EEP Lua Automatic Block Control

Rudy Boer, Frank Buchholz, April 2022
Version 2.1.0

Contents:

Purpose

This Lua software generates automatic train traffic on any EEP layout, without the need to write much Lua code yourself. All it takes to control a layout is to enter some data, on trains and on turnouts that need to be switched for a certain route, into a set of Lua tables and variables.

Versions

Version 2.0 is the initial version of this package from April 2022.

Version 2.1 from May 2022 contains corrections of the module and shows a new demo layout using double slip turnouts.

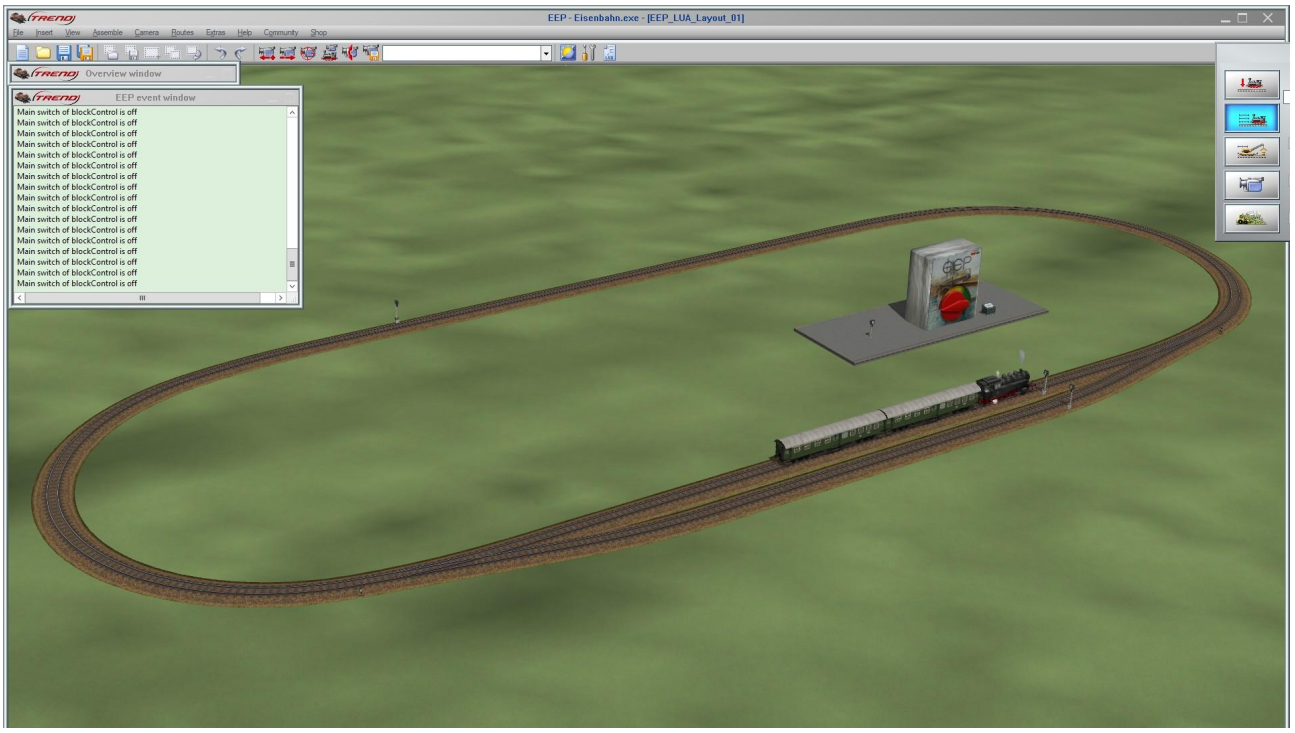
Contents of the `zip` file

User manual
Available in both English and German.

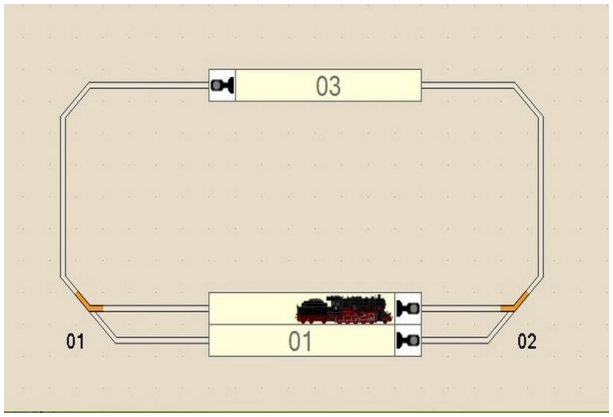
The file `blockControl.lua`
This file needs to be placed in your EEP program installation `\LUA` folder.

Demo files `EEP_Lua_Layout_01 ... _05` etc.
The demo layouts are described in this manual to serve as examples on how to configure the Lua data tables that define a layout.

Demo EEP_Lua_Layout_01



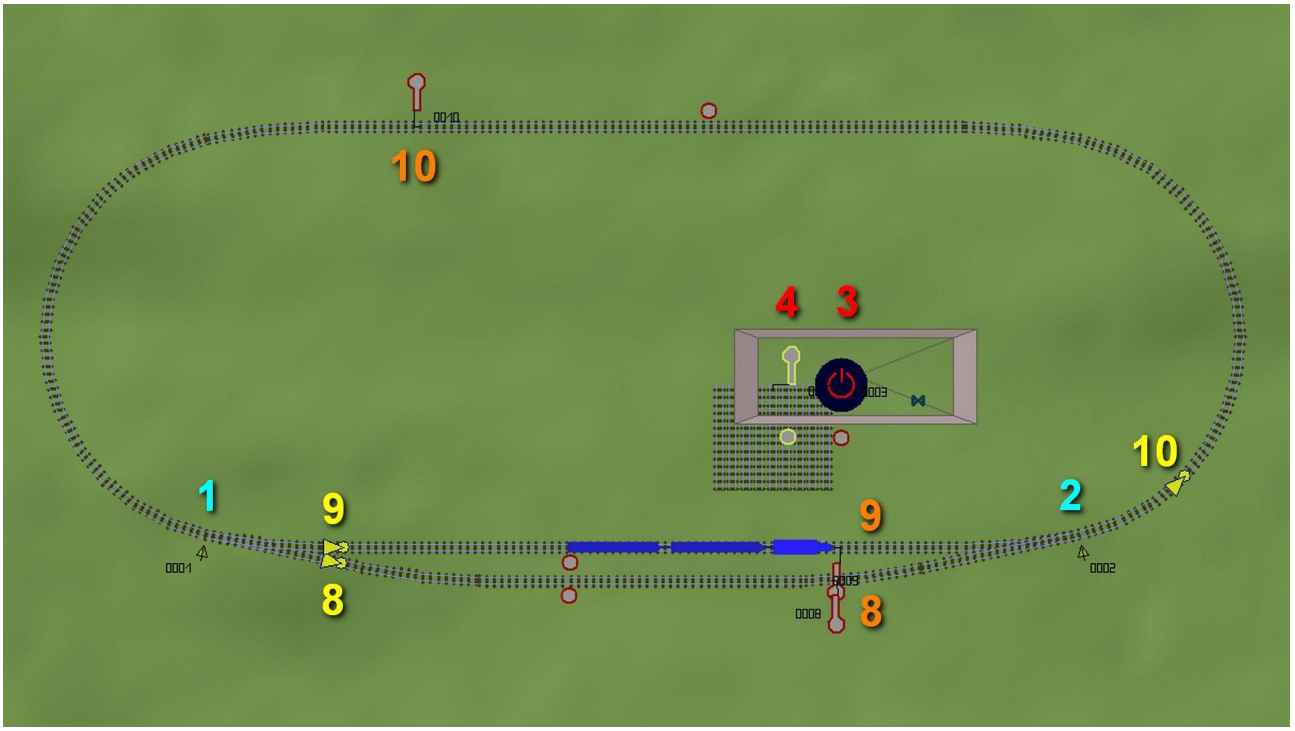
Let's write the Lua configuration for this simple layout and have the train drive around fully automatic, with waiting times at the station.



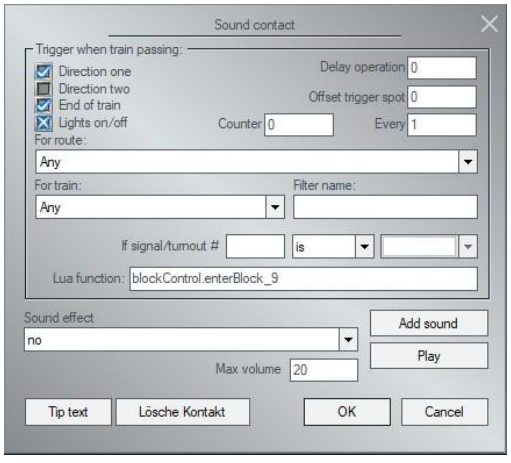
With automatic traffic, trains drive from block to block. The first decision to make is which blocks we want to define. There's only one rule here: **turnouts can not be part of a block, they are part of the routes between blocks.**

For this layout it seems logical to have the three blocks as shown here.

To define a block in EEP, a signal is placed at the position where we want the train to stop.



The picture above shows the turnout numbers (cyan), the block signal numbers (orange) and the on / off switch numbers (red) for this layout. If you create a similar layout your numbers may differ, as they are automatically generated by EEP, we can't choose them.



Besides the signal that is placed at the end of a block, a track contact has to be placed in every block at its entry. Note that there should not be any turnouts between the contact and the signal.

These block entry contacts usually are of the type "Sound" [1]. A reference to a Lua function has to be entered, like: `blockControl.enterBlock_9`[2]. The number must be that of the block signal. This contact now lets Lua know when a train has arrived in the block.

In most cases 'End of train' is the safe choice, previous blocks and turnouts will be released once the final wagon has passed the contact. If there is no opposing traffic via the turnouts behind the train, this tick can be left away, in which case previous blocks and turnouts are released as soon as the head of the train passes the contact, which may lead to a somewhat quicker follow up of trains.

[1] Any kind of contact works well. The important part is to enter the Lua function which tells the module about trains entering a block.

[2] Keep in mind that you have to work in a specific order:

1. Place block signals and optionally the contacts (but without referring to a Lua function in these contacts - the field has to be empty)
2. Adjust the Lua code to list all block signals in some of the tables
3. Run the layout in 3D mode
4. Place the contacts if not already done
5. Enter the reference to the Lua function for the block signals into the contacts

These restrictions can be avoided by using the Lua module "BetterContacts".

The configuration code for `EEP_Lua_layout_01` is:

```
main_switch = 3

local trains = {
  { name = "Steam", signal = 4, allowed = {[8]=15,[9]=1,[10]=1,}},
}

local routes = {
  { 8,10, turn={2,1} },
  { 9,10, turn={2,2} },
  {10, 8, turn={1,1} },
  {10, 9, turn={1,2} },
}
```

This is all that's needed to describe this layout. If you'd open the "Lua script editor" window, or if you'd open the Lua file in an editor like Notepad++, you'll notice the code actually is longer than this, but it's only this top part that defines the layout. The bottom part of the Lua code only requires editing if the states of the signals that are used differ (see the chapter on "Signal States") or if you like to change some of the available options (see the chapter on "Options").

`main_switch = 3`
The ID number of the signal that functions as the main on / off switch for the blockControl module is given here.

```
local trains = {
  { name = "Steam", signal = 4, allowed = {[8]=15,[9]=1,[10]=1,}},
}
```

`name = "Steam"`
For the automatic train detection to work, the train name (with or without leading "#") used here has to be identical to the name entered in the popup window when the train was placed on the track. See the chapter on "How to Place Trains on the Track and Initialize Lua".

`signal = 4`
Every train could have its own individual start / stop signal. This ID number is given here[3].

`allowed = {[8]=1,[9]=15,[10]=1,}`
The `allowed` sub-table specifies which blocks (identified by their signal number) this train is allowed to drive to and if there is a stop time:

- If a block is not mentioned[4] it means this train will never go there. If for example you won't allow the train in block 8, it should read: `allowed = {[9]=15,[10]=1,}`
- A block mentioned like `[8]=1` means this train will use block 8 and it will only stop if the signal stays red because blocks or turnouts ahead are not free yet.
- A block mentioned like `[9]=15` means this train will use block 9 and it will stay in the block for at least 15 seconds. This includes the drive time from the sensor to the signal, which depends on the speed of the train, the length of the block and the position of the pre-signal. For higher accuracy stop times you may want to measure this drive time.
- If the allowed sub-table is omitted this train can drive to every block and it will not have any stop times.

```
local routes = {
```

[3] You can reuse the same signal for multiple trains which you want to start or stop together.
[4] …or the block has value `nil` or 0

```
  { 8,10, turn={2,1} },
  { 9,10, turn={2,2} },
  {10, 8, turn={1,1} },
  {10, 9, turn={1,2} },
}
```

The routes table tells Lua how the turnouts have to be switched[5] when driving from one block to the next[6]. In this example there's exactly one turnout between the blocks of all routes. There could also be none[7], or there could be multiple turnouts between blocks:

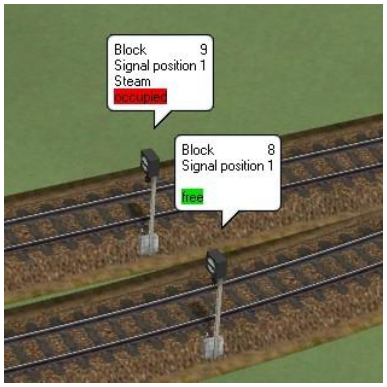
- If blocks 13 and 14 don't have a turnout between them the route description would look like this: `{13,14, turn={} },`
- If blocks 23 and 24 have three turnouts, with numbers 14, 5 and 6, in between them the route description would look like this: `{23,24, turn={14,1, 5,2, 6,1} },`

You can find a video about this layout 01 here:
https://www.youtube.com/watch?v=6X1fmBAHgpY&ab_channel=Rudysmodelrailway

Signal States

Unfortunately in EEP not all signal types have the same states. For some signals you see stop / 'red' = 1 while for others stop / 'red' = 2. We have to tell Lua if the states for red and green of the signals that are used on the layout is 1 or 2.

BEWARE: all block signals used on the layout need to have the same states. The same holds for all train signals.



Signal states are shown in the popups that show when a layout is first started as "Signal position #". They can easily be read out here and the Lua code can be changed accordingly if needed. This is done in the code lines that read[8]:

```
MAINON      = 1, -- ON    state of main switch
MAINOFF     = 2, -- OFF   state of main switch
BLKSIGRED   = 1, -- RED   state of block signals
BLKSIGGRN   = 2, -- GREEN state of block signals
TRAINSIGRED = 1, -- RED   state of train signals
TRAINSIGGRN = 2, -- GREEN state of train signals
```

To turn these popups on or off, quickly toggle the main switch twice via Shift + left click.

How to place trains on the track and initialize Lua

When the Lua code is run for the first time, or after clicking "Reload script" in the "Lua script editor" window, Lua starts in "*find mode for trains*". It will automatically find any train that is halted by a red signal.

---

[5] The exact value of the position for a turnout does not matter if the train cuts the turnout open, however, it's good practice to provide correct values in any case.
[6] The order of the from and to block is important but not the exact position. You can define routes this way as well: `{ 23, turn={14,1 5,2, 6,1,}, 24 }`
[7] If there is no turnout you can define an empty list for parameter `turn` or you can omit this parameter.
[8] You do not need to define signal states if the default matches the default values as described.

If there are no trains on the layout yet, we have to place at least one. Enable 3D edit mode in the “Control dialogue” window and place a train in front of the pre-signal of a block where you allow this train to drive. In the popup window that opens enter the train name, 100% identical to the name it has been, or is going to be, given in the Lua configuration.

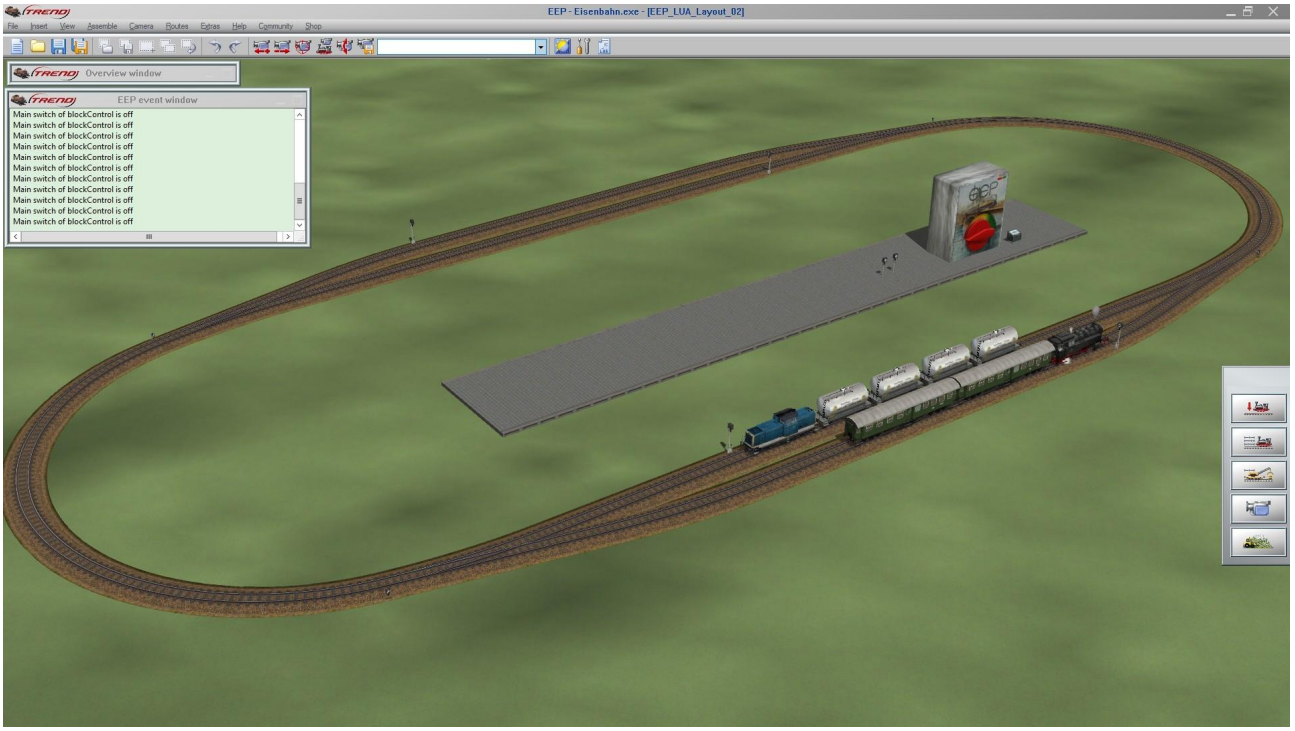
Now switch to drive mode in the “Control dialogue” window. Select the train via a left mouse click and manually enter a train speed. The train will drive to the block signal, where it will stop. Lua will now have detected the train, it tells about this fact in the EEP event window.

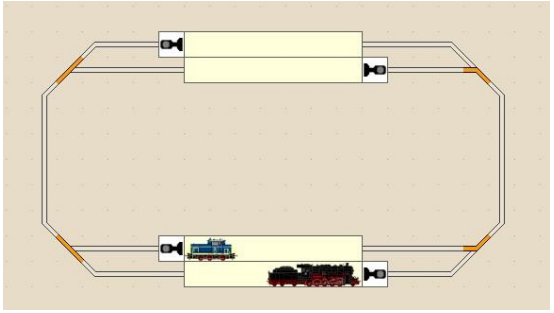
When all desired trains have been placed and detected, the main switch can be switched on, followed by the individual train switch(es). Lua will automatically control the train(s) from here on.

In case you later want to add or remove trains, open the “Lua script editor” window and click “Reload script”. Lua will now be in “*find mode for trains*” again.

Demo EEP_Lua_Layout_02

Let’s add a block and a second train and have traffic in two directions.

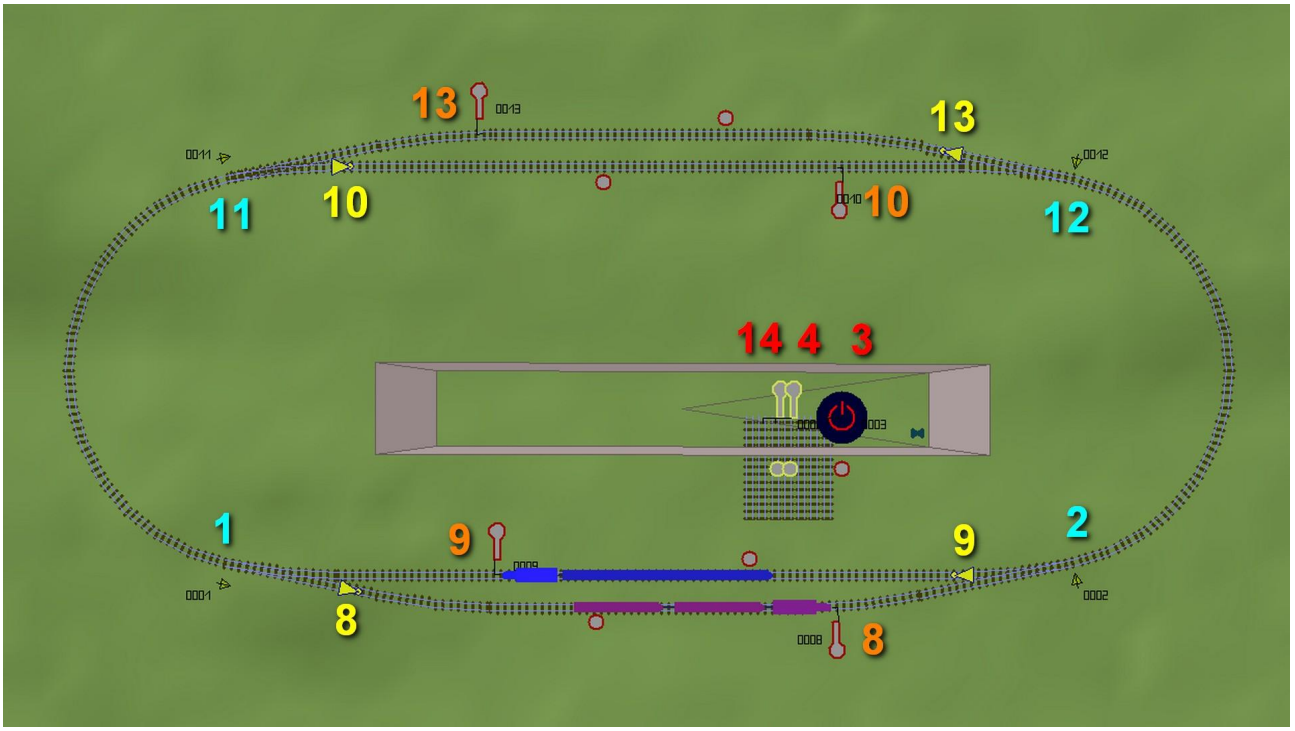




For this layout we define 4 blocks, 2 at station North, 2 at station South. Even though trains drive in opposite directions, the traffic in all blocks is still in one direction. We’ll see how to create blocks with traffic in two directions in the next demo layout 03.

The 2D overview below shows the turnouts (cyan), the block signals (orange) and the on / off switch signals (red). Because we now have opposing traffic we need to take care that the block entry sensors are set to

trigger on “End of train”, otherwise the turnout could be released too soon and the opposing train may start to drive already while the tail of the train that enters the station still is on the turnout.



The configuration code for `EEP_Lua_layout_02` is:

```
local main_signal = 3

local counterclockwise = {[8]=15, [13]=1, } -- allowed blocks for CCW trains
local clockwise        = {[9]=20, [10]=1, } -- allowed blocks for CW  trains

local trains = {
  { name = "Steam", signal = 14, allowed = counterclockwise },
  { name = "Blue",  signal =  4, allowed = clockwise        },
}

local routes = {
  {  8, 13, turn={  2,1, 12,1 }}, -- from block A, to block B, turnouts
  {  9, 10, turn={  1,2, 11,2 }},
  { 10,  9, turn={ 12,2,  2,2 }},
  { 13,  8, turn={ 11,1,  1,1 }},
}
```

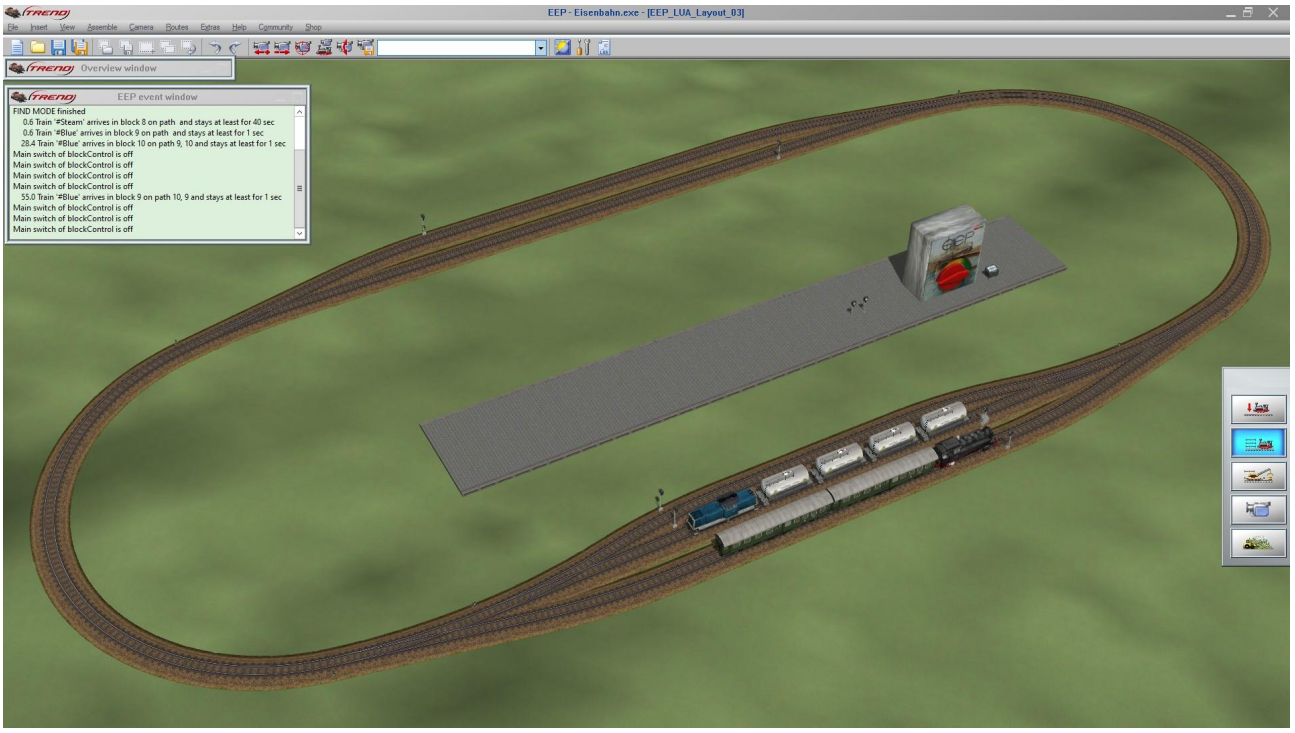
In this example the allowed blocks are denoted in groups in separate tables. One table specifies the allowed blocks, and the waiting times, for “counterclockwise” trains. The other table is likewise for “clockwise” trains. These are just names, you can use any name, like “cargo_trains” or “ICE”, as long as the same names are used in the `trains` table.

In this layout, with just two trains, the benefit of specifying allowed blocks this way may seem small. On a larger layout though, with multiple trains that are allowed on the same blocks and multiple other trains that are allowed on other blocks, this allowed block grouping becomes quite beneficial as it avoids endless repetition of identical “allowed” sub-tables for every train.

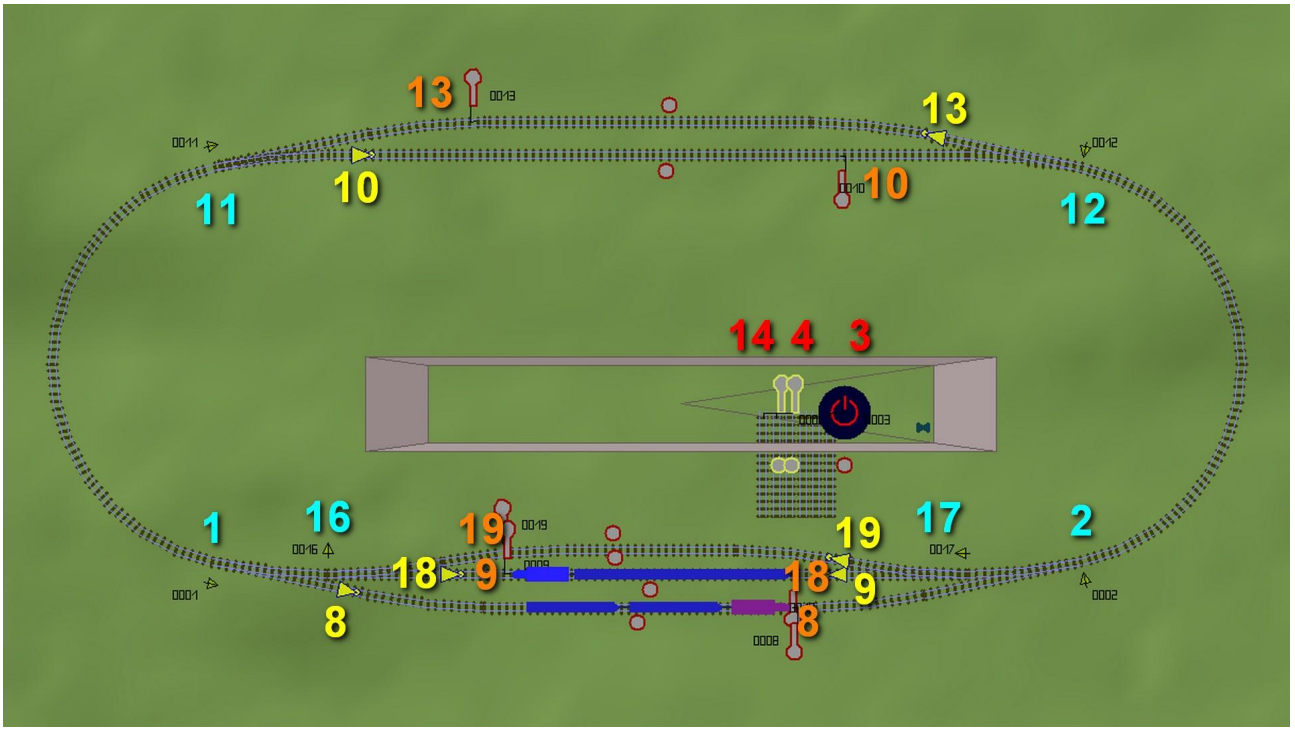
You can find a video about this layout 02 here:
https://www.youtube.com/watch?v=qEFNnP-s14c&ab_channel=RudysmodelIrailway

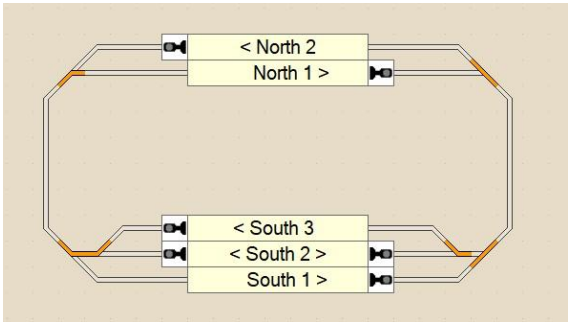
Demo EEP_Lua_Layout_03

Let's have two way traffic in a block now. Station South is extended with a middle track on which we plan to allow trains to drive both Eastbound and Westbound.



This means we need to place a signal at both ends of this track, they have numbers 9 and 18 in the image below. We also need a block entry sensor at the opposing sides. This effectively turns this block into a pair of overlapping twin blocks.





As soon as a train reserves one of these twin blocks, Lua also has to reserve the other block. The same holds when the train arrives at its destination block and the departure block can be released, the twin block then also has to be released.

So, we will have to tell Lua about these two way twin blocks. This is done via the `two_way_blocks` table.

The configuration code for `EEP_Lua_layout_03` is:

```
local main_signal = 3

local counterclockwise = { -- allowed blocks for CCW trains
  [ 8] = 40, -- station South track 1 -> East
  [18] =  1, -- station South track 2 -> East
  [13] =  1, -- station North track 2 -> West
}

local clockwise = { -- allowed blocks for CW trains
  [ 9] =  1, -- station South track 2 -> West
  [19] = 20, -- station South track 3 -> West
  [10] =  1, -- station North track 1 -> East
}

local trains = {
  { name = "Steam", signal = 14, allowed = counterclockwise },
  { name = "Blue",  signal =  4, allowed = clockwise },
}

local two_way_blocks = { {9, 18} }
```

```
local routes = {
  {  8, 13, turn={  2,1, 12,1 }}, -- from block A, to block B, turnouts
  { 18, 13, turn={  2,2, 12,1, 17,1 }},
  {  9, 10, turn={ 16,1,  1,2, 11,2 }},
  { 19, 10, turn={ 16,2,  1,2, 11,2 }},
  { 10,  9, turn={ 12,2,  2,2, 17,1 }},
  { 10, 19, turn={ 12,2,  2,2, 17,2 }},
  { 13,  8, turn={ 11,1,  1,1 }},
  { 13, 18, turn={ 11,1,  1,2, 16,1 }},
}
```

The `two_way_blocks` table is only needed if there's at least one pair of two way blocks to declare. If there are none, the table can simply be omitted. The twin blocks are always declared in pairs. If there are multiple two way blocks, the table could look like this[9]:

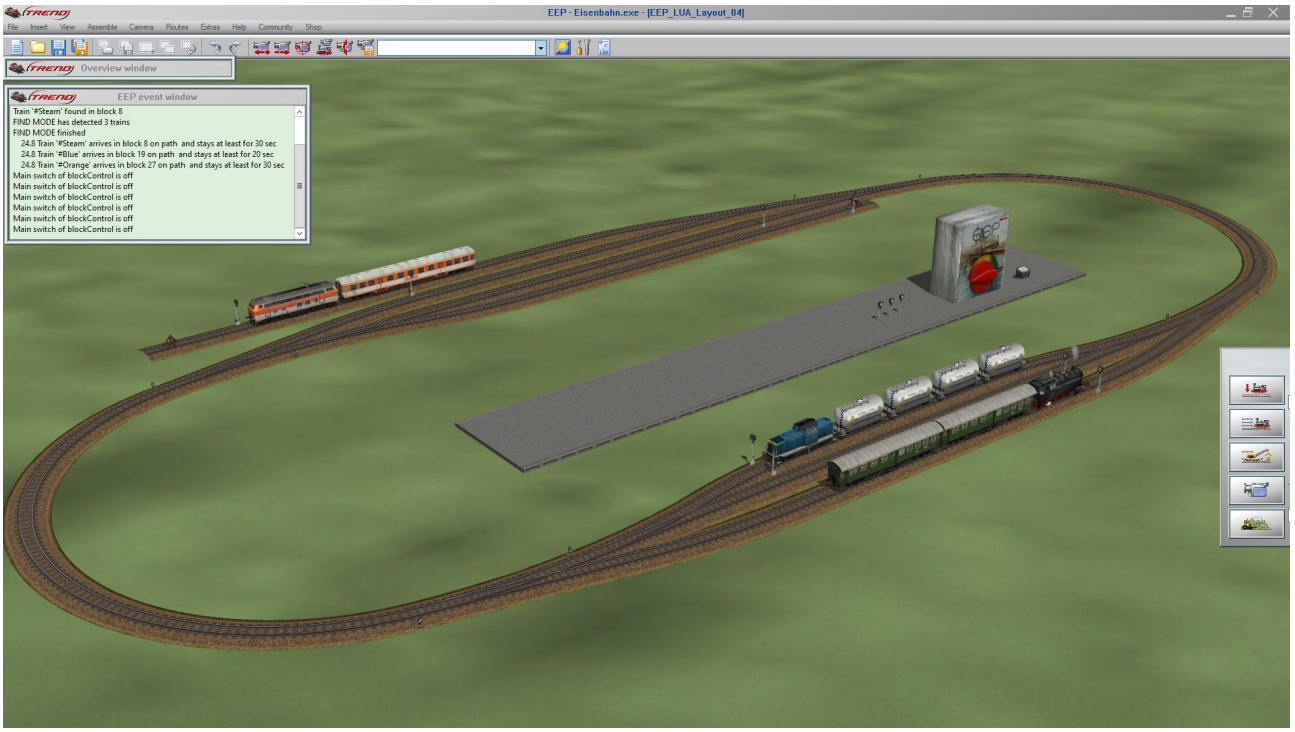
```
local two_way_blocks = { {82, 81}, {32, 33}, {74, 38}, }
```

What also changed is the way the allowed blocks are listed. Using multiple lines like shown here makes it possible to add comments to describe which block is which on larger layouts.

You can find a video about this layout 03 here:
https://www.youtube.com/watch?v=YouDOfVNHgk&ab_channel=Rudysmodelrailway

Demo EEP_Lua_Layout_04

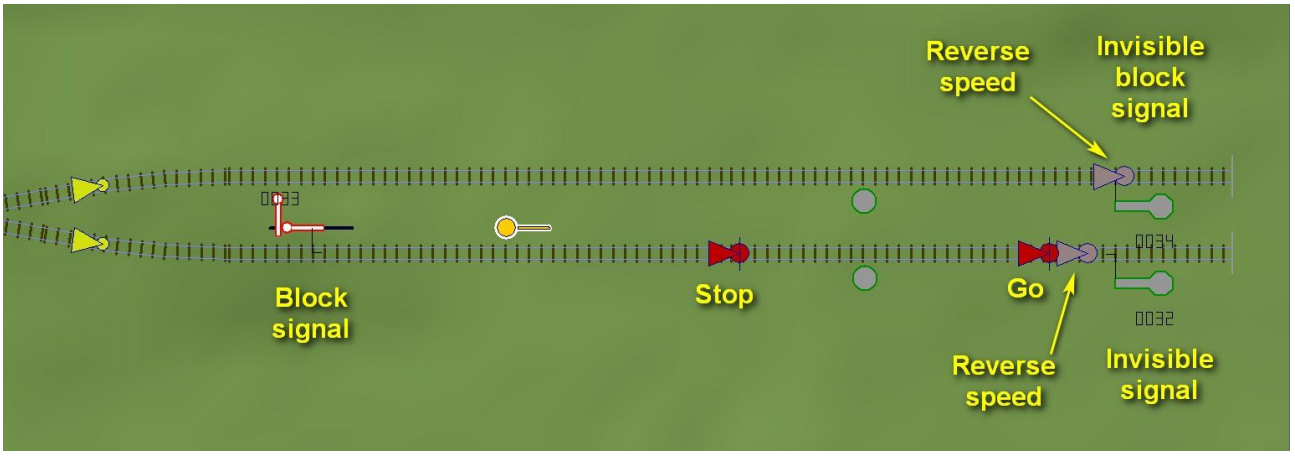
Two dead end tracks are added to the layout and we'll also add a third train.



For the Lua control system a dead end block works similar to a normal block. And even though traffic on a dead end track is both ways, it still is a single block having only one block signal.

[9] The order of the pairs as well as the order of the blocks within the pairs does not matter.

The image below shows two ways to build a dead end block in EEP that both work well with the Lua control system for automatic traffic.

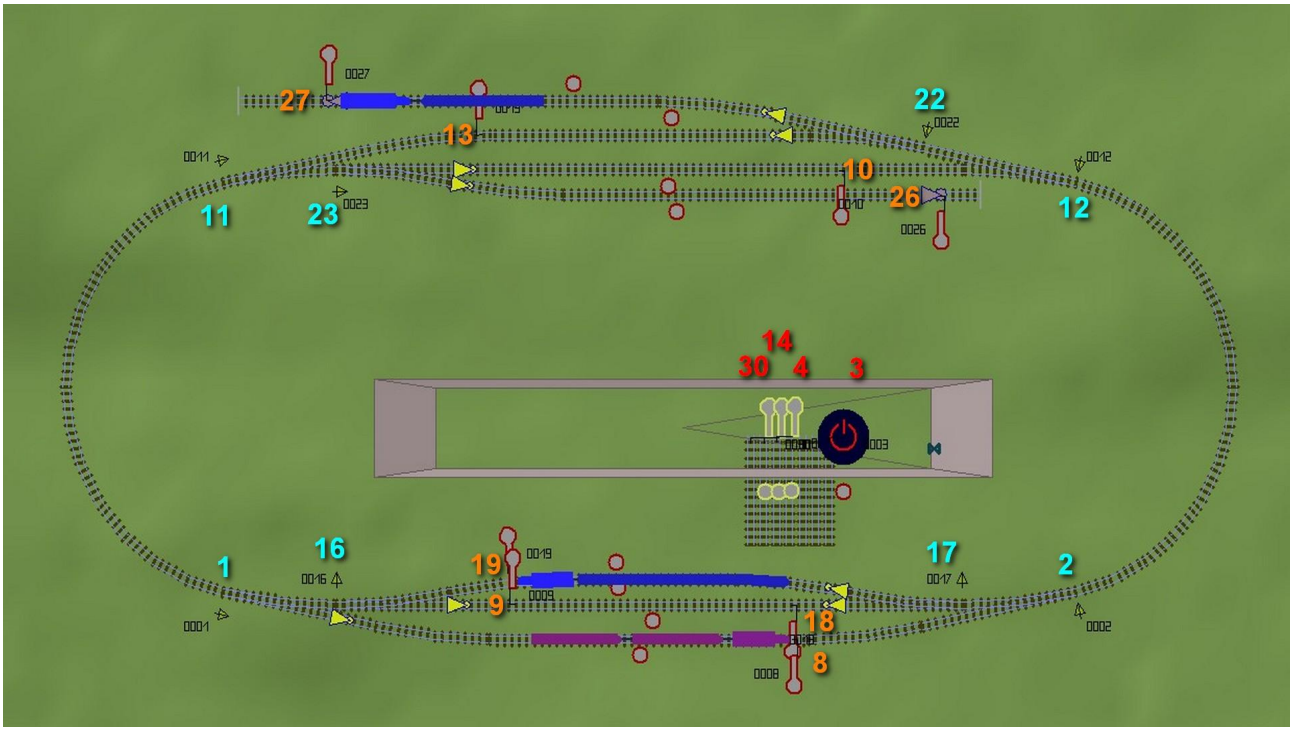


The top block has its (invisible) block signal at the end. The train will stop there. When Lua allows it to go again, it will start driving towards the dead end, but immediately behind the signal there's a "Vehicle" sensor that reverses its speed. It may require a little fine tuning to place this sensor as close to the signal as possible, such that the train speed is still so low that the speed reversal almost does not show. The plus of this method is that it is quick to build. The drawback is that there is no visible signal on this block that shows the train is allowed to leave the block, although there's a workaround for this: place a visible signal at the block exit and in the properties of the Lua block signal 'connect' to this visible signal. It'll now switch to green when the (invisible) block signal is switched to green by Lua, and similarly it'll switch back to red.

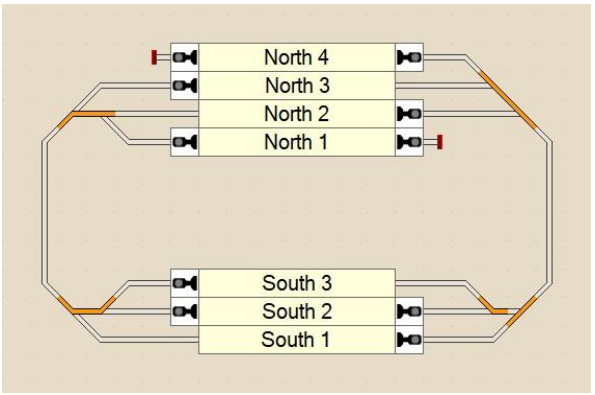
The bottom block has a visible block signal that is directly controlled by Lua. With this dead end method the train is stopped by an invisible signal at the end of the track, which is only there to help reverse the train, the blockControl module does not know about this signal. This signal is set to "red"/"halt" via a "Signal" sensor that must be placed in front of the pre-signal. Just before the train comes to a halt the signal is set to "green"/"go" by a second "Signal" sensor. The train is now reversed via a "Vehicle" sensor close to the signal which is to be hit at a very low speed. The exact placement of these sensors may require some fine tuning to make the speed reversal look good. The train now drives back to the block signal, where it stops.

You can choose whichever method you prefer, both work fine with the Lua automatic train control.

The image below shows the 2D view of `EEP_Lua_Layout_04`.



The configuration code for `EEP_Lua_layout_04` is:




```
local main_signal = 3

local block_signals = {
   8, -- South 1 -> East
  18, -- South 2 -> East two way
   9, -- South 2 -> West two way
  19, -- South 3 -> West
  26, -- North 1 dead end
  10, -- North 2 -> East
  13, -- North 3 -> West
  27, -- North 4 dead end
}

local everywhere = {
  [ 8] = 30, -- numbers > 1 are stop times
  [18] =  1,
  [ 9] =  1,
  [19] = 30,
  [26] = 30,
  [10] = 20,
  [13] = 20,
  [27] = 30,
}

local trains = {
  { name = "Orange", signal =  4, allowed = everywhere },
  { name = "Steam",  signal = 30, allowed = everywhere },
  { name = "Blue",   signal = 14, allowed = everywhere },
}

local two_way_blocks = { {18, 9} }
```

```
local routes = {
  {  8, 13, turn={  2,1, 12,1, 22,2 }},
  {  8, 27, turn={  2,1, 12,1, 22,1 }},
  { 18, 13, turn={ 17,1,  2,2, 12,1, 22,2 }},
  { 18, 27, turn={ 17,1,  2,2, 12,1, 22,1 }},
  {  9, 26, turn={ 16,1,  1,2, 11,2, 23,2 }},
  {  9, 10, turn={ 16,1,  1,2, 11,2, 23,1 }},
  { 19, 26, turn={ 16,2,  1,2, 11,2, 23,2 }},
  { 19, 10, turn={ 16,2,  1,2, 11,2, 23,1 }},
  { 26,  8, turn={ 23,2, 11,2,  1,1 }},
  { 26, 18, turn={ 23,2, 11,2,  1,2, 16,1 }},
  { 10,  9, turn={ 12,2,  2,2, 17,1 }},
  { 10, 19, turn={ 12,2,  2,2, 17,2 }},
  { 13,  8, turn={ 11,1,  1,1 }},
  { 13, 18, turn={ 11,1,  1,2, 16,1 }},
  { 27,  9, turn={ 22,1, 12,1,  2,2, 17,1 }},
  { 27, 19, turn={ 22,1, 12,1,  2,2, 17,2 }},
}
```

Because in the blockControl module the dead end blocks behave similar to normal blocks, there's nothing that needs to be done differently in the configuration code for the dead ends to work.

What is new here is the introduction of the `block_signals` table. This table simply lists all the block signals, in no particular order. The table is not mandatory, if it's omitted the Lua code will work perfectly fine. There can be two reasons to include the table:

- Having a list of all the blocks available, maybe with some comments on which is which, can help to not make mistakes while filling the `allowed` tables or the `routes` table.
- The Lua code has some consistency checks built in, one of which if all the listed blocks are used in the `routes` table at least once as departure and also at least once as destination, or if accidentally a non-existing block is used in a route. A warning is given if this check fails. See the chapter on Consistency Checks for more details.

In the example above all three trains use all blocks. Suppose we want something different, like:

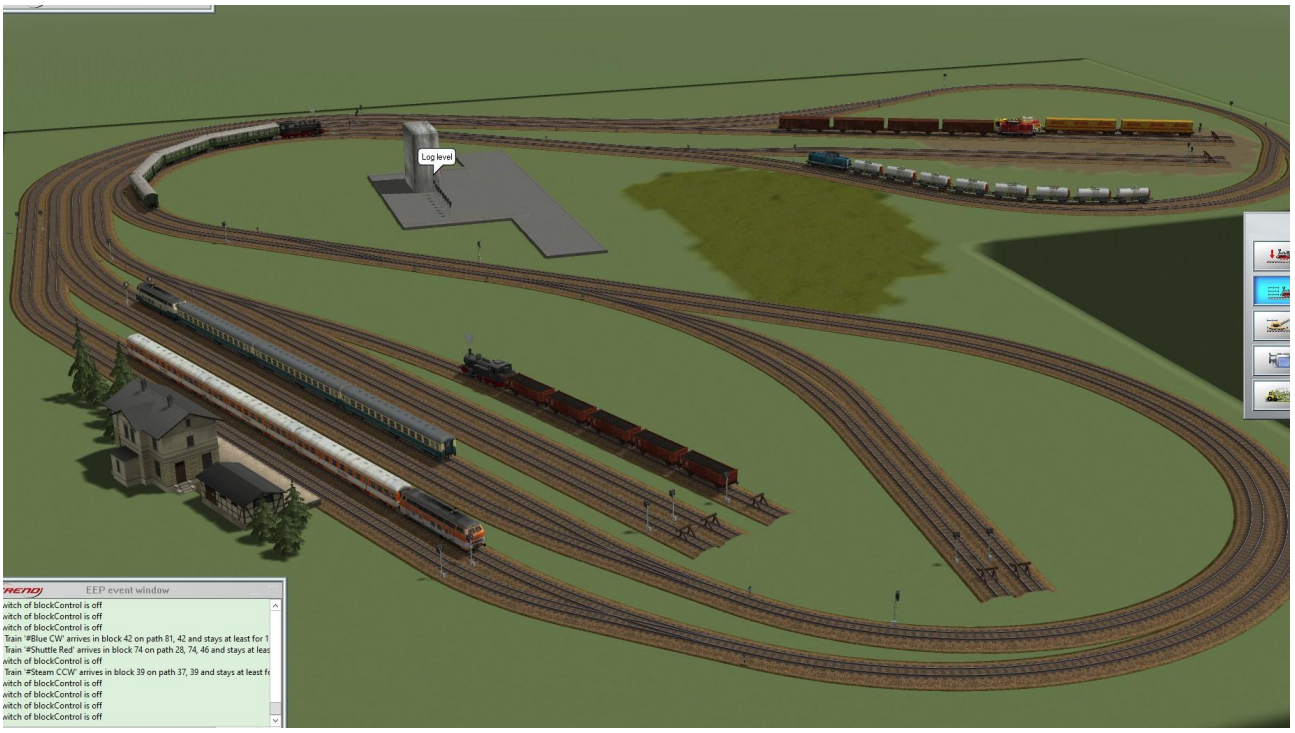
- The "Steam" train drives CCW on blocks 8 and 13, with a stop at 8.
- The "Orange" train drives CW on blocks 10 and 9 or 19. Only at 19 there's a stop, at 9 it behaves as an intercity and drives through the station.
- The "Blue" cargo train drives back and forth between the dead ends 26 & 27 and always drives via the two way blocks 9 & 18.

To get this working all you have to do is to ensure that the current direction of the trains match to the future settings - train "Orange" has to go CW, train "Steam" has to go CCW, train "Blue" must be on permitted blocks - and then change the allowed blocks and the trains tables:

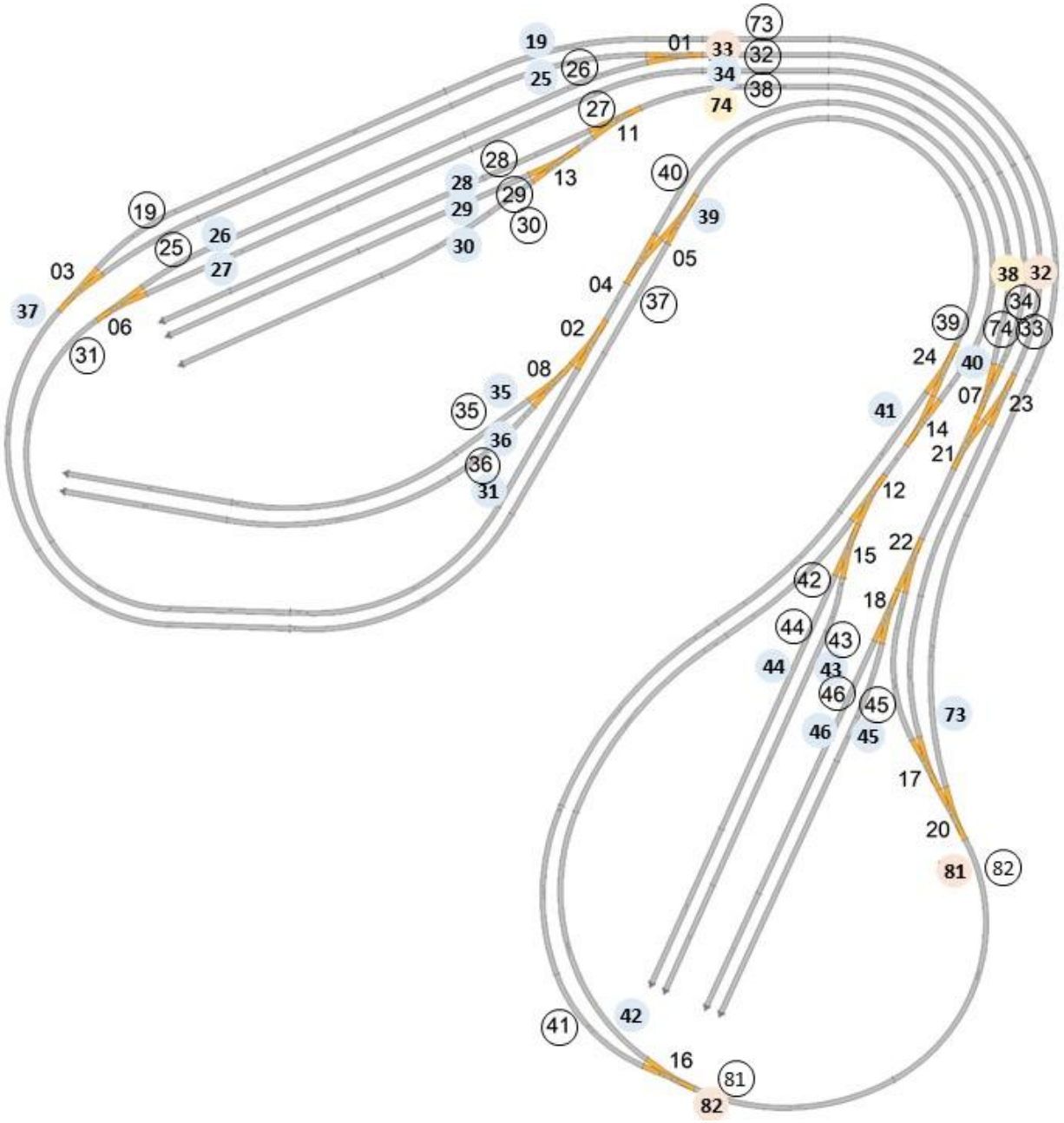
```
local CCW   =
{ [ 8]=30, [13]= 1, }          -- counterclockwise
local CW    = { [ 9]= 1, [10]= 1, [19]=30, } -- clockwise
local cargo = { [ 9]= 1, [18]= 1, [26]=30, [27]=30, }

local trains = {
  { name = "Orange", signal =  4, allowed = CW },
  { name = "Steam",  signal = 30, allowed = CCW },
  { name = "Blue",   signal = 14, allowed = cargo },
}
```

Demo EEP_Lua_Layout_05

It's time for something a bit more challenging … a layout with 27 blocks on which 7 trains drive fully automatic, Lua controlled.



To design this layout the model railway editor program called SCARM was used. The SCARM drawing below shows the block signal numbers (white circles) and their track contacts (colored circles) as well as the turnout numbers, as they were generated by EEP.



The configuration code for `EEP_Lua_layout_05` is:

```
local main_signal = 80

local block_signals  = { -- This table is optional, just for info
  19, 25, 26, 27,             -- Station North
  28, 29, 30,                 -- Cargo Station North
  35, 36,                     -- Cargo Station West
  43, 44,                     -- Cargo Station South-West
  45, 46,                     -- Cargo Station South-East
  37, 39, 41, 82, 73, 32, 38, -- Connecting tracks CCW
  33, 34, 74, 81, 42, 40, 31, -- Connecting tracks CW
}

local CCW = { -- allowed blocks and wait times CCW trains
  [19] = 45, [25] = 45,
  [37] = 1,
```

```
  [39] = 1,
  [41] = 1,
  [82] = 1,
  [73] = 1, [32] = 1,
}

local CW = { -- allowed blocks and wait times CW trains
  [26] = 40, [27] = 30,
  [33] = 1,  [34] =  1,
  [81] = 1,
  [42] = 1,
  [40] = 1,
  [31] = 1,
}

local shuttles = { -- allowed blocks and wait times cargo shuttles
  [28] = 28, [29] = 28, [30] = 28,
  [74] = 1,  [38] = 1,
  [45] = 28, [46] = 28,
  [32] = 1,
  [25] = 1,
  [37] = 1,
  [39] = 1,
  [43] = 28, [44] = 28,
  [40] = 1,
  [35] = 28, [36] = 28,
  [31] = 1,
  [26] = 1, [27] = 1,
  [33] = 1, [34] = 1,
}

local trains = {
  { name="Steam CCW",       signal =  9, allowed = CCW },
  { name="Orange CCW",      signal = 72, allowed = CCW },
  { name="Blue CW",         signal = 77, allowed = CW  },
  { name="Cream CW",        signal = 78, allowed = CW  },
  { name="Shuttle Red",     signal = 79, allowed = shuttles },
  { name="Shuttle Yellow", signal = 92, allowed = shuttles },
  { name="Shuttle Steam",  signal = 93, allowed = shuttles },
}

local two_way_blocks = { {82, 81}, {32, 33}, {74, 38}, }

local f = 1 -- turnout position "fahrt / main"
local a = 2 -- turnout position "abzweig / branch"
local routes = {
  { 19, 37, turn={  3,f }},
  { 25, 37, turn={  3,a }},
  { 26, 33, turn={  1,a }},
  { 27, 34, turn={}},
  { 28, 74, turn={ 11,f }},
  { 29, 74, turn={ 13,f, 11,a }},
  { 30, 74, turn={ 13,a, 11,a }},
  { 31, 26, turn={  6,f }},
  { 31, 27, turn={  6,a }},
  { 32, 25, turn={  1,f }},
  { 33, 45, turn={ 23,a, 21,a, 22,f, 18,a }},
  { 33, 46, turn={ 23,a, 21,a, 22,f, 18,f }},
  { 33, 81, turn={ 23,f, 17,a, 20,f }},
```

```
  { 34, 45, turn={  7,f, 21,f, 22,f, 18,a }},
  { 34, 46, turn={  7,f, 21,f, 22,f, 18,f }},
  { 34, 81, turn={  7,f, 21,f, 22,a, 17,f, 20,f }},
  { 35, 39, turn={  8,a,  2,a,  4,a,  5,f }},
  { 36, 39, turn={  8,f,  2,a,  4,a,  5,f }},
  { 37, 39, turn={  5,a }},
  { 38, 28, turn={ 11,f }},
  { 38, 29, turn={ 11,a, 13,a }},
  { 38, 30, turn={ 11,a, 13,f }},
  { 39, 41, turn={ 24,a }},
  { 39, 44, turn={ 24,f, 14,a, 12,a, 15,f }},
  { 39, 43, turn={ 24,f, 14,a, 12,a, 15,a }},
  { 40, 31, turn={  4,f,  2,f }},
  { 40, 35, turn={  4,f,  2,a,  8,a }},
  { 40, 36, turn={  4,f,  2,a,  8,f }},
  { 41, 82, turn={ 16,f }},
  { 42, 40, turn={ 12,f, 14,f }},
  { 43, 40, turn={ 15,a, 12,a, 14,f }},
  { 44, 40, turn={ 15,f, 12,a, 14,f }},
  { 45, 32, turn={ 18,a, 22,f, 21,a, 23,a }},
  { 45, 38, turn={ 18,a, 22,f, 21,f,  7,a }},
  { 46, 32, turn={ 18,f, 22,f, 21,a, 23,a }},
  { 46, 38, turn={ 18,f, 22,f, 21,f,  7,a }},
  { 73, 19, turn={}},
  { 74, 45, turn={  7,a, 21,f, 22,f, 18,a }},
  { 74, 46, turn={  7,a, 21,f, 22,f, 18,f }},
  { 74, 81, turn={  7,a, 21,f, 22,a, 17,f, 20,f }},
  { 81, 42, turn={ 16,a }},
  { 82, 32, turn={ 20,f, 17,a, 23,f }},
  { 82, 73, turn={ 20,a }},
}
```

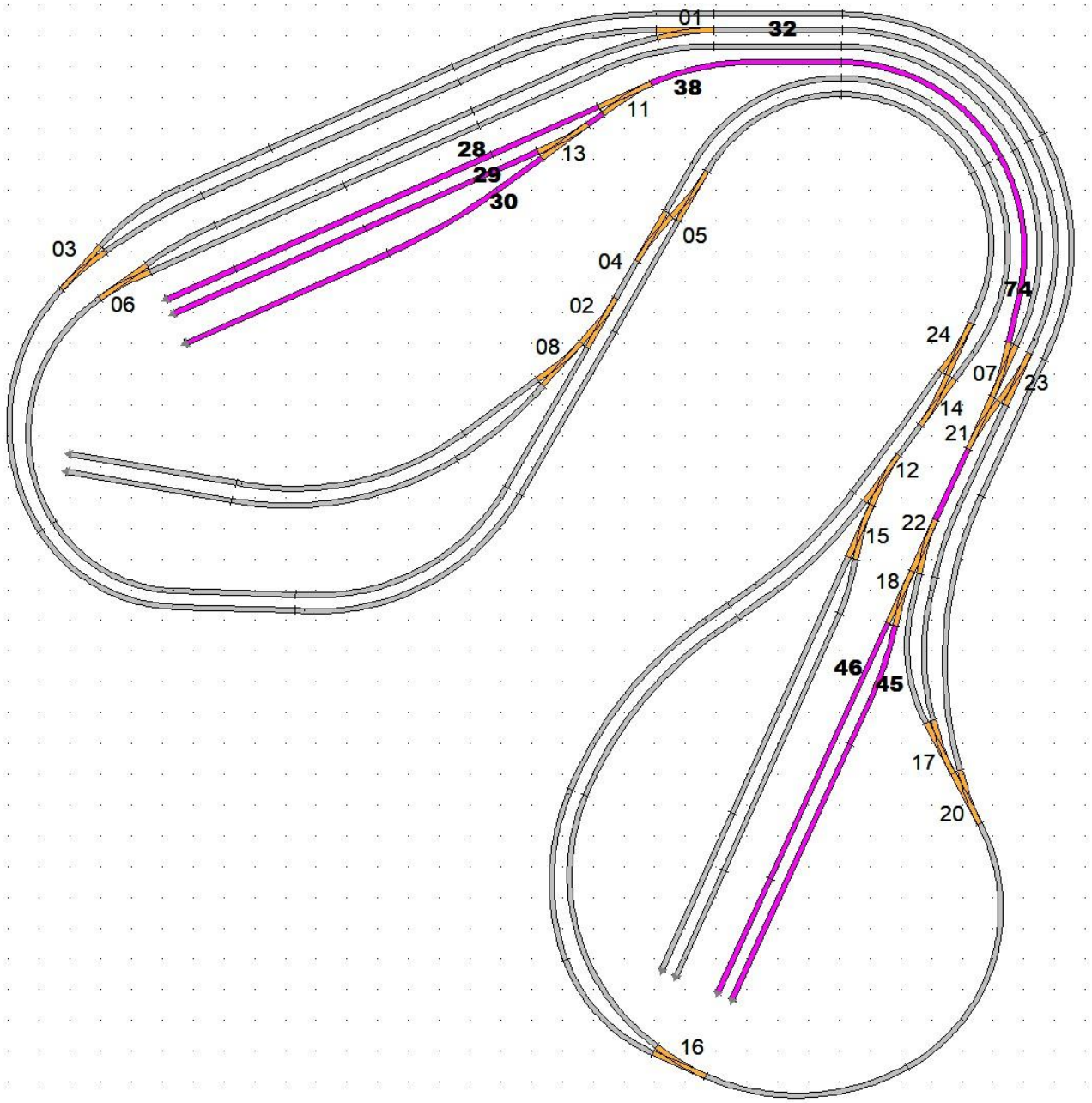
The only new principle introduced here is the use of the variables `f=1` and `a=2`. They make the “fahrt/main” and “abzweig/branch” states of the turnouts easier to read.

Yes, it’s some more work to define a layout like this, but it is rather straightforward. The challenge is to stay focused and not make a single mistake in any of the tables. There is some help … the Lua code has several consistency checks built in that will generate warnings if not all blocks are used in routes, or if accidentally a number is used that is not an existing signal- or turnout number. See the chapter on Consistency Checks for more info.

How to avoid deadlocks

With some layouts a situation can occur where trains are driving in opposing directions and none of them can find a free track to go to. The trains will come to a halt, a so-called “deadlock” has now occurred.

The picture below shows a section of `layout_05` in pink where a deadlock can occur if we’d have 3 shuttle trains on this section and we would not allow the trains to ‘escape’ via block 32. Suppose both blocks 45 and 46 are occupied and the third train, that is on one of the blocks 28, 29, 30, starts to drive to block 74. Once it arrives in block 74 none of the 3 trains can find a new route anymore because there are no free blocks ahead … we have a deadlock.



How to avoid deadlocks?

- The obvious solution would be to design a layout where deadlocks cannot occur in the first place. In `layout_05` this is easily done by allowing trains to also drive to block 32 and thus free up one of the blocks 45 or 46.
- Another solution could be to not define 38 & 74 as blocks in Lua, essentially making this track part of the turnouts. Lua checks if an adjacent block is free. If 74 is not a block anymore Lua checks 45 and 46. If both are occupied, a train at 28,29,30 will not start to drive, first a train from 45,46 will go the other way. To still have visible train signals, other signals that are controlled via track contacts can be used, which are not under control of the blockControl module.
- The third solution is to tell Lua about the potential deadlock path and to look further than one block ahead. Lua should not allow a train on 28,29,30 to start driving if there is no free block in 45,46. We tell Lua to do this via the `anti_deadlock_paths` table.

```
local anti_deadlock_paths = {
  { {28,29,30}, 74, {46,45} },
```

```
    }
```

The path in the other direction, `{ {46,45}, 38, {28,29,30} },` can also be specified, but in this case it's not required because there are 3 blocks available for the 3 trains, a deadlock in that direction can simply not occur.

In case you have a very long in between track that you like to divide into multiple blocks, that's perfectly fine, the anti deadlock path then could look like, say:
`{ {28,29,30}, 74, 75, 76, {46,45} },`

How to prevent collisions on crossings

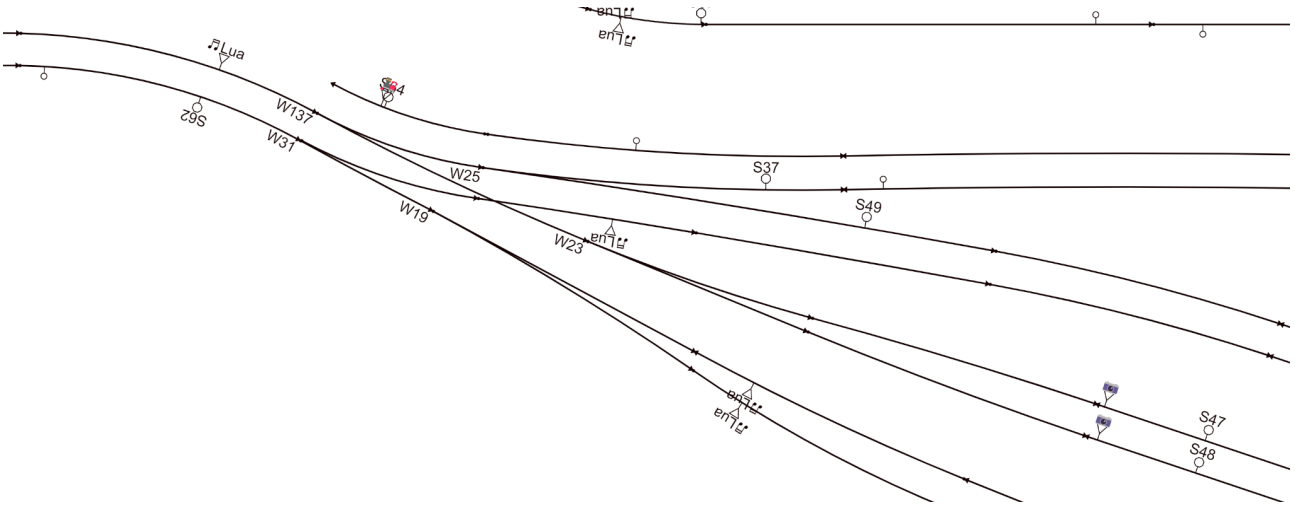
Because crossings are not turnouts they are not declared in the routes table which means they are not reserved when Lua selects the route, as blocks and turnouts are. As a consequence, without precautions two trains can drive on a crossing at the same time. This doesn't give problems, the automatic traffic keeps functioning, it just doesn't look good.

Collisions on a crossing can be prevented in two ways:

1. Find the two blocks on the tracks immediately behind the crossing and declare them as `two_way_blocks` (see demo layout_03 on how to declare two way blocks). The result is that as soon as one of these blocks is reserved for a route, the other block will be reserved too. No train will go there until it is released, which keeps the crossing free of other traffic.
2. If route B that uses the crossing contains at least one turnout, then include this turnout in route A's declaration. It's of not importance if you switch the turnout to 1 or 2, the only reason to include it is to make sure this turnout gets reserved as soon as route A is activated. Do the same vice versa, include a turnout of route a in the declaration of route B. The result is that no train can start route B as long as route A has reserved a turnout that belongs to route B, the crossing will be safe to use by the train on route A.

Here is an example about solving the crossing issue using the second option. First let's have a look to the penetration of trains on a crossing:

Here is the corresponding part of the layout:



The path from block 62 on the left via turnout 31 to block 77 (not visible on the right) and the routes on the right from blocks 47 and 48 via turnout 23 and 137 to block 50 (not visible on the left) both share the same crossing.

This crossing is not part of any block - it's between blocks - and this allow an easy solution:

1. Generate the routes

2. Extend the routes manually by adding a turnout from the other route to lock this additional turnout. The position for this turnout does not matter, therefore you can use the dummy value 0.

This leads to following modified routes:

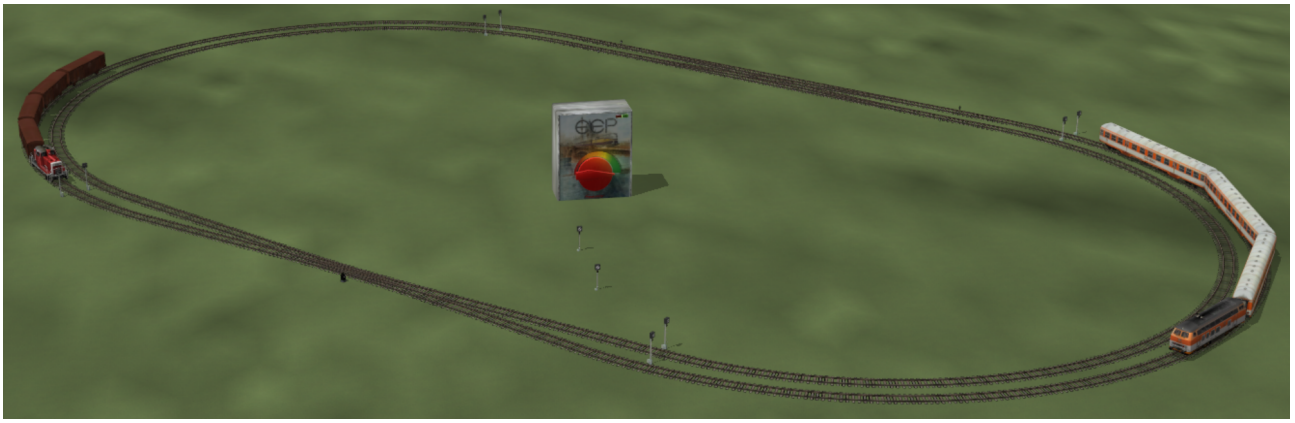
```
-- The generated routes are extended to avoid crashes at the crossing.
-- The extended routes include a turnout from the other route.
-- The turnout setting does not matter, therefore use dummy value 0.

local routes = {
  ...
--{ 47, 50, turn={ 23,2, 137,2, }},
  { 47, 50, turn={ 23,2, 137,2, 31,0, }}, -- Crossing
--{ 48, 50, turn={ 23,1, 137,2, }},
  { 48, 50, turn={ 23,1, 137,2, 31,0, }}, -- Crossing
  ...
--{ 62, 77, turn={ 31,1, }},
  { 62, 77, turn={ 31,1, 23,0, }}, -- Crossing
  ...
}
```

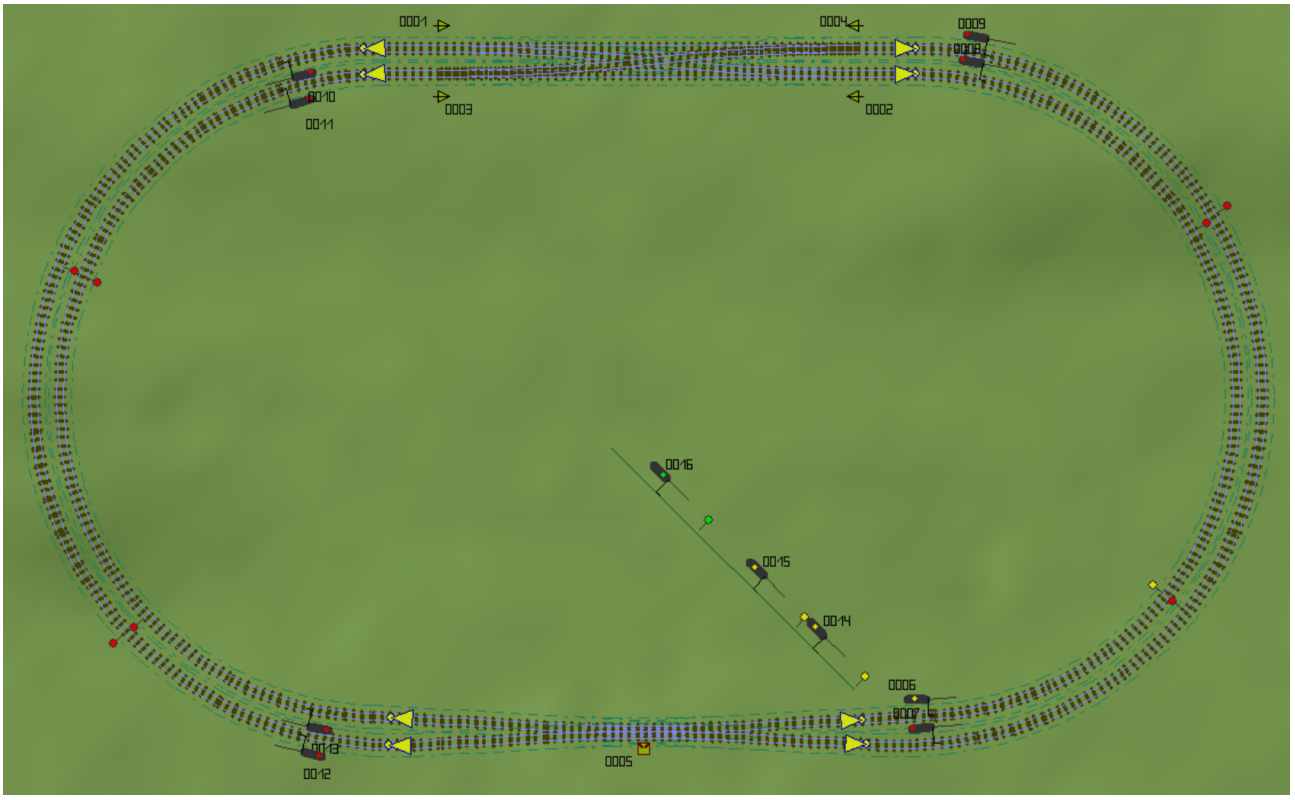
Demo Double_slip_turnouts

Double slip turnouts (DST) exist in two flavor which require some special treatments:

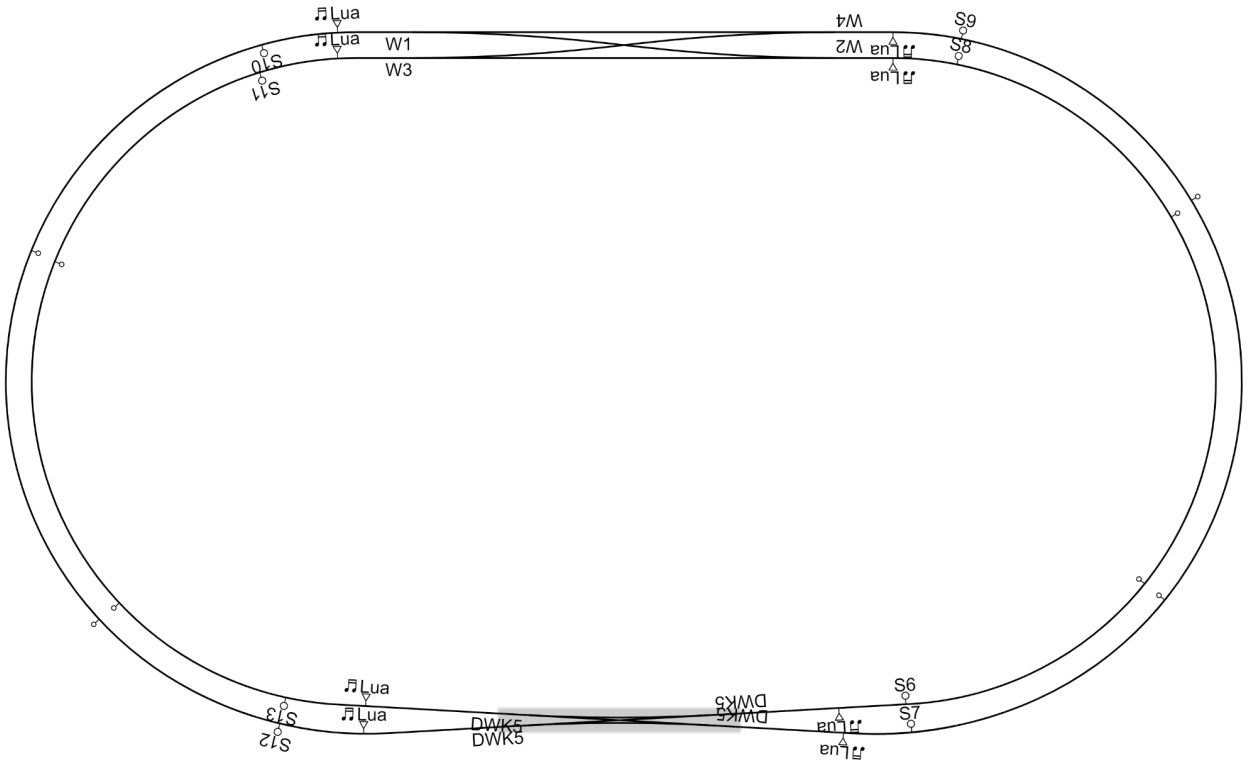
a) You can create a DST constinsing 4 turnouts manually or using the templates in EEP. You should secure the crossing of that DST by extending the direct routes with a turnout from the other part of the DST.
b) You can use a track object DST consisting of 1 turnout having 4 positions. You can define the routes as usual.

This demo layout shows both flavors, a 4-turnout-DST at the top and a track object DST at the bottom:





Here is the corresponding picture from the Gleisplan program:



The configuration code for `Double_slip_turnouts` is:

```
-- Allowed blocks with wait time
```

```
local passenger = { [6]=1,  [7]=1,  [8]=1,  [9]=1,  [10]=1,  [11]=1,  [12]=1,
[13]=1,  }
local cargo     = { [6]=20, [7]=30, [8]=20, [9]=20, [10]=20, [11]=30, [12]=30,
[13]=30, }

local trains = {
{ name="#Orange",      signal=14, allowed=passenger },
{ name="#Shuttle Red", signal=15, allowed=cargo     },
}

local main_signal = 16

local block_signals = { 6, 7, 8, 9, 10, 11, 12, 13, }

local two_way_blocks = { { 6, 8 }, { 7, 9 }, { 10, 12 }, { 11, 13 }, }

local routes = {
-- CCW via DST using 4 turnouts (manually adjusted to secure the crossing)
 { 8, 12, turn={ 2,2, 1,2,     3,0 }},    -- crossing
 { 8, 13, turn={ 2,1, 3,1, }},            -- straight
 { 9, 12, turn={ 4,1, 1,1, }},            -- straight
 { 9, 13, turn={ 4,2, 3,2,     1,0 }},    -- crossing

-- CCW via track object DST              (manually created)
 { 13, 8, turn={ 5,1 }},                  -- left/left
 { 13, 9, turn={ 5,2 }},                  -- left/right
 { 12, 9, turn={ 5,3 }},                  -- right/right
 { 12, 8, turn={ 5,4 }},                  -- right/left

-- CW via DST using 4 turnouts  (manually adjusted to secure the crossing)
 { 10, 6, turn={ 1,2, 2,2,     3,0 }},    -- crossing
 { 10, 7, turn={ 1,1, 4,1, }},            -- straight
 { 11, 6, turn={ 3,1, 2,1, }},            -- straight
 { 11, 7, turn={ 3,2, 4,2,     2,0 }},    -- crossing

-- CW via track object DST               (manually created)
 { 6, 11, turn={ 5,1 }},                  -- left/left
 { 7, 11, turn={ 5,2 }},                  -- left/right
 { 7, 10, turn={ 5,3 }},                  -- right/right
 { 6, 10, turn={ 5,4 }},                  -- right/left
}
```

All trains have different velocities and can go everywhere with different wait times. This way different encounters occur showing passing and waiting situations..

The direct routes of the 4-turnout-DST are extended to secure the crossing by adding a turnout from the other part of the crossing. The constant distance between the parallel tracks of that DST allows to omit this for the straight parts.

The routes of the track object DST show the 4 different positions.

Tipp: Activate the checkbox about turnout events in the Lua script editor to view the position numbers of this turnout depending on the positions. This way it is quite easy to define the routes for this DST.

Options

Shown below is the code that follows the configuration part. This part of the code is the same for every layout and can often be left untouched, unless signals are used on the layout that have different red / green states, or if you like to change options.

```
clearlog()
```
This clears the EEP Event Screen after startup and after a reload of the Lua code. You can delete or comment this line if you don't want to clear the screen.

```
local blockControl = require("blockControl")
```
This loads the file `blockControl.lua` which is the actual code that generates automatic train traffic. It came with the download zip and it has to be placed in your EEP installation `\LUA` folder.

```
blockControl.init({
  trains       = trains,
  blockSignals = block_signals,
  twoWayBlocks = two_way_blocks,
  routes       = routes,
  paths        = anti_deadlock_paths,
  MAINSW       = main_signal,

  MAINON       = 1, -- ON    state of main switch
  MAINOFF      = 2, -- OFF   state of main switch
  BLKSIGRED    = 1, -- RED   state of block signals
  BLKSIGGRN    = 2, -- GREEN state of block signals
  TRAINSIGRED  = 1, -- RED   state of train signals
  TRAINSIGGRN  = 2, -- GREEN state of train signals
})
```

The left column are the parameters as used in the blockControl code, they may not be changed. The right column are the variables as they are used in your layout configuration code, you are free to use any names you like for them.

See the chapter on Signal States to find out about the states of the signals you have in use on the layout. You can change the signal states here if necessary.

In addition you can set some runtime parameters at any time (before or after the `init` call or at some times within `EEPMain`):

```
-- Optional: set these parameters to you personal liking
blockControl.set({
  logLevel        = 1,     -- 0:off, 1:normal, 2:more, 3:extreme
  showTippText    = true,  -- Show signal popups: true / false
  start           = false, -- Activate the main signal: true / false
  startAllTrains  = true,  -- Activate all train signals: true / false
})
```
You can set these self-explanatory parameters to your personal preference. You can define parameter `logLevel` in the `init` call as well..

```
-- Optional: Activate a control desk for the EEP layout
-- This only works as of EEP 16.1 patch 1
if EEPActivateCtrlDesk then
```

```
  local ok = EEPActivateCtrlDesk("Block control")
  if ok then print("Show control desk 'Block control'") end
end
```

As of EEP version 16.1 patch 1 `EEPActivateCtrlDesk` is a built-in function. All 5 demo layouts have a Control Panel on which train traffic can be controlled and monitored. This code opens the Control Panel by default. If you don't want that, then simply delete or comment this code. The Control Panel can also be opened via shift-click on the small console that resides next to the main switch.

```
-- Optional: Use a counter signal to set the log level
local counterSignal = 47
blockControl.set({ logLevel = EEPGetSignal( counterSignal ) - 1 })
EEPRegisterSignal( counterSignal )
_ENV["EEPOnSignal_"..counterSignal] = function(pos)
  local logLevel = pos - 1
  if logLevel > 3 then
    logLevel = 0
    blockControl.set({ logLevel = logLevel })
    EEPSetSignal( counterSignal, logLevel + 1 )
  else
    blockControl.set({ logLevel = logLevel })
  end
  print("Log level set to ", logLevel)
end
```

A counter signal can be placed on the layout via which the logLevel can be manually changed on the fly, without opening the Lua Editor Window. An example is shown on `EEP_Lua_Layout_05`.

```
function EEPMain()
  blockControl.run()
  return 1
end
```

The main EEP loop. The only line of code inside the main loop, which by EEP convention runs 5 times per second, is to run the blockControl code.

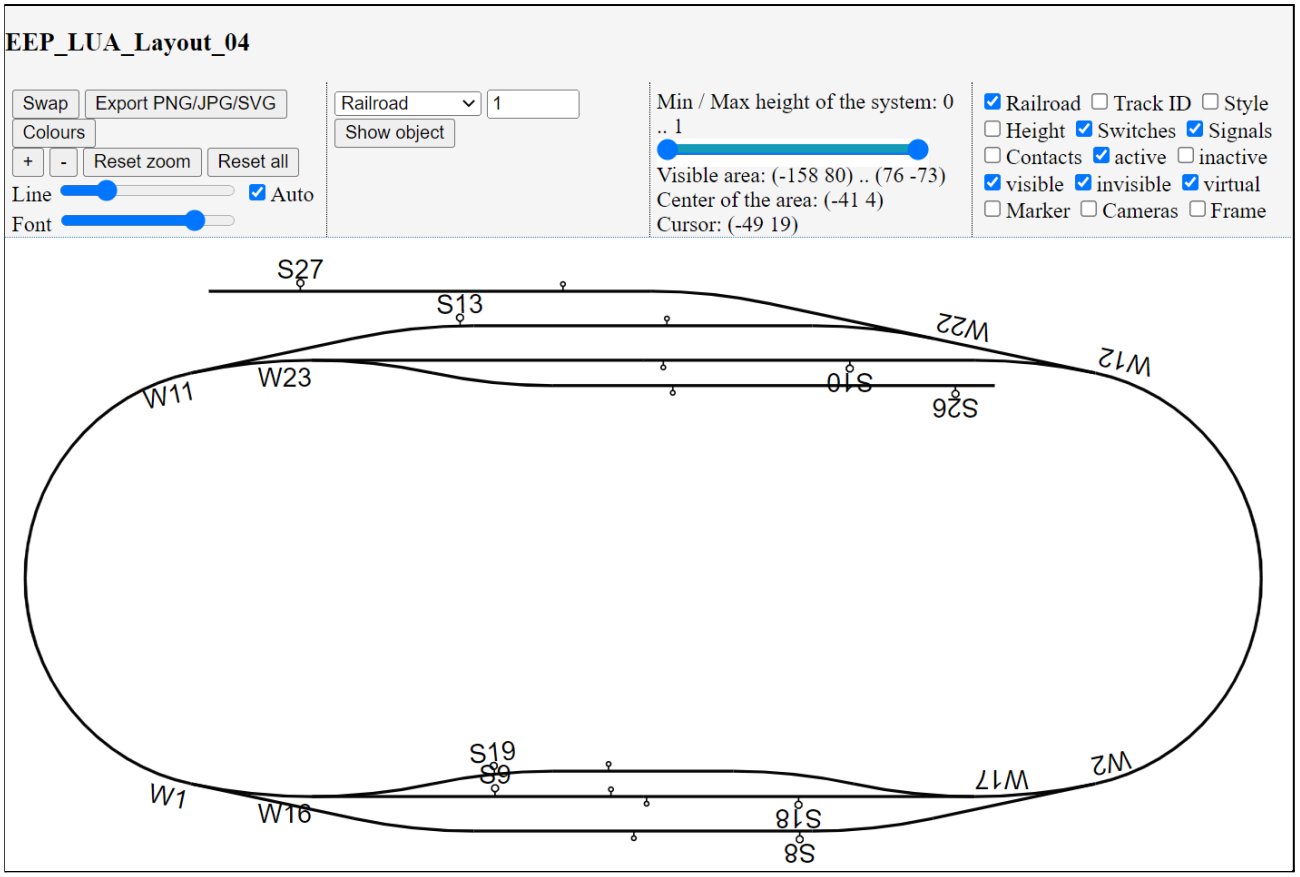
Tools to generate the Lua configuration tables

The `routes` table in the Lua configuration section describes the available routes from one block to the next by defining the 'main' / 'branch' states of the turnouts between the two blocks. This table can be created by hand but this requires great focus … one small mistake with a turnout state can cause a train to drive to an unexpected block, resulting in erratic automated traffic.

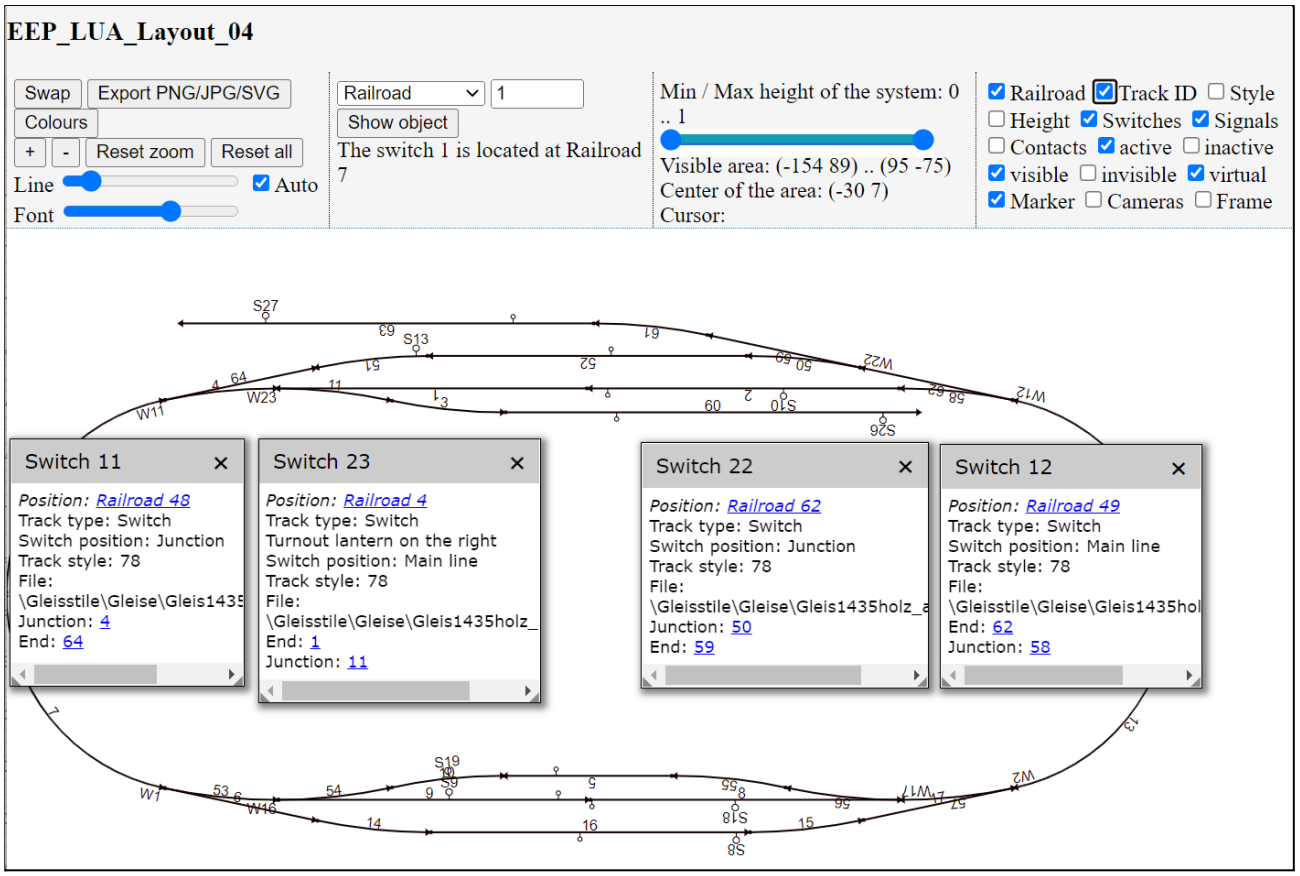
Two tools come to the rescue.

As a first step you can use the **Gleisplan program**[10]. It shows the EEP layout in your browser window, including signals and turnout numbers, in an easily readable format.

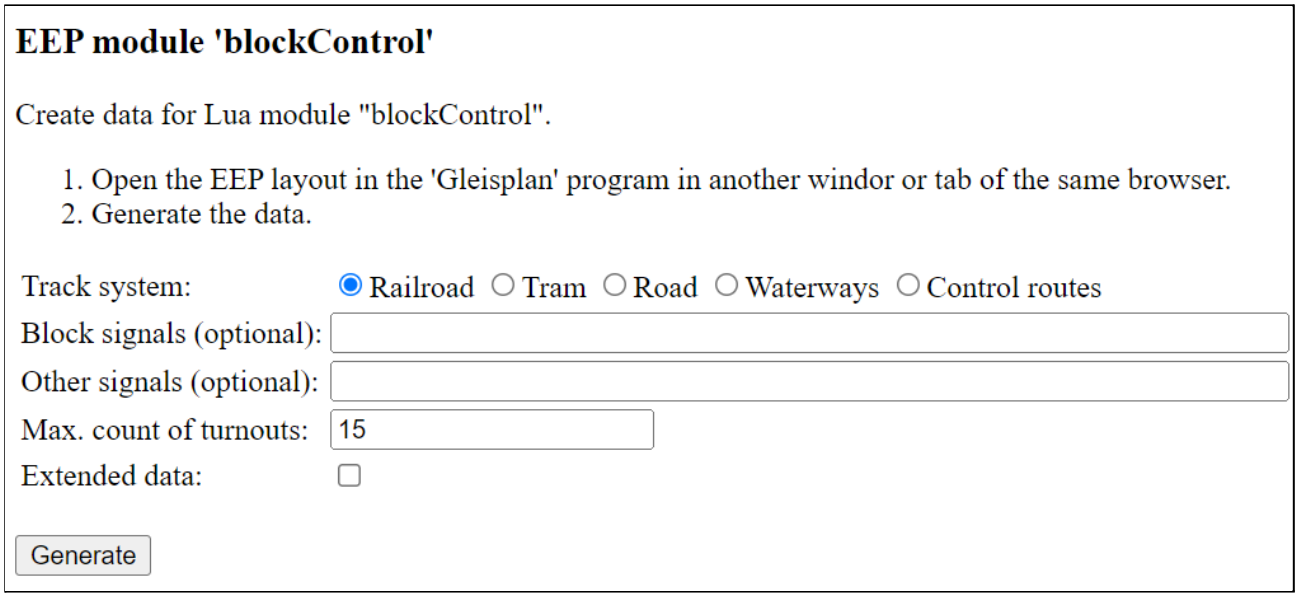
[10] https://frankbuchholz.github.io/EEP_convert_anl3_file/EEP_Gleisplan.html



With EEP sometimes a turnout's 'main' and 'branch' states can be confusing. What visibly looks like going 'main' EEP could label 'branch'. What can help is to first switch all the turnouts on the EEP layout such that the train drives to what you think should be called 'main' and then open the file in the **Gleisplan program**. If you left click on a turnout, an info popup opens that, among other info, shows its state. Shift + left click opens multiple of these pop ups.



A second tool goes a big step further, it can generate a proposal for all the Lua configuration tables, except the `paths` table. First open the layout in the **Gleisplan program**. Now open this link **blockControl program**[11] in another tab in the same browser and click the 'Generate' button:



Some layouts contain additional signals which should not get controlled by the module. In such a case you can either enter all block signal numbers which should get controlled, or you can enter

[11] https://frankbuchholz.github.io/EEP_convert_anl3_file/EEP_blockControl.html

the block signal numbers which should not get controlled into the corresponding field. You can use any kind of separator between the numbers.

The option to set the maximum count of turnouts per route is not very important. It mainly prohibits endless loops in case of incomplete layouts.

You can activate the option “extended data” which adds information about signal positions (if known by the program) and visited tracks to the result.

Here is a typical result:

EEP module 'blockControl'

Create data for Lua module "blockControl".

1. Open the EEP layout in the 'Gleisplan' program in another windor or tab of the same browser.
2. Generate the data.

Track system: (•) Railroad ( ) Tram ( ) Road ( ) Waterways ( ) Control routes
Block signals (optional):
Other signals (optional):
Max. count of turnouts: 15
Track data of blocks: ☐
Extended data: ☐

Generate

```
-- EEP File 'EEP_LUA_Layout_04'
-- Lua program for module 'blockControl.lua' for track system 'Railroad'

local main_signal = 3

local block_signals = { 8, 9, 10, 13, 18, 19, 26, 27, }

local two_way_blocks = { { 9, 18 }, }

-- Allowed blocks with wait time
local all = { [8]=1, [9]=1, [10]=1, [13]=1, [18]=1, [19]=1, [26]=1, [27]=1, }

local trains = {
 { name="#Blue",   signal=0, allowed=all },
 { name="#Orange", signal=0, allowed=all },
 { name="#Steam",  signal=0, allowed=all },
}

local routes = {
  { 8, 13, turn={ 2,1, 12,1, 22,2, }},
  { 8, 27, turn={ 2,1, 12,1, 22,1, }},
  { 9, 10, turn={ 16,1, 1,2, 11,2, 23,1, }},
  { 9, 26, turn={ 16,1, 1,2, 11,2, 23,2, }},
  { 10, 9, turn={ 12,2, 2,2, 17,1, }},
  { 10, 19, turn={ 12,2, 2,2, 17,2, }},
  { 13, 8, turn={ 11,1, 1,1, }},
  { 13, 18, turn={ 11,1, 1,2, 16,1, }},
  { 18, 13, turn={ 17,1, 2,2, 12,1, 22,2, }},
  { 18, 27, turn={ 17,1, 2,2, 12,1, 22,1, }},
  { 19, 10, turn={ 16,2, 1,2, 11,2, 23,1, }},
  { 19, 26, turn={ 16,2, 1,2, 11,2, 23,2, }},
  { 26, 8, turn={ 23,2, 11,2, 1,1, }},
  { 26, 18, turn={ 23,2, 11,2, 1,2, 16,1, }},
  { 27, 9, turn={ 22,1, 12,1, 2,2, 17,1, }},
  { 27, 19, turn={ 22,1, 12,1, 2,2, 17,2, }},
}
```

You can click on signal numbers and turnout numbers to position the view in the “Gleisplan” program onto the selected object.

Now you can copy & paste the result into your Lua file and adjust it according to your requirements.

The variable `start_signal` shows the number of the main signal ("Switch_standing", "Switch_lying", or "StartSwitch_usertex") with the lowest number if there exists one on the layout. Otherwise you get number 0 and you can change this by yourself.

The tables `block_signals` and `two_way_blocks` contain the full list of blocks.

```
local block_signals = { 8,  9,  10,  13,  18,  19,  26,  27, }

local two_way_blocks = { { 9, 18 }, }
```

Then you will get one or more tables with a full list of blocks with default wait time 1 for every EEP route which is assigned to some trains. If no EEP routes are used, you get one table called `all`. You can use these proposals to adjust the wait time and to remove blocks which should not be allowed for specific trains.

```
-- Allowed blocks with wait time
local all = { [8]=1, [9]=1, [10]=1, [13]=1, [18]=1, [19]=1, [26]=1,
[27]=1, }
```

Table `trains` shows all trains on the layout and in depots with assigned `allowed` tables but without a specific train signal number.

```
local trains = {
 { name="#Blue",   signal=0, allowed=all },
 { name="#Orange", signal=0, allowed=all },
 { name="#Steam",  signal=0, allowed=all },
}
```

Finally, you get table `routes` with correct turnout positions. Sometimes you will be surprised about the findings, especially if there exist multiple routes between blocks. You can either remove (or comment) such superfluous routes or you can add more blocks to the layout.
On the other hand you might miss some routes, especially if you work with two way blocks. Keep in mind that adjacent tracks between turnouts either belong to no block (no signal), to a one way block (one signal), or to a two way block (two signals for opposite directions).

```
local routes = {
  { 8, 13, turn={ 2,1, 12,1, 22,2, }},
  { 8, 27, turn={ 2,1, 12,1, 22,1, }},
  { 9, 10, turn={ 16,1, 1,2, 11,2, 23,1, }},
  { 9, 26, turn={ 16,1, 1,2, 11,2, 23,2, }},
  { 10, 9, turn={ 12,2, 2,2, 17,1, }},
  { 10, 19, turn={ 12,2, 2,2, 17,2, }},
  { 13, 8, turn={ 11,1, 1,1, }},
  { 13, 18, turn={ 11,1, 1,2, 16,1, }},
  { 18, 13, turn={ 17,1, 2,2, 12,1, 22,2, }},
  { 18, 27, turn={ 17,1, 2,2, 12,1, 22,1, }},
  { 19, 10, turn={ 16,2, 1,2, 11,2, 23,1, }},
  { 19, 26, turn={ 16,2, 1,2, 11,2, 23,2, }},
  { 26, 8, turn={ 23,2, 11,2, 1,1, }},
  { 26, 18, turn={ 23,2, 11,2, 1,2, 16,1, }},
```

```
  { 27, 9, turn={ 22,1, 12,1, 2,2, 17,1, }},
  { 27, 19, turn={ 22,1, 12,1, 2,2, 17,2, }},
}
```

Limitations:

- At the moment the module cannot process alternative routes for the same pair of start and end blocks, which would lock all the turnouts of all alternative routes. Therefore only one of the given routes is used, which contains the least number of switches..
- You do not get table `paths` to resolve potential lockdown situations. If you run into lockdowns you have to develop your own solution using the tips shown above.
- Double slip turnouts consisting of 4 single turnouts can be used easily because the module simply sees these 4 turnouts. Do not forget to secure the crossing by adding a turnout from the other part of the crossing to the (generated) routes. (If the distance of the tracks is sufficient then you can omit that for the straight parts of the crossing).
- Track object double slip turnouts have only 1 turnout with 4 positions.The generation program does not know anything about such turnouts yet. Therefore you have to create the required routes manually, which works fine, too.
- Garbage in - garbage out: If the layout does not have proper block signals you cannot expect sound results.

Finally you can us the **Inventar program**[12] to review the settings of the contacts including the Lua function in contacts:

You may want to hide some columns to optimize the view or you may want to filter by entering `[nonempty]` (including the brackets) into the filter field for column “Lua function”:

Contacts: 10 Hide columns ▼ ? ✖

Filter ...						
Type of contact	**Type of contact**	**Track**	**Trigger**	**Train position**	**Lua function**	**Tip's text**
5	Vehicle -0 km/h max	Railroad 60	in the direction of the track	Front		Block 26 reverse direction
5	Vehicle -0 km/h max	Railroad 63	in the direction of the track	Front		Block 27 reverse direction
256	Sound undefined	Railroad 1	against track direction	End	blockControl.enterBlock_10	Block 10
256	Sound undefined	Railroad 3	in the direction of the track	End	blockControl.enterBlock_26	Block 26
256	Sound undefined	Railroad 6	in the direction of the track	End	blockControl.enterBlock_8	Block 8
256	Sound undefined	Railroad 8	against track direction	End	blockControl.enterBlock_9	Block 9
256	Sound undefined	Railroad 9	in the direction of the track	End	blockControl.enterBlock_18	Block 18
256	Sound undefined	Railroad 52	in the direction of the track	End	blockControl.enterBlock_13	Block 13
256	Sound undefined	Railroad 55	in the direction of the track	End	blockControl.enterBlock_19	Block 19
256	Sound undefined	Railroad 61	in the direction of the track	End	blockControl.enterBlock_27	Block 27

[12] https://frankbuchholz.github.io/EEP_convert_anl3_file/EEP_Inventar.html

How to stop driving and save the current state

Before exiting EEP it is good practice to switch off the main switch and wait until all trains have come to a stop at a signal. However, thanks to the "*find mode for trains*", after loading a layout file or after reloading the Lua code, the layout can be saved at any time, in the next session Lua will re-find the trains. There's no need to save the layout via the menu, the current state is saved automatically when EEP is exited.

How to use the module "BetterContacts" to simplify configuration

The EEP editor for contacts has the restriction that you only can enter existing Lua functions. Therefore you have to work in a specific order:

1. Place block signals and optionally the contacts (but without referring to a Lua function in these contacts - the field has to be empty)
2. Adjust the Lua code to list all block signals in some of the tables
3. Run the layout in 3D mode
4. Place the contacts if not already done
5. Enter the reference to the Lua function for the block signals into the contacts

Whenever you add more block signals you have to follow this order again for these new blocks.

You can avoid this restriction by using the module "BetterContacts" from Benny, which you can get from here: https://emaps-eep.de/lua/bettercontacts

Put the module into the LUA folder and prepare your layout for using both modules "BetterContacts" and "blockControl" by adding following lines to the beginning of the Lua code (In this case it is necessary to use a global variable `blockControl`, therefore, you do not declare it as `local`)[13]:

```
require("BetterContacts_BH2")
blockControl = require("blockControl")
```

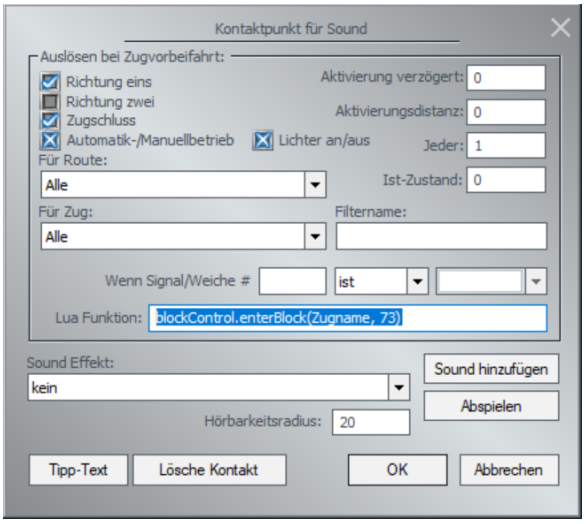
Run the layout once to inform Lua about these modules.

Now you can place signals and corresponding contacts including the Lua function calls without any particular order. Use the parametrized form of the Lua function (replace N with the block signal number)[14]:

```
blockControl.enterBlock(Zugname, N)
```

Here is an example:

[13] In addition you can use the option `BetterContacts = true`, in the `init` call of module `blockControl` to prevent generating global functions for this module.
[14] The module "BetterContacts" offers an additional option to change the variable name `Zugname` as described in the documentation of the module.



Overview how Lua generates automatic train traffic

When a train arrives in a block it runs over the entry sensor, which triggers the following sequence of events via the function `enterBlock` which you have entered into the Lua field of contacts on the EEP layout.

The function simply stores the event. The next call of function `blockControl.run` in `EEPMain` executes all following steps:

1. Decrement the remaining wait time for all trains within blocks.

2. Check if a train has entered a block. If this is the case:
   a. Register the block for the train.
   b. Set the wait time for the train within this block.
   c. Release the previous block and set the previous block signal to “red”.[15]
   d. Release any corresponding two way block.
   e. Release the turnouts on the path behind the train.
   f. Continue traveling if the train has not yet reached the end of the current path, otherwise create a path request.

3. If the main switch is on: Make a list of all possible paths for all trains
   a. who’s train signal (if available) is “green” and
   b. having a path request and
   c. who’s wait time has run out.

   To create this list find paths starting from the current block for which
   a. all via blocks (if there are any) and the destination block are allowed for the train and are free and
   b. for which all turnouts on the path are free.

4. Randomly select a single path from this list and let the corresponding train traveling on this path:

[15] It is possible to release the previous block as soon as the train leaves this block. To inform the program about this event you have to place an additional concat behind the block signal. Use this contact to call function `blockControl.leaveBlock_nn`

a. Lock all blocks and turnouts on the path.
b. Set all block signals (except the last one) of the path to “green”.
c. Set all turnouts of the path according to the defined routes.

Table parameters that define the layout, configured by the user

`trains = {}`
```
trains[1] = {name="Passenger", signal=14, allowed={...}}
trains[2] = {name="Cargo", signal=15, allowed={...}}
```

- **key**. You can use any key to identify entries in the table (Here: 1 and 2).
- ***name=”Passenger”***. Train names (with or without leading “#”) have to match to the train names shown in EEP.
- ***signal=14***. (optional) The signal number that is used as this train’s on/off switch.
- ***allowed={...}***. (optional) The block numbers and wait times where this train is allowed to go (details see below).

Whenever you open a layout file or reload the Lua code, the program does not know where the trains are and starts the “*find mode for trains”*, while trains from this table are identified as well as any other train which enters or stays in a block.

`allowed = {}`
```
allowed[1] = 1
allowed[5] = 23
```

This table specifies if a train is allowed in a block and if it has a stop time:
- not defined or 0: this train is not allowed in this block
- 1: this train is allowed in this block, no stop time
- >1: this train is allowed in this block and it has a stop time. The time the train requires to drive from the block entry sensor to the block signal has to be included. Example: if the measured drive time from sensor to signal of this train in this block is 15 seconds and a stop time of 8 seconds is desired, the number to fill in is 15+8=23.

In the above example:
- The train is allowed in block 1. No wait time is specified.
- The train is allowed in block 5. If the drive time from the sensor to the signal is about 15 seconds It has a stop time of approximately 8s in front of the block signal assuming the drive time from sensor to signal in both blocks is 15s).

`blockSignals = {14,18,16,…}`

(optional)
Every block has a signal where trains will stop if it’s red. The signal is controlled by Lua. It is switched to green when the train is allowed to start a new route to an adjacent block. It is switched to red again upon arrival of the train in the next block.

If a block is used in two directions, this is treated as being two blocks, in opposite directions, each with their own block signal. This is specified in the *`twoWayBlock`* parameter.

IMPORTANT: In EEP not all signals have identical states for “red” and “green”. With some signals 1 means ‘closed’, 2 means ‘open’, while with others it’s vice versa. To deal with this, only use block signals that have the same “red” / ”green” states for the entire layout. The correct states of the block signals in use are defined with variables, see the ‘Variables …’ chapter below.

`twoWayBlocks = { {4, 3} }`

In this example blocks 3 and 4 are two way block 'twins'. If a train reserves block 3 for its new route, Lua reads in `two_way_blocks` that it also has to reserve block 4. Vice versa if a train reserves block 4, Lua reads in `twoWayBlocks` that it also has to reserve block 3.

`routes = {}`

```
route[1]={2,3, turn={ }}
route[2]={3,8, turn={ 7,1, 12,2 } -- 1:main, 2:branch, 3:alternate branch
```

Routes are specified from one block to the next. The table holds the departure block, the destination block and possibly one or more turnouts that need to be switched. In the above example:

- for route 1, 2 is the departure block, 3 is the destination block, and there are no turnouts to be switched.
- for route 2, 3 is the departure block, 8 is the destination block, turnout 7 needs to be switched to ’main’ and turnout 12 to ‘branch’.

IMPORTANT: Currently, only one route can be used between two specified blocks. If several routes are specified, e.g. because the generation program suggests so, then only one of the shortest routes will be used, i.e. one of the routes with the least number of switches.

Sometimes such simple routes could lead to a lockdown situation. A simple but typical example would show 2 parallel tracks in one station, 3 parallel tracks in the other station and a single-track in between.

```
===T1==1===\               /===5==T3===
===T2==2====W1=====3=====W2===4=======
                           \===6=======
```

You can run two trains on this small layout without issues but if you add a third one you might get into trouble if both tracks of the small station are occupied by some waiting trains and the third trains finds a route starting from the larger station: it would enter the single-track in between and then no train can move anymore.
To solve that type of issue you would create but block the critical routes (in this example the routes starting from the larger station) using an additional parameter `routeRequired = true`:

```
routes = {
  { 1,3, turn={ 1,2 }},
  { 2,3, turn={ 1,1 }},
  { [... five routes starting from block 3 ...] },
  { 5,3, turn={ 2,2 }},
  { 4,3, turn={ 2,1 }},
  { 6,3, turn={ 2,3 }},
}
```

You would cover the critical routes in specific paths as described in the next paragraph.

paths = {}
paths[1]= { {4,5,6}, 3, {1,2} }

You define paths starting from some blocks, via one or more blocks and ending in some blocks. You enclose parallel blocks in brackets.

The example above shows a typical situation with 2 parallel tracks in one station, 3 parallel tracks in the other station and a single-track in between.

You can define paths in expanded form as well. However, typical situations like above require multiple primitive paths. For the example above this would lead to this expanded definition:

paths = { {4,3,1}, {4,3,2}, {5,3,1}, {5,3,2}, {6,3,1} {6,3,2}, }

Parameter configured by the user

MAINSW = 27
The address of the signal that is used as the main switch. If the main switch is turned on this enables trains to drive. In addition each train could have an individual on/off switch. If a train switch or the main switch is turned off, trains will first continue driving on their current route, until they reach their destination block where they are halted by a red signal. From tere they will not start a new route until switched on again.

MAINON = 1, MAINOFF = 2, or vice versa
Values can be either 1 or 2, depending on the type of signal used as the main switch. The user has to check this inside EEP.

BLKSIGRED = 1, BLKSIGGRN = 2, or vice versa
The value used in EEP for a red and green block signal, which can be 1 or 2 depending on the type of block signals used. The user has to check this in EEP. Beware to only use block signals that have the same states on the entire layout.

TRAINSIGRED = 1, TRAINSIGGRN = 2, or vice versa
The value used in EEP for a red and green train signal, which can be 1 or 2 depending on the type of train signals used. The user has to check this inside EEP. Beware to only use train signals that have the same states on the entire layout.

Troubleshooting

General tips

- During “*find mode for trains”* all block signals should show “red” and any trains should stop at these signals. During “*find mode for trains”* you will see the signal position in the tipp text as well. Use this to verify if this value matches the parameter `BLKSIGRED`.

- Parameter `blockSignals`
  Use this optional parameter for documentation purpose and to trigger additional consistency checks:
  - Do all blocks used in parameters `allowed`, `twoWayBlocks`, `routes` and `paths` exist?
  - Do all blocks in `blockSignals` have a starting and ending block in `routes`?

- Option `showTippText`
  You can activate/deactivate the tipp texts on signals by toggling the main switch twice or by setting it via function set.

- Option `logLevel`
  You can set this option via function `init` or `set` to show less or more events in the EEP log:
  - 0: no log
  - 1: normal
  - 2: full
  - 3: extreme

How to show the status of signals and turnouts

Use the following piece of Lua code to show the current status of all signals and turnouts in the tipp texts:

```
local function showStatus()
  for i = 1, 1000 do
    -- Show signal status
    local pos = EEPGetSignal( i )
    if pos > 0 then
      local trainName = EEPGetSignalTrainName( i,  1)
      EEPChangeInfoSignal( i,
        string.format("<c>Signal %d = %d<br>%s", i, pos, trainName) )
      EEPShowInfoSignal( i, true )
    end

    -- Show turnout status
    local pos = EEPGetSwitch( i )
    if pos > 0 then
      EEPChangeInfoSwitch( i, string.format("Switch %d = %d", i, pos) )
      EEPShowInfoSwitch( i, true )
    end
  end
end

function EEPMain()
  showStatus()
  return 1
end
```

Technical details about the module

You do not need to read this chapter if you just want to use the module.

The block control program is implemented as a module having a specific interface and not spoiling any global name spaces (except for generated Lua functions which you enter into track contacts on the layout). The module does not use any data slots ot tag texts.

The module contains following parts:
- Initialization (once)
  - Consistency checks
  - Copy user data into internal tables
- Set runtime parameters (at any time)
- Find mode for trains (once)
- Automatic mode (called in EEPMain)
- Show status (at any time)

Initialization (once)

The initialization of the module is implemented in function `init`.
This function runs some consistency checks on the user data, copies the user data into the internal tables, and activates the "*find mode for trains"*.

Consistency checks

It's optional to provide data for parameter `blockSignals` (which is an unordered array of block signal numbers describing which signals are under control of the module). If you provide this parameter, then you get additional consistency checks as part of the init call:
- Do all blocks used in parameters `allowed`, `twoWayBlocks`, `routes` and `paths` exist?
- Do all blocks in `blockSignals` have a starting and ending block in `routes`?

Copy user parameter into internal tables

The user data is not used in its original form but transformed and copied into internal tables. This additional transformation decouples the format of the user input from the internal representation and prepares for performance optimizations.
Example: a complex path like

```
{ {1,2}, 3, {4,5} }
```

in parameter `paths` which is a compact notation and still easy to understand for users is converted into a set of four simple paths like

```
{ {1,3 4}, {1,3,5}, {2,3,4}, {2,3,5} }
```

which allows optimized access for the program.

Set runtime parameters (at any time)

`logLevel`: The module uses two auxiliary functions to print messages in the EEP log window. Function `printLog` checks the current log level as defined with parameer `logLevel` and calls the original `print` function passing all other parameters to this function..
Function `check` does not use option `logLevel` but uses a similar approach to check a condition and prints the message if the condition is not met. This is similar to Lua function `assert` but without stopping the program flow.

showTippText: The function `showSignalStatus` is called in every cycle of function `EEPMain`. The boolean value of option `showTippText` is directly used to activate/deactivate the tipp texts.[16]

start: You can use this option to start (true) or stop (false) the main signal (instead of calling `EEPSetSignal` by yourself). During"*find mode for trains"*, the start is delayed until all trains are identified and assigned to a block. Therefore you can set this option even during "*find mode for trains"* without disturbing the program.

startTrains: Use this option to activate or deactivate all train signals (instead of calling `EEPSetSignal` for all train signals by yourself).

Find mode for trains (once)

The purpose of the "*find mode for trains"* is to find all trains running on the layout and identify the blocks where these trains are located. Because of this mode no data slots are required to keep track of the internal status of EEP between closing and reopening a track layout. You can save and load the layout at any time!
The "*find mode for trains"* first sets the main signal and all block signals to "red". Trains are either already within the reach of a block signal or are traveling towards the next block. In any case these trains get identified and will stop at the block signal.
The tipp texts on signals are activated and show the status of the blocks. (In addition you will see the signal position as well to verify if all signals show "red").
To end the"*find mode for trains"* you have two options:

a) manually: simply check the layout if all trains came to a stop at a block signal. Then activate the main signal.
b) automatic: if you provide a complete table *`trains`* containing all trains then you can request automatic start via runtime parameter *`start`* = *`true`* which you can set using the function `set`. As soon as the"*find mode for trains"* has identified all trains it will start run mode.

Run mode (called in EEPMain)

The sensors (contacts) inform the module about trains entering a block by calling a generated function per block.
Function `run` is the core of the module. The function

- Identifies trains which have left blocks and releases these blocks (optional, using additional contacts),
- identifies trains which have entered blocks and initializes the wait time for these trains,
- decrements the wait time for all trains,
- searches for available routes/paths as soon as the wait time is over,
- chooses one of the available paths by random,
- releases blocks and turnouts behind the trains,
- locks blocks and turnouts on the path of the trains,
- and finally sets the block signals to let the trains travel.

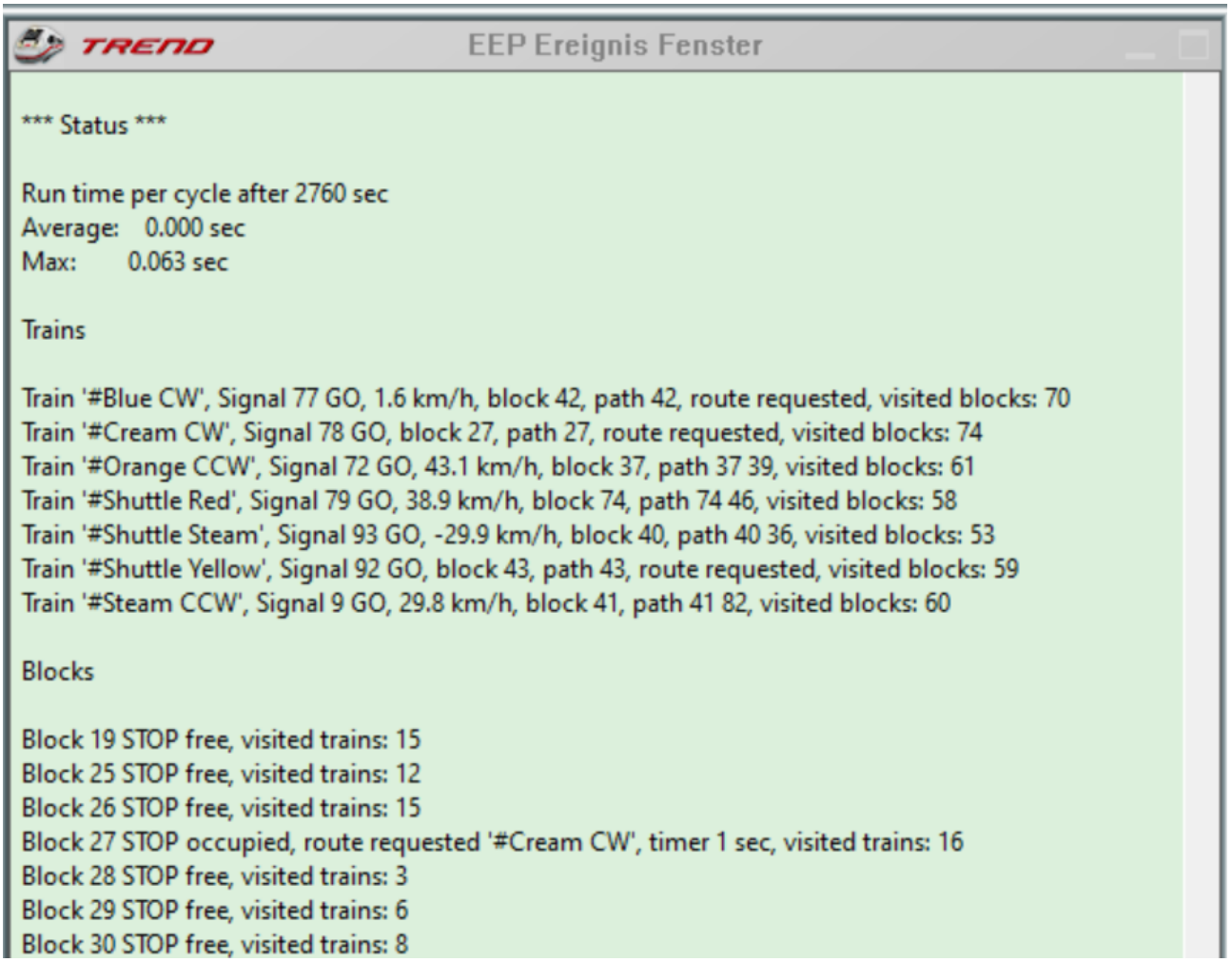
[16] Some of the following format codes are used in the tipp texts: `<j>` left-aligned, `<c>` centered, `<r>` right-aligned, `<br>` line break, `<b>` bold `</b>`, `<i>` italic `</i >`, `<fgrgb=0,0,0>` font color, `<bgrgb=0,0,0>` background color, for details see https://www.eepforum.de/forum/thread/34860-7-6-3-tipp- texts-f%C3%BCr-objects-and-contact-points/

Show status (at any time)

With a call to the function `blockControl.printStatus()` you can display the current status of trains, blocks and routes in the EEP event window. The call can be made via the Lua function of a contact or via any other function. If you insert the call directly or indirectly in `EEPMain`, then you can specify an additional parameter with which the repetition rate in seconds is determined. In this example, the status is displayed every 60 seconds:

```
function EEPMain()
  blockControl.run()
  blockControl.printStatus(60)
  return 1
end
```

Result:



Data internally used by the Lua module

Table `TrainTab`

In addition to the components `name`, `signal` and `allowed` we see following components in this table:

`block`

Current block where the train is (or `nil`)

`path`

Current remaining path on which the train is traveling (or `nil`). When a train gets a new path, it's copied into this component. Whenever a train reaches a next block on its current path this block gets removed.

Table `pathTab`

This table contains the expanded paths which are available for routing.

Table `routeTab`

This table contains the turnout settings for the path between two adjacent blocks.

Table `BlockTab`

In addition to the components `signal` and `twoWayBlock` we see following components in this table:

`reserved`

This component holds `nil` if a block is free. If a block is reserved, it holds the train that reserves it. These blocks are not available for new routes.
If a train reserves a route, it not only reserves the via blocks (if there are any) and the destination block, it also reserves the turnouts between the blocks to prevent other trains from using them. When a train arrives at the next block of its route, the block and the turnouts behind the train that were reserved for this route are released.

`occupied / occupiedOld`

These components are used to catch events for arriving trains in blocks.

`request`

When a train is ready to start a new route it is registered here.

`stoptimer`

Stop times, per train per block, are specified in the *`allowed`* table. Upon arrival *`stoptimer`* is initialized with 5x the defined stop time (because Lua cycles 5x per second) and the stop timers are decremented each cycle. When `stoptimer` reaches 0, the stop time for this train in this block has run out and the train becomes available for a new route.

Table `availablePath`

This table is cleared every Lua cycle and then built up again. It holds the 'available routes for all the trains that requested new routes. Lua randomly selects one route from the list.
That train will be the only one allowed to start a new route during this Lua cycle. Lua will now reserve the via blocks (if there are any) and the destination block and the turnouts of this route.

This could mean that other routes that were available this same cycle may now not be available anymore, which is why the table is cleared and recalculated every Lua cycle.