Signal Setup & Scripting Guide

Release version 1.0 (2007-11-09)

1 Introduction

Every signal in the Rail Simulator world is an *instance* of a particular signal *blueprint* which defines that signal type. The blueprint controls everything from the 3D model that's used in the game and the name that's displayed in the Editor to which LUA script is used to control the signal's behaviour.

These scripts react to events such as a train passing the signal or a message being received from another signal, and then change the appearance of the signal as appropriate, switching lights on or off or animating any moving parts the signal might have.

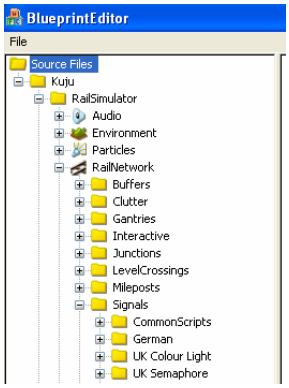
This document will explain how to setup your own signals using our Blueprint Editor and the LUA scripting language, and how the various signal scripting functions work. It assumes that you have some understanding of LUA, although even without any scripting experience you should still be able to make modifications to the existing scripts to make them work for your own signal.

2 Setting up and using Signals

To begin with, we're going to show you how to define a signal's characteristics using the blueprint editor, export that signal ready to use in the game, and then place it on a piece of track in the editor.

2.1 Using the Asset Editor

The Asset Editor is a simple tool that allows you to create and edit blueprints for objects that you want to place in the game world. In this case a signal. Let's start by opening the Depot and finding a simple signal.



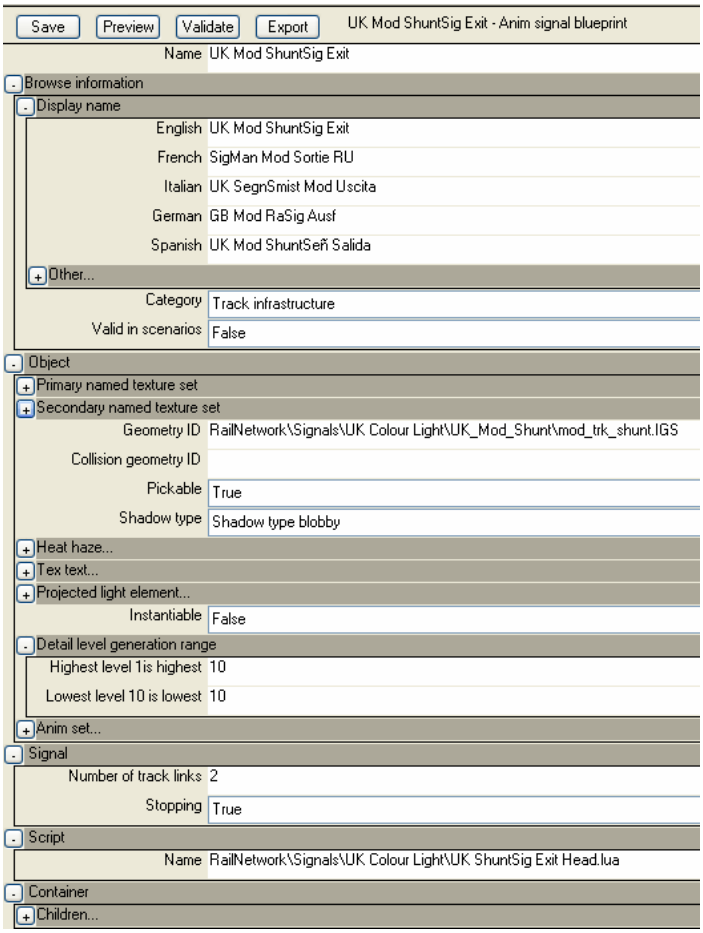
- Start up the Asset Editor using its shortcut in the Rail Simulator Start Menu group.
- Click on the + next to where it says "**Kuju**" to expand the list of **Source Files**.
- Do the same for **RailSimulator**, **RailNetwork**, **Signals, UK Colour Light**
- In this folder you'll find all the scripts that control the signals. We'll deal with them later. For now, scroll to the bottom of the list. Here you'll find folders containing blueprints for the modern UK signals. Expand the **UK_Mod_Shunt** folder.
- Double click on **UK Mod ShuntSig Exit**.

The Asset Editor will now display the blueprint for the UK Shunt Exit signal. The entries you should know about for now are –

- **Display Name** – The name displayed for this signal in the object list in the Editor, for each of the languages that is supported by Rail Simulator. The Display Name needs to be unique, or you won't be able to tell which signal is which. It also needs

to be fairly short, or the end of the name won't be visible in the editor, particularly at lower resolutions.

- **Category** – Signals should be flagged as *Track infrastructure* to ensure they appear in the correct part of the object list in the editor.
- **Geometry ID** – This is the 3D model used in the game for this signal type.
- **Signal - Number Of Track Links** – This sets how many *links* the signal will have when you place it in the world. We'll explain what this means in a moment.
- **Signal - Stopping** – Whether a train ever needs to stop at this signal. In most cases this should be *true* so that trains will stop at the signal if the track ahead is blocked, but for certain signals (such as a *distant* or *repeater* signal, which shows the state of the next signal up the line rather than the track itself) this should be *false*.



- **Script** – The LUA script which will be used to control this signal's behaviour.

To see what your signal is going to look like in the game, click on the **Preview** button just above the blueprint. This will load up the signal in the game engine. You may need to move the Blueprint Editor window out of the way to see it.

As in the route editor, you can click on the model and move it around. This signal consists of a single model, so moving it around won't actually change anything. But if your signal was made up of multiple models (for example, a signal light mounted on a post) you could change the relative positions of the models in this way to make sure they all lined up correctly (eg, making sure the light was mounted on the top of the post).

Once you're happy that your signal is setup correctly, you can save your changes by clicking on the **Save** button above the blueprint. **DO NOT** do this now, as this is a real signal used by the game, and changing it could break the signalling on the British routes. If you have made any changes to the blueprint, double click on another blueprint and click "no" when it asks if you want to save changes. Then go back to the shunt exit blueprint and you'll see that any changes you made to it have disappeared.

2.2 Exporting Blueprints

If you create a new blueprint or edit an existing one, before you can see your changes in the game you must first *export* the blueprint. This will compile all the source files referenced by the blueprint and copy them to the appropriate folders. To do this, just click on the **Export** button above the blueprint.

At the bottom of the blueprint editor window there's a debug output area. You may need to pull up the window separator that stretches across the screen below the blueprint to expand this area so that you can see the text displayed there.

If your export succeeded, you will see a list of files that have been compiled and copied, with the words "*Successful build*" at the end. Any errors will be flagged by a .

Don't worry if it didn't compile – this is a standard signal that shipped with the game, so you'll still be able to use it in the editor.

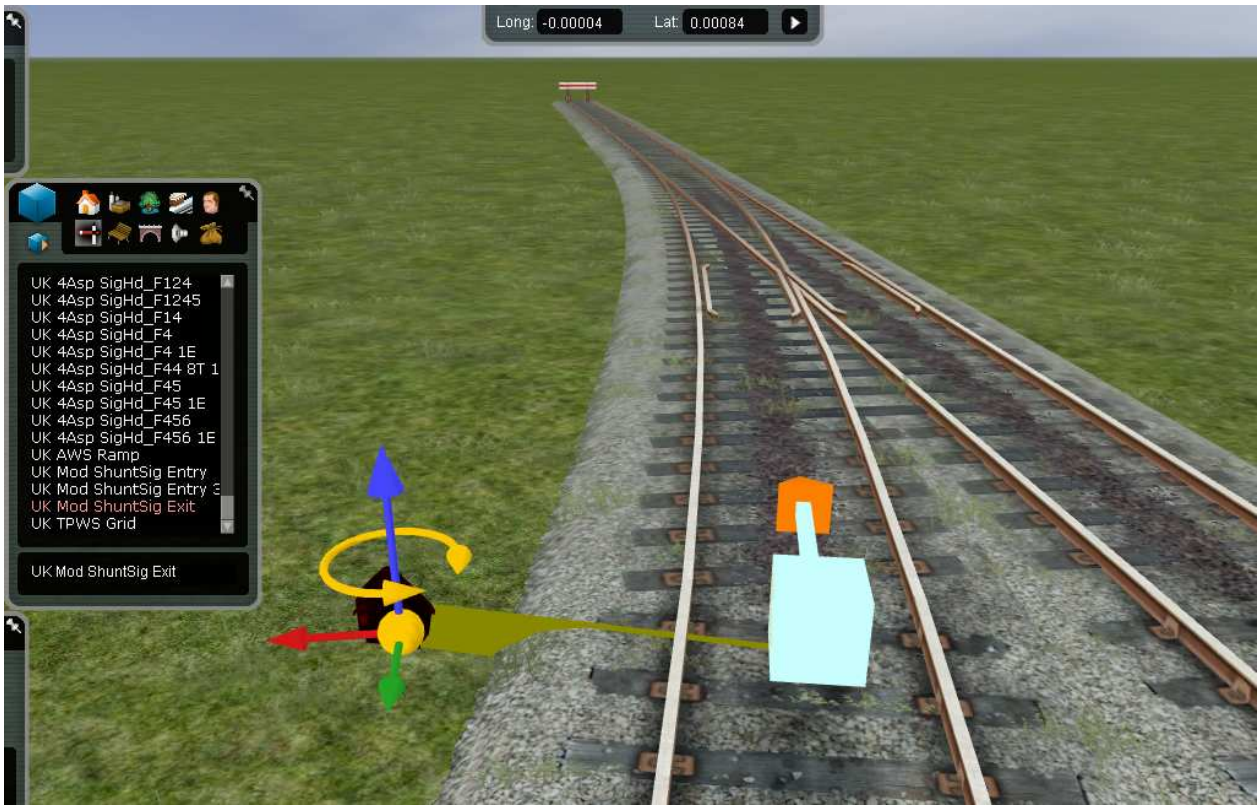
2.3 Signal Link Placement

Now shut down the Asset Editor and load the game. Select **Routes** -> **New Route** -> **Default** template, and give your new route a name. When the game finishes loading, select the **Linear Object Tools** mode and lay down a length of track with another line diverging from it to form a junction.

Then go back to **Object Tools** mode to place the signal. Click on the little signal icon in the second box down on the left to list all the signals ("Track Infrastructure" – remember this was one of the options set in the blueprint), and scroll down all the way to the bottom to find *UK Mod ShuntSig Exit* (the display name that was set in the blueprint for this signal).

Click on the signal's name and then position it so that it's just next to one of the two lines that split at the junction, a little way up the track. Press the left mouse button to place it in the world and (keeping the button pressed down) move your mouse left or right to rotate it until it's facing in the correct direction – if you're looking towards the junction, the front of the signal with the lights on it should be facing you.

As soon as you've placed the signal, you'll see a pale blue box with an arrow coming out of it, connected to the signal by a line. This is a *link*. The box shows you where the link is on the track, and the arrow tells you which direction it's facing in – we'll explain that later. For now, just move the cursor so that it's over the track just next to the signal. You'll notice that it snaps to the middle of the track. When you're happy with its positioning, click the left mouse button again to place the link there.



This signal has two links (again, this was set in the signal's blueprint), so you'll need to place the second link now.

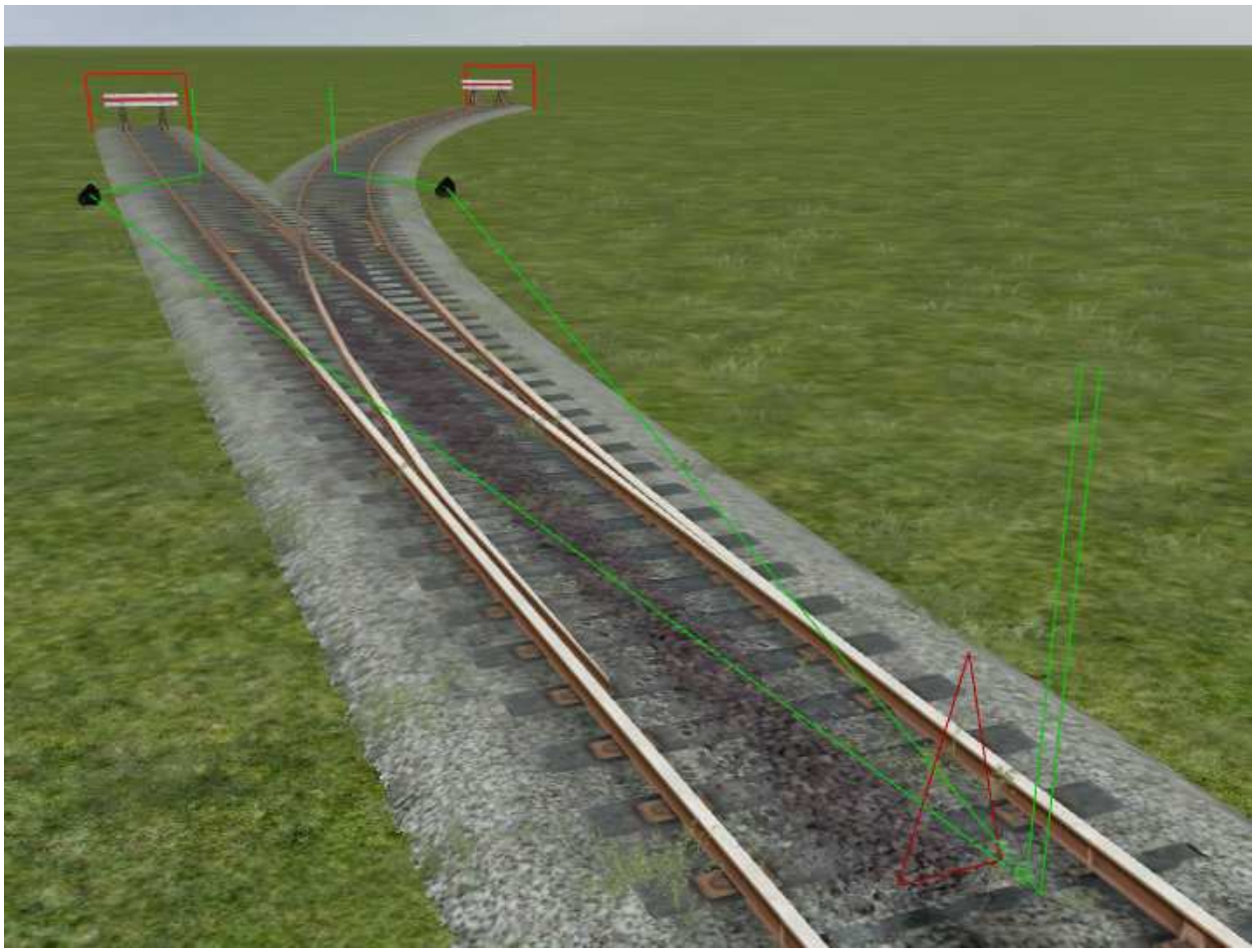
Notice that this second link has a number 1 above it. The first link you placed is link 0. Link 0 always goes near the signal it belongs to. Link 1 (and any other links after that) should be positioned further down the line, beyond the junction(s) that the signal is protecting.

In this case, the shunt signal's link 1 should go on the track a couple of meters beyond the junction, after the lines have merged. Again, hover the mouse cursor over the track and then click the left button when you've got the link in the right place.

You've now positioned all the links for this signal, and are ready to place another signal. Put a second signal next to the other line, and again position its first link near the signal and the second link beyond the junction, after the two tracks have merged together.

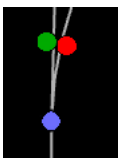
That's enough for now, so right click to stop placing signals. Let's just check that you've got the links in the right place. Click on the Linear Objects Tool button in the top left box to go into track editing mode. You should now see a red triangle hovering just above the track near the junction. This is the point at which the track splits ready for the junction.

Next press the space bar. This changes the track view mode in the editor, giving you useful visual information about track directionality and other settings. After you've pressed space a few times, you should see green lines appear on the track representing the position of the signal links.



As shown above, both signals should have their link 1 on the other side of the red triangle from their link 0. If the link 1 isn't far enough out, just select the signal whose link is in the wrong place, hover the mouse over its link 1 until it highlights, and then left click and drag the link along the track until it's in the right place.

Once you're happy all the links are correctly positioned, click on the Drive icon in the bottom right of the screen to save your route and go back into the game. The two signals will now initialise – one of them will keep its lights on, the other will turn its lights off.



If you go to the 2D map and zoom in on the bit of track you placed, you should see the junction represented by a blue dot. Just beyond where it splits you'll see a red dot representing one of the signals and a green dot representing the other. This indicates that one of the signals is blocked and the other is clear.

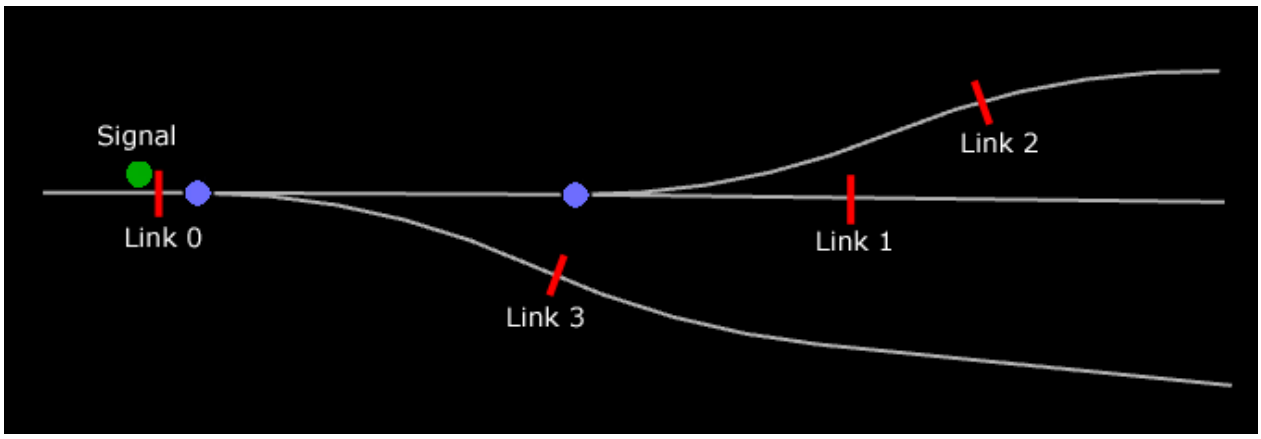
Hold down shift and left click on the blue dot representing the junction to switch it. After a second you should see the green dot turn red and the red dot turn green. Exit the 2D map and you'll see that the lights have switched as well. This shows that your signals are working correctly.

2.3.1 Link Placement Rules

Although the shunt exit signal is very simple, some junction signals have more than two links, and it's important to ensure that all of these are in the right places if the signal is to work correctly. So when you're placing signals, bear in mind the following rules -

- A signal's link 0 should always be placed on the track close to the signal.
- Any other links the signal has should be placed beyond link 0, further up the line.
- Every valid route for a train passing this signal going forwards should have a link on it. Link 1 normally goes on the route that's straight ahead, and any other links go on diverging routes to either side.
- These links should be placed beyond *all* of the junction(s) covered by this signal - there shouldn't be any more junctions between one of those links and the next home signal on that route.
- A signal's links should normally all be facing in the same direction. In other words, if you pass link 0 going forwards and keep travelling in the same direction, you should pass the next link belonging to this signal going forwards as well. The editor usually places links the right way round automatically, but it can get confused if you place a signal on a piece of track that curves by more than 90 degrees. If one of your links is facing in the wrong direction, you can reverse its direction manually by holding down SHIFT and clicking on the link.

Here's an example of correct link placement for a signal covering multiple junctions -



3 Basic Signal Scripting

So now you've seen what a signal blueprint looks like and how the settings in it effect the way the signal is placed in the editor, and you've positioned a pair of shunt exit signals to cover a junction.

Shunt exit signals just tell a driver which way the junction ahead of the signal is set. If the junction is ready for you to drive over it, the lights are on. If the junction is set against you (in which case driving over it would derail the train!), the lights are off.

In Rail Simulator, all of this behaviour is controlled by the script that was referenced in the signal's blueprint – "\RailNetwork\Signals\UK Colour Light\UK ShuntSig Exit Head.lua". In this section we'll be showing you how that works...

3.1 Script Functions

A signal script is made up of a series of *functions* that are called either by the game code or by other functions within the script. A function definition looks something like this –

```
-- This defines a function called MyFunction
function MyFunction()
        -- Does whatever it says in here
        -- And then we...
end
```

The shunt exit signal is a very simple case, as all it needs to do is keep track of the state of the junction(s) it spans, and switch its lights on or off depending on whether the junction is connected. These are the functions that control that behaviour -

3.1.1 Initialise()

Every signal must have an **Initialise** function, which is called by the game code when the signal is first activated as the route loads. Here you setup any information the scripts need to know about this signal, and define any global constants and variables you want to use elsewhere in the script.

The shunt signal's *Initialise* function looks like this –

```
-- INITIALISE
function Initialise ()

        -- Signal Message constants
        RESET_SIGNAL_STATE      = 0
        JUNCTION_STATE_CHANGE   = 2
        SIGNAL_BLOCKED          = 12
        SIGNAL_CLEARED          = 13

        -- 2D Map State constants
        CLEAR                   = 0
        BLOCKED                         = 2

        -- Initialise global variables
```

```
	gConnectedLink		= -1
	gInitialised		= false

	-- Tells the game to do an update tick once the route finishes loading
	Call( "BeginUpdate" )
end
```

First we define several **constants**. These represent numbers that are used in messages sent between signal scripts and the game code. We could just use the numbers in our scripts, but it's better to give them an easy-to-remember name so that you can see at a glance what the script is doing at each stage. By convention, all of our constants have names that are written ENTIRELY_IN_CAPITALS.

Next we define two **global variables** - **gConnectedLink** (which will be used to keep track of whether the junction beyond this signal is connected) and **gInitialised** (which starts as *false* and is set to *true* once the route finishes loading). Our global variables generally have names beginning with a lower case "g" to indicate that they're "global" (ie, you can check and change their value from any function in the script, not just the one they're defined in).

Note that a global variable is only global to this particular signal – each signal has its own instance of the script running on it, so you can have lots of signals of the same type in a route, all running the same script, and they'll all remember their own individual state.

Finally we "**Call**" a function "**BeginUpdate**". *BeginUpdate* is not a LUA function defined in the signal script, it's part of the main game code. Basically "*Call*" sends a message to the game engine telling it to run a particular code function for us - in this case, one called "*BeginUpdate*", which tells the game to start running the *Update* function.

3.1.2 Update(*time*)

The **Update** function *is* a script function, so when we *called BeginUpdate* we were telling the game code to start running this script function. The reason we did this is because the game will now keep running the *Update* script function every frame until we tell it to stop. The "*time*" parameter is the game's way of telling the script how much time has passed since the game last did an *Update*. This is useful for a variety of things, such as making lights flash and controlling animations and sound effects.

In this case though we've got a very simple *Update* function –

```
-- UPDATE
function Update (time)

	gInitialised = true

	-- Check state of junction
	OnJunctionStateChange( 0, "", 1, 0 )

	Call( "EndUpdate" )
end
```

The game will only start running the *Update* script function once the route has pretty much finished loading, by which time the state of all the junctions is known by the game. As

signals can span several junctions, we wait until they all know which way they're set before calling the **OnJunctionStateChange** script function to check if we're connected. Otherwise as the route loads the signal would keep checking whether it's connected every time a junction between the signal's two links initialises.

Having set *gInitialised* to true (so we know the route has finished loading) and checked the state of the junction, we *Call* the code function "**EndUpdate**". As the name suggests, this tells the game code to stop triggering the *Update* script function every frame. So the game will run *Update* once to check whether the junction is connected, and then stop.

Obviously, any signal that *calls BeginUpdate* from anywhere in its script requires an *Update* script function for the code to run, or an error will be generated and the signal won't work properly.

3.1.3 OnJunctionStateChange(*junction_state, parameter, direction, linkIndex*)

As noted above, this function checks whether the line ahead of us is connected. Here's how -

```
-- JUNCTION STATE CHANGE
-- Called when a junction is changed. Tests if the links are connected.
--
function OnJunctionStateChange(junction_state, parameter, direction, linkIndex)

	-- Check to see if the track is still connected
	gConnectedLink = Call( "GetConnectedLink", "", 1, 0 )

	if (gConnectedLink == -1) then
		Call ("Set2DMapSignalState", BLOCKED)
		Call( "ActivateNode", "mod_trk_shunt_lightsOn", 0 )
	elseif (gConnectedLink == 1) then
		Call ("Set2DMapSignalState", CLEAR)
		Call( "ActivateNode", "mod_trk_shunt_lightsOn", 1 )
	end
end
```

The words in brackets after "*OnJunctionStateChange*" are the names of parameters that the function expects you to provide when you trigger it. You don't need to worry about any of them right now though.

As you can see, this script function is also very simple for the shunt exit signal. First we *call* a code function "**GetConnectedLink**" and assign the value that it returns to the global variable *gConnectedLink*, so that the script knows which link (if any) is connected.

Again, this function requires a number of parameters – these are the values separated by commas after it says "*GetConnectedLink*". The first parameter isn't used anymore, so we'll leave it blank (""). The second parameter (1) is the direction we want to check the line in. This will almost always be 1, which means we're looking forwards up the line beyond the signal. The last parameter (0) is the link we want to start searching from.

So this call of *GetConnectedLink* will start at the signal's link 0 and look up the line until it reaches a broken junction, the end of the line, or another link belonging to this signal. Then it lets us know which link (if any) is connected. As the shunt exit signal only has two links (0

and 1), the value it sends back to the script is either 1 (if all the junctions between the two links are set ready for us to drive past the signal) or -1 (if any of the junctions between the two links is set against us, blocking our path).

Once we know whether the junction is connected, we can set the signal to the appropriate state. We do this by *calling* two more code functions –

Set2DMapSignalState sets the colour that the signal appears as on the 2D map and driver's guide. **BLOCKED** will make it appear as red circle, **CLEAR** will make it turn green.

ActivateNode is used to switch a signal's lights on and off. The first parameter is the name of the node we want to activate / deactivate (in this case, "mod_trk_shunt_lightsOn"), which is set in the 3D model file for the signal. The second parameter is 0 (to switch the light off) or 1 (to switch it on).

3.1.4 OnSignalMessage(*message, parameter, direction, linkIndex*)

This is another function that every signal script must contain. Signal messages are sent up and down the track by trains, junctions and other signals to give a signal information about the state of the route around it. Whenever one of those messages hits a signal's link, the game engine triggers the *OnSignalMessage* function for that signal.

OnSignalMessage's parameters let the game code provide the script with information about the message - what kind of *message* it is, whether there's any other information attached to it (*parameter*), which *direction* the message was travelling in when it hit our link (1 if it came from in front of us, -1 if it came from behind us), and the *linkIndex* of the link it was received by (in the case of a shunt exit signal, that will be either 0 or 1). Remember that arrow on the links of the signals you placed earlier? They were showing you which direction the link was facing in. If a message arrives at the side with the arrow on it, it's coming from in front of the link.

There are only two messages a shunt exit signal cares about – **RESET_SIGNAL_STATE** and **JUNCTION_STATE_CHANGE**. All other messages should be forwarded on in the direction they were travelling in, in case another signal further down the line is interested in them. The script to handle that looks like this –

```
-- ON SIGNAL MESSAGE
-- Called when a message is received by one of this signal's links
--
function OnSignalMessage( message, parameter, direction, linkIndex )

        -- This message is to reset the signals after a scenario / route is reset
    -- DO NOT FORWARD IT!
        if (message == RESET_SIGNAL_STATE) then

                -- Re-initialise the signal
                Initialise()

        -- This message lets us know that a junction has just been switched
        elseif (message == JUNCTION_STATE_CHANGE) then

                -- Only act on the message if it arrived at link 0
```

```
        -- and the parameter is "0" (junction has finished switching)
            if linkIndex == 0 and parameter == "0" then

                -- Trigger the OnJunctionStateChange script function
                OnJunctionStateChange( 0, "", 1, 0 )

            -- Pass on the message in case more signals protect that junction
                Call( "SendSignalMessage", message, parameter, -direction, 1, 0 )
            end

        -- All other messages should just be forwarded
        else

            -- Forward it on in that direction from link 0
            Call( "SendSignalMessage", message, parameter, -direction, 1, linkIndex )
        end
end
```

Again, this is fairly simple for a shunt signal.

If the message is *RESET_SIGNAL_STATE*, the game is letting us know that we need to reset the signal, as the player has just reloaded a saved position or returned to the game from the scenario editor. Luckily we can do this by just calling the *Initialise* script function again. Note that this message is sent directly to every signal in the route by the game engine, so there's no need to forward it on.

If the message is *JUNCTION_STATE_CHANGE*, we first check that the message arrived on link 0. As this type of message is sent by the junction itself when it switches, if it came from a junction that's between our two links it must arrive at link 0 from in front of us. If the message was received by link 1 instead, it must have come from another junction further up the line that we don't care about. We also ignore the message unless it came with its parameter value set to "0" (indicating that the junction has now finished switching).

Once we're sure the *JUNCTION_STATE_CHANGE* message is from a junction we're interested in, we trigger the *OnJunctionStateChange* script function that we explained earlier. Now the parameters that are passed to that function make more sense. "*junction_state*" should always be 0, as it's confirming that the junction has finishing switching - if it was still in motion, neither track would be connected to the junction and there wouldn't be any point in checking its state. "*parameter*" is no longer used and should always be left blank. "*direction*" is also obsolete and should be left as 1. Finally "*linkIndex*" is the link that received the junction state change message (and therefore the link that we need to look ahead of to see if the junction that just switched is connected). For any normal signal this will always be 0.

If the message was of any other type, our shunt signal isn't interested in it, so it just forwards it on from the link it arrived at. This is done by *calling* the **SendSignalMessage** code function. As you can see, it takes all the same parameters as *OnSignalMessage*, plus an extra one. The first is the type of *message*, the second is the *parameter* containing any additional information about the message, the third is the *direction* the message should be sent in (note that we reverse this value – if the message arrived from in front of us we want to send it backwards, which is direction -1, and vice versa), and the last parameter is the *linkIndex* to send the message from.

The fourth parameter (that extra 1 between *direction* and *linkIndex*) controls which signals can receive this message. This should almost always be 1, which means that only signal links which are pointing in the same direction as the one we're sending the message from will receive the message. If for some reason you wanted the message to be picked up only by signal links facing in the opposite direction, you would use -1 instead.

3.1.5 OnConsistPass(*prevFrontDist, prevBackDist, frontDist, backDist, linkIndex*)

The last function that every script must have is *OnConsistPass*. "Consist" is just a technical term for a train so, as the name suggests, this script function is triggered by the game code whenever a train drives past a link belonging to the signal.

The good news is that our shunt signal doesn't care about trains, it only cares about whether or not the line ahead of it is connected. So although we do need an *OnConsistPass* function in our script, in this case it doesn't actually do anything -

```
-- ON CONSIST PASS
-- Shunt signals do not care about consist passes
--
function OnConsistPass( prevFrontDist, prevBackDist, frontDist, backDist, linkIndex )
        -- do nothing on consist pass
end
```

And that's the lot. The shunt exit signal's script just consists of a single .lua file with all of those function definitions in it. If you look at the actual "UK ShuntSig Exit Head.lua" script file used by the real shunt signal (you'll find it in RailNetwork\Signals\UK Colour Light) you'll see that it's *slightly* more complex than we've shown here. We'll explain some of that later on, but for now don't panic - the script functions above are all you need to make a fully functional shunt signal.

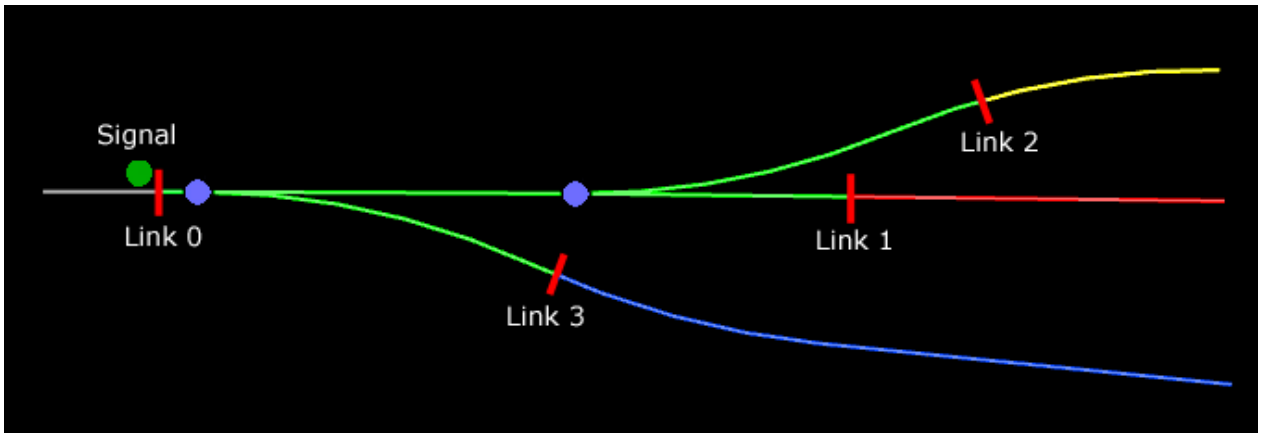
4 Signal Scripting Essentials

The shunt exit signal we just looked at is the simplest signal script used in the game. In this section we'll be introducing you to some of the additional scripting that's required to get other types of signal working.

4.1 Track Occupancy

All a shunt signal cares about is which way the junction they span is set. This is fine when you're just shunting wagons around a yard at low speeds, but when you're driving up a mainline you also need to know whether there's another train ahead of you. This is handled in Rail Simulator by "**occupation tables**".

Each "home" signal has its own occupation table, which is the script's way of remembering how many trains are in each part of the track covered by that signal (known as a "signal block"). This data is stored by the signal scripts in a table called "**gOccupationTable**". Each section of the track covered by this signal has its own entry in *gOccupationTable*.



For example, in the situation shown above, the number of trains on the track marked in green between link 0 and the forward links (1, 2 and 3) is stored in *gOccupationTable*[0], *gOccupationTable*[1] stores the number of trains on the track marked in red beyond link 1, *gOccupationTable*[2] covers the yellow track beyond link 2, and *gOccupationTable*[3] covers the blue track beyond link 3.

Normally on a real railway there would be another home signal further up each of those three lines. In that case, the occupation table entry for each of this signal's links would cover the track between that link and the link 0 of the next signal up the line. Once a train passes the next signal up the line, it's no longer our responsibility.

As trains drive past links belonging to a signal, the signal's occupation table entries go up and down. For example, when a train starts driving forwards past a signal's link 0, its *gOccupationTable*[0] is set to 1. If the train is going straight ahead, *gOccupationTable*[1] is set to 1 when the train starts to pass link 1. When the train finishes passing link 1 there is no longer anybody in the first bit of track, so *gOccupationTable*[0] is set back to 0. Finally, when the train finishes passing link 0 of the next signal up the line beyond link 1, that signal sends back a message to let us know it's gone, and *gOccupationTable*[1] is set to 0 again.

The reality is a little more complex than that, as you'll see from the script examples below, but this is basically how home signals know whether or not they have a train blocking the line ahead of them.

4.1.1 OnConsistPass(*prevFrontDist, prevBackDist, frontDist, backDist, linkIndex*)

Every signal script must have an *OnConsistPass* function, which is triggered by the game code when a train drives past one of the signal's links. In the case of the shunt signal, this function didn't actually do anything. But for a home signal it's our main way of keeping track of how many trains are in our block.

The parameters that the code sends to a script when it triggers the *OnConsistPass* function tell us exactly what's going on. *prevFrontDist* and *prevBackDist* tell us how far the front and back of the train were from the signal link last time it was checked. *frontDist* and *backDist* are the current positions. In all cases, a positive distance means it's behind the link and a negative distance means it's beyond the link. Finally, *linkIndex* is the number of the link that's being crossed.

From this information we can tell whether the train has just started or finished crossing the link, and which direction it's travelling in. This can be broken down as following -

If the train just started crossing a link...
    If it's going forwards...
        If it's passing link 0, link 0 is now blocked and the signal needs to turn red
        If it's passing any other link, that link is now blocked but the signal is already red
    If it's going backwards...
        If it's passing link 0, send a message to let the signal behind us know it's coming
        If it's passing the connected link, it's now blocking link 0

If the train just finished crossing a link...
    If it's going forwards...
        If it's passing link 0, send a message to let the signal behind us know it's cleared
        If it's passing the connected link, link 0 was blocked before but is now clear
    If it's going backwards...
        If it's passing link 0, link 0 is now cleared and the signal may need to turn green
        If it's passing any other link, that link is now clear

Here's what that looks like in script, with plenty of comments to explain each step -

```
-- ON CONSIST PASS
-- Called when a train passes one of the signal's links
--
function OnConsistPass ( prevFrontDist, prevBackDist, frontDist, backDist, linkIndex )

    -- Declare local variables that we'll use to remember what the train's doing
    local crossingStart = 0
    local crossingEnd = 0
```

```
-- If the front of the train is before the link and the back is beyond it,
-- or vice versa, the train is in the process of crossing the link
if ( frontDist > 0 and backDist < 0 ) or
( frontDist < 0 and backDist > 0 ) then

        -- If the front and back of the train were previously both beyond or
        -- both before the link, the train has just started crossing the link
        if ( prevFrontDist < 0 and prevBackDist < 0 ) or
    ( prevFrontDist > 0 and prevBackDist > 0 ) then

        -- Set crossingStart to 1 so we know we've just started crossing
                crossingStart = 1
        end

-- Otherwise the front and back of the train are both on the same side of
-- the link now, in which case it's already passed the link
else

        -- If the train previously had its front and back on opposite sides of
        -- the link, it's just finished crossing the link
        if ( prevFrontDist < 0 and prevBackDist > 0 ) or
    ( prevFrontDist > 0 and prevBackDist < 0 ) then

        -- Set crossingEnd to 1 so we know we've just finished crossing
                crossingEnd = 1
        end
end

-- If the train just started crossing the link...
if (crossingStart == 1) then

        -- If the train was behind the link before, it's crossing it forwards
        if (prevFrontDist > 0 and prevBackDist > 0) then

                -- If the train just started crossing link 0 forwards...
                if (linkIndex == 0) then

                        -- Let the signal know its block is now occupied
                        Occupied( )

                        -- And increment the number of trains blocking link 0
                        gOccupationTable[0] = gOccupationTable[0] + 1

                -- If the train just started crossing another link forwards...
                elseif (linkIndex > 0) then

                        -- Increment the number of trains on the track beyond that link
                        gOccupationTable[linkIndex] = gOccupationTable[linkIndex] + 1
                end

        -- If the train was beyond the link before, it's crossing it backwards
        elseif (prevFrontDist < 0 and prevBackDist < 0) then
```

```
                -- If the train just started crossing link 0 backwards...
                if (linkIndex == 0) then

                        -- Send a message back down the line to let the signal behind us know
                        -- that a train has just entered its block
                        Call( "SendSignalMessage", OCCUPATION_INCREMENT, "", -1, 1, 0 )

                -- If the train just started crossing another link backwards...
                elseif (linkIndex > 0) then

                        -- And that link is connected to link 0...
                        if (gConnectedLink == linkIndex) then

                                -- Increment the number of trains blocking link 0
                                gOccupationTable[0] = gOccupationTable[0] + 1

                        -- If the link the train is crossing isn't connected to link 0...
                        else
                   -- The train isn't going to pass link 0, so do nothing
                        end
                end
        end

-- If the train just finished crossing the link...
elseif (crossingEnd == 1) then

        -- If the train is before the link now, it just crossed it backwards
        if (frontDist > 0 and backDist > 0) then

                -- If the train just finished crossing link 0 backwards...
                if (linkIndex == 0) then

                        -- Decrement the number of trains blocking link 0
                        gOccupationTable[0] = gOccupationTable[0] - 1

              -- The signal's block might not be occupied now
                        NotOccupied( 0 )

                -- If the train just finished crossing another link backwards...
                elseif (linkIndex > 0) then

                        -- Decrement the number of trains on the track beyond that link
                        gOccupationTable[linkIndex] = gOccupationTable[linkIndex] - 1
                end

        -- If the train is beyond the link now, it just crossed it forwards
        elseif (frontDist < 0 and backDist < 0) then

                -- If the train just finished crossing link 0 forwards...
                if (linkIndex == 0) then
```

```
                                -- Send a message back down the line to let the signal behind us know
                                -- that a train has just left its block
                                Call( "SendSignalMessage", OCCUPATION_DECREMENT, "", -1, 1, 0 )

                        -- If the train just finished crossing another link forwards...
                        elseif (linkIndex > 0) then

                                -- If this link is connected to link 0...
                                if (gConnectedLink == linkIndex) then

                                        -- Decrement the number of trains blocking link 0
                                        gOccupationTable[0] = gOccupationTable[0] – 1

                    -- If this link isn't connected to link 0...
                                else
                                        -- The train didn't pass link 0 to get here, so do nothing
                                end
                        end
                end
        end
end
```

If you look at a real signal script, you'll see that it's slightly more complex. For example, we often don't put links on tracks that are only used by trains going in the opposite direction to the one the signal is facing in. If a train comes from that bit of track, the signal will be red anyway because none of its links are connected to link 0, but the signal won't know that a train is approaching it until it passes the signal's link 0. So the real signal scripts include a check to make sure that *gOccupationTable*[0] is greater than 0 before trying to decrement its value. Also, most signals can also show "warning" aspects which indicate that although their block is clear, the next signal up the line isn't. We'll come to that later though...

4.1.2 Occupation Increment / Decrement Messages

In the meantime, let's consider the two new signal messages we just used in that last script function – **OCCUPATION_INCREMENT** and **OCCUPATION_DECREMENT**. These are sent back by a signal when a train passes its link 0, to let the signal behind it know that the train has just entered / left its signal block. The message will always arrive on the link that the train it's warning about is approaching / leaving.

As with the junction state change message we looked at earlier, this message triggers the *OnSignalMessage* script function when the message reaches a signal's link.

Here's the extra scripting that needs adding to that function to handle the new messages –

```
        -- This message lets us know that a train has just passed the next signal up the
        -- line and is no longer in our signal block
        elseif (message == OCCUPATION_DECREMENT) then

                -- Decrement the occupation table for the link the message arrived on
                gOccupationTable[linkIndex] = gOccupationTable[linkIndex] - 1

                -- The signal's block might not be occupied now
```

```
        NotOccupied( linkIndex )

    -- This message lets us know that a train has just passed the next signal up the
    -- line backwards and is now entering our signal block
    elseif (message == OCCUPATION_INCREMENT) then

        -- Increment the occupation table for the link the message arrived on
        gOccupationTable[linkIndex] = gOccupationTable[linkIndex] + 1

        -- If this is the connected link, the signal's block is now occupied
        if (gConnectedLink == linkIndex) then
            Occupied( )
        end
```

4.2 Signal State

Now that we know if there's a train blocking the track ahead and which way any junction(s) between us and the next signal up the line are set, we can work out the state of a simple two aspect (stop or go) home signal...

4.2.1 Occupied() / NotOccupied(*linkIndex*)

You've no doubt noticed that we've just sneaked in two new functions called **Occupied** and **NotOccupied**. These are called from *OnConsistPass* and *OnSignalMessage* whenever a train enters or leaves a signal's block. Here's what those functions look like for a simple signal –

```
-- NOTOCCUPIED
-- Tells the signal to display a clear track ahead
--
function NotOccupied( linkIndex )

    -- If link 0 has been unoccupied, find the connected link
    if (linkIndex == 0 and gConnectedLink > 0) then
        linkIndex = gConnectedLink
    end

    -- If link 0 or the connected link has been unoccupied...
    if (gConnectedLink == linkIndex) then

        -- And the track ahead is clear...
        if (gOccupationTable[0] == 0) and
      (gOccupationTable[linkIndex] == 0) then

            -- And we weren't already set as clear...
            if gSignalState ~= SIGNAL_CLEARED then

                -- Set the signal state to clear
                SetState( SIGNAL_CLEARED )
            end
        end
    end
end
```

```
-- OCCUPIED
-- Tells the signal to display a blocked track ahead
--
function Occupied()

	-- If we weren't already blocked...
	if gSignalState ~= SIGNAL_BLOCKED then

		-- Set the signal state to blocked
		SetState( SIGNAL_CLEARED )
	end
end
```

These functions are just a way of keeping your script tidy, and making it easy to maintain and update – instead of repeating the scripting that checks for a signal being blocked or cleared several times in different functions, you put it all in a pair of simple subfunctions which you can then call from anywhere else in the script.

We've also introduced a new global variable called **gSignalState**. As the name suggests, this keeps track of the current state of the signal. In the examples we've used so far, that's either clear or blocked. Doing this means that we don't waste time trying to change a signal to a state it's already in.

4.2.2 SetState(*newState*)

We've also introduced another new function – **SetState**. Again, this just keeps commonly used functionality wrapped up in a simple subfunction which you can call from anywhere in your script. Here's what that function looks like for a simple 2 aspect signal head -

```
-- SET STATE
-- Sets the current state of the signal
--
function SetState( newState )

	-- If the signal is cleared now...
	if (newState == SIGNAL_CLEARED) then

		-- Set this signal to green on the 2D map
		Call ("Set2DMapSignalState", CLEAR)

		-- Turn on the green light and turn off the red light
		Call ( "ActivateNode", "mod_hd2_green", 1 )
		Call ( "ActivateNode", "mod_hd2_red", 0 )

	-- If the signal is blocked now...
	elseif (newState == SIGNAL_BLOCKED) then

		-- Set this signal to red on the 2D map
		Call ("Set2DMapSignalState", BLOCKED)

		-- Turn on the red light and turn off the green light
		Call ( "ActivateNode", "mod_hd2_green", 0 )
```

```
            Call ( "ActivateNode", "mod_hd2_red", 1 )
        end

        -- Remember the signal state
        gSignalState = newState
end
```

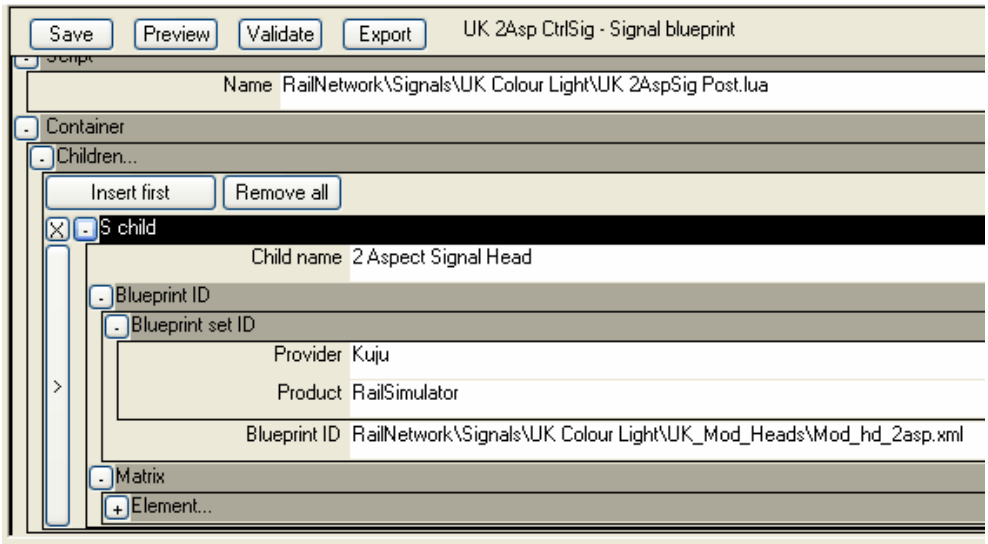
This should be familiar from the shunt signal script we looked at earlier. In that case we were switching a light on or off directly from the *OnJunctionStateChange* function, based on whether or not the track ahead was connected. Here we have two lights (red and green) which we toggle on and off based on whether the track ahead is clear or blocked. It's the same basic functionality, we've just split it off into its own subfunction for convenience.

Also, after switching the lights and 2D map state of the signal, we set *gSignalState* to its new value. Again, we could have put that line in the individual functions that call *SetState*, but as we always want to remember the state of the signal it makes more sense to include that line here.

Note that different types of signal have different names for their light nodes. These are set in the 3D model itself, so if you're not sure what a signal's nodes are called you'll need to ask the artist (if it's a new custom-made 3D model) or check an existing script used by a signal that has the same signal head as the one you're using (if it's a model that shipped with Rail Simulator) to find out what name to use.

Another important thing to bear in mind is that not every signal is just a free floating head. Some signal blueprints have their head ready mounted on a post. In this case, the signal head is a "child" of the post it's mounted on, and you'll need to let the *ActivateNode* function know the name of the "child" object you're trying to activate a light on.

To find out what the child's name is, open the signal blueprint in the Asset Editor and scroll to the bottom of the information displayed there. You should see an entry marked "Children". Click on the + next to it to expand the view of that section, and then click on the + next to the child entry stored inside it to see the child's information.



At the top of the child's settings is an entry called "child name". This is the name that you need to refer to in your script when you switch one of this head's lights on or off. For example, the post-mounted version of the 2 aspect signal we were looking at before has its signal head as a child called "2 Aspect Signal Head". To change lights on the signal, we would use the following lines –

```
-- Turn on the red light and turn off the green light
Call ( "2 Aspect Signal Head:ActivateNode", "mod_hd2_green", 0 )
Call ( "2 Aspect Signal Head:ActivateNode", "mod_hd2_red", 1 )
```

As you can see, all we've done is add the child's name and a colon before *ActivateNode*. This tells the game code that we want it to run the *ActivateNode* function on the part of the signal called "2 Aspect Signal Head". And that's all there is to it...

4.2.3 GetSignalState()

There's one last script function which most home signals require - **GetSignalState**. All of the home signals on the Oxford - Paddington and Newcastle - York routes have this function, and if you're using a warning system such as AWS or TPWS (more on those later) on your route, your signals will probably need this function too.

Luckily it's very simple - all it does is return the current state of the signal when it's triggered by the code. As we're keeping track of the signal's state as a global variable *gSignalState* anyway, all we need to do is turn that information into a form that the code will understand. Here's how we do that -

```
-- GET SIGNAL STATE
-- Gets the current state of the signal - blocked, warning or clear.
--
function GetSignalState( )

	-- Initialise signal state to -1 (to signify an error)
	local signalState = -1

	-- Set signalState based on the current state of the signal
	if (gSignalState == SIGNAL_CLEARED) then
		signalState = CLEAR

	elseif (gSignalState == SIGNAL_WARNING) or
		(gSignalState == SIGNAL_WARNING2) then
		signalState = WARNING

	elseif (gSignalState == SIGNAL_BLOCKED) then
		signalState = BLOCKED
	end

	return signalState
end
```

As you can see, this sends the code the same constants (*CLEAR*, *WARNING* and *BLOCKED*) that the *Set2DMapSignalState* function uses. If you're wondering what **SIGNAL_WARNING** and **SIGNAL_WARNING2** are, read on...

4.3 Route State

Using the functions we've shown you so far, we can determine which of the signal's links is connected to its link 0 (using *OnJunctionStateChange*) and whether there are any trains blocking the track ahead (using the occupancy table generated by *OnConsistPass* and the *OCCUPATION_INCREMENT* and *DECREMENT* signal messages). Based on that, we can switch the lights on a signal to red (stop) or green (go).

This is fine for a simple two aspect signal, but most signals in Rail Simulator can show one or more "warning" aspects as well, indicating the state of the next signal up the line. To handle this we'll need a new global variable to remember the state of the signal(s) ahead of us, and a new set of signal messages to pass on that information. We also need to adjust the functions we've already looked at already to take account of this new data.

4.3.1 gLinkState

We're going to store this information in a table called **gLinkState**. Each of our signal's links will have an entry in the table storing the state of the next signal up the line in front of it. That way if a junction our signal is covering switches, we'll always know the state of the next signal beyond the link that's now connected, and can show the appropriate aspect.

Like any variable, gLinkState needs to be initialised. In this case we're going to create one entry for each link and default all their values to *SIGNAL_CLEARED*.

```
-- Find out how many links this signal has
gLinkCount = Call( "GetLinkCount" )

-- Create an empty table called gLinkState
gLinkState = {}

-- Cycle through all this signal's links
for link = 0, gLinkCount - 1 do
	-- Set the default state for that link to CLEARED
	gLinkState[link] = SIGNAL_CLEARED
end
```

Notice that we've *called* another new code function here – **GetLinkCount**. As the name suggests, this counts how many links the signal has. As the signal's links are numbered starting at 0, the number of the signal's last link is one less than the link count – eg, a signal with 3 links has links 0, 1 and 2. This information can be useful elsewhere, so we're remembering that number as a global variable called **gLinkCount**.

Once we know how many links the signal has, we create an empty table called *gLinkState* and then cycle through all the signals links, creating an entry for each of them in that table and setting its default value to *SIGNAL_CLEARED*.

4.3.2 Signal State Messages

As each signal is controlled by its own independent script, we need to send messages between them to keep track of the state of any other signals ahead of us. Whenever a signal's state changes from blocked to clear or vice versa, it sends back a message to let any signals behind it know its new state.

Here's the extra scripting that we need to add to the *OnSignalMessage* function to handle these messages –

```
	-- If the signal ahead of us is showing clear or warning, we're clear
	elseif ( message == SIGNAL_CLEARED or message == SIGNAL_WARNING ) then
		NotOccupied( linkIndex )

	-- If the signal ahead of us is showing blocked, we should be at warning
	elseif ( message == SIGNAL_BLOCKED ) then
		Warning( linkIndex )
```

As you can see, we're using a new script function called **Warning** to set the signal to its warning state when we're told the next signal up the line is blocked.

```
-- WARNING
-- Tells the signal to display that there is a warning ahead
--
function Warning( linkIndex )

	-- If the message arrived on the connected link...
	if (gConnectedLink == linkIndex) then

		-- And there aren't any trains between us and the next signal...
		if (gOccupationTable[0] == 0) and
	   (gOccupationTable[linkIndex] == 0) then

			-- If the signal wasn't already set to WARNING...
			if gSignalState ~= SIGNAL_WARNING then

				-- Set the signal state to WARNING
				SetState( SIGNAL_WARNING )

				-- Send back a message to let signals behind us know we're at warning
				Call( "SendSignalMessage", SIGNAL_WARNING, "", -1, 1, 0 )
			end
		end
	end

	-- Remember that this link is at WARNING
	gLinkState[linkIndex] = SIGNAL_WARNING
end
```

This is just like the *NotOccupied* function we showed you earlier, except that it uses *SIGNAL_WARNING* instead of *SIGNAL_CLEARED*, and we've added a couple of new bits of functionality.

First, we're now sending a message back to let the signal behind us know what state we're in if our state has just changed. The same thing should be done in the *Occupied* and *NotOccupied* functions, but using *SIGNAL_BLOCKED* / *SIGNAL_CLEARED* as appropriate.

Also, at the end of the function we've added a new line to store the state of the line ahead of us on this link. In this case we're remembering that if this link is connected we should be at warning, because the next signal is blocked. The *NotOccupied* function should behave the same way, setting *gLinkState*[*linkIndex*] to *SIGNAL_CLEARED*.

However, the *Occupied* function should NOT set the link state to *SIGNAL_BLOCKED* – the *Occupied* function is only called if the track directly ahead of the signal is blocked by a train or a broken junction, and *gLinkState* is only interested in what the next signal up the line is showing, not what this signal is showing.

4.3.3 OnJunctionStateChange

The whole point of remembering the state of the next signal in front of each of our links is so that when a junction between us and that next signal switches, we know what we should be showing. This requires some changes to our *OnJunctionStateChange* function. Also, we need to take account of the fact that our signal might have more (or less) than two links, in which case we need to know not only whether or not the line ahead is connected, but which of the possible lines our signal's link 0 is connected to.

Here's an example of a generic junction state change function that handles this –

```
-- JUNCTION STATE CHANGE
-- Called when a junction is changed. Tests if the links are connected.
--
function OnJunctionStateChange(junction_state, parameter, direction, linkIndex)

        -- Only need to check which link is connected if we have more than one link!
    if gLinkCount > 1

            -- Find the link that is now connected to the signal
            local newConnectedLink = Call( "GetConnectedLink", "" , 1, 0 )

                -- Only continue if the link we're connected to has changed
                if newConnectedLink ~= gConnectedLink then

                    -- Remember which link we're now connected to
                    gConnectedLink = newConnectedLink

                    -- If we're connected to a valid link...

                    if gConnectedLink > 0 then

                        -- If there aren't any trains on the connected path
                        if (gOccupationTable[0] == 0) and
                (gOccupationTable[gConnectedLink] == 0) then

                    -- If the connected link is clear...
                                if gLinkState[gConnectedLink] == SIGNAL_CLEARED
then

                                    -- Set the signal as NotOccupied
                                    NotOccupied(gConnectedLink)
```

```
                                        -- If the connected link is at warning...
                                        elseif gLinkState[gConnectedLink] ==
SIGNAL_WARNING then

                                                -- Set the signal to Warning
                                                Warning(gConnectedLink)
                                        end

                                -- If a train is blocking the connected path
                                else
                                        -- Set the signal as Occupied
                                        Occupied()

                        -- If the track ahead is blocked by a junction set against us...
                        elseif gConnectedLink == -1 then

                                -- Set the signal as Occupied
                                Occupied()
                        end
                end
        end
    end
end
```

As you can see, this is still fairly simple. All we're doing is checking which link is currently connected, and then (assuming a valid link is connected and there isn't a train blocking the path to the next signal) checking the state of that link before deciding how to set the signal.

4.3.4 Signal Message Directionality

Before we go any further, it's worth mentioning the directionality of signal messages. As we've already seen, unless we specify otherwise, messages are only picked up by signals whose links are facing in the same direction as the link that the message was sent from. But it's also important to take account of the direction we send the message in from that link.

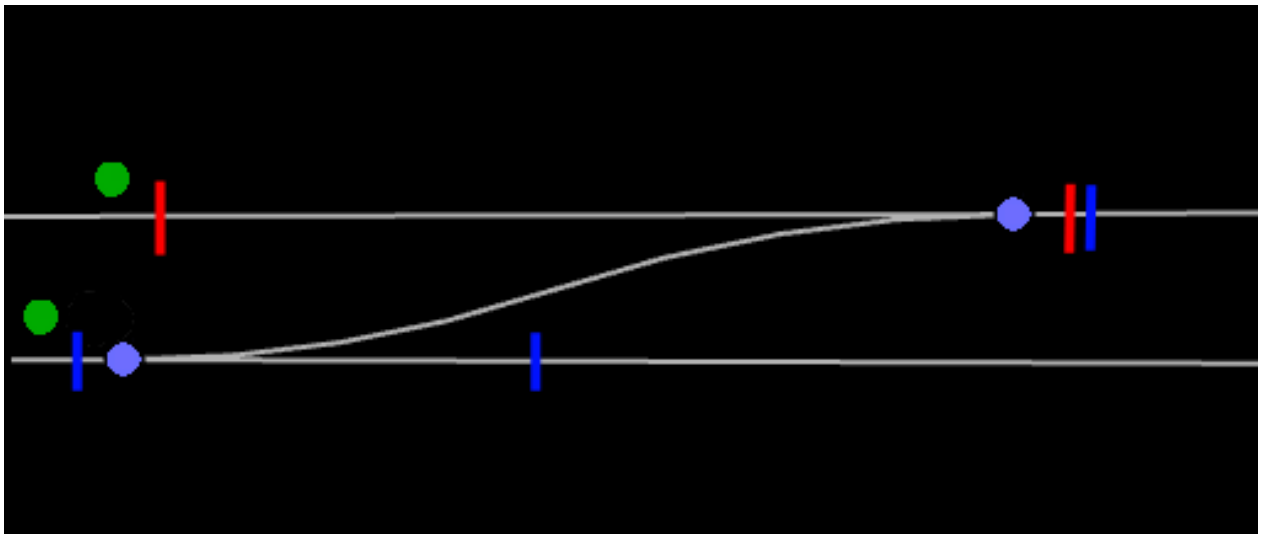
Most messages are sent backwards (direction -1) to let the signal behind us know what state we're in. When you send a message backwards, it should normally be sent from link 0. This ensures that it doesn't get intercepted by any other links belonging to the signal that sent it, or other signals covering the same junction(s).

Messages that are sent forwards (direction 1) are occasionally used to tell the next signal up the line that there's a train approaching it. When you send a message forwards, it should normally be sent from the connected link. Again, this ensures that it doesn't get intercepted by any other links belonging to the same signal.

Sometimes it makes sense to break these rules, but normally they should be observed.

4.3.5 PASS_ Messages

The OnSignalMessage script functions we've shown you so far work fine for a simple signal setup, but in the real world things are rarely simple. For example, consider a crossover between two parallel tracks.



Both tracks have signals on them just before the crossover, and both of those signals have a link just beyond the converging junction at the top right of the diagram. But signal links consume the messages they receive, so if a message comes down the track from beyond that junction, it will only be received by the first signal link it reaches (the blue one, belonging to the signal on the lower track). For the other signal to receive any messages from further up that line, the blue link will need to forward them on to the red link.

The way we do this is using PASS_ messages. For every signal message in the game there is an equivalent PASS_ message. For example, PASS_SIGNAL_BLOCKED is the PASS_ version of SIGNAL_BLOCKED. When a signal receives SIGNAL_BLOCKED on any link other than link 0, it forwards on a PASS_SIGNAL_BLOCKED in the same direction from that link.

If PASS_SIGNAL_BLOCKED is received by another non-zero link (in this example, the red link just beyond the junction), that link must belong to another signal covering the same junction, and so it will react to the message and then forward it on too.

This continues until the PASS_ message reaches a link 0. When that happens it must have gone past the junction, and can now be safely ignored and consumed by that link.

NOTE: this is one situation where the normal message directionality rules that we mentioned earlier don't apply - in this case we *want* the message to be intercepted by all of the other signal links covering this junction, rather than skipping over the junction as normal, so we send the messages on from the link they arrived at.

The scripting that handles passing on messages looks like this –

```
-- ON SIGNAL MESSAGE
-- Called when a message is received by one of this signal's links
--
```

```
function OnSignalMessage( message, parameter, direction, linkIndex )

	-- Check for signal receiving a message it might need to forward on
	if (linkIndex > 0) then

		-- If we've received a PASS_ message
		if message > PASS_OFFSET then

			-- Forward it on from this link
			Call( "SendSignalMessage", message, parameter, -direction, 1, linkIndex )

		-- Don't pass on a reset signal or junction state change message!
	   elseif message ~= RESET_SIGNAL_STATE and
		  message ~= JUNCTION_STATE_CHANGE then

			-- Forward the message as a PASS_ message by adding PASS_OFFSET to it
			message = message + PASS_OFFSET
			Call( "SendSignalMessage", message, parameter, -direction, 1, linkIndex )
		end
	end

	-- always check for a valid link index
	if (linkIndex >= 0) then

		-- If the message is a PASS_ message...
		if message > PASS_OFFSET then

			-- Only pay attention to it if we're not the base link of the signal
			if linkIndex > 0 then

				-- Convert it back into a normal message and process it
				message = message – PASS_OFFSET
				ReactToSignalMessage( message, parameter, direction, linkIndex )
			end

		-- Otherwise, it's a normal signal so just process it as normal
		else
			ReactToSignalMessage( message, parameter, direction, linkIndex )

		end
	end
end
```

Notice that we've introduced a new constant called **PASS_OFFSET**. Every PASS_ message is defined as the original signal message constant plus *PASS_OFFSET*. That means that we can turn any signal message into a PASS_ message by adding *PASS_OFFSET*, and then convert it back into a normal signal message by subtracting *PASS_OFFSET*. And we also know that any message with a value greater than *PASS_OFFSET* must be a PASS_ message.

You'll also see that to keep things simple we've removed from this function all the scripting that reacts to the various signal messages, and created a new subfunction to handle that called **ReactToSignalMessage**. This is then called from *OnSignalMessage* when necessary.

As well as keeping the script tidy, this also means that the *OnSignalMessage* function is now generic enough that it can be used by pretty much any signal in the game without needing any modification – all the scripting which handles the specific messages that various signals need to react to is now held within the *ReactToSignalMessage* subfunction, and all that the *OnSignalMessage* function does is decide when to forward on a message and when to react to it. Why is this so useful? Read on...

4.4 Including Common Script Functions

As we've just seen, some of the LUA script functions that control a signal can be written in such a way that they're generic enough to work on a wide range of different signals. Rather than just copying and pasting these functions into every script that needs them, you can put them all into a separate .lua file (which we refer to as Common script files) and "include" that file in your individual signal scripts.

For example, the script for a UK 3 aspect signal post has the following lines at the top of it –

```
--include=CommonScripts\Common UK Colour Light Script.lua
--include=..\CommonScripts\Common Signal Script.lua
```

This just means that when the Asset Editor comes to build that script ready for use in the game, it includes the entire contents of those two Common .lua files at the bottom of the signal's own individual script. Which is incredibly useful, because if you have several variants of the same basic signal, you can create individual scripts for each of them that only have a handful of functions in them, and then put any functions that are the same for all of those signals into a Common file and *include* that in each of the individual scripts.

Now if you need to make a change to one of the core functions that's in the Common file, rather than having to edit every signal-specific script you can just edit that one Common script and then re-export all the signal-specific scripts from the Asset Editor to update them ready for use in the game.

It also means that if you're making a new signal which is similar to one that already exists in the game, you can probably use many of the generic functions that we've already scripted by including the appropriate Common files into your own script.

Most of our signal scripts work on three or four levels –

At the top is a signal-specific script which is tailored to a particular signal or range of signals (for example, standard UK 3 aspect signal posts with one or more links). This contains all of the core functions that every signal script requires, plus a few subfunctions that include information that's specific to that type of signal, such as *ReactToSignalMessage* and a function to switch the appropriate lights on and off depending on the state of the signal.

However, many of the functions within this script will look something like this –

```
-- ON CONSIST PASS
function OnConsistPass (prevFrontDist, prevBackDist, frontDist, backDist, linkIndex)

    -- Use the Default function for this
    DefaultOnConsistPass (prevFrontDist, prevBackDist, frontDist, backDist, linkIndex)
```

```
end
```

As this particular signal doesn't need to do anything unique when a train passes it, it can use a generic function called *DefaultOnConsistPass* to handle this event. These **Default** functions can be used by a range of similar signals – in this case, UK Colour Lights – and are kept in a separate script file called "Common UK Colour Light Script.lua", which is kept in a subfolder of the "UK Colour Lights" signal folder called "CommonScripts".

If you look at that script though, you'll see that some of the functions in there call a further level of **Base** functions. For example, *DefaultOnConsistPass* just calls *BaseOnConsistPass*. These Base functions are generic enough that they can often be used by signals from different routes and countries, and are kept in a script called "Common Signal Script.lua" in a "CommonScripts" folder in RailNetwork\Signals.

In some cases we have an extra layer between the signal-specific and Default functions. For example, modern British signals often have route indicators known as "feathers" that light up to show which route is connected. There are literally dozens of variations of these signals in the game, with different numbers of links, feathers and lights. Each one has its own unique script, but if you open up one of those scripts all you'll see is something like this –

```
--------------------------------------------------------------------------------
-- UK 4 Aspect Signal Post
-- KUJU / Rail Simulator
--------------------------------------------------------------------------------

--include=CommonScripts\Common UK 4 Aspect Feather Script.lua
--include=CommonScripts\Common UK Colour Light Script.lua
--include=..\CommonScripts\Common Signal Script.lua

--------------------------------------------------------------------------------
-- INITIALISE
--
function Initialise ()

	-- for POST signals, we manipulate the lights via a child node,
	-- and therefore we need to specify the child's name.
	gLightNodeName = "4 Aspect Signal Head:"

	DefaultInitialise( )
end

--------------------------------------------------------------------------------
-- ACTIVATE LINK
--
function ActivateLink( connectedLink )

	if (connectedLink == 1) then
		Call( "Route Indicator:ActivateNode", "feather_1", 0 )
		Call( "Route Indicator:ActivateNode", "feather_2", 0 )
	elseif (connectedLink == 2) then
		Call( "Route Indicator:ActivateNode", "feather_1", 1 )
		Call( "Route Indicator:ActivateNode", "feather_2", 0 )
```

```
	elseif (connectedLink == 3) then
		Call( "Route Indicator:ActivateNode", "feather_1", 0 )
		Call( "Route Indicator:ActivateNode", "feather_2", 1 )
	end
end
```

The script only contains an *Initialise* function and a subfunction called **ActivateLink** which lights up the appropriate "feather" depending on which link is connected. The rest of the functions required by this signal are kept in the "Common UK 4 Aspect Feather Script.lua" script, which (as the name suggests) is included in all of the scripts for UK 4 Aspect signals which have one or more feathers. If we'd put all those functions into each of the individual signal scripts, every time we wanted to make a change we would have had to edit dozens of scripts instead of just three (Common UK 2, 3 and 4 Aspect Feather scripts).

You may also have noticed that the *Initialise* function calls a *DefaultInitialise* function after it has set some information that's specific to this signal. This is another way in which common script functions are used. *Initialise* sets data specific to this particular signal type (how many arms a semaphore signal has), *DefaultInitialise* sets data specific to this group of signals (initialising the state of each of those arms), and *BaseInitialise* sets data that's common to all signal types (initialising the occupation table for the signal).

In the Common Signal Script you'll also find definitions of all the global constants and global variables we've referred to in these script functions – things like signal messages, 2D map states, *PASS_OFFSET* and *gConnectedLink*. Again, defining these basic constants in a single file that's shared by all our scripts means that if (for some reason) the value of one of those constants had to be changed, you would only need to edit one file and then re-export all of the signal scripts to update them.

4.4.1 SetLights(*newState*)

In many cases, functions we've already looked at can be made generic enough to be moved to common script files by splitting any signal-specific functionality off into a subfunction. For example, earlier we created a function called *SetState* that switched a signal's lights on and off based on it's new state.

As it stands, that function can only be used for UK 2 aspect signals. If you tried to run it on a UK 3 aspect signal, for example, none of the lights would ever change (3 aspect signals have different names for their light nodes), and the signal wouldn't know what to do when it was told to switch to "warning" (because 2 aspect signals don't have a warning state).

But we can easily turn *SetState* into a generic function by making a new subfunction called **SetLights** to switch the lights on and off. For example, here's the *SetLights* function for a UK 3 Aspect Signal Post –

```
-- SET LIGHTS
-- Called by SetState in Common UK Colour Light script to switch the appropriate
-- lights for this signal type on/off according to its new state

function SetLights ( newState )

	if (newState == SIGNAL_CLEARED) then
		Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_green", 1 )
```

```
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_orange", 0 )
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_red", 0 )
        elseif (newState == SIGNAL_BLOCKED) then
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_green", 0 )
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_orange", 0 )
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_red", 1 )
        elseif (newState == SIGNAL_WARNING) then
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_green", 0 )
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_orange", 1 )
                Call ( "3 Aspect Signal Head:ActivateNode", "mod_hd3_red", 0 )
        end
end
```

Our generic *SetState* would then look like this –

```
-- SET STATE
-- Sets the current state of the signal
--
function SetState( newState )

        -- If the signal is cleared now...
        if (newState == SIGNAL_CLEARED) then

                -- Set this signal to green
                Call ("Set2DMapSignalState", CLEAR)
                SetLights(SIGNAL_CLEARED)

    -- If the signal is blocked now...
        elseif (newState == SIGNAL_BLOCKED) then

                -- Set this signal to red
                Call ("Set2DMapSignalState", BLOCKED)
                SetLights(SIGNAL_BLOCKED)

    -- If the signal is at warning now...
        elseif (newState == SIGNAL_WARNING) then

                -- Set this signal to yellow
                Call ("Set2DMapSignalState", WARNING)
                SetLights(SIGNAL_WARNING)

    -- If the signal is at warning2 now...
        elseif (newState == SIGNAL_WARNING2) then

                -- Set this signal to double yellow
                Call ("Set2DMapSignalState", WARNING)
                SetLights(SIGNAL_WARNING2)
        end

        -- Remember the signal state
        gSignalState = newState
end
```

As you can see, there are no references to specific light nodes in this function anymore, and it can also handle the *SIGNAL_WARNING* and *SIGNAL_WARNING2* states now, which means that it can be used by 2, 3 *and* 4 aspect UK signals. As a result it's kept in the Common UK Colour Light Script file, and gets called by the *Occupied*, *NotOccupied*, *Warning* and *Warning2* functions of *all* modern UK signals (apart from shunt signals, which are so simple they don't really need any common functionality).

The same thing can be done for most functions, and doing this will help keep your scripts as concise, tidy and easy to edit as possible.

5 Advanced Signal Scripting

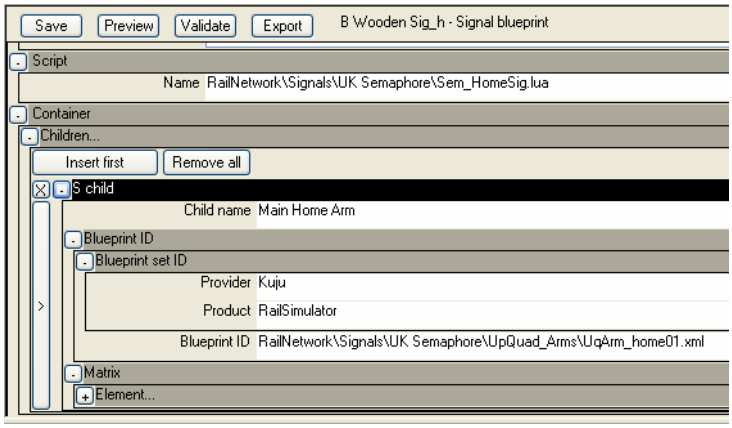
So far we've covered the basics of how to get a signal up and running. However, there are a few more features that we haven't looked at yet which are required by some signals...

5.1 Animation

Up to this point we've only looked at signals that have lights which are switched on and off using the *ActivateNode* code function. However, some signals have moving parts that are animated using the **AddTime** code function. For example, the old fashioned semaphore signals used on Rail Simulator's Bath to Templecombe route have one or more "arms" that tilt up and down to indicate whether the route ahead is clear or blocked.

Moving parts on a signal are usually separate "child" objects. Just as a set of lights mounted on a post is a child of that post, so the arms of a semaphore signal are separate models that are each defined in the signal's blueprint as a child of the post they're attached to.

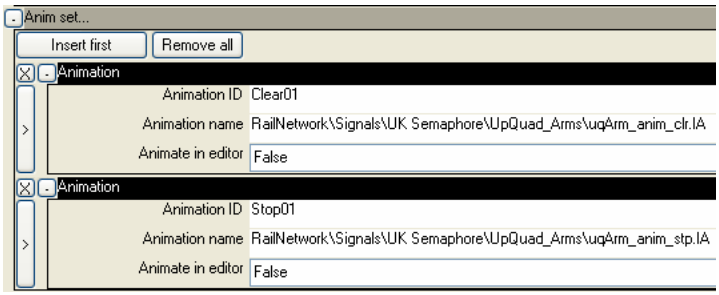
In the Asset Editor, open a semaphore signal (for example, "RailNetwork\Signals\UK Semaphore\Wooden_Posts\B Wooden Sig_h") and scroll to the bottom of it. Click on the + next to where it says "Children" to expand its list of child objects, and then click on the + next to "S child" to view the child's settings.



There are two important bits of information for us here. As we mentioned earlier, the child name ("Main Home Arm") is what we need to put in front of any code functions we want to run on that particular bit of the signal. So to animate this signal's arm, we'd *call* "*Main Home Arm:AddTime*".

The other thing we need to know is the blueprint ID for the signal's arm. Like any child object, the arm has its own blueprint which defines which model and (for moving objects like this one) which animations it uses. In this case it's called UpQuad_Arms\UqArm_home01.

If you open up that blueprint, you'll see an entry at the bottom saying "anim set". If you expand that you'll see that this signal arm has two animations. If you expand their entries their names are displayed – "Clear01" and "Stop01". Those are the names that we reference this signal's animations by in our scripts.



Animating an object is a bit fiddly compared to switching a light on or off. We can't just tell the signal to animate and then leave it to it – we need to move the arm through its animation one frame at a time from the script. To do that we use the *Update* function.

As we explained earlier, once activated by the *BeginUpdate* code function, this script function is triggered every frame and has a *time* parameter that tells us how long it is since the script last updated. So to animate a semaphore signal, we call *BeginUpdate* from the *SetState* function when the signal changes state. Then in the Update function itself, we check whether we're animating to clear or blocked and then *AddTime* to the appropriate animation. While the arm is animating, *AddTime* returns 0. As soon as it returns anything other than zero, we know the animation has finished, and can call *EndUpdate* to stop the Update function running.

So to run the "Clear01" animation on the signal arm described above, we would use the following bit of scripting in the signal's *Update* script function -

```
if Call( "Main Home Arm:AddTime", "Clear01", time) ~= 0 then
        Call("EndUpdate")
end
```

If we want to stop an animation in mid flow (if the signal changes state again before we finish animating, for example), we use the **Reset** code function, as follows -

```
Call( "Main Home Arm:Reset", "Stop01" )
```

Of course, in reality things are usually more complex than this. For example, UK semaphore signals sometimes have multiple arms, so we need to keep a global table containing the child name, animation names and current state of each arm. And because more than one of the arms could be animating at the same time, we also need to keep track of which and how many arms are animating, so that we know when to start and stop the *Update* function.

If you want to see how this all works, look at the file "Common UK Semaphore Script.lua" in RailNetwork\Signals\UK Semaphore\CommonScripts.

The *DefaultInitialise* function initialises the tables that will contain information about each of the signal's arms, their state, and which of them (if any) are currently animating, as well as some global constants that we use for the various animation states the signal can be in, and to make it clear which arm each entry in the table refers to.

The *SetState* function changes the animation state of an arm to open, closed, opening or closing, keeps track of which arms are currently animating, and starts and stops the *Update* function at the appropriate time.

The *DefaultUpdate* function cycles through all the arms on the signal, checks which of them are animating, and updates the animation as shown above. When an arm's animation ends, *DefaultUpdate* calls *SetState* again to change its animation state to open or closed.

The script "Sem_HomeSig.lua" in RailNetwork\Signals\UK Semaphore is a simple example of a semaphore signal with just one arm. If you look at its *Initialise* function, you can see that it sets the child and animation names of this particular signal, filling in the blanks in the table that was setup by *DefaultInitialise*.

Again, this all means that our scripts can use a lot of generic functions that will work across multiple signals, with information about the specific signal the script is running on stored in global variables and tables.

5.2 Flashing Lights

Another use for the *Update* function is making a signal's lights flash on and off. For example, say we want our red light (let's say its node name is "Red01") to flash on and off every half second. First we would declare some constants to set how long the light is switched on and off for during each cycle, and global variables to keep track of the current state of the light -

```
-- How long to stay off/on in each flash cycle
LIGHT_FLASH_OFF_SECS		= 0.5
LIGHT_FLASH_ON_SECS		= 0.5

-- State of flashing light
gTimeSinceLastFlash		= 0
gLightFlashOn			= false
gFirstLightFlash		= true
```

Whenever we want the red light to start flashing, we just set *gFirstLightFlash* to true (so we know we're just starting to flash) and *call BeginUpdate* to start our *Update* script function running each frame. In that, we put the following scripting -

```
-- UPDATE
-- Makes lights flash
--
function Update( time )

	-- If this is our first flash, reset time since last flash and light flash on
	if gFirstLightFlash then

		-- Reset flash state
		gTimeSinceLastFlash = 0
		gFirstLightFlash = false
		gLightFlashOn = false

	-- Otherwise...
	else
		-- Increment the timer by however much time has passed since our last update
		gTimeSinceLastFlash = gTimeSinceLastFlash + time

		-- If we're off and we've been off long enough, switch on
		if (not gLightFlashOn) and gTimeSinceLastFlash >= LIGHT_FLASH_OFF_SECS then
			Call ( "ActivateNode", "Red01", 1 )
			gLightFlashOn = true
			gTimeSinceLastFlash = 0

		-- If we're on and we've been on long enough, switch off
		elseif gLightFlashOn and gTimeSinceLastFlash >= LIGHT_FLASH_ON_SECS then
			Call ( "ActivateNode", "Red01", 0 )
			gLightFlashOn = false
			gTimeSinceLastFlash = 0
		end
	end
```

```
end
```

When we want the light to stop flashing again, we simply *call EndUpdate* and then switch the red light on / off based on our new state.

5.3 Train Warning Systems

All the European routes modelled in Rail Simulator (apart from the old Bath - Templecombe route, for obvious reasons) have some form of train warning system that sounds an alarm in the train cab and/or applies the emergency brakes if you drive past a signal without acknowledging it, exceed the track speed limit or pass a red signal. And all of the routes' signals also include a check for "SPADs" - Signals Passed At Danger.

Here's how these systems work...

5.3.1 SPADs

SPAD messages are very easy to generate, and don't require any additional objects to be added to your track. All you need to do is add a few lines to the *OnConsistPass* function of your signal. Here's the relevant bit of the standard European *BaseOnConsistPass* function -

```
-- If the train just started crossing link 0 forwards...
if (linkIndex == 0) then

    -- Check for SPADs
    if (gSignalState == SIGNAL_BLOCKED) then
        -- If we passed a blocked signal, send a consist message
        Call( "SendConsistMessage", SPAD_MESSAGE, "" )
    end

    -- Then let the signal know its block is now occupied
    Occupied( )

    -- And increment the number of trains blocking link 0
    gOccupationTable[0] = gOccupationTable[0] + 1
```

As you can see, all we're doing is checking if the signal is blocked when a train starts to pass link 0. If it is, a SPAD is generated by *calling* a code function called **SendConsistMessage** which (as the name suggests) sends a message (in this case the **SPAD_MESSAGE**) to the offending train.

5.3.2 AWS

The AWS (Automatic Warning System) is used on our Newcastle to York and Oxford to Paddington routes. About 180m before most signals there's an "AWS ramp", which is placed just like a signal except that the ramp goes in the middle of the track instead of next to it. Like a signal it has a link, which in this case goes right above the ramp. When a train drives over the ramp it passes that link, triggering the *OnConsistPass* function in its script.

All that function needs to do is check the state of the next home signal up the line when a train starts passing its link. It does this by *calling* a code function **GetNextSignalState**,

which looks up the line until it reaches another signal link facing in the same direction and then triggers that signal's *GetSignalState* script function. Then it *calls SendConsistMessage* to send the appropriate **AWS_MESSAGE** to the train.

Here's what that looks like -

```
-- ON CONSIST PASS
--
function OnConsistPass ( prevFrontDist, prevBackDist, frontDist, backDist, linkIndex )

	-- If the front of the train is before the link and the back is beyond it,
	-- or vice versa, the train is in the process of crossing the link
	if ( frontDist > 0 and backDist < 0 ) or
	   ( frontDist < 0 and backDist > 0 ) then

		-- If the front and back of the train were previously both before the
		-- link, the train has just started crossing it forwards
		if ( prevFrontDist > 0 and prevBackDist > 0 ) then

			-- Request state of next signal
			local nextSignalState = Call( "GetNextSignalState", "", 1, 1, 0 )

			if (nextSignalState == CLEAR) then
				Call( "SendConsistMessage", AWS_MESSAGE, "clear" )

			elseif (nextSignalState == WARNING) or (nextSignalState == BLOCKED) then
				Call( "SendConsistMessage", AWS_MESSAGE, "blocked" )
			end
		end
	end
end
```

If the signal ahead is clear, we send our *AWS_MESSAGE* with the parameter "clear", and a bell rings in the train's cab to let the driver know he's clear to proceed. If the signal ahead is blocked or showing some kind of warning, we send the *AWS_MESSAGE* with the parameter "blocked" and a warning sounds in the cab until the driver hits a button to acknowledge it. If the driver doesn't press the button soon enough, the train applies its brakes automatically, but that's handled by the code and we don't need to worry about it here.

Like any other signal, the AWS ramp also needs to have *Initialise* and *OnSignalMessage* script functions. We don't need to *Initialise* anything though, so that function can be left empty, and all the *OnSignalMessage* function needs to do is forward on any message that it receives. Apart from that, all the script requires is for the necessary global constants (*CLEAR*, *WARNING*, *BLOCKED*, which are 0, 1 and 2, and *AWS_MESSAGE*, which is 11) to be declared. Like the shunt signal, the script is simple enough that it doesn't need to have any common files *included*.

5.3.3 TPWS

The TPWS (Train Protection & Warning System) is an additional system which is used on our Oxford - Paddington route. Like AWS, it requires a separate object to be added to the track, in this case a "TPWS Grid" which is placed just before the home signal it protects. Like the

AWS ramp, this grid has a link which should be placed directly above it (making sure that a train going in that direction will pass the grid's link *before* it passes the home signal's link 0), and the grid's script contains an *OnConsistPass* function which checks the state of the home signal just in front of it when a train drives past that link.

Unlike the AWS ramp, as well as checking whether the signal ahead is blocked the TPWS grid also checks if the train is going faster than the speed limit for this bit of track. If it is, it sends a **TPWS_MESSAGE** (12) to the train, which causes it to automatically apply its emergency brakes.

Here's how that works -

```
-- ON CONSIST PASS
--
function OnConsistPass ( prevFrontDist, prevBackDist, frontDist, backDist, linkIndex )

	-- If the front of the train is before the link and the back is beyond it,
	-- or vice versa, the train is in the process of crossing the link
	if ( frontDist > 0 and backDist < 0 ) or
	   ( frontDist < 0 and backDist > 0 ) then

		-- If the front and back of the train were previously both before the
		-- link, the train has just started crossing it forwards
		if ( prevFrontDist > 0 and prevBackDist > 0 ) then

			-- Request state of next signal
			local nextSignalState = Call( "GetNextSignalState", "", 1, 1, 0 )

			-- Find out how fast the train is going and the speed limit at this point
			local consistSpeed = Call ( "GetConsistSpeed" )
			local speedLimit = Call ( "GetTrackSpeedLimit", 0 )

			-- Send the appropriate TPWS_MESSAGE to the train
			if (nextSignalState == BLOCKED) then
				Call( "SendConsistMessage", TPWS_MESSAGE, "blocked" )

			elseif (consistSpeed > speedLimit) then
				Call( "SendConsistMessage", TPWS_MESSAGE, "overspeed" )

			elseif (nextSignalState == WARNING) then
				Call( "SendConsistMessage", TPWS_MESSAGE, "warning" )

			elseif (nextSignalState == CLEAR) then
				Call( "SendConsistMessage", TPWS_MESSAGE, "clear" )
			end
		end
	end
end
```

As you can see, this is very similar to the AWS script, except that we're calling two new code functions (**GetConsistSpeed** and **GetTrackSpeedLimit**) to check if the train is going too fast. *GetConsistSpeed* can be called from any signal's *OnConsistPass* function and will return

the speed of the train that triggered it. *GetTrackSpeedLimit* can be called from any signal script function and returns the speed limit at a specific link belonging to that signal (in this case we've only got one link, so we're checking the speed at link 0). Both functions return the speed in meters per second, so we can compare them directly to see if the train is speeding.

5.3.4 Creating Your Own Warning System

Using the existing code, it's possible to create your own warning systems. For example, the German Indusi / PZB system can be replicated using a mixture of AWS and TPWS messages. This can either be done by adding separate PZB inductor magnets next to the track in the appropriate places, or by simply adding the necessary checks to the *OnConsistPass* function used by standard German home and repeater signals.

When you pass the 0 link of a repeater signal, if the next home signal up the line is red the Indusi system triggers a warning just like the AWS ramp, so an *AWS_MESSAGE* with the "blocked" parameter should be sent to the train. When you pass the 0 link of the home signal, if it's red the train will automatically be stopped, just like the TPWS grid, so you can send a *TPWS_MESSAGE* with the "blocked" parameter to the train.

The Indusi system also checks to make sure that if you're passing a home signal that's set to warning you have slowed to 40kmph. Again, this can be done by sending the "overspeed" *TPWS_MESSAGE* if the consist's speed is more than 40kmph (which is 40 / 3.6 in meters per second).

It should be possible to simulate the behaviour of many other warning systems from around the world using a similar combination of AWS and TPWS messages.

5.4 Handling Yard Entries

Although most of a route will normally be covered by home signals, some areas (particularly in yards) are "dark". Once a train goes into a yard it probably won't encounter another home signal until it leaves the yard, and the signals outside the yard generally won't know or care about trains inside the yard.

To handle this, there are special signals which have one or more of their links flagged in their *Initialise* script function as being a "yard entry". These links ignore the occupancy of the track beyond them, which involves making a few simple changes to their *OnConsistPass* function -

- When a train starts passing a yard entry link forwards, we don't increment the link's *gOccupationTable* entry, because once the train is in the yard the signal is no longer interested in it.
- When a train finishes passing a yard entry link forwards, if that was the only train in *gOccupationTable*[0] the signal is now clear and we run the *NotOccupied* function to make it turn green.
- When a train starts passing a yard entry link backwards, we didn't know about that train before so we need to run the *Occupied* function to make it turn red now.
- When a train finishes passing a yard entry link backwards, we don't decrement the link's *gOccupationTable* entry.

As we don't care about trains inside the yard or the state of any signals covering the other exits from the yard, we also need to edit the *OnSignalMessage* function to ignore any messages it receives on a yard entry link, if the message is coming from within the yard (ie, from direction 1).

And that's it. Just place the signal using that modified script on the mainline somewhere before the yard entry you want to cover, and put the appropriate link a little way inside the yard, and your signal will turn green again as soon as a train finishes passing that link and disappears into the yard.

If you need one signal to cover multiple yard entries, you can just flag more of the links as yard entries. Again, by making your changes as generic as possible you can re-use the same basic script functions for multiple signals with different numbers of links and yard entries. In the case of the UK Colour Light signals, for example, we have lots of yard entry signal scripts, but they only contain *Initialise* and (if they have any feathers) *ActivateLink* functions - the rest of the scripting functions for them are kept in a small set of common files that the signal-specific scripts *include*.

5.5 Covering Reverse Junctions

On the Bath - Templecombe route, several of the junctions are designed for trains to cross between tracks or enter a siding by driving past the junction and then reversing up it.

If the signals before the junction are only covering that one junction, that's fine. But if you want one signal to cover multiple junctions of this type, you run into a problem. Normally you would place link 0 next to the signal and link 1 beyond the last junction you're covering. But on this route, trains will often drive past the signal and then reverse back up another track, without the rear of the train ever passing link 1. This means that gOccupationTable[0] never gets decremented back to 0, so even after the train has reversed onto another line and the junction has switched again behind them, the signal they passed remains blocked.

The way around this is to do something which would normally be incorrect - placing multiple links for the same signal on the same line, putting one after each junction. Each link keeps track of which link (if any) it's connected to, and so wherever you drive your train and however you switch the junctions, the signal will always know whether the route in front of it is blocked.

Let's consider a simple case - a signal that has to cover two junctions that merge onto the track ahead of it. This signal will have three links - link 0 next to the signal as normal, link 1 just beyond the first junction, and link 2 just beyond the second junction.

First, in the *Initialise* function we need to declare a new table **gSpecialConnectedLink** to keep track of which links are connected to each other -

```
-- Default to all links disconnected until we know otherwise
gSpecialConnectedLink = {}
gSpecialConnectedLink[0] = -1
gSpecialConnectedLink[1] = -1
```

Next, in our *ReactToSignalMessage* function, we need to make sure the signal reacts to messages arriving on the correct links. Most messages should be ignored unless they arive on our last link, except the *INITIALISE_SIGNAL_TO_BLOCKED* and *RESET_SIGNAL_STATE*

messages (which can be received by any link) and the *JUNCTION_STATE_CHANGE* message (which should trigger our *OnJunctionStateChange* function when it arrives on any link *except* the last one). We also need to make a slight change to how the OCCUPATION_DECREMENT message is handled, to make sure that it checks the occupation table entries for all of the links (0, 1 and 2) are zero before setting the signal as *NotOccupied*.

Our *OnJunctionStateChange* function needs changing next, so that it keeps track of which link each of the other links is connected to.

```
-- JUNCTION STATE
--
function OnJunctionStateChange( junction_state, parameter, direction, linkIndex )

	-- If this is the first time we've checked our junction state, check all links
	if not gInitialised then

		-- Check if link 0 is connected to link 1 and link 1 is connected to link 2
		gSpecialConnectedLink[0] = Call( "GetConnectedLink", "", 1, 0 )
		gSpecialConnectedLink[1] = Call( "GetConnectedLink", "", 1, 1 )

	-- Otherwise, if the junction change message didn't arrive on the last link...
	elseif linkIndex < 2 then

		-- check if this link is connected to the next one up the line
		gSpecialConnectedLink[linkIndex] = Call("GetConnectedLink", "", 1, linkIndex)
	end

	-- If all the links are connected, gConnectedLink = 2
	if gSpecialConnectedLink[0] == 1 and gSpecialConnectedLink[1] == 2 then
		gConnectedLink = 2

	-- Otherwise gConnectedLink = -1
	else
		gConnectedLink = -1
	end

	-- If we're connected all the way through and route is clear, signal is CLEAR
	if gConnectedLink == 2 and
	  gOccupationTable[0] == 0 and
	  gOccupationTable[1] == 0 and
	  gOccupationTable[2] == 0 then
		NotOccupied( 0 )

	-- Otherwise set signal as BLOCKED
	else
		Occupied( 0 )
	end
end
```

Finally we need to make a couple of changes to the *OnConsistPass* function to handle the fact that our links are now all on the same line covering converging junctions, instead of on different diverging routes.

When a train starts to pass a link other than link 0 in reverse, we should increment the occupancy of the link behind it *if* that link is connected to us -

```
-- If the train just started crossing a link > 0 going backwards...
elseif (linkIndex > 0) then

    -- And this link is connected to the previous one...
    if (specialConnectedLink[linkIndex - 1] == linkIndex) then

        -- Increment the previous link's occupation table
        gOccupationTable[linkIndex - 1] = gOccupationTable[linkIndex - 1] +
1
    end
end
```

Similarly, when a train finishes passing one of those links going forwards, we should only decrement the occupancy of the link behind it if that link is connected to us -

```
-- If the train just finished crossing a link > 0 going forwards...
elseif (linkIndex > 0) then

    -- And this link is connected to the previous one...
    if (specialConnectedLink[linkIndex - 1] == linkIndex) then

        -- Decrement the previous link's occupation table
        gOccupationTable[linkIndex - 1] = gOccupationTable[linkIndex - 1] - 1
    end
end
```

Finally, when a train finishes passing link 0 in reverse, we need to check *gConnectedLink* is 2 and that the occupation table is empty for *all* our links (0, 1 and 2) before telling the signal it's *NotOccupied*.

Now if a train drives past this signal, stops just past the first junction (link 1), switches the junction and reverses back up it onto another line, all of the occupation tables will update correctly and the signal will know it no longer has a train blocking the track ahead of it. If you're feeling really ambitious, you can even script signals that handle a mix of diverging *and* converging junctions in this way. An example of this is the UK Semaphore signal script "Sem_DvgeRte_hh Special 1.lua", which is used by a signal near Evercreech Junction.

5.6 Handling Signals With Lots Of Aspects

The European signals we've created for Rail Simulator are mostly fairly straightforwards - they show if the line ahead is blocked or clear and (in some cases) have one or two warning aspects they can show as well.

But some signal systems (for example, North American ones) are incredibly complex and (using a combination of multiple heads, flashing lights and sometimes even moving parts) can show several different warning aspects depending on everything from the state of junctions further up the line to the type of train approaching the signal.

In these cases, using simple functions like *Occupied* and *NotOccupied* to control the state of the signal is obviously not sufficient. Instead, it sometimes makes more sense to combine all that functionality into a single function which can be called whenever something happens that might change the state of the signal (such as a signal message being received or a junction being switched), and takes all of the necessary factors into account to decide what state the signal should be in now, and which of its lights should be switched on or off.

For example, this would control a two headed US signal as used by BNSF in California -

```
-- DETERMINE SIGNAL STATE
-- Figures out what lights to show on each head based on the state of the signal
--
function DetermineSignalState()

	local newState = -1

	-- If the junction is broken
	if gConnectedLink == -1 then

		-- Stop
		SetState( { ANIMSTATE_RED, ANIMSTATE_RED } )
		newState = SIGNAL_BLOCKED

	else
		local switchSpeed = gSwitchSpeed[gConnectedLink]

		-- Adjust speed according to train type
		if switchSpeed > SPEED_SLOW then

			-- If we're a freight train, use next speed down if diverging at switch
			if gApproaching == CONSIST_TYPE_FREIGHT then
				switchSpeed = switchSpeed - 1
			end
		end

		-- If there's a train in this signal's block
		if (gOccupationTable[0] > 0) or
		   (gConnectedLink > 0 and gOccupationTable[gConnectedLink] > 0) then

			-- Stop
			SetState( { ANIMSTATE_RED, ANIMSTATE_RED } )
			newState = SIGNAL_BLOCKED

		-- If we're going straight on at this signal...
		elseif gConnectedLink < 2 then

			-- And the route ahead is restricting
			if gRestricted[gConnectedLink] then

				-- Restricting
				SetState( { ANIMSTATE_LUNAR, ANIMSTATE_RED } )
				newState = SIGNAL_RESTRICTED
```

```
		-- And the route ahead is at warning or approach restricting
		elseif gLinkState[gConnectedLink] == SIGNAL_WARNING
			or gLinkState[gConnectedLink] == SIGNAL_APPROACH_RESTRICTED
then

			-- Approach
			SetState( { ANIMSTATE_YELLOW, ANIMSTATE_RED } )
			newState = SIGNAL_WARNING

		-- And the route ahead is at clear, warning2 or approach restricting2
		elseif gLinkState[gConnectedLink] == SIGNAL_CLEARED
			or gLinkState[gConnectedLink] == SIGNAL_WARNING2
			or gLinkState[gConnectedLink] == SIGNAL_APPROACH_RESTRICTED2
then

			-- And we're also going straight on at the next signal
			if gRouteState[gConnectedLink] == SIGNAL_STRAIGHT then

				-- Clear
				SetState( { ANIMSTATE_GREEN, ANIMSTATE_RED } )
				newState = SIGNAL_CLEARED

			-- And we're diverging at the next signal
			elseif gRouteState[gConnectedLink] == SIGNAL_DIVERGING then

				-- And the next signal's switch speed is SLOW (40mph)
				if switchSpeed == SPEED_SLOW then

					-- Approach Medium
					SetState( { ANIMSTATE_FLASHING_YELLOW,
ANIMSTATE_RED } )

				-- And the next signal's switch speed is MEDIUM (50mph)
				elseif switchSpeed == SPEED_MEDIUM then

					-- Advance Approach
					SetState( { ANIMSTATE_YELLOW,
ANIMSTATE_GREEN } )

				-- And the next signal's switch speed is LIMITED (60mph)
				elseif switchSpeed == SPEED_LIMITED then

					-- Approach Limited
					SetState( { ANIMSTATE_YELLOW,
ANIMSTATE_FLASHING_GREEN } )
				end

				newState = SIGNAL_CLEARED
			end
		end
```

```
		-- If we're diverging at this signal...
		else

			-- And the route ahead is restricting
			if gRestricted[gConnectedLink] then

				-- Restricting
				SetState( { ANIMSTATE_RED, ANIMSTATE_LUNAR } )
				newState = SIGNAL_RESTRICTED

			-- And the route ahead is at warning or approach restricting
			elseif gLinkState[gConnectedLink] == SIGNAL_WARNING
				or gLinkState[gConnectedLink] == SIGNAL_APPROACH_RESTRICTED
then

				-- Diverging Approach
				SetState( { ANIMSTATE_RED, ANIMSTATE_YELLOW } )
				newState = SIGNAL_WARNING

			-- And the route ahead is at clear, warning2 or approach restricting2
			elseif gLinkState[gConnectedLink] == SIGNAL_CLEARED
				or gLinkState[gConnectedLink] == SIGNAL_WARNING2
				or gLinkState[gConnectedLink] == SIGNAL_APPROACH_RESTRICTED2
then

				-- Diverging Clear
				SetState( { ANIMSTATE_RED, ANIMSTATE_GREEN } )
				newState = SIGNAL_CLEARED
			end
		end
	end

	-- If our state has changed and we've got a valid new state...
	if newState >= 0 and newState ~= gSignalState then

		-- Update our state and send back a message to let the signal behind us know
		gSignalState = newState
		Call( "SendSignalMessage", newState, "", -1, 1, 0 )
	end
end
```

As you can see, this function has to deal with a bewildering set of factors - what the state of the next signal up the line is, whether we're diverging or going straight ahead at both this signal and the next signal, the speed we should cross the next junction at, whether we're going into an area with a restricted speed limit (such as a yard), what type of train is approaching the signal... Handling all of this through separate functions would be completely impractical. Instead we have this one function which can be called from the *OnConsistPass*, *OnJunctionStateChange* and *ReactToSignalMessage* functions whenever the signal's state might need to change.

Notice that we've also included some functionality that's normally found in the *SetState* function. Instead of telling *SetState* what state we're switching to and letting it decide which

lights to turn on and off based on that, we're handling all of that inside this function and telling *SetState* which lights need to be active on each of the signal's two heads.

6 Signal Debugging

When you're scripting a signal, particularly if they're as complex as the ones we've just been looking at, it's often useful to be able to see exactly what your script is doing in the game to ensure that it's behaving as you expected it to.

DEBUGGING SUPPORT STILL TO BE CONFIRMED - TOOLS WON'T SHIP WITH LOGMATE, SHOULD HAVE AN EQUIVALENT TOOL TO DISPLAY DEBUG INFORMATION THOUGH

6.1 Using LogMate

The tool we use to get that information from our signal scripts while the game is running is called LogMate. You can activate LogMate by right clicking on your Rail Simulator shortcut to bring up its properties, and then adding the following options to the end of the "Target".

```
-LogMate -SetLogFilters="Script Manager"
```

When you next start up the game it will automatically launch LogMate, and enable its script debug channel. Any error messages generated by your scripts (for example, if the game code tries to call a function that doesn't exist in your script) will appear in LogMate.

6.2 Debug Messages

To get more useful information from scripts, you can add debug messages to them. For example, say you want to know which line is connected when the *OnJunctionStateChange* function runs. You can do this by adding the following line to the end of the function -

```
Print( ("DEBUG: OnJunctionStateChange - connected link = " .. gConnectedLink) )
```

The **Print** function prints whatever you put inside its brackets in LogMate's script channel. In this case, we're outputting the name of the function and the value of *gConnectedLink*. Anything inside the brackets is displayed in LogMate exactly as you typed it. The .. after that tells the *Print* function to add whatever comes next to the end of the preceding text - this is called "concatenation".

Notice that we put two pairs of brackets around the text we're outputting - this is necessary when we're creating a message by concatenating two or more pieces of text. If we were just outputting a simple preset string (for example, "Hello world"), we would only need one pair of brackets.

In this case, we're using concatenation to add gConnectedLink to the end of our message. Because we've put it outside of quote marks, this doesn't output "gConnectedLink" - it outputs the current value of the variable gConnectedLink. So if link 1 is now connected, the message that appears in LogMate would look like this -

```
DEBUG: OnJunctionStateChange - connected link = 1
```

Or, to be more precise, it would look something like this -

Trace C:\Dev\RailSim\Code\DLLs\ScriptManager\cScriptState.cpp : 734 = ID: railnetwork\signals\uk semaphore\lattice_posts\b lattice combsig_hd 2t.xml long: -2.376435, lat: 51.382504 DEBUG: OnJunctionStateChange - connected link = 1

Don't worry about the "trace" - the useful information is everything after it says 734. The *ID* is the filename of the blueprint used by the signal that outputted the message, so you know exactly what type of signal each debug message was generated by. The *long* and *lat* gives the precise position of the signal in the world, so you can quickly locate the signal that outputted the message. And finally there's the debug message itself. Also, if you look at the main window of LogMate rather than the Script Manager window, you'll also see a time stamp at the start of each line, so you can see when the debug message was outputted.

Debug messages can be incredibly useful, and can output a lot of information in a single line. For example, say we want to check if a message is reaching a signal. We can do that by adding this line to the top of that signal's *OnSignalMessage* function -

```
Print (("DEBUG: OnSignalMessage - message = " .. message .. ", parameter = " .. parameter .. ", direction = " .. direction .. ", linkIndex = " .. linkIndex))
```

As you can see, we're concatenating multiple text strings and variable values here. Anything between a pair of quote marks is outputted as is (to make it clearer, we've highlighted that plain text in green). For everything else, the game fills in the value of the specified variable. So in this case, the debug output would look something like this -

DEBUG: OnSignalMessage - message = 13, parameter = , direction = 1, linkIndex = 1

If you look up this message value in Common Signal Script.lua, you'll find that message 13 is SIGNAL_CLEARED. So we now know that this signal has received an all-clear from the next signal up the line beyond link 1.

As you can imagine, this kind of information can be incredibly useful when you're trying to figure out why a signal isn't working as you expected it to. However, outputting lots of *Print* messages to LogMate slows the game down, so you won't want to leave them in. On the other hand, going through your scripts and commenting out all your debug messages, and then having to comment them back in again if you want to do some more debugging later on, can be incredibly tedious. So our Common Signal Script file includes a handy function called DebugPrint. Here's what that looks like -

```
function DebugPrint( message )
	if (DEBUG) then
		Print( message )
	end
end
```

All this does is check a global variable called DEBUG. If DEBUG is true, the message is *Printed*. If DEBUG is false, it isn't. You can now switch your script's debug output on and off simply by changing the value of the DEBUG variable in your script and re-exporting it.

If you look at our signal scripts, you'll see that they contain a *lot* of DebugPrints, so that we can see exactly what the signal is doing at each step of every function - which link is connected, how many trains are in its occupancy table, which messages it's receiving etc. All

of that information can be accessed simply by setting DEBUG to true in the script of the signal type(s) you want to check.

6.3 Signal Validation Tool

If your signal isn't working, it isn't necessarily a problem with the signal script - it could be the way the signal has been placed in the route. Luckily some of the most common errors can be picked up by the Signal Validation Tool. To activate the SVT, add the following to the Target of your Rail Simulator shortcut -

-ValidateSignals

You'll also need to modify the log filters to allow the SVT output to appear in LogMate -

-SetLogFilters="Script Manager,Signals"

The next time you run the game, it will search through all the signals in whatever route you load and check that their links are correctly placed. If everything is correct, you won't get any output in LogMate. But if it finds anything it suspects might be an error, it will output a message in the Signals window in LogMate.

These are the errors that the SVT outputs -

Signal link found when junction expected -

The SVT searched forwards from the signal's link 0 and found another link beyond it with no junction between them. Occasionally this is intentional, but usually it means the link it mentions needs to be moved a bit further out. The error also tells you the longitude and latitude of the link that it thinks is misplaced.

As we mentioned at the start of this doc, if you want to be sure that your link is in the right place you can check it by clicking on the Linear Object Tool icon in the editor. When you do this, all the junctions in the game will be indicated by red triangles.

1 unprotected exit: 1 routes found from... -

The SVT searched forwards from one of the signal's links and found a line which doesn't have another link belonging to that signal on it. The error message tells you which link the unprotected line was found beyond, and the longitude and latitude of that link.

Sometimes this is intentional. For example, you may have not put a link on that route because you don't want trains going in this direction to ever go down that line (for example, if the line is abandoned or is only for trains going in the other direction). And occasionally this error seems to pop up when there's nothing obviously wrong with the link placement. But it's always worth checking them out, to be safe.

Unprotected junction -

The SVT found a junction that doesn't appear to be protected by a signal. The first longitude and latitude is the position of the junction that's unprotected. The second set of co-ordinates is the position of the signal link which the junction is beyond. You often get a *lot* of these in a route, and a single missing or misplaced link can result in several unprotected junction errors for the same signal.

As with the unprotected exit error though, this isn't always a problem - the junction might be on a route that you don't want to put a link on. But it's usually worth quickly checking through these errors, in case one of them is a genuine issue.

7 Conclusion

Hopefully having read this document you now have a working knowledge of how signals are setup, how scripts control their behaviour, which functions the scripts require, and how those functions all fit together.

Although Rail Simulator's signalling system is undeniably complex, the good news is that we've already done a lot of the hard work for you. As you've seen from the examples in this document, a lot of the signal scripts that shipped with the game are fairly generic, and can be easily adapted to work with a wide range of different signal types. New signals can often be created simply by copying an existing signal script and editing it to suit your own needs.

8 Further Reading

If you're still unsure about anything, looking at existing scripts that behave similarly to the signal you're trying to create is often a good place to start. Our signal scripts tend to contain a lot of comments and debug messages, which should help you understand what they're doing and why.