.0

```
bitcoin-cli estimatefee 6
```

Result:

```
0.00026809
```

Requesting data the node can't calculate yet:

```
bitcoin-cli estimatefee 100
```

Result:

```
-1.00000000
```

See also

- EstimatePriority: estimates the priority that a transaction needs in order to be included within a certain number of blocks as a free high-priority transaction.
- SetTxFee: sets the transaction fee per kilobyte paid by transactions created by this wallet.

EstimatePriority

Added in Bitcoin Core 0.10.0.

The `estimatepriority` RPC estimates the priority that a transaction needs in order to be included within a certain number of blocks as a free high-priority transaction.

Transaction priority is relative to a transaction's byte size.

Parameter #1—how many blocks the transaction may wait before being included as a free high-priority transaction

Name	Type	Presence	Description
Blocks	number (int)	Required (exactly 1)	The maximum number of blocks a transaction should have to wait before it is predicted to be included in a block based purely on its priority

Result—the priority a transaction needs

Name	Type	Presence	Description
`result`	number (real)	Required (exactly 1)	The estimated priority the transaction should have in order to be included within the specified number of blocks. If the node doesn't have enough information to make an estimate, the value `-1` will be returned

Examples from Bitcoin Core 0.10.0

```
bitcoin-cli estimatepriority 6
```

Result:

```
718158904.10958910
```

Requesting data the node can’t calculate yet:

```
bitcoin-cli estimatepriority 100
```

Result:

```
-1.00000000
```

See also

- EstimateFee: estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks.

GetAccountAddress

Requires wallet support.

The `getaccountaddress` RPC returns the current Bitcoin address for receiving payments to this account. If the account doesn’t exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address.

Parameter #1—an account name

Name	Type	Presence	Description
Account	string	Required (exactly 1)	The name of an account. Use an empty string (“”) for the default account. If the account doesn’t exist, it will be created

Result—a bitcoin address

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	An address, belonging to the account specified, which has not yet received any payments

Example from Bitcoin Core 0.10.0

Get an address for the default account:

```
bitcoin-cli -testnet getaccountaddress ""
```

Result:

```
msQyFNYHkFUo4PG3puJBbpesvRCyRQax7r
```

See also

- GetNewAddress: returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.
- GetRawChangeAddress: returns a new Bitcoin address for receiving change. This is for use with raw transactions, not normal use.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.

GetAccount

Requires wallet support.

The `getaccount` RPC returns the name of the account associated with the given address.

Parameter #1—a Bitcoin address

Name	Type	Presence	Description
Address	string (base58)	Required (exactly 1)	A P2PKH or P2SH Bitcoin address belonging either to a specific account or the default account (“”)

Result—an account name

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	The name of an account, or an empty string (“”, the default account)

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getaccount mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN
```

Result:

```
doc test
```

See also

- GetAddressesByAccount: returns a list of every address assigned to a particular account.

GetAddedNodeInfo

The `getaddednodeinfo` RPC returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the `addnode` RPC will have their information displayed.

Parameter #1—whether to display connection information

Name	Type	Presence	Description
Details	bool	Required (exactly 1)	Set to `true` to display detailed information about each added node; set to `false` to only display the IP address or hostname and port added

Parameter #2—what node to display information about

Name	Type	Presence	Description

Node	string	Optional (0 or 1)	The node to get information about in the same `<IP address>:<port>` format as the `addnode` RPC. If this parameter is not provided, information about all added nodes will be returned

Result—a list of added nodes

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing objects describing each added node. If no added nodes are present, the array will be empty. Nodes added with `onetry` will not be returned
→ Added Node	object	Optional (0 or more)	An object containing details about a single added node
→ → `addednode`	string	Required (exactly 1)	An added node in the same `<IP address>:<port>` format as used in the `addnode` RPC. This element is present for any added node whether or not the Details parameter was set to `true`
→ → `connected`	bool	Optional (0 or 1)	If the Details parameter was set to `true`, this will be set to `true` if the node is currently connected and `false` if it is not
→ → `addresses`	array	Optional (0 or 1)	If the Details parameter was set to `true`, this will be an array of addresses belonging to the added node
→ → → Address	object	Optional (0 or more)	An object describing one of this node’s addresses
→ → → → `address`	string	Required (exactly 1)	An IP address and port number of the node. If the node was added using a DNS address, this will be the resolved IP address
→ → → → `connected`	string	Required (exactly 1)	Whether or not the local node is connected to this addnode using this IP address. Valid values are: • `false` for not connected • `inbound` if the addnode connected to us • `outbound` if we connected to the addnode

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getaddednodeinfo true
```

Result (real hostname and IP address replaced):

```
[
    {
        "addednode" : "bitcoind.example.com:18333",
        "connected" : true,
        "addresses" : [
            {
                "address" : "192.0.2.113:18333",
                "connected" : "outbound"
            }
        ]
    }
]
```

See also

- AddNode: attempts to add or remove a node from the addnode list, or to try a connection to a node once.

- GetPeerInfo: returns data about each connected network node.

GetAddressesByAccount

Requires wallet support.

The `getaddressesbyaccount` RPC returns a list of every address assigned to a particular account.

Parameter #1—the account name

Name	Type	Presence	Description
Account	string	Required (exactly 1)	The name of the account containing the addresses to get. To get addresses from the default account, pass an empty string (“”)

Result—a list of addresses

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing all addresses belonging to the specified account. If the account has no addresses, the array will be empty
Address	string (base58)	Optional (1 or more)	A P2PKH or P2SH address belonging to the account

Example from Bitcoin Core 0.10.0

Get the addresses assigned to the account “doc test”:

```
bitcoin-cli -testnet getaddressesbyaccount "doc test"
```

Result:

```
[
    "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN",
    "mft61jjkmiEJwJ7Zw3r1h344D6aL1xwhma",
    "mmXgiR6KAhZCyQ8ndr2BCfEq1wNG2UnyG6"
]
```

See also

- GetAccount: returns the name of the account associated with the given address.
- GetBalance: gets the balance in decimal bitcoins across all accounts or for a particular account.

GetBalance

Requires wallet support.

The `getbalance` RPC gets the balance in decimal bitcoins across all accounts or for a particular account.

Parameter #1—an account name

Name	Type	Presence	Description
Account	string	Optional	The name of an account to get the balance for. An empty string (“”) is the default account. The string

		(0 or 1)	`*` will get the balance for all accounts (this is the default behavior)

Parameter #2—the minimum number of confirmations

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Parameter #3—whether to include watch-only addresses

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—the balance in bitcoins

Name	Type	Presence	Description
`result`	number (bitcoins)	Required (exactly 1)	The balance of the account (or all accounts) in bitcoins

Examples from Bitcoin Core 0.10.0

Get the balance for the "test1" account, including transactions with at least one confirmation and those spent to watch-only addresses in that account.

```
bitcoin-cli -testnet getbalance "test1" 1 true
```

Result:

```
1.99900000
```

See also

- ListAccounts: lists accounts and their balances.
- GetReceivedByAccount: returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetReceivedByAddress: returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.

GetBestBlockHash

Added in Bitcoin Core 0.9.0

The `getbestblockhash` RPC returns the header hash of the most recent block on the best block chain.

Parameters: none

Result—hash of the tip from the best block chain

Name	Type	Presence	Description
`result`	string (hex)	Required (exactly 1)	The hash of the block header from the most recent block on the best block chain, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getbestblockhash
```

Result:

```
0000000000075c58ed39c3e50f99b32183d090aefa0cf8c324a82eea9b01a887
```

See also

- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlockHash: returns the header hash of a block at the given height in the local best block chain.

GetBlock

The `getblock` RPC gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.

Parameter #1—header hash

Name	Type	Presence	Description
Header Hash	string (hex)	Required (exactly 1)	The hash of the header of the block to get, encoded as hex in RPC byte order

Parameter #2—JSON or hex output

Name	Type	Presence	Description
Format	bool	Optional (0 or 1)	Set to `false` to get the block in serialized block format; set to `true` (the default) to get the decoded block as a JSON object

Result (if format was `false`)—a serialized block

Name	Type	Presence	Description
`result`	string (hex)/null	Required (exactly 1)	The requested block as a serialized block, encoded as hex, or JSON `null` if an error occurred

Result (if format was `true` or omitted)—a JSON block

Name	Type	Presence	Description
`result`	object/null	Required (exactly 1)	An object containing the requested block, or JSON `null` if an error occurred

→ `hash`	string (hex)	Required (exactly 1)	The hash of this block's block header encoded as hex in RPC byte order. This is the same as the hash provided in parameter #1
→ `confirmations`	number (int)	Required (exactly 1)	The number of confirmations the transactions in this block have, starting at 1 when this block is at the tip of the best block chain. This score will be -1 if the the block is not part of the best block chain
→ `size`	number (int)	Required (exactly 1)	The size of this block in serialized block format, counted in bytes
→ `height`	number (int)	Required (exactly 1)	The height of this block on its block chain
→ `version`	number (int)	Required (exactly 1)	This block's version number. See block version numbers
→ `merkleroot`	string (hex)	Required (exactly 1)	The merkle root for this block, encoded as hex in RPC byte order
→ `tx`	array	Required (exactly 1)	An array containing the TXIDs of all transactions in this block. The transactions appear in the array in the same order they appear in the serialized block
→ → TXID	string (hex)	Required (1 or more)	The TXID of a transaction in this block, encoded as hex in RPC byte order
→ `time`	number (int)	Required (exactly 1)	The value of the *time* field in the block header, indicating approximately when the block was created
→ `nonce`	number (int)	Required (exactly 1)	The nonce which was successful at turning this particular block into one that could be added to the best block chain
→ `bits`	string (hex)	Required (exactly 1)	The value of the *nBits* field in the block header, indicating the target threshold this block's header had to pass
→ `difficulty`	number (real)	Required (exactly 1)	The estimated amount of work done to find this block relative to the estimated amount of work done to find block 0
→ `chainwork`	string (hex)	Required (exactly 1)	The estimated number of block header hashes miners had to check from the genesis block to this block, encoded as big-endian hex
→ `previousblockhash`	string (hex)	Required (exactly 1)	The hash of the header of the previous block, encoded as hex in RPC byte order
→ `nextblockhash`	string (hex)	Optional (0 or 1)	The hash of the next block on the best block chain, if known, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

Get a block in raw hex:

```
bitcoin-cli -testnet getblock \
            000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39 \
            false
```

Result (wrapped):

```
02000000df11c014a8d798395b5059c722ebdf3171a4217ead71bf6e0e99f4c7\
000000004a6f6a2db225c81e77773f6f0457bcb05865a94900ed11356d0b7522\
8efb38c7785d6053ffff001d005d437001010000000100000000000000000000\
00000000000000000000000000000000000000000000ffffffff0d03b4770301\
64062f503253482fffffffff0100f9029500000000232103adb7d8ef6b63de74\
313e0cd4e07670d09a169b13e4eda2d650f529332c47646dac00000000
```

Get the same block in JSON:

```
bitcoin-cli -testnet getblock \
            000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39 \
            true
```

Result:

```
{
    "hash" : "000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39",
    "confirmations" : 88029,
    "size" : 189,
    "height" : 227252,
    "version" : 2,
    "merkleroot" : "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a",
    "tx" : [
        "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a"
    ],
    "time" : 1398824312,
    "nonce" : 1883462912,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "chainwork" : "000000000000000000000000000000000000000000000000083ada4a4009841a",
    "previousblockhash" : "00000000c7f4990e6ebf71ad7e21a47131dfeb22c759505b3998d7a814c011df",
    "nextblockhash" : "00000000afe1928529ac766f1237657819a11cfcc8ca6d67f119e868ed5b6188"
}
```

See also

- GetBlockHash: returns the header hash of a block at the given height in the local best block chain.
- GetBestBlockHash: returns the header hash of the most recent block on the best block chain.

GetBlockChainInfo

Added in Bitcoin Core 0.9.2

The `getblockchaininfo` RPC provides information about the current state of the block chain.

Parameters: none

Result—A JSON object providing information about the block chain

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Information about the current state of the local block chain
→ `chain`	string	Required (exactly 1)	The name of the block chain. One of `main` for mainnet, `test` for testnet, or `regtest` for regtest
→ `blocks`	number (int)	Required (exactly 1)	The number of validated blocks in the local best block chain. For a new node with just the hardcoded genesis block, this will be 0
			Added in Bitcoin Core 0.10.0

→ `headers`	number (int)	Required (exactly 1)	The number of validated headers in the local best headers chain. For a new node with just the hardcoded genesis block, this will be zero. This number may be higher than the number of *blocks*
→ `bestblockhash`	string (hex)	Required (exactly 1)	The hash of the header of the highest validated block in the best block chain, encoded as hex in RPC byte order. This is identical to the string returned by the `getbestblockhash` RPC
→ `difficulty`	number (real)	Required (exactly 1)	The difficulty of the highest-height block in the best block chain
→ `verificationprogress`	number (real)	Required (exactly 1)	Estimate of what percentage of the block chain transactions have been verified so far, starting at 0.0 and increasing to 1.0 for fully verified. May slightly exceed 1.0 when fully synced to account for transactions in the memory pool which have been verified before being included in a block
→ `chainwork`	string (hex)	Required (exactly 1)	The estimated number of block header hashes checked from the genesis block to this block, encoded as big-endian hex

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getblockchaininfo
```

Result:

```
{
    "chain" : "test",
    "blocks" : 315280,
    "headers" : 315280,
    "bestblockhash" : "000000000ebb17fb455e897b8f3e343eea1b07d926476d00bc66e2c0342ed50f",
    "difficulty" : 1.00000000,
    "verificationprogress" : 1.00000778,
    "chainwork" : "0000000000000000000000000000000000000000000000015e984b4fb9f9b350"
}
```

See also

- GetMiningInfo: returns various mining-related information.
- GetNetworkInfo: returns information about the node’s connection to the network.
- GetWalletInfo: provides information about the wallet.

GetBlockCount

The `getblockcount` RPC returns the number of blocks in the local best block chain.

Parameters: none

Result—the number of blocks in the local best block chain

Name	Type	Presence	Description
`result`	number (int)	Required (exactly 1)	The number of blocks in the local best block chain. For a new node with only the hardcoded genesis block, this number will be 0

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getblockcount
```

Result:

```
315280
```

See also

- GetBlockHash: returns the header hash of a block at the given height in the local best block chain.
- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.

GetBlockHash

The `getblockhash` RPC returns the header hash of a block at the given height in the local best block chain.

Parameter—a block height

Name	Type	Presence	Description
Block Height	number (int)	Required (exactly 1)	The height of the block whose header hash should be returned. The height of the hardcoded genesis block is 0

Result—the block header hash

Name	Type	Presence	Description
`result`	string (hex)/null	Required (exactly 1)	The hash of the block at the requested height, encoded as hex in RPC byte order, or JSON `null` if an error occurred

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getblockhash 240886
```

Result:

```
00000000a0faf83ab5799354ae9c11da2a2bd6db44058e03c528851dee0a3fff
```

See also

- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBestBlockHash: returns the header hash of the most recent block on the best block chain.

GetBlockTemplate

The `getblocktemplate` RPC gets a block template or proposal for use with mining software. For more information, please see the following resources:

- Bitcoin Wiki GetBlockTemplate
- BIP22

- BIP23

See also

- SetGenerate: enables or disables hashing to attempt to find the next block.
- GetMiningInfo: returns various mining-related information.
- SubmitBlock: accepts a block, verifies it is a valid addition to the block chain, and broadcasts it to the network. Extra parameters are ignored by Bitcoin Core but may be used by mining pools or other programs.
- PrioritiseTransaction: adds virtual priority or fee to a transaction, allowing it to be accepted into blocks mined by this node (or miners which use this node) with a lower priority or fee. (It can also remove virtual priority or fee, requiring the transaction have a higher priority or fee to be accepted into a locally-mined block.)

GetChainTips

Added in Bitcoin Core 0.10.0.

The `getchaintips` RPC returns information about the highest-height block (tip) of each local block chain.

Parameters: none

Result—an array of block chain tips

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array of JSON objects, with each object describing a chain tip. At least one tip—the local best block chain—will always be present
→ Tip	object	Required (1 or more)	An object describing a particular chain tip. The first object will always describe the active chain (the local best block chain)
→ → `height`	number (int)	Required (exactly 1)	The height of the highest block in the chain. A new node with only the genesis block will have a single tip with height of 0
→ → `hash`	string (hex)	Required (exactly 1)	The hash of the highest block in the chain, encoded as hex in RPC byte order
→ → `branchlen`	number (int)	Required (exactly 1)	The number of blocks that are on this chain but not on the main chain. For the local best block chain, this will be `0`; for all other chains, it will be at least `1`
→ → `status`	string	Required (exactly 1)	The status of this chain. Valid values are: • `active` for the local best block chain • `invalid` for a chain that contains one or more invalid blocks • `headers-only` for a chain with valid headers whose corresponding blocks both haven't been validated and aren't stored locally • `valid-headers` for a chain with valid headers whose corresponding blocks are stored locally, but which haven't been fully validated • `valid-fork` for a chain which is fully validated but which isn't part of the local best block chain (it was probably the local best block chain at some point) • `unknown` for a chain whose reason for not being the active chain is unknown

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getchaintips
```

```
[
    {
        "height" : 312647,
        "hash" : "000000000b1be96f87b31485f62c1361193304a5ad78acf47f9164ea4773a843",
        "branchlen" : 0,
        "status" : "active"
    },
    {
        "height" : 282072,
        "hash" : "00000000712340a499b185080f94b28c365d8adb9fc95bca541ea5e708f31028",
        "branchlen" : 5,
        "status" : "valid-fork"
    },
    {
        "height" : 281721,
        "hash" : "000000006e1f2a32199629c6c1fbd37766f5ce7e8c42bab0c6e1ae42b88ffe12",
        "branchlen" : 1,
        "status" : "valid-headers"
    },
]
```

See also

- GetBestBlockHash: returns the header hash of the most recent block on the best block chain.
- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlockChainInfo: provides information about the current state of the block chain.

GetConnectionCount

The `getconnectioncount` RPC returns the number of connections to other nodes.

Parameters: none

Result—the number of connections to other nodes

Name	Type	Presence	Description
`result`	number (int)	Required (exactly 1)	The total number of connections to other nodes (both inbound and outbound)

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getconnectioncount
```

Result:

```
14
```

See also

- GetNetTotals: returns information about network traffic, including bytes in, bytes out, and the current time.
- GetPeerInfo: returns data about each connected network node.
- GetNetworkInfo: returns information about the node’s connection to the network.

GetDifficulty

The `getdifficulty` RPC

Parameters: none

Result—the current difficulty

Name	Type	Presence	Description
`result`	number (real)	Required (exactly 1)	The difficulty of creating a block with the same target threshold (nBits) as the highest-height block in the local best block chain. The number is a a multiple of the minimum difficulty

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getdifficulty
```

Result:

```
1.00000000
```

See also

- GetNetworkHashPS: returns the estimated current or historical network hashes per second based on the last *n* blocks.
- GetMiningInfo: returns various mining-related information.

GetGenerate

Requires wallet support.

The `getgenerate` RPC returns true if the node is set to generate blocks using its CPU.

Parameters: none

Result—whether the server is set to generate blocks

Name	Type	Presence	Description
`result`	bool	Required (exactly 1)	Set to `true` if the server is set to generate blocks; set to `false` if it is not

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getgenerate
```

Result:

```
false
```

See also

- SetGenerate: enables or disables hashing to attempt to find the next block.
- GetMiningInfo: returns various mining-related information.
- GetHashesPerSec: was removed in Bitcoin Core master (unreleased). It returned a recent hashes per second performance measurement when the node was generating blocks.

GetHashesPerSec

Requires wallet support.

The `gethashespersec` RPC was removed in Bitcoin Core master (unreleased). It returned a recent hashes per second performance measurement when the node was generating blocks.

Parameters: none

Result—the number of hashes your computer generated per second

Name	Type	Presence	Description
`result`	number (int)	Required (exactly 1)	If generation is enabled, the number of hashes per second your computer most recently generated. If generation is disabled, the value `0`

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet gethashespersec
```

Result:

```
1995356
```

See also

- SetGenerate: enables or disables hashing to attempt to find the next block.
- GetMiningInfo: returns various mining-related information.

GetInfo

The `getinfo` RPC prints various information about the node and the network.

⚠ **Warning:** `getinfo` will be removed in a later version of Bitcoin Core. Use the RPCs listed in the See Also subsection below instead.

Parameters: none

Result—information about the node and network

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Information about this node and the network
→ `version`	number (int)	Required (exactly 1)	This node’s version of Bitcoin Core in its internal integer format. For example, Bitcoin Core 0.9.2 has the integer version number 90200
→ `protocolversion`	number (int)	Required (exactly 1)	The protocol version number used by this node. See the protocol versions section for more information
→ `walletversion`	number (int)	Optional (0 or 1)	The version number of the wallet. Only returned if wallet support is enabled
→ `balance`	number (bitcoins)	Optional (0 or 1)	The balance of the wallet in bitcoins. Only returned if wallet support is enabled
→	number	Required	The number of blocks in the local best block chain. A new node with only the

`blocks`	(int)	(exactly 1)	hardcoded genesis block will return `0`
→ `timeoffset`	number (int)	Required (exactly 1)	The offset of the node's clock from the computer's clock (both in UTC) in seconds. The offset may be up to 4200 seconds (70 minutes)
→ `connections`	number (int)	Required (exactly 1)	The total number of open connections (both outgoing and incoming) between this node and other nodes
→ `proxy`	string	Required (exactly 1)	The hostname/IP address and port number of the proxy, if set, or an empty string if unset
→ `difficulty`	number (real)	Required (exactly 1)	The difficulty of the highest-height block in the local best block chain
→ `testnet`	bool	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if this node is on testnet; set to `false` if this node is on mainnet or a regtest
→ `keypoololdest`	number (int)	Optional (0 or 1)	The date as Unix epoch time when the oldest key in the wallet key pool was created; useful for only scanning blocks created since this date for transactions. Only returned if wallet support is enabled
→ `keypoolsize`	number (int)	Optional (0 or 1)	The number of keys in the wallet keypool. Only returned if wallet support is enabled
→ `paytxfee`	number (bitcoins)	Optional (0 or 1)	The minimum fee to pay per kilobyte of transaction; may be `0`. Only returned if wallet suuport is enabled
→ `relayfee`	number (bitcoins)	Required (exactly 1)	The minimum fee a low-priority transaction must pay in order for this node to accept it into its memory pool
→ `unlocked_until`	number (int)	Optional (0 or 1)	The Unix epoch time when the wallet will automatically re-lock. Only displayed if wallet encryption is enabled. Set to `0` if wallet is currently locked
→ `errors`	string	Required (exactly 1)	A plain-text description of any errors this node has encountered or detected. If there are no errors, an empty string will be returned. This is not related to the JSON-RPC `error` field

Example from Bitcoin Core 0.10.0 with wallet support enabled

```
bitcoin-cli -testnet getinfo
```

Result:

```
{
    "version" : 100000,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 1.27007770,
    "blocks" : 315281,
    "timeoffset" : 0,
    "connections" : 9,
    "proxy" : "",
    "difficulty" : 1.00000000,
    "testnet" : true,
    "keypoololdest" : 1418924649,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "relayfee" : 0.00001000,
    "errors" : ""
}
```

See also

- GetBlockChainInfo: provides information about the current state of the block chain.
- GetMemPoolInfo: returns information about the node’s current transaction memory pool.
- GetMiningInfo: returns various mining-related information.
- GetNetworkInfo: returns information about the node’s connection to the network.
- GetWalletInfo: provides information about the wallet.

GetMemPoolInfo

Added in Bitcoin Core 0.10.0

The `getmempoolinfo` RPC returns information about the node’s current transaction memory pool.

Parameters: none

Result—information about the transaction memory pool

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	A object containing information about the memory pool
→ `size`	number (int)	Required (exactly 1)	The number of transactions currently in the memory pool
→ `bytes`	number (int)	Required (exactly 1)	The total number of bytes in the transactions in the memory pool

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getmempoolinfo
```

Result:

```
{
    "size" : 37,
    "bytes" : 9423
}
```

See also

- GetBlockChainInfo: provides information about the current state of the block chain.
- GetRawMemPool: returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object.
- GetTxOutSetInfo: returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.

GetMiningInfo

The `getmininginfo` RPC returns various mining-related information.

Parameters: none

Result—various mining-related information

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Various mining-related information
→ `blocks`	number (int)	Required (exactly 1)	The height of the highest block on the local best block chain
→ `currentblocksize`	number (int)	Required (exactly 1)	If generation was enabled since the last time this node was restarted, this is the size in bytes of the last block built by this node for header hash checking. Otherwise, the value `0`
→ `currentblocktx`	number (int)	Required (exactly 1)	If generation was enabled since the last time this node was restarted, this is the number of transactions in the last block built by this node for header hash checking. Otherwise, this is the value `0`
→ `difficulty`	number (real)	Required (exactly 1)	If generation was enabled since the last time this node was restarted, this is the difficulty of the highest-height block in the local best block chain. Otherwise, this is the value `0`
→ `errors`	string	Required (exactly 1)	A plain-text description of any errors this node has encountered or detected. If there are no errors, an empty string will be returned. This is not related to the JSON-RPC `error` field
→ `genproclimit`	number (int)	Required (exactly 1)	The limit on the number of processors to use for generation. If generation was enabled since the last time this node was restarted, this is the number used in the second parameter of the `setgenerate` RPC (or the default). Otherwise, it is `-1`
→ `networkhashps`	number (int)	Required (exactly 1)	An estimate of the number of hashes per second the network is generating to maintain the current difficulty. See the `getnetworkhashps` RPC for configurable access to this data
→ `pooledtx`	number (int)	Required (exactly 1)	The number of transactions in the memory pool
→ `testnet`	bool	Required (exactly 1)	Set to `true` if this node is running on testnet. Set to `false` if this node is on mainnet or a regtest
→ `chain`	string	Required (exactly 1)	Set to `main` for mainnet, `test` for testnet, and `regtest` for regtest
→ `generate`	bool	Optional (0 or 1)	Set to `true` if generation is currently enabled; set to `false` if generation is currently disabled. Only returned if the node has wallet support enabled
→ `hashespersec`	number (int)	Optional (0 or 1)	*Removed in Bitcoin Core master (unreleased)* The approximate number of hashes per second this node is generating across all CPUs, if generation is enabled. Otherwise `0`. Only returned if the node has wallet support enabled

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getmininginfo
```

Result:

```
{
    "blocks" : 313168,
    "currentblocksize" : 1819,
    "currentblocktx" : 3,
    "difficulty" : 1.00000000,
    "errors" : "",
    "genproclimit" : 1,
    "networkhashps" : 5699977416637,
    "pooledtx" : 8,
    "testnet" : true,
    "chain" : "test",
    "generate" : true,
    "hashespersec" : 921200
}
```

See also

- GetMemPoolInfo: returns information about the node's current transaction memory pool.
- GetRawMemPool: returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object.
- GetBlockTemplate: gets a block template or proposal for use with mining software.
- SetGenerate: enables or disables hashing to attempt to find the next block.

GetNetTotals

The `getnettotals` RPC returns information about network traffic, including bytes in, bytes out, and the current time.

Parameters: none

Result—the current bytes in, bytes out, and current time

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object containing information about the node's network totals
→ `totalbytesrecv`	number (int)	Required (exactly 1)	The total number of bytes received since the node was last restarted
→ `totalbytessent`	number (int)	Required (exactly 1)	The total number of bytes sent since the node was last restarted
→ `timemillis`	number (int)	Required (exactly 1)	Unix epoch time in milliseconds according to the operating system's clock (not the node adjusted time)

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getnettotals
```

Result:

```
{
    "totalbytesrecv" : 723992206,
    "totalbytessent" : 16846662695,
    "timemillis" : 1419268217354
}
```

See also

- GetNetworkInfo: returns information about the node’s connection to the network.
- GetPeerInfo: returns data about each connected network node.

GetNetworkHashPS

The `getnetworkhashps` RPC returns the estimated current or historical network hashes per second based on the last *n* blocks.

Parameter #1—number of blocks to average

Name	Type	Presence	Description
Blocks	number (int)	Optional (0 or 1)	The number of blocks to average together for calculating the estimated hashes per second. Default is `120`. Use `-1` to average all blocks produced since the last difficulty change

Parameter #2—block height

Name	Type	Presence	Description
Height	number (int)	Optional (0 or 1)	The height of the last block to use for calculating the average. Defaults to `-1` for the highest-height block on the local best block chain. If the specified height is higher than the highest block on the local best block chain, it will be interpreted the same as `-1`

Result—estimated hashes per second

Name	Type	Presence	Description
`result`	number (int)	Required (exactly 1)	The estimated number of hashes per second based on the parameters provided. May be 0 (for Height=`0`, the genesis block) or a negative value if the highest-height block averaged has a block header time earlier than the lowest-height block averaged

Example from Bitcoin Core 0.10.0

Get the average hashes per second for all the blocks since the last difficulty change before block 227255.

```
bitcoin-cli -testnet getnetworkhashps -1 227255
```

Result:

```
79510076167
```

See also

- GetDifficulty: returns the proof-of-work difficulty as a multiple of the minimum difficulty.
- GetBlock: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.

GetNetworkInfo

Added in Bitcoin Core 0.9.2.

The `getnetworkinfo` RPC returns information about the node’s connection to the network.

Parameters: none

Result—information about the node's connection to the network

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Information about this node's connection to the network
→ `version`	number	Required (exactly 1)	This node's version of Bitcoin Core in its internal integer format. For example, Bitcoin Core 0.9.2 has the integer version number 90200
→ `subversion`	string	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* The user agent this node sends in its `version` message
→ `protocolversion`	number (int)	Required (exactly 1)	The protocol version number used by this node. See the protocol versions section for more information
→ `timeoffset`	number (int)	Required (exactly 1)	The offset of the node's clock from the computer's clock (both in UTC) in seconds. The offset may be up to 4200 seconds (70 minutes)
→ `connections`	number (int)	Required (exactly 1)	The total number of open connections (both outgoing and incoming) between this node and other nodes
→ `proxy`	string	Required (exactly 1)	The hostname/IP address and port number of the proxy, if set, or an empty string if unset
→ `relayfee`	number (bitcoins)	Required (exactly 1)	The minimum fee a low-priority transaction must pay in order for this node to accept it into its memory pool
→ `localservices`	string (hex)	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* The services supported by this node as advertised in its `version` message
→ `networks`	array	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* An array with three objects: one describing the IPv4 connection, one describing the IPv6 connection, and one describing the Tor hidden service (onion) connection
→ → Network	object	Optional (0 to 3)	An object describing a network. If the network is unroutable, it will not be returned
→ → → `name`	string	Required (exactly 1)	The name of the network. Either `ipv4`, `ipv6`, or `onion`
→ → → `limited`	bool	Required (exactly 1)	Set to `true` if only connections to this network are allowed according to the `-onlynet` Bitcoin Core command-line/configuration-file parameter. Otherwise set to `false`
→ → → `reachable`	bool	Required (exactly 1)	Set to `true` if connections can be made to or from this network. Otherwise set to `false`
→ → → `proxy`	string	Required (exactly 1)	The hostname and port of any proxy being used for this network. If a proxy is not in use, an empty string
→ → → `localaddresses`	array	Required (exactly 1)	An array of objects each describing the local addresses this node believes it listens on

→ → → → Address	object	Optional (0 or more)	An object describing a particular address this node believes it listens on
→ → → → → `address`	string	Required (exactly 1)	An IP address or .onion address this node believes it listens on. This may be manually configured, auto detected, or based on `version` messages this node received from its peers
→ → → → → `port`	number (int)	Required (exactly 1)	The port number this node believes it listens on for the associated `address`. This may be manually configured, auto detected, or based on `version` messages this node received from its peers
→ → → → → `score`	number (int)	Required (exactly 1)	The self-assigned score this node gives to this connection; higher scores means the node thinks this connection is better

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getnetworkinfo
```

Result (actual addresses have been replaced with reserved addresses):

```
{
    "version" : 100000,
    "subversion" : "/Satoshi:0.10.0/",
    "protocolversion" : 70002,
    "localservices" : "0000000000000001",
    "timeoffset" : 0,
    "connections" : 51,
    "networks" : [
        {
            "name" : "ipv4",
            "limited" : false,
            "reachable" : true,
            "proxy" : ""
        },
        {
            "name" : "ipv6",
            "limited" : false,
            "reachable" : true,
            "proxy" : ""
        },
        {
            "name" : "onion",
            "limited" : false,
            "reachable" : false,
            "proxy" : ""
        }
    ],
    "relayfee" : 0.00001000,
    "localaddresses" : [
        {
            "address" : "192.0.2.113",
            "port" : 18333,
            "score" : 6470
        },
        {
            "address" : "0600:3c03::f03c:91ff:fe89:dfc4",
            "port" : 18333,
            "score" : 2029
        }
    ]
}
```

See also

- GetPeerInfo: returns data about each connected network node.

- GetNetTotals: returns information about network traffic, including bytes in, bytes out, and the current time.

GetNewAddress

Requires wallet support.

The `getnewaddress` RPC returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.

Parameter #1—an account name

Name	Type	Presence	Description
Account	string	Optional (0 or 1)	The name of the account to put the address in. The default is the default account, an empty string (“”)

Result—a bitcoin address never previously returned

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	A P2PKH address which has not previously been returned by this RPC. The address will be marked as a receiving address in the wallet. The address may already have been part of the keypool, so other RPCs such as the `dumpwallet` RPC may have disclosed it previously. If the wallet is unlocked, its keypool will also be filled to its max (by default, 100 unused keys). If the wallet is locked and its keypool is empty, this RPC will fail

Example from Bitcoin Core 0.10.0

Create a new address in the “doc test” account:

```
bitcoin-cli -testnet getnewaddress "doc test"
```

Result:

```
mft61jjkmiEJwJ7Zw3r1h344D6aL1xwhma
```

See also

- GetAccountAddress: returns the current Bitcoin address for receiving payments to this account. If the account doesn’t exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address.
- GetRawChangeAddress: returns a new Bitcoin address for receiving change. This is for use with raw transactions, not normal use.
- GetBalance: gets the balance in decimal bitcoins across all accounts or for a particular account.

GetPeerInfo

The `getpeerinfo` RPC returns data about each connected network node.

Parameters: none

Result—information about each currently-connected network node

Name	Type	Presence	Description

`result`	array	Required (exactly 1)	An array of objects each describing one connected node. If there are no connections, the array will be empty
→ Node	object	Optional (0 or more)	An object describing a particular connected node
→ → `id`	number (int)	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* The node's index number in the local node address database
→ → `addr`	string	Required (exactly 1)	The IP address and port number used for the connection to the remote node
→ → `addrlocal`	string	Optional (0 or 1)	Our IP address and port number according to the remote node. May be incorrect due to error or lying. Many SPV nodes set this to `127.0.0.1:8333`
→ → `services`	string (hex)	Required (exactly 1)	The services advertised by the remote node in its `version` message
→ → `lastsend`	number (int)	Required (exactly 1)	The Unix epoch time when we last successfully sent data to the TCP socket for this node
→ → `lastrecv`	number (int)	Required (exactly 1)	The Unix epoch time when we last received data from this node
→ → `bytessent`	number (int)	Required (exactly 1)	The total number of bytes we've sent to this node
→ → `bytesrecv`	number (int)	Required (exactly 1)	The total number of bytes we've received from this node
→ → `conntime`	number (int)	Required (exactly 1)	The Unix epoch time when we connected to this node
→ → `pingtime`	number (real)	Required (exactly 1)	The number of seconds this node took to respond to our last P2P `ping` message
→ → `pingwait`	number (real)	Optional (0 or 1)	The number of seconds we've been waiting for this node to respond to a P2P `ping` message. Only shown if there's an outstanding `ping` message
→ → `version`	number (int)	Required (exactly 1)	The protocol version number used by this node. See the protocol versions section for more information
→ → `subver`	string	Required (exactly 1)	The user agent this node sends in its `version` message. This string will have been sanitized to prevent corrupting the JSON results. May be an empty string
→ → `inbound`	bool	Required (exactly 1)	Set to `true` if this node connected to us; set to `false` if we connected to this node
→ → `startingheight`	number (int)	Required (exactly 1)	The height of the remote node's block chain when it connected to us as reported in its `version` message
→ → `banscore`	number (int)	Required (exactly 1)	The ban score we've assigned the node based on any misbehavior it's made. By default, Bitcoin Core disconnects when the ban score reaches `100`
→ →	number	Required	*Added in Bitcoin Core 0.10.0* The highest-height header we have in common with this node based the last P2P

`synced_headers`	(int)	(exactly 1)	`headers` message it sent us. If a `headers` message has not been received, this will be set to `-1`
→ → `synced_blocks`	number (int)	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* The highest-height block we have in common with this node based on P2P `inv` messages this node sent us. If no block `inv` messages have been received from this node, this will be set to `-1`
→ → `syncnode`	bool	Required (exactly 1)	*Removed in Bitcoin Core 0.10.0* Whether we're using this node as our syncnode during initial block download
→ → `inflight`	array	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* An array of blocks which have been requested from this peer. May be empty
→ → → Blocks	number (int)	Optional (0 or more)	The height of a block being requested from the remote peer
→ → `whitelisted`	bool	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if the remote peer has been whitelisted; otherwise, set to `false`. Whitelisted peers will not be banned if their ban score exceeds the maximum (100 by default). By default, peers connecting from localhost are whitelisted

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getpeerinfo
```

Result (edited to show only a single entry, with IP addresses changed to RFC5737 reserved IP addresses):

```
[
    {
        "id" : 9,
        "addr" : "192.0.2.113:18333",
        "addrlocal" : "192.0.2.51:18333",
        "services" : "0000000000000002",
        "lastsend" : 1419277992,
        "lastrecv" : 1419277992,
        "bytessent" : 4968,
        "bytesrecv" : 105078,
        "conntime" : 1419265985,
        "pingtime" : 0.05617800,
        "version" : 70001,
        "subver" : "/Satoshi:0.8.6/",
        "inbound" : false,
        "startingheight" : 315280,
        "banscore" : 0,
        "synced_headers" : -1,
        "synced_blocks" : -1,
        "inflight" : [
        ],
        "whitelisted" : false
    }
]
```

See also

- GetAddedNodeInfo: returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the `addnode` RPC will have their information displayed.

- GetNetTotals: returns information about network traffic, including bytes in, bytes out, and the current time.
- GetNetworkInfo: returns information about the node’s connection to the network.

GetRawChangeAddress

Requires wallet support.

The `getrawchangeaddress` RPC returns a new Bitcoin address for receiving change. This is for use with raw transactions, not normal use.

Parameters: none

Result—a P2PKH address which can be used in raw transactions

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	A P2PKH address which has not previously been returned by this RPC. The address will be removed from the keypool but not marked as a receiving address, so RPCs such as the `dumpwallet` RPC will show it as a change address. The address may already have been part of the keypool, so other RPCs such as the `dumpwallet` RPC may have disclosed it previously. If the wallet is unlocked, its keypool will also be filled to its max (by default, 100 unused keys). If the wallet is locked and its keypool is empty, this RPC will fail

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getrawchangeaddress
```

Result:

```
mnycUc8FRjJodfKhaj9QBZs2PwxxYoWqaK
```

See also

- GetNewAddress: returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.
- GetAccountAddress: returns the current Bitcoin address for receiving payments to this account. If the account doesn’t exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address.

GetRawMemPool

The `getrawmempool` RPC returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object.

Parameter—desired output format

Name	Type	Presence	Description
Format	bool	Optional (0 or 1)	Set to `true` to get verbose output describing each transaction in the memory pool; set to `false` (the default) to only get an array of TXIDs for transactions in the memory pool

Result (format `false`)—an array of TXIDs

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array of TXIDs belonging to transactions in the memory pool. The array may be empty if there are no transactions in the memory pool
→ TXID	string	Optional (0 or more)	The TXID of a transaction in the memory pool, encoded as hex in RPC byte order

Result (format: `true`)—a JSON object describing each transaction

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	A object containing transactions currently in the memory pool. May be empty
→ TXID	string : object	Optional (0 or more)	The TXID of a transaction in the memory pool, encoded as hex in RPC byte order
→ → `size`	number (int)	Required (exactly 1)	The size of the serialized transaction in bytes
→ → `fee`	number (bitcoins)	Required (exactly 1)	The transaction fee paid by the transaction in decimal bitcoins
→ → `time`	number (int)	Required (exactly 1)	The time the transaction entered the memory pool, Unix epoch time format
→ → `height`	number (int)	Required (exactly 1)	The block height when the transaction entered the memory pool
→ → `startingpriority`	number (int)	Required (exactly 1)	The priority of the transaction when it first entered the memory pool
→ → `currentpriority`	number (int)	Required (exactly 1)	The current priority of the transaction
→ → `depends`	array	Required (exactly 1)	An array holding TXIDs of unconfirmed transactions this transaction depends upon. Those transactions must be part of a block before this transaction can be added to a block, although all transactions may be included in the same block. The array may be empty
→ → → Depends TXID	string	Optional (0 or more)	The TXIDs of any unconfirmed transactions this transaction depends upon, encoded as hex in RPC byte order

Examples from Bitcoin Core 0.10.0

The default (`false`):

```
bitcoin-cli -testnet getrawmempool
```

Result:

```
[
    "2b1f41d6f1837e164d6d6811d3d8dad2e66effbd1058cd9ed7bdbe1cab20ae03",
    "2baa1f49ac9b951fa781c4c95814333a2f3eda71ed3d0245cd76c2829b3ce354"
]
```

Verbose output (`true`):

```
bitcoin-cli -testnet getrawmempool true
```

Result:

```
{
    "2baa1f49ac9b951fa781c4c95814333a2f3eda71ed3d0245cd76c2829b3ce354" : {
        "size" : 191,
        "fee" : 0.00020000,
        "time" : 1398867772,
        "height" : 227310,
        "startingpriority" : 54545454.54545455,
        "currentpriority" : 54545454.54545455,
        "depends" : [
        ]
    }
}
```

See also

- GetMemPoolInfo: returns information about the node's current transaction memory pool.
- GetTxOutSetInfo: returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.

GetRawTransaction

The `getrawtransaction` RPC gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings.

Note: if you begin using `txindex=1` after downloading the block chain, you must rebuild your indexes by starting Bitcoin Core with the option `-reindex`. This may take several hours to complete, during which time your node will not process new blocks or transactions. This reindex only needs to be done once.

Parameter #1—the TXID of the transaction to get

Name	Type	Presence	Description
TXID	string (hex)	Required (exactly 1)	The TXID of the transaction to get, encoded as hex in RPC byte order

Parameter #2—whether to get the serialized or decoded transaction

Name	Type	Presence	Description
Verbose	number (int)	Optional (0 or 1)	Set to `0` (the default) to return the serialized transaction as hex. Set to `1` to return a decoded transaction

Result (if transaction not found)—`null`

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	If the transaction wasn't found, the result will be JSON `null`. This can occur because the transaction doesn't exist in the block chain or memory pool, or because it isn't part of the transaction index. See the Bitcoin Core `-help` entry for `-txindex`

Result (if verbose=`0`)—the serialized transaction

Name	Type	Presence	Description
`result`	string (hex)	Required (exactly 1)	If the transaction was found, this will be the serialized transaction encoded as hex

Result (if verbose=`1`)—the decoded transaction

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	If the transaction was found, this will be an object describing it
→ `txid`	string (hex)	Required (exactly 1)	The transaction's TXID encoded as hex in RPC byte order
→ `version`	number (int)	Required (exactly 1)	The transaction format version number
→ `locktime`	number (int)	Required (exactly 1)	The transaction's locktime: either a Unix epoch date or block height; see the Locktime parsing rules
→ `vin`	array	Required (exactly 1)	An array of objects with each object being an input vector (vin) for this transaction. Input objects will have the same order within the array as they have in the transaction, so the first input listed will be input 0
→ → Input	object	Required (1 or more)	An object describing one of this transaction's inputs. May be a regular input or a coinbase
→ → → `txid`	string	Optional (0 or 1)	The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction
→ → → `vout`	number (int)	Optional (0 or 1)	The output index number (vout) of the outpoint being spent. The first output in a transaction has an index of `0`. Not present if this is a coinbase transaction
→ → → `scriptSig`	object	Optional (0 or 1)	An object describing the signature script of this input. Not present if this is a coinbase transaction
→ → → → `asm`	string	Required (exactly 1)	The signature script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The signature script encoded as hex
→ → → `coinbase`	string (hex)	Optional (0 or 1)	The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction
→ → → `sequence`	number (int)	Required (exactly 1)	The input sequence number
→ `vout`	array	Required (exactly 1)	An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0
→ → Output	object	Required (1 or more)	An object describing one of this transaction's outputs

→ → → `value`	number (bitcoins)	Required (exactly 1)	The number of bitcoins paid to this output. May be `0`
→ → → `n`	number (int)	Required (exactly 1)	The output index number of this output within this transaction
→ → → `scriptPubKey`	object	Required (exactly 1)	An object describing the pubkey script
→ → → → `asm`	string	Required (exactly 1)	The pubkey script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The pubkey script encoded as hex
→ → → → `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for `nulldata` or `nonstandard` script types (see the `type` key below)
→ → → → `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script • `pubkeyhash` for a P2PKH script • `scripthash` for a P2SH script • `multisig` for a bare multisig script • `nulldata` for nulldata scripts • `nonstandard` for unknown scripts
→ → → → `addresses`	string : array	Optional (0 or 1)	The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for `nulldata` or `nonstandard` script types
→ → → → → Address	string	Required (1 or more)	A P2PKH or P2SH address
→ `blockhash`	string (hex)	Optional (0 or 1)	If the transaction has been included in a block on the local best block chain, this is the hash of that block encoded as hex in RPC byte order
→ `confirmations`	number (int)	Required (exactly 1)	If the transaction has been included in a block on the local best block chain, this is how many confirmations it has. Otherwise, this is `0`
→ `time`	number (int)	Optional (0 or 1)	If the transaction has been included in a block on the local best block chain, this is the block header time of that block (may be in the future)
→ `blocktime`	number (int)	Optional (0 or 1)	This field is currently identical to the time field described above

Examples from Bitcoin Core 0.10.0

A transaction in serialized transaction format:

```
bitcoin-cli -testnet getrawtransaction \
  ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e
```

Result (wrapped):

```
0100000001268a9ad7bfb21d3c086f0ff28f73a064964aa069ebb69a9e437da8\
5c7e55c7d7000000006b483045022100ee69171016b7dd218491faf6e13f53d4\
0d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed28\
8d374397d30dff541b2dd45a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc\
8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326ffffffff0350ac60020000\
00001976a91456847befbd2360df0e35b4e3b77bae48585ae06888ac80969800\
000000001976a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac002d\
3101000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac\
00000000
```

Get the same transaction in JSON:

```
bitcoin-cli -testnet getrawtransaction \
ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e \
1
```

Result:

```
{
    "hex" : "0100000001268a9ad7bfb21d3c086f0ff28f73a064964aa069ebb69a9e437da85c7e55c7d7000000006b483045022100ee69171016b7dd218491f
    "txid" : "ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "d7c7557e5ca87d439e9ab6eb69a04a9664a0738ff20f6f083c1db2bfd79a8a26",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed2
                "hex" : "483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982e
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.39890000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "moQR7i8XM4rSGoNwEsw3h4YEuduuP6mxw7"
                ]
            }
        },
        {
            "value" : 0.10000000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN"
                ]
            }
        },
        {
            "value" : 0.20000000,
            "n" : 2,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 0dfc8bafc8419853b34d5e072ad37d1a5159f584 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe"
                ]
            }
        }
    ],
    "blockhash" : "00000000103e0091b7d27e5dc744a305108f0c752be249893c749e19c1c82317",
    "confirmations" : 88192,
    "time" : 1398734825,
    "blocktime" : 1398734825
}
```

See also

- GetTransaction: gets detailed information about an in-wallet transaction.

GetReceivedByAccount

Requires wallet support.

The `getreceivedbyaccount` RPC returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.

Parameter #1—the account name

Name	Type	Presence	Description
Account	string	Required (exactly 1)	The name of the account containing the addresses to get. For the default account, use an empty string (“”)

Parameter #2—the minimum number of confirmations

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Result—the number of bitcoins received

Name	Type	Presence	Description
`result`	number (bitcoins)	Required (exactly 1)	The number of bitcoins received by the account. May be `0`

Example from Bitcoin Core 0.10.0

Get the bitcoins received by the “doc test” account with six or more confirmations:

```
bitcoin-cli -testnet getreceivedbyaccount "doc test" 6
```

Result:

```
0.30000000
```

See also

- GetReceivedByAddress: returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.
- ListAccounts: lists accounts and their balances.

GetReceivedByAddress

Requires wallet support.

The `getreceivedbyaddress` RPC returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.

Parameter #1—the address

Name	Type	Presence	Description
Address	string	Required (exactly 1)	The address whose transactions should be tallied

Parameter #2—the minimum number of confirmations

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Result—the number of bitcoins received

Name	Type	Presence	Description
`result`	number (bitcoins)	Required (exactly 1)	The number of bitcoins received by the address, excluding coinbase transactions. May be `0`

Example from Bitcoin Core 0.10.0

Get the bitcoins received for a particular address, only counting transactions with six or more confirmations:

```
bitcoin-cli -testnet getreceivedbyaddress mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN 6
```

Result:

```
0.30000000
```

See also

- GetReceivedByAccount: returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.
- ListAccounts: lists accounts and their balances.

GetTransaction

Requires wallet support.

The `gettransaction` RPC gets detailed information about an in-wallet transaction.

Parameter #1—a transaction identifier (TXID)

Name	Type	Presence	Description
TXID	string (hex)	Required (exactly 1)	The TXID of the transaction to get details about. The TXID must be encoded as hex in RPC byte order

Parameter #2—whether to include watch-only addresses in details and calculations

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—a description of the transaction

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing how the transaction affects the wallet
→ `amount`	number (bitcoins)	Required (exactly 1)	A positive number of bitcoins if this transaction increased the total wallet balance; a negative number of bitcoins if this transaction decreased the total wallet balance, or `0` if the transaction had no net effect on wallet balance
→ `fee`	number (bitcoins)	Optional (0 or 1)	If an outgoing transaction, this is the fee paid by the transaction reported as negative bitcoins
→ `confirmations`	number (int)	Required (exactly 1)	The number of confirmations the transaction has received. Will be `0` for unconfirmed and `-1` for conflicted
→ `generated`	bool	Optional (0 or 1)	Set to `true` if the transaction is a coinbase. Not returned for regular transactions
→ `blockhash`	string (hex)	Optional (0 or 1)	Only returned for confirmed transactions. The hash of the block on the local best block chain which includes this transaction, encoded as hex in RPC byte order
→ `blockindex`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block height of the block on the local best block chain which includes this transaction
→ `blocktime`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block header time (Unix epoch time) of the block on the local best block chain which includes this transaction
→ `txid`	string (hex)	Required (exactly 1)	The TXID of the transaction, encoded as hex in RPC byte order
→	array	Required	An array containing the TXIDs of other transactions that spend the same inputs

`walletconflicts`		(exactly 1)	(UTXOs) as this transaction. Array may be empty
→ → TXID	string (hex)	Optional (0 or more)	The TXID of a conflicting transaction, encoded as hex in RPC byte order
→ `time`	number (int)	Required (exactly 1)	A Unix epoch time when the transaction was added to the wallet
→ `timerecived`	number (int)	Required (exactly 1)	A Unix epoch time when the transaction was detected by the local node, or the time of the block on the local best block chain that included the transaction
→ `comment`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction. Only returned if a comment was added
→ `to`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction identifying who the transaction was sent to. Only returned if a comment-to was added
→ `details`	array	Required (exactly 1)	An array containing one object for each input or output in the transaction which affected the wallet
→ → `involvesWatchonly`	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if the input or output involves a watch-only address. Otherwise not returned
→ → `account`	string	Required (exactly 1)	The account which the payment was credited to or debited from. May be an empty string (“”) for the default account
→ → `address`	string (base58)	Optional (0 or 1)	If an output, the address paid (may be someone else’s address not belonging to this wallet). If an input, the address paid in the previous output. May be empty if the address is unknown, such as when paying to a non-standard pubkey script
→ → `category`	string	Required (exactly 1)	Set to one of the following values: • `send` if sending payment • `receive` if this wallet received payment in a regular transaction • `generate` if a matured and spendable coinbase • `immature` if a coinbase that is not spendable yet • `orphan` if a coinbase from a block that’s not in the local best block chain
→ → `amount`	number (bitcoins)	Required (exactly 1)	A negative bitcoin amount if sending payment; a positive bitcoin amount if receiving payment (including coinbases)
→ → `vout`	number (int)	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* For an output, the output index (vout) for this output in this transaction. For an input, the output index for the output being spent in its transaction. Because inputs list the output indexes from previous transactions, more than one entry in the details array may have the same output index
→ → `fee`	number (bitcoins)	Optional (0 or 1)	If sending payment, the fee paid as a negative bitcoins value. May be `0`. Not returned if receiving payment
→ `hex`	string (hex)	Required (exactly 1)	The transaction in serialized transaction format

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet gettransaction \
  5a7d24cd665108c66b2d56146f244932edae4e2376b561b3d396d5ae017b9589
```

Result:

```
{
    "amount" : 0.00000000,
    "fee" : 0.00000000,
    "confirmations" : 106670,
    "blockhash" : "000000008b630b3aae99b6fe215548168bed92167c47a2f7ad4df41e571bcb51",
    "blockindex" : 1,
    "blocktime" : 1396321351,
    "txid" : "5a7d24cd665108c66b2d56146f244932edae4e2376b561b3d396d5ae017b9589",
    "walletconflicts" : [
    ],
    "time" : 1396321351,
    "timereceived" : 1418924711,
    "details" : [
        {
            "account" : "",
            "address" : "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN",
            "category" : "send",
            "amount" : -0.10000000,
            "vout" : 0,
            "fee" : 0.00000000
        },
        {
            "account" : "doc test",
            "address" : "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN",
            "category" : "receive",
            "amount" : 0.10000000,
            "vout" : 0
        }
    ],
    "hex" : "0100000001cde58f2e37d000eabbb60d9cf0b79ddf67cede6dba58732539983fa341dd5e6c010000006a47304402201feaf12908260f666ab369bl
}
```

See also

- GetRawTransaction: gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings.

GetTxOut

The `gettxout` RPC returns details about a transaction output. Only unspent transaction outputs (UTXOs) are guaranteed to be available.

Parameter #1—the TXID of the output to get

Name	Type	Presence	Description
TXID	string (hex)	Required (exactly 1)	The TXID of the transaction containing the output to get, encoded as hex in RPC byte order

Parameter #2—the output index number (vout) of the output to get

Name	Type	Presence	Description
Vout	number (int)	Required (exactly 1)	The output index number (vout) of the output within the transaction; the first output in a transaction is vout 0

Parameter #3—whether to display unconfirmed outputs from the memory pool

Name	Type	Presence	Description
Unconfirmed	bool	Optional (0 or 1)	Set to `true` to display unconfirmed outputs from the memory pool; set to `false` (the default) to only display outputs from confirmed transactions

Result—a description of the output

Name	Type	Presence	Description
`result`	object/null	Required (exactly 1)	Information about the output. If output wasn't found or if an error occurred, this will be JSON `null`
→ `bestblock`	string (hex)	Required (exactly 1)	The hash of the header of the block on the local best block chain which includes this transaction. The hash will encoded as hex in RPC byte order. If the transaction is not part of a block, the string will be empty
→ `confirmations`	number (int)	Required (exactly 1)	The number of confirmations received for the transaction containing this output or `0` if the transaction hasn't been confirmed yet
→ `value`	number (bitcoins)	Required (exactly 1)	The amount of bitcoins spent to this output. May be `0`
→ `scriptPubKey`	string : object	Optional (0 or 1)	An object with information about the pubkey script. This may be `null` if there was no pubkey script
→ → `asm`	string	Required (exactly 1)	The pubkey script in decoded form with non-data-pushing op codes listed
→ → `hex`	string (hex)	Required (exactly 1)	The pubkey script encoded as hex
→ → `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for `nulldata` or `nonstandard` script types (see the `type` key below)
→ → `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script • `pubkeyhash` for a P2PKH script • `scripthash` for a P2SH script • `multisig` for a bare multisig script • `nulldata` for nulldata scripts • `nonstandard` for unknown scripts
→ → `addresses`	string : array	Optional (0 or 1)	The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for `nulldata` or `nonstandard` script types
→ → → Address	string	Required (1 or more)	A P2PKH or P2SH address
→ `version`	number (int)	Required (exactly 1)	The transaction version number of the transaction containing the pubkey script
→		Required	Set to `true` if the transaction output belonged to a coinbase transaction; set to

`coinbase`	bool	(exactly 1)	`false` for all other transactions. Coinbase transactions need to have 101 confirmations before their outputs can be spent

Example from Bitcoin Core 0.10.0

Get the UTXO from the following transaction from the first output index (“0”), searching the memory pool if necessary.

```
bitcoin-cli -testnet gettxout \
  d77aee99e8bdc11f40b8a9354956f0346fec5535b82c77c8b5c06047e3bca86a \
  0 true
```

Result:

```
{
    "bestblock" : "00000000c92356f7030b1deeab54b3b02885711320b4c48523be9daa3e0ace5d",
    "confirmations" : 0,
    "value" : 0.00100000,
    "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 a11418d3c144876258ba02909514d90e71ad8443 OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a914a11418d3c144876258ba02909514d90e71ad844388ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
            "mvCfAJSKaoFXoJEvv8ssW7wxaqRPphQuSv"
        ]
    },
    "version" : 1,
    "coinbase" : false
}
```

See also

- GetRawTransaction: gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings.
- GetTransaction: gets detailed information about an in-wallet transaction.

GetTxOutSetInfo

The `gettxoutsetinfo` RPC returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.

Parameters: none

Result—statistics about the UTXO set

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Information about the UTXO set
→ `height`	number (int)	Required (exactly 1)	The height of the local best block chain. A new node with only the hardcoded genesis block will have a height of 0
→ `bestblock`	string (hex)	Required (exactly 1)	The hash of the header of the highest block on the local best block chain, encoded as hex in RPC byte order
→ `transactions`	number (int)	Required (exactly 1)	The number of transactions with unspent outputs

→ `txouts`	number (int)	Required (exactly 1)	The number of unspent transaction outputs
→ `bytes_serialized`	number (int)	Required (exactly 1)	The size of the serialized UTXO set in bytes; not counting overhead, this is the size of the `chainstate` directory in the Bitcoin Core configuration directory
→ `hash_serialized`	string (hex)	Required (exactly 1)	A SHA256(SHA256()) hash of the serialized UTXO set; useful for comparing two nodes to see if they have the same set (they should, if they always used the same serialization format and currently have the same best block). The hash is encoded as hex in RPC byte order
→ `total_amount`	number (bitcoins)	Required (exactly 1)	The total number of bitcoins in the UTXO set

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet gettxoutsetinfo
```

Result:

```
{
    "height" : 315293,
    "bestblock" : "00000000c92356f7030b1deeab54b3b02885711320b4c48523be9daa3e0ace5d",
    "transactions" : 771920,
    "txouts" : 2734587,
    "bytes_serialized" : 102629817,
    "hash_serialized" : "4753470fda0145760109e79b8c218a1331e84bb4269d116857b8a4597f109905",
    "total_amount" : 13131746.33839451
}
```

See also

- GetBlockChainInfo: provides information about the current state of the block chain.
- GetMemPoolInfo: returns information about the node’s current transaction memory pool.

GetUnconfirmedBalance

Requires wallet support.

The `getunconfirmedbalance` RPC returns the wallet’s total unconfirmed balance.

Parameters: none

Result—the balance of unconfirmed transactions paying this wallet

Name	Type	Presence	Description
`result`	number (bitcoins)	Required (exactly 1)	The total number of bitcoins paid to this wallet in unconfirmed transactions

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getunconfirmedbalance
```

Result (no unconfirmed incoming payments):

```
0.00000000
```

See also

- GetBalance: gets the balance in decimal bitcoins across all accounts or for a particular account.

GetWalletInfo

Requires wallet support. Added in Bitcoin Core 0.9.2.

The `getwalletinfo` RPC provides information about the wallet.

Parameters: none

Result—information about the wallet

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the wallet
→ `walletversion`	number (int)	Required (exactly 1)	The version number of the wallet
→ `balance`	number (bitcoins)	Required (exactly 1)	The balance of the wallet. The same as returned by the `getbalance` RPC with default parameters
→ `txcount`	number (int)	Required (exactly 1)	The total number of transactions in the wallet (both spends and receives)
→ `keypoololdest`	number (int)	Required (exactly 1)	The date as Unix epoch time when the oldest key in the wallet key pool was created; useful for only scanning blocks created since this date for transactions
→ `keypoolsize`	number (int)	Required (exactly 1)	The number of keys in the wallet keypool
→ `unlocked_until`	number (int)	Optional (0 or 1)	Only returned if the wallet was encrypted with the `encryptwallet` RPC. A Unix epoch date when the wallet will be locked, or `0` if the wallet is currently locked

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet getwalletinfo
```

Result:

```
{
    "walletversion" : 60000,
    "balance" : 1.45060000,
    "txcount" : 17,
    "keypoololdest" : 1398809500,
    "keypoolsize" : 196,
    "unlocked_until" : 0
}
```

See also

- ListTransactions: returns the most recent transactions that affect the wallet.

GetWork

The `getwork` RPC was removed in Bitcoin Core 0.10.0.. If you have an older version of Bitcoin Core, use the `help` RPC to get help. For example: `help getwork`

See also

- GetBlockTemplate: gets a block template or proposal for use with mining software.
- SubmitBlock: accepts a block, verifies it is a valid addition to the block chain, and broadcasts it to the network. Extra parameters are ignored by Bitcoin Core but may be used by mining pools or other programs.
- SetGenerate: enables or disables hashing to attempt to find the next block.

Help

The `help` RPC lists all available public RPC commands, or gets help for the specified RPC. Commands which are unavailable will not be listed, such as wallet RPCs if wallet support is disabled.

Parameter—the name of the RPC to get help for

Name	Type	Presence	Description
RPC	string	Optional (0 or 1)	The name of the RPC to get help for. If omitted, Bitcoin Core 0.9x will display an alphabetical list of commands; Bitcoin Core 0.10.0 will display a categorized list of commands

Result—a list of RPCs or detailed help for a specific RPC

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	The help text for the specified RPC or the list of commands. The `bitcoin-cli` command will parse this text and format it as human-readable text

Example from Bitcoin Core 0.10.0

Command to get help about the `help` RPC:

```
bitcoin-cli -testnet help help
```

Result:

```
help ( "command" )

List all commands, or get help for a specified command.

Arguments:
1. "command"     (string, optional) The command to get help on

Result:
"text"     (string) The help text
```

See also

- The RPC Quick Reference

ImportAddress

Requires wallet support. Added in Bitcoin Core 0.10.0.

The `importaddress` RPC adds an address or pubkey script to the wallet without the associated private key, allowing you to watch for transactions affecting that address or pubkey script without being able to spend any of its outputs.

Parameter #1—the address or pubkey script to watch

Name	Type	Presence	Description
Address or Script	string (base58 or hex)	Required (exactly 1)	Either a P2PKH or P2SH address encoded in base58check, or a pubkey script encoded as hex

Parameter #2—The account into which to place the address or pubkey script

Name	Type	Presence	Description
Account	string	Optional (0 or 1)	An account name into which the address should be placed. Default is the default account, an empty string(“”)

Parameter #3—whether to rescan the block chain

Name	Type	Presence	Description
Rescan	bool	Optional (0 or 1)	Set to `true` (the default) to rescan the entire local block database for transactions affecting any address or pubkey script in the wallet (including transaction affecting the newly-added address or pubkey script). Set to `false` to not rescan the block database (rescanning can be performed at any time by restarting Bitcoin Core with the `-rescan` command-line argument). Rescanning may take several minutes. Notes: if the address or pubkey script is already in the wallet, the block database will not be rescanned even if this parameter is set

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	If the address or pubkey script is added to the wallet (or is already part of the wallet), JSON `null` will be returned

Example from Bitcoin Core 0.10.0

Add an address, rescanning the local block database for any transactions matching it.

```
bitcoin-cli -testnet importaddress \
  muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt "watch-only test" true
```

Result:

(No output; success.)

Show that the address has been added:

```
bitcoin-cli -testnet getaccount muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt
```

Result:

```
watch-only test
```

See also

- ImportPrivKey: adds a private key to your wallet. The key should be formatted in the wallet import format created by the `dumpprivkey` RPC.
- ListReceivedByAddress: lists the total number of bitcoins received by each address.

ImportPrivKey

Requires wallet support. Wallet must be unlocked.

The `importprivkey` RPC adds a private key to your wallet. The key should be formatted in the wallet import format created by the `dumpprivkey` RPC.

Parameter #1—the private key to import

Name	Type	Presence	Description
Private Key	string (base58)	Required (exactly 1)	The private key to import into the wallet encoded in base58check using wallet import format (WIF)

Parameter #2—the account into which the key should be placed

Name	Type	Presence	Description
Account	string	Optional (0 or 1)	The name of an account to which transactions involving the key should be assigned. The default is the default account, an empty string (“”)

Parameter #3—whether to rescan the block chain

Name	Type	Presence	Description
Rescan	bool	Optional (0 or 1)	Set to `true` (the default) to rescan the entire local block database for transactions affecting any address or pubkey script in the wallet (including transaction affecting the newly-added address for this private key). Set to `false` to not rescan the block database (rescanning can be performed at any time by restarting Bitcoin Core with the `-rescan` command-line argument). Rescanning may take several minutes. Notes: if the address for this key is already in the wallet, the block database will not be rescanned even if this parameter is set

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	If the private key is added to the wallet (or is already part of the wallet), JSON `null` will be returned

Example from Bitcoin Core 0.10.0

Import the private key for the address mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe, giving it a label and scanning the entire block chain:

```
bitcoin-cli -testnet importprivkey \
              cU8Q2jGeX3GNKNa5etiC8mgEgFSeVUTRQfWE2ZCzszyqYNK4Mepy \
              "test label" \
              true
```

(Success: no result displayed.)

See also

- DumpPrivKey: returns the wallet-import-format (WIP) private key corresponding to an address. (But does not remove it from the wallet.)
- ImportAddress: adds an address or pubkey script to the wallet without the associated private key, allowing you to watch for transactions affecting that address or pubkey script without being able to spend any of its outputs.
- ImportWallet: imports private keys from a file in wallet dump file format (see the `dumpwallet` RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.

ImportWallet

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `importwallet` RPC imports private keys from a file in wallet dump file format (see the `dumpwallet` RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.

Parameter #1—the file to import

Name	Type	Presence	Description
Filename	string	Required (exactly 1)	The file to import. The path is relative to Bitcoin Core's working directory

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	If all the keys in the file are added to the wallet (or are already part of the wallet), JSON `null` will be returned

Example from Bitcoin Core 0.10.0

Import the file shown in the example subsection of the `dumpwallet` RPC.

```
bitcoin-cli -testnet importwallet /tmp/dump.txt
```

(Success: no result displayed.)

See also

- DumpWallet: creates or overwrites a file with all wallet keys in a human-readable format.
- ImportPrivKey: adds a private key to your wallet. The key should be formatted in the wallet import format created by the `dumpprivkey` RPC.

KeyPoolRefill

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `keypoolrefill` RPC fills the cache of unused pre-generated keys (the keypool).

Parameter #1—the new keypool size

Name	Type	Presence	Description

Key Pool Size	number (int)	Optional (0 or 1)	The new size of the keypool; if the number of keys in the keypool is less than this number, new keys will be generated. Default is `100`. The value `0` also equals the default. The value specified is for this call only—the default keypool size is not changed

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	If the keypool is successfully filled, JSON `null` will be returned

Example from Bitcoin Core 0.10.0

Generate one extra key than the default:

```
bitcoin-cli -testnet keypoolrefill 101
```

(No result shown: success.)

See also

- GetNewAddress: returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.
- GetAccountAddress: returns the current Bitcoin address for receiving payments to this account. If the account doesn't exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address.
- GetWalletInfo: provides information about the wallet.

ListAccounts

Requires wallet support.

The `listaccounts` RPC lists accounts and their balances.

Parameter #1—the minimum number of confirmations a transaction must have

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Parameter #2—whether to include watch-only addresses in results

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—a list of accounts and their balances

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	A JSON array containing key/value pairs with account names and values. Must include, at the very least, the default account (“”)
→ Account : Balance	string : number (bitcoins)	Required (1 or more)	The name of an account as a string paired with the balance of the account as a number of bitcoins. The number of bitcoins may be negative if the account has spent more bitcoins than it received. Accounts with zero balances and zero transactions will be displayed

Example from Bitcoin Core 0.10.0

Display account balances with one confirmation and watch-only addresses included.

```
bitcoin-cli -testnet listaccounts 1 true
```

Result:

```
{
    "" : -2.73928803,
    "Refund from example.com" : 0.00000000,
    "doc test" : -498.45900000,
    "someone else's address" : 0.00000000,
    "someone else's address2" : 0.00050000,
    "test" : 499.97975293,
    "test account" : 0.00000000,
    "test label" : 0.48961280,
    "test1" : 1.99900000
}
```

See also

- GetAccount: returns the name of the account associated with the given address.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.
- ListReceivedByAccount: lists the total number of bitcoins received by each account.

ListAddressGroupings

Requires wallet support.

The `listaddressgroupings` RPC lists groups of addresses that may have had their common ownership made public by common use as inputs in the same transaction or from being used as change from a previous transaction.

Parameters: none

Result—an array of arrays describing the groupings

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing the groupings. May be empty
→ Groupings	array	Optional (0 or more)	An array containing arrays of addresses which can be associated with each other
→ →			

Address Details	array	Required (1 or more)	An array containing information about a particular address
→ → → Address	string (base58)	Required (exactly 1)	The address in base58check format
→ → → Balance	number (bitcoins)	Required (exactly 1)	The current spendable balance of the address, not counting unconfirmed transactions
→ → → Account	string	Optional (0 or 1)	The account the address belongs to, if any. This field will not be returned for change addresses. The default account is an empty string (“”)

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet listaddressgroupings
```

Result (edited to only the first two results):

```
[
    [
        [
            "mgKgzJ7HR64CrB3zm1B4FUUCLtaSqUKfDb",
            0.00000000
        ],
        [
            "mnUbTmdAFD5EAg3348Ejmonub7JcWtrMck",
            0.00000000,
            "test1"
        ]
    ]
]
```

See also

- GetAddressesByAccount: returns a list of every address assigned to a particular account.
- GetTransaction: gets detailed information about an in-wallet transaction.

ListLockUnspent

Requires wallet support.

The `listlockunspent` RPC returns a list of temporarily unspendable (locked) outputs.

Parameters: none

Result—an array of locked outputs

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing all locked outputs. May be empty
→ Output	object	Optional (1 or more)	An object describing a particular locked output
→ → `txid`	string (hex)	Required (exactly 1)	The TXID of the transaction containing the locked output, encoded as hex in RPC byte order

→ → `vout`	number (int)	Required (exactly 1)	The output index number (vout) of the locked output within the transaction. Output index `0` is the first output within the transaction

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet listlockunspent
```

Result:

```
[
    {
        "txid" : "ca7cb6a5ffcc2f21036879493db4530c0ce9b5bff9648f9a3be46e2dfc8e0166",
        "vout" : 0
    }
]
```

See also

- LockUnspent: temporarily locks or unlocks specified transaction outputs. A locked transaction output will not be chosen by automatic coin selection when spending bitcoins. Locks are stored in memory only, so nodes start with zero locked outputs and the locked output list is always cleared when a node stops or fails.

ListReceivedByAccount

Requires wallet support.

The `listreceivedbyaccount` RPC lists the total number of bitcoins received by each account.

Parameter #1—the minimum number of confirmations a transaction must have to be counted

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Parameter #2—whether to include empty accounts

Name	Type	Presence	Description
Include Empty	bool	Optional (0 or 1)	Set to `true` to display accounts which have never received a payment. Set to `false` (the default) to only include accounts which have received a payment. Any account which has received a payment will be displayed even if its current balance is `0`

Parameter #3—whether to include watch-only addresses in results

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—account names, balances, and minimum confirmations

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing objects each describing an account. At the very least, the default account (“”) will be included
→ Account	object	Required (1 or more)	An object describing an account
→ → `involvesWatchonly`	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if the balance of this account includes a watch-only address which has received a spendable payment (that is, a payment with at least the specified number of confirmations and which is not an immature coinbase). Otherwise not returned
→ → `account`	string	Required (exactly 1)	The name of the account
→ → `amount`	number (bitcoins)	Required (exactly 1)	The total amount received by this account in bitcoins
→ → `confirmations`	number (int)	Required (exactly 1)	The number of confirmations received by the last transaction received by this account. May be `0`

Example from Bitcoin Core 0.10.0

Get the balances for all non-empty accounts, including only transactions which have been confirmed at least six times:

```
bitcoin-cli -testnet listreceivedbyaccount 6 false
```

Result (edited to only show the first two results):

```
[
    {
        "account" : "",
        "amount" : 0.19960000,
        "confirmations" : 53601
    },
    {
        "account" : "doc test",
        "amount" : 0.30000000,
        "confirmations" : 8991
    }
]
```

See also

- ListReceivedByAddress: lists the total number of bitcoins received by each address.
- GetReceivedByAccount: returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetReceivedByAddress: returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.

ListReceivedByAddress

Requires wallet support.

The `listreceivedbyaddress` RPC lists the total number of bitcoins received by each address.

Parameter #1—the minimum number of confirmations a transaction must have to be counted

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an externally-generated transaction must have before it is counted towards the balance. Transactions generated by this node are counted immediately. Typically, externally-generated transactions are payments to this wallet and transactions generated by this node are payments to other wallets. Use `0` to count unconfirmed transactions. Default is `1`

Parameter #2—whether to include empty accounts

Name	Type	Presence	Description
Include Empty	bool	Optional (0 or 1)	Set to `true` to display accounts which have never received a payment. Set to `false` (the default) to only include accounts which have received a payment. Any account which has received a payment will be displayed even if its current balance is `0`

Parameter #3—whether to include watch-only addresses in results

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—addresses, account names, balances, and minimum confirmations

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing objects each describing a particular address
→ Address	object	Optional (0 or more)	An object describing an address
→ → `involvesWatchonly`	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if this address is a watch-only address which has received a spendable payment (that is, a payment with at least the specified number of confirmations and which is not an immature coinbase). Otherwise not returned
→ → `address`	string (base58)	Required (exactly 1)	The address being described encoded in base58check
→ → `account`	string	Required (exactly 1)	The account the address belongs to; may be the default account, an empty string ("")
→ → `amount`	number (bitcoins)	Required (exactly 1)	The total amount the address has received in bitcoins

→ → `confirmations`	number (int)	Required (exactly 1)	The number of confirmations of the latest transaction to the address. May be `0` for unconfirmed
→ → TXIDs	array	Required (exactly 1)	An array of TXIDs belonging to transactions that pay the address
→ → → TXID	string	Optional (0 or more)	The TXID of a transaction paying the address, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

List addresses with balances confirmed by at least six blocks, including watch-only addresses:

```
bitcoin-cli -testnet listreceivedbyaddress 6 false true
```

Result (edit to show only two entries):

```
[
    {
        "address" : "mnUbTmdAFD5EAg3348Ejmonub7JcWtrMck",
        "account" : "test1",
        "amount" : 1.99900000,
        "confirmations" : 55680,
        "txids" : [
            "4d71a6127796766c39270881c779b6e05183f2bf35589261e9572436356f287f",
            "997115d0cf7b83ed332e6c1f2e8c44f803c95ea43490c84ce3e9ede4b2e1605f"
        ]
    },
    {
        "involvesWatchonly" : true,
        "address" : "n3GNqMveyvaPvUbH469vDRadqpJMPc84JA",
        "account" : "someone else's address2",
        "amount" : 0.00050000,
        "confirmations" : 34714,
        "txids" : [
            "99845fd840ad2cc4d6f93fafb8b072d188821f55d9298772415175c456f3077d"
        ]
    }
]
```

See also

- ListReceivedByAccount: lists the total number of bitcoins received by each account.
- GetReceivedByAddress: returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.
- GetReceivedByAccount: returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions.

ListSinceBlock

Requires wallet support.

The `listsinceblock` RPC gets all transactions affecting the wallet which have occurred since a particular block, plus the header hash of a block at a particular depth.

Parameter #1—a block header hash

Name	Type	Presence	Description
			The hash of a block header encoded as hex in RPC byte order. All transactions affecting the wallet

Header Hash	string (hex)	Optional (0 or 1)	which are not in that block or any earlier block will be returned, including unconfirmed transactions. Default is the hash of the genesis block, so all transactions affecting the wallet are returned by default

Parameter #2—the target confirmations for the lastblock field

Name	Type	Presence	Description
Target Confirmations	number (int)	Optional (0 or 1)	Sets the lastblock field of the results to the header hash of a block with this many confirmations. This does not affect which transactions are returned. Default is `1`, so the hash of the most recent block on the local best block chain is returned

Parameter #3—whether to include watch-only addresses in details and calculations

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object containing an array of transactions and the lastblock field
→ `transactions`	array	Required (exactly 1)	An array of objects each describing a particular **payment** to or from this wallet. The objects in this array do not describe an actual transactions, so more than one object in this array may come from the same transaction. This array may be empty
→ → Payment	object	Optional (0 or more)	An payment which did not appear in the specified block or an earlier block
→ → → `involvesWatchonly`	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if the payment involves a watch-only address. Otherwise not returned
→ → → `account`	string	Required (exactly 1)	The account which the payment was credited to or debited from. May be an empty string ("") for the default account
→ → → `address`	string (base58)	Optional (0 or 1)	The address paid in this payment, which may be someone else's address not belonging to this wallet. May be empty if the address is unknown, such as when paying to a non-standard pubkey script
→ → → `category`	string	Required (exactly 1)	Set to one of the following values: • `send` if sending payment • `receive` if this wallet received payment in a regular transaction • `generate` if a matured and spendable coinbase • `immature` if a coinbase that is not spendable yet • `orphan` if a coinbase from a block that's not in the local best block chain

→ → → `amount`	number (bitcoins)	Required (exactly 1)	A negative bitcoin amount if sending payment; a positive bitcoin amount if receiving payment (including coinbases)
→ → → `vout`	number (int)	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* For an output, the output index (vout) for this output in this transaction. For an input, the output index for the output being spent in its transaction. Because inputs list the output indexes from previous transactions, more than one entry in the details array may have the same output index
→ → → `fee`	number (bitcoins)	Optional (0 or 1)	If sending payment, the fee paid as a negative bitcoins value. May be `0`. Not returned if receiving payment
→ → → `confirmations`	number (int)	Required (exactly 1)	The number of confirmations the transaction has received. Will be `0` for unconfirmed and `-1` for conflicted
→ → → `generated`	bool	Optional (0 or 1)	Set to `true` if the transaction is a coinbase. Not returned for regular transactions
→ → → `blockhash`	string (hex)	Optional (0 or 1)	Only returned for confirmed transactions. The hash of the block on the local best block chain which includes this transaction, encoded as hex in RPC byte order
→ → → `blockindex`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block height of the block on the local best block chain which includes this transaction
→ → → `blocktime`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block header time (Unix epoch time) of the block on the local best block chain which includes this transaction
→ → → `txid`	string (hex)	Required (exactly 1)	The TXID of the transaction, encoded as hex in RPC byte order
→ → → `walletconflicts`	array	Required (exactly 1)	An array containing the TXIDs of other transactions that spend the same inputs (UTXOs) as this transaction. Array may be empty
→ → → → TXID	string (hex)	Optional (0 or more)	The TXID of a conflicting transaction, encoded as hex in RPC byte order
→ → → `time`	number (int)	Required (exactly 1)	A Unix epoch time when the transaction was added to the wallet
→ → → `timerecived`	number (int)	Required (exactly 1)	A Unix epoch time when the transaction was detected by the local node, or the time of the block on the local best block chain that included the transaction
→ → → `comment`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction. Only returned if a comment was added
→ → → `to`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction identifying who the transaction was sent to. Only returned if a comment-to was added
→ `lastblock`	string (hex)	Required (exactly 1)	The header hash of the block with the number of confirmations specified in the *target confirmations* parameter, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

Get all transactions since a particular block (including watch-only transactions) and the header hash of the sixth most recent block.

```
bitcoin-cli -testnet listsinceblock \
              00000000688633a503f69818a70eac281302e9189b1bb57a76a05c329fcda718 \
              6 true
```

Result (edited to show only two payments):

```
{
    "transactions" : [
        {
            "account" : "doc test",
            "address" : "mmXgiR6KAhZCyQ8ndr2BCfEq1wNG2UnyG6",
            "category" : "receive",
            "amount" : 0.10000000,
            "vout" : 0,
            "confirmations" : 76478,
            "blockhash" : "000000000017c84015f254498c62a7c884a51ccd75d4dd6dbdcb6434aa3bd44d",
            "blockindex" : 1,
            "blocktime" : 1399294967,
            "txid" : "85a98fdf1529f7d5156483ad020a51b7f3340e47448cf932f470b72ff01a6821",
            "walletconflicts" : [
            ],
            "time" : 1399294967,
            "timereceived" : 1418924714
        },
        {
            "involvesWatchonly" : true,
            "account" : "someone else's address2",
            "address" : "n3GNqMveyvaPvUbH469vDRadqpJMPc84JA",
            "category" : "receive",
            "amount" : 0.00050000,
            "vout" : 0,
            "confirmations" : 34714,
            "blockhash" : "00000000bd0ed80435fc9fe3269da69bb0730ebb454d0a29128a870ea1a37929",
            "blockindex" : 11,
            "blocktime" : 1411051649,
            "txid" : "99845fd840ad2cc4d6f93fafb8b072d188821f55d9298772415175c456f3077d",
            "walletconflicts" : [
            ],
            "time" : 1418695703,
            "timereceived" : 1418925580
        }
    ],
    "lastblock" : "0000000000984add1a686d513e66d25686572c7276ec3e358a7e3e9f7eb88619"
}
```

See also

- ListReceivedByAccount: lists the total number of bitcoins received by each account.
- ListReceivedByAddress: lists the total number of bitcoins received by each address.

ListTransactions

Requires wallet support.

The `listtransactions` RPC returns the most recent transactions that affect the wallet.

Parameter #1—an account name to get transactions from

Name	Type	Presence	Description
Account	string	Optional (0 or 1)	The name of an account to get transactinos from. Use an empty string ("") to get transactions for the default account. Default is `*` to get transactions for all accounts

Parameter #2—the number of transactions to get

Name	Type	Presence	Description
Count	number (int)	Optional (0 or 1)	The number of the most recent transactions to list. Default is `10`

Parameter #3—the number of transactions to skip

Name	Type	Presence	Description
Skip	number (int)	Optional (0 or 1)	The number of the most recent transactions which should not be returned. Allows for pagination of results. Default is `0`

Parameter #4—whether to include watch-only addresses in details and calculations

Name	Type	Presence	Description
Include Watch-Only	bool	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* If set to `true`, include watch-only addresses in details and calculations as if they were regular addresses belonging to the wallet. If set to `false` (the default), treat watch-only addresses as if they didn't belong to this wallet

Result—payment details

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array containing objects, with each object describing a **payment** or internal accounting entry (not a transaction). More than one object in this array may come from a single transaction. Array may be empty
→ Payment	object	Optional (0 or more)	A payment or internal accounting entry
→ → `account`	string	Required (exactly 1)	The account which the payment was credited to or debited from. May be an empty string ("") for the default account
→ → `address`	string (base58)	Optional (0 or 1)	The address paid in this payment, which may be someone else's address not belonging to this wallet. May be empty if the address is unknown, such as when paying to a non-standard pubkey script or if this is in the *move* category
→ → `category`	string	Required (exactly 1)	Set to one of the following values: • `send` if sending payment • `receive` if this wallet received payment in a regular transaction • `generate` if a matured and spendable coinbase • `immature` if a coinbase that is not spendable yet • `orphan` if a coinbase from a block that's not in the local best block chain • `move` if an off-block-chain move made with the `move` RPC
→ → `amount`	number (bitcoins)	Required (exactly 1)	A negative bitcoin amount if sending payment; a positive bitcoin amount if receiving payment (including coinbases)
→ → `vout`	number (int)	Optional (0 or 1)	*Added in Bitcoin Core 0.10.0* For an output, the output index (vout) for this output in this transaction. For an input, the output index for the output being spent in its transaction. Because inputs list the output indexes from previous transactions, more than one entry in the details array

			may have the same output index. Not returned for *move* category payments
→ → `fee`	number (bitcoins)	Optional (0 or 1)	If sending payment, the fee paid as a negative bitcoins value. May be `0`. Not returned if receiving payment or for *move* category payments
→ → `confirmations`	number (int)	Optional (0 or 1)	The number of confirmations the transaction has received. Will be `0` for unconfirmed and `-1` for conflicted. Not returned for *move* category payments
→ → `generated`	bool	Optional (0 or 1)	Set to `true` if the transaction is a coinbase. Not returned for regular transactions or *move* category payments
→ → `blockhash`	string (hex)	Optional (0 or 1)	Only returned for confirmed transactions. The hash of the block on the local best block chain which includes this transaction, encoded as hex in RPC byte order
→ → `blockindex`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block height of the block on the local best block chain which includes this transaction
→ → `blocktime`	number (int)	Optional (0 or 1)	Only returned for confirmed transactions. The block header time (Unix epoch time) of the block on the local best block chain which includes this transaction
→ → `txid`	string (hex)	Optional (0 or 1)	The TXID of the transaction, encoded as hex in RPC byte order. Not returned for *move* category payments
→ → `walletconflicts`	array	Optional (0 or 1)	An array containing the TXIDs of other transactions that spend the same inputs (UTXOs) as this transaction. Array may be empty. Not returned for *move* category payments
→ → → TXID	string (hex)	Optional (0 or more)	The TXID of a conflicting transaction, encoded as hex in RPC byte order
→ → `time`	number (int)	Required (exactly 1)	A Unix epoch time when the transaction was added to the wallet
→ → `timereceived`	number (int)	Optional (0 or 1)	A Unix epoch time when the transaction was detected by the local node, or the time of the block on the local best block chain that included the transaction. Not returned for *move* category payments
→ → `comment`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction. Only returned in regular payments if a comment was added. Always returned in *move* category payments. May be an empty string
→ → `to`	string	Optional (0 or 1)	For transaction originating with this wallet, a locally-stored comment added to the transaction identifying who the transaction was sent to. Only returned if a comment-to was added. Never returned by *move* category payments. May be an empty string
→ → `otheraccount`	string	Optional (0 or 1)	Only returned by *move* category payments. This is the account the bitcoins were moved from or moved to, as indicated by a negative or positive *amount* field in this payment

Example from Bitcoin Core 0.10.0

List the most recent transaction from the account “someone else’s address2” including watch-only addresses.

```
bitcoin-cli -testnet listtransactions \
  "someone else's address2" 1 0 true
```

Result:

```
[
    {
        "involvesWatchonly" : true,
        "account" : "someone else's address2",
        "address" : "n3GNqMveyvaPvUbH469vDRadqpJMPc84JA",
        "category" : "receive",
        "amount" : 0.00050000,
        "vout" : 0,
        "confirmations" : 34714,
        "blockhash" : "00000000bd0ed80435fc9fe3269da69bb0730ebb454d0a29128a870ea1a37929",
        "blockindex" : 11,
        "blocktime" : 1411051649,
        "txid" : "99845fd840ad2cc4d6f93fafb8b072d188821f55d9298772415175c456f3077d",
        "walletconflicts" : [
        ],
        "time" : 1418695703,
        "timereceived" : 1418925580
    }
]
```

See also

- GetTransaction: gets detailed information about an in-wallet transaction.
- ListSinceBlock: gets all transactions affecting the wallet which have occurred since a particular block, plus the header hash of a block at a particular depth.

ListUnspent

Requires wallet support.

The `listunspent` RPC returns an array of unspent transaction outputs belonging to this wallet. **Note:** as of Bitcoin Core 0.10.0, outputs affecting watch-only addresses will be returned; see the *spendable* field in the results described below.

Parameter #1—the minimum number of confirmations an output must have

Name	Type	Presence	Description
Minimum Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations the transaction containing an output must have in order to be returned. Use `0` to return outputs from unconfirmed transactions. Default is `1`

Parameter #2—the maximum number of confirmations an output may have

Name	Type	Presence	Description
Maximum Confirmations	number (int)	Optional (0 or 1)	The maximum number of confirmations the transaction containing an output may have in order to be returned. Default is `9999999` (~10 million)

Parameter #3—the addresses an output must pay

Name	Type	Presence	Description
Addresses	array	Optional (0 or 1)	If present, only outputs which pay an address in this array will be returned
→ Address	string (base58)	Required (1 or more)	A P2PKH or P2SH address

Result—the list of unspent outputs

Name	Type	Presence	Description
`result`	array	Required (exactly 1)	An array of objects each describing an unspent output. May be empty
→ Unspent Output	object	Optional (0 or more)	An object describing a particular unspent output belonging to this wallet
→ → `txid`	string (hex)	Required (exactly 1)	The TXID of the transaction containing the output, encoded as hex in RPC byte order
→ → `vout`	number (int)	Required (exactly 1)	The output index number (vout) of the output within its containing transaction
→ → `address`	string (base58)	Optional (0 or 1)	The P2PKH or P2SH address the output paid. Only returned for P2PKH or P2SH output scripts
→ → `account`	string	Optional (0 or 1)	If the address returned belongs to an account, this is the account. Otherwise not returned
→ → `scriptPubKey`	string (hex)	Required (exactly 1)	The output script paid, encoded as hex
→ → `redeemScript`	string (hex)	Optional (0 or 1)	If the output is a P2SH whose script belongs to this wallet, this is the redeem script
→ → `amount`	number (int)	Required (exactly 1)	The amount paid to the output in bitcoins
→ → `confirmations`	number (int)	Required (exactly 1)	The number of confirmations received for the transaction containing this output
→ → `spendable`	bool	Required (exactly 1)	*Added in Bitcoin Core 0.10.0* Set to `true` if the private key or keys needed to spend this output are part of the wallet. Set to `false` if not (such as for watch-only addresses)

Example from Bitcoin Core 0.10.0

Get all outputs confirmed at least 6 times for a particular address:

```
bitcoin-cli -testnet listunspent 6 99999999 '''
  [
    "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe"
  ]
'''
```

Result:

```
[
    {
        "txid" : "d54994ece1d11b19785c7248868696250ab195605b469632b7bd68130e880c9a",
        "vout" : 1,
        "address" : "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe",
        "account" : "test label",
        "scriptPubKey" : "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
        "amount" : 0.00010000,
        "confirmations" : 6210,
        "spendable" : true
    }
]
```

See also

- ListTransactions: returns the most recent transactions that affect the wallet.
- LockUnspent: temporarily locks or unlocks specified transaction outputs. A locked transaction output will not be chosen by automatic coin selection when spending bitcoins. Locks are stored in memory only, so nodes start with zero locked outputs and the locked output list is always cleared when a node stops or fails.

LockUnspent

Requires wallet support.

The `lockunspent` RPC temporarily locks or unlocks specified transaction outputs. A locked transaction output will not be chosen by automatic coin selection when spending bitcoins. Locks are stored in memory only, so nodes start with zero locked outputs and the locked output list is always cleared when a node stops or fails.

Parameter #1—whether to lock or unlock the outputs

Name	Type	Presence	Description
Lock Or Unlock	bool	Required (exactly 1)	Set to `true` to lock the outputs specified in the following parameter. Set to `false` to unlock the outputs specified. If this is the only argument specified, all outputs will be unlocked (even if this is set to `false`)

Parameter #2—the outputs to lock or unlock

Name	Type	Presence	Description
Outputs	array	Optional (0 or 1)	An array of outputs to lock or unlock
→ Output	object	Required (1 or more)	An object describing a particular output
→ → `txid`	string	Required (exactly 1)	The TXID of the transaction containing the output to lock or unlock, encoded as hex in internal byte order
→ → `vout`	number (int)	Required (exactly 1)	The output index number (vout) of the output to lock or unlock. The first output in a transaction has an index of `0`

Result—`true` if successful

Name	Type	Presence	Description
`result`	bool	Required (exactly 1)	Set to `true` if the outputs were successfully locked or unlocked

Example from Bitcoin Core 0.10.0

Lock two outputs:

```
bitcoin-cli -testnet lockunspent true '''
```

```
  [
    {
      "txid": "5a7d24cd665108c66b2d56146f244932edae4e2376b561b3d396d5ae017b9589",
      "vout": 0
    },
    {
      "txid": "6c5edd41a33f9839257358ba6ddece67df9db7f09c0db6bbea00d0372e8fe5cd",
      "vout": 0
    }
  ]
'''
```

Result:

```
true
```

Unlock one of the above outputs:

```
bitcoin-cli -testnet lockunspent false '''
[
  {
    "txid": "5a7d24cd665108c66b2d56146f244932edae4e2376b561b3d396d5ae017b9589",
    "vout": 0
  }
]
'''
```

Result:

```
true
```

See also

- ListLockUnspent: returns a list of temporarily unspendable (locked) outputs.
- ListUnspent: returns an array of unspent transaction outputs belonging to this wallet.

Move

Requires wallet support.

The `move` RPC moves a specified amount from one account in your wallet to another using an off-block-chain transaction.

⚠ **Warning:** it's allowed to move more funds than are in an account, giving the sending account a negative balance and giving the receiving account a balance that may exceed the number of bitcoins in the wallet (or the number of bitcoins in existence).

Parameter #1—from account

Name	Type	Presence	Description
From Account	string	Required (exactly 1)	The name of the account to move the funds from

Parameter #2—to account

Name	Type	Presence	Description
To Account	string	Required (exactly 1)	The name of the account to move the funds to

Parameter #3—amount to move

Name	Type	Presence	Description
Amount	number (bitcoins)	Required (exactly 1)	The amount of bitcoins to move

Parameter #4—an unused parameter

Name	Type	Presence	Description
Unused	number (int)	Optional (0 or 1)	This parameter is no longer used. If parameter #5 needs to be specified, this can be any integer

Parameter #5—a comment

Name	Type	Presence	Description
Comment	string	Optional (0 or 1)	A comment to assign to this move payment

Result—`true` on success

Name	Type	Presence	Description
`result`	bool	Required (exactly 1)	Set to `true` if the move was successful

Example from Bitcoin Core 0.10.0

Move 0.1 bitcoins from “doc test” to “test1”, giving the transaction the comment “Example move”:

```
bitcoin-cli -testnet move "doc test" "test1" 0.1 0 "Example move"
```

Result:

```
true
```

See also

- ListAccounts: lists accounts and their balances.
- SendFrom: spends an amount from a local account to a bitcoin address.
- SendToAddress: spends an amount to a given address.

Ping

The `ping` RPC sends a P2P ping message to all connected nodes to measure ping time. Results are provided by the `getpeerinfo` RPC pingtime and pingwait fields as decimal seconds. The P2P `ping` message is handled in a queue with all other commands, so it measures processing backlog, not just network ping.

Parameters: none

Result—`null`

Name	Type	Presence	Description
`result`	null	Required	Always JSON `null`

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet ping
```

(Success: no result printed.)

Get the results using the `getpeerinfo` RPC:

```
bitcoin-cli -testnet getpeerinfo | grep ping
```

Results:

```
        "pingtime" : 0.11790800,
        "pingtime" : 0.22673400,
        "pingtime" : 0.16451900,
        "pingtime" : 0.12465200,
        "pingtime" : 0.13267900,
        "pingtime" : 0.23983300,
        "pingtime" : 0.16764700,
        "pingtime" : 0.11337300,
```

See also

- GetPeerInfo: returns data about each connected network node.
- P2P Ping Message

PrioritiseTransaction

Added in Bitcoin Core 0.10.0.

The `prioritisetransaction` RPC adds virtual priority or fee to a transaction, allowing it to be accepted into blocks mined by this node (or miners which use this node) with a lower priority or fee. (It can also remove virtual priority or fee, requiring the transaction have a higher priority or fee to be accepted into a locally-mined block.)

Parameter #1—the TXID of the transaction to modify

Name	Type	Presence	Description
TXID	string	Required (exactly 1)	The TXID of the transaction whose virtual priority or fee you want to modify, encoded as hex in RPC byte order

Parameter #2—the change to make to the virtual priority

Name	Type	Presence	Description
Priority	number (real)	Required (exactly 1)	If positive, the priority to add to the transaction in addition to its computed priority; if negative, the priority to subtract from the transaction's computed priory. Computed priority is the age of each input in days since it was added to the block chain as an output (coinage) times the value of the input in satoshis (value) divided by the size of the serialized transaction (size), which is `coinage * value / size`

Parameter #3—the change to make to the virtual fee

Name	Type	Presence	Description
Fee	number (int)	Required (exactly 1)	**Warning:** this value is in satoshis, not bitcoins If positive, the virtual fee to add to the actual fee paid by the transaction; if negative, the virtual fee to subtract from the actual fee paid by the transaction. No change is made to the actual fee paid by the transaction

Result—`true` if the priority is changed

Name	Type	Presence	Description
`result`	bool (true only)	Required (exactly 1)	Always set to `true` if all three parameters are provided. Will not return an error if the TXID is not in the memory pool. If fewer or more than three arguments are provided, or if something goes wrong, will be set to `null`

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet prioritisetransaction \
    fe0165147da737e16f5096ab6c1709825217377a95a882023ed089a89af4cff9 \
    1234 456789
```

Result:

```
true
```

See also

- GetRawMemPool: returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object.
- GetBlockTemplate: gets a block template or proposal for use with mining software.

SendFrom

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `sendfrom` RPC spends an amount from a local account to a bitcoin address.

Parameter #1—from account

Name	Type	Presence	Description
From Account	string	Required (exactly 1)	The name of the account from which the bitcoins should be spent. Use an empty string (“”) for the default account

Parameter #2—to address

Name	Type	Presence	Description
To Address	string	Required (exactly 1)	A P2PKH or P2SH address to which the bitcoins should be sent

Parameter #3—amount to spend

Name	Type	Presence	Description
Amount	number (bitcoins)	Required (exactly 1)	The amount to spend in bitcoins. Bitcoin Core will ensure the account has sufficient bitcoins to pay this amount (but the transaction fee paid is not included in the calculation, so an account can spend a total of its balance plus the transaction fee)

Parameter #4—minimum confirmations

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an incoming transaction must have for its outputs to be credited to this account's balance. Outgoing transactions are always counted, as are move transactions made with the `move` RPC. If an account doesn't have a balance high enough to pay for this transaction, the payment will be rejected. Use `0` to spend unconfirmed incoming payments. Default is `1`

⚠ **Warning:** if account1 receives an unconfirmed payment and transfers it to account2 with the `move` RPC, account2 will be able to spend those bitcoins even if this parameter is set to `1` or higher.

Parameter #5—a comment

Name	Type	Presence	Description
Comment	string	Optional (0 or 1)	A locally-stored (not broadcast) comment assigned to this transaction. Default is no comment

Parameter #6—a comment about who the payment was sent to

Name	Type	Presence	Description
Comment To	string	Optional (0 or 1)	A locally-stored (not broadcast) comment assigned to this transaction. Meant to be used for describing who the payment was sent to. Default is no comment

Result—a TXID of the sent transaction

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	The TXID of the sent transaction, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

Spend 0.1 bitcoins from the account "test" to the address indicated below using only UTXOs with at least six confirmations, giving the transaction the comment "Example spend" and labeling the spender "Example.com":

```
bitcoin-cli -testnet sendfrom "test" \
            mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe \
            0.1 \
            6 \
            "Example spend" \
            "Example.com"
```

Result:

```
f14ee5368c339644d3037d929bbe1f1544a532f8826c7b7288cb994b0b0ff5d8
```

See also

- SendToAddress: spends an amount to a given address.
- SendMany: creates and broadcasts a transaction which sends outputs to multiple addresses.

SendMany

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `sendmany` RPC creates and broadcasts a transaction which sends outputs to multiple addresses.

Parameter #1—from account

Name	Type	Presence	Description
From Account	string	Required (exactly 1)	The name of the account from which the bitcoins should be spent. Use an empty string (“”) for the default account. Bitcoin Core will ensure the account has sufficient bitcoins to pay the total amount in the *outputs* field described below (but the transaction fee paid is not included in the calculation, so an account can spend a total of its balance plus the transaction fee)

Parameter #2—the addresses and amounts to pay

Name	Type	Presence	Description
Outputs	object	Required (exactly 1)	An object containing key/value pairs corresponding to the addresses and amounts to pay
→ Address/Amount	string (base58) : number (bitcoins)	Required (1 or more)	A key/value pair with a base58check-encoded string containing the P2PKH or P2SH address to pay as the key, and an amount of bitcoins to pay as the value

Parameter #3—minimum confirmations

Name	Type	Presence	Description
Confirmations	number (int)	Optional (0 or 1)	The minimum number of confirmations an incoming transaction must have for its outputs to be credited to this account’s balance. Outgoing transactions are always counted, as are move transactions made with the `move` RPC. If an account doesn’t have a balance high enough to pay for this transaction, the payment will be rejected. Use `0` to spend unconfirmed incoming payments. Default is `1`

⚠ **Warning:** if account1 receives an unconfirmed payment and transfers it to account2 with the `move` RPC, account2 will be able to spend those bitcoins even if this parameter is set to `1` or higher.

Parameter #4—a comment

Name	Type	Presence	Description
Comment	string	Optional (0 or 1)	A locally-stored (not broadcast) comment assigned to this transaction. Default is no comment

Result—a TXID of the sent transaction

Name	Type	Presence	Description

`result`	string	Required (exactly 1)	The TXID of the sent transaction, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

From the account *test1*, send 0.1 bitcoins to the first address and 0.2 bitcoins to the second address, with a comment of "Example Transaction".

```
bitcoin-cli -testnet sendmany \
  "test1" \
  '''
    {
      "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN": 0.1,
      "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe": 0.2
    } ''' \
  6       \
  "Example Transaction"
```

Result:

```
ec259ab74ddff199e61caa67a26e29b13b5688dc60f509ce0df4d044e8f4d63d
```

See also

- SendFrom: spends an amount from a local account to a bitcoin address.
- SendToAddress: spends an amount to a given address.
- Move: moves a specified amount from one account in your wallet to another using an off-block-chain transaction.

SendRawTransaction

The `sendrawtransaction` RPC validates a transaction and broadcasts it to the peer-to-peer network.

Parameter #1—a serialized transaction to broadcast

Name	Type	Presence	Description
Transaction	string (hex)	Required (exactly 1)	The serialized transaction to broadcast encoded as hex

*Parameter #2–whether to allow high fees**

Name	Type	Presence	Description
Allow High Fees	bool	Optional (0 or 1)	Set to `true` to allow the transaction to pay a high transaction fee. Set to `false` (the default) to prevent Bitcoin Core from broadcasting the transaction if it includes a high fee. Transaction fees are the sum of the inputs minus the sum of the outputs, so this high fees check helps ensures user including a change address to return most of the difference back to themselves

Result—a TXID or error message

Name	Type	Presence	Description
`result`	null/string (hex)	Required (exactly 1)	If the transaction was accepted by the node for broadcast, this will be the TXID of the transaction encoded as hex in RPC byte order. If the transaction was rejected by the node, this will set to `null`, the JSON-RPC error field will be set to a code, and the JSON-RPC message field may contain an informative error message

Examples from Bitcoin Core 0.10.0

Broadcast a signed transaction:

```
bitcoin-cli -testnet sendrawtransaction 01000000011da9283b4ddf8d\
89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a4730\
4402200ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a\
0485a20220172ecaf6975902584987d295b8dddf8f46ec32ca19122510e22405\
ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45c29a\
2fd0228c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc84198\
53b34d5e072ad37d1a5159f58488ac00000000
```

Result:

```
f5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad
```

See also

- CreateRawTransaction: creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
- DecodeRawTransaction: decodes a serialized transaction hex string into a JSON object describing the transaction.
- SignRawTransaction: signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.

SendToAddress

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `sendtoaddress` RPC spends an amount to a given address.

Parameter #1—to address

Name	Type	Presence	Description
To Address	string	Required (exactly 1)	A P2PKH or P2SH address to which the bitcoins should be sent

Parameter #2—amount to spend

Name	Type	Presence	Description
Amount	number (bitcoins)	Required (exactly 1)	The amount to spent in bitcoins

Parameter #3—a comment

Name	Type	Presence	Description
Comment	string	Optional (0 or 1)	A locally-stored (not broadcast) comment assigned to this transaction. Default is no comment

Parameter #4—a comment about who the payment was sent to

Name	Type	Presence	Description

Comment To	string	Optional (0 or 1)	A locally-stored (not broadcast) comment assigned to this transaction. Meant to be used for describing who the payment was sent to. Default is no comment

Result—a TXID of the sent transaction

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	The TXID of the sent transaction, encoded as hex in RPC byte order

Example from Bitcoin Core 0.10.0

Spend 0.1 bitcoins to the address below with the comment “sendtoadress example” and the comment-to “Nemo From Example.com”:

```
bitcoin-cli -testnet sendtoaddress mmXgiR6KAhZCyQ8ndr2BCfEq1wNG2UnyG6 \
  0.1 "sendtoaddress example" "Nemo From Example.com"
```

Result:

```
a2a2eb18cb051b5fe896a32b1cb20b179d981554b6bd7c5a956e56a0eecb04f0
```

See also

- SendFrom: spends an amount from a local account to a bitcoin address.
- SendMany: creates and broadcasts a transaction which sends outputs to multiple addresses.
- Move: moves a specified amount from one account in your wallet to another using an off-block-chain transaction.

SetAccount

Requires wallet support.

The `setaccount` RPC puts the specified address in the given account.

Parameter #1—a bitcoin address

Name	Type	Presence	Description
Address	string (base58)	Required (exactly 1)	The P2PKH or P2SH address to put in the account. Must already belong to the wallet

Parameter #2—an account

Name	Type	Presence	Description
Account	string	Required (exactly 1)	The name of the account in which the address should be placed. May be the default account, an empty string (“”)

Result—`null` if successful

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Set to JSON `null` if the address was successfully placed in the account

Example from Bitcoin Core 0.10.0

Put the address indicated below in the “doc test” account.

```
bitcoin-cli -testnet setaccount \
    mmXgiR6KAhZCyQ8ndr2BCfEq1wNG2UnyG6 "doc test"
```

(Success: no result displayed.)

See also

- GetAccount: returns the name of the account associated with the given address.
- ListAccounts: lists accounts and their balances.
- GetAddressesByAccount: returns a list of every address assigned to a particular account.

SetGenerate

Requires wallet support.

The `setgenerate` RPC enables or disables hashing to attempt to find the next block.

Parameter #1—whether to enable or disable generation

Name	Type	Presence	Description
Enable/Disable	bool	Required (exactly 1)	Set to `true` to enable generation; set to `false` to disable generation

Parameter #2 (regular)—the number of processors to use

Name	Type	Presence	Description
Processors	number (int)	Optional (0 or 1)	The number of processors to use. Defaults to `1`. Set to `-1` to use all processors

Parameter #2 (regtest)—the number of blocks to generate

Name	Type	Presence	Description
Blocks	number (int)	Optional (0 or 1)	In regtest mode, set to the number of blocks to generate. Defaults to `1`

Result (regular)—generating is enabled

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always JSON `null`

Result (regtest)—the generated block header hashes

Name	Type	Presence	Description
`result`	array/null	Required (exactly 1)	An array containing the block header hashes of the generated blocks, or JSON `null` if no blocks were generated

→ Header Hashes	string (hex)	Required (1 or more)	The hashes of the headers of the blocks generated in regtest mode, as hex in RPC byte order

Examples from Bitcoin Core 0.10.0

Enable generation using two logical processors (on this machine, two cores of one physical processor):

```
bitcoin-cli -testnet setgenerate true 2
```

(Success: no result displayed. Process manager shows 200% CPU usage.)

Using regtest mode, generate 2 blocks:

```
bitcoin-cli -regtest setgenerate true 101
```

Result:

```
[
    "7e38de938d0dcbb41be63d78a8353e77e9d1b3ef82e0368eda051d4926eef915",
    "61d6e5f1a64d009659f45ef1c614e57f4aa0501708641212be236dc56d726da8"
]
```

See also

- GetGenerate: returns true if the node is set to generate blocks using its CPU.
- GetHashesPerSec: was removed in Bitcoin Core master (unreleased). It returned a recent hashes per second performance measurement when the node was generating blocks.
- GetMiningInfo: returns various mining-related information.
- GetBlockTemplate: gets a block template or proposal for use with mining software.

SetTxFee

Requires wallet support.

The `settxfee` RPC sets the transaction fee per kilobyte paid by transactions created by this wallet.

Parameter #1—the transaction fee amount per kilobyte

Name	Type	Presence	Description
Transaction Fee Per Kilobyte	number (bitcoins)	Required (exactly 1)	The transaction fee to pay, in bitcoins, for each kilobyte of transaction data. The value `0` will not be accepted. Be careful setting the fee too low—your transactions may not be relayed or included in blocks

Result: `true` on success

Name	Type	Presence	Description
`result`	bool (true)	Required (exactly 1)	Set to `true` if the fee was successfully set

Example from Bitcoin Core 0.10.0

Set the transaction fee per kilobyte to 100,000 satoshis.

```
bitcoin-cli -testnet settxfee 0.00100000
```

Result:

```
true
```

See also

- GetWalletInfo: provides information about the wallet.
- GetNetworkInfo: returns information about the node's connection to the network.

SignMessage

Requires wallet support. Requires an unlocked wallet or an unencrypted wallet.

The `signmessage` RPC signs a message with the private key of an address.

Parameter #1—the address corresponding to the private key to sign with

Name	Type	Presence	Description
Address	string (base58)	Required (exactly 1)	A P2PKH address whose private key belongs to this wallet

Parameter #2—the message to sign

Name	Type	Presence	Description
Message	string	Required (exactly 1)	The message to sign

Result—the message signature

Name	Type	Presence	Description
`result`	string (base64)	Required (exactly 1)	The signature of the message, encoded in base64. Note that Bitcoin Core before 0.10.0 creates signatures with random *k* values, so each time you sign the same message, it will create a different signature

Example from Bitcoin Core 0.10.0

Sign a the message "Hello, World!" using the following address:

```
bitcoin-cli -testnet signmessage mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe \
    'Hello, World!'
```

Result:

```
IL98ziCmwYi5pL+dqKp4Ux+zCa4hP/xbjHmWh+Mk/lefV/0pWV1p/gQ94jgExSmgH2/+PDcCCrOHAady2IEySSI=
```

See also

- VerifyMessage: verifies a signed message.

SignRawTransaction

The `signrawtransaction` RPC signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.

Parameter #1—the transaction to sign

Name	Type	Presence	Description
Transaction	string (hex	Required (exactly 1)	The transaction to sign as a serialized transaction

Parameter #2—unspent transaction output details

Name	Type	Presence	Description
Dependencies	array	Optional (0 or 1)	The previous outputs being spent by this transaction
→ Output	object	Optional (0 or more)	An output being spent
→ → `txid`	string (hex)	Required (exactly 1)	The TXID of the transaction the output appeared in. The TXID must be encoded in hex in RPC byte order
→ → `vout`	number (int)	Required (exactly 1)	The index number of the output (vout) as it appeared in its transaction, with the first output being 0
→ → `scriptPubKey`	string (hex)	Required (exactly 1)	The output's pubkey script encoded as hex
→ → `redeemScript`	string (hex)	Optional (0 or 1)	If the pubkey script was a script hash, this must be the corresponding redeem script

Parameter #3—private keys for signing

Name	Type	Presence	Description
Private Keys	array	Optional (0 or 1)	An array holding private keys. If any keys are provided, only they will be used to sign the transaction (even if the wallet has other matching keys). If this array is empty or not used, and wallet support is enabled, keys from the wallet will be used
→ Key	string (base58)	Required (1 or more)	A private key in base58check format to use to create a signature for this transaction

Parameter #4—signature hash type

Name	Type	Presence	Description
SigHash	string	Optional (0 or 1)	The type of signature hash to use for all of the signatures performed. (You must use separate calls to the `signrawtransaction` RPC if you want to use different signature hash types for different signatures. The allowed values are: `ALL`, `NONE`, `SINGLE`, `ALL|ANYONECANPAY`, `NONE|ANYONECANPAY`, and `SINGLE|ANYONECANPAY`

Result—the transaction with any signatures made

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	The results of the signature
→ `hex`	string (hex)	Required (exactly 1)	The resulting serialized transaction encoded as hex with any signatures made inserted. If no signatures were made, this will be the same transaction provided in parameter #1
→ `complete`	bool	Required (exactly 1)	The value `true` if transaction is fully signed; the value `false` if more signatures are required

Example from Bitcoin Core 0.10.0

Sign the hex generated in the example section for the `createrawtransaction` RPC:

```
bitcoin-cli -testnet signrawtransaction 01000000011da9283b4ddf8d\
89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e0000000000ffff\
ffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a51\
59f58488ac00000000
```

Result:

```
{
    "hex" : "01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a47304402200ebea9f630f3ee35fa467ff
    "complete" : true
}
```

See also

- CreateRawTransaction: creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
- DecodeRawTransaction: decodes a serialized transaction hex string into a JSON object describing the transaction.
- SendRawTransaction: validates a transaction and broadcasts it to the peer-to-peer network.

Stop

The `stop` RPC safely shuts down the Bitcoin Core server.

Parameters: none

Result—the server is safely shut down

Name	Type	Presence	Description
`result`	string	Required (exactly 1)	The string "Bitcoin server stopping"

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet stop
```

Result:

```
Bitcoin server stopping
```

See also: none

SubmitBlock

The `submitblock` RPC accepts a block, verifies it is a valid addition to the block chain, and broadcasts it to the network. Extra parameters are ignored by Bitcoin Core but may be used by mining pools or other programs.

Parameter #1—the new block in serialized block format as hex

Name	Type	Presence	Description
Block	string (hex)	Required (exactly 1)	The full block to submit in serialized block format as hex

Parameter #2—additional parameters

Name	Type	Presence	Description
Parameters	object	Optional (0 or 1)	A JSON object containing extra parameters. Not used directly by Bitcoin Core and also not broadcast to the network. This is available for use by mining pools and other software. A common parameter is a `workid` string

Result—`null` or error string

Name	Type	Presence	Description
`result`	null/string	Required (exactly 1)	If the block submission succeeded, set to JSON `null`. If submission failed, set to one of the following strings: `duplicate`, `duplicate-invalid`, `inconclusive`, or `rejected`. The JSON-RPC `error` field will still be set to `null` if submission failed for one of these reasons

Example from Bitcoin Core 0.10.0

Submit the following block with the workid, “test”.

```
bitcoin-cli -testnet submitblock 02000000df11c014a8d798395b5059c\
722ebdf3171a4217ead71bf6e0e99f4c7000000004a6f6a2db225c81e77773f6\
f0457bcb05865a94900ed11356d0b75228efb38c7785d6053ffff001d005d437\
0010100000001000000000000000000000000000000000000000000000000000\
0000000000000ffffffff0d03b477030164062f503253482ffffffff0100f90\
29500000000232103adb7d8ef6b63de74313e0cd4e07670d09a169b13e4eda2d\
650f529332c47646dac00000000 \
'{ "workid": "test" }'
```

Result (the block above was already on a local block chain):

```
duplicate
```

See also

- GetBlockTemplate: gets a block template or proposal for use with mining software.

ValidateAddress

The `validateaddress` RPC returns information about the given Bitcoin address.

Parameter #1—a P2PKH or P2SH address

Name	Type	Presence	Description
Address	string (base58)	Required (exactly 1)	The P2PKH or P2SH address to validate encoded in base58check format

Result—information about the address

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	Information about the address
→ `isvalid`	bool	Required (exactly 1)	Set to `true` if the address is a valid P2PKH or P2SH address; set to `false` otherwise
→ `address`	string (base58)	Optional (0 or 1)	If the address is valid, this is the address
→ `ismine`	bool	Optional (0 or 1)	Set to `true` if the address belongs to the wallet; set to false if it does not. Only returned if wallet support enabled
→ `iswatchonly`	bool	Optional (0 or 1)	Set to `true` if the address is watch-only. Otherwise set to `false`. Only returned if address is in the wallet
→ `isscript`	bool	Optional (0 or 1)	Set to `true` if a P2SH address; otherwise set to `false`. Only returned if the address is in the wallet
→ `script`	string	Optional (0 or 1)	Only returned for P2SH addresses belonging to this wallet. This is the type of script: • `pubkey` for a P2PK script inside P2SH • `pubkeyhash` for a P2PKH script inside P2SH • `multisig` for a multisig script inside P2SH • `nonstandard` for unknown scripts
→ `hex`	string (hex)	Optional (0 or 1)	Only returned for P2SH addresses belonging to this wallet. This is the redeem script encoded as hex
→ `addresses`	array	Optional (0 or 1)	Only returned for P2SH addresses belonging to the wallet. A P2PKH addresses used in this script, or the computed P2PKH addresses of any pubkeys in this script. This array will be empty for `nonstandard` script types
→ → Address	string	Optional (0 or more)	A P2PKH address
→ `sigrequired`	number (int)	Optional (0 or 1)	Only returned for multisig P2SH addresses belonging to the wallet. The number of signatures required by this script
→ `pubkey`	string (hex)	Optional (0 or 1)	The public key corresponding to this address. Only returned if the address is a P2PKH address in the wallet
→ `iscompressed`	bool	Optional (0 or 1)	Set to `true` if a compressed public key or set to `false` if an uncompressed public key. Only returned if the address is a P2PKH address in the wallet
→		Optional	The account this address belong to. May be an empty string for the default account.

`account`	string	(0 or 1)	Only returned if the address belongs to the wallet

Example from Bitcoin Core 0.10.0

Validate the following P2PKH address from the wallet:

```
bitcoin-cli -testnet validateaddress mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe
```

Result:

```
{
    "isvalid" : true,
    "address" : "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe",
    "ismine" : true,
    "iswatchonly" : false,
    "isscript" : false,
    "pubkey" : "03bacb84c6464a58b3e0a53cc0ba4cb3b82848cd7bed25a7724b3cc75d76c9c1ba",
    "iscompressed" : true,
    "account" : "test label"
}
```

Validate the following P2SH multisig address from the wallet:

```
bitcoin-cli -testnet validateaddress 2MyVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq
```

Result:

```
{
    "isvalid" : true,
    "address" : "2MyVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq",
    "ismine" : true,
    "iswatchonly" : false,
    "isscript" : true,
    "script" : "multisig",
    "hex" : "522103ede722780d27b05f0b1169efc90fa15a601a32fc6c3295114500c586831b6aaf2102ecd2d250a76d204011de6bc365a56033b9b3a149f67
    "addresses" : [
        "mjbLRSidW1MY8oubvs4SMEnHNFXxCcoehQ",
        "mo1vzGwCzWqteip29vGWWW6MsEBREuzW94",
        "mt17cV37fBqZsnMmrHnGCm9pM28R1kQdMG"
    ],
    "sigsrequired" : 2,
    "account" : "test account"
}
```

See also

- ImportAddress: adds an address or pubkey script to the wallet without the associated private key, allowing you to watch for transactions affecting that address or pubkey script without being able to spend any of its outputs.
- GetNewAddress: returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.

VerifyChain

The `verifychain` RPC verifies each entry in the local block chain database.

Parameter #1—how thoroughly to check each block

Name	Type	Presence	Description

Check Level	number (int)	Optional (0 or 1)	How thoroughly to check each block, from 0 to 4. Default is the level set with the `-checklevel` command line argument; if that isn't set, the default is `3`. Each higher level includes the tests from the lower levels Levels are: **0.** Read from disk to ensure the files are accessible **1.** Ensure each block is valid **2.** Make sure undo files can be read from disk and are in a valid format **3.** Test each block undo to ensure it results in correct state **4.** After undoing blocks, reconnect them to ensure they reconnect correctly

Parameter #2—the number of blocks to check

Name	Type	Presence	Description
Number Of Blocks	number (int)	Optional (0 or 1)	The number of blocks to verify. Set to `0` to check all blocks. Defaults to the value of the `-checkblocks` command-line argument; if that isn't set, the default is `288`

Result—verification results

Name	Type	Presence	Description
`result`	bool	Required (exactly 1)	Set to `true` if verified; set to `false` if verification failed for any reason

Example from Bitcoin Core 0.10.0

Verify the most recent 10,000 blocks in the most through way:

```
bitcoin-cli -testnet verifychain 4 10000
```

Result (took 4 minutes and 25 seconds on a generic PC laptop; it would've taken much longer on mainnet):

```
true
```

See also

- GetBlockChainInfo: provides information about the current state of the block chain.
- GetTxOutSetInfo: returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.

VerifyMessage

The `verifymessage` RPC verifies a signed message.

Parameter #1—the address corresponding to the signing key

Name	Type	Presence	Description
Address	string (base58)	Required (exactly 1)	The P2PKH address corresponding to the private key which made the signature. A P2PKH address is a hash of the public key corresponding to the private key which made the signature. When the ECDSA signature is checked, up to four possible ECDSA public keys will be reconstructed from from the signature; each key will be hashed and compared against the P2PKH address provided to see if any of them match. If there are no matches, signature

			validation will fail

Parameter #2—the signature

Name	Type	Presence	Description
Signature	string (base64)	Required (exactly 1)	The signature created by the signer encoded as base-64 (the format output by the `signmessage` RPC)

Parameter #3—the message

Name	Type	Presence	Description
Message	string	Required (exactly 1)	The message exactly as it was signed (e.g. no extra whitespace)

Result: `true`, `false`, or an error

Name	Type	Presence	Description
`result`	bool/null	Required (exactly 1)	Set to `true` if the message was signed by a key corresponding to the provided P2PKH address; set to `false` if it was not signed by that key; set to JSON `null` if an error occurred

Example from Bitcoin Core 0.10.0

Check the signature on the message created in the example for `signmessage`:

```
bitcoin-cli -testnet verifymessage \
  mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe \
  IL98ziCmwYi5pL+dqKp4Ux+zCa4hP/xbjHmWh+Mk/lefV/0pWV1p/gQ94jgExSmgH2/+PDcCCrOHAady2IEySSI= \
  'Hello, World!'
```

Result:

```
true
```

See also

- SignMessage: signs a message with the private key of an address.

WalletLock

Requires wallet support. Requires an unlocked wallet.

The `walletlock` RPC removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call `walletpassphrase` again before being able to call any methods which require the wallet to be unlocked.

Parameters: none

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always set to JSON `null`

Example from Bitcoin Core 0.10.0

```
bitcoin-cli -testnet walletlock
```

(Success: nothing printed.)

See also

- EncryptWallet: encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.
- WalletPassphrase: stores the wallet decryption key in memory for the indicated number of seconds. Issuing the `walletpassphrase` command while the wallet is already unlocked will set a new unlock time that overrides the old one.
- WalletPassphraseChange: changes the wallet passphrase from 'old passphrase' to 'new passphrase'.

WalletPassphrase

Requires wallet support. Requires an encrypted wallet.

The `walletpassphrase` RPC stores the wallet decryption key in memory for the indicated number of seconds. Issuing the `walletpassphrase` command while the wallet is already unlocked will set a new unlock time that overrides the old one.

⚠ **Warning:** if using this RPC on the command line, remember that your shell probably saves your command lines (including the value of the passphrase parameter).

Parameter #1—the passphrase

Name	Type	Presence	Description
Passphrase	string	Required (exactly 1)	The passphrase that unlocks the wallet

Parameter #2—the number of seconds to leave the wallet unlocked

Name	Type	Presence	Description
Seconds	number (int)	Required (exactly 1)	The number of seconds after which the decryption key will be automatically deleted from memory

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always set to JSON `null`

Example from Bitcoin Core 0.10.0

Unlock the wallet for 10 minutes (the passphrase is "test"):

```
bitcoin-cli -testnet walletpassphrase test 600
```

(Success: no result printed.)

See also

- EncryptWallet: encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.
- WalletPassphraseChange: changes the wallet passphrase from 'old passphrase' to 'new passphrase'.
- WalletLock: removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call `walletpassphrase` again before being able to call any methods which require the wallet to be unlocked.

WalletPassphraseChange

Requires wallet support. Requires an encrypted wallet.

The `walletpassphrasechange` RPC changes the wallet passphrase from 'old passphrase' to 'new passphrase'.

⚠ **Warning:** if using this RPC on the command line, remember that your shell probably saves your command lines (including the value of the passphrase parameter).

Parameter #1—the current passphrase

Name	Type	Presence	Description
Current Passphrase	string	Required (exactly 1)	The current wallet passphrase

Parameter #2—the new passphrase

Name	Type	Presence	Description
New Passphrase	string	Required (exactly 1)	The new passphrase for the wallet

Result—`null` on success

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always set to JSON `null`

Example from Bitcoin Core 0.10.0

Change the wallet passphrase from "test" to "example":

```
bitcoin-cli -testnet walletpassphrasechange test example
```

(Success: no result printed.)

See also

- EncryptWallet: encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.
- WalletPassphrase: stores the wallet decryption key in memory for the indicated number of seconds. Issuing the `walletpassphrase` command while the wallet is already unlocked will set a new unlock time that overrides the old one.
- WalletLock: removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call `walletpassphrase` again before being able to call any methods which require the wallet to be unlocked.

HTTP REST

As of version 0.10.0, Bitcoin Core provides an **unauthenticated** HTTP REST interface. The interface runs on the same port as the JSON-RPC interface, by default port 8332 for mainnet and port 18332 for testnet. It must be enabled by either starting Bitcoin Core with the `-rest` option or by specifying `rest=1` in the configuration file.

The interface is not intended for public access and is only accessible from localhost by default.

⚠ **Warning:** A web browser can access a HTTP REST interface running on localhost, possibly allowing third parties to use cross-site scripting attacks to download your transaction and block data, reducing your privacy. If you have privacy concerns, you should not run a browser on the same computer as a REST-enabled Bicoin Core node.

The interface uses standard HTTP status codes and returns a plain-text description of errors for debugging.

Quick Reference

- GET Block gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block. **New in 0.10.0**
- GET Block/NoTxDetails gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block. The JSON object includes TXIDs for transactions within the block rather than the complete transactions GET block returns. **New in 0.10.0**
- GET Tx gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so this method may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings. **New in 0.10.0**

Requests

⚠ **Warning:** the block chain and memory pool can include arbitrary data which several of the commands below will return in hex format. If you convert this data to another format in an executable context, it could be used in an exploit. For example, displaying a pubkey script as ASCII text in a webpage could add arbitrary Javascript to that page and create a cross-site scripting (XSS) exploit. To avoid problems, please treat block chain and memory pool data as an arbitrary input from an untrusted source.

GET Block

The `GET block` operation gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.

Request

```
GET /block/<hash>.<format>
```

Parameter #1—the header hash of the block to retrieve

Name	Type	Presence	Description
Header Hash	path (hex)	Required (exactly 1)	The hash of the header of the block to get, encoded as hex in RPC byte order

Parameter #2—the output format

Name	Type	Presence	Description

Format	suffix	Required (exactly 1)	Set to `.json` for decoded block contents in JSON, or `.bin` or `hex` for a serialized block in binary or hex

Response as JSON

Name	Type	Presence	Description
Result	object	Required (exactly 1)	An object containing the requested block
→ `hash`	string (hex)	Required (exactly 1)	The hash of this block's block header encoded as hex in RPC byte order. This is the same as the hash provided in parameter #1
→ `confirmations`	number (int)	Required (exactly 1)	The number of confirmations the transactions in this block have, starting at 1 when this block is at the tip of the best block chain. This score will be -1 if the the block is not part of the best block chain
→ `size`	number (int)	Required (exactly 1)	The size of this block in serialized block format, counted in bytes
→ `height`	number (int)	Required (exactly 1)	The height of this block on its block chain
→ `version`	number (int)	Required (exactly 1)	This block's version number. See block version numbers
→ `merkleroot`	string (hex)	Required (exactly 1)	The merkle root for this block, encoded as hex in RPC byte order
→ `tx`	array	Required (exactly 1)	An array containing all transactions in this block. The transactions appear in the array in the same order they appear in the serialized block
→ → Transaction	object	Required (1 or more)	An object describing a particular transaction within this block
→ → → `txid`	string (hex)	Required (exactly 1)	The transaction's TXID encoded as hex in RPC byte order
→ → → `version`	number (int)	Required (exactly 1)	The transaction format version number
→ → → `locktime`	number (int)	Required (exactly 1)	The transaction's locktime: either a Unix epoch date or block height; see the Locktime parsing rules
→ → → `vin`	array	Required (exactly 1)	An array of objects with each object being an input vector (vin) for this transaction. Input objects will have the same order within the array as they have in the transaction, so the first input listed will be input 0
→ → → → Input	object	Required (1 or more)	An object describing one of this transaction's inputs. May be a regular input or a coinbase
→ → → → → `txid`	string	Optional (0 or 1)	The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction
→ → → → → `vout`	number (int)	Optional (0 or 1)	The output index number (vout) of the outpoint being spent. The first output in a transaction has an index of `0`. Not present if this is a coinbase transaction

→ → → → → `scriptSig`	object	Optional (0 or 1)	An object describing the signature script of this input. Not present if this is a coinbase transaction
→ → → → → → `asm`	string	Required (exactly 1)	The signature script in decoded form with non-data-pushing op codes listed
→ → → → → → `hex`	string (hex)	Required (exactly 1)	The signature script encoded as hex
→ → → → → `coinbase`	string (hex)	Optional (0 or 1)	The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction
→ → → → → `sequence`	number (int)	Required (exactly 1)	The input sequence number
→ → → `vout`	array	Required (exactly 1)	An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0
→ → → → Output	object	Required (1 or more)	An object describing one of this transaction's outputs
→ → → → → `value`	number (bitcoins)	Required (exactly 1)	The number of bitcoins paid to this output. May be `0`
→ → → → → `n`	number (int)	Required (exactly 1)	The output index number of this output within this transaction
→ → → → → `scriptPubKey`	object	Required (exactly 1)	An object describing the pubkey script
→ → → → → → `asm`	string	Required (exactly 1)	The pubkey script in decoded form with non-data-pushing op codes listed
→ → → → → → `hex`	string (hex)	Required (exactly 1)	The pubkey script encoded as hex
→ → → → → → `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for `nulldata` or `nonstandard` script types (see the `type` key below)
→ → → → → → `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script • `pubkeyhash` for a P2PKH script • `scripthash` for a P2SH script • `multisig` for a bare multisig script • `nulldata` for nulldata scripts • `nonstandard` for unknown scripts
→ → → → → → `addresses`	string : array	Optional (0 or 1)	The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for `nulldata` or `nonstandard` script types
→ → → → → → → Address	string	Required (1 or more)	A P2PKH or P2SH address

→ `time`	number (int)	Required (exactly 1)	The value of the *time* field in the block header, indicating approximately when the block was created
→ `nonce`	number (int)	Required (exactly 1)	The nonce which was successful at turning this particular block into one that could be added to the best block chain
→ `bits`	string (hex)	Required (exactly 1)	The value of the *nBits* field in the block header, indicating the target threshold this block's header had to pass
→ `difficulty`	number (real)	Required (exactly 1)	The estimated amount of work done to find this block relative to the estimated amount of work done to find block 0
→ `chainwork`	string (hex)	Required (exactly 1)	The estimated number of block header hashes miners had to check from the genesis block to this block, encoded as big-endian hex
→ `previousblockhash`	string (hex)	Required (exactly 1)	The hash of the header of the previous block, encoded as hex in RPC byte order
→ `nextblockhash`	string (hex)	Optional (0 or 1)	The hash of the next block on the best block chain, if known, encoded as hex in RPC byte order

Examples from Bitcoin Core 0.10.0

Request a block in hex-encoded serialized block format:

```
curl http://localhost:18332/rest/block/000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39.hex
```

Result (wrapped):

```
02000000df11c014a8d798395b5059c722ebdf3171a4217ead71bf6e0e99f4c7\
000000004a6f6a2db225c81e77773f6f0457bcb05865a94900ed11356d0b7522\
8efb38c7785d6053ffff001d005d437001010000000100000000000000000000\
00000000000000000000000000000000000000000000ffffffff0d03b4770301\
64062f503253482fffffffff0100f9029500000000232103adb7d8ef6b63de74\
313e0cd4e07670d09a169b13e4eda2d650f529332c47646dac00000000
```

Get the same block in JSON:

```
curl http://localhost:18332/rest/block/000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39.json
```

Result (whitespaced added):

```
{
    "hash": "000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39",
    "confirmations": 91785,
    "size": 189,
    "height": 227252,
    "version": 2,
    "merkleroot": "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a",
    "tx": [
        {
            "txid": "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a",
            "version": 1,
            "locktime": 0,
            "vin": [
                {
                    "coinbase": "03b477030164062f503253482f",
                    "sequence": 4294967295
                }
            ],
            "vout": [
                {
                    "value": 25.0,
                    "n": 0,
                    "scriptPubKey": {
                        "asm": "03adb7d8ef6b63de74313e0cd4e07670d09a169b13e4eda2d650f529332c47646d OP_CHECKSIG",
                        "hex": "2103adb7d8ef6b63de74313e0cd4e07670d09a169b13e4eda2d650f529332c47646dac",
                        "reqSigs": 1,
                        "type": "pubkey",
                        "addresses": [
                            "muXeUp1QYscuPRFH3qHtSrHyG6DQpvg7xZ"
                        ]
                    }
                }
            ]
        }
    ],
    "time": 1398824312,
    "nonce": 1883462912,
    "bits": "1d00ffff",
    "difficulty": 1.0,
    "chainwork": "000000000000000000000000000000000000000000000000083ada4a4009841a",
    "previousblockhash": "00000000c7f4990e6ebf71ad7e21a47131dfeb22c759505b3998d7a814c011df",
    "nextblockhash": "00000000afe1928529ac766f1237657819a11cfcc8ca6d67f119e868ed5b6188"
}
```

See also

- GET Block/NoTxDetails gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block. The JSON object includes TXIDs for transactions within the block rather than the complete transactions GET block returns.
- GetBlock RPC: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlockHash RPC: returns the header hash of a block at the given height in the local best block chain.
- GetBestBlockHash RPC: returns the header hash of the most recent block on the best block chain.

GET Block/NoTxDetails

The `GET block/notxdetails` operation gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block. The JSON object includes TXIDs for transactions within the block rather than the complete transactions GET block returns.

Request

```
GET /block/notxdetails/<hash>.<format>
```

Parameter #1—the header hash of the block to retrieve

Name	Type	Presence	Description
Header Hash	path (hex)	Required (exactly 1)	The hash of the header of the block to get, encoded as hex in RPC byte order

Parameter #2—the output format

Name	Type	Presence	Description
Format	suffix	Required (exactly 1)	Set to `.json` for decoded block contents in JSON, or `.bin` or `hex` for a serialized block in binary or hex

Response as JSON

Name	Type	Presence	Description
Result	object	Required (exactly 1)	An object containing the requested block
→ `hash`	string (hex)	Required (exactly 1)	The hash of this block's block header encoded as hex in RPC byte order. This is the same as the hash provided in parameter #1
→ `confirmations`	number (int)	Required (exactly 1)	The number of confirmations the transactions in this block have, starting at 1 when this block is at the tip of the best block chain. This score will be -1 if the the block is not part of the best block chain
→ `size`	number (int)	Required (exactly 1)	The size of this block in serialized block format, counted in bytes
→ `height`	number (int)	Required (exactly 1)	The height of this block on its block chain
→ `version`	number (int)	Required (exactly 1)	This block's version number. See block version numbers
→ `merkleroot`	string (hex)	Required (exactly 1)	The merkle root for this block, encoded as hex in RPC byte order
→ `tx`	array	Required (exactly 1)	An array containing all transactions in this block. The transactions appear in the array in the same order they appear in the serialized block
→ → TXID	string (hex)	Required (1 or more)	The TXID of a transaction in this block, encoded as hex in RPC byte order
→ `time`	number (int)	Required (exactly 1)	The value of the *time* field in the block header, indicating approximately when the block was created
→ `nonce`	number (int)	Required (exactly 1)	The nonce which was successful at turning this particular block into one that could be added to the best block chain
→ `bits`	string (hex)	Required (exactly 1)	The value of the *nBits* field in the block header, indicating the target threshold this block's header had to pass
→ `difficulty`	number (real)	Required (exactly 1)	The estimated amount of work done to find this block relative to the estimated amount of work done to find block 0
→ `chainwork`	string (hex)	Required (exactly 1)	The estimated number of block header hashes miners had to check from the genesis block to this block, encoded as big-endian hex
→ `previousblockhash`	string (hex)	Required (exactly 1)	The hash of the header of the previous block, encoded as hex in RPC byte order
→ `nextblockhash`	string (hex)	Optional (0 or 1)	The hash of the next block on the best block chain, if known, encoded as hex in RPC byte order

Examples from Bitcoin Core 0.10.0

Request a block in hex-encoded serialized block format:

```
curl http://localhost:18332/rest/block/notxdetails/000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39.hex
```

Result (wrapped):

```
02000000df11c014a8d798395b5059c722ebdf3171a4217ead71bf6e0e99f4c7\
000000004a6f6a2db225c81e77773f6f0457bcb05865a94900ed11356d0b7522\
8efb38c7785d6053ffff001d005d437001010000000100000000000000000000\
00000000000000000000000000000000000000000000ffffffff0d03b4770301\
64062f503253482fffffffff0100f9029500000000232103adb7d8ef6b63de74\
313e0cd4e07670d09a169b13e4eda2d650f529332c47646dac00000000
```

Get the same block in JSON:

```
curl http://localhost:18332/rest/block/notxdetails/000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39.json
```

Result (whitespaced added):

```
{
    "hash": "000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39",
    "confirmations": 91807,
    "size": 189,
    "height": 227252,
    "version": 2,
    "merkleroot": "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a",
    "tx": [
        "c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a"
    ],
    "time": 1398824312,
    "nonce": 1883462912,
    "bits": "1d00ffff",
    "difficulty": 1.0,
    "chainwork": "000000000000000000000000000000000000000000000000083ada4a4009841a",
    "previousblockhash": "00000000c7f4990e6ebf71ad7e21a47131dfeb22c759505b3998d7a814c011df",
    "nextblockhash": "00000000afe1928529ac766f1237657819a11cfcc8ca6d67f119e868ed5b6188"
}
```

See also

- GET Block: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlock RPC: gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.
- GetBlockHash RPC: returns the header hash of a block at the given height in the local best block chain.
- GetBestBlockHash RPC: returns the header hash of the most recent block on the best block chain.

GET Tx

The `GET tx` operation

Note: if you begin using `txindex=1` after downloading the block chain, you must rebuild your indexes by starting Bitcoin Core with the option `-reindex`. This may take several hours to complete, during which time your node will not process new blocks or transactions. This reindex only needs to be done once.

Request

```
GET /tx/<txid>.<format>
```

Parameter #1—the TXID of the transaction to retrieve

Name	Type	Presence	Description
TXID	path (hex)	Required (exactly 1)	The TXID of the transaction to get, encoded as hex in RPC byte order

Parameter #2—the output format

Name	Type	Presence	Description
Format	suffix	Required (exactly 1)	Set to `.json` for decoded transaction contents in JSON, or `.bin` or `hex` for a serialized transaction in binary or hex

Response as JSON

Name	Type	Presence	Description
Result	object	Required (exactly 1)	An object describing the request transaction
→ `txid`	string (hex)	Required (exactly 1)	The transaction's TXID encoded as hex in RPC byte order
→ `version`	number (int)	Required (exactly 1)	The transaction format version number
→ `locktime`	number (int)	Required (exactly 1)	The transaction's locktime: either a Unix epoch date or block height; see the Locktime parsing rules
→ `vin`	array	Required (exactly 1)	An array of objects with each object being an input vector (vin) for this transaction. Input objects will have the same order within the array as they have in the transaction, so the first input listed will be input 0
→ → Input	object	Required (1 or more)	An object describing one of this transaction's inputs. May be a regular input or a coinbase
→ → → `txid`	string	Optional (0 or 1)	The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction
→ → → `vout`	number (int)	Optional (0 or 1)	The output index number (vout) of the outpoint being spent. The first output in a transaction has an index of `0`. Not present if this is a coinbase transaction
→ → → `scriptSig`	object	Optional (0 or 1)	An object describing the signature script of this input. Not present if this is a coinbase transaction
→ → → → `asm`	string	Required (exactly 1)	The signature script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The signature script encoded as hex
→ → → `coinbase`	string (hex)	Optional (0 or 1)	The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction
→ → → `sequence`	number (int)	Required (exactly 1)	The input sequence number

→ `vout`	array	Required (exactly 1)	An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0
→ → Output	object	Required (1 or more)	An object describing one of this transaction's outputs
→ → → `value`	number (bitcoins)	Required (exactly 1)	The number of bitcoins paid to this output. May be `0`
→ → → `n`	number (int)	Required (exactly 1)	The output index number of this output within this transaction
→ → → `scriptPubKey`	object	Required (exactly 1)	An object describing the pubkey script
→ → → → `asm`	string	Required (exactly 1)	The pubkey script in decoded form with non-data-pushing op codes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The pubkey script encoded as hex
→ → → → `reqSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for `nulldata` or `nonstandard` script types (see the `type` key below)
→ → → → `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: • `pubkey` for a P2PK script • `pubkeyhash` for a P2PKH script • `scripthash` for a P2SH script • `multisig` for a bare multisig script • `nulldata` for nulldata scripts • `nonstandard` for unknown scripts
→ → → → `addresses`	string : array	Optional (0 or 1)	The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for `nulldata` or `nonstandard` script types
→ → → → → Address	string	Required (1 or more)	A P2PKH or P2SH address
→ `blockhash`	string (hex)	Optional (0 or 1)	If the transaction has been included in a block on the local best block chain, this is the hash of that block encoded as hex in RPC byte order
→ `confirmations`	number (int)	Required (exactly 1)	If the transaction has been included in a block on the local best block chain, this is how many confirmations it has. Otherwise, this is `0`
→ `time`	number (int)	Optional (0 or 1)	If the transaction has been included in a block on the local best block chain, this is the block header time of that block (may be in the future)
→ `blocktime`	number (int)	Optional (0 or 1)	This field is currently identical to the time field described above

Examples from Bitcoin Core 0.10.0

Request a transaction in hex-encoded serialized transaction format:

```
curl http://localhost:18332/rest/tx/ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e.hex
```

Result (wrapped):

```
0100000001268a9ad7bfb21d3c086f0ff28f73a064964aa069ebb69a9e437da8\
5c7e55c7d7000000006b483045022100ee69171016b7dd218491faf6e13f53d4\
0d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed28\
8d374397d30dff541b2dd45a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc\
8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326ffffffff0350ac60020000\
00001976a91456847befbd2360df0e35b4e3b77bae48585ae06888ac80969800\
000000001976a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac002d\
3101000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac\
00000000
```

Get the same transaction in JSON:

```
curl http://localhost:18332/rest/tx/ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e.json
```

Result (whitespaced added):

```
{
    "txid": "ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e",
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "txid": "d7c7557e5ca87d439e9ab6eb69a04a9664a0738ff20f6f083c1db2bfd79a8a26",
            "vout": 0,
            "scriptSig": {
                "asm": "3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed28
                "hex": "483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed
            },
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 0.39889999999999998,
            "n": 0,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068 OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac",
                "reqSigs": 1,
                "type": "pubkeyhash",
                "addresses": [
                    "moQR7i8XM4rSGoNwEsw3h4YEuduuP6mxw7"
                ]
            }
        },
        {
            "value": 0.10000000000000001,
            "n": 1,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020 OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac",
                "reqSigs": 1,
                "type": "pubkeyhash",
                "addresses": [
                    "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN"
                ]
            }
        },
        {
            "value": 0.20000000000000001,
            "n": 2,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160 0dfc8bafc8419853b34d5e072ad37d1a5159f584 OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
                "reqSigs": 1,
                "type": "pubkeyhash",
                "addresses": [
                    "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe"
                ]
            }
        }
    ],
    "blockhash": "00000000103e0091b7d27e5dc744a305108f0c752be249893c749e19c1c82317",
    "confirmations": 91916,
    "time": 1398734825,
    "blocktime": 1398734825
}
```

See also

- GetRawTransaction RPC: gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default `txindex=1` in your Bitcoin Core startup settings.
- GetTransaction RPC: gets detailed information about an in-wallet transaction.