EEP Lua Automatic Train Control (ATC)

Rudy Boer, Frank Buchholz, July 2022
Version 2.3.2

Contents

These are clickable links.

Purpose

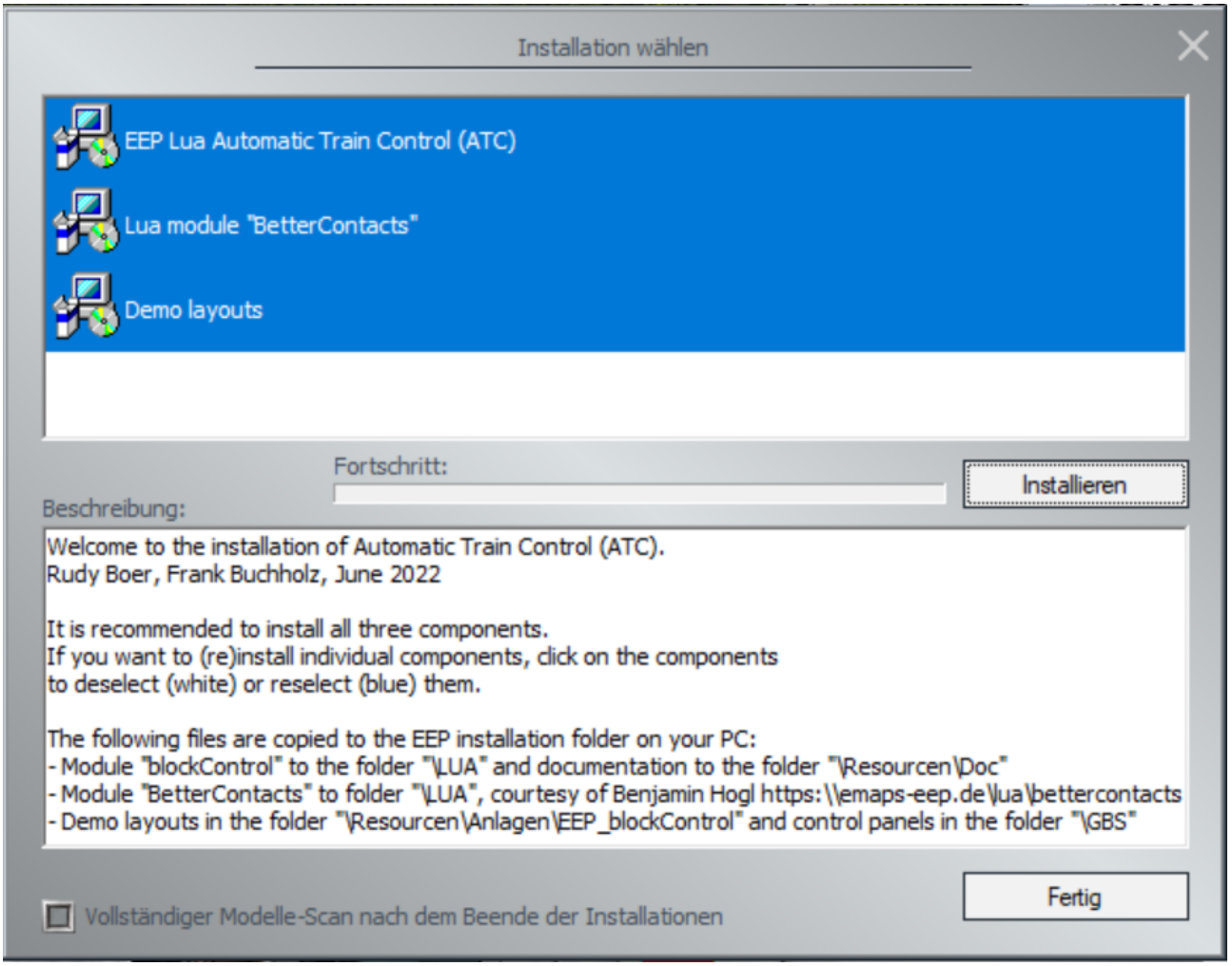
This Lua module automates train traffic on any EEP layout, without the need to write any Lua code yourself. All that is needed from the user is to describe which trains are allowed in which blocks in Lua tables. Stop times, for example at railway stations, can be specified in these tables too, per train (group) and per block.

The Lua module is going to drive trains from block to block. A block is a stretch of track that starts with a contact that triggers a Lua function when a train has entered the block, and that ends with a signal that is controlled by the Lua module to halt or to release a train.

The turnouts between the blocks are switched by the Lua module. Which turnouts to switch, is defined in the routes table. This table can be manually created by the user, but it can also be automatically generated with the Lua ATC Code Generator tool.

Installation

Download the `EEP_blockControl.zip` file and install it via the “Install models” button on the EEP front page followed by selecting the downloaded ZIP.



This opens the EEP installer. It's recommended to install all three components that are selected by default. If you want to install individual components, click the components to deselect (white) or reselect (blue) them.

The following files are copied to the EEP installation folder on your PC:

- Module **blockControl** to the folder `\LUA`
  Documentation to the folder `\Resourcen\Doc`
- Module **BetterContacts** to folder `\LUA`
  Courtesy of Benjamin Hogl https:\\emaps-eep.de\lua\bettercontacts
- **Demo layouts** to the folder `\Resourcen\Anlagen\EEP_blockControl`
  Control panels to the folder `\GBS`

No models are included, therefore it's not necessary to scan new models afterwards.

Another option for the installation is to unzip the archive file and to move the files manually:

- LUA folder:
  - Move the file `blockControl.lua` to your EEP installation `\LUA` folder. With this the installation is actually complete.
  - The files `blockControl_template_ENG.lua` respective `blockControl_template_GER.lua` can be used as a starting point to develop your own Lua ATC scripts. Save them anywhere you like for later use.
  - Move the file `BetterContacts_BH2.lua` to your EEP installation `\LUA` folder. This module from Benny can be used to simplify the definition of the Lua line in contacts.
- EEP_blockControl folder:
  - This folder contains multiple demo layouts with fully working examples of ATC. Save these anywhere you like.
- GBS folder:
  - Move the files inside the GBS folder to your EEP installation `\GBS` folder. This is only needed if you plan to adjust the GBS of the demos.
- User manuals. Save these anywhere you like. The⁰ standard location for EEP documents is `\Resourcen\Doc` in your EEP installation.
  - English
  - German

For your convenience you may want to add these links to your browser's favourites. These are the tools used to auto-generate the Lua code that specifies the routes in an EEP layout:

- EEP Layout Tool
- Lua ATC Code Generator

You may also like this tool that shows the complete inventory of an EEP layout:

- EEP Inventar Tool

Steps to Create Automatic Train Traffic in EEP

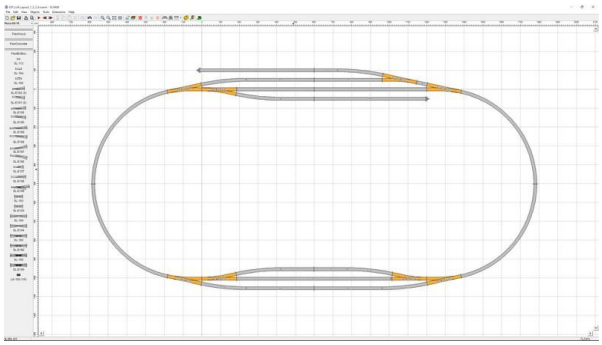
1. Design a (model) railway layout and plan the train flow, which means, decide where different trains should (be allowed to) go, and where they should (be able to) stop.[1]
2. Lay the tracks in EEP.
3. Trains will drive from block to block. Create the blocks that will make the intended train traffic possible by placing a signal at the end, and a contact at the start of each block.
4. Go to 3D mode and place the desired trains. Give them a name and give them their intended speed. Drive them to a red signal, where they will stop.
5. Save the EEP layout.
6. Open the EEP Layout Tool in a browser tab and load the layout.
7. Open the Lua ATC Code Generator in another browser tab and press the 'Generate' button.
8. While in 3D mode, paste the Lua code into the Script Editor and click 'Reload script' to run the code. This creates the Lua functions that the contacts are going to trigger.
9. In each contact enter the Lua function: `blockControl.enterBlock_#` , where # is the block signal number.[2]
10. Create the tables that define the allowed blocks per train, or train groups.
11. Edit the trains table such that each train has a table containing allowed blocks (`allowed=...`), and optionally an on/off train signal (`signal=#`), and, only for EEP versions 14.1 or lower, a slot (`slot=#`) to store data.
12. Save the EEP layout. Go to 3D mode, open the Script Editor and click 'Reload script' to run the code. Lua starts up in 'Train Find Mode'.
13. When all trains are found, turn the main switch on, as well as one or more of the individual train switches and … be amazed and have fun!

Each step is discussed in detail in the following chapters.

Follow this link to YouTube-Video 9 for an introduction into version 2.3.

[1] Decide if you want to use the module betterContacts from Benny
[2] If you are using betterContacts, then you use another Lua function `blockControl.enterBlock(Zugname, ##)` which you already can add in the contacts while in step 3.

1 Design a (Model) Railway Layout

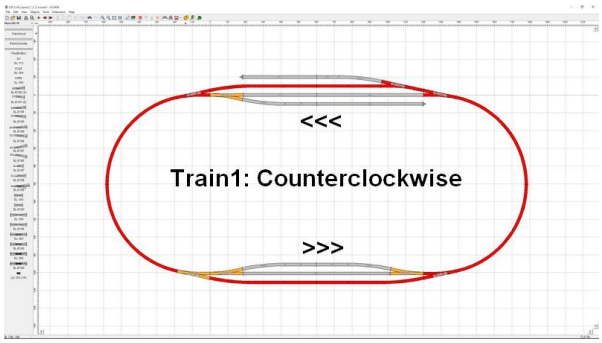


When designing a (model) railway layout, it can be shaped in our mind, or sketched on paper, or a model railway design program can be used.

This drawing of a small model railway layout has been made with SCARM.

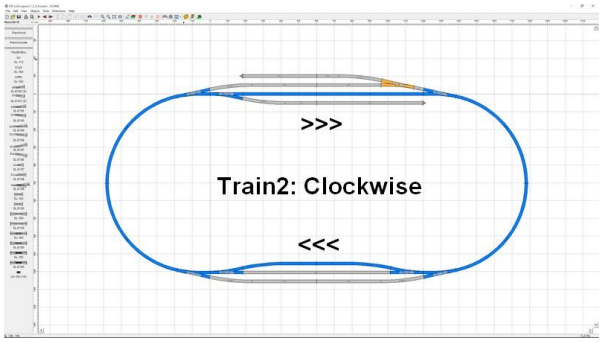
The plan is to drive with 3 trains on this layout.

An easy choice in designing traffic flow may seem to simply allow every train to drive everywhere. This probably won't look very realistic though. In reality trains drive right- or left handed, depending on the country. Also we don't want passenger trains to end up in an industry area. And … allowing all trains everywhere can lead to a deadlock when opposing trains, that both have nowhere else to go than the block the other train is occupying. For more info, see chapter How to Avoid Deadlocks.



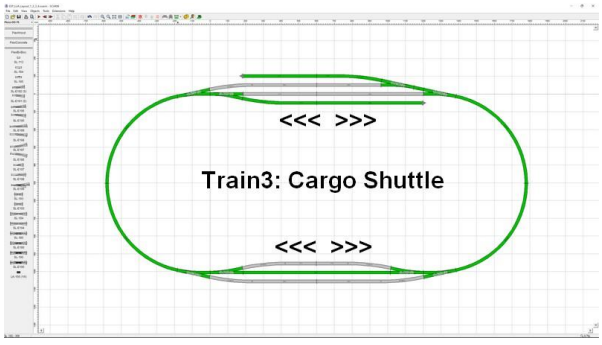
Careful traffic planning helps to run the trains with increased realism and without issues. Let's see how 3 trains can nicely drive around on this layout.

Train 1 is a passenger train that drives around counterclockwise, with a scheduled stop of say 30 seconds at South, and no scheduled stop at North. Which doesn't mean it will never stop there, it'll have to wait if there's traffic on the single track curve



Train 2 also is a passenger train, it drives around clockwise. It'll also have a scheduled stop of say 30 seconds at South, and no stop at North.

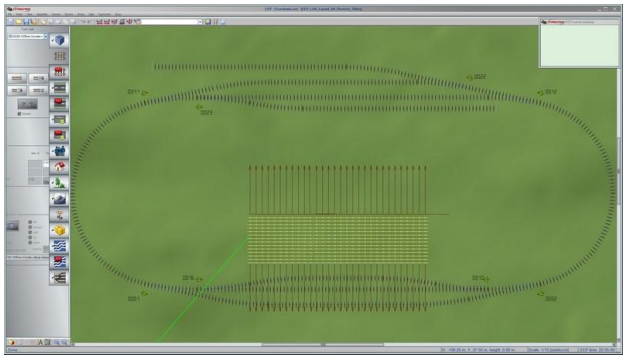
For both passenger trains we use right hand driving at the stations.

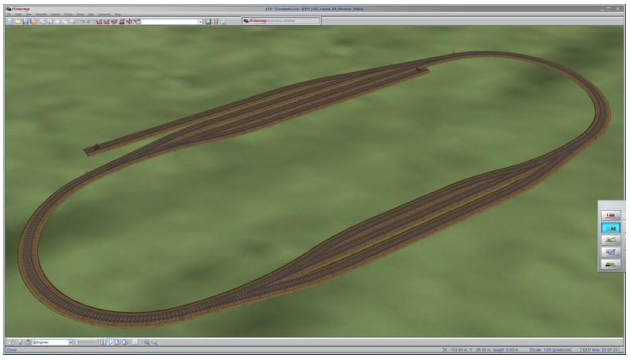


Train 3 is a cargo train that shuttles between the two dead ends at North, where it has scheduled stops of say 30 seconds. At South it uses the middle track, with no scheduled stop. This middle track has two way traffic.

2 Lay the Tracks in EEP

The EEP file for this layout is `EEP_LUA_ATC_Demo_4`, which came with the installation.





Of course you can also choose to lay the track for this layout yourself. If you like to recreate a (model) railway layout with great precision, this video shows how an image can be used as a template in the EEP 2D / 3D view, to lay the track on.

The vertical tracks in the middle, that only show in 2D view, are a row of invisible tracks, intended to place signals for our main switch and individual train on / off switches.

3 Create the Blocks for the Intended Train Traffic

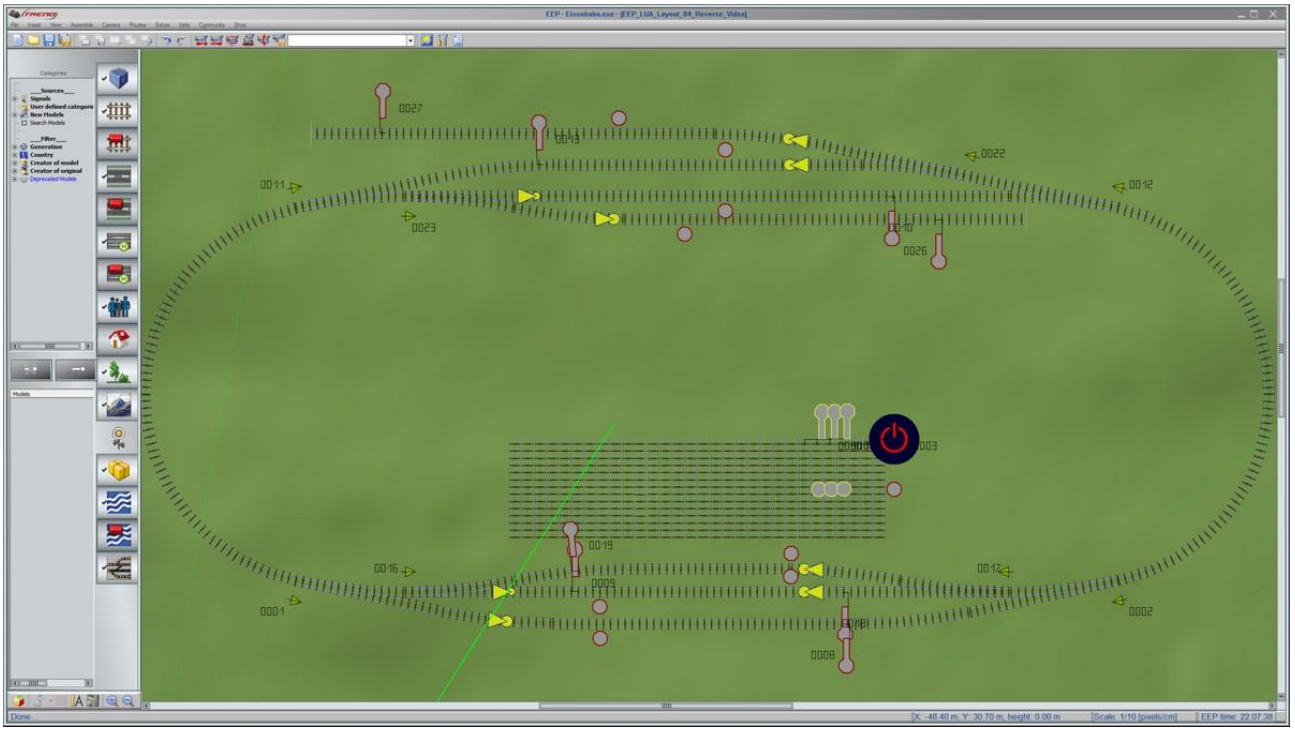
Lua ATC will send the trains from block to block. A block is a stretch of track with a contact at the beginning and a signal at the end. The block entry contacts can be of any type. In the demos ‘Sound’ contacts are used, with their yellow colour they have good visibility. The block signals are going to be controlled by Lua. Trains will be released when their stop time has passed and when the next block and the turnouts in between are not reserved by another train.

How to divide the layout into blocks is entirely up to you. There’s only one important **RULE**: there should never be a turnout inside a block. Turnouts are only allowed on the routes between blocks. Lua switches the turnouts into the states needed to reach the next block.

The goal of the division into blocks is to enable the train flow we designed in step 1. The signals are the only places where trains will stop, so, a logical place for signals is at the railway stations, and at the end of dead end tracks. On long stretches of track blocks can be placed in series to allow a smoother follow up of trains, just like in reality.

On our demo layout we create a block on every track in station North as well as in -South. The middle track in South is two way, it therefore gets a signal and a contact at both ends to create actually two blocks, one for each direction.

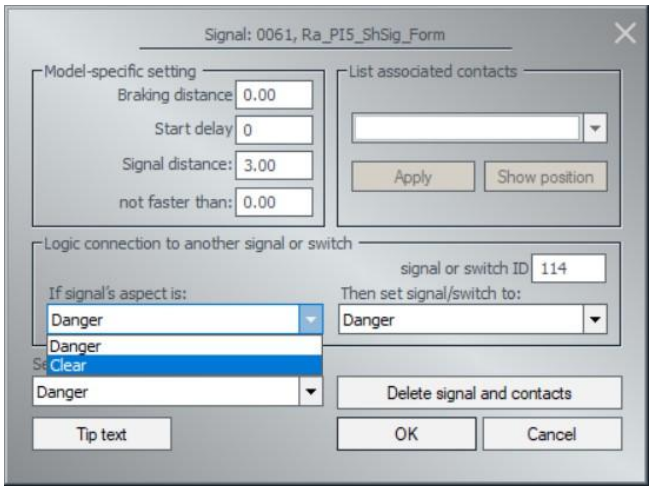
Why not make the curves a block you may ask? Well … it’s not necessary for traffic flow, but if you like to have some more signals, as eye-candy, both curves can be turned into two-way blocks.



We also place a main switch on one of the invisible tracks. A signal of the type *Signals* > *Other* > *On-Off switch* will be auto-detected by the Lua ATC Code Generator.

Every train may also get its own individual on / off switch. Signals of any type can be used for these. Trains without such a signal will always drive and can not be switched off.

Unfortunately in EEP different signal types can have different numeric values for their on / off states. Lua needs consistency, therefore there is this **RULE:** only use signal types that have similar on / off states. This holds for both the block- and the train signals, although their states may differ because both can be specified separately. Refer to the chapter Change Signal States how to tell Lua about the states of the used signals, if needed (only required if non-default)[3].



Note that the dead ends don't require a train contact for speed reversal. The train will stop at the (invisible) signal. Lua knows this is a dead end and it will reverse the train speed for us. More on this in chapter Train Reversal on Dead End Blocks.

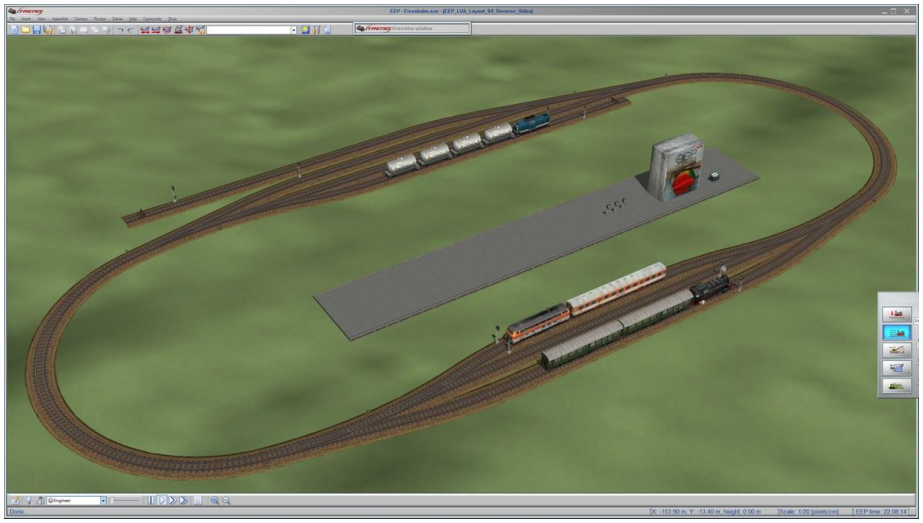
For the sake of realism you may want to place a visible signal at the point where the train leaves the dead end. Such a signal has to be connected to the block signal that is controlled by Lua, red to red, green to green, as shown in the image.

[3] It is possible to connect other visible signals to invisible signals used for ATC. Such connected signals could have other signal patterns.

This signal is just for ‘eye candy’, it has no role in the Lua ATC and in fact it must be excluded from the Lua ATC Code Generator or things will go wrong. In step 7, when we generate the Lua code, we’ll see how we can exclude these signals.

Trains can also be reversed on non dead end blocks. This enables more variation in the train traffic. See chapter Train Reversal on Non Dead End Blocks.

4 Place Trains in 3D Mode



In step 1, layout design, we decided how many and what type of trains we wanted to drive on this layout. This is a good moment to put EEP in 3D mode and place the trains.

In the pop up window that shows after placement, give each train a name by which it can be easily recognized later. This name will be used in the Lua configuration.

Wagons can be added too. Their names are irrelevant.

Ensure that there is a free block in front of a new placed train. Give the train a speed and let it drive to the red signal of the next block, taking care to first switch the turnouts such that the train ends up in a block where it is allowed according to our traffic plan.

For our demo layout, the image above shows the result so far. Notice how the blue train is placed in a dead end. This generally isn’t a good idea because at first startup Lua doesn’t know yet in which direction it has to reverse the train, it may run into the buffer. Best place all trains in non dead end blocks.

It’s good practice to deactivate the front and rear couplings of the trains to avoid that trains merge when they accidentally run into each other.

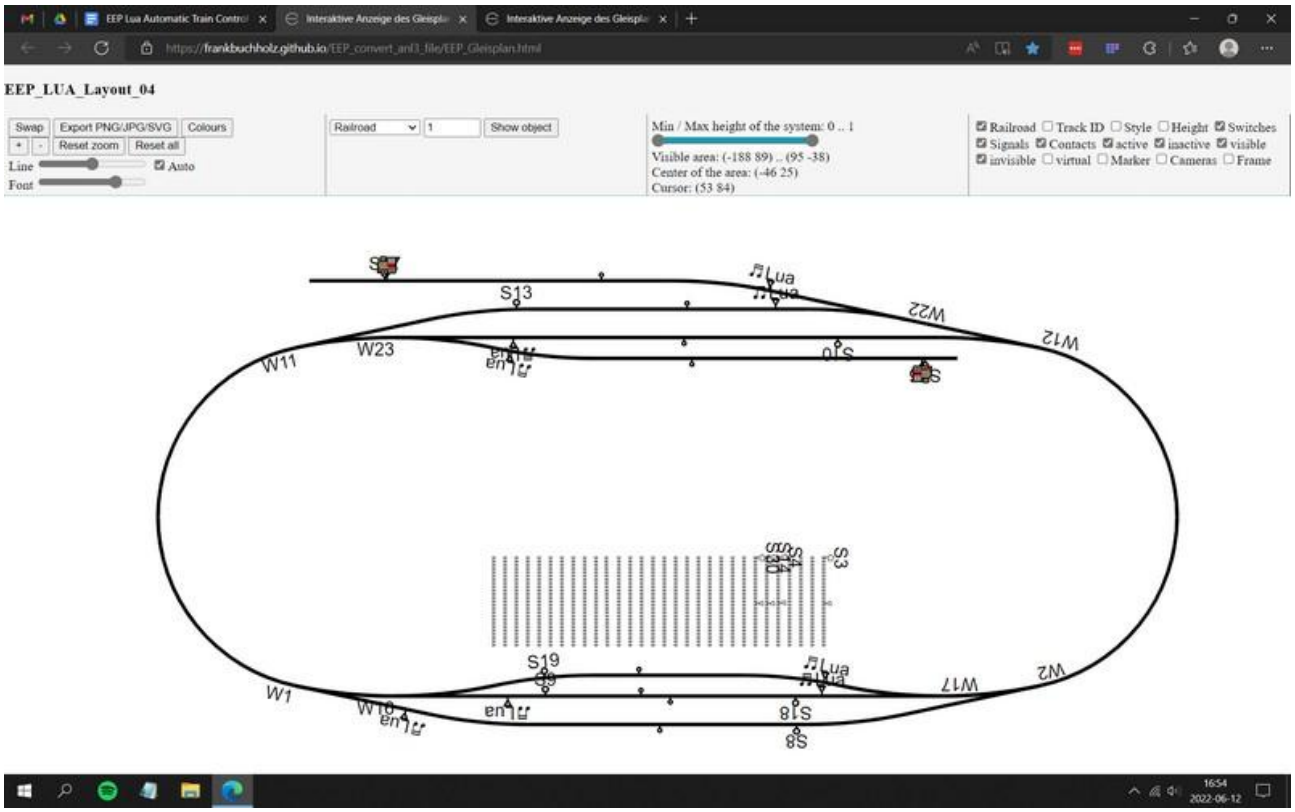
5 Save the Layout

We need the EEP `.anl3` file in the next step. When all signals, contacts and trains have been placed and all trains have been given a name and been driven to a red signal, it’s time to save the layout. This can be done via the menu *File* > *Save layout as*.

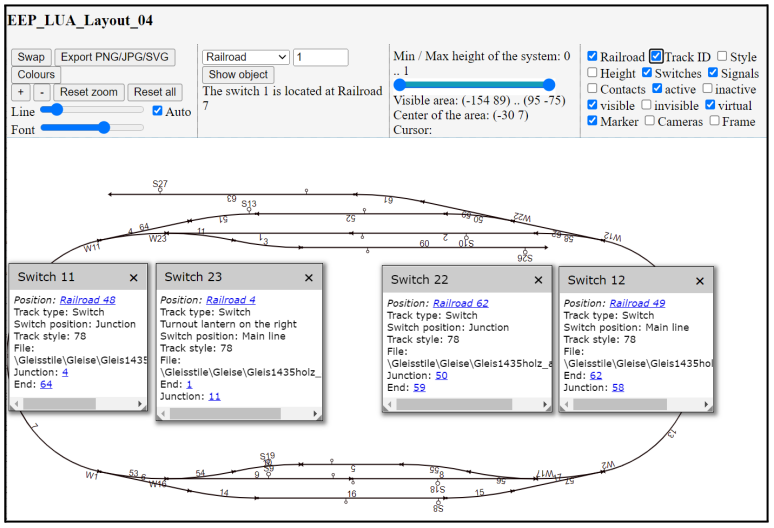
6 Open the Layout in the EEP Layout Tool

Open the EEP Layout Tool in a web browser tab. The display language can be changed with the buttons at top right.

Load the EEP layout file via the ‘Choose file’ button. The screen should look something like this. Visibility of items can be changed via the tick marks top right. Colours, line thickness and text size can be changed via the buttons and sliders top left.



You can now check whether the signals, switches and contacts correspond to your plan.



If you like to create the routes table yourself and not use the auto generator (next chapter) be aware that with EEP sometimes a turnout’s ‘main’ and ‘branch’ states can be confusing … what visibly looks like ‘main’ can have the label ‘branch’.

What can help is to 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 EEP Layout Tool. If you left click on a turnout, an info popup opens that shows its state and some other info as well. Shift + left click opens multiple of these pop ups.

7 Generate the Code with the Lua ATC Code Generator

Open the Lua ATC Code Generator in another browser tab. The previously selected language will also be used here.

Press the ‘Generate’ button. The tool now generates the Lua ATC code for this layout, where the main job is to create the routes table … It's a great help not to have to figure this table out manually.

```
-- EEP File 'EEP_LUA_Layout_04'
-- Lua program for module 'blockControl.lua' for track system 'Railroad'
-- For every block entry contact enter the following into field 'Lua function' (where ## is the
number of the block signal): blockControl.enterBlock_##

-- 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, speed=44 },
  { name="#Orange", signal=0, allowed=all, speed=67 },
  { name="#Steam",  signal=0, allowed=all, speed=62 },
}

local main_signal = 3

local block_signals = { 8, 9, 10, 13, 18, 19, 26, 27, }
-- 8 signals use BLKSIGRED = 1, BLKSIGGRN = 2

local two_way_blocks = { { 9, 18 }, }

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, }, reverse=true },
  { 26, 18, turn={ 23,2, 11,2, 1,2, 16,1, }, reverse=true },
  { 27, 9, turn={ 22,1, 12,1, 2,2, 17,1, }, reverse=true },
  { 27, 19, turn={ 22,1, 12,1, 2,2, 17,2, }, reverse=true },
}

-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- Remaining part of main script in EEP
-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

clearlog()

local blockControl = require("blockControl")   -- Load the module

blockControl.init({                -- Initialize the module
  logLevel        = 1,             -- (Optional) Log level 0 (default): off, 1: normal, 2: full, 3:
extreme

  trains          = trains,        -- (Optional) Unknown trains get detected automatically,
however, such trains do not have a train signal and can go everywhere.
```

```
  blockSignals    = block_signals,  -- Block signals
  twoWayBlocks    = two_way_blocks, -- Two way twin blocks (array or set of related blocks)
  routes          = routes,         -- Routes via turnouts from one block to the next block
  paths           = anti_deadlock_paths, -- Critical paths on which trains have to go to avoid
lockdown situations
  MAINSW          = main_signal,    -- ID of the main switch (optional)

  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
})

--[[ Optional: Set one or more runtime parameters at any time
blockControl.set({
  logLevel        = 1,              -- (Optional) Log level 0 (default): off, 1: normal, 2: full, 3:
extreme
  showTippText    = true,           -- (Optional) Show tipp texts true / false (Later you can toggle
the visibility of the tipp texts using the main switch.)
  start           = false,          -- (Optional) Activate / deactivate main signal. Useful to start
automatic block control after finding all known train.
  startAllTrains  = true,           -- (Optional) Activate / deactivate all train signals
})
--]]

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

NOTE: If you have placed additional signals as eye candy, for instance at dead end blocks, these need to be excluded from the Lua code generation. If they're not excluded, wrong blocks and routes will be created and the trains won't run properly. Signals can be excluded by listing them in the appropriate field on the Lua Code Generator page.

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.
- You get reversing routes for dead-ends automatically, however, you have to define any reversing routes for two-way-blocks by yourself.
- Garbage in - garbage out: If the layout does not have proper block signals you cannot expect sound results.

Follow this link for a YouTube video 6 on the Lua ATC Code Generator

For info, you can us the Inventar program[4] to review the settings of the contacts including the Lua function in contacts. You may want to hide some columns to optimise 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 ...

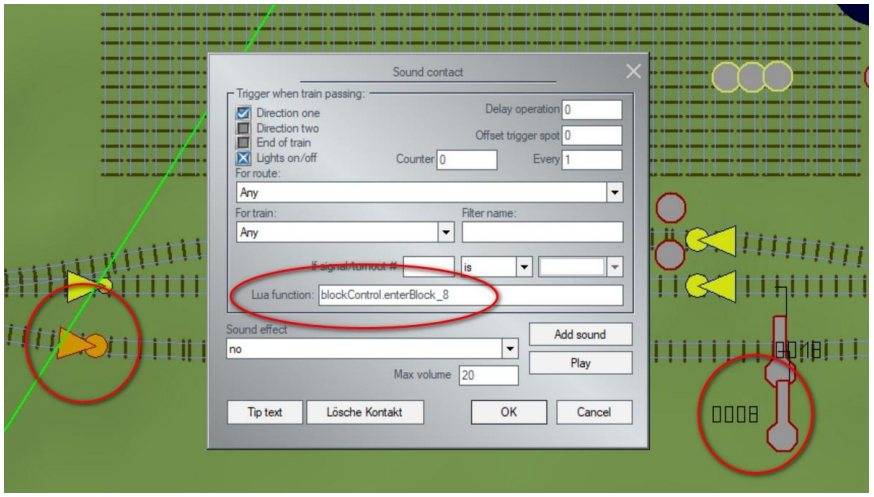
Type of contact	Type of contact	Track	Trigger	Train position	Lua function	Tip's text
5	Vehicle -0 km/h max	Railroad 60	in the direction of the track	Front		Block 26 reverse direction
5	Vehicle -0 km/h max	Railroad 63	in the direction of the track	Front		Block 27 reverse direction
256	Sound undefined	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

8 Paste the Code in the Lua Script Editor and Reload

The code generated by the tool is automatically copied into the clipboard. Return to the EEP 3D view, open the Lua Script Editor and paste the code there.

Now press ‘Reload script’ to run the code. This will create the necessary functions for the block entry contacts.

9 Place the ‘enterBlock’ Functions in the contacts



We need to add a Lua function call to every block entry contact.

Go to 2D mode and paste

`blockControl.enterBlock_#`

into the Lua function field[5], where # is the number of the block signal. Do this for each block entry contact.

10 Create the Allowed Blocks Tables and Wait Times

The code generator does not know our intentions with the traffic plan. It created a table called ‘all’, which contains all blocks, and in the trains table it allows each train in ‘all’ blocks:

```
-- 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, speed=44 },
  { name="#Orange", signal=0, allowed=all, speed=67 },
  { name="#Steam",  signal=0, allowed=all, speed=62 },
}
```

In the traffic plan that we designed in step 1, ‘#Steam’ runs counterclockwise, ‘#Orange’ runs clockwise and ‘#Blue’ shuttles between the dead ends. To define this, we replace the ‘all’ table by three tables:

```
-- Allowed blocks with wait time
local CW      = { [10]= 1, [19]=30, }
local CCW     = { [ 8]=30, [13]= 1, }
local Shuttle = { [ 9]= 1, [18]= 1, [26]=30, [27]=30, }
```

The number behind the = represents a wait time. For example `[19]=30` means the train will stay in this block at least 30 seconds. This includes the driving time from the block entry contact to the signal. A value of 1 second effectively means the train will drive thru, unless other traffic forces it to stop. For more information on wait times and how to randomise them see chapter Random Wait Times.

See the chapter How to Optimise the Definition of Allowed Blocks for more info on ways to define the allowed blocks tables and how to avoid repetition, if the allowed blocks for different trains differ by only a couple of blocks there are elegant ways to specify this efficiently.

NOTE: the EEP Lua Script Editor isn’t the nicest editor to work with. Editing the code is much easier with an external editor like Notepad++.

11 Edit the Trains Table

We now allocate an allowed blocks table to each train in the trains table:

```
local trains = {
  { name="#Blue",   signal= 4, allowed=Shuttle, speed=40, },
  { name="#Orange", signal=14, allowed=CW,      speed=70, },
  { name="#Steam",  signal=30, allowed=CCW,     speed=60, },
}
```

In this example each train has its own table of allowed blocks. It is perfectly fine though to allocate the same table to multiple trains, as it was the case with the default ‘all’ table.

[5] If you are using BetterContacts, then use this Lua function instead:
`blockControl.enterBlock(Zugname, #)`

It's not mandatory to point to an allowed blocks table. The allowed blocks can also be listed inside the trains table, like this:

```
{ name="#Blue", signal=1, speed=44, allowed={ [9]=1, [19]=1, [26]=30, [27]=30, }, },
```

For small layouts this can work fine. As soon as more than one train is allowed on the same blocks though, the named tables are more convenient. They avoid excessive repetition of similar data, comments are easier to include, and changes are easier to perform.

The signal number for each train is the one used as their individual on / off switch.

NOTE for users of EEP versions 11 - 14.1:
Users of EEP v14.1 and lower have to add `,slot=#,` to each train, where # is a unique number for every train. Example:

```
{ name="#Blue", signal=1, allowed=Shuttle, speed=44, slot=1, },
```

These slots are file memory locations where Lua stores data between sessions. EEP v14.2 and higher don't need these slot numbers, in these versions data is stored in a different way.

12 Train Find Mode

We're ready for a try out.

- If the EEP Event Window does not show yet, enable it via the tick mark in the EEP Settings.
- If an external editor was used, this is the moment to copy & paste your code into the Lua Script Editor.
- Click 'Reload script' and keep an eye on the EEP Event Window. It should say 'Find Mode is Active'.
- The Lua module starts in 'Find Train Mode'. It'll automatically try to detect all the trains on the layout, which it is able to do if trains are 'caught' by a block signal.
- If it doesn't succeed in finding all trains it won't get out of find mode. Manual intervention is needed. On the Event Window find out which train(s) have not been detected. Set the appropriate turnouts to allow it to drive to the next block and switch its signal to green. It'll now drive to the next block and as soon as it's 'caught' by a signal, Lua will find it.
- Repeat this until all trains are found.

13 Ready to Run Some Trains!

Turn the main switch on, and, if not already on, turn the individual train switches on.

Be amazed and have fun!

How to Stop Driving and Save the Current State

Before exiting EEP it's good practice to turn off the main switch and wait until all trains have come to a stop at a red signal.

However, thanks to the 'Train Find Mode' which starts when loading a layout file or after 'Reload script', the layout can be saved at any moment, even while trains are driving. In the next session

Lua will re-find the trains. If you haven't made any changes to the layout and haven't done any Lua editing, there's no need to save the layout via the menu, the current state is automatically saved when EEP is exited.

Add or Remove Trains Later

If later you like to add or remove trains, then:

- Switch off the main switch and let all trains come to a stop.
- In 3D mode remove a train, or place a new train and give it a name
- Give it a speed
- Let it drive to a block where it is allowed, where it will stop at the red signal
- Open the Lua Script Editor
- If needed add a new 'allowed' table
- Edit the trains table to add the train, its on / off signal and its allowed blocks.
- Click 'Reload script'
- Lua will now be in 'Train Find Mode' again. Refer to chapter 12 Train Find Mode to find all trains. Then restart the layout.

Train Reversal on Dead End Blocks

To reverse trains, the 'classical; method would be to use a train contact with speed reversal as its action, at the moment the train was already at very low speed, or just started accelerating such that the reversal is almost unnoticeable and without jerk.

With Lua ATC the contacts can be left out. We tell Lua to reverse a train with the token `reverse=true`. For dead ends, this is automatically generated by the Lua ATC Code Generator.

Advantages:

- Less train contacts are needed.
- No more precise contact positioning is required to make the reversal look good without jerkiness.
- There's no visible speed reversal anymore. Trains start to drive and accelerate smoothly in the correct direction.
- In addition to reversal at dead end blocks, trains can now also be reversed at non dead end blocks. A block can have forward routes as well as reversing routes, with even distributed probability over all possible routes.

For Lua to know the desired (reversed) train speed we add a 'speed' configuration setting in the trains table. Lua uses this setting when it reverses the train direction[6]:

```
local trains = {
  { name = "#Orange", signal = 4,  allowed = everywhere, speed = 80, slot=1, },[7]
  { name = "#Steam",  signal = 30, allowed = everywhere, speed = 45, slot=2, },
  { name = "#Blue",   signal = 14, allowed = everywhere, speed = 50, slot=3, },
}
```

[6] Always use positive speed values.
[7] With EEP versions as of 14.2 the slot numbers can be omitted.

These speeds will automatically be listed in the trains table if the Lua ATC code Generator is used with trains already placed on the layout. It's only used to reverse trains though, it has no function for trains that never reverse direction. Speeds mentioned for those trains are just for information.

The speed of trains can still be changed while driving manually using the EEP controls, or via train contacts, however, when a train reverses direction it will get the speed from the trains table.

Train Reversal on Non Dead End Blocks

Train reversal via Lua code instead of via contacts makes it possible to reverse trains on a normal block, or on a two-way block. This is done by manually adding a route to the routes table, from the current block to a previous block, and adding `reverse = true`. Example:

```
{ 8, 26, turn={ 23,2, 11,2, 1,1, }, reverse=true },
```

Something to be aware of with two-way blocks is that when a train reverses, what previously was the end of the train now becomes the head of the train. If the train departs in the opposite direction, it's this new head of the train that passes contacts and pre-signal and triggers action.

For two way blocks where we want a train reversal to take place this can be the pre-signal of the two way block itself, or it can be the pre-signal of the next block, depending on the placement of the pre signal and the length of the train.

We now can have two situations:

1. The pre-signal of the two way twin block is the first to come across. In this case the route from the current block to its two-way twin block must be defined in the routes table.
2. The pre-signal of the next block is the first one. The route from the current block to this next block must be defined in the routes table.

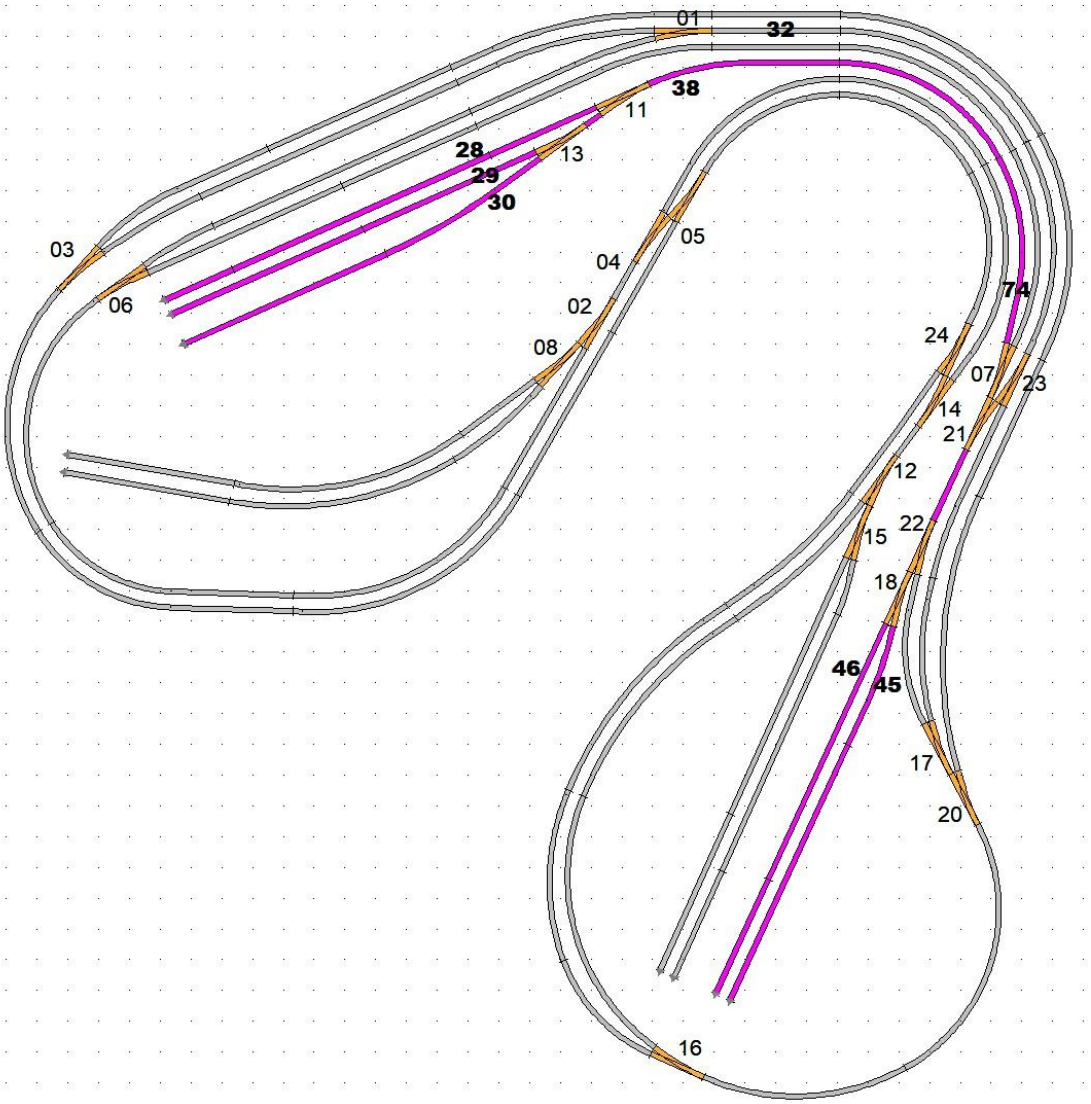
RULE: it's important that this situation is the same for ALL trains. If we'd mix these situations, trains might get stuck at the twin block red signal. Please carefully place the pre signal in the twin block such that it is either behind, or in front of, the tail of ALL trains.

This YouTube video 8 shows how to reverse trains in any block, not necessarily a dead end.

How to Avoid Deadlocks

A situation can occur where trains are driving towards each other in opposing directions and none of them can find a free block to go to. Both trains will come to a halt and a so-called 'deadlock' has now occurred.

The picture below shows a section of demo layout "EEP_LUA_ATC_Model_Railway_Layout_1" 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 the demo layout 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 Lua module.
- The third solution is to tell the Lua module about the potential deadlock path and to look further than one block ahead. Lua should not allow a train on 28, 29, or 30 to start driving if there is no free block in 45, or 46. We tell the Lua module 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

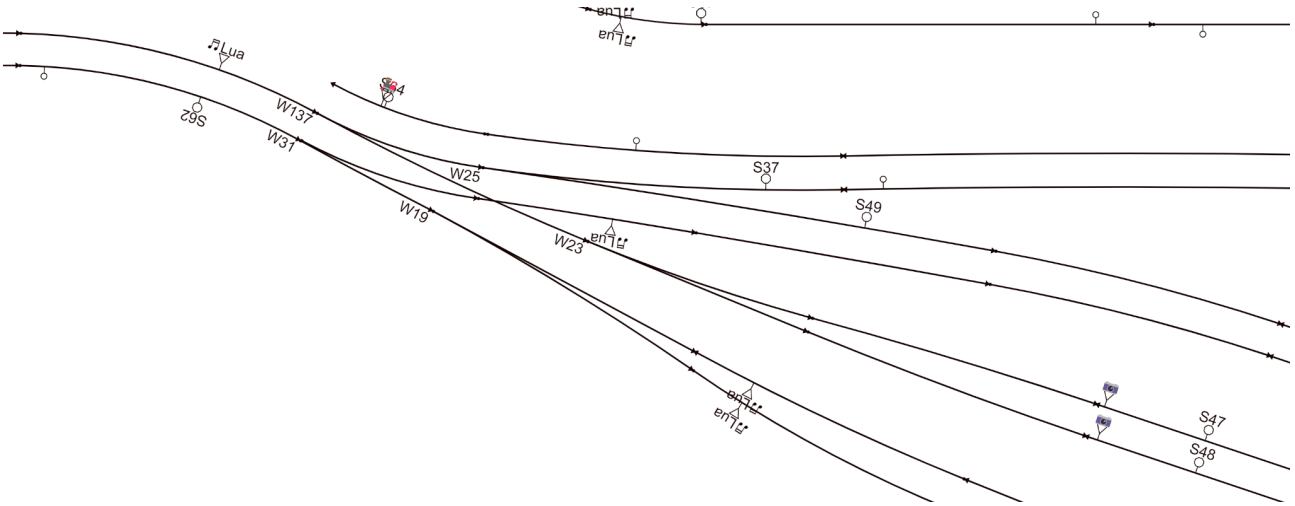
Crossings are not declared in the routes table. This means they are not reserved when Lua selects the route, as blocks and turnouts are. As a consequence, without precautions two trains can drive onto a crossing at the same time. This doesn't give rise to problems, the trains keep driving, there's no derailment … it just doesn't look realistic.

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`. 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 no 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.

Without precautions this is what can happen … two trains run through each other:

Here is an example of a crossing after turnouts 19 and 25 in the ‘Peace River’ Layout:



This crossing is not part of any block, it’s between blocks. This allows an easy solution: manually add a turnout from the other route. This additional turnout will now get reserved too which will prevent trains from entering the crossing.

This leads to the following modified routes for the Peace River layout:

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

How to Let Trains Enter the Next Block Sooner

If a train leaves a block and has to run over e series of turnouts, maybe even with stretches of track between them, it takes a while before it reaches the next block and is finally detected there by the block entry contact, even more so if the contact is set to ‘end of train’.

The block this train has left is already empty for quite a while and a new train could have been allowed into the block already long ago. This can be sped up by placing an extra contact close behind the block signal. This frees the block, such that the next train can already enter. The turnouts stay reserved.

The Lua function to be placed in this contact is[8]:

```
blockControl.leaveBlock_#
```

where # is the number of the block signal.

How to Make Definition of Allowed Blocks Easier

In step 10 Create the Allowed Blocks Tables we saw how we can specify which trains we want to allow in which blocks and how long the wait time will be in the blocks. Most probably we want to deviate from allowing all trains in all blocks and not have any wait time, which is what the Lua code generator creates for us.

There are several ways the allowed blocks can be specified in the trains table. Let’s have a look at the different methods that can be used.

An inline table inside the trains table

```
local trains = {
  { name="#Steam", signal=1, speed=44, allowed={ [9]=1, [19]=1, [26]=30, [27]=30, }, },
}
```

In this example the train #Steam is allowed in blocks 9, 19, 26, 27. In blocks 26 and 27 it has a wait time. The specified 30 seconds is not the net stop time, it includes the driving time from the block

[8] If you are using BetterContacts, then use this Lua function instead:
`blockControl.leaveBlock(Zugname, #)`

entry sensor to the signal. A value of 1 second effectively means the train will drive thru, unless it has to stop because of other traffic.

Blocks that are not mentioned in the table (or which have the value 0) are blocks where this train is NOT allowed, it will never drive into those.

References to tables outside the trains table

```
local CW  = { [10]= 1, [19]=30, }
local CCW = { [ 8]=30, [13]= 1, }

local trains = {
  { name="#Blue",   signal=4, allowed=CW,  speed=40, slot=1, },
  { name="#Orange", signal=5, allowed=CW,  speed=70, slot=2, },
  { name="#Steam",  signal=6, allowed=CCW, speed=50, slot=3, },
}
```

This is a way of working that keeps the trains table easier to read and it also avoids repetition when multiple trains are allowed on the same blocks, like the two CW (clockwise) trains.

References to multiple tables outside the trains table

```
local CW  =   { [10]= 1, [19]=30, }
local CCW =   { [ 8]=30, [13]= 1, }
local Depot = { [34]=30, [35]=30, }

local trains = {
  { name="#Blue",   signal=4, allowed= { CW,  Depot, }, speed=40, },
  { name="#Orange", signal=5, allowed=   CW,            speed=70, },
  { name="#Steam",  signal=6, allowed= { CCW, Depot, }, speed=60, },
}
```

In this example train #Blue is allowed on the blocks mentioned in the table CW as well as the blocks in the table Depot. #Orange is not allowed in the Depot. #Steam is allowed on the CCW blocks and the Depot.

Combine an inline table and a reference to an outside table

```
local CW = { [10]= 1, [19]=30, }

local trains = {
  { name="#Blue",   signal=4, allowed=   CW,                     speed=40, },
  { name="#Orange", signal=5, allowed= { CW, [34]=30, },         speed=70, },
  { name="#Steam",  signal=6, allowed= { CW, [34]=30, [35]=30 }, speed=60, },
}
```

The train #Blue is allowed in the blocks mentioned in the table CW. The train #Orange can additionally also drive to 34. The train #Steam is allowed in CW, 34 and 35.

Combine references to outside tables and add blocks

```
local CW  =   { [10]= 1, [19]=30, }
local Depot = { [34]=30, [35]=30, }

local trains = {
  { name="#Blue", signal=4, allowed= { CW, Depot, [12]=20 }, speed=40, },
}
```

Train #Blue is allowed on the CW- and the Depot blocks, and on block 12.

Combine references to outside tables but remove blocks

```
local CW  =   { [10]= 1, [19]=30, }
local depot = { [34]=30, [35]=30, [36]=30, [37]=30,}

local trains = {
  { name="#Blue", signal=4, allowed= { CW, Depot, [37]=0 }, speed=40, },
}
```

Train #Blue is allowed on the CW- as well as the Depot blocks, however ... NOT on block 37. Blocks having the value 0 will be excluded.

Random Wait Times

Trains can be given a scheduled stop in a block. This is done by stating a wait time in seconds:

```
local CCW = { [12] = 40, [13] = 1, }
```

In block 12 the train will have a delay of 40 seconds. This includes the driving time from the block entry contact to the signal.

A value of 1 second effectively means the train will drive thru, unless other traffic forces it to stop.

A value of 0 means the train is not allowed in this block.

To avoid predictability trains can be given a random wait time by specifying a min and max value:

```
local CCW = { [12]={20,40}, [13]=1, }
```

In block 12 the train will have a random delay anywhere between 20 to 40 seconds.

If we like to allocate this same random time of 20 - 40 to multiple trains and / or multiple blocks, the following construction can be used to avoid endless repetition of similar values in tables:

```
local T1 = { 20,80 } -- Random Wait Time 1
local T2 = { 15,30 } -- Random Wait Time 2

local Depot = { [34]=T1, [35]=T1, [36]=T1, [37]=T2, }
local CCW   = { [12]=T2, [13]=1, }

local trains = {
  { name="#Blue", signal=4, allowed= { CW, depot, [37]=0 }, speed=40, },
}
```

Options

The code that follows the configuration section is identical for every layout and can often be left untouched, unless signals are in use on the layout that have different red / green states, or if you like to change some of the available options.

Clear the Event Window

```
clearlog()
```

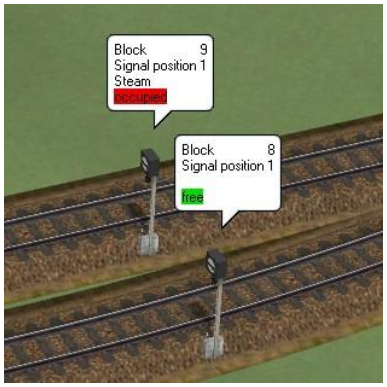
This clears the EEP Event Screen after startup and after a ‘Reload script’. If you don’t want to clear the screen, this line can be deleted or turned into a comment: `-- clearlog()`.

Change Signal States

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

NOTE: all block signals used on the layout need to have identical states. The same holds for all train signals, their states need to be identical, although they may differ from the block signals.

If you want to show other signals having a different signal pattern you can connect them to invisible block signals. Take care to exclude such signals from the generation process.



Signal states are shown in the popups that show when a layout is first started as “Signal position #”. They can easily be read out and the Lua code can be changed accordingly if needed. This is done in the code lines that read[9]:

```
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 the pop ups on or off, toggle the main switch twice quickly via Shift + left click.

Select language

The module shows status messages in the event window either in German, English or French according to the language with which EEP is installed. The desired language can also be set with a parameter via the `init` or `set` functions:

```
language        = "ENG",     -- GER: German, ENG: English, FRA: French
```

Toggle Parameters on / off

The following parameters can be switched on or off:

```
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
```

Activate the Control Desk by Default

```
-- Optional: Activate a control desk for the EEP layout
```

[9] You do not need to define signal states if the default matches the default values as described.

```
-- 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. Most 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.

Use a counter signal to set the log level

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 in demo `EEP_LUA_ATC_Model_Railway_Layout_1`.

The associated Lua code for this signal 47 is as follows:

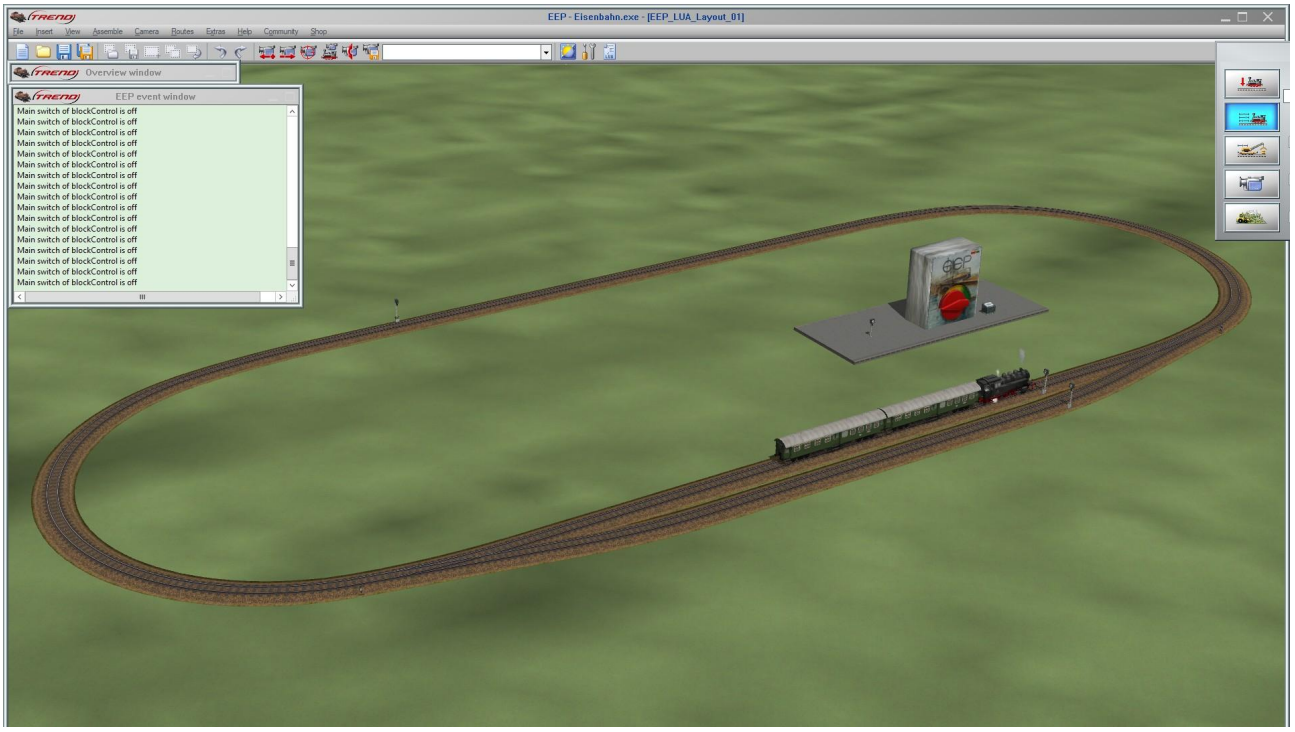
```
-- Optional: Use a counter signal to set the log level
local counterSignal = 47
blockControl.set({ logLevel = EEPGetSignal( counterSignal ) - 1 })
EEPRegisterSignal( counterSignal )
_ENV["EEPOnSignal_"..counterSignal] = function(pos)
  local logLevel = pos - 1
  if logLevel > 3 then
    logLevel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

Demo Layouts

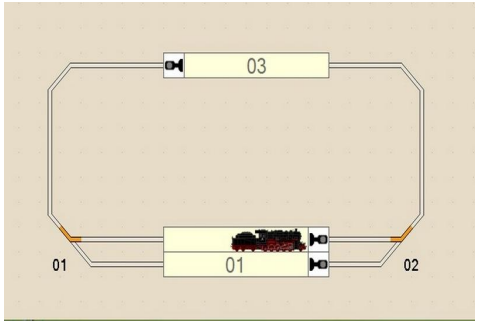
If you've installed the demos that came with the package, you should have this folder in your EEP installation `Resourcen\Anlagen\EEP_blockControl` which contains a collection of demo layouts.

The demo layouts are described in detail in the following chapters.

EEP_LUA_ATC_Demo_1

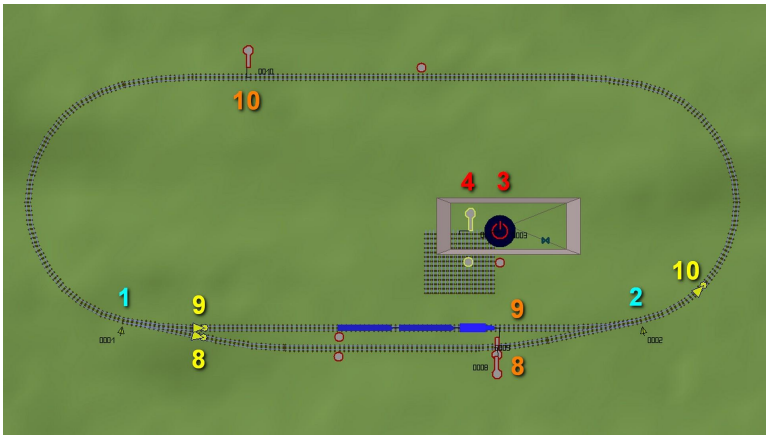


Let's write the Lua configuration for this simple layout and have the train drive around fully automatic, with a waiting time at the station.



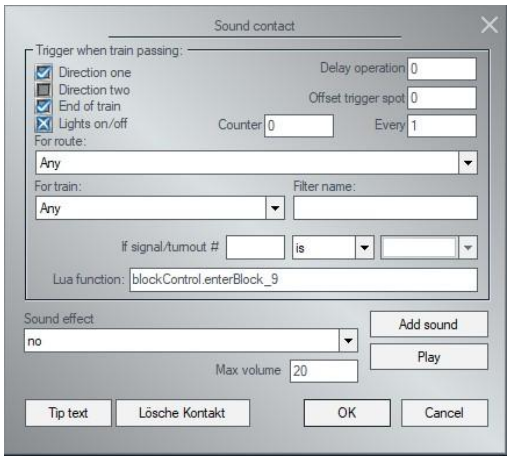
With automatic traffic, trains drive from block to block. The first decision to make is which blocks we want to define. There's only one rule here: **turnouts can not be part of a block, they are part of the routes between blocks.**

For this layout it seems logical to create 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. A contact is placed at the beginning of the block.

The picture shows the turnouts (cyan), the block signals (orange), the block entry contacts (yellow), and the on / off switches (red) for this layout. If you create a similar layout your numbers may differ, as they are automatically generated by EEP.



Block entry contacts can be of any type. Yellow ‘Sound’ contacts have good visibility and are easy to use. In the Lua function field this function must be entered:

`blockControl.enterBlock_#`[10].

where # is the number of the block signal. This contact lets Lua know when a train has arrived in the block.

In most cases ‘End of the train’ is the safe choice. In this case previous blocks and turnouts are released once the final wagon has passed the contact. If there’s no opposing traffic that needs to use the turnouts, triggering on the head of the train will release the turnouts and the previous block a train length sooner, which can speed up traffic.

The configuration code for this layout 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 to describe the 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 changes if the

[10] Keep in mind that you have to work in a specific order:
1. Adjust the Lua code to list all block signals
2. Run the layout in 3D mode
3. Place the contacts if not already done
4. 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”.

states of the signals that are used differ (see chapter Change Signal States) or if you like to change some of the available options (see chapter Toggle Parameters on / off).

Explanation of the configuration code:

```
main_switch = 3
```
The ID number of the signal that functions as the main on / off switch.

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

The components have the following meaning::

```
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 chapter 4 Place Trains in 3D Mode.

```
signal = 4
```
Every train could have its own individual start / stop signal. This ID number is given here[11].

```
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[12] 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 contact 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, run the layout and have a look at the event window. The drive time and stop time are measured and are shown there. The times can be modified in the table now to create a specific stop 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 = {
  { 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[13] when driving from one block to the next[14]. In this example there's exactly one turnout between the blocks of all routes. There could also be none[15], or there could be multiple turnouts between blocks. Examples:

- `{13,14, turn={} },` Blocks 13 and 14 don't have a turnout between them

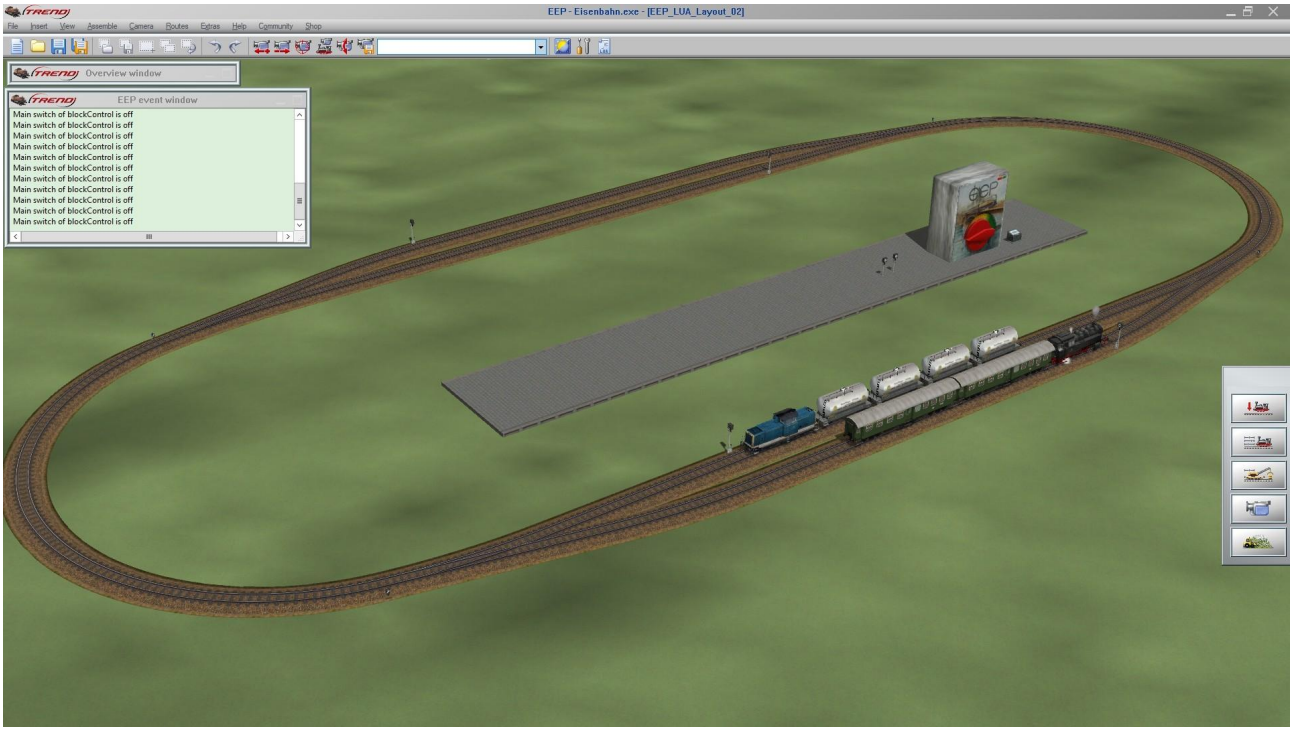
[11] You can reuse the same signal for multiple trains which you want to start or stop together.
[12] …or the block has value `nil` or 0
[13] The exact value of the position for a turnout does not matter if the train cuts the turnout open, however, it's good practice to provide correct values in any case.
[14] 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 }`
[15] If there is no turnout you can define an empty list for parameter `turn` or you can omit this parameter.

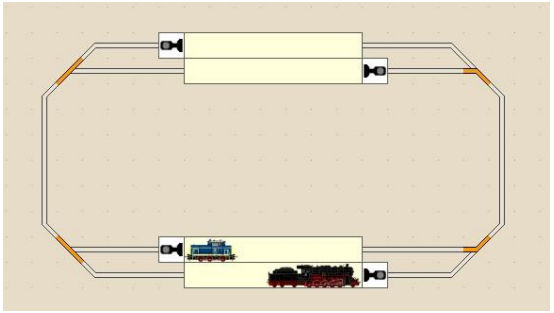
- `{23,24, turn={14,1, 5,2, 6,1} },` Blocks 23 and 24 have three turnouts between them.

Follow this link for a YouTube video on Demo 1.

EEP_LUA_ATC_Demo_2

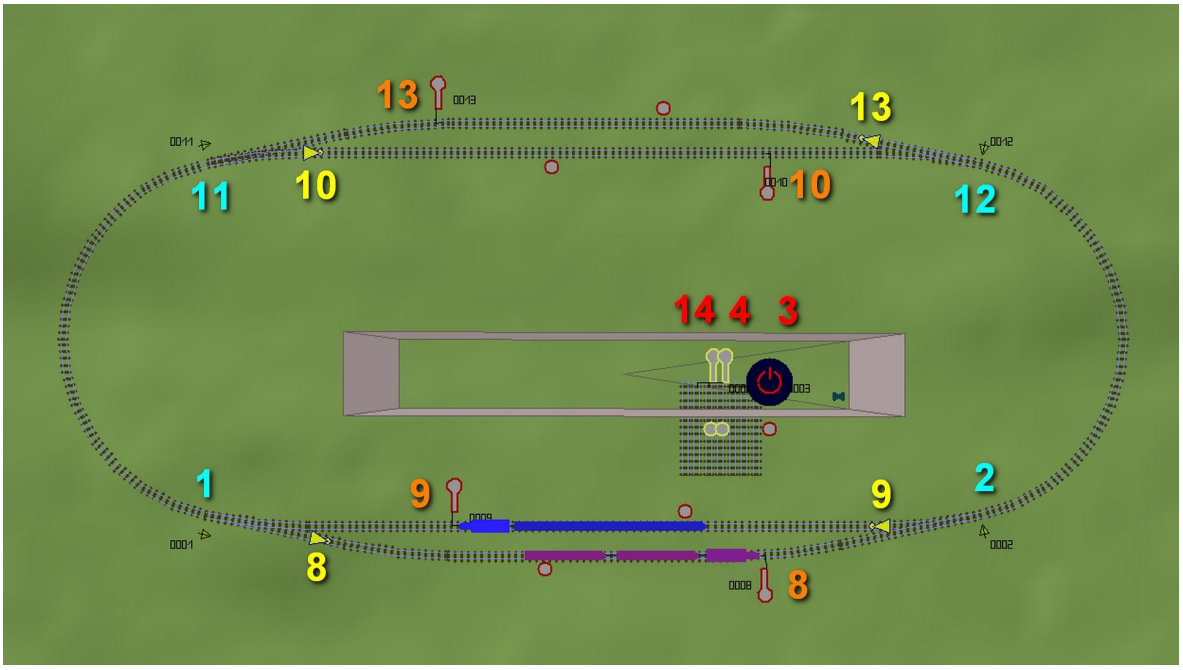
Let's add a block to create 'Station North', place a second train and have traffic in two directions.





For this layout we define 4 blocks, 2 at station North and 2 at station South. We'll drive right handed. Even though trains drive in opposite directions, the traffic in the blocks is in one direction. We'll see how to create blocks with traffic in two directions in demo 3.

The 2D overview below shows the turnouts (cyan), the block signals (orange), the contacts (yellow) 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 for this layout is:

```
local main_signal = 3

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

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

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 }},
}
```

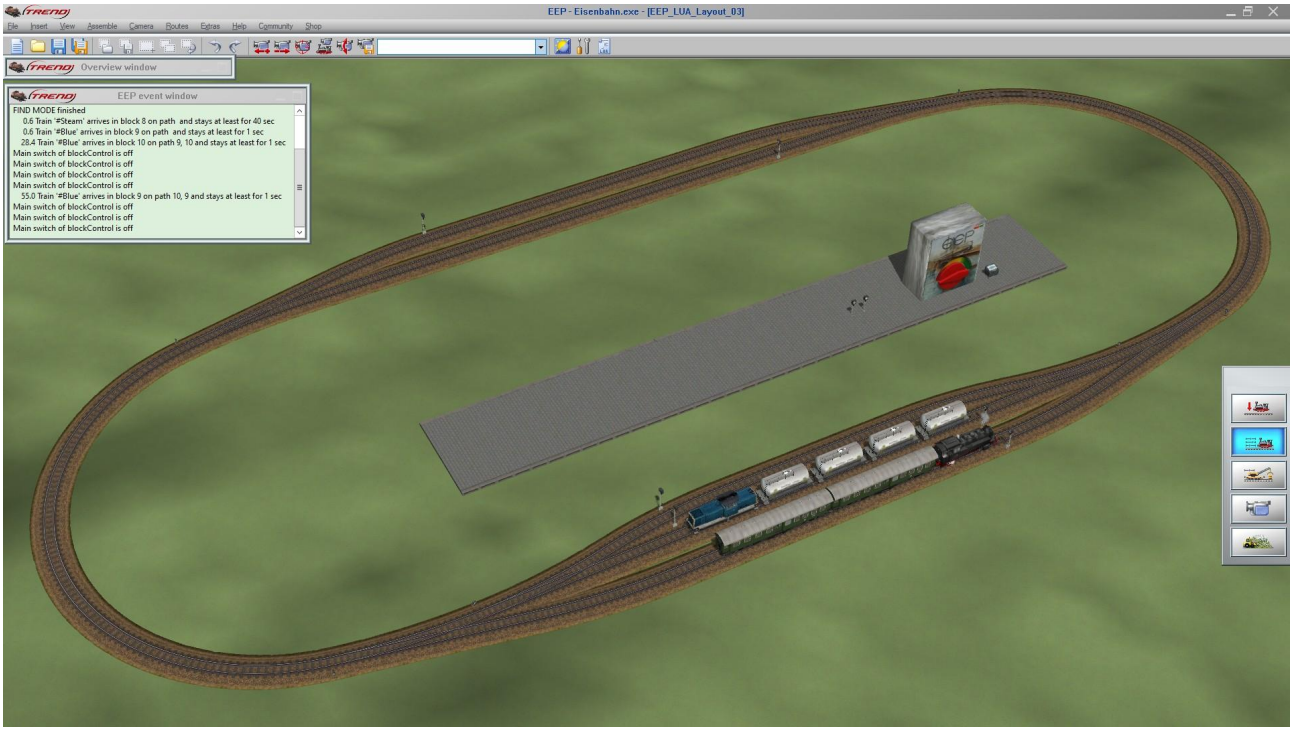
In this example the allowed blocks and the wait times are specified in separate tables, one for ‘counterclockwise’ trains, one for ‘clockwise’ trains. These are just names, you can use any name, ‘cargo_trains’, ‘ICE’, as long as the same names are used in the `trains` table.

In this layout with just two trains, the benefit of specifying allowed blocks is small, but on a larger layout that may have multiple trains allowed on the same blocks, these ‘allowed blocks’ tables avoid endless repetition of identical tables for every train.

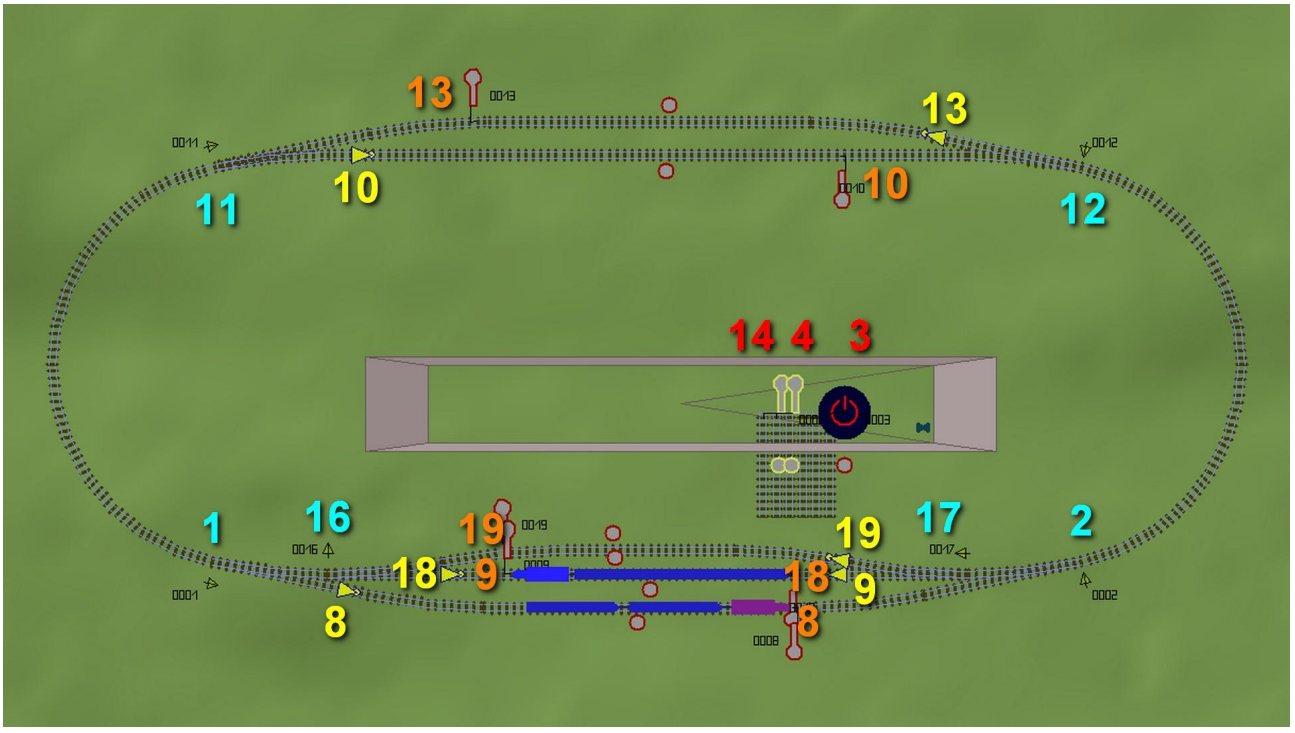
Follow this link for a YouTube video on Demo 2.

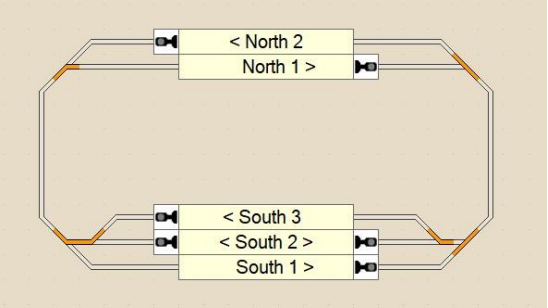
EEP_LUA_ATC_Demo_3

Station South is extended with a middle track, on which we plan to allow trains to drive in both directions. Let's see how to configure the Lua tables to make this possible.



First 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 contact at both 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'll have to tell Lua about these two way twin blocks. This is done via the `two_way_blocks` table.

The configuration code for this layout is:

```
local main_signal = 3

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

local CW= {   -- allowed blocks for clockwise 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 = CCW, },
  { name = "Blue",  signal =  4, allowed = CW,  },
}

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[16]:

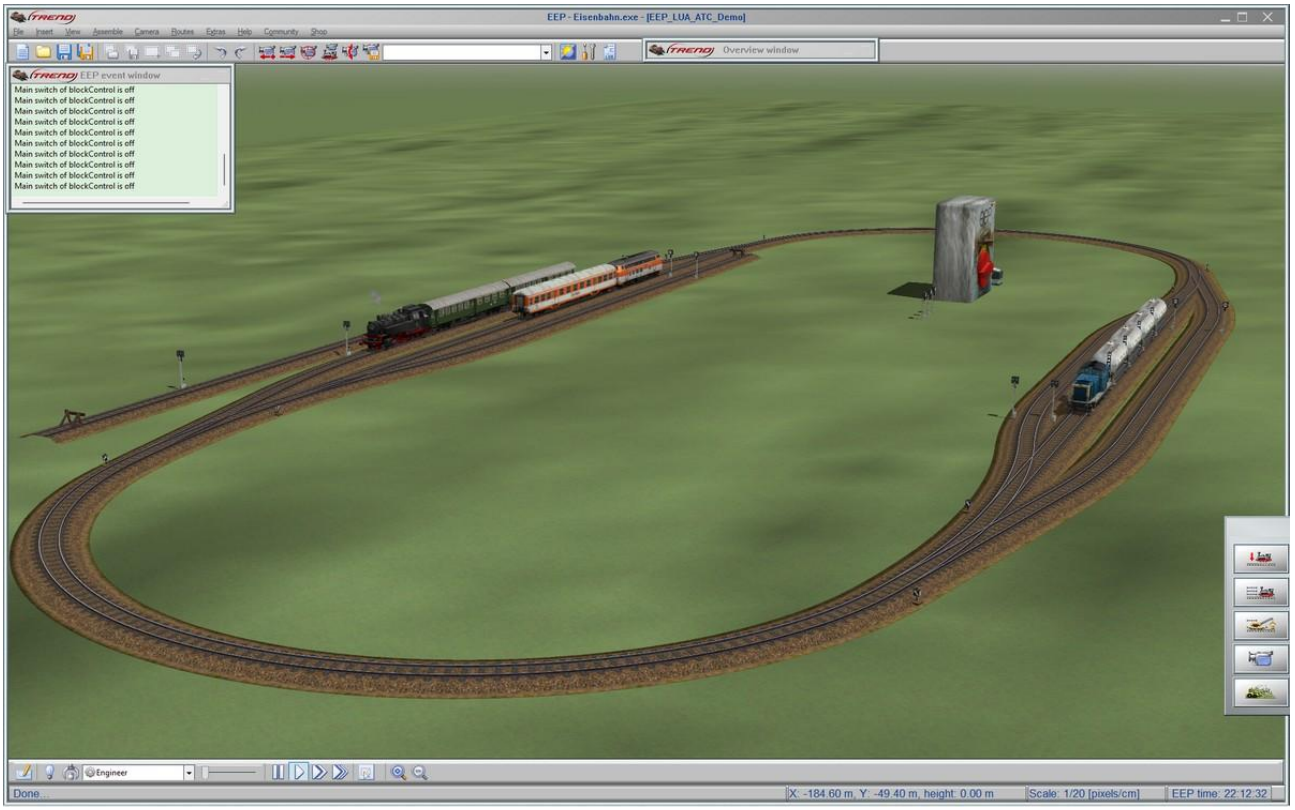
```
local two_way_blocks = { {82, 81}, {32, 33}, {74, 38}, }
```

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.

Follow this link for a YouTube video on Demo 3.

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

EEP_LUA_ATC_Demo_4



This is the layout that has been used throughout this User Manual in chapters 1 - 13, to demonstrate how to auto-generate the Lua code and how to create allowed blocks tables.

The layout contains two dead ends and a two-way track to demonstrate how these are configured in the Lua tables that describe the layout.

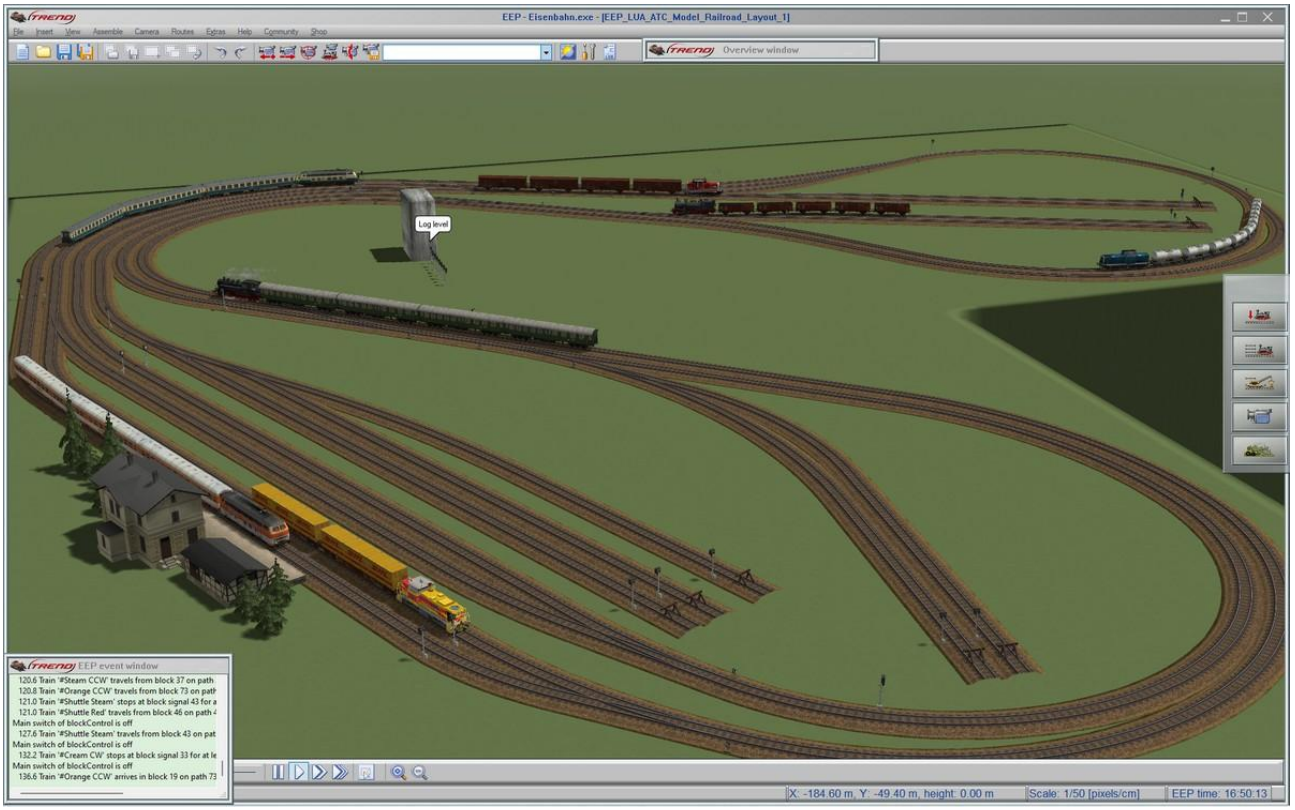
Three trains drive around on this layout.

Follow these links for a YouTube video 4A on Demo 4 and -YouTube video 4B on Demo 4.

This YouTube video 7 demonstrates all steps as covered in this User Manual, including train reversal without using contacts for the reversal.

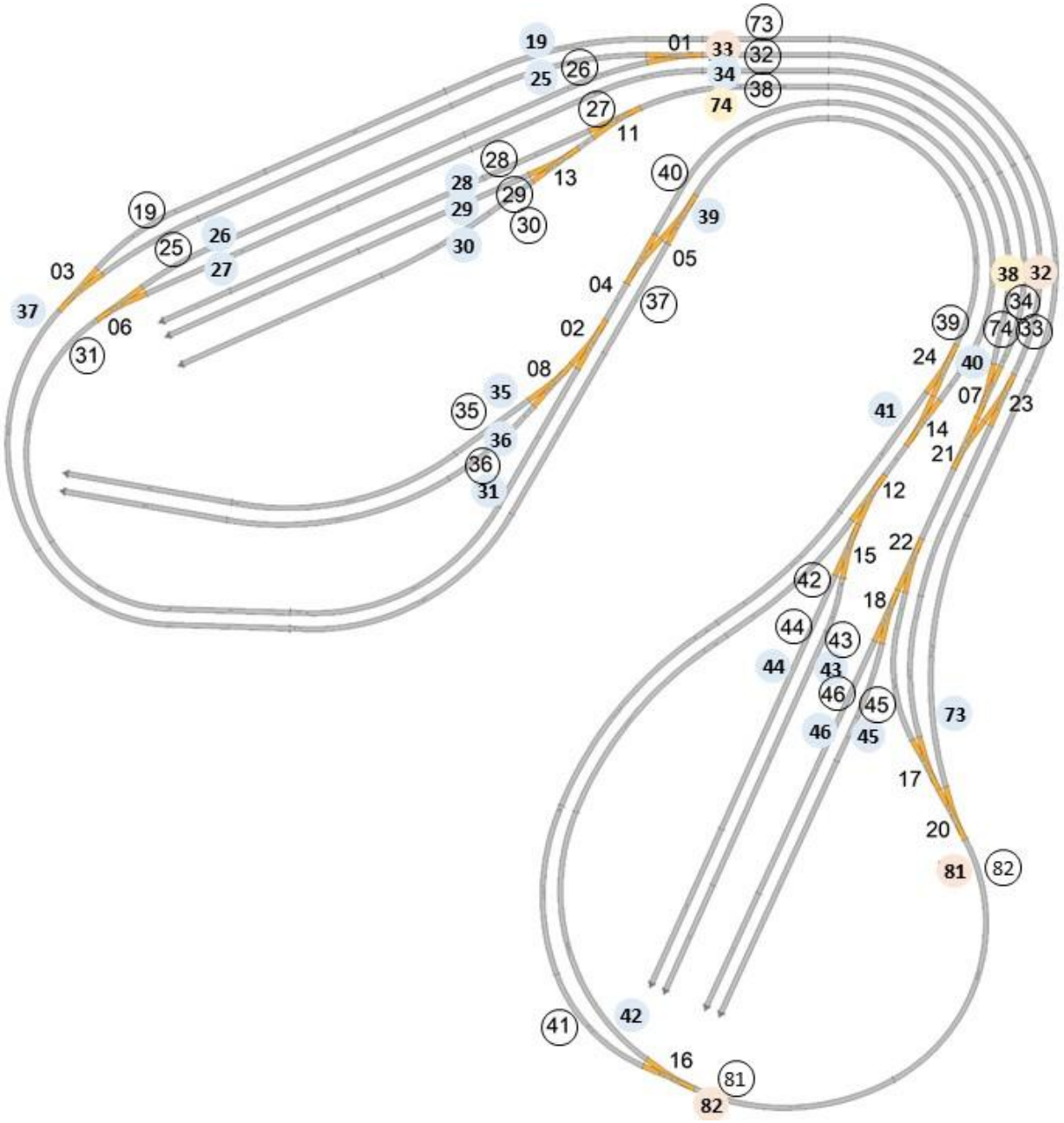
This YouTube video 8 shows how to reverse trains in any block, not necessarily a dead end.

EEP_LUA_ATC_Model_Railway_Layout_1

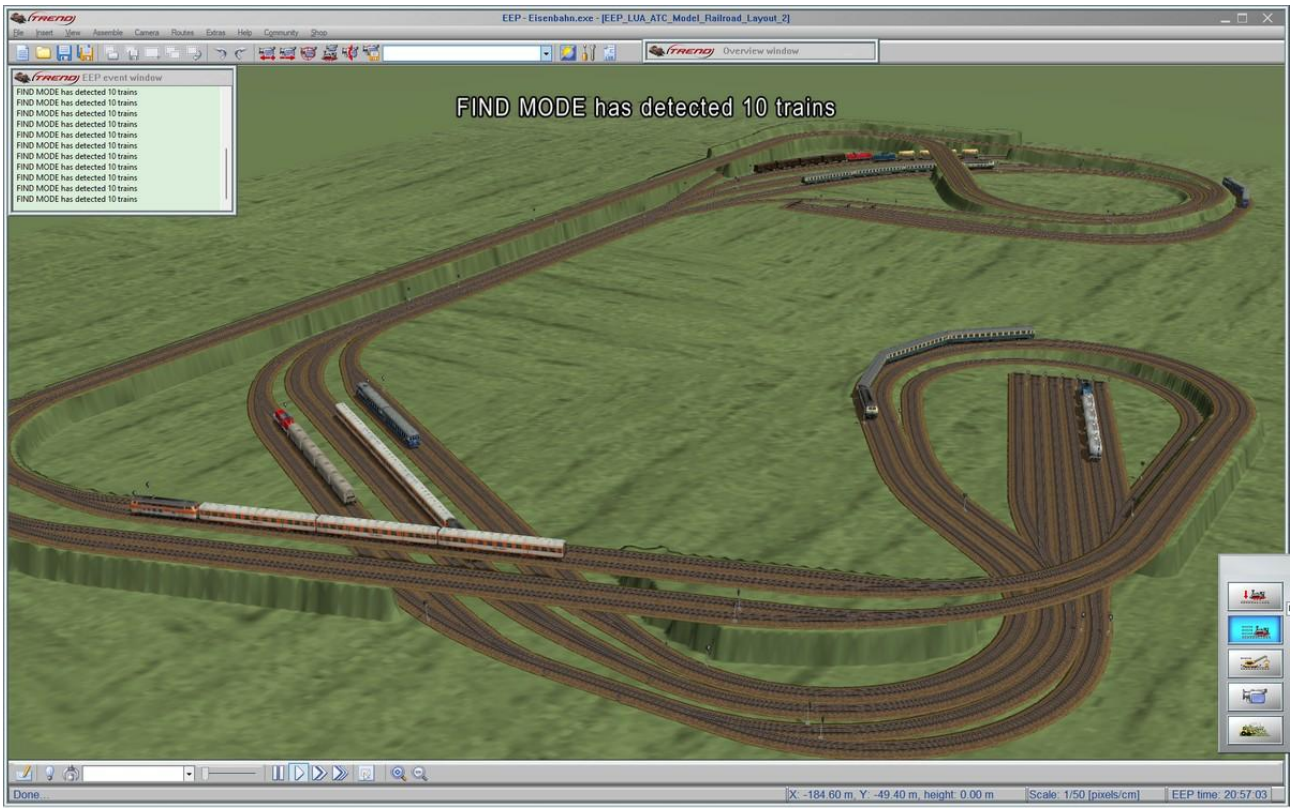


This is an EEP simulation of an existing model railway layout on which seven trains drive around.

Follow this link for a YouTube video 5 on Model Railway Layout 1 .


73
19
01
33
32
26
25
34
38
27
74
11
28
40
28
13
29
39
19
29
30
25
26
05
03
30
27
04
37
38
32
37
06
02
34
39
74
33
31
08
24
40
35
07
23
35
41
14
36
21
36
12
31
15
22
42
18
44
43
44
43
46
45
46
45
73
17
20
81
82
41
42
16
81
82

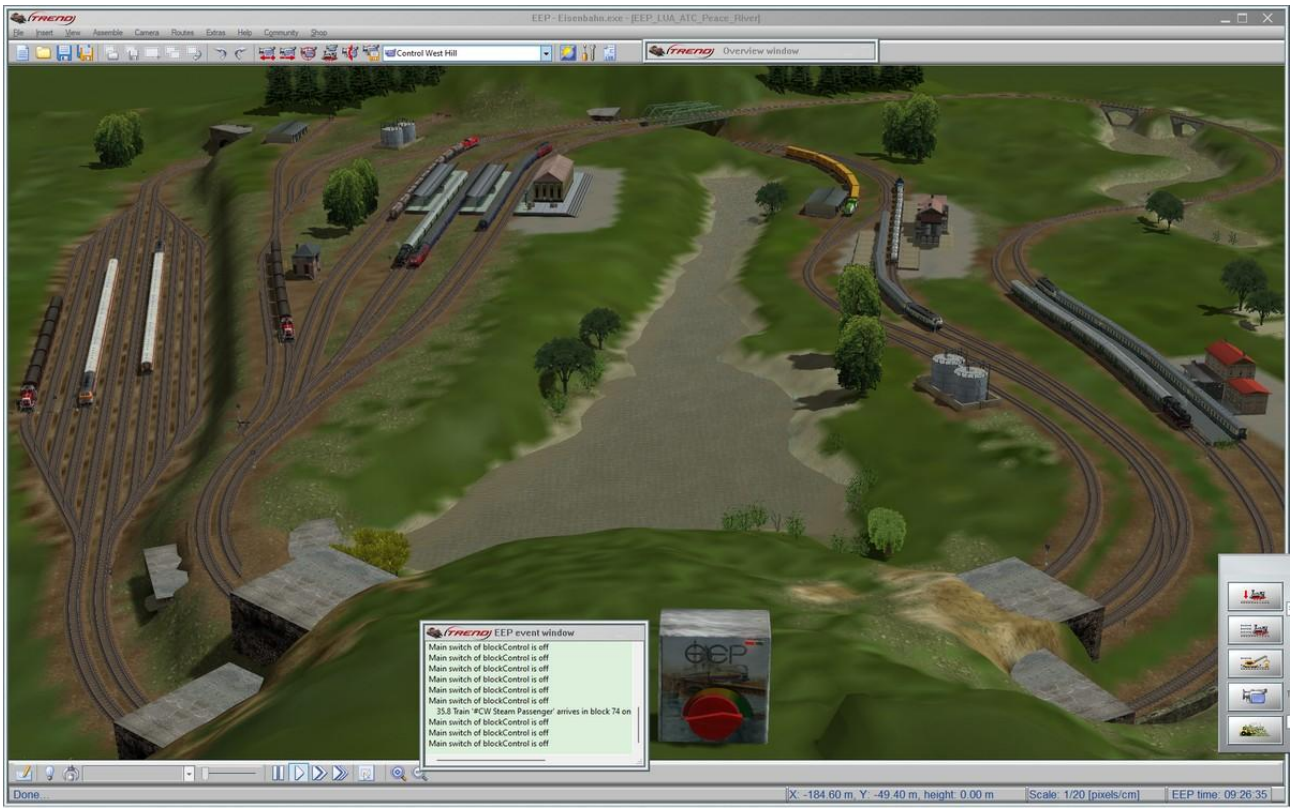
EEP_LUA_ATC_Model_Railway_Layout_2



This also is a simulation of an existing model railway layout, on which ten trains drive around.

Follow this link for a YouTube video 6 on Model Railway Layout 2.

EEP_LUA_ATC_Peace_River



This EEP layout is roughly based on a model railway design found on the internet. Thirteen trains drive around on this layout.

Follow this link for a [YouTube video 10 on Model Railway Peace River](#).

You find an extended demo as well, which demonstrates how you could adjust and extend a layout: The original depot uses 2*5 tracks to manage trains in separate directions. Why not use all 10 tracks for managing trains in both directions?

For the extended demo we added 4 turnouts to create a crossing and 5 block signals and contacts to build two-way-blocks on both sides of the depot and regenerated the Lua code. Then we extended the "allowed" tables for trains to use the new blocks and added extra turnouts to routes to protect both crossings. Works fine!

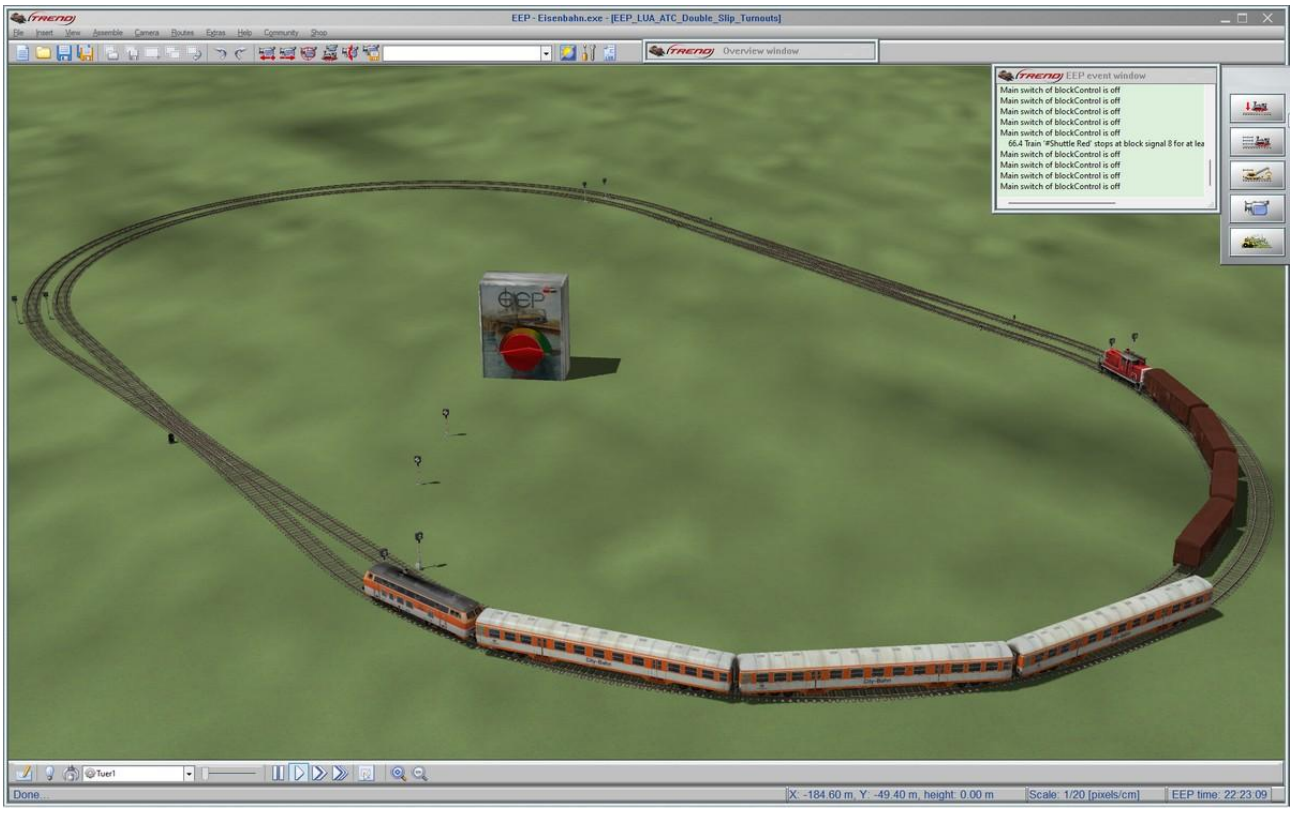
In addition we allowed the train "#North Red Cargo" to leave the shuttle station and to take a complete CCW run. Because of this we added a similar train "#North Blue Cargo" to have more traffic on the shuttle station.

EEP_LUA_ATC_Double_Slip_Turnouts

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

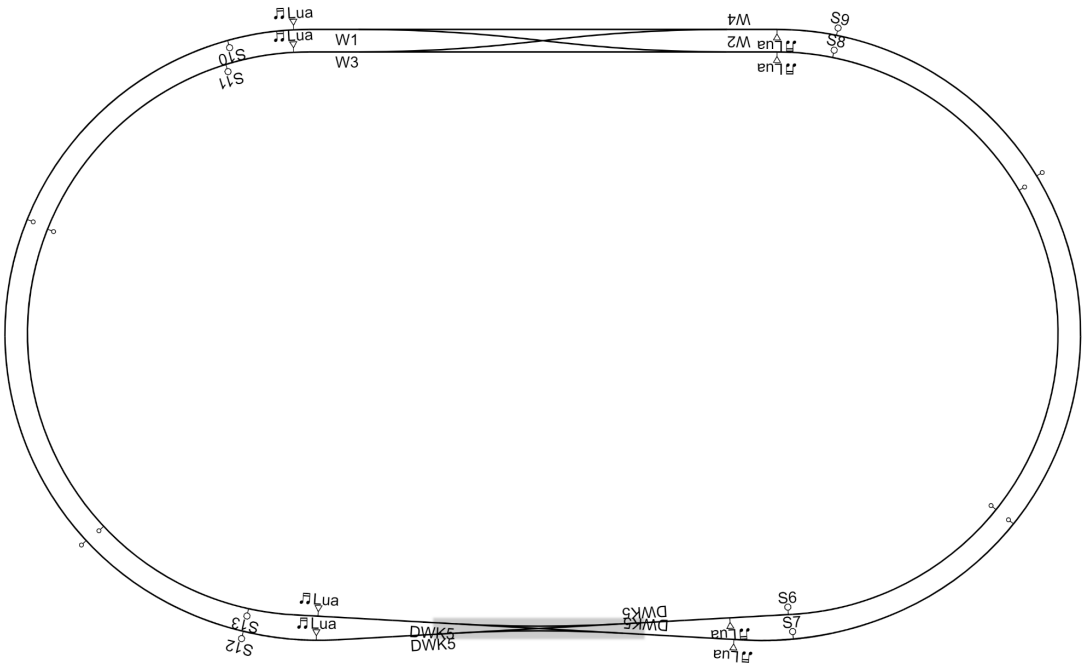
a) You can create a DST constinsing 4 turnouts manually or using the templates in EEP. You should secure the crossing of that DST by extending the direct routes with a turnout from the other part of the DST.
b) You can use a track object DST consisting of 1 turnout having 4 positions. You can define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

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



This Lua code of this layout demonstrates the different ways to build double slip turnouts / crossings and how the crossings can be protected to avoid collisions. This is described in chapter How to Prevent Collisions on Crossings.

Here is the corresponding picture from the EEP Layout Tool:



The configuration code 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
}
```

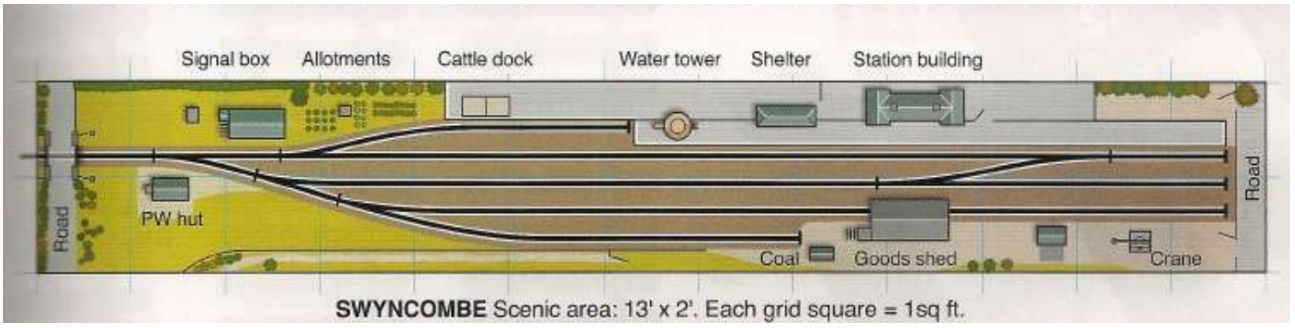
All trains have different velocities and can go everywhere with different wait times. This way different encounters occur showing passing and waiting situations..

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.

EEP_LUA_ATC_Swyncombe

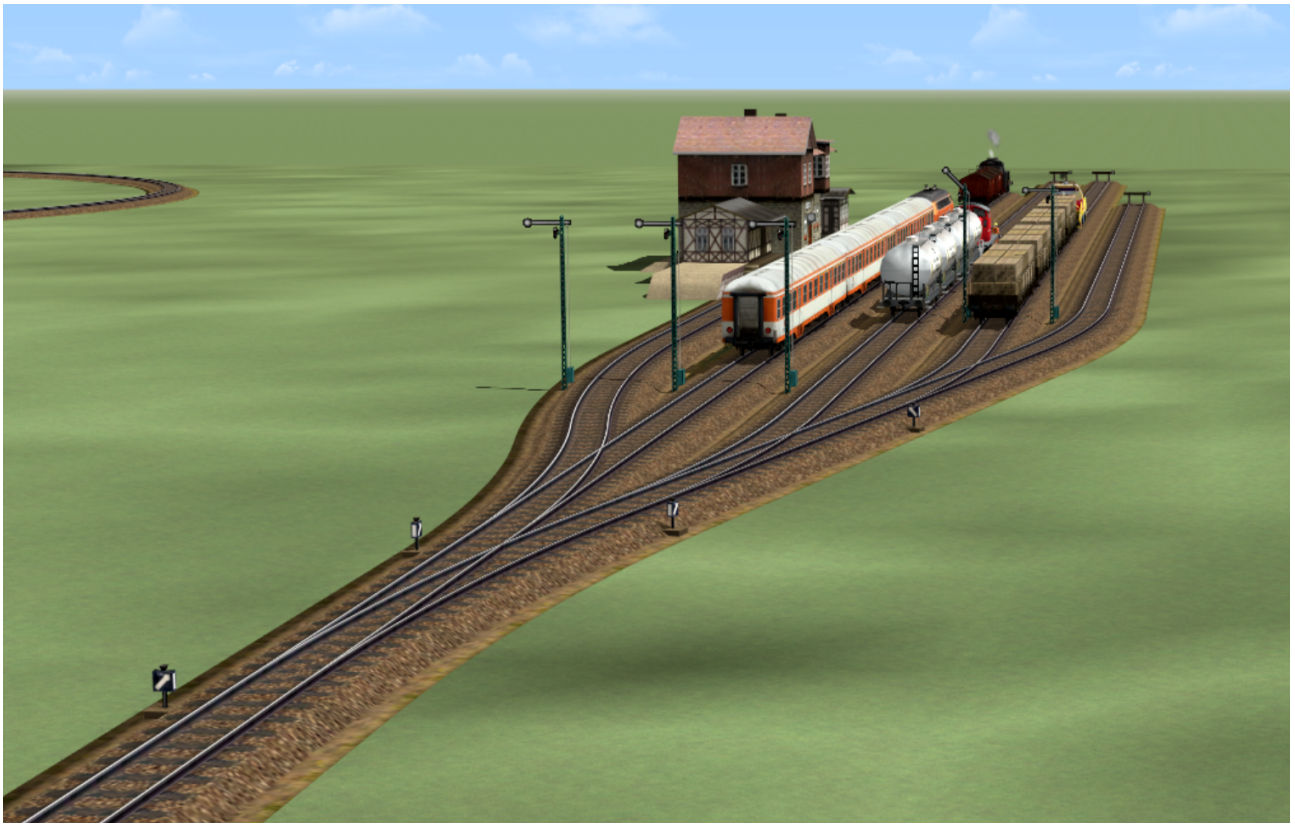


Swyncombe is a model railway ‘bookshelf’ layout described in the British Railway Modelling magazine, June 2013. It’s a dead end station with only a single track in/out[17].

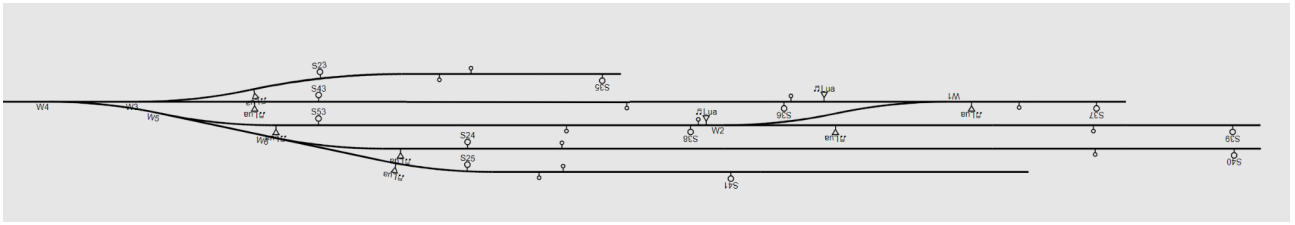
A 7-track depot was added where trains can reverse and drive back to the station, or they can drive on into a loop that effectively turns them around before they arrive back at the station.

With 12 trains on the layout, that all can drive into the station head or tail first, a maximum of variation is guaranteed.

[17] Source: Phil Gliebe, Designing Small Shelf Layouts for Operations
http://thoroughbredlimited2015.yolasite.com/resources/Clinic_Presentations/Gliebe%20Designing%20small%20shelf%20Layouts%20for%20operating%20fun%202015-2.pdf
Pictures of the beautiful scenery:
http://www.gwr.org.uk/layoutsswyncombe1.html



Here is the corresponding picture from the EEP Layout Tool:



Short trains can go everywhere in the station, mid length trains and long trains have less options. All trains have full access to the depot. To accommodate this we define 4 tables with allowed blocks: short, mid, long, depot.

All depot tracks have a route back to the station in both directions, one with `reverse=true`, one moving forward. Random wait times are used by defining `rt={10,40}`, which we can now use in the allowed blocks tables.

```
local rt = { 10, 40 }  -- Random (minimum) waiting time for the depot

local depot = {
  [44]=rt, [45]=rt, [46]=rt, [47]=rt, [48]=rt, [49]=rt, [50]=rt, -- depot
  [42]=1,  [51]=20, [52]=20,                                  -- access to depot
}
local short = {
  [35]=20, [36]=1, [37]=120, [38]=20, [39]=120, [41]=20, [43]=1, [53]=1, }
local mid   = {
  [35]=20, [36]=20,          [38]=20,           [41]=20,                 }
local long  = {
```

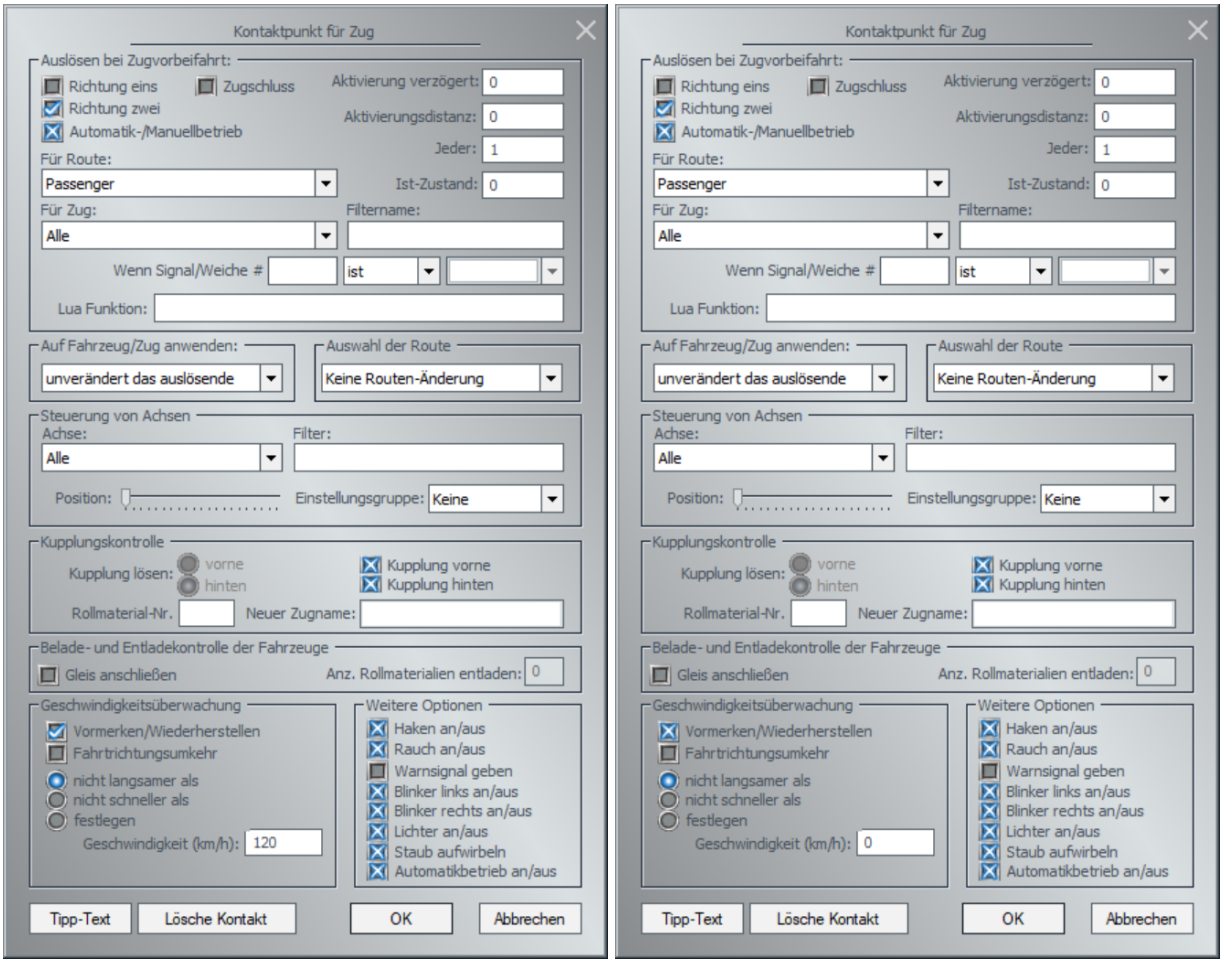
```
            [36]=20,          [38]=20,          [40]=20,
}

local trains = {
  { name = "#Long1",  speed = 30, allowed = { long,  depot, } },
  { name = "#Long2",  speed = 32, allowed = { long,  depot, } },
  { name = "#Mid1",   speed = 30, allowed = { mid,   depot, } },
  { name = "#Mid2",   speed = 32, allowed = { mid,   depot, } },
  { name = "#Short1", speed = 30, allowed = { short, depot, } },
  { name = "#Short2", speed = 32, allowed = { short, depot, } },
}
```

The demo shows a sample solution about speed changes too. On one side, trains have their specific cruise speed - slow for cargo trains, fast for passenger trains - and on the other hand there exist restrictions in stations and on turnout sections about a maximum speed. In an EEP layout you define such speed changes using train contacts, and this works well together with the definition of a speed in the trains table which is used whenever a train reverses its direction.

The demo shows rather small speeds in the trains table to match the restrictions for stations. In addition, train contacts on the long path from the depot to the station increase respective restore the speed of passenger trains using the corresponding checkbox settings:

Tipp text: “Read out and save the current speed of the triggering vehicle or restore the memorised speed. If the box is checked (✓), the current speed is memorised. If the box is marked with an X, the saved speed is restored. A grey box means ’no action’.”



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.

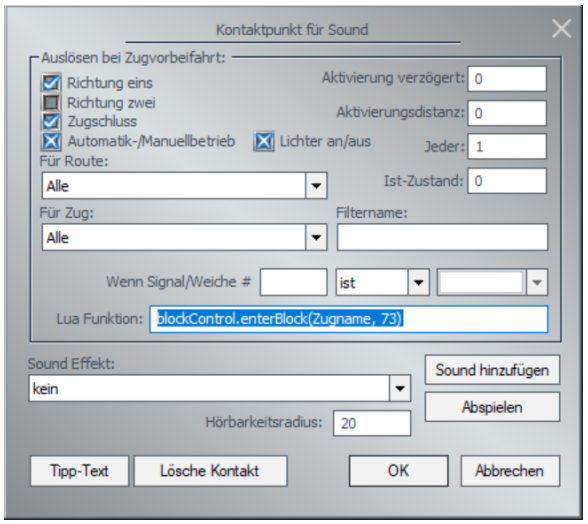
You can avoid this restriction by using the module “BetterContacts” from Benny, which you can get from here: https://emaps-eep.de/lua/bettercontacts

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`)[18]:

```
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)[19]:




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

[18] In addition you can use the option `BetterContacts = true`, in the `init` call of module `blockControl` to prevent generating global functions for this module.
[19] 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”.[20]
    d. Release any corresponding two way block.
    e. Release the turnouts on the path behind the train.
    f. Continue travelling 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 travelling on this path:
    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.
    d. Set the train speed for the opposite direction if the first part of the route is a reversing route.

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

[20] 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`

as well. Use this to verify if this value matches the parameter `BLKSIGRED`.

- Parameter `blockSignals`
  Use this optional table for documentation purpose and to trigger additional consistency checks:
  - Do all blocks used in tables `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

Typical issues when adding blockControl to existing layouts

Let's assume you're working on non-trivial layouts and you are converting the given control system of an existing layout into automated blockControl. That works fine in general but takes some time to configure. You have to select and define block signals carefully. It is possible to have additional signals (which are ignored by blockControl), e.g. for display purpose or to support a road crossing, but this requires caution: the automated blockControl assumes that it controls all block signals and related turnouts completely. No other process or no user interaction is allowed to touch them.

Let's have a look to some special cases:

Issue "Turnout between approaching signal and main signal"

You may find signals with turnouts being located between the approaching signal and the main signal. Depending on the direction of trains running over this turnout this could be possible and useful and it could be possible that you can define the required route definition manually. However, the generation program cannot handle such a complicated case. Remember the golden rule from the beginning of this document: *"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."*

Issue "Two signals on same track"

You may find two signals on the same track. The generation program does not deal with the exact positions of signals on tracks but uses only the connection between tracks to find routes between signals. This simplifies the calculation but runs into trouble if two signals are located on the same track. Most likely serve both signals different purposes and only one of them is a real block signal controlling automated traffic. You can put the other signal onto the "ignored list" when generating the Lua code.

Issue "Dead end without block signal"

You may find dead-ends behind turnouts that do not have block signals yet. The blockControl generation program requires a block signal on each dead end. The reason for this is that the program follows tracks and reflects the direction of searching at dead ends. Without a block signal, the same turnout is visited twice while searching for a route into and out of a dead end. This is interpreted as a forbidden cycle.

Potential issue "Too many two way blocks"

You have defined many two way blocks. That's possible and works fine in the first place. It allows trains to go everywhere in any direction. However, this increases the risk of lockdown situations especially if there are many trains running which force you to define a long list of anti deadlock paths manually. We suggest starting with only required two way blocks and extending it later carefully.

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
```

… THE END …