Bitcoin Protocol Specification

Prasad Kumkar
prasad@chainscorelabs.com
Pune, INDIA

Updated: June 25, 2026

Abstract.

Bitcoin is the peer-to-peer electronic cash protocol and a payment network introduced by Satoshi Nakamoto. This document provides a technical reference and a consolidated knowledge base for the Bitcoin protocol. It is intended to be helpful for Bitcoin-based applications, node clients, and other infrastructure development within the ecosystem. It covers most of the modules implemented by contemporary full nodes: consensus rules, primitive data structures, serialization, peer networking, and relay behavior.

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Why this specification exists . . . 4
- 1.3 Related work . . . 4
- 1.4 Methodology . . . 5

2 Notation **5**
- 2.1 Types and values . . . 5
- 2.2 Sequences and records . . . 5
- 2.3 Encoding and byte order . . . 6
- 2.4 Hashes . . . 6
- 2.5 Identifiers . . . 6
- 2.6 Chain context . . . 6
- 2.7 Cryptographic relations . . . 7

3 Serialization **7**
- 3.1 Fixed-width primitives . . . 7
- 3.2 CompactSize . . . 7
- 3.3 Constructors . . . 8

4 Ledger Model **8**
- 4.1 State . . . 8
- 4.2 Coins . . . 9
- 4.3 Rule context . . . 9
- 4.4 Validity classes . . . 10
- 4.5 Transition relations . . . 10
- 4.6 Invariants . . . 10

5 Transactions **11**
- 5.1 Identifier and amount primitives . . . 11
- 5.2 Transaction records . . . 11
- 5.3 Legacy serialization . . . 12
- 5.4 Witness serialization . . . 12
- 5.5 Transaction identifiers . . . 13
- 5.6 Context-free validity . . . 13
- 5.7 Coinbase transaction . . . 13
- 5.8 Value conservation . . . 13
- 5.9 Locktime and sequence . . . 14
- 5.10 Signature hashes . . . 14

6 Script Execution **14**
- 6.1 Spend forms . . . 15
- 6.2 Machine state . . . 15
- 6.3 Evaluation . . . 16
- 6.4 Primitive relations . . . 16
- 6.5 Signature messages . . . 17
- 6.6 Opcode classes . . . 17

7 Blocks **24**
7.1 Block header . . . 24
7.2 Block record . . . 24
7.3 Merkle root . . . 25
7.4 Coinbase transaction . . . 25
7.5 Witness commitment . . . 25
7.6 Context-free block checks . . . 26
7.7 Contextual block checks . . . 26

8 Proof of Work and Chain Selection **26**
8.1 Target encoding . . . 27
8.2 Proof-of-work check . . . 27
8.3 Difficulty adjustment . . . 28
8.4 Accumulated work . . . 28

9 Consensus Validation **29**
9.1 Validation relation . . . 29
9.2 Header acceptance . . . 29
9.3 Block-data acceptance . . . 30
9.4 Best-chain selection . . . 30
9.5 Block connection . . . 31
9.6 UTXO transition . . . 31
9.7 Rule context . . . 31
9.8 Subsidy and fees . . . 32
9.9 Undo . . . 32
9.10 Reorganization correctness . . . 32

10 Mempool **32**
10.1 Entry . . . 33
10.2 Graph Invariants . . . 33
10.3 Admission . . . 34
10.4 Removal . . . 34
10.5 Orphans and Relay View . . . 35

11 Peer-to-Peer Network Protocol **35**
11.1 Message envelope . . . 35
11.2 Handshake . . . 35
11.3 Control and negotiation payloads . . . 36
11.4 Service flags . . . 36
11.5 Inventory and object retrieval . . . 36
11.6 Header and block synchronization . . . 37
11.7 Compact block relay . . . 38
11.8 Address relay . . . 38
11.9 Light-client services . . . 39

12 Acknowledgements **39**

Appendix A. Constants **40**

Appendix B. Mainnet Parameters **41**
B.1 Mainnet parameters 41
B.2 Historical exceptions 41

Appendix C. Remote Procedure Calls (RPC) **42**

1 Introduction

1.1 Background

Bitcoin was introduced by Satoshi Nakamoto as a peer-to-peer electronic cash system: a network in which payments can be validated by participants directly, without requiring a financial intermediary to order transactions or prevent double spending [19]. Its core design combines digital signatures, a public append-only history, a proof-of-work timestamp chain, and local validation by independently operated nodes. Transactions spend previously created outputs and create new outputs; blocks commit ordered transaction sets; nodes accept the valid chain with the greatest accumulated proof of work.

The consequence is a protocol whose authority is operational rather than institutional. No server, committee, or publication grants validity to a transaction or block. A block is useful only if economically relevant validating nodes accept it under their consensus rules, and a transaction is final only to the extent that it remains in the accepted proof-of-work history. This makes Bitcoin unusually sensitive to precise definitions: serialization, hash preimages, script evaluation, subsidy, difficulty adjustment, lock-time, witness commitments, and deployment rules are not merely engineering details. They are the boundary between a node that follows the same network and a node that has forked into a different system.

1.2 Why this specification exists

Bitcoin does not have a single normative protocol specification in the style of an Internet RFC. Its design is distributed across the original paper, Bitcoin Improvement Proposals, Bitcoin Core source code, release notes, developer documentation, test vectors, and long-running network practice [4, 5, 6, 7]. That distribution has served Bitcoin well: it keeps proposals reviewable, lets implementations evolve conservatively, and avoids granting authority to a document detached from deployed validation behavior. It also leaves developers with a practical problem. To implement, audit, test, or reason about Bitcoin, they must assemble one protocol from many artifacts that use different levels of precision and often mix consensus, policy, wallet behavior, and operational guidance.

This specification is intended to be the consolidated technical reference for that protocol. It states Bitcoin as data structures, encodings, identifiers, state variables, validation predicates, state transitions, and peer messages. It is not a proposal to change Bitcoin, not a replacement for BIPs, and not an authority over consensus. Where this document conflicts with economically relevant node validation, node validation wins. Its purpose is to make the protocol easier to inspect: one place where the rules needed by independent implementers, reviewers, researchers, and protocol engineers are expressed coherently.

1.3 Related work

The original Bitcoin paper defines the motivating problem and the high-level mechanism: timestamped proof of work over a chain of transaction-containing blocks [19]. BIPs refine that design by documenting concrete changes, conventions, and interoperability standards, including consensus soft forks, peer services, wallet formats, and application-level conventions [16, 4]. Bitcoin Core is the dominant full-node implementation and the principal executable reference for contemporary validation and relay behavior [5, 6]. Developer references and generated API documentation are valuable guides, but they are not a complete formal specification of the network protocol [7].

Other cryptocurrency systems have benefited from standalone protocol specifications. The Zcash protocol specification separates high-level protocol concepts from concrete encodings and upgrade-specific consensus rules [33]. Ethereum's Yellow Paper presents Ethereum as a transaction-based state machine with explicit state-transition functions [25]. The JAM graypaper follows the same tradition of a single coherent formal model for an independently implementable protocol [24]. This document applies that style to Bitcoin while respecting Bitcoin's different governance reality: no paper, repository, or committee has authority to redefine the deployed protocol.

1.4 Methodology

The baseline for this draft is Bitcoin Core source, its implemented-BIPs index, and the BIPs cited throughout this document are treated as evidence for contemporary full-node behavior [6]. The specification separates three classes of behavior:

1. Consensus rules: predicates that determine whether blocks and transactions can be accepted into the active chain.
2. Peer and relay rules: network records and local admission behavior used by nodes to exchange blocks, transactions, addresses, and negotiation state.
3. Local implementation behavior: storage layout, indexes, caches, RPC surfaces, wallet policy, user interface behavior, and performance choices that may explain Bitcoin Core but are not themselves protocol requirements.

Only the first two classes belong in the main specification. Implementation details are included only when they expose a protocol boundary or prevent an ambiguous reading of consensus or peer behavior. This keeps the document useful as a source-of-truth reference without turning it into a commentary on one code base.

2 Notation

This section fixes the symbols used across the specification. It defines only shared notation and reusable primitive relations; concrete record layouts, script execution, and signature-message construction are defined in their respective sections.

2.1 Types and values

Notation	Meaning
Bool	Truth values $\{\bot, \top\}$.
Byte	The integers $\{0, \ldots, 255\}$.
Bytes	Finite byte strings.
Bytes$[n]$	Byte strings of exactly n bytes.
N, **Z**	Nonnegative integers and integers.
$x : T$	Value x has type T.
$f : S \to T$	Total function from S to T.
$f(x) = \bot$	f is undefined, fails, or rejects input x.

2.2 Sequences and records

Notation	Meaning
$[x_0, \ldots, x_{n-1}]$	Ordered sequence with zero-based indexing.
$x[i]$	Element at index i.
$\lvert x \rvert$	Length of sequence or byte string x.
$x \lvert\lvert y$	Concatenation of sequences or byte strings.

Hex strings are written in display order. Sections that describe serialized fields state whether display order differs from wire order.

Record tables define field order, type, and meaning. Unless explicitly marked as logical state, a record table gives the order used by E_T.

2.3 Encoding and byte order

For a type or record T, $\mathrm{E}_T(x)$ is the canonical byte encoding of x as T. The partial inverse $\mathrm{E}_T^{-1}(b)$ parses bytes b as T and is undefined for non-canonical or invalid encodings. When parsing succeeds, $\mathrm{E}_T^{-1}(\mathrm{E}_T(x)) = x$.

$\mathrm{le}(b)$ is the unsigned little-endian integer represented by byte string b. Fixed-width integer encodings and CompactSize are defined in Section 3.

2.4 Hashes

Hash values are protocol bytes. Human display strings often reverse the bytes of 256-bit hashes.

$$\begin{aligned} \mathrm{H}(b) &= \mathrm{SHA256}(\mathrm{SHA256}(b)), \\ \mathrm{HASH160}(b) &= \mathrm{RIPEMD160}(\mathrm{SHA256}(b)), \\ \mathrm{TagHash}_{tag}(b) &= \mathrm{SHA256}(\mathrm{SHA256}(tag) \,||\, \mathrm{SHA256}(tag) || b). \end{aligned}$$

H is `hash256` in Bitcoin source material. Taproot and BIP340 Schnorr signatures use TagHash [19, 30, 31].

For a nonempty list of 32-byte hashes L, $\mathrm{MR}(L)$ is the Merkle root obtained by repeatedly replacing adjacent pairs (a, b) with $\mathrm{H}(a||b)$. If a level has odd length, the final hash is duplicated for that level. $\mathrm{MR}([]) = \bot$.

2.5 Identifiers

$$\begin{aligned} txid(tx) &= \mathrm{H}(\mathrm{E}_0(tx)), \\ wtxid(tx) &= \begin{cases} \mathrm{H}(\mathrm{E}_w(tx)), & \text{witness serialization is used,} \\ txid(tx), & \text{otherwise,} \end{cases} \\ \mathrm{bh}(h) &= \mathrm{H}(\mathrm{E}_{\mathrm{BlockHeader}}(h)). \end{aligned}$$

2.6 Chain context

For a chain C, $C[k]$ is the block at height k, with $C[0]$ as genesis. $\mathrm{height}(B)$ is the height of block B, and $\mathrm{tip}(C) = C[|C| - 1]$. $\mathrm{MTP}(C, k)$ is the median timestamp of $C[k]$ and its ten nearest ancestors, or all available ancestors when $k < 10$.

2.7 Cryptographic relations

All public-key signature relations use the secp256k1 group. ECDSAVerify(Q, m, σ) is true iff ECDSA signature σ verifies digest m under public key Q, subject to active script encoding rules. SchnorrVerify(Q, m, σ) is the BIP340 verification relation [28, 30].

3 Serialization

Bitcoin protocol objects are byte strings with fixed field order. Unless a field table states otherwise, E_T serializes integers in little-endian byte order, vectors with a CompactSize element count, and byte vectors with a CompactSize byte length followed by exactly that many bytes.

3.1 Fixed-width primitives

Primitive	Width	Serialization rule
`uint8` / `int8`	1 byte	Byte value as stored.
`uint16` / `int16`	2 bytes	Least significant byte first.
`uint32` / `int32`	4 bytes	Least significant byte first.
`uint64` / `int64`	8 bytes	Least significant byte first.
`byte[n]`	n bytes	Exactly the given bytes, in index order.
`bool`	1 byte	Zero is false; nonzero is true unless a field table restricts the value.
`uint160`	20 bytes	160-bit hash value.
`uint256`	32 bytes	256-bit hash value. Display hex commonly reverses these bytes.

Fixed-width integer fields are serialized in little-endian byte order unless a field table explicitly states otherwise [6].

3.2 CompactSize

Bitcoin transaction and block structures use CompactSize integers to encode counts and byte-vector lengths. CompactSize is canonical: a decoder rejects a value encoded with a longer form than necessary.

Value range	Encoding	Canonicality rule
$0 \le n < 253$	one byte: n	First byte is less than 0xfd.
$253 \le n \le 0xffff$	0xfd followed by `uint16`	Decoded value must be at least 253.
$2^{16} \le n \le 2^{32} - 1$	0xfe followed by `uint32`	Decoded value must be at least 0x10000.
$2^{32} \le n \le 2^{64} - 1$	0xff followed by `uint64`	Decoded value must be at least 0x100000000.

A parser rejects non-canonical CompactSize encodings and any decoded count or length that exceeds the enclosing field's rule.

3.3 Constructors

Constructor	Serialized form	Used for
`vector<T>`	`compactSize count || T[0] || ... || T[count-1]`	Transaction inputs, outputs, block transactions, inventory vectors, and payload lists.
`varbytes`	`compactSize length || byte[length]`	Scripts, signatures, public keys, witness items, and variable network fields.
`string`	`compactSize length || byte[length]`	P2P user-agent and payload strings. Character validity is context-specific.
`witnessStack`	`compactSize itemCount || varbytes[0] || ...`	One witness stack for one transaction input.

Scripts, signatures, public keys, witness items, and network payload fields reuse `varbytes` in different contexts. Parsing bytes into opcodes, signatures, keys, or scripts is a later semantic step.

For transactions and blocks, $\mathrm{E}_0(x)$ is the serialization with witness data stripped, and $\mathrm{E}_w(x)$ is the serialization including witness data where such data is defined. Define:

$$\mathit{strippedSize}(x) = |\mathrm{E}_0(x)|, \qquad \mathit{totalSize}(x) = |\mathrm{E}_w(x)|.$$

The consensus weight formula is:

$$\mathit{weight}(x) = 3 \cdot \mathit{strippedSize}(x) + \mathit{totalSize}(x).$$

For relay and fee accounting, virtual transaction size is the integer ceiling of weight divided by witness scale:

$$\mathit{vsize}(tx) = \left\lfloor \frac{\mathit{weight}(tx) + \texttt{WITNESS\_SCALE\_FACTOR} - 1}{\texttt{WITNESS\_SCALE\_FACTOR}} \right\rfloor.$$

Block consensus enforces `MAX_BLOCK_WEIGHT` against `weight(block)` [17, 6].

4 Ledger Model

Bitcoin validation is modeled as deterministic state transition over a proof-of-work header graph, available block data, and the active UTXO set. This section fixes the shared relations used by later sections.

4.1 State

A full-validation node's protocol state is:

$$S = (C, H, B, U, Q, \Pi)$$

Symbol	Type	Meaning
C	`ActiveChain`	Ordered sequence of connected blocks from genesis to the active tip.
H	`HeaderIndex`	Known block-header graph keyed by block hash, including height, predecessor, target, timestamp, and accumulated work.
B	`BlockDataSet`	Full block data known to the node, keyed by block hash.
U	`UTXOSet`	Map from unspent outpoints to coin records at the active-chain tip.
Q	`Mempool`	Locally accepted unconfirmed transactions. This is relay state, not consensus state.
Π	`PeerSet`	Connected-peer state and negotiated network capabilities.

Consensus validation reads C, H, B, U and mutates only C, H, B, U. Mempool and peer transitions read consensus state but do not define block validity.

4.2 Coins

The UTXO set maps outpoints to coins:

$$U[(txid, n)] = (value, scriptPubKey, height, coinbase).$$

Coin field	Type	Meaning
`value`	`Amount`	Output amount in satoshis. Consensus-valid values are in the money range.
`scriptPubKey`	`Script`	Locking script committed by the creating transaction output.
`height`	`uint32`	Active-chain height of the creating transaction.
`coinbase`	`bool`	True iff the creating transaction is the block coinbase.

4.3 Rule context

Let N be the selected network parameter set. For candidate height h, candidate block time t, and active chain C, define:

$$R = \text{Rules}(N, C, h, t).$$

R is the set of consensus rules active for the candidate object. It includes network constants, buried activation heights, versionbits state, timestamp rules, and script flags. Later sections use R

rather than restating whether P2SH, BIP34, strict DER, locktime, sequence locks, SegWit, Taproot, Tapscript, or network-specific timewarp mitigations are active.

4.4 Validity classes

Class	Meaning
$\mathrm{DecodeValid}_T(b)$	$\mathrm{E}_T^{-1}(b)$ succeeds and consumes a canonical encoding of type T.
$\mathrm{FormValid}(x)$	Object x is internally well-formed independent of chain state.
$\mathrm{ContextValid}_R(x, C)$	Object x satisfies height, time, and active-rule predicates under R.
$\mathrm{StateValid}_R(x, U)$	Object x is valid against the current spendable coin set.
$\mathrm{RelayValid}_P(x, Q, U)$	Object x is accepted by local relay policy. This is not a consensus predicate.

4.5 Transition relations

For a non-coinbase transaction, the transaction transition is:

$$\mathrm{ApplyTx}(R, U, tx) = \begin{cases} U', & \text{if } tx \text{ is valid against } U \text{ under } R, \\ \bot, & \text{otherwise.} \end{cases}$$

For a block B_h at height h, block connection is:

$$\mathrm{ConnectBlock}(R, (C, U), B_h) = \begin{cases} (C', U'), & \text{if every block and transaction predicate succeeds,} \\ \bot, & \text{otherwise.} \end{cases}$$

Header acceptance, block-data acceptance, chain selection, and reorganization are the consensus transitions defined in Section 9. Mempool admission is the local transition defined in Section 10.

4.6 Invariants

Every accepted active-chain transition preserves:

Invariant	Statement
Determinism	For fixed N, C, U, B_h, successful connection produces a unique C', U'.
Outpoint uniqueness	At any active tip, each outpoint appears at most once in U.
No double spend	A connected transaction consumes each input outpoint at most once, and a connected block consumes each live outpoint at most once.
Conservation	A non-coinbase transaction cannot create value: $\sum outputs(tx) \le \sum spent(tx)$.
Issuance bound	For each connected block, $\sum outputs(coinbase) \le subsidy(h) + \sum fees(B_h)$.

Work selection	The active chain is selected from valid candidates by greatest accumulated work.

5 Transactions

A Bitcoin transaction consumes previously created outputs and creates new outputs. Consensus validation distinguishes the serialized transaction bytes, the identifiers derived from those bytes, and the contextual checks that depend on the UTXO set and block height.

5.1 Identifier and amount primitives

Type	Encoding	Meaning
`Txid`	`uint256`	$txid(tx)$, as defined in Section 2.
`Wtxid`	`uint256`	$wtxid(tx)$, as defined in Section 2.
`OutPoint`	`Txid || uint32`	Reference to one transaction output by transaction identifier and output index.
`Amount`	`int64`	Satoshis. Consensus-valid output amounts satisfy $0 \leq value \leq$ `MAX_MONEY`.

5.2 Transaction records

`TxOut`

Field	Type	Encoding	Meaning
`value`	`Amount`	`int64`	Output value in satoshis.
`scriptPubKey`	`Script`	`varbytes`	Locking script.

`TxIn`

Field	Type	Encoding	Meaning
`previousOutput`	`OutPoint`	36 bytes	Output spent by this input, or the null outpoint for coinbase.
`scriptSig`	`Script`	`varbytes`	Unlocking script or coinbase data.
`sequence`	`uint32`	4 little-endian bytes	Finality, replacement, and relative-lock field.

`Transaction`

Field	Type	Encoding	Meaning

`version`	`int32`	4 little-endian bytes	Transaction version. Versions 1 and 2 are standard by default relay policy; consensus accepts the integer subject to contextual rules.
`inputs`	`vector<TxIn>`	CompactSize count, then inputs	Spent outputs or coinbase input.
`outputs`	`vector<TxOut>`	CompactSize count, then outputs	Created outputs.
`witnesses`	`vector<Witness>`	witness serialization only	One witness stack per input when witness serialization is used.
`lockTime`	`uint32`	4 little-endian bytes	Absolute height or time lock.

`Witness` is `vector<varbytes>`. It is present only in witness serialization and has exactly one stack per input.

5.3 Legacy serialization

$\mathrm{E}_0(tx)$, the legacy transaction serialization, is:

$$version||inputs||outputs||lockTime.$$

Part	Encoding	Meaning
`version`	`int32`	Transaction version.
`inputCount`	`compactSize`	Number of inputs.
`inputs`	`TxIn[inputCount]`	Serialized inputs in order.
`outputCount`	`compactSize`	Number of outputs.
`outputs`	`TxOut[outputCount]`	Serialized outputs in order.
`lockTime`	`uint32`	Absolute lock field.

5.4 Witness serialization

$\mathrm{E}_w(tx)$, the witness transaction serialization, is:

$$version||marker||flag||inputs||outputs||witnesses||lockTime.$$

Part	Encoding	Meaning
`marker`	`0x00`	Distinguishes witness serialization from legacy serialization.
`flag`	`0x01`	Indicates witness data follows. Other nonzero flag bits are invalid.

`witnesses`	`Witness[inputCount]`	One witness stack per input.

The witness form is valid only when at least one witness stack is nonempty. A transaction with no witness data uses legacy serialization for both `Txid` and `Wtxid` [17, 18].

5.5 Transaction identifiers

Identifier	Definition
`txid(tx)`	Defined in Section 2.
`wtxid(tx)`	Defined in Section 2.
`GenTxid`	Pair of identifier kind and hash used by relay to distinguish txid and wtxid announcements.

5.6 Context-free validity

A transaction is context-free valid only if:

1. it has at least one input and at least one output;
2. its stripped size satisfies $strippedSize(tx) \cdot \texttt{WITNESS\_SCALE\_FACTOR} \leq \texttt{MAX\_BLOCK\_WEIGHT}$;
3. every output value is in range;
4. the sum of output values is in range;
5. no two inputs reference the same previous output;
6. if it is coinbase, it has exactly one input whose previous output is the null outpoint and whose `scriptSig` length is between `COINBASE_SCRIPTSIG_MIN` and `COINBASE_SCRIPTSIG_MAX` bytes; and
7. if it is not coinbase, no input uses the null outpoint.

Context-free validity does not prove the inputs exist or signatures are valid; those checks require the UTXO set and rule context [6].

5.7 Coinbase transaction

A coinbase transaction has exactly one input whose previous output is the null outpoint defined by `NULL_TXID` and `NULL_INDEX`. Its input does not spend a UTXO. Its outputs create the block subsidy and collect fees, subject to the coinbase amount rule during block connection.

Coinbase-created outputs require `COINBASE_MATURITY` confirmations before they may be spent.

5.8 Value conservation

For a non-coinbase transaction:

$$fee(tx) = \sum value(spentInputs(tx)) - \sum value(outputs(tx)).$$

The transaction is invalid if any referenced input is absent, any spent coin is immature coinbase, any output is outside the money range, the output sum is outside the money range, or `fee(tx)` is negative [6].

5.9 Locktime and sequence

`lockTime` values below `LOCKTIME_THRESHOLD` are block heights; values at or above that threshold are Unix times. For a candidate block at height h with rule context R and locktime cutoff t, define the selected lock boundary:

$$\ell(tx, h, t) = \begin{cases} h, & tx.\texttt{lockTime} < \texttt{LOCKTIME\_THRESHOLD}, \\ t, & tx.\texttt{lockTime} \geq \texttt{LOCKTIME\_THRESHOLD}, \end{cases}$$

$$\begin{aligned} \text{AbsFinal}(tx, h, t) \iff tx.\texttt{lockTime} = 0 \lor tx.\texttt{lockTime} < \ell(tx, h, t) \lor \\ \forall i : \ tx.\texttt{inputs}[i].\texttt{sequence} = \texttt{SEQUENCE\_FINAL}. \end{aligned}$$

If R includes BIP113, $t = \text{MTP}(C, h-1)$; otherwise t is the candidate block timestamp.

If R includes BIP68, relative locks apply only to version-2-or-higher transactions and only to inputs whose `sequence` does not set `SEQUENCE_LOCKTIME_DISABLE_FLAG`. For validation view V, define $\text{SeqFinal}(tx, C, V, h)$ as follows. If $tx.\texttt{version} < 2$, it is true. Otherwise, for each applicable input i, let s_i be its sequence value, $v_i = s_i \,\&\, \texttt{SEQUENCE\_LOCKTIME\_MASK}$, and c_i be the height of the coin spent by input i in V. Height locks require:

$$h > c_i + v_i - 1.$$

When s_i sets `SEQUENCE_LOCKTIME_TYPE_FLAG`, the input is time locked instead. Let $\tau_i = \text{MTP}(C, c_i - 1)$. Time locks require:

$$\text{MTP}(C, h-1) > \tau_i + v_i \cdot \texttt{SEQUENCE\_LOCKTIME\_GRANULARITY} - 1.$$

The transaction satisfies relative locks iff every applicable input satisfies its selected height or time constraint; disabled inputs impose no constraint. If R does not include BIP68, SeqFinal is true [10, 8, 13, 6].

5.10 Signature hashes

Signature validation signs a digest derived from the transaction, input index, spent output, hash type, and active script version. Legacy, SegWit v0, and Taproot/Tapscript spend paths use different digest algorithms, specified in Section 6. These digests are consensus-critical because the signature checks validate exactly those bytes.

6 Script Execution

For each non-coinbase input i, spend authorization is the predicate:

$$\mathrm{Authorize}_R(tx, i, coin, u, H) \in \{\top, \bot\},$$

where $u = (scriptSig, witness)$ is the unlocking data, $coin$ is the spent UTXO, R is the active rule context, and H is the candidate height and median-time context. Authorization chooses a spend form, constructs a script machine state, and evaluates the committed program.

6.1 Spend forms

Form	Required relation
Legacy	$\mathrm{Eval}_R(scriptSig, \mathrm{BASE})$ then $\mathrm{Eval}_R(scriptPubKey, \mathrm{BASE})$; witness empty; final stack top is true.
P2SH	R includes P2SH; `scriptSig` is push-only; final pushed item is `redeemScript`; $\mathrm{HASH160}(redeemScript)$ matches; $\mathrm{Eval}_R(redeemScript, \mathrm{BASE})$ succeeds on the pushed stack.
Witness v0 key hash	Native witness program length 20; `scriptSig` empty unless P2SH-wrapped; witness stack is $(signature, pubkey)$; $\mathrm{HASH160}(pubkey)$ matches; ECDSA verifies M_{wit0}.
Witness v0 script hash	Native witness program length 32; final witness item is `witnessScript`; $\mathrm{SHA256}(witnessScript)$ matches; $\mathrm{Eval}_R(witnessScript, \mathrm{WITNESS\_V0})$ succeeds on the remaining witness stack.
Taproot key path	R includes Taproot; witness stack, after optional annex removal, contains one Schnorr signature; BIP340 verifies M_{tr} under the output key.
Taproot script path	Control block commits a tapleaf to the output key. Leaf version `0xc0` evaluates as Tapscript; unknown leaf versions succeed after commitment validation.
Unknown witness version	A syntactically valid witness program with no assigned meaning succeeds without execution.

Native witness spends require empty `scriptSig`; P2SH-wrapped witness spends require `scriptSig` to push exactly the witness program [2, 17, 14, 30, 31, 32].

6.2 Machine state

Script evaluation is a deterministic transition over:

$$X = (tx, i, coin, script, pc, S, A, C, R, \sigma, k, \alpha, \beta).$$

Field	Meaning
`script`, `pc`	Program bytes and instruction cursor.
S, A, C	Main stack, altstack, and conditional-execution stack.
R, σ	Active rule context and signature version: BASE, WITNESS_V0, or TAPSCRIPT.

k	Last executed code-separator position for signature-message construction.
α	Taproot annex, if present.
β	Tapscript validation-weight budget.

Legacy and witness-v0 execution enforce the script limits named in Appendix A. Tapscript removes the script-size and per-script-opcode limits, keeps the stack-item and stack-size limits, and sets:

$$\beta = \texttt{TAPSCRIPT\_SIGOPS\_BUDGET\_BASE} + \text{serializedWitnessSize}(\mathit{input}).$$

Each executed signature opcode with a nonempty signature subtracts `TAPSCRIPT_SIGOPS_COST`; negative β fails.

6.3 Evaluation

$$\text{Eval}_R(\mathit{script}, \sigma, S_0) = \begin{cases} \top, & \text{execution halts successfully with required final stack,} \\ \bot, & \text{parsing, execution, resource, or final-stack validation fails.} \end{cases}$$

If σ = TAPSCRIPT, the script is scanned for any `OP_SUCCESSx` index before ordinary execution; encountering one succeeds. Otherwise, instructions are decoded left to right and Step is applied while all entries of C are true. Non-control opcodes in inactive branches do not mutate S, but parse-stage failures and Tapscript `OP_SUCCESSx` detection still apply.

Successful termination requires an empty C. Legacy and witness-v0 require a nonempty S whose top element casts to true. Tapscript requires exactly one stack element, and that element must cast to true.

6.4 Primitive relations

Relation	Definition
CastToBool(x)	False iff x is empty, zero, or negative zero; true otherwise.
Num(x, n, min)	Signed-magnitude little-endian integer from byte vector x, length at most n; if min, nonzero encodings must be minimal.
MinimalPush(op, x)	True iff op is the shortest permitted push form for x.
LockTime(X, n)	Units match `tx.lockTime`, $n \leq tx.lockTime$, and the input sequence is nonfinal.
Sequence(X, n)	BIP68 is active in R, disable/type bits permit comparison, and the input sequence satisfies n.
CheckSig(X, sig, key, h)	Empty signatures return false. Legacy and witness-v0 use ECDSA over secp256k1; Taproot and Tapscript use BIP340 Schnorr. Tapscript key size 0 fails, size 32 verifies, and other nonzero key sizes are unknown key types that pass current consensus.

6.5 Signature messages

Signature checks verify exactly one digest:

$$\begin{aligned} M_{\text{legacy}} &= \text{SigMsg}_{\text{legacy}}(tx, i, k, h), \\ M_{\text{wit0}} &= \text{SigMsg}_{\text{wit0}}(tx, i, coin, k, h), \\ M_{\text{tr}} &= \text{SigMsg}_{\text{tr}}(tx, i, coin, \alpha, k, h). \end{aligned}$$

Hash type bytes are `0x00 DEFAULT` for Taproot, `0x01 ALL`, `0x02 NONE`, `0x03 SINGLE`, and `0x80 ANYONECANPAY` ORed with non-default forms. `ALL`, `NONE`, and `SINGLE` select the output commitment set; `ANYONECANPAY` restricts input commitment to the signed input. Legacy out-of-range `SINGLE` signs `uint256::ONE`; Taproot out-of-range `SINGLE` fails [14, 31, 32].

6.6 Opcode classes

Effects use `(before - after)` for stack transitions, with the rightmost item at the top of S. Parameterized direct byte pushes are represented as a single family; every named opcode is listed separately.

Index	Opcode	Effect
0	`OP_0 / OP_FALSE`	Push empty vector.
1-75	`OP_PUSHBYTES_n`	Push the next n script bytes.
76	`OP_PUSHDATA1`	Next `uint8` gives pushed byte length.
77	`OP_PUSHDATA2`	Next little-endian `uint16` gives pushed byte length.
78	`OP_PUSHDATA4`	Next little-endian `uint32` gives pushed byte length.
79	`OP_1NEGATE`	Push -1.
81	`OP_1`	Push integer 1.
82	`OP_2`	Push integer 2.
83	`OP_3`	Push integer 3.
84	`OP_4`	Push integer 4.
85	`OP_5`	Push integer 5.
86	`OP_6`	Push integer 6.
87	`OP_7`	Push integer 7.
88	`OP_8`	Push integer 8.
89	`OP_9`	Push integer 9.
90	`OP_10`	Push integer 10.
91	`OP_11`	Push integer 11.
92	`OP_12`	Push integer 12.
93	`OP_13`	Push integer 13.
94	`OP_14`	Push integer 14.
95	`OP_15`	Push integer 15.
96	`OP_16`	Push integer 16.

Index	Opcode	Effect
97	OP_NOP	No effect.
99	OP_IF	(v -); push CastToBool(v) to C.
100	OP_NOTIF	(v -); push negated condition to C.
101	OP_VERIF	Fail.
102	OP_VERNOTIF	Fail.
103	OP_ELSE	Toggle top of C; unmatched ELSE fails.
104	OP_ENDIF	Pop top of C; unmatched ENDIF fails.
105	OP_VERIFY	(v -) if true; otherwise fail.
106	OP_RETURN	Fail.

Index	Opcode	Effect
107	OP_TOALTSTACK	Move top of S to A.
108	OP_FROMALTSTACK	Move top of A to S.
109	OP_2DROP	(x1 x2 -).
110	OP_2DUP	(x1 x2 - x1 x2 x1 x2).
111	OP_3DUP	(x1 x2 x3 - x1 x2 x3 x1 x2 x3).
112	OP_2OVER	(x1 x2 x3 x4 - x1 x2 x3 x4 x1 x2).
113	OP_2ROT	(x1 x2 x3 x4 x5 x6 - x3 x4 x5 x6 x1 x2).
114	OP_2SWAP	(x1 x2 x3 x4 - x3 x4 x1 x2).
115	OP_IFDUP	Duplicate top item iff it casts true.
116	OP_DEPTH	Push stack depth.
117	OP_DROP	(x -).
118	OP_DUP	(x - x x).
119	OP_NIP	(x1 x2 - x2).
120	OP_OVER	(x1 x2 - x1 x2 x1).
121	OP_PICK	(n - x); copy stack item n to the top.
122	OP_ROLL	(n - x); move stack item n to the top.
123	OP_ROT	(x1 x2 x3 - x2 x3 x1).
124	OP_SWAP	(x1 x2 - x2 x1).
125	OP_TUCK	(x1 x2 - x2 x1 x2).
130	OP_SIZE	(x - x len(x)).

Index	Opcode	Effect
135	OP_EQUAL	Byte-vector equality.
136	OP_EQUALVERIFY	Equality followed by VERIFY.
139	OP_1ADD	(x - x+1) over Num($x, 4, min$).
140	OP_1SUB	(x - x-1) over Num($x, 4, min$).
143	OP_NEGATE	(x - -x) over Num($x, 4, min$).
144	OP_ABS	(x - |x|) over Num($x, 4, min$).
145	OP_NOT	Push 1 iff numeric input is zero; otherwise push 0.

146	OP_ONOTEQUAL	Push 0 iff numeric input is zero; otherwise push 1.
147	OP_ADD	(x y - x+y) over script numbers.
148	OP_SUB	(x y - x-y) over script numbers.
154	OP_BOOLAND	Push 1 iff both numeric inputs are nonzero.
155	OP_BOOLOR	Push 1 iff at least one numeric input is nonzero.
156	OP_NUMEQUAL	Numeric equality.
157	OP_NUMEQUALVERIFY	Numeric equality followed by VERIFY.
158	OP_NUMNOTEQUAL	Numeric inequality.
159	OP_LESSTHAN	Numeric less-than comparison.
160	OP_GREATERTHAN	Numeric greater-than comparison.
161	OP_LESSTHANOREQUAL	Numeric less-than-or-equal comparison.
162	OP_GREATERTHANOREQUAL	Numeric greater-than-or-equal comparison.
163	OP_MIN	Push the smaller numeric input.
164	OP_MAX	Push the larger numeric input.
165	OP_WITHIN	(x min max - ok) where $min \leq x < max$.

Index	Opcode	Effect
166	OP_RIPEMD160	(x - RIPEMD160(x)).
167	OP_SHA1	(x - SHA1(x)).
168	OP_SHA256	(x - SHA256(x)).
169	OP_HASH160	(x - HASH160(x)).
170	OP_HASH256	(x - H(x)).
171	OP_CODESEPARATOR	Set k to this opcode position.
172	OP_CHECKSIG	(sig key - ok) using CheckSig.
173	OP_CHECKSIGVERIFY	CHECKSIG followed by VERIFY.
174	OP_CHECKMULTISIG	(dummy sig[m] m key[n] n - ok); enforces historical dummy rule when active; disabled in Tapscript.
175	OP_CHECKMULTISIGVERIFY	CHECKMULTISIG followed by VERIFY; disabled in Tapscript.
186	OP_CHECKSIGADD	Tapscript only: (sig n key - n+ok).

Index	Opcode	Effect
176	OP_NOP1	No effect.
177	OP_CHECKLOCKTIMEVERIFY / OP_NOP2	If active in R, require LockTime(X, n); otherwise no effect.
178	OP_CHECKSEQUENCEVERIFY / OP_NOP3	If active in R, require Sequence(X, n); otherwise no effect.

179	OP_NOP4	No effect unless redefined by active soft-fork rules.
180	OP_NOP5	No effect unless redefined by active soft-fork rules.
181	OP_NOP6	No effect unless redefined by active soft-fork rules.
182	OP_NOP7	No effect unless redefined by active soft-fork rules.
183	OP_NOP8	No effect unless redefined by active soft-fork rules.
184	OP_NOP9	No effect unless redefined by active soft-fork rules.
185	OP_NOP10	No effect unless redefined by active soft-fork rules.

Index	Opcode	Effect
80	OP_RESERVED / OP_SUCCESS80	Reserved failure if executed outside Tapscript; Tapscript success.
98	OP_VER / OP_SUCCESS98	Reserved failure if executed outside Tapscript; Tapscript success.
126	OP_CAT / OP_SUCCESS126	Disabled failure outside Tapscript; Tapscript success.
127	OP_SUBSTR / OP_SUCCESS127	Disabled failure outside Tapscript; Tapscript success.
128	OP_LEFT / OP_SUCCESS128	Disabled failure outside Tapscript; Tapscript success.
129	OP_RIGHT / OP_SUCCESS129	Disabled failure outside Tapscript; Tapscript success.
131	OP_INVERT / OP_SUCCESS131	Disabled failure outside Tapscript; Tapscript success.
132	OP_AND / OP_SUCCESS132	Disabled failure outside Tapscript; Tapscript success.
133	OP_OR / OP_SUCCESS133	Disabled failure outside Tapscript; Tapscript success.
134	OP_XOR / OP_SUCCESS134	Disabled failure outside Tapscript; Tapscript success.
137	OP_RESERVED1 / OP_SUCCESS137	Reserved failure if executed outside Tapscript; Tapscript success.
138	OP_RESERVED2 / OP_SUCCESS138	Reserved failure if executed outside Tapscript; Tapscript success.
141	OP_2MUL / OP_SUCCESS141	Disabled failure outside Tapscript; Tapscript success.
142	OP_2DIV / OP_SUCCESS142	Disabled failure outside Tapscript; Tapscript success.
149	OP_MUL / OP_SUCCESS149	Disabled failure outside Tapscript; Tapscript success.

150	OP_DIV / OP_SUCCESS150	Disabled failure outside Tapscript; Tapscript success.
151	OP_MOD / OP_SUCCESS151	Disabled failure outside Tapscript; Tapscript success.
152	OP_LSHIFT / OP_SUCCESS152	Disabled failure outside Tapscript; Tapscript success.
153	OP_RSHIFT / OP_SUCCESS153	Disabled failure outside Tapscript; Tapscript success.
187	OP_SUCCESS187	Tapscript success; invalid outside Tapscript.
188	OP_SUCCESS188	Tapscript success; invalid outside Tapscript.
189	OP_SUCCESS189	Tapscript success; invalid outside Tapscript.
190	OP_SUCCESS190	Tapscript success; invalid outside Tapscript.
191	OP_SUCCESS191	Tapscript success; invalid outside Tapscript.
192	OP_SUCCESS192	Tapscript success; invalid outside Tapscript.
193	OP_SUCCESS193	Tapscript success; invalid outside Tapscript.
194	OP_SUCCESS194	Tapscript success; invalid outside Tapscript.
195	OP_SUCCESS195	Tapscript success; invalid outside Tapscript.
196	OP_SUCCESS196	Tapscript success; invalid outside Tapscript.
197	OP_SUCCESS197	Tapscript success; invalid outside Tapscript.
198	OP_SUCCESS198	Tapscript success; invalid outside Tapscript.
199	OP_SUCCESS199	Tapscript success; invalid outside Tapscript.
200	OP_SUCCESS200	Tapscript success; invalid outside Tapscript.
201	OP_SUCCESS201	Tapscript success; invalid outside Tapscript.
202	OP_SUCCESS202	Tapscript success; invalid outside Tapscript.
203	OP_SUCCESS203	Tapscript success; invalid outside Tapscript.
204	OP_SUCCESS204	Tapscript success; invalid outside Tapscript.
205	OP_SUCCESS205	Tapscript success; invalid outside Tapscript.
206	OP_SUCCESS206	Tapscript success; invalid outside Tapscript.

207	OP_SUCCESS207	Tapscript success; invalid outside Tapscript.
208	OP_SUCCESS208	Tapscript success; invalid outside Tapscript.
209	OP_SUCCESS209	Tapscript success; invalid outside Tapscript.
210	OP_SUCCESS210	Tapscript success; invalid outside Tapscript.
211	OP_SUCCESS211	Tapscript success; invalid outside Tapscript.
212	OP_SUCCESS212	Tapscript success; invalid outside Tapscript.
213	OP_SUCCESS213	Tapscript success; invalid outside Tapscript.
214	OP_SUCCESS214	Tapscript success; invalid outside Tapscript.
215	OP_SUCCESS215	Tapscript success; invalid outside Tapscript.
216	OP_SUCCESS216	Tapscript success; invalid outside Tapscript.
217	OP_SUCCESS217	Tapscript success; invalid outside Tapscript.
218	OP_SUCCESS218	Tapscript success; invalid outside Tapscript.
219	OP_SUCCESS219	Tapscript success; invalid outside Tapscript.
220	OP_SUCCESS220	Tapscript success; invalid outside Tapscript.
221	OP_SUCCESS221	Tapscript success; invalid outside Tapscript.
222	OP_SUCCESS222	Tapscript success; invalid outside Tapscript.
223	OP_SUCCESS223	Tapscript success; invalid outside Tapscript.
224	OP_SUCCESS224	Tapscript success; invalid outside Tapscript.
225	OP_SUCCESS225	Tapscript success; invalid outside Tapscript.
226	OP_SUCCESS226	Tapscript success; invalid outside Tapscript.
227	OP_SUCCESS227	Tapscript success; invalid outside Tapscript.
228	OP_SUCCESS228	Tapscript success; invalid outside Tapscript.
229	OP_SUCCESS229	Tapscript success; invalid outside Tapscript.
230	OP_SUCCESS230	Tapscript success; invalid outside Tapscript.

231	OP_SUCCESS231	Tapscript success; invalid outside Tapscript.
232	OP_SUCCESS232	Tapscript success; invalid outside Tapscript.
233	OP_SUCCESS233	Tapscript success; invalid outside Tapscript.
234	OP_SUCCESS234	Tapscript success; invalid outside Tapscript.
235	OP_SUCCESS235	Tapscript success; invalid outside Tapscript.
236	OP_SUCCESS236	Tapscript success; invalid outside Tapscript.
237	OP_SUCCESS237	Tapscript success; invalid outside Tapscript.
238	OP_SUCCESS238	Tapscript success; invalid outside Tapscript.
239	OP_SUCCESS239	Tapscript success; invalid outside Tapscript.
240	OP_SUCCESS240	Tapscript success; invalid outside Tapscript.
241	OP_SUCCESS241	Tapscript success; invalid outside Tapscript.
242	OP_SUCCESS242	Tapscript success; invalid outside Tapscript.
243	OP_SUCCESS243	Tapscript success; invalid outside Tapscript.
244	OP_SUCCESS244	Tapscript success; invalid outside Tapscript.
245	OP_SUCCESS245	Tapscript success; invalid outside Tapscript.
246	OP_SUCCESS246	Tapscript success; invalid outside Tapscript.
247	OP_SUCCESS247	Tapscript success; invalid outside Tapscript.
248	OP_SUCCESS248	Tapscript success; invalid outside Tapscript.
249	OP_SUCCESS249	Tapscript success; invalid outside Tapscript.
250	OP_SUCCESS250	Tapscript success; invalid outside Tapscript.
251	OP_SUCCESS251	Tapscript success; invalid outside Tapscript.
252	OP_SUCCESS252	Tapscript success; invalid outside Tapscript.
253	OP_SUCCESS253	Tapscript success; invalid outside Tapscript.
254	OP_SUCCESS254	Tapscript success; invalid outside Tapscript.

255	OP_INVALIDOPCODE	Fail.

Consensus script behavior is defined by the active P2SH, DER, locktime, sequence, NULLDUMMY, SegWit, Taproot, and Tapscript rules [22, 28, 10, 8, 13, 15, 31, 32, 6].

7 Blocks

A block is a header plus an ordered transaction vector. The header commits to the previous block header, transaction Merkle root, timestamp, target encoding, and nonce. The transaction vector begins with exactly one coinbase transaction.

7.1 Block header

A serialized `BlockHeader` is exactly 80 bytes in the field order shown:

`BlockHeader`			
Field	Type	Encoding	Meaning
`version`	`int32`	4 little-endian bytes	Block version. Version bits use the high-bit pattern defined by BIP 9.
`previous BlockHash`	`uint256`	32 bytes	Double-SHA256 hash of the previous block header in protocol byte order.
`merkleRoot`	`uint256`	32 bytes	Merkle root of transaction identifiers. Witness data is committed separately.
`time`	`uint32`	4 little-endian bytes	Miner-supplied Unix timestamp.
`bits`	`uint32`	4 little-endian bytes	Compact proof-of-work target encoding.
`nonce`	`uint32`	4 little-endian bytes	Nonce varied during proof-of-work search.

For a block B, $\mathrm{bh}(B) = \mathrm{bh}(B.\texttt{header})$.

7.2 Block record

`Block`			
Field	Type	Encoding	Meaning
`header`	`BlockHeader`	80 bytes	Serialized block header.
`txCount`	`compactSize`	CompactSize	Number of serialized transactions.

`transactions`	`vector<Transaction>`	transaction encodings in order	First transaction must be coinbase; all later transactions must not be coinbase.

When witness data is present, block serialization includes each transaction's witness serialization for relay and block-weight computation. The transaction identifier Merkle root continues to use `txid`, not `wtxid`.

7.3 Merkle root

For a block transaction vector $[tx_0, \ldots, tx_{n-1}]$, define:

$$L = [txid(tx_0), txid(tx_1), \ldots, txid(tx_{n-1})].$$

A block commits to $\mathrm{MR}(L)$. A block with empty L is invalid.

During root computation, a node rejects the duplicated-subtree mutation case: if two identical hashes are paired at a level before an odd-last duplication step, the block is invalid as mutated [6].

7.4 Coinbase transaction

The first transaction in each block is the coinbase transaction. It is the only transaction permitted to have a null prevout. No other transaction in a block may be coinbase. The coinbase creates the block subsidy and collects transaction fees; its total output value must not exceed subsidy plus fees when the block is connected.

$\mathrm{Rules}(N, C, h, t)$ determines whether the candidate height must be serialized at the beginning of the coinbase `scriptSig` [3, 6].

7.5 Witness commitment

If $R = \mathrm{Rules}(N, C, h, t)$ includes SegWit and any transaction has witness data, the coinbase must commit to the witness Merkle root. The witness tree leaves are:

$$W = [0^{256}, wtxid(tx_1), \ldots, wtxid(tx_{n-1})].$$

The coinbase input witness contains a 32-byte reserved value r. The witness commitment is:

$$wc = \mathrm{H}(\mathrm{MR}(W)||r).$$

The commitment appears in a coinbase output whose script begins with the SegWit commitment header. If multiple outputs match, the highest-index matching output is used [17].

7.6 Context-free block checks

Check	Rule
Transaction count	The block contains at least one transaction.
Transaction count weight	`txCount * WITNESS_SCALE_FACTOR <= MAX_BLOCK_WEIGHT`.
Coinbase position	Transaction zero is coinbase; no later transaction is coinbase.
Merkle root	Header `merkleRoot` equals $\mathrm{MR}(L)$ and is not mutated.
Stripped size	`strippedSize(block) * WITNESS_SCALE_FACTOR <= MAX_BLOCK_WEIGHT`.
Transaction validity	Every transaction passes context-free transaction checks.
Legacy sigops	`legacySigOps(block) * WITNESS_SCALE_FACTOR <= MAX_BLOCK_SIGOPS_COST`.

7.7 Contextual block checks

For a candidate block at height h extending chain C, the previous block is $C[h-1]$, and $R = \mathrm{Rules}(N, C, h, \texttt{time})$.

Check	Rule
Previous header	The previous header exists and is valid for extension.
Median time	`time` is greater than $\mathrm{MTP}(C, h-1)$.
Future time	A node does not accept the block while `time` is more than `MAX_FUTURE_BLOCK_TIME` after the node's current clock time; this is temporary future status, not permanent consensus invalidity.
BIP94 timewarp	On networks enforcing BIP94, the first block of each difficulty-adjustment interval except genesis must have `time >= previous.time - BIP94_TIMEWARP_GRACE`.
Difficulty	`bits` encodes the target required for the height and network.
Finality	All transactions satisfy $\mathrm{AbsFinal}(tx, h, t)$, where $t = \mathrm{MTP}(C, h-1)$ if R includes BIP113 and the candidate block time otherwise.
Coinbase height	If R includes BIP34, the block commits to the correct height in the coinbase.
Witness commitment	If R includes SegWit and the block has witness data, the block contains the correct coinbase witness commitment.
Weight	Block weight is at most `MAX_BLOCK_WEIGHT`.

The timestamp checks above follow the active validation predicates for median time past, future-time acceptance, and BIP94 timewarp mitigation [6, 12].

8 Proof of Work and Chain Selection

Bitcoin blocks commit to proof of work through the block header hash. A block is valid only if its header hash satisfies the target encoded by the header's `nBits` field.

8.1 Target encoding

The `nBits` field is a 32-bit compact target. For compact value c, let:

$$e = c \gg 24, \qquad m = c \,\&\, 0x007fffff, \qquad s = c \,\&\, 0x00800000.$$

The decoded target $\mathrm{C}^{-1}(c)$ is:

$$\mathrm{C}^{-1}(c) = \begin{cases} m \gg 8(3-e), & e \leq 3, \\ m \ll 8(e-3), & e > 3. \end{cases}$$

The sign bit is inherited from an older multiple-precision integer format and is not permitted for proof-of-work targets. A decoded target is invalid if any of the following conditions holds:

- s is set and $m \neq 0$;
- $\mathrm{C}^{-1}(c) = 0$;
- decoding overflows 256 bits, which occurs when $m \neq 0$ and either $e > 34$, or $m > 0xff$ and $e > 33$, or $m > 0xffff$ and $e > 32$;
- $\mathrm{C}^{-1}(c)$ is greater than the network's proof-of-work limit.

This compact-target decoding matches contemporary Bitcoin Core consensus validation [6].

The compact encoding $\mathrm{C}(T)$ of target T is computed as follows. Let `size = ceil(bitLength(T) / 8)`. If `size <= 3`, set `word = T « 8(3 - size)`; otherwise set `word = T » 8(size - 3)`. If `word` has bit `0x00800000` set, right-shift `word` by 8 and increment `size`. Then:

$$\mathrm{C}(T) = (size \ll 24) \mid (word \,\&\, 0x007fffff).$$

Consensus proof-of-work targets are nonnegative, so the sign bit is not set in valid block targets.

8.2 Proof-of-work check

For header h, let:

$$T_h = \mathrm{C}^{-1}(h.\texttt{bits}), \qquad p_h = \mathrm{le}(\mathrm{bh}(h)).$$

A candidate block satisfies proof of work iff T_h is a valid target and:

$$p_h \leq T_h.$$

A node rejects the block header if target decoding fails or if the hash integer is greater than the target. The network-specific proof-of-work limit is part of the chain parameters, not a value inferred from the genesis block [6].

8.3 Difficulty adjustment

Mainnet difficulty adjusts over fixed-height intervals. Test networks can have different adjustment behavior. Network-specific exceptions must be cited from chain parameter sources rather than inferred from mainnet behavior.

For mainnet:

$$\texttt{DIFFICULTY\_ADJUSTMENT\_INTERVAL} = \frac{\texttt{TARGET\_TIMESPAN}}{\texttt{TARGET\_SPACING}}.$$

A candidate block's `nBits` must equal the value returned by the network's difficulty-adjustment function for the previous block and candidate timestamp [6].

For networks without the minimum-difficulty exception, the algorithm is:

1. If the candidate height is not an exact difficulty-adjustment interval, return the previous block's `nBits`.
2. Otherwise, identify the first block in the previous adjustment period. For mainnet this is the ancestor `DIFFICULTY_ADJUSTMENT_INTERVAL - 1` blocks before `previous`.
3. Compute $\Delta = time(previous) - time(first)$.
4. Clamp Δ to the range

$$\frac{\texttt{TARGET\_TIMESPAN}}{\texttt{MAX\_RETARGET\_FACTOR}} \leq \Delta \leq \texttt{TARGET\_TIMESPAN} \cdot \texttt{MAX\_RETARGET\_FACTOR}.$$

5. Decode the previous period's base target. On testnet4-like networks with BIP94 enforcement, the base target is taken from the first block of the period; otherwise it is taken from the previous block.
6. Set:

$$T' = \frac{baseTarget \cdot \Delta}{\texttt{TARGET\_TIMESPAN}}.$$

7. If T' exceeds the network proof-of-work limit, use the proof-of-work limit.
8. Return $\mathrm{C}(T')$.

On no-retarget networks, the function returns the previous block's `nBits`. On networks that allow special minimum-difficulty blocks, the function may return the proof-of-work limit when the candidate timestamp is more than $2 \cdot$ `TARGET_SPACING` after the previous block; otherwise it searches backward for the most recent non-special difficulty in the current adjustment period [6].

8.4 Accumulated work

Nodes compare competing valid chains by accumulated work rather than by block count. This follows the proof-of-work security model described in the Bitcoin whitepaper [19].

For each accepted header, the node adds the work implied by that header's target to the predecessor's accumulated work. Headers and blocks with invalid proof of work are rejected before they can contribute to active-chain selection.

For a valid compact target, Bitcoin Core computes the work represented by a header as:

$$work(c) = \left\lfloor \frac{2^{256} - \mathrm{C}^{-1}(c) - 1}{\mathrm{C}^{-1}(c) + 1} \right\rfloor + 1.$$

This is algebraically equivalent to $\lfloor 2^{256}/(\mathrm{C}^{-1}(c)+1) \rfloor$ while avoiding construction of the unrepresentable integer 2^{256} in a 256-bit integer type [6]. The active chain is selected from valid candidates by greatest accumulated work, not by height.

9 Consensus Validation

Consensus validation determines whether a node may accept a chain history as valid. It instantiates the transition relations from Section 4 for headers, block data, active-chain connection, and reorganization.

9.1 Validation relation

Let N be the network parameter set. For candidate block b, candidate height h, candidate block time t, active chain C, and UTXO set U, let $R = \mathrm{Rules}(N, C, h, t)$. A successful connection produces:

$$(C, U) \xrightarrow{b} (C', U')$$

where C' extends or reorganizes C to include b, and U' is the UTXO set obtained after applying every transaction in the connected branch. If any consensus check fails, the active pair (C, U) is unchanged.

9.2 Header acceptance

A header is accepted into the header graph only if all conditions below hold. Header acceptance does not imply that block transaction data is known or valid.

Condition	Required rule
Uniqueness	If $\mathrm{bh}(h)$ is already known, the existing header state is reused.
Previous header	Every non-genesis header references a known previous header.
Proof of work	$\mathrm{le}(\mathrm{bh}(h)) \leq \mathrm{C}^{-1}(h.\texttt{bits})$.
Target	$\mathrm{C}^{-1}(h.\texttt{bits})$ is valid for the height, network, and difficulty-adjustment interval.
Timestamp	The header time is greater than the median timestamp of the previous block and up to ten ancestors. A header whose time is more than `MAX_FUTURE_BLOCK_TIME` after the node's current clock time is future-dated and not eligible for active validation until time advances.
BIP94 timewarp	On networks enforcing BIP94, a difficulty-adjustment block other than genesis has time at least `previous.time - BIP94_TIMEWARP_GRACE`.

Rule context	R permits the candidate height, time, version field, and network-specific activation state.
Invalid ancestry	A header descending from a known-invalid header is not eligible for best-chain selection.

For each accepted header, the node records height, predecessor, $\mathrm{C}^{-1}(h.\texttt{bits})$, $\mathrm{bh}(h)$, and accumulated work. Accumulated work is the sum of the work represented by each header from genesis through that header.

9.3 Block-data acceptance

Full block data is accepted only after the corresponding header is accepted. The following checks are performed before block data can become a candidate for active-chain connection:

Check	Required rule
Block form	The block has at least one transaction, satisfies the stripped-size, transaction-count, and weight limits in Section 7, and the first transaction is the only coinbase transaction.
Merkle root	Header `merkleRoot` equals $\mathrm{MR}(L)$ for the block's txid leaves, and the mutated-root condition is rejected.
Transaction form	Every transaction passes context-free transaction checks.
Legacy sigops limit	`legacySigOps(block) * WITNESS_SCALE_FACTOR <= MAX_BLOCK_SIGOPS_COST` before P2SH and witness sigops are counted during connection.
Contextual finality	Every transaction satisfies AbsFinal for the candidate block height and locktime cutoff.
Coinbase height	If R includes BIP34, the coinbase script begins with the candidate height.
Witness commitment	If R includes SegWit and the block has witness data, the coinbase witness commitment is present and correct.
Weight	The block weight is at most `MAX_BLOCK_WEIGHT`.

9.4 Best-chain selection

Among branches whose headers are valid and whose required block data is available, the selected branch is the valid branch with greatest accumulated work. A candidate branch is not eligible while any block on the path from the active fork to the candidate tip is missing block data or is known invalid.

1. Choose the highest-work eligible candidate tip.
2. Find the fork point with the current active chain.
3. Disconnect active-chain blocks back to the fork.
4. Connect candidate-branch blocks forward in height order.
5. If a candidate block is consensus-invalid, exclude that block and its descendants from future selection and retry with the next candidate.

9.5 Block connection

Connecting a non-genesis block applies the block's transactions to a temporary UTXO view whose best block is the previous block hash. The view becomes active only after all checks below pass.

Rule	Required behavior
BIP30 no-overwrite	A transaction output must not overwrite an existing unspent outpoint except for the historical BIP30 exceptions and buried rules that make the check unnecessary after BIP34.
Input availability	Every non-coinbase input references an unspent coin in the candidate view.
Duplicate spends	A transaction must not spend the same outpoint more than once.
Coinbase maturity	Coinbase-created outputs may be spent only after `COINBASE_MATURITY` confirmations.
Sequence locks	Every non-coinbase transaction satisfies the BIP68 relative-lock predicate from Section 5.
Value range	Every output and every sum used during validation remains in the money range.
Value conservation	For each non-coinbase transaction, input value is at least output value; the difference is the transaction fee.
Script validity	Each input satisfies the previous output's locking script under the script flags active for the block.
Sigops cost	Total block signature-operation cost is at most `MAX_BLOCK_SIGOPS_COST`.
Subsidy	Coinbase output value is at most block subsidy plus total fees.

The genesis block is special: its coinbase transaction is part of the block history but is not connected as a spendable coin by normal block-connection rules.

9.6 UTXO transition

For a valid non-coinbase transaction tx, let $I(tx)$ be its input outpoints and $O(tx)$ be its outputs indexed by output number. The candidate UTXO set is updated as follows:

$$U_{after} = (U_{before} \setminus I(tx)) \cup \{((txid(tx), n), coin(out_n, height, false)) : out_n \in O(tx)\}.$$

For the coinbase transaction, there are no spent inputs, and created outputs are inserted with `coinbase = true`. Outputs whose value is zero are still outputs and are inserted unless another consensus rule rejects the transaction.

9.7 Rule context

Rules(N, C, h, t) determines the script flags and contextual block predicates used for connection. Mainnet consensus rule context includes the buried and deployed rules for P2SH, strict DER

signatures, locktime and sequence verification, SegWit, Taproot, and Tapscript when included in R [2, 22, 28, 10, 8, 13, 17, 14, 15, 31, 32].

Policy-only flags may be stricter for mempool relay, but policy failure is not block invalidity.

9.8 Subsidy and fees

The block subsidy is determined by height and network parameters. For mainnet, the subsidy starts at 50 BTC and halves every 210,000 blocks until it reaches zero by integer right shift [27]. The consensus rule for a connected block is:

$$\sum outputs(coinbase) \leq subsidy(height) + \sum fees(noncoinbase).$$

Fees are computed from connected inputs and outputs. A transaction whose output sum exceeds its input sum is invalid.

9.9 Undo

Before applying a non-coinbase transaction, a node records the spent coin for each input in input order. The undo record for a block is the ordered sequence of those per-transaction records, excluding the coinbase. Undo is sufficient to restore the previous UTXO set when the block is disconnected; the byte encoding of undo data is not a consensus rule.

9.10 Reorganization correctness

A reorganization from old tip o to new tip n is valid only if:

1. the branch ending at n has greater accumulated work than the current active branch or is otherwise selected by the same greatest-work rule;
2. every disconnected block is reversed exactly using the UTXO state that existed after that block was connected;
3. every connected block on the new branch passes normal block connection; and
4. after the transition, the active UTXO set equals the result of applying the active chain from genesis to n.

Transactions removed from disconnected blocks reenter relay only under current mempool policy.

10 Mempool

The mempool is local, non-consensus state. It is a node's current set of unconfirmed transactions that the node is willing to keep, relay, and consider for block-template construction. A transaction can be valid in a block and absent from a node's mempool.

Let U be the active UTXO set. A mempool is:

$$Q = (M, G, I),$$

where M is a finite map from transaction identifier to mempool entry, G is the dependency graph induced by unconfirmed spends, and I is the set of indexes needed to query entries by transaction identifier, witness transaction identifier, spent outpoint, created outpoint, feerate, and entry time. In equations, $t \in M$ ranges over transactions in the entry map, and $Q \cup \{tx\}$ denotes the mempool obtained by adding tx to M and recomputing G and I. For validation against mempool ancestors, define:

$$\mathrm{View}(Q, U) = U \cup \bigcup_{t \in M} \mathrm{Out}(t).$$

10.1 Entry

Field	Type	Meaning
`tx`	`Transaction`	Serialized transaction accepted into the mempool.
`txid`, `wtxid`	`uint256`	Transaction identifiers from Section 5.
`fee`	`int64`	Input value minus output value under $\mathrm{View}(Q, U)$.
`weight`, `vsize`	`int64`	Weight and virtual size from Section 3.
`sigopsCost`	`int64`	Signature-operation cost under the selected script rules.
`spends`	OutPoint set	Previous outputs consumed by `tx`.
`creates`	OutPoint set	Outputs created by `tx`.
`parents`	Txid set	Mempool transactions directly spent by `tx`.
`children`	Txid set	Mempool transactions that directly spend `tx`.
`entryTime`, `entryHeight`	local clock, height	Admission time and active-chain height at admission.

The entry fields are logical fields: an implementation may derive, cache, or index them differently, but externally observable mempool behavior must be equivalent.

10.2 Graph Invariants

For a transaction t, let $\mathrm{In}(t)$ be its spent outpoints and $\mathrm{Out}(t)$ be its created outpoints. Direct mempool dependencies are:

$$\mathrm{parents}_Q(t) = \{u \in M : \mathrm{In}(t) \cap \mathrm{Out}(u) \neq \emptyset\},$$
$$\mathrm{children}_Q(t) = \{u \in M : \mathrm{In}(u) \cap \mathrm{Out}(t) \neq \emptyset\}.$$

A mempool is well-formed relative to U iff:

$$\mathrm{TxAvailable}_Q(t, U) \iff \forall x \in \mathrm{In}(t) : x \in U \cup \bigcup_{u \in M \setminus \{t\}} \mathrm{Out}(u).$$

$$\begin{aligned}\mathrm{WellFormed}(Q,U) \iff \forall t \in M : \neg\,\mathrm{Coinbase}(t) \wedge \mathrm{ContextFreeValid}(t) \wedge \mathrm{TxAvailable}_Q(t,U) \wedge \\ \forall a,b \in M,\ a \neq b : \mathrm{In}(a) \cap \mathrm{In}(b) = \emptyset \wedge \\ G = \{(u,t) : u \in \mathrm{parents}_Q(t)\} \wedge \mathrm{Acyclic}(G).\end{aligned}$$

The graph order is the spend order: a parent precedes every child. Any block template or package derived from Q must include transactions in a topological order of G.

10.3 Admission

Admission is a local transition:

$$Q \xrightarrow[P]{tx,U,H} Q',$$

where P is the node's local policy parameter set and H is the next-block height and median-time context. Write $H = (C,h,t)$, where C is the active chain, h is the next block height, and t is the next block locktime cutoff. Let $V_Q = \mathrm{View}(Q,U)$. A single transaction admission succeeds iff:

$$\begin{aligned}\mathrm{Admit}_P(Q,U,H,tx) \iff tx \notin M \wedge \mathrm{WellFormed}(Q \cup \{tx\},U) \wedge \\ \mathrm{ConsensusValid}(V_Q,tx,H) \wedge \mathrm{AbsFinal}(tx,h,t) \wedge \mathrm{SeqFinal}(tx,C,V_Q,h) \wedge \\ \mathrm{Policy}_P(Q,U,tx,H), \\ Q' = Q \cup \{tx\} \iff \mathrm{Admit}_P(Q,U,H,tx).\end{aligned}$$

Policy_P includes node-local standardness, feerate, resource, replacement, package, and denial-of-service rules. Those rules do not define consensus validity; when they are specified, they belong with the transaction, script, package, or relay behavior they constrain.

A package π is admitted atomically by applying the same transition to a topologically sorted sequence of transactions. If any member fails, the package transition fails and Q is unchanged.

10.4 Removal

Let B be a connected block and U' the UTXO set after connecting it. Let H' be the next-block context after connecting B. Connecting B removes every included transaction and every mempool transaction that conflicts with the block:

$$\begin{aligned}\mathrm{ConnectBlock}(Q,B,U') = \{t \in M : t \notin B.\texttt{txs} \wedge \\ \mathrm{In}(t) \cap \mathrm{Spent}(B) = \emptyset \wedge \\ \mathrm{ConsensusValid}(\mathrm{View}(Q,U'),t,H')\}.\end{aligned}$$

After removal, all entry indexes and dependency edges are recomputed so that $\mathrm{WellFormed}(Q,U')$ holds.

Disconnecting a block does not automatically restore its transactions to the mempool. The disconnected transactions become admission candidates and may reenter only through the current admission relation under the post-disconnect chain state.

10.5 Orphans and Relay View

A transaction with unavailable inputs is not a mempool entry. A node may keep an orphan cache O of such transactions and retry admission when parents become available, but O is local resource state and has no consensus meaning.

Transaction relay announces transactions from Q by `txid` or `wtxid`, serves requested transactions if still present, and may filter, delay, or suppress announcements according to local policy. Relay choices do not change the definition of Q [6].

11 Peer-to-Peer Network Protocol

The P2P protocol transports blocks, transactions, headers, addresses, filters, and relay-control messages between nodes. This section specifies interoperable wire behavior.

11.1 Message envelope

Legacy unencrypted Bitcoin P2P messages use a 24-byte header followed by a payload:

Field	Type	Meaning
`start`	`byte[4]`	Network magic identifying mainnet, testnet, signet, or regtest.
`command`	`byte[12]`	ASCII command name padded with zero bytes.
`payloadSize`	`uint32`	Number of payload bytes.
`checksum`	`byte[4]`	First four bytes of H(*payload*).
`payload`	`byte[]`	Command-specific serialized payload of length `payloadSize`.

BIP324 defines an encrypted v2 transport that changes transport framing after negotiation, but command payload semantics remain command-specific [29, 6].

11.2 Handshake

A connection becomes usable after version negotiation:

1. each side sends `version`;
2. each side validates network, protocol version, services, timestamp, nonce, and user-agent limits;
3. each side sends `verack`; and
4. optional relay, compact-block, address-relay, filter, and transport features are negotiated by later messages.

`version` field	Type	Meaning
`version`	`int32`	Protocol version offered by the sender.
`services`	`uint64`	Service bits offered by the sender.
`timestamp`	`int64`	Sender Unix time in seconds.

addrRecv	network address	Address of the receiving peer as understood by the sender.
addrFrom	network address	Address of the sender.
nonce	uint64	Random connection nonce used to detect self-connections.
userAgent	string	BIP14 user-agent string.
startHeight	int32	Sender's current best height estimate.
relay	bool, optional	BIP37 transaction-inventory relay preference when present.

11.3 Control and negotiation payloads

Message	Payload	Meaning
verack	empty	Completes version negotiation.
sendheaders	empty	Requests direct headers announcements instead of block inventory announcements.
wtxidrelay	empty	Negotiates transaction relay by witness transaction identifier.
sendaddrv2	empty	Negotiates BIP155 address relay.
ping	uint64 nonce	Liveness probe.
pong	uint64 nonce	Response echoing the received ping nonce.
feefilter	uint64 feerate	Asks the peer not to announce transactions below the feerate in satoshis per 1000 virtual bytes.

11.4 Service flags

Flag	Bit	Meaning
NODE_NETWORK	0	Peer can serve the full block chain.
NODE_BLOOM	2	Peer supports BIP37 bloom-filter service.
NODE_WITNESS	3	Peer supports witness transaction and block relay.
NODE_COMPACT_FILTERS	6	Peer supports BIP157 compact block filter service.
NODE_NETWORK_LIMITED	10	Peer can serve at least a limited recent block window.

Service flags advertise capability. A node must still validate responses and apply local resource limits.

11.5 Inventory and object retrieval

Inventory relay announces objects by type and hash. A peer requests announced objects with getdata; the responder sends the corresponding object or a notfound response when unavailable.

Record	Fields	Meaning
InventoryVector	type:uint32, hash:uint256	Identifies a transaction, block, filtered block, compact block, witness transaction, or witness block.
inv	vector<InventoryVector>	Announces available objects.
getdata	vector<InventoryVector>	Requests announced objects.
notfound	vector<InventoryVector>	Reports requested objects not available from the responder.

Witness inventory types request serialization that includes witness data. A peer that has not negotiated witness support must not rely on receiving witness objects from that connection [18].

Inventory type	Value	Object requested or announced
MSG_TX	1	Transaction by txid.
MSG_BLOCK	2	Block without requiring witness serialization.
MSG_FILTERED_BLOCK	3	BIP37 filtered block.
MSG_CMPCT_BLOCK	4	BIP152 compact block.
MSG_WITNESS_TX	$2^{30} + 1$	Transaction including witness data, identified by txid request type with witness flag.
MSG_WITNESS_BLOCK	$2^{30} + 2$	Block including witness data.
MSG_FILTERED_WITNESS_BLOCK	$2^{30} + 3$	Filtered block including witness data.

11.6 Header and block synchronization

Headers synchronize chain work before full block data. A locator is a sequence of block hashes, newest first, stepping back from a known tip toward genesis.

Messages getheaders and getblocks carry a locator and stop hash. Message headers returns block headers in forward order, each followed by CompactSize zero. Message block carries serialized full block data. A node must validate proof of work, linkage, target, timestamp, and deployment context for received headers before using them for chain selection.

Record	Fields	Encoding rule
BlockLocator	version, hashes	int32 version followed by vector<uint256> locator hashes.
getheaders/ getblocks	version, locator, stopHash	int32, locator vector, then uint256 stop hash.

`headers` item	`header`, `txCount`	80-byte block header followed by CompactSize zero. A nonzero count is invalid for `headers`.

11.7 Compact block relay

BIP152 compact blocks reduce block-relay bandwidth by sending a header, short transaction identifiers, selected prefilled transactions, and requesting any missing transactions [9].

Message `sendcmpct` negotiates compact-block mode and announcement preference. Messages `cmpctblock`, `getblocktxn`, and `blocktxn` carry the compact block and missing-transaction exchange. Compact reconstruction is an optimization only. The reconstructed full block must pass the same block validation rules as any received `block` message.

Record	Fields	Encoding rule
`Compact Block`	`header`, `nonce`, `shortIds`, `prefilledTx`	Header, `uint64` nonce, CompactSize count plus 6-byte short ids, then `vector<PrefilledTx>`.
`PrefilledTx`	`indexDelta`, `tx`	CompactSize differential index from the previous prefilled transaction, then full transaction.
`getblocktxn`	`blockHash`, `indexes`	Block hash followed by CompactSize differential transaction indexes.
`blocktxn`	`blockHash`, `transactions`	Block hash followed by transaction vector.

11.8 Address relay

Legacy address records and BIP155 address records use different encodings:

Record	Fields	Encoding rule
`NetAddress NoTime`	`services`, `address`, `port`	`uint64` services, 16 address bytes, then `uint16` port in big-endian network byte order. Used inside `version`.
`NetAddress Legacy`	`time`, `services`, `address`, `port`	`uint32` time plus `NetAddressNoTime`. Used by `addr`.
`AddrV2`	`time`, `services`, `networkId`, `address`, `port`	`uint32` time, CompactSize services, `uint8` network id, CompactSize address length, address bytes, then big-endian port.

Messages `addr` and `addrv2` relay legacy and BIP155 address records. Message `sendaddrv2` announces BIP155 address-relay support, and `getaddr` requests known peer addresses subject to local rate limits. Address relay is unauthenticated gossip. Received addresses are hints, not validated reachability claims [23].

11.9 Light-client services

Bitcoin P2P includes two optional light-client service families:

Service	Messages	Status
BIP37 bloom filters	`filterload`, `filteradd`, `filterclear`, `merkleblock`	Optional peer service; disabled unless a node chooses to provide bloom-filter support.
BIP157/158 compact filters	`getcfilters`, `cfilter`, `getcfheaders`, `cfheaders`, `getcfcheckpt`, `cfcheckpt`	Optional compact-filter service for clients that verify filters against block-filter headers.

These services do not change block or transaction validity [11, 20, 21].

Message	Payload fields	Encoding rule
`filterload`	`filter`, `hashFuncs`, `tweak`, `flags`	`varbytes`, `uint32`, `uint32`, `uint8`.
`filteradd`	`data`	`varbytes`.
`filterclear`	empty	Clears the peer's BIP37 filter state.
`merkleblock`	`header`, `total`, `hashes`, `flags`	Block header, `uint32` total transactions, `vector<uint256>` hashes, and flag bytes.
`getcfilters`	`type`, `startHeight`, `stopHash`	`uint8`, `uint32`, `uint256`.
`cfilter`	`type`, `blockHash`, `filter`	`uint8`, `uint256`, `varbytes`.
`getcfheaders`	`type`, `startHeight`, `stopHash`	`uint8`, `uint32`, `uint256`.
`cfheaders`	`type`, `stopHash`, `prevHeader`, `filterHashes`	`uint8`, two `uint256` values, then `vector<uint256>`.
`getcfcheckpt`	`type`, `stopHash`	`uint8`, `uint256`.
`cfcheckpt`	`type`, `stopHash`, `headers`	`uint8`, `uint256`, `vector<uint256>`.

12 Acknowledgements

Bitcoin protocol was invented by Satoshi Nakamoto and has been advanced, maintained, and secured by generations of Bitcoin Core developers, reviewers, maintainers, researchers, miners, node operators, and users; we thank them for their contributions. This specification is a derivative reference work over their protocol design, implementation, review, and maintenance.

Appendix A. Constants

Constant	Value
COIN	100,000,000 satoshis
MAX_MONEY	21,000,000 · COIN
WITNESS_SCALE_FACTOR	4
MAX_BLOCK_WEIGHT	4,000,000 weight units
MAX_BLOCK_SIGOPS_COST	80,000
COINBASE_MATURITY	100 blocks
SUBSIDY_HALVING_INTERVAL	210,000 blocks
NULL_TXID	0^{256}
NULL_INDEX	0xffffffff
COINBASE_SCRIPTSIG_MIN	2 bytes
COINBASE_SCRIPTSIG_MAX	100 bytes
LOCKTIME_THRESHOLD	500,000,000
SEQUENCE_FINAL	0xffffffff
SEQUENCE_LOCKTIME_DISABLE_FLAG	$1 \ll 31$
SEQUENCE_LOCKTIME_TYPE_FLAG	$1 \ll 22$
SEQUENCE_LOCKTIME_MASK	0x0000ffff
SEQUENCE_LOCKTIME_GRANULARITY	512 seconds
MAX_SCRIPT_ELEMENT_SIZE	520 bytes
MAX_SCRIPT_SIZE	10,000 bytes
MAX_OPS_PER_SCRIPT	201
MAX_STACK_SIZE	1,000 items
MAX_SCRIPT_NUM_SIZE	4 bytes
MAX_LOCKTIME_NUM_SIZE	5 bytes
MAX_PUBKEYS_PER_MULTISIG	20
TAPSCRIPT_SIGOPS_BUDGET_BASE	50
TAPSCRIPT_SIGOPS_COST	50
TARGET_SPACING	600 seconds
TARGET_TIMESPAN	1,209,600 seconds
DIFFICULTY_ADJUSTMENT_INTERVAL	2,016 blocks
MAX_RETARGET_FACTOR	4
MAX_FUTURE_BLOCK_TIME	7,200 seconds
BIP94_TIMEWARP_GRACE	600 seconds

These constants are specified by deployed BIPs and Bitcoin Core consensus [10, 12, 8, 13, 17, 32, 6].

Appendix B. Mainnet Parameters

This appendix collects network parameters and historical consensus exceptions that are inputs to Rules(N, C, h, t). Mainnet is the default network for this specification.

B.1 Mainnet parameters

Parameter	Value
Genesis hash	`000000000019d6689c085ae165831e93 4ff763ae46a2a6c172b3f1b60a8ce26f`
Genesis Merkle root	`4a5e1e4baab89f3a32518a88c31bc87f 618f76673e2cc77ab2127b7afdeda33b`
Message start	`f9 be b4 d9`
Subsidy halving interval	`SUBSIDY_HALVING_INTERVAL`
Proof-of-work target spacing	`TARGET_SPACING`
Proof-of-work target timespan	`TARGET_TIMESPAN`
Bech32 human-readable part	`bc`

Other networks, including testnet, testnet4, signet, and regtest, have distinct genesis blocks, proof-of-work limits, retargeting exceptions, message-start bytes, ports, address prefixes, and activation parameters. Their rules are not inferred from mainnet.

B.2 Historical exceptions

Exception	Rule
Genesis coinbase	The genesis block's coinbase transaction is part of the block history but its output is not inserted into U by normal block connection.
BIP30 duplicates	Historical duplicate-transaction cases are handled by the BIP30 no-overwrite rule and its deployed exceptions. Outside those exceptions, creating an output whose outpoint is already unspent is invalid.
Buried activations	Mainnet rules whose activation heights are buried in contemporary software are represented by Rules(N, C, h, t), not by live versionbits state.
Future-dated blocks	A block whose timestamp is more than `MAX_FUTURE_BLOCK_TIME` after the node's current clock is temporarily ineligible for active validation, not permanently invalid.

Signet uses a block signature challenge in the witness commitment payload. Testnet4 and regtest can enforce BIP94 timewarp mitigation through network parameters; mainnet does not set that parameter in the Bitcoin Core 31.0 baseline [26, 3, 12, 1, 6].

Appendix C. Remote Procedure Calls (RPC)

TODO

References

[1] Karl-Johan Alm and Anthony Towns. Bip 325: Signet. https://github.com/bitcoin/bips/blob/master/bip-0325.mediawiki.

[2] Gavin Andresen. Bip 16: Pay to script hash. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki.

[3] Gavin Andresen. Bip 34: Block v2, height in coinbase. https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki.

[4] Bitcoin contributors. Bitcoin improvement proposals. https://github.com/bitcoin/bips.

[5] Bitcoin Core contributors. Bitcoin core source repository, 2009. https://github.com/bitcoin/bitcoin.

[6] Bitcoin Core contributors. Bitcoin core v31.0 source tree, 2026. https://github.com/bitcoin/bitcoin/tree/v31.0.

[7] Bitcoin.org contributors. Bitcoin developer reference. https://developer.bitcoin.org/reference/.

[8] BtcDrak, Mark Friedenbach, and Eric Lombrozo. Bip 112: Checksequenceverify. https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki.

[9] Matt Corallo. Bip 152: Compact block relay. https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki.

[10] Mark Friedenbach, BtcDrak, Nicolas Dorier, and Kalle Rosenbaum. Bip 68: Relative lock-time using consensus-enforced sequence numbers. https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki.

[11] Mike Hearn and Matt Corallo. Bip 37: Connection bloom filtering. https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki.

[12] Fabian Jahr. Bip 94: Testnet 4. https://github.com/bitcoin/bips/blob/master/bip-0094.mediawiki.

[13] Thomas Kerin and Mark Friedenbach. Bip 113: Median time-past as endpoint for lock-time calculations. https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki.

[14] Johnson Lau. Bip 143: Transaction signature verification for version 0 witness program. https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki.

[15] Johnson Lau. Bip 147: Dealing with dummy stack element malleability. https://github.com/bitcoin/bips/blob/master/bip-0147.mediawiki.

[16] Eric Lombrozo. Bip 123: Bip classification. https://github.com/bitcoin/bips/blob/master/bip-0123.mediawiki.

[17] Eric Lombrozo, Johnson Lau, and Pieter Wuille. Bip 141: Segregated witness. https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[18] Eric Lombrozo, Johnson Lau, and Pieter Wuille. Bip 144: Segregated witness (peer services). https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.

[20] Olaoluwa Osuntokun and Alex Akselrod. Bip 157: Client side block filtering. https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki.

[21] Olaoluwa Osuntokun and Alex Akselrod. Bip 158: Compact block filters for light clients. https://github.com/bitcoin/bips/blob/master/bip-0158.mediawiki.

[22] Peter Todd. Bip 65: Op_checklocktimeverify. https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki.

[23] Wladimir J. van der Laan. Bip 155: addrv2 message. https://github.com/bitcoin/bips/blob/master/bip-0155.mediawiki.

[24] Gavin Wood. The jam specification (graypaper source repository), 2024. https://github.com/gavofyork/graypaper.

[25] Gavin Wood and Ethereum contributors. Ethereum yellow paper. https://github.com/ethereum/yellowpaper.

[26] Pieter Wuille. Bip 30: Duplicate transactions. https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki.

[27] Pieter Wuille. Bip 42: A finite monetary supply for bitcoin. https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki.

[28] Pieter Wuille. Bip 66: Strict der signatures. https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki.

[29] Pieter Wuille, Dhruv Mehta, Tim Ruffing, and Jonas Nick. Bip 324: Version 2 p2p encrypted transport protocol. https://github.com/bitcoin/bips/blob/master/bip-0324.mediawiki.

[30] Pieter Wuille, Jonas Nick, and Tim Ruffing. Bip 340: Schnorr signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

[31] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 341: Taproot: Segwit version 1 spending rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

[32] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 342: Validation of taproot scripts. https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki.

[33] Zcash contributors. Zcash protocol specification. https://zips.z.cash/protocol/protocol.pdf.