Replicating a Multivibrator in Ladder Logic

## By

## Klint Mahne

## Understanding the Multivibrator

A multivibrator is a switching circuit that alternates between two states. One of the most popular multivibrator configurations uses a power supply, four resistors, and two each of capacitors, transistors, and LEDs. To recreate this, we will need to understand the relationship each part has with the operation of the circuit. The challenging part when working with a digital version of analog parts is that we may need multiple variables to capture the function of a single analog component. To help keep track of the purpose of any variable, its name will include the analog component it replicates.

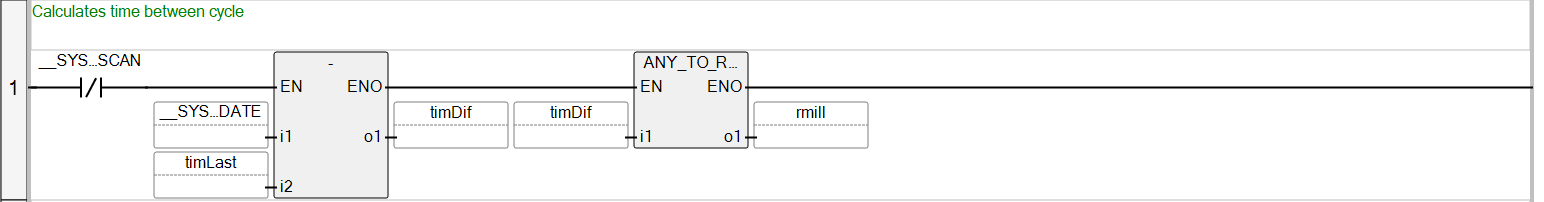
Although it would be easy to create a multivibrator using static inputs for each component, this tutorial will go through the actual calculations used that influences the outputs (in this case would be the two LEDS) so that the timing between states can be changed. This will show the impact that any single component’s value may have on the circuit. This should produce the same result as replacing one of the actual components in the analog circuit.

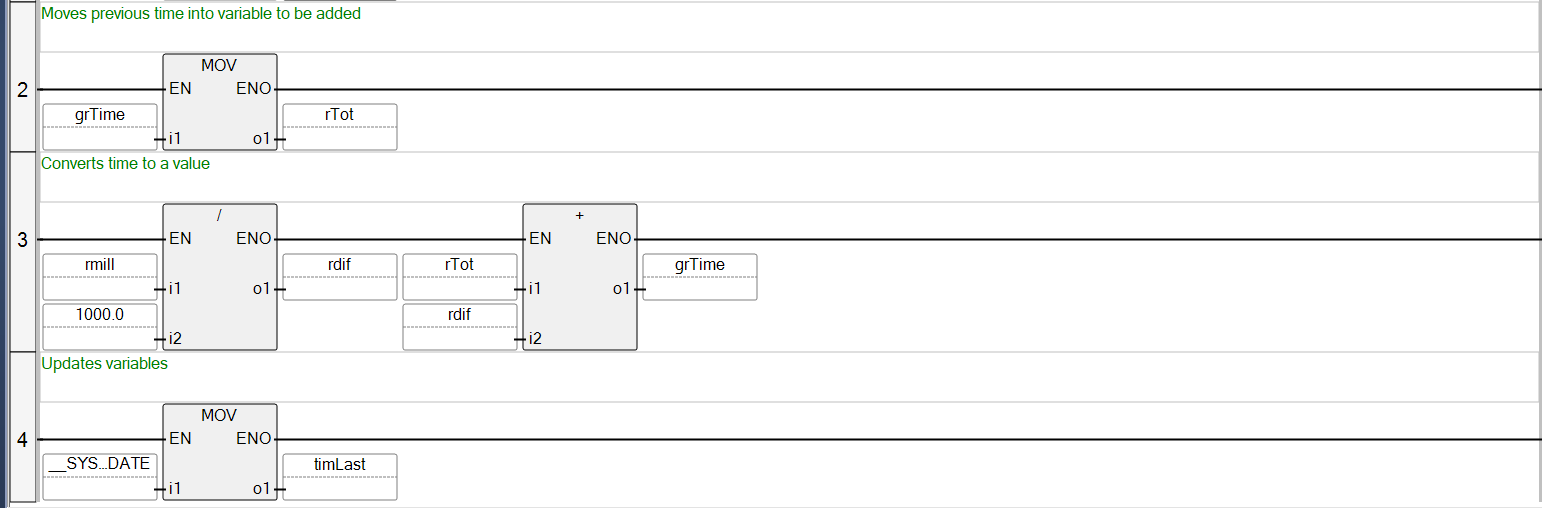
## Parting Programs and Varying Variables

This tutorial is using the Connected Components Workbench version 12 program available on the Allen Bradley website. The programming will be done on the Micro850 simulator using the LEDs on the output to replicate the LEDs in the multivibrator. The main program will be made up of four sub-programs the have specialized tasks designed to make it easy and intuitive to follow the functions. The tutorial will also be using Hungarian notation which indicates the location of the variable (global is denoted with a g in front of it) and the type of variable (x for Boolean, r for real, and tim for time.) In some of the programs, there may be a bit of redundancy and a few variables that might seem unnecessary. It is true that the code could be condensed down farther and some of the variables trimmed out, however for the tutorial it was decided to ensure that it was easy to isolate any potential problems by being able to trace the composition of each variable. To avoid any faults that may occur from variables not being loaded and essentially dividing by zero, it is recommended to put a default valuable of your choosing for variables grVoltage and grCap\_Value, as well as a value of .7 for the variable grCap\_1\_Volts before starting the PLC for the first time.

## Tackling the Timing

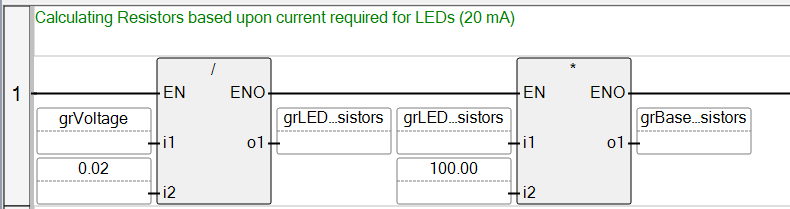
One of the biggest challenges of digitally recreating an analog system is capturing the element of time. If everything was static, all of the calculations could be done once, and a single value could be entered for the time of each gate to open. Because this tutorial allows a user to enter both the voltage and the capacitance, however, time becomes a very important factor in determining how long a portion of the circuit is active. The first sub-program developed is handling the measurement of time – specifically how much time has elapsed from one point to another. The need for this function will become apparent later in the tutorial. For now, let’s start by looking at how to capture and calculate time elapsed.

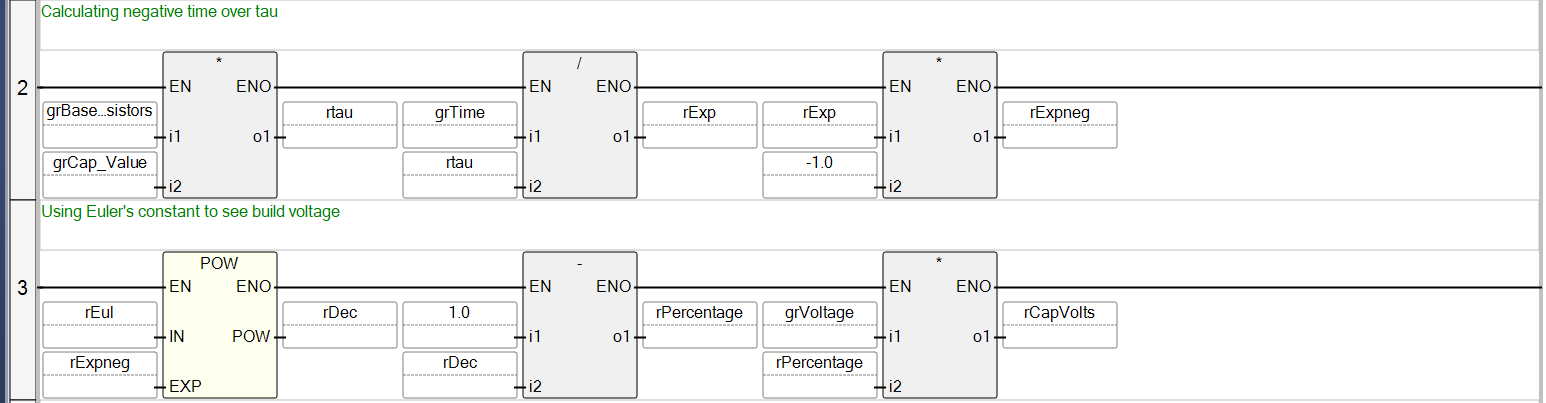


Notice on the first rung there is a normally closed contact named \_\_Sys…Scan. This is actually the abbreviation for \_\_SYSVA\_FIRST\_SCAN. This is not a user created variable, but rather a variable that already exists in the system’s default programming. The \_\_SYSVA\_FIRST\_SCAN is a variable that is only true for the very first cycle when the program is first actioned. Because there are variables that are captured after the first rung, an error would occur since the system doesn’t recognize the variable as having a value. By adding in the \_\_SYSVA\_FIRST\_SCAN variable normally closed, we are telling this program to skip the rung for the very first cycle, which allows later rungs a change to populate the variables first. After the first scan is completed, the gate returns to its normal position and allows the subtraction function to be performed. In the i1 slot of the subtraction you will notice another system variable. This one is called \_\_SYS\_CYCLEDATE; a system variable that captures the time that has elapsed since the program first started. It is best thought as the system’s timestamp at that exact point in time. The i2 slot of the subtraction is timLast – this is a local (noted by its absence of the letter g that otherwise precedes the variable type call-out) time (noted by the ‘tim’ preface of the name) variable. This variable is populated on a later rung and serves as the main reason \_\_SYS\_FIRST\_SCAN exists on this rung. The result of the difference between the two variables is then placed into the variable timDif. Because the difference is still going to be noted in units of time (most likely either milliseconds or microseconds), the ANY\_TO\_REAL function must follow in order to convert the timDif value, convert it, and then store it (in which this case is rmillis, a local ‘real’ type variable).   
The next rung takes a global real variable and stores it into a different memory location in order to use it in an running counter (via means of an adder function).   


The third line takes the result from the first rung and converts it into a smaller variable so the operates in milliseconds rather than full seconds in the first function, and then adds it to the previous time elapsed. As you can see on the second and third rung, grTime is essentially holding it’s total value and then adding the new difference. On the fourth rung, the value in \_\_SYS\_CYCLEDATE is them moved into the timLast memory location that is also used on the first line. Walking through it all together from start to finish, we see that for the first cycle grTime (which starts at 0) moves to rTot, then rmill is divided by 1000 (which is also 0 as 0/1000 is 0), and the result is added to the original (which makes it 0+0). Then, the time is moved to the timLast location. Depending on how fast this happened, it could be either 0 or 1ms, but for this example we’ll call it 1ms. The next cycle, the first rung is now activated so then takes the time, which we can say is 3ms, and subtracts the timestamp of the last cycle. 3ms minus 1ms is 2ms, which is then converted to a real number, 2. On the next rung, grTime moves to rTot, however they are still 0 at this point. Rung 3 activates and turns rmillis, 2, to rdiff, .002. It then adds that value to rTot and places the result in grTime, which would then be .002. Finally, rung 4 moves the current timestamp (2ms) to timLast to be subtracted off of the result of the next cycle. This is repeated thousands of times per second, each one to the total of the grTime.

## Creating Capacitor Control

The second sub-program holds the functions that recreates the role that capacitors play in the multi-vibrator. There are resistors in this sub-program, however they are automatically calculated based upon a few different factors. Firstly, for a transistor to work as a gate, the beta (that is, the product of the resistor over the base to the resistor over the collector on the transistor) will be 100. What this means is that whatever the value of the resistance on the LED is, the resistor over the base of the transistor will be 100 times that value. The value of the resistor on the LED is determine by taking the value entered for the voltage source and dividing it by .02, which represents the 20mA required to drive the LED. Put simply, the first rung of the program calculates both values of resistors necessary to run the multi-vibrator by using the voltage source input by the user.

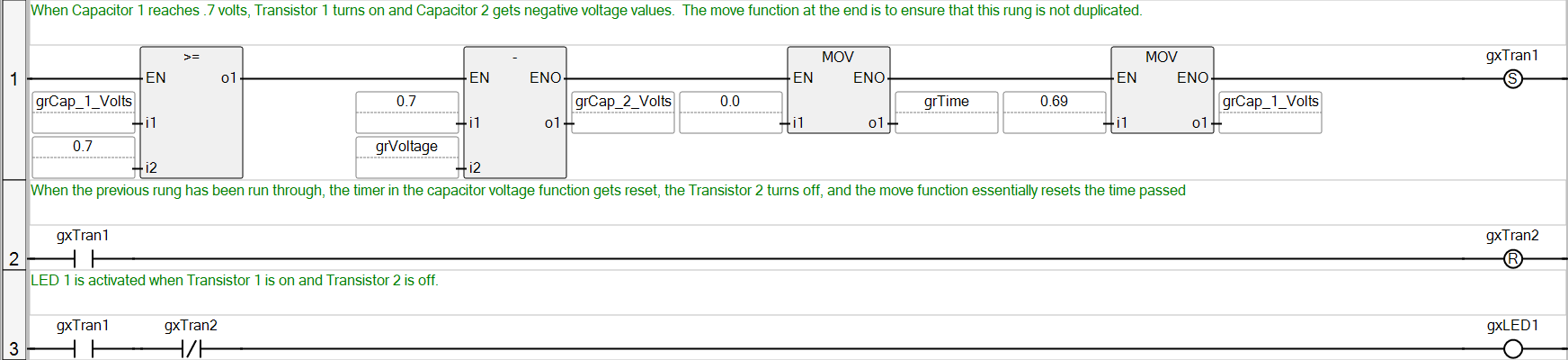
The reason the resistors play such an important role in the charge of the capacitor is because of the formula that determines the voltage over the capacitor at any point in time. Remember that the transistor will turn on when the voltage on the base reaches .7 volts. This happens when the voltage of the capacitor reaches .7v – so we must know the voltage over the capacitor to trigger the gate. Voltage over the capacitor is equal to the voltage source multiplied by one minus euler’s constant to the negative power of time divided by the product of resistance and capacitance (also known as tau). Written, it looks like Vc = Vs(1-e(-t/RC)). This means that time, voltage source, resistance, and capacitance are all required to determine whether the transistor will be on.   


Rung 2 takes care of the negative time over tau by multiplying the base resistors and capacitor value to get the value for tau. Note that tau is a local variable because it doesn’t need to be used outside of this sub-program. grTime is brought over from the sub-program that calculated the time and is divided by tau to get the exponent. rExp needs to be a negative value, and although we could have just used the neg function, we instead multiplied it by -1 for the same effect. Once this was done, everything in the exponent of Euler’s constant was taken care of, allowing the rest of the calculation to be performed on Rung 3.

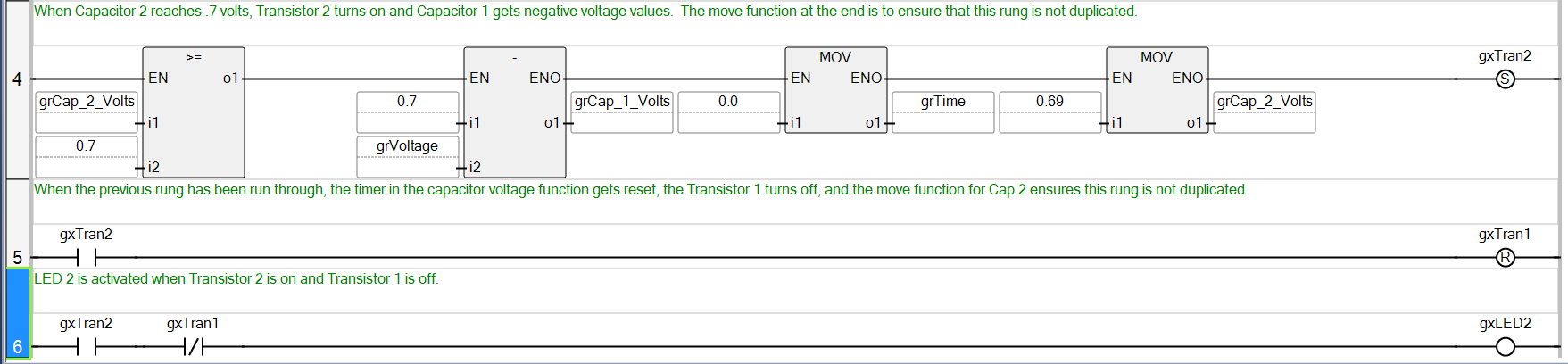
Note that although this tutorial uses a variable called rEul for Euler’s constant with a value of 2.71828, one could just as easily enter the number itself into the box. The result of the power function goes into rDec, which stands for a real decimal number, and is then subtracted by 1 to get rPercentage. grVoltage is multiplied by the rPercentage with the resulting voltage in the capacitor value being storing in the variable rCapVolts . You may be asking yourself how we can have one variable of capacitor charge when there are two capacitors on the circuit? This is where ladder logic’s strength shines, and we use the same variable go to two separate variables depending on the which transistor is charging.

Recall that when a transistor is on as a gate, the capacitor allows current to flow it, effectively making the capacitor’s charge irrelevant if it’s above .7 volts. Rungs 4 and 5 do the same thing but by using the transistors as normally open contacts. That is, when transistor 1 is on, only the capacitor on the base of transistor 2 is charging. Then, when it is switched, the capacitor over the base of transistor 2 stops charging and the value instead goes to the capacitor over the base of transistor 1. This functionality is shown a bit more in the next sub-program.

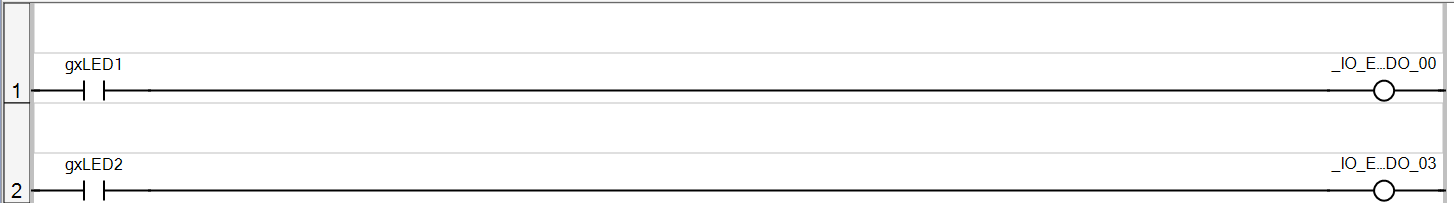
## Bringing the Back and Forth

The third sub-program takes components from both the timing sub-program and the capacitor charge sub-program and causes them to interact on a high level. This program essentially does the setting and resetting that gives the back-and-forth relationship of the circuit.

The program starts off immediately with a comparator on rung 1 which will ensure that the rest of the rung is not performed until the capacitor over transistor 1 has reached .7 volts, in which it will be activated. This rung represents the ‘switchover’ moment from transistor 2 to transistor 1. The subtraction function takes the .7 volts over the base of the transistor 2 and subtracts the source voltage as a representation of the value over the capacitor as the instant of transition. It then takes this value and moves it into the global capacitor 2 memory location in order to be used in the capacitor sub-program. The next function virtually resets the time elapsed to 0, so that the voltage over the capacitor starts over. The final function on the first rung takes the value of .69 and puts it into capacitor 1. This is to denote that the capacitor is essentially at .7 volts while ensuring that the program does not go through this line again unnecessarily. If it did, then every instance after capacitor 1 reached .7 volts would cause capacitor 2 to go negative and would reset the timer, creating a standstill since capacitor 2 would never reach the .7 volts necessary for the switchover. Finally, once the functions have been completed, transistor 1 is now considered ‘set’ and turns on.

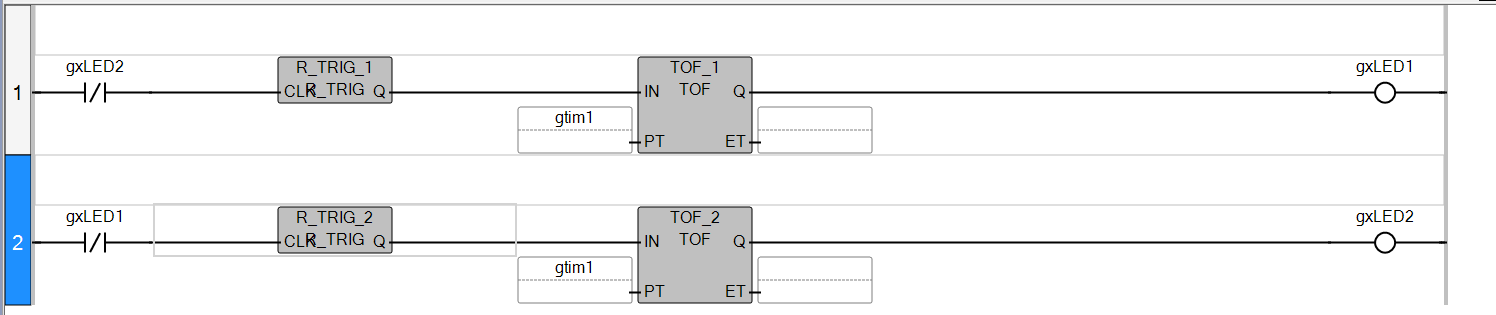
Rung 2 activates when transistor 1 becomes active, which then turns off transistor 2 by activating the ‘reset’ coil. Where set turns something on, reset turns it off. The third rung activates LED 1, but in order to do so two conditions must be met. Firstly, transistor 1 must be on, and transistor 2 must be off. If either of these conditions aren’t met, then something in the circuit has gone awry and the LED’s status will indicate that. Rungs 4, 5, and 6 are functionally duplicates of rungs 1, 2, and 3, but on the other side of the circuit.

## Lighting the LEDs

The final sub-program is very simple and mainly exists for ease of reading and setting changes on the outputs. Note in the third sub-program that controlled the back-and-forth nature of the multivibrator that when a transistor was on while the other was off, it activated the corresponding LED. What was actually happening is that it wrote the result (either true or false) into the memory location, however we still needed to map the memory location to the actual outputs.

This program only has two rungs and is extremely simple. When the variable gxLED1 is true, it lights up the LED indicator for output 00, and for gxLED2 it lights up the LED indicator for output 03. The LED indicators are on the PLC itself (in this case on the screen of the simulator), however having this separate sub-program allows you to put actual LEDs on the outputs of your choosing and change them without having to restructure the entire code.

## Code Condensing

Now that you have seen how to make a digital representation of the analog components for a multivibrator, it’s time to condense the code and let the PLC do the work for us. Using a trigger pulse and a time off counter, we can create a multivibrator with only two lines of code (not counting the sub-program to map the gxLED variables to the outputs).

Walking through this we can easily see what is happening. When the program starts, it will start at the leftmost side of the first rung. At this point in time, gxLED2 is not active, so the contact remains closed and continues along the rung. Then, it hits a rising edge trigger which turns a constant source into a pulse. The time-off function (TOF\_1) registers the pulse and then starts its count. It is important to note that without the rising edge trigger pulse, the time-off function will not see anything as off and will remain on constantly. The count is a global time variable set at 3 seconds; however, this can be changed to any length of time. Until the time-off function reaches that time, gxLED1 will be set to true (or on). The program then goes to the next rung, and immediately stops at the first contact. Because gxLED1 is on, this contact will be considered open, and the rest of the rung is ignored.

This continues until TOF\_1 reaches the specified length of time, in which we see the switch happen. gxLED1 turns off at the end of rung 1, and the contact on rung 2 closes. This allows the rising edge trigger on the second rung to send a pulse to TOF\_2 which then keeps gxLED2 true until the amount of time in gtim1 is reached, and the process repeats. This method relies solely on the internal clock of the PLC which makes it much easier to program. Be careful though – this shortcut on the PLC is not so easily translated back to analog whereas the extended program will tell you the value of each component, allowing you to find the parts and build a real version of it. With ladder logic, there are many ways to end up with the same result so make sure to use the proper program for the job.