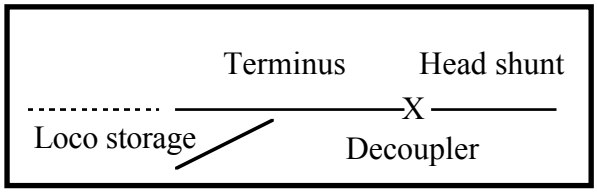
One of the reasons I designed the Quad Throttle Unit (QTU) was that controlling a layout over an RS232 link is sometimes too slow. If you want a train to stop over a standard N-gauge decoupler then precision is important. Having the computer poll a sensor a few times, then send a command to stop the train simply isn't fast enough, even if you are sending ten or twenty messages per second.

My real objective for this project is to bring a train into a terminus, uncouple the loco, bring up a fresh loco to the rear, couple up and send the train back down the line with a new loco.



However, I currently test new ideas on my test layout (see to right). This has lots of very short sections which are connected (along with sensors, signals and turnouts) to molex-style terminals. I make cables to plug into these terminals to wire up the layout into whatever configuration suits a particular experiment.

As my test layout only has one siding that could be used as a terminus, and departing trains cannot turn around before returning. I therefore decided to make life more interesting, so this project is defined as:

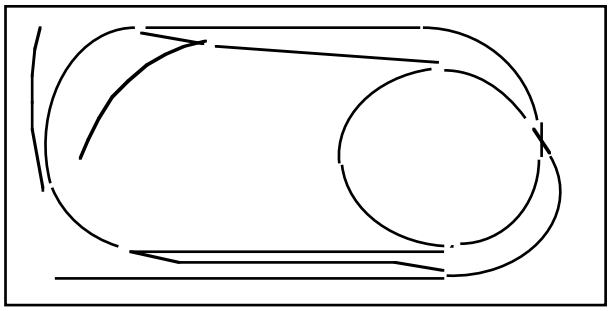
A train is progressing around a loop.
After some period of looping the train slows and stops at the 'terminus' (a decoupler in the loop).
The current loco uncouples and pulls away from the train.
A fresh loco arrives from its shed and couples up.
The old loco returns to its shed, and the new train traverses the loop in the opposite direction to before.
After a while the train once again slows down and stops again to switch locos and direction.
Repeat forever.
If uncoupling or coupling is unsuccessful repeat attempts are made until it works.

The train uses standard N gauge couplings and the decoupler is the standard Peco decoupler which is a bar mounted between the tracks which lifts to activate.
Yes, I an aware that alternative couplings are available, but wish to investigate just how reliable (or unreliable) the standard N-gauge couplings actually are. I have connected the decoupler to a slow-motion switch machine (Peco expect it to be driven by a solenoid motor, but I could envisage carriages being thrown into the air if the train stopped in the wrong place). If other couplings are used then an extra stage might be required in the script to release the tension in the first coupling.

What is a QTU?

Four independent throttles (with inertia and feedback) on a single card.
Can be used as manual throttles, autonomous throttles, or computer controlled throttles.
Built-in adjustable track circuiting.
Foldback current limiting (shutoff output until short is removed)
Motor speed detection feedback can maintain constant train speed and can keep track of where a loco is by dead-reckoning.
Integral reversing relays and spare power routing relays.
Integral turnout drive circuits
General purpose input and output lines with logic function block to define what each signal does.
Fully configurable using a script.
Operates stand-alone, or compatible with RS232, RS485 and MERG RPC interface standards up to 38,400 baud.
For further details see reference 1.

My test layout:



To stop the train with the loco-to-first-carriage coupling over the decoupler we must detect the gap between the loco and first carriage.

As normal under-baseboard reflective sensors such as IRDOT or Hector can detect the couplings themselves sometimes, these sensors are unsuitable. Instead I chose a light-beam that crosses above the coupling height, and below the loco or carriage roof-height.

Clearly we must first wait for the loco itself to arrive at the decoupler - it will then break the light beam. Then we wait for the beam to detect the gap between loco and carriage. It is at this moment we must stop the train. Note that this script assumes no separate tender is involved. If your loco has a tender then either the sensor must be arranged to

not 'see' the loco to tender join, or the script must count to the second 'gap'.

This sort of sequence of operations can be nicely represented by the following stages:

Train is progressing around the loop.
Train is slowing and approaching the decoupler.
Loco is passing the decoupler.
Stop at decoupler.
Decoupler activates.
Old loco pulls forward.
New loco arriving.
Old loco returns to shed
Train progressing around the loop the other way.

These are more fully described (without error handling conditions) as:

stage 1: Running around the loop.
Apply full speed to the track (in the current direction).
At a suitable time change to stage 2.
The suitable time could be a button press, a time delay, a loop count, or a random probability.

stage 2: Approaching the decoupler.
Apply slow speed to the track (with inertia to smooth the transition from fast to slow)
When the decoupler sensor is obscured, change to stage 3.

stage 3: Loco is passing the decoupler
Keep slow speed on the track.
When the decoupler sensor is clear (loco to first carriage coupling is over decoupler), stop the train quickly.
Change to stage 4.

stage 4: Stopped at decoupler.
Wait for a short while, and change to stage 5.

stage 5: Activating Decoupler.
Activate the decoupler (after the train has settled), then wait for a short while and change to stage 6.

stage 6: Old loco pulls forward.
Apply power for a short time to move the old loco away from the train.
Change to stage 7

stage 7: New loco arriving.
Apply slow speed to the fresh loco to bring it to rear of train.
When loco joins train, stop train and goto stage 8.

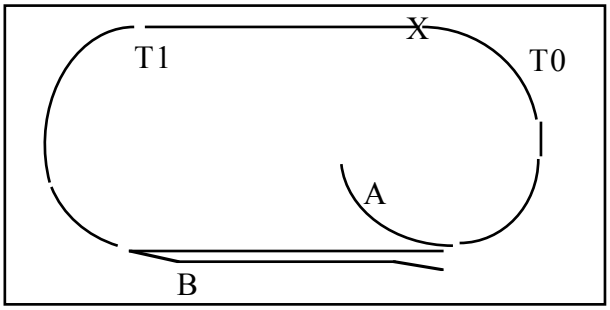
stage 8: Test new coupling
New loco moves forward to clear sensor. Change to stage 9

stage 9: Old loco returns to shed
Apply full speed to old loco, after a short time change to stage 10.

stage 10: New train moves away
Apply slow speed (in opposite direction) to new loco. Both trains are moving in opposite directions now.
When old loco has reached its shed, clear the points and set train to full speed (with inertia).
Change to stage 1, but with power reversed.

This sequence is rather complex to implement with dedicated gates, or relays, especially when you consider that we have to add tests that uncoupling and re-coupling actually worked, and retry if required. But for a scriptable controller this is much simpler.

For this project I chose a single loop of track from my test layout:



and wired it as two halves, Track0 and track1 (labelled T0 & T1 in the drawing above). X marks the decoupler and light beam sensor. The decoupler is located at a join between the two tracks.
A & B are used as loco sidings, and each has an LDR sensor to allow the loco to be stopped fully inside the siding. The turnouts to A & B are controlled, as is power to the sidings.

Lets define the hardware wiring from the QTU to the layout in terms of names used in the script:

Tracks.

The two track sections are each fed from a throttle output.

track0 - to the clockwise side of the decoupler.
track1 - the counterclockwise side of the decoupler.

Sensor inputs:

decouplerLocation - the light beam over the decoupler.
locoSiding0 - occupancy sensor in siding A entered in clockwise direction (on track 0).
locoSiding1 - occupancy sensor in siding B entered in counterclockwise direction (on track 1).

Control outputs

sidingSwitch0 - when set, the turnout to siding 0 is set to reverse.
sidingSwitch1 - ditto, for siding 1.

Relays

activateDecoupler - lifts the decoupler beam
sidingFeed0 - connects siding 0 power to track 0
sidingFeed1 - connects siding 1 power to track 1.

Control knobs (potentiometers providing control voltages to analogue input lines)

fastSpeed0 - Sets full speed for train 0
fastSpeed1 - Sets full speed for train 1
slowSpeed0 - Sets slow speed for train 0
slowSpeed1 - Sets slow speed for train 1
stopChance - defines the probability of stopping each time the train loops.
inertia - sets the throttle inertia for both trains.

As locos can vary in performance significantly, and I have spare control potentiometers, I have allocated a full speed and a slow speed control to each loco.

In addition to the external connections, the Q-script needs a items of data to keep track of what it is doing:

One bit control variables used in the script:

direction - 0 for clockwise, 1 for counterclockwise.
reverseArrival - reverse power is required on the track approaching the decoupler.
reverseDeparture - reverse power is required on the track leaving the decoupler.

8 bit control variables

stage - hold the stage number to indicate what the program is currently doing.
arrivalIndex, departureIndex - see notes.

QTU scripting basics

The QTU is programmed with compiled code in flash memory. This code gives the QTU its basic functions of communicating with a computer, handling RPC stack interfaces, driving throttles, inertia and so on.
When the code is not doing any of these basic functions, it interprets a ‘Q-script’ that is stored in EEPROM memory. This Q-script can easily be edited on a computer and downloaded to the QTU. Once downloaded the computer is not required unless you wish to control the QTU from the computer, debug the Q-script, or monitor the layout from the computer.

The Q-script is written in a high-level language, which is compiled (using Tcc, see reference 2) to a simple assembler language, and assembled into a byte code which is downloaded to the QTU board. Each QTU on a layout can have a different Q-script. The byte code can also be disassembled using Tcc so you can examine what is programmed into a QTU. The disassembled version is similar to, but not identical to the original high level language.

The Q-script is executed repeatedly, as fast as spare processing power permits. Typically several tens of thousands of byte code instructions are executed every second. In fact you can see how fast your script is running because the green LED flashes every 256 complete executions of the whole Q-script. The Q-script for this automatic uncoupling project flashes the light twice every second.

For more information on QTU scripting see reference 3 and appendix 1.

Notes:

I could have used the other two throttle outputs to feed the two sidings, and thus avoid using two of the relays, but that seemed a waste.

The decoupler is driven from a slow motion switch machine that takes around 500 ma when operating, and so I chose to drive it from a relay which reversed the power. The switch machine cuts off its own power at the end-stops. My test layout did not have space for a tortoise, and so I could not use the tortoise outputs that are part of each QTU.

The sensor over the decoupler is an IR LED and an IR transistor, mounted above coupling height. A shroud was fitted around the sensing transistor to avoid reflected light - otherwise the train would not stop at the correct position.

Introduction to the script for this project

Apart from the 'stage' variable holding the number of the stage we are in, we have the 'direction' flag. This is zero when the train is running clockwise, and one for counterclockwise. In essence this is used to reverse the throttle outputs.

The loop of track is split into two tracks, connected to the two throttle outputs, but most of the script considers the two track segments to be 'arrival' and 'departure'. In other words the track on which a train arrives at the decoupler from, or the track it departs on. This is because there is only one set of code that handles the loco change operation regardless of which way the train is running. Two index registers ('arrivalIndex' and 'departureIndex') are set to 0 & 1 or 1 & 0 dependant upon which direction the train is running:

```
 if (direction)
{                    // running counterclockwise
```

```
    arrivalIndex = 0;              // from throttle 0
    departureIndex = 1;            // to throttle1
}
else
{                                  // running clockwise
    arrivalIndex = 1;              // from throttle 1
    departureIndex = 0;            // to throttle 0
}
```

These are used to index arrays. We have two throttles, two current detectors, two full speed settings two turnouts, two siding feed relays, two siding feed sensors and so on. For all of these we access them using the index registers, so for example to set the siding turnout on the departure leg (where the old loco will go after it has uncoupled) to reverse, we use:

```
sidingSwitch [departureIndex] = !zero;
```

This means that we only need one set of code for running either clockwise, or counterclockwise.

Note that the // (two slash characters) indicates that the rest of the line is a comment - it is completely ignored by the compiler.

One of the one-bit-accumulators is reserved in this script to hold the value zero, and here we are storing a one (to set the turnout to reverse). !zero means the inverse of zero, which of course is a one.

In addition to stage, arrivalIndex and departureIndex we have 'reverseArrival' and 'reverseDeparture' flags. These allow selective reversal of each track. For example if uncoupling fails the old loco must backup again, so 'reverseDeparture' is set. Also when the old loco is returning to its shed, and the new loco is pulling away the two locos are moving in opposite directions simultaneously.

These are used to set the throttle direction controls:

```
reverse [arrivalIndex] = direction ^ reverseArrival;
reverse [departureIndex] = direction ^ reverseDeparture;
```

The value ‘reverse’ is the direction control on a throttle. If set to one, that throttle output is reversed. ‘reverse [arrivalIndex]’ is effectively the direction switch on the throttle which is connected to whichever track approaches the decoupler (depending upon the current direction of travel).

The '^' means exclusive-or. So the throttle currently used for the arrival track is set to reverse if either direction is set to reverse OR reverseArrival is set, but not if both are set, or neither.

Controlling train speed

We set both tracks to full speed, but some stages will modify one or the other track if required. The throttle settings are not given to the throttles themselves until the end of one pass of the script (see appendix 2 for an explanation), so setting speed to full, and then perhaps later setting it to slow does not create pulsed output, or any conflict. Setting the speed globally (outside of any particular stage handling code) merely means we do not need to set the speed in every stage.

```
// always feed power to track.
speed0 = fullSpeed [departureIndex];
speed1 = fullSpeed [departureIndex];
```

Note the use of departureIndex here. This does not refer to which track is being fed, but in fact to which loco is being controlled (one loco always pulls clockwise and the other pulls counter clockwise). By setting speed0 and speed1, we default to the whole loop having the current full speed setting.

Description of code for stage 1

In stage 1 the train is running around the loop. It is running clockwise if direction is zero, or counterclockwise if direction is set to one.

The task for stage 1 is to decide when we should reach our destination - stopping every time around the loop might look a little silly. Rather than simply counting loops this code uses a random number generator to decide when to stop. The QTU has four random probability functions, called random0 to random3.

stage 1 used the first random function (using values random0 and probability0). The probability of stopping is set by one of the analogue input lines, named stopChance by this script.

In order to begin slowing at a sensible time, we invoke the random decision when the train reaches the furthest point from the decoupler, when the loco changes from occupying the departing section, to the arriving section.

```
probability0 = stopChance;

// Train running around loop until decision made to stop
if (stage == 1)
{
   // Random(x) only set to 1 with probability(x) chance.
    Random0 = cur [arrivalIndex];
    if (Random0)                    // Time to stop
     {
       stage = 2;
     }
}
```

The statement "Random0 = cur [arrivalIndex]" copies the current detect stage of whichever throttle is driving the track on which the train will arrive at the decoupler, into the Random0 store. Thus each time the arrival section becomes occupied we have a chance of Random0 becoming set.

If Random0 does indeed become set then we move onto stage 2.

Each random function is a single bit store in memory. Writing a one to a random store has a probability of succeeding. If it succeeds then reading the store will indeed return a one. If the writing fails then the store retains its previous value. Writing a zero always succeeds.

The probability of writing a one succeeding is defined by the 8-bit variables probability0 to probability3. Only the first attempt to write a one is considered. Even if the write fails you must write a zero before another random attempt can be made (otherwise continuously writing a one is bound to succeed eventually).

Description of code for stage 2

In stage 2 we have decided that the train should stop. We set the speed to slow, and the inertia to whatever is defined on the control panel ('inertiaSetting' is the name we have given to one of the analogue inputs).

We probably only need to control the arrival track as the voltage is unlikely to have reduced much by the time the loco clears the departure track. Remember that we entered stage 2 when the arrival track detected the loco, so pickups might still be on the departure track.

We need to stop the train when the loco has passed the decoupler - when the sensor beam is clear, but that looks the same as before the loco reaches the sensor. So in this stage we look for the front of the loco breaking the light beam and move to stage 3.

In addition we reverse the turnout for this locos siding, and turn on track feed to its siding. We could do this later, but it has to be done sometime.

```
// stage 2.
// Train is stopping. Slow down until we approach
   decoupler
if (stage == 2)
{
    inertia0 = inertiaSetting; inertia1 = inertiaSetting;
    speed0 = slowSpeed [departureIndex];
      speed1 = slowSpeed [departureIndex];
    if (decouplerLocation)          // Loco breaks beam
     {
       SidingSwitch [departureIndex] = !zero;
            // Prepare to park this loco
       SidingFeed [departureIndex] = !zero;
        stage = 3;                  // prepare to stop
     }
}
```

Remember that the speed and inertia setting code is executed continuously, as long as we are in stage 2, but the code inside the "if (decouplerLocation)" test is only executed once because once it has been executed ‘stage’ then holds the value 3, so the stage 2 code doesn't run anymore.

Description of code for stage 3

In stage 3 the loco is passing the decoupler. We are looking for the tail end of the loco.

While the loco is passing the decoupler we keep the slow speed on the track. Here we must drive both tracks as the loco is traversing the inter-section gap (which is where the decoupler is).

Note we no longer apply inertia. The train has already slowed from full speed to slow speed smoothly, and we want the final stop to be a sudden, dead-stop. If we kept inertia on the train would run past the decoupler. As long as the slow speed is indeed fairly slow the dead stop doesn't look unprototypical.

When the light beam is clear (the loco has passed, and the coupling to the first carriage is over the decoupler) then we move to stage 4 (which stops the train).

So that uncoupling doesn't appear too rushed we want the train to be halted for a few moments before activating the decoupler. This delay is purely aesthetic, and not required functionally.

The memory location ms10time is a 16 bit counter which increments every 10 milliseconds. ms10time will wrap around every 256 counts, or every 2.56 seconds. This makes this counter suitable for delays less than this limit.

So we add 200 to ms10timeL and store the result in a memory location to use in stage 4. See the explanation of stage 4 for more details.

```
// stage 3.
// Loco of stopping train is over decoupler
if (stage == 3)
{
    speed0 = slowSpeed [departureIndex];
      speed1 = slowSpeed [departureIndex];
   if (!decouplerLocation)      // At loco-carriage coupling
     {
            // Stop for two seconds
       waitUntil = ms10timeL + 200;
        stage = 4;
     }
}
```

Description of code for stage 4

Stage 4 is purely a cosmetic addition so that the train is stopped for a short while before the decoupler activates.

In stage 4 we want the train to be stationary, so we activate the brake. In fact we only really need to

brake the departing section as the loco must have left the arrival section (otherwise we would not be able to bring in the new loco!).

All we do is look at the ms10time location and wait for it to have the same value as we saved into 'waitUntil' in the previous stage. Currently the compiler only supports comparing variables with constants, so we use:

"if (waitUntil-ms10time == 0)"

instead of the more natural:

"if (waitUntil == ms10time)"

When ms10time has counted 200 times (2 seconds have passed) we switch to stage 5. In stage 5 we want to wait for three seconds which means that the ms10time trick is inappropriate (it is limited to 2.55 seconds) so here we illustrate another mechanism.

The variables Period and Duration (there are four of each of these) can be used to generate a delay. Duration is given a value measured in 100ms increments. Period resets itself back to zero that time after it is set to a one.

```
// stage 4.
// Loco stopped before lifting decoupler
if (stage == 4)
{
    appplyBrake = !zero;            // stop dead
   if (waitUntil-ms10timeL == 0)    // time to move loco
     {
       Duration0 = 30;              // Stop for 3 seconds
           Period0 = !zero;         // start timer
       stage = 5;
     }
}
```

Description of code for stage 5

The train is stopped over the decoupler, the decoupler must be activated. We must wait for the decoupler to finish moving because it is driven by a slow-motion switch machine. In addition we must wait for the workmen to clear the track and signal all-clear to the driver, so we allow a full 3 seconds (as set in stage 4).

While we are in stage 5 we hold both the brake on (which has the effect of setting speed to zero), and we activate the decoupler.

The decoupler is connected to one of the output pins and so its state is set to whatever was last received from the computer, so we hold it on while in stage 5.

When another 2 seconds have passed we reverse the turnout feeding the siding holding the new loco, and connect power to it.

We need to move the old loco away from the train to avoid a risk of derailing the train when the new loco shunts it, so we reset the delay function for another second and switch to stage 6.

```
// stage 5.
// Train stopped, wait a moment
if (stage == 5)
{
    applyBrake = !zero;             // stop dead
    decouple = !zero;               // activate decoupler
    if (!Period0)                   // time to move loco
     {
           Period0 = zero;          // so it works next time
            // Prepare to fetch new loco
       SidingSwitch [arrivalIndex] = !zero;
        waitUntil=ms10time+250;// Forward for 2.5 sec
       stage = 6;
     }
}
```

Description of code for stage 6

The decoupler has activated and we need to take the old loco away. We do not want to park the loco in its siding in case the new loco fails to couple up to the train (we would need the old loco to push the train back into position for a repeat attempt).

While in stage 6 we apply slow speed to the old loco (which is in the departure track). No inertia is used as that would make it difficult to determine whether the the loco has actually moved away from the train. Actually the QTU has another function that can move the loco a certain distance by dead-reckoning, but that is the subject of another project.

The fresh loco is held standing in case the old loco hasn't successfully uncoupled.

Either of two things can happen here.

Either the loco failed to uncouple, and so the sensor light beam will be broken when the loco pulls the carriages forward.

Or uncoupling was successful, and the loco moves for a little without the carriages following.

If uncoupling was successful, and time passes, then we simply switch to stage 7.

If uncoupling fails, and the light beam is broken then we want to backup again to put the carriages back into position. To do this we reverse the power to the departure track and act as if the loco is just passing the decoupler in its initial approach. Actually it is the first carriage passing the sensor, but the logic is the same. So all we have to do is reverse power and let stage 3 take over.

If uncoupling never succeeds the train will jiggle back and forth forever. If this is unacceptable then a

counter of attempts might be incremented and we could give up if it fails too often, perhaps returning to stage 1 to let the train run around the loop again.

```
// stage 6.
// Time for old loco to pull away
if (stage == 6)
{
      // Move old loco clear, keep fresh loco standing
   speed0[departureIndex] = slowSpeed[departureIndex];
   brake [arrivalIndex] = !zero;
   if (waitUntil-ms10timeL == 0)
     { // Loco has moved for 1 second
       stage = 7;                    // So uncoupling is OK
     }
   if (decouplerLocation)           // did not decouple
     {
       reverseDeparture = !zero; // back up a bit
       stage = 3;
     }
}
```

In fact there is some extra code in stage 4 to restore the reverseDeparture flag that was omitted here for clarity. See the full listing for details.

Description of code for stage 7

Uncoupling was successful and the old loco has moved forward, and we need to bring it to a halt.

Bring in the new loco. When the new loco arrives the carriages will be shunted slightly and the light beam obscured.

So we apply the brake on the departure track to stop the old loco, and apply slow speed to the arrival track to bring the fresh loco out of its siding.

When the carriage that was previously the front carriage breaks the beam we can switch off power to the siding that the fresh loco just came out of, and set its turnout normal.

To prepare for travelling in the opposite direction we invert the direction flag. But to allow the old loco to return to its siding we set the reverseArrival flag. Note that now we have toggled the direction bit the old loco is now on the track designated arrival, and the train is preparing to depart on the newly designated departure track.

By setting the reverseArrival flag the old loco can carry on in its previous direction to its siding.

Note that we do not stop the new loco after it couples up to the train. Toggling the direction flag will reverse the direction of the train so that the train once again clears the sensor. As this is done with no inertia it is fast enough that to a casual glance it looks like the train is shunted a small amount by coupling the new loco.

We start a one second timer to check that coupling was successful, in stage 8.

```
// stage 7.
// Old loco uncoupled and pulled forward. Bring in new loco
if (stage == 7)
{
      // Stop old loco until new coupled
   brake [departureIndex] = !zero;
     SidingFeed [arrivalIndex] = !zero;
      // Fresh loco arrives slowly
   speed [arrivalIndex] = slowSpeed [arrivalIndex];
     inertia [arrivalIndex] = inertiaSetting;
   if (decouplerLocation)           // Carriages shunted
    {
          // Clear the line
      SidingSwitch [arrivalIndex] = zero;
      SidingFeed [arrivalIndex] = zero;
          // train goes the other way
      direction = !direction;
          // Old loco now on arrival side
     waitUntil = ms10timeL+150;       // Test coupling
       stage = 8;                     // See above for stage 6
    }
}
```

Description of code for stage 8

The new loco has shunted the train and should be pulling the carriages to clear the sensor (just a few millimetres). The old loco is sent on its way to its siding.

If coupling occurred OK, then the sensor clears and we change rapidly to stage 9 where the train will wait for another two seconds before departing.

If coupling failed then the one second time will pass and we need to reattempt coupling. For now we just give up and stop everything in stage 15.

```
// stage 8.
// New loco clear decoupler. See if loco is coupled
// Positioned before stage 7 code so that indexes are
   updated.
if (stage == 8)
{
   // Train pulls away slowly
  speed0[departureIndex] = slowSpeed[departureIndex];
     brake [arrivalIndex] = !zero;
   if (!decouplerLocation)          // coupled OK
    {
          // leave train another 2 seconds
      waitUntil = ms10time+200;
          reverseArrival = !zero;    // Old loco now arrival
       stage = 9;                    // Park old loco
    }
   if (waitUntil-ms10TimeL == 0)  // Coupling failed
    {
      reverseDeparture = !zero; // backup the old loco
          direction = !direction;      // back to original
         stage = 3;                    // try again
    }
}
```

Description of code for stage 9

The new loco is pulling away with carriages coupled successfully.
The old loco is finished with, so park it in its siding. The old loco (in the arrival track) is sent (backwards in the new travelling direction) to its shed at full speed so that it clears the track before the new train (running at slow speed) gets around to the arrival track.
As soon as the old loco has been travelling a short while (or indeed if it arrives quickly) then switch to stage 10 to move the new loco.

```
if (stage == 9)
{
   brake [departureIndex] = !zero;  // Hold new loco & train
   // Park old loco
   speed [arrivalIndex] = fullSpeed [arrivalIndex];
   inertia [arrivalIndex] = inertiaSetting;
   if (waitUntil-ms10time == 0    // Old loco has a head start
   |  opt [arrivalIndex])             // or old loco parked quickly
  {
      stage = 10;
  }
}
```

Description of code for stage 10

The old loco is on its way to the shed, the new loco has coupled successfully.
Now slowly let the new loco (and train) depart.
When the old loco reaches its sensor in the shed, set its brake on (which will slow the loco with its inertia).
When the old loco finally stops (after the inertia delay) disconnect the siding and set the points normal, revert the arrival track to normal operations.
The old loco is given a low inertia value so it will stop fairly quickly, and not run past its sensor.

```
if (stage == 10)
{
   // Park old loco
   speed [arrivalIndex] = fullSpeed [arrivalIndex];
   speed [departureIndex] = slowSpeed [departureIndex];
   inertia [arrivalIndex] = 20;          // low inertia
   if (opt [arrivalIndex])         // Reached sensor in siding
  {
      brake [arrivalIndex] = !zero;   // stop in siding
      if (CurrentSpeed [arrivalIndex] == 0)
      {   // loco has stopped
         // disconnect siding
         SidingFeed [arrivalIndex] = zero;
         SidingSwitch [arrivalIndex] = zero;
         reverseArrival = zero;
         stage = 0;                    // Resume normal running
      }
  }
}
```

Full program listing

```
// Two-way shuttle with two locos.
// Two throttles for two locos

equ manualReverse = input4;
equ manualBrake = input5;
equ reset = input6;
equ decouplerLocation = input8;          // Light beam across track
equ opt = input9;                        // 9 & 10
equ leds = (byte) output0;
equ SidingSwitchDrive [2] = output9;
equ SidingFeedDrive [2] = output24;      // relay 5, 6
equ decouple = output26;                 // relay 7
equ direction = memory0;                 // 0 = counterclockwise
equ reverseDeparture = memory1;     // Train has to back up again
equ reverseArrival = memory2;            // old loco returning to shed.
equ applyBrake = memory3;
equ cur [2] = memory16;                  // 16..23 as required
equ SidingSwitch [2] = memory18;         // 18 & 19
equ SidingFeed [2] = memory20;           // 20/21 = relay 30/31
equ stage = byteMemory0;
equ waitUntil = byteMemory1;             // Time delay function
equ zero = accumulator1;                 // always left at zero
equ fullSpeed [2] = AdcUser0;            // Global speed control
equ slowSpeed [2] = AdcUser2;
equ stopChance = AdcUser4;               // % chance of stopping
equ inertiaSetting = AdcUser5;
equ arrivalIndex = index0;               // throttle on approach stop
equ departureIndex = index1;             // throttle after stop

MaxDetect = 20;
DetectCurrent = 20;                      // Make sure no glitches
cur0 = state0.occupied;                  // may change to be an input
cur1 = state1.occupied;
if (reset)
{
   stage = 0;
   direction = zero;
   SidingSwitch0 = zero;
   SidingSwitch1 = zero;
   SidingFeed0 = zero;
   SidingFeed1 = zero;
}

// always feed power to track. Modified by brake or StopAt.
speed0 = fullSpeed [departureIndex];
speed1 = fullSpeed [departureIndex];
inertia0 = 0; inertia1 = 0;
probability0 = stopChance;
applyBrake = zero;

if (direction)
{                                        // running counterclockwise
   arrivalIndex = 0;                     // from throttle 0
   departureIndex = 1;                   // to throttle1
}
else
{                                        // running clockwise
   arrivalIndex = 1;                     // from throttle 1
   departureIndex = 0;                   // to throttle 0
}

// old loco may have to backup
reverse [departureIndex] = direction ^ reverseDeparture ^
manualReverse;
reverse [arrivalIndex] = direction ^ reverseArrival ^ manualReverse;

// stage 0.
// System just started
if (stage == 0)
{
   inertia0 = inertiaSetting; inertia1 = inertiaSetting;
   if (!cur [arrivalIndex])
   {
      stage = 1;
   }
}

// Train running around loop until decision made to stop
if (stage == 1)
{
   SidingSwitch0 = zero; // Clear the line
   SidingSwitch1 = zero;
   SidingFeed0 = zero;
   SidingFeed1 = zero;
   inertia0 = inertiaSetting; inertia1 = inertiaSetting;
   // Random(x) variables only set to 1 with Probability(x) chance.
   Random0 = cur [arrivalIndex];         // Train enters arrival section
   if (Random0)                          // Time to stop
   {
      stage = 2;
   }
}
```

```
// stage 2.
// Train is going to stop. Slow down until we approach decoupler
if (stage == 2)
{
   speed0 = slowSpeed [departureIndex];
   speed1 = slowSpeed [departureIndex];
   inertia0 = inertiaSetting; inertia1 = inertiaSetting;
   if (decouplerLocation)                    // Loco breaks beam
   {
      // Prepare to park this loco
      SidingSwitch [departureIndex] = !zero;
      SidingFeed [departureIndex] = !zero;
      stage = 3;                             // prepare to stop
   }
}

// stage 3.
// Loco of stopping train over decoupler
if (stage == 3)
{
   speed [departureIndex] = slowSpeed [departureIndex];
   if (!reverseDeparture)                    // if not re-trying
   {
      speed [arrivalIndex] = slowSpeed [departureIndex];
   }
   else
   {
      brake [arrivalIndex] = !zero;
   }
   decouple = zero;
   if (!decouplerLocation)                   // At loco-carriage coupling
   {
      waitUntil = ms10time + 200;            // Stop for two seconds
      stage = 4;
   }
}

// stage 4.
// Loco stopped before lifting decoupler
if (stage == 4)
{
   applyBrake = !zero;
   reverseDeparture = zero;          // in case we have just backed up.
   if (waitUntil-ms10time == 0)              // time to move loco
   {
      Duration0 = 30;                        // Three second delay
      Period0 = !zero;                       // Stop for 3 seconds
      stage = 5;
   }
   if (decouplerLocation)                    // did not stop in time
   {
      reverseDeparture = !zero;              // backup and try again
      stage = 3;
   }
}

// stage 5.
// Train stopped, wait a moment
if (stage == 5)
{
   applyBrake = !zero;                       // stop dead
   decouple = !zero;                         // activate decoupler
   if (!Period0)                             // time to move loco
   {
      Period0 = zero;                        // So it works next time.
      // Prepare to fetch new loco
      SidingSwitch [arrivalIndex] = !zero;
      waitUntil = ms10time+250;              // Pull forward for 2.5 sec
      stage = 6;
   }
}

// stage 6.
// Time for old loco to pull away
if (stage == 6)
{
   // Just move old loco
   speed [departureIndex] = slowSpeed [departureIndex];
   brake [arrivalIndex] = !zero;             // Keep new loco standing
   if (waitUntil-ms10time == 0)              // Loco has moved for 1 sec
   {
      stage = 7;                             // So uncoupling is OK
   }
   if (decouplerLocation)                    // did not decouple
   {
      reverseDeparture = !zero;              // back up a bit
      stage = 3;
   }
}

// stage 8.
// New loco clear decoupler. See if loco is coupled
// Positioned before stage 7 code so that indexes are updated.
if (stage == 8)
{
   // Train pulls away slowly
   speed [departureIndex] = slowSpeed [departureIndex];
   brake [arrivalIndex] = !zero;
   if (!decouplerLocation)                   // coupled OK
   {
      waitUntil = ms10time+200;              // leave train another 2 sec
      reverseArrival = !zero;       // Old loco now on arrival side
      stage = 9;                             // Park old loco
   }
   if (waitUntil-ms10time == 0)              // Coupling failed
   {   // We have moved off a safe distance
      reverseDeparture = !zero;              // backup the old loco
      direction = !direction;                // back to original train
      stage = 3;                             // try again
   }
}

// stage 7.
// Old loco uncoupled and pulled forward. Bring in new loco
if (stage == 7)
{
   SidingFeed [arrivalIndex] = !zero;
   brake [departureIndex] = !zero;    // Stop old loco until new coupled
   // Loco arrives slowly
   speed [arrivalIndex] = slowSpeed [arrivalIndex];
   inertia [arrivalIndex] = inertiaSetting;
   if (decouplerLocation)                    // Carriages shunted
   {
      SidingSwitch [arrivalIndex] = zero; // Clear the line
      SidingFeed [arrivalIndex] = zero;
      direction = !direction;                // train goes the other way
      waitUntil = ms10time+150;              // Test coupling
      stage = 8;                             // See above for stage 6
   }
}

// stage 9.
// New loco has successfully coupled up
if (stage == 9)
{
   brake [departureIndex] = !zero;           // Hold new loco & train
   speed [arrivalIndex] = fullSpeed [arrivalIndex]; // Park old loco
   inertia [arrivalIndex] = inertiaSetting;
   if (waitUntil-ms10time == 0          // Old loco has has a head start
   |  opt [arrivalIndex])                    // or old loco parked quickly
   {
      stage = 10;
   }
}

// stage 10
// Old loco is on its way to the shed.
// New loco has coupled successfully
// Train has picked up passengers
if (stage == 10)
{
   // until old loco out of the way
   speed [arrivalIndex] = fullSpeed [arrivalIndex]; // Park old loco
   speed [departureIndex] = slowSpeed [departureIndex];
   inertia [arrivalIndex] = 20;              // low inertia
   if (opt [arrivalIndex])                     // Reached sensor in siding
   {
      brake [arrivalIndex] = !zero;          // stop in siding
      if (CurrentSpeed [arrivalIndex] == 0) // loco has stopped
      {
         SidingFeed [arrivalIndex] = zero; // disconnect siding
         SidingSwitch [arrivalIndex] = zero;
         // Now synchronise the two outputs (for inertia delays)
         reverseArrival = zero;
         stage = 0;                          // Resume normal running
      }
   }
}

if (applyBrake | manualBrake)
{
   brake0 = !zero;
   brake1 = !zero;
}

leds = stage;
output4 = direction;
output5 = cur [arrivalIndex];
```

```
output6 = state0.overload | state1.overload; // either channel overload
output7 = reverseArrival;
SidingSwitchDrive0 = SidingSwitch0 ^ input0; // output9
SidingSwitchDrive1 = SidingSwitch1 ^ input1; // output10
output17 = input0;
SidingFeedDrive0 = SidingFeed0 ^ input2;
SidingFeedDrive1 = SidingFeed1 ^ input3;
end "Push-Pull service".
```

References

Ref 1. QTU information
a. Detailed description.
b. Circuit diagram

Ref 2. Tcc information

Ref 3. QTU scripting information
a. QTU scripting user guide.
[not written yet]
b. QTU Script byte codes

Appendix 1: Downloading scripts

In order to transfer a Q-script onto a QTU the following steps are necessary:

1. Run Tcc
2. Open the Network Configuration window.
3. Locate the appropriate QTU in the list.
(or add a new QTU module if required).
4. Click on the QTU to get a menu
5. Select Script from the menu
6. Open your script file, or type in the script.
7. Use the build menu to compile.
8. Fix any errors reported and compile again.
9. Start the network
10. Use the build menu to download.

After a successful compile of a Q-script that differs from the current QTU contents, the QTU will be highlighted in the Network Configuration window. This indicates that the Q-script on the board is out of date.

If the network is stopped (or you don't actually have any QTU modules connected) this highlighting will not happen, nor can the script be downloaded.

After downloading the highlighting should disappear, but note that the process of downloading takes several seconds.

Once the Q-script is running on the QTU you could click on the QTU in the network configuration window and select 'Show RAM'. This opens a window which displays all the memory accessible by your Q-script, and shows the values of each of the items used in the script.

Appendix 2: QTU scripting

As the QTU is designed to be used either on its own (manual or autonomous mode), or with a computer (monitored manual, as a safety layer, or fully computer controlled) the inputs and outputs can be driven from either the Q-script, or the computer, or even both in cooperation.

To achieve this end, the following philosophy has been adopted:

The computer is assumed to be in control, but the Q-script can override all computer initiated actions.

This is done by assuming that we are using the computer outputs, but giving the Q-script a chance to modify these outputs before they are fed to the output pins. The same happens in reverse for the inputs.

To achieve this goal, the Q-script must run continuously (and not stopping to wait for some condition), and after the Q-script has been fully executed the following happens automatically before the script is started once again:

1. Throttle settings and outputs as seen by the Q-script are copied to the throttles and output registers (ready to be output to the pins).

2. Throttle settings and outputs received over the serial link (if the computer is trying to control the layout) are made available to the Q-script.

3. Inputs and sensor values as seen by the Q-script are copied so they can be sent to the computer if required.

4. The latest filtered inputs read from the pins are made available to the Q-script.

5. All accumulators are set to zero.

This means that either or both the computer and the Q-script could control the layout. If the Q-script does nothing then the computer controls all outputs and sees all inputs. If the Q-script updates any input or output values every time the Q-script runs then its control overrides any computer control.

The key point here is that for output lines and throttle control (speed, direction, brake, inertia) the Q-script must set these values every time the Q-script runs, otherwise the last value received from the computer takes over. This is important even if the computer interface is not connected.

Other data (such as data used internally by the Q-script) is not affected.