Mutilating a Multivibrator with Function Block

## By

## Klint Mahne

## Project Synopsis

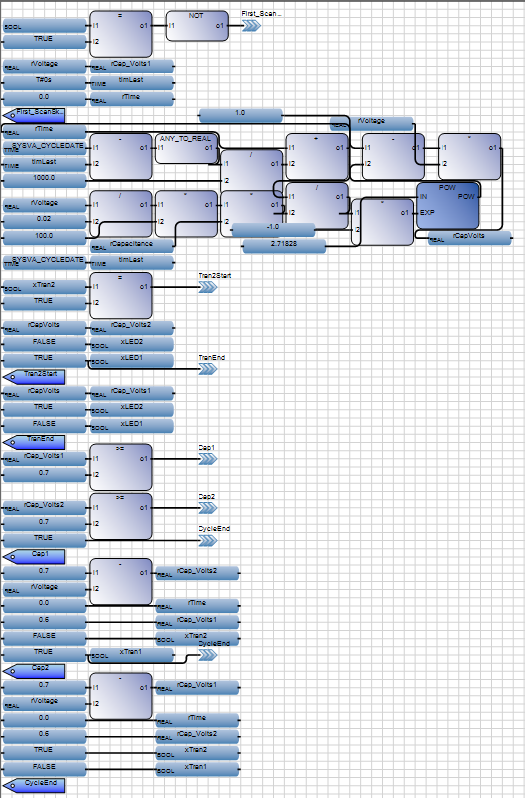
This project is a continuation of the Multivibrator series showing how to replicate a multivibrator in PLCs in multiple different ways. If you are not familiar with the previous iterations of the project, you may want to check them out [HERE](https://forum.digikey.com/t/replicating-a-multivibrator-in-ladder-logic-tutorial/25894) for reference. In this project the aim is to take Structured Text version of the multivibrator and recreate it in the Function Block language. In some of the previous versions there were separate programs and sub-programs that handled different focuses; this project will attempt to have the entire process be contained in one program. This project will highlight both the benefits of using the Function Block language as well as some of the difficulties of implementation.

## Fun or Function – The Balance

One may find it challenging to try and understand where Function Block lies in terms of whether it is appropriate for a program, especially considering that Ladder Logic and Structure Text are both robust enough to fill any gaps the other might leave. A key question you should ask yourself when determining the most appropriate language for your program is “Will other people need to quickly look at this and understand what it does.” That is, if you are working on a project for yourself that nobody else will need to dissect, then you can take whatever liberties you need to. If, however, it is expected that someone else will need to dissect your work, and especially in an environment where time is money, then you will want to prioritize its readability. Function Block language gives you great freedom in how your program is set up, however that is not necessarily a good thing.

Just as Ladder Logic goes rung by rung and Structured Text goes line by line, Function Block has a left to right and top to bottom order of execution. Unlike the other two, however, it is not as rigid and clear in the order of execution. Even though the general structure is left to right and top to bottom, you might have some block that appear out of order due to its input requirements. For instance, you can have a block at the top left corner of the ‘page’, however if the input is connected to the output of another block, the other block resolves before this one is complete. It might not be long until you need unhook a lot of wires when trying to move something, requiring that you understand what went where.

A unique strength of Function Block is that you can connect the blocks directly, meaning you do not need to assign a variable between the function blocks. Ultimately, that allows you to reduce the number of variables you need to keep track of. Unfortunately, this can also translate to unclear trails where it is not apparent how an output was calculated. For instance, in the below picture is the fully functional multivibrator program. With the need to approach if statements in a unique manner, the ‘all of the place’ style of the blocks can create a hard to follow program. If I was to ask what all went into the Tau calculation, as an example, one might not be able to find it despite understanding that it involves capacitance and resistance. The code itself works, and so if I were to build it, instantiate it, and never look at it again then although ugly it would be completely acceptable for its purpose. If someone were to attempt a troubleshoot or modification of the code, they will find themselves frustrated with all the lines intersecting and when they continuously run on without going into a variable to track it might be difficult to understand the purpose.

IAPICTURE 1

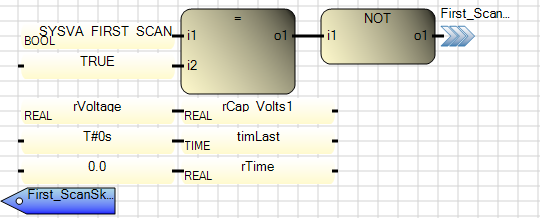
## Organization

The above picture is just one example of how the code can look; there are a few different styles you can use – essentially as long as you have the general left to right nature of the blocks, you can make it work. There are many ways to make it easier to read and feel a bit more organized though. Spreading the blocks out can help so that no blocks “wrap around” and have an output that goes into an input positioned before it. Unfortunately, that can lead to some expanded programs that require a lot of scrolling. Another way to prevent the wrap around would be to create variables that carry results from the output of one function into the input of another. The downside of doing this is that it essentially removes one of the biggest draws to using function block, since you’re no longer reducing the number of variables and what they do. For the multivibrator program, I have separated them out a little bit so that the functions are grouped together but have whitespace to keep them visually away from the other functions. Additionally, I added color to the functions for the ability to distinguish the sections, therefore the examples you will see going forward will look different than the example above. Of course, we could add another layer by adding comments, which I did not do for this one. Ultimately, though, the best course of action to ensure that it’s easy to read for everyone would be to do a little bit of everything.

## Initialization

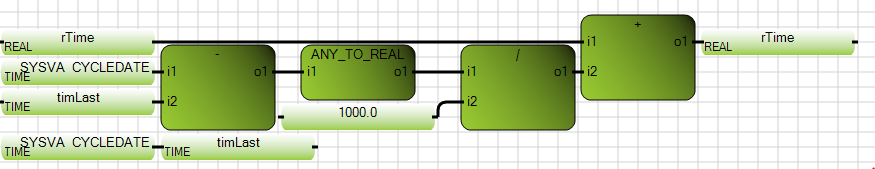
In the Ladder Logic language, we had to use a not gate in conjunction with the \_\_SYSVA\_FIRST\_SCAN to allow some variables to populate so that we could avoid the run-time error of dividing by zero (or the non-existent). In Structured Text, the lockout was stated as an if statement that assigned the variables the values. Function Block has a little bit of both, in which there is an if statement that checks to see if it is the first scan and if so then it assigns values to certain variables. Because Function Block will process everything in order, using the “skip to” is absolutely critical. Although it essentially translates to the same thing, it creates a need to think of the gate as where it needs to go rather than just whether it should be entered or not.

Just as with the other languages, it’s important to establish a state to start in as well. Instead of putting transistor 1 as true this time, however, we instead move the full voltage into the first capacitor and zeroed out the timing variables. This essentially does the same thing as setting transistor 1 as high, it’s just another way to get to it. You can see that with the NOT gate applied to the logic, it will always skip to the First\_ScanSkip tag, meaning the only time the variables under the logic gate are looked at are on the first scan itself. This is essentially how IF statements are handled in Function Block.



## Timing

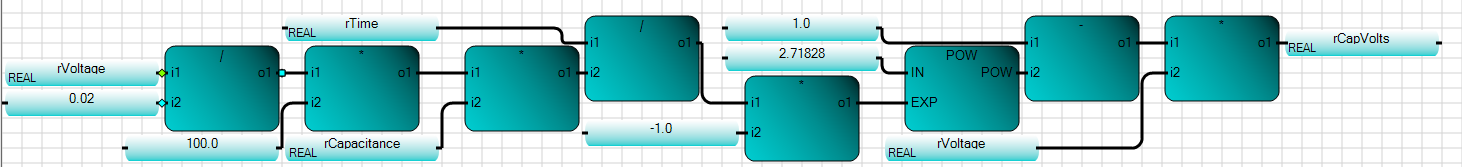
Just as in Structured Text, we do not need to have the timing separated out into its own UDFB nor program as was done in Ladder Logic. Ultimately, the timing itself is really only important for determine the level of voltage over the capacitor. This is significant because it could immediately integrate into the capacitance equation, however I have elected instead to create a variable that the numeric version of time of held and then used that in the capacitance equation for ease of following. In the unorganized version, however, you will find that the time does go directly into the capacitance equation without a variable as the memory location holding the data. Finally, you will see here how important it is to follow the “left to right, top to bottom” format of the language as timLast gets updated after it is used in the subtraction from the \_SYSVA\_CYLEDATE.

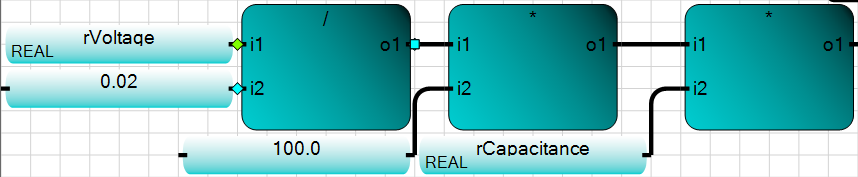


## Tau & More

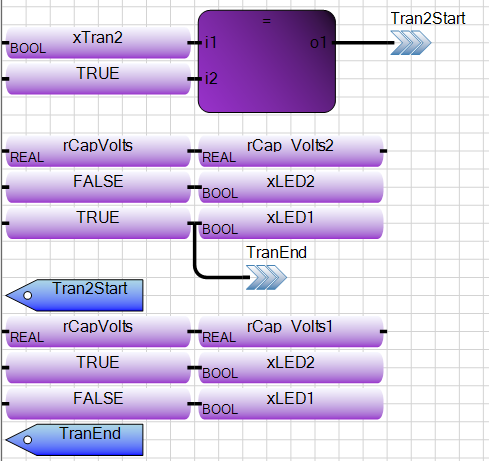
In the Ladder Logic and Structured Text versions of the multivibrator, we made specific variables to house the values of the resistors so that if someone were to build a multivibrator using the components given in the project, they would know what resistances they need. Because one of the major features of Function Block is to eliminate the need for unnecessary variables, however, the resistors are not directly called out. This, too, is another thing to consider when thinking about whether function block is the best course of action for your program, as the lack of variables may lead to confusion for your users.

Using this single chain of blocks (although it is the longest chain with the most variables), we are able to calculate the resistors, tau, and integrate it with time to get the voltage of the capacitor of interest. Again, we would have been able to directly integrate time into it without the variable, but I feel it is much easier to follow this way.



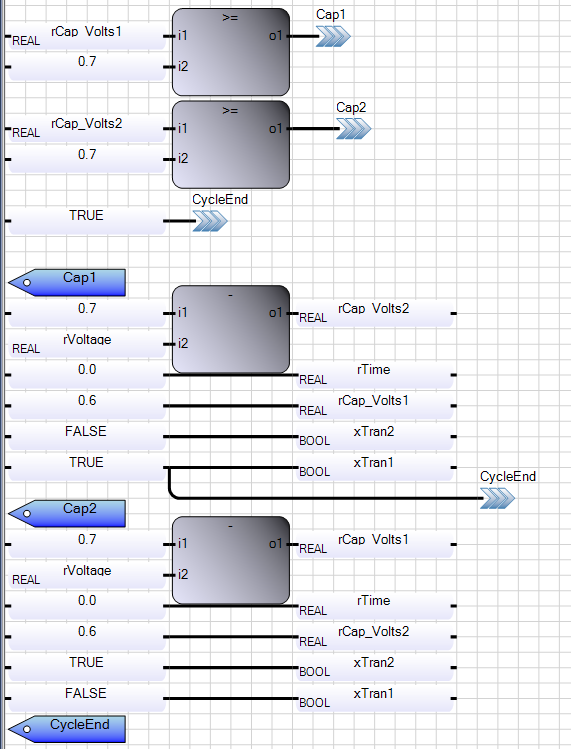
The first part of this chain is the calculation of tau. We see the voltage divided by .02 to represent the 20mA driving the LEDs, multiplied by 100 to represent the Base Resistors, multiplied by rCapacitance to get the tau. This particular string highlights Function Blocks’ ability to reduce the number of variables and instead continue running numbers through until you’re ready for the desired output variable, which in this case is rCapVolts.   


## Transistors

When building the Transistor behaviors in Structured Text, we used an If/Else statement that allow the manipulation of multiple variables simply from one variable being true or not. That same thing can be done in Function Block, however it does look quite a bit different. This section showcases just how the if/Else will work.   
  
 

The placement of the skips and the variable changes are extremely important when using the Function Block ‘If/Else’ logic. Although a bit backward in how one would initially perceive it, the above If/Else makes sense when you go through it. If xTran2 equals True, then jump to Tran2Start. That means it will skill over what would be considered the xTran1 traits. If xTran2 does not equal True, then it the skip is never activated, and it continues into the xTran1 traits. At the end of xTran1, xLED1 is activated as is the skip to the TranEnd tag. This ensures that the xTran2 traits are skipped. Again, it looks a bit odd since normally an If statement executes the code immediately under it when true, but because all of the IF/Elses operate more like a case statements in Function Block the transition is quite bearable.

## Flip-Flop

Just as with the Transistor behavior, the Flip-Flop and Reset portion of the code requires If/Else, however because it should only activate on one of two very specific moments, it actually showcases an If/ElseIf/Else, in which there are two conditions tested. The function block code essentially states that if rCapVolts1 is greater than .7, go to the Cap1End section. If not, check if rCapVolts2 is greater than .7 and if so then go to the Cap2End section. If not, go to the end and skip the rest of the code. Like with the If/Else above, this code requires another skip to the end of the first ‘if’ to ensure that the consecutive code is not performed. Having taken care of the Flip-Flopping and Resetting, that concludes the processes for the multivibrator. 

## Lessons Learned

Thankfully having done the Structured Text multivibrator first, I was able to replicate that version of the multivibrator instead of trying to replicate the original Ladder Logic code. Had I attempted to, I feel the end result would have been even messier than when it was. Having done the 3 languages, I can confidently say that for replicating a multivibrator, an analog device by nature, Structured Text felt the most natural. With that being said, I do feel that the best use of the languages is mixing them together to represent different levels of the program. That is, I feel like Structured Text does the best to build out the base structure of a function, especially if it’s fairly complicated.

Because Structured Text does not indicate what variables are activated or what the values of the variables are, it can make it difficult to know exactly how it’s interacting with everything around it. Therefore, I think on the medium level, Function Blocks do well. That is, take a Structured Text function and instantiate it into a Function Block. This will allow you to be able to track certain inputs and outputs while keeping the code hidden away behind block interfaces. This is best to allow you to design machines with multiple pieces that you can see how the pieces are related to one another. Still, because of it’s ability to get messy and complicated very easily, I think then that multiple Function Blocks that make up a machine should be instantiated together and represented in Ladder Logic.

Ladder Logic, to me, is the most intuitive way to quickly see how multiple machines are interfacing from a high level. When you don’t need to necessarily see the different pieces of a machine, but rather whether or not the machine is correctly interfacing with another machine, Ladder Logic wins. It’s pretty easy to follow, and when done right (and by that I mean do not use it to make a multivibrator), you can have someone who was not behind the initial design get an idea of what is going on fairly easily.

So to recap, I believe that small pieces of a machine (in which each thing that does a specific job) is best built out in Structured Text, where the pieces come together in Function Block to make a machine (in which the machine does a specific job) that is instantiated and represented as a User Defined Function Block in Ladder Logic, which showcases how the different machines are interfacing. Put simply, you could pound in a nail with a screwdriver, loosen a nut with a hammer, and pull out a screw with a wrench, but if you use the right tool for the application you may find your effectiveness (and the enjoyment that comes with it) much greater, and PLC languages are no exception.