TPL – Introduction

My aim was to see if I could create an automated layout with lots going on that didn't just run around in circles. Having looked at JMRI (briefly I must say) and DCC++ I began to wonder whether I could actually make a simpler automation system and run it entirely on the Arduino used for DCC++.

Some of the automation techniques I read about using python scripts in JRMI made my blood run cold… there's a lot I could say here but won't without a pint or two.

It seemed to me that basing an automation on "signals controlling trains" leaves a lot of complex technical problems to be solved… and wanting to be cheap, I didn't want to invest in a range of block occupation detectors or ABC braking modules which are all very well on circular layouts but not good at complex crossings or single line operations with passing places. Also I didn't want the automation to be an obvious cycle of movements… some random timings and decisions need to be introduced so that two trains don't always arrive at the same place in the same order, nor go on the same journey in a predictable cycle.

By reversing the usual assumptions, and slaughtering a few sacred cows, I think I have a workable, extendable and cheap solution. DCC-EX, which grew out of the early TPL system, provides a clean and efficient API for the TPL automation to call.

A small amount of code (already in CommandStation-EX v4) , sits between the layout owner and DCC so that the layout owner can write automation scripts in a form that is much more user friendly. In fact the automation is written in the Arduino IDE (or PlatformIO) as per a normal Arduino script but all the C++ boilerplate code is stripped away where you don't need to see or understand it. This means that you already have all the tools you will need and there is nothing else to download or install.

[NOTE: For memory/performance worriers… The TPL code is surprisingly small and requires very little PROGMEM or RAM to execute. It is only included in the compilation of the CommandStation code if the compiler detects a "myTPL.h" file. During execution, a TPL automation is much (perhaps 2 orders of magnitude) more time efficient than the code required to process incoming requests from an external automation processor.]

When TPL is enabled, there are a few command differences in comparison to a basic CommandStation build.

> The automation takes place entirely within the Command Station. Thus TPL can only make sense of routes if the turnouts, sensors, signals pins etc are already defined. The user must include these in the myTPL.h file (for details see later) which is used to provide a definitive list of these items, effectively describing your layout and how the command station can manipulate it electronically.
>
> Having these items defined in one, easily editable, file and being able to upload them to the

Command station in a single click with existing tools makes this a simple and elegant solution. For convenience you may choose to keep the layout setup in a separate file from your automation and simply #include it in the myTPL.h file

It is assumed that JMRI, if used at all, will only be used for programming or manual throttle operations, where TPL is not controlling the loco or is paused.

Turnouts <T command>

The JMRI idea of defining turnouts externally and loading some of them to EEPROM and handling others through a mixture of DCC accessory calls or Output pins, without the CommandStation even knowing that it is a turnout, is unsuitable.

The DCC++ commands for defining or deleting turnouts will be trapped and ignored.

Turnouts already known to TPL may still be manipulated by the <T id 0/1> command but TPL refers to turnout state as LEFT or RIGHT to avoid confusion. Turnout definition can choose whether an “activate” request from JMRI means LEFT or RIGHT on an individual basis.

Sensors

Sensors defined in TPL are not polled continuously. JMRI will not be informed if they change.
Commands to define, delete or query sensors will be rejected.

Throttles

- Throttle operations through JMRI, Withrottle (Engine Driver etc) will be limited to locos that are not taking part in, or have been released from, an automation (such as manual shunting) or when TPL automation is paused (for placement of locos before being sent on an automated journey)
- Additional TPL diagnostic and control commands become available. See later.

-

Sensors

- T{PL allows for sensors that are LOW-on or HIGH-on, this is particularly important for IR sensors that have been converted to detect by broken beam, rather than reflection.

- Magnetic/Hall sensors are not particularly useful as they cant be used to detect the non-loco end of a train approaching a buffer or clearing a crossing, but are still supported.

- Handling sensors in the automation is made easy because TPL throws away the concept of interrupts (“oh… sensor 5 has been detected… which loco was that and what the hell do I do now?”) and instead has the route scripts work on the basis of “do nothing, maintain speed until sensor 5 triggers and then carry on in the script”

- TPL supports the <1 JOIN> feature of CommandStation-EX. This allows a script to automatically detect the address of a loco on the programming track, then drive it onto the main track to join in the fun.

Introduction to the Automation process

The first thing to understand is the concept of a route… this is the steps required to get from A to B (but can also be the steps to handle an animation such as a level crossing, sludge farm or fairground)

These steps may be something like:

- Wait between 10 and 20 seconds for the guard to stop chatting up the girl in the ticket office.
- Move forward at speed 30
- When I get to sensor B stop.

Similarly, the route from B to A could be something like this

- Wait 15 seconds for the tea trolley to be restocked
- Move backwards at speed 20
- When I get to A stop.

Notice that the sensors at A and B are near the ends of the track (allowing for braking distance but don't care about train length or whether the engine is at the front or back.)
Fort the time being, we'll assume they are IR reflection sensors and will go LOW when a train is detected.
[You could have converted them to IR beam-breaking sensors which would go HIGH when the beam is broken. Other options see reference.]

So your Arduino script looks like this in the Arduino IDE: in file **myTPL.h**
The presence of this file is what makes this compilation TPL and not just CommandStation.

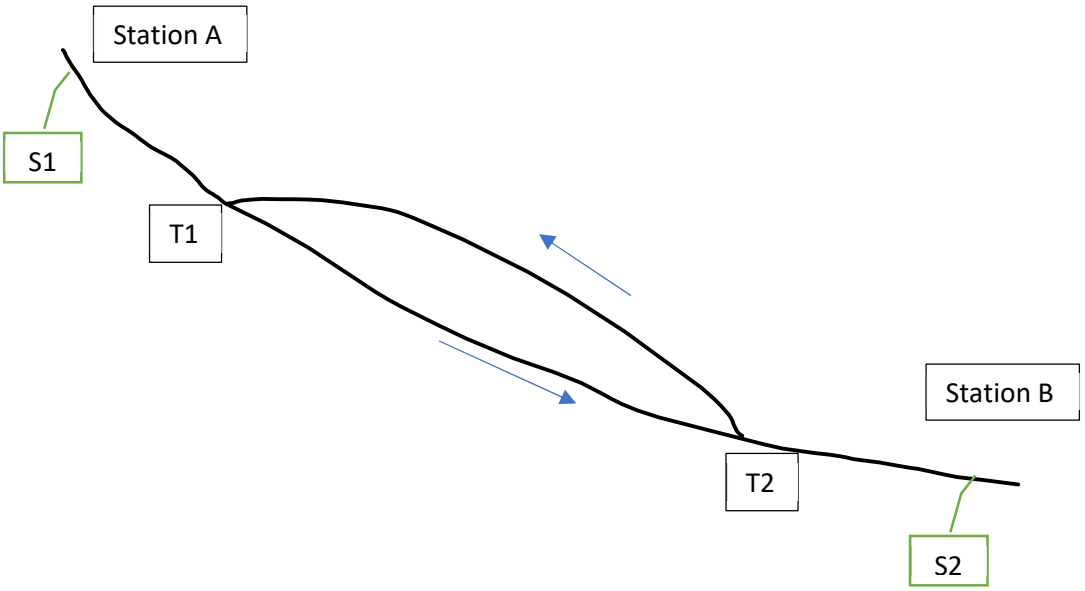
```
LAYOUT
  PIN_SENSOR(1,21,LOW) // PIN 21 goes LOW when detected
  PIN_SENSOR(2,22,LOW) // PIN 22 goes LOW when detected
  ENDLAYOUT

BEGINROUTES
  SETLOCO(3)

ROUTE(1)
  DELAYRANDOM(100,200)  // random wait between 10 and 20 seconds
  FWD(30)
  AT(2)          // sensor 2 is at the far end of platform B
  STOP
  DELAY(150)
  REV(20)
  AT(1)
  STOP
  FOLLOW(1)    // follows Route 1 again… forever

ENDROUTES
```

When the CommandStation is powered up or reset, the process starts at BEGINROUTES and in this case sets the loco address to 3 and drops through to Route(1) . If there are going to be multiple locos, it's a bit different as we will see.

Notice that the route instructions are followed in sequence by loco 3, the AT command just leaves the loco running until that sensor is detected. Although the above is trivial, the routes are designed to be independent of the loco address so that we can have several locos following the same route at the same time (not in the end to end example above!) perhaps passing each other or crossing over with trains on other routes.

The example above assumes that loco 3 is sitting at A and pointing in the right direction. A bit later I will show how to script an automatic process to take whatever loco is placed on the programming track and send it on it's way to join in the fun.

OK, that was too easy, what about routes that cross (passing places etc) … lets add a passing place between A and B. S= sensors, T=Turnout number. So now our route looks like this:



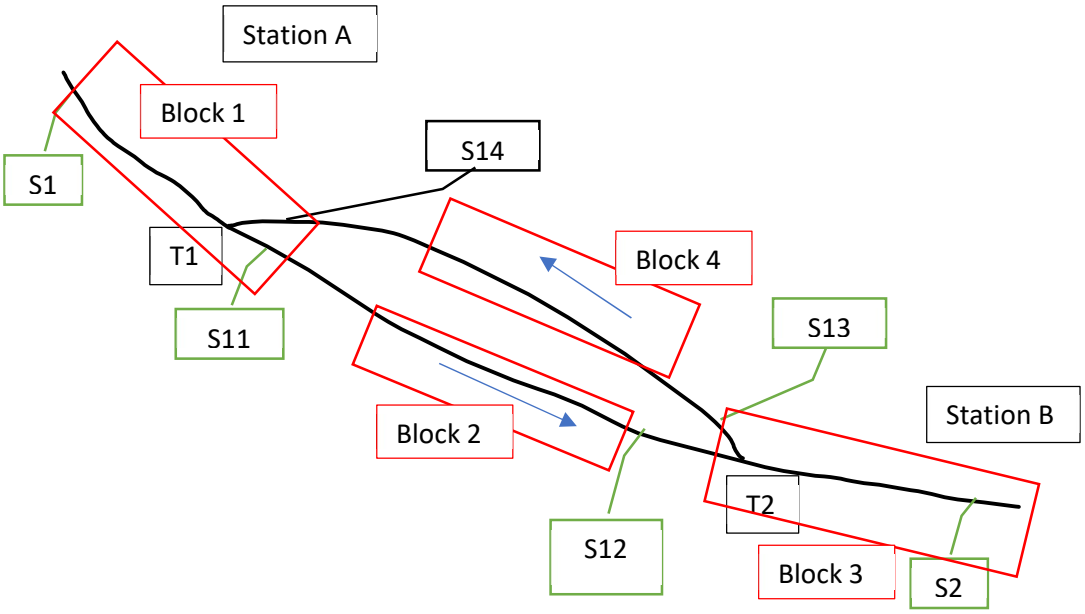

```
LAYOUT
  PIN_SENSOR(1,21,LOW) // PIN 21 goes LOW when detected
  PIN_SENSOR(2,22,LOW) // PIN 22 goes LOW when detected
  I2C_TURNOUT(1,1,150,190)  // see reference for meanings here
  I2C_TURNOUT(2,2,150,190)
  ENDLAYOUT

BEGINROUTES
    SETLOCO(3)
     ROUTE(1)
         DELAYRANDOM(100,200)  // random wait between 10 and 20
seconds
         TR(1)
         TL(2)
         FWD(30)
         AT(2)          // sensor 2 is at the far end of platform B
         STOP
         DELAY(150)
        TR(2)
        TL(1)
        REV(20)
         AT(1)
         STOP
         FOLLOW(1)     // follows Route 1 again… forever
     ENDROUTES
```

All well and good for 1 loco, but with 2 (or even 3) on this track we need some rules. The principle behind this is

- To enter a section of track that may be shared, you must RESERVE it. If you cant reserve it because another loco already has, then you will be stopped and the script will wait until such time as you can reserve it. When you leave a shared section you must free it.

- Each “section” is merely a logical concept, there are no electronic section breaks in the track.

So we will need some extra sensors (hardware required) and some logical blocks (all in the mind!):



We can use this map to plan routes, when we do so, it will be easier to imagine 4 separate routes, each passing from one block to the next. Then we can chain them together but also start from any block.

So… lets take a look at the routes now. For convenience I have used route numbers that help remind us what the route is for… any number up to 255 is Ok. Anyone want more than that and I will fix it.

```
BEGINROUTES
   … see later for startup
ROUTE(12) // From block 1 to block 2
  DELAYRANDOM(100,200)  // random wait between 10 and 20 seconds
  RESERVE(2)     // we wish to enter block 2… so wait for it
  TR(1)          // Now we "own" the block, set the turnout
  FWD(30)        // and proceed forward
  AFTER(11)      // Once we have reached AND passed sensor 11
  FREE(1)        // we no longer occupy block 1
  AT(12)         // When we get to sensor 12
  FOLLOW(23)     // follow route from block 2 to block 3

ROUTE(23)            // Travel from block 2 to block 3
  RESERVE(3)     // will STOP if block 3 occupied
  TL(2)          // Now we have the block, we can set turnouts
  FWD(20)        // we may or may not have stopped at the RESERVE
  AT(2)          // sensor 2 is at the far end of platform B
  STOP
  FREE(2)
  DELAY(150)
  FOLLOW(34)

ROUTE(34)  // you get the idea
  RERSERVE(4)
  TR(2)
  REV(20)
  AFTER(13)
  FREE(3)
  AT(14)
  FOLLOW(41)

ROUTE(41)
  RESERVE(1)
  TL(1)
  REV(20)
  AT(1)
  STOP
  FREE(4)
  FOLLOW(12)     // follows Route 1 again… forever

ENDROUTES
```

Does that look long? Worried about memory on your Arduino…. Well the script above takes just 70 BYTES of program memory and no dynamic.

If you follow this carefully, it allows for up to 3 trains at a time because one of them will always have somewhere to go. Notice that there is common theme to this…

- RESERVE where you want to go, if you are moving and the reserve fails, your loco will STOP and the reserve waits for the block to become available. (these waits and the manual WAITS do not block the Arduino process… DCC and the other locos continue to follow their routes)

- Set the points to enter the reserved area.. do this ASAP as you may be still moving towards them. (TPL knows if this is a panic and switches the points at full speed, if you are not moving then the switch is a more realistic sweep motion(feature not yet))
- Set any signals (see later)
- Move into the reserved area
- Reset your signal (see later)
- Free off your previous reserve as soon as you have fully left the block

Starting the system

Starting the system is tricky because we need to place the trains in a suitable position and set them off. We need to have a starting position for each loco and reserve the block(s) it needs to keep other trains from crashing into it.

For a known set of locos, the easy way is to define the startup process at the beginning of ROUTES , e.g. for two engines, one at each station

```
// ensure all blocks are reserved as if the locos had arrived there
RESERVE(1)  // start with a loco in block 1
RESERVE(3)  // and another in block 3
SENDLOCO(3,12) // send Loco DCC addr 3 on to route 12
SENDLOCO(17,34) // send loco DCC addr 17 to route 34
ENDPROG   // don't drop through to the first route
```

CAUTION: this isn't ready to handle locos randomly placed on the layout after a power down.

Some interesting points about the startup… You don't need to set turnouts because each route is setting them as required. Signals default to RED on powerup and get turned green when a route decides.

Startup can also SCHEDULE a "route" that is merely a decorative automation such as flashing lights or moving doors but has no loco attached to it. For example, using a signal connection to flash a red light on the pin for signal 7, green will turn it off!

```
ROUTE(66)
  RED(7)
  DELAY(15)
  GREEN(7)
  DELAY(15)
  FOLLOW(66)
ENDROUTES
```

Fancy Startup

TPL can switch a track section between programming and mainline automatically.

Here for example is a startup route that has no predefined locos but allows locos to be added at station 1 while the system is in motion. Let's assume that the track section at Station1 is isolated and connected to the programming track power supply. Also that we have a "launch" button connected where sensor 17 would be and an optional signal (ie 2 leds) on the control panel connected where signal 17 would be (see Signals below).

```
BEGINROUTES
     PROG_TRACK(0)  // start as program track connected to mainline

ROUTE(99)
     AFTER(17)     // user presses and releases launch button
     RESERVE(1)  // Wait until block free and keep others out
      PROG_TRACK(1) // power on the programming track
     GREEN(17)    // Show a green light to user
     // user places loco on track and presses "launch" again
     AFTER(17)
      READ_LOCO // identify the loco
      RED(17)   // show red light to user
      PROG_TRACK(0) // connect prog track to main
      SCHEDULE(12) // send loco off along route 12
      FOLLOW(99)   // keep doing this for another launch
```

The READ_LOCO reads the loco address and the current route takes on that loco. By altering the script slightly and adding another sensor, it's possible to detect which way the loco sets off and switch the code logic to send it in the correct direction. (easily done with diesels!)

Signals

Signals are now simply a decoration to be switched by the route process… they don't control anything.

GREEN(5) would turn signal 5 green and RED(5) would turn it red.

Sounds

You can use FON(n) and FOFF(n) to switch loco functions… eg sound horn

Numbers:

All route, sensor, output, turnout or signal ids are limited to 0- 255 (A UNO does not have enough RAM so the compiler limits this to 0-63 on a UNO device)

The same id may be used for a route, turnout, sensor, output or signal without confusing the software (the same may not be true of the user!).

Its OK to use sensor ids that have no physical item in the layout. These can only be set, tested or reset in the scripts. If a sensor is set on by the script, it can only be set off by the script… so AT(5) SET(5) for example effectively latches the sensor 5 on when detected once.

You can give names to routes turnouts signals and sensors etc using #define or "const byte " statements.

Future plans

- Some of the constructs above are not yet in the code, or need cleaning up a bit. Its early days but world situation suggests I will have plenty of time on my hands.
- I want to add some more commands for controlling animations, such as SERVO, STEPPER and LED

COMMAND REFERENCE

There are some diagnostic and control commands added to the <tag> language normally used to control the command station over USB, Wifi or Ethermet:

<D TPL ON|OFF>	Turns on/off diagnostic traces for TPL events
<S …> <Q …> <Z …> <E> <e>	These JMRI related commands are rejected as they are incompatible with TPL
<t …>	Throttle commands are only accepted for locos that are not currently being controlled by TPL (This not yet implemented)
</ PAUSE>	Pauses automation, all locos ESTOP.
</ RESUME>	Resumes automation, Locos are restarted at speed when paused.
</ STATUS>	Displays TPL running thread information
</ SCHEDULE [loco] route>	Starts a new thread to send loco onto route. or Start a non-loco animation route)
</ RESERVE id>	Manually reserves a virtual track block.
</ FREE id>	Manually frees a virtual track block
</ TL Id>	Set turnout LEFT
</ TR id >	Set turnout RIGHT
</ SET id>	Lock sensor
</ RESET id>	Unlock sensor

LAYOUT REFERENCE

LAYOUT	Identifies start of LAYOUT section. Only one layout section is permitted.
--- Turnouts ---	Each turnout must have a unique id (0-255) which is used in TL and TR commands.
SERVO_TURNOUT(id,pin,left,right)	Pin= pin number on I2C xxxx board. (0-64, over 4 chained boards) Left=servo PWM value for turnout LEFT Right=servo PWM value for turnout RIGHT
DCC_TURNOUT(id,addr,subaddr,leftActive)	Addr=DCC accessory address Subaddr= DCC accessory subaddress leftIsActive (true/false) set true if TL command should “activate” turnout.
PIN_TURNOUT(id,pin,leftValue)	Pin= Arduino CPU pin to drive turnout leftvalue= (HIGH/LOW) TL sets pin to this.
I2CPIN_TURNOUT(id,pin,leftValue)	Pin= pin n umber on I2C xxx board (0-64, over 4 chained boards) leftvalue= (HIGH/LOW) TL sets pin to this.
--- Sensors ---	Each sensor must have a unique id (0-255) which is used in AT or AFTER commands.
I2C_SENSOR(id,pin,activeWhen)	Pin= pin n umber on I2C xxx board (0-64, over 4 chained boards) activeWhen= (HIGH/LOW) value indicating sensor triggered.
PIN_SENSOR(id,pin,activeWhen)	Pin= Arduino CPU pin number activeWhen= (HIGH/LOW) value indicating sensor triggered.
--- Outputs ---	Each output must have a unique id (0-255) which is used in AT or AFTER commands.
I2CO_UTPUT(id,pin,activeValue)	Pin= pin n umber on I2C xxx board (0-64, over 4 chained boards) activeValue = (HIGH.LOW) value used for SET command
PINOUTPUT(id,pin,activeWhen)	Pin=Arduino CPU pin n umber activeValue = (HIGH.LOW) value used for SET command
ENDLAYOUT	

Routes and animations.

The TPL system operates on a number of concurrent “threads”. Each thread is following a route through the system and usually has an associated loco that it is driving. Some threads may be driving animations and have no loco attached. The thread keeps track of the position withing the route and the loco speed. A thread may be delayed deliberately or when waiting for a sensor or block section, this does not affect other threads.

At system startup, a single thread is created to follow the first entry in the routes table, with no loco.
.

ROUTES	Start of routes table.
ROUTE(routeid)	Start if a route routeid=0-255
AFTER(sensorid)	Waits until sensor reached, then waits until sensor no longer active for 0.5 seconds
AT(sensorid)	Waits until sensor reached
DELAY(duration)	Waits for duration/10 seconds
DELAYRANDOM(minduration.maxduration)	Waits a random time between minDuration/10 and maxDuration/10 seconds.
ENDIF	Marks end of IF block (see IF command)
FOFF(func)	Switches loco function off
FON(func)	Switches loco function on
FOLLOW(routeid)	Continue at ROUTE(routeid) command
FREE(blockid)	Frees a previously reserved block. See RESERVE(blockid)
FWD(speed)	Drive loco at given speed (0-127) forwards (0=stop, 1=ESTOP)
GREEN(signalId)	Sets signal to green
IF(sensorId)	Checks if sensor is activated, if NOT then processing skips to the matching ENDIF command (allowing for nested IF/IFNOTs)
IFNOT(sensorId)	Checks if sensor is activated, if it is active then processing skips to the matching ENDIF command (allowing for nested IF/IFNOT/IFRANDOMs)
IFRANDOM(percent)	Randomly decides whether to continue or skip to the matching ENDIF
INVERT_DIRECTION	Causes current loco FWD and REV commands to be reversed (e.g. used if loco is pointing in wrong direction)
PAUSE	Sets TPL into paused mode, all animations and locos are stopped and manual control is possible
PROGTRACK_JOIN	See DCCEX cmd <1 JOIN>
PROGTRACK_OFF	See DCC cmd <0 PROG> (Disconnects a JOIN)
READ_LOCO	Reads loco id from prog track and assigns it to current route
RED(signalId)	Sets Signal to RED
RESERVE(blockId)	Blockid=(0-255) If block is already reserved by another train, this loco will STOP and wait for the block to become free. block is marked as reserved and this train continues.. When you leave a block that you have reserved, you must FREE it.
RESET(sensorId)	Clears a sensor flag (see SET)

RESUME	Resumes TPL from PAUSE mode. Locos stopped by PAUSE are restarted.
REV(speed)	Move loco in reverse (see FWD)
SCHEDULE(routeid)	Starts a new thread at ROUTE(routeid) and transfers current loco to it.
SETLOCO(locoid)	Sets the loco id of the current thread.
SET(sensorId)	Locks on the software part of a sensor. If a sensor is tested by AT/AFTER/IF etc and the software part is locked on, then the sensor is seen as active without a hardware check. NOTE: This can be used for debounce. It can also be used for virtual sensors that ONLY exist in software and have no hardware equivalent. Can be used for example to pass information from a travelling train thread to a lineside animation thread.
SPEED(speed)	Changes loco speed in current direction.
STOP	=SPEED(0)
ESTOP	=SPEED(1) DCC emergency stop
TL(turnoutId)	Sets turnout LEFT
TR(turnoutId)	Sets turnout RIGHT
ENDROUTE	Terminates a route thread
ENDROUTES	End of ROUTES table, must be last entry.