Chunk Format

Revision as of 21:35, 18 July 2018 by Pokechu22 (talk | contribs) (→Direct: Fix typo)
(diff) ← Older revision | Latest revision (diff) | Newer revision → (diff)

This article describes in additional detail the format of the Chunk Data packet.

Contents

Concepts

Chunks columns and Chunk sections

You've probably heard the term "chunk" before. Minecraft uses chunks to store and transfer world data. However, there are actually 2 different concepts that are both called "chunks" in different contexts: chunk columns and chunk sections.

A **chunk column** is a 16×256×16 collection of blocks, and is what most players think of when they hear the term "chunk". However, these are not the smallest unit data is stored in in the game; chunk columns are actually 16 chunk sections aligned vertically.

Chunk columns store biomes, block entities, entities, tick data, and an array of sections.

A **chunk section** is a 16×16×16 collection of blocks (chunk sections are cubic). This is the actual area that blocks are stored in, and is often the concept Mojang refers to via "chunk". Breaking columns into sections wouldn't be useful, except that you don't need to send all chunk sections in a column: If a section is empty, then it doesn't need to be sent (more on this later).

Chunk sections store blocks and light data (both block light and sky light). Additionally, they can be associated with a section palette. A chunk section can contain at maximum 4096 (16×16×16, or 2^{12}) unique IDs (but, it is highly unlikely that such a section will occur in normal circumstances).

Chunk columns and chunk sections are both displayed when chunk border rendering is enabled (`F3+G`). Chunk columns borders are indicated via the red vertical lines, while chunk sections borders are indicated by the blue lines.

Empty sections and the primary bit mask

As previously mentioned, chunk sections can be **empty**. Sections which contain no useful data are treated as empty[concept note 1], and are not sent to the client, as the client is able to infer the contents[concept note 2]. For the average world, this means around 60% of the world's data doesn't need to be sent, since it's all air; this is a significant save.

It is important to note that a chunk composed entirely of empty sections is different from an empty (ie, unloaded) chunk column. When a block is changed in an empty section, the section is created (as all air), and the block is set. When a block is changed in an empty chunk, the behavior is undefined (but generally, nothing happens).

The **primary bit mask** simply determines which sections are being sent. The least significant bit is for the lowest section (y=0 to y=15). Only 16 bits can be set in it (with the 16th bit controlling the y=240 to y=255 section); sections above y=255 are not valid for the notchian client. To check whether a section is included, use `((mask & (1 << sectionY)) != 0)`.

Global and section palettes

Minecraft also uses palettes. A palette maps numeric IDs to block states. The concept is more commonly used with colors in an image; Wikipedia's articles on color look-up tables, indexed colors, and palettes in general may be helpful for fully grokking it.

There are 2 palettes that are used in the game: the global palette and the section palette.

The **global palette** is the standard mapping of IDs to block states. Currently, it is a combination Block ID and Metadata (`(blockId << 4) | metadata`). Note that thus, the global palette is not continuous[concept note 3]. Entries not defined within the global palette are treated as air (even if the block ID itself is known, if the metadata is not known, the state is treated as air). Note that the global palette is currently represented by 13 bits per entry[concept note 4], with the block ID (https://minecraft.wiki/w/Data_values%23Block_IDs) for the first 9 bits, and the block damage value for the last 4 bits. For example, Diorite (block ID `1` for `minecraft:stone` with damage 3) would be encoded as `000000001 0011`. If a block is not found in the global palette (either due to not having a valid damage value or due to not being a valid ID), it will be treated as air.

The basic implementation looks like this:

```
long getGlobalPaletteIDFromState(BlockState state) {
    if (state.isValid()) {
        return (state.getId() << 4) | state.getMetadata();
    } else {
        return 0;
    }
}

BlockState getStateFromGlobalPaletteID(long id) {
    int blockID = (id >> 4);
    byte metadata = (id & 0x0F);
    BlockState state = new BlockState(blockID, metadata);
    if (state.isValid()) {
        return state;
    } else {
        return new BlockState(0, 0);  // Air
    }
}
```

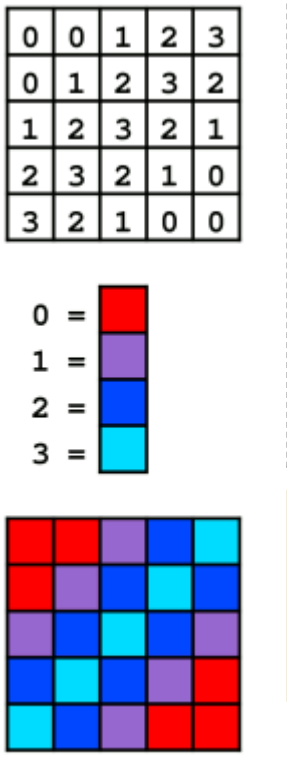

Illustration of an indexed palette (Source)

⚠ Don't assume that the global palette will always be like this; keep it in a separate function. Mojang has stated that they plan to change the global palette to avoid increasing the total size. Equally so, though, do not hardcode the total size of the palette; keep it in a constant.

A **section palette** is used to map IDs within a chunk section to global palette IDs. Other than skipping empty sections, correct use of the section palette is the biggest place where data can be saved. Given that most sections contain only a few blocks, using 13 bits to represent a chunk section that is only stone, gravel, and air would be extremely wasteful. Instead, a list of IDs are sent mapping indexes to global palette IDs (for instance, `0x10 0xD0 0x00`), and indexes within the section palette are used (so stone would be sent as `0`, gravel `1`, and air `2`)[concept note 5]. The number of bits per ID in the section palette varies from 4 to 8; if fewer than 4 bits would be needed it's increased to 4[concept note 6] and if more than 8 would be needed, the section palette is not used and instead global palette IDs are used[concept note 7].

⚠ Note that the notchian client (and server) store their chunk data within the compacted, paletted format. Sending non-compacted data not only wastes bandwidth, but also leads to increased memory use clientside; while this is OK for an initial implementation it is strongly encouraged that one compacts the block data as soon as possible.

Ground-up continuous

The **ground-up continuous** value (tentative name) is one of the more confusing properties of the chunk data packet, simply because there's no good name for it. It controls two different behaviors of the chunk data packet, one that most people need, and one that most don't.

When ground-up continuous is set, the chunk data packet is used to create a *new* chunk. This includes biome data, and all (non-empty) sections in the chunk. Sections not specified in the primary bit mask are empty sections.

⚠ Sending a packet with ground-up continuous enabled over a chunk that already exists will **leak memory** clientside.

Make sure to unload chunks before overwriting them with the Unload Chunk packet. That packet can always be sent even on unloaded chunks, so in situations where the chunk might or might not be loaded already, it's valid to send it again (but avoid sending it in excess).

The `MultiplayerChunkCache` values in F3 show the number of chunks in the client's 2 chunk storage mechanisms; if the numbers aren't equal, you've leaked chunks.

When ground-up continuous is not set, then the chunk data packet acts as a large Multi Block Change packet, changing all of the blocks in the given section at once. This can have some performance benefits, especially for lighting purposes. BIome data is *not* sent when ground-up continuous is not set; that means that biomes can't be changed once a chunk is loaded. Sections not specified in the primary bit mask are not changed and should be left as-is.

⚠ As with Multi Block Change and Block Change, it is not safe to send this packet in unloaded chunks, as it can corrupt notchian client's shared empty chunk. Clients should *ignore* such packets, and servers should not send non-ground-up continuous chunk data packets into unloaded chunks.

Notes

1. Empty is defined by the notchian server as being composed of all air, but this can result in lighting issues (MC-80966 (https://bugs.mojang.com/browse/MC-80966)). Custom servers should consider defining empty to mean something like "completely air and without lighting data" or "completely air and with no blocks in the neighboring sections that need to be lit by light from this section".
2. Generally meaning, "it's all air". Of course, lighting is an issue as with before - the notchian client assumes 0 block light and 15 sky light, even when that's not valid (underground sections shouldn't be skylit, and sections near light sources should be lit).
3. The global palette is not continuous in more ways than 1. The more obvious manner is that not all blocks have metadata: for instance, dirt (ID 3) has only 3 states (dirt, coarse dirt, and podzol), so the palette surrounding it is `000000011 0000; 000000011 0001; 000000011 0010; 000000100 0000`. The second way is that structure blocks have an ID of 255, even though there is currently no block with ID 254; thus, there is a large gap.
4. The number of bits in the global palette via the ceil of a base-2 logarithm of the highest value in the palette.
5. There is no requirement for IDs in a section palette to be monotonic; the order within the list is entirely arbitrary and often has to deal with how the palette is built (if it finds a stone block before an air block, stone can come first). (However, although the order of the section palette entries can be arbitrary, it can theoretically be optimized to ensure the maximum possible GZIP compression. This optimization offers little to no gain, so generally do not attempt it.) However, there shouldn't be any gaps in the section palette, as gaps would increase the size of the section palette when it is sent.
6. Most likely, sizes smaller than 4 are not used in the section palette because it would require the palette to be resized several times as it is built in the majority of cases; the processing cost would be higher than the data saved.

7. Most likely, sizes larger than 8 use the global palette because otherwise, the amount of data used to transmit the palette would exceed the savings that the section palette would grant.

Packet structure

Packet ID	State	Bound To	Field Name	Field Type	Notes
0x20	Play	Client	Chunk X	Int	Chunk coordinate (block coordinate divided by 16, rounded down)
			Chunk Z	Int	Chunk coordinate (block coordinate divided by 16, rounded down)
			Ground-Up Continuous	Boolean	See §Ground-up continuous
			Primary Bit Mask	VarInt	Bitmask with bits set to 1 for every 16×16×16 chunk section whose data is included in Data. The least significant bit represents the chunk section at the bottom of the chunk column (from y=0 to y=15).
			Size	VarInt	Size of Data in bytes
			Data	Byte array	See data structure below
			Number of block entities	VarInt	Number of elements in the following array
			Block entities	Array of NBT Tag	All block entities in the chunk. Use the x, y, and z tags in the NBT to determine their positions.

Data structure

The data section of the packet contains most of the useful data for the chunk.

Field Name	Field Type	Notes
Data	Array of Chunk Section	The number of elements in the array is equal to the number of bits set in Primary Bit Mask. Sections are sent bottom-to-top, i.e. the first section, if sent, extends from Y=0 to Y=15.
Biomes	Optional Byte Array	Only sent if Ground-Up Continuous is true; 256 bytes if present

Chunk Section structure

A Chunk Section is defined in terms of other data types. A Chunk Section consists of the following fields:

Field Name	Field Type	Notes
Bits Per Block	Unsigned Byte	Determines how many bits are used to encode a block. Note that not all numbers are valid here.
Palette	Varies	See below for the format.
Data Array Length	VarInt	Number of longs in the following array
Data Array	Array of Long	Compacted list of 4096 indices pointing to state IDs in the Palette
Block Light	Byte Array	Half byte per block
Sky Light	Optional Byte Array	Only if in the Overworld; half byte per block

Data Array, Block Light, and Sky Light are given for each block with increasing x coordinates, within rows of increasing z coordinates, within layers of increasing y coordinates. For the half-byte light arrays, even-indexed items (those with an even x coordinate, starting at 0) are packed into the *low bits*, odd-indexed into the *high bits*.

Palettes

The bits per block value determines what format is used for the palette. In most cases, invalid values will be interpreted as a different value when parsed by the notchian client, meaning that chunk data will be parsed incorrectly if you use an invalid bits per block. Servers must make sure that the bits per block value is correct.

Indirect

There are two variants of this:

- For bits per block <= 4, 4 bits are used to represent a block.
- For bits per block between 5 and 8, the given value is used.

This is an actual palette which lists the block states used. Values in the chunk section's data array are indices into the palette, which in turn gives a proper block state.

The format is as follows:

Field Name	Field Type	Notes
Palette Length	VarInt	Number of elements in the following array.
Palette	Array of VarInt	Mapping of block state IDs in the global palette to indices of this array

Direct

This format is used for bits per block values greater than or equal to 9. The number of bits used to represent a block are the base 2 logarithm of the number of block states, rounded up. For the current vanilla release, this is 13 bits per block.

> The "palette" uses the following format:

Field Name	Field Type	Notes
Dummy Palette Length	VarInt	Should always be 0. Only exists to mirror the format used elsewhere.

If Minecraft Forge is installed and a sufficiently large number of blocks are added, the bits per block value for the global palette will be increased to compensate for the increased ID count. This increase can go up to 16 bits per block (for a total of 4096 block IDs; when combined with the 16 damage values, there are 65536 total states). You can get the number of blocks with the "Number of ids" field found in the RegistryData packet in the Forge Handshake.

Compacted data array

The data array stores several entries within a single long, and sometimes overlaps one entry between multiple longs. For a bits per block value of 13, the data is stored such that bits 1 through 13 are the first entry, 14 through 26 are the second, and so on. Note that bit 1 is the *least* significant bit in this case, not the most significant bit. The same behavior applies when a value stretches between two longs: for instance, block 5 would be bits 53 through 64 of the first long and then bit 65 of the second long.

The Data Array, although varying in length, will never be padded due to the number of blocks being evenly divisible by 64, which is the number of bits in a long.

Example

13 bits per block, using the global palette.

The following two longs would represent...

`1001880C0060020` = `0000000100000000011000100000000110000000110000000100000100000`
`200D0068004C020` = `0000001000000011010000000110000000010110000000100000000`

9 blocks, with the start of a 10th (that would be finished in the next long).

1. Grass, 2:0
2. Dirt, 3:0
3. Dirt, 3:0
4. Coarse dirt, 3:1
5. Stone, 1:0
6. Stone, 1:0
7. Diorite, 1:3
8. Gravel, 13:0
9. Gravel, 13:0
10. Stone, 1:0 (or potentially emerald ore, 129:0)

Biomes

The biomes array is only present when ground-up continuous is set to true. Biomes cannot be changed unless a chunk is re-sent.

The structure is an array of 256 bytes, each representing a Biome ID (https://minecraft.wiki/w/Biome/ID) (it is recommended that 127 for "Void" is used if there is no set biome). The array is indexed by `z * 16 | x`.

Tips and notes

There are several things that can make it easier to implement this format.

- The `13` value for full bits per block is likely to change in the future, so it should not be hardcoded (instead, it should either be calculated or left as a constant).
- Servers do *not* need to implement the palette initially (instead always using 13 bits per block), although it is an important optimization later on.
- The Notchian server implementation does not send values that are out of bounds for the palette. If such a value is received, the format is being parsed incorrectly. In particular, if you're reading a number with all bits set (15, 31, etc), then you're probably reading sky light data.
- The number of longs needed for the data array can be calculated as ((16×16×16 blocks)×Bits per block)÷64 bits per long (which simplifies to 64×Bits per block). For instance, 13 bits per block requires 832 longs.

Also, note that the Notchian client does not render chunks that lack neighbors. This means that if you only send a fixed set of chunks (and not chunks around the player) then the last chunk will not be seen, although you can still interact with it. This is intended behavior, so that lighting and connected blocks can be handled correctly.

Sample implementations

How the chunk format can be implemented varies largely by how you want to read/write it. It is often easier to read/write the data long-by-long instead of pre-create the data to write; however, storing the chunk data arrays in their packed form can be far more efficient memory- and performance-wise. These implementations are simple versions that can work as a base (especially for dealing with the bit shifting), but are not ideal.

Shared code

This is some basic pseudocode that shows the various types of palettes. It does not handle actually populating the palette based on data in a chunk section; handling this is left as for the implementer since there are many ways of doing so. (This does not apply for the direct version).

```
private uint GetGlobalPaletteIDFromState(BlockState state) {
    // NOTE: This method will change in 1.13
    byte metadata = state.getMetadata();
    uint id = state.getBlockId();

    return id << 4 | metadata;
}

private BlockState GetStateFromGlobalPaletteID(uint value) {
    // NOTE: This method will change in 1.13
    byte metadata = data & 0xF;
    uint id = data >> 4;

    return BlockState.ForIDAndMeta(id, metadata);
}

public interface Palette {
    uint IdForState(BlockState state);
    BlockState StateForId(uint id);
    byte GetBitsPerBlock();
    void Read(Buffer data);
    void Write(Buffer data);
}

public class IndirectPalette : Palette {
    Map<uint, BlockState> idToState;
    Map<uint, BlockState> stateToId;
    byte bitsPerBlock;

    public IndirectPalette(byte palBitsPerBlock) {
        bitsPerBlock = palBitsPerBlock;
    }

    public uint IdForState(BlockState state) {
        return stateToId.Get(state);
```

```
    }

    public BlockState StateForId(uint id) {
        return idToState.Get(id);
    }

    public byte GetBitsPerBlock() {
        return bitsPerBlock;
    }

    public void Read(Buffer data) {
        idToState = new Map<>();
        stateToId = new Map<>();
        // Palette Length
        int length = ReadVarInt();
        // Palette
        for (int id = 0; id < length; id++) {
            uint stateId = ReadVarInt();
            BlockState state = GetStateFromGlobalPaletteID(stateId);
            idToState.Set(id, state);
            stateToId.Set(state, id);
        }
    }

    public void Write(Buffer data) {
        Assert(idToState.Size() == stateToId.Size()); // both should be equivalent
        // Palette Length
        WriteVarInt(idToState.Size());
        // Palette
        for (int id = 0; id < idToState.Size(); id++) {
            BlockState state = idToState.Get(id);
            uint stateId = GetGlobalPaletteIDFromState(state);
            WriteVarInt(stateId);
        }
    }
}

public class DirectPalette : Palette {
    public uint IdForState(BlockState state) {
        return GetGlobalPaletteIDFromState(state);
    }

    public BlockState StateForId(uint id) {
        return GetStateFromGlobalPaletteID(id);
    }

    public byte GetBitsPerBlock() {
        return Ceil(Log2(BlockState.GetTotalNumberOfStates)); // currently 13
    }

    public void Read(Buffer data) {
        // Dummy Palette Length
        ReadVarInt(0);
    }

    public void Write(Buffer data) {
        // Dummy Palette Length (ignored)
        ReadVarInt();
    }
}

public Palette ChoosePalette(byte bitsPerBlock) {
    if (bitsPerBlock <= 4) {
        return new IndirectPalette(4);
    } else if (bitsPerBlock <= 8) {
        return new IndirectPalette(bitsPerBlock);
    } else {
        return new DirectPalette();
    }
}
```

Deserializing

When deserializing, it is easy to read to a buffer (since length information is present). A basic example:

```
public Chunk ReadChunkDataPacket(Buffer data) {
    int x = ReadInt(data);
    int z = ReadInt(data);
    bool full = ReadBool(data);
    Chunk chunk;
    if (full) {
        chunk = new Chunk(x, z);
    } else {
        chunk = GetExistingChunk(x, z);
    }
    int mask = ReadVarInt(data);
    int size = ReadVarInt(data);
    ReadChunkColumn(chunk, full, mask, data.ReadByteArray(size));

    int blockEntityCount = ReadVarInt(data);
    for (int i = 0; i < blockEntityCount; i++) {
        CompoundTag tag = ReadCompoundTag(data);
        chunk.AddBlockEntity(tag.GetInt("x"), tag.GetInt("y"), tag.GetInt("z"), tag);
    }

    return chunk;
}

private void ReadChunkColumn(Chunk chunk, bool full, int mask, Buffer data) {
    for (int sectionY = 0; sectionY < (CHUNK_HEIGHT / SECTION_HEIGHT); y++) {
        if ((mask & (1 << sectionY)) != 0) {  // Is the given bit set in the mask?
            byte bitsPerBlock = ReadByte(data);
            Palette palette = ChoosePalette(bitsPerBlock);
            palette.Read(data);

            // A bitmask that contains bitsPerBlock set bits
            uint individualValueMask = (uint)((1 << bitsPerBlock) - 1);

            int dataArrayLength = ReadVarInt(data);
            UInt64[] dataArray = ReadUInt64Array(data, dataArrayLength);

            ChunkSection section = new ChunkSection();

            for (int y = 0; y < SECTION_HEIGHT; y++) {
                for (int z = 0; z < SECTION_WIDTH; z++) {
                    for (int x = 0; x < SECTION_WIDTH; x++) {
                        int blockNumber = (((blockY * SECTION_HEIGHT) + blockZ) * SECTION_WIDTH) + blockX;
                        int startLong = (blockNumber * bitsPerBlock) / 64;
                        int startOffset = (blockNumber * bitsPerBlock) % 64;
                        int endLong = ((blockNumber + 1) * bitsPerBlock - 1) / 64;

                        uint data;
                        if (startLong == endLong) {
                            data = (uint)(dataArray[startLong] >> startOffset);
                        } else {
                            int endOffset = 64 - startOffset;
                            blockId = (uint)(dataArray[startLong] >> startOffset | dataArray[endLong] << endOffset);
                        }
                        data &= individualValueMask;

                        // data should always be valid for the palette
                        // If you're reading a power of 2 minus one (15, 31, 63, 127, etc...) that's out of bounds,
                        // you're probably reading light data instead

                        BlockState state = palette.StateForId(data);
                        section.SetState(x, y, z, state);
                    }
                }
            }

            for (int y = 0; y < SECTION_HEIGHT; y++) {
                for (int z = 0; z < SECTION_WIDTH; z++) {
                    for (int x = 0; x < SECTION_WIDTH; x += 2) {
                        // Note: x += 2 above; we read 2 values along x each time
                        byte value = ReadByte(data);
```

```
                        section.SetBlockLight(x, y, z, value & 0xF);
                        section.SetBlockLight(x + 1, y, z, (value >> 4) & 0xF);
                    }
                }
            }

            if (currentDimension.HasSkylight()) { // IE, current dimension is overworld / 0
                for (int y = 0; y < SECTION_HEIGHT; y++) {
                    for (int z = 0; z < SECTION_WIDTH; z++) {
                        for (int x = 0; x < SECTION_WIDTH; x += 2) {
                            // Note: x += 2 above; we read 2 values along x each time
                            byte value = ReadByte(data);

                            section.SetSkyLight(x, y, z, value & 0xF);
                            section.SetSkyLight(x + 1, y, z, (value >> 4) & 0xF);
                        }
                    }
                }
            }

            // May replace an existing section or a null one
            chunk.Sections[SectionY] = section;
        }
    }

    for (int z = 0; z < SECTION_WIDTH; z++) {
        for (int x = 0; x < SECTION_WIDTH; x++) {
            chunk.SetBiome(x, z, ReadByte(data));
        }
    }
}
```

Serializing

Serializing the packet is more complicated, because of the palette. It is easy to implement with the full bits per block value; implementing it with a compacting palette is much harder since algorithms to generate and resize the palette must be written. As such, this example **does not generate a palette**. The palette is a good performance improvement (as it can significantly reduce the amount of data sent), but managing that is much harder and there are a variety of ways of implementing it.

Also note that this implementation doesn't handle situations where full is false (ie, making a large change to one section); it's only good for serializing a full chunk.

```
public void WriteChunkDataPacket(Chunk chunk, Buffer data) {
    WriteInt(data, chunk.GetX());
    WriteInt(data, chunk.GetZ());
    WriteBool(true);  // Full

    int mask = 0;
    Buffer columnBuffer = new Buffer();
    for (int sectionY = 0; sectionY < (CHUNK_HEIGHT / SECTION_HEIGHT); y++) {
        if (!chunk.IsSectionEmpty(sectionY)) {
            mask |= (1 << chunkY);  // Set that bit to true in the mask
            WriteChunkSection(chunk.Sections[sectionY], columnBuffer);
        }
    }
    for (int z = 0; z < SECTION_WIDTH; z++) {
        for (int x = 0; x < SECTION_WIDTH; x++) {
            WriteByte(columnBuffer, chunk.GetBiome(x, z));  // Use 127 for 'void' if your server doesn't support biomes
        }
    }

    WriteVarInt(data, mask);
    WriteVarInt(data, columnBuffer.Size);
    WriteByteArray(data, columnBuffer);

    // If you don't support block entities yet, use 0
```

```
    // If you need to implement it by sending block entities later with the update block entity packet,
    // do it that way and send 0 as well.  (Note that 1.10.1 (not 1.10 or 1.10.2) will not accept that)

    WriteVarInt(data, chunk.BlockEntities.Length);
    foreach (CompoundTag tag in chunk.BlockEntities) {
        WriteCompoundTag(data, tag);
    }
}

private void WriteChunkSection(ChunkSection section, Buffer buf) {
    Palette palette = section.palette;
    byte bitsPerBlock = palette.GetBitsPerBlock();

    WriteByte(bitsPerBlock);
    palette.Write(buf);

    int dataLength = (16*16*16) * bitsPerBlock / 64; // See tips section for an explanation of this calculation
    UInt64[] data = new UInt64[dataLength];

    // A bitmask that contains bitsPerBlock set bits
    uint individualValueMask = (uint)((1 << bitsPerBlock) - 1);

    for (int y = 0; y < SECTION_HEIGHT; y++) {
        for (int z = 0; z < SECTION_WIDTH; z++) {
            for (int x = 0; x < SECTION_WIDTH; x++) {
                int blockNumber = (((blockY * SECTION_HEIGHT) + blockZ) * SECTION_WIDTH) + blockX;
                int startLong = (blockNumber * bitsPerBlock) / 64;
                int startOffset = (blockNumber * bitsPerBlock) % 64;
                int endLong = ((blockNumber + 1) * bitsPerBlock - 1) / 64;

                BlockState state = section.GetState(x, y, z);

                UInt64 value = palette.IdForState(state);
                value &= individualValueMask;

                data[startLong] |= (Value << startOffset);

                if (startLong != endLong) {
                    data[endLong] = (value >> (64 - startOffset));
                }
            }
        }
    }

    WriteVarInt(dataLength);
    WriteLongArray(data);

    for (int y = 0; y < SECTION_HEIGHT; y++) {
        for (int z = 0; z < SECTION_WIDTH; z++) {
            for (int x = 0; x < SECTION_WIDTH; x += 2) {
                // Note: x += 2 above; we read 2 values along x each time
                byte value = section.GetBlockLight(x, y, z) | (section.GetBlockLight(x + 1, y, z) << 4);
                WriteByte(data, value);
            }
        }
    }

    if (currentDimension.HasSkylight()) { // IE, current dimension is overworld / 0
        for (int y = 0; y < SECTION_HEIGHT; y++) {
            for (int z = 0; z < SECTION_WIDTH; z++) {
                for (int x = 0; x < SECTION_WIDTH; x += 2) {
                    // Note: x += 2 above; we read 2 values along x each time
                    byte value = section.GetSkyLight(x, y, z) | (section.GetSkyLight(x + 1, y, z) << 4);
                    WriteByte(data, value);
                }
            }
        }
    }
}
```

Full implementations

- Java, 1.12.2, writing only, with palette (https://github.com/GlowstoneMC/Glowstone/blob/dev/src/main/java/net/glowstone/chunk/ChunkSection.java)
- Java, 1.9, both sides (https://github.com/Steveice10/MCProtocolLib/blob/4ed72deb75f2acb0a81d641717b7b8074730f701/src/main/java/org/spacehq/mc/protocol/data/game/chunk/BlockStorage.java#L42)
- Python, 1.7 through 1.13 (https://github.com/barneygale/quarry). Read/write, paletted/unpaletted, packets (https://github.com/barneygale/quarry/blob/master/quarry/types/buffer/v1_7.py#L403)/arrays (https://github.com/barneygale/quarry/blob/master/quarry/types/chunk.py)
- Python, 1.9, reading only (https://github.com/SpockBotMC/SpockBot/blob/0535c31/spockbot/plugins/tools/smpmap.py#L144-L183)
- C, 1.9, reading only (https://github.com/Protryon/Osmium/blob/fdd61b9/MinecraftClone/src/ingame.c#L512-L632)
- C, 1.11.2, writing only (https://github.com/Protryon/Basin/blob/master/basin/src/packet.c#L1124)
- C++, 1.12.2, writing only (https://github.com/cuberite/cuberite/blob/master/src/Protocol/ChunkDataSerializer.cpp#L190)

Old format

The following implement the previous (http://wiki.vg/index.php?title=SMP_Map_Format&oldid=7164) (before 1.9) format:

- Java, 1.8 (https://github.com/GlowstoneMC/Glowstone/blob/d3ed79ea7d284df1d2cd1945bf53d5652962a34f/src/main/java/net/glowstone/GlowChunk.java#L640)
- Python, 1.4 (https://github.com/barneygale/smpmap)
- Node.js, 1.8 (https://github.com/PrismarineJS/prismarine-chunk)
