



www.trainzland.com

presents

A small Guide to Scripting a Cabin

David "Marinus" Hamann © 2005

trainz@marinus.org

This was originally planned as answer to a question regarding scripting a cabin, however I thought this might be useful for others too. Consider this document as an overview how the different functions of a cabin script work together and how this is linked to the settings in the config.txt File.

Basics

If you start to create an interior, you have different possibilities. If you want to use the default features of TRS, you don't have to write a script for the interior. As long as you're using the same names for your levers as Auran put in the documentation (Content Creators Guide) there is no need for any scripting. If you don't specify a script for your cabin, Trainz will use the default locomotive script (DefaultLocomotiveCabin.gs in your Trainz\Scripts folder).

If you are creating a steam cabin, and don't want to end up with Diesel/Electric controls, you have to tell Trainz to use another cabin script Auran wrote already (DefaultSteamCabin.gs in your Trainz\Scripts folder). Because you cannot link to the Trainz\Scripts folder directly from your config.txt, you have to create a script file (.gs) yourself, but all you do in this file, is reference to the Auran default script.

It would be enough to have a script file that has this content:

```
include "DefaultSteamCabin.gs"

class MySteamCabin isclass DefaultSteamCabin
{
    public void Init(void)
    {
        hasAnimatedFireman = false;
        inherited();
    }
};
```

If we assume this script has a filename: mysteamcabin.gs the setting in config.txt would be:

```
script "mysteamcabin"
class "MySteamCabin"
```

If your controls have the right name (i.e. the ones you can find in the CCG) this is all you need for a default steam cabin.

The same procedure can be used for non steam cabins too. You only have to change the include statement to *include "DefaultLocomotiveCabin.gs"* and the command after *isclass* to *DefaultLocomotiveCabin*.

Enhancing the default cabin features

As soon as you want to implement features that differ from the default TRS2004 features, you have to extend the default cabin scripts (either *DefaultLocomotiveCabin* or *DefaultSteamCabin*), or you do anything from scratch.

Both classes *DefaultLocomotiveCabin* and *DefaultSteamCabin* base on the same parent class *Cabin*. If you decide you cannot use any of the functions / features that these two classes offer, you should base your script on the *Cabin* class. This might be the best solution for special cabins like those of Subway cars, or streetcars.

If you only want to enhance the default functions on minor points (like implementing fans, or wipers, etc.) you can base your script on the default scripts, and implement only your enhancements, and leave the handling of the remaining features to the default scripts, by calling the implementation of the parent class's function (Keyword: *inherited()*)

What all these functions do, will be described on the next pages of this guide.

Connection between config.txt and script file

Of course you need some kind of connection between the config.txt of a cabin and its script file. If there was no connection, how would a script file know which levers, needles, etc. exist in the cabin, which values, positions, etc they can have and so on.

Every control in a cabin can be represented in a script as a CabinControl Object. You can execute various functions on a CabinControl Object. For example you can read its position.

If you are not using one of the default scripts, or you are enhancing one of the default scripts with an additional cabin control, you have to define all cabin controls you want to use in your script.

```
CabinControl speedometer;           //Speed indicator.
CabinControl throttle_lever;        //Throttle lever.
CabinControl reverser_lever;        //Reverser lever.
```

But how does a CabinControl object know what cabin control (defined in the config.txt) it represents? The answer to this question can be found in the *Init* function of the cabin script.

```
public void Init(void)
{
    speedometer = GetNamedControl("speedo_needle");
    throttle_lever = GetNamedControl("throttle_lever");
    reverser_lever = GetNamedControl("reverser_lever");
}
```

The *GetNamedControl* function returns a CabinControl object that is linked to the config.txt control, with the name you passed as parameter. Look at parts of the config.txt for this example:

```

Mesh-Table
{
  Default
  {
    Mesh "interior.im"
    Auto-Creat 1
  }
  reverser_lever
  {
    Kind "lever"

    ...

  }
  speedo_needle
  {
    Kind "needle"

    ...

  }
  throttle_lever
  {
    Kind "lever"

    ...

  }
}

```

The “Update” Function

You can think of the *public void Update(void)* function as an output function. This function is running almost continuously, and adjusts the visual display of levers and needles based on train physics. A good example for this is the speedometer needle:

```

if (speedometer)
  speedometer.SetValue(train.GetVelocity());

```

This does nothing else than query the actual speed of the train and set the value of the Speedometer needle. The same would work for example for the reverser lever

```

if (reverser_lever)
{
  reverser_lever.SetValue(loco.GetEngineSetting("reverser"));
}

```

The `config.txt` entry for the `reverser_lever` could look like this:

```

reverser_lever
{
  kind "lever"
  auto-create 1
  mesh "reverser_lever.pm"
  att "a.reverser_lever"
  limits 0,2
  angles 0.55,-0.55
  notches 0,0.5,1
}

```

```
notchheight 1,1,1
att-parent "default"
}
```

But let's stick with this example for a moment.

loco.GetEngineSetting("reverser") returns the values 0, 1 and 2 (see Trainz Script API Document for "Reverser Traction Modes") depending on the engine setting of the reverser (Note: This is completely independent of the position of the reverser lever you can see in your cabin). Now that we know which value the engine setting of the reverser is we can adjust the visual display of the reverser lever.

You can see in the config.txt that this lever has defined notch positions on 0, 0.5 and 1. So the question is how SetValue() and the defined notch positions come together? The secret is the limits parameter. This parameter tells Trainz the mathematical boundaries for values this lever can represent, or in other words, which range of values we can pass with the SetValue() function for this lever.

The defined notch positions could be considered as percentage of this range. In other words, this means if we use the SetValue Function Trainz calculates how much percent of our range (in this case $[[0,2]] = 2$) the passed value is, and sets the lever to the appropriate notch. For example:

```
reverser_lever.SetValue(1);
```

The passed value of 1 is 50% of 2, so the lever will be set to notch position 0.5.

Let's take a look on another example, this time we look at a throttle lever.

```
throttle_lever
{
kind "lever"
auto-create 1
mesh "throttle_lever.pm"
att "a.throttle_lever"
limits 0,16
angles 1.2,0
notches 0,0.0625,0.125,0.1875,0.25,0.3125,0.375,0.4375,0.5,0.5625,
0.625,0.6875,0.75,0.8125,0.875,0.9375,1
notchheight 2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2
radius 0.35
att-parent "default"
}
```

This throttle lever has 17 Positions (from 0 to 16). This means the limits are 0 and 16, and we need 17 defined notch positions. Each position is a "16% of 1" increment of the previous position, starting with 0 (16% of 1 is 0.0625). A call of

```
throttle_lever.SetValue(4)
```

will cause the throttle_lever to be set to notch position 0.25 since 4 is 25% of 16.

Note: Please keep in mind that a call of throttle_lever.SetValue(x) won't affect the train physics in any way. The train will not start to move, or something like this. This function call will only change the visual position of the throttle lever. This is why I said the Update function can be thought of as an "Output" function.

If you are enhancing one of the default scripts, your script's update function could look like this:

```
public void Update(void)
{
    if (!loco or !train)
        return;

    if (speedometer)
    {
        speedometer.SetValue(train.GetVelocity());
    }

    inherited();
}
```

In this example the update function only handles the speedometer. All other visual updates should be handled in the parent class (e.g. DefaultLocomotiveCabin).

The “UserSetControl” Function

The *public void UserSetControl(CabinControl control, float value)* function is one of two input functions. This means this function handles some kind of user input. In this case the function handles if the user changes one of the levers of the cabin, using the mouse. *UserSetControl* has two parameters:

- *CabinControl control* – this is the lever the user changed
- *Float value* – this is the value the lever was set to (the same value we already discussed for the Update function)

So what should we do if this function is called? Well the user has changed one of our cabin's levers, so let's adjust the train physics engine parameter.

```
void UserSetControl(CabinControl control, float value)
{
    ...
    if (control == reverser_lever)
    {
        loco.SetEngineSetting("reverser", value);
    }
    else if (control == throttle_lever)
    {
        loco.SetEngineSetting("throttle", value);
    }
    ...
}
```

What happens first is that we find which cabin control has been altered. As soon as we find the right control, the *SetEngineSettings* function is called, and sets the engine parameter to the value of the lever.

As you can see already, we are now setting the engine parameter that we were reading in the *Update* function. If we would not set the engine parameter, two things would happen. First we would not see any effect in the train physics (e.g.. the train would not throttle up if we move the throttle lever forwards) and second the next time the *Update* function is called, the lever would be moved back to the old position, because the update function reads the engine parameter which we didn't change.

If you are enhancing the default scripts, you only need to implement the part of this function that handles your extended controls. For the other controls you can rely on the implementation in the default script. Use the *inherited()* Keyword like for the *Update* function.

```
void UserSetControl(CabinControl control, float value)
{
    ...
    if (control == reverser_lever)
    {
        loco.SetEngineSetting("reverser", value);
    }
    else if (control == throttle_lever)
    {
        loco.SetEngineSetting("throttle", value);
    }

    ...

    else
    {
        inherited(control, value);
    }
}
```

The UserPressKey Function

The *void UserPressKey(string s)* function is the second input function of your cabin. This function is called when a key on the keyboard is pressed.

Unfortunately Auran has not implemented this for any key but only for a few, specific keys.

The string parameter of this function does not contain the key itself like "w" or "s", but a descriptive name of the key (e.g. "steam-regulator-up"). The following table lists the keys that were implemented by Auran:

Key	Descriptive name
W	steam-regulator-up
S	steam-regulator-down
A	train_cabin_brake_application
I	steam-injector-up
O	steam-injector-down
Space	shovel-coal
Shift+Space	train_cabin_coalman_wave
N	steam-blower-up
Shift+N	steam-blower-down
F	steam-reverser-up
R	steam-reverser-down
Shift+F	cabin-fans

If you are enhancing one of the default scripts, or your script is based on the *Cabin* class, you might want to handle pressed keys in your script.

```
void UserPressKey(string s)
{
    if (!loco)
    {
        return;
    }
    ...

    if (s == "steam-regulator-up")
    {
        ...
    }
    else if (s == "steam-regulator-down")
    {
        ...
    }
    else if (s == "train_cabin_brake_application")
    {
        ...
    }
    else
        inherited(s);
}
```

In this example we check if the user pressed *W* (*steam-regulator-up*), *S* (*steam-regulator-down*) or *A* (*train_cabin_brake_application*). If any other button has been pressed, the parent implementation of *UserPressKey* would be called, because of the *inherited()* command in the last if/else branch.

Let's take a look on what we could do after we found out which button the user pressed. In this example we will look on what might happen if the user pressed the *W* (or *Num 8*)

steam-regulator-up key. This script won't do anything else than throttle up the engine one notch. Yes this is the same as the default script does anyways, but you could write anything else you want to happen if this key is pressed.

```
...  
  
if (s == "steam-regulator-up")  
{  
    int notch = loco.GetEngineSetting("throttle");  
    if (notch < 16)  
    {  
        loco.SetEngineSetting("throttle", notch + 1);  
    }  
}  
  
...
```

The first thing we do is, to read the current engine parameter for throttle using the *GetEngineSetting("throttle")* function. In our previous example we had 17 throttle positions (from 0 to 16). As long as the current position is lower than 16 we still can throttle up one notch. We do this using the *SetEngineSetting("throttle", notch + 1)* function. This function only affects the internal train physics, but won't change the visual display of our throttle lever. Don't worry, the next time Trainz calls the Update function (see above) our script will read the engine parameter we changed right now, and update the position of the visual throttle lever.