Block hashing algorithm

Bitcoin mining uses the [hashcash](https://en.bitcoin.it/wiki/Hashcash" \o "Hashcash) proof of work function; the hashcash algorithm requires the following parameters: a service string, a nonce, and a counter. In bitcoin the service string is encoded in the block header data structure, and includes a version field, the hash of the previous block, the root hash of the merkle tree of all transactions in the block, the current time, and the difficulty. Bitcoin stores the nonce in the extraNonce field which is part of the coinbase transaction, which is stored as the left most leaf node in the merkle tree (the coinbase is the special first transaction in the block). The counter parameter is small at 32-bits so each time it wraps the extraNonce field must be incremented (or otherwise changed) to avoid repeating work. The basics of the hashcash algorithm are quite easy to understand and it is described in more detail [here](https://en.bitcoin.it/wiki/Hashcash). When mining bitcoin, the hashcash algorithm repeatedly hashes the block header while incrementing the counter & extraNonce fields. Incrementing the extraNonce field entails recomputing the merkle tree, as the coinbase transaction is the left most leaf node. The block is also occasionally updated as you are working on it.

A block header contains these fields:

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Purpose | Updated when... | Size (Bytes) |
| Version | Block version number | You upgrade the software and it specifies a new version | 4 |
| hashPrevBlock | 256-bit hash of the previous block header | A new block comes in | 32 |
| hashMerkleRoot | 256-bit hash based on all of the transactions in the block | A transaction is accepted | 32 |
| Time | Current timestamp as seconds since 1970-01-01T00:00 UTC | Every few seconds | 4 |
| Bits | Current target in compact format | The difficulty is adjusted | 4 |
| Nonce | 32-bit number (starts at 0) | A hash is tried (increments) | 4 |

The body of the block contains the transactions. These are hashed only indirectly through the Merkle root. Because transactions aren't hashed directly, hashing a block with 1 transaction takes exactly the same amount of effort as hashing a block with 10,000 transactions.

The compact format of target is a special kind of floating-point encoding using 3 bytes mantissa, the leading byte as exponent (where only the 5 lowest bits are used) and its base is 256. Most of these fields will be the same for all users. There might be some minor variation in the timestamps. The nonce will usually be different, but it increases in a strictly linear way. "Nonce" starts at 0 and is incremented for each hash. Whenever Nonce overflows (which it does frequently), the extraNonce portion of the generation transaction is incremented, which changes the Merkle root.

Moreover, it is extremely unlikely for two people to have the same Merkle root because the first transaction in your block is a generation "sent" to one of *your* unique Bitcoin addresses. Since your block is different from everyone else's blocks, you are (nearly) guaranteed to produce different hashes. Every hash you calculate has the same chance of winning as every other hash calculated by the network.

Bitcoin uses: SHA256(SHA256(Block\_Header)) but you have to be careful about byte-order.

For example, this python code will calculate the hash of the block with the smallest hash as of June 2011, [Block 125552](http://blockexplorer.com/block/00000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d). The header is built from the six fields described above, concatenated together as little-endian values in hex notation:

>>> **import** hashlib

>>> header\_hex = ("01000000" +

"81cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000" +

"e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b" +

"c7f5d74d" +

"f2b9441a" +

"42a14695")

>>> header\_bin = header\_hex.decode('hex')

>>> hash = hashlib.sha256(hashlib.sha256(header\_bin).digest()).digest()

>>> hash.encode('hex\_codec')

'1dbd981fe6985776b644b173a4d0385ddc1aa2a829688d1e0000000000000000'

>>> hash[::-1].encode('hex\_codec')

'00000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'

Endianess

Note that the hash, which is a 256-bit number, has lots of leading zero bytes when stored or printed as a big-endian hexadecimal constant, but it has trailing zero bytes when stored or printed in little-endian. For example, if interpreted as a string and the lowest (or start of) the string address keeps lowest significant byte, it is little-endian.

The output of [blockexplorer](http://blockexplorer.com/) displays the hash values as big-endian numbers; notation for numbers is usual (leading digits are the most significant digits read from left to right).

For another example, [here](http://pastebin.com/bW3fQA2a) is a version in plain C without any optimization, threading or error checking.

Here is the same example in plain PHP without any optimization.

**<?**

*//This reverses and then swaps every other char*

**function** SwapOrder($in){

$Split = str\_split(strrev($in));

$x='';

for ($i = 0; $i < count($Split); $i+=2) {

$x .= $Split[$i+1].$Split[$i];

}

return $x;

}

*//makes the littleEndian*

**function** littleEndian($value){

return implode (unpack('H\*',pack("V\*",$value)));

}

$version = littleEndian(1);

$prevBlockHash = SwapOrder('00000000000008a3a41b85b8b29ad444def299fee21793cd8b9e567eab02cd81');

$rootHash = SwapOrder('2b12fcf1b09288fcaff797d71e950e71ae42b91e8bdb2304758dfcffc2b620e3');

$time = littleEndian(1305998791);

$bits = littleEndian(440711666);

$nonce = littleEndian(2504433986);

*//concat it all*

$header\_hex = $version . $prevBlockHash . $rootHash . $time . $bits . $nonce;

*//convert from hex to binary*

$header\_bin = hex2bin($header\_hex);

*//hash it then convert from hex to binary*

$pass1 = hex2bin( hash('sha256', $header\_bin ) );

*//Hash it for the seconded time*

$pass2 = hash('sha256', $pass1);

*//fix the order*

$FinalHash = SwapOrder($pass2);

echo $FinalHash;

**?>**