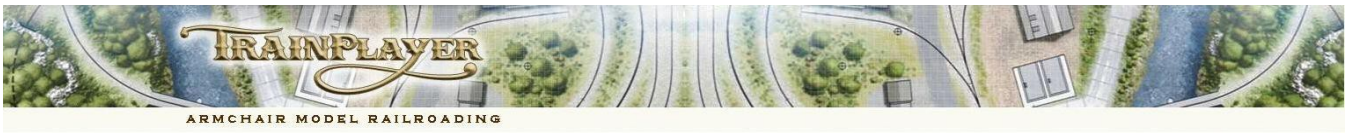

A Guide to the TrainPlayer Programming Language (TPL)

It is intended that this document will assist an inexperienced scripter to familiarize themselves with the syntax used in the various TPL statements. The full set of TPL statements can be used in any of the four script types (Master Script, Train Script, Subroutine or Junction Action) which were described in an earlier document ***TPL_Ref_ Doc_1*** *- An introduction to the TrainPlayer Programming Language (TPL).*

Example script segments have been included where it is felt these will help to clarify the purpose of a particular statement, the comments prefixed with ** are not essential components of the script. A concise list of all the TPL scripting statements is included at Appendix A.

Part 1 - Key Information needed to Start Scripting

Blank Lines and Comment Lines

Spaces before and after TPL statements and blank lines within a script are ignored during processing but can be used to advantage to break up and indent the code into manageable segments, thereby improving the appearance and readability of the script..

When you first open a new Train Script dialogue, Junction Action dialogue or create a new Master Script in Script Central, the first thing you will see is one or two automatically generated Comments preceded by an * asterisk. Although comments are not essential components for script processing it is recommended that you make liberal use of these to remind yourself of the purpose of each part of your script.

Any text following an * or // or a # followed by at least one space is treated as a comment and ignored when processing the script.

By default these comments will echo to the Status Bar and the Schedule Window, and while this can be useful during program development, the option can be switched off by right clicking in the Schedule Window and changing its settings.

Comments starting with ** plus one space, instead of just a single *, will not echo to the schedule window in any circumstances and may be preferable in scripts intended for distribution or publication to the web chooser.

Wait Conditions

Most TPL commands are preceded by a wait condition, the wait condition and command can both be entered on the same line, or the wait condition can be on one line with the subsequent command beneath it. A command without any wait condition executes immediately and a wait condition doesn't necessarily have to be followed by a command.

```
** Example TPL code demonstrating Wait Conditions combined with Commands on the last two lines
** Here the train will start moving forward at 10 mph until it reaches junction 123 when it will stop
** It will wait for five seconds at J123 before moving off again at 10 mph
   Forward
   Speed 10
   AT J123 STOP
   AFTER 0:0:05 SPEED 10
```

Although Wait Conditions may be followed on the same line by a Command Statement, a TPL statement normally ends with a line break, a semicolon (;) to separate two statements on the same line, or a * character which signifies that the rest of the line is a comment.

It is important to be aware that a Wait Condition refers to making the Script wait for the required condition to be met, it is not an instruction for the train to wait. In the event that we also want the train to stop and wait (e.g. when waiting for ON THROW before moving) then we must ensure we script the train to STOP before the script reaches the wait condition.

Particular care is needed when using AT and AFTER close together within a script, if the train has already fulfilled the condition for the second of these statements before the condition for the first is met, the script will fail because it cannot try to detect the second condition until after it has processed the first. For example if a script requires the last car of a train to trigger an AFTER <jxn> command before the first car triggers an AT <jxn> command and the train is longer than the distance between the two junctions - then the second command will fail because the front of the train will have already passed the junction when the script starts waiting for it to do so.

Wait Conditions Syntax *(Note: TPL Statements are not case sensitive, you can use AT, at, AT or even aT, use whatever you are most comfortable with.)*	
AT <jxn>	script waits until midpoint of lead car crosses junction (jxn can be a number only or include a j prefix)
AT <t j d>	script waits until midpoint of lead car crosses (% dist d from jxn j on track t) (In this case j cannot have a j prefix)
AFTER <jxn>	script waits until midpoint of last car crosses junction (jxn can include the optional j prefix)
AFTER <(t j d)>	script waits until midpoint of last car crosses (% dist d from jxn j on track t) (In this case j cannot have a j prefix)
AT <station>	script waits for lead car of train to enter the area defined as a station ("quotes are needed for station names with embedded spaces")
AFTER <station>	script waits until last car of train leaves the area defined as a named station ("quotes are needed for station names with embedded spaces")
AT <h:m[:s]>	script waits until specified time is reached on the layout clock; h:m required, secs optional, layout clock must be operating to use this
AFTER <h:m:s>	script waits until specified time has elapsed on the real time clock (not the layout clock), seconds must be included
ON STOP	script waits until the train comes to a complete stop, this has no effect if the train is already stationary
ON COUPLE	script waits until train couples with another car, script commands then operate the longer train
ON THROW <jxn>	script waits until the specified switch is thrown by any means (i.e. by the operator or by another script) (if the <jxn> is to be thrown only by a script command then any jxn reference can be used, even if the specified jxn it is not a switch)
ON TABLESTOP	script waits until turntable finishes rotating (used after a rotate command, see separate section on Turntables below)
ON KEY [<key>]	script waits until the user presses the specified key. e.g. `On Key F1, On Key A,` if <key> omitted then any key press is accepted On Key should be preceded by a prompt in an echo or note command to explain the action required.

Commands used for Train Control	
Forward	sets direction to forward, has no effect if engine is already set to forward, command is affected by Autopause (see below)
Reverse	sets direction to reverse, has no effect if engine is already set to reverse, command is affected by Autopause (see below)
Speed <mph>	sets speed to given mph (kph if metric); starts train moving if it is stopped.
Stop	decelerates the scripted train to a stop, command is affected by Autopause (see below)
Horn [<msec>]	sounds train horn for 3 secs or for stipulated time in millisecs (uses the horn sound already attached to the train)
Autopause <secs>	turns on pauses of <secs> duration between certain commands during train movements, use Autopause 0 to cancel Autopause only affects Forward, Reverse, Stop and Uncouple
Uncouple <slot>	<slot> is an integer number, 1=behind 1st car, 2=behind 2nd etc., uncouple is affected by Autopause (see above), slot is calculated from head end of train, if train is in reverse this is not the engine and slot can be unpredictable, better to use uncouple carID if ID is known
Uncouple <carID>	uncouple specified car from the portion of train which has the engine attached, uncouple is affected by Autopause (see above)
Uncouple <carID><carID>	uncouples between the specified cars, uncouple is affected by Autopause (see above)

```
** Example showing Wait Conditions used with Commands
   on couple stop            ** Stop the reversing train when it couples with another car
   forward                   ** Set train ready to move forward
   uncouple 5                ** Uncouple at the fifth slot (to retain five cars including engine)
   speed 6                   ** Start the train moving forward at 6 mph
   after j33                 ** Pause the script until the center of the last car has reached j33
   after 0:0:02 stop         ** Allow the train two more seconds to clear the junction and stop
   reverse                   ** Set the train ready to move in reverse
   throw j33 1               ** Throw the switch immediately behind the train to open the siding
   speed 6                   ** Start the train moving in reverse into the siding
   at j695 stop              ** When the lead car reaches junction 695 stop
   forward                   ** Set the train ready to move forward
   uncouple 1                ** Uncouple immediately behind the engine to leave the cut in the siding
   speed 6                   ** Move forward (ready to reverse back after junction for rest of train)
```

Commands used for Layout Control	
Throw <jxn> [<pos>]	throws switch to stated position or to the next available position if not stated, <pos> is 0 or 1 (or 2 for 3 way switch) (jxn can be a number only or include a j prefix, e.g. Throw 123 0; or Throw J123 0)
Load [Toggle] Train [name]	loads all cars in the specified train with their default loads, if optional name is absent the current train is assumed, if optional keyword toggle is present, the loads are toggled.
Load [Toggle] Car <ids> [<loadname>]	loads the specified car <ids> which can be a comma-delimited list of car ids with the specified loadname or with their default loads if loadname not present, if keyword toggle is present, toggle the loads. Cars can be anywhere on the layout.
Load [Toggle] Cut <ids> [<loadname>]	loads the cut of identical car types around the specified car with the specified loadname, or with their default loads if the loadname is not present, if keyword toggle is present, toggle the loads. Specified cut can be anywhere on the layout.
Unload [Toggle] Car/Train/Cut <ids>	unloads specified car(s), same syntax used as in load
Sound <spath>	plays the wav file from spath once only. spath is either a full pathname, or a path relative to the TP Sounds folder.
Sound loop <spath>	plays the wav file from spath repeatedly until told to stop. If you put a comment on the same line you need a ";" after filename.
Sound stop <spath>	stops the repeated playing of the wav file specified. If you put a comment on the same line you need a ";" after filename.

```
** Examples of the use of the load and unload commands
   load toggle car X44,H22,H24          ** Toggles the load status of the four listed cars
   load car H33                         ** Loads the car H33 with its default load
   load car H33,H35,H36 aggregates      ** Loads the specified list of cars with the load "aggregates"
   load cut X14                         ** Loads cut of identical cars around car X14 with default load
   unload train                         ** Unloads all cars in the train running the script

** Example of the Sound Command:plays the specified sound file continuously for 3 seconds and stops
   SOUND LOOP Loco\Heavy freight.wav;    ** Starts to continuously play the specified sound
   after 0:0:03                         ** SOUND commands can't have comment unless ; delimits filename
   SOUND STOP Loco\Heavy freight.wav;    ** Stops the Sound loop from playing, filename can be omitted
```

Additional Train Control Commands	
Train <train>	Selects the named train as the train of focus in the train window, (for <train> use "name", ID or label of any car in the train). Note: this only changes the train window focus, it does **not** make the named train run this script unless it is already running it. In any place a train needs identifying TPL will accept the train name, "Train" + ID, or the ID label of any car in the train.
Start <train> [<mph>]	If the specified train has no script it will start moving at the given speed in whatever direction it is facing (5 mph if speed is omitted). if the specified train has a script the script is started, (or rewound and started so caution is needed if that train has already been moved) Note: The statements following the Start command continue to affect the current scripted train, not the one you have just started.
Drive <train>	Transfers control temporarily to specified train, subsequent commands apply to that train until an Enddrive command is reached. Statements in the drive block are packaged as a little script, which is attached to the target train and started. Control then returns immediately to the calling train, executing the statement after ENDDRIVE.
Enddrive	Transfers control back to the original train after attaching code to another train from within a "Drive; .....; Enddrive" block. Note: The Drive; ....; Enddrive block also allows Train Control commands to be used in a Master Script which is not attached to any train.

```
** Example using the Train command to change focus of the train connected to the controller
   Train "Stock Hauler"    ** Changes focus of the train window and train controller to the "Stock Hauler"

** Example of using Start to start another train running that is not connected to this script
   Start "Jim Dill" 10  ** Starts the named train at 10 mph, if it has a script this is rewound/restarted.
```

```
** Example showing the use of the "Drive; ...; Enddrive" block which directs commands to a different train
** The command "Drive" is followed by a train identifier (name as found on the train menu)
** Assume our script is operating a freight train and we want to start another train moving
   speed 15       ** this is a command sent to the train which is attached to this script
   drive Train28  ** the following script statements shown in italics are attached to Train28 and started
      speed 20
      after 0:0:04 reverse
      stop
    enddrive      ** original script continues without waiting for the drive block script to complete
   AT J123 Speed 10  ** This command applies to the original train (the one controlled by this script)
** In principle you should be able to nest one "drive" block within a another should you wish to do so.
```

Commands used for Turntable Control	
Rotate <ttbl> <jxn> [cw/ccw]	rotates the turntable to the specified junction on its rim, <ttble> is ID # of turntable, <jxn> is the ID of the junction, cw specifies clockwise which is the default and can be omitted, ccw specifies rotation will be counter clockwise
Tablestop	stops turntable rotation
ON TABLESTOP	Wait Condition to hold up running the script until the turntable finishes rotating (can only be used after a rotate command)

```
** Example turntable script, for an engine entering track 376 which is on turntable 377
** Stop engine in center of turntable track (i.e 50% along track 376 from the end marked as junction 374)
   at (376 374 50) stop   ** wait condition to hold up script processing until engine is on turntable
   on stop                ** wait condition to prevent turntable rotation until engine has stopped
   rotate 377 279 ccw     ** rotate turntable 377 to junction 279 in a counter clockwise direction
   on tablestop           ** wait condition to prevent engine starting until rotation has stopped
   Forward speed 15       ** command to drive train off turntable and continue with script
```

Note: Turntable Scripts prepared with the Script Recorder will need editing to insert the ccw attribute if needed, this is because mouse clicks on the turntable rim always cause the turntable to take the shortest route, whereas on playback the script will always default to clockwise rotation unless counter clockwise (ccw) is specified in the script.

User Interface Commands	
Echo <display_string>	Echo displays the given "display string" in the schedule window and on the status bar. (see definition of a "display string" on page 4) All display strings may contain substitutable variables prefixed by $, @ or @@ (see section on variable substitution on page 6).
Note [<display_string>]	Note displays a given "display string" in a popup window; or hides the window if no "display string" argument is given. Note has the big advantage over Input and $msgbox in that it does not stop the action on the layout while awaiting a user response. Only one Note can be displayed at a time, if a note is open when another is called the first one will be overwritten *(this can be useful)*. The style and position of a Note window can be modified, repositioned and resized (Right-click and choose Properties to open the note in the scenery object dialog, where changes can be made to the background color, font, etc. After saving the layout, the next time the script is run, the window will appear in the position it was in when the layout was saved.) There can only be one Note window open at any time but its size and position can be modified explicitly from within a script by resetting the temporary registry variables containing the information about the positioning of the Note. (See Registry Variables section on page 6)
$MSGBOX([type,]<text>)	$msgbox is a system function but as it is used for conversing with and soliciting responses from the user we have also included it here. Displays the given <text> as a prompt, Type is optional and if omitted only an OK button is displayed. If "type" is present it must be OC for a box with OK/Cancel buttons, or YN for a box with Yes/No buttons. The function returns 1 if ok or yes is clicked, returns 0 if no or cancel is clicked (the response can be used for subsequent conditional processing). *Note: This command is best used at start up as it stops all train movements while the function is awaiting the user's response.*
Input <var> [<prompt>]	Input sets the value of a user variable (see below) from an input dialog box by displaying a prompt dialog and waiting for a response. The result is copied into the specified variable. If the variable does not already exist it will be created. You may supply your own prompt string; if this is omitted a default string is used. Input is like a Wait Condition, the script holds and waits for a response, the next command does not execute until the user hits OK or Cancel *Note: This command is best used at layout start up as it stops all train movements while the function is awaiting the user's response.*

```
** Example of using Echo for passing information to the Schedule Window
   At J123
   Echo The train has reached Junction 123

** Example of using Note to obtain a simple user response
   After 123 STOP
   Note "The train has now stopped
Press a key when ready to start"      ** See following section for info on multi-line display strings.
   On Key     ** Script waits for the user to press a key on the keyboard (any key will suffice)
   FORWARD Speed 10

** An introductory note in a Master Script extending over three lines (here quotes are essential).
   Note "RHEILFFORD BACH
Welsh Narrow Gauge
Press S to start the trains"
   On Key S    ** Script waits for the user to press S on the keyboard, other keys will be ignored
   Note        ** Using Note with no text closes down and hides the Note window

** Example of using Input to obtain a value for a variable from the user
   Input myvariable "Enter the ID of the car you want to set down"
...Uncouple @myvariable  ** See section on variables for information about @

** Examples of using $msgbox
   call $msgbox(This is a note!, all trains are stopped, click OK to continue)
   let v1 = $msgbox(YN, Want to continue?)  ** makes the value of v1=1 if Yes is clicked, or 0 if No
```

Rules for using Display Strings **Display Strings can be used with Echo, Note, and $msgbox to present information to the user, and with the $write file function**	
Single Line Display Strings	May be enclosed in quotation marks but need not be. Can include \n (backslash n) to force a new line with Note, $msgbox and $write, but not with echo. Can include embedded information in the form of substitutable variables represented by $, @ or @@ Quotation marks can be included in display strings by doubling them, e.g. "" reproduces as ".
Multiple Line Display Strings	Must always be enclosed in quotation marks. Can include embedded information in the form of substitutable variables represented by $, @ or @@ Quotation marks can be included in display strings by doubling them, e.g. "" reproduces as ".
Where Quotation Marks are Mandatory.	1. Around a multi-word command, registry setting name, or filename -- that is any reference name that contains a space. 2. In a menu command where the item to be selected contains a space, this applies only to a single level of a cascading menu. 3. Around a string containing a parsable character, where the meaning of that depends on the context. In a function call, an unquoted string must not contain the character ")", because that would terminate the call to the function. In a comma-delimited list, a string must be quoted if it contains a comma, otherwise the list would terminate at the first comma. 4. Around a multi-line text block. This is the only way to define such a block, by enclosing it in double quotes and including carriage returns:

```
** Example of a single line string, either of these is valid:
   echo this is a string to be output
   echo "this is a string to be output"

** In a display string, variables are substituted whether or not the string is quoted.
   echo the speed of the crossing train is $train($x_train, speed)
   echo "today is $date and it is already $time"

** Quotes are optional around most function arguments and keywords (both of these are valid)
   echo the speed is $train(train66, speed)
   echo the speed is $train("train66", "speed")

** Examples where quotes are essential:
   file open "c:\my file.txt"       ** filename with space
   file "revert to saved"           ** menu command
   set "Default Car Length" 400     ** reg setting name

** String in a function call which includes a ")" must be quoted to avoid terminating the function.
   call $write(c:\myfile.txt, "a string with ) paren")

** Multiple line string must be quoted
   echo "this is a string
with multiple lines

and a blank line above this one"
```

Menu Commands In addition to the comprehensive range of TPL statements any command accessible from the TrainPlayer menus can also be used within a script. Executing a menu command from a script causes the same action as choosing it from the menu.

Menu commands apply only to the train depicted in the train window. Therefore to use a menu command for the scripted train, that train must be the train of focus in the Train Window (Use the command Train "train name" to achieve this). You need to be aware that there could be consequences for anyone driving a train with the train controller while the script is running, as changing the selection in the script would take control away from their train. It is therefore advisable not to use the Menu Commands unless there is no alternative, or unless you are sure that all other activity on the layout is suspended. These commands can be very useful at the beginning of a Master Script as part of the set up procedure before any of the trains have started running.

If an item in the menu at any level consists of more than one word, then that item must be enclosed in quotes. Popup menu items are also available for the Car, Layout, Switch, Turntable, Track, Circle, Horn and Station context menus. As a shortcut, you do not need to spell out all the words in full. You may abbreviate a menu item to its first few characters. e.g. "view toolbars customize" may be shortened to "vi to cu."

Note: Many menu commands will make little sense in the context of a script. Some bring up dialogs, which are not scriptable. Some duplicate the functions available using Train Commands. Some are toggle switches, but since a script does not always have a way to know the current setting, the results can be unpredictable. Context menus often need to reference a point on the layout -- where you right-clicked to bring up the menu -- but since a script cannot supply this, context menus may do nothing or work unpredictably.

```
** Examples of using Menu Commands, note that commands with embedded spaces must be quoted.
   file save                  ** bring up File Save dialog to save the current layout
   train freight1             ** select train Freight1
   train name "SF Express"    ** change the name of the selected train to the name given
   view tool cust             ** bring up Tools Customization dialog
   train car X40              ** first select a car in the train
   car "add car" hopper       ** then insert a hopper at the current insert point of the selected train
   tools "enable yard mode"   ** turn on Yard Mode operation
   train new                  ** create new four-car train at the default location
   window "tile horiz"        ** if multiple document windows are visible, tile them
   train speed double         ** double the speed of the selected train
   train car HK22             ** select a car in a train
   car exclude                ** exclude the selected car from Ops by adding the XO flag
                              ** exclude is a toggle and the same command can be used to cancel the flag
```

Part 2 - An introduction to Variables in the context of TPL

Variables are a powerful element of TPL and these can broadly be categorized into three types, System Variables, Registry Variables and User Variables. By extracting and testing the data stored in these variables the scripts can take branching decisions on how to proceed.

System Variables all have a $ prefix. These variables are being continually updated by the program and cannot be SET by the user, however the values of a System Variable can be extracted to a user variable for testing or later use in the current script or in another script. The system variables can also be tested using conditional statements such as IF (see flow commands below). System variables can be embedded directly into strings where required (without needing the @ symbol).

$X_TRAIN	returns the name of the crossing train -- i.e. the train owning the script	Returns the name of the train owning the script. In a Junction Action script this is the Train that crossed the junction and picked up the script.
$X_SPEED	returns the speed of the crossing train that is running the script	
$X_CAR	returns the label of the car when a JA script is triggered by the car	Only relevant to a Junction Action Script triggered by a crossing car.
$TIME	returns current time of day in (h:m:s in 24-hour format), i.e real time, not layout time.	
$DATE	returns today's date (mm/dd/yyyy)	
$SPEED	returns the speed of the train running the script in MPH (KPH if metric settings in effect)	
$KEY	returns the numeric code of last key hit on keyboard for processing (full list of key codes can be found in Appendix B)	

```
** Example to report location of train running the script
   AFTER J123
   NOTE $x_train passing through Blindman's Bluff

** In a Junction Action triggered by a car with a specific aar code, load the car
   load Sx_car                          ** $x_car can't be used in a train script, only a Junction Action

** Extract the current value from a System Variable and send it to a Note Window
   Note The time is $TIME

** Processing Car Properties using $car
   let carlabel = "X333"
   let isLoaded = $CAR(@carlabel, loaded)
   if (isLoaded = "1")
     echo car @carlabel is loaded with $CAR(@carlabel, "loadname")
   endif

** Allocating the name of the train running the script to a user variable for later use in another script.
   LET savedtrain = $x_train
** and then later in another script
   Start @savedtrain                ** for an explanation of @ see following page
```

Set and Let are the commands used for allocating values to User Variables and to Registry Variables	
Set <var> <expr>	sets a new or existing variable to an initial value or to a new value; equivalent to LET <var>=<expr>. The name given to a User Variable must start with an alphabetic character. There must be a space before and after the variable name. Note: If SET uses a multiple word variable name it must be enclosed in quotation marks. This is not the case with LET which reads to the = sign.
Let <var>=<expr>	sets a given variable to the value of the expression; creates variable if needed (Set and Let perform the same task and are interchangeable) <var> = a user variable name or a registry setting name. The = sign is mandatory. <expr> = a number, a string, an arithmetic combination using + - * or /, or a string concatenation using +. Values and operators must be separated by spaces.
Reset <registryvariable>	reset is provided for the specific purpose of resetting a temporary registry variable that has previously been modified with set or let back to the original value that is stored in the Windows registry. If modified registry variables are not reset they will retain their temporary values until TrainPlayer is close down and restarted.

User Variables are manipulated using the SET and LET commands, if you allocate a value to a variable not previously defined it will be automatically created as the value is allocated. However it is considered good programming practice to initialize any variables in the Master Script as a layout is opened. A User Variable can have any name without spaces, they are Global in scope and can be accessed by any layout that is open during a TrainPlayer session. Any numeric or string value can be assigned, matches on names and values are case-insensitive. The values of User Variables are not lost when a layout is closed, but they are lost when TrainPlayer is closed.

```
** EXAMPLES

** Define a variable called "v1" and give it the value 44
** If the variable has been previously used its value will be overwritten with 44
   LET v1 = 44

** Values do not have to be numeric
   SET Block23 "Occupied"

** Examples of using Let for Arithmetic and string concatenation
   LET vnum = 22+33-4
   LET vstr = "abc" + "def"     ** Quotes required for concatenation, optional for single literal string.
   LET vst2 = vstr + "ghi"
   LET v2 = 9 + 6

** Allocate the value of a System Variable to a User Variable for subsequent processing by IF
   AFTER J123
   SET siding $x_train

** Embedding System Variables in strings
   set v2 "Time and date:" +$TIME + " " + $DATE
   echo This train is $x_train
```

Registry Variables are effectively copies of the entries in the TrainPlayer registry folder hkcu\Software\TrainPlayer\TrainPlayer\Settings, TPL now permits these to be modified from within a script. Most are true/false (1/0), strings, or numeric values. To get the current value of any registry setting, use the $SETTING function (see Part 4). In order to maintain the integrity of the Windows Registry any changes made during a session are lost and the Registry defaults to its original values when TrainPlayer is closed.

```
** Examples of setting registry variables to temporary values for the duration of a session
   SET AccelFactor 20          ** Sets the Acceleration Factor to 20 for the duration of the session.
   Let Max MPH = 100           ** Sets a Global Max Speed of 100 MPH for the duration of the session.
   SET "Max MPH" 100           ** also Sets a Global Max Speed of 100 MPH for the duration of the session.
   SET PassengerCarsLoadable 1 ** allows passenger cars to be loaded and unloaded. Default setting is 0

** Changing the fast clock reset time.
   SET ClockResetTime 900      ** changes the fast clock reset time from its default of 05:00 to 09:00
   ** Note this command must be followed by a CLOCK RESET and a CLOCK START command in the script

** Adjusting the Scroll Distance Factor
   SET ScrollDistFactorx10 25   ** Default value is 70, changing to 25 will keep Loco central on screen.
   ** The Value represents a fraction of the size of the layout window, default 1/7 from the edge.
   ** Set this to 30 and it would start 1/3 from the edge.
   ** Don't set below 20 as results are undefined and unpredictable.

** Reset previously adjusted Registry Variables back to their original values
   RESET AccelFactor             ** Resets the Acceleration Factor to its original value from the Registry
   RESET Max MPH                 ** Resets Max Speed to the users own usual setting from the Registry
   RESET PassengerCarsLoadable   ** Resets passenger cars to its original value from the Registry


** Reset the size, properties and position of the Note Window within the application window.
      set NoteWndRect "0,0,400,300" ** Co-ordinates represent "top,left,bottom,right" corners of window.
      set NoteWndAutoFit 1          ** Allows vertical expansion of Note to autofit text, 0 is fixed size.
      set NoteWndTextPos L          ** Sets text alignment within Note, L left, R right, C center.
```

@variablename allows the value of a variable to be used in place of a literal value anywhere in a script, this applies to both numerical values and text strings. @ references can be used anywhere that a value is needed in a line of script such as in SET or LET statements, providing a reference for a CarID, a TrainID, or for use within the text of an Echo or Note statement.

@ references must appear on the right-hand side of SET and LET statements, not on the left, but they can be used on either or both sides of an IF statement or in a While loop comparison. When a script encounters @variablename it is replaced by the value of the named variable, this also allows for the value to be embedded in string statements providing a space follows the @variable statement. If a trailing space is needs to be avoided @variablename@ can be used without spaces.

```
** Examples showing the use of the @variablename reference
   Let v1 = (@v1 + 1)         ** Reads the value of v1, adds 1 to it and places the result back in v1
   Set v1 22                  ** Places the value 22 in v1, if the variable doesn't exist it is created.
   Set v2 @v1                 ** Extracts the value from v1 and places it in a new variable v2
   Let v2 = (@v2 +@v1)        ** Takes the of v2, adds the value of v1 to it and stores the result in v2.
   Let train_name = $x_train  ** Takes the name of scripted train from $x_train,stores it in train_name
   Echo Train is @train_name  ** Sends the text and the name of the train to the Schedule Window
   Let v1 = @v2 * 6.3         ** Multiplies the contents of v2 by 6.3 and stores the result in v1
   Let v2 = "a string" + @v1  ** Places a string in v2 comprising the text followed by the number from v1
   Let car = $x_car           ** Places the name of the car triggering a Junction Action in the variable.
   Let train = $x_car(@car,name) ** Places the name of the train containing the above car in the variable
   Echo The car @car is located in train @train  ** Name of the car and train are embedded in the string.

** Practical example as used in a Junction Action
   set BSA $x_train           ** Place the name of the train crossing the junction into the variable BSA
   set BM "clear"             ** Remove the name of the train from the previous block it occupied
   echo @BSA in tunnel        ** Report position of train to schedule window using name stored in variable
   speed @yardspeed           ** Reduce the speed to the speed already defined as the yard speed
   throw 102 0                ** Throw the switch behind the train back to its default setting
   AT 750 Stop                ** Stop train when hidden in tunnel
   IF (@BSA="Train4")         ** Check the variable to see if this is the Freight Train (Train4)
      load cut g30            ** If it is load the cut of cars while the train is out of sight
   endif                      ** End the conditional IF statement
```

@@variable references the "contents of" a variable which contains the name of another variable in order to extract the data from the second referenced variable. So the expression "@@<var>" gets you the contents of the variable whose name is stored in <var>.

```
** Examples of redirection with @@ to obtain access to a second variable whose name is stored in the first
   LET v1 = "v2"          ** v2 is the name of a variable not its contents.
   LET v2 = "some text"   ** We want to be able to access this without using v2 as a name.
   Echo @@v1              ** Will print the text stored in v2 to the Schedule Window.

** LET allows @ on the left hand side for the special purpose of placing values in a referenced variable
   LET @v1 = abc    ** means put the value abc not into v1 but into the variable whose name it stores
   Echo @v1         ** will print a variable name (lets assume this to be v2)
   Echo @@v1        ** will print the contents of v2, which in this case is abc
```

Part 3 - Testing the Variables and using the Script Flow commands

Goto & Label:	
<label>:	a word ending with a colon is a label for use with "goto", a label must be on a line by itself.
Goto <label>	causes script to jump directly to the statement after the label, skipping all commands in between, can also jump back to an earlier label

If ; Elseif; Else; Endif	
If (<expr> <comp-op> <expr>)	compares two expressions and branches the program based on the result of the comparison, <expr> is a number, string, variable, or complex expression; <comp-op> is an operator comparing the values of two expressions. (brackets are required around conditions) <comp-op>-- comparison operator must be one of:: =, <, >, <=, >=, <>.
Elseif (<expr> <comp-op> <expr>)	evaluates an alternate comparison after the IF (optional component, there can be more than one Elseif if required) <comp-op>-- comparison operator must be one of:: =, <, >, <=, >=, <>.
Else	begins an alternate block of code after IF or ELSEIF (optional component but if used there can only be one Else)
Endif	ends the IF block (essential component) If ... Endif blocks can be nested inside other blocks.

```
** Example of monitoring keyboard waiting for a specific key press to define next action
   start:                 ** This is a label (a word ending in a colon: on a line of its own)
   on key                 ** A Wait Condition which suspends the script until a key is pressed
   echo $key              ** A report to the Schedule Window as to which key was pressed
   if ($key = 65)         ** Conditional check to see if A was pressed. If so speed 20 and ignore the rest
       speed 20
       GoTo carryon
   elseif ($key = 66)     ** If first condition not met, check to see if B was pressed. If so Reverse.
       reverse
       GoTo carryon
   elseif ($key = 67)     ** If neither A or B pressed, check for C, if so Stop
       stop
       GoTo carryon
   endif
   goto start             ** If the script is here, none of the conditions were met, go back to start:
   carryon:
   ** carryon: is a label, if we are here the appropriate change is made and the script can continue.

** The IF statement structure may also be combined on one line with statements separated by semicolons.
   If (BSA=@BS);set BSA "clear"; endif  ** If variable BSA contains the contents of BS, SET it to clear.
```

While Loops	
While (<expr> <comp-op> <expr>)	compares two expressions, executes or skips block up to endwhile based on result of test <comp-op>-- comparison operator must be one of:: =, <, >, <=, >=, <>.
Break	breaks out of WHILE loop if necessary (optional component usually within an embedded IF)
Endwhile	ends a WHILE block (essential component)

While loops are extremely powerful structures that enable you to emulate almost any customized Wait Condition you require.

```
** Example: Script on Northbound train standing in a siding.
   While (DPM<>@T3); AFTER 0:0:01; endwhile   ** While DPM does not hold the name of the Southbound, Wait.
** This train is waiting until a Southbound train whose name is stored in T3 has cleared the North Switch.
** The Southbound train will SET its name in DPM after clearing the switch and enable this loop to end.
   Forward speed 10                          ** When While loop has terminated route must be clear again.
```

Defining Subroutines as separate text files for repetitive use	
Subroutine.txt	A subroutine is a separate text file containing Script Code that is stored in a separate file within the Scripts folder. A subroutine can be accessed from any Script using the CALL command (details below). Some users prefer to place all their code in subroutine files and restrict the scripts within the layout files to a list of CALLs to the different subroutine files - this is down to personal preferences.

```
** Example of Subroutine code stored in a separate text file
** Requires three parameters %1 %2 %3
** %1 is the name of a variable
** %2 is the value waited for, control will not return to the calling script until the value is correct.
** %3 is the speed required when restarting the train
   SET Trainspeed %3                  ** Save the current speed
   WHILE (%1 <> %2); STOP; ENDWHILE   ** Hold this script here until the value is set by another script.
   Speed @Trainspeed                  ** Start the train again and return control to the calling script.
** To use this three line subroutine a script would need to use the CALL command (see below)

** Example of a subroutine to set or change the properties of the Note Window
** Requires three arguments, %1 dimensions, %2 Autofit flag 0 or 1, %3 text justification L C or R
** Example call >> call setupnote "0,0,400,300" 0 C
      set NoteWndRect %1
      set NoteWndAutoFit %2
      set NoteWndTextPos %3
```

Defining Procedures within the layout file	
Proc <routine>	begins the definition of a procedure of a given name, this can be placed in any script but is more usually found in the Master Script. A Procedure can then be accessed from any script using the CALL command (details below). Care needs to be taken in naming Procedures to avoid duplicating any names that may have already been used for Subroutine files.
Endproc	ends a procedure definition (essential component)

```
** Example of a procedure to hold up a script until specific train conditions have been met.
** This can be used during user interaction to check if instructions from Notes have been complied with.

** The procedure requires four parameters
** %1 Name of train for testing. (Can be passed literally, as contents of a variable, or as $x_train).
** %2 S if instructions require train to Stop, or M if instructions require train to start Moving again.
** %3 F or R, the direction setting needed on the Train Controller before allowing the script to continue.
** %4 Train length (car count) needed before allowing script to continue, to check couple/uncouple events.

PROC HoldUntil
   ** If train is not the required length, hold up the script.
      While ($train(%1,Ncars)<>%4); endwhile
   ** If train direction is not correctly set on the controller, hold up the script.
      While ($train(%1,Direction)<>%3); endwhile
   IF (%2="S")
      ** If train is required to stop, hold up script until it stops.
      While ($train(%1,Speed)>0); endwhile
   ELSE
      ** If train is required to move, hold up script until it starts moving.
      While ($train(%1,Speed)<1); endwhile
   ENDIF
ENDPROC

** Procedure to set up and start the fast time clock
** Requires two arguments: %1 speed clock is to run, %2 start time (without colon)
** Called by: CALL startclock x10 0800

PROC startclock
   clock stop
   set clockresettime %2
   call $rrclock(%1)
   clock reset
   call $view(show,clock)
   clock start
endproc
```

Calling Procedures, Subroutines and System Functions	
Call <routine> <arglist>	Calls a given subroutine, a given procedure and passes any required arguments to it. Also used for calling System Functions. Arguments are separated by blanks so if you want to pass an argument which itself contains blanks, it must be quoted.

Calls made to subroutines and procedures are identical in structure, the word CALL, followed by the name of the procedure or subroutine file (without the txt extension) and then a list of any arguments required separated by spaces, quotes should not be used unless the individual arguments contain spaces.

When a script encounters a CALL statement that is not followed by $ (see below) it first checks to see if a procedure has been defined in the layout, if so it copies the code from the procedure into the script and runs it. If no procedure is found the Scripts folder is checked to see if there is a subroutine of that name. If there is it opens the file, copies the code into the script at the position of the CALL statement, closes the text file and runs the code it has inserted into the script.

Because the program checks for an embedded procedure first, it is possible to maintain a standard set of subroutines in the Scripts folder, but to customize them for specific purposes by copying them into a procedure definition in the layout file and making the required adjustments. Before you can CALL a procedure it must have already been read into the program and for this reason it makes sense to define procedures in the Master Script as this runs automatically when the layout first opens.

```
** Example Call to a subroutine script
   CALL filename signal23 clear 15
** i.e. CALL followed by the filename, the variable to be tested, the result required, the current speed

** Example Call to the procedure HoldUntil shown above on this page
   CALL HoldUntil @T1 S F 12
** i.e CALL followed by defined procedure name, "trainname" from var T1, then "S" we want to stop,
** then "F" we want to be facing forward and "12" is the number of cars we want to be in the train.
```

Calls to System Functions use a different syntax, the argument list must be in parenthesis, with arguments separated by commas:

```
** Example Calls to system functions  Details of system functions are included in Part 4.
   Call $rrclock(x12)                          ** Resets the layout clock to run at 12x normal speed.
   Call $write("c:\outfile.txt", "here is a display string to be written to file")
   ** Writes the specified display string to an text file.
   Call $view(show, schedule)                  ** Opens/shows the Schedule Window on the screen.
```

Part 4 - System Functions and the Technical Stuff

System functions, like system variables, start with the $ prefix but can provide a wider range of results dependent on the argument offered.

$Train Function - Used for extracting status information of any train on the layout	
$TRAIN	If no argument is given, $Train returns the name of the selected train. Otherwise the first argument must be the name of a train on the layout, "Train" plus a numeric ID or any car ID that is in the train.
$TRAIN(name, Ncars)	returns NCars (or Length) of train
$TRAIN(name, car<i>)	returns ID number of car <i> in train (between 0 and Ncars)
$TRAIN(name, speed)	returns speed in MPH of selected train (KPH if metric) (equivalent data to $speed)
$TRAIN(name,dir)	returns F if train is going forward, otherwise R

```
** Example to discover the number of cars (or length) of a train.
   let trainname = $x_train                  ** Identify the train running the script, then get car count
   let trainLen = $TRAIN(@trainname, Ncars)  ** let trainLen = $TRAIN(@trainname, length)also acceptable.

** Example to obtain Car ID of the fifth car in the train.
   echo car 5 label is $TRAIN(@trainname, car 5)

** Delay loop to hold up a script until the operator starts moving the train whose ID is in variable hty.
   While ($train(@hty,Speed)<1); endwhile

** Example to check that operator reversing an engine has coupled to consist and started moving Forward.
   NOTE Couple up to your train\nProceed to next station
   While ($train(@hty,Ncars)<1); endwhile         ** Holds up script until engine has coupled.
   While ($train(@hty,Direction)<>"F"); endwhile  ** Holds up script until train set to go forward.
   While ($train(@hty,Speed)<1); endwhile         ** Holds up script until train is moving.
   After J123                                     ** Holds up script until train has passed J123
```

$Car Functions - Used for extracting status information of any car on the layout	
$CAR	If no argument is given, $CAR returns the label of the selected car. Otherwise the first argument must be the carID of a car on the layout.
$CAR(carID, loaded)	carID must be the label of a car on the layout. returns 1 if car is loaded otherwise 0
$CAR(carID, loadname)	carID must be the label of a car on the layout, returns name of the load assigned to the car (if any)
$CAR(carID,aar)	carID must be the label of a car on the layout, returns aar code of car
$CAR(carID,track)	carID must be the label of a car on the layout, returns the ID number of the track the car is standing on.
$CAR(carID,note)	carID must be the label of a car on the layout, returns the text stored in the note field of the car properties.
$CAR(carID,revengine)	carID must be the label of a car on the layout, returns 1 if the Reverse Engine flag applies to the car.

```
** Example to check and see if a car is loaded or unloaded
   let carlabel = "X333"
   let isLoaded = $CAR(@carlabel, "loaded")
   if (isLoaded = "1")
      echo car @carlabel is loaded with $CAR(@carlabel, "loadname")
   else
      echo car @carlabel is not loaded
   endif
```

$Switch Function - Used for checking the current position of any named switch	
$SWITCH(jxn)	Takes one argument, a junction number, returns the switch position (0 or 1 for a standard switch) , -1 if the junction is not a switch. Argument can optionally include a prefix J or j.

```
** Example: Stop and throw switch ahead if not correctly set to required 0
   at 122                              ** When we reach here check the switch ahead
   if ($Switch(123)<>0)                ** If not set to position 0 stop and instruct operator to throw it
      Stop
      note Throw upcoming switch
      while ($Switch(123)<>0); endwhile  ** Hold up script until switch is in correct position
      speed 10                           ** Restart train and carry on
      note                               ** Hide the Note window
   endif
```

$Layout Function - Used for extracting information about the active layout	
$LAYOUT	with no arguments returns the name of the active layout
$LAYOUT(name)	returns the name of the active layout:
$LAYOUT(Ntrains)	returns the number of trains on the layout
$LAYOUT(train <i>)	returns the name of train <i> (where <i> is a number between 0 and Ntrains)

```
** Example of using $layout to get a report list of the trains on a layout
   let i = 0
   while (i < $LAYOUT(ntrains))
      echo Train @i is named $LAYOUT(train @i)
      let i = i + 1
   endwhile
```

$RRClock Functions - A powerful family of functions used for managing the Scale Clock	Usage depends on the arguments supplied.
$RRCLOCK	with no argument returns the current time shown on the scale clock $RRCLOCK() with empty parens is equivalent
$RRCLOCK(hhmm)	with time argument sets the clock to the specified time which should be in the 24 hour format HHMM (though a leading zero can be omitted)
$RRCLOCK(x)	with argument X returns the time speed multiplier
$RRCLOCK(x<n>)	sets the time speed multiplier to the value given by <n>
$RRCLOCK(hhmm,hhmm)	with two arguments performs clock arithmetic, second value can be a negative value (Does not alter the clock settings in any way)

Time values passed as arguments do not need the colon between hours and minutes but the processor will accept these, for times before 10:00 the leading 0 is optional (0700, 07:00, 700 and 7:00 are all accepted). Values returned from the clock function will always include the colon in the return value (e.g. 23:45) so if you need to make a string comparison you must include a colon in your comparison variable.

Reading and setting the clock

```
** Examples of getting and setting the time on the layout clock
   echo Current time on layout is $RRCLOCK      ** Reports the text and clock time to Schedule Window
   let curtime = $RRCLOCK                       ** Extracts the current clock time to a variable
   let factor = $RRCLOCK(x)                     ** Extracts the current clock speed to a variable
   call $RRCLOCK(x12)                           ** Sets the clock speed to x12
   call $RRCLOCK(0530)                          ** Sets the layout clock to 0530 (using 24 hour format)
   clock reset                                  ** Menu command to reset to the new time
   clock start                                  ** Menu command needed to start the clock running

** Example of setting and starting the layout clock
   clock reset                                  ** Reset the layout clock to current defaults
   call $rrclock(1045)                          ** Set the clock to 1045 (a.m.)
   call $rrclock(x10)                           ** Set the clock speed factor to x10
   clock start                                  ** Start the scale clock running

** Example of using the clock for a random delay, also uses $rand function (see below)
   Clock Start                                  ** Start the clock running
   STOP                                         ** Stop the train
   Let start=$rrclock                           ** Extract current time from clock
   let delay =$rand(10,59)                      ** Use $rand function to get delay between 10 & 59 mins
   let end=$rrclock(@start,@delay)              ** Set the end time by adding delay to start time
   While $rrclock<@end; endwhile                ** Hold up the script until the end time is reached
   speed 40                                     ** Restart the train moving

** Example of using the $rrclock function within an IF statement
   set ferrytime 07:00                          ** Set variable for testing against the clock
   if ($rrclock<@ferrytime)                     ** Compare ferrytime with current time and report
      echo The ferry sails at @ferrytime, the time now is $rrclock
   else
      echo Wait here, time is $rrclock, ferry left at @ferrytime
   endif
   ** In this case the comparison done by the IF statement is a string comparison, this means a colon
   ** is needed in the ferrytime variable for comparison with the value returned from the function
```

Manipulating time with clock arithmetic

When $rrclock is used with two arguments it can perform clock arithmetic, neither of the values used have any effect on the current clock setting which retains its present value regardless of the times used as arguments for the arithmetic. The value returned by the argument is the sum of the two arguments, a negative value can be used as the second argument.

```
** Example to record a period of elapsed time
   Echo $rrclock                        ** Outputs the current layout time to the Schedule Window
   Let Start = 16:00                    ** Stores a start time in variable start
   Let Finish = 19:30                   ** Stores a finish time in variable finish
   Echo $rrclock(@finish, -@start)      ** Outputs the result of deducting start time from finish time
   Echo $rrclock                        ** Demonstrates the calculation has not changed the clock value

** Example to calculate what the layout time will be one hour from now to dispatch a train
...Stop                                      ** Stop the train and wait for one hour
   Let arrivetime = $rrclock                 ** Get current layout time from the $rrclock function
   Let departtime = $rrclock(@arrivetime,0100)     ** Add one hour to the current time
   echo Train arrived at @arrivetime and will depart at @departtime
   AT @departtime Speed 10                   ** Restart the train

** Examples to demonstrate that the arithmetic works properly from late evening to midnight and beyond.
   echo sum(2200,0159) = $rrclock(2200,0159)       ** returns sum(2200, 0159) = 23:59
   echo sum(22:00,02:00) = $rrclock(22:00,02:00)   ** returns sum(22:00,02:00) = 00:00
   echo sum(2200,0201) = $rrclock(2200,0201)       ** returns sum(2200,0201) = 00:01
   echo sum(2200,0301) = $rrclock(22:00,03:01)     ** returns sum(2200:0301) = 01:01
   echo sum(600,-200) = $rrclock(600,-200)         ** returns sum(600,-200) = 04:00
   echo sum(2200,0301) = $rrclock(2200, 0301)      ** returns sum(2200,0301) = 01:01
   echo sum(100,-200) = $rrclock(100,-200)         ** returns sum(100,-200) = 23:00
   echo sum(00,300) = $rrclock(00,300)             ** returns sum(00,300) = 03:00 (midnight as first arg)
```

$Setting Function - Used for extracting the current value of any named registry setting	
$SETTING(regsettingname)	Returns the value of the named registry setting stored under the key: HKEY_CURRENT_USER\Software\TrainPlayer\Trainplayer\Settings The name used in the argument must match a value name in that key folder. Quotes round the argument are accepted but not required.

```
** Examples of extracting current values from the registry (the registry settings are not documented)
** If required these can be changed for the duration of a session with SET or LET
   echo Current accel factor = $setting("AccelFactor") ** Sends current value to the Schedule Window
   LET couplespeed = $setting(Coupling Speed)          ** Stores the current coupling speed in a variable
   SET Coupling Speed @couplespeed*2                   ** Doubles safe coupling speed for this session
   SET RailSeparation 40                               ** Set rail separation for narrow gauge operation
   echo Rail Separation has been reset to $setting(RailSeparation) ** Check it has been adjusted
```

$Rand Function - Used for generating random numbers for use in scripts	
$RAND	with no arguments returns a random number between 0 and 100
$RAND(minimum,maximum)	with arguments returns a random number between minimum and maximum, the arguments supplied must both be integers.
$RAND(maximum)	if only one argument is supplied it is taken as maximum and minimum defaults to zero

```
** Examples:
   let vrand1 = $rand(1,10)                ** places a random number between 1 and 10 and vrand1
   let vrand2 = $rand                      ** places a random number between 0 and 100 in vrand2
   echo Pick up $RAND(1,5) cars here       ** on a junction action gives instruction to operator
   echo random percent = $RAND             ** outputs a percentage figure to the Schedule Window
   let WaterUsage = $rand(2000,3000)       ** places a random number between 2000 and 3000 in a variable
```

$DataDir Function - Used to locate the TrainPlayer AppData folder on any PC	
$DATADIR	returns the path to the TrainPlayer application folder, no arguments are required, this can be used to set paths to the sub directories

```
** Examples
   let scriptsDir = $DATADIR + "\Scripts"  ** places the full path to the Scripts Directory in ScriptsDir
   echo $DataDir                           ** outputs full path to the TP apps data to the Schedule Window
```

$MsgBox Function - Used to pass a string message to the user and accept a mouse click response		
$MSGBOX(message)	with single string argument, displays the message string and offers only an OK button to acknowledge message. *strings are unquoted.*	
$MSGBOX(OC,message)	with OC as first argument, displays the message string and offers OK and Cancel buttons for response.	*strings are unquoted*
$MSGBOX(YN,message)	with YN as first argument, displays the message string and offers Yes and No buttons for response.	*strings are unquoted*

This function returns a value of 1 if the OK or Yes buttons are clicked, or 0 if the Cancel or No buttons are clicked. These values can be used for subsequent processing. It is important to be aware that while the $MSGBOX function is waiting for a response all other action on the layout is suspended. This tends to limit its use somewhat, although it can be valuable to use during a setting up procedure from the Master Script or when all train action has been deliberately stopped for some other reason (such as the end of a days operations).

```
** Example as an introductory note when a layout first opens
   let prompt = This layout has a script to run the passenger train.\n\nWould you like to start it now?
   if ($msgbox(YN, @prompt) = "1")
      train script play
   endif
```

```
** call $msgbox(This is a note!)                    ** for a single argument this only has an OK button
   let v1 = $msgbox(YN, Do you want to continue?)    ** returns 1 if click Yes
   IF (v1<>1); GOTO end; endif                       ** jumps to a label end: at end of script if true
   echo OK we are continuing                         ** otherwise script continues
```

$View Function - Used for opening or closing any window or feature from the View>Windows or the View Features sub menus	
$VIEW (IsVisible,Name)	returns a value of 1 if the named window or feature is already displayed, returns 0 if it is not.
$VIEW(Show,Name)	opens and displays the window or feature named, Name must come from either the View>Windows or the View>Features sub menu.
$VIEW(Hide,Name)	closes or hides the window or feature named, Name must come from either the View>Windows or the View>Features sub menu.

```
** Example to ensure Clock Window is on display
   call $view(show, clock)               ** Display Clock Window (does nothing if it's already showing)

** Example to check if the Schedule Window is visible
   let v1 = $view(isvisible, schedule)
   if (v1 = 0)
      echo The schedule window is not visible, but I expect you knew that already  :>)
   endif
```

$Read and $Write Functions - Used for reading and writing data to an external file	
$READ(filename)	returns the contents of a text file as a string, arguments are unquoted
$WRITE(filename,string)	writes the given string of text to the given filename, function returns 1 if successful, 0 if write failed, *arguments are unquoted*
$WRITE(filename,append,string)	appends the given string of text to the given filename, function returns 1 if successful, 0 if write failed, *arguments are unquoted*

If you APPEND to a file that doesn't exist it is created, if there is no CR /LF at the end of the output you need to add it. If you wish you can use CALL to activate these functions and ignore the return value. $read and $write will both accept Windows relative pathnames.

```
** Example of reading in a file from the Scripts folder and displaying it to the Schedule Window
   let pathname = $DATADIR + "\Scripts" + "\myfile.txt"    ** for $Read the file should already exist
   let file_data = $READ(@pathname)
   echo @file_data

** Example to write a new text file to the Scripts folder
   let pathname = $DATADIR + "\Scripts" + "\outfile.txt"
   call $write(@pathname, "here is some data to be written to a file")
   ** If your text is intended for a Note window when reloaded you can include \n to throw a new line.

** Example to append text to a file in the current layout directory using a relative path
** If the file does not exist it will be created
   let pathname = "c:outfile.txt"
   call $write(@pathname,append,and here is some data)
```

String Processing Functions - A powerful set of functions for manipulating the data in text strings	
$STRLEN(string)	returns length of string, returns zero if the string is empty.
$SUBSTR(i1,n,string)	returns substring of s starting at i1, n chars long; if n = -1, go to end (note position i1 is taken as 0 in front of first character)
$FINDSTR(sub,string)	returns position of sub within s, or -1 if not found

$STRLEN Function

```
** Example to evaluate the number of characters in a given string. (Note: Quotes are optional).
   let len = $STRLEN("12345")                          ** Allocates the value 5 to the variable len
   let len = $STRLEN("abcde")                          ** Allocates the value 5 to the variable len
   let anystring = "Now is the time for all good men to come to the aid of the party"
   echo the length of anystring is $Strlen(@anystring)     ** Outputs "the length of anystring is 64"
```

$SUBSTR Function

```
** Example to demonstrate extracting a substring from within a longer string
   let anystring = "Now is the time for all good men to come to the aid of the party"
   let substring = $substr(20,12,@anystring)
   echo @substring
   ** Outputs "all good men" to the Schedule Window, i.e. 12 characters starting after character 20

** Example to demonstrate sending only the first few words to the Schedule Window
   echo $substr(0,15,@anystring)
   ** Outputs "Now is the time" to the schedule window, 15 characters after character 0)

** Example to demonstrate sending the text from a given point to the end of the string.
   let totheend = $substr(16, -1,@anystring)  ** -1 means from the stated start to the end of the string
   echo @totheend
   ** Stores the string from after character 16 to the end of the string
   ** echo statement reports as "for all good men to come to the aid of the party"
```

$FINDSTR Function

```
** Example to find the start position of a stated substring within another string
   let anystring = "Now is the time for all good men to come to the aid of the party"
   let searchstring = "good men"
   let position = $findstr(@searchstring,@anystring)
   echo @searchstring starts after character @position
   ** position is found as after character 24

** Example to show that if the substring is not found the function returns -1 to the variable
   let anystring = "Now is the time for all good men to come to the aid of the party"
   let searchstring = "Jim Dill"
   let position = $findstr(@searchstring,@anystring)
   echo @searchstring starts after character @position
   ** position is found as after character -1 (in other words it doesn't exist)
```

==========

Document compiled from information supplied by Jim Dill to the Script Development Group

Richard Fletcher, April 2014 (updated August 2014)

Appendix A - Concise Summary of all the TPL statements available for Scripting

Wait Conditions - Are used to delay scripts and ensure subsequent commands are not processed until the specified events have occurred.

AT <jxn>.....waits until midpoint of lead car crosses junction (jxn can include j prefix)
AT <t j d>.....waits until midpoint of lead car crosses (% dist d from jxn j on track t)
AT <h:m[:s]>.....waits until specified time is shown on layout clock; h:m required, secs optional
AT <station>.....waits until lead car of train enters named station
AFTER <jxn>.....waits until midpoint of last car crosses junction (jxn can include j prefix)
AFTER <(t j d)>.....waits until last car of train crosses exact spot
AFTER <h:m:s>.....waits until specified time has elapsed on real time clock (not layout clock)
AFTER <station>.....waits until last car of train leaves station
ON STOP.....waits until train comes to a complete stop
ON COUPLE.....waits until train couples with another car
ON THROW <jxn>.....waits until specified switch is thrown by any means
ON TABLESTOP.....waits until turntable finishes rotating
ON KEY [<key>].....waits until user presses any key or the specified key

Train Commands - Are used with wait conditions to start, stop and drive the trains, set direction, control peed, and start other train scripts.

Forward.....sets direction to forward
Reverse.....sets direction to reverse
Speed <mph>.....sets speed to given mph; starts train moving if stopped
Stop.....decelerates to a stop
Train <train>.....selects the named train as the train of focus in the train window
Drive <train>.....transfers control temporarily to specified train
Enddrive.....transfers control back to the original train
Start <train> [<mph>].....starts train script, or starts train moving if no script (at 5 mph or given speed)
Autopause <secs>.....turns on pauses between commands of x secs during train movements
Uncouple <slot> OR <car> OR <car><car> .....uncouples at given position, at given car, or between given cars
Horn [<msec>].....sounds train horn for 3 secs or for stipulated time in millisecs

Layout Commands - Are used to set routes, control turntables, play sounds and load and unload cars at set points on the layout.

Throw <jxn> [<pos>].....throws switch to stated position or next available position if not stated
Rotate <ttbl> <jxn> [CCW].....rotates stipulated turntable to junction, clockwise unless CCW added
Tablestop.....stops turntable rotation
Sound [loop|stop] <spath>.....plays or stops sound in named wav file. If LOOP is omitted plays once.
Load [Toggle] [Car/Train/Cut] <ids> [<loadname>]. loads specified car(s) with named load or default load if loadname omitted
Unload [Toggle] [Car/Train/Cut] <ids>.....unloads specified car(s)

User Interface Commands - Are used to pass messages to users, process responses, record key presses, and set values for processing.

Echo <string>.....displays the given string in the schedule window
Input <var> [<prompt>].....sets a user variable from the input dialog, optionally using a given prompt
Note [<string>].....displays a given string in a popup window; hides window if no string present
Set <var> <expr>.....sets a given variable to a value; equivalent to LET <var>=<expr>
Let <var>=<expr>.....sets a given variable to value of expression; creates variable if needed
Reset <registryvar>.....resets the temporary registry variable to the original value from the registry

Flow Commands - Are used to control the flow or order of events based on different conditions encountered while running the trains.

Goto <label>.....jumps to the statement after a given label
If (<expr> <comp-op> <expr>).....compares two expressions, branches program based on result
Elseif (<expr> <comp-op> <expr>).....evaluates an alternate comparison after the IF (optional component)
Else.....begins an alternate block of code after IF or ELSEIF (optional component)
Endif.....ends the IF block (essential component)
Call <routine> <arglist>.....calls a given subroutine or procedure and passes any required arguments
Proc <routine>.....begins the definition of a procedure of a given name
Endproc.....ends a procedure definition (essential component)
While (<expr> <comp-op> <expr>).....compares two expressions, executes or skips next block based on result
Break.....breaks out of WHILE loop if necessary (optional component)
Endwhile.....ends a WHILE block (essential component)

System Variables - Are used to access rapidly changing data which is automatically being updated by the program as the trains run.

$TIME.....returns current time (h:m:s in 24-hour format), i.e real time, not layout time.
$DATE.....returns today's date (mm/dd/yyyy)
$LAYOUT(prop).....returns layout properties: Name, NTrains, Train <i>
$CAR(label, prop).....returns properties of given car: Loaded, Loadname, Label, AAR
$TRAIN(name, prop).....returns properties of given train: NCars (or Length), Car <i>, Name, Speed, Dir
$SPEED.....returns speed of train in MPH (KPH if metric settings in effect)
$KEY.....returns numeric code of last key hit on keyboard
$X_TRAIN.....returns name of crossing train -- train owning calling script
$X_SPEED.....returns speed of crossing train
$X_CAR.....returns label of crossing car

Appendix A continued:

System Functions - Are used to process the data held in system variables and manipulate the information found.

$RAND(i1,i2).....returns random integer between i1 and i2; defaults 0,100 if not specified
$SUBSTR(i1,n,s).....returns substring of s starting at i1, n chars long; if n = -1, go to end
$FINDSTR(sub,s).....returns position of sub within s, or -1 if not found
$SETTING(name).....returns named registry setting
$RRCLOCK(arg).....rrclock functions: get, set t, add t
$READ(f).....returns contents of text file f as string
$WRITE(f,s).....writes string s to file f; return 1 if succeed, 0 if fail
$DATADIR.....returns TP application data directory path
$STRLEN(s).....returns length of string s
$MSGBOX([type,]s).....shows message s with ok/cancel or yes/no buttons; return 1=ok/yes 0=no
$VIEW(op,name).....gets or sets visibility of named window or feature; op=IsVisible,Show,Hide
$SWITCH(j).....returns switch pos (0 or 1 for standard), -1 if j is not a switch

Appendix B - Key Code Reference Table

0		10		20	Caps Lock	30		40	Arrow Down
1		11		21		31		41	
2		12		22		32		42	
3		13	Enter	23		33	Page Up	43	
4		14		24		34	Page Down	44	
5		15		25		35	End	45	Insert
6		16	Shift	26		36	Home	46	Delete
7		17	Ctrl	27	Esc	37	Arrow Left	47	
8	Backspace	18	Alt	28		38	Arrow Up	48	0
9	Tab	19	Pause/Break	29		39	Arrow Right	49	1
50	2	60		70	f	80	p	90	z
51	3	61	=+	71	g	81	q	91	Windows
52	4	62		72	h	82	r	92	
53	5	63		73	i	83	s	93	Right Click
54	6	64		74	j	84	t	94	
55	7	65	a	75	k	85	u	95	
56	8	66	b	76	l	86	v	96	0 (Num Lock)
57	9	67	c	77	m	87	w	97	1 (Num Lock)
58		68	d	78	n	88	x	98	2 (Num Lock)
59	;:	69	e	79	o	89	y	99	3 (Num Lock)
100	4 (Num Lock)	110	. (Num Lock)	120	F9	130		140	
101	5 (Num Lock)	111	/ (Num Lock)	121	F10	131		141	
102	6 (Num Lock)	112	F1	122	F11	132		142	
103	7 (Num Lock)	113	F2	123	F12	133		143	
104	8 (Num Lock)	114	F3	124		134		144	Num Lock
105	9 (Num Lock)	115	F4	125		135		145	Scroll Lock
106	* (Num Lock)	116	F5	126		136		146	
107	+ (Num Lock)	117	F6	127		137		147	
108		118	F7	128		138		148	
109	- (Num Lock)	119	F8	129		139		149	