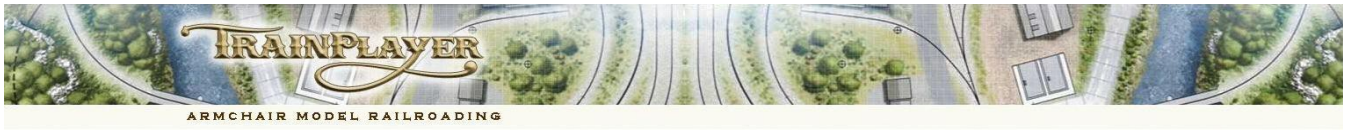


Simplified Scripting with Subroutines

PART 1 - WHY WE NEED THE SUBROUTINES

Introduction

The original TrainPlayer Scripting Language was relatively easy for a novice "scripter" to understand with a Command structure that included simple keywords like Forward, Reverse, Stop, Speed 10 , Throw j10 0, Uncouple H14 etc. The downside was that it took a lot of patience and attention to detail to get things right and that scripts became very lengthy and quite difficult to follow (especially if you tried to read and modify any of the scripts produced by the script recorder).

The much more powerful TrainPlayer Programming Language (TPL) released with version 6.0 can appear even more intimidating to the Novice with its extremely powerful inventory of User Variables, System Variables, Conditional Statements and System Functions.

While this powerful language can be exploited by anyone with a modicum of computer programming experience, it is recognized that such individuals may not represent the vast majority of TrainPlayer users. The newly released v6.1 adds even more powerful structures to the language to satisfy the whims of a few keen scripters, but this has the side effect of presenting the average user who may have no programming experience with *incomprehensible statements* to deal with; like the following block of code:

```
let t = $x_train
let tt = $train(@t,timetable)
if (tt <> "")
   let departure = $string(@tt,nexttoken,",")
   if (tt <> "")
      let $train(@t,timetable) = @tt
   endif
   if (@departure = "thru")
      echo $rrclock - @t does not stop here.
      return
   endif
   if ($string(@departure,contains,":") = 0)
      Let departure = $substr(0,2,@departure) + ":" + $substr(2,2,@departure)
   endif
   echo $rrclock - @t is scheduled to depart at @departure
   while (@departure > $rrclock)
      stop
   endwhile
   echo $rrclock - @t is now scheduled to depart.
endif
```

If you can understand the above "*gobbledegook*" there is no need to read any further, you can just happily "Carry on Scripting" .

This document is intended for TrainPlayer users who would like to be able to take advantage of some of the power of the upgraded scripting language for automating various parts of their layouts without having to learn all the complexities of the TPL code.

The Subroutine Library

In an effort to generate more interest in the scripting and automation aspects of TrainPlayer, version 6.1 now comes with a comprehensive library of subroutines which are designed to demystify the more abstruse aspects of TPL and provide a powerful set of scripting tools that can be used to disguise some of the more complicated aspects of the language behind an easier to use interface.

The purpose of this document is to demonstrate that by using the new subroutine library you can have access to many of the powerful TPL features for automating your trains without having to learn how to use all of the complex function calls. However to use the Subroutine Library you will still need to learn a few of the basic TPL keywords, and you will be making extensive use of the **CALL** command.

For example the *gobbledegook* block of code in the opening paragraph represents the TPL commands you would have to write to hold up a train until a scheduled departure time is reached. The timetable is carried on the train and the subroutine continually checks the scheduled departure against the layout clock time, when the departure time is reached the subroutine terminates and the script continues.

To use this *gobbledegook* block of subroutine code you need only enter in your script:

```
call holdfortimetable
```

What you need to know to use the subroutines from the library

Each subroutine must be called from either the layout Master Script, a Train Script or a Junction Action Script, so you will need to know how to create each of these script types. You can of course continue to include all of the basic train control commands such as Forward, Reverse, Stop, Speed 10 , Throw j10 0, Uncouple H14 etc. in these same calling scripts.

To create a Master Script.
Open the Script Central dialog, ensure you are on the Scripts Tab, right click in the grid and select "Create Master Script".

To create a Train Script Select the appropriate train and click the Edit icon on the Script toolbar.

To create a Junction Action
Ensure the TrackLayer pointer is selected on the TrackLayer toolbar, right-click on the track where you want to place your script, and select "Create action here". Alternatively right-click on an existing junction (track connection or switch) and select "Action".

The best script method to use is largely a matter of personal preference. Subroutines that perform layout specific tasks such as setting the fast time clock, setting a size and position for the default Note window, or just starting trains moving are probably best suited to the Master Script, though they can also go in Train Scripts or Junction Actions.

Subroutines for train control are best suited to calling from Train Scripts or from Junction Actions (my personal preference), while scripts related to actions for specific cars must be called from Junction Actions set up to be triggered by the specific cartype or aar code.

An explanation of Arguments

Many of the subroutines in the library only require the use of the keyword CALL followed by the Subroutine name, while others require arguments passing to the subroutine for processing.

So you will need to be aware of the concept of what constitutes an argument and how it is passed. When arguments are needed, the subroutine name is followed by a space then the first argument, another space and then the second argument etc. If any of the arguments contain spaces themselves then they must be quoted.

Example of a subroutine call with three arguments:

```
call setstartclock 0800 x10 S
```

Argument 1 is the time to set on the fast clock, argument 2 is the clockspeed required, argument 3 is an instruction to show the clock.

Within the subroutine the first argument is represented by a marker %1, the second by %2, the third by %3 etc. You don't need to know this to use the supplied subroutines but you will need to understand it if you start writing your own subroutines.

Example of a subroutine call with a single argument (this must be "quoted" as the argument itself contains spaces):

```
call setroute "156 0,153 1,143 0,124 1"
```

This subroutine takes a list of switch numbers each followed by a space then the required throw position (switches delimited by commas). When the subroutine is called it quickly runs through the list setting all the switches to the required positions. Without a subroutine you would need a separate line in your script for each switch and position (e.g. THROW 156 0).

How to Do "*Almost*" Anything using the supplied set of Subroutines

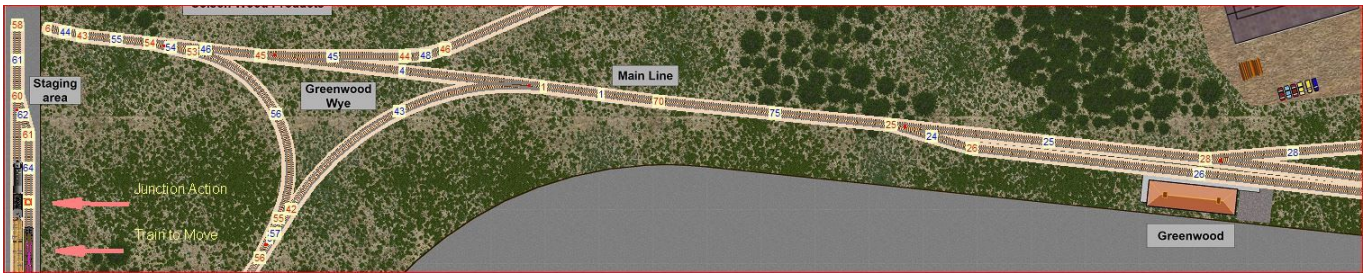
Well not quite everything, but if you need something that is not covered please let us know and we will try to accommodate you. The key point being that a subroutine in the Subroutine Library needs to be generic so it can be used on *any* layout. Subroutines that are layout specific are best stored within the layout folder or defined as a procedure within the rrw file (which is beyond the scope of this document).

For convenience the Subroutine Library is divided into themed subfolders to help you find a subroutine to suit your requirement, the subfolders can be browsed in the Subroutines tab of Script Central, each subroutine is heavily commented to explain its purpose. By all means study the comments carefully but please don't worry too much about the abstruse nature of some of the code.

That's enough for preamble, lets move on to explaining how to achieve some basic layout automation using the supplied subroutines.

PART 2 - THE BASICS, HOW TO GET YOUR TRAINS MOVING

Controlling Trains with Subroutines from the "Control" subfolder. *For clarity arguments are shown in blue ink.*



We will use this example to explain how to script the movement of the Doodlebug "Daily Mixed 14" from its position in the staging area over the first part of its run which ends in Greenwood station. The script to perform this task will be placed in a Junction Action depicted just above the train (at the bottom of track 64). This is Junction 71 though the number is obscured by the Junction Action marker circle.

First we need some code in the Master Script to *Call* a subroutine to move the train onto the Junction to collect its script.
call starttrain "Daily Mixed 14" F 5
This single line of code runs a subroutine to identify the train, set it to forward, and start it moving at 5 mph towards the Junction Action.
Note that the second argument **F** is for forward, in a different situation you may need to use **R** for reverse, or **T** for toggle direction.

Next we need some code in the Junction Action itself, first we need to stop the train to make it wait for its scheduled departure time.
stop
Now we need to place a timetable on the train, this becomes a user defined property of the train that can be accessed by the script.
call settimetable "Daily Mixed 14" "0830,0915,1630,1715"
The first argument is the train name, this can be the user name (as here), the engine ID, or $x_train (meaning the train owning the script).
The second argument is a comma separated list of departure times from: 1 Staging, 2 Greenwood, 3 Staging again, 4 Greenwood again.

The next subroutine call is used to hold up the script until the required departure time is reached, this requires no arguments.
call holdfortimetable
Note that the Fast Clock needs to be running for this to work, for details of how to set up and start the clock see the Clock section below.

When the departure time is reached we need to check that the route to Greenwood is clear.
call checktrackahead "62,61,44,55,54,46,4,1,75,24,26"
Here the string argument represents the track sections the train will move through as they are depicted in the Show Numbers view. If any of these tracks are obstructed by another train or a loose car, a message will appear in the schedule window to this effect and the train will not move until the obstruction is cleared. *If you like to live dangerously you can skip this subroutine call.*

When the script gets here the tracks have been found to be clear and we can start to move the train.
call followroute "60 0,54 0,45 0,1 0,25 0" F 30
The first argument is a sequence of switch numbers, a space, switch position required, comma and then the same data for the next switch.
The second argument is **F** which tells the train to move forward, you could use **R** for reverse, or **T** to toggle the direction setting.
The third argument is the required speed. Here the subroutine throws the first switch and starts the train moving forward at 30 mph
Should you need a change of speed on route you would only need to split this line into two separate CALL commands.
When the train reaches the first switch, it throws the second, when it reaches the second it throws the third etc.
The switch numbers are found by studying the "Show Numbers" view (see image above) and the switch positions are found by first setting the route manually and then reading the data from the tooltip while hovering over the switch.

Our automated train is now rolling towards Greenwood at 30 mph, throwing switches as it goes, we will need to stop it when it gets there.
call slowandstop 26 8 "(t26 j26 50)"
This slows the train to 8 mph at junction 26 and then stops it 50% along track 26 from junction 26 ; i.e. alongside the station building.
Note that the first junction (argument 1) must come after the last listed switch number in FollowRoute or this script will fail as FollowRoute doesn't return to the calling script until after the train crosses the final switch on the list.
**** reset mainline switch behind train**
throw 25 1 The mainline is reset by throwing switch 25 back to position 1 to protect the standing train and the script continues with:
call holdfortimetable
This waits for the second departure time (0915) and the script uses a similar sequence for controlling the next movement back to staging.
You can see the complete set of scripts for this layout if you download Bruno's ***"Greenwood_s"*** file from the TrainPlayer website.

Setting, starting and displaying the Fast Time Clock *For clarity arguments are shown in blue ink.*

The Clock subroutines are stored in the Clock subfolder, this is detailed fully in a separate section. However you will need to know how to set and start the clock to run trains to a timetable as shown above. So I have duplicated the details of the SetStartClock subroutine here.

call SetStartClock 0700 x6 S
Is a subroutine to Set and Start the fast time clock on layouts that need to have the clock running (e.g. for managing timetabled trains).
The first argument is the time to be set when the clock starts.
The second argument is an integer preceded by **x** to represent the clock speed required on the fast clock.
The third argument must be **S** to show the clock on screen while it is running or **H** to hide the clock from view.
This subroutine would typically be placed in the Master Script where it could be run automatically when the layout opens.

Time for a Short Primer on Variables

We have seen above how to pass an argument into a subroutine for processing, but to get some information back we need to use a variable, forgive me if I am teaching you to suck eggs but there may be some users who have no knowledge of the concepts of variables.

Variables can be best thought of as some sort of envelope, or little box, into which we pop a piece of information which we, or our train, will need later in the script. We don't need to remember, or even know, what the information is, but when writing the script we do need to know where we put it. So the variable is given a name we can refer to when we are ready to retrieve our information.

Variables are assigned their contents with the Set or Let statements. e.g. Let Box1 = J10. We can then use this information later by reading its contents. To extract the contents of a variable we use the symbol @. So to use the information we just stored in Box 1 as a place to stop our train we only need to say At @Box1 Stop (This is translated by TrainPlayer to mean At J10 Stop).

We can also pass the contents of a variable as an argument to a subroutine simply by prefixing it with the @ symbol and including it in the list of arguments passed to the subroutine @Box1. Similarly if we just pass a variable name to the subroutine Box1, our subroutine can put something in it for our script to read later with @Box1. You will quickly see that this allows our example to use the same line of code to stop different trains in different places just by allocating different Junction numbers to the variable dependent upon which train was detected.

Here endeth the short primer on variables. If you need more information then you will have to consult the TrainPlayer Manual.

Achieving more control from Subroutines in the "Control" subfolder. *For clarity arguments are shown in blue ink.*

call StopUntil Varname Value Speed
Stops and holds a train until a designated variable contains a specified value.
Argument 1 is the name of a variable (e.g. Signal1) which will be populated with a value by another script. *e.g. Set Signal1 Clear*
Argument 2 is the value that must be appear in the stipulated variable to allow the train to be released. Example Clear
Argument 3 is the Speed required when the train starts to move once the variable has been allocated the required value.

call setroute "60 0,54 0,45 0,1 0,25 0"
This subroutine call sets all the switches for a planned move in a single operation before the train starts moving, the syntax of the argument is exactly the same as that used for the FollowRoute subroutine referred to in the first section above.
The switch numbers come from the "Show Numbers" view and the positions are found by setting the route manually and reading the data from the tooltip while hovering over the switch. You will need to follow this subroutine call with separate instructions to move your train.

call ChangeDirection	Changes the direction of the train owning the script, F to R, or R to F". No arguments are required.
call Autofollow On **call Autofollow Off**	Switches Autofollow On for the selected train. Does nothing if already on. Argument must be On or Off Switches Autofollow Off for the selected train. Does nothing if already off. Note: You can select a different train to follow from within a script using **Train Trainname**
call FlipTrain	Picks up up the scripted train and turns it around so it faces in the opposite direction. Intended for use in staging areas or hidden track spurs. No arguments are required.
No argument is needed to flip the scripted train, however an optional "train name" argument is allowed to flip a different train.	
call ChangeCabs	For use with twin cab engines that have reached the end of their run. No arguments are required. The engine must first be detached from its train. Forward then becomes the opposite direction.
No argument is needed for the scripted train, however an optional "train name" argument is allowed to change cabs on another train.	
call RunningSounds On **call RunningSounds Off**	Switches the running sounds on. Switches the running sounds off.
call ReserveBlock train blockvarname Checks if the next notional block ahead is clear and if so reserves it by allocating the train name to it. The first argument, train name, can be a literal name, contents of a variable (e.g. @name), or $x_train for the scripted train. The second argument is the name or id number of the variable representing the block required. *A detailed discussion of Block Control is beyond the scope of this document but block variables are set and cleared by trains crossing junction actions. An example of scripting for Block Control can be found on my Rheilfford Bach demo layout.*	
call StopAfter junctionID	Scripted train passes specified junction, slows down and stops. Example **call stopafter** j10
call TurnAt junctionID	Scripted train passes specified junction, throws the switch and runs back through at 5 mph.
call LoadCarHere speed	Loads a car with its default load, but only if the car is travelling at (or less than) the speed stated. Can only be used from a Junction Action which should be set to be triggered by a car or an aar code. The single argument speed is an integer value. Example **call loadcarhere 4**
call UnloadCarHere speed	Unloads the triggering car but only if it is travelling at (or less than) the speed stated. Can only be used from a Junction Action which should be set to be triggered by a car or an aar code. The single argument speed is an integer value. Example **call unloadcarhere 4**
Call AlignTurntable turntableID trackID junctionID cw(or)ccw Unlike the Rotate command, this subroutine will not rotate the turntable bridge if is already in the correct position. The first argument is the turntable ID number as seen in the show numbers view. The second argument is the approach track ID number as seen in the show numbers view. The third argument is the connection junction ID number as seen in the show numbers view. The fourth argument is either cw for clockwise rotation or ccw for counter clockwise rotation.	

Manipulating Railroad Time with Subroutines from the "Clock" subfolder. *For clarity arguments are shown in blue ink.*

call SetStartClock 0700 x6 S
Is a subroutine to Set and Start the fast time clock on layouts that need to have the clock running (e.g. for managing timetabled trains).
The first argument is the time to be set when the clock starts.
The second argument is an integer preceded by **x** to represent the clock speed required on the fast clock.
The third argument must be **S** to show the clock on screen while it is running or **H** to hide the clock from view.
This subroutine would typically be placed in the Master Script where it could be run automatically when the layout opens.

call ShowClock	Displays the RR Fast Clock on the screen. No arguments needed. Does not start or reset the clock.
call HideClock	Hides the display of the Fast Clock. No arguments needed. Does not stop the clock running.
call SetRRTime 0700	Resets the time on the TrainPlayer fast clock. Argument can be in format 0800 or 08:00 To stop and start the clock from within a script use "**Clock Stop**" and "**Clock Start**"
call GetRRtime varname	Reads the current time from the RR fast clock and places it in a variable for display or processing.
call SetClockSpeed x10	Adjust the scale time running speed of the TrainPlayer fast clock.
call WaitForTime 1630	Stops the train that is running the script and holds it until the designated departure time is reached. WaitForTime requires a single argument, the time when the train should move on. The move will be in the same direction of travel as the arrival. The train will start at 5 mph, any other speed changes should be applied in the calling script.

call ElapsedFastTime starttime resultvar	Reads the current time from the Railroad Clock and calculates time elapsed since a given start time. First argument can be a literal time 0800 or 08:00, or contents of a variable @time Second argument is the name of a variable which is to accept the result for display or processing.
call ClockCalc basetime calctime resultvariable	Performs an arithmetical calculation on two stipulated times (hours and minutes). The first and second arguments must each be a valid time in the format hh:mm or hhm The second argument is added to the first, if this argument is negative it is subtracted. The third argument is a variable name which will accept the result for display or processing. *This routine can perform clock arithmetic across the 24 hour clock boundary (e.g 2300 + 0200 = 0100).*
call timedecho "text string"	Echoes the string to the Schedule Window with a time stamp prefix. Example output>> 16:30 - This is the string of text supplied as the argument.

Extracting Real Time Data with Subroutines from the "Clock" subfolder. *For clarity arguments are shown in blue ink.*

call GetTime varname	Reads the current time from the system clock and places it in a variable for display or processing.
call GetDate varname	Reads the current date from the system clock and places it in a variable for display or processing.
call GetTimeDate varname	Reads the current time and system date and places it into a variable as a string for display.
call ElapsedRealTime starttime varname	Reads the current time (real time) and calculates the time elapsed since a given start time. Stores the result of the calculation in a named variable for display or further processing. The first argument can be a literal string in the form hh:mm:ss, or the contents of a variable @var holding a string in the same format.
call ConvertTimeText varname	Takes a real time value from a named variable in the format hh:mm:ss or hh:mm Converts it to a text string suitable for a report and puts it in the same variable that supplied it.

Manipulating Scenery with Subroutines from the "Scenery" subfolder. *For clarity arguments are shown in blue ink.*

call HideScenery id	Hides a specified Scenery item to reveal the background image, argument must be a Scenery id number.
call ShowScenery id	Displays a previously hidden Scenery item over the background, argument must be a Scenery id number.
call ToggleScenery id	Toggles the Show/Hide status of a specified Scenery item, can achieve flashing effect if used in a loop.

PART 3 - THE REST IS JUST ICING ON THE CAKE

Managing Train Properties using the Subroutines in the "Train" subfolder. *For clarity arguments are shown in blue ink.*

call BuildTrainString train varname Builds a space delimited string of all the cars in the specified train for testing with the string features and stores it in the variable. First argument can be a train name, the id of any car in the train, or $x_train for the train owning the script. Second argument is a variable name which will take the returned string in the format "ES1 ET2 X10 X14 R12 N9" This result can be tested and compared with a predefined string to see if a user has completed a task to build a specified train.
call GetTrainName carID varname Gets the name of the train containing the specified car and stores it in the designated variable for display or processing. Argument 1 is the name of any car, contents of a variable @car, or $x_car for the name of the train owning the script. Argument 2 is the name of the variable that will be used to store the train name once it is found.
call GetSelectedTrainName varname Gets the name of the currently selected train and stores it in the specified variable for display or processing.
call RenameTrain currentname newname Renames the specified train with the newname given. Argument 1 can be the name of the train, the label of any car in the train; either literally, from a variable @var, or $x_train.
call RenameThisTrain newname Renames the train currently running the calling script to the newname given. Requires only the newname as an argument.
call SetTrainSpeed trainname speed Changes the speed setting on a specified train, not necessarily the scripted train.
call GetTrainSpeed trainname varname Gets the speed of the train named in argument one and stores the result in the variable named in argument two.
call SetTrainDirection trainname direction Change the direction setting of the specified train (not necessarily the scripted train). Argument one can be the train name, the label of any car in the train, or a name stored in a variable @train, or $x_train. Argument two must be one of **F** forward, **R** reverse or **T** toggle.

call GetTrainDirection trainname varname
Gets the current direction of the train named in argument one and stores the result in the variable named in argument two.

call GetTrainLength trainname varname
Gets the length (number of cars) of the train named in argument one and stores the result in the variable of argument two.

call GetTrainID carID varname
Gets the TrainPlayer generated ID of the train from the ID number of any car in the train.
Argument one can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script
Argument two is the name of a variable used to store the result for display or further processing.

call GetTrainID2 carID varname
Gets the TrainPlayer generated ID of the train from the ID number of any car in the train. (As above but different method).

call SetTrainProp trainID propname value
Adds a user defined property to a train, this remains with the train, even across linked layouts. It is also saved with the rrw file
Argument one is the train name, the name of any car in the train, the contents of a variable @train, or $x_train.
Argument two is a user defined name for the property (example Timetable).
Argument 3 is any numeric or string value. If argument three is omitted the property is removed from the train.

call GetTrainProp trainID propname varname
Gets the value from a user defined train property and stores the result in a variable for display or processing.
Argument one is the train name, the name of any car in the train, the contents of a variable @train, or $x_train.
Argument two is a user defined name for the property (example Timetable).
Argument 3 is the name of a variable used to store the result for display or further processing.
Note: User defined train properties can only be extracted through a script, they are not displayed in the Train Properties dialog.

Managing Car Properties with Subroutines from the "Car" subfolder. *For clarity arguments are shown in blue ink.*

call GetTrackID carID varname
Extracts the ID of the track segment the specified car is located on and places it in the variable provided in argument 2.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script

call GetLocation carID varname
Extracts the named Station location of the specified car to the variable. If not at a station the track ID number is returned.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script

call GetPosition carID varname
Extracts the exact position of the specified car to the variable in the format (t10 j11 50) .
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script

call SetCarLabel carID newlabel
Changes the label of the specified car to a new label.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script
Argument 2, the new label, can be a literal value, the contents of a variable @var, or data extracted from another function.

call GetCarLabel carID varname
Extracts the car label data (e.g HK121) from a specified car and places it in the variable name supplied by argument 2.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script

call GetCarLabel2 carnumber varname
Extracts the label of a car (e.g HK121) based on its actual position within the scripted train and places it in the named variable.
Argument 1, is the position of the car in the train running the script (the lead car is 1, the last car is equal to the total carcount)

call SetCarNote carID newnote
Writes the specified data directly into the Note field on the car. This can be seen in the Car Props dialog.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script.
Argument 2, the new Note, can be a literal value, the contents of a variable @var, or data extracted from another function.

call GetCarNote carID varname
Extracts the stored data from the Note field of a specified car and places it in a variable for display or processing.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script.

call GetCarAAR carID varname
Extracts the AAR code (e.g. HK) of the specified car and places it in the named variable for display or processing.

call GetCarID carID varname
Extracts the Car ID number (without the AAR prefix) of the specified car and places it in the variable for display or processing.

call GetCarType carID varname
Extracts the Type designation (e.g. 2-bay hopper) of the specified car and places it in the variable for display or processing.

call GetCarClass carID varname
Extracts the Car Class (e.g. 2-bay hopper) of the specified car and places it in the variable for display or processing.

call GetConsist carID varname
Extracts the consist type that contains the specified car. (e.g. Freight) and stores it in the named variable for processing.

call GetLoad carID varname
Extracts the loadname from the specified car if it is loaded and places it in the named variable for display or processing.
If the car is not loaded this subroutine returns Empty or Unloadable.

call SetLoadname carID newloadname
Changes the default loadname used on the specified car.
Argument 1, the carID can be a literal string, the contents of a variable @car, or $x_car if the call is from a Junction Script.
Argument 2, the new Loadname, can be a literal value, the contents of a variable @var, or data extracted from another function.

call GetLoadname carID varname
Extracts the default loadname from the specified car and places it in the named variable for display or processing.
The loadname will be found and reported whether or not the car is loaded.

call SetLoadStatus carID flag
Changes the load status on a specified car. Argument 2, the flag, must be 1 for Loaded or 0 for Unloaded.

call GetLoadStatus carID varname
Extracts the loaded flag from the specified car and places the result in the named variable for processing.
to identify whether or not the car is loaded. Variable becomes 1 if car is loaded otherwise 0.

call SetExOpsFlag carID flag
Used to exclude specified cars required for scripts from being selected by the Ops generator. Use 1 to set the flag, 0 to clear it.

call GetExOpsFlag carID varname
Used to extract the XO flag label from the specified car. Returns X if the flag is set, otherwise empty string.

call SetRevEngFlag carID flag
Sets or cancels the Reverse Engine flag on the specified engine/car. Flag must be 1 to set the flag, 0 to cancel the flag.

call GetRevEngFlag carID varname
Extracts the RevEngine flag from the specified car and store it in a variable for processing. Returns 1 if flag is set otherwise 0.

call GetSortOrder carID varname
Used to extract the Sort Order of the specified car in the car inventory and store it in a variable for processing.

call SetCarProp carID propname value
Adds a user defined property to a car, this remains with the car, even across linked layouts. It is also saved with the rrw file
Argument one is the carID, this can be a literal string, the contents of a variable @car, or $x_car if called from a Junction Action.
Argument two is a user defined name for the property (example ReturnEmptyTo).
Argument 3 is any numeric or string value. If argument three is omitted the property is removed from the car.

call GetCarProp carID propname varname
Gets the value from a user defined car property and stores the result in a variable for display or processing.
Argument one is the carID, this can be a literal string, the contents of a variable @car, or $x_car if called from a Junction Action.
Argument two is the user defined name for the property (example ReturnEmptyTo).
Argument 3 is the name of a variable used to store the result for display or further processing.
Note: User defined car properties can only be extracted through a script, they are not displayed in the Car Properties dialog.

Managing Layout Data and Menus using Subroutines from the "Layout" subfolder. *For clarity arguments are shown in blue ink.*

call GetLayoutName varname
Gets the name of the current layout in focus.
Requires one argument, the name of a variable to store the result for display or processing. e.g for adding into a filename.

call GetTrainCount varname
Obtains a count of the number of trains on the current layout. Each separate cut of cars will be counted as one train.
Requires one argument, the name of a variable to store the result for display or processing.

call RenameLayout newname
Can be used to rename the current layout from within any script on the layout.

call ShowControlPanel
Displays the original Train Control Panel on the screen if not currently visible. No arguments needed.

call HideControlPanel
Hides away the original Train Control Panel if visible. No arguments needed.

call ShowSwitchWindow
Opens the Switch Window if it is not already open. No arguments needed.

call HideSwitchWindow
Hides away the Switch Window if it is on display. No arguments needed.

call ShowTrainBar
Opens the Train Control Bar if it is not on display. No arguments needed.

call HideTrainBar
Hides away the Train Control Bar if it is on display. No arguments needed.

call ShowTrainTree
Displays the Train Tree window on the screen if not already open. No arguments needed.

call HideTrainTree
Closes down the Train Tree window if it is open. No arguments needed.

call ShowScriptCentral
Opens up the Script Central dialogue. No arguments needed. To close Script Central you must click on OK or Cancel.

Managing File Functions with Subroutines from the "File" subfolder. *For clarity arguments are shown in blue ink.*

call WriteFile filepath datastring
Is a subroutine to write a string of data to a file.
Argument 1 is the filename or filepath, this can be an explicit path or a relative path from the layout directory.
Argument 2 is the data string, this can be a literal string, or the contents of a variable.
If the file does not exist it will be created. If the file already exists it will be overwritten.
Direct filepath can use $Datadir to identify the Trainplayer Apps folder.
Example Direct Path: let filepath = $datadir + \subroutines\filename.txt call writefile @filepath @data
Example Relative Path: let filepath = .\filename.txt call writefile @filepath @data

call AppendData filepath datastring
Is a subroutine to append a string of data to the end of an existing file
Argument 1 is the filename or filepath, this can be an explicit path or a relative path from the layout directory.
Argument 2 is the data string, this can be a literal string, or the contents of a variable.
If the file does not exist it will be created.

call ReadFile filepath varname
Is used to read the complete contents of an existing file and store the result in a user variable for display or processing.
Argument 1 is the filename or filepath, this can be an explicit path or a relative path from the layout directory.
Argument 2 is the name of a variable which will be used to accept the data from the file, for display or further processing.

call GetLineByContent filename varname searchstring
Identifies and extracts the first line of a file that contains the specified sequence of characters anywhere within the line.
Argument 1 is the filename, Argument 2 is a variable to accept the extracted line, Argument 3 is the substring to search for.

call GetLineByStart filename varname searchstring
Extracts a specified line from a file based on the first few characters on the line. Returns the first matching line.
Argument 1 is the filename, Argument 2 is a variable to accept the extracted line, Argument 3 is the substring to search for.

call GetLineByEnd filename varname searchstring
Extracts a specified line from a file based on the the searchstring matching the characters at the end of the line.
Argument 1 is the filename, Argument 2 is a variable to accept the extracted line, Argument 3 is the substring to search for.

call GetLineByNumber filename varname linenumber
Extracts a specified line from a file based by on its line number position in the file.
Argument 1 is the filename, Argument 2 is a variable to accept the extracted line, Argument 3 is the line number to extract.
The line number can be a literal integer number or the contents of a variable in the form @variable.

call SaveVariables
Writes the current list of user global variables and values to a comma-delimited file in the same folder as the layout.
No arguments are required, the filename will be generated automatically based on the name and location of the layout.

call ReloadVariables
Retrieves a previously saved list of user variables and values from a comma-delimited file alongside the layout.
No arguments are required, the filename will be generated automatically based on the name and location of the layout.

Managing String Functions using the Subroutines in the "String" subfolder. *For clarity arguments are shown in blue ink.*

call CompareString firststring secondstring varname
Is used to compare two strings and check to see if they are an exact match.
Either string can be a literal string or the contents of a variable @string, or a function call returning a string.
If the strings match 1 is placed in variable, if they don't match 0 is placed in variable. This can then be tested with IF.
Typical use would be to test a string from BuildTrainString with a predefined String to check if a train was correctly assembled.

call ConcatString varname "text to be added"
Is a procedure to append characters, words, car IDs or other text to the value in an existing string variable.
Argument 1 is a variable name which contains some existing text.
Argument 2 is the text to be added, this can be a literal string in quotes, or contents of a variable @variable

call ConcatNumbers varname numericstring

Is a subroutine to concatenate two numeric variable values into a string as opposed to performing normal arithmetic.
Argument 1 is a variable name which contains an existing numeric string. e.g. 12345
Argument 2 is the numeric string added, this can be a literal string in quotes "67890" or contents of a variable @variable

call GetFixedWord string field wordlen varname

Is for extracting fixed length words from a delimited text string with consistent word lengths. It requires four arguments.
Argument 1: The string can be a literal string "0010,0120,0230,0340,0450,0500,0600", or the contents of a variable @string.
Argument 2: The fieldnumber required (i.e. the position of the word in the string).
Argument 3: The word length required (excluding the delimiter character, the example above is word length 4).
Argument 4: The name of a variable used to accept the extracted data for display or further processing.

call GetNext stringvarname resultvarname optionaldelimiter

Is a subroutine to take the next token (or field) from a delimited string for processing, the third argument "delimiter" is optional.
Argument 1 is the name of the variable containing the string to be accessed.
Argument 2 is the name of the variable which is to accept the extracted data (or token).
An optional third argument can be used to specify the delimiter character.
The delimiter character is assumed to be either a comma or a blank by default unless there is a specified third argument.
If the third argument is absent the list is delimited at both commas and spaces.
The third argument for Delimiter can be any character (including space) but must be enclosed in quotes. e.g. ";" or "/="
Each time GetNext takes the next token, both the token and next delimiter are removed from the original string.
Removing the entry from the original string ensures that the next listed item will be found when the call is made again.
This allows you to use a string like a set of sequenced instructions for tasks like reading timetables in sequence.
Two adjacent delimiters in a string with no intervening text are treated as a single delimiter.
Example calls: **CALL getnext stringname resultvar** or **CALL getnext stringname resultvar ";"**

call GetWord stringvarname resultvarname entryposition optionaldelimiter

Is a subroutine to extract a field or token from a delimited string based on its position in the string
Unlike GetNext, this subroutine does not remove the extracted data from the original string.
Argument 1 is the name of the variable containing the string to be accessed.
Argument 2 is the name of the variable which is to accept the extracted data (or token).
Argument 3 is the position of the entry to be extracted (i.e. an integer number).
An optional fourth argument can be used to specify the delimiter character use in the stringvariable (must be in quotes ",").
The delimiter character is assumed to be either a comma or a blank by default unless specified as a third argument.

call FindSubString string substring varname

Is a subroutine to search a string for a specific substring and return its 0 based start position in a variable for further processing.
Argument one can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument two can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument three is a variable name to accept an integer number giving the start position of the substring for further processing.

call StringContains string substring varname

Is a subroutine to check if the required substring is contained anywhere within the specified string.
Argument one can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument two can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument three is a variable name which will accept the result of the call. This will be 1 if the substring is found, 0 if not found.

call StringStarts string substring varname

Is a subroutine to check if the required substring is at the start of a specified string.
Argument one can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument two can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument three is a variable name which will accept the result of the call. This will be 1 if the substring is found, 0 if not found.

call StringEnds string substring varname

Is a subroutine to check if the required substring is at the end of the specified string.
Argument one can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument two can be a literal string or the contents of a variable @string, or a function call returning a string.
Argument three is a variable name which will accept the result of the call. This will be 1 if the substring is found, 0 if not found.

Communicating with users using Subroutines from the "Interface" subfolder. *For clarity arguments are shown in blue ink.*

call GetKey keycodelist varname

GetKey is a subroutine to hold up parsing a script until any one of a list of specified keys has been pressed.
The calling script (not the train) will pause at this point until a valid selection is made.
Argument one is a comma delimited list of key codes in quotes. Example: "112,113,114,115,116,117,118,119,120"
Argument two is the name of a variable which is to accept the position of the key selected as it appears in the list.
In the example above 113 is the code for the F2 key, if this is pressed the value returned will be 2 (second choice on the list).
You should precede calls to this subroutine with an appropriate prompt or menu in a Note Window or in the Schedule Window.
The value returned can then be used in a conditional IF structure to initiate the required action.

call CheckKey keycode varname

CheckKey is a subroutine that can be placed within any "looping" script. It requires two arguments.
It is used to check if a user has pressed a specified key since the routine was last "Called" in a previous iteration of the loop.
This has the advantage over the ON KEY command in that the script does not Stop and Wait for the required key press.
This means the check can be made without losing control of any other actions being managed by the script.
Argument one is the key code being monitored for.
Argument two is the name of a variable that will receive the key value if it has been pressed since the last check.
If the required key has been pressed, the value of $key is reset to zero so that further checks can be made by the script.
A typical call to this subroutine would occur from within a larger While loop and take the form:

CALL CheckKey 65 var (Check to see if key A [code 65] has been pressed since the last check)
Followed by a line to perform some action If (@var = 65) etc. etc.

call SetUpNote coordinatestring type textalign

SetUpNote is a subroutine to set the position and size of the Note Window
Typical use would be to modify the size and position for a different type of note while running a script.
Argument one is a quoted string specifying the position of the note window within the application window. e.g. "0,0,400,300"
Argument two specifies if the vertical size of the window should be fixed or adjustable to the text (0 = Fixed, 1 = Adjustable)
Argument three specifies the required text alignment and must be one of **L C** of **R** (Left aligned, Centred or Right aligned).
Example call: **CALL setupnote "0,0,400,300" 0 C**

The co-ordinates represent "top,left,bottom,right" position of the Note Window. within the TP application window.
If argument two is set to **1** the top and bottom will adjust themselves dependent on the number of lines in the text.

This subroutine call would normally be followed by a NOTE command to display the required information in the defined window.

call Message messagestring

Message displays a message on the layout in a windows type dialogue box with a single OK Button.
The call requires a single argument of a text string to display in the message box
The text to be displayed can be a literal string in quotes, or the contents of a multiline variable.
Single line strings can be split across two lines using the \n character.
Note: This feature freezes all other action on the layout until a button is clicked.

call MessageOC messagestring

MessageOC is the same as Message but has the the option of OK or Cancel Buttons.

call MessageYN messagestring

MessageYN is the same as Message but allows a question to be asked which can be answered with Yes and No Buttons.

call SysCommand commandstring

SysCommand allows you to execute any command that you can type into the system command prompt box.
The single argument must be a valid command string, this can be a literal string or the contents of a variable @variable
Example: CALL SysCommand "Type myfile.txt & Pause"
Example: CALL SysCommand "rename abc.txt def.txt"
Note that in most versions of Windows the Command box closes immediately.
If you want time to read the output from the command window you should append "& pause" to your command line.

Setting Default Values using Subroutines from the "Settings" subfolder. *For clarity arguments are shown in blue ink.*

call BigCarLabels

Is a subroutine to increase the label size on cars for use in shunting puzzles
The subroutine does not require an argument, but will accept a single argument Reset to return to the users default size.
Example: **CALL BigCarLabels** or **CALL BigCarLabels Reset**

call SetGauge gauge

Is a subroutine for modifying the rail spacing, tie dimensions and roadbed width for different track gauges.
Typically this would be used in the Master Script to set up the layout as soon as it loaded.
The subroutine takes a single argument, one digit representing feet or two digits representing feet and inches
The argument must be a single number fom the following list 48, 13, 2, 24, 26, 3, 36, 5, 53, 7 or M. or the word Reset.
Note that single digits represent feet, double digits such as 48 and 26 represent feet then inches, M is for Metre gauge.
Use Reset to retrieve your original registry defaults. *Note: If you prefer you can use the syntax **call SetGage gage***

call SetValue regvar value

Can be used to modify the values of any of the temporary registry variables
Argument one is the name of the temporary registry variable, argument two is the value required.
You can also set these values by just using the Set command (which is actually easier than messing with these functions).
CALL setvalue ClosedJxnColor 167711680 (changes the closed light color to blue as does **Set ClosedJxnColor 167711680**).

call SetSpeeds maxMPH couplingspeed accelfactor

Is a subroutine to set the maxMPH, the coupling speed, and the acceleration factor from within a script.
Argument one is an integer number representing the maximum speed setting (in MPH) for the train controller.
Argument two is the maximum permissible coupling speed on this layout. Note this should be set 1 mph higher to avoid crashes.
Argument three is the acceleration factor for the trains. Example call: **CALL SetSpeeds 40 6 15**
Optionally you can call this routine with a single argument Reset to restore your settings to the stored registry values.

call SetCrashMode On

Is a subroutine to switch on or off the crash mode settings. Requires one argument which must be **On** or **Off** or **Reset**
When set to On trains will crash when crossing closed switches, running off spurs or exceeding specified coupling speeds.
When set to Off trains will barrel thru and throw closed switches and bounce at the end of spurs.
When set to Reset the users usual settings will be restored from the registry.

call ResetAll

Can be used to reset all the temporary variables extracted from the registry back to the individual users normal defaults.
The subroutine requires no arguments, the syntax is simply **CALL resetall**

Monitoring User Operated Trains with Subroutines from the "Monitoring" subfolder. *For clarity arguments are in blue ink.*

call TrackOccBy trackID varname

Is used to obtain a space delimited list of cars to be found on a designated track section in the format ES1 ET2 X10 X14 R12 N9
Argument one is the ID number of the track to be checked (as seen in the show numbers view), literal number or @variable.
Argument two is the name of a variable designated to receive the list of cars for processing with the **CompareString** subroutine.
Example: **CALL trackoccby 45 carlist** will populate the variable named carlist with a list of cars found on track 45.

call StationOccBy stationID varname

Is used to obtain a list of cars to be found within a defined station. Argument 1 can be station ID number or station Name.

call TurntableOccBy turntableID varname

Is used to obtain a car name or list of cars to be found on a turntable bridge track. Details otherwise as for TrackOccBy above.

call CheckConsist trainname carstring varname

Compares the make up a particular train with a predfined check string.
Argument 1, name of the train to check, can be a literal name, a name in a variable @train, or train owning the script $x_train.
Argument 2, list of a string of cars which the car order of the train has to match exactly; can be literal string or @variable.
Argument 3, a variable containing the word **No**, when the consist matches the string in argument 2 the variable is set to **Yes**.
Example script to demonstrate the use of this subroutine:

```
Let Done = No
While (Done = No)
    Call CheckConsist $train "ES37 ET38 H39 H40 H41 H42 H43 H44 H45 H46 H47 H48 H49 H50 H51 H52 H53" Done
    .... additional checks or instructions for the designated task can go here ....
endwhile
```

call CheckSwitch trainname switchID positionreqd ab

Is for use in an interactive script where the operator is driving a train to scripted instructions.
The instructions appear in a Note window which should first be set using **Call SetUpNote "169,109,229,490" 0 L**
Argument 1: The name of the train, can be a literal name, contents of a variable @train, or the train running the script $x_train
Argument 2: The switch ID number. An integer number which can optionally be preceded by a **J**
Argument 3: The position required (usually **0** or **1**)
Argument: 4: Either **A** or **B**, to signify if it is the switch **A**head of the train or the switch **B**ehind the train that is to be thrown.
Example call: **call CheckSwitch @T1 138 1 A**

call WaitForDirection direction

Holds up the processing of a script until the scripted train is set to the specified direction on the train control bar.
Requires a single argument which must be either **F** or **R** (Forward or Reverse). Example: **CALL WaitForDirection F**

call WaitForLength length

Holds up the processing of a script until the scripted train is the required length.
This is used to monitor train length to ensure required switching movements have been completed.
Requires a single argument of an integer number representing the number of cars that should be in the train (including engine).

call WaitTilMoving

Holds up the processing of a script until the scripted train starts moving under operator control. Requires no parameters.

call WaitTilStopped

Holds up the processing of a script until the scripted train is stopped under operator control. Requires no parameters.

call WaitForConsist consist

Holds up the processing of a script until the cars and order of the scripted train matches a specified consist.
Used to monitor train composition to ensure any required switching movements have been completed.
The argument should be a space delimited string of car ID numbers in the form ES37 ET38 H39 H40 H41 H42 H51 H52 N10
This can be passed as a literal string in quotation marks, or as the contents of a variable @variable.
Example: **CALL WaitForConsist "ES37 ET38 H39 H40 H41 H42 H43 H50 H51 H52 N10"**
The script will not progress until the make up of the scripted train exactly matches the supplied string.

call WaitUntil trainname arg2 arg3 arg4

Holds up the execution of a script until all of the stipulated train conditions have been met.
The subroutine is designed to monitor the condition of a selected train which is being controlled by the operator
It enables a script to be paused until the length, direction and speed of the train meet the required conditions

This does not prevent the user from moving the train, but it does pause the script and prevent it from running on ahead.
Argument 1: Name of train for testing. (Can be passed literally, as contents of a variable, or as $x_train).
Argument 2: Must be **S** if instructions require a moving train to Stop, or **M** if they require a standing train to start Moving again.
Argument 3: Must be **F** or R, i.e. the direction setting needed on the Train Controller before allowing the script to continue.
Argument 4: Train length (car count) needed before allowing script to continue, used to check couple and uncouple events.
Example: **call WaitUntil @mty M F 5** (Where mty is the name of a variable representing the block containing the train name)
This might follow a Note instruction to Start a standing train only if it is set to go forward and only if it is five cars long.

Producing Reports using Subroutines from the "Reports" subfolder. *For clarity arguments are shown in blue ink.*

call wheel_report_modern

Wheel_Report_Modern is intended to simulate compilation of a wheel report in the Modern Era where reports
are generated by computer from data collected by ACI (automatic car identification), bar codes or RFID.
This subroutine does not require the train to be travelling at the slow speeds required by the Wheel_Report_Steam routine.
To call the subroutine use the syntax "call wheel_report_modern", note that no parameters/arguments are needed.
The subroutine can be called from either a Junction Action Script or a Train Script
If a Junction Action is used the subroutine can be triggered by "Any Last Car" (more realistic) or "Any Train".
If called from a Train Script it can be set to Trigger by either AT <jxn> or AFTER <jxn>
The subroutine produces a Wheel Report of the train crossing the Junction and writes this to a file in the subroutine folder
The filename will be prefixed WR_ and will include the name of the layout and the name of the train.
Each car entry will be in the form: AAR, CAR_ID, LODNAME (or MT), plus any information from the CAR NOTE property field.
This enables extraction of any data in the carnote field of the car properties dialogue (e.g destination on another layout).

call wheel_report_steam

Wheel_Report_Steam is intended to simulate compilation of a wheel report in the steam era where
reports were compiled at a register station by an observer monitoring the passage of a slowly moving train.
In this version of the Wheel_Report the cars are individually checked and the data written as each car passes the junction.
The routine requires you to maintain slow train speeds past the register station as it is called repeatedly by each car.
Please be aware that the processer overhead required for this may tax the capability of some older computers.
In this event we would suggest you use the alternative "Wheel_Report_Modern" subroutine which produces identical output.
This subroutine MUST be called from a Junction Action Script, it is not suitable for calling from a Train Script.
The Junction Action must be set to be triggered by "Any Car", the direction setting is up to the user.
To call this subroutine use the syntax "**call wheel_report_steam**", note that no parameters/arguments are needed.
The subroutine produces a Wheel Report of the train crossing the Junction and writes this to a file in the subroutine folder
The filename will be prefixed WR_ and will include the name of the layout and the name of the train.
Each car entry will be in the form: AAR, CAR_ID, LODNAME (or MT), plus any information from the CAR NOTE property field.
This enables extraction of any data in the carnote field of the car properties dialogue (e.g destination on another layout).

Richard Fletcher, November 2014