Brett's TCL Code

Brett Kennedy
mytrains@tpg.com.au
www.users.tpg.com.au/bsken

June 2009

This booklet started out as a record for my own benefit so that when I go to change a feature or fix a problem down the track I have half a chance of remembering how I set it up in the first place, and why I did it that way. But hopefully it will help you if you want to include any of these features in your own layout. I have purposely used sub routines to do the bulk of the work so that in many cases you should be able to simply copy the sub routines into your own code and then simply change the calling 'When Do' statements to suit your own layout. But everyone's layout is unique so jump onto the http://groups.yahoo.com/group/cti_users/ to ask questions or feel free to email me at mytrains@tpg.com.au

Features incorporated

- Automatic routing of trains – Each train can be allocated a route to be followed and the computer will automatically check and set the turnouts and signals as the train approaches if a route has not already been set manually.
- Trains can also be set as random trains which will cause the PC to 'flip a coin' before routing the train.
- Automatic and realistic braking at red signals.
- If the driver is stopping the train manually at a signal the PC monitors the brake ramp and only intervenes if the train is not slowing fast enough to stop at the signal.
- Trains marked as 'Automatic' also stop automatically at scheduled station stops and auto start and accelerate to cruising speed when signals turn green.
- The operator can take temporary or permanent control of automatic trains using the handheld controllers or the mouse or keyboards. In this mode the PC effectively moves over to the other seat and only intervenes if the driver doesn't respond in time.
- Trains can also be run in 'Manual' mode where driving and route control is left up to the driver and/or controller.
- 'Manual' trains will only start when the engineer increases the throttle but will stop automatically at red signals if the driver does not respond in time.
- The location of each train is displayed on the CTC panel using an icon created from a photo of the model itself. The train's route number is also displayed alongside the image.
- Signals can be 'locked' in the red aspect to prevent the computer clearing a route from that signal and therefore hold an approaching train until the controller unlocks the signal or manually sets a route for it.
- The operator can change the route of each train at any time and can instruct a train to stop and be held at the next station if desired, regardless of which station or which platform it uses.
- Driving trains and control of most features can be performed by mouse clicks and keyboard commands using keyboards/mouse, or via 15 button hand held controllers that interface directly to the PC. The wall mounted monitor is visible from all parts of the layout.
- The handheld controllers include an onboard LED to show the state of the next signal.
- All tracks are bi-directionally signaled, including the double track mainline.
- Phantom blocks have been created to enable shunt locos to enter the same sensor block as another train without the PC losing either train.
- The automation can be easily disabled on an individual train or system wide basis to give the driver full responsibility for their train while another person(s) acts as Train Controller to route trains on their correct paths. In this mode the PC merely tracks the movement of trains and displays them on the CTC screen, and monitors the safety systems.

Layout Overview

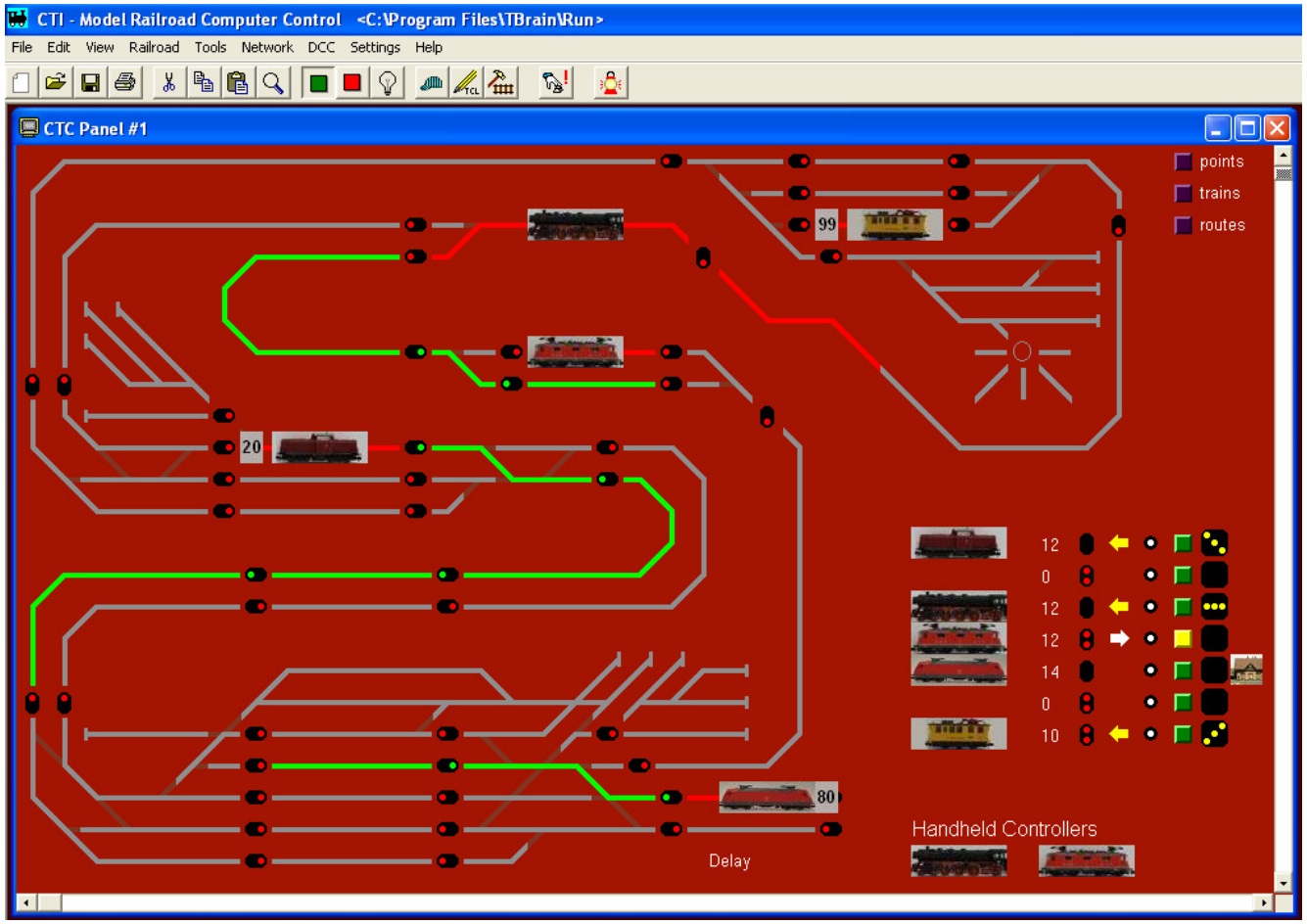
Here's a snapshot of my CTC display. I have chosen not to use the onscreen throttles or Q-Keys so this screen is all that I use. From here I can act as train driver or train controller, or I can sit back and watch as the PC routes trains on their scheduled route; halts passenger trains at stations; and supervises the crossing of trains on the single track sections.

All driving controls and other features can be accessed by clicking on the display or through simple keyboard commands as well as through the handheld controllers to allow full walk around capability.

The Train Properties area at the lower right of the screen displays: (in order from left to right)

- An image of the loco
- each loco's speed (14 throttle notches)
- the current condition of the brake
- the direction of travel
- whether the loco's light is on or off
- the driving mode - automatic or manual
- the route the train will take
- whether it is to stop at stations

When drawing your layout on the CTC screen I recommend you make each of your blocks 5 grid squares long where possible if you want to display an image of the loco and its schedule.



Hardware List

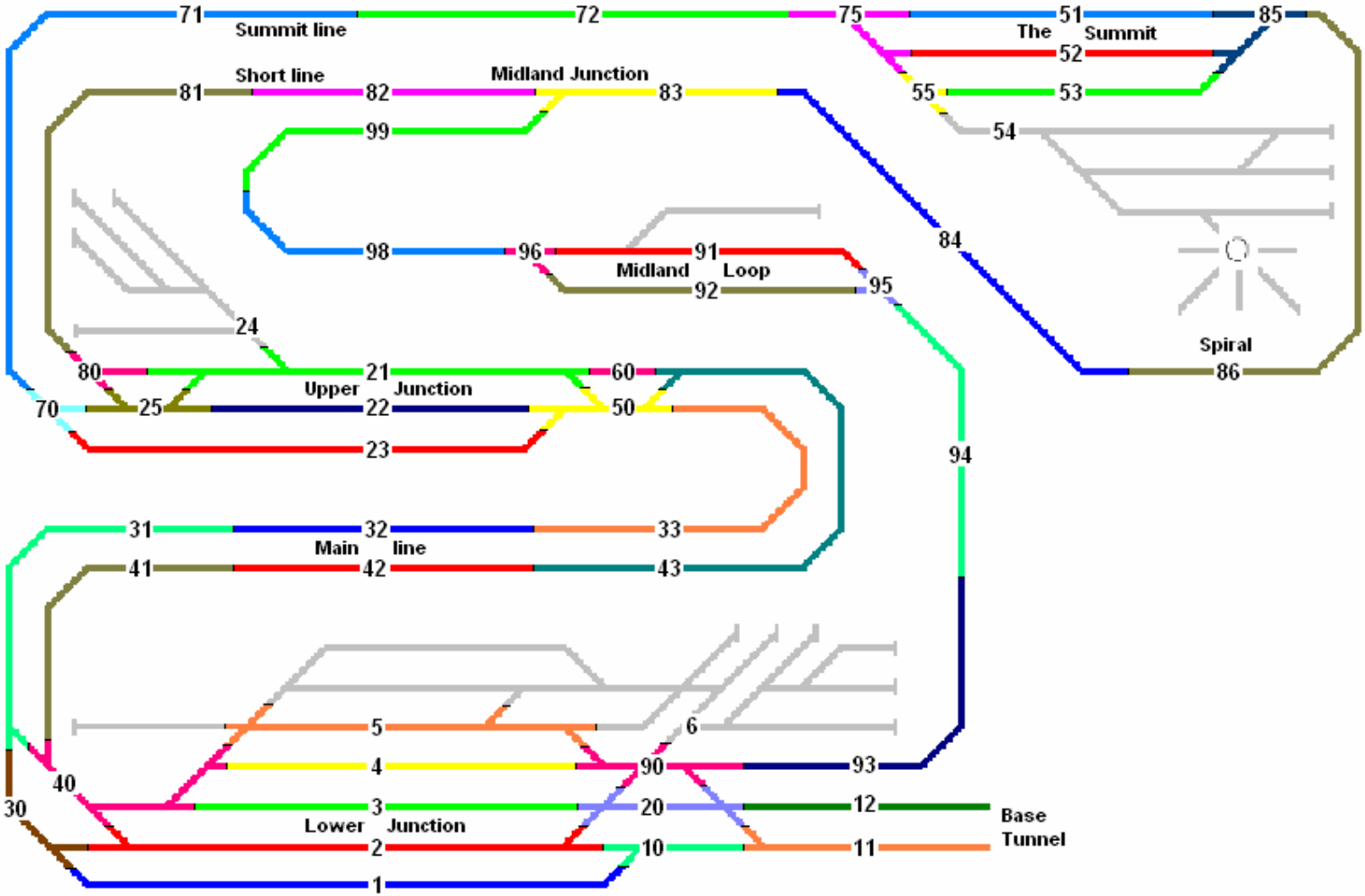
Intellibox IR DCC controller 65 050
PC using Windows XP
USB to serial port adapter
Peco PL10 point motors
Digitrax DS64s for point control
4 CTI Sentry Units
1 CTI Signalman Unit
Home made current detector sensors
1 Digitrax AR1 autoreverser
1 Lenz LK100 autoreverser
2 Home made handheld controllers

My layout consists of 50 track blocks. The plan shows the position of the blocks and approximate location of the insulation gaps for current detecting sensors.

The Base Tunnel (bottom right) is the lowest point of the layout and blocks 11 and 12 are actually connected via a reverse loop inside this tunnel to allow continuous running. One day it may be expanded into a hidden staging area or the layout could expand from this point. (!!!???)

Trains emerging from the Base Tunnel can use any of the 5 platforms at Lower Junction before continuing up the double track Mainline to Upper Junction.

Two single track sections diverge at the Western end of Upper Junction. Trains scheduled as ‘ Clockwise’ will take the Summit Line (blocks 70 to 72) on the more direct route to the Summit, while ‘Anticlockwise’ trains will head off through blocks 81 and 82 to Midland Junction, then up the Spiral to enter The Summit from the South.



Trains marked as ‘Midland’ will branch off at either Midland Junction or Lower Junction and take the single track line via Midland. All this leads to plenty of variety as the operator or the computer deals with the correct routing of trains and the numerous meeting of trains on the single track sections.

TCL Code Overview

Hopefully this guide will help you as you work on your TCL code. Most of the features on my layout revolve around one or more sub routines that do all the work and then very simple 'When Do' statements that pass the necessary info on to the sub routines. If you want to copy a feature into your own code, in many cases you will be able to simply copy the relevant sub routine across to your own file and then edit the 'When Do' statements to suit your own layout. Shortly we'll go through each of these features one by one but before we do here's a quick overview of some of the key variables that crop up regularly throughout the script. You'll need a general understanding of how these inter relate before we go much further.

To use many of the following features you will need to set up each of these arrays of variables on your own layout.

Block Variables B[100]

Each track block has its own Block Variable and it is these block variables that trigger many of the features and form the backbone for much of the code. The values for blocks are:

0=vacant,	3=waiting to clear up direction,	6=waiting to cancel down direction,	11=occupied by an up train,
1=cleared up direction,	4=waiting to clear down direction,	7=block reserved for shunting	12=occupied by a down train
2=cleared down direction,	5=waiting to cancel up direction,	10=occupied but no direction specified,	

Block variables have been created in the array B[100] and therefore are in the range B[0] to B[99]. See the notes on the next page re Numbering Conventions for some advice on how big to make the arrays.

Loco Variables L[100]

Each track block also has its own Loco Variable. These loco variables are also set up in an array and indexed to the same value as the block variable. i.e. L[51] points to the loco currently on B[51]. Using a common index number makes it extremely easy to check the state of the relevant block and identify the loco currently occupying the block, but the real advantage is the ability to check other linked variables such as this loco's schedule and mode variables that provide the program all it needs to know to automatically route and control this train.

CTI have developed the beacon feature to allow the PC to control the train on a particular block without needing to know which train it is, however for us to route individual trains differently we need to know which train is approaching and which way it wants to go. This means the program needs to constantly check not just which train is on the block but also that train's route, schedule and driving mode.

The program's inbuilt beacon feature does not easily link to other variable arrays (beacon arrays are ordered alphanumerically rather than numerically) so I use my loco variables instead of beacons, and by using the loco's CTI address as the value it means these variables operate in exactly the same way as a beacon. I.e. *L[51].brake = on applies the brake to whichever train is currently on Block 51. However through the use of indexing these loco variables also provide direct access to all the information about this train including route, schedule, stopping details and if I want to in the future even train length, maximum speed, type of train and priority etc. In this way the block and loco variables work together to provide all the necessary information for the program. And all this info can be passed to the sub routines through just the index number of the block.

Control Variables Control[100]

In addition, each block also has a Control Variable which is used to manage additional features. The use of the Control Variables depends on the type of block;

- for platform blocks the control variables store information about the current train to ensure it stops at the station if required and takes the correct route on departure
- for blocks at the end of each line the control variables are used in the interlocking to prevent trains being routed onto a line currently cleared in the opposite direction
- for blocks that form part of a station throat the control variables store the current path through this series of turnouts. (More about these later on)

Schedule Variables Schedule[10]

Each loco has its own Schedule Variable which stores information on which route this train is to take and whether it is to stop at stations. The values for my schedule variables are:

0=clockwise,	3=random, will take clockwise option at next junction	+10 if train is to stop at stations, i.e. values between 10 and 15
1=anticlockwise,	4=random, will take anticlockwise option	+20 if train is to be held at next station, i.e. values 20 to 25 or 30 to 35
2=midland,	5=random, will take midland line	

The operator can change each train's schedule at any time or leave the train as random. If the train is random the PC will set the train's schedule to either 3, 4 or 5 and change it each time the train passes a junction to provide maximum variety. (The Layout Overview section gave a brief explanation of the possible routes).

Interaction between variables

So how do our sub routines access all this information from just the index number of the block? Let's look at the first portion of the AutoBrake sub routine as an example. As well as stopping trains at red signals this sub routine also ensures trains stop at stations if scheduled to do so. In order to do this it needs to know:

1. The value of the relevant block variable
2. The loco currently occupying this block
3. The value of the schedule variable for this loco
4. The current speed of this loco (to provide realistic braking at the signal)

All of this information can be passed to the sub routine by simply sending the index value for this block as a parameter. Let's say that our train has just entered Block 21 heading in the down direction. If that is the case Block 21 will have been set to 12 (for a down train) by the train movement section of the script and the following code will then call the AutoBrake sub routine to check if the train needs to stop for either a red signal or a scheduled station stop. The index value of 21 is passed to the sub routine as the first parameter, the coordinates of the signal and momentum settings for this location would also need to be sent through but we'll leave them out for now.

```
When B[21] = 12        Do      AutoBrake (21, …)
```

```
SUB AutoBrake (index, … block, loco, schedule, speed)
Block = B[index],                          'sets Block = to the value of the relevant block
Loco = L[index], Loco = 2724-, Loco = 13/, 'sets Loco = to this loco's number.  0,1,2,3, etc.
Schedule = Schedule[Loco],                 'copies this train's schedule to local variable
Speed = Speed[loco]
… the sub routine continues by checking these variables and applying the brake if necessary.
```

The sub routine contains four local variables which are used solely within this sub routine (block, loco, schedule and speed). These variables are set up as local copies of this train's variables which can be manipulated as required without changing the originals. The first line sets 'Block' equal to the current value of B[index] which in this case is B[21]. The next line needs to go a bit further in order to identify the index number for this loco. Firstly the variable 'Loco' is set to equal the value stored at L[21]. Now remember that we are storing the loco's CTI address on the Loco Variables so 'Loco' will now equal the address of this loco. This is a 4 digit number in the range 2724, 2737, 2750 etc. so we need to break this number down to the pure loco number that we can use as the index number for our schedule and other variables, i.e. we need to get it down to the correct value between 0 and 9. (If you have more than 10 locos you would simply increase the size of the arrays to accommodate your fleet)

Each loco in the DCC fleet roster has a permanent memory location allocated when the loco is first created. The first loco created will be stored at the CTI address 2724, and each subsequent loco will have an address 13 higher (the intermediate addresses are used to store each train's direction, speed, brake, momentum and functions as per the right hand table. Appendix A also has more information on CTI addresses.)

So to strip this down to our pure loco number we initially subtract 2724. This reduces the range to 0, 13, 26, 39 etc. Then we simply divide by 13 to give us the required range of 0, 1, 2, 3 etc.

Loco number	**CTI address**
1st loco created	2724
2nd loco	2737
3rd loco	2750
4th loco	2763
5th loco	2776
6th loco	2789
7th loco	2802

Loco function	**CTI sub addresses (for 1st loco)**
Speed	2724
Direction	2725
Brake	2726
Momentum	2727
Light	2728
F1	2729
F2 to F8	2730 to 2736

Now pointing to the correct schedule or other variable is a cinch, we simply use this loco number as the index. i.e. Schedule = Schedule[Loco] will copy the schedule for this train onto the local variable 'Schedule'. The same can then be done for any other variables needed such as Speed, Mode etc. and the sub routine can then perform whatever checks and functions it needs to.

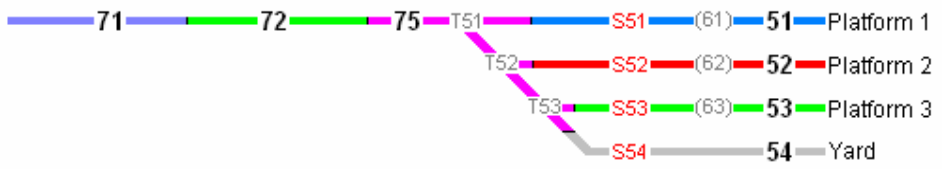
Block Numbering Conventions

When you come to planning your block numbering the only rule is that your platforms need to be numbered consecutively to use some of the sub routines that rely on indexing. This is only the case within each station, for example my main station's platforms are blocks 1 to 5. Upper Junction uses blocks 21 to 23 and The Summit uses 51 to 54. Shunt Blocks also need to be consecutively numbered, but because you don't need to ever use these numbers again just find a cluster of spare ones and use them. Other than this you can use any numbering convention that makes sense for your layout.

Don't forget to allow for Shunt Blocks when determining the size of your arrays. Shunt Blocks are used to allow a second loco to enter an already occupied platform without the PC losing track of either train. Shunt Blocks don't have any hardware such as sensors or track but merely exist in software.

So if your layout has 20 blocks including 6 platforms your array will need to allow for at least 32 blocks (20 + 2 x 6). But I'd round it up to at least 50 to give you more flexibility in numbering. If you plan to use one or more of my handheld controllers they also use another index number for each handset. My layout only has 50 physical track blocks but I have set my arrays at 100 to give me plenty of flexibility. (Appendix B shows how I have allocated the numbers within my arrays)

I also find it much easier to remember signal numbers and turnout numbers if I keep their numbers similar to the track blocks they are connected to, for example in the diagram to the right Turnout 51 (T51) and Signal 51 (S51) are at the end of Block 51 (which is platform 1 at The Summit). When I got to the end of my numbering 61 to 63 were free so I have used them as the Shunt Blocks at this end of the platforms.



To set routes from your keyboard or handsets you'll also need to give each of your routes a number that you can easily remember, but there's no need to change these if you already have route numbers you like and can remember. I have given my routes a number based on block numbers. The first part of the route number is the originating track and the second part of the number is the destination track. For example in the diagram above the route from Platform 1 out across Block 75 would be Route 175. The reverse route from Block 75 to platform 1 would be Route 751. For this reason I like to use numbers ending in 0 for my throat blocks (i.e. 10, 20, 30, 40 etc.) because it makes route numbers nice and easy to remember, but in this case 70 had already been used as the throat block at the other end of block 71, and block 50 had also been used elsewhere so 75 was the next best thing.

Now that you have an overview of the layout and the code the rest of the script should be fairly easy to follow as you work your way through each of the features one at a time, or pick the ones that you are most interested in.

TrainBrain requires that all sub routines be listed at the beginning of the script however for convenience in this booklet each sub routine is listed alongside the appropriate section of the script. My TCL script is broken up into the following sections; these sections also form the chapters for the rest of this booklet:

Section 1 Description of Sensors, Constants and Variables

This section lists the sensors, user defined constants and variables used throughout the code. The sensors must be listed in the order they connect to the CTI Sentry Units. Those sensors followed by a '#' symbol are current detectors. Those followed by '*~' are infrared detectors set to indicate occupied when the phototransistor is off – i.e. the beam is broken. Sensor HH11 is Handheld 1 button 1. HH23 is handheld 2 button 3. Sensor S1 is the sensor for block1 etc.

Sensors:	HH11,	HH12,	HH13,	HH14,	HH15,	HH16,	spare,	spare,	spare,	spare,	HH21,	HH22,	HH23,	HH24,	HH25,	HH26,
Signals:	LED1(2),		LED2(2),		spare(12)											
Sensors:	S1#,	S2#,	S3#,	S4#,	S5#,	S81#,	S31#,	S30#,	S40#,	S41#,	S25#,	S70#,	S80#,	S71#,	S82*~,	S99*~,
	S10#,	S20#,	S90#,	S21#,	S22#,	S23#,	S50#,	S60#,	S34#,	S44#,	S32*~,	S42*~,	S96*~,	S11#,	S12#,	S83#,
	S51#,	S52#,	S53#,	S72#,	S75#,	S55#,	S85#,	S86#,	S84#,	S93#,	S33*~,	S43*~,	S95*~,	spare,	spare,	spare,

Constants: Reversed = on, Normal = off, Occupied = true, Vacant = false, Grey = $RGB_8F8F8F, Green = $RGB_FF00, Yellow = $RGB_FFFF, Blue = $RGB_FFFF00

Variables:	The TCL Code Overview above has more info on how the variables work together.
B[100]	Block Variables store the current state of the track block.
L[100]	Loco Variables store the CTI address of the loco currently on this block. I use these instead of the inbuilt Beacons
Throat[100]	Throat Variables store the current path through a series of turnouts forming a station throat. (See section 15 on train movement)
Schedule[10]	Each train has its own schedule variable to control route selection and station stops
Mode[10]	Each train also has its own mode variable to mark a train as Automatic or Manual
Speed[10]	Train Speed variables are used to display each train's speed on the CTC screen. (See section 19 for details)
Index	Is only used where arrays need to be managed outside of sub routines. All sub routines have their own local index variable to remove any conflict problems
B0	Is a phantom variable that always equals 0 and is used for passing blank parameters to sub routines (this has proven more reliable than passing a 0).

Each route also has its own variable; these are used to request and manage routes. (See Sections 1-8) The route numbering system is explained in the following chapter. The routes are grouped together to make it easier to cancel all routes that cross a certain track block. The labels such as 'Midland_Junction' have no purpose other than to make it easier to find the route you are looking for in the Variables Window. The acceptable values for routes are: 0=idle, 1=set, 5=held until signal unlocked, 6=called by PC, 8=called manually

{Block 83 Routes}	Midland_Junction	R848,	R849,	R884,	R984													
{Block 75 Routes}	Summit_West	R751,	R752,	R753,	R754,	R175,	R275,	R375,	R475									
{Block 85 Routes}	Summit_South	R185,	R285,	R385,	R186,	R286,	R386,	R851,	R852,	R853								
{Block 95 Routes}	Midland_East	R951,	R952,	R195,	R295													
{Block 96 Routes}	Midland_West	R961,	R962,	R196,	R296													
{Block 70 Routes}	Upper_West_70	R701,	R702,	R703,	R170,	R270,	R370											
{Block 80 Routes}	Upper_West_80	R801,	R802,	R180,	R280													
{Block 21 Routes}	Upper_Yard	R14,	R41															
{Block 50 Routes}	Upper_East_50	R501,	R502,	R503,	R150,	R250,	R350,	R602	R603,	R260,	R360							
{Block 60 Routes}	Upper_East_60	R601,	R160															
{Block 30 Routes}	Lower_West_30	R301,	R302,	R130,	R230													
{Block 40 Routes}	Lower_West_40	R402,	R403,	R404,	R405,	R240,	R340,	R440,	R540,	R303,	R304,	R305,	R330,	R430,	R530			
{Block 10 Routes}	Lower_East_10	R101,	R102,	R110,	R210													
{Block 20 Routes}	Lower_East_20	R202,	R203,	R902,	R903,	R62,	R63,	R310,	R103,	R410,	R510,	R220,	R320,	R420,	R520,			
{Block 90 Routes}	Lower_East_90	R104,	R105,	R204,	R205,	R904,	R905,	R64,	R65,	R290,	R390,	R490,	R590,	R26,	R36,	R46,	R56	

RoutePointer The variable RoutePointer is used when canceling all pending routes (see section 22 – System Reset Commands)

Accurate braking at signals is performed through 4 Brake Ramps, each with its own variable (Section 14) Ramp1 Ramp2 Ramp3 Ramp4 Brake

The Base Tunnel is where trains exit the layout heading for destinations beyond this little fictitious world, however in reality a train entering the base tunnel turns on a hidden reverse loop and reenters the layout from the same tunnel. In the interest of variety a random timer can be used to hold up to 2 trains in the tunnel before reentering. (See the end of Section 4)
DelayOn Delay

Handheld controllers allow the operator to drive trains, set and cancel signals and operate various other features. The script uses the variables below to manage input from the handhelds. (See Section 23 for details) HH[10] RouteEntry SignalEntry

The script for calling routes using the keyboard or handsets is extremely simple as shown below. Each route has a unique number which is made up of 2 parts. The first part is the originating track and the second part is the destination track.

For example the route from Lower Junction's Platform 1 to Line 30 is 130. Platform 2 to Line 40 is 240 etc. The reverse route from Line 40 to Platform 2 is 402. These codes can also be entered from the handheld controllers, in this case the variable RouteEntry is set equal to the route number hence the statement "or RouteEntry =". Each route is prefaced by an 'R' in the script so the full line of script simply reads:

2.....…70

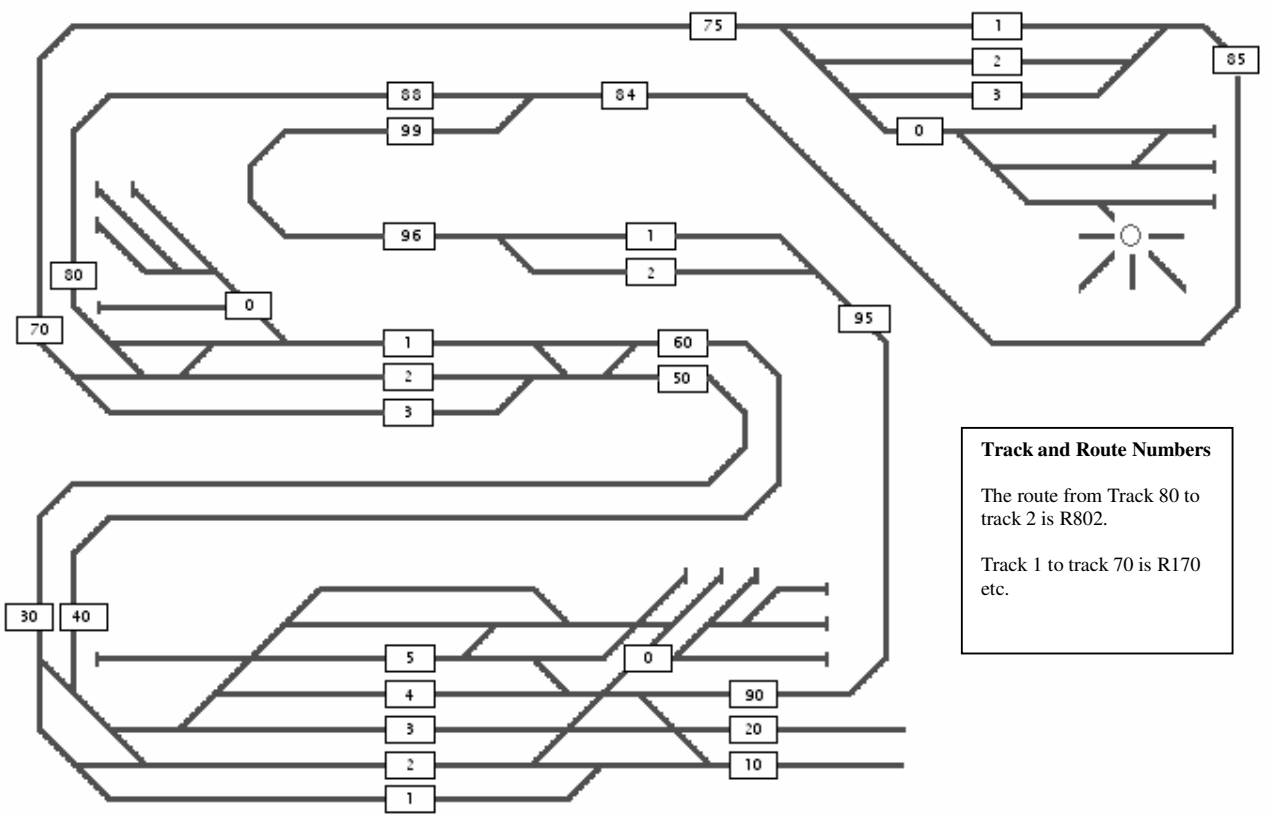
Platform 2 to Line 70

```
When $command = 270   or   RouteEntry = 270   Do   R270=8
When $command = 280   or   RouteEntry = 280   Do   R280=8
```

This repeats for all routes on the layout.

The route's variable is set to the value of 8 whenever a route is manually requested.

Once a route has been set to 8 the 'Route Activation' section of the script checks the interlocking for each route and activates the route if safe to do so. (See section 5)



Track and Route Numbers

The route from Track 80 to track 2 is R802.

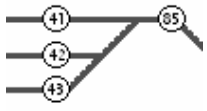
Track 1 to track 70 is R170 etc.

Section 3 Manual Route Setting by Mouse Click

Routes can also be called by clicking on a signal on the CTC screen and then clicking on the next signal along the chosen route. When an idle signal is left clicked the script checks to see if a neighbouring signal has already been selected as a starter signal. If so the appropriate route is set to the value '8', if no neighbouring signals have been marked as a starter signal then the clicked signal is marked as a starter signal by marking it as "YR" or "RY" (This will appear on the CTC screen as Yellow over Red). If a neighbouring signal is then clicked the route will be set from the first signal to the second signal.

But first an explanation about signals. All signals displayed on my CTC screen are 2 aspect signals. Each signal can display the following aspects:

East or South facing signal	West or North facing signal	
"G-"	"-G"	Green – train will proceed on the set route. The dash indicates that the second lamp is unlit.
"-R"	"R-"	Red – brake will automatically apply and the train will pull up gently in front of the signal
"RR"	"RR"	Double red signifies the signal is locked at red to prevent the PC from clearing this signal.
"YR"	"RY"	The signal has been selected as a starter signal and is waiting for a valid destination signal to be selected.



So the code for this track layout below would be:

```
When $Left_Mouse = {Sig 41} (31,1,1)    Do If   $Signal {Sig85} (36,3,1) = "YR"   Then   R851=8
                                        Else    $Signal {Sig41} (31,1,1) = "RY"   Endif
When $Left_Mouse = {Sig 42} (31,2,1)    Do If   $Signal {Sig85} (36,3,1) = "YR"   Then   R852=8
                                        Else    $Signal {Sig42} (31,2,1) = "RY"   Endif
When $Left_Mouse = {Sig 43} (31,3,1)    Do If   $Signal {Sig85} (36,3,1) = "YR"   Then   R853=8
                                        Else    $Signal {Sig43} (31,3,1) = "RY"   Endif
When $Left_Mouse = {Sig85} (36,3,1)     Do If   $Signal {Sig41} (31,1,1) = "RY"   Then   R185=8
                                        ElseIf  $Signal {Sig42} (31,2,1) = "RY"   Then   R285=8
                                        ElseIf  $Signal {Sig43} (31,3,1) = "RY"   Then   R385=8
                                        Else    $Signal {Sig85} (36,3,1) = "YR"   Endif
```

The first line of the example can be read as: When the operator left clicks on Signal 41 (at coordinates 31,1,1) if Signal 85 has already been chosen as a starter signal (YR) then set route 851 equal to 8. Otherwise (if the adjacent signal has not been selected as a starter signal) set this signal (Signal 41) as a starter signal (RY). This then repeats for signals 42 and 43.
In the remaining lines if the operator clicks on Signal 85 the program checks if Signal 41 has already been selected, if it has then route 185 is set to 8, otherwise if Signal 42 has been set as a starter then route 285 = 8, the same occurs for Signal 43, but if none of the adjacent signals are set as a starter signal then the program will set Signal 85 as a starter signal.

The only signal that does not follow the standard code is Signal 24 below. This is the yard signal at Upper Junction and activates the route between platform 1 and the yard as soon as it is selected. If there is currently a loco in the yard the outgoing route (R41) is set, otherwise R14 is set for a train to move into the yard from platform 1.

```
When $Left_Mouse = {Sig 24} (7,9,1)    Do If   B24 < 10    Then   R14 = 8  Else   R41 = 8  Endif
```

Section 4 Automatic Route Setting

Programming Principles:

1. When an automatic train approaches a red signal the PC will attempt to set the correct route ahead of it. If only 1 route exists from that signal the route variable will be given the value of 6. The Route Activation section of the script then tests the route's interlocking and if available will activate the route.
2. If there are diverging routes from this signal the PC must check the train's schedule to determine which route to use. If multiple routes are viable for this train - such as when approaching a multi-platform station – all viable routes are given the value of 6. The Route Activation section of the script then selects one of these routes and cancels the others.
3. The operator can lock a signal to prevent the PC auto setting a route from it until the signal is unlocked. This is a handy way of ensuring certain trains receive priority at a junction, or to hold a train in a loop to wait for another train. A signal can be locked by right-clicking on an idle signal. A locked signal will appear on the CTC screen in the double red aspect (depicted as RR in the script). In this case the routes are still selected however they are given the value of 5 rather than 6. When the signal is unlocked the route value automatically changes to 6 to allow the Route Activation section of the script to determine the most appropriate route and activate it when available.
4. The operator can also tag a train to be held at the next station. When such a train arrives at a station no departure route will be established and the train will stop in the platform and not restart until the signal is manually cleared for it.
5. When a train is arriving at a platform the PC will check if that train is scheduled to stop at the station. If it is scheduled to stop then the outgoing route is not cleared until after the station dwell time has passed. Even if the departure signal is manually cleared the train will be held until the dwell time has passed.
6. If the train is not scheduled to stop at the station the PC will attempt to set the outgoing route as soon as the approaching train enters the station limits. By waiting until the train is almost in the platform it allows maximum time for the operator to manually route the train if desired but still allows sufficient time for the train to safely and realistically pull up at the signal if a route is not available.
7. The route the train is scheduled to take is displayed alongside the train's image on the CTC screen to allow the controller to plan ahead.
8. If the operator has marked the train as a random train then the PC will randomly select a route for it as early as possible and display the chosen route as above.
9. The automatic route setting feature has been designed in such a ways as to allow Shunt Locos to bustle backwards and forwards without auto-setting unwanted routes.
10. The automatic route setting feature can be turned off for all trains or individual trains to give total control to the operator.

There are a couple of sub routines used for automatic route setting. The first we're going to look at is the CallRoute sub routine which is called as trains approach Home Signals to allocate them to an available platform.

Controlling Home Signals (The CallRoute subroutine)

In the example below a down train has entered block 81 (B[81]=12) and is therefore approaching the Home Signal at the Western end of Upper Junction. The block index number and the coordinates of the signal ahead of the train are passed to the subroutine CallRoute along with the possible routes from this signal, in this case Routes 801 and 802. (note: B0 does not exist but is used whenever there are fewer variables to send than the number of parameters expected by the sub routine. B0 always equals zero so this is a simple way of sending a dummy number to the sub routine and seems more reliable than using a zero)

```
When B[81]=12,        Do    CallRoute (81, (2,8,1), &R801, &R802, &B0, &B0)
```

The CallRoute sub routine (below) begins by setting the local variable 'Loco' equal to the value of the Loco Variable for this block – i.e. the CTI address of the loco approaching the signal. It then reduces this number to the index number for this loco. i.e. 0, 1, 2 ,3 etc. (For more on this refer to the TCL Code Overview Section). It then sets the local variable 'Mode' equal to this loco's mode variable. If the train is a manual train (mode = 1) or any of the possible routes are already set or pending then the sub routine ends without any action taken.

The parameters for the CallRoute sub routine are:

1. the index number for this block
2. the coordinates of the signal immediately ahead of the train

3-6 up to 4 possible routes from this signal

7-8. are local variables called 'index' and 'mode' used only within this subroutine

On my layout most platforms are available for all trains but there are a couple of exceptions; namely platform 1 at Lower Junction which is not available for trains heading up the Midland Line, and Platform 3 at Upper Junction which is not available for Anticlockwise trains. So we will need to check the schedule for these trains at those locations. For this reason the sub routine sets 'schedule' equal to the schedule variable of this loco and since we don't need to know at this stage if it is a stopping train we can simplify things by reduced it to a single digit number if it is currently > 10. The next line then reduces it even further until it is 0, 1 or 2 (for clockwise, anticlockwise, and midland bound trains respectively).

If the signal is locked the variable 'Value' is set to 5 otherwise it is set to 6. This value will be copied to all viable routes at the end of the sub routine.

```
SUB CallRoute (index, coordinates, Route1, Route2, Route3, Route4, loco, schedule, mode, value)
Loco = L[index], Loco = 2724-,     Loco = 13/,                                          'sets Loco = to loco #       0,1,2,3,4,5, etc.
Mode = Mode[loco]                                                                       'copy Train's mode to local variable
If       Mode = 1 or *Route1>0 or *Route2>0 or *Route3>0 or *Route4>0                    'if train is a manual train or route
Then     Return    Endif                                                               'already set, return without setting a route
Schedule = Schedule[loco],          Until Schedule < 10                                 'set local variable = to train's schedule
                                    Loop    Schedule = 10-    Endloop                   'and reduce to a single digit number
If Schedule > 2                     Then    Schedule = 3-     Endif                     'further reduce Schedule to 0,1 or 2
If $signal(coordinates)="RR"        Then    Value=5 Else      Value=6 Endif             'if signal is locked set value to 5 else 6
If index = 33 or index = 43         Then    If Schedule = 1                             'for anticlockwise trains on blocks 33 or 43
                                    Then    Route3=&B0                                  'delete route3 as an option (B0 always = 0)
                                    Else    Route1=&B0        Endif                     'otherwise delete route1 as an option
ElseIf index = 31                   Then    If Schedule = 2                             'if midland train on block 31
                                    Then    Route1 = &B0      Endif Endif               'then delete route1 as an option
*Route1=Value, *Route2=Value, *Route3=Value, *Route4=Value,                             'set remaining routes to 3 or 5
B0=0       ENDSUB                                                                       'and reset B0 to ensure it always equals 0
```

If the train is approaching Upper Junction from either the up or down main (i.e. it is on block 33 or 43 so index = 33 or 43) then if it is an anticlockwise train (schedule = 1) Route3 is replaced by our dummy variable (B0) to ensure that the train is not routed onto platform 3. If it is not an anticlockwise train we remove Route1 as an option due to the awkward track layout from Platform 1 to the Summit Line. The following line provides similar checks for Midland bound trains arriving at Lower Junction from block 31 to ensure they are not routed onto platform 1. Finally all viable routes are called by setting them to either 6 or 5. We will look at how the sub routine 'AutoSet' selects one of these routes and cancels the others later in the Route Activation section. Our phantom variable B0 is also reset to 0 at the end of the sub.

Controlling Starter Signals (The Platform subroutine)

So following our train from the previous example, the CallRoute sub routine will by now have called each of the routes into the station and the Route Activation section of the script will have chosen and activated one of those routes once safe to do so. So our train is now passing the home signal and is entering Block 80 (the station throat) on its way into one of the platforms. The program now needs to check whether this train is scheduled to stop at the station and which route it is to take when leaving. The Platform sub routine does this for us. The role of the Platform sub routine is to interrogate the train's schedule and set the Platform's control variable in response. The control variable is then used to ensure the train stops and waits at the station until any station stop time has expired and then sets the correct outgoing route. At the end of this sub routine the platform's control variable will be set to one of the following:

1. Up train ready to depart on Option 1
2. Up train ready to depart on Option 2
3. Down train ready to depart on Option 1
4. Down train ready to depart on Option 2
10. Manual train stopped - no route to be set
11. Up train stopped at station (will take Option 1)
12. Up train stopped at station (will take Option 2)
13. Down train stopped at station (will take Option 1)
14. Down train stopped at station (will take Option 2)

The code used to call the Platform sub routine simply takes the form: `When B80=12 Do Platform (80)`

The only parameter that we need to pass to the platform sub routine (below) is the index number of the block the train is currently on, i.e. the throat block on approach to the platform, however there are also 7 local variables used within this sub routine. The sub starts off by setting the local variable 'loco' equal to the loco number for this train. 'Block' is set to equal

the current value of the block variable; this tells us whether the train is traveling in the up or down direction. The sub also sets 'platform' equal to the control variable for this block, which – as it is a throat block - means it will equal the index number of the platform that the points are currently set for. (See the TCL code overview section for more info on Control Variables). The next step sets 'platform' equal to the address of the control variable for the relevant platform. This is the address where we will finally copy the correct value at the end of the sub routine. Following this 'schedule' and 'mode' are also set to equal this train's variables.

```
SUB Platform (index, loco, block, platform, schedule, value, stopping, mode)
Loco = L[index], Loco = 2724-, Loco = 13/, 'sets Loco = to loco #.  0,1,2,3,4,5, etc.
Block = B[index]                                        'sets Block = to the value of this block (11 = up train, 12 = down)
Platform = Control[index],                              'set platform to equal the block number of this platform (the control variable
                                                        'for Throat Blocks is equal to the platform number the points are set for)
Platform = &Control[platform]                           'then set platform = address for this platform's control variable.
Schedule = Schedule[Loco]                               'copy Train's schedule to local variable
Mode = Mode[Loco]                                       'copy Train's mode to local variable
If      Schedule > 19   Then    schedule[Loco]=20-,     'if train is to be held at the station reduce the schedule
        ScheduleDisplay(Loco)   Return Endif            'variable by 20, and redraw the schedule display. Then Return
If      Schedule > 9    Then                            'if the train is a stopping commuter train then
        If      Mode=1 Then     *Platform = 10, Return  'if manual train set the platform's control variable to 10
        Else    Schedule = 10-, Stopping = 1            'if auto train reduce schedule to <10 and set stopping = 1
Else    Stopping = 0    Endif   Endif                   'if the train is an express train set stopping = 0.
If      Mode = 1                Then    Return Endif    'if this a manual train the sub can now end
If      Schedule > 2    Then    Schedule = 3- Endif     'if the schedule is >2 strip a further 3 off (now = 0, 1 or 2)
If      Schedule = 0    Then    Value = 1               'if clockwise train set value to 1
Elseif  Schedule = 1    Then                            'if anticlockwise train
        If      index = 50 or index = 60                'then if train is on block 50 or 60
        Then    Value=2 Else    Value=1 Endif           'set value = 2. For trains on any other block value = 1
ElseIf Schedule = 2     Then                            'if midland train
        If      index = 50 or index = 60                'then if train on block 50 or 60
        Then    Value=1 Else    Value=2 Endif   Endif   'set value = 1. For trains on any other block value = 2
If      Stopping = 1    Then    Value = 10+     Endif   'if it is a stopping train increase Value by 10
If      Block=12 or index=75    Then  Value=2+  Endif   'if it is a down train or train is on block 75 add another 2
*Platform = Value,                      ENDSUB          'copy value to the control variable for this platform
```

The first If .. then … statement checks if the train is due to be held at the next station (Schedule > 19). If it is the schedule of this loco is reduced by 20 to return it to what it was before being set to stop, the ScheduleDisplay is called to redraw the schedule display in the train properties area and the sub routine ends as no route needs to be set.

The second If…then… statement checks if the train is due to perform a station stop (schedule > 9), if it is and it is a manual train (mode = 1) then the platform variable is set to 10 to ensure the station dwell time is observed and the sub routine ends as no outgoing route needs to be set. If the train is an auto train then 'stopping' is set to 1 and the local 'schedule' variable can now be reduced by 10 to simplify later equations. If the train is not due to stop 'stopping' is set to 0. We will refer to the stopping variable again at the end of the sub.

The next line simply checks if the train is a manual train and if it is we don't need to take any further action so the sub routine ends.

This sub routine doesn't need to know whether or not the train is a random train so we can reduce the local schedule variable even further until it is either 0, 1, or 2. Now if the train is a Clockwise train (schedule = 0) we set 'Value' equal to 1 to indicate this train will take option 1. If it is marked as an anticlockwise train (schedule = 1) then 'Value' will normally be set to 1 however if the train is approaching Upper Junction on blocks 50 or 60 then 'Value' needs to be set to 2 to ensure this train takes the turnout onto the ShortLine. The following line does likewise for trains whose schedule equals 2.

Now we're nearly finished and simply need to add 10 to 'Value' if it is a stopping train, and add another 2 if it is traveling in the down direction (block = 12). We need to treat block 75 as though it is a down block also as an up train on block 75 automatically becomes a down train as it enters the platforms at The Summit, hence the inclusion of 'Index=75'. By now 'Value' will have been set to one of the numbers shown in the table at the start of this section so we simply copy it onto the control variable for this platform block, and we're done. The actual testing and setting of the outgoing route is performed by the AutoRoute subroutine below, but only after the following script ensures that any scheduled station stops are observed first.

Monitoring Station Dwell Times

So our train has now entered Platform 1 at Upper Junction and the Platform Sub has set the control variable for this platform according to the train's schedule. The next part of the process is to ensure that any train scheduled to stop at the station does so for a predefined dwell time. This is easily achieved by delaying the outgoing route until after the dwell time has passed. This is done through the code below which waits 30 seconds and then subtracts 10 from the Platform's control variable. This tells the program the train is now ready to depart and causes the AutoRoute sub routine to call the correct route for this train (see below for a description of the AutoRoute sub). For manual trains the platform variable will be returned to 0 after the dwell time and thereby not call the AutoRoute subroutine, however the colour of the track block on the CTC screen will change from yellow to green to tell the driver/controller that the train is ready to depart.

```
When Control[21] > 9        Do      Wait 30         Control[21] = 10-
```

With no outgoing route set the train will be stopped by the Train Braking section of the script but it is also important that the train does not restart until the dwell time has elapsed, even if the signal is manually cleared for it. Therefore the additional clause 'Control[21]<10' is inserted in the Train Acceleration code below to ensure the train does not auto start until the platform stop timer has finished. (see the section on Train Braking and Acceleration Controls for details)

```
When BU1 = 11 and $signal (7,10,1) = "G-", Control[21] < 10        Do      … 'apply the brake
```

Calling the Outgoing Route (The AutoRoute subroutine)

So the train we've been following through the Automatic Route Setting section of the script has now entered Platform 1 at Upper Junction and performed any necessary station stop. Being a down train Platform 1's control variable will now be equal to 3 or 4 which indicates the train is ready to depart and triggers the sub routine AutoRoute, this sub routine will call the correct route for the train. Of course if our train is an express train the Platform's control variable will have been set to 3 or 4 immediately which means the outgoing route will be called as soon as the train enters the station limits and if the route is available the train will not need to stop. The AutoRoute sub routine is called using the code below. This example is for platform 1at Upper Junction (block 21). When the platform's control variable has been set (>0) and the train is ready to depart (<10) it calls the AutoRoute sub.

```
When Control [21] > 0, Control[21] < 10     Do      AutoRoute ((7,12,1), (13,12,1), &Control[21], &R170, &R180, &B0, &R160, &R160, &R150, &B0)
```

The parameters for this sub routine are:

1. the coordinates of the Up Starter Signal
2. the coordinates of the Down Starter Signal
3. the control variable for this platform
4. the route to use for an Up train taking option 1
5. the route to use for an Up train taking option 2
6. any non automatic Up route
7. the route for a Down train taking option 1
8. the route for a Down train taking option 2
9. any non automatic Down route
10. any additional non automatic Down route
11. a local variable called 'Value'

Note that often Option 1 and Option 2 trains will both take the same route, on such occasions the same route

```
SUB AutoRoute (coordinatesUp, coordinatesDn, Platform, Up1, Up2, Up3, Dn1, Dn2, Dn3, Dn4, value)
If *Platform < 3  Then    If $signal(coordinatesUp) = "RR"                  'if it is an up train then if the up signal is locked
        Then    Value=5         Else    Value=6         Endif   'set value equal to 5. If not locked set value = to 6.
        If      *Up1>0 or *Up2>0 or *Up3>0                      Then    'do nothing if an outgoing route is already set
        ElseIf  *Platform=1                                             'if the train is to take option 1
        Then    *Up1=Value      Else    *Up2=Value      Endif   'set route for option 1 = to value. Else set option 2
Else    If      $signal(coordinatesDn) = "RR"                           'if it a down train and the signal is locked
        Then    Value=5         Else    Value=6         Endif   'set value equal to 5. If not locked set value = to 6.
        If      *Dn1>0 or *Dn2>0 or *Dn3>0 or *Dn4>0    Then    'do nothing as a route is already active
        ElseIf  *Platform=3                                             'if the train is to take option 1
        Then    *Dn1=Value      Else  *Dn2=Value  Endif Endif   'set route for option 1. Else set option 2
*Platform=0     ENDSUB                                                  'then reset the platform's control variable to 0
```

will be entered twice, once for option 1 and again for option 2.

The non automatic routes are used for routes into sidings or wrongroad movements that you don't want the PC to use. However we want the PC to check these routes so that the PC does not autoset a route if you have already manually set a route into a siding etc. You may not need to pass as many non automatic routes through to the sub routines so you could reduce or remove these if you don't need them. If you have more than 2 options for trains to take from either the up or down end of a platform you may need to change the sub routine and parameters to suit.

The first line of the sub routine checks if the train is an up train (Platform's control variable will equal 1 or 2). If it is traveling in the up direction then the Up Starter Signal is checked. If the signal is locked then the local variable 'value' is set to 5, otherwise 'value' = 6. Next all the up routes are checked and if any of them – including any non automatic routes such as a route entering the yard– are already set or pending, then nothing happens and the sub routine will end without a route being set. Otherwise if the train is scheduled to take option 1 (Platform=1) then the first of the up routes is set to equal the value of the local variable 'Value'. If the train is set to take option 2 then the second up route is used. The second half of the sub routine is a copy of the first designed to handle the down routes. The sub routine ends by resetting the Platform's control variable to 0 ready for the next train.

Automatic Route Selection for Random Trains

The operator can mark a train as random which will cause the program to 'flip a coin' and route the train according to the result. The example below calls the Sub Routine 'Random' when a train enters the Base Tunnel. The only parameter passed to the sub routine is the index value for the block the train is currently on.

```
When B11 = 12   Do      Random(11)
```

The task of the sub routine is to check if the train is a random train and if it is then it tosses a coin and set's the loco's schedule accordingly. The parameters 'CoinToss', 'Schedule' and Stopping' are local variables used only within this sub.

```
SUB Random (Index, CoinToss, Schedule, Stopping, Index)
Schedule = Schedule[index]                          'copy Train's schedule to local variable
If      Schedule > 19   Then    Return              'if train to be held at this station end sub
Elseif  Schedule > 9    Then    Schedule = 10-,     'if stopping train set 'stopping'=1 and reduce schedule to
        Stopping = 1    Else    Stopping = 0        Endif   'single digit. If express train 'stopping'=0
If      Schedule < 3    Then    Return              Endif   'if train is not a random train end sub
CoinToss = $Random, CoinToss = 10#,                         'randomly pick a number from 0 to 9
If      CoinToss > 6    Then    Schedule = 5                'if toss > 6 set as Midland Random Train
Elseif  CoinToss > 2    Then    Schedule = 4                'if 3, 4, 5 or 6 set as Anticlockwise Random
                        Else    Schedule = 3        Endif   'if 0, 1 or 2 set as Clockwise Random
If      Stopping = 1    Then    Schedule = 10+      Endif   'add 10 for stopping trains
Schedule[index] = Schedule      ENDSUB                      'copy new value back to train's schedule variable
```

The sub begins by copying the value of the train's schedule variable to the local variable 'Schedule'. It then goes on to check if the train is scheduled to be held at the next station (> 19) in which case the sub routine ends without any action. If the schedule variable is greater than 9 (for a stopping train) it is reduced by 10 and the local variable 'Stopping' is set to '1'. Reducing the value of schedule simply makes it easier to manipulate.

If the schedule is less than 3 then the train is not a random train and so the sub ends with no action taken. Otherwise the sub routine randomly chooses a number from 0 to 9, and the local variable 'Schedule' is set according to the result. If 'Stopping' was set to 1 earlier then 10 is now added to the value of 'Schedule'. Finally the new value of 'Schedule' is copied back onto the loco's schedule variable.

Delaying trains in the Base Tunnel

The Base Tunnel is where trains exit the layout heading for destinations beyond this little fictitious world, however in reality a train entering the base tunnel turns on a hidden reverse loop and reenters the layout from the same tunnel. In the interest of variety a random timer can be used to hold up to 2 trains in the tunnel before reentering.

The first ‘When…Do statement’ below allows the operator to turn on or off the feature by left clicking on the screen display. This toggles the variable ‘DelayOn’ between 1 (on) or 0 (off) and also causes the random counter to select a number between 0 and 59. This is then displayed on the screen. When the feature is turned off the delay counter is set to 0 and ‘off’ is displayed.

```
When $Left_Mouse = (24,23,1)     Do     If DelayOn = 0   Then   DelayOn = 1,    Delay = $Random, Delay = 60#,   $draw message (24,23,1) = "@Delay"
                                                         Else   DelayOn = 0    Delay = 0,                      $draw message (24,23,1) = "off"          Endif
When B[11] = 11 or B[12] = 11    Do     If DelayOn = 1   Then   Until Delay = 0  Loop   Wait 1, Delay = 1-     $draw message (24,23,1) = "@Delay"       Endloop
Endif
When B[11] <> 11, DelayOn = 1    Do     Delay = $Random, Delay = 60#,   $draw message (24,23,1) = "@Delay",   $color sprite (24,23,1) = white
When B[12] <> 11, DelayOn = 1    Do     Delay = $Random, Delay = 60#,   $draw message (24,23,1) = "@Delay"    $color sprite (24,23,1) = white
```

When a train has turned on the reverse loop and enters Block 11 or 12 ready to reenter the layout, if the feature is turned on the timer begins to count down each second until ‘Delay’ = 0. The condition ‘Delay = 0’ has been added to the code for setting home signals to ensure the outgoing route is not called until the count down reaches 0. When the feature is turned off ‘Delay’ is set to 0 so the outgoing routes from the tunnel will clear without any delay.

The final lines set a new delay time and display it on the screen as soon as the previous train departs (when Block 11 or 12 no longer equal 11).

Section 5 Route Activation (The AutoSet subroutine)

As we’ve seen when a route has been called manually the route's variable will be set to 8. When it has been called automatically it will equal 6. This section of the code now checks the route's interlocking and activates it by setting the route variable to 1 if there are no conflicting routes set. If called manually the route will be checked once and either set or cancelled. If called automatically the loop statements will cause the PC to keep trying until it can be set. The AutoSet sub routine does this for us.

For routes that are never called automatically such as routes into sidings the calling script follows the format of:

```
When R26=8                                          Do                      AutoSet (&R26, 6, (17,20,1), (16,21,1),(15,22,1), 0, 0, 0)
```

For all other routes they follow the format below. This group of routes covers all the routes in and out of the Western end of The Summit station.

```
When R175>5                                         Do Until R175<6 Loop    AutoSet (&R175, 72, (22,1,1), 0, 0, 0, 0, 0)   Wait 2   Endloop
When R275>5                                         Do Until R275<6 Loop    AutoSet (&R275, 72, (22,1,1), 0, 0, 0 ,0 ,0)   Wait 2   Endloop
When R375>5                                         Do Until R375<6 Loop    AutoSet (&R375, 72, (22,1,1), 0, 0, 0, 0, 0)   Wait 2   Endloop
When R475>5                                         Do Until R475<6 Loop    AutoSet (&R475, 72, (22,1,1), 0, 0, 0, 0, 0)   Wait 2   Endloop
When R751>5 or R752>5 or R753>5 or R754>5            Do Until R751<6, R752<6, R753<6, R754<6
                                                            Loop    AutoSet (&R751, 51, (22,1,1), 0, 0, &R752,&R753,&R754)
                                                                    AutoSet (&R753, 53, (22,1,1), 0, 0, &R752,&R751,&R754)
                                                                    AutoSet (&R752, 52, (22,1,1), 0, 0, &R751,&R753,&R754)
                                                                    AutoSet (&R754, 54, (22,1,1), 0, 0, &R751,&R752,&R753)   Wait 2   Endloop
```

The first line can be read as: When route 175 has been called either manually or automatically (>5) then until the route has been set or cancelled (<6) call the AutoSet sub routine every 2 seconds to check the interlocking and set the route if possible. This repeats for each of the outgoing routes from the platforms to block 75 (R175 to R475), however all the incoming routes (R751 to R754) are handled within a single When Do statement. This is because the PC sets all possible incoming routes to 6 as a train approaches and the autoSet

sub now needs to choose one of these routes and cancel the others. By handling them within a single When Do statement we can tell the PC the priority order we want and also ensure the PC only activates one route.

This section can be read as: When any of the incoming routes have been called (R751 or R752 or R753 or R754) then until all the routes have been set or cancelled (<6) call the AutoSet subroutine to check each route in turn. If no routes were set the first time then try again every 2 seconds. The sub routine ensures that if one of the routes is set the others are immediately reset to 0 to prevent them setting and cause the loop statement to end.

The parameters for the AutoSet sub routine are:

1. the route to be checked
2. the check block – this is the first block along the route to be cleared

3-5 up to 3 track coordinates to check for conflicting train movements.

6-8 up to 3 alternate routes that will need to be reset to 0 if this route is activated. Only relevant for incoming routes.

9-11 Three local variable called Control, Platform and Locked.

The sub command starts by setting Control equal to the control variable for the first block along the desired line (this will tell us if this line is already reserved for a train heading toward us).

Next Block is set equal to the value of this first block so we can check if it is idle.

The first if statement ensures that we still want to set this route. If one of the alternate routes has been set then this sub may still have been called by the loop statement even though we no longer need to set this route, so *Route<6 will cause the sub to end without taking any action if the route has been cancelled.

The 2nd If statement checks the 3 coordinates you've sent through to ensure these blocks are idle. If the track coordinate is not grey (idle) then the variable Locked is set to 1. If less than 3 coordinates are needed for the interlocking at this location a 0 will have been passed through so the clause 'Throat<>0' will allow the sub to skip over this coordinate.

The 3rd if statement checks whether this is an incoming or outgoing route. (None of the outgoing routes will have any alternate routes listed so they will always have a 0 passed through as Alt1.)

```
SUB AutoSet (Route, Block, Direction, Throat1, Throat2, Throat3, Alt1, Alt2, Alt3, Control, Platform, Locked)
{this sub checks the interlocking for all routes and sets them if possible.}
Control = Control[Block]                                        'set Control = to the control variable for the 1st block
Block = B[Block]                                                'set local Block = to this block (11 for up, 12 for down)
If      *Route < 6      Then    Return          Endif           'if this route has not been called return
If      *Throat1<>Grey or Throat2<>0, *Throat2<>Grey            'this and the following line check that all conflicting
        or Throat3<>0, *Throat3<>Grey                           'routes are idle (grey). If any are active then
Then    Locked = 1      Else    Locked = 0      Endif           'Locked is set to 1. If idle Locked is set to 0
If      Direction = 0   Then                                    'if this is an outgoing route
        If  *Route = 6  Then                                    'and it has been called automatically then
                If      Block = 0,                              'if the 1st block is idle, and the line is not cleared in
                        Control = 0,                            'the opposite direction (Control=0) and there are no
                        Locked = 0                              'conflicting routes (Locked=0), then activate the route.
                Then    *Route = 1      Endif   Return          'The sub now ends whether the route has been set or not
        ElseIf *Route = 8 Then                                  'else if this is a manual outgoing route,
                If      Control = 0,                            'if the line is not cleared in the opposite direction and
                        Locked = 0                              'there are no conflicting routes
                Then    *Route = 1                              'then set the route
                Else    *Route = 0, Endif   Endif   Return      'otherwise cancel the request and return
ElseIf *Route = 6 Then                                          'this is an incoming auto route
        If      Block = 0, Locked = 0                           'if the platform is idle and there are no conflicting routes
        Then    *Route=1, *Alt1=0, *Alt2=0, *Alt3=0 Endif       'set this route and cancel any alternate routes
ElseIf Block>0, Block<10 or Locked=1  Then *Route=0             'else this must be a manual incoming route so check the block
Else    *Route = 1,     Endif           ENDSUB                  'is idle and set if possible. Block<10 allows the route to be
                                                                'manually set for shunt trains even if the platform is occupied
```

If this is an outgoing route and it has been called automatically (*Route=6) and if the 1st block along the line is idle (Block=0) and the line is not cleared for a train heading toward us (Control=0) and there are no conflicting train movements (Locked=0) then we can activate the route by setting the route equal to 1. The sub now ends whether the route has been set or not.

However if this is an outgoing route but it has been called manually (*route=8) the same checks are performed but we will allow the route to set even if there is currently a train on the first block along the line. (The signal won't turn green until the block is clear but this allows us to manually set a route in readiness.) Because the route was called manually we want to reset the route to 0 if it has not been set to prevent the loop statement from trying again every 2 seconds. The sub now ends whether the route has been set or not.

Else if this is an incoming route that has been called automatically (*Route=6) and the 1st block along the line is idle and there no conflicting train movements then we can activate this route (*Route=1) and we also want to cancel any alternate routes that are also pending (*Alt1=0 etc.).

Finally if this is an incoming route that has been called manually we want to deny the route if the platform already has a route cleared into it, but this time we do want to allow the route to be set if there is a train currently standing at the platform to allow us to set a route for a shunt loco to enter an occupied platform. So the clause 'Block>0, Block<10' will allow the route to be set if there is a train currently in the platform but deny the route if there is a route already set into the platform from another direction. Of course 'Locked=1' will also prevent the route if there are conflicting train movements, otherwise the route is activated and the sub ends.

Cancelling Pending Routes if the Operator Sends a Train by an Alternate Route

The script we have looked at so far for calling the subroutines leaves one problem; if the operator manually calls an alternate route for a train waiting at a departure signal, the queued route remains active and will activate after the train has passed. The example below ensures that whenever Route 170 is activated that the alternate route (Route 180) is reset to 0. This will also cause any active loop statement from the above code to cease. You only need to enter 1 line for each outgoing route from your platforms. Incoming routes are automatically reset by the AutoSet subroutine.

When R170 = 1 Do R180 = 0

Section 6 Interlocking Controls

Most of the interlocking is done through checking the color of our track blocks but we need to enter a few extra controls for added safety. To us slow human beings the track blocks appear to turn green fairly quickly when a route is set but in the split second it takes for that to happen our PC has probably performed a complete run through of our code and actioned any When Do statements, which means that we cannot rely totally on block coloration for our interlocking. When we were looking at the RouteSet sub routine in Section 5 we saw that the sub checks the Control Variable for the first block along the desired route to ensure the line has not already been cleared in the opposite direction. Well this is where we set the value of those Control Variables. In the example below when any of the routes are activated for a train to head up the Summit line from Upper Junction (R170, R270, or R370) the Control Variable for the last block along that line is set to 1, in this case that is Block72 so we will set Control[72] = 1. Once block 72 returns to idle (either after the train has passed or if the route is cancelled) Control[72] is also returned to 0.

When R170=1 or R270=1 or R370=1 Do Control[72]=1 When B[72]=0 Do Control[72]=0

Section 7 Additional Route Controls for Spiral Routes

Hopefully you can ignore this section for your layout, but on my layout I need to take special care of trains heading down the Spiral to avoid deadlocking two trains on the single track at Midland Junction. If a train has already departed Midland Loop heading toward the Summit a train can still be cleared down the Spiral as long as it is scheduled to take the ShortLine when it reaches Midland Junction. In this case the two trains will safely pass at Midland Junction. However if the train departing The Summit is bound for the Midland Line both trains will come to an impasse at the junction. Therefore we need to check each train's schedule before it can be cleared down the Spiral. First of all the sub routine 'Platform' checks the train's schedule as it arrives at The Summit, and sets the platform variable in the normal way. Once any station stop has been performed the sub routine AutoRoute is called but this time with 2 possible outgoing routes. A quick glance at the track diagram will show that there is only 1 possible route at the Spiral end of the station however to help us monitor the destination of trains on the Spiral, trains that will eventually take the Short Line when they arrive at Midland Junction are routed using route R185 from the Summit, while trains bound for the Midland Line are routed using R186. Both R185 and R186 operate the same way as far as the point and signal settings at the Summit are concerned, however you will see in the Interlocking Controls that R185 sets Control[84] to 2, while R186 sets Control[84] to 3. These values are used in the interlocking for all routes heading toward the Spiral from Midland and Upper Junction. The value of Control[84] is also used to choose the correct route for down trains when they arrive at Block 84.

Control[84] is set automatically for automatic trains when that train arrives at The Summit but for manual trains or trains originating at The Summit I have included the extra code to the right that is triggered when any down train crosses B[85].

This code sets Control[84] equal to 1 for manual trains, and equal to 3 for Short Line trains and equal to 2 for Midland bound trains.

```
When B[85]=12   Do                                          'ensures Control[84] is set correctly
Index = &L[85],         Index = 2724-,     Index = 13/,     'sets Index = to this loco's number
If Mode[index] = 1      Then      Control[84] = 1           'if train is manual set Control[84] = 1
Else                    Index = Schedule[index],           'set Index = to schedule of this loco
If Index > 9            Then      Until      Index < 10
                        Loop  Index = 10-  Endloop Endif   'reduce index to value of 0 to 5
If Index = 2 or Index=5 Then  Control[84]=3
                        Else Control[84]=2 Endif Endif     '3 for Midland Trains, else 2

{The route at Midland Junction is set in response to the value of Control[84]. 1=manual train - no route set. 2=Shortline Train.
3=Midland Train. }
When B[84]=12   Do      If R848=0, R849=0          Then    'when down train on Block84 and no routes have been called
If      Control[84]=2   Then    If $signal {84} (22,4,1) = "RR"  Then R848=5        Else R848=6  Endif
ElseIf  Control[84]=3   Then    If $signal {84} (22,4,1) = "RR"  Then R849=5        Else R849=3  Endif Endif Endif
```

The second section of code to the right is used to set the correct route when the train arrives at the Junction.

Section 8 Automatic Cancelling of Used Routes

After a train has passed or a route has been cancelled the route variable must be reset to 0. You might remember that when we defined our route variables right back at the beginning of the TCL file we inserted a number of 'bookmarks' such as Summit_West and Summit_South etc. and then followed each bookmark with all the routes which cross that particular throat block. So all we need to do to cancel all the routes that cross Summit_West is to tell the sub routine to start at the CTI address of our bookmark 'Summit_West' and check all the routes until it gets to the next bookmark. So the When Do statement reads as: When Block 75 returns to 0 (after a train passes or if the route is cancelled) call the sub routine and start at bookmark 'Summit_West' and stop when you get to the next bookmark 'Summit_South'. The sub routine then checks each route and if the route is set (=1) then it is cancelled (=0).

```
When B[75] = 0  Do      CancelUsedRoute (&Summit_West, &Summit_South)
```

```
SUB CancelUsedRoute (Start, End)
Until Start = End                       Loop
Start = +,          If *Start = 1       Then    *Start = 0  Endif  Endloop   ENDSUB
```

Section 9 Route Controlled Point Setting

After a route has been called and its interlocking checked the route is activated by setting the route variable to the value of 1 (see previous sections on Route Setting and Route Activation). This section deals with setting the points and then setting the RouteBlock variable to actually launch the route and display it on the CTC panel. Technically this script only changes the orientation of the onscreen turnouts however the operation of the physical turnouts is tied directly to these through the section on Point Motor Controls.

In the first example below you will see that both the up and down routes between 2 tracks are handled together as the position of the turnouts is identical with the only difference being the direction of the signaling. So when either route R751 or R175 have been activated the program must set turnout 51 to the normal position. If turnout 31 is currently reversed it is set to normal, as this is the only turnout required by this route all that is required now is for the signal to be cleared and the route displayed on the CTC panel. This is done through the sub routine RouteBlock. The parameters of which are:

1. The RouteBlock involved – this is the first block of the route and is usually the block covering the junction itself.
2. The UpRoute - the route to be cleared if the train is moving in the up direction (from point A to point B)
3. The DnRoute - the route to be cleared if the train is moving in the down direction (from point B to point A)

```
When R751 = 1 or R175 = 1          Do                                  {set switches     51Normal}
      If $switch (23,1,1) = reversed   Then   $switch (23,1,1) = Normal   Endif          'set 51N
      RouteBlock (&B75, &R751, &R175)
```

The first line of the sub routine is essentially just there to prevent the route block being set a second time if the program has been shut down and then restarted for another session. The When Do statement above unfortunately calls the sub routine at start up so this line forces the sub to end if the block has already been set.

```
SUB RouteBlock (RouteBlock, UpRoute, DnRoute)
      If      *RouteBlock > 0      Then    Return
      ElseIf  *UpRoute = 1         Then    Wait 0.1 *RouteBlock = 3
      ElseIf  *DnRoute = 1         Then    Wait 0.1 *RouteBlock = 4
      Endif   ENDSUB
```

The remainder of the sub routine simply checks if it is the up route that has been activated, if it is then it waits 0.1 of a second (to allow the onscreen turnouts to operate to the correct position first) then it sets the route block to the value of 3 (block waiting to clear in the up direction). This will cause the surrounding blocks to activate (see the section on block coloration) and the signal to clear to green as described in the section on Signal Operation. The second line of the code is simply a copy of this for the down route.

Most routes involve more than 1 turnout as shown in the second example for R220 and R202. This includes one of the double slips. The values for the double slips are:

0. Straight through on the horizontal
1. Straight through on the diagonal
2. Turnout from the horizontal to the diagonal
3. Turnout from the diagonal to the horizontal

```
When R220 = 1 or R202 = 1          Do                                  {set switches     61N, 62N, 64N, 65R,}
      If $switch (21,22,1) = Reversed  Then $switch(21,22,1) = Normal      Endif    'set 61N
      If $switch (19,20,1) = Reversed  Then $switch(19,20,1) = Normal      Endif    'set 62N
      If $switch (17,21,1) <> 3        Then $switch(17,21,1) = 3           Endif    'set 64/65 Dbl Slip tk2 to 20
      RouteBlock (&B20, &R202, &R220)
```

Section 10 Block Control

As we have seen earlier the final stage of activating a route is to assign the Route Block for that route to the value of 3 for up trains or 4 for down trains. The remaining track blocks are then set in a cascading form from these blocks as we will see below. However firstly here is a reminder of the permissible values for a track block:

0. Track is idle – track will be shown as grey on the CTC screen.
1. Track cleared for a train traveling in the up direction – track will be shown as green on the CTC screen
2. Track cleared for a train traveling in the down direction – track green.
3. Track waiting to clear for an up train – i.e. route has been set but is waiting for the next track block to become available – track blue.
4. Track waiting to clear for a down train – track blue.
5. Track waiting to cancel, up direction – i.e. train has entered the next block but a time delay is in place to ensure the end of the train is clear of this block – track red.
6. Track waiting to cancel, down direction – track red.
10. Track is occupied by a stationary train – used primarily when assigning a new train or empty stock to a block - track red.
11. Track is occupied by an up train – track red.
12. Track is occupied by a down train – track red.

An extensive series of When Do statements control the value of the block variables in relation to their adjacent blocks. In the first example below Block 70 is the RouteBlock for a number of routes. When Block 70 has been set to 3 by the Route Control part of the script the first line below checks if the next block is idle, if it is the next block is also set to 3. The second line of script then checks if the following block is also idle, if it is and the first 2 blocks are still equal to 3 then all 3 blocks are set to 1. This will set all three track blocks to green on the CTC display and cause the signal to turn green. (Refer to section 12 for an explanation of the Signal Operation)

```
When B[70] = 3, B[71] = 0                    Do      B[71] = 3
When B[70] = 3, B[71] = 3, B[72] = 0         Do      B[70] = 1, B[71] = 1, B[72] = 1
```

When Block 70 has been set to 4 for a down train the state of the points also must be considered. Here if Block 70 = 4 and B23 (Upper Platform 3) is idle with Turnout 27 normal then block 70 and block 23 will both be set to 2. The following 2 lines control routes into platforms 1 and 2.

```
When B[70] = 4, B[23] = 0, {T27}$switch(3,11,1)=normal                    Do      B[70] = 2, B[23] = 2
When B[70] = 4, B[22] = 0, {T27}$switch(3,11,1)=reversed, {T25N}$switch(6,11,1)=normal, B[25] < 5
                        Do      B[70] = 2, B[25]=2, B[22] = 2
When B[70] = 4, B[21] = 0, {T27}$switch(3,11,1)=reversed, {T25R}$switch(6,11,1)=reversed, B[25] < 5
                        Do      B[70] = 2, B[25]=2, B[21] = 2
```

The most complex examples cover the eastern end of Lower Junction with its double slips and multiple route options. In the first example below the 2 halves of the script are effectively the same however there are 2 different states possible for the double slip when turnout 63 is reversed hence the double entry.

```
When B[90] = 4, B[6] = 0, {T90R}$switch(17,20,1)=1      or      B[90] = 4, B[6] = 0, {T90R}$switch(18,20,1)=2      Do      B[90] = 2, B[6] = 2
```

A down train traveling through Lower Junction will become an up train when crossing the boundary between block 90 (covering the crossover at the eastern end of platform 4) and block 93 (the first block at the bottom of the Midland Line). Therefore the next example shows even numbers for block 90 and odd numbers for blocks 93 and 94 even though the

train is continuing in the same direction. A similar arrangement exists at the Summit where up trains approaching from the Summit Line become down trains as they enter the platforms.

```
When B[90] = 4, B[93] = 3, B[94] = 0                    Do    B[90] = 2, B[93] = 1, B[94] = 1
```

The other function of the Block Control section is to reduce block variables to 0 when required. In most cases the blocks will be returned to idle after a train passes but if a route is manually cancelled or a train is reversed the adjacent blocks may need to be cancelled using this section of the script. In the first example below Block 51 (Summit Platform 1) has been cleared for a down train however the previous block (Block75) has been cancelled either by the route being cancelled or a train reversing on Block 75 and heading away from the Summit. Therefore this script also cancels Block 51 by reducing it to 0.

```
When B[75] = 0, B[51] = 2        Do    B[51] = 0
```

The following line is effectively the same but is used when the block to be cancelled could have the value of either 2 or 4

```
When B[75] = 0                   Do    If B[72] = 2 or B[72] = 4        Then    B[72] = 0        Endif
```

The following example takes into account the state of the points between the 2 blocks.

```
When B[80] = 0, B[21] = 2, {T25N}$switch(6,11,1)=normal        Do    B[21] = 0
```

Again we need a unique piece of code for Block 25 protecting the crossover at the western end of Platform 2 at Upper Junction. The following line will cancel block Block25 whenever Block 70 is idle and turnout 26 is in the reverse position, however it will not cancel the block if there is a train standing on the crossover at the western end of Upper platform 2 (B[25]=10 or more).

```
When B[25] < 10 and B[70] = 0 and {T26N} $switch (5,11,1) = normal        Do    B[25] = 0
```

The code in the final example is used where multiple platforms connect directly to the same block and can therefore be handled together. The state of the turnouts in the junction is irrelevant as none of the platforms can legitimately be cleared for a down train if the preceding block (Block40) is idle.

```
When B[40] = 0,    Do    If      B[3] = 2    Then    B[3] = 0
                         ElseIf  B[4] = 2    Then    B[4] = 0
                         ElseIf  B[5] = 2    Then    B[5] = 0    Endif
```

Because there is no route block associated with the routes in and out of the yard at Upper Junction the route variables themselves are used to pull down the neighbouring blocks. When the route from the yard to platform 1 (R41) is reset, if the platform is still set for a down train, the platform block (B[21]) is also cancelled.

```
When R41 = 0       Do    If      B[21] = 2    Then    B[21] = 0    Endif
```

Section 11 CTC Display Block Colouration

The CTC display uses different colours to display the state of each track block or block joiner dependant on the value of the block variable. The colours I use are:

1. Grey ($RGB_8F8F8F) – this is the default state and indicates the block is idle, i.e. there is no rolling stock on this block and there are no routes cleared over this block. (value = 0)
2. Blue ($RGB_FFFF00) – a route has been set across this block but is waiting for the next block to clear before the signal turns green (value 3 or 4).
3. Green ($RGB_FF00) – this block is cleared for a train to enter it (value 1 or 2).
4. Red – the block is currently occupied by a train or the train has passed but a time delay is in place to ensure the train has cleared the block before it is released (value > 4).
5. Yellow ($RGB_FFFF) – only used on platform blocks to show that the train is performing a scheduled station stop (Control Variable>9). It turns red once the stop timer expires.

These colors are defined in the Constants section at the very beginning of the code.

Most blocks follow the straight forward syntax below. When the block's variable = 1 or 2 colour the block green. When the block = 3 or 4 colour the block blue. When the block is greater than 4 colour the block red. When the block = 0 colour the block grey.

```
When B[33] = 1 or B[33] = 2          Do    $color block (17,14,1) = Green,
When B[33] = 3 or B[33] = 4          Do    $color block (17,14,1) = Blue,
When B[33] > 4                       Do    $color block (17,14,1) = Red,
When B[33] = 0                       Do    $color block (17,14,1) = Grey,
```

Platform blocks include an additional line to mark the block yellow if the train is due to stop at the station. The third line will turn the track red once the station stop is complete and the train is ready to depart (when the control variable drops to less than 10).

```
When B[21] = 1 or B[21] = 2          Do    $color block (11,10,1) = Green,
When Control[21] > 9                 Do    $color block (11,10,1) = Yellow,
When B[21] > 4, Control[21] < 10     Do    $color block (11,10,1) = Red,
When B[21] = 0                       Do    $color block (11,10,1) = Grey,
```

The station approaches and junctions require additional script to colour each of the short sections of track in between the turnouts and the actual track blocks. These sections of track do not have their own block variable and therefore are coloured in response to the state of the adjoining tracks and points. The "When … Do" statements begin the same way as before by colouring the actual track block itself according to the value of the block variable, but the additional "If…Then" statements check which way the adjoining turnouts are set and colours the connected tracks accordingly.

The example below can be interpreted as: When the value of Block40 changes to 1 or 2 colour Block 40 green. If Turnout 30 is in the normal position then also colour the section of track between block 40 and block 41 green. Otherwise (if Turnout 30 is reversed) then colour the section of track between Block 40 and Block 31 green. This example then continues with a second If statement to check the position of Turnout 2 at the other end of Block 40. If it is in the reversed position then the track between Block 40 and Platform 2 is coloured green. Otherwise (if Turnout 2 is normal) then the section between Block 40 and Turnout 3 will be coloured green. If this is the case then it goes on to check Turnouts 3 and 4 and set the connecting tracks accordingly. This example would be repeated for each of the possible colours; i.e. red, green, blue and grey.

```
When B[40] = 1 or B[40] = 2          Do                          $color block (3,21,1)=Green
        If $switch{T30}(1,19,1)=normal          Then             $color block(2,19,1)= Green
                                                Else             $color block(1,19,1)= Green          Endif
```

```
If $switch{T2}(4,22,1)=reversed          Then    $color block(4,22,1)= Green
                                         Else    $color block(4,21,1)= Green
        If $switch{T3}(5,21,1)=normal    Then    $color block(6,21,1)= Green
                                         Else    $color block(6,20,1)= Green
        If $switch{T4}(6,20,1)=normal    Then    $color block(7,19,1)= Green
                                         Else    $color block(7,20,1)= Green  Endif Endif Endif
```

There are a couple of locations where special clauses are added such as the example below which colours the track and turnouts at the western end of the Summit platforms. Special consideration has to be made here of the turnout to the goods yard which could be fouled by a train standing on platform 3. The additional clause "Elseif B55 < 5" has been added to ensure that if there is a train currently sitting on the turnout the colour of the corresponding section of track remains red. Only if block B55 is less than 5 will the colour change.

```
When B[75] = 1 or B[75] = 2      Do                                  $color block (22,1,1)= Green
        If $switch{T31}(22,1,1)=normal           Then    $color block(23,1,1)= Green
                                                 Else    $color block(23,2,1)= Green
                If $switch{T32}(23,2,1)=reversed Then    $color block(24,2,1)= Green
                        Elseif  B[55] < 5        Then    $color block(24,3,1)= Green
                If $switch{T33}(24,3,1)=normal   Then    $color block(25,4,1)= Green      Endif Endif Endif
When B[55] > 4                                   Do      $color track (24,3,1) = Red
When B[55] = 0                                   Do      $color track (24,3,1) = Grey
```

In the example below when Block 83 (at Midland Junction) is occupied, Block 83 is coloured red as well as Turnout 8. The adjoining piece of track is also coloured red depending on the position of Turnout 8. That's all fairly standard, however if the train is traveling in the up direction then the additional lines of script will reset Turnout 8 and the adjacent tracks to idle after 8 seconds. This is to allow the points to be released and the train to shunt back from Block 83 onto the other line to allow another train to pass on the single track section.

```
When B[83] > 4          Do                                  $color block (18,3,1) = Red
                                                            $color block (15,3,1) = Red
        If $switch{T8}(15,3,1)=normal           Then        $color track (14,3,1) = Red
                                                Else        $color track (14,4,1) = Red     Endif
    If B[83] = 11 {Up train only}       Then    wait 8      $color block (15,3,1) = Grey    'turn T8 grey
                                                            $color track (14,4,1) = Grey    'and joiners
                                                            $color track (14,3,1) = Grey    Endif
```

It is important that Block 20 (the centre track of the 3 track throat at the Eastern end of Lower Junction) is only coloured if the train is moving along block 20 between the 2 double slips at the end of platform 3. Therefore the code checks to see if either Turnout 20 or Turnout 13 is reversed. If this is the case the train is cutting across rather than traveling along block 20 so the code will not make any changes (the track will remain grey). Otherwise if the train is traveling along Block 20 the final 3 lines colour Block 20 and both double slips accordingly.

```
When B[20] = 1 or B[20] = 2      Do      If {20R}$switch(18,20,1)=reversed or {13R}$switch(16,21,1)=1 or $switch(16,21,1)=2      Then    'do nothing
                                         Else    $color block (18,21,1)= Green
                                                 $color track (16,21,1)= Green
                                                 $color track (19,21,1)= Green      Endif
```

The colour of the double slip 10/20 is then extended onto the adjacent track joiners through the code below. There is a similar code for double slip 12/13.

```
When $color(19,21,1) = $RGB_FF00          Do                {extend colour from dbl slip 10/20 toward Block 12 or Turnout 11}
        If $switch(19,21,1)=0 or $switch(19,21,1)=3          Then      $color track (20,21,1) = Green
                                                             Else      $color track (20,22,1) = Green      Endif
```

Section 12 Signal Operation

I have not installed physical signals on my layout yet so this chapter talks about setting the state of the signals on the CTC screen. However it should not be hard to drive your physical signals in response to the state of the onscreen signals.

The aspect of each signal on the CTC panel is tied to the colour of the track immediately beyond the signal. In essence when the track is green the signal will be green, if the track is any other colour the signal will be red. All signals on my layout are 2 aspect signals so each signal can display the following aspects:

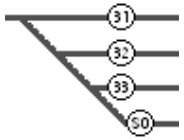
East or South facing signal	West or North facing signal	
"G-"	"-G"	Green – train will proceed on the set route. The dash indicates that the second lamp is unlit.
"-R"	"R-"	Red – brake will automatically apply and the train will pull up gently in front of the signal
"RR"	"RR"	Double red signifies the signal is locked at red to prevent the PC from clearing this signal.
"YR"	"RY"	The signal has been selected as a starter signal and is waiting for a valid destination signal to be selected.

Most signals follow a straight forward code such as the example for Signal 70 below. When the colour of the track immediately beyond signal 70 (CTC coordinates 2,9,1) is green ($RGB_FF00) colour the signal green. And when the colour of the track immediately beyond signal 70 is not green colour the signal red as long as it is not locked ("RR"). You may be wondering why the script needs to check if the signal is locked at "RR" as the adjacent track could never have been green with the signal locked, however the program checks these When Do statements at start up and was therefore unlocking signals and returning them to "R-" at startup without this extra clause added.

```
When $color (2,9,1) =     $RGB_FF00     Do {Sig70}$signal (2,8,1) = "-G"
When $color (2,9,1) <>    $RGB_FF00     Do If    $signal (2,8,1) <> "RR"    Then    $signal (2,8,1) = "R-"      Endif
```

Some signals such as Signal 12 in this example need to take into account the position of the adjacent points before setting the signal. When the turnout is in the normal position and the colour of the track beyond it is green, colour Signal 12 green. But when the colour of the track is not green or the points are reversed colour the signal red.

```
When $color (5,11,1) =    $RGB_FF00,     {T25N} $switch (6,11,1) = Normal      Do {Sig12}$signal (8,11,1) = "G-"
When $color (5,11,1) <>   $RGB_FF00  or  {T25N} $switch (6,11,1) = Reversed    Do {Sig12}$signal (8,11,1) = "-R"
```



Some signals are dealt with together such as this example of Summit West signals as they are tied to the same piece of track. When the track is green and the points are set the right way set the appropriate signal green. If the track is red then colour both signals red. The 1st line of code below can be read as: When the turnout in front of signal 32 is green and the points are set for the 3rd track, then set signal 33 green. The next line reads: if the track is green but this time the points are set for the 4th track, set signal S0 (the yard signal) to green. The final line sets both signals red if the track is not green.

```
When $color (24,2,1) =     $RGB_FF00,     {T32R} $switch (24,2,1) = Normal, {T33N} $switch (25,3,1) = Reversed
                                                Do        {Sig33}$signal (26,3,1) = "G-"
When $color (24,2,1) =     $RGB_FF00,     {T32R} $switch (24,2,1) = Normal, {T33R} $switch (25,3,1) = Normal
                                                Do        {SigS0}$signal (27,4,1) = "G-"
When $color (24,2,1) <>    $RGB_FF00      Do        $signal {Sig33} (26,3,1) = "-R",     $signal {S0} (27,4,1) = "-R"
```

Signal 24 at Upper Junction is a very simple one as it only controls the route in or out of the yard, so this signal simply turns green whenever either route is active and set to red when both are inactive.

```
When R41 = 1 or R14 = 1                 Do {Sig24} $signal (7,9,1) = "G-"
When R41 = 0, R14 = 0                   Do {Sig24} $signal (7,9,1) = "-R"
```

Section 13 Signal Cancel or Lock Commands (The RightClickSignal subroutine)

This section of the code performs 3 different functions depending on the current state of the signal.

1. if the signal is currently green the code cancels any active routes from this signal and returns the signal to red
2. if the signal is currently red the code locks the signal in the double red aspect ("RR") to prevent the PC setting a route from it
3. if the signal is already locked the code unlocks the signal and returns it to red

All of this is handled by the RightClickSignal subroutine. You can use whatever means you like to call the sub routine, I have set up my code so that I can cancel signals from the keyboard or my handhelds or mouse clicks. So my code to call the sub routine looks like:

```
When $command = C30 or SignalEntry = 30 or $right_mouse = {Sig30} (1,18,1)          Do        RightClickSignal ( … )
```

In other words this code allows me to cancel signals by:

- typing C (for cancel) at the keyboard followed by the signal number and the enter key, or
- pressing button 4 (the signal cancel button) on one of the handsets followed by the signal number and the enter button (this sets the variable Signalentry = to the signal number) or
- right clicking on a signal with the mouse

The parameters for the 'RightClickSignal' sub are:

1. the CTC coordinates of the signal
2. the normal aspect of this signal when idle – either "-R" or "R-" depending on its direction/location

3-5 the 1st RouteBlock followed by up to 2 routes that cross that block – see explanation below

6-8 a 2nd RouteBlock and 2 more routes if required

7-9 a 3rd RouteBlock and final route if required

Most signals involve only 1 RouteBlock, this is generally the block immediately beyond the signal and includes the turnouts that form the station throat. The RouteBlock acts as the control block to launch or cancel the route (i.e. it is this block that is set to 3 or 4 when the route is set and the following blocks are set in a cascading method from there.) The

example for Signal 51 below is such an example. Here the coordinates of the signal and the orientation of the signal are passed to the sub routine, followed by the only RouteBlock valid for this signal (Block 75) and the only possible route (route175). B0 is substituted for the remaining unused parameters.

```
When $command = C51 or $right_mouse = {Sig51} (25,1,1)
                Do      RightClickSignal ((25,1,1), "-R", &B[75], &R175, &B0, &B0, &B0, &B0, &B0, &B0)
```

Other signals - such as Signal 30 in the second example – require 2 or 3 RouteBlocks to be passed to the subroutine. In this example Block30 is the RouteBlock for Routes R301 and R302, while Block40 is the RouteBlock for the remaining routes R303, R304 and R305. (Trains using these latter routes use the crossover and travel across block 40 rather than block 30.) Note also that the 5th route in this example (R305) uses the same RouteBlock as the previous routes and so Block 40 is entered for both the 2nd and 3rd RouteBlocks.

```
When $command = C30 or $right_mouse = {Sig30}   (2,18,1),
                Do      RightClickSignal ((2,18,1), "R-", &B30, &R301, &R302, &B40, &R303, &R304, &B40, &R305)
```

The sub routine can be read as: When the signal has been right clicked if any of the routes originating at that signal are active, check that there is no train on the block directly beyond the signal and if it is free set the routeblock equal to 0. If no routes are currently set the sub routine checks if the signal has been chosen as a starter signal but no valid destination signal has yet been selected (either "YR" or "RY"). If it has then this feature is cancelled by returning the signal to its normal idle state. If the signal has been locked in the red aspect ("RR") it is unlocked by returning the signal to its normal idle state and the sub then checks if any of the possible routes are equal to 5 (route chosen but waiting for the signal to be unlocked before activating). If any are equal to 5 they are given the value 3 and the Route Activation section of the script will then check the interlocking and activate the route if available.

```
SUB RightClickSignal (coordinates, Lamp, RouteBlock1, Route1, Route2, RouteBlock2, Route3, Route4, RouteBlock3, Route5)
If      *Route1=1 or *Route2=1  Then     If *RouteBlock1<5                       'if route 1 or 2 is set and there is no train on the throat
Then    *RouteBlock1=0, *Route1 = 0, *Route2 = 0        Endif                   'reset both routes and the throat block to 0.
ElseIf  *Route3=1 or *Route4=1  Then     If *RouteBlock2<5                       'if route 3 or 4 set and there is no train on the throat
Then    *RouteBlock2=0, *Route3 = 0, *Route4 = 0        Endif                   'reset both routes and the throat block to 0.
ElseIf  *Route5=1               Then     If *RouteBlock3<5                      'if route 5 is set and there is no train on the throat
Then    *RouteBlock3=0, *Route5 = 0                     Endif                   'reset both routes and the throat block to 0.
Elseif  $Signal(coordinates)="YR" or $Signal(coordinates)="RY"                  'if the operator was setting a route
Then    $Signal (coordinates)=Lamp                                              'then reset signal to it's normal idle state
Elseif  $Signal(coordinates)="RR"  Then  $Signal (coordinates)=Lamp             'if signal is locked (RR) reset it to idle
        If      *Route1=5       Then    *Route1=6       Endif                   'and unlock any pending route by setting it
        If      *Route2=5       Then    *Route2=6       Endif                   'from 5 to 6.
        If      *Route3=5       Then    *Route3=6       Endif
        If      *Route4=5       Then    *Route4=6       Endif
        If      *Route5=5       Then    *Route5=6       Endif
Elseif  $Signal (coordinates) = "R-" or $Signal (coordinates) = "-R"            'if the signal is currently idle
Then    $Signal (coordinates)="RR"                                              'then lock it (RR)
        If      *Route1=6       Then    *Route1=5       Endif                   'and lock any pending routes by changing
        If      *Route2=6       Then    *Route2=5       Endif                   'them from 6 to 5.
        If      *Route3=6       Then    *Route3=5       Endif
        If      *Route4=6       Then    *Route4=5       Endif
        If      *Route5=6       Then    *Route5=5       Endif   Endif   ENDSUB
```

If the signal is currently red then the sub routine locks the signal at "RR". This displays a double red aspect and prevents the PC from auto-setting any routes from this signal until the signal is unlocked. If there is already a route pending the route is set to 5.

The signal controlling the routes in and out of the yard at Upper Junction (Sig24) is much simpler and does not need the sub routine. When it is cancelled it immediately cancels the 2 possible routes R41 and R14.

```
When $command = C24 or $right_mouse = {Sig24}   (7,9,1)          Do      R41 = 0, R14 = 0
```

One drawback with the CTI system is that the communication between the DCC controller and the PC is only one way. I.e. any changes made by the PC are reflected by the DCC controller however changes made by the DCC controller are not transmitted back up the line to the PC. So if a train is stopped manually using the DCC controller the PC will still show the loco traveling at its previous speed. The train will remain stopped for a period of time however if the brake has not been applied at the PC, or the PC speed reduced to zero the PC will restart the train after an undefined period.

Despite this drawback the only chance of derailment or collision is when a train enters a platform with a green departure signal and is manually stopped in the station using the DCC controller, but then the signal is cancelled and a conflicting route set. As a result the script below acts as a safety feature to ensure that the brake is automatically applied to any train standing at a platform if the departure signal is cancelled. It also serves to apply the emergency brake (zero momentum) if the signal is dropped in front of the train.

```
When {Sig27}$signal (7,12,1) <> "-G", {Sig23}$signal (13,12,1) <> "G-"
                                                    Do *L[23].momentum = 0, *L[23].brake = on
```

The code for applying the brake to a train stopped at Midland Loop differs slightly due to the use of infrared detectors rather than the current detectors used elsewhere. Because the train is moved directly from the approach track to the mainline or loop without first being moved onto a throat block, the beacon will already appear on the Midland track when Block 95 or Block 96 turns red and therefore turns the adjacent signal red. With the standard script above this automatically applies the brake with zero momentum and overrides the momentum settings in the "Train Braking" section of the script. Therefore the script below is used to ensure that only the signal in the direction of travel is checked.

```
When {Sig93}$signal (16,7,1) <> "-G"      Do If B[91] = 11  Then     *L[91].momentum = 0, *L[91].brake = on   Endif
When {Sig91}$signal (21,7,1) <> "G-"      Do If B[91] = 12  Then     *L[91].momentum = 0, *L[91].brake = on   Endif
```

Section 14 Train Braking and Acceleration Controls

You could replace this section of code with much simpler brake control to simply apply the brake whenever a train arrives at a red signal such as:

```
When B[1] = 11 (when an up train arrives on block 1)     Do     If     $Signal(x,y,z) <> Green          (if the signal is not green)
                                                                Then   *L[1].brake = on Endif           (apply the brake to the train on Block 1)
```

And then take the brake off again with code such as:

```
When $signal(x,y,z) = Green (when the signal turns green)     Do     *L[1].brake = off          (remove the brake on any train standing on block 1)
```

However I wanted to include a few extra features.

- If a train is scheduled to stop at a station its brake will be applied even if the departure signal is green, so we need to check the train's schedule
- The train's speed is also taken into consideration to ensure all trains stop accurately at the signal regardless of their speed
- If a train has been stopped automatically the PC will remove the brake again once the signal turns green, however if the driver manually applies the brake the train will not restart until the driver removes the brake, even if the signal turns green.
- If the driver is stopping the train manually the PC leaves control up to the driver but acts as a fail safe system to monitor the braking curve and apply the brake if the train is not slowing sufficiently to stop at the signal.

Calculating When to Stop - Inserting Wait Times

In order to do this we apply the brake through a set of brake ramps and a sub routine called AutoBrake. Every time a train approaches a signal one of the brake ramps is called to check the train's speed and calculate the exact point where the brake needs to be applied to stop this train accurately at the signal. Once that point is reached the AutoBrake sub then checks if the signal is red or if the train is scheduled to stop at the station and applies the brake with the correct amount of momentum if the train needs to stop.

The code used when a train is approaching a signal looks like the example below: When an up train arrives on block 99 call the sub routine called Ramp and pass through the index number for this block. You'll note that some lines include an initial wait time before the brake ramp is called depending on the distance between the sensor and the signal. This initial wait time should equal the length of time it takes to reach the point where trains traveling at full speed will need to brake to stop at this signal.

```
When B[99] = 11,        Do      Wait 1   Ramp (99)
```

You'll also note that some blocks use the condition 'When B[4] > 10' to call the brake ramp. This is for blocks where trains traveling in both the up and down direction are approaching a signal and may need to stop. If the initial wait time is similar for both scenarios then both can be handled together.

The Ramp sub simply looks for a ramp that is not being used (if Ramp1 = 0 then that ramp is available). It then activates that ramp by setting its variable equal to the index number of the block.

```
SUB Ramp (Block)
If      Ramp1 = 0       Then    Ramp1 = Block
ElseIf  Ramp2 = 0       Then    Ramp2 = Block
ElseIf  Ramp3 = 0       Then    Ramp3 = Block
ElseIf  Ramp4 = 0       Then    Ramp4 = Block
Else    $status = "You need another Brake Ramp!"  Endif    ENDSUB
```

The number of brake ramps you need will depend on the number of trains you have on the move at any one time and the likelihood they will all be approaching a signal at the same time. At the moment I only have 5 trains but find that the 3rd ramp is needed regularly, but the 4th one is hardly ever used. You'll see to the right that I've added an extra line at the end of the Ramp sub that will display a warning message on the status bar if you do not have enough Brake Ramps.

The ramps operate by inserting additional wait times when the train is traveling at reduced speed. As you'll see to the right if the train is traveling at full speed we jump straight to the final If statement without adding any additional wait times. But if the train is traveling at less than 14 the first If statement causes us to wait 0.8 of a second and then check again. At the end of that wait time if the train is now traveling at a speed of 13 we jump straight to the end without any further wait times, otherwise a second wait time of 0.8 of a second is inserted. This process continues through the rest of the wait times until the final If statement.

```
When Ramp1 > 0   Do    If *L[ramp1].speed < 14   Then    Wait 0.8   Endif
                       If *L[ramp1].speed < 13   Then    Wait 0.8   Endif
                       If *L[ramp1].speed < 12   Then    Wait 0.8   Endif
                       If *L[ramp1].speed < 11   Then    Wait 1.2   Endif
                       If *L[ramp1].speed < 10   Then    Wait 1.2   Endif
                       If *L[ramp1].speed > 8    Then    Brake = ramp1
                       Endif     Ramp1 = 0       'set brake = ramp1
```

This last If statement checks that the train is traveling at at least half speed (>8) and if it is then the variable Brake is set equal to the value stored on Ramp1, which is the index number of the block the train is on. If the train is traveling at 8 or less I assume the train is being driven manually and so leave it up to the driver to stop in time, but you could continue to insert more wait times for all speeds if you wish. I use 14 speed steps on my layout but if you are using 128 speed steps you could replace 14 with 120, 13 with 112, 12 with 104, 11 with 96 etc. to create the same result.

The very last line resets the Ramp variable to 0 which makes it available for another train.

Have I lost you totally? Well let's recap where we have got to so far by seeing how it works in practice. Let's say one of our trains is traveling in the up direction at a speed of 12 and has just entered Block 72. And in 2 seconds time it will be at the point where trains traveling at full speed will need to brake to stop at this signal, so the code for this block will be:

```
When B[72] = 11        Do      Wait 2   Ramp (72)        'When an up train enters block 72 wait 2 seconds, then call an available brake ramp.
```

So the first available ramp will now check the speed of our train and because it is traveling at less than 14 it waits 0.8 of a second, then checks again. This time because our train is traveling at less than 13 we insert another wait time of 0.8 of a second. However the remaining If statements will not insert wait times because our train is traveling at 12, so we skip down to the line that says: "If *L[Ramp1].speed > 8 Then Brake = ramp1"

Because our train is cruising along at a speed of 12 the index number of the block is copied from Ramp1 to the variable Brake. This now triggers the second half of the process to actually apply the brake to the train as below.

But before we go on, take a look at how the brake ramp also acts as a supervisor if the train is being stopping manually? Even if the driver is slowing the train manually the brake ramp will still be called and will keep an eye on the speed at each step. So if the train is slowing sufficiently the PC will not interfere, however if the train is not slowing fast enough the PC can apply the brake at any stage to ensure it stops safely at the signal.

Applying the Brake – The AutoBrake sub routine

Once any wait times have been observed the AutoBrake sub routine checks the signal and the train's schedule, and applies the brake if needed. The last thing our Brake Ramp did for us was to set the variable 'Brake' equal to the index number for the block the train is on. This Brake variable is then used to call the AutoBrake sub through code such as the line below.

```
When Brake = 51        Do Autobrake (51, (25,1,1), (30,1,1), 65, 80, 1)        'When a train has reached its braking point call the Autobrake sub
```

The same line of code is used when either an up or down train is on the block so we need to pass through to it the coordinates for both the up and down signals. However for blocks such as block 84 below where trains never stop at the up end of the block and there is no signal at that end, you can simply pass a 0 through as the up coordinates.

```
When Brake = 84        Do Autobrake (84, 0, (22,4,1), 45, 65, 0)
```

The parameters for the AutoBrake sub are:

1. the index number of the block the train is on
2. the coordinates of the signal facing a train traveling in the up direction
3. the coordinates of the signal facing a train traveling in the down direction
4. the momentum to use for fast trains (speed > 11)
5. the momentum to use for slow trains (speed < 12)
6. a parameter called Station which equals 1 if the train is at a platform, otherwise 0

7-11 local variables called; schedule, loco, block, green and speed

I have found that using different momentum for fast and slow trains gives a more realistic result. Finding the perfect momentum settings for each location will be a bit of trial and error until you get it just right. You will find this much easier if you can speed match your locos as closely as you can using their decoder CVs. Once you have done that experiment with a single train running at its top speed until you get the initial wait time and momentum settings right for each location so the train pulls up right where you want it to. Now drop its speed and try it again. If you need to you can change the wait times within your brake ramps but remember that will change the timing for all locations on your layout.

```
SUB AutoBrake (index, signal, signalDn, fast, slow, station, schedule, loco, block, green, speed)
Loco = L[index], Loco = 2724-,     Loco = 13/,                                  'sets Loco = to loco #.  0,1,2,3,4,5, etc.
Schedule = Schedule[Loco],         Green = 0                                    'copy Train's schedule to local variable
block = B[index],                                                               'copy value of the block to local variable
speed = speed[loco]                                                             'copy train's speed to local variable
If block = 12      Then    signal = SignalDn          Endif                     'use down signal if train traveling in down direction
If $Signal (Signal) ="-G" or $Signal (Signal) ="G-"   Then  Green = 1  Endif
If Station = 0     Then    If Green=1        Then     Return   Endif            'if not at station and signal is green end the sub
ElseIf Schedule < 10,      Green=1           Then     Return   Endif            'return if express train at station and signal green.
Index = L[index], Index = 3+,                                                   'set index = to the momentum address for this loco
If      speed < 12         Then    *Index = slow                                'if speed is less than 12 apply 'slow' momentum
                           Else    *index = fast,     Endif                     'otherwise apply 'fast' momentum
Index = 1-,        *Index = on                        ENDSUB                    'set index = to brake for this loco and turn on.
```

The AutoBrake sub starts by checking which loco is on the block and sets the local variable Loco equal to the number for this loco (0, 1, 2 or 3 etc.). It then copies this train's schedule and speed onto the local variables and sets Block equal to the value of the block the train is on, this will be either 11 (for an up train) or 12 (down train). If it is a down train we then copy the coordinates for the down signal over the top of the coordinates for the up signal.

The sub now checks the signal and if it is green then it sets 'Green' equal to 1. If the train is not at a station (Station=0) then if the signal is green we don't need to apply the brake so the sub ends. Otherwise the sub goes on to check the schedule of the train and if it is not due to stop at the station (schedule<10) and the signal is green then again the sub ends without applying the brake.

Otherwise we set the variable 'index' equal to the CTI address for this loco and then add 3 to it so that 'index' now points to the momentum setting for this loco. We now copy our chosen momentum value across to the train's momentum address. If the train is traveling at less than 12 we use the momentum for slow trains, if it is traveling 12 or higher we use the fast momentum. Again if you are using 27, 28 or 128 speed steps simply change that figure from 12 to whatever you require for your layout.

The final line now subtracts 1 from the address stored on the variable index so that index is now pointing to the brake setting for this loco, and the brake is turned on.

Train Acceleration – The BrakeControl Sub Routine

We have seen above how the brake is applied automatically when a train approaches a red signal. In most instances we want the brake to be removed again and the train to accelerate automatically once the signal turns green. However there are a few instances where I don't want the train to auto start so I use a sub routine called BrakeControl to perform a few additional checks before removing the brake. On my layout the sub also displays the state of the brake on the CTC display.

The current state of the brake for each train is shown on the CTC display by means of a 2 aspect signal beside each train in the Train Properties area. If the brake is off then both lamps of the signal are extinguished. If the brake has been applied automatically by the PC then the lower lamp turns red. If the brake has been applied manually by the operator then both lamps will be red. This allows us to ensure that if the train has been stopped deliberately by the driver the train will not autostart even if the signal turns green – therefore leaving total control up to the driver.

Tip: If you want the brake to be removed as soon as the signal turns green you don't need the BrakeControl sub. Simply use code such as: When B[72]=11 and &signal(22,1,1)= "G-" Do L[72].brake=off

For trains waiting at most signals the following code is used:

```
When B[72] = 11 and $signal (22,1,1) = "G-"          Do    BrakeControl (&L[72], 60, 0)    'when there is an up train on block 11 and the signal turns green do …
```

For trains departing platforms the platform's control variable must be less than 10 to indicate the train has completed any scheduled station stop before the brake is released, hence the code:

```
When B[21] = 11 and $signal (7,10,1) = "G-", Control[21]<10 Do    BrakeControl (&L[21], 60, 0)    'when an up train is ready to depart block 21 and signal turns green …
```

The parameters that need to be passed to the sub routine are:

1. the loco stopped at the signal
2. the start up momentum to be used
3. the device trying to remove the brake (1 = Handheld controller trying to release the brake, 2 = Handheld controller applying the brake, 0 for all others)
4. a local variable called 'index' used to point to the correct entity within CTI
5. another local variable used to calculate and store the CTI address of the brake display for this train. (Appendix A contains a table of CTI addresses for the grid coordinates in the train properties area)

The sub routine starts in the usual way by setting the local variable 'Index' equal to the beacon value and then reducing it down to the single digit loco number. This is then copied to the local variable 'coordinates' which is then set equal to the CTI address of the grid coordinates for the brake display for this loco. On my layout the status of the brake for loco 1 is displayed on a signal at coordinates (34, 13,1). This grid square has the CTI address of 8221 and the address for each subsequent loco increases by 50. (The CTC screens are 50 lines wide so increasing the CTI address by a factor of 50 will give you the square directly beneath).

The next line sets the momentum for this loco by first setting Index equal to the CTI address for this loco and then increasing it by 3, causing it to point to the loco's momentum address. The momentum value past to the sub routine is then copied to the memory location being pointed to by index.

```
SUB BrakeControl (beacon, momentum, device, index, coordinates)
Index = *beacon, Index = 2724-,     Index = 13/,                          'sets Index = to loco #.  0,1,2,3,4,5, etc.
Coordinates=Index, Coordinates=50*, Coordinates=8171+                     'value=8221, 8271, 8321 etc. (brake display coordinates)
Index = *beacon,     Index = 3+,    *Index = momentum,                    'point index to address of the momentum for this loco
If       Device = 0       Then     $Signal (coordinates) = "x-",         'if brake released by PC then turn off bottom light only
ElseIf   Device = 1       Then     $Signal (coordinates) = "--",         'if brake released by HH then turn off both lights
Else     $Signal (coordinates) = "RR",                                    'else if brake applied by HH turn both lights on
         Index = 1-,      *Index = on       Endif                        'and apply the brake for this loco
If       $Signal (coordinates) = "--",                                    'if both lights are now off,
Then     Index = 1-,      *Index = off      Endif    ENDSUB              'release the brakes of this loco
```

If the sub routine has been called by the PC attempting to automatically release the brake when a signal turns green the parameter 'Device' will equal 0 and the bottom lamp of the signal at these coordinates will be extinguished, however the top lamp will not change, therefore stopping the brake from releasing at the end of the sub routine.

If the sub routine has been called by the operator releasing the brake through one of the handheld controllers then both lamps of the signal are extinguished by the following line. Otherwise the sub routine must have been called by the operator applying the brake through one of the handhelds so both lamps of the signal are set to red ("RR"), the value of Index is reduced by one causing it to now point to the brake for this loco, and the brake is then turned on.

Finally the sub checks the state of the signal and if both lamps have been extinguished then the brake is removed by again pointing 'Index' to this loco's brake setting and turning it on.

Section 15 Train Movement

Before we get stuck into this in detail here's a quick overview of what we want the system to do.

- When a sensor turns on, the PC needs to check the adjoining blocks to work out where the train has come from
- The train stored on the loco variable for the old block is copied onto the loco variable for the new block. The old loco variable is then reset to 0. (Just like a beacon)
- If this is an up movement the new block is marked as occupied by giving it the value of 11, for down movements it is given the value of 12
- The new block being set to 11 or 12 triggers the 'Image' sub routine to draw the train's image and its schedule on the CTC screen
- The old block is given the value of 5 for up movements or 6 for down movements.
- The old block being set to 5 or 6 will still show the block as occupied (red) but cause the train's image and schedule to be erased from this block. When the sensor for the old block turns off it starts a timer to release the old block by resetting it to 0.

On my layout this is done through four different sub routines named:

- MoveTrain used for fairly simple moves where the program looks for a train on an adjacent block
- MoveTrainIR similar to the above but used for infrared sensors
- MoveTrainPlat used for platform blocks to check the blocks at both ends of the platform to determine where the train has come from.
- MoveTrainThroat is the most complex one and starts by checking the path across the turnouts at the end of a station and then finding the correct train to move.

The MoveTrain subroutine

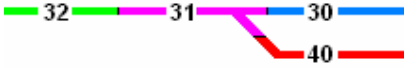
Let's start by looking at the MoveTrain sub routine. Use this sub for current detecting sensors. The sub checks the block on either side of this block to see where the train has come from. If there is a turnout involved the sub checks which way the turnout is set and checks the correct block.

The standard code to call the sub simply follows the form:

```
When S71 = on   Do      MoveTrain (71, 70, 75)          'when sensor 71 turns on check the adjacent blocks 70 and 75.
```

But when a turnout is involved we need to add an If statement so the PC knows which blocks to check

```
When S31 = on   Do      If $switch (1,19,1) = normal    Then    'if turnout set for block 30 then
                        MoveTrain (31, 30, 32)          Else    'check for a train on block 30 or 32
                        MoveTrain (31, 40, 32)          Endif   'otherwise check for train on block 40 or 32
```



The parameters for the MoveTrain sub are:

1. the index number of the new block
2. the block on the down side of the new block
3. the block on the up side

The sub starts by ensuring the local variable direction has been reset to 0, and then sets the other local variables to point to the address of the 3 blocks involved.

If the new block is already occupied (*newblock>4) then we don't want the sub to do anything as the sensor has probably turned off and on again due to dirty wheels, so the sub ends.

```
SUB MoveTrain (new, dn, up, newblock, upblock, dnblock, direction)
direction = 0                                                   'reset direction to 0
newblock = &B[new]                                              'newblock points to the address of the new block.
upblock = &B[up]                                                'upblock points to the address of the block on the up side.
dnblock = &B[dn]                                                'dnblock points to the address of the block on the down train.
If      *newblock > 4   then    Return                          'sensor may have been turned on by dirty wheels
ElseIf  *upblock = 12   then    direction = 1                   'train is moving down from the block on the up side
ElseIf  *dnblock = 11   then    direction = 2                   'train is moving up from the block on the down side
ElseIf  *upblock > 9    then    direction = 1                   'train is moving down
ElseIf  *dnblock > 9    then    direction = 2       Endif       'train is moving up
If      direction = 2   then    *newblock = 11, *dnblock = 5,   'mark new block 11 and old block 5 for an up train
                                L[new] = L[dn], L[dn] = 0       'copy beacon to new beacon and erase old
ElseIf  direction = 1   then    *newblock = 12, *upblock = 6,   'mark new block 12 and old block 6 for a down train
                                L[new] = L[up], L[up] = 0       'copy beacon to new beacon and erase old
Else    *newblock = 10          Endif   ENDSUB                  'if no trains around set new block to 10 for a new train
```

Then we start looking to see where this train has come from. Firstly we check the block on the up side of the new block. If there is a down train on that block then we get ready to move that train by setting 'direction' equal to 1. If not then we look at the block on the down side and if there is an up train on that block then we will move that train so direction is set to 2. If neither of those were true then a train may have changed direction on an adjacent block so now we check for any train on the up or down blocks (>9).

If this process has set direction equal to 2 for an up train movement then we now mark the new block equal to 11 and set the old block equal to 5 (waiting to cancel for an up train). We also copy the loco from the old block to the new one and cancel the old loco variable. If direction has been set to 1 for a down train then the following lines do the same thing but set the new block to 12 and the old block to 6.

If no trains were found on the adjacent blocks the new block is set to the value of 10 by the final line. By setting the block to 10 it marks it as occupied even though no train will be assigned to the block at this stage.

The MoveTrainIR subroutine

This sub is very similar to the MoveTrain sub but designed for infrared sensors.
The parameters are:

1. the block on the down side of the sensor.
2. the block on the up side.
3. the rescue block – if a train is blocking a sensor at start up it may be moved incorrectly so this rescue block puts it right again
4. the direction for the rescue block (2 if a train being rescued is moving in the down direction, 1 if moving up)

In the example to the right we have 3 blocks divided by infrared sensors S32 and S33. The code below would be used for Sensor 32.


```
When S32 = on    Do        MoveTrainIR (31, 32, 33, 2)
```

Block 31 is the block on the down side, Block 32 is the block on the up side, Block 33 is the rescue block, the direction is set to 2 because a train being rescued from Block 33 would have to be moving in the down direction.

If you don't need to use the rescue block feature you do not need to, just pass zeros through to the sub for the last 2 parameters, but if you do want to use it here's a bit more explanation. When a train on Block 33 reaches Sensor 33 it will correctly be moved onto Block 32. But if that train is still blocking the sensor during a reset or when the system is restarted the PC will see the train braking the sensor and incorrectly move the train back from Block 32 to 33. A few seconds later the train will reach Sensor 32 but the PC won't know what's going on because it thinks the train is on Block 33. So by passing the rescue block to the sub routine we are telling the PC: If sensor 32 turns on but there are no trains on Block 31 or Block 32 look at Block 33 and if there is a train there then move that one down to Block 31.

When a turnout is involved we use an if statement in the same way as we did for the previous subroutine. In this example sensor 96 protects the turnout at the western end of Midland Loop, this turnout forms Block 96 so as soon as the sensor turns on, Block 96 is marked as occupied by setting it to 10. However, the train is moved directly to the next block, either block 91 or 92 depending on the position of the turnout.

```
When S96 = on    Do        B[96] = 10,       If $switch (14,7,1) = normal      Then
                                               MoveTrainIR (91, 98, 0, 0)        Else
                                             MoveTrainIR (92, 98, 0, 0)

          Endif
```

The sub simply checks if there is a train on the up block (block 98 in this case) and if there is it is moved to the down block. Otherwise if there is a train on the down block it is moved to the up block.

```
SUB MoveTrainIR (dn, up, rescue, direction, upblock, dnblock)
upblock = &B[up]                    'upblock points to the address of the block on the up side.
dnblock = &B[dn]                    'dnblock points to the address of the block on the down train.
rescue =  &B[rescue]                'rescue = address of block at other end of the mainline
If        *upblock > 9                                          'if there is a train on the upblock
then      *dnblock=12, *upblock=6, L[dn]=L[up], L[up]=0         'move it to the dnblock
ElseIf    *dnblock > 9                                          'if there is a train on the dnblock
then      *upblock=11, *dnblock=5, L[up]=L[dn], L[dn]=0         'move it to the upblock
ElseIf    *rescue > 9       then    If      direction = 1       'if there are no adjacent trains check the
then      *dnblock = 12, L[dn] = L[rescue]                      'next block and move the train from there
Else      *upblock = 11, L[up] = L[rescue]  Endif
*rescue = 0                         Endif   ENDSUB
```

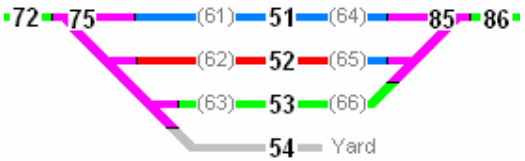
If the PC can't find a train on either block it will check to see if you have sent through a rescue block, if there is a train on that block it will move that train and reset the rescue block to 0.

The MoveTrainThroat subroutine

This sub routine looks after trains moving across the turnouts at the end of your platforms. Because everyone's layout is different you're going to have to do a bit of work here to customise the code for your layout, but it's not that bad. All we need to do is check the position of the turnouts to determine which platform is connected to which approach block, and then we call the MoveTrainThroat sub to do the rest of the job for us. You'll remember that right back at the beginning we gave each of our blocks a Block Variable, a Loco Variable and a Control Variable; well this is where the Control Variables earn their money.

Once we determine which platform the turnouts are set for we save the platform number into the Control Variable. The MoveTrainThroat sub and the MoveTrainPlat sub (that we will look at next) then use this information to painlessly manage train movements in and out of the most complex stations. The MoveTrainThroat sub also manages the movement of shunt locos entering an already occupied platform.

Normally if a 2nd loco enters an occupied platform the PC loses track of either the first or the second train. I like to do a lot of shunting so I designed phantom shunt blocks to store the second train's info. As well as the normal platform block (Blocks 51 to 53 in the example to the right) each platform also has a Shunt Block at each end (Blocks 61 to 63 and 64 to 66) that way the PC knows which loco is closest to which end of the platform and can therefore move the correct train when it needs to. These Shunt Blocks do not have a sensor or any physical identity, they only exist in software. Here's how they work:



When a 2nd loco enters an already occupied platform, instead of moving the train to the platform block it is stored on the Shunt Block for this end of the platform. Then when a train is detected leaving this end of the platform the PC first checks if there is a train stored on the shunt block. If there is, then it is this train that is moved, otherwise the main train is moved.

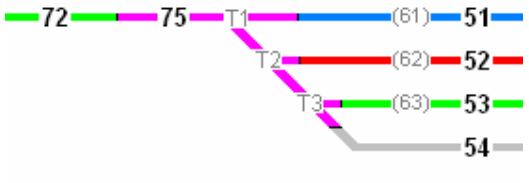
Note that for this sub to work you must have your platform blocks numbered consecutively starting at the first platform for this station. i.e. 51, 52, 53, 54 etc. Your next station can have any number range you like but again they must be consecutive to each other, for example 41, 42, 43 or 1, 2, 3, 4 etc. The Shunt Blocks must also be consecutive to each other (61, 62, 63) but the shunt blocks on the other end of the platform do not need to be in the same number range, for example they could be 77, 78, 79. You do not need to remember Shunt Blocks so just use any left over numbers in your array.

So let's have a look at the code.

You'll need to write some code like the example to the right for each of your throat blocks. This example is for Block 75 which is the throat block at the Western end of The Summit on my layout. When the sensor turns on the code checks that the block is vacant (<5), if not then the sensor has probably turned off and on again due to dirty wheels so this If statement prevents any action being taken.

```
When S75 = on   Do      If B[75] < 5        Then                    'when sensor 75 turns on and Block 75 is vacant
        If      $switch {T1}(22,1,1) = normal   Then    Control[75] = 1         'route is set for platform 1
        ElseIf  $switch {T2}(23,2,1) = reversed Then    Control[75] = 2         'route is set for platform 2
        ElseIf  $switch {T3}(24,3,1) = reversed Then    Control[75] = 3         'route is set for platform 3
        Else                                            Control[75] = 4 Endif   'route is set for the yard
        MoveTrainThroat (75, 72, 51, 61)                                Endif   'call the sub to move the right train
```

Then we check each of the turnouts in turn. (I've inserted T1, T2 and T3 into the code and the diagram to the right to make it easier to follow.) If T1 is normal then the route is set for Platform 1 so we set the Control Variable for this block (Control[75]) equal to 1.



If T1 is not normal we test T2 and if it is reversed then the route is set for platform 2 so we set Control[75] = 2. This continues until we have set the control variable for each platform and the yard track. Then the final line calls the sub routine to do the rest.

The parameters for this sub are:

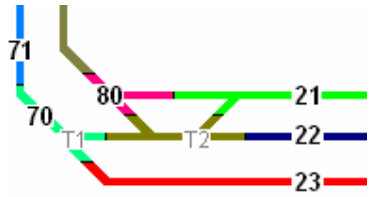
1. the index number of the new block – the throat block
2. the index number of the approach track
3. the index number of platform1 at this station

5-7. the index number of the shunt block for this end of platform1

You shouldn't need to change anything in the sub routine so you can simply cut and paste it into your code, however I will go through it in a moment for those who like that sort of thing. But first let's look at a few more examples of the code you may need to write. Here are a few more examples of code for my throat blocks.

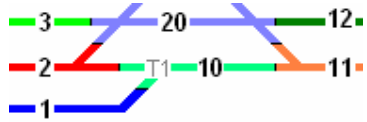
This one is for Block 70. Again I've added T1, T2 etc. in each of these examples so you can follow it.

```
When S70 = on   Do      If      B[70] < 5           Then                    'when sensor 70 turns on and Block 70 is vacant
        If      $switch {T1}(2,11,1) = normal       Then    Control[70] = 3         'route is set for platform 3
        ElseIf  $switch {T2}(5,11,1) = normal       Then    Control[70] = 2         'route is set for platform 2
        Else                                                Control[70] = 1 Endif   'route is set for platform 1
        MoveTrainThroat (70, 71, 21, 77)                                    Endif   'call the sub to move the right train
```



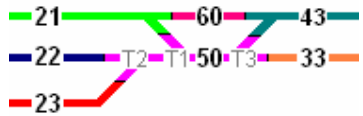
Block 10 only has 1 turnout to check as a train on Block 10 can only get to platform 1 or 2. So the code is simply:

```
When S10 = on   Do      If      B[10] < 5           Then                    'when sensor 10 turns on and Block 10 is vacant
        If      $switch {T1} (17,22,1) = reversed   Then    Control[10] = 1         'route is set for platform 1
        Else                                                Control[10] = 2 Endif   'route is set for platform 2
        MoveTrainThroat (10, 11, 1, 15)                                     Endif   'call the sub to move the right train
```



Note that the code below for Block 50 has to choose between 2 main line tracks on the other side of the throat. So the final 3 lines check T3 and call the sub routine with the correct 'approach' parameter.

```
When S50 = on   Do      If      B[50] < 5           Then                    'when sensor 50 turns on and Block 50 is vacant
        If      $switch {T1}(16,11,1) = reversed    Then    Control[50] = 1         'route is set for platform 1
        ElseIf  $switch {T2}(15,11,1) = normal      Then    Control[50] = 2         'route is set for platform 2
        Else                                                Control[50] = 3 Endif   'route is set for platform 3
        If      $switch {T3}(17,11,1) = normal      Then                    'check the turnout on the other side of the block
                MoveTrainThroat (50, 33, 21, 77)                                    'if T3 set for UpMain B[33]
        Else    MoveTrainThroat (50, 43, 21, 77)                Endif Endif     'if T3 set for Dn Main B[43]
```



Now let's have a closer look at the MoveTrainThroat sub routine. The sub starts by setting the local variables 'newblock' to point to the throat block and 'approach' to point to the approach block.

The next line uses the Control Variable to convert 'platform' to equal the index number for the correct platform. (i.e. 51, 52 , 53 or 54 in our example.)

Then this value is copied to 'index', and platform is then set to point to the correct platform block.

'Shunt' is also set to point to the correct shunt block for this platform.

```
SUB MoveTrainThroat (new, track, platform, shunt, index, newblock, approach)
newblock = &B[new]                                          'newblock points to the new block - the throat block.
approach = &B[track]                                        'approach points to the approach block
platform = -,  platform = Control[new]+,                    'platform = index # of platform turnouts are set for
index = platform                                            'index equals the index number of correct platform
platform = &B[platform]                                     'platform now points to the correct platform block
shunt = -,  shunt = Control[new]+,  shunt = &L[shunt]       'shunt points to the correct shunt block
If      *approach > 9    then    If *platform > 9           'if this is a shunt movement
        then    *newblock = 10,  *shunt = L[track]          'mark throat = 10 and copy loco to the shunt beacon
        Else    *newblock = *approach                Endif  'make throat block the same as approach and clear old
                *approach = 5, L[new] = L[track], L[track] = 0  'copy approach beacon to new and cancel the old beacon
ElseIf *platform > 9     then    *newblock = *platform,     'otherwise move the train from the platform but if
        If    *shunt > 0   then    L[new] = *shunt, *shunt = 0   'there is a loco on the shunt block move it.
        Else  L[new] = L[index], L[index] = 0, *platform = 6 Endif
Else    B[new] = 10              Endif                      'if no trains around mark the block = 10 for a new train
Control[new] = index     ENDSUB                             'set throat's control var.=to the index # of the platform
```

Now we can start checking for the right train to move. If there is a train on the approach track (*approach>9) and there is also a train on the platform block (*platform>9) then this must be a shunt movement so we set the throat block equal to 10 and copy the loco from the approach track straight onto this platform's shunt block. Otherwise if there is not a train in the platform then the train on the approach block is moved onto the throat block and the approach block is set to 5 ready to cancel.

If there was no train on the approach block then the sub checks the platform, if there is a train there then the throat block is marked as occupied by giving it the same value as the platform. Before moving the train in the platform the sub checks if there is a train stored on the shunt block at this end of the platform. If there is (*shunt>0) then it is the shunt loco that is moved, otherwise the train in the platform is moved onto the throat block and the platform is set to 6 ready to cancel.

If the sub has not found any trains on adjoining blocks then the second last line causes the throat block to be marked as occupied by setting it to 10 but no train is assigned to the block.

The very last line is important as it remembers which platform the turnouts are set for by copying the value of index into the Control Variable for the throat block. We're going to need that again in a moment when the train reaches the platform and activates the MoveTrainPlat sub routine below.

The MoveTrainPlat sub

The MoveTrainPlat sub is the last sub for handling train movements and as the name suggests it is called when a train is detected entering one of our platforms. Its function is to check the throat block at each end of the platform to find where this train has come from and move it onto the platform block. You'll be pleased to know that we are back to nice simple single line When Do statements to call the sub routine, such as the example below:

```
When S22 = on    Do        MoveTrainPlat    (22, 50, 0, 70, 80)
```

The parameters are:

1. the platform block – this example is for platform 2 at Upper Junction which is block 22

2-3) up to 2 throat blocks on the down side of the platform – in most cases there is only 1 throat block on the down side (block 50) so a 0 is passed through as parameter 3.

4-5) up to 2 blocks on the up side – in this case there are 2 throat blocks that connect to the up end of platform 2 depending on the turnouts (blocks 70 and 80)

The sub starts by setting ‘check’ to point to the platform block. ‘Block’ points to the first throat block on the down side of the platform and ‘throat’ points to the control variable for this thoat block.

If there is a train already on this block (>4) then the sensor has probably turned off and back on again due to dirty wheels so the sub ends without taking any action. Otherwise we go on to check the first block on the down end of the platform. If there is a train there (>9) and the control variable for this throat block shows that the turnouts are set for this platform (*throat=platform) then we set the platform block equal to 11 since the train is moving in the up direction. We also copy the loco from the throat block to the platform block and reset the throat block and its loco variable.

The sub now ends once the train has been moved, but if it didn’t find a train on the first throat block it performs the same tests on each of the other 3 blocks until it finds a train to move. If it gets all the way to the end without finding a train then it assumes a new loco has been placed on the track and sets the platform block to 10 to mark it as occupied.

```
SUB MoveTrainPlat (platform, A, B, C, D, check, block, throat)
check = &B[platform],     block = &B[A],   throat = &Control[A],
If      *check > 4        then Return                Endif    'dirty wheels may have turned on the sensor
If      *block > 9,       *throat=platform           Then     'if A is occupied and points set for this platform then
        B[platform]=11,   L[platform] = L[A],                 'move train from blockA to this platform - up train
        L[A] = 0,         B[A] = 5,                 Return
Else    block = &B[B],    throat = &Control[B]      Endif
If      *block > 9,       *throat=platform          Then     'if B is occupied and points set for this platform then
        B[platform]=11,   L[platform] = L[B],                 'move train from blockB to this platform - up train
        L[B] = 0,         B[B] = 5,                 Return
Else    block = &B[C],    throat = &Control[C]      Endif
If      *block > 9,       *throat = platform        then     'if C is occupied and points set for this platform
        B[platform]=12,   L[platform] = L[C],               'move train from blockC to this platform - down train
        L[C] = 0,         B[C] = 6,                 Return
Else    block = &B[D],    throat = &Control[D]      Endif
If      *block > 9,       *throat = platform        then     'if D is occupied and points set for this platform
        B[platform] = 12, L[platform] = L[D],               'move train from blockD to this platform - down train
        L[D] = 0,         B[D] = 6,                 Return
Else    B[platform] = 10  Endif     ENDSUB                  'otherwise mark block = 10 for a new train
```

Clearing blocks once the train has passed.

You may have noticed that the sub routines we just looked at have not been resetting the old block to 0 but rather to 5 for up trains and 6 for down trains. That’s because we don’t want to release the previous blocks immediately. By setting them to 5 or 6 we are telling the PC the blocks are ready to be released as soon as the sensor for the old block turns off and any inbuilt delay time has been observed.

The following code is used to check the old block and clear it when the sensor for the old block turns off. You’ll see on my code that the code for some sensors include a wait time of 6 seconds but others don’t. The ones that have no delay time are generally infrared sensors where the sensor will not turn off until the whole train has safely passed the sensor. I haven’t got around to installing resistive wheel sets yet so for current detecting sensors I (scarily) assume the train will have cleared the block within 6 seconds of the loco moving out of the block.

```
When S86 = off   Do      wait 6    ClearBlock (86)          or for throat bocks          When S40 = off   Do      wait 6    ClearThroat (40)
```

Throat blocks are slightly different because we need these blocks to clear if the block = 5 or 6 or 10. This allows the block to clear after shunt locos have been moved onto a shunt block.

```
SUB ClearBlock (block)   block = &B[block]     If *block = 5 or   *block = 6     then *block = 0   Endif   ENDSUB
SUB ClearThroat (block)  block = &B[block]     If *block < 11,    *block > 4     then *block = 0   Endif   ENDSUB
```

Section 16 Train Image and Schedule Display – The Draw sub routine

This section of code can be used to display an icon or an actual image of your locos as well as the schedule the train is on. As you can see from the calling script below the train's image and schedule is displayed on a block as soon as that block is occupied (value greater than 9), and is erased again as soon as that block drops below 9.

```
When B[21] > 9    Do    Draw (21, 0, (9,10,1), 1)        When B[21] < 10    Do    Erase ((8,10,1))
```

The parameters for the Draw subroutine are:

1. The block the train is on
2. The direction – 0 if an up train in this block should appear facing left, 1 if an up train should face right.
3. The coordinates for the left end of the loco's image
4. The display control – 0 to display the loco's image but not the schedule on this block, 1 to display both the loco and the schedule

The only parameter needed for the Erase sub is the grid coordinate one to the left of the one you used for the Draw sub.

If you want to make your own images for displaying your locos simply set up a suitable photo shoot. I sat my loco on a sheet of white paper and then curved the sheet up the wall behind the loco to form a backdrop, then used a couple of desk lights as studio lights so I could get in close with the camera and turn the flash off. Try to take the photo exactly side on for the best results.

```
SUB Draw (index, facing, coordinates, display, schedule, block, loco, spiral)
block = &B[index],  loco = L[index],  loco = 2724-,  loco = 13/,                    'sets loco = to loco #.  0,1,2,3,4,5, etc.
If      loco = 1          Then    $draw picture (coordinates) = "C:\Program Files\TBrain\V100a.jpg",
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\V100b.jpg",
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\V100c.jpg",
ElseIf  loco = 3          Then    If *block=12, facing=0 or *block=11, facing=1
                          Then    $draw picture (coordinates) = "C:\Program Files\TBrain\Class03righta.jpg"
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\Class03rightb.jpg"
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\Class03rightc.jpg"
                          Else    $draw picture (coordinates) = "C:\Program Files\TBrain\Class03a.jpg"
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\Class03b.jpg"
        coordinates = +,          $draw picture (coordinates) = "C:\Program Files\TBrain\Class03c.jpg"  Endif     Endif
<keep copying the code above for the rest of your locos>
If      display = 0               Then    Return   Endif                            'no schedule display needed
Schedule = Schedule[loco]                                                           'copy Train's schedule to local variable
If      schedule > 9      Then    Until   schedule < 10                             'strip schedule down to a single digit number
                                  Loop    schedule = 10-   Endloop Endif            'to make it easier
If      schedule > 2      Then    schedule = 3-            Endif                    'reduces schedule to 0, 1, 2 depending on route.
If      *block =  11      Then                                                      'if its an up train
        If Facing = 1     Then    coordinates = 3-  Else coordinates = +  Endif          'set coordinates for the up schedule
Else    If Facing = 0     Then    coordinates = 3-  Else coordinates = +  Endif Endif    'set coordinates for the down schedule
If      schedule = 0      Then    $draw picture (coordinates) = "C:\Program Files\TBrain\70.jpg",          'clockwise train
ElseIf schedule = 1       Then    $draw picture (coordinates) = "C:\Program Files\TBrain\80.jpg",          'anticlockwise train
Else                              $draw picture (coordinates) = "C:\Program Files\TBrain\99.jpg", Endif 'midlandtrain
ENDSUB
```

Once you have your photo on the computer you can edit it in a program like Microsoft Paint (found under Accessories on the program list) you'll need to crop it and shrink it until the image is 25 bits high by 75 bits wide. Then you need to chop it up and save it as 3 separate images each 25 by 25. For my V100 diesel I created 3 images named V100a.jpg (the left end) V100b.jpg (middle) and V100c.jpg (surprisingly enough the right hand end).

Using 3 small images rather than 1 means the program doesn't need to redraw the full screen when erasing train images. The smaller pictures fit neatly into each of the 3 grid squares used for each image. (The $erase command requires the full screen to be redrawn while the $erase smallpicture command only redraws each individual grid square. Redrawing of the full screen can cause some track elements to be drawn over the top of the loco images and wastes processor time).

Once you've got your images prepared you'll need to save them into your TBrain folder and then edit the sub routine to tell it which images to use for each loco. The sub routine starts by setting the local variable 'block' to point to the block the train is on and then sets Loco to equal the pure loco number of the loco on the block. i.e. the first loco listed in

your CTI fleet roster will be loco 0, the 2nd loco listed will be loco 1 etc. You'll then need to copy the next 3 lines for every loco in your fleet. The first line tells the PC which image to use for the left hand end of this loco, the 2nd line is the middle and the 3rd line is the right hand end.

Use the example in the sub for Loco 1 for all your locos that are pretty much symmetrical, but for steam locos or other locos that have an obvious front and back you'll probably want a right facing version as well as a left facing version. You can create those images by just flipping the images you already have and saving them with a new name. You'll see that the example for loco 3 in the sub is my class 03 steam loco which has the usual images (class03a, b and c) plus 3 additional right facing jpg files. Just copy the lines as they appear here and the sub will work out which ones to use when.

The second half of the sub deals with the schedule display. I have created a jpg file for each schedule that looks something like this 

. But you could use a symbol, picture or whatever you want, just make it fit into a 25 by 25 bit jpeg or bitmap. We'll talk more about schedules when we get to the train properties area soon, but for now all you need to do is tell your PC the name of the file to use for each schedule in the last 3 lines of the sub. Of course if you have more than 3 schedules just add the extra ones at the bottom and you will need to edit the line that starts If schedule > 2 to reflect the number of different schedules you have.

The Erase sub simply starts at the left most grid square and erases the 5 squares used for displaying the loco and schedule.

```
SUB Erase (coordinates, count)
count = 0,          Until count > 4,   Loop     $erase smallpicture (coordinates), coordinates = +,     count = +,          Endloop ENDSUB
```

Section 17 New Train Commands

This section allows the operator to assign a new train to a track block by clicking on a block on the CTC screen and then choosing the train from a list. When a block is left clicked the code below calls a sub routine called AssignTrain, with the following parameters:

1. the index number for this block
2. the left most grid square for drawing the locos image on the display

```
When $left_mouse = (10,10,1)        Do        AssignTrain (21, (9,10,1))
```

```
SUB AssignTrain (index, coordinates, beacon, block)
beacon = &L[index]          block = &B[index]
Wait until $QueryBusy = false       Then     $Query"1$Which train do you want to assign?
$cancel request$empty wagons$V100$Steam$Re44$DB101$Track Machine"
Wait until $QueryBusy = false                Then
          If        $QueryResponse = 0       then      Return
          ElseIf    $QueryResponse = 1       then      *beacon = &wagons
          ElseIf    $QueryResponse = 2       then      *beacon = &V100
          ElseIf    $QueryResponse = 2       then      *beacon = &Steam
          ElseIf    $QueryResponse = 4       then      *beacon = &Re44
          ElseIf    $QueryResponse = 5       then      *beacon = &DB101
          ElseIf    $QueryResponse = 6       then      *beacon = &W740          Endif
If        *Block > 9                         then      Draw (index, 0, coordinates, 0)
                                             else      *block = 10        Endif     ENDSUB
```

Firstly the sub sets the local variable 'beacon' to point to the loco on the block, and sets block to point to the block. It then ensures no other query windows are open and a pop up window asks the operator; "Which train do you wish to assign?" and a number of buttons to select from. You'll obviously need to insert the names of your locos here instead of mine. The 2nd statement of 'Wait until $QueryBusy = false' ensures the sub routine waits for the operator to make their selection before testing the response. If the 1st button (cancel request) is selected the sub ends without taking any action, otherwise the appropriate loco or rolling stock is assigned to the loco variable for that block. Finally the sub checks to see if the block is already occupied (*block>9) and if it is the Draw sub is called to draw the new loco on the block. Otherwise the block is set to 10 to mark it as occupied.

Section 18 Block/Train Cancel by Mouse Click

Whenever a block is right clicked the block is cancelled by setting its value to 0. Any train currently assigned to that block will also be removed.

```
When $right_mouse = (10,10,1)  Do  B[21] = 0, L[21] = off,
```

Section 19 Train Properties and Throttle Commands

The Train Properties section of the CTC display allows the operator full control of each train with the click of a mouse, as well as the ability to set the train's route, schedule, and driving mode. The examples given throughout this section are generally for loco #1, my V100 diesel loco. You'll need to repeat the code for each loco within your fleet. Please note that any changes to speed or other functions made by the PC will be reflected on the display of the DCC controller, however the interface between the PC and DCC system is not 2 way. So if a train is being driven manually by the DCC system the PC will not reflect those changes. For example if a train is started by the PC but the operator stops the train using the DCC controller the train will stop but its speed will remain unchanged on the PC display. Most of these features can also be operated from the handheld controllers.

Speed Display and Controls

The speed of each train is permanently displayed beside each train's image by the Always Do statement below. The speed setting is copied to the train's speed variable and this is then displayed as a text message.

```
Always Do  speed[1]=V100.speed  $Draw Message (33.5,14,1) = "@speed[1]", $color sprite (33,14,1) = white
```

Left clicking on the locos image in the Properties area or typing S#F will throttle the loco up to its cruising speed while right clicking on the image or typing S#0 will reduce its speed to zero. (Replace the # symbol with the loco number). For example:

```
When  $left_mouse = (29-31,13,1)   or $command = S1F   Do  V100.speed = 12
When  $Right_Mouse = (29-31,13,1)  or $command = S10   Do  V100.speed = 0
```

Left clicking on the speed display will increase the speed by one throttle notch. Right clicking will reduce its speed by one.

```
When  $left_mouse = (34,14,1)   Do  V100.speed = +
When  $Right_Mouse = (34,14,1)  Do  V100.speed = -
```

Typing S#H is a short cut to set the loco's speed to approximately half speed.

```
When  $command = S1H  Do  V100.speed = 9
```

Braking Display and Controls

The state of the brake is displayed alongside each loco in the train properties area by a signal with 2 red lamps. If the brake is off then both lamps will be extinguished. If the brake has been applied automatically by the PC then the bottom lamp will be red. If the brake has been applied manually then both lamps will be red. This has been done so that the automatic braking section does not release the brake if the driver has manually stopped the train, even if the signal turns green (see the train braking and acceleration section for more details). The script below sets the bottom lamp to red whenever the brake is turned on.

```
When V100.brake = on  Do  $Signal (34,13,1) = "xR"
```

When the operator left clicks on the brake display or types B# at the keyboard the code below checks if the brake is already on, if it is then the brake is removed and both lamps of the display are extinguished. If the brake is not on then the brake is applied and both lamps are turned red.

```
When  $left_mouse = (34,13,1) or $command = B1        Do If V100.brake = on   Then  V100.brake = off $Signal (34,13,1) = "--"
                                                                              Else  V100.brake = on          $Signal (34,13,1) = "RR"  Endif
```

The code below allows the brake to be applied to all trains simultaneously by typing BA (brake all) at the keyboard.

```
When $command = BA   Do   Index = &V100.brake,   Until Index > 2850   Loop   'set index = to address of 1st loco's brake
                          *Index = on   Index = ++   Endloop Index = 0       'turns brake on for all trains
$Signal (34,13,1) = "RR", $Signal (34,15,1) = "RR", $Signal (34,16,1) = "RR", $Signal (34,17,1) = "RR", $Signal (34,19,1) = "RR"
```

Light Display and Controls

The state of each loco's headlight – either on or off – is displayed using a white lamp on a single aspect signal alongside each loco's image in the Train Properties area of the CTC screen. The operator can toggle the headlight on and off by clicking on the signal or typing L# at the keyboard (#=loco number).

```
When V100.Fl = on     Do   $Signal (36,13,1) = "W"
When  V100.Fl = off   Do   $Signal (36,13,1) = "-"
When  $left_mouse = (36,13,1) or $command = L1   Do   If V100.Fl = on  Then  V100.Fl = off  Else V100.Fl = on  Endif
```

Direction Display and Controls

```
When  $left_mouse = (35,13,1) or $command = D1   Do   Direction (1)
```

The operator can toggle the direction of each train by clicking on the arrow beside each loco's image or typing D# at the keyboard. The only parameter you need to pass through to the sub is the loco number.

This sub is also used when one of the handheld controllers is trying to change direction so the top 2 lines convert the index number to the pure loco number of 0, 1, 2 etc.

```
SUB Direction (index, coordinates, schedule, flash)
If *index > 2720                                          'if this sub has been called by a HandHeld controller
Then    index = *index, index = 2724-, index = 13/, Endif 'convert index from CTI address to loco # (0,1,2,3 etc.)
Coordinates = index, Coordinates=50*, Coordinates=8172+,  'sets coordinates = to address of the direction display
                                                          'for this loco (8222, 8272 etc.)
Index = 13*,    Index = 2724+,   Index = +                'sets Index to the CTI address of this loco's direction
If *Index = 0   Then   *Index = 1, Flash=0, Until Flash=2 'if loco is currently forward change to rev and flash
        Loop    $erase sprite(coordinates) Wait 0.3       'arrow symbol 3 times then leave arrow on
                $Draw Sprite(coordinates) = Arrow_West in Yellow
                Wait 0.3 Flash = +   Endloop
Else            *Index = 0,   Flash=0, Until Flash=2      'else if loco currently reversed change direction and
        Loop    $erase sprite(coordinates) Wait 0.3       'flash arrow symbol again before leaving on
                $Draw Sprite(coordinates) = Arrow_East in White
                Wait 0.3 Flash = +   Endloop Endif   ENDSUB
```

Then the sub calculates the coordinates for the direction display for the chosen loco. The CTC screen is 50 squares wide so if you set your display up with the locos above each other all you need to do is find the CTI address of the grid square for the direction display of your first loco and enter it instead of '8172'.

To find out the CTI address of a coordinate enter some temporary code such as `When $command = test  Do B[1] = (x,y,z),  $status = "@B[1]"` and the CTI address will be conveniently displayed on the status bar for you when you type 'Test' at the keyboard. Appendix A also has more info about CTI addresses.

The sub now sets 'index' to point to the direction setting for this loco and if the direction is currently forward (*Index=0) then it is changed to reverse (*Index=1). The following lines cause the arrow to change direction and flash 3 times before staying on.

Mode Display and Controls
The programming allows for individual trains to be driven in automatic or manual mode.

Manual Trains leave driving and route setting up to the driver or controller; however they will stop at red signals if the driver does not respond in time. Features such as station stops can still be turned on for manual trains if desired to force trains to stop even if the driver forgets about his poor passengers. The best means of driving manual trains is via the mouse / keyboard or handheld controllers.

Automatic trains can be left to run fully automatically under control of the PC with the train starting and stopping at signals. The PC will also set the appropriate route ahead of the approaching train and will observe any station stops required. The operator can still take control of these trains at any time by using the mouse / keyboard or handhelds, in fact I leave my trains in automatic mode most of the time and drive the one(s) I want with one of my handhelds and leave the other trains to the PC.

If the train is currently automatic the push button in the Train Properties area will be green. If it is manual it will be yellow.

The code below calls the Mode sub when someone left clicks on the push button or M# is typed at the keyboard.

```
When    $left_mouse = (37,13,1) or $command = M1        Do      Mode (1)
```

The sub routine copies this train's mode into the local variable 'mode'. It then finds the correct grid square for the push button for this loco. To tailor this to your layout simply replace (37,12,1) with the grid coordinates of the push button for your first loco (loco 0).

```
SUB Mode (index, coordinates, mode, grid)
mode = mode[index]                                          'copy current mode to local variable. 0=auto, 1=manual
grid = index,  grid = 50*,
coordinates =  &(37,12,1),  coordinates = grid+         'points coordinates to the grid square for this loco's Mode display
If mode = 1 then  mode[index] = 0,  $color track (coordinates) = Green,                         'change to auto train
Else              mode[index] = 1,  $color track (coordinates) = Yellow        Endif ENDSUB   'change to manual mode
```

If the train is currently manual (mode=1) then the mode variable for this loco is changed to 0 and the button is coloured green. Otherwise the mode is set to 1 and the button turns yellow.

Schedule Controls
When an automatic train approaches a red signal the PC will attempt to set the points and signals ahead of it. In order for that to happen each loco has its own Schedule Variable which stores information on which route this train is to take. It also records whether this train is to stop at stations or be held at the next station. The values for my schedule variables are:

0=clockwise train,	3=random, will take clockwise option at next junction	+10 if train is to stop at stations, i.e. values between 10 and 15
1=anticlockwise train,	4=random, will take anticlockwise option	+20 if train is to be held at next station, i.e. values 20 to 25 or 30 to 35
2=midland bound train,	5=random, will take midland line	

Clockwise Trains – this is the default schedule. Trains turn in the base tunnel and at Upper Junction they take the more direct 'clockwise' route to the Summit
Anticlockwise trains – also use the Base Tunnel but branch off onto the Short Line at Upper Junction and go anticlockwise via the Summit
Midland Trains – branch off onto the Midland Line at either Midland Junction or Lower Junction. These trains can be running in either the up or down direction

If you have more than 3 schedules that you want to assign trains to you'll need to edit a number of the sub routines that use the schedule variable.

The operator can change each train's schedule at any time or set the train as random. If the train is random the PC will set the train's schedule to either 3, 4 or 5 and change it each time the train passes a junction to provide maximum variety. (The Layout Overview section gave a brief explanation of the possible routes). I use an addressable 9 lamp signal to display the current schedule for each train in the Train Properties area.

The code below allows the operator to change a train's schedule by clicking on the schedule display in the Train Properties area or typing R# on the keyboard. (#=loco number). The schedule display is also redrawn when the system is reset.

```
When $left_mouse = (38,13,1)    or $command = R1    or $reset=true    Do Schedule (1)
```

This calls a sub routine called Schedule with the loco number as the single parameter. The sub starts by setting the local variable 'stopping' to 1 if the train is a stopping train, once it has done that then the schedule can be reduced to a single digit number. Now it's a simple case of checking what schedule the train is currently set to and changing it to the next one. If it is currently 0 then it changes to 1 etc. When it changes to 3 (for a random train) then the sub Random is called to choose a route (either 3, 4 or 5). Finaly if stopping had been set to 1 earlier then we add 10 to the value to again mark it as a stopping train, and the sub then passes the loco number on to the schedule display sub which redraws the 9 lamp signal for us.

You don't need to add any calling code for the ScheduleDisplay sub as it is called directly by the Schedule sub as well as by the Hold sub and Platform Sub whenever they change the value of a train's schedule. Just copy the sub in and you are done (you need to insert it higher in the list of subs than the Hold, Platform and Schedule subs).

It starts by calculating the grid coordinates for the schedule display for this loco. To make this work on your layout simply replace '8175' with the CTI address of the grid coordinates of the display for your first loco. (See Direction Display for info on working out CTI addresses).

Then it reduces the schedule value down to a single digit number to make it easier to work with. If the schedule is 1 (anticlockwise train) then the signal is drawn with 3 yellow lamps angled from top left to bottom right. If the schedule is 0 (clockwise train) the signal has 3 horizontal yellow lamps. If it is random (>2) then all lamps are extinghuished. If it is 2 (Midland Train) then the signal is drawn with 3 yellow lamps angled from bottom left to top right.

```
SUB Schedule (index, schedule, stopping)
schedule = schedule[index]                    'copy train's schedule to local variable, value is 0-5, 10-15, 20-25 or 30-35
If      schedule > 9     then    stopping = 1,                          'set 'stopping' = to 1 for stopping trains
Until   schedule < 10    Loop    schedule = 10-    Endloop  Endif       'value now 0-5
If      schedule = 0     then    schedule[index] = 1,                   'if the schedule is currently 0 change to 1
ElseIf  schedule > 2     then    schedule[index] = 0,                   'if it is currently random (>2) set to 0
ElseIf  schedule = 2     then    schedule[index] = 3, Random(index)     'the random sub is called to choose the route
ElseIf  schedule = 1     then    schedule[index] = 2,        Endif      'if it is currently 1 change to 2
If      stopping = 1     then    schedule[index] = 10+,                 'if it is a stopping train add 10 to the
                                 stopping = 0,               Endif      'train's schedule and reset 'stopping' to 0
ScheduleDisplay (index)          ENDSUB                                 'ScheduleDisplay redraws the schedule display
```

```
SUB ScheduleDisplay (index, coordinates, schedule)
schedule = schedule[index]          'copy train's schedule to local variable, value is 0-5, 10-15, 20-25 or 30-35
index = 50*, Index = 8175+,         'sets index = to address of the schedule display for this loco, (8225, 8275 etc.)
If      schedule > 9     then    Until schedule < 10                    'strip schedule down to a single digit number
                         Loop    schedule = 10-    Endloop Endif        'value is now in the range 0-5
If      schedule = 1     then    $signal (index) = "Y---Y---Y"          'if it is a Clockwise train set signal display
ElseIf schedule = 0      then    $signal (index) = "---YYY---"          'ditto for other schedules
ElseIf schedule > 2      then    $signal (index) = "---------"
ElseIf schedule = 2      then    $signal (index) = "--Y-Y-Y--"          Endif     ENDSUB
```

Setting Passenger Trains to stop at Platforms

The operator can instruct the PC to stop trains at each station for a short period. This is done by increasing the train's 'schedule' variable by the value of 10. Hence trains with a schedule value of 0 to 9 will operate as express trains, while trains with a schedule value greater than 9 will stop at each station. In essence this is done by delaying the setting of the outgoing route from the platform, leaving the departure signal red until the preset station stop time has elapsed (see section 4 for details).

The operator can toggle this feature on or off by typing P# on the keyboard (where # = the loco number) or clicking on the appropriate location of the Train Properties area of the CTC screen. I use a small picture of one of my stations as an icon but you could use any symbol you liked. This symbol is drawn alongside the schedule display in the train properties area when station stops are turned on for this loco.

The script below draws the picture of a station building when the loco's schedule variable is greater than 9, and erases it when the variable is less than 9.

```
When  Schedule[1] > 9    Do  $Draw Picture (39,13,1) = "C:\Program Files\TBrain\Station.jpg"
When  Schedule[1] < 9    Do  $Erase SmallPicture (39,13,1)
```

[This repeats for each loco]

The following line turns this feature on or off by increasing or decreasing this loco's schedule by 10 with a mouse click or the keyboard command P#.

```
When  $left_mouse = (39,13,1) or $command = P1  Do If schedule[1] > 9  Then  schedule[1] = 10- Else  schedule[1]=10+ Endif
```

Holding Trains at the Next Station

The operator can also instruct a train to stop at the next station and not proceed until a route is manually cleared for it. The code below calls the Hold sub when the operator types H# at the keyboard. Typing HA will also cause all trains to stop and be held at the next station (handy when shutting down at the end of the day). The reference to L[7] allows me to turn this feature on from a function button on my handhelds.

```
When $command = H1  or  L[7] = &V100  or $command = HA  Do  Hold (1, (38,13,1))
```

The parameters for the Hold sub are the loco number and the coordinates of the schedule display (the 9 lamp signal). Again simply replace '8175' with the CTI address for the coordinates of the schedule display for your first loco to tailor it to your layout. When the sub is called it adds 20 to the current value stored on this train's schedule variable and draws the signal with 9 white lamps. Or if the train has already been set to be held then 20 is subtracted and the ScheduleDisplay sub is called to redraw the schedule.

```
SUB Hold (index, coordinates, schedule)
schedule = schedule[index]                                  'copy this train's schedule to the local variable
Coordinates = index,  Coordinates=50*, Coordinates=8175+,   'sets coordinates = to address of the schedule display
                                                            'for this loco (8225, 8275 etc.)
If schedule < 20  Then    schedule[index] = 20+,            'add 20 to the train's schedule
                  $signal(coordinates) = "wwwwwwwww"        'and set the schedule display to show all lights white
Else              schedule[index] = 20-,                    'else if the train is already set to be held take 20 off
                  ScheduleDisplay(index) Endif ENDSUB       'and call ScheduleDisplay to redraw the schedule
```

The Automatic Route Setting section of the script (Section 4) ensures the departure signal remains red and the train is stopped at the station. It also ensures the schedule variable is returned to its previous value once the stop has been completed by subtracting 20 from it.

Section 20 Keyboard Commands for Direct Point Control

Usually turnouts are operated as part of a route however it is possible to operate individual turnouts through keyboard commands, by typing 'T' followed by the number of the turnout to operate the points to the opposite position. The first example below simply checks the current position of the turnout and throws it the other way.

```
When $command = T1    Do If $switch (2,22,1) = Normal
                          Then    $switch (2,22,1) = Reversed    Else    $switch (2,22,1) = Normal    Endif
```

This second example is for a crossover that involves 2 turnouts that always operate together.

```
When $command = T30   Do If $switch (1,19,1) = Normal
                          Then    $switch (1,19,1) = Reversed    Else    $switch (1,19,1) = Normal    Endif
                          When $switch (1,19,1) = Normal         Do      $switch (2,20,1) = Reversed
                          When $switch (1,19,1) = Reversed       Do      $switch (2,20,1) = Normal
```

The double slips are more complex. In the example below turnout 10 operates with the standard code we saw above. Its position then triggers a change in the position of the double slip. If the double slip is already straight through on the diagonal then it is changed to turnout from the diagonal to the horizontal etc.
The values for the double slips are:

0. Straight through on the horizontal
1. Straight through on the diagonal
2. Turnout from the horizontal to the diagonal
3. Turnout from the diagonal to the horizontal

```
When $command = T10   Do If $switch (20,22,1) = Normal
                      Then $switch (20,22,1) = Reversed    Else    $switch (20,22,1) = Normal    Endif
                          When    $switch (20,22,1) = Normal                  Do
                          If      $switch (19,21,1) = 1                       Then    $switch (19,21,1) = 3
                          Elseif  $switch (19,21,1) = 2                       Then    $switch (19,21,1) = 0    Endif
                          When    $switch (20,22,1) = Reversed                Do
                          If      $switch (19,21,1) = 3                       Then    $switch (19,21,1) = 1
                          Elseif  $switch (19,21,1) = 0                       Then    $switch (19,21,1) = 2    Endif
                          When    $switch (19,21,1) = 1 or $switch (19,21,1) = 2  Do  $switch (20,22,1) = Reversed
                          When    $switch (19,21,1) = 0 or $switch (19,21,1) = 3  Do  $switch (20,22,1) = Normal
```

Section 21 Point Motor Controls

The operation of the solenoids to change the physical points on the layout is tied to the position of the turnouts on the CTC screen. In the example below when the image of Turnout 11 on the CTC screen changes to the normal position, a 0.1 second pulse is sent to the solenoid on the normal side of turnout 11. The same occurs when the turnout reverses.

```
When $switch (15,7,1) = Normal      Do      T11N = Pulse 0.1
When $switch (15,7,1) = Reversed    Do      T11R = Pulse 0.1
```

Section 23 System Reset Commands

You will see a number of commands that are performed on my layout when the reset button is pressed. These include ensuring all turnouts are in the position shown on the CTC screen, resetting all routes, and applying the brake to all trains.

Section 24 Handheld Controllers

The code for the handheld controllers comes with the installation pack for the handhelds. These handhelds can be used for driving trains, setting signals and routes, and includes additional buttons for your favourite features. See www.users.tpg.com.au/bsken for more info.

I have written the code that comes with the installation pack so that it can be easily copied into anyone's code, but if you have already set up loco variables and block variables and used some of the sub routines from this file you can take some shortcuts. Follow the instructions that come with the installation pack but with these changes.

- When you come to install the code you won't need to set up Loco[n] if you have already set up Loco Variables.
- You also won't need to set up beacons for your handsets. Simply find enough spare Loco Variables to allocate 1 for each of your handsets (they must be consecutive). And in the second line of the Handheld sub replace '&beaconHH1' with '&L[n]' where n is the lowest of the index numbers you found for your handsets.
- You need to replace these beacons in the OnboardSignal sub as well. i.e. replace '&beaconHH1' with '&L[n]', and replace '&beaconHH2' with '&L[n+1]' etc.
- About 8 lines into the Handheld sub you'll find the line "NewLoco = &Loco[Slot2]". Replace that line with "NewLoco =&L[Slot2], NewBlock = &B[Slot2]"
- Half way through you'll also find the line "ElseIf Slot1 = 7 Then *NewLoco = Loco" Replace it with the 3 lines below.

```
ElseIf Slot1=7   then    *NewLoco = Loco                    'copy the loco onto the new block
    If           *NewBlock>9  Then   *NewBlock = 8          'if the block is already occupied then set it to
                 Wait 0.2     Endif  *NewBlock = 10         '8 temporarily to force it to redraw. Then set it to 10.
```

- Similarly for the line a little further down that says "ElseIf Slot1 = 10 Then *NewLoco = off" Endif Replace this line with

```
ElseIf Slot1=10  then    *NewBlock = 0,  *NewLoco = off  Endif   'copy the loco onto the new block
```

- If you are using the BrakeControl sub go to the line "ElseIf Digit = 2 Then Loco = 2+, *Loco = off" and replace it with

```
ElseIf Digit=2   then    BrakeControl (Beacon, 60, 1)       'if button 2 was the first button pressed then remove the brake
```

And also go to the line "ElseIf Digit = 5 Then Loco = 2+, *Loco = on" and replace it with

```
ElseIf Digit=5   then    BrakeControl (Beacon, 60, 2)       'if button 5 was the first button pressed then apply the brake
```

- If you are using the Direction sub go to the line that starts "ElseIf Digit=8 Then" and replace it and the following 2 lines with

```
ElseIf Digit = 8  Then   Direction (Beacon)                 'if button 8 is the first button pressed change the direction
```

- You can also skip over the sections entitled:

- Enter Code for Setting Signals and Routes – if you have already set up RouteEntry (see section 2 of this file)
- Enter Code for Cancelling Signals – if you have set up SignalEntry (see section 13)
- Enter Code for Assigning Trains to Blocks
- Enter Code for Canceling Blocks

- When installing the code for the onboard signaling use the following example for each of your blocks. The parameters are the Loco Variable for this block, followed by the coordinates of the signal on the up side of this block, and the signal on the down side.

  While B[81] > 9 Do OnboardSignal (L[81], (13,3,1), (2,8,1))

Appendix A CTI Addresses

Each loco in the DCC fleet roster has a permanent memory location allocated when the loco is first created. The first loco created will be stored at the CTI address 2724, and each subsequent loco will have an address 13 higher (the intermediate addresses are used to store each train's direction, speed, brake, momentum and functions as per the right hand table below.)

Loco number	CTI address
1st loco created	2724
2nd loco	2737
3rd loco	2750
4th loco	2763
5th loco	2776
6th loco	2789
7th loco	2802

Loco function	CTI sub address (for loco 1)
Speed	2724
Direction	2725
Brake	2726
Momentum	2727
Light	2728
F1	2729
F2 to F7	2730 to 2736

Each grid coordinate also has a CTI address. The top left square (1,1,1) has the address 7588. The CTC screen is 50 squares wide so the address for the first square on the second line can be calculated by adding 50 to this figure. The table below shows the grid squares within the Train Properties area which begins at (33,13,1).

	Speed	Brake	Direction	Light	Mode	Schedule	Station
	(33,13,1)	(34,13,1)	(35,13,1)	(36,13,1)	(37,13,1)	(38,13,1)	(39,13,1)
V100	8220	8221	8222	8223	8224	8225	8226
[2]	8270	8271	8272	8273	8274	8275	8276
Steam	8320	8321	8322	8323	8324	8325	8326
Re44	8370	8371	8372	8373	8374	8375	8376
DB101	8420	8421	8422	8423	8424	8425	8426
[6]	8470	8471	8472	8473	8474	8475	8476
W740	8520	8521	8522	8523	8524	8525	8526
	(33,19,1)	(34,19,1)	(35,19,1)	(36,19,1)	(37,19,1)	(38,19,1)	(39,19,1)

Appendix B Block, Loco and Throat Arrays

The program uses 3 sets of variables indexed between 0 and 99. The program uses the index value to swap between the block in question and the loco currently on that block.

- B[index] are track blocks and take the value of 0 to 12 depending on their status at the time, such as occupied by down train, cleared for an up train, idle etc.
- L[index] are loco variables and store the CTI address of the loco currently assigned to the related block. i.e. 2737 etc. See Appendix A for a list of CTI addresses.
- Control[index] are variables used to record additional info such as the current path across a series of turnouts or schedule info for a train currently waiting at a platform.

Index (blocks, beacons, throats)

1	Lower	1		34				67	Upper	1	East
2	Lower	2		35				68	Upper	2	East
3	Lower	3		36				69	Upper	3	East
4	Lower	4		37				70	Upper	70	
5	Lower	5		38				71	Summit	Line	1
6	Lower	yard		39				72	Summit	Line	2
7	HoldTrain			40	Lower	40		73			
8	HandHeld1			41	Dn Main	1		74			
9	HandHeld2			42	Dn Main	2		75	Summit	75	
10	Lower	10		43	Dn Main	3		76			
11	Base Tunnel			44				77	Upper	1	West
12	Base Tunnel			45	Lower	1	West	78	Upper	2	West
13				46	Lower	2	West	79	Upper	3	West
14				47	Lower	3	West	80	Upper	80	
15	Lower	1	East	48	Lower	4	West	81	Short	Line	1
16	Lower	2	East	49	Lower	5	West	82	Short	Line	2
17	Lower	3	East	50	Upper	50		83	Spiral	1	
18	Lower	4	East	51	Summit	1		84	Spiral	2	
19	Lower	5	East	52	Summit	2		85	Summit	85	
20	Lower	20		53	Summit	3		86	Spiral	3	
21	Upper	1		54	Summit	yard		87			
22	Upper	2		55	Summit	3B		88			
23	Upper	3		56				89			
24	Upper	yard		57				90	Lower	90	
25	Upper	2B		58				91	Midland	1	
26				59				92	Midland	2	
27				60	Upper	60		93	Midland	Line	1
28				61	Summit	1	West	94	Midland	Line	2
29				62	Summit	2	West	95	Midland	95	
30	Lower	30		63	Summit	3	West	96	Midland	96	
31	Up Main	1		64	Summit	1	South	97			
32	Up Main	2		65	Summit	2	South	98	Midland	Line	3
33	Up Main	3		66	Summit	3	South	99	Midland	Line	4