TECHMC.WIKI

Graduate Texts in Minecraft

An Introduction to Technical Minecraft

KNOWLEDGE EXISTS. STRUCTURE MATTERS.

ENGLISH EDITION

2026-07-13

Contents

Preface 4

CHAPTER 1 Block Update 6

Introduction 7
1.1 Update Concepts and Update Types 8
1.2 Continuous Block Updates and Analysis Methods 23
1.3 Special Update Behaviors 31

CHAPTER 2 Micro Timing 39

2.1 Tick and Inter-Tick Timing 40
2.2 A First Look at Intra-Tick Timing 44
2.3 Scheduled Ticks and Scheduled Tick Components 51
2.4 Block Entities 68
2.5 Block Events 76

CHAPTER 3 Redstone Components & Features 81

3.1 Rails 82
3.2 Pistons 90

CHAPTER 4 Blocks 154

The Nature of Blocks 155
4.1 Blocks and Block States 156
4.2 Placing, Changing, and Breaking Blocks 166

CHAPTER 5 Chunks 175

Introduction 176
5.1 Fundamental Concepts and Structure 179
5.2 Generation State and ChunkStatus 187
5.3 Levels and Runtime State Layers 193
5.4 Server Master Dispatcher TACS 203
5.5 Main Thread Async Tasks 209
5.6 ChunkHolder Lifecycle 214
5.7 Scheduled Ticks and Chunk Runtime Levels 222
5.8 Random Ticks and Block Entity Ticks 229
5.9 Entity Loading and Tracking 237
5.10 Save and Unload Pipeline 243
5.11 POI and Bypass Storage 252
5.12 Client Visibility and Chunk Packets 259
5.13 Game Event and In-Chunk Listeners 265

CHAPTER 6 Masa Mod 269

6.1 Masa Gadget 270
6.2 Litematica Tutorial 272

CHAPTER 7 Tree Farm 277

Foreword 278
7.1 Prerequisites and the Basic Structure of Tree Farms 279
7.2 Designing the Simplest Tree Farm from Scratch 295
7.3 Attempting to Design a Multi-Species Tree Farm 305
7.4 Why Your Tree Farm Is Slow — Introduction to High-Speed Tree Farms 320
7.5 Why Your Tree Farm Stalls — Introduction to Dustless Wiring 342
7.6 Everything About 4gt Tree Farms 355
7.7 Large Spruce Tree Farms 378

CHAPTER 8 绿萌科技 384

Molforte

Overview 385

8.1 Preparation: Environment Setup and Fundamentals 388

8.2 Appendix 0a 395

Twisuki

Preface 396

8.3 Introduction 399

8.4 Prerequisites and Setup 400

8.5 Component Basics 413

8.6 Flying Machine Basics 418

8.7 Engines 425

8.8 Mobility 426

8.9 Semi-Dynamic Engines and Extensions 435

8.10 Linkages 442

APPENDIX A 附录 450

A.1 Stacks 451

A.2 Glossary 453

Preface

Graduate Texts in Minecraft (GTMC) is a community-driven project offering technical and redstone tutorials for Minecraft. It serves both as a beginner's guide and as in-depth documentation for experienced players looking to understand Minecraft's internal mechanics.

The editorial team is committed to producing content that is clear, accurate, and accessible. Community contributors are also welcome to help write and improve the project by submitting Pull Requests.

1 Explanatory Notes

Each article in the documentation can be divided into a **Basic section** and an **Advanced section**, targeting beginners and experienced players respectively.

Some **Basic section** content that is relatively rare or uncommon will be marked with an asterisk (*) in the chapter title.

For ease of understanding, the **Basic** section may include some generalized statements that aren't perfectly rigorous but hold true in typical situations. These statements will be marked with a superscript[1] and explained further in the **Advanced** section or in footnotes.

The documentation assumes readers have a **fairly basic** understanding of Minecraft and its redstone components. If you run into issues while reading, feel free to consult the Wiki. Likewise, special or atypical cases fall outside the scope of this article and are usually documented on the Wiki. When such cases exist for a given concept, they will be marked with a superscript[2].

The code in this documentation is decompiled using `1.20.1-yarn`. Any special cases will be noted with a superscript[3].

Feel free to join the QQ group for discussion: 1031431170

2 Copyright Notice and Licensing

Copyright © 2024-2026 GTMC Wiki.

Original content in this documentation is licensed under CC BY-NC-SA 4.0.

Minecraft ® is a registered trademark of Mojang AB.
Minecraft © Mojang AB / Microsoft Corporation. All Rights Reserved.

Game screenshots, textures, source code, and other content referenced in this documentation are protected by copyright law and are quoted in a limited manner in compliance with the Minecraft EULA and Minecraft Usage Guidelines.

The source code referenced in this documentation comes from decompilation research of Minecraft, using the Yarn mappings for readable decompilation. These mappings are licensed under CC0 1.0.

This documentation does not constitute redistribution of Minecraft game code or any form of commercial use, nor does it provide complete game code or executable files that can be run directly. All descriptions of the game's internal mechanisms are limited to learning and explanatory purposes.

Contact emails: tanh_heng@outlook.com, fannbryan@gmail.com, yuhan2680@qq.com, suyang233@hotmail.com, Molforte@outlook.com

GTMC Team:

- Writers / ABxl_lly, `@BFladderbean`, `@Molforte`, `@Ryan100c`, `@tanh_Heng`, `@Twisuki`, `@xhbsh`, `@yuhan2680`
- Development / `@BFladderbean`, `@4rcadia`
- Collaboration / Petris, `@void`, XJH_Jorhai
- Special Thanks / `@Fallen_Breath`, 五羊飞kaniol

This is not an official documentation. We are not affiliated with Mojang or Microsoft by any means.

Footnotes

1. Like this ↩
2. Like like this this ↩
3. Like like like this this this ↩

CHAPTER 1

1

Block Update

IN THIS CHAPTER

1.1 Update Concepts and Update Types

1.2 Continuous Block Updates and Analysis Methods

1.3 Special Update Behaviors

Introduction

1 Overview

This chapter aims to analyze the update behavior in Minecraft from both practical and code perspectives in an accessible yet thorough manner, primarily explaining concepts including NC updates, PP updates, other types of updates, depth-first search, and update chains. Each section will be divided into basic explanations and advanced source code analysis.

1.1 Update Concepts and Update Types

This article is an introduction to update theory.

Basics

- Understanding Updates
- Different Types of Updates
- Block Self-Inspection Behavior

Advanced (Source Code Analysis)

- Brief Overview of NC Update and PP Update Invocation and Response
- Invocation of Block Self-Inspection

1 Updates in the General Sense

In Minecraft, blocks establish connections with each other through mutual "notifications".

Suppose you place a note block on the ground, then place a redstone block next to it. The note block plays a sound. How did it know to do that? Does it constantly ask the player whether it should make a sound? Obviously not. The note block played because the redstone block notified it: something changed nearby.

This **notification** behavior is what we call an **update** in the general sense.

Note that the redstone block doesn't say "I am a redstone block" or "this changed from air to a redstone block"—it simply signals that something changed. In other words, update behavior carries no details about what caused it.

2 Types of Updates

In Minecraft, there is more than one type of update.

Let's continue with the example above. Now consider another scenario: you place a redstone block on the ground, then place a fence gate one block away, and a note block between them. The note block doesn't play. Right-clicking the gate to open it still doesn't trigger the note block. Both placing the redstone block and opening the gate are changes that should notify the note block. Why doesn't it play? Breaking the fence gate, however, does trigger it.

Why is this?

You might wonder: perhaps the notifications from placing a redstone block and opening a fence gate differ somehow.

These two notifications are indeed different. They are called **NC Update** and **PP Update**, respectively.

You might also wonder: why doesn't the note block play immediately when placed? Doesn't it check its state when it first appears? It doesn't—note blocks skip this check.

This self-check on placement is called **self-inspection**. **Self-inspection** is a **special type of update**. Note blocks do not perform self-inspection.

You may already know that comparators detect the number of items in a container, outputting redstone signals of strength 0–15 based on the item count. When item count changes, the container doesn't need to emit an **NC Update** or **PP Update**—only the internal quantity changed, with little effect on surrounding blocks. Yet comparators still update their output signal strength, meaning they receive change notifications. Containers emit a special notification when item count changes.

Such notifications are called **Comparator Updates**.

3 NC Update

3.1 NC Update: Concept and Behavior

Blocks emit **NC Updates** when they are placed, broken, or undergo changes that significantly affect surrounding blocks. For example:

- Placement and breaking of blocks
- Changes in redstone dust power level
- Piston push/pull*
- ...

> In the official deobfuscation, NC Update is `neighborChanged`, same as the mcp deobfuscation. In yarn deobfuscation it's called `updateNeighbors`

However, some changes do not emit **NC Updates**, the most common being:

- Connection state changes of connectable blocks, such as glass panes connecting to other blocks, various walls connecting to other blocks
- Connection state changes of redstone dust, such as redstone dust connected to north-south redstone dust also connecting to east redstone dust
- Opening/closing of trapdoors and fence gates
- Activation state changes of dispensers and droppers (**! Important**)
- Activation state changes of hoppers
- ...

> The specific behavior of pistons is not within the scope of this article.

A block that emits an NC Update is called an **update source**[1]. For most blocks, the update source is the block itself when emitting updates.

When an update source emits an NC Update, it follows the order **West East Down Up North South**, sending NC Updates to each immediately adjacent block in turn.

Some blocks have more than one update source when emitting NC Updates, with special update sources and ranges:

- Redstone dust power level changes are second-order neighbor updates[2].
- Directional redstone components (repeaters, comparators, observers) first update the block their output end points to, then emit NC Updates with that block as the update source. They do not update themselves.
- When flat powered rails switch activation state, they first generate NC Updates with themselves as the update source, then generate NC Updates with the block below as the update source.
- When sloped powered rails switch activation state, the update order is up, middle, down, middle, down, up (self-source abbreviated **middle**, above-source **up**, below-source **down**; see [Update Behavior of Sloped Rails](./特殊的更新行为.md#3.2-:advanced 斜侧铁轨的更新行为)).

These are common and typical examples. More special cases can be found at Wiki-Block Update

3.2 Properties of NC Update, QC Activation, BUD Devices

The most typical property of NC Updates is their ability to trigger **BUD devices**.

A block whose current state differs from its **intended state** is a **BUD device** (Block Update Detector). This state is called a **BUD state**.

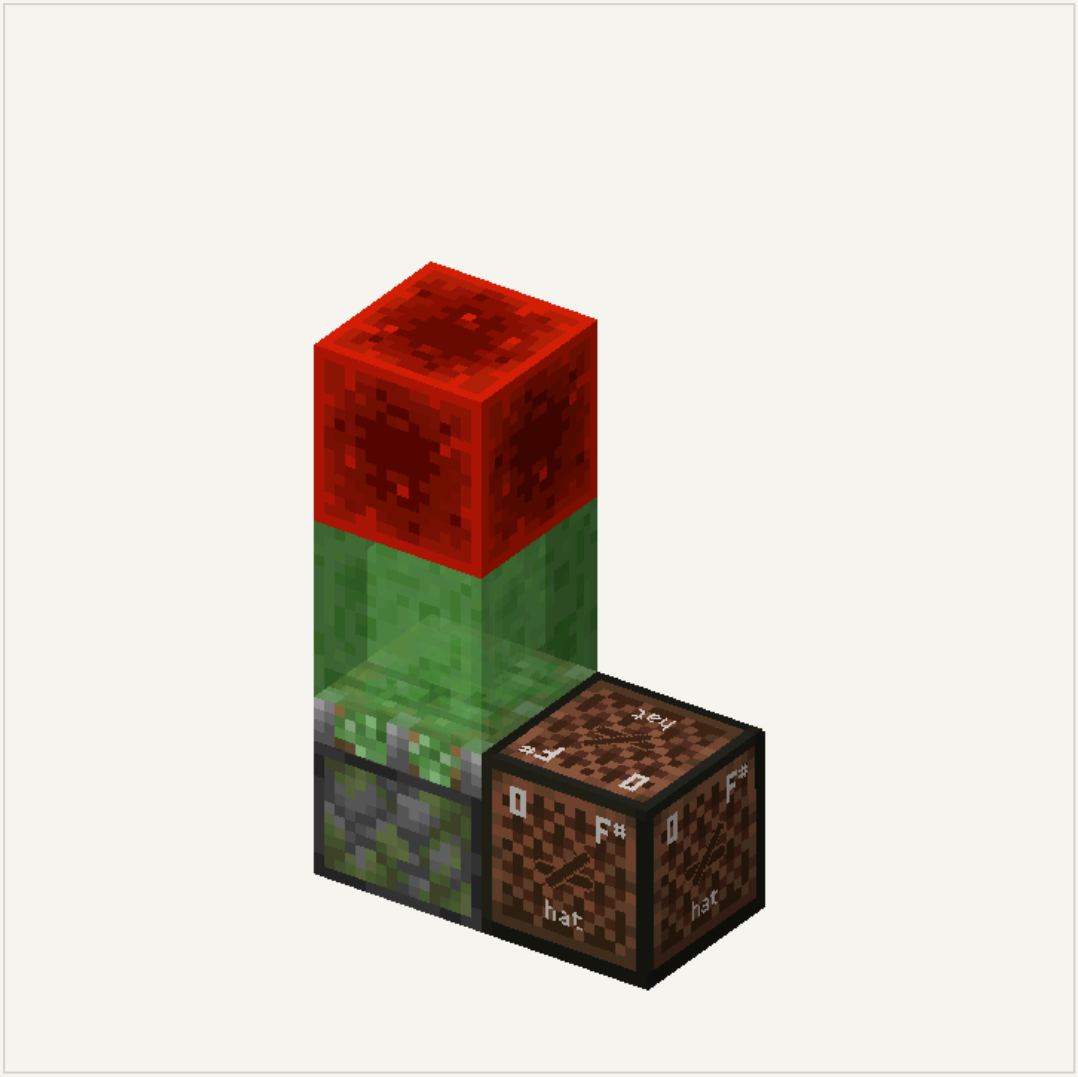
The most typical BUD device is a piston that is powered (receiving a redstone signal) but not activated (not extended). Pistons have a special powering range: they count as powered when both the piston block and the block above it (even air) receive power. Powering a piston via the air above it is called **QC powering**. When such a piston extends after receiving an NC Update, that's **QC activation**.

QC (quasi-connectivity) is a property shared by pistons, sticky pistons, droppers, and dispensers.

Consider a redstone block placed diagonally above a piston. The piston is now QC powered. However, placing the redstone block only updates the six adjacent blocks—the piston isn't among them, so nothing notifies it of the signal. The piston should extend but hasn't, because it received no NC Update. Its current state differs from its intended state—the piston is in a **BUD state** and acts as a **BUD device**.

Placing a block next to the piston emits an NC Update. The piston receives it and exits the BUD state.

The simplest commonly used self-resetting BUD device is as follows:



▶ This figure is animated. View original

In the image, the redstone block **QC powers** the sticky piston below. When the sticky piston receives an NC Update and extends, the redstone block no longer powers it. The sticky piston retracts, and the redstone block QC powers it again.[3]

4 PP Update

4.1 PP Update: Concept and Behavior

Almost all changes emit **PP Updates**, with a few exceptions:

- Placing blocks using commands
- Changing block states using the debug stick

There is one additional exception:

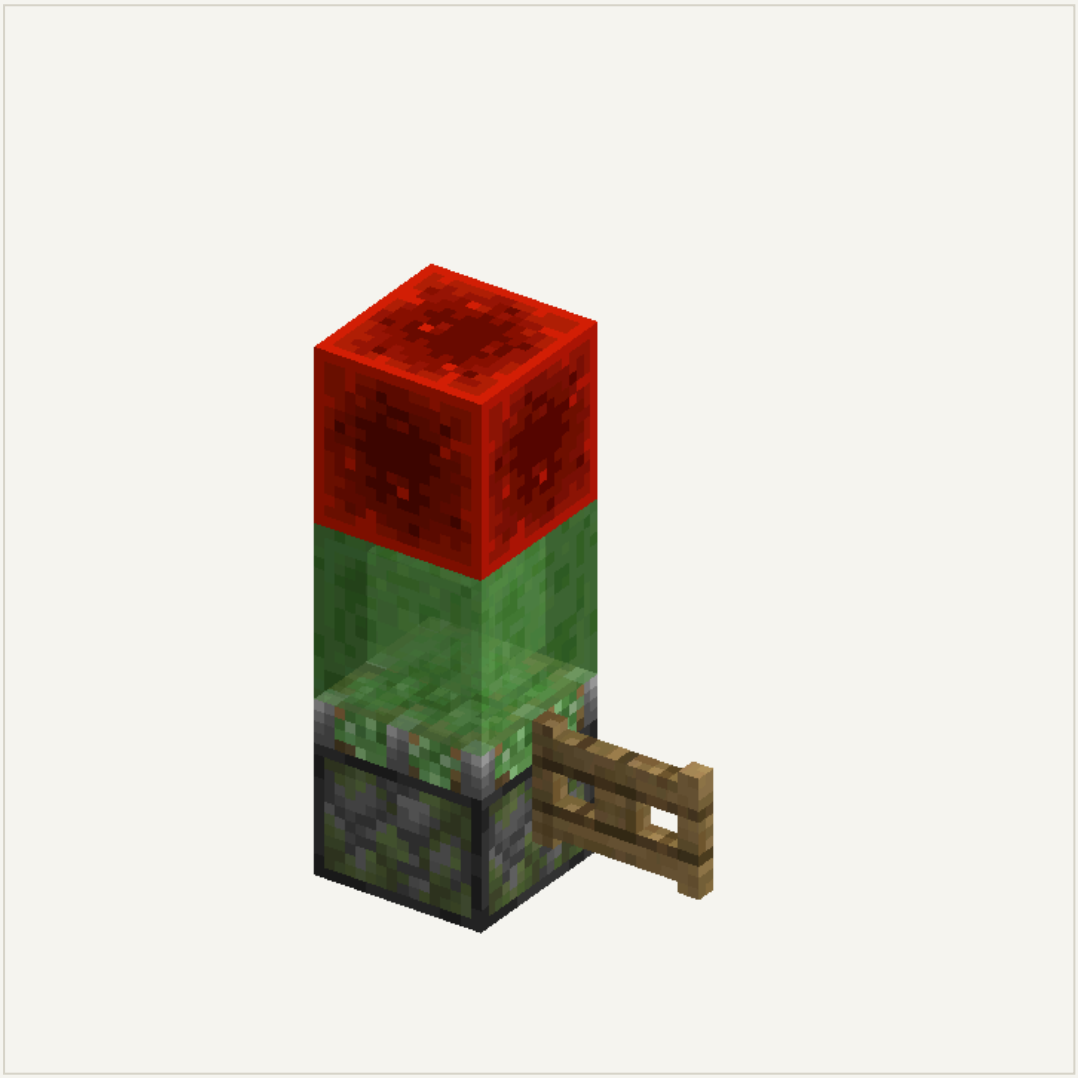
- When a sticky piston pushes an activating observer, the observer doesn't emit a PP Update when it arrives.

> In the official deobfuscation, PP Update is `updateShape`, meaning update shape (for example, the shape of various walls, opening/closing of trapdoors, activation state changes of redstone components can all be called "shape" here). This deobfuscation name is more helpful for understanding PP Updates. The PP Update naming comes from the `postPlacement` method in mcp deobfuscation. In yarn deobfuscation it's called `getStateForNeighborUpdate`

Unlike NC Updates, the order of PP Updates is **West East North South Down Up**.

The behaviors listed above that *do not emit **NC Updates*** are typical cases that emit **PP Updates** but not **NC Updates**. Similarly, **BUD devices** cannot detect **PP Updates**.

For example, the fence gate state change emits a PP Update, but the piston BUD device won't detect it.



▶ This figure is animated. View original

Except when placing or breaking blocks, most redstone components have different NC and PP Update ranges. For example, a repeater's NC Update range covers the output end and its neighbors (excluding the repeater itself), while its PP Update range covers the repeater's own neighbors.

PP Updates reflect **changes to the block itself**; NC Updates **notify neighboring blocks** that may need to change state.

For example, a repeater's on/off state is a **change to itself**, communicated via PP Updates. When a repeater turns on, it powers the block at its output end. If that block can transmit redstone signals, it propagates them to its neighbors—so the repeater may affect both the output block and its neighbors. The repeater uses NC Updates to **notify potentially affected blocks to recheck their state**.

Another example: when a dropper fires and drops an item, the action doesn't affect surrounding blocks[4], so dropper activation doesn't emit NC Updates. However, the dropper's activation state does change, so it emits PP Updates.

This makes the distinction between PP and NC Updates—and their different ranges—clearer.

4.2 Properties of PP Update

Since BUD devices cannot detect PP Updates, what can?

Observers. The face detects PP Updates; the red-dot side outputs a redstone signal.

Observers respond only to PP Updates.[5] Since NC Updates usually accompany PP Updates, this distinction isn't always obvious. But it can be demonstrated: an observer facing a repeater activates when the repeater toggles; an observer facing the air at the repeater's output does not. Air doesn't change, so the observer doesn't respond.

Using BUD devices and observers together, you can test NC and PP Update ranges without reading source code.

5 Comparator Update

5.1 Comparator Update: Concept and Behavior

As the name suggests, Comparator Updates are specifically for comparators.

The following behaviors emit Comparator Updates:

- General containers (chests, barrels, hoppers, etc.) when item count changes. Signal strength calculation: see below.
- Containers in a broader sense—blocks with a "capacity" concept such as composters (signal strength equals composting level, range 0–9) and cauldrons (signal strength equals liquid level, range 0–3).
- Item frames. Output signal strength depends on the rotation angle of the displayed item (strength 1–8). When no item is placed, signal strength is 0.
- Lecterns. Signal strength calculation: see below.
- Detector rails with dropped items above them.
- Detector rails with minecarts containing items (chest minecart, hopper minecart). When item count changes, a Comparator Update emits after 20 gt. Signal strength calculation is the same as general containers.

- Chiseled bookshelves (1.20+). Signal strength equals the position of the last book placed or removed (first row 1–3, second row 2–6).

Comparators detect container capacity both directly and through an intervening block that transmits redstone signals. Comparator Updates reach comparators horizontally adjacent to the container or within second-order range through a signal-transmitting block. They do not affect non-comparator blocks.

5.2 Comparator Signal Strength Calculation

For general containers:

- Output signal strength = floor of `average fill ratio of each item slot * 14`. If the container is not empty, add `+1`. Empty container outputs 0; non-empty container outputs the average fill ratio mapped to 1–15, floored.

For lecterns:

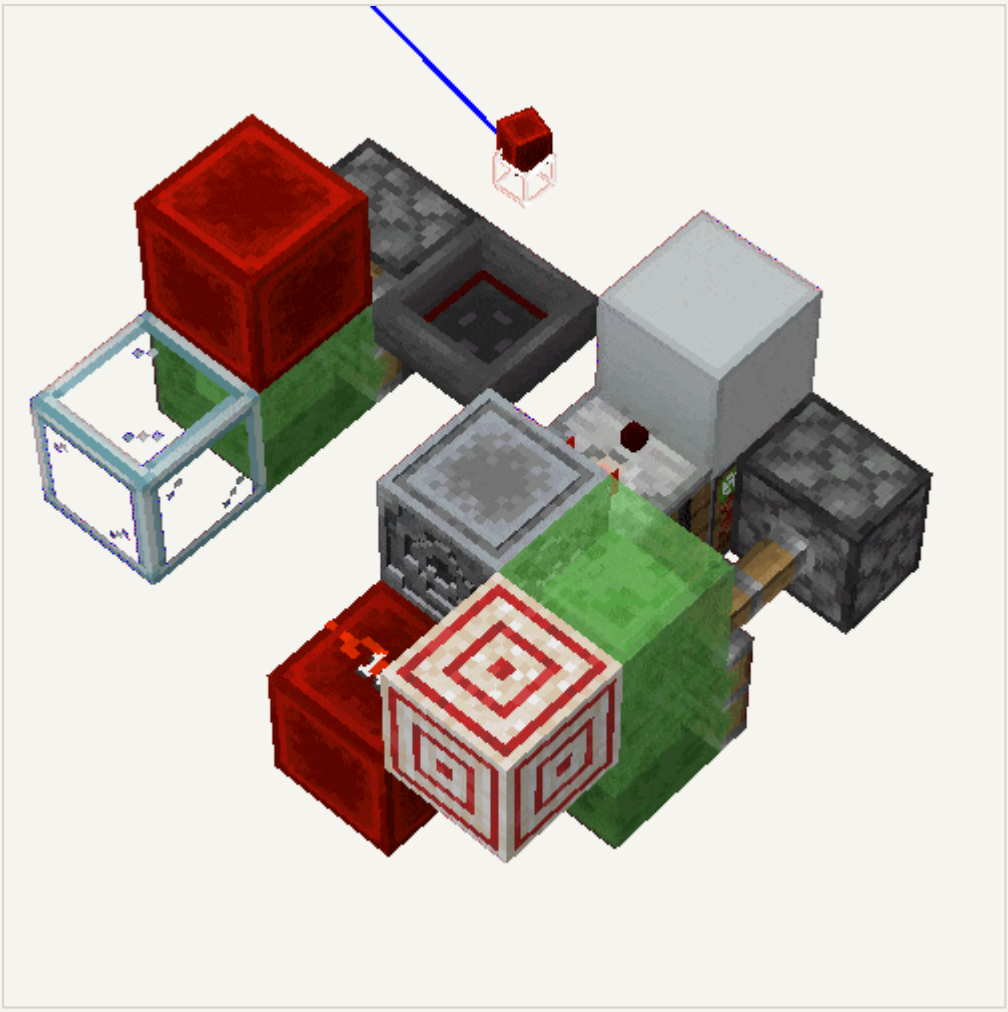
- When no book is placed on the lectern, output signal strength is 0.
- When a book is placed on the lectern and the book has only one page, output signal strength is 15.
- When a book is placed on the lectern and the book has more than one page, output signal strength is floor of `current page number / (total pages - 1) * 14` then `+1`. Here "current page number" starts from 0; if counting from 1 as players see, it's equivalent to `(current page - 1) / (total pages - 1) * 14` floored then `+1`.
- Empty lectern outputs 0; non-empty lectern outputs the current page ratio mapped to 1–15, floored.

5.3 Comparator Update Detector (CUD)

When a comparator's current state differs from its **intended state**, it constitutes a Comparator Update Detector (CUD). The simplest example: use a comparator to detect a filled composter through an intervening block, then push the composter away with a piston. The comparator still outputs a signal.

Other resettable CUD structures are complex; they are demonstrated here but not explained in detail.





▶ This figure is animated. View original

> *Image shows a CUD based on redstone dust turning.*
>
> You can see the BUD didn't activate, while the CUD detected the Comparator Update emitted by the hopper.

6 Block Self-Inspection

Redstone components **self-inspect** when placed—checking whether they should change state. For example, place a redstone block, then place a repeater with its input adjacent to it. The repeater hasn't received an update, yet it lights up. It self-inspected on placement, found it should power on, and did.

A piston arriving at its position counts as "being placed," so pistons self-inspect once after extending and once after retracting.

7 Invocation and Response of NC Update and PP Update

7.1 `setBlockState` Function

`setBlockState` controls update behavior through a `bitflags` parameter (i.e., `FLAG`) in the `World` class (`World.java`):

```java
public boolean setBlockState(BlockPos pos, BlockState state, int flags, int maxUpdateDepth) {
        ...
        BlockState blockState = worldChunk.setBlockState(pos, state, (flags & Block.MOVED) != 0);
        if (blockState != null) {
            BlockState blockState2 = this.getBlockState(pos);
            if (blockState2 == state) {
                ...
                // NC Update invocation controlled by FLAG
                if ((flags & Block.NOTIFY_NEIGHBORS) != 0) {
                    this.updateNeighbors(pos, blockState.getBlock());
                    if (!this.isClient && state.hasComparatorOutput()) {
                        this.updateComparators(pos, block);
                    }
                }
                // PP Update invocation controlled by FLAG
                if ((flags & Block.FORCE_STATE) == 0 && maxUpdateDepth > 0) {
                    int i = flags & ~(Block.NOTIFY_NEIGHBORS | Block.SKIP_DROPS);
                    blockState.prepare(this, pos, i, maxUpdateDepth - 1);
                    state.updateNeighbors(this, pos, i, maxUpdateDepth - 1);
                    state.prepare(this, pos, i, maxUpdateDepth - 1);
                }
                ...
            }
            return true;
        }
        return false;
    }
```

7.2 Invocation and Response of NC Update

In the source code, NC Updates are mainly invoked through two methods:

- `updateNeighbor` and its derivatives
- `setBlockState` with `flag` where `bit0 = 1`

`updateNeighbor` and its derivatives: `updateNeighbor`, `updateNeighborsExcept`, `updateNeighborsAlways`.

`updateNeighbor` is the most fundamental method, taking three parameters: `pos` (coordinates of the block to update), `sourceBlock` (block type emitting the update), and `sourcePos` (coordinates of the update source).

`updateNeighborsAlways` and `updateNeighborsExcept` perform updates by calling `NeighborUpdater.updateNeighbors` in `NeighborUpdater.java`:

```java
public static final Direction[] UPDATE_ORDER = new Direction[]{Direction.WEST, Direction.EAST, Direction.DOWN, Direction.UP, Direction.NORTH, Direction.SOUTH};
default public void updateNeighbors(BlockPos pos, Block sourceBlock, @Nullable Direction except) {
    for (Direction direction : UPDATE_ORDER) {
        if (direction == except) continue;
        this.updateNeighbor(pos.offset(direction), sourceBlock, pos);
    }
}
```

Therefore, the NC Update order is West East Down Up North South.

Defined in `Block.java`:

```java
// Broadcast block update events to surrounding blocks
public static final int NOTIFY_NEIGHBORS = 1;
// Notify listeners and clients that need to react to this block change
public static final int NOTIFY_LISTENERS = 2;
// Default behavior when block changes: same effect as enabling both NOTIFY_NEIGHBORS and NOTIFY_LISTENERS above.
public static final int NOTIFY_ALL = 3;
// Skip updates, force set block state.
public static final int FORCE_STATE = 16;
```

Of these methods, only the `sourcePos` in `updateNeighborsAlways`/`updateNeighborsExcept` and the `pos` in `setBlockState` qualify as an "update source"—the core that updates the six adjacent blocks (or all but one direction) around a block. `updateNeighbor` is a simpler, direct update.

For PP Updates only, `FLAG` is usually `NOTIFY_LISTENERS`; when emitting both PP and NC Updates, `FLAG` is usually `NOTIFY_ALL`. Some blocks that override `onBlockAdded` have different invocation methods (see below).

Each block responds to NC Updates through its `neighborUpdate` function.

7.3 Invocation of PP Update

PP Updates are invoked through `setBlockState`. When `bit4 = 0` in `FLAG`, a PP Update is generated.

`state.updateNeighbors` is defined in `AbstractBlockState` (`AbstractBlock.java`):

```java
public final void updateNeighbors(WorldAccess world, BlockPos pos, int flags, int maxUpdateDepth) {
    BlockPos.Mutable mutable = new BlockPos.Mutable();
    for (Direction direction : DIRECTIONS) {
        mutable.set((Vec3i)pos, direction);
        world.replaceWithStateForNeighborUpdate(direction.getOpposite(), this.asBlockState(), mutable, pos, flags, maxUpdateDepth);
    }
}
```

`AbstractBlock` defines:

```java
protected static final Direction[] DIRECTIONS = new Direction[]{Direction.WEST, Direction.EAST, Direction.NORTH, Direction.SOUTH, Direction.DOWN, Direction.UP};
```

Therefore, PP Update order is West East North South Down Up.

`world.replaceWithStateForNeighborUpdate`:

```java
default public void replaceWithStateForNeighborUpdate(Direction direction, BlockState neighborState,
BlockPos pos, BlockPos neighborPos, int flags, int maxUpdateDepth) {
NeighborUpdater.replaceWithStateForNeighborUpdate(this, direction, neighborState, pos, neighborPos,
flags, maxUpdateDepth - 1);
}
```

This calls `replaceWithStateForNeighborUpdate` in `NeighborUpdater`:

```java
public static void replaceWithStateForNeighborUpdate(WorldAccess world, Direction direction,
BlockState neighborState, BlockPos pos, BlockPos neighborPos, int flags, int maxUpdateDepth) {
    BlockState blockState = world.getBlockState(pos);
    BlockState blockState2 = blockState.getStateForNeighborUpdate(direction, neighborState,
    world, pos, neighborPos);
    Block.replace(blockState, blockState2, world, pos, flags, maxUpdateDepth);
}
```

Finally, it calls each block's `getStateForNeighborUpdate` method—the method by which blocks respond to PP Updates.

7.4 NC Update and PP Update Invocation for Redstone Components

For most redstone components, `setBlockState` passes `FLAG=2`. NC Updates are emitted by `worldChunk.setBlockState` calling the overridden `onBlockAdded` method:

In `worldChunk.setBlockState`:

```java
public BlockState setBlockState(BlockPos pos, BlockState state, boolean moved) {
    ...
    if (!this.world.isClient) {
        state.onBlockAdded(this.world, pos, blockState, moved);
    }
    ...
    return blockState;
}
```

PP Updates are normally invoked by `setBlockState` controlled by `FLAG`.

Using the redstone repeater as an example, in `AbstractRedstoneGateBlock` (`AbstractRedstoneGateBlock.java`), which the repeater inherits:

```java
public void scheduledTick(BlockState state, ServerWorld world, BlockPos pos, Random random) {
    if (this.isLocked(world, pos, state)) {
        return;
    }
    boolean bl = state.get(POWERED);
    boolean bl2 = this.hasPower(world, pos, state);
    if (bl && !bl2) {
        world.setBlockState(pos, (BlockState)state.with(POWERED, false), Block.NOTIFY_LISTENERS);
    } else if (!bl) {
```

```
        world.setBlockState(pos, (BlockState)state.with(POWERED, true), Block.NOTIFY_LISTENERS);
        if (!bl2) {
            world.scheduleBlockTick(pos, this, this.getUpdateDelayInternal(state),
            TickPriority.VERY_HIGH);
        }
    }
}
@Override
public void onBlockAdded(BlockState state, World world, BlockPos pos, BlockState oldState, bool-
ean notify) {
    this.updateTarget(world, pos, state);
}
    protected void updateTarget(World world, BlockPos pos, BlockState state) {
    Direction direction = state.get(FACING);
    BlockPos blockPos = pos.offset(direction.getOpposite());
    world.updateNeighbor(blockPos, this, pos);
    world.updateNeighborsExcept(blockPos, this, direction);
}
```

`scheduledTick` is called when the repeater executes a scheduled tick. It calls `setBlockState`, which calls `worldChunk.setBlockState`, which in turn calls the overridden `onBlockAdded`—generating NC Updates for the block at the repeater's output end and its neighbors (excluding the repeater itself). Then `setBlockState` calls `state.updateNeighbors` to generate PP Updates.

7.5 Order of NC Update and PP Update

For most blocks using `setBlockState`, NC Updates fire before PP Updates. This ordering matters because it affects subsequent update chains. This article does not cover continuous block updates.

7.6 Observer's NC and PP Update

The observer is an exception. It doesn't call NC Updates in the `onBlockAdded` method, but directly calls `setBlockState` first in `scheduledTick` to emit PP Updates, then calls `updateNeighbors` to emit NC Updates. Therefore, observers follow a PP-first-then-NC order when turning on and off.

In the observer's `onBlockAdded`:

JAVA

```
@Override
public void onBlockAdded(BlockState state, World world, BlockPos pos, BlockState oldState, bool-
ean notify) {
    if (state.isOf(oldState.getBlock())) {
        return;
    }
    if (!world.isClient() && state.get(POWERED).booleanValue() &&
    !world.getBlockTickScheduler().isQueued(pos, this)) {
        BlockState blockState = (BlockState)state.with(POWERED, false);
        world.setBlockState(pos, blockState, Block.NOTIFY_LISTENERS | Block.FORCE_STATE);
        this.updateNeighbors(world, pos, blockState);
    }
}
```

The first `if` contains `state.isOf(oldState.getBlock())` which is used to determine whether the block at the current position was still the current block before its state was changed. For example, if a piston pushes an observer, the position where the observer arrives is `air` before the change and `observer` after the change, at which point this function returns `true`. While a normal observer always returns `false` when calling this function.

Therefore, the observer's `onBlockAdded` method usually doesn't execute subsequent content, and thus doesn't emit NC Updates from `onBlockAdded`. Instead, as mentioned above, it "directly calls `setBlockState` first in `scheduledTick` to emit PP Updates, then calls `updateNeighbors` to emit NC Updates."

8 Invocation of Block Self-Inspection

Blocks are called with the `onPlaced` function when placed, and redstone components complete self-inspection in this function. For example, the redstone repeater:

```java
@Override
public void onPlaced(World world, BlockPos pos, BlockState state, LivingEntity placer, ItemStack itemStack) {
    if (this.hasPower(world, pos, state)) {
        world.scheduleBlockTick(pos, this, 1);
    }
}
```

The source code for piston self-inspection is not within the scope of this article.

9 Control of Update Behavior

In the ServerWorld.setBlockState method, an integer parameter is passed in: flag
This flag is actually treated as a 9-bit binary code, where each of the 9 bits controls different behaviors

10 Definition of flag's 9-bit Flags

- **Bit 0**: Emit neighbor updates.
- **Bit 1**: Notify listeners (server and client).
- **Bit 2**: Suppress client rendering.
- **Bit 3**: Force client sync redraw.
- **Bit 4**: Force state storage.
- **Bit 5**: Prevent item drops.
- **Bit 6**: Mark piston movement.
- **Bit 7**: Light update.
- **Bit 8**: Trigger behavior side effects.

11 Constant Explanation

Multiple update behaviors are defined in Block.java:

CONSTANT	VALUE	BINARY (9 BITS)	BEHAVIOR	USAGE
`NOTIFY_NEIGHBORS`	1	`00000000`**`1`**	Trigger surrounding block updates	Commonly called NC Update
`NOTIFY_LISTENERS`	2	`0000000`**`1`**`0`	Notify server and client listeners	Sync state
`NO_REDRAW`	4	`00000`**`1`**`00`	Prevent client redraw (used with `NOTIFY_LISTENERS`)	Invisible update
`REDRAW_ON_MAIN_THREAD`	8	`0000`**`1`**`000`	Force client sync redraw	Block appearance change
`FORCE_STATE`	16	`000`**`1`**`0000`	Bypass virtual state, directly store specified state	Force block replacement
`SKIP_DROPS`	32	`00`**`1`**`00000`	Prevent container blocks from dropping items when broken	Command replacement etc.
`MOVED`	64	`0`**`1`**`000000`	Mark block moved by piston	Piston push/pull logic trigger

`SKIP_REDSTONE_WIRE_STATE_REPLACEMENT`	128	`0`**`1`**`0000000`	Skip redstone wire state replacement	-
`SKIP_BLOCK_ENTITY_REPLACED_CALLBACK`	256	**`1`**`00000000`	Skip block entity replacement callback	-
`SKIP_BLOCK_ADDED_CALLBACK`	512	**`1`**`000000000`	Skip self-inspection	Don't trigger `onBlockAdded`
`SKIP_REDRAW_AND_BLOCK_ENTITY_REPLACED_CALLBACK`	260	**`1`**`0000`**`1`**`00`	Combine `NO_REDRAW` and `SKIP_BLOCK_ENTITY_REPLACED_CALLBACK`	Minimize update, skip rendering and entity callback
`NOTIFY_ALL`	3	`000000`**`11`**	Combine `NOTIFY_NEIGHBORS` and `NOTIFY_LISTENERS`	Default full server-side update
`NOTIFY_ALL_AND_REDRAW`	11	`00000`**`1`**`0`**`11`**	Combine `NOTIFY_ALL` and `REDRAW_ON_MAIN_THREAD`	Server update and force client redraw
`FORCE_STATE_AND_SKIP_CALLBACKS_AND_DROPS`	816	**`11`**`00`**`11`**`0000`	Combine `FORCE_STATE`, `SKIP_DROPS`, `MOVED`, `SKIP_BLOCK_ADDED_CALLBACK`	Force change, skip self-inspection and drops

Footnotes

1. Understanding of **update source** is not explained extensively in the basics section, and not all NC Updates have an update source. See Advanced Section. ↩
2. Redstone dust has more complex update behavior, not expanded in this article. ~~May add redstone dust special DLC later~~ ↩
3. Brief explanation of the piston mechanics in the 1.3.2 BUD device (for reference only). The sticky piston doesn't receive any NC Update while extending, so it won't drop the block mid-extension. After extending into position, it triggers self-inspection, finds it's not powered, and retracts. During retraction, the piston block arrives first (due to sticky piston block arrival

order). Self-inspection triggers, but the redstone block hasn't arrived yet (still in block entity state). The redstone block arrives later, so the sticky piston won't extend again from self-inspection. ↩

4. Not considering items being picked up by hoppers. Hopper pickup is the hopper's own behavior, not controlled by updates. ↩
5. This is considered only from the perspective of update types. Observers only decide whether to change state when receiving PP Updates. If "respond" is taken broadly to mean "observer changes," then besides scheduled tick behavior, when an observer is pushed into position by a piston, it calls its own `onBlockAdded` method. Only when there's no scheduled tick queued at the arrival position and the observer's original state was lit does it call `setBlockstate` internally, ending in the extinguished state (no scheduled tick exists). Observers originally extinguished produce no changes. The scheduled tick added when an extinguished observer arrives via piston comes from the tile entity's "update" `postProcessState`, which still calls the observer's PP response function `getStateForNeighborUpdate`. ↩

1.2 Continuous Block Updates and Analysis Methods

This section teaches deeper update mechanisms in Minecraft and how to analyze update order.

1 General Block Update Behavior

Blocks always emit updates in the order of **NC update first, PP update second**. To deeply understand update theory, you must grasp one concept: **an update is not a single event, but a "process"**.

The diagram is as follows:

```
Block A:
├── Emit NC update
│   ├── NC update triggered some events
│   │   └── These events triggered more events
│   └── These events finished running
│   * NC update complete *
└── Emit PP update
    ├── PP update triggered some events
    └── These events finished running
    * PP update complete *
```

For example, when Block A emits an NC update, it updates Block B. At this point, B changes state and emits an NC update. The update process is as follows:

```
├── Block A emits NC update
│   ├── Block B receives NC update
│   ├── Block B changes state
│   ├── Block B emits NC update
│   │   └── Nothing happens
│   │   * Block B NC update complete *
│   └── Block B emits PP update
│       └── Nothing happens
│       * Block B PP update complete *
│   * Block A NC update complete *
└── Block A emits PP update
    └── Nothing happens
    * Block A PP update complete *
```

The commonly discussed **NC update order** refers to **the sequence in which NC updates are emitted**, and **PP update order** similarly refers to **the sequence in which PP updates are emitted**. From the diagrams and examples above, we can see that in more complex updates, the PP update order can also be approximately considered as **the sequence in which block NC update processes complete**.

2 Update Methods in Minecraft

We will use some clear, simple examples to help you gradually understand how this complex update process occurs.

For example, consider the BUD rail chain below. What will happen when A1 is updated?



2.1 Analysis Method for Update Order

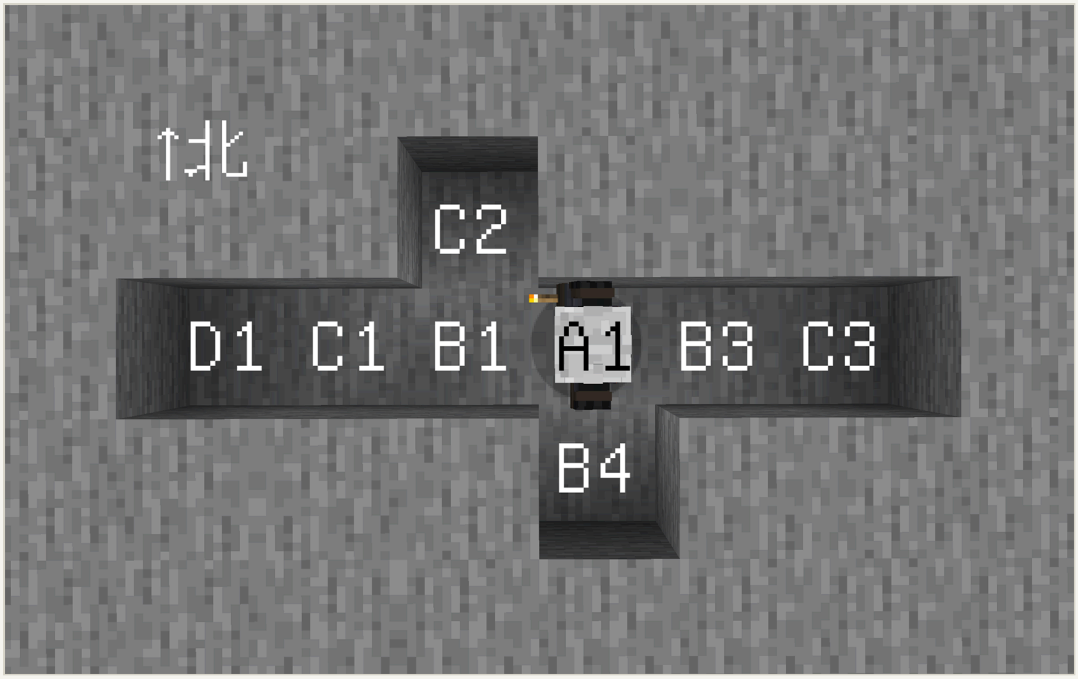
~~To understand updates, we need to first set aside updates~~

Imagine you are someone with OCD, and right now you are holding many torches in a mine cavern.

Because you have OCD, you will strictly follow these rules:

1. You place a torch at every block you walk to
2. At each block, you search for available blocks in the order **West, East, Down, Up, North, South**
3. If there is a block you can walk to, enter it immediately
4. When there is no path forward, walk back
5. When walking back, pick up the torches you placed
6. When walking back to a junction, if there are **other directions** not yet checked, continue checking them

Following the rules above, let's see how you would explore the cavern below:

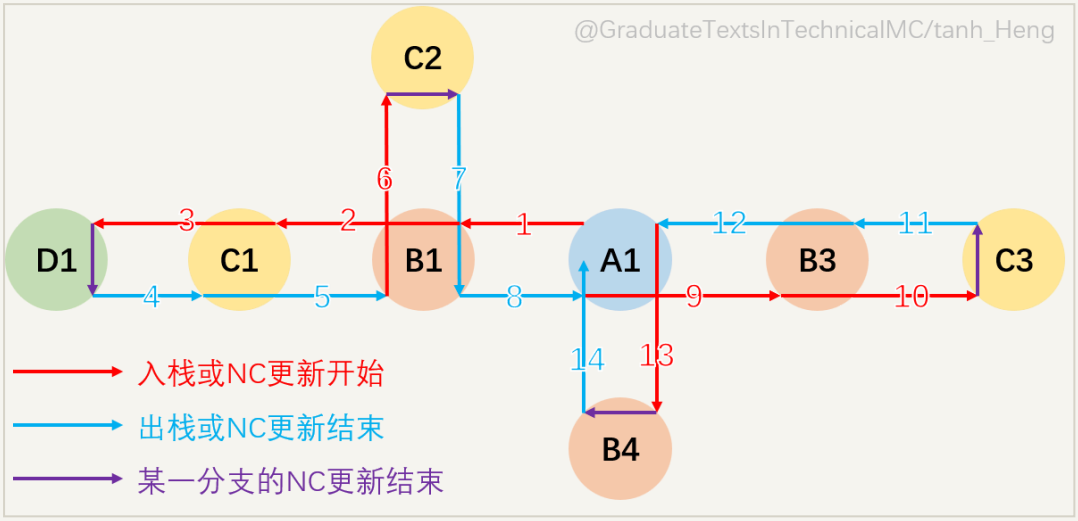


1. Initially, you are at A1, so you place a torch at A1
2. You start searching for paths in the order **West, East, Down, Up, North, South**, and you find a block to the west
3. You enter this block (B1), place a torch, continue checking → find a path to the west, enter C1, place a torch, check → path to the west, enter D1, place a torch
4. At this point you check in the order West, East, Down, Up, North, South and find no paths, so you pick up the torch and walk back (enter C1)
5. You already checked west, so you check East, Down, Up, North, South, find no paths, pick up torch, return to B1
6. East, Down, Up have no paths, but when you check North you find a path, so enter (C2), place a torch
7. After checking, you find no paths, pick up torch return to B1
8. B1 has no more paths after checking, pick up torch return to A1. You start checking other directions.
9. There's a path to the east, enter B3 place torch → path to the east, enter C3 place torch
10. No more paths, pick up torch return to B3 → no paths, pick up torch return to A1
11. Check remaining directions, path to the south, enter B4, place torch
12. No paths, pick up torch return to A1
13. A1 has no remaining paths, pick up torch, exploration complete

This process of exploring the cavern is the update process, where "placing torches" corresponds to emitting NC updates (pushing onto the stack), and picking up torches corresponds to emitting PP updates (popping from the stack). As for what this "stack" means, you can refer to Appendix - Terminology Explanation → Stack and Call Stack, and this update method is depth-first search (DFS).

Recommended to watch with BiliBili-tanh_Heng: NC Update × Depth-First Search.exe

Let's understand the update process in the example in a more visual way. We can view the pushing of NC updates onto the stack as "progressively going deeper", and the popping of NC updates from the stack, which is PP updates, as "progressively returning". So, let's redraw the diagram above in a more visual way and add some more nodes:



The diagram shows the update chain after A1 emits an update. The top is the north side.

From the diagram, we can clearly see:

NC update order: `A1->B1->C1->D1->C2->B3->C3->B4`

PP update order: `D1->C1->C2->B1->C3->B3->B4->A1`

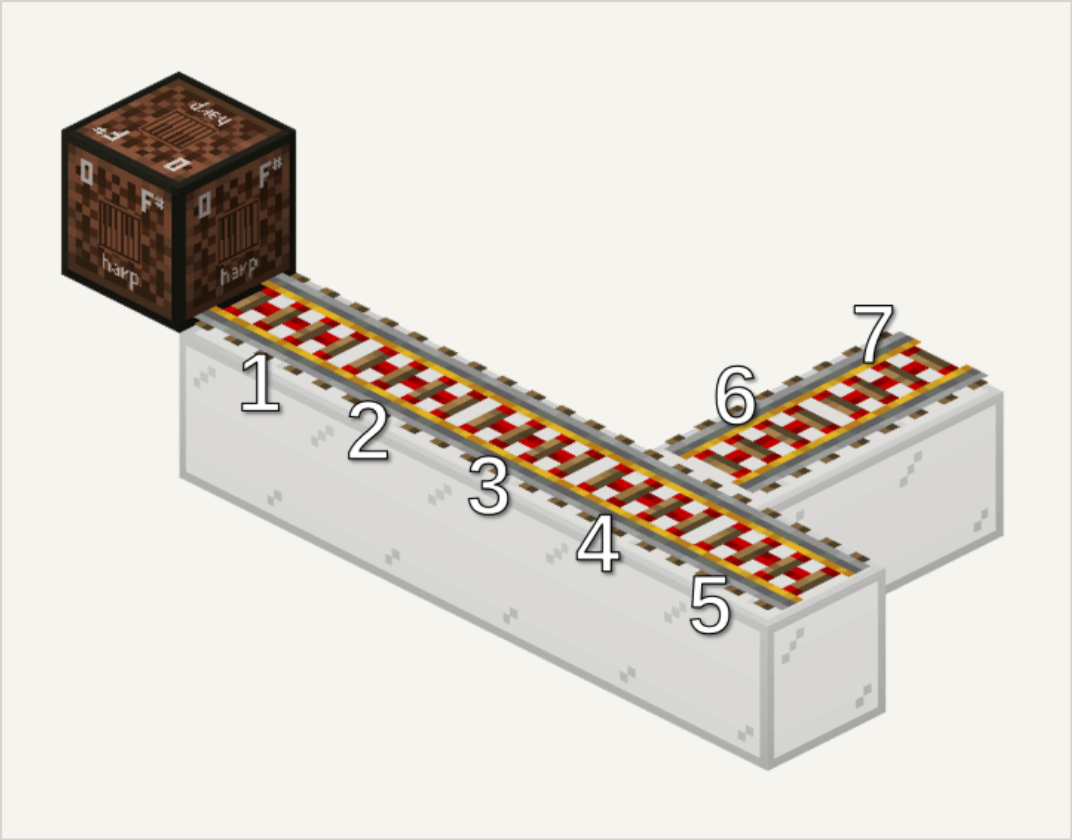
Imagine the update behavior as a point that starts from A1, moves according to a certain update order, and "doesn't turn back until hitting a wall". Then, the movement path of this point forms the arrows shown in the diagram. The process of progressively going deeper is pushing onto stack or NC update, and the process of "turning back" is popping from stack or PP update.

Note that this analysis simplifies PP updates. This is because typically, PP updates do not cause **continuous block updates**[1]. To analyze PP updates, the method is similar to NC updates—analyze from the stack perspective.

3 Common Update Order Analysis

3.1 BUD Rail Chain Update Order Analysis

Let's try to analyze update order with a practical example:



When rails change state, they emit NC updates first, then PP updates. The diagram shows a chain of rails in BUD state, with rails 1-5 oriented east-west. Question: **After the note block is pressed, what are the NC update and PP update orders for the rail extinguishing?**

When the note block is pressed, it emits an NC update. The adjacent rail 1 receives the NC update, changes its own state, emits an NC update, updating rail 2, then 3 and 4. At rail 4, it first updates rail 5 on the east-west side. After rail 5 receives the update, it changes its own state and emits an NC update. There are no subsequent NC updates, so rail 5 then emits a PP update. Return to rail 4, which then updates rails 6 and 7 on the north-south side. Go deeper to rail 6, which emits an NC update; then go deeper to rail 7, which emits an NC update. Rail 7 finishes updating and emits a PP update. Return to rail 6, which emits a PP update. Return to rail 4, which emits a PP update. Then return to rails 3, 2, 1, which emit PP updates.

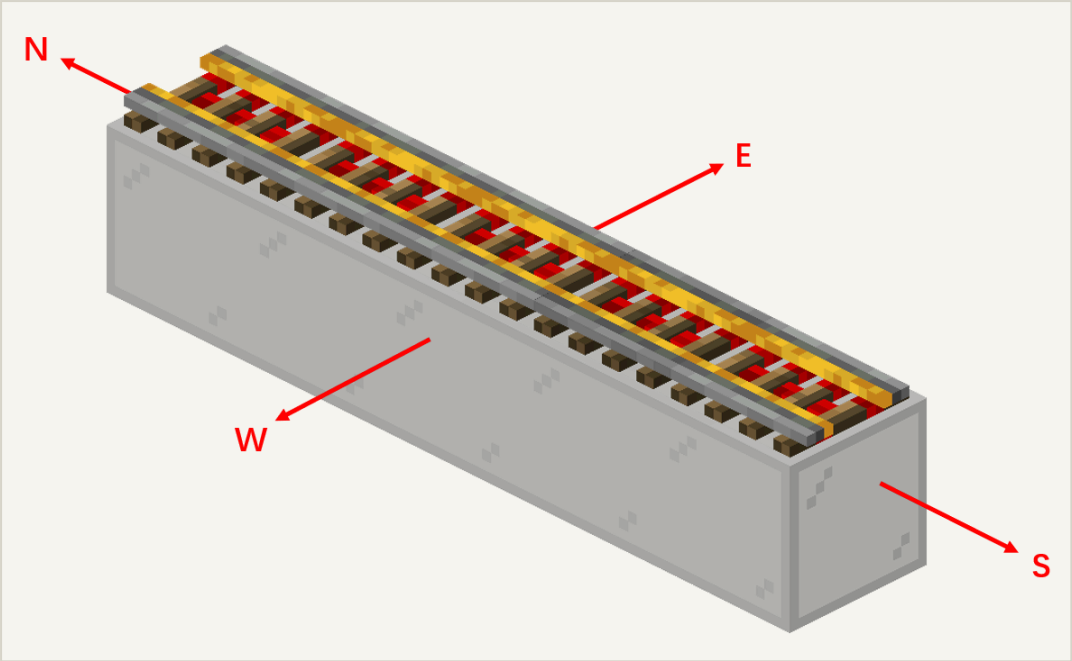
Therefore,

NC update order is: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

PP update order is: 5 -> 7 -> 6 -> 4 -> 3 -> 2 -> 1

4 Analysis Considering NC Update Direction*

This section is not commonly used; readers may choose to skip it.



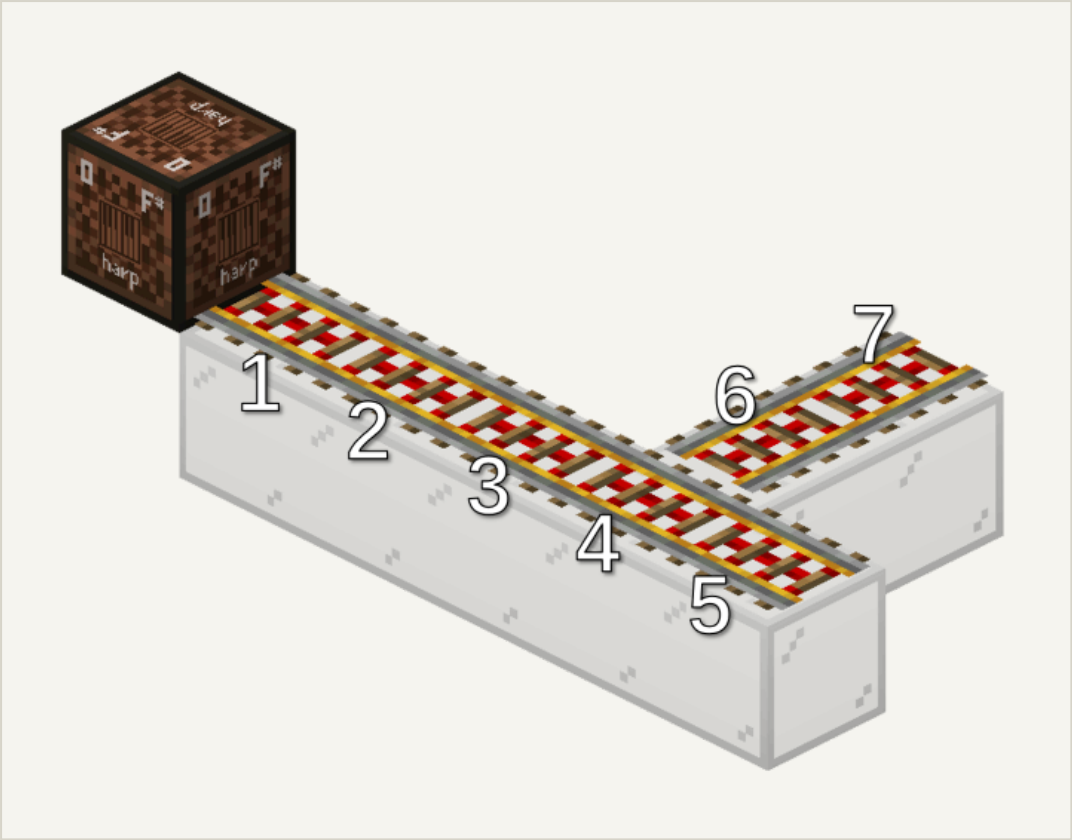
The rail chain direction in the diagram is as marked, with the rail chain oriented north-south. We will use the northernmost rail as an example to analyze its NC update order in different directions.

NC updates are emitted in the directional order of West, East, Down, Up, North, South. When the northernmost rail extinguishes, it first emits west and east updates—nothing happens; then emits down and up updates—nothing happens; then emits a north update—nothing happens; finally emits a south update, updating the south rail, and subsequent updates can be analyzed using the method described above.

If we create a BUD device on the north side of this rail, then this BUD device will be updated first, and then the south rail will undergo continuous NC updates.

And if we consider **the order of NC updates to the block above among the NC updates in six directions**, then it is **from north to south**, that is, **from near to far**.

Let's reconsider the example in 2.3.1:



The note block is at the westernmost end of the 1-5 rail chain. Consider only the NC updates of the 1-5 rail chain.

When the note block is pressed, rail 1 receives the update and extinguishes, first emitting a west update—nothing happens; then emitting an east update, updating rail 2, rail 2 starts updating... After rails 2, 3, 4... finish updating, rail 1's **east NC update** is complete, and rail 1 continues to update down, up, north, south.

At this point, if we consider **the order of NC updates to the block above**, it becomes 5, 4, 3, 2, 1, which is **from far to near**.

Then readers might wonder: since the order of NC updates also needs to consider directionality, what is the point of learning the order of emitting NC updates in 2.3.3?

There are two uses:

1. Used to analyze PP updates. Usually when we use rail chains, we use observers to detect, so this is actually to help us analyze PP updates.
2. The directionality of NC updates builds on 2.3.3. Only by understanding 2.3.3 can you better understand directionality.

5 Exercise

In the second example above, we did not consider rails 6 and 7. Readers can try to analyze the order of **NC updates to the block above** for the entire BUD rail chain when considering rails 6 and 7.

Answer: 5 -> 4 -> 6 -> 7 -> 3 -> 2 -> 1

Footnotes

1. Elements like walls and fences change state after receiving PP updates and emit PP updates. These types of blocks may trigger continuous block updates caused by PP updates. However, this is not within the scope of typical analysis. ↩

1.3 Special Update Behaviors

This section introduces special update behaviors of certain blocks.

1 Redstone Dust Update Behavior

1.1 The Concept of 2nd-Order Neighbor Updates

In #01, we introduced the concept of "update source" for NC updates. The most common block updates, such as those triggered by placing or breaking blocks, use the position of the changed block as the update source and send NC updates in six directions. This is called a "1st-order neighbor update."

Redstone dust behaves differently. For example, when the power level of redstone dust changes, it emits a "2nd-order neighbor update," meaning the redstone dust uses its own position and the six blocks adjacent to it as update sources, each sending NC updates in six directions.

The naming of "1st-order" and "2nd-order" comes from the Manhattan distance[1] of the farthest block receiving an NC update from the block emitting the update. A 1st-order neighbor update means the farthest block receiving an NC update has a Manhattan distance of 1 from the emitting block, while a 2nd-order neighbor update means this distance is 2.

1.2 Update Source Order for Redstone Dust 2nd-Order Neighbor Updates (Locational Nature of Redstone Dust Updates)

When redstone dust emits 2nd-order neighbor updates, the order of the 7 update sources is determined by hash values derived from the redstone dust's coordinates. This is what gives redstone dust updates their locational nature. These update sources have a 97% probability of being divided into three groups (`-Y, +Z, +X`, `O`, `+Y, -Z, -X`), and emit updates in the following order:

Note: In the table, O represents the source redstone dust, -X etc. represent the direction of the update source relative to the source redstone dust

FIRST	THEN	LAST	PROBABILITY OF THIS ORDER
-Y, +Z, +X	O	+Y, -Z, -X	24.267%
+Y, -Z, -X	O	-Y, +Z, +X	24.267%
O	-Y, +Z, +X	+Y, -Z, -X	12.133%
O	+Y, -Z, -X	-Y, +Z, +X	12.133%
-Y, +Z, +X	+Y, -Z, -X	O	12.133%
+Y, -Z, -X	-Y, +Z, +X	O	12.133%
	Others		<0.2%

The update order within each group is fixed, but the arrangement order of groups is random. Additionally, there are some other extremely low-probability arrangement options.

2 Update Behavior of Diagonal Rails

2.1 Overview

When the activation state (`powered=true|false`) of a diagonally placed powered rail (`PoweredRailBlock`) changes, it emits two groups of updates in the following order:

1. First group: Sequentially emit NC updates with the block above, itself, and the block below as update sources.
2. Second group: Sequentially emit NC updates with itself, the block below, and the block above as update sources.

2.2 Source Code Analysis

2.2.1 Call Stack

```perl
PoweredRailBlock.updateBlockState(...)
├── world.setBlockState(pos, ..., 3) // pos, flags=3
│   ├── worldChunk.setBlockState(...)
│   │   └── PoweredRailBlock.onStateReplaced(...)
│   │       ├── world.updateNeighborsAlways(pos.up(), this)   // 1.1 上方
│   │       ├── world.updateNeighborsAlways(pos, this)        // 1.2 自身
│   │       └── world.updateNeighborsAlways(pos.down(), this) // 1.3 下方
│   └── world.updateNeighbors()     // 2.1 自身
├── world.updateNeighborsAlways(pos.down(), this)   // 2.2 下方
└── world.updateNeighborsAlways(pos.up(), this)     // 2.3 上方
```

2.2.2 Source Code Explanation

```java
protected void updateBlockState(BlockState state, World world, BlockPos pos, Block neighbor) {
    boolean bl = (Boolean)state.get(POWERED);
    boolean bl2 = world.isReceivingRedstonePower(pos) || this.isPoweredByOtherRails(world, pos, state, true, 0) || this.isPoweredByOtherRails(world, pos, state, false, 0);
    if (bl2 != bl) {
        world.setBlockState(pos, (BlockState)state.with(POWERED, bl2), 3); // 2.1 自身
        world.updateNeighborsAlways(pos.down(), this); // 2.2 下方
        if (((RailShape)state.get(SHAPE)).isAscending()) {
            world.updateNeighborsAlways(pos.up(), this); // 2.3 仅斜铁轨 上方
        }
    }
}
// 见调用栈，由于 setBlockState 先调用到 onStateReplaced 然后再 updateNeighbors
// 所以先执行这一组 NC 更新，然后再以自身为更新源发出 NC 更新，即上文中标注的 2.1
public void onStateReplaced(BlockState state, World world, BlockPos pos, BlockState newState, boolean moved) {
    if (!moved) {
        super.onStateReplaced(state, world, pos, newState, moved);
```

```
        if (((RailShape)state.get(this.getShapeProperty())).isAscending()) {
            world.updateNeighborsAlways(pos.up(), this); // 1.1 仅斜铁轨 上方
        }
        if (this.forbidCurves) { // 所有不能弯曲的铁轨 也就是除了普通铁轨
            world.updateNeighborsAlways(pos, this); // 1.2 自身
            world.updateNeighborsAlways(pos.down(), this); // 1.3 下方
        }
    }
}
```

3 Redstone Dust Updates and Code Analysis

4 Recursive Checking Behavior of Rail Chains

4.1 Overview

When a rail receives an NC update, it checks its own powered state. It sets itself to `powered=true` when either of the following two conditions is met:

1. Directly receiving a redstone signal, i.e., being directly powered
2. Being powered by connected rails, i.e., being indirectly powered

A rail checking whether it is indirectly powered does not simply check if connected rails are powered. Instead, it recursively searches the entire rail chain for directly powered rails within 8 rails of itself *(considering the distance between two directly connected rails as 1 rail)*. The process is as follows:

1. Initially, the distance variable is `0`. If `distance=8`, the check terminates and returns `False`, meaning not indirectly powered.
2. A rail has two directions. When checking begins, which direction the rail checks first is controlled by a boolean value. When the boolean is `True` and the rail shape is north-south/east-west oriented, it checks the south/west side rail; when the rail shape is ascending `shape=ascending_*`, it checks the rail above on the south/west side. When the boolean is `False`, the opposite occurs. This boolean value is first `True`, then `False` when checking begins. Due to short-circuit evaluation[2]: if the first check with boolean `True` succeeds, the second check with boolean `False` will not be performed. During the recursive checking process *(see step 6)*, this boolean value does not change.
3. Move coordinates `(i, j, k)` to the position of the next rail to check.
4. Determine whether the current rail could possibly connect with a rail below in the current checking direction. If the current rail is ascending east/west/south/north and the checking direction is east/west/south/north, it is considered impossible. This step prepares for step 8. For example: if the current rail is ascending north, and checking toward the south, then this rail could connect with a rail below on the south side; if the current rail is ascending north and checking toward the north, then this rail cannot connect with a rail below on the south side.

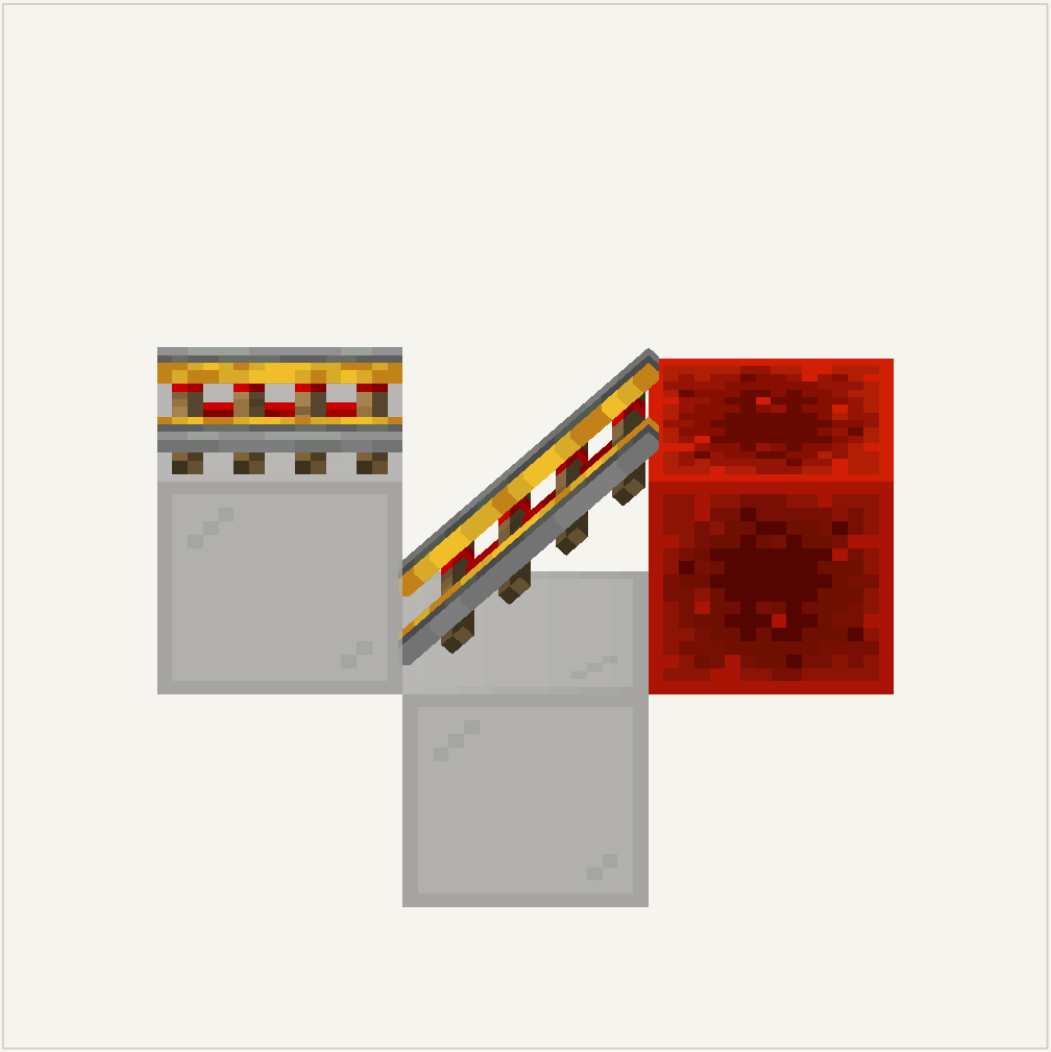
5. If the current rail shape is ascending south/north or ascending east/west, consider the current rail shape as north-south oriented or east-west oriented—this shape change prepares for the next step *(6.b.)*.
6. Check coordinates `(i, j, k)`:
    1. Check if the current position is a rail. If not, return `False` and end the current step check.
    2. Check if the current position is connected to the original rail. This step does not check "connected," but rather checks "not connected": if the original rail shape is east-west oriented, and the current rail shape is north-south/ascending north/ascending south, they are not connected, return `False`, and end step 6 check. If the original rail is north-south oriented, and the current rail shape is east-west/ascending east/ascending west, they are not connected, return `False`, and end the current step check.
    3. Steps i and ii are actually preconditions for iii and iv. This step checks if the rail at the current position is directly powered. If so, return `True` and end the current step check.
    4. Jump to step 1, increase distance by `1`, and check the next rail according to the starting boolean value *(recursion)*.
7. If step 6 returns `True`, terminate the check and return `True`.
8. *(See step 4 judgment)* If the current rail could possibly connect with a rail below in the current checking direction, check the block below coordinates `(i, j, k)`, i.e., jump to step 6, but check coordinates `(i, j-1, k)`. In the example from step 4, if step 3 returns `False`, then check the rail below on the south side of the original rail.

4.2 Source Code Analysis

```java
// This is the starting point
@Override
protected void updateBlockState(BlockState state, World world, BlockPos pos, Block neighbor) {
    boolean bl2;
    boolean bl = state.get(POWERED);
    boolean bl3 = bl2 = world.isReceivingRedstonePower(pos) || this.isPoweredByOtherRails(world, pos, state, true, 0) || this.isPoweredByOtherRails(world, pos, state, false, 0);
    // 逻辑运算短路说的是这里 bl3。
    // 如果被直接充能，那么直接结束。否则先 bl = true，后 false。见后
    if (bl2 != bl) {
        world.setBlockState(pos, (BlockState)state.with(POWERED, bl2), Block.NOTIFY_ALL);
        world.updateNeighborsAlways(pos.down(), this);
        if (state.get(SHAPE).isAscending()) {
            world.updateNeighborsAlways(pos.up(), this);
        }
    }
}
// 便于区分，这块函数标记为 isPoweredByOtherRails(01)
protected boolean isPoweredByOtherRails(World world, BlockPos pos, BlockState state, boolean bl, int distance) {
    // bl，即上文中控制 "先检查哪一方向" 的布尔值
    if (distance >= 8) {
        return false;
    }
```

```
23      int i = pos.getX();
24      int j = pos.getY();
25      int k = pos.getZ();
26      boolean bl2 = true;
27      RailShape railShape = state.get(SHAPE);
28      switch (railShape) { // 根据当前铁轨形状移位检查，由 bl 可知铁轨递归检查总是先检查 - x（西）或 + z
        （南）方向
29          case NORTH_SOUTH: {
30              if (bl) {
31                  ++k; // 铁轨南北向，则先 z 增（向南）
32                  break;
33              }
34              --k; // 若南侧检查失败则 z 减（回过来向北），若南侧成功则熔断不再向北
35              break;
36          }
37          case EAST_WEST: {
38              if (bl) { // 同上，铁轨东西向，则先 x 减（向西）
39                  --i;
40                  break;
41              }
42              ++i; // 同上，x 增（向东）
43              break;
44          }
45          case ASCENDING_EAST: { // 铁轨东侧上升，即东高西低的斜铁轨
46              if (bl) {
47                  --i; // 因为西低，所以 - x 西侧可能存在同 y 的铁轨连接
48              } else {
49                  ++i; // 因为东高，所以 + x 东侧不可能存在同 y 的铁轨连接
50                  ++j; // 所以 y + 1，东侧若有连接那么铁轨必然高 1 格
51                  bl2 = false; // 上文中步骤 4 的 "当前铁轨能否可能在当前检查方向上与当前方向的下方的一个
                    铁轨相连" 的判断，这一判断默认是成立的。
52                  // 举例来说，一个 y = 1 的平放铁轨，如果边上有一个下降的铁轨与它相连，那么这个铁轨一定比
                    它低一格。
53                  // 而此时检查东侧，因为东高，所以不可能东侧有低一格的铁轨相连。判断变为 false
54              }
55              railShape = RailShape.EAST_WEST; // 上文种步骤 5 的形状更改
56              break;
57          }
58          case ASCENDING_WEST: { // 同理
59              if (bl) {
60                  --i;
61                  ++j;
62                  bl2 = false;
63              } else {
64                  ++i;
65              }
66              railShape = RailShape.EAST_WEST;
67              break;
68          }
69          case ASCENDING_NORTH: { // 同理
70              if (bl) {
71                  ++k;
72              } else {
73                  --k;
74                  ++j;
75                  bl2 = false;
76              }
```

```
            railShape = RailShape.NORTH_SOUTH;
            break;
        }
        case ASCENDING_SOUTH: { // 还是同理
            if (bl) {
                ++k;
                ++j;
                bl2 = false;
            } else {
                --k;
            }
            railShape = RailShape.NORTH_SOUTH;
        }
    }
    if (this.isPoweredByOtherRails(world, new BlockPos(i, j, k), bl, distance, railShape)) {
        return true;
        // 注意这里调用的不是当前代码块的这一函数。java 有通过入参类型匹配不同函数的性质。
        // 这里调用的是下文中的 isPoweredByOtherRails(02)
    }
    return bl2 && this.isPoweredByOtherRails(world, new BlockPos(i, j - 1, k), bl, distance,
    railShape);
    // 同上，调用的也是(02)
    // 这里的 bl2 也就对应上文中步骤 8。j-1 也就是 y 坐标向下移了一格。
}
// 便于区分，这块函数标记为 isPoweredByOtherRails(02)
protected boolean isPoweredByOtherRails(World world, BlockPos pos, boolean bl, int distance,
RailShape shape) {
    // 这里 RailShape shape 是调用这一函数的铁轨的 shape，也就是上一个铁轨的 shape
    BlockState blockState = world.getBlockState(pos);
    if (!blockState.isOf(this)) { // 略
        return false;
    }
    RailShape railShape = blockState.get(SHAPE); // 略
    if (shape == RailShape.EAST_WEST && (railShape == RailShape.NORTH_SOUTH || railShape ==
    RailShape.ASCENDING_NORTH || railShape == RailShape.ASCENDING_SOUTH)) {
        return false; // 上一个铁轨与当前铁轨不相连，那么不可能被简介充能。对应步骤 6.ii.
        // 在 isPoweredByOtherRails(01)中也就是上文步骤 5 有形状更改，形状更改的影响在这里就体现了
        // 盲猜本意是为了便于检查，但是这导致斜铁轨由于被改变形状了，导致一个平铁轨可能与看似不相连的斜铁
        轨判断为相连
        // 而且这种判断是单向的，因为铁轨递归判断不存在 + y 的行为。例子见后图
    }
    if (shape == RailShape.NORTH_SOUTH && (railShape == RailShape.EAST_WEST || railShape ==
    RailShape.ASCENDING_EAST || railShape == RailShape.ASCENDING_WEST)) {
        return false; // 同上
    }
    if (blockState.get(POWERED).booleanValue()) { // 如果铁轨是亮的(这里不是 "被直接/间接" 充能的判
    断，而是只要铁轨是亮的就行(nbt 中 powered = true)
        if (world.isReceivingRedstonePower(pos)) {
            return true; // 一直到找到被直接充能的铁轨才结束。这里就是整个递归判断为 true 的返回点
        }
        return this.isPoweredByOtherRails(world, pos, blockState, bl, distance + 1);
        // 这里回到 isPoweredByOtherRails(01)，distance + 1，递归。
    }
    return false;
}
```



A flat rail may be considered connected to a diagonal rail that appears unconnected. With the same structure, if a redstone block directly powers the flat rail, the diagonal rail will not be powered

5 Lit Observer Placement Without Neighbor NC Updates

When a piston pushes a lit observer into position and there is no observer scheduled tick at the destination position, it will not emit the conventional NC update triggered by piston movement to neighboring blocks (except the block pointed to by the output face) *(referring to the NC update triggered by* `setBlockState` *with* `flags=3` *that normally occurs after a piston pushes a regular block into position)*. The principle is as follows:

1. Placement has two cases:
    1. Normal placement: Calls `PistonBlockEntity.tick`, which calls `setBlockState` with `flags=67`
    2. Forced placement: Calls `PistonBlockEntity.finish`, which calls `setBlockState` with `flags=3`
2. In `setBlockState`, calls `worldChunk.setBlockState`, then calls `onBlockAdded`
3. In `ObserverBlock.onBlockAdded`:

1. Checks if the old block at the current position is an observer. Since the old block is b36, judgment is `true`, proceed to next step
2. Checks if itself is activated and there is no observer scheduled tick at the current position. Since itself is lit `powered=true` and there is no observer scheduled tick at the current position, judgment is `true`, proceed to next step
3. `setBlockState` with `flags=18`, changes its activation state from `powered=true -> powered=false`, and emits PP update
4. Emits conventional observer NC updates, i.e., emits NC updates to the block pointed to by the output face and neighboring blocks of the pointed block

4. Since in step 3.iii, the observer's `powered` property changed, step 1's `setBlockState` continues, but because `blockState2 == state` (i.e., whether the state changed) is `false`, `setBlockState` no longer emits conventional NC updates and ends directly

Call stack:

```PERL
PistonBlockEntity.tick(...) | PistonBlockEntity.finish(...)
└── world.setBlockState(pos, state, flags=67) | world.setBlockState(pos, state, flags=3)
    ├── // state: Block{minecraft:observer}[facing=..., powered=true]
    ├── worldChunk.setBlockState(pos, state, (flags & 64) != 0)
    │   ├── BlockState blockState = chunkSection.setBlockState(j, k, l, state); // Block{minecraft:moving_piston}[..]
    │   └── state.onBlockAdded(this.world, pos, blockState, moved)
    │       └── ObserverBlock.onBlockAdded(state, this.world, pos, oldState, moved)
    │           ├── if (!state.isOf(oldState.getBlock())): // true
    │           │   // state: Block{minecraft:observer}[...], oldState: Block{minecraft:moving_piston}[..]
    │           │   └── if (!world.isClient() && (Boolean)state.get(POWERED) && !world.getBlockTickScheduler().isQueued(pos, this)): // true
    │           │       ├── world.setBlockState(pos, state.with(powered=false), flags=18)
    │           │       │   ├── 状态变为 powered = false
    │           │       │   └── 发出 PP 更新
    │           │       └── this.updateNeighbors(world, pos, blockState) // 对输出端指向及指向的毗邻发出 NC 更新
    │           └── return
    ├── BlockState blockState2 = getBlockState(pos) // Block{minecraft:observer}[facing=..., powered=false]
    └── if (blockState2 == state): // false , powered 不同
        └── 跳过 NC 更新和 PP 更新，结束
```

Footnotes

1. Manhattan distance is the sum of distances along each coordinate axis. In Minecraft's three-dimensional space: `d=|x|+|y|+|z|` ↩
2. Short-circuit evaluation is a "circuit breaker" mechanism when computers evaluate logical expressions. For example, in `A and B`, if A is false, the logical judgment short-circuits, returns `False`, and no longer judges whether B is true; in `A or B`, if A is true, the logical judgment short-circuits, returns `True`, and no longer judges whether B is true. ↩

CHAPTER 2

2

Micro Timing

IN THIS CHAPTER

2.1 Tick and Inter-Tick Timing
2.2 A First Look at Intra-Tick Timing
2.3 Scheduled Ticks and Scheduled Tick Components
2.4 Block Entities
2.5 Block Events

2.1 Tick and Inter-Tick Timing

> We have already learned about update theory in the previous article, but curious Little B decided not to stop there. He took out the ~~Redstone Repeater~~ Repeater from the creative inventory and pulled down a lever.
>
> Perhaps Little B was being overly sensitive, but when he pulled the lever, he noticed that the Repeater didn't seem to light up immediately. He thought the game had a bug and tried several more times. He quickly realized this was a stable, reproducible phenomenon. Clearly, unlike the numerous instantaneous phenomena we discussed in the previous article, an **asynchronous** phenomenon appeared here, which he called **delay**.

1 Game Tick, Redstone Tick, and Delay

In Minecraft, many events need to be processed. These events are processed sequentially in a large loop, which we call the game's main loop. When the game is running normally (without the */tick* command), this main loop cycles approximately **20 times per second**. We call this minimum interval a **GameTick**, abbreviated as **gt**. For historical reasons, we still use the notation **RedstoneTick** in some cases, abbreviated as **rt**. It has the following conversion relationship with gt: $1rt = 2gt$.

This means:

1. 1gt = $\frac{1}{20}$ seconds = 0.05 seconds
2. Even if Minecraft has sufficient computing power and completes all the calculations needed for this gt in less than 0.05 seconds, it will stop calculating until the next gt arrives.

Therefore, players use two metrics to measure lag: **TPS** and **mspt**. TPS stands for Tick Per Second and refers to the number of game ticks executed per second. Normally, this value is 20, but when the game lags, it drops below 20. mspt stands for Millisecond Per Tick and is the number of milliseconds the game needs to execute each tick. The lower the value, the less the game lags.

Note that mspt always maintains the following relationship with TPS:

- When mspt is below 50, TPS is 20
- When mspt is greater than 50, TPS = $\frac{1000}{mspt}$

In the game, most redstone components have **delayed** responses. We measure a component's **macroscopic** delay in **gt**. The delays of some components are as follows:

- Repeater: 2-8gt
- Comparator: 2gt
- Observer: 2gt
- ...

2 Repeater and Comparator Basics

Repeaters and Comparators have macroscopic delays. The delay of a Repeater is related to its gear setting, while the delay of a Comparator is always 2gt.

The basic function of a Repeater is to amplify signals. It converts any redstone signal with non-zero strength into a strength-15 signal after the specified number of game ticks.

When a Repeater's side is directly powered by an adjacent Repeater or Comparator, it enters a "locked" state. In this state, any changes to its input do not affect its on/off state until the side input ends. After the corresponding gt delay, if the state needs to change, it updates accordingly.

A Comparator has three inputs: the front and two side inputs. When there is no input on the sides, it directly outputs a signal matching the front input's signal strength. When there is input on its sides, the Comparator's behavior depends on its mode. The Comparator has two modes:

- Compare mode
- Subtract mode

By default, it uses **Compare mode**. When the Comparator is in **Compare mode**, it always follows this logic for output:

> Take the maximum of the left and right inputs. If this maximum is greater than the front input, no output is produced; otherwise, output the front input.

When the Comparator is in **Subtract mode**, it always follows this logic for output:

> Take the maximum of the left and right inputs. If this maximum is greater than the front input, no output is produced; otherwise, output (front input - maximum of side inputs).

The interaction between Comparators and containers is not discussed here.

3 Charging Theory

We establish charging theory to explain a phenomenon that occurs with components such as Repeaters and Comparators, where blocks power Redstone Dust or other Repeaters/Comparators.

We define that for any **chargeable** block, it has two charging states: **strong charging** and **weak charging**.

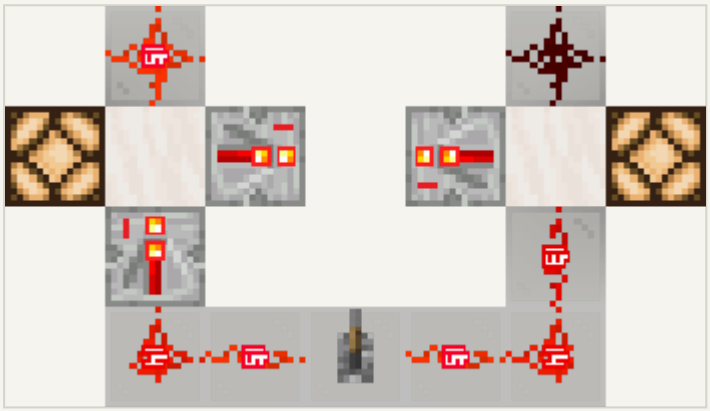
When a chargeable block is strongly charged, it can activate Redstone Dust in all six adjacent directions, regardless of orientation. When weakly charged, it can only activate Repeaters, Comparators, and some other redstone components whose inputs face the chargeable block. This is also an important way to distinguish between the two types of charging.

The following methods can make a block strongly charged:

- Repeaters, Comparators, Redstone Torches, and Observers directly emit signals toward the chargeable block.

The following methods can make a block weakly charged:

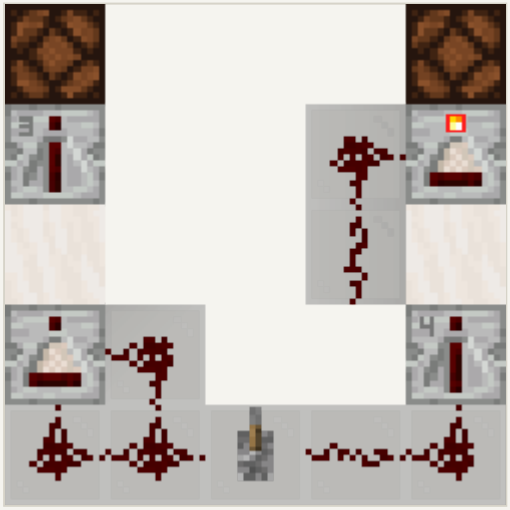
- A chargeable block is pointed to by Redstone Dust with a signal strength not equal to 0.



In Minecraft, most blocks are chargeable, so we distinguish chargeable from non-chargeable blocks by whether they cut lines. Cutting lines is when a block cuts off vertically connected Redstone Dust. However, some blocks, such as glass, slabs, half-slabs, chests, etc., do not cut lines, making them non-chargeable. We define Pistons/Sticky Pistons as non-chargeable blocks as well.

4 Inter-Tick Timing

> This section explains basic timing analysis.



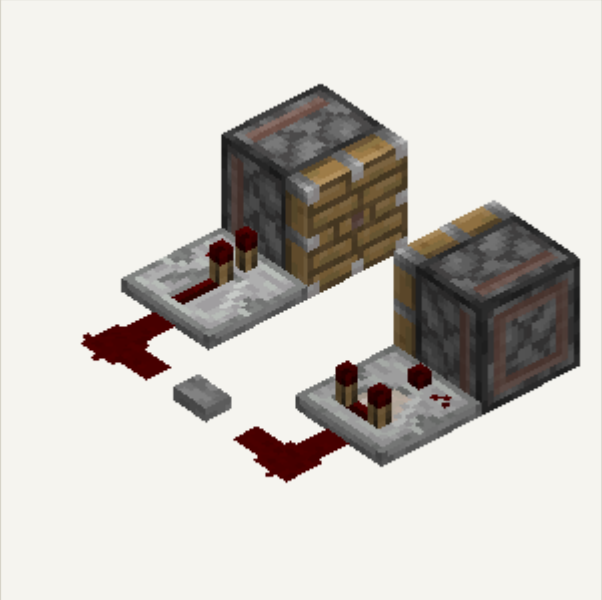
This example uses gt to analyze simple timing.

As shown in the figure, when the lever is pulled down, the Repeater on the right activates after a delay of 8gt and strongly charges the block. Now, the Comparator's side input in Subtract mode is 14, while the front input is 15. 15-14=1>0, so it activates the redstone lamp after a delay of 2gt, for a total delay of 10gt.

The Comparator on the left is in Compare mode. Its front and side inputs are both 14. Since they're equal, after a delay of 2gt, it outputs a strength-14 signal, strongly charging the block in front of it. Then the Repeater activates the redstone lamp after a delay of 6gt, for a total delay of 8gt.

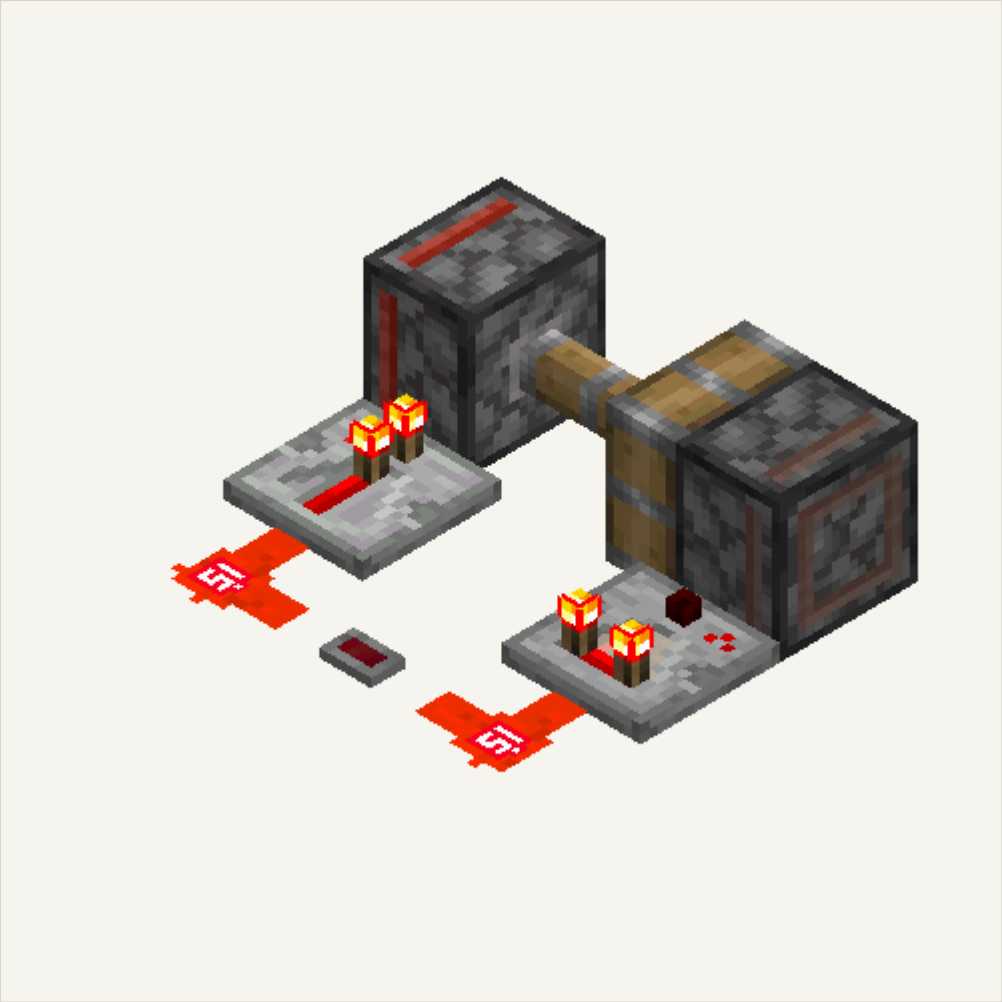
2.2 A First Look at Intra-Tick Timing

1 Observing Intra-Tick Timing

> One day, Little B built a contraption:
>
> 

Can you guess what will happen when the button is pressed?

1. Nothing will happen
2. The left piston will extend
3. The right piston will extend
4. Both pistons will extend simultaneously
5. The game will crash

> So Little B pressed the button
>
> 
>
> The... the left piston extended!!!!!!
>
> Based on what they learned from 01-Ticks and Inter-Tick Timing a few days ago, they worked through the following calculation:
>
> This repeater has a 2gt delay, that comparator has a 2gt delay...? Wow, both pistons should extend simultaneously! So why did only one piston extend? They're clearly in the same gt but there's a sequence, isn't gt the smallest unit anymore?
>
> Faced with this question, the extremely clever Little B came to an astonishing conclusion: ~~The game has a bug!~~

Of course not! Every player should understand this: in Minecraft, most game logic runs on a single thread. That means, from a logical standpoint, events always execute in some order, never truly at the same time!

So it all adds up. Within a single 1gt, there must be some finer timing at work. We call this fine-grained timing within one tick **intra-tick timing**. As shown above, running a test like this—~~where things theoretically happen in the same gt yet still have a sequence~~—is how you **observe intra-tick timing**.

Here's another example:
When the button is pressed, the two contraptions show the following behavior:



▶ This figure is animated. View original

Both are clearly in the same gt, yet adding a single repeater changes everything. What's going on? This article takes a closer look at these phenomena.

You might be wondering: what's the practical use?

All sorts of things! Whether you're building extremely complex redstone contraptions or doing basic precise timing analysis, intra-tick timing comes into play. Research into intra-tick timing theory has also pushed redstone research forward in major ways, enabling upper limit detection, BED, and other **advanced** redstone contraptions and theories.

Here's a passage from Mr. Void:

> ...
> Learning intra-tick timing is necessary not just to solve design problems or explain previously mysterious observations. More importantly, many people underestimate intra-tick timing and refuse to study it, seeing it only as a source of headaches. They run into intra-tick timing issues in redstone design all the time, yet they don't realize that intra-tick timing is one of the most powerful tools available. Using it well can greatly simplify redstone circuit design and improve machine performance. Without this knowledge, many people never even realize the tool exists. For the sake of healthy community growth, this situation should be addressed.

2 Micro-Timing Theory

2.1 Phase Division of Micro-Timing

Think of **1gt** like **one day** in the real world. Within a day, things happen in a set order: waking up, breakfast, lunch, dinner, sleeping... In Minecraft, timing measured in whole 1gt units is called macro timing, while the finer subdivisions within a single 1gt are called micro-timing.

As we established earlier, MC always executes things in a fixed priority order. Through source code analysis, experiments, and other methods, we've found that Minecraft roughly follows this sequence within each 1gt:

1. **World Tick Update**, abbreviated as WTU. The game maintains a "timer" bound to the world. When the world is created, this "timer" starts at 0. During **this phase**, the "timer" increments by 1. The nth tick is the **World Tick Update phase** that brings the "timer" to n.
2. **Schedule Tick / Tile Tick / Next Tick Entry***, abbreviated as TT/NTE*. Most components with delayed execution behavior are controlled by "Scheduled Tick" and executed during the **Scheduled Tick phase**. *(The mention of NTE and the **incorrect** translation of Next Tick Entry is included to help readers understand some early documents; we don't recommend continuing to use these two names.)*
3. **Chunk Tick**, abbreviated as CT. Chunk Tick covers **lightning, snow**, and **Random Tick**. During this phase, the game first handles **lightning, snow**, and similar events, then processes **Random Tick**: when a player is nearby, "crop growth", "grass block spread", "water freezing", and other random tick events may occur. In each **Chunk Tick phase**, the game iterates through all chunks near the player, then randomly selects blocks within those chunks to trigger these events.
4. **Block Event**, abbreviated as BE. Things like **piston pushing/pulling and note blocks playing sounds** happen in this phase. Take pistons as an example: when a piston detects that its [**actual state**] doesn't match its [**powered state**], it adds a "**Block Event**". "**Block Events**" added during the **Block Event phase** execute in that same phase, while "**Block Events**" added during other phases are queued for the next **Block Event phase**. Continuing the piston example, during **this phase** the piston places moving blocks (b36) at the target positions of blocks it's about to push or pull.
5. **Entity Update**, abbreviated as EU. Every tick, entities **handle movement, AI, and other behaviors**. All **entity behaviors**, such as "creature movement", "TNT explosions", and "monster attacks", occur in **this phase**. Non-player "activation" of pressure plates and tripwires, which produces redstone signals, also happens in this phase.

6. **Block Entity / Tile Entity**, abbreviated as TE. Some blocks need to **run their own logic** every tick, and this happens in the **Block Entity phase**. During the **Block Entity phase**, hoppers "absorb" and "transfer" items. Blocks pushed by pistons become moving blocks (b36), which execute "push entity" logic during the first two **Block Entity phases** after creation, then revert to normal blocks on the third **Block Entity phase**.
7. **Async Task / Network Update**, abbreviated as AT/NU, also called Player Action. "Player actions" are actually network packets sent from the client to the server. During **this phase**, at the end of each tick, the server will uniformly execute all "player action" packets received during this tick.

These are the main game phases you'll encounter in typical analysis. The rest of this article builds on these events.

2.2 Instant

As we've seen, 1gt is divided into different phases. Just as breakfast is for morning and lunch is for noon, many events in Minecraft can only happen in specific phases and follow a set order. But some events respond "immediately", the way you can grab a drink whenever you're thirsty, regardless of the time of day.

If a block's behavior can occur in any phase and is triggered only by **block updates**, it's called an **instant component**.

For example, redstone dust turning on or off is instant. If a player flips a lever, the redstone dust can turn off during the player action phase; if a piston pushes away a redstone block, the dust can turn off during the block event phase.

2.3 Delayed

If there are instant events, there must also be "delayed" ones. For example, if a player places a redstone block that activates a repeater, the repeater will light up 2gt later. This lighting has a **delay**, is **controlled by the game** (via scheduled ticks), and occurs in a **specific phase** (the Scheduled Tick phase), so it's not an instant event.

In the following sections, we'll go into detail about each of these **game phases**.

2.4 Common Components and Their Operating Phases

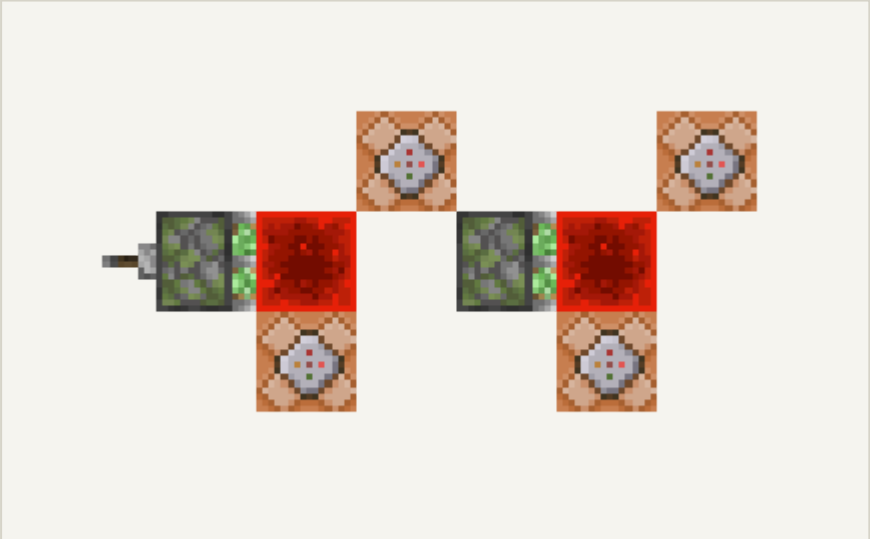
COMPONENT TYPE	OPERATING PHASE
Command block runs command	Scheduled Tick TT
Repeater, Comparator, Redstone Torch, Observer on/off	Scheduled Tick TT
Redstone Dust, Rails change state	Instant
Fence Gates, Trapdoors change state	Instant
Hopper changes state due to redstone signal	Instant
Hopper absorbs, transfers items	Block Entity TE

COMPONENT TYPE	OPERATING PHASE
Note Block, Bell changes state due to redstone signal	Instant
Note Block, Bell makes sound	Block Event BE
Dispenser, Dropper changes state due to redstone signal	Instant
Dispenser dispenses/Dropper drops items	Scheduled Tick TT
Redstone Lamp lights up	Instant
Redstone Lamp turns off	Scheduled Tick TT
Button, Pressure Plate, Tripwire lights up	Instant[1]
Button, Pressure Plate, Tripwire turns off	Scheduled Tick TT
Falling block determines fall, creates entity	Scheduled Tick TT
Falling block falls, lands	Entity Update EU
Piston extends or retracts	Block Event BE
b36[2] pushes entity	Block Entity TE
b36 naturally lands	Block Entity TE
b36 retracted and landed by sticky piston[3]	Block Event BE

2.5 Basic Analysis of Intra-Tick Timing

Let's walk through a basic example.

In the figure below, every command block runs the command `time query gametime`



Rising edge *(Pull down lever, sticky piston extends)*:

TICK	PHASE	EVENT
0	AT	Player pulls down lever
0	AT	First piston adds block event
1	BE	First piston extends
3	TE	First redstone block lands
3	TE	Command block schedules to run at tick 4 (1 tick delay)
3	TE	Second piston adds block event
4	TT	First command block reads 4
4	BE	Second piston extends
6	TE	Second redstone block lands
7	TT	Second command block reads 7

Falling edge *(Pull back lever, sticky piston retracts)*:

TICK	PHASE	EVENT
0	AT	Player turns off lever
0	AT	First piston adds block event
1	BE	First piston retracts
1	BE	First redstone block is removed
1	BE	Second piston adds block event
1	BE	Second piston retracts
3	TE	Both redstone blocks land
4	TT	Both command blocks read 4

(Adapted from an example by void)

Footnotes

1. However, pressure plates can only be activated by entity movement, so in practice they only trigger during Entity Update or Player Action. Similarly, buttons can only be pressed by players or arrows, so they're also limited to these two phases. ↩
2. b36: minecraft:moving_piston, moving piston. ↩
3. When a sticky piston retracts and there's b36 in front of it, the b36 is immediately placed in its final position. This is what's known as a sticky piston short pulse. ↩

2.3 Scheduled Ticks and Scheduled Tick Components

This section explains how Scheduled Ticks and Scheduled Tick Components work.

Basic Section

- Concept and content of Scheduled Ticks
- Common Scheduled Tick Components
- Simple examples

Advanced Section

- Maintenance of Scheduled Tick queues
- Timing analysis of 4gt Observer high-frequency circuits
- In-depth analysis of Comparator Scheduled Tick execution logic
- Scheduled Tick suppression

1 Concept and Content of Scheduled Ticks

1.1 What is a Scheduled Tick?

In Minecraft, many redstone components don't change immediately after being triggered, such as repeaters and comparators. These components always change state after a delay.

Here's a real-world analogy. In the morning, you receive an email, so you plan to handle it in the afternoon and set an alarm. When afternoon comes, the alarm rings, you remember you need to handle the email, so you open your inbox and start processing it.

Note: **This "alarm" carries no notes or labels. It only reminds you on your phone, at the correct time, that "something needs doing". As for what specifically needs to be done, the alarm doesn't care.** This alarm contains only these elements:

- When to ring
- Which alarm number this is
- How important it is
- On whose phone it rings
- Who to remind

A Scheduled Tick works like this alarm. When a redstone component is triggered, it adds a Scheduled Tick for itself. When the Scheduled Tick executes, it's like the alarm ringing, and the redstone component changes state.

Redstone components whose behavior is controlled by Scheduled Ticks are called **Scheduled Tick Components**.

1.2 Content of Scheduled Ticks

Like the alarm above, **a Scheduled Tick contains only these elements**, which we call the **information structure of a Scheduled Tick**:

- Execution time `triggerTick`: when to execute, or how long to delay before execution[1]
- Sub-order `subTickOrder`: the order in which Scheduled Ticks are added
- Priority `priority`: how high the Scheduled Tick's priority is
- Position `pos`: the coordinates where execution occurs
- Block type `type`: which block type executes this Scheduled Tick

Position and **block type** are straightforward attributes. A block's own Scheduled Tick cannot be executed by other blocks ~~(unless you happen to push it away or break and replace it with another block before it executes its Scheduled Tick)~~, nor can it execute somewhere else in the world.

Execution time refers to when the Scheduled Tick executes at the macro timing level, i.e., which gt the Scheduled Tick should run in. For example, when a repeater set to 1 tick is triggered, it adds a Scheduled Tick for 2gt later, and after 2gt, that Scheduled Tick will execute. What we commonly call "the delay of a Scheduled Tick component" refers to how many gt later the Scheduled Tick executes.

Sub-order refers to the order in which Scheduled Ticks are added at the same time. For example, within the same gt, repeater A is triggered first and repeater B is triggered later. Then, in the sub-order, repeater A comes before repeater B.

Priority is an integer from -3 to 3[2], where smaller values mean higher priority. That is, within the same gt, a Scheduled Tick with priority -3 always executes before those with priority -2, -1, or 0.

> When discussing priority, "higher priority" and "lower priority value" mean the same thing. To avoid ambiguity caused by the concept of "priority" and "priority value", we recommend readers use "a certain component's Scheduled Tick is more prioritized" when explaining to others.

Similarly, just like the alarm above: **A Scheduled Tick is just an "alarm reminding the block that something needs to be done". It doesn't care what the block actually does. All behavior when executing a Scheduled Tick is controlled by the block itself; "the behavior of executing a Scheduled Tick" is not part of "the information structure of a Scheduled Tick".**

When we say "a certain component already has a Scheduled Tick", we mean that at the current position, there exists an unexecuted Scheduled Tick with the same block type as the current block.[3]

1.3 Execution Order of Scheduled Ticks

In the previous two articles, we learned about macro timing analysis and became familiar with micro timing. We know that **macro timing always takes precedence over micro timing**, and the same applies to Scheduled Ticks. For Scheduled Ticks with different execution times, those with earlier execution times al-

ways execute first. For Scheduled Ticks with the same execution time, those with higher priority always execute first. For Scheduled Ticks with the same priority, those added earlier (smaller sub-order values) always execute first.

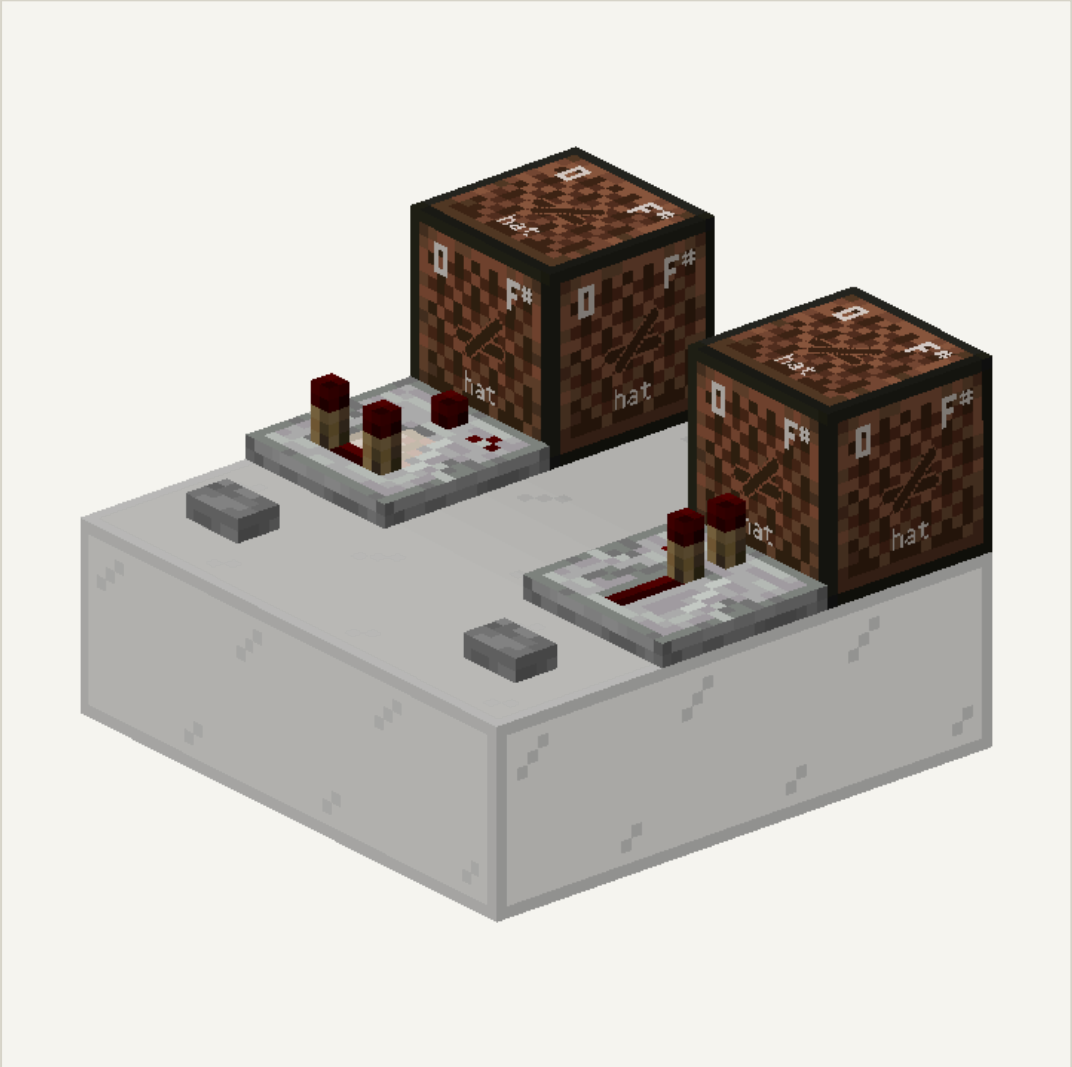
Therefore, when comparing the execution order of Scheduled Ticks, we can follow this logic:

1. **Compare macro timing**: earlier macro timing executes first.
2. **Compare Scheduled Tick priority**: higher priority executes first.
3. **Compare Scheduled Tick addition order (sub-order)**: earlier addition executes first.

This is like comparing numbers: macro timing is the hundreds place, priority is the tens place, and addition order is the ones place.

1.4 Example of Scheduled Tick Order

Now, let's look at a practical example.



Given the situation shown in the diagram, the comparator has priority 0 and the repeater has priority -1. Both the comparator and repeater have a delay of 2gt.

1. If triggered in different game ticks, you press the comparator's button first, then press the repeater's button. Which note block will sound first?
2. If triggered in the same game tick, you press the comparator's button first, then press the repeater's button. Which note block will light up first?

Answers:

1. The comparator lights up first. In macro timing, the comparator is triggered first, and then after a certain number of gt, the repeater lights up.
2. The repeater lights up first. In terms of Scheduled Ticks, although the comparator adds its Scheduled Tick before the repeater, the repeater's priority is -1, which is more prioritized than the comparator's priority 0. Since Scheduled Tick priority takes precedence over Scheduled Tick addition order, the repeater lights up first.

2 Common Scheduled Tick Components

2.1 Repeater

Repeaters and comparators are collectively called **Redstone Diodes** or **Redstone Gates**.[4]

In Tick and Inter-Tick Timing, we already introduced repeaters and comparators. Now, let's examine the Scheduled Tick behavior of repeaters in more detail.

If a repeater is locked, it will not add a Scheduled Tick, nor will it change state when executing a Scheduled Tick. If a repeater is not locked, it has the following behaviors:

Adding Scheduled Tick behavior:

- When a repeater receives an NC update, it checks its own state. If it does not have a Scheduled Tick and should change state *(i.e., it is not lit but has a redstone signal at the input, or it is lit but has no redstone signal at the input)*, then it adds a Scheduled Tick.
- All Scheduled Ticks added by repeaters have a delay of `repeater delay * 2 gt`, except for the following case:
- When a repeater is placed, it checks its own state, and if it should change state, it adds a Scheduled Tick for 1gt later.

Executing Scheduled Tick behavior:

- If the repeater is lit, it immediately turns off.
- If the repeater is not lit, it immediately lights up. This step is not affected by the input signal.
    - If there is no input signal at this time, it adds another Scheduled Tick (for turning off).

For example:

For instance, if a repeater set to 2 ticks receives a signal with `duration <= 4gt`[5], the repeater's behavior is as follows:

- Receives NC update, adds Scheduled Tick

- After 4gt, executes Scheduled Tick
  - Detects that it is not lit, so it immediately lights up
  - Detects that there is no redstone signal at the input, adds Scheduled Tick (for turning off)
- After another 4gt, executes Scheduled Tick
  - Detects that it is lit, so it immediately turns off

This example covers all Scheduled Tick behaviors of a repeater.

Special priority changes:

- If a repeater points to a horizontally placed redstone diode or points to the input of a redstone diode, the Scheduled Tick's priority is `-3`.
- Otherwise, if the repeater is in a lit state when adding a Scheduled Tick (i.e., this Scheduled Tick is for turning off), the Scheduled Tick's priority is `-2`.
- Otherwise, the Scheduled Tick's default priority is `-1`.

2.2 Comparator

Adding Scheduled Tick behavior:

- When a comparator receives an NC update, it checks its own state. If it does not have a Scheduled Tick and should change state *(i.e., it is not lit but has a redstone signal at the input, or it is lit but has no redstone signal at the input)*, then it adds a Scheduled Tick.
- All Scheduled Ticks added by comparators have a delay of `2gt`, except for:
- When a comparator is placed, it checks its own state, and if it should change state, it adds a Scheduled Tick for 1gt later.

Executing Scheduled Tick behavior:

When a comparator executes a Scheduled Tick, it actually only does one thing:

- Update its own output power level based on the comparator's current input state and comparator mode.

The **output power level** refers to the output power level calculated by the comparator *(specific calculation method described later)*. This **update** includes two things: **updating the power level** and **sending updates**. Moreover, for comparators, the output power level may appear unchanged from the outside, but in reality the comparator still "updates" its own output power level.

Comparators also have slightly more complex update behavior:

Comparator's update sending behavior:

1. When executing a Scheduled Tick, if the comparator's powered property changes *(i.e., it changes from off to lit, or from lit to off)*, it first sends an NC update, then sends a PP update.
2. If the comparator is in compare mode, or the comparator is in subtract mode and its own output power level changes, it sends an NC update.

For example:

- First, this entire section assumes the comparator **actually executed a Scheduled Tick**:

- As long as the comparator is in compare mode, even if its output level doesn't change, including changing from 0 to 0, there will be one NC update.
- If the comparator's output power level changes from 0 (off) to 1, 2, 3, 4, 5...14, 15 (lit), or from lit to off, it will first send one NC update, then send one PP update, and finally send another NC update.
- If the comparator is in subtract mode, and its output power level changes from 1, 2, 3, 4, 5... to any other power level, there will only be one NC update.

These behaviors may be useful in some special wiring setups. If readers find this section difficult to understand, they can simply remember the most common uses of comparators: comparing signal sizes and determining whether to output, detecting container capacity, and subtract mode, or refer to the table.

COMPARATOR MODE	SCHEDULED TICK BEHAVIOR	UPDATE BEHAVIOR
Compare Mode	Powered property changes	NC->PP->NC
Compare Mode	Powered property unchanged	NC
Subtract Mode	Powered property changes	NC->PP->NC
Subtract Mode	Power level changes, powered property unchanged	NC
Subtract Mode	Both power level and powered property unchanged	No update

Comparator Output Power Level Calculation

If the signal input is redstone dust or another comparator, the comparator inherits the input power.

If it's a container, refer to Comparator Signal Strength Calculation.

In particular, when there are both container signals and redstone signal inputs, the comparator will prioritize different signal inputs for calculation based on the situation. This phenomenon is also known as container shielding.

INPUT	OUTPUT
Comparator input directly connected to container	Only calculates container input, ignores redstone signal
Comparator input detects container through solid block	When redstone signal is 15, outputs 15 signal strength; otherwise prioritizes container input calculation

Case (barrels have no items):



Special priority changes:

- If a comparator points to a horizontally placed redstone diode or points to the input of a redstone diode, the Scheduled Tick's priority is `-1`
- Otherwise, the Scheduled Tick's default priority is `0`

At this point, we can analyze the case from the previous chapter.

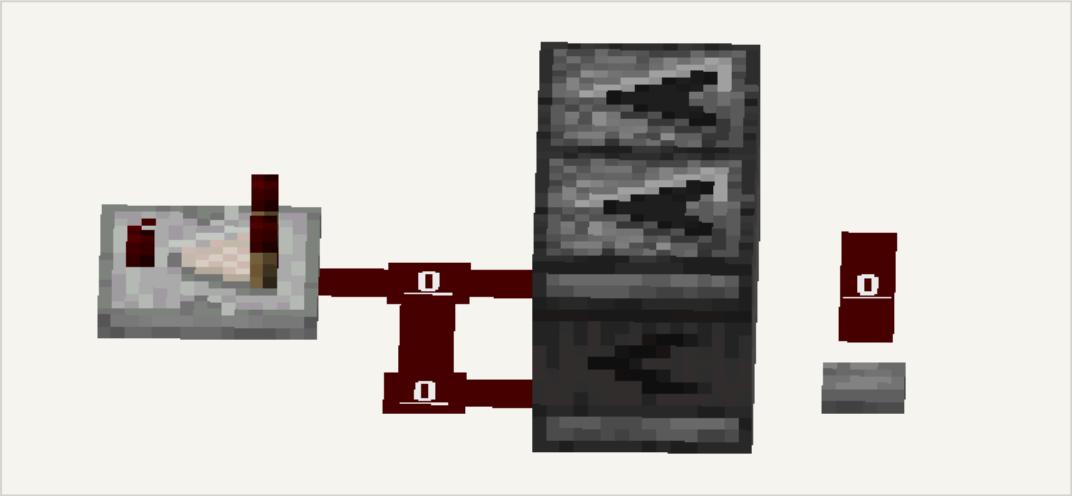


▶ This figure is animated. View original

	LEFT SIDE	RIGHT SIDE

gt0	Button pressed, rail activated Observer adds 2gt Scheduled Tick	Button pressed, rail activated Observer adds 2gt Scheduled Tick
gt2	Observer lights up Observer adds 2gt Scheduled Tick (priority 0) Comparator adds 2gt Scheduled Tick (priority -1)	Observer lights up Observer adds 2gt Scheduled Tick (priority 0) Comparator adds 2gt Scheduled Tick (priority 0)
gt4	Comparator Scheduled Tick priority -1, Observer Scheduled Tick 0, Comparator Scheduled Tick executes first, Comparator lights up Observer turns off	Since Observer and Comparator have the same priority, Observer Scheduled Tick was added first, Observer Scheduled Tick executes first, Observer turns off Comparator executes Scheduled Tick, since Observer has already turned off, Comparator does not light up

Readers can analyze the following similar case on their own: after pressing the button, will the comparator light up?



2.3 Observer

Adding Scheduled Tick behavior:

- After an observer receives a PP update from the block in front of it, if it is not lit and there is no Scheduled Tick for itself at the current position[6], it adds a Scheduled Tick for 2gt later.

Executing Scheduled Tick behavior:

- If it is not lit, it lights up, sends a PP update, then adds a Scheduled Tick for 2gt later, and finally sends an NC update.
- If it is lit, it turns off.

2.4 Redstone Torch

Adding Scheduled Tick behavior

- After a redstone torch receives an NC update, if it is lit but should turn off and there is no Scheduled Tick for itself at the current position, it adds a Scheduled Tick for 2gt later.

Executing Scheduled Tick behavior

- If it is lit but should turn off, it turns off.
  - If it **burns out**, it **turns off**[7], and adds a Scheduled Tick for 160gt later.
- If it is off but should light up, and has not burned out, it lights up.

Should turn off/light up: Whether the block the redstone torch is attached to is a solid block and receives a redstone signal.

Burn out: If a redstone torch lights up 8 times within 60gt, it burns out.

After a redstone torch burns out, you can make it "light up" by breaking and replacing it with a new redstone torch, or wait until 160gt later for the torch to execute its Scheduled Tick and light up on its own.

3 Simple Scheduled Tick Timing Analysis

3.1 Can an Observer Turn Off a Redstone Torch?

Usually not. Let's briefly analyze this:

Both observers and redstone torches have a priority of **0**. When an observer lights up, it **first sends a PP update**, then **adds a Scheduled Tick**, and finally **sends an NC update**. Redstone torches **receive NC updates**. The **order of behaviors within Scheduled Tick execution** and **what updates a component receives** fundamentally affect the following analysis. So assuming an observer is facing a solid block that a redstone torch is attached to, its Scheduled Tick order is as follows:

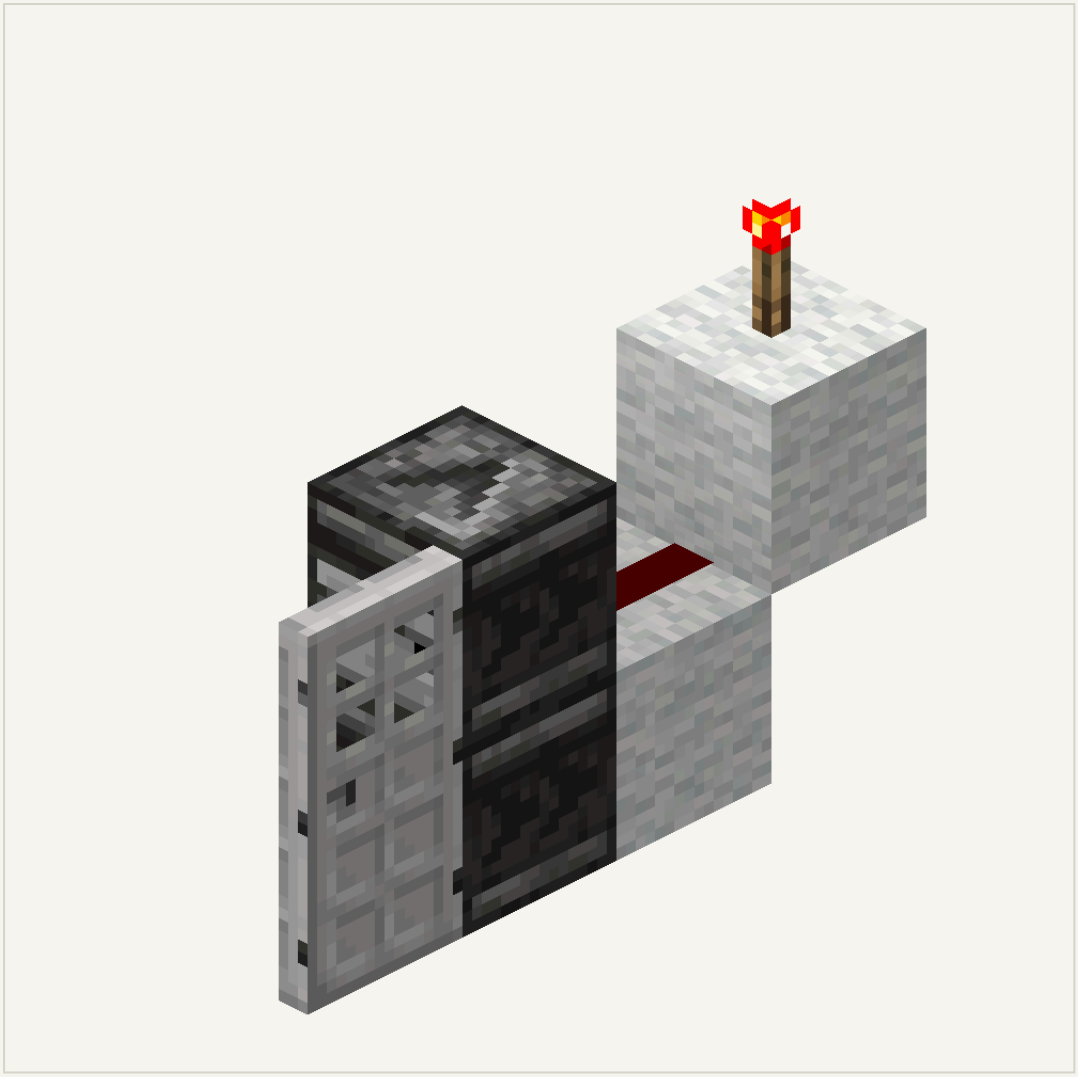
- Observer lights up:
  - Sends PP update, nothing happens.
  - Observer **adds Scheduled Tick**
  - Sends NC update:
    - Redstone torch receives NC update:
      - **Adds Scheduled Tick**

2gt later:

- Observer **executes Scheduled Tick**, turns off.
- Redstone torch **executes Scheduled Tick**, at this time checks whether it should turn off, finds it should not turn off (because the observer turned off, the attached block is not powered), so it performs no action.

So in summary: the redstone torch finds there's no power when it's about to light up, so it does nothing.

However, at this point we can discover a problem: if I place two observers, one observer used to **"trigger" the redstone torch to add a Scheduled Tick**, and another used to **"continue" the redstone signal to turn off the torch**, then can the redstone torch light up? Yes. A simple structure is shown in the figure below:



▶ This figure is animated. View original

3.2 Can an Observer Make a Comparator Light Up?

3.3 What is an "End Always-On" Repeater?

4 Maintenance of Scheduled Tick Queues

In this section's explanation, we will omit some variables that have been declared but not called or are rarely involved.

To help readers follow the **Scheduled Tick execution process** more smoothly, we will first explain all important concepts and methods involved, then explain the execution logic of Scheduled Ticks. When reading, you can first get a preliminary understanding of these concepts and methods, then refer back when encountering difficulties in understanding the Scheduled Tick execution logic.

4.1 Basics

The Scheduled Ticks of the entire world are actually controlled by a **WorldTickScheduler** (or **World-level Scheduled Tick Scheduler**)[8] during execution. The Scheduled Ticks at each position are controlled by a **ChunkTickScheduler** (or **Chunk-level Scheduled Tick Scheduler**) of the corresponding chunk when being added and stored.

4.2 Scheduled Tick Class

An object containing all information of a Scheduled Tick can be called a **Scheduled Tick object**. The Scheduled Tick class `OrderedTick` defines the properties and methods of Scheduled Tick objects. In addition to the five basic properties of block type `type`, coordinates `pos`, execution time `triggerTick`, priority `priority`, and sub-order `subTickOrder`, it also implements the following four methods:

4.2.1 Execution Time Comparison

First compare `triggerTick` values; if the same, then compare `priority` values; if still the same, then compare `subTickOrder` values.

This method implements the "priority" comparison in the **priority queue** of the **Chunk-level Scheduler** (see below).

4.2.2 Simple Comparison

Does not compare `triggerTick` values. Only compares `priority` values first; if the same, then compares `subTickOrder` values.

This method implements the "priority" comparison in the **Chunk-level Scheduler priority queue** of the **World-level Scheduler** (see below)

4.2.3 Hashing

Calculates the hash code of the Scheduled Tick object by operating on the hash codes of `type` and `pos`:

```java
31 * pos.hashCode() + type.hashCode()
```

In Java, all hash codes are integers.

4.2.4 Fast Instantiation

In addition to the regular instantiation method, the Scheduled Tick class also implements a `create` method for temporarily creating a Scheduled Tick object based only on `type` and `pos`. Its other three properties are all `0`. This temporary Scheduled Tick object is used to determine whether a Scheduled Tick for a certain block exists at a certain position.

4.3 Chunk-level Scheduler

The Chunk-level Scheduler manages the **addition**, **"checking"**, and **storage** of Scheduled Ticks. It mainly maintains a **priority queue** (`PriorityQueue`) and a **hash set** (`ObjectOpenCustomHashSet`).

4.3.1 Adding Scheduled Ticks

Attempts to add the **hashed Scheduled Tick** to the **hash set**. If the addition is successful (i.e., the Scheduled Tick does not exist in the set), it simultaneously adds the **Scheduled Tick object** to the **priority queue**.

4.3.2 Scheduled Tick Storage

All hash codes of Scheduled Ticks are stored in the hash table of the Chunk-level Scheduler of the corresponding chunk, and all Scheduled Tick objects are stored in order in the priority queue of the Chunk-level Scheduler of the corresponding chunk.

The priority queue's sorting is based on **Execution Time Comparison**, meaning it considers `triggerTick`, `priority`, and `subTickOrder` simultaneously.

4.3.3 Access Next Tick `peekNextTick`

Or "access head Scheduled Tick". This method returns the Scheduled Tick object of the next Scheduled Tick to execute in the current chunk, i.e., the first Scheduled Tick in the priority queue.

4.3.4 Poll Next Tick `pollNextTick`

Or "poll head Scheduled Tick". This method attempts to poll the Scheduled Tick object of the next Scheduled Tick to execute in the current chunk. If polling is successful, it simultaneously deletes the hash code of that Scheduled Tick object from the hash table.

4.4 World-level Scheduler

The World-level Scheduler manages the "checking" and execution of Scheduled Ticks. During the Scheduled Tick execution phase, it coordinates all Chunk-level Schedulers, polls Scheduled Ticks according to specific logic, places them into the executable Scheduled Tick list, and finally executes them all together.

See Executing Scheduled Ticks.

4.5 Checking if a Scheduled Tick Exists

This method is what we commonly call "a certain component does not have a Scheduled Tick" or "a certain component does not have a Scheduled Tick for itself at the current position". It describes two core parameters: "a certain component" — block type `type`, "itself" "current position" — coordinates `pos`.

4.5.1 Method 1: Checking Method Controlled by Chunk-level Scheduler

This is the **Adding Scheduled Ticks** method described above. Adding a hashed Scheduled Tick to the hash set is actually a kind of "checking": if a Scheduled Tick for a certain block at a certain coordinate already exists in the hash set, when this block tries to add a Scheduled Tick at that location again, it will fail.

4.5.2 Method 2: Check if Scheduled Tick is "About to Execute" `isTicking`

This method is used to check whether a component will execute a Scheduled Tick in the current gt.

It receives two parameters: block type and coordinates. It first copies the queue of Scheduled Ticks that will execute in the current gt, then checks whether there is a Scheduled Tick with the same block type and coordinates in this queue.

When executing Scheduled Ticks, all executable Scheduled Ticks of the current tick (if the Scheduled Tick limit has not been reached) will be polled from the hash set and priority queue of each Chunk-level Scheduler (see **Executing Scheduled Ticks** below). That is to say, if a component will execute a Scheduled Tick in this gt, and it happens to try to add a Scheduled Tick before execution, then this addition behavior cannot be "intercepted" by **Method 1**.

Components such as repeaters, comparators, and redstone torches use this method to check if a Scheduled Tick is "about to execute" when adding Scheduled Ticks. Taking a repeater as an example, due to the effect of this method, if a repeater will light up at gt 2, then at gt 2, before the repeater lights up, since the repeater detects that it will execute a Scheduled Tick in the current gt, no matter how the signal changes, it will not add a Scheduled Tick. At gt 0 and gt 1, if the powered state of the repeater's input changes and the repeater receives an update, it will try to add a Scheduled Tick, but due to Method 1's "checking", this addition behavior will not succeed. So Method 1 and Method 2 together ensure that the repeater will not repeatedly add Scheduled Ticks before executing a Scheduled Tick.

4.5.3 Method 3: Check if Scheduled Tick is in Queue `isQueued`

This method is the "check if Scheduled Tick exists" method called by observers, and its actual function is completely identical to Method 1.

> I really don't know what use it has other than possibly reducing lag.
> ——tanh_Heng

It receives two parameters: block type and coordinates. It first maps the coordinates to a chunk, then accesses the Chunk-level Scheduler corresponding to that chunk, then quickly instantiates a Scheduled Tick with the corresponding `type` and `pos`, and finally checks whether there is a Scheduled Tick with the same hash code as this temporary Scheduled Tick in the hash set.

Obviously, the fundamental difference from Method 2 is that this method does not check Scheduled Ticks that will execute in the current gt. This is the core principle of 4gt Observer high-frequency.

4.6 Executing Scheduled Ticks

Executing Scheduled Ticks is divided into three phases: **collection** (`collect`), **execution** (`run`), and **cleanup** (`cleanup`). The most important is the collection phase. In the collection phase, the World-level Scheduler adds Scheduled Ticks to the **executable Scheduled Tick list** (`tickableTicks`) according to the following steps:

4.6.1 Descriptive Explanation

Head Scheduled Tick of Chunk-level Scheduler: In the Chunk-level Scheduler, the Scheduled Tick at the front that will execute first.

All descriptions that do not explicitly state **polling Scheduled Ticks** are **access** methods.

Reached Scheduled Tick limit: The number of Scheduled Ticks that can be executed per gt in the game is limited. **Reached Scheduled Tick limit** means the number of Scheduled Ticks in the executable Scheduled Tick list is greater than or equal to the Scheduled Tick limit `maxmaxTicks`.

4.6.2 Step 1: Filter Executable Chunk-level Schedulers `collectTickableChunkTickSchedulers`

1. Scan all Chunk-level Schedulers (responsible for managing Scheduled Ticks within a single chunk). Only select those that:
   - Have Scheduled Ticks waiting to execute internally (Scheduled Ticks exist).
   - The execution time of the head Scheduled Tick in their queue is less than or equal to the current gt.
   - The chunk corresponding to that Scheduled Tick meets loading requirements.
2. Put these qualified Chunk-level Schedulers into a priority queue, called the priority queue of executable Chunk-level Schedulers (`tickableChunkTickSchedulers`, hereinafter referred to as the scheduler queue).

So, how does this priority queue sort? The queue sorts based on the head Scheduled Tick of each Chunk-level Scheduler. Sorting uses Simple Comparison (compared to the **Execution Time Comparison** of Chunk-level Scheduled Ticks, Simple Comparison does not consider the `triggerTick` property). This sorting mechanism is the key to implementing Scheduled Tick Suppressors later.

4.6.3 Step 2: Extract Executable Scheduled Ticks from Queue `collectNextTicks`

1. **Poll the first scheduler:** Poll a Chunk-level Scheduler from the front of the scheduler queue. We call it the **current Chunk-level Scheduler**.
2. **Poll its first Scheduled Tick:** Poll the head Scheduled Tick from the Scheduled Tick queue of the current Chunk-level Scheduler.
3. **Mark as executable:** Add this polled Scheduled Tick to the executable Scheduled Tick list, waiting for actual execution.
4. **Attempt to continuously extract more Scheduled Ticks:** Next, the game will try to continuously poll as many subsequent Scheduled Ticks as possible from the current Chunk-level Scheduler (as long as specific conditions are met):
   1. **Limit check:** If the Scheduled Tick limit allowed for this processing has been reached, stop the entire continuous extraction process.
   2. **Access next scheduler in queue:** Access the Chunk-level Scheduler at the front of the current scheduler queue (because the first scheduler has already been polled, this is the second scheduler in the original queue). We call it **Chunk-level Scheduler 2**.

3. **Check if queue is non-empty:** If the scheduler queue is not empty (i.e., successfully accessed Chunk-level Scheduler 2), then access the head Scheduled Tick of Chunk-level Scheduler 2.
4. **Start continuous polling:** If the Scheduled Tick limit has not been reached, loop through the following steps (5 and 6).
5. **Look at the next Scheduled Tick of the current scheduler:** Access the Scheduled Tick queue of the current Chunk-level Scheduler again, obtaining the Scheduled Tick currently at its front (same as step 2, because the first Scheduled Tick has already been polled, this is the one that was behind the original head Scheduled Tick).
6. **Decide whether to poll (core condition):**
    - If the current Chunk-level Scheduler still has Scheduled Ticks (successfully accessed its current head Scheduled Tick),
    - And this Scheduled Tick also needs to execute in the current game tick (gt),
    - And using the **Simple Comparison** rule, this Scheduled Tick of the current Chunk-level Scheduler has an earlier execution time than the head Scheduled Tick of Chunk-level Scheduler 2,
    - Then poll this Scheduled Tick from the current Chunk-level Scheduler and add it to the executable Scheduled Tick list.
    - If any of the above conditions is not met (no more Scheduled Ticks, not executing in current gt, or not earlier than the next scheduler's Scheduled Tick in the queue), exit this continuous extraction loop, i.e., **no longer attempt to continuously extract more Scheduled Ticks**.

5. **Handle unfinished schedulers:** When the continuous extraction loop of step 4 ends, it means that temporarily no more Scheduled Ticks meeting the conditions can be continuously polled from the current Chunk-level Scheduler. At this time, check:
    - If the current Chunk-level Scheduler still has Scheduled Ticks that have not been polled (queue not empty),
    - And its current front Scheduled Tick (new head Scheduled Tick) still needs to execute in the current game tick (gt),
    - And the entire processing has not reached the Scheduled Tick limit,
    - Then re-add the current Chunk-level Scheduler to the scheduler queue (re-sorted according to the execution priority of its new head Scheduled Tick), waiting for subsequent processing again.
6. Loop through the above steps (steps 1 to 5) until the Scheduled Tick limit is reached or the scheduler queue is empty.

In summary:

The game first filters out all Chunk-level Schedulers that have Scheduled Ticks to execute in the current gt and whose chunks are loaded, then queues them according to the Simple Comparison order of their most prioritized Scheduled Tick (head Scheduled Tick). Next, it polls a Chunk-level Scheduler from the queue head, adds its most prioritized Scheduled Tick to the ready-to-execute queue, and attempts to "continuously" extract more tasks from this scheduler that also execute in the current gt and are more prioritized than the most prioritized Scheduled Tick of the next scheduler in the queue, adding them to the ready-to-

execute queue together (to improve efficiency). If continuous extraction is interrupted (e.g., the next Scheduled Tick is not prioritized enough, or the execution limit is reached), and this chunk still has Scheduled Ticks for the current gt that have not been added and the limit has not been exceeded, the current scheduler is put back into the scheduler queue according to the priority order of its head Scheduled Tick, so that it can continue processing its remaining tasks later.

4.6.4 Step 3: Delay Unadded Chunk-level Schedulers `delayAllTicks`

Add all Chunk-level Schedulers in the scheduler queue that have not yet been polled to a `chunk coordinates -> next execution time` mapping table (`nextTriggerTickByChunkPos`) for subsequent execution.

4.7 Scheduled Tick Suppressors

From the logic of executing Scheduled Ticks, we can notice: the sorting of Chunk-level Schedulers does not consider `triggerTick`. That is to say, if the head Scheduled Tick of a certain Chunk-level Scheduler or several Chunk-level Schedulers has sufficiently high priority, and the number of Scheduled Ticks in that scheduler is sufficiently large, the World-level Scheduler can continuously poll Scheduled Ticks only from these few schedulers until the Scheduled Tick limit is reached, without polling Scheduled Ticks from other schedulers. And this suppressor can only take effect on chunks other than the one where the suppressor is located.

For example, if repeaters are activated at a certain high frequency in certain chunks, components with priority 0 in other chunks will be "suppressed" during the high-frequency period, such as comparators not lighting up or Observer high-frequency not working. These Scheduled Ticks will only execute again once the high-frequency period ends.

Footnotes

1. "How long to delay before execution" is not strictly accurate. In reality, the execution time records the world tick, "delay `x` gt" means `execution time world tick = current world tick + x`. When `current world tick >= execution time world tick`, the Scheduled Tick executes. However, this distinction does not affect Scheduled Tick analysis in most cases, except for chunk unloading situations. ↩
2. However, there are currently no Scheduled Ticks with priority 1, 2, or 3. They are only declared in the code. ↩
3. The basic section only provides a simple description here. For specific content involving Scheduled Tick queue maintenance, see the advanced section. ↩
4. Because in the source code, they inherit from the same abstract base class `AbstractRedstoneGateBlock` ↩
5. To be precise, it should be "a signal that turns off before the repeater executes its Scheduled Tick (to light up)". ↩
6. Due to the Scheduled Tick execution logic, Scheduled Ticks that are about to execute in the current gt are not considered as "having a Scheduled Tick". That is, if an observer will light up in a certain gt, and a PP update is sent to the observer before it lights up, the observer will also add a Scheduled Tick. This logic is the core principle of 4gt Observer high-frequency. Specific behavior and examples are detailed in the advanced section. ↩
7. This turning off is controlled through `WorldEvent` (world event), rather than directly calling `setblockstate`. ↩

> 8. The translations used in this section (3.4) are all drafted by the editors themselves, and no corresponding translations have been found in other related literature so far. If readers have better translations or suggestions for modifications, please submit an issue to contact us directly. ↩

2.4 Block Entities

1 What Are Block Entities

Blocks store data using a limited set of predefined BlockStates, render models through default rendering behavior, and execute logic via update and scheduling systems.
However, this approach has limitations and cannot directly support certain block behaviors required for game interactions.

Block entities solve this problem. They give ordinary blocks three key capabilities:

- ***Store data using NBT***
- ***Custom rendering behavior***
- ***Update every tick***

1.1 Block Entity Contents

Block entities vary by block type, but a basic block entity typically contains:

- type
- world (the world it exists in)
- pos (position)
- removed (whether it has been removed)
- cachedState (cached state of the corresponding block)

1.2 Block Entities and Blocks

Block entities are bound to blocks in the world. Each block position can only have one block entity instance.

Only blocks that are specifically declared can have block entities attached.
Block entities are added and removed alongside their blocks. However, through techniques like **update suppression**, you can retain a block entity even after its block is removed.

2 Notable Blocks with Block Entities

2.1 Hopper

In Minecraft, the hopper is a block entity that handles automated item transfer.

2.1.1 Hopper Block Entity Data

A hopper block entity stores the following information:

- `inventory` storage space (5 slots)
- `transferCooldown` cooldown (default is -1)

- `lastTickTime` world time of the last tick
- `facing` direction

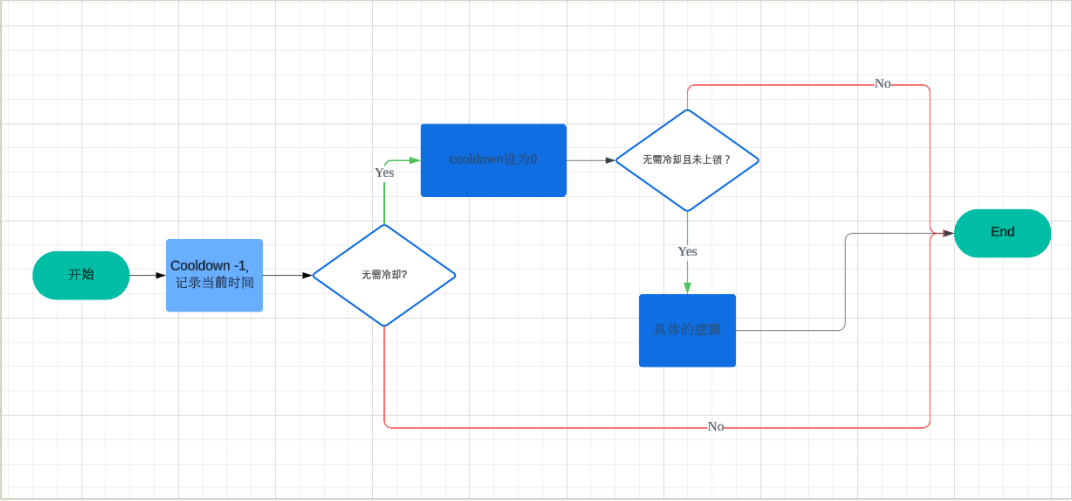
2.1.2 Hopper Block Entity Functions

- **Transfer items to target container**
- **Extract items from container**
- **Attempt to pick up item entities**

2.1.3 Hopper Block Entity Workflow

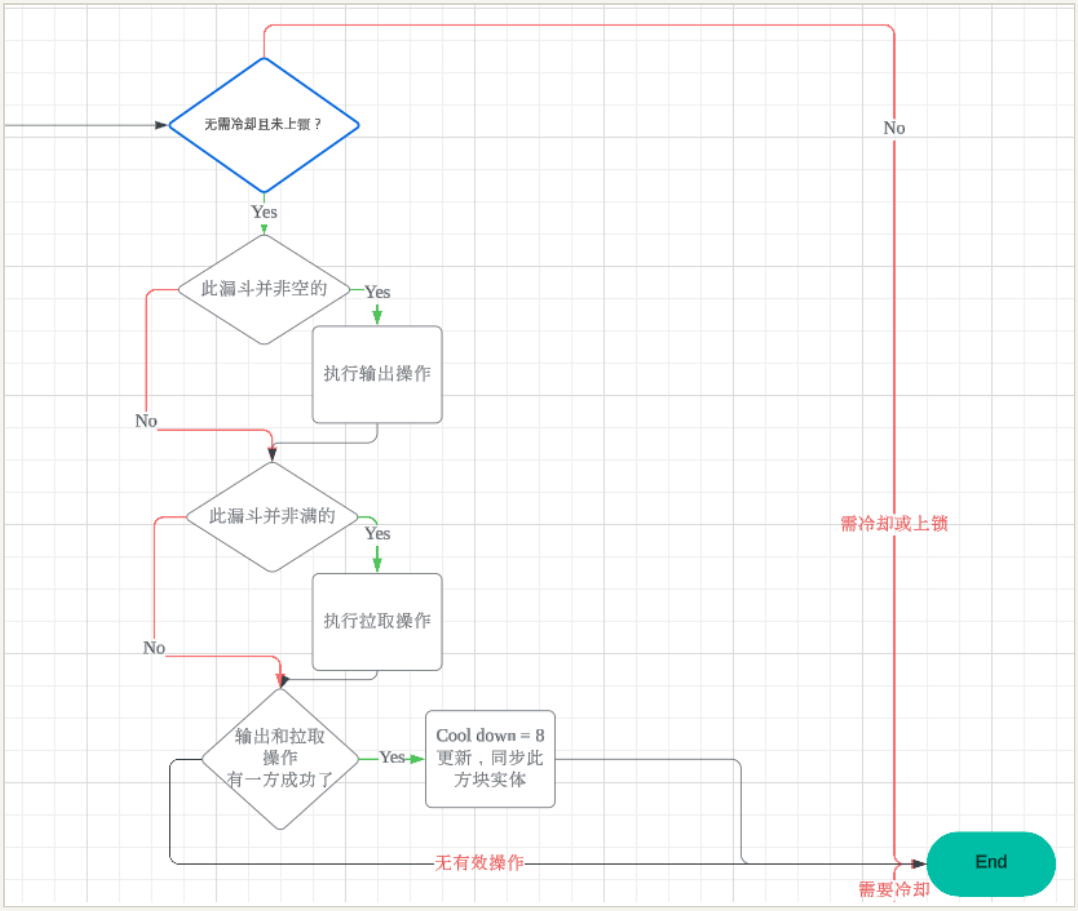
Every game tick, the hopper executes these steps:

1. **Decrease cooldown (cd)**
    - If cd is still greater than 0, the hopper skips all pull and output operations and stops processing.
    - If cd reaches zero, the hopper begins attempting **output** and **pull** operations.
2. **Execute item transfer logic**
    - **First attempt output** (place items into target container)
    - **Then attempt pull** (extract items from container above)
    - **If either succeeds**, the hopper resets the cooldown and syncs its data.



Now, let's look at the ***detailed logic for pulling and outputting***.

When the hopper is ready to operate, it first attempts to output to the container it's pointing at. Then it tries to extract items from above. Finally, if either operation succeeds, it sets the cooldown and syncs its data.

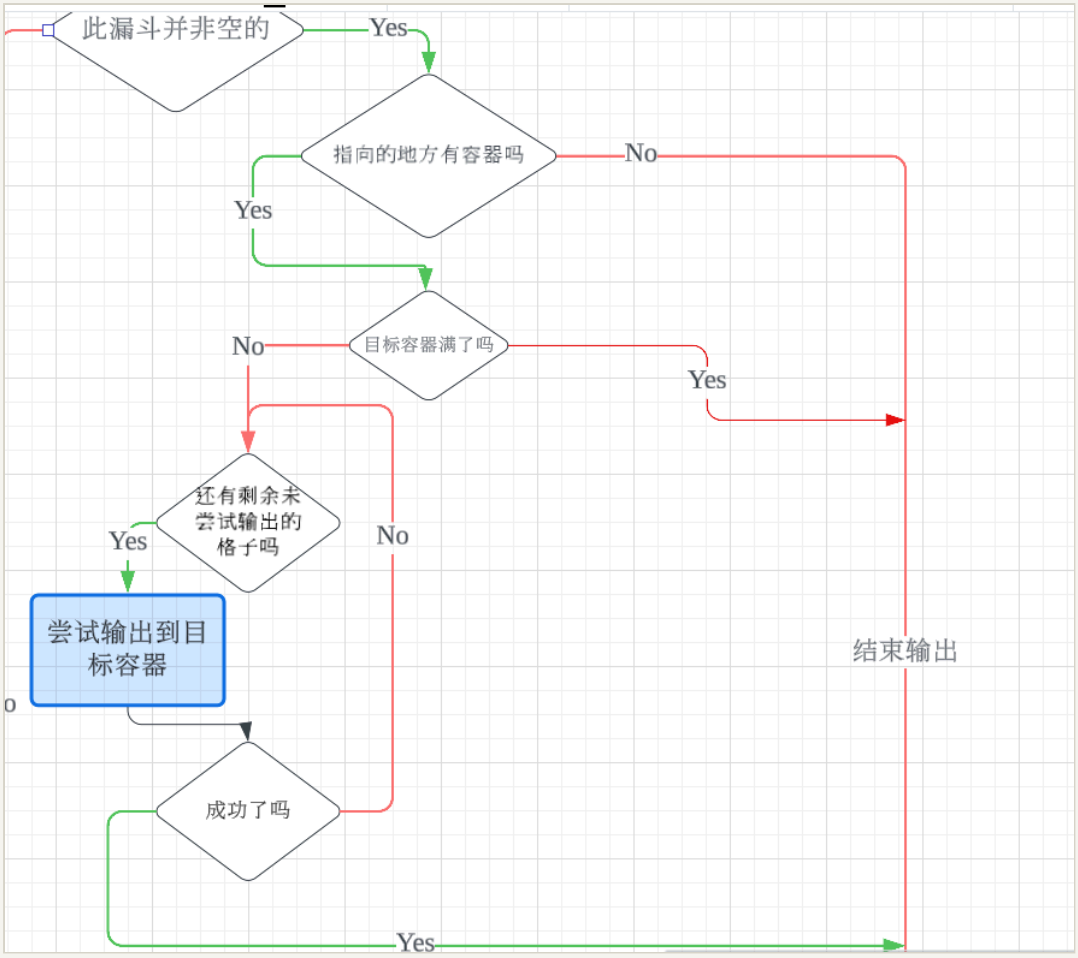


Let's look at output and pull separately.

First, ***output***...
The ***output*** operation uses the `insert()` method, which transfers items from the hopper to the target container and returns a boolean indicating success. If the transfer succeeds, the hopper enters cooldown.

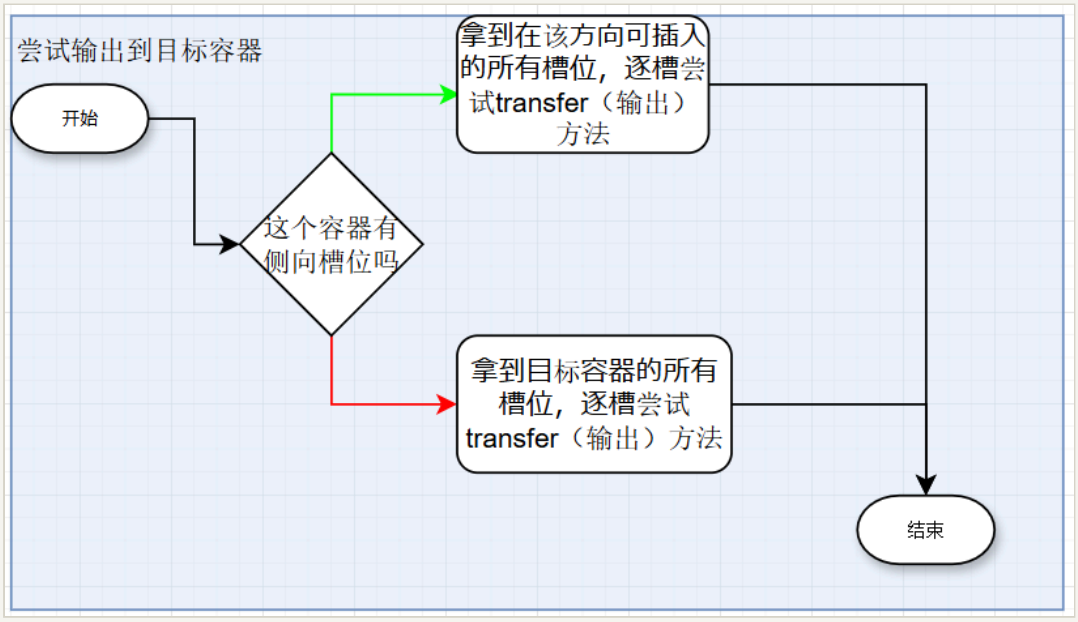
Basic output logic:

1. **Check if the target container is available**
    - If the target container is full, output fails.
    - If the target container has available slots, attempt to transfer items.
2. **Attempt to output items**
    - Iterate through the hopper's item slots, attempting to transfer items to the target container one by one.
    - If any transfer succeeds, return "success"; otherwise return "failure".



When calling `transfer()` to move items, it returns an item stack:

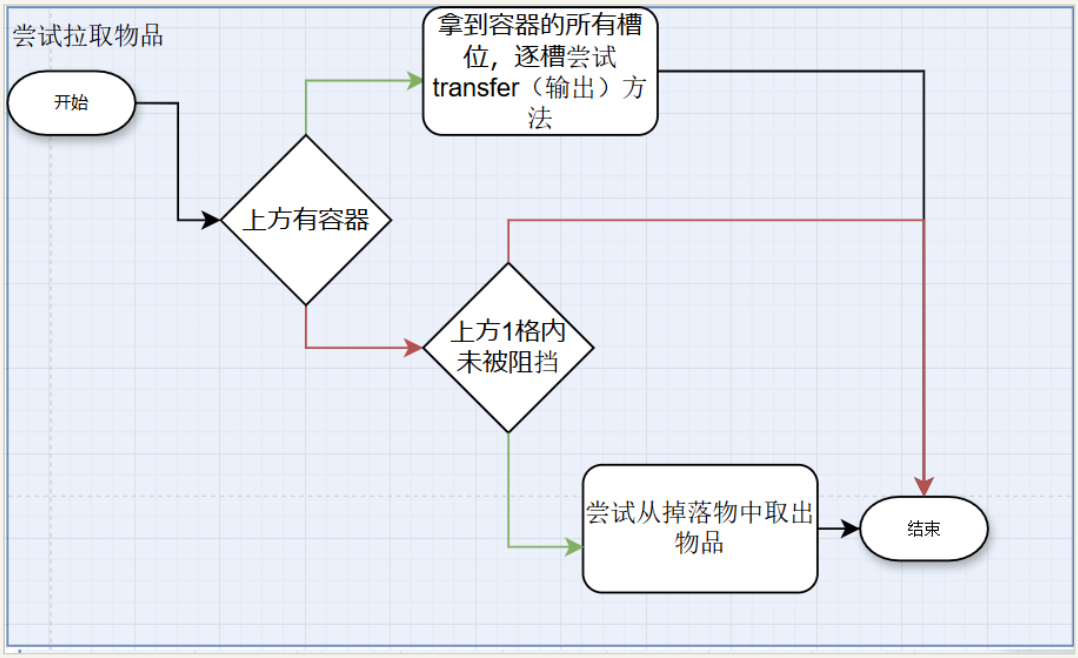
- An empty return value means the items were fully transferred, so output succeeds.
- A non-empty return value means the items could not be fully transferred, so output fails.



That covers output. Now let's look at ***pulling***.
The pull logic is similar to output, but in the opposite direction:

- During **output**: the hopper `transfer()`s items to the **target container**.
- During **pull**: a **container** `transfer()`s items to the **hopper**.

Since we'll cover `transfer()` in detail next, we won't repeat it here.



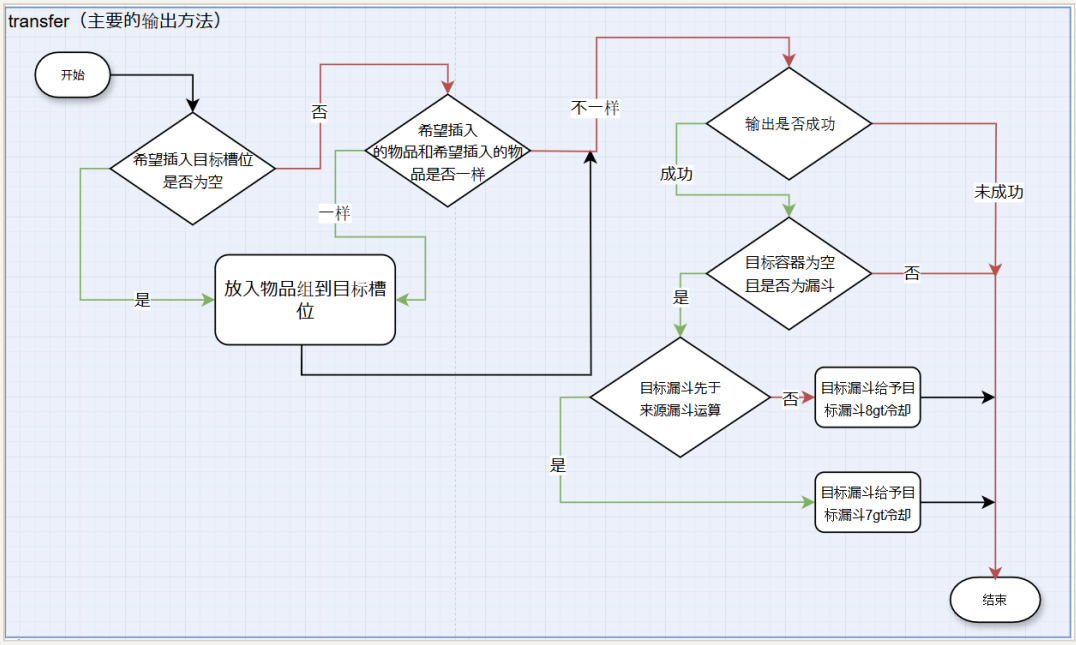
`transfer()` is the core method that handles item transfer in hoppers. It takes the following parameters:

- **from**: item source (can be a hopper or other container)
- **to**: transfer target (can be a container or another hopper)
- **itemStack**: the item stack to transfer
- **slot**: target slot in the target container

Here's how it works:

1. **Check the target slot**
   - If the slot is empty, place `itemStack` directly into it and clear `itemStack`.
   - If the slot already has items of the same type with remaining space, merge the items and clear `itemStack`.
2. **Determine if the transfer succeeded**
   - After transfer, if `itemStack` is empty, the transfer succeeded; otherwise it failed.
3. **Special timing for hopper-to-hopper transfers**
   - If the target is another hopper, the target hopper receives `8gt` of cooldown.
   - If the target hopper `to` has already ticked before the source hopper `from`, the target hopper `to` only receives `7gt` of cooldown.

> Note that this only occurs when the target hopper is empty.
> Additionally, even if `from` ticks after `to` and applies `8gt` of cooldown, `to` will immediately subtract 1 when it ticks next, leaving only `7gt` of cooldown.



That covers the basic hopper mechanism. This foundation will help with the chapters ahead.

2.1.4 Moving Piston

Another important block is `moving_piston`, commonly known as B36.

2.1.5 Moving Piston Block Entity Data

A moving piston block entity stores the following information:

- `pushedBlock` the block being pushed
- `facing` direction
- `extending` whether the piston is extending (determines animation direction)
- `source` whether this is the piston head retracting
- `progress` current movement progress
- `savedWorldTime` world time when logic was last executed

2.1.6 Moving Piston Block Entity Functions

- **Handle piston pushing entities**
- **Update animation progress**
- **If `progress>=1`, replace MOVING_PISTON with the block stored in `pushedBlock`**
- **Render animation on the client**

2.1.7 Moving Piston Block Entity Workflow

For details, see 5.7.

3 Timing Between Block Entities

Block entity ticking is managed by two lists in `LevelChunk`:

- `pendingBlockEntityTickers`: block entities added during the current tick cycle.
- `blockEntityTickers`: block entities scheduled for ticking.

Both lists are ArrayLists, so traversal order depends on insertion order. Here's the process:

- `tickBlockEntities()` first appends `pendingBlockEntityTickers` to `blockEntityTickers`, then clears `pendingBlockEntityTickers`.
- When iterating through `blockEntityTickers`, removed entries (`isRemoved()` returns `true`) are deleted; others have their `tick()` method called.

New block entities are added via `addBlockEntityTicker()`. If ticking is in progress (`tickingBlockEntities == true`), they go to `pendingBlockEntityTickers`; otherwise to `blockEntityTickers`.

When unloading, block entities in the same chunk are stored in a `Map<BlockPos, BlockEntity>`. On reload, the map is iterated and each entity is added to `blockEntityTickers` via `addBlockEntityTicker()`, so execution order follows `BlockPos.hashCode()` order.

Across chunks, block entity order depends on which chunk loads first.

TL;DR:

Before unloading, block entity tick order matches placement order. After reloading, order within a chunk depends on position hash; across chunks, it depends on chunk load order.

Here's the relevant code (with some irrelevant content removed):

```java
// Used to temporarily store block entities dynamically added during the tickBlockEntities process
private final List<TickingBlockEntity> pendingBlockEntityTickers = Lists.newArrayList();
// List of block entities to be ticked
protected final List<TickingBlockEntity> blockEntityTickers = Lists.newArrayList();
private void tickBlockEntities() {
   this.tickingBlockEntities = true;
   if (!this.pendingBlockEntityTickers.isEmpty()) {
      this.blockEntityTickers.addAll(this.pendingBlockEntityTickers);
      this.pendingBlockEntityTickers.clear();
   }
   // Traverse the list of block entities being ticked
   Iterator<TickingBlockEntity> iterator = this.blockEntityTickers.iterator();
   while (iterator.hasNext()) {
      TickingBlockEntity tickingBlockEntity = iterator.next();
      if (tickingBlockEntity.isRemoved()) {
         // If the block entity has been removed, remove it from the list
         iterator.remove();
      } else if (this.shouldTickBlocksAt(tickingBlockEntity.getPos())) {
         // If the position should be ticked, call the tick() method for updating
         tickingBlockEntity.tick();
```

```
      }
   }
   this.tickingBlockEntities = false;
}
// Add a block entity's ticker
public void addBlockEntityTicker(TickingBlockEntity tickingBlockEntity) {
   (this.tickingBlockEntities ? this.pendingBlockEntityTickers : this.blockEntityTickers)
           .add(tickingBlockEntity);
}
// All block entities within the chunk
protected final Map<BlockPos, BlockEntity> blockEntities = Maps.newHashMap();
// After chunk loading is complete, register all existing block entities (add them to the tick system)
public void registerAllBlockEntitiesAfterLevelLoad() {
   this.blockEntities.values().forEach(blockEntity -> {
      // Add a ticker for this block entity, i.e., add it to blockEntityTickers
      this.updateBlockEntityTicker(blockEntity);
   });
}
```

2.5 Block Events

This article covers how Block Events (BE) work under the hood.

Basics

- What Block Events are and why they exist
- Block Event structure and related components
- A hands-on understanding of Block Event Delay (BED)

Advanced

- Block Event queuing and the breadth-first search mechanism

1 What are Block Events?

In Minecraft, to minimize network traffic between server and client, the game avoids syncing all the detailed data from every complex block every tick.

Instead, the server tells the client "a specific event happened here" and lets the client simulate those events locally.

Block Events were designed for this efficient "notify and simulate" pattern.

1.1 Block Event Structure

When a Block Event is created, it contains exactly four pieces of information:

- Position `pos`: where the event occurs
- Block type `block`: which type of block is executing this event
- Type `type`: which specific event action (for pistons, represents extend or retract)
- Data `data`: necessary parameters attached when executing the event (for pistons, represents the direction of push)

2 Block Event Sources

> Some redstone components produce Block Events for various purposes, including but not limited to:

COMPONENT TYPE	EVENT TYPE	ATTACHED DATA
Piston	(Extend `0`, Retract `1`, Instant retract `2`)	Piston direction (0~5)
Note Block	`0`	`0`
Bell	`1`	Hit direction (0~5)

...

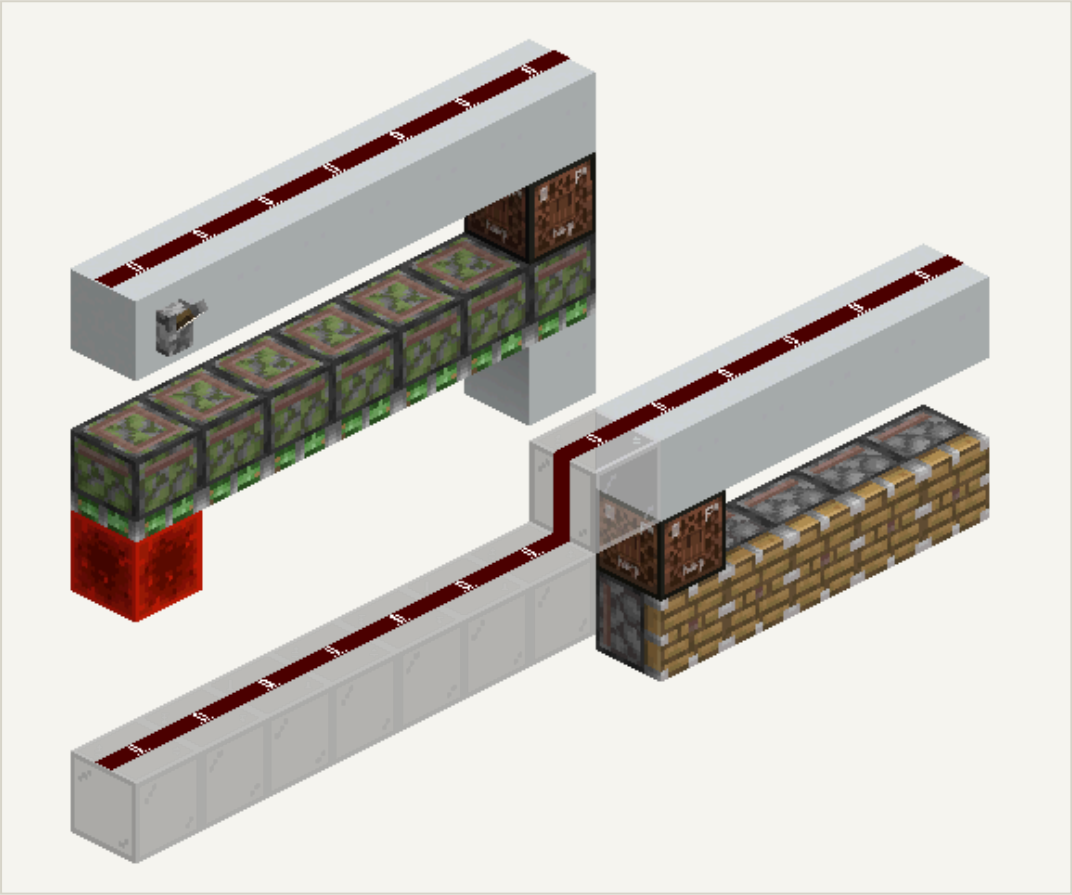
---

3 Block Event Delay (BED)

When multiple blocks are activated at the same time and generate Block Events, do they execute simultaneously?

By Minecraft's design, nothing in the game is truly "simultaneous." Even within the same game tick, events must execute in a fixed order. This ordering effect is called **Block Event Delay (BED)**.

Here's what Block Event Delay looks like in practice:



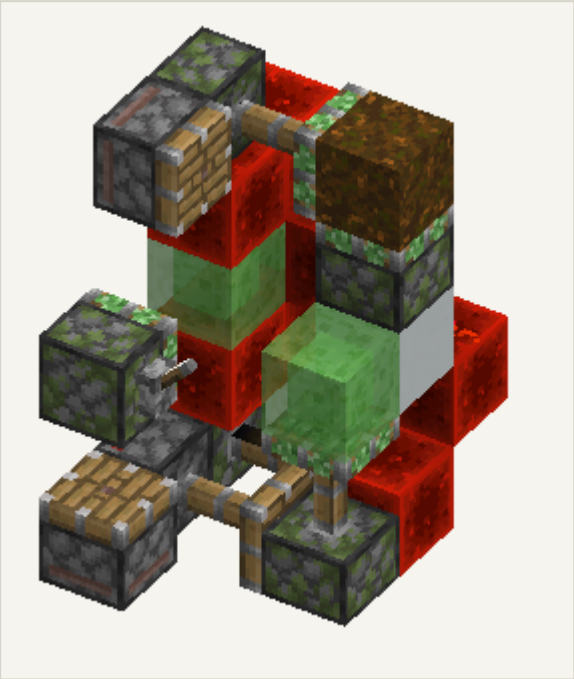
Flip the lever and you'll notice that moving the redstone block changes how many pistons extend nearby. When the redstone block sits closer to the pressure plate block, fewer pistons extend; when it moves farther away, more extend.

3.1 Depth and Queuing: A Mental Model

Think of Block Event execution like a ticket queue: **first come, first served**.

When a Block Event executes, it may trigger new events. These get appended to the end of the queue. To track which "generation" produced each event, we use **depth**: the initial event has depth 0, events it directly produces have depth 1, and so on, capturing the chain of cause and effect.

Below is an example of a 0t bottom retraction base in a tree farm, showing how Block Event depth works in practice:



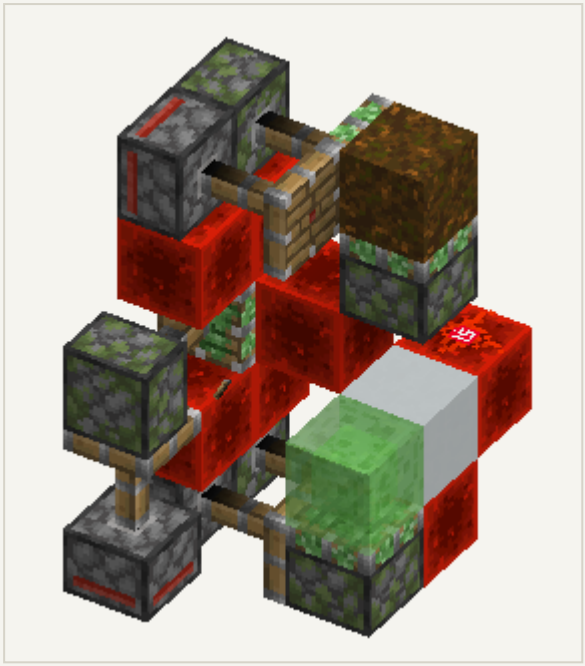
0gt AT: Lever is pulled down

1gt BE depth 0: Sticky piston starts retracting

1gt BE depth 1: Sticky piston pulls back podzol, redstone dust changes direction

1gt BE depth 2: Bottom retraction piston receives an update, self-checks, sees it's activated, extends

1gt BE depth 3: Powered block is removed, updating the bottom retraction piston. It self-checks, sees it should retract, queues a retract Block Event, but notices it's still extending, triggering a 0t.



3gt TE: All blocks are in place.

4gt BE depth 0: Top and bottom regular pistons extend, dirt's sticky piston adds Block Event

4gt BE depth 1: Regular piston extends, updating the sticky piston. The pushback sticky piston extends, base activates piston self-check. Redstone block is removed.

4gt BE depth 2: Top and bottom regular pistons retract, sending updates. Dirt's sticky piston receives an update, self-checks, sees it shouldn't extend, queues a retract Block Event. Bottom sticky piston follows the same logic.

4gt BE depth 3: Redstone block sticky piston and podzol sticky piston trigger 0t, podzol snaps into place. Redstone block lands in position, sticky piston controlling the bottom retraction piston's powered block extends.

6gt TE: Redstone dust changes direction, powered block in place, base finishes resetting.

---

4 Block Event Queue and Breadth-First Search

Under the hood, Block Event "queuing" is essentially a breadth-first search mechanism.

In the game's main loop, Block Events sit in a first-in-first-out queue. When a source (like a central piston triggered by a lever) propagates updates to neighbors in NC update order, each neighboring piston pushes its Block Event to the end of the queue.

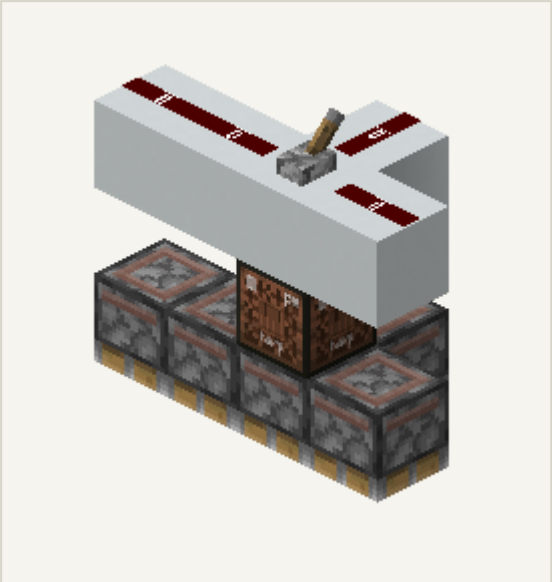
During execution, the game pulls an event from the head of the queue:

1. Dequeue and execute the first node with depth `0`
2. If this update triggers new updates, the resulting Block Events (depth `1`) get pushed to the tail of the queue in update order.

3. The game keeps executing depth `0` nodes until they're all gone, then moves on to depth `1` events.

This is exactly equivalent to a breadth-first search.

Here's an example:



The long edge extends from +x toward -x.

Pulling the lever activates the central piston, which updates the three surrounding pistons in -x, +x, +z order, pushing them into the queue one by one. The game then executes the -x and +x pistons in sequence, followed by the next piston in the +x direction. That piston adds a Block Event to the queue, placing it after the +z piston. Next the +z piston executes, and finally the last +x piston runs.

We can abstract each piston as a node, treating depth-0 pistons as root nodes. Note that in practice, this graph may have multiple sources. We run multi-source BFS on this abstracted "piston graph," where child nodes join the queue in NC update order, determined by their direction relative to the parent. Children that have already been visited are not added again.

CHAPTER 3

3

Redstone Components & Features

IN THIS CHAPTER

3.1 Rails

3.2 Pistons

3.1 Rails

1 Rail Updates

Special rails (Powered Rails, Activator Rails, Detector Rails) all have special update ranges, unlike plain rails.

1.1 Powered Rail/Activator Rail Updates

Powered Rails and Activator Rails have identical update behavior.[1]

When a Powered Rail/Activator Rail's activation state changes, it emits NC updates in sequence from these positions:

1. One block above (if the rail is sloped)
2. Itself
3. One block below
4. Itself
5. One block below
6. One block above (if the rail is sloped)

In other words, a flat Powered Rail/Activator Rail emits 4 NC updates with sources: **itself->below->itself->below**.

A sloped Powered Rail/Activator Rail emits 6 NC updates with sources: **above->itself->below->itself->below->above**.

1.2 Detector Rail Updates

Not yet completed

2 Rail Activation Detection

Throughout this section, "rails" refers to Powered Rails and Activator Rails.

2.1 Activation Methods

There are two conditions that can activate a rail:

- **Direct activation:** the rail is directly adjacent to a redstone signal source.
- **Indirect activation:** among the 8 rails **connected** to it (excluding itself), at least one is directly adjacent to a redstone signal source.

For example, if a rail is directly adjacent to a Block of Redstone, it is **directly activated**.

If a rail is connected to several other rails forming a chain, and a Block of Redstone is adjacent to any rail within 8 blocks along that chain, then the rail is **indirectly activated**.

2.2 "Connected" Judgment Conditions

A rail that is not directly activated uses a "search" mechanism to determine whether it is indirectly activated. Specifically, it searches for a **connected**, **directly activated** rail within 8 blocks of itself.

To put it simply: imagine you're going to a restaurant. You open a map, search for the restaurant, and see if you can walk there. The restaurant won't fly to you and say "I'm 800m away, come walk over."

Rails work the same way. Each rail "searches" for qualifying signal sources on its own. It tries to "walk" along the rail chain for up to 8 blocks to see if it can find a signal source, that is, a rail directly connected to a signal source.

Intuitively, if the rail "walks" halfway and the rail chain is "broken," it can't continue walking.

When a rail searches for connected rails at a certain position, it searches for rails at **specific positions** with the **same direction** as itself, based on its own **shape** (i.e., whether it is sloped). Whether a rail considers the next rail connected depends only on these three factors:

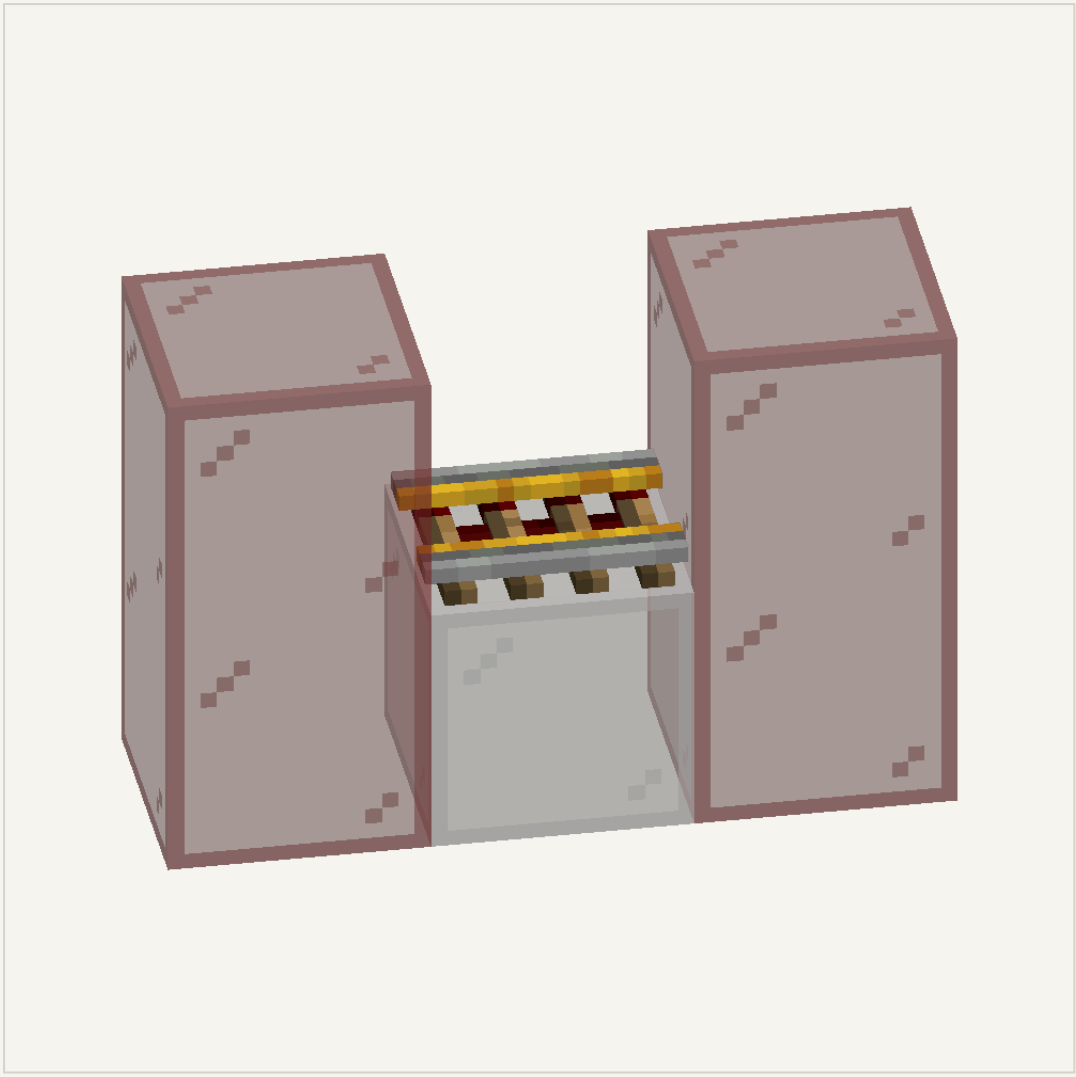
- Its own shape (whether it is sloped)
- The position of the next rail
- The direction of the next rail

It is independent of **the shape of the next rail**.

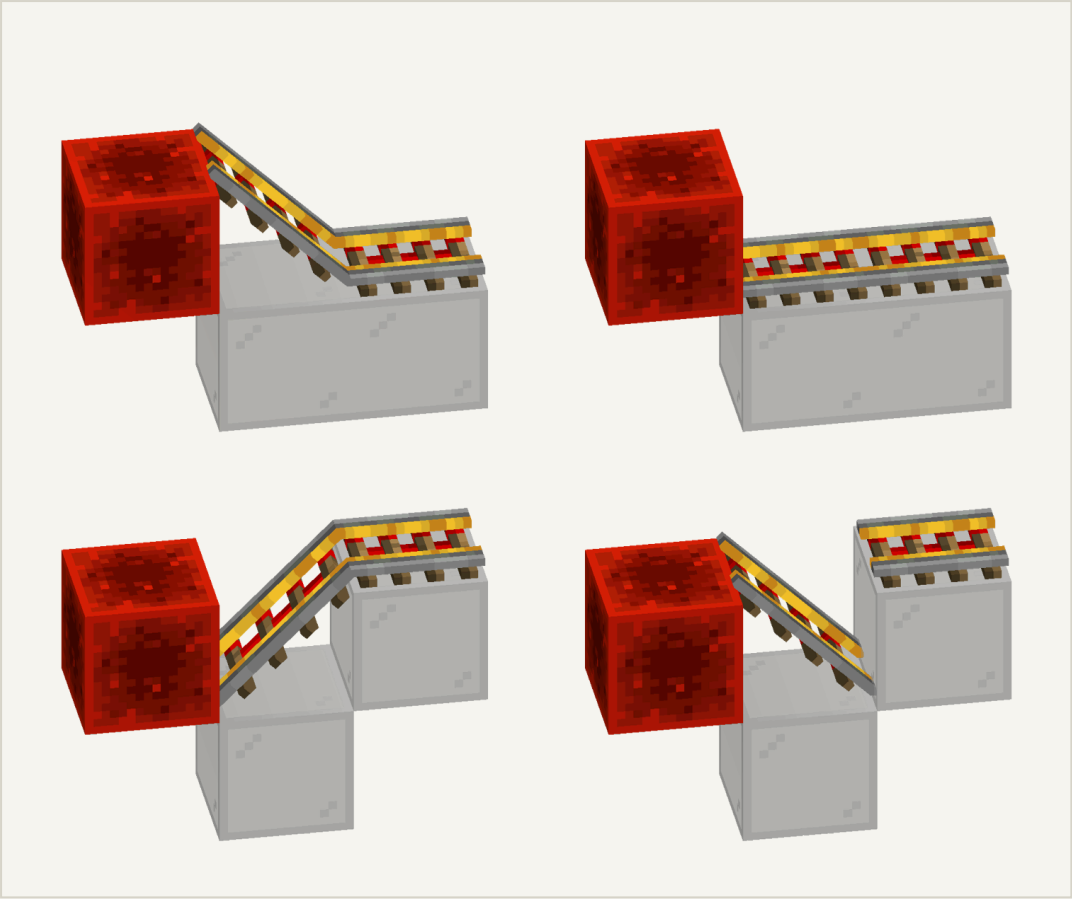
Flat rails will check these positions:

- Positions at both ends of the rail, at the same height as itself
- Positions at both ends of the rail, one block lower than itself

The range is shown by the red glass positions in the image:



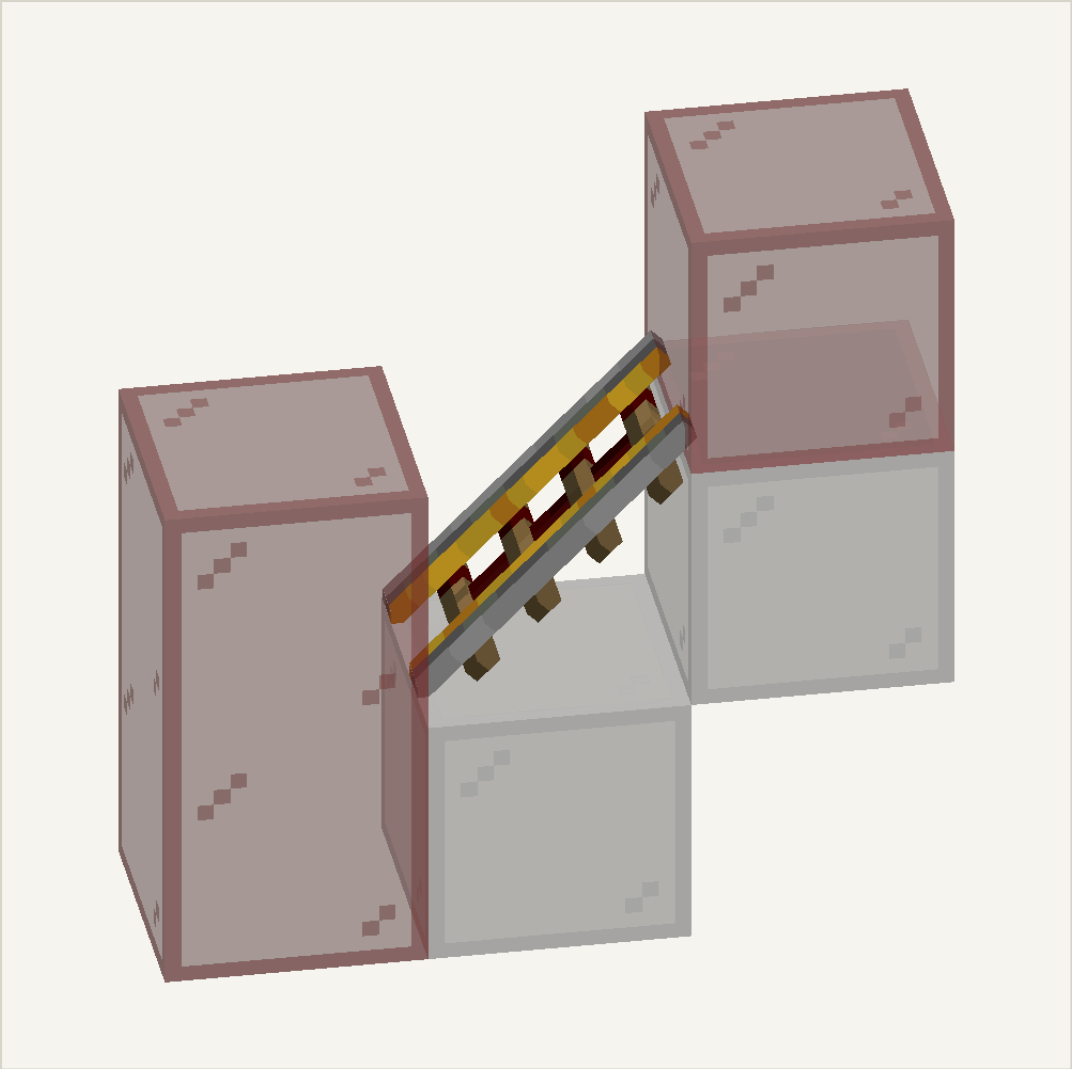
For example, when the flat rail on the far right is "searching" to the left for directly activated rails, it considers the rail on the left connected to itself. The first three cases in the image are straightforward, while the last one occurs because **rails do not consider the shape of the next rail when checking connectivity**:



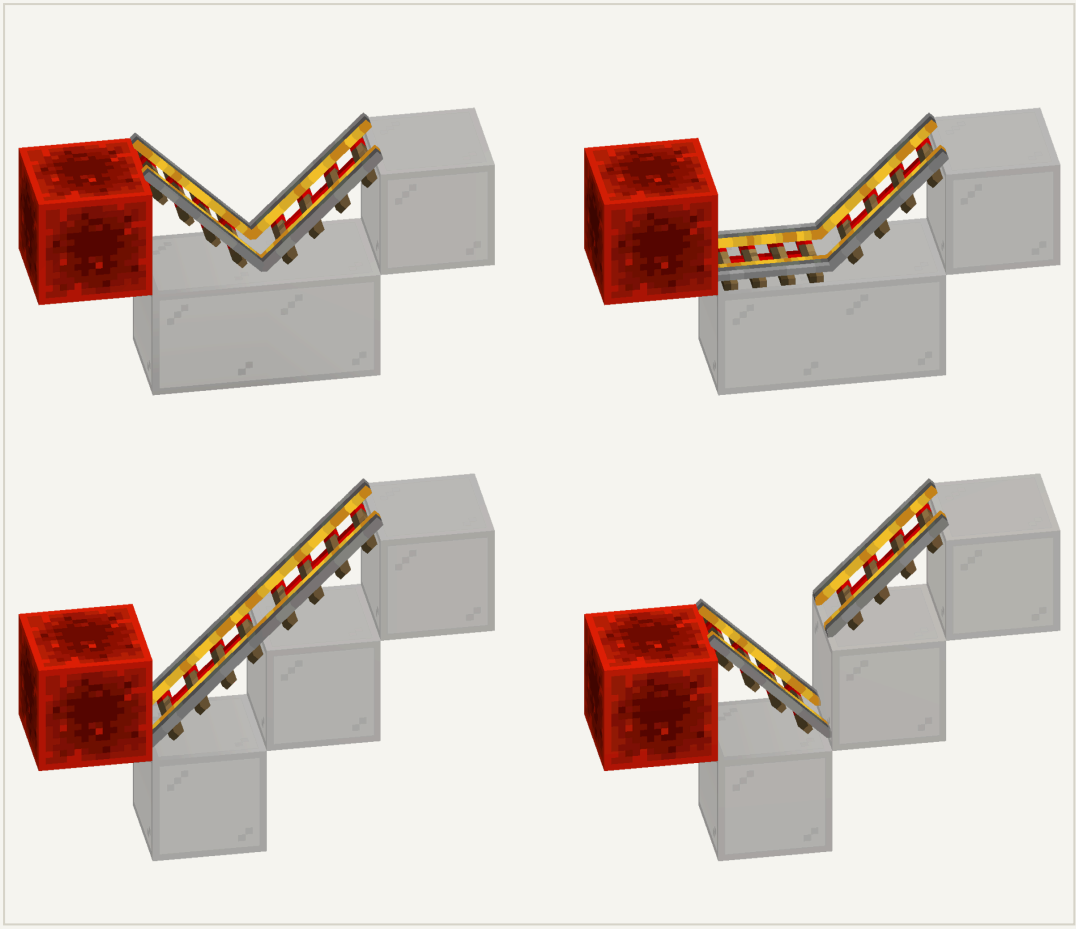
Sloped rails will check these positions:

- Position at the lower end of the sloped rail, at the same height as itself
- Position at the lower end of the sloped rail, one block lower than itself
- Position at the higher end of the sloped rail, one block higher than itself

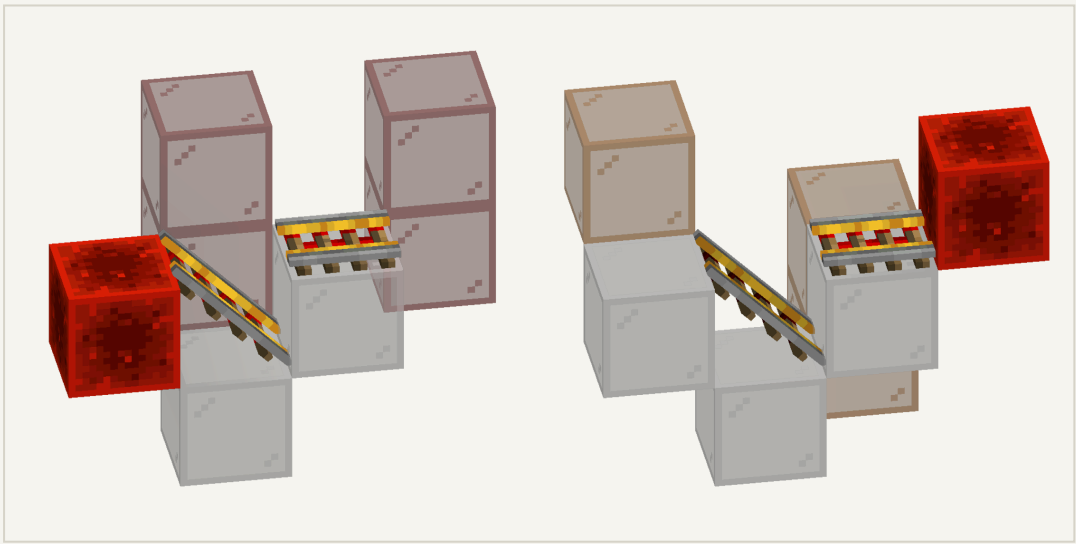
The range is shown by the red glass positions in the image:



For example, when the sloped rail on the far right is "searching" to the left for directly activated rails, it considers the rail on the left connected to itself. The first three cases in the image are straightforward, while the last one occurs because **rails do not consider the shape of the next rail when checking connectivity**.



Combining the above two points, we can see that rail "connectivity" is unidirectional: when rail A considers rail B connected to itself, rail B does not necessarily consider rail A connected to itself:



In the rail chain shown in the image, the redstone signal can propagate from right to left, but not from left to right. From the "search" perspective, this means: the sloped rail on the left considers the flat rail on the right not connected to itself, while the flat rail on the right considers the sloped rail on the left connected to itself.

Looking at the positions where these two rails check for connectivity: in the image above, the horizontal positions corresponding to the orange glass are where the sloped rail checks for connectivity, and the horizontal positions corresponding to the red glass are where the flat rail checks. The sloped rail is at the checking position of the flat rail, while the flat rail is not at the checking position of the sloped rail, which makes **rail connectivity unidirectional**.

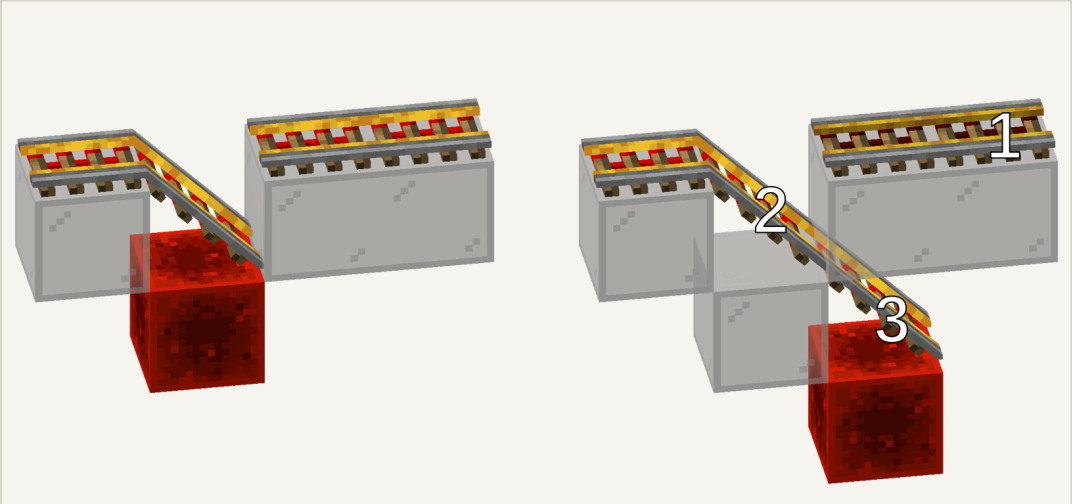
This can be used to create rail chains that can only transmit rail signals in one direction.

2.3 Direction of Search

Rails follow a fixed order when searching for connected rails: each end is searched independently, and each search is unidirectional (no backtracking).

- **Search process:** First, search outward from one end along that direction. During the search, the direction does not change (i.e., only "forward," no "backward"). After the search at that end is complete, repeat the same process for the other end.
- **Direction rules:** North-south oriented rails always search south first, then north; east-west oriented rails search west first, then east.

Therefore, whether multiple rails are considered "connected" depends not only on distance, but also on the search direction order. This explains the difference shown in the diagram: in the first case, the rails on both sides of the Block of Redstone can be found and activated. In the second case, only the rail on the left side of the Block of Redstone can be activated, because the search direction on the right side cannot pass through a certain rail in the middle to reach the target position, and thus cannot find the directly activated rail.



As shown in the image, in the first case, the rails on both the left and right sides of the Block of Redstone can be activated.

In the second case, only the rail on the left side of the Block of Redstone can be activated. When the rail on the left side searches **to the right** for directly activated rails, it first finds rail #2, then searches **to the right** (down) to find rail #3.

When the rail on the right side of the Block of Redstone searches **to the left** for directly activated rails, it also first finds rail #2. But since the search direction at this time is **to the left**, and rail #3 is on the **right side** of rail #2, it cannot search for rail #3, and thus cannot find the directly activated rail or be activated.

Footnotes

1. Powered Rails and Activator Rails are both implemented by the `PoweredRailBlock` class, so their behavior is completely identical at the source code level. The different effects they produce on minecarts are determined and handled by the minecarts themselves, not by the rails. ↩

3.2 Pistons

1 Mechanics

Basic Section

- Piston composition
- Creating headless pistons
- Piston self-check trigger methods and redstone signal detection (QC quasi-connectivity)
- Structural analysis and common logic for pistons attempting to move blocks
- Piston update order and block event response

Advanced Section

- Source code calls for piston self-check and signal detection
- B36 conversion order and push/pull structure analysis
- Low-level implementation of block movement and hash table usage
- B36 placement and entity calculations

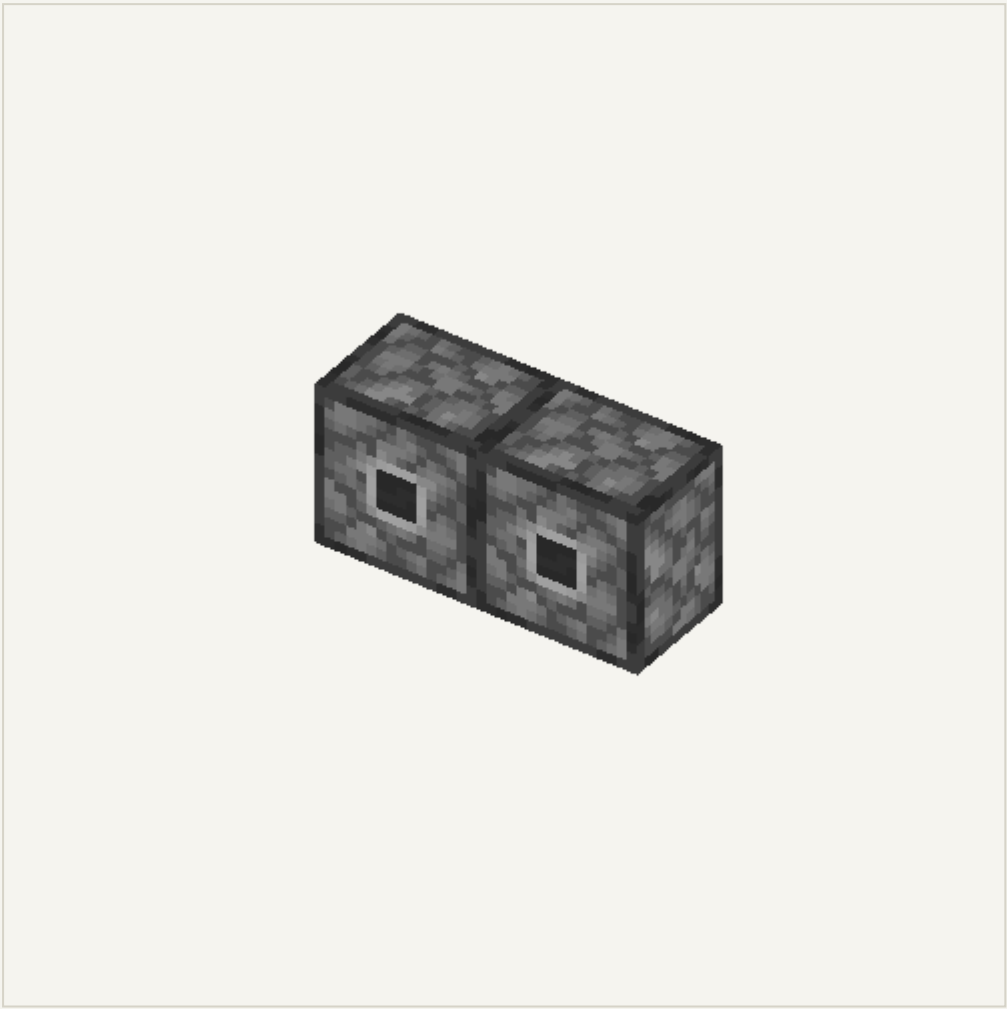
1.1 Piston Composition

A complete piston consists of two parts: the **piston base** and the **piston head**.

The piston base fully controls the piston head, and pistons have two states: `retracted` and `extended`. Normally, when a piston is activated, the piston base enters the extended state and the piston head appears. When the piston retracts, the piston head disappears. It is also possible to create a **headless piston** with only the piston base and no piston head, or conversely a piston with only a piston head and no piston base.

1.2 Headless Pistons

1.2.1 Creating Headless Pistons



There are essentially only two ways to create headless pistons:

1. When the piston is extended and the piston head is in the b36 state, use TNT or other methods to remove the b36
2. Through specific means, trigger a retraction event on an unextended piston. This will be explained in more detail later

1.2.2 Basic Mechanics of Headless Pistons

From the piston base's perspective, the piston head still exists directly in front of it. This belief persists regardless of whether a piston head actually exists there. If an unpowered headless piston receives an update, it will self-check and retract.

1.3 Piston Self-Check Mechanism

1.3.1 Trigger Methods

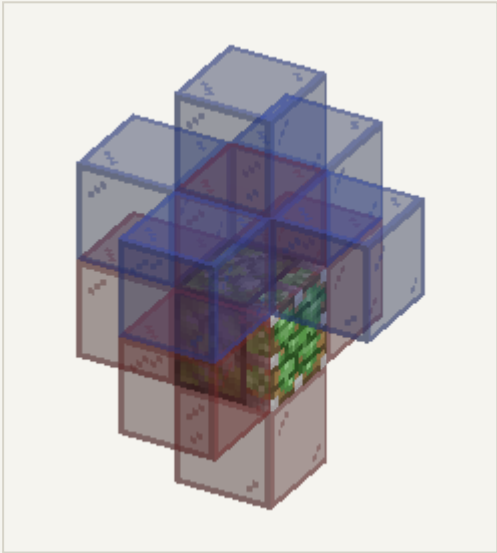
Pistons trigger self-checks if and only if the following two situations occur:

1. Being placed (including being placed by a player or being pushed/pulled into position by another piston)
2. Receiving an NC update

It must be emphasized that piston self-checks can occur at any stage. In other words, the self-check is instantaneous. When the above situations occur, the piston will immediately self-check and attempt to extend or retract through certain logic. This "attempt" does not guarantee success.

1.3.2 Redstone Signal Check and QC Powering

The essence of self-checking is to determine whether the piston needs to act, so it first checks whether it is powered.



The piston's signal check range includes:

- Whether all first-order adjacent blocks (except in the push direction) are outputting redstone signals
- Whether the first-order adjacent blocks **centered on the block directly above** are outputting redstone signals

Because pistons check the adjacent powering status of the block above them, they can be powered "remotely" through the space above. This powering cannot directly send an NC update to the piston, so an additional NC update is needed. This is the origin of Quasi-Connectivity (QC), and also why BUDs can only detect NC updates.

1.3.3 Self-Check Logic Explanation

When a piston self-checks, it always follows this behavior tree:

1. Should extend but not extended: Create an analysis instance to attempt to analyze the push structure in front. If analysis fails (exceeds push limit, blocked by immovable blocks, etc.), no action is taken; if analysis succeeds, add an **extend** block event to its current position
2. Should not extend but already extended: Piston needs to retract
    - If there is a "moving piston" (Moving_Piston, also called B36 or block 36, hereafter b36) in front moving in the same direction that is extending and won't finish this tick (progress <50%), trigger **instant retraction** and add the corresponding block event
    - Otherwise, execute **normal retraction** and add the corresponding block event.

1.4 Piston Push/Pull and Movable Blocks

1.4.1 Movable Block Determination

For a block to be pushable by a piston, it must simultaneously satisfy the following requirements:

1. Within the world's valid height range
2. Cannot be obsidian, crying obsidian, respawn anchor, reinforced deepslate, or other hardcoded immovable blocks
3. Block hardness cannot be -1
4. If it's glazed terracotta, the push direction must match the piston's push direction
5. The block cannot have a block entity (such as chests, furnaces, hoppers)

1.4.2 Attempting to Move Blocks and Structure Analysis

When a piston attempts to push, it dynamically maintains two lists: the **moved blocks list** and the **broken blocks list**.

It will determine which blocks to push and which to break in a certain order. The piston only considers breakable blocks along the push axis; blocks attached to the side of the push structure will be updated and drop later rather than entering the broken list.

> Tip: Due to the extreme complexity of structure analysis (alternating linear and branch analysis on the stack), manual analysis is impractical. It is strongly recommended to use Fallen_Breath's PistOrder mod to directly view the piston's attempted block placement order in-game.

1.5 Piston Update Order

The piston update order can be divided into five situations:

- Piston/sticky piston only extends
    1. Same as piston/sticky piston pushing blocks, just with empty moving blocks
- Piston/sticky piston only retracts
    1. Create piston base b36, send PP update and NC update
    2. Remove piston head, send PP update and NC update

- Piston/sticky piston pushes blocks
    1. Remove blocks at broken block positions
    2. Create b36 at moving target positions, send PP update
    3. Create piston head b36, send redstone dust prepare update and PP update
    4. Send redstone dust prepare update and NC update at broken block positions
    5. Send NC update at moving original positions
    6. Send NC update at piston head position
    7. Send PP update and NC update at piston base.
- Sticky piston retracts blocks
    1. Create piston base b36, send PP update and NC update
    2. Remove blocks at broken block positions
    3. Create b36 at moving target positions, send PP update
    4. Remove uncovered blocks, send redstone dust prepare update, PP update
    5. Send NC update at moving original positions
- Sticky piston retraction fails
    1. Create piston base b36 block, send PP update and NC update

Note that this behavior does not include any updates at the piston head, which is commonly referred to as sticky piston retraction failure with no updates. Here are a few examples:





▶ This figure is animated. View original

1.6 B36 and Block Placement

1.6.1 Normal Placement

In the block tick phase of each game tick, b36 adds 0.5 to its push progress

- 0gt BE: Start pushing, create b36, progress is 0 at this time
- 0gt TE: Progress +0.5, perform entity displacement calculation
- 1gt TE: Progress +0.5, perform entity displacement calculation, total progress reaches 1.0
- 2gt TE: Discovers progress is already greater than or equal to 1.0, b36 becomes the original block, sends final PP and NC updates.

If we start observing from 0gt AT, the block placement is fully complete at 3gt AT, hence the term "3-gt piston delay"

1.6.2 Instant Placement

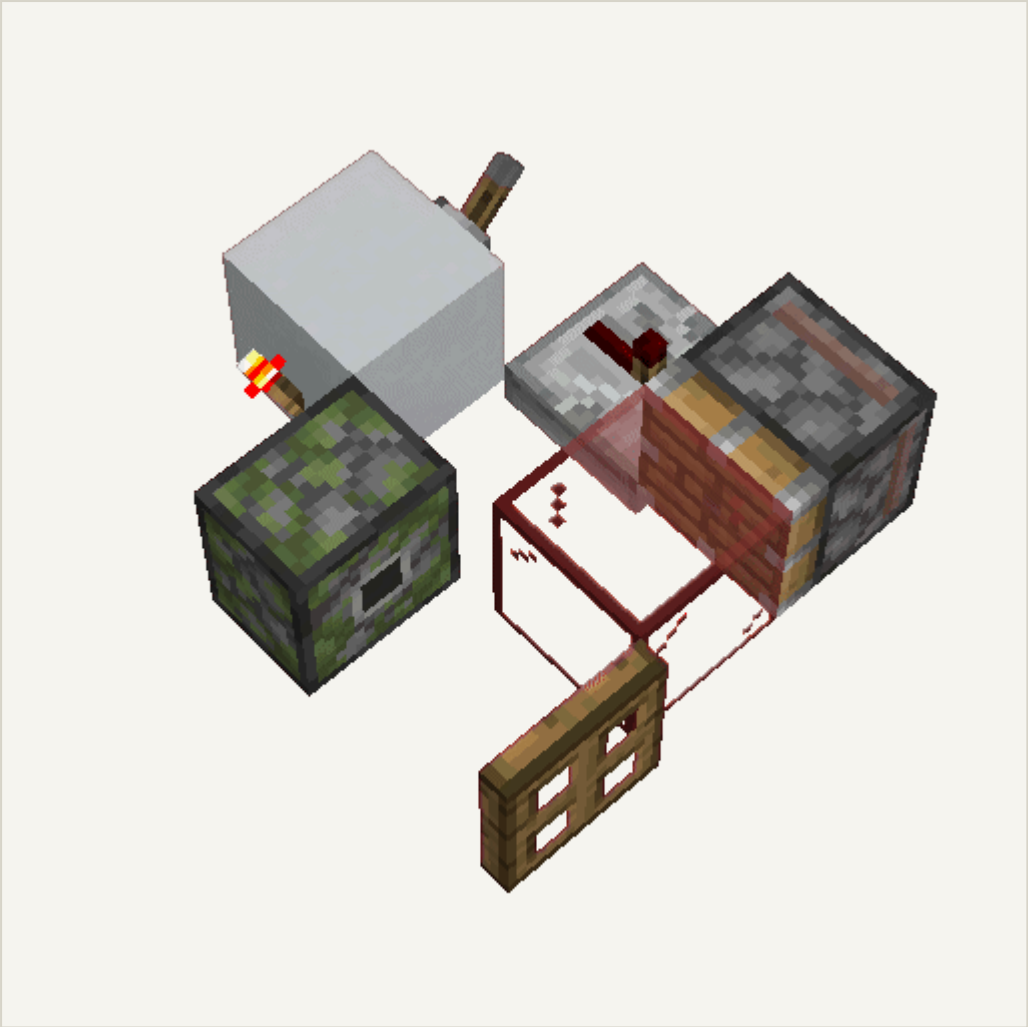
When a piston triggers instant retraction, blocks skip the normal TE phase accumulation and are forced to place immediately. Trigger conditions are:

- Retraction event, and the piston head position's current block is b36.
- Retraction event, sticky piston has b36 one block outside the extension direction and that b36 is moving in the same direction as the piston's push axis.

Here are two examples for demonstration:

Piston head position is b36

Let's look at the following example:



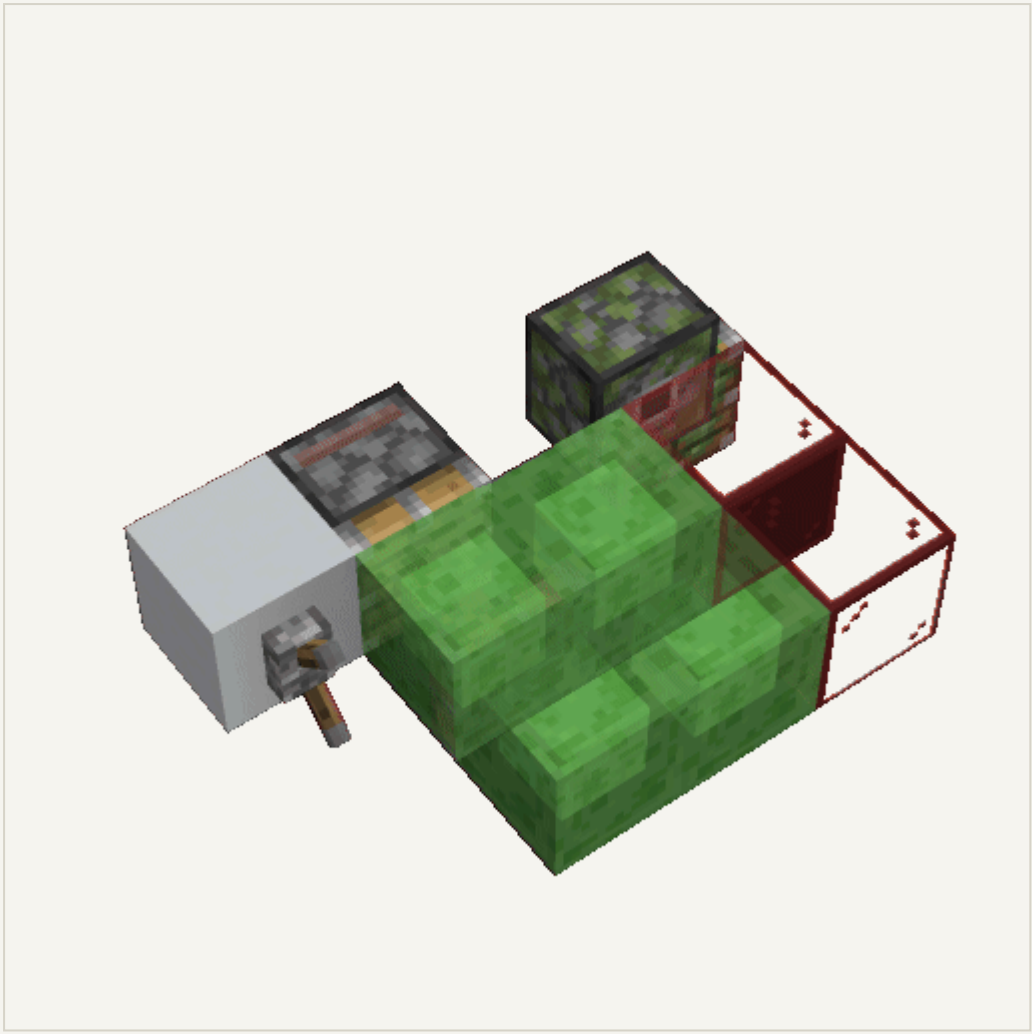
▶ This figure is animated. View original

In this example, the red glass is instantly placed from an observational standpoint. Theoretically, we provide the following explanation:

The repeater has higher priority than the redstone torch, so the normal piston adds a block event first, then the headless sticky piston adds a block event, attempting to pull back the block 1 block further forward (at this point actionType=1)

However, in the BE phase, due to the insertion of b36, the sticky piston's pull back is blocked and will not destroy this b36. At this point, the piston head position is b36, triggering instant retraction, and the red glass is instantly placed.

Sticky piston has b36 one block outside extension direction and b36 movement direction matches piston movement direction

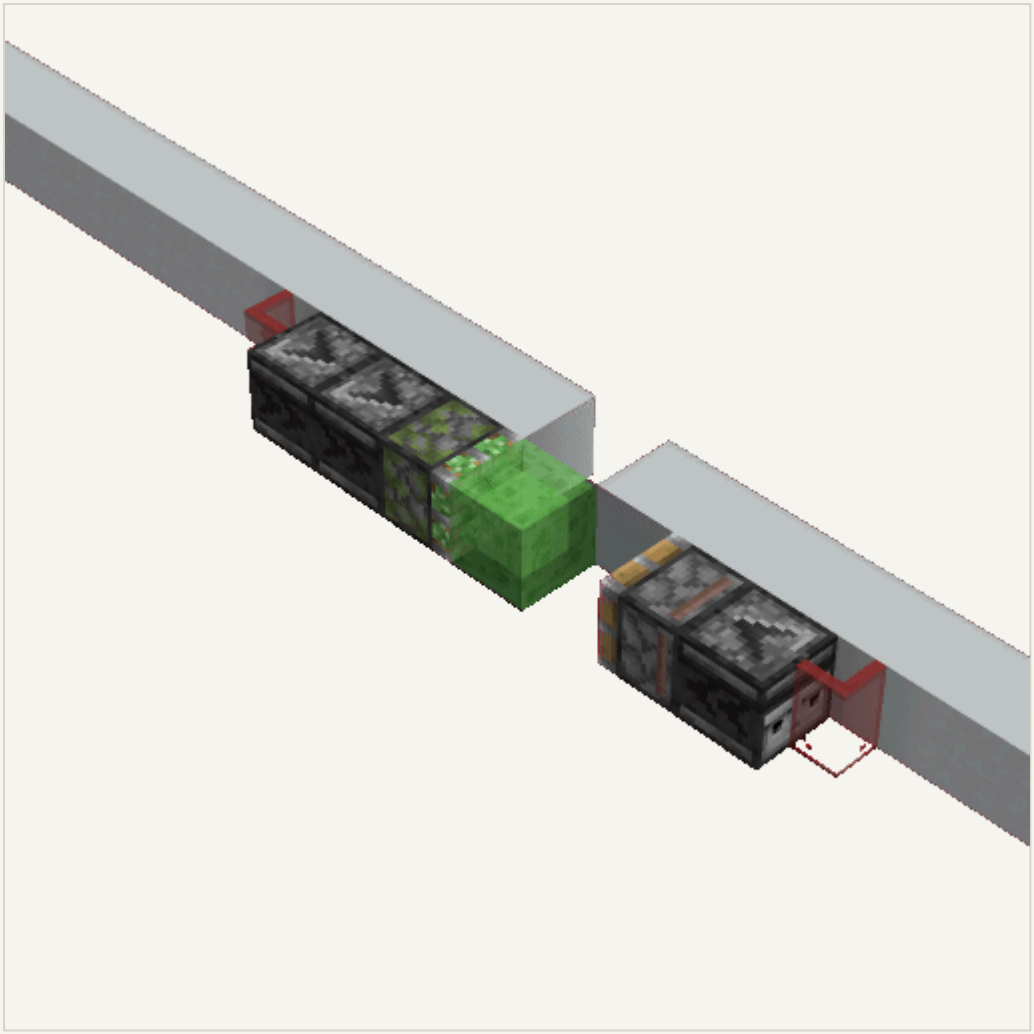


▶ This figure is animated. View original

In the above example, the red glass is instantly placed because:

The piston pushes the slime stick, moving the red glass and sending an NC update, which triggers the headless sticky piston's self-check. At this point, the sticky piston can successfully pull back, and there is b36 one block outside the extension direction. That b36's push direction matches the sticky piston's push direction, so it is instantly placed.

Of course, note that this instant placement only affects one block: the one directly in front of the sticky piston. Other blocks will still be converted to b36. We can use this to explain a device that seems like magic to many people:

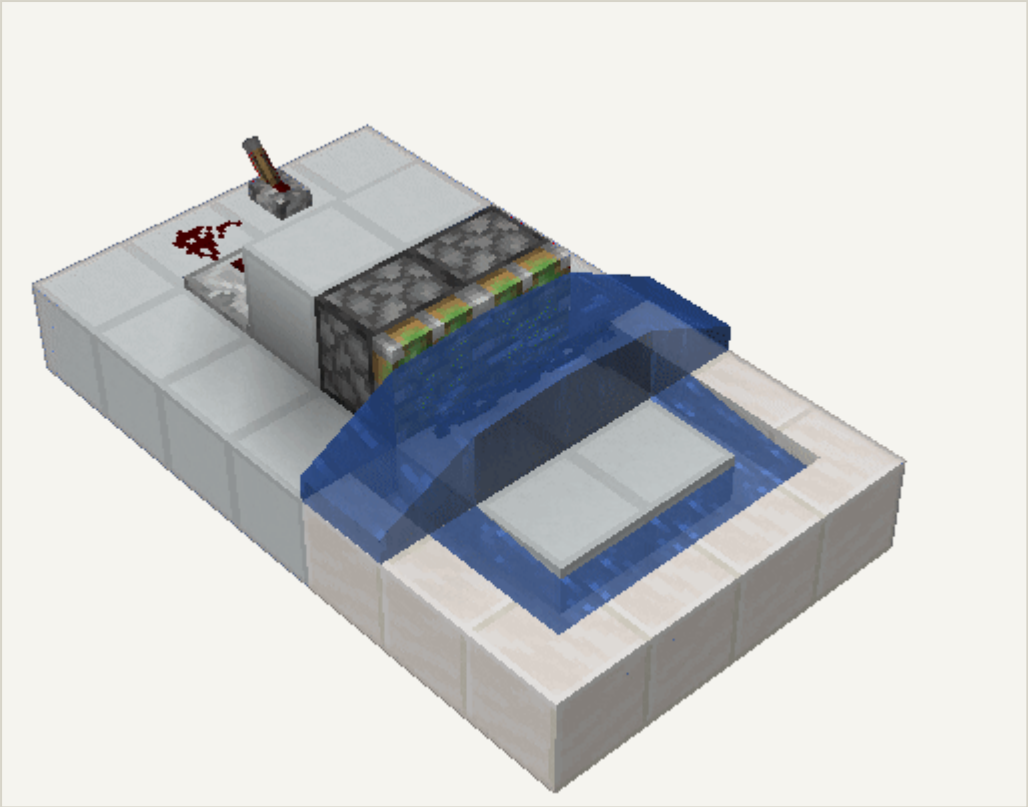


▶ This figure is animated. View original

In this device, when the lever is pulled, the sticky piston moves the block on its side 1gt after the block is placed. At this point, the block is carried forward by the slime block, but because the sticky piston is activated by the observer, the slime block is placed early and pushed back by the piston. Since the block is still in the b36 state, it gets left behind in the position it was carried to, thus extending the block flow.

Instant placement directly sets the progress to 1.0, removes the b36 block entity, replaces it with the original block, and sends updates. Note that instant push does not include displacement push/pull calculations for surrounding entities.

Note that since Mojang didn't include any entity calculations or special handling for waterlogged blocks in this part, instant push won't affect entities and won't remove waterlogged status. However, if TE-phase entity calculations occurred previously, those calculations are preserved; only the current tick's TE-phase entity calculations are ignored.



▶ This figure is animated. View original

```
            ++frontBlockCount; // Increase offset
```

}

After completing the pull analysis, the game proceeds to handle the push part. Starting from the block we initially analyzed, it searches forward along the movement direction for blocks to be pushed. If a traversed block is already in the push list, the linear structure ends there. Further traversal would collide with previously analyzed structures, at which point the push block list is rearranged.

Traverse the blocks that have been processed (depending on the addition order). If a block is sticky, enter branch structure checking to find blocks stuck to the sides. If side sticking causes push failure, return the push failure result.

If the block directly ahead is air, the linear structure ends. If the block directly ahead is an immovable block or the piston itself, push fails. If the block directly ahead is a destroyable block, add that block to the destroyable block list and the linear structure ends.

Add all blocks to the moved block list. Check the moved block list; if its size exceeds 12, the push limit is reached and push fails.

Attempting to Add Branches `tryMoveAdjacentBlock`

```java
private boolean tryMoveAdjacentBlock(BlockPos pos) {
    // Get the block state at the specified position
```

```
    BlockState targetBlockState = this.world.getBlockState(pos);
    // Iterate through all directions
    for (Direction direction : Direction.values()) {
        // Only process directions with different axes from the push direction
        if (direction.getAxis() != this.motionDirection.getAxis()) {
            // Get the adjacent block
            BlockPos adjacentBlockPos = pos.offset(direction);
            BlockState adjacentBlockState = this.world.getBlockState(adjacentBlockPos);

            // If current block and adjacent block are sticky blocks, try to move the adjacent block
            if (isAdjacentBlockStuck(adjacentBlockState, targetBlockState) &&
!this.tryMove(adjacentBlockPos, direction)) {
                return false; // If adjacent block cannot be moved, return false
            }
        }
    }

    // If all adjacent blocks are successfully moved, return true
    return true;
}
```

The input parameter block here is always a sticky block, so the following logic is executed:

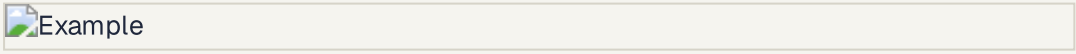
1. Iterate through all surrounding blocks, implemented by iterating through directions, in the same order as NC updates
2. Since analyzing the two adjacent blocks in the same direction as the push axis is unnecessary, they are excluded here. If a side block is mutually stuck with this block, attempt to analyze its linear structure. If linear structure addition fails, return failure result level by level.

1.6.3 Moved Block List Processing

`movedBlocks` and `brokenBlocks` reversed directly give the b36 arrival order and destruction order. We'll explain the reason in detail later.

1.6.4 Analysis of b36 Arrival Order

Let's analyze the following example:



The push axis in this diagram is +z towards -z direction

First analyze the linear structure. The two slime blocks in front of the piston head are added to the moved block list first.

Since both blocks are sticky, the slime block closest to the piston attempts to analyze the regular block in the -x direction and adds it to the push block list, then analyzes the slime block in the +y direction and adds it to the push block list. It will have a next call, but this call task is pushed onto the system stack. The second block in the linear structure is pushed onto the stack to analyze the slime block in the -x direction. This slime block finds it conflicts with the regular block behind it, so it handles the collision. The merged new list

has the regular block at index 3 and the slime block at index 4. The task of checking surrounding blocks for the slime block is pushed onto the system stack, waiting for the next round of calls. Finally, by analogy, the push order of the entire structure can be obtained.

Of course, if you're feeling lazy, just use pistorder directly.

1.7 Piston Event Response and Moving Block Source Code

1.7.1 Unified Entry Point for Events `onSyncedBlockEvent`

```java
public boolean onSyncedBlockEvent(BlockState state, World world, BlockPos pos, int actionType, int directionData) {
    // Get the block's direction (e.g., the direction the piston is facing)
    Direction facing = (Direction)state.get(FACING);
    // Get the activated block state (indicating the block is extending)
    BlockState extendedState = (BlockState)state.with(EXTENDED, true);
    // If not on the client side, perform some server-side operations
    if (!world.isClient) {
        // Check if the piston needs to extend (whether there's a redstone signal)
        boolean shouldExtend = this.shouldExtend(world, pos, facing);
        // If it needs to extend and the event type is retract, update block state and return
        if (shouldExtend && (actionType == 1 || actionType == 2)) {
            world.setBlockState(pos, extendedState, 2);
            return false;
        }
        // If it doesn't need to extend and the event type is 0, return false directly
        if (!shouldExtend && actionType == 0) {
            return false;
        }
    }
    // Event type 0: piston extension event
    if (actionType == 0) {
        // If moving blocks fails, return false
        if (!this.move(world, pos, facing, true)) {
            return false;
        }
        // Set block to extended state and play extension sound
        world.setBlockState(pos, extendedState, 67);
        world.playSound((PlayerEntity)null, pos, SoundEvents.BLOCK_PISTON_EXTEND, SoundCategory.BLOCKS, 0.5F, world.random.nextFloat() * 0.25F + 0.6F);
        world.emitGameEvent(GameEvent.BLOCK_ACTIVATE, pos, Emitter.of(extendedState));
    }
    // Event type 1 or 2: piston retraction event
    else if (actionType == 1 || actionType == 2) {
        // Get the block entity in front of the piston
        BlockEntity blockEntityInFront = world.getBlockEntity(pos.offset(facing));
        // If there's a piston block entity at that position, complete the retraction operation
        if (blockEntityInFront instanceof PistonBlockEntity) {
            ((PistonBlockEntity)blockEntityInFront).finish();
        }
        // Create moving piston block state
        BlockState pistonHeadState =
        Blocks.MOVING_PISTON.getDefaultState().with(PistonExtensionBlock.FACING, direction)
                .with(PistonExtensionBlock.TYPE, this.sticky ? PistonType.STICKY :
                PistonType.DEFAULT);
```

```
        // Set new piston block state and create new block entity
        world.setBlockState(pos, pistonHeadState, 20);
        world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(pos, pistonHeadState,
        (BlockState)this.getDefaultState().with(FACING, Direction.byId(directionData & 7)), direc-
        tion, false, true));
        world.updateNeighbors(pos, pistonHeadState.getBlock());
        pistonHeadState.updateNeighbors(world, pos, 2);
        // If it's a sticky piston, execute special logic
        if (this.sticky) {
            // Calculate target position
            BlockPos targetpos = pos.add(facing.getOffsetX() * 2, facing.getOffsetY() * 2,
            facing.getOffsetZ() * 2);
            BlockState targetState = world.getBlockState(targetpos);
            boolean hasMovingPistonInFront = false;
            // If target position already has a moving piston block, try to complete its extension
            if (targetState.isOf(Blocks.MOVING_PISTON)) {
                BlockEntity blockEntity2 = world.getBlockEntity(targetpos);
                if (blockEntity2 instanceof PistonBlockEntity) {
                    PistonBlockEntity pistonBlockEntity = (PistonBlockEntity)blockEntity2;
                    if (pistonBlockEntity.getFacing() == facing && pistonBlockEntity.isExtend-
                    ing()) {
                        pistonBlockEntity.finish();
                        hasMovingPistonInFront = true;
                    }
                }
            }
            // If no special case, remove target block or continue moving
            if (!hasMovingPistonInFront) {
                if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world, tar-
                getpos, facing.getOpposite(), false, direction)
                        || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
                        !targetState.isOf(Blocks.PISTON) &&
                        !targetState.isOf(Blocks.STICKY_PISTON)) {
                    world.removeBlock(pos.offset(facing), false);
                } else {
                    this.move(world, pos, facing, false);
                }
            }
        }
### :advanced Piston Self-Check Logic and Source Code Analysis
#### Piston Self-Check Function
Piston self-checks are triggered by the following methods, with the core being that they only call
the unified `trymove` method on the server (`!world.isClient`) side when specific conditions are
met:
```java
//Placed by player
public void onPlaced(World world, BlockPos pos, BlockState state, LivingEntity placer, ItemStack
itemStack) {
    if (!world.isClient) {
        this.tryMove(world, pos, state);
    }
}
//NC update
public void neighborUpdate(BlockState state, World world, BlockPos pos, Block sourceBlock,
BlockPos sourcePos, boolean notify) {
    if (!world.isClient) {
        this.tryMove(world, pos, state);
```
```

```
    }
}
//Placed by other things
public void onBlockAdded(BlockState state, World world, BlockPos pos, BlockState oldState, boolean notify) {
    if (!oldState.isOf(state.getBlock())) {
        if (!world.isClient && world.getBlockEntity(pos) == null) {
            this.tryMove(world, pos, state);
        }
    }
}
```

1.7.2 Signal Detection Source Code `shouldExtend`

The piston checks the first-order adjacency of two core positions. The traversal order strictly checks the piston itself first, then the first-order adjacency of the upper core. The order of checking adjacency strictly follows the NC update order.

JAVA

```java
private boolean shouldExtend(RedstoneView world, BlockPos pos, Direction pistonFace) {
    // Traverse all directions (up, down, north, south, east, west)
    for (Direction direction : Direction.values()) {
        if (direction != pistonFace && world.isEmittingRedstonePower(pos.offset(direction), direction)) {
            return true;// If the piston receives a redstone signal, the piston should extend
        }
    }
    if (world.isEmittingRedstonePower(pos, Direction.DOWN)) {
        return true;  // If there is a redstone signal below, the piston should extend
    } else {
        // If there is no redstone signal below, check the block above the piston
        BlockPos blockPos = pos.up(); // Get the position of the block above the piston (this is for QC powering)
        for (Direction direction2 : Direction.values()) {
            // If the current direction is not down (the piston body's position), and the block in that direction emits a redstone signal
            if (direction2 != Direction.DOWN &&
            world.isEmittingRedstonePower(blockPos.offset(direction2), direction2)) {
                // If the block above the piston receives a redstone signal, the piston should extend
                return true;
            }
        }
        // If no block in any direction emits a redstone signal, the piston should not extend
        return false;
    }
}
```

1.7.3 Attempt to Move Source Code `tryMove`

`tryMove` determines the type of block event (`actionType`), 0 for extend, 1 for retract, 2 for instant retract

JAVA

```java
private void tryMove(World world, BlockPos pos, BlockState state) {
    // Get the piston's facing direction (through the FACING property in the state)
    Direction facing = (Direction)state.get(FACING);
```

```
    // Determine if there is a redstone signal
    boolean shouldExtend = this.shouldExtend(world, pos, facing);
    // If should extend and piston is not currently extended (EXTENDED property is false)
    if (shouldExtend && !(Boolean)state.get(EXTENDED)) {
        // Create a PistonHandler object to attempt to push blocks
        // If successfully pushed, add a block event
        if ((new PistonHandler(world, pos, facing, true)).calculatePush()) {
            world.addSyncedBlockEvent(pos, this, 0, facing.getId());
        }
    }
    // If should not extend and piston is currently extended (EXTENDED property is true)
    else if (!shouldExtend && (Boolean)state.get(EXTENDED)) {
        // Calculate the position in front of the piston head (current piston position offset 2
        units in the piston push direction)
        BlockPos pistonHeadFront = pos.offset(facing, 2);
        // Get the block state at the target position
        BlockState headFrontState = world.getBlockState(pistonHeadFront);
        // Initialize status code (default is 1)
        // Status code meanings:
        // 0: extend
        // 1: retract
        // 2: instant retract
        int actionType = 1;
        if (headFrontState.isOf(Blocks.MOVING_PISTON) // If the target block is a moving piston
                && headFrontState.get(FACING) == facing //and its facing matches the current
                piston
                && world.getBlockEntity(pistonHeadFront)
                        instanceof PistonBlockEntity pistonBlockEntity // If the block entity is a
                        moving piston (b36)
                && pistonBlockEntity.isExtending()// If the target b36 is extending
                && (pistonBlockEntity.getProgress(0.0F) < 0.5F //If extension progress is less
                than 50%
                        || world.getTime() == pistonBlockEntity.getSavedWorldTime()
                        || ((ServerWorld)world).isInBlockTick())
        ) {
            actionType = 2;// Should instant retract
        }
        // Add block event
        world.addSyncedBlockEvent(pos, this, actionType, facing.getId());
    }
}
```

```
    }
    // If it's not a sticky piston, directly remove the target block
    else {
        world.removeBlock(pos.offset(facing), false);
    }

    // Play retraction sound
    world.playSound(null, pos, SoundEvents.BLOCK_PISTON_CONTRACT, SoundCategory.BLOCKS, 0.5F,
world.random.nextFloat() * 0.15F + 0.6F);
    world.emitGameEvent(GameEvent.BLOCK_DEACTIVATE, pos, Emitter.of(pistonHeadState));
}

// Return true to indicate successful event handling
return true;
```

}

Server Side

If not on the client side, several server-specific events are executed:

- Check activation status. If activated and the block event is retract or instant retract, set the piston to extended state, perform pathfinding update, and the block event execution fails.
- If not activated and the block event is extend, the block event execution fails.

Extension Event

> Note: This part must be executed on both client and server sides

The piston will attempt to move the blocks in front, implemented through the `move` method. I'll supplement the details of this method later. If movement fails, the piston event execution fails.

Next, set the piston to extended state. Here the `bitflag` is `0b01000011` for placement removal update, pathfinding update, and NC update.

Finally, play the piston extension sound. Block event execution succeeds.

Instant Retraction and Retraction Event

First, get the piston head's block entity. If it's b36, make it arrive instantly.

Next, set the piston to b36. Here the `bitflag` is `0b00010100`, no PP update, pathfinding update.

Then set the piston as b36 block entity, first emit NC update, then emit PP update.

If this piston is a sticky piston, at the position 1 block beyond where the piston head extends, if there is a b36 1 block beyond where it tries to place the piston head and that b36's push direction matches the piston's, make the block arrive instantly.

If the piston behavior is not instant retract, the block is not air, the piston can move the block 1 block in front, and this block can be normally pulled back or is a piston or sticky piston, then attempt to pull back the front block. If the block 1 block in front cannot be pulled, delete the block at the piston head position.

If it's a regular piston, directly delete the block at the piston head position.

In any case, play the piston retraction sound. Block event execution succeeds.

Piston Moving Blocks `move`

```java
// retract=0: retract
// retract=1: extend
private boolean move(World world, BlockPos pos, Direction dir, boolean retract) {
    // Piston head position
    BlockPos headPos = pos.offset(dir);

    // If it's a retraction action and the target position is a piston head
    if (!retract && world.getBlockState(headPos).isOf(Blocks.PISTON_HEAD)) {
        // Set piston head to air
```

```
        // flag=0b00010100, no PP update, pathfinding update, client
        world.setBlockState(headPos, Blocks.AIR.getDefaultState(), 20);
    }

    // Create PistonHandler instance to calculate legality of push or pull operation
    PistonHandler pistonHandler = new PistonHandler(world, pos, dir, retract);

    // If push/pull operation cannot be executed, cannot move
    if (!pistonHandler.calculatePush()) {
        return false;
    } else {
        // Use hash table to store blocks to be moved and their states for subsequent processing
        Map<BlockPos, BlockState> map = Maps.newHashMap();

        // Create moved block list and get moved block list
        List<BlockPos> blocksToMove = pistonHandler.getMovedBlocks();

        // Used to record original states of blocks to be moved
        List<BlockState> blocksOriginal = Lists.newArrayList();

        // Iterate through blocks to be moved, save their original states to list and store in map
        for (int i = 0; i < blocksToMove.size(); ++i) {
            BlockPos blockPos2 = (BlockPos) blocksToMove.get(i);
            BlockState blockState = world.getBlockState(blockPos2);
            blocksOriginal.add(blockState);
            map.put(blockPos2, blockState);
        }

        // Get list of all block positions to be destroyed
        List<BlockPos> list3 = pistonHandler.getBrokenBlocks();

        // Used to record states of destroyed blocks
        BlockState[] blockStates = new BlockState[blocksToMove.size() + list3.size()];

        // Determine movement direction
        Direction direction = retract ? dir : dir.getOpposite();

        // Process destroyed blocks, from back to front
        int j = 0;
        for (int k = list3.size() - 1; k >= 0; --k) {
            BlockPos blockPos3 = (BlockPos) list3.get(k);
            BlockState blockState2 = world.getBlockState(blockPos3);

            // If block has block entity, handle drop logic first
            BlockEntity blockEntity = blockState2.hasBlockEntity() ? world.getBlockEntity(blockPos3) :
null;
            dropStacks(blockState2, world, blockPos3, blockEntity);

            // Set block to air and trigger destroy event
            world.setBlockState(blockPos3, Blocks.AIR.getDefaultState(), 18);
            world.emitGameEvent(GameEvent.BLOCK_DESTROY, blockPos3, Emitter.of(blockState2));

            // If block is not a fire-type block, add destruction particle effect
            if (!blockState2.isIn(BlockTags.FIRE)) {
                world.addBlockBreakParticles(blockPos3, blockState2);
            }
```

```
            // Save destroyed block state
            blockStates[j++] = blockState2;
        }

        // Process blocks to be moved
        for (int k = blocksToMove.size() - 1; k >= 0; --k) {
            BlockPos targetPos = (BlockPos) blocksToMove.get(k);
            BlockState targetState = world.getBlockState(targetPos);

            // Move to target position
            targetPos = targetPos.offset(direction);
            map.remove(targetPos);

            // Set target position to MOVING_PISTON state for animation handling
            BlockState blockState3 = (BlockState) Blocks.MOVING_PISTON.getDefaultState().with(FACING,
dir);
            world.setBlockState(targetPos, blockState3, 68);

            // Create b36 block entity to control movement animation
            world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(targetPos, blockState3,
(BlockState) blocksOriginal.get(k), dir, retract, false));
            blockStates[j++] = targetState;
        }

        // If it's an extension operation, add piston head
        if (retract) {
            PistonType pistonType = this.sticky ? PistonType.STICKY : PistonType.DEFAULT;
            BlockState blockState4 = (BlockState) ((BlockState)
Blocks.PISTON_HEAD.getDefaultState().with(PistonHeadBlock.FACING, dir)).with(PistonHeadBlock.TYPE, pis-
tonType);
            BlockState targetState = (BlockState) ((BlockState)
Blocks.MOVING_PISTON.getDefaultState().with(PistonExtensionBlock.FACING,
dir)).with(PistonExtensionBlock.TYPE, this.sticky ? PistonType.STICKY : PistonType.DEFAULT);
            map.remove(headPos);

            // Set current piston position to MOVING_PISTON state
            world.setBlockState(headPos, targetState, 68);

            // Add b36 block entity to control animation

### :advanced B36 Conversion and Structure Analysis Source Code

> Friendly reminder, this is one of the most complex parts of this chapter. If you prefer, you can use
the pistOrder mod directly without studying this section

'''java
public boolean calculatePush() {
    // Clear the moved blocks and broken blocks lists to recalculate
    this.movedBlocks.clear();
    this.brokenBlocks.clear();

    // Get the starting point that the piston will push
    BlockState pistonHeadFrontState = this.world.getBlockState(this.posTo);

    // If the starting point cannot be pushed
    if (!PistonBlock.isMovable(pistonHeadFrontState, this.world, this.posTo, this.motionDirection,
false, this.pistonDirection)) {
```

```
        // If the block is immovable and currently in retracted state, and the block behavior is
DESTROY, mark it for destruction
        if (this.retracted && pistonHeadFrontState.getPistonBehavior() == PistonBehavior.DESTROY) {
            this.brokenBlocks.add(this.posTo); // Add to broken list
            return true; // Piston can still operate (by destroying blocks)
        } else {
            return false; // Piston cannot push and cannot destroy target block
        }
    }
    // If the target block is movable but cannot add linear structure, cannot push
    else if (!this.tryMove(this.posTo, this.motionDirection)) {
        return false;
    }
    else {
        // Traverse all blocks marked as needing to move
        for (int i = 0; i < this.movedBlocks.size(); ++i) {
            BlockPos posToValidate = (BlockPos) this.movedBlocks.get(i);

            // If the current block is a sticky block, try to move its surrounding sticky blocks
            if (isBlockSticky(this.world.getBlockState(posToValidate)) &&
!this.tryMoveAdjacentBlock(posToValidate)) {
                return false; // If moving surrounding sticky blocks fails, overall push fails
            }
        }

        return true; // All blocks successfully moved or processed
    }
}
```

When a piston needs to push, it will first execute the above code, which is used to analyze the movement structure. More detailed descriptions are needed here to help readers understand: `movedBlocks` and `brokenBlocks` are actually two `arrayList`s. Readers unfamiliar with Java can think of them as lists that support arbitrary insertion and retrieval. This code essentially executes operations in the following order:

1. Clear the `movedBlocks` and `brokenBlocks` lists.
2. Check if the starting position can be pushed
    - If it cannot be pushed, check its response to piston behavior
        - If it can be destroyed, add the block to the `brokenBlocks` list, push succeeds
        - If it cannot be destroyed, push fails
    - If it can be pushed, try to add its linear structure (we will analyze how pistons analyze linear structures later)
        - If the linear structure cannot be added, push fails
        - If the linear structure is successfully added, traverse all blocks that need to be pushed
            - If there are sticky blocks (slime blocks and honey blocks), try to add branches
                - If addition fails, push fails
        - If none of the above failed, push succeeds

Mojang implemented the analysis of linear structures and branch structures through two methods here, called `tryMove`[1] and `tryMoveAdjacentBlock`.

1.7.4 Linear Structure Analysis `tryMove`

First, clarify when this method will be called:

1. This block is directly pushed by a block
2. This block is directly pulled by a sticky block

It's worth noting that these two methods are strictly independent of each other, meaning there is no overlap between the two situations.

When called for the first reason, this block itself must not be an immovable block (see section 5.4.2 above for details). When called for the second reason, if this block itself is immovable, there is no need to analyze it since the block trying to pull it is sticky (this is the situation shown in the figure below).

With that analysis complete, let's examine the first part of this method's code.

```
        world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(headPos, targetState,
blockState4, dir, true, true));
    }

    // Set remaining blocks in map to air
    BlockState blockState5 = Blocks.AIR.getDefaultState();
    for (BlockPos blockPos4 : map.keySet()) {
        world.setBlockState(blockPos4, blockState5, 82);
    }

    // Update neighbor block states
    for (Map.Entry<BlockPos, BlockState> entry : map.entrySet()) {
        BlockPos blockPos5 = entry.getKey();
        BlockState blockState6 = entry.getValue();
        blockState2.prepare(world, blockPos5, 2);
        blockState5.updateNeighbors(world, blockPos5, 2);
        blockState5.prepare(world, blockPos5, 2);
    }

    // Update neighbor states of destroyed blocks
    for (int l = list3.size() - 1; l >= 0; --l) {
        BlockState targetState = blockStates[j++];
        BlockPos blockPos5 = (BlockPos) list3.get(l);
        targetState.prepare(world, blockPos5, 2);
        world.updateNeighborsAlways(blockPos5, targetState.getBlock());
    }

    // Update neighbor states of moved blocks
    for (int l = blocksToMove.size() - 1; l >= 0; --l) {
        world.updateNeighborsAlways((BlockPos) blocksToMove.get(l), blockStates[j++].getBlock());
    }

    // If it's a retraction action, update piston position neighbors
    if (retract) {
        world.updateNeighborsAlways(headPos, Blocks.PISTON_HEAD);
    }
```

```
    return true; // If execution succeeds, return true
}
```

}

When the piston attempts to move blocks, it distinguishes between two cases: extension and retraction. The `retract` boolean represents the piston's action. When false, the piston retracts; when true, the piston extends. The piston then handles these two actions separately.

When the piston **attempts** to retract, the game first checks whether the expected piston head position actually contains a piston head. Retraction only proceeds if a piston head is present. Each time retraction executes, the game first sets the piston head position to air (deleting the piston head). The `setBlockState` bitflag at this point is `0b00010100`, meaning no PP update and client pathfinding update.

Before starting movement, whether retracting or extending, the game calls the `calculatePush` method analyzed in section [5.4.2](#进阶542-移动结构分析) to verify that movement is possible. If movement cannot succeed, the push/pull operation fails.

The game then begins moving blocks. It first processes the destroyed block list (`brokenBlocks`) in reverse order. If a destroyed block has a block entity that drops items, the items are dropped, then the block is set to air. At this point, no PP update is sent, but a pathfinding update is emitted.

After destroying the necessary blocks, the game begins moving blocks. Traverse the moved block list (`movedBlocks`) from back to front, set each block to b36 with `bitflag` `0b01101000` (placement removal update), then add a b36 block entity and store it in the `blockStates` list for later use.

If extending, create b36 at the piston head position (placement removal update), then add a b36 block entity to the piston head position.

After completing the above operations, set all uncovered blocks to air (placement removal update and pathfinding update). No PP update is sent at this point. Updates are then emitted uniformly: for all blocks in the hash table, first emit a redstone dust prepare update, then a block update (at the block's original position). If extending, the piston head completes its block update.

The above code well explains why b36 addition is in reverse order to the `movedBlocks` list. Additionally, since the hash table is unordered, this also explains why some machines that depend on b36 arrival order break after unloading (the original index is lost).

:advanced Arrival Source Code Analysis

Normal Arrival `tick`

> This is straightforward, so the detailed explanation was provided earlier. Therefore, only code is provided here for reference

```java
public static void tick(World world, BlockPos pos, BlockState state, PistonBlockEntity blockEntity) {
        // Record current gt
    blockEntity.savedWorldTime = world.getTime();
    blockEntity.lastProgress = blockEntity.progress;
    // If push is completed
    if (blockEntity.lastProgress >= 1.0F) {
        // If it's client side and current deathTick hasn't exceeded 5
```

```
        if (world.isClient && deathTicks < 5) {
            // Increase deathTicks count
            ++deathTicks;
        } else {
            // Remove block entity
            world.removeBlockEntity(pos);
            blockEntity.markRemoved();
            // If this block is b36
            if (world.getBlockState(pos).isOf(Blocks.MOVING_PISTON)) {
                // Give PP update
                BlockState blockState = Block.postProcessState(blockEntity.pushedBlock, world, pos);
                // If it's air
                if (blockState.isAir()) {
                    // Set to corresponding block, no update, bitflag=0b01010100;
                    world.setBlockState(pos, blockEntity.pushedBlock, 84);
                    // Update or destroy this block
                    Block.replace(blockEntity.pushedBlock, blockState, world, pos, 3);
                } else {
                    // If it's a waterlogged block
                    if (blockState.contains(Properties.WATERLOGGED) &&
(Boolean)blockState.get(Properties.WATERLOGGED)) {
                        // Remove water from waterlogged block
                        blockState = (BlockState)blockState.with(Properties.WATERLOGGED, false);
                    }
                    // Set to corresponding block, give NC update, PP update, client update,
bitflag=0b01000011
                    world.setBlockState(pos, blockState, 67);
                    // Receive NC update itself
                    world.updateNeighbor(pos, blockState.getBlock(), pos);
                }
            }
        }
    }
    // If push is not completed
    // Push progress +0.5
    else {
        float f = blockEntity.progress + 0.5F;
        // Perform entity calculations
        pushEntities(world, pos, f, blockEntity);
        moveEntitiesInHoneyBlock(world, pos, f, blockEntity);
        blockEntity.progress = f;
        // If push progress >=1, set to 1
        if (blockEntity.progress >= 1.0F) {
            blockEntity.progress = 1.0F;
        }
    }
}
```

1.7.5 Instant Arrival `finish`

```java
public void finish() {
    // World exists and progress not full or client calculation
    if (this.world != null && (this.lastProgress < 1.0F || this.world.isClient)) {
        // Set progress to complete
        this.progress = 1.0F;
        this.lastProgress = this.progress;
```

```
        // Remove block entity
        this.world.removeBlockEntity(this.pos);
        this.markRemoved();
        // If this is b36 block
        if (this.world.getBlockState(this.pos).isOf(Blocks.MOVING_PISTON)) {
            // If it's piston arm
            BlockState blockState;
            if (this.source) {
                // Set to air
                blockState = Blocks.AIR.getDefaultState();
            } else {
                // Otherwise arrive after PP update
                blockState = Block.postProcessState(this.pushedBlock, this.world, this.pos);
            }
            // Set block, PP update, pathfinding update, NC update
            this.world.setBlockState(this.pos, blockState, 3);
            // Receive block update itself
            this.world.updateNeighbor(this.pos, blockState.getBlock(), this.pos);
        }
    }
}
```

JAVA

```
BlockState stateToValidate = this.world.getBlockState(posToValidate);
// If the block is air, directly return true, because air doesn't need processing
if (stateToValidate.isAir()) {
    return true;
}
// If the block is immovable, directly return true based on current conditions
else if (!PistonBlock.isMovable(stateToValidate, this.world, posToValidate, this.motionDirection,
false, dir)) {
    return true;
}
// If the block position is the piston's initial position, directly return true
else if (posToValidate.equals(this.posFrom)) {
    return true;
}
// If the block is already in the moved list, directly return true to avoid duplicate calculation
else if (this.movedBlocks.contains(posToValidate)) {
    return true;
}
```

The logic here is as follows:

1. If the piston is trying to push air, then it can push
2. If this block is immovable, then it can push (see the justification above)
3. If this block is the piston itself, then it can push
4. If this block is already being pushed, then it can push

JAVA

```
while (isBlockSticky(stateToValidate)) {
// Get the next block in the opposite direction of the sticky block
    BlockPos blockBehind = posToValidate.rearBlockCount(this.motionDirection.getOpposite(),
    rearBlockCount);
```

```
    BlockState currentStateSaved = stateToValidate; // Current block state
    stateToValidate = this.world.getBlockState(blockBehind); // Next block state
    // If the next block is air, not sticky, immovable, or is the piston's initial position, end loop
    if (stateToValidate.isAir()
    || !isAdjacentBlockStuck(currentStateSaved, stateToValidate)
    || !PistonBlock.isMovable(stateToValidate, this.world, blockBehind, this.motionDirection, false, this.motionDirection.getOpposite())
            || blockBehind.equals(this.posFrom))
    {
            break;
    }
    ++rearBlockCount; // Increase offset distance
    if (rearBlockCount + this.movedBlocks.size() > 12) {
        // If the total number of moved blocks exceeds 12, return false
        return false;
    }
}
```

Next, the game defines a variable `offset`, which represents the block the piston is trying to push. If the `movedBlocks` list size plus 1 already exceeds the push limit, the push fails (adding this block would exceed the pushable size limit).

The game starts from this block and recursively pushes blocks in the opposite direction of the push direction, looking for where the linear pull is interrupted. When this position is air, a block that doesn't stick to this sticky block, an immovable block, or the piston itself, this row of blocks is the part pulled by the sticky block (the purpose of this part is to handle the following situation:)

If the newly found linear structure's pull part plus the original `movedBlocks` list size already exceeds 12, the push fails.

Next, add this pulled part to the moved blocks list in the reverse direction of the push direction: the frontmost block is added first, the rearmost block is added last.

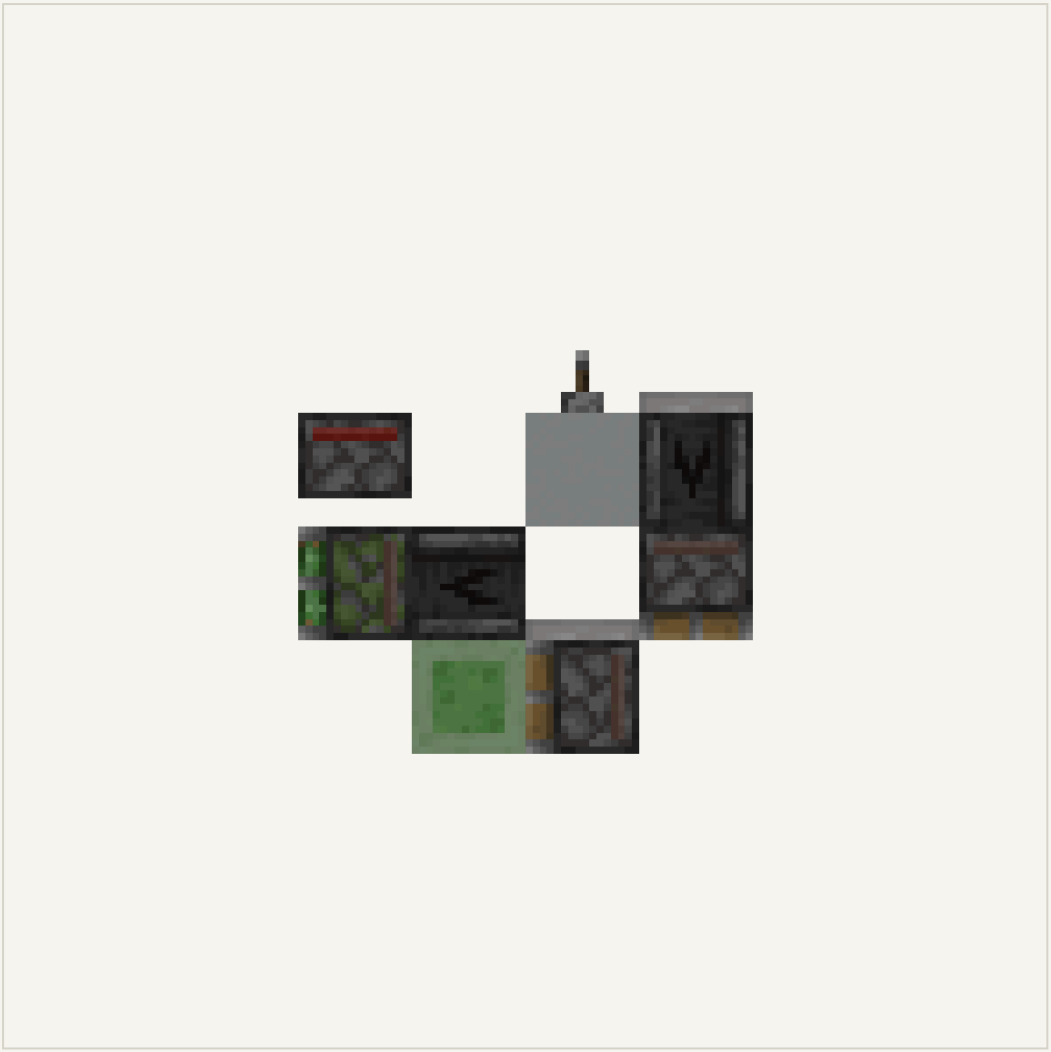
Most of the content has been explained before, so we'll skip ahead here. We only look at the b36 special case in the latter half.

In 1.20, only the piston arm's `source` attribute is `true`; all others are `false`. Therefore, when this method is called, if the position is a b36 with the target block as a piston arm, it will directly become air. The rest won't be detailed, as it's all been covered before.

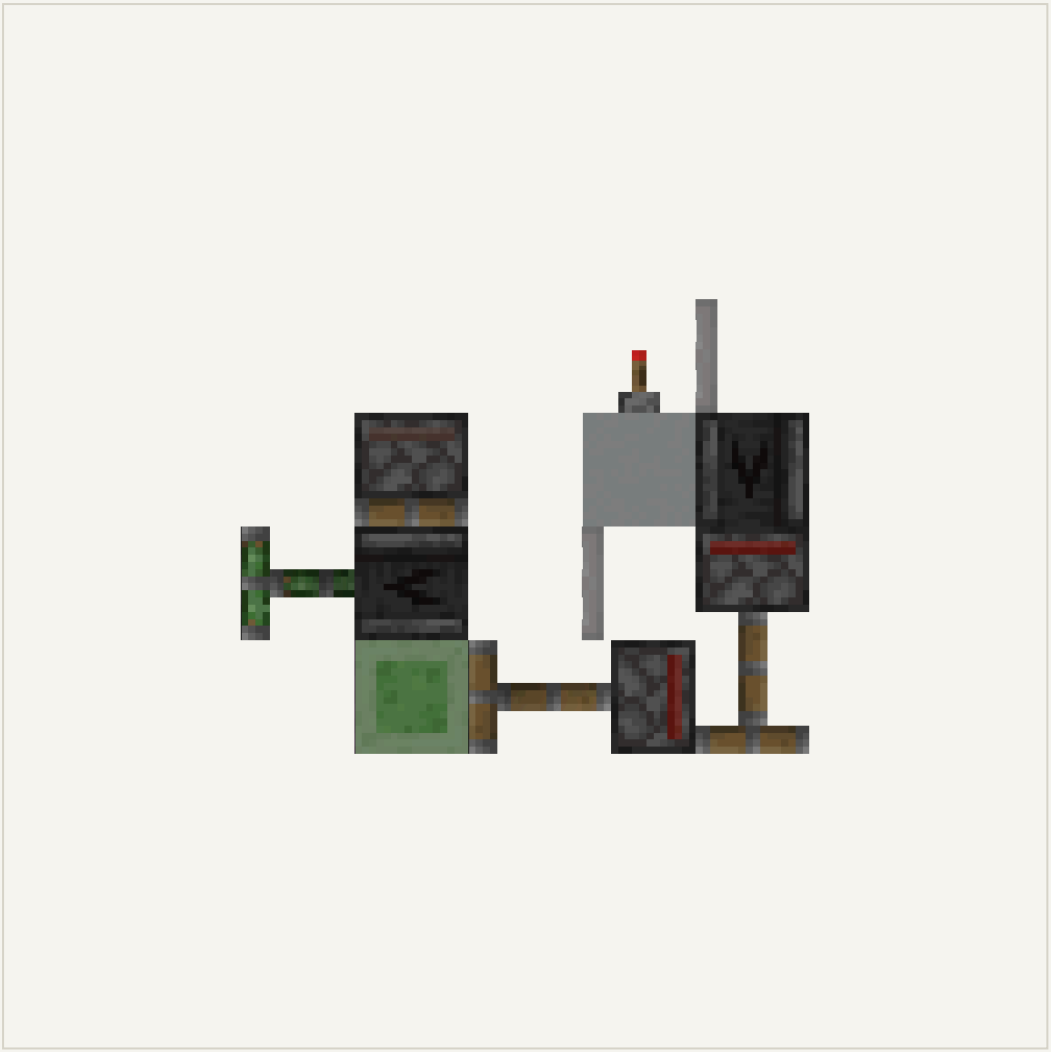
1.8 Piston Head

> This chapter actually isn't difficult, but since apart from the device demonstrated by Menggui233 in PetrisAFE's video recently, no other uses have been seen, it's placed in the advanced section here.

We first observe the following device:



Now, pull down the lever



Observe that the piston base was replaced with an observer, while the piston head was retained. Why?

We first review what happens when b36 arrives:

- Give PP update at its own position
- Give NC and PP updates to first-order neighbors at its own position
- Give NC update at its own position

Therefore, we separately examine what happens when the piston head receives PP update and NC update

1.8.1 Piston Head Existence Condition Determination `canSurvive`

```java
private boolean isFittingBase(BlockState baseState, BlockState extendedState) {
    Block block = baseState.getValue(TYPE) == PistonType.DEFAULT ? Blocks.PISTON :
    Blocks.STICKY_PISTON;
    return extendedState.is(block) && extendedState.getValue(PistonBaseBlock.EXTENDED) &&
    extendedState.getValue(FACING) == baseState.getValue(FACING);
}
@Override
public boolean canSurvive(BlockState state, LevelReader level, BlockPos pos) {
    BlockState blockState = level.getBlockState(pos.relative(state.getValue(FACING).getOppo-
    site()));
```

```
    return this.isFittingBase(state, blockState) || blockState.is(Blocks.MOVING_PISTON) &&
    blockState.getValue(FACING) == state.getValue(FACING);
}
```

The `isFittingBase` method essentially states that a piston head is `fitting` if and only if the block behind it (more precisely, 1 block in the opposite direction of the push direction, hereafter referred to as behind) is an extending piston with the same extension direction. The `canSurvive` method requires that a piston head's existence is valid if and only if the piston head is `fitting`, or there is a b36 behind it with the same push direction.

1.8.2 Piston Head Removal `onRemove`

JAVA

```java
@Override
public void onRemove(BlockState state, Level level, BlockPos pos, BlockState newState, boolean isMoving) {
    if (!state.is(newState.getBlock())) {
        super.onRemove(state, level, pos, newState, isMoving);
        BlockPos blockPos = pos.relative(state.getValue(FACING).getOpposite());
        if (this.isFittingBase(state, level.getBlockState(blockPos))) {
            level.destroyBlock(blockPos, true);
        }
    }
}
```

One sentence summary: When removed, if the state is `fitting`, the base is removed along with it and the piston drops.

1.8.3 Piston Head Receiving PP Update and NC Update

JAVA

```java
@Override
public BlockState updateShape(BlockState state, Direction facing, BlockState facingState, LevelAccessor level, BlockPos currentPos, BlockPos facingPos) {
    return facing.getOpposite() == state.getValue(FACING) && !state.canSurvive(level, currentPos)
        ? Blocks.AIR.defaultBlockState()
        : super.updateShape(state, facing, facingState, level, currentPos, facingPos);
}
@Override
public void neighborChanged(BlockState state, Level level, BlockPos pos, Block block, BlockPos fromPos, boolean isMoving) {
    if (state.canSurvive(level, pos)) {
        level.neighborChanged(pos.relative(state.getValue(FACING).getOpposite()), block, fromPos);
    }
}
```

PP Update

The piston head actually only cares about PP updates from behind. When a PP update occurs behind, the piston head calls `canSurvive` to check if it's valid. If invalid, it disappears immediately.

NC Update

In any case, the piston head will not disappear due to receiving an NC update. When receiving an NC update, if it's valid, the update is treated as an NC update received by the block behind.

1.8.4 Case Analysis

Therefore, for the case mentioned at the beginning of the chapter:

- 0gt AT: Player pulls down lever, trapdoor triggers
- 2gt BE depth 0: Sticky piston is activated, updates headless piston
- 2gt BE depth 1: Headless piston eats sticky piston base, slime block & lit observer are pushed
- 4gt TE: Sticky piston head arrives first, checks that behind is b36 in the same direction as itself, self-check passes. Then lit observer b36 arrives, no update.

Therefore, the observer successfully replaced the original piston base.

1.9 b36-Related Entity Calculations

```java
// This method handles piston pushing of entities. It gets the movement direction and distance of b36 in the current game tick, and finds all entities that may contact along the path. Then, it precisely calculates the maximum overlap of each entity with b36 in the movement direction, and pushes the entity accordingly (while adding a tiny displacement of 0.01 to prevent getting stuck). This process also specially handles the bounce effect of slime blocks and the effect of b36 retraction on entities.
private static void pushEntities(World world, BlockPos pos, float f, PistonBlockEntity blockEntity) {
    // Get the direction b36 is moving
    Direction direction = blockEntity.getMovementDirection();
    // Calculate the distance b36 needs to move in the current tick
    double d = f - blockEntity.progress;
    // Get the collision box voxel model of b36
    VoxelShape voxelShape = blockEntity.getHeadBlockState().getCollisionShape(world, pos);
    // Check if b36 has actual collision volume. If not, do nothing
    if (!voxelShape.isEmpty()) {
    // Get the bounding box of b36 after moving this tick
    Box box = offsetHeadBox(pos, voxelShape.getBoundingBox(), blockEntity);
        // Based on the area scanned before and after b36 movement, roughly filter entities that may be pushed
        List<Entity> list = world.getOtherEntities(null, Boxes.stretch(box, direction, d).union(box));
```java
while (true) {
    BlockPos blockFront = posToValidate.offset(this.motionDirection, frontBlockCount); // Next block's position
    int l = this.movedBlocks.indexOf(blockFront); // Check if block is already in moved list
    if (l > -1) {
        // If block is already in moved list, adjust the order of the moved list
        this.setMovedBlocks(succeededCount, l);
        // Traverse the adjusted list, check all sticky blocks and process their connected blocks
                        for (int m = 0; m <= l + succeededCount; ++m) {
        BlockPos stuckBlock = (BlockPos) this.movedBlocks.get(m);
        if (isBlockSticky(this.world.getBlockState(stuckBlock))
            && !this.tryMoveAdjacentBlock(stuckBlock)) {
                return false;
```
```

```
            }
        }
        return true; // All related blocks processed successfully
    }
    // Get the block state at the current offset position
    stateToValidate = this.world.getBlockState(blockFront);
    // If the current block is air, push succeeds
                    if (stateToValidate.isAir()) {
            return true;
            }
            // If the current block is immovable or is the piston's initial position, return false
            if (!PistonBlock.isMovable(stateToValidate, this.world, blockFront, this.motionDirec-
            tion, true, this.motionDirection)
                        || blockFront.equals(this.posFrom)) {
            return false;
            }
            // If the current block is destroyable, add it to the broken list
            if (stateToValidate.getPistonBehavior() == PistonBehavior.DESTROY) {
            this.brokenBlocks.add(blockFront);
                        return true;
                                }
                                // If the moved list has reached 12 blocks, return false
                                if (this.movedBlocks.size() >= 12) {
            return false;
            }
        // Add the current block to the moved list
        this.movedBlocks.add(blockFront);
                ++succeededCount; // Increase moved count
                ++frontBlockCount; // Increase offset
}
```

After completing the pull part analysis, the game will continue to process the push part. Starting from the block we initially analyzed, search forward along the movement direction for blocks that will be pushed. If the traversed block is already in the push list, it means this linear structure ends. Further traversal would collide with the previously analyzed structure, at which point the push block list is rearranged.

Traverse the blocks that have been completed (depending on the addition order). If it is a sticky block, enter branch structure checking to find blocks stuck to the side. If side sticking causes push failure, return push failure result.

If the block directly in front is air, the linear structure ends. If the block directly in front is an immovable block or the piston itself, push fails. If the block directly in front is a destroyable block, add that block to the destroyable blocks list, linear structure ends.

Add all blocks to the moved blocks list. Check the moved blocks list; if its size is greater than 12, the push limit is reached, push fails.

1.9.1 Attempt to Add Branch `tryMoveAdjacentBlock`

JAVA

```
private boolean tryMoveAdjacentBlock(BlockPos pos) {
    // Get the block state at the specified position
    BlockState targetBlockState = this.world.getBlockState(pos);
    // Traverse all directions
    for (Direction direction : Direction.values()) {
```

```
        // Only process directions different from the push direction axis
        if (direction.getAxis() != this.motionDirection.getAxis()) {
            // Get the adjacent block
            BlockPos adjacentBlockPos = pos.offset(direction);
            BlockState adjacentBlockState = this.world.getBlockState(adjacentBlockPos);
            // If the current block and adjacent block are sticky blocks, try to move the adjacent
            block
            if (isAdjacentBlockStuck(adjacentBlockState, targetBlockState) && !this.tryMove(adja-
            centBlockPos, direction)) {
                return false; // If adjacent block cannot move, return false
            }
        }
    }
    // If all adjacent blocks successfully moved, return true
    return true;
}
```

```
    // If there are entities in the scan area, start processing
    if (!list.isEmpty()) {

        // Get the list of all sub-bounding boxes that make up the b36 voxel model
        List<Box> list2 = voxelShape.getBoundingBoxes();

        // Check if b36 is a slime block
        boolean bl = blockEntity.pushedBlockState.isOf(Blocks.SLIME_BLOCK);
        Iterator var12 = list.iterator();

        // Iterate through all entities that may be affected
        while (true) {
            Entity entity;
            while (true) {
                if (!var12.hasNext()) {
                    return;
                }

                entity = (Entity)var12.next();
                // Check if entity ignores b36 pushing
                if (entity.getPistonBehavior() != PistonBehavior.IGNORE) {
                    if (!bl) {
                        break;
                    }

                    // If slime block is pushing, for entities other than server players, force their
velocity component along the movement axis to 1.0
                    if (!(entity instanceof ServerPlayerEntity)) {
                        Vec3d vec3d = entity.getVelocity();
                        double e = vec3d.x;
                        double g = vec3d.y;
                        double h = vec3d.z;
                        switch (direction.getAxis()) {
                            case X:
                                e = direction.getOffsetX();
                                break;
                            case Y:
                                g = direction.getOffsetY();
                                break;
```

```
                            case Z:
                                h = direction.getOffsetZ();
                        }

                        entity.setVelocity(e, g, h);
                        break;
                    }
                }
            }

            // Record the maximum overlap distance between entity and b36 in the movement direction
            double i = 0.0;

            // Iterate through each sub-bounding box of the b36 model
            for (Box box2 : list2) {

                // Check if the movement path of b36's sub-bounding box intersects with the entity's
collision box
                Box box3 = Boxes.stretch(offsetHeadBox(pos, box2, blockEntity), direction, d);
                Box box4 = entity.getBoundingBox();

                // If they intersect, calculate the overlap distance
                if (box3.intersects(box4)) {
                    i = Math.max(i, getIntersectionSize(box3, direction, box4));

                    // Overlap distance is greater than actual push distance, this is the worst case,
no need for extra checks
                    if (i >= d) {
                        break;
                    }
                }
            }

            // If the final calculated maximum overlap distance is greater than 0, perform push
            if (!(i <= 0.0)) {

                // Determine the final push distance. This distance cannot exceed b36's own movement
distance at this moment
                // And add an extra 0.01 to prevent entity from getting stuck in b36 next tick
                i = Math.min(i, d) + 0.01;

                // Along the movement direction, push the entity the calculated distance
                moveEntity(direction, entity, i, direction);

                // If b36 is retracting, specially handle entities affected by it
                if (!blockEntity.extending && blockEntity.source) {
                    push(pos, entity, direction, d);
                }
            }
        }
    }
}
```

}

// This method is used to solve the problem of entities getting stuck inside b36. It checks if the entity overlaps with the block at b36's position, and if so, calculates the minimum distance to push the entity out in the opposite direction. After a precise overlap depth comparison, it calibrates and executes this displacement, ensuring the entity is smoothly pushed away from b36.
private static void push(BlockPos pos, Entity entity, Direction direction, double amount) {

```
// Get the entity's bounding box
Box box = entity.getBoundingBox();

// Create a standard-sized (1x1x1) block bounding box at b36's position
Box box2 = VoxelShapes.fullCube().getBoundingBox().offset(pos);

// If the entity's bounding box doesn't overlap with the block's bounding box, the entity is not stuck,
return directly
```

The input parameter block here is always sticky, so the following logic executes:

1. Traverse all surrounding blocks by iterating through directions, in the same order as NC updates
2. Since analyzing the two adjacent blocks in the same direction as the push axis is unnecessary, they are excluded here. If a side block is mutually stuck with this block, attempt to analyze its linear structure. If linear structure addition fails, return failure result recursively.

1.9.2 Moved Blocks List Processing

`movedBlocks` and `brokenBlocks` reversed directly give the b36 placement order and destruction order. The reason will be explained in detail later.

1.9.3 B36 Placement Order Analysis

Let's analyze the following example:



The push axis in this diagram is from +z to -z.

First, analyze the linear structure: the two slime blocks in front of the piston head are added to the moved blocks list first.

Since both blocks are sticky, the slime block closest to the piston attempts to analyze the regular block in the -x direction and adds it to the push block list, then analyzes the slime block in the +y direction and adds it to the push block list. It will have a next call, but this call task is pushed onto the system stack. The second block in the linear structure is pushed onto the stack to analyze the slime block in the -x direction. This slime block finds it conflicts with the regular block behind it, so it handles the collision. The merged list has the regular block at index 3 and the slime block at index 4. The task of checking surrounding blocks for the slime block is pushed onto the system stack, waiting for the next round of calls. Finally, by this process, the push order of the entire structure can be obtained.

Of course, if you're feeling lazy, just use pistorder directly.

```
if (box.intersects(box2)) {

    // Get the opposite direction of the original push
    Direction direction2 = direction.getOpposite();

    // Calculate the minimum push-out distance needed to free the entity from the block
    double d = getIntersectionSize(box2, direction2, box) + 0.01;

    // Calculate the depth of the actual overlap area between entity and block
    double e = getIntersectionSize(box2, direction2, box.intersection(box2)) + 0.01;

    // If the two calculated push-out distances are almost equal (indicating the entity is only
slightly touching), perform position calibration (float align)
    if (Math.abs(d - e) < 0.01) {
        d = Math.min(d, amount) + 0.01;

            // Along the opposite direction, push the entity the calculated distance
        moveEntity(direction, entity, d, direction2);
    }
}
```

}

// This method implements the sticky effect of moving honey blocks on entities above, and only works when b36 pushes horizontally. Its core logic is: define a "sticky area" above the honey block surface, then synchronously move all entities that meet the conditions in that area with the honey block b36.
private static void moveEntitiesInHoneyBlock(World world, BlockPos pos, float f, PistonBlockEntity blockEntity) {

```
// The entire logic only triggers when b36 is a honey block
if (blockEntity.isPushingHoneyBlock()) {

    // Get the direction b36 is moving
    Direction direction = blockEntity.getMovementDirection();

    // Honey block's sticky effect only works when pushing horizontally
    if (direction.getAxis().isHorizontal()) {

        // Get the actual top height of the honey block b36's collision box
        double d = blockEntity.pushedBlockState.getCollisionShape(world, pos).getMax(Direction.Axis.Y);

        // Define a sticky area above the honey block surface (from its top to Y=1.5), and make it move
synchronously with b36
        Box box = offsetHeadBox(pos, new Box(0.0, d, 0.0, 1.0, 1.5000010000000001, 1.0), blockEntity);

        // Calculate the distance the honey block b36 needs to move in the current tick
        double e = f - blockEntity.progress;

        // Iterate through all entities in the sticky area that meet movement conditions
        for (Entity entity : world.getOtherEntities((Entity)null, box, entityx -> canMoveEntity(box,
entityx, pos))) {

            // Along the movement direction, push all entities that meet conditions the calculated dis-
tance
            moveEntity(direction, entity, e, direction);
```

```
        }
    }
}
```

}

`pushEntities`: Handling Piston Pushing

This method is the core of piston-entity interaction. It gets the movement direction and distance of b36 in the current game tick, and finds all entities that may contact along the path. Then, it precisely calculates the maximum overlap of each entity with b36 in the movement direction, and pushes the entity accordingly (while adding a tiny displacement of 0.01 to prevent getting stuck). This process also specially handles the bounce effect of slime blocks and the effect of b36 retraction on entities.
Its workflow is as follows:

1. Determine movement parameters: First, get the piston head's movement direction and precise distance in the current game tick.
2. Roughly filter entities: By calculating the space swept by the piston head in this frame, quickly find all entities that may be pushed.
3. Precisely calculate overlap: Iterate through each entity that may be affected, and compare its collision box with each sub-collision box of the piston head model. It precisely calculates the maximum overlap distance between the entity and piston head in the piston's movement direction.
4. Execute push: Based on the calculated maximum overlap distance, apply a displacement to the entity. To prevent the entity from getting stuck in the next frame, the push distance is increased by a tiny value (0.01).
5. Special case handling:
   - Slime blocks: If the pushed block is a slime block, it produces a bounce effect on non-player entities, setting their velocity in the movement direction to a fixed value.
   - Piston retraction: If the piston is retracting, it calls the push method to specially handle entities that may be pulled, ensuring they don't get stuck in the space after the piston arm disappears.

`push`: Solving Entity Stuck Problem

This method is a helper function specifically used to solve the problem of entities accidentally getting stuck in the block where the piston arm is located.
Its workflow is as follows:

1. Check overlap: First, check if the entity's collision box overlaps with the block collision box at the piston arm's target position. If there's no overlap, the entity is not stuck, and the method ends directly.
2. Calculate push-out distance: If overlap is detected, it calculates how far the entity needs to be pushed in the opposite direction to just separate. It performs two calculations and comparisons to ensure the push-out distance is precise and minimal, avoiding excessive displacement.
3. Execute displacement: Finally, it pushes the entity the calculated distance in the opposite direction of piston movement, smoothly solving the stuck problem.

`moveEntitiesInHoneyBlock`: Implementing Honey Block Sticky Effect

This method is specifically used to implement the "sticky" effect of moving honey blocks on entities above.
Its workflow is as follows:

1. Condition check: This function only triggers when the piston is pushing a honey block and moving horizontally.
2. Define sticky area: It defines a specific "sticky area" above the moving honey block surface (from

the honey block top to Y=1.5 height).
3. Synchronous movement: The system finds all entities located in this sticky area.
4. Apply displacement: Finally, move these entities together with the honey block in the same direction for the same distance, achieving the effect of entities being "stuck" on the honey block moving together.

:advanced Headless Piston Creation Method Based on `Extended=False` Piston Retraction Event

Let's first recall the `onSyncedBlockEvent` mentioned earlier. Note that when the server executes this method and returns false, it doesn't send a packet to the client, so the client won't execute any animation or turn anything into b36 (that is, the client won't execute this method).

The `flag` input parameter of `setBlockState` mentioned earlier is only for pathfinding updates in the usual sense. A more detailed explanation is needed here.

We observe the following phenomenon:

1. Place a piston of any direction and any type. Place a redstone block at a first-order neighbor position (except the piston's facing direction). This step will activate the piston.
2. `/tick freeze`
3. Remove the redstone block
4. Replace the original piston with a piston of the same type in any direction
5. Place back the redstone block. The position requirement is the same as in step 1.

:advanced Piston Event Response and Moving Blocks Source Code

Unified Event Entry `onSyncedBlockEvent`

```java
public boolean onSyncedBlockEvent(BlockState state, World world, BlockPos pos, int actionType, int directionData) {
    // Get the block's direction (e.g., the direction the piston faces)
    Direction facing = (Direction)state.get(FACING);

    // Get the activated block state (indicates the block is extending)
    BlockState extendedState = (BlockState)state.with(EXTENDED, true);

    // If not on the client side, perform some server-side operations
    if (!world.isClient) {
        // Check if the piston should extend (if there is a redstone signal)
        boolean shouldExtend = this.shouldExtend(world, pos, facing);

        // If should extend and event type is retract, update block state and return
        if (shouldExtend && (actionType == 1 || actionType == 2)) {
            world.setBlockState(pos, extendedState, 2);
            return false;
        }

        // If should not extend and event type is 0, directly return false
        if (!shouldExtend && actionType == 0) {
            return false;
        }
    }

    // Event type 0: piston extension event
    if (actionType == 0) {
        // If moving blocks fails, return false
```

```
        if (!this.move(world, pos, facing, true)) {
            return false;
        }

        // Set block to extended state and play extension sound
        world.setBlockState(pos, extendedState, 67);
        world.playSound((PlayerEntity)null, pos, SoundEvents.BLOCK_PISTON_EXTEND, SoundCategory.BLOCKS,
0.5F, world.random.nextFloat() * 0.25F + 0.6F);
        world.emitGameEvent(GameEvent.BLOCK_ACTIVATE, pos, Emitter.of(extendedState));
    }
    // Event type 1 or 2: piston retraction event
    else if (actionType == 1 || actionType == 2) {
        // Get the block entity in front of the piston
        BlockEntity blockEntityInFront = world.getBlockEntity(pos.offset(facing));

        // If there is a piston block entity at that position, complete the retraction operation
        if (blockEntityInFront instanceof PistonBlockEntity) {
            ((PistonBlockEntity)blockEntityInFront).finish();
        }

        // Create moving piston block state
        BlockState pistonHeadState =
Blocks.MOVING_PISTON.getDefaultState().with(PistonExtensionBlock.FACING, direction)
                .with(PistonExtensionBlock.TYPE, this.sticky ? PistonType.STICKY : PistonType.DEFAULT);

        // Set new piston block state and create new block entity
        world.setBlockState(pos, pistonHeadState, 20);
        world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(pos, pistonHeadState,
(BlockState)this.getDefaultState().with(FACING, Direction.byId(directionData & 7)), direction, false,
true));
        world.updateNeighbors(pos, pistonHeadState.getBlock());
        pistonHeadState.updateNeighbors(world, pos, 2);

        // If it's a sticky piston, execute special logic
        if (this.sticky) {
            // Calculate target position
            BlockPos targetpos = pos.add(facing.getOffsetX() * 2, facing.getOffsetY() * 2,
facing.getOffsetZ() * 2);
            BlockState targetState = world.getBlockState(targetpos);
            boolean hasMovingPistonInFront = false;

            // If there is already a moving piston block at the target position, try to complete its
extension
            if (targetState.isOf(Blocks.MOVING_PISTON)) {
                BlockEntity blockEntity2 = world.getBlockEntity(targetpos);
                if (blockEntity2 instanceof PistonBlockEntity) {
                    PistonBlockEntity pistonBlockEntity = (PistonBlockEntity)blockEntity2;
                    if (pistonBlockEntity.getFacing() == facing && pistonBlockEntity.isExtending()) {
                        pistonBlockEntity.finish();
                        hasMovingPistonInFront = true;
                    }
                }
            }

            // If no special case, remove target block or continue moving
            if (!hasMovingPistonInFront) {
                if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world, targetpos,
```

```
facing.getOpposite(), false, direction)
                        || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
!targetState.isOf(Blocks.PISTON) && !targetState.isOf(Blocks.STICKY_PISTON)) {
                    world.removeBlock(pos.offset(facing), false);
                } else {
                    this.move(world, pos, facing, false);
                }
            }
        }
        // If not a sticky piston, directly remove target block
        else {
            world.removeBlock(pos.offset(facing), false);
        }

        // Play retraction sound
        world.playSound(null, pos, SoundEvents.BLOCK_PISTON_CONTRACT, SoundCategory.BLOCKS, 0.5F,
world.random.nextFloat() * 0.15F + 0.6F);
        world.emitGameEvent(GameEvent.BLOCK_DEACTIVATE, pos, Emitter.of(pistonHeadState));
    }

    // Return true indicating event was successfully processed
    return true;
}
```

6. Place an immovable block in the piston's facing direction
7. Unfreeze with `/tick freeze`

If the operation is completely error-free, this piston becomes a headless piston.

This is because in `onSyncedBlockEvent`:

```java
if (!world.isClient) {
    // Check if the piston needs to extend (whether there's a redstone signal)
    boolean shouldExtend = this.shouldExtend(world, pos, facing);
    // If it needs to extend and the event type is retract, update block state and return
    if (shouldExtend && (actionType == 1 || actionType == 2)) {
        world.setBlockState(pos, extendedState, 2);
        return false;
    }
    // If it doesn't need to extend and the event type is 0, return false directly
    if (!shouldExtend && actionType == 0) {
        return false;
    }
}
```

Block events only verify block type and position when executing, not the block's facing direction. Therefore, at this piston's position, we get a piston with Extended=False state that can pass verification and execute, and this piston will execute a retraction event. On the server side, the piston is verified whether it should extend. At this point, the piston should indeed extend because it's activated by the redstone block, so it enters the first if branch condition. This branch condition first calls `setBlockState`, and let's see what `setBlockState` does when `flag=2`:

```java
public boolean setBlockState(BlockPos pos, BlockState state, int flags, int maxUpdateDepth) {
    if (this.isOutOfHeightLimit(pos)) {
```

```
        return false;
    } else if (!this.isClient && this.isDebugWorld()) {
        return false;
    } else {
        WorldChunk worldChunk = this.getWorldChunk(pos);
        Block block = state.getBlock();
        BlockState blockState = worldChunk.setBlockState(pos, state, (flags & 64) != 0);
        if (blockState == null) {
            return false;
        } else {
            BlockState blockState2 = this.getBlockState(pos);
            if (blockState2 == state) {
                if (blockState != blockState2) {
                    this.scheduleBlockRerenderIfNeeded(pos, blockState, blockState2);
                }
                if ((flags & 2) != 0 && (!this.isClient || (flags & 4) == 0) && (this.isClient ||
                worldChunk.getLevelType() != null &&
                worldChunk.getLevelType().isAfter(ChunkLevelType.BLOCK_TICKING))) {
                    this.updateListeners(pos, blockState, state, flags);
                }
                ...
                this.onBlockChanged(pos, blockState, blockState2);
            }
            return true;
        }
    }
}
```

We find that in the branch call `if ((flags & 2) != 0 && (!this.isClient || (flags & 4) == 0) && (this.isClient || worldChunk.getLevelType() != null && worldChunk.getLevelType().isAfter(ChunkLevelType.BLOCK_TICKING)))`, a method `updateListeners` is called. We'll find this `updateListeners` has two implementations:

1. `ServerWorld`
2. `ClientWorld`

In `ServerWorld`, this is just a normal pathfinding update. The pathfinding update here assumes the piston arm exists, but that's beside the point.

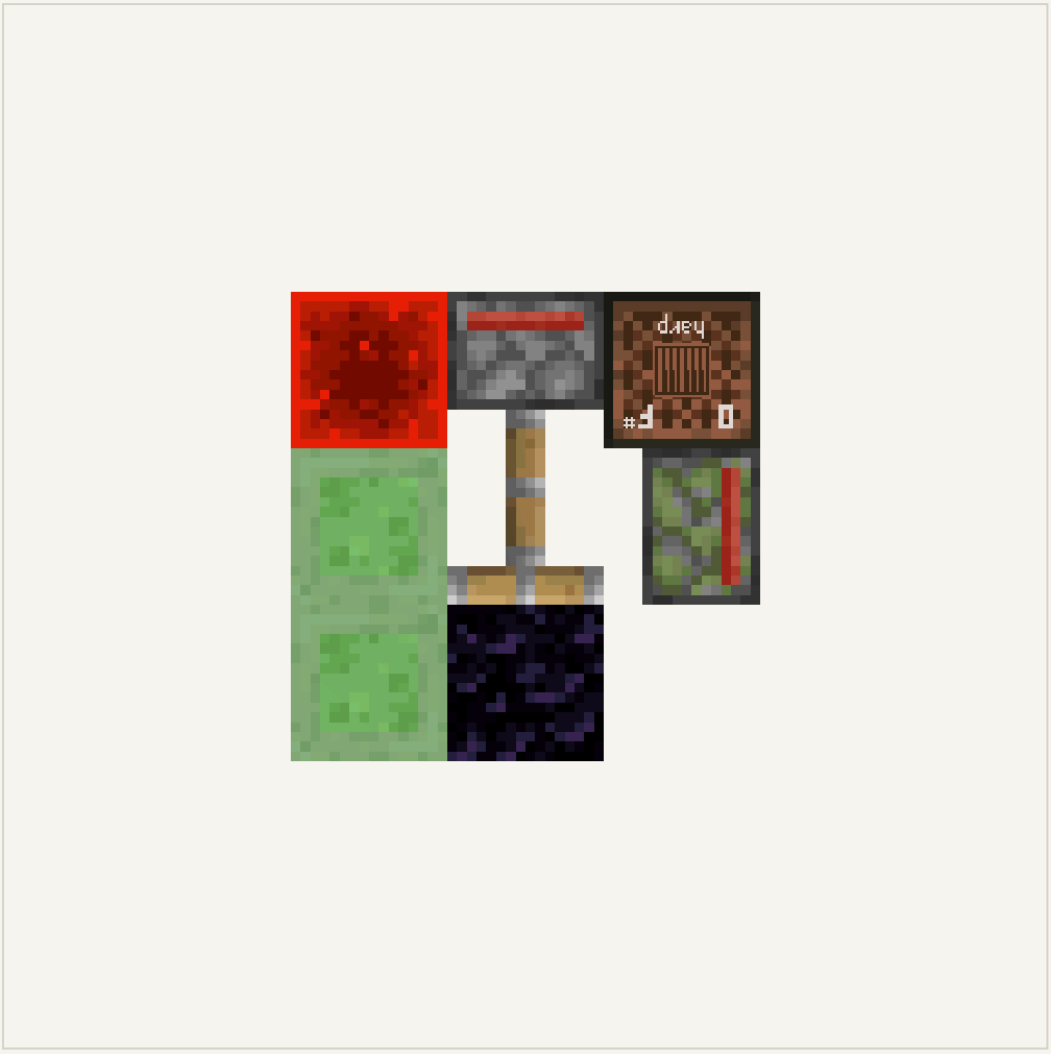
In `ClientWorld`, this `updateListeners` is responsible for rendering. Therefore, the client jumps ahead here, directly rendering the extended piston arm without rendering the piston head.

Next is the second extension event caused by piston self-check. This extension event will fail directly when executed. Whether client or server, no further explanation is needed here.

Therefore, a headless piston is created this way.

1.10 Case Analysis

1.10.1 Case 1



What happens when you hit the note block?

Piston drops

The sticky piston receives NC update and self-checks. Calls `trymove`, adds retract block event, calls `onSyncedBlockEvent`. Performs the following judgment:

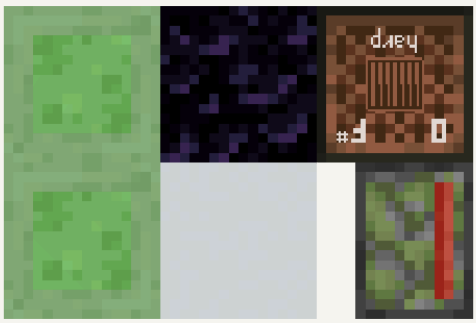
```java
if (!hasMovingPistonInFront) {
    if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world, targetpos,
    facing.getOpposite(), false, direction)
            || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
            !targetState.isOf(Blocks.PISTON) && !targetState.isOf(Blocks.STICKY_PISTON)) {
        world.removeBlock(pos.offset(facing), false);
    } else {
        this.move(world, pos, facing, false);
    }
}
```

After some judgment, enters the `move` method.

```java
**Server Side**
If not on the client, several server-specific events will be executed:
- Check activation status. If activated and the block event is retract or instant retract, set the piston to extended state and perform pathfinding update. Block event execution fails.
- If not activated and the block event is extend, block event execution fails.
**Extension Event**
> Note: This part must be executed on both client and server
The piston will attempt to move the blocks in front, this is implemented through the `move` method, which I will supplement with details later. If the move fails, the piston event execution fails.
Next, set the piston to extended state. The `bitflag` here is `0b01000011`, placement removal update, pathfinding update, NC update
Finally, play the piston extension sound. Block event execution succeeds.
**Instant Retraction and Retraction Event**
First get the piston head's block entity. If it's b36, make it instantly place.
Next, set the piston to b36. The `bitflag` here is `0b00010100`, no PP update, pathfinding update
Then set the piston to b36 block entity, first send NC update, then send PP update
If this piston is a sticky piston, then at the position 1 block outside the piston head extension, if the position 1 block outside where it tries to place the piston head is a b36, and that b36's push direction matches the piston's direction, make the block instantly place.
If the piston behavior is not instant retract, and the block is not air, the piston can move the block 1 block in front, and this block can be normally pulled back or this block is a piston or sticky piston, then attempt to pull back the block in front. If the block 1 block in front cannot be pulled, delete the block at the piston head position.
If it's a normal piston, directly delete the block at the piston head position.
In any case, play the piston retraction sound. Block event execution succeeds.
if (!retract && world.getBlockState(headPos).isOf(Blocks.PISTON_HEAD)) {
    // Set piston head to air
    // flag=0b00010100, no PP update, pathfinding update, client
    world.setBlockState(headPos, Blocks.AIR.getDefaultState(), 20);
}
```

Since the target position is a piston head and it's a retraction action, set the piston head position to air, triggering the piston head's `onRemove`. The piston head drops along with the piston base.

1.10.2 Case 2



What happens when you hit the note block?

Sticky piston returns to complete state, nothing else happens

Steps are as described in the previous example, but here after entering `move`, it doesn't meet the condition for removing the piston head. So it proceeds to the next step

```java
if (!pistonHandler.calculatePush()) {
return false;
}
```

Since the structure in front cannot be pulled back, it directly `return false`, nothing happens.

1.10.3 Case 3



What happens when you hit the note block?

Regular block is removed

Back to `onSyncedBlockEvent`, still this judgment:

```java
if (!hasMovingPistonInFront) {
    if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world, targetpos,
    facing.getOpposite(), false, direction)
            || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
            !targetState.isOf(Blocks.PISTON) && !targetState.isOf(Blocks.STICKY_PISTON)) {
        world.removeBlock(pos.offset(facing), false);
    } else {
        this.move(world, pos, facing, false);
    }
}
```

But here, the if branch is true (obsidian is immovable, `!isMovable` is `true`), so it directly removes the block in front of the piston, and the regular block is removed.

1.10.4 Case 4



What happens when you hit the note block?

Nothing happens

Same as case 2, enters the `move` method. When the piston tries to calculate if it can pull back, it checks itself, and the activated piston base itself is immovable, so `return false`, nothing happens.

1.10.5 Case 5



What happens when you pull down the lever?

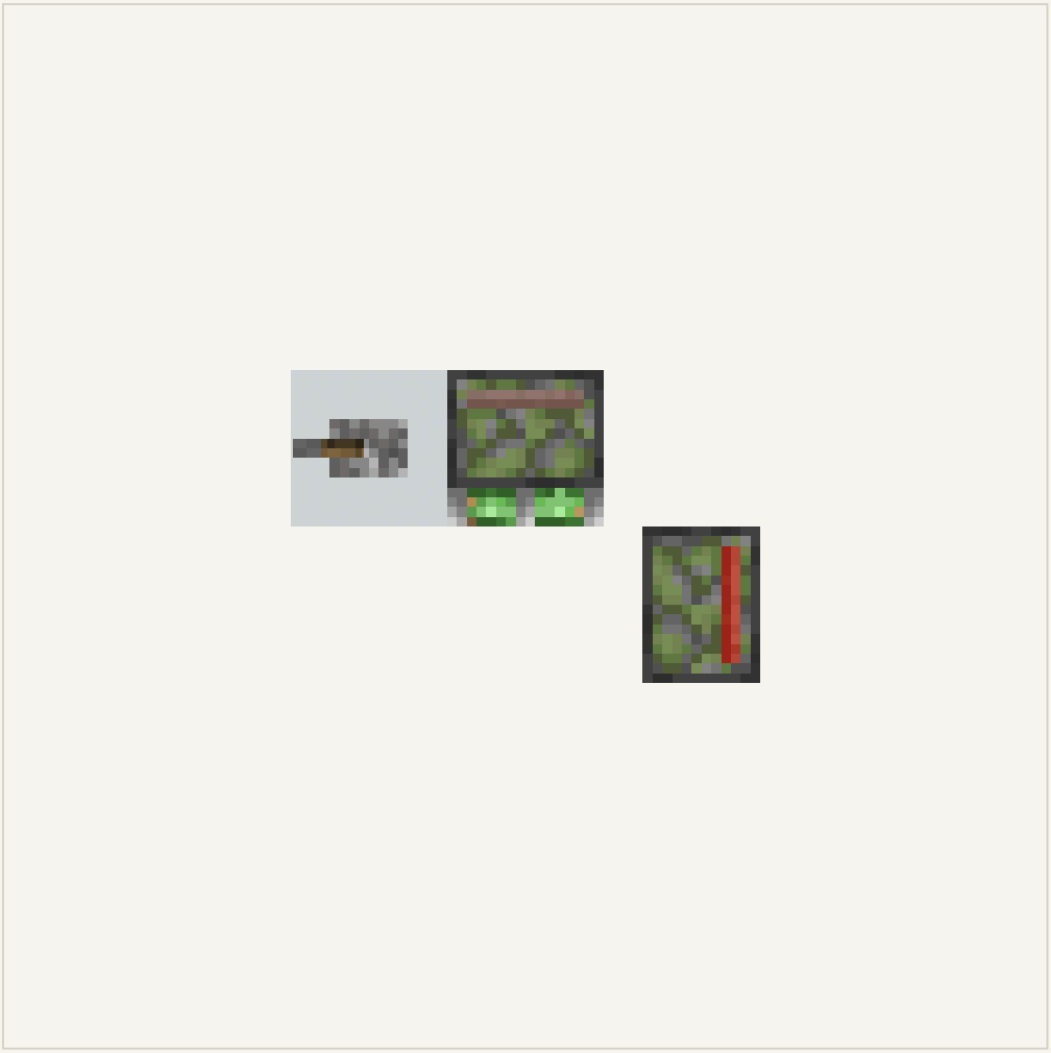
Front piston becomes headless piston, back piston returns to normal piston

Still examining `onSyncedBlockEvent`, notice this code here:

```java
BlockPos targetpos = pos.add(facing.getOffsetX() * 2, facing.getOffsetY() * 2, facing.getOffsetZ() * 2);
BlockState targetState = world.getBlockState(targetpos);
boolean hasMovingPistonInFront = false;
// If target position already has a moving piston block, try to complete its extension
if (targetState.isOf(Blocks.MOVING_PISTON)) {
    BlockEntity blockEntity2 = world.getBlockEntity(targetpos);
    if (blockEntity2 instanceof PistonBlockEntity) {
        PistonBlockEntity pistonBlockEntity = (PistonBlockEntity)blockEntity2;
        if (pistonBlockEntity.getFacing() == facing && pistonBlockEntity.isExtending()) {
            pistonBlockEntity.finish();
            hasMovingPistonInFront = true;
        }
    }
}
```

When pulling down the lever, the front piston first creates a b36 at the position in front (note the piston base hasn't become b36 yet), then executes the above operation, calling the `finish` method of the previous piston's piston arm. As mentioned before, this method has special handling for piston arm b36, directly turning it to air. Therefore, the previous piston's piston arm is removed, becoming a headless piston. The sticky piston at the back position retracts, becoming a piston with `Extended=False`.

1.10.6 Case 6



What happens when you pull down the lever?

Two pistons swap heads

The principle is similar to case 5, but the code called is here:

1.10.7 Piston Moving Blocks `move`

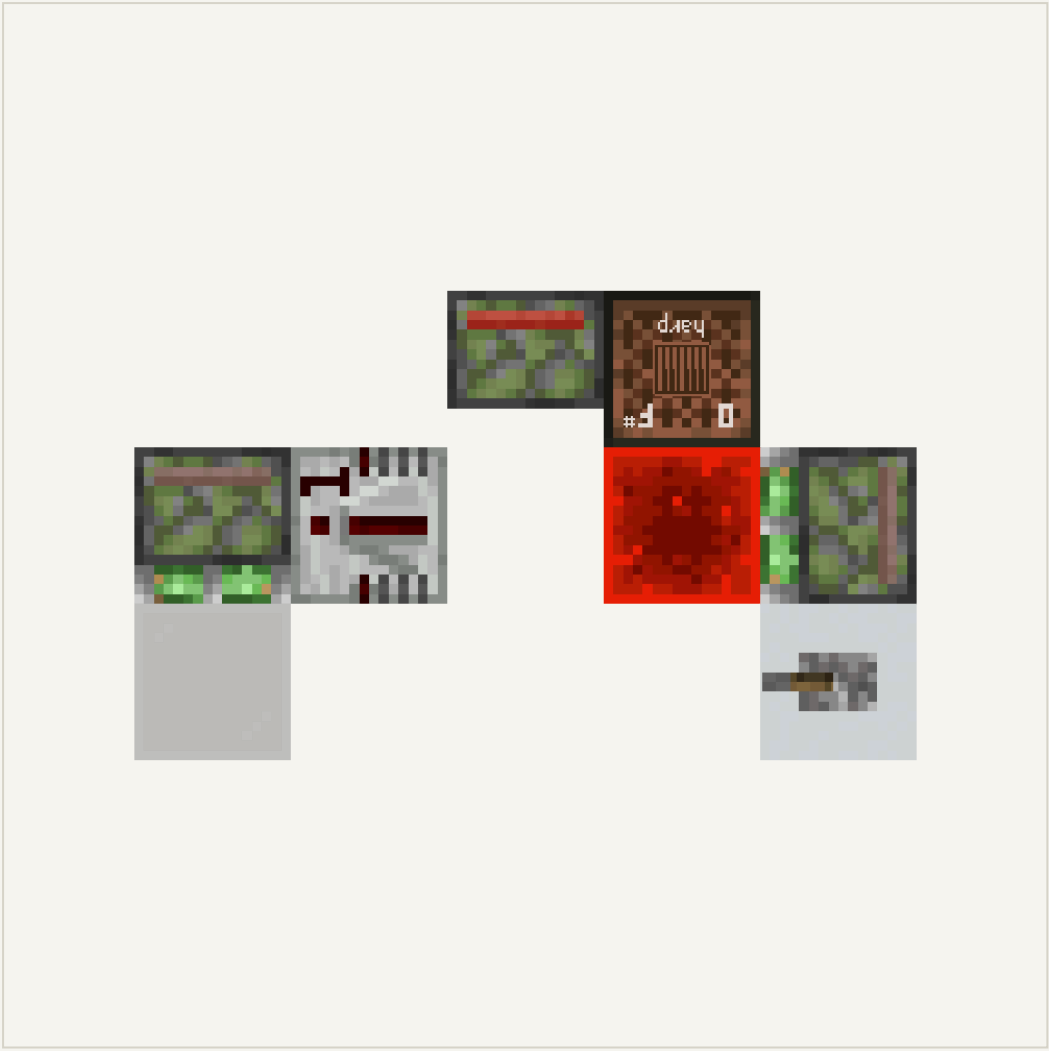
```java
// retract=0: retract
// retract=1: extend
private boolean move(World world, BlockPos pos, Direction dir, boolean retract) {
    // Piston head position
    BlockPos headPos = pos.offset(dir);
```

```
    // If it's a retraction action and the target position is a piston head
    if (!retract && world.getBlockState(headPos).isOf(Blocks.PISTON_HEAD)) {
        // Set the piston head to air
        // flag=0b00010100, no PP update, pathfinding update, client
        world.setBlockState(headPos, Blocks.AIR.getDefaultState(), 20);
    }
    // Create PistonHandler instance to calculate the legality of push or pull operations
    PistonHandler pistonHandler = new PistonHandler(world, pos, dir, retract);
    // If push/pull operation cannot be executed, cannot move
    if (!pistonHandler.calculatePush()) {
        return false;
    } else {
        // Use hash table to store blocks that need to be moved and their states for subsequent
        processing
        Map<BlockPos, BlockState> map = Maps.newHashMap();
        // Create moved blocks list and get moved blocks list
        List<BlockPos> blocksToMove = pistonHandler.getMovedBlocks();
        // Used to record the original states of blocks that need to be moved
        List<BlockState> blocksOriginal = Lists.newArrayList();
        // Traverse blocks that need to be moved, save their original states to the list and store
        in map
        for (int i = 0; i < blocksToMove.size(); ++i) {
            BlockPos blockPos2 = (BlockPos) blocksToMove.get(i);
            BlockState blockState = world.getBlockState(blockPos2);
            blocksOriginal.add(blockState);
            map.put(blockPos2, blockState);
        }
        // Get list of all block positions that need to be destroyed
        List<BlockPos> list3 = pistonHandler.getBrokenBlocks();
        // Used to record the states of destroyed blocks
        BlockState[] blockStates = new BlockState[blocksToMove.size() + list3.size()];
        // Determine movement direction
        Direction direction = retract ? dir : dir.getOpposite();
        // Process destroyed blocks, from back to front
        int j = 0;
        for (int k = list3.size() - 1; k >= 0; --k) {
            BlockPos blockPos3 = (BlockPos) list3.get(k);
            BlockState blockState2 = world.getBlockState(blockPos3);
            // If the block has a block entity, process drop logic first
            BlockEntity blockEntity = blockState2.hasBlockEntity() ?
            world.getBlockEntity(blockPos3) : null;
            dropStacks(blockState2, world, blockPos3, blockEntity);
            // Set the block to air and trigger destroy event
            world.setBlockState(blockPos3, Blocks.AIR.getDefaultState(), 18);
            world.emitGameEvent(GameEvent.BLOCK_DESTROY, blockPos3, Emitter.of(blockState2));
            // If the block is not a fire-type block, add block break particle effects
            if (!blockState2.isIn(BlockTags.FIRE)) {
                world.addBlockBreakParticles(blockPos3, blockState2);
            }
            // Save the destroyed block's state
            blockStates[j++] = blockState2;
        }
        // Process blocks that need to be moved
        for (int k = blocksToMove.size() - 1; k >= 0; --k) {
            BlockPos targetPos = (BlockPos) blocksToMove.get(k);
            BlockState targetState = world.getBlockState(targetPos);
            // Move to target position
```

```
            targetPos = targetPos.offset(direction);
            map.remove(targetPos);
            // Set target position to MOVING_PISTON state for animation processing
            BlockState blockState3 = (BlockState)
            Blocks.MOVING_PISTON.getDefaultState().with(FACING, dir);
            world.setBlockState(targetPos, blockState3, 68);
            // Create b36 block entity to control movement animation
            world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(targetPos,
            blockState3, (BlockState) blocksOriginal.get(k), dir, retract, false));
            blockStates[j++] = targetState;
        }
```java
BlockEntity blockEntityInFront = world.getBlockEntity(pos.offset(facing));
// If there's a piston block entity at that position, complete the retraction operation
if (blockEntityInFront instanceof PistonBlockEntity) {
    ((PistonBlockEntity)blockEntityInFront).finish();
}
// Create moving piston block state
BlockState pistonHeadState =
Blocks.MOVING_PISTON.getDefaultState().with(PistonExtensionBlock.FACING, direction)
        .with(PistonExtensionBlock.TYPE, this.sticky ? PistonType.STICKY : PistonType.DEFAULT);
// Set new piston block state and create new block entity
world.setBlockState(pos, pistonHeadState, 20);
world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(pos, pistonHeadState,
(BlockState)this.getDefaultState().with(FACING, Direction.byId(directionData & 7)), direction,
false, true));
world.updateNeighbors(pos, pistonHeadState.getBlock());
pistonHeadState.updateNeighbors(world, pos, 2);
```

If there's b36 in front, directly call the `finish` method to turn it to air.
```

1.10.8 Case 7



What happens when you pull down the lever?

Redstone block disappears, leftmost sticky piston drops block

Pull down the lever, the block in front of the headless piston is b36 and is updated. The redstone block arrives instantly, causing NC update, updating to the repeater, adding TT event

```java
BlockEntity blockEntityInFront = world.getBlockEntity(pos.offset(facing));
// If there's a piston block entity at that position, complete the retraction operation
if (blockEntityInFront instanceof PistonBlockEntity) {
    ((PistonBlockEntity)blockEntityInFront).finish();
}
```

But note, `onSyncedBlockEvent` hasn't returned here yet, continues executing the following operation:

```java
if (!hasMovingPistonInFront) {
    if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world, targetpos,
    facing.getOpposite(), false, direction)
            || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
            !targetState.isOf(Blocks.PISTON) && !targetState.isOf(Blocks.STICKY_PISTON)) {
        world.removeBlock(pos.offset(facing), false);
```

```
    } else {
        this.move(world, pos, facing, false);
    }
}
```

So the redstone block is removed. Emits NC update. Adds TT event.

The repeater lights up for 2gt, so the leftmost sticky piston drops the block.

2 Applications

2.1 Duplication

2.1.1 Lit Observer Removal Calculation

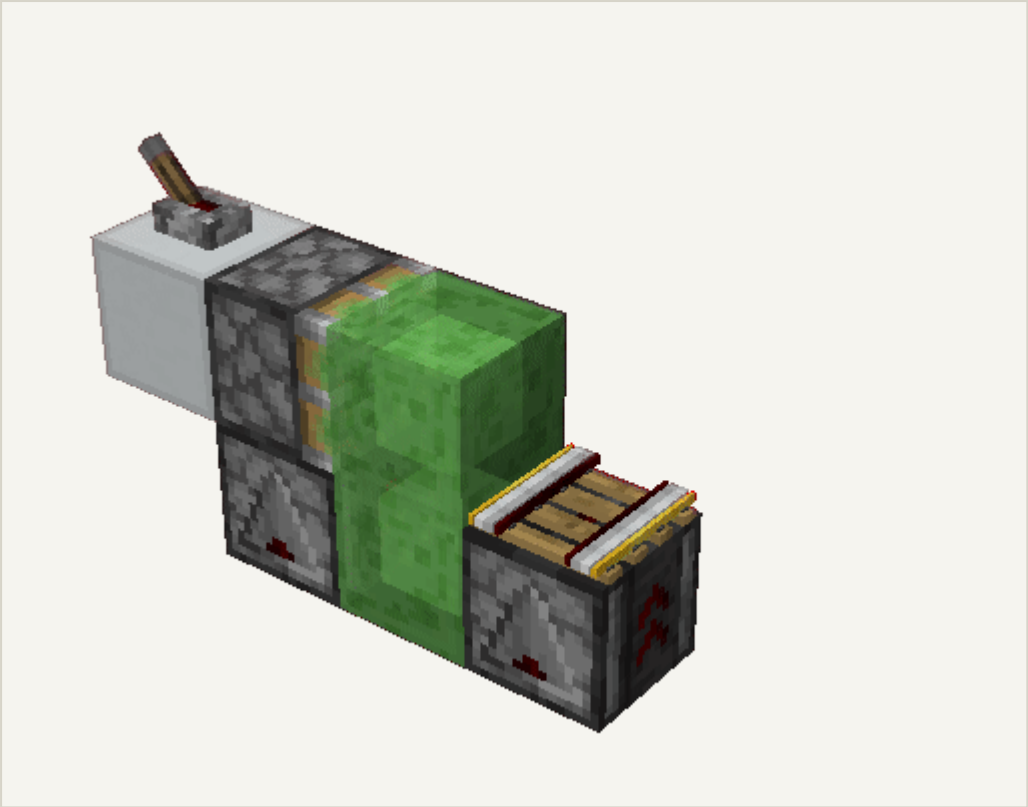
When a lit observer is removed, it emits NC update backward and extinguishes itself.

2.1.2 Observer-Based Duplication

The essence of duplication is actually that this block is added to the moved list, but at this point the block hasn't been converted to b36 yet. I insert an update in a lightning-fast manner, and this update causes something to be activated/dropped, but because this block itself has already rolled into the moved block list, when it arrives it's placed down again in the form of b36.

We divide duplication into two types: observer-based and attachment block-based. Observer-based duplication precisely utilizes the fact that as long as the observer is removed (replaced with b36), it will instantly produce an NC update.

Here's observer-based rail duplication:



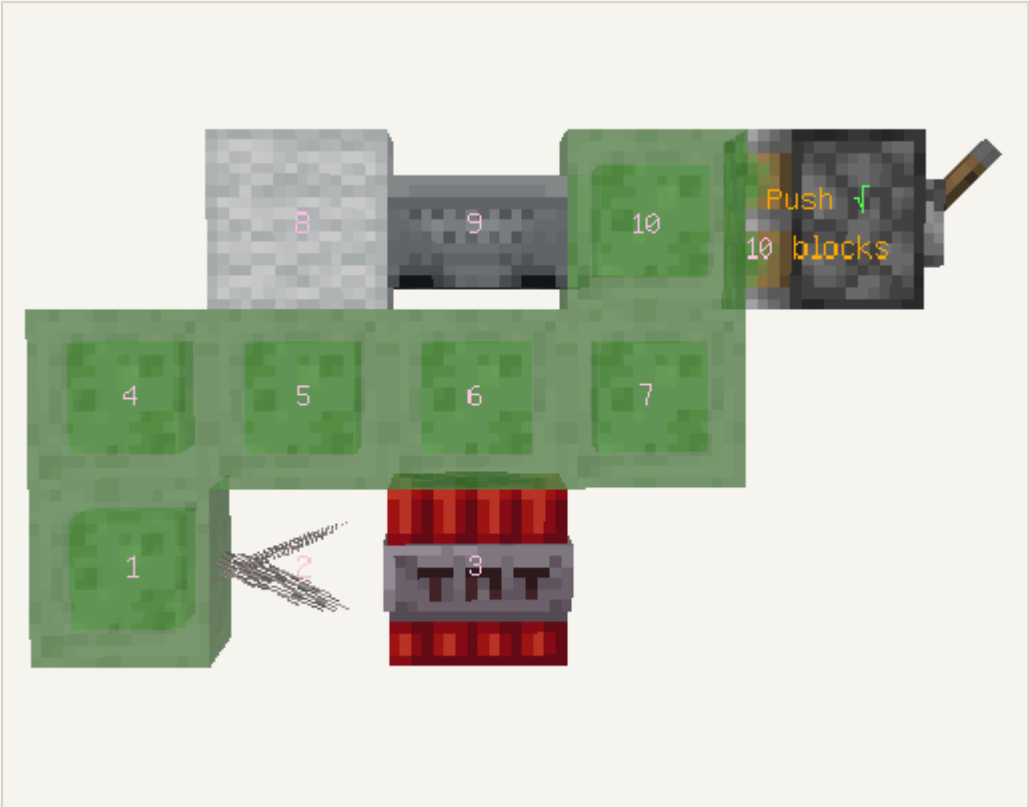
▶ This figure is animated. View original

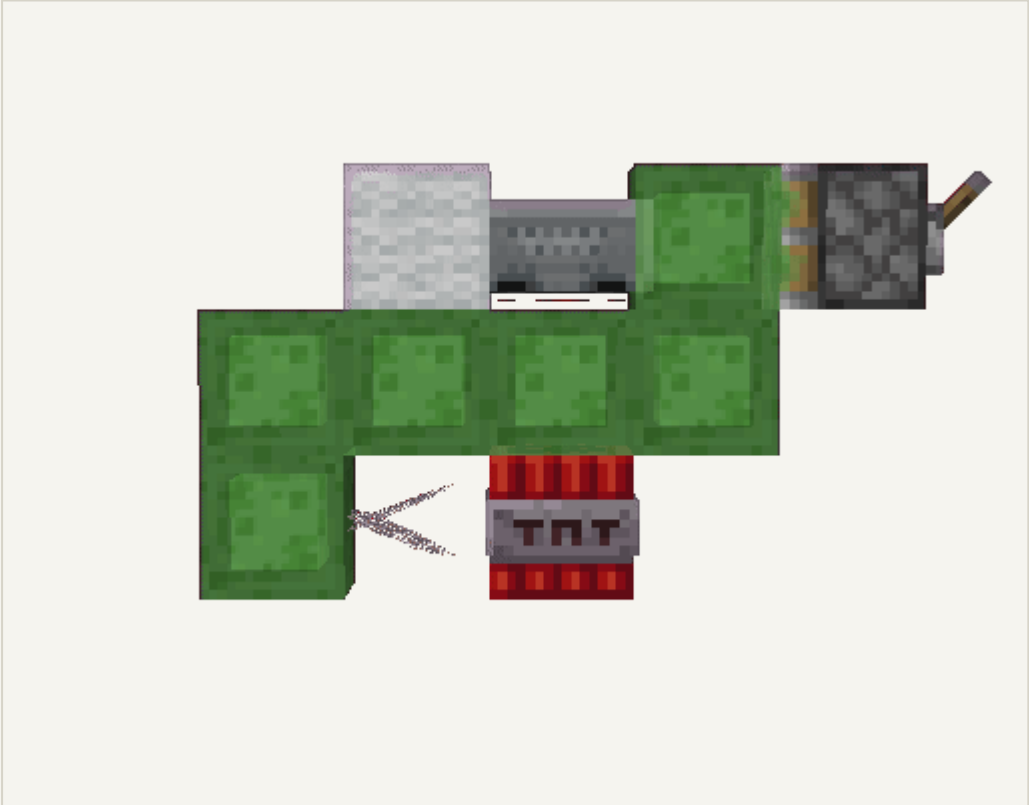
In the moved block list, when the sticky piston extends, it first moves the left observer, creates b36 at the target position, then moves the slime block, covering the lit observer, update, the rail finds that below is actually b36, drops. Move the rail, create b36 at the target position.

This way we complete one duplication

2.1.3 Attachment Block Breaking-Based Duplication

2.1.3.1 BUD-BASED TNT DUPLICATION

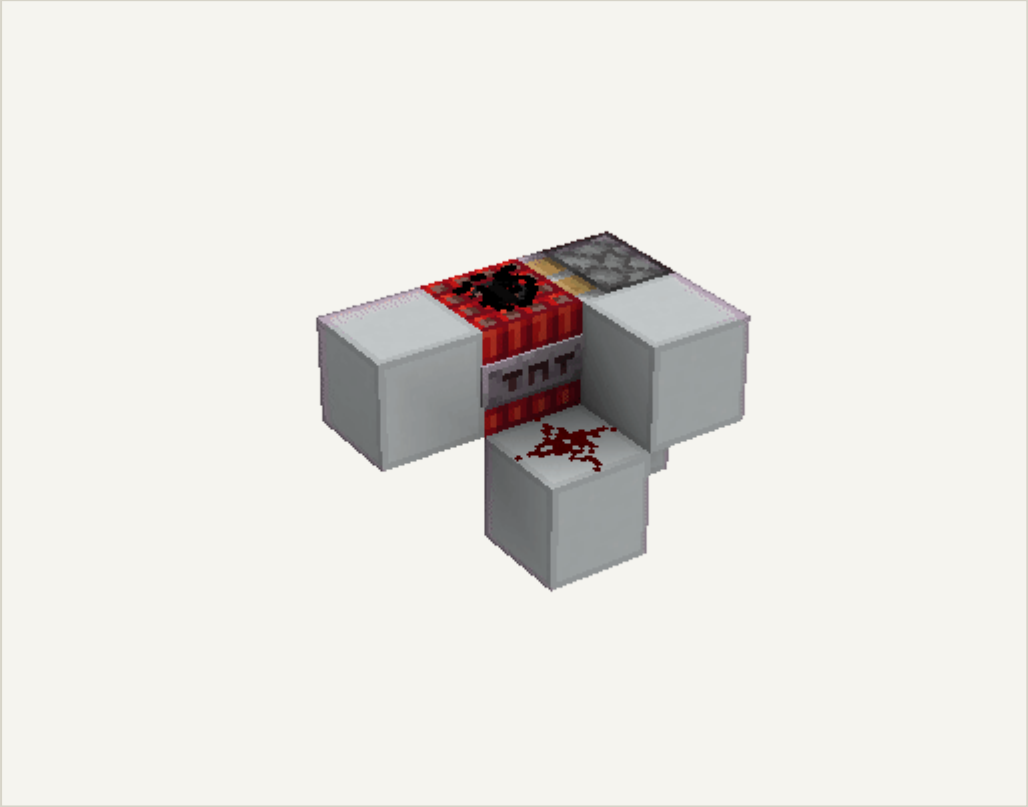




▶ This figure is animated. View original

Attachment block dropping also produces updates, but since attachment blocks usually drop directly and are difficult to reuse. For example:

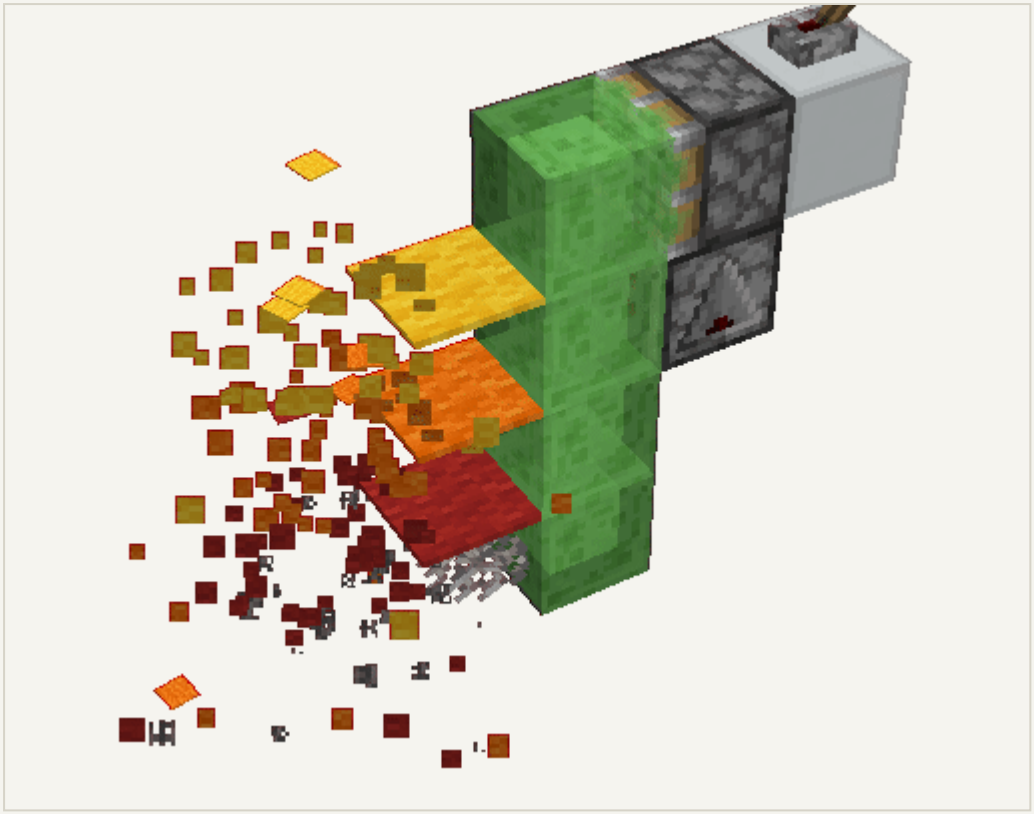
2.1.3.2 DUST-DIRECTED MINIMALIST TNT DUPLICATION



▶ This figure is animated. View original

2.1.3.3 CARPET DUPLICATION

The figure below shows a very popular carpet duplication:



▶ This figure is animated. View original

When the piston retracts, it completes one duplication. We'll analyze it briefly here:

```
    // If it's an extend operation, add piston head
    if (retract) {
        PistonType pistonType = this.sticky ? PistonType.STICKY : PistonType.DEFAULT;
        BlockState blockState4 = (BlockState) ((BlockState)
Blocks.PISTON_HEAD.getDefaultState().with(PistonHeadBlock.FACING, dir)).with(PistonHeadBlock.TYPE, pis-
tonType);
        BlockState targetState = (BlockState) ((BlockState)
Blocks.MOVING_PISTON.getDefaultState().with(PistonExtensionBlock.FACING,
dir)).with(PistonExtensionBlock.TYPE, this.sticky ? PistonType.STICKY : PistonType.DEFAULT);
        map.remove(headPos);

        // Set current piston position to MOVING_PISTON state
        world.setBlockState(headPos, targetState, 68);

        // Add b36 block entity to control animation
        world.addBlockEntity(PistonExtensionBlock.createBlockEntityPiston(headPos, targetState,
blockState4, dir, true, true));
    }

    // Set remaining blocks in map to air
    BlockState blockState5 = Blocks.AIR.getDefaultState();
    for (BlockPos blockPos4 : map.keySet()) {
        world.setBlockState(blockPos4, blockState5, 82);
    }

    // Update neighbor block states
    for (Map.Entry<BlockPos, BlockState> entry : map.entrySet()) {
        BlockPos blockPos5 = entry.getKey();
```

```
        BlockState blockState6 = entry.getValue();
        blockState2.prepare(world, blockPos5, 2);
        blockState5.updateNeighbors(world, blockPos5, 2);
        blockState5.prepare(world, blockPos5, 2);
    }

    // Update destroyed blocks' neighbor states
    for (int l = list3.size() - 1; l >= 0; --l) {
        BlockState targetState = blockStates[j++];
        BlockPos blockPos5 = (BlockPos) list3.get(l);
        targetState.prepare(world, blockPos5, 2);
        world.updateNeighborsAlways(blockPos5, targetState.getBlock());
    }

    // Update moved blocks' neighbor states
    for (int l = blocksToMove.size() - 1; l >= 0; --l) {
        world.updateNeighborsAlways((BlockPos) blocksToMove.get(l), blockStates[j++].getBlock());
    }

    // If it's a retraction action, update piston position neighbors
    if (retract) {
        world.updateNeighborsAlways(headPos, Blocks.PISTON_HEAD);
    }

    return true; // If execution succeeds, return true
}
```

}

When a piston attempts to move blocks, it distinguishes between two situations: extend and retract. The `retract` boolean value here represents the piston's action. When its value is `false`, the piston retracts; when its value is `true`, the piston extends. Next, the piston will distinguish between the two piston actions and process them separately.

When the piston **attempts** to retract, the game will first check whether the position that **should be** the piston head is actually a piston head. Only if it is a piston head will retraction be executed. Each time retraction is executed, the game will first set the piston head position to air (delete the piston head). The `setBlockState` `bitflag` passed in at this time is `0b00010100`, i.e., no PP update and client pathfinding update.

Before formally starting to move, whether retracting or extending, the game will call the `calculatePush` method analyzed in section [5.4.2](#进阶542-移动结构分析) to analyze again whether it can move. If it cannot successfully move, the piston fails to execute push/pull.

Next, the game will formally start moving blocks. It will first process the broken blocks list (`brokenBlocks`) in reverse order. If the destroyed block has a block entity that drops items, drop the items, then set the anchored destroyed block (i.e., the block currently being processed) to air. At this time, no PP update, send pathfinding update.

After destroying the blocks that should be destroyed, the game will start moving blocks. Traverse the moved blocks list (`movedBlocks`) from back to front, set that block to b36, `bitflag` is `0b01101000`, i.e., placement removal update, then add b36 block entity to that block and add to the `blockStates` list for backup.

If extending, create b36 at the piston head position, placement removal update, then add b36 block entity to the piston head position.

After completing the above operations, set all uncovered blocks to air with placement removal update and pathfinding update. No PP update at this time. Then uniformly send updates. For all blocks in the hash table, first uniformly send redstone dust prepare update, then send block update (at the block's original position). If extending, the piston head completes block update.

The above code well explains why b36 addition is in reverse order with the `movedBlocks` list. Also, because the hash table is unordered, this also explains why some machines that depend on b36 placement order break after unloading. (After unloading, the original index is lost)

Moved block list:

1. Move the bottom slime block, create b36 at target position
2. Move the dead coral fan, create b36 at target position, emit PP update, self-check finds it should drop, emit PP update and NC update, guide the carpet above to drop in chain
3. Move remaining slime blocks and carpet

This completes the duplication.

Block Events with Inconsistent Information

According to sections 5.1, 2.1.5, etc., we can find: when the piston executes a block event, it only verifies its `block position` and `block type`. Although `piston facing` is recorded, it only governs the direction of b36.

The `actual direction` when the piston executes the block event is sampled in real-time.

This means that before a piston's block event instance is executed in the block event phase, there exists a time window: at this point, we can freely change the properties of the piston at the corresponding position, or even destroy or replace this piston itself.



As shown in the figure, from the EU of the previous game tick to the BE of the current game tick, during this time we can modify the target object of a certain block event.
Similarly, from the EU of the current game tick to the BE of the next game tick, such a time window also exists.
But as long as when executing this piston's block event instance, if there's a piston at the execution position and the piston type hasn't changed, then the direction the piston attempts to push/pull will be determined by the facing at execution time, even though this facing may not be the facing when the block event was added.

> Interestingly, the animation when b36 retracts still follows the data parameter passed in the block event,
> so, even though the piston facing at block event execution differs from when it was added, the piston head animation will still face the piston facing when the block event was added (but the collision box is accurate).
> ![piston.gif](img/piston.gif)

Breaking Bedrock

Cross-Game-Tick Bedrock Breaking

> The Piston's Revenge Journey

As mentioned before, we can sneakily modify the piston's properties, including facing, before the piston's block event phase.
This means we can change its facing to face bedrock when the piston extends, and then when the piston retracts, it will destroy the bedrock as if it were a piston head.
The demonstration animation below is a typical example of breaking bedrock by changing piston facing before the block event phase.

During the entire process, the following important events occurred

CECECE{gt0}-EU phase

- Left TNT explodes.
  - Explosion destroys lever.
  - Explosion destroys cobblestone.

- Right TNT explodes.
  - Explosion destroys piston head, emits NC update.
  - Piston base is updated. Since the lever has been blown up, the piston adds a **retract** block event to itself.
    > Unfortunately, it has already missed the BE phase of this game tick, so it can only wait silently for the BE phase of gt1...
  - Piston head destroys piston base.

CECECE{gt0}-AT phase

- Player places downward-facing new piston.

CECECE{gt0}-BE phase

- Piston retracts according to current facing, destroys bedrock.

The above is the entire process of breaking bedrock.

However, in structures like bedrock breakers, it's impossible for players to place pistons. We must achieve this effect through other means.

Bedrock Breaking Completed in Block Event Phase

The "before" in "before the piston's block event phase" doesn't only apply to other game phases. Even within the block event phase, you can still change the block corresponding to a certain block event before that event executes.

As you can see, the figure below is a simplified version of the bedrock breaking structure in some bedrock breakers...

As shown in the animation, during the entire process, the following important events occurred

> Premise--The numbers marked on the surface in the figure above are the depths of each piston, they attempt to extend or retract in this order in the BE phase.

CECECE{gt0}-BE phase

:advanced Placement Source Code Analysis

Normal Placement `tick`

> This is too simple to explain much, so the plain explanation was moved to the front (laugh). Therefore, only code is provided here for reference

```java
public static void tick(World world, BlockPos pos, BlockState state, PistonBlockEntity blockEntity) {
        //Record current gt
    blockEntity.savedWorldTime = world.getTime();
    blockEntity.lastProgress = blockEntity.progress;
    //If push is complete
    if (blockEntity.lastProgress >= 1.0F) {
        //If it's the client and current deathTick hasn't exceeded 5
        if (world.isClient && deathTicks < 5) {
            //Increase deathTicks count
            ++deathTicks;
        } else {
            //Remove block entity
            world.removeBlockEntity(pos);
            blockEntity.markRemoved();
            //If this block is b36
            if (world.getBlockState(pos).isOf(Blocks.MOVING_PISTON)) {
                //Send PP update
                BlockState blockState = Block.postProcessState(blockEntity.pushedBlock, world, pos);
                //If it's air
                if (blockState.isAir()) {
                    //Set to corresponding block, no update, bitflag=0b01010100;
                    world.setBlockState(pos, blockEntity.pushedBlock, 84);
                    //Update or destroy this block
                    Block.replace(blockEntity.pushedBlock, blockState, world, pos, 3);
                } else {
                    //If it's a waterlogged block
                    if (blockState.contains(Properties.WATERLOGGED) &&
(Boolean)blockState.get(Properties.WATERLOGGED)) {
                        //Remove water from waterlogged block
                        blockState = (BlockState)blockState.with(Properties.WATERLOGGED, false);
                    }
                    //Set to corresponding block, send NC update, PP update, client update,
bitflag=0b01000011
                    world.setBlockState(pos, blockState, 67);
                    //Receive NC update itself
                    world.updateNeighbor(pos, blockState.getBlock(), pos);
                }
            }
        }
    }
    //If push is not complete
    //Push progress +0.5
    else {
        float f = blockEntity.progress + 0.5F;
        //Perform entity calculations
        pushEntities(world, pos, f, blockEntity);
        moveEntitiesInHoneyBlock(world, pos, f, blockEntity);
```

```
        blockEntity.progress = f;
        //If push progress >=1, set to 1
        if (blockEntity.progress >= 1.0F) {
            blockEntity.progress = 1.0F;
        }
    }
}
```

- Piston #1 retracts, destroys piston head, emits NC update
  - The piston base in front of piston #1 is updated. Since it's in BUD state, the piston adds a **retract** block event to itself.

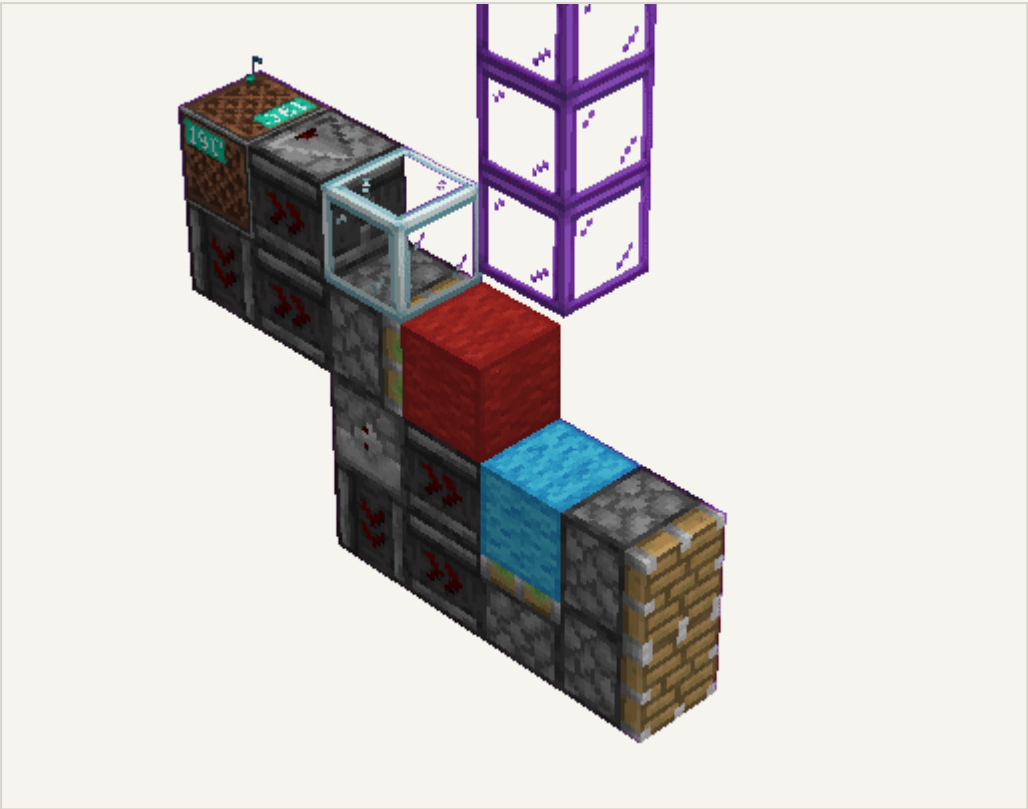
    > Since depths 1, 2, 3 are occupied, it will be scheduled to #4, which is "execute last" in the current situation.

  - Piston head destroys piston base.
- Piston #2 extends, creates 3 b36s.
- Sticky piston #3 retracts, forcing the b36 of the downward-facing piston directly in front to arrive instantly.
- The downward-facing piston's current position has the block event caused by piston #1 earlier, retracts according to current facing, destroys bedrock.

2.2 Deceiving Piston to Add/Cancel Adding Block Events

2.2.1 Deceiving Piston to Cancel Adding Block Event: Push Limit Detection

Block events can also have situations where the piston thinks the event it wants to execute is inconsistent with the actual situation. The most typical is push limit detection.



▶ This figure is animated. View original

The push limit detection mechanism was originally discovered by players _Kayleigh and Landmining. Its core lies in "deceiving" the piston. What's shown here is a simplified version made by Bright_Observer, convenient for analysis and understanding.

Although this device has directional design, this doesn't affect its detection function, so the construction direction can be freely chosen.

Operating phenomenon:

- After placing purple wool (the wool is connected to a very long pillar...) and hitting the note block, the regular piston will activate
- After breaking the purple wool and hitting, the regular piston remains unchanged

Principle analysis:

1. The upward-pushing piston self-checks when updated
2. The system checks push limit conditions
3. If wool is placed, because the glass pillar above has exceeded the push limit, the piston won't add a block event
4. At this point, activate two regular pistons
5. The sticky piston is updated by the regular piston, self-checks and extends
6. When wool doesn't exist:
   Sticky piston operates normally
   The extension state of the upper regular piston depends on facing
   The lower regular piston remains in place due to depth relationship

Key finding:
The sticky piston's planned action may differ from the push structure at actual execution time. This characteristic is cleverly used for the detection mechanism.

2.2.2 Deceiving Piston to Add Block Event

Unfortunately, if you try to replicate the layout and activation sequence of ***5.10.1 Breaking Bedrock***

2.2.3 Instant Placement `finish`

```java
public void finish() {
    // World exists and progress not full or client calculation
    if (this.world != null && (this.lastProgress < 1.0F || this.world.isClient)) {
        // Set progress to complete
        this.progress = 1.0F;
        this.lastProgress = this.progress;
        // Remove block entity
        this.world.removeBlockEntity(this.pos);
        this.markRemoved();
        //If this is a b36 block
        if (this.world.getBlockState(this.pos).isOf(Blocks.MOVING_PISTON)) {
            // If it's a piston arm
            BlockState blockState;
            if (this.source) {
                // Set to air
                blockState = Blocks.AIR.getDefaultState();
            } else {
                // Otherwise place after PP update
                blockState = Block.postProcessState(this.pushedBlock, this.world, this.pos);
            }
            // Set block, PP update, pathfinding update, NC update
            this.world.setBlockState(this.pos, blockState, 3);
            // Receive block update itself
            this.world.updateNeighbor(this.pos, blockState.getBlock(), this.pos);
        }
    }
}
```

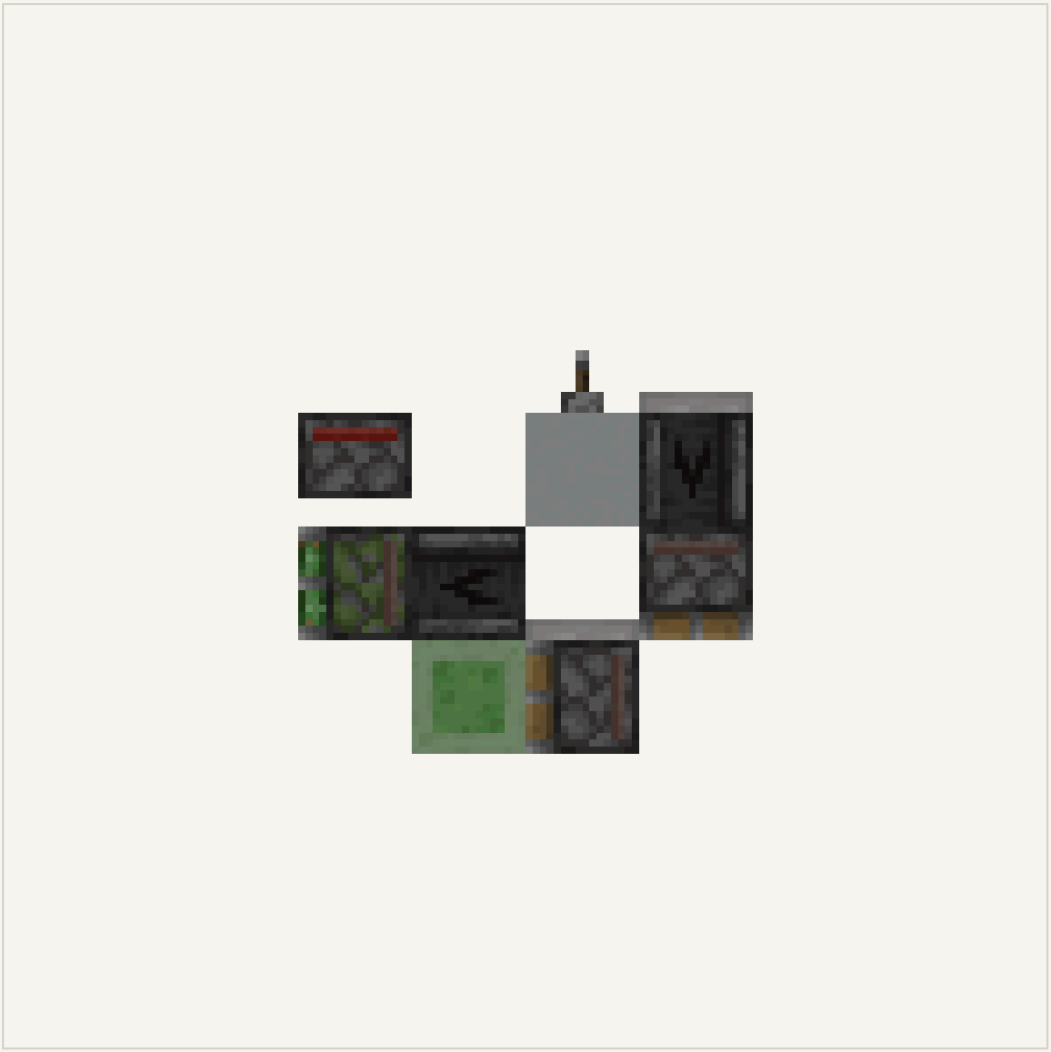
Most of the content was explained before, so it's skipped here. We only look at the latter part of the b36 special case.

In 1.20, only the piston arm's `source` property is `true`, all others are `false`. Therefore, when this method is called, if the position is b36 with target block as piston arm, it will directly become air. The rest won't be detailed, as it's all been covered before.

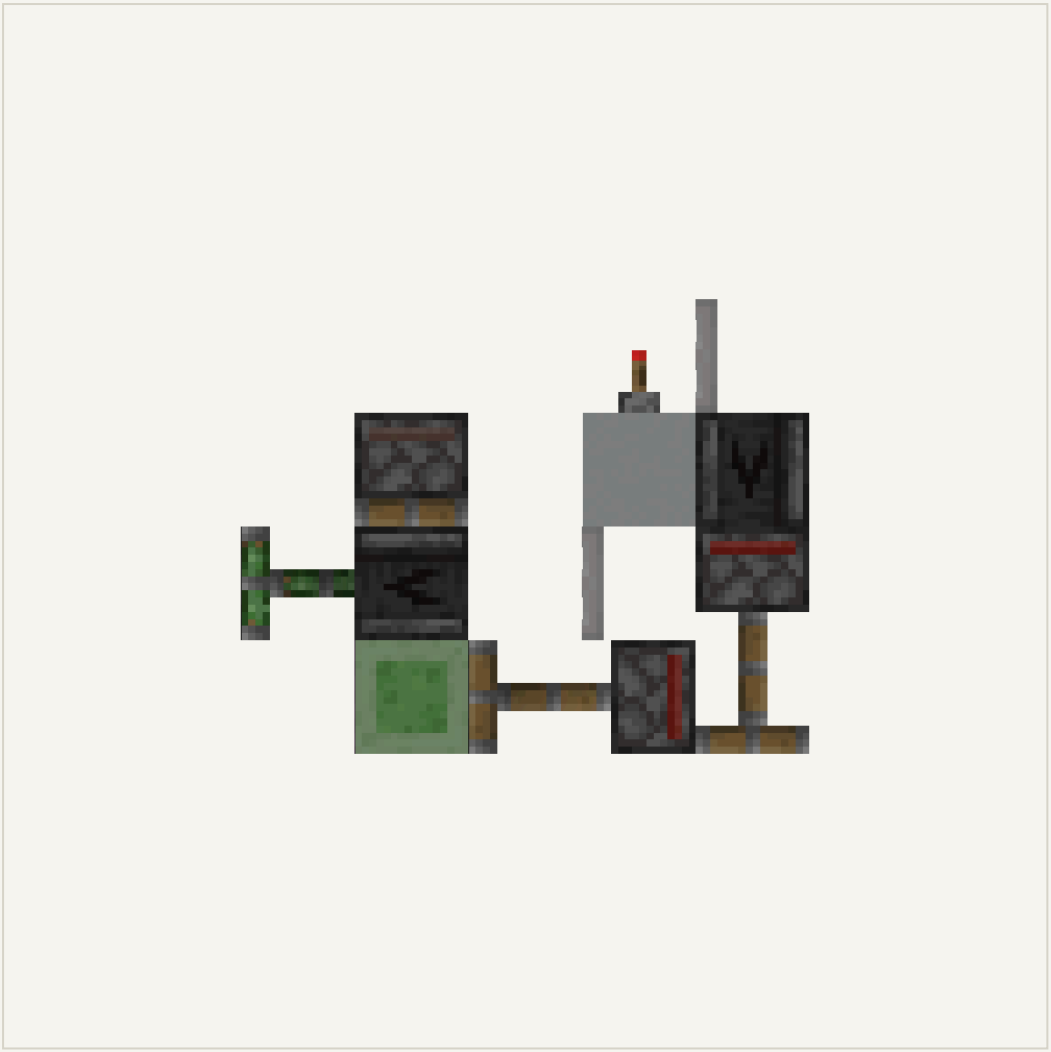
2.3 Piston Head

> This chapter actually has no difficulty, but since apart from Menggui233's recent demonstration in PetrisAFE's video, no other uses have been seen, so it's placed in the advanced section here.

We first observe the following device:



Now, pull down the lever



We observe that the piston base was replaced with an observer, while the piston head was preserved. Why?

We first review what happens when b36 arrives:

- Send PP update at its own position
- Send NC and PP updates to first-order adjacency at its own position
- Send NC update at its own position

Therefore, we separately check what happens when the piston head receives PP update and NC update

2.3.1 Piston Head Existence Condition Determination `canSurvive`

```java
private boolean isFittingBase(BlockState baseState, BlockState extendedState) {
    Block block = baseState.getValue(TYPE) == PistonType.DEFAULT ? Blocks.PISTON :
    Blocks.STICKY_PISTON;
    return extendedState.is(block) && extendedState.getValue(PistonBaseBlock.EXTENDED) &&
    extendedState.getValue(FACING) == baseState.getValue(FACING);
}
@Override
public boolean canSurvive(BlockState state, LevelReader level, BlockPos pos) {
```

```
    BlockState blockState = level.getBlockState(pos.relative(state.getValue(FACING).getOppo-
    site()));
    return this.isFittingBase(state, blockState) || blockState.is(Blocks.MOVING_PISTON) &&
    blockState.getValue(FACING) == state.getValue(FACING);
}
```

The `isFittingBase` method essentially states that a piston head is `fitting` if and only if the block behind the piston head (more precisely, 1 block in the opposite direction of the push direction, hereafter referred to as behind) is an extending piston with the same extension direction. The `canSurvive` method requires that a piston head's existence is valid if and only if the piston head is `fitting`, or there is a b36 behind it with the same push direction.

2.3.2 Piston Head Removal `onRemove`

JAVA

```
@Override
public void onRemove(BlockState state, Level level, BlockPos pos, BlockState newState, boolean is-
Moving) {
    if (!state.is(newState.getBlock())) {
        super.onRemove(state, level, pos, newState, isMoving);
        BlockPos blockPos = pos.relative(state.getValue(FACING).getOpposite());
        if (this.isFittingBase(state, level.getBlockState(blockPos))) {
            level.destroyBlock(blockPos, true);
        }
    }
}
```

One sentence summary: When removed, if the state is `fitting`, the base is removed along with it and the piston drops.

2.3.3 Piston Head Receives PP Update, NC Update

JAVA

```
@Override
public BlockState updateShape(BlockState state, Direction facing, BlockState facingState,
LevelAccessor level, BlockPos currentPos, BlockPos facingPos) {
    return facing.getOpposite() == state.getValue(FACING) && !state.canSurvive(level, currentPos)
        ? Blocks.AIR.defaultBlockState()
        : super.updateShape(state, facing, facingState, level, currentPos, facingPos);
}
@Override
public void neighborChanged(BlockState state, Level level, BlockPos pos, Block block, BlockPos
fromPos, boolean isMoving) {
    if (state.canSurvive(level, pos)) {
        level.neighborChanged(pos.relative(state.getValue(FACING).getOpposite()), block, fromPos);
    }
}
```

PP Update

The piston head actually only cares about PP updates from behind. When a PP update occurs behind, the piston head will call `canSurvive` to check if it is valid. If invalid, it immediately disappears.

NC Update

In any case, the piston head will not disappear due to receiving an NC update. When receiving an NC update, if it is valid, the update is treated as an NC update received by the block behind.

2.3.4 Case Analysis

Therefore, for the case mentioned at the beginning of the chapter:

- 0gt AT: Player pulls down lever, trapdoor triggers
- 2gt BE depth 0: Sticky piston is activated, updates headless piston
- 2gt BE depth 1: Headless piston eats sticky piston base, slime block & lit observer are pushed
- 4gt TE: Sticky piston head places first. At this point, it checks that behind is b36 in the same direction as itself, self-check passes. Then lit observer b36 places, no update.

Therefore, the observer successfully replaced the original piston base.

2.4 B36 Related Entity Calculations

Footnotes

1. The `tryMove` here needs disambiguation. In `Java`, methods in different classes can be given the same name. The `tryMove` here is in `PistonHandler.java` ↩

CHAPTER 4

4 Blocks

IN THIS CHAPTER

4.1 Blocks and Block States

4.2 Placing, Changing, and Breaking Blocks

The Nature of Blocks

1 Overview

Blocks are the basic building units of the Minecraft world. But the word "block" hides an important distinction: in the source code, `Block` and `BlockState` are two completely different concepts.

This chapter starts from that distinction, then introduces the design of block states, the full process of placement and breaking, and how they connect to chunk mechanics and the update system.

2 Structure

1. **Blocks and Block States** — the essential difference between `Block` and `BlockState`, the hard limits on properties and values, the subchunk palette storage model, the boundary between NBT format and block entities, and how fluid state is derived
2. **Placing, Changing, and Breaking Blocks** — the shared `setBlockState` entry point, the exact steps of the placement flow, the two breaking entry points, the meaning of each flag, and the chain of events triggered after a write

4.1 Blocks and Block States

Consider the following pairs of blocks:

- a stair facing east and a stair facing west;
- an extended piston and an unextended piston;
- redstone dust with power level 0 and redstone dust with power level 15;
- a waterlogged stair and a normal stair.

Sometimes they look completely different, yet they still belong to the same kind of block. How does Minecraft describe that difference?

The answer is: **block** and **block state** are not the same concept.

1 A block is a rule set; a state is its current values

In the source code, the essence of `Block` is **"the shared set of rules for one kind of block"**. For example, all oak stairs share the same `StairsBlock` instance, which defines:

- which properties can exist (facing, top/bottom half, shape, whether it is waterlogged);
- how placement orientation is computed;
- what collision shape it has;
- how its shape is adjusted when it receives an NC update;
- which loot table it uses when broken.

However, merely knowing that "this is an oak stair" is still not enough to reconstruct it in the world. The game also needs to know which way it faces, whether it occupies the top or bottom half, whether its shape is straight or a corner, and whether it is waterlogged.

Those current values together form a **block state**. Therefore, **the essential relationship between `Block` and `BlockState` is this: `Block` defines "what this thing can become, and how the rules change after it does"; `BlockState` records "which values it currently has."**

2 Properties and values

A block state is made up of a set of **properties** (`Property`) and their values. Every property has three hard constraints:

1. **Each property provides only a finite set of allowed values** — `StairsBlock`'s `half` can only be `top` or `bottom`; it cannot temporarily become something else.
2. **All combinations of all properties over all allowed values are created in advance** — when a `Block` is constructed, `StateManager` enumerates every combination and creates an immutable `BlockState` object for each one.

3. **Created states cannot be modified** — calling `state.with(property, value)` does not mutate the original state in place; instead, it returns another `BlockState` from a prebuilt lookup table.

Take stairs as an example. `StairsBlock` declares four properties:

PROPERTY	MEANING
`facing`	which way the stair faces
`half`	whether it is in the top or bottom half
`shape`	straight, inner corner, outer corner, etc.
`waterlogged`	whether it contains water

Total number of combinations: $4 \times 2 \times 5 \times 2 = \mathbf{80}$. All 80 of those `BlockState` objects are created when the block is registered, and any one of them can then be found with an O(1) table lookup.

Key rule: you cannot temporarily add a new property to a block. A piston's `extended` can only be `true` or `false`; redstone dust's `power` can only be 0 through 15. Any write outside the allowed value set throws an exception inside `State.with()`.

3 One set of coordinates corresponds to one state

When the game queries a set of coordinates in the world, it returns a `BlockState`, not just a `Block`.

The subchunk's `PalettedContainer<BlockState>` stores block states directly. That way, reading one set of coordinates immediately tells you:

- what block it is (via `BlockState.getBlock()`);
- what values all of its state properties currently hold (via `BlockState.get(Property)`).

This leads to an important property: **large numbers of blocks in the world with the same state can all share the same pre-generated `BlockState` object**. Each set of coordinates does not need its own tiny mutable object; they share immutable state instances.

Internally, `BlockState` holds its owning `Block` and an immutable property map. Calling:

```java
state.with(property, value)
```

does not modify the original state. Instead, it retrieves another state from the prebuilt state table. If the specified value is not in the allowed set for that property, the source code throws an exception.

That means many blocks in the world with the same state can share the same pre-generated state object, instead of every coordinate owning its own freely mutable little object.

4 Subchunks and palettes: how states are stored

A world contains millions of coordinate positions. If every position stored its own complete `BlockState` object, memory usage and storage size would explode quickly. Minecraft solves this with **subchunks** and **palettes**.

4.1 Subchunks

A 16×16×16 region is called a **subchunk** (`ChunkSection`). A normal chunk can contain up to 24 subchunks (Y = -64 to Y = 319, one layer every 16 blocks). Each subchunk manages 4096 coordinate positions, but **it does not store one separate `BlockState` object per position**.

4.2 Palette: using IDs instead of objects

Each subchunk maintains a **palette**, recording "which block states appear in this subchunk":

- The palette is a list, and each entry is a unique `BlockState`.
- Each of the 4096 positions in the subchunk stores a small numeric ID pointing to one palette entry.
- When a position's block state needs to be read, the ID is used to look up the corresponding `BlockState`.
- When a new state is written, if that state is not already in the palette, it is added, and the position stores the new entry's ID.

In the source code, this process is handled by `PalettedContainer<BlockState>`. It is called "Paletted" precisely because its core is a palette (similar to indexed colors in images) plus an index array. The block data inside a subchunk is really "a table plus 4096 IDs," not "4096 independent `BlockState` objects."

4.3 Why this matters

This design has several direct consequences that are important for understanding later mechanics:

1. **Large numbers of blocks with the same state take almost no extra space**: a whole stone floor occupies only one entry in the palette, and every position shares the same ID.
2. **Palette size is limited**: if too many block state types appear in a subchunk, the palette upgrades to a larger storage format (such as direct global-palette mapping), but the principle stays the same.
3. **Writing = changing an ID**: changing the block state at one position is, in essence, changing that position's ID to the ID of another state. If the new state is already in the palette, the write is just a cheap ID replacement.
4. **Shared instances**: because the `BlockState` objects in the palette are pre-generated immutable objects, every position that refers to that state is effectively pointing to the same object.

Once you understand palettes, the bridge between "what the world stores" and "what `BlockState.with()` does" becomes clear: the world stores IDs, the IDs point to the palette, the palette points to pre-generated immutable `BlockState` objects, and `BlockState.with()` returns another pre-generated object.

5 The default state is the starting point for construction

Every block has a **default state**. At the end of `Block` construction, it is set to `StateManager.getDefaultState()` — that is, the first enumerated state combination.

Specific blocks can adjust their default properties in their own constructors. For example, a piston sets its default state to:

```
facing=north
extended=false
```

The default state does not mean that all newly placed pistons face north. During placement, `PistonBlock#getPlacementState` starts from the default state, then uses `.with()` based on the player's look direction to obtain the actual state that should be written to the world. A more accurate description of the default state is: **the starting point for constructing other states — all states are reachable from that point through a finite number of `.with()` calls.**

6 What block state can store

Block state is suitable for storing information that:

- has a finite number of possible values;
- is frequently used in block behavior checks;
- needs to be read quickly;
- usually also needs to participate in rendering or synchronization.

Examples include facing, on/off, power level, age, and waterlogged state.

But data such as the 27 inventory slots inside a chest, the text on a sign, or a hopper's cooldown can take many different values. If all of that were encoded as block state, the number of state combinations would explode rapidly.

That kind of data is stored by a **block entity** (`BlockEntity`). Block state describes "which finite state this position is currently in," while a block entity attaches more complex, independently changing data to certain positions.

6.1 Block entities are not entities

Although both names contain the word "entity," block entities (`BlockEntity`) and entities (`Entity`) are completely different systems:

- **Entity** has coordinates, motion, and a collision box; it can move and actively participates in per-tick computation. Pigs, minecarts, TNT, and item frames all fall into this category.
- **BlockEntity** is fixed to a specific set of block coordinates and does not move on its own. The only reason it exists is that the block at that position needs to store or process data beyond what `BlockState` can represent. Chests, hoppers, signs, and beacons all belong here.

Not every block has a block entity. Whether a position should have one is determined first by the `BlockState` at that position — for example, the chest block state declares "I need a block entity," and only then does the game create a chest block entity for that coordinate. Block entity data is stored in the chunk's block-entity section using NBT format (see the next section), separately from the palette data inside subchunks.

Most block entities participate in some kind of scheduled computation, but with different periods: hoppers check transfers every 8gt, beacons update effects every 80gt, while signs do not participate in any periodic computation at all. For the detailed behavior of block entities, see Block Entities.

7 NBT: the general format for complex data

As mentioned above, `BlockState` can only store information with a finite range of values. So where do highly variable data such as the items inside a chest, the text on a sign, or a hopper's cooldown timer go?

The answer is **NBT** (Named Binary Tag), the tree-structured format Minecraft uses to store complex structured data.

7.1 What NBT is

You can think of NBT as "JSON with types": the data is organized as a tree of key-value pairs, but each value has an explicit type tag (integer, string, list, compound tag, and so on). An NBT tree has a top-level **compound tag** containing multiple named child tags.

NBT is widely used anywhere complex data needs to be stored:

- **Block entities**: chest contents, sign text, hopper cooldowns, beacon effects, etc.
- **Entities**: mob attributes, equipment, potion effects, custom names, etc.
- **Items**: enchantments, durability, custom names, `BlockStateTag`, `BlockEntityTag`, etc.
- **Chunk storage**: block entity data, entity data, scheduled tick queues, etc.
- **Player data**: inventory, ender chest, advancements, statistics, etc.
- **Level data**: world settings, game rules, etc.

7.2 The division of labor between NBT and BlockState

`BlockState` and NBT solve completely different problems:

- `BlockState` answers "which finite state is this position currently in" — the range of values is known, the number of possibilities is controlled, and fast lookup is required.
- NBT answers "what other complex data is attached to this position/entity/item" — the range of values is open-ended, nested structures may be involved, and the data does not need to participate in every block behavior check.

So, at the same set of coordinates in the world, there may simultaneously exist:

- one `BlockState` (guaranteed by the palette), and
- one piece of NBT data (held by the block entity).

The former answers "what block is this, which way does it face, is it on or off"; the latter answers "what is inside the chest, what is written on the sign, how much cooldown is left on the hopper." They have clearly separated responsibilities and do not replace one another.

8 Where fluid state lives

Subchunks do not store another full-size fluid array next to the block state data.

`ChunkSection#getFluidState` first obtains the `BlockState` at that position, then calls the block state's `getFluidState`. Taking stairs as an example:

- when `waterlogged=false`, it returns the empty fluid state;
- when `waterlogged=true`, it returns the still water fluid state.

So at the same set of coordinates, "block state" and "fluid state" are not two completely unrelated pieces of data. Fluid state can be derived from the block state at that coordinate.

9 Why state changes matter

When you pull a lever, open a trapdoor, or change the power of redstone dust, the block type usually does not change. What changes is the `BlockState` of that same block.

However, the game still needs to write the new state back into the chunk. After that write, it may trigger:

- changes to the model and collision shape;
- lighting checks;
- client synchronization;
- NC updates;
- comparator updates;
- block entity creation, removal, or updates.

Therefore, "it didn't turn into another kind of block" does not mean "the world data didn't change." The next article starts from one concrete write operation and follows how those things happen.

10 Summary

- `Block` describes the rules shared by one kind of block.
- `BlockState` describes the finite state combination currently used at one position.
- Properties have finite allowed values, and the state manager pre-creates all combinations.
- `.with(...)` returns another pre-generated state; it does not modify the old one in place.
- Every set of block coordinates in the world corresponds to one block state.
- Complex and highly variable data is usually stored by block entities.
- Fluid state is derived from the block state at the current position.

11 Source Analysis

11.1 Block: the shared rules of a block kind

The `Block` constructor reveals the relationship between a block type and its states:

```java
// Block.java
protected final StateManager<Block, BlockState> stateManager;
private BlockState defaultState;
public Block(AbstractBlock.Settings settings) {
    super(settings);
    StateManager.Builder<Block, BlockState> builder = new StateManager.Builder<>(this);
    this.appendProperties(builder);   // subclasses declare properties here
    this.stateManager = builder.build(Block::getDefaultState, BlockState::new);
    this.setDefaultState(this.stateManager.getDefaultState());
}
```

`appendProperties` is where a subclass declares "which properties this block has." Each subclass overrides it to register its own property set.

11.2 StateManager: enumerating every state combination

The `StateManager` constructor is responsible for building every possible state for a block:

```java
// StateManager.java
// Use flatMap to combine property values layer by layer
Stream<List<Pair<Property<?>, Comparable<?>>>> stream = Stream.of(Collections.emptyList());
for (Property<?> property : this.properties.values()) {
    stream = stream.flatMap(list ->
        property.getValues().stream().map(value -> {
            List<Pair<Property<?>, Comparable<?>>> list2 = Lists.newArrayList(list);
            list2.add(Pair.of(property, value));
            return list2;
        })
    );
}
// Create a BlockState for each combination and store it in an ImmutableList
stream.forEach(list -> {
    ImmutableMap<Property<?>, Comparable<?>> immutableMap = ...;
    S state = factory.create(owner, immutableMap, mapCodec);
    states.add(state);
});
// Build withTable for each state so lookups are O(1)
for (S state : this.states) {
    state.createWithTable(map);
}
this.states = ImmutableList.copyOf(states);
```

`getDefaultState()` returns `this.states.get(0)` — the first state in the list is the default state.

11.3 State: immutable state lookup

`State`'s `entries` is an immutable `ImmutableMap<Property<?>, Comparable<?>>` recording the current value of every property in the current state. `get()` reads directly from `entries`; `with()` uses the pre-built `withTable` to perform an O(1) state transition:

```java
// State.java
private final ImmutableMap<Property<?>, Comparable<?>> entries;
private Table<Property<?>, Comparable<?>, S> withTable;
public <T extends Comparable<T>> T get(Property<T> property) {
    Comparable<?> comparable = this.entries.get(property);
    if (comparable == null) throw ...;
    return property.getType().cast(comparable);
}
public <T extends Comparable<T>, V extends T> S with(Property<T> property, V value) {
    Comparable<?> comparable = this.entries.get(property);
    if (comparable.equals(value)) return (S) this;    // same value -> return self
    S object = this.withTable.get(property, value);    // table lookup
    if (object == null) throw ...;
    return object;
}
```

`with()` does not modify the original state. It retrieves and returns another `BlockState` from the prebuilt state table. Large numbers of blocks in the world with the same state can therefore share the same `BlockState` object instead of each coordinate storing its own independent state instance.

11.4 Concrete block examples

Stairs (`StairsBlock`) declares four properties:

```java
// StairsBlock.java
public static final DirectionProperty FACING = HorizontalFacingBlock.FACING;
public static final EnumProperty<BlockHalf> HALF = Properties.BLOCK_HALF;
public static final EnumProperty<StairShape> SHAPE = Properties.STAIR_SHAPE;
public static final BooleanProperty WATERLOGGED = Properties.WATERLOGGED;
protected void appendProperties(StateManager.Builder<Block, BlockState> builder) {
    builder.add(FACING, HALF, SHAPE, WATERLOGGED);
}
```

`FACING` has 4 horizontal directions, `HALF` has 2 values, `SHAPE` has 5 values, and `WATERLOGGED` has 2 values, for a total of `4 × 2 × 5 × 2 = 80` states.

Pistons (`PistonBlock`) declare two properties and adjust the default value in the constructor:

```java
// PistonBlock.java
public static final BooleanProperty EXTENDED = Properties.EXTENDED;
public PistonBlock(boolean sticky, AbstractBlock.Settings settings) {
    super(settings);
    this.setDefaultState(this.stateManager.getDefaultState()
        .with(FACING, Direction.NORTH)
        .with(EXTENDED, false));        // default state: facing north, not extended
}
protected void appendProperties(StateManager.Builder<Block, BlockState> builder) {
```

```
    builder.add(FACING, EXTENDED);
}
public BlockState getPlacementState(ItemPlacementContext ctx) {
    return this.getDefaultState()
        .with(FACING, ctx.getPlayerLookDirection().getOpposite())
        .with(EXTENDED, false);
}
```

During placement, `getPlacementState` starts from the default state and computes the actual state to write based on the player's look direction. This demonstrates the pattern of "compute the initial state first, then write it."

Redstone dust (`RedstoneWireBlock`) declares five properties: `POWER` (0-15) plus four directional connection properties:

JAVA

```
// RedstoneWireBlock.java
public static final IntProperty POWER = Properties.POWER;  // 0-15
protected void appendProperties(StateManager.Builder<Block, BlockState> builder) {
    builder.add(WIRE_CONNECTION_NORTH, WIRE_CONNECTION_EAST,
        WIRE_CONNECTION_SOUTH, WIRE_CONNECTION_WEST, POWER);
}
```

11.5 Fluid state is derived from block state

`ChunkSection`'s `getFluidState` proves that fluid state is not stored independently:

JAVA

```
// ChunkSection.java
public FluidState getFluidState(int x, int y, int z) {
    return this.blockStateContainer.get(x, y, z).getFluidState();
}
```

Take stairs' `getFluidState` as an example:

JAVA

```
// StairsBlock.java
public FluidState getFluidState(BlockState state) {
    return state.get(WATERLOGGED)
        ? Fluids.WATER.getStill(false)
        : super.getFluidState(state);
}
```

When `waterlogged=true`, it returns water's fluid state; when `false`, it returns the empty fluid state. So the "block state" and "fluid state" at one set of coordinates are in a derived relationship, not two separately stored pieces of data.

11.6 Block's flag constants

JAVA

```
// Block.java
public static final int NOTIFY_NEIGHBORS = 1;      // emit NC updates
public static final int NOTIFY_LISTENERS = 2;      // notify listeners / sync clients
public static final int NOTIFY_ALL = 3;            // combination of 1 | 2, default behavior
public static final int NO_REDRAW = 4;             // suppress client redraw
```

```
6  public static final int REDRAW_ON_MAIN_THREAD = 8; // force synchronous redraw on main thread
7  public static final int FORCE_STATE = 16;         // force-write the state, skip PP update chain
8  public static final int SKIP_DROPS = 32;          // do not generate drops
9  public static final int MOVED = 64;               // mark as piston-moved
```

These flags drive all the follow-up behavior after a write in `World#setBlockState`, which the next article will examine in detail.

11.7 Reference class list

- `net.minecraft.block.Block`
- `net.minecraft.block.BlockState`
- `net.minecraft.state.State`
- `net.minecraft.state.StateManager`
- `net.minecraft.state.property.Property`
- `net.minecraft.block.StairsBlock`
- `net.minecraft.block.PistonBlock`
- `net.minecraft.block.RedstoneWireBlock`
- `net.minecraft.world.chunk.ChunkSection`

4.2 Placing, Changing, and Breaking Blocks

Place a lever on the ground, then pull it.

The first action turns air into a lever; the second does not change the block type, but only changes the lever's state from off to on. Although the two changes are different, they both ultimately need to do the same thing: **write a new block state into the world.**

This article focuses on what happens before and after that write. NC updates, PP updates, and their ordering will be discussed in detail in the Block Updates chapter.

1 The common entry point for changing block state

In 1.20.1, a large number of block changes ultimately go through:

```java
World#setBlockState(BlockPos pos, BlockState state, int flags)
```

It receives three key pieces of information:

- `pos`: which coordinates to change;
- `state`: which block state to write;
- `flags`: which systems must be notified after the write.

`setBlockState` first checks the build-height limits, then obtains the `WorldChunk` that contains those coordinates, and calls `WorldChunk#setBlockState` to actually modify the chunk data.

Therefore, all block changes follow a strict three-step model:

```
Determine the position and the new state
-> Write into the chunk (modify subchunks, heightmaps, lighting, block entities)
-> Notify other systems according to flags (client sync, NC updates, PP update chain)
```

2 Placing a block

When a player right-clicks to place a normal block, the main entry point is `BlockItem#place`.

2.1 Determining the position and state

The game does not simply take the default state and write it directly into the world. Instead, it:

1. checks whether the block is enabled in the current feature set;
2. checks whether the player is allowed to place at the target coordinates;
3. creates an `ItemPlacementContext`;
4. calls the block's `getPlacementState` to compute the initial state;
5. checks whether that state can exist there and whether it collides with entities.

For stairs, this is where facing, top/bottom half, and waterlogging are determined; for pistons, this is where facing is computed from the player's look direction; for torches, this is where the attachment position is checked.

So, "placing a block" does not mean writing the default state first and then adjusting it later. Instead, the correct state for the current environment is computed first, and only then is the write attempted.

2.2 Writing into the chunk

`BlockItem#place` uses, by default:

```
NOTIFY_ALL | REDRAW_ON_MAIN_THREAD
```

when it calls the world's `setBlockState`.

Based on the block coordinates, the chunk finds the corresponding subchunk and subchunk-local coordinates, then replaces the old state in the palette container with the new state. After the replacement succeeds, it also:

> **What do these terms mean?**
>
> - **Palette container**: the way a subchunk compresses and stores the block states of 4096 positions using a "state table + IDs" model; see Blocks and Block States.
> - **Heightmap**: a cache maintained by the chunk so it can quickly find the highest block of a certain kind (such as the highest non-transparent block or highest ocean-floor block).
> - **Lighting check**: a block change may alter light propagation, so the lighting engine must be notified to recompute light in this chunk and neighboring chunks.
> - **`onStateReplaced` / `onBlockAdded`**: logical hooks run before the old state is replaced and after the new state takes effect.
> - **Block entity maintenance**: creating, updating, or removing block entities according to the needs of the new state.
> - **Save mark**: marking the chunk as "changed" so the data will be written to disk later.

- updates the relevant heightmaps;
- schedules lighting checks when needed;
- calls the old state's `onStateReplaced`;
- calls the new state's `onBlockAdded`;
- creates, removes, or updates block entities;
- marks the chunk as needing to be saved.

Here, the "old state" is usually air, but a placement operation may also replace grass, snow layers, fluids, or other replaceable blocks.

2.3 Behavior after placement completes

After the state has been written successfully, `BlockItem#place` also:

- applies any `BlockStateTag` carried by the item;
- writes the block entity NBT from the item into the new block entity;
- calls the block's `onPlaced`;
- triggers the placed-block advancement criterion;
- plays the placement sound;
- emits the `BLOCK_PLACE` game event;
- consumes one item in non-creative mode.

These actions are not part of the block state itself. If the state write fails, the later completion logic for placement does not run either.

3 Changing the state of an existing block

Blocks such as levers, buttons, and trapdoors typically take a new state from the old state during interaction, then call `setBlockState` to write it back to the same position.

For example, the change can be abstracted as:

```
lever[powered=false]
-> lever[powered=true]
```

The block type is still a lever, but the `BlockState` in the subchunk has already been replaced with another state. Therefore, client display changes, NC updates, and save marks may still occur.

`flags` determine what else must happen after the write:

- `NOTIFY_LISTENERS`: notify listeners; on the server this is commonly how clients are synchronized;
- `NOTIFY_NEIGHBORS`: emit NC updates and update comparators when needed;
- if `FORCE_STATE` is not set: continue running the PP update chain caused by the old and new states;
- flags such as `NO_REDRAW` and `REDRAW_ON_MAIN_THREAD` control client redraw behavior.

For those flags and update ordering, see The Concept of Updates and Different Types of Updates.

4 Changes to block entities

When the chunk writes a state, it compares whether the old and new states require a block entity:

- if the old block entity is no longer applicable, it is removed;
- if the new state requires a block entity and the position does not already have one, one is created through `BlockEntityProvider#createBlockEntity`;
- if the block entity remains applicable, its cached block state and ticker are updated.

This shows that a block entity cannot exist independently of block state. Whether a position should have a block entity is determined first by the block state at that position.

5 Player breaking a block

The entry point for a player mining a block is `ServerPlayerInteractionManager#tryBreakBlock`.

The server processes the following in order:

1. whether the tool is allowed to mine the block;
2. permission checks, adventure-mode restrictions, and operator-block restrictions;
3. calling the block's `onBreak`;
4. calling `World#removeBlock`, which replaces the current coordinates with the block state corresponding to the original fluid state;
5. if successful, calling `onBroken`;
6. damaging the tool in non-creative mode;
7. if harvest conditions are met, calling `afterBreak` to generate drops and experience.

There is one easy-to-miss detail here: after breaking a waterlogged block, the target coordinates do not necessarily become air. `World#removeBlock` first gets the `FluidState` at that position, then writes the block state corresponding to that fluid. So when water remains after breaking a waterlogged block, that is not because an extra water block was placed afterward; it is because the replacement state chosen during block removal was a fluid state.

`World#breakBlock` is another general-purpose breaking entry point, used by commands, updates, or other game logic. It can directly handle break effects, drops, state replacement, and the `BLOCK_DESTROY` game event. Do not treat it as exactly the same call path as player mining.

6 Why a chain of changes happens after the write

When a state is written into a chunk, only one thing has been completed: the answer to "what is at these coordinates now" has been changed. To keep surrounding blocks consistent with that result, the game may still need to:

- notify clients;
- re-check lighting;
- update heightmaps;
- emit NC updates;
- make neighboring blocks recompute their own states;
- update comparators;
- create or remove block entities;
- mark the chunk as needing to be saved.

That is exactly where block update theory starts: **one local state write can continue propagating along the relationships between blocks.**

This article only identifies the starting point of that propagation. Its range, direction, and ordering will be expanded in later chapters.

7 Summary

- Many block changes ultimately write into the world through `World#setBlockState`.
- Before placement, the initial state is computed and checked for validity.
- The actual block state is stored inside subchunks.
- A write handles heightmaps, lighting, old/new block callbacks, and block entities.
- `flags` control later behavior such as client notification, NC updates, and PP updates.
- Changing a property of the same block type is still a write to world data.
- Player breaking and generic `World#breakBlock` use different entry points.
- When removing a waterlogged block, the replacement state can be fluid rather than air.

8 Source Analysis

8.1 setBlockState: the common entry point for all changes

`World#setBlockState` is the core method at the "write a block state" level:

```java
// World.java
public boolean setBlockState(BlockPos pos, BlockState state, int flags, int maxUpdateDepth) {
    // 1. boundary checks
    if (this.isOutOfHeightLimit(pos)) return false;
    if (!this.isClient && this.isDebugWorld()) return false;
    // 2. write chunk data
    WorldChunk worldChunk = this.getWorldChunk(pos);
    Block block = state.getBlock();
    BlockState blockState = worldChunk.setBlockState(pos, state, (flags & Block.MOVED) != 0);
    if (blockState == null) return false;
    // 3. confirm the write, then send notifications
    BlockState blockState2 = this.getBlockState(pos);
    if (blockState2 == state) {
        // NOTIFY_LISTENERS -> notify listeners and clients
        if ((flags & Block.NOTIFY_LISTENERS) != 0 && ...) {
            this.updateListeners(pos, blockState, state, flags);
        }
        // NOTIFY_NEIGHBORS -> emit NC updates
        if ((flags & Block.NOTIFY_NEIGHBORS) != 0) {
            this.updateNeighbors(pos, blockState.getBlock());
            if (!this.isClient && state.hasComparatorOutput()) {
                this.updateComparators(pos, block);
            }
        }
        // If FORCE_STATE is not set and depth remains -> run PP update chain
```

```
        if ((flags & Block.FORCE_STATE) == 0 && maxUpdateDepth > 0) {
            int i = flags & ~(Block.NOTIFY_NEIGHBORS | Block.SKIP_DROPS);
            blockState.prepare(this, pos, i, maxUpdateDepth - 1);
            state.updateNeighbors(this, pos, i, maxUpdateDepth - 1);
            state.prepare(this, pos, i, maxUpdateDepth - 1);
        }
        this.onBlockChanged(pos, blockState, blockState2);
    }
    return true;
}
```

Three steps: first write to the chunk, then decide which systems to notify based on `flags`, and finally run the PP update chain. `maxUpdateDepth` limits how far PP updates may propagate (default 512).

8.2 WorldChunk.setBlockState: where the data is really written

JAVA

```java
// WorldChunk.java
public BlockState setBlockState(BlockPos pos, BlockState state, boolean moved) {
    int i = pos.getY();
    ChunkSection chunkSection = this.getSection(this.getSectionIndex(i));
    boolean bl = chunkSection.isEmpty();
    if (bl && state.isAir()) return null;   // writing air into an empty subchunk -> skip
    // 1. write into the subchunk palette container
    int j = pos.getX() & 15;
    int k = i & 15;
    int l = pos.getZ() & 15;
    BlockState blockState = chunkSection.setBlockState(j, k, l, state);
    if (blockState == state) return null;
    // 2. update the four heightmaps
    this.heightmaps.get(Heightmap.Type.MOTION_BLOCKING).trackUpdate(j, i, l, state);
    this.heightmaps.get(Heightmap.Type.MOTION_BLOCKING_NO_LEAVES).trackUpdate(j, i, l, state);
    this.heightmaps.get(Heightmap.Type.OCEAN_FLOOR).trackUpdate(j, i, l, state);
    this.heightmaps.get(Heightmap.Type.WORLD_SURFACE).trackUpdate(j, i, l, state);
    // 3. if subchunk empty/non-empty status changed, update lighting provider
    boolean bl2 = chunkSection.isEmpty();
    if (bl != bl2) {
        this.world.getChunkManager().getLightingProvider().setSectionStatus(pos, bl2);
    }
    // 4. schedule lighting checks when light must be updated
    if (ChunkLightProvider.needsLightUpdate(this, pos, blockState, state)) {
        this.world.getChunkManager().getLightingProvider().checkBlock(pos);
    }
    // 5. call onStateReplaced on the old state
    if (!this.world.isClient) {
        blockState.onStateReplaced(this.world, pos, state, moved);
    } else if (!blockState.isOf(block) && blockState.hasBlockEntity()) {
        this.removeBlockEntity(pos);    // client side: remove old block entity if no longer
        applicable
    }
    // 6. abort if the new block no longer matches (to guard against recursive overwrite in
    onStateReplaced)
    if (!chunkSection.getBlockState(j, k, l).isOf(block)) return null;
    // 7. call onBlockAdded on the new state
    if (!this.world.isClient) {
        state.onBlockAdded(this.world, pos, blockState, moved);
```

```
    }
    // 8. maintain block entities
    if (state.hasBlockEntity()) {
        BlockEntity blockEntity = this.getBlockEntity(pos, WorldChunk.CreationType.CHECK);
        if (blockEntity == null) {
            blockEntity = ((BlockEntityProvider) block).createBlockEntity(pos, state);
            if (blockEntity != null) this.addBlockEntity(blockEntity);
        } else {
            blockEntity.setCachedState(state);  // update cached state
            this.updateTicker(blockEntity);      // update ticker
        }
    }
    this.needsSaving = true;   // mark chunk as needing save
    return blockState;
}
```

Pay attention to the defensive check in steps 5 through 7: recursive `setBlockState` may be triggered inside the `onStateReplaced` callback, so step 6 reads the chunk again to confirm that the current state still matches the new block. If it does not, the method returns `null` early, and the outer `World#setBlockState` also returns `false`.

8.3 BlockItem.place: the full placement flow

JAVA

```
// BlockItem.java
public ActionResult place(ItemPlacementContext context) {
    // 1. check whether the block is enabled in the current feature set
    if (!this.getBlock().isEnabled(context.getWorld().getEnabledFeatures())) return FAIL;
    // 2. check whether the player can place here
    if (!context.canPlace()) return FAIL;
    // 3. compute the initial state
    BlockState blockState = this.getPlacementState(itemPlacementContext);
    if (blockState == null) return FAIL;
    // 4. write into the world
    if (!this.place(itemPlacementContext, blockState)) return FAIL;
    // 5. follow-up work after a successful write
    BlockState blockState2 = world.getBlockState(blockPos);
    if (blockState2.isOf(blockState.getBlock())) {
        blockState2 = this.placeFromNbt(blockPos, world, itemStack, blockState2);
        this.postPlacement(blockPos, world, playerEntity, itemStack, blockState2);
        blockState2.getBlock().onPlaced(world, blockPos, blockState2, playerEntity, itemStack);
        // trigger advancement criterion
        if (playerEntity instanceof ServerPlayerEntity) {
            Criteria.PLACED_BLOCK.trigger((ServerPlayerEntity) playerEntity, blockPos, itemStack);
        }
    }
    // 6. play sound and emit game event
    world.playSound(playerEntity, blockPos, getPlaceSound(blockState2), SoundCategory.BLOCKS,
    ...);
    world.emitGameEvent(GameEvent.BLOCK_PLACE, blockPos, ...);
    // 7. consume the item in non-creative mode
    if (playerEntity == null || !playerEntity.getAbilities().creativeMode) {
        itemStack.decrement(1);
    }
    return ActionResult.success(world.isClient);
```

```
31 }
```

Step 3 here confirms the pattern of "compute the appropriate state first, then write it." Inside step 4, the internal `place` method calls `setBlockState` with `NOTIFY_ALL | REDRAW_ON_MAIN_THREAD` by default. Step 5's `placeFromNbt` and `onPlaced` are only run after the write succeeds — if the write fails, the whole placement flow stops and all later work is skipped.

8.4 World.removeBlock: remove the block, replace it with fluid

```java
// World.java
public boolean removeBlock(BlockPos pos, boolean move) {
    FluidState fluidState = this.getFluidState(pos);
    // Replace it with the block state corresponding to the fluid state at this position
    return this.setBlockState(pos, fluidState.getBlockState(),
        Block.NOTIFY_ALL | (move ? Block.MOVED : 0));
}
```

This is where the behavior of water remaining after a waterlogged block is broken comes from: when removing the block, the replacement state is "the block state corresponding to the fluid at that position," not fixed air.

8.5 World.breakBlock: a general breaking entry point

```java
// World.java
public boolean breakBlock(BlockPos pos, boolean drop, @Nullable Entity breakingEntity,
                          int maxUpdateDepth) {
    BlockState blockState = this.getBlockState(pos);
    if (blockState.isAir()) return false;
    FluidState fluidState = this.getFluidState(pos);
    if (!(blockState.getBlock() instanceof AbstractFireBlock)) {
        this.syncWorldEvent(WorldEvents.BLOCK_BROKEN, pos, Block.getRawIdFromState(blockState));
    }
    // Generate drops when needed
    if (drop) {
        BlockEntity blockEntity = blockState.hasBlockEntity()
            ? this.getBlockEntity(pos) : null;
        Block.dropStacks(blockState, this, pos, blockEntity, breakingEntity, ItemStack.EMPTY);
    }
    // Replace block state with the fluid state's block state
    boolean bl = this.setBlockState(pos, fluidState.getBlockState(),
        Block.NOTIFY_ALL, maxUpdateDepth);
    if (bl) {
        this.emitGameEvent(GameEvent.BLOCK_DESTROY, pos, ...);
    }
    return bl;
}
```

Unlike `removeBlock`, `breakBlock` adds one more step: it handles drops before replacing the block state.

8.6 ServerPlayerInteractionManager.tryBreakBlock: player breaking entry point

```java
// ServerPlayerInteractionManager.java
public boolean tryBreakBlock(BlockPos pos) {
    BlockState blockState = this.world.getBlockState(pos);
    // 1. whether the tool is allowed to mine
    if (!this.player.getMainHandStack().getItem().canMine(blockState, this.world, pos,
    this.player))
        return false;
    // 2. operator-block permission check
    Block block = blockState.getBlock();
    if (block instanceof OperatorBlock && !this.player.isCreativeLevelTwoOp()) { ... return false;
    }
    // 3. adventure-mode restriction
    if (this.player.isBlockBreakingRestricted(this.world, pos, this.gameMode)) return false;
    // 4. call onBreak
    block.onBreak(this.world, pos, blockState, this.player);
    // 5. remove the block (-> removeBlock -> setBlockState)
    boolean bl = this.world.removeBlock(pos, false);
    // 6. call onBroken
    if (bl) block.onBroken(this.world, pos, blockState);
    // 7. non-creative mode: damage tool
    ItemStack itemStack = this.player.getMainHandStack();
    ItemStack itemStack2 = itemStack.copy();
    itemStack.postMine(this.world, blockState, pos, this.player);
    // 8. if harvestable, generate drops and experience
    boolean bl2 = this.player.canHarvest(blockState);
    if (bl && bl2) {
        block.afterBreak(this.world, this.player, pos, blockState, blockEntity, itemStack2);
    }
    return true;
}
```

The flow is clear: permission checks -> `onBreak` -> `removeBlock` (which internally calls `setBlockState`) -> `onBroken` -> tool damage -> drops. Note that `removeBlock(false)` passes `move=false`, so the `MOVED` flag is not set.

8.7 Reference class list

- `net.minecraft.world.World`
- `net.minecraft.world.chunk.WorldChunk`
- `net.minecraft.world.chunk.ChunkSection`
- `net.minecraft.item.BlockItem`
- `net.minecraft.block.Block`
- `net.minecraft.block.BlockEntityProvider`
- `net.minecraft.server.network.ServerPlayerInteractionManager`

CHAPTER 5

5 Chunks

IN THIS CHAPTER

5.1 Fundamental Concepts and Structure
5.2 Generation State and ChunkStatus
5.3 Levels and Runtime State Layers
5.4 Server Master Dispatcher TACS
5.5 Main Thread Async Tasks
5.6 ChunkHolder Lifecycle
5.7 Scheduled Ticks and Chunk Runtime Levels
5.8 Random Ticks and Block Entity Ticks
5.9 Entity Loading and Tracking
5.10 Save and Unload Pipeline
5.11 POI and Bypass Storage
5.12 Client Visibility and Chunk Packets
5.13 Game Event and In-Chunk Listeners

Introduction

1 Research Background and Significance

For a long time, community discourse on 1.14+ chunk loading has focused on analyzing **steady states** — we know that nether portal loaders will cause 3×3 chunks in the target dimension to be loaded, but are these chunks loaded instantly or after multiple game ticks? This question is rarely addressed.

Meanwhile, chunk management is an important and quite complex system in Minecraft, intimately related to nearly every aspect of the world. From the most fundamental "block queries" to "entity simulation," from "player view distance" to "chunk saving," the chunk management system precisely controls "which chunks exist, which chunks are simulating, which chunks should be saved" behind the scenes.

This chapter aims to enhance readers' understanding of the underlying working mode of the chunk management system, help readers understand the source code organization structure, and thereby discover more creative applications based on chunk management mechanisms.

2 Research Environment and Conventions

This chapter is based on the following environment:

- Minecraft Java Edition **1.20.1**
- Deobfuscation mapping: **Yarn Mapping** `1.20.1+build.10`
- Parts of the analysis are based on lovexyn0827's early research on 1.16.4 and 1.20.1 (CC0 license)

Special conventions used in the text:

- **Module** refers to an object unique within an entire dimension, usually corresponding to a Java class, sometimes also referring to a relatively independent subsystem.
- **"Loading"** does not specifically refer to the commonly understood accessible, weak-loaded, or strong-loaded states, but rather the state of chunks existing in memory and the process of chunks being read from disk into memory.
- Distances in the text, unless otherwise specified, are all **Chebyshev distance** (Chebyshev Distance), not Euclidean distance. For example, the Chebyshev distance from chunk `(0, 0)` to chunk `(2, 2)` is `max(2, 2) = 2`.

3 Basic Structure of This Chapter

1. **Basic Concepts and Structure**: Starting from the mathematical definition of chunks, understand sub-chunks, local coordinate packing, differences between ProtoChunk and WorldChunk, and the client chunk pipeline.
2. **Generation Status and ChunkStatus**: Deep dive into the 12-step chunk generation pipeline, understand outer ring dependencies, taskMargin, and bypass systems like lighting/storage upgrades.

3. **Level and Runtime Stratification**: Distinguish ChunkStatus (generation state) and ChunkLevelType (runtime qualification), establish the mental model "generated ≠ simulating."
4. **Server Master Scheduler TACS**: The complete responsibilities of ThreadedAnvilChunkStorage — from currentChunkHolders to chunksToUnload, from worldgen/light/main three executor types to ChunkTaskPrioritySystem's priority scheduling.
5. **Main Thread Async Tasks**: Understand the core role of CompletableFuture in chunk management — almost all operations aren't executed directly, but planned as Futures and then executed in async task queues.
6. **Loading Ticket Mechanism**: Full table of Ticket types, lifecycle, player loading, ticket propagation and expiry reclamation.
7. **ChunkHolder Lifecycle**: How the three core Futures (accessibleFuture, tickingFuture, entityTickingFuture) act like a thermometer controlling three "runtime views" of chunks.
8. **Scheduled Ticks and Chunk Runtime Level**: How scheduled ticks depend on a chunk's BLOCK_TICKING level — tickingFutureReadyPredicate determines whether scheduled ticks can execute, and where scheduled ticks go when chunks aren't ticking.
9. **Random Ticks and Block Entity Ticks**: Random tick sampling mechanism, how ChunkSection counters skip empty sections, block entity ticker registration and execution, and how all three collectively constitute "block ticking."
10. **Entity Loading and Entity Tracking**: The design philosophy of two independent management systems for entities and blocks, ServerEntityManager's responsibilities, how EntityTracker synchronizes — and why ENTITY_TICKING requires 5×5 FULL.
11. **Save and Unload Pipeline**: Three-level unload structure (unloadedChunks → chunksToUnload → unloadTaskQueue), save throttling, ChunkSerializer's serialization responsibilities, and how StorageIoWorker ultimately writes to .mca.
12. **POI and Bypass Storage**: Why Point of Interest is stored independently from regular chunks — its data structure, how block changes trigger updates, and query relationships with villagers, raids, and wandering traders.
13. **Client Visibility and Chunk Packets**: How the server decides which chunks to send to clients, division of labor between chunk data packets and incremental update packets, geometric determination of watchDistance, and the batch sending mechanism of flushUpdates.
14. **Game Event and In-Chunk Listeners**: The essential difference between the game event system and NC/PP updates, WorldChunk's section-based management of event dispatchers, and cleanup of sculk listeners during chunk unload.
15. **Save Throttling and Savestate Phenomenon**: TACS's 10-second save cooldown, StorageIoWorker async disk write, differences between auto-save/flush/shutdown, and how Watchdog and OOM form or break cross-chunk state forks.

4 Prerequisites

This text assumes readers have some familiarity with basic concepts in Minecraft:

- Blocks and Block States — The difference between Block and BlockState

5.1 Fundamental Concepts and Structure

Press `F3 + G`, and colored grids appear on the ground. These grids, spaced 16 blocks apart, show **chunk boundaries**.

Understanding the chunk management system requires more than knowing "16×16 grid squares". Let's start from the bottom.

1 Chunk and Chunk Section Definition

1.1 Mathematical Definition of a Chunk

A chunk is a data storage and loading unit in Minecraft. Strictly speaking, a chunk corresponds to the region:

$$\{(x, y, z) \mid 16m \leq x < 16(m+1),\ 16n \leq z < 16(n+1),\ m, n \in \mathbb{Z}\}$$

This means:

- Chunks are fixed at 16×16 in the **horizontal direction**, with boundaries at multiples of 16;
- Chunks have no upper limit in the **vertical direction** — they span the **full buildable height** of the dimension. In the Overworld, all 384 blocks in a column from Y=`-64` to Y=`319` belong to the same chunk;
- Chunks have only X and Z coordinates, no independent Y coordinate.

1.2 Chunk Coordinate Conversion

Conversion rules are strictly defined by three bitwise operations:

```
Chunk X  = Block X >> 4    (arithmetic right shift 4 bits = floor division by 16)
Chunk Z  = Block Z >> 4
In-chunk X = Block X & 15  (take low 4 bits = remainder after division by 16)
In-chunk Z = Block Z & 15
```

Reverse calculation follows the same strict rules:

```
Chunk start block X = Chunk X << 4
Chunk eastmost block X = Chunk start block X + 15
```

> Chunk ownership for negative coordinates is easy to miscalculate. `>>` is arithmetic right shift (floor division), so `-1 >> 4 = -1`, `-16 >> 4 = -1`, `-17 >> 4 = -2`. Block X=`-1` is not in chunk 0, but in chunk **-1** at the eastmost position (in-chunk X=15). Memory aid: **negative coordinates round toward -∞**.

1.3 Chunk Sections

Chunks themselves are tall, but the game doesn't need to traverse entire vertical columns every time. The chunk continues to be sliced along the Y axis into **16×16×16 chunk sections**. Each chunk section contains exactly `16³ = 4096` coordinates.

Conversion rules are **completely isomorphic** to chunk coordinates:

```
Section Y  = Block Y >> 4
In-section Y = Block Y & 15
```

`Chunk` creates a `ChunkSection[]` array based on dimension height. The Overworld has 384 blocks of height, so a complete Overworld chunk contains `384 / 16 = 24` chunk sections.

Section coordinate ≠ array index. In the Overworld:

- Bottommost section coordinate: `-64 >> 4 = -4`
- Array index 0 corresponds to section coordinate -4
- `sectionArray[23]` corresponds to section coordinate `-4 + 23 = 19` (covering Y=`304` to Y=`319`)

Conversion formula: `Array index = Section coordinate - Bottommost section coordinate`

1.4 Local Coordinate Packing

The game frequently needs to represent "which row, column, layer" as local coordinates within a 16³ chunk section. `ChunkSectionPos` provides a method to pack three 0-15 integers into a single `short`:

```java
// Pack: x(4bit) | z(4bit) | y(4bit) → total 12 valid bits
short packed = (short)(x << 8 | z << 4 | y);
```

Unpacking is the reverse:

```java
int x = (packed >> 8) & 15;
int z = (packed >> 4) & 15;
int y = packed & 15;
```

This packing **isn't just a space-saving trick** — it's used throughout the codebase:

- Chunk section skylight and blocklight storage use 16-bit arrays indexed by packed coordinates;
- Random tick candidate list uses 16-bit arrays to store which coordinates need ticking;
- Block entity position references often use packed coordinates rather than `BlockPos`.

The cost of packing is negligible: two shift operations plus two bitwise ANDs. The gain is halving the storage space and improving CPU cache locality.

2 Why ProtoChunk and WorldChunk Exist

Minecraft doesn't use a single "Chunk" class for all scenarios. It distinguishes between two fundamentally different situations:

	PROTOCHUNK	WORLDCHUNK
Purpose	Temporary carrier during generation	Data structure in the running world
Existence phase	Only during `EMPTY` → `FULL` generation process	From `FULL` to unload
Modifiability	Writable (builder mode)	Read-write (runtime state)
Block entity operations	No complex ticker scheduling	Full block entity ticking system
Entity handling	Only stores entity list	Full entity simulation (when ENTITY_TICKING)
Client visibility	Non-existent	Sent to clients (when in view)

2.1 ProtoChunk: Generation Carrier

`ProtoChunk` is the mutable data structure used during chunk generation. Each generation step (from `EMPTY` to `FEATURES`) gradually fills its internal data:

- `sections[]` progressively fills with block states
- `heightmaps` records terrain height during `NOISE`
- `carvingMask` marks cave/ravine boundaries during `CARVING`
- `lights` accumulates light propagation ranges
- `entities` simply stores entity NBT for later transfer

After finishing generation, `ProtoChunk` is directly converted to `WorldChunk`:

```java
public WorldChunk convertToWorldChunk(ServerLightingProvider lighting, Chunk chunk) {
    // Batch transfer entities
    chunk.getEntitySectionStorage().addAll(this.entities);
    // Transfer block ticks
    this.transferTicksToSections(this.blockTickScheduler, Chunk::createBlockTickScheduler);
    this.transferTicksToSections(this.fluidTickScheduler, Chunk::createFluidTickScheduler);
    // ...other transfer logic
    return worldChunk;
}
```

Note that the conversion **doesn't copy data from `ProtoChunk` to `WorldChunk`** — instead, it **transfers ownership** of sections, heightmaps, block tick schedulers, etc. Memory addresses remain unchanged; only the wrapper changes. This is why generation and post-generation use different classes: generation needs build pattern design, runtime needs minimal state machine overhead.

2.2 WorldChunk: Runtime Carrier

Once generation is complete, the chunk becomes `WorldChunk`. This class adds runtime-specific functionality:

- **Block entity ticker registration**: WorldChunk maintains `blockEntityTickers` for efficient tick iteration

- **Entity section management**: Entity storage switches to section-sliced indexing for spatial queries
- **Client synchronization**: Tracks dirty sections (`chunkNeedsLight`, `lightUpdateTime`) to determine sync packets
- **Game event listeners**: Manages per-section `GameEventDispatcher` (Sculk Sensor, Warden)

Reading a block state from `WorldChunk` is simpler than from `ProtoChunk`, because the `PalettedContainer` in sections is already initialized and doesn't need fallback logic. This is also why `setBlockState()` in WorldChunk directly triggers neighbor updates and block entity state sync — because WorldChunk **is the world**.

3 Palette: The Data Structure Behind Block Storage

Most players know chunks are 16×16×384 (Overworld), but how does the game store 98,304 block states? Not as a 16^3 three-dimensional array — that would make chunk files gigabytes large. The real structure is `PalettedContainer<BlockState>`.

3.1 Basic Structure

```java
public class PalettedContainer<T> {
    private Palette<T> palette;              // Dictionary: index → actual value
    private PackedIntegerArray data;         // Bit-packed array: coordinate → palette index
    private int bits;                        // Bits per entry
}
```

The core idea: instead of storing each block state's full ID (16 bits), store an **index** into a local palette (typically 4-8 bits).

3.2 Read/Write Process

Write:

```java
public T swap(int index, T value) {
    int paletteIndex = palette.index(value);
    if (paletteIndex < 0) {                  // Not in palette
        paletteIndex = palette.add(value);   // Add to palette
        if (paletteIndex >= (1 << bits)) {   // Palette full
            resize();                        // Expand bit width or switch to global palette
        }
    }
    return data.swap(index, paletteIndex);
}
```

Read:

```java
public T get(int index) {
    int paletteIndex = data.get(index);
    return palette.get(paletteIndex);
}
```

3.3 Palette Types

Minecraft uses three kinds of palettes based on diversity:

PALETTE TYPE	CONDITION	INDEX BITS	CHARACTERISTICS
`SingleValuePalette`	Only 1 unique block state	0	No data array needed, directly returns the same value
`ArrayPalette`	2-16 unique block states	4-8	Most space-efficient, common case
`GlobalPalette`	>16 unique block states	16	Directly stores global registry IDs, no compression

When placing new blocks in a section, the palette may expand from `ArrayPalette` to `GlobalPalette`. This process triggers `resize()`, which reallocates the `data` array and remaps all entries. This is why chunk generation performance degrades with more diverse block states — more reallocation overhead.

Design decision — why separate palette and data arrays? The `palette` stores unique block states that appear in the section, while `data` stores indices into that palette. If a section contains only stone and air:

- Without palette: 4096 entries × 16 bits (block state ID) = 8192 bytes
- With palette: `palette=[stone, air]` + 4096 entries × 1 bit (index 0 or 1) = ~512 bytes

The palette is **dictionary compression** — the more homogeneous the chunk, the higher the compression ratio. A section filled with air stores only `palette=[air]` and doesn't even need a `data` array.

3.4 Block State ID Packing

Block state IDs are composed of:

```
Block ID (registry order) + State properties (metadata)
```

For example, `minecraft:oak_stairs` has properties `facing`, `half`, `shape`, `waterlogged`, with combinations `4 × 2 × 5 × 2 = 80` states. The registry assigns 80 consecutive IDs to this block. When storing, the game packs:

```java
int stateId = Block.STATE_IDS.getRawId(blockState);
```

This ID is what gets stored in `PalettedContainer`. When reading:

```java
BlockState blockState = Block.STATE_IDS.get(stateId);
```

Why not store `(block ID, properties)` separately? Packing into a single integer allows bitwise operations and table lookups — both faster than map queries. The cost is larger ID space, but since the palette compresses it, actual storage doesn't increase significantly.

4 Client Chunk Pipeline

This chapter focuses on server-side chunk management, but it's worth briefly understanding the client side to clarify where responsibilities are divided.

4.1 Client-Server Division

RESPONSIBILITY	SERVER	CLIENT
Chunk generation	☒	☒
Block entity ticking	☒	☒ (client-side block entities are only for rendering)
Entity simulation	☒	☒ (client entities interpolate based on server updates)
Chunk rendering	☒	☒
Lighting computation	☒ (authoritative source)	☒ (receives and applies server light data)

The client **doesn't generate chunks** — it only receives chunk data from the server and renders it. Even in singleplayer mode, the integrated server generates chunks and sends them to the client running in the same process.

4.2 ProtoChunk and WorldChunk: The Design Philosophy of Two Chunk Types

```java
// ProtoChunk.java
public class ProtoChunk extends Chunk {
    // No levelTypeProvider
    // No ServerEntityManager integration
    // No BlockTickScheduler integration
    // -- Its job is to "be generated," not to "be run"
}
// WorldChunk.java
public class WorldChunk extends Chunk {
    private Supplier<ChunkLevelType> levelTypeProvider;
    public ChunkLevelType getLevelType() {
        return this.levelTypeProvider == null
            ? ChunkLevelType.FULL : this.levelTypeProvider.get();
    }
    public void setLevelTypeProvider(Supplier<ChunkLevelType> levelTypeProvider) {
        this.levelTypeProvider = levelTypeProvider;
    }
    // Holds BlockTickScheduler and FluidTickScheduler
    // Holds GameEventDispatcher
    // Manages entities through ServerEntityManager
    // -- Its job is to "be run"
}
```

The design idea behind this is **separation of concerns**. `ProtoChunk` does not need to know what a "tick" is or what "mob spawning" is. It only needs to be gradually filled by the world-generation pipeline on the generation thread. `WorldChunk` does not need to know what "noise terrain" is or what "cave carving" is. It only needs to participate in per-gt game logic on the main thread.

But both share the `Chunk` base class. This means they have the same data structures such as `sectionArray`, `heightmaps`, and `blockEntities`. The elegance of this design is that at the `FULL` stage, `ProtoChunk` does not need to be "copied." Its `ChunkSection[]` is passed directly to the converted `WorldChunk`. The other parts of the `ProtoChunk` object are discarded (and later reclaimed by GC), but the block data itself enters the runtime state intact.

Why is `levelTypeProvider` a `Supplier`? Because a chunk's runtime level can **change dynamically**. The same `WorldChunk` may switch back and forth among 31 (`ENTITY_TICKING`), 32 (`BLOCK_TICKING`), and 33 (`FULL`). If `levelType` were stored directly as a field, it would need to be updated every time the level changed. But the trigger for that update lives in `ChunkHolder`, not in `WorldChunk` itself. Using a `Supplier` lets `WorldChunk` **query `ChunkHolder` directly for the current level**, avoiding the synchronization cost of having to notify `WorldChunk` on every level change.

`setLevelTypeProvider` is called by `ThreadedAnvilChunkStorage` when the chunk first enters runtime state (around line 744):

```java
// ThreadedAnvilChunkStorage.java (simplified)
worldChunk.setLevelTypeProvider(() -> ChunkLevels.getType(chunkHolder.getLevel()));
```

4.3 TACS Table Structure Design

```java
// ThreadedAnvilChunkStorage.java
private final Long2ObjectLinkedOpenHashMap<ChunkHolder> currentChunkHolders;
private volatile Long2ObjectLinkedOpenHashMap<ChunkHolder> chunkHolders
    = this.currentChunkHolders.clone();
private final Long2ObjectLinkedOpenHashMap<ChunkHolder> chunksToUnload;
private final LongSet loadedChunks;
final LongSet unloadedChunks;
```

Why is `chunkHolders` (the snapshot) necessary? `currentChunkHolders` changes frequently. Each tick may create or remove dozens of chunks. But operations such as `tick()` and `save()` need to traverse all holders stably. If they iterated `currentChunkHolders` directly, additions or removals during traversal could trigger `ConcurrentModificationException`. `chunkHolders` is a `volatile` clone that provides an **immutable snapshot of the current moment**. Iterators can safely traverse it without being affected by later modifications. The performance cost is one full clone, usually tens to hundreds of entries, which is far lower than the cost of locking during traversal.

What is the difference between `unloadedChunks` and `chunksToUnload`? `unloadedChunks` is the `LongSet` that `setLevel()` immediately adds to when a chunk leaves accessible range. It is only a "marker set" and does not carry the `ChunkHolder` itself. `chunksToUnload`, by contrast, stores the complete `ChunkHolder` object. In `setLevel()`, if a `ChunkHolder` is found to already exist in `chunksToUnload` (meaning this chunk left accessible range before and then came back), it is reused directly rather than recreated. This avoids frequent object creation in a repeated "leave-return-leave" cycle.

5 References

- Discovering Minecraft - Basic Chunk Management Concepts (CC0 License)
- `net.minecraft.world.chunk.Chunk`
- `net.minecraft.world.chunk.ProtoChunk`
- `net.minecraft.world.chunk.WorldChunk`
- `net.minecraft.world.chunk.ChunkSection`
- `net.minecraft.world.chunk.PalettedContainer`
- `net.minecraft.util.math.ChunkSectionPos`
- `net.minecraft.util.math.ChunkPos`

5.2 Generation State and ChunkStatus

We already know chunks have two forms: `ProtoChunk` and `WorldChunk`. But how does a `ProtoChunk` evolve from nothing, step by step, into a `WorldChunk`?

The answer lies in `ChunkStatus`.

1 Chunk Generation Pipeline

`ChunkStatus` corresponds to a stage in chunk generation. In 1.20.1, there are **12 stages**, arranged in order:

INDEX	STATUS	PRIMARY TASK
0	`EMPTY`	Starting point, contains no data
1	`STRUCTURE_STARTS`	Determine which structures (villages, strongholds) may start in the chunk
2	`STRUCTURE_REFERENCES`	Add structure references for coordination between structures during world generation
3	`BIOMES`	Determine biomes (based on noise sampling)
4	`NOISE`	Generate rough terrain outline and bedrock layer
5	`SURFACE`	Replace blocks near the surface (e.g., stone → dirt, grass)
6	`CARVERS`	Generate caves and canyons
7	`FEATURES`	Generate features (trees, ores, flowers, etc.)
8	`INITIALIZE_LIGHT`	Initialize lighting calculations (added in 1.20)
9	`LIGHT`	Complete lighting calculations
10	`SPAWN`	Spawn initial creatures (wild animals, etc.)
11	`FULL`	Convert `ProtoChunk` to `WorldChunk`

Each stage has a corresponding **`GenerationTask`** (when generating from scratch) or **`LoadTask`** (when loading from disk), defining the operations needed to advance from the previous state to the current state. The `ChunkStatus.ChunkType` enum distinguishes two chunk types:

- `PROTOCHUNK`: Under generation, uses `ProtoChunk`;
- `LEVELCHUNK`: Completed (`FULL` stage), uses `WorldChunk`.

> A mnemonic for generation-to-completion stages (1.16.4 version, slightly different from 1.20.1):
>
> Empty, starts, references, biomes, noise, surface.
> Carvers, features, lighting, spawn — done at last.

2 Outer Ring Dependencies

While `FULL` marks "chunk generation complete", this doesn't mean a single isolated chunk can progress all the way to `FULL` alone.

Outer ring dependencies (outskirt dependency) mean: for a chunk to advance to a certain `ChunkStatus`, chunks within a certain distance must reach at least a minimum state. This distance is determined by **`taskMargin`**.

Take noise terrain as an example: to generate terrain at `(0, 0)`, the generator needs to read biome information from surrounding chunks to complete interpolation. If surrounding chunks haven't even determined biomes, the central chunk's terrain cannot be correctly generated. Therefore, the `NOISE` stage's `taskMargin` is set to 8 — chunks within an 8-chunk radius must have at least completed `BIOMES`.

Each stage's `taskMargin` is roughly:

STAGE	TASKMARGIN	EXPLANATION
`STRUCTURE_STARTS`	0	Structure starting positions depend only on this chunk
`STRUCTURE_REFERENCES`	8	Needs reference information from surrounding structures
`BIOMES`	8	Biome interpolation needs surrounding noise data
`NOISE`	8	Terrain generation needs surrounding biome information
`SURFACE`	8	Surface building needs surrounding noise information
`CARVERS`	8	Cave generation needs to connect to surrounding chunks
`FEATURES`	8	Features (like large trees) may cross chunk boundaries
`INITIALIZE_LIGHT`	0	Only initializes this chunk's lighting
`LIGHT`	1	Lighting calculations need coordination with neighboring chunks
`FULL`	0	Conversion requires no external information

> The existence of `taskMargin` means: **even if loading tickets only require loading chunks in the central area, chunks within a certain surrounding range will also be triggered to the corresponding ChunkStatus**. This is why terrain shown by F3+G is always one ring larger than the strong loading range — those outer ring chunks don't participate in ticking, but their generation is partially complete.

2.1 taskMargin and Chebyshev Distance

`taskMargin` uses **Chebyshev Distance** as the distance metric. For chunk coordinates `(x₁, z₁)` to `(x₂, z₂)`:

$$\text{Distance} = \max(|x_1 - x_2|, |z_1 - z_2|)$$

For example, distance from `(0, 0)` to `(8, 0)` = 8, and to `(8, 8)` is also 8. This means when `taskMargin = 8`, generation affects a square region centered on the target chunk with side length `2×8+1 = 17`.

3 Handling Unreachable Generation

Not all chunks can reach `FULL`:

- A chunk far from any ticket may only reach `BIOMES` and stop;
- A chunk close to a ticket but with old save data may load to `SURFACE` and stop (if further stages aren't needed).

When requesting a higher `ChunkStatus` than a chunk can reach:

```java
// In ChunkHolder.getOrCreateFuture()
if (targetStatusIndex > completedStatusIndex) {
    // Target status not yet reached → return pending Future, suspend task
}
```

The caller receives a `CompletableFuture` that remains **pending** until the chunk reaches that state or is unloaded — at which point the Future completes with `ChunkHolder.UNLOADED_WORLD_CHUNK`.

4 Lighting and Save Upgrade Bypass

4.1 Why Lighting Has Two Stages

1.20 introduced `INITIALIZE_LIGHT` between `FEATURES` and `LIGHT`. Why two stages?

The problem: Old lighting algorithms (`LIGHT` stage) required reading a large area around the chunk — if neighboring chunks weren't generated, the operation would block. This caused stuttering.

The solution: Split into two stages:

- **`INITIALIZE_LIGHT`** (taskMargin=0): Pre-calculate this chunk's own block light and skylight, without reading neighbors;
- **`LIGHT`** (taskMargin=1): Use neighboring chunk data to refine edge lighting.

This way, most lighting work completes in the first stage (no wait), with only edge refinement in the second stage (short wait).

4.2 Save Upgrade Path

When loading chunks saved in old versions (e.g., 1.12 → 1.20), the generation pipeline can't simply replay all 12 stages — old terrain doesn't need regeneration, only adaptation.

`ChunkSerializer` defines a **save upgrade path**:

```java
// ChunkSerializer.deserialize() simplified
if (chunkNbt.contains("Status", NbtType.STRING)) {
    String statusName = chunkNbt.getString("Status");
```

```
    ChunkStatus savedStatus = ChunkStatus.byName(statusName);
    // Apply DataFixer for version migration
    chunkNbt = DataFixTypes.CHUNK.update(...);
}
```

Upgrade logic:

- If saved status ⩾ `FEATURES` → skip all generation stages, directly set to saved status;
- If saved status < `FEATURES` → replay from `EMPTY` (old terrain incompatible, must regenerate);
- Lighting stages (`INITIALIZE_LIGHT`, `LIGHT`) always re-execute even if saved (marked `shouldAlwaysUpgrade=true`).

Why force lighting recalculation? Because surrounding chunks may have changed after saving. Old lighting data is no longer trustworthy.

5 Summary

- `ChunkStatus` represents a stage in the chunk generation pipeline, with 12 stages total;
- Each stage may depend on surrounding chunks (taskMargin), using Chebyshev distance;
- Chunks may stop at any stage, with the furthest reachable stage determined by loading tickets;
- Lighting is split into two stages (`INITIALIZE_LIGHT` + `LIGHT`) to reduce blocking;
- Old saves bypass partial generation stages but force lighting recalculation.

6 Code Walkthrough

6.1 Why Generation Tasks Use Executor

Each `ChunkStatus` has an associated `Executor` type:

```java
// ChunkStatus.java (excerpt)
public enum TaskType {
    MAIN_THREAD,      // Execute on server main thread
    ASYNC             // Execute on independent generation thread pool
}
```

Most generation stages (NOISE, SURFACE, CARVERS, FEATURES) use `ASYNC` — they run on independent threads without blocking the main thread. Only a few stages (like FULL conversion, certain lighting operations) use `MAIN_THREAD` — because they need to access runtime state (entities, ticking logic).

Why can't FULL stage be ASYNC? Because `WorldChunk` participates in runtime scheduling — it needs to register scheduled ticks, load entities, trigger adjacent block updates. These operations can only run on the main thread, otherwise they'd face concurrency conflicts.

6.2 taskMargin Design Details

```java
// ChunkStatus.java (excerpt)
```

```
    private final int taskMargin;
    public int getDistanceFromFull() {
        return ChunkStatus.FULL.index - this.index;
    }
    public int getMaxDistanceFromFull() {
        return this.getDistanceFromFull() + this.taskMargin;
    }
```

`getMaxDistanceFromFull()` returns "how many chunks away from the center, at minimum, must outer ring chunks reach to allow center chunk to advance to this stage". For example:

- `NOISE.taskMargin = 8` → outer ring chunks need to reach `BIOMES` (one stage before NOISE);
- Center chunk reaching `NOISE` requires: `taskMargin=8` chunks away reach at least `BIOMES`.

Scheduling logic uses this value to pre-trigger outer ring chunk generation:

JAVA

```
// ChunkHolder.createFuture() simplified
for (int r = 0; r <= status.taskMargin; r++) {
    // Ensure chunks at distance r reach the required preceding status
    if (!canReach(r, status.getPreviousStatus())) {
        // Cannot reach → returned CompletableFuture stays pending, task suspended
    }
}
```

Why does `CARVERS` need taskMargin=8 while `LIGHT` only needs 1? During cave generation, a cave may span multiple chunks. If caves are only generated within one chunk, "suddenly severed caves" appear at boundaries — a visual error. taskMargin=8 ensures chunks within 8 blocks of the cave-generating center chunk have at least completed preceding stages (`SURFACE`), allowing caves across the entire region to generate seamlessly.

Lighting calculations only need taskMargin=1 because light propagation only needs brightness values from adjacent chunks — after the first layer of neighbors completes, lighting can be correctly calculated.

The meaning of `shouldAlwaysUpgrade`: Certain states (like `INITIALIZE_LIGHT` and `LIGHT`) are marked `shouldAlwaysUpgrade=true`. This means even when loading old chunks from disk where that state has already "completed", the system will **forcibly re-execute** that stage's calculations. Why does lighting need this? Because lighting depends on surrounding chunk states — and surrounding chunks may have been modified after saving. Forcing lighting recalculation guarantees loaded chunk lighting is correct.

6.3 GenerationTask Scheduling

Each `ChunkStatus` carries a `GenerationTask` when registered (using the default `STATUS_BUMP_LOAD_TASK` if not specified):

JAVA

```
// ChunkStatus.java
interface GenerationTask {
    CompletableFuture<Either<Chunk, Unloaded>> doWork(
        ChunkStatus targetStatus, Executor executor, ServerWorld world,
        ChunkGenerator generator, ..., List<Chunk> chunks, Chunk chunk);
}
```

> Scheduling logic in `runGenerationTask()`: it takes the middle chunk from the `chunks` list (collected by `getRegion` from surrounding chunks) as the target, calls `generationTask.doWork()`. After task completion, it marks the current stage complete with `chunk.setStatus(this)`, then passes the result to `futuresByStatus[this.index]`, triggering Futures waiting for this stage.
>
> Two design details worth noting:
>
> 1. **Tasks execute on independent generation threads** (`worldGenExecutor`), not the main thread. This is why `CompletableFuture` is necessary — the main thread cannot block waiting for generation to complete.
> 2. **getRegion's margin parameter**: For `makeChunkTickable`, `margin=1` means requiring all surrounding 9 chunks (3×3) to reach `FULL`. This is not taskMargin — taskMargin controls propagation **within the same stage**, while margin controls **dependencies between different stages**.

7 References

- Discovering Minecraft - ChunkStatus and World Generation (CC0 License)
- `net.minecraft.world.chunk.ChunkStatus`
- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.server.world.ServerLightingProvider`
- `net.minecraft.world.chunk.ChunkSerializer`
- `com.mojang.datafixers.DataFixer`

5.3 Levels and Runtime State Layers

In the last chapter we learned about `ChunkStatus` — it answers "how much of this chunk has been generated".

But Minecraft has another dimension of question: "what is this chunk **currently doing**?" — Can it be accessed by normal code? Is it running block ticks? Are entities moving?

These two questions are answered by two different systems.

1 ChunkStatus ≠ Chunk Heat

This is a common confusion that must first be clarified:

	CHUNKSTATUS	CHUNKLEVEL / CHUNKLEVELTYPE
Question Answered	"How far has this chunk been generated?"	"What operations is this chunk currently allowed to perform?"
Determining Factor	Whether generation pipeline stages are complete	Loading level calculated by the loading ticket system
Change Timing	Generation progression (usually irreversible)	Loading ticket additions/removals (dynamic up/down)
Typical Values	`EMPTY`, `BIOMES`, …, `FULL`	`ENTITY_TICKING`, `BLOCK_TICKING`, `FULL`, `INACCESSIBLE`

One-sentence summary: **ChunkStatus manages "how much generated", ChunkLevelType manages "what can currently do".**

A chunk can be in `ChunkStatus = FULL` (generation complete) while simultaneously in `ChunkLevelType = INACCESSIBLE` (not accessible) — this is the precise source code explanation for "distant terrain generated but not ticking".

2 Loading Level (ChunkLevel)

Loading level is an **integer** that determines a chunk's loading or generation state. Key rules:

- Value range is 44 and below (usually not lower than 11);
- **Lower values mean the chunk is loaded more "thoroughly"**;
- Relationship between loading level and loading distance: `Loading level = 33 - Loading distance`.

For example, a ticket with `radius = 2` gives the center chunk loading level `33 - 2 = 31` (strong loading), and each step outward in Chebyshev distance adds +1 to loading level.

> The upper limit for loading level is **44**. `INACCESSIBLE = 33 + ChunkStatus.getMaxDistanceFromFull()`. In 1.20.1, `getMaxDistanceFromFull()` returns 11, so `INACCESSIBLE = 33 + 11 = 44`. The highest loading level for inaccessible chunks is 44.

3 ChunkLevelType: Four Types of Runtime Eligibility

`ChunkLevels.getType(level)` maps loading level to four **`ChunkLevelType`** values:

```java
// Judgment rules
public static ChunkLevelType getType(int level) {
    if (level <= 31) return ChunkLevelType.ENTITY_TICKING;    // Strong loading
    else if (level <= 32) return ChunkLevelType.BLOCK_TICKING; // Weak loading
    else return level <= 33 ? ChunkLevelType.FULL : ChunkLevelType.INACCESSIBLE;
}
```

Precise meaning of the four types:

CHUNKLEVELTYPE	CORRESPONDING LEVEL	BLOCK STATE ACCESSIBLE	BLOCK TICKS/REDSTONE	ENTITY TICKING	MOB SPAWNING	CLIENT RENDERING
`ENTITY_TICKING`	⩽ 31	☒	☒	☒	☒	☒
`BLOCK_TICKING`	32	☒	☒	☒	☒	☒
`FULL`	33	☒	☒	☒	☒	☒
`INACCESSIBLE`	⩾ 34	☒	☒	☒	☒	☒

Explained one by one:

3.1 ENTITY_TICKING (Strong Loading, ⩽31)

Most complete runtime state. Almost all computational logic proceeds normally within the chunk — block ticks (including redstone operations and random ticks), entity AI and movement, mob spawning checks, block entity updates. "Chunks around the player" are typically in this state.

3.2 BLOCK_TICKING (Weak Loading, 32)

The **only difference from strong loading is entity ticking is suspended.** Redstone devices, fluids, scheduled ticks, etc. still run normally. In `World#setBlockState`, you can see the server only triggers client synchronization when the current chunk reaches `BLOCK_TICKING` level. Weak loading chunks are still rendered on the client.

3.3 FULL (Border Loading, 33)

Chunk data is **in memory for reading and writing**, but almost no active operations are executed. Block states can be queried, entities can be recorded (count toward mob cap), but there are no block ticks, redstone operations, or entity movement. The client doesn't render chunks at this level.

This is the "transition zone" — the chunk is loaded enough for nearby chunks to reference its state, but not enough to waste CPU on running its own operations.

3.4 INACCESSIBLE (Unloaded, $\geqslant$34)

Not in active memory. May be waiting to be generated, or may be queued for unloading. Attempting to access block states will return null or air. Cannot be accessed by normal game logic.

> [!NOTE]
> Why is it called "INACCESSIBLE" rather than "UNLOADED"?
> Because chunks at this level may not have been completely unloaded — they're just "not yet available" or "about to be unavailable". `UNLOADED` would imply the memory state has been cleared, but the ticket system only decides "access eligibility", not directly managing memory release.

4 Level Transition: Two-Way Upgrade/Downgrade

Unlike `ChunkStatus` (which advances in one direction during generation), `ChunkLevelType` **changes dynamically bidirectionally**:

- Tickets added → loading level decreases → type may upgrade (`FULL` → `BLOCK_TICKING` → `ENTITY_TICKING`);
- Tickets removed → loading level increases → type may downgrade (`ENTITY_TICKING` → `BLOCK_TICKING` → `FULL` → `INACCESSIBLE`).

Upgrade and downgrade operations in `ChunkHolder`:

```java
// Upgrade: Request higher status
private CompletableFuture<Either<Chunk, Unloaded>> createUpdateFuture(
    ChunkStatus targetStatus, ...) {
    // If current status < target status → schedule generation task
}
// Downgrade: Cancel unreachable status
public void tick() {
    // If new level < old level → set unnecessary Futures to UNLOADED
}
```

Two points deserve attention:

Upgrade is asynchronous (via CompletableFuture): Because generation requires computation time, upgrade tasks are scheduled into the task queue for execution in a future async phase.

Downgrade is synchronous (immediately invalidate Futures): Because downgrade is just "declaring unavailable", no actual data needs to be cleared — just mark corresponding Futures as complete with `UNLOADED_WORLD_CHUNK`, notifying all waiters "this Future is now invalid".

5 Dimension-Level Idle Timeout

Beyond chunk-level `ChunkLevelType` control, there's a **dimension-level** global switch: **idle timeout**.

When a dimension contains **no players** and **no forced chunks**, `ServerWorld` starts a timer. Even if individual chunks have loading levels that permit entity ticking (level $\leqslant 31$), once the dimension idle timeout reaches its threshold, all entity and block entity ticking in that dimension is skipped.

5.1 Mechanism Implementation

In `ServerWorld`'s main tick loop:

```java
// ServerWorld.java lines 369-374
boolean bl = !this.players.isEmpty() || !this.getForcedChunks().isEmpty();
if (bl) {
    this.resetIdleTimeout();
}
if (bl || this.idleTimeout++ < 300) {
    profiler.push("entities");
    // entity ticking loop
    profiler.pop();
    this.tickBlockEntities();
}
```

Key field (line 180):

```java
private int idleTimeout;
```

Logic flow:

1. **Determine activity**: `hasActivity = !players.isEmpty() || !getForcedChunks().isEmpty()`
2. **Has activity** → `resetIdleTimeout()` resets `idleTimeout` to zero
3. **No activity** → `idleTimeout++` increments every tick
4. **Threshold check**: when `idleTimeout >= 300`, skip entity loop and block entity ticking

5.2 Behavior Timeline

TIME PERIOD	IDLETIMEOUT VALUE	ENTITY TICKING	BLOCK ENTITY TICKING	BLOCK TICKS (SCHEDULED/RANDOM)
Players present / has forceload	0 (reset)	☒	☒	☒
0-299 gt after players leave	0 → 299	☒	☒	☒

TIME PERIOD	IDLETIMEOUT VALUE	ENTITY TICKING	BLOCK ENTITY TICKING	BLOCK TICKS (SCHEDULED/RANDOM)
300+ gt after	300+	☒	☒	☒
Players return / forceload activates	0 (reset)	☒ immediate resume	☒ immediate resume	☒

> Idle timeout **only skips entity and block entity ticking**, and does not affect block ticks. Even when `idleTimeout >= 300`, scheduled ticks (`blockTickScheduler.tick()`) and random ticks still execute normally for chunks at `BLOCK_TICKING` level.

5.3 Orthogonal Relationship with Chunk-Level

Idle timeout is a **dimension-level global check**, independent from chunk-level `ChunkLevelType`:

- If a chunk's level > 31 (not `ENTITY_TICKING`), entities don't tick anyway: idle timeout doesn't affect it.
- If a chunk's level ⩽ 31, but the dimension idle timeout reaches 300 gt, entities still don't tick, even though the chunk is "eligible".

In other words, entity ticking requires **both conditions to be satisfied**:

1. Chunk-level: `ChunkLevelType = ENTITY_TICKING` (level ⩽ 31)
2. Dimension-level: `idleTimeout < 300`

If either condition fails, entities won't tick.

5.4 Design Intent

The 300 gt (15 seconds) buffer period is a classic performance optimization case:

- **Goal**: Dimensions without players don't need continuous entity AI computation. Villager pathfinding, mob movement, projectile flight are all high-cost operations.
- **Balance**: The 300 gt delay balances "saving performance" with "seamless user experience". When a player teleports to the Nether, lights a portal, passes through, and returns, the entire process takes less than 15 seconds, so entities remain ticking. But dimensions empty for extended periods will halt entity ticking, conserving server resources.

Worth noting: **PORTAL loading tickets also have a 300 gt lifetime** (see Chapter 6). These two 300s aren't the same timer, but they coincide in cross-dimension teleportation scenarios: when a player leaves a dimension through a portal, the PORTAL ticket expires after 300 gt, and the dimension idle timeout is also 300 gt. Both mechanisms layer together to ensure chunks and entities near portals keep ticking briefly after teleportation, but don't persist indefinitely.

6 Common Misconceptions

MISCONCEPTION	REALITY
"If the client can see it, the server is ticking it"	Client rendering is controlled by view distance, while server ticking is controlled by simulation distance. They are independent parameters.
"FULL means random ticks run"	`FULL` only guarantees that data is accessible. Random ticks require `BLOCK_TICKING` (`level <= 32`).
"Loaded means mobs can spawn"	Mob spawning requires `ENTITY_TICKING` (`level <= 31`).
"Near the player means spawning definitely happens"	Spawning still requires additional conditions such as light, space, and mob cap. `ENTITY_TICKING` is necessary, but not sufficient.
"Force-loading means everything will run"	`/forceload` achieves `ENTITY_TICKING` via a `FORCED` ticket, but actual operations are still constrained by gamerules, chunk state, and other conditions.

7 Summary

- `ChunkStatus` governs "generation progress," while `ChunkLevelType` governs "runtime eligibility." They answer different questions.
- The four runtime eligibility states progress by level as `INACCESSIBLE` → `FULL` → `BLOCK_TICKING` → `ENTITY_TICKING`.
- Strong loading (`<=31`) allows all operations, weak loading (`32`) suspends entities, border loading (`33`) allows queries only, and inaccessible (`>=34`) is externally invisible.
- The essence of "generated but no longer ticking" is `ChunkStatus = FULL` while `ChunkLevelType != ENTITY_TICKING`.

8 Code Walkthrough

8.1 ChunkLevels Threshold Design: Why 31/32/33?

```java
// ChunkLevels.java
private static final int ENTITY_TICKING = 31;
private static final int BLOCK_TICKING = 32;
private static final int FULL = 33;
public static final int INACCESSIBLE = 33 + ChunkStatus.getMaxDistanceFromFull(); // = 44
public static ChunkLevelType getType(int level) {
    if (level <= 31)      return ChunkLevelType.ENTITY_TICKING;
    else if (level <= 32) return ChunkLevelType.BLOCK_TICKING;
    else                  return level <= 33
        ? ChunkLevelType.FULL : ChunkLevelType.INACCESSIBLE;
}
```

These three thresholds were not chosen arbitrarily. Their design is based on the ticket propagation model: a loading ticket has `level = 33 - radius`, and each chunk step outward adds +1 to the level. Therefore:

- `radius >= 2` → level `<= 31`: chunks within Chebyshev distance 2 of the player run **both entities and blocks**. This is the most important gameplay area, where mobs the player can directly see, redstone contraptions, and flowing water all run.
- `radius = 1` → level `= 32`: the ring one step farther out. **Blocks still run** there, including redstone, fluids, and scheduled ticks, but entities are suspended. The design intent of this ring is to preserve visual continuity while saving the CPU cost of entity simulation.
- `radius = 0` → level `= 33`: the outermost "border layer." Data is in memory, but **no active computation runs**. This layer exists so that when the player moves, chunks newly entering strong-loading range do not need to be regenerated. They are already prepared in memory in the `FULL` state.

Why is `BLOCK_TICKING` 32 rather than some other value? Notice that 32 = 33 - 1. It corresponds exactly to `radius = 1`, meaning chunks at Chebyshev distance 1 from the player. If it were 31, then even the `radius = 1` chunks would need strong loading, which would mean all entities in a 5×5 area around the player would be moving, at too high a cost. If it were 33, then `radius = 1` and `radius = 0` would be no different, meaning distant water would stop flowing and the visual effect would break.

8.2 Enum Order Determines "Priority"

`ChunkLevelType` is declared as a standard Java enum:

```java
public enum ChunkLevelType {
    INACCESSIBLE,    // ordinal() = 0
    FULL,            // ordinal() = 1
    BLOCK_TICKING,   // ordinal() = 2
    ENTITY_TICKING;  // ordinal() = 3
    public boolean isAfter(ChunkLevelType chunkLevelType) {
        return this.ordinal() >= chunkLevelType.ordinal();
    }
}
```

`isAfter` semantically means "current level **at least meets** the requirement of parameter level". Because the declaration order is progressive (INACCESSIBLE → FULL → BLOCK_TICKING → ENTITY_TICKING), `ordinal()` directly reflects "strength of computational ability".

Here's an easily confused point: `isAfter` is **not** comparing "size of level numbers" (that's already done in `ChunkLevels`), but comparing "containment relationship of computational eligibility". `ENTITY_TICKING.isAfter(BLOCK_TICKING)` is true, because strong loading chunks automatically meet all conditions of weak loading. `FULL.isAfter(BLOCK_TICKING)` is false, because "can only read" doesn't meet the requirement of "must run block ticks".

8.3 Why fill with UNLOADED rather than null during downgrade

When ChunkHolder downgrades, it needs to clear generation status Futures that are no longer needed:

```java
// ChunkHolder.tick()
```

```
if (bl) {  // old level was accessible
    Either<Chunk, Unloaded> either = Either.right(new Unloaded() { ... });
    for (int i = bl2 ? chunkStatus2.getIndex() + 1 : 0;
         i <= chunkStatus.getIndex(); i++) {
        if (this.futuresByStatus.get(i) == null) {
            this.futuresByStatus.set(i,
                CompletableFuture.completedFuture(either));
        }
    }
}
```

The loop runs from `0` to `old ChunkStatus.index`: if still accessible after downgrade (`bl2=true`), keep Futures for new level's corresponding status and above (start clearing from `new ChunkStatus.index + 1`); if no longer accessible after downgrade (`bl2=false`), clear everything starting from `0`.

Why use `CompletableFuture.completedFuture(either)` instead of `set(i, null)`? Because external code may have already obtained this Future through `getFutureFor(someStatus)` and registered callbacks (`.thenAccept()`). If set directly to `null`, those callbacks will wait forever, causing memory leaks. Setting to an already-completed Future (value is `Unloaded`) immediately notifies all waiters of "unavailable" and cleans up callbacks — this is the **graceful termination pattern** of `CompletableFuture`.

Note that `futuresByStatus` uses `AtomicReferenceArray` rather than a normal array — because generation threads and the main thread may access it simultaneously. `AtomicReferenceArray` provides lock-free concurrent safety guarantees, at the cost of one extra CAS operation overhead per element.

8.4 Dimension Idle Timeout: Why a Counter Instead of a Boolean?

In `ServerWorld`, idle timeout is implemented as an **integer counter**, not a simple boolean flag:

JAVA

```
// ServerWorld.java line 180
private int idleTimeout;
// ServerWorld.java lines 735-737
public void resetIdleTimeout() {
    this.idleTimeout = 0;
}
```

Why not design it as `boolean isIdle`? Because **the buffer period requires a counter**.

If a boolean flag were used, the state would only have two values: "idle" and "not idle". As soon as a player leaves the dimension, `isIdle` would immediately flip to `true`, halting entity ticking instantly. This would create a jarring user experience: a player briefly teleporting to the Nether and immediately returning to the Overworld would find all mobs and villagers frozen in place, pistons stuck mid-action. This doesn't match player intuition.

The `int idleTimeout` design allows the system to express "how long until truly idle":

- Player just left: `idleTimeout = 0`
- 1 tick later: `idleTimeout = 1`
- 299 ticks later: `idleTimeout = 299`, entities still ticking
- 300 ticks later: `idleTimeout = 300`, skip entity ticking

- Player returns: `idleTimeout` resets to zero, immediate resume

This **gradual transition** design is more aligned with real-world time progression than a binary switch. "Idle" isn't an abrupt state change, but an accumulated process.

8.5 Why Doesn't the Idle Timeout Control Block Ticks?

In `ServerWorld.tick()`, block tick scheduling logic executes **before the idle timeout check**:

```java
// ServerWorld.java (simplified)
public void tick(BooleanSupplier shouldKeepTicking) {
    // ...
    this.blockTickScheduler.tick(tickTime, 65536, this::tickBlock);
    this.fluidTickScheduler.tick(tickTime, 65536, this::tickFluid);
    boolean bl = !this.players.isEmpty() || !this.getForcedChunks().isEmpty();
    if (bl) {
        this.resetIdleTimeout();
    }
    if (bl || this.idleTimeout++ < 300) {
        // entity and block entity ticking here
    }
}
```

Note the order:

1. **Block ticks execute first** (`blockTickScheduler.tick()` / `fluidTickScheduler.tick()`)
2. **Then check idle timeout** (`if (bl || this.idleTimeout++ < 300)`)
3. **Finally decide whether to skip entity ticking**

Why aren't block ticks affected by idle timeout? Two design reasons:

1. **Determinism vs randomness**: Block ticks (redstone, fluids, crop growth) are **deterministic operations**. Given the same input, output is always identical. Even when unobserved, redstone repeaters should propagate signals on schedule, water should spread at fixed speed. Entity AI (villager pathfinding, mob tracking) is **highly random and path-dependent**. Continuing to compute these operations without players wastes CPU, and results can't be observed by users anyway.
2. **Performance cost ratio**: Entity ticking involves heavy collision detection, pathfinding, behavior tree evaluation. CPU cost far exceeds block ticks. Block tick scheduling uses a **heap structure** (`OrderedTick`), processing only the small number of expired entries each tick, so cost is manageable. Skipping entity ticking saves 70%+ of dimension tick overhead, skipping block ticks would only save another 10%. Disproportionate benefit.

Therefore, idle timeout **only targets entities and block entities**, while block ticks execute normally. Redstone contraptions in empty dimensions keep working, but villagers and mobs stop.

8.6 Post-Increment Semantics of idleTimeout++

In `if (bl || this.idleTimeout++ < 300)`, `idleTimeout++` is **post-increment**: it uses the current value for comparison, then increments.

> This means:
>
> - First tick without players: `idleTimeout` value is 0, expression `0 < 300` is true, entities tick normally, **then** `idleTimeout` becomes 1.
> - Second tick without players: `idleTimeout` value is 1, expression `1 < 300` is true, **then** becomes 2.
> - ...
> - 300th tick without players: `idleTimeout` value is 299, expression `299 < 300` is true, **then** becomes 300.
> - 301st tick without players: `idleTimeout` value is 300, expression `300 < 300` is **false**, skip entity ticking.
>
> If pre-increment were used (`++this.idleTimeout < 300`), the 300th tick's evaluation would become `300 < 300 = false`, giving an actual buffer period of only 299 gt, off by 1 tick.
>
> Post-increment guarantees **exactly 300 gt of buffer**: during the 300 ticks from `idleTimeout=0` to `idleTimeout=299`, entities tick normally. From `idleTimeout=300` onward, entities stop. This "precise threshold alignment" semantic is a common pattern in Mojang's time-sensitive logic (like ticket `expiryTicks--`, also a count-to-zero design).

9 References

- Discovering Minecraft - Basic Concepts in the Chunk Management System (CC0 License)
- `net.minecraft.server.world.ChunkLevels`
- `net.minecraft.server.world.ChunkLevelType`
- `net.minecraft.server.world.ChunkHolder`
- `net.minecraft.server.world.ChunkTicketManager`
- `net.minecraft.server.world.ServerWorld`

5.4 Server Master Dispatcher TACS

If the chunk management system is a factory, `ThreadedAnvilChunkStorage` (hereafter **TACS**) is the workshop director — it doesn't directly produce products, but coordinates every production line's operation.

1 Name Breakdown

The name `ThreadedAnvilChunkStorage` itself explains its responsibilities:

- **Threaded**: Threaded — chunk generation runs on independent threads;
- **Anvil**: Anvil — refers to the modern Minecraft Anvil chunk storage format (`.mca`);
- **ChunkStorage**: Chunk Storage — manages chunk persistence.

Together: "**Threaded Anvil chunk storage manager**". Though multithreading use is actually quite limited — mainly world generation and NBT access, with the latter not even fully managed in TACS.

2 Responsibilities Overview

TACS sits at the bottom coordination position in the chunk management system, connecting to almost all other subsystems:

- **Manages chunks in `ChunkHolder` form**: Including `ChunkHolder` creation and unloading, jointly managing their loading levels and states.
- **Defines and schedules chunk loading, generation, and light update tasks**: Through `CompletableFuture`, tasks are made asynchronous.
- **Defines logic for sending chunk data to players**: Methods like `sendWatchPackets()`, `sendChunkDataPackets()`, etc.
- **Manages entity data tracking**: Data synchronization with clients, `EntityTracker` manages entity position tracking.
- **Handles player addition and removal**: `handlePlayerAddedOrRemoved()` and `updatePosition()`.

3 Core Table Structures

TACS maintains multiple mapping tables, each with a clear purpose:

3.1 currentChunkHolders

```java
Long2ObjectMap<ChunkHolder> currentChunkHolders
```

All chunks with **existing ChunkHolders** in the current dimension. Key is `ChunkPos.toLong()` serialized `long`, value is the corresponding `ChunkHolder`.

A `ChunkHolder` is placed in this table when created (when the chunk first enters accessible range); removed from this table when unloaded.

3.2 chunkHolders

```java
// ThreadedAnvilChunkStorage.java (Yarn 1.20.1)
private final Long2ObjectLinkedOpenHashMap<ChunkHolder> currentChunkHolders = new
Long2ObjectLinkedOpenHashMap<>();
private volatile Long2ObjectLinkedOpenHashMap<ChunkHolder> chunkHolders =
this.currentChunkHolders.clone();
```

`ThreadedAnvilChunkStorage` does have a field named `chunkHolders` — it's a **volatile snapshot copy** of `currentChunkHolders`, used for scenarios like `tick()` and `save()` that need stable traversal. Updates happen on the main thread (Minecraft Server), while the read side can safely traverse without locks, avoiding `ConcurrentModificationException`. This is a **Copy-On-Write** pattern: write side updates on main thread, read side has zero overhead.

> [!NOTE] Difference from `ChunkTicketManager.chunkHolders`
>
> - `TACS.chunkHolders`: `Long2ObjectLinkedOpenHashMap<ChunkHolder>`, complete mapping table of **all** active `ChunkHolder` in current dimension, key is `ChunkPos.toLong()` serialized `long`.
> - `ChunkTicketManager.chunkHolders`: `Set<ChunkHolder>`, only collects `ChunkHolder` whose **loading levels changed** in this tick round, used for subsequent notification callbacks.

3.3 chunksToUnload

```java
LongSet chunksToUnload
```

Records `long` coordinates of chunks about to be unloaded. When a `ChunkHolder`'s loading level drops from accessible to inaccessible, it moves from `currentChunkHolders` into `chunksToUnload`, waiting for an appropriate time to actually unload.

3.4 loadedChunks

```java
// Indirectly managed via VersionedChunkStorage
```

Set of chunks loaded from disk or finished generating, available for access. "Loaded" here means "data in memory", not "currently ticking".

3.5 chunkToType

```java
Long2ByteMap chunkToType
```

Records each chunk's `ChunkLevelType`. Key is chunk pos long, value is `ChunkLevelType.ordinal()` (byte). Used for quickly checking "is this chunk in ENTITY_TICKING state without having to find its ChunkHolder?"

This is a performance-optimized redundant cache — `ChunkHolder` already stores level type, but looking it up requires one more hash table query. `chunkToType` allows direct querying during high-frequency operations (like entity movement collision detection).

3.6 unloadedTaskQueue

```java
Queue<Runnable> unloadedTaskQueue
```

Stores tasks (usually save and cleanup) that need to be executed after a chunk unloads. When a chunk is completely removed from memory, tasks in the queue execute sequentially — typical use: `saveChunk()`, `removeFromTracking()`.

4 Three Executors

TACS delegates different types of tasks to three executors:

4.1 mainThreadExecutor

```java
private final Executor mainThreadExecutor;
```

Main thread executor. All operations that modify world state (placing/breaking blocks, entity movement, chunk state changes) must run here. Guarantees serialized execution, avoiding concurrency issues.

4.2 worldGenExecutor

```java
private final Executor worldGenExecutor;
```

World generation thread pool. Chunk generation tasks (like `NOISE`, `FEATURES`, `CARVERS` stages) run here. Multiple chunks can generate in parallel, but each chunk's generation is still single-threaded (same chunk's different stages execute sequentially).

4.3 lightingExecutor

```java
private final Executor lightingExecutor;
```

Lighting calculation thread pool. Light propagation calculations (like `LIGHT` stage) run here. Separated from generation threads because lighting depends on neighboring chunks — dedicated threads avoid blocking generation tasks.

> These three executors **don't directly execute tasks** — they're wrappers for the underlying `ChunkTaskPrioritySystem`, ensuring tasks enter a priority queue first and are scheduled uniformly, rather than executing immediately.

5 ChunkTaskPrioritySystem: Priority Scheduling

All asynchronous tasks are coordinated by `ChunkTaskPrioritySystem` before execution. It's not a simple FIFO queue, but a **priority queue** — priority determined by **loading level**:

```java
// ChunkTaskPrioritySystem.java
public static Task<Runnable> createMessage(ChunkHolder holder, Runnable task) {
    // Task priority determined by holder.getCompletedLevel()
    // Lower completedLevel (stronger loading) → higher priority
    return createMessage(task, holder.getPos().toLong(), holder::getCompletedLevel);
}
```

Design principle: **Chunks closer to completion, closer to target level, get higher priority**. This avoids "a level=22 chunk with EMPTY progress" taking priority over "a level=33 chunk that only needs disk loading" — the latter might finish in 1 tick.

> Why use `completedLevel` rather than `level` to determine priority?
> `level` is the "target" (what level we want this chunk to reach), `completedLevel` is the "status quo" (what level it's actually reached). Sorting by `completedLevel` means: **tasks already largely complete, closest to target, get highest priority**.

6 Code Walkthrough

6.1 TACS Lifecycle and Chunk Holder Creation

TACS is created when `ServerWorld` initializes:

```java
// ServerWorld.java
this.chunkManager = new ServerChunkManager(
    this, session, dataFixer, structureManager, executor,
    chunkGenerator, viewDistance, simulationDistance, ...);
// ServerChunkManager constructor creates TACS internally
```

When a chunk holder is needed:

```java
// ThreadedAnvilChunkStorage.setLevel()
ChunkHolder holder = this.currentChunkHolders.get(pos);
if (holder == null) {
    // Chunk doesn't exist → create new holder
    holder = new ChunkHolder(pos, level, ...);
    this.currentChunkHolders.put(pos, holder);
}
// Update level and schedule generation/loading tasks
```

Key decision: **ChunkHolder is created lazily** — only when a ticket requires it. Chunks outside any ticket range don't have holders, saving memory.

6.2 Why Use Volatile Snapshot Copy

```java
private volatile Long2ObjectLinkedOpenHashMap<ChunkHolder> chunkHolders =
    this.currentChunkHolders.clone();
public void tick() {
    // Clone before traversal
    this.chunkHolders = this.currentChunkHolders.clone();
    // Traverse cloned copy
    for (ChunkHolder holder : this.chunkHolders.values()) {
        holder.tick();
    }
}
```

Why clone? `tick()` may indirectly modify `currentChunkHolders` (creating new holders or removing them). If directly traversing `currentChunkHolders`, `ConcurrentModificationException` would result. Cloning creates a snapshot — the original may change during traversal, but the copy remains stable.

Volatile ensures visibility: Main thread updates `currentChunkHolders`, other threads (like save threads) read `chunkHolders`. `volatile` guarantees visibility — once main thread updates the snapshot, other threads immediately see the new value.

This pattern is typical **Copy-On-Write**: only pay cloning cost when iteration is needed, read side has zero overhead.

6.3 setViewDistance: Client Data Resend When View Distance Changes

```java
protected void setViewDistance(int watchDistance) {
    int i = MathHelper.clamp(watchDistance, 2, 32);  // Hard limit 2~32
    if (i != this.watchDistance) {
        int oldDist = this.watchDistance;
        this.watchDistance = i;
        this.ticketManager.setWatchDistance(i);
        // Traverse all existing ChunkHolders
        for (ChunkHolder holder : this.currentChunkHolders.values()) {
            ChunkPos pos = holder.getPos();
            // For each player watching this chunk
            this.getPlayersWatchingChunk(pos, false).forEach(player -> {
                // Check if chunk is in player view under old/new distance
                boolean wasInRange = isWithinDistance(pos, playerPos, oldDist);
                boolean nowInRange = isWithinDistance(pos, playerPos, i);
                // Only send/cancel packets when boundary changes
                if (wasInRange != nowInRange) {
                    this.sendWatchPackets(player, pos, ...);
                }
            });
        }
    }
}
```

Note the logic here: **Only send packets when a chunk's "entering view/leaving view" state changes.** If a chunk is in view under both old and new distances, no resend — avoids network storms when players adjust view distance. But if a chunk first enters view under the new distance, the entire

> `ChunkDataS2CPacket` is sent to the client.

7 References

- Discovering Minecraft - Basic Architecture of the Chunk Management System (CC0 License)
- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.server.world.ChunkTaskPrioritySystem`
- `net.minecraft.server.world.ServerChunkManager`

5.5 Main Thread Async Tasks

> Adapted from lovexyn0827's *1.14+ Chunk Management System and Dynamic Analysis* Chapter 4 (CC0 License).

If you've understood the architecture from previous chapters by the time you reach this one, you might ask: "`ChunkTicketManager.tick()` updated the loading level and notified `ChunkHolder`, so when is the chunk actually loaded?"

The answer is: **not "immediately", but "executed in some future async task"**.

1 CompletableFuture's Core Role

In the chunk management system, `CompletableFuture` is the pervasive async coordination tool. It has two roles:

1.1 Role One: Task Planning and Execution

A `CompletableFuture` corresponds to a basic task — updating player loading tickets, loading a chunk, completing one generation stage of a chunk. Tasks aren't run directly, but are **scheduled** as a `CompletableFuture`, then **executed** somewhere later in the operation.

This "planning-execution" separation means: you might add a task at the start of a game tick (gt), but it won't actually run until the async task execution phase several game ticks later.

1.2 Role Two: Chunk Availability Marker

In `ChunkHolder`, three `CompletableFuture`s serve as availability markers for different "views" of chunks:

- `accessibleFuture`: Completes when chunk data is accessible
- `tickingFuture`: Completes when chunk can participate in block ticking
- `entityTickingFuture`: Completes when chunk can participate in entity operations

When a chunk no longer meets a certain level requirement, the corresponding Future is `complete(UNLOADED_WORLD_CHUNK)`, and any operations depending on that view receive a "not available" result.

2 Task Queue: MainThreadExecutor

Async tasks are managed by `ServerChunkManager.MainThreadExecutor`. It's essentially a **first-in-first-out (FIFO)** task queue, ensuring all tasks execute on the server main thread.

Each task entering the queue will eventually be taken out and executed on the main thread, but the timing is not fixed — see the next section.

3 Execution Timing

Locations that may trigger async task execution are scattered throughout the code. Three main categories:

3.1 Category One: Blocking Execution During Chunk Save

During chunk saves triggered by pause, exit, autosave, or `/save-all` command, saving a single chunk requires ensuring that chunk's data is ready. At this point, related async tasks are **executed blockingly and immediately** — can't continue saving without finishing execution.

3.2 Category Two: Need-Based Execution Triggered by getChunk()

This is the most common and most hidden category. When external code obtains a chunk through `ServerChunkManager.getChunk()`:

1. A task T for "get this chunk" is added to the async task queue;
2. Tasks in the queue are executed in FIFO order **until T completes**;
3. After T completes, task execution **immediately stops** — tasks queued after T **won't be executed here**.

This means: if 50 other tasks were already queued before you called `getChunk()`, those 50 tasks will be executed before your T. Conversely, tasks after T can only wait for the next execution opportunity.

Locations that may trigger `getChunk()` number at least in the **hundreds** — including (but not limited to):

- When using `World.getBlockState()` / `setBlockState()`
- When using `World.getBlockEntity()` / `setBlockEntity()`
- When using `World.getFluid()`
- Player respawn, entering dimension, joining dimension
- Getting region difficulty
- Nether portal teleporting entities

> Getting entities in a region generally doesn't cause related chunks to load — entity queries and block queries are two independent mechanisms in the source code.

3.3 Category Three: Fallback Execution at Game Tick End

At the end of each server game tick is an async task operation phase. In this phase:

- If not lagging (current gt runtime $\leqslant$ 50ms), task execution can continue until task queues for **all dimensions** are emptied;
- If lagging, task execution terminates immediately, remaining tasks left for next gt;
- New tasks produced during execution will also continue to be executed in this phase.

Execution order is fixed:

1. Server's own async tasks (like processing operations from client)
2. Main world's `ServerChunkManager` async tasks

3. Other dimensions' `ServerChunkManager` async tasks (Nether before End)

4 Dynamic Understanding

Design intent behind the three execution timing categories can be understood this way:

- **Category One** is to complete a task that must be completed blockingly immediately ("if we don't save this chunk, the server can't safely shut down");
- **Category Two** is to complete a task the caller must immediately receive ("`getBlockState()` can't return a real state, subsequent logic can't proceed");
- **Category Three** is a fallback mechanism — ensures all async tasks produced in the current gt are processed within the current gt under normal circumstances.

But precisely because execution timing is scattered across hundreds of call points, **async task queues for the same dimension can be partially executed anywhere** — this makes chunk loading timing highly unpredictable dynamically. This is also why lovexyn0827's original text mentioned "this part is really written very 'awesome'".

5 Summary

- `CompletableFuture` plays a dual role in chunk management: task carrier and chunk availability marker.
- Async tasks are managed by `MainThreadExecutor`, FIFO queue, guaranteed main thread execution.
- Execution timing falls into three categories: blocking execution during save, need-based execution from `getChunk()`, fallback execution at gt end.
- Category Two execution timing first executes all tasks queued before the target task, but doesn't execute subsequent tasks.
- Precisely because execution timing is scattered, "which operations cause chunk loading" is a question requiring analysis combined with specific call chains.

6 Code Walkthrough

6.1 Why Async Task Mechanism Is Needed

Assume there's no async task mechanism — every time `World.getBlockState()` is called, if the target chunk isn't in memory, **synchronously** wait for it to finish generating. This means the entire server main thread would block, unable to maintain 20 TPS.

Minecraft uses `CompletableFuture` to solve this problem. Chunk generation and loading don't execute directly, but are wrapped as a `CompletableFuture` and submitted to `ChunkTaskPrioritySystem`'s queue:

```java
// ChunkTaskPrioritySystem.java
public static Task<Runnable> createMessage(ChunkHolder holder, Runnable task) {
    // Task priority determined by holder.getCompletedLevel()
    // Lower (more "strong" loading) completedLevel = higher task priority
```

```
    return createMessage(task, holder.getPos().toLong(), holder::getCompletedLevel);
}
```

`ChunkTaskPrioritySystem` internally maintains a `LevelPrioritizedQueue` — not a simple FIFO queue, but sorted by **level priority**.

Design Decision — Why is priority determined by `completedLevel` rather than `level`? `level` is the "target" (what level we want this chunk to reach), `completedLevel` is "current state" (what level actually reached). Sorting by `completedLevel` means: **tasks already largely complete and closest to target have highest priority**. This avoids "a chunk with level=22 but generation progress EMPTY" jumping ahead of "a chunk with level=33 but only needs disk read" — the latter might complete in 1 tick.

6.2 Design Logic of Three Execution Timings

```java
// ServerChunkManager.getChunk() simplified flow
public Chunk getChunk(int x, int z, ChunkStatus leastStatus, boolean create) {
    // 1. If cache hit (last 4 requested chunks), return directly
    // 2. Cache miss → create UNKNOWN loading ticket + schedule async task
    // 3. Call tick() to execute queue's async tasks until target chunk task completes
    // 4. Return result
}
```

Design intent for three timings:

Category One (blocking execution during save): "If we don't save this chunk before closing server, data will be lost". This is the only scenario that can block the main thread, because user has already initiated server shutdown, safety priority over performance.

Category Two (getChunk triggered): "If we don't load this chunk, caller will get air blocks, all subsequent logic breaks". Caller depends on return value being correct `BlockState`, so must **semi-block** here — execute queue until target completes, but stop after completion. Stopping reason is not wanting to "incidentally" help other chunks complete tasks, extending this call's latency.

Category Three (gt end fallback): "These tasks should have been incidentally executed in Category Two, but didn't get their turn. Now gt is almost over, if there's time handle them." This is fallback mechanism — ensures all tasks complete within current gt under normal non-lag. If lagging (current gt exceeds 50ms), incomplete tasks left for next gt — hard real-time constraints priority over completeness.

6.3 The "Partial Execution" Problem

Category Two execution timing's "execute until target completes then stop" causes a counter-intuitive phenomenon: **If you call getChunk() to get chunk A, and someone happened to call getChunk() for chunk B earlier with B's task still in queue, B's task will execute first (because it's ahead of A in queue), then comes A**. But tasks after B (if any) won't execute — because execution stops after A completes.

This is what lovexyn0827's original text calls "very awesome": async task execution order depends on call order, and call order depends on hundreds of `getBlockState()`, `setBlockState()`, `getBlockEntity()` call sites — these sites are scattered across mob AI, redstone operations, block updates, entity collision

> detection, and various corners. **Two completely identical player actions may lead to completely different task execution orders due to tiny server timing differences**.

7 References

- Discovering Minecraft - Main Thread Async Tasks and Execution Timing (CC0 License)
- `net.minecraft.server.world.ServerChunkManager.MainThreadExecutor`
- `net.minecraft.server.world.ServerChunkManager.getChunk()`

5.6 ChunkHolder Lifecycle

If the ticket system is the "brain" (deciding what to do) and TACS is the "arm" (coordinating departments), then `ChunkHolder` is the "thermometer" — it measures and responds to a chunk's current "heat," then precisely controls which operational state the chunk is in.

1 ChunkHolder's Responsibilities

We've mentioned `ChunkHolder` many times in previous chapters. It is the **internal representation** (or container) of a chunk within the chunk management system. A separate `Holder` object is needed because:

- Chunks at loading range boundaries **may not exist in memory**, but storing their load level is still necessary;
- A chunk's load level and generation state **should be invisible to external code** — they are internal management information;
- The availability of chunks under different operational views **is managed uniformly by Holder**, avoiding confusion about "is this chunk usable right now?";
- Facilitates atomic operations on individual chunks like setting load level, triggering loading, and scheduling unloading.

2 Three Core Futures

`ChunkHolder` uses three `CompletableFuture` objects to manage a chunk's three "operational views":

```java
// ChunkHolder.java
private volatile CompletableFuture<Either<WorldChunk, Unloaded>> accessibleFuture
    = UNLOADED_WORLD_CHUNK_FUTURE;
private volatile CompletableFuture<Either<WorldChunk, Unloaded>> tickingFuture
    = UNLOADED_WORLD_CHUNK_FUTURE;
private volatile CompletableFuture<Either<WorldChunk, Unloaded>> entityTickingFuture
    = UNLOADED_WORLD_CHUNK_FUTURE;
```

FUTURE	MEANING	WHEN COMPLETED	CORRESPONDING CHUNKLEVELTYPE
`accessibleFuture`	Chunk data available for read/write	level ⩽ 33 and chunk loaded	`FULL`
`tickingFuture`	Block ticking can execute	level ⩽ 32 and surrounding chunks reached FULL	`BLOCK_TICKING`
`entityTickingFuture`	Entity ticking can execute	level ⩽ 31 and chunk entered ticking	`ENTITY_TICKING`

The initial value `UNLOADED_WORLD_CHUNK_FUTURE` is an **already completed** Future with value `Either.right(Unloaded.INSTANCE)` — indicating "unavailable." Any code waiting on this Future immediately gets an "unavailable" result.

> The three Futures are **progressively tiered**: when `entityTickingFuture` completes, `tickingFuture` and `accessibleFuture` must already be complete. But the reverse is not true — a chunk can be "accessible" but not "ticking," or "ticking" but not "entity ticking."

3 tick(): Warming Up and Cooling Down

`ChunkHolder.tick()` is what actually makes a chunk "come alive" or "cool down." It's called by `ChunkTicketManager.tick()` — only `ChunkHolder` objects with changed load levels are collected into the `chunkHolders` set and trigger `tick()`.

The core logic of `tick()` is to compare **the ChunkLevelType from the last tick** with **the current ChunkLevelType**, then execute warming or cooling operations based on the level difference:

```java
// ChunkHolder.tick() simplified logic
protected void tick(ThreadedAnvilChunkStorage chunkStorage, Executor executor) {
    ChunkLevelType oldType = ChunkLevels.getType(this.lastTickLevel);
    ChunkLevelType newType = ChunkLevels.getType(this.level);
    // Warm up: from inaccessible → FULL
    if (!oldType.isAfter(FULL) && newType.isAfter(FULL)) {
        this.accessibleFuture = chunkStorage.makeChunkAccessible(this);
    }
    // Cool down: from FULL → inaccessible
    if (oldType.isAfter(FULL) && !newType.isAfter(FULL)) {
        this.accessibleFuture.complete(UNLOADED_WORLD_CHUNK);
    }
    // Warm up: from FULL → BLOCK_TICKING
    if (!oldType.isAfter(BLOCK_TICKING) && newType.isAfter(BLOCK_TICKING)) {
        this.tickingFuture = chunkStorage.makeChunkTickable(this);
    }
    // Cool down
    if (oldType.isAfter(BLOCK_TICKING) && !newType.isAfter(BLOCK_TICKING)) {
        this.tickingFuture.complete(UNLOADED_WORLD_CHUNK);
    }
    // Warm up: from BLOCK_TICKING → ENTITY_TICKING
    if (!oldType.isAfter(ENTITY_TICKING) && newType.isAfter(ENTITY_TICKING)) {
        this.entityTickingFuture = chunkStorage.makeChunkEntitiesTickable(this);
    }
    // Cool down
    if (oldType.isAfter(ENTITY_TICKING) && !newType.isAfter(ENTITY_TICKING)) {
        this.entityTickingFuture.complete(UNLOADED_WORLD_CHUNK);
    }
    this.lastTickLevel = this.level;
}
```

Warming up means making the chunk available — creating an incomplete Future and waiting for the generation or loading process to complete it.

Cooling down means immediately completing the corresponding Future with `UNLOADED_WORLD_CHUNK` — any code waiting on this Future immediately receives an "unavailable" signal and stops accessing the chunk.

4 makeChunkAccessible: From ProtoChunk to WorldChunk

`makeChunkAccessible(holder)` is responsible for converting a chunk from "not yet loaded or generated" to "data accessible."

```java
public CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>> makeChunkAccessible(ChunkHolder holder) {
    ChunkPos pos = holder.getPos();
    CompletableFuture<Either<List<Chunk>, ChunkHolder.Unloaded>> regionFuture =
        this.getRegion(holder, 1, distance -> ChunkStatus.FULL);
    return regionFuture.thenApplyAsync(either -> {
        return either.flatMap(chunks -> {
            WorldChunk worldChunk = (WorldChunk)chunks.get(chunks.size() / 2);
            worldChunk.runPostProcessing();
            this.world.startTickingChunk(worldChunk);
            return Either.left(worldChunk);
        });
    }, this.mainExecutor);
}
```

Key operations:

1. **getRegion(holder, 1, ...)**: Wait for a 3×3 chunk region centered on `holder` to all reach `ChunkStatus.FULL`.
2. **runPostProcessing()**: Execute delayed post-processing (like updating fence connections).
3. **startTickingChunk()**: Add the chunk to the world's ticking list.
4. **Return WorldChunk**: Complete the `accessibleFuture` with the completed `WorldChunk`.

Now this chunk's data can be read and written, `World.getChunk(pos)` returns a valid `WorldChunk`, and block entity data and scheduled ticks are all accessible.

5 makeChunkTickable: Enabling Block Ticking

`makeChunkTickable(holder)` prepares the chunk to execute block ticking:

```java
public CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>> makeChunkTickable(ChunkHolder holder) {
    return this.getRegion(holder, 1, distance -> ChunkStatus.FULL)
        .thenApplyAsync(either -> {
            return either.mapLeft(chunks -> {
                WorldChunk worldChunk = (WorldChunk)chunks.get(chunks.size() / 2);
                worldChunk.runPostProcessing();
                return worldChunk;
            });
```

```
        }, this.mainExecutor);
}
```

Wait for a 3×3 region, then run post-processing. After this completes, `tickingFuture` is resolved, the chunk can execute:

- Scheduled ticks (Scheduled Tick)
- Random ticks (Random Tick)
- Block entity ticks (Block Entity Tick)

6 makeChunkEntitiesTickable: Enabling Entity Ticking

`makeChunkEntitiesTickable(holder)` prepares entity ticking:

```java
public CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>>
makeChunkEntitiesTickable(ChunkHolder holder) {
    return this.getRegion(holder, 2, distance -> ChunkStatus.FULL)
        .thenApplyAsync(either -> {
            return either.mapLeft(chunks -> (WorldChunk)chunks.get(chunks.size() / 2));
        }, this.mainExecutor);
}
```

Unlike `makeChunkTickable`, this waits for a **5×5 region** — entities need more context for movement, pathfinding, and collision detection.

7 Common Lifecycle Paths

7.1 Cold Start: From Nothing to Entity Ticking

1. **Add PLAYER ticket** → level=22 (assuming simulation distance 10)
2. **ChunkHolder.tick() triggered** → `oldType=INACCESSIBLE`, `newType=ENTITY_TICKING`
3. **All three Futures created simultaneously**:
   - `accessibleFuture = makeChunkAccessible(this)`
   - `tickingFuture = makeChunkTickable(this)`
   - `entityTickingFuture = makeChunkEntitiesTickable(this)`
4. **Generation chain starts** → `ChunkStatus.EMPTY` → … → `ChunkStatus.FULL`
5. **3×3 region ready** → `accessibleFuture` completes
6. **3×3 region ready** → `tickingFuture` completes
7. **5×5 region ready** → `entityTickingFuture` completes
8. **Chunk becomes entity ticking**

7.2 Gradual Cooling: From Entity Ticking to Unload

1. **Remove PLAYER ticket** → level rises

2. **ChunkHolder.tick() triggered** → level crosses 31 boundary
3. **entityTickingFuture.complete(UNLOADED_WORLD_CHUNK)** → entity ticking stops
4. **Level continues rising** → crosses 32 boundary
5. **tickingFuture.complete(UNLOADED_WORLD_CHUNK)** → block ticking stops
6. **Level crosses 33** → `accessibleFuture.complete(UNLOADED_WORLD_CHUNK)` → chunk marked for unload
7. **ChunkTicketManager collects unloadable chunks** → scheduled for saving
8. **TACS.save()** → writes to region file
9. **ChunkHolder removed from TACS** → memory freed

8 Code Walkthrough

8.1 ChunkHolder.tick() Complete Logic

JAVA

```
protected void tick(ThreadedAnvilChunkStorage chunkStorage, Executor executor) {
    ChunkStatus chunkStatus = ChunkHolder.getTargetGenerationStatus(this.level);
    ChunkStatus chunkStatus2 = ChunkHolder.getTargetGenerationStatus(this.lastTickLevel);
    boolean bl = ChunkLevels.isLoaded(this.level);
    boolean bl2 = ChunkLevels.isLoaded(this.lastTickLevel);
    ChunkLevelType chunkLevelType = ChunkLevels.getType(this.level);
    ChunkLevelType chunkLevelType2 = ChunkLevels.getType(this.lastTickLevel);
    // Generate if needed
    if (!chunkStatus.isAtLeast(chunkStatus2)) {
        this.combinedChunkFuture = chunkStorage.generate(this, chunkStatus);
    }
    // Warm up: inaccessible → FULL
    if (!chunkLevelType2.isAfter(ChunkLevelType.FULL) &&
    chunkLevelType.isAfter(ChunkLevelType.FULL)) {
        this.accessibleFuture = chunkStorage.makeChunkAccessible(this);
        this.tickingFuture = UNLOADED_WORLD_CHUNK_FUTURE;
        this.entityTickingFuture = UNLOADED_WORLD_CHUNK_FUTURE;
        this.updateFutures(chunkStorage, this.accessibleFuture);
    }
    // Cool down: FULL → inaccessible
    if (chunkLevelType2.isAfter(ChunkLevelType.FULL) &&
    !chunkLevelType.isAfter(ChunkLevelType.FULL)) {
        CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>> future = this.accessibleFu-
        ture;
        this.accessibleFuture = UNLOADED_WORLD_CHUNK_FUTURE;
        this.updateFutures(chunkStorage,
            future.thenApply(either -> either.ifLeft(chunkStorage::onChunkStatusChange)));
    }
    // Warm up: FULL → BLOCK_TICKING
    if (!chunkLevelType2.isAfter(ChunkLevelType.BLOCK_TICKING) &&
    chunkLevelType.isAfter(ChunkLevelType.BLOCK_TICKING)) {
        this.tickingFuture = chunkStorage.makeChunkTickable(this);
        this.entityTickingFuture = UNLOADED_WORLD_CHUNK_FUTURE;
        this.updateFutures(chunkStorage, this.tickingFuture);
    }
    // Cool down: BLOCK_TICKING → FULL
```

```
    if (chunkLevelType2.isAfter(ChunkLevelType.BLOCK_TICKING) &&
    !chunkLevelType.isAfter(ChunkLevelType.BLOCK_TICKING)) {
        this.tickingFuture.complete(UNLOADED_WORLD_CHUNK);
        this.tickingFuture = UNLOADED_WORLD_CHUNK_FUTURE;
    }
    // Warm up: BLOCK_TICKING → ENTITY_TICKING
    if (!chunkLevelType2.isAfter(ChunkLevelType.ENTITY_TICKING) &&
    chunkLevelType.isAfter(ChunkLevelType.ENTITY_TICKING)) {
        if (this.entityTickingFuture != UNLOADED_WORLD_CHUNK_FUTURE) {
            throw new IllegalStateException("Unexpected entity ticking future");
        }
        this.entityTickingFuture = chunkStorage.makeChunkEntitiesTickable(this);
        this.updateFutures(chunkStorage, this.entityTickingFuture);
    }
    // Cool down: ENTITY_TICKING → BLOCK_TICKING
    if (chunkLevelType2.isAfter(ChunkLevelType.ENTITY_TICKING) &&
    !chunkLevelType.isAfter(ChunkLevelType.ENTITY_TICKING)) {
        this.entityTickingFuture.complete(UNLOADED_WORLD_CHUNK);
        this.entityTickingFuture = UNLOADED_WORLD_CHUNK_FUTURE;
    }
    // Schedule unload if no longer loaded
    if (bl2 && !bl) {
        this.sendPacketToPlayersWatching(new ChunkDataS2CPacket(this.getWorldChunk(),
        this.world.getLightingProvider(), null, null));
        this.scheduledForUnload = true;
    }
    this.lastTickLevel = this.level;
    this.levelsInRange = ChunkLevels.isOutOfRange(this.level);
}
```

8.2 getRegion: Why Surrounding Chunks Are Needed

JAVA

```
public CompletableFuture<Either<List<Chunk>, ChunkHolder.Unloaded>> getRegion(
    ChunkHolder holder, int margin, IntFunction<ChunkStatus> distanceToStatus
) {
    List<CompletableFuture<Either<Chunk, ChunkHolder.Unloaded>>> list = Lists.newArrayList();
    ChunkPos centerPos = holder.getPos();
    int centerX = centerPos.x;
    int centerZ = centerPos.z;
    // Collect chunks in (2*margin+1)×(2*margin+1) range
    for (int dx = -margin; dx <= margin; dx++) {
        for (int dz = -margin; dz <= margin; dz++) {
            int distance = Math.max(Math.abs(dx), Math.abs(dz));
            ChunkPos pos = new ChunkPos(centerX + dx, centerZ + dz);
            ChunkHolder neighborHolder = this.getCurrentChunkHolder(pos.toLong());
            if (neighborHolder == null) {
                return UNLOADED_CHUNKS_FUTURE;
            }
            ChunkStatus requiredStatus = distanceToStatus.apply(distance);
            CompletableFuture<Either<Chunk, ChunkHolder.Unloaded>> chunkFuture =
                neighborHolder.createFuture(requiredStatus, this);
            list.add(chunkFuture);
        }
    }
    return Util.combineSafe(list).thenApply(eithers -> {
```

```
        // If any chunk unavailable, whole region unavailable
        List<Chunk> chunks = Lists.newArrayList();
        for (Either<Chunk, ChunkHolder.Unloaded> either : eithers) {
            if (either.right().isPresent()) {
                return Either.right(ChunkHolder.Unloaded.INSTANCE);
            }
            chunks.add(either.left().get());
        }
        return Either.left(chunks);
    });
}
```

Key design:

- **All-or-nothing**: If any chunk in the region is unavailable, the entire region is considered unavailable.
- **Different margins for different needs**: `makeChunkTickable` uses margin=1 (3×3), `makeChunkEntitieTickable` uses margin=2 (5×5).
- **Distance-based status requirement**: Center chunks need higher status (FULL), farther chunks may only need lower status.

8.3 makeChunkTickable's margin=1: Why Block Ticking Needs 3×3

JAVA

```
public CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>> makeChunkTickable(ChunkHolder chunk) {
    CompletableFuture<Either<List<Chunk>, ChunkHolder.Unloaded>> regionFuture =
        this.getRegion(chunk, 1, distance -> ChunkStatus.FULL);
    return regionFuture.thenApplyAsync(either -> {
        return either.mapLeft(chunks -> {
            WorldChunk worldChunk = (WorldChunk)chunks.get(chunks.size() / 2);
            worldChunk.runPostProcessing();
            return worldChunk;
        });
    }, this.mainThreadExecutor);
}
```

`getRegion(holder, 1, distance -> ChunkStatus.FULL)` collects all chunks within 1 Chebyshev distance from holder (9 chunks in 3×3), waiting for all to reach `FULL`.

Why are surrounding chunks needed? Making a chunk enter ticking means it starts executing block ticks and random ticks — these operations need to query block states in adjacent chunks (e.g., redstone signal propagation, fluid flow, piston push/pull checks). If adjacent chunks are still generating (still `ProtoChunk`), query results may be incomplete — causing redstone device behavior anomalies. Requiring all surrounding chunks to reach FULL ensures tick operation predictability.

Note the `chunk.runPostProcessing()` line: during generation, some operations (like updating fence connection states, adjusting redstone wire shapes) can't execute immediately at stage 10 (SPAWN) because adjacent chunks may not yet exist. These operations are deferred to `postProcessingLists`, executed centrally in `makeChunkTickable` — at this point chunks in the 3×3 range have all completed generation, giving these delayed operations complete context.

8.4 makeChunkEntitiesTickable's margin=2: Why Entities Need Larger Range

Unlike `makeChunkTickable`, `makeChunkEntitiesTickable` prepares for entity ticking and requires a larger range:

```java
public CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>>
makeChunkEntitiesTickable(ChunkHolder chunk) {
    return this.getRegion(chunk, 2, distance -> ChunkStatus.FULL)
        .thenApplyAsync(either -> either.mapLeft(chunks -> (WorldChunk)chunks.get(chunks.size() / 2)), this.mainThreadExecutor);
}
```

`getRegion(holder, 2, distance -> ChunkStatus.FULL)` collects all chunks within 2 Chebyshev distance from holder (25 chunks in 5×5), waiting for all to reach `FULL`.

Why do entities need 5×5 instead of 3×3? Entity ticking involves three aspects:

1. **Collision detection**: When entities move, their bounding boxes may span multiple chunks. If an entity is near a chunk boundary, its collision detection needs to query block bounding boxes and entity lists in adjacent chunks. The 5×5 range ensures all chunks needed for collision detection are ready even when entities move in corners.
2. **AI pathfinding**: Entity pathfinding systems evaluate paths across multiple chunk ranges. If chunks along the path aren't loaded, pathfinding may fail or produce dead ends. The 5×5 range provides sufficient buffer for most entities (including players, mobs, item drops) pathfinding.
3. **Mob spawning checks**: Hostile mob spawning needs to check player distance, light levels, and block types within a 5×5 chunk range. If your spawning range only covers 3×3, spawning checks near edges will fail — lacking distance checks for more distant players.

Therefore, `makeChunkEntitiesTickable`'s margin=2 is a **safety margin**, providing complete context for all cross-chunk queries in entity ticking, avoiding entity behavior anomalies.

9 References

- Discovering Minecraft - ChunkHolder (CC0 License)
- `net.minecraft.server.world.ChunkHolder`
- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.server.world.ChunkTicketManager`

5.7 Scheduled Ticks and Chunk Runtime Levels

In MicroTiming: Scheduled Ticks and Scheduled Tick Components, we discussed scheduled ticks from a micro-timing perspective, covering information structure, ordering, and scheduled tick suppressors. This chapter supplements one question: **What happens to a scheduled tick when its chunk hasn't entered `BLOCK_TICKING` (weak loading)?**

One-sentence answer: Scheduled ticks can continue to exist in the chunk-level scheduler, but the world-level scheduler won't retrieve and execute scheduled ticks from that chunk. It's not "execute unconditionally when time arrives," but also requires permission from the chunk runtime level.

1 Basics

1.1 Scheduled Ticks Are Just Reservation Slips

A scheduled tick can be understood as "at a certain world time, have a certain position's block or fluid execute a scheduled logic once." As for `TickPriority`, `subTickOrder`, `triggerTick`, and specific component behavior, those were covered in the MicroTiming chapter and won't be repeated here.

What we need to grasp here is another layer of relationship: scheduled ticks aren't isolated objects hanging directly on the entire world, but are first grouped by chunk ownership and placed into the corresponding chunk's `ChunkTickScheduler` (chunk-level scheduler); then `WorldTickScheduler` (world-level scheduler) picks out scheduled ticks from tickable chunks each tick and executes them.

1.2 How Scheduled Ticks Bind to Chunks

Each `WorldChunk` holds its own block and fluid `ChunkTickScheduler`. When a chunk is truly loaded into the world as a `WorldChunk`, its chunk-level scheduler is attached to the world-level scheduler:

```java
// WorldChunk.java
public void addChunkTickSchedulers(ServerWorld world) {
    world.getBlockTickScheduler().addChunkTickScheduler(this.pos, this.blockTickScheduler);
    world.getFluidTickScheduler().addChunkTickScheduler(this.pos, this.fluidTickScheduler);
}
```

When unloading entities and cleaning up chunks, the attachment relationship is removed:

```java
// WorldChunk.java
public void removeChunkTickSchedulers(ServerWorld world) {
    world.getBlockTickScheduler().removeChunkTickScheduler(this.pos);
    world.getFluidTickScheduler().removeChunkTickScheduler(this.pos);
}
```

This shows that "having a chunk-level scheduler recognized by the world-level scheduler" and "this chunk can currently execute scheduled ticks" are not the same thing. The former indicates chunk data has already been involved in world management as a `WorldChunk`; the latter still depends on whether the chunk has

reached the runtime level required for block ticking.

In Chapter 3 Level and Runtime State Layering, we already defined `BLOCK_TICKING` as weak loading: it allows block ticks, redstone computation, random ticks, and scheduled ticks to run, but doesn't run entities. `FULL` below this qualification only guarantees block data is accessible, not active computation.

1.3 How the World-Level Scheduler Determines "This Chunk Can Execute"

`WorldTickScheduler` doesn't just check "are there due scheduled ticks in the queue?" It also holds a `tickingFutureReadyPredicate` (ticking readiness predicate). When a chunk's earliest scheduled tick is already due, the world-level scheduler first asks this predicate: is this chunk now in a ticking state?

If the answer is true, that chunk's `ChunkTickScheduler` is placed in the current tick's executable queue; if the answer is false, it stays in `nextTriggerTickByChunkPos`, waiting to be rechecked in a later tick.

Therefore, chunks not at `BLOCK_TICKING` won't "compensate by executing immediately." They just keep already-due scheduled ticks in the queue. When the chunk re-enters weak loading or entity ticking, the world-level scheduler will retrieve these scheduled ticks again and execute them according to the sorting rules described in the MicroTiming chapter.

> `triggerTick` only indicates "earliest world time when execution is possible," not "must execute at this tick regardless of chunk load state." When chunks aren't ticking, scheduled ticks are delayed, not executed through the runtime level.

1.4 How Scheduled Ticks Cross Chunk Boundaries

A scheduled tick's position determines which chunk it belongs to. `WorldTickScheduler.scheduleTick()` uses `OrderedTick.pos()` to calculate chunk coordinates, then delivers this `OrderedTick` (scheduled tick object) to the corresponding chunk's `ChunkTickScheduler`:

```java
// WorldTickScheduler.java
public void scheduleTick(OrderedTick<T> orderedTick) {
    long l = ChunkPos.toLong(orderedTick.pos());
    ChunkTickScheduler<T> chunkTickScheduler = this.chunkTickSchedulers.get(l);
    if (chunkTickScheduler == null) {
        Util.throwOrPause(new IllegalStateException("Trying to schedule tick in not loaded position
        " + orderedTick.pos()));
    } else {
        chunkTickScheduler.scheduleTick(orderedTick);
    }
}
```

So when a piston pushes a block across a chunk boundary, subsequent scheduled tick ownership depends on "where the new scheduled tick is arranged." If the move process clears scheduled ticks within the original position's range and reschedules scheduled ticks at the new position, then the new scheduled tick enters the

target chunk's chunk-level scheduler; if there's just an unexecuted scheduled tick from the original chunk, it still records the original `pos` and won't automatically rebind to the new coordinates just because the block later moved.

This is why cross-chunk contraptions need extra care: scheduled ticks themselves are "reservations on coordinates," not "tags following block entities as they move." Once coordinates cross a chunk boundary, whether it can execute in the current gt depends on whether the chunk at the new coordinates has reached `BLOCK_TICKING`.

2 Common Misconceptions

MISCONCEPTION	REALITY
"Scheduled ticks must execute when time arrives"	They require both `triggerTick` arrival AND the chunk being in a tickable state.
"Chunks can execute scheduled ticks as long as they're loaded"	"Loaded" needs to distinguish FULL (accessible) from BLOCK_TICKING (tickable). Only the latter allows scheduled tick execution.
"Scheduled ticks in unloaded chunks execute earlier"	They don't execute at all — they wait in the chunk-level scheduler until the chunk re-enters ticking state.
"Pistons automatically transfer scheduled ticks across chunk boundaries"	Scheduled ticks belong to coordinates. New scheduled ticks from block moves need to be rescheduled at the new position; the old scheduled tick doesn't automatically follow.

3 Summary

- Scheduled ticks are grouped by coordinate ownership into chunk-level schedulers; the world-level scheduler only handles cross-chunk collection and execution.
- `WorldChunk` attaches `ChunkTickScheduler` when loaded into the world, removes attachment when unloading and cleaning.
- `BLOCK_TICKING` (weak loading) is the minimum runtime level for scheduled tick execution; `FULL` only guarantees data accessibility, not scheduled tick execution.
- `tickingFutureReadyPredicate` is the chunk ticking threshold before scheduled tick execution, checking whether `ChunkHolder.getTickingFuture()` currently provides a usable `WorldChunk`.
- When crossing chunk boundaries, scheduled ticks follow coordinates and rescheduled positions, not automatically following moved blocks.
- Scheduled tick suppressors must also be in tickable chunks; non-ticking chunks cannot participate in the world-level scheduler's current tick competition.

4 Code Walkthrough

4.1 WorldTickScheduler: tickingFutureReadyPredicate

`WorldTickScheduler` is responsible for collecting and executing scheduled ticks from all chunks. During initialization, it receives a `tickingFutureReadyPredicate`:

```java
// ServerChunkManager.java
this.tickScheduler = new ServerTickScheduler<>(
    (pos) -> this.threadedAnvilChunkStorage.getTickingFuture(pos)
);
```

This lambda passes `ChunkHolder.getTickingFuture()` to the predicate. When `WorldTickScheduler` needs to determine whether a chunk can currently execute scheduled ticks, it calls:

```java
// WorldTickScheduler.java
private boolean shouldTick(ChunkPos pos, long currentTick) {
    CompletableFuture<Either<WorldChunk, ChunkHolder.Unloaded>> future =
        this.tickingFutureReadyPredicate.apply(pos);
    return future != null && future.getNow(ChunkHolder.UNLOADED_WORLD_CHUNK).left().isPresent();
}
```

Interpretation:

1. Get the `tickingFuture` for that chunk position.
2. If the future is null (chunk holder doesn't exist), can't tick.
3. If the future is completed and the result is `Either.left(WorldChunk)`, can tick.
4. If the future is not yet completed or the result is `Either.right(Unloaded)`, can't tick.

This check doesn't happen every tick for every chunk — only chunks whose `nextTriggerTickBy-ChunkPos` records indicate "time has arrived" are checked.

4.2 Chunk-Level Scheduler Collection Logic

When `WorldTickScheduler.tick()` executes, it first traverses `nextTriggerTickByChunkPos` to find chunks whose earliest scheduled tick has arrived:

```java
// WorldTickScheduler.java
public void tick(long time, int maxTicks, Consumer<OrderedTick<T>> tickConsumer) {
    profiler.push("collect");
    this.collectTickableChunkTickSchedulers(time);
    profiler.pop();
    ...
}
private void collectTickableChunkTickSchedulers(long time) {
    ObjectIterator<Long2LongMap.Entry> objectIterator =
        this.nextTriggerTickByChunkPos.long2LongEntrySet().iterator();
    while (objectIterator.hasNext()) {
        Long2LongMap.Entry entry = objectIterator.next();
        long l = entry.getLongKey();
        long m = entry.getLongValue();
```

```
        if (m <= time) {
            ChunkTickScheduler<T> chunkTickScheduler = this.chunkTickSchedulers.get(l);
            if (chunkTickScheduler == null) {
                objectIterator.remove();
            } else {
                OrderedTick<T> orderedTick = chunkTickScheduler.peekNextTick();
                if (orderedTick == null) {
                    objectIterator.remove();
                } else if (orderedTick.triggerTick() > time) {
                    entry.setValue(orderedTick.triggerTick());
                } else if (this.tickingFutureReadyPredicate.test(l)) {
                    objectIterator.remove();
                    this.tickableChunkTickSchedulers.add(chunkTickScheduler);
                }
            }
        }
    }
}
```

This code has four filtering layers:

1. `m <= time`: This chunk's recorded next scheduled tick time has reached current world time.
2. `chunkTickScheduler == null`: If the chunk-level scheduler no longer exists, remove this index.
3. `orderedTick == null` or `orderedTick.triggerTick() > time`: If the head scheduled tick doesn't exist, or the head scheduled tick hasn't actually arrived yet, correct or clean up the index.
4. `this.tickingFutureReadyPredicate.test(l)`: After time has arrived and queue exists, finally check whether the chunk can actually tick.

The key is the last layer. Only by passing the ticking readiness check does the chunk-level scheduler enter `tickableChunkTickSchedulers`. If it doesn't pass, the code neither `remove()`s nor `add()`s; it stays in `nextTriggerTickByChunkPos`, to be tried again next time the world-level scheduler collects.

4.3 tickingFutureReadyPredicate Implementation

When `ServerWorld` creates block and fluid world-level schedulers, it passes the same method reference to the constructor:

JAVA

```
// ServerWorld.java
private final WorldTickScheduler<Block> blockTickScheduler = new WorldTickScheduler<>(this::isTicking-
FutureReady, this.getProfilerSupplier());
private final WorldTickScheduler<Fluid> fluidTickScheduler = new WorldTickScheduler<>(this::isTicking-
FutureReady, this.getProfilerSupplier());
```

This method in `ServerWorld` first confirms the chunk entity management side still considers it loaded, then hands off to `ServerChunkManager`:

JAVA

```
// ServerWorld.java
private boolean isTickingFutureReady(long chunkPos) {
    return this.isChunkLoaded(chunkPos) && this.chunkManager.isTickingFutureReady(chunkPos);
}
```

The actual chunk runtime level check is in `ServerChunkManager.isTickingFutureReady()`:

```java
// ServerChunkManager.java
public boolean isTickingFutureReady(long pos) {
    ChunkHolder chunkHolder = this.getChunkHolder(pos);
    if (chunkHolder == null) {
        return false;
    }
    if (!this.world.shouldTickBlocksInChunk(pos)) {
        return false;
    }
    Either<WorldChunk, ChunkHolder.Unloaded> either = chunkHolder.getTickingFuture().getNow(null);
    return either != null && either.left().isPresent();
}
```

Two thresholds here:

- `world.shouldTickBlocksInChunk(pos)` goes to the ticket system's `shouldTickBlocks(chunkPos)`, which is the block computation qualification discussed in Chapter 3;
- `chunkHolder.getTickingFuture().getNow(null)` must have obtained `Either.left(WorldChunk)`, indicating `ChunkHolder`'s `tickingFuture` is completed and the completion result is a usable `WorldChunk`, not `Unloaded`.

So `tickingFutureReadyPredicate` is essentially asking: **Does this chunk already have a usable ticking Future?** In plainer terms, it's checking whether a condition like `ChunkHolder.getTickingFuture().isDone()` is met, and the completion result must be `WorldChunk`. In Yarn 1.20.1+build.10's actual implementation, it doesn't directly call `isDone()`, but uses `getNow(null)` to get the current result, then checks whether the result is `Either.left(WorldChunk)`.

Note this doesn't block waiting for Future completion. `getNow(null)` means "is it completed now? If not, give me null." Therefore the scheduled tick system won't wait for a certain chunk to finish loading; it just checks in passing during each tick's collection, and skips if not ready.

4.4 Execution Location in ServerWorld.tick

Block scheduled ticks actually execute in `ServerWorld.tick()`'s `tickPending` phase:

```java
// ServerWorld.java
profiler.swap("tickPending");
if (!this.isDebugWorld()) {
    long m = this.getTime();
    profiler.push("blockTicks");
    this.blockTickScheduler.tick(m, 65536, this::tickBlock);
    profiler.swap("fluidTicks");
    this.fluidTickScheduler.tick(m, 65536, this::tickFluid);
    profiler.pop();
}
```

The `65536` here is the maximum number of scheduled ticks retrieved in this call. `this::tickBlock` is the entry point where scheduled tick objects finally trigger block `scheduledTick` behavior; but before reaching it, scheduled ticks have already gone through world-level scheduler collection, chunk-level scheduler

> filtering, and `tickingFutureReadyPredicate`'s chunk runtime level check.
>
> This explains a common phenomenon: if a scheduled tick is already due when the chunk leaves weak loading, it won't continuously execute while offline, nor will it force execution when world time reaches `triggerTick`. Only when the chunk re-enters `BLOCK_TICKING` will it be re-collected in subsequent `ServerWorld.tick()` and executed as an already-due scheduled tick.

5 References

- MicroTiming: Scheduled Ticks and Scheduled Tick Components
- ChunkMechanics: Level and Runtime State Layering
- `net.minecraft.world.tick.WorldTickScheduler`
- `net.minecraft.world.tick.ChunkTickScheduler`
- `net.minecraft.server.world.ServerWorld`
- `net.minecraft.server.world.ServerChunkManager`
- `net.minecraft.server.world.ChunkHolder`

5.8 Random Ticks and Block Entity Ticks

Chapter 3 already mentioned: after a chunk reaches **block ticking level** (`BLOCK_TICKING`), the server allows three types of block computation — **Scheduled Tick**, **Random Tick**, and **Block Entity Tick**. For scheduled tick details, see MicroTiming: Scheduled Ticks and Scheduled Tick Components; this chapter only fills in the other two: how random ticks and block entity ticks depend on chunk runtime levels.

1 Random Ticks

Random tick is a mechanism that samples several random positions within a chunk each game tick, then has the sampled blocks or fluids execute their own random logic. It's not a scheduled tick: scheduled ticks have explicit trigger times, positions, types, and priorities, while random ticks have no reservation queue and don't guarantee a certain block will be executed at a certain gt. A crop block not being sampled for a long time is normal; the same position in a chunk being sampled on consecutive gt is equally possible.

Random ticks first depend on the **random tickable block count** (`randomTickableBlockCount`) introduced in Chapter 1. Each **chunk section** incrementally maintains this count when writing block states; when the count is 0, `ChunkSection.hasRandomTicks()` directly returns false, and `ServerWorld.tickChunk()` won't sample within this 16×16×16 region. The significance of this design isn't to make random ticks more "random," but to skip large numbers of stone, air, and building material sections with no random tick blocks in O(1) time.

Sampling frequency is controlled by **random tick speed** (`randomTickSpeed`, game rule). The server executes `randomTickSpeed` samples for each random tickable section; the default value is 3, setting it to 0 completely skips the random tick loop. Note this game rule controls "how many coordinate samples per random tickable section per gt," not "how many times each crop ticks per gt." Therefore, increasing `randomTickSpeed` increases sampling opportunities but doesn't turn random ticks into deterministic periodic ticks.

What typically needs random ticks are blocks or fluids that "continuously exist in the world, occasionally change spontaneously," such as crop growth, grass block spread, leaf decay, fungus spread, fire changes, ice/snow-related changes, and some fluid logic. Their common characteristic is: they don't need to execute at a precise gt, yet need to gradually progress while the chunk maintains block ticking qualification.

From a macro timing perspective, random ticks and scheduled ticks both belong to the server's per-gt block computation phase (commonly called TT phase), and both depend on chunks reaching `BLOCK_TICKING`. But the two mechanisms are completely different: scheduled ticks are first retrieved from `WorldTickScheduler` as expired tasks during world tick progression, then call `scheduledTick()`; random ticks traverse sections and sample coordinates during chunk tick, then call `randomTick()` or fluid random tick methods. When studying scheduled tick order, don't insert random ticks into the scheduled tick queue; when studying crops, grass blocks, or leaves, don't assume they have a hidden scheduled tick.

2 Block Entity Ticks

Block entities are used to store or compute complex data that `BlockState` can't express. BlockMechanics Chapter 1 already outlined the division of labor: **Block State** is suitable for storing finite-valued, quick-lookup information; block entities are suitable for storing complex data like chest items, hopper cooldowns, beacon effects, and sign text. Not all block entities compute every gt, but those that do compute need to enter block entity ticks.

The entry point for block entity ticks isn't "traverse all block entities in a chunk then guess who needs to tick," but is registered by **block entity tickers**. `WorldChunk.updateTicker()` reads the block entity's cached `BlockState`, asks through `blockState.getBlockEntityTicker(world, type)` whether this state provides a ticker behind the **block entity tick invoker** (`BlockEntityTickInvoker`). If there's no ticker, it removes the old invoker; if there is a ticker, it wraps it as a `DirectBlockEntityTickInvoker`, then adds it to the world-level list through `World.addBlockEntityTicker()`.

Here the chunk field directly affects whether the ticker actually executes. `WorldChunk` holds `levelTypeProvider`, can query current `ChunkLevelType` at any time; the wrapped invoker checks `WorldChunk.canTickBlockEntity(pos)` before each `tick()`. On the server, this check requires the block position to be within world boundaries, the chunk still loaded, and `getLevelType().isAfter(ChunkLevelType.BLOCK_TICKING)`. In other words, the ticker can already be in the list, but after the chunk drops to `FULL`, the invoker is blocked before execution.

`World.addBlockEntityTicker()` is responsible for placing the invoker in the world's `blockEntityTickers` list; if currently traversing block entities, it first places it in `pendingBlockEntityTickers`, merging next round to avoid modifying the list during traversal. Then `World.tickBlockEntities()` traverses these invokers each gt: removed invokers are deleted from the list, still-tickable positions call `tick()`. The actual behavior is still determined by the block entity's own ticker.

Different block entities' internal cycles vary widely. Furnaces and moving pistons typically advance state each gt; hoppers enter the ticker each gt but use `transferCooldown` to form the common 8gt transfer rhythm; beacons update effects at roughly 80gt cycles in their own logic; sign block entities mainly store data and may not provide a server ticker. The chunk management system only determines "does this invoker qualify for execution," not the business cycle for each block entity type.

When chunks downgrade or unload, block entity ticks stop. When dropping to `FULL` or lower, `canTickBlockEntity()` no longer passes; even if the invoker remains in the world list, it won't execute the actual ticker; when a block entity is removed, `removeBlockEntityTicker()` replaces the wrapped invoker with a removed empty invoker, which is cleaned from the list during the next `World.tickBlockEntities()` traversal; when a chunk unloads, `WorldChunk.clear()` marks all block entities removed and clears the chunk's own ticker map.

3 How It Connects with BLOCK_TICKING

`BLOCK_TICKING` isn't a separate "redstone switch," but a ticking view. Chapter 7 explained that after `ChunkHolder.tickingFuture` completes, the chunk enters a view that can perform block computation; this view supports scheduled ticks, random ticks, and block entity ticks upward.

The three computations depend on it in slightly different ways: scheduled ticks depend on `WorldTickScheduler` determining whether the target chunk's tick future is ready; random ticks depend on `ServerChunkManager` only delivering `WorldChunk` objects that have reached block computation qualification to `ServerWorld.tickChunk()`; block entity ticks check once more during world-level list traversal through `shouldTickBlockPos()` and `WorldChunk.canTickBlockEntity()` whether the chunk is still at `BLOCK_TICKING` or higher. The commonality: as long as the chunk returns to `FULL`, it can still be readable and writable in memory, but these active block computations should all stop.

4 Common Misconceptions

MISCONCEPTION	REALITY
"Random ticks are just randomly delayed scheduled ticks"	Random ticks have no reservation structure and don't enter the scheduled tick queue. They're per-gt coordinate sampling.
"`randomTickSpeed = 3` means each crop ticks once every 3gt"	It means each random tickable section samples 3 coordinate positions per gt. Whether a single block is sampled is uncertain.
"Block entities tick when chunks are accessible"	Accessible `FULL` isn't enough; server block entity tickers require chunks to at least reach `BLOCK_TICKING`.
"Block entity ticks equal entity ticks"	Block entities are fixed at block coordinates, belonging to block computation; normal entity movement and AI require `ENTITY_TICKING`.

5 Summary

- Random ticks are a per-gt coordinate sampling mechanism, not part of the scheduled tick queue.
- `randomTickableBlockCount` lets `ServerWorld.tickChunk()` skip sections with no random tick requirements.
- `randomTickSpeed` controls sampling frequency per random tickable section, not a fixed cycle for individual blocks.
- Block entity tickers are registered or removed by `WorldChunk.updateTicker()` based on current `BlockState`.
- `World.tickBlockEntities()` traverses the world-level invoker list, but still checks whether the block position allows ticking before actual execution.
- Scheduled ticks, random ticks, and block entity ticks all depend on the `BLOCK_TICKING` view; after chunks return to `FULL`, data can exist in memory but active block computation stops.

6 Code Walkthrough

6.1 ServerWorld.tickChunk(): Random Tick Section

`ServerWorld.tickChunk(WorldChunk chunk, int randomTickSpeed)` processes a `WorldChunk` selected to participate in chunk tick. After weather, thunderstorm, ice/snow logic, the random tick body is in the `tickBlocks` profiler section:

```java
// ServerWorld.java
profiler.swap("tickBlocks");
if (randomTickSpeed > 0) {
    ChunkSection[] chunkSections = chunk.getSectionArray();
    for (int m = 0; m < chunkSections.length; m++) {
        ChunkSection chunkSection = chunkSections[m];
        if (chunkSection.hasRandomTicks()) {
            int n = chunk.sectionIndexToCoord(m);
            int o = ChunkSectionPos.getBlockCoord(n);
            for (int p = 0; p < randomTickSpeed; p++) {
                BlockPos blockPos = this.getRandomPosInChunk(i, o, j, 15);
                BlockState blockState = chunkSection.getBlockState(
                    blockPos.getX() - i,
                    blockPos.getY() - o,
                    blockPos.getZ() - j
                );
                if (blockState.hasRandomTicks()) {
                    blockState.randomTick(this, blockPos, this.random);
                }
                FluidState fluidState = blockState.getFluidState();
                if (fluidState.hasRandomTicks()) {
                    fluidState.onRandomTick(this, blockPos, this.random);
                }
            }
        }
    }
}
```

Three key points.

First, when `randomTickSpeed <= 0`, the entire random tick loop doesn't enter, so gamerule 0 turns off random ticks.

Second, the outer traversal is of the chunk's `ChunkSection[]`, not all 16×16×height blocks. `chunkSection.hasRandomTicks()` is exactly the counter filtering introduced in Chapter 1: if that section has no blocks or fluids declaring random tick requirements, no sampling occurs.

Third, sampling first retrieves the sampled `BlockState`, then separately checks whether the block state and fluid state truly `hasRandomTicks()`. This step shows "section is random tickable" is just coarse filtering; when a specific coordinate samples air, stone, or other non-random-ticking blocks, this sample produces no block random tick behavior.

6.2 World.tickBlockEntities(): List Traversal and Execution

`World` maintains two lists: `blockEntityTickers` stores currently traversable block entity tick invokers, `pendingBlockEntityTickers` temporarily stores invokers added during traversal.

```java
// World.java
public void addBlockEntityTicker(BlockEntityTickInvoker ticker) {
    (this.iteratingTickingBlockEntities
        ? this.pendingBlockEntityTickers
        : this.blockEntityTickers).add(ticker);
}
protected void tickBlockEntities() {
    Profiler profiler = this.getProfiler();
    profiler.push("blockEntities");
    this.iteratingTickingBlockEntities = true;
    if (!this.pendingBlockEntityTickers.isEmpty()) {
        this.blockEntityTickers.addAll(this.pendingBlockEntityTickers);
        this.pendingBlockEntityTickers.clear();
    }
    Iterator<BlockEntityTickInvoker> iterator = this.blockEntityTickers.iterator();
    while (iterator.hasNext()) {
        BlockEntityTickInvoker blockEntityTickInvoker = iterator.next();
        if (blockEntityTickInvoker.isRemoved()) {
            iterator.remove();
        } else if (this.shouldTickBlockPos(blockEntityTickInvoker.getPos())) {
            blockEntityTickInvoker.tick();
        }
    }
    this.iteratingTickingBlockEntities = false;
    profiler.pop();
}
```

`shouldTickBlockPos(pos)` converts the block coordinate to chunk coordinates and calls `shouldTickBlocksInChunk()`. In `ServerWorld`, it ultimately queries the ticket manager's `shouldTickBlocks(chunkPos)`. This binds block entity ticks to the `BLOCK_TICKING` level like scheduled ticks and random ticks.

But there's a second gate here: `BlockEntityTickInvoker.tick()` is implemented in `WorldChunk.DirectBlockEntityTickInvoker`, which calls `WorldChunk.canTickBlockEntity(pos)` again. This step checks world boundaries, whether the chunk is loaded, and `getLevelType().isAfter(ChunkLevelType.BLOCK_TICKING)`. Two layers of checks ensure even if the world-level list briefly retains an invoker, it won't execute actual block entity logic at the wrong level.

6.3 WorldChunk.updateTicker(): Registration, Replacement, and Removal

When block state changes, block entities load, or chunks enter the world, `WorldChunk` updates the corresponding block entity's ticker. The core method is `updateTicker()`:

```java
// WorldChunk.java
private <T extends BlockEntity> void updateTicker(T blockEntity) {
    BlockState blockState = blockEntity.getCachedState();
    BlockEntityTicker<T> blockEntityTicker =
        blockState.getBlockEntityTicker(this.world, (BlockEntityType<T>)blockEntity.getType());
```

```
    if (blockEntityTicker == null) {
        this.removeBlockEntityTicker(blockEntity.getPos());
    } else {
        this.blockEntityTickers.compute(blockEntity.getPos(), (pos, ticker) -> {
            BlockEntityTickInvoker blockEntityTickInvoker =
                this.wrapTicker(blockEntity, blockEntityTicker);
            if (ticker != null) {
                ticker.setWrapped(blockEntityTickInvoker);
                return (WorldChunk.WrappedBlockEntityTickInvoker)ticker;
            } else if (this.canTickBlockEntities()) {
                WorldChunk.WrappedBlockEntityTickInvoker wrappedBlockEntityTickInvoker =
                    new WorldChunk.WrappedBlockEntityTickInvoker(blockEntityTickInvoker);
                this.world.addBlockEntityTicker(wrappedBlockEntityTickInvoker);
                return wrappedBlockEntityTickInvoker;
            } else {
                return null;
            }
        });
    }
}
```

Three branches:

1. If `blockEntityTicker == null`, the current block state doesn't provide a ticker, remove the old ticker.
2. If `ticker != null` (invoker already exists), update its internal wrapped object through `setWrapped`, reusing the existing `WrappedBlockEntityTickInvoker`.
3. If `ticker == null` and `canTickBlockEntities()` passes, create a new `WrappedBlockEntityTickInvoker` and add it to the world list.

`canTickBlockEntities()` is the same check as `canTickBlockEntity()`: it depends on `ServerChunkManager.canTick()`, which ultimately checks whether `ChunkHolder.getTickingFuture()` is completed.

Therefore, when a block state changes — whether placed, removed, or state updated — `WorldChunk.setBlockState()` eventually calls `updateTicker()` to adjust the corresponding block entity's ticker registration.

6.4 Block Entity Tick Order and HashMap Resize (Historical Context)

This section records a historically well-known block entity tick order issue: **HashMap resize causing tile entity iteration order changes**.

6.4.1 Historical Background

In Minecraft versions before 1.13, many blocks that didn't truly need per-tick computation (like redstone dust, comparators) were also implemented as TileEntity for historical reasons. Block entity tick execution order directly affected redstone update propagation order, making this topic particularly sensitive.

Although 1.13+ removed unnecessary TileEntities, community discussion and related tooling (like `te_swap_calculator.py`) still circulates. This content is kept as historical reference, helping readers understand why "block entity order flip after reload" remains discussed.

6.4.2 Root Cause: WorldChunk.blockEntities is a HashMap

```java
// WorldChunk.java
private final Map<BlockPos, BlockEntity> blockEntities = new HashMap<>();
```

`HashMap` doesn't guarantee iteration order. But in actual server operations, barring hash collisions or resize, iteration order tends to remain stable — producing an **illusion of reproducibility**.

However, when the number of block entities in a chunk exceeds the current HashMap capacity threshold (default initial capacity 12, load factor 0.75), resize is triggered:

```java
// HashMap.java (simplified)
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int newCap = oldCap << 1;  // Double capacity
    Node<K,V>[] newTab = new Node[newCap];
    for (int j = 0; j < oldCap; ++j) {
        Node<K,V> e;
        if ((e = oldTab[j]) != null) {
            // Redistribute entries to new buckets based on hash & (newCap - 1)
            ...
        }
    }
    return newTab;
}
```

After resize, entries are redistributed based on new capacity's bitmask, potentially changing iteration order.

6.4.3 Trigger Condition

For chunks with 12 or fewer block entities, the default HashMap never resizes — iteration order remains stable. Once the 13th block entity is added:

```
12 < threshold (12 * 0.75 = 9) → no resize
13 > threshold → resize to capacity 32
```

After resize to 32, `BlockPos` entries are redistributed based on `hash(pos) & 31` instead of `hash(pos) & 11`, **causing some coordinate pair iteration orders to swap**.

6.4.4 Propagation Path to Save File

When `WorldChunk` saves, `ChunkSerializer.serialize()` traverses `blockEntities.values()` and writes block entity NBT to the chunk's `block_entities` list:

```java
// ChunkSerializer.java (simplified)
for (BlockEntity blockEntity : chunk.getBlockEntities().values()) {
    NbtCompound nbt = blockEntity.toNbt();
    blockEntitiesNbt.add(nbt);
}
```

ArrayList insertion order is entirely determined by "which `addBlockEntityTicker()` executed first." And this order comes from NBT list storage order — the iteration order from `getBlockEntityPositions()` during the previous save. **Therefore when `blockEntities` resizes and changes traversal order, it cascades to affect tick execution order on next load.**

6.4.5 Actual Impact

For most players, **no noticeable effect**. Reasons:

- Most chunks have far fewer than 12 block entities, never triggering resize
- Even if triggered, impact is limited to **relative order within that chunk**
- Different chunks' block entity tick order is jointly determined by chunk tick order interleaving and each ticker's independent position; resize only exposes order issues in specific, fragile timing-dependent scenarios

Truly affected scenarios:

- **Redstone dust update order** (historical): Before 1.13, redstone dust was also a TileEntity; subsequent versions removed this, but community awareness of "reload causes TE order flip" persists
- **Hopper order and item distribution**: Hopper `BlockEntityTicker` execution order determines item flow; in hopper chains, TE order flip can change item routing
- **Piston timing chains**: Multiple pistons executing sequentially within the same gt; order changes can alter overall behavior

6.4.6 Tool: Find Resize Flip Coordinate Pairs

Community open-source tool `te_swap_calculator.py` calculates coordinate pairs whose order flips after resize based on `Vec3i.hashCode()` formula and Java `HashMap` resize rules for a given chunk and TE count:

```
# In chunk(0, 0), find flip pairs when 12 TEs resize to 32 capacity
python te_swap_calculator.py 0 0 -k 12 -n 50
```

Filtering condition: For chunks at BLOCK_TICKING level, only TE order changes falling within `canTickBlockEntity()` range actually affect tick order. This tool is primarily used for redstone contraption debugging and order-sensitive design reference.

7 References

- Discovering Minecraft - Basic Concepts in Chunk Management System (CC0 License)
- Yarn Mapping `1.20.1+build.10`
- `net.minecraft.server.world.ServerWorld`
- `net.minecraft.world.World`
- `net.minecraft.world.chunk.WorldChunk`
- `net.minecraft.world.chunk.ChunkSection`

5.9 Entity Loading and Tracking

Previous chapters discussed chunks themselves: `ChunkStatus` determines generation progress, `ChunkLevelType` determines runtime qualification. But entities aren't simply stuffed into `WorldChunk` to compute alongside blocks. Minecraft uses a separate specialized management system for entities: block data answers "what block is at this coordinate," the entity system answers "what movable, syncable, savable objects exist in this space."

1 Why Entities and Blocks Are Managed Separately

Block storage is in a chunk's `ChunkSection[]`, where each `ChunkSection` is a 16×16×16 block state array; entities are stored in `ServerEntityManager`, and the entity manager builds indexes by chunk / section. In other words, blocks and entities are two independent data structures, two independent loading/unloading mechanisms: `World.getBlockState()` can only retrieve `BlockState` from a chunk's block array, can't "incidentally" get entities; similarly, entity queries go through `EntityLookup`, won't trigger new chunk loading just to find entities.

2 ServerEntityManager's Responsibilities

`ServerEntityManager` is the server entity manager. It doesn't manage "where a specific entity currently is," but rather entity storage and indexing grouped by chunk / section: `SectionedEntityCache` is responsible for placing entities into corresponding `EntityTrackingSection`, `EntityIndex` handles quick lookups by UUID / id, `trackingStatuses` records each chunk's current corresponding `EntityTrackingStatus`. When an entity moves to another section, the entity's own `EntityChangeListener` calls back to the manager, moving it from the old section to the new section, and decides whether to start or stop tracking, start or stop ticking based on new/old tracking status.

Loading and unloading are also coordinated here. `WorldChunk.loadEntities()` executes the `EntityLoader` saved during chunk construction, ultimately delivering the entity stream read from NBT to `ServerWorld.addEntities()` / `ServerEntityManager.loadEntities()`, mounting them into the entity manager. During unloading, `ServerEntityManager.unload(ChunkPos)` first calls `trySave()` to write this chunk's savable entities to the `entities` region file, then marks these entities as `UNLOADED_TO_CHUNK` and removes them from memory management state. Whether entities are visible to clients, whether they participate in ticking, is represented by `EntityTrackingStatus`: `HIDDEN` doesn't track or tick, `TRACKED` syncs but doesn't tick, `TICKING` both syncs and ticks.

3 EntityTracker and Player Sync

An entity "existing on the server" and "being visible to a certain player's client" are not the same thing. TACS (`ThreadedAnvilChunkStorage`) internally maintains `entityTrackers`: an `Int2ObjectMap<EntityTracker>`, where keys are entity network ids and values are corresponding entity trackers. When an entity enters a track-

able state, `ServerEntityManager` calls `EntityHandler.startTracking()`, then `ServerWorld.ServerEntityHandler` delivers the entity to TACS's `loadEntity()`, creating an `EntityTracker`.

Each `EntityTracker` holds an `EntityTrackerEntry`, recording the entity's last synced state to clients. `EntityTracker.updateTrackedStatus(player)` compares horizontal distance between player and entity: when distance doesn't exceed `min(entity type tracking distance, view distance * 16)`, and the entity allows observation by that player, it adds the player's network connection to the listener set and sends a spawn packet; otherwise removes the listener and sends a remove packet. The tracking range here is jointly limited by "entity type's own maximum tracking distance" and "server view distance," and whether the entity actually computes still depends on simulation distance and `ENTITY_TICKING` — they're related but not the same switch.

4 ENTITY_TICKING Prerequisites

Chapter 7 already showed that `makeChunkEntitiesTickable()` doesn't just wait for the center chunk itself to reach `FULL`, but calls `getRegion(holder, 2, distance -> ChunkStatus.FULL)`, requiring a 5×5 range centered on the center chunk to all reach `FULL`. This is stricter than `makeChunkTickable()`'s 3×3: block ticking only needs a closer neighborhood, while entity movement, collision, pathfinding, and spawn determination more easily cross chunk boundaries. The final level threshold still comes from `ChunkLevels.shouldTickEntities(level)`: only chunks with `level <= 31` have entity computation qualification.

5 Entity Queries Don't Trigger Chunk Loading

The underlying implementation of `World.getEntitiesByClass()` / `World.getEntitiesByType()` is `getEntityLookup().forEachIntersects(...)`: it only searches for intersecting entities in the entity index currently held by the entity manager, won't request `ChunkStatus.FULL` like chunk reading, and won't add loading tickets. This echoes Chapter 5's conclusion: entity queries and block queries are two separate mechanisms. Block queries ask "is a certain chunk's block array accessible," entity queries ask "are there objects in the currently loaded, mounted entity index."

6 Summary

- Block data is in `ChunkSection[]`, entity data is in `ServerEntityManager`; they're independently stored, loaded, and unloaded.
- `ServerEntityManager` handles entity chunk / section grouping, loading/saving, tracking state switching, and start/stop tracking and start/stop ticking lifecycle callbacks.
- `EntityTracker` handles network sync: maintains nearby player sets for each entity, sends spawn / update / remove packets based on distance.
- `ENTITY_TICKING` requires level $\leqslant 31$, and `makeChunkEntitiesTickable()` requires a 5×5 `FULL` chunk context.

- Entity queries only search entities already loaded into the entity manager, won't trigger new chunk loading.

7 Code Walkthrough

7.1 ServerEntityManager.trySave() and unload()

`ServerEntityManager`'s save logic starts from `trySave(long chunkPos, Consumer<T> action)`:

```java
private boolean trySave(long chunkPos, Consumer<T> action) {
    ServerEntityManager.Status status = this.managedStatuses.get(chunkPos);
    if (status == ServerEntityManager.Status.PENDING) {
        return false;
    }
    List<T> list = this.cache
        .getTrackingSections(chunkPos)
        .flatMap(section -> section.stream().filter(EntityLike::shouldSave))
        .collect(Collectors.toList());
    if (list.isEmpty()) {
        if (status == ServerEntityManager.Status.LOADED) {
            this.dataAccess.writeChunkData(new ChunkDataList<>(new ChunkPos(chunkPos),
            ImmutableList.of()));
        }
        return true;
    } else if (status == ServerEntityManager.Status.FRESH) {
        this.scheduleRead(chunkPos);
        return false;
    } else {
        this.dataAccess.writeChunkData(new ChunkDataList<>(new ChunkPos(chunkPos), list));
        list.forEach(action);
        return true;
    }
}
```

Three states need clarification: `PENDING` means the chunk's entity data is still being read asynchronously, can't save now; `FRESH` means the manager hasn't read this chunk's entity data yet, saving rashly would overwrite old entities on disk, so it first calls `scheduleRead()` and returns failure; `LOADED` means entity data has been read or confirmed empty, can safely write back. The save target comes from `cache.getTrackingSections(chunkPos)`, meaning all entities in all sections under this chunk that `shouldSave()`, not from `WorldChunk`'s block data.

`unload(ChunkPos chunkPos)` first calls `trySave` to write data, then marks entities as unloaded:

```java
public void unload(ChunkPos chunkPos) {
    long l = chunkPos.toLong();
    this.trySave(l, entity -> entity.setRemovalReason(Entity.RemovalReason.UNLOADED_TO_CHUNK));
    this.managedStatuses.remove(l);
    this.cache.getTrackingSections(l).forEach(section -> {
        section.getEntities().filter(entity -> !entity.shouldSave()).forEach(entity -> {
            entity.setRemovalReason(Entity.RemovalReason.DISCARDED);
        });
```

```
    });
}
```

`setRemovalReason(UNLOADED_TO_CHUNK)` marks the entity as "temporarily unloaded," distinguished from permanent removal (`DISCARDED` / `KILLED`). These entities will be reloaded in the next `WorldChunk.loadEntities()` when the chunk loads again.

7.2 `trackingStatuses`: How Entity Tracking Status Changes

`ServerEntityManager.trackingStatuses` is an `Int2ObjectMap<Tracking>`, recording the current `EntityTrackingStatus` for each chunk:

```java
public enum EntityTrackingStatus {
    HIDDEN,   // Not tracked, not ticking
    TRACKED,  // Tracked, not ticking
    TICKING   // Tracked and ticking
}
```

`HIDDEN` means entities in this chunk neither sync to clients nor participate in ticking — often regions far beyond view distance but not yet unloaded.

`TRACKED` means entities can sync to clients but don't tick — regions within view distance but outside simulation distance.

`TICKING` means entities both sync to clients and participate in ticking — regions within simulation distance.

When a chunk's level changes, `ChunkTicketManager.tick()` calls `ServerEntityManager.updateTrackingStatus(chunkPos)`:

```java
void updateTrackingStatus(long chunkPos, EntityTrackingStatus trackingStatus) {
    Tracking tracking = this.trackingStatuses.get(chunkPos);
    if (tracking != null && tracking.getStatus() == trackingStatus) {
        return;
    }
    if (trackingStatus == EntityTrackingStatus.HIDDEN) {
        if (tracking != null) {
            tracking.updateStatus(EntityTrackingStatus.HIDDEN, this.listener);
            this.trackingStatuses.remove(chunkPos);
        }
    } else {
        if (tracking != null) {
            tracking.updateStatus(trackingStatus, this.listener);
        } else {
            this.trackingStatuses.put(chunkPos, new Tracking(chunkPos, trackingStatus, this.listener));
        }
    }
}
```

Key point: when the new status is `HIDDEN`, remove the record from `trackingStatuses`; otherwise create or update. The actual start/stop tracking and ticking logic is in `Tracking.updateStatus()`:

```java
void updateStatus(EntityTrackingStatus status, Listener listener) {
    EntityTrackingStatus old = this.status;
    this.status = status;
    boolean oldTracked = old.shouldTrack();
    boolean newTracked = status.shouldTrack();
    if (oldTracked && !newTracked) {
        this.sections.forEach(section -> section.getEntities().forEach(listener::stopTracking));
    } else if (!oldTracked && newTracked) {
        this.sections.forEach(section -> section.getEntities().forEach(listener::startTracking));
    }
    boolean oldTicking = old.shouldTick();
    boolean newTicking = status.shouldTick();
    if (oldTicking && !newTicking) {
        this.sections.forEach(section -> section.getEntities().forEach(listener::stopTicking));
    } else if (!oldTicking && newTicking) {
        this.sections.forEach(section -> section.getEntities().forEach(listener::startTicking));
    }
}
```

The listener here is `ServerWorld.ServerEntityHandler`, which actually calls TACS's `loadEntity()` / `unloadEntity()` and entity system's `startTick()` / `stopTick()`.

7.3 WorldChunk.loadEntities(): Mounting Entities from NBT

When `WorldChunk` is created, the constructor saves an `EntityLoader`:

```java
public WorldChunk(ServerWorld world, ProtoChunk protoChunk, ...) {
    ...
    if (protoChunk instanceof ReadOnlyChunk) {
        this.entityLoader = ((ReadOnlyChunk)protoChunk).getEntityLoader();
    } else {
        this.entityLoader = null;
    }
}
```

`ReadOnlyChunk` is the result from `SerializedChunkDataLoader.loadChunk()`, which wraps the entity NBT list read from the region file. When the chunk is actually mounted (`makeChunkAccessible`), TACS calls `chunk.loadEntities()`:

```java
public void loadEntities() {
    if (this.entityLoader != null) {
        this.entityLoader.run(this);
        this.entityLoader = null;
    }
}
```

This loader ultimately calls `addEntitiesFromNbt()`:

```java
private static void addEntitiesFromNbt(ServerWorld world, List<NbtCompound> nbt) {
    if (!nbt.isEmpty()) {
        world.addEntities(EntityType.streamFromNbt(nbt, world));
    }
}
```

`ServerWorld.addEntities()` then delivers the entity stream to `ServerEntityManager.loadEntities()`. Therefore, `WorldChunk` here is more like a "transfer station triggering mounting": it holds entity NBT from chunk NBT, but entities actually entering the running, query, save, and tracking systems is completed in `ServerEntityManager`. Correspondingly, `ServerWorld.unloadEntities(WorldChunk chunk)` only cleans up block entities and tick scheduler on `WorldChunk`:

```java
public void unloadEntities(WorldChunk chunk) {
    chunk.clear();
    chunk.removeChunkTickSchedulers(this);
}
```

The method name here is misleading: it doesn't directly retrieve and unload regular entities from `WorldChunk`, because regular entities weren't stored in `WorldChunk`'s block array to begin with; the actual entity save and unload is still completed by `ServerEntityManager`'s `pendingUnloads`, `trySave()`, and `unload()`.

8 References

- `net.minecraft.server.world.ServerEntityManager`
- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.world.chunk.WorldChunk`
- `net.minecraft.server.world.ServerWorld`
- `net.minecraft.world.entity.EntityTrackingStatus`

5.10 Save and Unload Pipeline

Chapter 7 explained that completing the three `ChunkHolder` futures as `UNLOADED` only closes the corresponding runtime views. Players can no longer access a ticking `WorldChunk` through these futures, but the data may still remain in memory, waiting to be saved, have entities removed, light state released, and be removed from the table structure.

This chapter covers the second half: before a chunk truly leaves memory, how `ThreadedAnvilChunkStorage` (hereafter **TACS**) places it into a three-level unload structure, determines whether it needs to write to disk, converts the in-memory `Chunk` to NBT, and finally hands it to the `.mca` file system.

1 Review: "Cooling" ≠ "Unloading"

Cooling is a **runtime state change**; unloading is a **resource reclamation process**. The former occurs in `ChunkHolder.tick()`: `accessibleFuture`, `tickingFuture`, and `entityTickingFuture` complete based on level changes. The latter occurs in TACS's unload-save pipeline: chunks are marked, moved from the current table to the pending unload table, and finally saved and cleared from memory by async tasks.

Therefore, seeing `UNLOADED` doesn't mean "data has disappeared." It's more like a door being closed: external access stops, but the contents inside still need to be inventoried, packed, and moved in order.

2 Three-Level Unload Structure

TACS maintains three data structures related to unloading:

```java
// ThreadedAnvilChunkStorage.java (Yarn 1.20.1+build.10)
final LongSet unloadedChunks = new LongOpenHashSet();
private final Long2ObjectLinkedOpenHashMap<ChunkHolder> chunksToUnload = new
Long2ObjectLinkedOpenHashMap<>();
private final Queue<Runnable> unloadTaskQueue = Queues.newConcurrentLinkedQueue();
```

The first level is `unloadedChunks`, the **unloaded chunk set**. It only stores `long` coordinates, not `ChunkHolder`. When `setLevel()` finds that a new chunk level is no longer accessible, it immediately adds the coordinate here:

```java
if (holder != null) {
    if (!ChunkLevels.isAccessible(level)) {
        this.unloadedChunks.add(pos);
    } else {
        this.unloadedChunks.remove(pos);
    }
}
```

This layer serves as "coordinate markers for chunks needing to enter the unload process." It's lightweight, only recording the fact: this position has dropped below the range where TACS should continue to maintain a `ChunkHolder` externally.

The second level is `chunksToUnload`, the **pending unload chunk mapping**. When `unloadChunks()` processes `unloadedChunks`, it removes the `ChunkHolder` from `currentChunkHolders` and puts it into `chunksToUnload`:

```java
ChunkHolder chunkHolder = this.currentChunkHolders.remove(pos);
if (chunkHolder != null) {
    this.chunksToUnload.put(pos, chunkHolder);
    this.chunkHolderListDirty = true;
    this.tryUnloadChunk(pos, chunkHolder);
}
```

The key here is "temporarily storing the complete `ChunkHolder`." When a chunk just leaves the current table, the save future may not yet be stable, and async generation or conversion tasks may have just completed. Discarding the holder immediately would leave these trailing tasks without a landing point. More importantly, if the same coordinate is quickly pulled back to accessible by the ticket system, `setLevel()` will first retrieve the old holder from `chunksToUnload`:

```java
holder = this.chunksToUnload.remove(pos);
if (holder != null) {
    holder.setLevel(level);
} else {
    holder = new ChunkHolder(new ChunkPos(pos), level, ...);
}
```

This is reuse: when players walk back and forth across chunk boundaries, the chunk doesn't need to create a new `ChunkHolder` every time, nor reassemble the same intermediate state after unloading and immediately reloading.

The third level is `unloadTaskQueue`, the **unload task queue**. `tryUnloadChunk()` doesn't save and clean up directly in the current call stack, but passes the callback of `holder.getSavingFuture()` to this queue:

```java
completableFuture.thenAcceptAsync(chunk -> {
// Save, remove entities, refresh light state, etc.
}, this.unloadTaskQueue::add);
```

Then `unloadChunks()` retrieves and executes a batch of `Runnable` from the queue each tick. The real trailing work happens here: `WorldChunk.setLoadedToWorld(false)`, `save(chunk)`, `world.unloadEntities(worldChunk)`, `lightingProvider.updateChunkStatus()`, `chunkToNextSaveTimeMs.remove()`, etc.

In summary, the three-level structure separates "marking that unload is needed," "temporarily holding state to allow reuse or future completion," and "actual resource cleanup and disk write." This avoids immediately discarding holders on level drop or immediately blocking the main thread waiting for file I/O.

3 Save Throttling

TACS doesn't save chunks unconditionally every time. After each chunk save, it records a timestamp, and the next save for the same chunk must wait at least 10 seconds. This throttling prevents short-term repeated saves from consuming too many I/O operations and avoids write amplification in areas with high activity.

The implementation uses a `Long2LongOpenHashMap chunkToNextSaveTimeMs` to map chunk positions to the next allowed save time. Before `save(ChunkHolder)` triggers serialization, it checks:

```java
long l = chunk.getPos().toLong();
long m = this.chunkToNextSaveTimeMs.getOrDefault(l, -1L);
long n = System.currentTimeMillis();
if (n < m) {
    return false;
}
```

If the current time hasn't reached the recorded cooldown, save is skipped and `false` is returned. Only after a successful save does the code update the next save time to `n + 10000L` (10 seconds later).

This throttling only applies to the incremental save path in `save(ChunkHolder)`. During normal shutdown, `save(flush=true)` bypasses the cooldown and forces all dirty chunks to be written.

4 When Save Is Triggered

Chunk saves aren't only initiated at server shutdown. During normal runtime, TACS saves chunks through multiple paths:

1. **Incremental saves during tick**: `unloadChunks()` saves up to 20 dirty chunks per tick.
2. **Auto-save every 6000 ticks**: `ServerChunkManager.save(boolean suppressLogs)` with `flush=false`, doesn't wait for I/O completion.
3. **Manual** `/save-all`: Same auto-save path, but user-initiated.
4. **Save before unload**: `save(Chunk)` is called in the callback of `tryUnloadChunk()`.
5. **Normal shutdown**: `save(false, true, false)` with `flush=true`, bypasses cooldown, waits for all async writes, and flushes to disk.

Shutdown save is most aggressive, with three key differences:

- `flush=true` calls `StorageIoWorker.completeAll(true)`, waits for all `CompletableFuture`, and force-flushes the file channel.
- Iterates over all holders, ignoring cooldown.
- Calls `pointOfInterestStorage.saveChunk()` and waits for POI writes to complete.

Auto-save and incremental saves use `flush=false`, meaning NBT tasks are submitted to `StorageIoWorker`, but the main thread doesn't wait for write completion. This avoids blocking normal tick progression.

5 ChunkSerializer: From Object to NBT

Chunk serialization is handled by `ChunkSerializer`, which provides two static methods:

```java
public static ChunkSerializer.ProtoChunkData deserialize(
    ServerWorld world,
    StructureTemplateManager structureTemplateManager,
    PointOfInterestStorage poiStorage,
    ChunkPos expectedPos,
    NbtCompound nbt
)
public static NbtCompound serialize(
    ServerWorld world,
    Chunk chunk
)
```

`serialize()` converts an in-memory `Chunk` into an `NbtCompound`:

- Writes block states for all sections (including empty sections if they contain non-air blocks)
- Serializes all block entities (chest inventories, furnace states, etc.)
- Serializes scheduled ticks (`block_ticks` and `fluid_ticks`)
- Writes heightmaps, biome palette, light data, entities (for `ProtoChunk`), and structure references

The serialized NBT is the complete snapshot that will be written to the region file. Note that POI (Point of Interest) is saved separately through `PointOfInterestStorage` and doesn't go through `ChunkSerializer`—Chapter 12 will cover this bypass path.

`deserialize()` does the reverse: reads NBT and reconstructs a `ProtoChunk` (for chunks not yet at `FULL` status) or wraps upgrade data into `ProtoChunkData`. If the chunk on disk is at `FULL` status, TACS will directly construct a `WorldChunk` from the deserialized data.

6 StorageIoWorker: Async Write Queue

`ChunkSerializer` only handles memory-to-NBT conversion. The actual file I/O is delegated to `StorageIoWorker`, which manages async write tasks.

`ThreadedAnvilChunkStorage` has one worker per dimension:

```java
private final StorageIoWorker worker;
```

When `save(Chunk)` needs to write a chunk, it calls:

```java
private boolean save(Chunk chunk) {
    this.pointOfInterestStorage.saveChunk(chunk.getPos());
    if (!chunk.needsSaving()) {
        return false;
    }
    chunk.setNeedsSaving(false);
    ChunkPos chunkPos = chunk.getPos();
    try {
```

```
        ChunkStatus chunkStatus = chunk.getStatus();
        if (chunkStatus.getChunkType() != ChunkType.LEVELCHUNK) {
            // ...
        }
        NbtCompound nbtCompound = ChunkSerializer.serialize(this.world, chunk);
        this.setNbt(chunkPos, nbtCompound);
        return true;
    } catch (Exception e) {
        // ...
    }
}
```

`setNbt()` pushes the NBT write task to `worker`:

```java
protected void setNbt(ChunkPos pos, NbtCompound nbt) {
this.worker.setResult(pos, nbt);
}
```

Inside `StorageIoWorker`, `setResult()` wraps a `CompletableFuture` representing the write task and adds it to the queue. The worker thread continuously polls tasks and writes them through `RegionBasedStorage` to the region file. If `completeAll(waitForResult=false)` is called, tasks are only submitted but not waited on; if `waitForResult=true`, the caller blocks until all futures complete and the file channel is flushed.

This design separates NBT serialization (done on the main thread) from file I/O (offloaded to worker threads). The main thread doesn't block on I/O during normal operation; only during shutdown does it synchronously wait for all writes to finish.

7 Unload Task Execution

The trailing work of chunk unloading is handled by `unloadTaskQueue`. When `tryUnloadChunk()` schedules a save callback, it includes these cleanup steps:

```java
completableFuture.thenAcceptAsync(chunk -> {
    CompletableFuture<?> savingFuture = CompletableFuture.completedFuture(chunk);
    if (chunk instanceof WorldChunk worldChunk) {
        worldChunk.setLoadedToWorld(false);
        savingFuture = this.save(chunk) ? CompletableFuture.completedFuture(chunk)
                                        : this.getUnloadedChunkFuture();
    }
    // Save, unload entities, update light state
    this.world.unloadEntities(worldChunk);
    this.lightingProvider.updateChunkStatus(pos);
    this.chunkToNextSaveTimeMs.remove(pos);
    this.chunksToUnload.remove(pos);
}, this.unloadTaskQueue::add);
```

Each tick, `unloadChunks()` drains up to 20 runnables from the queue (controlled by `shouldKeepTicking` to avoid timeout):

```java
while (this.unloadTaskQueue.size() > 0 && shouldKeepTicking.getAsBoolean()) {
    Runnable runnable = this.unloadTaskQueue.poll();
```

```
    if (runnable != null) {
        runnable.run();
    }
}
```

Tasks run on the main thread since they involve world state (unloading entities, updating light). After completion, the chunk holder is removed from `chunksToUnload`, and the position is freed.

8 Save vs. Unload: Two Parallel Concerns

It's crucial to distinguish: **save and unload are not atomic**. Unload refers to removing from memory; save refers to writing to disk. A chunk may be saved without being unloaded (incremental save during runtime), or unloaded without being saved (if marked as not needing save, e.g., `needsSaving() == false`).

`save(Chunk)` checks `chunk.needsSaving()` before serializing. Only if the chunk is dirty (block states, block entities, or scheduled ticks changed) does it proceed with NBT conversion and submit to `StorageIoWorker`. If not dirty, save is skipped and the method returns `false`.

Unload, on the other hand, is controlled by level changes and ticket expiry. Even if a chunk hasn't changed since last save, when tickets drop and the level becomes inaccessible, the chunk still enters `unloadedChunks` → `chunksToUnload` → `unloadTaskQueue` to release memory.

During shutdown save, since `flush=true`, the server bypasses cooldown, waits for all write futures, and force-flushes channels. This ensures dirty state is committed to disk before process exit. But during normal runtime auto-save or incremental save, `flush=false` means the main thread submits write tasks without waiting—write completion is fire-and-forget.

9 Summary

- Unload is not immediate level-drop eviction but a three-stage pipeline: mark → pending table → async cleanup queue.
- TACS maintains `unloadedChunks`, `chunksToUnload`, and `unloadTaskQueue` to manage the unload lifecycle.
- Save throttling uses a 10-second cooldown per chunk, skipping saves if the cooldown hasn't expired; shutdown save bypasses this cooldown.
- `ChunkSerializer` converts `Chunk` to NBT; `StorageIoWorker` handles async file I/O.
- Auto-save uses `flush=false` (fire-and-forget); shutdown save uses `flush=true` (wait for all writes and force flush).
- Save and unload are orthogonal: a chunk can be unloaded without being saved (if not dirty), or saved without being unloaded (incremental save).

10 Code Walkthrough

10.1 unloadChunks: Top-Level Orchestration

`unloadChunks()` is called every server tick from `tick()` and orchestrates all three unload levels:

```java
private void unloadChunks(BooleanSupplier shouldKeepTicking) {
    LongIterator longIterator = this.unloadedChunks.iterator();
    for (int i = 0; longIterator.hasNext() && (i < 200 || shouldKeepTicking.getAsBoolean()); ++i)
    {
        long pos = longIterator.nextLong();
        longIterator.remove();
        ChunkHolder chunkHolder = this.currentChunkHolders.remove(pos);
        if (chunkHolder != null) {
            this.chunksToUnload.put(pos, chunkHolder);
            this.chunkHolderListDirty = true;
            this.tryUnloadChunk(pos, chunkHolder);
        }
    }
    int k = 0;
    ObjectIterator<ChunkHolder> objectIterator = this.chunkHolders.values().iterator();
    while (k < 20 && shouldKeepTicking.getAsBoolean() && objectIterator.hasNext()) {
        if (this.save(objectIterator.next())) {
            k++;
        }
    }
    Runnable runnable;
    while (this.unloadTaskQueue.size() > 0 && shouldKeepTicking.getAsBoolean()) {
        runnable = this.unloadTaskQueue.poll();
        if (runnable != null) {
            runnable.run();
        }
    }
}
```

Phase 1 moves marked chunks from `currentChunkHolders` to `chunksToUnload` and schedules their save futures. Phase 2 performs incremental saves on up to 20 dirty holders. Phase 3 drains the unload task queue, executing finalization runnables on the main thread.

10.2 tryUnloadChunk: Scheduling Save and Cleanup

```java
private void tryUnloadChunk(long pos, ChunkHolder holder) {
    CompletableFuture<Chunk> completableFuture = holder.getSavingFuture();
    completableFuture.thenAcceptAsync(chunk -> {
        CompletableFuture<?> savingFuture = CompletableFuture.completedFuture(chunk);
        if (chunk instanceof WorldChunk worldChunk) {
            worldChunk.setLoadedToWorld(false);
            savingFuture = this.save(chunk) ? CompletableFuture.completedFuture(chunk)
                                            : this.getUnloadedChunkFuture();
        }
        this.world.unloadEntities(worldChunk);
        this.lightingProvider.updateChunkStatus(ChunkPos.fromLong(pos), null);
        this.chunkToNextSaveTimeMs.remove(pos);
```

```
        this.chunksToUnload.remove(pos);
    }, this.unloadTaskQueue::add);
}
```

The callback waits for `getSavingFuture()` to complete (ensuring generation or upgrade is finished), then saves if the chunk is a `WorldChunk`, unloads entities, clears light state, removes cooldown entry, and finally removes the holder from `chunksToUnload`.

10.3 save(Chunk): Serialization and Worker Submission

```java
private boolean save(Chunk chunk) {
    this.pointOfInterestStorage.saveChunk(chunk.getPos());
    if (!chunk.needsSaving()) {
        return false;
    }
    chunk.setNeedsSaving(false);
    ChunkPos chunkPos = chunk.getPos();
    try {
        ChunkStatus chunkStatus = chunk.getStatus();
        if (chunkStatus.getChunkType() != ChunkType.LEVELCHUNK) {
            if (this.shouldSkipSave(chunkStatus)) {
                return false;
            }
            NbtCompound nbtCompound = ChunkSerializer.serialize(this.world, chunk);
            this.setNbt(chunkPos, nbtCompound);
            return true;
        }
        NbtCompound nbtCompound = ChunkSerializer.serialize(this.world, chunk);
        this.setNbt(chunkPos, nbtCompound);
        return true;
    } catch (Exception e) {
        // ...
    }
}
```

First saves POI via `pointOfInterestStorage.saveChunk()`. Then checks `needsSaving()`—if false, skip. If true, serializes with `ChunkSerializer.serialize()` and submits NBT to `StorageIoWorker` via `setNbt()`.

10.4 StorageIoWorker.completeAll: Flush vs. Fire-and-Forget

```java
public CompletableFuture<Void> completeAll(boolean sync) {
    CompletableFuture<Void> completableFuture = CompletableFuture.allOf(
        this.results.values().toArray(new CompletableFuture[0])
    );
    if (sync) {
        return completableFuture.thenCompose(v -> this.run(() -> {
            try {
                this.storage.sync();
            } catch (Exception e) {
                LOGGER.error("Failed to synchronize chunks", e);
            }
        }));
```

```
    }
    return completableFuture.thenCompose(v -> this.run(() -> {}));
}
```

When `sync=true`, the method waits for all write futures and calls `storage.sync()`, which flushes the underlying `FileChannel`. When `sync=false`, it just waits for all futures without forcing a flush—the OS may buffer writes.

11 References

- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.world.chunk.ChunkSerializer`
- `net.minecraft.world.storage.StorageIoWorker`
- `net.minecraft.server.world.ChunkHolder`
- `net.minecraft.world.chunk.WorldChunk`
- `net.minecraft.world.chunk.ProtoChunk`
- `net.minecraft.world.storage.RegionBasedStorage`
- `net.minecraft.world.poi.PointOfInterestStorage`

5.11 POI and Bypass Storage

Previous chapters covered chunk loading, cooling, saving, and unloading, but there's one type of data that runs alongside chunks yet doesn't live in regular chunk NBT: `PointOfInterestStorage`. It stores "points of interest" frequently queried by AI and world logic, known as POI.

This chapter fills in this bypass path: why POI is stored independently, how it's structured in memory, how block changes update it, and how it participates in villager, raid, and wandering trader logic.

1 What is POI

`Point of Interest` (POI) refers to "block positions queryable by AI or world logic": beds, job sites, bells, beehives, nether portals, lodestones, lightning rods, etc. can all be registered as POI. It's not part of `BlockState` itself, nor is it queried alongside regular chunk block states; it's an index extracted from block state. When villagers look for beds and job sites, when the raid system determines village centers, or when wandering traders find meeting points, they query `PointOfInterestStorage` rather than scanning nearby blocks.

2 Why POI Requires Independent Storage

If every time a villager wants to find a job site, the server scanned every block's `BlockState` within a range of several dozen blocks, it would waste massive time on repeated traversal. A 48-block horizontal radius already spans multiple chunks; multiply that by vertical sections, villager count, and per-tick AI scheduling, and this "scan-on-demand" cost quickly spirals out of control.

POI storage converts the problem into an index query. `PointOfInterestStorage` organizes `PointOfInterestSet` by chunk section, with each set internally storing POI by type. During queries, code first derives which chunks to check from the radius, then only traverses POI already registered in those chunks, finally filtering by type, position predicate, and `OccupationStatus`. In other words, the server queries "what known points of interest are nearby," not "is every nearby block a point of interest."

This is why it requires independent storage. Regular chunk block data is written to the `region/` subdirectory under the dimension directory; POI data is written to the `poi/` subdirectory under the same dimension directory. TACS sets up this bypass path during construction:

```java
// ThreadedAnvilChunkStorage.java (Yarn 1.20.1+build.10)
super(session.getWorldDirectory(world.getRegistryKey()).resolve("region"), dataFixer, dsync);
Path path = session.getWorldDirectory(world.getRegistryKey());
this.pointOfInterestStorage = new PointOfInterestStorage(path.resolve("poi"), dataFixer, dsync, dynamicRegistryManager, world);
```

Thus, a single dimension has at least two sets of persistent data related to chunk coordinates: `region/` stores blocks, terrain, block entities, and scheduled ticks; `poi/` stores the point-of-interest index. Chapter 11 mentioned that when TACS saves a chunk, it first calls

`pointOfInterestStorage.saveChunk(chunk.getPos())` to keep this bypass path in sync with the regular chunk save rhythm.

3 POI Data Structure

`PointOfInterestStorage` is the top-level manager. It extends `SerializingRegionBasedStorage<PointOfInterestSet>`, with the parent class handling loading, caching, marking dirty, and writing back region files by `ChunkSectionPos`; the child class handles POI queries, distance propagation, and scanning block states from chunk sections. Although the file directory saves by chunk, the memory index key is actually the section coordinate: `ChunkSectionPos.toLong(pos)`.

`PointOfInterestSet` is the set of all POI within a single chunk section. It has two tables: `pointsOfInterestByPos` uses section-local coordinates to quickly find the POI at a position; `pointsOfInterestByType` groups by `RegistryEntry<PointOfInterestType>`, facilitating type-based streaming queries. This class name can be misleading ("one set per chunk"), but in 1.20.1 source, it corresponds to one section; a chunk's POI file will save multiple y sections under `Sections`.

`PointOfInterest` is a single POI record, containing block position `pos`, interest point type `type`, free tickets `freeTickets`, and an update callback `updateListener`. Occupation status isn't stored as a separate boolean, but derived from ticket count: `hasSpace()` checks `freeTickets > 0`, `isOccupied()` checks `freeTickets != type.value().ticketCount()`. In Yarn 1.20.1+build.10, `PointOfInterest` has no standalone "last reference time" field; an occupation or release modifies `freeTickets`, then triggers `updateListener.run()` to mark for save.

`PointOfInterestType` describes "which `BlockState` belong to the same POI type, how many objects can occupy it, and what search distance is allowed." Strictly speaking, it's not a Java `enum`, but a record registered in `RegistryKeys.POINT_OF_INTEREST_TYPE`; `PointOfInterestTypes` is the interest point type registry, responsible for registering `HOME`, `MEETING`, profession job sites, `NETHER_PORTAL`, `LODESTONE`, `LIGHTNING_ROD`, etc., and maintaining the reverse mapping `BlockState -> RegistryEntry<PointOfInterestType>`. If you remember "doors" from the old village system, note that Yarn 1.20.1+build.10's default `PointOfInterestTypes` no longer registers doors as POI.

4 How Block Changes Affect POI

POI indices aren't static after generation. Every time the server detects a block state change, `ServerWorld.onBlockChanged()` checks whether the old and new `BlockState` correspond to POI types:

```java
Optional<RegistryEntry<PointOfInterestType>> oldType = PointOfInterestTypes.getTypeForState(oldBlock);
Optional<RegistryEntry<PointOfInterestType>> newType = PointOfInterestTypes.getTypeForState(newBlock);
if (!Objects.equals(oldType, newType)) {
    oldType.ifPresent(type -> this.getServer().execute(() -> this.getPointOfInterestStorage().remove(blockPos)));
    newType.ifPresent(type -> this.getServer().execute(() -> this.getPointOfInterestStorage().add(blockPos, type)));
}
```

Placing a POI block, like a loom, matches the new `BlockState` to the `SHEPHERD` type in `PointOfInterestTypes`, then enters `PointOfInterestStorage.add(pos, type)`. `add()` derives the section coordinate from the block position, retrieves or creates the corresponding `PointOfInterestSet`, and puts the new `PointOfInterest` into both the position index and type index. When destroying this block, the old state still matches the POI type, the new state doesn't, so it calls `remove(pos)`, removing the record from the same section's set.

Adding, removing, occupying, and releasing all trigger update callbacks. The source doesn't have a public method directly called `markDirty()`; this role is fulfilled by `SerializingRegionBasedStorage.onUpdate(long pos)`. After `PointOfInterestSet` receives a change, it runs `updateListener`, and the parent class adds the section's long coordinate to `unsavedElements`. Later, TACS's `tick()` calls `pointOfInterestStorage.tick(shouldKeepTicking)`, or when TACS saves a chunk it calls `pointOfInterestStorage.saveChunk(chunk.getPos())`, serializing dirty sections under the same chunk into `poi/`.

Another easily overlooked path is preloading. `PointOfInterestStorage.preloadChunks(world, pos, radius)` checks whether POI sections are valid within a specified range; if invalid, it adds the corresponding chunk coordinate to `preloadedChunks` and triggers minimum-status loading with `world.getChunk(chunkX, chunkZ, ChunkStatus.EMPTY)`. This isn't to generate complete chunks, but to get frequently queried POI data into memory early; typical callers include portal search logic that repeatedly queries POI.

5 POI and Villagers/Raid System

When villagers look for job sites, beds, or meeting points, they typically don't read blocks directly, but ask `PointOfInterestStorage` to find positions satisfying type and occupation conditions within range. For example, "find nearest POI of a certain type" goes through `getNearestPosition(typePredicate, pos, radius, occupationStatus)`; when reserving a job site, it may go through `getPosition(...)` with `HAS_SPACE`, then call `reserveTicket()` after finding one to occupy a ticket.

The raid system also uses POI as input for village determination. `RaidManager.startRaid()` queries all POI within a 64-block radius around the player that belong to `PointOfInterestTypeTags.VILLAGE` and have `OccupationStatus.IS_OCCUPIED`, averages these positions, and uses that as the raid center. The "village" here isn't a separate Village object saved to disk, but derived at runtime from occupied POI like beds and bells.

Wandering traders seek meeting points. `WanderingTraderManager` uses `getPosition(poiType -> poiType.matchesKey(PointOfInterestTypes.MEETING), pos -> true, blockPos, 48, OccupationStatus.ANY)` to find `MEETING` POI (bells) within 48 blocks; if none found, it falls back to the player's position. The three values of `OccupationStatus` are straightforward: `ANY` means ignore occupation, `HAS_SPACE` means free tickets remain, `IS_OCCUPIED` means at least one object has occupied it.

6 POI in the Chunk Lifecycle

POI isn't part of regular chunks, but it's tightly integrated with the chunk lifecycle. When a chunk section is first loaded or the palette needs to be filled, `PointOfInterestStorage.initForPalette(sectionPos, chunkSection)` first uses `shouldScan(chunkSection)` to determine whether this section contains any POI block states; if so, it scans every position in the section and fills types found by `PointOfInterestTypes.getTypeForState(blockState)` into the corresponding `PointOfInterestSet`. This step converts real block states into the bypass index.

During runtime, block changes cause POI additions/deletions; AI queries read POI and may modify ticket counts; TACS tick advances POI dirty data saves; when TACS saves a chunk, it first calls `saveChunk()` as a fallback. The shutdown check also includes `pointOfInterestStorage.hasUnsavedElements()`, indicating that as long as POI still has unwritten sections, the server cannot consider this dimension fully finalized.

POI can be understood as "an AI index table alongside chunks": it derives from chunk block states, saves by chunk coordinates, but queries, occupation, and disk writes all have their own lifecycle. Regular chunks answer "what block is here"; POI quickly answers "what AI-relevant points are nearby."

7 Source Code Walkthrough

7.1 PointOfInterestStorage Constructor

The `PointOfInterestStorage` constructor is very short:

```java
public PointOfInterestStorage(Path path, DataFixer dataFixer, boolean dsync,
DynamicRegistryManager registryManager, HeightLimitView world) {
    super(path, PointOfInterestSet::createCodec, PointOfInterestSet::new, dataFixer,
    DataFixTypes.POI_CHUNK, dsync, registryManager, world);
    this.pointOfInterestDistanceTracker = new
    PointOfInterestStorage.PointOfInterestDistanceTracker();
}
```

The important parts are hidden inside the `super(...)` arguments. `path` is the `path.resolve("poi")` passed in by TACS, determining the disk directory. `PointOfInterestSet::createCodec` determines how each section is serialized. `PointOfInterestSet::new` is the factory for missing sections. `DataFixTypes.POI_CHUNK` means old-version POI NBT uses the dedicated POI data-fixer type. `registryManager` lets the codec resolve `PointOfInterestType` registry entries. `world` provides the height range, telling it which sections a chunk contains.

The parent class `SerializingRegionBasedStorage` creates its own `StorageIoWorker`:

```java
this.worker = new StorageIoWorker(path, dsync, path.getFileName().toString());
```

So the async I/O worker for POI is not the same object as the `VersionedChunkStorage` used by ordinary chunks. TACS merely holds both and advances both during saving and ticking.

At the end of the constructor, it creates `PointOfInterestDistanceTracker`. This is a `SectionDistanceLevelPropagator` used to maintain "how far a section is from the nearest occupied village POI." `ServerWorld.isNearOccupiedPointOfInterest()` ultimately reads this distance. It serves village-related checks, but it is still only a POI distance cache, not full villager AI.

7.2 Internal Logic of add() / remove()

`PointOfInterestStorage.add()` itself only performs one routing step:

```java
public void add(BlockPos pos, RegistryEntry<PointOfInterestType> type) {
this.getOrCreate(ChunkSectionPos.toLong(pos)).add(pos, type);
}
```

`ChunkSectionPos.toLong(pos)` packs the block position into section coordinates. `getOrCreate()` comes from the parent class. If the section is already loaded, it returns the existing `PointOfInterestSet`. If the disk has no POI data for that section, it creates an empty set with `PointOfInterestSet::new` and binds the update callback to `onUpdate(pos)`.

The actual insertion happens in `PointOfInterestSet.add()`:

```java
public void add(BlockPos pos, RegistryEntry<PointOfInterestType> type) {
    if (this.add(new PointOfInterest(pos, type, this.updateListener))) {
        this.updateListener.run();
    }
}
```

The internal private `add(PointOfInterest poi)` first uses `ChunkSectionPos.packLocal(blockPos)` to get the section-local short coordinate. If the same local position already has a POI of the same type, it returns `false` to avoid repeated dirty marking. If a POI already exists there but with a different type, it records a "POI data mismatch". On success, it writes to both `pointsOfInterestByPos` and `pointsOfInterestByType`.

`remove()` is also split into two layers:

```java
public void remove(BlockPos pos) {
this.get(ChunkSectionPos.toLong(pos)).ifPresent(poiSet -> poiSet.remove(pos));
}
```

It does not create a set unconditionally. It only removes when the corresponding section exists. `PointOfInterestSet.remove(pos)` removes the entry from `pointsOfInterestByPos` using the local short coordinate, then removes it from the corresponding typed set in `pointsOfInterestByType`. On successful deletion it runs `updateListener`. If no record is found, it logs a "never registered" error.

Occupation state also uses the same save chain. `reserveTicket()` decreases `freeTickets`, and `releaseTicket()` increases `freeTickets`. Both call `updateListener.run()`. Therefore, a logic change such as "a villager occupied a workstation" also makes the POI section dirty even though the block itself did not change.

7.3 Save Mechanism Triggered by markDirty()

In this set of source files, the "mark dirty" action for POI is not a public method called `markDirty()`, but `SerializingRegionBasedStorage.onUpdate(long pos)`:

```java
protected void onUpdate(long pos) {
    Optional<R> optional = this.loadedElements.get(pos);
    if (optional != null && optional.isPresent()) {
        this.unsavedElements.add(pos);
    } else {
        LOGGER.warn("No data for position: {}", ChunkSectionPos.from(pos));
    }
}
```

`PointOfInterestSet`'s `updateListener` ultimately calls this. `unsavedElements` stores section coordinates rather than chunk coordinates. This lets POI know exactly which section changed. During actual disk write, all sections under the same chunk are merged into one NBT:

```java
private <T> Dynamic<T> serialize(ChunkPos chunkPos, DynamicOps<T> ops) {
    Map<T, T> map = Maps.newHashMap();
    for (int y = this.world.getBottomSectionCoord(); y < this.world.getTopSectionCoord(); y++) {
        long sectionPos = chunkSectionPosAsLong(chunkPos, y);
        this.unsavedElements.remove(sectionPos);
        Optional<R> optional = this.loadedElements.get(sectionPos);
        if (optional != null && optional.isPresent()) {
            // Write Sections[y]
        }
    }
    // Return NBT containing Sections and DataVersion
}
```

There are two save entry points. The first is POI's own tick:

```java
protected void tick(BooleanSupplier shouldKeepTicking) {
    while (this.hasUnsavedElements() && shouldKeepTicking.getAsBoolean()) {
        ChunkPos chunkPos = ChunkSectionPos.from(this.unsavedElements.firstLong()).toChunkPos();
        this.save(chunkPos);
    }
}
```

TACS calls it every tick in the `poi` profiler section:

```java
profiler.push("poi");
this.pointOfInterestStorage.tick(shouldKeepTicking);
```

The second entry point is the fallback before regular chunk save:

```java
private boolean save(Chunk chunk) {
    this.pointOfInterestStorage.saveChunk(chunk.getPos());
    if (!chunk.needsSaving()) {
        return false;
    }
    // Regular chunk NBT save
}
```

`saveChunk(chunkPos)` checks whether any section of that chunk is in `unsavedElements`. As long as one section matches, it saves the whole POI chunk. Note that this happens before the `chunk.needsSaving()` check. So even if the regular chunk body has no dirty data, POI can still be written back. This is exactly the core of "bypass storage": POI is derived from chunks and advances alongside chunk coordinates, but its dirty marking and persistence logic exist independently.

8 References

- `net.minecraft.world.poi.PointOfInterestStorage`
- `net.minecraft.world.poi.PointOfInterestSet`
- `net.minecraft.world.poi.PointOfInterest`
- `net.minecraft.world.poi.PointOfInterestType`
- `net.minecraft.world.poi.PointOfInterestTypes`
- `net.minecraft.world.storage.SerializingRegionBasedStorage`
- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.server.world.ServerWorld`
- `net.minecraft.village.raid.RaidManager`
- `net.minecraft.world.WanderingTraderManager`

5.12 Client Visibility and Chunk Packets

Chapter 1 explained that client chunk data **comes entirely from server synchronization**. This chapter pursues a more specific question: when does the server decide "this player should see this chunk," and which packet does it use to send the changes?

1 Server Authoritative, Client Only a Snapshot

The terrain rendered by the client is not a locally authoritative world, but a snapshot of chunks sent by the server. The server's `WorldChunk` is the authoritative data source; after the client receives a `ChunkDataS2CPacket`, it only rebuilds a locally renderable and queryable copy in its `ClientWorld`.

This also explains why "can see" and "is simulating" are not the same thing. Client rendering is controlled by `watchDistance` (the chunk watch range related to player view distance); server block ticking and entity ticking are controlled by load levels and simulation distance covered in Chapter 3. Players may see distant terrain, but redstone, random ticks, or entity AI there are no longer running.

2 Types of Chunk Packets

The server doesn't synchronize chunks with "one big packet for everything." The heaviest type is `ChunkDataS2CPacket`, the **chunk data packet (S2C)**. When a chunk newly enters a player's view, `ThreadedAnvilChunkStorage.sendWatchPackets()` sends it via `sendChunkDataPackets()`. This packet carries the chunk's main client-visible data: block states, biomes, block entity data, and lighting data filled in by `ServerLightingProvider`. The client can then build the corresponding renderable chunk locally.

After entering view, most changes don't require resending the entire chunk. Single block changes typically use `BlockUpdateS2CPacket`, the **block update packet (S2C)**; when multiple block changes accumulate within the same sub-chunk, `ChunkHolder.flushUpdates()` switches to `ChunkDeltaUpdateS2CPacket`, the **chunk delta update packet (S2C)**. Light changes are sent separately via `LightUpdateS2CPacket`, the **light update packet (S2C)**, to players at the edge of watch distance. The division of labor among these packets is essentially bandwidth optimization: the first time entering view requires a complete snapshot, subsequent local changes only send differences.

> [!NOTE]
> After sending `ChunkDataS2CPacket`, `sendChunkDataPackets()` also refreshes entity tracking status for entities in that chunk to that player, and resends packets for leashes, passengers, etc. But these belong to the entity sync system, not chunk data packets themselves.

3 When to Send Chunk Data

When a player joins, leaves, or moves, TACS recalculates which chunks the player observes. Joins and leaves go through `handlePlayerAddedOrRemoved()`; moves go through `updatePosition()`. Both enumerate candidate chunks around the player's new and old `ChunkSectionPos` and use `isWithinDistance()` to determine whether a chunk is within the current `watchDistance`.

When a chunk changes from "not in view" to "in view," `sendWatchPackets()` retrieves the corresponding `ChunkHolder`, then its `WorldChunk`, and sends `ChunkDataS2CPacket`. When a chunk changes from "in view" to "not in view," the same method only calls `player.sendUnloadChunkPacket(pos)`, telling the client to discard the local copy of that chunk.

Block changes take a different path. `ChunkHolder.markForBlockUpdate()` doesn't send packets immediately, but packs the local coordinate into a `short` and puts it into `blockUpdatesBySection`. After each tick's subsequent refresh, `flushUpdates()` batch-sends these changes to players watching that chunk. This way, many adjacent block changes within the same tick can be merged into one chunk delta update, rather than creating many scattered small packets.

View distance changes also trigger recalculation. `setViewDistance()` first clamps the new value to 2-32, then updates the `ticketManager`'s watch distance, then iterates all current `ChunkHolder`. For each player watching that chunk, it calculates `isWithinDistance()` results under old and new distances; only boundary changes like "old in, new out" or "old out, new in" call `sendWatchPackets()`.

4 Geometric Determination of `watchDistance`

The `watchDistance` determination is not a simple Chebyshev distance. Load ticket propagation commonly uses "maximum horizontal or vertical offset" to form a square range, while `ThreadedAnvilChunkStorage.isWithinDistance()` uses a different geometry: it first shrinks the X/Z direction differences by 1, then uses an approximate Euclidean distance expression for comparison.

The core logic can be rewritten as:

```java
int dx = Math.max(0, Math.abs(chunkX - playerChunkX) - 1);
int dz = Math.max(0, Math.abs(chunkZ - playerChunkZ) - 1);
long longSide = Math.max(0, Math.max(dx, dz) - 1);
long shortSide = Math.min(dx, dz);
return shortSide * shortSide + longSide * longSide < distance * distance;
```

The initial 1-block subtraction gives a more lenient core area around the player's chunk; then `shortSide² + longSide² < distance²` cuts off corners. Therefore, the client watch range is more like a **rounded rectangle**, not the pure square of ticket propagation. This difference matters: both are called "distance," but `watchDistance` determines whether the client should keep a chunk snapshot, while ticket propagation determines what load level the server should push a chunk to.

5 Player Chunk Watching Manager

TACS holds a `PlayerChunkWatchingManager`, the **player chunk watching manager**. It records which players are currently watching each chunk and handles player watch center moves, watch disables, watch re-enables, and other states.

This manager doesn't directly generate chunk data packets, but it determines "who should receive them." `ChunkHolder.flushUpdates()` queries the observer list via `PlayersWatchingChunkProvider`: light updates query players at the edge of watch distance, block updates query all players watching that chunk. In other words, `PlayerChunkWatchingManager` is the index table for the final delivery targets of full packets, incremental packets, and light packets.

6 Summary

- Client chunks are snapshots synchronized from the server; client rendering and server simulation are controlled by different distance systems.
- `ChunkDataS2CPacket` is for full chunk sync when newly entering view, `BlockUpdateS2CPacket` and `ChunkDeltaUpdateS2CPacket` for block state changes, `LightUpdateS2CPacket` for light changes.
- Player joins, leaves, moves, and view distance changes all trigger watch range recalculation; only boundary changes entering or leaving watch range send full packets or unload packets.
- `watchDistance` uses a rounded-rectangle geometric determination, not equivalent to the Chebyshev distance in load ticket propagation.
- `PlayerChunkWatchingManager` tracks observers for each chunk and is the key index for deciding which players receive data packets.

7 Code Walkthrough

7.1 sendWatchPackets: Four Enter/Leave Cases

In `sendWatchPackets()` parameters, `oldWithinViewDistance` indicates whether the chunk was in the player's watch range before the call, `newWithinViewDistance` indicates whether it's in range after:

```java
protected void sendWatchPackets(
    ServerPlayerEntity player,
    ChunkPos pos,
    MutableObject<ChunkDataS2CPacket> packet,
    boolean oldWithinViewDistance,
    boolean newWithinViewDistance
) {
    if (player.getWorld() == this.world) {
        if (newWithinViewDistance && !oldWithinViewDistance) {
            // Case 1: Just entered watch range
            ChunkHolder chunkHolder = this.getChunkHolder(pos.toLong());
            if (chunkHolder != null) {
                WorldChunk worldChunk = chunkHolder.getWorldChunk();
                if (worldChunk != null) {
```

```
                    this.sendChunkDataPackets(player, packet, worldChunk);
                }
            }
        }
        if (!newWithinViewDistance && oldWithinViewDistance) {
            // Case 2: Just left watch range
            player.sendUnloadChunkPacket(pos);
        }
    }
}
```

Only case 1 sends the chunk data packet; case 2 sends an unload packet. The other two combinations (`newWithinViewDistance == oldWithinViewDistance`) represent "still in" or "still out," neither requiring watch status change.

7.2 sendChunkDataPackets: Lazy Packet Construction and Reuse

`sendChunkDataPackets()` doesn't immediately serialize the chunk into a packet. It uses a `MutableObject<ChunkDataS2CPacket>` to reuse the same packet across multiple players:

```java
private void sendChunkDataPackets(ServerPlayerEntity player, MutableObject<ChunkDataS2CPacket>
cachedDataPacket, WorldChunk chunk) {
    if (cachedDataPacket.getValue() == null) {
        cachedDataPacket.setValue(new ChunkDataS2CPacket(chunk, this.lightingProvider, null,
        null));
    }
    player.sendPacket(cachedDataPacket.getValue());
    // ...
    this.updateEntityTrackers(player, chunk.getPos());
}
```

When the first player needs this chunk, `ChunkDataS2CPacket` is constructed and cached in `cachedDataPacket`; subsequent players directly read the cached value. This avoids multiple serializations of the same chunk data within a single tick.

7.3 markForBlockUpdate: Local Coordinate Packing

`markForBlockUpdate()` uses `short` to pack block positions within a chunk:

```java
public void markForBlockUpdate(BlockPos pos) {
    if (this.pendingBlockUpdates != null) {
        int i = this.levelHeightAccessor.getSectionIndex(pos.getY());
        if (this.blockUpdatesBySection[i] == null) {
            this.blockUpdatesBySection[i] = new ShortOpenHashSet();
        }
        this.blockUpdatesBySection[i].add(ChunkSectionPos.packLocal(pos));
        this.pendingBlockUpdates = true;
    }
}
```

`packLocal(pos)` packs the chunk-local x, y, z coordinates (each 0-15) into a `short`: `(x & 15) << 8 | (y & 15) << 4 | (z & 15)`. This saves memory: each position only occupies 2 bytes, and accumulating thousands of changes within a tick doesn't create memory pressure.

7.4 flushUpdates: Batch Send Selection

`flushUpdates()` determines whether to send `BlockUpdateS2CPacket` or `ChunkDeltaUpdateS2CPacket` by section change count:

```java
public void flushUpdates(WorldChunk chunk) {
    if (this.pendingBlockUpdates) {
        ServerWorld world = (ServerWorld) chunk.getWorld();
        List<ServerPlayerEntity> list2 = this.getPlayersWatchingChunk(this.pos, false);
        for (int i = this.levelHeightAccessor.getBottomSectionCoord(); i <
        this.levelHeightAccessor.getTopSectionCoord(); ++i) {
            ShortSet shortSet = this.blockUpdatesBySection[i];
            if (shortSet != null) {
                this.blockUpdatesBySection[i] = null;
                if (!list2.isEmpty()) {
                    int j = this.levelHeightAccessor.sectionCoordToIndex(i);
                    ChunkSectionPos chunkSectionPos = ChunkSectionPos.from(this.pos, j);
                    if (shortSet.size() == 1) {
                        BlockPos blockPos =
                        chunkSectionPos.unpackBlockPos(shortSet.iterator().nextShort());
                        BlockState blockState = world.getBlockState(blockPos);
                        this.sendPacketToPlayers(list2, new BlockUpdateS2CPacket(blockPos,
                        blockState));
                        this.tryUpdateBlockEntityAt(list2, world, blockPos, blockState);
                    } else {
                        ChunkSection chunkSection = chunk.getSection(i);
                        ChunkDeltaUpdateS2CPacket chunkDeltaUpdateS2CPacket = new
                        ChunkDeltaUpdateS2CPacket(chunkSectionPos, shortSet, chunkSection);
                        this.sendPacketToPlayers(list2, chunkDeltaUpdateS2CPacket);
                        chunkDeltaUpdateS2CPacket.visitUpdates((pos, state) ->
                        this.tryUpdateBlockEntityAt(list2, world, pos, state));
                    }
                }
            }
        }
    }
    this.pendingBlockUpdates = false;
}
```

`list2` uses `getPlayersWatchingChunk(this.pos, false)`, all players watching this chunk. Each non-empty sub-chunk is judged separately: if `shortSet.size() == 1`, send `BlockUpdateS2CPacket`; if multiple positions changed in the same sub-chunk, send `ChunkDeltaUpdateS2CPacket`. Both paths check block entities: the single-block path directly calls `tryUpdateBlockEntityAt()`, the delta path iterates each changed position via `visitUpdates()`.

This design separates "change recording" and "network sending": block changes only do lightweight recording when they occur, and when actually sending packets, choose the most appropriate packet based on observers, sub-chunks, and change count.

8 References

- `net.minecraft.server.world.ThreadedAnvilChunkStorage`
- `net.minecraft.server.world.ChunkHolder`
- `net.minecraft.server.world.PlayerChunkWatchingManager`
- `net.minecraft.network.packet.s2c.play.ChunkDataS2CPacket`
- `net.minecraft.network.packet.s2c.play.ChunkDeltaUpdateS2CPacket`
- `net.minecraft.network.packet.s2c.play.LightUpdateS2CPacket`
- `net.minecraft.network.packet.s2c.play.BlockUpdateS2CPacket`

5.13 Game Event and In-Chunk Listeners

Previous chapters mainly discussed how chunks are loaded, run, saved, and unloaded. This chapter supplements a smaller but easily confused system: **game events** (`GameEvent`).

It often appears alongside sculk sensors, sculk shriekers, allays, and related mechanics, but this chapter doesn't focus on sculk gameplay itself, but rather how chunks save and manage these listeners.

Basics

- The difference between `GameEvent` and NC/PP updates
- Game event dispatchers organized by section in `WorldChunk`
- Registration, removal, and unload lifecycle of game event listeners
- The relationship between `VibrationListener` and the game event system

1 Game Event ≠ Block Update

NC updates and PP updates are notification mechanisms between blocks. They care about: after a block or neighboring position changes, should other blocks re-evaluate their own state? For example, should a piston extend, should an observer light up, should walls and fences update connection shapes. In other words, block updates handle "how blocks should change."

`GameEvent` is a different event system. It broadcasts "what happened in the world" to nearby listeners, but the event itself doesn't directly require blocks to change state. Common events include `GameEvent.BLOCK_PLACE`, `GameEvent.BLOCK_DESTROY`, `GameEvent.STEP`, `GameEvent.PROJECTILE_LAND`, `GameEvent.ENTITY_DAMAGE`. These events can be sensed by sculk sensors, sculk shriekers, or other listeners, which then decide whether to respond based on their own logic.

Therefore, don't understand `emitGameEvent()` as a new type of NC or PP update. Block update messages are "coarse": something changed here, check it yourself. Game event messages are more like recording facts: a `STEP` happened here, a `BLOCK_PLACE` happened here. The former serves block state propagation, the latter serves listener perception.

2 Event Dispatchers in WorldChunk

`WorldChunk` maintains a game event dispatcher mapping:

```java
private final Int2ObjectMap<GameEventDispatcher> gameEventDispatchers = new Int2ObjectOpenHashMap<>();
```

The key is the section Y coordinate, i.e., the vertical 16-block interval obtained by `ChunkSectionPos.getSectionCoord(y)`; the value is the `GameEventDispatcher` corresponding to that section. This indicates game event listeners aren't stuffed into one big list for the entire chunk, but are managed split by section.

When external code needs a dispatcher for a section, it calls:

```java
public GameEventDispatcher getGameEventDispatcher(int ySectionCoord) {
    return this.world instanceof ServerWorld serverWorld
        ? this.gameEventDispatchers.computeIfAbsent(
            ySectionCoord,
            sectionCoord -> new SimpleGameEventDispatcher(
                serverWorld,
                ySectionCoord,
                this::removeGameEventDispatcher
            )
        )
        : super.getGameEventDispatcher(ySectionCoord);
}
```

Server-side `WorldChunk` lazily creates `SimpleGameEventDispatcher`: a dispatcher appears in the mapping only when a section truly needs listeners. Client or non-server paths fall back to parent class logic, not maintaining server-side listener collections here.

One dispatcher per section has two intuitive benefits. First, when emitting events, you don't need to mix all listeners from the entire chunk and filter them; sections provide an initial spatial grouping. Second, listener addition and removal are concentrated in smaller lists, avoiding piling all chunk listeners onto a single dispatcher or making a single dispatcher a unified contention point.

`SimpleGameEventDispatcher`'s internal structure is lightweight:

```java
private final List<GameEventListener> listeners = Lists.newArrayList();
private final Set<GameEventListener> toRemove = Sets.newHashSet();
private final List<GameEventListener> toAdd = Lists.newArrayList();
private boolean dispatching;
```

`listeners` stores the current section's game event listeners. If an event is being dispatched, additions and removals don't immediately modify the list being traversed, but first enter `toAdd` / `toRemove`, then merge after dispatch ends. This is similar to the pending list approach for block entity tickers: avoid modifying the current collection during traversal.

3 Listener Registration and Lifecycle

Take sculk sensors as an example. When a sculk sensor is placed, the block state's corresponding block entity is created as `SculkSensorBlockEntity`. This block entity implements `GameEventListener.Holder<Vibrations.VibrationListener>`, and its `getEventListener()` returns its own `Vibrations.VibrationListener`. Sculk shriekers `SculkShriekerBlockEntity` follow the same pattern.

`WorldChunk` doesn't need to recognize every type of sculk block. It only queries through `BlockEntityProvider.getGameEventListener(ServerWorld, blockEntity)`: does this block entity provide a game event listener? The default implementation checks if the block entity is a `GameEventListener.Holder<?>`, and if so, extracts `holder.getEventListener()`.

Registration occurs in `WorldChunk.updateGameEventListener()`:

```java
private <T extends BlockEntity> void updateGameEventListener(T blockEntity, ServerWorld world) {
    Block block = blockEntity.getCachedState().getBlock();
    if (block instanceof BlockEntityProvider) {
        GameEventListener gameEventListener =
            ((BlockEntityProvider)block).getGameEventListener(world, blockEntity);
        if (gameEventListener != null) {
            this.getGameEventDispatcher(
                ChunkSectionPos.getSectionCoord(blockEntity.getPos().getY())
            ).addListener(gameEventListener);
        }
    }
}
```

This code registers the listener to "the section dispatcher corresponding to the block entity's Y coordinate." After `addListener(GameEventListener)`, the dispatcher will consider this listener during this section's event dispatch; whether it actually receives depends on the listener's position, range, tags, and callback logic.

Note this lifecycle is bound to the block entity. When a block entity joins a chunk, `WorldChunk.addBlockEntity()` calls `updateGameEventListener()` when the chunk has already entered world runtime state, then updates the block entity ticker. When the chunk fills block entities from NBT or network packets, `updateAllBlockEntities()` also re-registers listeners for existing block entities.

Removal takes the reverse path. When a block is destroyed, replaced, or a block entity is removed from a chunk, `WorldChunk.removeBlockEntity()` extracts the `GameEventListener` corresponding to that block entity, finds the `GameEventDispatcher` for its section, then calls:

```java
gameEventDispatcher.removeListener(gameEventListener);
```

After `SimpleGameEventDispatcher.removeListener()` removes the listener, if `listeners` is now empty, it calls the disposal callback passed during construction:

```java
if (this.listeners.isEmpty()) {
this.disposalCallback.apply(this.ySectionCoord);
}
```

In `WorldChunk`, this callback is precisely `removeGameEventDispatcher(int ySectionCoord)`:

```java
private void removeGameEventDispatcher(int ySectionCoord) {
this.gameEventDispatchers.remove(ySectionCoord);
}
```

Thus, dispatcher cleanup isn't a separate global table scan, but when the last listener in a section is removed, it removes that section's dispatcher from `gameEventDispatchers` on the side. Empty sections don't keep empty dispatchers.

Chunk unloading can also be understood following this logic. As Chapter 11 explained, unloading isn't an instant "evaporation" when `ChunkHolder` disappears, but TACS moving the holder from the current table to the pending unload structure, then executing save, entity unload, light state refresh, and other cleanup in unload tasks. For game event listeners, the key is that block entities must be removed or marked for removal

when exiting the running set with the chunk, and corresponding listeners should be unregistered from their section's dispatcher; when all listeners in a section leave, `removeGameEventDispatcher()` clears the dispatcher mapping entry.

This is also why `gameEventDispatchers` is placed on `WorldChunk`, not in a world-level global list: listener and chunk section lifecycles are naturally bound. When a chunk loads, block entities restore and listeners re-register; when block entities disappear or the chunk unloads, listeners are removed and empty dispatchers disappear with them.

4 Relationship with the Vibration System

`VibrationListener` is a specialized implementation of game event listeners. Sculk sensors and sculk shriekers use it to receive specific `GameEvent`, then decide whether to activate based on distance, frequency, tags, and their own state. Allays (`AllayEntity`) also use vibration-related listeners, but they're attached to the entity's `EntityGameEventHandler`, not in `WorldChunk`'s block entity listener mapping.

In other words, the vibration system isn't a separate parallel entry to `GameEvent`, but a set of rules built on top of game event listeners: which events can be heard, what transmission distance is allowed, how to convert frequency, and how listeners respond upon receiving. These vibration propagation and frequency classification details aren't covered in this chapter.

5 Key Points Review

- NC/PP updates are block state propagation mechanisms; `GameEvent` is event notification for listeners.
- `GameEvent` records "what happened," it doesn't directly require blocks to change state.
- `WorldChunk.gameEventDispatchers` is `Int2ObjectMap<GameEventDispatcher>`, storing dispatchers by section Y coordinate.
- `getGameEventDispatcher(int ySectionCoord)` lazily creates `SimpleGameEventDispatcher` on the server.
- If a block entity implements `GameEventListener.Holder<?>`, it can expose a listener via `BlockEntityProvider.getGameEventListener()`.
- Sculk sensors and sculk shriekers' `Vibrations.VibrationListener` register to their section's dispatcher.
- After listener removal, empty `SimpleGameEventDispatcher` callbacks `removeGameEventDispatcher()`, removing itself from the `WorldChunk` mapping.
- The chunk unload cleanup process exits block entities and listeners from running structures; section dispatchers are also cleaned with listener lifecycle.

CHAPTER 6

6

Masa Mod

IN THIS CHAPTER

6.1 Masa Gadget

6.2 Litematica Tutorial

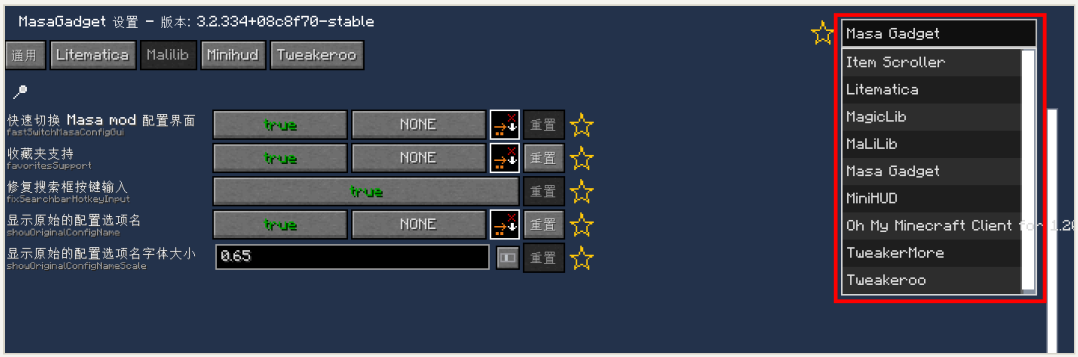
6.1 Masa Gadget

Masa Gadget adds extra features to various masamods, and also provides standalone functionality such as villager information rendering and spawn location searching.

1 General Features

1.1 Quick Switch Between Masamod Interfaces

Enabled by default after installing Masa Gadget. You can quickly switch between masamod configuration interfaces using the button in the top-right corner.
No more forgetting which hotkey opens which mod's settings.



1.2 Favorites Feature

Disabled by default. When enabled, you can add masamod entries to favorites for quick access later.
Example:



2 Litematica

2.1 Better Easy Place Mode & Easy Place Protocol Fix

Enabled by default. After enabling Easy Place, you can still open containers (may not always work). Fixes some bugs with incorrect block orientation when placing schematics (some errors may still occur).

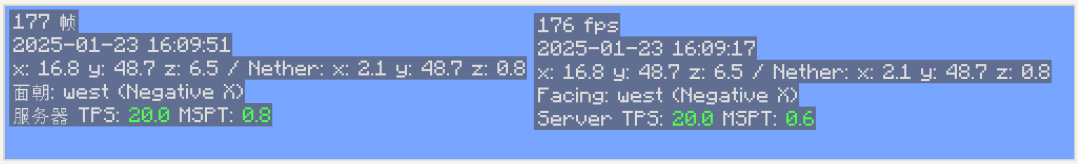
2.2 Save Container Data When Saving Schematics on Server

Enabled by default. Requires support from plusls-carpet-addition.

3 Minihud

3.1 Minihud Translation

Text displayed by Minihud can be translated. Example:



3.2 PCA Sync Protocol for Beehive Data

Pressing the §6Container Preview§r hotkey uses the PCA sync protocol to synchronize beehive and bee nest data.

6.2 Litematica Tutorial

1 Brief Introduction to Litematica

By default, the hotkey for the Litematica main menu is M. Press it to open the main menu.

In the configuration menu, you can change hotkeys and toggle various Litematica features.
When holding the Litematica tool in your main hand or offhand, the Litematica menu bar will appear in the lower left corner.



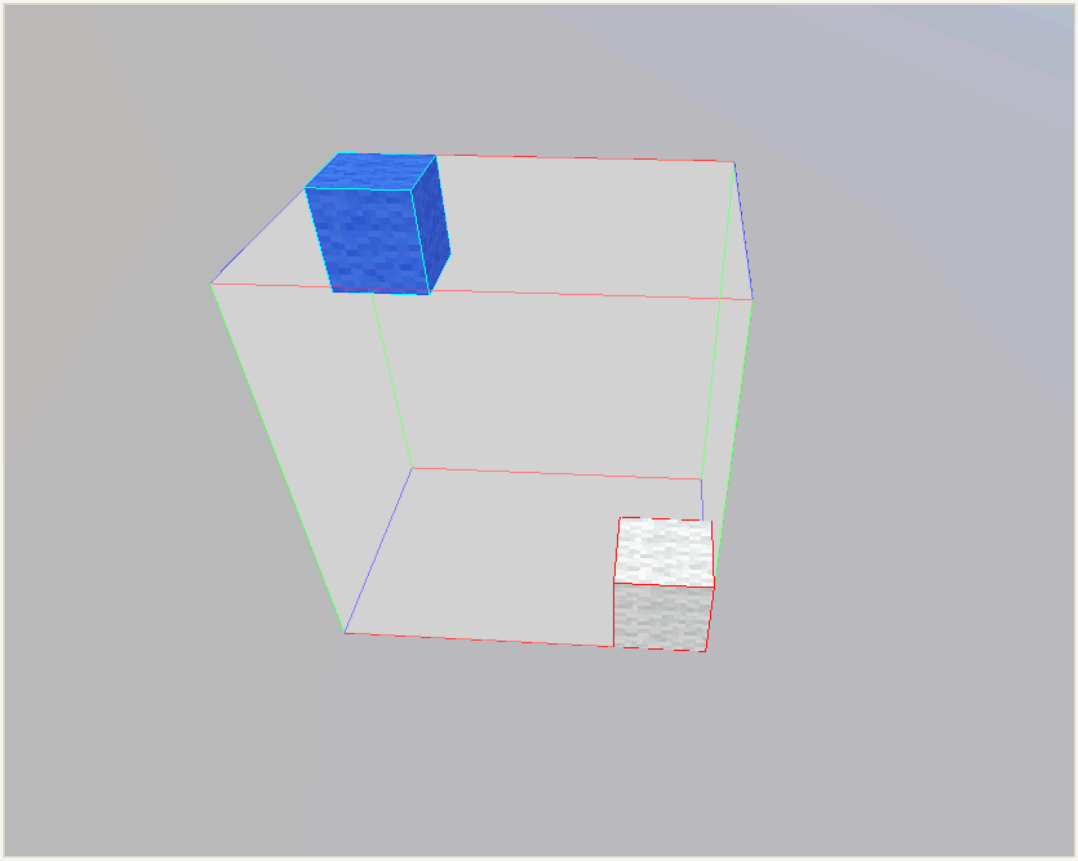
Use ctrl+scroll wheel to switch modes.

1.1 Mode 1 -- Area Selection

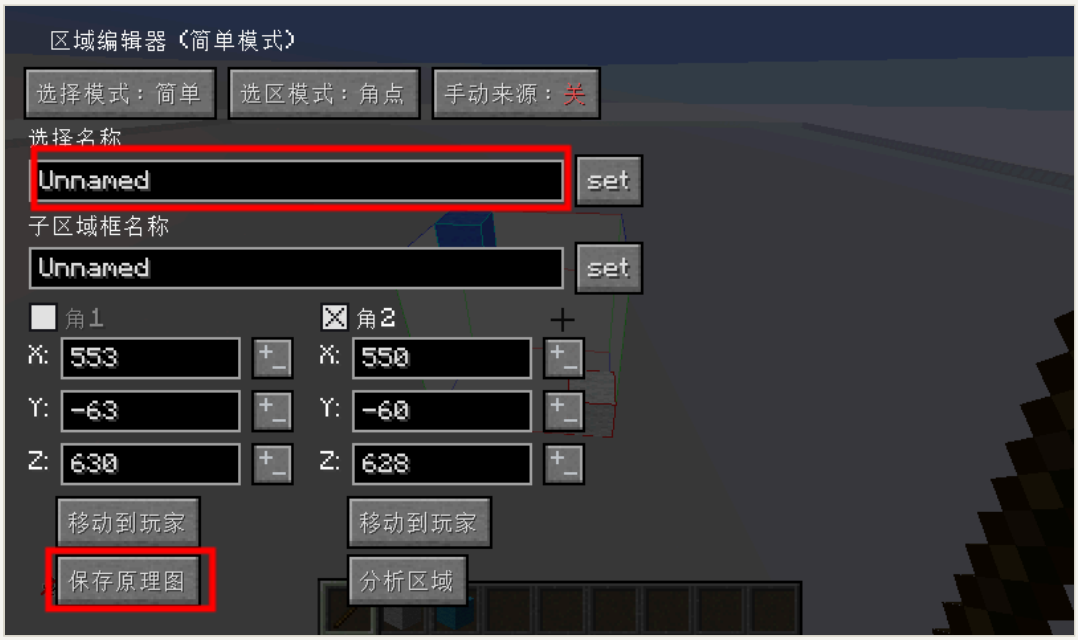
There are two area selection modes, switchable from the main menu. Beginners should use Simple mode, and set the corner mode to "Corners" in the Area Editor.

1.1.1 Simple Mode

In simple mode, there is only one selection area. Left-click and right-click to set the two corner points of your selection.
Use middle-click on a corner (or elsewhere in the selection) to select it. Once selected, hold `alt` and scroll the mouse wheel to move the corner (or the entire selection).



In the selection editor, you can change the name of the selection or save the schematic.

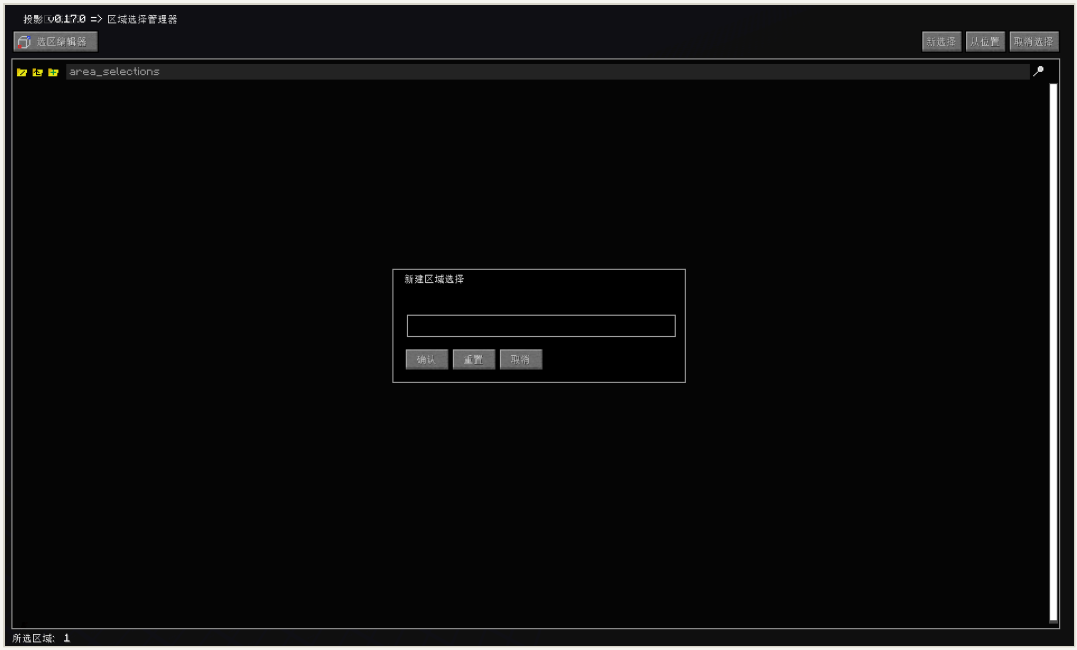


1.1.2 Normal Mode

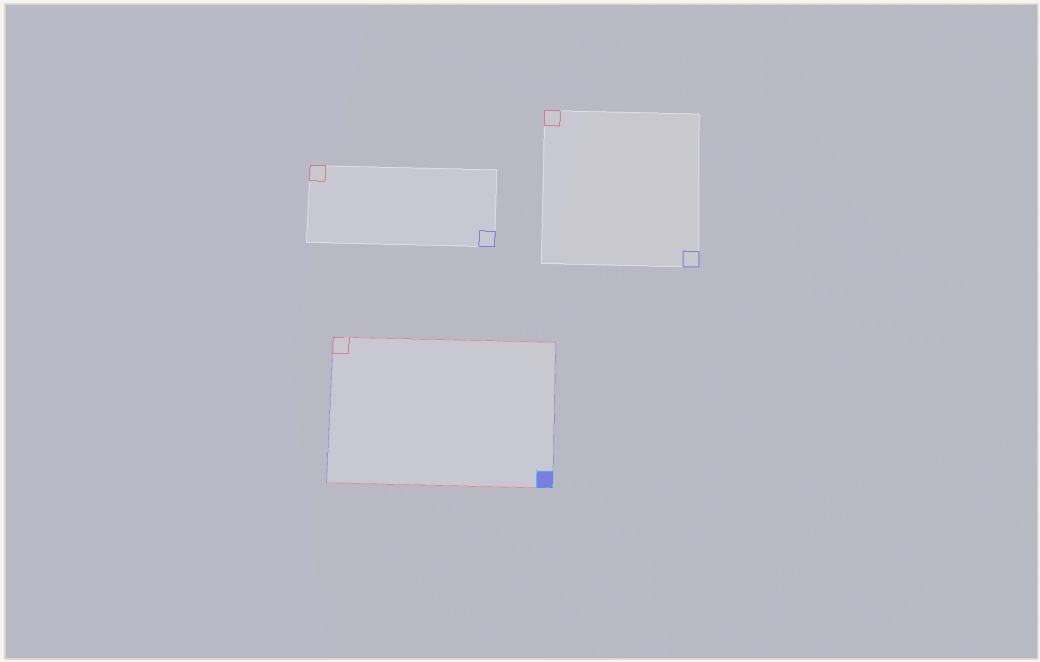
For large machines, saving in simple mode can cause inconveniences during construction. Normal mode solves this.

In normal mode, selections are not created automatically. You need to create them manually:

1. Open `Litematica Main Menu - Selection Manager - New Selection`



2. To create a sub-selection, select "New Sub-region" in `Litematica Main Menu - Area Editor`. You can also rebind the `Add Selection` hotkey in Configuration Menu - Hotkeys for faster access.
   Use middle-click to select a sub-selection corner (or the sub-selection itself).
   The final result is shown in the image



The steps to save the schematic are the same as in simple mode.

1.2 Mode 2: Schematic Placement & Mode 5: Schematic Paste

Import external schematics (skip if none).
It is recommended to enable the custom schematic directory in Litematica settings, so all game versions can share the same schematic save location.



Then place the downloaded schematics into this folder.

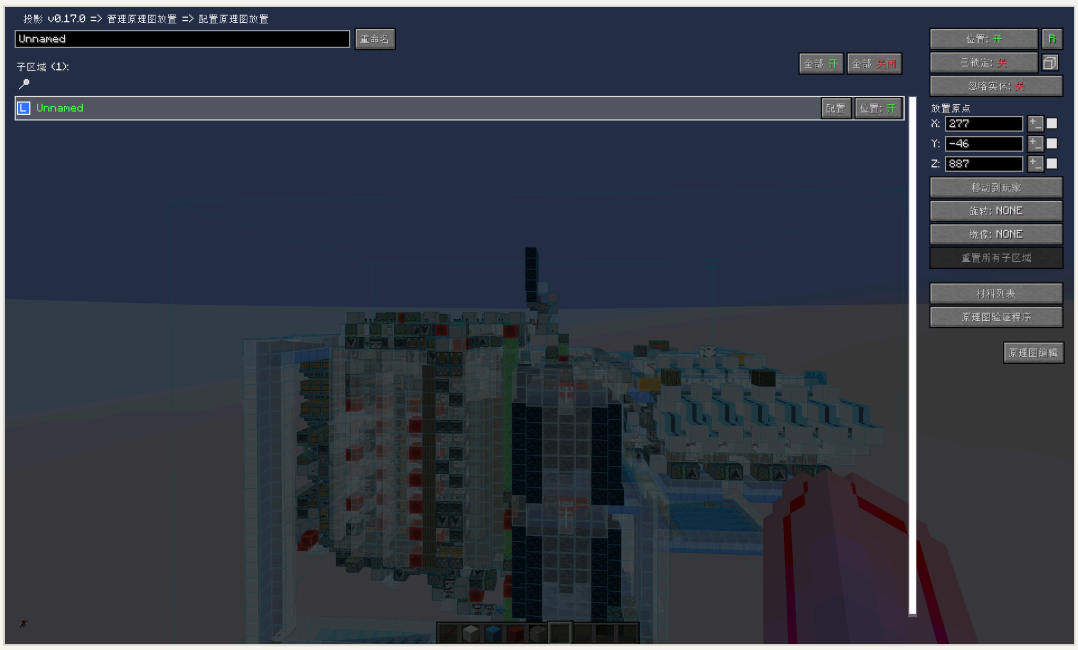
1. Load the schematic in-game

`Litematica Main Menu - Load Schematics` Select the desired schematic and click "Load Schematic" in the lower left corner.

1. Adjust and rotate the schematic

Use middle-click to select the schematic. Hold alt and scroll the mouse wheel to move it.

`Litematica Main Menu - Configure Schematic` Select the desired schematic and click the configure button to enter the configuration interface.



On the right side of the configuration interface, you can adjust the schematic origin position and rotate/mirror the schematic.

Warning: Unless the author states otherwise, do not rotate or mirror machines, as this will likely cause them to malfunction.

1. Paste the schematic (creative only)

CHAPTER 7

7 Tree Farm

IN THIS CHAPTER

7.1 Prerequisites and the Basic Structure of Tree Farms
7.2 Designing the Simplest Tree Farm from Scratch
7.3 Attempting to Design a Multi-Species Tree Farm
7.4 Why Your Tree Farm Is Slow — Introduction to High-Speed Tree Farms
7.5 Why Your Tree Farm Stalls — Introduction to Dustless Wiring
7.6 Everything About 4gt Tree Farms
7.7 Large Spruce Tree Farms

Foreword

At the invitation of BFladderbean, I have decided to lead the writing of foundational teaching materials and accompanying exercises on tree farms to help everyone understand, use, and design tree farms.

Due to the irreplaceable role of tree farms in basic production for Technical Minecraft Survival, and their representativeness in timing and wiring for Mechanical Redstone, we place them at the beginning of the Mechanical Redstone volume, hoping to bring more inspiration to everyone.

In terms of course structure, we separate the 1×1 section from the 2×2 section to provide more targeted explanations of tree farm design and operating principles. This also includes cutting-edge achievements in tree farms such as 4gt all-tree-species, 8gt large spruce, and 12gt dark oak, along with some challenging wiring exercises. We hope students can gain some inspiration from these and contribute to the development of the next generation of tree farms.

Finally, the editor's abilities are limited. If there are errors or shortcomings, please feel free to correct them. Additionally, we will record related video courses as early as possible. Stay tuned.

7.1 Prerequisites and the Basic Structure of Tree Farms

1 Prerequisites

> This section briefly covers the basics you need before diving into tree farm research. For simplicity, some content here is technically "incorrect." For fuller explanations, see the **Update Theory** and **Timing Theory** sections of GTMC.

1.1 Updates

In Minecraft, updates come in two types: **NC updates** and **PP updates**.

The main sources of **NC updates** in tree farms are **TT (Tile Tick) components** (typically Redstone Repeaters, Comparators, and Observers), **Redstone Dust**, **powered or activated rails** **we'll just call these "rails" from here on, since tree farms rarely involve the other two types**, **Pistons** along with the blocks they push/pull, and **Note Blocks**.

When these components change state (e.g., becoming powered or activated, or a block changing), they send **NC updates** to the following:

- Redstone Dust (on energy level change): second-order neighbors
- Repeaters, Comparators, Observers: first the block their output points to, then that block's neighbors (excluding themselves)
- Horizontally placed rails: its own neighbors, then the neighbors of the block below; diagonally placed rails: its own neighbors, then the neighbors of the block below, and finally the neighbors of the block above
- Pistons: neighbors of both the piston itself and the original position of the blocks it pushes/pulls
- Note Blocks: neighbors

These are **NC updates** and can be detected by **BUDs**.

Since tree farm structures are mostly made of Pistons, all **NC updates** in tree farms generally come with **PP updates** as well. Observers specifically detect **PP updates**.

In tree farms, pure PP updates (without accompanying NC updates) typically come from:

- Trapdoors/Fence Gates opening or closing
- Dispensers/Droppers/Hoppers changing activation state
- Glass Panes/Fences/Iron Bars/Walls changing connection state

> In the wireless redstone section, we'll cover three important signal transmission methods: tree power (through leaves, transmitting NC and PP updates), scaffolding power (through scaffolding, transmitting NC and PP updates), and wall power (through walls, vertically transmitting PP updates). Taking advantage of their properties can greatly simplify wiring in tree farms. More details will be in the wireless redstone special topic.

1.2 Timing

In Minecraft, timing comes in two forms: **inter-tick timing** and **intra-tick timing**.

Each second is split into 20 game ticks, or gt (*we won't explain the concept here*). Some components have fixed **macroscopic** delays, such as Repeaters (2-8gt), Comparators and Observers (2gt).
Apart from TT components, other components used in tree farms generally **have no** macroscopic delay.

Each gt is further divided into multiple phases that execute different operations. The phases most relevant to tree farms, in order of execution, are:

- **Tile Tick** (TT) -- TT component operations run here
- **Block Event** (BE) -- Piston and Note Block operations run here
- **Entity Update** (EU) -- entity-related calculations (vehicles, mobs, etc.) run here
- **Tile Entity** (TE) -- Hopper operations and blocks being pushed/pulled by Pistons run here
- **Async Task** (AT/NU) -- player-related calculations run here

Apart from the components listed above, **instant components** execute their operations immediately upon receiving an update, regardless of which phase the update is in. Common instant components include **Redstone Dust and rails**, **Fence Gates and Trapdoors**, **Note Blocks**, **Droppers and Dispensers**.

The execution order of TT components depends on **macroscopic timing**, **TT priority**, and **sub-order**. In practice, TT components execute in this order:
Any Repeater >= Comparator pointing to a Comparator > general Comparator or Observer

BE components (Pistons and Note Blocks) execute in the order they **receive updates and confirm a state change is needed**. This ordering is sometimes called **depth**. Note Blocks do not increase depth (i.e., they don't delay the execution of the components they update within BE).

When analyzing timing, look at **macroscopic timing** first, then **which phase the operation falls in**, and finally **the execution order within that phase**.

In tree farms, each Piston action (push/pull/0t) takes **3gt** by default. Piston actions triggered by **1gt and 2gt** powering take **4gt and 5gt** respectively.

2 Basic Structure of Tree Farms

From Minecraft 1.15 onward, a tree farm's basic structure consists of:

- Bonemealing
- Trunk processing (sometimes split into **main trunk processing** and **root processing**)

- Leaf processing
- Sapling recycling
- Block-to-drop conversion

> Most tree farms also include a **detection** module that detects sapling growth and triggers the farm, but this isn't required. We'll cover it in the next chapter.
>
> In Minecraft 1.14 and below, jungle and acacia trees have special growth detection requirements, so tree farms sometimes need a **height increase** module. Spruce trees require no logs in a 5x5 area to grow, which means a **retractable wall** is needed. This article focuses on 1.15+ tree farms, so we'll skip these cases for now.

2.1 Bonemealing

Typically, **Dispensers** fire Bone Meal onto saplings.

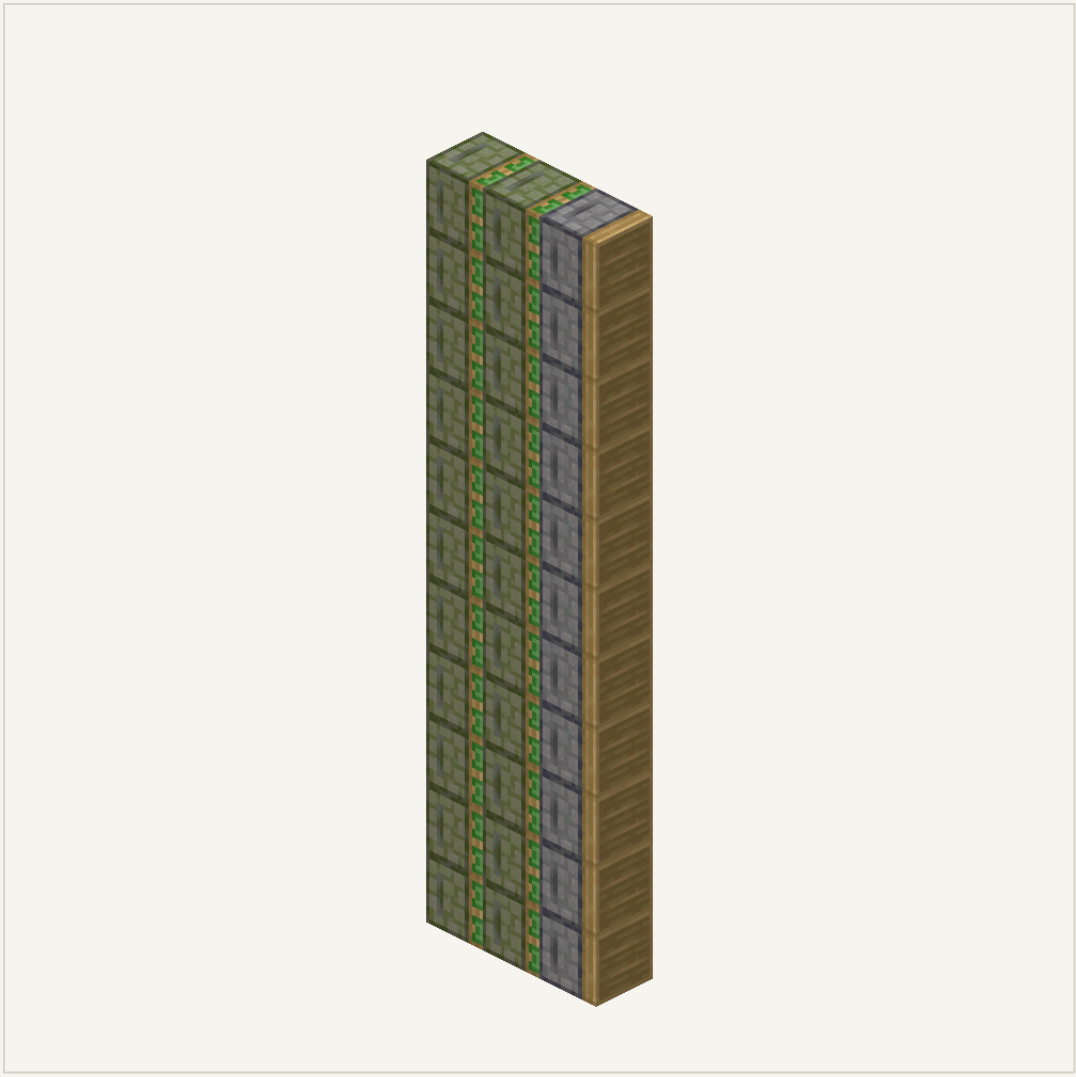
Since Dispensers hold very little Bone Meal, it's usually fed in through **Hoppers or Droppers**, with an **unloader** transferring Bone Meal from Shulker Boxes into the Hopper or Dropper chain.


Basic bonemealing structure

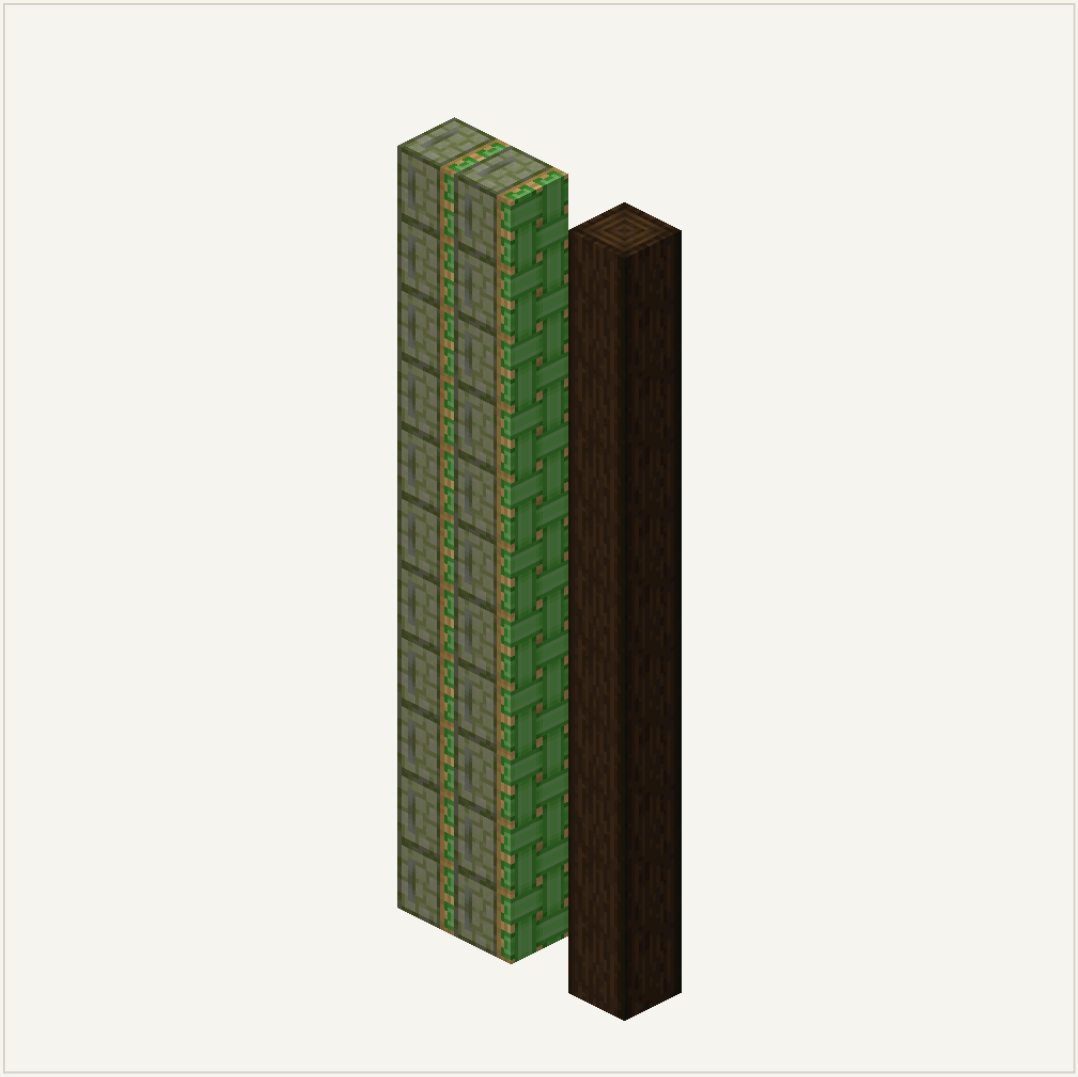
2.2 Trunk Processing

Main trunk processing is straightforward: move the trunk away from where it grew. The basic approaches are **triple push extension** and **pseudo double push extension**. For birch and oak, a simple push/pull setup works fine.

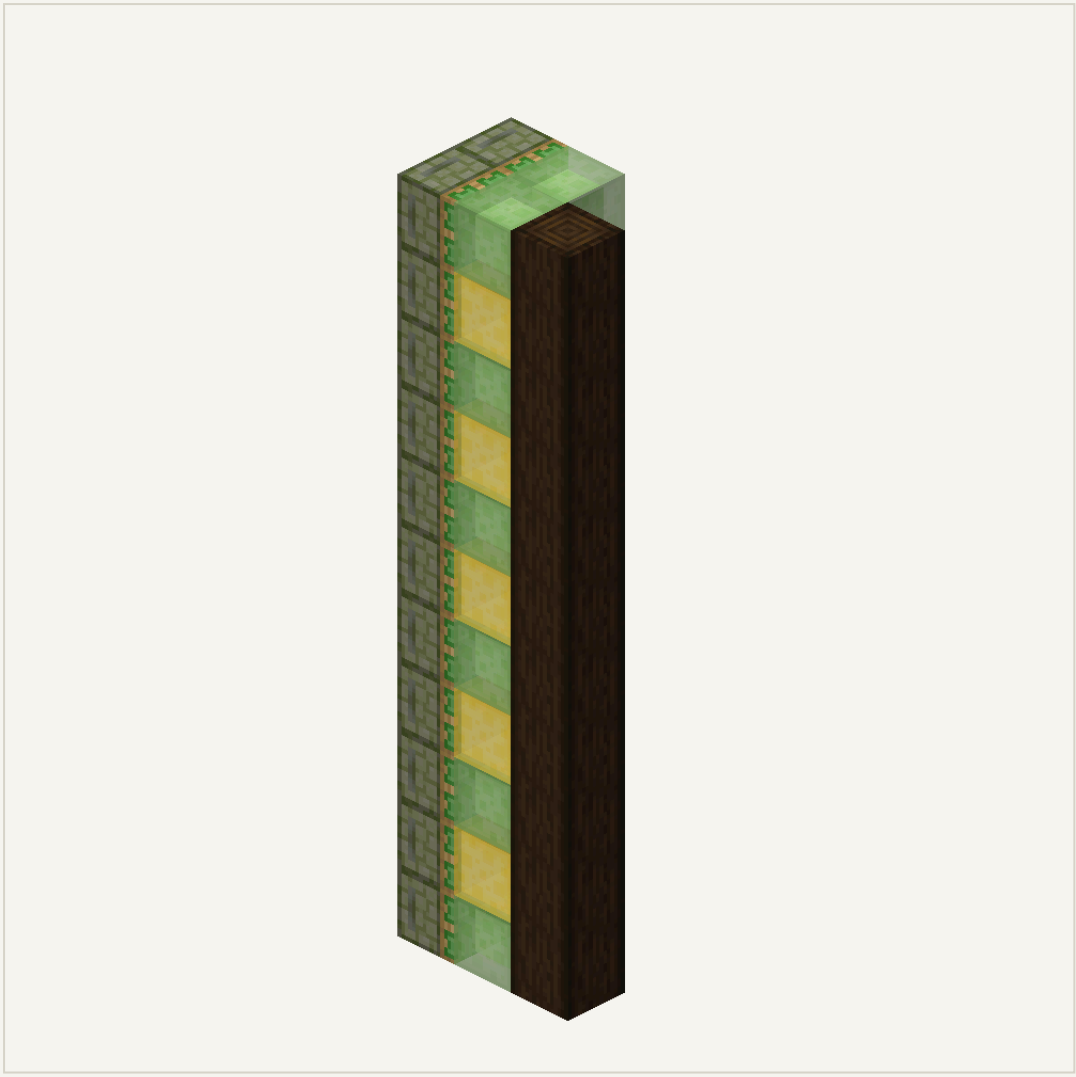
Triple push extension:



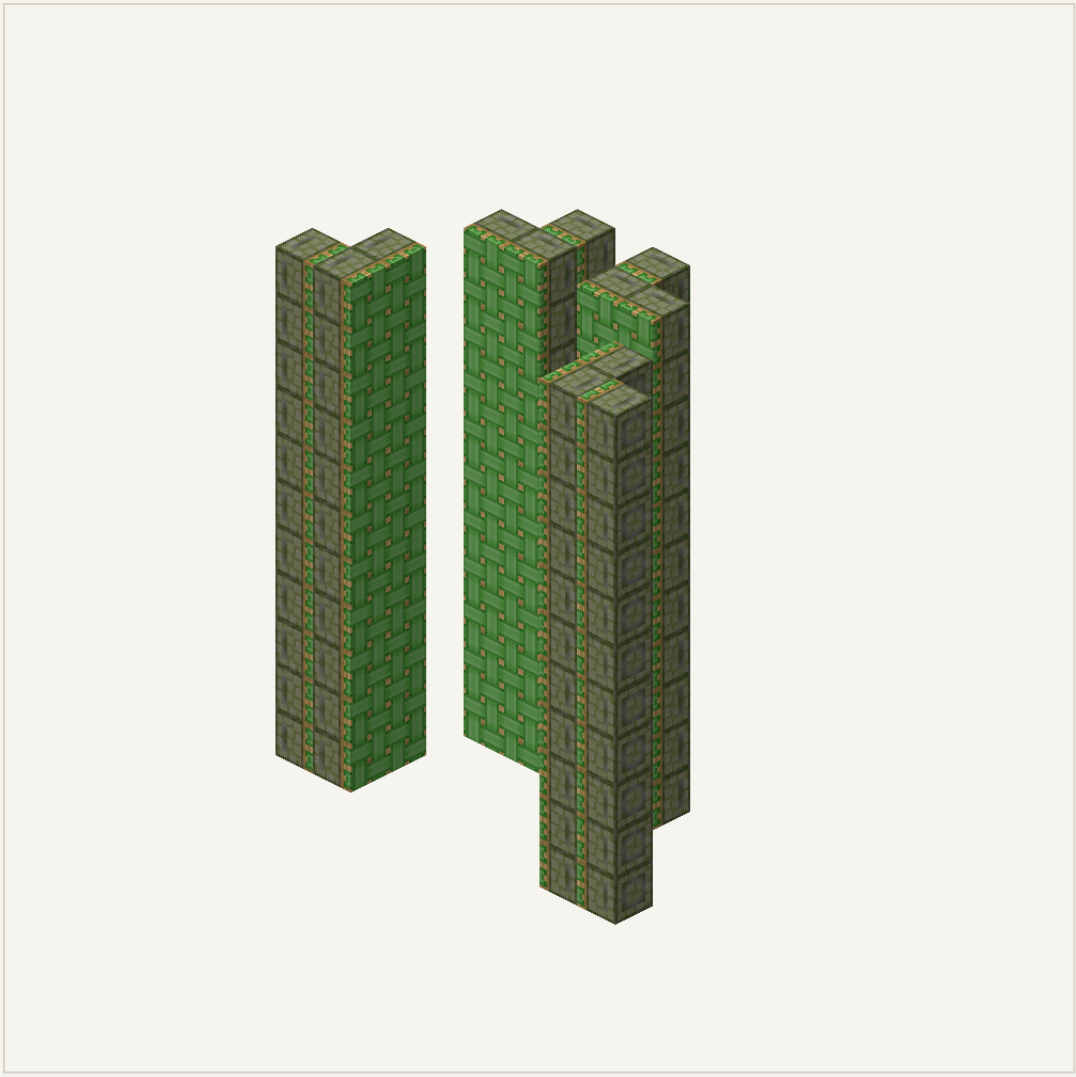
Pseudo double push extension:



Honey-slime wall double push extension:



Retraction-type pseudo double push extension (a design where the secondary Piston only performs retraction (0t), shaving one Piston action off the timing):



> The most widely used design today is honey-slime wall double push extension.

Root processing generally has four approaches: processing alongside the main trunk, **upward push**, **downward pull**, and **side pull**.

Upward push:





Downward pull:

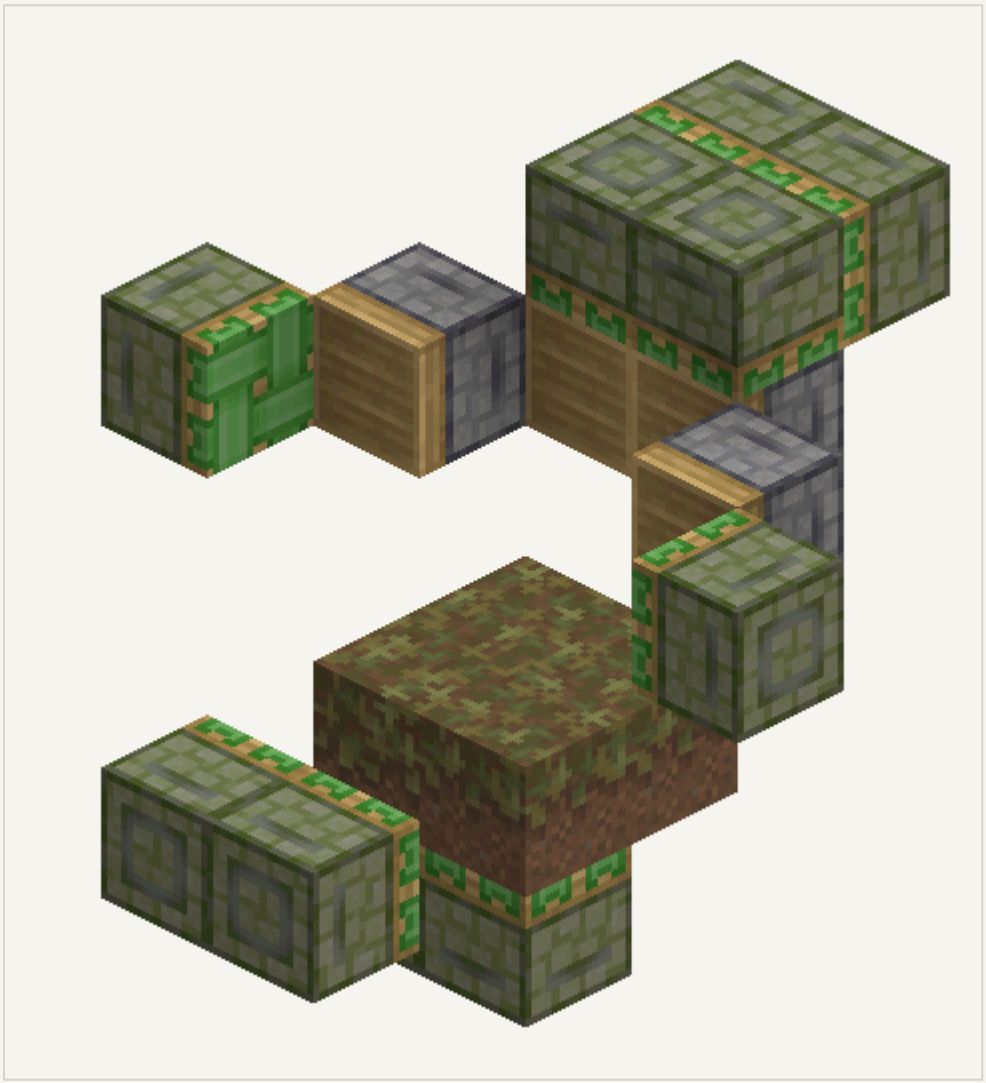


Side pull:





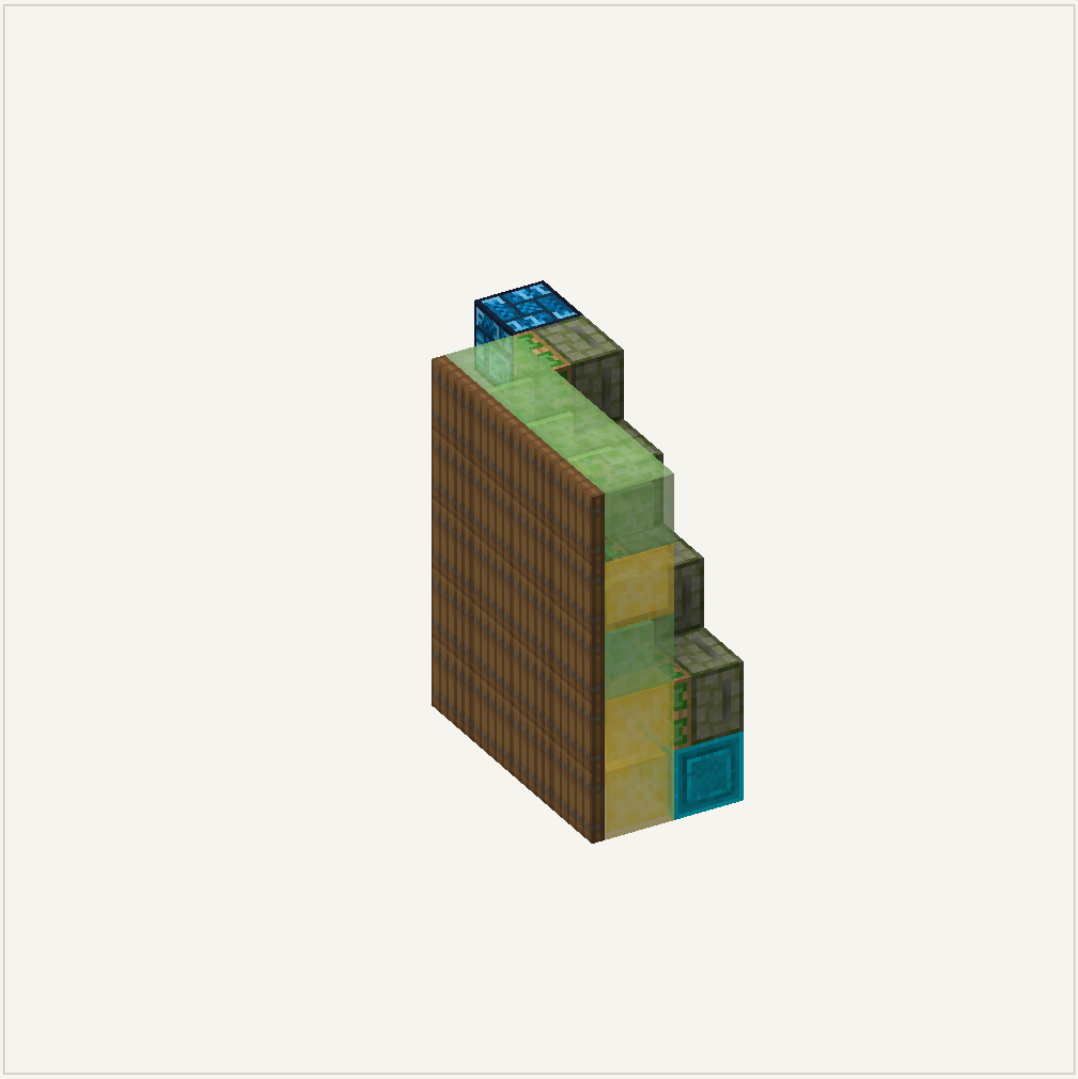
For dark oak, an **upward pull** method is sometimes used.



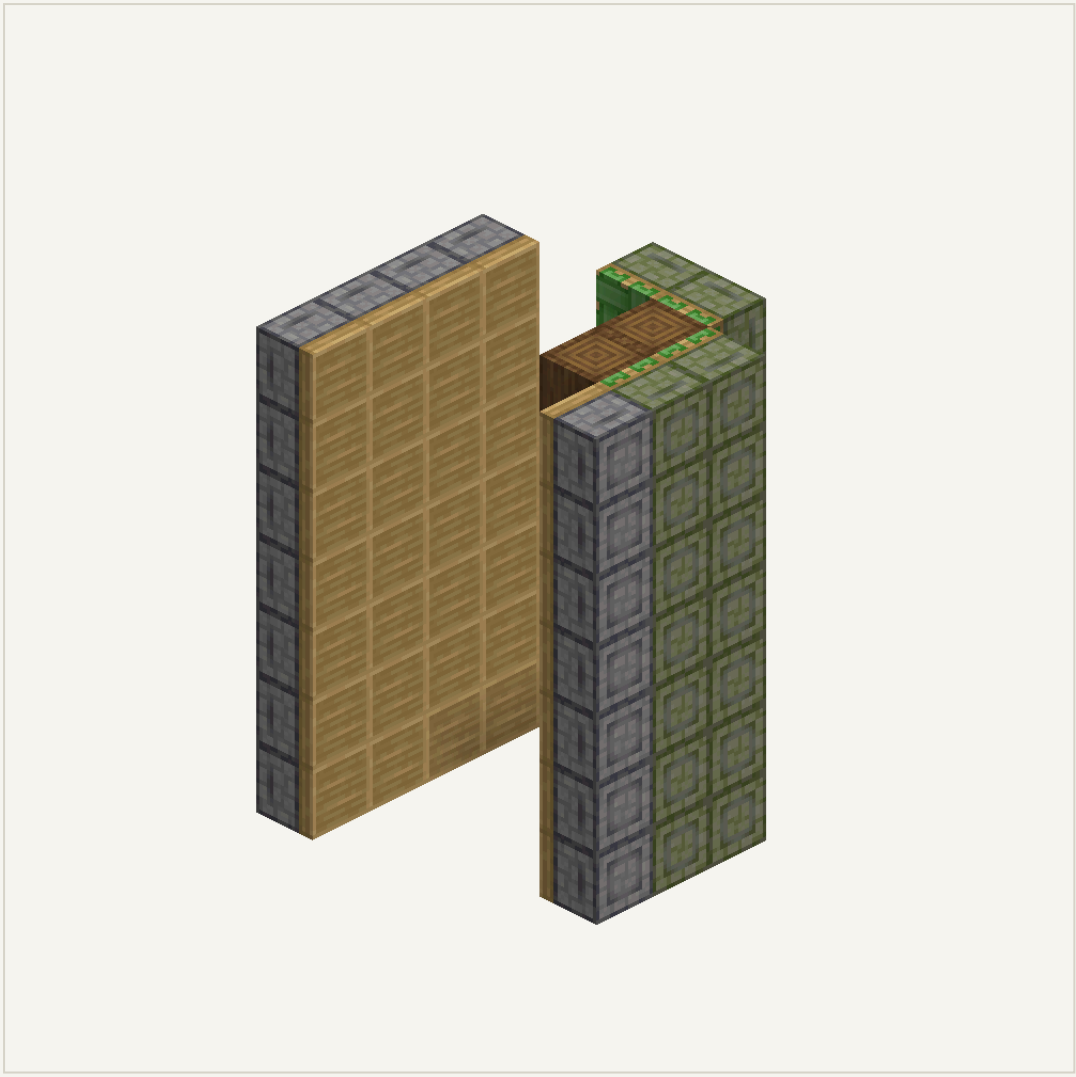
2.3 Leaf Processing

Also straightforward: use Pistons or honey-slime walls to push away **enough leaves** to get the saplings you need.

Honey-slime wall:



Piston:



Each leaf block has a fixed chance to drop a sapling: jungle 1/40, all other tree types 1/20.

2.4 Sapling Recycling

Saplings dropped by the leaf processing module are collected near the player using **Hoppers**, **water flow**, or similar methods.

Hopper:



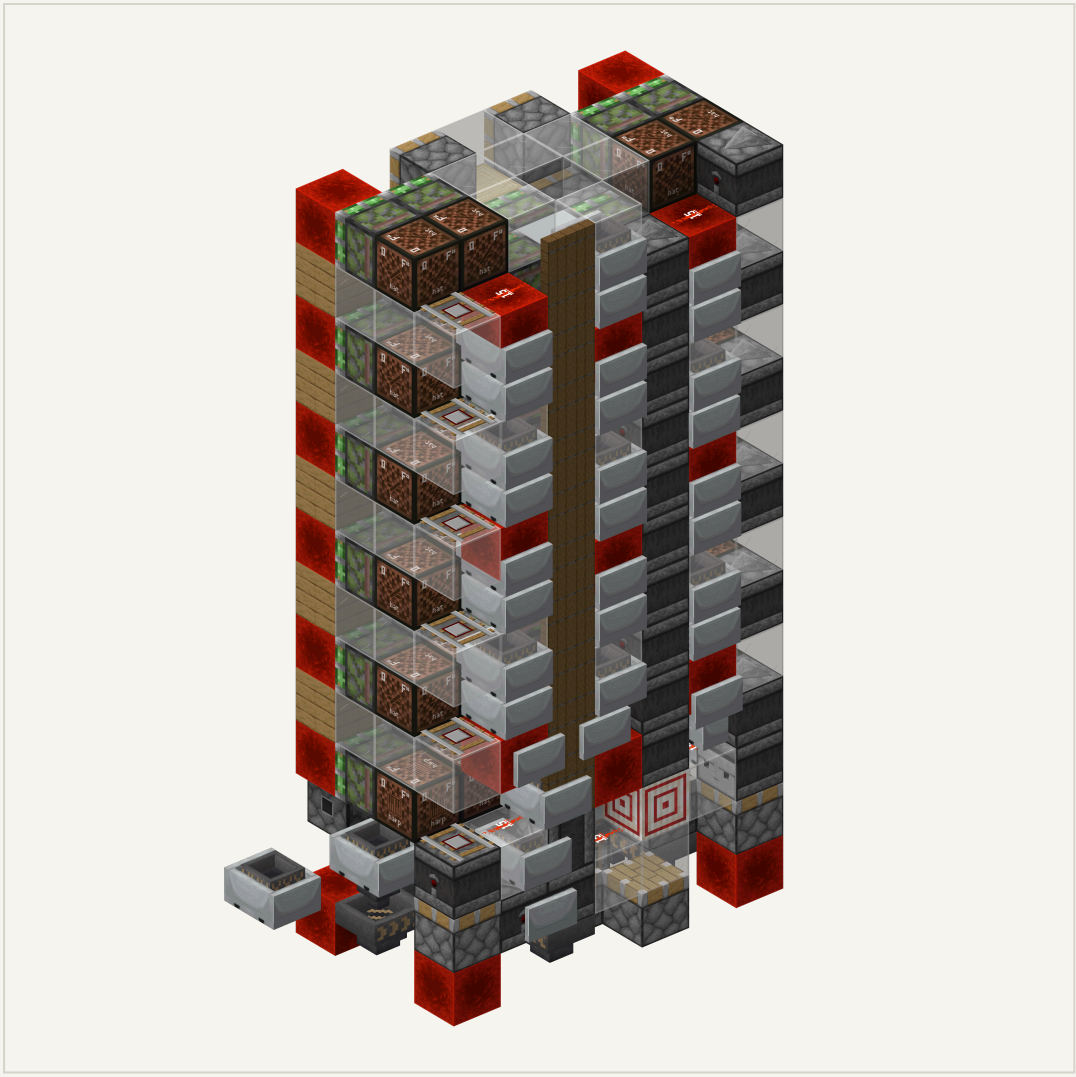
@Scorpio 天蝎君 4gt birch

Water flow:



@Feng_Bl, Sunflower_Lin 6gt birch

In tree farms with fast processing cycles and tight internal space, **Hopper Minecarts** are sometimes used for sapling recycling.

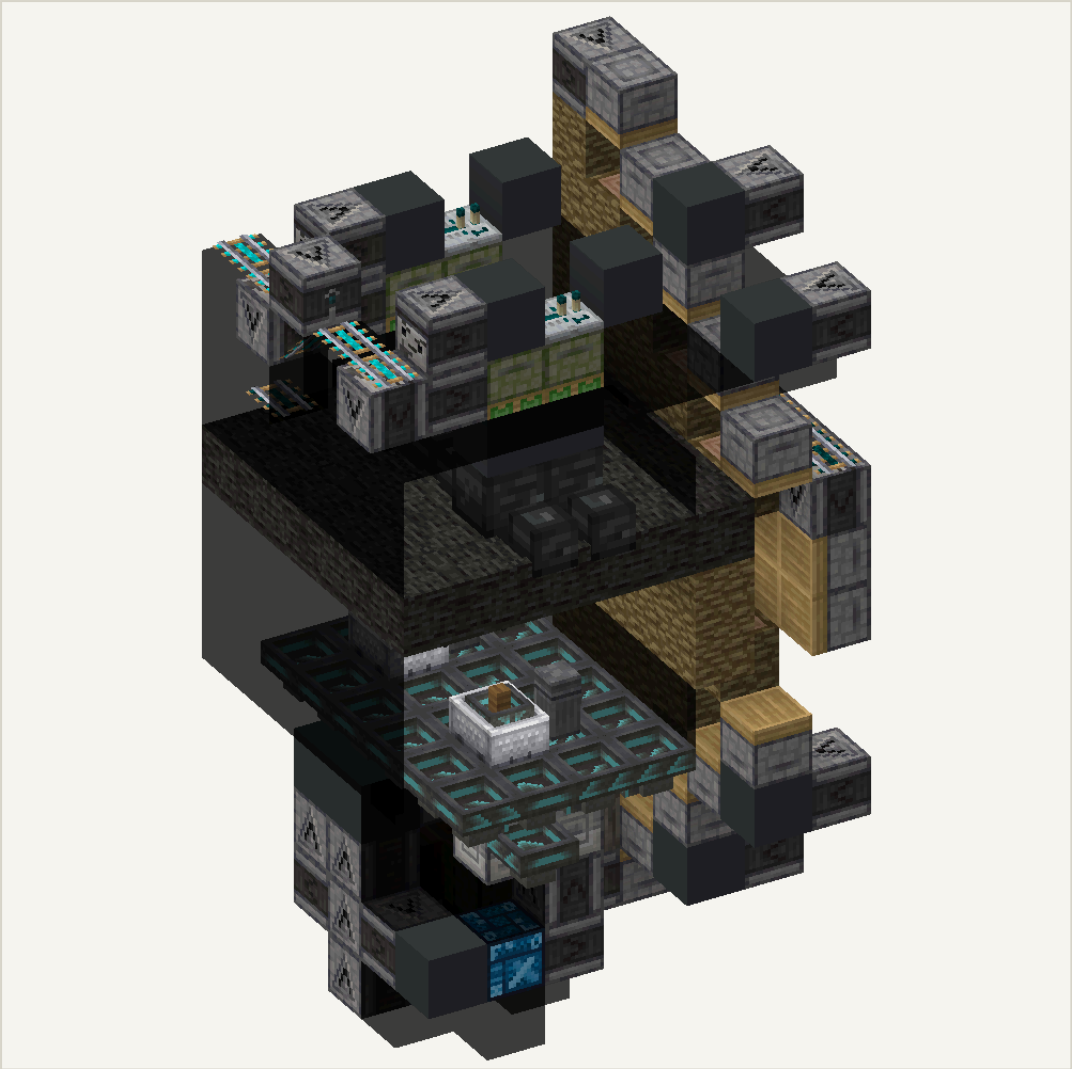


@Qonctrol, PUTF sharp

2.5 Block-to-Drop Conversion

There are two main methods: **Wither** and **TNT**.

Wither:



@Scorpio 天蝎君, PUTF flat wither

TNT:



@Feng_Bl

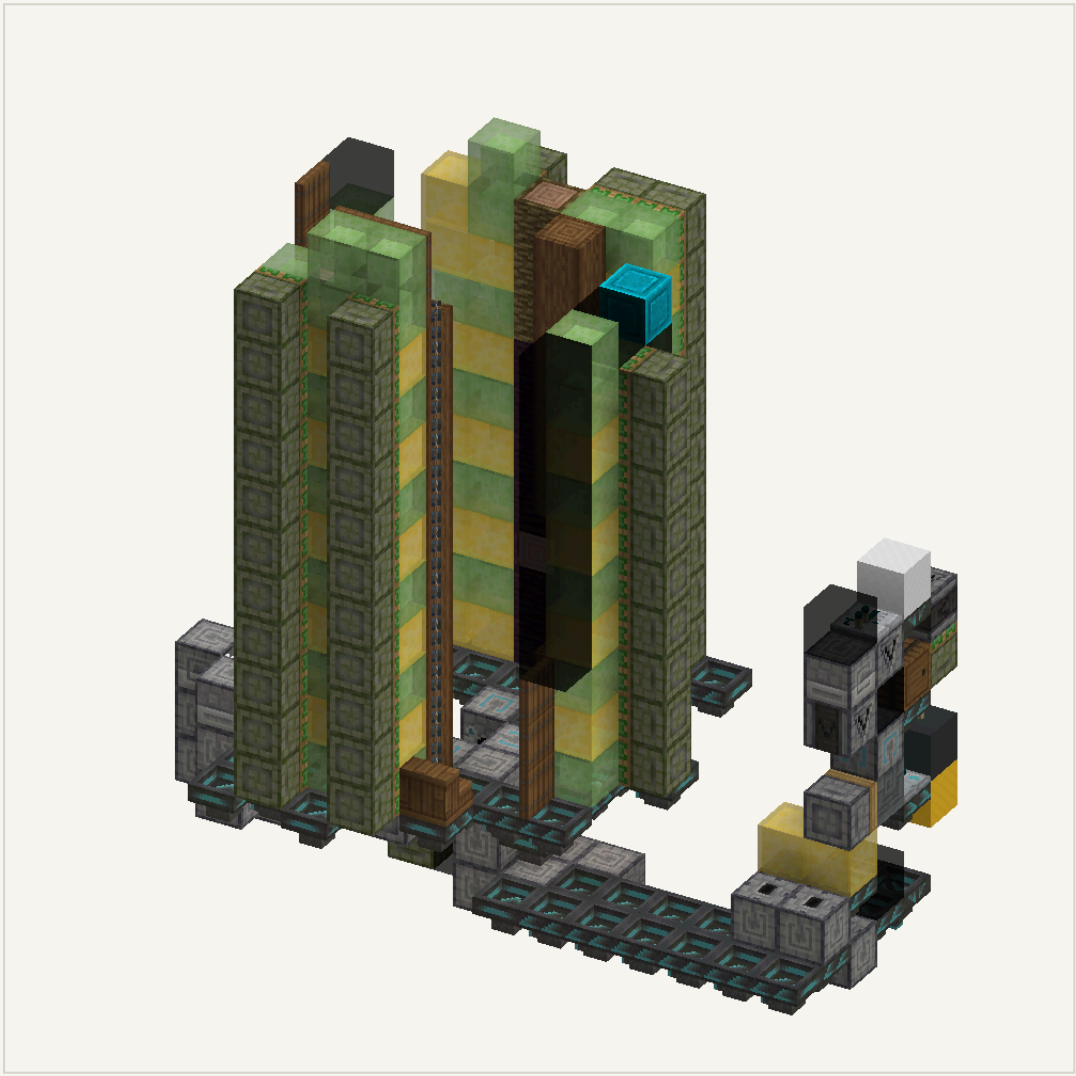
7.2 Designing the Simplest Tree Farm from Scratch

The design process for tree farms generally consists of **architecture design** and **wiring**. Here we'll use a very simple Birch tree farm as an example to guide you through designing a tree farm from scratch.

1 Architecture Design

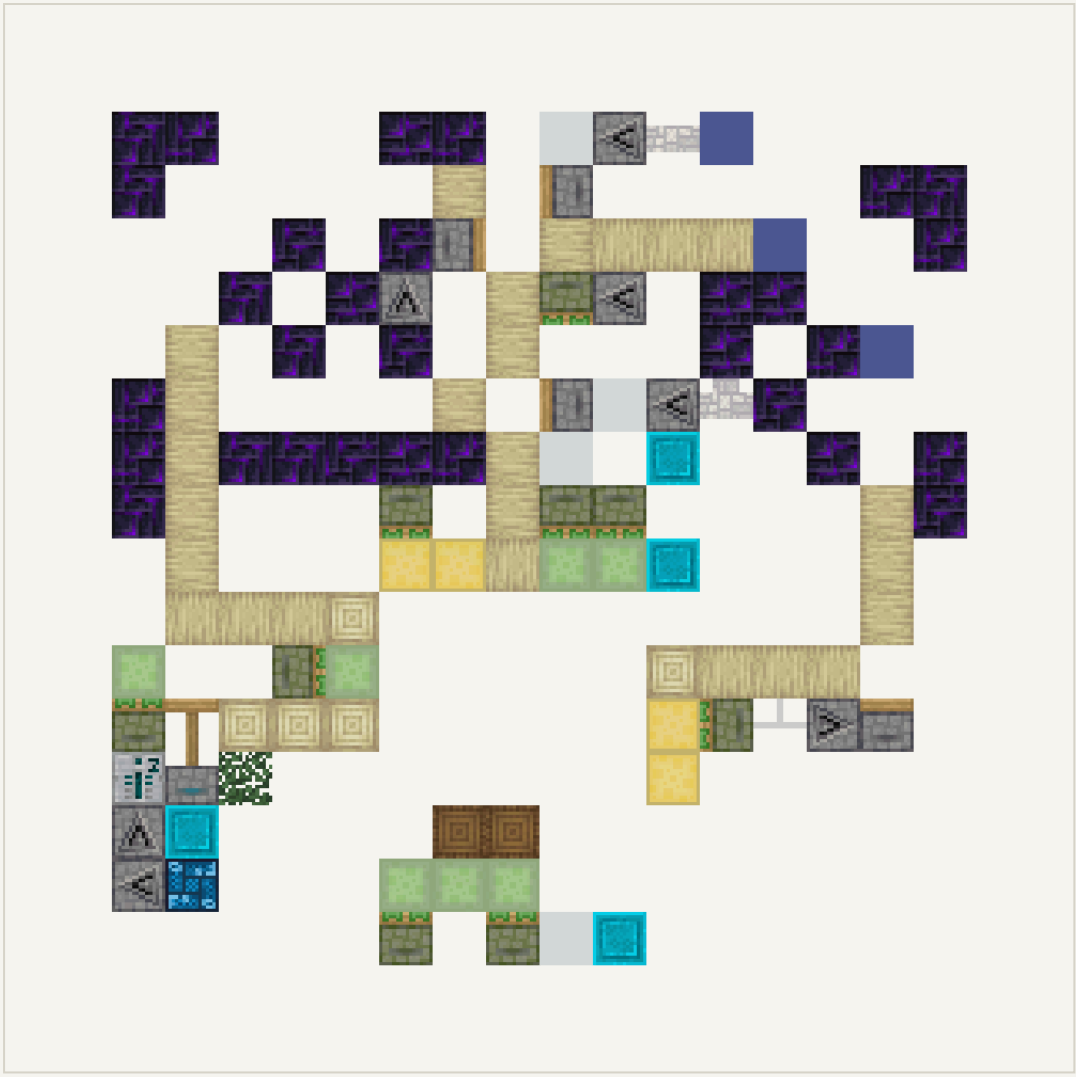
Architecture in tree farms generally refers to the **specific arrangement** of the tree farm structure. It can be divided into **core architecture** and **processing architecture** — *the latter covering block flow and drop collection modules.*

Core architecture:



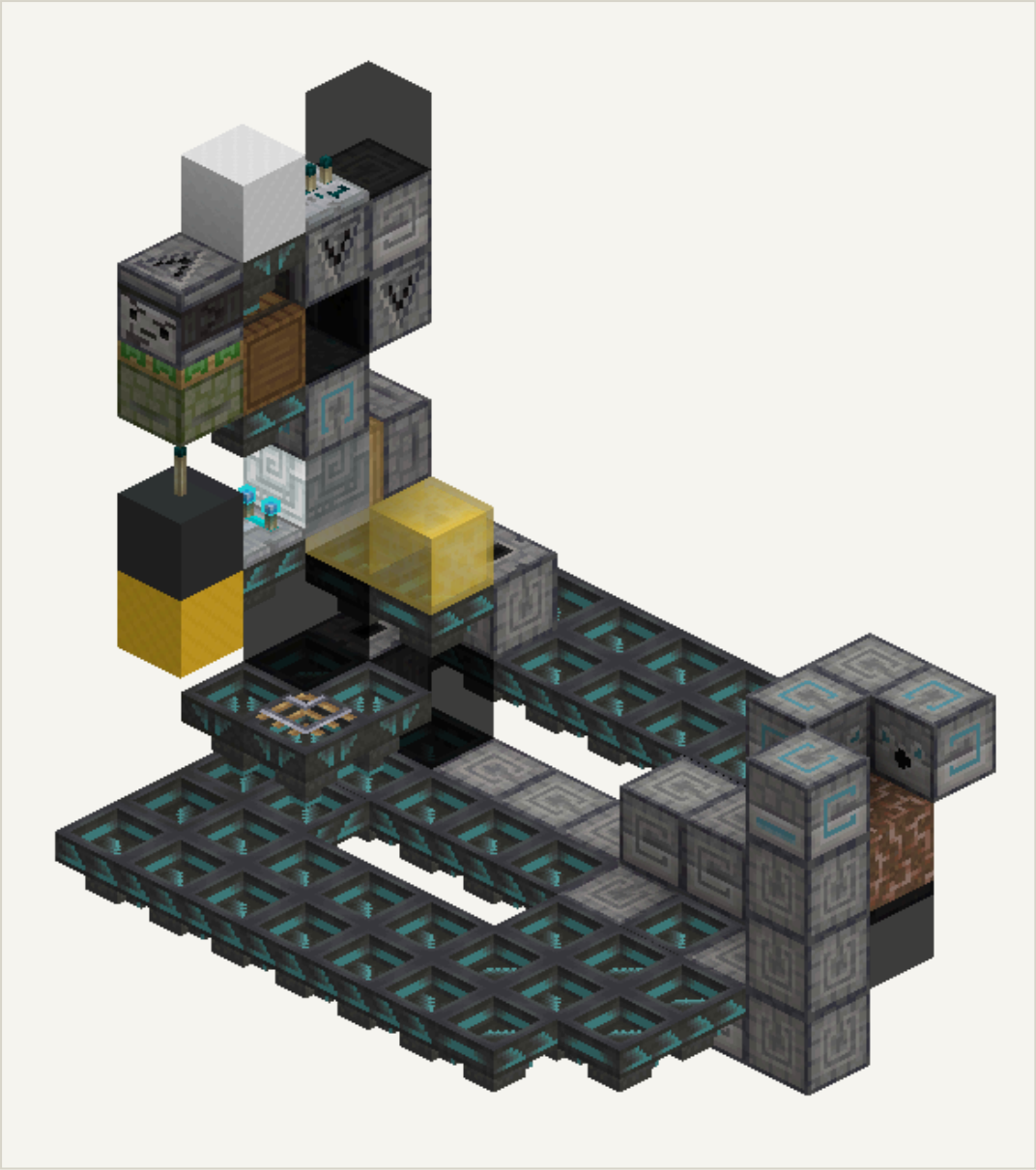
@Chuo_di

Processing architecture:



@Chuo_di

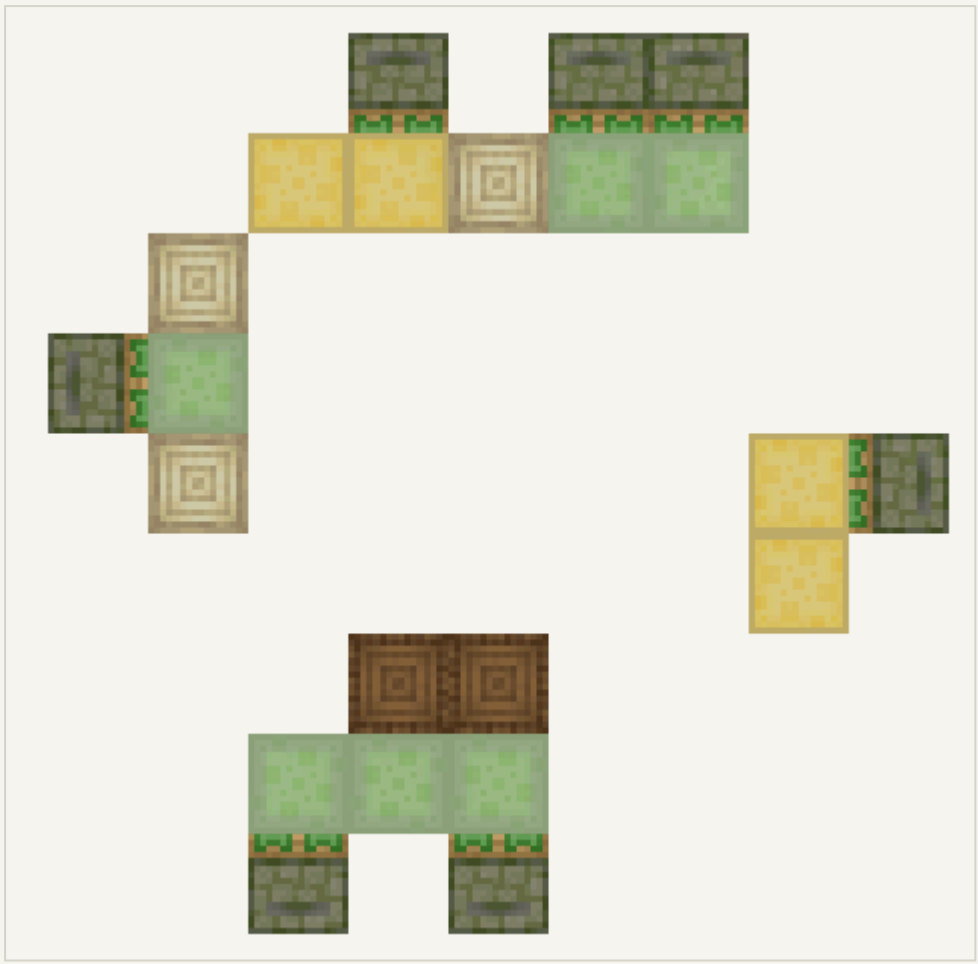
Core architecture generally includes **bone meal architecture**,



trunk processing architecture,



leaf processing architecture,



and **sapling recycling architecture**,



Generally, the **bone meal structure** or **trunk processing architecture** is designed first, followed by the **leaf processing architecture** and **sapling recycling architecture**.

The basic form of trunk processing architecture was covered in the previous chapter. Here we choose **direct push** for trunk processing (*trunk processing that uses a push-out structure is called "main push"*), with the root processed together with the trunk. For bone meal architecture, we use a **single dispenser**.



Generally, tree farms with processing cycles longer than 4gt shouldn't use clock activation. Therefore, we need to introduce a **detection** structure to detect tree growth and trigger the tree farm's mechanical structure.

Basic detection methods include: **Comparator detection**, **QC detection**, **BUD detection**, **push limit detection**

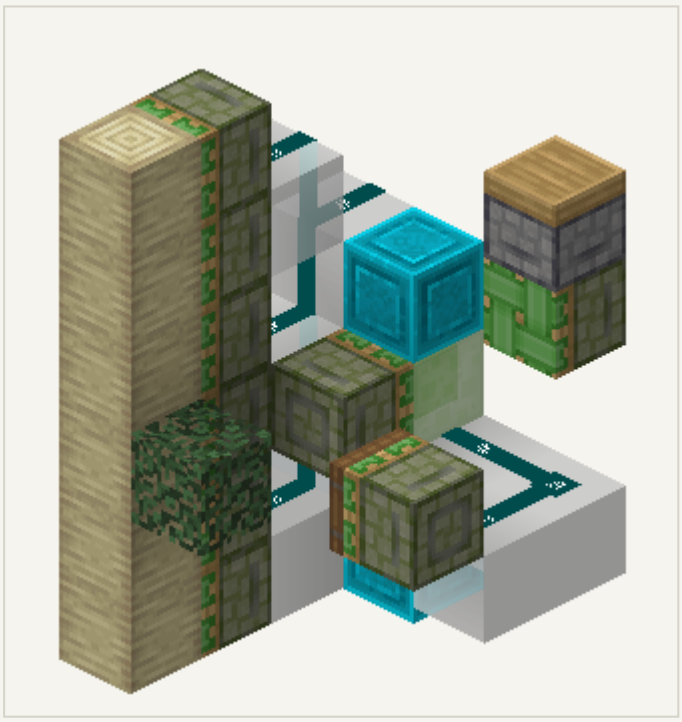
Comparator detection:



QC detection:



BUD detection (here BF got lazy and just used leaf detection):



Push limit detection:



Here we use QC detection. The basic principle of QC detection is that since trunks can be powered, once the sapling grows, the tree root gets powered, and the piston diagonally below receives a QC update. The piston just needs one more update to push out, thereby triggering the tree farm's processing structure.

> Although we won't cover it in detail yet, push limit detection is currently the most important detection method. After this chapter, almost all detection-based tree farm designs we analyze use push limit detection, which is worth noting.

For leaf processing architecture, Birch only needs 20 leaves, so a simple piston wall can meet the requirements. Since pistons cause lag, we instead use a **honey-slime wall** with trapdoors and chains attached — *this increases the space for sapling splashing, keeping the leaf processing volume low and largely avoiding the need for hopper minecarts.*

> The lag from unoptimized mechanical-electrical design is quite significant, with one of the biggest culprits being pistons. Readers should always consider whether pistons in the tree farm can be reduced (another major culprit is Redstone Dust, whose reduction we'll discuss in another chapter).

> For more complex tree farms, operation timing must also be designed alongside the core architecture. Here we'll temporarily ignore it and leave it for detailed discussion when we get to branch trees.

For sapling recycling architecture, we cover the bottom of the tree farm with hoppers and connect them to a dropper. When AFK, the player can stand next to this dropper to collect the saplings recycled by the tree farm. We also enclose the area around the core to keep saplings from splashing too far.

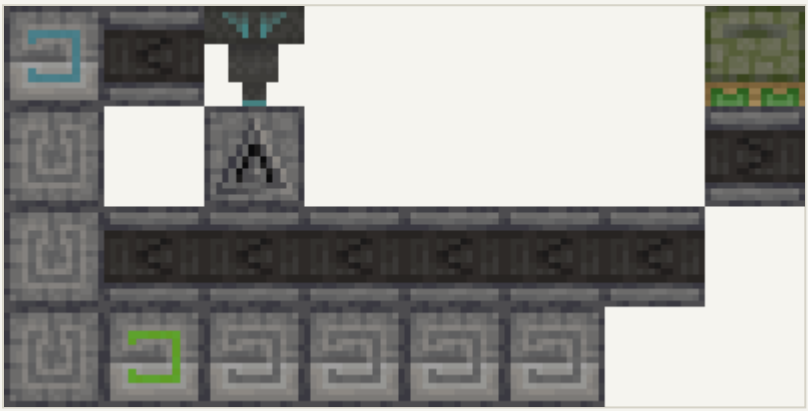
For processing architecture, we use a simple **TNT explosion chamber**.

> It's worth noting that since current high-speed all-tree-type farms have multiple log block flow outputs, the processing architecture will also be very complex. We'll discuss general design approaches for these situations in the analysis of PUTF std2.

2 Wiring

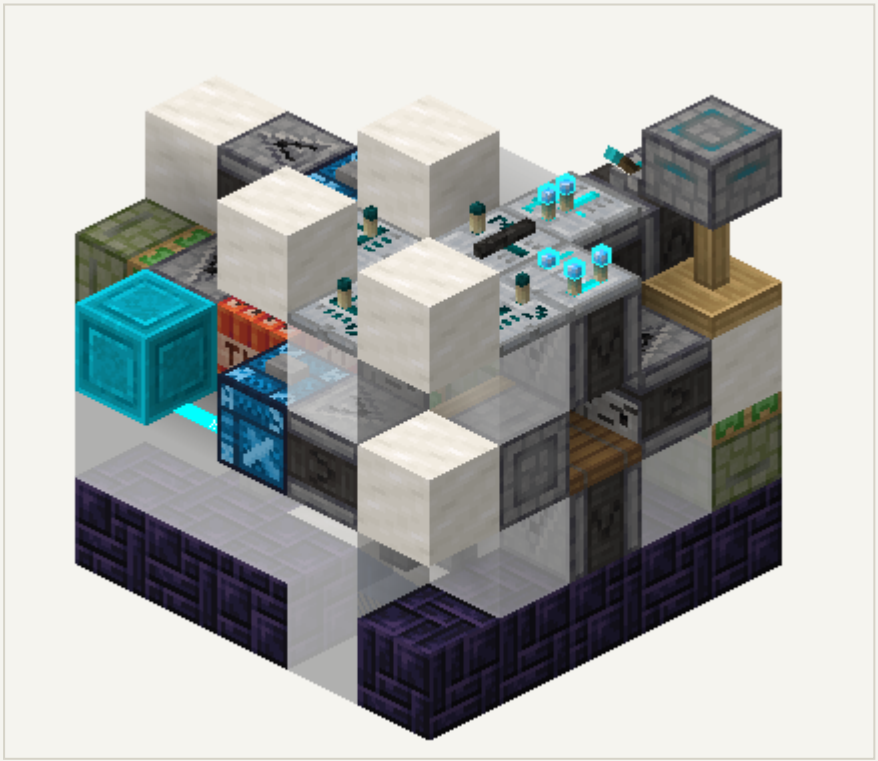
The basic idea is to connect the various parts of the tree farm without affecting its timing.

First, we connect a **4gt clock** to the bone meal dispenser and sapling dropper.



Then use Redstone Dust to activate the pistons in each section.

For the TNT explosion chamber, we simply place a **TNT duplicator**. To save space, we use water flow to **buffer TNT** and collect drops.



At this point, the most basic tree farm design is complete. If you didn't consult the illustrations, this is arguably the first tree farm you've designed. Now try analyzing its timing!

> Timing analysis is the most basic method of analyzing tree farms. Being able to read a tree farm's operation timing from its architecture and wiring is an essential skill for every tree farm enthusiast.

Based on the design process described above, you can design a Birch tree farm with a cycle as fast as 6gt. Here is a reference design~~Lazy Xinghe never made an unloader, so don't follow their example, kids~~

7.3 Attempting to Design a Multi-Species Tree Farm

In the previous section, we completed the simplest tree farm. However, this tree farm has a serious problem: it can only process birch trees and cannot handle other tree species. To solve this problem, in this section we attempt to design a multi-species tree farm.

1 Core Architecture Design

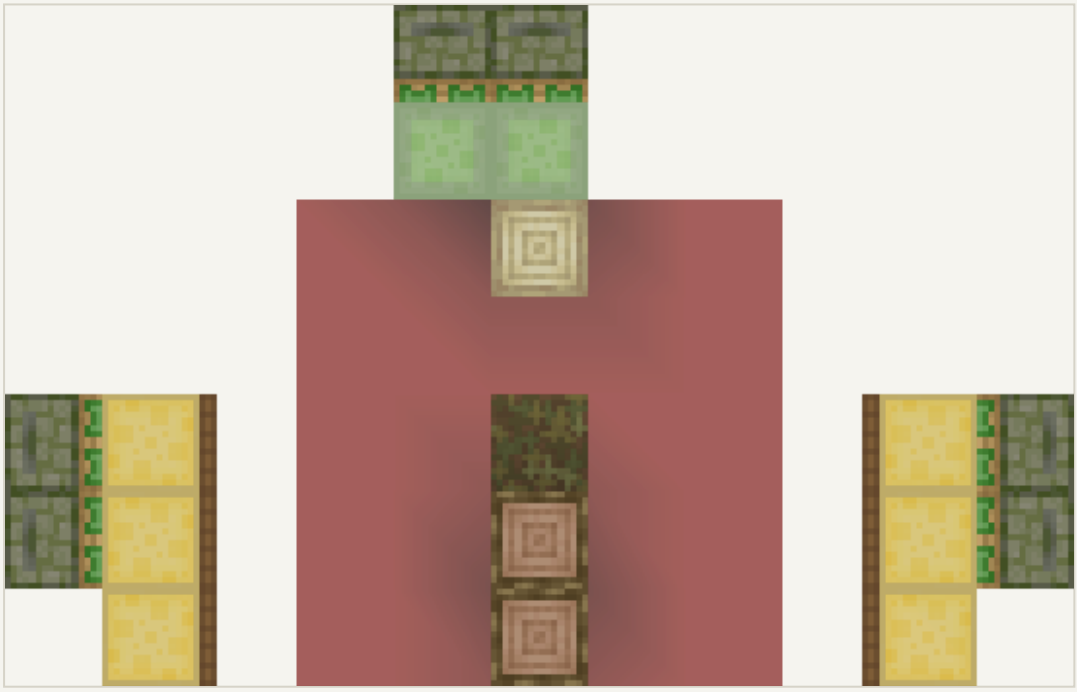
For a multi-species tree farm, the architecture needs to accommodate the growth requirements of **every tree species it handles**. In practice, this means **taking the union**—in each aspect of the design, we must meet the requirements of **the most demanding species**. For now, we'll focus on the five classic tree species: **Birch, Oak, Jungle, Spruce, and Acacia**. We'll walk through them one by one in the typical design order.

Birch has the fewest growth restrictions, so we'll start with spruce.

1.1 Spruce Architecture Requirements

Spruce requires a 5x5 area centered on the sapling with no blocks obstructing growth. For trunk processing, we can use **triple recursion**, **pseudo-double recursion**, or **honey-slime double recursion**.

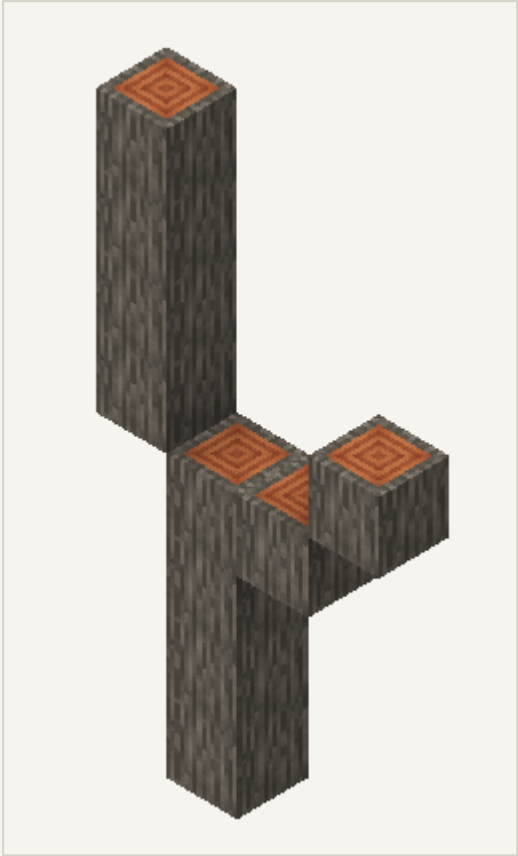
Example architecture:

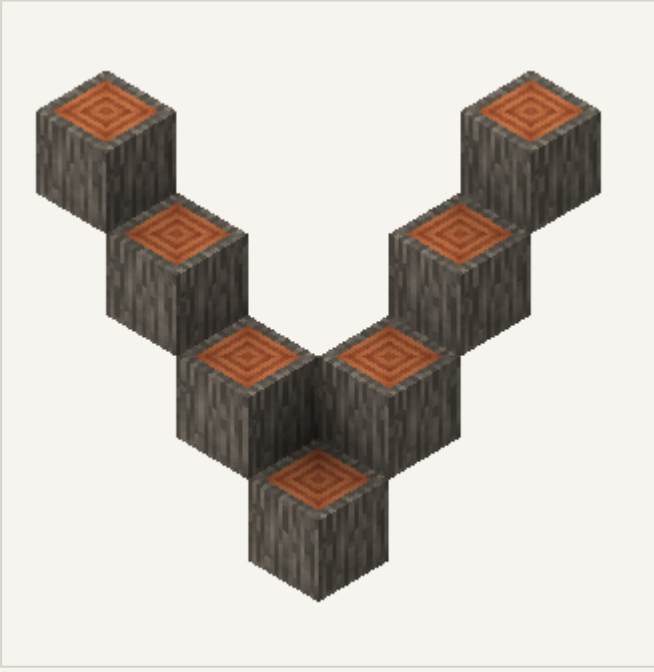


> The main pusher design will significantly impact your oak height-limiting options, so choose carefully

1.2 Acacia Architecture Requirements

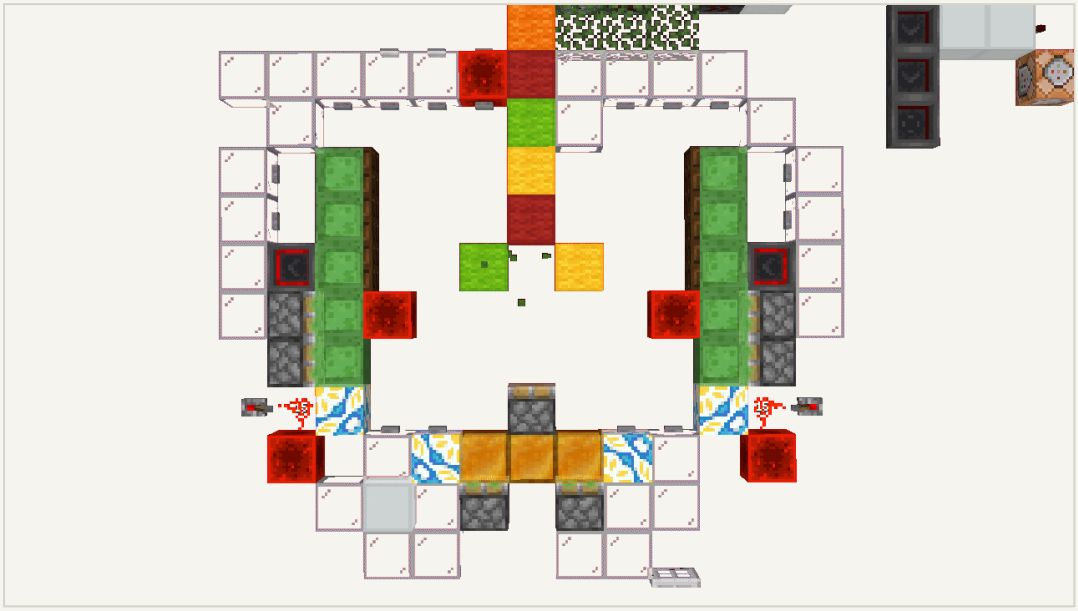
Acacia grows **up to 2** logs on the same y-level, extending along the **x or z axis** and **at most 4 blocks** away from the sapling. These logs can form straight lines, corners, and other configurations, as shown below.





This means the main pusher alone cannot clear all the logs, so we need **side branch processing**. There are two design approaches: **center consolidation** and **additional log outputs**.

For center consolidation, we first push the logs on the main pusher side to the center, then push the left-side logs to the center and eject them with the main pusher, and finally do the same for the right-side logs.



▶ This figure is animated. View original

A typical example of additional log outputs is nutstory's architecture. The animation demonstrates how it works.



▶ This figure is animated. View original

@Dreaming_Galaxy, XJH_Jorhai, Feng_Bl, Bright_Observer PUTF standard

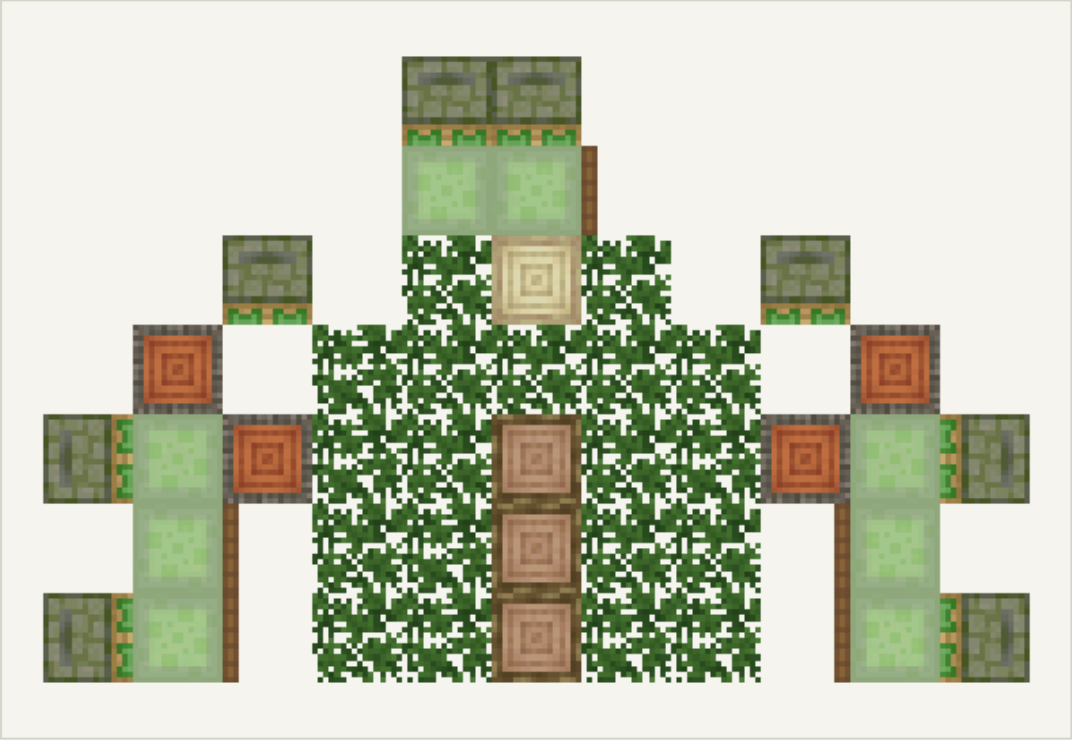
> Some architectures use both methods at once, such as PUTF -2. PUTF// also offers a different approach: adding one block to the main pusher, consolidating side branches into the middle 2 blocks, then pushing them out directly. However, this method doesn't work for cherry trees and isn't particularly useful in GTMC's 1.20.1

1.3 Jungle Architecture Requirements

Jungle tree trunks can grow up to **12 blocks** tall, so we need a tall enough main pusher to handle as many logs as possible.

> Multi-species tree farms that don't process full height do exist—typically 8 blocks tall (for easier oak height limiting) or 10 blocks tall (for easier wiring)—but we don't recommend this approach

Jungle saplings have a **1/40** drop rate, so we need extra leaf processing for jungle trees. The simplest approach is to use **double recursion** or **honey-slime walls** to cover as many potential growth positions as possible. From a top-down view, if **no more than 5 columns** of leaves within the 5x5 range go unprocessed, the farm can still collect enough jungle saplings.



@nutstory 5 columns cannot be processed

> This estimation method is somewhat imprecise. Jungle leaves are denser near the trunk. Strictly speaking, you'd need to calculate the average processing coverage across all possible leaf growth patterns. Generally, an average above 42 is enough to get sufficient jungle saplings

1.4 Oak Architecture Requirements

Oak trees can randomly grow into **large oaks** (*commonly called drumstick trees*), so we need to height-limit them to prevent this. The height-limiting block goes at **the 9th block above the dirt**.

> Designs that can directly handle drumstick trees do exist, but so far none of them can collect enough jungle saplings, so they fall outside the scope of multi-species tree farms

For our example, we'll use **honey-slime triple recursion** on the main pusher, **center consolidation** for side branches, **honey-slime walls** for leaf processing, and manually placed obsidian for height limiting.

> If you're familiar with older tree farms, you'll notice this is nearly identical to PTHSUTF's architecture. Given how varied redstone engineering designs can be, the best way to learn a particular design is to "imitate" it. In other words, when you're still learning redstone engineering, don't be afraid to "copy"—even just replicating the wiring of a well-built machine teaches you a lot

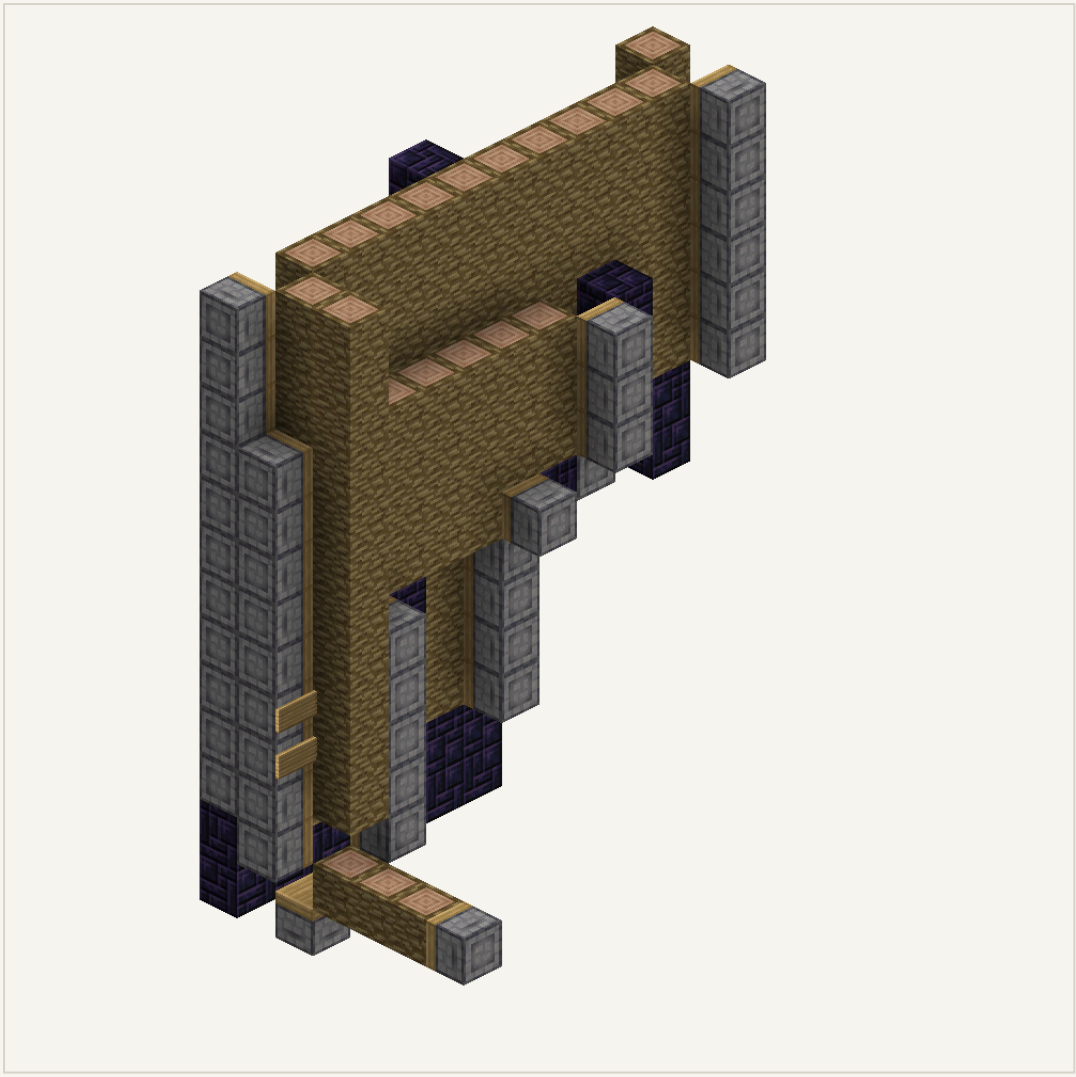
2 Block Stream to Drops Architecture Design

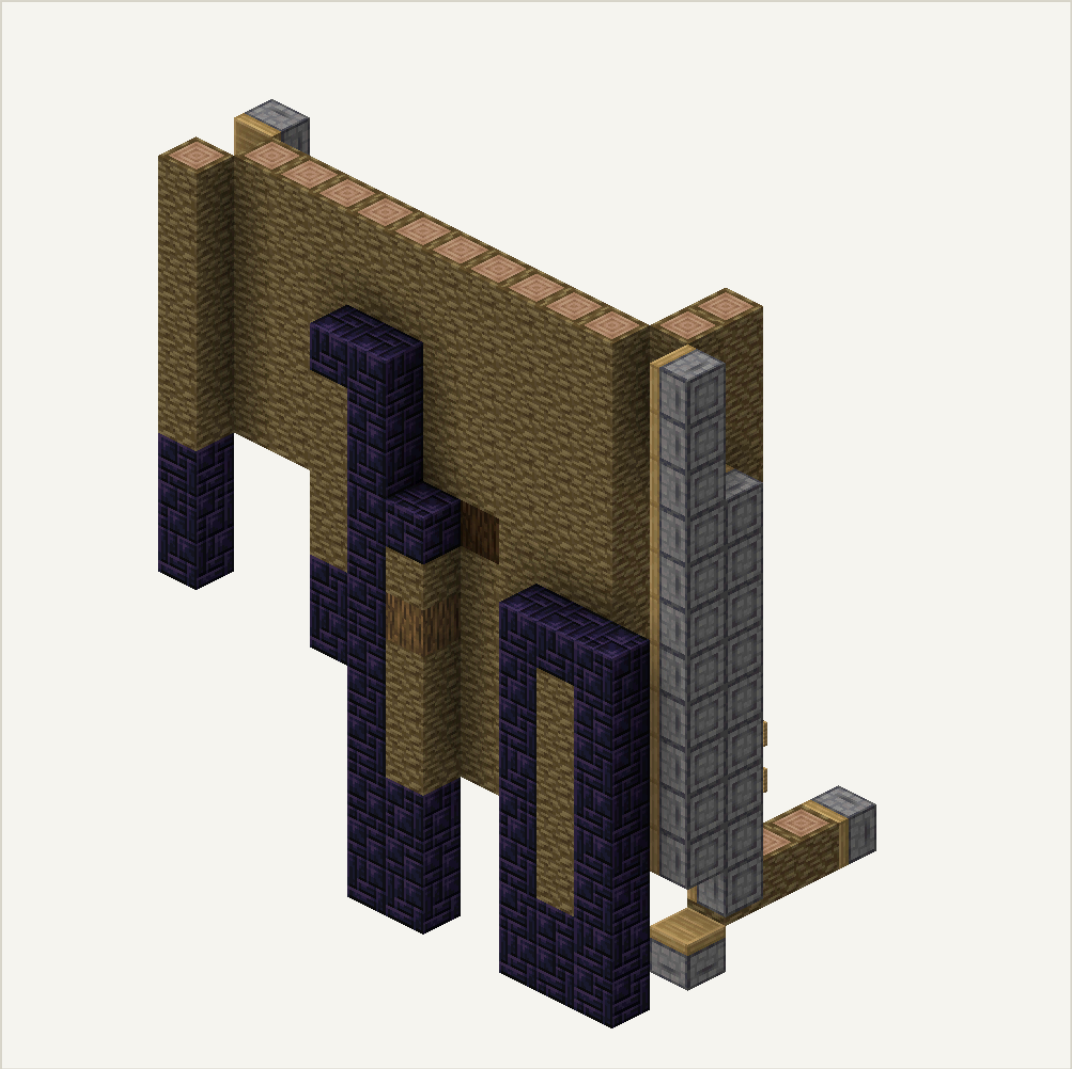
Because multi-species tree farms produce an irregular block stream output, we need to **reorganize** the block stream before feeding it into a specially designed explosion chamber.

2.1 Reorganization

In our example design, all logs are output to the front of the trunk, with at most two blocks output per operating cycle.

If you're familiar with DLHSUTF/PTHSUTF/DLPTUTF, you'll immediately think of using **two rows of pistons side by side** to merge the two log outputs (at unpredictable intervals) into a single 2-wide output. Alternatively, we can use a structure similar to a **side-suction base** to convert the two outputs with uncertain intervals into two outputs with a fixed interval (*typically 6gt*).





@Dawnemo DLPTUTF

> Since acacia's two logs on the same y-level can be distributed at the center and within 1 block around it (as shown below), even with additional log outputs, the trunk still needs this type of reorganization. For the extra outputs, we can simply add explosion chambers to handle their block stream to drops

2.2 Explosion Chamber

For now, we'll just attach some explosion chambers to process the block stream. We try to avoid pushing out a 2-block-thick stream at once, because **that greatly reduces the explosion chamber's processing efficiency and the final drop recovery rate**.

> For multi-species tree farms, we can use two specialized explosion structures: **pure milk explosion chambers** (commonly called "milk explosions") and **b36 explosion chambers** (commonly called "push explosions") to handle large volumes of irregular block streams. These are widely used in modern high-speed multi-species farms, but since nobody has updated the explosion chamber for our example, we'll cover them together in the speed-up chapter (

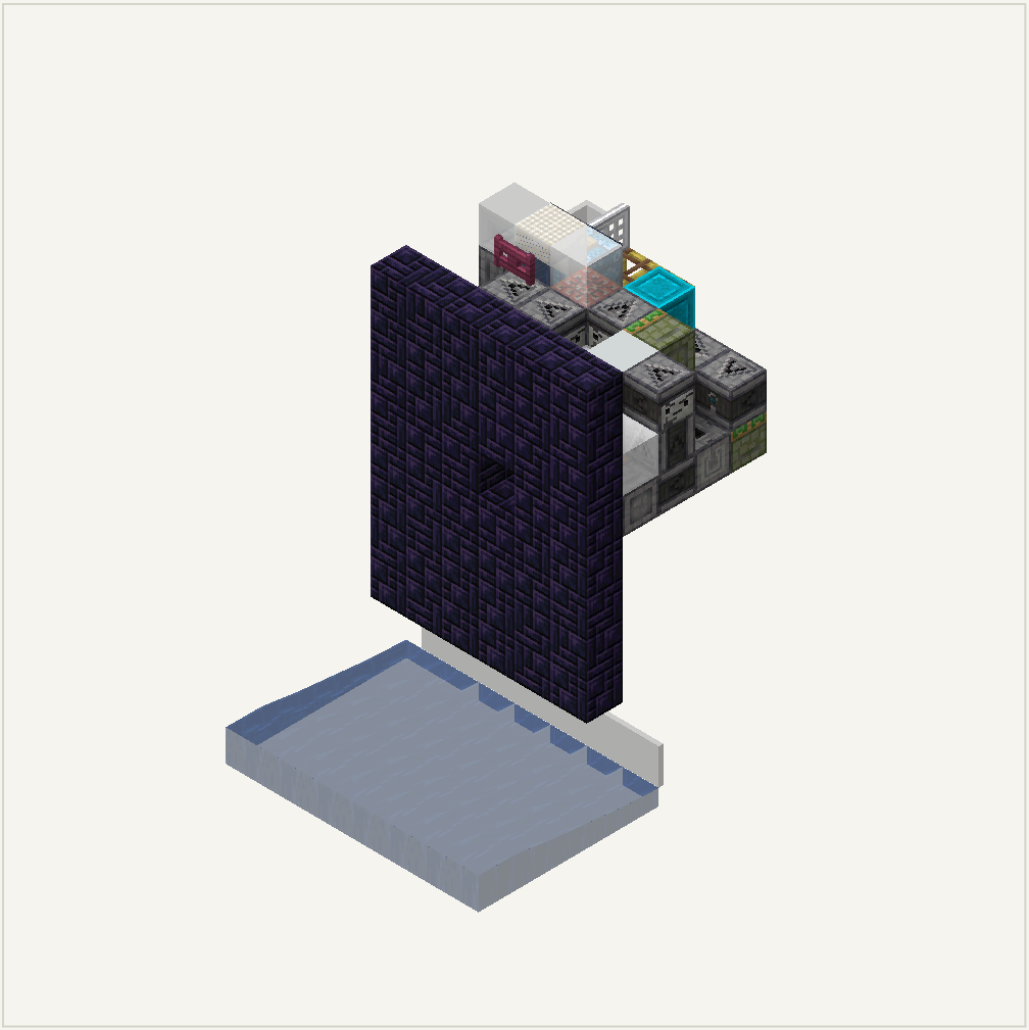
Milk explosion:



@从来只喝纯牛奶

@Qonctrol, Dreaming_Galaxy, BFladderbean, Feng_Bl PUTF std2

Push explosion:



@嘟了个嘟噜

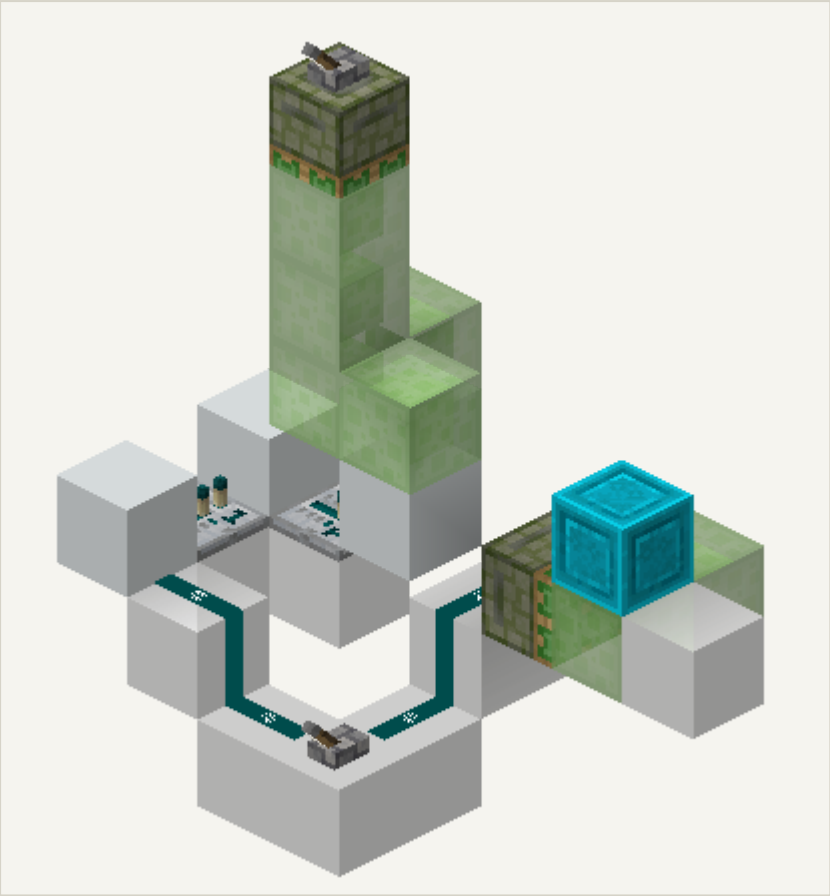
3 Wiring—Mode Switching and Speed Limiter

Once you've worked out the timing, you just need to activate each piston group as precisely as possible according to your design. Here, we want to focus specifically on the wiring for **mode switching**.

3.1 Acacia Timing Switching

If you look closely, you'll notice that **for every tree species except acacia, we can push out both side honey-slime walls (*hereafter called side walls*) at the same time, shortening the processing cycle**. We can go further and push the main pusher simultaneously too, cutting the cycle down even more. This creates different operating timings.

The solution isn't complicated. Switching from normal timing to acacia timing **is simply adding delay to the two side walls**. So we only need to run a second line with more delay to the side walls, then build a device that switches between the two lines.

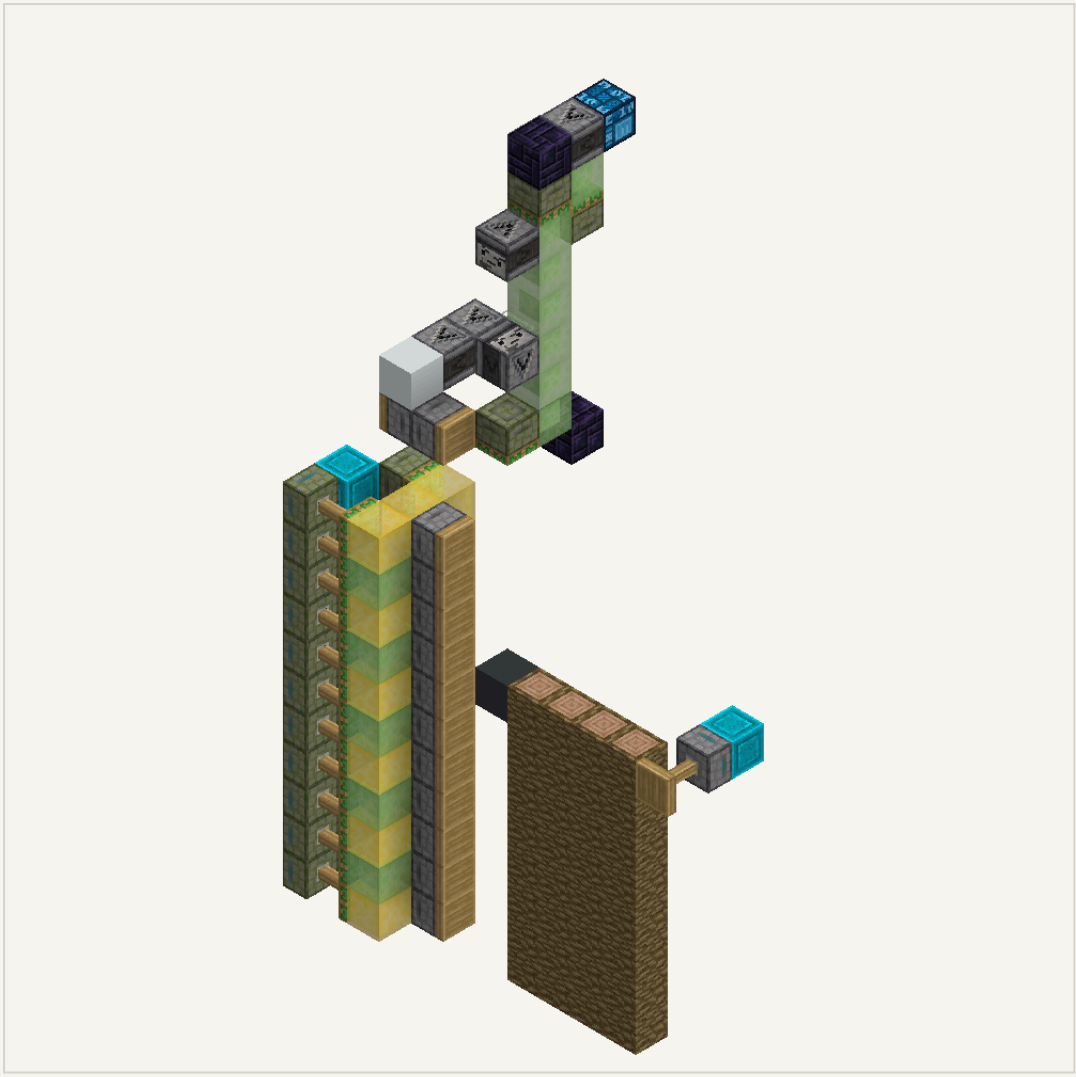


3.2 Automatic Oak Height Limiting

Many players find manually placing obsidian tedious. There's a solution for that: automatic height limiting.

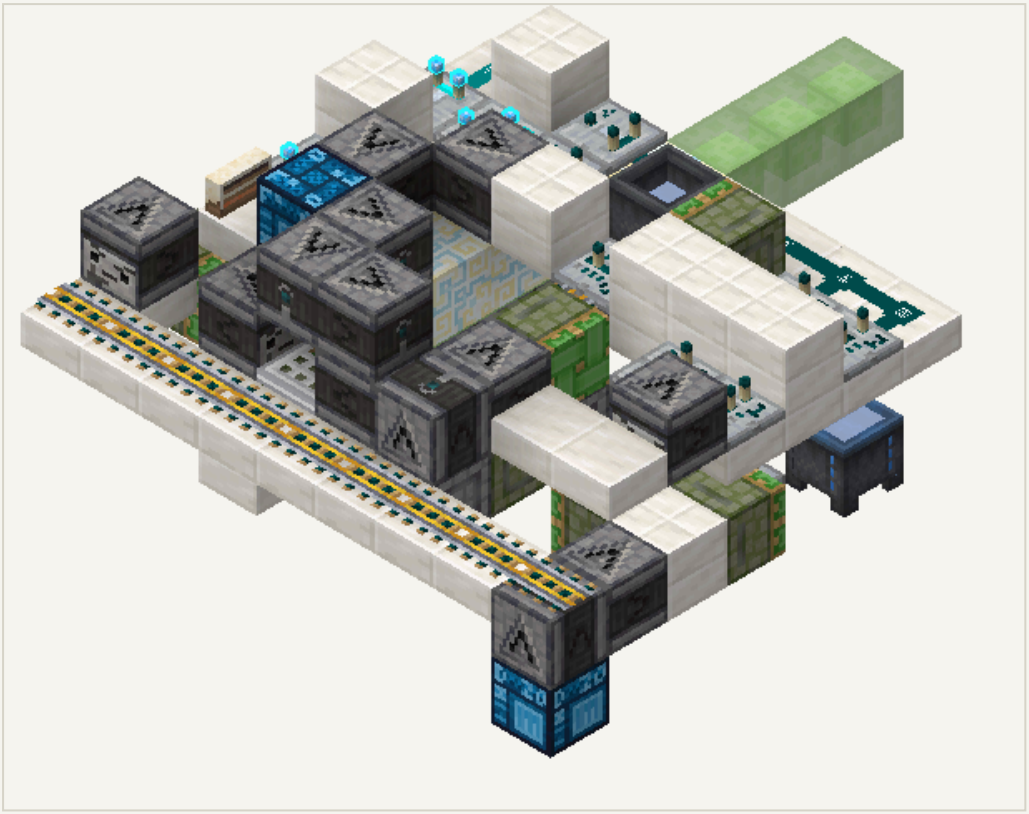
Since automatic height-limiting methods vary widely between designs, we'll only cover the one used in our example architecture.

The idea is simple: place a **block that the main pusher can't push** in the right spot to achieve height limiting. But to send that block down (usually via a flying machine), we can only use pushable blocks. The solution is to fill the main pusher's push direction with logs, then insert an **activated piston** on the reorganization side to prevent it from moving in the main pusher's direction.



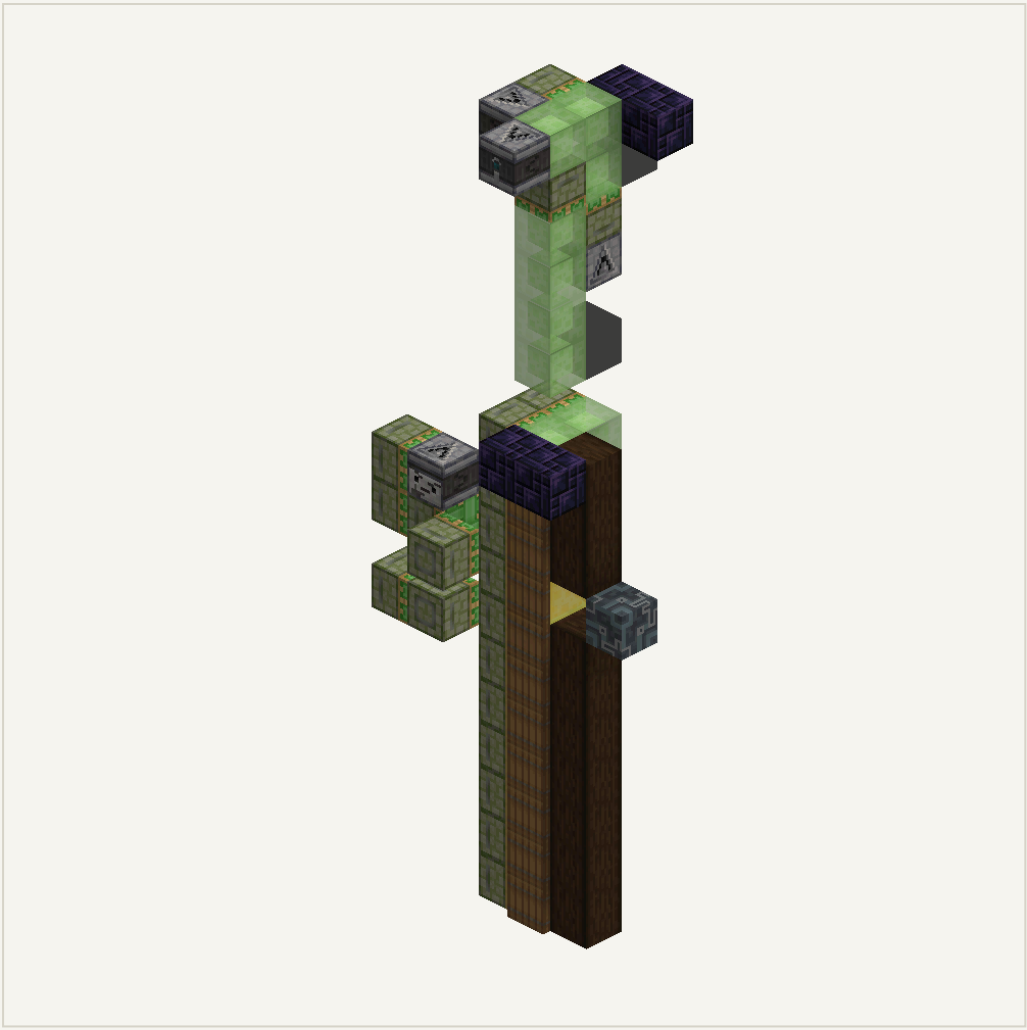
There are other solutions too. Here are some examples:

Slime stick cutting:



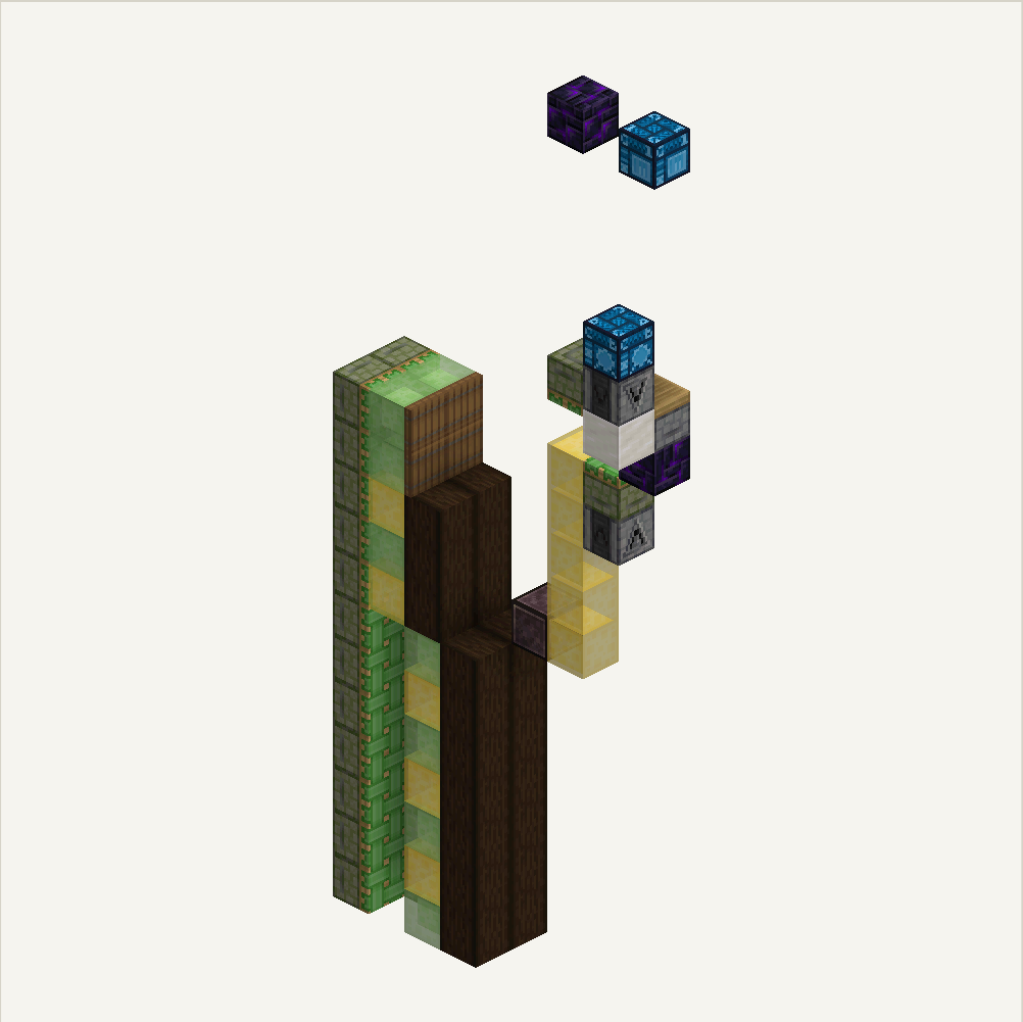
@Jellie_

Flying terracotta:



@XJH_Jorhai

Wall-cutting main pusher with flying machine insertion:



@Feng_Bl

3.3 * Jungle and Acacia Height Increase (1.14 and below content)

In version 1.14 and below, jungle and acacia growth detection checks a **3x3 area for the trunk and 5x5 for the canopy**. This means we can place blocks within the 5x5 area below a certain height (*typically 5 blocks above the dirt*) to **force** jungle and acacia trees that would grow too short to reach a minimum height, improving the efficiency of both species.

> Since this mechanism prevents jungle or acacia below a certain height from growing at all, it increases bone meal consumption. Also, even in 1.20.1, dark oak retains the same growth requirements (centered on the northwest corner sapling), so we can still "height-boost" dark oak

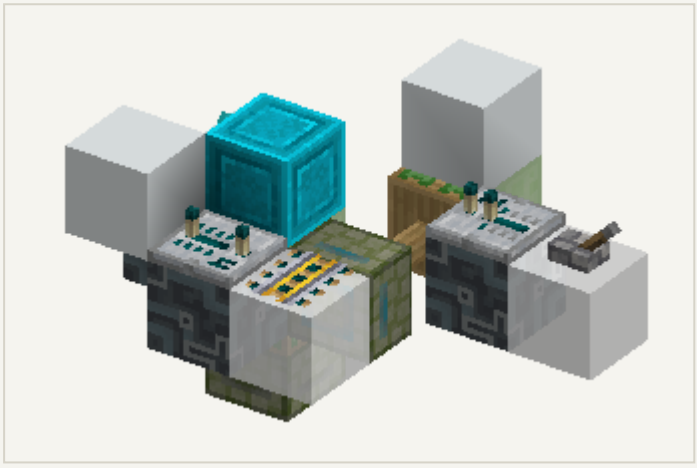
3.4 Speed Limiter

There's one more issue. Trees can grow before our architecture has fully reset, which would trigger the detection structure prematurely and break the farm. So we need to add a speed limiter to the detection structure. The limiter's timing must also account for the operating timing (reset time) of the rest of the farm in each

mode.

For QC detection, we can directly power the retraction piston, or attach something to the structure that pushes out after detection to prevent it from activating too early—like PTHSUTF's solution.

Since QC detection speed limiting is hard to modularize, building your own is the best approach (

> For comparator detection, we can use a similar approach—cutting off the signal output before a certain node
>
> For push limit detection, we can sever the connection between the detection structure and the dirt
>
> 
>
> 
>
> For BUD detection, if you want to ensure stability, you must use a 4gt reset BUD and add delay at the signal output. If you use a 6gt reset BUD, there may be cases where trees growing at 4gt fail to trigger—~~like the noob Galaxy's early powder-free perfect-timing oak modes and TT's 6gt jungle tree farm, which both broke this way~~

At this point, we have completed a relatively simple multi-species tree farm.

Here is the example we made!

7.4 Why Your Tree Farm Is Slow — Introduction to High-Speed Tree Farms

Barring any surprises, the tree farm you designed in the previous chapter probably tops out around 10,000 efficiency, with processing cycles easily stretching past 30 or even 40gt.

What's causing this? Let's break it down.

1 Before We Begin — 0-tick

In 1.1.2, we mentioned that Minecraft's timing is divided into **inter-tick timing** and **intra-tick timing**. If we can move functionality from inter-tick to intra-tick timing, we've successfully sped up a tree farm. The most important way to do this is through the **piston action and wiring pattern** known as **0-tick**.

1.1 The Concept of 0-tick

In 1.1.2, we also mentioned:

> BE components (pistons and note blocks) execute operations in the order they **receive updates confirming state changes**. This order is sometimes called **depth**. Note blocks do not increase depth (i.e., they don't delay the execution order of objects they update within BE).

So imagine a piston:

At `0gt BE`, it receives a **rising edge** signal at a certain depth. After it extends, it receives a **falling edge** signal at a deeper depth (deeper means later in the BE execution queue).

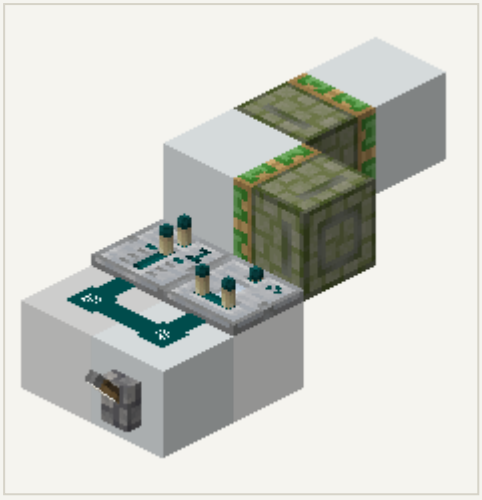
Obviously, it should extend when powered and retract when depowered. This completes a **0-tick action**. Specifically, for sticky pistons, the first block it pushes arrives **one piston depth deeper than the falling edge signal**, while pulling behaves normally.

> In fact, using piston-added depth as a depth unit isn't precise. Due to differences in piston push order, you can actually create depth differences smaller than "one piston" by creating b36 order during extension. Sometimes "coral" is used as a unit (because the most widespread use seems to be in coral-based TNT duplication). This is occasionally useful, but doesn't have a major impact on overall wiring.

1.2 Common 0-tick Generators

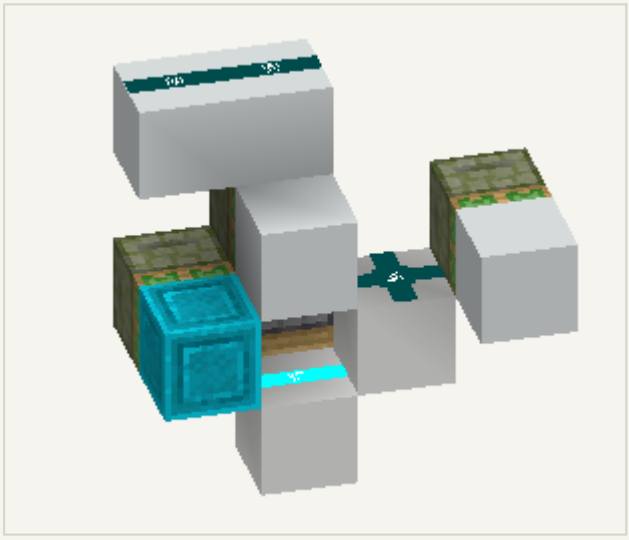
The idea is very simple: **before BE ends within the same gt**, give the piston a rising edge at an earlier point and a falling edge at a later point, and we get a 0-tick action.

For example, here's a very classic 0-tick generator based on **the execution order difference between comparators and repeaters in TT**:



Since repeaters generally have priority over comparators, at 2gt TT, BE gets added to the 0-tick piston first, then to the depowered piston. So the depowered piston must act later than the 0-tick piston, thus creating a **falling edge signal that comes after the 0-tick piston extends**, completing the 0-tick.

Here's another very classic 0-tick generator based on **Redstone Dust** (*hopefully you still remember that Redstone Dust is instantaneous*):



At `0gt NU`, the redstone block is placed. At `1gt BE`, the sticky piston removing the wire-pressing block moves. Since the sticky piston removing the redstone block is quasi-connected (QC'd), **BE only gets added after the sticky piston removing the wire-pressing block updates it**, thus creating a rising edge at **1 piston depth** and a falling edge at **2 piston depth** at `1gt BE`. This signal is enough for the powered piston to complete 0-tick. *Of course, you can also add depth to the falling edge through more QC pistons updating along the way, thereby activating pistons at the output end that can only be QC'd and are updated by other pistons.*

> Note that Redstone Dust turning only triggers PP updates. We need additional NC updates to the target piston being powered to complete the rising edge signal (that is, for pistons, rising edge and falling edge signals actually refer to receiving NC updates and confirming the need to extend or retract).
>
> 
>
> Actually, the 0-tick here refers to the depth at which the normal piston updates the sticky piston, not the depth of the Redstone Dust turning (although there's no big difference here).

2 "Base"

For high-speed tree farms, we need to design **bonemealing**, **detection**, **trunk processing**, and **sapling cycling** as an integrated unit, called the **base**.

2.1 Trunk Processing

This time we need to choose the trunk processing mode first.

In 1.2.2, we mentioned:

> Trunk processing generally has four methods: processing together with the trunk, **upward push**, **downward suction**, and **side suction**.

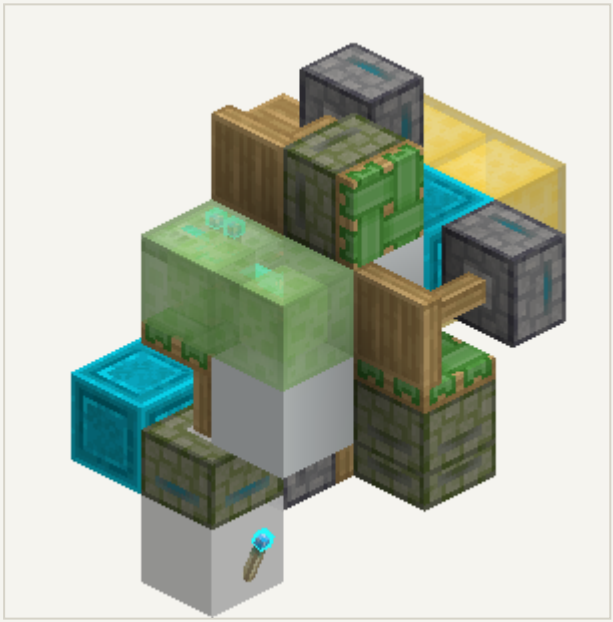
If you think about it, processing together with the trunk means **we can only use one dispenser for bonemealing**. With **upward push**, we can plant trees at the earliest at **6gt** of tree farm operation. **Downward suction** gets us to **3gt**, and **side suction** can be as fast as **0gt**. In high-speed tree farms, we generally choose **downward suction** or **side suction**.

Downward suction:



@Scorpio 天蝎君

Side suction:



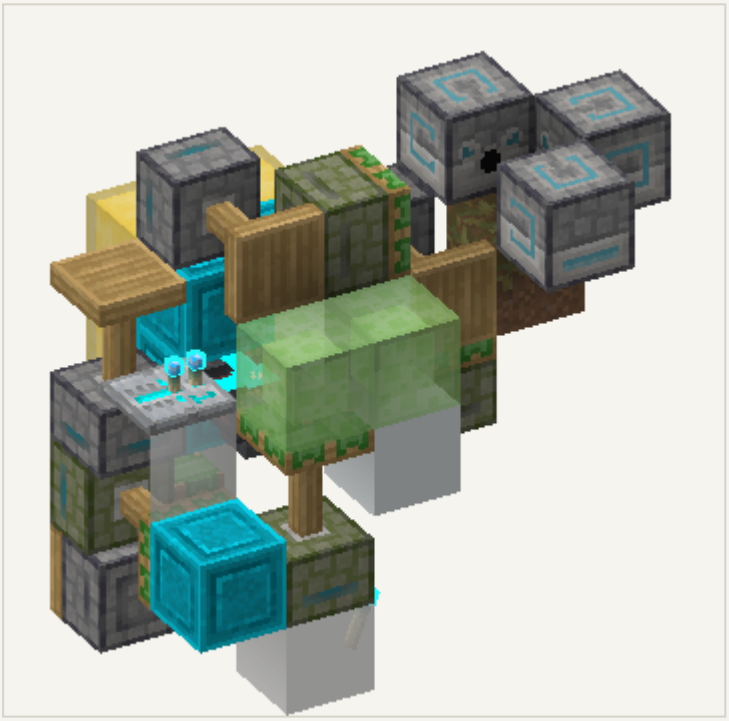
@Qonctrol, Dreaming_Galaxy, BFladderbean, Feng_Bl PUTF std2

> Since player operations happen in the NU phase and dirt returns to position in the BE or TE phase (at the end of the gt), we can plant trees in the same gt that dirt returns.

> In fact, upward push has problems beyond just slowness. Remember that jungle trees can grow up to 12 blocks tall? Including dirt, that exceeds the piston push limit. And remember acacia side branches? Upward push can result in up to three logs at the same y-level, greatly increasing processing difficulty. That's why upward push is now basically abandoned.

2.2 Bonemealing and Sapling Cycling

~~Although Scorpio really wants to explain the bonemealing mechanism, I don't think it's necessary, because~~ bonemealing in high-speed tree farms essentially means **stacking dispensers**. Simply put, **don't worry about bone meal consumption**. Place as many bonemealing dispensers as possible to increase the number of **pre-bonemealing** cycles between when the player plants the sapling and when it can grow, and to widen the **growth window** after the sapling can grow (this is why we introduce **cross bonemealing**, which staggers when dispensers fire bone meal, generally by 2gt).



@PUTF std2



@Scorpio 天蝎君

Self-explanatory cross bonemealing/synchronized bonemealing switching device diagram.jpg

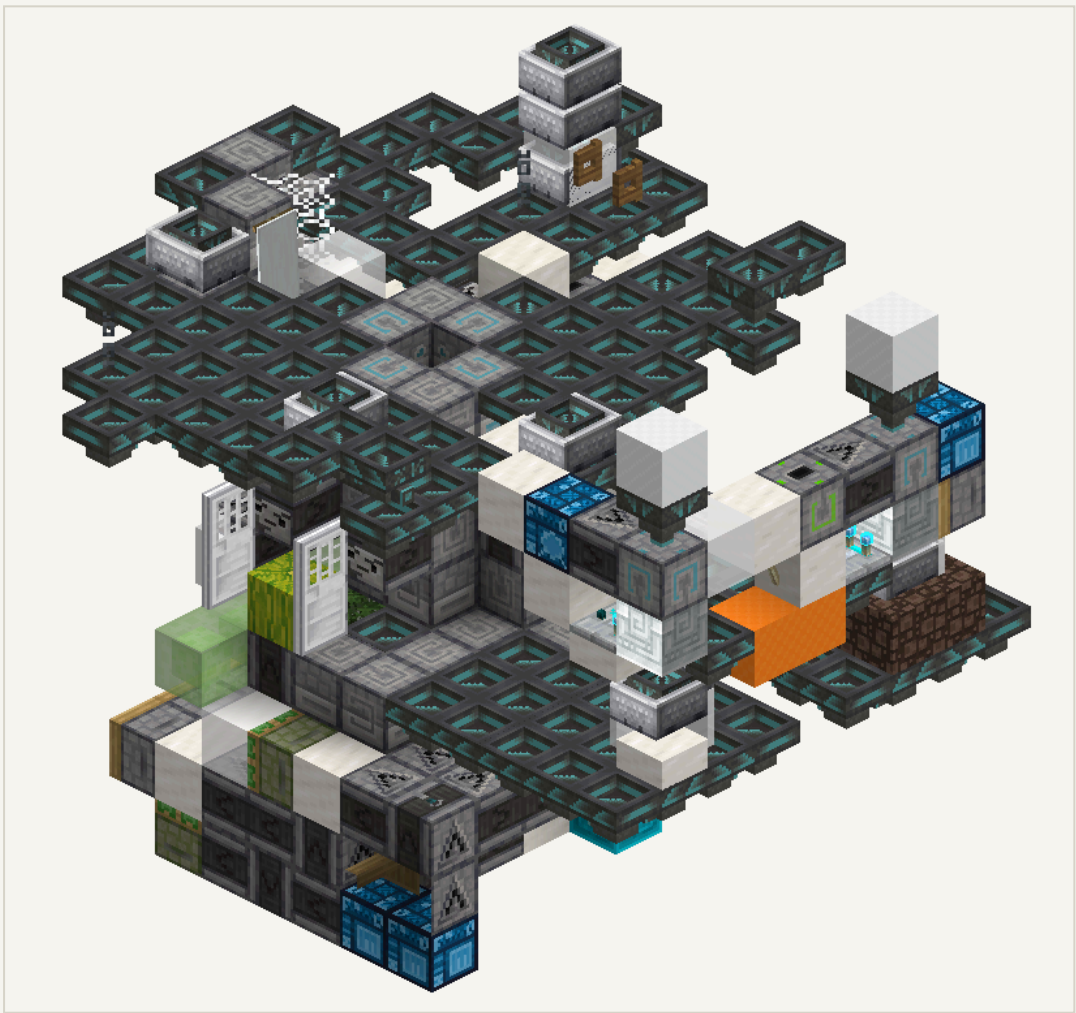


The biggest problem high-speed tree farm bonemealing structures actually face comes from **sapling cycling**. Crowded base space causes the bone meal supply chain and sapling cycling hoppers to **constantly clash**. Generally, we design the bone meal supply chain to be as neat and compact as possible, leaving more space for sapling cycling. ~~In fact, there's nothing that can really be written into textbooks; you just have to do it yourself a few times.~~

An example of a very compact bone meal chain:





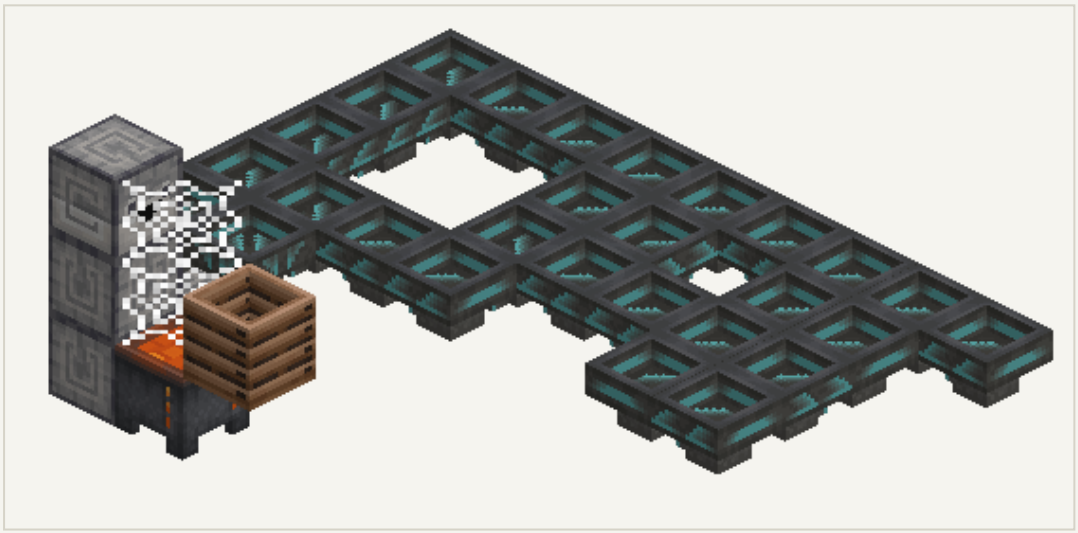


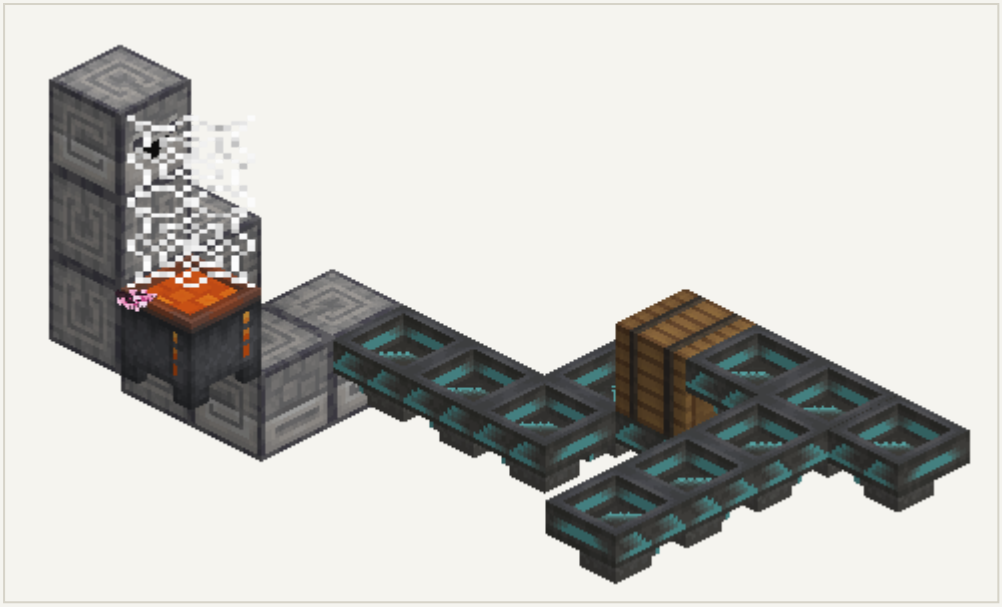
@Islyric, Qonctrol, BFladderbean PUTF+

You can see that even a very compact bone meal chain still gets packed with other components.

For sapling cycling, since ~~Mojang's annoying random changes to leaf drops caused~~ leaves dropping sticks, we need more droppers to handle the large quantities of items. We also need to "straighten" the hopper chain as much as possible and adopt a zoned collection strategy to maximize item transfer efficiency:





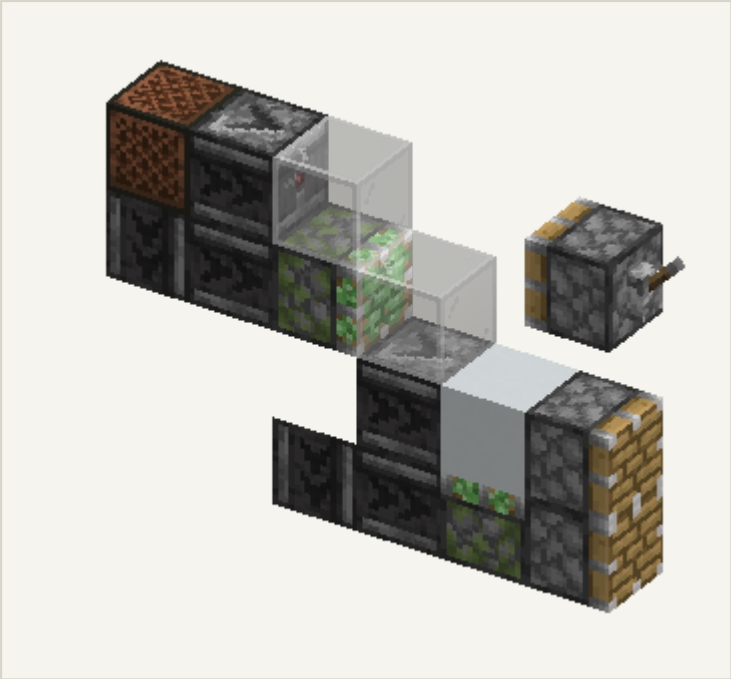


@Chuo_di, Islyric, Qonctrol, BFladderbean, Sarumi_QAQ PUTF std2c

In fact, sapling cycling is also affected by the trunk and leaf processing architecture. For high-speed tree farms, you sometimes need to add hoppers and hopper minecarts for collecting saplings to the trunk and leaves processing wiring.



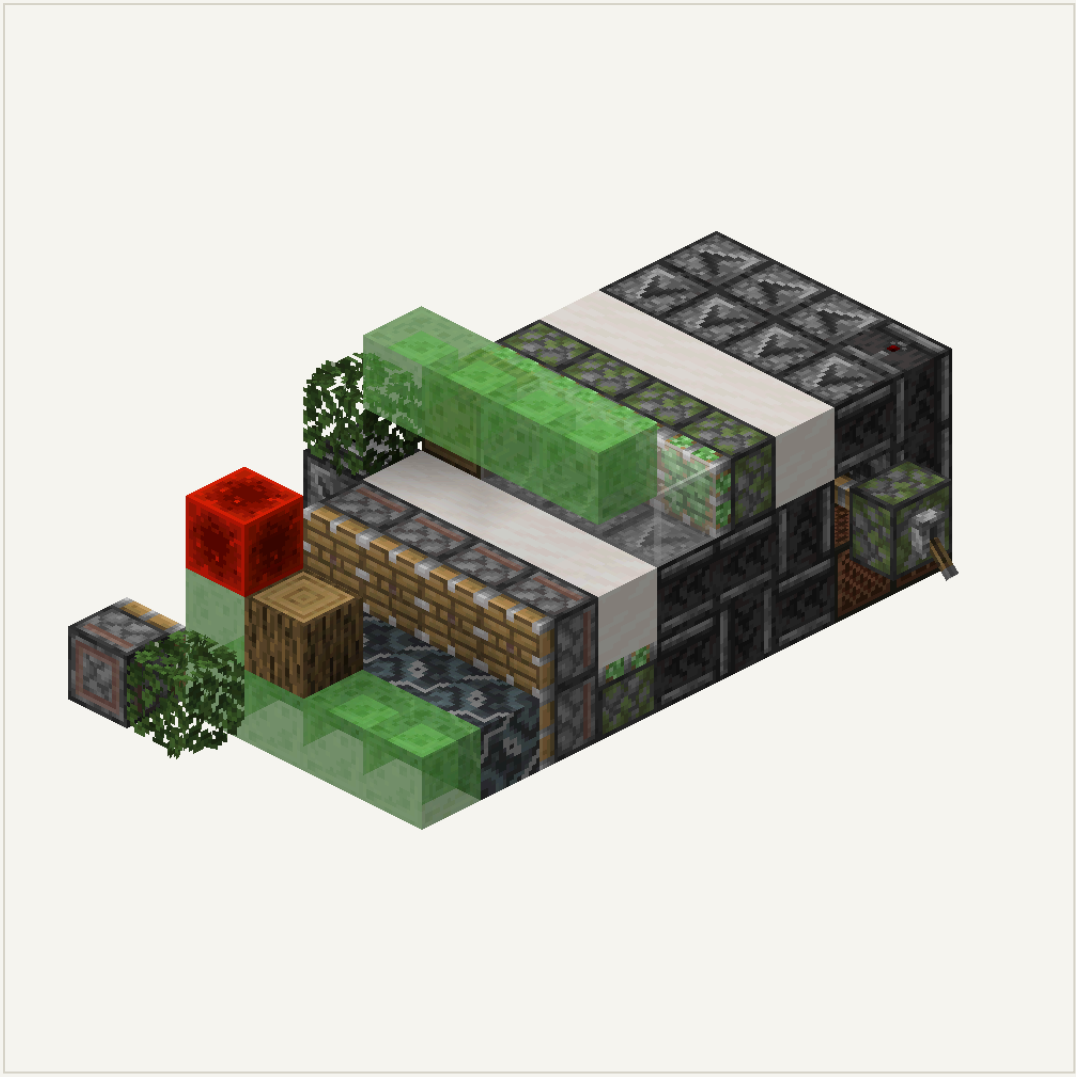
2.3 Push-Limit Detection



This is a push-limit detection unit made by Bright_Observer

> When the upward-pushing piston is updated during the TT phase, it plans its push. When a piston plans to push, it checks whether it has reached the push limit. At this point the glass above hasn't retracted yet. If the lever is pulled down and the piston extends, it will be unable to push. This activates two normal pistons, which update to the sticky piston during the BE phase. The sticky piston self-checks again and extends. If the lever is released, everything proceeds normally: the sticky piston extends normally, whether the upper of the two normal pistons extends depends on orientation, and due to depth influence the lower piston never extends. This completes the detection. — GTMC Timing Theory 5.10.1

In summary, this unit runs at an **8gt cycle**. To align with the cross bonemealing clock, we can stack four layers of this unit, each staggered by 2gt, place them under the dirt, and connect them with slime block linkages. This gives us a detection module that takes up almost no dispenser or other wiring space. ~~\Magic Detection/~~



@Islyric

3 Core Architecture — Trunk and Leaves Processing

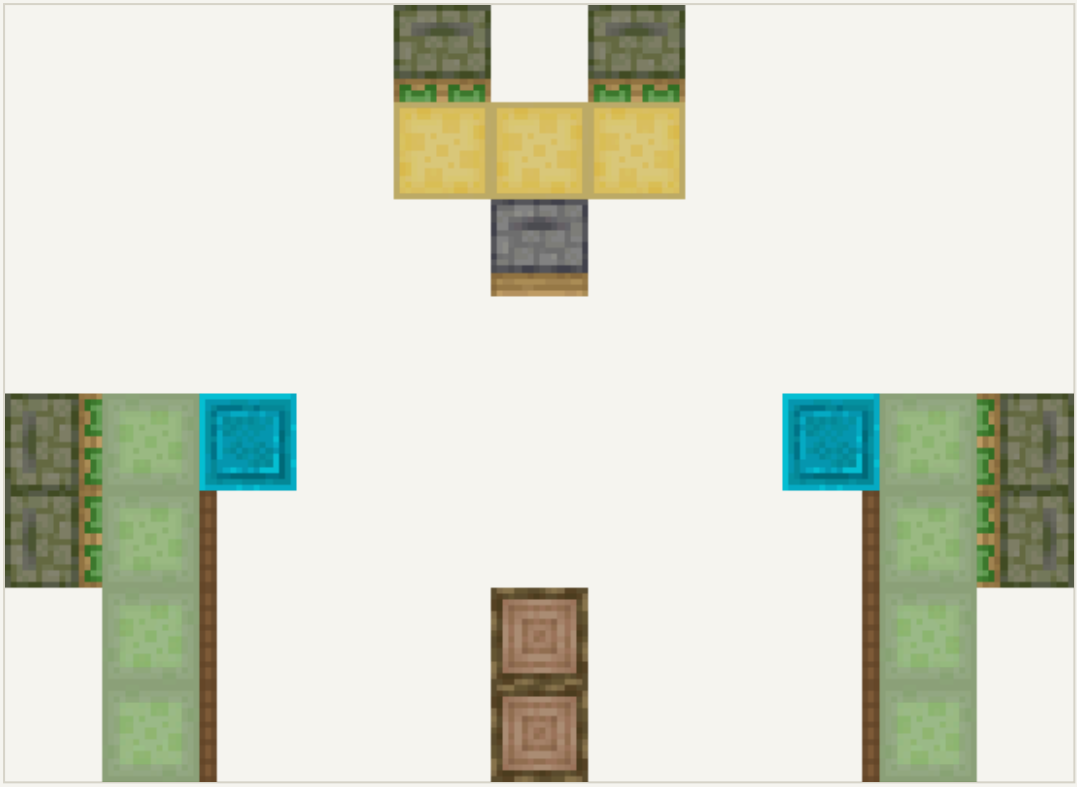
> This is the part that determines the minimum operating cycle of a tree farm. Think carefully about this.

Before designing the architecture, we first need to understand how to analyze an architecture's timing.

3.1 Timing Design

For a given architecture, we need to analyze how it processes trees and design the shortest possible timing.

Taking the PTHSUTF architecture as an example, let's try to design the shortest processing timing for acacia (review processing principles in 3.1.2):



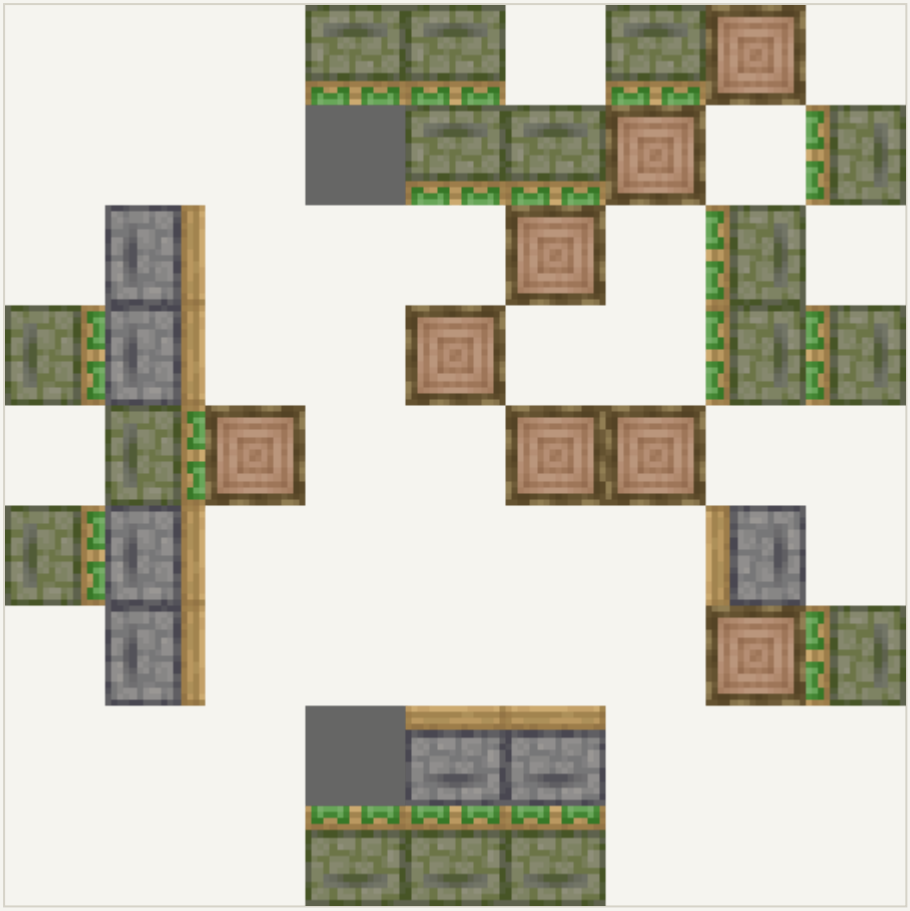
@Fallen_Breath, CCS_Convenant

`0gt` Main push and side wall first level extend *Perform first-level centering on all side branches*
`3gt` Main push and side wall first level in place, main push second level extends *Let main trunk direction side branches center first to avoid conflicts*
`6gt` Main push second level in place, left side wall second level extends *Center side branches still not pushed out to the center and right one block*
`9gt` Left side wall second level in place, activate main push third level and begin retracting, making main push third level 0-tick; then, right side wall second level extends *Clear side branches located in the center and center side branches in the right one block to the center*
`12gt` Right side wall second level in place, activate main push third level and begin retracting, making main push third level 0-tick *Complete processing, begin reset*
`15gt` Main push second level begins retracting, side wall first level begins retracting
`18gt` Main push first level begins retracting
`21gt` Reset

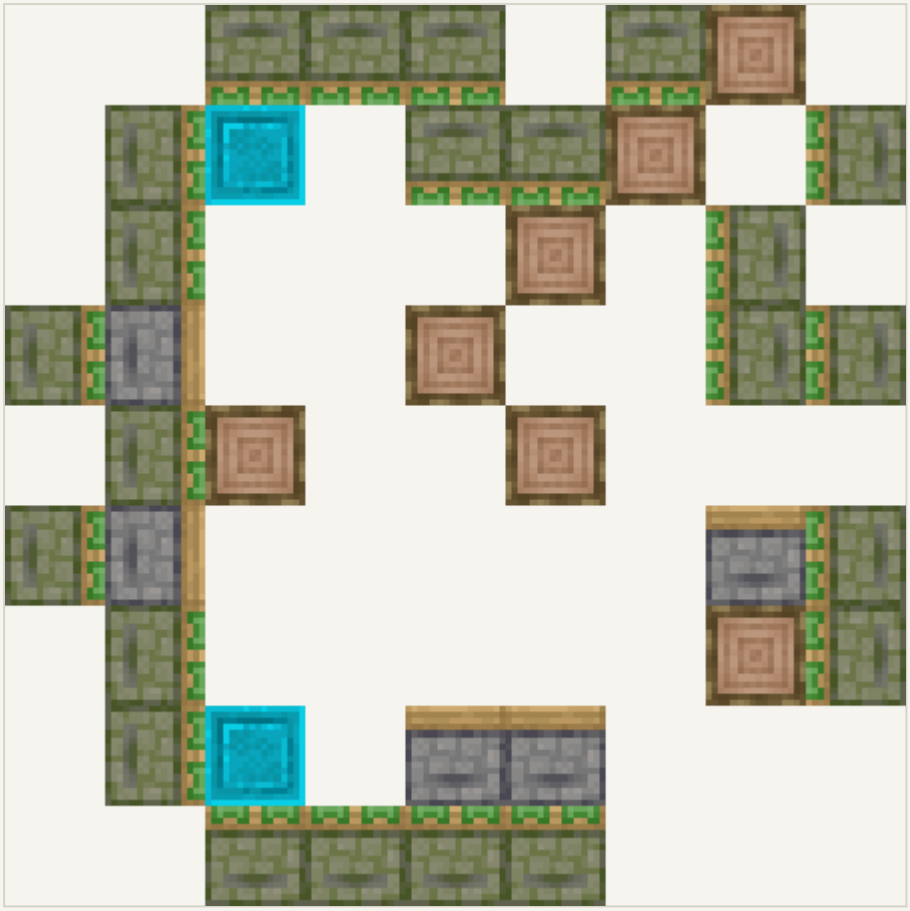
> In tree farms, we can generally think of time in 3gt units, although this isn't strictly correct. In fact, this architecture's acacia timing only needs to satisfy: at 6gt the first side wall second level extends, and at 9gt one side wall retracts while the other extends, to complete a 21gt reset. This isn't the only timing design method.

Try designing the timing for TT1998's 6gt jungle architecture *(Note: the image below is an architecture modified by @大尸凶一只, but it doesn't affect timing analysis)*

First layer:



Second layer:



(Here we only analyze the trunk processing and "redstone block wheel" timing):

`0gt` Side pseudo-double-recursion first level 0-tick, then second level 0-tick, extracting the log in front of the front pseudo-double-recursion; then the front pseudo-double-recursion pulls away the log at the trunk position; redstone block wheel first piston extends, activating the leaves processing piston on the other side, then the second piston extends, 0-ticking the leaves processing piston and activating the trunk processing piston, finally the third piston 0-ticks to retract the redstone block, 0-ticking the trunk processing piston

`3gt` Two pseudo-double-recursions first level 0-tick retract; redstone block wheel last piston 0-ticks to retract the redstone block

`6gt` Reset

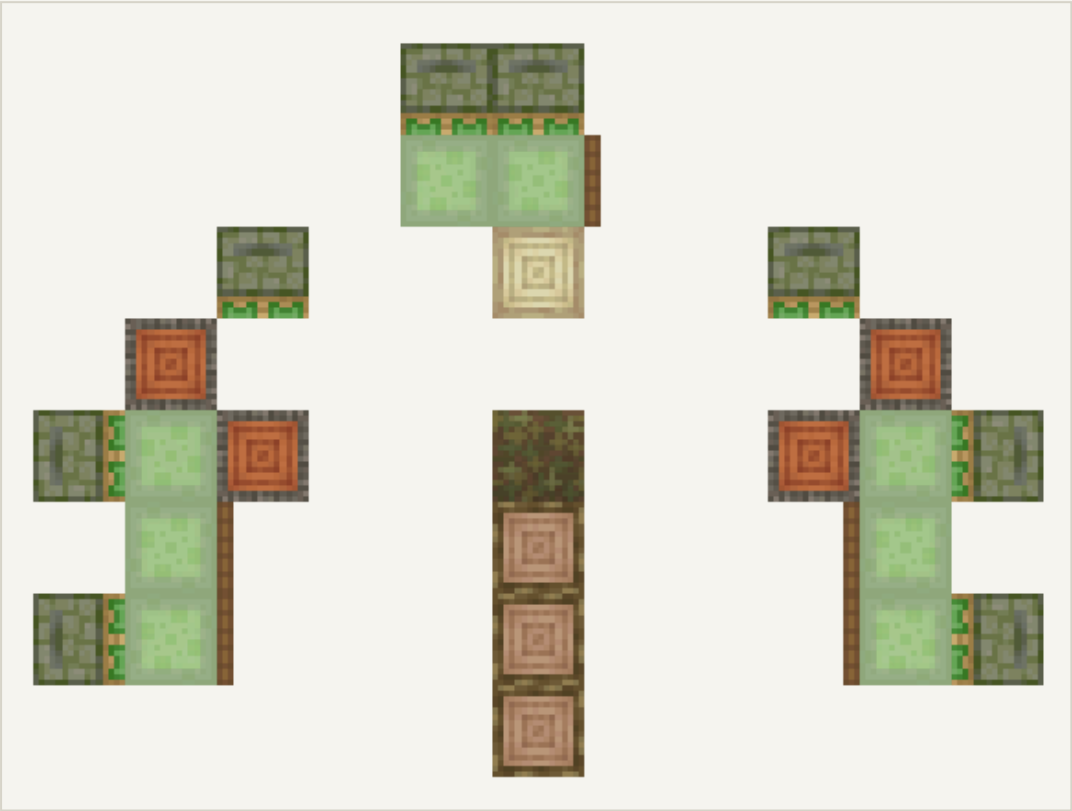
Performing timing design on a given architecture is very useful for wiring. It also helps you understand where the **culprit** behind a long operating cycle is, enabling targeted optimization.

3.2 Architecture Design

If timing design on existing architectures still doesn't get you to the speed you need, you need to design **faster architectures**.

Here are two classic approaches to speeding things up:

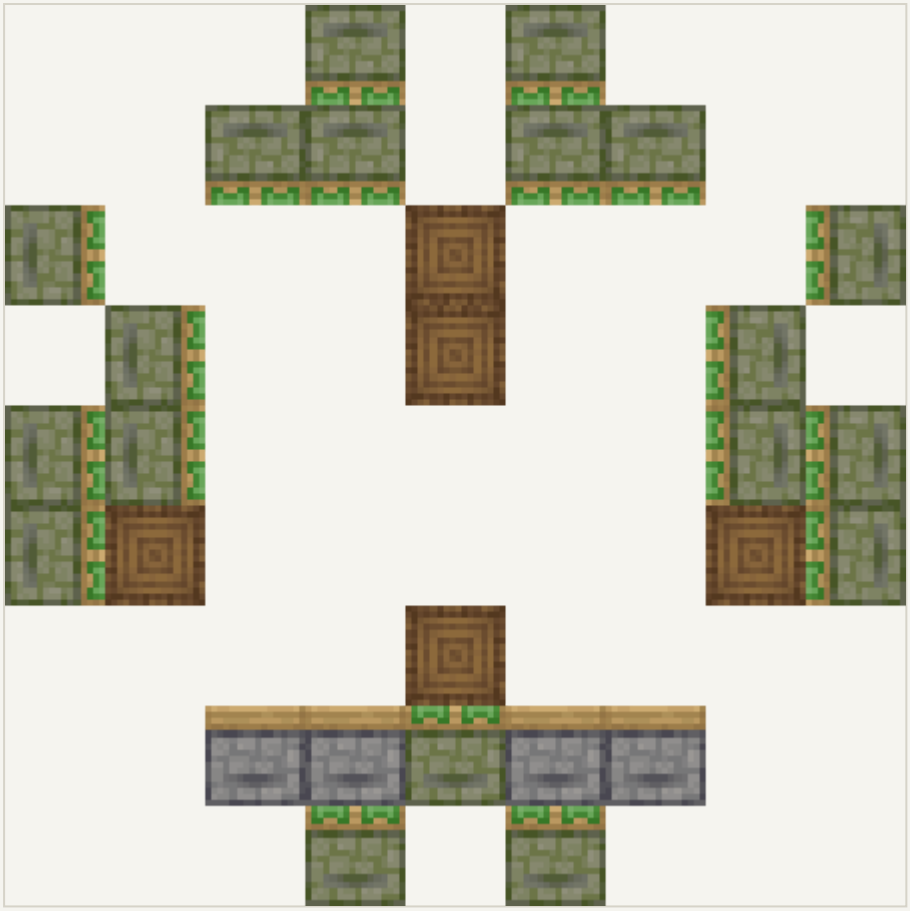
- Nutstory architecture: In 3.1.2 we mentioned that acacia side branches can be handled not only by centering but also by **increasing log output**. Now that we've learned timing analysis, you should notice: this architecture doesn't need timing for side branch centering, so it can run at 12gt, much faster than the PTHSUTF architecture.



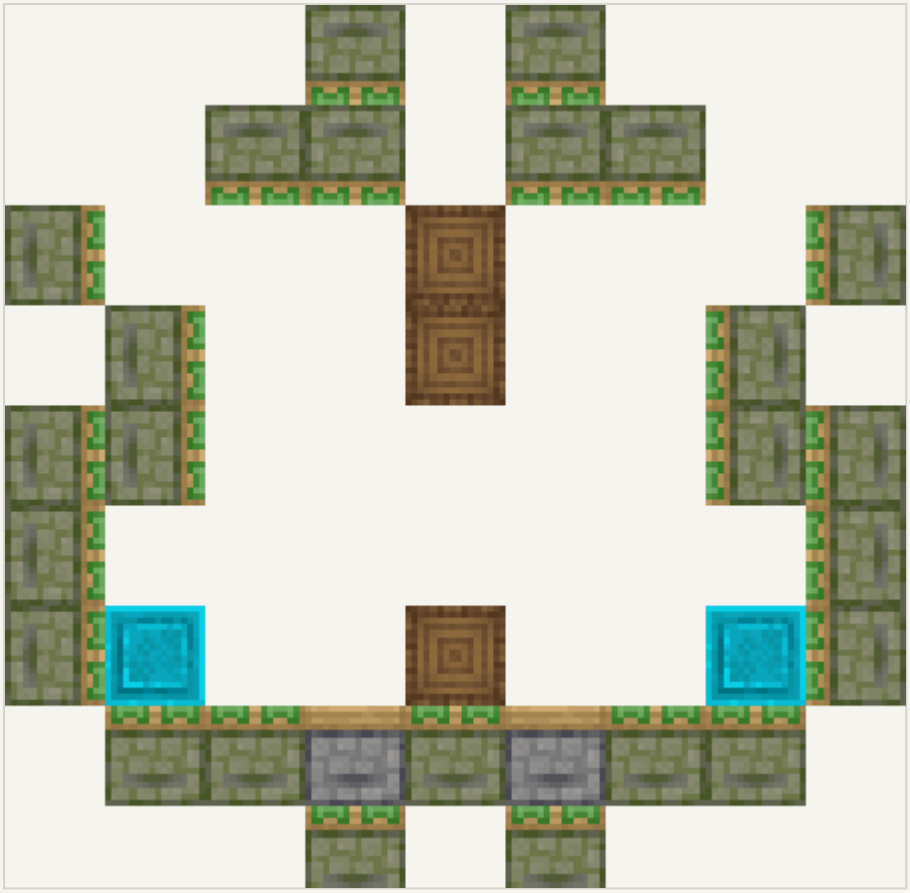
@Nutstory

- TT1998's 6gt/Shixiong's 4gt architecture: They're actually the same architecture. We've already done timing analysis above. Additionally, at 4gt, **the second-level piston of the pseudo-double-recursion has already been removed and created b36 at the target position**, so saplings can grow normally. As long as we build another pseudo-double-recursion on the other side and solve the sapling problem, we can easily get a 4gt tree farm.

First layer:

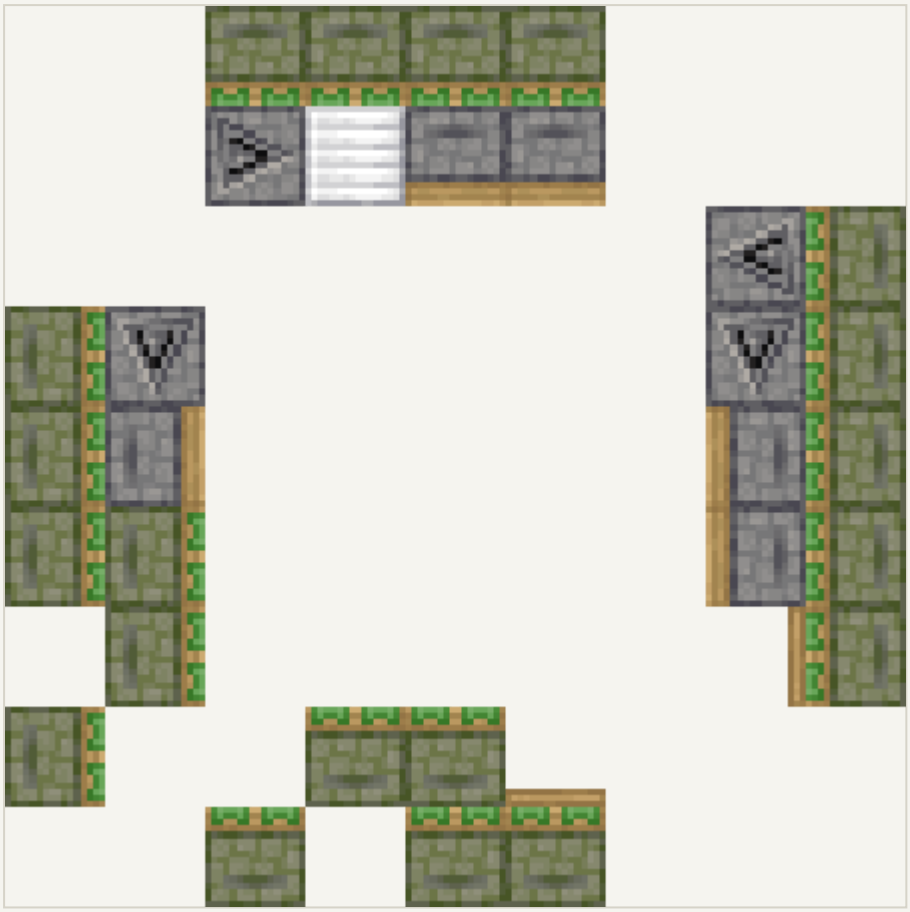


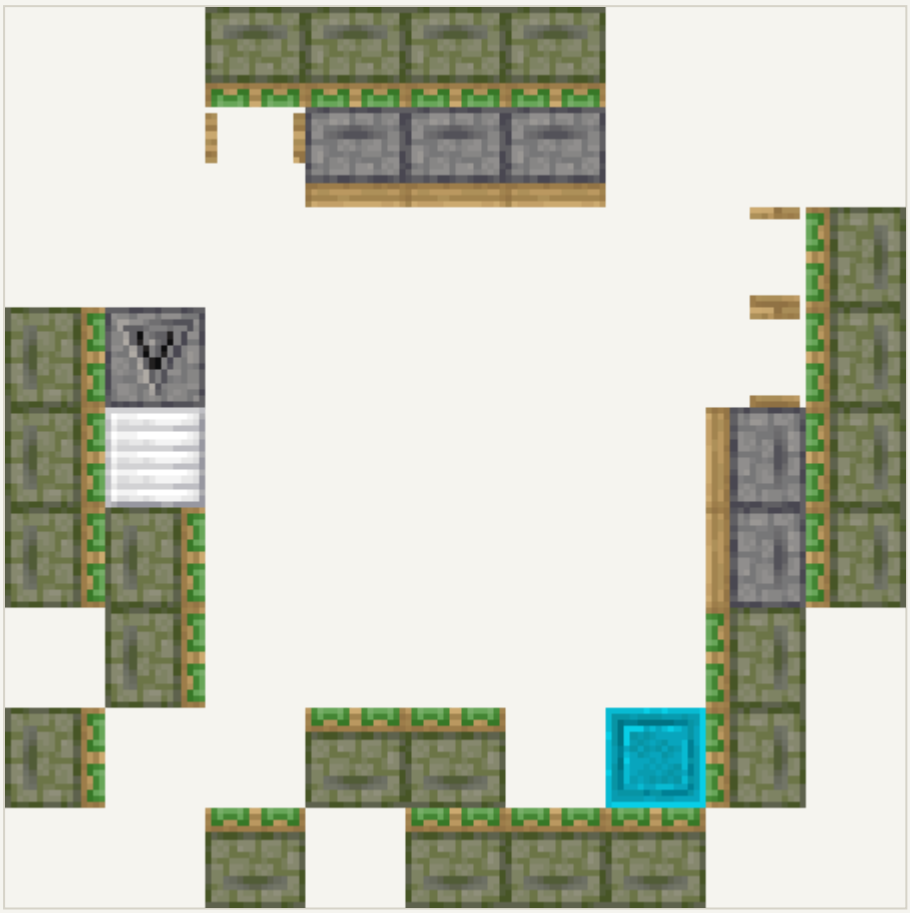
Second layer:



In fact, designing faster architectures comes down to a simple idea: **reduce piston actions that must occur at different macro timings**. Be sure to think about the corresponding timing while designing the architecture, **otherwise you'll easily lose track of what you're doing**.

In fact, timing and architecture design also have their exceptions. yunj's STFU is an 8gt all-tree-species tree farm, with some architectures not running on 3ngt timing. ~~But since we now have 4gt tree farms, we don't need to worry too much about this~~





For side-branch trees, we don't need to obsess over the minimum operating cycle; we can instead increase side branch processing capacity. ~~You can totally build a 12gt cherry blossom tree farm that outperforms certain 6gt ones~~

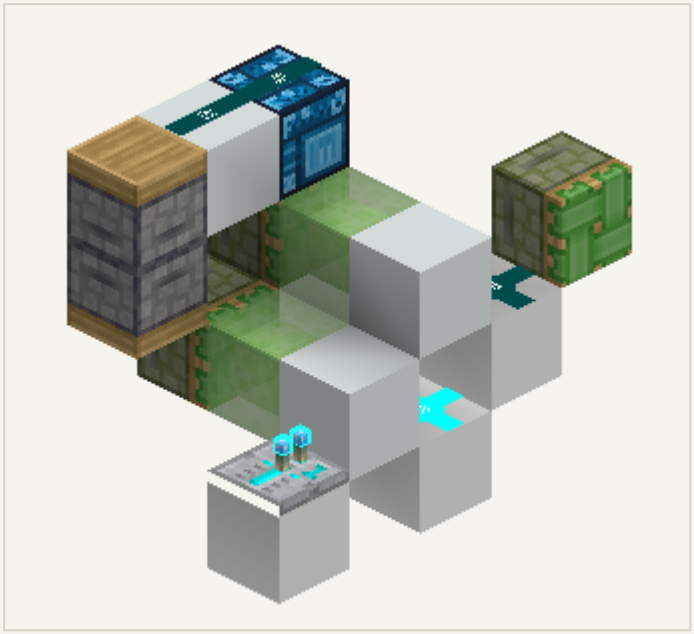
In summary, an excellent architecture is the **most important prerequisite** for a good tree farm. Before starting wiring, **optimize your architecture as much as possible**.

4 Wiring

4.1 Wiring Methods

Let's look at the simplest and most useful high-speed tree farm wiring method:

This is a very classic **double-edge 0-tick generator** based on Redstone Dust and depth (i.e., both rising and falling edges of the input signal get converted into a 0-tick signal). We can connect a sticky piston to its output, pushing and pulling a redstone block, to get a signal with **the same macro timing** as the input that **doesn't increase macro delay but has a deeper rising edge**. Using this, we can easily control wiring depth: **just connect the Redstone Dust that activates pistons in sequence, following the designed depth order in the same way**.



Classic example based on this wiring method

4.2 Wiring Inspection

Although we keep saying "economic base determines superstructure," we can't design a 6gt architecture and end up with wiring that runs at several dozen gt, can we? Analyze your wiring and see **exactly where the timing problem is that's dragging down the overall tree farm speed**.

However, for high-speed tree farm wiring, we can't give you a more specific method. **The best approach is to study and wire a few yourself**.

7.5 Why Your Tree Farm Stalls — Introduction to Dustless Wiring

Many of you have probably been waiting for this. Dustless tree farms have become a popular way to "flex" in recent years. However, Xinghe has always emphasized:

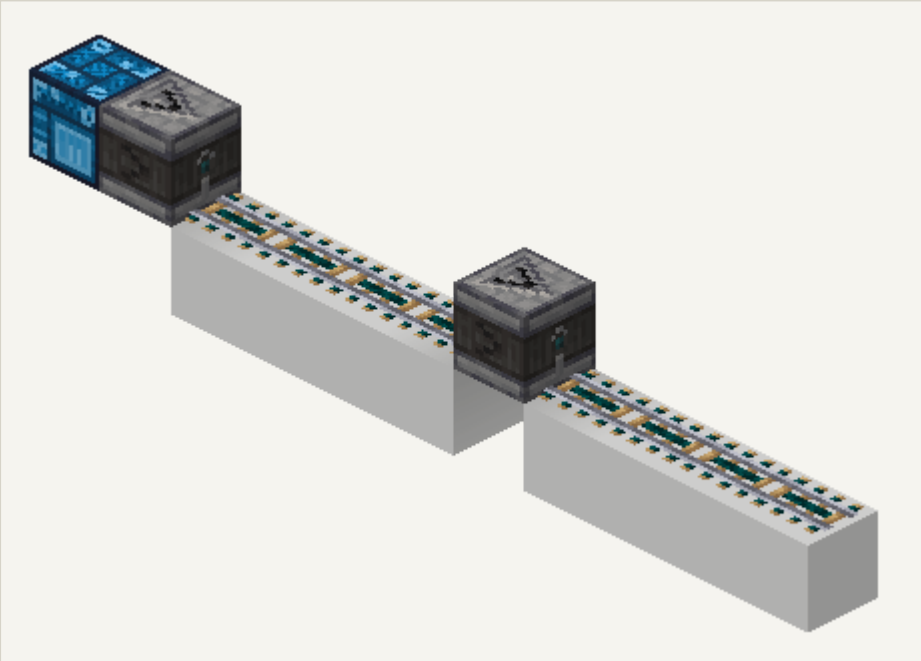
> Dustless has never been the goal—it's a means to an end. The ultimate objective should be to reduce lag, not to force dustless designs. Forcing dustless designs can easily lead to increased lag, which goes against the very point of dustless wiring.

Once we understand this, we can truly make good use of dustless wiring.

Now let's walk through the main methods of dustless wiring.

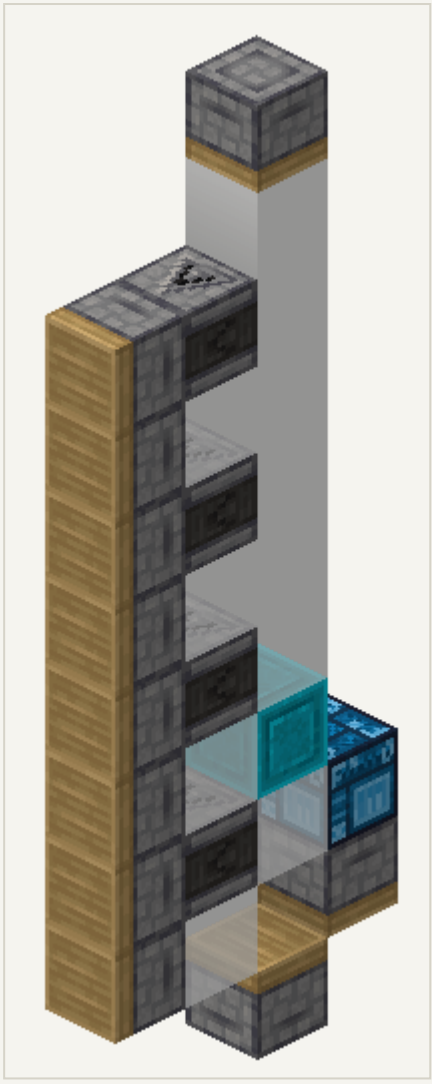
1 Rails and Observers

Rails **emit PP updates simultaneously** when their activation state changes, and can be detected by Observers. Together, they form the most basic dustless signal transmission method.



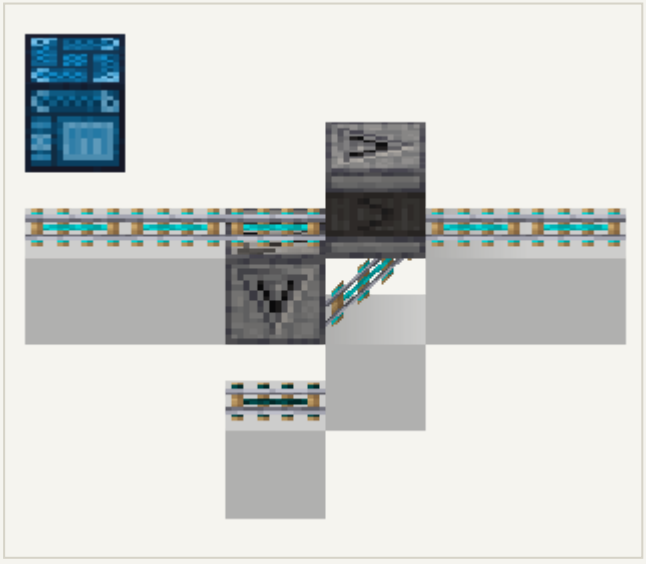
In tree farms, Pistons are generally arranged vertically. We only need to activate a vertical column of Observers simultaneously to activate an entire row of Pistons at once. There are two basic approaches: pushing **the Observer itself** or **the block the Observer is facing** with a Piston. ~~Resetting this row can be done casually~~





Since Observers have a built-in 2gt delay, we can use them to build macro-level timing circuits. There are already many examples of this, so we won't analyze them in detail.

You can also use BUD rails for zero-delay signal transmission:



2 Tree Power / Scaffolding Power / Wall Power

There's also water power, but since nobody uses it, we'll skip it here

2.1 Tree Power

Starting from 1.14, after receiving a PP update, leaves schedule a tick (if in TT, it executes immediately; otherwise it executes **in the TT, 1gt later**) to check their distance from logs. If the distance changes, they emit **NC and PP updates**. In tree farms, we generally push or pull logs with Pistons or Slime Blocks to trigger distance changes.

> In fact, tree power's update mechanism works somewhat like Redstone Dust: it checks the distance of surrounding logs and leaves. If there's an adjacent log, it sets its own distance to 0; if there are no logs around, it sets its own distance to the minimum distance of adjacent leaves plus 1.

With tree power, we can create many useful **odd-gt delay** devices, such as making an upward-pushing column start pushing downward at 3gt.

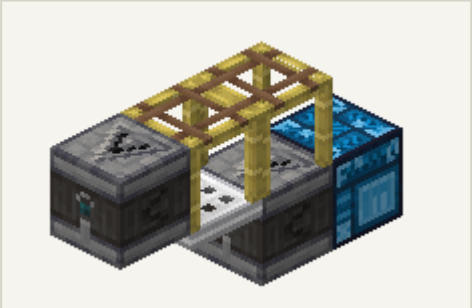


> Tree power detection takes advantage of the fact that tree power has zero delay within TT. The tree farm's growth module **runs within TT**, so trees inevitably grow and place logs within TT. Therefore, leaves immediately execute the check within TT, then emit NC and PP updates, triggering the BUD and achieving growth detection. However, due to space constraints, this approach is almost no longer used.



2.2 Scaffolding Power

When a scaffold receives a PP update, it schedules a tick to check its **horizontal distance** from the supporting block. If the distance changes (and is not greater than 6; greater than 6 will cause it to drop), it emits NC and PP updates. Scaffolding power adds **1gt delay per scaffold**. We generally trigger it through **trapdoor state changes** below the scaffold.



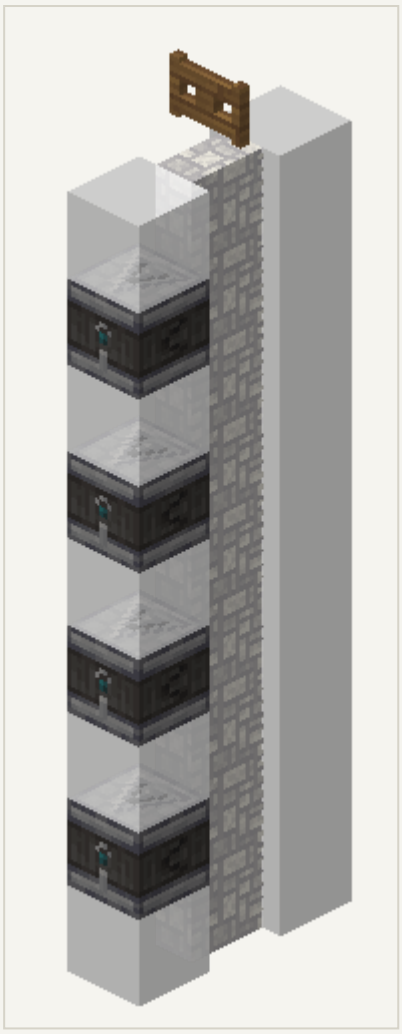
Generally, we use scaffolding power to create **odd-gt delays** ~~it looks similar to tree power, and the use cases are almost the same~~

> Like tree power, scaffolding power runs within TT. If it can confirm the change in distance from the supporting block within TT, theoretically it can achieve zero delay. However, scaffolding power still feels somewhat unpredictable in current practical applications (

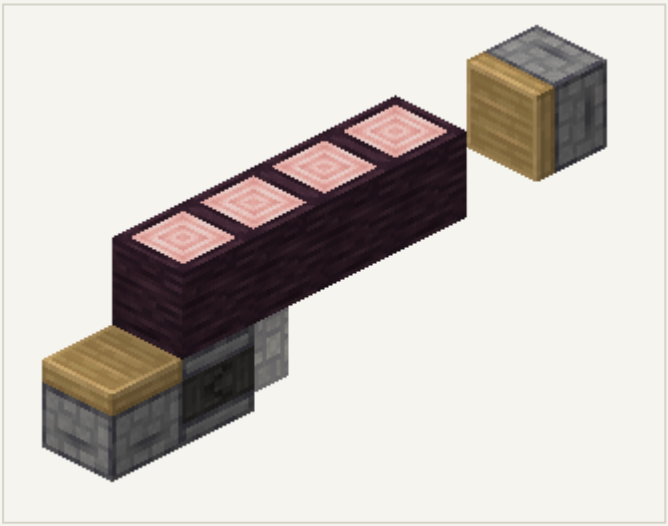
2.3 Wall Power

Starting from 1.16, if there are connecting blocks on both the x and z axes, the wall is low; otherwise it's high. If there's one high wall in an entire column of walls, then starting from that wall, all walls below become high walls (low walls don't override). We generally trigger wall state changes by **opening/closing fence gates above** the topmost wall or **opening/closing doors/trapdoors on the side**.





Additionally, walls actively connect to **certain blocks above them**. Whether a wall is connected is also a wall state, which can be used to redirect block streams upward.



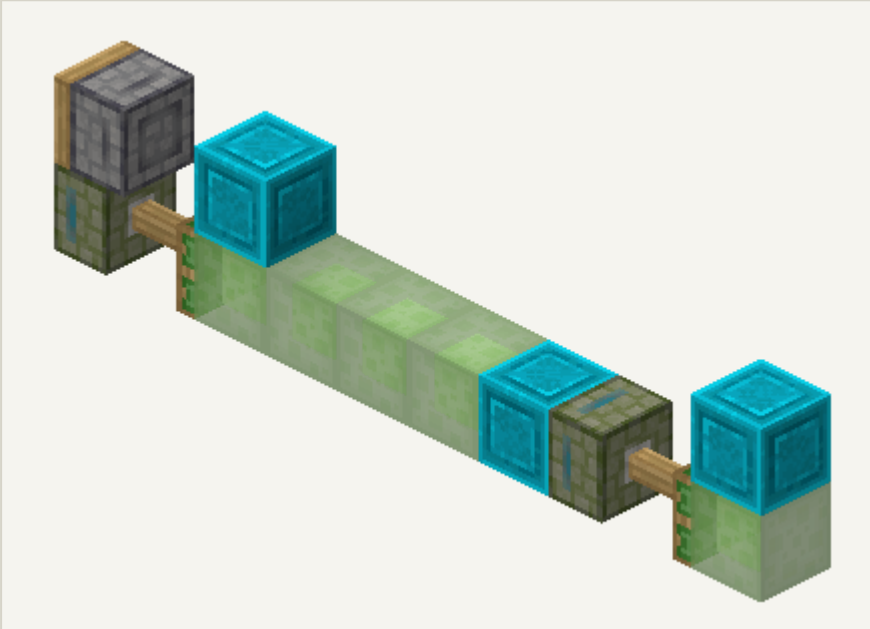
Walls are **instant components**. All wall state changes emit **PP updates** and can be detected by Observers.

> Designs that push the connecting blocks or even the walls themselves also exist, especially for single-edge wall power. For more detailed content, refer to Dongdong's video

3 Signal Transmission Without Macro Delay

The signal transmission methods above all have **macro delay**. If you tried wiring after reading the previous section, you probably found that **we really need signal transmission methods without macro delay**. Here, let's look at the two most important signal transmission methods without macro delay in dustless wiring.

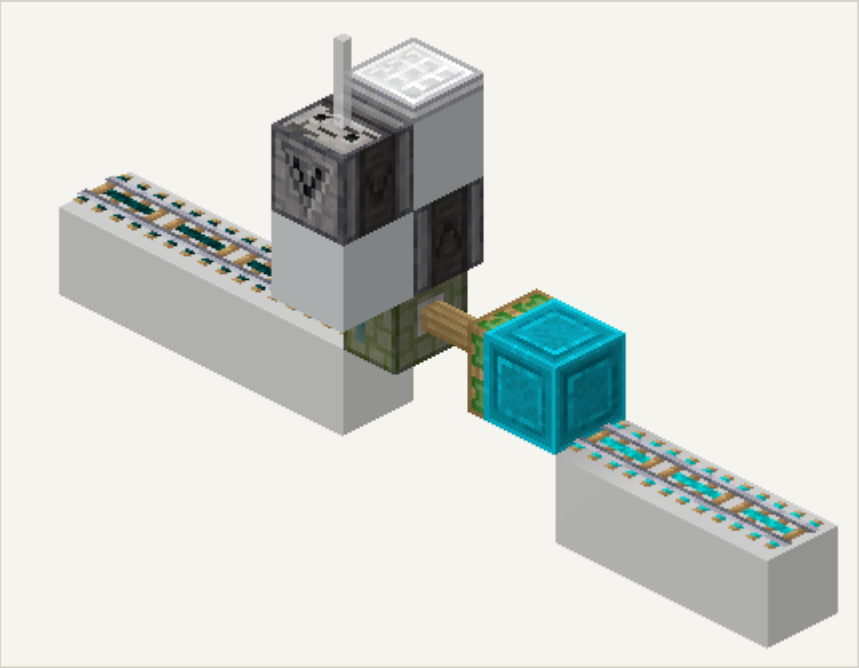
3.1 Slime Sticks



Slime sticks, as the name suggests, use Slime Blocks to pull things and transmit signals. When a Piston pushes or pulls a chain of blocks, **it deletes the blocks from their original positions during the BE phase**, so **it can change another Piston's activation state within the same gt**, eliminating macro delay.

For sticky Pistons in such chains, **falling edge transmission has no macro delay, but rising edges have 3gt delay per Piston**. So what do we do? Generally, we add an auto-reset device to each stick that resets at 3gt, allowing them to reset synchronously and **output rising edge signals in sync**.

3.2 Rails + BUD



Remember that rails are instant components? By using rails to update BUDs, and having the BUD change the activation state of another chain of rails, we can keep the signal propagating within the BE phase.

These structures generally also have zero-delay falling edges (*because in a dustless environment, our only wiring method that can output rising edges with zero delay is relatively complex*) and auto-reset to output rising edges.

> In fact, BUDs only need updates, regardless of whether it's a rising or falling edge signal. The distinction here is just for consistency with the earlier discussion.

4 Redstone Redirection

Since the English name is Redstone Redirection, we don't use the more common translation "转向"

This is the **only method in a dustless environment that can output rising edge signals with zero delay**, as mentioned earlier.

Redstone Dust **only powers in the direction it points**, so if we create a **powered Redstone Dust that doesn't point in a certain direction**, and then when a rising edge signal arrives, **redirect it to point in that direction**, we can output a rising edge signal in that direction with zero delay.



Redstone Dust is redirected by the following blocks:

- Repeaters, Comparators, tripwire hooks *immovable*
- **Redstone blocks, Observers, targets, lightning rods, detector rails** *movable*

Of these, **the movable ones** are the most important for practical redstone redirection.

Note that redstone redirection **only produces PP updates**. To activate certain components, we need to **provide NC updates separately**.

Here are a few components based on redstone redirection that see **heavy use** in tree farms.

4.1 Columns

This is straightforward: arrange several redstone redirections vertically. Naturally, you can arrange the blocks used for redirection in a column and move them up and down.



This gives us an entire column of **synchronized** redstone redirections, making it easy to activate an entire row of Pistons.

4.2 3gt Generator

This is a standard 3gt generator.



`0gt BE`: update the piston driving the upward-pushing redstone block column. The Redstone Dust **redirects and points to the block that needs to be powered**; the rail above receives the NC update from the column push and activates, sending an NC update to the Piston column; the Piston responds to the NC update and extends.

`2gt TE`: the redstone block column is in position, the powered Piston completes extension, **self-checks**, finds it's **not powered**, and schedules retraction.

`3gt BE`: the powered Piston starts retracting, the redstone block column starts pushing down at a slightly deeper position. **Although this powers the target Piston, the target Piston will prioritize completing its retraction**.

`5gt TE`: reset. The target Piston self-checks again, finds it's not powered, and doesn't extend.

This completes a **3gt push, 3gt pull** Piston action, which is especially important in modern honey-slime tree farms.

> Notice that if we don't give the target Piston an NC update at 0gt BE, but instead give it at 3gt BE, we can delay the target Piston's action by 3gt. This is the timing switching scheme used by perfect-timing all-tree-type farms: providing updates at both 0gt BE and 3gt BE. Since the target Piston is the second level of a two-level recursion, pushed into position by the first level, it won't extend from the first powering and will instead be activated by the second.

4.3 0t Generator

This is a simple 0t generator based on redstone redirection.



Basically, after the redstone redirection powers the target block and provides an NC update to make the Piston extend, **remove the powered block**, achieving 0t.

You can design many similar devices, including vertical stacking as mentioned in [5.4.1].

> In fact, since there's only a single Redstone Dust here, directly removing its power source (the redstone block behind it) to create 0t won't cause much lag (roughly equivalent to 3 rails). If you're not aiming for completely dustless designs, this is also a solid wiring method.

Those are the basic dustless wiring methods. Let me say it again: **tearing apart a few farms and wiring a few yourself is worth more than anything else**. Don't skip hands-on practice.

7.6 Everything About 4gt Tree Farms

Everything About 4gt Tree Farms

This chapter requires maximum integration of previous knowledge. For parts mentioned before, we will indicate where they were first introduced. ~~hope you enjoy overbraining~~

Before we begin, we must mention: although detection-based 4gt tree farms do exist, they are quite complex, and **we will briefly discuss them at the end**. All other tree farms here **run on a 4gt clock**.

> Also, since a piston action takes 3gt, you could run the tree farm on a 3gt clock, but obviously this would require using an autoclicker for bonemealing. Traditionally, any tree farm using an autoclicker is not recognized. However, 12gt and 8gt mega spruce have now become exceptions.

1 4gt Birch

No matter what, we still need to start with the simplest: 4gt birch.

Since the 4gt main trunk and base are relatively independent ~~and the base can almost follow a formula~~, we will first discuss the trunk—the core architecture design.

1.1 Core Architecture and Timing Design

Here we discuss two typical and relatively simple architectures:



This is a very ancient architecture. In a sense, the original designer seems untraceable, but it is also one of the excellent examples for analyzing 4gt design.

First, analyze the timing (it is recommended to try designing it yourself first, then look at the timing we provide):

- First cycle: First, piston 1 at 0t pushes the processing log to the center and pushes the log downward; then piston 2 at 0t retracts the log.
- Second cycle: First, piston 2 at 0t pushes the processing log to the center and pushes the log to the right; then piston 1 at 0t retracts the log, **resetting**.

Obviously, each cycle only contains "one piston action" (*all pistons only perform one synchronized action*), requiring only 3gt to complete, so this can clearly serve as a 4gt tree farm architecture.

> To keep the architecture clear, we removed the leaf processing. You can try designing one yourself.



This architecture is not as ancient. It was made by Xinghe, and the timing is slightly more complex.

Timing: (all at `0gt BE`) Piston 1 extends at 0t to push out the upper log; then piston 3 retracts at 0t the lower log, piston 4 extends at 0t to push the log downward; then piston 2 retracts at 0t the log horizontally, **resetting**.

This way we also complete the task of processing logs within "one piston action".

> I did not introduce retraction-based architecture design here: its complexity is completely unnecessary for birch. Later in the 4gt multi-species section, we will solve many, many, many problems related to it.

1.2 4gt Base and Sapling Circulation System

As mentioned in 4.2.1, generally we use bottom suction or side suction to handle the tree roots. For 4gt, **three-shot + side suction** is a more convenient choice.

Since the side suction piston completes the retraction action at 2gt TE, we only need to make the horizontally extending piston extend immediately **and retract before 4gt BE ends**, and we can easily design a 4gt side suction root processing.



> In fact, bottom suction can also be used. Traditionally, we would use two 3gt dirt-returning bottom suction modules on the x-axis and z-axis respectively; later, hampter published a bottom suction architecture that achieves overall 4gt reset through slime block push-pull, as shown below.

The side suction architecture leaves us a very wonderful AFK position: standing below the side suction, crouching (sneaking), aiming at the upper side corner of one of the dispensers on the left or right, we can start planting trees.

The next issue is sapling circulation and bone meal supply. We generally **do bone meal supply first** ~~because there is a formula~~.

By using droppers to throw bone meal upward like this, pulling it horizontally to the periphery of the tree farm, and connecting it to an unpacker, you get a very standard bone meal supply module. Note that **since clock-based 4gt tree farms only have one tree-growing window every 4gt, we do not use alternating bonemealing, but use synchronized bonemealing**.

For sapling circulation, since 4gt brings a large number of items that need to be processed, we generally use **3-4 droppers** to throw saplings to the player. Since the common wiring method for 4gt is dustless (*especially observer 0t, which we will discuss later*), and there is not much space inside for sapling splashing, **we often need to collect saplings that fly outside the tree farm core**. The conventional approach is to build a wall around the core, then **fill it with hoppers**. Sometimes we also need to collect saplings stuck above the core. Generally, we will **flush them down with water** or **collect them with hopper minecarts**.

After these, you only need to do the wiring, and you can get a 4gt birch tree farm. However, obviously, if using redstone dust, this tree farm will become very laggy. Therefore, we need to understand the dustless wiring method for 4gt tree farms.

2 Dustless 4gt Wiring—Dustless 0t Generators

For the overall approach to dustless wiring, it can be roughly said:

- First, we must understand that not everything must run at different depths. For things that "must run at different depths," we actually have **different operational spaces** for **rising edges** and **falling edges**.
- Second, we must understand that **4gt tree farms operate on clocks**. We do not need to strictly design the wiring for starting the clock according to the running timing. **As long as the correct timing can be maintained after everything starts**, the rest is just a matter of turning on/off and resetting. These two things are especially important for dustless wiring. They can help us greatly **save BED devices**, thereby reducing lag.

After understanding these two things, you will understand that **tree farms designed based on dustless 4gt wiring must be modular**. The larger the tree farm, the more this is true.

> For wiring based on redstone dust, the previously mentioned "stringing them one by one in depth order" method is still the simplest, most straightforward, and useful. This is different from the dustless wiring situation.

However, since 4gt multi-species is driven by a clock, **connecting the various parts together** is actually very simple. What we really need to understand is **vertically stackable dustless 0t generators**. With them, we can truly make our dustless 4gt tree farm run according to the timing we want.

2.1 Observer-Related 0t Generators

This is an observer-based, 8gt cycle "single-edge" 0t generator (*i.e., it only produces 0t when pushing down (generally called rising edge here[1]). For the principle, please refer to the timing theory section of GTMC*). We directly connect it to an 8gt clock here, and all subsequent analysis is conducted under 8gt clock drive.

Anyway, since 4gt tree farms run on a clock, we can give **the piston activated by the clock to push down a certain depth**. Theoretically, after pushing the observer upward, if the observer can provide power and the time before removing the observer is enough for the target piston to start pushing, the upward push 0t is completed.

The next question is, obviously the observer will not activate after being pushed up, so what to do? Since the observer can only produce 0t on one edge, we let the observer **be activated during the first downward push**. This way it will only be activated after being pushed up, **and produce 0t when pushed down**.



▶ This figure is animated. View original

> In fact, we can attach a row of blocks to the observer face and move them together with the observer, but that thing is very laggy. I do not want any of you to abuse it. It is only superior to the above design when you need to fill the surrounding space to facilitate sapling collection.

Since the 0t signal produced by this 0t generator has a **rising edge at TT**, we **can only control its falling edge signal**. The method is very simple: just chain a bunch of pistons that update each other.

> Actually, we can do some more optimized processing combined with the wiring of the tree farm itself. Anyway, just give the upward pushing piston a deeper NC update.

> In fact, the rising edge depth is not uncontrollable. We only need to push in a powered block at 0t during the BE phase, and we control the rising edge depth of the power. But this method has a serious problem: it is too large. Many times, if the architecture design does not happen to allow inserting this structure in the middle, we will use redstone dust redirection.

Another method is to directly use sticky pistons (or slime blocks) to push-pull the observer. You only need to make the NC update to the sticky piston have a certain depth to make this 0t work. It is generally only used for rising edge 0t (double-edge is possible but generally unnecessary, you can try it yourself).

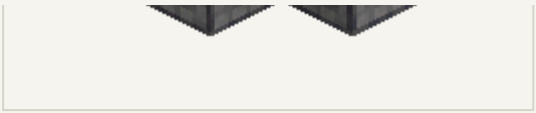
2.2 Redstone Dust Redirection-Related 0t Generators

As shown









Since the piston next to the piston that pushes down the redirection pillar will emit an NC update when pushing down, **simultaneously updating the note block and the piston that pushes down the note block**, as everyone knows, **note blocks do not add depth**, so the note block will **update all target pistons to make them start extending** before the downward push begins, thus completing 0t.

Redstone dust redirection is **the most convenient 0t method for controlling the rising edge signal depth of 0t** in our dustless 4gt wiring.

> In fact, not everywhere you must control depth. Controlling the activation order of pistons in BE through different activation orders of observers in TT is also possible.

> The biggest problem with redstone dust redirection 0t generators is that you need to trap minecarts, otherwise a large number of saplings will accumulate in the middle, so please use it carefully (but actually it is still a bit more convenient than the observer 0t that pushes and pulls powered blocks).

2.3 Wall Power-Based 0t Generators

As shown



After activating the observer through wall power, use a piston to remove the powered block. A very simple design.

> Actually, it seems not used very much due to volume issues.

At this point, you only need to do the wiring assembly yourself, and you can complete this dustless 4gt birch. Here we recommend Scorpion's design as a reference

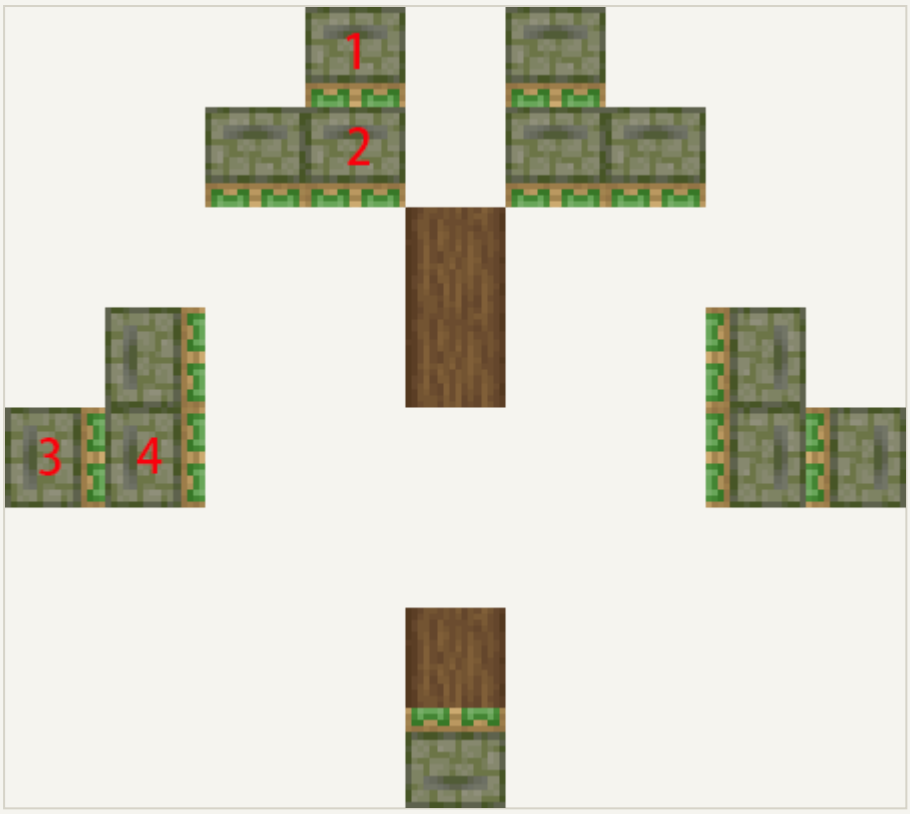
> Actually, if you look carefully at Scorpion's wiring, you will find: we do not necessarily have to use 0t. The 2gt signal given by the observer is also usable. This is also a quite important technique for reducing lag in 4gt tree farms. Folen's 4gt multi-species still suffers from lag compared to things made with Kay and Land's laggier architecture because it uses 0t everywhere.

3 4gt Multi-Species

Since the content of 4gt multi-species is quite extensive, we will go through it bit by bit.

3.1 Branch Trees—Basic Timing for Acacia/Azalea/Cherry Blossom

In 4.3.2 we mentioned that we only need to do the diagonally retracting trunk processing designed in TT again on the other side, and we can easily get a 4gt architecture. Here, due to the growth detection requirements of multi-species, **we cannot use the birch architecture**. This pseudo-double-recursion architecture first advocated by Shixiong becomes our only choice.



Here we temporarily use Shixiong's original architecture as an example, but temporarily ignore some additional parts used to process leaves.

If you still remember the content of 3.1.2, you should be able to see that this architecture can handle branch trees. The question is the timing design.

Currently ~~except for plexi's alien architecture~~ all 4gt multi-species use the branch tree processing timing from Fanhua Qianmu. Here is the timing he designed:

Piston 1 extends at 0t to push out piston 2, piston 2 retracts at 0t the logs on the side of the trunk; after that, piston 3 extends at 0t, piston 4 extends at 0t (if there is a log in front of 4 before 3 extends, then 4 will not extend).

This successfully **first transfers part of the branches out of the processing position**, then performs centering and conventional processing. As for the remaining piston at the bottom, just give it a 0t ~~it won't break even if it can't extend~~.

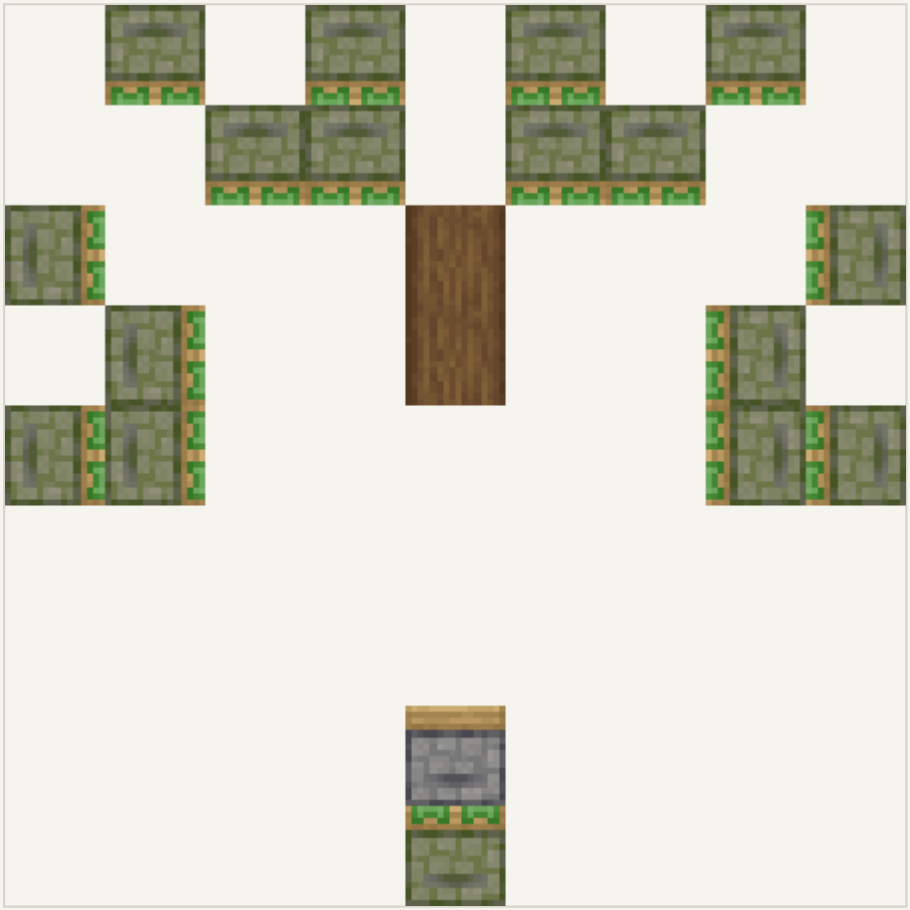
> Actually, if you remember, you will find: this is the timing TT originally designed for 6gt jungle. But the earliest 4gt jungle tree farm did not use this timing. We will mention the reason soon.

You should be able to imagine that this timing can even handle the situation where the core is completely filled with logs along the x and z axes.

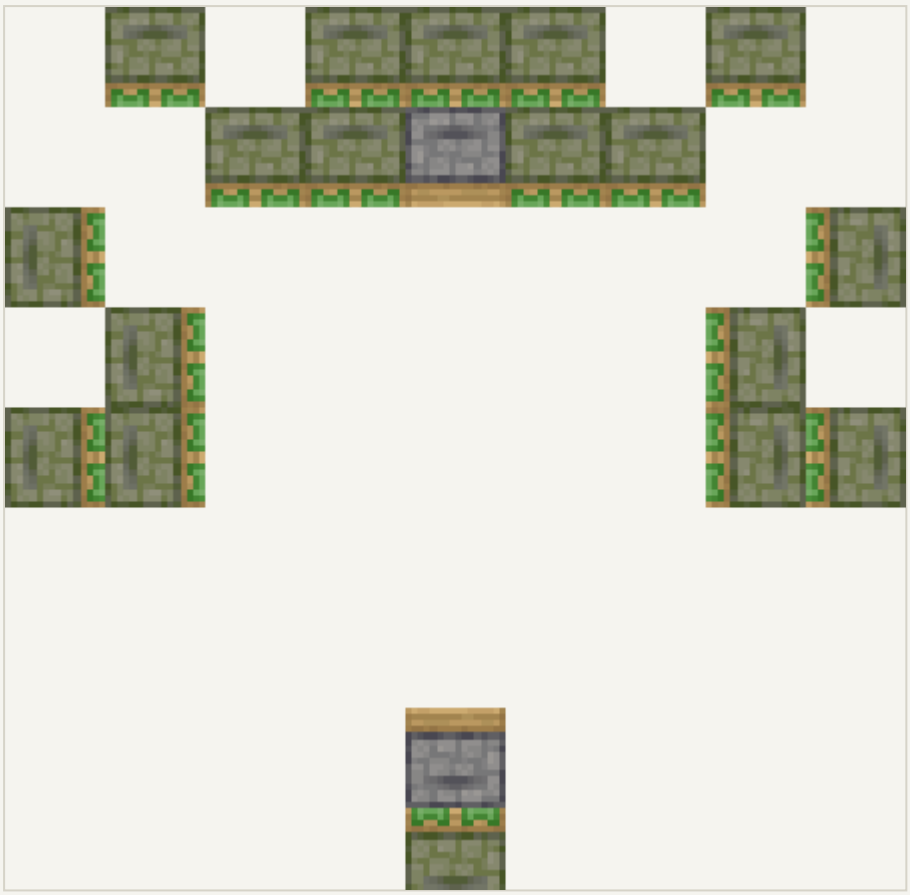
> To increase branch processing capacity, we can actually replace the poor piston at the bottom with double recursion (you can think about other architecture designs yourself. The standard answer is plexi's design).

> Of course, we have more extreme processing: insert a double recursion every other block in the entire row of logs used to activate the side pseudo-double-recursion, and make the entire side pseudo-double-recursion synchronize at an 8gt cycle, only letting the pseudo-double-recursion that directly extracts the trunk run alternately (also plexi's design).

Layer1:



Layer2:



3.2 [Special Case of Plexi Architecture] Special Handling for Azalea and Cherry Blossom

Actually, if you use Shixiong's original architecture, you theoretically do not need to worry about this. But if you replace the sticky piston pushing logs at the bottom with double recursion, things become much more complicated.

Suppose due to the special growth mechanism of azalea and cherry blossom, we get three branches between the upper suction wall secondary activation logs and the lower double recursion. At this time, if we do not add delay to the rising edge 0t of the double recursion, **the situation will occur where the trunk has not been sucked away and the double recursion cannot push**. After 3gt, the double recursion is pushed out, and you will lose efficiency. For azalea, this will be even worse. **Azalea's growth detection is 3x3**, so after the secondary piston of the double recursion is thrown out, **azalea will not stop growing trees, but will generate logs between the primary and secondary**, then send your secondary piston away.

After plexi inserts double recursion in the middle of the suction wall activation logs, we need to process that side as well. But obviously, **there is no place for you to put rising edge control there**. What to do?

> In fact, the main reason there is no way to control the rising edge is that the log diagonally above the dpe secondary is powered by the observer 1gt before extending. If we do not want it to be powered at this time, we need to make it start retracting after 4gt, which will conflict with the double recursion on the opposite side. Either give up this activation, or it will become the situation of sending away the piston mentioned above. So if we control the rising edge of the primary extension, we cannot use the same secondary falling edge signal as the suction walls on both sides that have no rising edge. Obviously there is no place here to put another different activation. At the same time, if we control the rising edge of the primary (in fact, as long as the signal provided starts later than the observer, it will be finished), after the tree grows, it will prevent leaves and NC update the secondary piston, making it extend early. Your primary cannot push, which is not an ideal situation in any case.

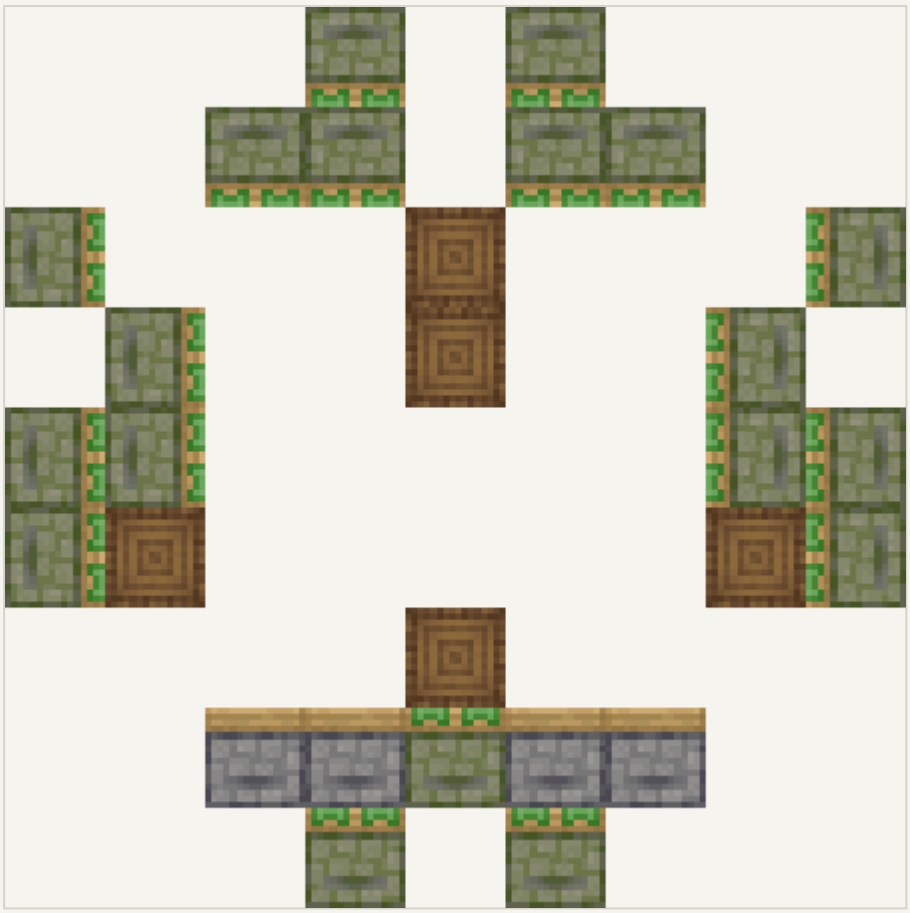
Plexi came up with a good solution: **if it cannot push, just pull the piston away directly**, thereby removing the push at 3gt, thus avoiding throwing out the regular piston. Theoretically there should be quite a few implementation methods. You can try more yourself, ~~although Xinghe and plexi, qontrol, tintex argued for a long time and only the wiring architecture plexi had done before could work~~.

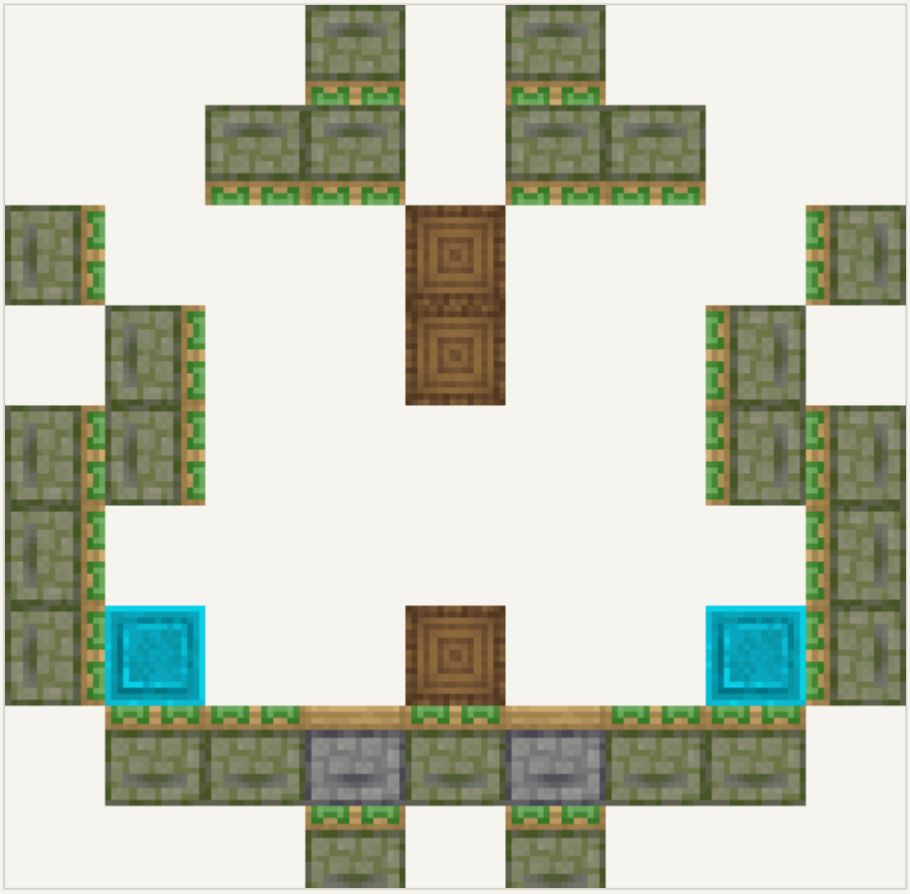
3.3 Jungle Wood

Welcome to the actual biggest boss in 4gt tree farms. Due to jungle's 1/40 sapling drop rate, 4gt tree farms actually only run half of each cycle. Jungle sapling circulation has become the biggest obstacle for 4gt multi-species ~~I'm sure you still remember Luoxi's extreme 100.5% recovery rate back then~~.

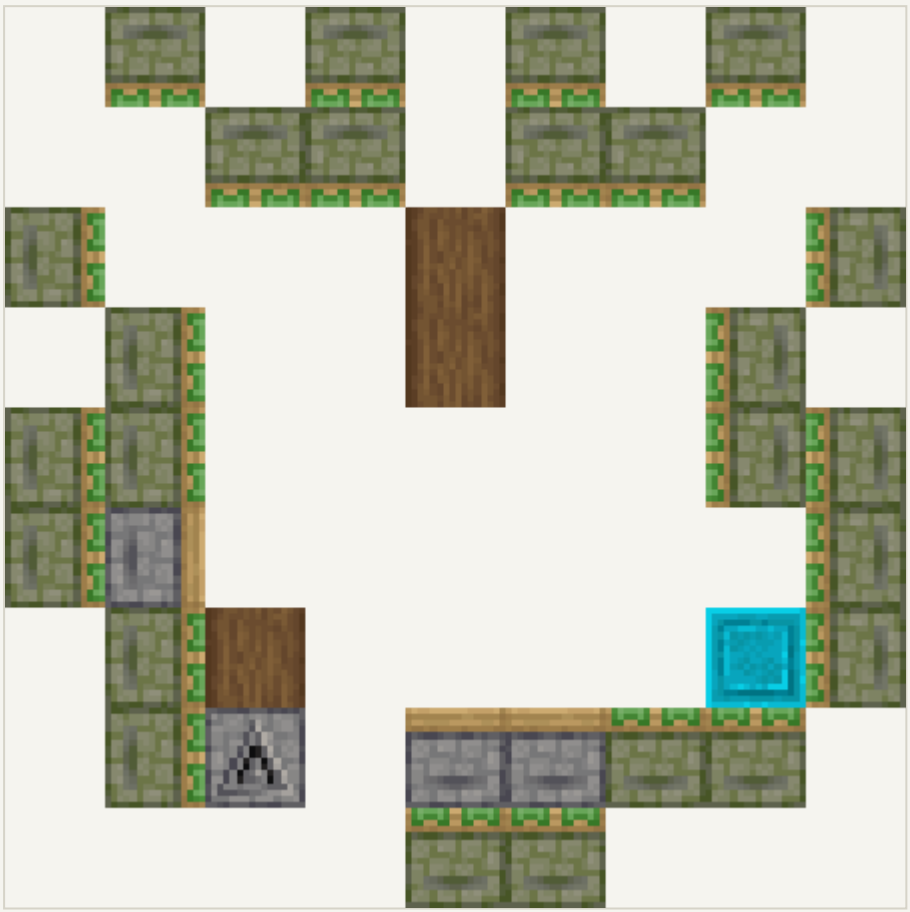
We mentioned above that in 4gt multi-species, we abandon TT's original timing for jungle, and **synchronize the pseudo-double-recursion used to extract the trunk on one side with the pseudo-double-recursion on the opposite side**, so that more leaves can be processed in the same cycle, thereby obtaining more saplings.

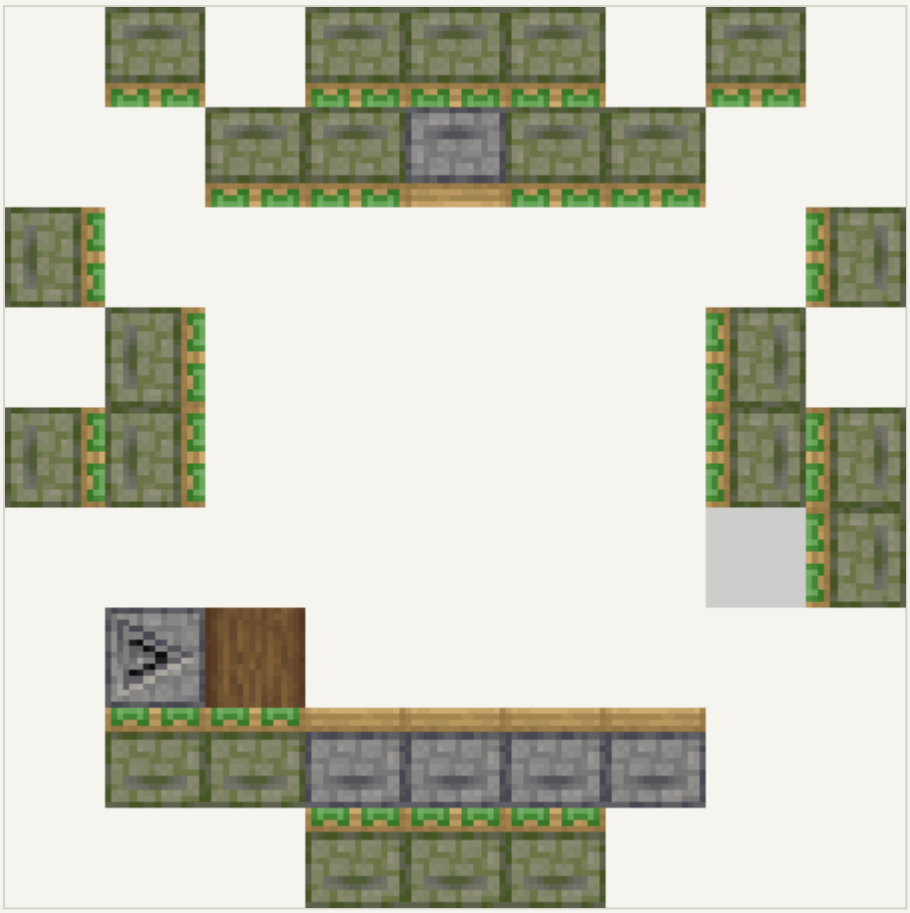
Can we process a bit more leaves? Let's complete Shixiong's original architecture first.





It seems plexi increased branch processing while also processing more positions with higher probability of growing leaves. Let's complete it and see.





Obviously, plexi's architecture processes quite a bit more leaves, which greatly alleviates the pressure on the sapling circulation system ~~but this does not mean you can casually pull hopper chains, you still need to do some design~~.

> Considering that jungle sapling drop rate is small, this will cause a large variance in the number of saplings dropped by each tree, which is very unstable. We need some method to cache excess saplings and re-throw them to the player when saplings are insufficient, called active sapling circulation. However, due to the large leaf processing capacity of modern 4gt tree farms, we no longer need its assistance to get a stable and sufficient jungle sapling supply.

In 4.2.2 we mentioned that high-speed tree farms often have a large number of drops to process. Obviously, the leaf processing capacity added for jungle is useless for other tree species; on the contrary, to reduce the pressure on hopper chains, **we need to reduce leaf processing capacity**. Therefore, generally we will make all other tree species run under the branch tree timing.

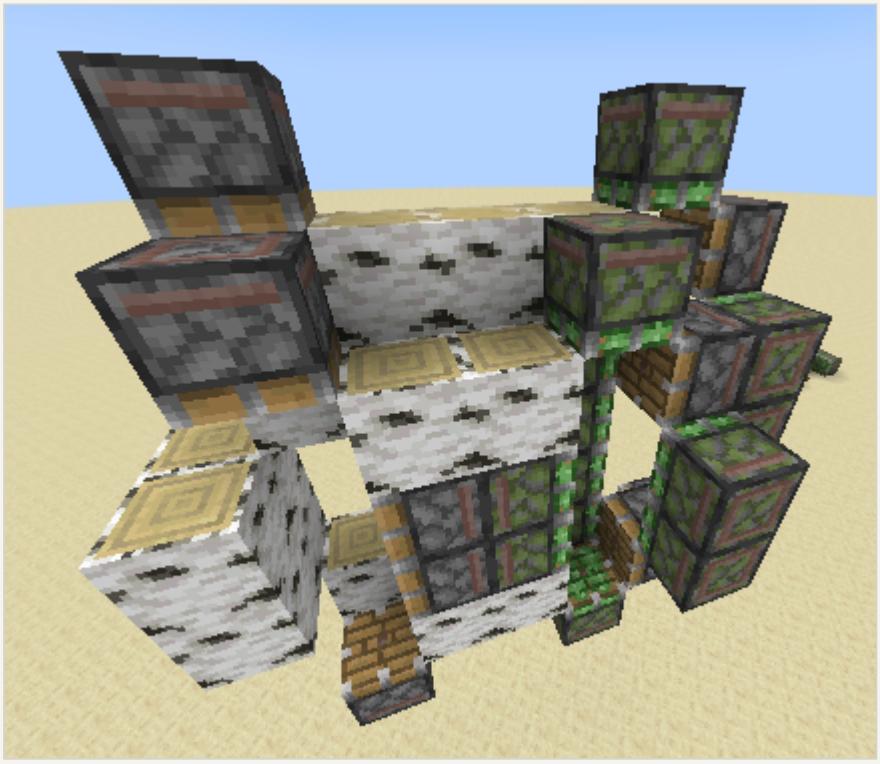
> In fact, you can also imitate perfect timing tree farms and shut down part of the structure, but since modern 4gt architecture has strong integrity, ~~shutting down is almost the same as not shutting down~~, so it's just made lighter.

3.4 Log Output—Suction-to-Push

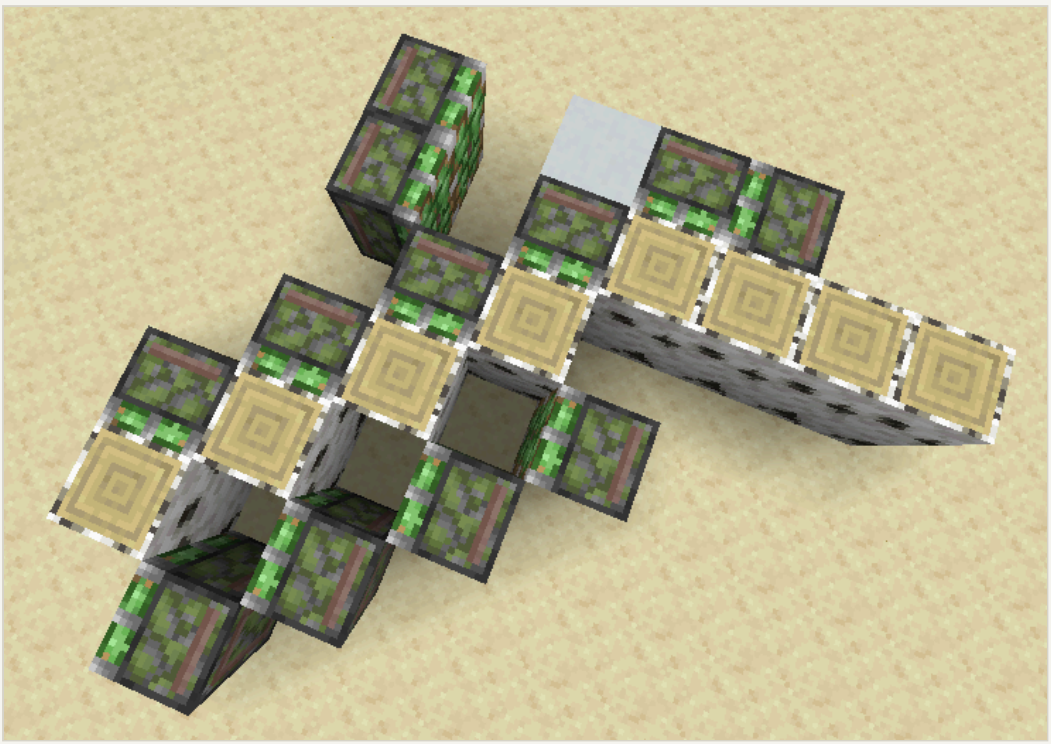
If you look carefully at the current tree farm, you will find a very serious problem: **every step of the block stream is pulled by an entire row of sticky pistons**. Such a block stream obviously cannot be processed. So here we need to fill a hole: log output for pure retraction-based pseudo-double-recursion architecture.

> As for why not in #4, it is because Xinghe believes that for modern high-speed tree farms, except for 4gt-related and 8gt mega spruce, this part can be avoided through architecture design. If you say that for certain architectures this output method can increase branch processing capacity, then you should know that there is something called PUTF+ in the world, which uses 12gt timing and achieves cherry blossom efficiency exceeding 6gt multi-species. So this processing method is still only necessary for 4gt-related.

The first method, we can notice that the top and bottom of the log stream are empty. We can **pull out logs from above, then push them out horizontally**, thus completing the suction-to-push for the top and bottom two logs. By operating on the log stream in this cycle, we can get a stepped block stream. **This block stream is especially suitable for making wither processing**, because we only need to push the top and bottom two groups vertically together and then horizontally insert them into the wither cage.



The second method, we can add a double recursion module like this at the end of the block stream, thereby **moving the block stream back two blocks at once**, and pushing the block stream out along the way.



Actually, if you think carefully, you will find that the essence of needing these is because **we have no place to put the pistons for pushing out**, so we find a place to put down the pistons for pushing out, then find a way to move the logs over. A common approach is to use honey-slime. **They can stick logs to their sides**, thus providing extra space to place a row of modules similar to side suction, thereby completing suction-to-push.



Obviously, honey-slime streams can reduce 0t and lower lag, while shortening the length of the block stream. It is our top priority now.

3.5 Wiring—Integrated/Independent/Modular?

Although we said before that 4gt is driven by a clock, so as long as the timing is correct when running, **we still need to consider the stability and performance overhead of the design**.

For stability considerations, using the same clock and connecting the timing lines that run with the tree farm after the clock is definitely more reliable. For example, the honey-slime block stream mentioned above, **if the timing of the honey-slime wall and other block stream pistons is wrong, those pistons may be stuck by the honey-slime wall and destroy the tree farm structure**. Therefore, for example, in plexi's design, the timing of some block stream pistons is derived from the clock that activates the honey-slime wall. This is **integration**.

However, if all parts are forcibly connected together, this will obviously cause a lot of lag. Therefore, **we also need to connect parts beyond a certain range to another clock**, thereby reducing the performance overhead caused by signal transmission. This is **independence**.

> In fact, we have very, very concise 4gt clock designs, which means that throwing away vertical signal transmission and directly adding clocks can sometimes reduce lag. You can see this design in plexi's honey-slime block stream output section.

Pistons in an area connecting to the same clock will lead to the **modularization** tendency we mentioned before. We can design the wiring of each part separately, and finally assemble them like building blocks. Taking the two 4gt multi-species architecture diagrams shown above as examples, modularization is especially obvious in the upper pseudo-double-recursion wall, **because it does not need to connect with the remaining parts, and as long as the wiring is behind it, it will not cause any additional space occupation and affect assembly**.

At this point, you only need to simply design the timing wiring and connect everything together, and you are done. Obviously, reading the textbook for the first time alone cannot fully understand the timing and wiring methods of 4gt multi-species. I recommend again: **go copy someone else's wiring**. Here I especially recommend plexi's 4gt multi-species. As long as you are a bit serious about dismantling the wiring, based on our previous knowledge, you can definitely fully understand how it works. The only regret is that the release page is on Tree Hungers Discord. You need to use a VPN, and you cannot open the release page link at all before joining this server, so you can only search for it yourself (.

Finally, we only have one somewhat niche thing left to discuss: detection-based 4gt tree farms.

4 Detection-Based 4gt Tree Farms

Actually, speaking of it, the biggest problem with detection-based 4gt tree farms is: **the only dustless method we can use that can provide zero-delay rising edge 0t** is **redstone dust redirection**. If you try to dismantle the wiring of a few 4gt multi-species or build one yourself, you will find: **it really takes up so**

much space.

So now the biggest problem is how to fit all this redstone dust redirection in. But about this I can only say: **architecture is more important than anything. If the architecture is not excellent enough, you have no possibility of fitting it in** (of course you can also hang part of it on the clock ~~but I think such a "detection-based tree farm" has no practical significance~~).

> If you are not afraid of death, you can of course try a redstone dust-based detection-based 4gt multi-species. ~~Xinghe's computer cannot run it anyway~~.

Another problem is that we need some method to make our detection unit **able to run a 4gt cycle**. The simplest and most straightforward solution is of course **to make the output signal structure reset within 4gt**. Another less elegant method is **to make another set, each set resets within 8gt**.

For this kind of detection-based dustless tree farm with extremely compact timing, you can see Qontrol's dustless detection-based 4gt birch and dustless detection-based 6gt multi-species PUTF# (there is no way, there is no completed dustless detection-based 4gt multi-species yet, can only use 6gt to make up the number).

> However, if you still remember, we said before that 4gt tree farms can also use bottom suction. Obviously, since bottom suction moves dirt, if the sapling has not grown, it will be destroyed, so we cannot connect bottom suction to the clock. At this time, you still need to use some ideas from dustless detection-based 4gt tree farms. But another question is, do you really need to add bottom suction to a 4gt tree farm? ~~Actually you still need to, we will mention this problem again in the dual-core section later~~

Footnotes

1. The rising edge concept here is different from what was mentioned before. Here it is clarified: the original meaning of rising edge is the "edge" where the input signal goes from nothing to something (0 to 1) (conversely, falling edge is from something to nothing (1 to 0)) ↩

7.7 Large Spruce Tree Farms

> Welcome to the definitive guide on mid-to-late game wood production. Here we'll walk you through designing the most log-efficient tree farm in Minecraft.

1 Introduction to Large Spruce Tree Farms

1.1 Large Spruce Growth Mechanics and Architecture Design

The growth detection range for large spruce is: **3x3 at the sapling layer (centered on the northwest corner sapling), expanding to 5x5 above**

The maximum height of a large spruce is 28 blocks (the northwest corner sapling gets one extra, for 29 total) ~~generally speaking, no one builds a large spruce tree farm without handling full height~~

Put simply, we need double recursion to handle the trunk logs, then push the leaves away with honey-slime walls.

1.2 Vertical Signal Transmission

Use stick BUD to pull it up. For trunk and side wall activation, consider redirecting dust or wall power.

1.3 Processing

After pushing out from the core, split left and right, then reassemble as usual.

At this point, you can already design a functional large spruce tree farm on your own. But if everything works as expected, the efficiency will only be around 100k, which is underwhelming. Next, we'll cover how to make a faster large spruce tree farm. For processing timing, I trust you've built up enough understanding from previous lessons, so here we'll only discuss how to squeeze out as much pre-bonemealing as possible.

> Of course, I also hope you'll get to experience the exploration journey of the past, so I'll include a lot of history about large spruce tree farms, along with some old version content. Hope you enjoy it.

2 High-Speed Large Spruce Part 1: Corner Down-Suction

2.1 The Origin of Corner Down-Suction

In 2016, Laoxian released his 12gt large spruce tree farm, using an upward-pushing architecture, breaking the 300k barrier for the first time. However, as time went on, Laoxian's timing no longer worked in 1.12, and players gradually discovered better architectures and timing designs.

On February 12, 2020, gpw released sprucemacy v1, using observer-powered tree root detection and corner down-suction, ushering in the modern high-speed large spruce era. It became the design standard for large spruce for quite some time, until April 15, 2021, when gpw released sprucemacy v2, bringing corner side-suction, which we'll cover a bit later. Now, let's go back to 2020.

2.2 The Concept of Corner Down-Suction

If you tried building a large spruce tree farm in 8.1, you probably noticed that adding dispensers is quite difficult. Without modifying the base architecture, even with push limit detection, we can only fit two dispensers.

Since we're still in 2020, we don't have push limit detection or tree power detection yet, so let's ignore them for now. We need to choose one of the four positions where dispensers can be placed for the powered tree root detection. Obviously, if we want to increase to three dispensers, we inevitably face a problem: no matter what, we cannot process the corner log through side-suction.

The solution is straightforward: down-suction it. But because dirt returns slowly with down-suction, we need a separate planting timing.

gpw's solution was to have the player plant the corner sapling at 8/10gt (in 1.12, player right-click placement runs on a simple 4gt interval), with the rest planted normally.

For player movement, gpw used minecarts.

2.3 Limitations of Corner Down-Suction

In 1.14, Mojang modified the player right-click placement behavior. If the player accidentally places the sapling on the closer side first, the sapling farther away cannot be placed. Currently, without modifying the planting timing, the only solution is llama boats ~~(but in 1.21, because Mojang changed mob behavior on boats again, llama boats also broke)~~

Corner down-suction in 1.12, even with 3gt dirt return, cannot guarantee planting the corner sapling before 4gt (due to the right-click placement logic mentioned above), so it cannot reach peak efficiency.

> However, I can spoil this in advance: in large spruce using auto-clickers, corner down-suction has made a comeback thanks to simpler timing and wiring design (lol

3 High-Speed Large Spruce Part 2: Corner Side-Suction (Corner Retraction)

3.1 The Origin of Corner Side-Suction

On April 15, 2021, gpw released sprucemacy v2, bringing us corner side-suction for the first time. On February 8, 2022, floppy's 12gt Spruce v2 introduced corner side-suction timing that completes bone-mealed sapling planting in 3gt. In August 2024, Qontrol developed a practically viable corner side-suction base with 6gt reset.

> Actually, not long after 12gt corner side-suction came out, Xinghe casually made an all-dust 6gt version, ~~but it didn't look usable at all~~

~~Because Mojang changed llama boats in 1.21, corner side-suction lost much of its significance~~

3.2 The Concept of Corner Side-Suction

If you want to plant the corner sapling first, you must side-suction it. The side of the corner trunk has two dispensers and two logs. If we can somehow remove one log from the side, we can fit in a side-suction piston. This is corner side-suction.

The current implementation is to down-suction one of the logs, while the other side is handled with conventional parallel side-suction.

For corner side-suction, we obviously need double recursion. For 16gt non-auto-clicker large spruce, just use normal double recursion; for 12gt, we use double recursion with secondary pistons pushing in from the side; for 6gt, we use a design similar to the "standard flow" of retract-rotate-push.

Generally, player movement uses llama boats with corner-priority planting timing; or pig boats, where you plant one on the parallel side-suction side first, then the corner, then the second on the parallel side-suction side, and finally the down-suction position.

> We'll describe the specific origin of pig boats in the next section.

3.3 Too Difficult to Build

If you still remember the 8gt large spruce, you'll know that corner side-suction didn't appear in that tree farm. The reason is actually very simple: neither lintex nor I could think of a good enough wiring method to handle 8gt corner side-suction. So we turned to corner down-suction and changed the planting timing (after all, no one would use a real person without auto-clicker for 8gt large spruce)

> Originally when we started the project, hampter said he would make a corner side-suction for 8gt large spruce, but his computer broke midway, ~~and his unit still hasn't been released to this day~~

This is the biggest drawback of corner side-suction: complexity.

~~*Of course there's an even bigger drawback: llama boats broke so you can only use pig boats and pig boat planting timing, which simply can't compete with corner down-suction*~~

4 High-Speed Large Spruce Part 3: The Return of Corner Down-Suction

/ Pig Boat Planting Timing

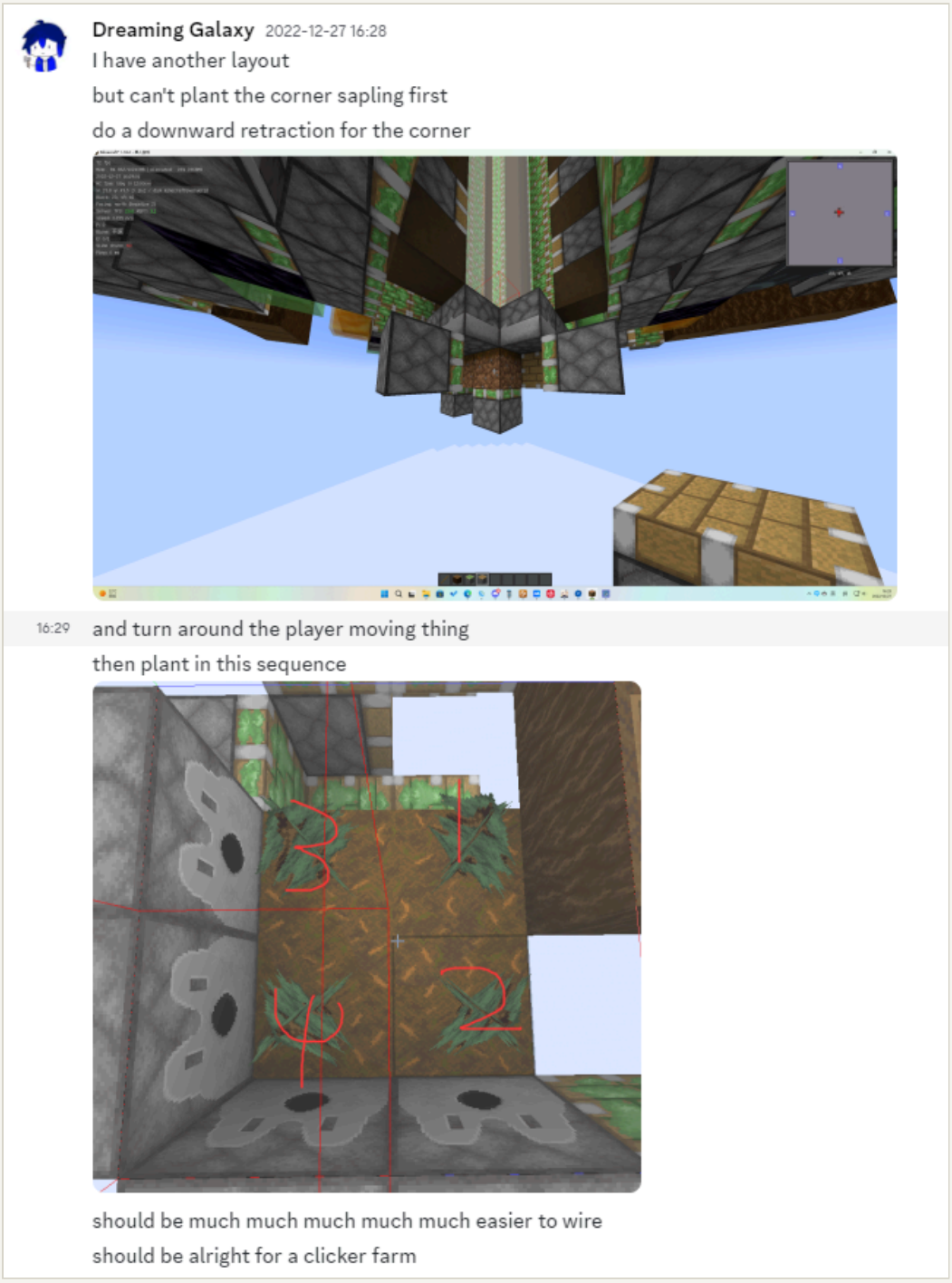
4.1 "No Matter What You Do, It Sticks to the Main Suction Piston"



~~In hindsight it wouldn't have stuck, but ending up with a simpler base was still a win~~

At the end of 2022, lintex started the 8gt large spruce and quickly got most of the trunk done, ~~then we got stuck on the base~~

From our perspective at the time, we couldn't make a sufficiently good corner side-suction base, so I proposed using down-suction for the corner and reversing the planting timing, allowing us to drop corner side-suction and its corresponding planting timing entirely.



This is the auto-clicker planting timing for corner down-suction.

From this, you should be able to design an auto-clicker base for corner down-suction on your own. If you're still stuck, you can refer to the 8gt large spruce or gushen's large spruce.

4.2 Why Pig Boats?

Actually, I don't really like minecarts. The reason is simple: give them a little push and they roll away, then you have to replace them.

Llama boats broke (sigh)

> For single boats, they're calculated separately on the client and server, which easily causes desync and breaks everything.
>
> At this point, you should be able to design a sufficiently fast large spruce tree farm on your own. However, we'd like to mention one more piece of god-tier tech from 1.12: ITT/IF 6gt large spruce.

5 ITT/IF 6gt Large Spruce

CHAPTER 8

8

绿萌科技

IN THIS CHAPTER

Molforte

8.1 Preparation: Environment Setup and Fundamentals

8.2 Appendix 0a

Twisuki

8.3 Introduction

8.4 Prerequisites and Setup

8.5 Component Basics

8.6 Flying Machine Basics

8.7 Engines

8.8 Mobility

8.9 Semi-Dynamic Engines and Extensions

8.10 Linkages

Molforte

Overview

1 Goals

This chapter aims to give readers the ability to design Slime Tech machines and, at the very least, complete an introductory pass through Slime Tech so they can analyze simple Slime Tech machines.

At present, this text is based only on Molforte's own limited perspective. **It does not represent the views or design philosophy of the entire Slime Tech community**, so I hope readers will approach it critically.

2 What Is Slime Tech?

As for the explanation of Slime Tech, the Wiki already has an answer:

> Flying technology based on slime blocks. Broadly, it refers to redstone technology centered on slime blocks; narrowly, it specifically refers to slime-block piston-worm technology.

On that basis, my explanation is:

> Slime Tech originally refers to a green, cute block (the slime block) or creature (the slime). By extension, it refers to redstone technology centered on slime blocks, including the slime-block portions of various machines and flying machines. In the narrow sense, it refers only to flying-machine technology. Note that we generally also include honey-block-related technology under Slime Tech.

Slime Tech and mobile machinery (mostly movable battleships, aircraft, and similar builds) are essentially the same kind of technology, but this chapter currently focuses only on survival-oriented flying-machine technology. Readers can transfer the ideas on their own, or I may look for a suitable mobile-machinery expert later.

Slime Tech is closely related to Mechanical Redstone. Slime Tech can be viewed as movable Mechanical Redstone technology, but the implementation approaches differ greatly.

3 Prerequisites

To keep you from getting lost while reading this text, before formally reading it we assume that you are familiar with the basic usage of vanilla Minecraft redstone components, and that you already know how to use foundational technical Minecraft mods such as Tweakeroo, Carpet, and Litematica.

4 Preface

This is a preface.

But it was written as an overview instead. Since it is just a demo, feel free to take it wherever it is useful.

I am Jimu.

This tutorial aims to explain how to gain the basic ability to use flying machines when designing flying wiring for technical Minecraft machines and when playing survival.

The tutorial assumes that you already understand the basic usage of components and have some foundation in microtiming.

A flying machine is a special kind of Mechanical Redstone.

A flying machine is a structure in vanilla Minecraft that can move itself several blocks as a whole.

That said, as long as it can move, it counts. ~~How is a two-way piston worm with an external signal source not a flying machine?~~

~~A TNT cannon might not be unable to count as a flying machine either.~~

I believe that **the essence of a flying machine is a movable way to route signals; it is not fundamentally different from fixed signal routing**.

Whether to use a flying machine, and to what extent, depends on the specific function you want to implement.

Technical Minecraft is a playstyle that requires circuits to be physically built. When the function is the same, the less manpower and material cost required, the better. If it is worth it, you can even trade some running speed for that. For example, using a world eater for clearing is obviously cheaper than an entire dark mass of TNT dupers. Because of this, flying machines still hold a significant place in today's environment of large technical projects.

For that reason, flying-machine design in technical Minecraft has slightly different priorities from other branches of Slime Tech.

Common uses of flying machines in technical Minecraft include large-span engineering machinery, such as three-way machines and world eaters; short- and medium-distance signal transmission and Mechanical Redstone applications; long-distance transport; disposable vehicles; and the extraction of non-renewable materials.

Simply put, the principle of a flying machine is **stepping on your own foot to rise into the sky**. After all, Minecraft blocks have neither gravity nor conservation of momentum.

In a flying machine, the structure that implements this principle can be called the **engine**. Parts outside it that exist for function or for added piston load and do not participate in the "stepping" can be called **extensions**.

Of course, you can also divide the structure according to the machine's actual purpose ~~and give each part an extremely impressive name~~. For example, in a tunnel-bore group, you might identify the drill head, connecting rods, timers, and so on. It does not really matter how you divide things, and overlap is fine too. As long as the terms clearly refer to something, nobody cares. However, to highlight the explanation of flying-machine principles in this tutorial, unless otherwise noted I will consistently use the engine-plus-extension model.

According to the number of directions in which the engine can fly, flying machines are classified as N-way flying machines. For example, a machine that goes and never returns is one-way; one that shuttles back and forth is two-way; adding a side pusher to a two-way machine makes it three-way. Under this classification, a world eater is also a three-way flying machine. On this point, technical-Minecraft flying machines differ somewhat from other branches of Slime Tech: for survival technical builds, a flying machine only needs to reach the destination and accomplish the goal, so reserved stations are acceptable.

Flying machines also have stations where they dock. A two-way flying machine is usually divided by its stopped positions into a departure station and a return station, though it can also have neither, or have more stations added. ~~You can also build it in place and let it complete its mission in spectacular explosions.~~

That is all.

8.1 Preparation: Environment Setup and Fundamentals

1 Overview

This article walks you through setting up your slime tech environment step by step, along with a brief introduction to the fundamentals.

The mods covered here are: Carpet mod, WorldEdit, Tweakeroo, pistorder (or Tweakmore; this article demonstrates with pistorder), and Replay mod (optional).

---

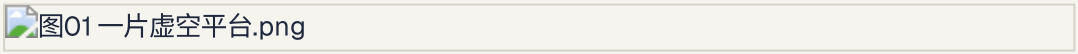
2 Environment Setup

Feel free to skip this section, though I'd still recommend giving it a read. Better tools always help with learning and design.

2.1 World Environment

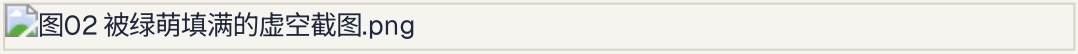
Open singleplayer, select **Creative Mode**, and create a **void world**.

`Create New World → World → World Type: Superflat → Customize → Presets → The Void`



If you spawn on a stone platform, you've succeeded.

It will eventually look like this:



Of course, given the community's research into bedrock breaking tech, we can change the preset to: `minecraft:bedrock;minecraft:the_void`

If you want multiple layers, just add $*x$ after the block name (where $0 < x <= 324$ and $x \in N^*$). For example: `minecraft:bedrock*5;minecraft:the_void` simulates a five-layer bedrock breaking environment.

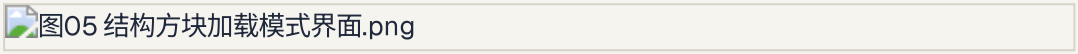


Of course, to ensure stability when building in survival, you'll eventually need to create a normal world for field testing. Releasing a machine without field testing is irresponsible, both to yourself and others.

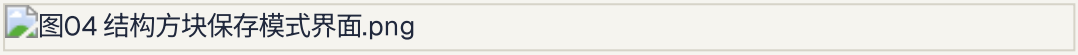
2.2 Structure Block

After countless explosions, breakdowns, and stuck-together messes, we found a local save tool simpler than backups: **structure blocks**. We mainly use two modes: **Save mode** and **Load mode**.

2.2.1 Save Mode



2.2.2 Load Mode

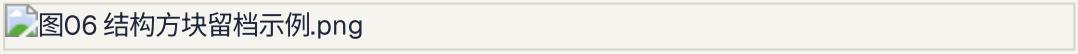


2.2.3 Necessary Notes

- Structure names cannot use Chinese characters.
- Both relative position and structure size have three input boxes, corresponding to the x, y, and z axes from left to right. You can check these by pressing F3 and looking at the crosshair in the center of the screen.
- **Save mode can only save structures at 0° rotation in the corresponding load mode structure box**.
- Include entities: In save mode, this saves entities within the structure box; in load mode, this loads entities that were saved in save mode.

2.2.4 Tips

You can save multiple versions of structures during iteration, making it easier to roll back and debug later. Here's an example:



In the save mode and load mode diagrams above, save mode handles archiving. In the archiving example, all the structure blocks are identical save-mode blocks the filename bedrock_breakers also shows they're a series of bedrock breakers, serving as the archive area. The work area has only one load mode structure block (not shown in the image).

The above is just what works for me; feel free to come up with your own approach.

2.3 Carpet Commands

Below is a brief overview of Carpet configuration. For more options, refer to other chapters or watch Carpet usage tutorials.

2.3.1 Carpet Configuration

Open your test world and enter commands in the chat. Once set, these settings generally won't need to be changed again.

`/carpet tntDoNotUpdate true`: When placing TNT, it won't be activated by redstone. This saves us from having to push minecarts or use other "loading" methods when making TNT duplicators.

`/carpet fillUpdates false`: Using fill, clone, setblock commands and structure blocks won't cause block updates. This especially solves the problem of machines "acting up" after loading structures.

`/carpet creativeNoClip true`: No collision in creative mode, letting you fly freely through machine structures without getting bounced around by slime blocks.

`/carpet structureBlockLimit 48/96/192/256`: The numeric options are separated by "/". Consider this option when you need a structure range larger than the default. **Note that in practice, overly large structure boxes can cause rendering issues and lag. Unless necessary, the default of 48 is sufficient**.

`/carpet explosionNoBlockDamage true`: When designing TNT-related structures, you can enable this debug option to prevent TNT explosions from destroying any blocks. Set it to false when you need to test whether TNT will destroy the expected area.

2.3.2 Game Tick Commands

- `/tick freeze`: Freezes game ticks, usually combined with `/tick step x` below to observe whether a structure operates as expected. Similar to setting breakpoints in code debugging.
- `/tick step x`: After freezing game ticks, this command advances the game by x game ticks.
- `/tick sprint x`: Similar to `/tick warp`, usually used for long-term machine stability testing or speeding up flying machine positioning (handy for debugging).
- `/tick unfreeze`: Unfreezes game ticks. Some versions use `/tick freeze` to toggle freezing on and off. Rant: I don't know why they split one command into two; it's a pain to type every time.

2.3.3 TNT Monitoring Commands

- `/log tnt`: Subscribe to TNT-related logging options.
- `/log tnt full` and `/log tnt brief`: When TNT explodes, these show you the spawn position and explosion point. Very useful for controlling TNT explosion points, which we'll cover in the practical section.
- `/log tnt clear`: Unsubscribe from TNT-related logging options.

2.3.4 Fake Player Commands

- `/player <Name> spawn`: The first step in using fake players; the most basic use is force-loading chunks when testing machines. For example, use `/player yu_xian520 spawn` to summon a fake player named Yu Xian. Follow Sanri Yuyin on Bilibili meow~
- `/player <name> use`: Makes the fake player use items, place blocks, or interact with blocks, equivalent to right-clicking. A similar command is `/player <name> use once`. Aiming the fake player at a note block or fence gate enables remote startup or stepping effects. For example, `/player Twisuki use` makes Twisuki click a note block or fence gate once.
- `/player <name> use continous`: Makes the fake player continuously use items, place blocks, or interact with blocks. Usually used to place pistons for bedrock breakers. Use `/player <name> use` to stop.

2.4 WorldEdit Commands

This section mainly covers `//set`, `//undo`, `//redo`, `//stack`, `//move` *and other commands. Players who have already mastered these can skip to the next section.*
This subsection only covers the most commonly used parts in slime tech design.

1. //set, //undo, //redo
   //set is a pretty basic command, right? Using the wooden axe, like with the projection mod, left and right click to select a range, then enter `//set 0`

If nothing goes wrong, the area you selected has been replaced with air. So how do you restore this platform?

- Use `//undo` to undo this operation;
- Use `//redo` to redo the operation you just undid.

1. `//move <Num>`
   This is also a basic command. After selecting an area, face a certain direction (or specify one) and use `//move <Num> <direction>` to move the blocks in that area. *For example, use //move 1 up to move up one block*. Without specifying a direction, it defaults to the player's facing direction. Use `//move <Num> -e` to preserve entities.

You can use this command to move created modules to appropriate positions.

1. `//stack <Num>`
   After selecting an area, you can batch copy it in a certain direction, which is very convenient for expanding slime tech units.
   This command works basically like `//move`, except instead of "move how many blocks" it's "copy how many times". No need to elaborate further.

Apologies if the above is a bit abstract. Try selecting an area and entering the commands yourself, or search for WE tutorials online.

2.5 Tweakeroo Configuration

1. **Disable Observer Option**:
   In the game interface, press `x+c` simultaneously. In the disable options, you can find the disable observer option. I suggest setting a shortcut key on the right side of the keyboard for easy access when debugging.

After enabling this option, observers become "stone" and no longer respond to updates. When using WE or other tools on slime tech structures, this avoids unpredictable results from observer activation. It also prevents accidentally updating observers.

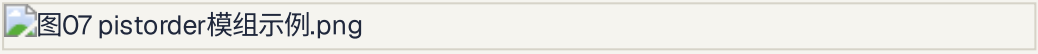
1. **Flyspeed (Flight Speed Control)**:
   Tunnel borers, mining machines, etc. often exceed 200 blocks in height. Moving around various parts of the machine is very tedious. Using the Flyspeed function greatly increases your flight speed, allowing quick travel between different sections.

2.6 Pistorder Usage

This is one of the essential mods for designing slime tech. Previous chapters have also covered pistorder and piston push order rules. Here's a brief overview of the slime tech-related aspects.

Right-click a regular piston with an empty hand, and the piston will display whether it can push (as shown in the image: Push √; if it cannot push, it displays Push ×). The blocks being pushed display their push order **from small to large**. This helps with compressing machines and quick debugging.

- For regular pistons: Right-click once with empty hand for direct push (piston arm directly contacts), right-click twice for pushing with a one-block gap (as shown below).
- For sticky pistons: Right-click once with empty hand for direct push, right-click twice for pulling back with a one-block gap.



2.7 Replay Mod Debug Tips

Replay mod is familiar to streamers or anyone who has recorded MC videos who are you calling unfamiliar!. It records events that occur in loaded chunks during the recording period.

Thanks to Replay mod's scenario replay feature, you can debug by scrubbing through the progress bar after playback. This trick is handy when reproducing a problem takes too long, or when debugging with `/tick freeze` is difficult.

3 Fundamentals

3.1 Timing Basics

3.1.1 Game Tick

We've already covered the basics of game ticks in previous chapters. This section focuses more on how game ticks apply to slime tech.

For readers who haven't studied the timing chapter, here's a quick summary of the key points:

- The basic unit of time in Minecraft is the game tick (abbreviated as **gt**).
- Different redstone components have different game tick delays. For example, observers have a 2gt delay, and repeaters delay 2gt per notch (so repeaters delay 2~8gt).

All content in this chapter will use **game ticks** instead of **Redstone Tick (RedTick, abbreviated as rt, 1rt = 2gt)**, because we'll later involve certain odd-tick timing relationships, and using game ticks is more convenient for calculation.

For slime tech, game ticks can be used to indicate the sequential movement order of each wiring path. For most cases, we only need to know their movement order. Below is my recreation of my first slime tech machine, a tunnel boring machine using piston chains as delays.

Image: Piston delay-type tunnel boring machine

Readers can also note that the upper sticky piston section has a 6gt delay, and the lower regular piston has a 4gt delay.

Players familiar with slime tech know that signal transmission starts from the upper sticky piston chain and ends at the lower regular piston chain. *This structure is used as TNT delay for tunnel boring machines, etc.* After becoming familiar with the signal delay above, we can calculate that the total delay in the above image is the sum of multiple sticky pistons and regular pistons, i.e., the sum of multiple 6gt and multiple 4gt.

Of course, through simple calculation we also know that with the above piston delay arrangements, we can create delays of 4gt and above, covering all even-numbered gt delays. **So how do we create a 2gt delay?** This seems a bit troublesome. We can usually only change a 6gt delay to a 4gt delay to advance by 2gt, or vice versa.

Since we already have even-numbered gt delays, **how do we create odd-numbered gt delays?** This seemingly difficult task is actually much simpler! Let's return to the 4gt delay piston structure in the image above. We can transform it into a transmission structure based on redstone blocks. With the redstone block relationship, the piston will trigger the next gt after arriving, thus creating a 3gt signal delay.

Image: Diagram of two types of piston delays (puzzle, combine two images into one)

Let's analyze the specific situation: For the observer structure in the upper image, at gt0 the piston and observer arrive, at gt1 the observer activates, at gt2 the piston extends, at gt3 the piston is extending, at gt4 the pushed block arrives. For the redstone block structure in the lower image, at gt0 the piston arrives, at gt1 the piston extends, at gt2 the piston is extending, at gt3 the pushed block arrives.

Combining the above 4gt and 6gt delay units, we can create 3gt, 7gt, and higher delay units.

Below we discuss a clever 7gt delay technique: Let's return to the 6gt delay structure in the image above. As shown, we place a powered block diagonally above and add another observer to QC it, creating a simple and clever 7gt delay. This is also applied in kades' three-way bedrock breaker:

Image: 7gt delay structure, with application part from kades' bedrock breaker.

We can still analyze the specific situation: I'll skip the detailed analysis for now.

Of course, based on the above principle, we can create flying machines with an 11gt cycle, or 11gt two-step structures, which are the perfect finishing touch when applied to trenchless world eaters.

Image: 11gt cycle flying machine and two-step structure

Of course, in many cases, we don't need to calculate delays so precisely. Many slime tech structures only need to reset before the next activation. Precisely because many structures are so-called "forgiving," there is a rather clever wiring style: **borrowing force**.

3.1.2 Borrowing Force

Since I first got into slime tech, I've heard something like this:

> Excellent wiring often applies a lot of borrowing force.

As mentioned in the previous section, borrowing force is a wiring technique where you reset structure A (which is not time-sensitive) through other structure B, or even a completely unrelated structure C, or even directly replace structure A's wiring. Sounds convoluted, right? Let's look at the image:

Image: Borrowing force diagram, preferably GIF

As shown in the image, the upper piston with no-delay linkage, after triggering, has its sticky piston reset by a lower no-delay linkage that may serve other purposes. This is borrowing force. This serves as a relatively simple example.

Combined with the game tick section above, we can use a certain moment of machine operation as the time origin (similar to the zero reference point in circuit analysis. Students in electronics are no strangers to this). Each module, and even each piston, can get a gt number label, so we can accurately analyze the sequential movement order of each part.

Image: My 14m high front-launching trenchless world eater

As shown, this is the departure station part of my 14m front-launching trenchless world eater. Using the signal output part of the TNT buffer as the 0gt point, we can measure that the time interval for launching the flying machine is 7gt. (Of course, different marking points will yield different times).

Except for situations with extreme requirements on machine size or timing, most of the time there's no need to calculate timing so rigorously. This subsection only provides a rational analysis approach to meet needs in extreme situations.

Image: Borrowing force unit of trenchless world eater, GIF

3.1.3 * b36 Push Order Time Analysis

This subsection can directly refer to the b36 push order time analysis in previous chapters. Here we mainly cover how to use b36's push order time to compress machines. For clarity, pistorder is still used here to assist in explanation.

(TBD)

8.2 Appendix 0a

1 Terminology Reference Table

Because different Slime Tech players may interpret terminology differently, this reference table is provided to ensure consistency throughout the chapter. The following terms will also appear as footnotes in the article (work in progress).

- Flying machine: Broadly, this refers to Slime Tech as a whole; narrowly, it refers to modules capable of self-propelled movement. This text uses the narrow definition.
- Engine: A flying machine, typically one that does not move in 10gt cycles.
- Piston worm: Broadly, any flying machine; narrowly, a movable flying machine where piston heads face directly opposite each other. This text uses the narrow definition.
- Self-returning flying machine: A flying machine that automatically returns after encountering an obstacle (such as hitting obsidian).
- Automatic: A flying machine that can move itself, i.e., self-propelled movement.
- Stepper: A module that performs limited movement after being activated, such as single-step (moves once per activation) or double-step (moves twice or two blocks per activation).
- Mounting: Attaching another module to a module via pistons or similar means. The mounted module is typically not automatic, or does not react to the module it is mounted on.
- Docking station: A Slime Tech structure that can dock and launch flying machines. Can launch flying machines after a step, or with zero steps (i.e., without the flying machine moving).
- Departure station: A docking station where machines such as world eaters are active by default.
- Return station: By default, the return station is located at the opposite end from the departure station. It can also serve as a docking station, though it may not always be able to dock flying machines.
- Extension: In this text, only repeating units that extend the machine, such as world eaters, mining machines, tunnel bores, etc. Can also be equivalent to stacking.
- Piston chain: Typically a chain-like device made of sticky pistons alternating with slime blocks, usually used for delay mechanisms or block transport.

Twisuki

Preface

Welcome to this Slime Tech tutorial, meow~ This is Suyang Twisuki.

Inject Twisuki, meow~ Inject Twisuki, thank you, meow~

Profile Bilibili Github

All right, enough fangirling. Back to the text. I had heard about this Slime Tech tutorial for quite some time, and at Molforte's invitation I joined the writing group and became responsible for writing this tutorial.

As a Slime Tech researcher (I do not dare call myself a developer; real developers all have quite a few original machines they can show. I can only count as a scholar), I first encountered Minecraft in 2013 and started learning redstone roughly two years later. But I have only been in contact with the modern Slime Tech community for the last three or four years. Although I do not have any particularly impressive machines of my own, I think that long-term, if unsystematic, study can still accumulate a great deal of knowledge and produce some valuable experience. I will exhaust my knowledge reserves and organize them into this small tutorial, hoping it can be of some help to you.

I think learning Slime Tech has roughly the following characteristics:

1. Slime Tech is not an especially arcane field. The gigantic machines you see are, at their core, only combinations of a limited set of components.

   You may marvel at how spectacular and grand an enormous machine looks, but in practice the larger a machine is, the more it proves that each small part of it is simpler, because it has not been compressed to an extreme degree. Once you understand the core and the operating logic, design becomes mostly a matter of wiring.

   Its timing requirements are not as precise as in some other fields. By this I mean that microtiming is basically not needed, and even when it appears, it is modularized; you only need to deal with that module and do not have to interfere with its internal logic. But Slime Tech is strict in that there is almost no fault tolerance: adding or removing a single block can break the machine, and manually changing one small part may also break the entire thing.

   There are very few moving components. Although there are many mechanics, compared with other fields there generally are not many especially magical alternative applications. In other words, you are unlikely to marvel at Slime Tech design at the component level, which greatly lowers the difficulty of understanding the machine.
2. Slime Tech cannot be learned metaphysically. If you do not practice, you will never know where the machine will break.

   Simulating it a thousand times in your head is not as useful as backing it up and running it once. When you think you have calculated the timing correctly, you will definitely have missed some connecting rod, some piston that may create a BUD, or some module stuck together so it cannot push or pull. Slime Tech design must be hands-on.

   If you only study theory without doing anything, you will end up like me: able to produce an elegant small module, but unable to make a complete machine yourself, and unable to build one even if you ignore elegant wiring...

Testing also requires backups. If you are worried that it will break, it definitely will; if you back it up before testing, it will still break, but at least you will suffer less.

3. As long as you study hard and practice more, you too can design your own machines like the experts.

   Do not believe the statement above. If you learn the knowledge well, you can only learn how to tear machines apart; if you practice well, you can only know how far you are from reassembling what you have understood...

   That is a bit discouraging, but it is true. Do not set your goals too high, because this tutorial only teaches basic literacy and introductory knowledge. We have to admit that the experts are geniuses. If you are the same, you probably do not need this tutorial at all. I cannot design machines myself either, so this is as far as I can teach...

Studying this assumes that you can read Chinese, know how to move, know how to open the inventory, and know how to interact with blocks. If commands or utility mods are involved, you need the ability to read English or use translation software. Most importantly, you must have the will to learn, to read through the tutorial, and to practice hands-on. I will cheer you on.

This tutorial is limited to Java Edition (offline is fine, but NetEase Edition is not included), and specifics should be analyzed version by version. In addition, all mods in this tutorial are used in English, including their names and functions, though the tutorial will introduce them.

If there are omissions, please contact me through the channels noted at the beginning.

Happy learning.

Twisuki Suyang

2025-01-20

After finishing Chapter 1, I rethought the purpose of writing this tutorial. Perhaps before preparing Chapter 2, I need to strengthen my own knowledge reserves first. My knowledge comes from practice, and practice is good, but it is also true that it is disconnected. I have fallen into "building behind closed doors," and what is more, the "vehicle" has not even been built yet...

This tutorial will try to follow these principles:

1. Standardization

   In this tutorial, whether for Minecraft itself or for various mods, terms are taken from the English-language interface wherever possible. I will try to use standardized and common terminology to keep the tutorial consistent.

   At the same time, because the Slime Tech field lacks beginner-oriented standard terminology, I will try to give accurate definitions and explanations. But this tutorial has limited influence and cannot become a unified standard for the entire Slime Tech community, so there will be many difficulties during writing.

2. Lightweight setup

In a technical-Minecraft environment where players often use enormous modpacks, configuring mods yourself or using only a small number of mods is not very common. But for beginners, directly downloading someone else's modpack, or asking others and receiving a large pile of mods, creates a lot of pressure when understanding is insufficient: compatibility problems, keybind conflicts, unknown mod functions, and unclear dependencies and dependents all become hard to solve independently.

I have always advised beginners not to use ready-made modpacks, nor to ask others for a large list of mods. I mainly write this tutorial using a 1.21.1 client with only 25 mods, and I am very familiar with the function of each one. During the tutorial, I will try to use the functions of existing mods rather than recommend installing new ones.

My recommended mod list is: `Carpet mod`, the `masa suite` without extensions (`Malilib`, `Tweakeroo`, `Litematica`, `Minihud`, `ItemScroller`), `Client Command`, `pistorder`, and `IMEBlocker`. Please explore other mods on your own.

3. Overall structure

   At the beginning of writing, this tutorial was remade several times, and it has now reached a relatively complete structure.

   The tutorial includes a `Preface`, a `General Outline (Table of Contents)`, the `Main Text`, and an `Afterword`. The main text is temporarily divided into 6 chapters and 14 sections. Each chapter is split into an introductory part and the main content, and the main content contains headings, section headings, and so on, forming a relatively complete structure.

As a beginner-oriented tutorial, this tutorial starts only from the basics of Slime Tech and does not explain other kinds of knowledge. Beginners will need to consult the `Minecraft Wiki` for those.

This updated preface will serve as an editorial standard to follow. I also hope readers can roughly understand the organization and structure of this tutorial and learn more effectively.

Happy learning.

Twisuki Suyang

2025-01-26

8.3 Introduction

> "A tree that fills a man's embrace grows from a tiny sprout; a nine-story tower rises from a heap of earth; a journey of a thousand miles begins with a single step"
>
> --- Laozi

In this chapter, you'll learn about commonly used mods for Slime Tech design, debugging methods, and other preparatory work. You'll also get an introduction to various Slime Tech components and simple flying machines.

The preparatory work section covers a lot of ground, and the first lesson alone might scare off quite a few people. I hope you can stick with it. Once you get through this stuff, the rest will feel much more natural (just kidding... mostly).

That's just how Slime Tech is. Happy learning!

8.4 Prerequisites and Setup

Slime Tech requires some mods, most of which are common in technical gameplay, such as Carpet, Tweakeroo, and Litematica. However, you also need some command skills, including using commands directly and working with command blocks. Additionally, we need to learn about structure blocks.

This tutorial does not require World Edit (~~because I don't know it either~~), so we need to be proficient with commands.

1 Vanilla Commands

This section explains vanilla `fill`, `setblock`, `clone` commands and `effect`, `enchant` commands. It's fairly basic and can be skipped if you're familiar with them.

1.1 a) Block Interaction Commands

1.1.1 /setblock

The basic syntax of the setblock command is `setblock <pos> <block>`, which places a block at `<pos>`, where `<pos>` is a set of `<x> <y> <z>` coordinates.

In Slime Tech, machines are typically designed up in the sky, so you generally place a block at `~ ~ ~` as a starting point for building.

Input like this:

```
/set<tab> <tab> <tab> <tab> <tab>
```

You will place an acacia button, and then place another block to start building.

1.1.2 /fill

The basic syntax of the fill command is `/fill <from> <to> <block>`, which fills the cuboid between the two diagonal corners `<from>` and `<to>` with the specified block, where `<from>` and `<to>` are each a set of `<x> <y> <z>` coordinates.

This command is very commonly used for things like clearing large areas of blocks, creating sites for tunnel bores and world eaters, digging world eater trenches, etc.

Operate like this:

```
<Look at the starting block>
/fill <tab> <tab> <tab> <Enter>
```

You will get the following prompt:

```
Unknown or incomplete command, see below for error
```

```
...<x> <y> <z> <-- [Here]
```

Next:

```
<Look at the ending block>
/<up> <tab> <tab> <tab> ...
```

This allows you to fill between two positions with the specified block.

Of course, the second position is often `~ ~ ~`, generally used for testing or clearing.

1.2 b) Structure Interaction Commands

1.2.1 /fill Advanced Usage

The fill command can take several additional parameters after `block`. The complete fill command syntax is:

`/fill <from> <to> <block> [replace|keep|outline|hollow|destroy]`

We will explain each one:

1. `destroy` parameter

After adding this parameter, the fill command destroys blocks, causing them to drop as items.

2. `hollow` parameter

After adding this parameter, the fill command only fills the outer layer, clearing the interior, forming a hollow shell.

This command can directly generate a small house, which is convenient for testing other things, but may not be very useful for Slime Tech.

3. `keep` parameter

After adding this parameter, filling will preserve existing blocks, only filling air blocks.

4. `outline` parameter

Similar to the `hollow` parameter, but the interior will not be cleared.

5. `replace` parameter

I personally think this is the strongest parameter in the fill command. The syntax is as follows:

`/fill <from> <to> <block> replace <block0>`

This command can detect `<block0>` within the range and only replace the specified block.

This command gives designers the ability to selectively replace blocks, such as after design is complete, changing all terracotta to face the same direction, or swapping various colored glass for a single type...

Operate as follows:

```
/fill ... ... minecraft:white_stained_glass replace #c:glass_blocks
```

Combined with tag selectors, replace all glass within the range with commonly used white glass.

Besides these parameters, here we will learn about block tag selection.

For example, if we fill a bunch of slabs, even when using the above `replace` parameter for selection, the fill will still produce bottom slabs. For top slabs (generally used to attach redstone dust, repeaters, etc.), this will cause serious problems.

So we need to learn how to select block properties.

Open F3, the bottom right area is the `Targeted Block` section, showing the properties of the block selected by the crosshair. Taking smooth stone slabs as an example, you will see the following information:

```
Targeted Block: ..., ..., ...
minecraft:smooth_stone_slab
type: bottom
waterlogged: false
#minecraft:mineable/pixkaxe
#minecraft:slabs
```

This information from top to bottom shows: block coordinates, block name, properties (for this block, `type` and `waterlogged`), and categories (for smooth stone slabs, mineable/tool type and slab type).

These properties are what we can use for selection. Below shows how to replace all smooth stone top slabs with smooth quartz top slabs:

```
/fill ... ... minecraft:smooth_quartz_slab[type=top] replace minecraft:smooth_stone_slab[type=top]
```

Both the middle `<block>` and the ending `<block0>` specify `[type=top]`, meaning "slab type = top".

Later we will explain how to conveniently view block properties and select the correct type (see 1.1.5).

1.2.2 /clone

The clone command has two basic syntax forms and can also take parameters. The `/help` description isn't very clear, so below we only introduce a simpler and commonly used one:

`/clone <begin> <end> <destination>`, where `<begin>`, `<end>`, `<destination>` are each a set of `<x>`, `<y>`, `<z>` coordinates.

This command copies the blocks in the cuboid between the two diagonal corners `<begin>` and `<end>` to the `<destination>` position, with `<begin>` mapping to `<destination>`.

Note: This command will produce updates when copying. You need to use Carpet features to disable updates (see 1.1.2 a)).

1.3 c) Other

1.3.1 /effect

The basic syntax of the effect command is `/effect give <targets> <effect> <seconds>` and `/effect clear [<target>]`. The former is used to give effects, the latter is used to clear effects.

During design, you may need to give yourself a night vision effect (although you can just adjust Gamma directly).

1. Effect giving

The syntax is `/effect give <target> <effect> <second> <amplifier> <hideParticles>`, where `<target>` is the target entity, such as `@a`; `<second>` is the duration; `<amplifier>` is the effect level; `<hideParticles>` is a `bool` value indicating whether to hide particle effects.

In higher versions, `<second>` can directly be filled with `infinite` to indicate infinite duration; in lower versions, you can fill in a bunch of 9s until it turns red (out of range).

`<amplifier>` does not apply to certain effects, such as night vision, which has no strength distinction.

`<hideParticles>` fill in `true` to prevent particles from interfering with vision.

Night vision operation is as follows:

```
/effect give @a minecraft:night_vision infinite 1 true
```

2. Effect clearing

The syntax is `effect clear <targets> <effect>`, where both `<target>` and `<effect>` are optional. If left blank, it clears all effects from yourself.

1.3.2 /enchant

The basic syntax of the enchant command is `/enchant <targets> <enchantment> [<level>]`, which can enchant the item held in the main hand of the selected `<target>` player.

This command seems to have no use for Slime Tech... But if a mob spawned on your machine or... ~~well, it really has no use~~ you can give yourself a strength effect, then enchant a netherite sword with sharpness 5 to quickly clear it ~~but why not just kill~~

~~Never mind~~, obtaining a Sharpness V netherite sword is as follows:

```
<Find a netherite sword in inventory and hold it in main hand>
<Close inventory>
/enchant @a minecraft:sharpness 5
```

2 Carpet Commands

Carpet Mod, commonly called "Carpet" in Chinese, is a powerful and easy-to-use utility mod written by Gnembon. Download and installation are not covered here.

Most of the commands in this section are explained in technical gameplay tutorials and videos, and can also be skipped if you're familiar with them.

2.1 a) Carpet Command Panel

Enter `/carpet` to open the Carpet command panel. You will see the following prompt:

```
Current Carpet Mod settings:
Carpet Mod version: ...
Browse categories:
[dispenser] [feature] ...
```

1. Language

Since Carpet has built-in Chinese, we don't need to "struggle with" English (although this tutorial still uses English mode for explanations).

The language switching command is `/carpet language [<value>]`, where `<value>` can be `en_us`, `fr_fr`, `pt_br`, `zh_cn`, `zh_tw`, corresponding to American English, French, Brazilian Portuguese, Simplified Chinese, etc.

Switching to Simplified Chinese is as follows:

```
/carpet language zh_cn
```

Now if we use the `/carpet` command again, we can get the following information:

```
目前启用的 Carpet Mod 选项:
   - 语言 (language) [en_us] [fr_fr] [pt_br] [zh_cn] [zh_tw]
   Carpet Mod 版本: ...
   浏览分类:
   [发射器] [特性] ...
```

We can see that `[en_us]` in `Language` is green, meaning default; `[zh_cn]` is underlined, meaning current.

2. Updates

Carpet can prevent TNT from being directly ignited when placed (useful for TNT duplication machines, no need to push minecarts during design), and prevent block updates when using commands like fill.

`/carpet tntDoNotUpdate true` can disable updates when placing TNT. When we directly place TNT next to a power source, the TNT will not be ignited and will only be ignited after receiving an update.

`/carpet fillUpdates false` can disable block updates during fill. In addition, updates from `setblock`, `clone`, and structure blocks will also be suppressed.

3. Explosion Protection

`/carpet explosionNoBlockDamage true` when enabled, explosions will not destroy blocks (explosions here include TNT, creepers, end crystals, etc.), but damage and knockback will still occur.

This command is very commonly used. ~~You don't want your machine to explode before it even starts, right?~~

4. Set Default

After you enter the above commands, a prompt will pop up as follows:

```
tntDoNotUpdate: true, [Change permanently?]
```

In Simplified Chinese mode:

```
tntDoNotUpdate: true, [永久更改?]
```

After clicking, the input box will automatically fill in the following command:

```
/carpet setDefault tntDoNotUpdate true
```

After confirming, every time we re-enter the game, this state will be maintained.

It is recommended to set the two update commands as default. The explosion protection command is not recommended (if you test a bedrock breaker and forget to turn off explosion protection, the machine will be completely stuck together).

5. Other

`/carpet flippinCactus true` cactus wrench. After enabling, you can use a cactus to adjust block direction; when the cactus is in your offhand, blocks will be placed in the opposite direction.

`/carpet creativeNoClip true` creative no clip. After enabling, when flying in creative mode, you lose your collision box like in spectator mode, meaning you can pass through walls but cannot push entities.

2.2 b) Fake Player Commands

`/player` should be the most used command in all of Carpet. Use this command to summon fake players, make fake players perform various actions, and kick out fake players.

1. Summoning and Clearing

Use `/player <name> spawn` to summon a fake player at the current position. If you want to control the spawn position and initial facing direction, use the following command:

```
/player molforte spawn at <pos> facing <direction> in <dimension> in <gamemode>
```

`<name>`: `molforte` is case-insensitive, prioritizes matching real players, then generates non-existent players. Here we use Molforte, the author of another tutorial, as an example.

`<pos>` is a set of `<x>`, `<y>`, `<z>` coordinates; after `facing`, `<direction>` is a set of `<yaw>`, `<pitch>`, i.e., azimuth and pitch angles.

After `in`, `<dimension>` is the game dimension, and after the next `in` is `<gamemode>` game mode.

Use `/player <name> kill` to clear an online fake player.

You can summon a fake player, throw items to them, then kill them, and go online at a new location to kill them. This is fake player express delivery.

2. Actions

Use `/player <name> (stop|use|jump|attack|drop|dropStack|swapHands|sneak|unsneak|sprint|unsprint|look|turn|move)` etc. to make fake players act. The functions are self-explanatory.

For example, `/player Molforte attack` makes the fake player Molforte perform an attack/dig (provided that the fake player Molforte exists).

Most operations can be followed by parameters. Taking attack as an example:

`/player Molforte attack` can be followed by parameters `continuous`, `interval`, `once`, which are continuous attack, interval attack, and attack only once, respectively. Interval attack needs to be followed by the interval tick count.

2.3 c) tick Commands

In higher versions, this command has been integrated into the MC main game, but it is still recommended to install Carpet.

1. Freeze

Use `/tick freeze` to freeze the game. Use it again to unfreeze; in higher versions, you need to use `/tick unfreeze` to unfreeze.

After freezing, the game stops running, but players are not affected.

2. Speed Adjustment

Use `/tick rate <rate>` to adjust the game tps (Ticks Per Second), the number of game ticks per second. Default is 20, i.e., 1s = 20 ticks, 1 tick = 0.05s.

3. Step-by-Step Action

With the game frozen, you can use `/tick step <time>` to run the game step by step. When observing something in motion, you can let time pass bit by bit for clearer observation.

For example, when observing a piston, first `/tick freeze`; then place a redstone block next to the piston, `/tick step 2` and you'll find the piston has extended. (This is the observed extension; the specific piston operation time is debatable.)

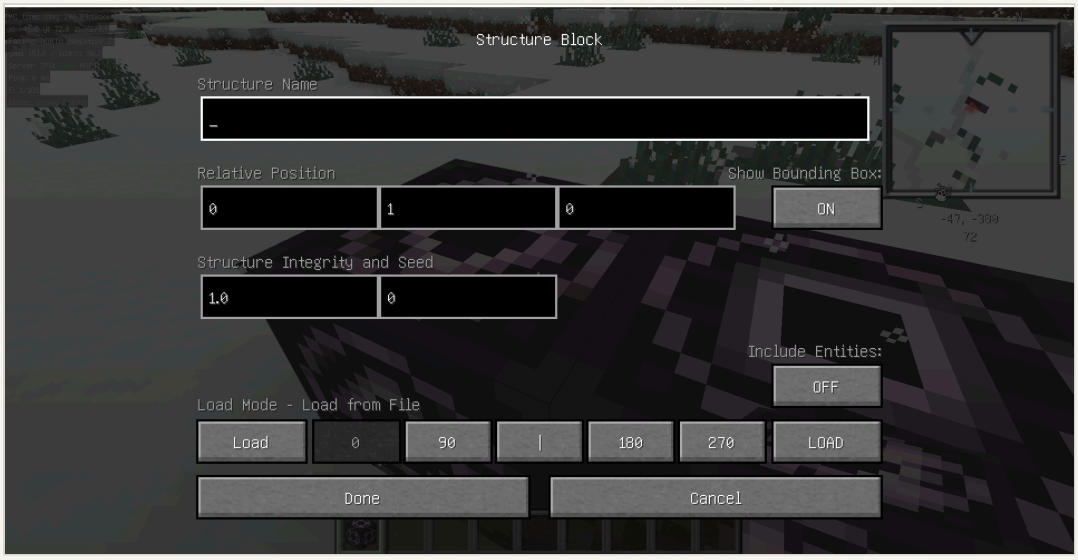
During testing, most tests are conducted at 4-tick time intervals, which is the time for a sticky piston in a flying machine to fully extend and retract in most cases.

3 Structure Blocks

3.1 a) Obtaining

Use the `/give @s minecraft:structure_block` command to obtain a structure block.

Structure blocks have three modes (four in older versions), which are `Load`, `Corner`, `Save` modes, in Chinese `加载模式`, `角落模式`, `保存模式`. Structure blocks default to `Load` mode when placed.



Click `Load` in the bottom left corner to switch modes.

3.2 b) Saving and Loading

In `Save` mode, first enter the `Structure Name`, then adjust `Relative Position` and `Structure Size` to frame the selection (the former is the relative position between the selection base point and the structure block, the latter is the selection size), and click `DETECT` to start detection.

At this point, you will see a white frame marking the selection. Now click `SAVE` in the bottom right corner to save the selection. The chat area will prompt as follows:

```
Structure saved as 'minecraft:...'
```

Now we prepare a `Load` mode structure block, fill in the name just saved in the `Structure Name`, then adjust the `Relative Position`.

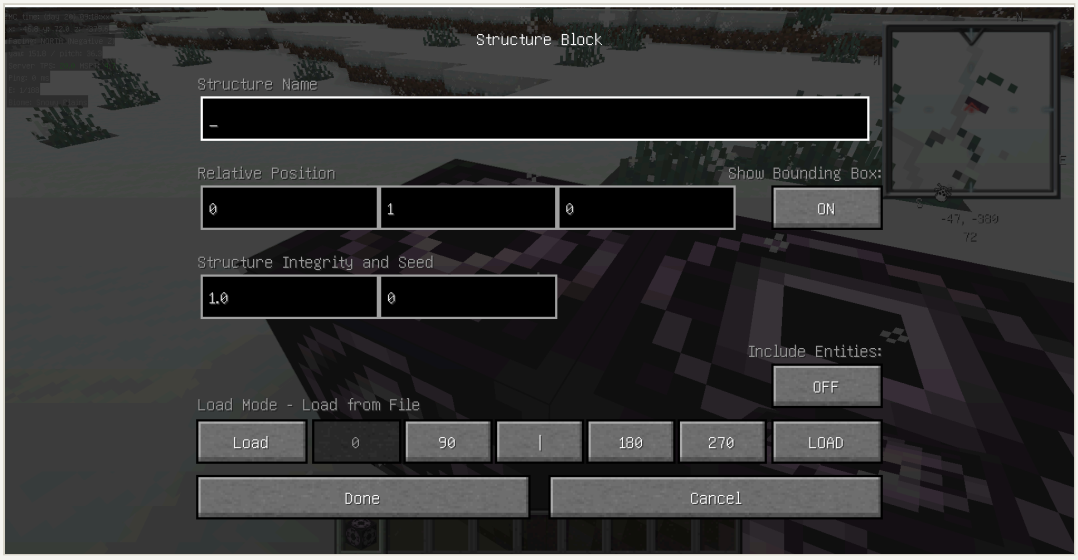
Click `LOAD` in the bottom right corner. A white frame appears in the game marking the selection range, and you get the following prompt:

```
Structure 'minecraft:...' position prepared
```

At this point, open the structure block and click the `LOAD` button again, and the structure will be loaded.

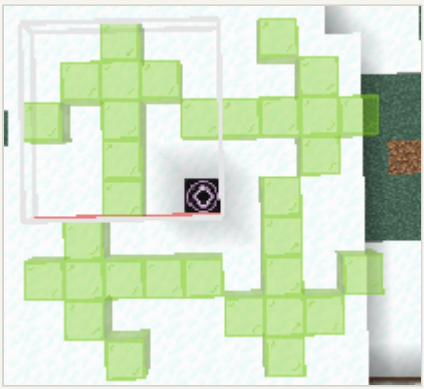
This loading will produce updates, see 1.1.2 a)

In addition, `Load` mode can also perform rotation and mirroring.



In the buttons `[0] [90] [|] [180] [270]` below, `[0] [90] [180] [270]` are mutually exclusive, indicating the structure rotates clockwise around the base point by that many degrees.

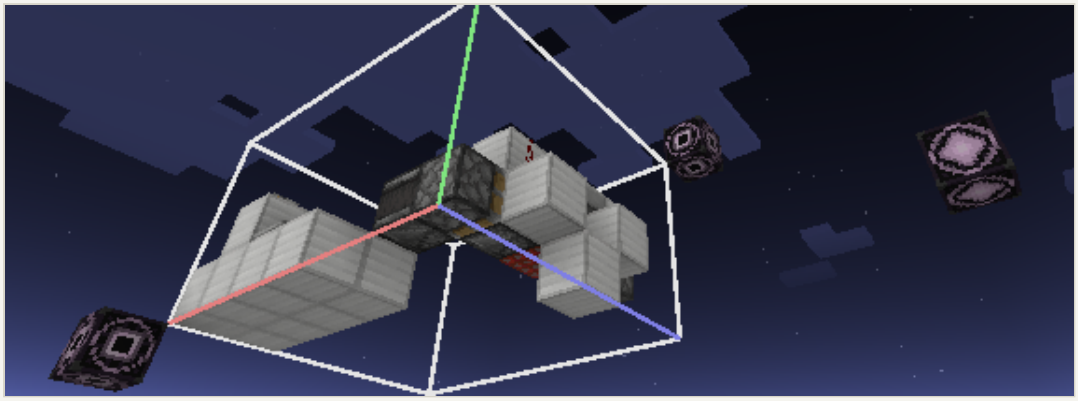
The image shows four arrows at different rotation angles, where the white frame marks 0 degrees.



`[|]` is a selection button with three modes: `[|]`, `[← →]`, `[↑ ↓]`. They mean normal, left-right mirror, and front-back mirror respectively. Please try them yourself.

In addition, when framing size in `Save` mode, you can use `Corner` mode for framing. Write the same `Structure Name` in `Corner` and `Save`, then place two `Corner` mode structure blocks at two diagonal corners of the structure. Click detect again, and you'll find the structure has been successfully detected.

Note: `Corner` mode detection size does not include the structure block itself.



4 Litematica

Litematica Mod, commonly called "投影" (projection) in Chinese, is a powerful, frequently used, and easy-to-learn mod.

Since Litematica Mod is very commonly used with many tutorials available, this tutorial will not cover the download, installation, or simple functions like selection/projection/paste. We will only explain some relatively niche features that I use.

4.1 a) Easy Place Protocol

`easyPlace` is a very commonly used function in Litematica, but it may have issues with stairs, slabs, and even note blocks. Below we explain how to select the easy place protocol.

The `easyPlaceProtocolVersion` path is `Litematica Main Menu/Configuration menu/Generic/easyPlaceProtocolVersion`, corresponding to Chinese `投影主菜单/配置菜单/通用/轻松放置协议` (also translated as "简单放置协议").

`easyPlaceProtocolVersion` has five options: `Auto`, `Version 3`, `Version 2`, `Slabs only`, `None`, with default being `Auto`. If you can't figure out the differences between these protocols, I recommend just picking the highest one: prioritize `Version 3`, and if the server doesn't support it, use `Version 2`. Other modes are generally not used.

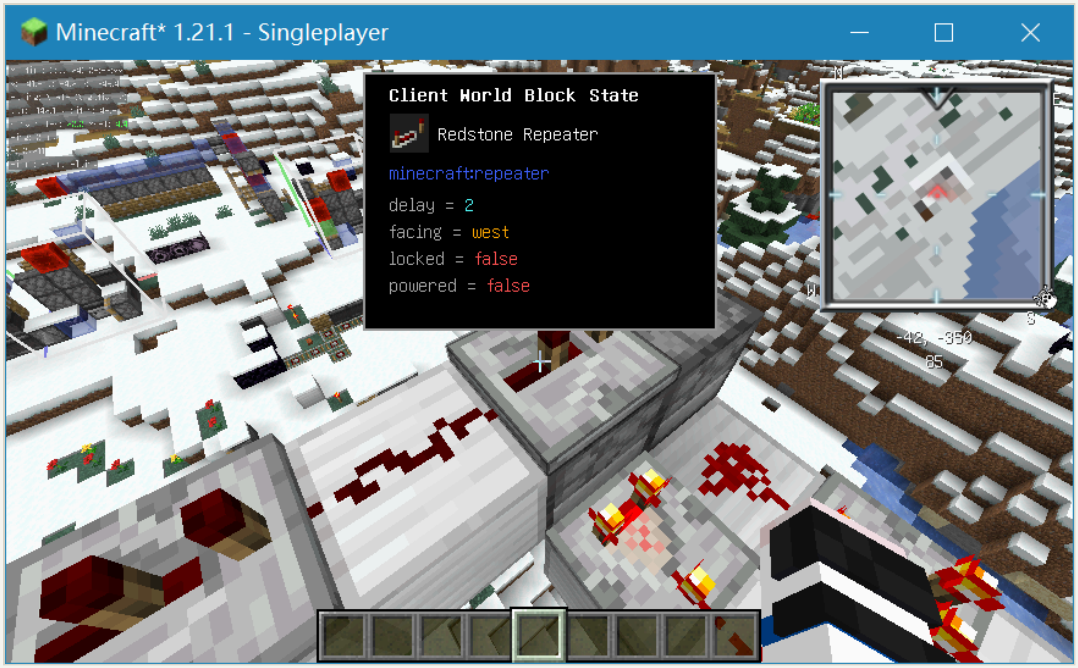
4.2 b) Block Information

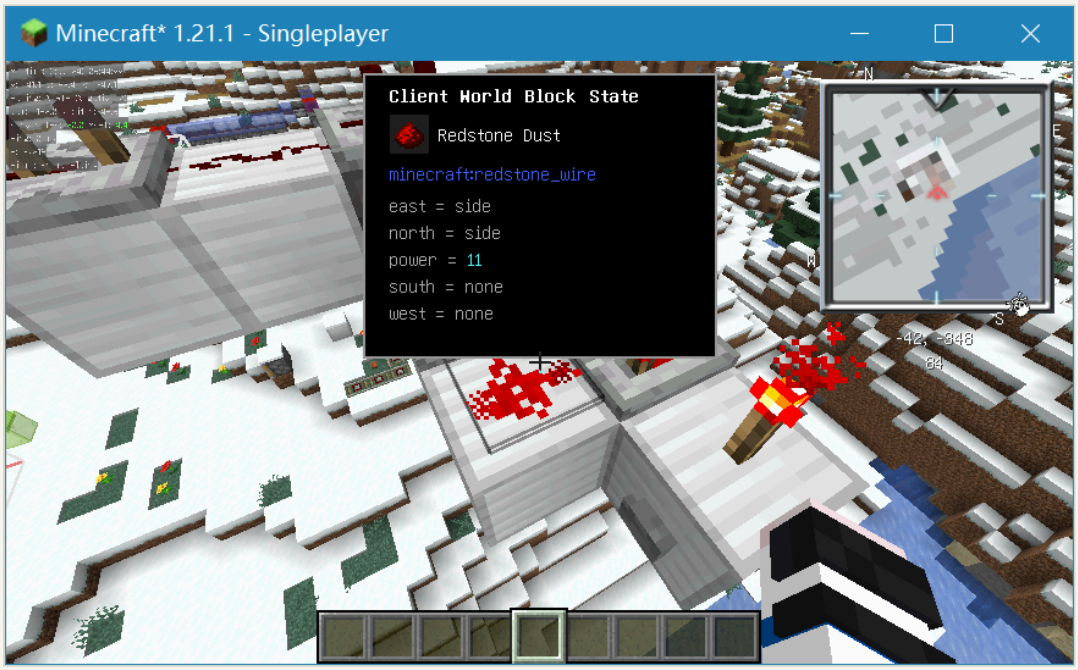
`renderInfoOverlay` displays property information for the selected block. Some mods (such as Jade) can also display block information, but I don't recommend beginners install a large number of mods, and Litematica's block information display is completely sufficient.

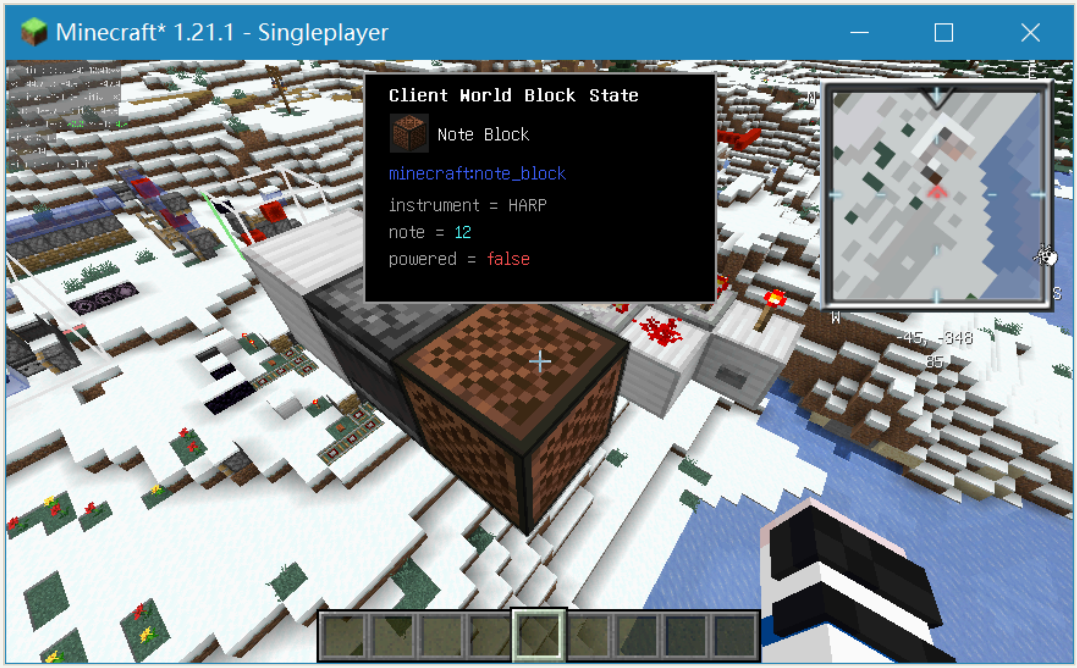
The `renderInfoOverlay` path is `Litematica Main Menu/Configuration menu/Hotkeys/renderInfoOverlay`, corresponding to Chinese `投影主菜单/配置菜单/热键/方块信息`.

Since the `hotkey` section has many functions, it's recommended to search directly for the `info` keyword. Make sure you don't pick `toggleInfoOverRendering` (`方块信息展示`) by mistake.

I set the hotkey to `LAlt`. When using it, just point the crosshair at the block and press left Alt, as shown in the image.

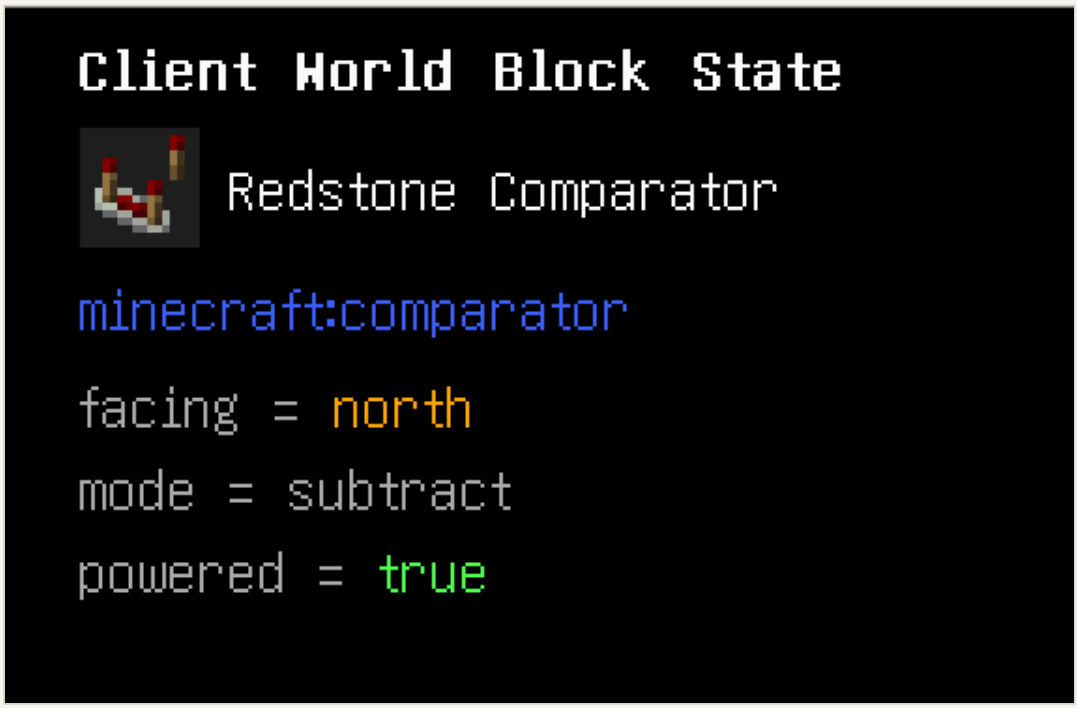

Minecraft* 1.21.1 - Singleplayer
Client World Block State
Redstone Repeater
minecraft:repeater
delay = 2
facing = west
locked = false
powered = false
-42, -359
85


Minecraft* 1.21.1 - Singleplayer
Client World Block State
Redstone Dust
minecraft:redstone_wire
east = side
north = side
power = 11
south = none
west = none
-42, -348
84



The images show the door control counter of a stacked village iron farm in versions 1.14 and below.

You can see the block thumbnail, block name, registry name, and then the block properties. Let's analyze using a comparator as an example.




```
Client World Block State
... Redstone Comparator
minecraft:comparator
facing = north
mode = subtract
powered = true
```

This is the same as the `Targeted Block` information in `F3`. Properties can be directly translated: facing = north, mode = subtract, powered = true.

8.5 Component Basics

It's "components" (元件), not "original parts" (原件). Circuit components, control components, these are all "components" - think of them as minimal structural units.

This lesson covers what various "active" components do and how they work. We won't go too deep here - this is just an introductory tutorial.

When learning Redstone, we often compare electric current to water flow, and Redstone energy to electric current. But in Slime Tech, we don't have any good analogies - you just have to memorize how things work.

1 Signals

Redstone Blocks can produce stable signals for use by pistons and other components.

Observers can produce a 2gt signal when they detect a change or when they themselves move. A signal of this length can cause a sticky piston to extend a block without retracting it.

Some signal sources may also use detector rails with minecarts, typically for TNT duper propulsion.

You can use observers paired with note blocks, fence gates, or trapdoors to transmit signals. Most of the time, they also serve as switches for the entire machine.

"Let's just start it directly" - breaking the fence gate directly produces an update on the observer, which also starts the entire machine

2 Pistons

Pistons come in two types: regular pistons and sticky pistons. Regular pistons extend when they receive a signal and retract when the signal cuts off. During extension, they can push blocks in front of them, up to a limit of 12 blocks. Sticky pistons work the same way, except when the signal cuts off and the piston retracts, it can pull blocks back. Both push and pull limits are 12 blocks.

Blocks that cannot be pushed by pistons include containers, both types of obsidian, bedrock, and extended pistons. Some blocks will be destroyed when pushed, such as plants and certain Redstone components. Notably, glazed terracotta can be pushed but cannot be pulled back.

When sticky pistons receive a signal of 2gt or less, they will extend the block without retracting it. This can be used to create bidirectional flying machines.

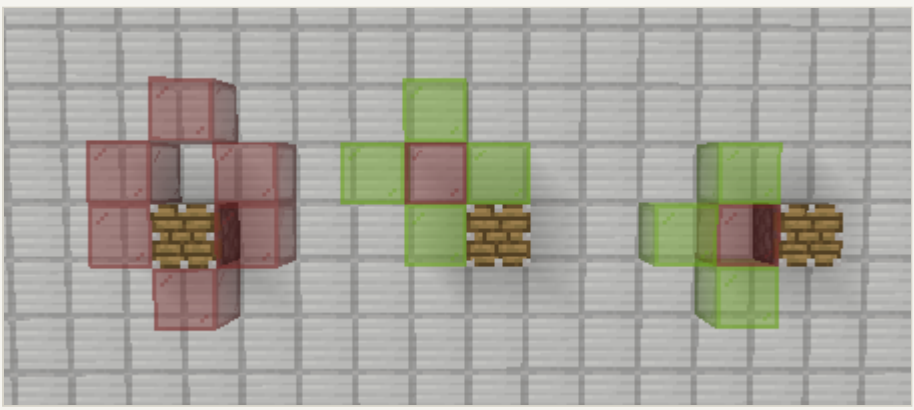


Sticky pistons, from left to right in the image: 2gt, 2gt, 4gt

Pistons also have a special property called "Quasi Connectivity", abbreviated as `QC`. Because pistons were coded like doors in the 1.5 era (**Note: this has been verified as incorrect**), signals from the 5 spaces around the block above the piston can also keep the piston extended.

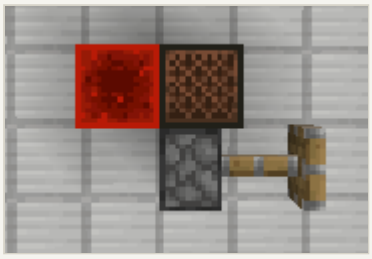
> **Note: there is an error here, which has been verified**
>
> 1. Pistons themselves do not have the *"coded as doors"* explanation; this claim is a rumor.
> 2. The explanation of `QC` here is incorrect. For details, see GTMC Update Theory 1.3.2 Properties of NC Updates, QC Activation, BUD Devices

Pistons need both a signal and an update to extend. When a signal is transmitted directly to the piston, it simultaneously updates the piston, so the piston extends. When a signal is transmitted above the piston and the piston hasn't received an update, it will retain its original state. After the piston is updated, it changes to the corresponding state. The update referred to here is an `NC update` (Neighbor Change). See the update theory tutorial for details.



The image shows the piston range. Left shows positions that can provide signals to the piston. Center and right show update positions that can be produced by two signal positions. The image only shows the same layer; a three-dimensional diagram is not shown

In the image, the Redstone Block can provide a signal to the piston but cannot update the piston. Here, the Redstone Block first updates the note block, and the note block updates the piston. Therefore, the piston can extend.



Most blocks in Java Edition are not "event listeners" that constantly check whether their state needs to change. Instead, they need updates from other blocks to trigger a check. So when a block is placed, its state changes, or it gets powered, it updates surrounding blocks. Only then will those surrounding blocks check their own state.

Therefore, you need to shift your thinking from "components actively detect" to "components passively detect"

3 Other Blocks

Note blocks, fence gates, and trapdoors can change state when they receive a signal, and this change can be detected by observers, thereby transmitting the signal onward. Note blocks, fence gates, and wooden trapdoors can also accept player clicks and act as switches.

When rails are pushed, they won't drop if the target position can maintain the current state; otherwise, they will be destroyed. (Maintaining the current state here refers to the rail's `shape` property. For example, if an inclined rail is pushed and there's no block for it to attach to after pushing, the rail will drop)

TNT will be ignited when it receives a signal and update. Redstone lamps will light up instantly when they receive a signal and stay lit for 4gt after losing the signal before turning off.

4 Blocks

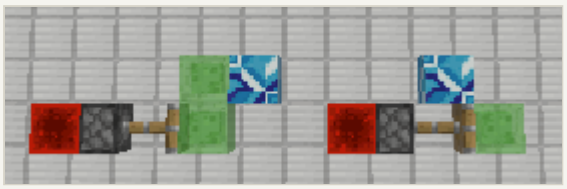
What kind of component is a block? It's not really one, but water systems can't exist without pipes, and circuits can't exist without wires. Blocks in Slime Tech serve this purpose.

4.1 a) Sticky Blocks

The traditional sticky block is the `Slime Block`, also known as `史莱姆方块`. This is also the origin of the name "Slime Tech" (绿萌).

Slime Blocks can stick to each other and move together when interacting with pistons. They can also stick to various components and auxiliary blocks. Since the piston push limit is 12, Slime Block chains are generally limited to 12 blocks.

Slime Blocks cannot stick to blocks that pistons cannot push, blocks that are destroyed when pushed, or glazed terracotta. Notably, glazed terracotta can be pushed but cannot stick to other blocks.



Honey Blocks were added in version 1.15, giving Slime Tech players more options. Honey Blocks have the same sticking properties as Slime Blocks but do not stick to Slime Blocks.

Honey Block surfaces are not continuous and cannot be powered or have rails attached.

4.2 b) Functional Blocks

In Slime Tech, besides components, some blocks also have specific functions. For example, cauldrons and composters are often used as "seats" for flying machines. There are also fences and walls used for collision box clipping, and various sized blocks used for precise positioning. These blocks are numerous and have complex classifications (after all, you can't imagine what designers stuff into machines to achieve certain functions). We'll analyze them as we encounter them.

4.3 c) Auxiliary Blocks

Auxiliary blocks are roughly divided into these categories:

1. Placeholder blocks: glass, etc.

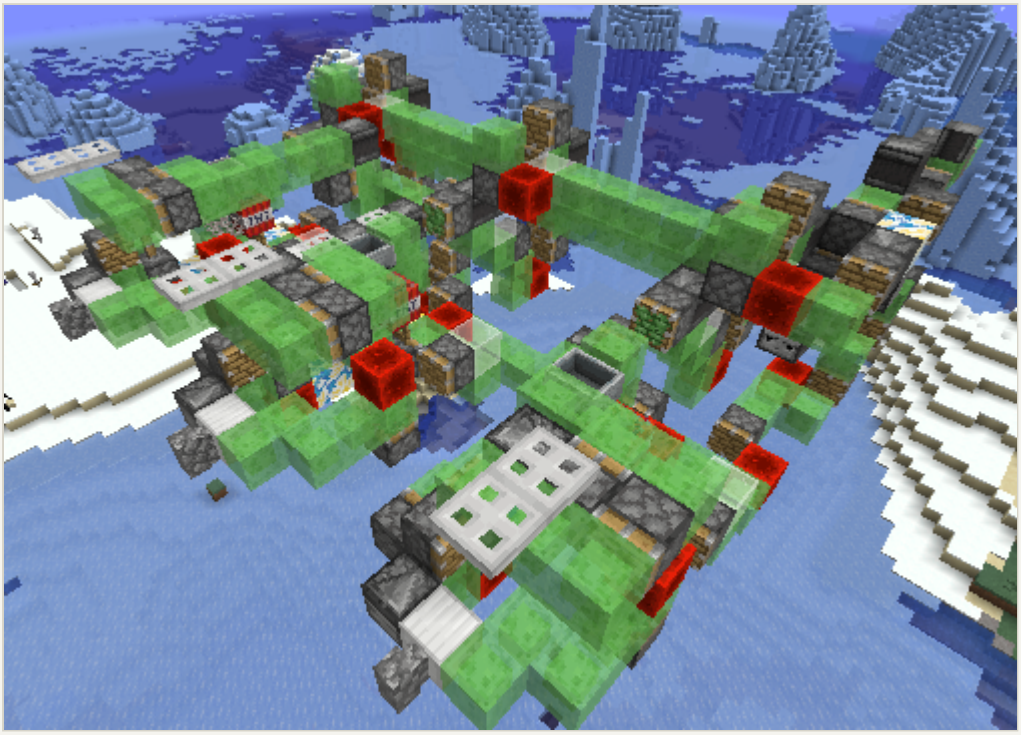
These blocks don't serve any functional purpose; they're only used to push (hold) other structures. They are the most basic blocks.

2. Powerable blocks: smooth stone, iron blocks, and other solid blocks

Building on placeholder blocks, these blocks need to be able to be powered and are used for signal transmission.

3. Special sticking blocks: glazed terracotta

The special nature of glazed terracotta (commonly abbreviated as "terracotta") has been explained above and won't be repeated here.



The image shows Starry Night's first-generation rapid-fire tunnel bore

8.6 Flying Machine Basics

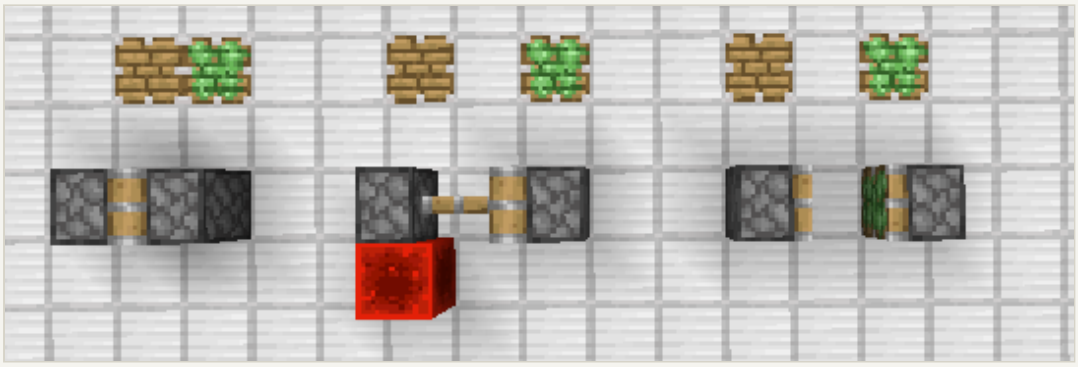
After two full lessons, we're finally building a machine that actually moves!

This lesson covers only basic flying machines. We won't go too deep into the motion mechanics, so it'll be much easier than the previous two lessons.

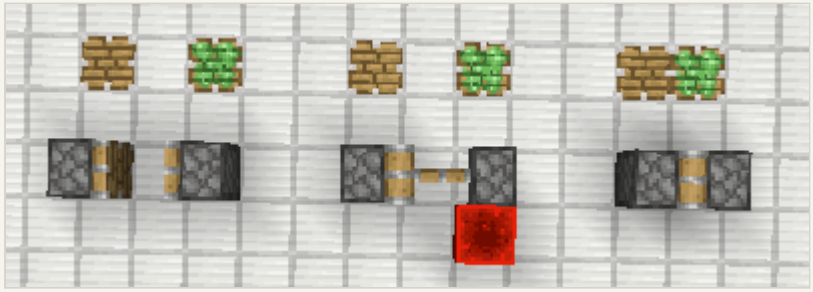
1 Principle

Step on your right foot with your left foot and spiral up to the sky!

High school physics and momentum ruined our martial arts dreams. Fortunately, Minecraft blocks don't have momentum. So we can pull off the following trick.



As shown, we've achieved the "stepping on the right foot with the left foot" process. During this, the regular Piston acting as the "right foot" doesn't change position.



As shown, this is the process of pulling back the "right foot". During this, the sticky piston's position doesn't change.

Through these two steps, the entire structure moves forward by one block. Let's connect them together.



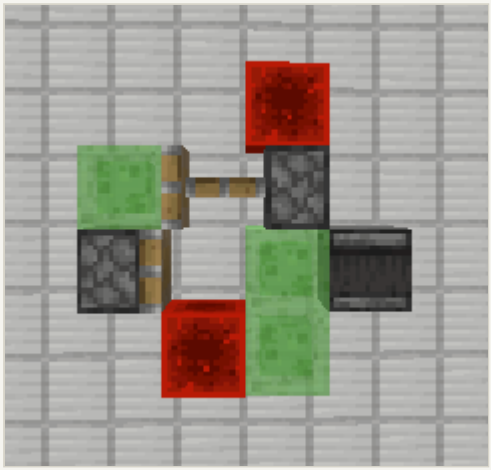
▶ This figure is animated. View original

By inserting a sequentially activating circuit into the flying machine body, we get a simple flying machine.

2 Simple Flying Machines

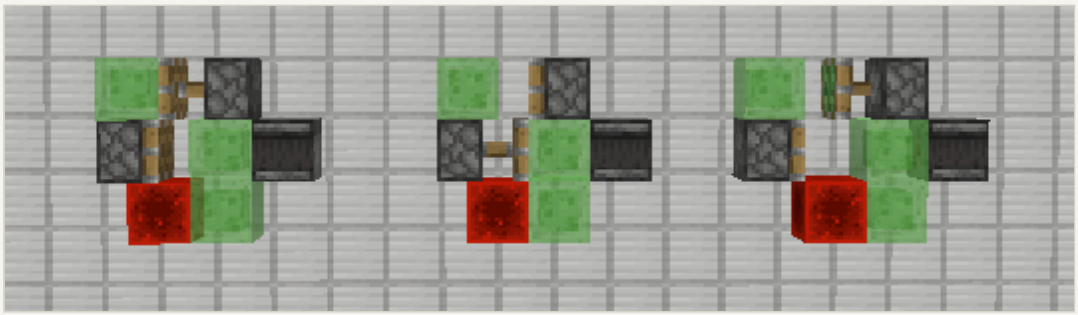
2.1 a) 9gt Unidirectional Flying Machine

As shown, this is a very straightforward 9gt unidirectional flying machine.



The right-facing one is a regular Piston, the left-facing one is a sticky piston. With right as forward, break the topmost Block of Redstone to start it.

Let's use `/tick freeze` and `/tick step` to observe how this flying machine operates.

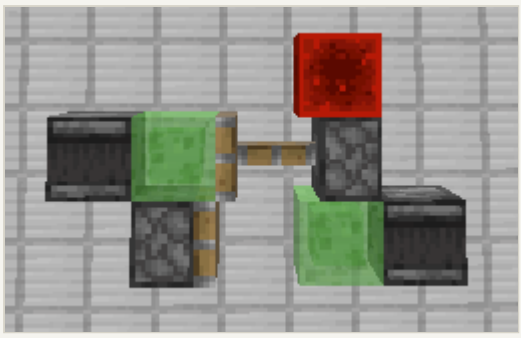


As shown, this is the flying machine in operation. The sticky piston first pulls back the rear section. Then the rear regular Piston, powered by the Block of Redstone, pushes out the front section. Once the front section is in place, the Observer fires, and the sticky piston pulls back the rear section again, completing the cycle.

This flying machine has a period of 9gt

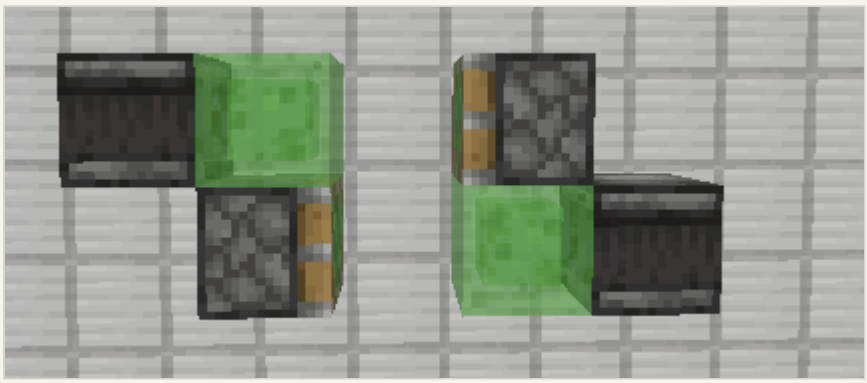
2.2 b) 10gt Observer Bidirectional Flying Machine

In the above flying machine, the regular Piston is activated by a Block of Redstone, while the sticky piston is activated by an Observer. Let's switch both to Observer activation, as shown.



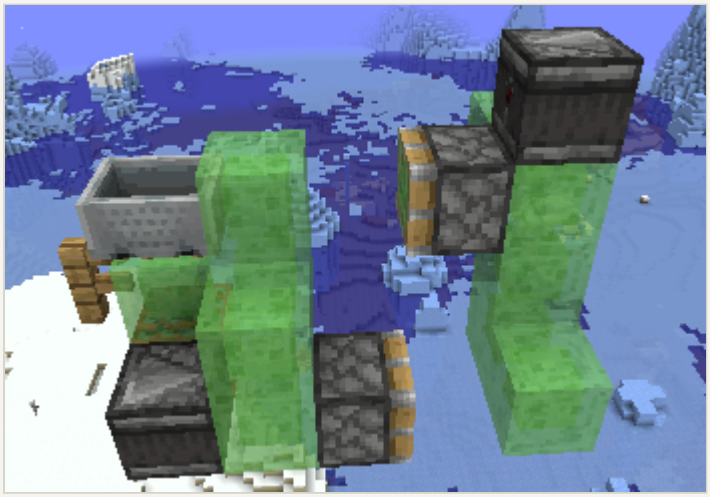
The operation is no different from the above flying machine, except the period changes from 9gt to 10gt.

Since the Observer signal is relatively short, it can make the sticky piston only push out without pulling back. So we can replace the rear regular Piston with a sticky piston without affecting its motion.



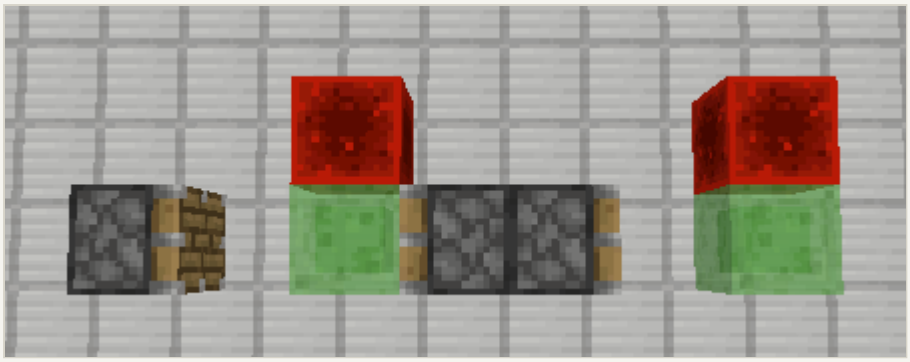
This flying machine is centrally symmetric, so if it can move right, it can also move left. This is the simplest bidirectional flying machine.

Almost all simple flying machines in Slime Tech are 10gt flying machines, variants of this flying machine.



The figure shows the bomber section of Mango's World Eater. You can see that this bomber simply re-arranged the pistons, Observers, and Slime Blocks - the abstract structure is the same

2.3 c) 12gt Piston BUD Flying Machine



In the figure, the three pistons from left to right are: regular Piston, sticky piston, regular Piston

Before version 1.11 when Observers didn't exist, this type of flying machine was popular, but it's basically no longer used. If you have spare time, try researching on your own how to start it and how it operates.

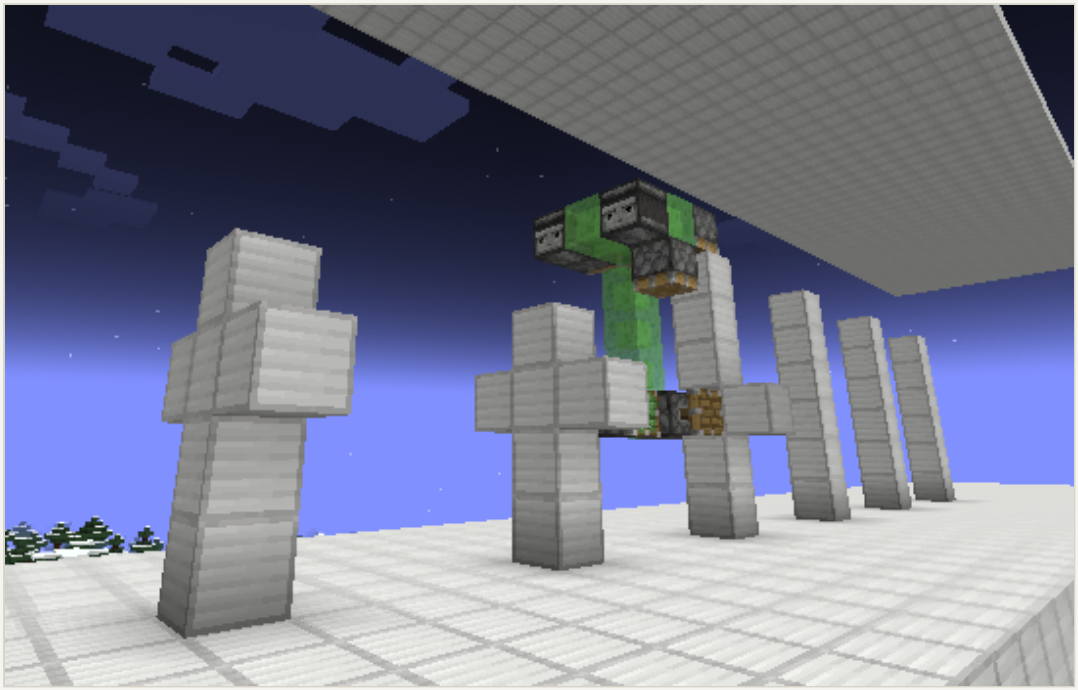
3 Mounting and Extension

Since the Slime Tech community has never had unified standards, only some conventional terms, I'll categorize things using my own naming conventions.

3.1 a) Mounting

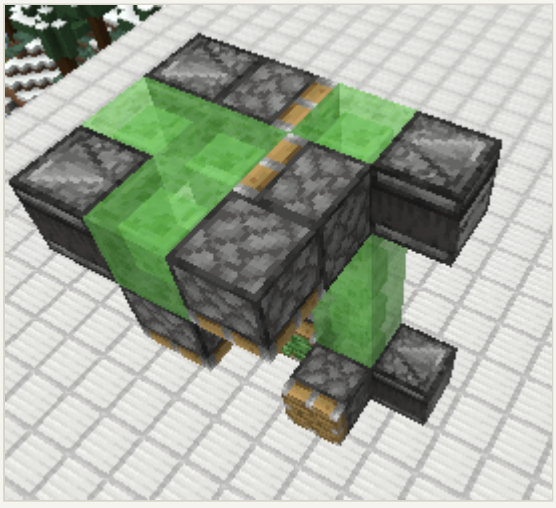
For the above 10gt bidirectional flying machine, we can change the structure's shape and then attach some other blocks as mounting. For example, the "bomber section of Mango's World Eater" above was created by attaching blocks to a 10gt flying machine as mounting.

Similarly, mounting can also attach additional Observers and pistons, giving your flying machine the ability to interact with blocks. We can implement some interesting functions this way.

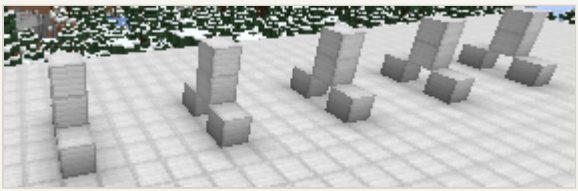


The figure shows a flying machine called "Grave Marker" that can convert 6-high pillars into a specific shape

The principle is to mount several pistons in specific directions on a 10gt simple flying machine. This doesn't really involve flying machine knowledge, so please try it yourself.



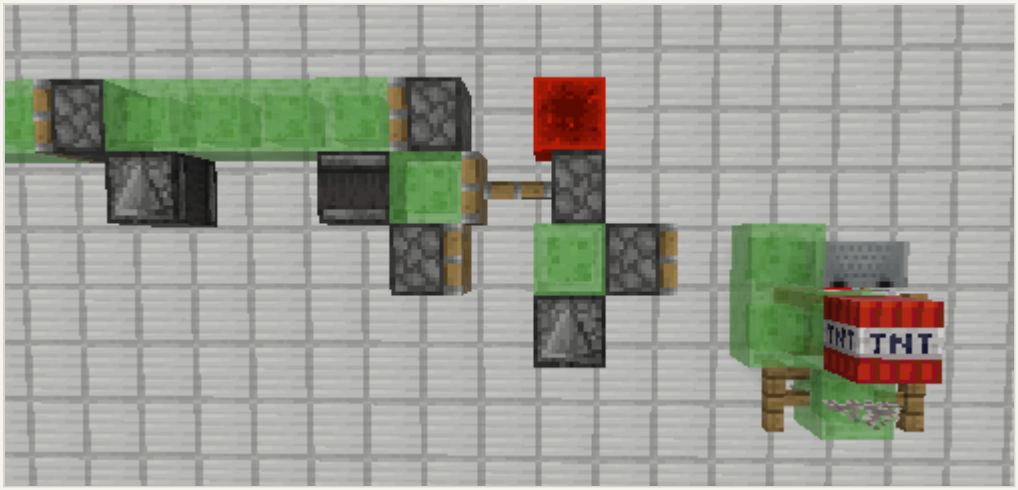
Homework: Please design a machine yourself that converts 4-high pillars into the following statue



3.2 b) Extension

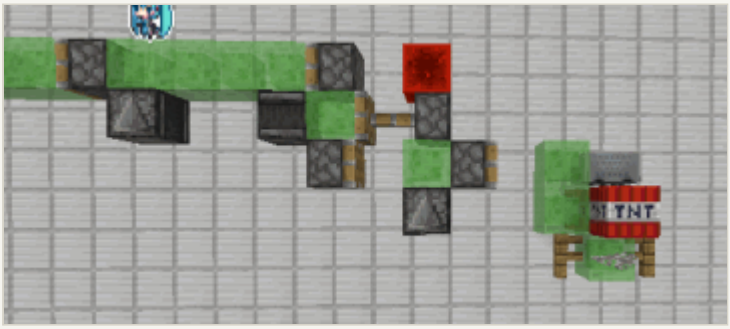
Since we can mount Observers and pistons for additional functions, the 12-block push limit of a single piston still imposes limitations. Therefore, we can extend the structure through pistons.

Based on a simple flying machine, we can push forward structures using forward-facing regular Pistons, or pull back rear structures using backward-facing sticky pistons, as shown:



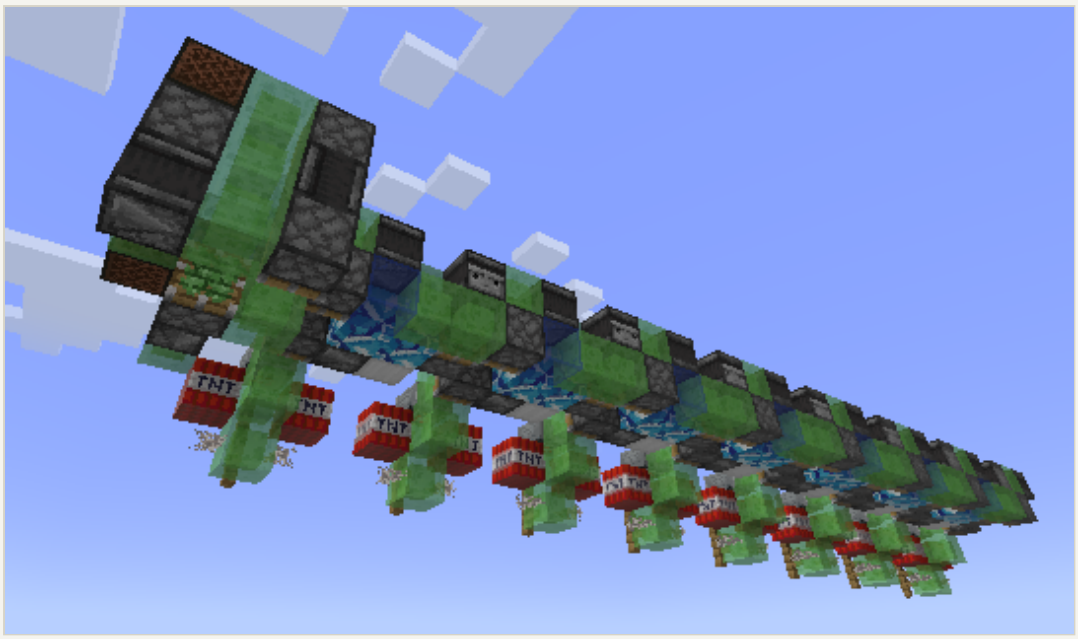
In the figure, from left to right, from top to bottom, they are: sticky piston, sticky piston, regular Piston, sticky piston, regular Piston

This flying machine, based on a simple flying machine, extends forward with a 12-block dual TNT duplicator, and extends backward off the screen.



▶ This figure is animated. View original

From now on, simple flying machines will be called "engines", and the concept of "mounting" also includes "extension". With this understanding, you've already stepped through the door of flying machines. The knowledge that follows, broadly speaking, covers various kinds of engines, various kinds of mounting and extension, and how to elegantly integrate multiple flying machines together. Keep it up!



The figure shows my unidirectional bulldozer

8.7 Engines

> "With these under your belt, you've already stepped into the world of flying machines. The knowledge that follows, broadly speaking, covers various engines, various mounts and extensions, and how to elegantly integrate multiple flying machines together. Keep it up!"
>
> --- T's Slime Tech Tutorial Chapter 1 Section 3

In this chapter, we'll expand the definition of engines, no longer limiting ourselves to the few simple flying machines from the previous section. We'll take a deep dive into what makes engines tick, with the goal of going from mounting functional structures onto engines to making the functional structures themselves into engines.

"Agency" might sound like a plain, unsexy word, but it's the perfect way to describe engines. In my opinion, anything with agency is an engine. It can be a small core, or it can be an entire assembly. Whether it has agency is what determines if it's an engine.

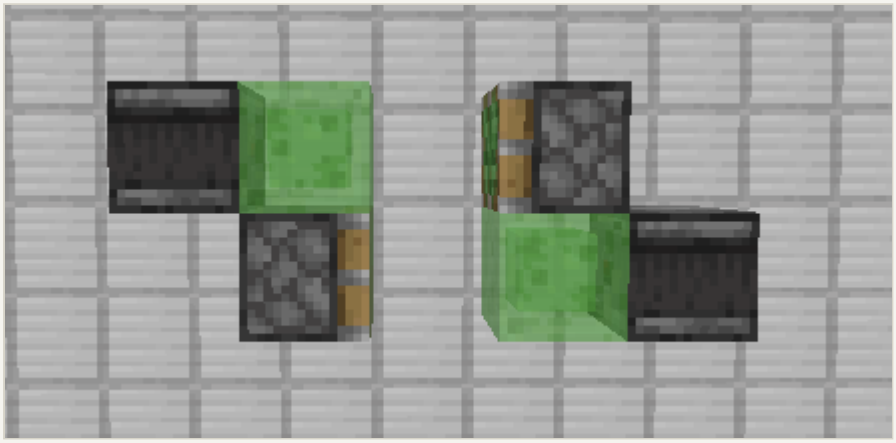
Of course, dwelling on definitions is boring. Go ahead and start learning, and I wish you a pleasant study.

8.8 Mobility

In this section, our goal is to *extend "mobility" into structures* and then *make those structures mobile*. In other words, instead of mounting things onto engines, we'll make functional structures fly directly.

1 Basic Mobility Structures

In the previous section, we explained the principles of mobility and provided the basic 10gt flying machine. From the 10gt flying machine, we can extract the following mobility structure:



The necessary structures are:

- Two opposing pistons, at least one being a sticky piston
- Two observers, each activating one piston
- Slime blocks for connection

To extend a functional structure into a flying machine, simply fill in these three necessary components.

Since functional structures almost always contain slime blocks (though occasionally they don't), prioritize using existing slime blocks. It's best not to introduce new ones.

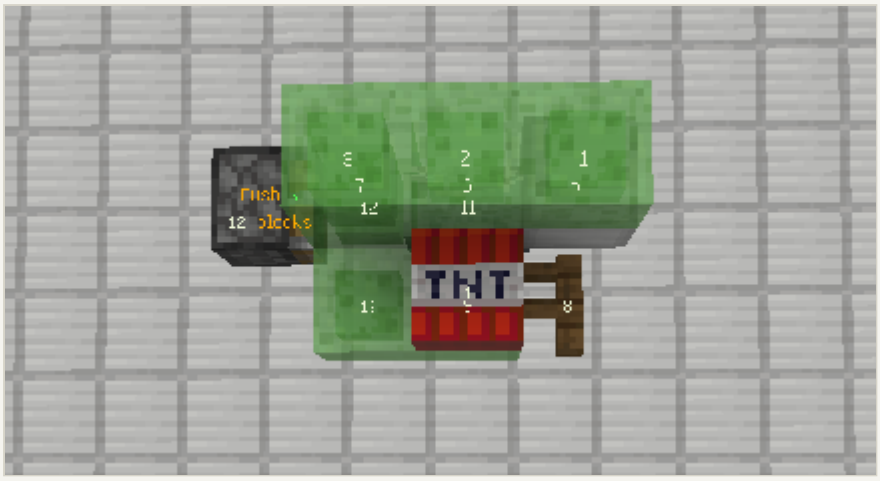
Next, if the functional structure contains observer-piston structures, prioritize using the existing observers. That is, add pistons to positions where observers can activate them. It's best not to introduce new observers.

Finally, place the pistons. Be careful to avoid exceeding the push limit and ensure all structures can be connected and moved.

2 Examples

What follows is practice. This section can't really be broken down further—it's just practice makes perfect. Also, the principles above aren't set in stone. Some players prefer introducing new slime blocks or observers, which can actually make the whole machine cleaner and more elegant.

2.1 a) Several TNT Duplicators

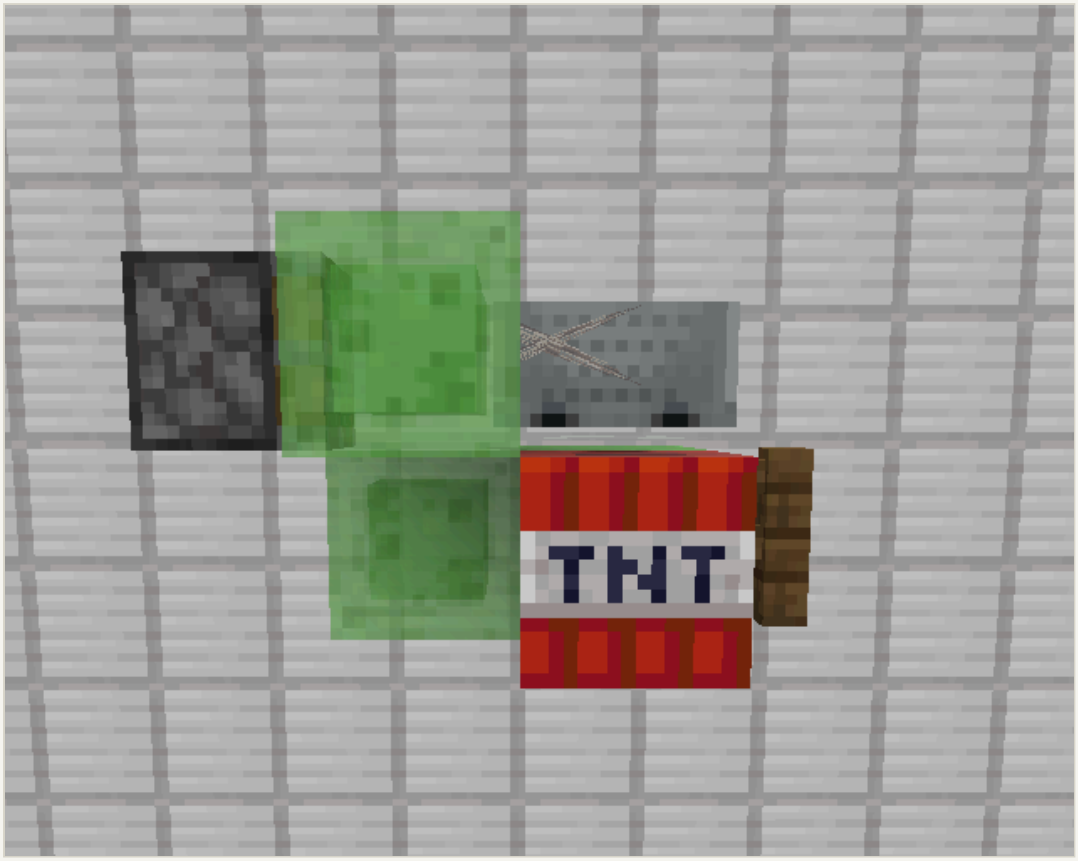


Pictured: Halwei's landing correction bidirectional TNT duplicator

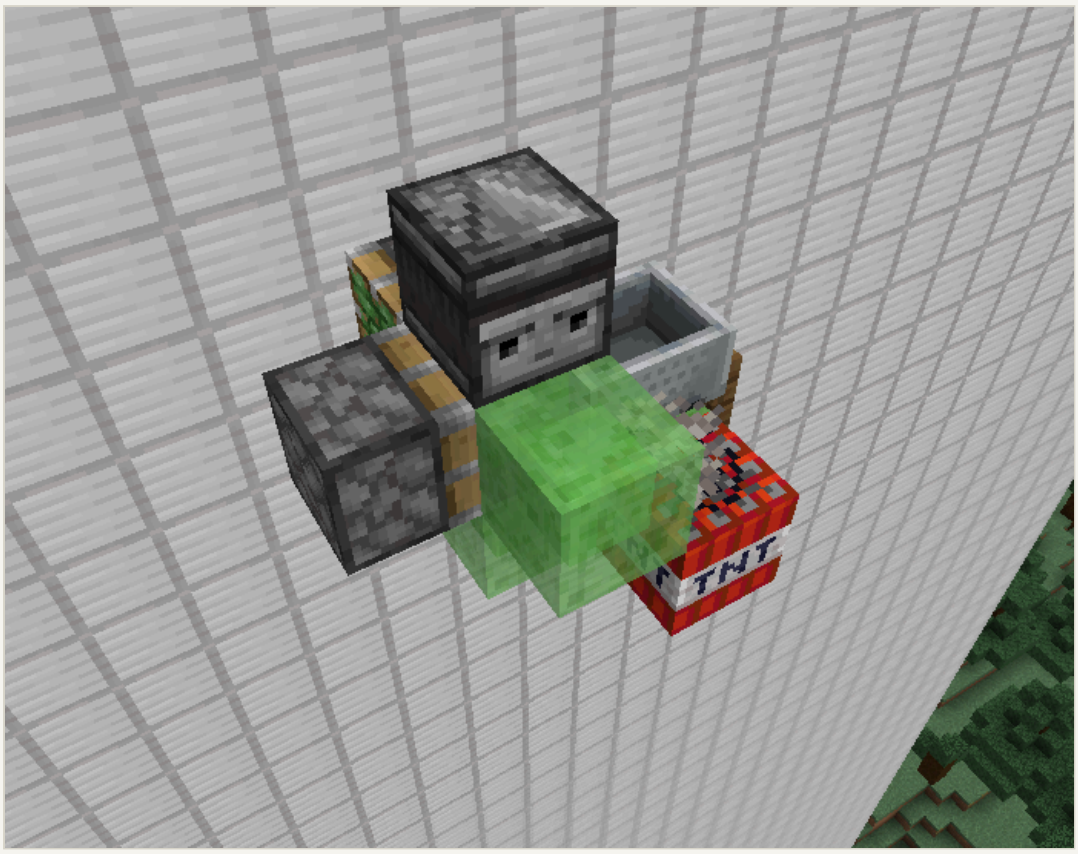
This TNT duplicator has reached 12b and cannot have any internal structure modified. It can only be mounted using a flying machine and cannot move directly.



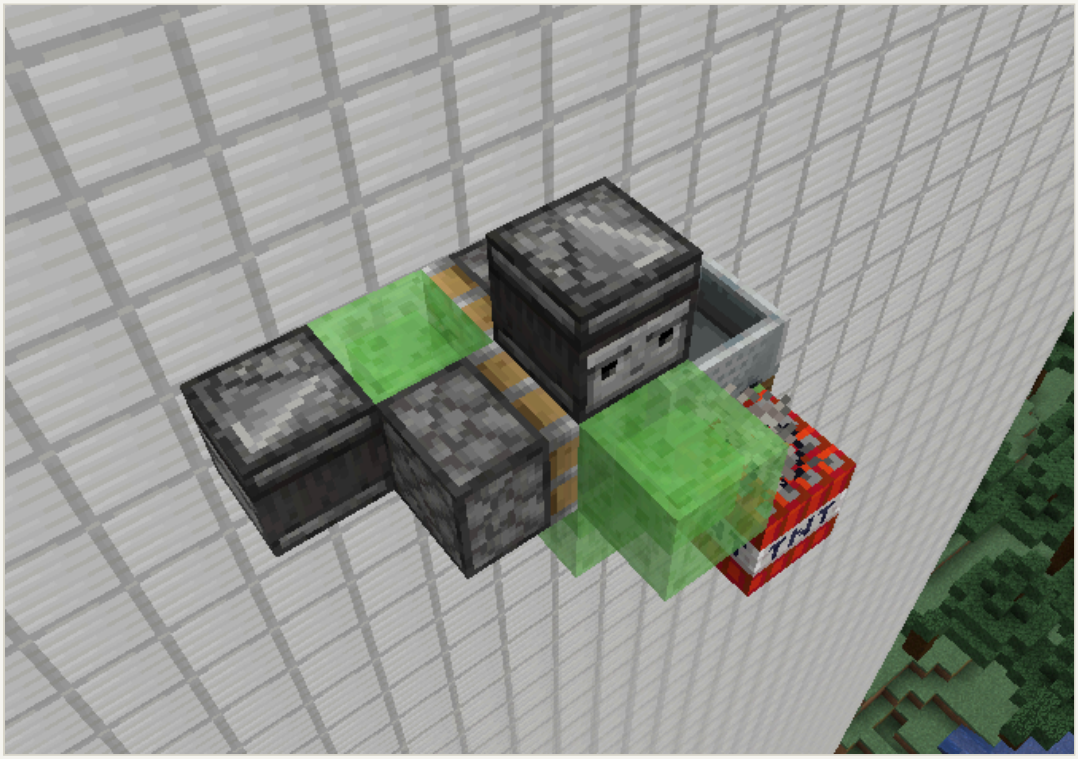
Similar structures include the high-firepower bidirectional TNT duplicator shown in the figure.



For the simple TNT duplication structure shown, we can use the slime block directly facing the piston as the slime block mentioned in the third point above. That is, both the piston and observer connect to this position, as shown.



This way, we complete the other part with slime blocks and observers.

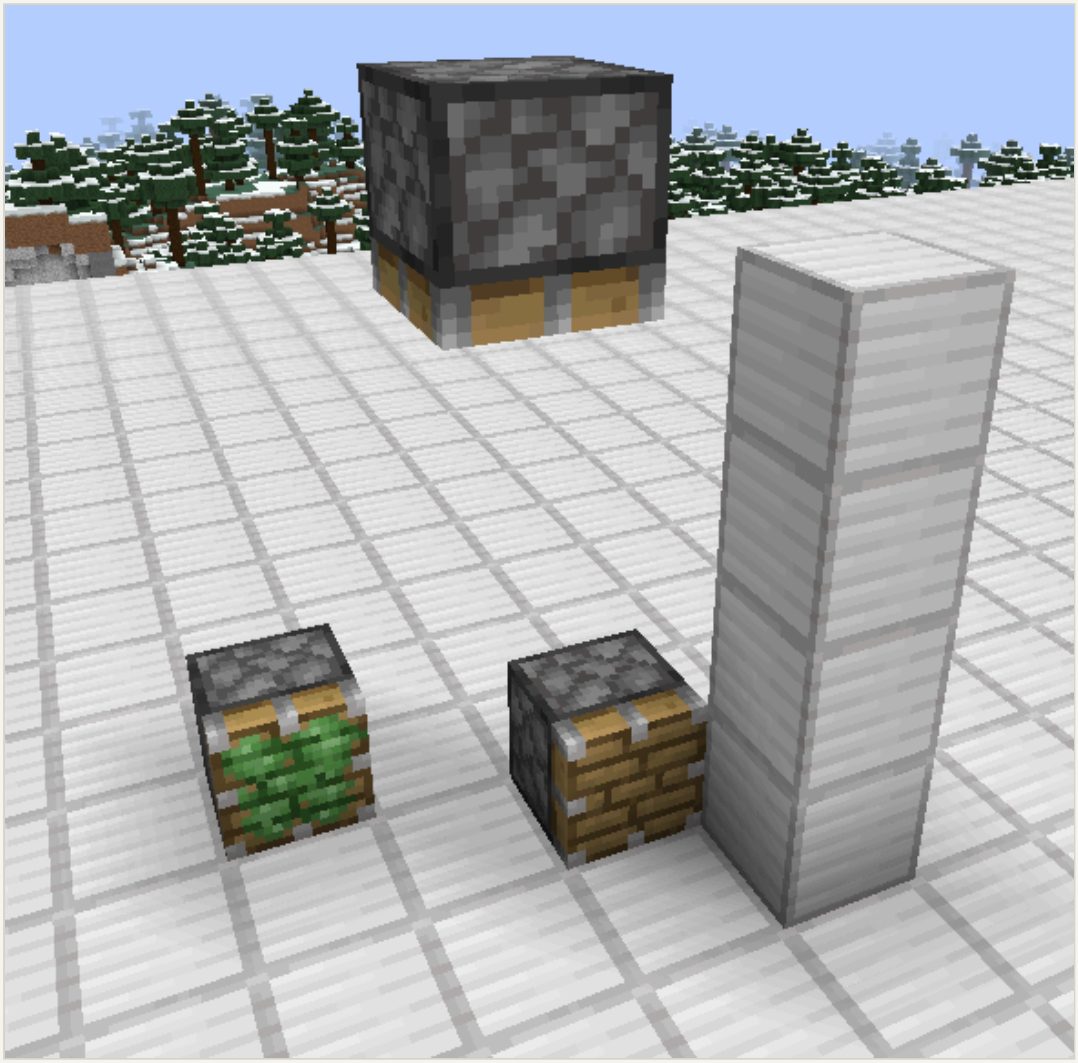


Now the simple TNT duplicator can move normally.

In practice, TNT duplicators mostly use ready-made structures, such as the bomber module from ilmango's world eater in the previous section.

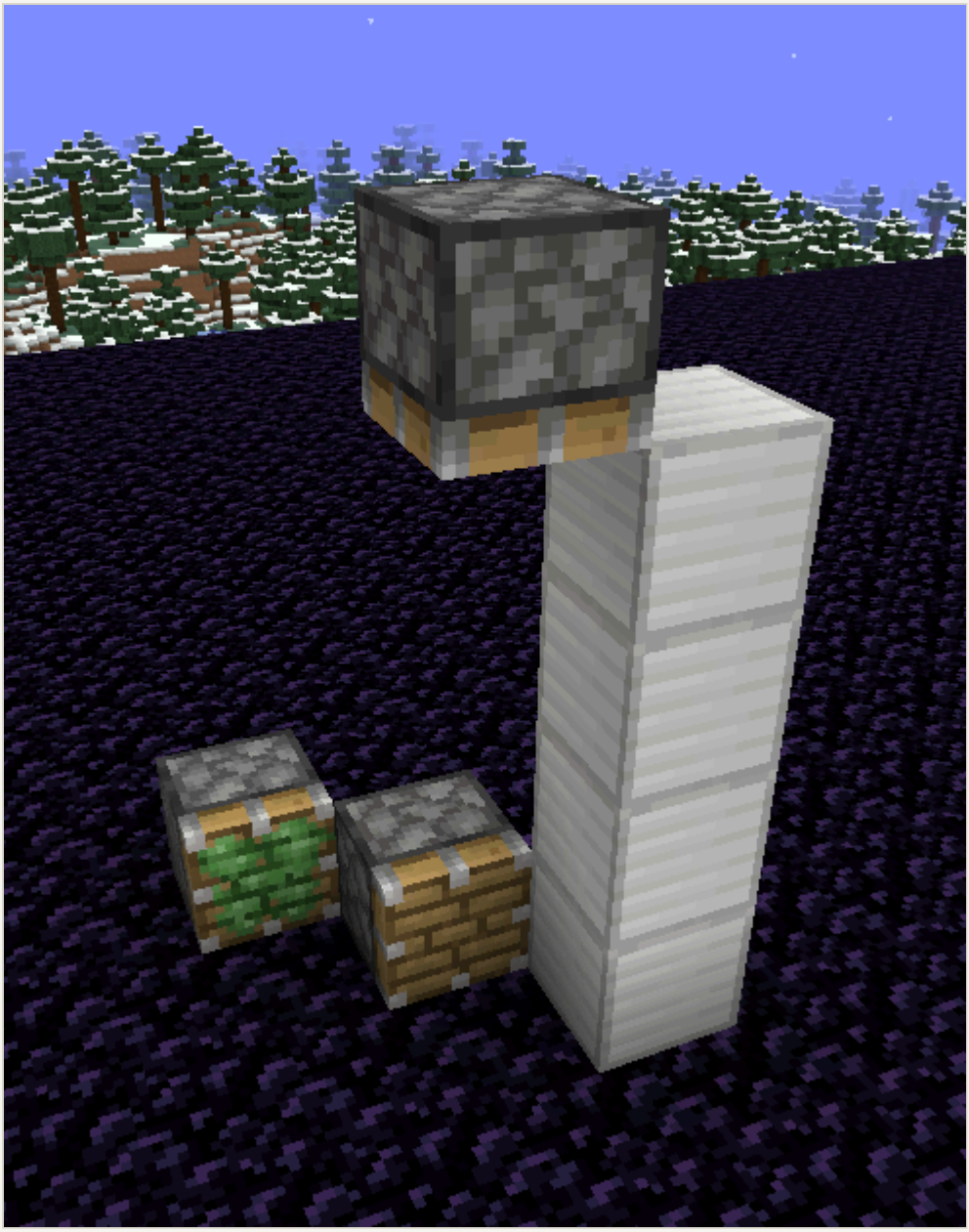
2.2 b) Known Piston Structures

For the problem left from the previous section, we can arrange the piston structure like this—just move the pistons in sequence:

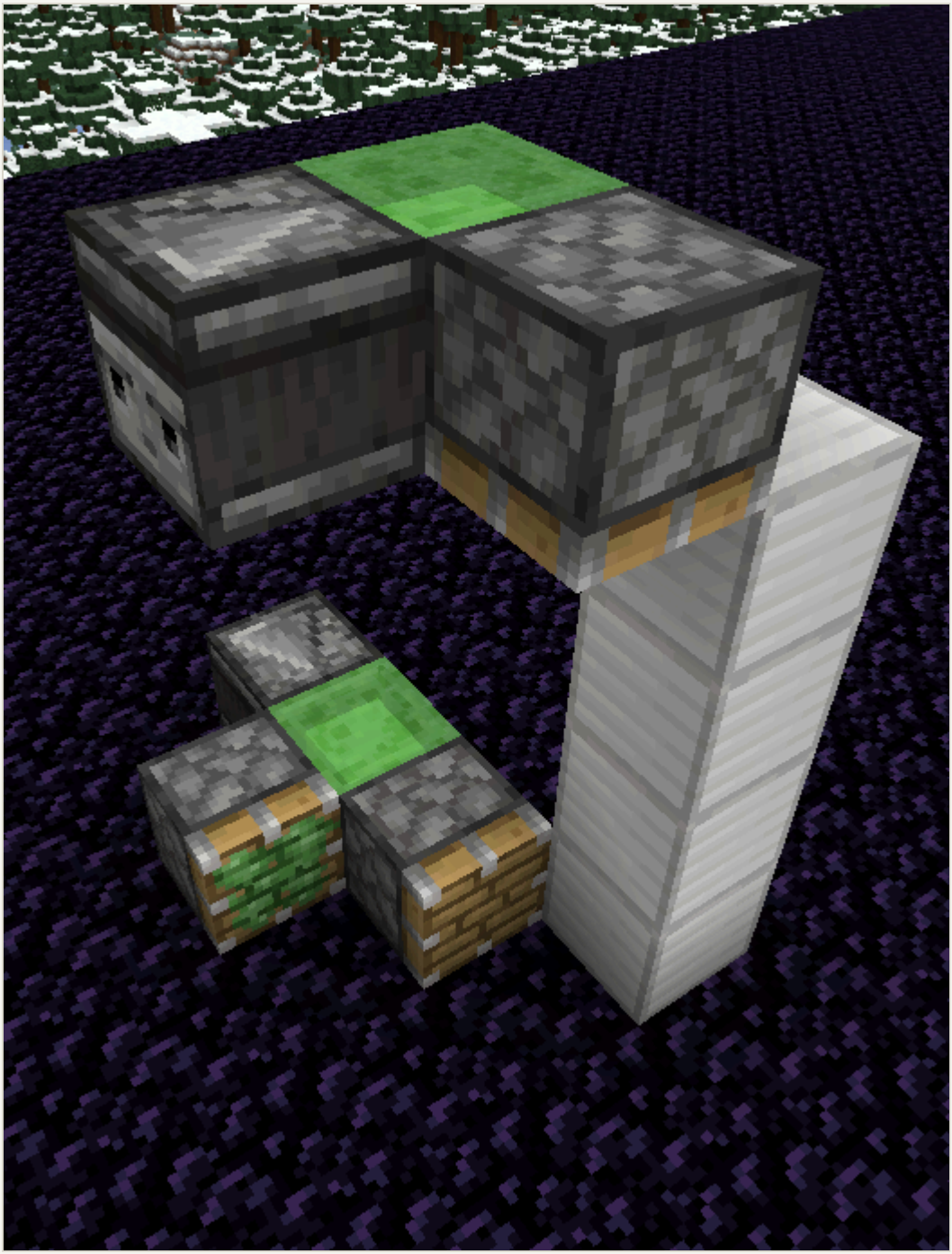


Due to unavoidable sticking issues, this is tricky for beginners to solve. Here we replace the floor with obsidian and set aside the floor sticking problem for now.

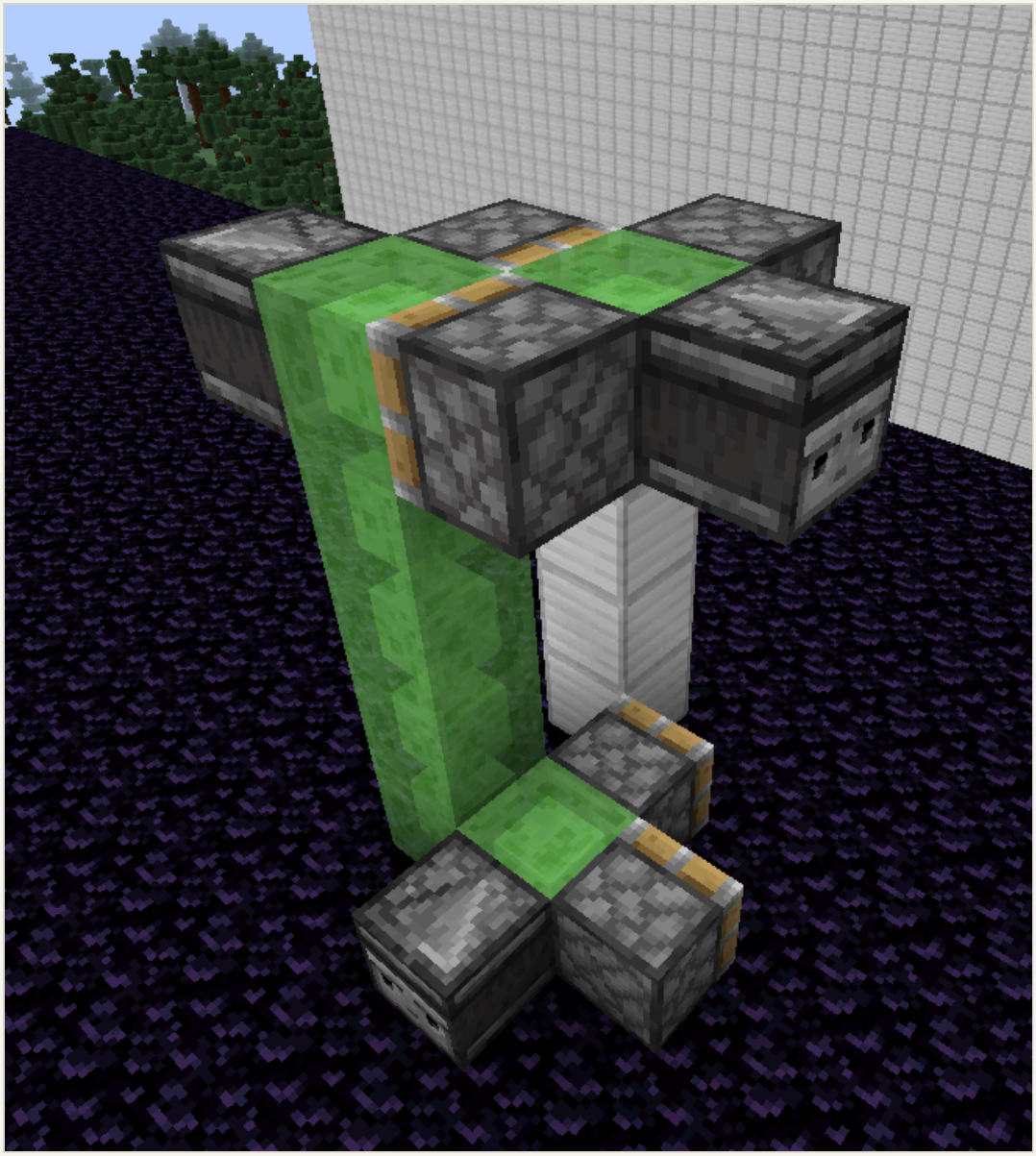
Compress the pistons. At this point, we need to make sure the lower part moves one block first, then the upper part moves one block, then the lower part moves again—this maps exactly to the two parts of the flying machine.



Connect the slime blocks and observers.



Next, complete the flying machine as follows:



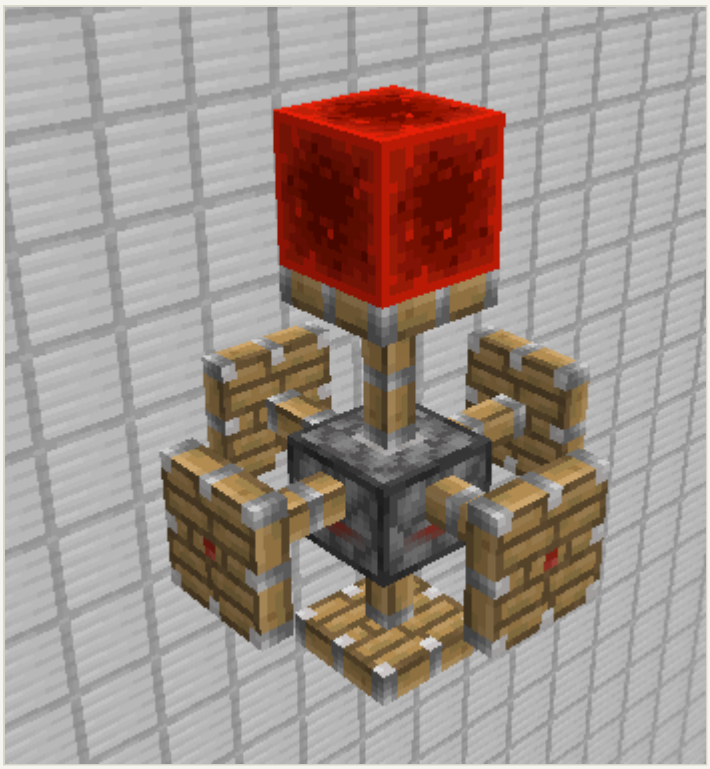
As shown, we achieved reuse of slime blocks and observers in the upper structure. The lower part can't be connected due to distance.

Next, back up and test (even though the structure is simple and backup isn't that useful, it's still good to build the habit).

2.3 c) Other Structures

Structures vary wildly—there's really no unified standard to measure them by. So we won't go into detail from here on. Just try to satisfy the three points above.

Similarly, the various priorities aren't mandatory either. Heck, the engine doesn't even have to be 10gt. We just need to satisfy the principles of `1.2.1`.

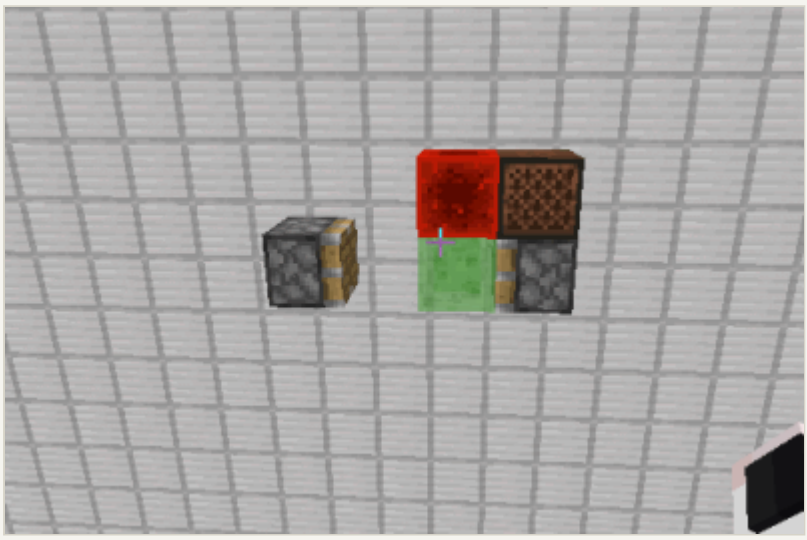


8.9 Semi-Dynamic Engines and Extensions

In this section, we explore two special flying machine engine designs that rely on other independent engines to pass motion forward.

1 Semi-Dynamic Engines

A `semi-dynamic engine` is a counting engine that moves one block when it receives an external signal. As shown in the figure:



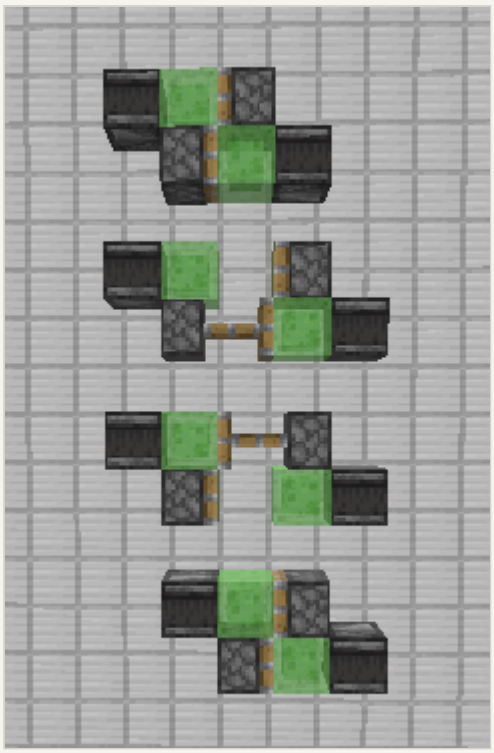
▶ This figure is animated. View original

The semi-dynamic engine shown above is simple and widely used, but it doesn't work properly in the now-common 10gt standard engine. So let's explore how to get a flying machine from `can't move`, to `always moving`, to `moves once`.

As mentioned in the previous section, the essential structure of a flying machine is:

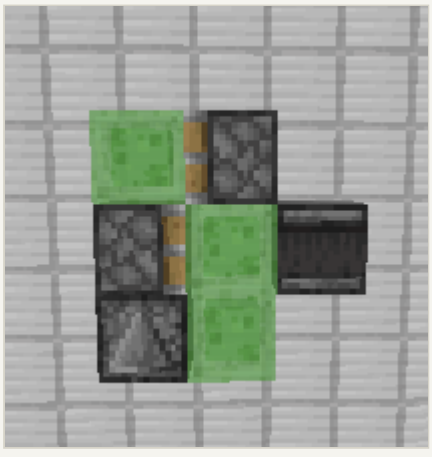
- Two pistons facing each other, at least one being a sticky piston
- Two observers, each activating one piston
- Slime Blocks for connection

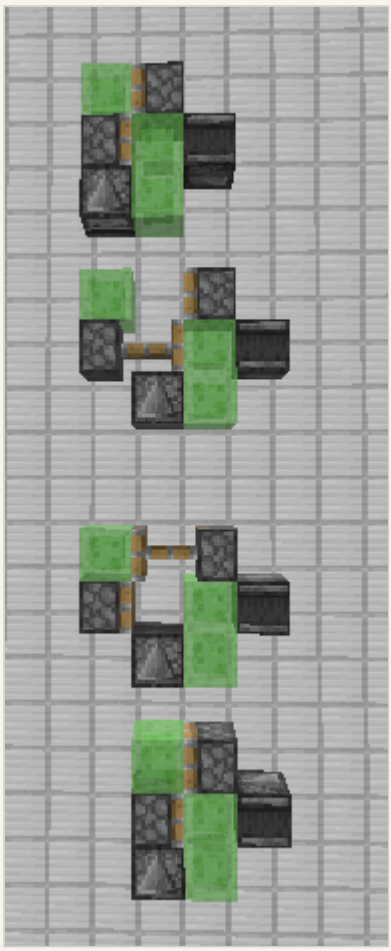
The two observers are what make `always moving` possible, since they provide the exact timing needed for continuous motion.



Analyzing the 10gt flying machine, we can see that the next movement happens because the left observer arrives in position and sends another signal, pushing the whole machine into the next cycle. To make the flying machine move only once, we just need to stop the observer from sending that signal when it reaches position.

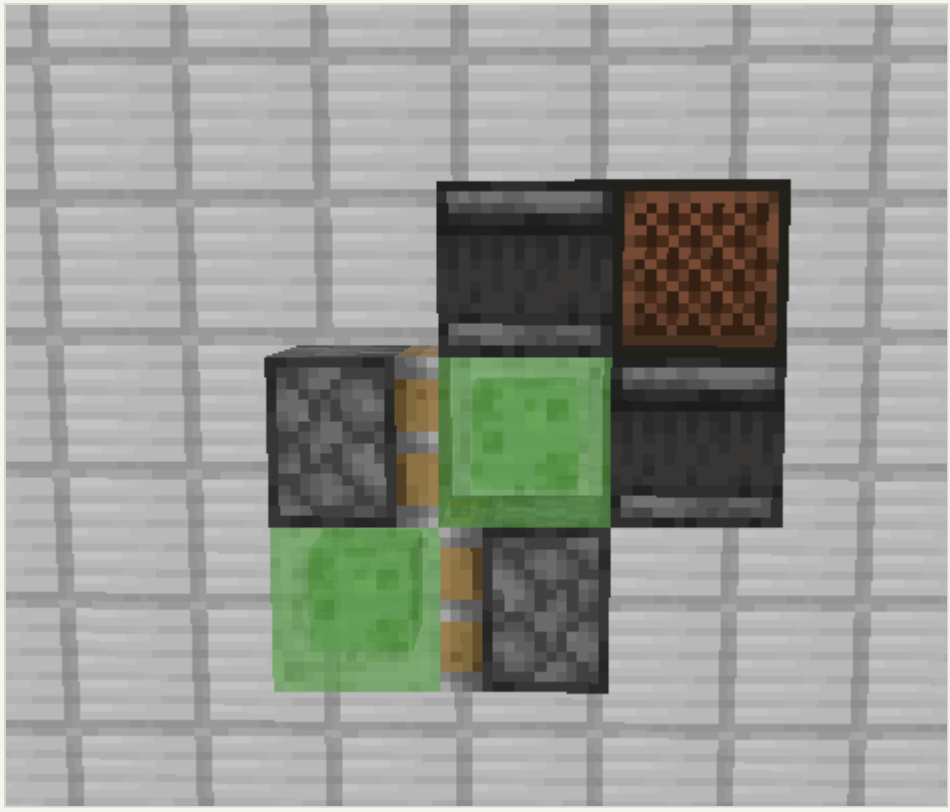
As shown in the figure, we mount the left observer onto the right structure. This way, at startup the observer can send a signal to the piston, but when it arrives in position, the observer and piston separate, preventing the next cycle from starting.





Note: the left piston in the figure cannot be a sticky piston. Due to signal timing, swapping it for a sticky piston would cause the structure to oscillate.

Let's simplify the semi-dynamic engine structure above and add a Note Block as a switch:



From this, we can summarize the key components of a simple semi-dynamic engine:

- Two pistons facing each other - the one pointing in the direction of travel is a regular piston, while the one pointing away is a sticky piston
- Two observers that can activate both pistons, but mounted on the same side
- Slime Blocks for connection

In fact, most tunnel bores and unidirectional bulldozers are essentially large semi-dynamic engines. They receive a signal from the player, move forward one block, reset, and wait for the next signal.



The figure shows my unidirectional bulldozer with the semi-dynamic switch isolated.

2 Extensions

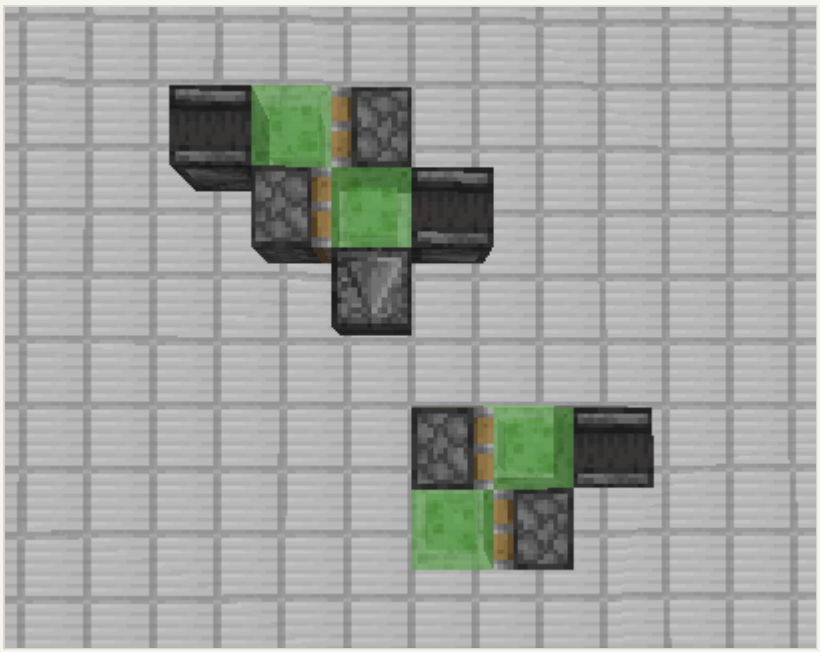
We covered flying machine mounting and extensions in `1.3.3`, so here we won't draw a distinction between the two.

2.1 a) Motion Extensions

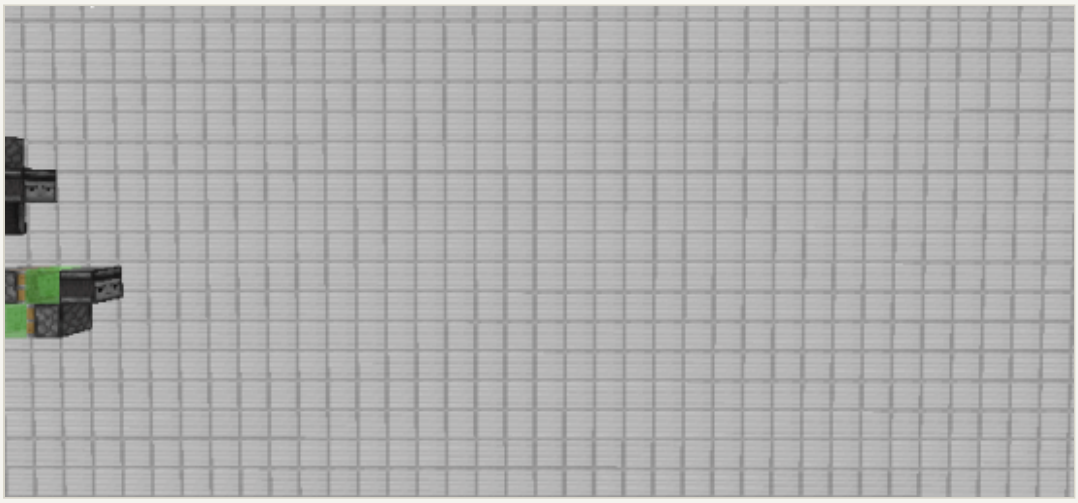
The simplest extension is just a forward-facing regular piston and a backward-facing sticky piston. Using observers and pistons, motion gets passed forward. We won't go into details here.

2.2 b) Signal Extensions

Building on the semi-dynamic engine above, we can use a flying machine to send it periodic signals. The semi-dynamic engine then moves in sync with the flying machine.



In the figure, the top structure is a regular 10gt flying machine with an extra observer mounted on it, and the bottom is a semi-dynamic engine. They move in sync.

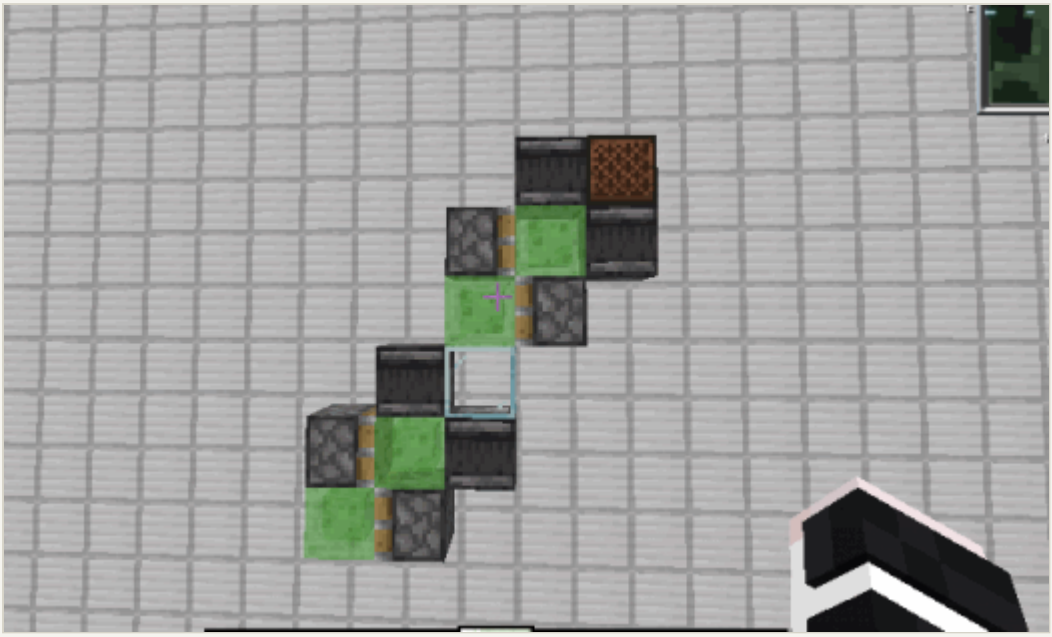


▶ This figure is animated. View original

You might be wondering: if a semi-dynamic engine's whole point is to move only once, but we're connecting it to a flying machine, doesn't that defeat the purpose? Why not just build another flying machine instead?

Actually, a semi-dynamic engine is a pretty simple extension structure. For one thing, the main flying machine might hit its push limit and can no longer pass motion through observers and pistons - in that case, you can use a lighter semi-dynamic engine instead. On the other hand, a semi-dynamic engine doesn't just `move once` *- it* `stops after moving`*. That means if the main flying machine stops for whatever reason, the semi-dynamic engine extension must immediately stop too. If you built a separate flying machine instead, you'd run into serious problems.*

There are also broader signal extensions where the signal isn't limited to just Redstone.



▶ This figure is animated. View original

To make this easier to see, I also made the main flying machine into a semi-dynamic engine. When it moves, the semi-dynamic engine extension detects the movement and follows along.

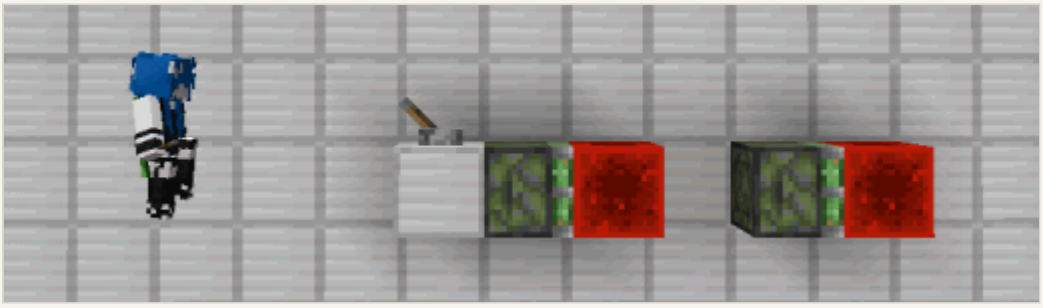
8.10 Linkages

Moving a big structure all at once is a real headache. Since pistons can only push 12 blocks, we have to chain multiple pistons to pass "motion" along. The catch is that you need to carefully tune the delays, and the longer the chain, the trickier the timing gets.

Therefore, we have the concept of "zero-delay linkages". In this section, we will introduce several types of linkages based on different principles.

1 Unidirectional Linkages Based on Piston Retraction

Sticky piston chain retraction has zero delay.



▶ This figure is animated. View original

In fact, sticky pistons respond instantly whether extending or retracting. The moment they act, the pushed or pulled blocks convert to `b36`, then 2gt later when the motion finishes, they convert back to their actual block type.

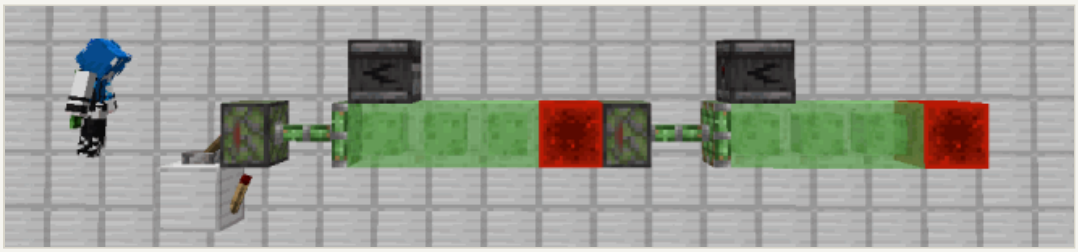
> **Supplement: Detailed Description of Piston Transmission Time**
>
> If initially all pistons are in the retracted state, and the first piston can be activated by a lever, pulling the lever at tick 0, the timing is as follows:

TICK	PHASE	EVENT
0	AT	Lever pulled
1	BE	Piston 1 extends
3	TE	Redstone Block 1 in place
4	BE	Piston 2 extends
6	TE	Redstone Block 2 in place
7	BE	Piston 3 extends
9	TE	Redstone Block 3 in place
…	…	…

However, with piston chains during extension, the next piston only activates after the block has moved and settled into place, so each piston in the chain adds some extra time.

But retraction works completely differently. The instant the Redstone Block becomes `b36`, the signal disappears, and the previous piston responds immediately. That's why piston chain retraction has zero delay.

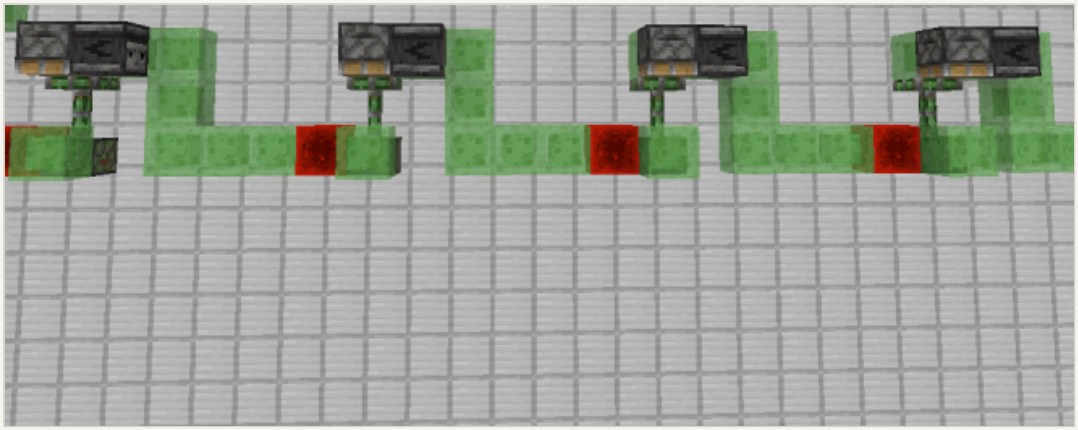
Let's modify this to make the pistons actively extend.



▶ This figure is animated. View original

As shown, we make the pistons automatically extend after retracting, which keeps the entire piston chain in sync.

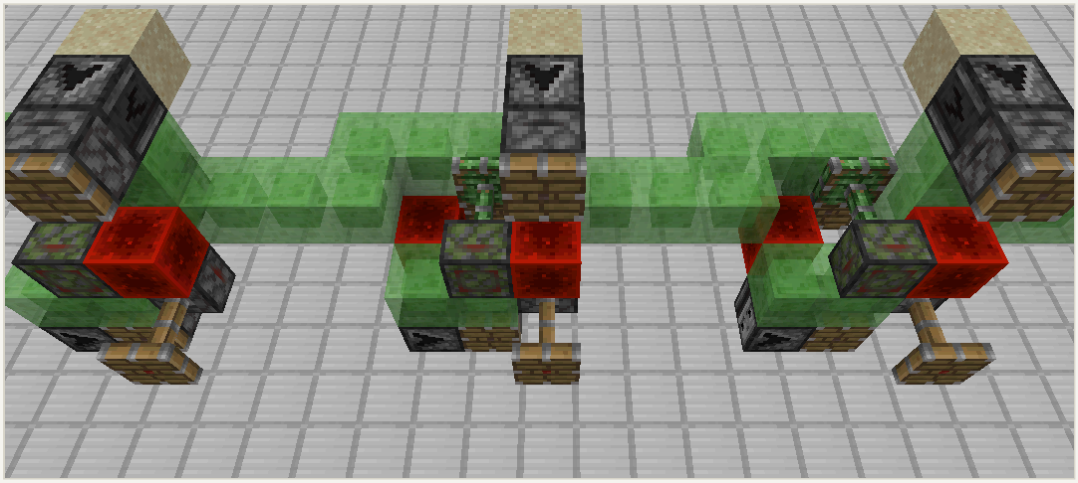
Based on this principle, we can apply it to flying machines, specifically the "semi-automatic machine" mentioned earlier:



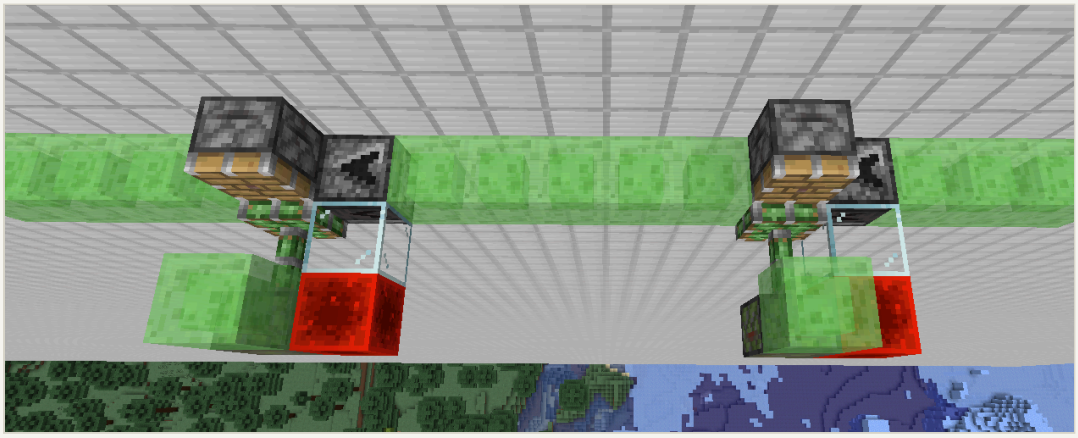
▶ This figure is animated. View original

In the diagram, each structural unit uses an upward-facing sticky piston for zero-delay signal transmission, along with a downward-facing regular piston to reset the sticky piston.

This is the most common linkage structure. Similarly, we can create many linkages with the same structure.



The image shows the docking frame on one side of `ilmango`'s world eater bomber, which is essentially a linkage with several Observers and Pistons attached

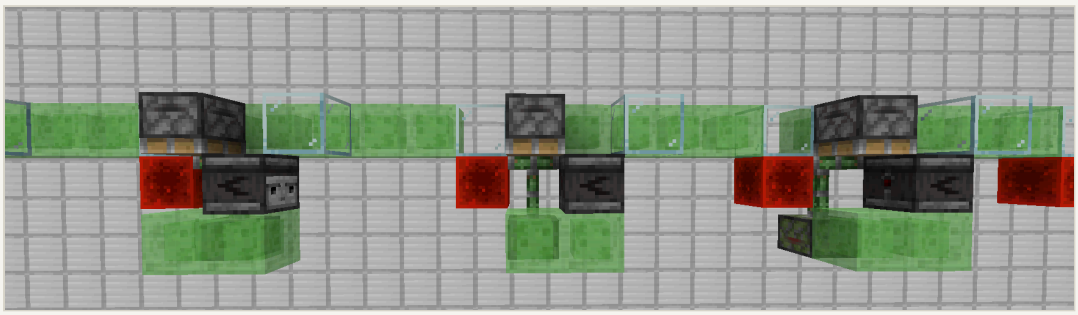


The image shows a linkage from `星空花夜、`'s burst tunnel bore, which moves horizontally but operates on the same principle

2 Extensions of Piston Retraction-Based Linkages

2.1 a) ???

The first time I saw this structure was in `Eular`'s narrow trench world eater, where he used this linkage:

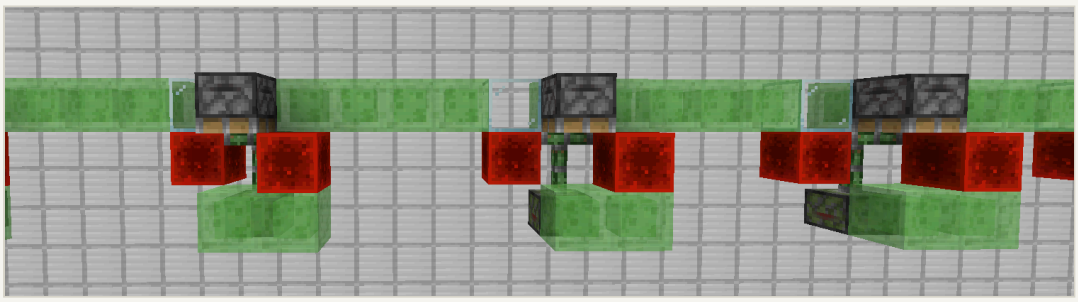


In this linkage, the sticky piston is activated by the Redstone Block at the BUD position—meaning it "loses power" the instant it retracts and the Redstone Block becomes `b36`. However, once the Redstone Block settles into place, it activates the piston again, preventing it from going down.

Therefore, this structure must push the whole thing down during the brief window after the piston retracts but before it extends again.

Note the Observer position in this structure—it's at the bottom, and it doesn't move during the sticky piston retraction phase. This means that once the regular piston above and the glass (used to update the Observer) are in place, it can fire off a signal immediately without any waiting.

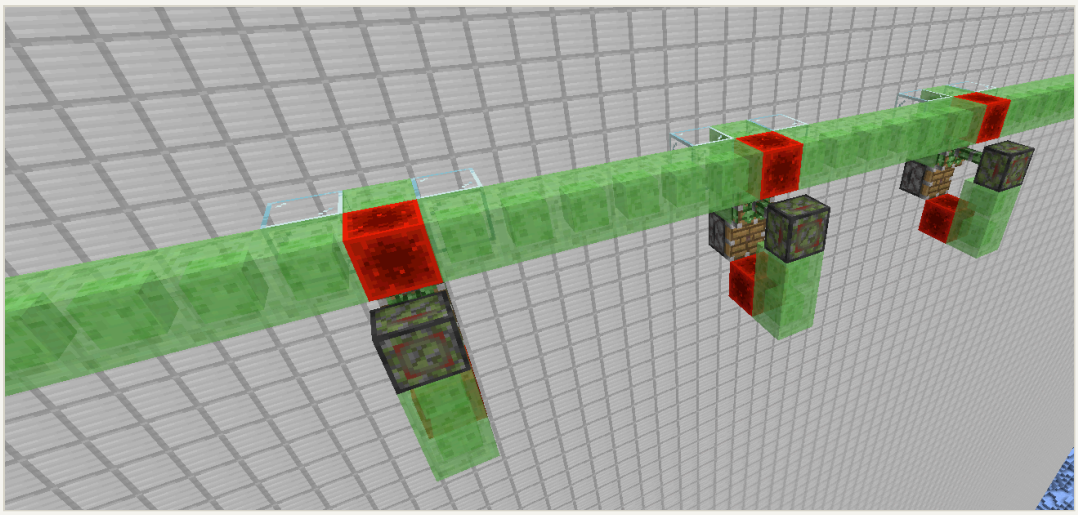
This way, the regular piston immediately pushes the structure down, perfectly catching the timing window.



The Observer above can also be replaced with a Redstone Block, as long as the regular piston can push down immediately without waiting.

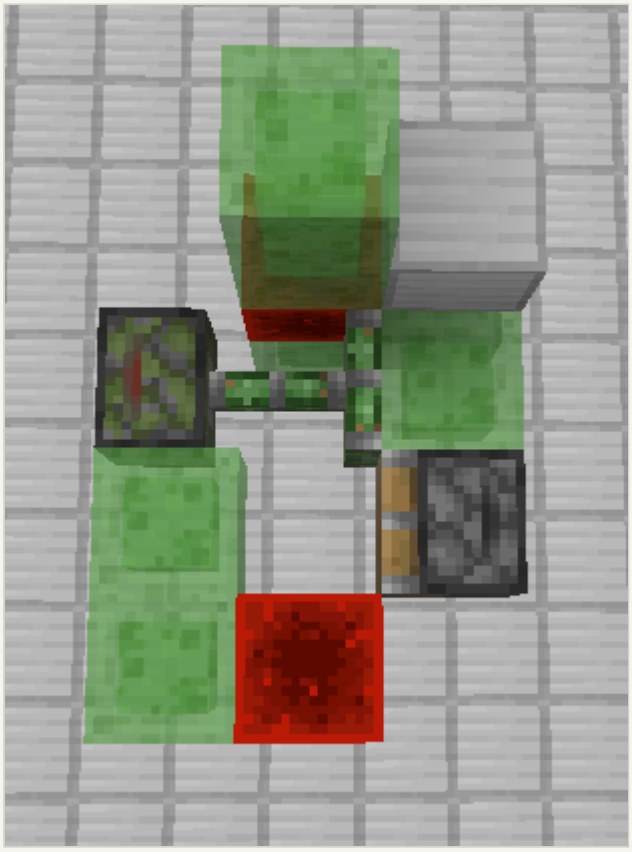
2.2 b) Bidirectional Linkages

The special linkage above was preparation for this one, from `comet107`'s expandable tunnel bore main linkage.

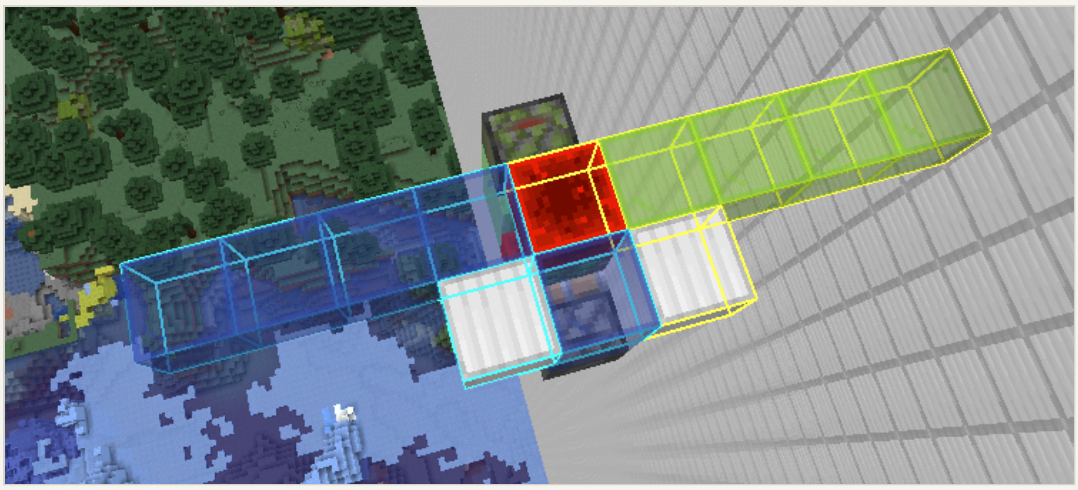


The linkage above can only be activated from one end, but this linkage can be activated from any position to make the entire structure advance.

For convenience, let's remove one unit to study it.



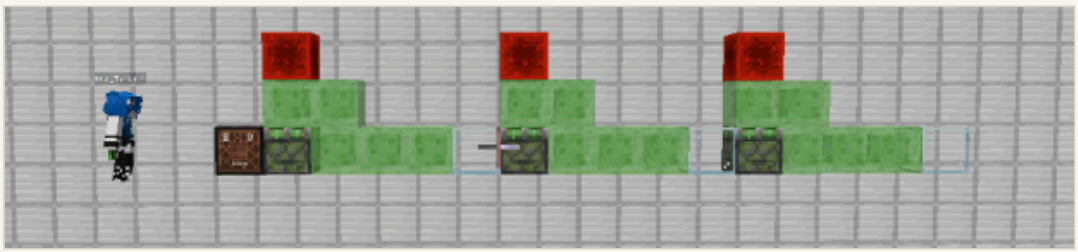
This unit advances using the same principle as `3.1.2 a)`. It connects symmetrically to the Redstone Blocks above on both sides, either of which can activate this structure. At the same time, the structure connects symmetrically above to transmit motion downward.



I've used different colors to distinguish the two parts. The green part moves first, taking the Redstone Block with it, and the blue part moves later, passing the signal downward (although they're simultaneous, there's a slight difference in micro-timing). The reverse works the same way.

3 BUD-Based Linkages

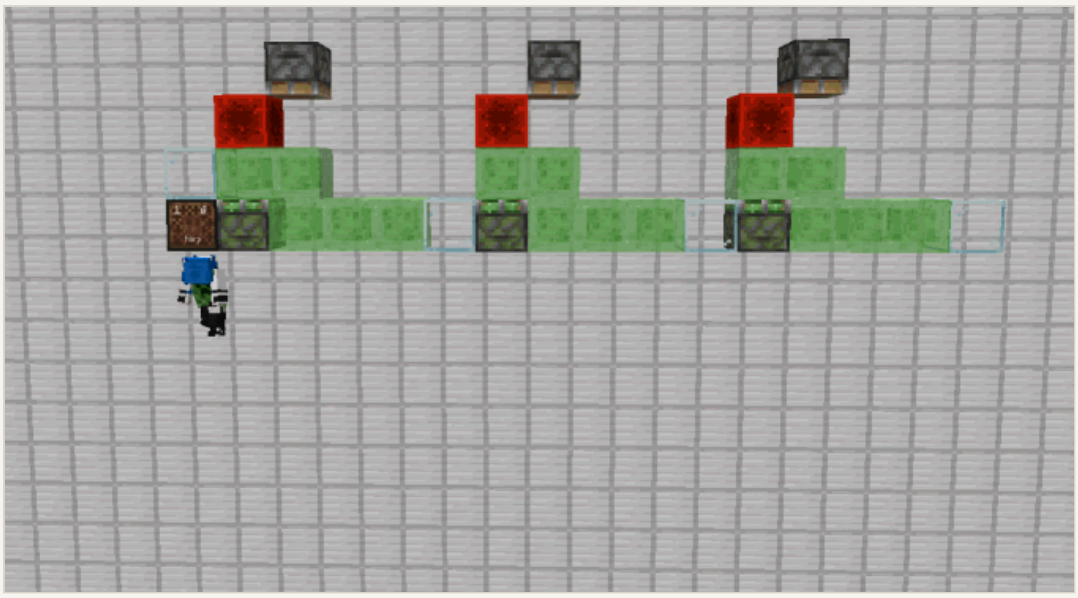
There's another common structure, as shown below.



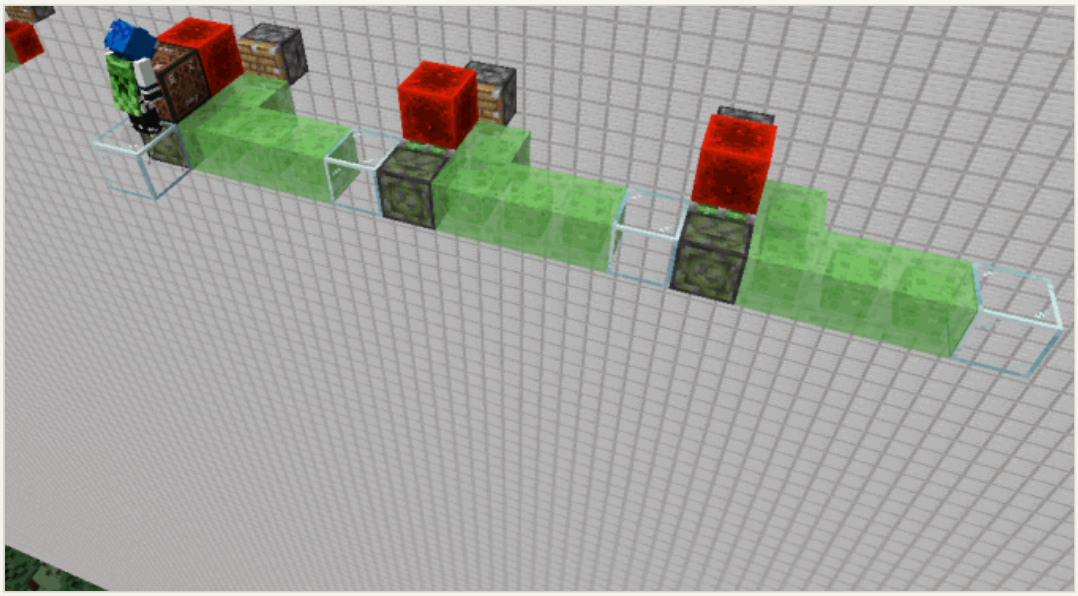
▶ This figure is animated. View original

A BUD-state piston moves when it receives an update, and going from stationary to moving is itself an update, so the signal transmits instantly and all modules move at the same time.

Combining this with the semi-automatic machine from `2.2.1`, we can build structures like these:



▶ This figure is animated. View original



▶ This figure is animated. View original

The second type of linkage is widely used in burst tunnel bores because if you block it at the front, the linkage can only oscillate in place and can't advance, making it useful for transmitting burst signals.

4 Other Linkages

The linkages above are all zero-delay linkages (after all, that's what "linkage" originally means). But functionally speaking, any structure that gets multiple modules moving together can be called a linkage—it doesn't have to be zero-delay.


图片加载失败请耐心等待

APPENDIX A

A

附录

IN THIS CHAPTER

A.1 Stacks

A.2 Glossary

A.1 Stacks

This chapter introduces stacks and call stacks, which will help you understand block updates.

1 Understanding Stacks

> A **stack** is a **first-in-last-out (FILO)** data structure, where elements can only be added or removed at one end.

1.1 First In Last Out

Imagine a **Pringles can**:

- Chips can only be placed in from the top, and can only be taken out from the top.
- Therefore, **the chip placed at the bottom first will be the last one that can be taken out**.

This is the "first in last out" principle.

1.2 Stack

A stack is like a Pringles can.
In data structure terminology:

- "Putting chips into the can" corresponds to **push**;
- "Taking chips from the top" corresponds to **pop**.

Additionally:

- The **top of the can** is equivalent to the **top of the stack**
- The **bottom of the can** is equivalent to the **bottom of the stack**

That's a stack.

2 Call Stacks

> A **call stack** is a mechanism that programs use at runtime to manage function calls. It relies on the first-in-last-out rule to ensure functions return correctly.

3 How It Works

Imagine you're making phone calls to ask a question:

1. You call your friend A to ask a question.

2. A doesn't know the answer, so they call their friend B to ask.
3. B isn't sure either, so they call their friend C to ask.

This time, C finds the answer and tells B first.
B then tells A.
Finally, A tells you the answer.

This follows the call stack pattern:

- Each phone call pushes a new "call task" onto the "call stack".
- When C is searching for the answer, **the task at the top of the stack** is "C searching for the answer".
- After C finds the answer, **the most recent call completes first**, then returns layer by layer to the callers (B → A → you).
- Your initial phone call was the first task pushed onto the call stack, so it sits at the **bottom of the stack**.

In a program:

- "Making a phone call" is equivalent to **calling a function**;
- "Getting an answer and passing it back" is equivalent to **a function returning a result**.

This way, when functions are nested, the program can ensure each function completes and returns to its caller correctly.

A.2 Glossary

1 Overview

This chapter explains the specialized terms and vocabulary used throughout GTMC.

COLOPHON

Graduate Texts in Minecraft — 52 articles.

Written and reviewed by the technical Minecraft community.

This edition was generated on 2026-07-13.

Articles revision: `0df2c46`

Articles are licensed under CC BY-NC-SA 4.0. Website code is licensed under Apache-2.0.

Read online, with animated figures and updates, at techmc.wiki