Applications Note: Automated Two-Train Operations Using Cab Control

This Applications Note describes the automated operation of two trains running on a single shared mainline using computer-automated *"cab control*".

Operation:

Two trains run simultaneously along a shared mainline loop, which is divided into a number of electrically isolated track blocks. A separate, dedicated throttle is assigned to each train traveling on the mainline. Each throttle is automatically routed to follow its train as it moves from block to block, providing seamless, independent speed control of each engine.

The traffic conditions ahead of each train are continually monitored. A train is automatically brought to a smooth stop whenever it approaches too close to a train ahead, and automatically returns smoothly to its previous running speed once the track ahead has cleared.

Track Layout:

The track layout for this example is shown in Figure 1. Common ground wiring is employed. Four insulated track blocks, here labeled A, B, C, and D are used.

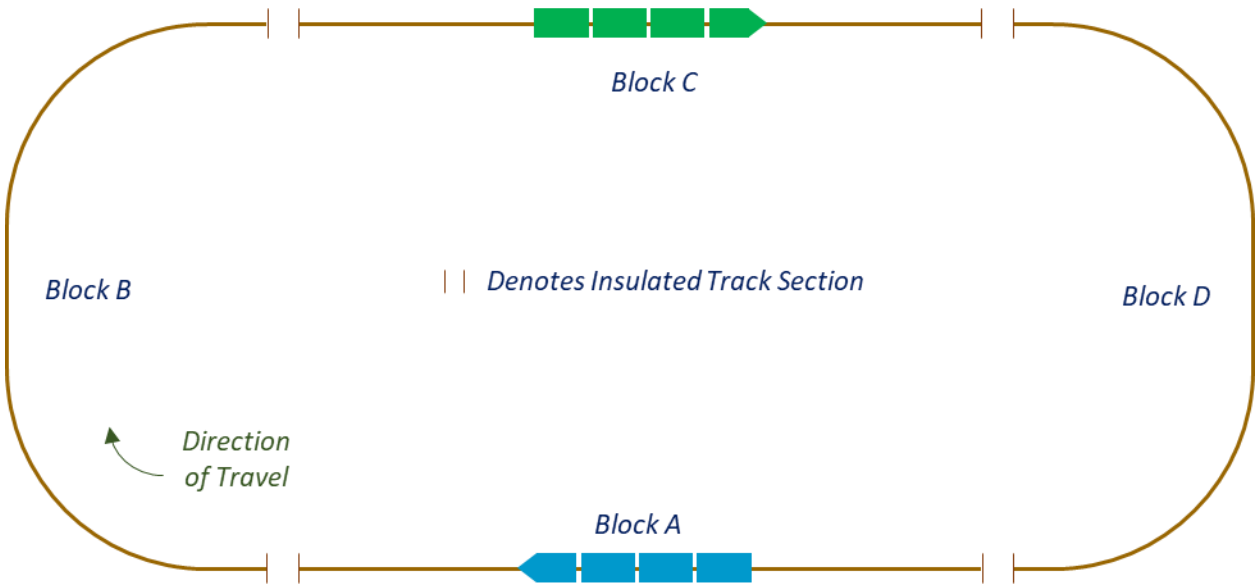

Figure 1. Track Layout

CTI Hardware:

The automated operation of the two trains requires one Train Brain and two Smart Cab modules (conventional manual throttles may also be used in lieu of the Smart Cabs).

The Train Brain's controllers are used to select the output of one of the two throttles to be routed to each track block. The Train Brain's sensor ports are used to detect the presence of a train in each block. Here, we assume the use of current detection sensors, but the code may be easily adapted to work with all sensor types. The modules are wired as shown in Figure 2.

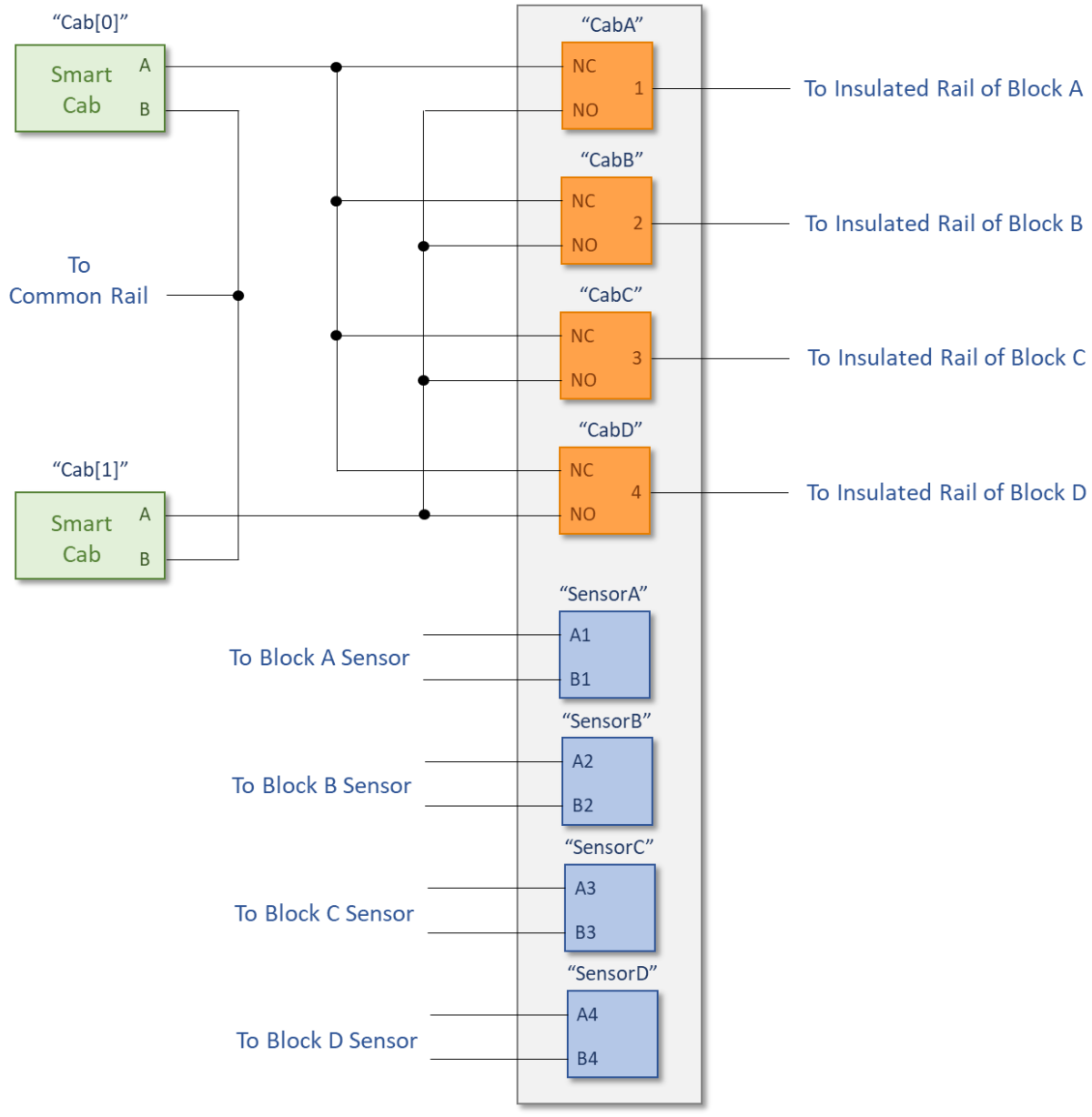

Figure 2. CTI Module Wiring Diagram

TCL Programming:

As always, it's best to begin with an "English language" description of what we want our TCL program to accomplish. Then, we'll translate that description into the more formal TCL language syntax that the TBrain program understands.

In this case, we have four track blocks to manage. But since the track layout is a simple oval, the behavior of all track blocks is identical. Thus, we'd like to develop a generic cab control "algorithm" that works for any track block. Then we'll simply apply that algorithm in each of our mainline's four blocks.

With that in mind, here's a description of the cab control algorithm we want our TCL program to perform in each track block:

1) When a train enters this block …

2) If there is traffic in the block ahead, then …

a) Apply the brake on the cab assigned to this block.

b) Wait until any traffic in the block ahead clears, then …

c) Release the brake on the cab assigned to this block.

3) Assign this block's cab to the block ahead.

Working through a few test cases should convince you that this sequence of operations maintains a buffer zone between trains, and routes each throttle ahead of its assigned train as it moves from block to block.

Now it's time to write the TCL program to implement our cab control algorithm. For purposes of discussion, we'll pick an arbitrary track block, block 'B', and examine in detail the code for that block. The code for the remaining three blocks will be identical. Only the block names will change. The entire TCL program is given later in this App note.

Referring to Step #1 of our algorithm, the first thing we'll need to do is detect the entry of a train into the block. Using our current detection sensors, for our representative block B, we can write:

```
When SensorB = True Do …
```

Next, it's on to Step 2, which states:

2) If there is traffic in the block ahead, then …

a) Apply the brake on the cab assigned to this block.

b) Wait until any traffic in the block ahead clears, then …

c) Release the brake on the cab assigned to this block.

To find out if there is traffic in the block ahead, we'll just need to examine the state of the sensor in that block (in this case the block ahead is block C, so we'll be checking the state of *SensorC*). TCL's ***"If-Then***" statement is the ideal tool for the job:

```
If SensorC = True Then …
```

In order to stop the train in this block, we first need to know which cab is presently routed to this block, so we can apply its brake. To find out, we can simply check the state of our cab assignment controller, *CabB*. Recall from the wiring diagram of Figure 1 that if *CabB* is *Off (i.e. = 0)*, then *Cab[0]* is assigned to block B; if *CabB* is *On (i.e. = 1)*, then *Cab[1]* is assigned to block B. Thus, we can use the value of controller *CabB* as the index into our array of cabs. (Incidentally, that's precisely why we declared our Smart Cabs as a two-element array.)

Our *If-Then* statement for Step 2 then begins:

```
If SensorC = True Then
   Cab[CabB].Brake = On
```

Then, it's on to steps 2b and 2c:

2) If there is traffic in the block ahead, then …

a) Apply the brake on the cab assigned to this block.

b) Wait until any traffic in the block ahead clears, then …

c) Release the brake on the cab assigned to this block.

Once again, we'll use the occupancy sensor of the block ahead to tell us when the coast is clear. This time, as the description above implies, TCL's ***Wait-Until*** statement is our tool of choice:

```
Wait Until SensorC = False Then …
```

The final action of Step 2 is to release the brake on the cab assigned to block B.

```
Cab[CabB].Brake = Off
```

And with that, Step 2 is complete. Here's the code for Step 2 in its entirety:

```
If SensorC = True Then
   Cab[CabB].Brake = On
   Wait Until SensorC = False Then
   Cab[CabB].Brake = Off
End If
```

Finally, in Step 3 we configure block C's cab select controller to assign to block C the same cab that's currently assigned to block B, allowing our train to transition smoothly into the next block.. That's a no-brainer:

```
CabC = CabB
```

That's all there is to it. To recap, here's the whole cab control algorithm for block B:

```
When SensorB = True Do                {Step 1}
  If SensorC = True Then              {Step 2}
    Cab[CabB].Brake = On              {Step 2a}
    Wait Until SensorC = False Then   {Step 2b}
    Cab[CabB].Brake = Off             {Step 2c}
  EndIf
  CabC = CabB                         {Step 3}
```

The cab control program for our entire layout simply adds similar copies of this When-Do for each of our remaining three track blocks, with the block names changed accordingly. The entire program is listed below.

Initialization:

Before we put our cab control system to work, we need a way to get it up and running. The When-Do statements of our cab control algorithm are intended to trigger as trains move in and out of track blocks. But at start-up our trains aren't moving; they're just sitting still. So, we'll need a way to get the ball rolling.

Since we're using current sensors, our TCL program can search for the trains on its own at start-up. Then it can set all controllers to properly initialize the cab assignments prior to operation. We can do it all with a single When-Do statement. Here it is:

```
When $Reset = True Do
     CabD = 0
     CabC = CabD, CabC = SensorD |
     CabB = CabC, CabB = SensorC |
     CabA = CabB, CabA = SensorB |
```

Whenever the system is reset, this When-Do assigns each train a cab. The lead train (the train in the "higher" lettered track block) is assigned to Cab[0], and the trailing train to Cab[1]. For example, with trains starting in Blocks B and C, the train in block C will be assigned to Cab[0] and the train in block B will be assigned Cab[1].

With that initialization complete, our cab control system is ready to roll. We can simply throttle up our two trains using their on-screen pop-up throttles. From then on, each throttle will automatically follow its train around the layout and our built-in collision-avoidance system will automatically keep our two trains safely separated.

Here's our complete TCL program for two-train operation on a four-block track loop:

```
Controls:  CabA, CabB, CabC, CabD
Sensors:   SensorA#, SensorB#, SensorC#, SensorD#
SmartCabs: Cab[2]

Actions:

  When $Reset = True Do
    CabD = 0,
    CabC = CabD, CabC = SensorD |
    CabB = CabC, CabB = SensorC |
    CabA = CabB, CabA = SensorB |

  When SensorA = True Do
    If  SensorB = True Then
      Cab[CabA].Brake = On
      Wait Until SensorB = False Then
      Cab[CabA].Brake = Off
    EndIf
    CabB = CabA

  When SensorB = True Do
    If SensorC = True Then
      Cab[CabB].Brake = On
      Wait Until SensorC = False Then
      Cab[CabB].Brake = Off
    EndIf
    CabC = CabB

  When SensorC = True Do
    If SensorD = True Then
      Cab[CabC].Brake = On
      Wait Until SensorD = False Then
      Cab[CabC].Brake = Off
    EndIf
    CabD = CabC

  When SensorD = True Do
    If SensorA = True Then
      Cab[CabD].Brake = On
      Wait Until SensorA = False Then
      Cab[CabD].Brake = Off
    EndIf
    CabA = CabD
```

Not So Fast:

There's one sticky situation that we may run into in the real world. We recognize the arrival of a train into a new block whenever current begins to be drawn from that block. Then, we immediately apply the brake on that train if there's traffic in the block ahead. That makes perfect sense.

But what happens in the real word if our train has little momentum, is a long multi-engine consist, or has lighted passenger cars. In that case, it's possible that the train could come to rest at a position in which it straddles the boundary between the block it just entered and the block it's about to vacate, continuing to draw current from both blocks. In that case, the train "owns" those two track blocks simultaneously. That's okay, and the algorithm, as stated, deals perfectly well with this situation – as long as there are enough track blocks. But consider our simple case with two trains and four blocks. If one train came to rest such that it owns two blocks and the other train similarly came to rest such that it owns two blocks, then all four blocks are owned, none are available to allow traffic to move ahead, and the system becomes deadlocked – no traffic can move!

Clearly, that's an unlikely scenario. But, while we're at it, we may as well account for this possibility. With a small addition to our TCL code, we can ensure that the program can deal with this situation.

The problem is, as written, our TCL code can't tell the difference between these two situations (both look identical with respect to the sensors, and both indicate occupancy in blocks A and B.

Case 1

Case 2

But actually, there's an easy way to tell the difference. In case 1 above, the train is transitioning between two track blocks. Using our algorithm, the only way that can happen is if the same cab is assigned to both blocks. In contrast, in case 2, the two blocks would have different cab assignments, since the blocks are owned by two different trains. We can use that fact to distinguish the two different cases. In case 1, we'll use a Wait-Until statement to delay application of the brake until the train has fully transitioned into the new block, thereby relinquishing ownership of the earlier block. For example:

```
When SensorB = True Do
  If CabB = CabA Then Wait Until SensorA = False Then EndIf  …
```

With that minor change, our cab control program becomes bulletproof. The final code is shown below.

```
Controls:  CabA, CabB, CabC, CabD
Sensors:   SensorA#, SensorB#, SensorC#, SensorD#
SmartCabs: Cab[2]

Actions:

  When $Reset = True Do
    'Initialize cab assignments: engine in higher lettered track block gets Cab[0]
    CabD = 0
    CabC = CabD, CabC = SensorD |
    CabB = CabC, CabB = SensorC |
    CabA = CabB, CabA = SensorB |

  When SensorA = True Do
    If CabA = CabD Then Wait Until SensorD = False Then EndIf
    If SensorB = True Then
      Cab[CabA].Brake = On
      Wait Until SensorB = False Then
      Cab[CabA].Brake = Off
    EndIf
    CabB = CabA

  When SensorB = True Do
    If CabB = CabA Then Wait Until SensorA = False Then EndIf
    If SensorC = True Then
      Cab[CabB].Brake = On
      Wait Until SensorC = False Then
      Cab[CabB].Brake = Off
    EndIf
    CabC = CabB

  When SensorC = True Do
    If CabC = CabB Then Wait Until SensorB = False Then EndIf
    If SensorD = True Then
      Cab[CabC].Brake = On
      Wait Until SensorD = False Then
      Cab[CabC].Brake = Off
    EndIf
    CabD = CabC

  When SensorD = True Do
    If CabD = CabC Then Wait Until SensorC = False Then EndIf
    If SensorA = True Then
      Cab[CabD].Brake = On
      Wait Until SensorA = False Then
      Cab[CabD].Brake = Off
    EndIf
    CabA = CabD
```

Using Cab Control with Conventional Throttles:

In this example, we've used the features of CTI's computer-controlled *Smart Cab* throttle to smoothly start and stop our trains based on traffic conditions ahead. But this same cab control technique can be used with conventional manual throttles as well.

In that case, a simple Train Brain controller can be substituted for each Smart Cab to provide the automated braking function. We simply modify our TCL code to activate the controller (instead of the Smart Cab's brake) to apply the brake and deactivate the controller to release it. Using this technique, we sacrifice the smooth starts and stops provided by the Smart Cab's simulated inertia feature, but functionally, things work just the same.

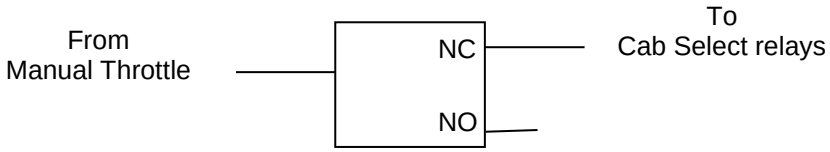

Figure 3. Controller-based brake function

Alternatively, some users may prefer to leave the control of the train completely in the hands of the operator. The computer can be used to handle the automated routing of cabs to follow trains as they move about the layout, leaving the operator free to run his train without worrying about the need to manually route cabs to track blocks. In this case, the operator is fully responsible for obeying trackside signals to avoid collision. He'll receive no help from the PC. For that, only Steps #1, 2b and #3 of the algorithm are required, and the TCL code for our representative block B, is reduced to:

```
When SensorB = True Do
    Wait Until SensorC = False Then
    CabC = CabB
  EndIf
```

Skinning the Cat:

This is by no means the only way to solve the cab control problem. Many other implementations are possible. We suggest you use this example as a starting point, and then let your imagination take over. Try adding code for operation in the reverse direction, add code to handle a passing siding, use Tbrain's graphics tools to create an on-screen CTC panel to show train locations and cab assignments, etc. And most of all, *have fun*.