Applications Note: Using TCL's Addressing Operators

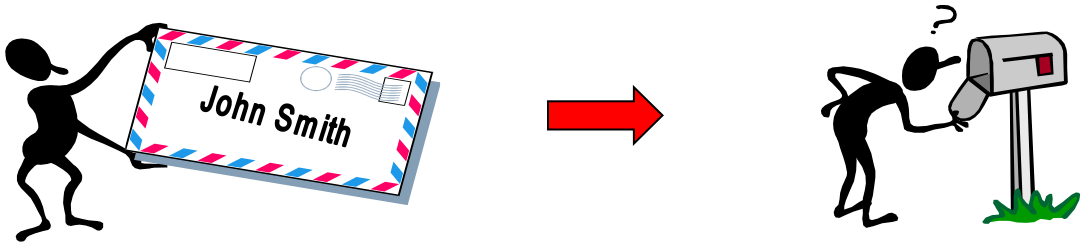
This Applications Note describes the use of the TCL language's "*address-of*" '&', *pointer-to* '*', and *indexing* operators '+' and '++'.

TCL's addressing operators are among of the most powerful features of the language. Using these operators will be fairly familiar to those who are fluent in a computer language such as 'C', however they can be a bit confusing to those with limited programming experience. The following discussion attempts to clarify the use of these TCL features.

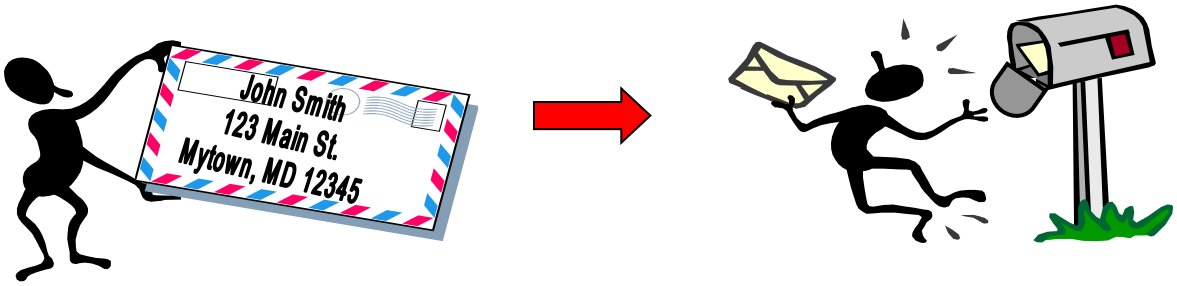
What is an "Address"?

Before discussing TCL's addressing operators, we'll first need to clarify what we mean by the term "*address*" itself.

To do that, let's consider an analogy. Suppose you'd like to send your best friend a letter. On the envelope, you include only his name. Despite the fact that his name is quite meaningful to you, unless your friend is incredibly famous, there's very little hope that he'll ever see your letter.



If however, on the envelope you also include your friend's address, you can feel fairly confident that your letter will get sorted out from the millions of other pieces of mail travelling through the postal system, and arrive promptly at it's destination.



Of course that's just common sense, but it serves to illustrate the power of an *address*. An address is simply a structured way in which just a few short pieces of information (e.g. a house number, a street name, and a zip code) can serve to locate anyone anywhere in the entire world. (Pretty amazing when you stop and think about it.)

Make sense? Good, because your computer works a lot like the post office. Consider that it can store billions of pieces of information inside it. Not to mention communicate with millions of other computers around the world, each with their own billions of pieces of information. How can one tiny computer possibly keep track of all that information? The answer is it does it exactly the same way the post office does. By giving each of those pieces of information an address.

A computer address is simply a number (directly akin to a street address) that tells your computer precisely where a particular piece of information resides. As with the post office, this computer address lets your PC find a piece of information quickly.

Your computer sees everything as simply a piece of data located at a particular address. For example, consider your CTI network. It's made up of a collection of physical entities (controllers for throwing turnouts, sensors for locating trains, throttles for driving locomotives, etc.). But to your PC (which isn't particularly interested in model railroading), those entities look just like any other. Each is simply a piece of data located at a particular address. Your PC can't tell a sensor used to detect a train arriving at the station from the amount of income tax you paid last year. To the PC, both are exactly the same, just pieces of data located at a particular address.

Why Do We Care?

At this point, you're probably wondering what all the fuss is about. After all, if we write the following TCL program everything works just fine. The gate lowers each time the train approaches the crossing, and rises once the train has passed. And we didn't have to know anything at all about that address stuff, right?

```
Controls: CrossingGate
Sensors:  ApproachingCrossing, PastCrossing

Actions:
      When ApproachingCrossing = True Do CrossingGate = On
      When PastCrossing = True Do CrossingGate = Off
```

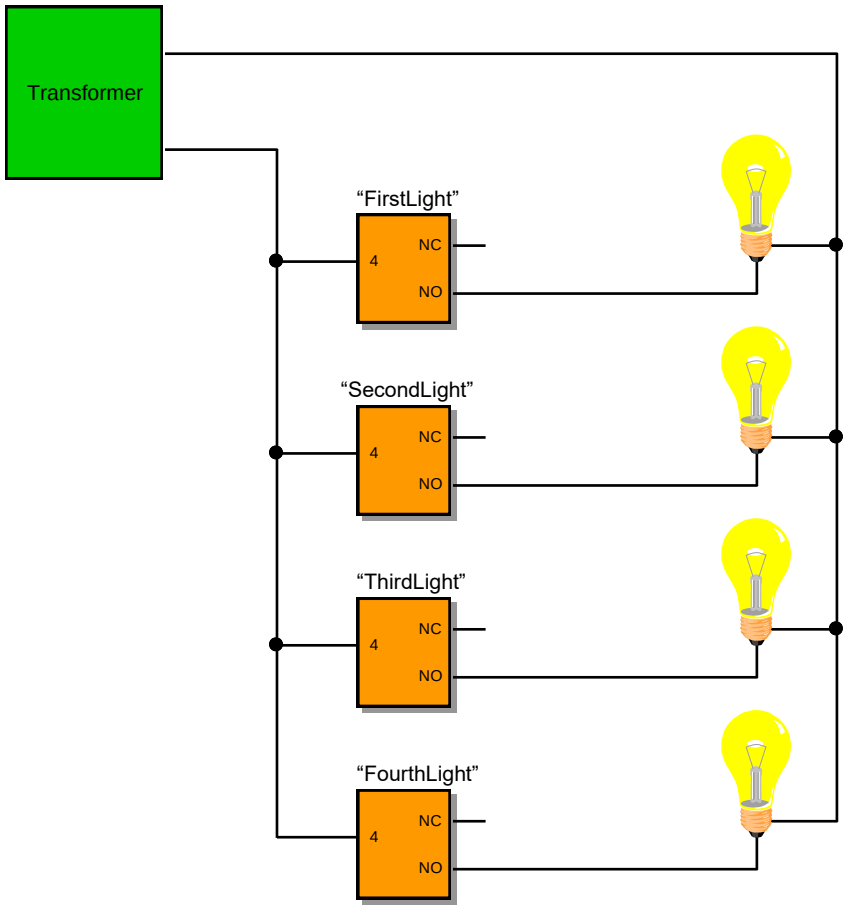
Well, yes that's right. And how did we manage to pull that off? The answer is *TBrain*. Your TBrain software includes a built-in "compiler" (there we go again, another computer term). Each time you run your railroad, TBrain's compiler reads your TCL program and converts all your nice meaningful (to a human anyway) names like ApproachingCrossing and CrossingGate into the bunch of addresses that seem meaningless to us, but which are so dear to your PC. Thanks to TBrain, you can view things in familiar human terms, ignoring all the computer details, and trust that everything will happen exactly the way you want it. (In fact, it's just as if you could get away with sending your letter using only your friend's name.)

So if that's the case, and TBrain takes care of all that for us, why would we ever care about all this address stuff?

Here's why. When we wrote the TCL program above, we knew exactly what we wanted to do. Whenever the sensor we've named *ApprochingCrossing* detects a train we want to activate the controller named *CrossingGate*. Simple enough.

But there will almost certainly be times when, as we're writing our TCL code, we can't specify exactly what we'll need our program to do so easily. Examples include cab control systems, signaling systems, etc, whose operation depends on the current state of the layout (where the trains are, which way switches are thrown, etc), which is unknown when we're writing our TCL code. Our only solution is to write When-Do's to account for every possible scenario, which for even a simple layout can quickly become an astronomical task.

To illustrate, let's first consider a simple nonsensical example: Let's assume we have the circuit shown below, and that we want to continually blink its four light bulbs in a random sequence once per second. (Hey, we said it was nonsensical.)



That poses a bit of a problem. How can we create a TCL program to sequence our lights, when as we're writing it, we have no idea what that random sequence will turn out to be?

Of course, we could always do the following:

```
Controls:  FirstLight, SecondLight, ThirdLight, FourthLight
Variables: LightSelect

Actions:

  Always Do
    LightSelect = $Random, LightSelect = 4#

  When LightSelect = 0 Do FirstLight  = Pulse 1
  When LightSelect = 1 Do SecondLight = Pulse 1
  When LightSelect = 2 Do ThirdLight  = Pulse 1
  When LightSelect = 3 Do FourthLight = Pulse 1
```

Here we've used a When-Do to generate a random number between 0 and 3 once every second, and four other When-Do's to exhaustively account for every possibility of the random number to light our bulbs. To be honest, in this simplistic case, with just four possible outcomes, that's not so bad. But this approach certainly wouldn't be appropriate if there were many more possibilities to account for.

Fortunately, there's a better way that uses TCL's addressing operators. In the hard-coded solution to our bulb lighting problem above, all entity names (*FirstLight, SecondLight*, etc.) are cast in stone in our TCL code, and can't be changed once our TCL program begins running. In contrast, an address is simply a number. As such, like any other number, it can be manipulated "on the fly" as part of our TCL program.

That has some big advantages. For example, here's another solution to our random bulb lighting problem that requires just a single When-Do statement:

```
Controls:  FirstLight, SecondLight, ThirdLight, FourthLight
Variables: LightPointer, LightIndex

Actions:

   Always Do
       LightIndex = $Random, LightIndex = 4#,       {1}
       LightPointer = &FirstLight,                  {2}
       LightPointer = LightIndex +,                 {3}
      *LightPointer = Pulse 1                       {4}
```

As before, in line 1, we generate a random number between 0 and 3, storing it in a variable named *LightIndex*.

Next, in line 2, we make use of TCL's *address-of* operator '&'. Recall that the *address-of* operator yields the address of it's associated entity, in this case the controller named *FirstLight*. Thus, line 2 sets the variable *LightPointer* equal to the address of the controller *FirstLight*.. (It's important

to understand that this is quite different from setting *LightPointer* equal to the value of *FirstLight*.) Having done so, we now say that the variable *LightPointer "points to" FirstLight*.

Now here's where things get interesting. In line 3 we add the random number that we earlier stored in *LightIndex* to the value in variable *LightPointer* (which we know was pointing to controller *FirstLight*). In computer parlance, this bit of arithmetic on a pointer variable is known as *indexing*.

If a variable points to a CTI entity and we add one to that variable, that variable now points to the next CTI entity. If we add two to it, it points to the entity two away, etc. So, since *LightPointer* was pointing to the first light bulb, once we add our random number to it, it now points to the light bulb selected by our random number.

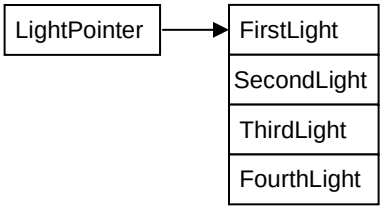
Finally, in line 4 we use the *pointer-to* operator '*'. Recall that the *pointer-to* operator accesses the entity pointed to by its operand, which in this case is variable *LightPointer*. Thus, the action statement of line 4 may be read "*pulse the controller pointed to by LightPointer*". Since *LightPointer* now points to our randomly selected bulb, that bulb will blink on for one second.

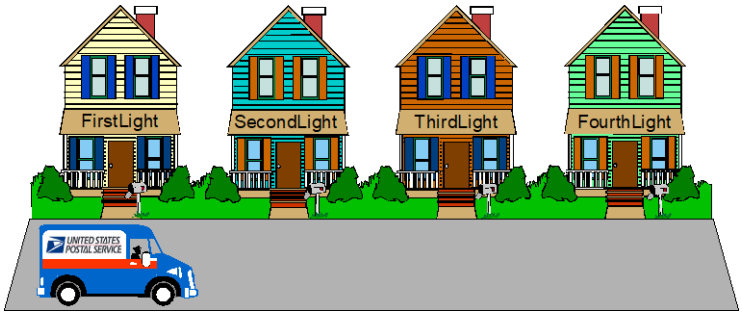
With that, the process is complete, and our Always Do statement executes again, selecting a new random number.

In this way, whether we have four light bulbs or four thousand, it doesn't matter. Using TCL's addressing operators, we can do the while job using just a single When-Do.

Still confused?, Okay, then let's look at the process graphically, again drawing upon our post office analogy. Our first step was to point a variable at the first of our light bulbs. This is analogous to using the street name of our letter's address. It tells the mailman our letter goes to one of the houses on "Light" Street.

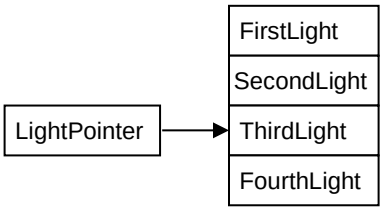
```
Step 1) LightPointer = &FirstLight
```

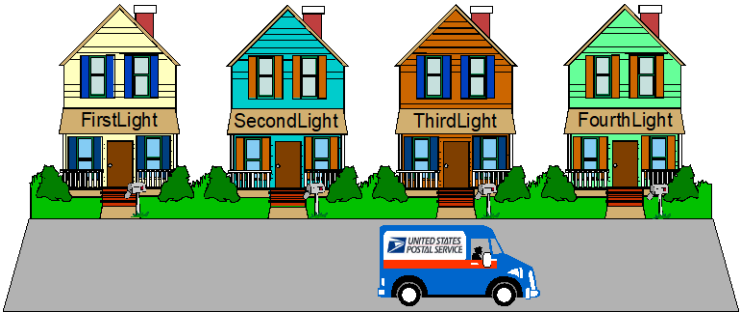




Next, we added an '*index'* to the pointer variable. This is analogous to using the house number of our letter's address. It tells the mailman which house on "Light" street to deliver our letter to.

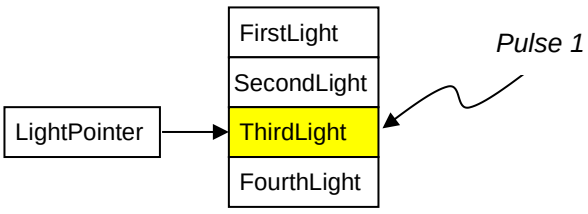
```
Step 2) LightPointer = LightIndex+
```
(here we'll assume LightIndex = 2)





The mailman has now arrived at our letter's destination, and can deliver our letter by placing it into the mailbox at that address (even if the mail carrier doesn't know the name of the person who lives there). Likewise our pointer variable is now pointing at the desired controller, and Tbrain can access that controller (by using the *pointer-to* operator), even though the name of the controller doesn't appear anywhere in the TCL action statement.

```
Step 3) *LightPointer = Pulse 1
```



It may be instructive to try out the above example by copying and pasting the code into TBrain's TCL editor. Run the program and use the View-Controllers and View-Variables menu items to open the controller and variable viewers.

Note the relationship between the values of variables LightIndex and LightPointer. LightIndex bounces randomly between the values 0 and 3 once each second. LightPointer cycles between one of four values. These values are the actual addresses of our four Train Brain controllers. Note how the value of LightPointer corresponds directly to the value of LightIndex, and how the controller that gets activated corresponds directly to the values of these two variables.

Quick Review: Here's what you should now know:

1) All CTI entities have a name, and an address

2) We can access a CTI entity "directly" using its name, or "indirectly" using its address and the '*' *pointer-to* operator.

3) We can find the address of a CTI entity using the '*&' address-of* operator

4) An address is a number. Like all numbers, it can be manipulated using TCL action statements.

5) Indexing using the *'+'* operator may be used to select one member of a group of CTI entities

More on "Address Arithmetic":

We've seen how to find the address of a CTI entity using the *address-of* operator '&'. Once we do, we can use any of the TCL language's arithmetic and logic operators to manipulate that address. Most frequently we'll use the '+' operator to add an *index* to a pointer variable, as in our example above.

However, one complication arises when we add an index. Consider a Smart Cab module, which has several attributes (speed, direction, brake, and momentum). Each of these attributes can be independently accessed in TCL. This is possible because each of these attributes occupies its own *sub-address*.

Once again, let's resort to our mail analogy to help understand. Think of a Smart Cab as an apartment building. An apartment building has a single street address. Yet we need a way to distinguish each individual resident of the building. For that, we typically use an apartment number appended to the apartment building's main street address.

It's the same for a Smart Cab. Each Smart Cab has a single "street address" and contains four "apartments" (speed, direction, brake, and momentum). A Smart Cab entity is identified using the Smart Cab's name followed by the attribute in question, e.g. *cab1.speed*

In computer lingo, such a collection of related entities is called a *data structure*. The TCL language has two built-in indexing operator (+ and ++) designed to make it easier to work with data structures like the Smart Cab. Here's how they work:

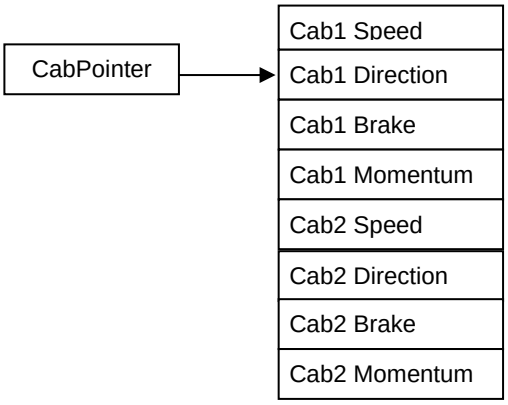
When we perform indexing operations on a Smart Cab, we use the '+' indexing operator to increment our pointer to the next attribute in the same Smart Cab (the next apartment in the same apartment building). We use the '++' operator to increment our pointer to the same attribute in the next Smart Cab (the same apartment number in the next apartment building).

Consider the two TCL action statements:

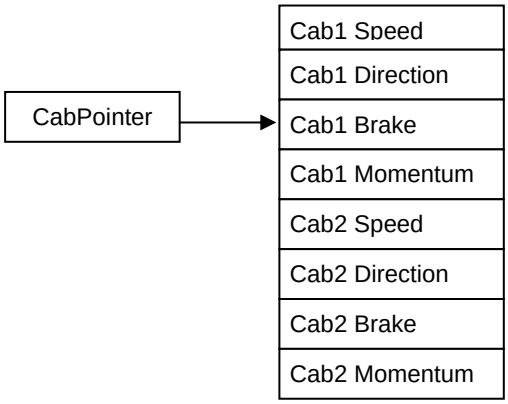
```
When … Do  CabPointer = &Cab1.Direction, CabPointer =  +
When … Do  CabPointer = &Cab1.Direction, CabPointer =  ++
```

The illustrations below show the differing effects of these two statements on our pointer variable.

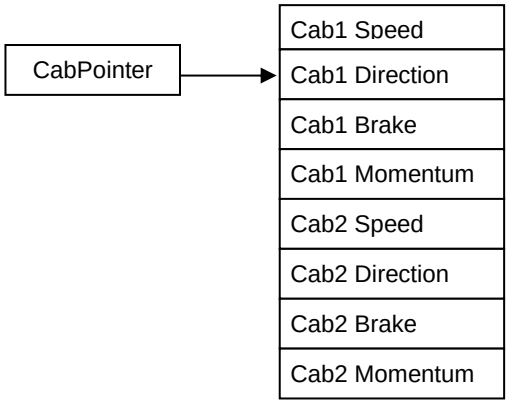
Step 1) CabPointer = &Cab1.Direction



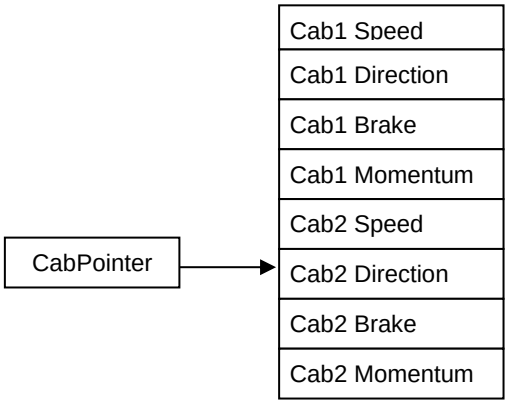
Step 2) CabPointer = +



Step 1) CabPointer = &Cab1.Direction



Step 2) CabPointer = ++



As you can see, the ++ operator causes Tbrain to automatically increment the pointer by the size of the data structure.

Note: When accessing simple entities like controllers, sensors, and signals, the '+' and '++' operators may be used interchangeably. Their effects are identical, since the data structure size of these entities is 1.