Contentpack Dokumentation v4

Jeronimo, Uhutown, MrTroble

30. Juli 2024

Inhaltsverzeichnis

1 Definitionen

- **muss** sei im folgenden definiert als etwas das getan werden muss um die richtige Funktionalität zu garantieren.
- **sollte** sei im folgenden definiert als etwas das wahrscheinlich von den meisten Nutzern getan werden muss um basis Funktionen zu Nutzen.
- **kann** sei im folgenden definiert als etwas das optional getan werden kann aber es nicht notwendig ist.

- **STRING** bezeichnet eine Zeichenkette, die mit ” am Anfang und Ende gekenzeichnet werden. Also: ”Text”
- **INT** bezeichnet eine Ganzzahl ohne Begrenzung der Speicherkapazität. Also: 120
- **FLOAT** bezeichnet eine Kommazahl ohne Begrenzung der Speicherkapazität. Also: 1.2
- **BOOLEAN** bezeichnet einen Wahrheitswert (1 bit): Also entweder true oder false
- **RELATION** bezeichnet eine spezielle STRING die eine Aussagenlogische Formel bilden. Diese werden gesondert in 1.1 behandelt
- **PROPERTIES** bezeichnet eine spezielle STRING die eine gewisse Property-Namen enthält. Diese werden gesondert in 5.1 behandelt

Kommt ein **...** nach einem dieser Typen in der Dokumentation so handelt es sich dabei um eine Liste, daher es können mehrere Unterschiedliche Einträge, auch mit Wertepaaren, gemacht werden. Beispiel: Für

```
"VARIABLE...": "VAR"
```

Kann also beliebige paare halten:

```
"variable1": "value1",
"variable2": "value2",
"variable3": "value3"
...
```

1.1 Relationen

Im folgenden Stellen wir wichtige Methoden vor und erklären kurz unsere verwendeten Operatoren zur Aussagenlogik. Generell bietet es sich an, sich dazu im Internet weiter zu informieren.

1.2 Funktionen

- **config()-Methode** Mit der config-Methode können Signaleigenschaften (PROPERTY) als Bedingung für etwas gesetzt werden, z.B. für andere Signaleigenschaften oder Signaleinstellungen. In dem Sinne, dass etwas gegeben sein muss, damit etwas anderes funktioniert. Wird in den Signalsystemen (5.1) verwendet. Es können auch mehrere Signaleigenschaften logisch miteinander verknüpft werden. Siehe Aussagenlogik 1.3.

  ```
  "config(PROPERTY.ENUM)"
  ```

- **with()-Methode** Mit der with-Methode können Signaleigenschaften (PROPERTY) und die zugehörigen Einstellmöglichkeiten (ENUM) als Bedingung für Teilmodelle gesetzt werden. In dem Sinne, dass etwas gegeben sein muss, damit etwas anderes funktioniert. Wird in der Modelldefinition (5.5) verwendet. Es können auch mehrere Signaleigenschaften logisch miteinander verknüpft werden. Siehe Aussagenlogik 1.3.

  `"with(PROPERTY.ENUM)"`

- **hasandis()-Methode** [1] Mit der hasandis()-Methode können Wahrheitswert-Eigenschaften (BOOLEAN-PROPERTY) als Bedingung für Teilmodelle gesetzt werden. Die Methode fragt ab, ob das jweilige BOOLEAN-PROPERTY gegeben ist oder nicht und bezieht sich im Gegensatz zur has()-Methode nur auf das eigene Modellteil. In dem Sinne, dass etwas gegeben sein muss, damit etwas anderes funktioniert. Wird in der Modelldefinition (5.5) verwendet. Es können auch mehrere Signaleigenschaften logisch miteinander verknüpft werden. Siehe Aussagenlogik 1.3.

  `"hasandis(BOOLEAN-PROPERTY)"`

- **has()-Methode** Mit der has()-Methode können Wahrheitswert-Eigenschaften (BOOLEAN-PROPERTY) als Bedingung für Teilmodelle gesetzt werden. Die Methode fragt ab, ob das jweilige BOOLEAN-PROPERTY gegeben ist oder nicht und bezieht sich im Gegensatz zur hasandis()-Methode nur auf ein anderes Modellteil. In dem Sinne, dass etwas gegeben sein muss, damit etwas anderes funktioniert. Wird in der Modelldefinition (5.5) verwendet. Es können auch mehrere Signaleigenschaften logisch miteinander verknüpft werden. Siehe Aussagenlogik 1.3.

  `"has(BOOLEAN-PROPERTY)"`

1.3 Aussagenlogik

- **&&** (UND Verknüpfung)
- || (ODER Verknüpfung)
- **( )** (Klammern)
- **!** (Negation)

A	B	&&
w	w	w
w	f	f
f	w	f
f	f	f

A	B	||
w	w	w
w	f	w
f	w	w
f	f	f

A	!A
w	f
f	w

A	B	C	(A || B) && C
w	w	w	w
w	w	f	f
w	f	w	w
w	f	f	f
f	w	w	w
f	w	f	f
f	f	w	f
f	f	f	f

A	B	C	(A && B) || C
w	w	w	w
w	w	f	w
w	f	w	w
w	f	f	f
f	w	w	w
f	w	f	f
f	f	w	w
f	f	f	f

Tabelle 1: Logische Verknüpfung von Wahrheitswerten.

[1] Eine Boolean-Property ist eine Signaleigenschaften deren **enumClass** BOOLEAN ist. Mehr dazu in 5.1.

2 Einleitung

In der folgenden Dokumentation werden wir euch erklären, wie man ein ContentPack, kurz CP, aufgebaut ist. Dazu werden wir kurz die Features vorstellen, die ihr für eure eigenen Signale nutzen könnt. Danch erklären wir den Aufbau des CP und stellen dann die einzelnen Elemente vor. Zum Schluss gibt gehen wir kurz darauf ein, wie man das CP ins Spiel bekommt, wie man mit Bugs und Fehlern umgeht und das fertige CP veröffentlichen kann.

Zum Erstellen von CPs empfehlen wir euch folgende Programme als Hilfsmittel zu nutzen, natürlich könnt ihr auch andere Programme verwenden.

- **BlockBench** - Modellieren und Texturieren
- **GIMP** - Bildbearbeitung, Erstellen von Texturen
- **Visual Studio Code** - Bearbeiten von Json Dateien
- **7-zip** - Erstellen und Verwalten von Zip Dateien
- **Audacity** - Bearbeiten von Sound Dateien und abspeichern im .ogg Format.

Unsere Mod beruht darauf, dass die Signale nicht als Ganzes modelliert werden, sondern in kleine Teile, die entweder immer da sind oder man optional auswählen kann, aufgeteilt sind. Diese Teile werden dann durch das System im Spiel zu einem Signal zusammengesetzt. Dadurch wird es ermöglicht, dass man sich die Signale nach Belieben konfigurieren kann. Eine weitere grundlegende Struktur unserer Mod ist es, dass wir Texturen austauschen können, um unterschiedliche Signalbegriffe zu zeigen (z.B. die Lampen, Kennbuchstaben auf den Zs2 bzw. Kennziffern auf Zs3 oder die Textur der Mastschilder).
Ein weiterer Bestandteil unserer Mod ist das Stellwerk, welches einem ermöglicht über Fahrstraßen Signale und andere Elemente zu steuern. Damit CP Signale mit dem Stellwerk funktionieren, muss man dem Stellwerk natürlich sagen, welche Signalbegriffe für welche Situation gezeigt werden sollen.
Durch diese Individualität, ist es für den Laien nicht unbedingt immer einfach in das System einzusteigen, auch wenn wir uns Mühe gegeben haben, es trotzdem relativ simpel aufzubauen. Wenn ihr Fragen habt, könnt ihr euch jederzeit an uns wenden. Theoretisch ist auch in Planung einen Editor zu erstellen, der einem hilft die Json Dateien zu generieren, dies liegt aber noch in ferner Zukunft, da unsere Kapazitäten begrenzt sind und die aktuelle Prio auf der Signalmod liegt.

3 Features

Der Einfachheit halber werden wir hier in der Anleitung nicht auf die generellen Features der Open-Signals Mod eingehen, sondern nur die Features beleuchten, die für die CPs relevant sind.
Folgende Möglichkeiten habt ihr:

- Neue Modelle hinzufügen
- Neue Texturen hinzufügen
- Tint-Indizies verwenden
- Veränderbare Texturen für z.B. Lampen hinzufügen und einstellen
- Neue Placement-Tools hinzufügen
- Einen Schadenswert zum Item hinzufügen und individuell konfigurieren
- Neue Signale nach eigenem Belieben konfigurieren
- Individuelle Texte auf Signalen rendern und einstellen, z.B. Größe, Farbe, Positionierung

- Sounds zu den Signalen hinzufügen
- Einen Redstone Output definieren
- Signale mit dem Stellwerk kombatibel machen
- Eure Kreativität mit den Signalen ausleben
- Eigene Signalbrücken(-blöcke) hinzufügen

4 Struktur des Content Packs

Aufgebaut ist ein CP generell wie ein normales Resourcepack für Minecraft. Nur haben wir noch zusätzliche Ordner für unsere Systeme. Die genaue Erklärung, was für was zuständig ist, erfolgt im nächsten Kapitel. Ein CP Template könnt ihr **hier** herunterladen.

```
assets/
└─ opensignals/
   ├─ enumdefinition/
   ├─ lang/
   ├─ modeldefinitions/
   ├─ models/
   │  ├─ block/
   │  │  └─ contentpackname/
   │  └─ item/
   │     └─ contentpackname/
   ├─ signalbridge/
   ├─ signalconfigs/
   │  ├─ change/
   │  ├─ default/
   │  ├─ reset/
   │  ├─ shunting/
   │  ├─ subsidiary/
   │  └─ subsidiaryenums/
   ├─ signalsystems/
   ├─ sounds/
   │  └─ contentpackname/
   ├─ textures/
   │  ├─ blocks/
   │  │  └─ contentpackname/
   │  └─ items/
   │     └─ contentpackname/
   └─ tools/
```

Erklärungen

- **assets** ist der Basisordner. Muss existieren.
- **opensignals** indetifiziert das das CP zur Opensignals mod gehört. Muss existieren.
- **enumdefinition** enthält eine oder mehrere Dateien die neue Einstellungsmöglichkeiten hinzufügen. Sollte existieren.
- **lang** enthält die Übersetzungensdateien zu euren Einstellungen und Signalen. Sollte existieren.
- **modeldefinitions** enthält die Dateien die das Verhalten eurer Modelle steuern. Sollte existieren.

- **models** enthält alle Modelle für die Items und die Signalteile die nachher das Gesamtmodell ergeben. Sollte existieren.
- **signalbridge** enthält eine oder mehrere Dateien, um Signalbrückenblöcke zu registrieren.
- **signalconfigs** enthält möglichen Konfigurationsdateien für die Stellwerke. Kann existieren.
- **change** enthält Konfigurationen, die beim Stellen einer Fahrstraße genutzt werden. Kann existieren.
- **default** enthält Konfigurationen, die beim Stellen auf einen Endblock genutzt werden. Kann existieren.
- **reset** enthält Konfigurationen, die beim auflösen von Fahrstraßen genutzt werden. Kann existieren.
- **shunting** enthält Konfigurationen, die beim Stellen von Rangierfahrstraßen genutzt werden. Kann existieren.
- **subsidiary** enthält Konfigurationen, die Zusatzsignale bzw. Signale mit besonderen Auftrag hinzufügen. Kann existieren.
- **subsidiaryenums** hier können neue Zusatzsignale bzw. Signale mit besonderen Auftrag in eine oder mehreren Dateien definiert werden. Kann existieren.
- **signalsystems** enthält alle Definitionen für die Signalsysteme. Sollte existieren.
- **sounds** enthält alle Sounds und ihre Konfigurationen. Kann existieren.
- **textures** enthält alle Texturen die euere Block bzw. Itemmodelle benötigen. Sollte existieren.
- **tools** enthält alle Definitionen für selbst definierte Auswahl-Items. Kann existieren.

Um Kompatibilität mit anderen CPs und unseren eigenen Modellen und Texturen zu erhöhen sollte in den Ordnern **models/block/**, **models/item/**, **textures/blocks/** und **textures/items/** jeweils einen Unterordner mit eurem Contentpacknamen gemacht werden, wie es in der o.a. Struktur gemacht ist. Beachtet das ihr das dann aber auch bei euren Modellen angeben müsst!

Beachte: Außer bei den angegebenen Ordnern sollte bei keinem anderen Ordner ein Unterordner mit eurem CP-Namen gemacht werden, da die Dateien in diesen Unterordnern dann von unserem System nicht genutzt werden können.

5 Wie erstelle ich ein CP?

In diesen Kapitel gehen wir darauf ein, wie ihr ein CP erstellen könnt. Dabei erläutern wir die Registrierung von Signalsystemen (5.1), die Einstellungsmöglichkeiten (5.2) und das Auswahl-Item (5.3). Wir gehen nur kurz auf die Modelle und Texturen (5.4) ein und erklären wie die Modell Definition (5.5) aufgebaut ist. Zudem stellen wir vor, wie man Sounds zu den Signalen hinzufügen kann (5.6), Signale mit dem Stellwerk kompatibel macht (5.7) und zum Schluss wie die Sprachdateien funktionieren (5.9).

Das, unserer Meinung, sinnvollste Vorgehen besteht darin, sich vorab erst einmal Gedanken zu machen, was man für Signale hinzufügen will. Danach sollte man das Modell erstellen und die dazugehörigen Texturen inklusive Texturen, die ausgetauscht werden sollen, zB. für Lampen. Falls man Sounds verwendet, können diese nach Möglichkeit schon in den ersten Schritten hinzugefügt werden, damit man alle später benötigten „Assets“ direkt vorhanden hat. Für die Signale selbst muss dein dann Auswahl-Item registriert werden und das Signalsystem registriert werden mit allen Einstellmöglichkeiten. Danach müssen die einzelnen Modell-Teile zusammengesetzt werden und definiert werden, wie diese sich verhalten, je nach Einstellung. Optional kann dann das Signalsystem mit dem Stellwerk kompatibel gemacht werden und zu guter letzt müssen noch Sprachdateien geschrieben werden, damit die Anzeigenamen in Minecraft auch stimmen.

Vorab noch als wichtige Information:

Damit es keine Überschneidung mit anderen CPs gibt, ist es ratsam seine Signalsysteme einzigartig zu benennen, bspw. „jeronimo_hvsignal“. Daher ist es auch ratsam bei den Modellen und Texturen Unterordner mit eurem induviduellen Contentpack-Namen zu erstellen. Auch an vielen anderen Stellen sollte beachtet werden, dass die Namen und Werte einzigartig gewählt werden, um Überschneidungen mit der Hauptmod oder anderen CPs zu vermeiden. Bitte beachtet, dass es sich bei diesen Namen nur um interne Namen handelt und nicht (unbedingt) um die Anzeigenamen. Diese werden über die Sprachschlüssel und die Sprachdateien erstellt.

5.1 Signalsystem registrieren

Wenn ihr ein neues Signalsystem erstellen wollt, müsst ihr dieses registrieren. Dazu müsst ihr im **signalsystems/**-Ordner eine Json Datei erstellen mit eurem Signalsystem-Namen. Diese Datei enthält dann wichtige Informationen zu den Einstellungen des Systems, sowie die Eigenschaften (PROPERTY) der Signale. Der Dateiname ist dann der interne Name des Signalsystems und wird an einigen Stellen verwendet, u.A. den Stellwerkseinstellungen (5.7). Außerdem muss die Modelldefinitionsdatei (5.5) den gleichen Namen tragen.

```
{
   "systemProperties": {
        "placementToolName": "NAME",
        "canLink": "BOOLEAN",
        "isBridgeSignal": "BOOLEAN"
        "defaultItemDamage": "INT",
        "defaultHeight": "INT",
        "customNameRenderHeight": "FLOAT",
        "offsetX": "FLOAT",
        "offsetY": "FLOAT",
        "signScale": "FLOAT",
        "signWidth": "FLOAT",
        "autoscale": "BOOLEAN",
        "textColor": "INT",
        "doubleSidedText": {
            "RELATION...": "BOOLEAN"
        },
        "colors": [
            "COLOR..."
        ],
        "renderHeights": {
            "RELATION...": "FLOAT"
        },
        "signalHeights": {
            "RELATION...": "FLOAT"
        },
        "sounds": {
            "RELATION...": {
                "name": "SOUND",
                "length": "INT"
            }
        },
        "redstoneOutputs": {
            "RELATION..." : "PROPERTY"
        },
        "remoteRedstoneOutputs": {
            "RELATION..." : "PROPERTY"
        }
    },
    "seProperties": [
        {
            "name": "PROPERTY",
            "enumClass": "ENUM-LIST",
            "defaultState": "STATE",
            "changeableStage": "STAGE",
            "autoname": "BOOLEAN",
            "itemDamage": "INT",
            "dependencies": "RELATION..."
        },
        {
            "name": "PROPERTY",
            "defaultState": "BOOLEAN",
            "changeableStage": "STAGE",
            "autoname": "BOOLEAN",
            "itemDamage": "INT",
            "dependencies": "RELATION..."
        }
    ]
}

```

Listing 1: Aufbau des Signalsystems.

5.1.1 Erklärungen

- **systemProperties** gibt die grundlegenden Signalsystemeinstellungen an. Muss vorhanden sein.
- **placementToolName** gibt an, in welchem Auswahl-Item das Signalsystem registriert wird. Muss vorhanden sein und ist als String angegeben. Hier muss der interne Name des Items verwendet werden. Bsp:

  ```
  "placementToolName": "signalplacer"
  ```

  

  Wenn ihr eigene Placementtools erstellt, dann beachtet bitte das die Namen klein, ohne Leerzeichen sowie ohne Unterstriche intern gespeichert sind.

- **canLink** gibt an, ob man das Signal mit einem Controller oder Stellwerk verlinken kann. Kann vorhanden sein. Standardmäßig ist es angeschaltet (true), kann jedoch auch abgeschaltet werden (false).

  ```
  "canLink": false
  ```

- **isBridgeSignal** gibt an, ob das Signal für Signalbrücken ist. Kann vorhanden sein. Standard ist false.

  ```
  "isBridgeSignal": true
  ```

- **defaultItemDamage** gibt an wie viel Schaden das Item standardmäßig beim Platzieren von Signalen nimmt. Dazu kommt der Schadenswert der ausgewählten Signaleigenschaften. Ein Item hat 100 Schadenspunkte. Kann vorhanden sein.
  Standardwert ist 1. Es sind nur Ganzzahlen ≥ 0 erlaubt.

  ```
  "defaultItemDamage": 1
  ```

- **defaultHeight** gibt an, wie groß das Signal ist. Sollte vorhanden sein. Der Standardwert ist 1. Es sind nur Ganzzahlen ≥ 0 erlaubt. Wobei 0 einer Höhe von 1 Block entspricht.

  ```
  "defaultHeight": 1
  ```

- **customNameRenderHeight** Verschiebung des gerenderten Textes auf der Vertikalen Achse. Kann vorhanden sein. Aktiviert den gerenderten Textes, wenn es vorhanden ist. Es sind Kommazahlen erlaubt.

  ```
  "customNameRenderHeights": 1.4
  ```

- **offsetX** Verschiebung des gerenderten Textes nach Links-Rechts. Kann vorhanden sein. Ist an das customNameRenderHeight gebunden. Der Standardwert ist 0. Es sind Kommazahlen erlaubt.

  ```
  "offsetX": -0.3
  ```

- **offsetY** Verschiebung des gerenderten Textes vor und zurück. Kann vorhanden sein. Ist an das customNameRenderHeight gebunden. Der Standardwert ist 0. Es sind Kommazahlen erlaubt.

  ```
  "offsetY": 0.15
  ```

- **signScale** gibt den Vergrößerungsfaktor der Textgröße an. Kann vorhanden sein. Ist an das customNameRenderHeight gebunden. Der Standardwert ist 1. Es sind Kommazahlen erlaubt.

  ```
  "signScale": 3.5
  ```

- **signWidth** gibt die Breite des Textfeldes in Schriftpixeln an. Kann vorhanden sein. Ist an das customNameRenderHeight gebunden. Der Standardwert ist 22. Es sind Kommazahlen erlaubt.

  ```
  "signWidth": 22
  ```

- **autoScale** gibt an, ob die Größe des gerenderten Textes automatisch skaliert werden soll. Kann vorhanden sein. Standardmäßig ist es ausgeschaltet (false), kann jedoch auch angeschaltet werden (true).

  ```
  "autoScale": false
  ```

- **textColor** ist die Farbe für den gerenderten Text. Kann vorhanden sein. Es sind nur die Dezimalzahlen (16777215 - weiß) von den Farben zulässig. Den Wert zur Farbe kann man bspw. auf der Website Hexadecial / Decimal Colors herausfinden.

  ```
  "textColor": 16777215
  ```

- **doubleSidedText** gibt an, ob der Text auf der anderen Seite des angegebenen Signals angezeigt werden soll. Kann vorhanden sein. Es können mehere Einträge gemacht werden.

  ```
  "doubleSidedText": {"config(SH_HIGH.HIGH)": true}
  ```

- **colors** ist die Farbe für Tint-Indizes in den Modellen. Kann vorhanden sein. Die erste Zahl in der Liste ist der Index 0, die zweite Zahl der Index 1, usw. Es sind nur die Dezimalzahlen (16777215 - weiß) von den Farben zulässig. Den Wert zur Farbe kann man bspw. auf der Website Hexadecial / Decimal Colors herausfinden. Auf den Tintindex werden wir in 5.4 genauer eingehen.

  ```
  "colors": [16777215, 0]
  ```

- **renderHeights** legt fest unter welchen Bedingungen (RELATION) die Standardhöhe (FLOAT) des gerenderten Textes überschrieben werden soll. Kann vorhanden sein. Es können mehere Einträge gemacht werden. Es sind Kommazahlen erlaubt.

  ```
  "renderHeights": {"config(SH_HIGH.HIGH)": 2.373}
  ```

- **signalHeights** legt fest unter welchen Bedingungen (RELATION) die defaultHeight der Signale überschrieben wird. Kann verwendet werden. Es können mehrere Einträge gemacht werden. Es sind Ganzzahlen ≥ 0 erlaubt.

  ```
  "signalHeights": {"config(SH_HIGH.HIGH)": 2}
  ```

- **sounds** gibt an, unter welchen Bedingungen (RELATION) ein Sound (SOUND) abgespielt und wiederholt (INT) werden soll. Kann vorhanden sein. Es können mehrere Einträge gemacht werden. SOUND ist der interne Sound Name. INT gibt die zeitliche Länge des Sounds in Ticks an. Wobei 20 Ticks einer Sekunde entsprechen. Der Wert muss nur angegeben werden, wenn sich der Sound nach der angegebenen Zeit wiederholen soll. Dabei sollte der Wert der exakten Länge des Sounds entsprechen.

```
{
    "sounds": {
        "config(ac_blink_light.TRUE)": {
            "name": "andreas_cross",
            "length": 84
        }
    }
}
```

- **redstoneOutputs** gibt an, unter welchen Bedingungen (RELATION) und welcher Signaleigenschaften (PROPERTY) gebunden ist. redstoneOutputs ist, wenn man mit dem Signal direkt, durch Rechtsklick, interagiert, um einen Redstone Output umzuschalten. Kann vorhanden sein. Es können mehrere Einträge gemacht werden.

```
"redstoneOutput": {"config(manual.TRUE)": "powered"}
```

- **remoteRedstoneOutputs** gibt an, unter welchen Bedingungen (RELATION) und welcher Signaleigenschaften (PROPERTY) gebunden ist. remoteRedstoneOutputs ist, wenn man mit dem Signal über einen Signalcontroller interagiert, um einen Redstone Output umzuschalten. Kann vorhanden sein. Es können mehrere Einträge gemacht werden.

  ```
  "remoteRedstoneOutput": {"config(manual.FALSE)": "powered"}
  ```

- **seProperties** listet die einzelnen Signaleigenschaften (PROPERTY) auf. Signaleigenschaften sind die einzelnen Bauteile an einem Signal, die man einstellen können soll. Muss vorhanden sein. Kann aber eine leere Liste [ ] enthalten oder auch mehrere Einträge.
- **name** ist der interen Name der jeweiligen Signaleigenschaft (PROPERTY). Muss vorhanden sein. Der Name und ein entsprechender Wert (ENUM) wird an vielen anderen Stellen verwendet.

  ```
  {"name": "kombisignal_distant"}
  ```

- **enumClass** gibt die Liste der Einstellmöglichkeiten für diese Eigenschaft an. Sollte vorhanden sein, wenn man mehrere Einstellmöglichkeiten hat. Wenn eine Signaleigenschaft nur an oder abschaltbar sein kann auch Boolean verwendet werden, dann wird enumClass nicht angegeben.

  ```
  "enumClass": "KSDistant"
  ```

- **defaultState** gibt die Standardeinstellung der Eigenschaft an. Muss vorhanden sein. Wenn eine enumClass angegeben ist, muss ein valider Wert aus der Einstellungsliste angegeben werden. Der Aufbau und die Funktion wird in 5.2 genauer erläutert. Wenn es sich um einen Boolean handelt, muss „true“ oder „false“ angegeben werden.

  ```
  "defaultState": "OFF"
  ```

  oder

  ```
  "defaultState": true
  ```

- **changeableStage** gibt an, wo man diese Signaleigenschaft ändern kann. Sollte vorhanden sein. Standardeinstellung ist APISTAGE. Mögliche Stadien sind:
  - **APISTAGE**: Ist sowohl im Auswahl-Item als auch im Signalsteuerer.
  - **GUISTAGE**: Ist nur im Auswahl-Item.
  - **APISTAGE_NONE_CONFIG**: Ist nur im Signalsteuerer.

  ```
  "changeableStage": "GUISTAGE"
  ```

- **autoname** gibt an, ob zu den Einstellungen wie angegeben im Spiel benannt ist oder ob man diese manuell in der Sprachdatei ändern will. Kann angegeben werden. Standardwert ist „false “ also aus.

  ```
  "autoname": true
  ```

- **itemDamage** gibt an wie viel Schaden das Item standardmäßig beim Platzieren von Signalen nimmt. Dazu kommt der Schadenswert des Signalsystems und der anderen Signaleigenschaften. Ein Item hat 100 Schadenspunkte. Kann vorhanden sein. Standardwert ist 1. Es sind nur Ganzzahlen ≥ 0 erlaubt.

  ```
  "itemDamage": 1
  ```

- **dependencies** gibt an, an welche Bedingungen (RELATION) die Signaleigenschaft gebunden ist. Kann vorhanden sein. Die Methode ist in 1.2 genauer erklärt.

  ```
  "dependencies": "config(KOMBITYPE.DISTANT)"
  ```

5.2 Enum Definition/Einstellungsmöglichkeiten

Mit den Einstellmöglichkeiten ist es möglich für eine Signaleigenschaft mehrere Werte anzunehmen, bspw. bei Signalbildern. Dazu muss man im **enumdefinition/**-Ordner eine enums.json Datei anlegen, in der Listen von Einstellungsmöglichkeiten angelegt werden.

Der Name der Liste muss eindeutig sein, damit es keine Überschneidung mit anderen CPs gibt.

```
{
    "KSDistant": [
        "OFF", "KS1", "KS1_BLINK", "KS2"
    ],
    "HP": [
        "OFF", "HP0", "HP1", "HP2", "SHUNTING"
    ]
}
```

Listing 2: Aufbau der enums.json.

5.3 Placement-Tool erstellen

Ihr könnt ein eigenes Auswahl-Item erstellen, mit dem man eure Signalsysteme platzieren kann. Dazu müsst ihr im **tools/**-Ordner eine **tools.json** erstellen, die eine Liste von Auswahl-Items enthält. Wenn ihr ein neues Auswahl-Item registriert, benötigt ihr ein Item-Modell und eine Item Textur.

```
{
    "placementtools": ["PLACEMENT_TOOL"]
}
```

Listing 3: Placement-Tool hinzufügen in der tools.json.

Dieses Tool würde jetzt intern als "placementtool"gespeichert sein. I.d.R. braucht ihr nur ein Auswahl-Item. Ihr könnt auf bereits vorhandene zugreifen, jedoch empfehlen wir der Übersichtlichkeit halber für jedes CP ein eigenes Placement-Tool für eure Signalsysteme zu erstellen. Dabei sollte eure Tool einen eindeutigen Namen besitzen, damit es nicht zu Überschneidungen mit anderen CPs kommt.

5.4 Modelle und Texturen

Zum aktuellen Zeitpunkt unterstützt unser System nur valide Json-Blockmodelle. Zum Modellieren empfehlen wir daher mit Blockbench zu arbeiten. Texturen könnt ihr ebenfalls mit Blockbench erstellen und bearbeiten oder mit anderen Bildbearbeitungsprogrammen, wie bspw. GIMP. Aufgrund dessen, dass es im Internet zu genüge Tutorials gibt, wie die Programme funktionieren, gehen wir nicht näher darauf ein, wie Modelle und Texturen erstellt werden. Lediglich werden wir ein paar Besonderheiten, die es mit den Json Dateien gibt, näher erläutern.

Am besten ist, dass ihr das Signal nicht als Ganzes modelliert. Alle Teile, die später zuschaltbar, auswechselbar sind oder andere Funktionen haben, müssen jeweils als extra Modell gemacht werden. Diese werden dann in der Modelldefinition zu einem Signal mit den jeweiligen Einstellmöglichkeiten zusammengeführt.

Die Texturen sollten in **textures/blocks/*deincontentpack*/** gespeichert und die Modelle in
models/block/*deincontentpack*/ gespeichert werden, um Überschneidungen mit anderen CPs zu verhindern.
Es gibt noch 2 Besonderheiten für Modelle. Zum einen kann ein Tintindex für eine Textur festgelegt werden und zum anderen können über die Texturvariable in der Modelsdefinition (5.5) für Austausch-Texturen verwendet werden.

```
{
    "textures": {
        "1": "opensignals:blocks/default/shield_black",
        "15": "opensignals:blocks/zs3/off",
        "overlay": "opensignals:blocks/zs3/z10",
        "particle": "opensignals:blocks/default/shield_black"
    },
    "elements": [
        {
            "name": "overlay",
            "from": [5.5, 1, 3.9],
            "to": [10.5, 8, 3.9],
            "faces": {
                "north": {"uv": [0, 0, 5, 7], "textures": "#overlay", "tintindex": 1}
            }
        },
        {
            "..."
        }
    ]
}
```

Listing 4: Besonderheiten der Json Modelle.

5.4.1 Tintindex

Der Tintindex ist ein „Färbungsindex“, mit dem Texturen nachträglich umgefärbt werden können. Die Zahl gibt den jeweiligen Eintrag in der colors-Liste im Signalsystem (5.1) an. Wobei der erste Eintrag den Index 0 hat.

I.d.R. wird dieses Features nur für Spezialfälle verwendet und kommt regulär gar nicht zum Einsatz.

5.4.2 Texturvariablen

Die Texturvariablen können genutzt werden, um für diverse Einstellungen die Texturen überschreiben zu können. Dies wird in der Modelldefinition 5.5 gemacht, indem für die angegebene Texturvariable eine neue Textur festgelegt wird. Die Texturvariable kann natürlich auch umgenannt werden, um sie für sich selbst besser zuordnen zu können.

5.4.3 Leuchtende Texturen

Wenn OptiFine installiert ist, können Texturen verwendet werden, die einen Leuchteffekt („emissive“) haben. Dazu müssen die entsprechenden Texturen in dem selben Ordner doppelt vorhanden sein und eine Datei muss „_e“ am Ende des Namen enthalten.
Bsp: **textur.png** und **textur_e.png**. Ebenfalls muss in den OptiFine Einstellungen „Emissive-Texture“ aktiviert sein.

5.5 Modelldefinition erstellen

In der Modelldefinition werden die einzelnen Modelle zu einem Signalsystem zusammengesetzt. Dazu können auch Bedingungen angegeben werden, wann einzelne Modelle angezeigt werden. Außedem können Tauschtexturen angegeben werden.

```
{
    "textures": {
        "VAR...": "TEXTUREFILE"
    },
    "models": {
        "MODELFILE...": {
            "textures": [
                {
                    "loadOFFstate": "BOOLEAN"
                    "blockstate": "RELATION",
                    "retexture": {
                        "VARIABLE...": "VAR || TEXTUREFILE"
                    },
                    "extentions": {
                        "EXTENTION.json...": {
                            "PROPERTY": "VARIABLE"
                        }
                    },
                    "x": "INT",
                    "y": "INT",
                    "z": "INT"
                },
                {
                    "..."
                }
            ],
            "x": "INT",
            "y": "INT",
            "z": "INT"
        }
    }
}
```

Listing 5: Modelle registrieren.

5.5.1 Erklärungen

- **textures** Hier können Texturdateien (TEXTUREFILE) in internen Variablen (VAR) zwischengespeichert werden, damit nicht überall die Texturdatei angegeben werden muss. Die Texturdatei ist der Pfad + Dateiname. Kann vorhanden sein. Kann mehrere Einträge enthalten.

  ```
  "texture1": "opensignals:blocks/path/texture1"
  ```

- **models** Hier werden die einzelnen Modelle (MODELFILE) für das Signalsystem aufgelistet. Zu den Modellen können Bedingungen zu Blockzuständen und Tauschtexturen definiert werden. Muss vorhanden sein. Die MODELFILE besteht aus „Ordner“/„Datei“ausgehend von „models/blocks/“.

  ```
  "models": {"ks/ks_ne2": {"..."}}
  ```

- **textures** ist eine Liste von Elementen. Muss vorhanden sein, kann mehrere Elemente sowie eine leere Liste enthalten.

  ```
  "textures": [{"..."}]
  ```

- **loadOFFstate** gibt an, ob ein angegebener Off/Aus-Zustand geladen werden soll. Kann vorhanden sein. Standardwert ist „true“ also an.

  ```
  "loadOFFstate": true
  ```

  

  Wichtig ist, dass wenn ihr loadOFFState auf false setzt, dass ihr auch in euerer EnumClass einen Wert habt der OFF heißt, sonst funktioniert das nicht.

- **blockstate** enthält Bedingungen (RELATION), wann dieses Modell bzw. dieser Blockzustand geladen werden soll. Muss vorhanden sein. Kann leer gelassen werden. Die Methoden werden in 1.2 genauer erklärt.

  ```
  "blockstate": "with(KOMBISIGNAL.OFF)"
  ```

  ```
  "blockstate": "has(KOMBITYPE)"
  ```

  ```
  "blockstate": "hasandis(NE2)"
  ```

- **retexture** ermöglicht es einzelne Texturen in einem Modell auszutauschen. Dadurch können bspw. Teilmodelle für verschiedene Signalbegriffe auf ein Teilmodell reduziert werden. Kann vorhanden sein. VARIABLE ist die Texturvariable aus dem Modell. VAR ist die interne Texturvariable, die unter „textures“ definiert wurden, alternativ kann auch direkt die Texturdatei (TEXTUREFILE) angegeben werden.

  ```
  "retexture": {"1": "texture1"}
  ```

- **extentions** ist eine Erweiterung. Damit lassen sich gewisse Sachen, die bei mehreren Signalsystemen gleich sind, in eigene Dateien auslagern. Die Erweiterungsdatei (EXTENTION) liegt im **modeldefinitions/**-Ordner vor und muss wie folgt benannt werden: **test.extention.json** VARIABLE ist die Texturvariable aus dem Modell. PROPERTY gibt die zugehörige Signaleigenschaft an. Kann vorhanden sein. Kann mehrere Elemente enthalten.

  ```
  "extentions": {"test.json": {"TEST_PROPERTY": "3"}}
  ```

- **x** Verschiebung des Teilmodells oder Blockzustandes nach Links-Rechts. Kann vorhanden sein. Der Standardwert ist 0. Es sind Kommazahlen erlaubt.

  ```
  "x": 0
  ```

- **y** Vetikale Verschiebung des Teilmodells oder Blockzustandes. Sollte vorhanden sein. Der Standardwert ist 0. Es sind Kommazahlen erlaubt.

  ```
  "y": 0
  ```

- **z** Verschiebung des Teilmodells oder Blockzustandes vor- und zurück. Kann vorhanden sein. Der Standardwert ist 0. Es sind Kommazahlen erlaubt.

  ```
  "z": 0
  ```

Im folgenden wird die Extention-Datei vorgestellt.

```
{
    "extention": {
        "ENUM...": "TEXTUREFILE"
    }
}
```

Listing 6: Extension Datei.

5.5.2 Erklärungen Extention Datei

- **extention** ist eine Erweiterung für die Modeldefintion. Damit lassen sich gewissen Sachen aus der Modeldefinition separieren, die in mehreren Signalsystemen gleich ist und somit dann nicht mehrfach aufgeführt werden müssen. TEXTUREFILE ist der Pfad zur Texturdatei. ENUM gibt die Einstellung aus der Einstellungliste **enum/enums.json** an, für die der jeweilige Textur gilt.
  Kann mehere Einträge enthalten. Muss nicht existieren.

  ```
  "extention": {"WRW": "opensignals:blocks/mast_sign/wrw"}
  ```

Wir nutzen dieses Feature z.B. für die Mastschilder, da diese an jedem Licht-Signalsystem vorhanden ist. I.d.R. wird dies aber nicht benötigt und kann der Einfachheit halber weggelassen werden.

5.6 Sounds hinzufügen

Um Sounds zum den Signalen hinzuzufügen, müssen diese im **sounds/**-Ordner abgelegt werden.

Es wird empfohlen hier auch wieder einen Unterordner mit dem Contentpack Name zu erstellen, um die Kompatiblität zu anderen CPs zu erhöhen.

Der jeweilige Sound muss noch in der **sounds.json**, die im **opensignals/**-Hauptordner liegt, registriert werden.

```
{
    "SOUND": {"category": "block", "sounds": [{"name": "opensignals:CONTENTPACKNAME/FILE"}]},
    "andreas_cross": {"category": "block", "sounds": [{"name": "opensignals:andreas_cross"}]}
}
```

Listing 7: Sounds registrieren.

5.6.1 Erklärungen

- **SOUND** ist der interne Name des Sounds.
- **CONTENTPACKNAME** Es wird empfohlen einen Unterordner mit dem Contentpacknamen zu erstellen, um die Kompatibilät mit anderen CPs zu verbessern.
- **FILE** ist die Sound-Datei, die abgespielt verwendet werden soll.

5.7 Signal-Configs für die Stellwerke

Um die Signale mit dem Stellwerk kompatibel zu machen, muss dem Stellwerk gesagt werden, in welcher Situation, welches Signalbild gezeigt werden soll. Dazu wird zuerst die Ordnerstruktur vorgestellt und anschließend wird auf die einzelnen Systeme eingegangen. Es muss für jedes Signalsystem jeweils eine eigene Datei erstellt werden.

5.7.1 Ordnerstruktur

Die Dateien für die Stellwerke befinden sich in den folgenden Unterordnern.

```
signalconfigs/
├── change/
├── default/
├── reset/
├── shunting/
├── subsidiary/
└── subsidiaryenums/
```

- **change/** enthält Informationen, was das Signal anzeigen soll entsprechend des Signalbildes vom nächsten Signal.
- **default/** enthält Informationen, was das Signal anzeigen soll, wenn das nächste Signal ein Dummy-Signal ist.
- **reset/** enthält Informationen, was das Signal anzeigen soll, wenn die Fahrstraße aufgelöst wird.
- **shunting/** enthält Informationen, was das Signal anzeigen soll, wenn eine Rangierfahrstraße eingestellt wird.
- **subsidiary/** enthält Informationen für Signale mit besonderen Auftrag bzw. Zusatzsignale.
- **subsidiaryenums/** hier können neue Zusatzsignale oder Signale mit besonderem Auftrag o.ä. hinzugefügt werden, sofern sie noch nicht schon anderweitig definiert sind.

5.7.2 reset Einstellung

Die „reset“-Einstellung, ist die Einstellung, die geladen wird, wenn die Fahrstraße des Signals zurückgenommen wird. Daher sollte hier immer ein Halt-Begriff gezeigt werden.

```
{
    "currentSignal": "SIGNALSYSTEM",
    "values": {
        "RELATION...": ["PROPERTY.ENUM..."],
        "..."
    }
}
```

Listing 8: Reset Config.

- **currentSignal** ist der Name des aktuellen Signalsystems.
- **values** ist eine Zuordnung von Stellwerkseinstellungen und einer Liste an Signaleigenschaften (PROPERTY) und deren entsprechenden Wert (ENUM). „true“ wird immer aktiviert.

Es wird auch empfohlen alle anderen Teile des Signals auf einen „OFF-State“ zu stellen. Siehe folgendes Beispiel.

```
{
    "currentSignal": "hvsignal",
    "values": {
        "true": ["stopsignal.HP0", "hphome.HP0", "hpblock.HP0", "distantsignal.VR0",
        "zs3.OFF", "zs3v.OFF", "zs1.FALSE", "zs7.FALSE"],
        "!config(HPTYPE.OFF)": ["distantsignal.OFF"]
    }
}
```

Listing 9: Beispiel für Reset Config.

5.7.3 default Einstellung

Die „default“-Einstellung wird geladen, wenn das nächste Signal nicht existent ist oder es sich um ein Dummy Signal handelt. Also dieses keine weiteren Informationen bereitstellt.

```
{
    "currentSignal": "hvsignal",
    "values": {
        "speed(<.1) || speed(>.15)": ["stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.1)": ["zs3.Z1", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.2)": ["zs3.Z2", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.3)": ["zs3.Z3", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.4)": ["stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.5)": ["zs3.Z5", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.6)": ["zs3.Z6", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.7)": ["zs3.Z7", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.8)": ["zs3.Z8", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.9)": ["zs3.Z9", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.10)": ["zs3.Z10", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.11)": ["zs3.Z11", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.12)": ["zs3.Z12", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.13)": ["zs3.Z13", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.14)": ["zs3.Z14", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.15)": ["zs3.Z15", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "true": ["distantsignal.VR0", "zs3v.OFF"],

        "zs2value(A)": ["zs3.A"],
        "zs2value(B)": ["zs3.B"],
        "zs2value(C)": ["zs3.C"],
        "...": ["..."],
        "zs2value(ZS6)" : ["zs3.ZS6"],
        "zs2value(ZS13)": ["zs3.ZS13"]
    }
}
```

Listing 10: Beispiel für Default Config.

- **currentSignal** ist der Name des aktuellen Signalsystems.
- **values** ist eine Zuordnung von Stellwerkseinstellungen und einer Liste an Signaleigenschaften (PROPERTY) und deren entsprechenden Wert (ENUM).

Hier sollten immer nach der eingestellten Geschwindigkeit in der Fahrstraße die Signalbegriffe gefiltert werden. speed ≤ 0 und > 15 bedeutet keine Beschränkung. „true“ wird immer aktiviert. Ebenfalls werden die Zusatzsignal-Werte gesetzt.

5.7.4 change Einstellung

Die „change“-Einstellung wird geladen, wenn ein Ziel-Signal existiert, dieses Informationen über seinen Zustand liefert und somit das aktuelle Signal entsprechend drauf reagiert. Dazu muss je nachdem welche Information rein kommt, eingestellt werden, wie das aktuelle Signal reagieren soll.

Der Dateiname sollte idealerweise den Namen des aktuellen und des Zielsignals enthalten, um es entsprechend zu unterscheiden. Es können auch Dateien erstellt werden für die Vermischung von Signalsystemen. Also z.B. von HV (current) auf KS (next) Signale.

```
{
    "currentSignal": "SIGNALSYSTEM",
    "nextSignal": "SIGNALSYSTEM",
    "savedPredicates": {
        "VARIABLE...": "RELATION..."
    },
    "values": {
        "map(VARIABLE)...": ["PROPPERTY.ENUM..."],

        "signalRepeater(true)": ["PROPPERTY.ENUM..."],

        "RELATION": ["PROPPERTY.ENUM..."],

        "speed(XYZ)": ["PROPPERTY.ENUM..."],

        "zs2value(XYZ)": ["PROPPERTY.ENUM..."]
    }
}
```

Listing 11: Change Config.

- **currentSignal** ist der Name des aktuellen Signalsystems.
- **nextSignal** vist der Name des nächsten Signalsystems.
- **savedPredicates** kann Bedingungen (RELATION) mit der config()-Methode in eigens definierten Variablen gespeichert werden, wenn man Code-Schnipsel hat, die wiederholt vorkommen, man aber nicht jedes Mal neu schreiben will. Kann vorhanden sein.
- **values** ist eine Zuordnung von Stellwerkseinstellungen und einer Liste an Signaleigenschaften (PROPERTY) und deren entsprechenden Wert (ENUM).
  - **map(VARIABLE)** ruft die gespeicherten Werte (RELATION) jeweilige Variable auf. Nimmt die Infos vom nächsten Signal.
  - **signalRepeater(true)** wenn das Vorsignal ein Wiederholer ist, kann im Stellwerk dies eingestellt werden. Damit die Signalbilder richtig gezeigt werden, muss hier angegeben werden, was gezeigt werden soll. signalRepeater(true) kann mit map(VARIABLE) verknüpft werden.
  - **RELATION** gibt direkt die Bedingung (RELATION) mit der config()-Methode an. Nimmt die Infos vom nächsten Signal.
  - **speed(XYZ)** ist für die eingestellte Geschwindigkeit in der aktuellen Fahrstraße. Nimmt die Infos aus der aktuellen Fahrstraße.
  - **zs2value(XYZ)** setzt für die aktuelle Fahrstraße die Zusatzsignale. Nimmt die Infos aus der aktuellen Fahrstraße.

```
{
    "currentSignal": "hvsignal",
    "nextSignal": "hvsignal",
    "savedPredicates": {
        "stop": "config(stopsignal.HP0) || config(stopsignal.SHUNTING)
        || config(hphome.HP0) || config(hphome.HP0_ALTERNATE_RED) || config(hpblock.HP0)",
        "drive": "config(stopsignal.HP1) || config(hphome.HP1) || config(hpblock.HP1)",
        "slow": "config(stopsignal.HP2) || config(hphome.HP2)"
    },
    "values": {
        "map(stop)": ["distantsignal.VR0", "zs3v.OFF"],
        "map(drive)": ["distantsignal.VR1"],
        "map(slow) && signalRepeater(true)": ["distantsignal.VR2"],

        "(config(zs3.Z1) || config(zs3plate.Z1))": ["zs3v.Z1"],
        "(config(zs3.Z2) || config(zs3plate.Z2))": ["zs3v.Z2"],
        "...": ["..."],
        "(config(zs3.Z15) || config(zs3plate.Z15))": ["zs3v.Z15"],

        "speed(<.1) || speed(>.15)": ["stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],
        "speed(==.1)": ["zs3.Z1", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "speed(==.2)": ["zs3.Z2", "stopsignal.HP2", "hphome.HP2", "hpblock.HP1"],
        "...": ["..."],
        "speed(==.15)": ["zs3.Z15", "stopsignal.HP1", "hphome.HP1", "hpblock.HP1"],

        "zs2value(A)": ["zs3.A"],
        "zs2value(B)": ["zs3.B"],
        "zs2value(C)": ["zs3.C"],
        "...": ["..."],
        "zs2value(ZS6)" : ["zs3.ZS6"],
        "zs2value(ZS13)": ["zs3.ZS13"],

        "config(zs3.A)": ["zs3v.A"],
        "config(zs3.B)": ["zs3v.B"],
        "config(zs3.C)": ["zs3v.C"],
        "...": ["..."]
    }
}
```

Listing 12: Beispiel für Change Config.

5.7.5 shunting Einstellung

Die „shunting“-Einstellung ist für Rangiersignale und Hauptsignale mit Rangierfunktion, um Rangierfahrten zu signalisieren.

```
{
    "currentSignal": "hvsignal",
    "values": [
        "stopsignal.SHUNTING"
    ]
}
```

Listing 13: Beispiel für Shunting Config.

- **currentSignal** ist der Name des aktuellen Signalsystems.
- **values** ist eine Liste an Signaleigenschaften (PROPERTY) und den entsprechenden Wert (ENUM).

5.7.6 subsidiary Einstellung

Die „subsidiary“-Einstellung gibt an, welche Zusatzsignale angezeigt werden sollen, wenn man das entsprechende einstellt. Dies ist für die Zusatzsignale, die man via Rechtsklick auf das Signal einstellen direkt aktivieren kann.

```
{
    "currentSignal": "hvsignal",
    "allStates": {
        "ZS1": ["zs1.TRUE"],
        "ZS7": ["zs7.TRUE"],
        "ZS8": ["zs3.ZS8"],
        "ZP9": ["zs3.ZP9"]
    }
}
```

Listing 14: Beispiel für Subsidiary Config.

- **currentSignal** ist der Name des aktuellen Signalsystems.
- **allStates** gibt für die verschiedenen Zusatzsignale an, welche Signaleigenschaften (PROPERTY) mit dem entsprechenden Wert (ENUM) geladen werden soll.

In „subsidiaryenums“ können auch eigene Zusatzsignale in einer eigenen Datei hinzugefügt werden, die man manuell aktivieren kann und in der Subsidiary Config für die jeweiligen Signalsysteme definiert werden. Die Standardeinträge sind immer vorhanden. Es können mehrere Einträge gemacht werden.

```
{
    "subsidiaryStates": [
        {
            "name": "NAME",
            "showSubsidiaryAtSignal": "SIGNALASPECT",
            "isCountable": "BOOLEAN"
        },
        {
        "..."
        }
    ]
}
```

Listing 15: Aufbau SubsidiaryEnums.

- **name** gibt den Namen des Zusatzsignals an.
- **showSubsidiaryAtSignal** gibt an, wie das Signal im Stellwerk aussehen soll, wenn das Zusatzsignal aktiv ist. Es können folgende Werte eingetragen werden: *SIGNAL_RED*, *SIGNAL_GREEN*, *SIGNAL_OFF*
- **isCountable** gibt an, ob das Zählwerk die Bedienung dieses Signals zählen soll oder nicht.

```
{
    "subsidiaryStates": [
        {
            "name": "Zs1",
            "showSubsidiaryAtSignal": "SIGNAL_RED",
            "isCountable": true
        }
    ]
}
```

Listing 16: Beispiel für SubsidiaryEnums.

5.8 Signalbrücken

Um eigene Signalbrücken mit eigenen Blöcken zu erstellen, müssen diese registriert werden, was unten beschrieben steht.

Wenn du eigene Signale oder Signalsysteme für Signalbrücken machen willst, muss in die Signalsystem-Datei bei den SystemProperties der Eintrag

`"isBridgeSignal": true`

ergänzt werden.

Die Datei kann einen beliebigen Namen tragen und ist wie folgt aufgebaut:

```
{
    "BLOCKNAME": {
        "type": "TYPE"
    },
    "...": {
    }
}
```

Listing 17: Signalbrücken Blöcke registrieren.

- **blockname** gibt den Namen des Blocks an. Der Name muss einzigartig sein.
- **type** gibt an, um welchen Typ von Signalbrückenblock es sich handelt. Es können folgende Werte eingetragen werden: *BASE*, *MAST*, *MAST_HEAD*, *CANTILEVER*, *CANTILEVER_END*

```
{
    "signalbridge_mast": {
        "type": "MAST"
    },
    "signalbridge_cantilever": {
        "type": "CANTILEVER"
    }
}
```

Listing 18: Beispiel: Signalbrücken Blöcke registrieren.

Zu den Signalbrückenblöcken müssen blockstate Dateien erstellt werden mit den Varianten *facing = east*, *facing = north*, *facing = south*, *facing = west*. Wir werden an der Stelle nicht näher auf blockstate Dateien eingehen. Dazu könnt ihr im Internet mehr finden, wenn ihr schaut, wie ein Standard Minecraft Ressourcenpack aufgebaut ist. Oder euch vorhandene blockstate Dateien in der Signalmod anschaut. Es sei folgendes Beispiel gegeben:

```
{
    "variants": {
        "facing=east": {
            "model": "opensignals:block/signalbridge/default/signalbridge_base",
            "y": 90
        },
        "facing=north": {
            "model": "opensignals:block/signalbridge/default/signalbridge_base"
        },
        "facing=south": {
            "model": "opensignals:block/signalbridge/default/signalbridge_base",
            "y": 180
        },
        "facing=west": {
            "model": "opensignals:block/signalbridge/default/signalbridge_base",
            "y": 270
        }
    }
}
```

Listing 19: Beispiel einer blockstate Datei für Signalbrücken

5.9 Sprachdateien erstellen

Für die Übersetzungen und Anzeigenamen im Spiel müssen die internen Sprachschlüssel mit Übersetzung in der Sprachdatei angegeben werden. Es sollte immer die englische (**en_us**) Datei vorhanden sein, weil diese Standard von Minecraft ist. Zusätzlich könnt ihr noch andere Übersetzungen angeben, wie z.B. Deutsch (**de_de**).

Beachte: Für Minecraft Version 1.12 und früher werden die Sprachdateien im eigenen **.lang** Format gespeichert. Ab Minecraft Version 1.13 dann im **.json** Format.

Die Sprachschlüssel kriegt ihr am besten raus, wenn ihr das Spiel ohne diese Dateien startet und sowohl im Auswahlitem als auch im Signalcontroller euch euere Signale genau anschaut und die „Schlüssel“ notiert.

```
{
    "block.opensignals.kssignal": "Ks - Combination signal"
}
```

Listing 20: Sprachdatei im Json Format.

```
tile.kssignal.name=Ks Signal
```

Listing 21: Sprachdatei im Lang Format.

6 Das Contentpack ins Spiel bringen

Das CP muss als komprimierte Zip-Datei in **minecraft/contentpacks/opensignals/** vorliegen, um ins Spiel geladen zu werden. Wenn man auf einem Server spielt, muss das CP sowohl auf dem Client als auch auf dem Server vorliegen.

7 Bugs und Fehler

Wenn Bugs und Fehler auftreten, ist es hilfreich erst einmal selbst in die Log Datei zu schauen, um herauszufinden wo das Problem herkommt. Meist fehlen irgendwelche Klammern oder „ “ in einer Json Datei. Wenn ihr nicht weiter kommt oder Fragen habt, könnt ihr euch auf unseren Discord im Support-Forum melden. Bitte versucht euren Fehler genau zu beschreiben, sowie die Log-Datei und Fehlermeldung mit anzuhängen sowie das fehlerhafte CP.

8 Contentpack Veröffentlichen

Wir geben später eine Empfehlung zur Veröffentlichung bekannt. Was wir euch aber schon empfehlen können ist, das CP als Resourcepack auf CurseForge hochzuladen. Es wäre wünschenswert, dass ihr eure CP mit der Allgemeinheit teilt, da wir unsere Mod ja auch mit der Allgemeinheit teilen und OpenSource haben.
Wenn ihr uns Bescheid gebt, werden wir euer Pack auch bei uns verlinken, sofern sie unseren Konventionen entsprechen.