Scenario scripting guide

For Trainsimulator 201x

Rudolf Heijink

Version 0.2, November 2014

Preface

I proudly present the second alpha version of my scenario scripting guide. Il spent a lot of time collecting all information needed, and it is far from done. This is as far as I got until now.

I hope it will encourage you to start crating scripts and experiment with it. I also hope you are willing to share your findings with me and with the TS community.

Contact me at rudolfjan@klantprocessenboek.nl if you have additional information or corrections.

Do not expect me to work as a helpdesk for scripting questions. All I know is in this guide.

You can use the guide for personal use, but it is not allowed to copy it or redistribute it in any form.

Acknowledgements

Tankski has started a google document that describes some of this information in more detail. It can be found here: https://docs.google.com/document/d/19gn ... ring&pli=1
VTrian for the cinematic camera tutorial

Richard Scott for the beautiful GWR Railcar on the front page.

Chris Longhurst for his research on playing sounds and the function with respect to signalling. (And of course for his beautiful Dutch train models!).

Bob Artim and Peter Hayes for their list of startup parameters

All members of the TS2014 community that somehow shared information that helps to create this guide.

New in this version

1. Texts restructured at some places.
2. New information on debugging.
3. New information on message function.
4. A fair number of code examples added.
5. Some information on advanced lua scripting techniques added.
6. Overview of statup commands added.

Rudolf Heijink

Contents

1 Introduction

TS2014 makes it easier to add scripting functions to scenarios. In fact, it already was possible, but a bit hidden in previous versions. The new functions are still undocumented. I did not want to wait till proper documentation is available, so I decided to create a guide for scripting.

It is still incomplete and maybe not always correct, so I hope you will contribute to new versions of this guide by providing me your tips and corrections.

Please contact me by mail: rudolfjan@klantprocessenboek.nl

This guide will be available on Steam, UKTS, Railworks America, Treinpunt and Dutchrail Europe.

2 Getting started

2.1 Knowledge prerequisites

It will help you a lot if you have some basic knowledge before you start to attempt using this guide:

I assume you know how to create scenarios and that you can use the different scenario event types. If not, I suggest you take some time to learn how to create scenarios.

Some general knowledge on programming is very helpful, e.g. Visual Basic, or programming languages like Pascal or C. Make sure you know what is meant by following concepts:

- Variable
- If … then … else
- Function
- Syntax
-
-

Since LUA is used as a scripting language, knowing LUA may help as well.

A book to learn LUA (I recommend to buy this book and study it carefully)

http://www.lua.org/pil/

A tutorial for experienced programmers:

http://lua-users.org/wiki/TutorialDirectory

The tutorial is part of a large wiki, which provides many solutions and examples for common problems.

http://lua-users.org/

A LUA reference manual online:

http://www.lua.org/manual/5.1/

You also need some HTML knowledge. HTML is mark-up programming language for internet pages.

http://www.htmldog.com/guides/html/beginner/

HTML is mainly used to make fancy dialogs, but you can include things like playing sounds, showing images etc. I will give some examples later in this guide.

Some useful tutorials at Railworks America:

- Creating fancy lua supported dialogs
- Adding Cinematics

2.2 Lua version

TS2014 uses lua version 5.02 You should be aware of this, because it is not the latest version of lua.

2.3 Tools

It helps a lot if you install SciTE on your local system

It is available at http://www.scintilla.org/SciTEDownload.html

You also may want to use Microsoft XML Notepad (http://www.microsoft.com/en-us/download/details.aspx?id=7973), which helps you to view and edit XML files.

http://www.pspad.com/ is a general code editor, which can be used for HTML code. It can interface to your browser, to get a preview of the edited code.

If you want to use HTML display messages, you need at least some understanding of HTML code. Several online introductions are available, e.g.:

http://www.w3schools.com/html/

It may be useful to have hex editor available:

http://mh-nexus.de/en/hxd/

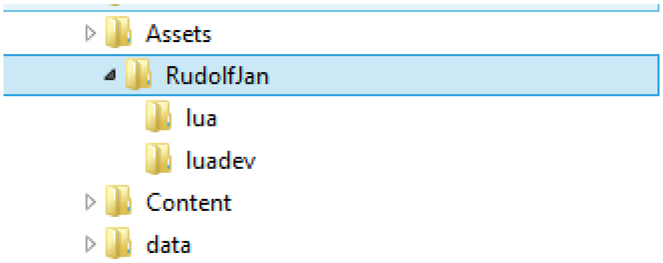
(This one seems to work. Be careful, some “free” tools are not free at all).

2.4 Development environment

One way to create scenario scripts is just create a separate independent script for each individual scenario. This method has serious disadvantages for maintenance:

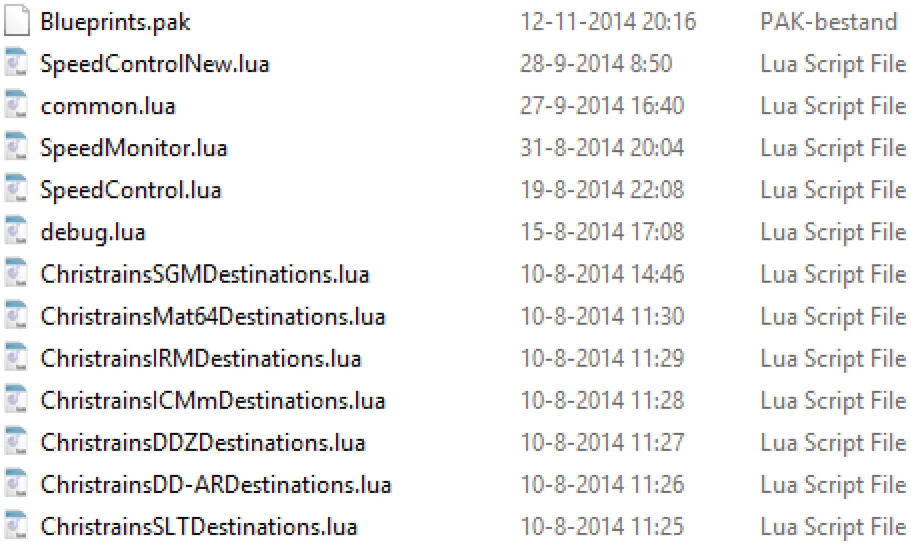
1. Copies of your script are cluttered over a number of locations, which are hard to find back, because all file names are the same and you may have trouble finding the last version.
2. If you need to change anything in a certain script, you need to copy the changes in all scripts.

Therefore I use a way of working that avoids these problems. I created in my assets folder two folders, called **luadev** and **lua**. Luadev I use to develop new scripts or to create new versions of scripts. When a script is stable I just copy the files to the folder lua.



- Assets
- RudolfJan
  - lua
  - luadev
- Content
- data

Most code goes in a script file that resides in this folder structure:



Blueprints.pak	12-11-2014 20:16	PAK-bestand
SpeedControlNew.lua	28-9-2014 8:50	Lua Script File
common.lua	27-9-2014 16:40	Lua Script File
SpeedMonitor.lua	31-8-2014 20:04	Lua Script File
SpeedControl.lua	19-8-2014 22:08	Lua Script File
debug.lua	15-8-2014 17:08	Lua Script File
ChristrainsSGMDestinations.lua	10-8-2014 14:46	Lua Script File
ChristrainsMat64Destinations.lua	10-8-2014 11:30	Lua Script File
ChristrainsIRMDestinations.lua	10-8-2014 11:29	Lua Script File
ChristrainsICMmDestinations.lua	10-8-2014 11:28	Lua Script File
ChristrainsDDZDestinations.lua	10-8-2014 11:27	Lua Script File
ChristrainsDD-ARDestinations.lua	10-8-2014 11:26	Lua Script File
ChristrainsSLTDestinations.lua	10-8-2014 11:25	Lua Script File

In the scenario folder, I create a script called **ScenarioScript.lua** as required by TS.

I try to keep the amount of code in this script to a bare minimum:

```
-[[
 Scenarioscript for testing Guard functions
 (C) 2014 Rudolf Heijink
 Version 0.1 alfa
 ]]--

DEBUG=true

scriptpath=".\\assets\\RudolfJan\\luadev\\"

dofile(scriptpath .. "common.lua")
dofile(scriptpath .. "debug.lua")
mydebug=DebugTS.new()
dofile(scriptpath .. "Guard.lua")

params={
      "22948520",
      "whistleconductorwhistle.wav",
      true
      }


function OnEvent(event)
      if event=="start" then
            guard=Guard.new(unpack(params))
            guard:Begin()
            return TRUE
      end
      return FALSE
end

function OnResume()
      mydebug=DebugTS.new()
      guard=Guard.new(unpack(params))
      guard:Begin()
      mydebug.writeDebug("Resumed guard")
end

function TestCondition ( condition )
      if condition == guard.ConditionName then
            guard:Check()
            return CONDITION_NOT_YET_MET
      end
      return CONDITION_NOT_YET_MET
end
```

The basic components you need here:

Code	Function
`DEBUG=true`	Variable to turn debugging support on or off
`scriptpath=".\\assets\\RudolfJan\\luadev\\"`	Path to the asset folder. On release, it is necessary to change the foldername here to lua
`dofile(scriptpath .. "common.lua")`	This loads the modules you need

`params={`
`"22948520",`
`"whistleconductorwhistle.wav",`
`true`
`}`
	This code I use to set scenario specific parameters for the script. It will become clear later in this guide how this works.

In the current scripts, you also see the event handler here. In principle you can create a separate eventhandling file as well. I have not done this yet, (this guide is still alpha version), but it is worth considering to change this.

I also am thinking about the use of version numbers in the script filename. This would solve possible compatibility problems if I change anything in a particular script. In a next version of this guide, more on that topic.

2.5 Debugging

The start of the debugging process is the reload function. This will reveal syntax errors. The advantage is that it is very fast. Also the compilation stage may reveal additional syntax errors.

The default solution provided for debugging scripts is to use logmate. Logmate is easy to use, but has one big disadvantage. It has avery negative effect on game performance. So, for testing scripts it helps to use a simple route (e.g. TestTrack, Seebergbahn or create your own testing route), otherwise you spend lots of time waiting for the game to load.

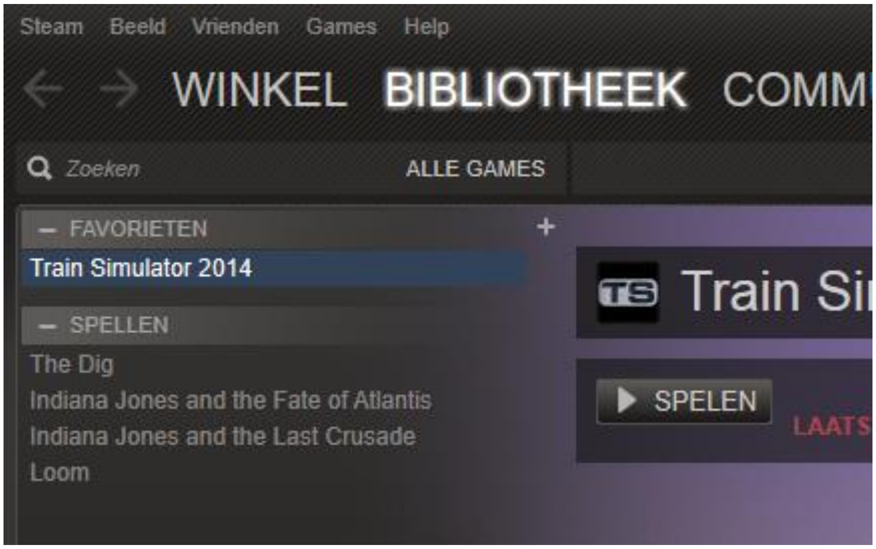
I created a better logging solution, which will be described in this chapter, but you still need LogMate.

LogMate is necessary to find hard programming bugs. In most cases it will report functions that do not exist or variables with nil values. There is no easy to use other way to find these errors.

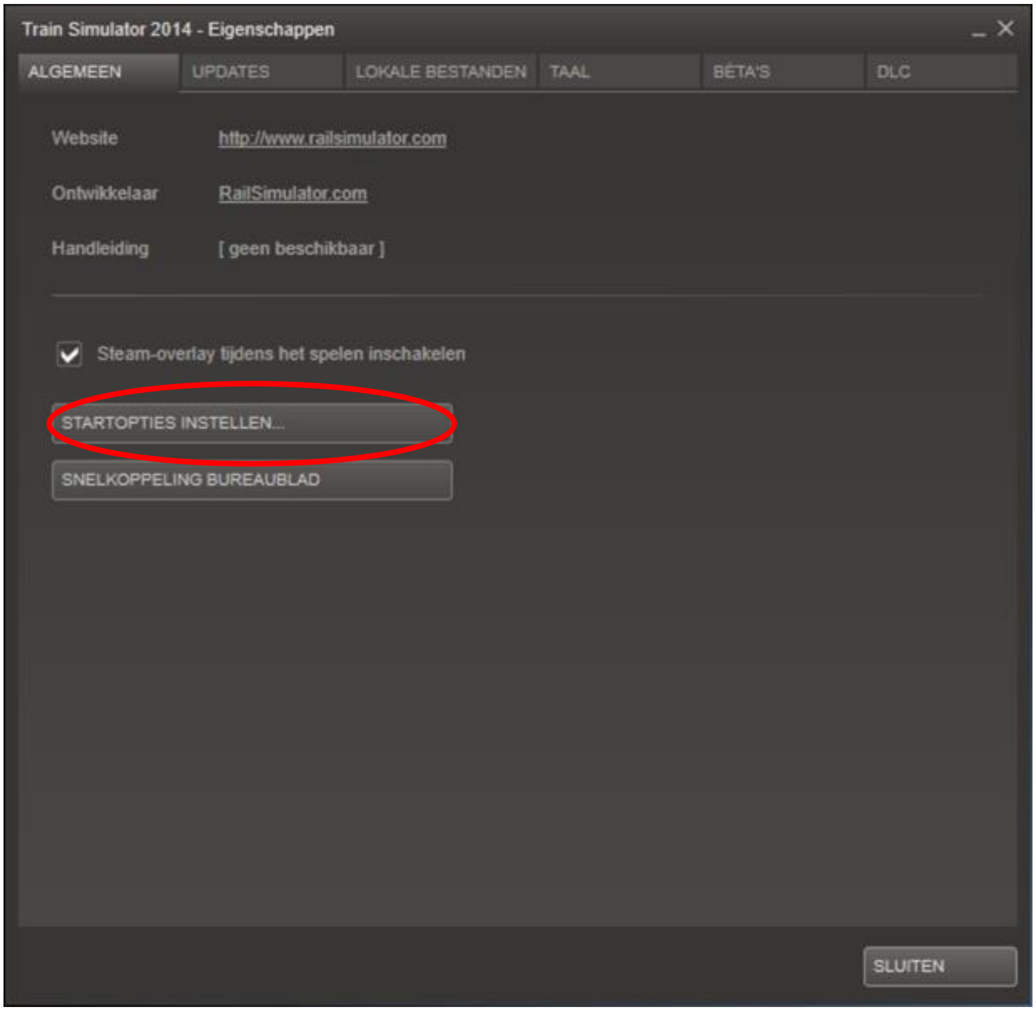
2.5.1 Using logmate

You can use logmate to display hard scripting errors. The lua "error" function will display in logmate in the "script manager" tab. Logmate will also show all output of "Print" statements, provided you have set the DEBUG variable in your script to value **true** NB true is not the same as TRUE!!!

The only way I know to activate logmate is to right click on TS 201x,



choose Properties, General, Set launch options and insert:

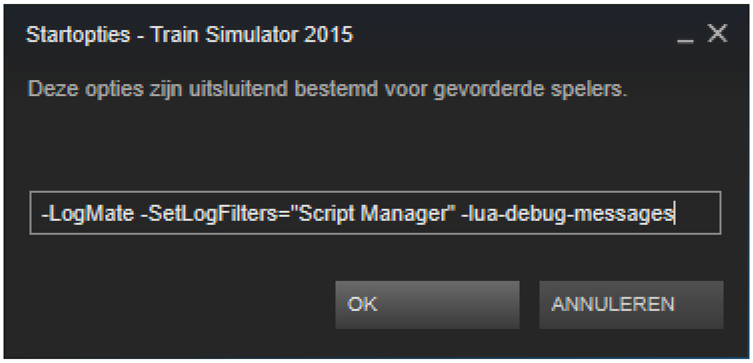


Website http://www.railsimulator.com

Ontwikkelaar RailSimulator.com

Handleiding [geen beschikbaar]

☑ Steam-overlay tijdens het spelen inschakelen



Deze opties zijn uitsluitend bestemd voor gevorderde spelers.

```
-LogMate -SetLogFilters="Script Manager" -lua-debug-messages
```

exactly as typed. It may or may not work.

```
-LogMate -SetLogFilters="Script Manager" -lua-debug-messages
```

The option **–lua-debug-messages** instructs logmate to show the results of "Print" statements, provided you have included the statement

```
DEBUG=true
```

In your script

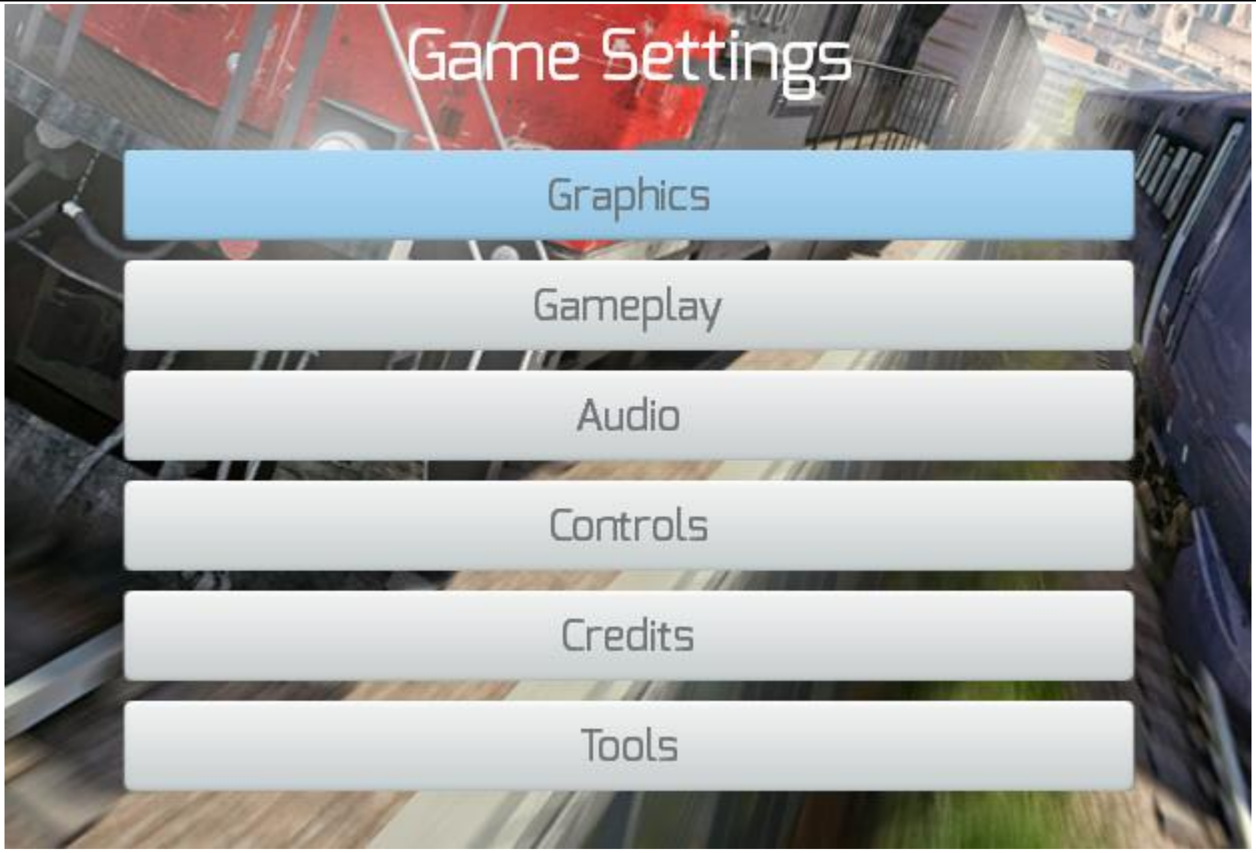
Note:

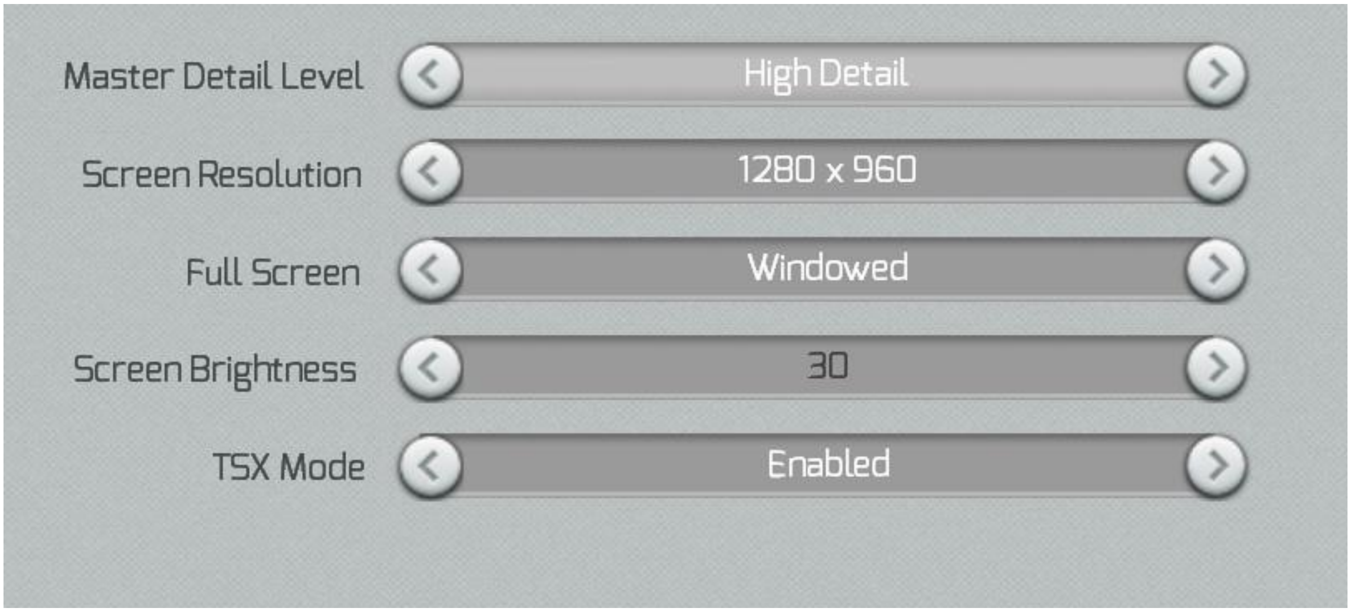
Use "Script Manager" as filter, not the "All" variant you usually see. Using the "All" filter reduces performance significantly.

Tip:

Run Trainsimulator windowed. Especially if you have a second screen, you can have a look at the Logmate while playing. Open TS, in the Main Menu select the Settings button (upper right on the screen)

Choose Graphics





Master Detail Level: High Detail
Screen Resolution: 1280 x 960
Full Screen: Windowed
Screen Brightness: 30
TSX Mode: Enabled

Set “Full screen” to Windowed and reduce the screen resolution a bit

2.5.2 Speed up scenario play

When you load up a scenario and hold [ctrl] while you left click on a train/loco and the AI will drive it for you through all instructions and you can just enjoy the landscape. You won't have access to the standard passengerview but you can adjust the camera position to go inside the wagons. With [PageUp] and [PageDown] you can switch between the trains.

Also: when you add the option "- EnableAsyncKeys" to the start options (as described in the previous paragraph) you can speed up the time through pressing [ctrl] + [shift] +[1] up to [5] (1 normal speed, 2 double, 5 is 5x speed).

2.5.3 Print for debug

Since there is not a full blown source level debugger for LUA, you need to use some form of print statements to monitor and debug script execution

The LUA “Print” function will print the results in LogMate, but you must set the value for the variable DEBUG:

```
DEBUG=true
Print(“Debugging is switched on now”)
```

You also must set the start option –lua-debug-messages

Note:

I have long been confused, because is many scripts you see constants defined:

```
TRUE=1
FALSE=0
```

Setting DEBUG to these constants does ***not*** work!

Also, people advocate to use this:

```
-- No need for this!
Function DebugPrint(message)
      If DEBUG then
            Print(message)
      end
end
```

There is no need to do so, Print is a debug print function. There is no reason to use this construct.

2.5.4 An alternative logging system

I designed a simple logging system, that allows you to write debugging information to a separate logfile, which you can review after program execution.

It works like this:

Step 1: create a logfile. This file will be opened in append mode, so existing data will not be removed automatically.

```
	file=io.open(“logfile.txt”, "a+")
```

You can find your logfile in the TS folder.

Step 2: define a function to write debug messages to a logfile:

```
-- file is a valid file handle
-- message is a string containing a text message
function writeDebug(file, message)
	dt= os.date("%d-%m-%Y/%X ")
	if(dt==nil) then
		dt=""
	end
	file:write(dt .. message ..'\n')
	file:flush()
end
```

I did not include error handling code, because you cannot write an error message is debug printing fails … Normally you would handle the situation where file has value nil.

As you see I include a timestamp for each message.

Step 3: close the logfile when done (optional)

```
file:close()
```

If you do not close the logfile, it will be locked by TS, so you cannot delete it while TS is running. I did not yet find a suitable function, something like OnExit() that is run at the end of a scenario to solve this issue.

You may want to use the print function to send data to logmate as well. In this example a debugging condition must be met. In your script you can activate debugging by defining a variable

```
 DEBUG = true
```

in your script. If DEBUG=0, nothing will be printed,

```
-- file is a valid file handle
-- message is a string containing a text message
function writeDebug(file, message)
	if(DEBUG) then
		…
	end
end
```

Alternatively, you can also send messages to your TS screen. In this case you do not need to create a file first, but you cannot include a large number of debug statements.

```
function DebugPrint( message )
      if (DEBUG) then
            SysCall ( "ScenarioManager:ShowMessage", message);
      End
End
```

This command will show a message in the upper right corner of the screen during some time.

Note: the code shown here is not optimally decent programmed. In a next update better functions will be provided and I will try to use more object oriented techniques.

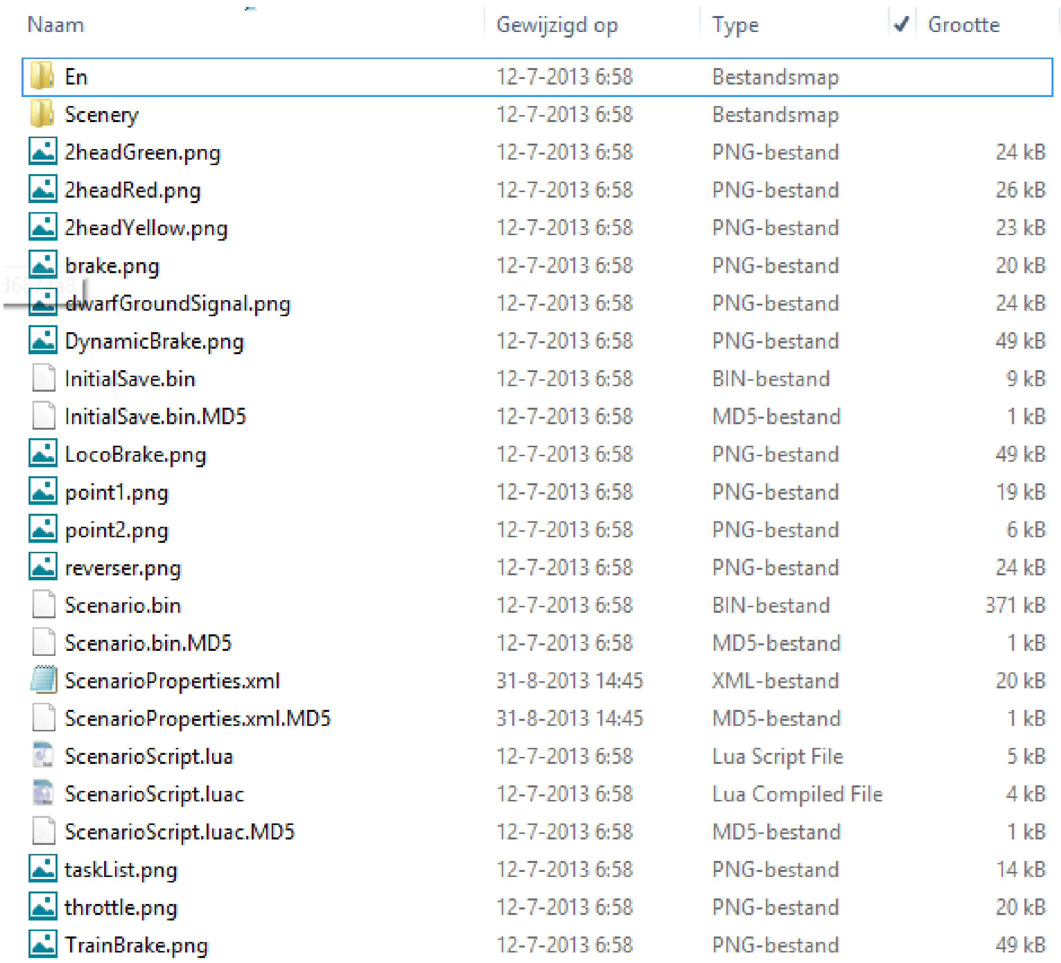
3 Organization

3.1 Directory structure

Your LUA scenario script must be called

```
ScenarioScript.lua
```

If you use media, create in the scenario directory a directory for the languages you support, e.g. EN for English. See example below.



Naam	Gewijzigd op	Type	Grootte
En	12-7-2013 6:58	Bestandsmap	
Scenery	12-7-2013 6:58	Bestandsmap	
2headGreen.png	12-7-2013 6:58	PNG-bestand	24 kB
2headRed.png	12-7-2013 6:58	PNG-bestand	26 kB
2headYellow.png	12-7-2013 6:58	PNG-bestand	23 kB
brake.png	12-7-2013 6:58	PNG-bestand	20 kB
dwarfGroundSignal.png	12-7-2013 6:58	PNG-bestand	24 kB
DynamicBrake.png	12-7-2013 6:58	PNG-bestand	49 kB
InitialSave.bin	12-7-2013 6:58	BIN-bestand	9 kB
InitialSave.bin.MD5	12-7-2013 6:58	MD5-bestand	1 kB
LocoBrake.png	12-7-2013 6:58	PNG-bestand	49 kB
point1.png	12-7-2013 6:58	PNG-bestand	19 kB
point2.png	12-7-2013 6:58	PNG-bestand	6 kB
reverser.png	12-7-2013 6:58	PNG-bestand	24 kB
Scenario.bin	12-7-2013 6:58	BIN-bestand	371 kB
Scenario.bin.MD5	12-7-2013 6:58	MD5-bestand	1 kB
ScenarioProperties.xml	31-8-2013 14:45	XML-bestand	20 kB
ScenarioProperties.xml.MD5	31-8-2013 14:45	MD5-bestand	1 kB
ScenarioScript.lua	12-7-2013 6:58	Lua Script File	5 kB
ScenarioScript.luac	12-7-2013 6:58	Lua Compiled File	4 kB
ScenarioScript.luac.MD5	12-7-2013 6:58	MD5-bestand	1 kB
taskList.png	12-7-2013 6:58	PNG-bestand	14 kB
throttle.png	12-7-2013 6:58	PNG-bestand	20 kB
TrainBrake.png	12-7-2013 6:58	PNG-bestand	49 kB

In the example you see .png images. If these a language neutral, you just store them in the scenario directory. If they are localised, store them in the language specific directory.

The execution directory for lua scripts is the Trainsimulator program content directory, normally:

```
C:\Program Files (x86)\Steam\SteamApps\common\RailWorks\Content
```

So, if you want to access files from other directories you need to take this into account. It is not a very good idea to use hard paths to directories, so you either need to find a way to access the registry from lua or you need to use relative paths.

Example, I use some common libraries in my assets folder, which can be accessed using the following path string:

```
.\assets\\RudolfJan\lua\common.lua
```

The . (dot) means the current directory. You also may use .. (two dots), meaning start in the parent directory.

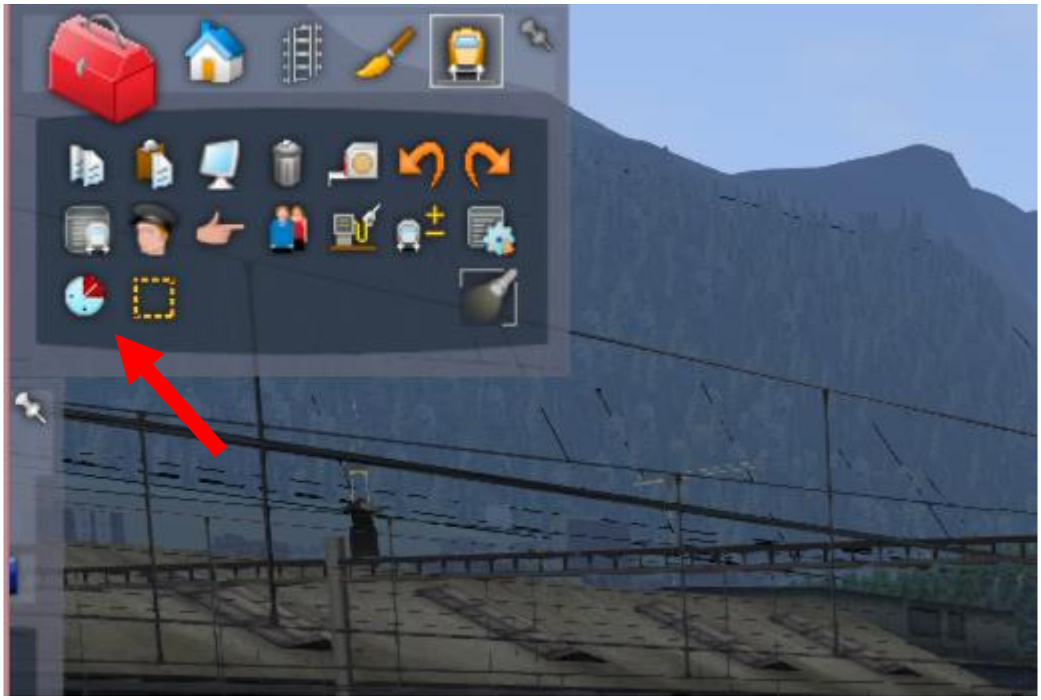
You need to escape the backslashes if you use this in lua commands, e.g. in the dofile command, which executes common.lu in this example:

```
dofile(".\\assets\\RudolfJan\\lua\\common.lua")
```

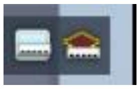
dofile is a lua library function, very useful.

3.2 Accessing your script from inside TS2014

You can access the LUA script for the scenario edit function in TS2014. You need to open the Timetable view (2D editor view),

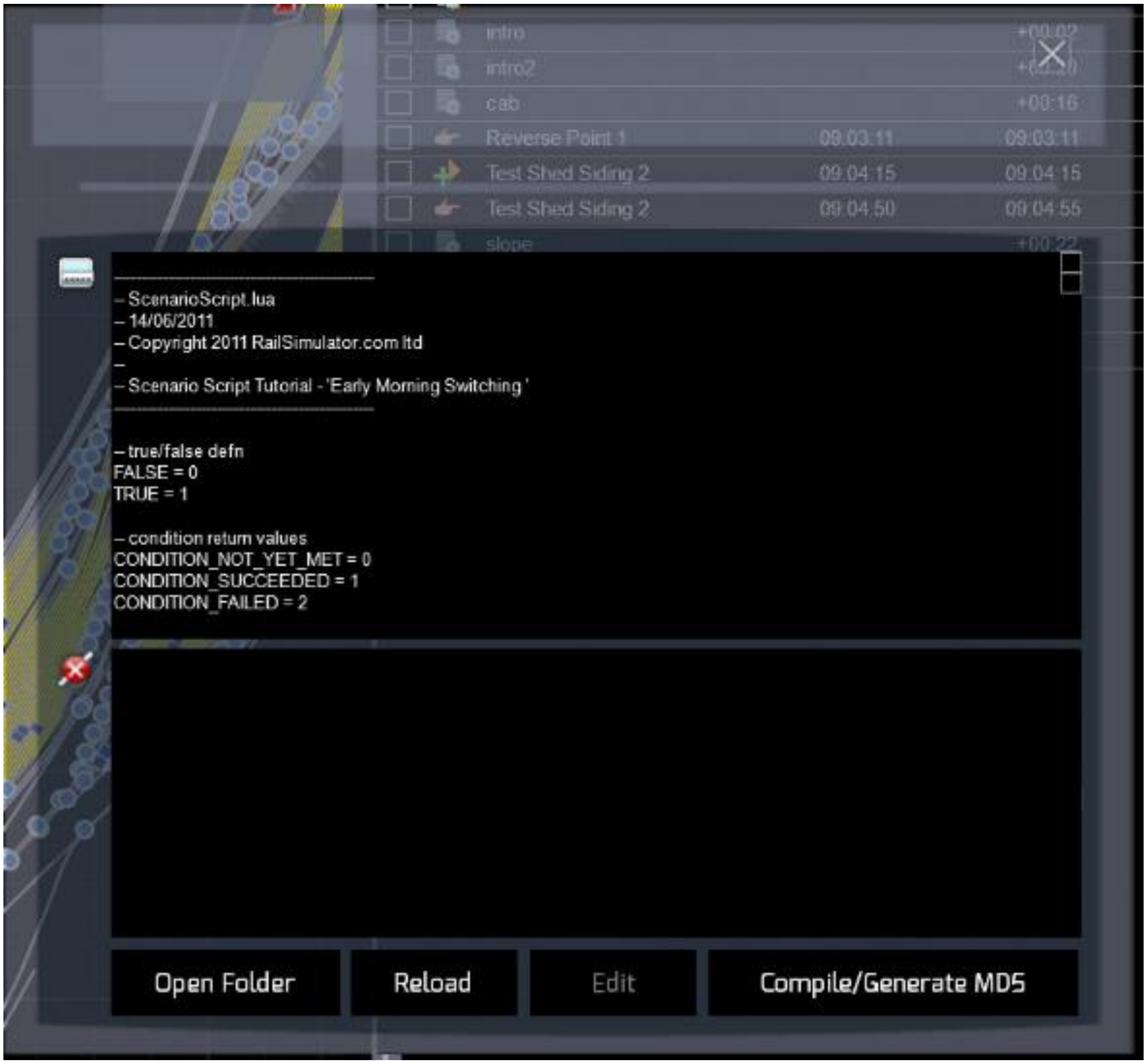


and will see then two new buttons:



The left button opens a view on the LUA script. The right button opens a string definition screen. <<not yet clear to me what it exactly does, more info comes later>>..

If you click on the LUA button, a pop-up screen appears:



```
------------------------------------
– ScenarioScript.lua
– 14/06/2011
– Copyright 2011 RailSimulator.com ltd
–
– Scenario Script Tutorial - 'Early Morning Switching '
------------------------------------

– true/false defn
FALSE = 0
TRUE = 1

– condition return values
CONDITION_NOT_YET_MET = 0
CONDITION_SUCCEEDED = 1
CONDITION_FAILED = 2
```

You can use this screen to have a look at the LUA script. It does not show embedded files. Unfortunately most scripts provided by RSC are compiled, so it is difficult to use them as examples for your own development.

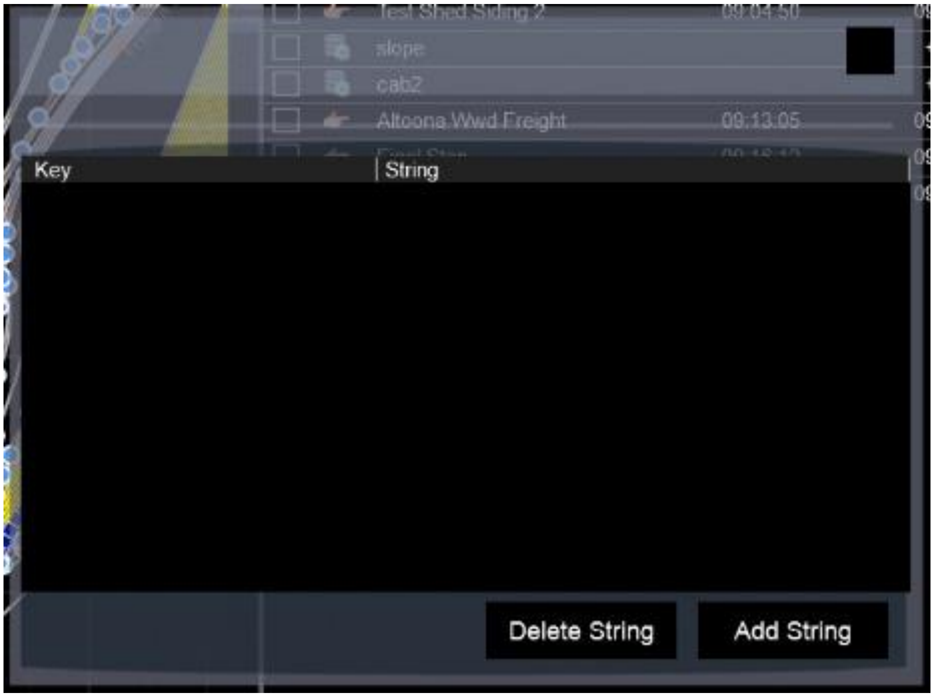
Open folder will open the scenario folder. If you have SciTE installed, you can now right click on the .lua file to edit it.

Before you can use the script, you need to compile it, using the Compile/Generate MD5 button.

Probably you need to press Reload before you can start executing the scenario again. Reload is useful anyway, because it will report syntax errors in your script.

As you see, you have no access to the HTML files that may be used, so I thing the best way to proceed, is to open the folder and do all editing using proper tools.

The Strings pop-up looks like this:



I have no idea what you can do with it. Adding a string creates u new GUID (Globally Unique IDentifier), to which you can add a string description.

3.3 Basic API call

The basic call for TS API is SysCall(command, ..) Please note that in documentation you may find the use of Call as the API calling function. This does not seem to work in scenarios. I wasted some time for a script to obtain the consist mass. It did not work using Call, but works perfectly well by just replacing Call by SysCall.

3.4 Include files

It may helpful to include common files you add as assets to a scenario. You can do this using following code:

```
assetpath=".\\assets\\RudolfJan\\luafile.lua"
dofile(assetpath)
```

dofile opens the path and executes its contents on the fly. I think this code will not be compiled and compressed, because you can determine which file to use dynamically. This makes the mechanism very powerful.

4 Events

4.1 The event loop

Scenario scripts are small pieces of code that are executed if an ***event*** fires. A scenario script therefore always needs to handle events.

An event is something that happens during your drive. For instance, a trigger instruction can be an event, but also after loading passengers, or a scheduled stop may trigger events.

Your script always should contain following code:

```
-- true/false defn
FALSE = 0
TRUE = 1

-- Fn OnEvent
--          event - name of the event
--        return - TRUE/FALSE if event handled

function OnEvent ( event )

      if ( event == “start”) then
            -- code here
            return TRUE – event handled
      end
return FALSE
end
```

This function allows you to pick up events, and choose to handle them. In the code above an if statement is used for each event. In the **OnEvent** function, you include a similar structure for each event you want to handle.

NB Event names are case sensitive, so ***Start*** denotes a separate event from ***start***.

The function **OnEvent** may return either TRUE or FALSE

In case you return FALSE TS2014 will execute its own system code, for instance to display the message after a stop. If you return TRUE, TS2014 will assume no further handling is required on this event.

4.2 Basic events

Even without lua programming, some basic events can be set, You know them well:

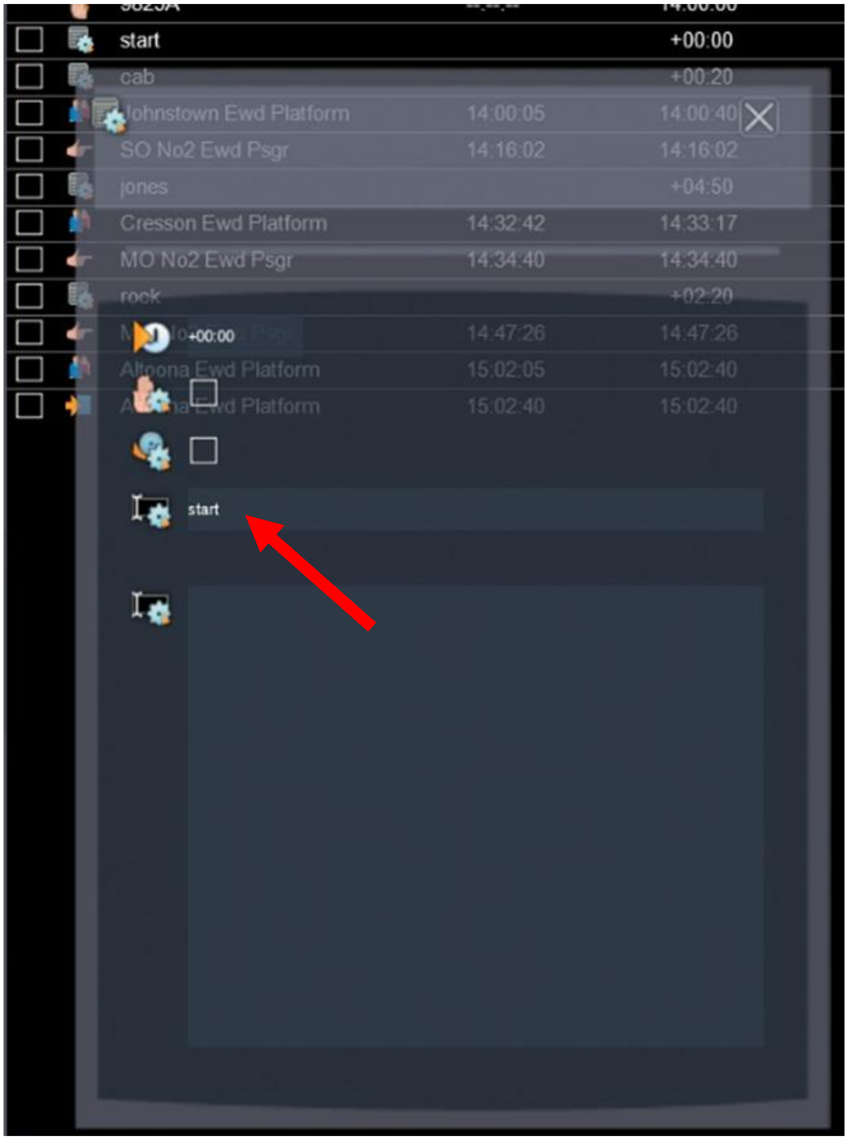
a) The trigger event
b) The stop event
c) The load event
d) The coupling event
e) The uncoupling event
f) The go via event

All but the Go via event allow to add additional triggers after the basic events have occurred.

4.3 Adding lua code to the trigger event

It is quite common to use a trigger instruction as an event.

If you want to do so, create a trigger event using the 2D scenario editor.



You just need to give the trigger event a name, as you see in the example above. In this case, the name “start” is used. Make sure that you never use a name twice in the same scenario.

Now, if the trigger event fires, you can handle it. A short code example:

```
function OnEvent ( event )

-- Instruction triggers
---------------------------
```

```
-- Timed triggers
---------------------------

      if event == "start" then – do something

      --     Your code here
      return TRUE;
      end
 return FALSE;
end
```

In the example you see that the function **OnEvent**, is extended now. The line

```
if event == “start”
```

contains instructions on what to do on this trigger. In this case it starts looking around, and disables all controls, so the engineer just has to wait. I will explain the instructions later into more detail.

Note that you return TRUE after handling the start event. If not any event is handled, FALSE is returned, to tell TS2014 it should handle the event itself.

NB. Note in the trigger instruction some events are predefined.:

a) A wheelslip event
b) An emergency stop event
c) An event that displays a text box

The first two events can be activated by placing a checkmark, the third one is activated if you enter a text.

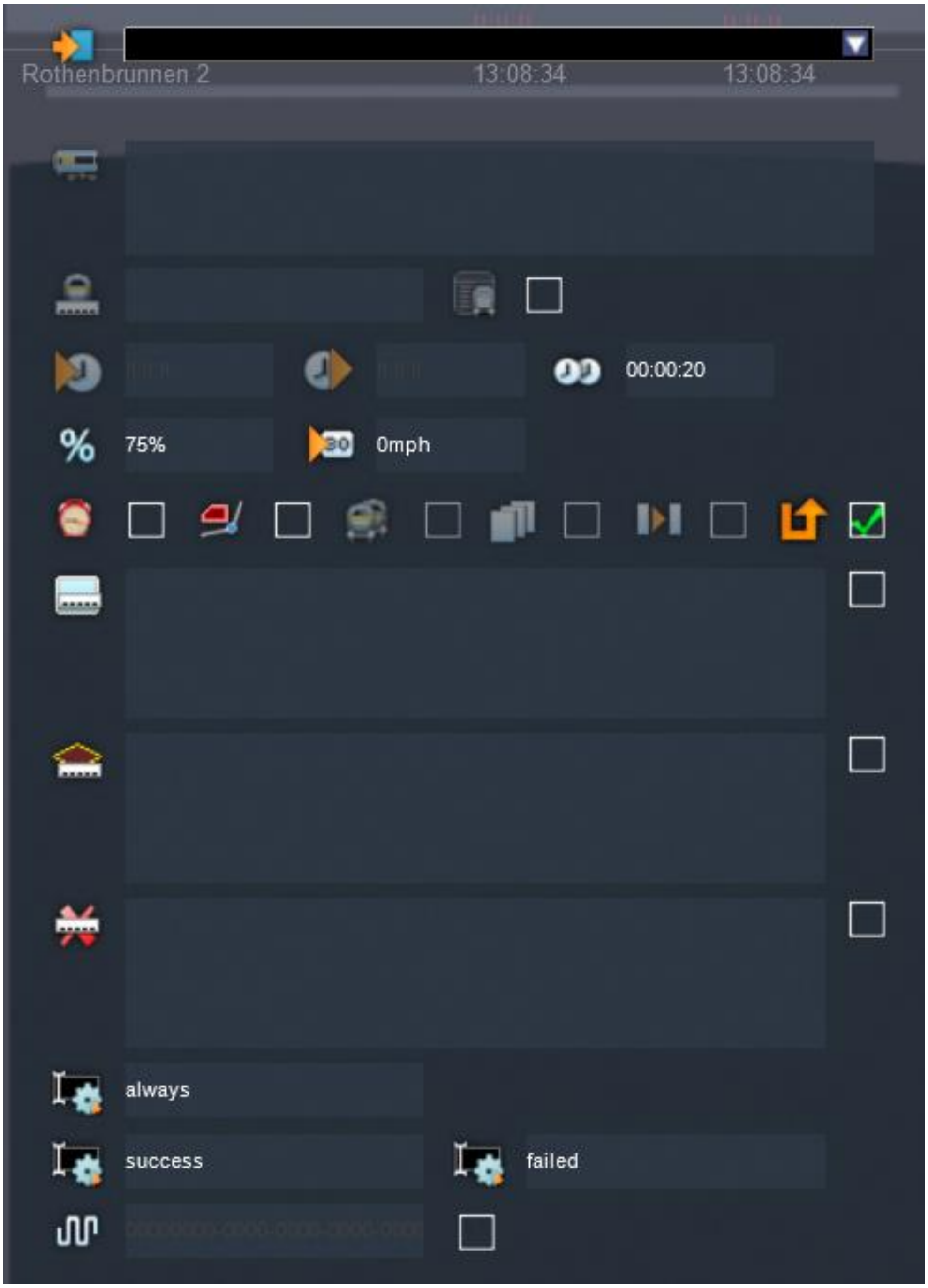
4.4 Adding lua code to the other basic events

Following basic events support adding lua event handling in the same way:

a) The stop event
b) The load event
c) The coupling event
d) The uncoupling event

For these events TS2014 always executes the basic event first. You can add events the for three different results of the basic event:

a) The event is always triggered.
b) The event is triggered in case the basis event failed
c) The event is triggerd in case the basic event succeeded



Rothenbrunnen 2 13:08:34 13:08:34

00:00:20

75%

0mph

always

success

failed

In the screenshot you can see where you can enter the event identifiers.

The names I used are samples. Any identifier text is allowed.

5 Lua functionality for TS2014

In the previous chapters I provided the basics to get started. Now we will focus on the functionality TS2014 provides to scenario scripting.

The good news is that al lot of things are possible. The bad news is the very poor documentation. To make it worse, for different engines commands may be implemented in a different way. You will need a lot of trial and error, even for simple actions like starting or stopping an engine.

I will describe in following chapters what I discovered. The chapters contain following topics:

a) General scenario manager functions
b) Displaying messages
c) Camera control
d) Engine functions
e) Other functions

Please, let me know if you discover new possibilities.

TS2014 distinguishes several different modules. Up till now I discovered following “managers”:

a) Scenario Manager
b) Camera Manager

For engines there does not appear to be an explicit management unit, you use specific commands to control an engine.

6 General scenario manager functions

6.1 Object identification

Any method call that begins `Call("<Entity>:...")` where `<Entity>` is anything such as `<ParticleEmitter>` or `<ChildName>` can also be called using the `Call("*:...")` qualifier so the parent and all descendant entities are checked, rather than specifying the entity to check.

You can always identify the player engine using the name "PlayerEngine", for example:

```
SysCall ( "PlayerEngine:SetControlValue", "Headlights", 0, 1 );
```

PlayerEngine replaces the "*" here.

SysCall ("PlayerEngine:SetControlValue", "Startup", 0, -1);

And the same for an AI :-

SysCall ("8510:SetControlValue", Startup, 0, 0.0);

8510 is the reference to a specific engine or waggon. I did not yet do very much testing with this, but it opens possibilities like starting or stopping engines in a multiple working separately. I already fancy the idea of a scenario where a helper engine suddenly fails…

6.2 Game function calls

No idea if this is useful.

1. SysCall("ControlSound:SetParameter", "SignalProgress", 1) - Sound trigger? "SignalProgress" = name/ID in xml? 1 = boolean for start/stop?
2. SysCall("Set2DMapSignalState", gSignalState) - Setting 2D map state: gSignalState holds int depending on constant inputted?
3. SysCall("SendSignalMessage", newBlockState, "", -1, 1, 0) - Send message along track, not sure which way round tho'
4. SysCall("SendConsistMessage", SPAD_MESSAGE, "") - Sends message to consist above signal link: SPAD_MESSAGE is text to send?
5. SysCall("GetConnectedLink", linkCountAsString, 1, 0) - Finds link num connected to link 0 of signal
6. SysCall("GetLinkCount") - Finds num of links on signal
7. SysCall("GetId") - gets name of signal

6.3 Event handling

The OnEvent function has been described before. There are few other functions that may be useful as well:

```
function Initialise()
```

Function that is called once upon scenario initialization. Should be used to set up variables/simulation elements of a script at the start of a scenario e.g. turning off/on lights. It is called before the signalling is set up properly.

Note: mind the spelling it only works when spelled with "s", initialize() will NOT work.

```
function Setup()
```

Function that is called once when the script is initialized (e.g. when the player clicks the consist.) Can be used for when an event is to occur when the consist is clicked, e.g. raising a pantograph.

Following functions seem NOT to work in scenario scripts. They probably can be used in engine scripts only.

```
function Update(interval)
```

Function that is called every new frame to update the entities simulation. Parameter `interval` is the time since last update (known as the delta time, or delta for short.).

Similar, there are eventhandlers like OnInitialise() that allow you to add simulation functionality.

```
function OnCameraEnter(cabEnd, carriageCam)
```

Probably not working in scenario scripts.

Function that is called when the player camera enters a cab or passenger view. Parameter `cabEnd` is the cab that the player has entered and is either 1 or 2 and `carriageCam` denotes whether it was a passenger view or not, being 1 if it is a passenger view.

```
function OnCameraLeave()
```

Function that is called when the player leaves a camera view for another.

6.4 Saving and resuming a scenario

The first step is initialization code.

```
function Initialise()

      HideFireworks();

end -- function Initialise()
```

There is also an event handler that fires when you resume simulation:

```
function OnResume()

      HideFireworks();

end -- function OnResume()
```

The example is from the Horse Shoe curve scenario, using fireworks.

I did not yet discover any way to store data when you save a scenario with the Save function. Save does not work, OnSave does not work etcetera… Let me know if you have any idea.

6.5 Locking controls

One thing you can do is lock all controls, forcing the engineer just to watch:

```
SysCall ( "ScenarioManager:LockControls");
```

You can just copy this code.

Do not forget to unlock the controls later:

```
SysCall ( "ScenarioManager:UnlockControls");
```

6.6 Forcing scenario completion

Two functions that terminate a scenario. The first triggers a success, the second failure.

```
SysCall ( "ScenarioManager:TriggerScenarioComplete, message);
```

Triggers a scenario failure and displays an appropriate message:

```
SysCall ( "ScenarioManager:TriggerScenarioFailure", message);
```

6.7 Condition checks

A function that checks at regular intervals for a specific condition, with a specific name.

```
SysCall ( "ScenarioManager:BeginConditionCheck", condition );
```

condition is a unique identifier representation a condition.

You need to define a function that checks for this condition:

```
function TestCondition ( condition )
```

Inside this function you evaluate the conditions

which returns a status with respect to the condition:

```
CONDITION_NOT_YET_MET = 0
CONDITION_SUCCEEDED = 1
CONDITION_FAILED = 2
```

I think this may be useful if you want to stop checking for a condition, without doing this explicitly from an event. I did not yet need or use this construct.

To stop the condition check, you need this call:

```
SysCall ( "ScenarioManager:EndConditionCheck", condition );
```

A complete sample skeleton may look like this:

Step 1 uses following code:

```
if (event== “startMonitoring”) then
      SysCall ( "ScenarioManager:BeginConditionCheck", "MyCondition" );
      return TRUE
end
```

“MyCondition” is a name you can choose for the condition you like to monitor.

Step 2 requires a function that performs the monitoring:

```
function TestCondition ( condition )
      if (condition == "MyCondition") then
      -- here goes your code on what to monitor
      end
      return CONDITION_NOT_YET_MET
end
```

This allows you to use monitor that runs forever but also monitoring that will stop, depending on the condition, e.g. you may want to wait till a specific time or till you meet an AI consist …

Step 3 is to stop monitoring:

```
if (event== “stopMonitoring”) then
      SysCall ( "ScenarioManager:EndConditionCheck", "MyCondition" );
      return TRUE
end
```

Tip:

You should be aware that is the user saves the game and continues later, state information of a condition check may be lost. For instance, in the speed check Thomas Ross created, the speed violations are counted, but the counter is reset when you resume the game. You need to make sure all relevant data are stored somewhere and retrieved on resume. In chapter **Fout! Verwijzingsbron niet gevonden.** it is explained you can use commands like OnResume() to achieve this.

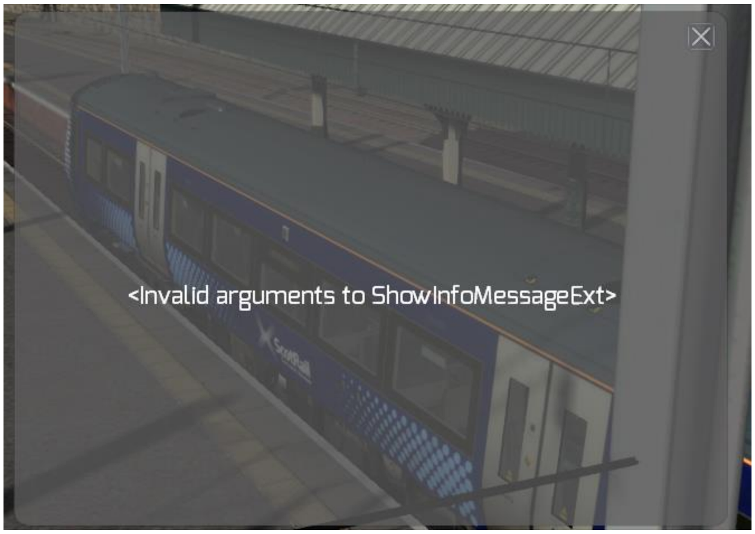
7 Displaying messages

7.1 Introduction

TS201x supports several different methods to display messages:

- Messages coupled to predefined events. These messages do not need any lua programming.
- You can add these simple messages as well by programming them in lua code and attach them to events. I found three different functions to show messages, each with own features.
- You also can use message with HTML markup. This allows the use of images, tables, colours and other fancy markup.

For programmed messages, lu will give you an error message if something is wrong, but unfortunately it does not give any further details:



<Invalid arguments to ShowInfoMessageExt>

In the next section four different message types will be discussed in more detail:

1. Simple messages, using the same simple layout and placement as the messages you can create without programming in the events.
2. Alert messages which appear in the upper right corner and therefore are far less intrusive for the game play.
3. Extended info massages which offer fine grained control in placement and behaviour
4. HTML messages with advanced markup features.

7.2 Message types

7.2.1 Simple message

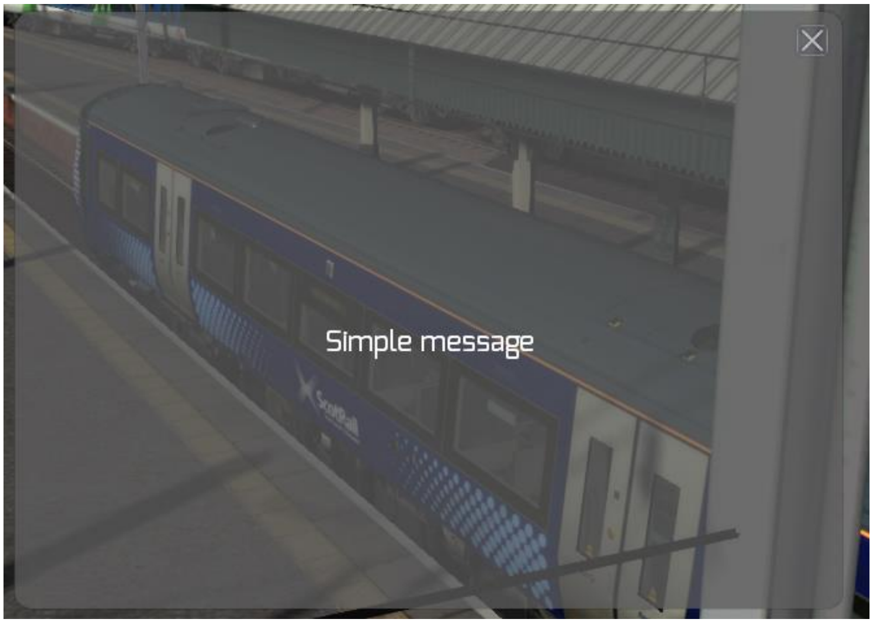
The easiest way to display a message is to use this command:

```
SysCall ( "ScenarioManager:ShowMessage", message);
```

It will display a message text at the centre of the screen.

```
Function OnEvent(event)
        if event==”start” then
                SysCall("ScenarioManager:ShowMessage",”Simple message”)
                return  TRUE
        end
        return FALSE
end
```

The result looks like this and appears in the middle of the screen. Game is not paused and the message is displayed for a fixed quite long time.



Simple message

7.2.2 Alert message

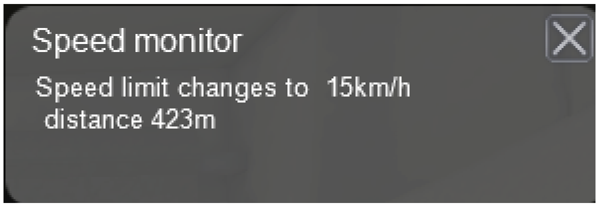
This command shows an alert message in the upper right corner of the screen.

```
SysCall("ScenarioManager:ShowAlertMessageExt" header, text, duration, pause)
```

Example:

```
SysCall("ScenarioManager:ShowAlertMessageExt", "Speed monitor", "Speed limit
changes to  " .. self.NextSpeedLimit .. unittext .."\n distance " ..
self.Distance .. "m", 10)
```

Results in this output:



Speed monitor

Speed limit changes to 15km/h
 distance 423m

Note: I used "\n" to force a new line.

7.3 Info message

The basic function to display messages has this syntax:

```
SysCall ( "ScenarioManager:ShowInfoMessageExt", header, message, duration,
position, size, pause);
```

Parameter	Description	Comments
command	Fixed value, see syntax	
header	Message header text	
message	Message body text	
duration	No of seconds the message is show, 0 means until user clicks	
position	Place on the screen where the message will be displayed	
size	Message size	
pause	If TRUE, pause the gameplay while you display the message.	Very useful, fi if you want to give a complex instruction to the user while driving. Note: do not use the Boolean variable true or false. This results in an invalid argument error message!

For position use following values:

Direction	Value	Description	Comments
Vertical	MSG_TOP	Top position	
Vertical	MSG_VCENTRE	Middle position between top and bottom	
Vertical	MSG_BOTTOM	Bottom position	
Horizontal	MSG_LEFT	Left position	
Horizontal	MSG_CENTRE	Middle position	

Horizontal	MSG_RIGHT	Right position

You should always combine a horizontal and vertical position. The screens you see without using lua have the position value MSG_CENTRE+MSG_VCENTRE, which places them in the middle of the screen. So lua allows you to do something about this annoy practice.

Position can be a combination of values. RSC defined some macros, which I extended to a more complete set. You need to define these values in your script:

```
-- Message positions
MSG_TOP = 1
MSG_VCENTRE = 2
MSG_BOTTOM = 4
MSG_LEFT = 8
MSG_CENTRE = 16
MSG_RIGHT = 32
```

Some useful combinations:

```
-- useful combinations

MSG_TOPLEFT = 9
MSG_TOPRIGHT = 33
```

The other goody is that you can define three different sizes:

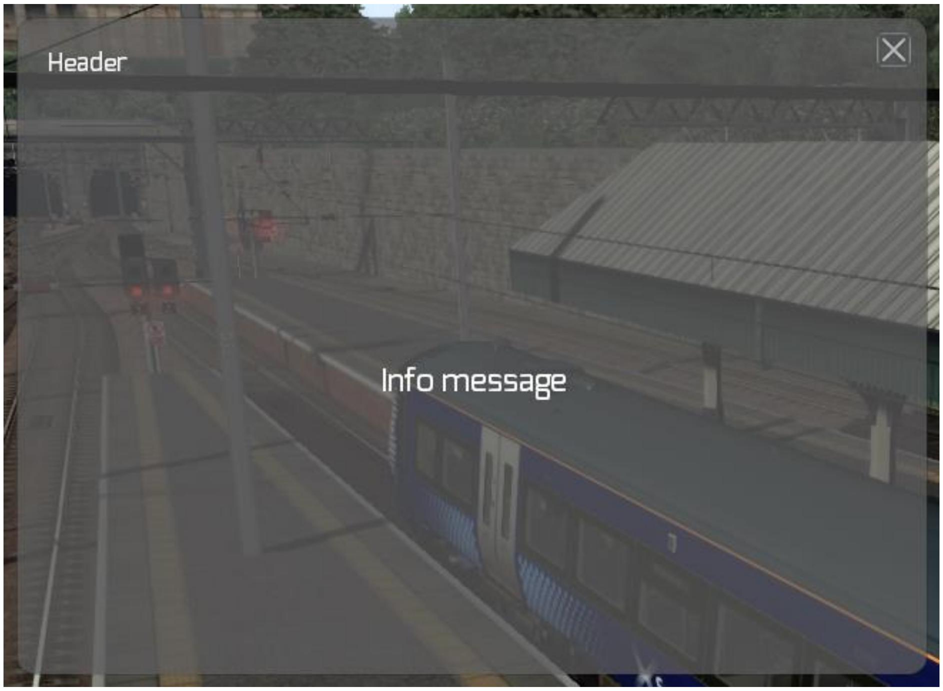
Size	Value	Description	Comments
Small	MSG_SMALL	Small messagebox (like alert message)	
Regular	MSG_REG	Normal sized message box	
Large	MSG_LARGE	Large sized message box	

You need to define these values in your script:

```

-- Message box size

MSG_SMALL = 0
MSG_REG = 1
MSG_LRG = 2
```

Example:

```
SysCall ( "ScenarioManager:ShowInfoMessageExt", "Header", "Info message", 30,
MSG_TOPLEFT, MSG_REG, TRUE);
```



7.4 HTML messages

The command for HTML messages is very similar as the one for Info messages, but the meaning of the third and fourth parameter is different:

```
SysCall ( "ScenarioManager:ShowInfoMessageExt", GUID , html, duration, position,
size, pause);
```

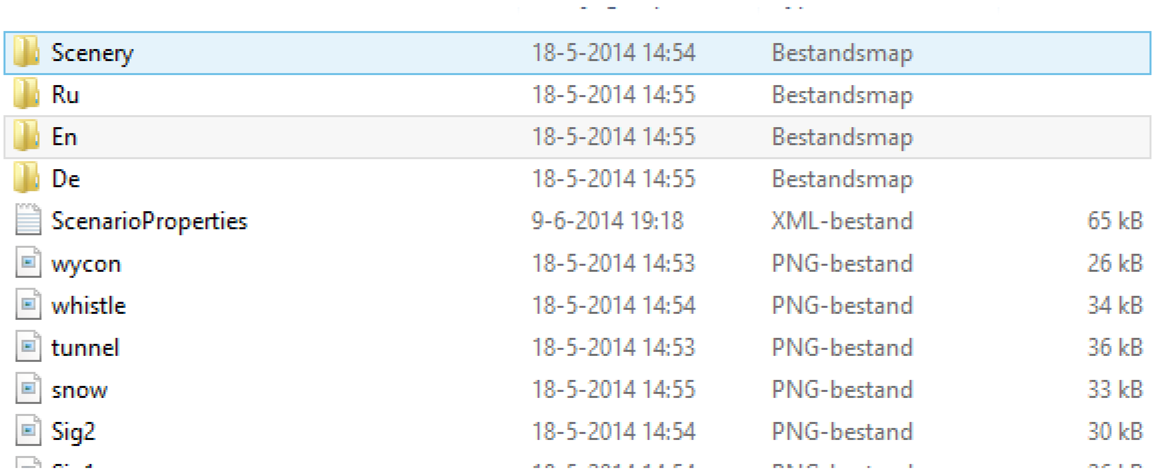
The parameters must be filled as described in the table:

Parameter	Description	Comment
command	As above, fixed	
GUID	A GUD	I suggest you use the GUID that identifies the scenario. Rumors say you must use the same GUID for all messages in one scenario. I think this GUID allows the html code to return properly
html	Filename of html script	See below for details, there some issues
duration	Time In seconds the message is shown, if zero, it is shown till the user closes the message	
position	See previous chapter for details, works the same way	
size	See previous chapter for details, works the same way	
pause	If TRUE, the game pauses during message display	

You need to create an html file. Because localization is supported for html messages, you must create a subdirectory, with the language abbreviation you want to support. I suggest you always create a directory named "EN" as a default language. Additionally, you can create directories for other languages, e.g "NL" for Dutch. If you do not know which abbreviation to use, check the TS2014 manuals directory.

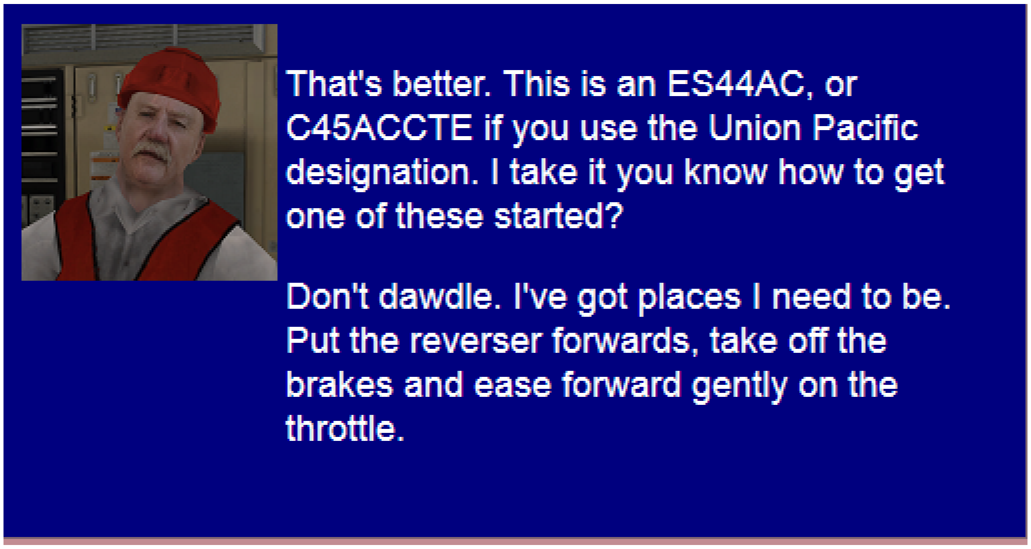
Note: the folder name may be case sensitive. Chris Longhurst reported this for playing sounds. I did not check this thoroughly for HTML messages yet.

Sample folder structure:



Scenery	18-5-2014 14:54	Bestandsmap	
Ru	18-5-2014 14:55	Bestandsmap	
En	18-5-2014 14:55	Bestandsmap	
De	18-5-2014 14:55	Bestandsmap	
ScenarioProperties	9-6-2014 19:18	XML-bestand	65 kB
wycon	18-5-2014 14:53	PNG-bestand	26 kB
whistle	18-5-2014 14:54	PNG-bestand	34 kB
tunnel	18-5-2014 14:53	PNG-bestand	36 kB
snow	18-5-2014 14:55	PNG-bestand	33 kB
Sig2	18-5-2014 14:54	PNG-bestand	30 kB

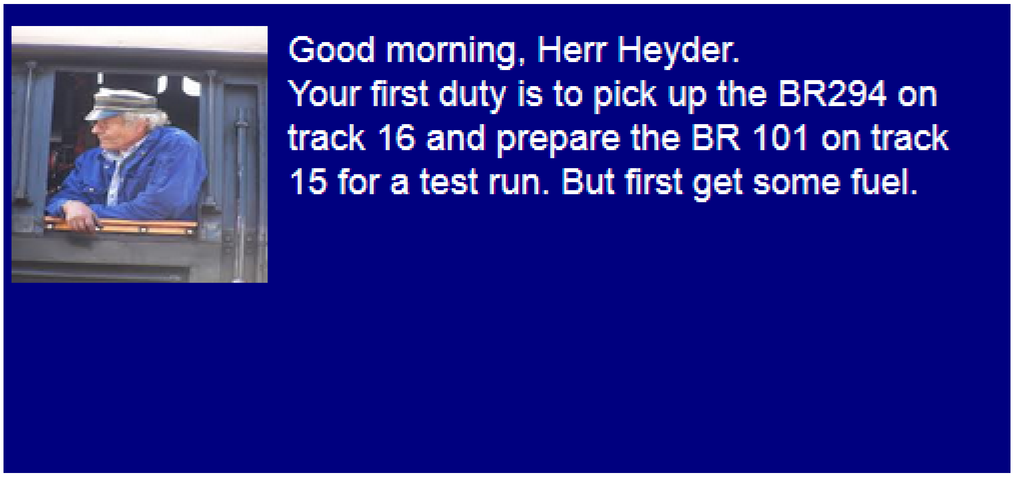
You see here three language dependent folders with html scripts. The images, e.g. wycom.jpg are not language dependent and therefore are stored in the scenario directory.



That's better. This is an ES44AC, or C45ACCTE if you use the Union Pacific designation. I take it you know how to get one of these started?

Don't dawdle. I've got places I need to be. Put the reverser forwards, take off the brakes and ease forward gently on the throttle.

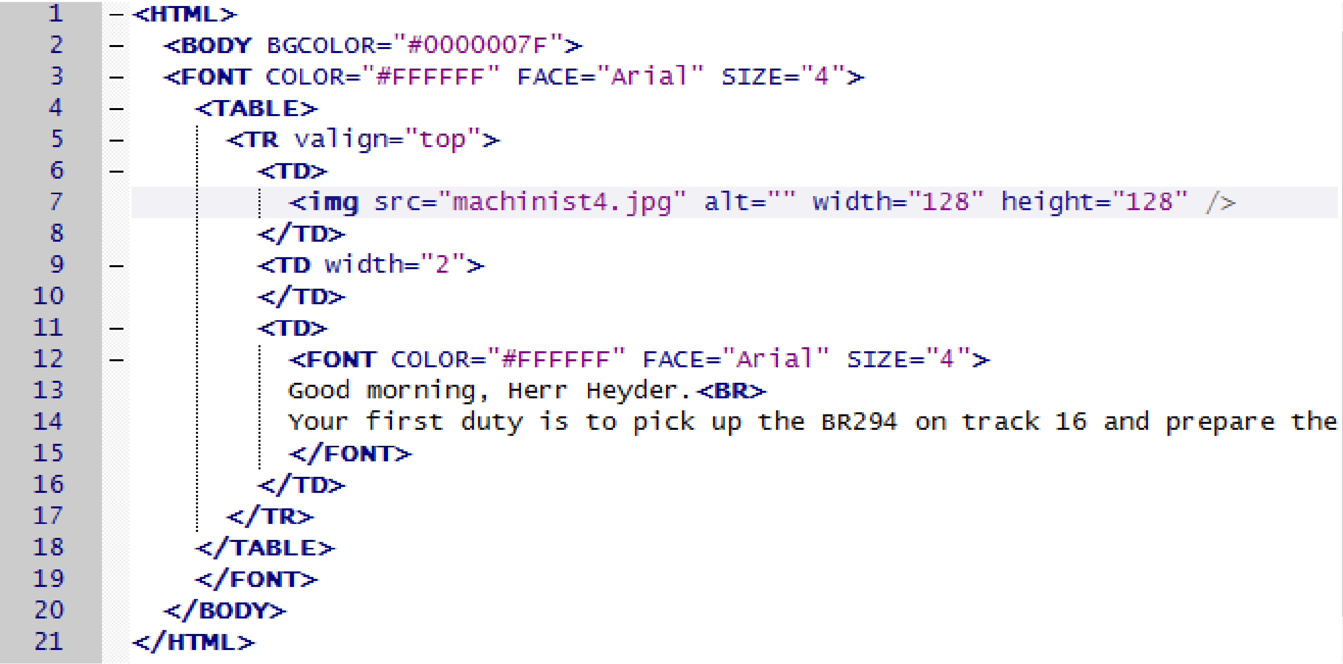
The html sample RSC uses are not of a very high quality. I provide a better sample here:



Good morning, Herr Heyder.
Your first duty is to pick up the BR294 on track 16 and prepare the BR 101 on track 15 for a test run. But first get some fuel.

In my sample you see the top line of the text is properly aligned with image. I also like the image better …

The html code is shown below:



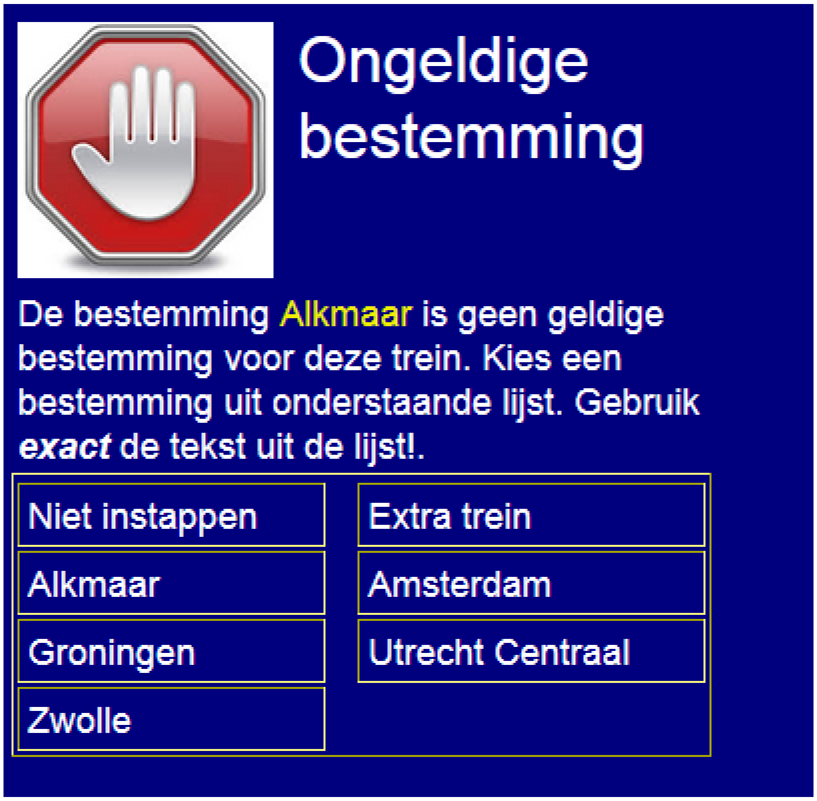
```
 1  <HTML>
 2    <BODY BGCOLOR="#0000007F">
 3    <FONT COLOR="#FFFFFF" FACE="Arial" SIZE="4">
 4      <TABLE>
 5        <TR valign="top">
 6          <TD>
 7            <img src="machinist4.jpg" alt="" width="128" height="128" />
 8          </TD>
 9          <TD width="2">
10          </TD>
11          <TD>
12            <FONT COLOR="#FFFFFF" FACE="Arial" SIZE="4">
13            Good morning, Herr Heyder.<BR>
14            Your first duty is to pick up the BR294 on track 16 and prepare the
15            </FONT>
16          </TD>
17        </TR>
18      </TABLE>
19      </FONT>
20    </BODY>
21  </HTML>
```

The most important trick is to use tables to structure the images and texts.

Line 4 opens a table, with a single row (line 5). The row has two columns, one with the image (line 6-8) and a second one with the text (line 11- 16).

I do not know what kind of stuff the html supports. It would be interesting if would be interesting if it supports javascript, or external content. I did not try. Let me know …

One last sample, showing more complex use of tables:



Ongeldige bestemming

De bestemming Alkmaar is geen geldige bestemming voor deze trein. Kies een bestemming uit onderstaande lijst. Gebruik ***exact*** de tekst uit de lijst!.

Niet instappen	Extra trein
Alkmaar	Amsterdam
Groningen	Utrecht Centraal
Zwolle	

And the html code used:

```
<html>
  <body bgcolor="#0000007F">
     <table width="350">
      <tr valign="top">
        <td width="130">
          <img src="stop.jpg" alt="" width="128" height="128" />
        </td>
        <td width="2">
        </td>
        <td>
          <font color="#FFFFFF" face="Arial" size="6">Ongeldige bestemming
            </font>
                  </td>
      </tr>
            <table>
                  <table width="350">
            <tr>
              <td>
                   <font color="#FFFFFF" face="Arial" size="4">
                              De bestemming <font color="#eeee00" face="arial"
size="4">Alkmaar</font> is geen geldige bestemming voor deze trein. Kies een
bestemming uit onderstaande lijst. Gebruik <b><i>exact</i></b> de tekst uit de
lijst!.
                </font><p></p>
        </td>
      </tr>
       </table>
      <font color="#ffffff" face="arial" size="4">
            <table valign="top" width="350" border="1" bordercolor="#eeee00"
cellpadding="4">
                  <tr>
                        <tr>
                              <td><font color="#ffffff" face="arial" size="4">Niet
instappen</font></td>
                              <td width="2"</td>
                              <td><font color="#ffffff" face="arial" size="4">
                        Extra trein</font></td>
                  </tr>
                  <tr>
                        <td><font color="#ffffff" face="arial"
size="4">Alkmaar</font></td>
                        <td width="2"</td>
                        <td><font color="#ffffff" face="arial" size="4">
                        Amsterdam</font></td>
                  </tr>
                  <tr>
                        <td><font color="#ffffff" face="arial"
size="4">Groningen</font></td>
                        <td width="2"</td>
                        <td><font color="#ffffff" face="arial" size="4">
                        Utrecht Centraal</font></td>
                  </tr>
                  <tr>
                        <td><font color="#ffffff" face="arial"
size="4">Zwolle</font></td>
                        <td width="2"</td>
                        <td></td>
                  </tr>
            </table>
    </font>
  </body>
</html>
```

The html code for this sample is completely generated in lua. As soon as I find out how to store the code in the correct directory, I will use it in my scripts. I intend to create this html on the fly from and display it for TS2014 users.

Note: a way to access the correct folder is to store both the route GUID and the scenario GUID in your script. This can be done but it is not a very elegant method, therefore I did not yet use this.

8 Camera control

TS201x has a separate camera manager. I am aware of two different functions:

a) Controlling the Engine cameras from lua scripting. E.g. you may use it to force cab view.
b) To control the cinematic camera.

Both subjects will be covered in different paragraphs.

8.1 Consist camera functions

The function to activate a camera is:

```
SysCall ( "CameraManager:ActivateCamera", camera, duration );
```

Parameter	Description	Comment
command	Fixed value	
camera	Camera name, see list below	
duration	Time in seconds during which this camera is active	.Any other value will return to the previous camera at the end of the time.

The cameras probably use these names:

Camera	Lua name	Keyboard mapping
Cab camera	CabCamera	1
	ExternalCamera	2
	ExternalCamera	3
Track side camera	TrackSideCamera	4
Passenger view	CarriageCamera	5
Coupling view	CouplingCamera	6
	YardCamera	7
	FreeCamera	8
	DerailmentCamera	
	EditCamera	
	CinematicCamera	
	ShapeViewerCamera	

TODO test all cameras

Example:

```
SysCall ( "CameraManager:ActivateCamera", "CabCamera", 0 );
```

forces Cabview. I do not know what the last parameter means.

You can also use the scenario marker to force cabview, but this does not allow you to switch later on to cab view.

Allowed camera types:

```
CabCamera
FreeCamera
TrackSideCamera
YardCamera
Coupling Camera ( or CouplingCamera?)
HeadOutCamera (seems to work only for left Head out, but maybe extra parameter
needed?)

SysCall ( "CameraManager:ActivateCamera", CameraType, Duration );

Parameters:
```

Allowed camera types:

```
CabCamera
FreeCamera
TrackSideCamera
YardCamera
CouplingCamera
HeadOutCamera
ExternalCamera
```

(HeadOutCamera seems to work only for left Head out, but maybe extra parameter needed?)

Possibly camera instructions:

- ActivateRearCamera
- ActivateHeadOutCamera
- ActivateCarriageCamera
- ActivateTrackSideCamera
- ActivateCouplingCamera
- ActivateYardCamera
- ActivateFollowCamera
- ActivateCabCamera
- ActivateDerailmentCamera
- ActivateFreeCam
- ActivateCamera
- SwitchToNextFrontCab
- SwitchToNextRearCab
-

```
function OnCameraEnter(cabEnd, carriageCam)
```

Function that is called when the player camera enters a cab or passenger view. Parameter `cabEnd` is the cab that the player has entered and is either 1 or 2 and `carriageCam` denotes whether it was a passenger view or not, being 1 if it is a passenger view.

Note: I could not get this working in a scenario script. Maybe it is not exposed in the scenario scripting API.

```
function OnCameraLeave()
```

Function that is called when the player leaves a camera view for another.

Note: same problem: not working for scenarios.

```
if cabside < 1.5 then

            Call("*:SetControlValue", "Frontcabactive", 0, 1)
            Call("*:SetControlValue", "Rearcabactive", 0, 0)

      end

      if cabside > 1.5 then

            Call("*:SetControlValue", "Frontcabactive", 0, 0)
            Call("*:SetControlValue", "Rearcabactive", 0, 1)

      end
```

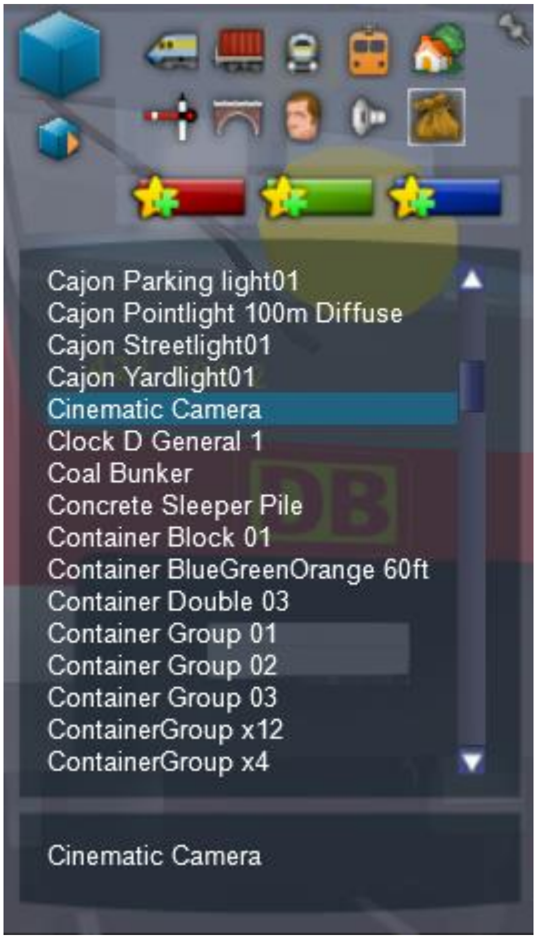
8.2 The cinematic camera

This text is not yet verified!

I've managed to get a cinematic moving camera working, but when it came to the freecamera I couldn't seem to get the co-ords correct and kept ending up miles away! Eventually tracked down that the Long value (second co-ordinate in the top "compass" drop down) goes in the X box, the Lat in the Y and Z is the height!

To use a cinematic camera, you first must select the cinematic camera object from your asset list.

You can find it in the list of miscellaneous assets:



Cajon Parking light01
Cajon Pointlight 100m Diffuse
Cajon Streetlight01
Cajon Yardlight01
Cinematic Camera
Clock D General 1
Coal Bunker
Concrete Sleeper Pile
Container Block 01
Container BlueGreenOrange 60ft
Container Double 03
Container Group 01
Container Group 02
Container Group 03
ContainerGroup x12
ContainerGroup x4

Cinematic Camera

You need to place it in a scenario and give it a name. You can use this name in the commands to control the camera.

```
SysCall ( "CameraManager:ActivateCamera", CinematicCameraName, 0 );
```

Use this to start the cinematic camera created in the scenario editor. This is used in the flying camera at the start of the first Horseshoe Curve scenario.
Video ogg files need to sit in the En folder. Only image files that work seem to be png (tried Gif and jpeg don't work).

Add a trigger event to this train with the default 00:00 time. Give this a name in the event trigger e.g. "StartCam".

Name the camera the same as the trigger event name "StartCam". There is an Object name box. Set the camera offset to 5-10 seconds and add another cinematic camera by clicking the Add control point button (Orange triangle with green cross). This adds another camera to track to.

4) Click the Script button. Next to the red flag and then click reload. This should open the script file provided it is present in the scenario folder. Will error if not found.
You can click the open folder button to open the folder to view/edit the script file.

Looking at the script and a sub section:-

```
------------------------------------------------
-- Fn OnEvent
-- event - name of the event
-- return - TRUE/FALSE if event handled
function OnEvent ( event )

-- Timed triggers
---------------------------

if event == "StartCam" then -- Pan around at train

SysCall ( "CameraManager:ActivateCamera", "StartCam", 0 );
SysCall ( "CameraManager:ActivateCamera", "Getsite", 0 );

return TRUE;
```

Enter the following 3 calls to make the camera point at any object placed in the scenario

SysCall ("CameraManager:ActivateCamera", "FreeCamera", Duration);
SysCall ("CameraManager:JumpTo" , Longitude, Latitude ,Height);
SysCall ("CameraManager:LookAt", "ObjNum or ObjName");

To obtain an object number, double click on the object and look in the right flyout panel.

9 Engine functions

9.1 Generics

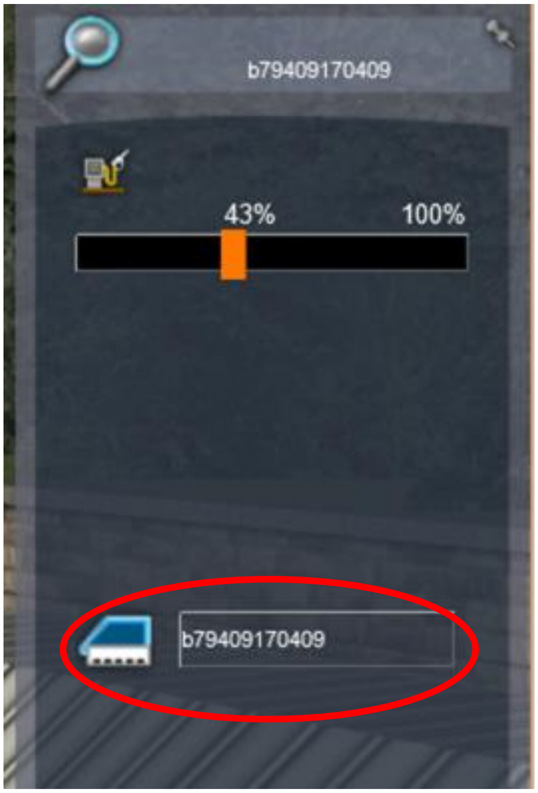
You can refer to an engine in two different ways:

The player engine is called “PlayerEngine” in scripts. AI engines can be referred to with a number, probably the consist number you in the scenario editor (I did not yet test anything for AI trains).

```
SysCall ( "PlayerEngine:SetControlValue", "Startup", 0, -1 );
SysCall ( "8510:SetControlValue", Startup, 0, 0.0 );
```

8510 is the ref to the engine or van, as shows in the right hand fly out when selecting a consist. (see image below).

Note that this makes engine commands work on both player engine and AI, but some commands are useful on coaches as well. A good example is door state monitoring.



Note: in this chapter sometimes the engine is referred with a “*“ symbol. You need to replace the * with the engine to want to use.

It seems that there are three methods to change engine controls:

a) Set a control directly to a value.
b) Toggle a control value between on and off
c) Emulate a gradual increase/decrease of a control value.

It is not obvious which one to use, for instance for a Glare I found a ToggleControl value, while the acknowledge AWS seems to use the third method. It also can be different between engine types. This makes using these commands very hard and time consuming.

9.1.1 Call or SysCall?

I found that in scenario scripts you need to use the SysCall function, not the Call function that is used in engine scripts. The curious thing is that I found one script, that is using Call as well. Maybe my assumption is not always valid, but I have no idea what is behind this.

9.1.2 Set a control directly to a value.

One method is to set them immediately to the desired value. You can do this with command:

```
SysCall("*:SetControlValue", "<ControlValue>", index, value)
```

Method call to set the value of a control that has been specified in the engine blueprint of a locomotive. Parameter **`<ControlValue>`** is the name of the control as it is in the engine blueprint, **`index`** is the index in an array of controls that share the same name (should generally be 0) and **`value`** is the value to set the control to.

For the asterisk you need to replace it with the engine/van/coach etc.

With

```
SysCall("PlayerEngine:GetControlValue", "<ControlValue>", index)
```

You can retrieve a current value.

Method call to retrieve the value of a control that has been specified in the engine blueprint of a locomotive. Parameter `<ControlValue>` is the name of the control as it is in the engine blueprint and `index` is the index in an array of controls that share the same name (should generally be 0).

```
SysCall("PlayerEngine:ControlExists", "<ControlValue>", index)
```

Method that will return 1 if a given control value exists in a blueprint or 0 if not. Parameter `<ControlValue>` is the control value to check, as specified in the engine blueprint and `index` is the index in an array of controls if there are multiple controls with the same name.

Useful function to check the existence for a control first.

```
-- Function to set a named control to the correct value if the control exists -
used to keep code compact.
function SetControl ( engine, name, value )
```

```
      if SysCall( engine .. ":ControlExists", name, 0 ) ==TRUE then
            SysCall( engine .. ":SetControlValue", name, 0, value )
            writeDebug(file,"SetControl success for ".. name)

            return true
      else
            writeDebug(file, "SetControl fails for ".. name)
            return false
      end

end
```

`This function is also mentioned somewhere . No idea what it adds:`

```
SysCall("PlayerEngine:SetEngineValue", "<ControlValue>", value)
```

Method that will set the lead engine's given control value(?) to a specified value. Parameter `<ControlValue>` is the control value to affect, and `value` is the new value to set the control to.

Found this in an engine script:

Call("*:SetEngineValue", "Reverser", controlValue)

9.1.3 Toggle a control value between on and off

I really do not know how to handle this kind of controls. Maybe SetControlValue will work as well. Or ToggleControlValue maybe?

9.1.4 Emulate a gradual increase/decrease of a control value

The idea is that you should use something like "StartIncreaseControl", "StopIncreaseControl", StartDecreaseControl", "StopDecreaseControl" to achieve a gradual setting.

The strange thing is I also saw an example which uses SetControlValue to achieve a throttle setting to zero.

This needs further investigation.

9.2 Engine controls

The big problem is the lack of documentation. Engine functions can be very specific for an model. This offers great flexibility, but if the commands are not documented it can be hard to find out what can be used.

Some examples:

- PantographControl does not work right at the start of a scenario, but if you wait a while, you can use it to lower the pantographs.

- Headlights works sometimes, but I could nog get it working to switch off headlights for a German BR151
- I managed to use SetDestinationBoards on all ChrisTrains models to set destinations.
- I succeeded in using DoorsOpenStateLft and DoorsopenStateRight to get door status
- I succeeded in getting AWS state, but could not control the beviour of AWS, e.g. set AWS works, but you cannot use the Q button to clear it.
- I did not succeed in sett Wipers on or off in the same BR151.
- For Reverser it works perfectly well, setting a value.
- Following paragraphs show the names of default controls used. Please note, these can be customized.

You may want to look for files like Inputmapper to discover the control names to use. In the appendix I included an unverified list.

Courtesy dtrainBNSF1

```
In the lua script you can use this to turn off the engine:
if event == "(customized name here)" then
    SysCall ( "PlayerEngine:SetControlValue", "Startup", 0, -1);
    return TRUE
end
```

One thing to keep in mind: when running with multiple engines the SysCall will shut off your engine and while you are in control of that engine it will shut off the other engines in the consist. But if you switch controls to a different loco in the consist the all of the engines will turn on again. If you switch back to your original engine then they'll all shut off again. Perhaps to shut off all the others where it says "PlayerEngine" replace it with the other loco's numbers (so if I was driving loco 4569 that would be "PlayerEngine"; if my 2nd engine in the consist was 4723 then type in "4723:SetControlValue" etc. in the next line). I haven't tested that out yet but I probably will in the future.

This sample limits the cruisecontrol for German loco’s (e.g. Munich-Augsburg, IN+C3, BR101 etc) to 40 km/h (Not tested by me).

```
CurrentAFB = SysCall( "PlayerEngine:GetControlValue", "AFB", 0 );
if (CurrentAFB > 0.15) then
     SysCall ( "PlayerEngine:SetControlValue", "AFB", 0, 0.15);
end
```

9.3 Track information

```
SysCall(“*:GetGradient”)
```

Method that will return the grade of the track that the consist is currently on.

```
SysCall(“*:GetCurvature”)
```

Method that will get the curvature of the track that the consist is currently on.

```
SysCall("*:GetCurvatureAhead", distance)
```

Method that will get the curvature of the track at a given distance in front of the consist. Parameter `distance` is the distance ahead to look in metres. It is measured as the reciprocal of the radius of the curve (1/radius) such that the curvature of a straight line is 0.

9.4 Consist information

```
SysCall("*:GetConsistLength")
```

Method that will return the length of the consist at that given point in time, in metres.

```
SysCall("*:GetConsistTotalMass")
```

Method that will get the total mass of the consist that the entity is in, inclusive of ballast/fuel. mass is measured in tons.

```
SysCall("*:GetTotalMass")
```

Method that will get the total mass of the entity that is calling this method, inclusive of ballast/fuel, measured in tons. Will only work within an engine script/blueprint, and not a wagon script/blueprint.

```
SysCall("*:GetFireboxMass")
```

Method that will retrieve the mass of the firebox as a fraction of its maximum value. (Not checked)

```
SysCall("*:GetRVNumber")
```

Method that will return the number of the entity that called this method. The number is the attribute that can be set in the scenario editor. It returns the number of the first van/engine in a consist. It seems not to be indexed, so you cannot retrieve other vans.

9.5 Identification and information

Functions not checked for scenarios

```
SysCall("*:GetIsPlayer")
```

Method call that will return if the consist is the player consist. 1 is returned if true, else 0 is returned.

```
SysCall("*:GetIsDeadEngine")
```

Method that will return if the engine is specified "dead" as an attribute that has been set in the scenario editor. If true 1 is returned, else 0.

```
SysCall("*:GetIsEngineWithKey")
```

Method that will return 1 if the engine is being driven by the player, otherwise it will return 0.

9.6 Speed

These functions all work properly. Thomas Ross uses them in a script for Western Lines of Scotland to monitor speed limits for steam engines.

```
SysCall("*:GetSpeed")
```

Method call that will return the speed the player is undergoing at that specific point in time, measured in m/s.

```
SysCall("*:GetCurrentSpeedLimit")
```

Method call that will retrieve the speed limit of the section of track that the consist is currently on, measured in m/s.

```
SysCall("*:GetNextSpeedLimit", direction)
```

Method call that will retrieve the next speed limit in a given direction within 10km of the entity in the consist. Parameter `direction` is the direction to check, with 0 for forward and 1 for backwards. Returns 3 pieces of data; `type`, `speed` and `distance`. `type` indicates the cause of restriction on the line, `speed` is the new speed restriction, measured in m/s and `distance` is the distance in metres that are remaining until the new speed limit. The values that `type` can undertake are:

- 0 - indicating end of line,
- 1 - indicating speed change but no linked trackside sign,
- 2 - indicating speed change with linked trackside sign,
- -1 - indicating no change in speed over the next 10km.

Not verified

```
SysCall("*:GetAcceleration")
```

Method call that will retrieve the acceleration the player is undergoing at that specific point in time, measured in m/s^2.

9.7 Signalling interaction

9.7.1 Signals

Not tested yet.

```
SysCall(“PlayerEngine:GetNextRestrictiveSignal”, direction, distance,
lookfromdistance)
```

Method call that will gather information about the next non-green signal (such that it is not “clear”). Parameter **`direction`** is the direction to look, with 0 for forward and 1 for backward.

Distance is the distance in meters to the signal

lookfromdistance is how far ahead of the player train to start looking. For example if you are sitting at a red light, you can set lookfromdistance to 100 to tell the script to start looking for the next restrictive signal from 100m ahead of you - so you can look "through" the red light where you are stopped.

Returns 4 values; `type`, `state`, `distance` and `aspect`. `type` is representative of the values that follow it, and will carry a range of values itself. `state` indicates the state of the signal in the signalling system; 0 is “clear”, 1 is “warning” and 2 is “stop”. `distance` is the distance, in metres, until the signal has been reached and `aspect` is the aspect set by the signal script (that is relative to `“Set2dMapProSignalState”`). For UK signals this can be:

0 - indicates signal is green,

1 - indicates signal is single yellow,

2 - indicates signal is double yellow,

3 - indicates signal is red,

10 - indicates signal is flashing single yellow,

11 - indicates signal is flashing double yellow.

The values that can be used by `type` are as follows:

-1 - indicates there are no restrictive signals within 10km of the consist entity,

0 - indicates end of line (such that `state` & `aspect` can be ignored),

1 - indicates there is a restrictive signal on the line within 10km, and as such, `state`, `distance` & `aspect` should be checked for more info.

Note that `state` and `aspect` shouldn’t be 0 as they indicate that the signal is not restrictive.

9.7.2 AWS functions

There are three AWS functions that can be used:

```
AlertReset = SysCall(self.Consist .. ":GetControlValue", "AWSReset", 0 )
AlertSound = SysCall(self.Consist .. ":GetControlValue", "AWSWarnCount", 0 )
AWSState = SysCall(self.Consist .. ":GetControlValue", "AWS", 0 )
```

“AWSReset” probably refers to the grace time for the driver to acknowledge an AWS alert.

“AWSWarnSound” has value 1 as long as the AWS alarm is on and not acknowledged.

“AWS” is the state of AWS, It’s value remains at 1 as long as alert status is needed.

You can read the value but also set it (not tested yet). An interesting application is to make AWS alerts outside the cab audible. See my script application in chapter 11.4.

I also tried to create a driver alert function using this, but it does not work properly, you cannot use the Q button to reset AWS if you activate it from a scenario.

10 Other functions

10.1 Time and season

```
local SecondsAfterMidnight= SysCall ("ScenarioManager:GetTimeOfDay")
```

```
local Season = SysCall ("ScenarioManager:GetSeason") –
```

```
Spring = 0, summer = 1, autumn/fall = 2, winter = 3
```

```
local timeNow = SysCall("*:GetSimulationTime", 0)
```

Day and night times, depending on season:

The seasons are as such: Spring- 1845 to 0815. Summer- 2100 to 0530. Fall- 1945 to 0645 (fall is different than spring). Winter- 1700 to 0915. The nodes work perfectly in winter, after the 'else' statement in the local season section. The bad news, is that no matter what season it is in the scenario, the script seems to only run on the winter time slot. Almost like it just ignores the 0, 1 and 2 season sys calls.

`(sources dogrokket see also http://forums.uktrainsim.com/viewtopic.php?f=360&t=136967&st=0&sk=t&sd=a&start=15 )`

10.2 Whistle function

I have not tried this one yet and have no idea what whistle1 is :

```
SysCall ( "whistle1:SetParameter", "ScenarioTrigger", 1.0 );
```

10.3 Play video

For video add the following line.

```

SysCall ( "ScenarioManager:PlayVideoMessage","0001-0250.ogg",0,1,0,0);
```

Ensure the ogg file is in the en folder. The various 0,1,0,0) seem to control things like video screen size, playback with sound, and if the script waits for the video to play first then continues.

10.4 Play audio

```
SysCall ( "ScenarioManager:PlayDialogueSound",audiofilename);
```

The file is looked up relative to

```
Content/Routes/<route-uuid>/Scenarios/<scenario-uuid>/en/.
```

You can escape from that path using the appropriate number of "../" sequence at the beginning of the path.

For some reason underscore in the file name seem to result in the file being ignored.

The file must be in the wav format (that is not the encoded form that use in other parts of the sim).

Maybe the folder name is case sensitive in lua.

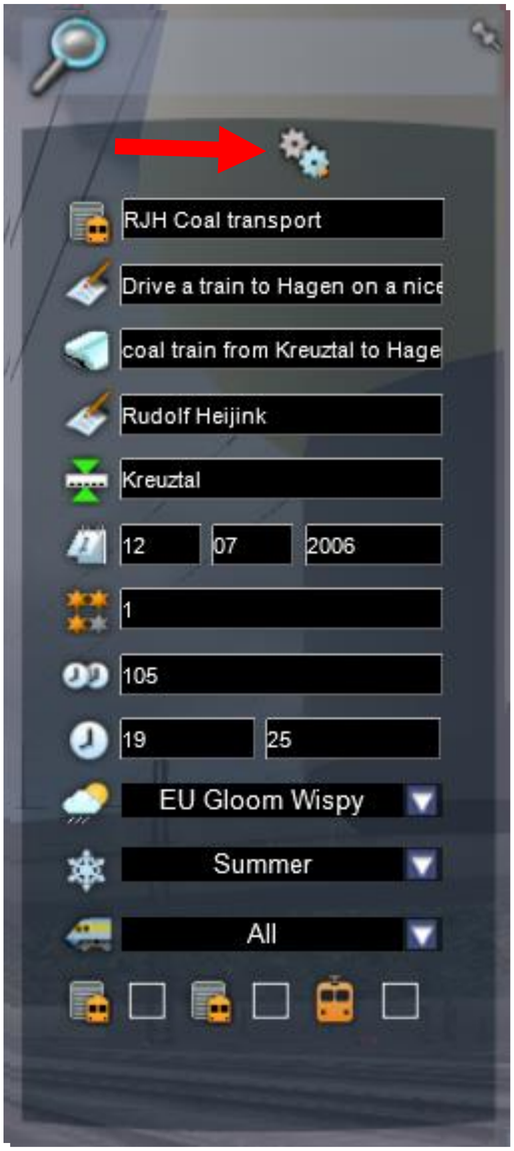
Chris Longhurst says: “Seems it was the capitalisation it didn't like. I had a folder called "En". When I changed that to "en" everything started working.”

10.5 Weather events

Call that allows to change weather conditions. Not sure how this works. You probably need to define a weather blue print.

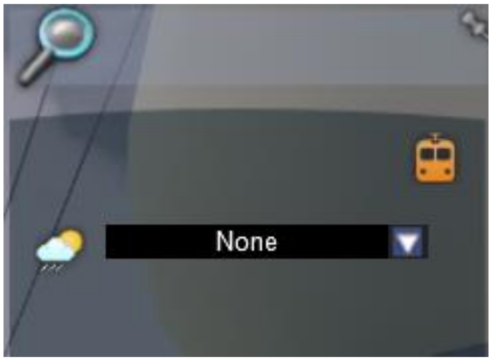
```
SysCall ( "WeatherController:SetCurrentWeatherEventChain", "FirstChain" )
```

You will need to set the extended weather option in the scenario.



RJH Coal transport
Drive a train to Hagen on a nice
coal train from Kreuztal to Hage
Rudolf Heijink
Kreuztal
12 07 2006
1
105
19 25
EU Gloom Wispy
Summer
All
☐ ☐ ☐

Click at the gear wheels and the select the extended weather.



None

I do not yet know how to create a proper working blueprint supporting extended weather. Coming soon …

From Railworks America (do not know who wrote this)

ok so i watch video of scenario called under the storm. and i like how guy made that thunder and lighting storm. https://www.youtube.com/watch?v=mzYIpaequQ4
so i was looking around on how he did that and found this topic
http://forums.steampowered.com/forums/s ... 475&page=4.
so i follow what he said to make that storm for my scenario but thunder storm i'm using is from sherman hill so it little different like i don't see a thunder in that advance weather list that he used to make it work. and i play scenario i made and i get the change clouds and sky but no lighting or thunder or rain.

Source RWA user dtrainBNSF1

Figured out how to get the weather working for WLoS and Sherman Hill by the way :) For WLoS it's just as you said in the guide. For Sherman Hill you need the code you mentioned in the guide and you need to have "SH Clear to Thunderheads" and "Thunderheads" selected in game. So far testing has shown that it won't work for any other combination. Sherman Hill has 3 storms, "FirstChain", "SecondChain" and "FinalStorm". "FirstChain" is just a sprinkling, not much and lasts for 10 minutes then fades. "SecondChain" begins like "FirstChain" but after 5 minutes intensifies into a downpour then at the end of its 20-minute cycle reverts back to "FirstChain" and fades out. "FinalStorm" is a full on thunderstorm and begins within seconds after its code activation for Sherman Hill but for WLoS the storm begins a few minutes after code activation. For both routes the storm lasts for 20 minutes then fades out. The Sherman Hill storm has more rain and actually causes the screen to darken. Because of how quickly the weather for Sherman Hill activates it would be best to wait for about 15 minutes into the scenario before activating any of the storms to allow the skies to animate properly and put the thunderheads in place otherwise the weather looks like something from a SciFi flick and doesn't convey the type of realism I'd personally expect. Because of the time delay of the storm for WLoS the thunderheads conceivably already move into proper position before the storm begins so you can start the scenario with the "FinalStorm" code on WLoS with no problem.
Another thing to note is that the storms DO NOT mesh together if you activate one after the other. The simulator causes a short break between the storms before activating another storm, clearing the skies for a few seconds. Not too realistic.
As expected if you want to use these storms on other routes the appropriate blueprints have to be selected in the scenario editor.

11 Scripting applications

11.1 Logging your debugging messages

Tested, This works properly

See this site for a short tutorial on the object oriented techniques That are used in this example.

```
RJHDebugversion="0.1 alfa"

DebugTS = {} -- the table representing the class, which will double as the
metatable for the instances
DebugTS.__index = DebugTS -- failed table lookups on the instances should fallback
to the class table, to get methods

-- syntax equivalent to "DebugTS.new = function..."
function DebugTS.new(logfile,mode)
  local self = setmetatable({}, DebugTS)
  if logfile== nil then
      self.logfile="logfile.txt" -- file name
  else
      self.logfile=logfile
  end
  if mode==nil then
    self.mode="a+" -- open mode
   else
     self.mode=mode
  end
-- You can find this file in the TS home directory
  self.debugfile= io.open(self.logfile,self.mode) --file handler for debug file
  self.writeDebug(self,"Debug script version " .. RJHDebugversion .. " loaded")

  return self
end

-- write debug message
-- message is a string containing a text message

      function DebugTS:writeDebug(message)
            local dt= os.date("%d-%m-%Y/%X ")
            if(dt==nil) then
                  dt=""
            end
            self.debugfile:write(dt .. message ..'\n')
            self.debugfile:flush()
      end
```

Note: You can find your logfile in the TS folder. I store the script file with the name debug.lua in my assets folder.

Usage in a scenarioscript file

```
scriptpath=".\\assets\\RudolfJan\\lua\\"

dofile(scriptpath .. "debug.lua")
```

```
function Initialise()
      mydebug=DebugTS.new()

end -- Initialise

function OnEvent(event)

      if event=="start" then

            mydebug:writeDebug("Wow, it works")

      end --start
end --OnEvent
```

11.2 Train length and train mass

Tested. This works properly.

```
function GetConsistMass(consist)
      result= SysCall(consist ..':GetConsistTotalMass' )
      if(result==nil) then
            result= 0
      end
      return result
end

function GetConsistId(consist)
      result= SysCall(consist .. ':GetRVNumber')
      if(result==nil) then
            result= 0
      end
      return result
end

function ReportConsistData(consist)
      if (consist==nil) then
            consist="PlayerEngine"
      end

      length= GetTrainLength(consist)
      mass= GetConsistMass(consist)
      id= GetConsistId(consist)

      if(mass==0 or length==0) then
                  SysCall ( "ScenarioManager:ShowInfoMessageExt", "Train report",
"Error, train report not available", 30, MSG_TOP+MSG_LEFT, MSG_SMALL, TRUE);
      else
            message= string.format("%s%s%s%d%s%d%s ","Train number= ",id," Consist
length= ",length," metres, consist mass = ",mass," tons")
            SysCall ( "ScenarioManager:ShowInfoMessageExt", "Train report",
message, 30, MSG_TOP+MSG_LEFT, MSG_REG, TRUE);
      end
end
```

11.3 Destination boards for ChrisTrains models

This is a much more complex example. It is tested and works in principle, but I do not present a complete example here.

ChrisTrains uses an index to pick up the correct destination. I prefer to use station names. The index numbers change from time to time, by using the table it is easy to change it for all scenarios where you use this. I also try to make the script more general. I hope it also works for other providers.

Create a file with a list of destinations. This file should be named <consisttype><provider>Destinations.lua

```
-- For each consist type specify the provider and consist, these must match the
parts in the filename
provider = "ChrisTrains"

-- Note, version 1.1 of the consist or higher is assumed
consist = "SLT"

-- Look for a file called "Inputmapper" This files gives you clues about the
command name to use.
setDestinationCommand = "DestinationBoards"

-- The first index number to be used. In most cases I assume this will be 0 or 1
indexStart = 1

-- You need to fille the table with all destinations.
destinationList= {
"Niet Instappen",
"Extra Trein",
"Alkmaar",
"Almelo",
"Almere Centrum",
"Almere Oostvaarders",
"Amersfoort Schothorst",
"Amsterdam Centraal",
"Apeldoorn",
"Arnhem",
"Baarn",
"Breda",
"Den Haag Centraal",
"Den Haag HS",
"Den Helder",
"Deventer",
"Dordrecht",
"Ede-Wageningen",
"Eindhoven",
"Enkhuizen",
"Enschede",
"Geldermalsen",
"Gouda Goverwelle",
"Groningen",
"Haarlem",
"Heerlen",
"Helmond",
"Hengelo",
"Hoek van Holland Haven",
"Hoek van Holland Strand",
"Hoofddorp",
"Hoorn",
"Hoorn Kersenboogerd",
"Leeuwarden",
"Leiden Centraal",
"Lelystad Centrum",
"Maassluis West",
"Maastricht",
"Maastricht Randwyck",
"Nijmegen",
"Rhenen",
"Rijswijk",
"Roermond",
"Roosendaal",
"Rotterdam Centraal",
"Schiedam Centrum",
"Schiphol Airport",
"Sittard",
"Tiel",
"Tilburg Universiteit",
"Uitgeest",
"Utrecht Centraal",
"Utrecht Maliebaan",
"Venlo",
"Vlaardingen Centrum",
"Vlissingen",
"Weert",
"Weesp",
"Woerden",
"Zandvoort aan Zee",
"Zwolle"
}

if TEST==1 then
      writeDebug(file, "Destinations loaded for "..provider.." "..consist)
end
```

I stored these files in an assets folder, so they are common for all scenarios.

The destinations are stored in a destination list. To get information from the list, you need following three functions:

```
-- function to iterate over a table and retieve its idex (i) and value (v)
-- Source: Programming Lua v5.2

local function iter (a, i)
      i = i + 1
      local v = a[i]
      if v then
            return i, v
      end
end

-- function that returns the next index, value pair of an associative array
-- Source: Programming Lua v5.2

function ipairs (a)
      return iter, a, 0
end

-- function to fill associative array with index values.
-- returns number of destinations

function createIndex(a)
      for i, v in ipairs(a) do
                  a[v]=i+indexStart-1
                  print(v," ", a[v])
      end
      return i
end
```

The last one translates picks up the index value for a specific destination, so you can use the name of the destination. These three functions are very generic, and are stored in a file called “common.lua” This also goes in the asset folder.

The function that does the real work is this one:

```
-- Function to set destination boards

function SetDestination(engine, provider,consist, destination)

            writeDebug(file,"Destination=".. destination)
            local assetpath=".\\"

            assetpath=".\\assets\\RudolfJan\\lua\\"
            local path= assetpath .. provider..consist.."Destinations.lua"
            writeDebug(file,"Destinations list file: ".. path)


            -- retrieve destinationlist

            -- verify file exists

            handle=io.open(path)
            if handle==nil then
                -- display error message and exit
                              writeDebug(file,"Destinations list file not found:
".. path)
                        return FALSE
            else
              writeDebug(file,"Destinations list file found: ".. path)
              io.close(handle)
            end

            dofile(path)

            count=createIndex(destinationList)
            if destinationList[destination] == nil then
                writeDebug(file,"Destination " .. destination .. " not found")
```

```
                  return FALSE
            end

            SetControl ( engine, setDestinationCommand,
destinationList[destination]);
end
```

As you can see it does a number of checks, reads the destinations file and sets the correct control. The control name in parameterised, increasing the chance to make it work for other providers.

In your script file, you need to include this code (example):

```
-- Scenario script to set destination boards
-- (C) 2014 Rudolf Heijink
-- Version 0.2 alfa

-- common lua scripting functions and constants

dofile(".\\assets\\RudolfJan\\lua\\common.lua")


function OnEvent ( event )
      if (event == "start") then
            SetDestination("ChrisTrains", "SLT", "Rhenen")

      end
end
```

Note in common.lua also stuff like debug support is added.

11.4 AWS audible alert outside the cab view

Tested and it works.

```
--[[
AWS.lua
Object class for audible AWS alerts
(C) 2014 Rudolf Heijink, all rights reserved.
Requires common.lua
Required debug.lua
--]]


RJHAWS="0.1 alfa New"
mydebug:writeDebug("AWS script version " .. RJHAWS .. " loaded")


-- AWS alert states
AWS_INIT =0
AWS_OFF =1
AWS_ON =2  --AWS active

params=
      {
      "PlayerEngine",
      TRUE,
      "buzzer.wav"
      }

DEBUG=true
Print("AWS script version " .. RJHAWS .. " loaded")

AWSControl = {} -- the table representing the class, which will double as the
metatable for the instances
AWSControl.__index = AWSControl -- failed table lookups on the instances should
fallback to the class table, to get methods


function AWSControl.new(consist, buzzer, buzzerfile)
  local self = setmetatable({}, AWSControl)
  Print("In AWS new")
  self.state=AWS_INIT
  self.Consist=consist or "PlayerEngine"
  self.Buzzer=buzzer or TRUE
  self.BuzzerFile= buzzerfile or ""
  self.ConditionName= "AWSAlert"
  self.AWSstate =0
  self.AlertReset=0
  self.AlertSound=0
  Print("AWS control created");
  return self
end

function AWSControl.BeginChecking(self,ConditionName)
      self.ConditionName= ConditionName or "AWSAlert"
      SysCall ("ScenarioManager:BeginConditionCheck", self.ConditionName );
      mydebug:writeDebug("AWSControl begin checking")
end


function AWSControl.StopChecking(self)
      SysCall ("ScenarioManager:EndConditionCheck", self.ConditionName );

      mydebug:writeDebug("AWScontrol stop checking")
end



function AWSControl.Check(self)
      AlertReset = SysCall(self.Consist .. ":GetControlValue", "AWSReset", 0 ) or -
1
      AlertSound = SysCall(self.Consist .. ":GetControlValue", "AWSWarnCount", 0 )
or -1
      AWSState = SysCall(self.Consist .. ":GetControlValue", "AWS", 0 ) or -1
      local changed=false
      local buzz=false

      if AlertSound==1 and self.AlertSound==0 then
            buzz=true
      end
      if buzz then
            SysCall ( "ScenarioManager:PlayDialogueSound",self.BuzzerFile);
      end
      if AlertReset ~= self.AlertReset then
            self.AlertReset=AlertReset
      end
      if AlertSound ~= self.AlertSound then
            changed=true
            self.AlertSound=AlertSound
      end
      if AWSState ~= self.AWSState then
            changed=true
            self.AWSState=AWSState
      end
      if changed then
            Print("AWS status =" .. AWSState .. " AlertSound = " .. AlertSound .. "
AlertReset = " .. AlertReset)
            mydebug:writeDebug("AWS status =" .. AWSState .. " AlertSound = " ..
AlertSound .. " AlertReset = " .. AlertReset)
            changed=false
      end
      return CONDITION_NOT_YET_MET
end
```

```
--[[
 Scenarioscript for testing AWS and signalling alert functions
 (C) 2014 Rudolf Heijink
 Version 0.1 alfa
 ]]--


CONDITION_NOT_YET_MET = 0
CONDITION_SUCCEEDED = 1
CONDITION_FAILED = 2

scriptpath=".\\assets\\RudolfJan\\luadev\\"

dofile(scriptpath .. "debug.lua")
mydebug=DebugTS.new()

dofile(scriptpath .. "common.lua")
dofile(scriptpath .. "AWS.lua")

function OnEvent(event)
      if event =="start" then
            aws = AWSControl.new(unpack(params))
            aws:BeginChecking("AWSAlert")
            Print("AWS start event")
            return TRUE
      end
      return FALSE
end

function OnResume()
      Print("Calling OnResume")
      aws=AWSControl.new(unpack(params))
      aws:BeginChecking()
```

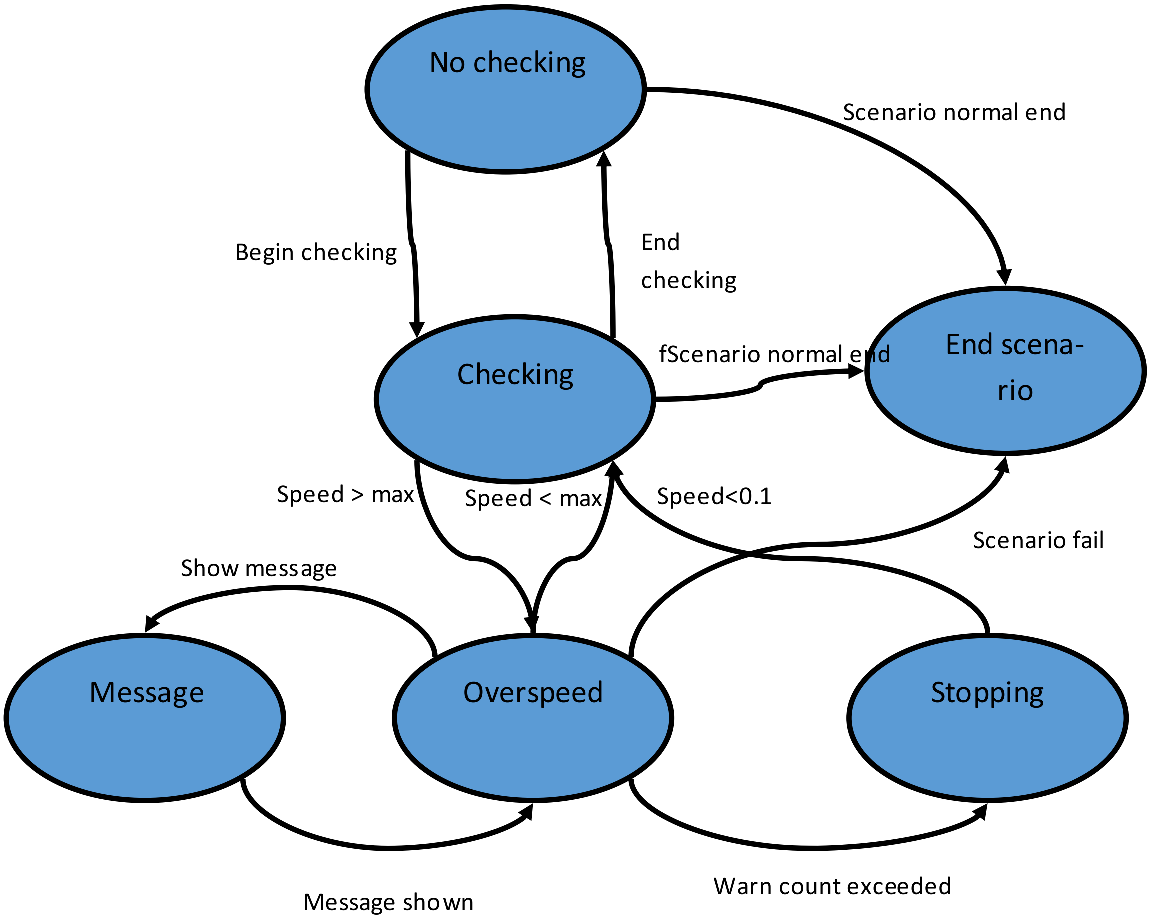
```
end

function TestCondition ( condition )
      if condition == aws.ConditionName then
            aws:Check()
            return CONDITION_NOT_YET_MET
      end
      return CONDITION_NOT_YET_MET
end
```

Things to do:

1. Make a difference between inside cab and outside cab. Does not work, because I cannot get the current camera view.

11.5 Overspeed detection

The overspeed detection sample is inspired by Thomas Ross, who used this for some scenarios for the Western Lines of Scotland Route. However, it is completely rewritten to make it more generic and flexible.



To understand the design, please look at figure … above.

A scenario starts in the state **Nochecking**. In this state you can set up the way the speed checking works. You can apply following settings to customize the script:

Parameter	Explanation
Limit	Sets the speed limit in either km/h or Mph
Units	Choose between the constants KMPH or MPH to use the correct units
Excess	A factor that gives a tolerance. Limit is multiplied by Excess to get the speed limit above which you are considered to be in overspeed
MaxWarnings	The maximum number of warnings. It can be zero or higher. If after the last warning the player is still speeding, an emergency stop will be issued.
Fail	If true, Scenario will fail if after one emergency stop you are in overspeed again, or if the penalty limit exceeds a predefined limit
MaxPenalty	When in overspeed, the player builds up a penalty score wich is number of seconds in overspeed times actual speed minus speedlimit. PenattyLimit

	defines how much penalty is allowed before a warning, emergency stop or fail is issued. Sensible values may be somewhere in the range of 100-500
TotalMaxPenalty	The total penalty allowed, if exceeded, the scenario fails.
FailMessage	Text shown on scenario end in case it fails
SuccessMessage	Text shown on scenario end for successful scenario
Buzzer	Sound a buzzer for each overspeed event
BuzzerFile	Sound file (.wav) to play as buzzer sound

You can define the parameters in an array:

```
params=
 {
 10,    -- limit
 KMPH,  -- units
 1.05,  -- excess
 3,     -- MaxWarnings
 TRUE,  -- fail
 200,   -- maxpenalty
 1000,  -- total penalty allowed before fail
 "Failed. You failed to complete the scenario",  -- fail message
 "Success. You completed the scenario succesfully" -- success message
 TRUE   -- sound buzzer
 "buzzer.wav" -- buzzer file name
 }
```

This is a convenient way to add many parameters to a script.

You need to create a speedcontrol using the new function:

```
speedcontrol=Speedcontrol.new(self, table.unpack(params))
```

This creates a Speedcontrol object in lua named speedcontrol sets up the state Nochecking and initialises the monitoring function.

If you want to use a buzzer, create a folder named “en” in your scenario folder. In this folder store the .wav file containing your buzzer sound file.

To start checking overspeed, you need to call the function

```
speedcontrol:BeginChecking()
```

Now the state **Checking** is active.

If the current consist speed ,exceeds limit*excess, the state changes in **Overspeeding.**

During overspeeding, several things may happen:

1. If you decrease the speed enough, the state is changed back to **Checking**.

2. If you build up MaxPenalty first warning messages will be displayed on screen, provided the number of warnings is less than MaxWarnings
3. If MaxWarnings is reached, an emergency stop is issued. This deactivates all controls, so the player experiences a real emergency stop.
4. If you have set the Fail parameter to true, on the next time you exceed maxPenalty, the scenario fails.

So now let's introduce the code:

The ScenarioScript.lu file is your basic interface to create new scripts using this function:

```
-- Sample script file for speed control function
-- (C)2014 Rudolf Heijink
-- version 0.1 alfa

-- load some script files

TRUE=1

CONDITION_NOT_YET_MET = 0
CONDITION_SUCCEEDED = 1
CONDITION_FAILED = 2

scriptpath=".\\assets\\RudolfJan\\lua\\"

dofile(scriptpath .. "debug.lua")
mydebug=DebugTS.new()
dofile(scriptpath .. "common.lua")
dofile(scriptpath .. "SpeedControl.lua")

-- You can supply all parameters here, by changing values with your own values
params=
 {
 10,    -- limit
 KMPH,  -- units
 1.05,  -- excess
 3,     -- MaxWarnings
 TRUE,  -- fail
 200,   -- maxpenalty
 1000,  -- total penalty allowed before fail
 "Failed. You failed to complete the scenario",  -- fail message
 "Success. You completed the scenario succesfully", -- success message
 TRUE,  -- sound buzzer
 "buzzer.wav" -- buzzer file name
 }

function OnEvent(event)

      if event=="start" then
            speedcontrol=SpeedControl.new(unpack(params))
            speedcontrol:BeginChecking()
            return TRUE
      end --start

      if event==" stop" then
```

```
        speedcontrol:StopChecking()
        return TRUE
      end
      if event==" end" then
        speedcontrol:EndGame()
        return TRUE
      end
      return FALSE
end -- OnEvent

function TestCondition ( condition )
      if condition == speedcontrol.gConditionName then
            speedcontrol:Check()
            return CONDITION_NOT_YET_MET
      end
      return CONDITION_NOT_YET_MET
end
```

Steps:

Load all required script modules

1. Set up your parameters by editing the contents of the params array
2. Create the Speed Monitor object and start checking (here in the event handler for “start”)
3. Stop monitoring (for instance at the last event in your scenario), here in a separate stop event.
4. In the end event you evaluate the performance with respect to adhering to speed restrictions and end the scenario
5. Finailly, you need to create a TestCondition function.

The good news is, you can do that without understanding all details of this fairly complicated script.

The script itself goes in the ***SpeedControl.lua*** file. For discussion< I split it up in the individual functions.

```
--[[
Speedcontrol.lua
Object class for overspeed check and penalties for TS2014
(C) 2014 Rudolf Heijink, all rights reserved.
Based on an idea of Thomas Ross but completely rewritten
Requires common.lua
Required debug.lua
--]]


RJHSpeedControlversion="0.1 alfa New"
mydebug:writeDebug("SpeedControl script version " .. RJHSpeedControlversion .. "
loaded")

KMPH=0
MPH=1

-- state values for speed control
NOCONTROL=0
CHECKING=1
OVERSPEED=2
```

```
PENALTY=3
STOPPING=4
```

The first part of the script sets up debugging support and some constants.

Then you need to create a SpeedControl object. Objects are maintained in a special table. See chapter E.3 for details).

```
SpeedControl = {} -- the table representing the class, which will double as the
metatable for the instances
SpeedControl.__index = SpeedControl -- failed table lookups on the instances should
fallback to the class table, to get methods

function SpeedControl.new(speedlimit, unit, excess, maxWarnings,canfail,
maxpenalty, totalmaxpenalty, failmessage, successmessage, buzzer, buzzerfile)
  local self = setmetatable({}, SpeedControl)
      self.SpeedLimitDisplay= speedlimit or 40
    self.Excess= excess or 1.05-- allowed percentage overspeed
      self.MaxWarnings=maxWarnings or 3
      self.unit= unit or KMPH
      mydebug:writeDebug("Units value=" .. self.unit)
      -- conversion factors of m/s to km/h or Mph
      if (self.unit== KMPH) then
            self.conversion= 3.6
      else
            self.conversion= 2.236932
      end
      self.gConditionName="SpeedCondition"
      self.SpeedLimit= self.SpeedLimitDisplay*self.Excess
      self.CurrentPenalty=0
      self.CurrentPenaltyLimit= maxpenalty or 200
      self.CanFail=canfail or TRUE
      self.Fail=FALSE
      self.TotalPenaltyLimit= totalmaxpenalty or 1000
      self.TotalPenalty=0
      self.PenaltyState=P_NOPENALTY
      self.PenaltyEnd=FALSE
      self.Warnings=0
      self.StartedSpeeding=0
      self.FailMessage= failmessage or "Scenario fails"
      self.SuccessMessage=successmessage or "Senario success"
      self.Buzzer= buzzer or FALSE
      self.BuzzerFile= buzzerfile or ""
      mydebug:writeDebug("Speedcontrol created")
      self.CurrentSpeed=0
      self.state=NOCONTROL
  return self
end
```

As you see, this function assures all variables are initialized properly. This function does not start the actual monitoring.

```
function SpeedControl.BeginChecking(self,ConditionName)
      self.state=CHECKING
      self.ConditionName= ConditionName or "SpeedCondition"
      SysCall ("ScenarioManager:BeginConditionCheck", self.ConditionName );
      mydebug:writeDebug("Speedcontrol begin checking")
end
```

The function BeginChecking sets the initial checking state, and starts monitoring for the appropriate condition.

The StopChecking function does the opposite:

```
function SpeedControl.StopChecking(self)
      self.state=NOCONTROL
      SysCall ("ScenarioManager:EndConditionCheck", self.ConditionName );

      mydebug:writeDebug("Speedcontrol stop checking")
end
```

The main function for overspeed checking is the Checking function:

```
function SpeedControl.Check(self)
      self.CurrentSpeed = SysCall( "PlayerEngine:GetSpeed")
      self.CurrentSpeed=self.CurrentSpeed*self.conversion

      if self.state==CHECKING then

            if self.CurrentSpeed> self.SpeedLimit then
                  self.state=OVERSPEED
                  self.StartedSpeeding = Call("*:GetSimulationTime", 0)
                  mydebug:writeDebug("Start overspeed " .. self.CurrentSpeed)
            end
            return
      end
      if self.state==OVERSPEED then
            self:Overspeed()
      end
      if self.state== PENALTY then
            self:Penalty()
            return
      end
      if self.state== STOPPING then
            self:EmergencyStop()
            return
      end
end
```

It will call more specialized functions in case overspeed is detected.

```
function SpeedControl.Overspeed(self)
      if self.CurrentSpeed> self.SpeedLimit then
            CurrentTime= Call("*:GetSimulationTime", 0)
            self.CurrentPenalty=self.CurrentPenalty+(CurrentTime-
self.StartedSpeeding)*(self.CurrentSpeed-self.SpeedLimit)
            self.StartedSpeeding=CurrentTime

            if self.CurrentPenalty> self.CurrentPenaltyLimit then
                  self.state=PENALTY
                  mydebug:writeDebug("Start penalty " .. self.CurrentSpeed ..
"penalty value " .. self.CurrentPenalty)
            end
      else
            self.state=CHECKING
            self.TotalPenalty=self.TotalPenalty+self.CurrentPenalty
            self.CurrentPenalty=0
            if self.Totalpenalty > self.TotalPenaltyLimit then
                  self.Fail=TRUE
                  self:EndGame()
            end
            mydebug:writeDebug("No longer overspeed")
      end
end
```

This function calculates the penalty increase, and end the game if the absolute limit is reached. Otherwise it will set the state to Penalty, leaving it to the Penalty function to punish the driver:

```
function SpeedControl.Penalty(self)
      self.CurrentPenalty=0
      self.TotalPenalty=self.TotalPenalty+self.CurrentPenalty
      if self.Fail==TRUE then
            self:EndGame()
            return
      end
      if self.MaxWarnings <=self.Warnings then
            self:EmergencyStop()
            self.Fail=TRUE
            return
      end
      if self.MaxWarnings >self.Warnings then
            self:Message()
            self.Warnings=self.Warnings+1
            return
      end
```

There are three penalties implemented: a warning message, a penalty stop and game end.

```
function SpeedControl.EndGame(self)
      mydebug:writeDebug("End game")
      if self.CanFail==TRUE then
            self.TotalPenalty=self.TotalPenalty+self.CurrentPenalty
            if self.TotalPenalty > self.TotalPenaltyLimit then
                  self.Fail=TRUE
            end
            if self.Fail== TRUE then
                  mydebug:writeDebug("Fail game")
                  SysCall ( "ScenarioManager:TriggerScenarioFailure", 
self.FailMessage);
            else
                  mydebug:writeDebug("End game, success")
                  SysCall ( "ScenarioManager:TriggerScenarioComplete", 
self.SuccessMessage );
            end
      end
end
```

EndGame will first determine wetter the scenario is completed successfully. The designer has a lot of control about this and can even decide that a scenario never fails due to overspeed.

The message function is straight forward:

```
function SpeedControl.Message(self)
      mydebug:writeDebug("Overspeed message, warning " .. self.Warnings)
            if(self.unit==KMPH) then
                  unittext="km/h"
            else
                  unittext= "Mph"
            end
            if self.Buzzer==TRUE then
                  SysCall ( "ScenarioManager:PlayDialogueSound",self.BuzzerFile);
            end
            SysCall("ScenarioManager:ShowAlertMessageExt", "Overspeed", "Exceeding 
speed limit of " .. round(self.SpeedLimitDisplay) .. unittext .."\nActual speed is 
" .. round(self.CurrentSpeed) .. unittext, 10, "")
      self.state=OVERSPEED
end
```

The penaltybreak disables the control and creates a full stop. Actually, this function maintains two states: initiate a full stop and watch till the train actually stopped before releasing the controls again.

```
function SpeedControl.EmergencyStop(self)

      if self.state ==STOPPING then
            CurrentSpeed = SysCall( "PlayerEngine:GetSpeed")
            CurrentSpeed=CurrentSpeed*self.conversion
            if (CurrentSpeed <1) then
                  SysCall ( "ScenarioManager:UnlockControls")
                  self.state= CHECKING
                  if self.CanFail==TRUE then
                        self.Fail=TRUE
                  end
                  return
            end
      else
            mydebug:writeDebug("Emergency stop")
            SysCall("ScenarioManager:ShowAlertMessageExt", "Penalty 
stop","Excessive overspeed.",10,"")
            SysCall ( "PlayerEngine:SetControlValue", "Regulator", 0, 0.0);
            SysCall ( "PlayerEngine:SetControlValue", "Reverser", 0, 0.0);
            SysCall ( "PlayerEngine:SetControlValue", "TrainBrakeControl", 0, 1.0);
            SysCall ( "ScenarioManager:LockControls")
            self.state=STOPPING
      end
end
```

Things to do:

1. Save game state and resume from a saved game.
2. Localizations and better customization of the messages.
3. Maybe add new punishments

11.6 Thunder

Coming someday …

11.7 Emergency brake

This works. You probably will add code to disable controls for the user till the train actually stops.

```
         SysCall ( "PlayerEngine:SetControlValue", "Regulator", 0, 0.0);
         SysCall ( "PlayerEngine:SetControlValue", "Reverser", 0, 0.0);
         SysCall ( "PlayerEngine:SetControlValue", "TrainBrakeControl", 0, 1.0);
```

11.8 Speed limit monitoring

The script in this chapter can be very useful for routes where speed restrictions do not appear in the HUD. An example is the Albula line. It has speed signs, but these are very small and not always visible.

This scenario warns you both inside the cab and outside the cab in advance, so you can adjust your speed.

The **scenarioscript.lua** is simple:

1. You need to create at least one event, that starts the monitoring. In this event you construct a speed monitoring object. (See the appendix op object classes for more background). You must specify four parameters:
   a. The desired warning distance
   b. The units km/h (KMPH) or MpH (MPH)
   c. The direction to look for (FORWARD or BACKWARD)
   d. A boolean (TRUE or FALSE) to indicate if you need an audible warning
2. If you want an audible warning, create a subfolder for each supported language and include a .wav sound file. It is now hard coded buzzer.wav.
3. You need to create a condition check, that calls the SpeedMonitorg.Check() function (see chapter 6.7 for the basics).
4. If you want to stop checking, create another event, calling SpeedMonitor:Finish()

5. The OnResume() function makes sure monitoring is restarted if you continue a saved game. It does not remember a complete game state, so you will get a warning for the next change right away.

```
--[[
 Scenarioscript for RJH Long run scenario Albula line
 (C) 2014 Rudolf Heijink
 Version 0.1 alfa
 ]]--

 RUE=1

CONDITION_NOT_YET_MET = 0
CONDITION_SUCCEEDED = 1
CONDITION_FAILED = 2

scriptpath=".\\assets\\RudolfJan\\lua\\"

dofile(scriptpath .. "common.lua")
dofile(scriptpath .. "debug.lua")
mydebug=DebugTS.new()
dofile(scriptpath .. "SpeedMonitor.lua")


function OnEvent(event)
      if event=="start" then
            speedmonitor=SpeedMonitor.new(250, KMPH, FORWARD,TRUE)
            speedmonitor:Begin()
            return TRUE
      end
      return FALSE
end

function OnResume()
      speedmonitor=SpeedMonitor.ew(250, KMPH, FORWARD,TRUE)
      speedmonitor:Begin()
      mydebug.writeDebug("Resumed speed monitor")
end

function TestCondition ( condition )
      if condition == speedmonitor.ConditionName then
            speedmonitor:Check()
            return CONDITION_NOT_YET_MET
      end
      return CONDITION_NOT_YET_MET
end
```

The actual script is in a class file **SpeedMonitor.lua**

There is a constructor to set up variables, which is straight forward. The interesting point is that you must translate the m/s speed values to km/h or Mph values.

The **Begin()** function starts the actual monitoring

The **Check()** function performs a check and displays the message. The **warned** state variable prevents that messages are displayed continuously.

```
--[[
SpeedMonitor.lua
(C) 2014 Rudolf Heijink
Monitors next change in speedlimit and informs you with a display message
Requires:
      common.lua
      debug.lua

Sound recorded by Mike Koenig http://soundbible.com/1206-Door-Buzzer.html
]]--


RJHSpeedMonitorversion="0.1 alfa New"
mydebug:writeDebug("SpeedMonitor script version " .. RJHSpeedMonitorversion .. "
loaded")

-- Some constants

EndOfLine = 0
LimitNoSigh = 1
LimitSign = 2
NoChange= -1
NotInitialised= -2


KMPH=0
MPH=1

FORWARD = 0
BACKWARD = 1

SpeedMonitor = {} -- the table representing the class, which will double as the
metatable for the instances
SpeedMonitor.__index = SpeedMonitor -- failed table lookups on the instances should
fallback to the class table, to get methods

function SpeedMonitor.new(distance, unit, direction, buzzer)
  local self = setmetatable({}, SpeedMonitor)
  self.WarningDistance =distance or 500 -- warning distance
  self.Distance=10000 -- distance to next speed limit
  self.CurrentSpeedLimit= 0 -- current speed limit
  self.LimitType= NotInitialised -- type of next speed limit
  self.NextSpeedLimit= 1000 -- value of next speed limit
  self.Buzzer= buzzer or FALSE -- use a buzzer
  self.unit= unit or KMPH --  unit for speed limit
  if (self.unit== KMPH) then
      self.conversion= 3.6
  else
      self.conversion= 2.236932
  end
  self.Direction= direction or FORWARD
  mydebug:writeDebug("SpeedMonitor created")
  self.warned=FALSE
  return self
end

function SpeedMonitor.Begin(self, ConditionName)
      self.ConditionName= ConditionName or "SpeedMonitor"
      SysCall ("ScenarioManager:BeginConditionCheck", self.ConditionName );
      limit=SysCall("PlayerEngine:GetCurrentSpeedLimit")
      self.CurrentSpeedLimit=round(limit*self.conversion) -- change m/s into km/h 
or Mph
      mydebug:writeDebug("SpeedMonitor begin checking")
      return TRUE;
end

function SpeedMonitor.Check(self)
      self.LimitType,self.NextSpeedLimit,self.Distance=SysCall("PlayerEngine:GetNex
tSpeedLimit",self.Direction)
      self.NextSpeedLimit=round(self.NextSpeedLimit* self.conversion)
      self.Distance=round(self.Distance)
      if(self.WarningDistance>  self.Distance and self.warned==FALSE) then
            if(self.unit==KMPH) then
                  unittext="km/h"
            else
                  unittext= "Mph"
            end
            if self.Buzzer==TRUE then
                  SysCall ( "ScenarioManager:PlayDialogueSound","buzzer.wav");
            end
            SysCall("ScenarioManager:ShowAlertMessageExt", "Speed monitor", "Speed 
limit changes to " .. self.NextSpeedLimit .. unittext .."\nDistance " .. 
self.Distance .. "m", 10, "")
            -- mydebug:writeDebug("Next speedlimit found " .. 
self.NextSpeedLimit .. "warningdistaNCE=" .. self.WarningDistance .. "DISTANCE=" .. 
self.Distance .. "warned=".. self.warned)
            self.warned=TRUE
      end
      if(self.warned==TRUE) then
            limit=SysCall("PlayerEngine:GetCurrentSpeedLimit")
            limit=limit*self.conversion -- change m/s into km/h or Mph
            if(limit ~= self.CurrentSpeedLimit) then
                  self.CurrentSpeedLimit=limit
                  self.warned=FALSE
                  mydebug:writeDebug("speedlimit changed " .. 
self.CurrentSpeedLimit)

            end
      end
end

function SpeedMonitor.Finish(self, condition)
      self.ConditionName= ConditionName or "SpeedMonitor"
      SysCall ( "ScenarioManager:EndConditionCheck", condition );
end
```

Things to do:

1. Test OnResume functionality
2. Support for miles for the warning distance
3. Small bug, at starting the script it creates a message twice
4. Make a difference between a higher speed limit and a more restrictive speed limit.
5. Make the warning distance depend of the difference between the current speed limit and the next speed limit
6. Localization support
7. Name of the sound file not hard coded, safety check for the existence of the sound file
8. Add code to interrupt warning messages, instead of stopping the script class.
9. Combine the basic code with additional functionality like warning for signals and AWS handling

11.9 Monitoring closing the train doors

This script is quite straight forward if you studied the other examples. I called it Guard because the idea is to do fancy things around the station stops. The start is easy, I want to check if the train doors are open or closed. When the doors are closed after a station call, the driver is informed and the guard blows the whistle.

The core piece of code is this function:

```
function Guard.Check(self)

      s1=SysCall(self.van .. ":GetControlValue","DoorsOpenCloseLeft",self.index) or
ERROR2
      s2=SysCall(self.van .. ":GetControlValue","DoorsOpenCloseRight",self.index)
or ERROR2
      mydebug:writeDebug("Doors state left=" .. s1 .." Right=".. s2)
      if (s1+s2==0) and (self.DoorsLeft+self.DoorsRight>0) then
            -- if s1+s2 = 0 then the doors are closed now, only perform action if
doors were open previously
            self.DoorsLeft=s1
            self.DoorsRight=s2
            self:ShowWarning()
            return
      end
      self.DoorsLeft=s1
      self.DoorsRight=s2
end
```

The statement

```
s1=SysCall(self.van .. ":GetControlValue","DoorsOpenCloseLeft",self.index) or
ERROR2
```

monitors the door state. 0 means door closed, 1 means door is open. I use here **self.van** as a parameter. For EMU’s and DMU’s you can use PlayerEngine, but for coaches towed by an engine, you need to refer to the vehicle number of the coach. This means that this control will only detect the door state of this particular coach. For the sample I used a coach in the middle of the train. You may change the code to support an array of coach numbers (or even better retrieve this list using the API, I wish I knew how to do this).

The rest of the code is straight forward, using the same techniques you see in other scripts, so I just publish the code here:

```
--[[
Guard.lua
(C) 2014 Rudolf Heijink
Monitors open/close doors and lets the guard blow the whistle
Requires:
      common.lua
      debug.lua


]]--


RJHGuardversion="0.1 alfa New"
mydebug:writeDebug("Guard script version " .. RJHGuardversion .. " loaded")

-- Some constants

DOORS_OPEN=1
DOORS_CLOSED=0
ERROR= -100
ERROR2= -200


Guard = {} -- the table representing the class, which will double as the metatable
for the instances
Guard.__index = Guard -- failed table lookups on the instances should fallback to
the class table, to get methods


function Guard.new(van, whistle, buzzer)
  local self = setmetatable({}, Guard)
  self.Buzzer=buzzer or false
  self.whistle= whistle or ""
  self.van= van or "PlayerEngine"
  self.index=0 -- currently not used

self.DoorsLeft=SysCall(self.van ..":GetControlValue","DoorsOpenCloseLeft",self.inde
x) or ERROR

self.DoorsRight=SysCall(self.van ..":GetControlValue","DoorsOpenCloseRight",self.in
dex) or ERROR

  mydebug:writeDebug("Guard created")
      mydebug:writeDebug("Initial Doors state left=" .. self.DoorsLeft .."
Right=".. self.DoorsRight)
  return self
end

function Guard.Begin(self, ConditionName)
      self.ConditionName= ConditionName or "Guard"
      SysCall ("ScenarioManager:BeginConditionCheck", self.ConditionName );
      mydebug:writeDebug("Guard begin checking van=" .. self.van)
      return TRUE;
end

function Guard.ShowWarning(self)

      if self.Buzzer then
            SysCall ( "ScenarioManager:PlayDialogueSound",self.whistle);
      end
      SysCall("ScenarioManager:ShowAlertMessageExt", "Guard:", "Doors are
closed",10)
end

function Guard.Check(self)

      s1=SysCall(self.van .. ":GetControlValue","DoorsOpenCloseLeft",self.index) or
ERROR2
      s2=SysCall(self.van .. ":GetControlValue","DoorsOpenCloseRight",self.index)
or ERROR2
      mydebug:writeDebug("Doors state left=" .. s1 .." Right=".. s2)
      if (s1+s2==0) and (self.DoorsLeft+self.DoorsRight>0) then
            -- if s1+s2 = 0 then the doors are closed now, only perform action if
doors were open previously
            self.DoorsLeft=s1
            self.DoorsRight=s2
            self:ShowWarning()
            return
      end
      self.DoorsLeft=s1
      self.DoorsRight=s2

end

function Guard.Finish(self, condition)
      self.ConditionName= ConditionName or "Guard"
      SysCall ( "ScenarioManager:EndConditionCheck", condition );
            mydebug:writeDebug("Guard stop checking")
end
```

And a usage example ScenarioScript.lua

```
--[[
 Scenarioscript for testing Guard functions
 (C) 2014 Rudolf Heijink
 Version 0.1 alfa
 ]]--

DEBUG=true

scriptpath=".\\assets\\RudolfJan\\luadev\\"

dofile(scriptpath .. "common.lua")
dofile(scriptpath .. "debug.lua")
mydebug=DebugTS.new()
dofile(scriptpath .. "Guard.lua")
```

```
params={
      "22948520",
      "whistleconductorwhistle.wav",
      true
      }


function OnEvent(event)
      if event=="start" then
            guard=Guard.new(unpack(params))
            guard:Begin()
            return TRUE
      end
      return FALSE
end

function OnResume()
      mydebug=DebugTS.new()
      guard=Guard.new(unpack(params))
      guard:Begin()
      mydebug.writeDebug("Resumed guard")
end

function TestCondition ( condition )
      if condition == guard.ConditionName then
            guard:Check()
            return CONDITION_NOT_YET_MET
      end
      return CONDITION_NOT_YET_MET
end
```

A. Appendix Framework code

B. Appendix verified control functions overview

C. Appendix Unverified control functions

Following paragraphs show the names of default controls used. Please note, these can be customized. You may want to look for files like Inputmapper to discover the control names to use. The list is not complete. EngineStart and EngineStop may be of interest as well.

C.1 Train Controls

SimpleThrottle >> Speed Up / Slow Down
SimpleChangeDirection >> Switch Directions
Reverser >> Increase / Decrease Reverser
Regulator >> Increase / Decrease Throttle
CombinedThrottleBrake >> CombinedThrottleBrake?
GearLever >> Increase / Decrease Gear
TrainBrakeControl >> Increase / Decrease Train Brake
EngineBrakeControl >> Increase / Decrease Locomotive Brake
DynamicBrake >> Increase / Decrease Dynamic Brake
EmergencyBrake >> Emergency Brakes
HandBrake >> Handbrake
Horn >> Horn
Bell >> Bell
Wipers >> Wipers
Sander >> Sander
Headlights >> Headlights
AWS >> Automated Warning System control
AWSReset >> Automated Warning System Reset control
AWSClearCount >> Increase remented for every ramp than sounds a bell
AWSWarnCount >> Increase remented for every ramp than sounds a buzzer
Startup >> Engine Startup / Shutdown control
Wheelslip >> Wheelslip - used for visibility objects
DoorsOpenCloseLeft >> Open / Close Doors on left side
DoorsOpenCloseRight >> Open / Close Doors on right side

C.2 Electric Locomotive Controls

PantographControl >> Raise / Lower Pantograph

0= down

1=up

Seems not working for all engines

FrontPantographControl >> Also set by PantographControl?
RearPantographControl >> Also set by PantographControl?

C.3 Steam Locomotive Related Controls

FireboxDoor >> Open / Close Firebox
ExhaustInjectorSteamOnOff >> On/Off Exhaust Injector Steam
ExhaustInjectorWater >> Increase /Decrease Exhaust Injector Water
LiveInjectorSteamOnOff >> On/Off Live Injector Steam
LiveInjectorWater >> Increase / Decrease Live Injector Water
Damper >> Increase / Decrease Damper
Blower >> Increase / Decrease Blower
Stoking >> Increase / Decrease Coal Shovelling
CylinderCock >> Open / Close Cylinder Cocks
SteamHeating >> Steam Heating
WaterScoopRaiseLower >> Raise / Lower Water Scoop
SmallCompressorOnOff >> On / Off Small Compressor

C.4 Output values for Dials*

General

Speedometer >> Speedometer
Ammeter >> Ammeter
RpmDial >> rpm
Accelerometer >> accelerometer (kN)

C.5 Steam

SteamChestPressureGauge >> Steam Chest Pressure Gauge
BoilerPressureGauge >> Boiler Pressure Gauge
SteamHeatingPressureGauge >> Steam Heating Pressure Gauge

WaterGauge >> the water level in the boiler
SafetyValve >> safety valve state for audio / effects

Brakes

BrakePipePressure** >> Pressure in the brake pipe
VacuumBrakePipePressure** >> Pressure in the vacuum brake pipe
AirBrakePipePressure** >> Pressure in the air vacuum brake pipe
TrainBrakeCylinderPressure** >> Pressure in the train brake cylinder
LocoBrakeCylinderPressure** >> Pressure in the loco brake cylinder
MainReservoirPressure** >> Pressure in the main res
VacuumBrakeChamberPressure** >> Pressure in the vacuum brake cylinder
EqReservoirPressure** >> Pressure in the equalising reservoir
BrakeBailOff* >> Release loco brake when train brake set

***Add **PSI** or **INCHES** to the end of the name depending on the units desired.E.g. BrakePipePressureINCHES*

Output controls added for the sound system

Current >> Current in Amps
TractiveEffort >> Tractive Effort in Unkown Units
RPM >> RPM in rev's per minute Units
RPMDelta >> RPMDelta - lagged and processed RPM DELTA value
CompressorState >> CompressorState? On(1.0)/Off(0.0)

D. Start options for TS 201x

Source:

RWA Bob Artim and Peter Hayes

-allowJump ---- Allows jumping AI trains in scenarios
-blocking
-BlockingPhysics
-ConvertDistanceData
-DisableAWS
-DisableDump
-DisableEAX
-DisableSignals
-DisableSound
-DisplayLocMe

-DontBakeDistances
-DontUseBlueprintCache
-DontUseBlueprintCache -- Possibly rebuilds blueprint .pak's each time???
-EnableAsyncKeys - ctrl-shift-1 through ctrl-shift-5 will at least speed up the clock
-enable-backups
-EnableEAX
-EnableFullEditor
-enable-perforce
-EnableSound
-EnableSoundDebugDialogs
-EndTrackCheck
-FilterForDirectionality
-FilterForManualJcts
-FlashStrings
-followaitrain
-ForceSWMix (that one should disable anything more than basic playback through your soundcard, if you're having problems)
-FPSLimit=xx xx = a value like 30
-fs -- Smaller fonts - narrative text and menus
-generateDetailLevels
-IgnoreCoupling
-ignoreTrackTypes
-Language
-LeipzigDemo
-LogLocStrings
-LogMate
-lua-debug-messages
-ManualCoupling
-NASKU
-NoClearType
-NoPlayerTrain
-NoSplashScreens -- Disable intro splash screens
-NoWagonBraking
-NoWagonBraking
-nvperfhud -- Create NVIDIA performance HUD (not confirmed)
-oogbb_noh ????
-QuickStartSteam
-relaunchmce
-ResetAchievements
-ResetStats
-SaveDistances

-SetDefaultButtonSound
-SetFOV
-SetLogFilters
-Show3DPaths
-ShowAudioConsists
-ShowControlStateDialog -- Expanded HUD
-ShowDestinationMarkerList
-ShowDriverList -- Adds a 'Driver' button inside the Driver Properties window (MK2?)
-ShowSoundDeviceSelector -- Allows choice of system sound device
-ShowTrackLinks
-SkipIntros
-TakeADump
-TrackPatternEntityFixup
-updateRVNumbers
-UseFastBlueprintCache
-UseFastStreamCache
-UseSoundDevice
-UseStreamCache
-ValidateNetwork
-ValidateSignals
-VerboseAudioDebug

E. Useful lua constructs

E.1 Case statement in lua

This version uses the function `switch(table)` to add a method `case(table,caseVariable)` to a table passed to it.

```
function switch(t)
  t.case = function (self,x)
    local f=self[x] or self.default
    if f then
      if type(f)=="function" then
        f(x,self)
      else
        error("case "..tostring(x).." not a function")
      end
    end
  end
  return t
end
```

Usage:

```
a = switch {
  [1] = function (x) print(x,10) end,
  [2] = function (x) print(x,20) end,
  default = function (x) print(x,0) end,
}

a:case(2)  -- ie. call case 2
a:case(9)
```

E.2 Associative arrays

```
-- Support for associative arrays

-- function to iterate over a table and retieve its idex (i) and value (v)
-- Source: Programming Lua v5.2

local function iter (a, i)
      i = i + 1
      local v = a[i]
      if v then
            return i, v
      end
end

-- function that returns the next index, value pair of an associative array
-- Source: Programming Lua v5.2

function ipairs (a)
      return iter, a, 0
end

-- function to fill associative array with index values.
-- returns number of destinations

function createIndex(a)
      for i, v in ipairs(a) do
                  a[v]=i+indexStart-1
                  print(v," ", a[v])
      end
      return i
end
```

This code is used in the Destinationboards example

E.3 Object classes

In lua you simulate object classes. This is done using a metastable. In the example a class DegugTS is created. You can create an instance by assigning the new function to a alua variable, e.g.

```
Debugger= DebugTS.new()
```

The call for instance:

```
Debugger:writeDebug(“ text”)
```

Or

```
Debugger.writeDebug(self“ text”)
```

Note the subtle difference in using a dot (.) or colon (☺. The colon is syntactic sugar, then you don’t need the self parameter.

```
DebugTS = {} -- the table representing the class, which will double as the
metatable for the instances
DebugTS.__index = DebugTS -- failed table lookups on the instances should fallback
to the class table, to get methods


function DebugTS.new(logfile,mode)
  local self = setmetatable({}, DebugTS)
  if logfile== nil then
      self.logfile="logfile.txt" -- file name
  else
      self.logfile=logfile
  end
  if mode==nil then
    self.mode="a+" -- open mode
   else
     self.mode=mode
  end
-- You can find this file in the TS home directory
  self.debugfile= io.open(self.logfile,self.mode) --file handler for debug file
  self.writeDebug(self,"Debug script version " .. RJHDebugversion .. " loaded")

  return self
end

-- write debug message
-- message is a string containing a text message

      function DebugTS:writeDebug(message)
            local dt= os.date("%d-%m-%Y/%X ")
            if(dt==nil) then
                  dt=""
            end
            self.debugfile:write(dt .. message ..'\n')
            self.debugfile:flush()
      end
```