3.2-Design.md

Customize Blockchain

Customizations

To customize our blockchain, we are going to make the following changes:

Branding

- Rename the project
- Change address prefixes

Networking

- Message prefix bytes
- RPC and P2P ports
- Seeds

Consensus Rules

- Coin Distribution
  - Max supply
  - Block rewards
  - Subsidy halving interval
- Block size
- Block time
- Activate BIPs
- Checkpoint data
- Genesis block
- Difficulty retargeting interval

Standards

- OP_RETURN data limit

Advanced Customizations

Coin Distribution

- Premine

Code

Branding

As an example we're going to call this project LearnCoin. Let's make it our own by making a few changes.

Rename the project

This is actually not trivial since there are thousands of references to Bitcoin in the source code, but we want the project to be called LearnCoin, so let's go through the exercise. To see how many references to Bitcoin there are, lets run:

```
$ find . -not -path "./.git/*" -type f | xargs grep -nrE "bitcoin|Bitcoin|BITCOIN" | wc -l # line count
23246
```

There are 23246 references to either bitcoin, Bitcoin, or BITCOIN that we need to replace with learncoin, Learncoin, and LEARNCOIN. In order to change them, we'll execute some small bash scripts:

```
$ sudo apt-get install rename -y
$ git clean -xdf # clean untracked, ignored build artifacts
$ sudo find -type f -not -path "./.git/*" -exec sed -i 's/bitcoin/learncoin/g' {} +
$ sudo find -type f -not -path "./.git/*" -exec sed -i 's/Bitcoin/Learncoin/g' {} +
$ sudo find -type f -not -path "./.git/*" -exec sed -i 's/BITCOIN/LEARNCOIN/g' {} +
$ sudo find . -iname "bitcoin*" -exec rename 's/bitcoin/learncoin/' '{}' \;
$ sudo find . -iname "*bitcoin*" -exec rename 's/bitcoin/learncoin/' '{}' \;
```

Now let's try to build

```
$ ./autogen.sh
$ ./configure
$ sudo make
```

Now, this was a bit overzealous because it is changing actual references to bitcoin urls as well as Bitcoin Core which we do not want to change, so we'll have to change those specific items back to make the documentation accurate. For now, though, we'll just fix the licensing and copyrights.

```
$ sudo find -type f -not -path "./.git/*" -exec sed -i 's/Learncoin Core developers/Bitcoin Core
developers/g' {} +
```

Change address prefixes

Finally, let's dive into the source code. Currently there are 3 standard address types in Bitcoin:

Pay to Pubkey Hash

- Mainnet prefix "1"
- Testnet/Regtest prefix "m or n"

Pay to Script Hash

- Mainnet prefix "3"
- Testnet/Regtest prefix "2"

Bech32 (Segwit)

- Mainnet prefix "bc1"
- Testnet prefix "tb1"
- Regtest prefix "bcrt1"

Since these addresses are base58check encoded (except for bech32 which is it's own custom encoding), we can set any prefix we want by using this helpful table of values found at https://en.bitcoin.it/wiki/List_of_address_prefixes. The source file containing these prefix settings is `src/chainparams.cpp` :

```
// Mainnet: Line 140
// ...
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,0);
```

```
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,5);
base58Prefixes[SECRET_KEY] =     std::vector<unsigned char>(1,128);
base58Prefixes[EXT_PUBLIC_KEY] = {0x04, 0x88, 0xB2, 0x1E};
base58Prefixes[EXT_SECRET_KEY] = {0x04, 0x88, 0xAD, 0xE4};

bech32_hrp = "bc";
// ...

// Testnet: Line 244
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,111);
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,196);
base58Prefixes[SECRET_KEY] =     std::vector<unsigned char>(1,239);
base58Prefixes[EXT_PUBLIC_KEY] = {0x04, 0x35, 0x87, 0xCF};
base58Prefixes[EXT_SECRET_KEY] = {0x04, 0x35, 0x83, 0x94};

bech32_hrp = "tb";
// ...

// Regtest: Line 344
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,111);
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,196);
base58Prefixes[SECRET_KEY] =     std::vector<unsigned char>(1,239);
base58Prefixes[EXT_PUBLIC_KEY] = {0x04, 0x35, 0x87, 0xCF};
base58Prefixes[EXT_SECRET_KEY] = {0x04, 0x35, 0x83, 0x94};

bech32_hrp = "bcrt";
...
```

We want our p2pkh addresses to start with the letter "K" on mainnet and "k" on testnet and regtest. Then we want p2psh addresses to start with "L" on mainnet and "t" on testnet and regtest. We'll keep secret keys the same, but we'll change the bech32 addresses to "lc", "tl", and "lcrt" for mainnet, testnet, and regtest respectively. Based on the values in the table linked above, our `chainparams.cpp` becomes:

```
// Mainnet: Line 140
// ...
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,45); // K
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,48); // L
// ...
bech32_hrp = "lc";
// ...

// Testnet: Line 244
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,107); // k
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,65);  // t
// ...
bech32_hrp = "tl";
// ...

// Regtest: Line 344
base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,107); // k
base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,65);  // t
// ...
bech32_hrp = "lcrt";
...
```

Networking

In order to avoid conflict with the Bitcoin network, we should customize our network parameters.

Message prefix bytes

Every message sent between nodes on each of the network begins with the 4 byte message prefixes defined in `chainparams.cpp`:

```
// Mainnet: Line 110
/**
 * The message start string is designed to be unlikely to occur in normal data.
 * The characters are rarely used upper ASCII, not valid as UTF-8, and produce
 * a large 32-bit integer with any alignment.
 */
pchMessageStart[0] = 0xf9;
pchMessageStart[1] = 0xbe;
pchMessageStart[2] = 0xb4;
pchMessageStart[3] = 0xd9;
// ...

// Testnet: Line 224
pchMessageStart[0] = 0x0b;
pchMessageStart[1] = 0x11;
pchMessageStart[2] = 0x09;
pchMessageStart[3] = 0x07;
// ...

// Regtest: Line 313
pchMessageStart[0] = 0xfa;
pchMessageStart[1] = 0xbf;
pchMessageStart[2] = 0xb5;
pchMessageStart[3] = 0xda;
```

We just want to pick random numbers that will be unlikely to occur in normal data. For reference, you can have a look at http://www.asciitable.com/ and https://www.utf8-chartable.de/unicode-utf8-table.pl for a list of used codes. The range that we want to use is the upper ASCII which is not valid UTF-8, this range is from `0x80` to `0xff`. So, let's generate some random bytes:

```
$ msgprefixgen
Mainnet
pchMessageStart[0] = 0xbb;
pchMessageStart[1] = 0xcd;
pchMessageStart[2] = 0xd7;
pchMessageStart[3] = 0xae;

Testnet
pchMessageStart[0] = 0xc0;
pchMessageStart[1] = 0xce;
pchMessageStart[2] = 0xeb;
pchMessageStart[3] = 0xad;

Regtest
pchMessageStart[0] = 0xd4;
pchMessageStart[1] = 0xf8;
pchMessageStart[2] = 0xbf;
pchMessageStart[3] = 0xdf;
```

We can then replace them with our new values generated above and rebuild (`$ sudo make`).

RPC and P2P ports

The rpc ports are defined in `src/chainparamsbase.cpp` and p2p ports are defined in `src/chainparams.cpp`.

```
// src/chainparamsbase.cpp Line 35
if (chain == CBaseChainParams::MAIN)
    return MakeUnique<CBaseChainParams>("", 8332);
else if (chain == CBaseChainParams::TESTNET)
    return MakeUnique<CBaseChainParams>("testnet3", 18332);
else if (chain == CBaseChainParams::REGTEST)
    return MakeUnique<CBaseChainParams>("regtest", 18443);
```

Let's change them to:

```
// src/chainparamsbase.cpp Line 35
if (chain == CBaseChainParams::MAIN)
    return MakeUnique<CBaseChainParams>("", 8432);
else if (chain == CBaseChainParams::TESTNET)
    return MakeUnique<CBaseChainParams>("testnet3", 18432);
else if (chain == CBaseChainParams::REGTEST)
    return MakeUnique<CBaseChainParams>("regtest", 18543);
```

Next, we will go back to `chainparams.cpp` to change the p2p ports:

```
// Mainnet Line 119
nDefaultPort = 8433;

// Testnet Line 228
nDefaultPort = 18433;

// Regtest Line 317
nDefaultPort = 18544;
```

Seeds

Seeds are the nodes that new nodes will connect to first when they are syncing. We won't have any of these when we start, so let's just comment them out for now.

```
// src/chainparams.cpp Mainnet: Line 132
//vSeeds.emplace_back("seed.learncoin.sipa.be"); // Pieter Wuille, only supports x1, x5, x9, and xd
//vSeeds.emplace_back("dnsseed.bluematt.me"); // Matt Corallo, only supports x9
//vSeeds.emplace_back("dnsseed.learncoin.dashjr.org"); // Luke Dashjr
//vSeeds.emplace_back("seed.learncoinstats.com"); // Christian Decker, supports x1 - xf
//vSeeds.emplace_back("seed.learncoin.jonasschnelli.ch"); // Jonas Schnelli, only supports x1, x5, x9,
and xd
//vSeeds.emplace_back("seed.btc.petertodd.org"); // Peter Todd, only supports x1, x5, x9, and xd
//vSeeds.emplace_back("seed.learncoin.sprovoost.nl"); // Sjors Provoost

// Testnet: Line 239
//vSeeds.emplace_back("testnet-seed.learncoin.jonasschnelli.ch");
//vSeeds.emplace_back("seed.tbtc.petertodd.org");
//vSeeds.emplace_back("seed.testnet.learncoin.sprovoost.nl");
//vSeeds.emplace_back("testnet-seed.bluematt.me"); // Just a static list of stable node(s), only
supports x9
```

Consensus Rules

The following changes are consensus-related, therefore if nodes have different consensus rules, they are incompatible blockchains. When nodes on an existing blockchain decide to change their consensus rules breaking past compatibility, it is called a hard fork. If the consensus changes are just more strict, but still backward-compatible, it is a soft fork. In our case, the blockchain is new, so there is no blockchain fork. Note, this is different than a software fork on github.

Coin Distribution

Max supply

Let's make the max supply a more round number, instead of 21M coins, let's do 10M for the sake of learning. There is no fixed max supply parameter that we can easily change. This is because of the distribution algorithm of Bitcoin. The total supply of bitcoin is designed to level off just below 21M coins, but the parameters that determine this are the block subsidy (miner reward) and the block halving interval. How long it takes to level off to the max supply is determined by the block time.

Bitcoin parameters:

- Initial block subsidy: 50BTC
- Block halving interval: 210,000 (blocks)

Let's do some quick math. If we want our max supply to be 10M coins, we could just change the initial mining subsidy. In order to find out what it will need to be, we can use this script:

```
$ distribution -m 10000000
Assumptions
==================
Target Max Time (months): 48
Halving Interval (blocks): 210000

Results
==================
Block Time (sec): 600
Initial Subsidy: 23.809523809523807
Halving Interval (blocks): 210000
Target Max Time (months): 48
Max Supply: 10000000
```

If we only provide the max supply parameter, the tool assumes default target max time and halving interval. If we make the initial subsidy equal to or above 23.81, we can be sure the max supply will never be greater than 10M Forkcoin. We're going to make a few more changes first before we edit the code.

Block rewards

Let's say we want the block rewards to be an even number, one that will be easy to remember, and easy to calculate in the future (after halvings). Let's go with 100 coins, and update our calculation:

```
$ distribution -m 10000000 -s 100
Results
==================
Block Time (sec): 600
Initial Subsidy: 100
Halving Interval (blocks): 50051.379461857425
Target Max Time (months): 48
Max Supply: 10000000
```

Then, we update this parameter in `src/validation.cpp`:

```
// Line 1169:
CAmount nSubsidy = 100 * COIN;
```

Subsidy halving interval

Now we have a subsidy halving interval of 50,050. This means after every time 50,050 blocks are mined, the block reward will be cut in half. Let's update this parameter in `src/chainparams.cpp`:

```
// Mainnet: Line 77
consensus.nSubsidyHalvingInterval = 50050;

// Testnet: Line 191
consensus.nSubsidyHalvingInterval = 50050;
```

For regtest, we will just leave it at 150.

Block time

If you also want to change the block time, you will find the parameters in `src/chainparams.cpp` :

```
// Mainnet: Line 85
consensus.nPowTargetSpacing = 10 * 60;

// Testnet: Line 199
consensus.nPowTargetSpacing = 10 * 60;

// Regtest: Line 292
consensus.nPowTargetSpacing = 10 * 60;
```

Every time the difficulty retargets, it will adjust based on the past block times whether to make the difficulty higher or lower in order to target the block time set by `nPowTargetSpacing` . Note changing the block time will affect how long it takes to reach the subsidy halving interval, and thus the supply will dimish proportionally.

Difficulty retarget time

By default this difficulty retargeting setting is every two weeks. Note that it is really after 2016 blocks, which is determined by how many blocks a 10 minute blocktime yields in two weeks. No matter how long it takes, it will always retarget after 2016 blocks by default. Let's say we want this to happen after 1 day. We can change it in `src/chainparams.cpp` :

```
// Mainnet: Line 84
consensus.nPowTargetTimespan = 24 * 60 * 60; // 1 day

// Testnet: Line 198
consensus.nPowTargetTimespan = 24 * 60 * 60; // 1 day

// Mainnet: Line 291
consensus.nPowTargetTimespan = 24 * 60 * 60; // 1 day
```

Block size

The block size has been an area of contentious debate. Bitcoin Cash was a major fork from Bitcoin that decided to scale transaction throughput by continuously increasing blocksize. Bitcoin Core decided it would be better to explore off-chain solutions to help scale in order to keep the system decentralized (larger blocks means more expensive hardware to run the software). If we want to change this setting, it is a little more nuanced than just number of bytes. This is because BIP141 introduced Segregated Witness.

Basically, this is just a way to separate some of the `scriptSig` from being a part of the transaction hash. Since there are some malleability problems (miners could pad signatures with zeros causing the transaction hash to change, but the signature would remain valid). To incentivize this, the block size will count witness data (the `scriptSig` data that is separated) as less than it's actual size in bytes. The formula is:

`Transaction Weight = Base transaction size * 3 + Total transaction size` , where

Base transaction size is the size of the transaction serialised with the witness data stripped.

Total transaction size is the transaction size in bytes serialized as described in BIP144, including base data and witness data.

The same method is used in calculating Block weight from Base size and Total size. Thus, the block size is not given in bytes, but in weight. This weight is set at 4000000 in `src/consensus/consensus.h` . This means if there is no witness data, the max is 1MB, but with witness data, blocks higher than 1MB are allowed.

```
// Line 13
/** The maximum allowed weight for a block, see BIP 141 (network rule) */
static const unsigned int MAX_BLOCK_WEIGHT = 4000000;
```

Activate BIPs

Ok, there were some past BIPS (Bitcoin Improvement Proposals) that had to get approval from the miners of the network to activate, and so there was an activation window set. Since we are past those dates, we need to activate them immediately in `src/chainparams.cpp` :

```
// Mainnet: Line 78
consensus.BIP16Exception = uint256();
consensus.BIP34Height = 0;
consensus.BIP34Hash = uint256();
consensus.BIP65Height = 0;
consensus.BIP66Height = 0;
// ...
// Mainnet: Line 96
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nStartTime = Consensus::BIP9Deployment::ALWAYS_ACTIVE;
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nTimeout = Consensus::BIP9Deployment::NO_TIMEOUT;

// Deployment of SegWit (BIP141, BIP143, and BIP147)
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nStartTime =
Consensus::BIP9Deployment::ALWAYS_ACTIVE;
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nTimeout = Consensus::BIP9Deployment::NO_TIMEOUT;


// Testnet: Line 192
consensus.BIP16Exception = uint256();
consensus.BIP34Height = 0;
consensus.BIP34Hash = uint256();
consensus.BIP65Height = 0;
consensus.BIP66Height = 0;
// ...
// Testnet: Line 210
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nStartTime = Consensus::BIP9Deployment::ALWAYS_ACTIVE;
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nTimeout = Consensus::BIP9Deployment::NO_TIMEOUT;

// Deployment of SegWit (BIP141, BIP143, and BIP147)
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].bit = 1;
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nStartTime =
Consensus::BIP9Deployment::ALWAYS_ACTIVE;
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nTimeout = Consensus::BIP9Deployment::NO_TIMEOUT;
```

We can leave the regtest as is.

Checkpoint data

When nodes first sync, there is some checkpoint data hard coded into the application, specific blocks that the application looks for. We have no blockchain data, so we want to clear these in `src/chainparams.cpp`

```
// Mainnet: Line 104
// The best chain should have at least this much work.
consensus.nMinimumChainWork = consensus.powLimit;

// By default assume that the signatures in ancestors of this block are valid.
consensus.defaultAssumeValid = uint256();
// ...
// Line 154
checkpointData = {
    {
    }
};

chainTxData = ChainTxData{
    // Data from rpc: getchaintxstats 4096
000000000000000002e63058c023a9a1de233554f28c7b21380b6c9003f36a8
    /* nTime    */ 0,//1532884444,
    /* nTxCount */ 0,//331282217,
    /* dTxRate  */ 0,//2.4
};
```

```
// Testnet: Line 205
// The best chain should have at least this much work.
consensus.nMinimumChainWork = consensus.powLimit;

// By default assume that the signatures in ancestors of this block are valid.
consensus.defaultAssumeValid = uint256();
/...
// Line 246
checkpointData = {
    {
    }
};

chainTxData = ChainTxData{
    // Data from rpc: getchaintxstats 4096
000000000000000037a8cd3e06cd5edbfe9dd1dbcc5dacab279376ef7cfc2b4c75
    /* nTime    */ 0,//1531929919,
    /* nTxCount */ 0,//19438708,
    /* dTxRate  */ 0,//0.626
};
```

Note the `chainTxData` is used for estimating sync progress.

Genesis block

Now, we are going to mine our own genesis block. The first block is called the genesis block, and it must be hard coded since it does not reference a previous block. It must also be mined because the proof-of-work checks still apply, though we may modify them for the genesis block in order to speed up the mining process. First, we must decide what will be in the genesis block.

- Message - "12-13-18 LinkedIn: 'Blockchain developer' is the fastest-growing U.S. job"
- Coinbase transaction (premine) - 1,000,000 LEARNCOIN
  - Pay to Public Key: We can generate a private key and get the uncompressed public key using `publickey.py` :

```
$ sha256 "Welcome to blockchain fundamentals"
2358feea3003a1d16af3454be4cec2f6a7db43bfc7daa101b8949fff91ed64b4
$ pubkey -u 2358feea3003a1d16af3454be4cec2f6a7db43bfc7daa101b8949fff91ed64b4
048f74dca316b3faa7e947919babe20e274d5c1f4cf3366652bd360bb51322f652b575fd0461fb982fd9aabf39c879db9f08a5f505b
```

- Timestamp (epoch time): `1544904235`
- `nBits = 0x1E7fffff` or `545259519` in decimal

Mining the genesis block

We know how to assemble a script, a transaction, and a block header, and we also know how to mine the block. For convenience, we can use the included `cpp_miner` program.

```
$ cpp_miner genesisgen
048f74dca316b3faa7e947919babe20e274d5c1f4cf3366652bd360bb51322f652b575fd0461fb982fd9aabf39c879db9f08a5f505b
"12-13-18 LinkedIn: 'Blockchain developer' is the fastest-growing U.S. job" 1000000 1544904235 545259519
0
block header:
010000000000000000000000000000000000000000000000000000000000000000000095ec912e86cb94934975e78d943728d6ef5
nonce: 0x00000006
block hash: 1918500b88eb211c30bf3ae7a5faa51305bace111344ab696efbd619fe99eb38
merkle root: 634c2897ab0decc26fce8dbedbdb5defd62837948de775499394cb862e91ec95
```

The parameters for the genesis block are found in `src/chainparams.cpp` . First we'll update the `nBits in the scriptSig`:

```
// Line 23
txNew.vin[0].scriptSig = CScript() << 545259519 << CScriptNum(4) << std::vector<unsigned char>((const
unsigned char*)pszTimestamp, (const unsigned char*)pszTimestamp + strlen(pszTimestamp));
```

Next, we'll update the message and public key:

```
// Line 51
const char* pszTimestamp = "12-13-18 LinkedIn: 'Blockchain developer' is the fastest-growing U.S. job";
const CScript genesisOutputScript = CScript() <<
ParseHex("048f74dca316b3faa7e947919babe20e274d5c1f4cf3366652bd360bb51322f652b575fd0461fb982fd9aabf39c879db9
<< OP_CHECKSIG;
```

Finally, we'll update the rest of the genesis block parameters for each network:

```
// Mainnet Line 122:
genesis = CreateGenesisBlock(1544904235, 0x00000006, 545259519, 1, 1000000 * COIN);
consensus.hashGenesisBlock = genesis.GetHash();
assert(consensus.hashGenesisBlock ==
uint256S("0x1918500b88eb211c30bf3ae7a5faa51305bace111344ab696efbd619fe99eb38"));
assert(genesis.hashMerkleRoot ==
uint256S("0x634c2897ab0decc26fce8dbedbdb5defd62837948de775499394cb862e91ec95"));

// Testnet Line 218:
// (same as mainnet)

// Regtest Line 306:
// (same as mainnet)
```

Allowing low difficulty genesis block

In order to mine the genesis block at a lower difficulty, we have to add exceptions in the block validation checks for the genesis block. We will also need to make the difficulty adjustment algorithm not take the difficulty of the genesis block into account.

`src/validation.cpp`

```
// Line 1093:
// Check the header
if (block.GetHash() != consensusParams.hashGenesisBlock)
{
    if (!CheckProofOfWork(block.GetHash(), block.nBits, consensusParams))
        return error("ReadBlockFromDisk: Errors in block header at %s", pos.ToString());
}

// Line 1829:
if (block.GetHash() != chainparams.GetConsensus().hashGenesisBlock)
{
    if (!CheckBlock(block, state, chainparams.GetConsensus(), !fJustCheck, !fJustCheck)) {
        if (state.CorruptionPossible()) {
            // We don't write down blocks to disk if they may have been
            // corrupted, so this should be impossible unless we're having hardware
            // problems.
            return AbortNode(state, "Corrupt block found indicating potential hardware failure; shutting
down");
        }
        return error("%s: Consensus::CheckBlock: %s", __func__, FormatStateMessage(state));
    }
}

// Line 3106:
if (block.GetHash() == consensusParams.hashGenesisBlock)
    fCheckPOW = false;
```

```
  if (!CheckBlockHeader(block, state, consensusParams, fCheckPOW))
      return false;
```

src/pow.cpp

```
  // Line 17:
  arith_uint256 lastTarget;

  // Immediately adjust to min. difficulty (genesis block was mined with very low difficulty)
  if (lastTarget.SetCompact(pindexLast->nBits) > UintToArith256(params.powLimit))
  {
      return nProofOfWorkLimit;
  }

  // Line 41:
  if (pindex->nHeight == 0) // If genesis block, return PoW limit
  {
      return nProofOfWorkLimit;
  }
  else
  {
      return pindex->nBits;
  }
```

Premine

In the original Bitcoin Core, there was a quirk with the genesis block which made it so the first transaction could not be spent. In order to maintain consensus, this quirk was left in the code. We want to be able to spend the genesis block so that we can premine some coins. In order to do this, we need to fix this quirk and allow the transaction to be spent.

src/validation.cpp

```
  // Line 1851: (comment this line out)
  // return true; Make genesis block spendable for premine

  // Line 1956:
  // Special case to allow genesis block to be spent
  if (block.GetHash() != chainparams.GetConsensus().hashGenesisBlock) {
      assert(pindex->pprev);
      CBlockIndex *pindexBIP34height = pindex->pprev->GetAncestor(chainparams.GetConsensus().BIP34Height);
      //Only continue to enforce if we're below BIP34 activation height or the block hash at that height
  doesn't correspond.
      fEnforceBIP30 = fEnforceBIP30 && (!pindexBIP34height || !(pindexBIP34height->GetBlockHash() ==
  chainparams.GetConsensus().BIP34Hash));
  }

  // Line 1980:
  if (block.GetHash() != chainparams.GetConsensus().hashGenesisBlock)
  {
      if (VersionBitsState(pindex->pprev, chainparams.GetConsensus(), Consensus::DEPLOYMENT_CSV,
  versionbitscache) == ThresholdState::ACTIVE) {
          nLockTimeFlags |= LOCKTIME_VERIFY_SEQUENCE;
      }
  }

  // Line 2067:
  if (block.vtx[0]->GetValueOut() > blockReward && block.GetHash() !=
  chainparams.GetConsensus().hashGenesisBlock)

  // Line 2081:
  // Skip for genesis block since it contains only a coinbase tx and
  // referencing a non-existent prevblock will cause a segfault
  if (block.GetHash() != chainparams.GetConsensus().hashGenesisBlock) {
      if (!WriteUndoDataForBlock(blockundo, state, pindex, chainparams))
```

```
            return false;
    }
```