Scenario authors guide

Part III

Create LUA scripts for Trainsimulator 2015 scenarios

Rudolf Heijink

Version 1.0, June 2015

Preface

I proudly present the first final version of my scenario scripting guide. II spent a lot of time collecting all information needed with help from the community. It will be part of a Scenario Authors Guide. This guide will consist of three parts:

Part I: Starters guide. Learn to create straight forward scenarios for your own use. I expect this part to be available in September 2015.

Part II: Advanced techniques for scenario creation. This covers a mix of topics, including publishing scenarios, weather templates, quick drive scenarios etcetera. A first version will be available in December 2015.

Part III: scenario scripting. Create your own LUA scripts.

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

Contact me at trainsimulator@hollandhiking.nl if you have additional information, corrections or comments. This email address shall not be used for asking help. For that purpose, please use the community forums.

Engine driver

This guide contains some fairly advanced stuff. For some readers it may be hard to start using the guide. I agreed with DTG to write some articles for the Trainsimulator community site Engine Driver (http://www.engine-driver.com/). This may help you to get a more easy start in creating scenario scripts. I do not know when these articles will be published. Possibly I will publish these tutorials elsewhere.

Images

Each chapter is introduced with a screenshot. They represent my favourite routes, rolling stock and developers. It's a mix between payware, donationware and freeware, often combined in a single picture and is meant as a thank you to the TS2015 development community.

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

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

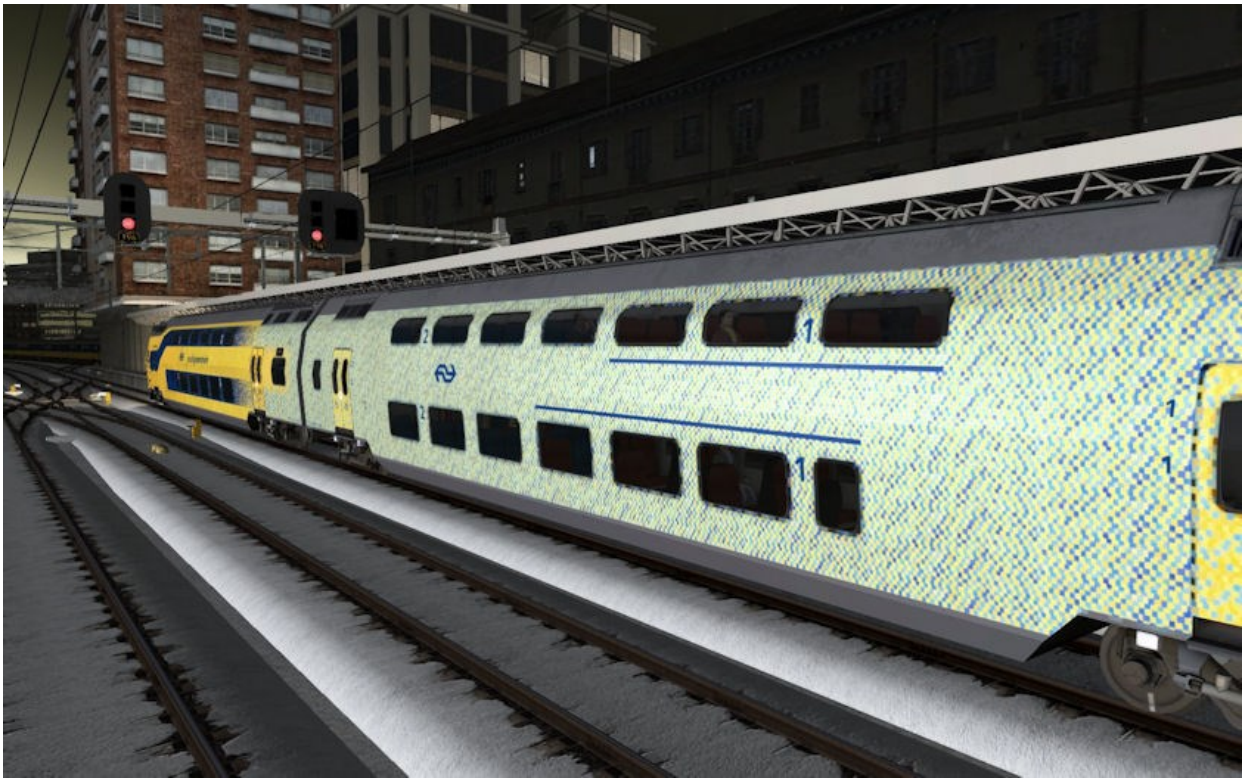
All members of the TS2015 community that somehow shared information that helps to create this guide, whether they are aware of that or not.

Disclaimer

Rudolf Heijink

Contents

List of figures



1 Getting started

1.1 Introduction

Trainsimulator provides wonderful opportunities to create challenging scenarios. In course of time many new features were added. TS2014 added to opportunity to use LUA scripts to create scenarios. Unfortunately documentation was missing. Using examples, information at community sites and lots of trial and error, helped me to create this guide. The good news is that DTG is helping me now to obtain the missing pieces and recently a reference guide with respect to scripting has been added to the developer documents.

LUA is a programming language. In scripts you can use it even if you are not proficient in software development. If you do not have programming experience, have look at the tutorials that will be published at Engine Driver.

You can do many things with scripting, but sometimes it's just easier to use other functions in the TS2015 scenario editor and other content creation tools. In this first edition, focus will be on scripting. In later editions, some other techniques will be explained as well.

1.2 Knowledge prerequisites

1.2.1 LUA basic knowledge

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. You can use the guides that are available in the TS2015 Manuals folder or use one of the many community guides.

Some general knowledge on programming is 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, procedure
- Syntax
- Object
- Array

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 few tutorial sites:

This one looks real good:

http://nova-fusion.com/2012/08/27/lua-for-programmers-part-1/

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/

An unofficial FAQ:

http://www.luafaq.org/

1.2.2 HTML

You may benefit from HTML knowledge. HTML is mark-up programming language for internet pages. One of the many tutorials is this one:

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.

1.2.3 TS2015 scripting

DTG just released some developer reference manuals. You can find the in the **`Railworks/Dev/Docs`** folder if you have version 50.5a or newer. I recommend reading them all, especially the two guides on engine scripting and scenario scripting. You should be aware that the current versions are not complete.

The manual on scripting suggests that you can use commands using a syntax like

```
ShowMessage("Good", "You just left the station")
```

I tested this, and discovered it does not work (yet). Maybe it will come in near future.

This guide contains some fairly advanced stuff. For some readers it may be hard to start using the guide. I agreed with DTG to write some articles for the Trainsimulator community site Engine Driver (http://www.engine-driver.com/). This may help you to get a more easy start in creating scenario scripts.

A scripting reference guide, but targeted to engine scripts and a bit signalling.

https://docs.google.com/document/d/19gnyrDe553WTy6AJqqLCkGyMfFwIFDnR9oeVGCflcdE/edit?pli=1#heading=h.gzu0moiqd5sc

The Railworks wiki contains a lot of information for content creation. There is no direct information about scenario scripting, but it may be useful as a last resort:

http://railworkswiki.com/tiki-index.php

This blog may contain additional information or clarify topics covered in this guide:

http://trainsimlive.blogspot.co.uk/

1.3 Lua version

TS2015 uses lua version 5.02 You should be aware of this, because it is not the latest version of lua and incompatibilities may occur if you installed a newer lua version on your computer

The statements

```
DEBUG=true
Print(_VERSION)
```

will print the version of the LUA interpreter to **LogMate** (see also chapter 0).

1.4 Tools

You can use Notepad to edit LUA scripts, but it is much easier to use a programming editor like **SciTE**.

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, but SciTE supports HTML as well.

It may be useful to have hex editor available, e.g.

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

If you want to edit or just look at the code in the compressed files, used in TS2015, RWTextEdit is an editor that can open compressed files, edit them and store them again in compressed form.

http://www.ivimey.org/content/rwtextedit

1.5 Debugging

Debugging your script may be cumbersome. In chapter 2.6 you find information on debugging scenario scripts.

2 Organization

2.1 Directory structure

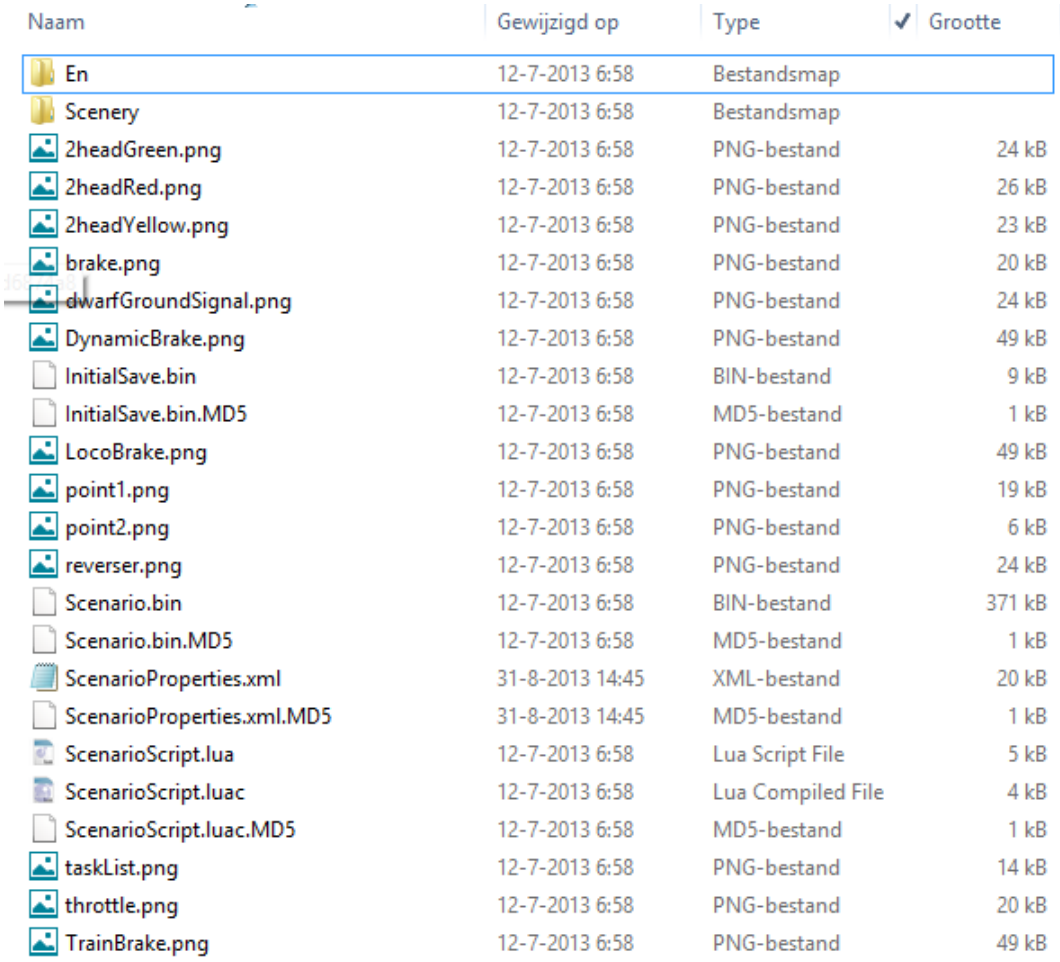
Your LUA scenario script must be called

```
ScenarioScript.lua
```

If you use media (e.g. sound files, html files or video files), create a directory in the scenario directory for the languages you support, e.g. En for English. See the example in Figure 1.

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.

LUA is case sensitive, as are folder names. Chris Longhurst reports that he needed to use **`en`** as folder name (so all lower case) for sound files. The localisation in TS2015 is not working properly. I recommend to use always English language localisation next to other languages. If you use only German, texts will not be shown if the user has English set as language and you do not provide English localisation.



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

Figure 1. Example scenario folder

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.lua** in this example:

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

dofile is a lua library function, which is very useful.

NB If you intend to publish the script as a workshop scenario, you probably should put all scripting in the **ScenarioScript.lua** file. I think the workshop upload function currently does not look for additional script files. This is logical because the workshop system intends to keep things simple and does not know that you use **dofile** constructs.

2.2 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 (Figure 2), called **luadev** and **lua**. I use **luadev** 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**.

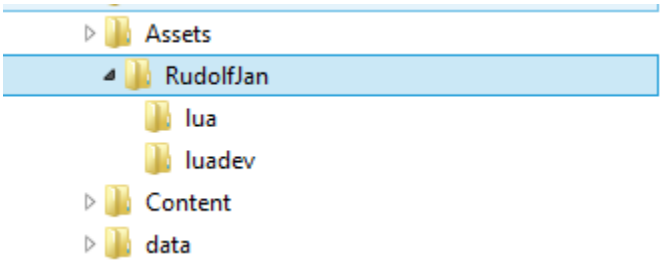


Figure 2. Example assets folder structure for scripting.

Most code goes in a script file that resides in the folder structure shown in Figure 3:

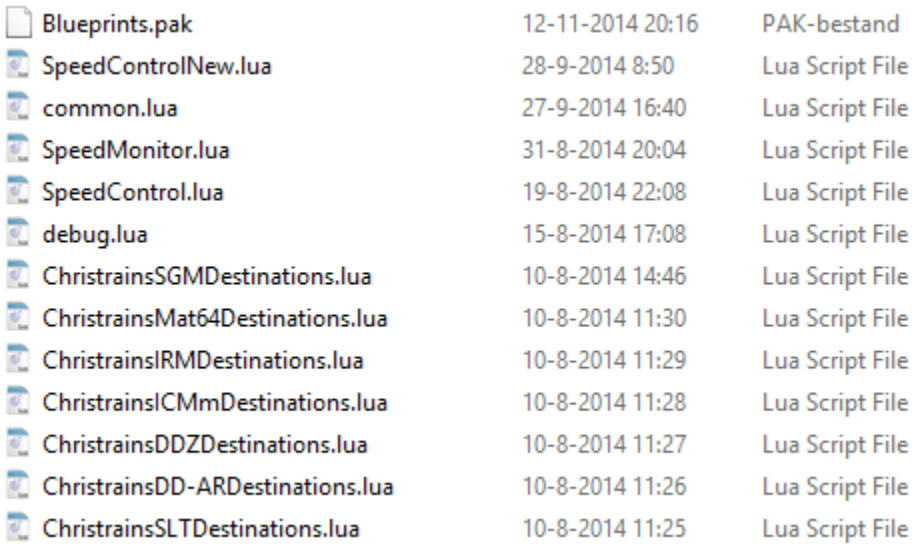


Figure 3. Sample scripting files in the asset folder

In the scenario folder, I create a script called **`ScenarioScript.lua`** as required by TS2015. I try to keep the amount of code in this script to a minimum. Have look at the sample code in chapter 0.

Note:

You probably cannot use this if you intend to publish scripted scenarios in Steam Workshop, because Steam Workshop cannot know you use this asset folder.

2.3 Accessing your script from inside TS2015

You can access the LUA script from the scenario edit function in TS2015. You need to open the Timetable view (2D editor view) see Figure 4,

Figure 4. Open timetable view in the scenario editor

and you will see two buttons supporting scripting functions (see Figure 5):

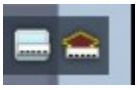


Figure 5. The left button opens the lua script dialog.

The left button opens a view on the LUA script. The right button opens a string definition screen. If you click on the LUA button, a pop-up screen appears as shown in Figure 6.

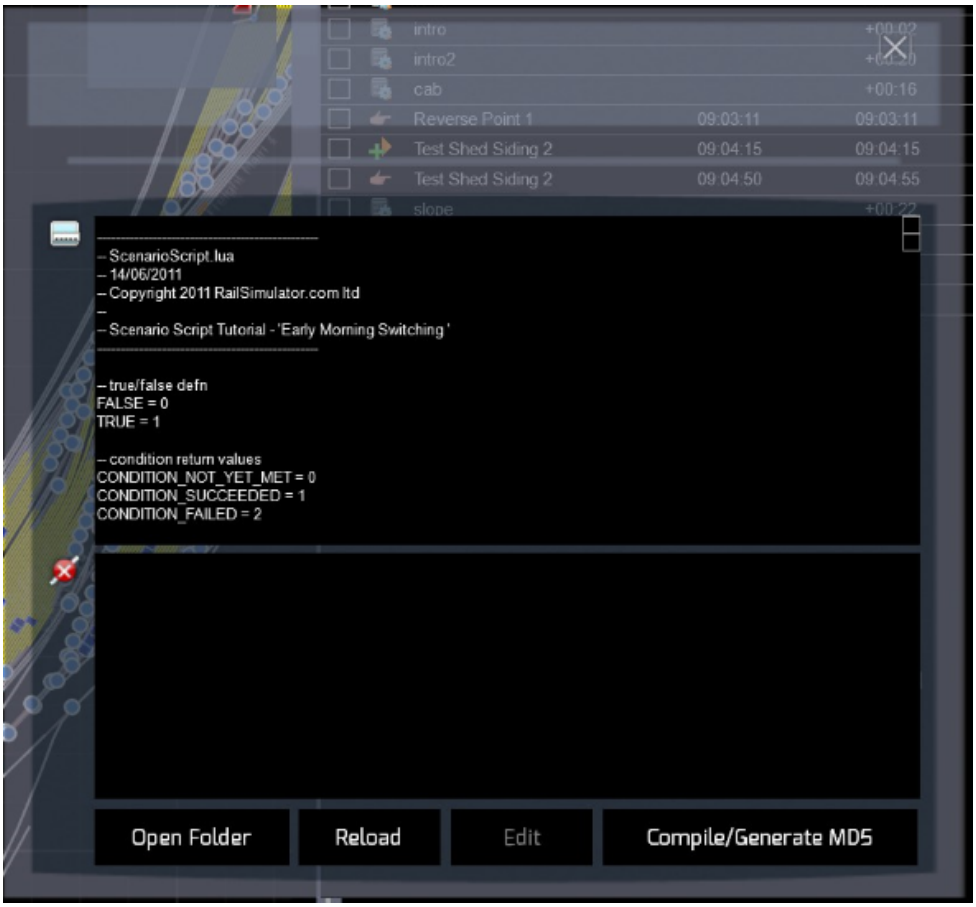


Figure 6. Lua script dialog screen

You can use this screen to have a look at the LUA script. It does not show embedded files. Unfortunately most scripts provided by DTG 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.

You need to press **Reload** after editing a script outside TS2015. Reload is useful anyway, because it will report syntax errors in your script. Before you can use the modified script, you need to compile it, using the **Compile/Generate MD5** button.

The Strings pop-up looks like Figure 7.

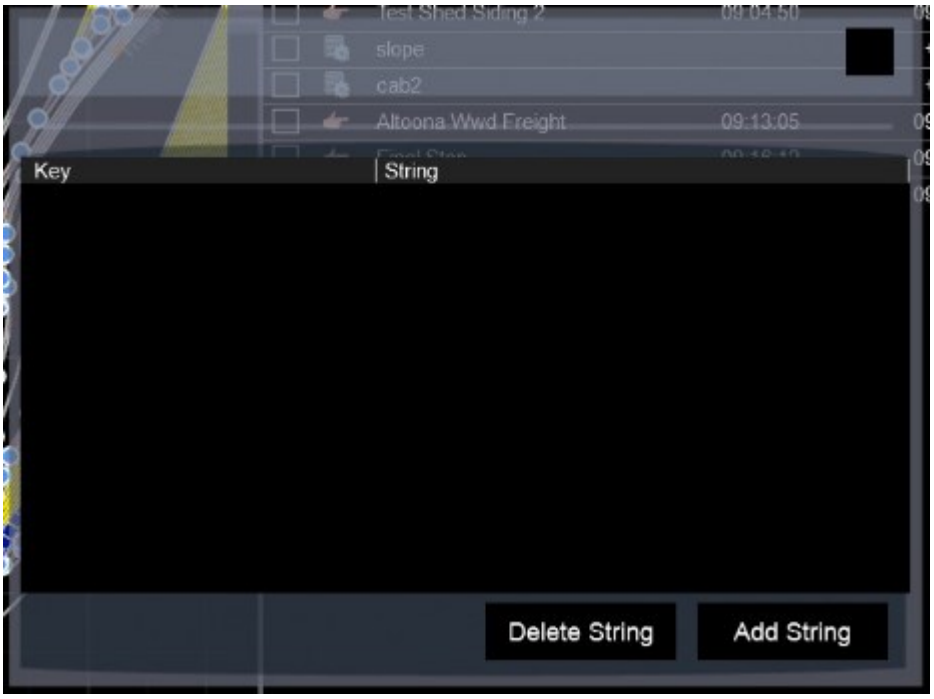


Figure 7. Create GUIDs dialog screen

Adding a string creates u new GUID (Globally Unique IDentifier), to which you can add a string description. This can be handy if you need a new GUID.

2.4 Basic API call

The generic syntax for calling game functions from scenarios looks like:

```
SysCall(<command>, <param1>,<param2> ,…)
```

<command> is the function to be used. In this guide most functions will be explained in due course. The "Hello World" example gives you an idea (see chapter 2.8).

In scenario scripts you need to use the **SysCall** function, not the **Call** function that is used in engine scripts.

2.5 Performance

Each time you use SysCall causes a 0.04 ms delay. If you use condition checks 5.6, be aware that the functions are called every frame. Suppose you want a frame rate of 30fps, this means that for each refresh 33ms is available. So as a rule of thumb, try to use less than 10 API calls in all condition checks together. My general onion is that LUA scripts perform quite will, so no need to worry very much about performance.

It helps to use a specific API call only once in a condition check and store results in local variables.

2.6 Debugging

The start of the debugging process is the **Reload** function in the TS2015 scenario editor. 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 may have a negative effect on game performance. for testing scripts it helps to use a simple route (e.g. TestTrak, Seebergbahn or create your own testing route), otherwise you spend lots of time waiting for the game to load.

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 other way to find these errors.

For errors in the programming logic, I created a simple logging system, which turns out to be much faster than LogMate.

Also useful is the capability to watch the engine control parameters during gameplay. This is described in chapter 9.2.

2.6.1 Debugging syntax errors in the Scenario Editor

If any syntax error occurs and you press “Reload“ an error message will be displayed. It shows both the file name and the line number and is very fast. It will not show more advanced compilation errors.

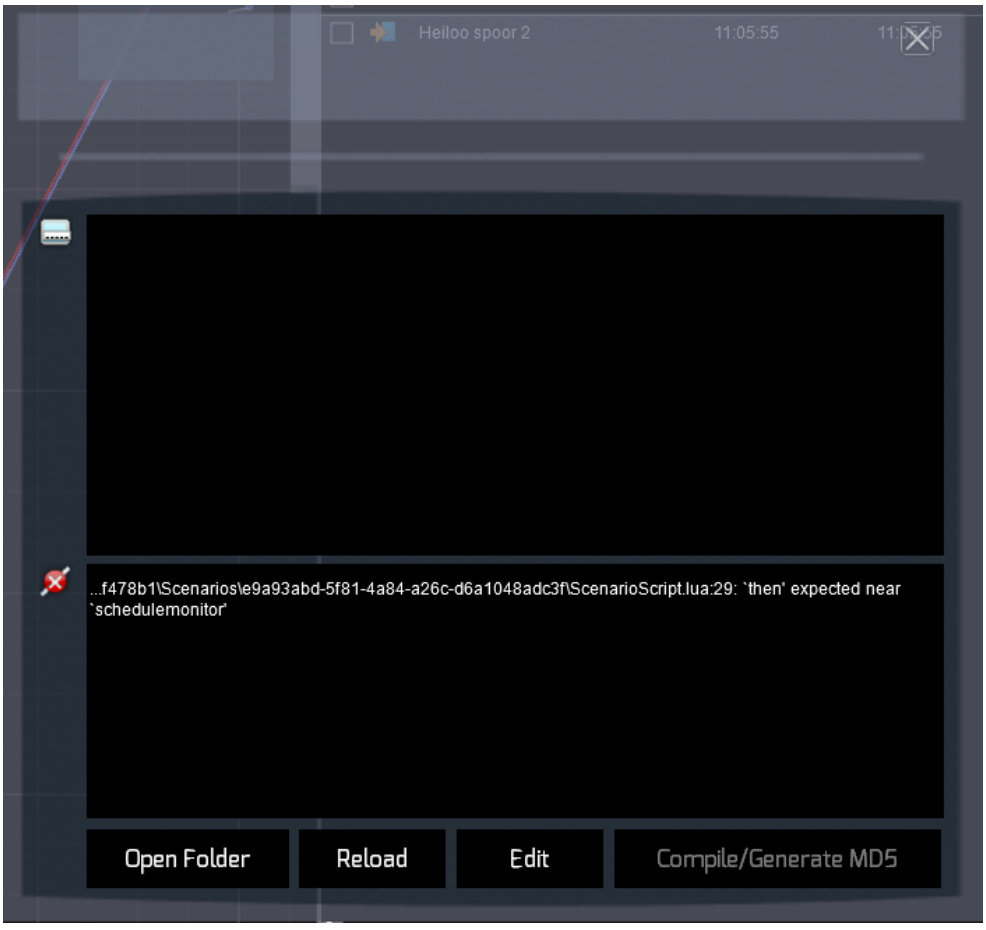


Figure 8. Example showing syntax error in the scenario script dialog.

Figure 9. Select Trainsimulator 2015 in the games list to access game options.

2.6.2 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**!!! **true** is the LUA Boolean value, **TRUE** is used as a semi-boolean but has value **1** in TS scripts.

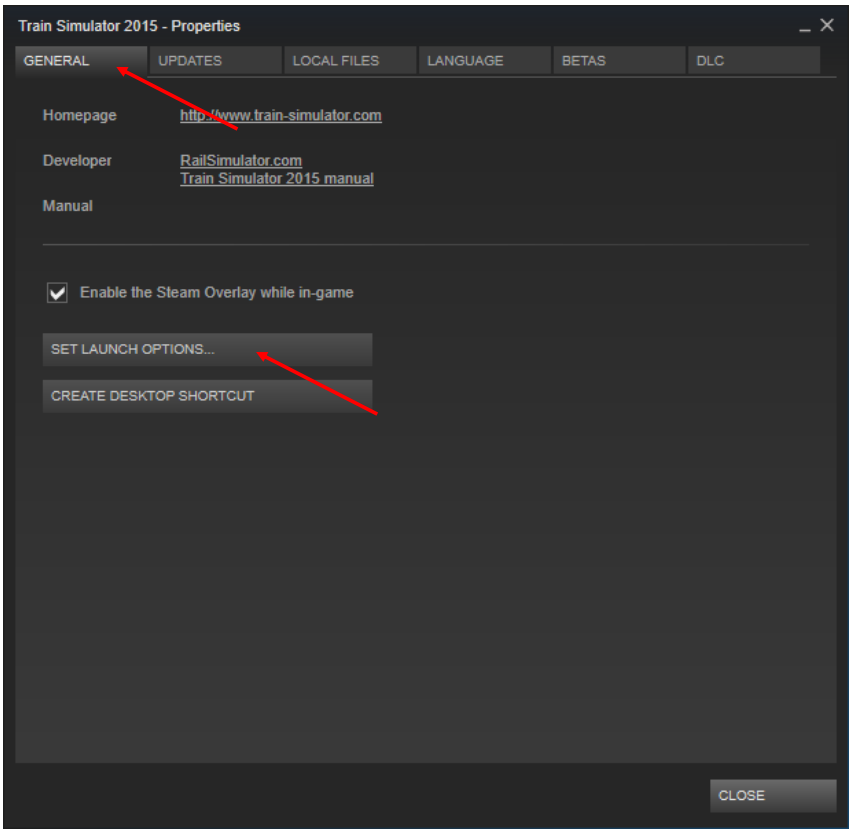


Figure 10. Select launch options here

To activate LogMate, you must use following steps:

1. Click in the games list with the right mouse button on Train Simulator 2015 (Figure 10).
2. A popup menu appears. Select the “Properties” entry. This opens a form (Figure 9).
3. Op the tab “General” click the button “SET LAUNCH OPTIONS … “ A new form appears (Figure 11).
4. Type the following text in the edit field:

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

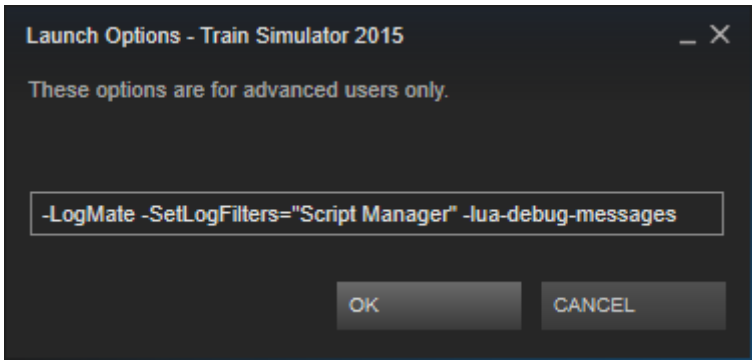


Figure 11. Set launch options for debugging in LogMate

The option **`-lua-debug-messages`** instructs LogMate to show the results of **`Print`** statements.

LogMate may have a significant negative effect on performance. You need LogMate only to find syntax errors in LUA scripts. Once you removed all typing errors, you can switch off LogMate, just by closing the LogMate Program. The performance problems disappear right away. The next time you start Train Simulator, LogMate will start again

Note:

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

2.6.3 Speed up scenario play

When you add the option **`- EnableAsyncKeys`** to the launch 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).

It is recommended to play a scenario always at least once at normal speed, to make sure timing is correct. Also, some scripts may not work properly when you try to speed up the scenario.

2.6.4 Print to LogMate

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 must set the launch option **`-lua-debug-messages`** to make this work. Note that this Print function is not the same as the print function in the LUA standard library (upper case P not the lower case p).

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.

2.6.5 A 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.

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 TS2015 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()
```

The logifile will be locked by TS, so you cannot delete it while TS is running, unless you close it explicitly in the script.

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 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:ShowAlertMessageExt", "Debug",
                   message, 10, TRUE)

     End
End
```

This command will show a message in the upper right corner of the screen during 10 seconds and pause the game.

2.7 Run Trainsimulator in windows mode

During development of scenario scripts, it may be convenient to run TS2015 in windowed mode.

1. You can watch LogMate while playing.
2. You can easy use a program to create screenshots of part of the game screen.
3. You need to run in windows mode to watch the engine parameterslist (see also chapter 9.2).

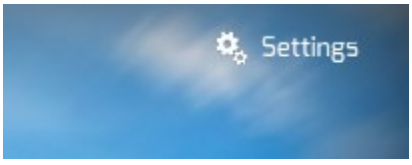


Figure 12. Locate the settings menu.

To set TS2015 to windowsmode, use following steps:

1. In the TS2015 “Main” Menu select the Settings button (upper right on the screen, Figure 13)
2. Press the “Graphics” button (Figure 13).
3. Set “Full screen“ settings to Windowed and reduce the screen resolution (Figure 14).
4. Restart TS2015

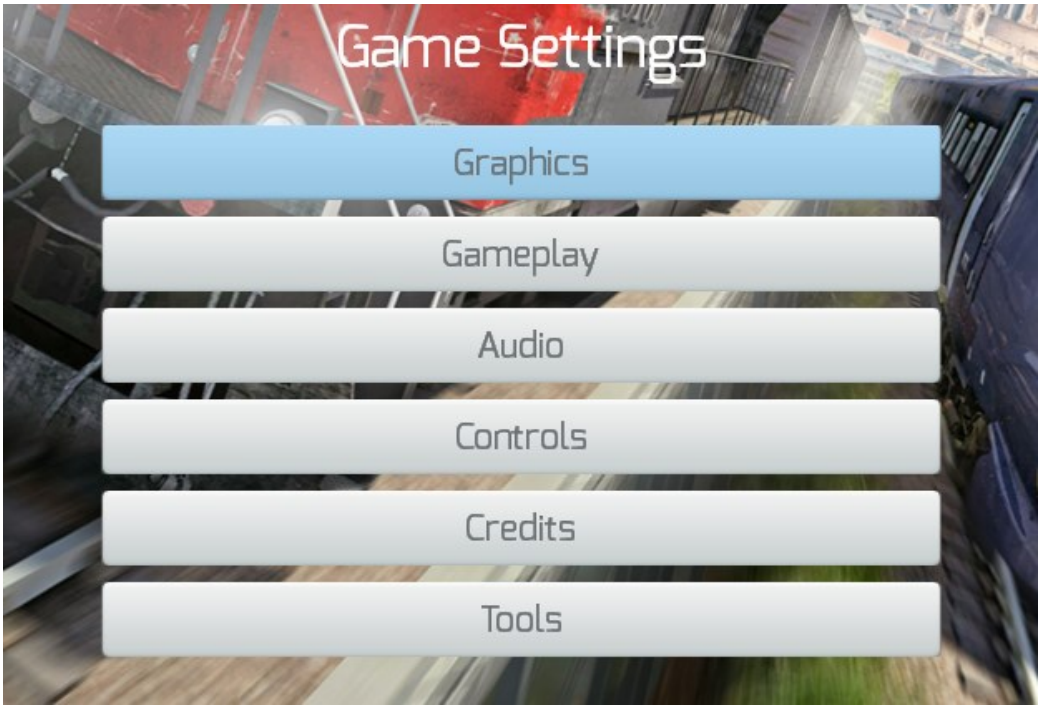


Figure 13. Select Graphics menu to set TS2015 to windowed mode

Figure 14. Set full screen option and adjust resolution.

2.8 Hello world

Courses for many programming languages start with the simple “Hello World” example. This example does nothing more than display the text “Hello World”. To give you a start here a “Hello world” scenario script is presented. The working example is published in Steam Workshop. It uses the TestTrak route and DTG Academy rolling stock. So you can actually play with the example and view the code if you like.

The first step is to create a scenario. I used TestTrak as route for this purpose, but that is not important. You need to place a consist as shown in Figure 15.

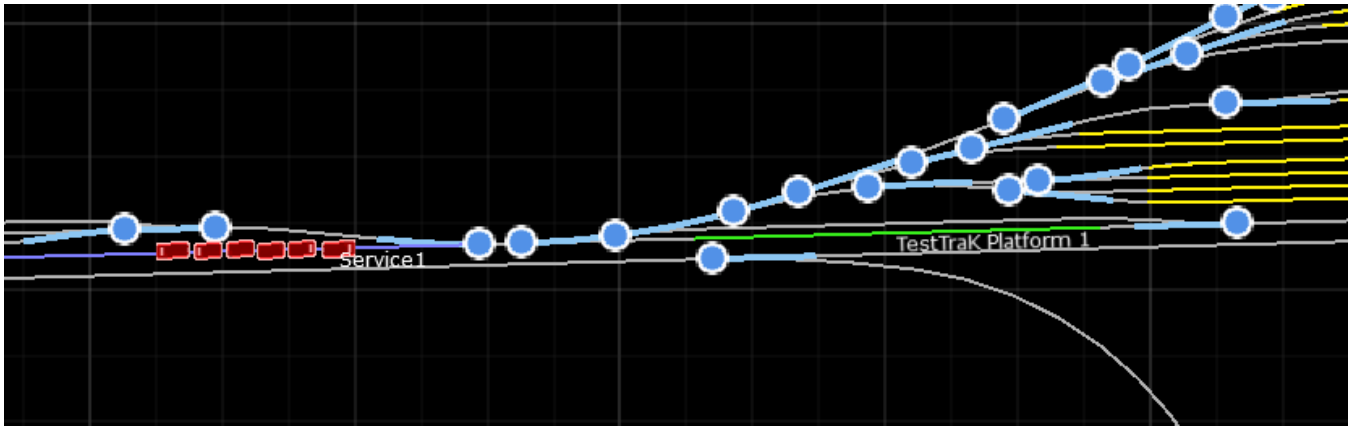


Figure 15. Step 1. Place a consist in the sample scenario

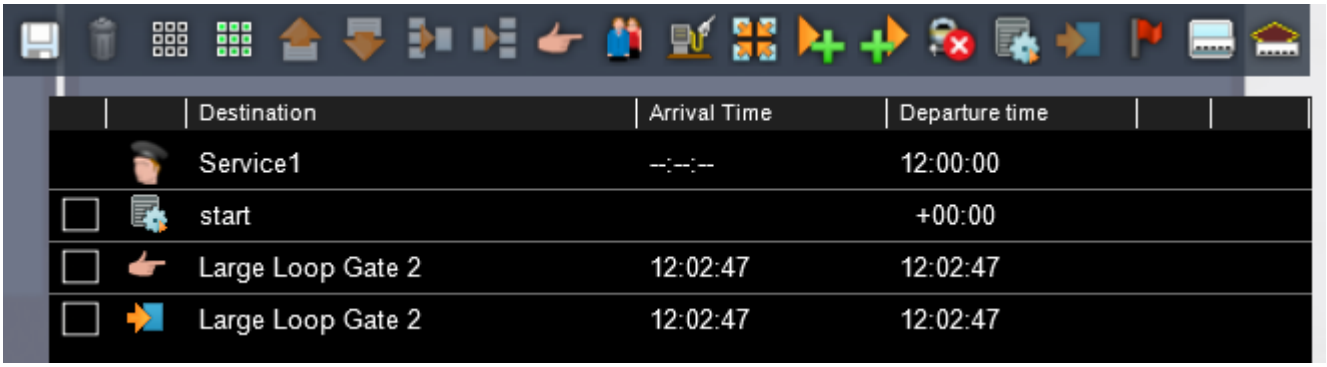


Figure 16. Step 2. Add driver instructions and final destination

Then create some driver instructions and the final destination.

The important step is that you must add an event trigger called “start” to the trigger instruction, as shown in Figure 17.

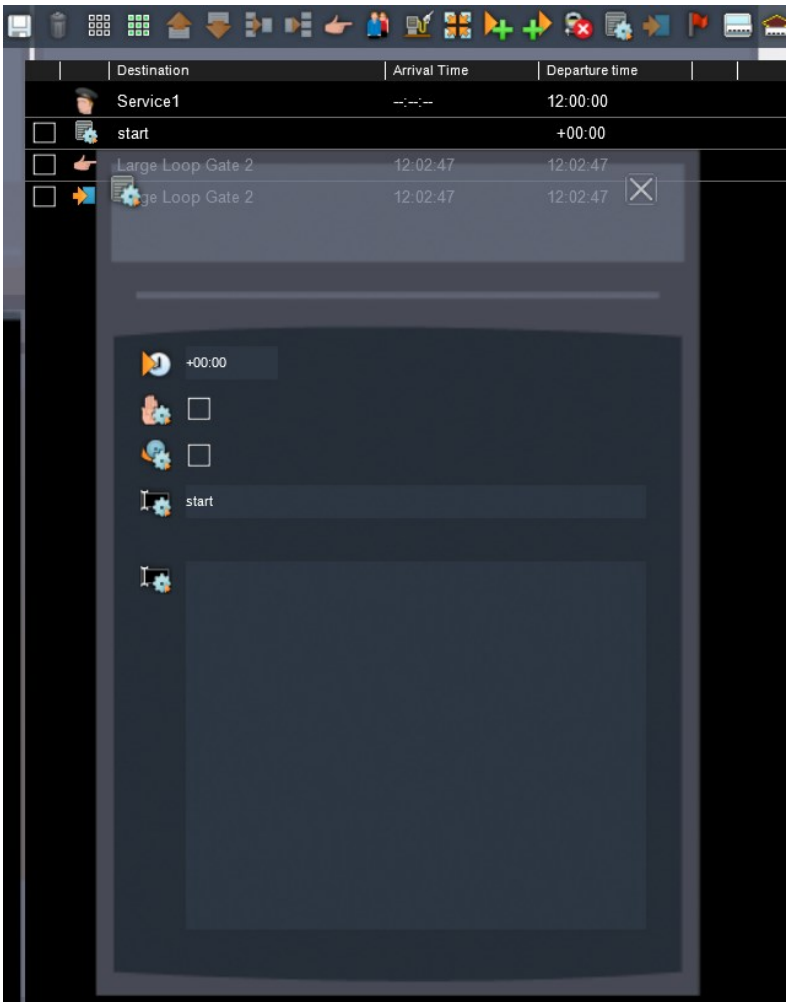


Figure 17. Step 3. Add an event trigger to the trigger instruction.

Now you can write your first script. The easy way to do this, is to open the script dialog and then select the scenario folder (see also chapter 2.3) if you forgot how to do this). Use an editor to type the text below (I recommend typing it yourself. You will learn a lot from that experience).

```
--[[
Hello world
Sample scenario script for Trainsimulator 2015
]]--

TRUE=1
FALSE=0

DEBUG=true

Print("\nLUA version is ".. _VERSION .. " Hello world")

function OnEvent(event)
     if event=="start" then
```

```
            SysCall("ScenarioManager:ShowMessage","Hello world")
            return  TRUE
      end
      return FALSE
end
```

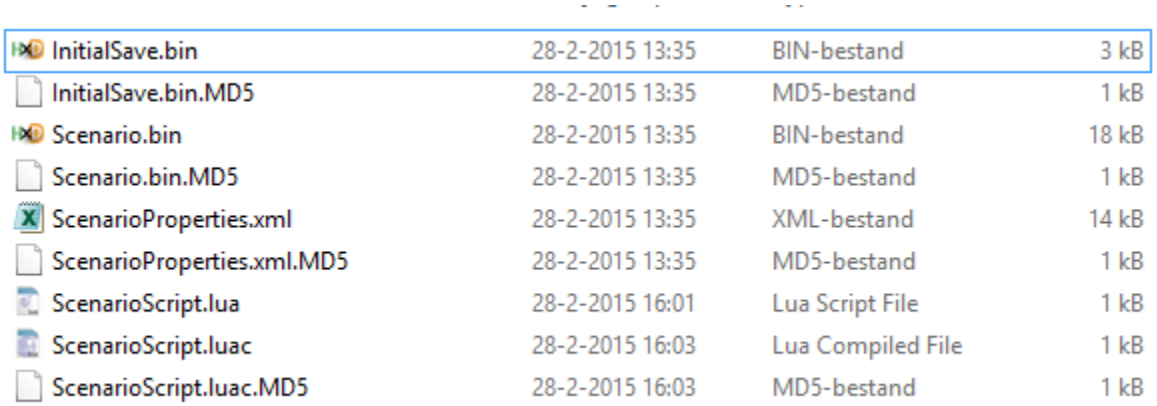


Figure 18. What your scenario folder must look like.

After saving the script, return to the scenario editor. Click on the “Reload“ button in the script dialog. The click on “Compile/Generate MD5“ and run the scenario.

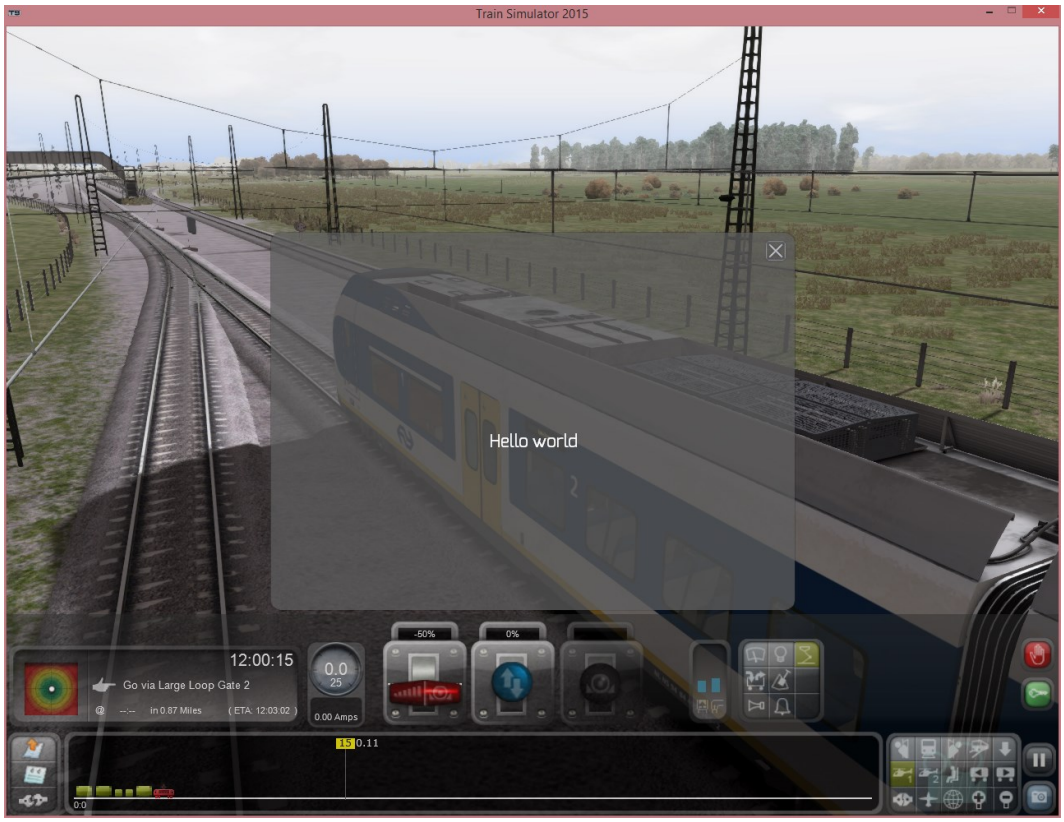


Figure 19. The "Hello world" script in action.

In the sample I added a debug message. If you did set up LogMate properly, the LogMate screen will show the print message (Figure 20). If both steps do not show any errors, start your scenario. It will show the screen in Figure 19.

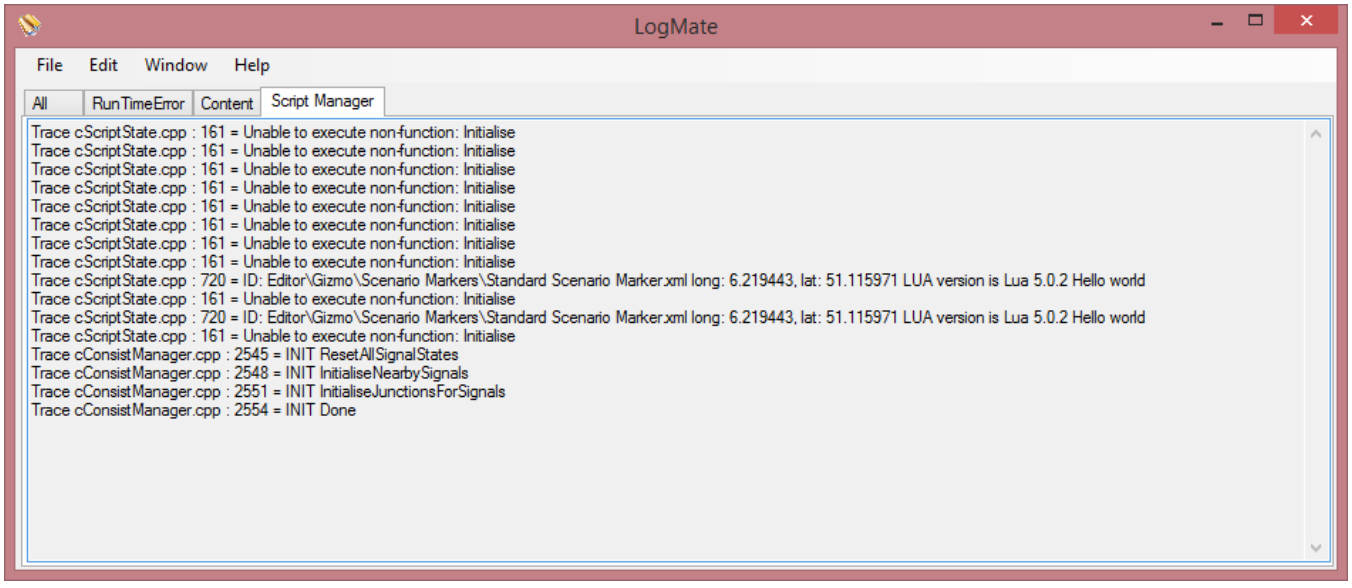


Figure 20. Hello world in LogMate.

3 Events

3.1 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 after a scheduled stop TS2015 will trigger an event, if you create one. Triggering an event, just means that TS2015 will call the **`OnEvent(event)`** function, you created. The **`event`** parameter passes to the name of the event that must be handled.

Your script code may look like this::

```
-- true/false
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 –event not handled
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.

Note

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

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

It is not clear to me which effect the return value has. It seems not yet fully implemented, but it is wise to follow the basic rule to return **`TRUE`** if you can handle the event and **`FALSE`** if you don't handle the event.

3.2 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.

3.3 Adding lua code to the trigger event

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

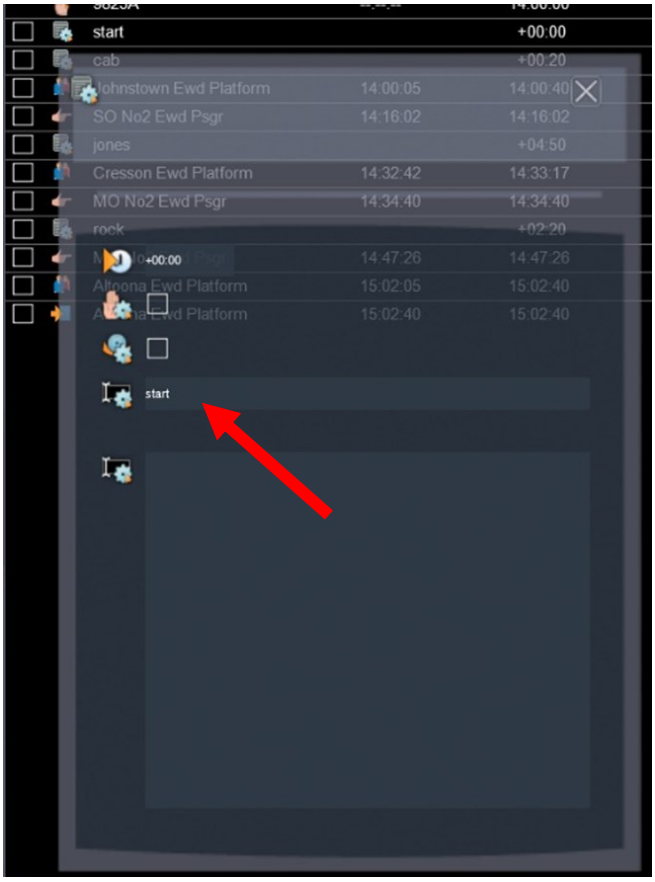


Figure 21. Trigger event dialog in timetable view

If you want to do so, create a trigger event using the "Timetable view" scenario editor (see also chapter 2.3) .

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. If you use the same event name in more than one trigger, each time the same event will be executed.

This is the trick that makes TS2015 call the **`OnEvent()`** function discussed in chapter 3.1

Note in the trigger instruction some events are predefined:

a) A wheelslip event
b) An emergency stop event
c) An event that displays a simple message box without title

The first two events can be activated by placing a checkmark, the third one is activated if you enter a text. The text will NOT be shown if you add a trigger name to the event showing its own text.

3.4 Adding lua code to the other basic events

Following **basic events** support adding lua event handling in the same way, which is slightly different from the rules in the trigger event :

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

For these events TS2015 always executes the basic event first. You can add events the for two different results of the basic event:

a) The event is triggered in case the basic event failed
b) The event is triggered in case the basic event succeeded

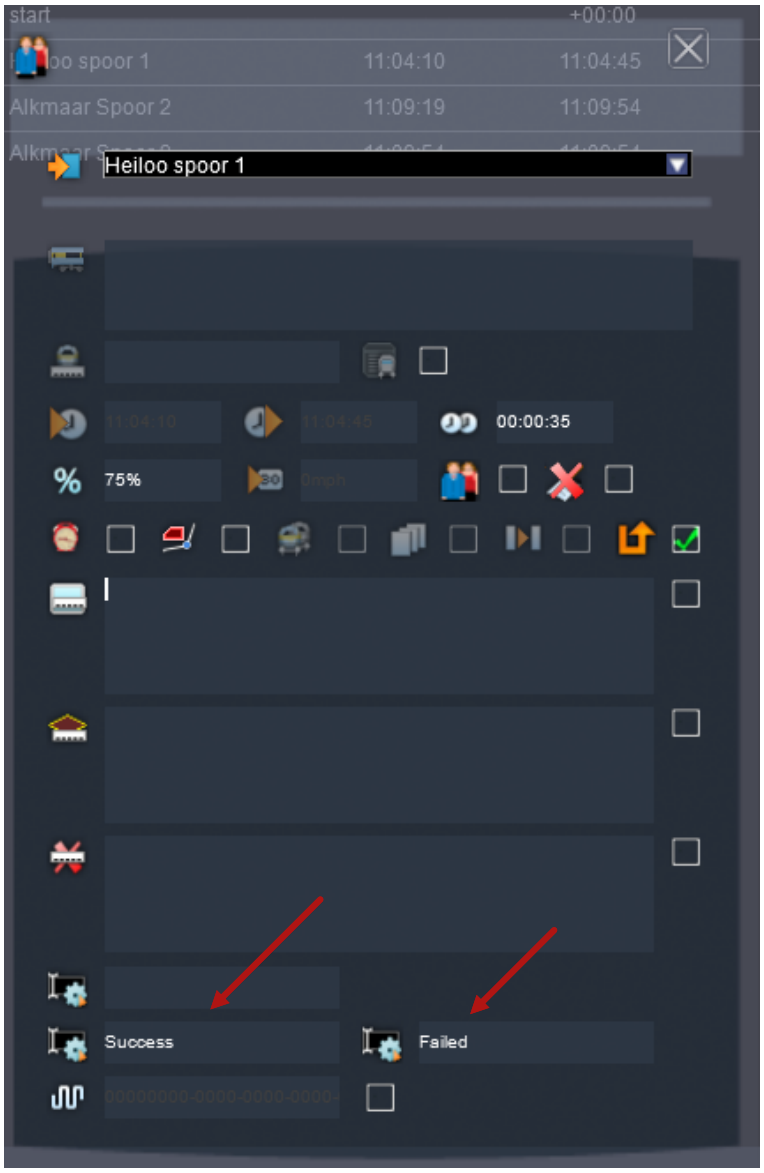


Figure 22. Adding event triggers for success and failure situations

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

The names used are samples. Any identifier is allowed. If you want to use the same event handling for both success and failure, use the same name for the event in the instruction, e.g. like in figure Figure 23.

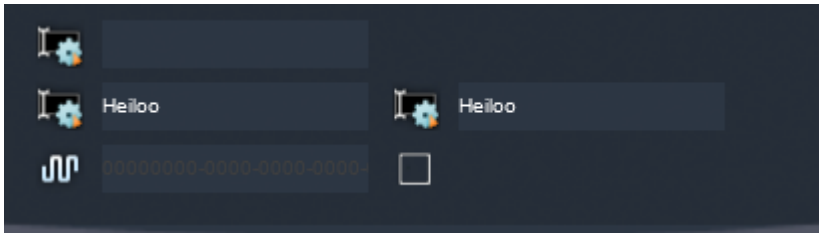


Figure 23. Triggering an event irrespective of success or failure.

3.5 Event handling for the last instruction

In the example in Figure 25. the scenario ends at another location than the location of the last instruction. In this case, the execution of the event after picking up passengers at TestTrak Platform 1 is deferred till the final destination is reached. This is consistent with the behaviour of the default event handling, which requires you to include a success message in the last instruction and not in the final destination event.

You should be aware of this behaviour when you try using scripting functions. A simple solution is to add a “Stop at” instruction at the final destination. This solves the problem. See Figure 24.

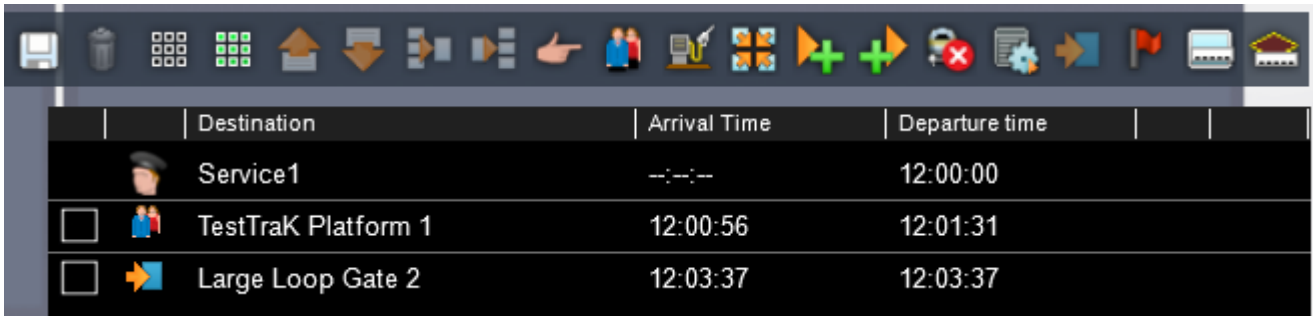


Figure 24. Scenario ends not at the last instruction

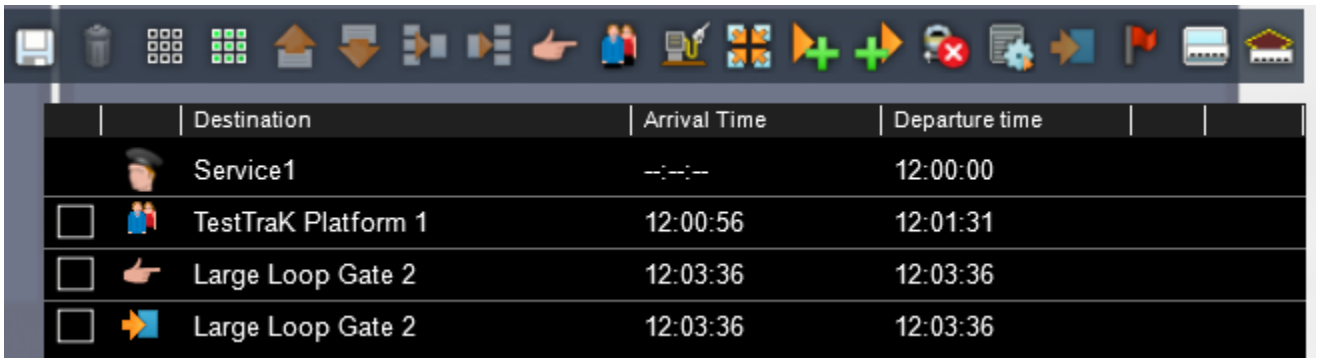


Figure 25. Fix for problem with deferred event handling.

4 Lua functionality for TS2015

In the previous chapters I provided the basics to get started. Now we will focus on the functionality TS2015 provides to scenario scripting.

The good news is that al lot of things are possible. The bad news is the lack of documentation for some aspects, especially for the different control values . You will need a lot of trial and error, even for simple actions like starting or stopping an engine.

The next chapters cover following topics:

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

Please, let me know if you discover new possibilities.

TS2015 distinguishes several different modules:

a) ScenarioManager
b) CameraManager
c) WindowsManager

d) WeatherController

For engines there does not appear to be an explicit management unit. You use specific commands to control an engine. The generic syntax for calling game functions from scenarios looks like:

```
SysCall(<command>, <param1>,<param2> ,…)
```

For engine and signal scripts, the API function **`Call()`** is used. In scenarios, always use SysCall(…) instead.

5 General scenario manager functions

5.1 Initialisation

This function works for scenarios, but is rarely used.

```
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.

5.2 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 after pausing the game:

```
function OnResume()

      HideFireworks();

end -- function OnResume()
```

The example is from a Horse Shoe Curve scenario, using fireworks.

Unfortunately, there is no way to save your script status when you save your scenario. Technically it is possible to save data, but there is no event defined you can use to run this code, though engine scripts listen to the OnSave() function.

You ca try using deferred events to save status at regular intervals. I did not yet use this in scripting. When resuming a scenario, you need to find out if you restarted the scenario or if you continued a scenario and also if the simulation time is later than the saved data. I think it is possible, but a bit challenging to make it work properly.

5.3 Deferred events

```
SysCall("ScenarioManager:TriggerDeferredEvent", <event name>, <delay>);
```

This command allows you to trigger an event from inside a lua script. The second parameter is the <event name>, the third parameter represents a delay in seconds. So the when you call this function an event will be scheduled for later execution, like a delayed GOTO statement.

You can use this after an error condition occurred, for instance a speed limit violation. After this occurs you may want to wait a while to allow the driver to slow down.

```
SysCall("ScenarioManager:CancelDeferredEvent", <event name>);
```

will cancel an event you scheduled.

5.4 Locking controls

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

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

Do not forget to unlock the controls later:

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

Note:

For engines there also is a function to lock/unlock an individual control see section 9.6).

5.5 Forcing scenario completion

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

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

<message> is a text to show if the condition in this function is met.

Triggers a scenario failure and displays an appropriate message:

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

5.6 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. **TestCondition** is called at every frame, so it is resources intensive. See chapter 2.5 for more information on performance.

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
```

There also is a function to query the status of the test function. This function will NOT invoke the test, but just return the last returned result of the test function.

```
SysCall ( "ScenarioManager:GetConditionStatus", <condition> );
```

It will return one of

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

Note:

You should be aware that if the user saves the game and continues after restarting TS2015, state information of a condition check may be lost. Unfortunately, there is no function in the interface like “OnSave” that allows you to save the scripting state when you save the game.

5.7 Hidden events

Starting with TS2015 it is possible to hide instructions from the driver instructions list. There is a lua function, that can make hidden instructions to show up:

```
SysCall("ScenarioManager:UnhideDriverInstruction", index)
```

The parameter **index** is the index of the instruction to show. I did not test the function yet, so it is not sure if the index starts at 0 or 1. You should be aware that if you add any instructions, the **index** may change. This makes the use a bit tricky.

6 Displaying messages

6.1 Introduction

TS2015 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 mark-up. This allows the use of images, tables, colours and other fancy mark-up.

For programmed messages, lua will give you an error message if something is wrong, but unfortunately it does not give any further details (see Figure 26).

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 messages which offer fine grained control in placement and behaviour
4. HTML messages with advanced mark-up features.

Figure 26. Error sample of message function.

6.2 Simple message

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

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

It will display a **`message`** text at the centre of the screen. The **`title`** is optional. If you insert one text parameter in the function call, only the message is shown, if you insert a title, the title is shown as well.

```
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.

Figure 27. Simple message example.

The parameter type allows to set the message type, choose from

```
INFO=0
ALERT=1
```

If you choose the ALERT version, a small message box in the upper right corner is shown.

Figure 28. Simple alert message example

6.3 Alert message

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

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

Example:

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

Results in this output:

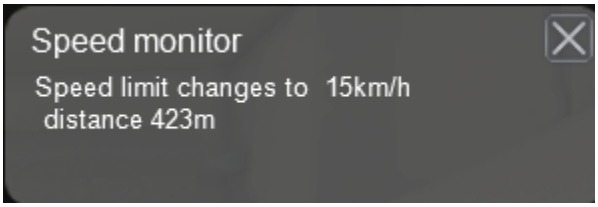


Figure 29. Alert message example.

Note: I used "\n" in this example to force a new line in the text.

You may add the name of an event to be triggered as the last parameter. (I did not try this).

6.4 Info message

The basic function to display messages has this syntax:

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

The message types discussed before, are in fact info messages, with preset parameters.

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, if you want to give a complex instruction to the user while driving. Note: do not use the boolean values **`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 annoying 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 message box (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);
```

Figure 30. InfoMessageExt example

6.5 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:

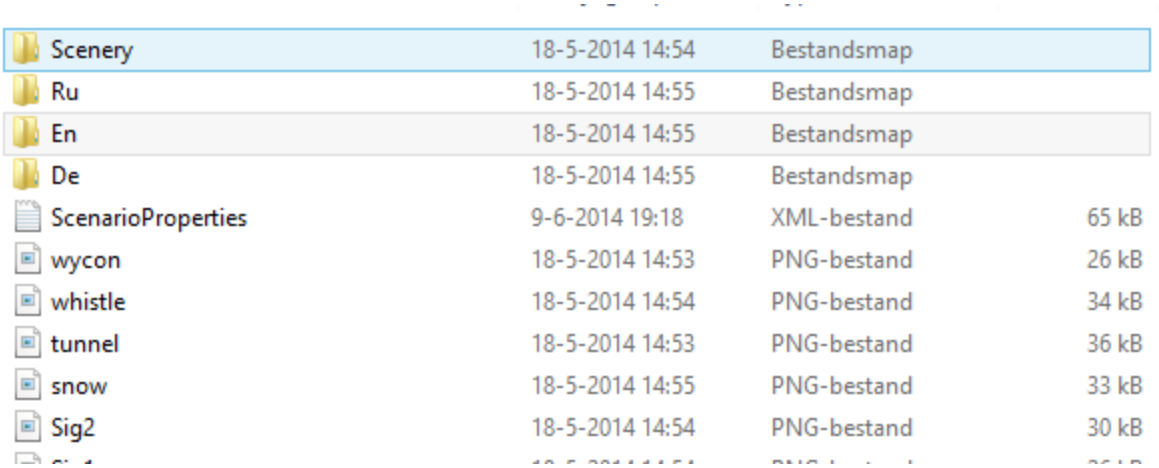


Figure 31. Localization folder structure for scenarios

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.

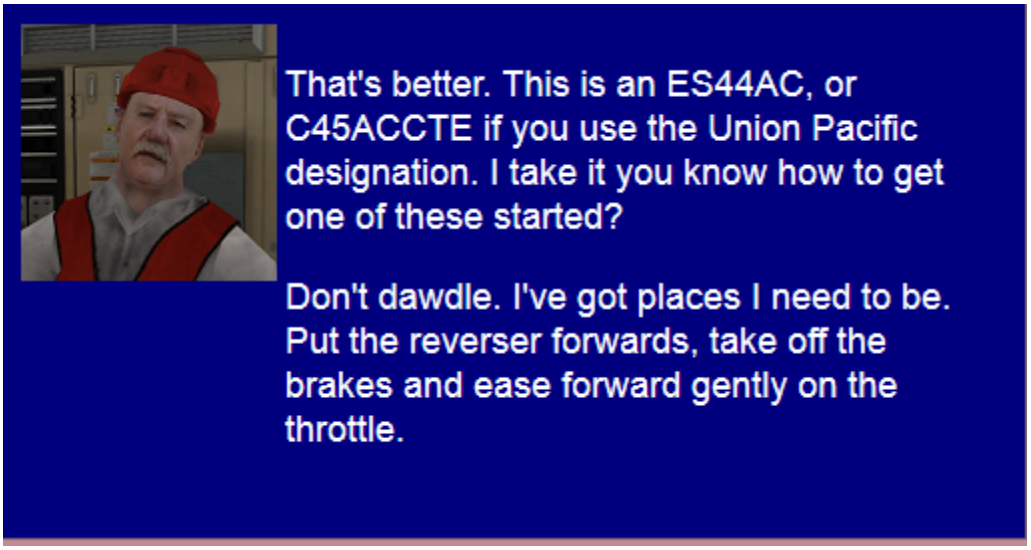


Figure 32. HTML message

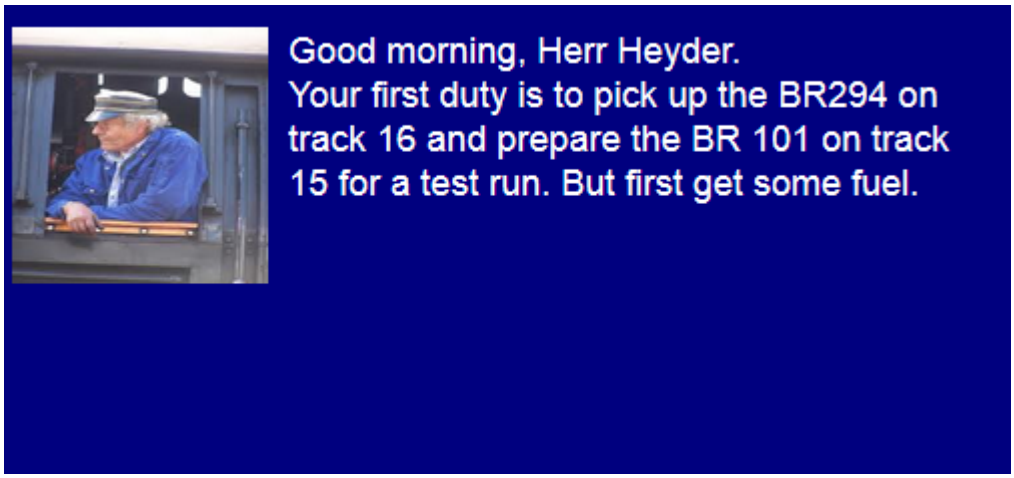


Figure 33. Example Improved layout for html messages using table constructs

The sample in Figure 32Figure 33 is not optimal, but it is easy to improve. In Figure 33 you see the top line of the text is properly aligned with image. The html code is shown in Figure 34.

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).

```
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>
```

Figure 34. HTML code for improved layout.

One last example, showing more complex use of tables:

Figure 35. By lua code generated html message

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">
Invalid destination
                      </font>
                </td>
      </tr>
           <table>
                <table width="350">
           <tr>
             <td>
                  <font color="#FFFFFF" face="Arial" size="4">

The destination
<font color="#eeee00" face="arial" size="4">
Alkmaar
</font>
 is not a valid destination for this consist. Use one of the destinations
from the list below. Take care to use the <b><i>exact</i></b> wording form
the list.

               </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">
<omitted some lines here>

           </table>
    </font>
  </body>
</html>
```

The html code for this sample is completely generated in a lua script.

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.

6.6 Keeping messages for later review

You may have noticed in the tutorials DTG provides for some locos that the messages containing instructions can be reviewed later in the game. This can be achieved with this function call:

```
SysCall("RegisterRecordedMessage", <Start function>, <Stop Function>,
immedately);
```

This function does several things:

1. It adds a function to a table with functions.
2. If **`Immediate`** is **`TRUE`** then execute the **`<Start function>`**

It creates a screen element that allows the user to browse through all registered messages

A reference to a function is saved, so in principle you can use this for other purposes than displaying messages, though at the moment I cannot think of any useful application.

You also must provide a stop function **`<Stop function>`** as the third parameter, to enable the system to stop executing a function.

If the last parameter, **`immediate`** is **`FALSE`**, nothing will happen, the function will be added to the table. An example showing how it works:

```
function DisplayThrottle()

     SysCall ( "ScenarioManager:ShowInfoMessageExt",
     "7a2a0b99-678b-4142-a930-f0509a154deb", "controlsThrottle.html",
     15, MSG_LEFT + MSG_TOP, MSG_REG, TRUE );
     SysCall("WindowsManager:HighlightControl", "Regulator", 15.0, 0);
     SysCall("ScenarioManager:LookAtControl", "Regulator", 5.0, 4.0, 0.5);
end
```

```
--functtion to cancel the displaying for the throttle
function StopDisplayThrottle()
     SysCall("WindowsManager:StopHighlightControl", "Regulator");
end

function OnEvent(event)
     if event == "throttle" then

          SysCall("RegisterRecordedMessage", "DisplayThrottle",
          "StopDisplayThrottle", 1);
          return TRUE;

     end
     return FALSE
end
```

Note that you first must create two functions, the first to start the action, the second to stop the action. You probably can use other instructions than messages in the stored functions. You may use this to give the user a bit more control over a sequence of actions. I did not yet try to use this.

Then, in the event handling function, you store and display the **`DisplayThrottle`** function, which shows an html popup message and highlights the throttle control. **`StopDisplayThrotl`**e stops highlighting the throttle control in the HUD.

This function is heavily used in the tutorials, but you can use them in other scenarios as well.

Figure 36. The user can cycle through stored functions.

7 Camera control

TS2015 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.

The cinematic camera is not covered in this version of the guide yet. It will come in a next edition.

7.1 Camera activation

The function to activate a camera is:

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

If the **`duration`** parameter is set to a value >0, the camera view returns to the previous view after **`duration`** seconds. You cannot use multiple **`ActivateCamera`** commands just one after the other to create a sequence. If you want to do that, you must create an event chain, for instance by using deferred events (see chapter 5.3) to control the sequence. The user can always overrule camera views. Maybe locking controls will prevent this.

Parameter	Description	Comment
command	Fixed value	
camera	Camera name, see list below	
duration	Time in seconds during which this camera is active	Any value >0 will cause the camera return to the previous camera at the end of the time. A value 0 will set the camera view for an indefinite time

The cameras use these names:

Camera	Lua name	Keyboard mapping
Cab camera	CabCamera	1
Frontview camera	ExternalCamera	2
Left head out camera	HeadOutCamera	Shift+2
Track side camera	TrackSideCamera	4
Passenger view	CarriageCamera	5
Coupling view	CouplingCamera	6
Yard camera	YardCamera	7
Free camera	FreeCamera	8
	CinematicCamera	

Following example can be used to force cabview:

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

You can force cabview as well (and more easily) by setting the check mark in the scenario properties dialog:

Figure 37. Force cab camera at load time.

7.2 Advanced camera functions

The camera can be targeted to a specific object. A nice example is used in the tutorials. The control of interest on the HUD is highlighted (see chapter 0 for details), the camera zoom into the control using this function:

```
SysCall("ScenarioManager:LookAtControl", <Control name>,
lerpTime,duration,dstFov);
```

Parameter	Description
lerpTime	The time (in seconds) of how long the camera should take to focus on the control (i.e. lerp to the control)
duration	How long the camera should focus on the control
dstFov	(degrees) the field of the view the camera will zoom to over **`lerpTime`** amount of time.

This function will lock the controls during **`lerpTime+duration`** seconds.

An example is provided in chapter 0.

In a similar way it is possible to create a scenery object. Like waggons and markers can get an in-game identification, scenery objects like people can get one as well. You can direct the camera to a scenery object using:

```
SysCall ( "CameraManager:LookAt", <ObjName> );
```

You can use this function to focus on specific rail vehicles or at named scenery objects. Like you can add wagon numbers to an engine or wagon in the scenario editor, you also can give a name to any scenery item.

To use LookAt for scenery items, you need to select the free camera first. For rail vehicles this is not necessary.

You can also revert to the default camera view (whatever that may be), using:

```
SysCall("ScenarioManager:RestoreCameraToDefault", time);
```

The parameter **`time`** allows to specify the time in seconds it should take to go back to the desired position.

The following call relocates the free camera to a specific location. If **`result==true`** the operation succeeded. (Not tested by me).

```
Result=SysCall("CameraManager:JumpTo", longitude, latitude, height)
```

7.3 The cinematic camera

Coming in the next edition of this guide!

8 Tutorial functions

You may have wondered if it is possible to use the advanced functions in tutorials by yourself. The good news is that this is possible. You have seen already the feature to browse through the tutorial messages in chapter 6.6. In this chapter it is shown how you can create the red boxes around a control on the HUD and zoom into this control.

Figure 38. Highlight a control.

The first step is to draw the red rectangle using this function:

```
SysCall("WindowsManager:HighlightControl", <control>, <duration>, <style>);
```

In this call, <control> refers to the name of a control at the HUD. <duration> determines how long the highlighting must be shown. The third parameter is not yet used. A usage example:

```
SysCall("WindowsManager:HighlightControl", "Regulator", 15.0, 0);
```

You can programmatically overrule the highlight, e.g. if the user sets the control to a specific value, which would mean the user has found the control.

```
SysCall("WindowsManager:StopHighlightControl", <control>);
```

You can use then ScenarioManager to direct the camera to the control (see also chapter 7.2)

```
SysCall("ScenarioManager:LookAtControl", "Regulator", 5.0, 4.0, 0.5);
```

And finally reset the camera view:

```
SysCall("ScenarioManager:RestoreCameraToDefault", 5.0);
```

The HUD controls I found in various samples until now are listed below:

Highlight name	LookAtControl name	Description
Regulator	Regulator	Regulator/Throttle
Reverser	SimpleChangeDirection	Reverser
TrainBrakeControl	TrainBrakeControl	Brake lever
TrainControl		Loco brake
VirtualBrake		??
DynamicBrake		Dynamic brake
TaskLog	TaskLog	Task log button
LoadUnload		Load/unload button
CamPreviousRV		??
CamNextRV		??
Speed		Speed indicator on the HUD

For the reverser I found an inconsistency in the example It is not yet clear why, maybe it's used to distinguish for the reverser of a steam engine, which works different.

An example of the code is provided below:

```
function DisplayThrottle()

     SysCall ( "ScenarioManager:ShowInfoMessageExt",
          "7a2a0b99-678b-4142-a930-f0509a154deb", "controlsThrottle.html",
          15, MSG_LEFT + MSG_TOP, MSG_REG, TRUE );
     SysCall("WindowsManager:HighlightControl", "Regulator", 15.0, 0);
     SysCall("ScenarioManager:LookAtControl", "Regulator", 5.0, 4.0, 0.5);
end

--function to cancel the displaying for the throttle
function StopDisplayThrottle()
     SysCall("WindowsManager:StopHighlightControl", "Regulator");
end
```

9 Engine controls

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, the consist number you enter in the scenario editor.

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

8510 is the reference to the engine or van, as shows in the right hand fly out when selecting a consist (Figure 39).

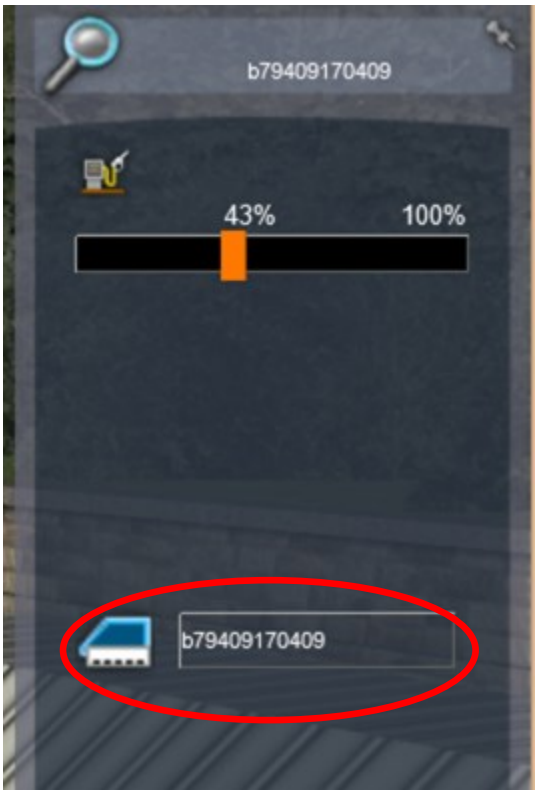


Figure 39. You can use this property to set a name for a rail vehicle.

This makes engine commands work on both player engine and AI, but some commands are useful on coaches or waggons as well.

9.2 Discovering control names

What can you do to find out which controls can be used?

One way to quickly see what controllers are on a specific locomotive is to run the game with a special command line parameter **`-ShowControlStateDialog`**

This launch parameter will cause TS2015 to show a dialog with all controls for a particular engine or van, including the actual value that is set. See chapter 0 for instructions to set a launch parameter. You need to run TS2015 in windowed mode to see this dialog (see chapter 2.7).

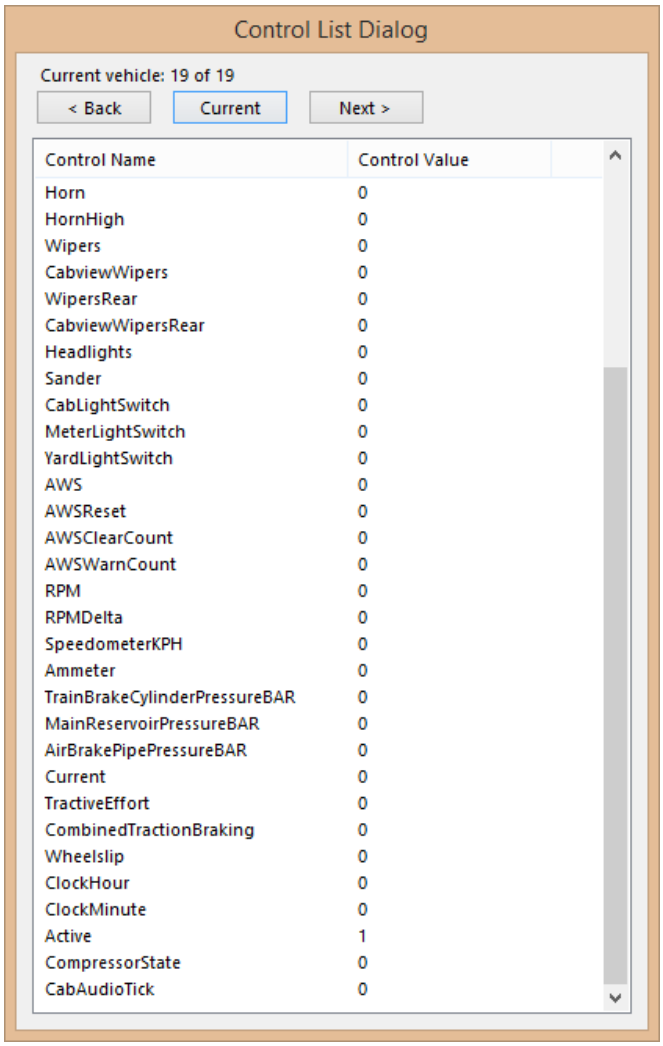


Figure 40. List of all controls for a consist.

Unfortunately it is not possible to use copy/paste to retrieve the list. It can be very helpful both in creating scripts and debugging, but there is no guarantee that the controls can be set from a scenario script.

You can also have look at the controls listed in appendix A. This list comes from the original wiki and there is a fair chance this may help you.

As a last source, have a look in the asset folder of the engine. You may look for files with the extension `.out`. These are compiled lua programs containing the engine simulation functions. I think it is possible to decompile them and to obtain the code, but it will not be very readable. I never tried until now. But you can open the files in a hex editor and look for readable strings that look like commands.

When you found them, use the lua calls in section 9.5 and section 9.7 to find out if the control is recognized and to find the value range. This may work or it may not work.

It would be nice if you are willing to share the information you found with me for eternal fame and for the community. I will publish it in the next version of this guide.

9.3 Set a control to a value.

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

```
SysCall("PlayerEngine: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.

9.4 Get the current value of a control

With the next call you can retrieve a current value.

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

This function will 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)
```

This function 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.

9.5 Test if a control exists

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 uses the logging system, see chapter 2.6.5 and the code listing in chapter 11.2

9.6 Lock or unlock a specific control

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

Method that will lock a control in place so it cannot be changed in game. The parameter **<ControlValue>** is the control to be locked. **index** is the index of the control in an array of controls with the same name and **value** is the value to determine whether the control is locked or not, with 1 being locked in place and 0 being unlocked.

You may want to define symbols for this to increase readability, e.g.

```
LOCK=1
UNLOCK=0
```

You can use this function to check if a specific control is locked:

```
SysCall("PlayerEngine:IsControlLocked","<ControlValue>")
```

9.7 Get the minimum or maximum value of a control

There are two system call that retrieve the minimum or maximum value for a control:

```
result=SysCall("PlayerEngine:GetControlMinimum", "Startup", 0) or
"undefined"
result=SysCall("PlayerEngine:GetControlMaximum", "Startup", 0) or
"undefined"
```

If the value is not available, the function will return **nil**. In the sample this is solved by adding a default value, because if you want to display a string, lua does not accept **nil** values.

9.8 Engine controls

The big problem is the lack of documentation. Engine functions can be very specific for a 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 of my experience:

- **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 not switch off headlights for a German BR151.
- I managed to use **DestinationBoards** on all ChrisTrains models to set destinations, but for the latest versions it doesn't work properly anymore.
- I succeeded in using **DoorsOpenCloseLeft** and **DoorsOpenCloseRight** to get door status, The function only works for the wagon you directly address, using the reference number of the van. You can refer to the **PlayerEngine**, but the control does not exist for a separate locomotive if it does not have its own passenger doors. The function did not work to set the door state. When I tried, it opened the doors, but no sound and the doors closed again immediately.
- I succeeded in getting **AWS** state, but could not control the behaviour of AWS, e.g. set AWS works, but you cannot use the Q button to clear it.

- I did not succeed in set **Wipers** on or off in the German BR151.
- The **Startup** command works on the vR Köf to turn the engine off, but I could not use the command to set an initial state for the engine motor to be off, because the command causes the engine to perform the complete animation to turn off the engine.
- I managed to activate the ATB (Dutch security system, like the German PZB but much easier to understand) for the ChrisTrain models, using **ATBEG** or **ATBNG** as control name.
- It is possible to set **Throttle**, **Reverser** and **Trainbrake**, e.g. to force an emergency stop:

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

This sample limits the cruise control for German loco's (e.g. Munich-Augsburg, BR101 etc) to 40 km/h.

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

(I did not test this script).

9.9 Rolling start

If you just want to use scripting to set the initial state of player engine, e.g. create a dead engine with all functions turned off, it can be much easier to use the Rolling Start feature. The rolling start feature is able to store other engine parameters that just the speed as well. It does not always work, so you need to try and see what works. Some people report it is not portable to other computers, but rolling start just creates another type of save file for the scenario.

It works like this:

1. Create your scenario, tick the rolling start check box on the scenario properties dialog (Figure 41).
2. Start playing the scenario. When you have done all settings (eventually bringing the consist to the desired speed), use **CTRL+F2** to save the scenario. This invokes a special save mode for rolling start.
3. Play the scenario again to verify the settings.

Note:

If you edit your scenario, it is recommended to remove the files **StartingSave.bin** and **StartingSave.bin.MD5** These files may corrupt your scenario if you add or remove events.

The rolling start seems not save all settings. You need to check if it works as you intended.

Figure 41. Enable rolling start.

9.10 Track information

```
SysCall("PlayerEngine:GetGradient")
```

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

```
SysCall("PlayerEngine:GetCurvature")
```

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

```
SysCall("PlayerEngine: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.

9.11 Consist information

```
SysCall("PlayerEngine:GetConsistLength")
```

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

```
SysCall("PlayerEngine:GetConsistTotalMass")
```

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

See also the programming example in chapter 11.3

```
SysCall("PlayerEngine:GetRVNumber")
```

Function will return the number of the rail vehicle (RV) that called the function. 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.

There is function to set the RV number as well:

```
SysCall("PlayerEngine:SetRVNumber",value)
```

Not tested, but it can be very useful to set destination boards dynamically, and maybe as well to set correct vehicle numbers using a script. I did not test this function yet.

```
SysCall("PlayerEngine:GetTotalMass")
```

Method that will get the total mass of the rail vehicle, inclusive of ballast/fuel, measured in tons. Not sure if this works for waggons as well.

```
SysCall("PlayerEngine:GetFireboxMass")
```

This function will retrieve the mass of the firebox as a fraction of its maximum value.

```
SysCall("PlayerEngine:GetConsistType")
```

Returns the consist type. This may be nice to add to a report for the driver. (Not tested)

It may return one of the following values:

```
eTrainTypeSpecial = 0
```

```
eTrainTypeLightEngine = 1
eTrainTypeExpressPassenger = 2
eTrainTypeStoppingPassenger = 3
eTrainTypeHighSpeedFreight = 4
eTrainTypeExpressFreight = 5
eTrainTypeStandardFreight = 6
eTrainTypeLowSpeedFreight = 7
eTrainTypeOtherFreight = 8
eTrainTypeEmptyStock = 9
eTrainTypeInternational = 10
```

9.12 Speed

Thomas Ross uses them in a script for Western Lines of Scotland to monitor speed limits for steam engines.

```
SysCall("PlayerEngine:GetSpeed")
```

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

You will need to perform a calculation to get the speed in km/h or Mph The correct factors are:

`conversion= 3.6` to obtain km/h

`conversion= 2.236932` to obtain Mph

```
SysCall("PlayerEngine:GetCurrentSpeedLimit", separate)
```

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

If you set separate to value 1, it will return both track speed limit and signal speed limit e.g.

```
trackLimit, signalLimit = SysCall("PlayerEngine:GetCurrentSpeedLimit", 1)
```

```
type, speed, distance=SysCall("PlayerEngine:GetNextSpeedLimit",
direction,mindistance, maxdistance)
```

This function 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 `type` can have:

- 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.

If type has the value -1, then **`speed`** and **`distance`** will have value **`nil`**.

`mindistance` is the minimum distance to look ahead. This helps to find a speed limit hidden behind a near speed limit.

`maxdistance` is the maximum distance to search for changes, default is 10km.

```
SysCall("PlayerEngine:GetAcceleration")
```

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

9.13 Signalling interaction

9.13.1 Signals

You can obtain signal states using following function:

```
type, state, distance, aspect=
SysCall("PlayerEngine:GetNextRestrictiveSignal", direction, distance,
lookfromdistance)
```

This function will gather information about the next signal. 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.

The function returns 4 values: **`type`**, **`state`**, **`distance`** and **`aspect`**.

`type` is representative of the values that follow it, and can be one of:

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

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

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

`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.

`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.

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

Let me know the proper values for other signalling systems if you discover them.

9.13.2 AWS functions

There are three AWS functions that can be used:

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

It is possible to retrieve season and time of day.

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

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

Spring = 0, summer = 1, autumn/fall = 2, winter = 3
local timeNow = SysCall("ScenarioManager:GetSimulationTime", 0)
```

Day and night times do not very accurately follow the day, but depend on the season. Night times are:

- Spring- 18:45 to 08:15.
- Summer- 21:00 to 05:30.
- Fall- 19:45 to 06:45
- Winter- 17:00 to 09:15.

10.2 Weather

You can control weather sequences using LUA scripting using statements like:

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

weatherId probably refers to a weather pattern blueprint. I have not yet tested this.

10.3 Play audio

You can play audio files in .wav or .dav format using LUA scripts with the function:

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

It is possible to play multiple sounds simultaneously. 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 is used in other parts of TS2015).

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.”

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

will interrupt playing the sound file.

To know if a sound file is playing, use:

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

Note:

Maybe you cannot use this function for workshop scenarios, because workshop may not upload the audio file

10.4 Play video

For video add the following command:

```
SysCall("ScenarioManager:PlayVideoMessage", videoAddress, type, paused,
controls, style )
```

Parameter	Description
videoAddress	The file path to the video (DTG documentation says it must be .flv format, but .ogg seems to work as well)
type	FullScreen (0) Front(1) large, at the centre of the screen

	VideoCall(2) small, upper left corner, useful to give instructions, e.g. a conductor etc.
paused	If TRUE pause the game (not sure, maybe the Boolean type must be used here)
controls	• play 1 • pause 2 • stop 4 • seek 8 You can use addition of the values to show more controls.
style	Not yet used,

Example:

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

Ensure the ogg file is in the **en** folder.

You can stop playing the video:

```
SysCall("ScenarioManager:StopVideoMessage", videoAddress)
```

Or query if the message is still playing:

```
SysCall("ScenarioManager:IsVideoMessagePlaying", videoAddress)
```

(Not sure what happens if the player pauses the video)

Note:

Maybe you cannot use this function for workshop scenarios, because workshop may not upload the video file

11 Scripting application examples

11.1 Emergency brake

You probably will want to 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);
```

This code is straight forward. You can create a function to hide the details and make it more advanced, or just copy the lines into a script.

11.2 Logging your debugging messages

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.lua`** 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
```

11.3 Train length and train mass

This script will obtain your train length and mass and report it to the driver. In reality this should be a standard procedure.

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

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
```

Suggestions for improvement: create an html file on the fly and display a neat report.

11.4 AWS audible alert outside the cab view

This script creates its own AWS alert sound. Useful for engines without audible signal, or for using camera positions outside the cab.

```
--[[
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. Until now I am not able to find out which is the actual 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.

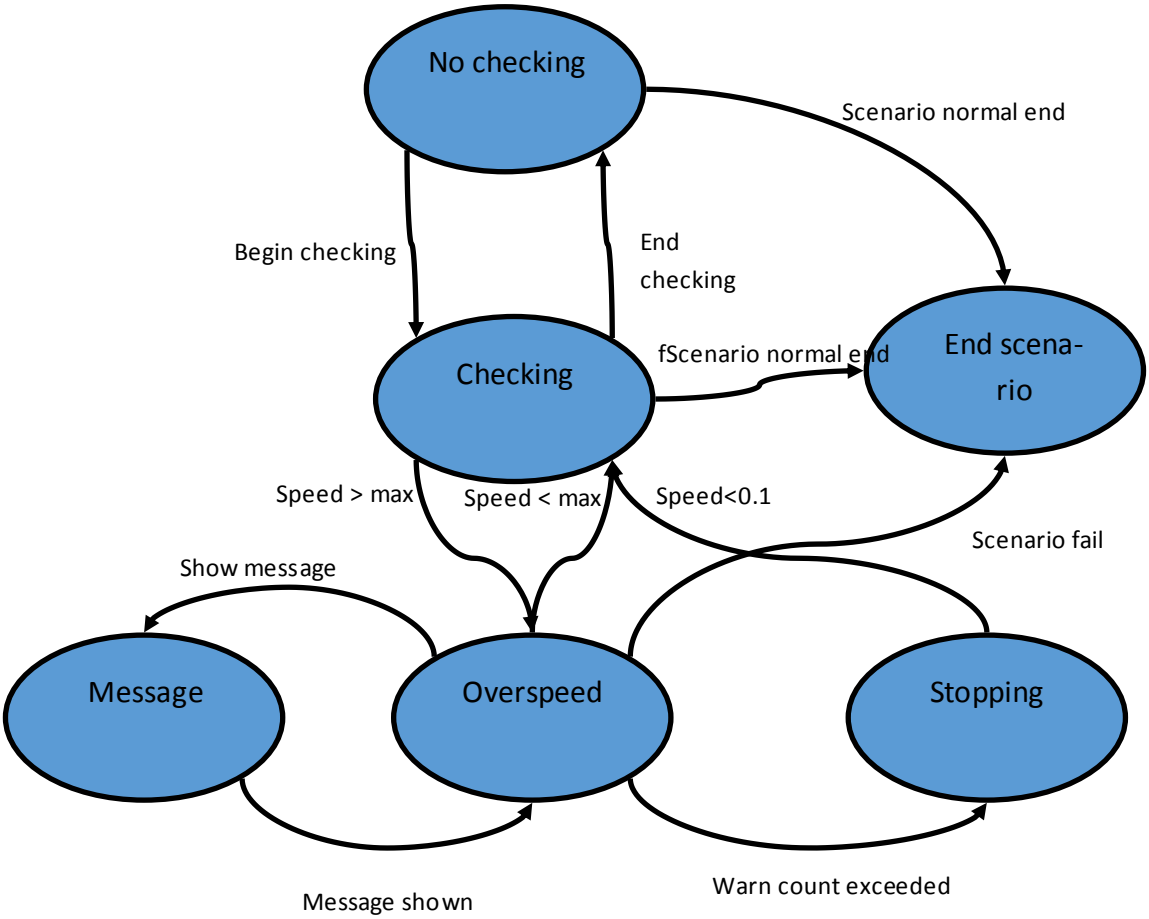


Figure 42. State diagram for speed monitoring application.

To understand the design, please look at the state diagram in Figure 42.

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.lua** 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. Finally, 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 B.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 **BeginChecking** starts 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 if 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 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 appendix B.3 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**. (See also chapter 10.3).

3. You need to create a condition check, that calls the **`SpeedMonitorg.Check()`** function (see chapter 5.6 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.new(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. 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
:GetNextSpeedLimit",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

11.7 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.

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"
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",se
lf.index) or ERROR

self.DoorsRight=SysCall(self.van ..":GetControlValue","DoorsOpenCloseRight",
self.index) 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",
      "conductorwhistle.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 Control functions

Following paragraphs show the names of default controls used. They may or may not work properly when activated in scripts.

A.1 Train Controls

Control name	Description	Comments
SimpleThrottle	Speed Up / Slow Down	
SimpleChangeDirection	Switch Directions	
Reverser	Increase / Decrease Reverser	
Regulator	Increase / Decrease Throttle	
CombinedThrottleBrake	Combined Throttle and Brake	
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	Represents the actual state of the AWS
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	Value 1= start, value -1= shut down
Wheelslip	Wheelslip	Used for visibility objects?
DoorsOpenCloseLeft	Open / Close Doors on left side	
DoorsOpenCloseRight	Open / Close Doors on right side	

A.2 Electric Locomotive Controls

Control name	Description	Comments
PantographControl	Raise / Lower Pantograph	0= down, 1= up
FrontPantographControl	Raise / Lower Front Pantograph	
RearPantographControl	Raise / Lower rear Pantograph	

A.3 Steam Locomotive Related Controls

Control name	Description	Comments
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	

A.4 Output values for Dials

Control name	Description	Comments
Speedometer	Speedometer	
Ammeter	Ammeter	
RpmDial	Revolutions per minute	
Accelerometer	Accelerometer (kN)	

A.5 Gauges steam

Control name	Description	Comments
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

A.6 Gauges brakes

Control name	Description	Comments
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	
MainReservoirPressure**	Pressure in the main reservoir	
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*

A.7 Gauges miscellaneous

Output controls added for the sound system

Control name	Description	Comments
Current	Current in Amps	
TractiveEffort	Tractive Effort	
RPM	RPM in revolutions per minute	
RPMDelta	Lagged and processed RPM DELTA value	
CompressorState	Compressor State	On(1.0)/Off(0.0)

B. Useful lua constructs

B.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)
```

B.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
```

B.3 Object classes

In lua you simulate object classes. This is done using a metastable. In the example a class **DebugTS** is created. You can create an instance by assigning the new function to a lua 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
```

Index