GRADUATE TEXTS IN MINECRAFT

An Introduction to Technical Minecraft

Locale: English

Generated: 2026-07-05

TABLE OF CONTENTS

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

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

BLOCK UPDATE

3 ARTICLES

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:

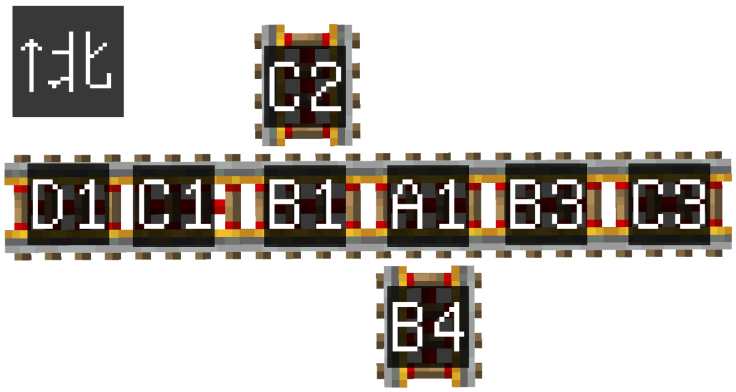
```
├── Block A emits NC update
│   ├── Block B receives NC update
│   ├── Block B changes state
│   ├── Block B emits NC update
│   │   └── Nothing happens
│   │   * Block B NC update complete *
│   └── Block B emits PP update
│       └── Nothing happens
│       * Block B PP update complete *
│   *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

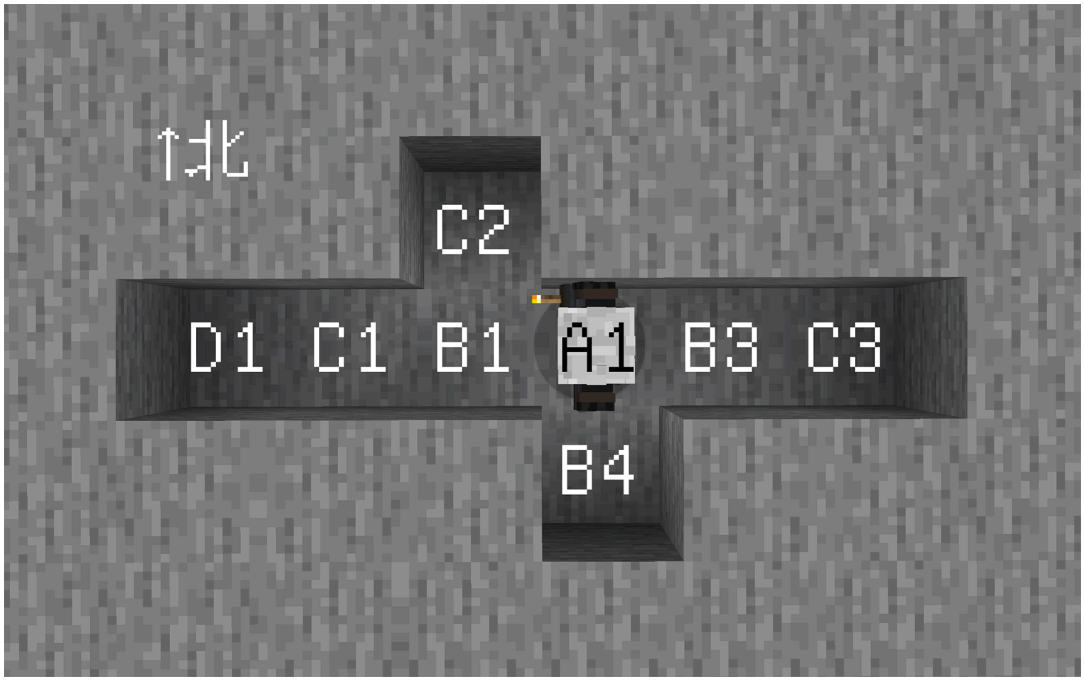
~~To understand updates, we need to first set aside updates~~

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:



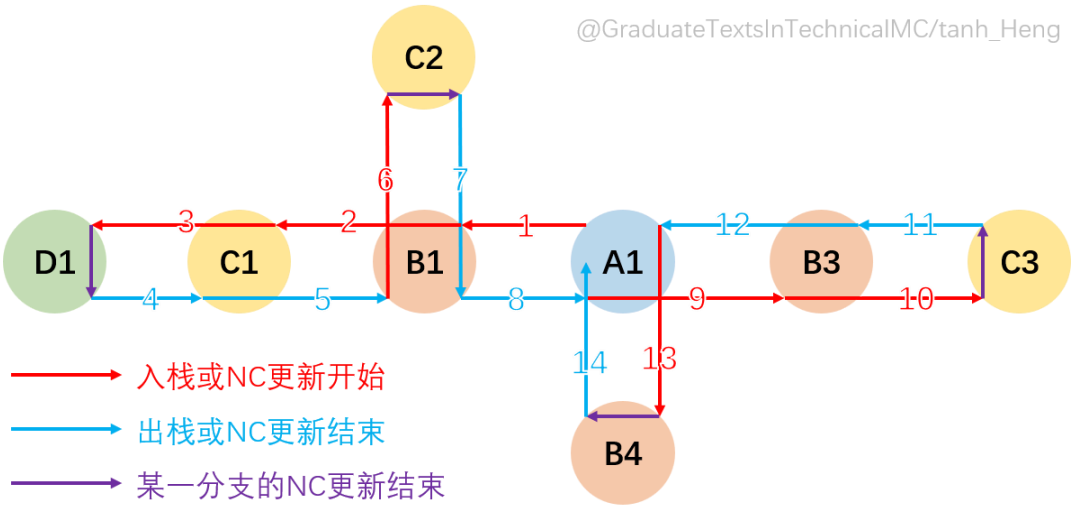
1. Initially, you are at A1, so you place a torch at A1
2. You start searching for paths in the order **West, East, Down, Up, North, South**, and you find a block to the west
3. You enter this block (B1), place a torch, continue checking → find a path to the west, enter C1, place a torch, check → path 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`

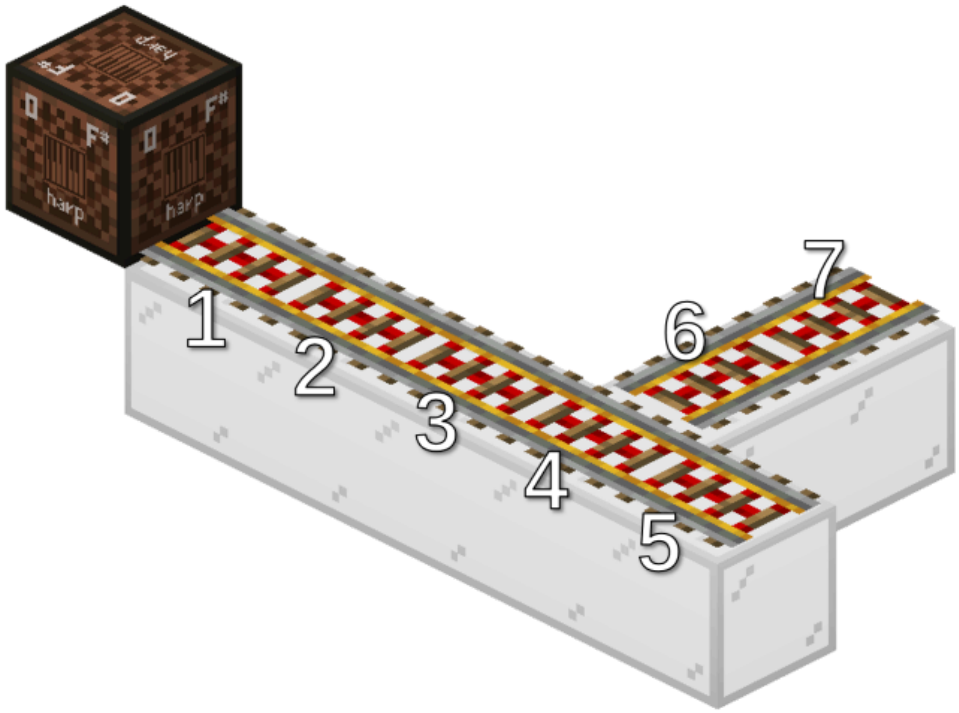
Imagine the update behavior as a point that starts from A1, moves according to a certain update order, and "doesn't turn back until hitting a wall". Then, the movement path of this point forms the arrows shown in the diagram. The process of progressively going deeper is pushing onto stack or NC update,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

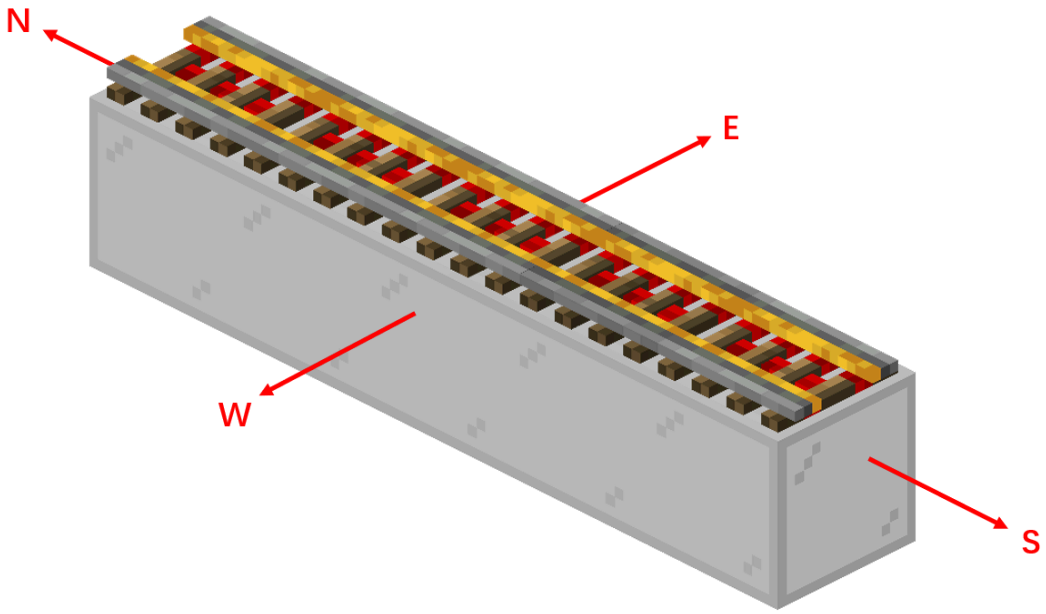
Therefore,

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.



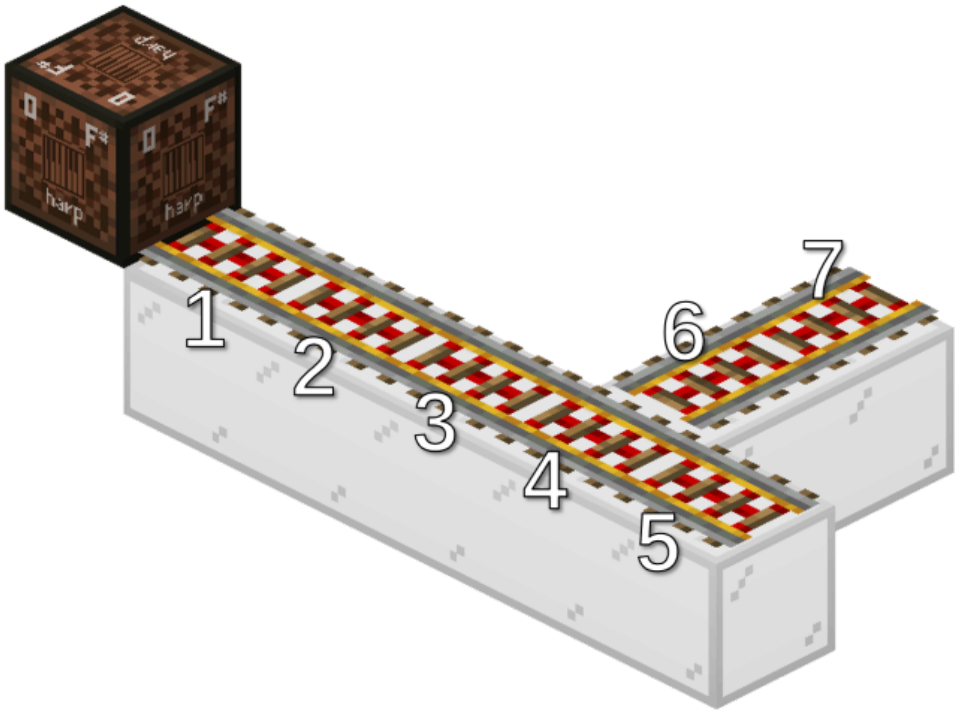
The rail chain direction in the diagram is as marked, with the rail chain oriented north-south. We will use the northernmost rail as an example to analyze its NC update order in different directions.

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. ↩

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

```
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

```
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.

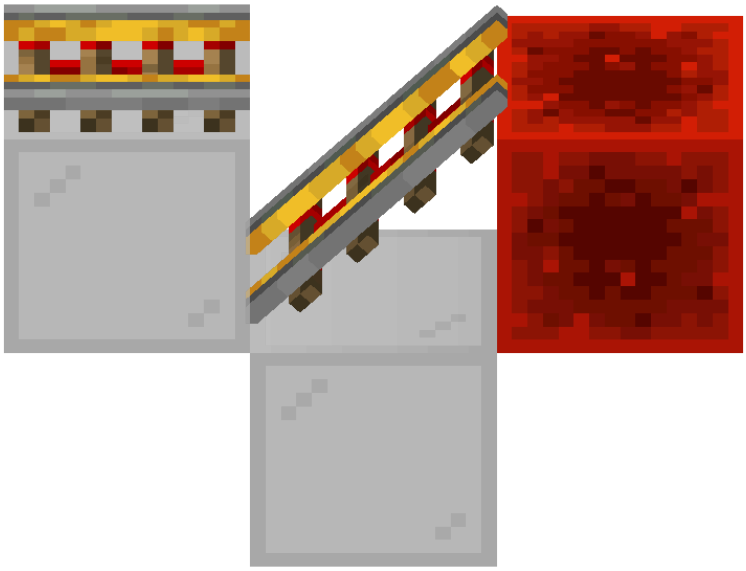
4.2 SOURCE CODE ANALYSIS

```
// 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;
    }
    int i = pos.getX();
    int j = pos.getY();
    int k = pos.getZ();
    boolean bl2 = true;
    RailShape railShape = state.get(SHAPE);
    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
        case NORTH_SOUTH: {
            if (bl) {
                ++k; // 铁轨南北向，则先 z 增（向南）
                break;
            }
            --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
            break;
        }
        case EAST_WEST: {
            if (bl) { // 同上，铁轨东西向，则先 x 减（向西）
                --i;
                break;
            }
            ++i; // 同上，x 增（向东）
            break;
        }
        case ASCENDING_EAST: { // 铁轨东侧上升，即东高西低的斜铁轨
            if (bl) {
                --i; // 因为西低，所以 - x 西侧可能存在同 y 的铁轨连接
            } else {
                ++i; // 因为东高，所以 + x 东侧不可能存在同 y 的铁轨连接
                ++j; // 所以 y + 1，东侧若有连接那么铁轨必然高 1 格
                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。
                // 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。
                // 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
            }
            railShape = RailShape.EAST_WEST; // 上文种步骤 5 的形状更改
            break;
        }
        case ASCENDING_WEST: { // 同理
            if (bl) {
                --i;
                ++j;
                bl2 = false;
            } else {
                ++i;
            }
            railShape = RailShape.EAST_WEST;
            break;
        }
        case ASCENDING_NORTH: { // 同理
            if (bl) {
```

```
                ++k;
            } else {
                --k;
                ++j;
                bl2 = false;
            }
            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, rail-
Shape)) {
        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 || rail-
Shape == 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 || rail-
Shape == 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:

```
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. ↩

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:

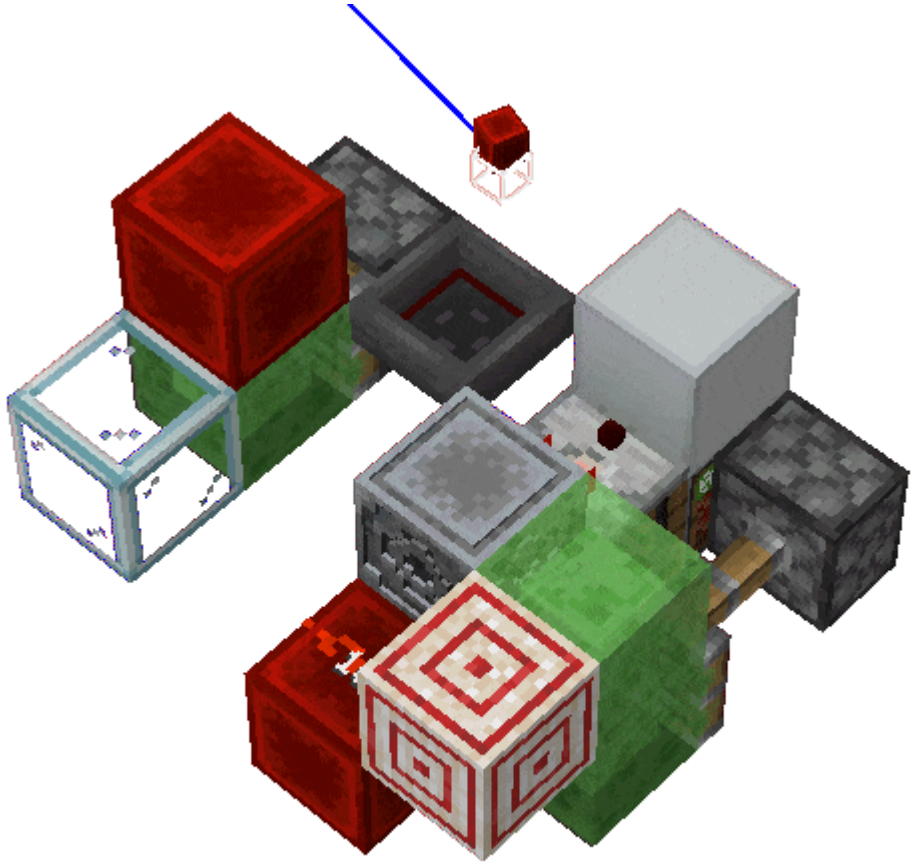
- When no book is placed on the lectern, output signal strength is 0.
- When a book is placed on the lectern and the book has only one page, output signal strength is 15.
- When a book is placed on the lectern and the book has more than one page, output signal strength is floor of `current page number / (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`):

```
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`:

```
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`:

```
// 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`):

```
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:

```
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`:

```
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`:

```
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`:

```
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:

```
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, boolean 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`:

```
@Override
public void onBlockAdded(BlockState state, World world, BlockPos pos, BlockState oldState, boolean 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:

```
@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	`000000001`	Trigger surrounding block updates	Commonly called NC Update
`NOTIFY_LISTENERS`	2	`000000010`	Notify server and client listeners	Sync state
`NO_REDRAW`	4	`000000100`	Prevent client redraw (used with `NOTIFY_LISTENERS`)	Invisible update
`REDRAW_ON_MAIN_THREAD`	8	`000001000`	Force client sync redraw	Block appearance change
`FORCE_STATE`	16	`000010000`	Bypass virtual state, directly store specified state	Force block replacement
`SKIP_DROPS`	32	`000100000`	Prevent container blocks from dropping items when broken	Command replacement etc.
`MOVED`	64	`001000000`	Mark block moved by piston	Piston push/pull logic trigger

| `SKIP_REDSTONE_WIRE_STATE_REPLACEMENT` | 128 | `010000000` | Skip redstone wire state replacement | - |

| `SKIP_BLOCK_ENTITY_REPLACED_CALLBACK` | 256 | `100000000` | Skip block entity replacement callback | - |

| `SKIP_BLOCK_ADDED_CALLBACK` | 512 | `1000000000` | Skip self-inspection | Don't trigger `onBlockAdded` |

| `SKIP_REDRAW_AND_BLOCK_ENTITY_REPLACED_CALLBACK` | 260 | `100000100` | Combine `NO_REDRAW` and `SKIP_BLOCK_ENTITY_REPLACED_CALLBACK` | Minimize update, skip rendering and entity callback |

| `NOTIFY_ALL` | 3 | `000000011` | Combine `NOTIFY_NEIGHBORS` and `NOTIFY_LISTENERS` | Default full server-side update |

| `NOTIFY_ALL_AND_REDRAW` | 11 | `000001011` | Combine `NOTIFY_ALL` and `REDRAW_ON_MAIN_THREAD` | Server update and force client redraw |

| `FORCE_STATE_AND_SKIP_CALLBACKS_AND_DROPS` | 816 | `1100110000` | 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`. ↩

CHAPTER 2

LOADING TICKETS

3 ARTICLES

This chapter introduces the basics of loading tickets.

1 LOADING TICKET

A loading ticket contains:

`net.minecraft.server.world.ChunkTicket`

- `type` The loading ticket type; records the reason for loading.
- `level` The loading level; determines what computations the chunk performs.
- `tickCreated` The creation time; used to determine whether the loading ticket has expired.

2 LOADING TICKET TYPE

There are 8 types of loading tickets in Minecraft:

`net.minecraft.server.world.ChunkTicketType`

```
    public static final ChunkTicketType<Unit> START = create("start", (a, b) -> 0);
    public static final ChunkTicketType<Unit> DRAGON = create("dragon", (a, b) -> 0);
        public static final ChunkTicketType<ChunkPos> PLAYER = create("player",
Comparator.comparingLong(ChunkPos::toLong));
        public static final ChunkTicketType<ChunkPos> FORCED = create("forced",
Comparator.comparingLong(ChunkPos::toLong));
         public static final ChunkTicketType<ChunkPos> LIGHT = create("light",
Comparator.comparingLong(ChunkPos::toLong));
        public static final ChunkTicketType<BlockPos> PORTAL = create("portal",
Vec3i::compareTo, 300);
             public static final ChunkTicketType<Integer> POST_TELEPORT =
create("post_teleport", Integer::compareTo, 5);
        public static final ChunkTicketType<ChunkPos> UNKNOWN = create("unknown",
Comparator.comparingLong(ChunkPos::toLong), 1);
```

- `start`: Used to load the spawn point when the world starts. The loading distance is 11, which corresponds to loading level 22. `net.minecraft.server.MinecraftServer#prepareStartRegion`
- `dragon`: Used to load the End main island during the dragon fight. The loading distance is 9, which corresponds to loading level 24. `net.minecraft.entity.boss.dragon.EnderDragonFight#tick`
- `player`: Used for player chunk loading.
- `forced`: Used to implement the `/forceload` command and force-load spawn points.
- `light`: Used to load chunks that need light calculation.
- `portal`: Used to load chunks in the target dimension when entities or players pass through a portal. If not used within 300gt (15s), the associated loading ticket is removed. The loading distance is 3. `net.minecraft.world.PortalForcer#getPortalRect`
- `post_teleport`: Used to load target chunks when entities or players teleport, such as with `/tp`. Removed after 5gt.
- `unknown`: Used when `getChunk` is called in the game code to load chunks. Removed after 1gt.

The loading level corresponds to the loading distance via the following algorithm:

Loading level = 33 - Loading distance

`net.minecraft.server.world.ChunkTicketManager#addTicket`

```
            ChunkTicket<T>      chunkTicket      =      new      ChunkTicket<>(type,
ChunkLevels.getLevelFromType(ChunkLevelType.FULL)      -      radius,      argument);
//ChunkLevelType.FULL = 33
```

Note that `ChunkTicketType` does not specify the loading distance. For example, `post_teleport` has a loading distance of 1 when used with the `/tp` command, but 0 in other cases.

3 LOADING LEVEL

The loading level is restricted to [22-45]. Here's why the upper limit is set this way:

```
public static final int INACCESSIBLE = 33 + ChunkStatus.getMaxDistanceFromFull();
```

In 1.20.1, ChunkStatus.getMaxDistanceFromFull() returns 12. See section 1.6 for details.

4 LOADING LEVEL TYPE

The loading level determines what operations a chunk can perform. There are four types:

`net.minecraft.server.world.ChunkLevelType`

```
public enum ChunkLevelType {
    INACCESSIBLE,
    FULL,
    BLOCK_TICKING,
    ENTITY_TICKING;
}
```

The relationship between loading level and level type:

`net.minecraft.server.world.ChunkLevels#getType`

```
    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;
        }
    }
```

The level type determines what computations the chunk performs.

- Entity Ticking (strong loading) `ENTITY_TICKING`: When the loading level is 31 or below, entity ticking occurs. All game processes are computed, including entity logic.

- Block Ticking (weak loading) `BLOCK_TICKING`: When the loading level is 32 or below, basic computation occurs. Entities are not ticked, but other systems operate normally, such as redstone components.
- Loading Boundary `FULL`: When the loading level is 33, it's the loading boundary. Almost no game processes are computed, but entities are tracked, meaning entities within the loading boundary are counted toward the mob cap.
- Inaccessible `INACCESSIBLE`: When the loading level is greater than 33, it's inaccessible. These chunks are not truly loaded; only partial world generation has been performed.

5 CHUNKHOLDER

This is outside the scope of this article, but here is a brief introduction to ChunkHolder's role.

`ChunkHolder` is the container for chunks in the underlying chunk management system. It stores the chunk itself, the chunk loading level, the chunk generation status, and the chunk's allowed uses. It can be roughly thought of as the in-game instance of a chunk.

6 CHUNKSTATUS

This is outside the scope of this article, but here is a brief introduction to ChunkStatus.

`ChunkStatus` corresponds to the chunk generation stage. In 1.20.1, the stages are:

- `empty`: Empty chunk;
- `structure_starts`: Identifies structures that may begin generating in the chunk;
- `structure_reference`: Used for world generation;
- `biomes`: Determines biomes;
- `noise`: Generates the rough terrain outline and bedrock layer;
- `surface`: Replaces blocks near the surface;
- `carvers`: Generates caves and ravines;
- `features`: Generates underwater caves and ravines;
- `initialize_light`: Calculates the initial lighting of the chunk?;
- `light`: Calculates the lighting of the chunk;
- `full`: Chunk loading is complete; `ProtoChunk` converts to `WorldChunk`.

Chunk status corresponding to each loading level:

- 34-: `full`;
- 35: `initialize_light`;
- 36: `carvers`;

- 37: `biomes`;
- 38~45: `structure_starts`.

Below is a glossary of terms used in this article. Most translations come from the wiki.

FULL NAME	TRANSLATION
Lazy Processing Chunk/Lazy Load/Block Ticking	Weak Loading
Entity Processing Chunk/Entity Ticking	Strong Loading
Loading Border Chunk/Border Load	Loading Border
Inaccessible	Inaccessible
RenderDistance/ViewDistance	Render Distance
SimulationDistance	Simulation Distance
Ticket	Loading Ticket
Level	Loading Level

1 CHUNKDEBUG

Link: Modrinth

A mod for viewing chunk loading status on the server.

2 INTRICARPET

Link: Modrinth

A Carpet extension that lets you disable player chunk loading for easier debugging.

`/interaction chunkloading false`

3 MESSMOD

Link: Modrinth

A multi-purpose utility toolkit.

CHAPTER 3

MASA MOD

2 ARTICLES

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.

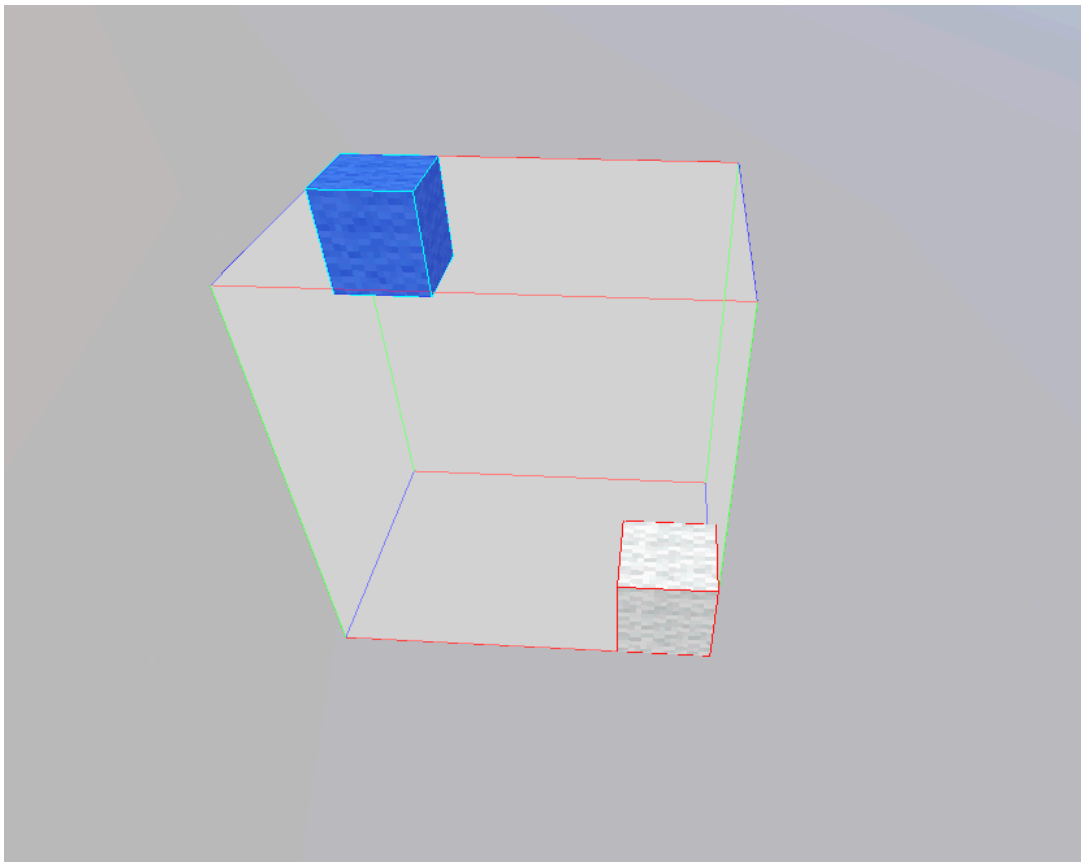
Use ctrl+scroll wheel to switch modes.

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).



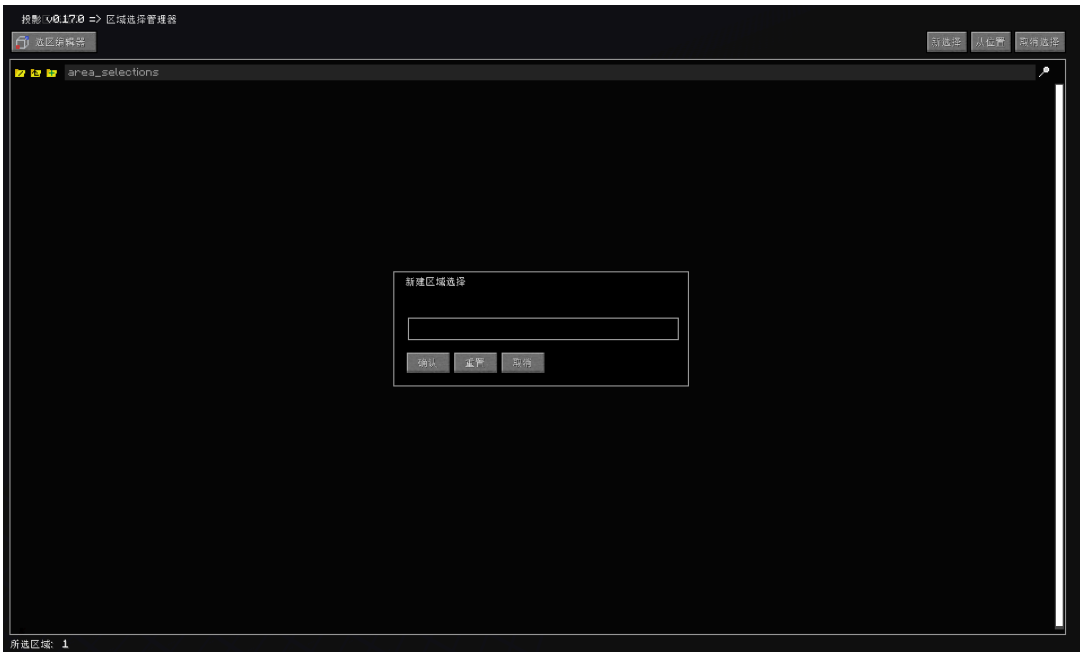
In the selection editor, you can change the name of the selection or save the schematic.

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:

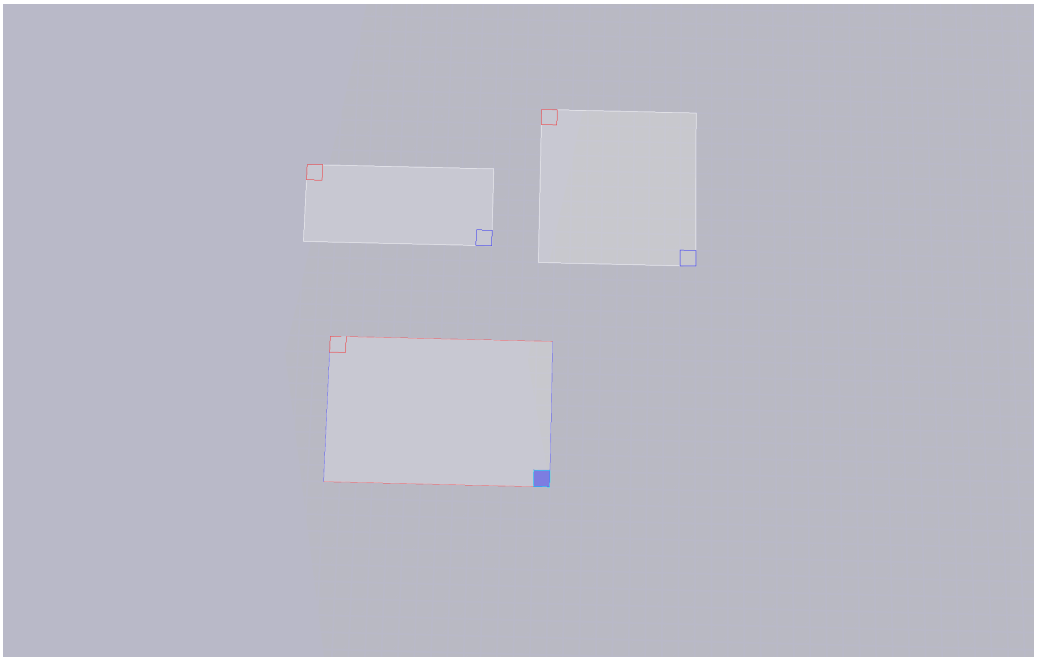
1. Open `Litematica Main Menu - Selection Manager - New Selection`



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.

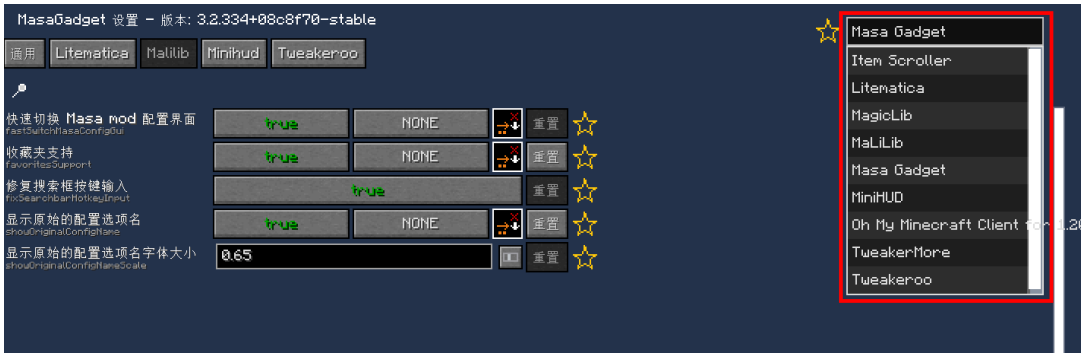
1. Paste the schematic (creative only)

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).

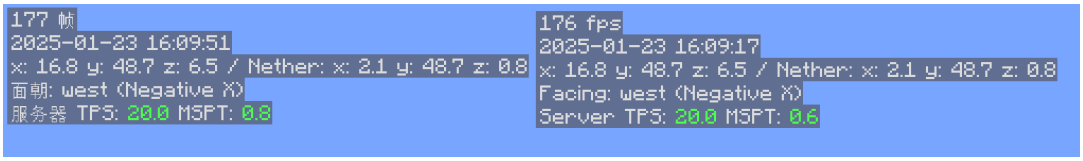
2.2 SAVE CONTAINER DATA WHEN SAVING SCHEMATICS ON SERVER

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

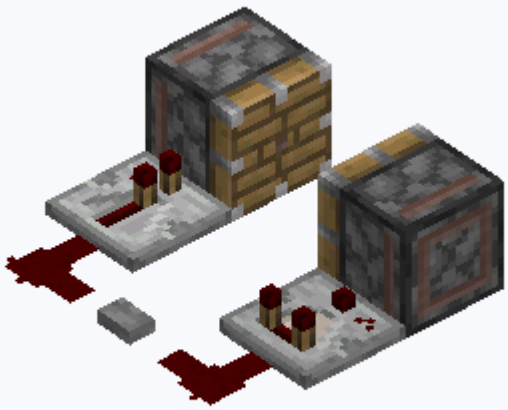
Pressing the §6Container Preview§r hotkey uses the PCA sync protocol to synchronize beehive and bee nest data.

CHAPTER 4

MICRO TIMING

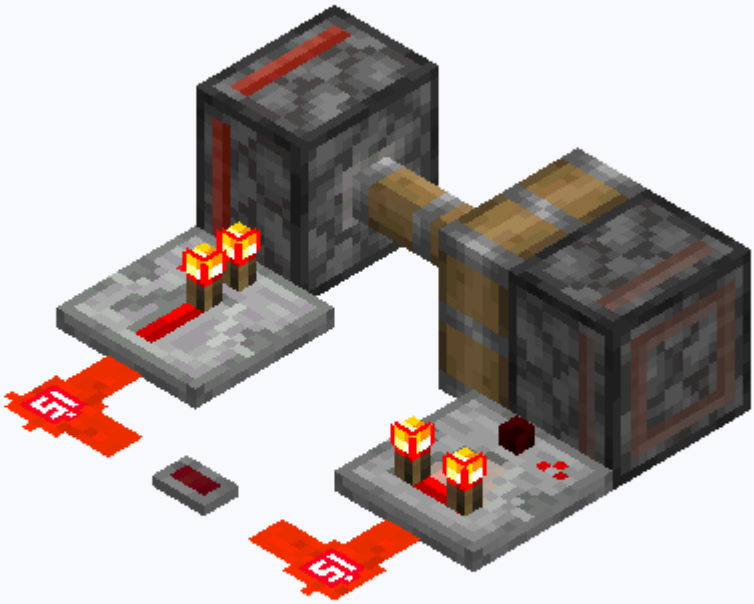
5 ARTICLES

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

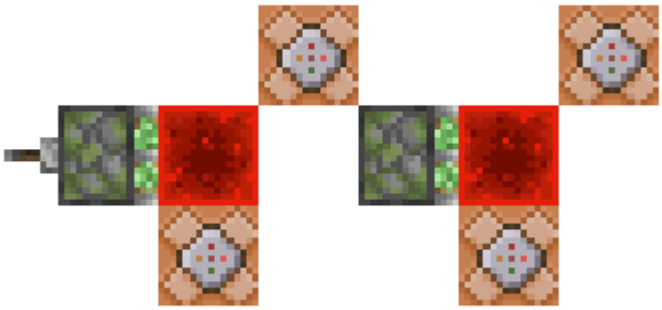
COMPONENT TYPE	OPERATING PHASE
Command block runs command	Scheduled Tick TT
Repeater, Comparator, Redstone Torch, Observer on/off	Scheduled Tick TT
Redstone Dust, Rails change state	Instant
Fence Gates, Trapdoors change state	Instant
Hopper changes state due to redstone signal	Instant
Hopper absorbs,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

COMPONENT TYPE	OPERATING PHASE
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

TICK	PHASE	EVENT
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. ↩

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 re-

quired 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

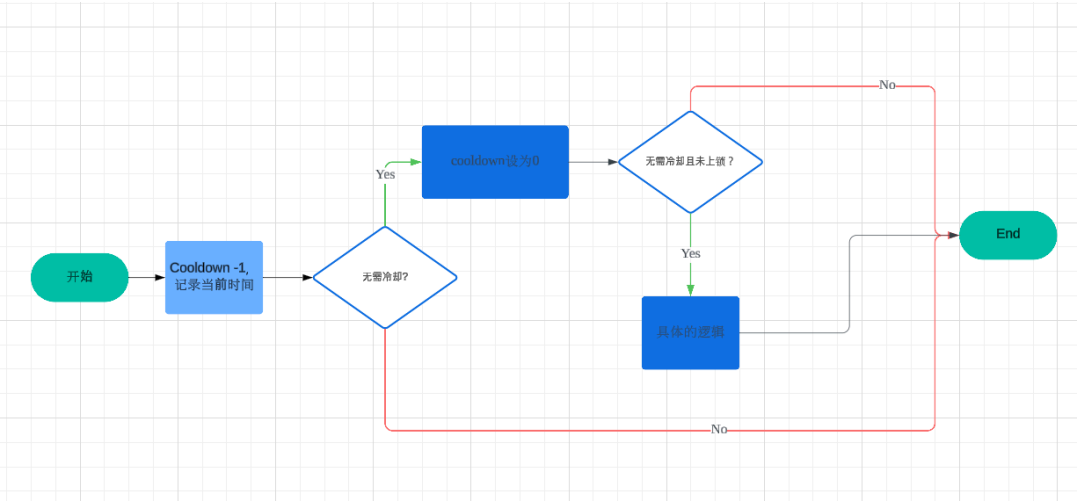
2.1.2 HOPPER BLOCK ENTITY FUNCTIONS

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

2.1.3 HOPPER BLOCK ENTITY WORKFLOW

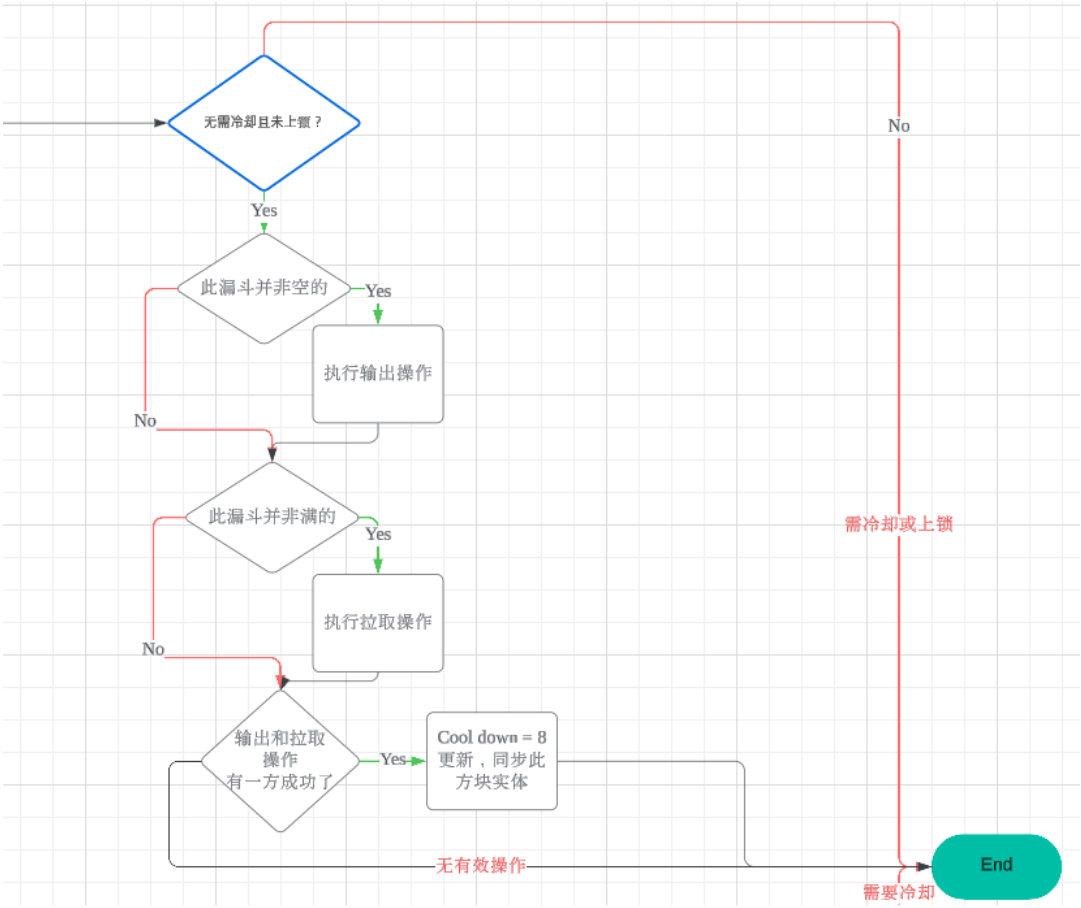
Every game tick, the hopper executes these steps:

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.

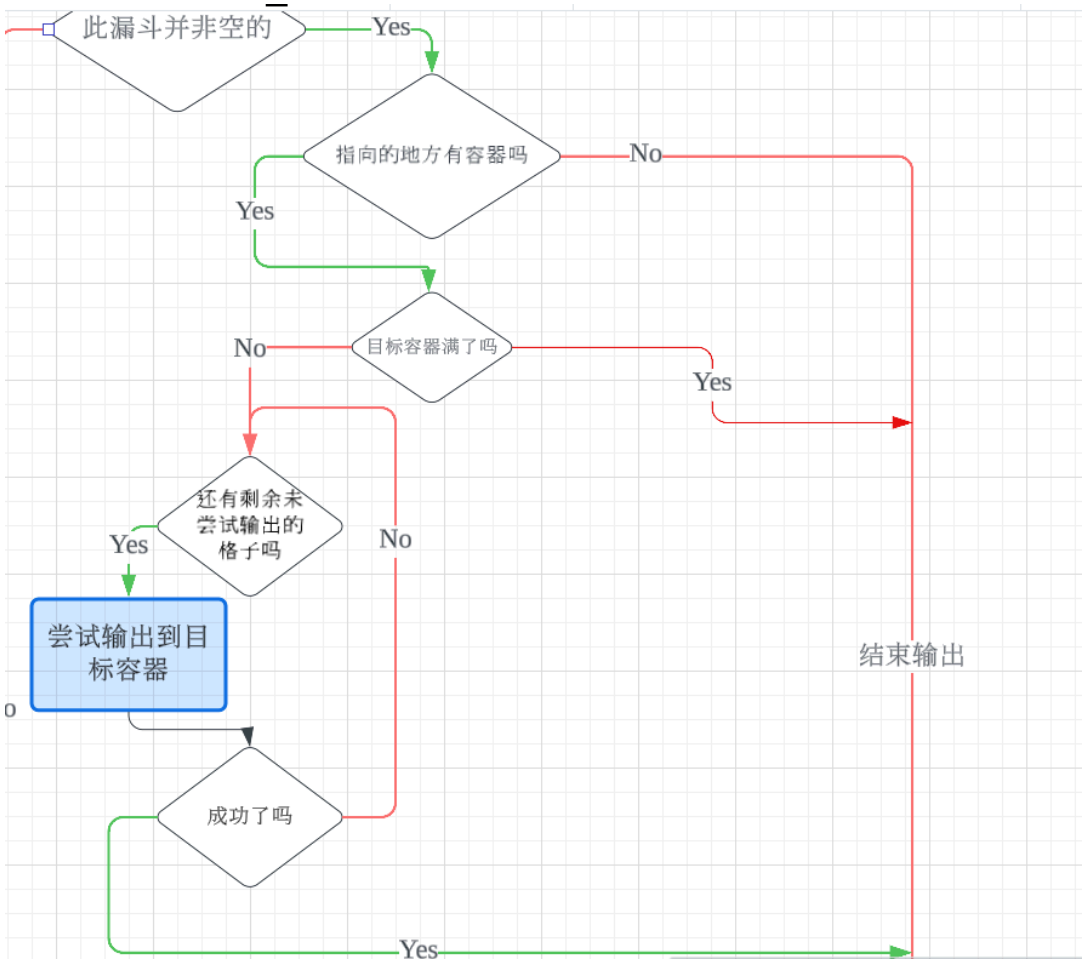


Let's look at output and pull separately.

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.

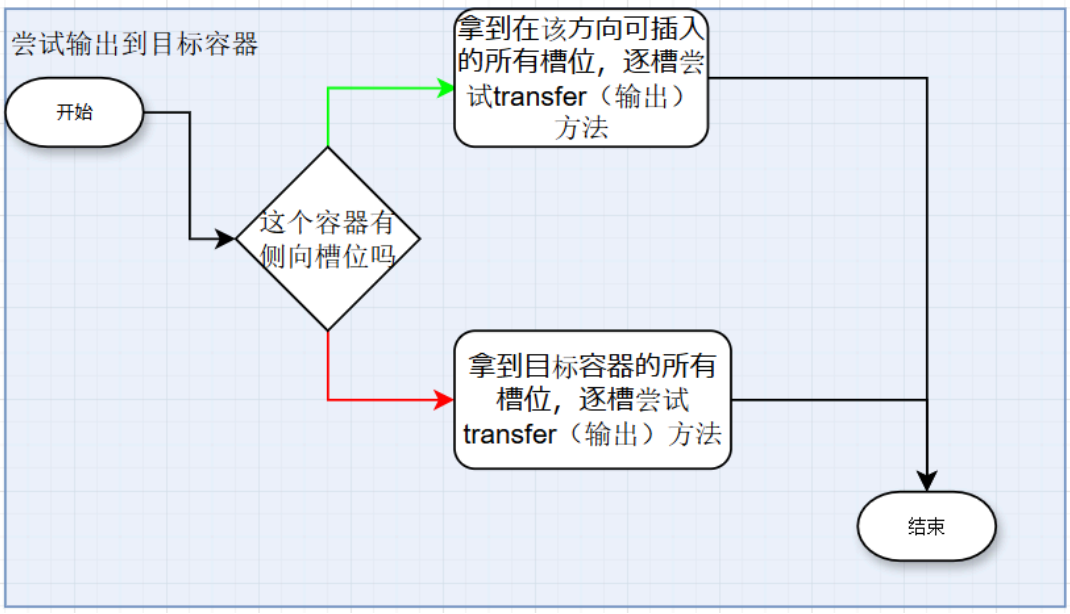
Basic output logic:

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:

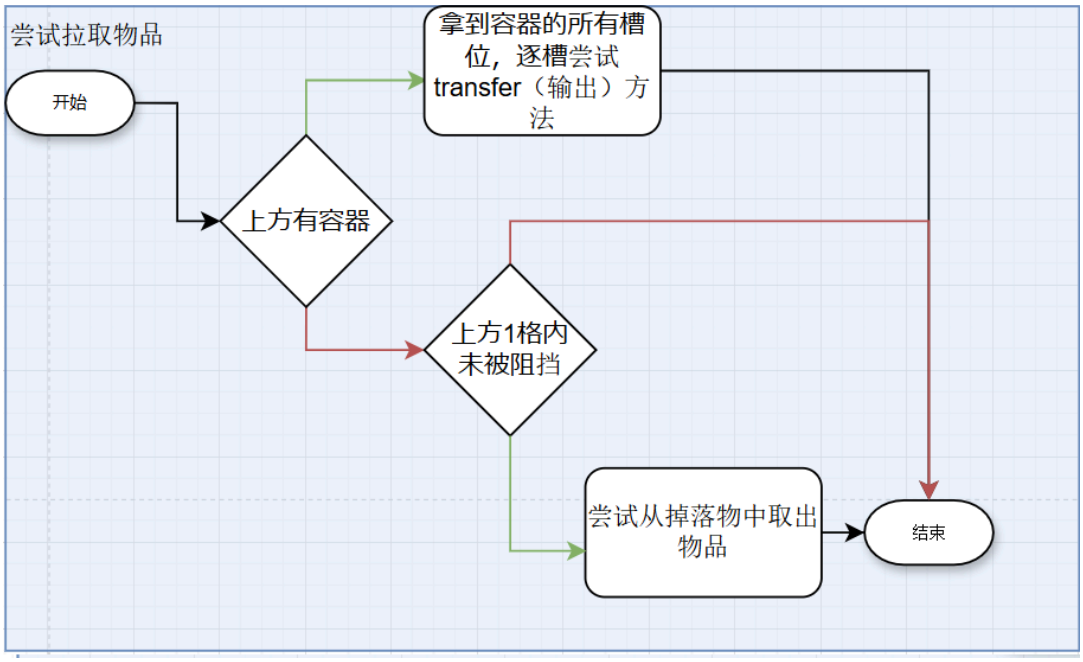
- An empty return value means the items were fully transferred, so output succeeds.
- A non-empty return value means the items could not be fully transferred, so output fails.



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:

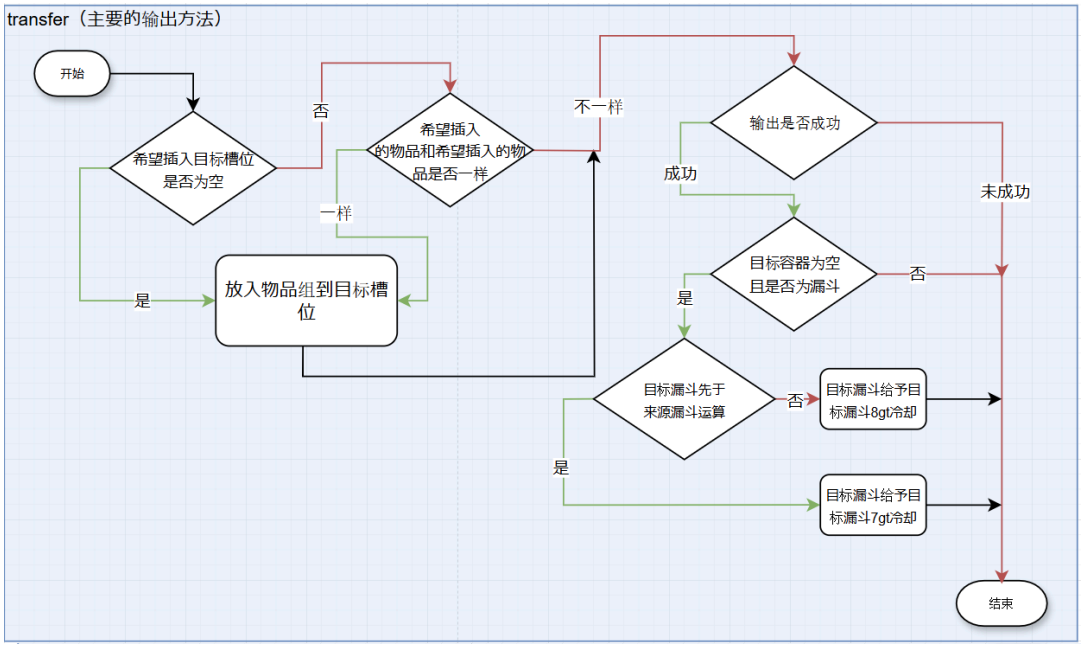
- **from**: item source (can be a hopper or other container)
- **to**: transfer target (can be a container or another hopper)
- **itemStack**: the item stack to transfer
- **slot**: target slot in the target container

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):

```
// Used to temporarily store block entities dynamically added during the tickBlock-
Entities process
private final List<TickingBlockEntity> pendingBlockEntityTickers = Lists.newAr-
rayList();
// 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.blockEntityT-
ickers)
            .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);
   });
}
```

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)

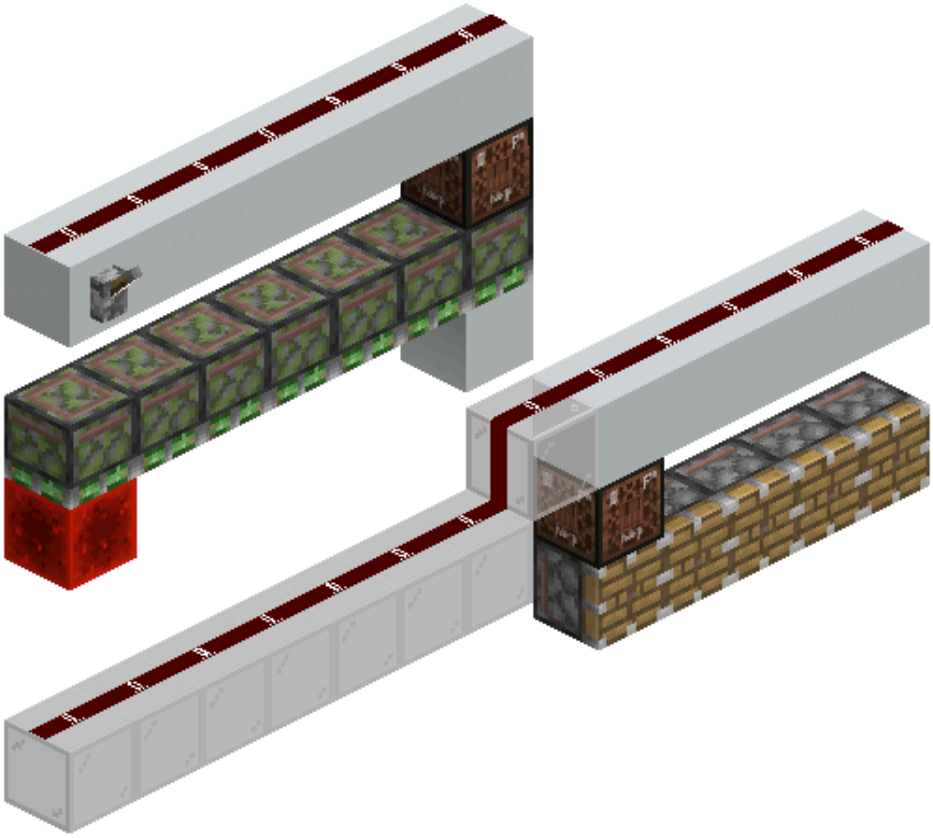
...

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:



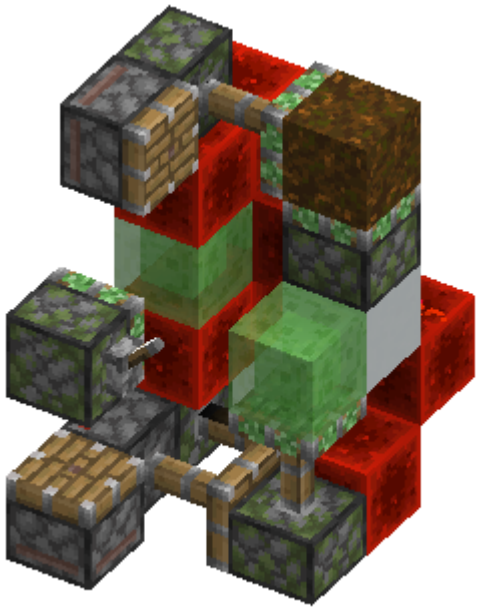
Flip the lever and you'll notice that moving the redstone block changes how many pistons extend nearby. When the redstone block sits closer to the pressure plate block, fewer pistons extend; when it moves farther away, more extend.

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:



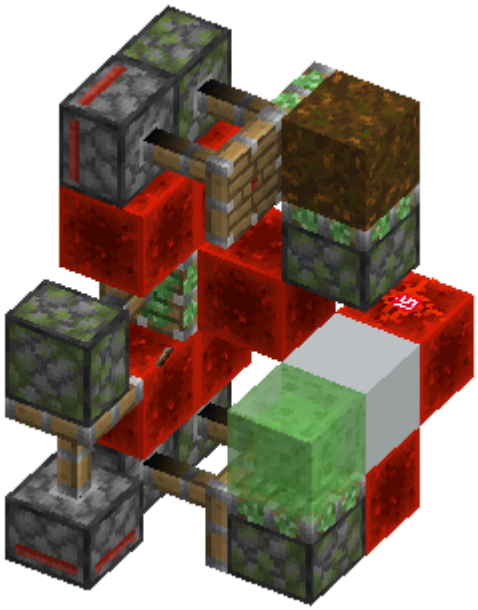
0gt AT: Lever is pulled down

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.

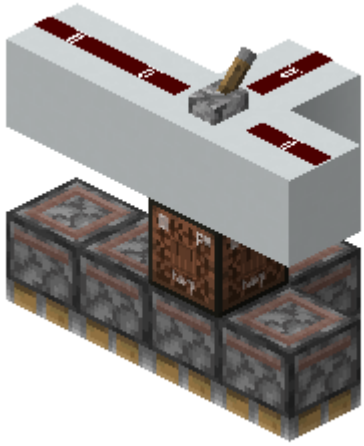
---

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.

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 always 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.

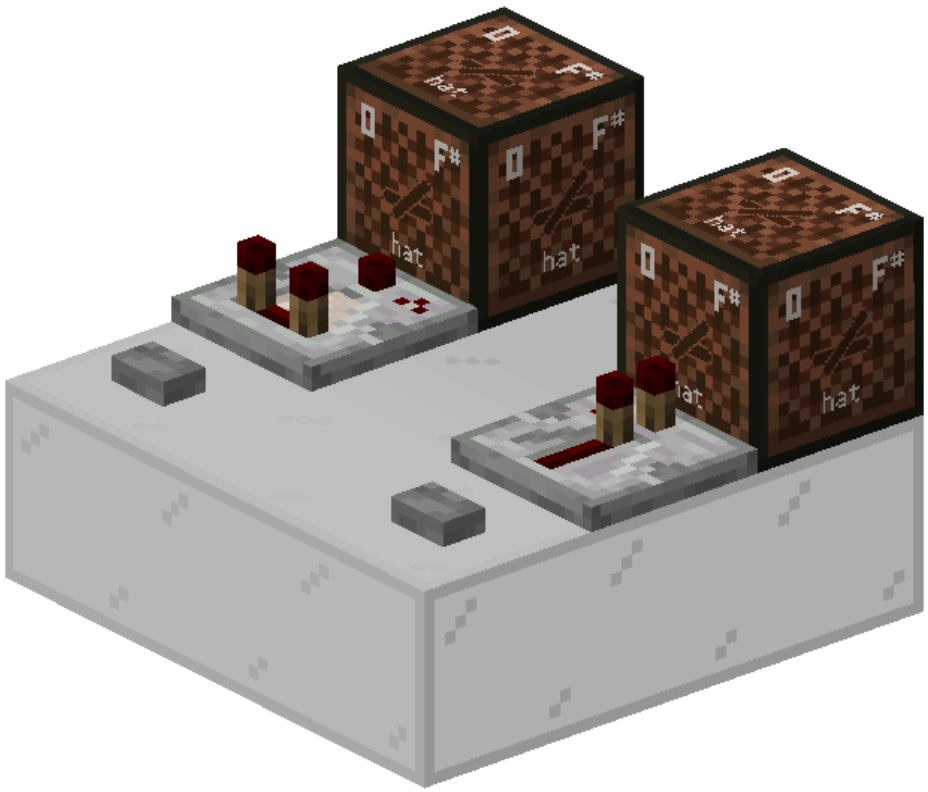
Therefore, when comparing the execution order of Scheduled Ticks, we can follow this logic:

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

INPUT	OUTPUT
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:

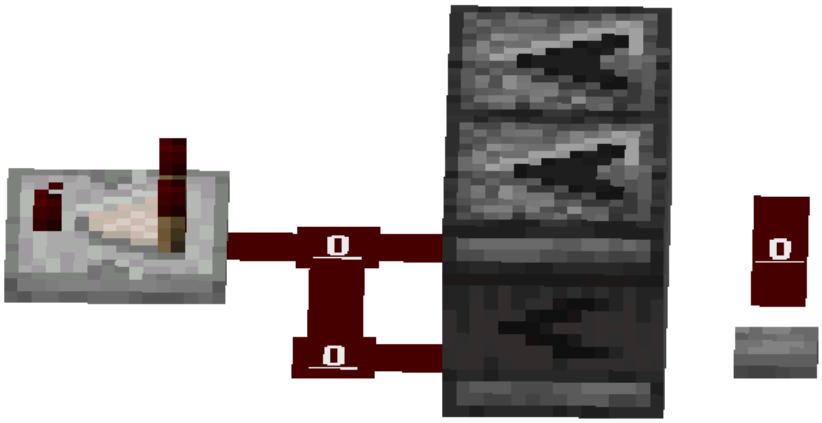
- If a comparator points to a horizontally placed redstone diode or points to the input of a redstone diode, the Scheduled Tick's priority is `-1`
- Otherwise, the Scheduled Tick's default priority is `0`

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:

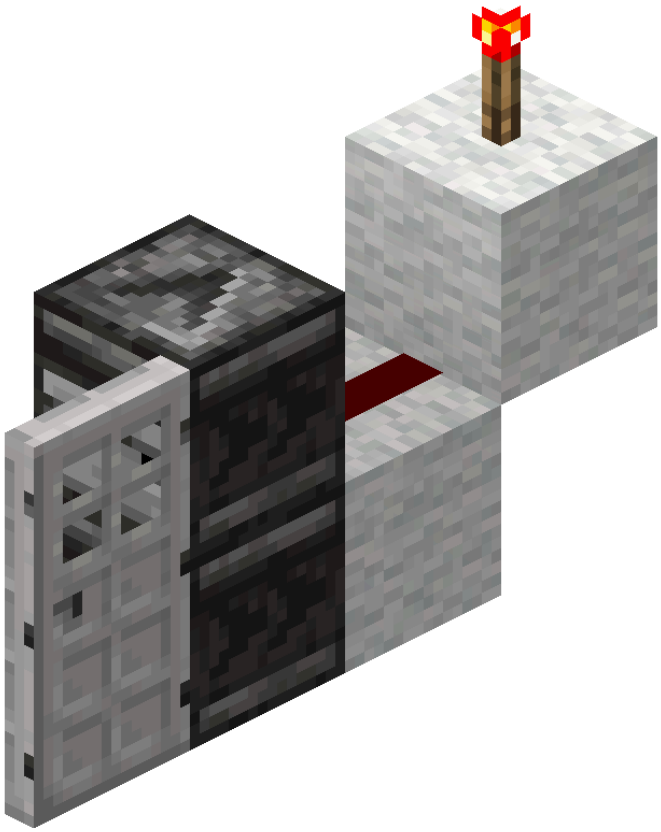
- Observer lights up:
  - Sends PP update, nothing happens.
  - Observer **adds Scheduled Tick**
  - Sends NC update:
    - Redstone torch receives NC update:
      - **Adds Scheduled Tick**

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`:

```
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. ↩

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**.

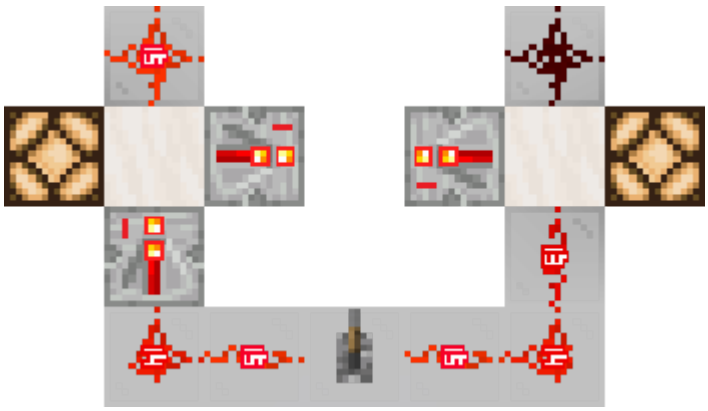
When a chargeable block is strongly charged, it can activate Redstone Dust in all six adjacent directions, regardless of orientation. When weakly charged, it can only activate Repeaters, Comparators, and some other redstone components whose inputs face the chargeable block. This is also an important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

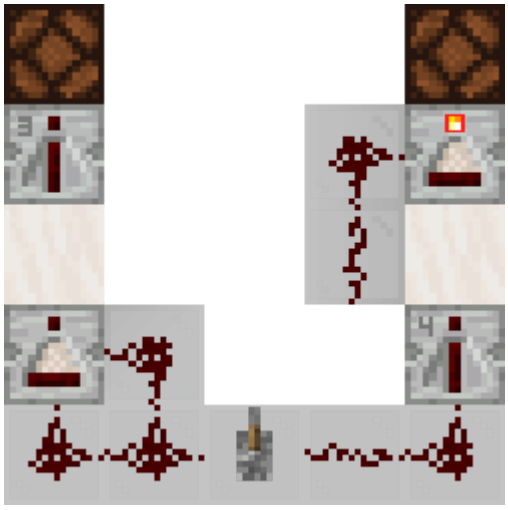
- A chargeable block is pointed to by Redstone Dust with a signal strength not equal to 0.



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.

CHAPTER 5

TREE FARM

7 ARTICLES

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

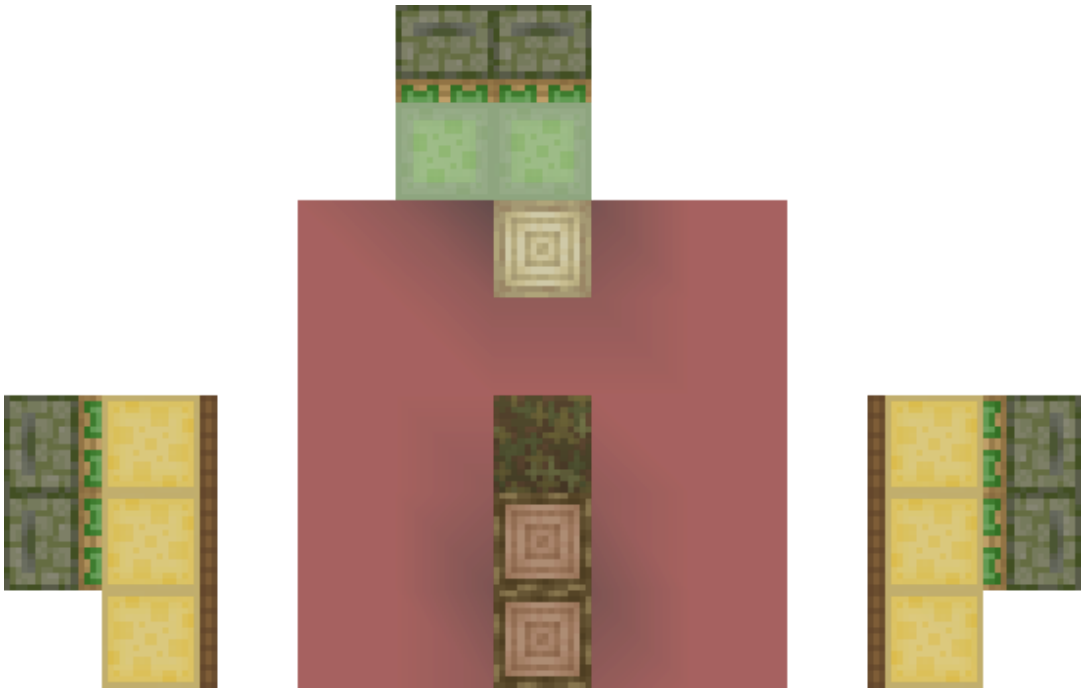
the five classic tree species: **Birch, Oak, Jungle, Spruce, and Acacia**. We'll walk through them one by one in the typical design order.

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**.

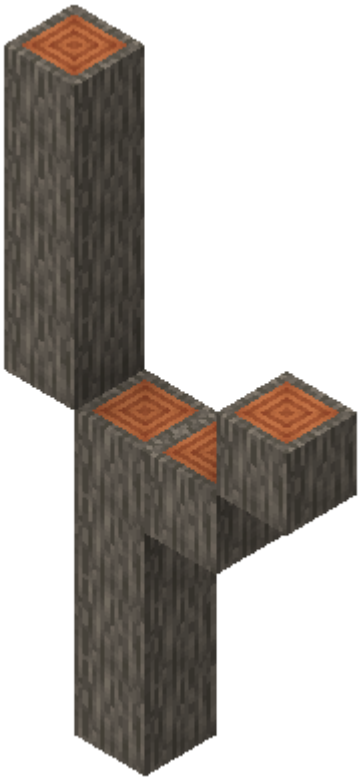
Example architecture:

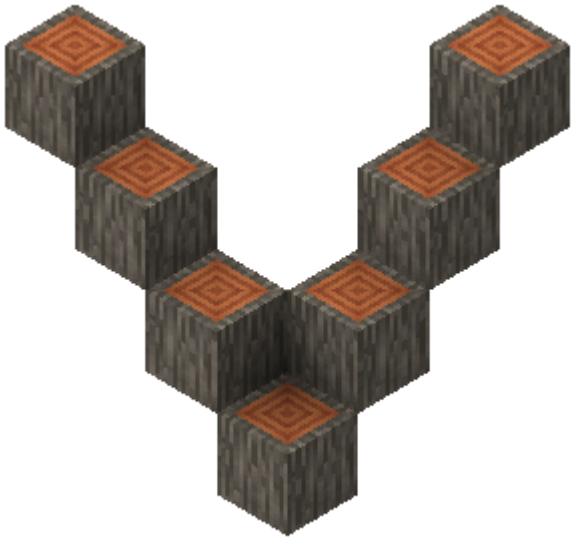


> *The main pusher design will significantly impact your oak height-limiting options, so choose carefully*

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.



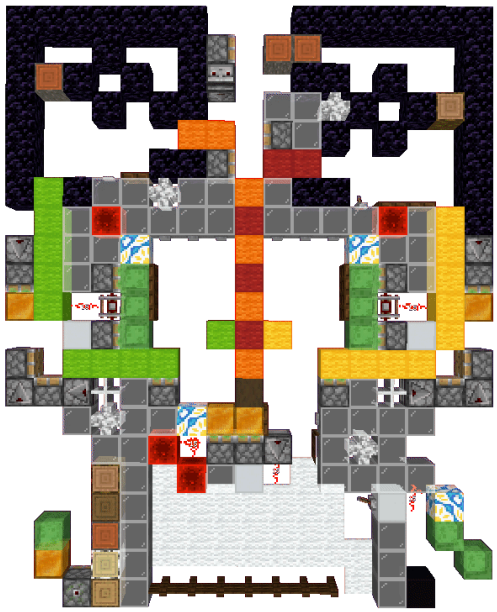


This means the main pusher alone cannot clear all the logs, so we need **side branch processing**. There are two design approaches: **center consolidation** and **additional log outputs**.

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

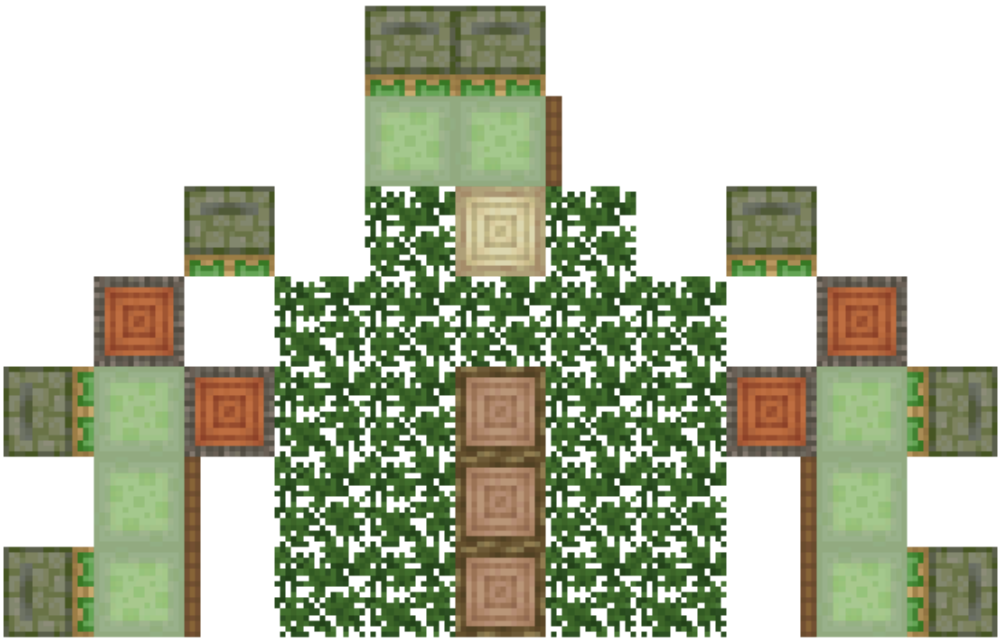
> *Some architectures use both methods at once, such as PUTF -2. PUTF// also offers a different approach: adding one block to the main pusher, consolidating side branches into the middle 2 blocks, then pushing them out directly. However, this method doesn't work for cherry trees and isn't particularly 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*

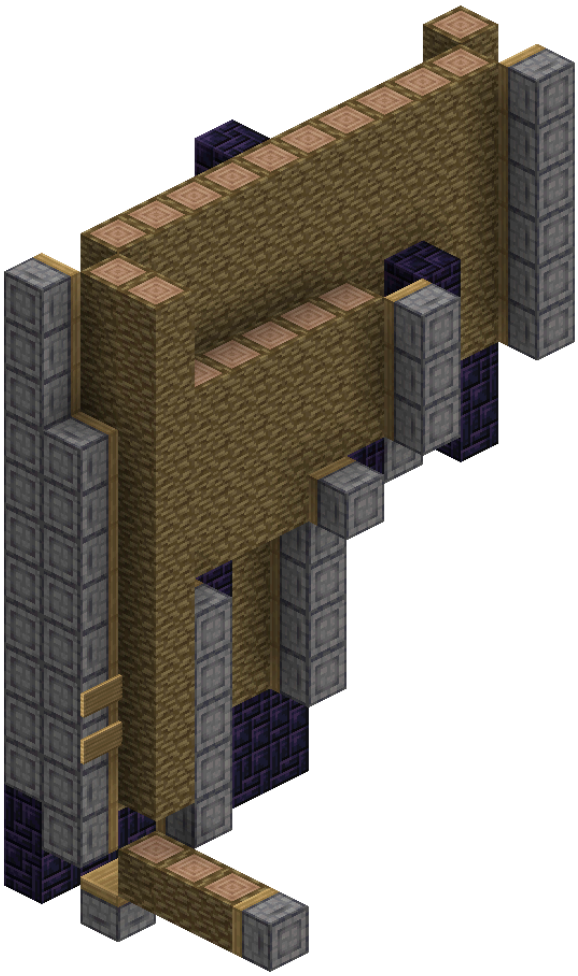
2 BLOCK STREAM TO DROPS ARCHITECTURE DESIGN

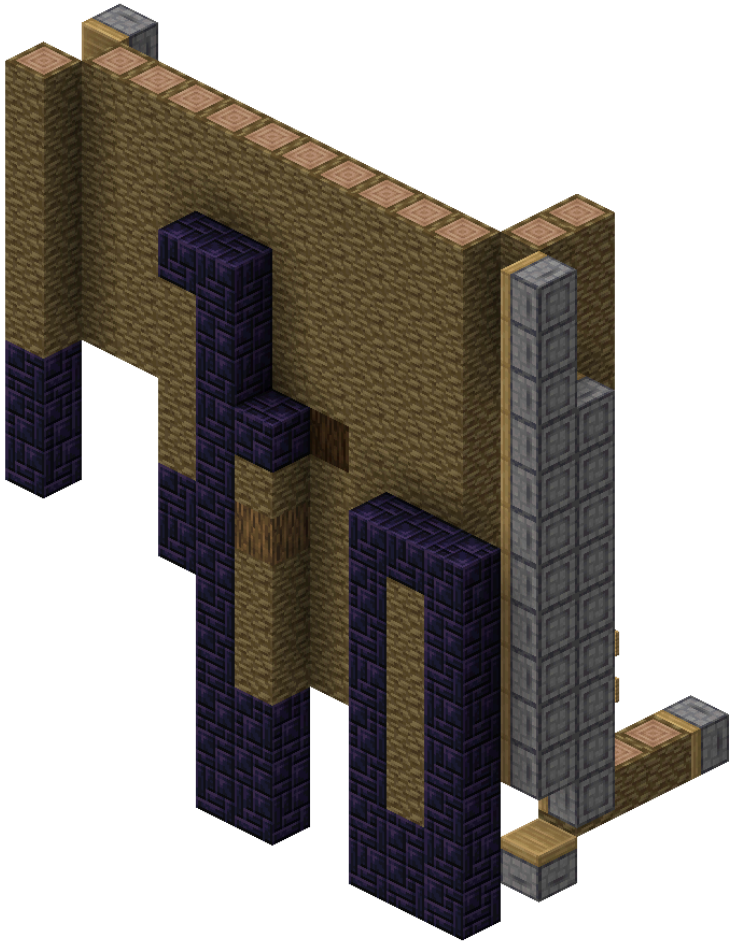
Because multi-species tree farms produce an irregular block stream output, we need to **reorganize** the block stream before feeding it into a specially designed explosion chamber.

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**.

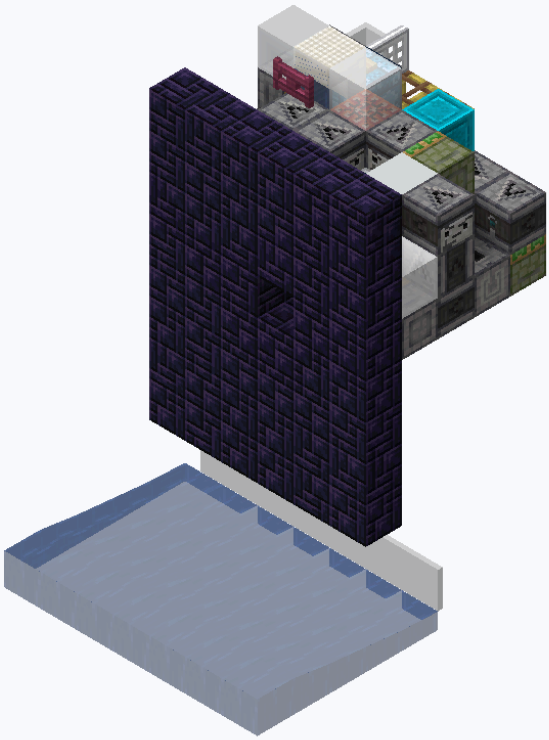
> *For multi-species tree farms, we can use two specialized explosion structures: **pure milk explosion chambers** (commonly called "milk explosions") and **b36 explosion chambers** (commonly called "push explosions") to handle large volumes of irregular block streams. These are widely used in modern 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

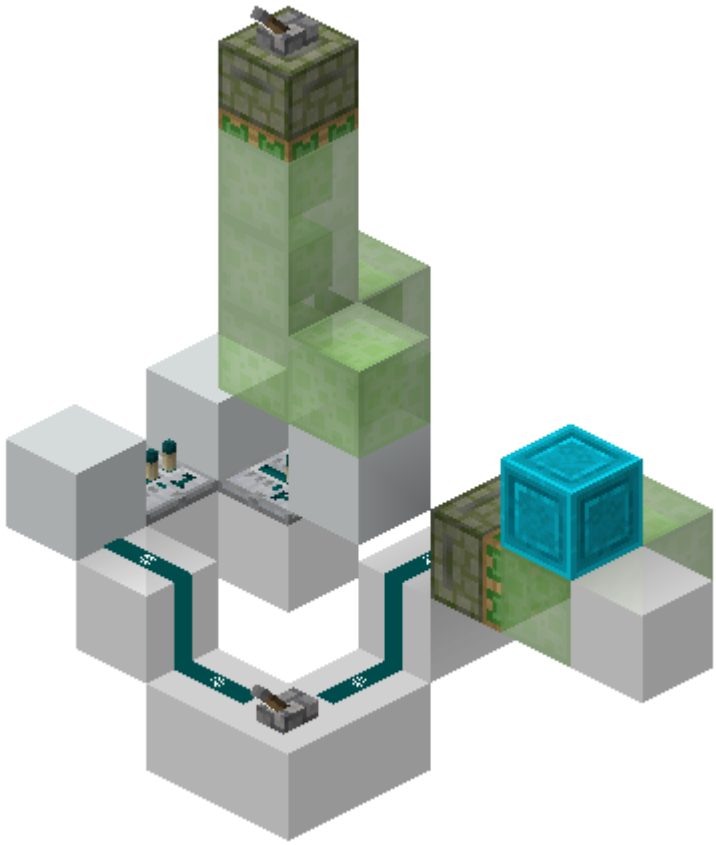
3 WIRING—MODE SWITCHING AND SPEED LIMITER

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.

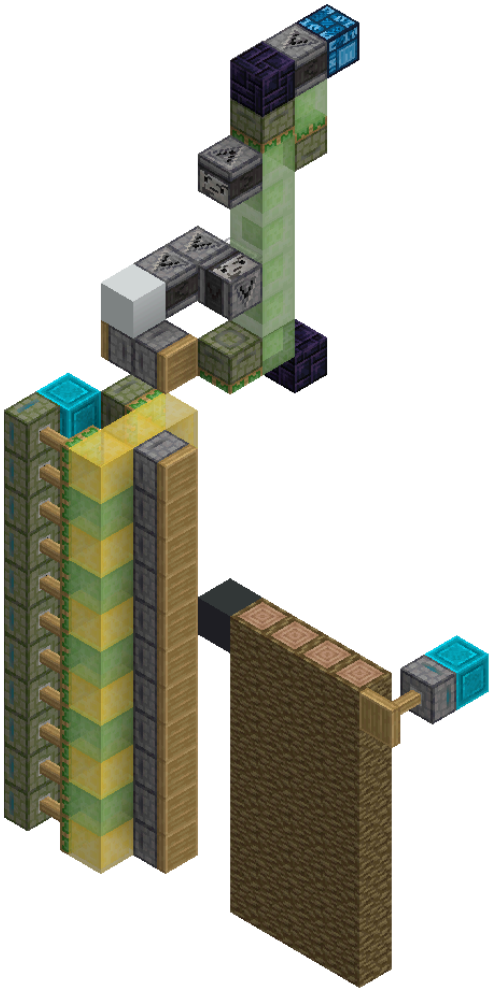


3.2 AUTOMATIC OAK HEIGHT LIMITING

Many players find manually placing obsidian tedious. There's a solution for that: automatic height limiting.

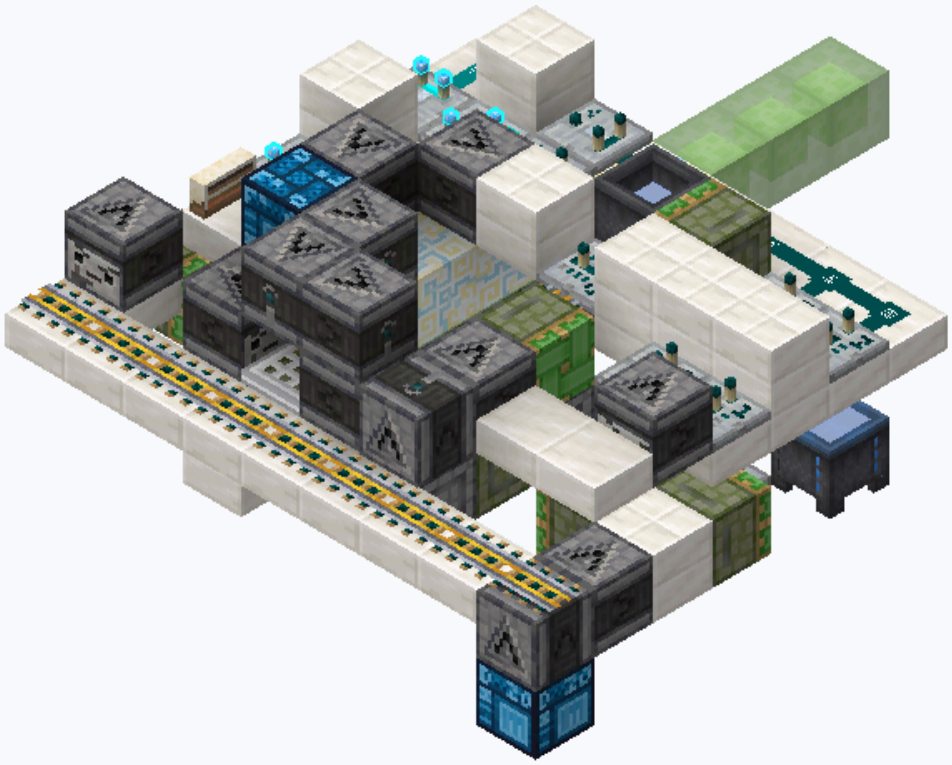
Since automatic height-limiting methods vary widely between designs, we'll only cover the one used in our example architecture.

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 push-able 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:

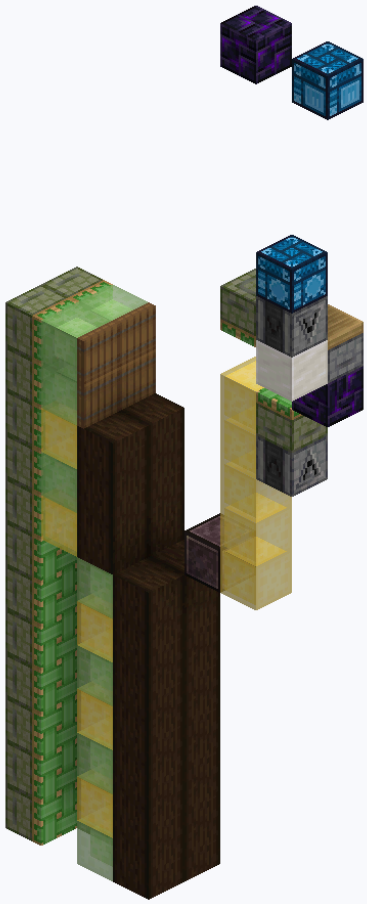


*@Jellie_*

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*

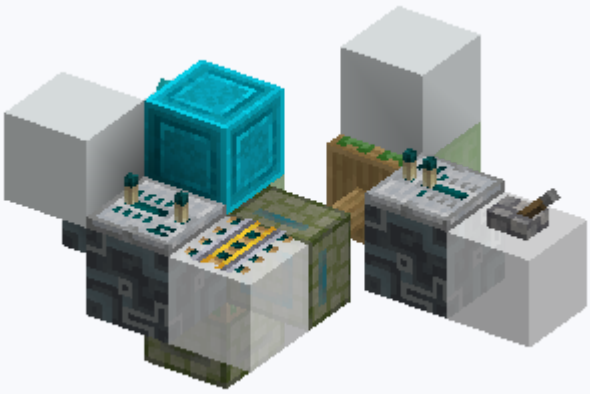
3.4 SPEED LIMITER

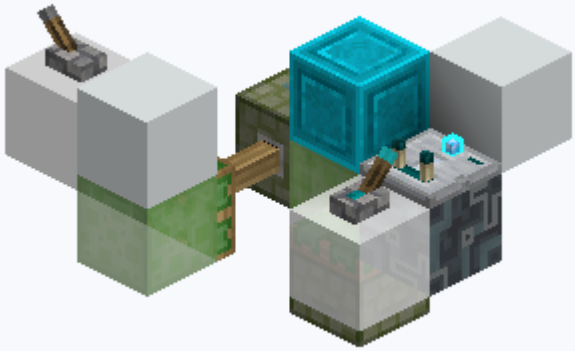
There's one more issue. Trees can grow before our architecture has fully reset, which would trigger the detection structure prematurely and break the farm. So we need to add a speed limiter to the detection structure. The limiter's timing must also account for the operating timing (reset time) of the 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.

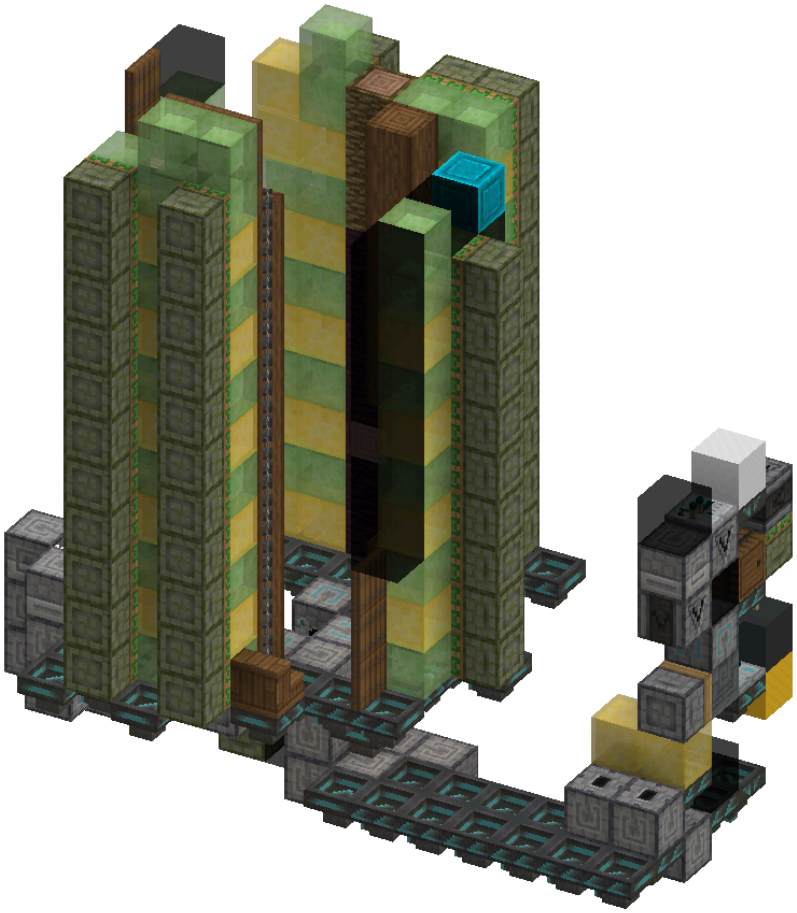
Here is the example we made!

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

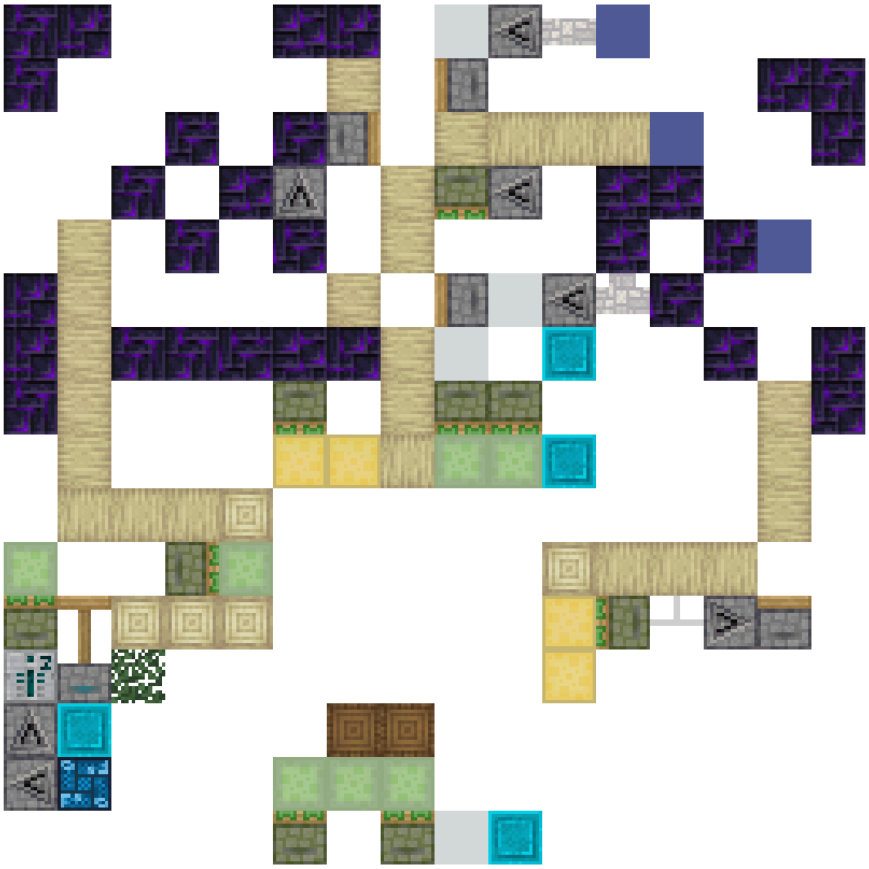
Architecture in tree farms generally refers to the **specific arrangement** of the tree farm structure. It can be divided into **core architecture** and **processing architecture** — *the latter covering block flow and drop collection modules.*

Core architecture:



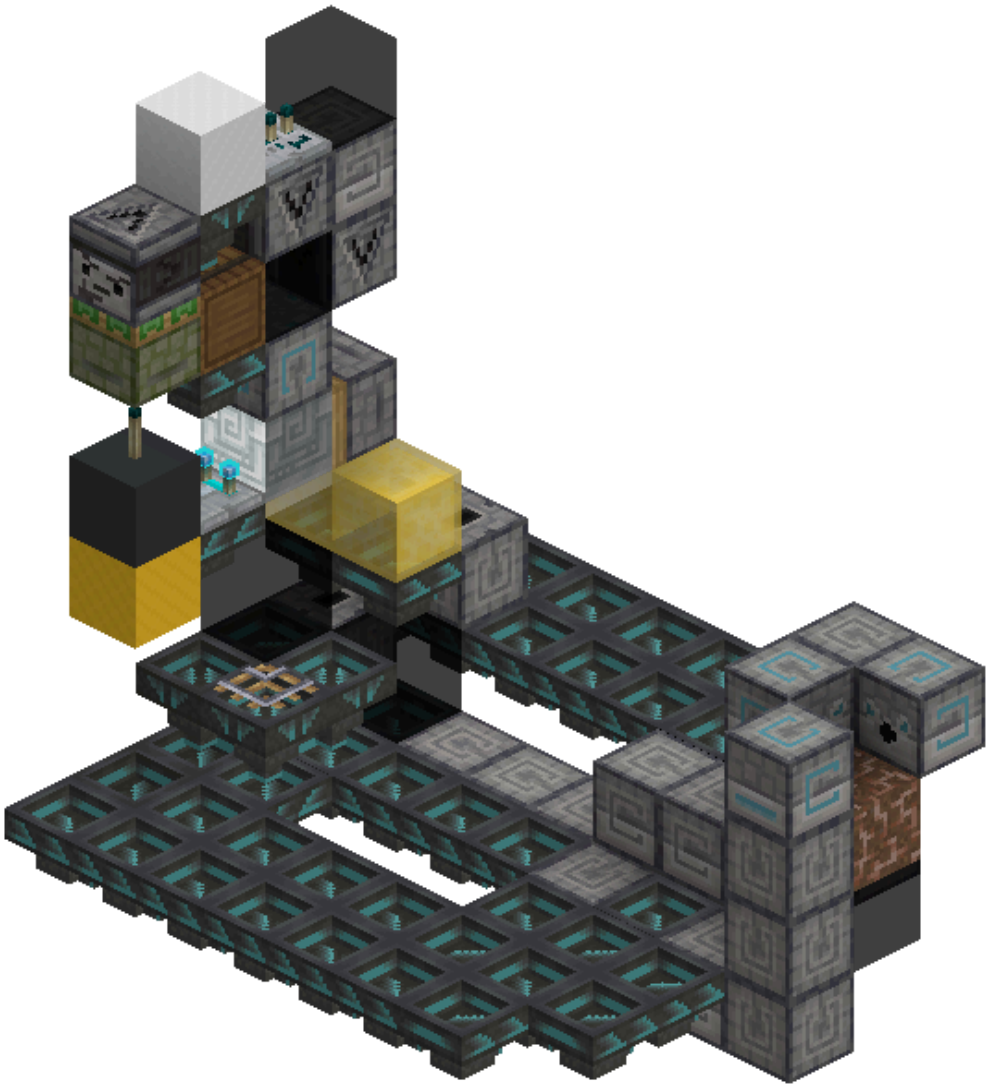
@Chuo_di

Processing architecture:



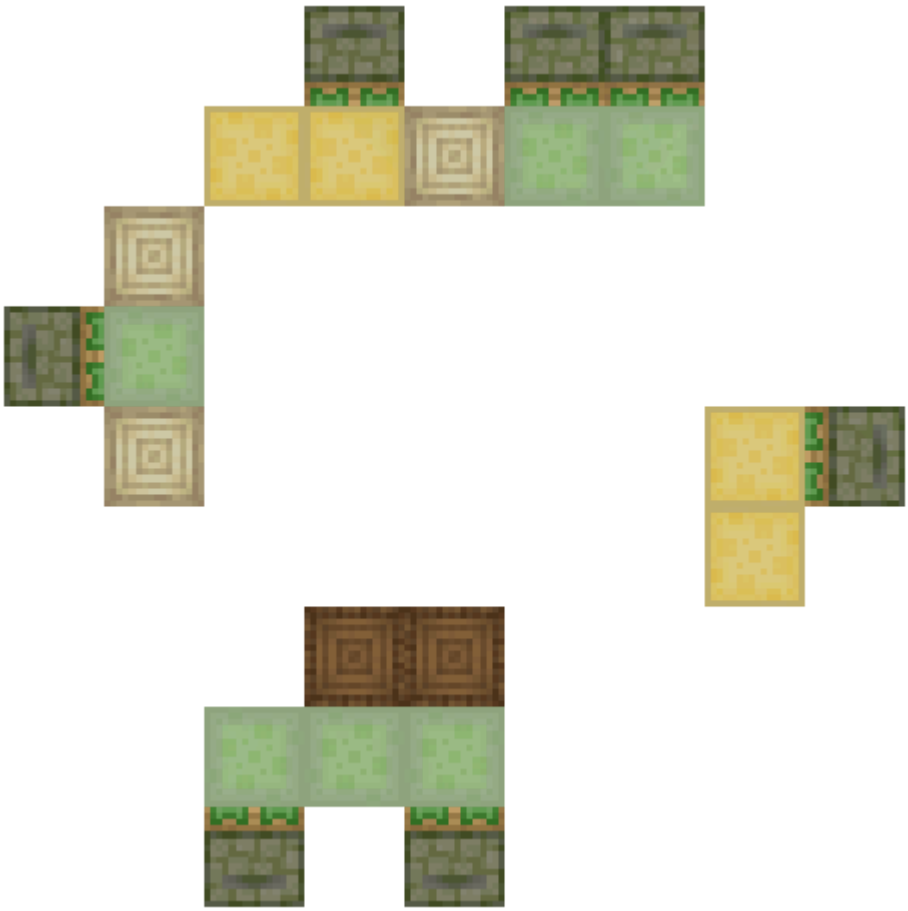
@Chuo_di

Core architecture generally includes **bone meal architecture**,

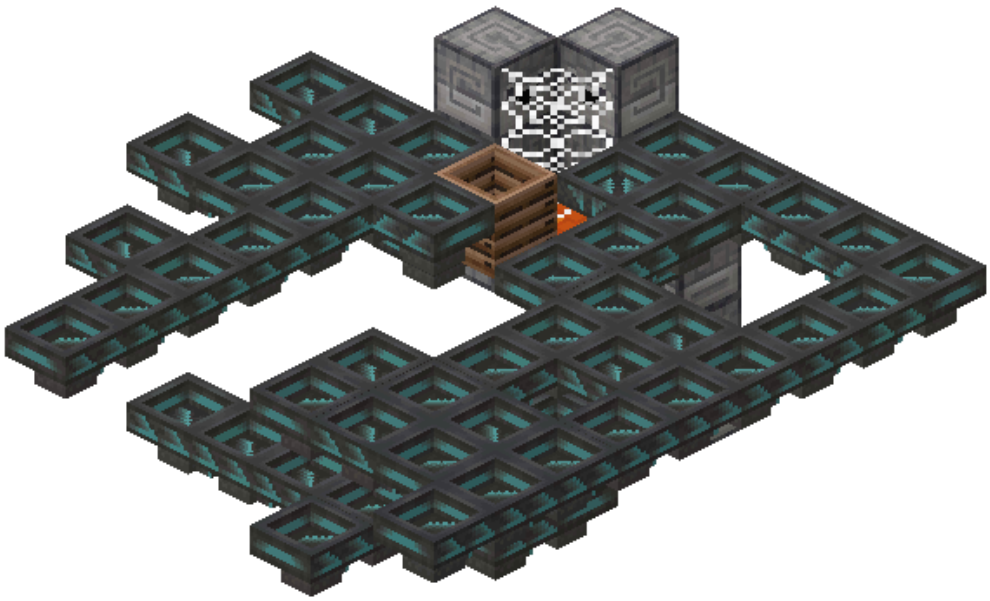


trunk processing architecture,

leaf processing architecture,

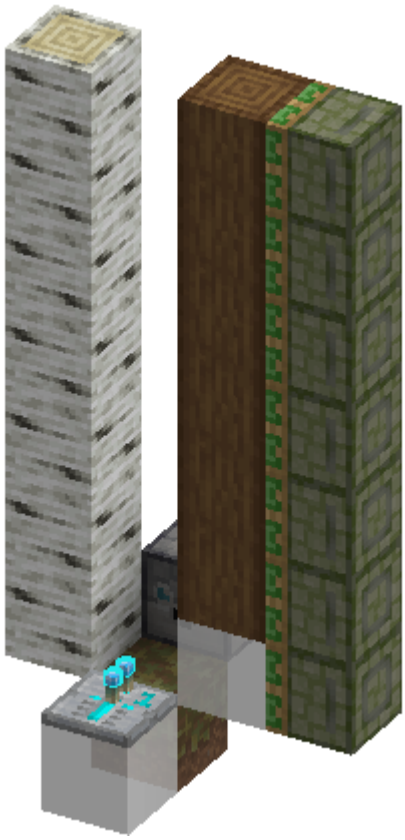


and **sapling recycling architecture**,



Generally, the **bone meal structure** or **trunk processing architecture** is designed first, followed by the **leaf processing architecture** and **sapling recycling architecture**.

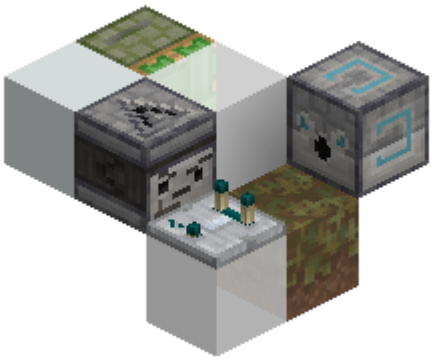
The basic form of trunk processing architecture was covered in the previous chapter. Here we choose **direct push** for trunk processing (*trunk processing that uses a push-out structure is called "main push"*), with the root processed together with the trunk. For bone meal architecture, we use a **single 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.

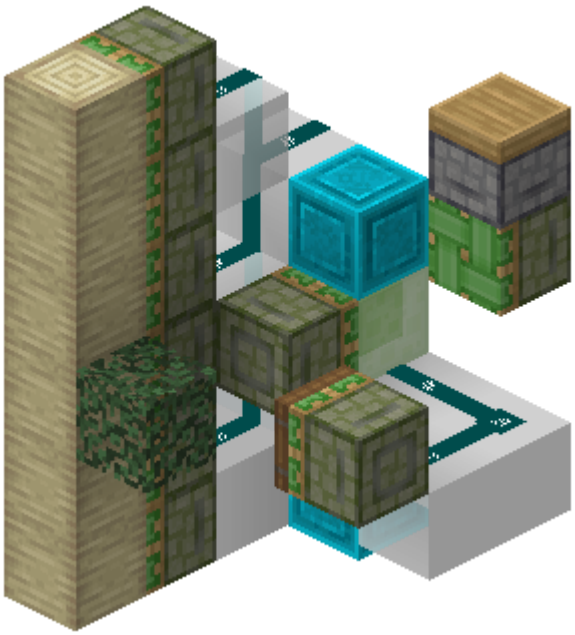
Basic detection methods include: **Comparator detection**, **QC detection**, **BUD detection**, **push limit detection**

Comparator detection:

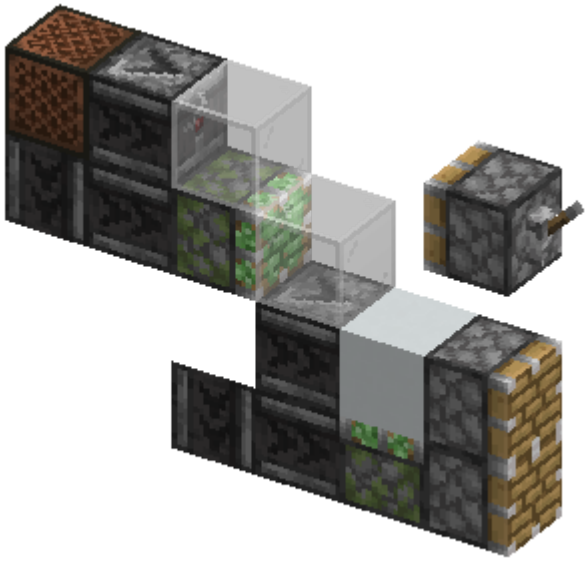


QC detection:

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.

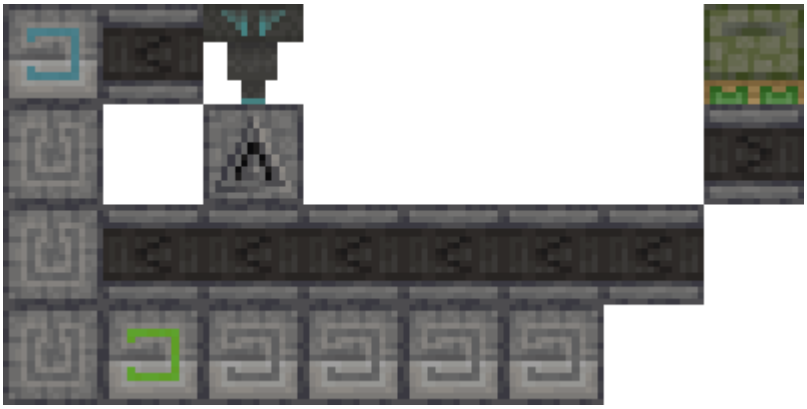
For processing architecture, we use a simple **TNT explosion chamber**.

> *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

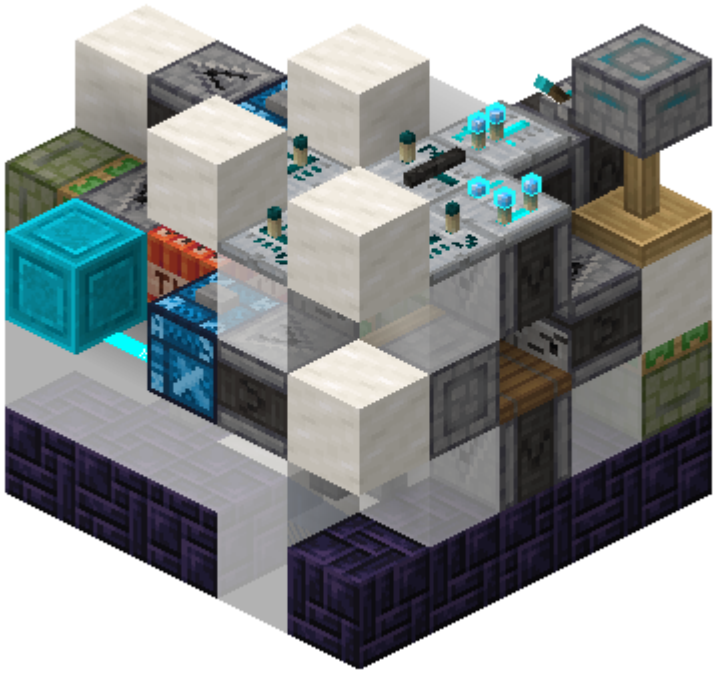
The basic idea is to connect the various parts of the tree farm without affecting its timing.

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~~

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.*

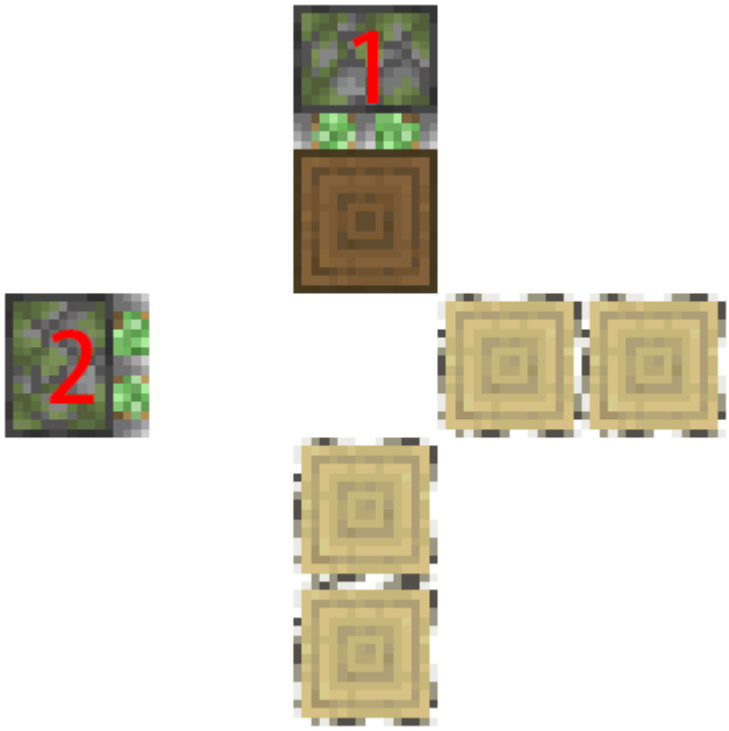
1 4GT BIRCH

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.

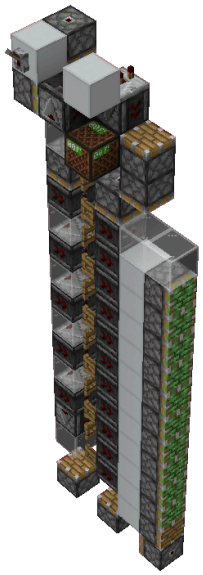
2.1 OBSERVER-RELATED 0T GENERATORS

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.

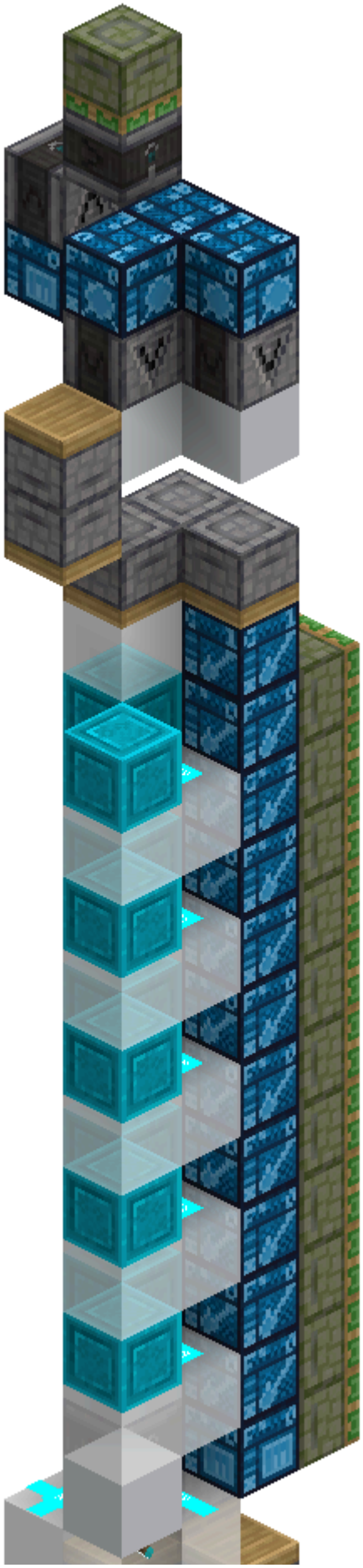
> *Actually, we can do some more optimized processing combined with the wiring of the tree farm itself. Anyway, just give the upward pushing piston a deeper NC update.*

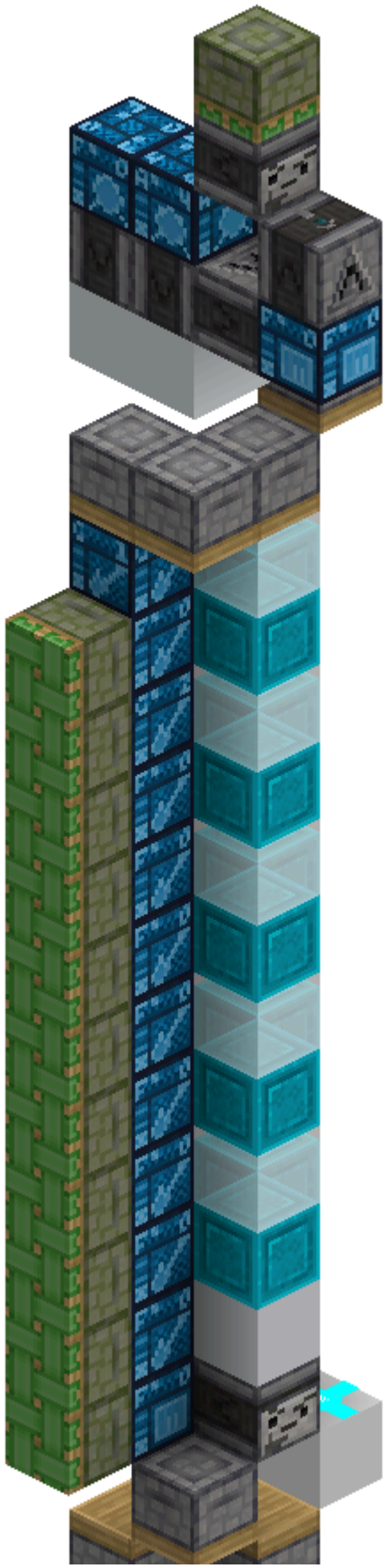
> *In fact, the rising edge depth is not uncontrollable. We only need to push in a powered block at 0t during the BE phase, and we control the rising edge depth of the power. But this method has a serious problem: it is too large. Many times, if the architecture design does not happen to allow inserting 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.

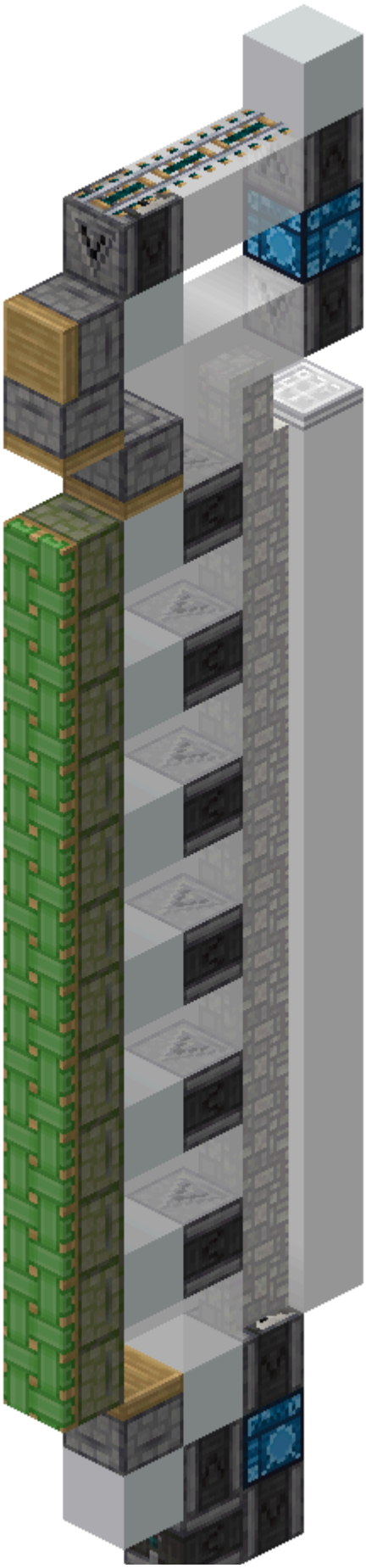
Redstone dust redirection is **the most convenient 0t method for controlling the rising edge signal depth of 0t** in our dustless 4gt wiring.

> *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

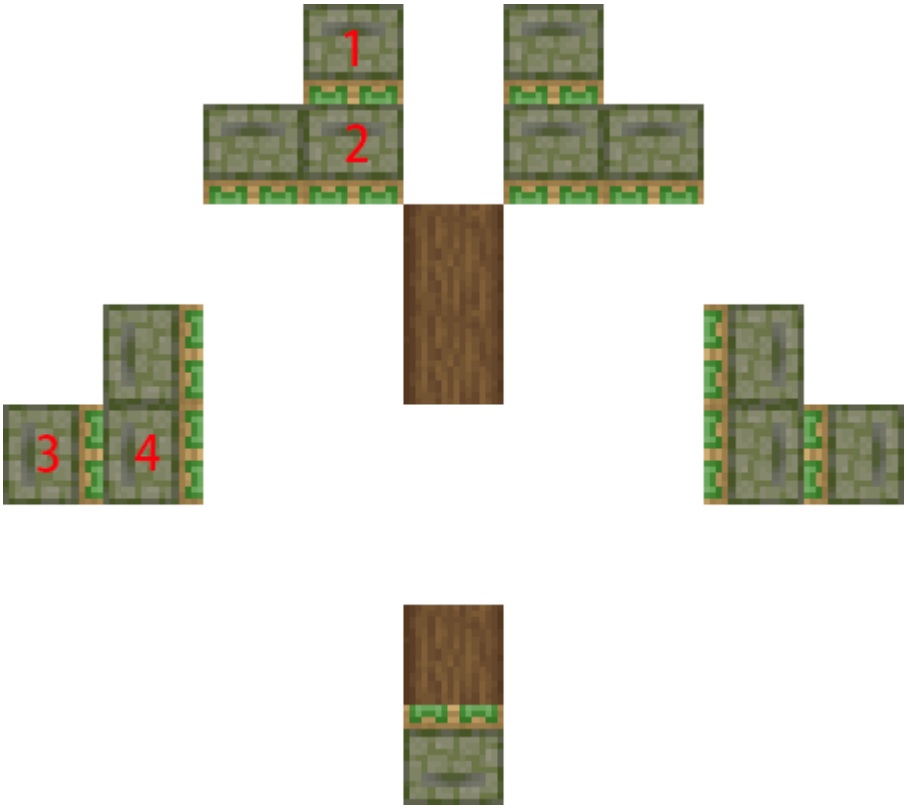
> *Actually, if you look carefully at Scorpion's wiring, you will find: we do not necessarily have to use 0t. The 2gt signal given by the observer is also usable. This is also a quite important technique for reducing lag in 4gt tree farms. Folen's 4gt multi-species still suffers from lag compared to 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~~.

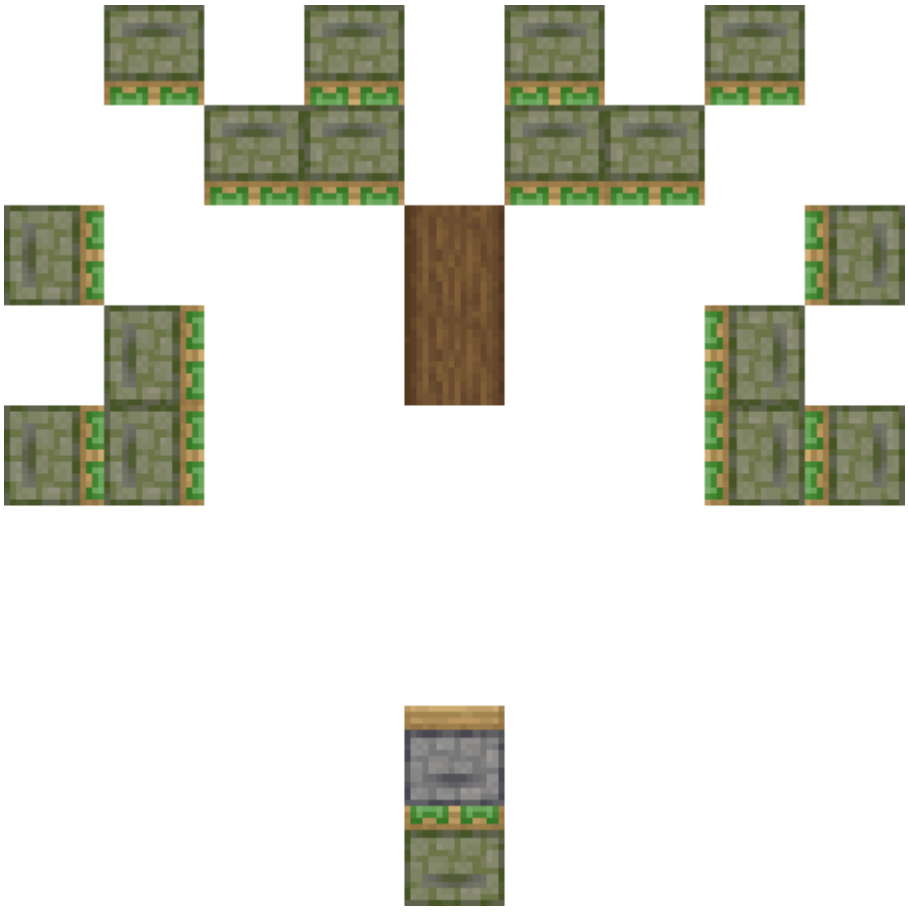
> *Actually, if you remember, you will find: this is the timing TT originally designed for 6gt jungle. But the earliest 4gt jungle tree farm did not use this timing. We will mention the reason soon.*

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.

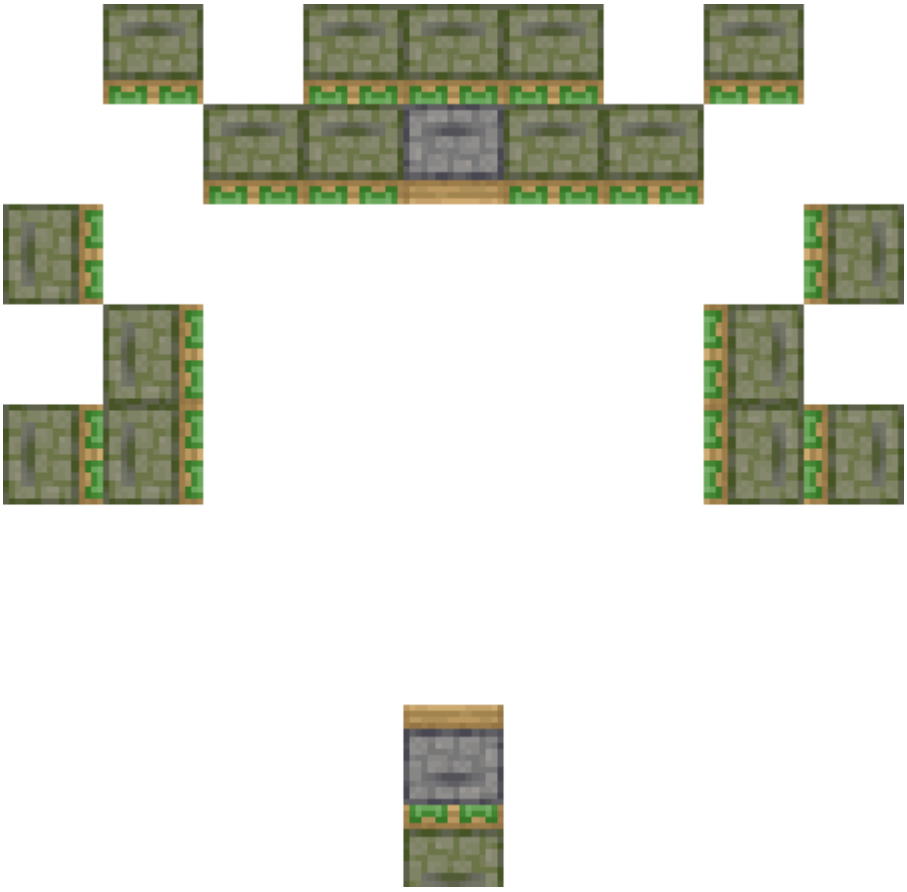
> *To increase branch processing capacity, we can actually replace the poor piston at the bottom with double recursion (you can think about other architecture designs yourself. The standard answer is plexi's design).*

> *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.*

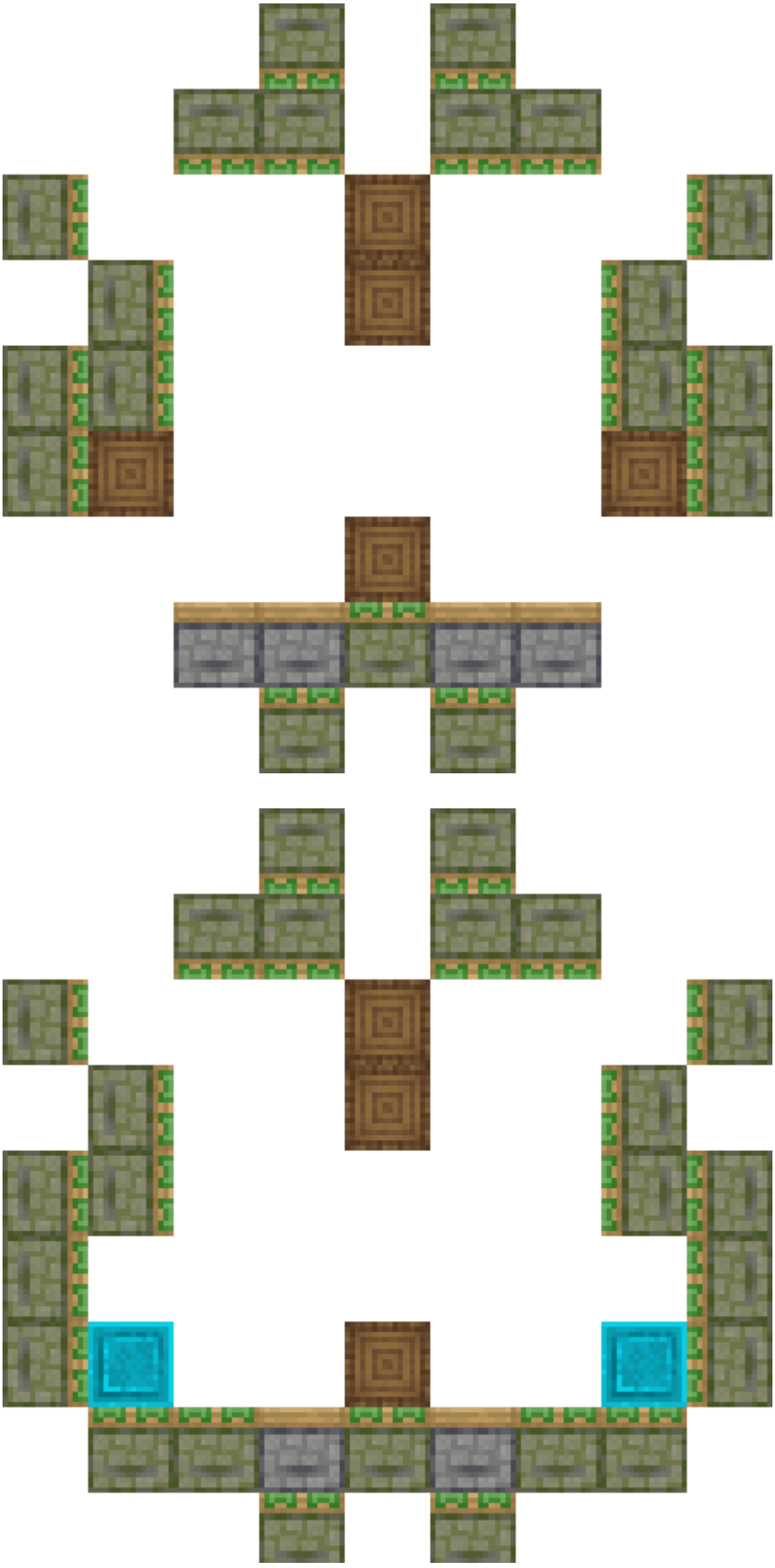
Plexi came up with a good solution: **if it cannot push, just pull the piston away directly**, thereby removing the push at 3gt, thus avoiding throwing out the regular piston. Theoretically there should be quite a few implementation methods. You can try more yourself, ~~although Xinghe and plexi, qontrol, lintex argued for a long time and only the wiring architecture plexi had done before could work~~.

3.3 JUNGLE WOOD

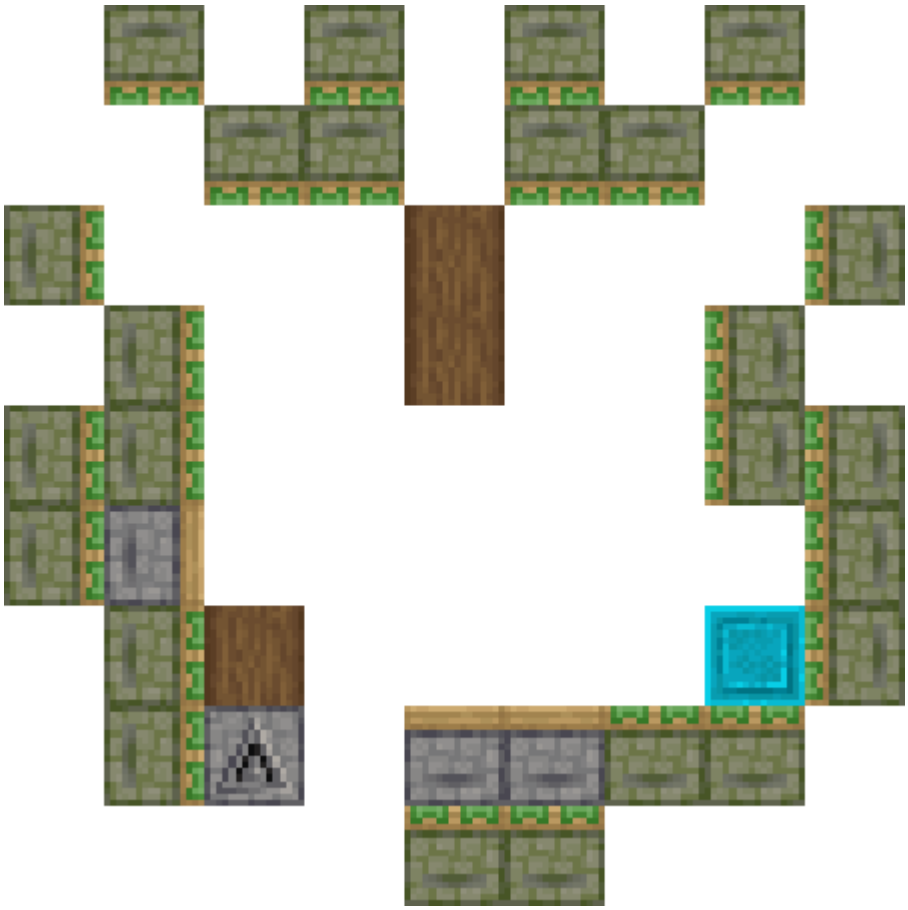
Welcome to the actual biggest boss in 4gt tree farms. Due to jungle's 1/40 sapling drop rate, 4gt tree farms actually only run half of each cycle. Jungle sapling circulation has become the biggest obstacle for 4gt multi-species ~~I'm sure you still remember Luoxi's extreme 100.5% recovery rate back then~~.

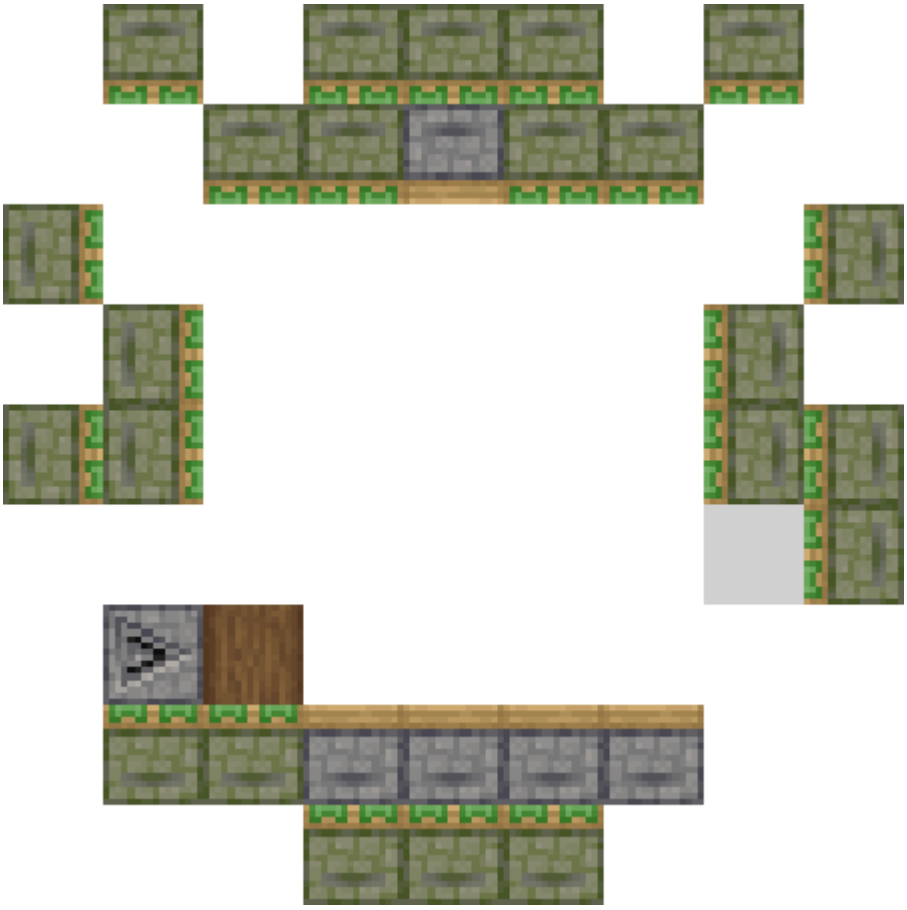
We mentioned above that in 4gt multi-species, we abandon TT's original timing for jungle, and **synchronize the pseudo-double-recursion used to extract the trunk on one side with the pseudo-double-recursion on the opposite side**, so that more leaves can be processed in the same cycle, thereby obtaining 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.

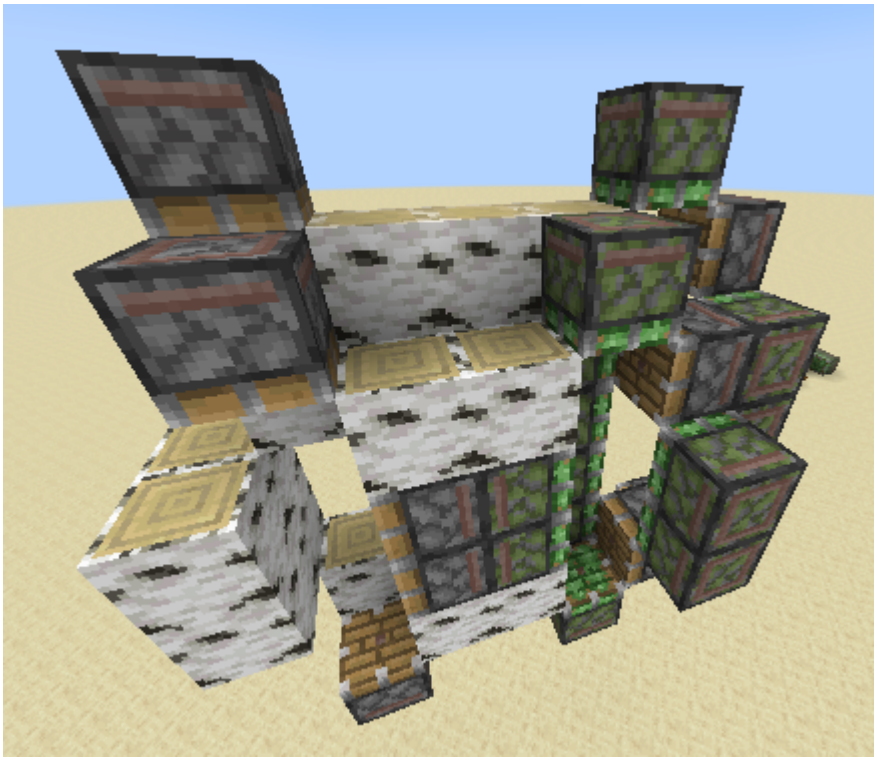
> *In fact, you can also imitate perfect timing tree farms and shut down part of the structure, but since modern 4gt architecture has strong integrity, ~~shutting down is almost the same as not shutting down~~, so it's just made lighter.*

3.4 LOG OUTPUT—SUCTION-TO-PUSH

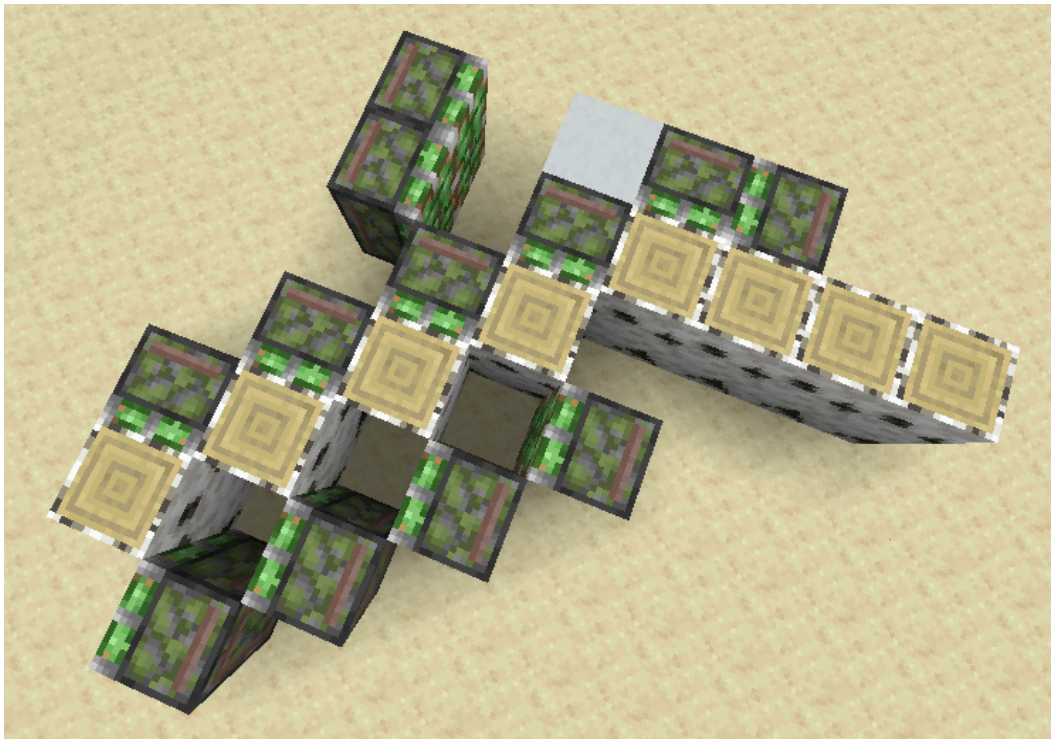
If you look carefully at the current tree farm, you will find a very serious problem: **every step of the block stream is pulled by an entire row of sticky pistons**. Such a block stream obviously cannot be processed. So here we need to fill a hole: log output for pure retraction-based pseudo-double-recursion 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)) ↩

> *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 bonemealed 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)

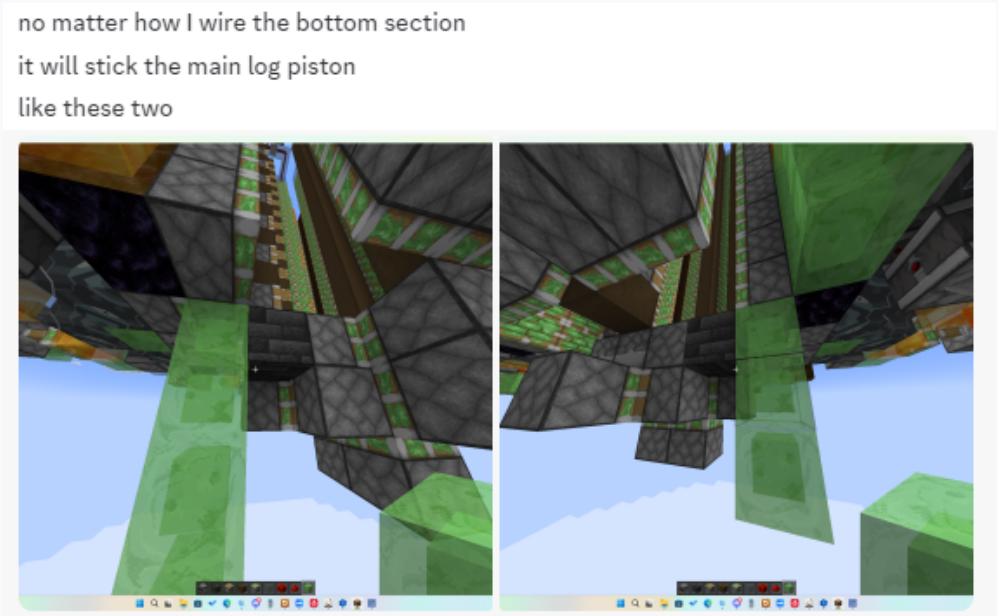
> *Originally when we started the project, hampter said he would make a corner side-suction for 8gt large spruce, but his computer broke midway, ~~and his unit still hasn't been released to this day~~*

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

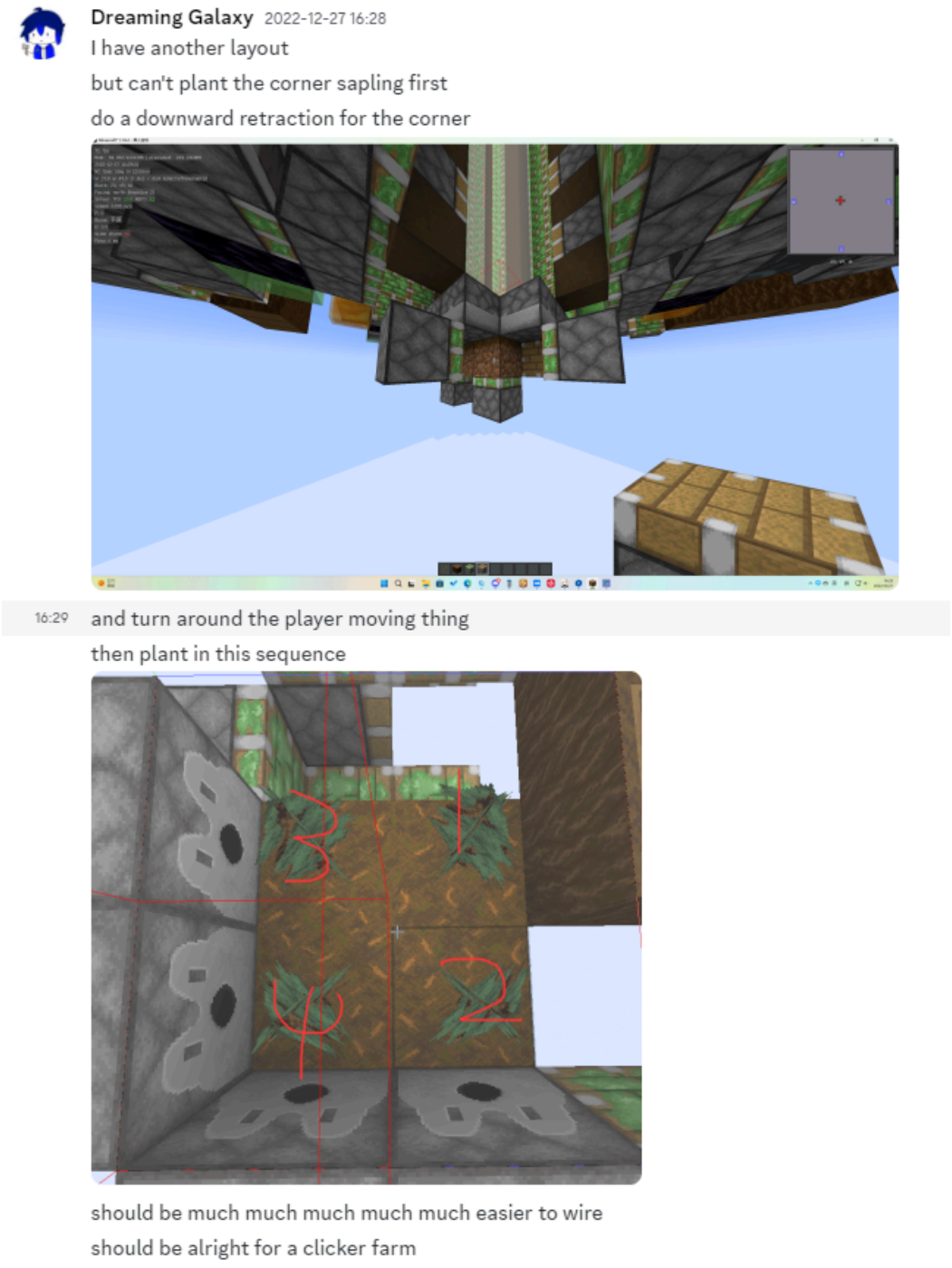
4.1 "NO MATTER WHAT YOU DO, IT STICKS TO THE MAIN SUCTION PISTON"



~~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

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.

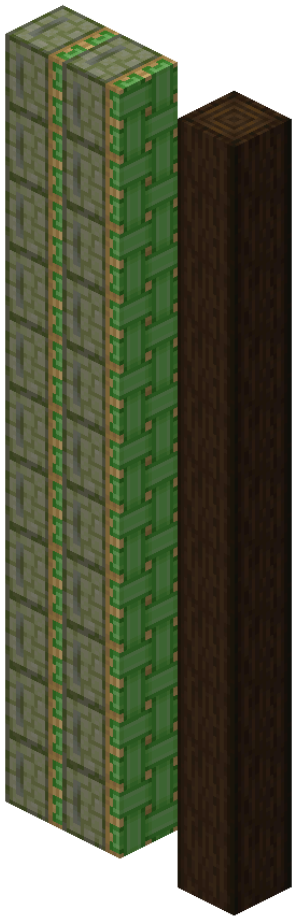

Basic bonemealing structure

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.

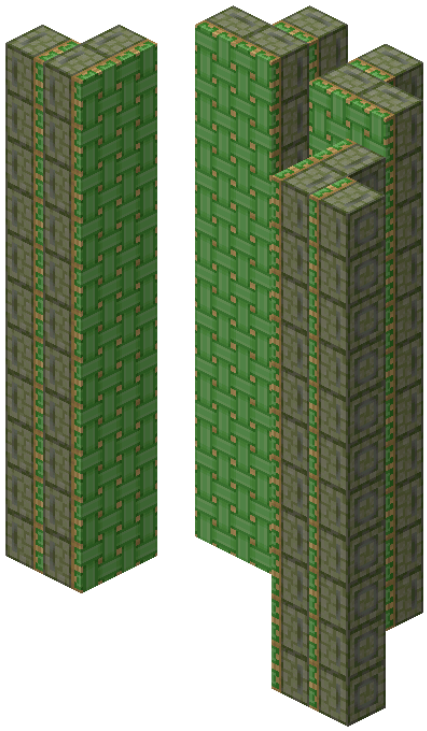
Triple push extension:

Pseudo double push extension:



Honey-slime wall double push extension:

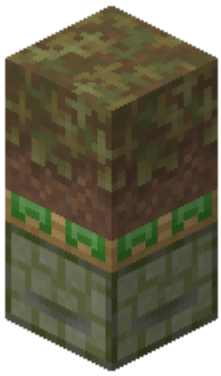
Retraction-type pseudo double push extension (a design where the secondary Piston only performs retraction (0t), shaving one Piston action off the timing):



> *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**.

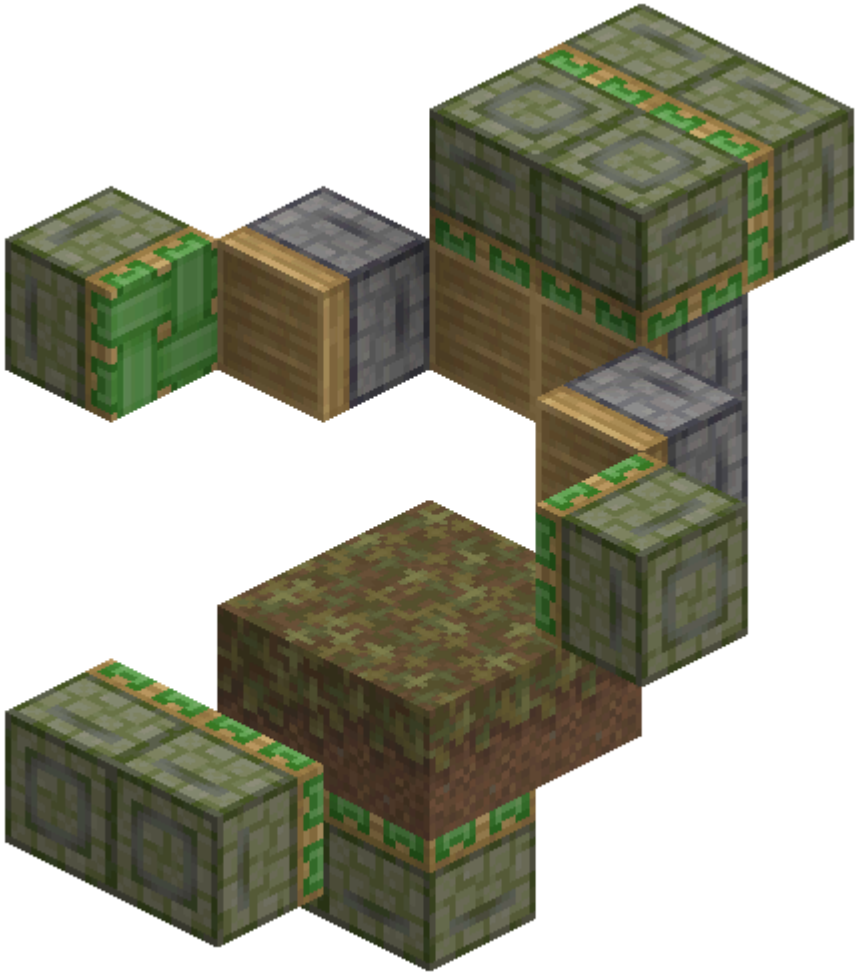
Upward push:



Downward pull:

Side pull:

For dark oak, an **upward pull** method is sometimes used.

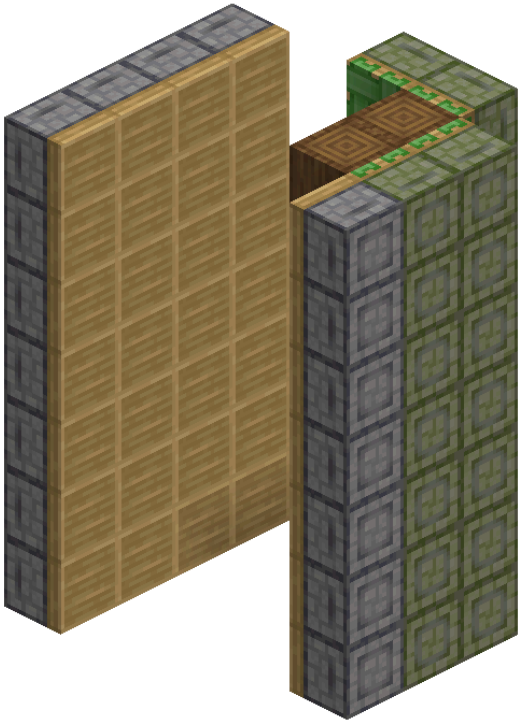


2.3 LEAF PROCESSING

Also straightforward: use Pistons or honey-slime walls to push away **enough leaves** to get the saplings you need.

Honey-slime wall:

Piston:

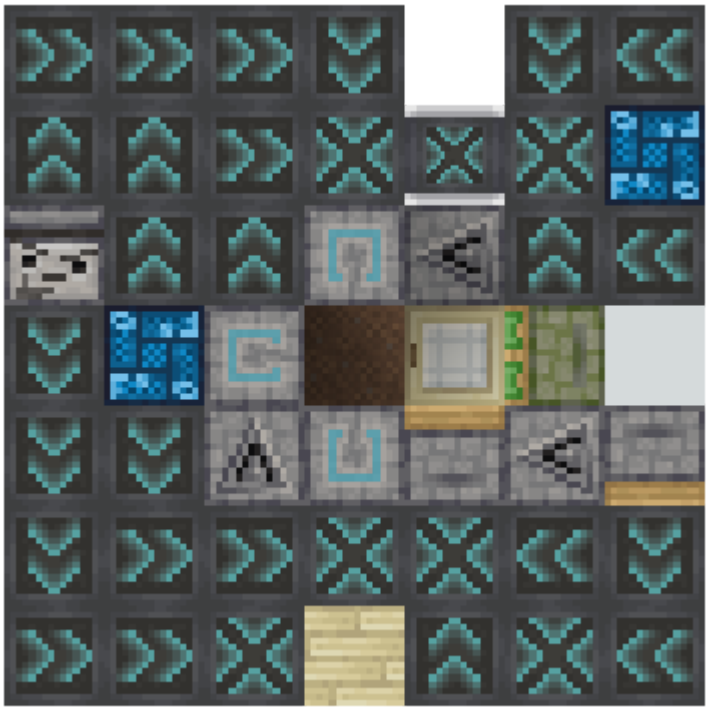


Each leaf block has a fixed chance to drop a sapling: jungle 1/40, all other tree types 1/20.

2.4 SAPLING RECYCLING

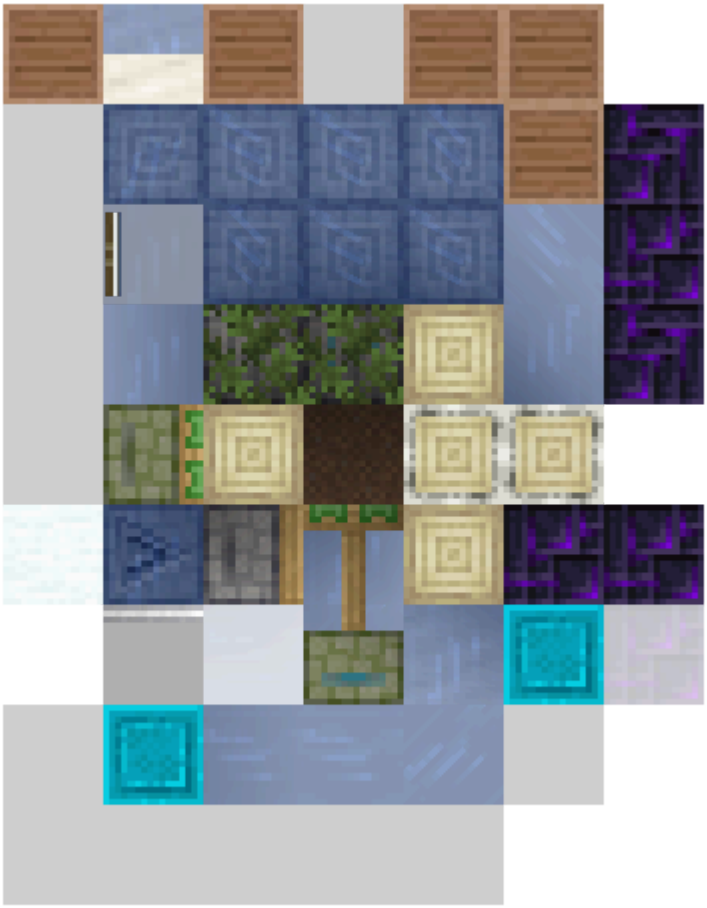
Saplings dropped by the leaf processing module are collected near the player using **Hoppers**, **water flow**, or similar methods.

Hopper:



@Scorpio 天蝎君 4gt birch

Water flow:



@Feng_Bl, Sunflower_Lin 6gt birch

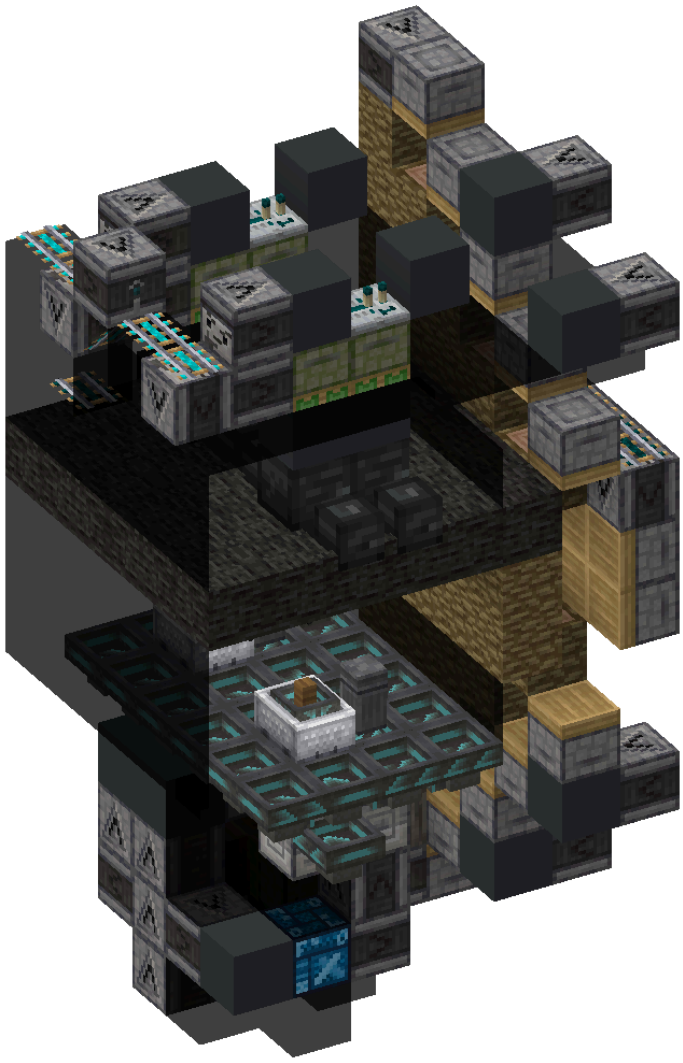
In tree farms with fast processing cycles and tight internal space, **Hopper Minecarts** are sometimes used for sapling recycling.

@Qonctrol, PUTF sharp

2.5 BLOCK-TO-DROP CONVERSION

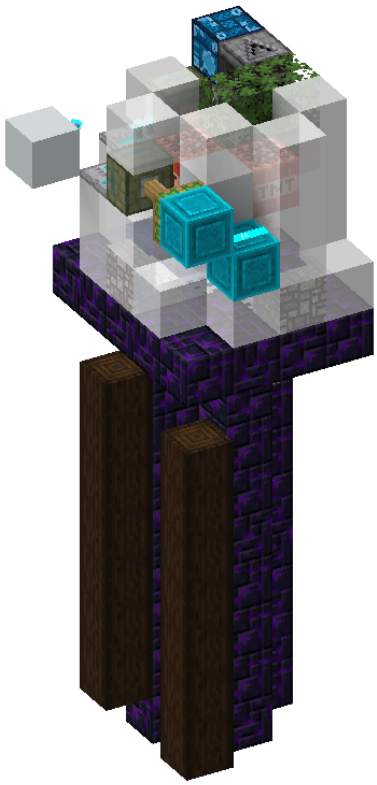
There are two main methods: **Wither** and **TNT**.

Wither:



@Scorpio 天蝎君, PUTF flat wither

TNT:



@Feng_Bl

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).

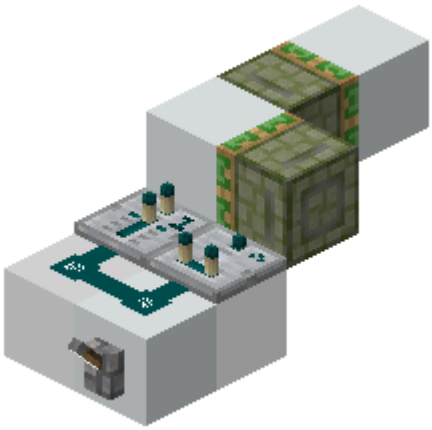
Obviously, it should extend when powered and retract when depowered. This completes a **0-tick action**. Specifically, for sticky pistons, the first block it pushes arrives **one piston depth deeper than the falling edge signal**, while pulling behaves normally.

> *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

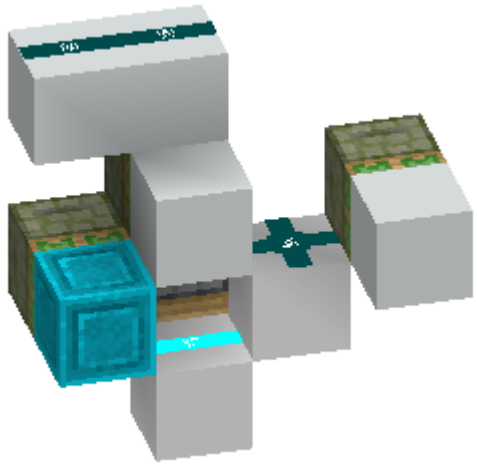
The idea is very simple: **before BE ends within the same gt**, give the piston a rising edge at an earlier point and a falling edge at a later point, and we get a 0-tick action.

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:

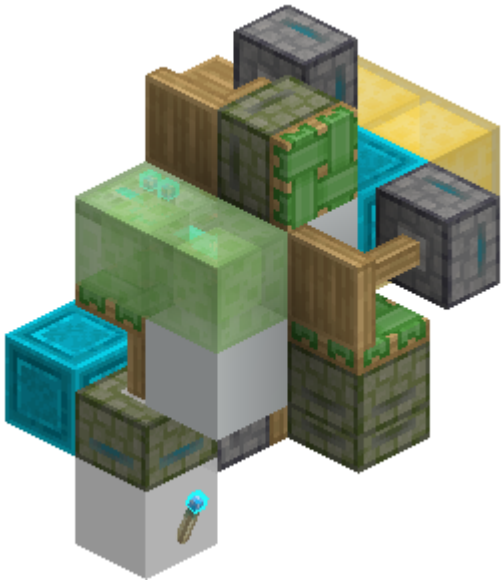
> *Trunk processing generally has four methods: processing together with the trunk, **upward push**, **downward suction**, and **side suction**.*

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:



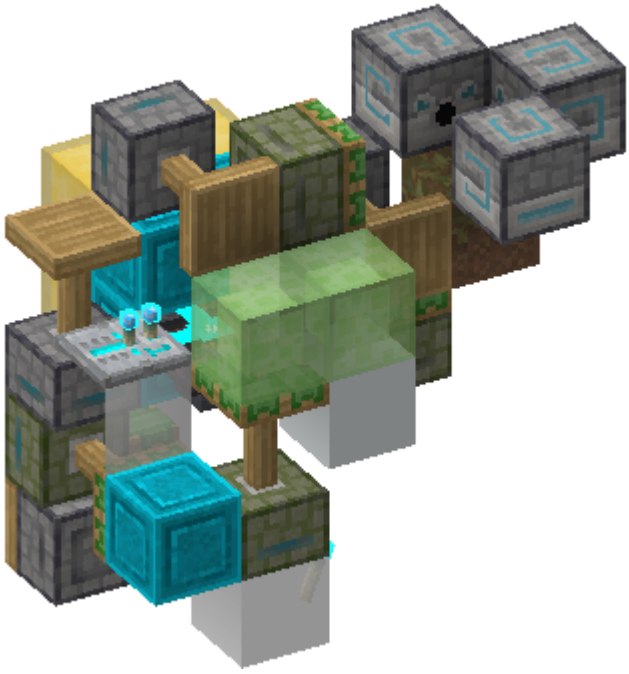
@Qonctrol, Dreaming_Galaxy, BFladderbean, Feng_Bl PUTF std2

> *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).



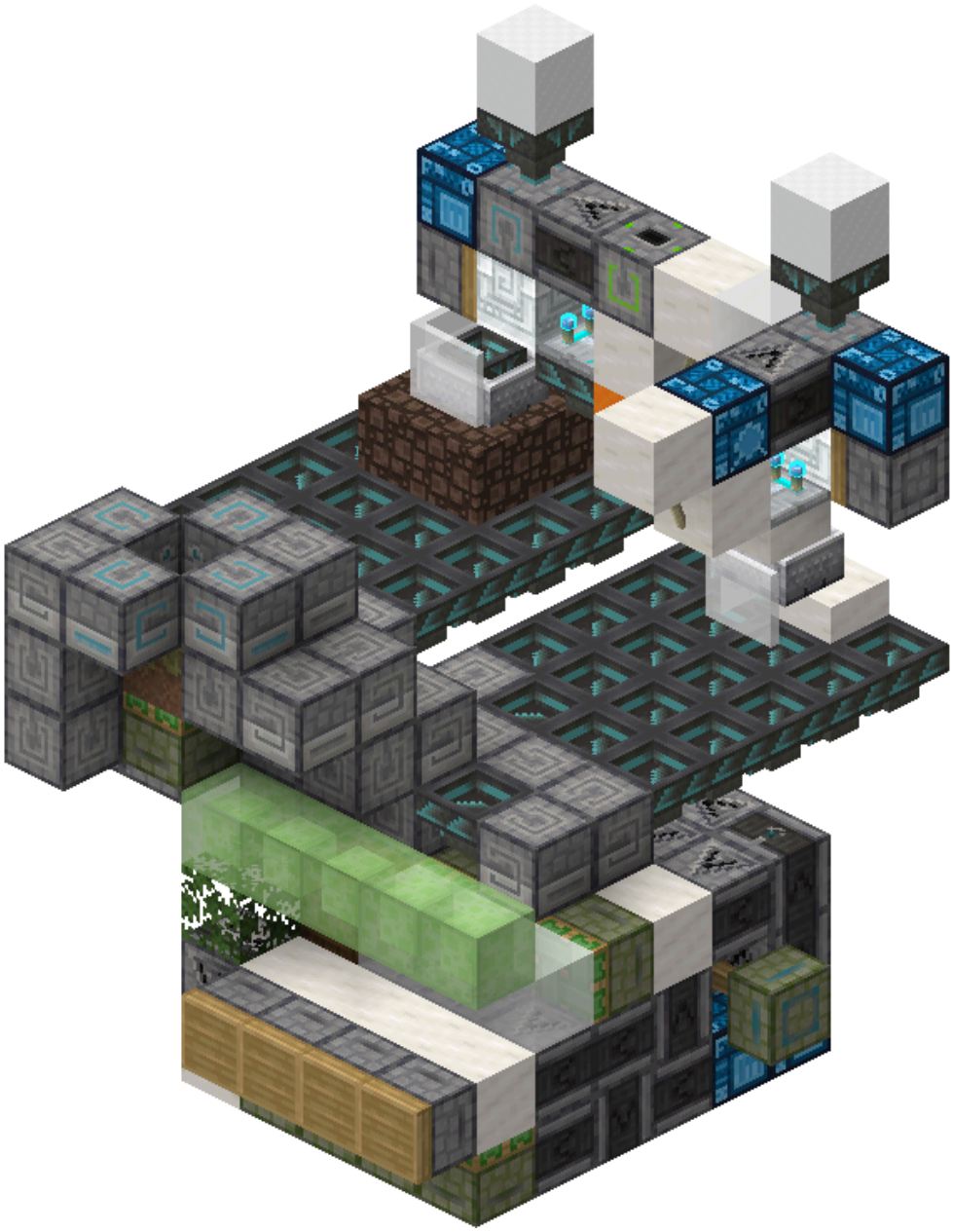
@PUTF std2

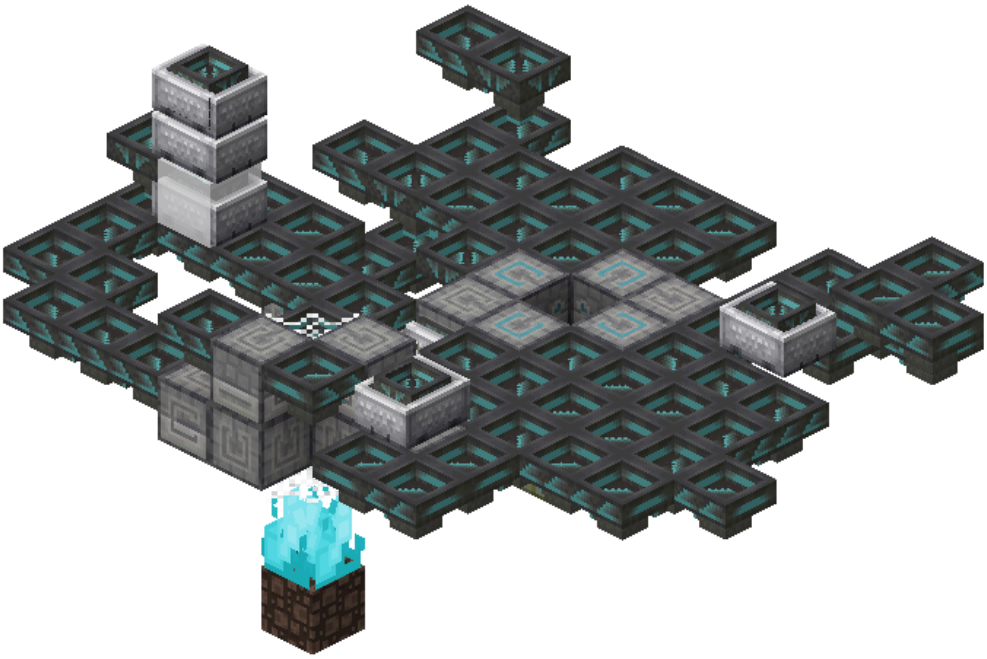
@Scorpio 天蝎君

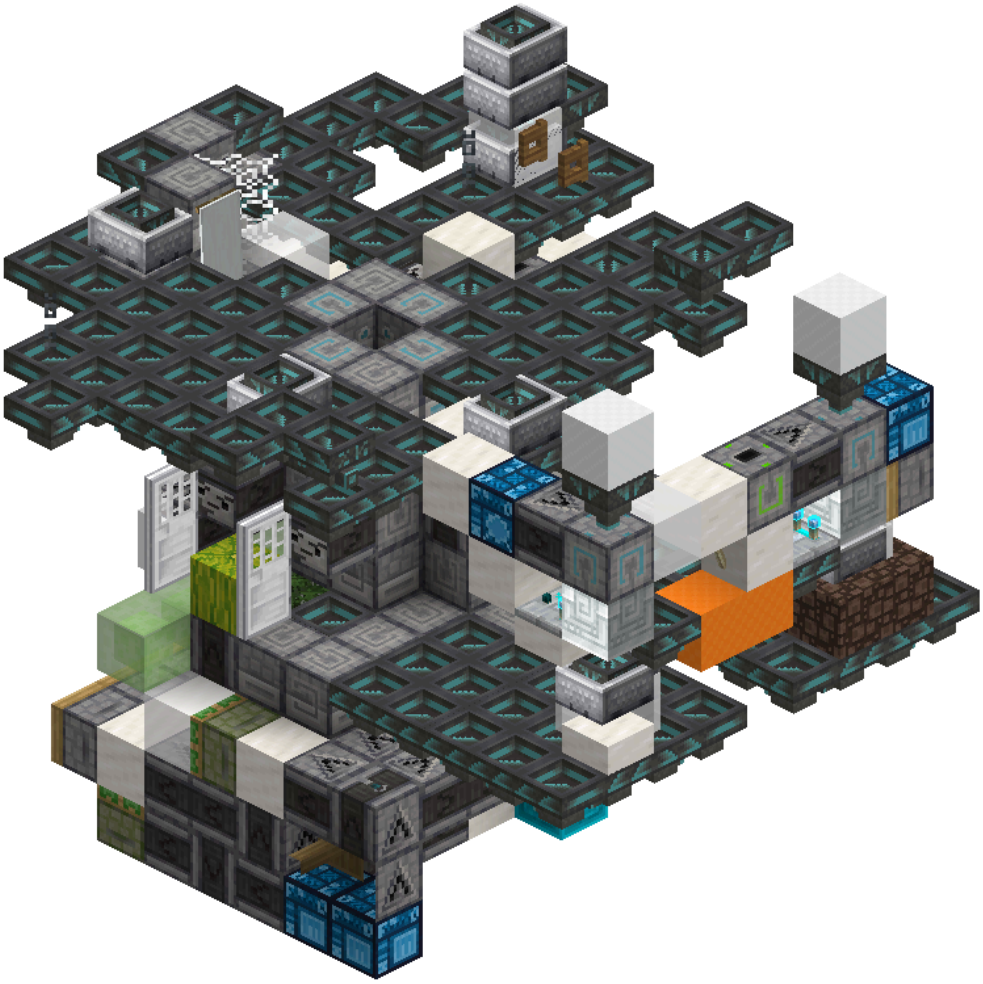
Self-explanatory cross bonemealing/synchronized bonemealing switching device diagram.jpg

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:



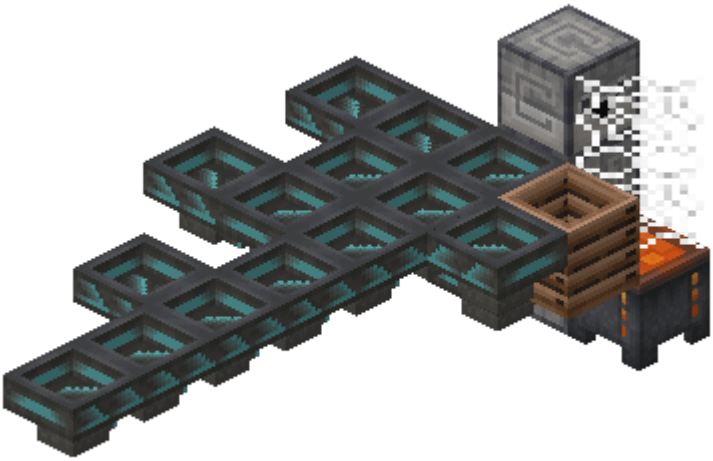


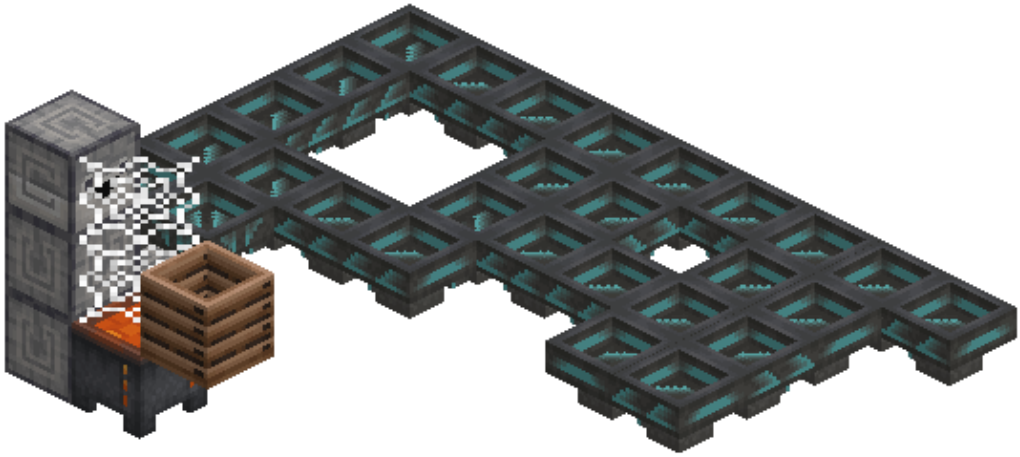


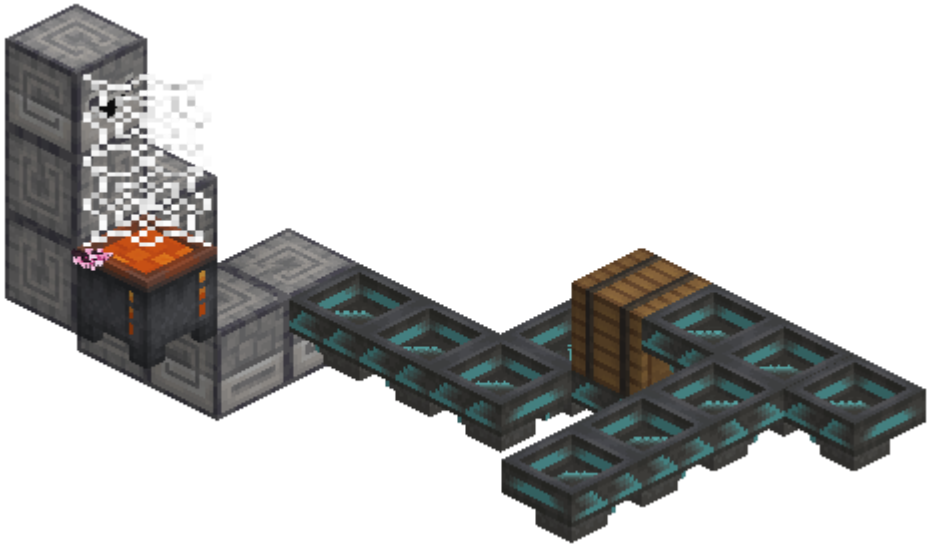
@Islyric, Qonctrol, BFladderbean PUTF+

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:



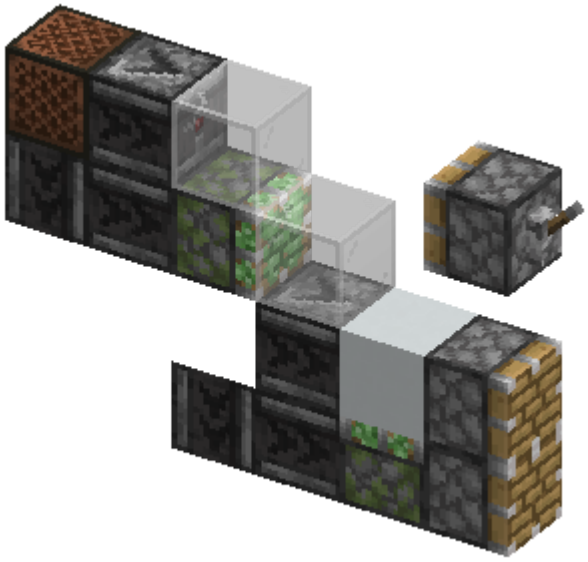




@Chuo_di, Islyric, Qonctrol, BFladderbean, Sarumi_QAQ PUTF std2c

> *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.*

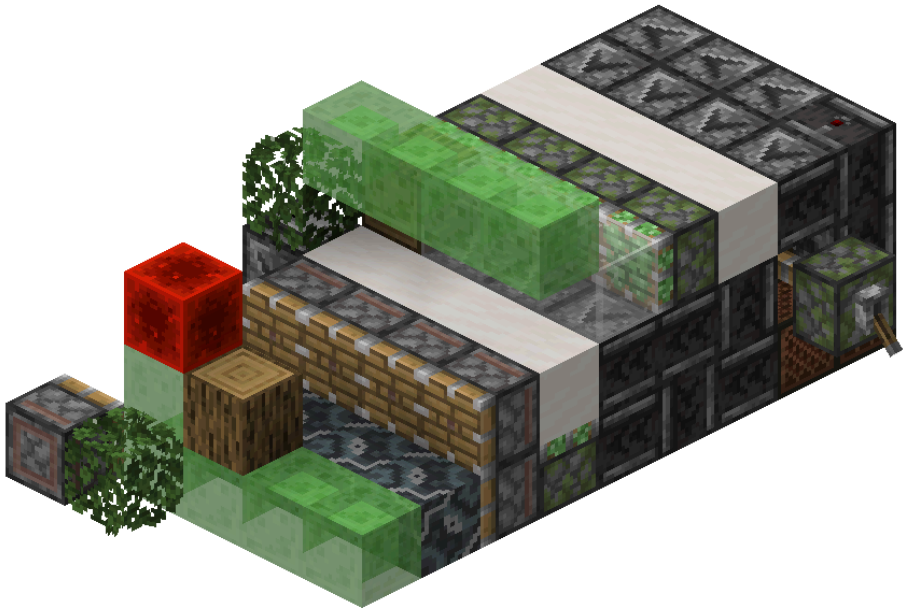
2.3 PUSH-LIMIT DETECTION



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

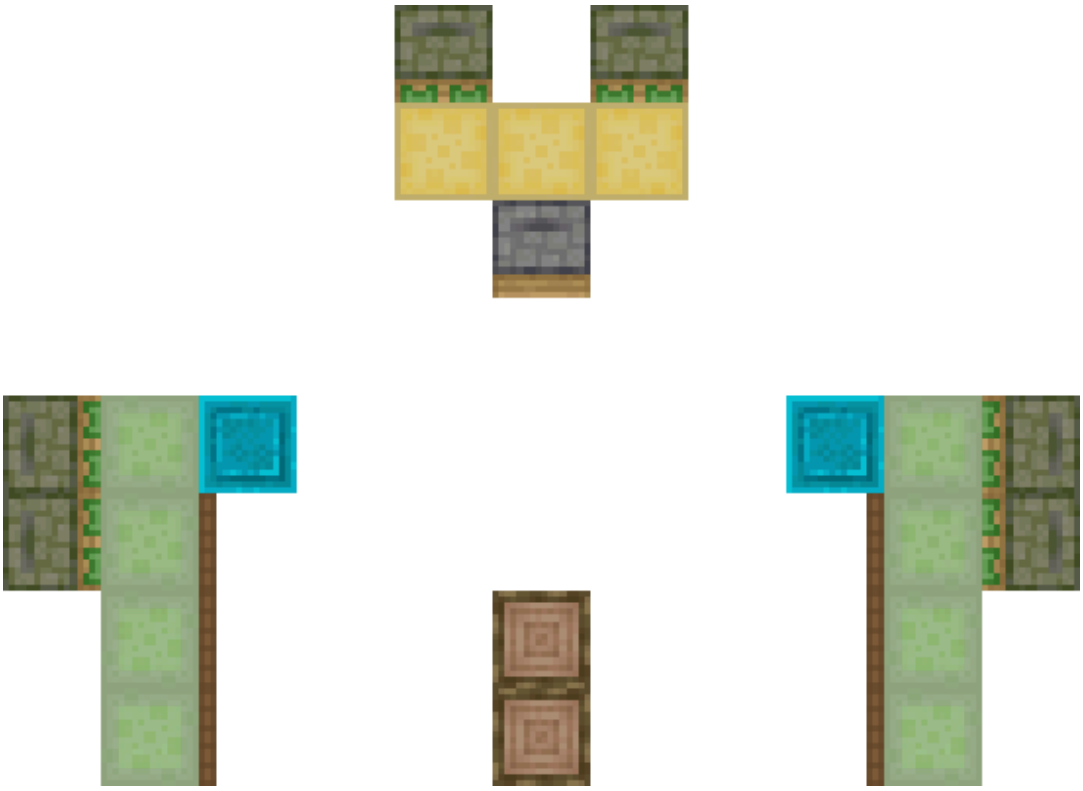
> *This is the part that determines the minimum operating cycle of a tree farm. Think carefully about this.*

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

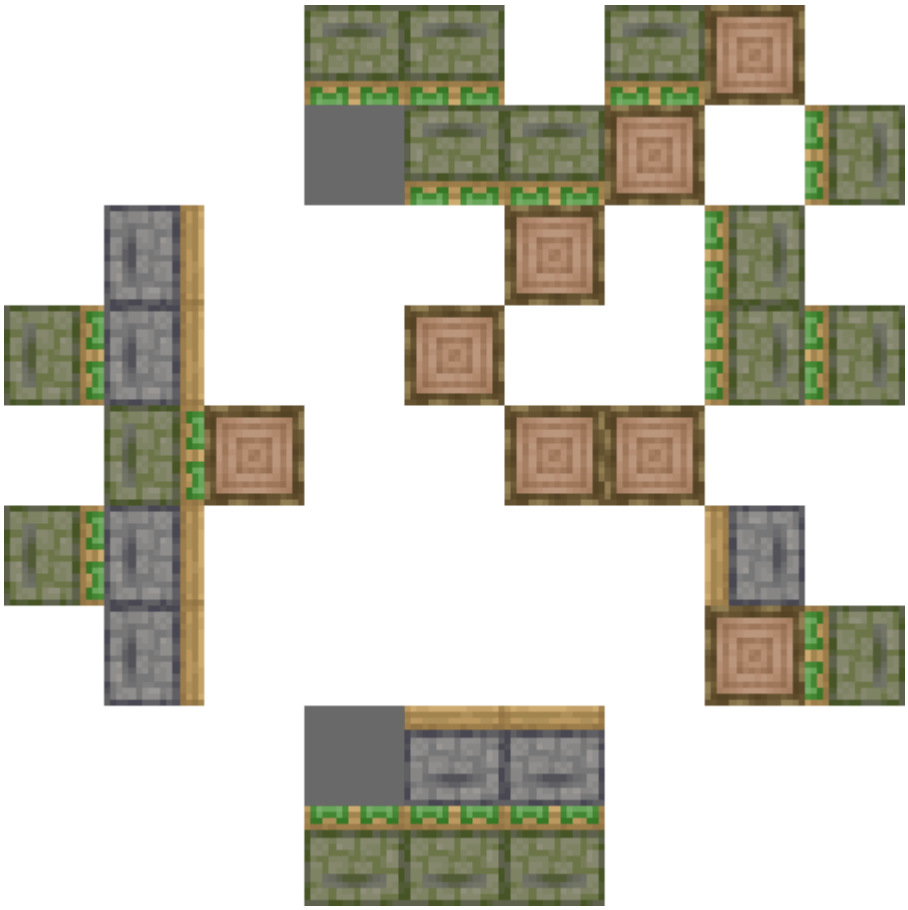
`18gt` Main push first level begins retracting

`21gt` Reset

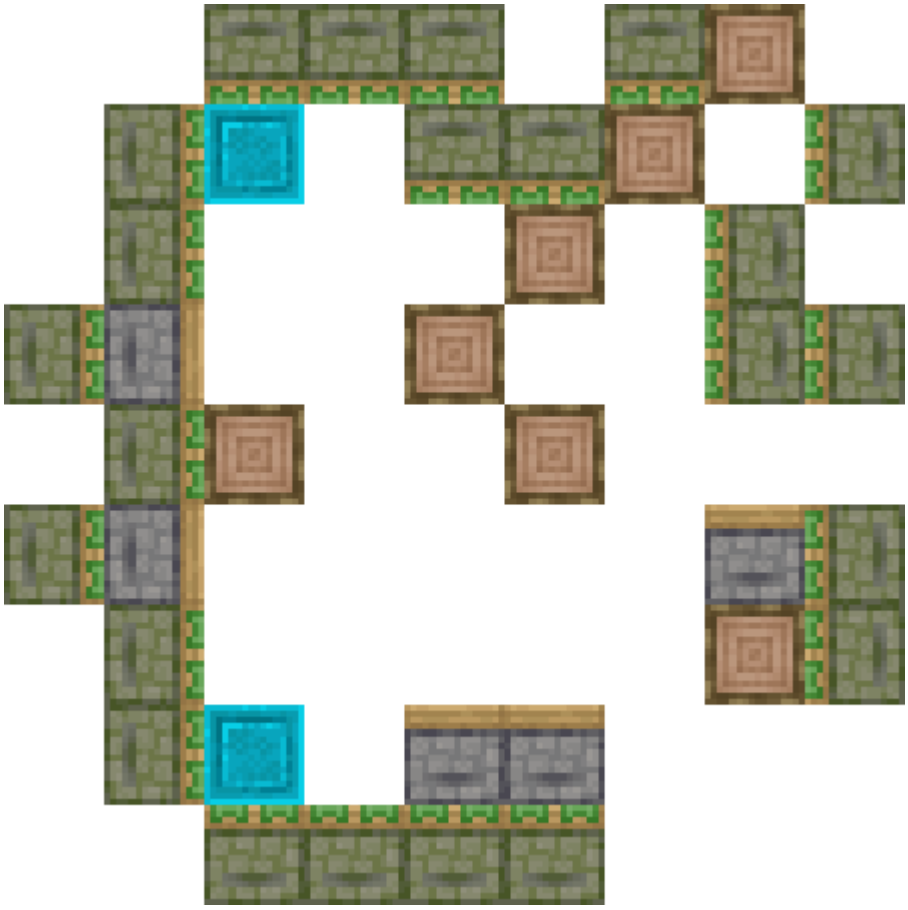
> *In tree farms, we can generally think of time in 3gt units, although this isn't strictly correct. In fact, this architecture's acacia timing only needs to satisfy: at 6gt the first side wall second level extends, and at 9gt one side wall retracts while the other extends, to complete a 21gt reset. 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

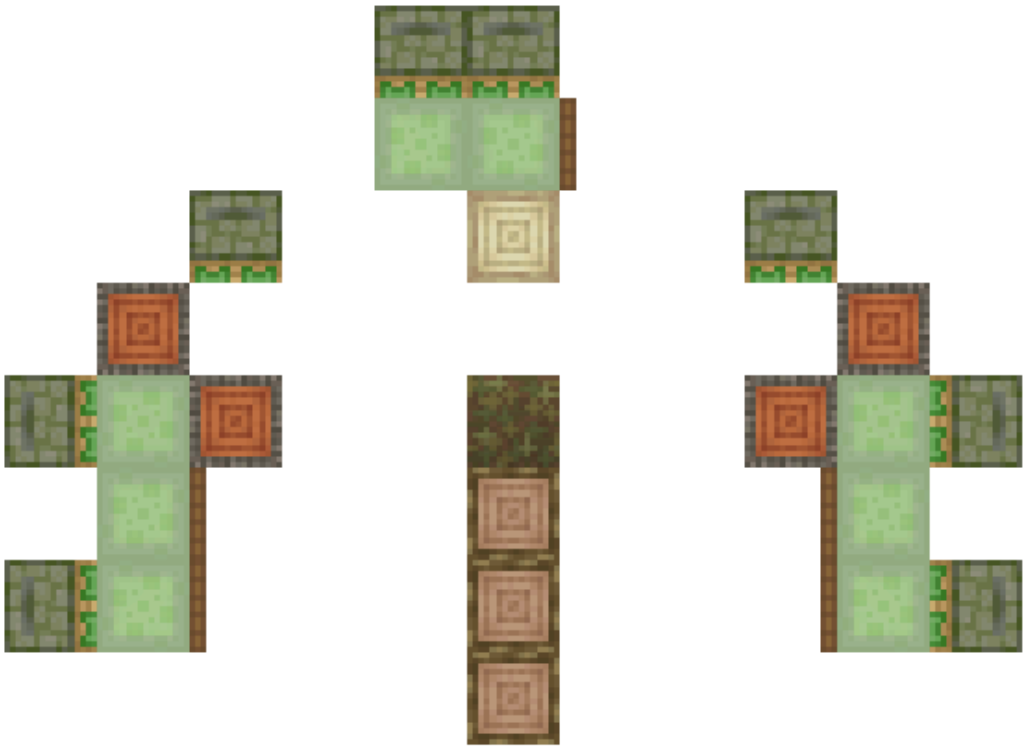
Performing timing design on a given architecture is very useful for wiring. It also helps you understand where the **culprit** behind a long operating cycle is, enabling targeted optimization.

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:

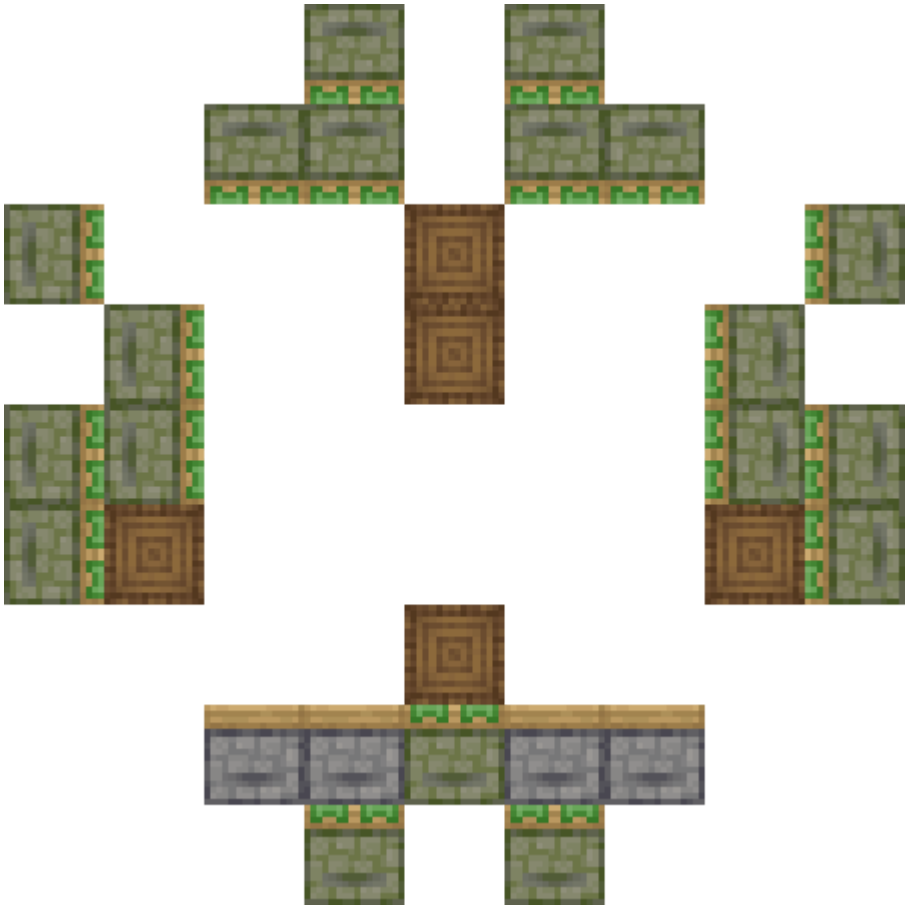
- Nutstory architecture: In 3.1.2 we mentioned that acacia side branches can be handled not only by centering but also by **increasing log output**. Now that we've learned timing analysis, you should notice: this architecture doesn't need timing for side branch centering, so it can run at 12gt, much faster than the PTHSUTF architecture.



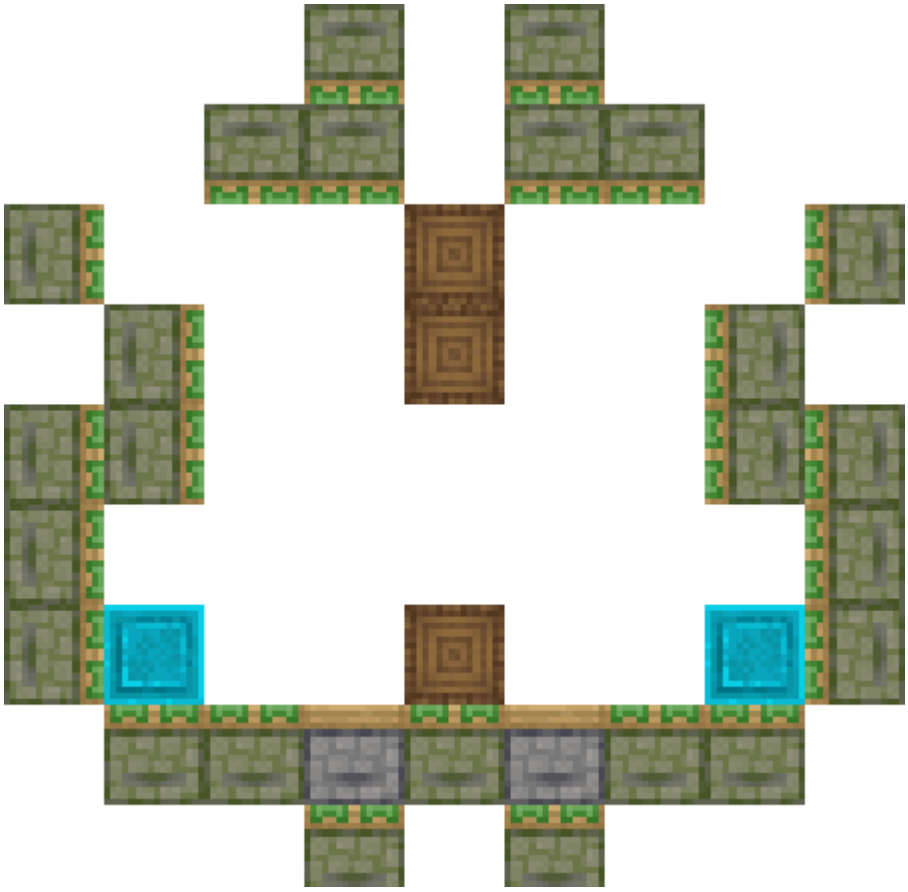
@Nutstory

- 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**.

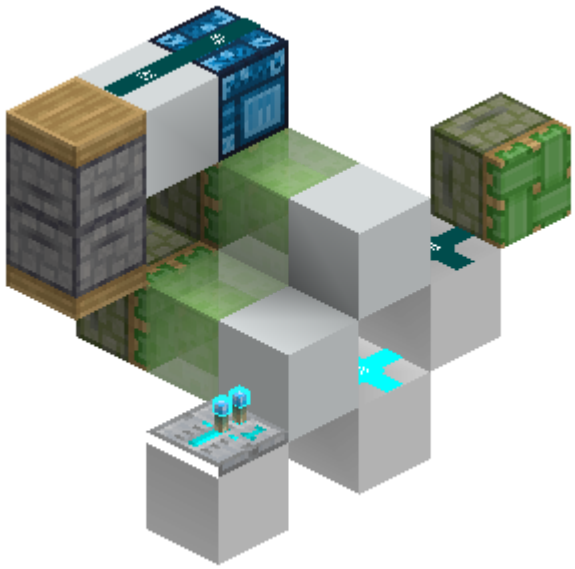
4 WIRING

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**.

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:

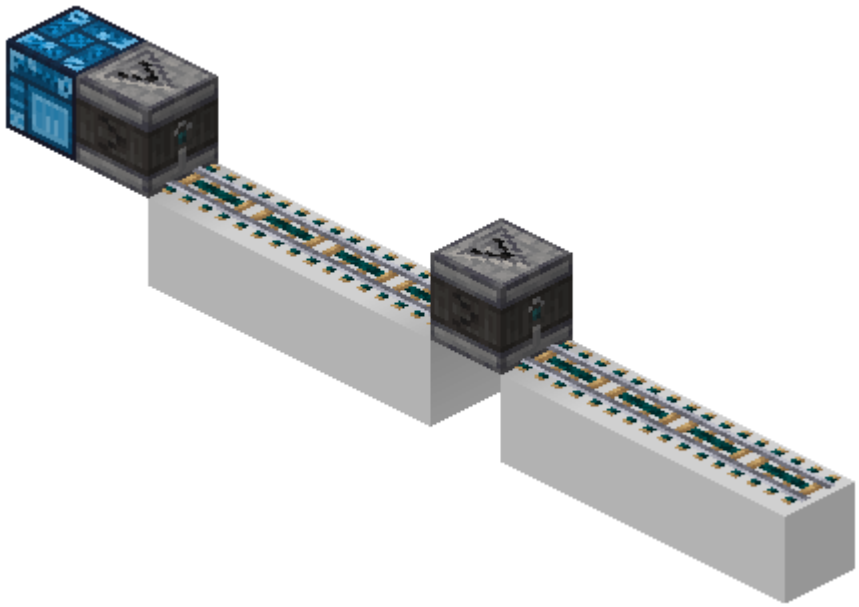
> *Dustless has never been the goal—it's a means to an end. The ultimate objective should be to reduce lag, not to force dustless designs. Forcing dustless designs can easily lead to increased lag, which goes against the very point of dustless wiring.*

Once we understand this, we can truly make good use of dustless wiring.

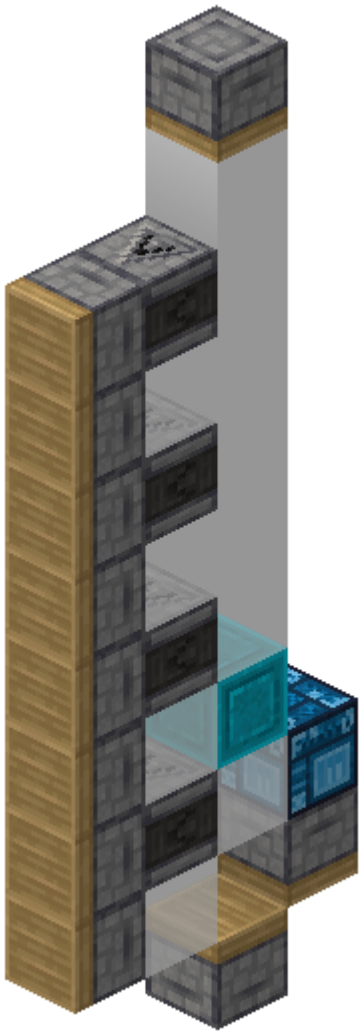
Now let's walk through the main methods of dustless wiring.

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.

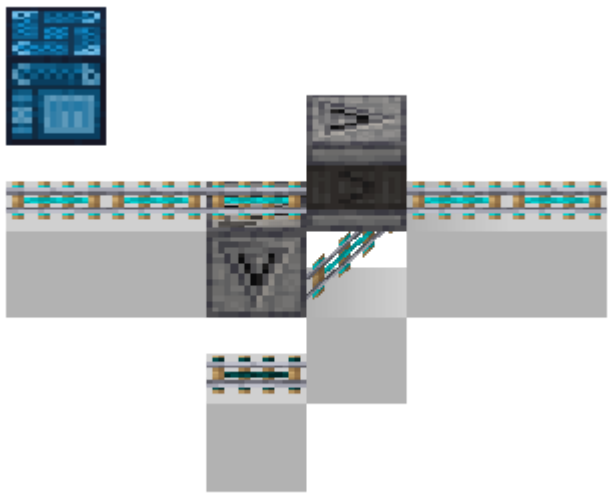


In tree farms, Pistons are generally arranged vertically. We only need to activate a vertical column of Observers simultaneously to activate an entire row of Pistons at once. There are two basic approaches: pushing **the Observer itself** or **the block the Observer is facing** with a Piston. ~~Resetting 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.

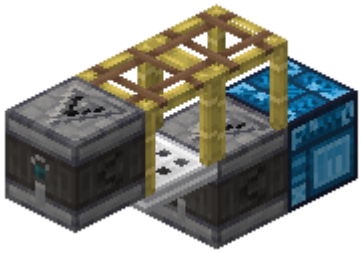
> *In fact, tree power's update mechanism works somewhat like Redstone Dust: it checks the distance of surrounding logs and leaves. If there's an adjacent log, it sets its own distance to 0; if there are no logs around, it sets its own distance to the minimum distance of adjacent leaves plus 1.*

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.

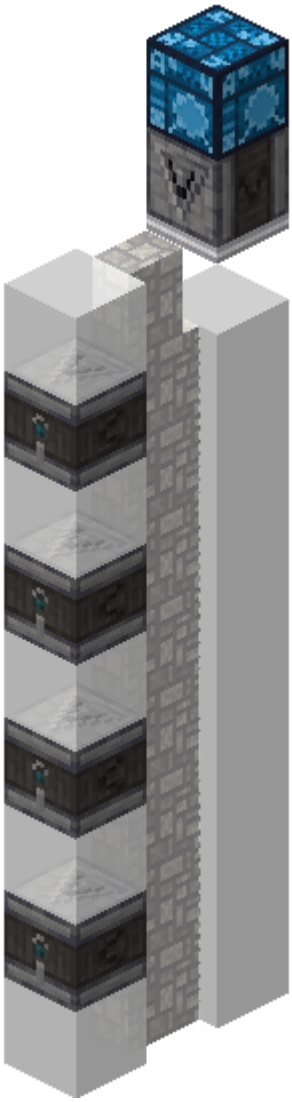


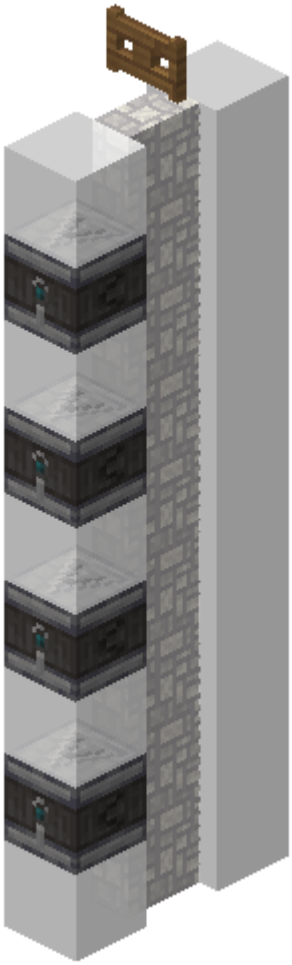
Generally, we use scaffolding power to create **odd-gt delays** ~~it looks similar to tree power, and the use cases are almost the same~~

> *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 (*

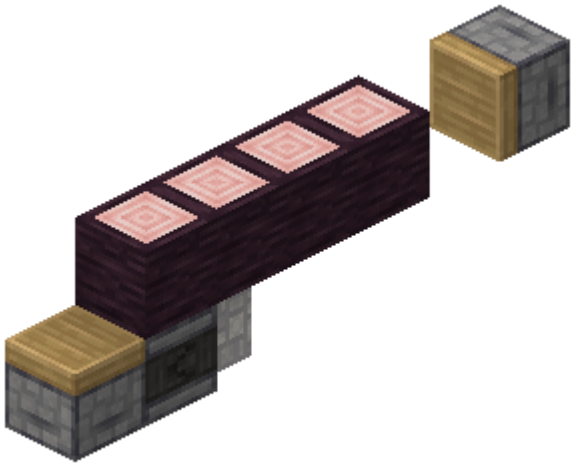
2.3 WALL POWER

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.



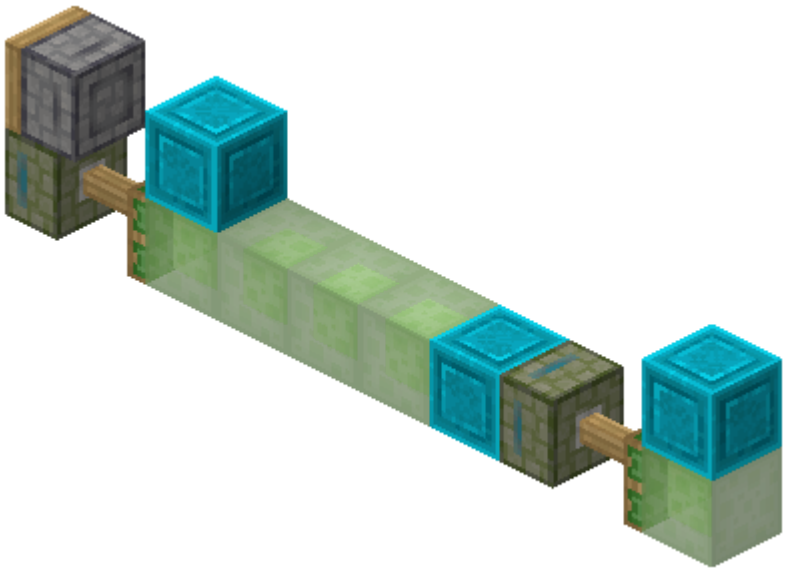
Walls are **instant components**. All wall state changes emit **PP updates** and can be detected by Observers.

> *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.

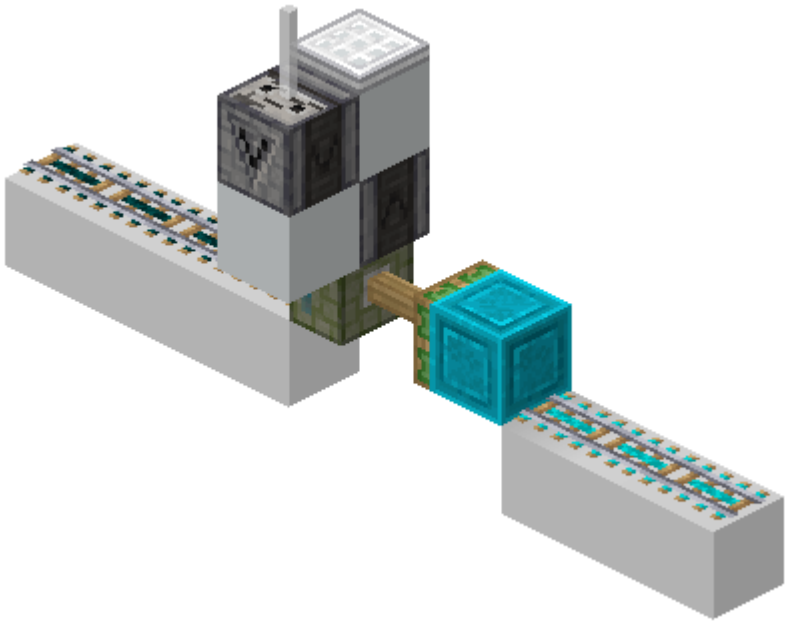
3.1 SLIME STICKS



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.

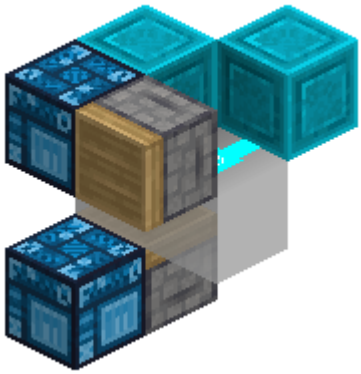
> *In fact, BUDs only need updates, regardless of whether it's a rising or falling edge signal. The distinction here is just for consistency with the earlier discussion.*

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*

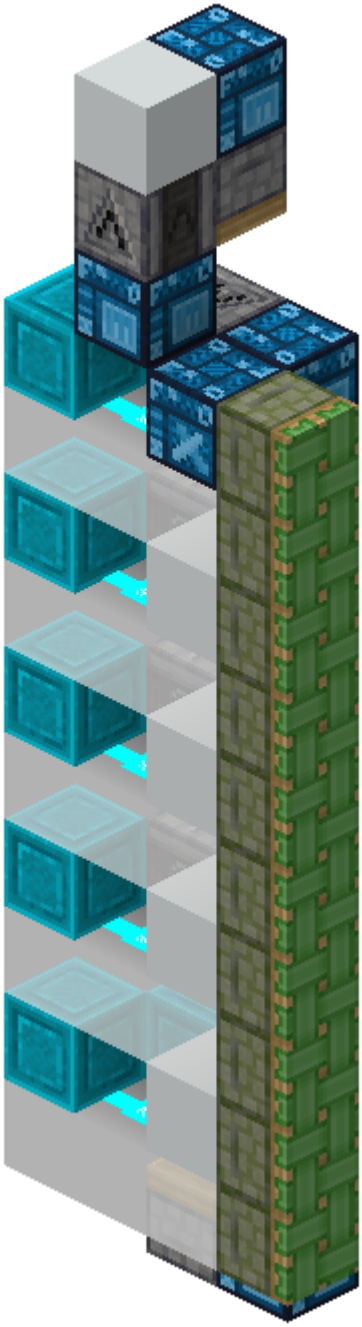
Of these, **the movable ones** are the most important for practical redstone redirection.

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**.

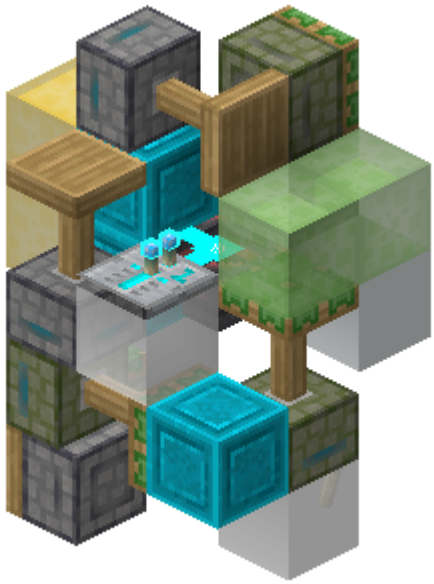
`5gt TE`: reset. The target Piston self-checks again, finds it's not powered, and doesn't extend.

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.

CHAPTER 6

红石元件与特性

3 ARTICLES

> *This document was written by HackerRouter. Mechanism discovered by BFladderbean.*

Reference video:

1 TL;DR

`TACS` (`ThreadedAnvilChunkStorage`) saves chunk A (old state) during incremental save in a normal tick and sets a 10-second cooldown →
States of chunks A and B are changed →
In subsequent save attempts, chunk A is skipped due to cooldown, chunk B is saved normally →
Main thread freezes, Watchdog force-exits →
After restart, chunk A is the old snapshot, chunk B is the new snapshot, items exist in both locations simultaneously.

2 TACS

`TACS` (`ThreadedAnvilChunkStorage`, corresponding to `ChunkMap` in Mojang mappings) is the core component for server-side chunk persistence.
It decides which chunks need saving, when to save them, and how to throttle saves.

`TACS` holds a `ChunkHolder` reference for every loaded chunk (one per loaded chunk).
These references live in `currentChunkHolders` / `chunkHolders`, `Long2ObjectLinkedOpenHashMap<ChunkHolder>` maps keyed by chunk coordinates encoded as a long.
`TACS` drives the serialization and write pipeline for chunks on every tick and during auto-saves.

All save logic discussed below (incremental saves, auto-saves, cooldown throttling) happens inside `TACS`.

3 HOW CHUNK SAVES ACTUALLY TRIGGER

Chunk saves don't just happen during the auto-save that runs every 6000 ticks.
The server also performs incremental saves every tick.

The call chain is as follows:

```
MinecraftServer.tick()
  → tickWorlds()
    → serverWorld.tick()
      → getChunkManager().tick()      // ServerChunkManager.tick()
        → threadedAnvilChunkStorage.tick()
          → unloadChunks()
            → Iterate chunkHolders, save up to 20 chunks per tick
```

At the end of `ThreadedAnvilChunkStorage.unloadChunks()` (1.20.1 Yarn `ThreadedAnvilChunkStorage.java`:522-529):

```
int k = 0;
ObjectIterator<ChunkHolder> objectIterator = this.chunkHolders.values().iterator();
while (k < 20 && shouldKeepTicking.getAsBoolean() && objectIterator.hasNext()) {
    if (this.save(objectIterator.next())) {
        k++;
    }
}
```

Each tick, the loop tries to save up to 20 chunks. This runs independently of auto-save, continuously in the background.

4 10-SECOND SAVE COOLDOWN

The 10-second save cooldown is how `TACS` throttles saves for individual chunks.
It prevents frequently changing chunks (such as those with ongoing redstone activity) from being serialized and written to disk every tick, reducing I/O overhead.

The specific implementation is in `ThreadedAnvilChunkStorage.save(ChunkHolder)` (1.20.1 Yarn `ThreadedAnvilChunkStorage.java`:790-813):

```
long m = this.chunkToNextSaveTimeMs.getOrDefault(l, -1L);
long n = System.currentTimeMillis();
if (n < m) {
    return false;   // Within cooldown period, skip save
}
boolean bl = this.save(chunk);
if (bl) {
     this.chunkToNextSaveTimeMs.put(l, n + 10000L);   // Set 10-second cooldown after
successful save
}
```

`chunkToNextSaveTimeMs` is a `Long2LongMap` (implemented as `Long2LongOpenHashMap`) keyed by chunk position (long-encoded), with values representing the timestamp (ms) when the next save is allowed.
After a successful save, the chunk's cooldown deadline is set to current time + 10000ms.

During the cooldown period, even if the chunk becomes dirty again from block changes (`needsSaving() == true`), `ThreadedAnvilChunkStorage.save(ChunkHolder)` returns `false` and skips the save.
The dirty flag is necessary but not sufficient for saving. If the cooldown hasn't expired, the chunk is skipped regardless.

5 SERIALIZATION COMPLETES SYNCHRONOUSLY ON MAIN THREAD

Key sequence in `ThreadedAnvilChunkStorage.save(Chunk)` (1.20.1 Yarn `ThreadedAnvilChunkStorage.java`:816-840):

```
if (!chunk.needsSaving()) { return false; }
chunk.setNeedsSaving(false);                                    // Clear
dirty flag
NbtCompound nbtCompound = ChunkSerializer.serialize(this.world, chunk);     // Main
thread synchronous serialization
this.setNbt(chunkPos, nbtCompound);                                  // Submit to
StorageIoWorker (background worker thread responsible for async disk writes)
```

`ChunkSerializer.serialize()` runs inline on the main thread, iterating through `chunk.getBlockEntityPositions()` and calling `chunk.getPackedBlockEntityNbt()` for each position.
The serializer captures the live chunk's logical state at the moment of the call.
The I/O thread never reads from the active chunk's memory.

6 ASYNC WRITE PIPELINE — THREE-LAYER CHAIN

The serialized NBT goes through three layers before reaching the disk:

Layer 1: Main thread → `StorageIoWorker.results` (JVM heap memory)

`VersionedChunkStorage.setNbt()` → `StorageIoWorker.setResult()`, stores NBT into `results`, a `LinkedHashMap`.

Layer 2: `StorageIoWorker` background thread → `RegionFile` (OS file cache)

`StorageIoWorker.writeResult()` takes one entry at a time and writes it through `RegionBasedStorage.write()` to `RegionFile`.
`RegionFile.writeChunk()` calls `FileChannel.write()` to write data to the file, but does not call `force()` / `fsync()` (OS-level forced flush system call).
Data enters the OS page cache (memory where the OS buffers file writes before flushing to physical disk), so it's not yet guaranteed to reach the physical disk.

Layer 3: OS page cache → Physical disk

Without `fsync`, flushing to disk depends on the OS's dirty page writeback strategy.
When the process terminates, unflushed data may be lost.

`StorageIoWorker` writes chunks entry by entry, not as an atomic commit of the entire world at once.

7 AUTO-SAVE DOES NOT BYPASS COOLDOWN

Auto-save triggered every 6000 ticks (1.20.1 Yarn `MinecraftServer.tick()`):

```
if (this.ticks % 6000 == 0) {
    this.saveAll(true, false, false);  // flush=false
}
```

The `flush` parameter controls how thoroughly the save runs:

- `flush=true`: Forces serialization of all dirty chunks (bypassing the 10-second cooldown) and waits for `StorageIoWorker` to finish writing and syncing all queued data to disk before returning. Used for `/save-all flush` or server shutdown.
- `flush=false`: Uses the normal `ThreadedAnvilChunkStorage.save(ChunkHolder)` path, subject to cooldown, and returns immediately after submitting to `StorageIoWorker` without waiting for the actual disk write.

Auto-save uses `flush=false`.

`saveAll` → `save` → `serverWorld.save()` → `serverChunkManager.save(false)` → `ThreadedAnvilChunkStorage.save(false)`.

The non-flush branch of `ThreadedAnvilChunkStorage.save(false)` (1.20.1 Yarn `ThreadedAnvilChunkStorage.java`:469-471):

```
this.chunkHolders.values().forEach(this::save);      // Reuses save(ChunkHolder), subject to cooldown
```

Auto-save calls the same `ThreadedAnvilChunkStorage.save(ChunkHolder)`.
If a chunk was just saved in a previous incremental save, auto-save will also skip it during the cooldown period.

8 WHEN CHUNKS BECOME DIRTY

When `WorldChunk.setBlockState()` executes on the main thread (1.20.1 Yarn):

1. Modify block state inside section

2. Call `onStateReplaced` on old state
3. `removeBlockEntity` if necessary
4. Finally directly set `this.needsSaving = true`

A chunk becomes dirty again as soon as a block change occurs.
However, the dirty state has no connection to the old NBT already queued in `StorageIoWorker` — that old NBT is already an independent data object.

9 WATCHDOG FORCE EXIT

`DedicatedServerWatchdog` monitors the main thread in a separate thread.
After detecting timeout:

1. Write crash report
2. Call `System.exit(1)`
3. Schedule `Runtime.halt(1)` as fallback after 10 seconds

`System.exit(1)` triggers the JVM shutdown process.
Before the process fully exits, the `StorageIoWorker` background thread may keep running briefly, continuing to write out chunks already queued in `results`.
This continues until the JVM exits normally or gets hard-killed by `halt(1)`.

10 PROCESS WALKTHROUGH

Consider the example from Comedy's video: a dispenser shoots a shulker box into an adjacent chunk.

T=0: Incremental save writes chunk A

During a normal tick, the incremental save loop reaches chunk A (containing the dispenser + shulker box). Conditions are met:

- Cooldown has expired
- `needsSaving() == true`

Main thread synchronously runs `ChunkSerializer.serialize()`, generating `A_old_nbt` (dispenser contains shulker box).
This is submitted to `StorageIoWorker`.
Then `chunkToNextSaveTimeMs[A] = now + 10000ms` is set.

Chunk A then enters a 10-second cooldown period.

T=1: Chunks A and B are changed

Dispenser is activated by redstone; 4 ticks later `ServerWorld`'s `blockTickScheduler.tick()` executes the scheduled tick and calls `DispenserBlock.scheduledTick()`, firing the dispenser.
Shulker box moves from chunk A (dispenser) to chunk B (landing position).
Chunk A becomes dirty (dispenser becomes empty), chunk B becomes dirty (shulker box block appears).

T=2: Auto-save/incremental save runs again

Auto-save triggers, or the incremental save loop reaches these two chunks:

- Chunk A: `ThreadedAnvilChunkStorage.save(ChunkHolder)` → cooldown not expired (still within 10 seconds) → **skipped**
- Chunk B: No cooldown or cooldown expired → save successful → generates `B_new_nbt` (contains shulker box)

At this point, the persistent state has diverged:

- Chunk A: On disk is `A_old_nbt` (dispenser contains shulker box)
- Chunk B: On disk is `B_new_nbt` (landing position has shulker box)

T=3: Main thread frozen by redstone + Watchdog force-exit

After main thread freezes, no new serialization occurs.
`StorageIoWorker` background thread continues writing remaining NBT in `results`.
Watchdog detects timeout, `System.exit(1)`.

Note: The state divergence had already formed at T=2 — chunk A was skipped due to cooldown while chunk B was saved as a new snapshot.
The Watchdog force-exit at T=3 only prevents any subsequent save that might have corrected this divergence (such as the next incremental save after cooldown expiration).
Additionally, since the normal async write path's `RegionFile.writeChunk()` doesn't call `fsync` (see Section 5), data in the OS page cache that hasn't been flushed to physical disk when the JVM exits may also be lost, but this is only a contributing factor, not the core cause.

After Restart

Chunk A loads from disk: dispenser contains shulker box (old snapshot).
Chunk B loads from disk: landing position has shulker box (new snapshot).
Shulker box is duplicated.

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

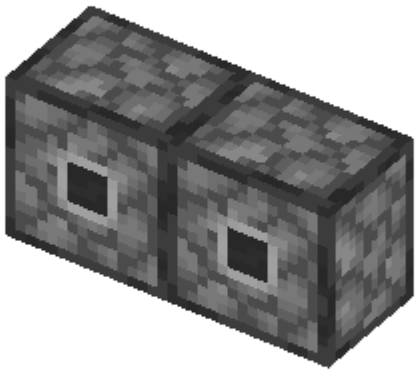
1.1 PISTON COMPOSITION

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

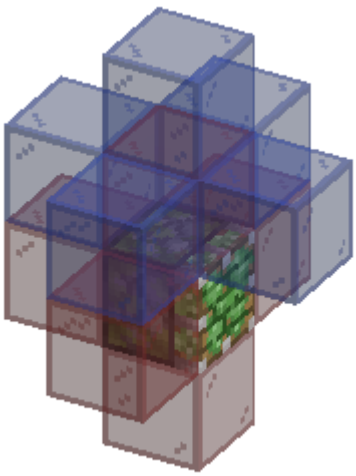
Pistons trigger self-checks if and only if the following two situations occur:

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.

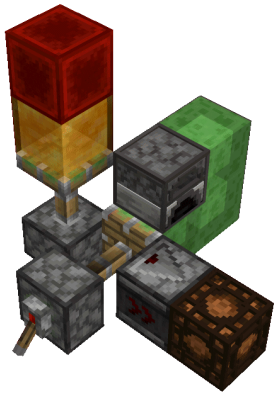
> *Tip: Due to the extreme complexity of structure analysis (alternating linear and branch analysis on the stack), manual analysis is impractical. It is strongly recommended to use Fallen_Breath's PistOrder mod to directly view the piston's attempted block placement order in-game.*

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

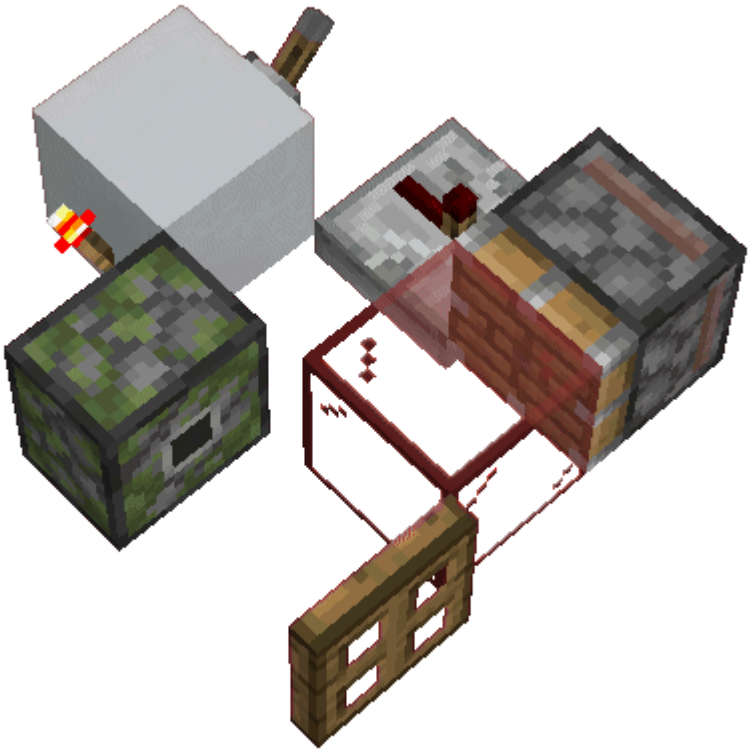
When a piston triggers instant retraction, blocks skip the normal TE phase accumulation and are forced to place immediately. Trigger conditions are:

- 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:



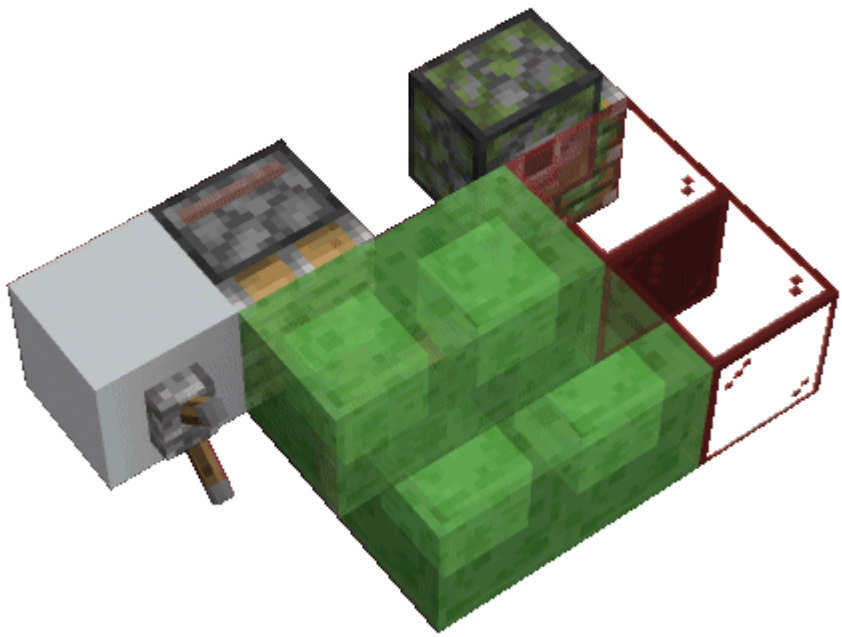
▶ This figure is animated. View original

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

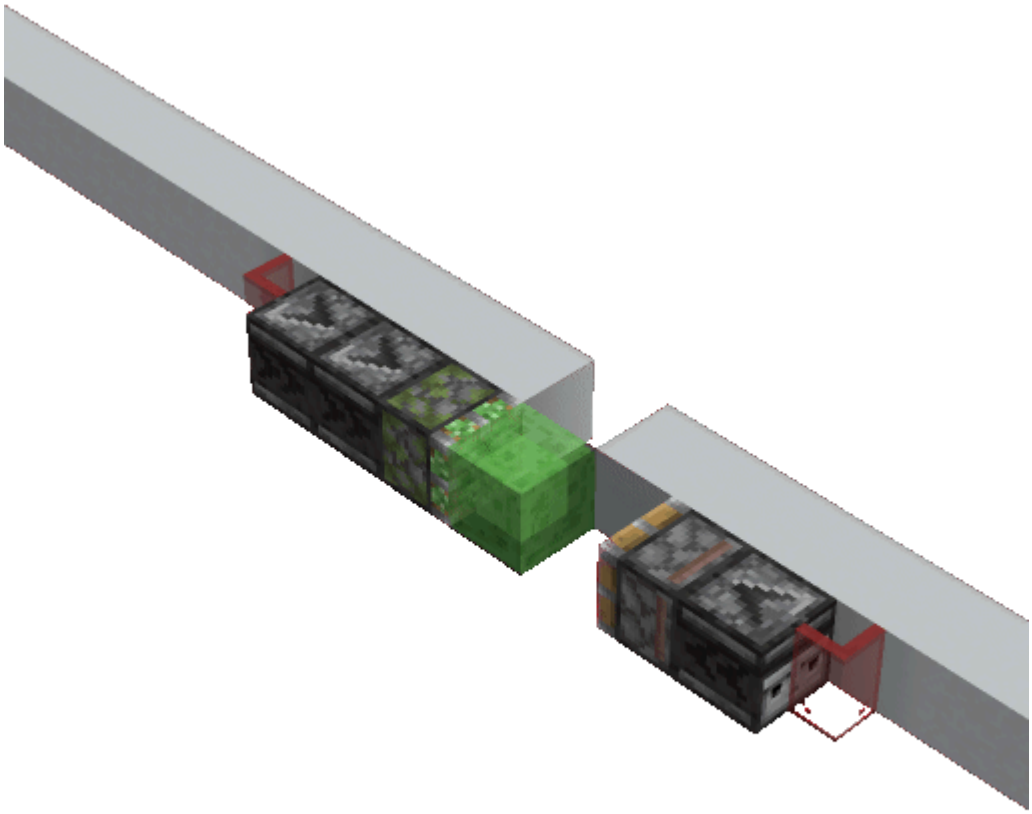


▶ This figure is animated. View original

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:

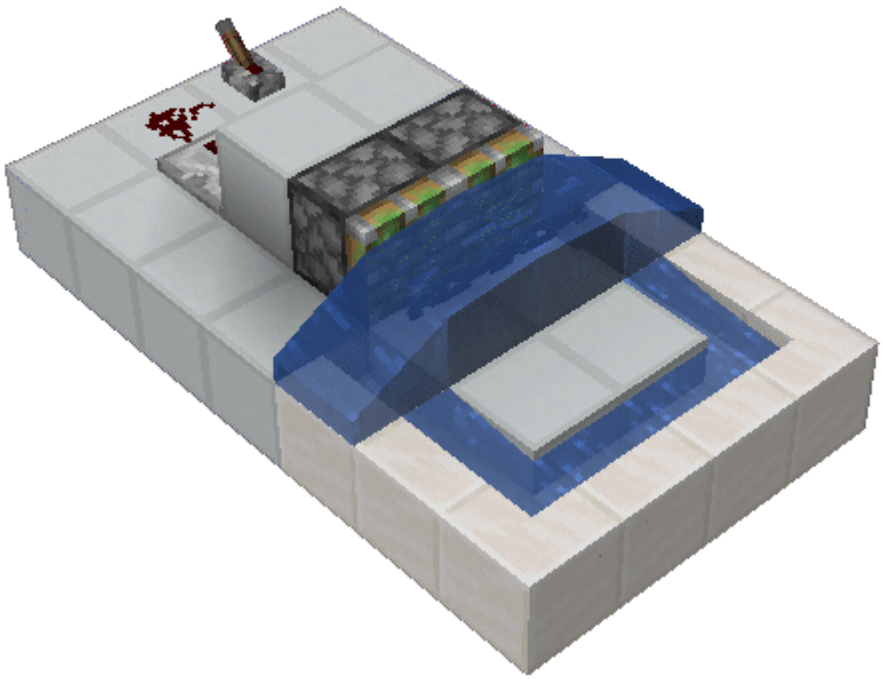


▶ This figure is animated. View original

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`

```
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, pis-
tonHeadState, (BlockState)this.getDefaultState().with(FACING, Direction.byId(direc-
tionData & 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.getOff-
setY() * 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;
                    }
                }
            }
            // If no special case, remove target block or continue moving
            if (!hasMovingPistonInFront) {
                   if (actionType != 1 || targetState.isAir() || !isMovable(target-
State, 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);
                }
            }
        }
### [!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.

```
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(di-
rection), 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

```
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 posi-
tion 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 subse-
quent 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, pistonType);
                            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`

```
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);
        }
    }
}
```

```
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.mo-
tionDirection, 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 dupli-
cate 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

```
while (isBlockSticky(stateToValidate)) {
// Get the next block in the opposite direction of the sticky block
     BlockPos blockBehind = posToValidate.rearBlockCount(this.motionDirection.getOp-
posite(), 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.mo-
tionDirection, 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.

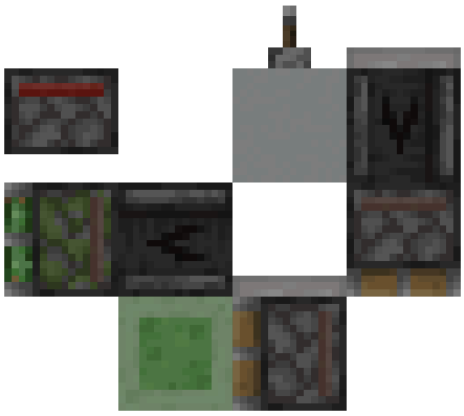
Most of the content has been explained before, so we'll skip ahead here. We only look at the b36 special case in the latter half.

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.

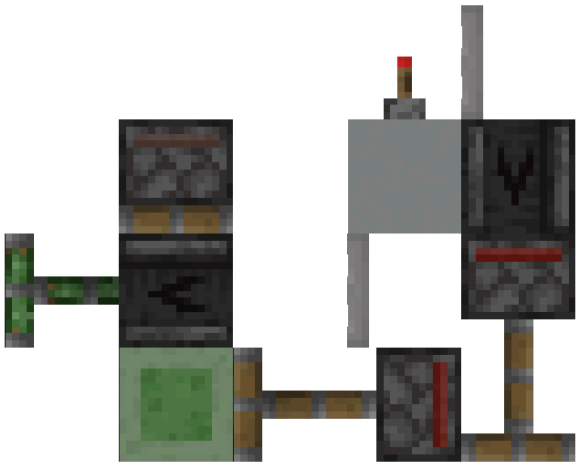
1.8 PISTON HEAD

> *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`

```
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).getOpposite()));
                        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`

```
@Override
public void onRemove(BlockState state, Level level, BlockPos pos, BlockState new-
State, 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

```
@Override
public BlockState updateShape(BlockState state, Direction facing, BlockState fac-
ingState, 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

```
// 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.motionDirection, 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`

```
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(adjacentBlock-
Pos);
              // If the 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 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), di-
rection, 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 dis-
tance
                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 en-
tity 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 condi-
tions
          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 distance
            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 retrac-
tion 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, pis-
tonHeadState,                          (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`:

```
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`:

```
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.

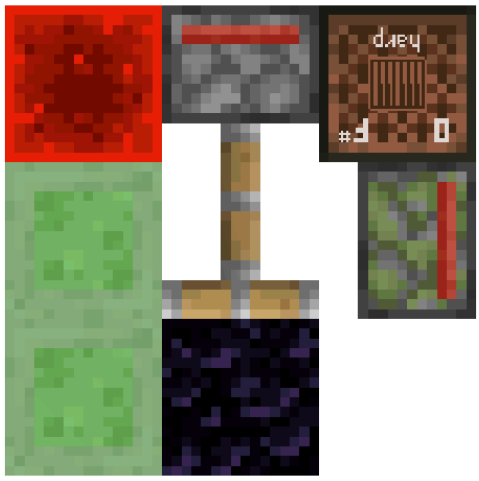
In `ClientWorld`, this `updateListeners` is responsible for rendering. Therefore, the client jumps ahead here, directly rendering the extended piston arm without rendering the piston head.

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:

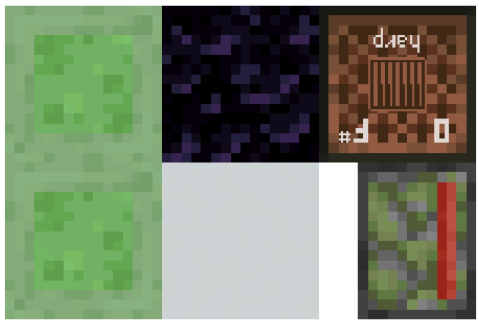
```
if (!hasMovingPistonInFront) {
      if (actionType != 1 || targetState.isAir() || !isMovable(targetState, world,
targetpos, facing.getOpposite(), false, direction)
                    || targetState.getPistonBehavior() != PistonBehavior.NORMAL &&
!targetState.isOf(Blocks.PISTON) && !targetState.isOf(Blocks.STICKY_PISTON))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

After some judgment, enters the `move` method.

```
**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`, place-
ment 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 pis-
ton 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

```
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?

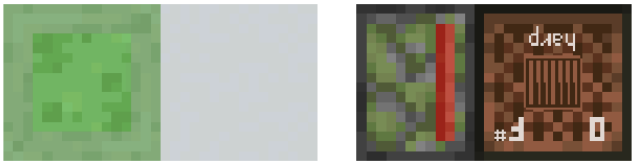
Regular block is removed

Back to `onSyncedBlockEvent`, still this judgment:

```
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:

```
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;
        }
    }
}
```

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 han-

dling 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:

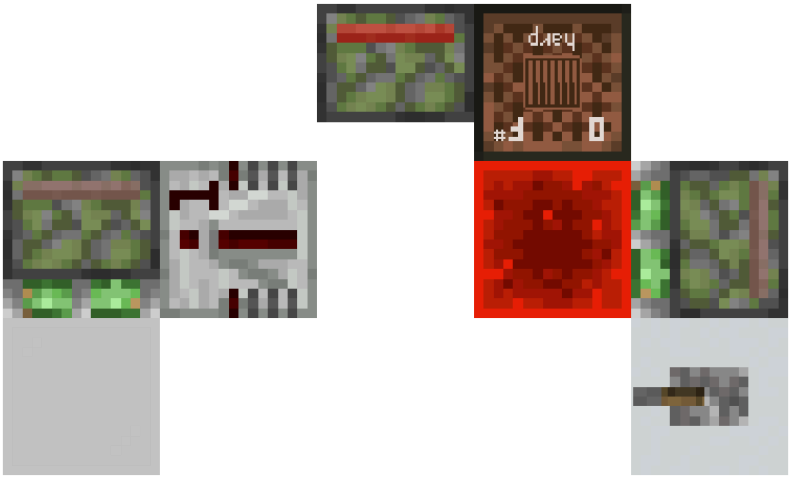
1.10.7 PISTON MOVING BLOCKS `MOVE`

```
// 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(tar-
getPos, 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,   pistonHead-
State,  (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

```
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:

```
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

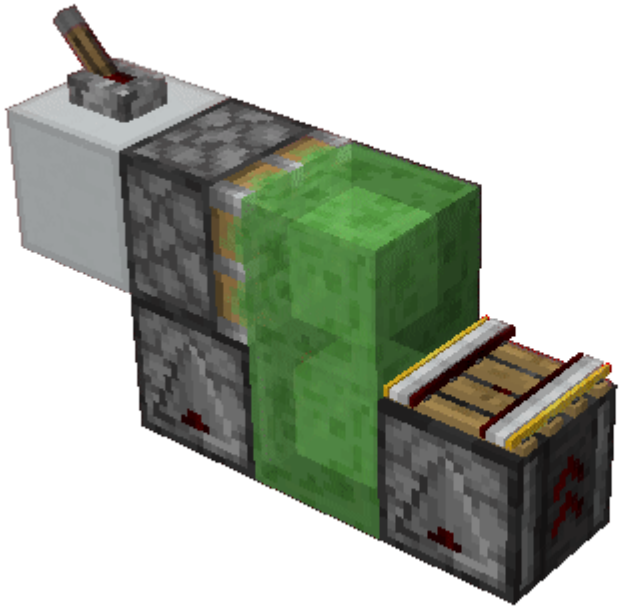
When a lit observer is removed, it emits NC update backward and extinguishes itself.

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:



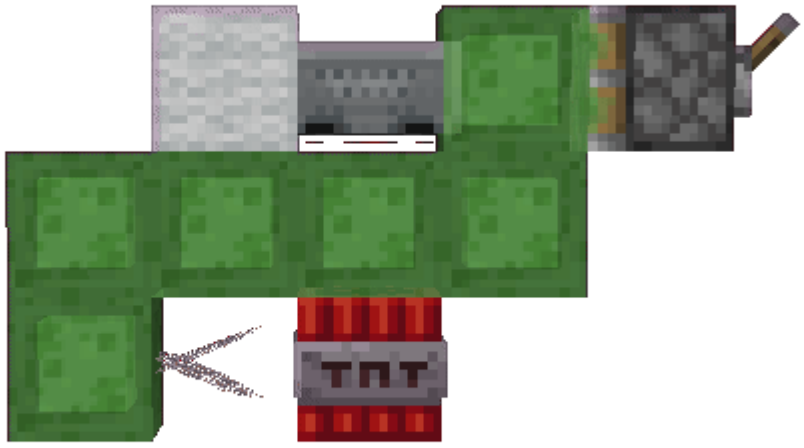
▶ This figure is animated. View original

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

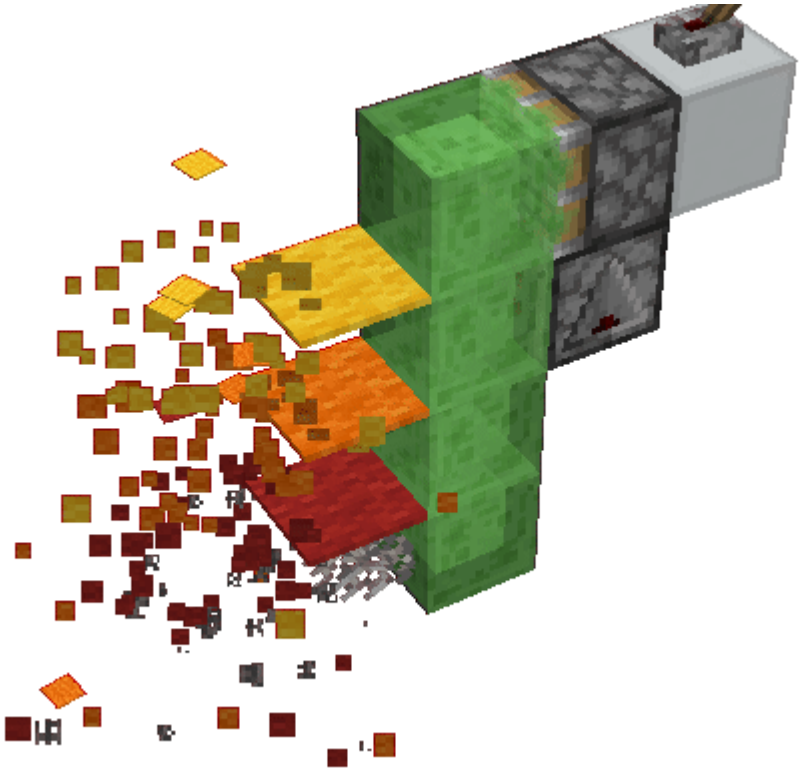
Attachment block dropping also produces updates, but since attachment blocks usually drop directly and are difficult to reuse. For example:

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, pistonType);
                           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.

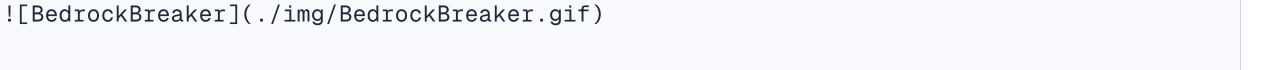
> Interestingly, the animation when b36 retracts still follows the data parameter passed in the block event,
> so, even though the piston facing at block event execution differs from when it was added, the piston head animation will still face the piston facing when the block event was added (but the 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.
![Modifiable time period for block event objects](img/ModifiableTimeInBlockEvent.png)

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

![BedrockBreakerInBlockEvent.png](img/BedrockBreaker-2.png)
![BedrockBreakerInBlockEvent.gif](img/BedrockBreaker-2.gif)

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.

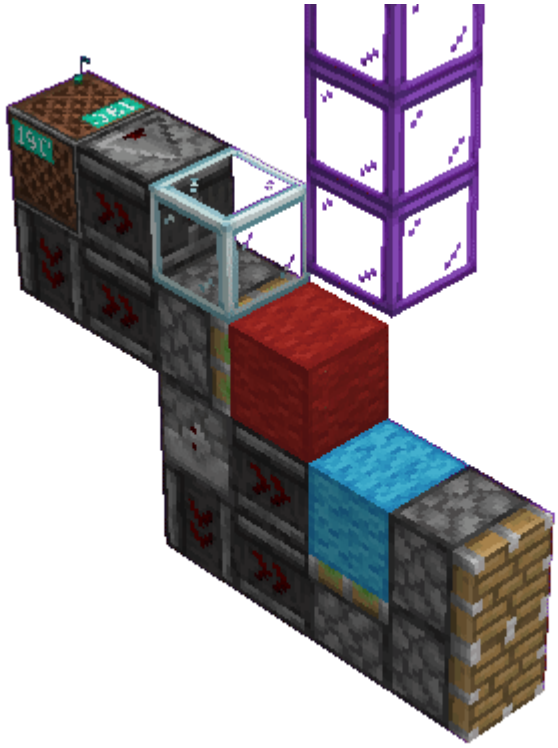
    > *Since depths 1, 2, 3 are occupied, it will be scheduled to #4, which is "execute last" in the current situation.*

  - 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`

```
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);
        }
    }
}
```

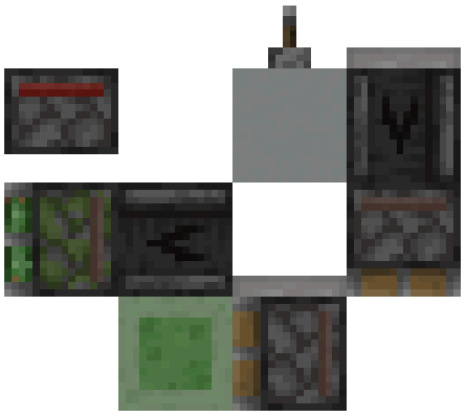
Most of the content was explained before, so it's skipped here. We only look at the latter part of the b36 special case.

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.

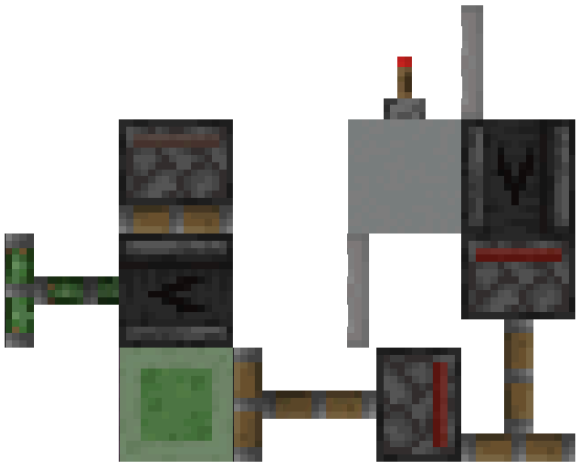
2.3 PISTON HEAD

> *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`

```
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).getOpposite()));
                        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`

```
@Override
public void onRemove(BlockState state, Level level, BlockPos pos, BlockState new-
State, 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.

2.3.3 PISTON HEAD RECEIVES PP UPDATE, NC UPDATE

```
@Override
public BlockState updateShape(BlockState state, Direction facing, BlockState fac-
ingState, 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` ↩

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:

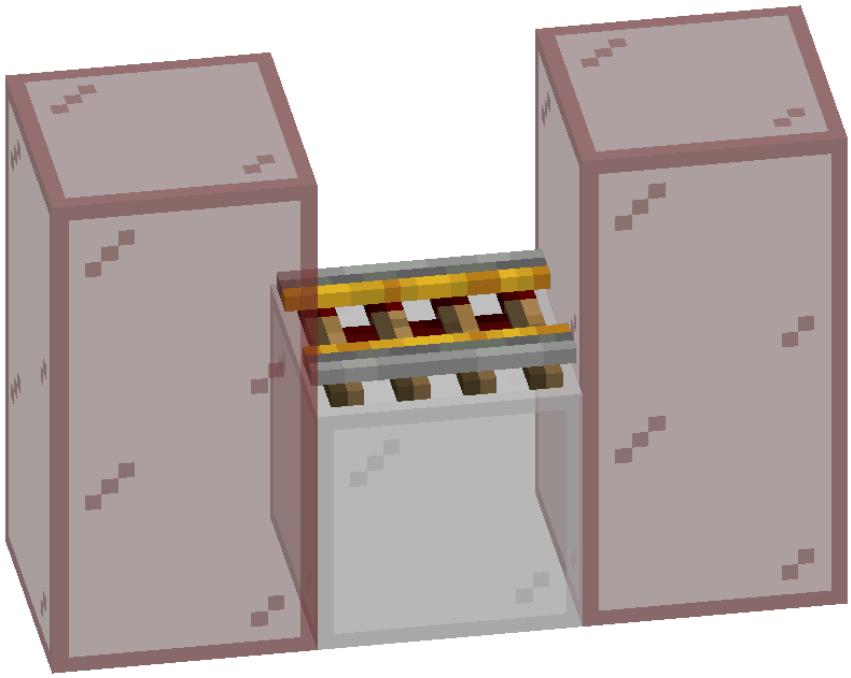
- Its own shape (whether it is sloped)
- The position of the next rail
- The direction of the next rail

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

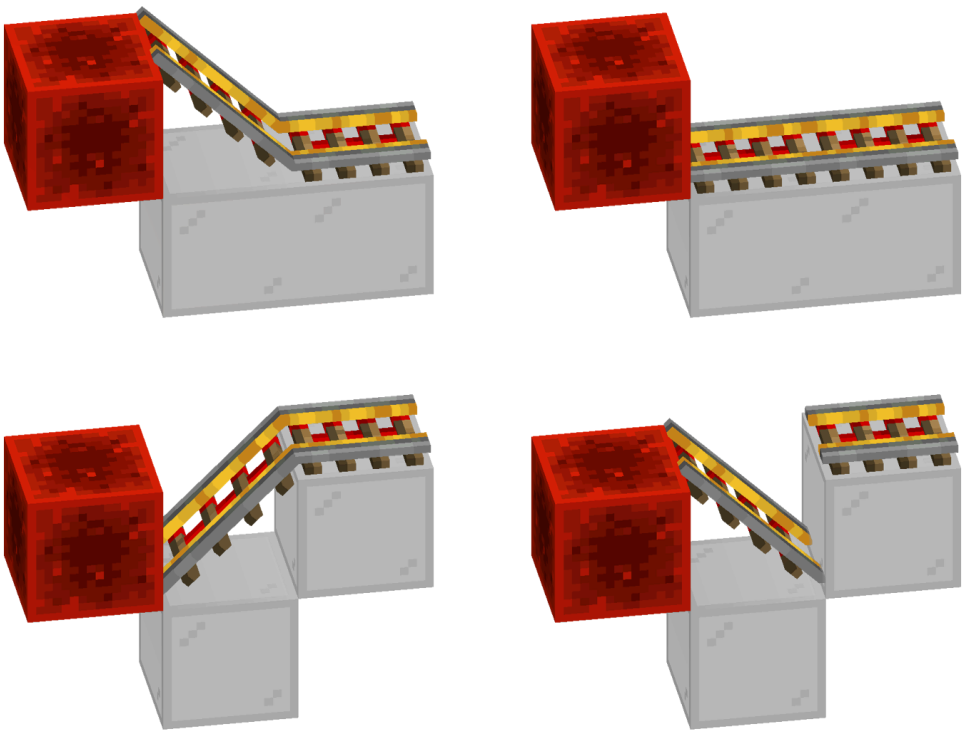
Flat rails will check these positions:

- 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:



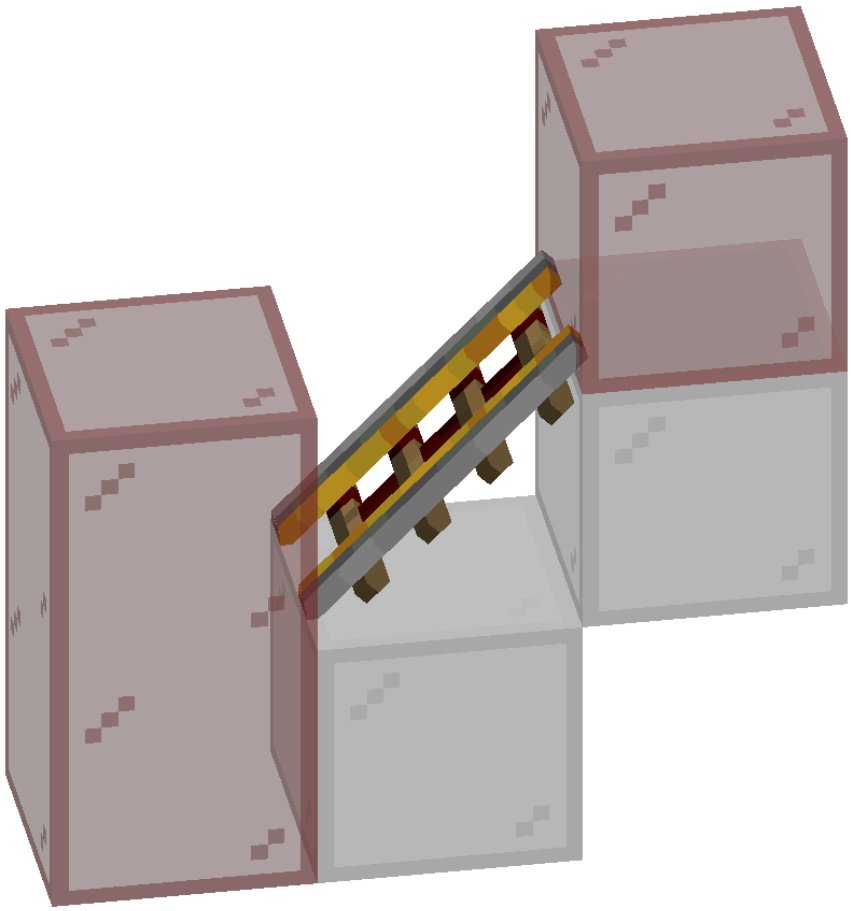
For example, when the flat rail on the far right is "searching" to the left for directly activated rails, it considers the rail on the left connected to itself. The first three cases in the image are straightforward, while the last one occurs because **rails do not consider the shape of the next rail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



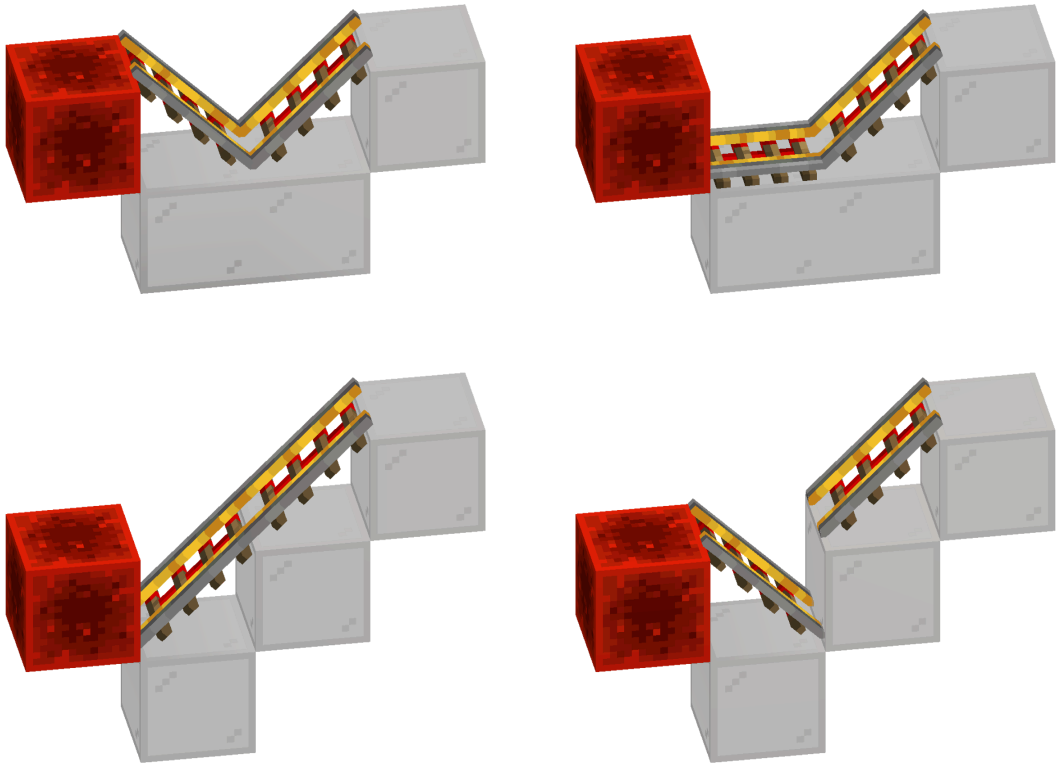
Sloped rails will check these positions:

- 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

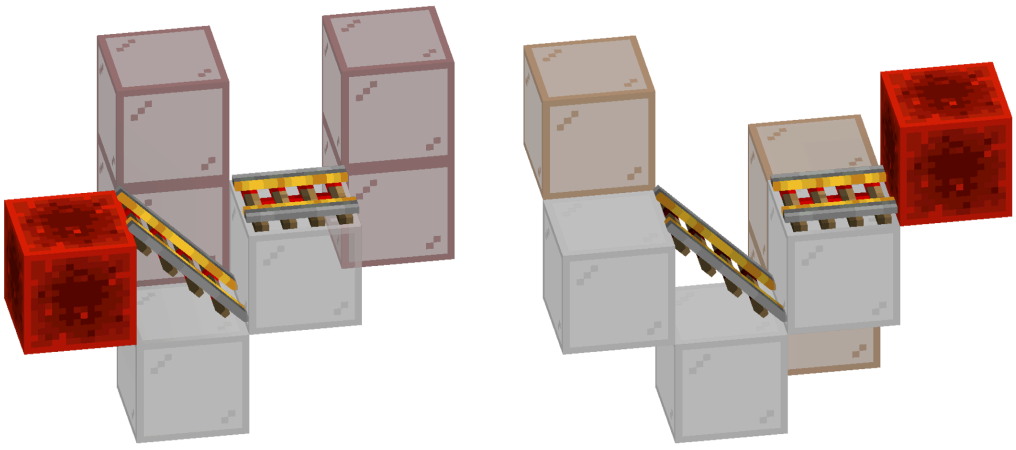
The range is shown by the red glass positions in the image:



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**.

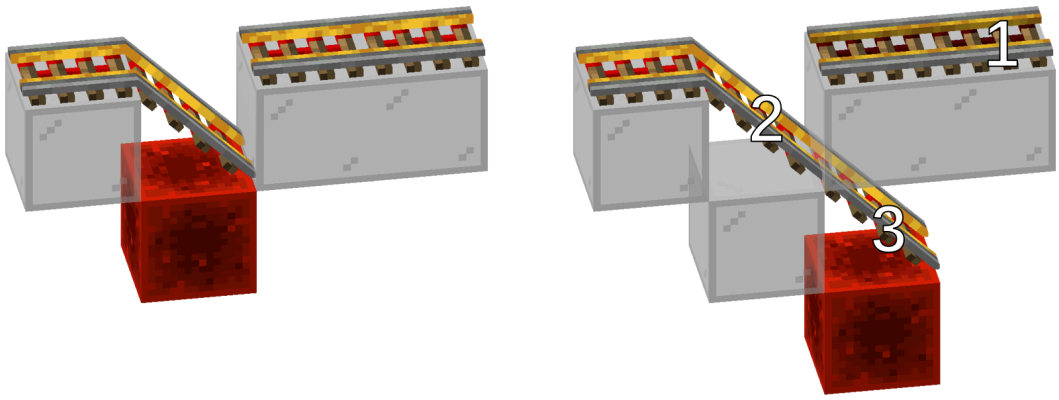
This can be used to create rail chains that can only transmit rail signals in one direction.

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. ↩

CHAPTER 7

MOLFORTE

2 ARTICLES

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)

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.

APPENDIX 8

附录

2 ARTICLES

1 OVERVIEW

This chapter explains the specialized terms and vocabulary used throughout GTMC.

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.