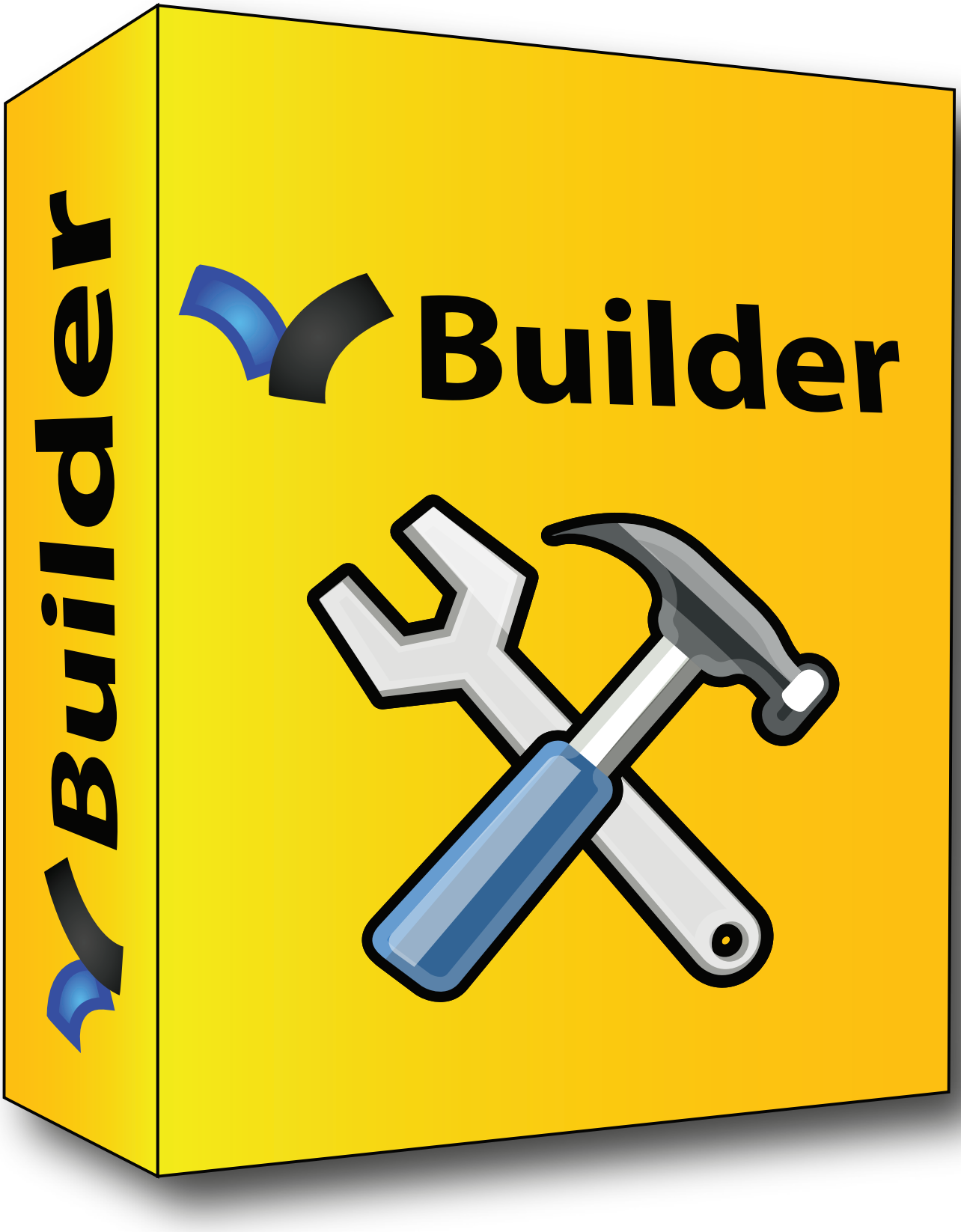


Velocio Builder

copyright 2014, Velocio Networks, Inc.
All rights reserved
No part of this manual may be reproduced, copied, or transmitted in any way without the express prior, written consent of Velocio Networks, Inc. Velocio Networks retains the exclusive rights to all information contained in this manual.

Contents

1. Developing Programs with Velocio Builder

Velocio Builder is, by far, the most advanced, most powerful, intuitive, easy to use and flexible graphical program development package in the automation market. It is very easy to learn and use, yet provides the power to accomplish complex program operations easily. It enhances progammer efficiency, program reliability, documentation and enables the development of re-usable progam components. With the introduction of Velocio Builder (vBuilder), PLC software technology has taken a generational leap.

Some of the major features incorporated into vBuilder are listed below.

- Ladder Logic programming
- Flow Chart programming
- True Subroutines
- Object oriented graphical progamming
- Reusable program objects
- Embedded Objects
- Multiprocessing
- Distributed program operation
- Tag names
- Numerical formats from bits to floating point
- Mixed format mathematical operations
- Debug operation
  - Break points, single step in/single step out/single step over
  - Tag data display during debug
- Advanced functions (ramp, statistics, drum, PID, etc.)
- Motion control

While the list, above, may appear intimidating, vBuilder is designed to be very easy to use. Start with the basics and you'll advance to objects and embedded objects in short order.

This manual will guide you through vBuilder's features, navigation and function details. The material is presented in a sequence starting with quick tutorials in vBuilder Flow Charts and vBuilder Ladder. This is followed by more detailed discussions of Navigation of the software, Ladder function block and Flow Chart function block details. Subsequent chapters get into more advanced features, including debug, program deployment, Embedded Objects and other topics.

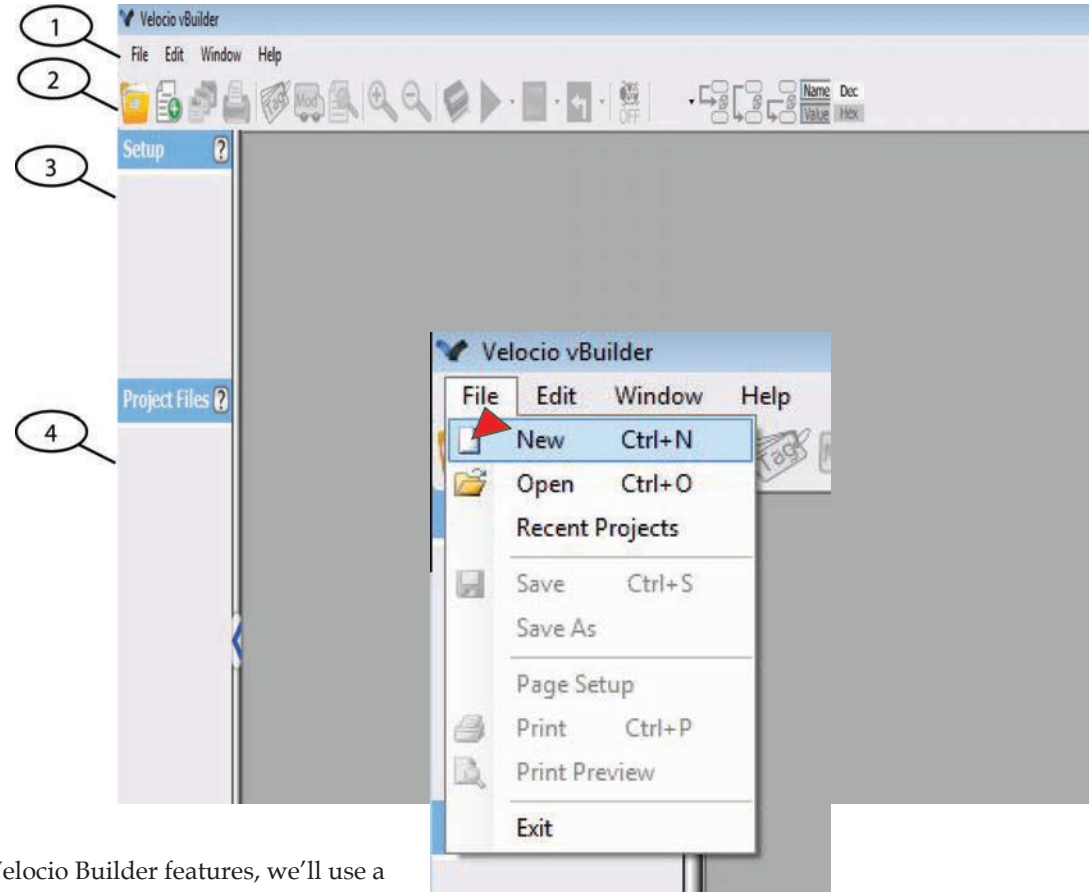
An inexpensive and powerful tool, which will accelerate your learning experience, as well as provide a platform to accelerate your real world application development is one of the available Velocio Simulators. The assortment of available Simulators can be found at Velocio.net. Also available at Velocio.net is a sequence of training lessons, which you can work through to quickly learn to master the advanced control programming skills of the future. Thirdly, training videos can be found at Velocio.net. ***[****understand that the number of available lessons and videos will grow over time****]***

---

2. Control Basics and Tutorials

The next few pages will guide you through entering, debugging and running a couple of simple programs, to introduce programming in vBuilder. You can also watch a video of these examples by selecting it on Velocio Networks' web site ; Velocio.net.

In these tutorials we will enter, debug and run some very simple programs. Actually, we'll do it twice - once using flow charts and once using ladder logic. If you have a Velocio test set, you should be able to implement the example applications and work with them very readily.



The first tutorial example is one that we start really small and grow in three phases. It introduces basic IO operations, followed by is little bit more complex basic operation, then adds a small analog function section to introduce dealing with analogs and numbers.

All that our second example program will do is check one digital input. While the digital input is on (active), it will step through three digital outputs; one step every 2 seconds. If the digital input is switched off, the cycling will stop at the current state. Since we want to give you a feel for some of the advanced Velocio Builder features, we'll use a subroutine in the example.

We first implement the tutorial examples with Flow Chart programming, then with Ladder Logic. We actually could mix the two programming modes (one type for the main program and the other for the subroutine) if we wanted. We won't, but you could try that yourself.

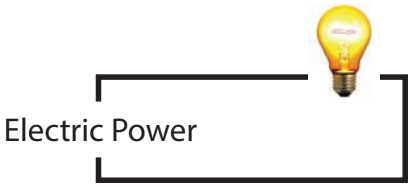
Control System Basics

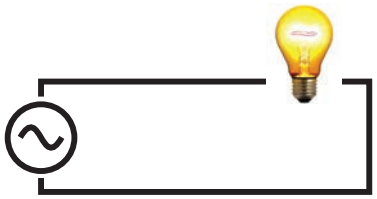
We realize that a lot of people who want to use Velocio PLCs have no background in control systems or PLCs at all. Everything looks mysterious and complicated. In reality, the fundamentals are really pretty simple. On top of that, Velocio's vBuilder makes developing even the most complicated applications a clear, straightforward process.

For those people without a control systems background, the next few pages will demystify the basics of control, bits, tagnames and terminology. If you are already a PLC or control system expert, you can skip these pages and go on to the first tutorial program.

DeMystifying Control, Beginning with Bulbs and Bits

You may not realize it, but you already understand the basics of control! Lets start with a light bulb. The illustration on the right shows a light bulb. It has two electrical connections. If you connect the two connections to an electrical power source, the light bulb will turn on.





The next illustration shows two light bulbs. One is an AC light bulb, like you have in your home. It operates when connected to 120 VAC (Volts of Alternating Current) power. The other is a DC light bulb, which is used in most indicator lights in machinery, or inside your car. Different DC light bulbs operate from different DC voltages. There are light bulbs that operate on 5VDC (Volts Direct Current). There are light bulbs that operate on 10VDC, 12VDC, 24VDC, etcetera. They all work the same way. If you connect the bulb to the electrical power source they are designed for, they will light.

Quick Aside to Explain AC and DC Power

You may have heard about AC and DC power all of your life, yet don't have a clear understanding of it. Let's quickly explain.

In both cases, you have two power connections. The terms AC (altenating current) and DC (Direct Current) are general terms that define the current flow, and more simply, the voltage relationship between the two connections.

Direct Current : Direct Current is a power source that has a constant voltage between the two terminals. A battery is a direct current power supply. Your battery charger for your cell phone converts the AC power from an AC outlet to DC power to charge the cell phone battery. The DC voltage from you car battery is usually about 12VDC. Cell phone batteries and associated chargers are whatever voltage your cell phone manufacturer designed them to be. In common practice, most electrical equipment will operate from a 5VDC, 12VDC or 24VDC power supply.

The illustration on the right shows the symbol commonly used to signify a DC power source. It consists of a stack of long and short parallel lines and labelling of the voltage level of each of the two connections (or teminals). Usually, you'll see this symbol as shown, with two long lines and two shorter lines, stacked long/short/long/short from top to bottom. Occasionally, you may see a symbol with just one long line above one short line. The key is it indicates a DC power source at whatever voltage level it is labelled.

Other DC symbols that you might see in an electrical diagram are on the right. The one with a horizontal line on top of a vertical line and a voltage label above it (5V in this case), is a symbol indicating that whatever is connected to it is connected to the positive voltage side of whatever voltage supply (5V in this case) is indicated.

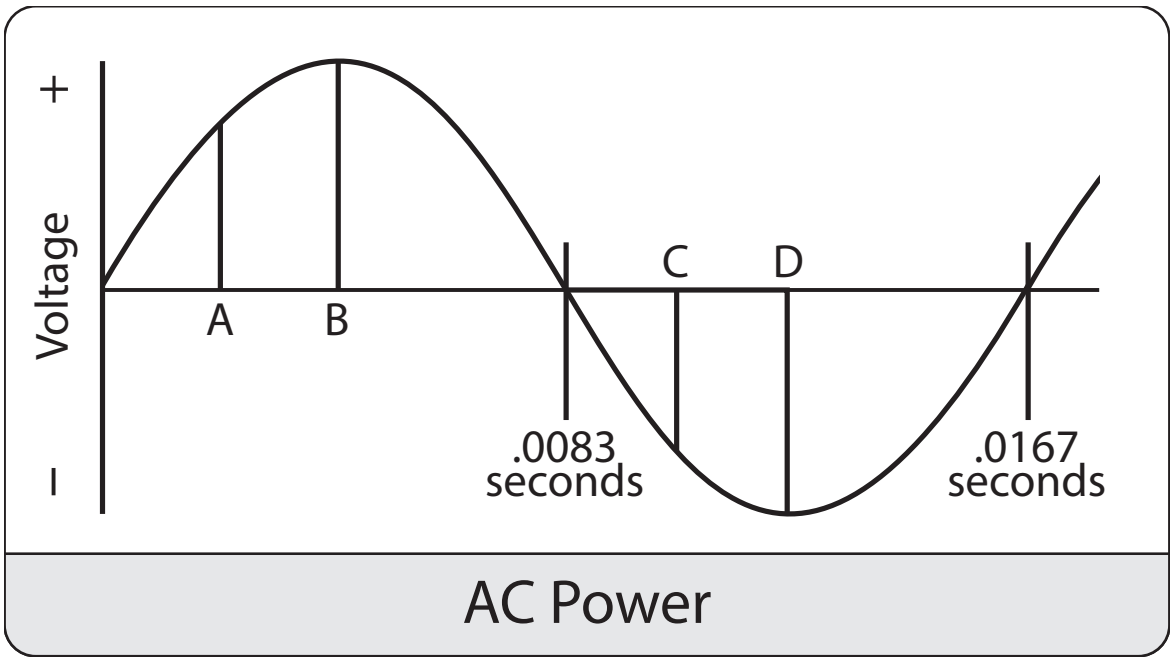
The next symbol is a ground symbol. It is commonly used to indicate connection to the negative side of the DC power source, whether the DC side of the power source is actually connected to physical ground or not.

So a DC power source can be symbolized in either of the two ways shown on the right.

In common practice, DC power either comes from a battery, or from an electrical device called a power supply. A power supply simply converts AC power to DC power.

Alternating Current : Alternating Current is a power source that has a varying (alternating) voltage between the two terminals. The difference in in voltage between the two terminals of an AC power source varies over time. Most common AC power is the 60 cycles per second (or 60 Hertz [abbreviated Hz]) power available from a standard power outlet.

The illustration, below, shows the difference in voltage between the two terminals over time. As you can see, the voltage difference goes from 0 to a positive peak value (at time B), to a negative peak value (at time D) and back to 0 every 0.016666667 seconds. The reason it is every 0.016666667 seconds is that it does this 60 times per second (divide 1 by 60). The voltage difference alternates between the first terminal being at a higher voltage than the second terminal to the being at a lower voltage and back again 60 times per second.



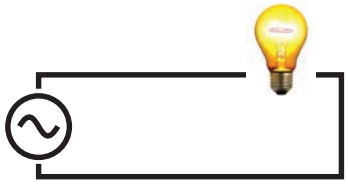
AC is used to transmit power from the power station to your home or business because it is easier and cheaper to convert to very high voltages for transmission and back down to lower voltages for use. Higher voltage transmission results in less loss of power in the transmission lines.

Very little equipment runs directly from AC power. Many motors will (including refrigerator, washing machines and air conditioners) and most light bulbs used for room lighting. If equipment has electronics in it, there will be a power supply that converts the AC to DC for use. Electronics operate on DC power, commonly 5VDC or less, although higher voltages are commonly used for "power" portions of the design (running DC motors, heaters, etc.)

The symbol for AC power source is shown on the right. It shows the sinusoidal alternating of the difference in voltage between the two terminals.

Turning Devices On and Off

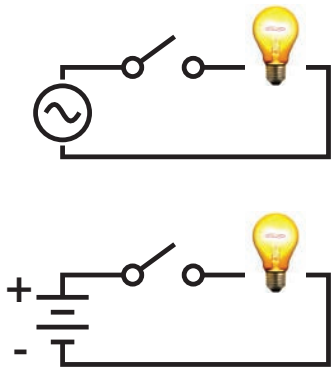
Lets get back to our light bulb. The illustration on the right shows the connection necessary to turn a light bulb on. Electrical current conducts from the terminal that is the higher voltage to the terminal at a lower voltage. In the DC illustration, current would flow from the positive terminal of the power supply, through the light bulb, to the negative terminal. It is this flow of current that heats up the light filament and causes the bulb to light (flourscent & other types of bulbs work fundamentally the same way).



In the AC example, when the terminal on the top is at a higher voltage than the one on the bottom, current flows from the top terminal, through the bulb to the bottom one. When the voltage reverses, the current flow the other way. It doesn't matter. Either way it flows, the filament heats up and provides light.

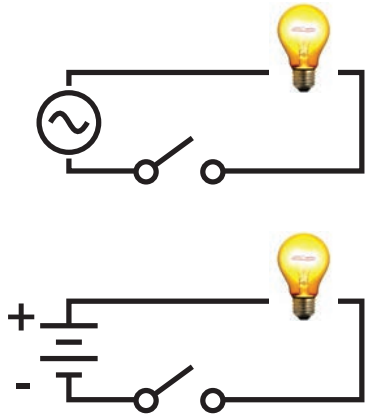
In both the AC and DC illustrations, we have a "circuit". All that is meant by circuit is that the current has a path to flow around from one terminal of the supply, through something, back to the other terminal of the supply - in other words, in a circular path or circuit. If there is not a complete path, the circuit is defined as "open". The light bulb will not turn on.

Now, look at the next illustration on the right. We've put a switch in each of our circuits. Its just like you have in your home. When you "close" the switch, you create a complete path for the current to flow and the light turns on. When you "open" the switch, you create an "open circuit" condition and the light will be off .



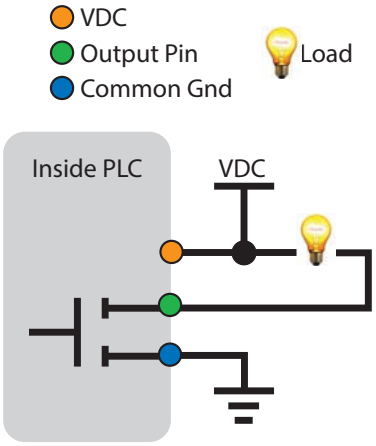
The next set of illustrations show the switch connected on the other side of the light bulb. It doesn't matter where the switch is. If it is closed, the bulb will turn on. If it is open, the bulb will be off.

This same principle works with motors, heaters, and any device that can be turned on or off. If you create a complete circuit for current flow it will be on. If you open the circuit, it will be off.

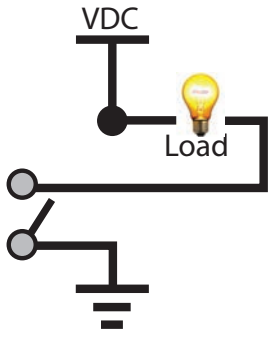


◊ Turning Devices On and Off with a Transistor

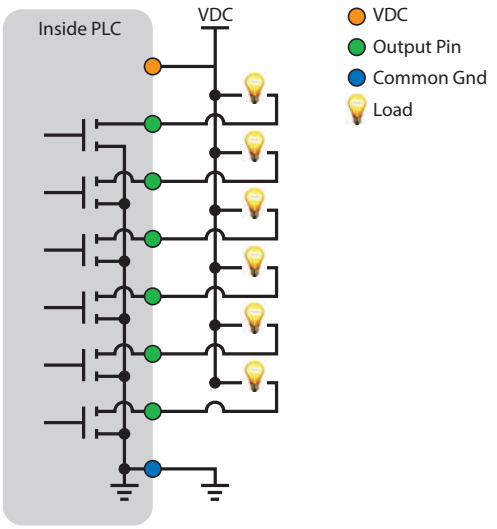
A transistor is an electronic switch for DC circuits. The direct digital outputs in Velocio PLCs are called "sinking" transistor outputs. All sinking means is that they provide the switch on the ground side of the circuit (like the last set of illustrations). They switch, or sink to ground. The figure on the right shows a sinking transistor output connected to a light bulb.



The equivalent circuit, using a switch is shown below it. The whole idea is that if the transistor is turned on, it physically closes a switch in the ground side of the circuit.



In a typical Velocio PLC output port, there are 6 sinking transistor outputs. That means each port can individually turn 6 DC devices on and off. A typical output port is electronically symbolized like the figure on the right. Its shown with connections to 6 light bulbs. In reality, the 6 output connections could be to any DC devices. The Power Supply negative, or ground terminal is connected to one terminal block pin on the port. Internally, all 6 transistors are connected to the ground. The other side of all six transistors are connected to 6 other terminal block pins. They can individually provide switches to ground to individually turn on 6 devices. The last terminal connection is to the positive terminal of the power supply. This is simply for circuit protection, that requires a knowledge of electronics to explain - so we won't here.



Outside of the PLC, one side of each device (symbolized by light bulbs) must be connected to the positive terminal of the power supply. The other side is connected to the PLC terminal (which in turn connects to a transistor).

The sinking transistor outputs in most Velocio PLCs are rated at up to 30 VDC and 200mA current. In a lot of cases, that is enough.

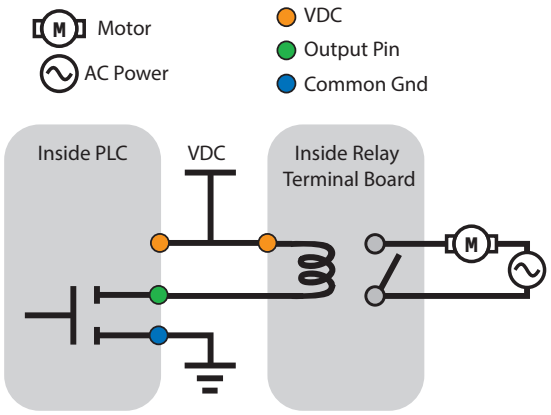
See the specifications for the particular PLC that you are using to learn the exact output voltage and current limits.

That begs the question, what do you do if you need to switch higher DC voltages, higher current or AC? That's covered too.

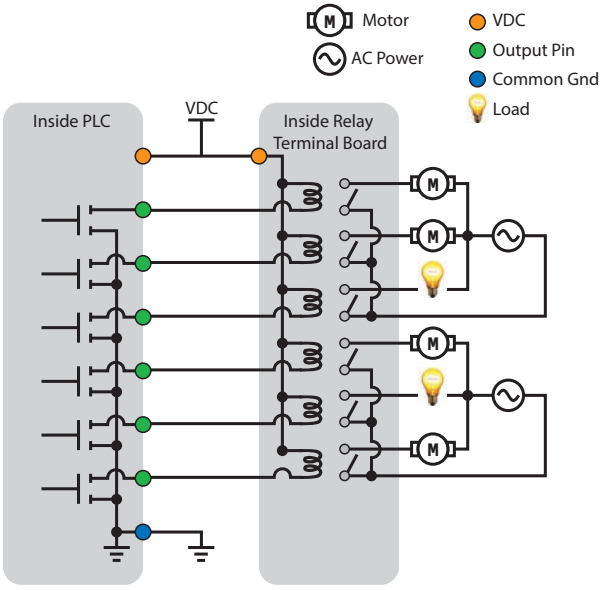
◊ Relay and Transistor Terminal Modules

If you need to turn on devices that require higher DC voltage or current than the direct rating of the PLC, a transistor terminal block module can be connected to the output port. This type of module works just like the one we've discussed in the PLC, so we won't go through it again. It simply has higher ratings.

For even higher DC ratings or to switch (turn on) AC powered devices, a Relay Terminal Block module can be connected to the output port. Typical relay modules have 6 relays. The figure on the right show an equivalent circuit for one of them. A relay is a switch that is controlled by an electrical circuit. If the output transistor is on, it creates a circuit through the relay's coil, the coil will create a magnetic pull on the contact to close the switch and turn the AC (or higher power DC) device on.



This is the basic circuitry for a digital output port connected to a Relay Terminal Block Module. The Relay module is connected to four motors and two lights. There is more to the actual circuitry in both the PLC and the Relay modules, but this is an accurate depiction of the fundamentals of how they work.



Output Bits

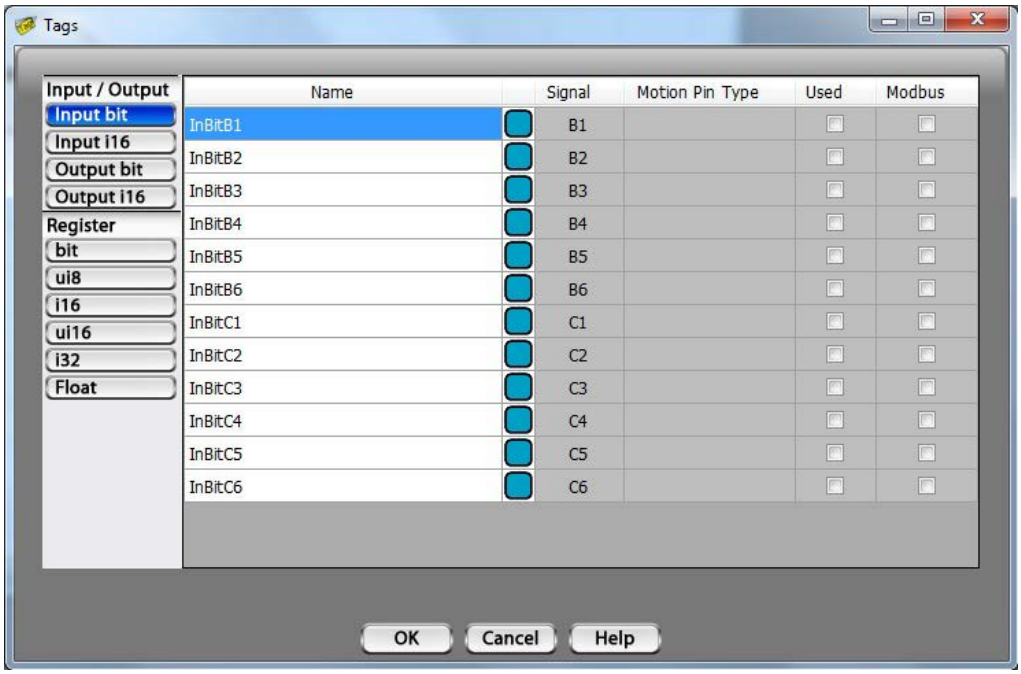
Before we get specifically to output bits, let's discuss what a bit is. Bits relate to the binary number system. In binary, there are only two numbers - 0 and 1. Bits can be used to keep track of, indicate or control anything that has two distinct states, such as on/off, true/false, enable/disable, open/close and so on.

In a PLC program, you can define a large number of what we call tagnames. Tagnames are just meaningful names you give to information. Some of those are bits. A lot of bit tagnames are just information that you keep track of. They are just information and are not directly associated with anything physical. We call those Register bits.

Some bits are directly associated with digital inputs and outputs. We'll give you a peek into a little bit of the programming.

The screen shot shows what we call a "dialog box". This particular dialog box is the one that you can use to see the names of all of the tagnamed data in your program, and to create new ones or change the names of existing ones. On the left side, you can see that there are is a list of different types of tagnames. If you click the selection button for one of these types, the button you have selected will turn blue and the list of all tagnames of that type will be displayed.

As you can see, the tagname types are divided into two sections. On the top, there is a list under "Input/Output" Every tagname in the lists that appear when one of these four categories in selected is directly tied to either a physical "input" or "output". The lower section is a list of "Register" tagname types. Tagnames in these lists are just information or "data". They do not directly sense or measure inputs or control outputs.



The list that is selected is for Input Bits. When you Begin entering your program by first selecting the PLC device or devices to include in your program design, vBuilder knows how many of each type of input and output you have. It automatically assigns a default name for each. What is displayed in this screen shot is the default names for all of the digital Input Bits.

You can rename each input or output bit anything you want. If your output is connected to a fill valve for a liquid tank, you might change the name to fillValve. Using names that are meaningful helps make your program understandable.

What is unique to "Input/Output" tagnames is that the tagnames are directly tied to a particular input or output device. The Output Bits are each directly tied to one of the output transistors on the output ports. For example, the Output Bit named OutBitD1 will directly control the transistor connected to port D, output 1. Any time that your program writes a 1 to an output bit, the associated output transistor turns, which turns on whatever is connected. Anytime a 0 is written to a tagname, the transistor turns off, opening the connected circuit and turning the connected device off.

Sensing Status

An awful lot of what we sense in any control application are binary conditions. We need to know certain things for our logic to determine what to do next. Some examples we might see in real applications are :

- Is the door open or closed?
- Has motion reached the "home" position?
- Does a proximity sensor sense the presence of an object?
- Is a liquid level at its high limit?
- Is power on?

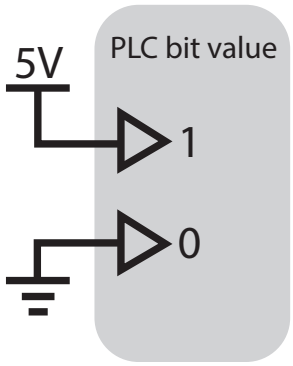
There are hundreds of possibilities.

What all of these conditions have in common is that they are binary - either they are true or they are not. Electrically, they can easily be determined by the state of a binary type sensor. Quite often these sensors are switches. They could be a limit switch, a human operated toggle switch, a proximity switch, an optical sensor, a float switch and so on.

Velocio PLCs sense these types of status with digital inputs. We'll illustrate how digital inputs work by use of a switch. Any other digital inputs works fundamentally the same way - its just a minor variation on a theme.

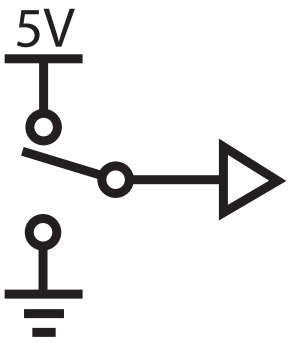
Digital Inputs

Digital inputs in Velocio PLCs are pretty simple. Connections directly into a PLC port digital input simply sense whether voltage level at that port pin is above or below a threshold level. The figure on the right illustrates. The first input is connected to 5V. 5V is above the threshold voltage level, so that input's bit value is 1. The second input is connected to ground. Ground is below the threshold voltage level, so the second digital input's bit value is 0 .



See the PLC's specifications for digital input threshold values. Typically, a maximum value is specified for input value of 0 and a minimum value given for an input value of 1. There will be a small gap between these to values to ensure that inputs clearly reflect the desired states. For example, the maximum value for a 0, might be 0.8V and the minimum value for a 1 might be 2.5V. If you design your system so that the signals fall in line, operation will be reliable.

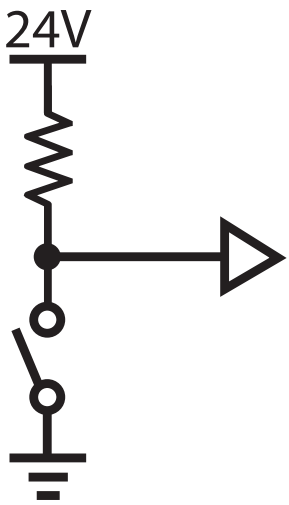
The next figure shows a digital input connected to a switch. If the switch is thrown in the direction shown, the input value will be a 1. If it is thrown in the opposite direction, it will be a 0.



Another example of a digital input circuit is shown next. In this example, the input is tied through a resistor to 24V. Generally, a circuit like this uses a resistor in the 10,000 - 22,000 ohm range. What this does, is pull the input voltage to 24V, if the switch is open.

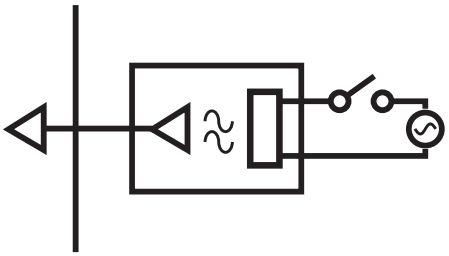
If the switch is closed, the direct connection to ground pulls the input voltage to ground. The fact that there is a connection to 24V through a resistor simply means that a very small (milliamp level) current will flow through the resistor. The input voltage will be ground.

Direct digital input signals to a Velocio PLC can be any DC voltage level up to 30V. That covers almost all DC signals used in most machinery. But the natural next question is, "what about sensing AC or high voltage DC". That's covered. Optocoupler Terminal Block modules are available for just that purpose.



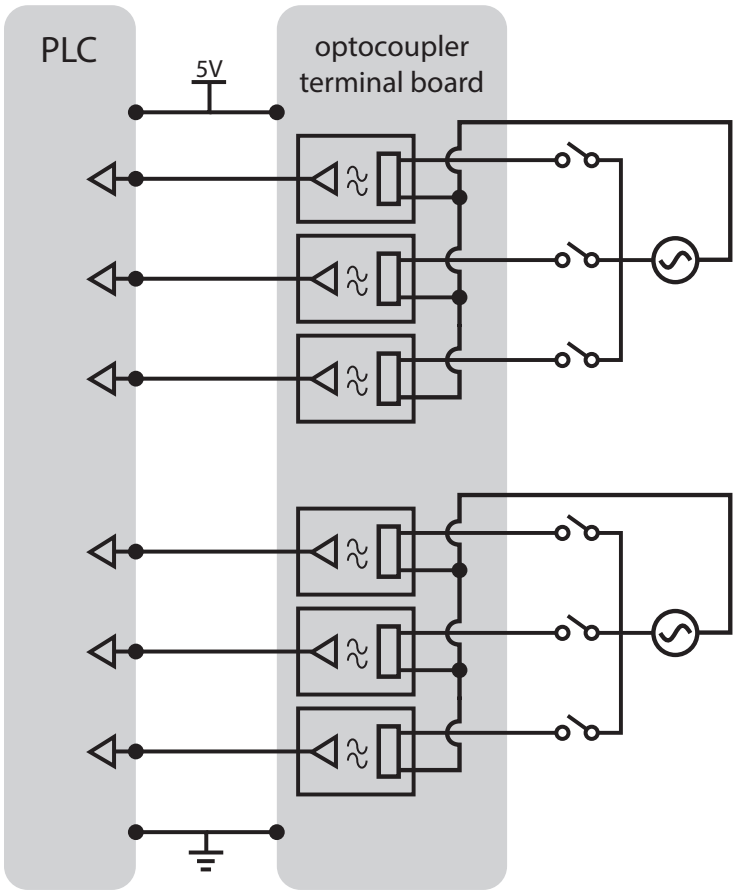
◊ Sensing AC Digital Inputs

AC Signals are sensed by converting them to low level DC signals with an opto-coupler. The figure on the right shows a simplified illustration of an opto-coupled digital input. An optocoupler is an small integrated circuit that has two sections that are not electrically connected. If power is applied to the input section, it will emit light (kind of like the light bulbs on the previous pages; but much smaller and inside the chip). The output section senses light. If it senses that light is on, it will output a voltage that is in the digital input '1' range. If it senses that no light is on, it will output a ground.



By using optocoupler terminal modules, connected to digital input ports, AC signals or high voltage DC signals can be sensed. Just use a Optocoupled Terminal Block module rated at the voltage level that you want to sense. 120VAC, 240VAC and 24VAC are common levels.

The next illustration shows a digital input port connected to an OptoCoupled Terminal Block module. It shows a number of 120VAC inputs connected.



Analog Measurements

In many control system applications, there is a need to perform portions of the control logic based on the value of some measured value. Common measured paramenters include the following.

- Temperature
- Pressure
- Weight
- Liquid level
- Flow rate
- Voltage
- pH
- chemical levels (carbon monoxide in atmosphere, oxygen level, nitrogen level, etc)

There are many other specialized measurements as well.

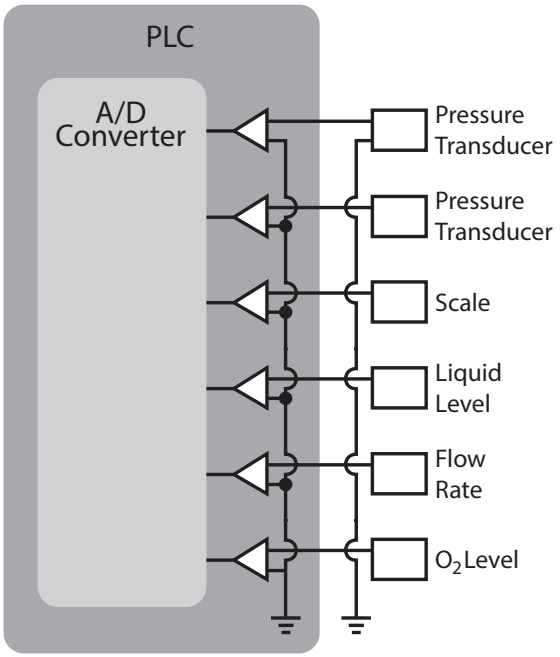
There are devices, called transducers that measure particular parameters and output an analog electrical signal proportionate to the measured parameter. For example, a particular pressure transducer might convert 0 - 1000 psi of pressure to 0 to 5V. In this example, a pressure of 200 psi would be output as 1V, 250 psi would be 1.25V, 750 psi would be 3.75V, etc.

Analog measurements are typically continuously variable electrical signals of some standard range. The most common ranges used in industry are 0-5V, 0-10V and 4-20mA. Velocio's PLC measure signals in these ranges. The particular PLC module that you use must be configured for the particular output range of the transducers that are attached.

As an example, consider an Ace PLC. It's part number is A100-IOAR, where :

I = number of digital input ports (1 or 2)
O = number of digital output ports (1 or 2)
A = number of analog input ports (0, 1 or 2)
R = analog range, where A = 0-5V, B = 0-10V, and C = 0-20mA (used for either 0-20mA or 4-20mA transducers)

Typical transducers have two connections, the output connection and the ground. Typical connection to an anolog input port is shown on the right.



Inside the PLC, the analog value is converted to a digital equivalent. The PLCs analog inputs have 12 bit resolution. That means the signal, whether it be 0-5V, 0-10V or 0-20mA is converted to a value between 0 and 4095. The convertion is linear.

Lets use a 0-3000psi tranducer that outputs a 0-5V signal to the PLC. If the pressure on the transducer is is measuring a pressure that is currently 540 psi, it will output :

$$540/3000 \times 5 = .900V$$

The PLC's analog to digital converter will convert the .9V signal to 0.9 / 5 x 4095 = 737

Now the number 737 probably doesn't mean anything to you. Don't worry, there's a function to take care of that. One of the standard program function blocks is a Scale function. You can set up the Scale to scale 0 to 4095 to 0 to 3000. (The Scale function is described in more detail in this manual.). When that happens, the converted value is 539.92 - the measured pressure with a small descrepency due to the limitations of the resolution.

Temperature measured with thermocouples or RTDs are special cases requiring either a module designed specifically for that purpose, or the use of a transmitter module (basically a transducer) to convert to a standard input range. That is covered and discussed in other Velocio documentation.

Tutorial Example 1 : Flow Chart Implementation

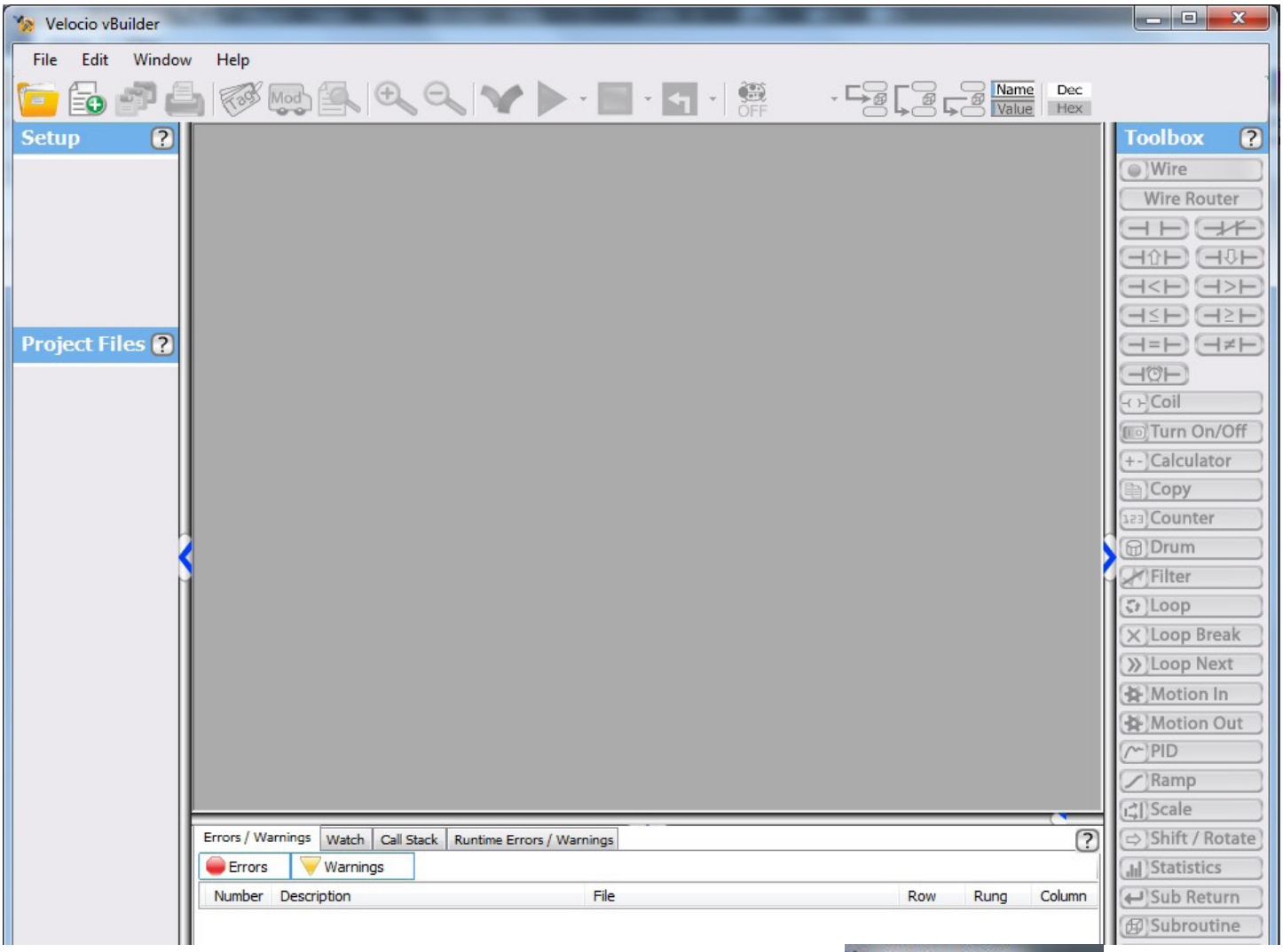
If you have installed Velocio Builder (referred to here as vBuilder) on your computer and followed all of the default settings, you should have a logo, like the one on the right, on your desktop. If you declined to put an icon on your desktop, locate it though the Start menu of your computer.

If you need help installing vBuilder, see Appendix A

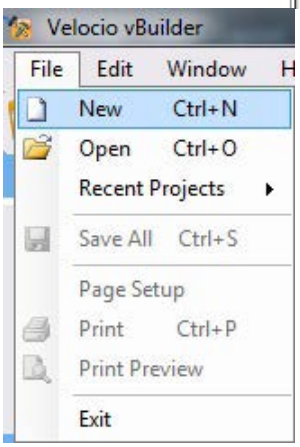
Double click on the Velocio Builder icon to start the program.

If you get a Windows message that asks whether you want to allow vBuilder to make changes to your computer, click Yes. The changes that vBuilder makes to your computer is saving your program files.

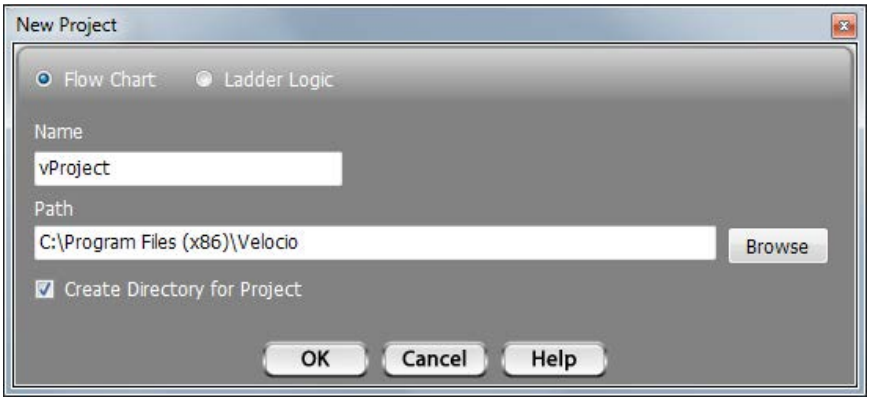
You should get an opening screen that looks like the one shown below. At this point you are ready to begin entering a program.



Begin your program entry by selecting the File Menu, then "New".



A dialog box, like that shown on the right will pop up.

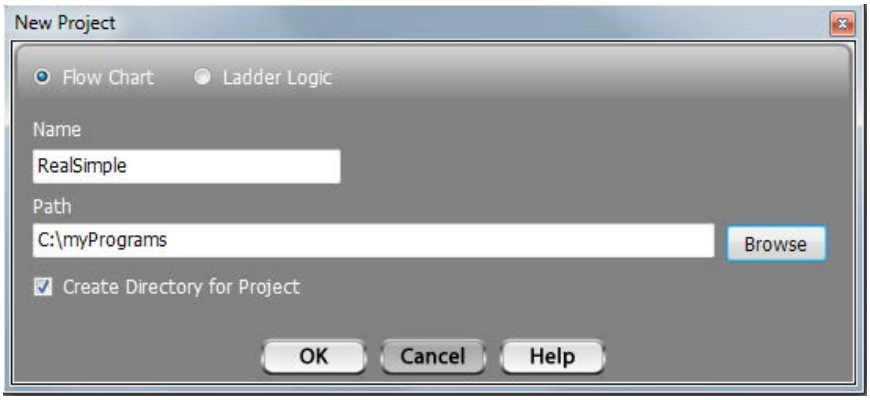


The dialog box opens with default information selected. You're not likely to want to just accept default selections, unless you are just testing things, so we'll make some changes.

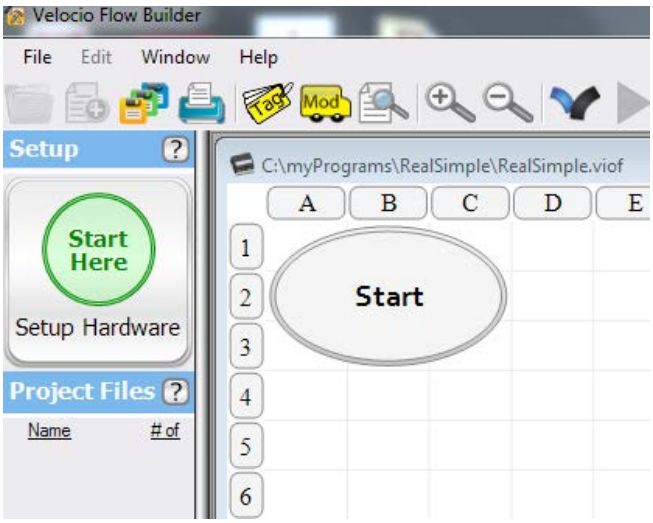
At the top of the dialog box, you see that we can select to write this program with either Flow Chart or Ladder Logic programming. We're going to do this one with Flow Chart programming - so leave the Flow Chart selected.

The Name box allows you to name your program. This first example will be the simplest program that you will ever write. Let's pick an appropriate name, like "RealSimple". Just type RealSimple in the box labeled Name.

The box under "Path" allows you to select the folder, on your computer, to place your program. The one shown is the default location. You can use this location, or select another one. For the purpose of illustration, create a folder called "myPrograms" off your computer's C: drive, and browse to it and select it. Your New Project dialog box should now look like the one shown on the right.



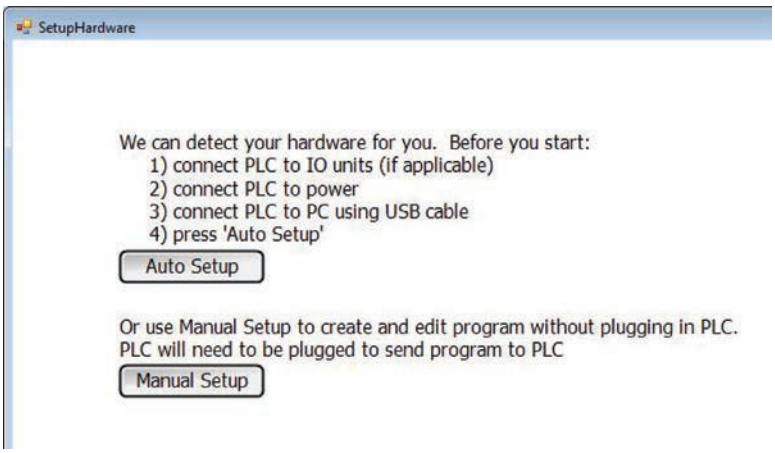
Notice that a programming window, labeled "RealSimple.viof" opens up. Also, in the Setup area, a big green circle labeled "Start Here" comes up. This is your starting point. The first thing you need to do is set up the hardware.



Select the big green button that says "Start Here".

A window will pop up, which explains your hardware configuration options. As stated in the pop up window, you have the choice between connecting up your target system and letting vBuilder read and auto configure, or you can manually define the target application hardware.

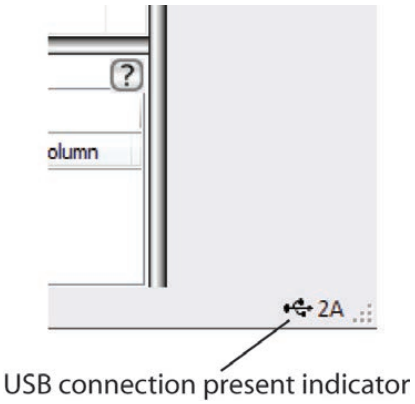
In this tutorial, we will go through both options. However, in order to actually program the PLC, debug and run it, you will need to have either an Ace or a Branch unit.



Using Auto Setup

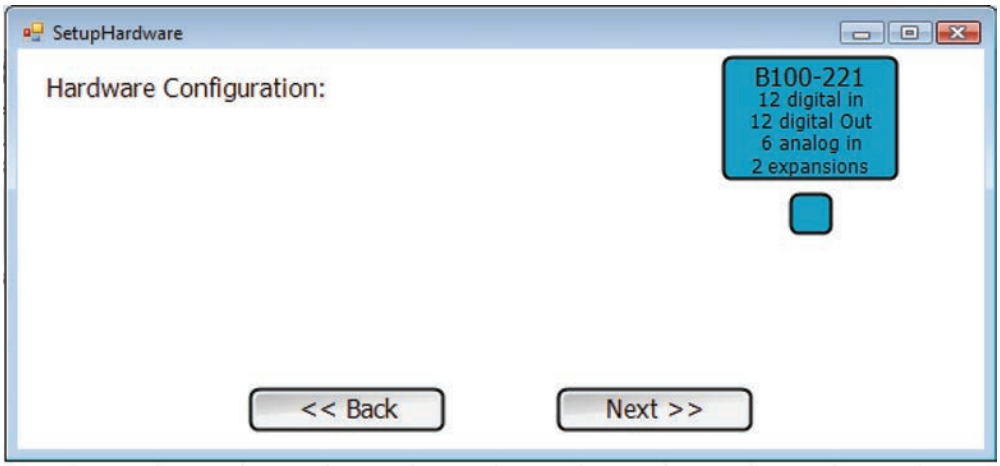
In order for vBuilder to automatically set up your hardware configuration, you must have the Velocio PLC connected like you want the system set up, powered on and connected to the PC.

- Connect up your target Velocio PLC system (in this case, simply either an Ace or a Branch module (no expansion modules are necessary - a Velocio Simulator is ideal))
- Connect a USB cable from your PLC to the Ace or Branch unit
- Power everything on.
- You should see a USB icon, shown on the right, in vBuilder's lower right hand corner. This is an indication that a Velocio PLC is connected (and on) to the PC.
- If everything listed above is OK, select "Auto Setup" in the "Setup Hardware" window.



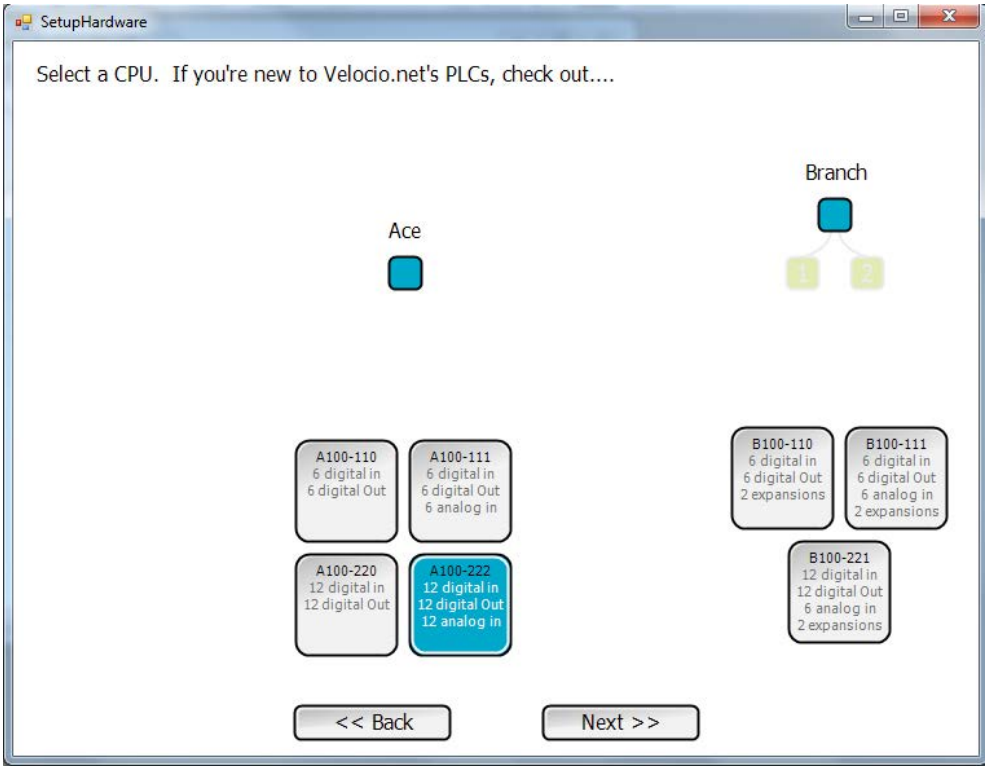
The "Setup Hardware" window should change to display something like what is shown on the right. In this case, it shows that it is configured for a Branch module that has 12 digital inputs (2 ports), 12 digital outputs (2 ports) and 6 analog inputs (1 port). It also has two expansion ports - with nothing attached.

This tutorial can be implemented on any Branch or Ace that has at least 6 digital inputs and 6 digital outputs.



Using Manual Setup

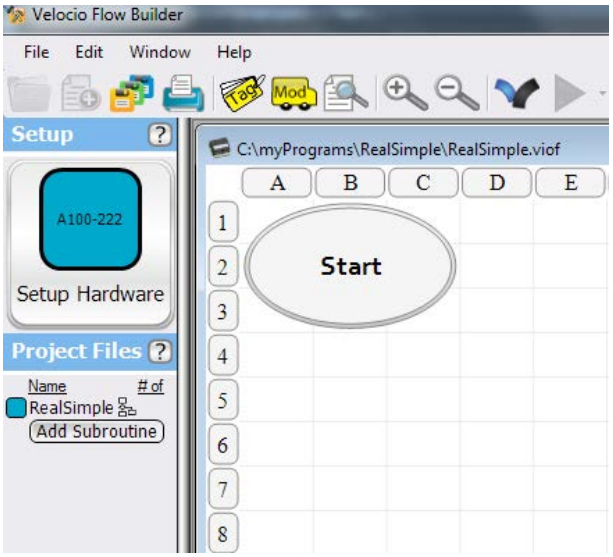
If you don't have the PLC hardware available, you can set the system up manually. Start by selecting "Manual Setup" from the "Setup Hardware" window. The Setup Hardware window will change to something like the image shown on the right.



The first step is to select the PLC main CPU. This will be either an Ace, or a Branch unit. The selections are the labeled, gray squares below the Ace & Branch icons. These selections identify both the type (Ace or Branch) and the associated IO configuration. Select the one that applies to the application. In this tutorial example, any of these units will work. The configuration that you select will turn blue, as shown.

After selecting the PLC hardware for the main PLC,, you must then select "Next" to continue with the configuration of expansion modules, if any. After all of the PLC modules are selected, there will be additional screens, which allow you to define whether any Embedded Subroutines are to be placed in Expansions (if you configure Expansions) and for Stepper Motion and High Speed Counter signal. We won't be doing any of that - just click the Next & finally Done.

When the hardware setup is defined, your screen should look like the image shown on the right. The Setup is shown on the left side, next to the top left corner of the program entry screen. It shows that we're configured for an Ace with 12 DI, 12 DO and 12 AI (the -222 means 2 digital input ports, 2 digital output ports and 2 analog input ports. Each port has 6 IO points). The area just below the hardware setup is the list of project files. The main file is automatically created with the name we defined when we created the new project. In this case it is called "RealSimple".

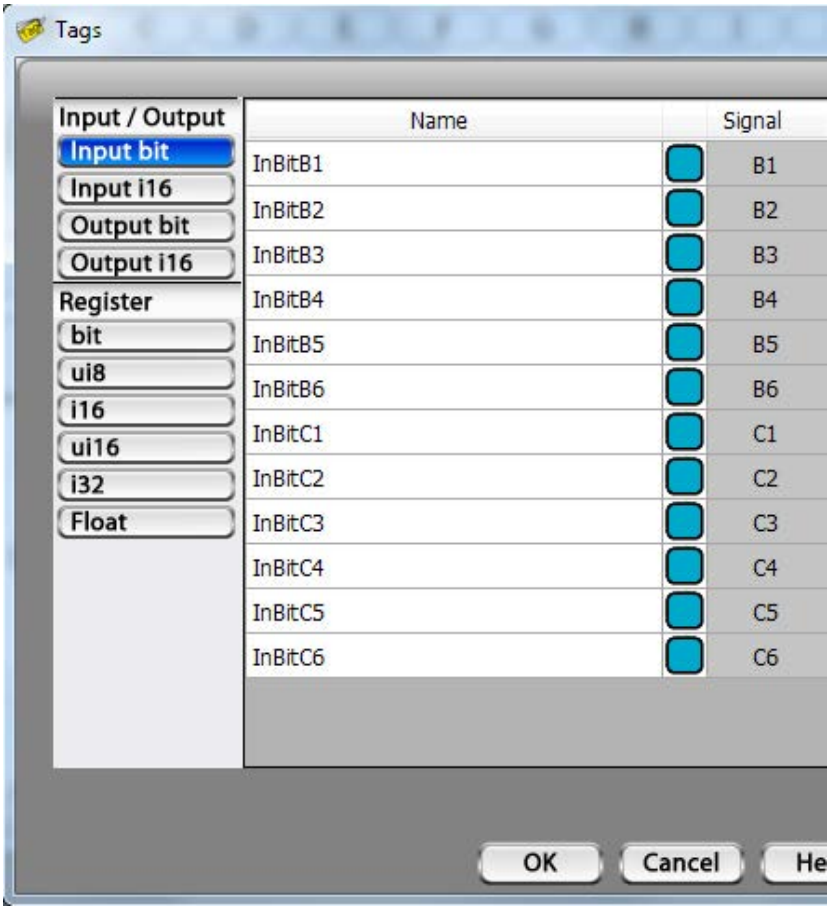


Select the Tag icon on the top tool strip.

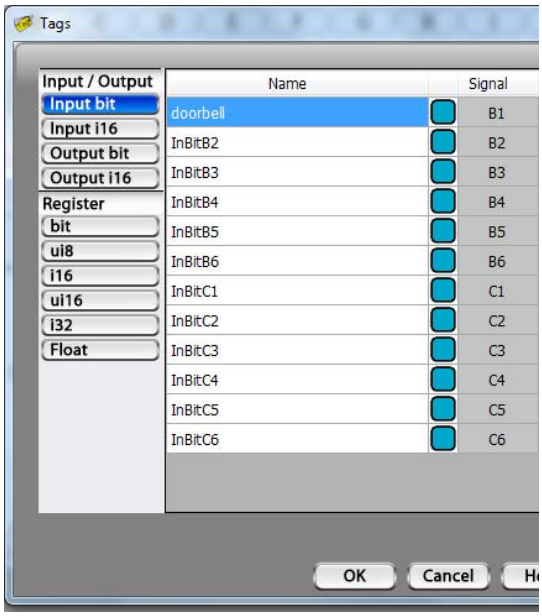
The tagname dialog box will open. Select the various options on the left under Input/Output and Register. You will see that when you select any of the options under Register, you get a blank table. That is because we haven't created any tagnames yet. Register tagnames are just general data identifiers that will mean something and hold information for your program. They are not directly tied to inputs or outputs.

Notice that there are lists of tagnames for Input bit, Input i16, and Output bit. These were automatically created when you set up the PLC hardware for your project. vBuilder knows that an A100-222 has 12 digital inputs. It automatically creates tagnames for each of the digital inputs. These are default names that you can change to more meaningful names, if you want. The default names are pretty simple. For example, InBitB1 is assigned to the digital input connected to the signal B1.

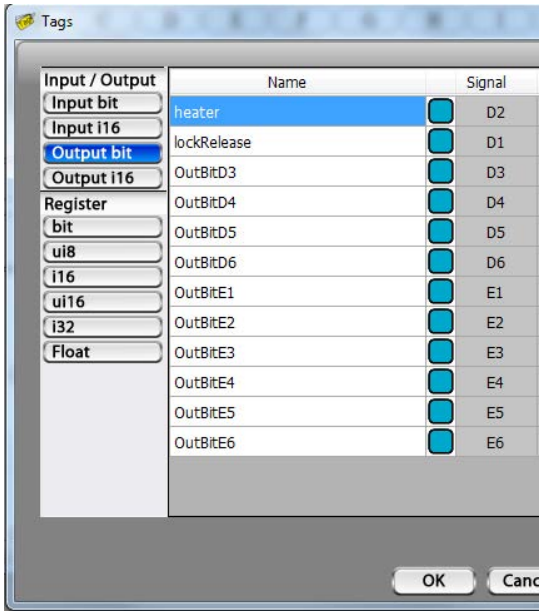
Likewise, tagnames are automatically created for the 12 digital outputs, located in the Output bit list and the 12 analog inputs, located in the Input i16 list. The raw analog input values are signed 16 bit numbers.



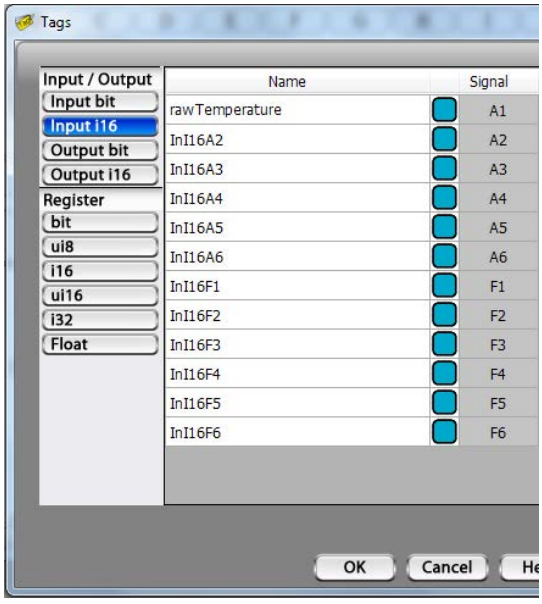
For our example, lets define meaningful names. In the Input bit list, change the first input bit's (the one connected to B1) name to "doorbell", as shown on the right.



In the Output bit list, rename the D1 output "lockRelease", and the D2 output "heater".



Next, change the tagname of the first analog input (Input i16) to "rawTemperature", as shown.



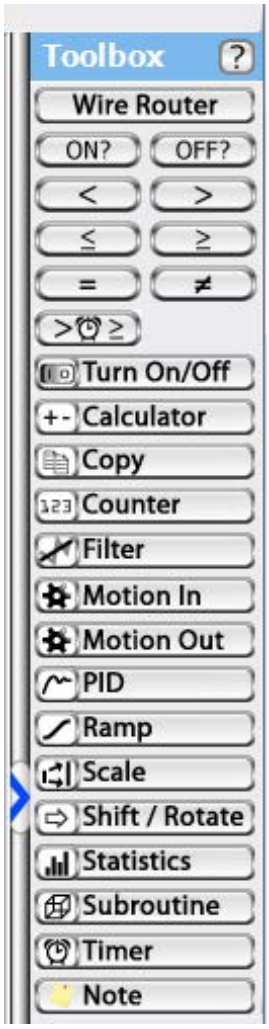
Program Entry

This example program will be one that is vey simple, yet provides you with the basic concepts to build upon. We will implement it in three phases.

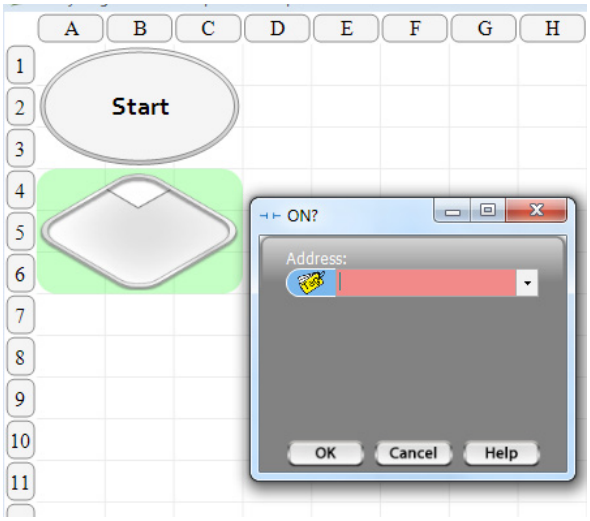
- Phase 1 : When the "doorbell" is active, we will turn on the "lockRelease". That's probably not a good idea in the real world, but its an easy understandable test program.
- Phase 2 : We'll add a timer to it. When the doorbell is activated, the "lockRelease" will be turned on for 5 seconds.
- Phase 3 : We'll add crude temperature control. When the temperature is below 68 degrees, we'll turn the heater on, when it rises above 72, we'll turn it off.

Along the right hand side of vBuilder is the Toolbox. The Toolbox contains all of the program tools necessary to build a program. There are two basic types of tools :

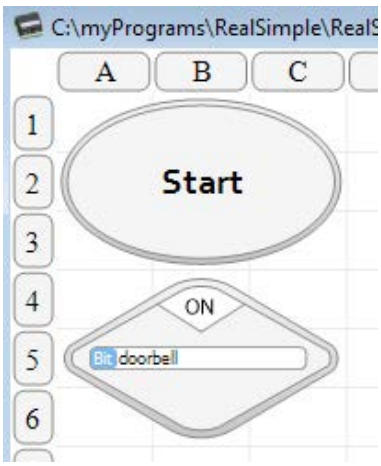
- Decision blocks : decision blocks make a decision whether something is true or not. If it is true, the program flow will follow one branch. If it is false, it will follow another. There are decision blocks for whether a bit is on or off, numeric comparison decision blocks and timer based decisions. The decision blocks are all near the top of the toolbox, below the WireRouter and above the Turn On/Off tool.
- Process blocks : Process blocks do something. That something can be to turn something on or off, perform a calculation, count, filter, control a stepper motor, PID, etc. The Process blocks begin with Turn On/Off and end with Timer.



Lets start our test program by selecting the decison block labeled "ON?". Just click it with your mouse, move over, directly below the Start block and click the mouse again to drop. Your screen should look like the screen shot on the right.



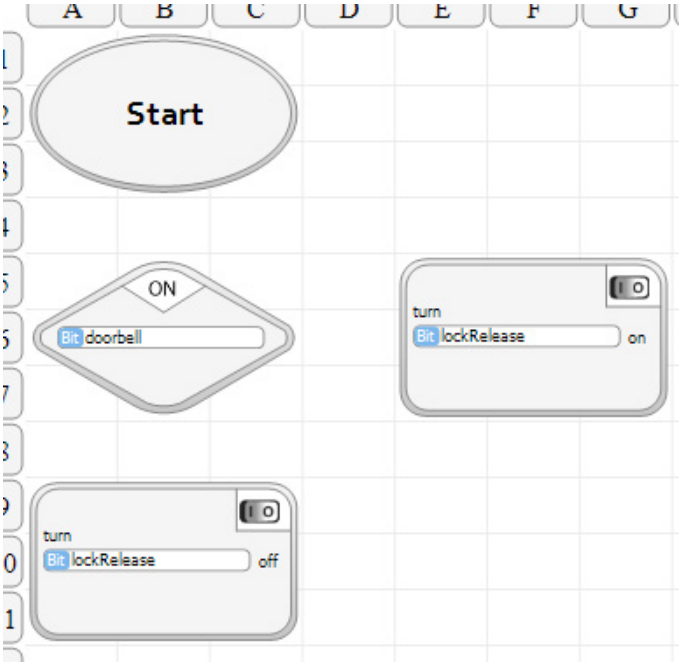
When you place the block, a dialog box will open to allow you to define the decision. If you click the arrow on the right side of the Address box, all of the bits will show up. Select "doorbell", then click OK. Your program will appear as shown below.

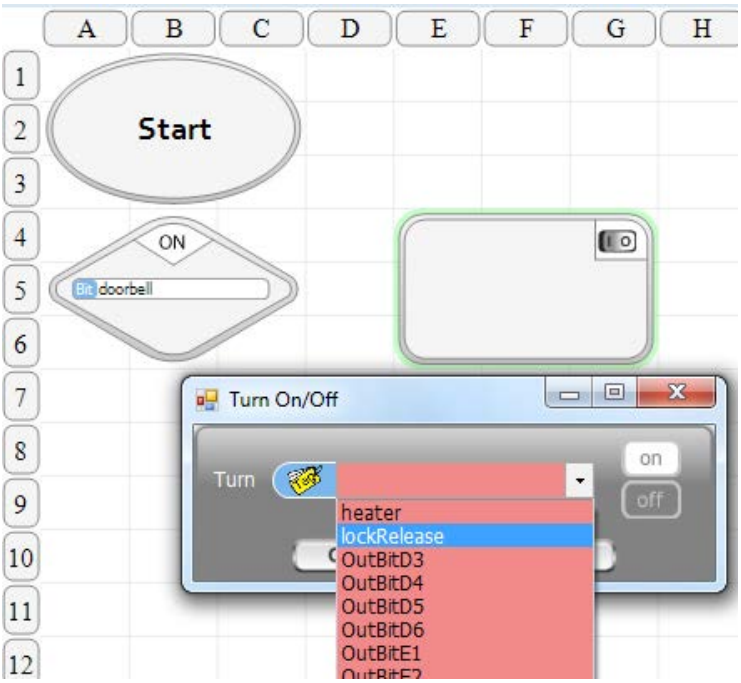


Next, select the Turn On/Off tool, and click to place it to the right of the decison block. Select "lockRelease", leave the On button highlighted and click OK.

Select the Turn On/Off tool again, and place another block below the decision block. Select the "lockRelease" again. This time, select the Off option. Click OK.

Your program should now appear as shown below.

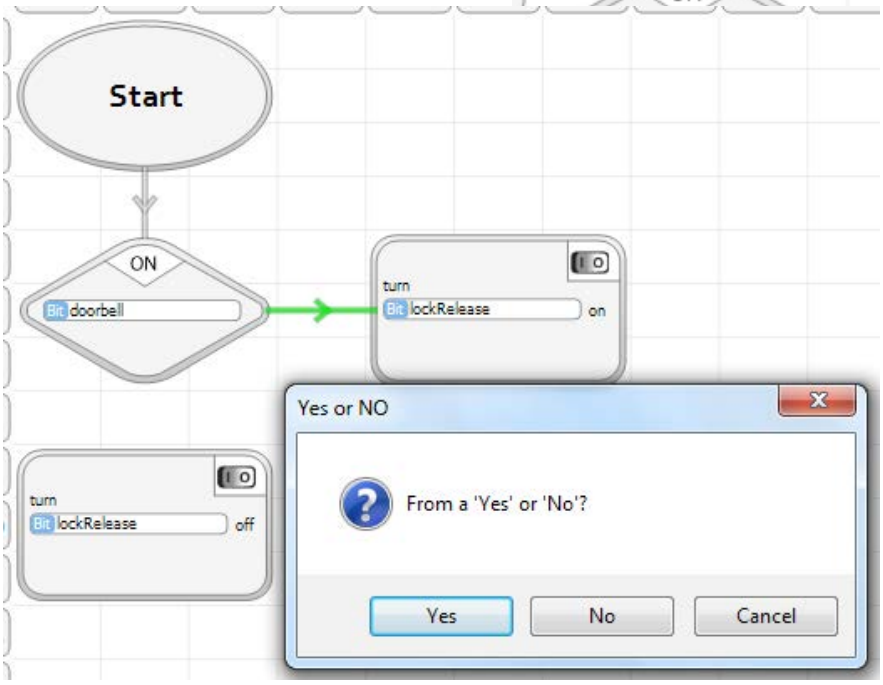




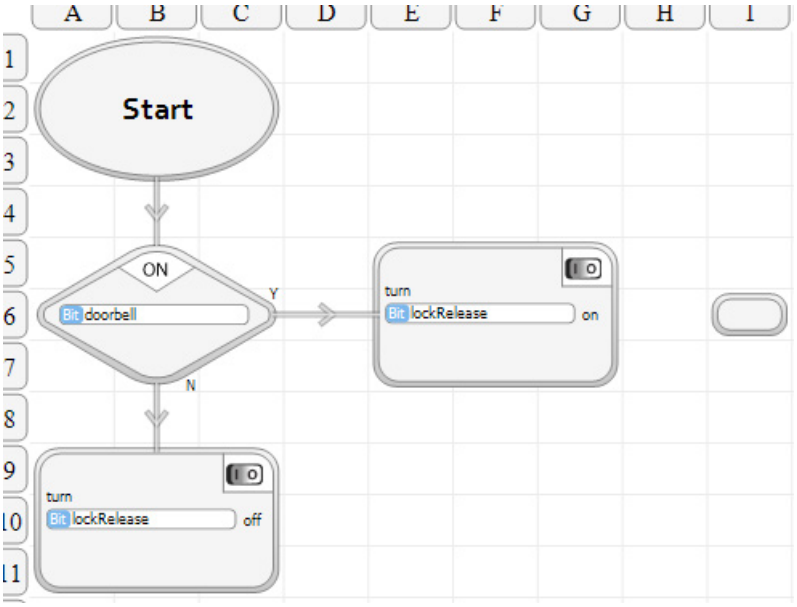
We now need to wire up the program flow. If you move your cursor over the bottom of the Start block, a bubble will appear and turn blue. This bubble is a connection point. When it turns blue, you are in a position to place a flow line beginning at this point. Click on the blue bubble, keep your left mouse button depressed and move to the top of the decision block. When a blue bubble appears on the top of the decision block, release the left mouse button. You will have placed a flow line from the Start block to the decision block.

Place another line from the right hand side of the decision block to the turn on block to its right. When you release, a question box will pop up, as shown on the right. A decision block has two exits. It exists one way if the answer to the decision is "Yes". It will exit another way if the answer is "No". This question block is asking whether this will be the a "Yes" exit or the "No". Click "Yes".

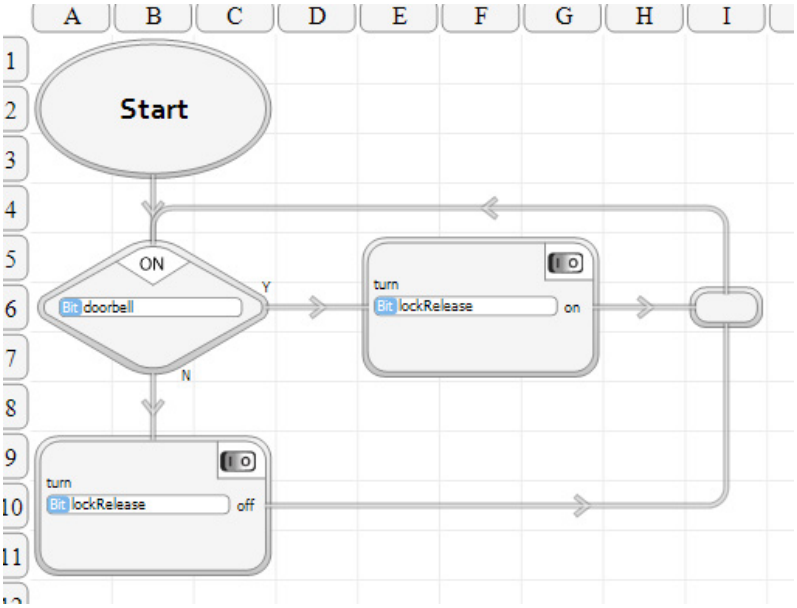
Place another line from the bottom of the decision block to the top of the turn off block.



At this point, you could actually program the PLC and the program would run. In operation, when the flow chart reaches the last block in its flow, the program flow will start over at the Start block - so there is a continuous loop, whether you explicitly complete the loop or not.



Let's explicitly enter the program flow to make everything very clear. Place a Wire Router block to the right of the turn on block. A Wire Router block does not do anything other that provide an attachment point so you can draw a "clean" flow chart - one in which the flow lines don't cross over program blocks.



Connect the rest of the blocks as shown. The full program flow for Phase 1, of this example is now complete. We're ready to try it.

STRONG RECOMMENDATION! The best tool for learning Velocio PLC programming, and doing program development is a Velocio Simulator. They are inexpensive and very convenient. A Velocio Simulator allows you to simulate an application, right at your desk, with switches, potentiometers and LED indicators to simulate actual IO. You can fully develop, debug and have your application programs ready, without waiting to connect to the final hardware. You can simulate operation without any repricussions for a mistake in your program - thereby enabling you to eliminate such mistakes before "going live".

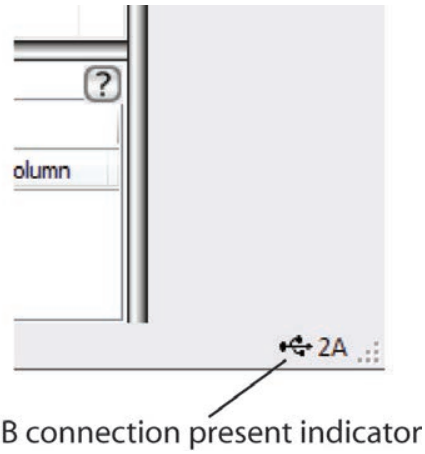
The download and debug portions of all of the examples in this manual assume that you either have a Velocio Simulator, or that you have wired up the equivalent inputs and outputs in order to simulated program operation.

Program Download and Debug

Notice the icon on the top toolbar, that is the Velocio logo.

This icon is the Program icon. When you click this icon, the program will be compiled and downloaded into the PLC.

Before we can program a PLC, we need to make sure that we are connected to the PLC and the PLC is powered on. The required connection is via a USB cable from the PC to the mini USB port on the main PLC (Ace or Branch unit). If this connection is made and the PLC is powered on, you should see a USB present indicator, as shown on the right, in the lower right corner of vBuilder.

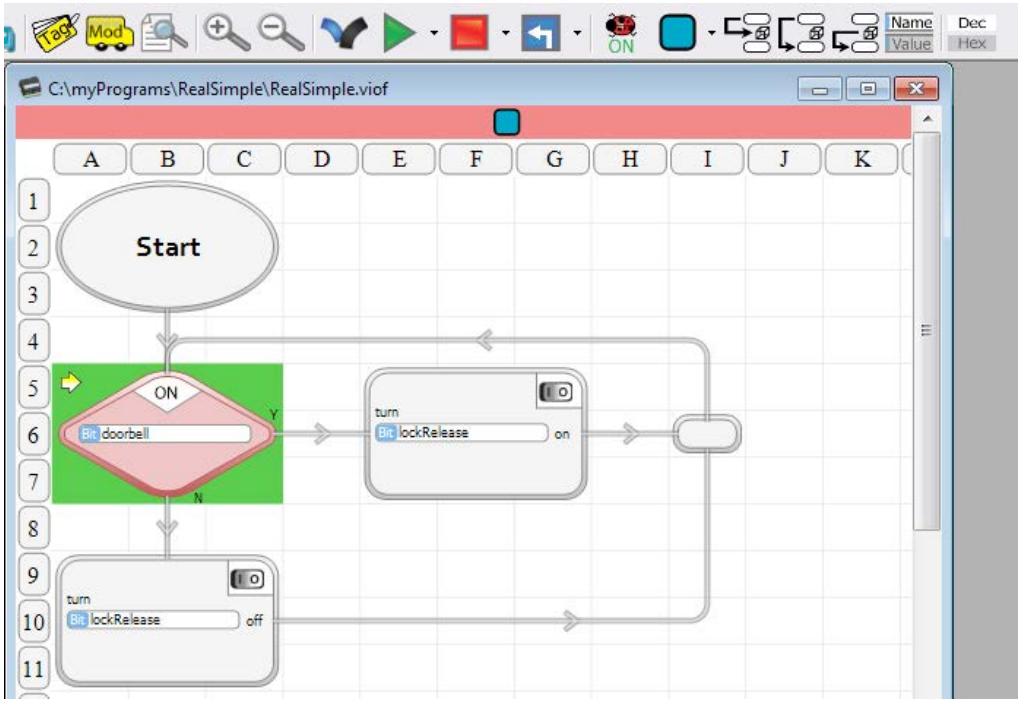


Also notice the lower left hand corner of vBuilder. When you press the Program icon to compile and download, a message saying "Programming" and a progress bar will appear. With such a small program, this may appear only instantaneously, so be watching as you click on the Program icon. When programming is completed, the status message in the lower left hand corner will say "Stopped"

Click on the Program icon and watch the download.

The top tool bar should now appear as shown below.

To illustrate program operation, let's begin by executing one program block at a time. To do that, we can put the PLC into Debug mode. See the icon on the top toolbar that looks like a ladybug. It is currently greyed out and says "OFF". That is the icon that controls whether we are in Debug mode. Click the icon. The toolbar and program window should change as shown on the right.



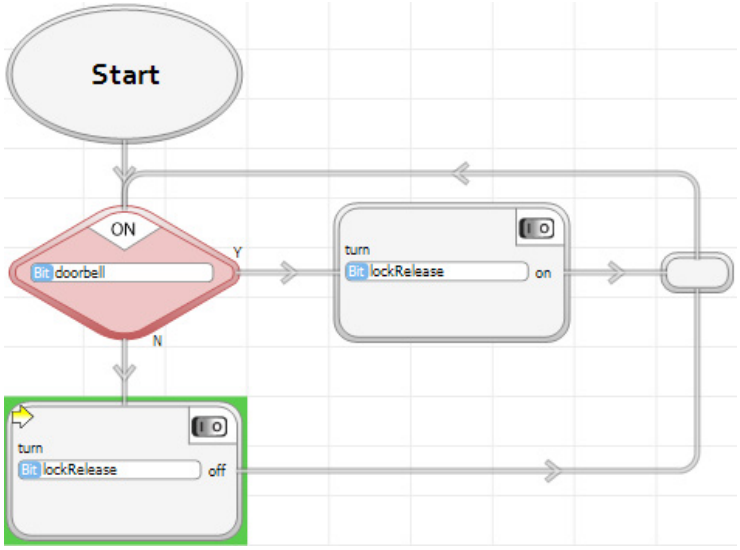
The ladybug Debug icon is now colored and says ON. That indicates that we are now in the Debug mode. Notice that the six icons to the right of the Debug icon have lighted up and become active. The first icon is "Breakpoint". The next three are "Step in", "Step Over" and "Step Out". The fifth one is the "Name/Value" switch. We'll use these icons.

Notice, also, that a red bar appeared across the top of the program window. This indicates that the program is currently stopped. It will change to green, in Debug Mode, when the program is running.

Finally, notice that the first logic block in your program is highlighted with a green background and a yellow arrow pointing at it. This tells you that your program is currently set to execute this program block next.

With the digital input's all switched to off, click the Single Step In icon, shown here.

When you single step the first time, the program window should look like this.



The program block that was just executed checked to see if the "doorbell" (input B1) was on. Since it was not, it took the No branch and went to the turn lockRelease off block.

Single step again. You will see that the program steps back to the decision block. When it executed the turn off block, it turned the output off. Since it was already off, you wouldn't see anything change.

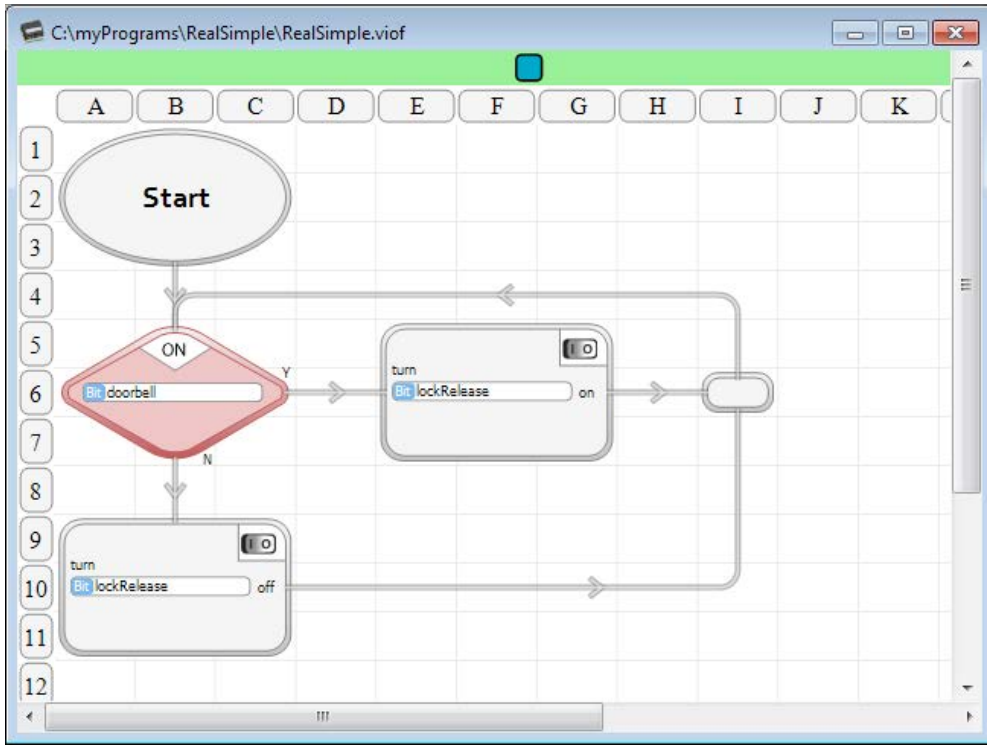
Now, turn the B1 input (doorbell) on. Single step twice. You should see that the program steps to the turn on block and back to the decision. When it executes the turn on block the lockRelease output (D1) will turn on.

Continue to step and play with it a while.

Now, click the "Run" icon, shown here.

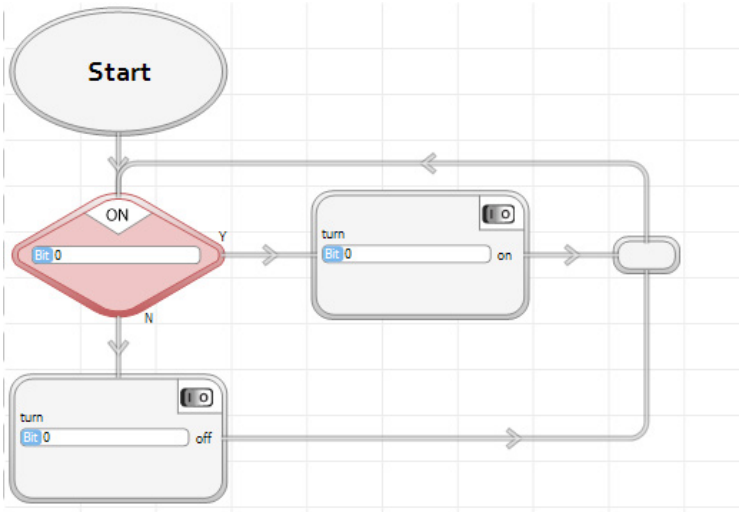
The color bar across the top of the program window will turn green. This indicates that the program is running.

The decision block will turn either red or green. The color indicates the state of the decision. As you toggle the doorbell input, you should see the color of the decision block change back and forth conforming to the current state. A green decision block indicates that it is taking the "Yes" branch. A red one indicates it is taking the "No" branch. At the same time you should see the lockRelease output toggle to reflect the doorbell state.

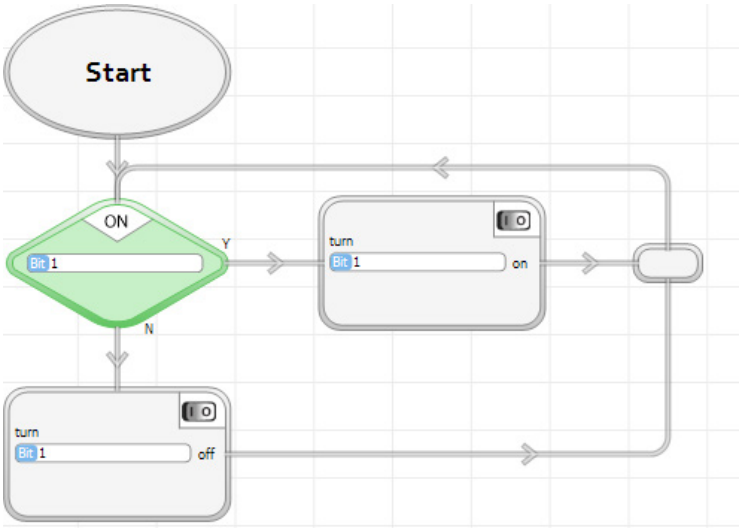


Click the Name/Value icon to put the debug display in the Value mode, as shown.

The program window display will now show the value of the tagnamed variables. Since we are only dealing with bits in this program, the values will be either 0 or 1.



Switch the doorbell input back and forth. You should see the program window switch between the two screen shots on the right, to reflect the states. You'll see the values of doorbell and lockRelease displayed to reflect the current state.



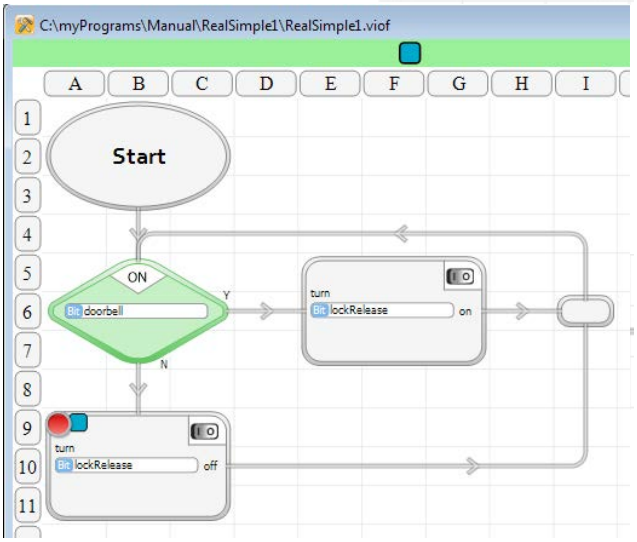
Toggle the Name/Value back and forth and watch what happens.

Leave the doorbell in the on state.

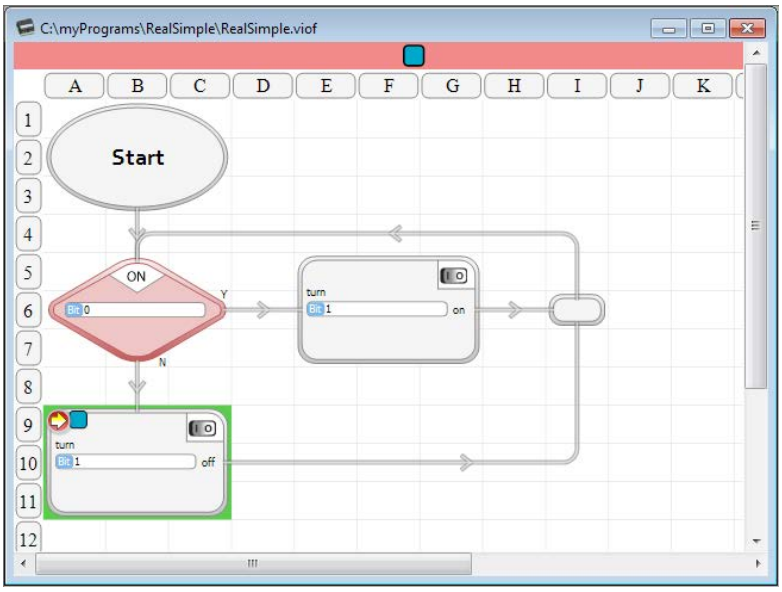
Hover your cursor over the turn off block. You should see a small circle appear in the upper left hand corner of the block, as shown. This is an indication that if you click on the block, a breakpoint will be placed there.

Click on the turn off block.

When you do, a red circle and blue rectangle will appear in the upper left corner of the turn off block. The red circle indicates that this block has a breakpoint. The blue rectangle indicates that it applies to the program in the blue module. Notice that the Setup window in the upper left contains a single blue rectangle. This means that there is only one PLC module in this application, indicated by a blue rectangle. If there were multiple modules in the application, there would be a tree of rectangles, each indicated by a colored rectangle and, for all modules other that the main Branch PLC, a number.



Switch the doorbell input to off. Notice that the program window will change to that shown on the right. Your program has "hit" the breakpoint.



When you switched the doorbell input to off, the program logic sensed it and the decision block directed the program flow to the turn off block. Since there is a breakpoint at that block, the program stops. The color bar at the top of the window changes to red, the turn off block background changes to green, and a yellow arrow appears in the breakpoint red circle in the block. All of this indicates that the program execution is stopped at this block.

Notice that the lockRelease output is still on. The block has not been executed. If you single step once, the lockRelease output will turn off.

If you select the Run icon, the program will start, then stop at the block with the breakpoint. If you switch the doorbell switch to on, then, select Run, the program flow will take the other branch and continue to run, until you switch the doorbell back to off.

To turn the breakpoint off, click on the program block with the breakpoint. The breakpoint will disappear from the block.

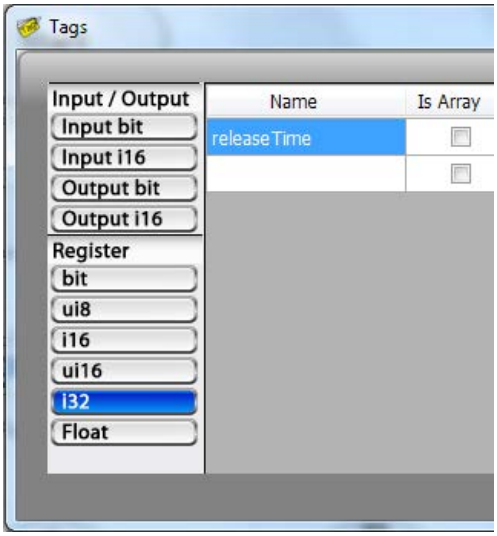
Phase II Program Entry

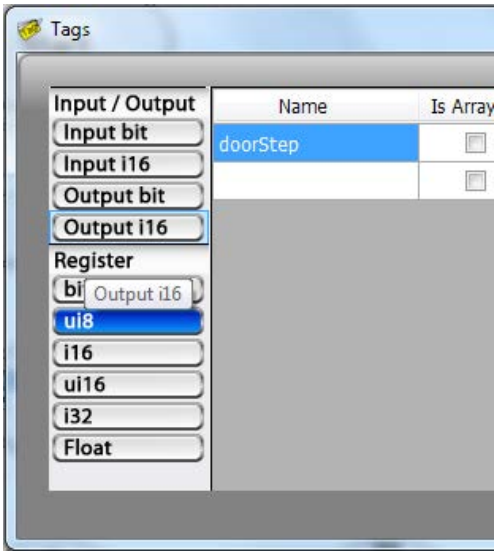
In phase 2, we'll change the program to one that turns on the lockRelease for 5 seconds each time the doorbell is pressed.

Start by creating two new tagnames :

- releaseTime : Timer for timing the 5 second release time. this will be a signed 32 bit integer, since all timers are signed 32 bit integers. To creste this tagname, select i32, under Register, click the box under Name, and type in the name, releaseTime
- doorStep : A variable to keep track of the current state with respect to the doorbell and lockRelease. This will illustrate a very simple "state machine" program. We'll define this as a ui8 (unsigned 8 bit integer number, which can hold a value between 0 and 255). Use the same process to create this name, by first selecting ui8, under Register.

The tagname definitions of these two new variables are shown in the screen shots on the right.

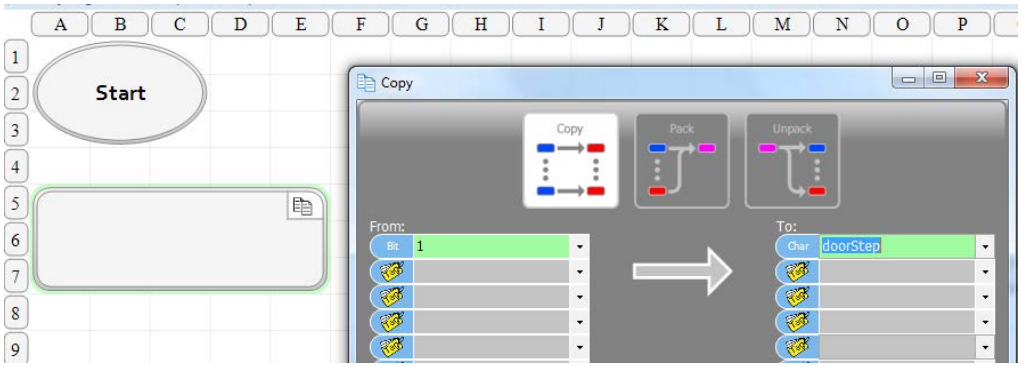




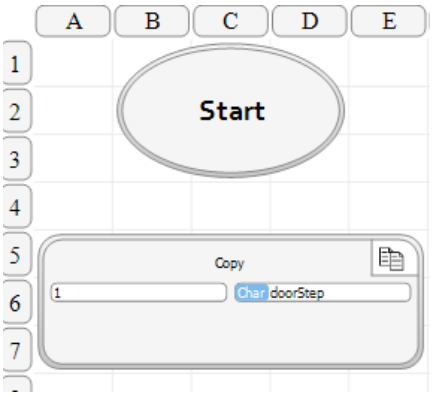
Turn Debug mode off by clicking the ladybug icon and stop the program execution by selecting the red program stop icon.

This phase of the program is going to have a lot more logic blocks, connected much differently than the Phase I example. Since it will have major changes, the easiest way to start is to select each of the program blocks and press the delete key on your keyboard. Delete the whole program (except the Start block). We'll start over.

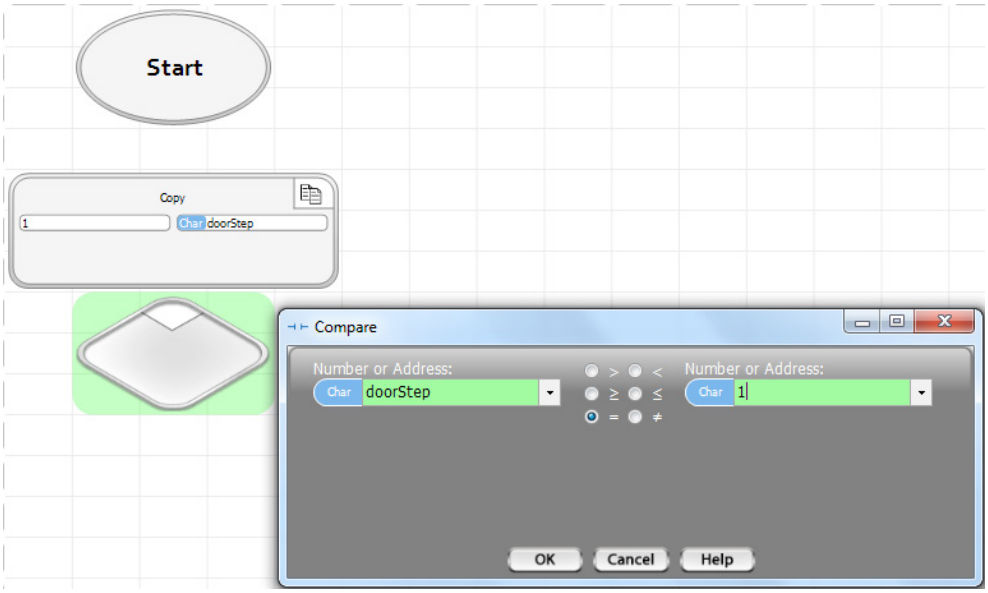
Select a Copy block from the Toolbox and place it below the Start block. In the dialog box, type a 1 in the first From and select doorStep in the first To, as shown on the right. Then click OK.



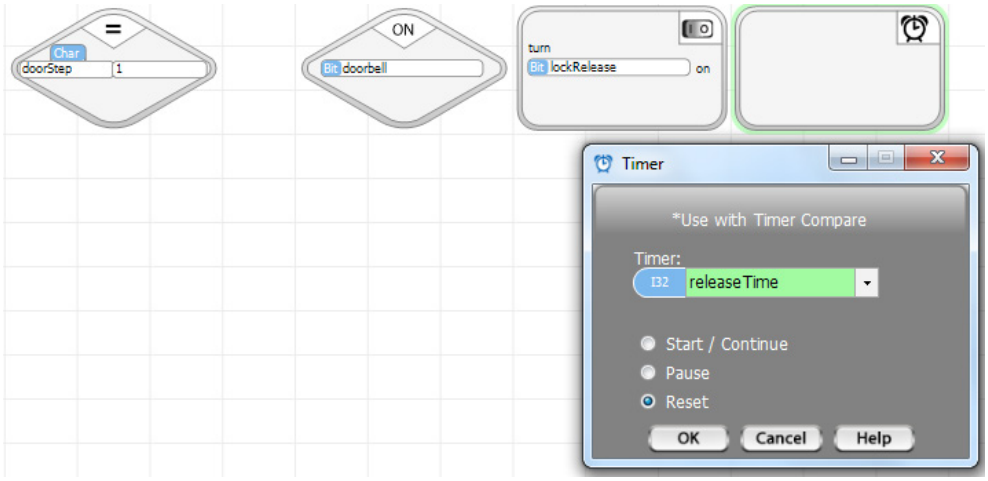
The program should now look like this (after you drag the Start block to the right to line things up. To drag a block, click and hold the block, move your cursor to where you want it, then release).



Next, select the "=" decision block, about a quarter of the way down from the top of the Toolbox, and place it below the Copy block. Select doorStep as the first Number or Address and type in 1 in the second Number or Address. Click OK.



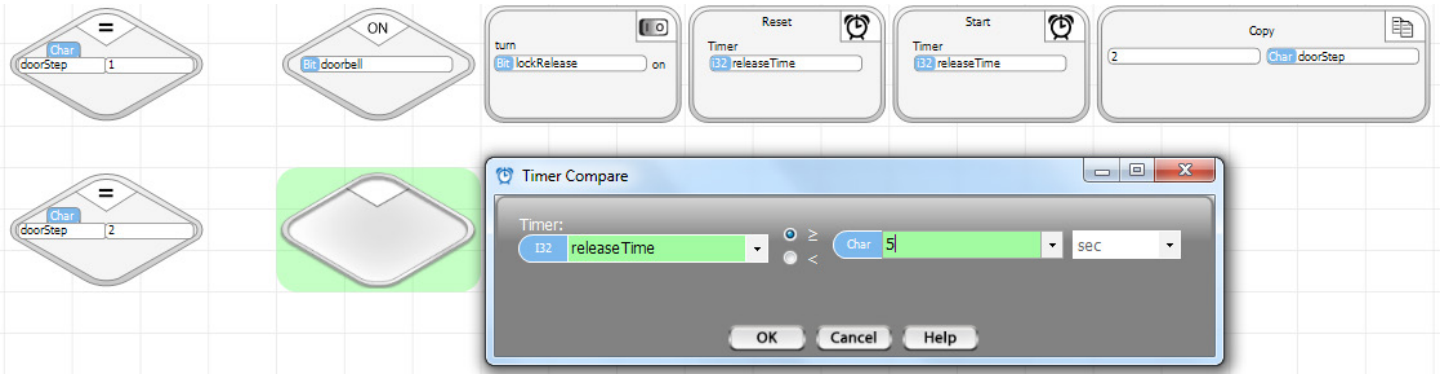
To the right of the doorStep = 1 decision block, place a check for doorbell On, a Turn On block for lockRelease, then a Timer block (near the bottom of the ToolBox). In the Timer block, select releaseTime as the timer we are using and Reset. This block will set the releaseTime value to zero. Click OK.



Place another Timer block next. Select the releaseTime again, but select the Start/Continue option. To the right of this, place a Copy block to copy the value 2 into doorStep. The doorStep 1 line should look like the screen shot shown below.

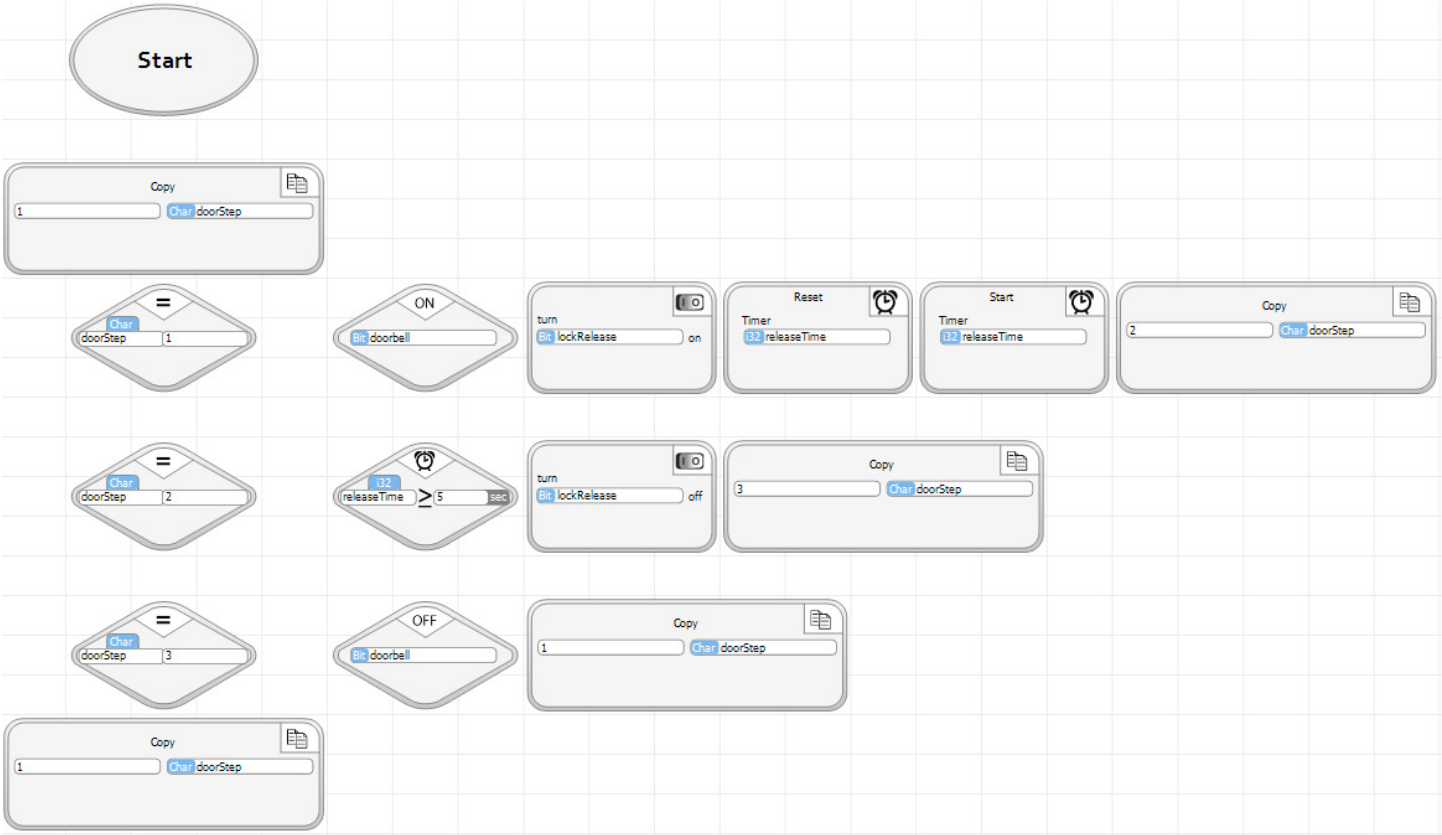
So far, it probably looks a little fuzzy as to where we're headed. That will clear up as we enter more blocks & wire up the flow.

Right below the doorStep = 1 decision block, place another decision block that checks whether doorStep = 2. Just to the right of

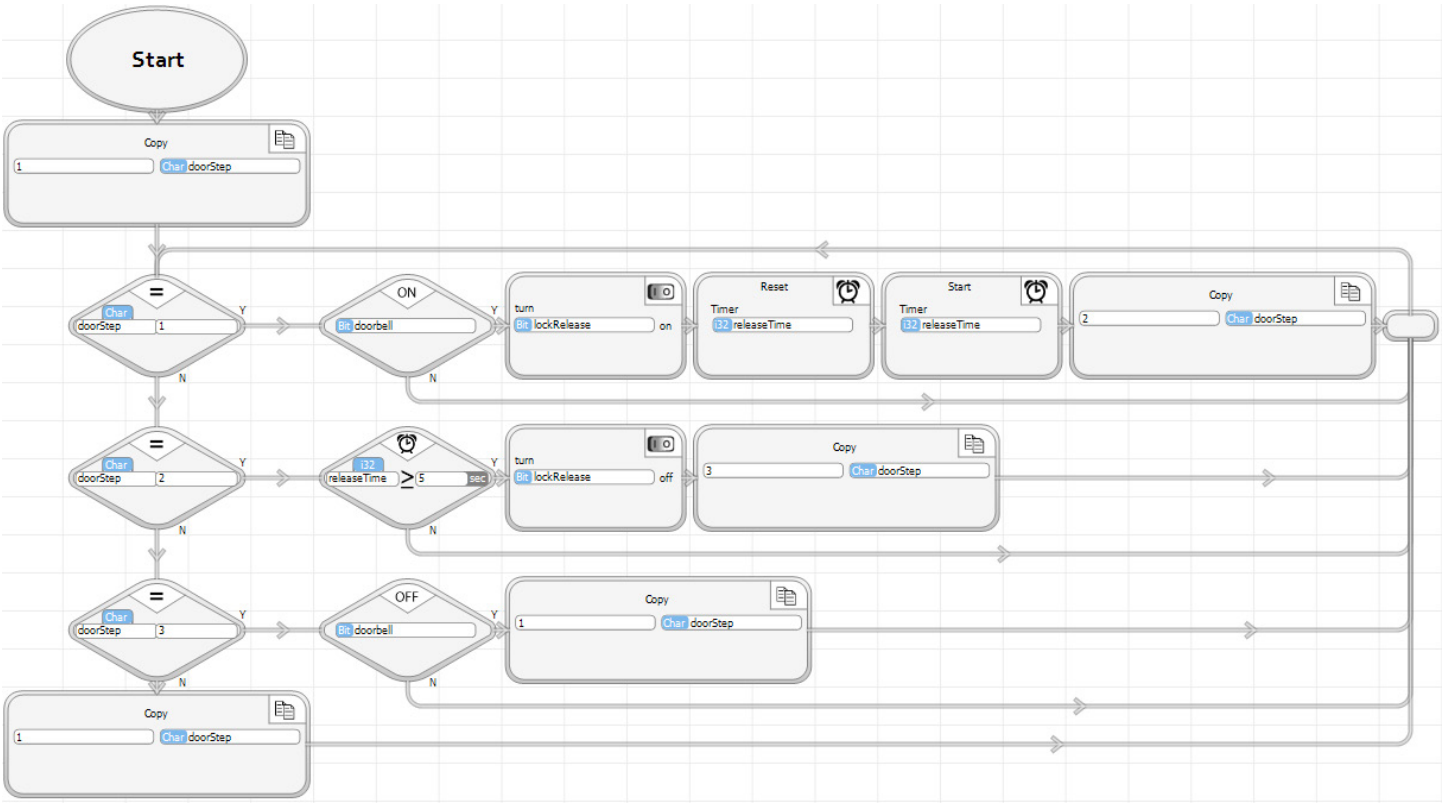


that block, place a Timer Compare block.. The Timer Compare decision block icon is the one with the clock. Select releaseTime, select the greater than or equal to comparison and type in 5. Leave the units selected to seconds (units are selectable as a variety of ranges. Click the down arrow to see the options).

Finish off the program block placements, as shown below.



Finally, put in a Wire Router on the right side and wire up the flow as shown below.



Lets go through how this program should operate. This program is written in "state machine" style. It is highly recommended that you write your programs as state machines also. State machines divide programs into states or steps. In each step, we know our program condition and we know what to look for to trigger an operation and to transition to another step. There is a chapter on state machine programming, later in this manual.

The program :

- Initializes to doorStep 1
- In doorStep 1 : Waits for the doorbell to activate. When the doorbell is activate, the lockRelease is turned on, the timer is started, and the doorStep transitions to step 2.
- In doorStep 2 : Waits for the 5 second delay, then turns the lockRelease off and transitions to doorStep 3.
- In doorStep 3 : Waits until the doorbell is off, then transitions to doorStep 1.
- In any other doorStep : This should never happen. If it does, sets the doorStep back to 1. (This is just a programming practice to ensure the impossible gets handled).

Try programming and running the program. Try putting it in Debug mode, setting breakpoints, single stepping and watching what happens. One thing that you will notice is that once the releaseTime is started, it will continue to time up until it is reset with the next doorbell activation. That is because we didn't put a Timer Pause in the program. Its not a problem, just something to notice.

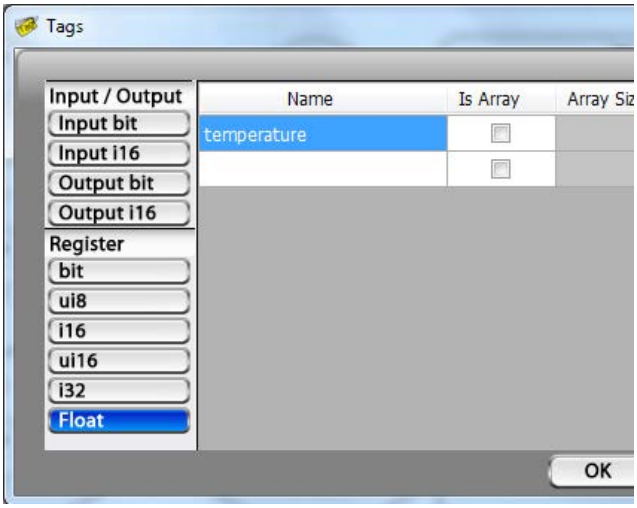
Another thing that you might notice is that the timer value in the Timer Compare block has two less digits than in the Timer Reset and Timer Start/Continue blocks. The Timer Reset and Timer Start/Continue are displaying the time as its raw value. Timers increment once every 10 milliseconds. The Timer Compare is showing the value in the units (seconds) that was selected for the comparison.

When you are finished seeing all that you want to see, turn Debug off and Stop the program execution.

Phase III Program Entry

Phase III will modify your program to add temperature control. A temperature transducer is connected to the A1 analog input. You've already changed its tagname to rawTemperature. Click the Tags icon on the top Tool bar, then Input i16, to see.

The temperature transducer measures temperatures between 0 and 100 degrees. It outputs an analog signal between 0 and 5V to the PLC. There is a linear relationship between the temperature reading and the voltage output.



It would be awkward to directly use the rawTemperature values. Remembering that 2866 is 70 degrees is too much brain clutter. We're going to convert the raw reading to temperature in degrees, and use that.

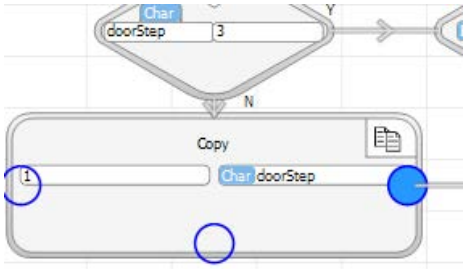
Click the Tag icon, then select Float, under the Register group. Enter 'temperature' as a new tagname.

While you have the Tags dialog box open, look at your Output bits. You should see that you already created the tagname 'heater' for output D2.

Click OK.

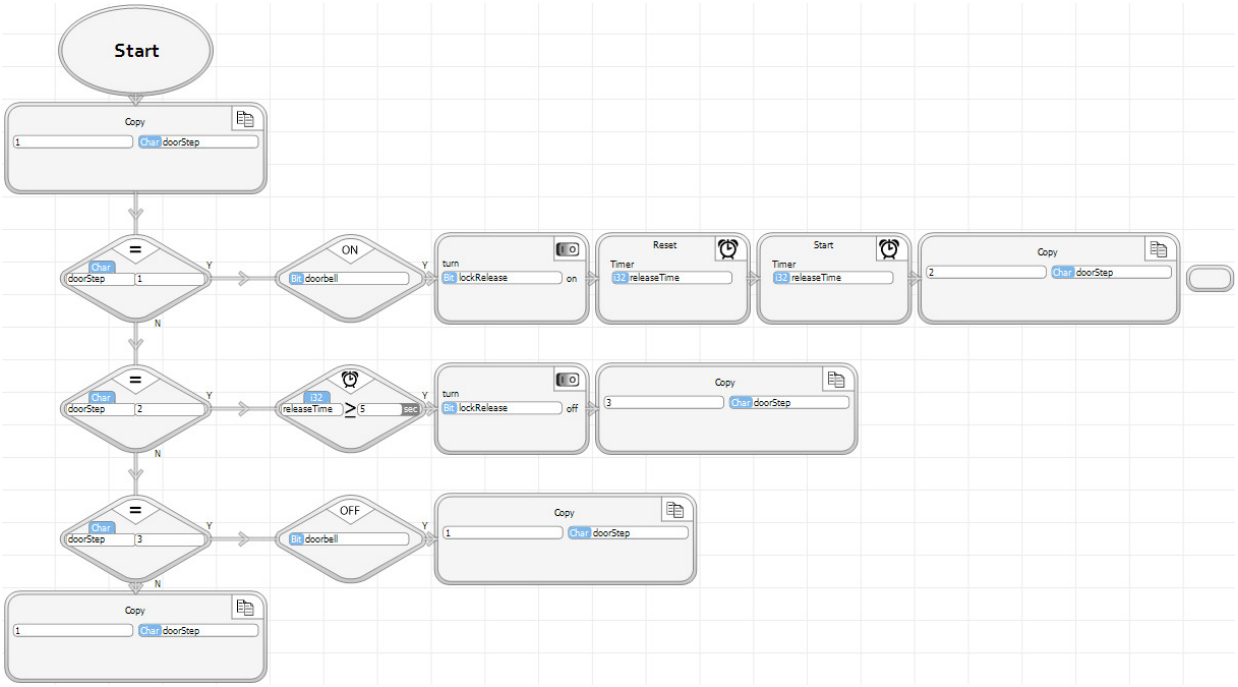
Since we are going to modify the existing program, the first thing to do is to remove some of the existing flow wiring. We'll re-wire the flow after adding some program function blocks to do temperature control.

Click the Debug icon to place it in the OFF mode.

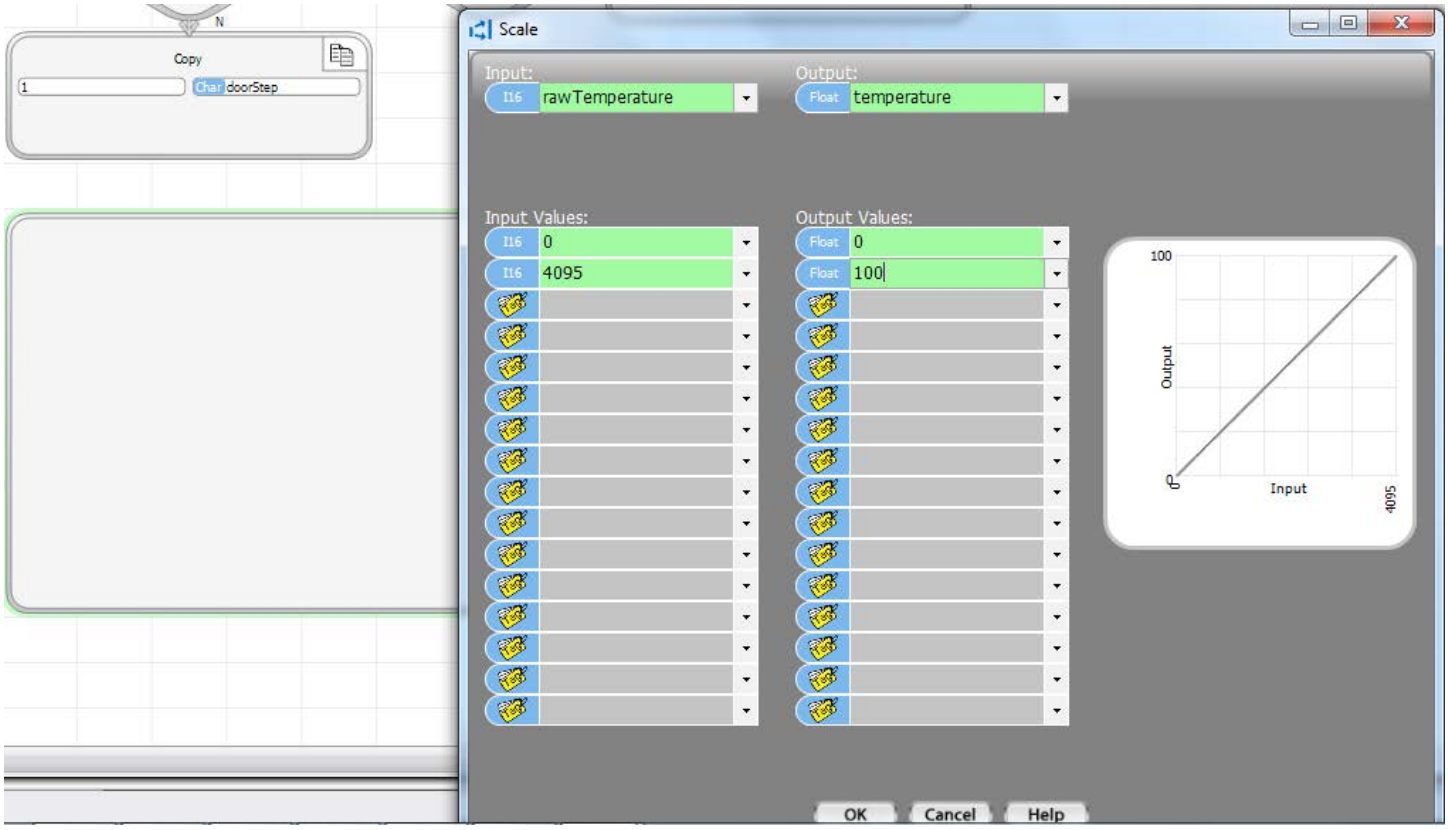


If you move your cursor over the connection point for any flow line, you will see that the connection turns blue. Move over the connection out of the Copy block at the bottom of your program. Click on the blue circle. The flow line out of the Copy block will be deleted.

Delete all of the flow lines into and out of the Wire Router. After you do, your program should look like the screen shot shown below.

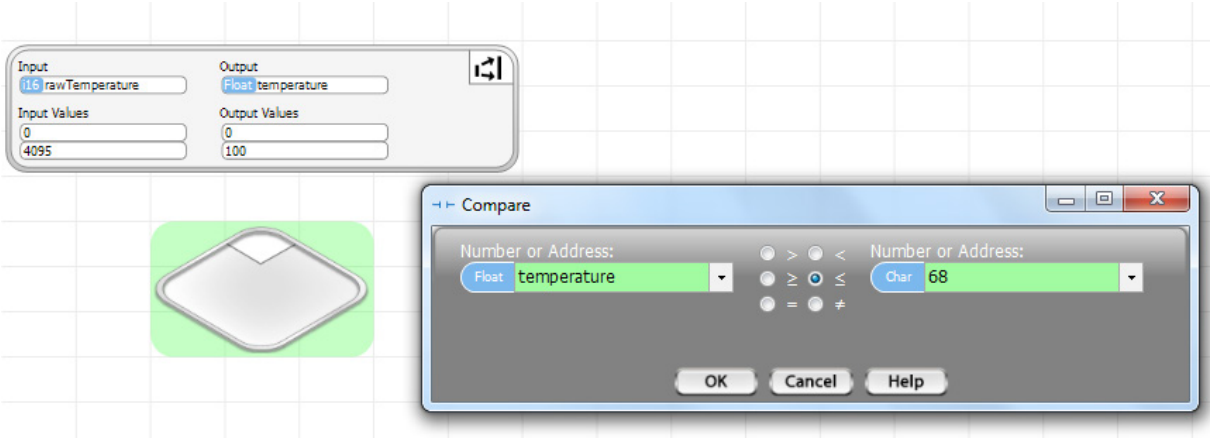


Select the Scale function icon from the Toolbox and place a Scale block below the Copy block at the bottom of your program. In the dialog box, select rawTemperature for the Input and temperature for the Output. Enter the Input and Output values as shown. The graph on the right of the dialog box illustrates what the Scale block will do. It will linearly convert the 0 - 4095 rawTemperature input reading to a 0 - 100 degree temperature value. Click OK.

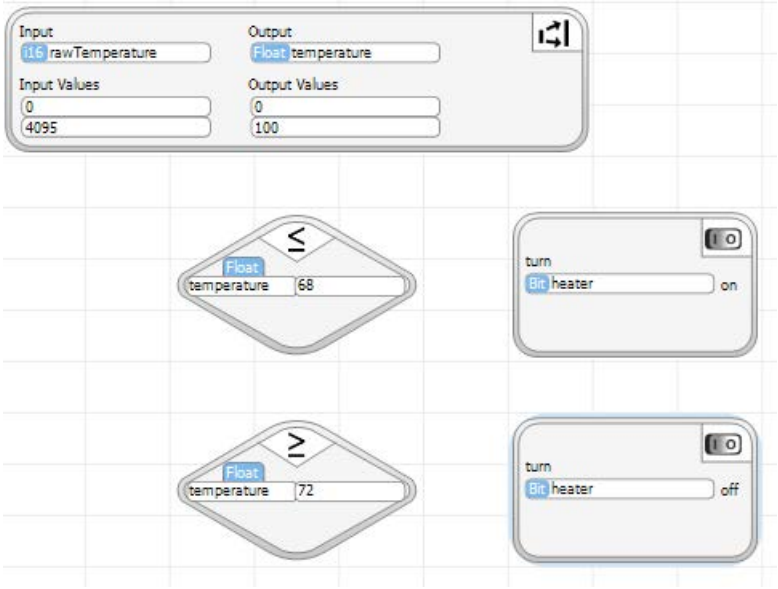


We want to turn the heater on when the temperature is 68 degrees or below, and turn it off whenever it is 72 degrees or above.

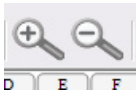
Select the less than or equal to decision icon from the toolbox and place the decison block below your Scale block. Select temperature as the first Number or Address. Type 68 as the second Number or Address. Click OK.



Place a Turn On block to turn on the heater, to the right of the less than 68 decision block. Place a decision block that checks whether the temperature is above 72 below the first temperature decision block. Place a Turn Off block to turn the heater off, to the right of the new decision block. The program should appear as shown on the right.

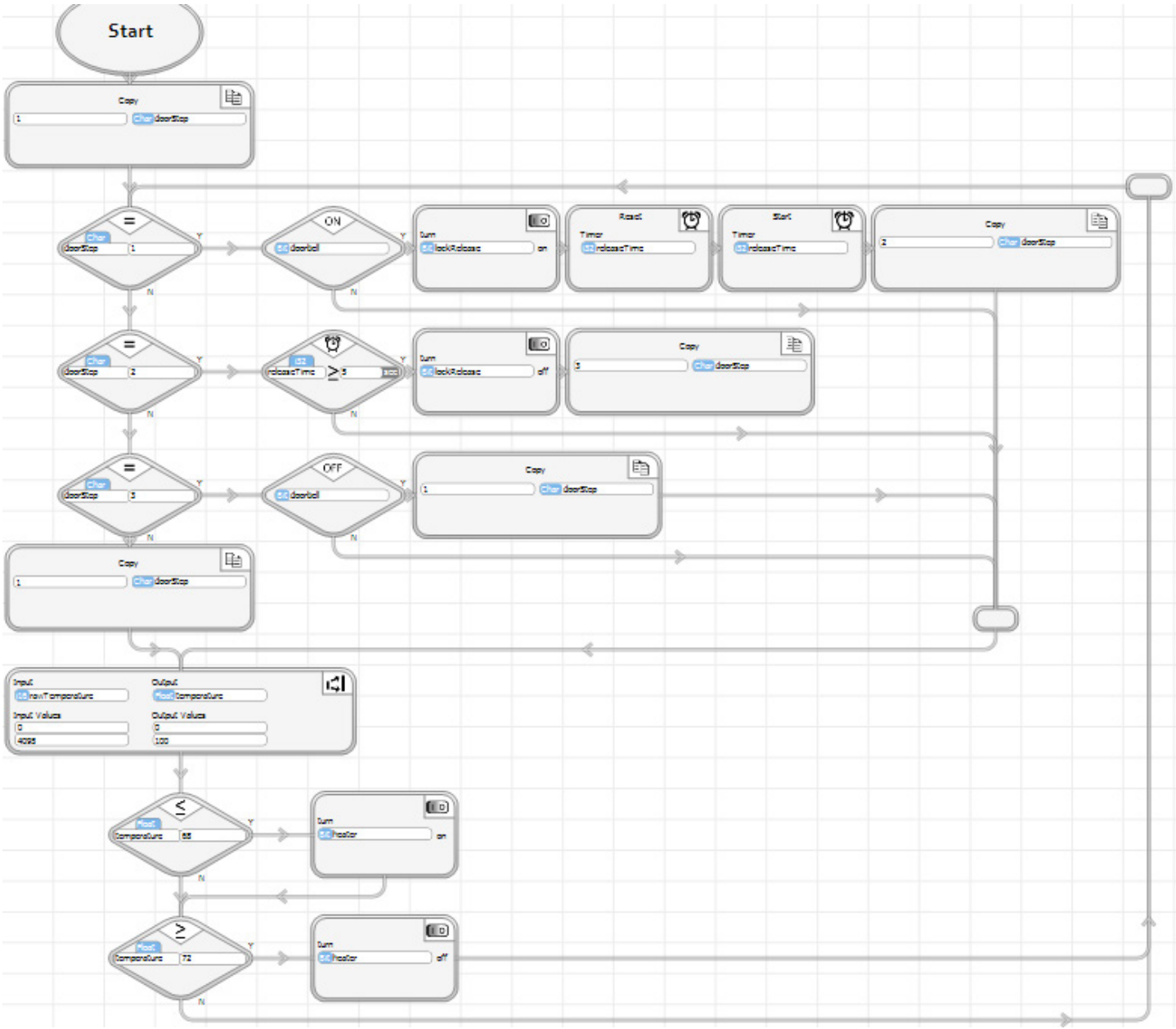


You can Zoom the program window display in and out using the Zoom icons on the top toolbar.



You can also use the View/Hide arrows on the three panes on the right, left and bottom (the blue V).

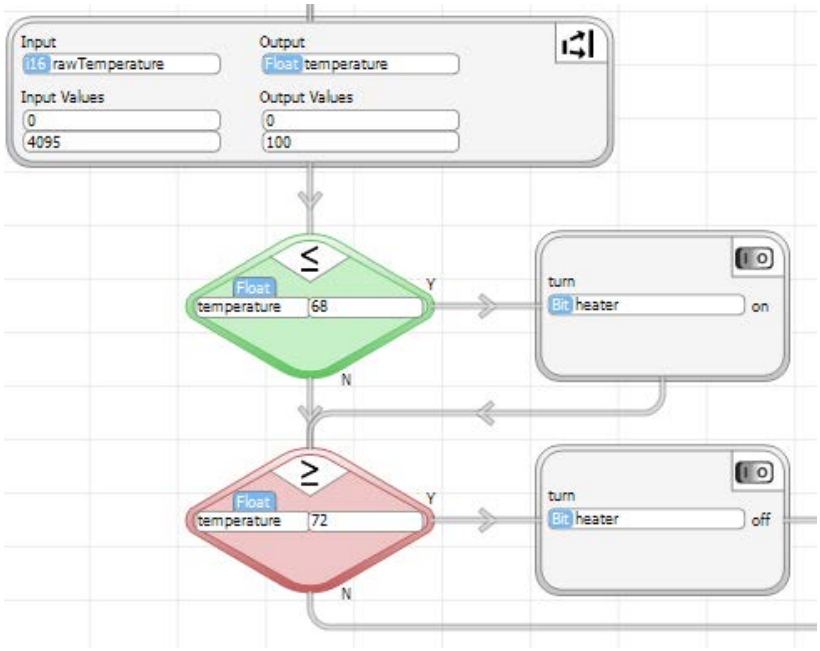
Now add the appropriate Wire Routers and wire the flow as shown below.



With your PLC attached via USB (verifiy that the USB icon is in the lower right corner), press the Velocio logo icon to program the PLC. Select the green Run icon to start program operation and put it in debug mode.

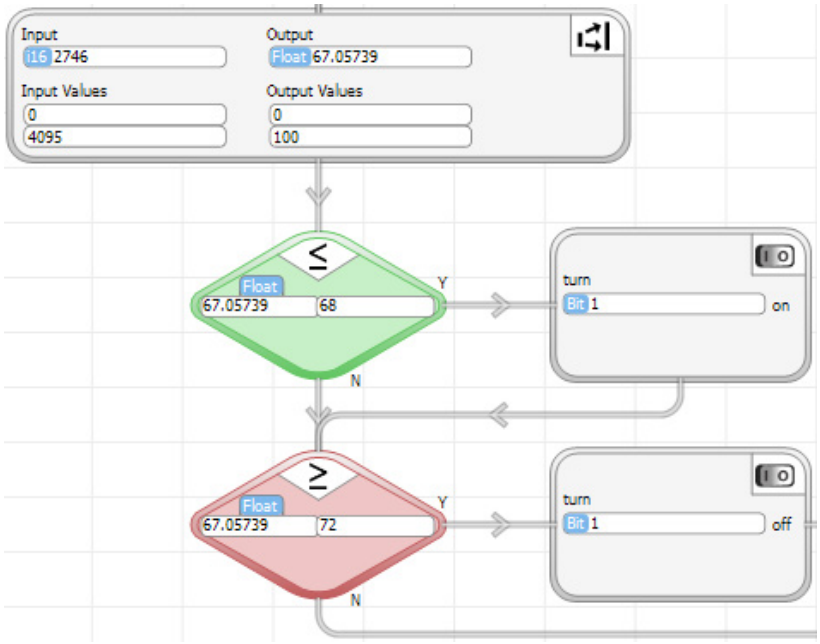
The doorbell/lockRelease portion of your program should work just like it did before. If it does not, check your progam . - especially the flow wiring, since that is the area that changed. Make corrections and reprogram if you see the problem. If its not readily apparent, you may need to employ the debugging skills that you have just learned. Single stepping and/or breakpoints should reveal any errors.

When the doorbell/lockRelease works as it previously did, turn your focus to the newly added temperature control logic. If you turn the A1 potentiometer on the simulator, which we have defined as the program's temperature input, you should see the heater output (D2) turn on and off. When the temperature falls below 68, it will turn on. When it rises above 72, its will turn off. The colors of the decision block windows will change, indicating changes in logic flow, when that happens.



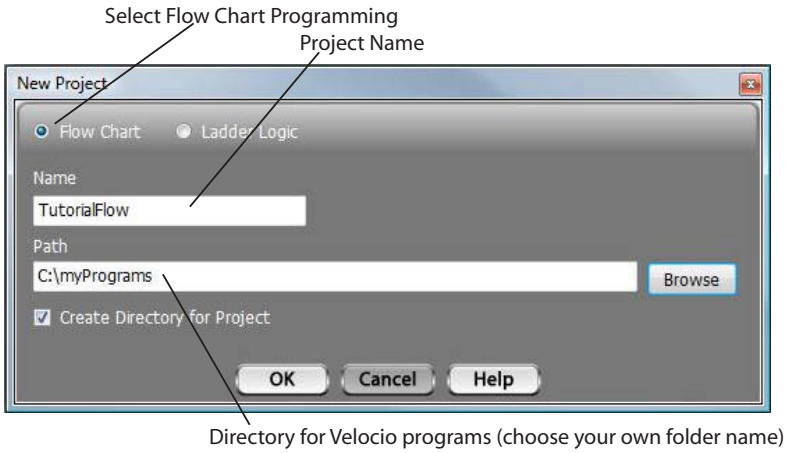
If you change the Name/Value selections (icon on the top tool bar) to Value, you can watch the values change. You can also observe the operations of the Scale block. The screen shot on the right shows that the Scale block converts a rawTemperature value of 2746 to 67.05739 degrees. The first decision block is green, indicating that 67 degrees is less than 68 and the Yes branch will be taken. The second decision block is red, indicating that 67 is not greater than 72, resulting in a No branch.

You can place breakpoints and perform single stepping, just to learn how these tools work and master them for future use. When you've exhausted learning opportunities with this program, you're ready for the next tutorial.

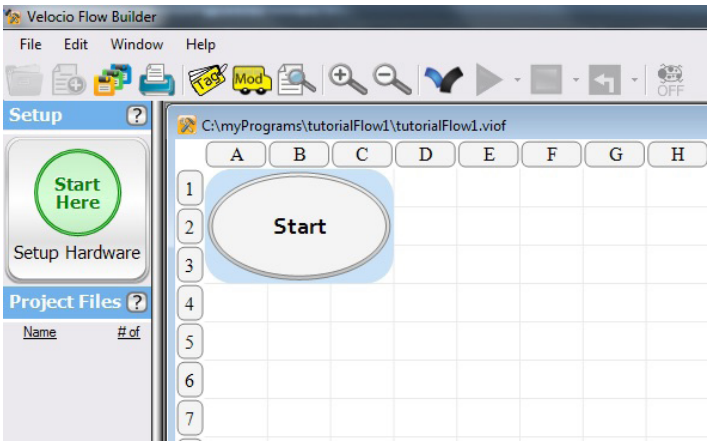


Tutorial Example 2 : Flow Chart Implementation

Start the Flow Chart implementation of our tutorial example by selecting a New project from the File menu. When the New Project window comes up, select "Flow Chart", give it a name and select the directory, on your computer, where you want to store it. Leave the "Create Directory for Project" checked, so Velocio Builder will create a project file, under the defined path, with the name of your program. All of the program files will be stored there.



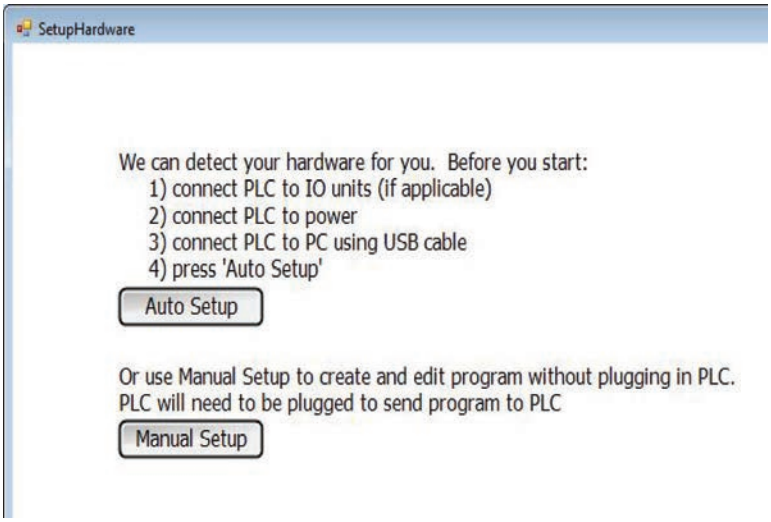
Once you have the chart type (Flow Chart) selected, Name entered and the path for program storage identified, select "OK". The screen will change to look like the one shown on the right.



Select the big green button that says "Start Here".

A window will pop up, which explains your hardware configuration options. As stated in the pop up window, you have the choice between connecting up your target system and letting vBuilder read and auto configure, or you can manually define the target application hardware.

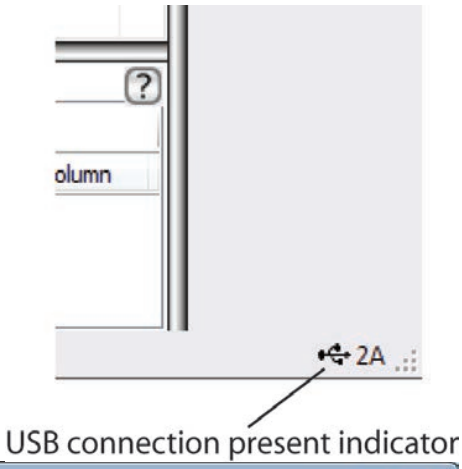
In this tutorial, we will go through both options. However, in order to actually program the PLC, debug and run it, you will need to have either an Ace or a Branch unit.



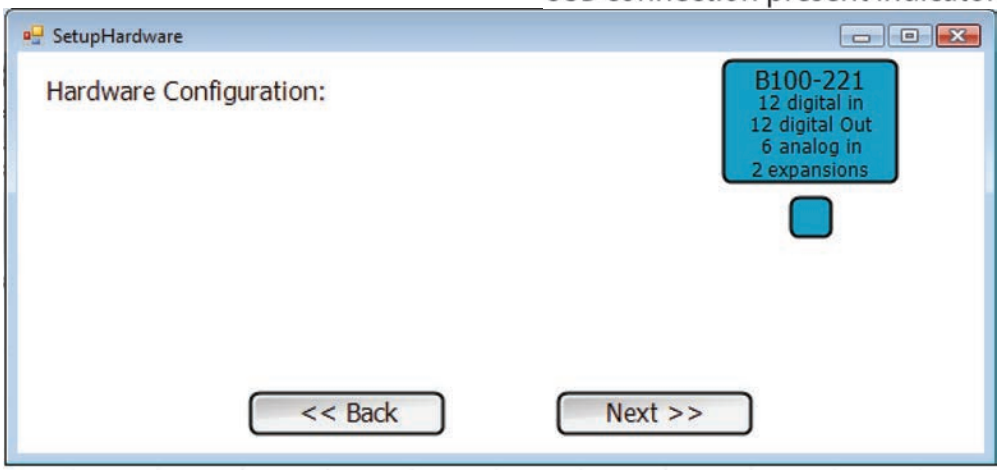
Using Auto Setup

In order for vBuilder to automatically set up your hardware configuration, you must have the Velocio PLC connected like you want the system set up, powered on and connected to the PC.

- Connect up your target Velocio PLC system (in this case, simply either an Ace or a Branch module (no expansion modules are necessary - a Velocio Simulator is ideal)
- Connect a USB cable from your PLC to the Ace or Branch unit
- Power everything on.
- You should see a USB icon, shown on the right, in vBuilder's lower right hand corner. This is an indication that a Velocio PLC is connected (and on) to the PC.
- If everything listed above is OK, select "Auto Setup" in the "Setup Hardware" window.



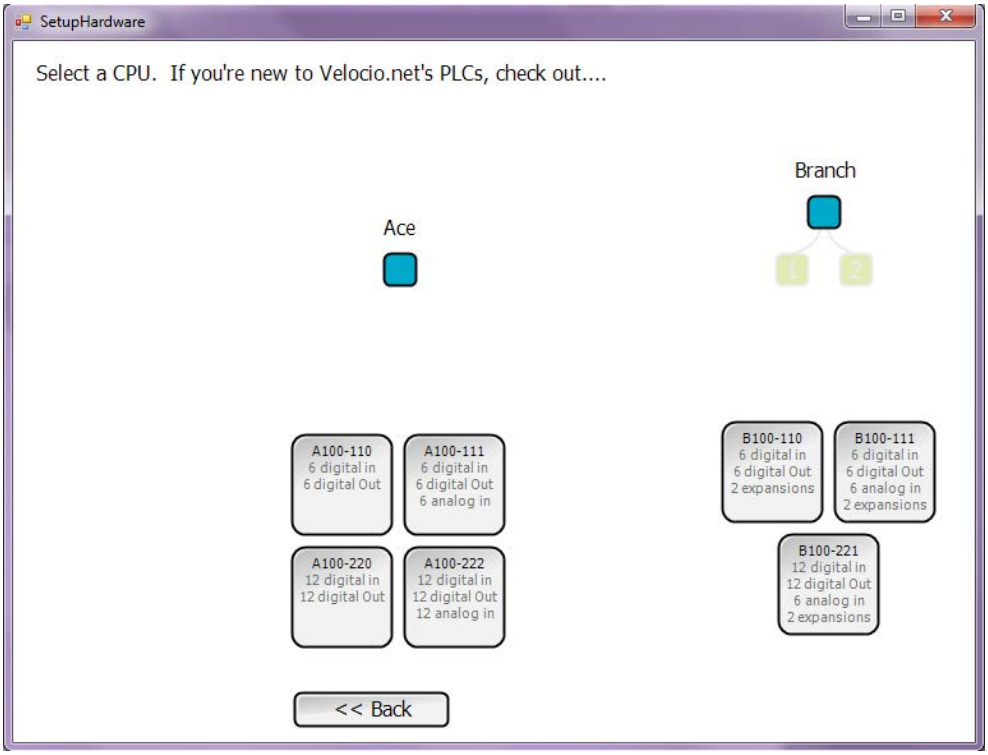
The "Setup Hardware" window should change to display something like what is shown on the right. In this case, it shows that it is configured for a Branch module that has 12 digital inputs (2 ports), 12 digital outputs (2 ports) and 6 analog inputs (1 port). It also has two vLink expansion ports - with nothing attached.



This tutorial can be implemented on any Branch or Ace that has at least 6 digital inputs and 6 digital outputs.

Using Manual Setup

If you don't have the PLC hardware available, you can set the system up manually. Start by selecting "Manual Setup" from the "Setup Hardware" window. The Setup Hardware window will change to something like the image shown on the right.

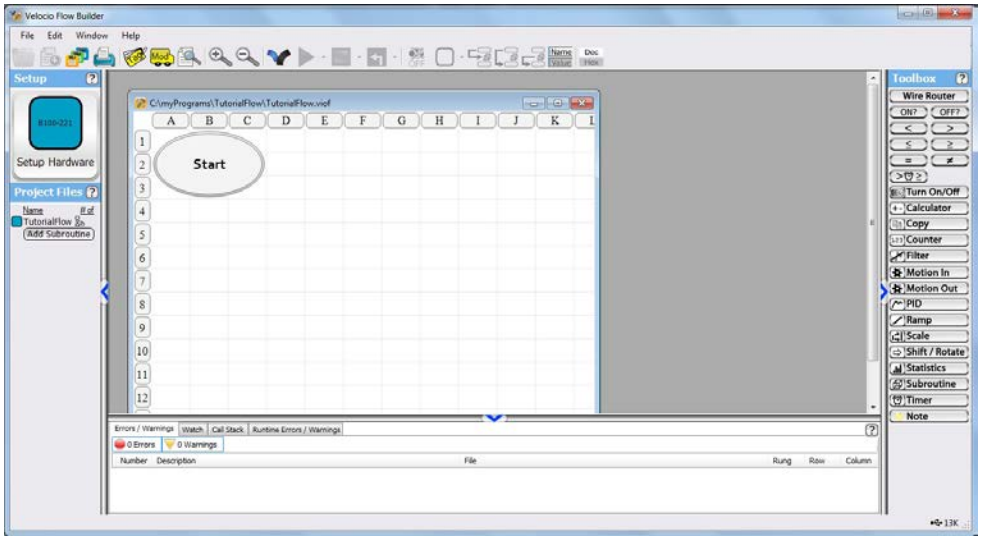


The first step is to select the PLC main CPU. This will be either an Ace, or a Branch unit. The selections are the labeled, gray squares below the Ace & Branch icons. These selections identify both the type (Ace or Branch) and the associated IO configuration. Select the one that applies to the application. In this tutorial example, any of these units will work. The configuration that you select will turn blue, as shown.

After selecting the PLC hardware for the main PLC,, you must then select "Next" to continue with the configuration of expansion modules, if any. After all of the PLC modules are selected, there will be additional screens, which allow you to define whether any Embedded Subroutines are to be placed in Branch Expansions (if you configure Brank Expansions) and for Stepper Motion and High Speed Counter signal. We won't be doing any of that - just click the Next & finally Done.

Tutorial 2 Flow Chart Program Entry

When the hardware setup is defined, your screen should look like the image shown on the right. The Setup is shown on the left side, next to the top left corner of the program entry screen. It shows that we're configured for a Branch with 12 DI, 12 DO and 6 AI. The area just below the hardware setup is the list of project files. The main file is automatically created with the name we defined when we created the new project. In this case it is called "TutorialFlow".

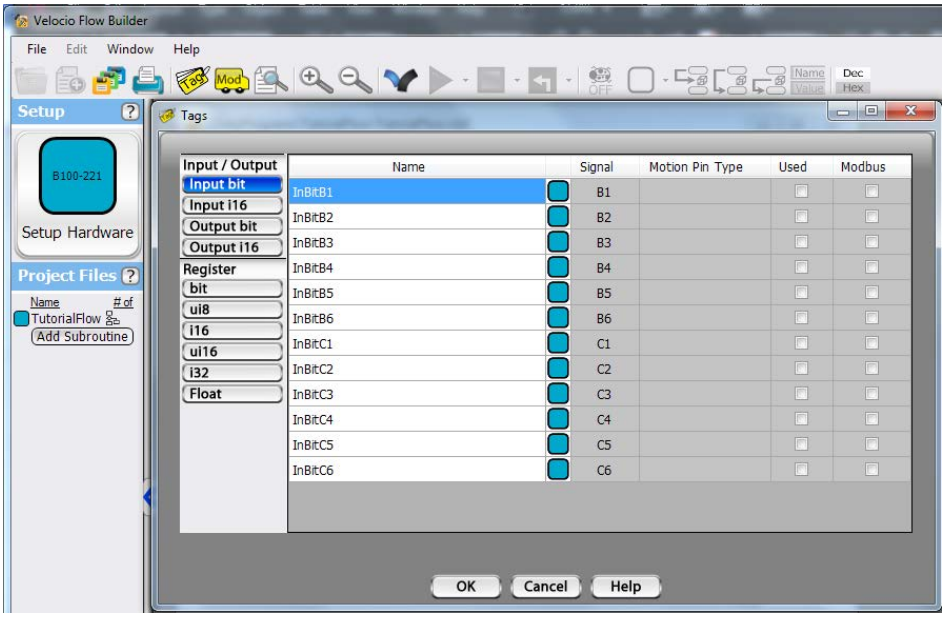


Since we selected flow chart programming for the main program, a flow chart grid is created with a "Start" block. The letters across the top and the numbers along the left side identify "grid blocks" in the program. Move your cursor to various points in the programming grid and look at the reference in the lower right corner (which says 13K on the snapshot above). Notice that wherever you move the cursor, the indication at the lower right will indicate the row (1 thru ...) and column (A thru ...) location of the cursor.

Next, take a look at the tags. All data in a vBuilder program is referenced by meaningful tag names. Tagnames are names that you choose that are meaningful to your application. You might name a digital output that controls a recirculation pump, "recircPump1", or an input connected to a limit switch that indicates the "home" position, "home". That way, in your program, your logic is very clearly defined.

Click the icon labeled "Tag". The tagname window will open, as shown on the right. Click the buttons on the left side of the window to bring up tags of various types. The types are data types :

- bit
- ui8 (unsigned 8 bit integer)
- i16 (signed 16 bit integer)
- ui16 (unsigned 16 bit integer)
- i32 (signed 32 bit integer)
- Float (floating point)

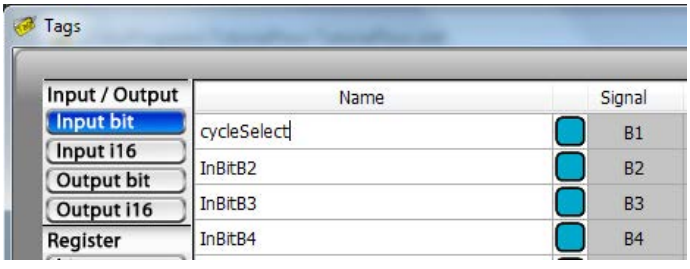


To further clarify the value ranges that the data types can hold :

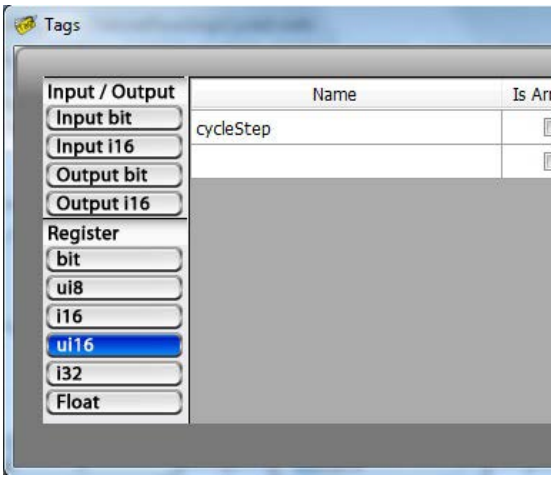
- bit : 0 or 1
- ui8 : 0 to 255
- i16 : -32,768 to 32,767
- ui16 : 0 to 65,535
- i32 : -2,147,483,648 to 2,147,483,647
- float : any floating point number to 32 bit precision.

There are also a few special special case selections for IO for some of these types, listed for convenience, such as input bits.

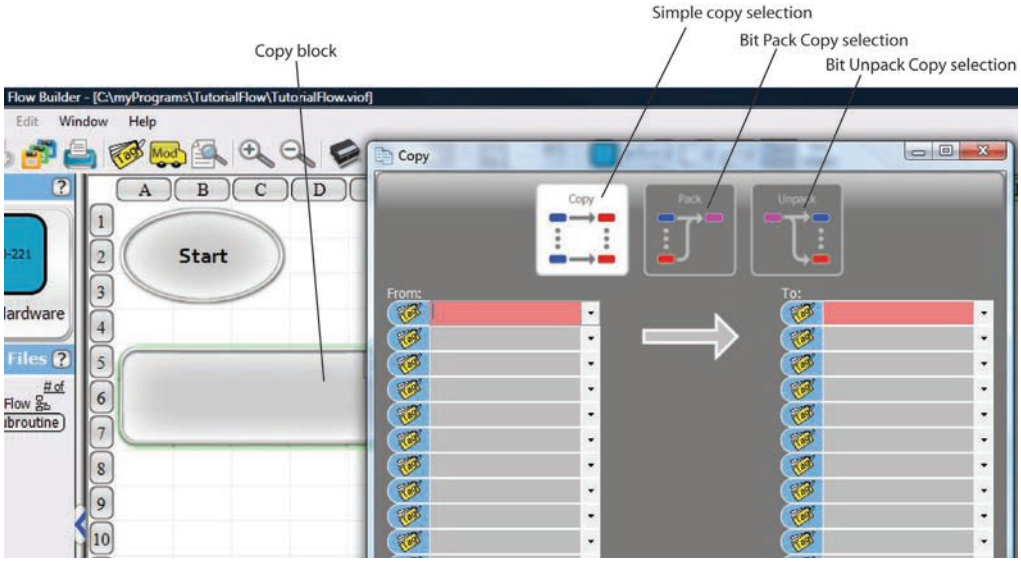
Since this is a new program, most of the tag types show an empty list, i.e. no tags of that type defined. When you configure your hardware definition (which we just did), default tag names are created for the IO. The figure shows the default names for the input bits. Select the name for the first input bit, and change the name to "cycleSelect", as shown. Leave the Output bit names as the defaults, but take a look at them.



Create an unsigned 16 bit integer variable, named "cycleStep" as shown on the right. Enter a second unsigned 16 bit integer, named "outStates", then select "OK'. The Tag window will close. You can select it, view and edit any time.

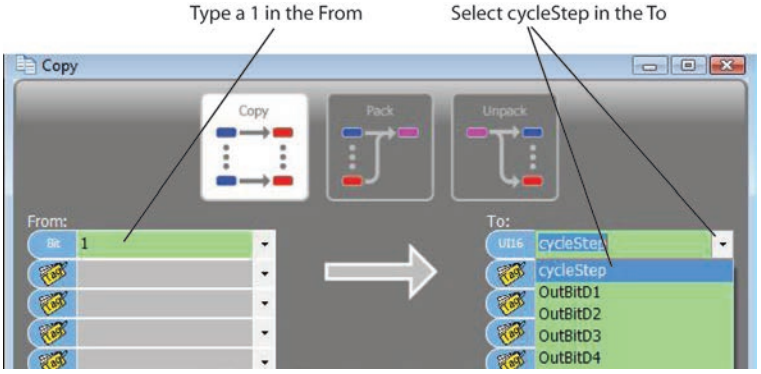


Move your cursor over to the "Copy" icon in the toolbox on the right. Left click and hold on the Copy icon, move under the start block, then release the block. [Alternatively, you can click & release the Copy icon to select, then move the cursor to where you want to drop it, and click and release again]



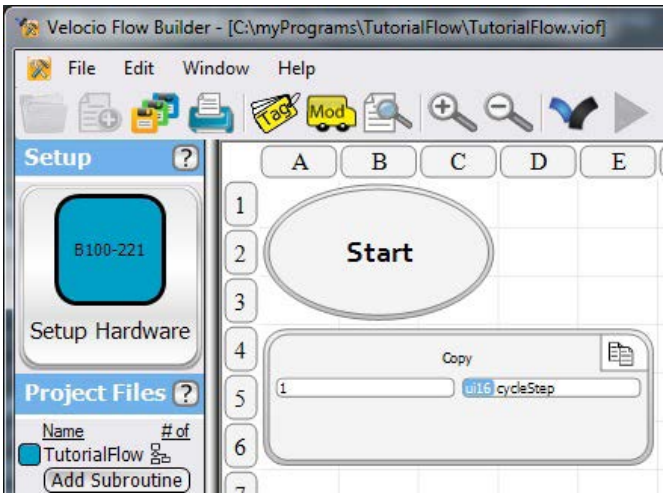
Notice that the Copy program block is placed where you released it. A Copy dialog box will also pop up. A dialog box provides the means for you to define exactly what the block will do.

Take a look at the top of the dialog box. You see three selections. The copy selection is highlighted. The Pack and Unpack options are grayed out. The simple Copy selection allows you to copy data into up to 16 tagnamed variables from other variables or constant values. The Pack copy performs a copy of up to 16 individual bits, into an integer. The Unpack copy moves selected individual bits from an integer into bit tagnames. If you click on Pack or Unpack, you will see that the selected option becomes highlighted, the other options are grayed out and the dialog box selections change.

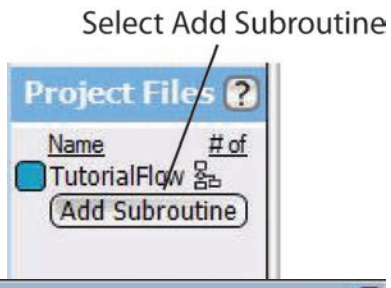


Enter '1' in the first "From" box and select "cycleStep" from the drop down list for the first "To" box., as shown on the right. Select "OK" at the bottom of the dialog box.

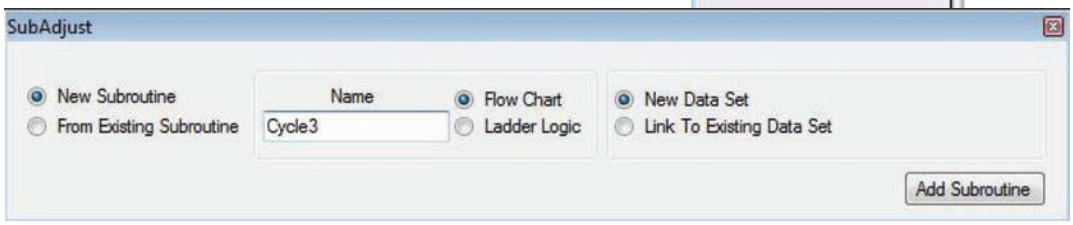
The program will look like the illustration on the right.



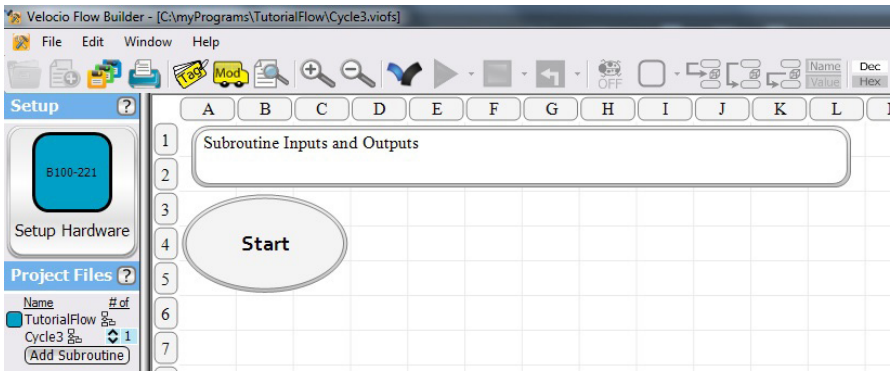
The next thing we need to do is create our subroutine. Start by selecting Add Subroutine on the Project Files list on the left.



A window, like that shown, will pop up. Select New Subroutine, give it the name "Cycle3" and select Flow Chart, then select Add Subroutine.

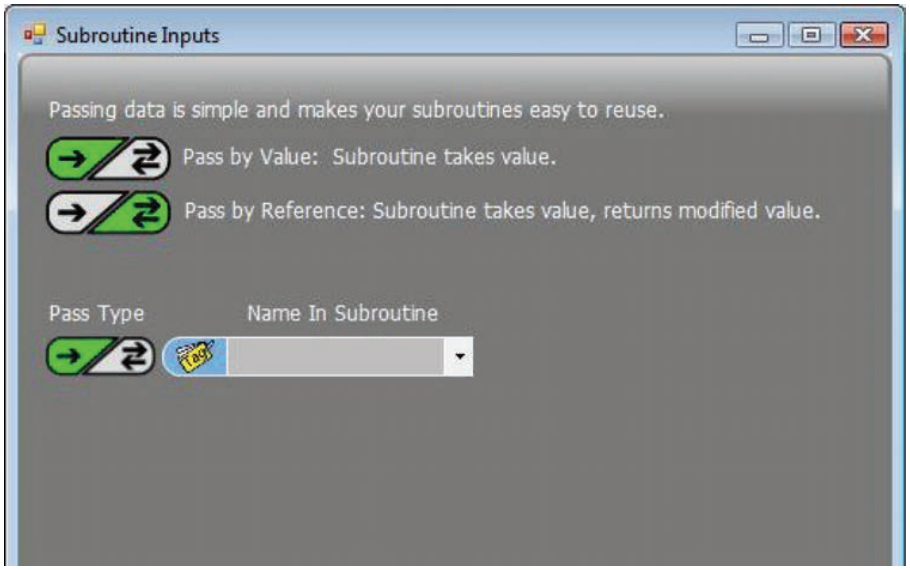


The subroutine window will come up looking like the illustration on the right. Notice that the subroutine, Cycle3, is listed in the Project Files list with a '1' beside it (indicating one instance).

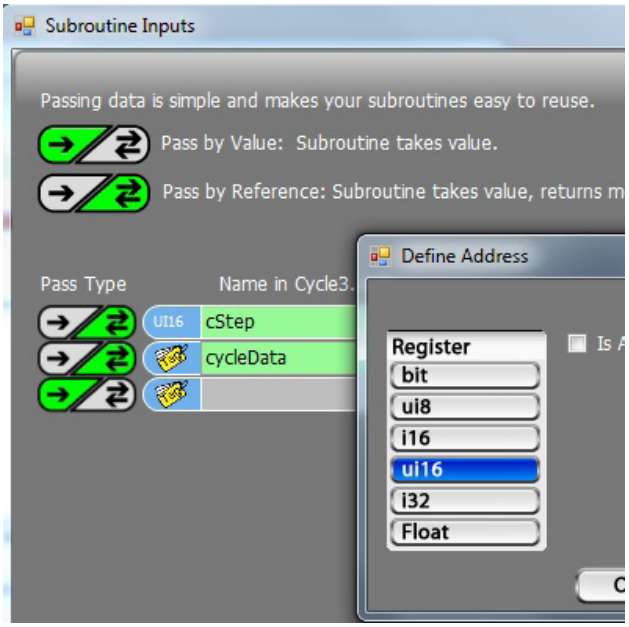


Click in the block labeled "Subroutine Inputs and Outputs" to enter the parameters that are passed in and out of the subroutine.

Take a look at the Subroutine Inputs dialog box. Passing of parameters is explained at the top. There are two pass options for passing to a subroutine object. Pass by value, indicated by a single arrow to the right or in, passes a numeric value into the subroutine variable that you define. Passing by reference, illustrated by the opposing arrows indicating in and out, passes a reference or handle to a variable in the calling routine. In actuality, data and references are passed into an object subroutine, no data is actually passed back out. By using a reference that is passed in, data in the calling routine can be accessed and changed. Changing data passed by reference is the equivalent of passing data out.

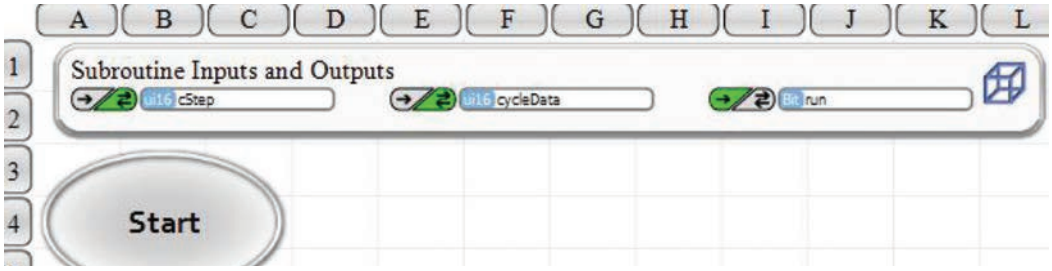


For this tutorial example, define two variables that are passed by reference and one passed by value. To do so, select the double arrows under Pass Type, then click in the Tag box under "Name in Cycle3". Type in the tag name : cStep and cycleData. After you type in each tag name, press Enter. A dialog box enabling the selection of the data type associated with the input tag will pop up. In the first two cases, select ui16 for unsigned 16 bit. Click OK. Notice that after you OK the data type, the Define Address dialog box will close and the label next to the name will change to the data type selected; in this case ui16. The illustration on the right shows this process. The third entry, not shown, will be a "bit" type, named "run, selected as a pass by value.



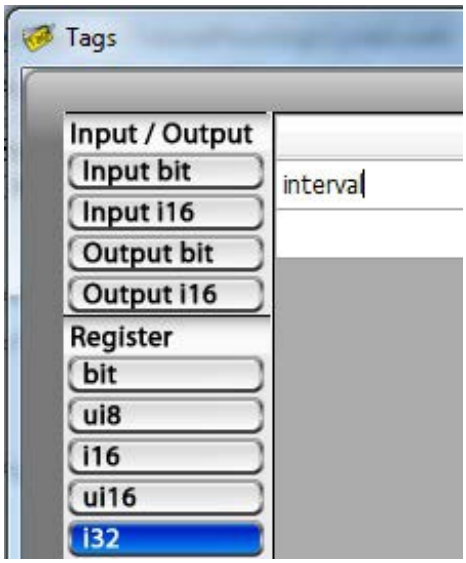
Click OK in the Subroutine Inputs dialog box when all three input parameters have been defined.

The Subroutine Inputs and Outputs block at the top of the subroutine window will now show the three items that you defined, including their data types and pass types, as shown on the right.

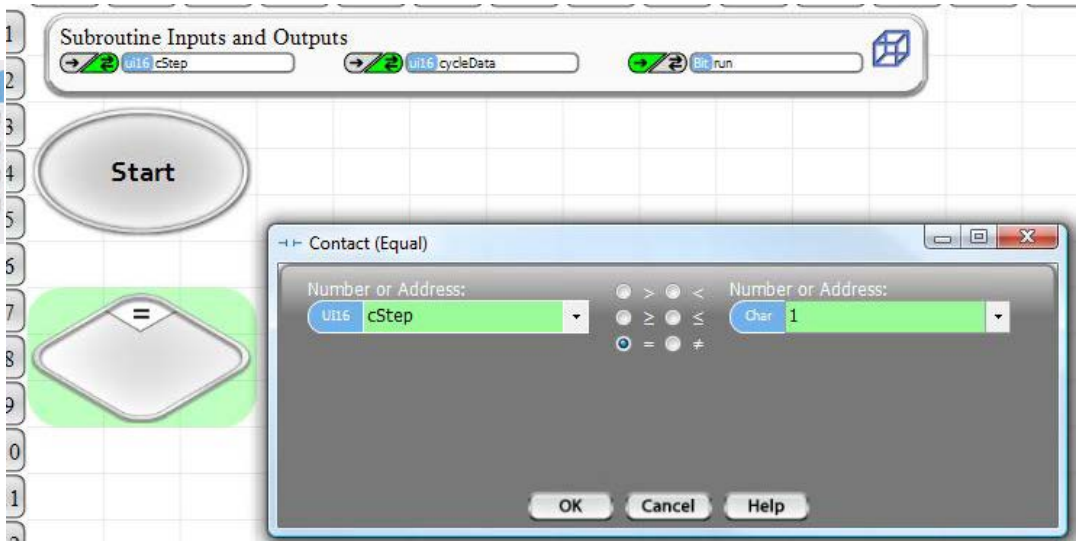


Now we need to define a local tagname variable for a timer that times the interval between changes in the output states. All timers are 32 bit integers.

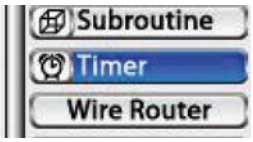
Start by selecting the Tag icon. When the Tags for the subroutine come up, select i32 and enter the tag name "interval". Click OK at the bottom.

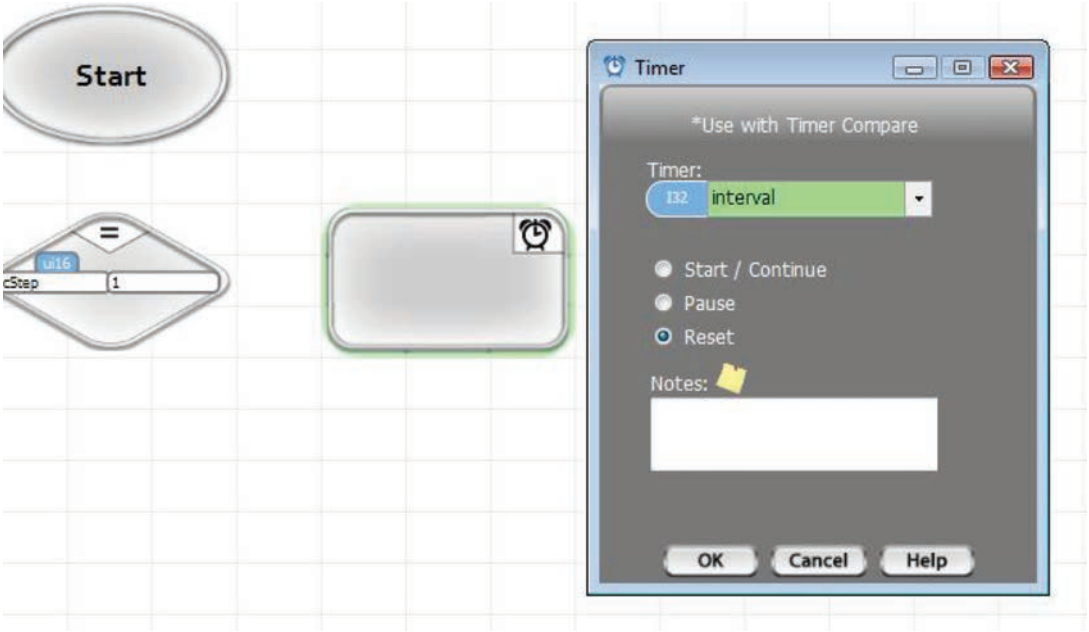


Select the Equal to icon from the Toolbox and place it's block below the Start block. That whole group of icons above the "Turn On/ Off" are decision blocks. This one checks whether two items are equal. Fill in the dialog box as shown on the right and select OK.

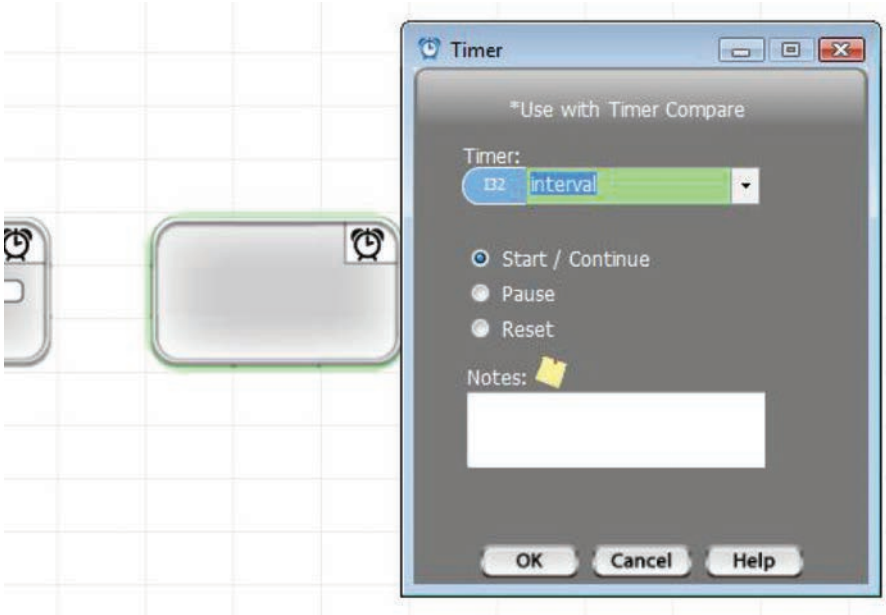


Next, select the Timer icon and drop a Timer block as shown. Select "interval" from the drop down list and select Reset. This will reset the interval timer to 0. Click OK.

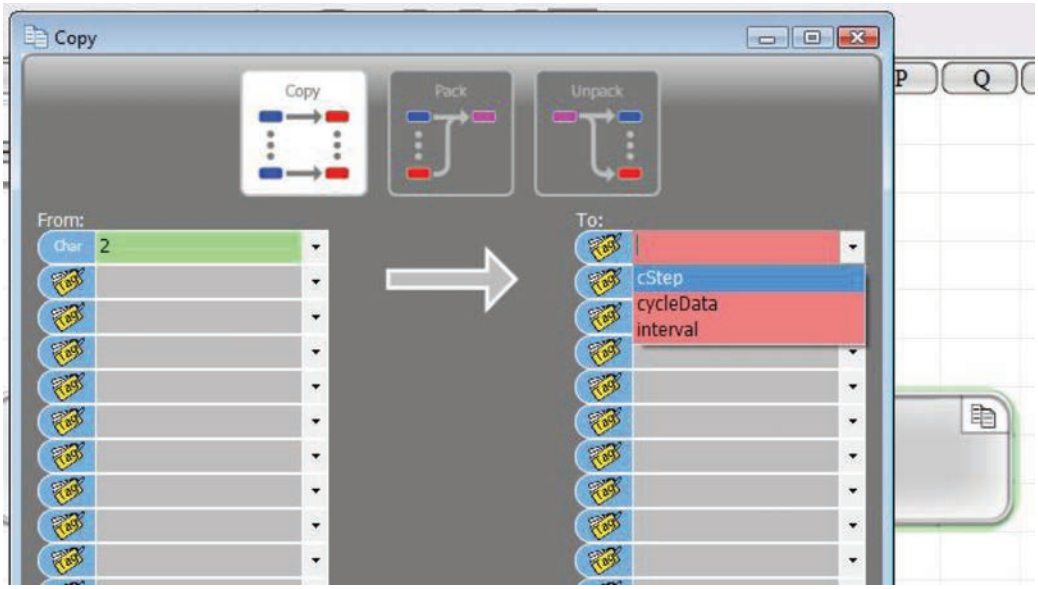




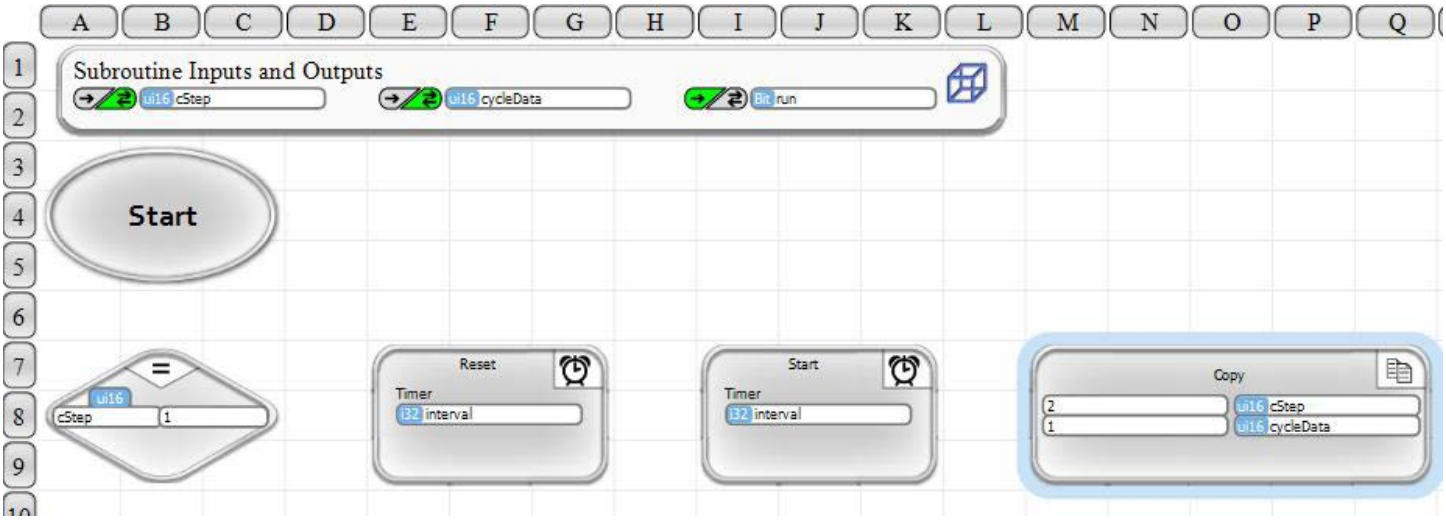
Select the Timer icon again and place a another Timer block next to the first one. This time, select Start/Continue, then OK.



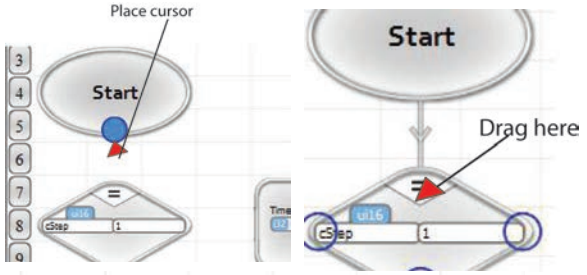
Place a Copy block next to the Start Timer block and copy 2 into cStep.



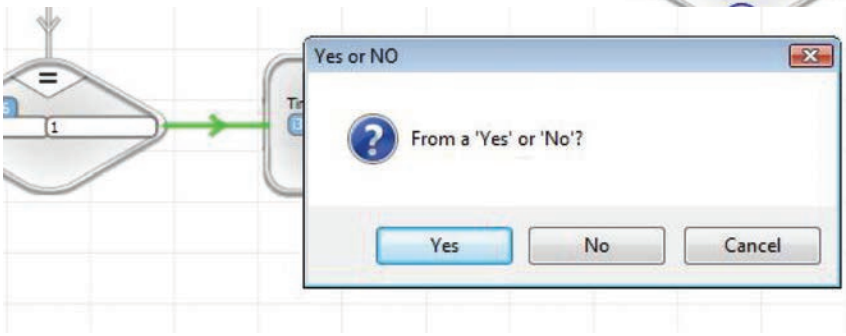
Your subroutine should now look like the illustration below.



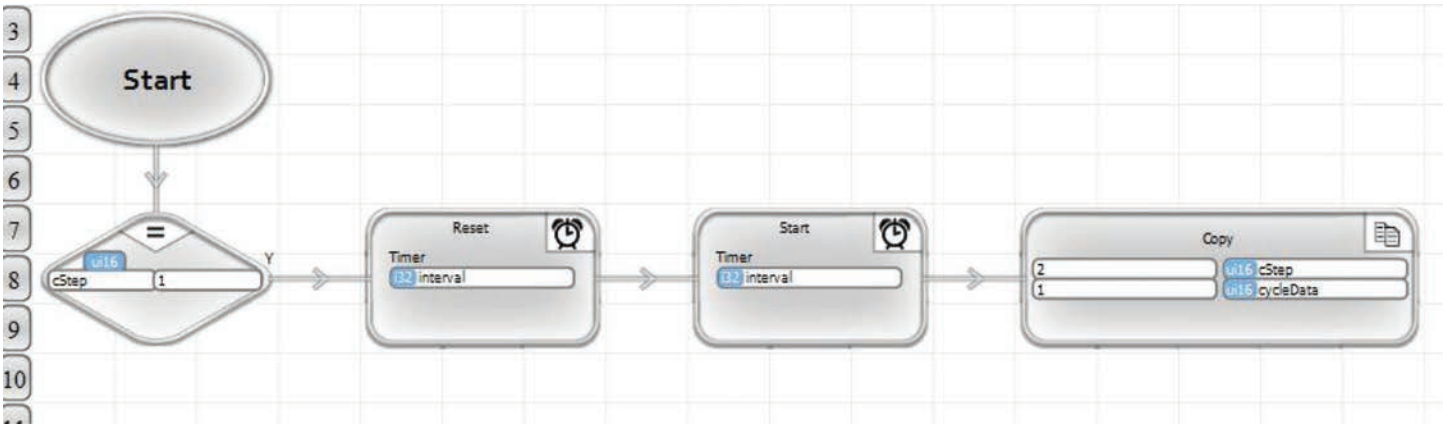
Let's connect up the blocks that we have now. Start by placing your cursor at the bottom edge of the Start block. Notice that a blue connection circle shows up. If you left click your mouse, hold it down and drag it to the top of the decision block, then release the mouse button, a connection line will be drawn from the Start to the decision block, as shown.



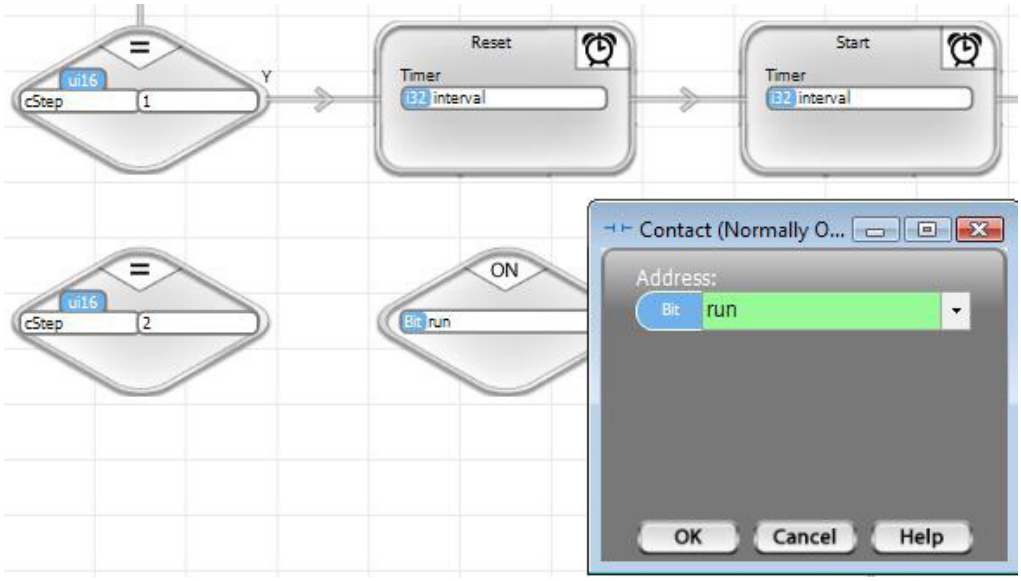
Now, make a connection from the right side of the decision block to the Timer Reset. A dialog box will pop up asking you whether this is the "Yes" or "No" connection. Decision blocks have two exits. They exit one way if it results in a "yes" the the question it answers and another way if it results in a "no'. Click the Yes button.



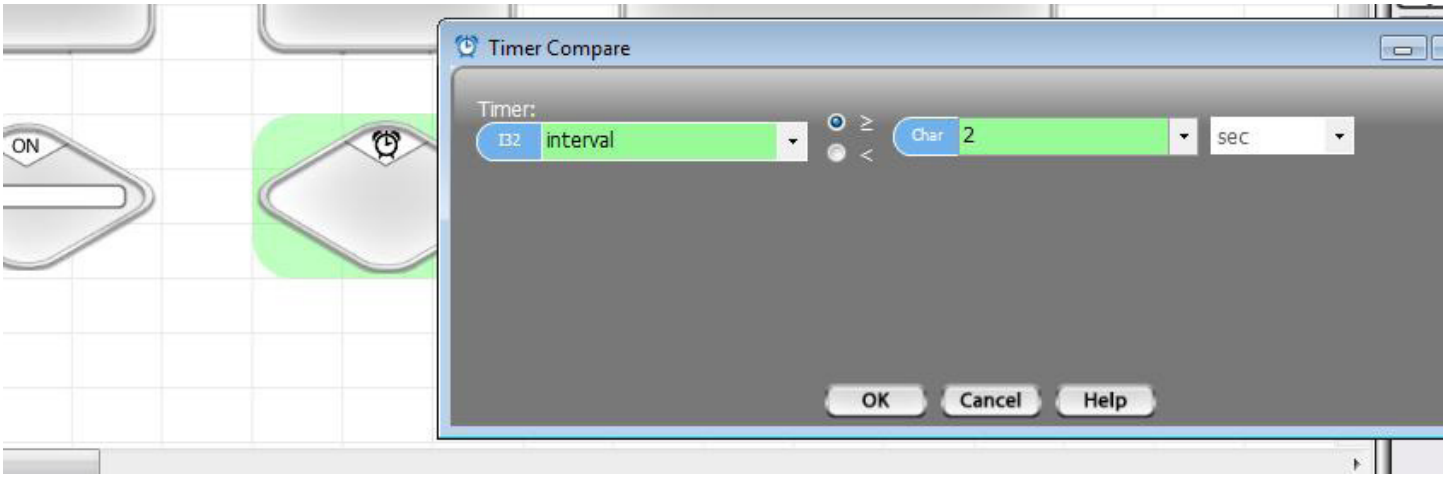
Your subroutine should now look like this.



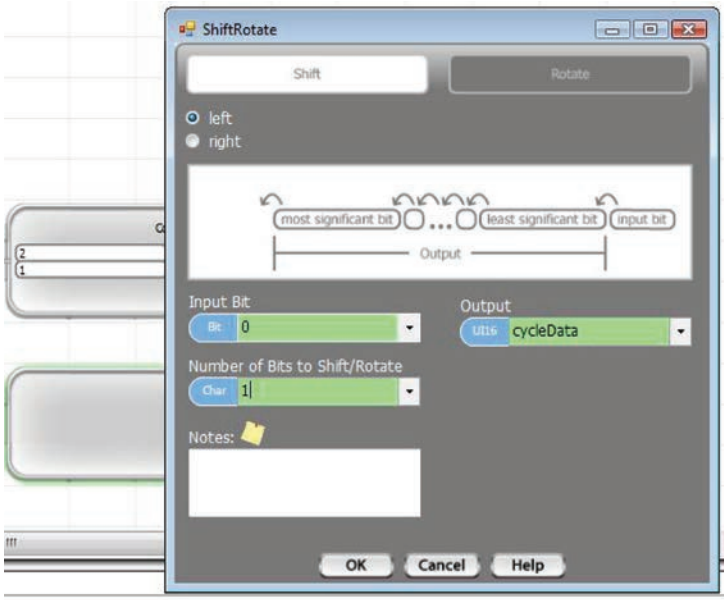
Continue on by placing another Equal decision block, below the check for cStep being 1, to check if cStep is 2 and an ON? decision block to the immediate right, that checks whether the "run" value, passed from the calling program, is "On'. When you place the "ON?" decision block, a dialog box will pop up. Select tagname "run" and OK.



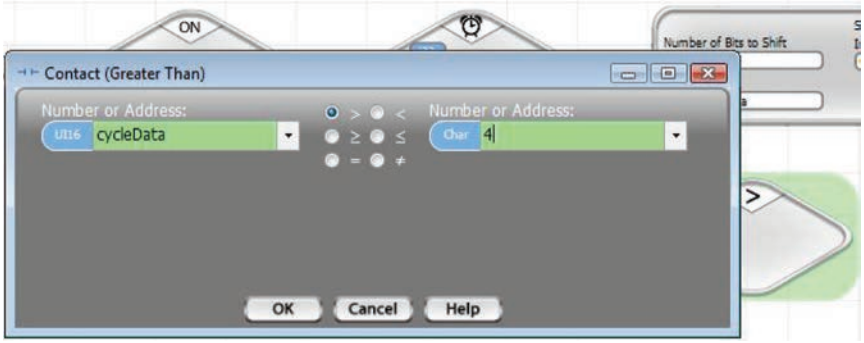
Place a Timer Compare to the right, as shown. The Timer Compare icon is shown on the right. When you place the Timer Compare a dialog box will pop up. Select the timer we defined (interval) by using the drop down selection and type a 2 in the second field, leaving the selection as greater than or equal to and the time units as seconds. When the timer value reaches 2 seconds, the program flow will execute the logic connected to the "yes" output of this block. Until then, it will execute the "No" logic. Select OK.



Select a “Shift/Rotate” block and place it next to the Timer Compare. In the dialog box, select Shift and left, enter a 0 as the Input bit, select cycleData for Output and enter a 1 as the number of bits to shift, as shown. This will create a program block that will shift the 1 in cycleData one position left, each time it is executed, and place a zero in the vacated, rightmost bit position.



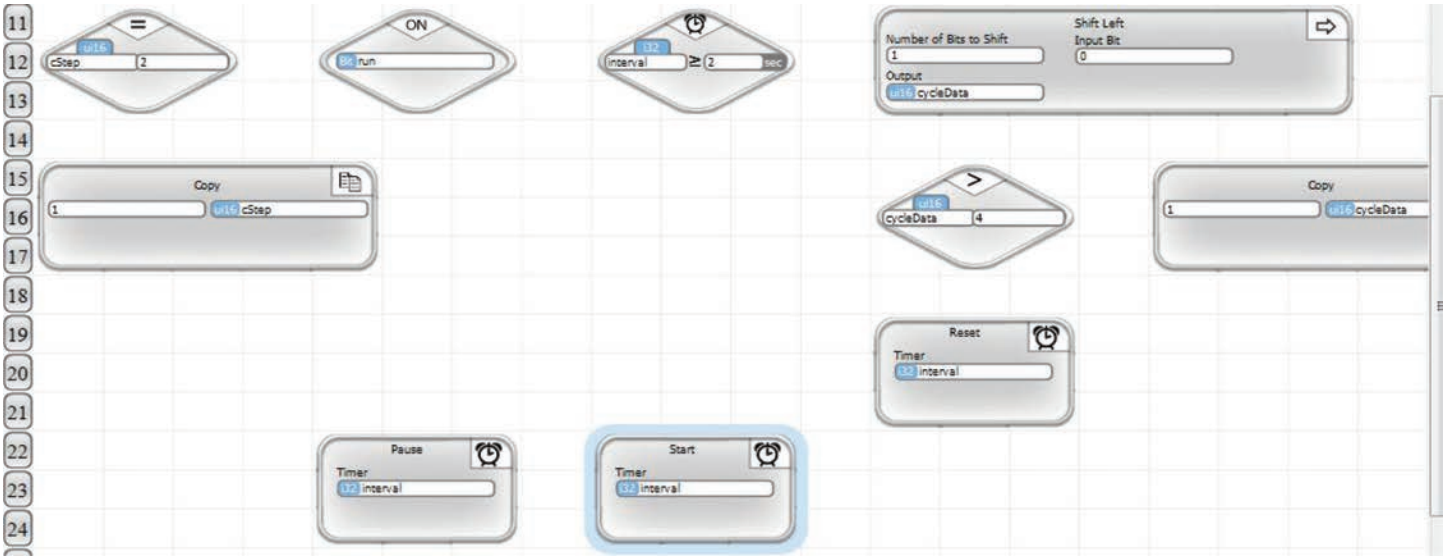
Below the Shift/Rotate block, place a Greater Than comparison. As we shift the bit left in an integer, the integer value increases from 1 to 2 to 4 to 8, etc.. Since this program is only stepping through three outputs, we want to start over again when we shift from 4 to 8. With this comparison, we’ll determine when that happens. Select cycleData for the comparison parameter on the left and 4 for the right. Leave the selection ‘>’ as shown.



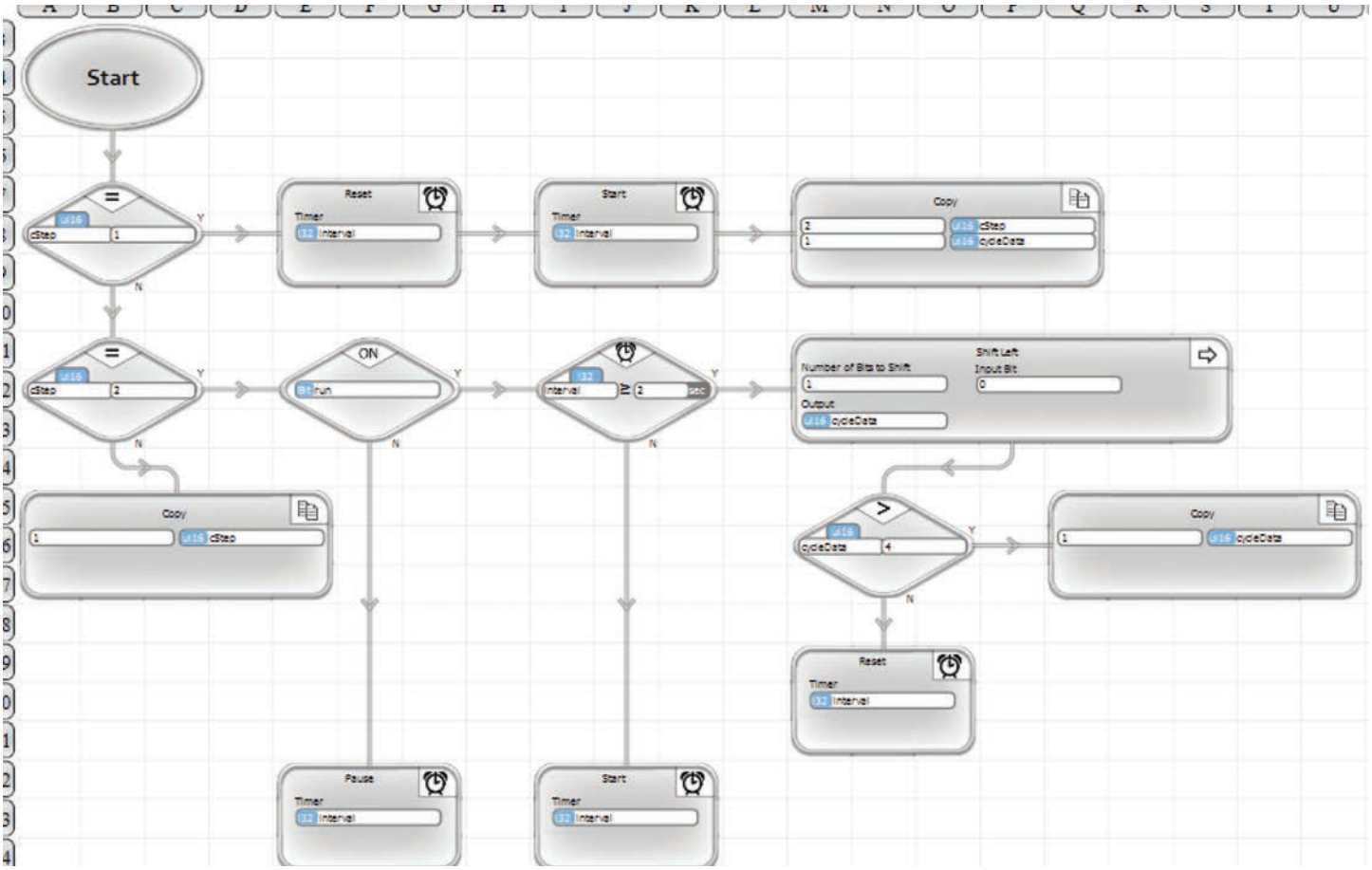
Add the following blocks :

- A copy block that copies the value 1 to cycleData, to the right of the Greater Than decision block.
- A Timer Reset of interval below the Greater Than block
- A Timer Start/Continue just below and to the left of the Timer Reset
- A Timer Pause just to the left of the Timer Start/Continue
- A Copy of the value 1 into cStep below the Equals comparison block

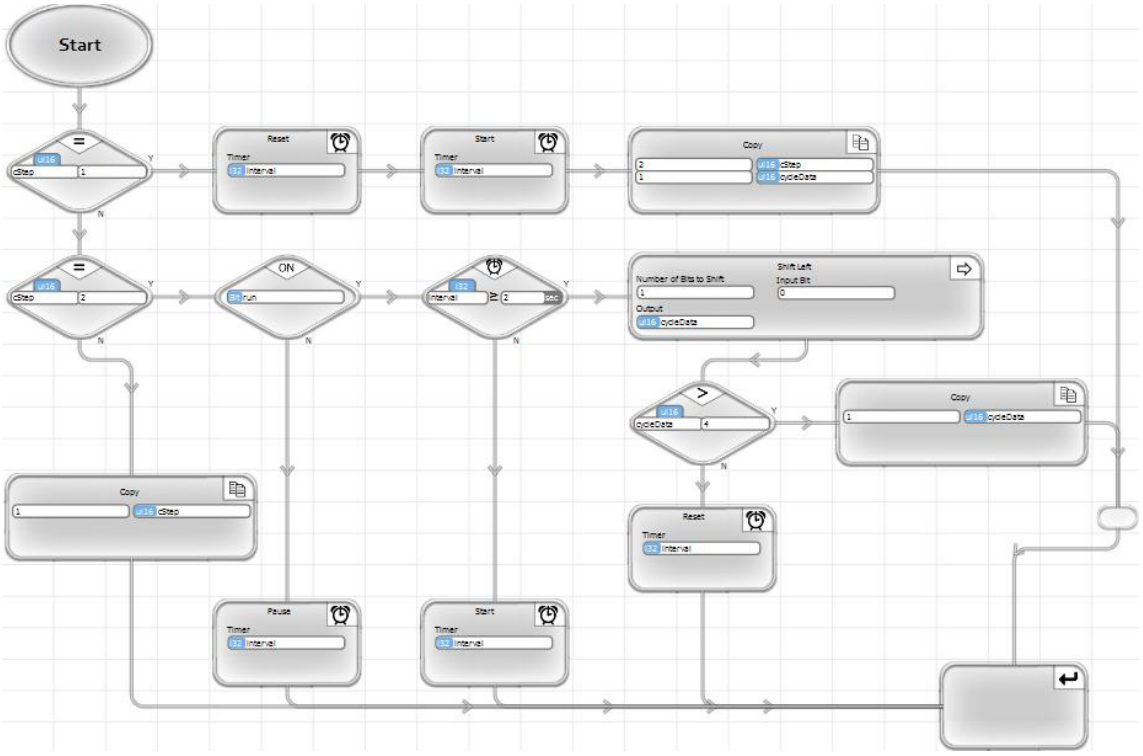
Your program should look like that shown below.



Wire up the subroutine as shown below.



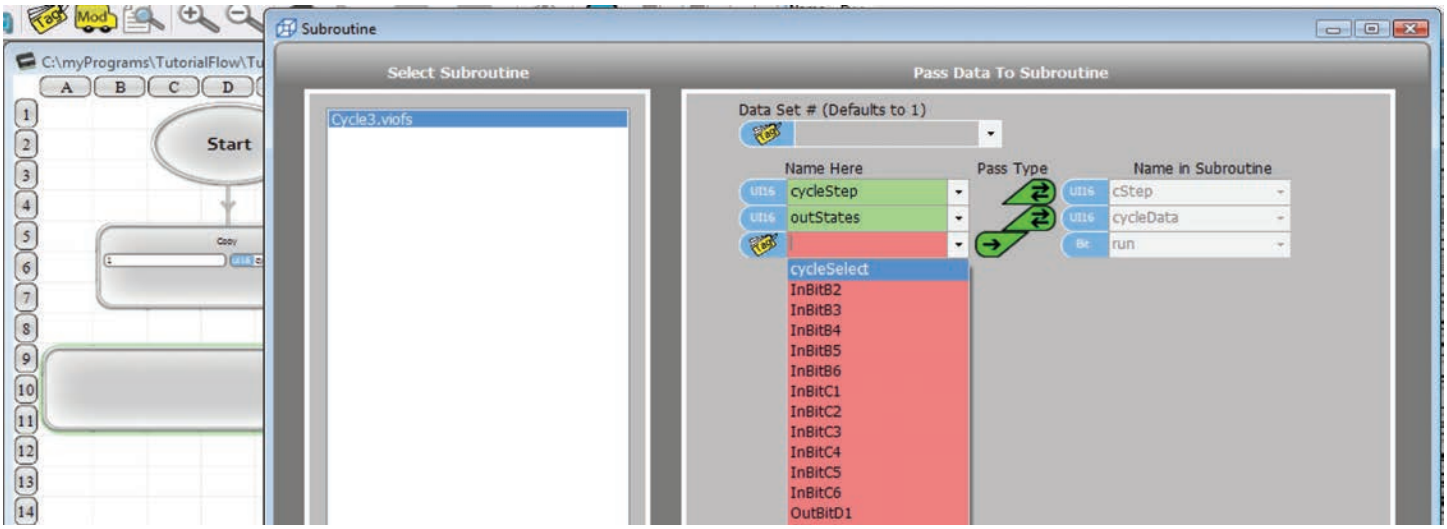
You could add a Sub Return block and wire up everything as shown below. However it's not necessary. When the program gets to the end of a line of execution in a subroutine it will return. Putting an explicit return block in is a matter of preference. It may make documentation more clear. Its your choice. Note : the small block on the right side that shows no operation is a Wire Router, available at the top of the tool bar. It can be used to make sure wires are routed the way you want. It does nothing.



We have now completed our Cycle3 subroutine. Take a look at what it does. It is a simple state machine (see the chapter on state machines for a more thorough discussion). In the first step, the timer and output data are initialized. In the second step, if the run switch is on, the data is shifted around a loop every 2 seconds. If the run is not on, the outputs remain the same. You should never get to any other step, but if you do (it would have to be some error somewhere else), this subroutine will restart step 1.

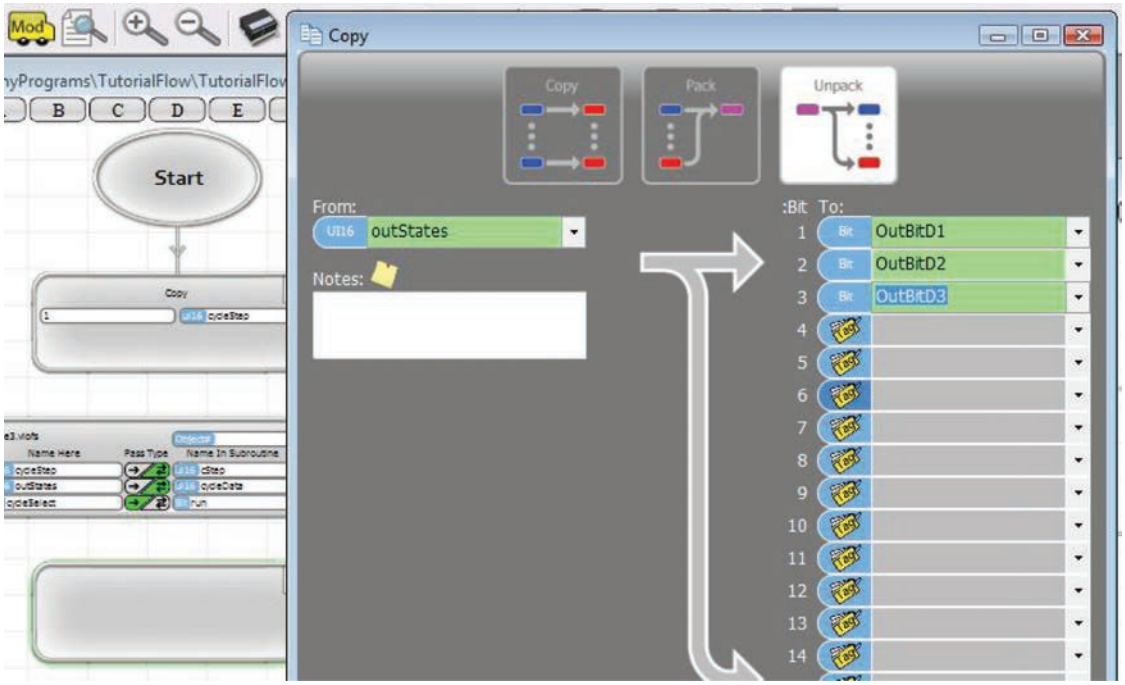
Click the icon in the upper right corner that shows layered windows. When you do, you will get smaller windows with both your main program and your subroutine.

Select the main program. Select and place a subroutine block. Notice when you select Cycle3 in the dialog box, the three passed parameters will automatically come up on the right side of the pass list. You need to select the main program parameters that you want to pass in. Select cycleStep, outStates and cycleSelect as shown, then click OK.

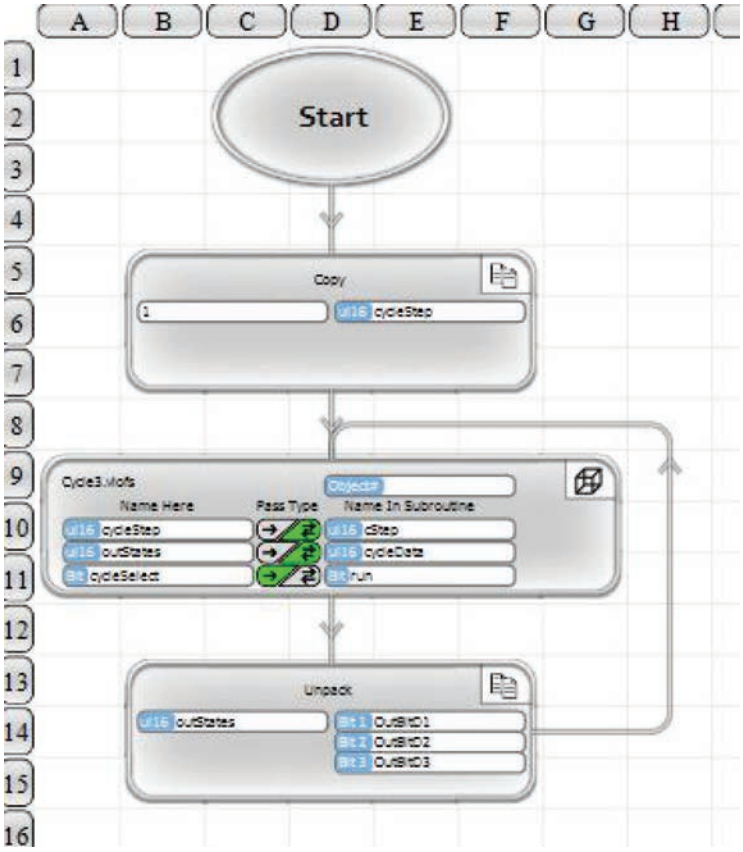


Under the subroutine call, place a Copy Unpack block. Select Copy, place the block, then select the Unpack icon at the top of the dialog box. Select outStates to unpack From and outBitD1, outBitD2 and outBitD3 to unpack the three least significant Bits To. Click OK.

,

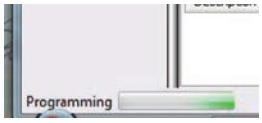


Wire your main program up as shown on the right, and you're in business! We've got a complete program.



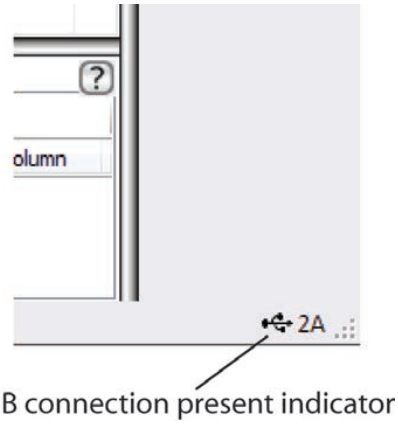
Tutorial Flow Chart Download, Debug and Run

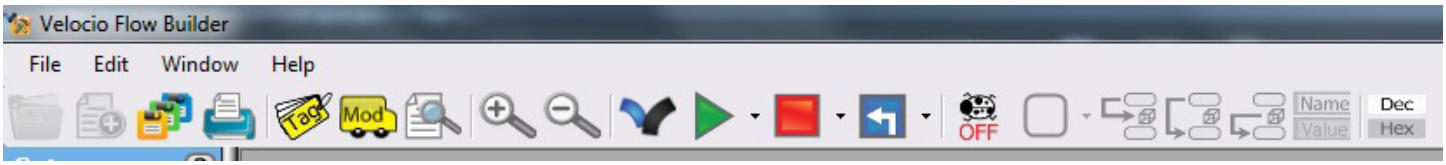
Now, we're ready for some fun. If you have a Branch or Ace PLC (a Velocio Simulator is ideal), power it up and connect the USB cable. You should see a USB connection present indicator in the lower right hand corner of vBuilder. If you have that USB connection, click the Program icon at the top of vBuilder. As you click the program button, watch the lower left corner status indicator. It will switch to "Programming" and show activity. This is a short program, so this will happen very quickly, then change to "Stop" status.



Once you've downloaded, you're ready to run and/or debug. If you've entered the program exactly as directed, it should run - with one little bug.

Take a look at the top tool bar. Next to the Program icon, you will see the Run, then the Stop button. Next is Reset, which will cause the program to start over from the beginning. The next set of button icons are debug functions, which we will use as we debug this program in the following pages.

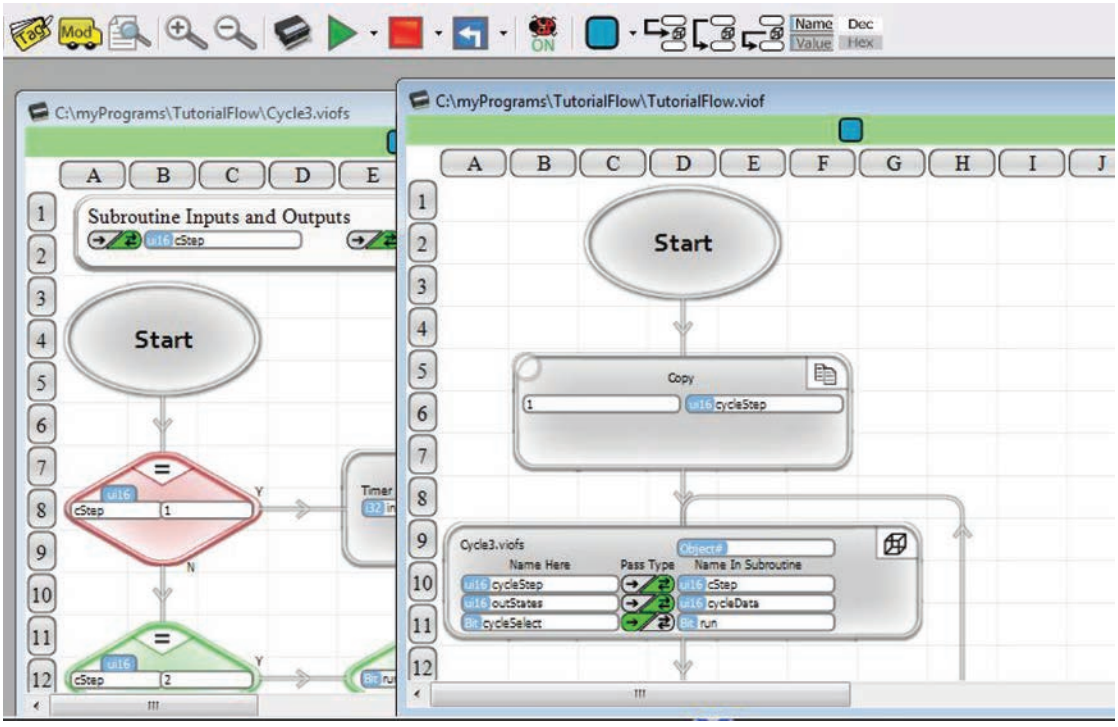




Try running the program. Click on the Run icon. The status indication in the lower left corner should change to "Running". If you have correctly entered the program as described and switch the input that we chose for "cycleSelect" (input B1) on, you should see output LEDs cycle every two seconds. We intentionally put a bug in the program though. You should see two outputs cycling, instead of the three that we intended. That gives us a chance to debug the program.

Select Stop. The status indication in the lower left corner will say "Stopped". Select the icon that looks like a ladybug. It is the debug mode selection. You should see it change from indicating that it is OFF to indicating that debug is ON. You will also see that your main and subroutine windows will have a red bar across the top. This red bar is an indication that the routine is not currently running.

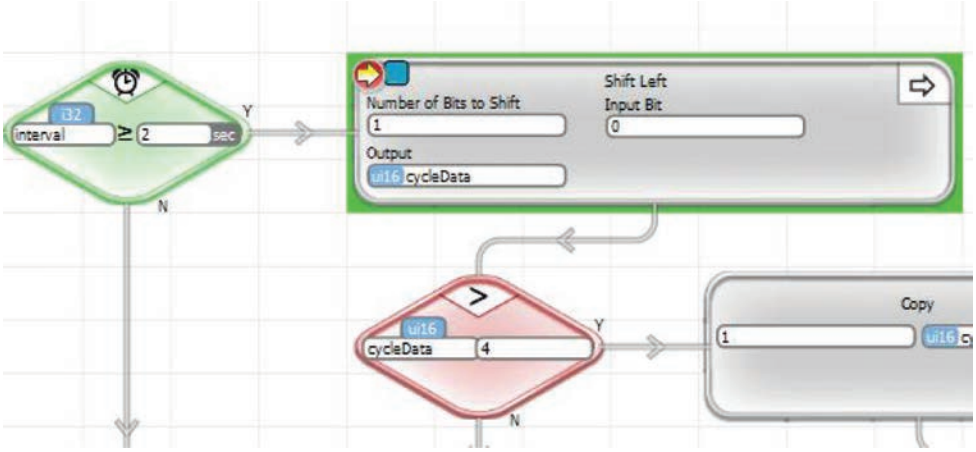
Select Run. The colored status bar on each program window will change to green, indicating that the program is running. You will also see some decision blocks turn red and others green. This is a high level indication of the predominent execution of each decision. To see the details, we have to look closer, but just this quick view tells us that we are executing the subroutine's step 2. Toggle the cycle-Select input and you will see the 'run" decision in step 2 change between red & green. The PLCs are only polled by vBuilder twice per second, so there will be a time lag on the screen.



While the program is running in Debug mode, notice the icons on the top tool bar labeled Name/Value and Dec/Hex. Look at your program & notice that your program blocks have tagnames in them. Click the Name/Value icon, so that the Value portion is highlighted and notice that your blocks now have numeric values. These values are the current values of the tag named variables. Toggle back & forth. Try the Dec/Hex icon, when selected for value. The values change between decimal and hexadecimal format. Since we are dealing with such small numbers, there is really no difference. When you have a program with bigger numbers, it will be more distinct.

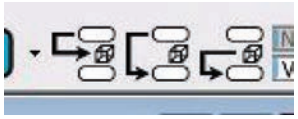
We need to figure out why the program is only cycling between the second and third outputs. Go to the Cycle3 flow chart and click on the Shift block.

Notice that a little blue box is placed in the upper left corner of the Shift block and within a couple of seconds the program stops with the Shift block highlighted and an arrow in the upper left corner. You have just placed a break point at that block and the program execution stopped at the break point.



We are interested in what happens when cycleData is shifted and what is different when we should shift from the third bit to the first bit. Toggle the Name/Value to Value and see what you have in the cycleData.

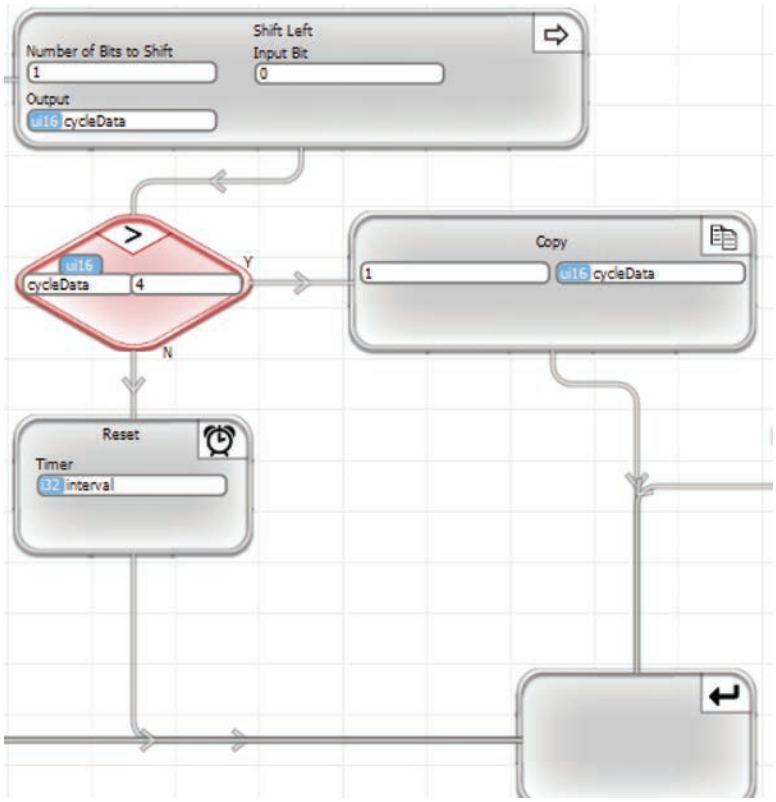
Notice the icons on the top tool bar that look like this -



These are single step functions. If you hover your cursor over them, you will see that the first one is “Step In”, the second “Step Over” and the third “Step Out”. The first one can be used to step through the execution of your program, one block at a time. The second one operates the same, except when the block is a subroutine call. Instead of stepping into the subroutine, it executes the entire subroutine. The third one will step through execution, but if you are in a subroutine, it will execute all blocks until you return to the calling program; in other words, “Step Out” of the subroutine. Try these functions.

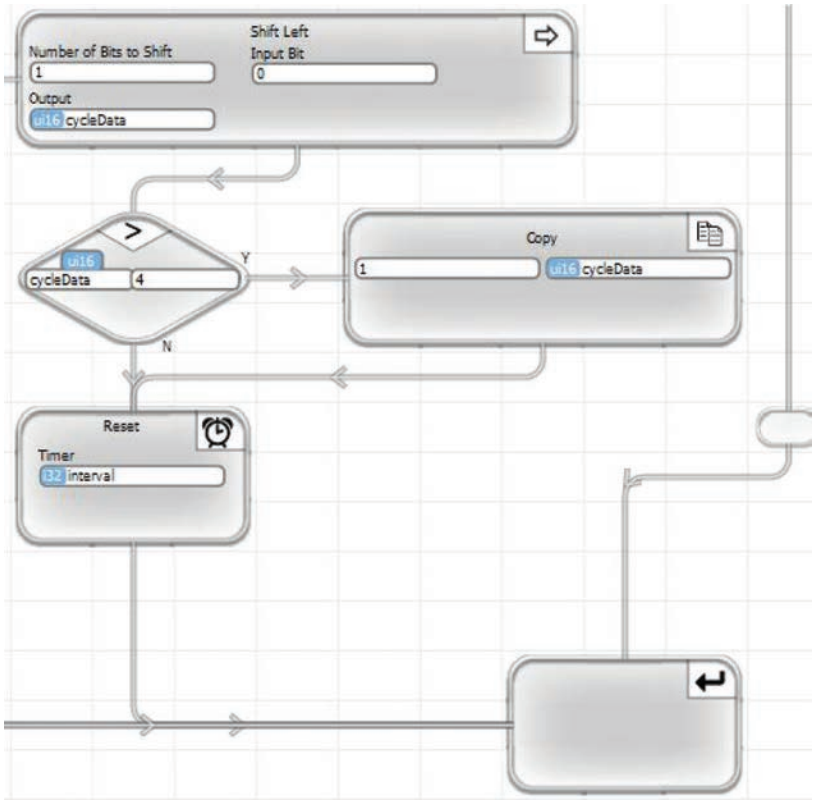
Click the Run and wait until the program breaks at the Shift block, then step through the execution for the next few blocks. Watch the cycleData when you do this. Notice that when cycleData is shifted from the value 4 to 8, then checked to see if it is greater than 4, the next block that executes, resets the cycleData to one (first bit on). That is what we want. If you keep stepping, you see that it does turn the first output on. But when we run it full speed, with no break points, we don’t see the first output coming on. What’s going on?

Take a look at your program to see what is different when we roll cycleData from 4 to 1, than the other two conditions. Notice that when cycleData is shifted from 1 to 2 and 2 to 4, the program resets the timer. When we go from 4 to & reset it to 1, we don't reset the timer. Single step through and see that the value of interval is still greater than 2 seconds, so we will shift again. It happens so fast that we never see the first output's LED come on. We need to change the program so that it resets the interval timer after every shift.



Select the Stop icon on the top tool bar. Click the Debug icon to exit Debug mode. Route the output of the Copy block to the Timer Reset, as shown. Download this program and run it. It will step through all three outputs, as we intended.

You have now written a flow chart program with a subroutine and debugged it. Congratulations!

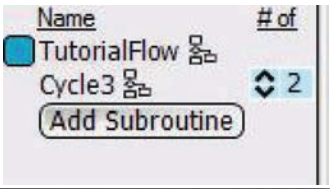


Adding a Second Object

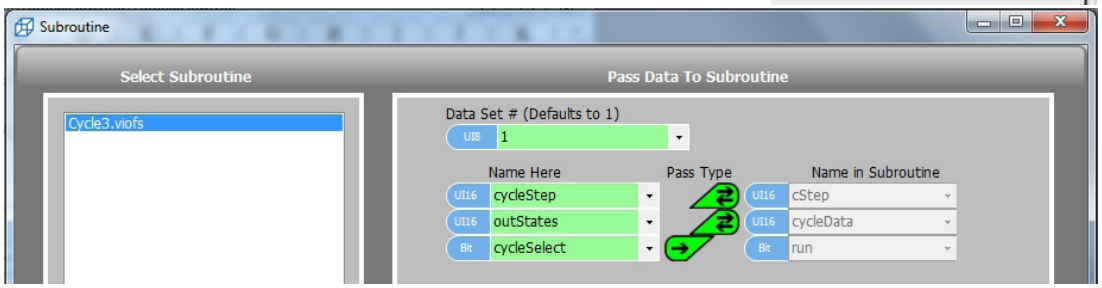
Your tutorial program works, but we can't resist demonstrating one of the advanced features of vBuilder. Subroutines, in vBuilder are actually Objects. An Object is a self contained program, coupled with its own captive data. In a vBuilder program, multiple Objects of the same type can be created and used. Objects can also be copied to other programs, giving you re-usable program components that you have already debugged and proven to work. We're going to implement a second Object as a demonstration.

Select your main program, select the Tag icon, and add the following tagname variables :

- Select Input bits and rename InBitB2 'cycle2Select'
- Select ui16 and add 'cycle2Step' and 'outStates2'

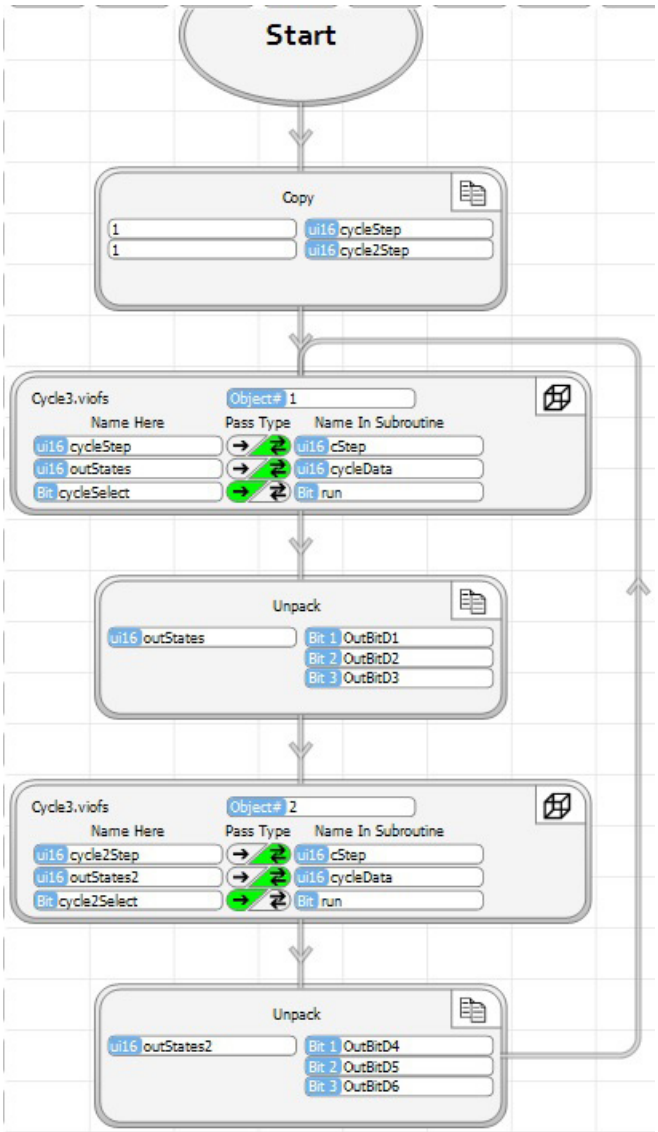


Take a look at the list of Project Files along the left hand side of vBuilder. Next to Cycle3 is an up and down dial and the number 1. Select up to get the number to change to 2. You've just created a second Cycle3 object. Now lets see what we can do with it.



First, double click on the Cycle3 Subroutine call in your main program to open up the dialog box. Notice that there is an entry for Data Set # that we didn't do anything with. We didn't need to since there was only one. Put a 1 in the box, as shown. You just defined this call to Cycle3 as using object 1.

Now add a second Subroutine call to Cycle3 and a second Copy Unpack and shown. Make sure you select Data Set 2. Pass cycle2Step, outStates2 and cycle2Select in. For the Copy Unpack, unpack from outStates2 to OutBitD4, OutDitD5, and OutBitD6. Your program should look like the illustration on the right



Program it and run it. You will see that when you flip the B1 switch (which you gave the tagname 'cycleSelect') on, the first three outputs will cycle at a 2 second step rate. When you switch the B2 switch (which you gave the tagname 'cycle2Select') on, outputs 4, 5, and 6 cycle. Try turning the inputs on and off. Notice that the step time for the 2 groups is not synchronized. That is because we created our timer in the Cycle3 object. The two objects each have their own independent timers. If we had wanted to synchronize the step times, there are simple ways we could have done so. The purpose was to demonstrate that we can create two totally distinct entities, or Objects, that operate independently, but use the same logic.

There are other powerful aspects of vBuilder that you have just unleashed - but we'll discuss those in later detail pages. Let's move on to Ladder Logic programming.

Tutorial Example 1 : Ladder Logic Implementation

For those who have been through the Flow Chart tutorials, you will notice that we repeat all of the explanations in the ladder logic tutorial. We do this for the benefit of people who have a strong preference for Ladder Logic and haven't gone through the Flow Chart tutorials. If you have been through flow charts, this is either review, or material that you can skim through.

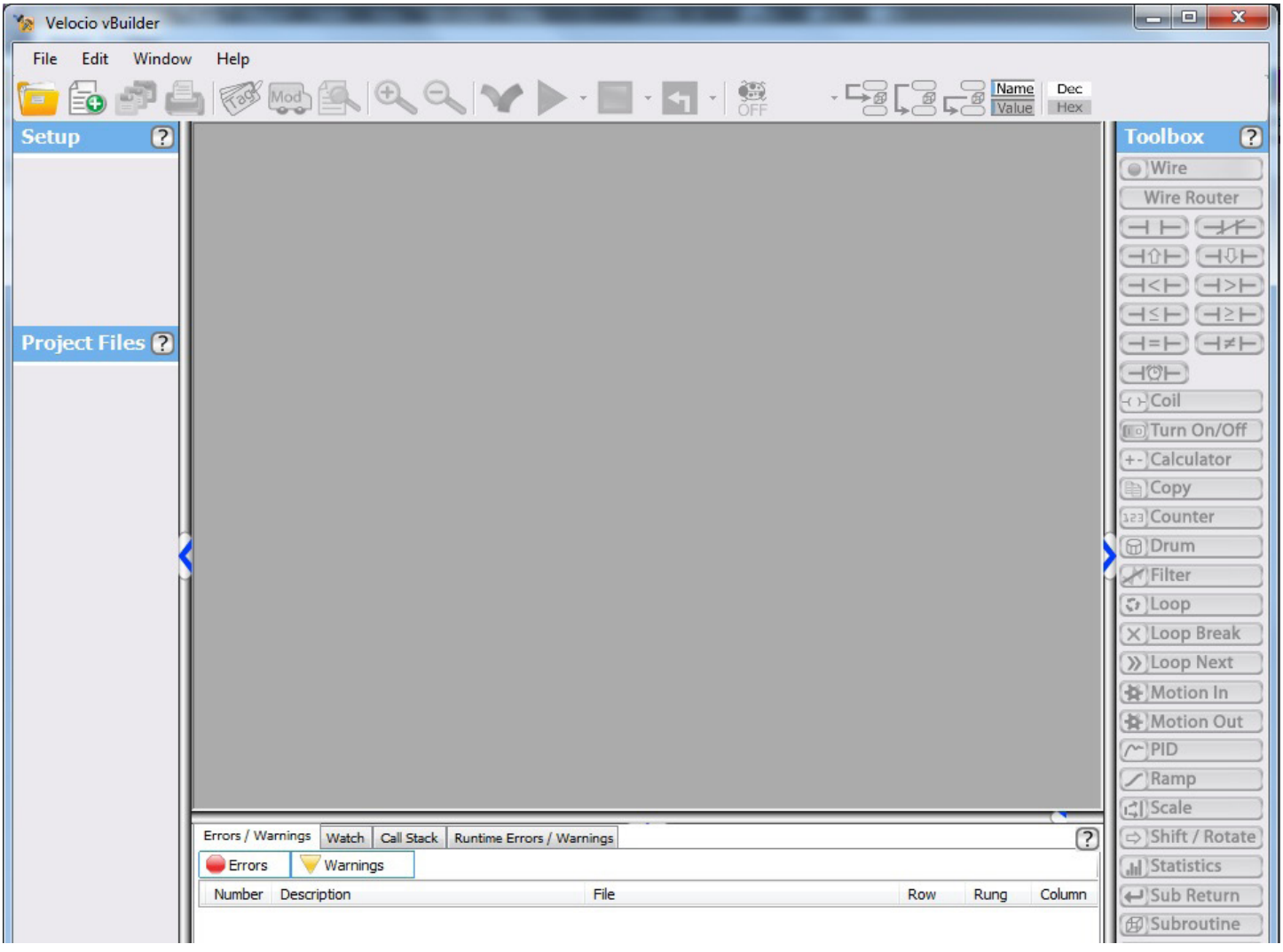
If you have installed Velocio Builder (referred to here as vBuilder) on your computer and followed all of the default settings, you should have a logo, like the one on the right, on your desktop. If you declined to put an icon on your desktop, locate it though the Start menu of your computer.

If you need help installing vBuilder, see Appendix A

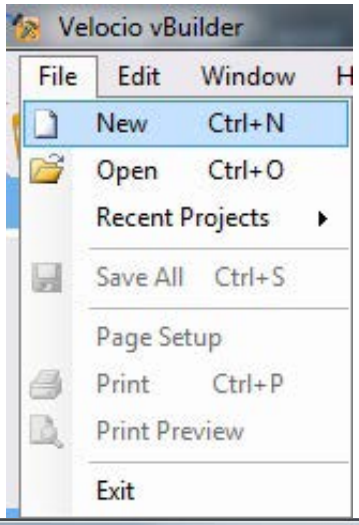
Double click on the Velocio Builder icon to start the program.

If you get a Windows message that asks whether you want to allow vBuilder to make changes to your computer, click Yes. The changes that vBuilder makes to your computer is saving your program files.

You should get an opening screen that looks like the one shown below. At this point you are ready to begin entering a program.



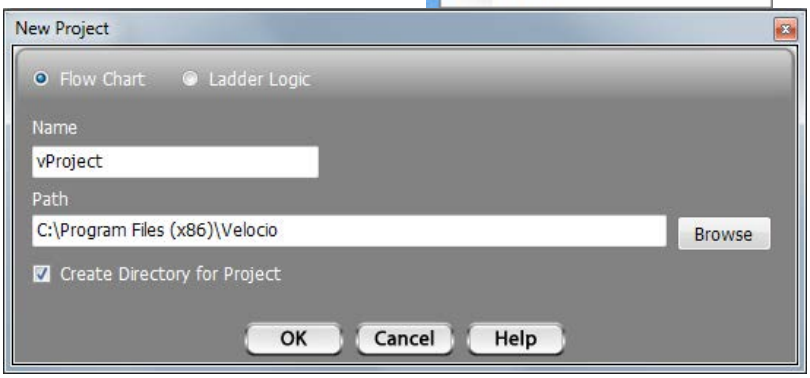
Begin your program by selecting the File Menu, then "New".



A dialog box, like that shown on the right will pop up.

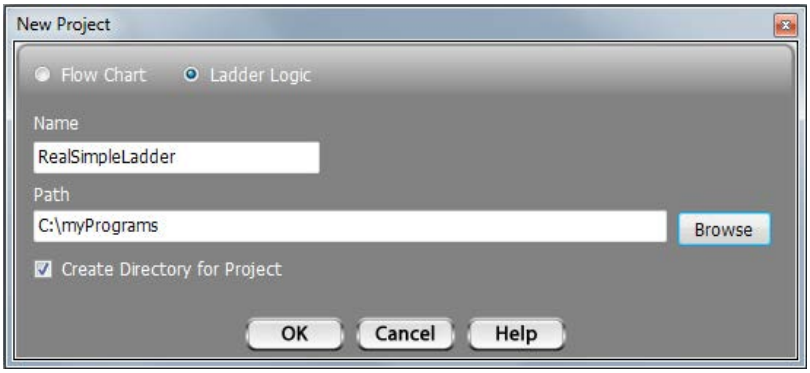
The dialog box opens with default information selected. You're not likely to want to accept default selections unless you are just testing things, so we'll make some changes.

At the top of the dialog box, you see that we can select to write this program in either Flow Chart or Ladder Logic programming. We're going to do this one in Ladder Logic - so click the radio button next to Ladder Logic.



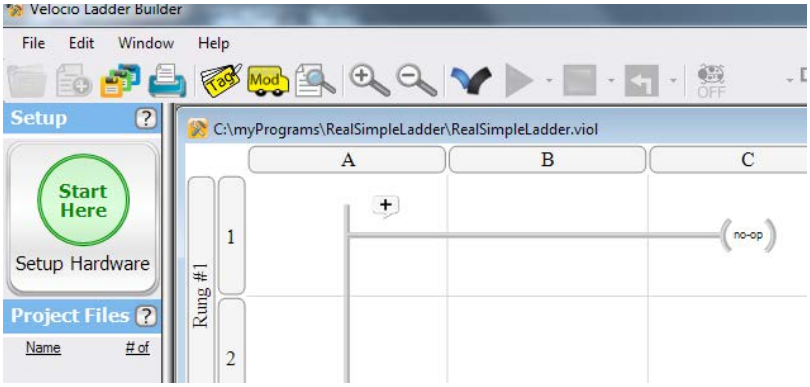
The Name box allows you to name your program. This example will be the simplest Ladder Logic program that you will ever write. Pick an appropriate name, like "RealSimpleLadder". Just type "RealSimpleLadder" in the box labeled Name.

The box under "Path" allows you to select the folder, on your computer, to store your program. The one shown is the default location. You can use this location, or select another one. For the purpose of illustration, create a folder called "myPrograms" off your computer's C: drive (using Windows Explorer), and browse to it and select it. Your New Project dialog box should now look like the one shown on the right.



Click "OK" to create the new project.

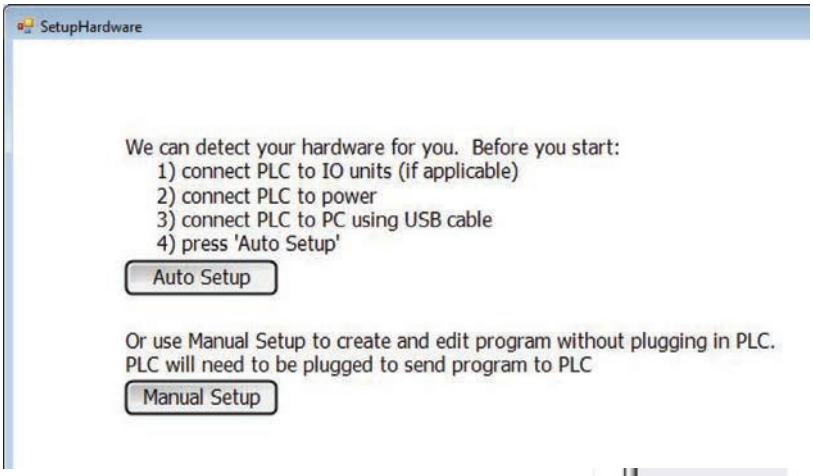
Notice that a programming window, named "RealSimpleLadder.viol" opens up. Also in the Setup area, a big green circle labeled "Start Here" comes up. This is your starting point in defining your program. The first thing you need to do is set up the hardware.



Select the big green button that says "Start Here".

A window will pop up, which explains your hardware configuration options. As stated in the pop up window, you have the choice between connecting up to your target system and letting vBuilder read and auto configure, or you can manually define the target application hardware.

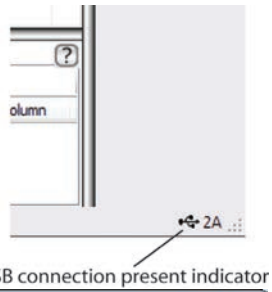
In this tutorial, we will go through both options. However, in order to actually program the PLC, debug and run it, you will need to have either an Ace or Branch unit.



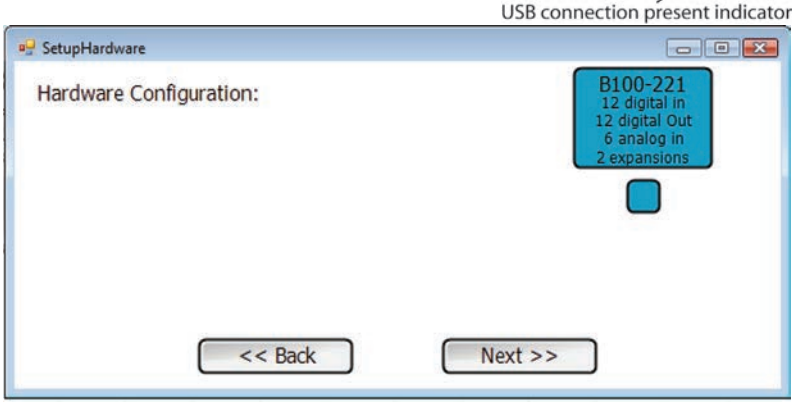
Using Auto Setup

In order for vBuilder to automatically set up your hardware configuration, you must have the Velocio PLC connected like you want the system set up, powered on and connected to the PC.

- Connect up your target Velocio PLC system (in this case, simply either an Ace or a Branch module (no expansion modules are necessary - a Velocio Simulator is ideal)
- Connect a USB cable from your PLC to the Ace or Branch unit
- Power everything on.
- You should see a USB icon, shown on the right, in vBuilder's lower right hand corner. This is an indication that a Velocio PLC is connected (and on) to the PC.
- If everything listed above is OK, select "Auto Setup" in the "Setup Hardware" window.



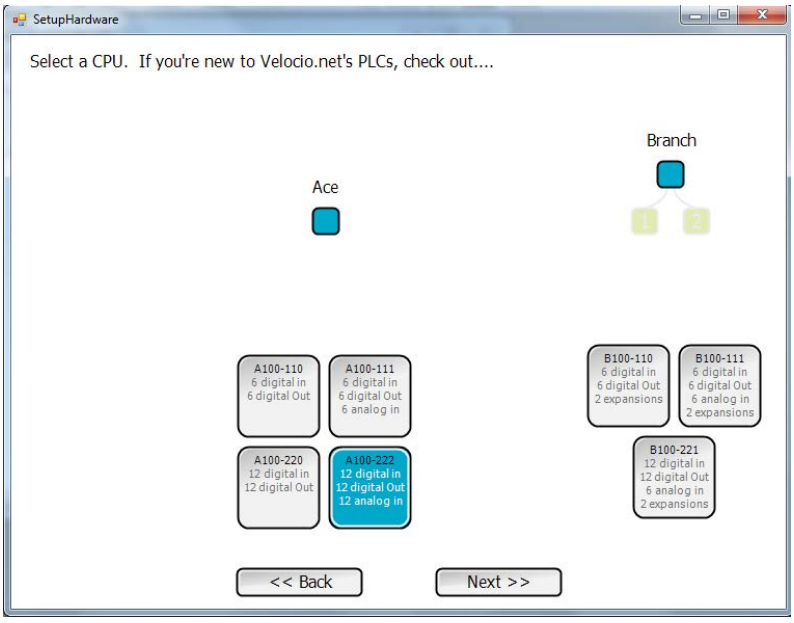
The "Setup Hardware" window should change to display something like what is shown on the right. In this case, it shows that it is configured for a Branch module that has 12 digital inputs (2 ports), 12 digital outputs (2 ports) and 6 analog inputs (1 port). It also has two expansion ports - with nothing attached.



This tutorial can be implemented on any Branch or Ace that has at least 6 digital inputs and 6 digital outputs.

Using Manual Setup

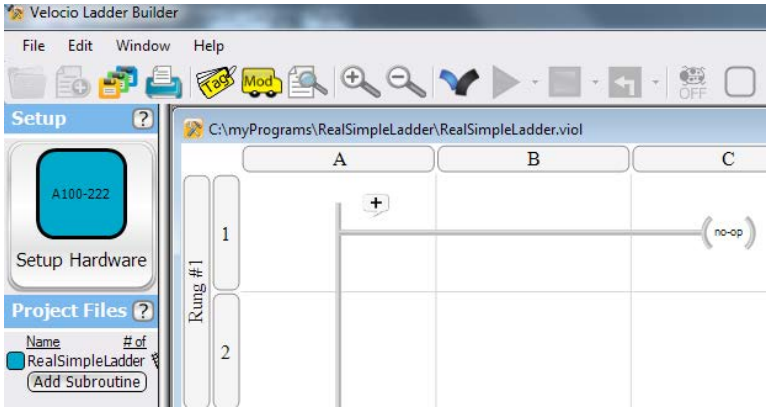
If you don't have the PLC hardware available, you can set the system up manually. Start by selecting "Manual Setup" from the "Setup Hardware" window. The Setup Hardware window will change to something like the image shown on the right.



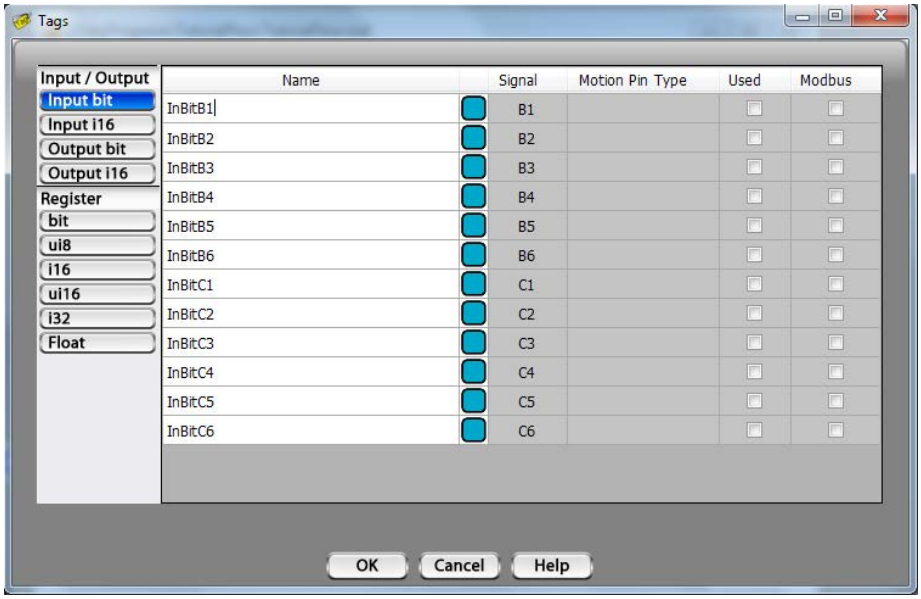
The first step is to select the PLC main CPU. This will be either an Ace, or a Branch unit. The selections are the labeled, gray squares below the Ace & Branch icons. These selections identify both the type (Ace or Branch) and the associated IO configuration. Select the one that applies to the application. In this tutorial example, any of these units will work. The configuration that you select will turn blue, as shown.

After selecting the PLC hardware for the main PLC,, you must then select "Next" to continue with the configuration of expansion modules, if any. After all of the PLC modules are selected, there will be additional screens, which allow you to define whether any Embedded Subroutines are to be placed in Branch Expansionss (if you configure Branch Expansions) and for Stepper Motion and High Speed Counter signal. We won't be doing any of that - just click the Next & finally Done.

When the hardware setup is defined, your screen should look like the image shown on the right. The Setup is shown on the left side, next to the top left corner of the program entry screen. It shows that we're configured for an Ace with 12 DI, 12 DO and 12 AI (the -222 means 2 digital input ports, 2 digital output ports and 2 analog input ports. Each port has 6 IO points). The area just below the hardware setup is the list of project files. The main file is automatically created with the name we defined when we created the new project. In this case it is called "RealSimpleLadder".



Select the Tag icon on the top tool strip.

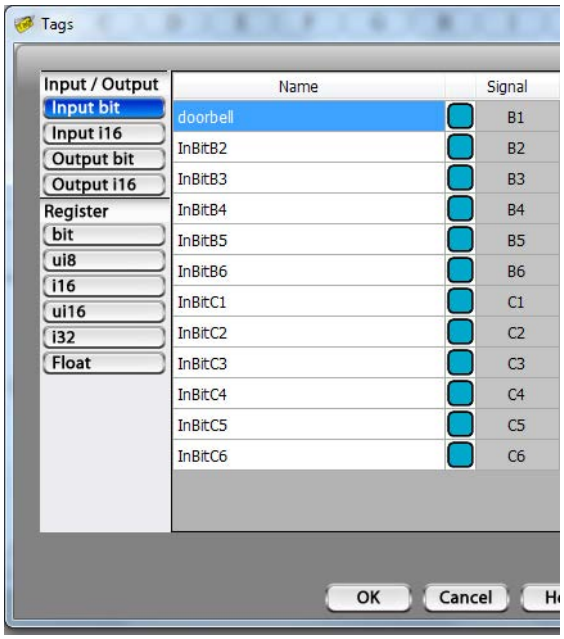


The tagname dialog box will open. Select the various options on the left under Input/Output and Register. You will see that when you select any of the options under Register, you get a blank table. That is because we haven't created any tagnames yet. Register tagnames are just general data identifiers that will mean something and hold information for your program. They are not directly tied to inputs or outputs.

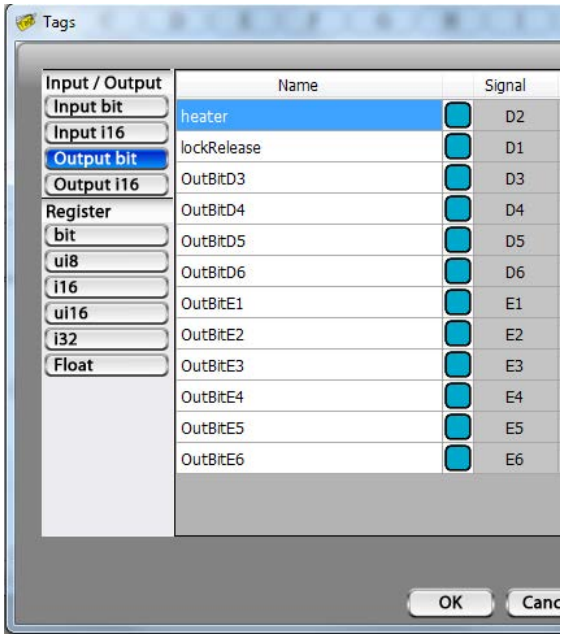
Notice that there are lists of tagnames for Input bit, Input i16 and Output bit. These were automatically created when you set up the PLC hardware to run on. vBuilder knows that an A100-222 has 12 digital inputs. It automatically creates tagnames for each of the digital inputs. These are default names. You can change to more meaningful names, if you want. The default names are pretty simple. For example, InBitB1 is assigned to the digital input connected to the signal B1.

Likewise, tagnames are automatically created for the 12 digital outputs, located in the Output bit list and the 12 analog inputs, located in the Input i16 list. The raw analog input values are signed 16 bit integer numbers.

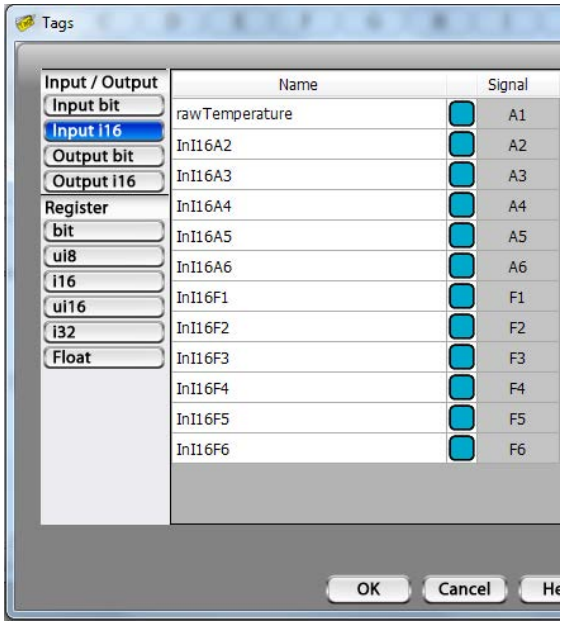
For our example, lets define meaningful names. In the Input bit list, change the first input bit's (the one connected to B1) to "doorbell", as shown on the right.



In the Output bit list, rename the D1 output "lockRelease", and the D2 output "heater".



Next, change the tagname of the first analog input (Input i16) to "rawTemperature", as shown.



Program Entry

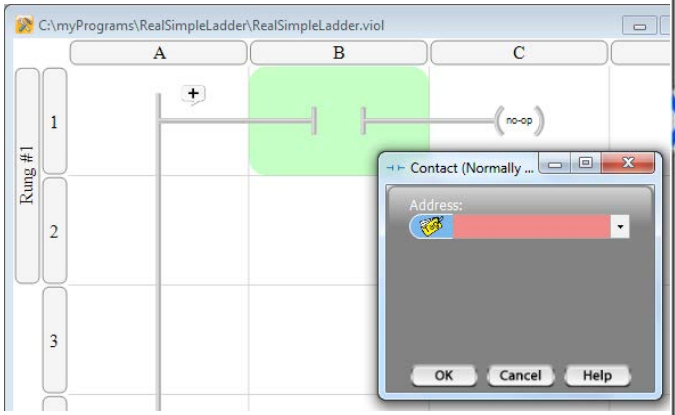
This example program will be one that is vey simple, yet will provide you with the basic concepts to build upon. We will implement it in three phases.

- Phase 1 : When the "doorbell" is active, we will turn on the "lockRelease". That's probably not a good idea in the real world, but its an easy, understandable test program.
- Phase 2 : We'll add a timer to it. When the doorbell is activated, the "lockRelease" will be turned on for 5 seconds.
- Phase 3 : We'll add crude temperature control. When the temperature is below 68 degrees, we'll turn the heater on, when it rises above 72, we'll turn it off.

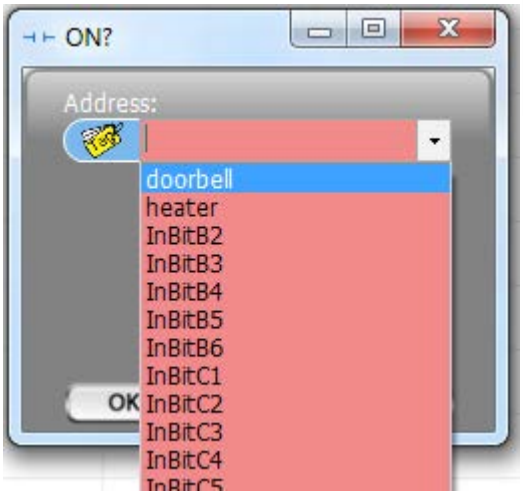
Along the right hand side of vBuilder is the Toolbox. The Toolbox contains all of the program tools necessary to build a program. There are two basic types of tools :

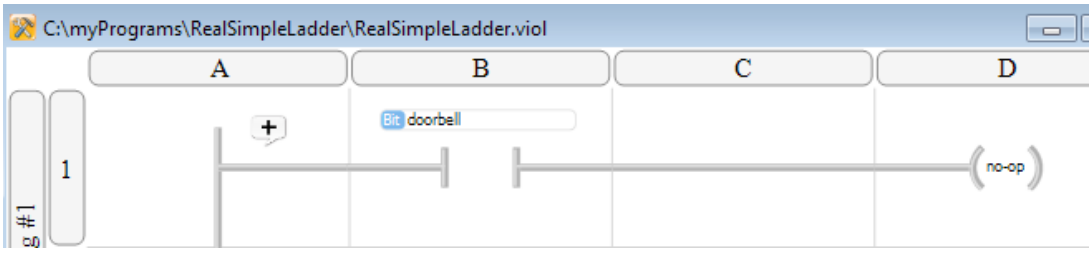
- Contact blocks : contact blocks make a decision whether something is true or not. If it is true, the program will flow through the contact to the next progam block. If it is false, program flow will stop, for the particular rung, at that contact. There are contact blocks for whether a bit is on or off, numeric comparison decision blocks and timer based decisions. The contact blocks are all near the top of the toolbox, below the Wire and above the Coil.
- Process blocks : Process blocks do something. That something can be to turn something on or off, perform a calculation, count, filter, control a stepper motor, PID, etc. The Process blocks begin with Coil and end with Timer.

Lets start our test program by selecting the Normally Open Contact icon. Just click it with your mouse move over to the first rung, next to the power rail and click the mouse again to drop. Your screen should look like the screen shot on the right.



When you place the block, a dialog box will open to allow you to define the decision. If you click the arrow on the right side of the Address box, all of the bits will show up. Select "doorbell", then click OK. Your program will appear as shown below.





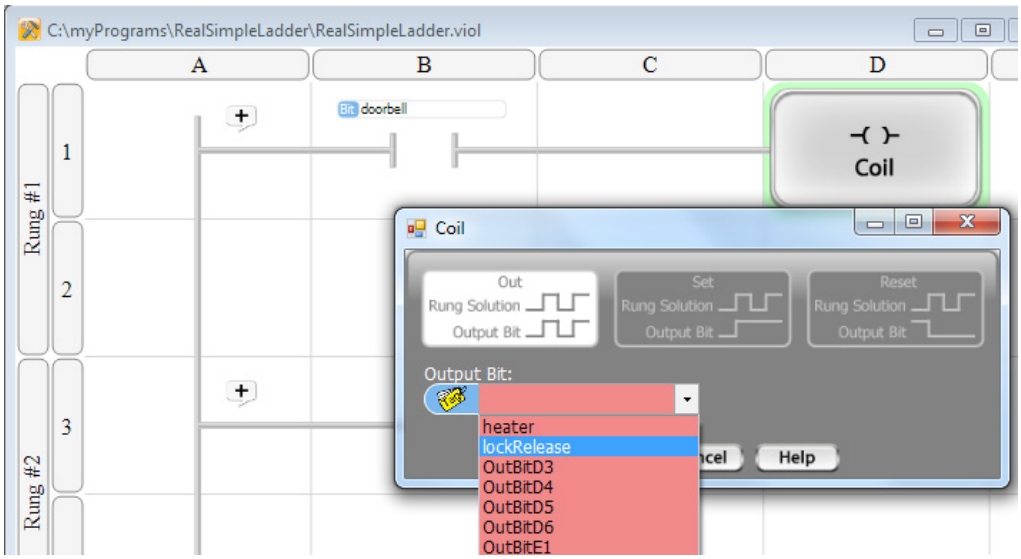
Next, select the Coil icon and place a coil where the "no-op" symbol is, and click again.

When the dialog box pops up, you have a choice of coil types.

» Out : Turns the coil (a tagnamed bit) to the solve state of the rung. If the rung solved true (a completed "yes" or "true" path from the power rail to the coil) the bit will be turned on or 1. If not it will be turned off or 0.

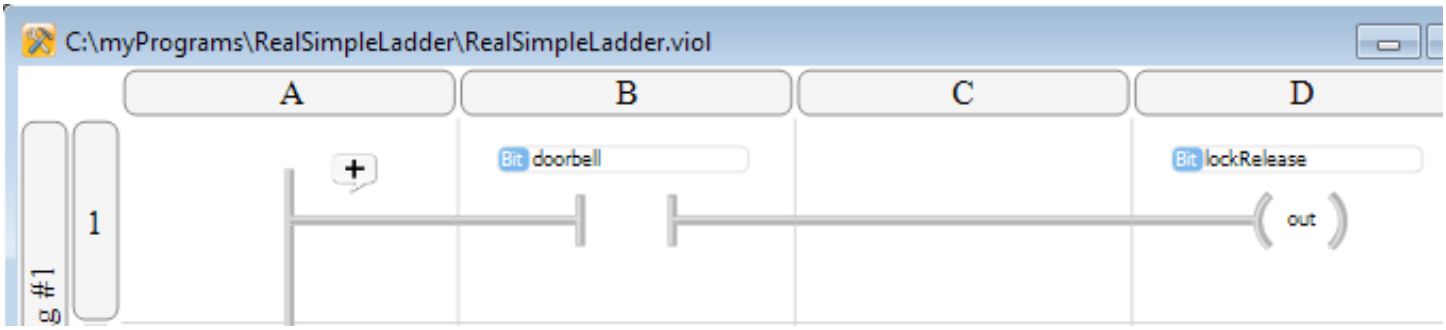
» Set : If the rung solved true, turns the coil (tagnamed bit) on or 1.

» Reset : If the rung solved true, turns the coil off or 0.



The default choice is "Out". Out is what we want, so leave the selection as Out. The next thing we need to do is select the tagname of what we want to "Out". Notice that there is an entry box, labeled "Output Bit". Click the little down arrow just to the right of the red selection box. A list of all available bit tagnames will come up. Select "lockRelease", then click OK.

Congratulations! You have just written the simplest Ladder Logic program that you will ever write. It should look like the illustration below. If so, we're ready to try it.



STRONG RECOMMENDATION! The best tool for learning Velocio PLC programming, and doing program development is a Velocio Simulator. They are inexpensive and very convenient. A Velocio Simulator allows you to simulate an application, right at your desk, with switches, potentiometers and LED indicators to simulate actual IO. You can fully develop, debug and have your application programs ready to deploy without waiting to connect to the final hardware. You can simulate operation without any repricussions for a mistake in your program - thereby enabling you to eliminate such mistakes before "going live".

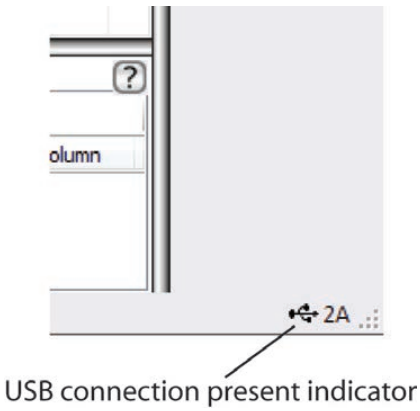
The download and debug portions of all of the examples in this manual assume that you either have a Velocio Simulator, or that you have wired up the equivalent inputs and outputs in order to simulated program operation.

Program Download and Debug

Notice the icon on the top toolbar, that is the Velocio logo.

This icon is the Program icon. When you click this icon, the program will be compiled and downloaded into the PLC.

Before we can program into a PLC, we need to make sure that we are connected to the PLC and the PLC is powered on. The required connection is via a USB cable from the PC to the mini USB port on the main PLC (Ace or Branch unit). If this connection is made and the PLC is powered on, you should see a USB present indicator, as shown on the right, in the lower right corner of vBuilder.

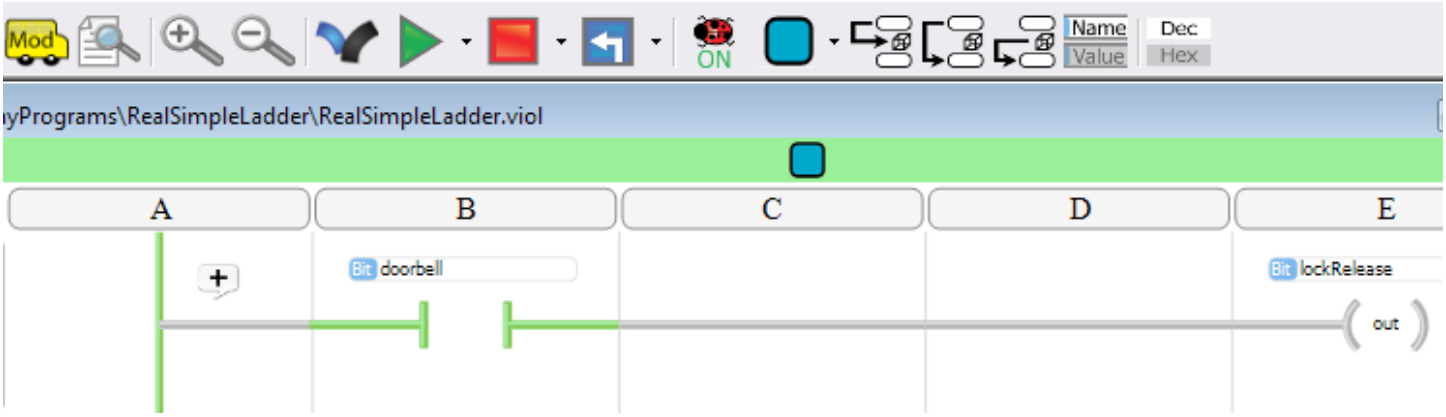


Also notice the lower left hand corner of vBuilder. When you press the Program icon to compile and download, a message saying "Programming" and a progress bar will appear. With such a small program, this may appear only instantaneously, so be watching as you click on the Program icon. When programming is completed, the status message in the lower left hand corner will say "Stopped"

Click on the Program icon and watch the download.

The top tool bar should now appear as shown below.

To illustrate program operation, click on the green arrow in the top toolbar. That's the Run icon. Also click the ladybug icon. The ladybug turns Debug on and off. When its in Debug mode, the ladybug, and the 5 icons to its right will all be lit up, like shown below. Turn Debug on.

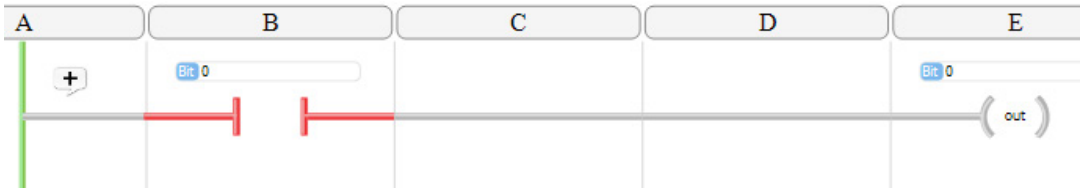


When a Ladder Logic program runs, it starts at the top rung of the ladder and works down. Each rung is solved, one at a time, then the next rung is solved, until the program reaches the last rung. When the ladder logic is completed, the PLC writes the outputs, reads the inputs, handles communications and other administrative tasks. Then it starts all over at the top of the ladder, solving again.

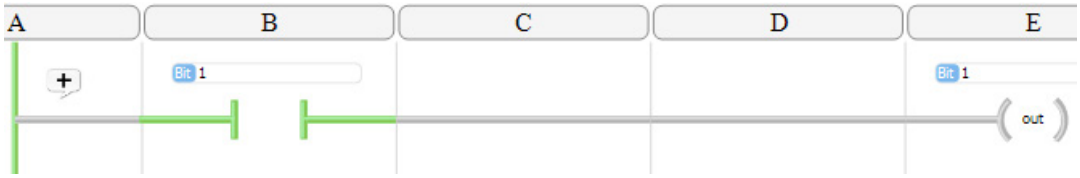
In this case, we only have one rung. The PLC will solve the rung, handle IO, communications, etc., and repeat in a continuous loop, as long as the program is in Run mode.

Notice that if you flip the switch for the B1 input, which you tagnamed, "doorbell", the D1 output (lockRelease) will turn on and off - reflecting the state of the doorbell. Also, you should see the doorbell contact in vBuilder turn between green (on) and red (off). Another thing to note is that the top of the program window turned green. This is an indication that the program is running.

Since you are in Debug mode, the program window will constantly display program status. Click the Name/ Value icon on the top tool bar, so that Value becomes highlighted. Notice that the contact label changed from the tagname "doorbell' to the value of the doorbell switch (B1). When the switch is off, the doorbell value is 0. When it is on, the doorbell value is 1.



The same thing happens with the lockRelease Coil. When the doorbell is on (1), the rung will solve to turn the Coil on (1).



Watch the program window and the D1 (lockRelease) output as you toggle the doorbell (B1) input back and forth. You will see that the program window reflects what is happening with the input switch and output LED.

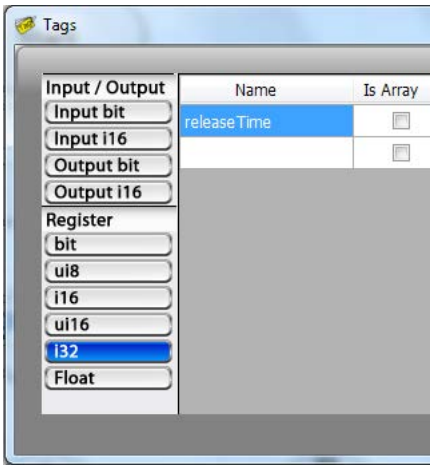
A one rung program is not long enough to illustrate other Debug functions, so let's go on the Phase 2 of our tutorial program.

Phase II Program Entry

In phase 2, we'll change the program to one that turns on the lockRelease for 5 seconds each time the doorbell is pressed.

Start by creating two new tagnames :

- releaseTime : Timer for timing the 5 second release time. This will be a signed 32 bit integer, since all timers are signed 32 bit integers. To create, select i32 on the left pane, then select the block under "Name", type in releaseTime and Enter.
- doorStep : A variable to keep track of the current state with respect to the doorbell and lockRelease. This will illustrate a very simple "state machine" program. We'll define this as a ui8 (unsigned 8 bit integer number, which can hold a value between 0 and 255).
- initialize : A bit variable that we will use to cause the doorStep to be initialized to 1, the first time through the logic rungs.

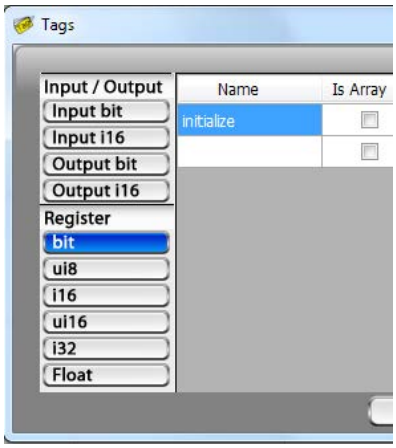


The tagname definitions of these three new variables is shown in the screen shots on the right.

Turn Debug mode off by clicking the ladybug icon and stop the program by selecting the red program stop icon.

This phase of the program is going to have several rungs that are different than the Phase I example. Since what we have now is a trivial program, the easiest way to start is to delete the whole program. We'll start over.

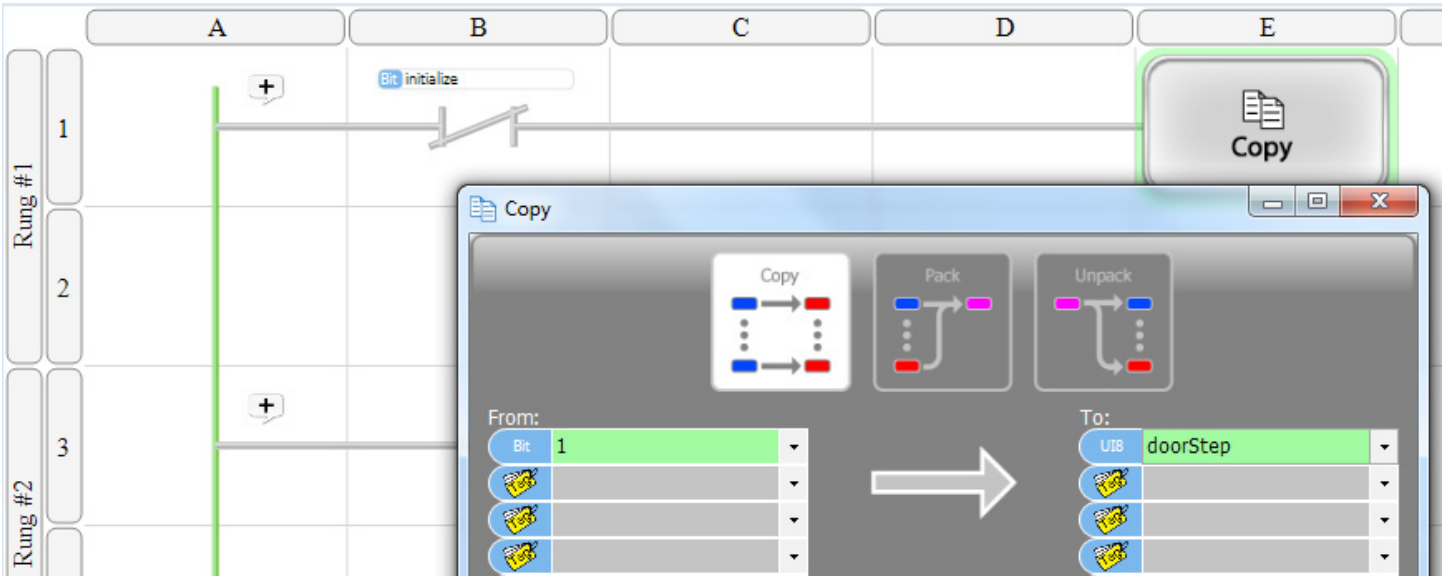
After you have stopped the program and turned Debug off, click the doorbell contact. Press the 'Del', or delete key on your keyboard. The contact should be deleted. Do the same for the lockRelease Coil. You should be back to a blank program.



Click the Normally Closed Contact icon on the toolbar, move over to the first rung and click again to place it. Select your new bit tagname, "initialize" in the dialog box.

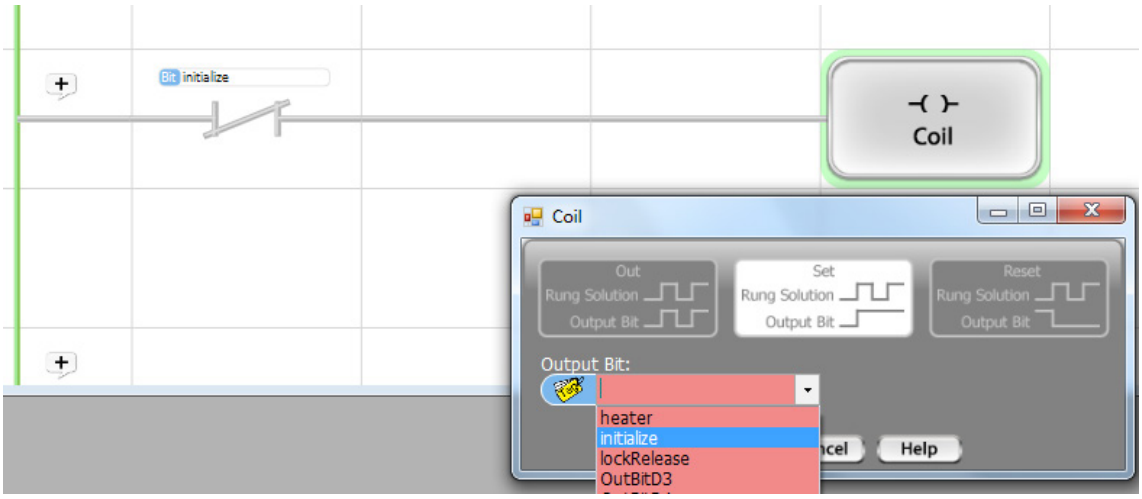
Next, select a Copy block from the Toolbox and place it at the end of the first rung. In the dialog box, select the basic copy (just labeled "Copy") Type '1' in the first From and select 'doorStep' as the first To. Click OK.

All data is initialized to zero at program start up. The reason we've placed a Normally Closed contact is that a Normally Closed contact is closed when it's value is 0. So at start up, the rung will solve true and the copy will happen.



Initialization is something that we want to happen only once. So we need to make sure the initialize contact is never closed again. We'll do that with the next rung. Place another normally closed initialize contact on the next rung, then select and place a coil. In the Coil dialog box, select a Set Coil, then select initialize as the Output Bit.

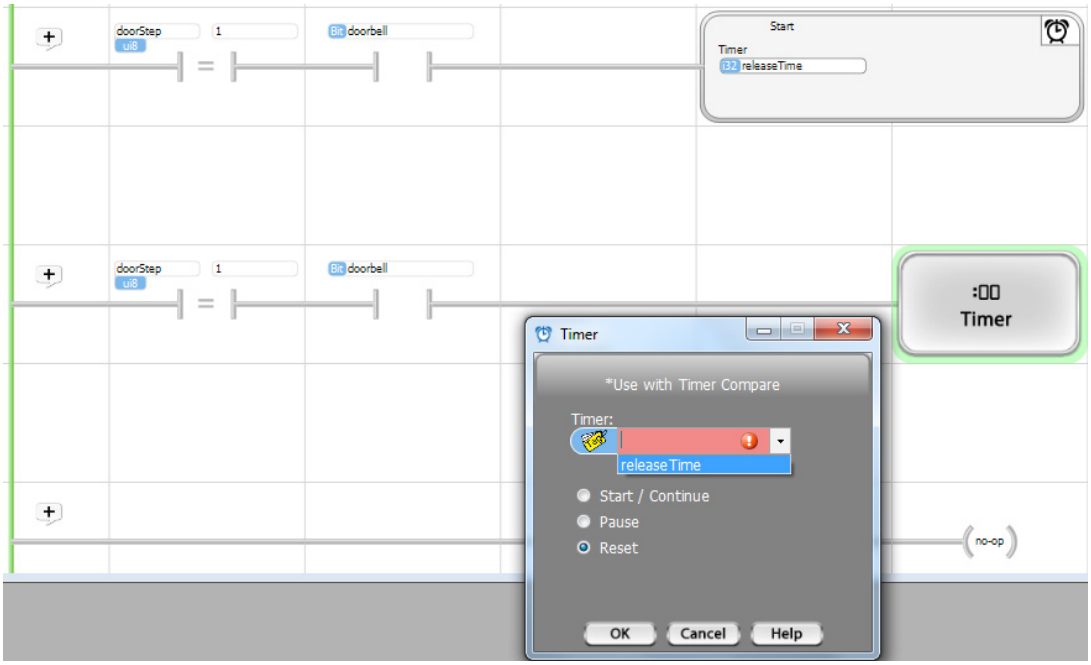
On the first logic solve pass, initialize will have a value of 0. Therefore the rung logic will solve as true. The Set Coil will "set" the tagnamed bit to 1. Once set to 1, it will remain as 1, until something else changes it (generally a Reset Coil on another rung). In this example, we will never change it, so both the first two rungs will solve to false after the first pass. The Copy in the first rung will never again be executed to initialize doorStep. That's what we want.



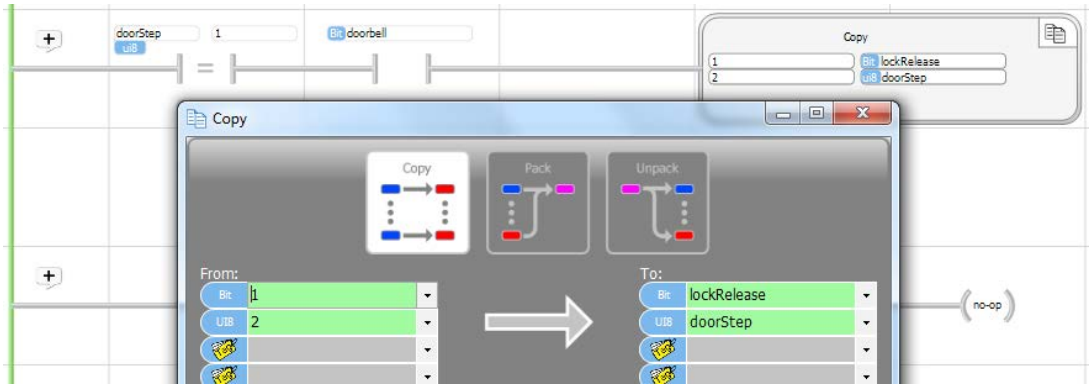
The difference between an Out Coil and either a Set or a Reset Coil is that an Out Coil will always turn the value of the tagnamed bit to the solve value (true (1) or false (0)) of the rung, while the Set Coil and Reset Coil set the tagnamed bit to a 1 or a 0, respectively, only if the rung solves to true.

The first step of our program sequence is going to take three rungs. All of these rungs start with a Equal Contact, set up in the dialog box to compare whether doorStep is Equal to 1, followed by a Normally Open Contact for doorbell. If the program is in doorStep 1 and the doorbell switch is on, the following logic will execute.

When the doorbel is sensed, we need to turn on the lockRelease, start the timer at a value of 0 and advance doorStep to 2. Create two rungs, as shown on the right. In the dialog box for the Start Timer, selet 'releaseTime' as the Timer and Start/Continue. In the next rung, select 'releaseTime' again and select Reset. The Start Timer makes the timer operation start (or continue). The Reset, initializes the time value to 0.

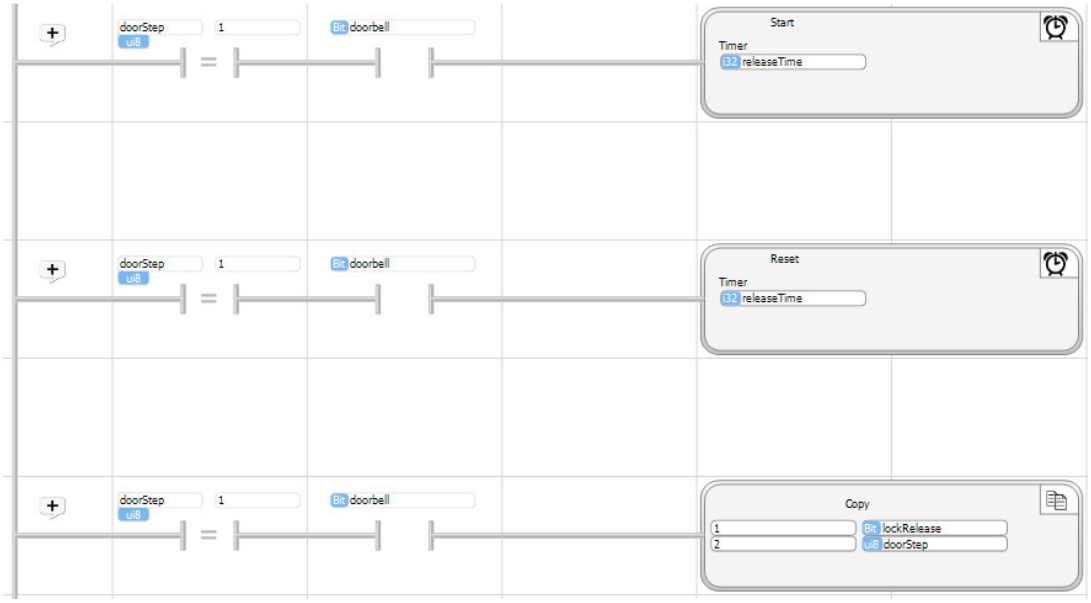


In the next rung, start with the Equal contact to check whether doorStep is 1 followed by the Normally Open doorbell contact. This time select a Copy block to place at the end of the rung. Select the basic Copy. In the first From box, type in a 1. In the first To, select lockRelease. In the next set of .boxes, set to Copy 2 to doorStep.



Your three rungs for doorStep 1 should look as shown on the right. With these three rungs, your program will Start the releaseTime timer, Reset releaseTime to 0, turn on lockRelease and set doorStep to 2.

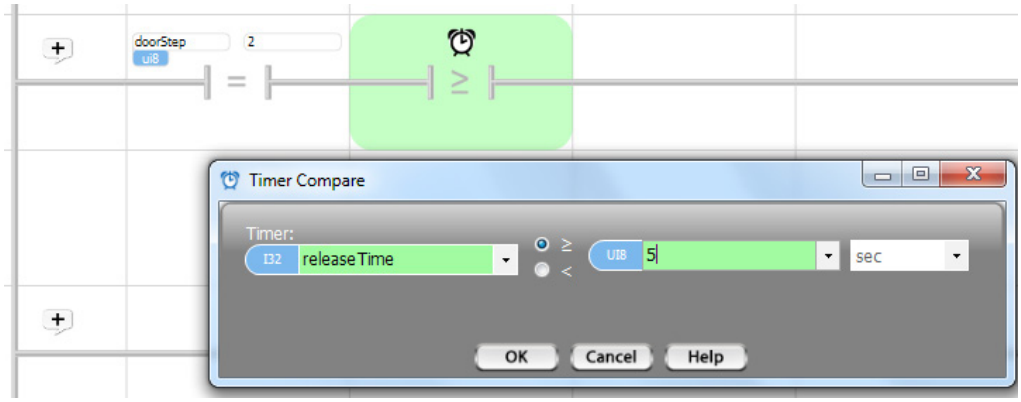
Notice that it is important to put the Copy, that includes setting doorStep to 2, as the last of the three rungs. This is because we want all three rungs to execute. After the doorStep is set to 2, any rungs that check whether doorStep is 1 will solve to false and not execute the associated function block.



In step 2 of your program, you want to wait 5 seconds, then turn off the lockRelease and advance doorStep to 3. That can be accomplished in one rung. Start the rung by placing an Equals contact to check for doorStep 2. Next, select the Timer Compare contact, shown on the right, and place it after the doorStep check. In the dialog box, select releaseTime, leave the comparison selection as greater than or equal to, enter 5 as the value to compare to, and leave the units selection as seconds.

Click OK.

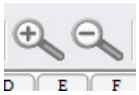
Finish the rung by placing a Copy block that copies 0 to lockRelease, to turn it off, and 3 to doorStep. Your rung, for doorStep 2, should look like the one shown below.



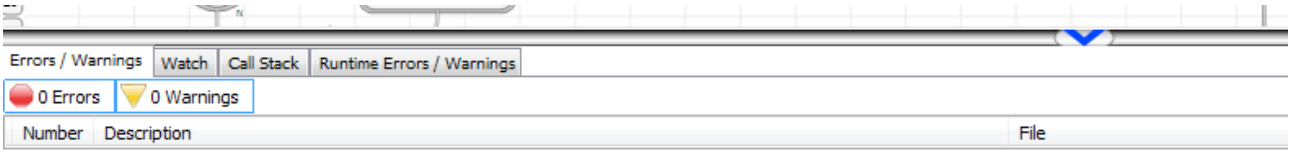
Step 3 just needs to wait until the doorbell switch is turned off, then set the doorStep back to 1 to begin the process all over again. This can be done with one rung. Enter the rung shown below. Notice that this rung will use a Normally Closed contact for doorbell. You want the Copy to execute when the doorbell switch is off.

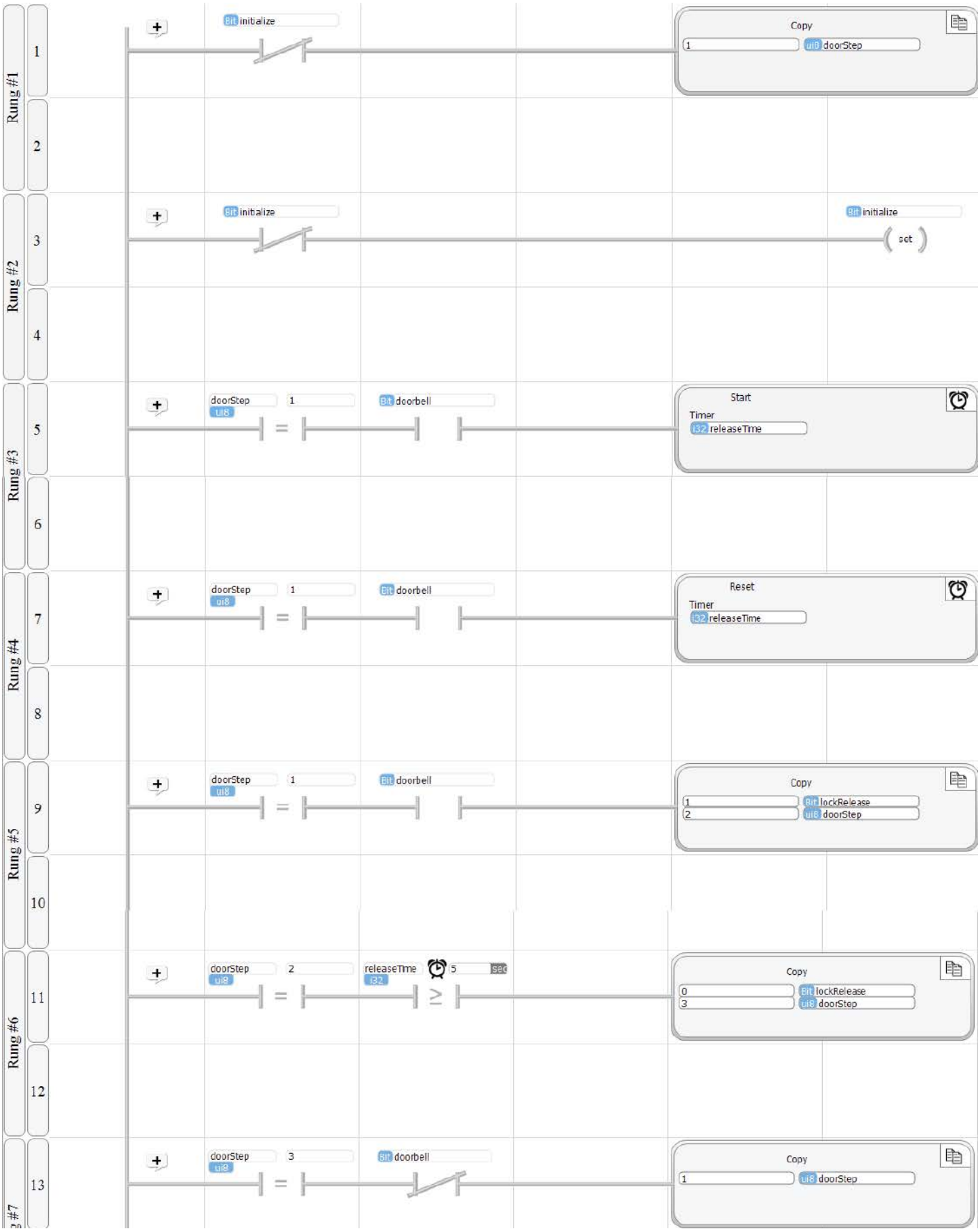
The entire program is shown on the next page.

You can Zoom the program window display in and out using the Zoom icons on the top toolbar.

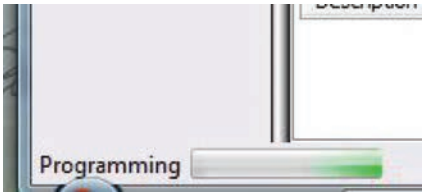


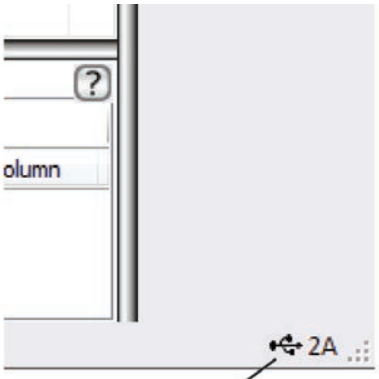
You can also use the View/Hide arrows on the three panes on the right, left and bottom (the blue V).




Rung #1
1
2
Bit initialize
Copy
1
ui8 doorStep
Rung #2
3
4
Bit initialize
Bit initialize
set
Rung #3
5
6
doorStep
ui8
1
Bit doorbell
Start
Timer
i32 releaseTime
Rung #4
7
8
doorStep
ui8
1
Bit doorbell
Reset
Timer
i32 releaseTime
Rung #5
9
10
doorStep
ui8
1
Bit doorbell
Copy
1
2
Bit lockRelease
ui8 doorStep
Rung #6
11
12
doorStep
ui8
2
releaseTime
i32
5
sec
Copy
0
3
Bit lockRelease
ui8 doorStep
13
doorStep
ui8
3
Bit doorbell
Copy
1
ui8 doorStep

If you have a Branch or Ace PLC (a Velocio Simulator is ideal), power it up and connect the USB cable. You should see a USB connection present indicator in the lower right hand corner of vBuilder. If you have that USB connection, click the Program icon at the top of vBuilder. As you click the program button, watch the lower left corner status indicator. It will switch to "Programming" and show activity. This is a short program, so this will happen very quickly, then change to "Stopped" status.





USB connection present indicator

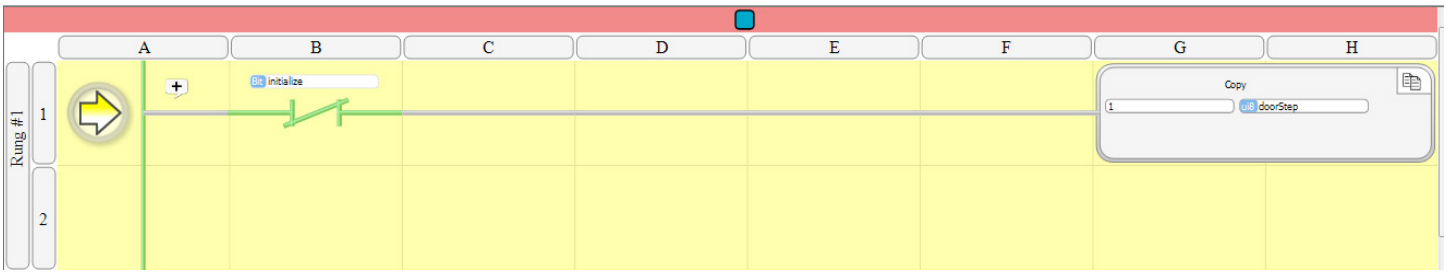
Once you've downloaded, you're ready to run and/or debug. If you've entered the program exactly as directed, it should run.

Notice the Top Toolbar. After you have downloaded your program, the Toolbar will look like this.

Click the icon that looks like a ladybug. When you do, you turn on the Debug mode. You can click it again to turn it off. Leave it on for now. When you turn debug on, the ladybug "Debug" icon will light up and its label will change to "ON". Also, all of the debug function icons will also light up.

Look at the program window. It should have a red strip across the top. The red indicates that the program is stopped. In the red strip is a small blue rectangle. The blue rectangle is the same indicator as in your Setup window. It indicates that the program is located in the main (or in this case, only) PLC unit.

The first rung of your program should appear with a yellow background and a yellow arrow, pointing to it, on the left side. This is a status indication, showing that the program is stopped at this rung. Whichever line is highlighted like this is the next rung to be executed.



Click the Name/Value icon a few times to toggle between displaying tagnames and the value of the data in the tags. Notice that before any rung is executed, the value of the initialize flag is 0 and the value of doorStep is 0.

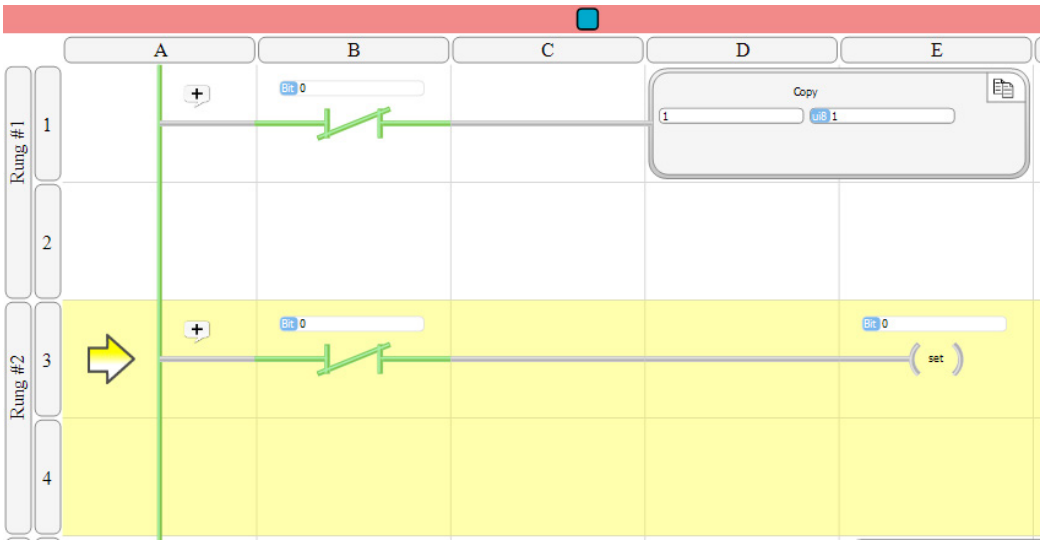
The initialize contact is displayed as green. This means that the path through this contact will solve "true". If there is a path from the power rail, on the left, all the way to the function block on the right (in this case, a Copy), the function block will execute. A Normally Closed contact, whose tagnamed bit value is 0, solves to true. Conversely, a Normally Open contact, with the same 0 value would solve to false.

Since there is only this one contact in the rung, the Copy will execute.

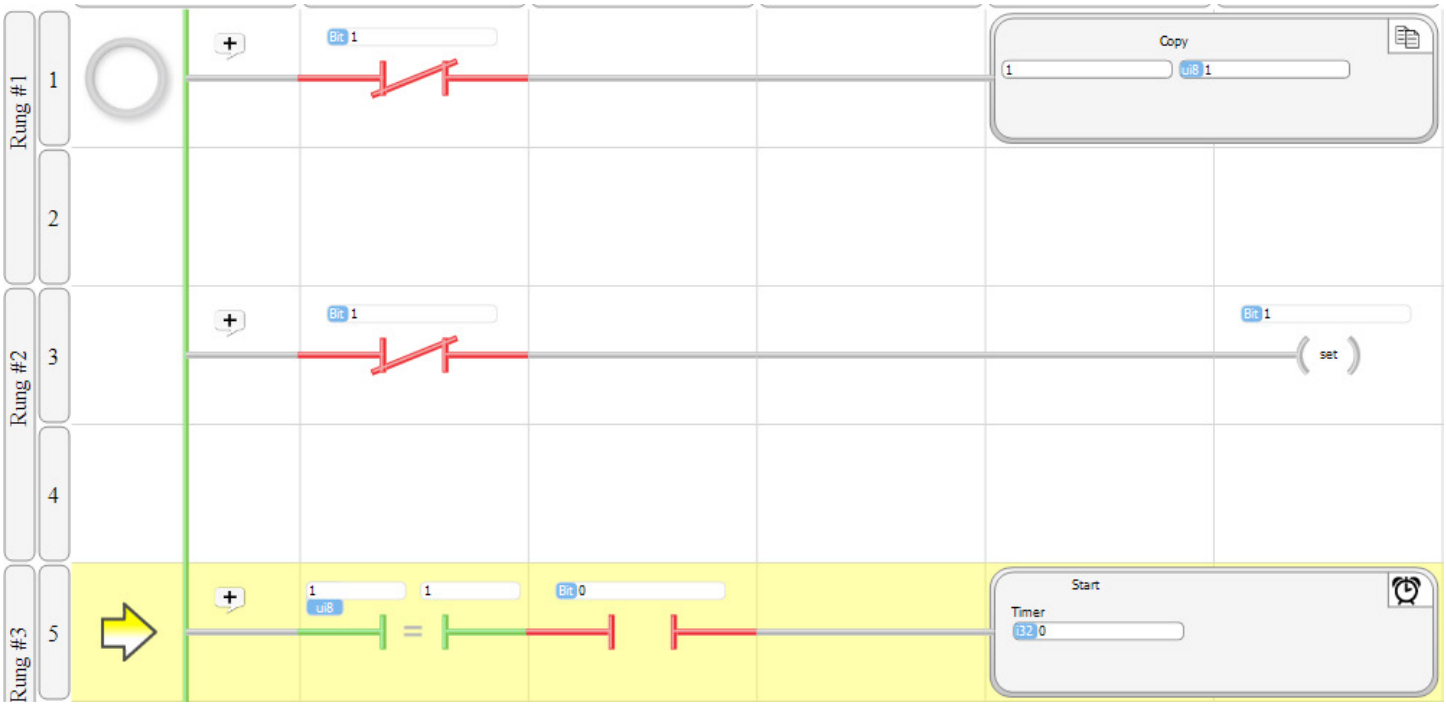
Click the "Step In" icon, on the top toolbar, shown on the right. The program will execute one rung and advance to the next. This is called single stepping. Notice that the yellow background and arrow are now at the second rung.

If you have the Value debug display selected, you should see that the value of doorStep changed to 1. That happened when the first rung solved true and executed the Copy block, as intended.

At this point, doorStep is 1 and initialize is still 0.



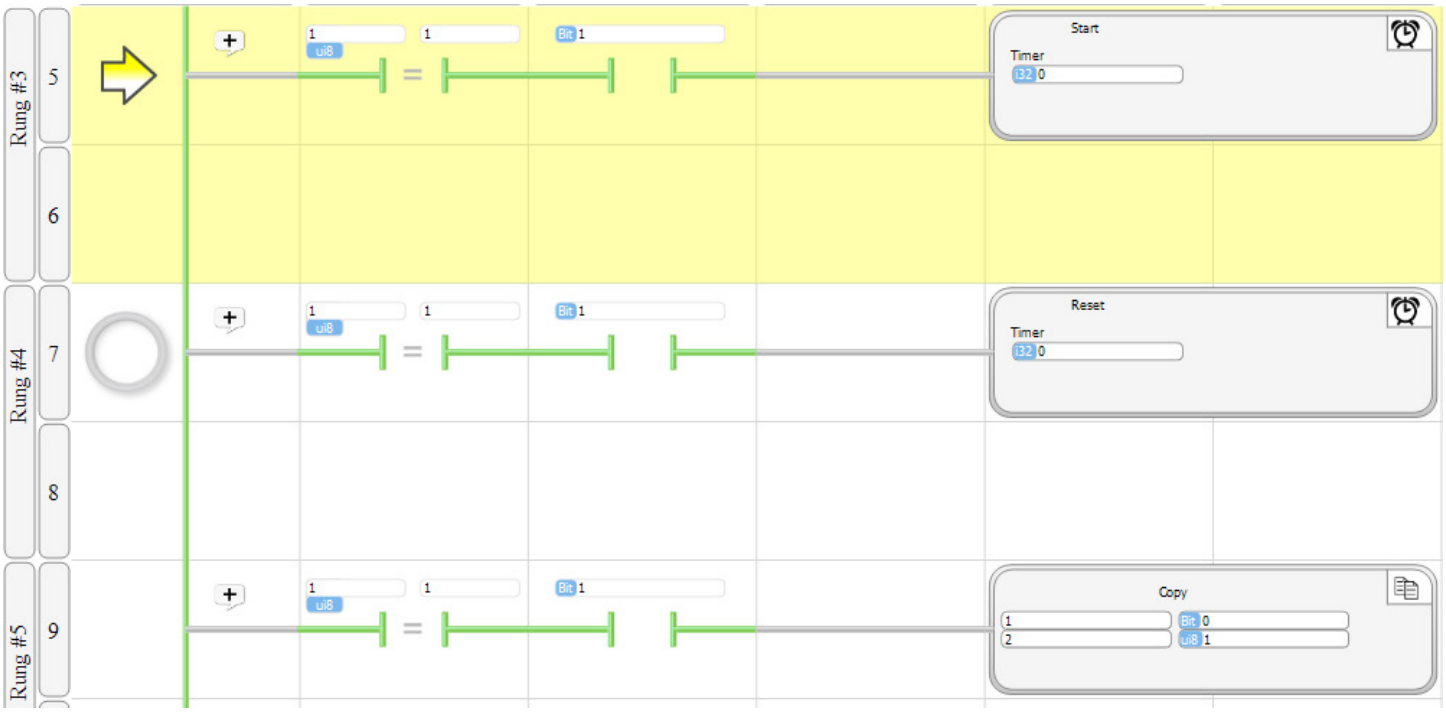
Single Step again. Now, doorStep is 1 and initialize is also 1. The execution of the second rung Set initialize to 1.



Another thing to notice is that the initialize Normally Closed contacts are now displayed in red. That mean that the rungs will solve false through those contacts and not execute the blocks at the end of the rungs. Since you only want to execute initialization once, this is what you want.

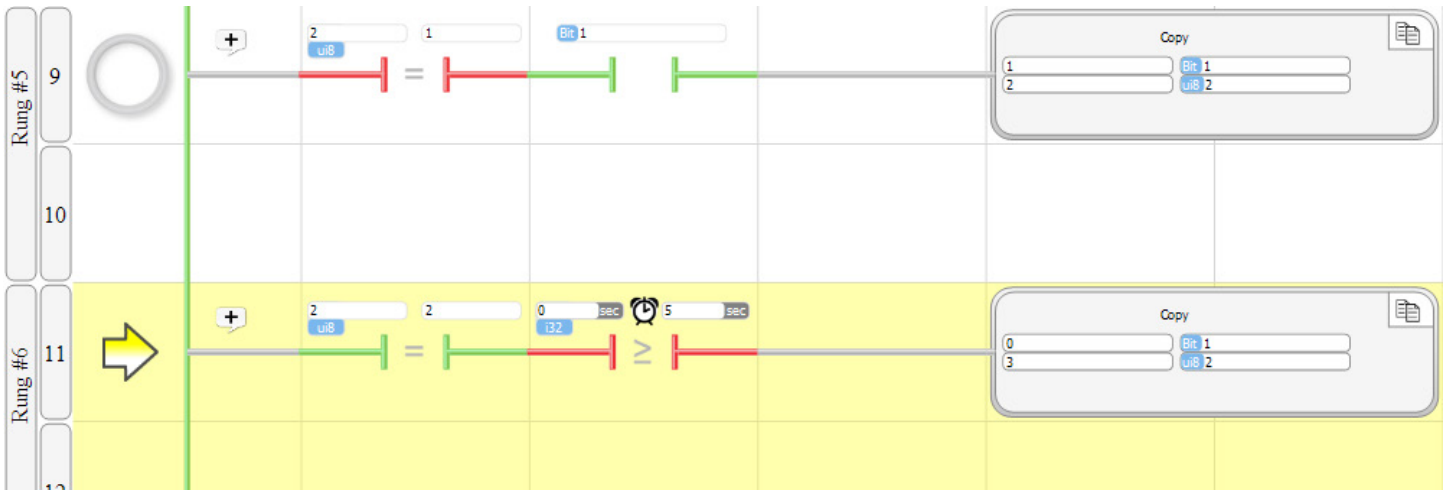
Your program is now in doorStep 1. Looking at the third, fourth and fifth rungs, you can see that the doorStep Equals 1 contact in each rung is green, indicating that that rung will solve true. However, unless you have switched the doorbell switch (B1) on, the doorbell contact is red, indicating that it will solve false. That means that there is not a complete 'true" path from the power rail to the function blocks in these rungs. As long as that is the case, the function blocks will not execute.

Keep single stepping through. You'll notice that until you flip the doorbell (B1) switch on, the program will stay in doorStep 1, waiting for the doorbell to be on. Turn the doorbell switch on and step to rung 3. Now, the doorbell Normally Open contacts



will be displayed in green. There should be a complete green path from the power rail to the function blocks of step 1 (rungs 3, 4 and 5). This means that the the function blocks will execute.

Keep stepping through. Notice that when you step through rung 5, that the lockRelease output (D1) will turn on and doorStep will change to 2. The program window display will now show that doorStep 1 rungs are blocked by the doorStep = 1 contacts and the doorStep 2 rungs will solve past the doorStep check contact.

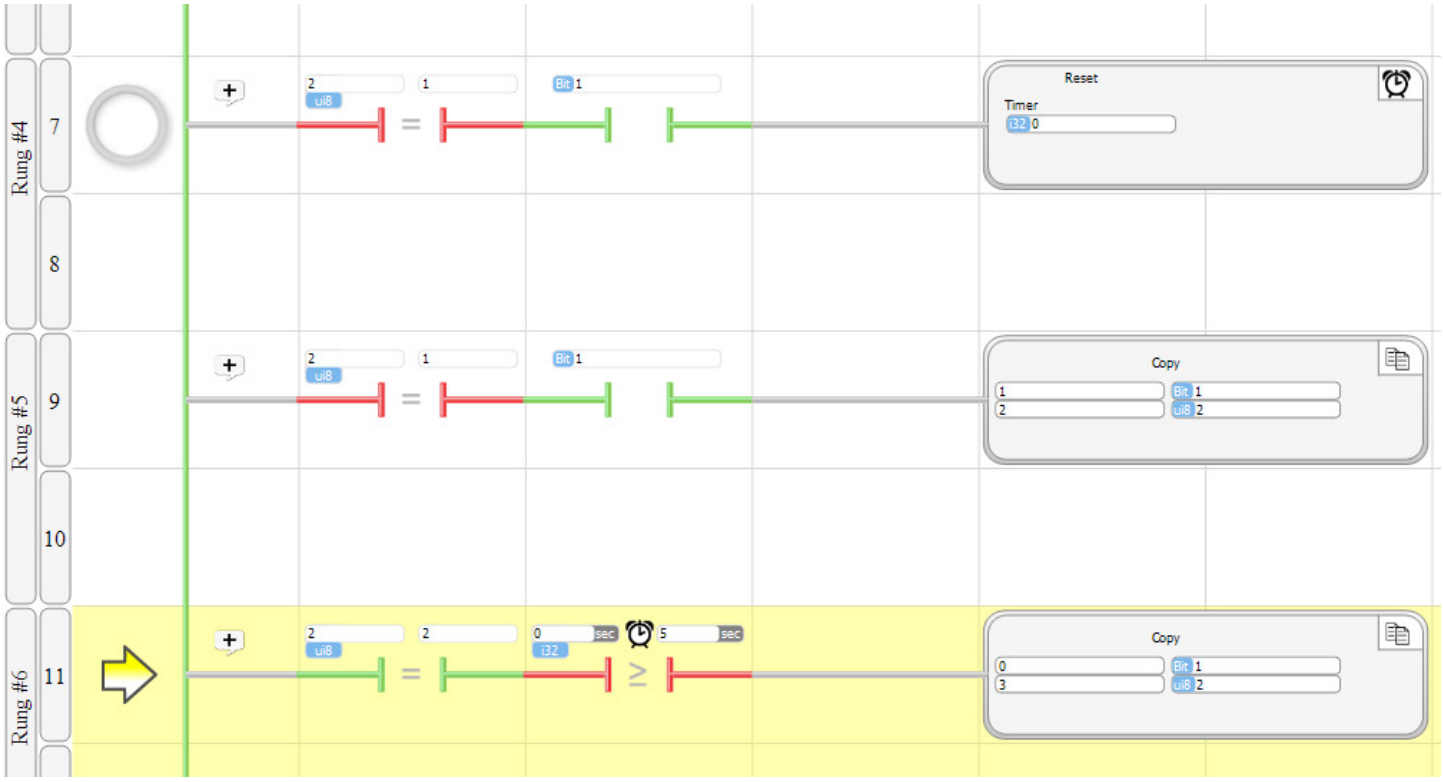


In rungs 3 and 4, the releaseTime timer was started and Reset to a value of 0.

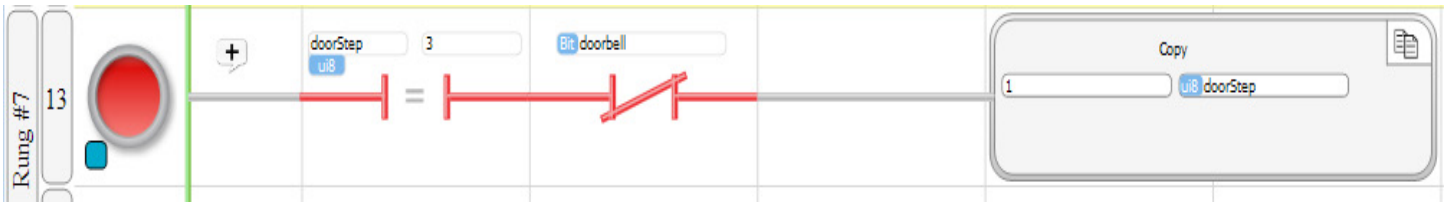
At this point, the debug features are not as effective for Ladder Logic as for Flow Chart programs. Timers don't time while single stepping, so we can't single step until the 5 second timeout times out. In Flow Chart debug, what you would do is put a breakpoint on a block that would be reached at the end of the timeout and run to the breakpoint. Since Ladder programs execute all rungs, a breakpoint to wait out the delay won't work (you would hit the breakpoint on every program pass).

If we had used subroutines in this program, it might be possible to effectively use breakpoints. However, we didin't & we're not ready for those yet. What we can do is run the program at full speed and watch what happens.

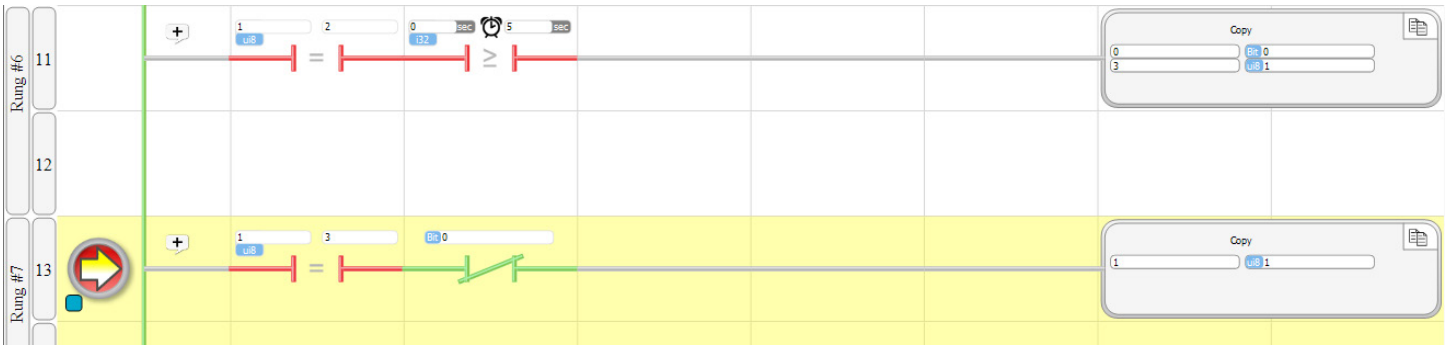
While in Debug, Value mode, you should see that the releaseTime value is 0. The numeric time values of all timers is a count of the number of 1/100's of a second that have elapsed (i.e. 10 millisecond resolution).



You may notice a hollow circle that follows your cursor, moving to whichever rung your cursor is over. This is an indication of where a breakpoint will be placed if you left click. Try it on one of the rungs. Just left click. You will see that the circle fills in, red. This indicates that if you Run the program, it will stop at this rung. Also, a little blue square shows up below it. The blue indicator shows that the breakpoint is in the main PLC's program (which is the only one we have in this example).

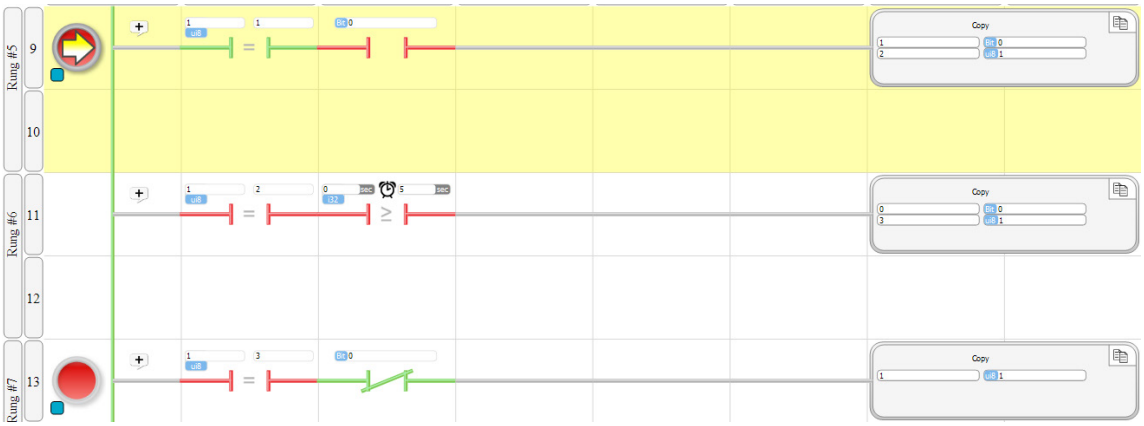


Let's see how this works. Click the Run icon on the top toolbar (the green arrow icon). Your program will begin running and will continue to run until it reaches the breakpoint you just set. This will be instantaneous. If you click the Run icon again, it will appear like nothing happen. Actually the program will run again, back to the breakpoint. It just happens so fast that it looks like nothing happened.



To make it clearer, set another breakpoint on another rung.

Click on Run again. The yellow arrow, which indicates at which rung the program is stopped, will change to the new breakpoint.



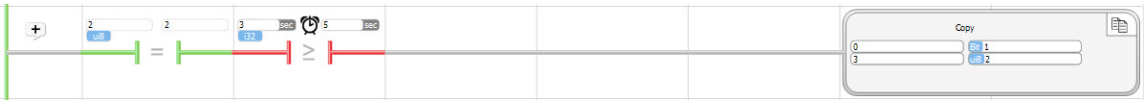
Every time you click Run, the program will run to the next breakpoint.

After you have played with this long enough to get a good understanding, remove both breakpoints by clicking on their rungs again.

Now, lets just run the program and watch it work. Click run.

Flip the doorbell switch (B1). Notice that when you do, the lockRelease output (D1) will turn on and stay on for 5 seconds, then turn off. Turn the doorbell switch off, then on again. If you watch, in the Value mode, a program block that has the releaseTime timer, you will see that the time counts up. The screen shot on the right shows a releaseTime value of 168. That means that 168 x 10milliseconds = 1.68 seconds has elapsed since the doorbell switch was turned on. If you watch it, you will see that when the releaseTime reaches 500 (5 seconds), the lockRelease will turn off.

Look at rung 6, which checks whether releaseTime is greater than or equal to 5 seconds. Here, you will see that the value of releaseTime is displayed and counts up in seconds. In a Timer Compare Contact block, the units displayed will always be the units selected for that block in the dialog box. When you entered the program, you selected the comparision for 5, with units of seconds.



You will also notice that releaseTime will continue to count up. Once a timer is Started, it will continue to time until the program executes a Pause for that timer. Your program doesn't have a Pause block. It doesn't matter. The timer continuing to count up has no effect on anything. If you really want it to stop, you could create another doorStep 3 rung and put a Pause in it.

Flip the doorbell switch back off. This should cause doorStep to be set to 1, where it again waits for the doorbell to be switched on. Each time you flip the doorbell switch on, the lockRelease will turn on for 5 seconds.

You may notice that occasionally, if you leave the doorswitch on for the 5 second releaseTime duration, then flip it off, the lockRelease may start again. This is not a error. It can occur if you transistion the switch slowly. As you do, the mechanical contact in the switch may reach a point where it there is "contact bounce". Contact bounce is when the switch condition transitions from closed to open to closed very quickly. This can happen with a manually actuated switch if there is momentary hesitation in the movement. With a Velocio simulator, you can see this if you have the switch in the on position and lightly press it towards off, but don't actually move it.

Contact bounce can happen in with electromechanical switches, but is less likely with electrically actuated devices than human actuated ones. If there is a chance for contact bounce, it is very easily handled in a vBuilder program. You just place debounce logic or a debounce subroutine in the program. We won't go into that here, but it is covered in an application note.

Continue to play with the operation and view it until you have a good understanding of everything that happens. You can put in breakpoints, single step & run at will. Another thing you can do is Run (clicking the green Run icon) and Stop (click the red Stop rectangle icon). You can go back and forth between Run and Stop.

When you are done, click the ladybug icon to turn Debug off.

Phase III Program Entry

Phase III will modify your program to add temperature control. A temperature transducer is connected to the A1 analog input. You've already changed its tagname to rawTemperature. Click the Tags icon on the top Tool bar, then Input i16, to see.

The temperature transducer measures temperatures between 0 and 100 degrees. It outputs an analog signal between 0 and 5V to the PLC. There is a linear relationship between the temperature reading and the voltage output.

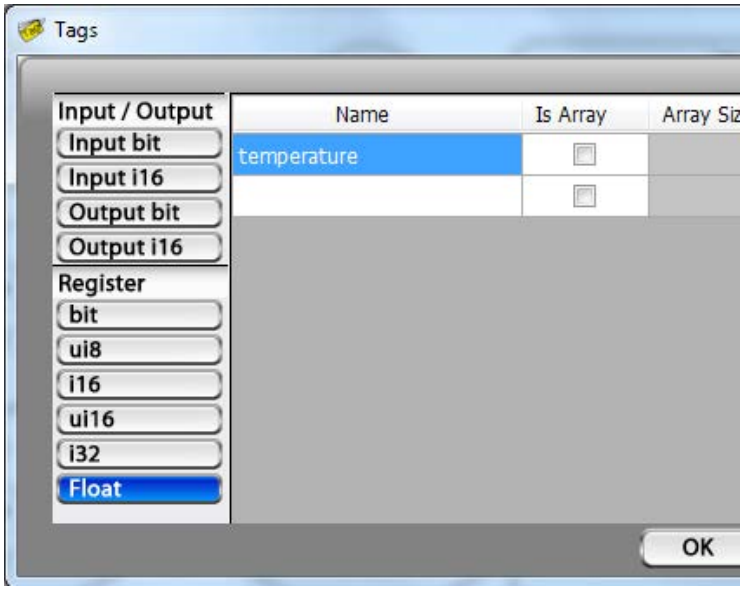
It would be awkward to directly use the rawTemperature values. Remembering that 2866 is 70 degrees is too much brain clutter. We're going to convert the raw reading to temperature in degrees, and use that.

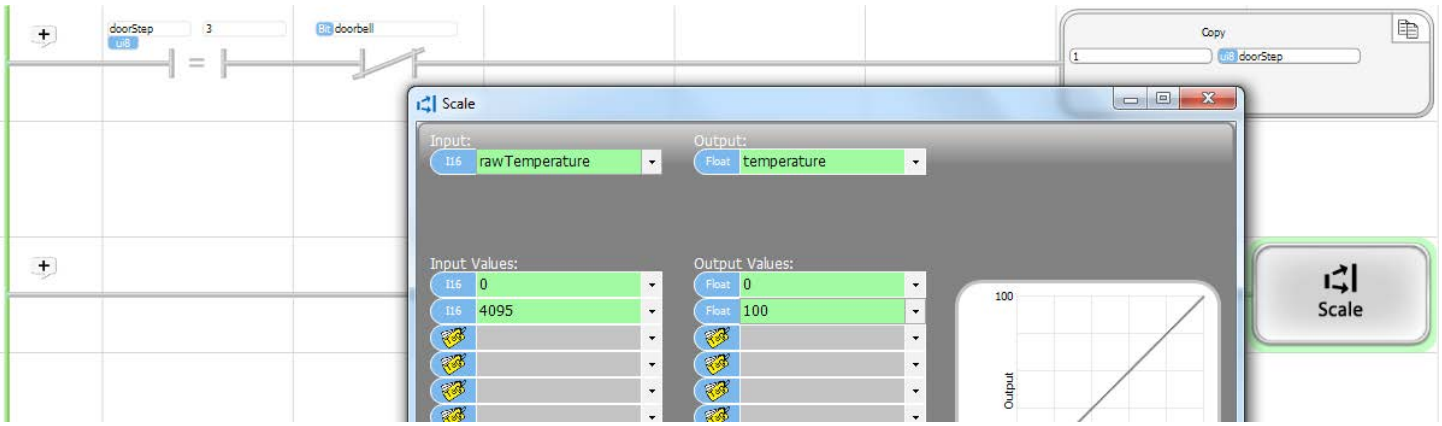
Click the Tag icon, then select Float, under the Register group. Enter "temperature" as a new tagname.

While you have the Tags dialog box open, look at your Output bits. You should see that you already created the tagname "heater" for output D2.

Click OK.

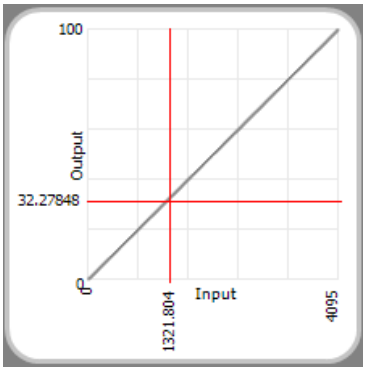
Select the Scale function icon from the Toolbox and place it at the end of the next available rung. In the dialog box, select rawTemperature as the Input and temperature as the Output. For the conversion, place 0 and 4095 as the first two Input values and 0 and 100 as the corresponding Output values. The graph in the dialog box will illustrate how what the Scale function will do.



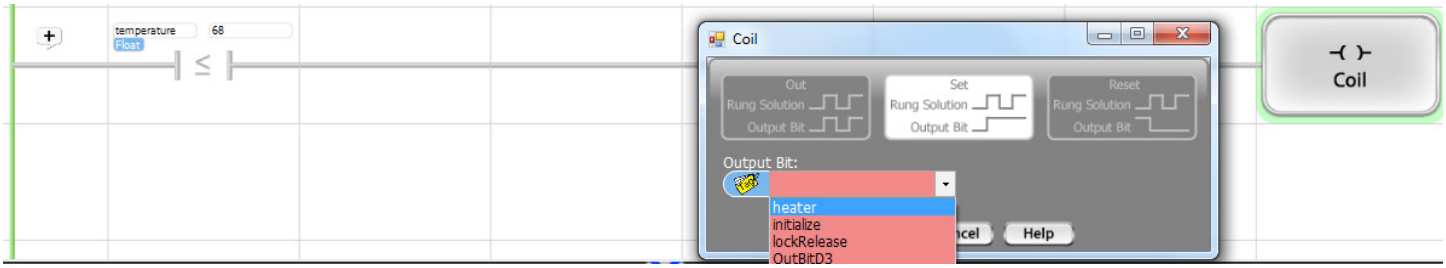


If you move your cursor into the graph area, as you move along the Input (X) axis, you can see the Output values that each Input will translate to.

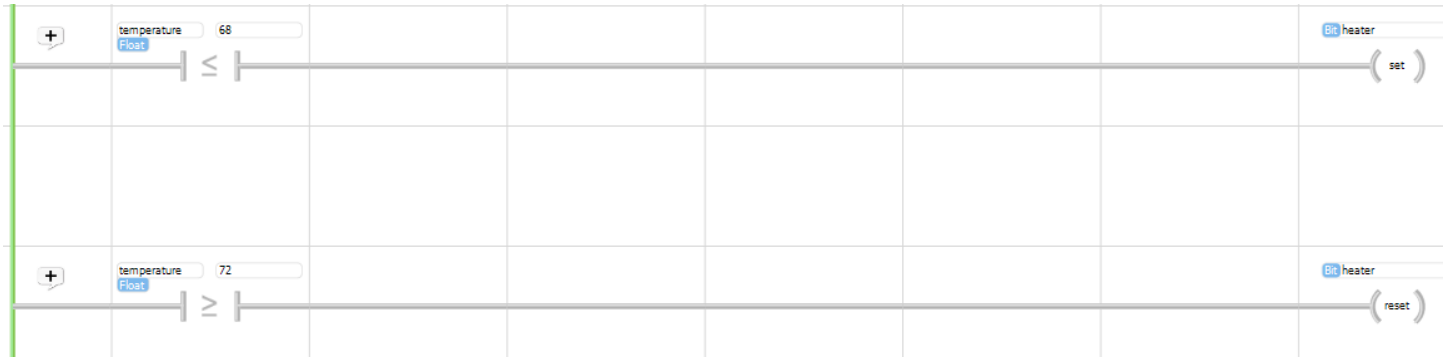
For this rung, we did not put any contacts before the Scale function block. That is because we want to the Scale to always execute, unconditionally.



On the next available rung, place a Less Than or Equal to contact. In its dialog box, set it to check if temperature is Less Than or Equal to 68. At the end of the rung, place a Coil. Select the "Set" coil option and select heater, as shown.



Follow this rung with another one that checks whether temperature is Greater Than or Equal to 72 and, if so, Reset the heater, as shown.



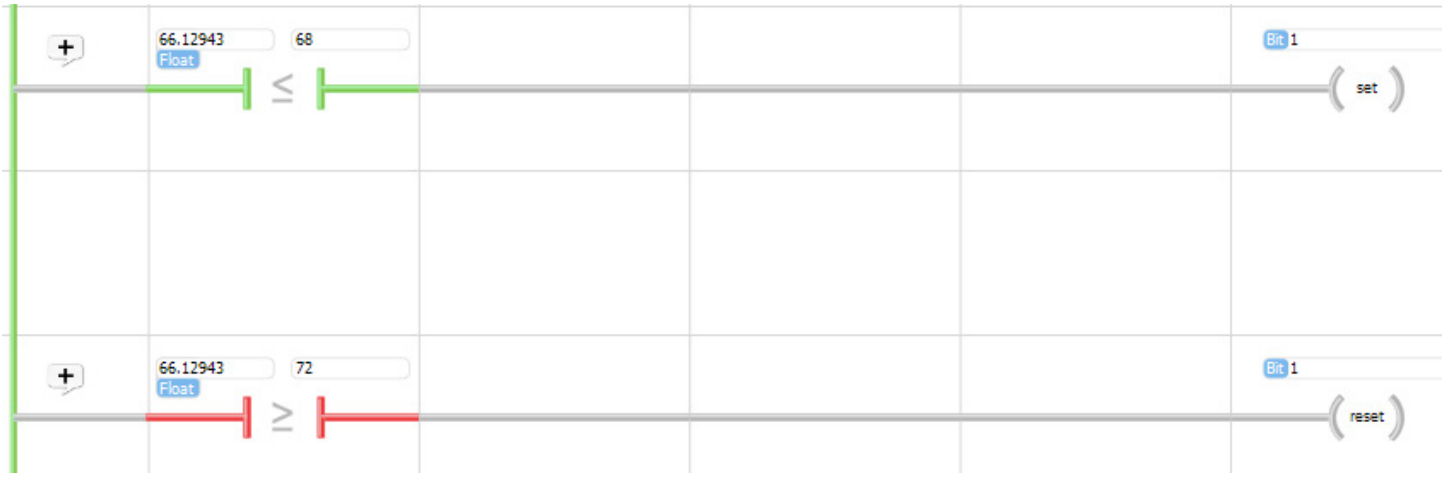
What you now have is a program that will still operate the same with the doorbell and lockRelease. It will also control the heater to maintain temperature between 68 and 72 degrees.

Click the program icon to program the PLC. Put it in Debug mode. Start it by clicking the Run icon. Try the doorbell again. The operation should be the same as previous.

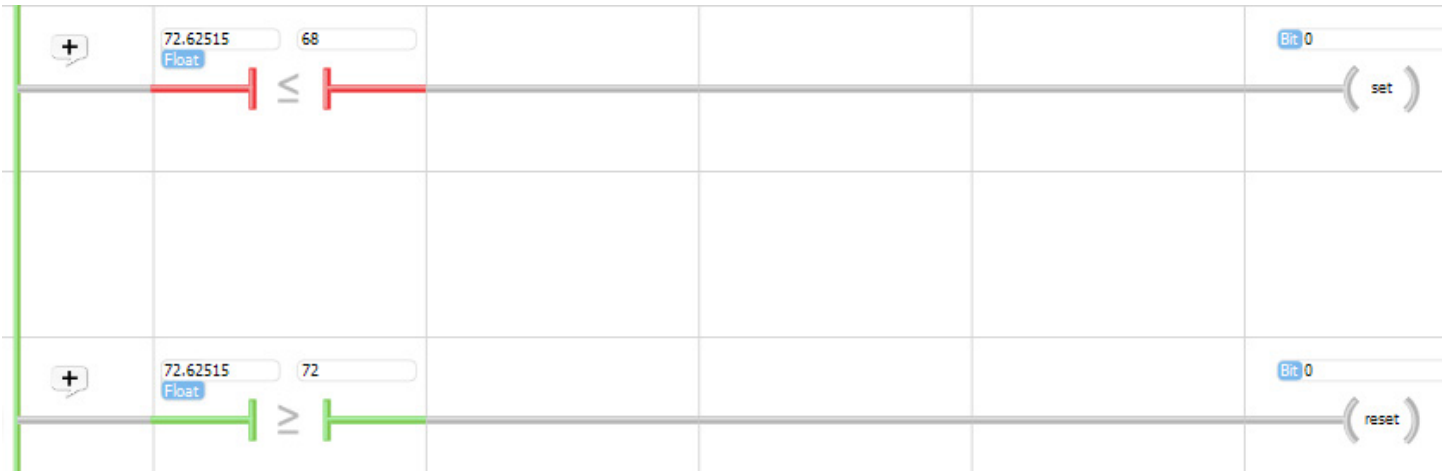
Put the PLC into Debug mode and select to display Values. Look at the rung that you placed the Scale block in. As you turn the A1 potentiomenter, which was assigned the tagname rawTemperature, you should see both the rawTemperature and tempera-

ture values change. The rawTemperature will be between 0 and 4095, which is the raw input range for all analog inputs. The temperature values should be between 0 and 100. The Scale block converts the rawTemperature to temperature. When you entered the Scale block, you set it up for a linear conversion between minimum and maximum values. Twist the A1 potentiometer back and forth and watch the rawTemperature value and the temperature that the Scale block converts it to.

Now look at the last two rungs. Watch as you turn the rawTemperature input potentiometer. When the converted temperature is Less Than or Equal to 68 degrees, the Set Coil will set the heater output bit to on.



As you move the temperature up, you should notice that the heater output (D2) will stay on until the temperature is Greater Than or Equal to 72. At that point the heater output bits will be Reset to 0, turning the output off. It will not turn on again until the temperature again drops to or below 68.



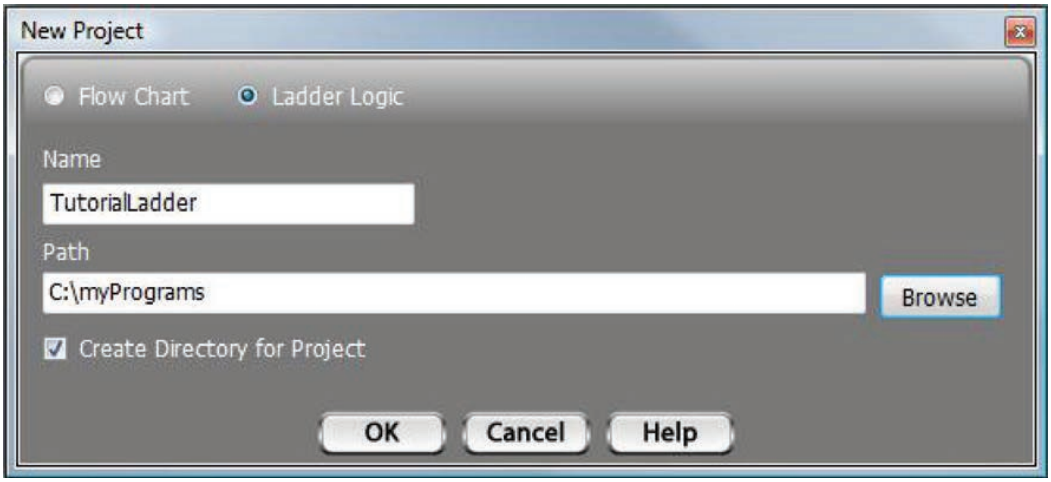
This type of temperature control is called hysterisis control. It is what a thermostat does.

Play with this example until you have mastered it. You can set breakpoints, single step, toggle between Name and Value, start and stop it. It should give you a good understanding of basic Ladder Logic programming.

When you are done, you are ready for the next tutorial. In the next tutorial, you'll learn about subroutines and objects. These are some of the advanced features of vBuilder, which enable powerful functionality, improve programming efficiency and make implementing more complex programs an easier task.

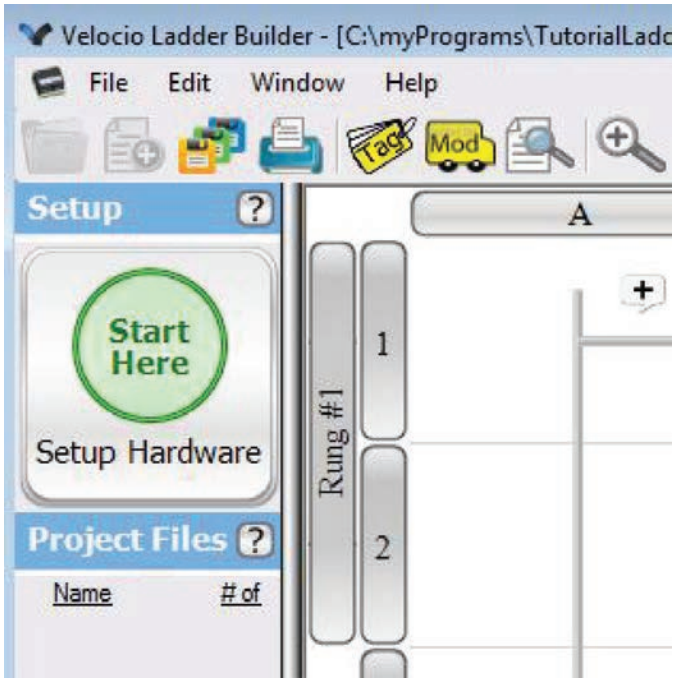
Tutorial 2 : Ladder Logic Implementation

We're going to write the same tutorial 2 example that we wrote using Flow Chart programming, using Ladder Logic. For the people who skipped the Flow Chart section, we will repeat all of the detailed explanations. If you did go through the Flow Chart example, you can skim through a lot of these.



Start the Ladder Logic implementation of our tutorial example by selecting a New project from the File menu. When the New Project window comes up, select "Ladder Logic", give it a name and select the directory, on your computer, where you want to store it. Leave the "Create Directory for Project" checked, so vBuilder will create a project file, under the defined path, with the name of your program. All of the program files will be stored there.

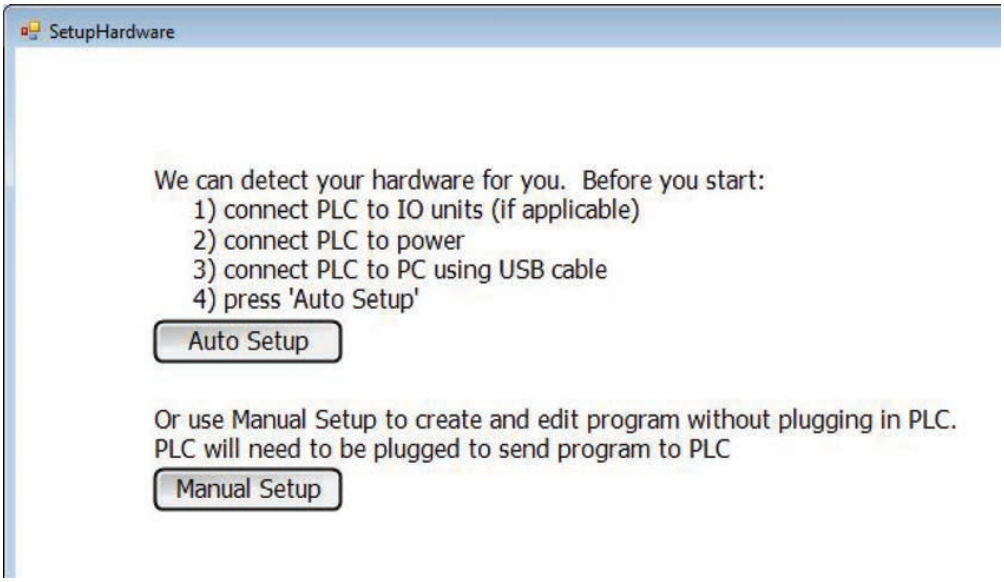
Once you have the chart type (Ladder Logic) selected, Name entered and the path for program storage identified, select "OK". The screen will change to look like the one shown on the right.



Select the big green button that says "Start Here".

A window will pop up, which explains your hardware configuration options. As stated in the pop up window, you have the choice between connecting up your target system and letting Velocio Builder read and auto configure, or you can manually define the target application hardware.

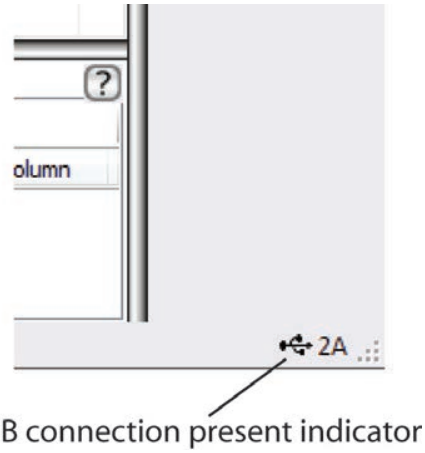
In this tutorial, we will go through both options. However, in order to actually program the PLC, debug and run it, you will need to have either an Ace or a Branch unit.



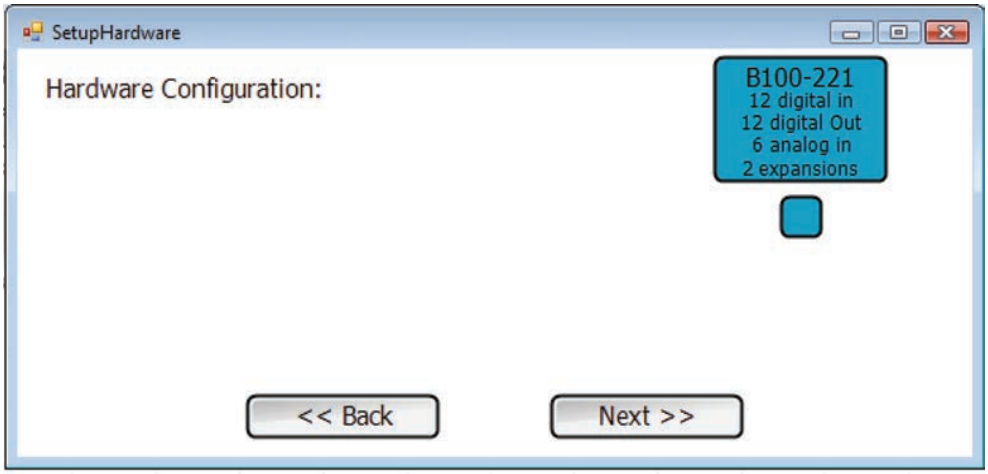
Using Auto Setup

In order for Velocio Builder to automatically set up your hardware configuration, you must have the Velocio PLC connected like you want the system set up, powered on and connected to the PC.

- Connect up your target Velocio PLC system (in this case, simply either an Ace or a Branch module (no expansion modules necessary) - (a Velocio Simulator is ideal)
- Connect a USB cable from your PLC to the Ace or Branch unit
- Power everything on.
- You should see a USB icon, shown on the right, in Velocio Builder's lower right hand corner. This is an indication that a Velocio PLC is connected (and on) to the PC.
- If everything listed above is OK, select "Auto Setup" in the "Setup Hardware" window.



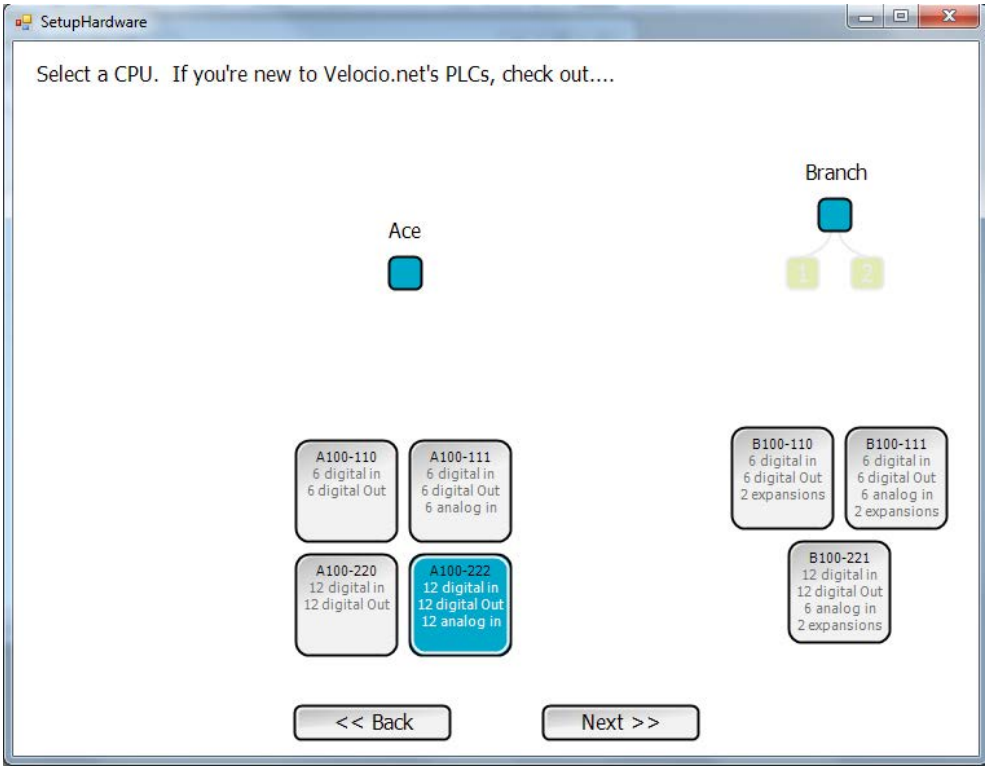
The "Setup Hardware" window should change to display something like what is shown on the right. In this case, it shows that it is configured for a Branch module that has 12 digital inputs (2 ports), 12 digital outputs (2 ports) and 6 analog inputs (1 port). It also has two vLink expansion ports - with nothing attached.



This tutorial can be implemented on any Branch or Ace that has at least 6 digital inputs and 6 digital outputs.

Using Manual Setup

If you don't have the PLC hardware available, you can set the system up manually. Start by selecting "Manual Setup" from the "Setup Hardware" window. The Setup Hardware window will change to something like the image shown on the right.

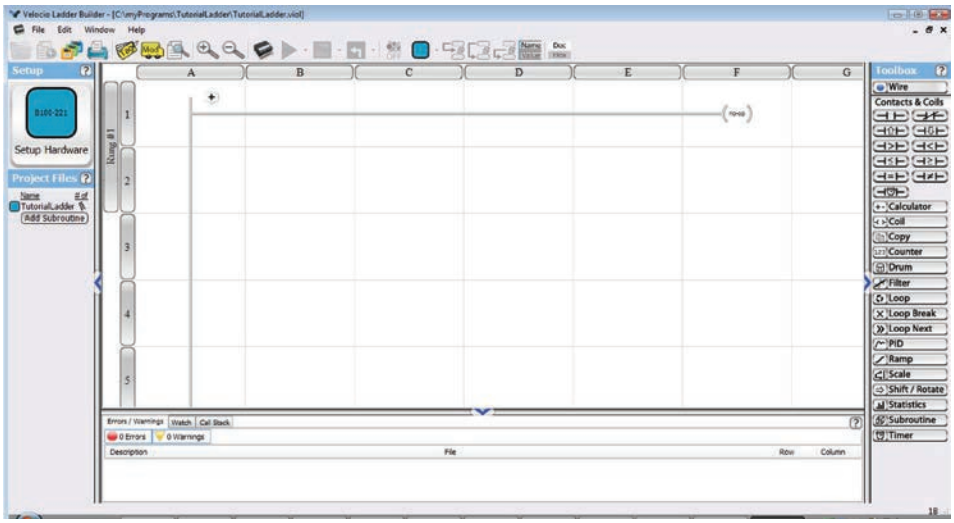


The first step is to select the PLC main CPU. This will be either an Ace, or a Branch unit. The selections are the labeled, gray squares below the Ace & Branch icons. The selections identify both the type (Ace or Branch) and the associated IO configuration. Select the one that applies to the application. In this tutorial example, any of these units will work. The configuration that you select will turn blue, as shown.

After selecting the PLC hardware for the main PLC, you must then select "Next" to continue with the configuration of expansion modules, if any. After all of the PLC modules are selected, there will be additional screens, which allow you to define whether any Embedded Subroutines are to be placed in Expansions (if you configure Expansions) and for Stepper Motion and High Speed Counter signals. We won't be doing any of that - just click the Next & finally Done.

Tutorial Ladder Logic Program Entry

When the hardware setup is defined, your screen should look like the image shown on the right. The Setup is shown on the left side, next to the top left corner of the program entry screen. It shows that we're configured for a Branch with 12 DI, 12 DO and 6 AI. The area just below the hardware setup is the list of project files. The main file is automatically created with the name we defined when we created the new project. In this case it is called "TutorialLadder".

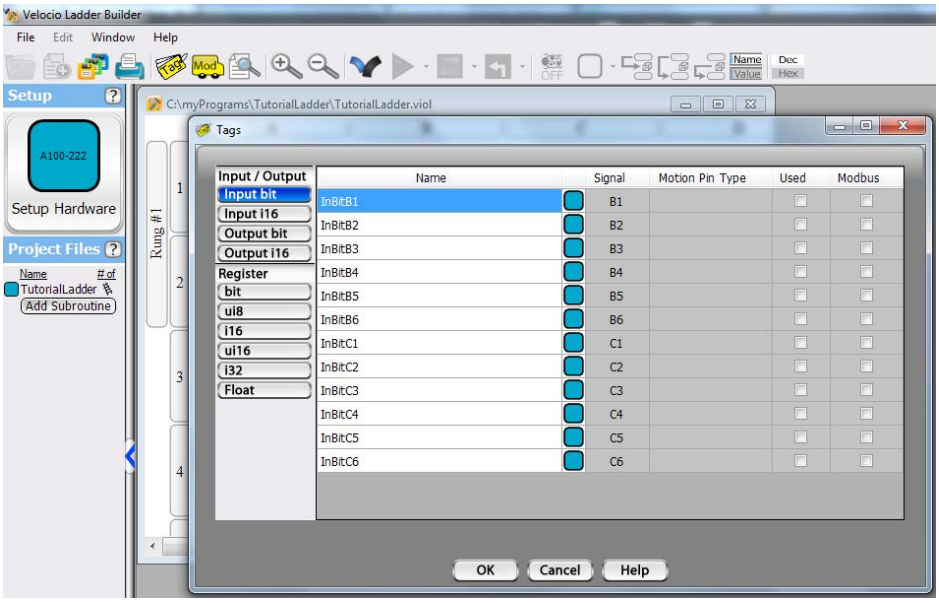


Since we selected ladder logic programming for the main program, a ladder grid is created with one no operation rung. The letters across the top and the numbers along the left side identify "grid blocks" in the program. Move your cursor to various points in the programming grid and look at the reference in the lower right corner (which says 1B on the snapshot). Notice that wherever you move the cursor, the indication at the lower right will indicate the row (1 thru ...) and column (A thru ...) location of the cursor. The row number will limit to the maximum row number that you can currently place an element. In this case, since we have not entered rung one, the position display will not show more than row 2.

Next, take a look at the tags. All data in a vBuilder program is referenced by meaningful tag names. Tagnames are names that you choose that are meaningful to your application. You might name a digital output that controls a recirculation pump, "recircPump1", or an input connected to a limit switch that indicates the "home" position, "home". That way, in your program, your logic is very clearly defined.

Click the icon labeled "Tag". The tagname window will open, as shown on the right. Click the buttons on the left side of the window to bring up tags of various types. The types are data types :

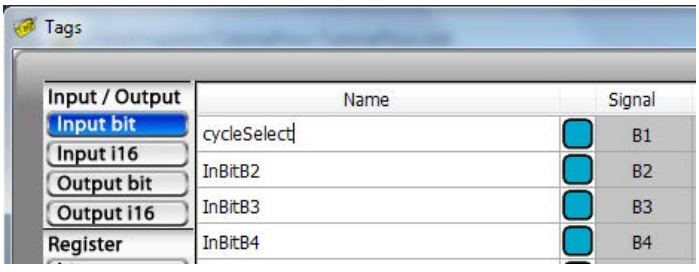
- bit
- char (unsigned 8 bit)
- i16 (signed 16 bit)
- ui16 (unsigned 16 bit)
- i32 (signed 32 bit)
- Float (floating point)



To further clarify the value ranges that the data types can hold :

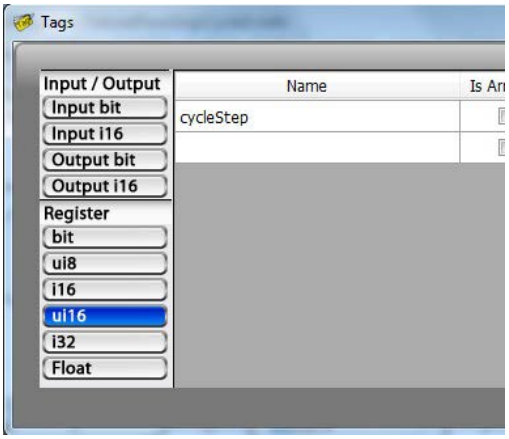
- bit : 0 or 1
- ui8 : 0 to 255
- i16 : -32,768 to 32,767
- ui16 : 0 to 65,535
- i32 : -2,147,483,648 to 2,147,483,647
- float : any floating point number to 32 bit precision.

There are also a few special special case selections for IO for some of these types, listed for convenience, such as input bits.

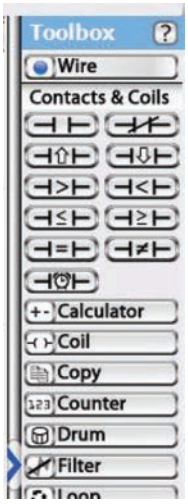


Since this is a new program, most of the tag types show an empty list, i.e. no tags of that type defined. When you configure your hardware definition (which we just did), default tag names are created for the IO. The figure shows the default names for the input bits. Select the name for the first input bit, and change the name to "cycleSelect", as shown. Leave the Output bit names as the defaults, but take a look at them.

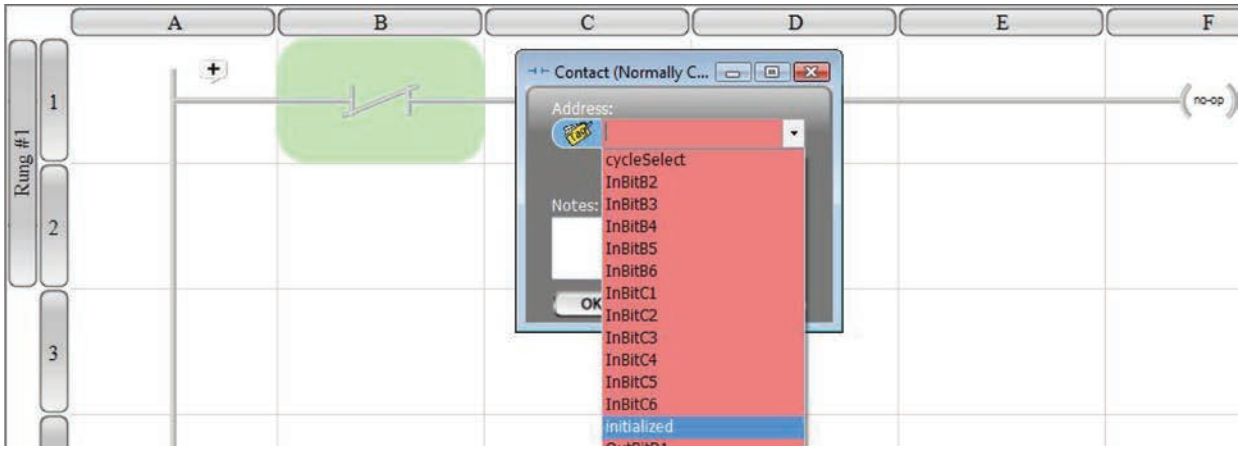
Create an unsigned 16 bit integer variable, named "cycleStep" as shown on the right. Enter a second unsigned 16 bit integer, named "outStates". Select bit types and enter tag name "initialize" then select "OK'. The Tag window will close. You can select it, view and edit any time.



Move your cursor over to the Toolbox, located along the right side of vBuilder. Select a Normally Closed Contact (top contact, right side - if you hoover over the icons, a help label will appear for each one). Left click and hold the Normally Closed Contact while you move your cursor to the left portion of the first rung, then release. [Alternatively, you can click & release the icon to select,then move the cursor to where you want to drop it, and click and release again]

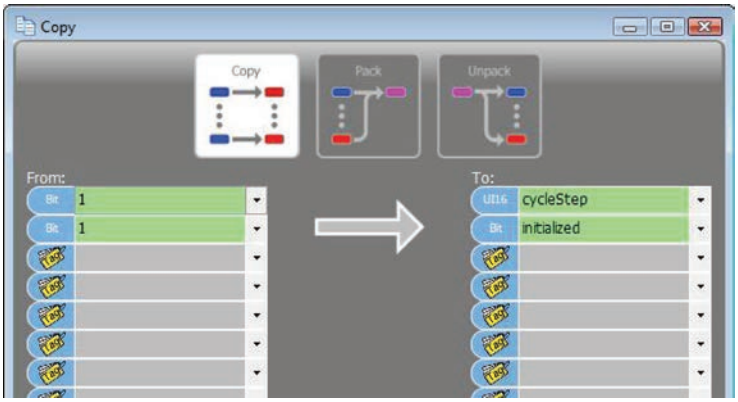


The Normally Closed Contact will drop onto the rung and a dialog box will open. The dialog box allows you to select which bit to associate with the contact. Go to the drop down box and select "initialized",as shown, and click OK.



Next, select the "Copy" icon and place a Copy block on the (no-op) symbol, just like you placed the Normally Closed Contact.

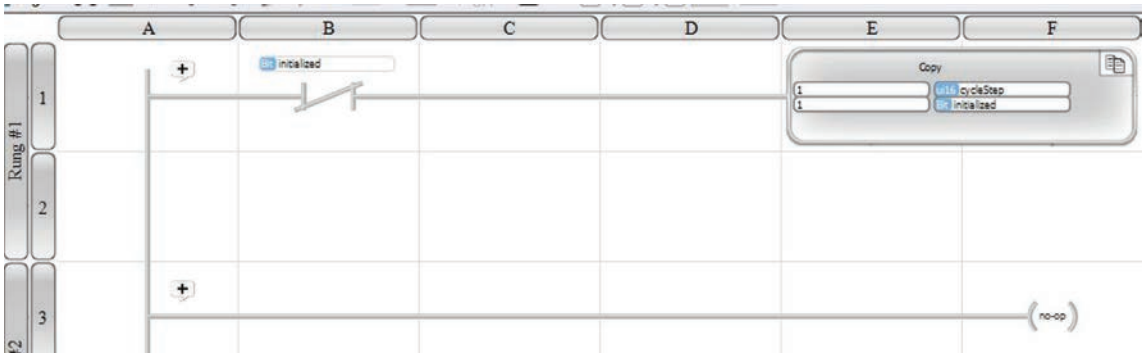
A Copy dialog box will also pop up. A dialog box provides the means for you to define exactly what the block will do.



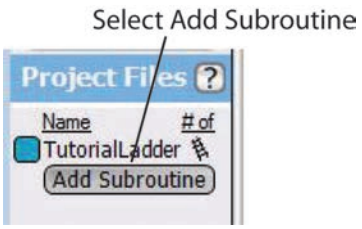
Take a look at the top of the dialog box. You see three selections. The Copy selection is highlighted. The Pack and Unpack options are grayed out. The simple Copy selection allows you to copy data into up to 16 tagnamed variables from other variables or constant values. The Pack copy performs a copy of up to 16 individual bits, into an integer. The Unpack copy moves selected individual bits from an integer into bit tagnames. If you click on pack or unpack, you will see that the selected option becomes highlighted, the other options are grayed out and the dialog box selections change.

Enter '1' in the first "From" box and select "cycleStep" from the drop down list for the first "To" box., as shown on the right. In the second "From" box, type in another 1. In its associated "To" box, select "initialized" from the drop down list. Select "OK" at the bottom of the dialog box.

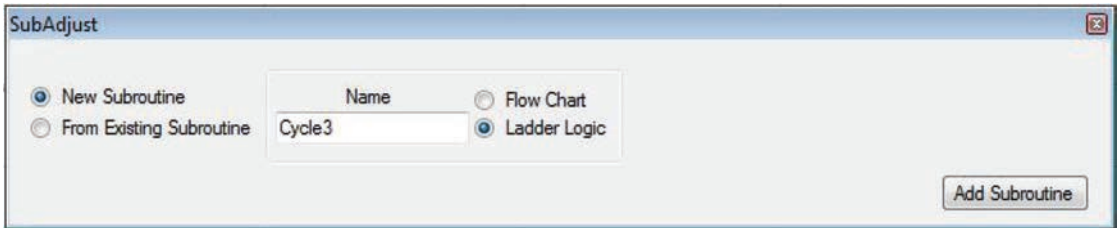
The program will look like the illustration on the right.



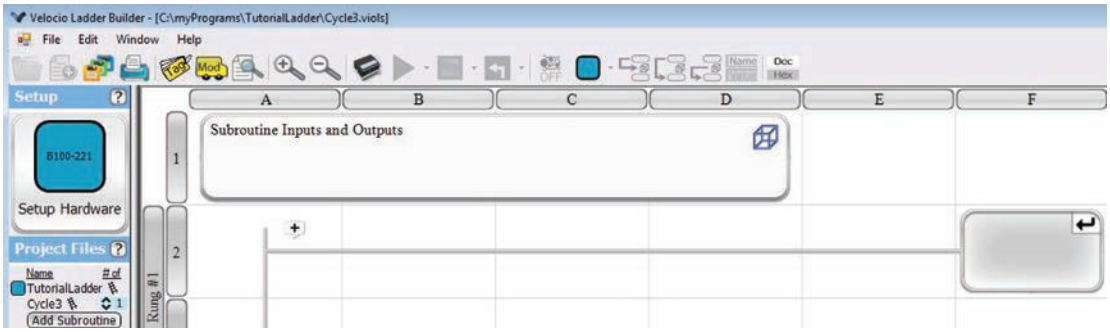
The next thing we need to do is create our subroutine. Start by selecting Add Subroutine on the Project Files list on the left.



A window, like that shown, will pop up. Select New Subroutine. Give it the name "Cycle3". Select Ladder Logic, then select Add Subroutine.

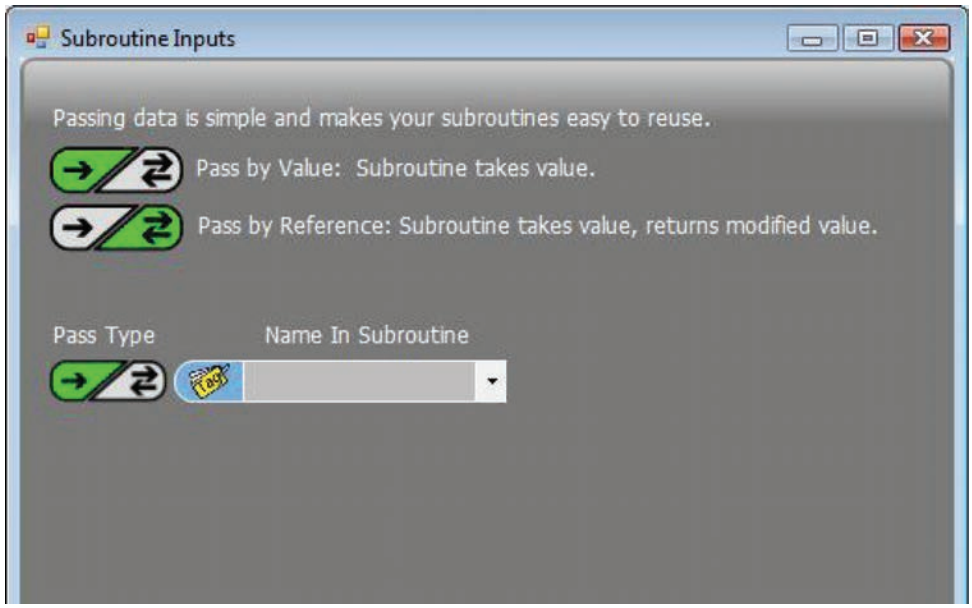


The subroutine window will come up looking like the illustration on the right. Notice that the subroutine, Cycle3, is listed in the Project Files list with a '1' beside it (indicating one object).

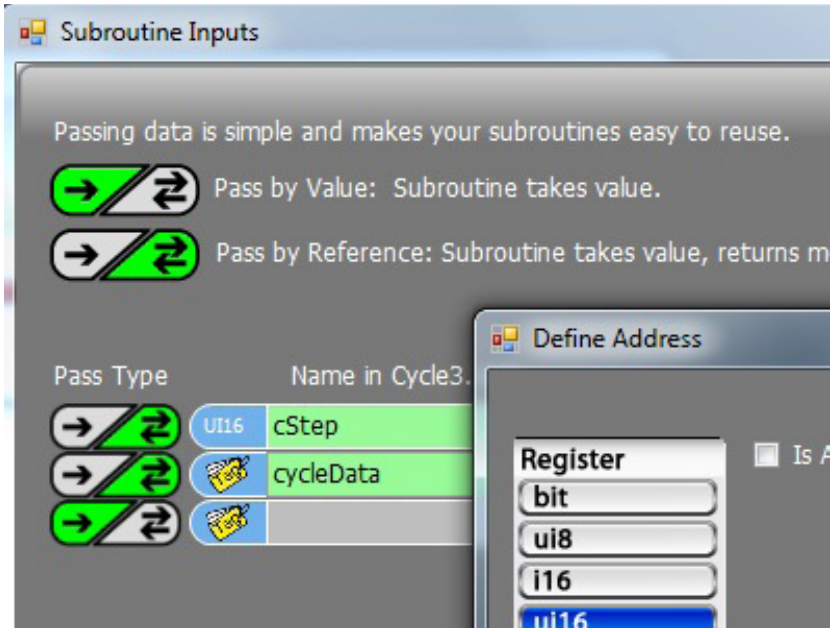


Click in the block labeled "Subroutine Inputs and Outputs" to enter the parameters that are passed in and out of the subroutine.

Take a look at the Subroutine Inputs dialog box. Passing of parameters is explained at the top. There are two pass options for passing to a subroutine object. Pass by value, indicated by a single arrow to the right or in, passes a numeric value into the subroutine variable that you define. Passing by reference, illustrated by the opposing arrows indicating in and out, passes a reference or handle to a variable in the calling routine. In actuality, data and references are passed in to an object subroutine, no data is actually passed back out. By using a reference that is passed in, data in the calling routine can be accessed and changed. Changing data passed by reference is the equivalent of passing data out.

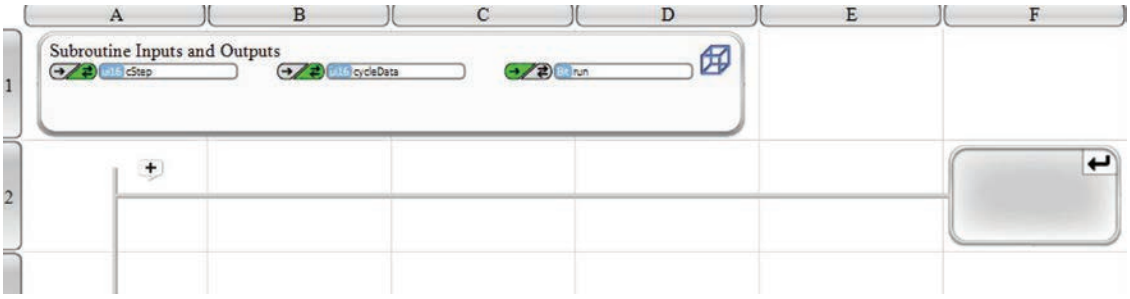


For this tutorial example, define two variables that are passed by reference and one passed by value. To do so, select the double arrows under Pass Type, then click in the Tag box under "Name in Cycle3". Type in the tag name : cStep and cycleData. After you type in the tag name, press Enter. A dialog box enabling the selection of the data type associated with the input tag will pop up. In the first two cases, select ui16 for unsigned 16 bit (numbers between 0 and 65,535). Click OK. Notice that after you OK the data type, the Define Address dialog box will close and the label next to the name will change to the data type selected; in this case ui16. The illustration on the right shows this process. The third entry, not shown will be a "bit" type, named 'run', selected as a pass by value.



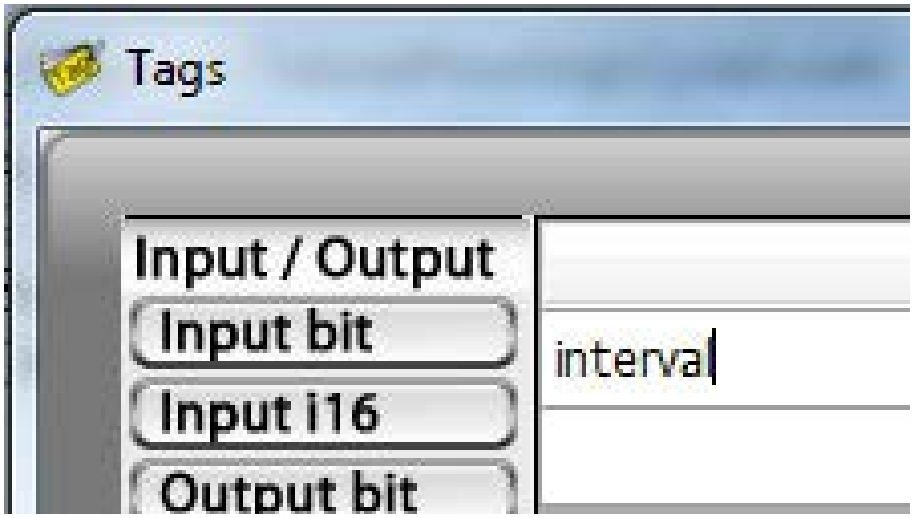
Click OK in the Subroutine Inputs dialog box when all input parameters have been defined.

The Subroutine Inputs and Outputs block at the top of the subroutine window will now show the three items that you defined, including their data types and pass types, as shown on the right.

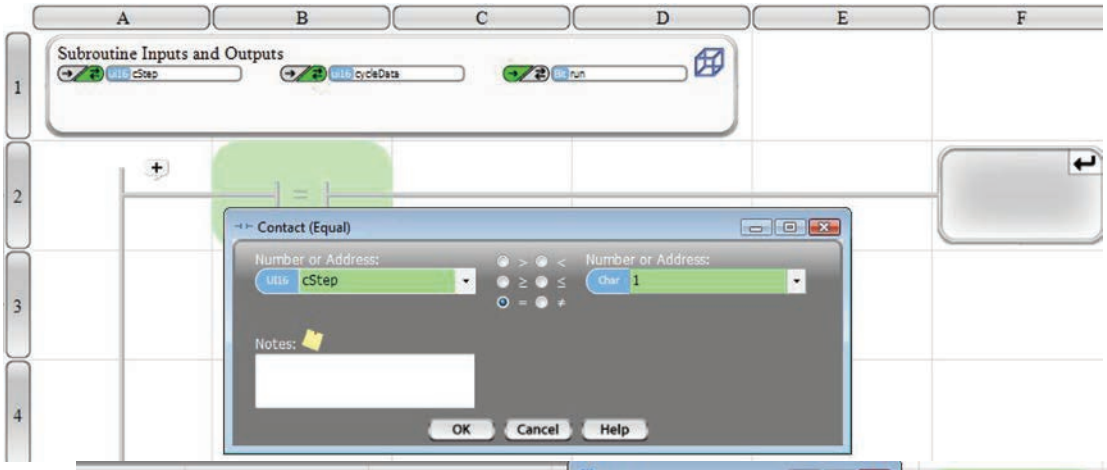


Now we need to define a local tagname variable for a timer that times the interval between changes in the output states. All timers are 32 bit integers.

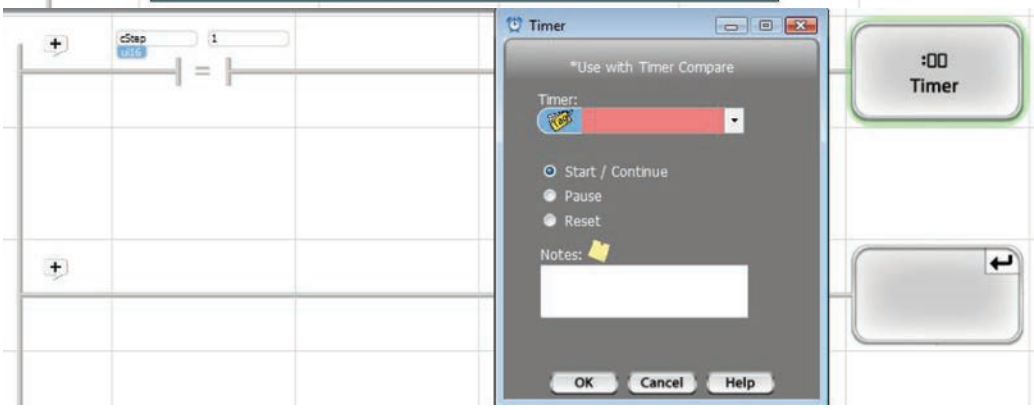
Start by selecting the Tag icon. When the Tags for the subroutine come up, select i32 and enter the tag name "interval". Click OK at the bottom.



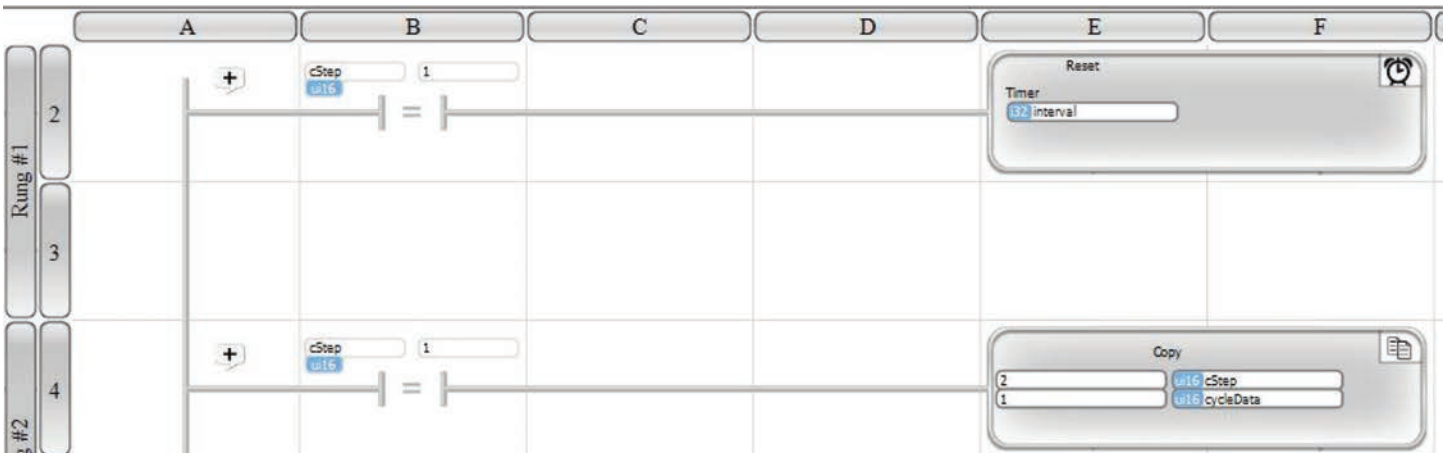
Select the Equals contact icon from the Toolbox and place it in the first rung. That whole group of icons below "Contacts & Coils" are decision blocks. This one checks whether two items are equal. Fill in the dialog box as shown on the right and select OK.



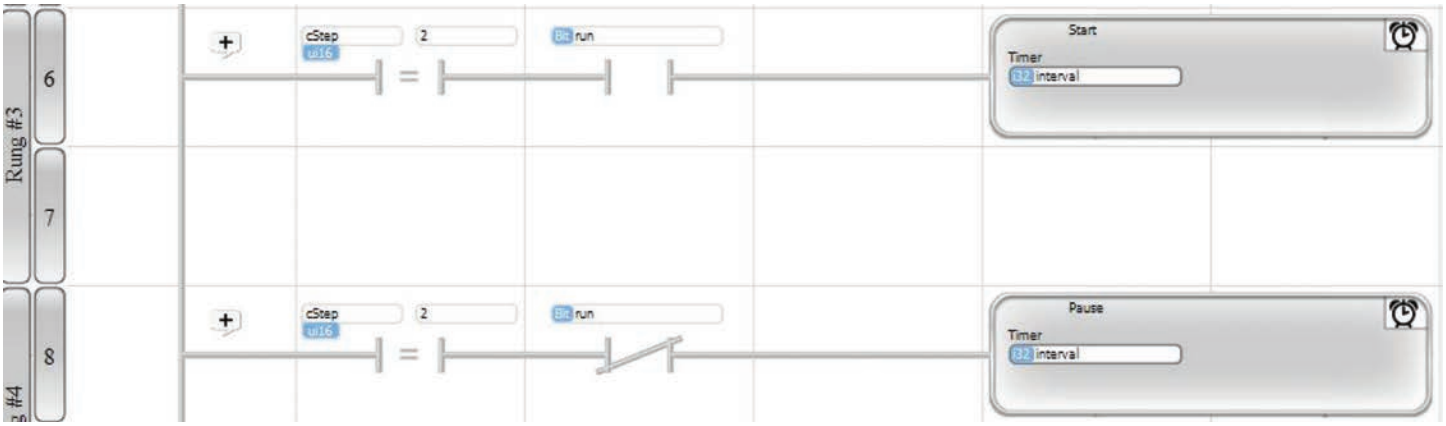
Next, select the Timer icon and drop a Timer block as shown. Select "interval" from the drop down list and select Reset. This will reset the interval timer to 0. Click OK.



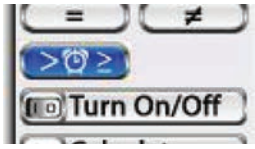
Create a second rung by placing an Equals contact to check for cStep equal to 1, and a Copy block, as shown below.

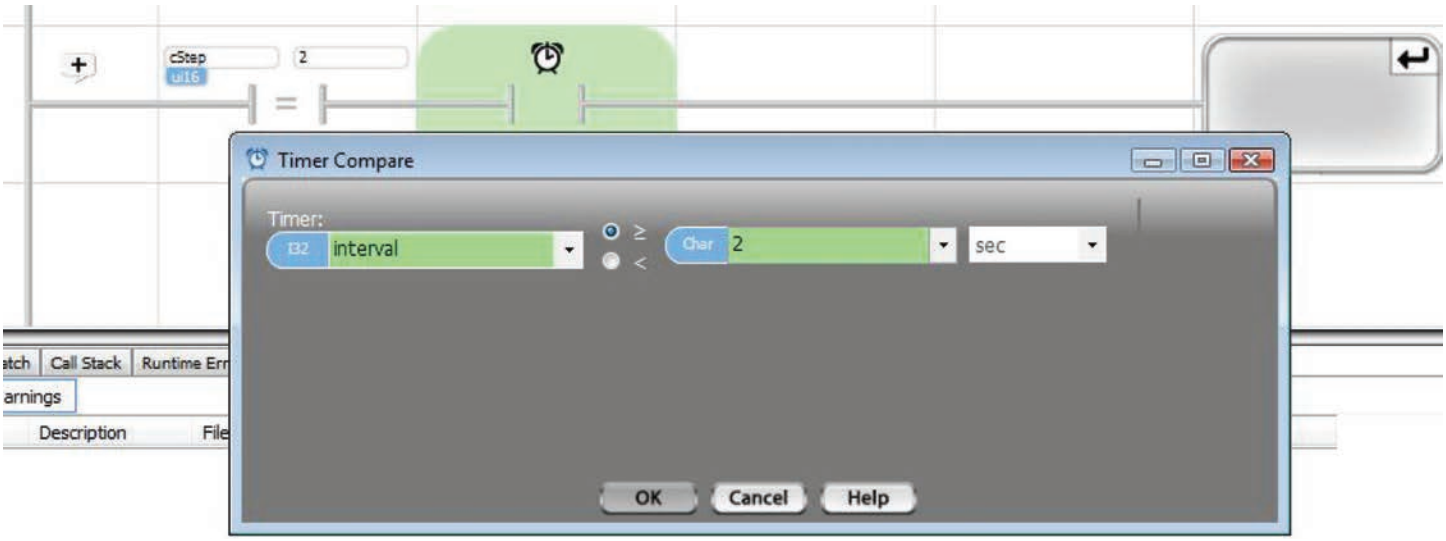


Enter the next two rungs as shown below. The first contact on each rung will check whether the current step (cStep) is step 2. If it is and the "run" bit is active (the run bit is passed from the main program), the Timer will be Started or Continued. If "run" is inactive, the Timer will be Paused.

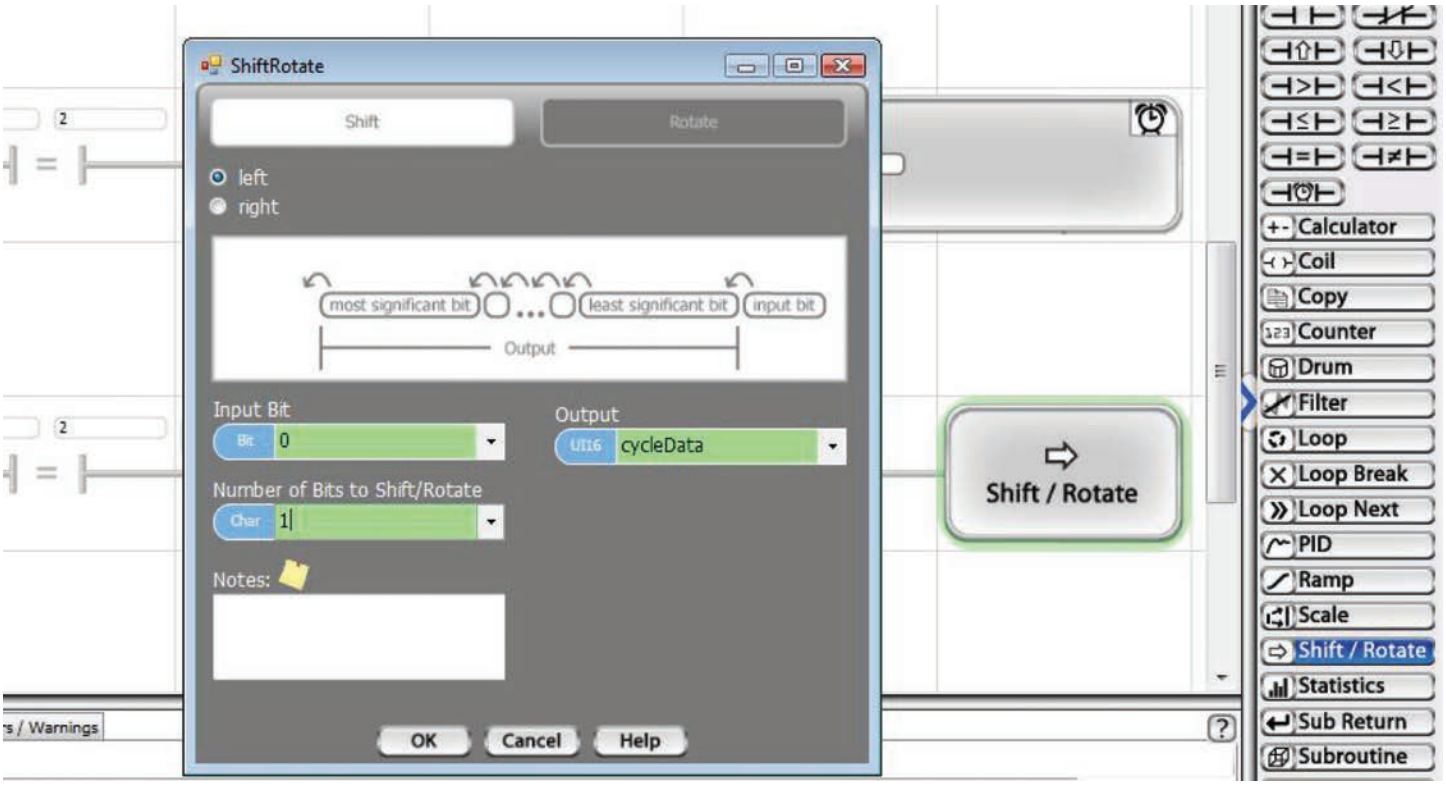


The next rung will check the time. If two seconds have passed since the last shift, it will shift the output one position. Begin entering the line as shown below. Start with the check for cStep 2. Next, select the Timer Compare icon, shown on the right and place it in the rung. When you place the Timer Compare contact, a dialog box will pop up. Select "interval" from the drop down as the timer to use. Select greater than or equal as the comparison type. Enter 2 as the value to compare and "sec" as the time units, as shown.

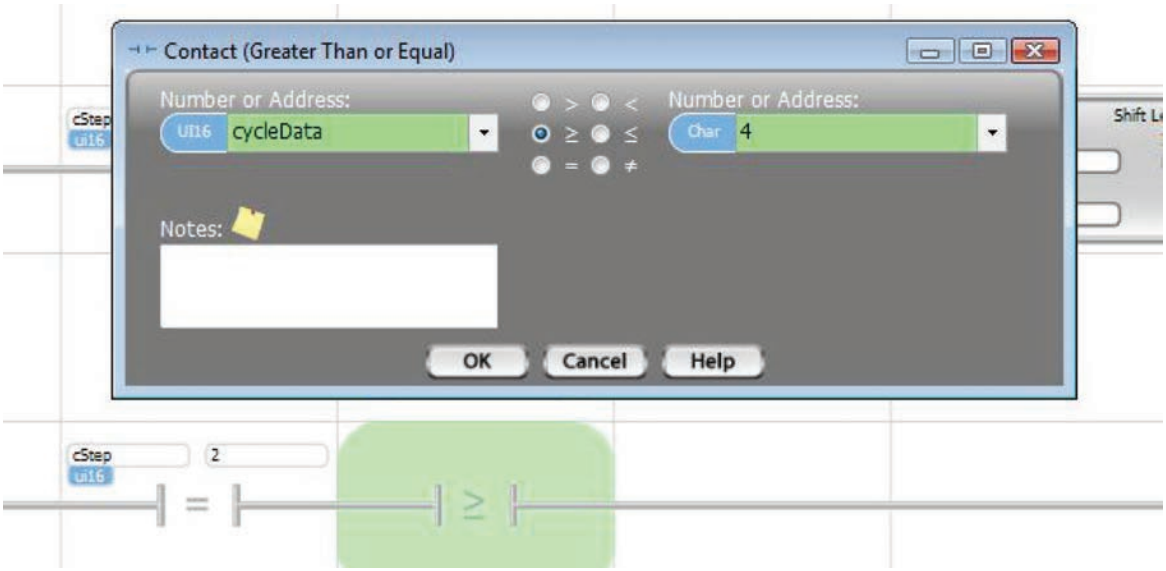




Now, for the rung action, select a "Shift/Rotate" block and place it at the end of the rung. In the dialog box, select Shift and left. Enter 0 as the input bit, select cycleData for output and enter a 1 as the number of bits to shift, as shown. This will create a program block that will shift the 1 in cycleData one position left, each time it is executed.



We only want to shift through 3 bit positions, then start over. To do this, we'll check whether the cycleData output has shifted into the last bit position. On the next rung (we're up to subroutine rung 6), begin with a check for cStep 2. Next, select a Greater Than or Equal To contact and place it. In the dialog box, select "cycleData" for the first item of the comparison. The greater than or equal selection should already be selected. If it is not, click on it to select. Enter the value '4' as the second comparison item.



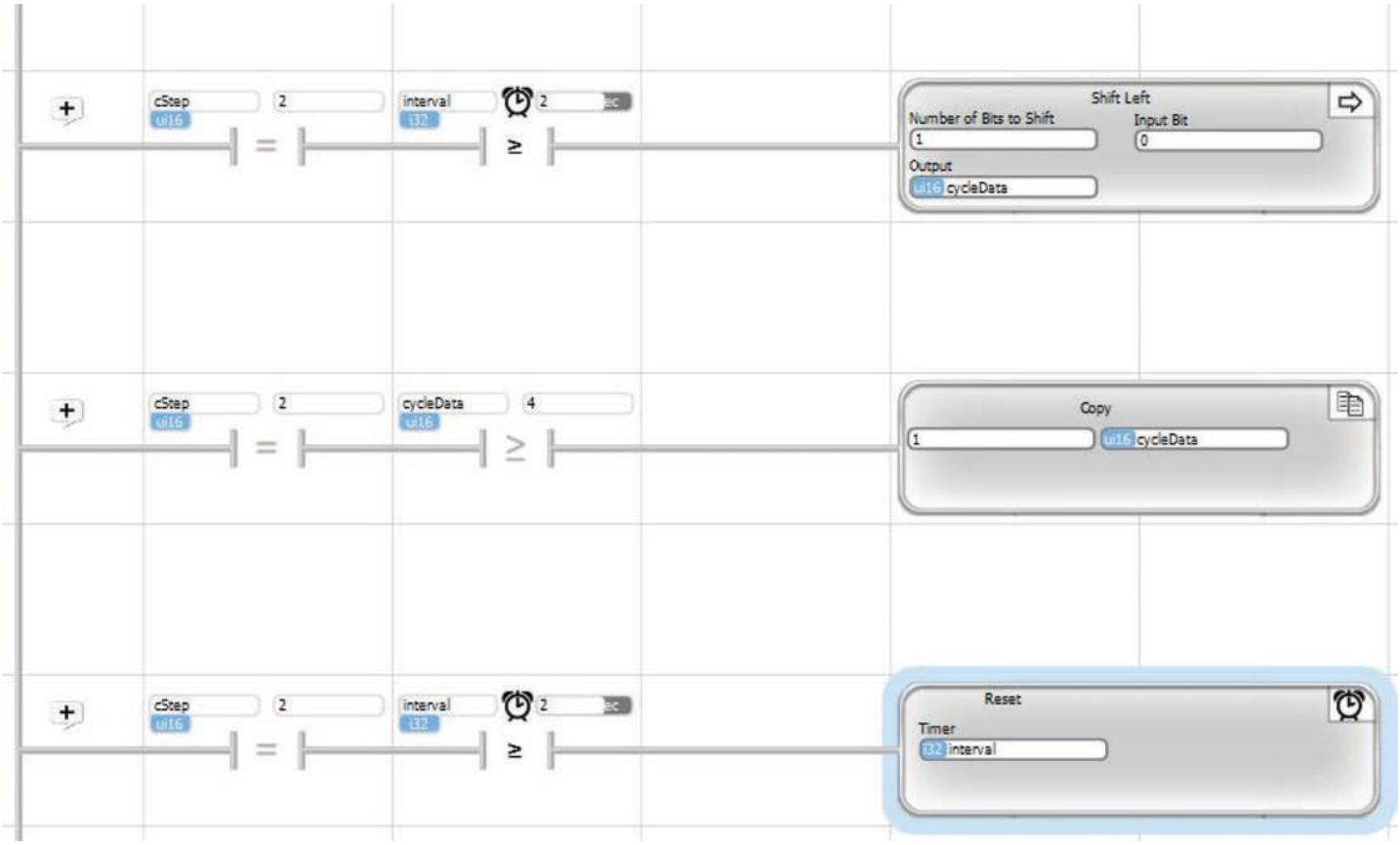
We are actually entering an error here. It will give us a chance to demonstrate some debug features.

Click 'OK'.

Finish the rung by placing a Copy block. Using the Copy block dialog box, create a copy of the value 1 to cycleData.

The last logical operation we need to do is restart the timer. On the next rung, start by performing the check for step 2, followed by a Timer Compare to check that the time is Greater Than or Equal to 2 seconds. Finish the rung by placing a Timer reset to clear the time to 0.

Rungs 5 through 7 should now appear as shown below.

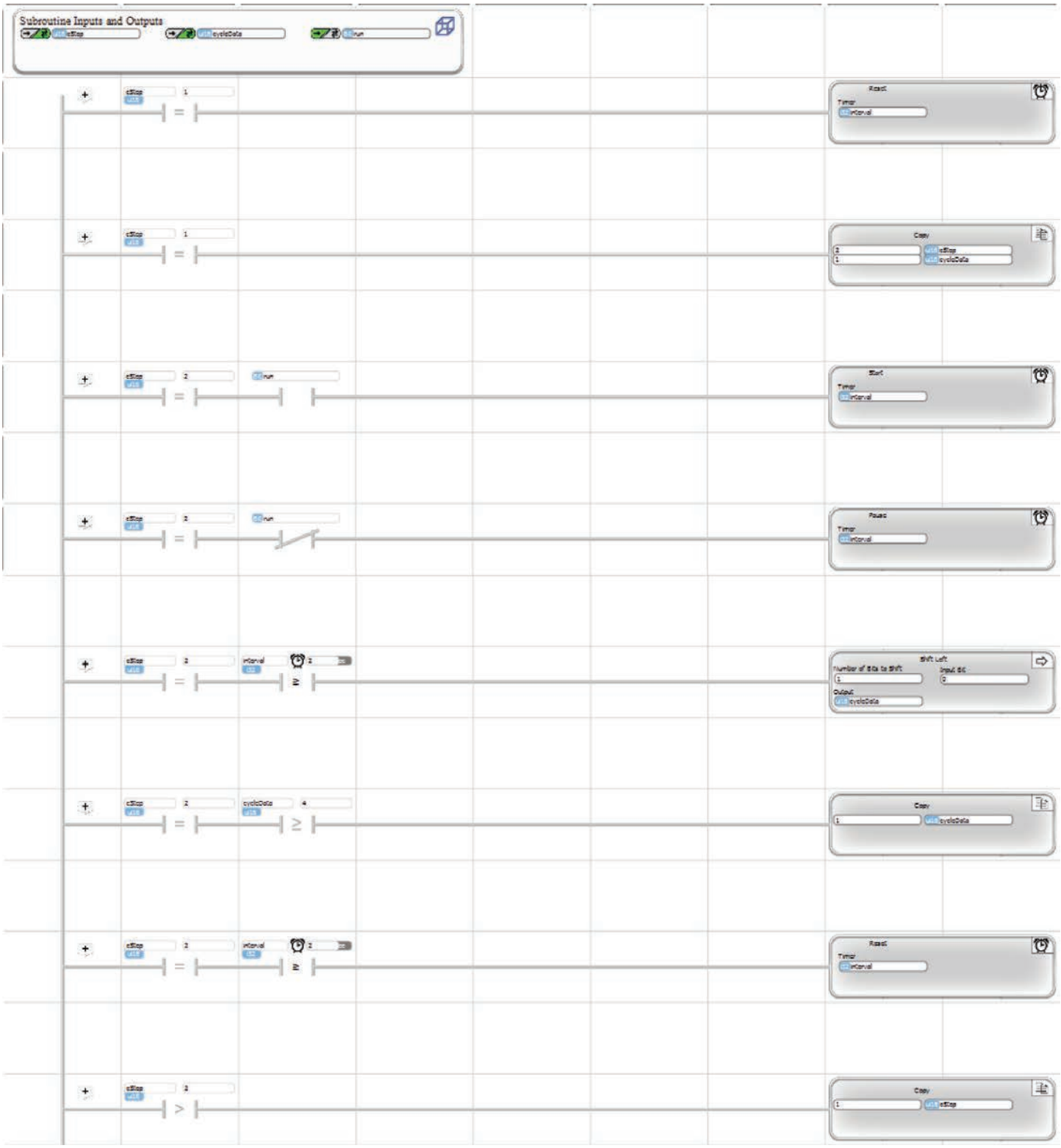


The last rung we'll add to the Cycle3 subroutine is something we like to add to any state machine routine (yes, you just programmed a simple state machine). The rung shown below checks for a cStep value greater than 2. If it should happen, the Copy block sets the step back to 1. Now, there is no way in the logic that we've programmed for cStep to ever be anything but 1 or 2. If it should happen, there is a logic problem, probably somewhere else in the program. Its just a good idea to protect against even those events that can't happen - especially in the program development phase. During program development, its a good

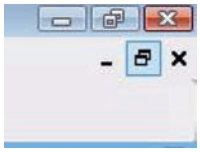
idea to put in these kinds of checks, then monitor if they ever happen. If it does happen, you can start debugging your program logic to determine why. It won't happen on such a simple program as this, and the object oriented program architecture vBuilder protect against one area of code clobbering another area. Particularly in development stage, its a good idea to put in checks anyway.

The full Cycle3 subroutine should appear as shown below. With this many rungs of ladder, the clarity isn't there to be able to read all of the detail, but you should be able to see the overall logic design.

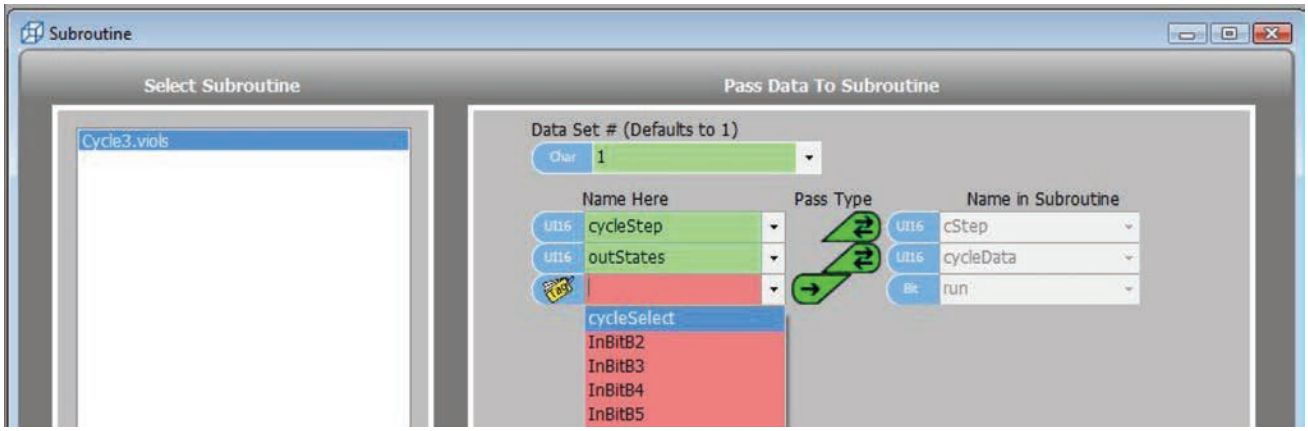
Cycle3 is a simple state machine (see the chapter on state machines for a more thorough discussion). In the first step, the timer and output data are initialized. In the second step, if the run switch is on, the data is shifted around a loop every 2 seconds. If the run is not on, it maintains the outputs in the condition when the run was switched off. You should never get to any other step, but if you do (it would have to be some error somewhere else), this subroutine will restart step 1.



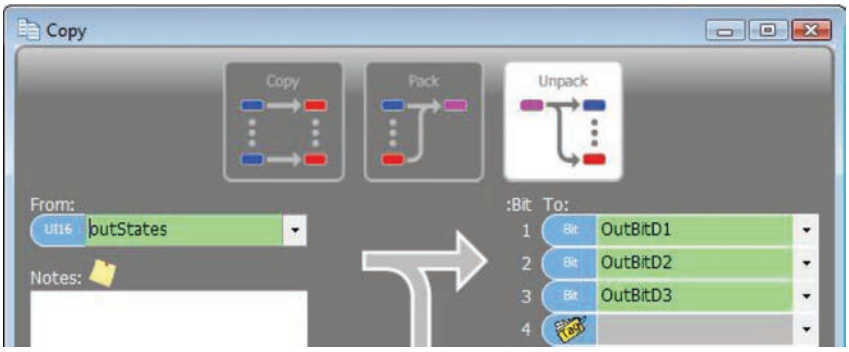
Click the icon in the upper right corner that shows layered windows. When you do, you will get smaller windows with both your main program and your subroutine.



Select the main program. Select and place a Subroutine block (near the bottom of the Toolbox) as the action for rung 2. Notice that when you select Cycle3 (the only subroutine available to select, since its the only one we have defined), the three passed parameters will automatically come up in the list on the right. You need to select the main program tagnamed parameters that you want to pass in. Select cycleStep, outStates and cycleSelect as shown, then click OK.



On the next rung, place a Copy Unpack block (select and place a Copy block, then select the Unpack option, as shown). Select outStates to Unpack From, and OutBitD1 through 3 to Unpack the Bits To. Click OK.

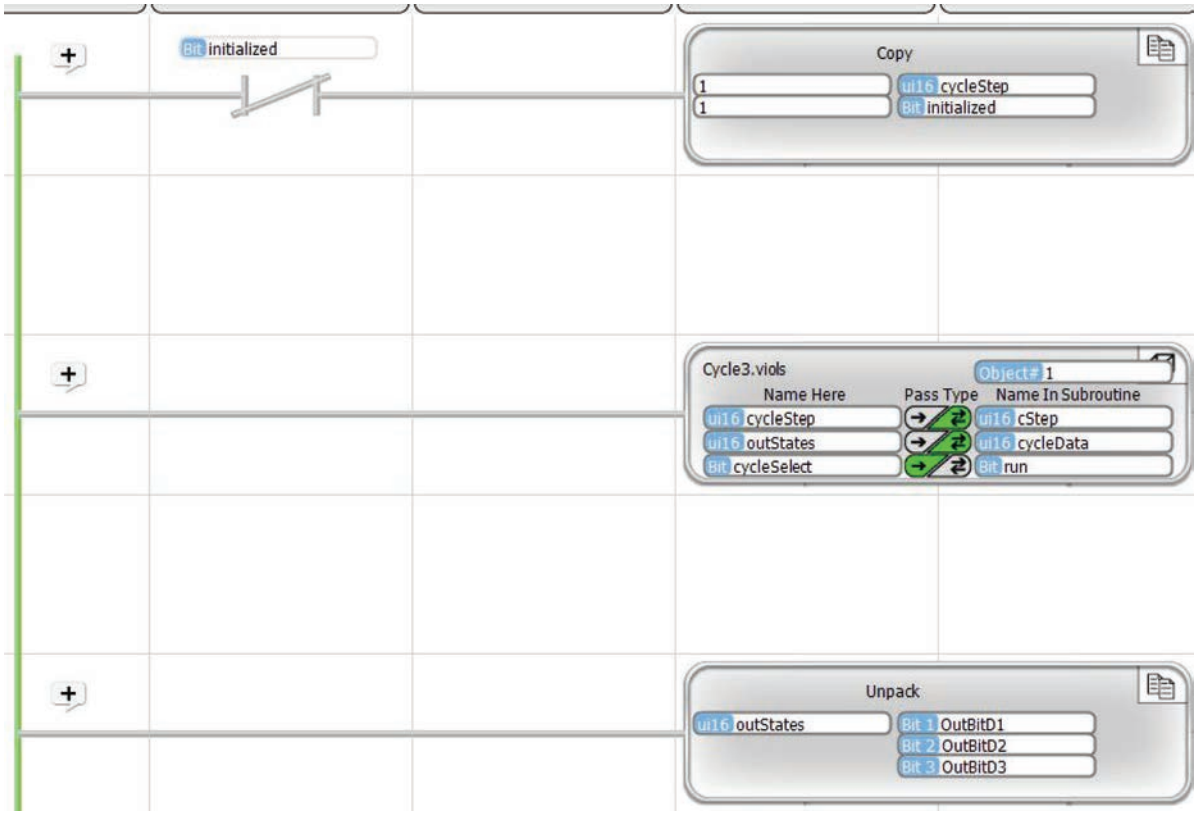


The main program should now look like the illustration on the right. The first rung of the program initializes cycleStep to 1, then effectively disables itself from running again by setting initialized to 1.

The second rung calls Cycle3 to perform the output cycling logic.

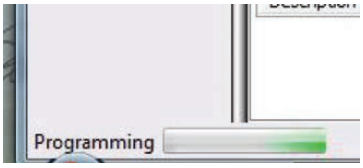
The third rung outputs the bit pattern to three digital outputs.

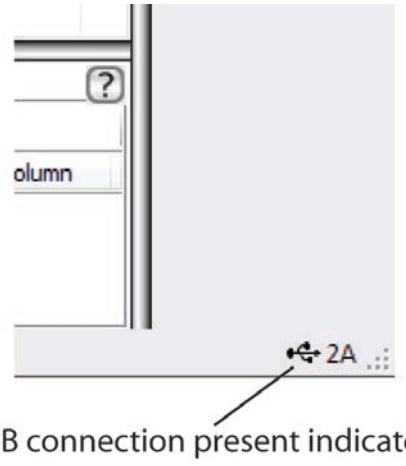
We've got a complete program. Let's try it!



Tutorial Ladder Logic Download, Debug and Run

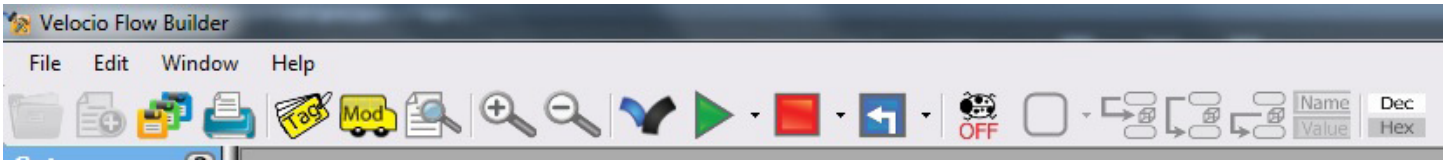
Now, we're ready for some fun. If you have a Branch or Ace PLC (a Velocio Simulator is ideal), power it up and connect the USB cable. You should see a USB connection present indicator in the lower right hand corner of vBuilder. If you have that USB connection, click the Program icon at the top of vBuilder. As you click the program button, watch the lower left corner status indicator. It will switch to "Programming" and show activity. This is a short program, so this will happen very quickly, then change to "Stopped" status.





Once you've downloaded, you're ready to run and/or debug. If you've entered the program exactly as directed, it should run - with one little bug.

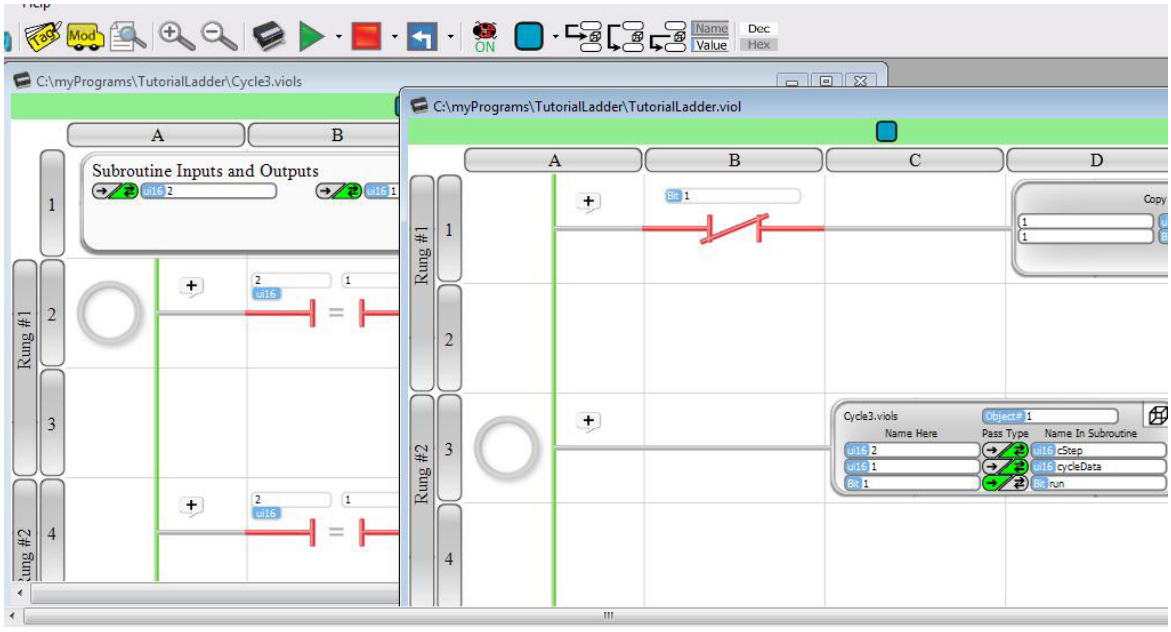
Take a look at the top tool bar. Next to the Program icon, you will see the Run then the Stop button. Next is the Reset, which will cause the program to start over from the beginning. The next set of button icons are debug functions, which we will use as we debug this program in the following pages.



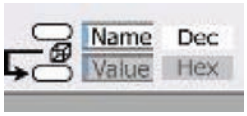
Try running the program. Click on the Run icon. The status indication in the lower left corner should change to "Running". If you have correctly entered the program as described and switch the input that we chose for "cycleSelect" (input B1) on, you should see output LEDs cycle every two seconds. We intentionally put a bug in the program though. You should see two outputs cycling, instead of the three that we intended. That gives us a chance to debug the program.

Select Stop. The status indication in the lower left corner will say "Stopped". Select the icon that looks like a ladybug. It is the debug mode selection. You should see it change from indicating that it is OFF to indicating that debug is ON. You will also see that your main and subroutine windows will have a red bar across the top. This red bar is an indication that the routine is not currently running.

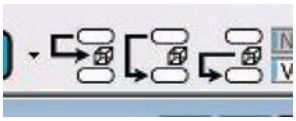
Select Run. The colored status bar on each program window will change to green, indicating that the program is running. You will also see some contacts turn red and others green. This is a high level indication of the predominent execution of each decision. To see the details, we have to look closer, but just this quick view tells us that we are executing the subroutine's step 2. Toggle the cycleSelect input and you will see the 'run" decision in step 2 change between red & green. The PLCs are only polled by vBuilder around twice per second, so there will be a slight time lag on the screen.



While the program is running in Debug mode, notice the icons on the top tool bar labeled Name/Value and Dec/Hex. Look at your program & notice that your program blocks have tagnames in them. Click the Name/Value icon, so that the Value portion is highlighted and notice that your blocks now have numeric values. These values are the current values of the tag named variables. Toggle back & forth. Try the Dec/Hex icon, when selected for value. The values change between decimal and hexadecimal format. Since we are dealing with such small numbers, there is really no difference. When you have a program with bigger numbers, it will be more distinct.

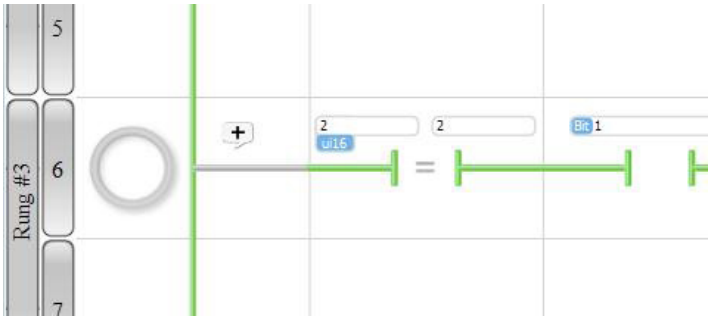


Notice the icons on the top tool bar that look like this -



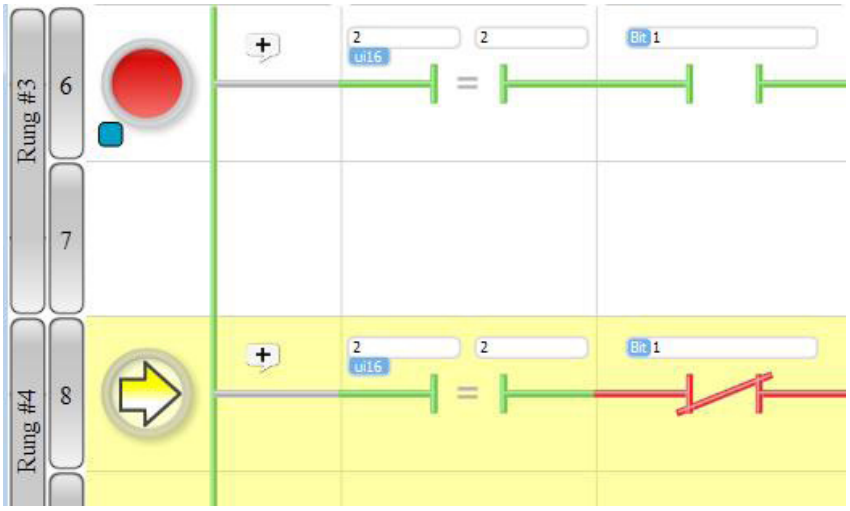
These are single step functions. If you hover your cursor over them, you will see that the first one is "Step In", the second "Step Over" and the third "Step Out". The first one can be used to step through the execution of your program, one rung at a time. The second one operates the same, except when the block is a Subroutine call. Instead of stepping into the subroutine, it executes the entire subroutine. The third one will step through execution, but if you are in a subroutine, it will execute all blocks until you return to the calling program, in other words, "Step Out" of the subroutine. Try these functions.

Click the "Run" green arrow again, while you still have Debug On. The program should be running full speed. Move your cursor around. Notice that whichever rung your cursor is over will have a hollow round indicator, like that shown on the right, just to the left of the rung. This is an indication that you can place a breakpoint here.



Click on a rung. A red filled circle and blue rectangle will appear - indicating the there is a breakpoint on this line. Since the program was running when you placed the breakpoint, it will continue to run until it 'hits' the breakpoint. The line will turn yellow and an arrow will appear in the circle. This indicates that the program is stopped at this line. The program hit the breakpoint and stopped.

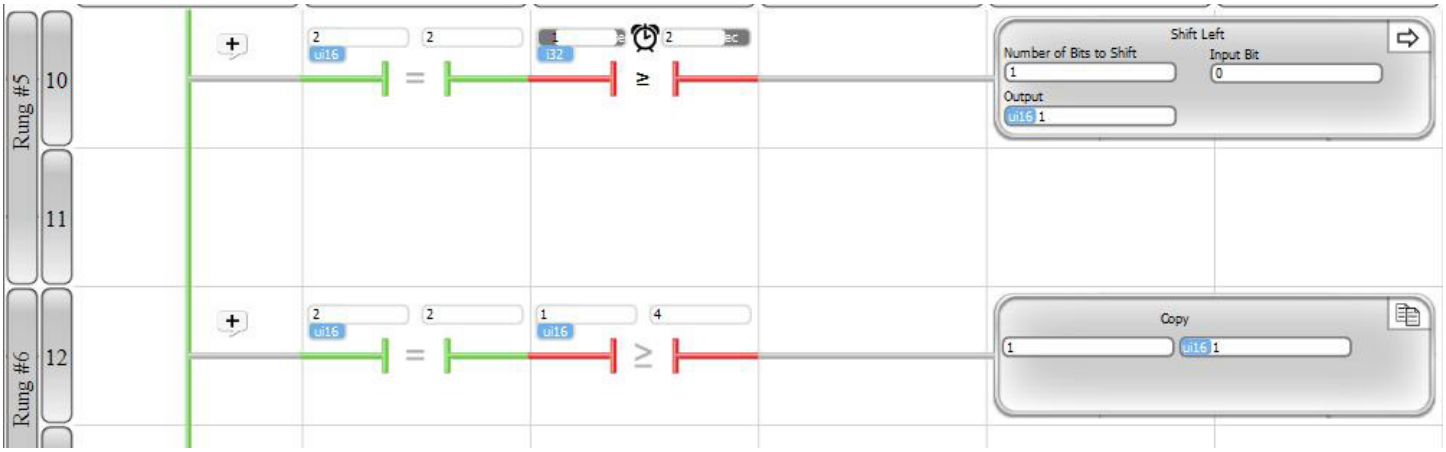
If you click single step, you see that the yellow highlight and the arrow move to the next rung, indicating that the program execution has moved to the next rung. As you single step again and again, the arrow and the yellow highlight will continue to move to each successive rung. The highlighted rung is the one that will execute during the single step.



We need to figure out why the program is only cycling between the first and second outputs. When we did this for the flow chart version of this example, we were able to put a break point on the Shift block, wait for the program to stop at the break-point when it shifted, check the output shift, run to breakpoint again until we get the shift that didn't appear to work, then step through to see what was happening.

With Ladder Logic, we can break on rungs, not individual program blocks. For a program where the error that we are trying to track is something that happens very intermittently, its a little more difficult to catch in ladder than with flow charts. In this case, the shift operation happens only every two seconds. The PLC will execute the program hundreds, possibly over a thousand passes through the program between each shift. Since we can't put a break on the exact program pass where the shift occurs, like we can in vBuilder Flow Chart programming, its nearly impossible to catch the exact program pass that we are interested in. This problem does not exist with logic that is not so intermittent.

Even with the inability to break on the exact Shift operation that we are interested in, the Debug features of vBuilder provide enough information to track down and solve this problem. Just put the program in Run, in the Debug mode and watch it. Select "Value" for the Name/Value selection. As you watch the 5th and 6th rung of the Cycle3 subroutine, you will notice that

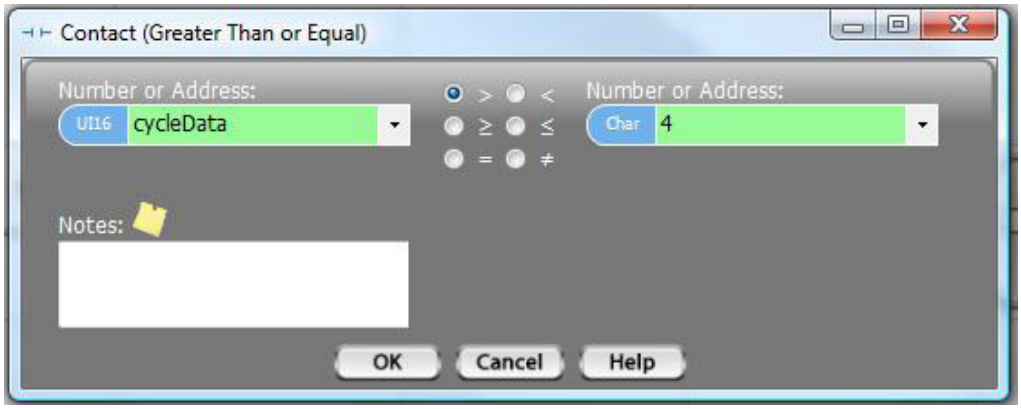


the Output of the Shift Left, which is cycleData, changes between 1 and 2, but never appears as 4. Our intent is for it to cycle between 1, 2 and 4.

Look at the next line and we see the problem. As soon as we shift cycleData to 4, the next line checks to see if cycleData is Greater Than or Equal to 4. If it is 4, the contact is closed and the program will copy the value 1 to cycleData. The subroutine will never return to the main program with cycleData being a 4. That's exaclty what we see in the operation.

The error that must be corrected is the check for Greater Than or Equal to 4. We could do that either by changing to check for Greater Than or Equal to 8, or change the check to just Greater Than 4. Let's do the second choice.

Stop the program. Turn off Debug mode. Double click on the Greater Than or Equal comparison contact in Cycle3, rung 6. Change the comparison to just Greater Than, as shown. Then click "OK".



Download the revised program. Run it. It should now cycle through 3 outputs. You can go to Debug mode and watch the difference in the execution.

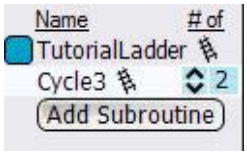
Adding a Second Object

Your tutorial program works. As with the Flow Chart tutorial, we can't resist demonstrating one of the advanced features of vBuilder. Subroutines, in vBuilder are actually Objects. An Object is a self contained program, coupled with its own captive data. In a vBuilder program, multiple Objects of the same type can be created and used. Objects can also be copied to other programs, giving you re-usable program components that you have already debugged and proven to work. We're going to implement a second Object as a demonstration.

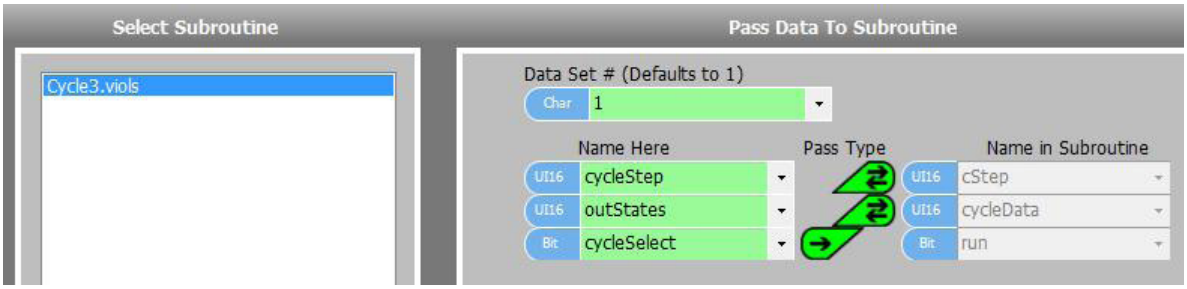
Select your main program, select the Tag icon, and add the following tagname variables :

- Select Input bits and rename InBitB2 'cycle2Select
- Select ui16 and add 'cycle2Step' and 'outStates2"

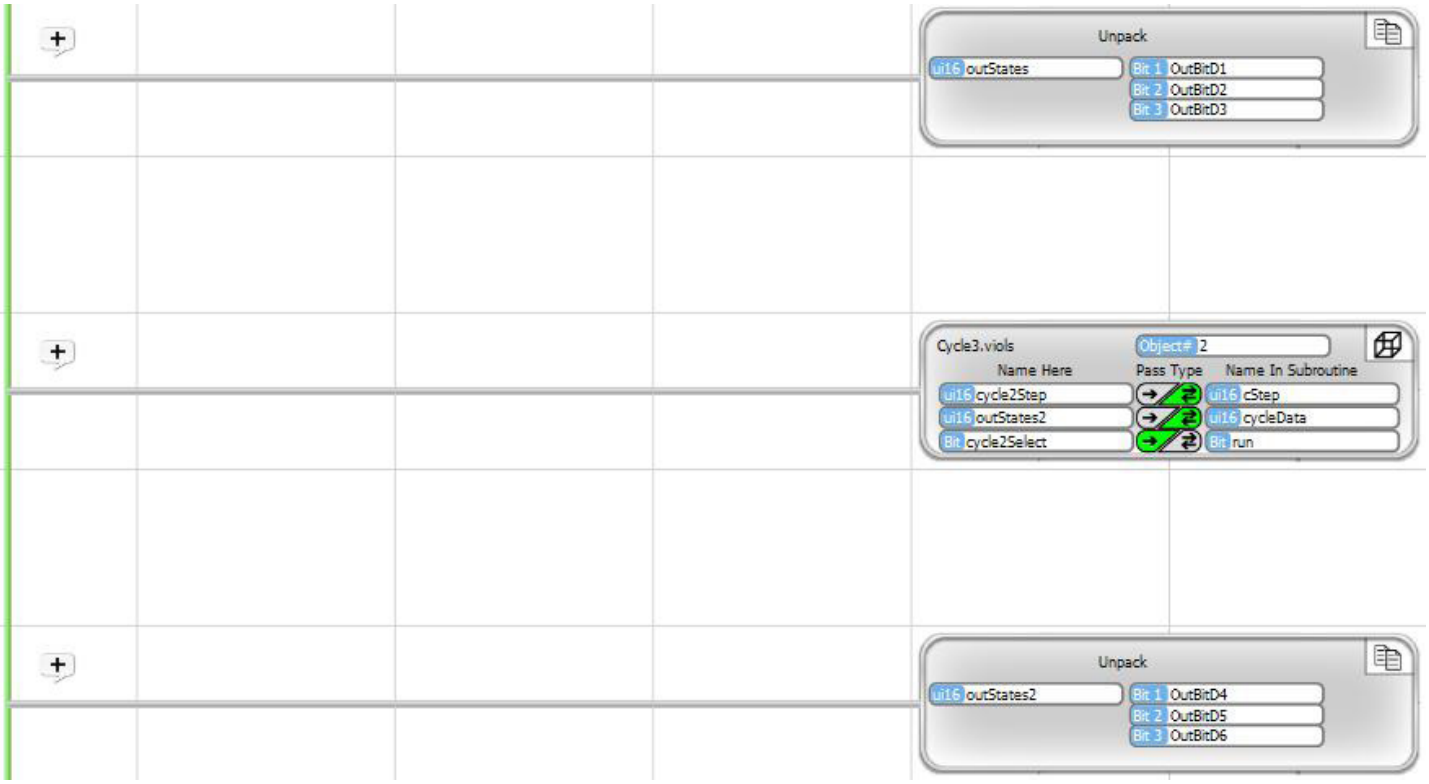
Take a look at the list of Project Files along the left hand side of vBuilder. Next to Cycle3 is an up and down dial and the number 1. Select up to get the number to change to 2. You've just created a second Cycle3 object. Now lets see what we can do with it.



First, double click on the Cycle3 Subroutine call in your main program to open up the dialog box. Notice that there is an entry for Data Set #. We didn't need to do anything with it, since there was only one object. If you didn't set the Data Set # to 1, previously, put a 1 in the box now, as shown. You just defined this call to Cycle3 as using object 1.



Now add a second Subroutine call to Cycle3 and a second Copy Unpack as shown. Make sure you select Data Set 2. Pass cycle2Step, outStates2 and cycle2Select in. For the Copy Unpack, unpack from outStates2 to OutBitD4, OutDitD5, and OutBitD6. Your program should look like the illustration on the right. Be sure to add the initialization of cycle2Step to 1 in the first rung copy statement.



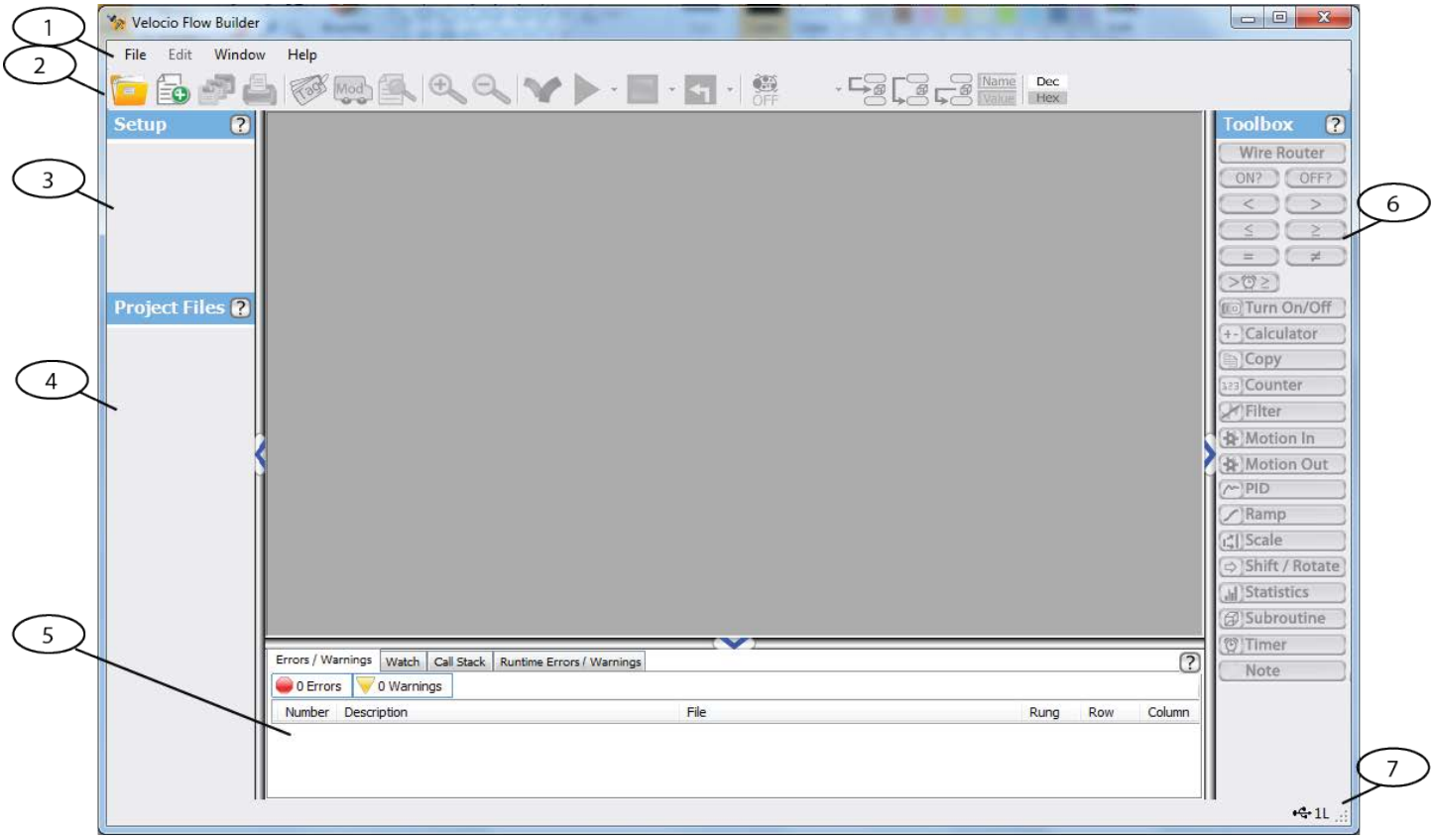
Program it and run it. You will see that when you flip the B1 switch (which you gave the tagname 'cycleSelect') on, the first three outputs will cycle on a 2 second step rate. When you switch the B2 switch (which you gave the tagname 'cycle2Select') on,

outputs 4, 5, and 6 cycle. Try turning the inputs on and off. Notice that the step time for the 2 groups is not synchronized. That is because we created our timer in the Cycle3 object. The two objects each have their own independent timer. If we had wanted to synchronize the step times, there are simple ways we could have done so. The purpose was to demonstrate that we can create two totally distinct entities, or Objects, that operate independently, but use the same logic.

There are other powerful aspects of vBuilder that you have just unleashed - but we'll discuss those in later detail pages.

3. Navigating Velocio Builder

To begin using Velocio Builder, double click on the desktop icon, or select Velocio Builder from your program selections. When you first open Velocio Builder, a screen similar to the one shown below, will appear.

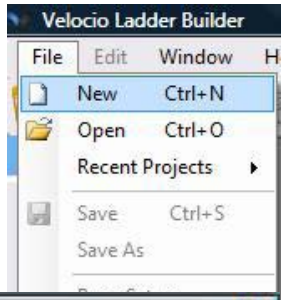


This is your starting point. The large area in the center is where you enter, edit and debug your application program. Everything else, surrounding that area, contains the tools that allow you to do so. The general screen areas are labeled with numbered bubbles, which are explained below.

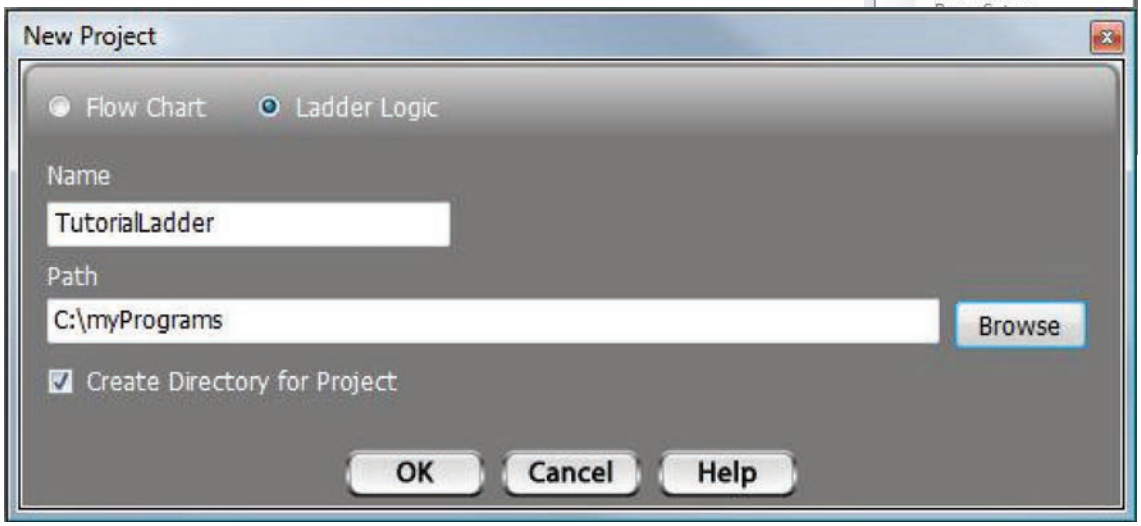
1) General menu bar (similar to most Windows applications).
2) Quick access tools which are available when not grayed out.
3) The program hardware set up
4) List of programs and subroutines associated with the application program.
5) Program information window.
6) Program Toolbox tools, which may be either ladder or flow chart components, depending on your selection
7) Status indication, including connection status to a PLC and cursor location within a ladder or flow chart.

Entering a New Program

To enter a brand new program, go to the File menu, in the upper left hand corner, and select "New" from the drop down list, as shown on the right.

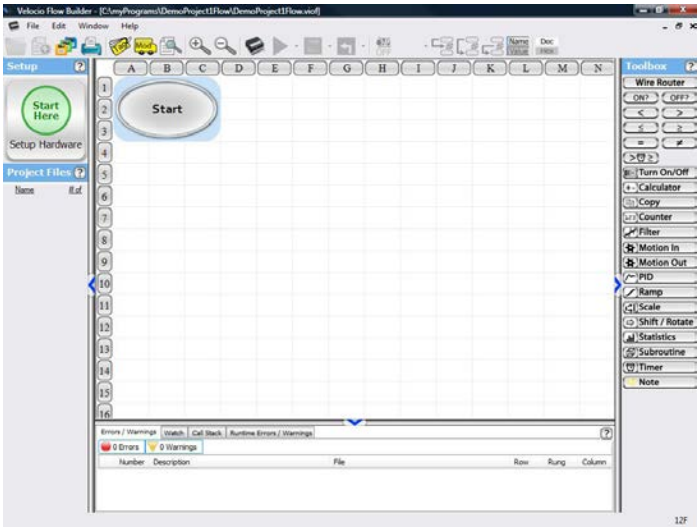


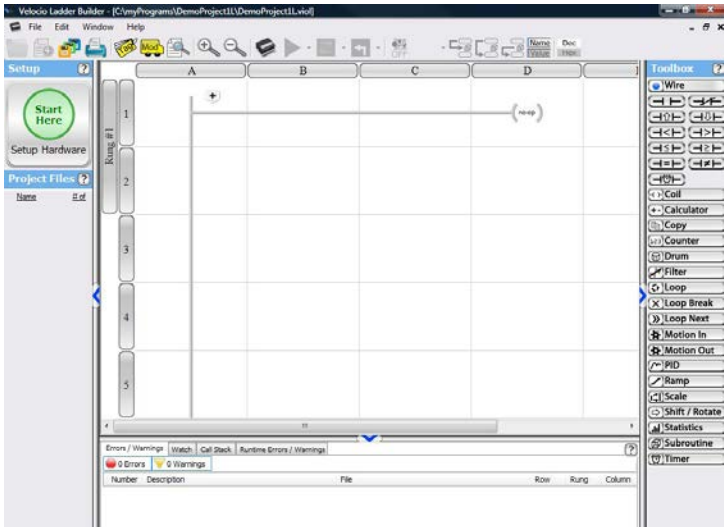
When you select New, a dialog box, like that shown below, will pop up. With this dialog box, you can create a name for your project, tell vBuilder where to put it on your computer and define whether you want the main program to be a Flow Chart or Ladder Logic program.



- Select Flow Chart or Ladder Logic by using the selection buttons at the top of the dialog box.
- Type in the name of your project in the text box below "Name"
- Under "Path" select the directory where you want to store your program. Its a good idea to create a directory to store all of your programs and select that directory. Once you create the directory, you can browse to it and select it using the Browse button.
- It is highly recommended that you keep the "Create Directory for Project" checked. If it is checked, it will create a subdirectory, under the directory the you defined for Path. That subdirectory will have the same name as your project name and will contain all project files.

Depending on whether you selected Ladder Logic or Flow Chart, your next screen will look like one of the two below. The large area in the middle is the area where you build your program. With a Flow Chart selected, it will open with a Start block and the Flow Chart tools in the Toolbox on the right. If you select Ladder Logic, the screen will contain an empty Ladder chart with one rung that is simply a NO-OP, and Ladder Logic tools in the Toolbox.





The next thing you need to do is configure your hardware.

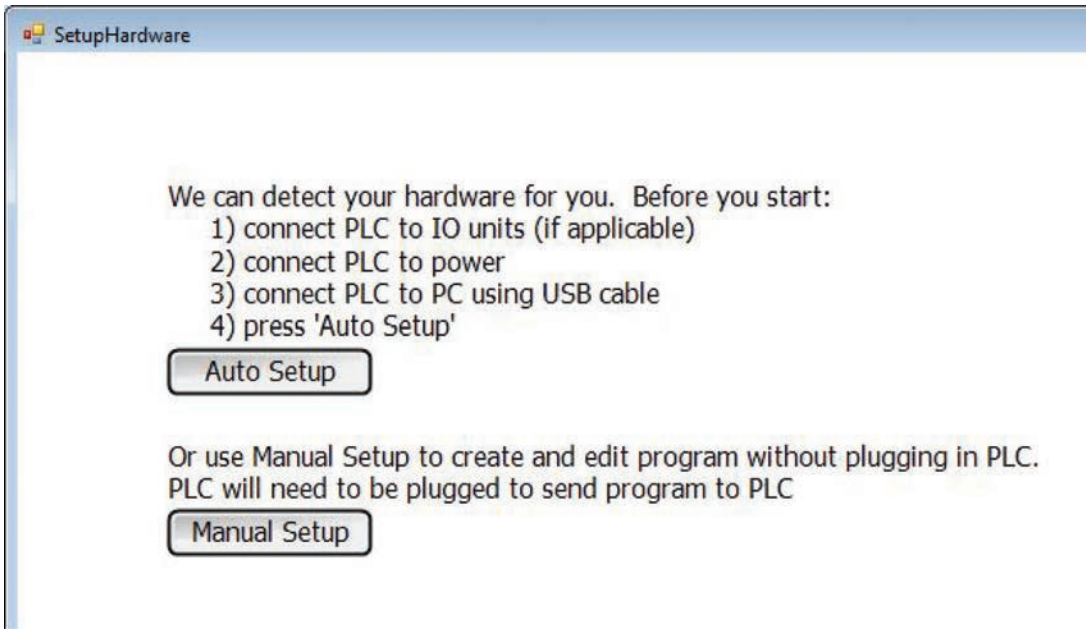
Setting Up the Project Hardware

Before you can do anything else, you must Set Up the project hardware. During set up, you define your main PLC and all sub-ordinate PLC units that are attached. For each of these units, there is selection of the IO that is present. For any PLC units, you can define certain input pins for high speed counter inputs and certain output pins for stepper motion control outputs. Other options may come up in the process, based on the modules selected.

The easiest way to Set Up the Project hardware is to have it all connected, powered up and connected to your computer with a USB cable. If you do that, you can auto configure. You will still have to configure the high speed counters and stepper motion controls and a few other details, but the majority of the set up will happen automatically.

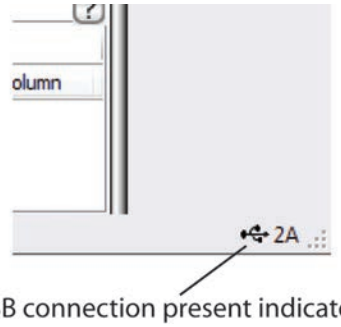
The other option is to go through the process of manually selecting each device and option. Manual configuration is pretty easy and quick, as well.

To start the process, click on the big green "Start Here" button in the upper left, under Setup. You will get a screen that looks like this.



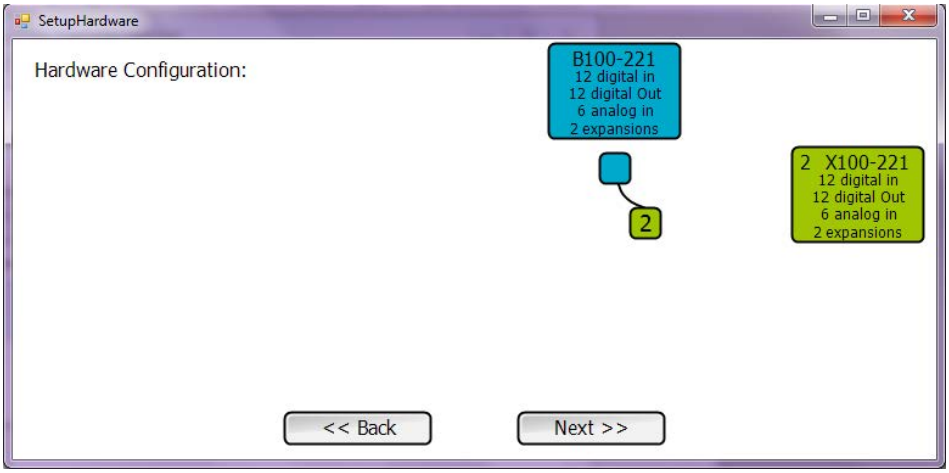
Auto Setup

For Auto Setup, follow the instructions given in the setup screens. During auto setup, vBuilder will querie the PLC to see what is attached. In order to auto configure properly, all of the PLC modules must be connected, powered up and the USB connected to your PC with vBuilder. Make sure the USB connection present indicator is present in the lower left hand corner. Then select Auto Setup.



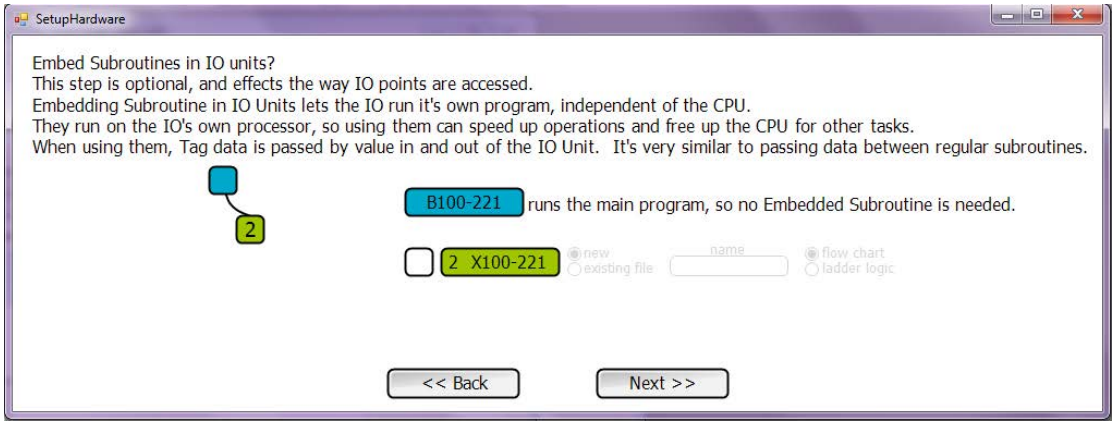
vBuilder will read the attached configuration, then display what it finds. In the example, below, we connected to a Branch unit that had 12 digital inputs, 12 digital outputs and 6 analog inputs, . The Branch has one Branch Expansion unit with 12 DI, 12 DO and 6AI attached to the second vLink expansion port. When we autoconfigured, the screen, shown below, appeared.

Notice that that all of the PLC modules, Ace, Branch or Branch Expansion are shown. For each module, the Input/Output and connection details are shown in a color coded box. This particular configuration shows one Branch unit connected to one Branch Expansion unit through the second vLink expansion port. The 2 in the Branch Expansion block indicates the port 2 connection.



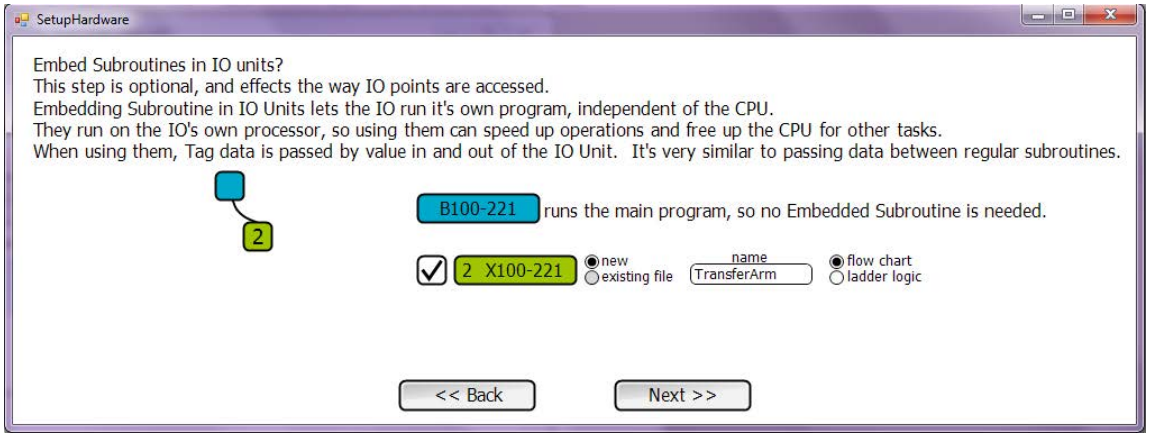
If everything looks OK, press the Next button. If it is not OK, press 'Back', make sure that everything in connected properly and turned on and try again.

If the auto configure has found Branch Expansion units attached, the screen shown on the right will appear. This screen allows you to define whether each Branch Expansion will contain an embedded subroutine, or not. Each unit is listed, as shown. If you want to put an embedded object subroutine in a Branch Expansion module, click the checkbox next the the unit listing.



If you want an expansion module to be expansion IO, just leave the checkbox unchecked.

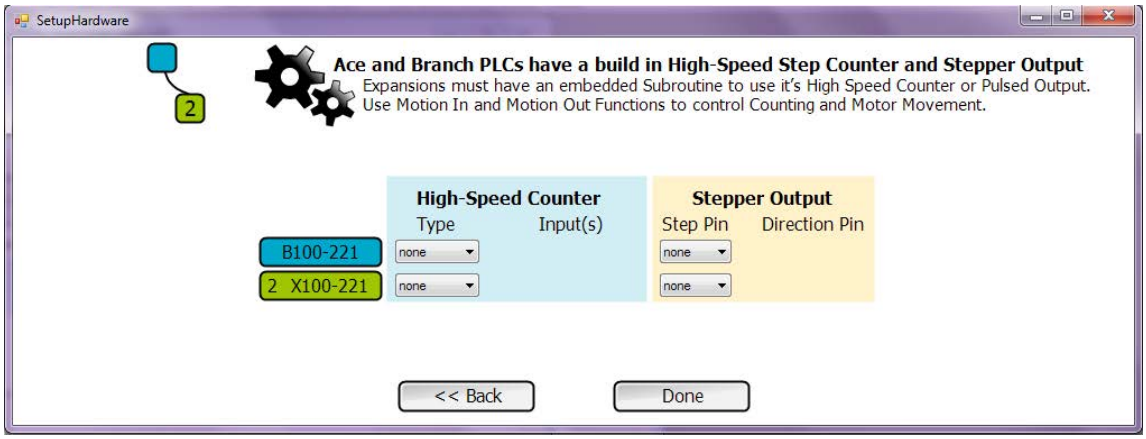
The illustration on the right shows a selection of an embedded object subroutine for the Branch Expansion unit. Once you select the checkbox, you must select either new or existing file, enter a name for the embedded subroutine and select whether the embedded subroutine's main program will be flow chart or ladder logic.



- New or existing file : If you will be creating the program that will reside in the Expansion unit, check new. Checking new means that a new program will be created as the embedded object subroutine. If you already have a subroutine written that you want to place in the Branch Expansion, select existing file. If you want to place the same new program in more than one expansion unit, select new for each Branch Expansion and type in the same name.
- Name : If this is a new embedded object subroutine, type in the name that you want to use for it. If you selected "existing file", use the browser to select the existing object subroutine that you want to place in the expansion unit. [One key feature of embedded object is that they can be reused]
- If this is a new embedded subroutine, select whether flow chart or ladder logic will be used to develop the main program.

Click Next.

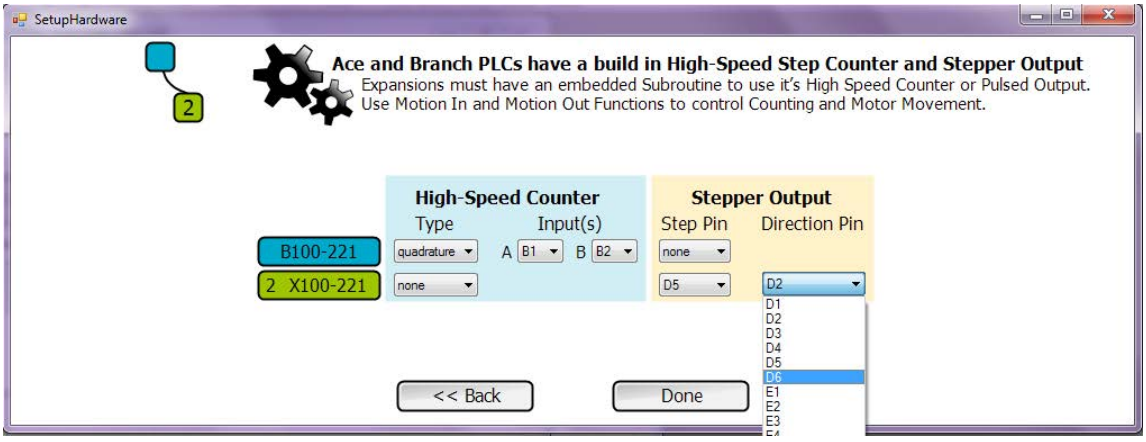
The next screen that will appear, is shown on the right. This screen gives you the option of configuring the main PLC and any Expansion units that have been set up for embedded objects, to have dedicated IO pins for high speed pulse counting and stepper motion control.



The figure on the right shows configuration for a high speed counter in the main PLC and stepper motion in the expansion.

Any IO points that you select for high speed pulse counting or stepper motion are dedicated for those functions and not available for general purpose IO.

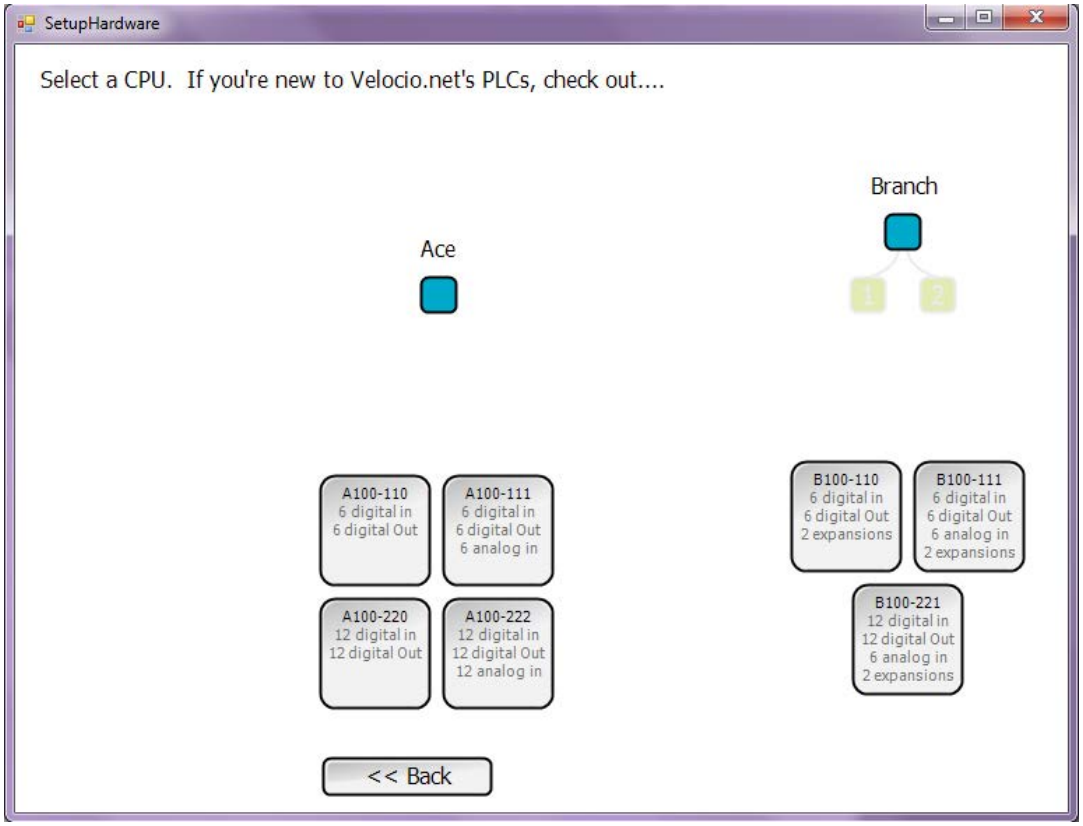
Click Done. You've got your hardware configured.



You can change the configuration later, if your system needs to change. To make a change, start with clicking on the Set Hardware area and go through the process again. To start the configuration again, after you click the Setup Hardware, click 'Back' until you reach the window for the change you want to make. For anything that you don't want to change, just click Next.

Manual Setup

For Manual Setup, click the green "Start Here" button, then "Manual Setup". The window, shown below, will appear. If you are configuring for an Ace project, just choose the IO configuration for the Ace from the options below "Ace", on the left. The selected configuration will turn blue.



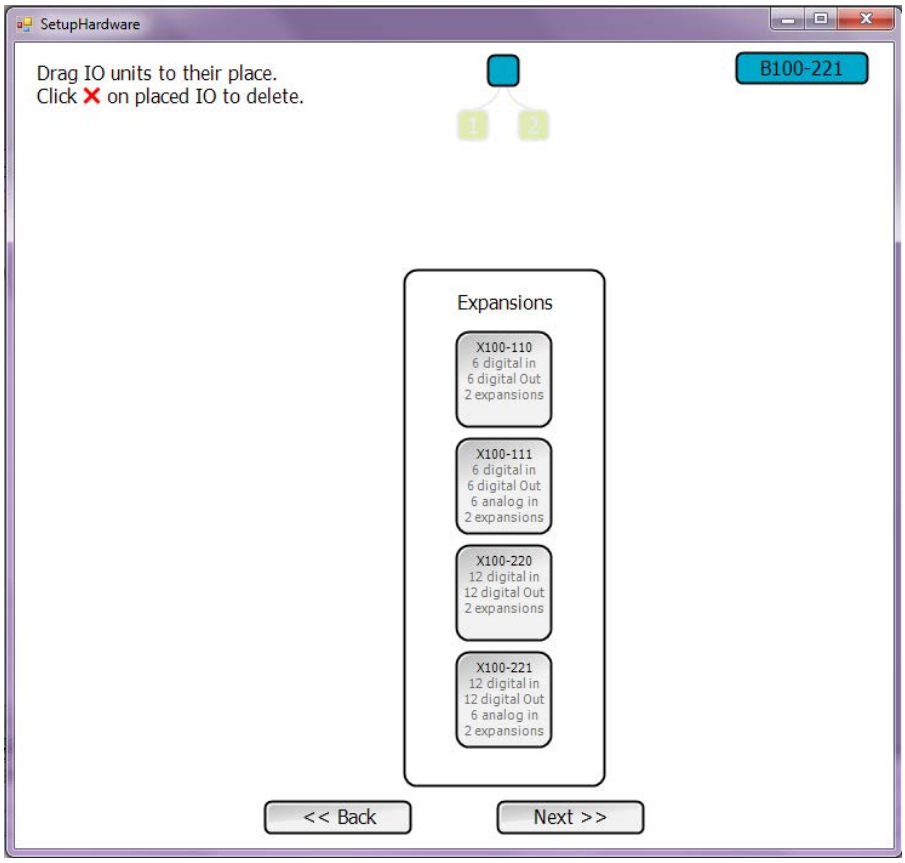
If you are configuring a project with a Branch, select the Branch IO configuration from the blocks on the right. If you select a Branch unit, the next page will allow you to select all of the modules connected to the Branch.

When you select either a Ace or Branch unit with the IO configuration that you want, it will be highlighed in blue. Blue indicates the highest level. The illustration on the right shows a Branch with 12 DI, 12 DO, 6AI and 2 expansion ports selected.

Click Next.

If you have selected a Branch, the next screen instructs you to drag Expansion modules into the positions that you want in order to configure the full system. Each unit has two vLink expansion ports. The configuration screen shows a faded block for port 1 and port 2. If you drag a Expansion unit to one of these faded blocks and drop it there, it will show up in a bold color with the port number that it is attached to in the middle. It will also show the next level of configuration possibilities.

In a Branch system, you can configure three (more) levels deep, for a maximum of 15 modules.

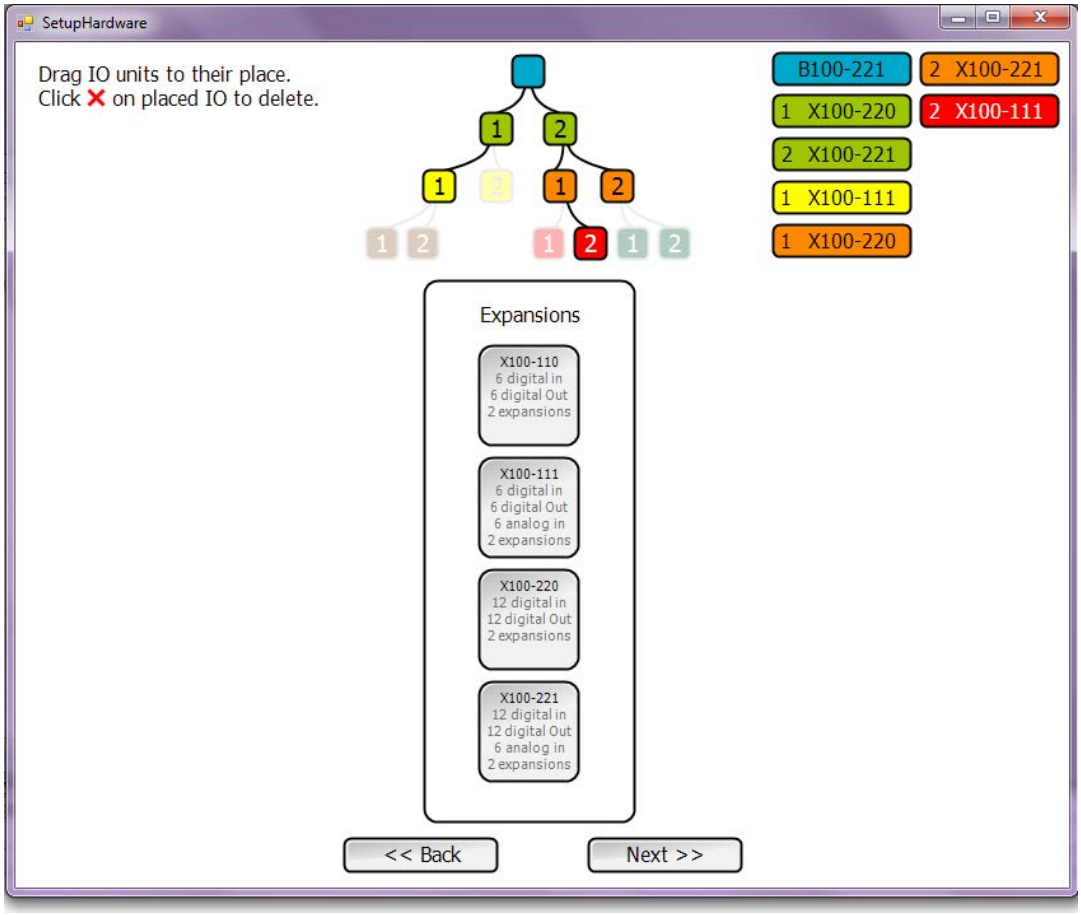


Here is a configuration consisting of 7 modules at four levels. Notice that each module is color coded and listed on the side.

The color codes signify level and port connections. When referencing them during tag configuration, the connection tree identifies the particular point. For example, digital input C3 on the Expansion shown as orange 1, is shown as pin 21C3 in the tag name table. The identifier 21C3 signifies the C3 input on the module connected through Branch Expansion vLink port 2, next level expansion port 1. Analog input A2 on the red module labeled 2, would be identified as pin 212A2.

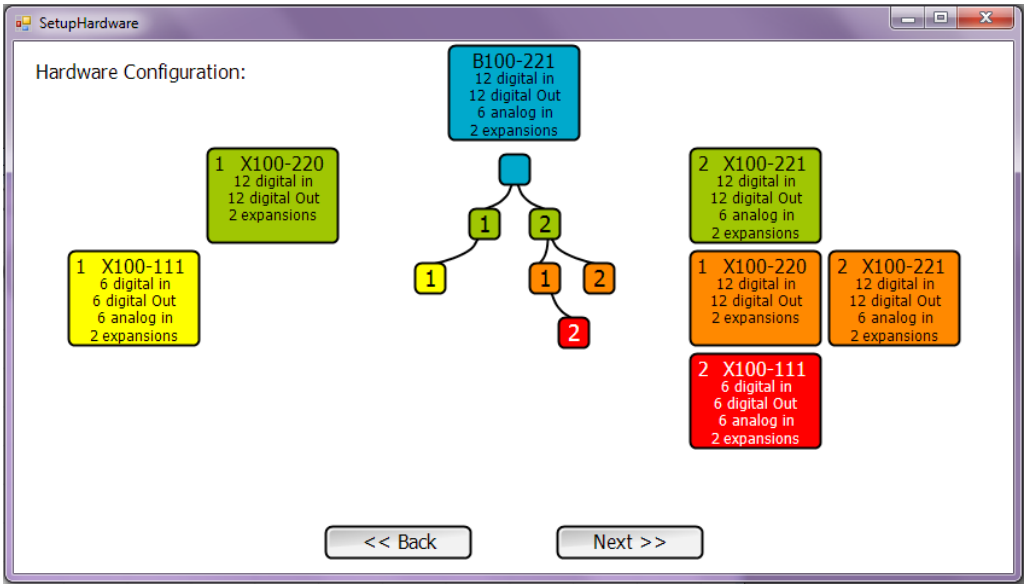
The listing on the right shows all modules connected, along with their IO configurations.

Click Next.



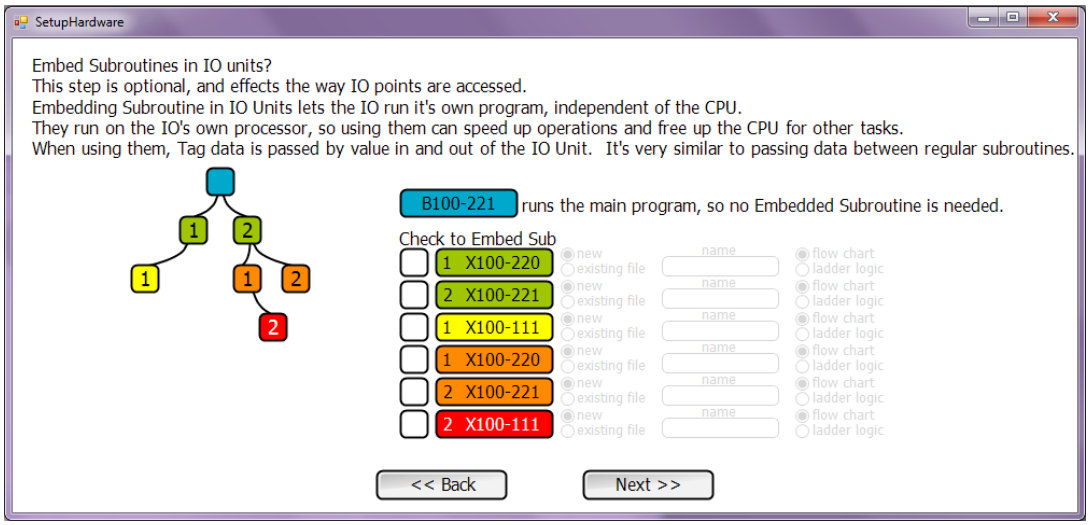
The next screen will display the system configuration, with color coded detail boxes showing the configuration of each individual PLC unit. That is shown on the right.

If everything is OK with the configuration shown, click Next. If not, click Back and make the required changes.



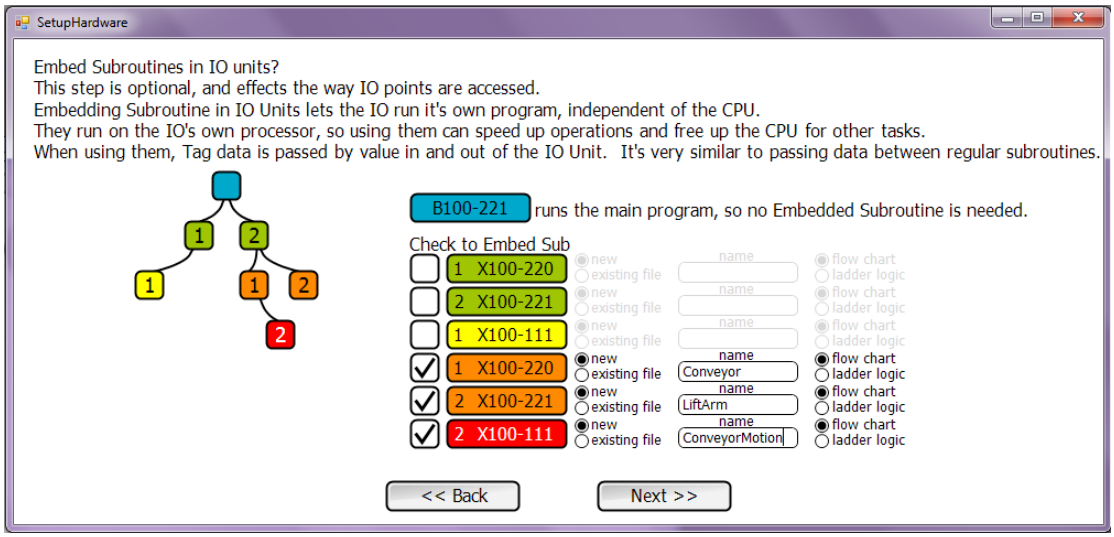
Any Expansion unit can have an embedded object subroutine in it. The next screen allows you to select any or all of the Branch Expansions for embedded objects, by checking the box next to its listing.

If you check for an embedded object, you can select a new or existing file (program). If the object program is one that already exists, you can browse to and select it. If it is one that you will be writing new, select New and type in the name. Also select whether you want to program in Flow Chart or Ladder Logic.



The screen shot on the right shows the system configured for embedded object programs in three of the Expansion units.

A key characteristic of Embedded Subroutines is that they are a Subroutine of the program above them in the tree structure. In the configuration shown, 'Converyor' and 'LiftArm' are Subroutines of the main program. 'ConveyorMotion' is a Subroutine of 'Conveyor'.

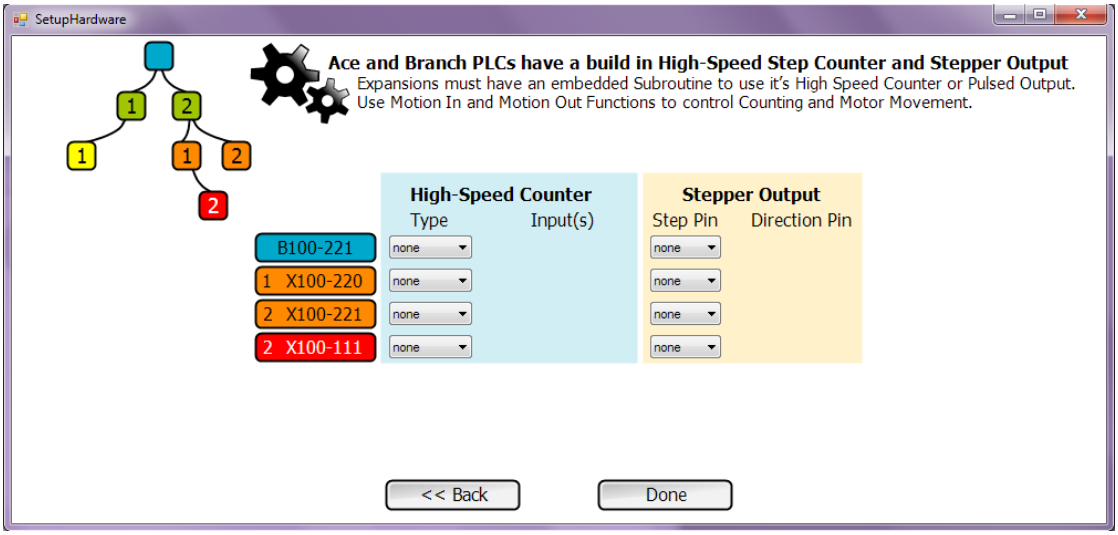


When you've got the definition set up like you want,

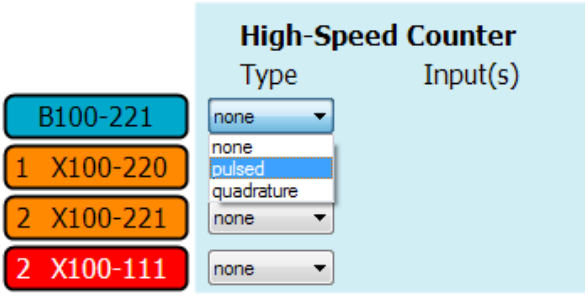
click Next.

The next Setup screen will allow you to configure selected digital inputs and outputs for dedicated use as high speed counter inputs or stepper motion control outputs.

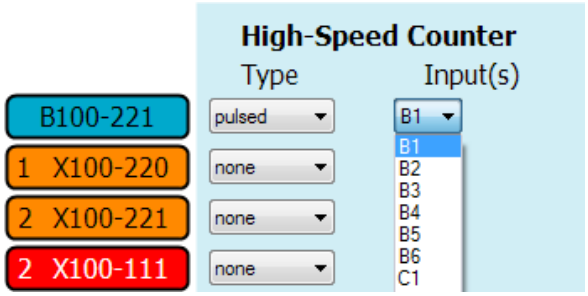
High speed counters and stepper motion can only be configured for modules that have a program. That would include the main PLC (Ace or Branch) and any Expansion units that have an embedded object subroutine program. The screen shot on the right shows the options that will be presented based on the configuration of the last few pages.



If you select the pull down arrow under High Speed Counter Type, you will see that you can configure for a simple pulse input, a quadrature input or none.

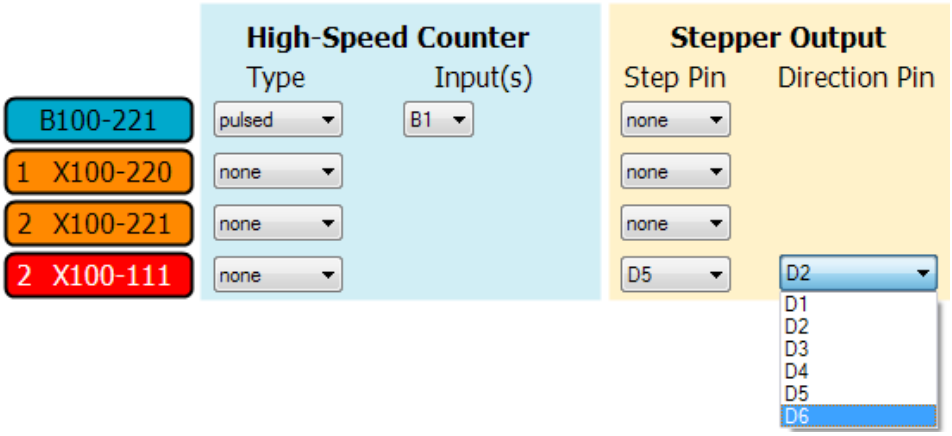


If you select a pulse input, you will next need to select one digital input on the module to dedicate to high speed pulse input. If you select configuration for a quadrature pulse counter, you must select one input each for the A and the B inputs. The selected inputs will be dedicated to HS counter usage and not available to use for general IO.



The same process applies to stepper motion contol. Any module that has a program can be configured to control one stepper motor. You must select a dedicated digital output for the step pulse and a second one for the direction signal, as shown.

When you've finished the selection of high speed counting and stepper motion IO, click done. You've completed your hardware Set Up.



Entering Tag Names

All data in a vBuilder program is referenced by tagnames that you assign. These tagnames can be up to 16 alphanumeric characters long. This should be sufficient to enable you to choose names that are clear and meaningful for your project. Tagnames can be added three different ways. You can define a tagname variable either by -

- Selecting the Tag icon, the particular data type and typing into the next unused entry
- In a function block dialog box, type in the new name of a tagname variable
- In a subroutine Input definition dialog box, type in a new tagname

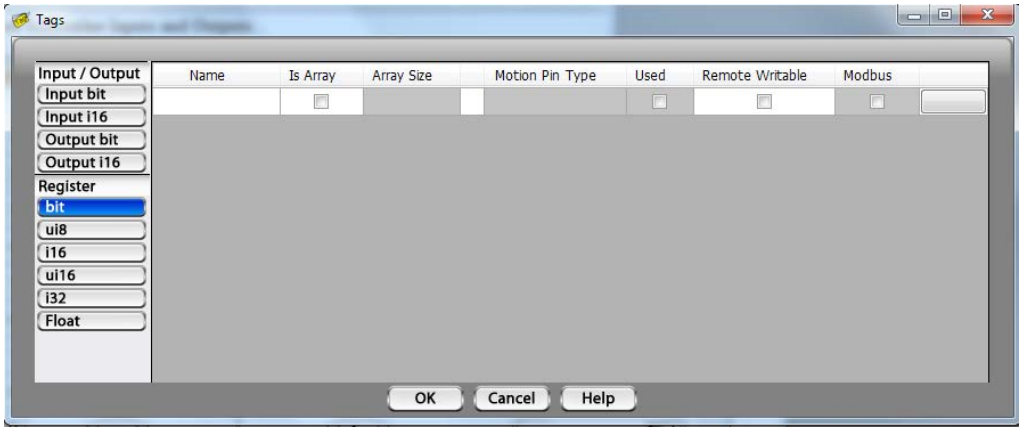
All three methods accomplish the definition of new tagname variables.

Tagname variables are only available for viewing and entry in context. In the main program, only main program variables are in context and can be defined, edited, deleted and viewed. In a subroutine, only the object data associated with the subroutine's object type are in context. In an embedded object, only the the embedded object's data is in context.

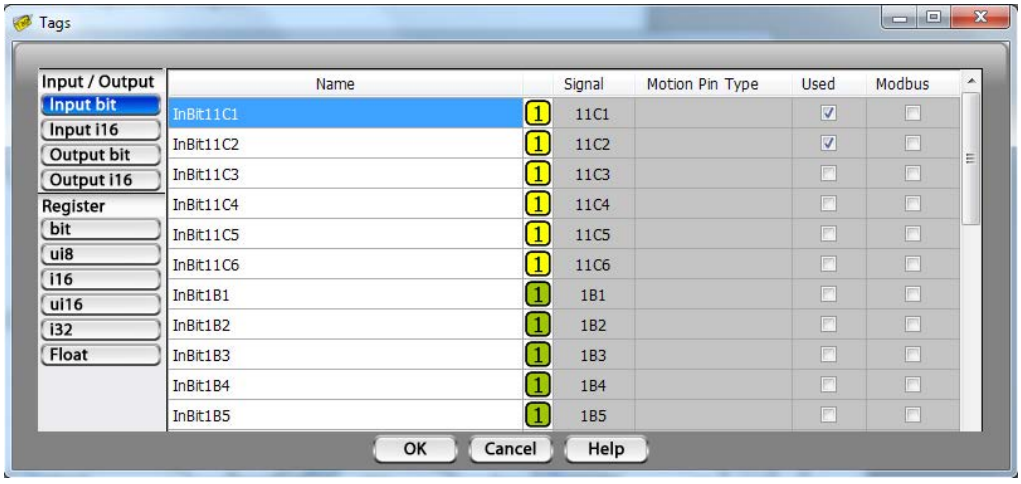
Defining a New Tagname Variable through the Tag Icon

Near the left hand end of the top tool bar is an icon, labeled "Tag", like the one shown here. Double click on it.

When you double click the Tag icon, a dialog box, like this one will pop up. Along the left hand side of the dialog box is a selection list for tagname variables of various types. At the top of the list is data types specifically associated with Inputs and Outputs. Below those, under "Register", you see all of the data types available in vBuilder, from bit to Float. These are general purpose data.



If you select one of the Input/ Output types, you will see a listing of tagnames. When you set up your project, default tagnames are created for all of the system IO. You can edit any of the tagnames and change them to whatever name you wish. Notice the column labeled "Signal". This column lists the module and signal identifier associated with each IO point. For example, the tagname InBit11C1 lists 11C1 as its connection. That indicates that it is signal C1 of a module connected two levels down from the main PLC - through the main PLC's port 1 (the first 1), then the expansion unit at the next level's port 1 (the second 1). The colored, numbered box also indicates the unit the IO point resides in, based on the Hardware Setup tree graphic.



If there is motion configured for the signal (remember that only IO points on modules with programs can be configured for high speed digital inputs or stepper motion outputs), it will be listed in the motion column. The "used" check box indicates whether the tagname has been used in the program. The Modbus check box indicates whether the tagname variable has been set up for Modbus communications. Only tagname data for the main program can be configured for Modbus data transfers.

For IO tags, all of the IO points will be assigned default tagnames at setup. If you want to modify these tagnames to more meaningful names (recommended), you must do so by going through the Tag icon, selecting the name and typing in the new name.

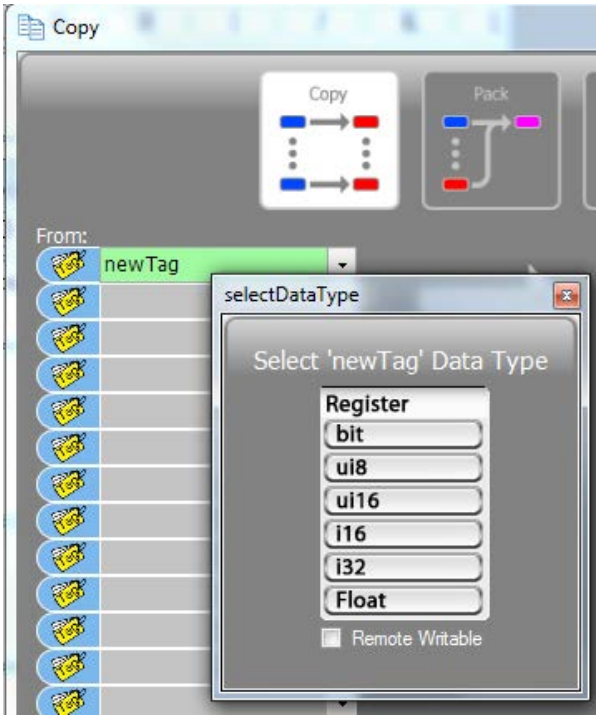
General purpose tagnames can be created by selecting the data type under the Registers selection list, then typing in your tagname. For each of these, there is a checkbox under the column header "Is Array". If you want to create a variable array, check this box. If you check the "Is Array" box, you must enter a number in the next column to define the size of the array.

You can come back and edit, add and delete tagnames at any time. Remember that if you delete or change the name of a tagnamed variable that you have used in your program, you will get an error indication informing you that you must resolve the problem

You can also perform exactly the same operation by selecting a Tag icon, wherever it appears. This happens in every dialog box that has parameters and the Input/Output dialog box.

Defining a New Tagname Variable through a Function Block Dialog Box

The second method for defining new tagnames is through a function dialog box. When you place a function block that requires the entry of data or a tagname for a variable, you can type in a new tagname for that variable. If the data type required in the dialog box is restricted to a particular data type, a tagname of that type will be created with the name you type in. If a variety of data types are available for the operation, when you type the name in and press Enter, a selection list, like the one shown on the right, will pop up. Select the data type that you want for the tagname.



During the data type selection, there may also be a check box for "Remote Writeable". Selecting this box will enable the variable to be written to via a communications link, such as Modbus. If you do not select this box, the variable will not be allowed to be written to by a remote device.

Note : Only main program tagnamed data can be remotely accessed through communications.

This method cannot be used to edit any tagnames, including IO tag names. To edit or delete tagnames, you must access them through the Tag icon.

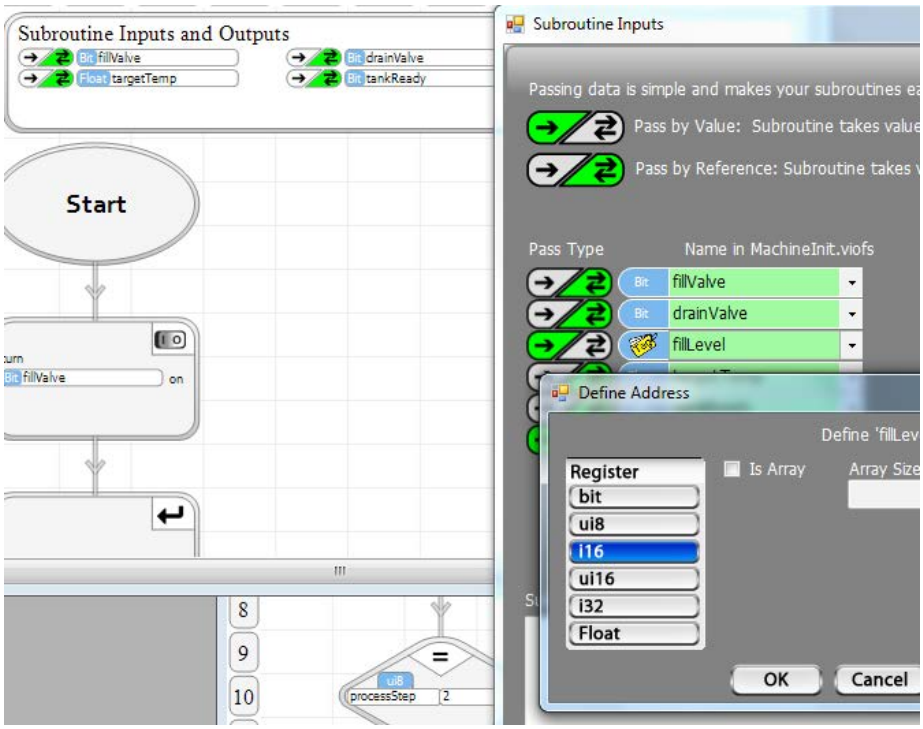
Defining Subroutine Object Tagname Variables through the Subroutine Inputs and Outputs Dialog Box

The third method that can be used to define new tagnamed variables occurs when you are defining the variables that are passed in and out of a subroutine. When you double click the Subroutine Inputs and Outputs block and bring up a dialog box to use to define the passed data, you will get a dialog box, like the one shown below. As you define the parameters to be passed, you can type in the name of a new tagnamed variable.

A new variable entry is shown in the illustratration. The variable "newTag" has been typed into the parameter definiton box. When Enter is pressed, the Define Address window, shown on the right, will pop up. You must select the data type of the variable that you are defining. In the process, you can define the variable to be an array and size the array.

Full arrays can only be passed by reference. Individual array elements can only be passed by value. If you configure to pass an array element, you must place the array index in square brackets [] .

As with the function dialog box definition of tagnames, you cannot edit or delete tagnames using this method. In order to do so, you must select a "Tag" icon.

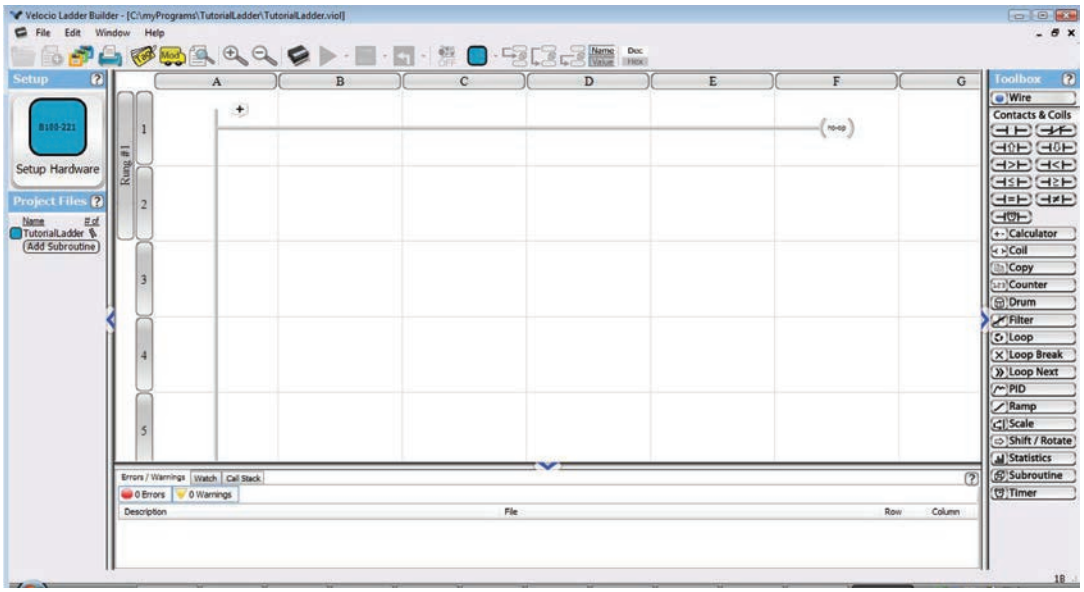


Program Entry

Program entry in vBuilder is very straightforward. If you have gone through the tutorials, you have likely learned most of the basics. For either Ladder Logic or Flow Chart programming, the entry of a program consists of selecting standard function blocks, dropping them in the required location and defining their operational details through the dialog boxes. In Ladder Logic, there will be some requirements to complete rung connections. In Flow Chart programming, you must connect the various blocks together to define the program flow.

With vBuilder, programs are normally developed with subroutines; maybe even embedded subroutines. A big part of the process is figuring out how to break up the program logic into subroutines that make logical sense. With just a little practice, this becomes pretty natural.

Entering a Ladder Logic Program

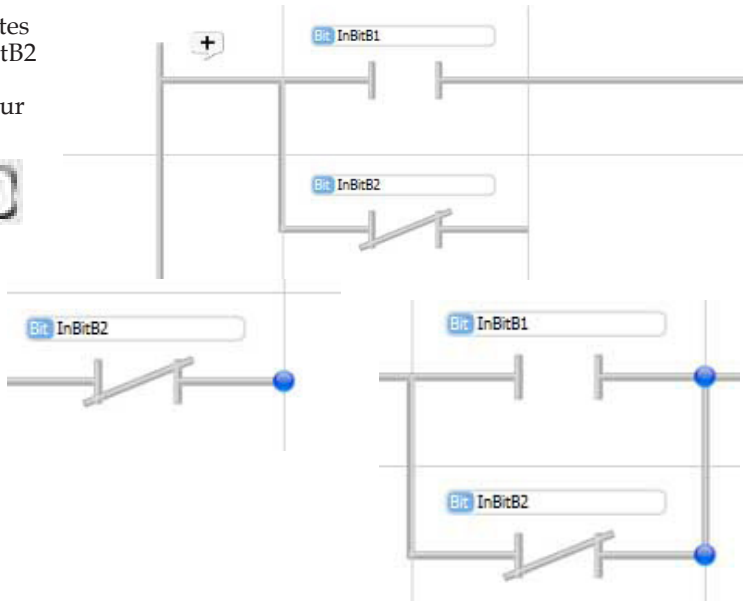


A ladder logic program starts out like the illustration on the right. The central area is where the ladder logic program is constructed. On the right side is the Toolbox of program components that are available. The Setup screen on the left shows the hardware configuration. The list under Project files shows all of the programs (main program, subroutines and embedded subroutines) in the project.

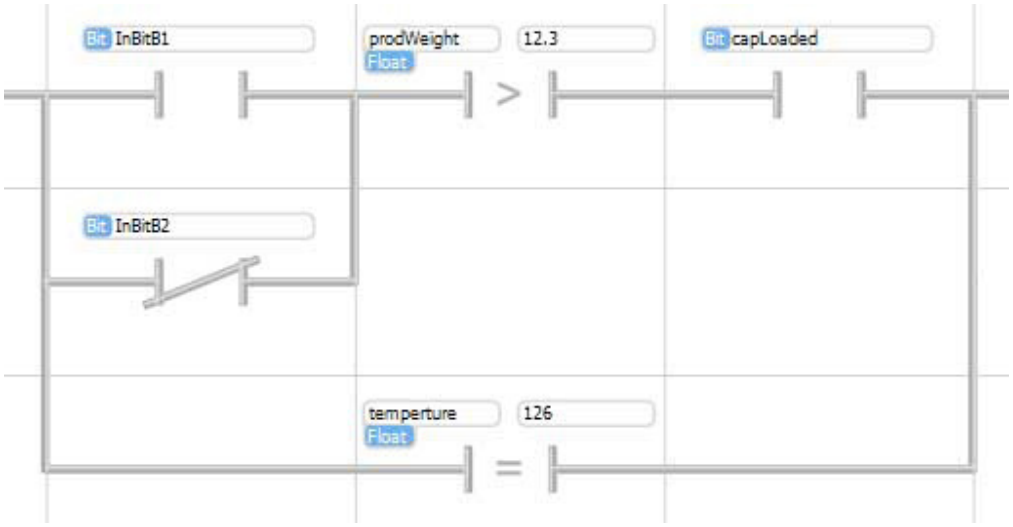
The ladder program starts with a power rail, on the left side. Each logic rung connects the power rail through a combination of contact blocks, to an operational block on the right hand side. Programs are executed one rung at at time, from top to bottom, then processes IO and repeat.

The description of all of the ladder function blocks is covered in Chapter 4 : Ladder Logic Programming.

As you are placing contacts, you will encounter situations when the full contact connection will not be made. This enables you to connect it as you wish. Such a situation is shown on the right. When the second contact is placed below the first one, the left side connection is made automatically. The right side terminates without a connection. To complete the connection so that InBitB2 is in parallel with InBit1, select Wire tool, from the top of the Toolbox, then move your cursor to where you want to start your wire connection. In this case, we want to draw a line from the end of the InBitB2 block to the end of the InBitB1 block, so we move the cursor to the end of InBitB2, as shown. Click on that spot, then move to where you want to connect, as shown on the right. The line with blue dots on each end shows where the connection will be placed. Click on the top connection point. The connection will be made and the blue dots will disappear.



Using this free form connection method, ladder rungs of any complexity can be created quite easily. The illustration on the right shows a little more complex rung.

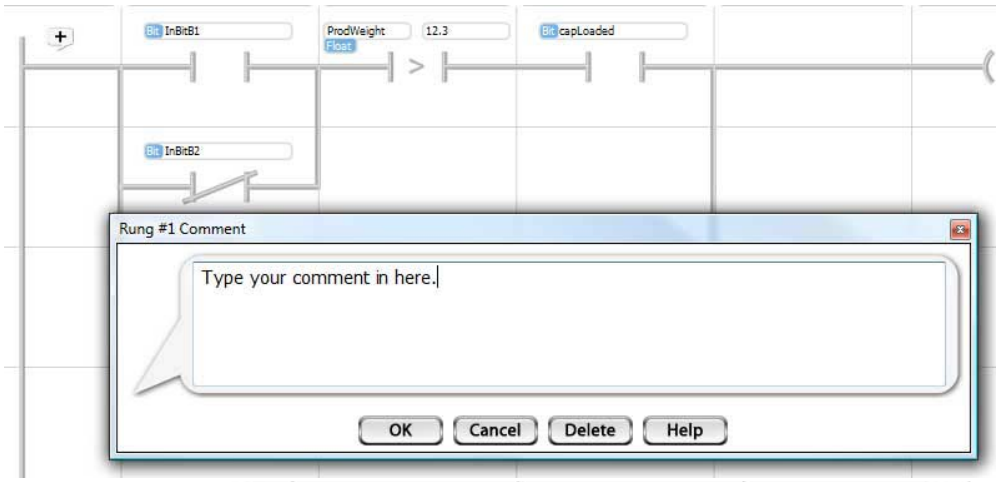


If you need to delete any connection, simply click on it to highlight it. After you have highlighted it, press the Delete key on your computer keyboard.

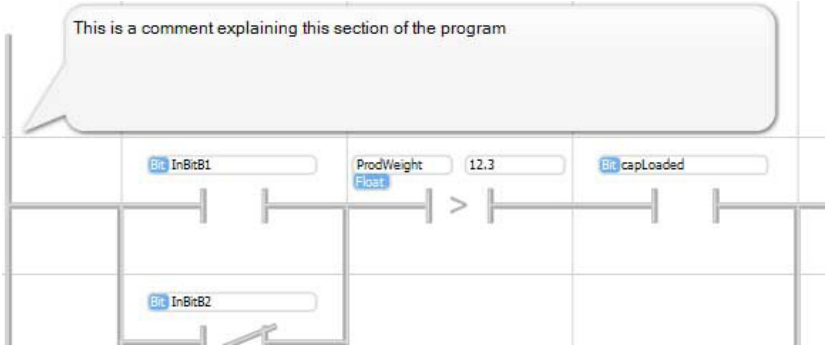
The program blocks that are located in the Toolbox, below the contacts (Coil though Timer), are operation blocks. They do something. The something could be turning an output on. It could be performing a calculation. It could be controlling a stepper motor.

Operation function blocks go on the right end of the rung. Every rung consists of "If this, then do that". The if portion is built up from contacts and can range from "if always" (no contacts), to something pretty involved. Every rung must have an operation block on the end. An example is shown below.

Notice the '+' symbol at the beginning of each rung. The plus symbol provides a way for you to add comments to your program. If you click on the '+' symbol, a window will pop up that will allow you to type in any comment that you want, as shown on the right.

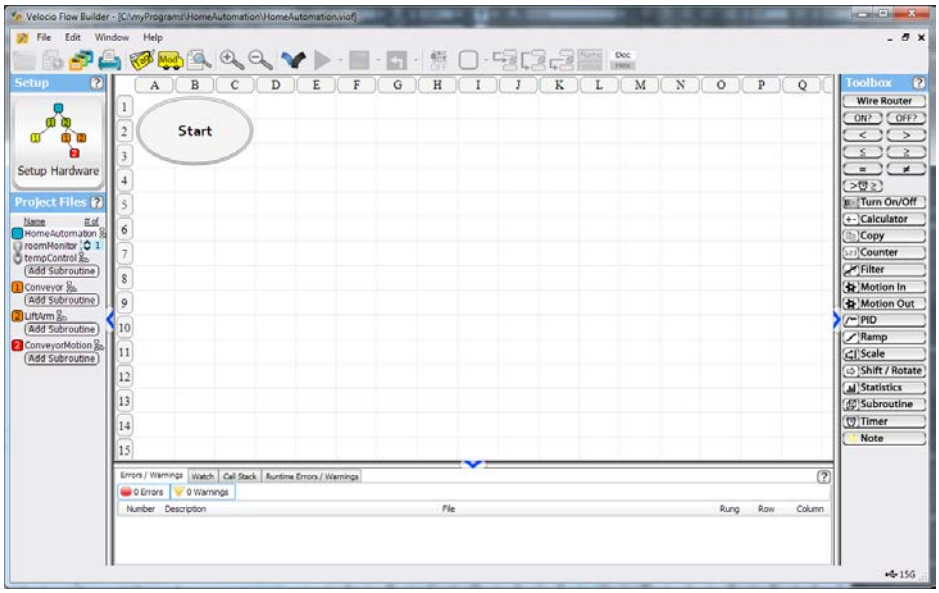


Type your comment and Enter. The comment will be placed above the rung, as shown. This isn't something that you are going to want to do on every rung. However, it is an effect tool for documenting sections of code or for explaining particularly complex rungs.



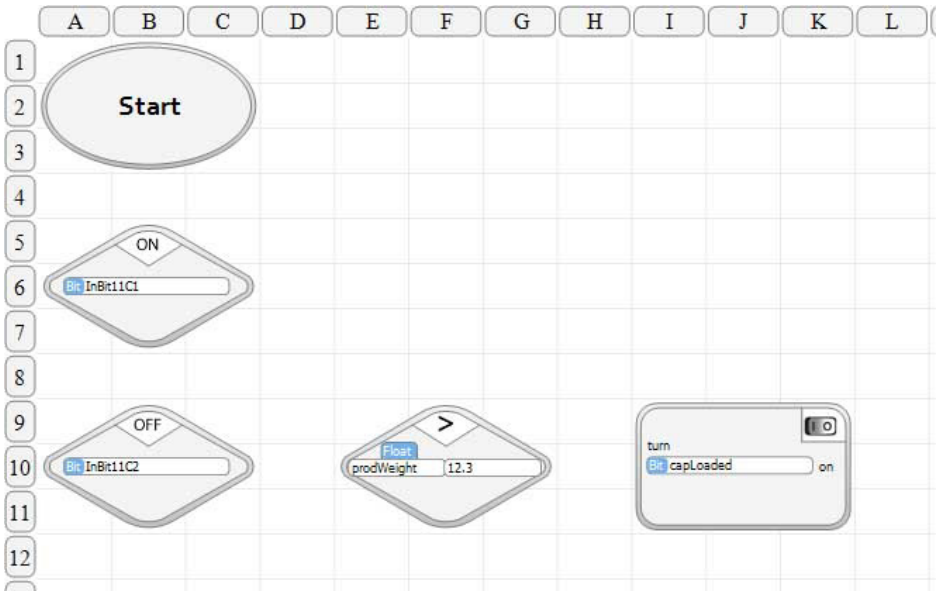
Entering a Flow Chart Program

A flow chart program starts out like the illustration on the right. The central area is where the flow chart program is constructed. On the right side is the Toolbox of program components that are available The Setup screen on the left shows the hardware configuration. The list under Project Files shows all of the programs (main program, subroutines and embedded subroutines) in the project.

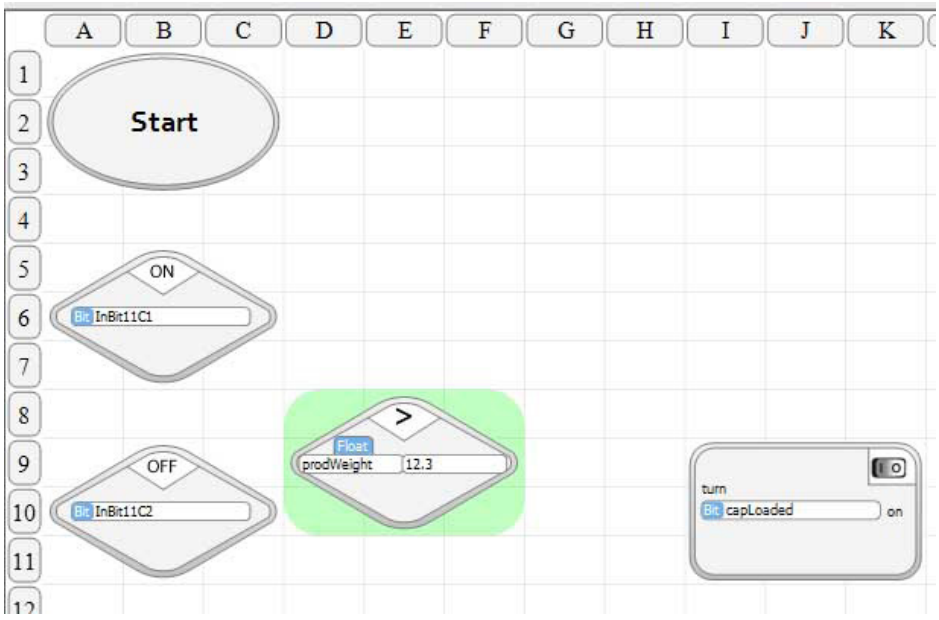


The flow chart program starts with a Start block. Entry of a flow chart program is free form. Pick a function block from the Toolbox, place it on the chart, configure it through the dialog box and connect it in the logic flow pattern required for your application. Program execution will follow the logic flow that you enter. Execution will flow through the logic, then process IO and execute the logic again, continuously. If you loop back to a previous program block, execution will pause for IO processing, then continue.

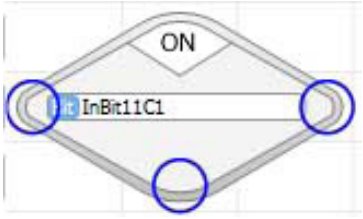
The illustration on the right shows a few function blocks placed in a program. When the blocks are placed, they are not connected. You should place them in the pattern that you want to connect them.



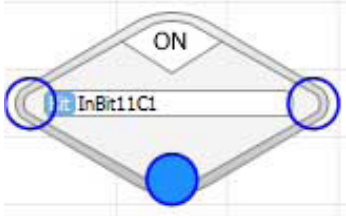
A program block can be moved at any time. To do so, left click the block and hold the left click while moving the cursor. The block will move where-ever you move it. While it is being moved, it will be highlighted with a background color.. A block can be place anywhere, except at a location that is already taken up by another block. If it is in a position where it is OK to place, the background will be green. If you move it over another block, the background will turn red, indicating that the location is not available for placement. When the block is where you want to place it, release the mouse left button.



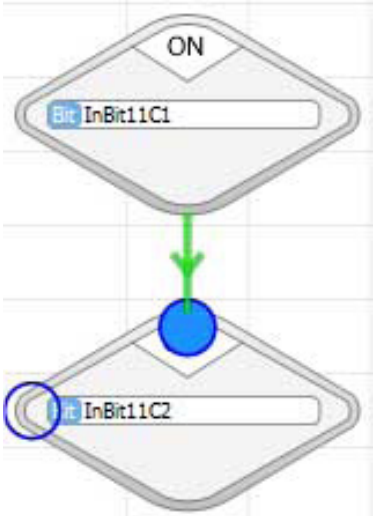
Roll your cursor over a program block. Notice that when you roll over a block, hollow blue circles will show up. These circles show potential block connection points where a connecting line can be placed to connect from the block to another block. As shown in the illustration on the right, decision blocks can exit to either side or out the bottom.



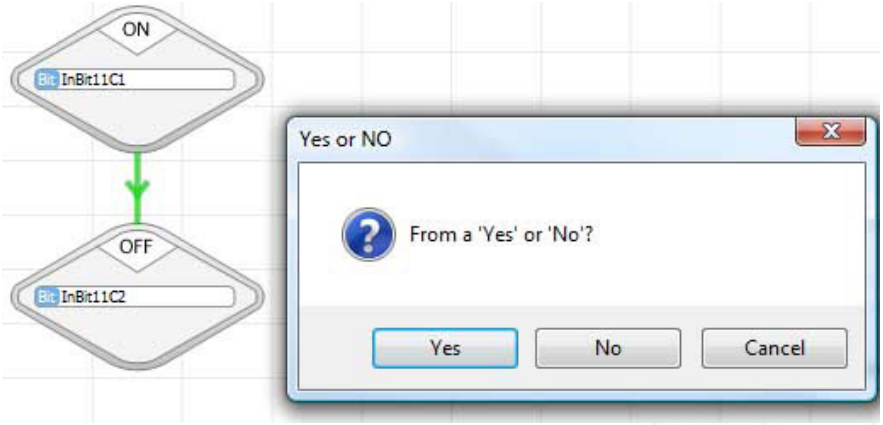
If you move your cursor over one of these points, it will turn solid blue. If you left click, when a point is solid blue a connection line will appear.



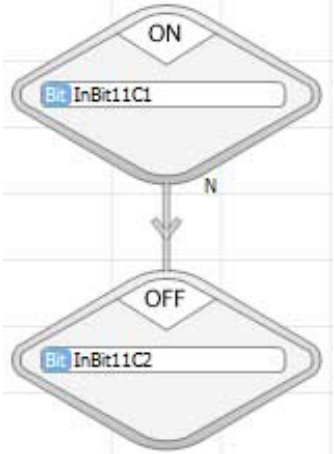
If you move the cursor, while keeping the mouse left button held down, to another block, that block's available entry points will show up as blue circles. If you move your cursor to one of those circle, it will highlight in solid blue, as shown on the right. If you then release the mouse, the connection will be made.



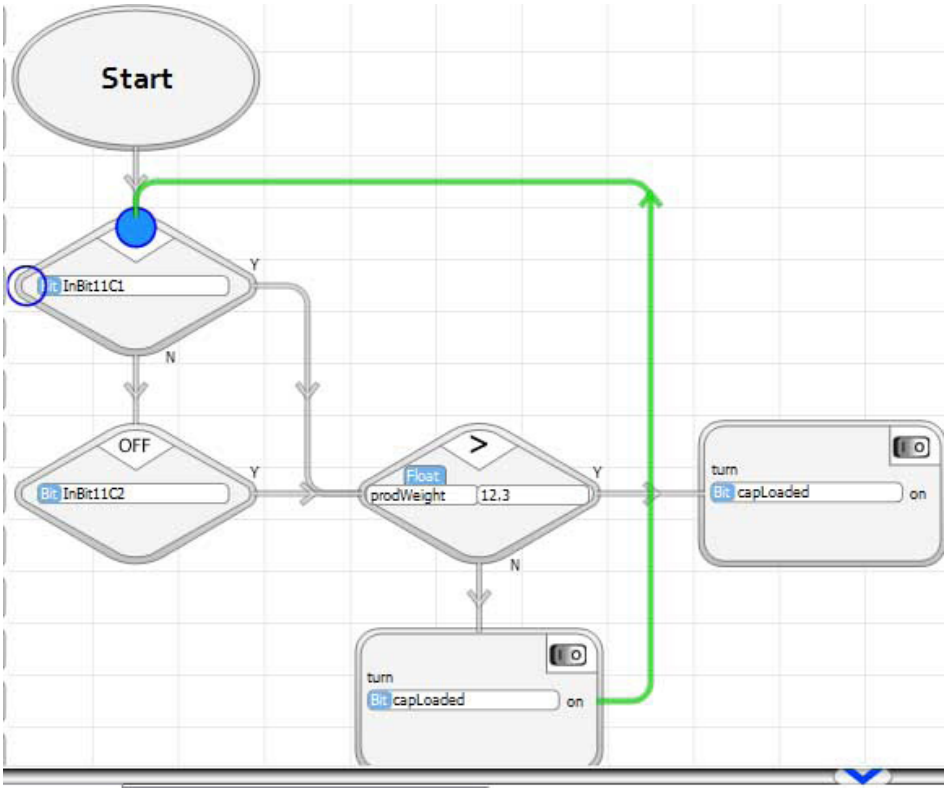
The first connection from a decision block to another block will require you to define whether this connection is for the case where the answer to the decision is "Yes" or "No". A dialog box, shown on the right, will pop up. If this is the flow for a "Yes" result, click the Yes selection. If it is for a "No" result, click the "No".



Once you place the connecting line and make the selection for Yes or No, the flow chart will show it, as shown on the right.

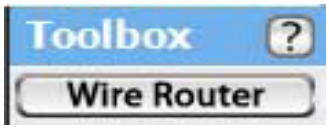


Continue to make connections, as shown.

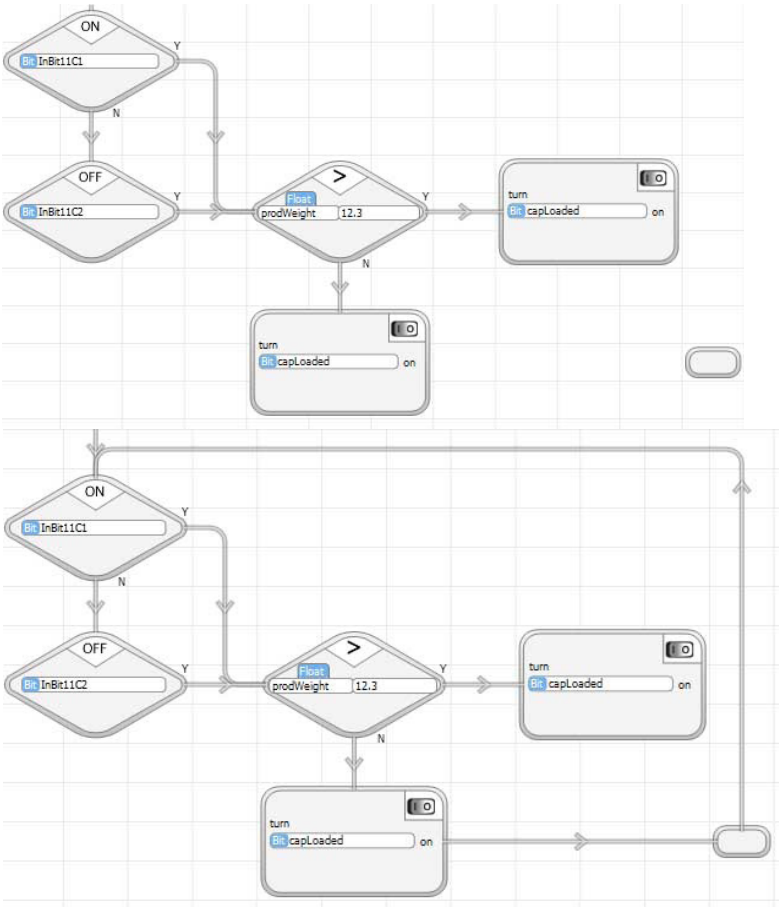


As you place connections, you may encounter a situation where the connection routing crosses paths or program blocks in a way that makes it difficult to follow. The illustration on the right shows such a case. In trying to make a connection from a turn off block back to the first decision block, vBuilder will route along the path shown in green. This is perfectly fine, program wise. However, it may make it more difficult to follow, when viewing it later.

The solution to this problem is found in the tool at the top of the Toolbox, labeled "Wire Router".



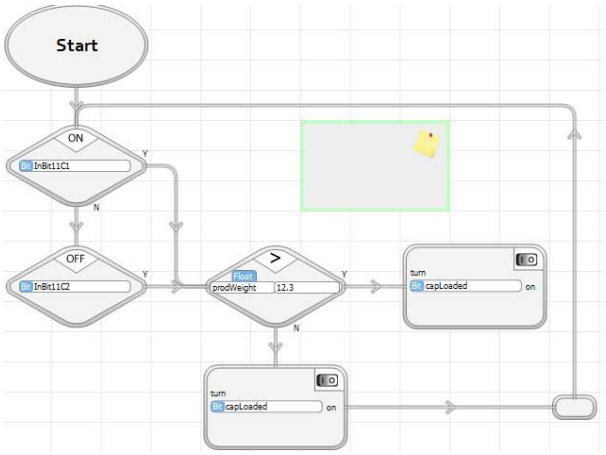
Select the Wire Router and place it in your flow chart, like shown on the right.



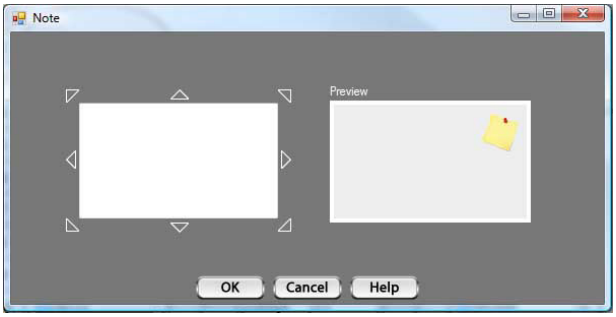
Finally, connect from the block that you want to flow "from", to the wire router. Connect from the wire router to the block that you want to flow "to". A wire router doesn't do anything for program execution. It simply is a tool to allow you to create "clean" flow charts.

At the bottom of the Toolbox is an icon for the Note tool. The Note tool allows you to place comments anywhere you want in a flow chart. Notes can be quite effective in documenting a program.

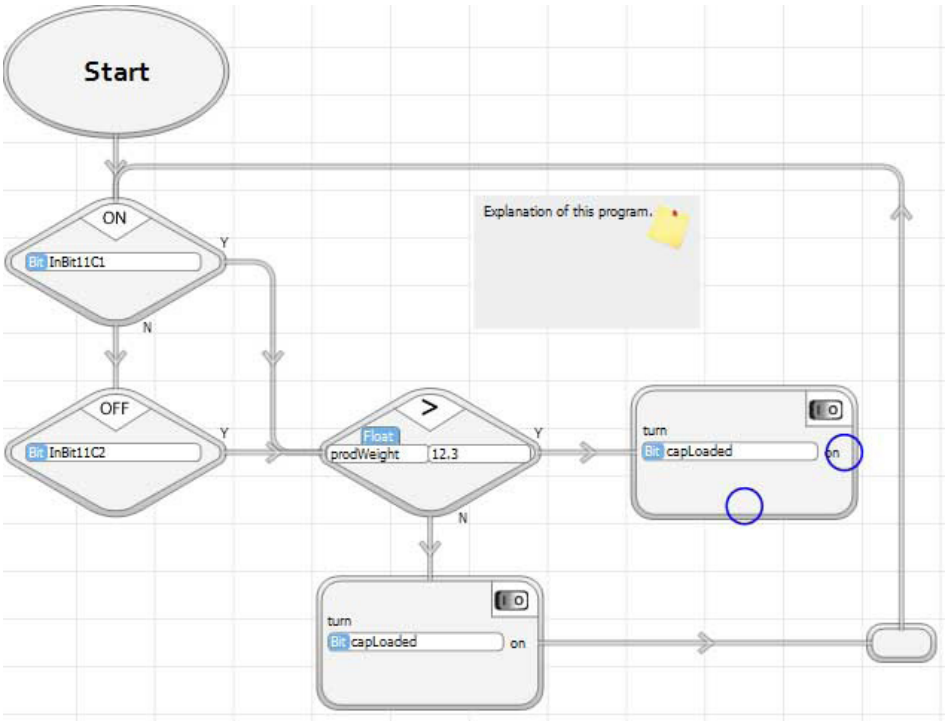
If you click the Note icon and move your cursor over the flow chart, the note block will move to where you move your cursor.



When you get the note where you want it, click the left cursor button again to place it. Immediately, a dialog box, shown on the right, will pop up. Click inside the box on the left and type whatever note you want. The box on the right will preview what the note will look like.



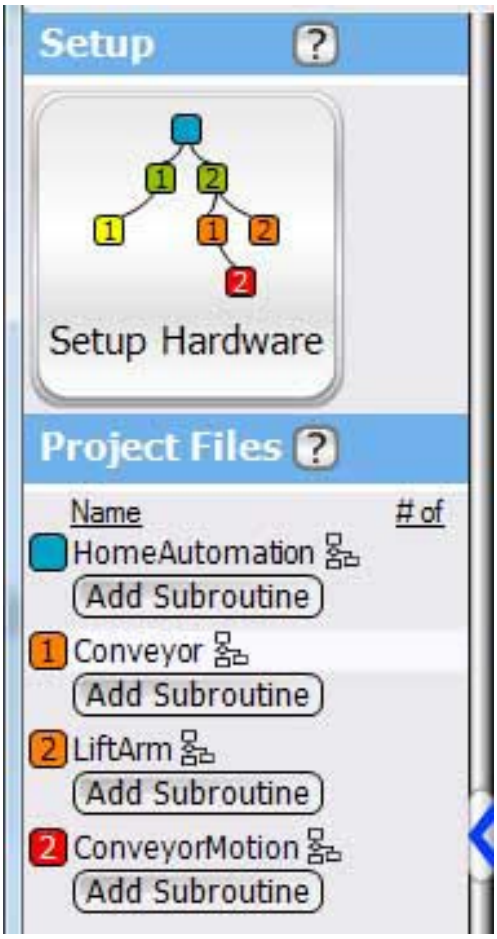
When you are done, click OK. The note will be placed in the program, as shown on the right.



Creating Subroutines

Creating subroutines is a fairly simple process.

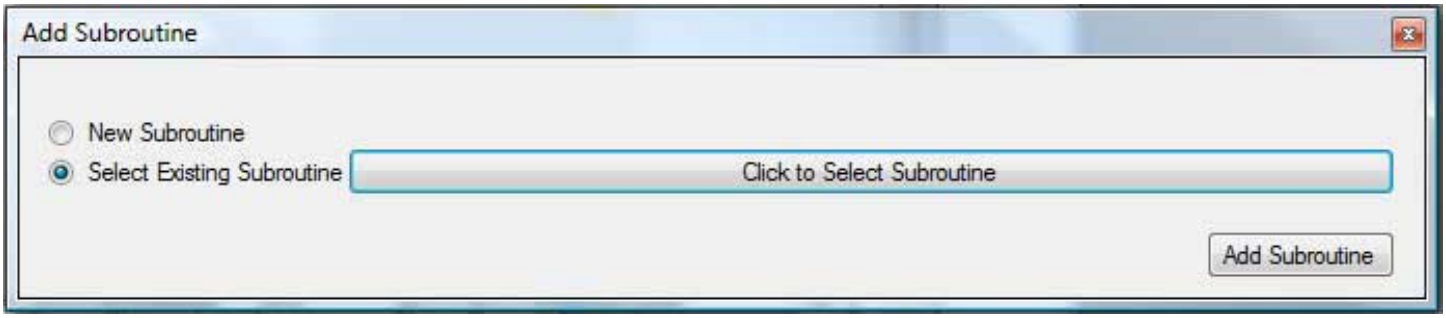
If you look on the left side of vBuilder, under Project Files, you will see a list of all programs that currently exist in the project. In the example shown on the right, HomeAutomation is the main program for project HomeAutomation (the main program is given the same name as the project). When Setup was performed to define the project hardware configuration and Embedded subroutines, three of the modules were defined as having embedded subroutines, which were given the names "Conveyor", "LiftArm" and "ConveyorMotion".



The colored, numbered boxes on the left of each program name indicates where the program resides. The unnumbered blue box is the main Branch module. The orange box labeled '1' indicates that the embedded object program, "Conveyor",is located in the Expansion unit labeled with an orange one in the Setup Hardware diagram. Likewise, the orange '2' and red '2' define the locations of "LiftArm" and "ConveyMotion".

Embedded Object programs are the main programs for each of the devices in which they are resident. They serve as subroutines to the program just above them. In other words, Conveyor is the main program residing in Branch Expansion orange 1 and serves as a subroutine to HomeAutomation in the main PLC. ConveyorMotion is the main program in the red 2 Expansion and serves as a subroutine to Conveyor in orange 1.

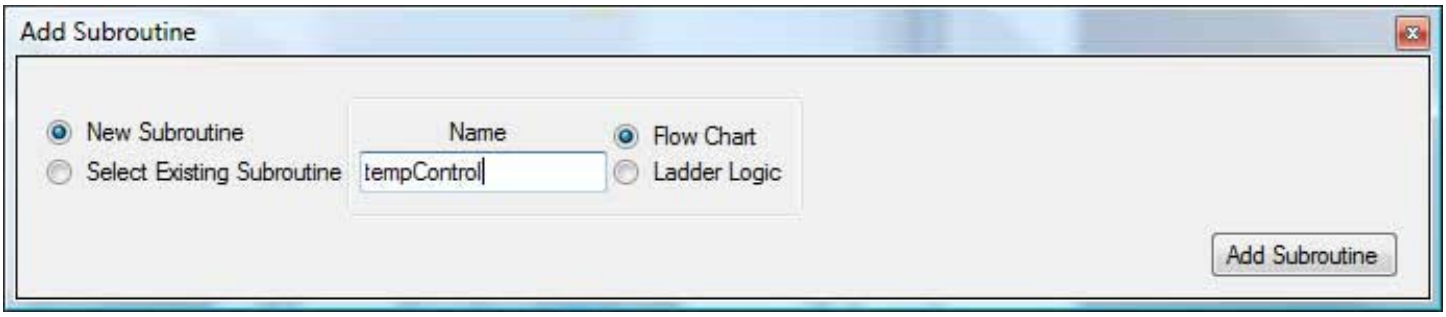
Additional subroutines can be created in each device. The creation of a subroutine starts by selecting the "Add Subroutine" button under the device where you want to put the subroutine. If you select "Add Subroutine", a dialog box, like that shown below will pop up. In this example, Add Subroutine under HomeAutomaton was selected.



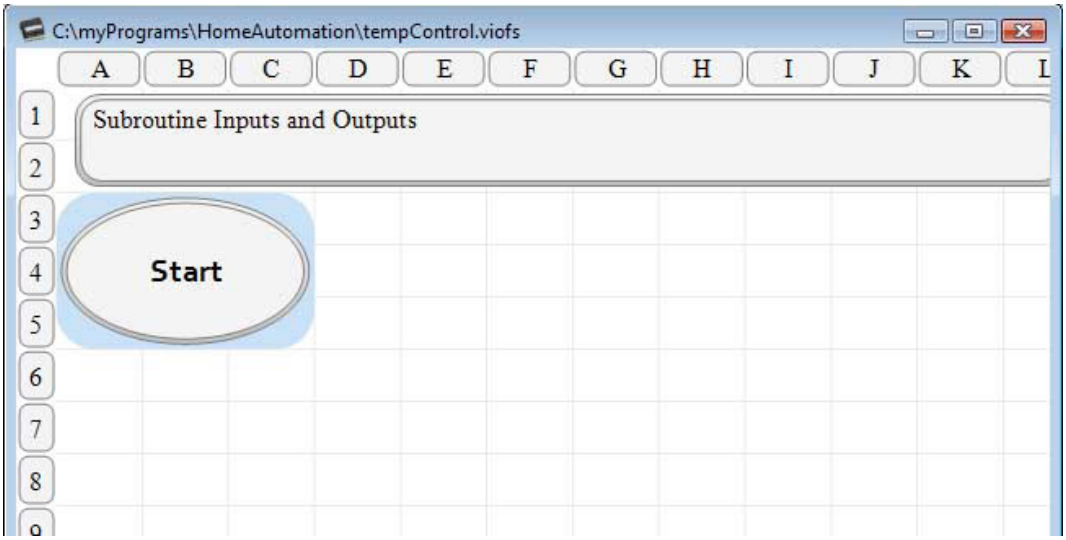
You can select either a new or an existing subroutine. If you have an existing subroutine that you want to re-use in this project, you simply select "Select Existing Subroutine", then select "Click to Select Subroutine" and browse to find the subroutine (probably in another project folder) to use. Once you've found it, click "Add Subroutine". The subroutine will be copied to this project's folder for use.

Re-using subroutines is a powerful capability of vBuilder. Over time, you can create your own library of standard subroutines that you can use in project after project. The potential improvement in development efficiency is very substantial.

The other option, and the one that you will use most often, is the creation of a New Subroutine. To do so, select "New Subroutine", type in the name that you want to use for the subroutine, and select Flow Chart or Ladder - then click "Add Subroutine".

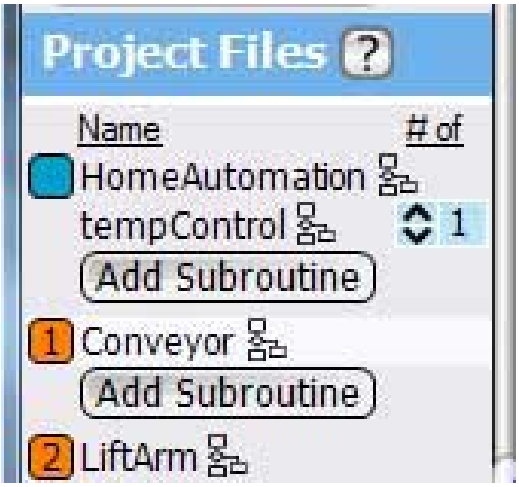


When the Add Subroutine button is pressed, the new subroutine will be created. A new window, with the subroutine will pop up. The subroutine starts out like all programs do - blank, with a Start block for a flow chart program or an empty ladder chart for ladder logic.

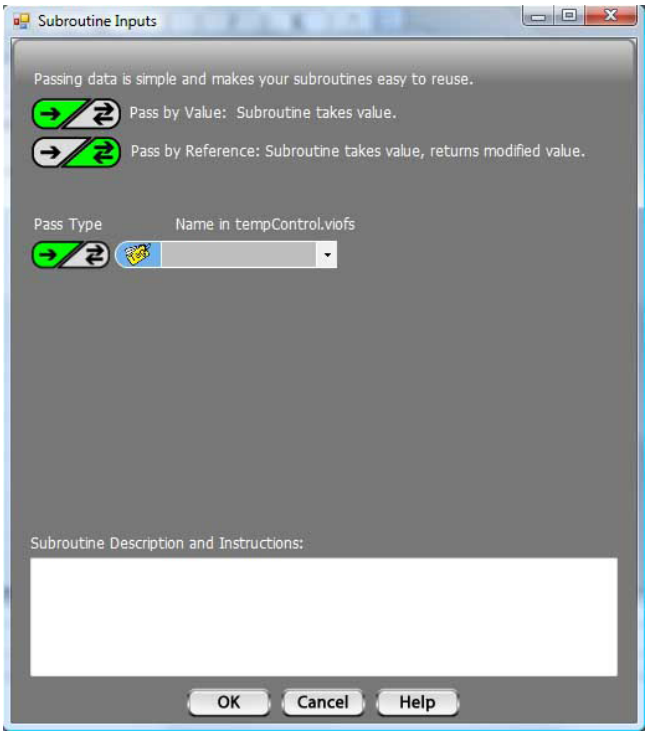


The new subroutine will show up on the list of Project Files, as shown on the right. Subroutine tempControl is shown under HomeAutomation, since we created it by clicking the "Add Subroutine" under HomeAutomation. That means it will be a subroutine located in the main PLC and can be called by the HomeAutomation or other subroutines under HomeAutomation.

Notice that there is a numeric adjustment control with the number '1' next to tempControl. The number (currently 1) is the number of instances of the object tempControl that are present. By adjusting with the arrows, the number of instances can be set to any value between 1 and the limit set by vBuilder (currently 16). Each instance of an object has its own object data.



Creating Subroutine Inputs and Outputs Lists



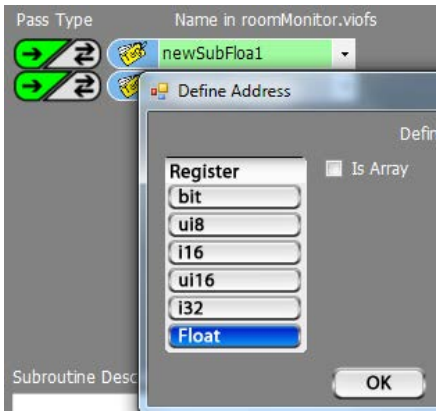
If you click on the "Subroutine Inputs and Outputs" block in the new subroutine, you will get a dialog box, shown on the right. Using this box, you can define the list of data and references to data that gets passed in, each time the subroutine is called.

The top of the dialog box shows a pass key. It shows that a pass type indicated by a single arrow coming in, is pass by value. Pass by value means that a numeric value is passed from the calling routine to this routine. If the numeric value comes from a tagnamed variable in the calling routine, the subroutine cannot affect the value in the variable in the calling program. It just uses the value passed in.

The symbol with the highlighted arrows going both ways is a pass by reference. Pass by reference means that a "reference" to a data item is passed in. No actual data is actually passed into the subroutine. The tagnamed data for the reference is data that the calling routine either has or has access to. This is the way data is passed out of a subroutine. To pass data out, the subroutine uses the reference to place a numeric value in tagnamed variables that the calling routine has access to.

When you first create a new subroutine, if you select the pull down to select from a list of available tags for the pass list, you will see that it is empty. All subroutines are objects, with separate object data that is "owned" by the instance of the object. Since you just created the subroutine, you haven't defined any tagnamed variable for it yet. You can either select the Tag icon, enter a list of tagnames to be used for Subroutine Inputs and Outputs, then come back and select them in this dialog box, or you can simply type in new tagnames in the boxes under "Name in xxxx". The most common solution is typing in the new names in this dialog box to create the new Subroutine Input and Output tags. After you've done that, you can go back into the Tag list, by selecting that Tag icon. You will see that all of the new tagnames are on the lists.

To create a new list of subroutine Input and Output tagnames, using the Inputs and Outputs dialog box, type in a new name (like the very unimaginative "newSubFloat1" shown) and hit Enter. When you do, a box will pop up, asking you to select the data type. Click on the data type the you want. For newSubFloat, the selection should be Float.



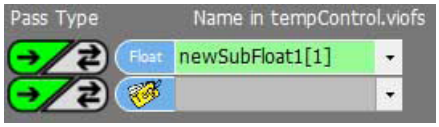
If you select OK, the next thing you need to do is select either the single arrow in or the arrows both ways to define whether this is a pass by value (single arrow in) or pass by reference (arrow both ways). Pass by value passes a numeric value into the tagnamed variable that you just created, each time the subroutine is called. Pass by reference passes a "handle", or a means to access a variable that is either located in the calling routine's data, or is a reference that was passed into it (meaning that the actual tagnamed data is located in another level of routine).

Passing Arrays and Array Data

When you are selecting the data type, you can also select "Is Array". Selecting "Is Array" will define the tagnamed varaible as a variable array. Once you select "Is Array", you must enter the Array Size.

In a vBuilder program, and entire array can be passed in by reference, only. Passing in by value is only available to a particular element of an array. If you click OK, after entering a new tagname, without an index, and select "Is Array", the item will be placed in the Inputs and Outputs list and automatically set to be a pass by reference. In this case, you cannot select pass by value.

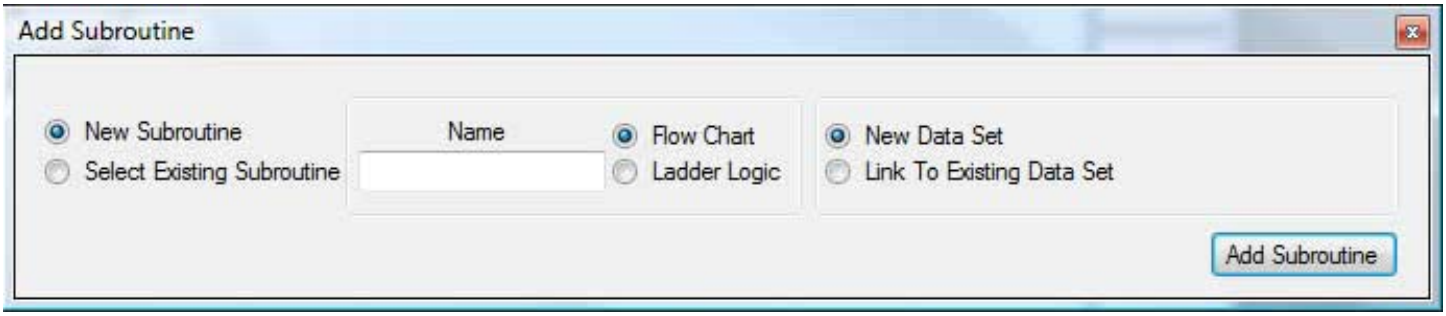
If you either give your array tagname an index when initially typing it in, or go back and edit it to place an index, you've put an element of an array on your pass list. An array element can be passed by passed by value only.



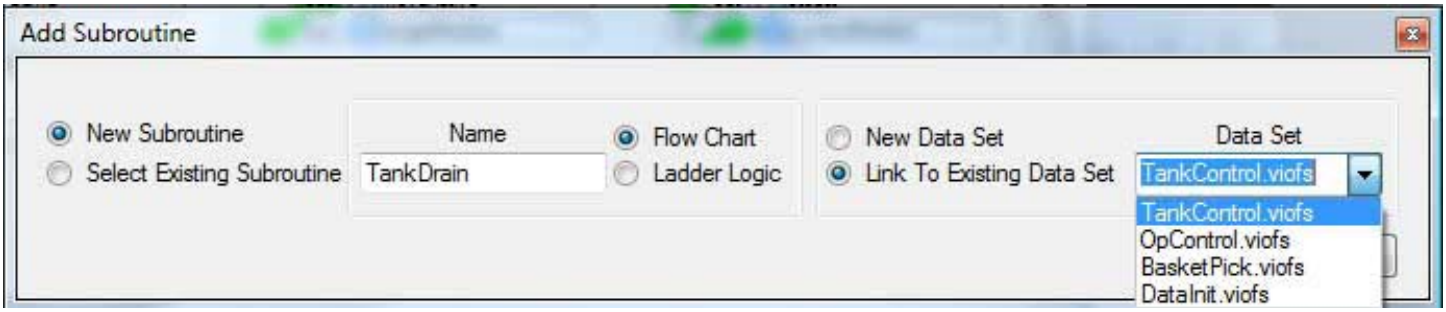
Linking Subroutines

Multiple subroutines can be linked to the same object data. What results is a single object, with multiple types of functionality. A common use for linked subroutines might be for controlling a machine or a subsystem. It very often makes sense to have an initialization subroutine, an operating subroutine and a shutdown subroutine. All three would operate on the same object, but they would be three different sets of functional logic.

The linking of subroutines occurs when a subroutine is added. Once you have created a subroutine, the next time you select "Add Subroutine", the dialog box has an additional feature. A selection for either "New Data Set" or "Link to Existing Data Set" is offered. If you select New Data Set, you will be telling vBuilder to not link this subroutine to any existing subroutine.



If you select "Link to Existing Data Set", you will be telling vBuilder that you are adding another subroutine to an existing object. The dialog box will allow you to select from all of the existing objects to link to. Each object is identified by the first subroutine created in the list of linked subroutines. The example below shows the creation of a new subroutine, given the name TankDrain. The selection is to make it a Linked Subroutine and link it to TankControl.



After the linked subroutine is created, it will show up in the Project Files list. The illustration on the right shows TankDrain as a subroutine that is part of the TankControl object. Two other subroutines, TankControl and TankInit are also part of the same object. As members of the object, all linked subroutines have access to all of the object data. If you create an object tagname while editing one subroutine, it is available to all linked subroutines.

Notice the number 3, with the adjustment controls next to TankControl. The three indicates that there are three instances of the TankControl object in the project. You can set the number of instances by using the up/down controls.

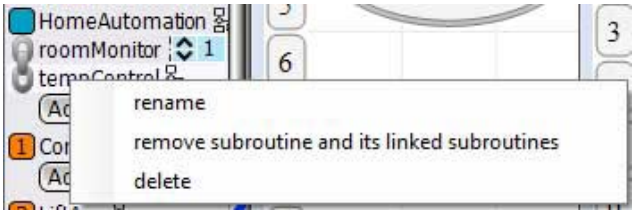
The chain symbol next to TankControl, TankInit and TankDrain indicates that these three subroutines are linked and provide different functionality for the same object.

Another case of subroutines that are linked, is object BasketPick. That object includes BasketPick, ArmInit, BatchProcess, TankTransfer and BasketDrop, all linked as the same object. The adjustment indicator show that there is one instance of this object.

Two other objects, OpControl and DataInit are unlinked subroutines with one instance each.

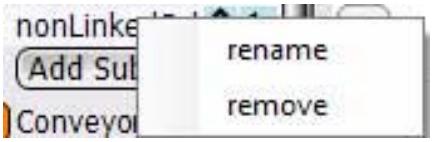
Renaming, Removing and Deleting Subroutines

Subroutines can be renamed or deleted. Entire groups of linked subroutines can also be removed together. The first step in doing so is to move your cursor over the subroutine name, under the Project files list and right click on the subroutine name. A box will pop up, which you can use to select to either rename, remove linked subroutines or delete.



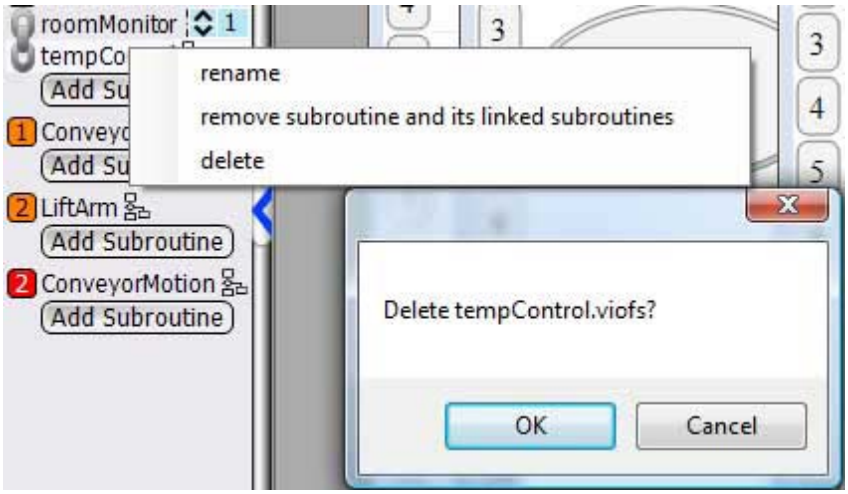
If you select 'rename' a dialog box will pop up to allow you to change the name. Just type the new name in the edit box and click on Rename. The subroutine name will be changed in the listing under Project Files, on the subroutine itself and everywhere in the project where it is used.

If the subroutine is not linked to other subroutines, the only two options that will be presented are rename and remove. If you select 'remove' the file will immediately disappear from the Project Files list and be disassociated from the project. It will remain in your project directory, however. If you want to add it back, simply do so with the standard 'Add Subroutine' function. If you do want to totally delete it, remove it, then use Windows Explorer to find and delete the file.



If you select a file that is linked to other subroutines, the three options shown above will be presented. If you select the 'remove subroutine and its linked subroutines' option, it will act like the remove of an unlinked subroutine, except that all of the linked subroutines will disappear from the Project Files list and disassociated from the project. The subroutine files will remain in the project directory. If you later add back any one of the linked subroutines, they will be all be added and linked.

You can also select a single subroutine within a group of linked subroutines to delete. If you do select delete, a verification box, shown on the right, will pop up. If you click OK, the file will be totally removed. It will be removed from the Project Files listing. Its file will be deleted from the project directory. Once you've deleted a linked subroutine, the only way to get it back is to recreate it.

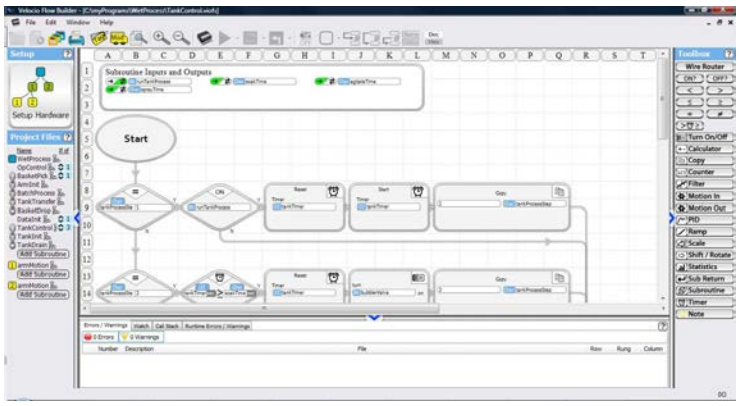


Basic Navigation

There are some basic navigation features that are useful during development and debug. Here is a description of a number of these.

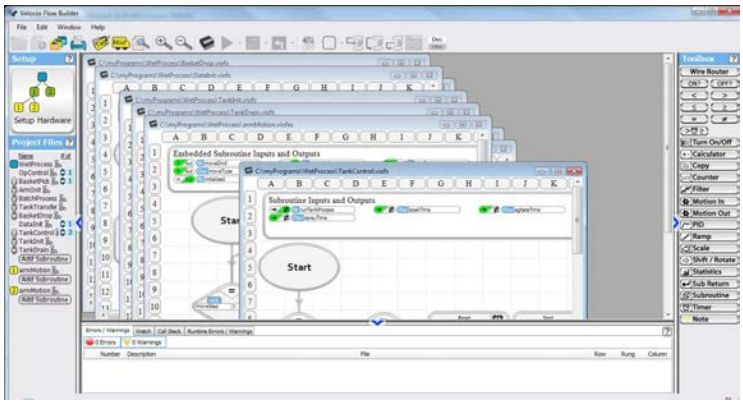
Switching Edit Screen between Full Screen and Tiled Modes

The edit screen, the screen in which the graphic programs are constructed, edited and debugged, can appear in either the Full Screen or Tiled mode. The Full Screen mode is shown on the right. If a program (main, subroutine or embedded sub) take up the complete area of the Edit Screen, it is in full screen mode. Full Screen mode is commonly used when programming or editing a program.



To change from Full Screen to Tiled mode, go to the upper right hand corner, above the Toolbox and select the small icon with two overlapping rectangles. When you do that, the Edit Screen will become Tiled and the little overlapping rectangle icon will go away.

In Tiled mode, shown on the right, multiple programs or subroutines are displayed at once and will overlap. The highlighted, or active routine will appear on top. You can size the individual program windows like any Windows program. Just grab the edge and pull or push.



To put any routine window in Full Screen mode, go to that routine's upper right hand corner and select the small rectangle icon (between the - and the X).

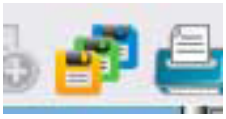
Selecting a Particular Routine Window

There are two ways to select a particular routine to come forward and become the active routine. Being active means that it is the subroutine (or main program) that is active for editing. If you are in the Debug mode, it is the routine whose data is available for display. The Tag database is set so any tag editing is for the active routine's object data. The active routine will come to the front of any tiling.

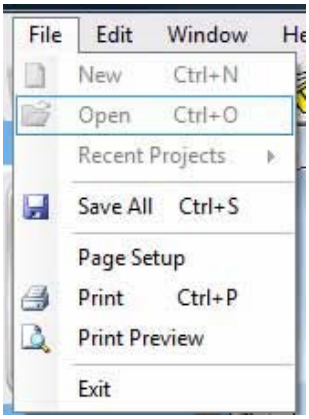
If you are in the Tiled mode, you can simply move your cursor over the routine that you want to make active and left click. That routine will immediately pop to the front.

The second method is to double click on the name of the routine in the list under Project Files. The selected routine will pop forward and its name will be highlighed under Project Files with a light background. This works in both the Full Screen and the Tiled modes.

Saving the Program



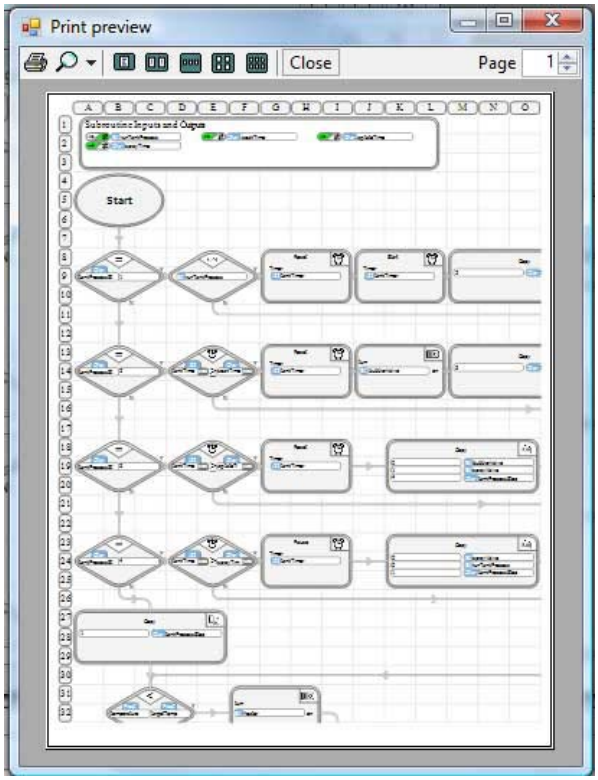
The program save function operates like most Windows programs. Either click on the three layed disks icon on the top toolbar, or select Save All from the File menu. In either case, all project files will be saved to the project directory.



Print/Print Preview

The active file can be printed or previewed for printing. This can be done, either by selecting Print or Print Preview from the File menu, or by selecting the small printer icon on the top toolbar.

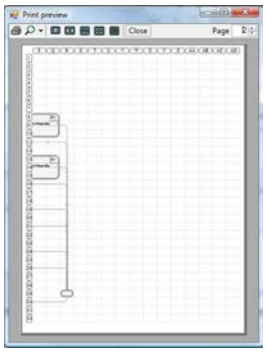
When you either select Print Preview from the file menu, or select the print icon, a preview of the active routine's printout will come up. Since vBuilder is a graphical programming language, in which you can build a program which may extend to any length or width, it commonly will not fit on a single printed page. vBuilder will print the program such that pages can be placed side by side and top to bottom to show the full program. An example is shown on the right and below.

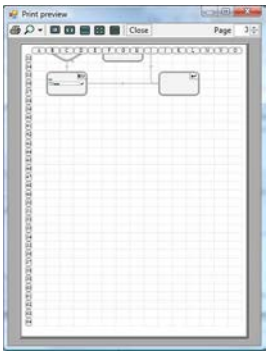


When you perform a print preview, a preview screen will pop up. Normally the screen will display the first page and have a selector wheel in the upper right corner, which allows you to select any other page. Since vBuilder programs are labeled with row and column numbers, it is clear how the various pages fit together.

Another thing to note in preview mode is the toolbar at the top allows you to zoom, show various views, print to a printer and close the preview.

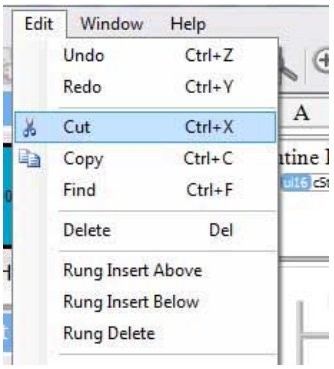
The second and third pages of this example are shown below on the right.



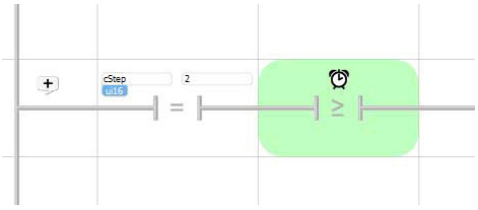


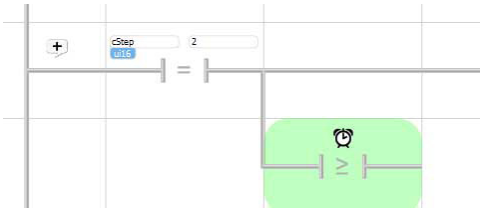
Cut (& Paste)

You can cut a function block from Ladder or Flow Chart program. If you cut it and move it somewhere else, it becomes a Cut & Paste. If you cut it then right click, it is effectively a delete. To Cut a function block, first select and highlight it. After you have highlighted the function block, you can either select Cut in the Edit menu, or you can press Ctrl+X. Either way, it will cut the block from the program.



After you have Cut a function block, it is attached to your cursor. If you right click your mouse, it will be deleted. If you don't right click the mouse, but move to another location, you can left click the mouse to drop it at a new location. The two screen shots below show a function block selected, then Cut and the cursor moved.

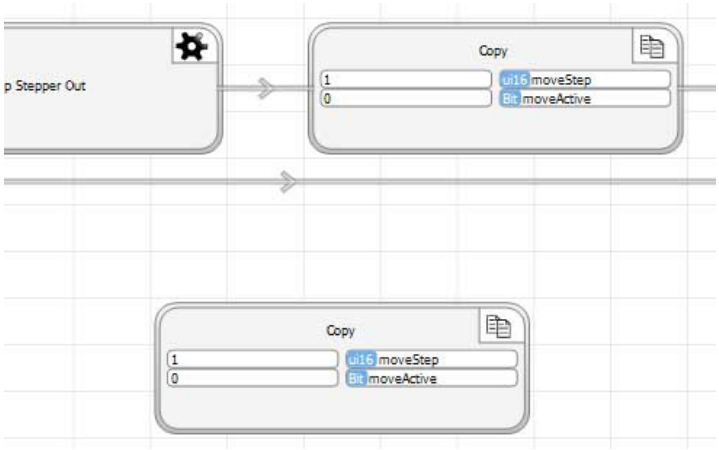




Copy & Paste

Copy and Paste is similar to Cut and Paste. To Copy and Paste a function block, first select and highlight it. Next, you can either select Copy from the Edit menu, or you can press Ctrl+C. A copy of the block becomes attached to your cursor. You can move it anywhere within the file, then left click to place it. All of the dialog box selected configuration details will be copied from the source block.

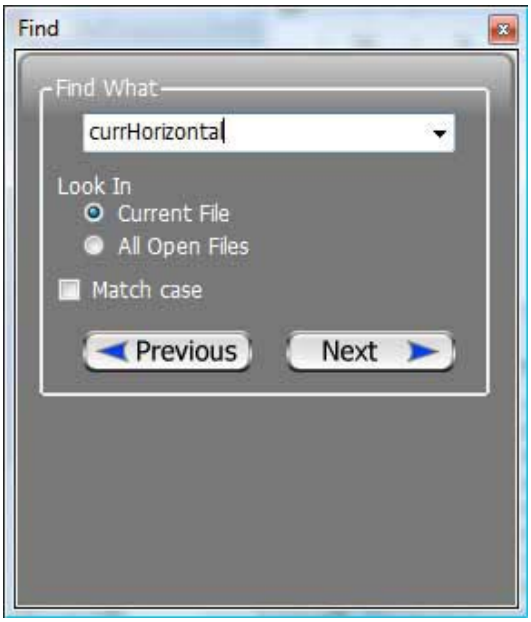
The screen shot on the right shows a block that has been copied and pasted.



Delete

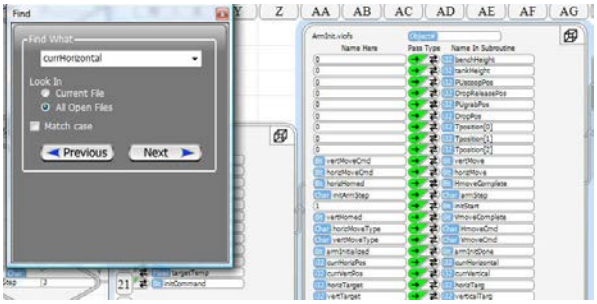
To Delete a function block, select it, then either select Delete from the Edit menu, or press the Delete key on your computer.

Find

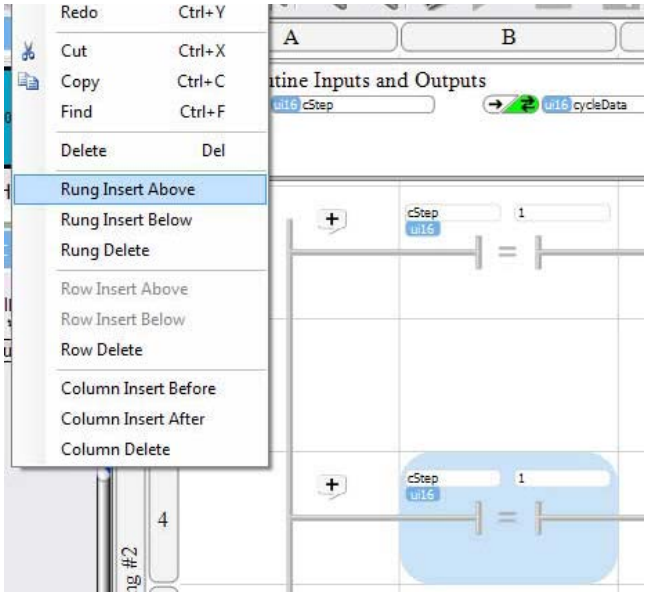


If you select Find from the Edit menu, you can search for any text. The dialog box at the right will pop up. Type in the text that you want to search for, select whether to look in the currently active file or in all open files, then press Next. If the text is found, the file that it finds the text in will come forward, become active and the block that the text is found in will be highlighted. Each time you click Next, it will find the next function block with the text in it. If the last instance is found, the next time you click Next, it will cycle through again.

The screen shot below shows a case where the text is found and the block highlighted.



Rung Insert Above
Rung Insert Below
Rung Delete



These three functions are only available when programming in Ladder Logic. If you select a function block on a rung, then open the Edit menu, you can do one of these functions.

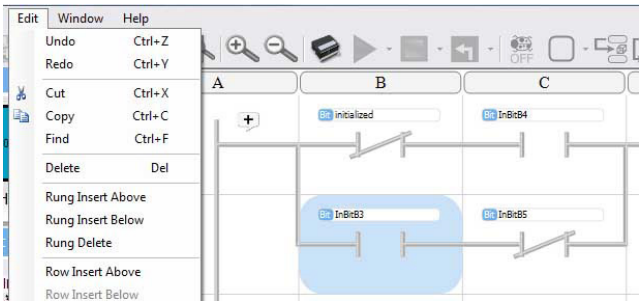
If you select Rung Insert Above or Below, a ladder rung will be inserted into the program (either above or below the line with the selected function block). The line will contain no logic. If the rung is added to the main program, it will be terminated with a no-op. If it is added to a subroutine, it will be terminated with a subroutine return. An example is shown below. You can edit the rung any way you like.

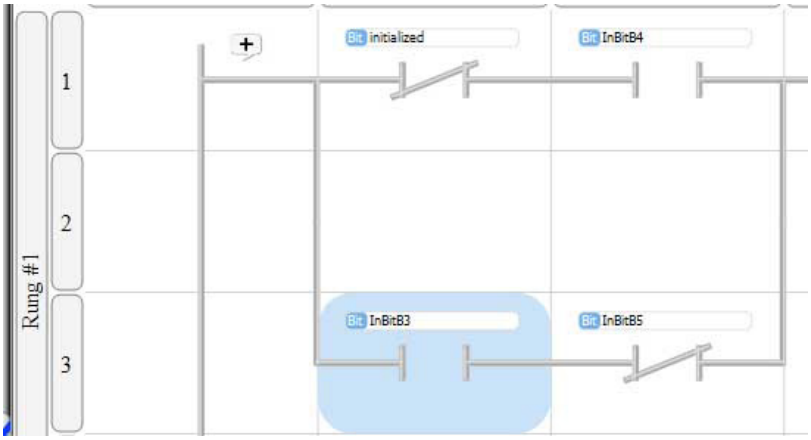
If you select the Rung Delete, the rung will simple be deleted.

Rung Insert Above
Rung Insert Below

These functions are only available in Ladder Logic. They only apply to rungs that have contacts in parallel. They are used for situations where you want to add some contact logic between existing logic.

If you select a contact, then select Rung Insert Above or Below, space for an additional row of contacts will be created, as shown in the example to the right and below.



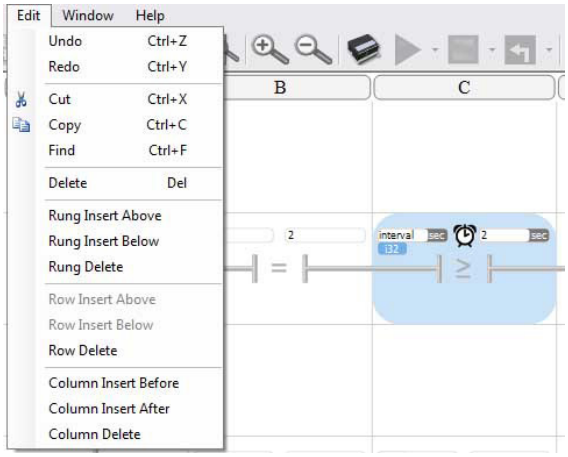


Column Insert Before
Column Insert After
Column Delete

These functions are only available in Ladder Logic.

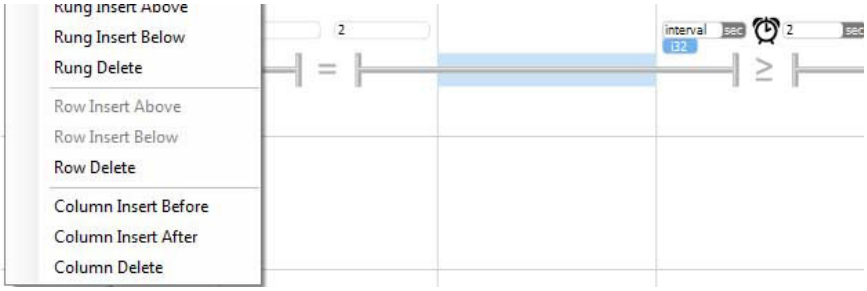
The Column InsertBefore or After can be used to insert a column, to make some space for placing another function block, either before or after a contact block. The Column Insert functions can only be used to create space in a rung from the rail to the last contact before the operation block at the end of the rung. They affect only the one rung.

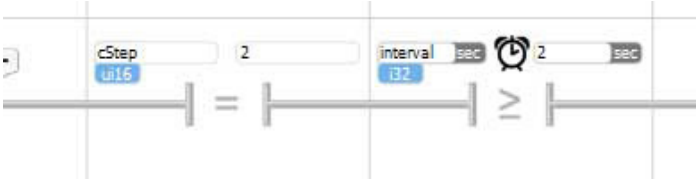
To Column Insert Before or After, first select a contact. Next, select the appropriate Column Insert Before or After from the Edit menu. The screen shots on the right show a contact selected and the result of a Column Insert Before.



A Column Delete can be used on any rung from the rail to the last contact. To perform a Column Delete, select an contact or connecting rail in the rung, up through its last contact, then select Column Delete from the Edit Menu.

The last two screen shots show a segment of connecting rail selected and the results of a Column Delete.

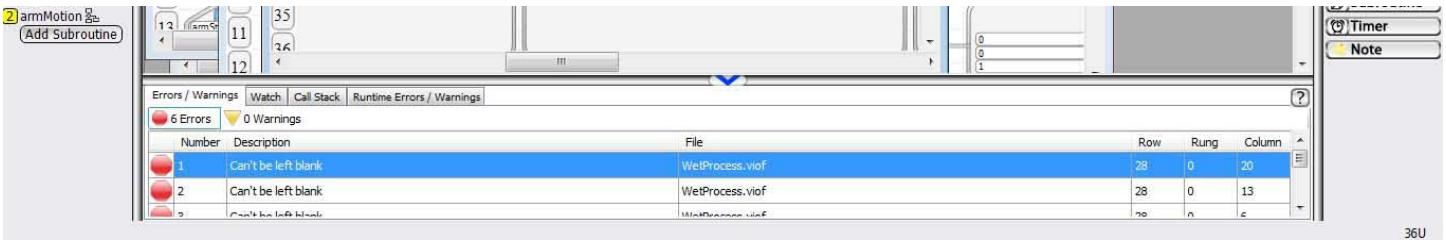




Status Bar

Along the bottom of the Edit screen is a tabbed bar, shown below. This is the Status Bar. The Status Bar provides information.

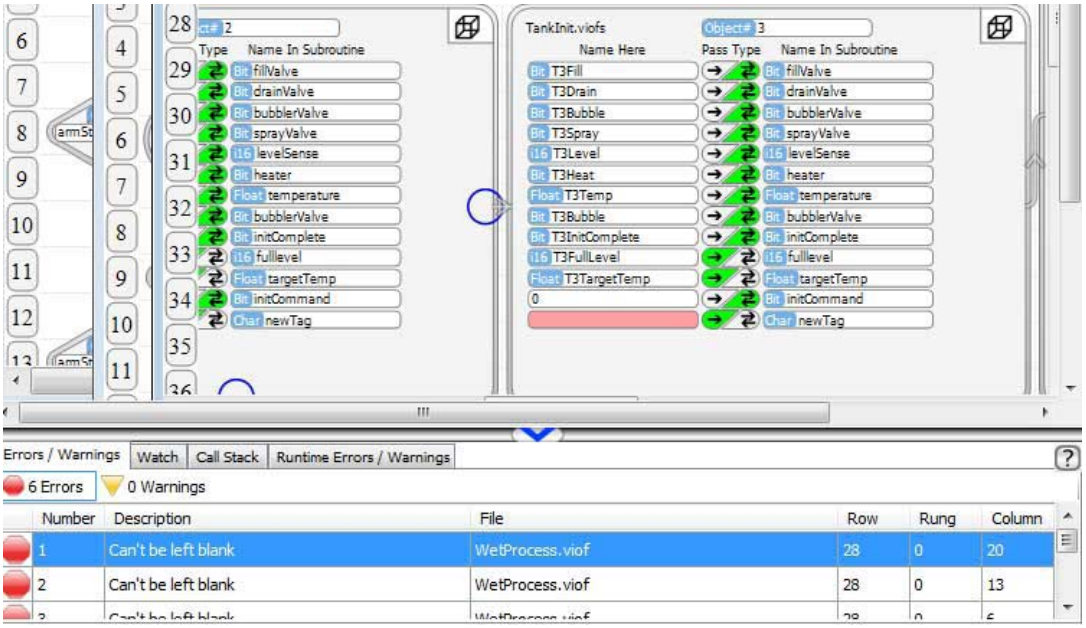
- Errors/Warning : This tab contains information that is only applicable while entering or editing a program. It is a list of Errors (things that you absolutely can't do) and Warnings (probably not a good idea) associated with the program that you are editing.
- Watch : Applicable only to debug mode. Used to watch selected tagname data while operating in debug mode.
- Call Stack : Applicable only to debug mode. Shows a "stack" of how the program got to where it is (who called who)
- Runtime Errors/Warnings : Applicable to debug mode. A list of errors or warnings that occured during run operation (example : divide by zero)



The three debug mode windows are discussed in more detail in the Debug chapter.

The Errors/Warnings window displays items that either must (errors) or should (warnings) be corrected while you are creating or editing a program.

An example of an error is shown below. Subroutine TankInit was edited after calls to it were placed in another program. An additional Input parameter (newTag) was added. This is fine. However, it creates an error in every block that calls TankInit. As the screen shot shows, the error is highlighted in red in the subroutine call, indicating that there is missing information. Also, in the Status Bar Error tab all of the errors are listed.



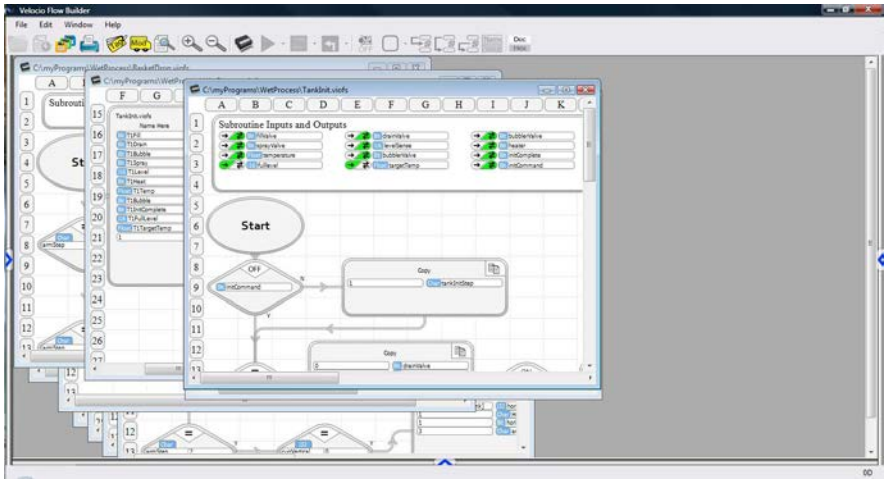
In this example, TankInit is called 6 different places. The Status Bar shows that there are 6 Errors. It also lists the Error Description, the File or subroutine where the error is located and the row and column (or rung for ladder) where the error is.

If you double click on an error, you will be brought, in the Edit window, to the location of the error.

Collapsing the Wings

The Setup and Project Files window on the left, the Toolbox on the right and the Status Bar on the bottom are all collapable wings. If you click on the arrow in the middle of the wing, the wing will collapse, giving more room for the Edit window. You can also select and hold the heavy bar on the inside edge of each and and move to adjust the size. The screen shot below shows all of the wings collapsed.

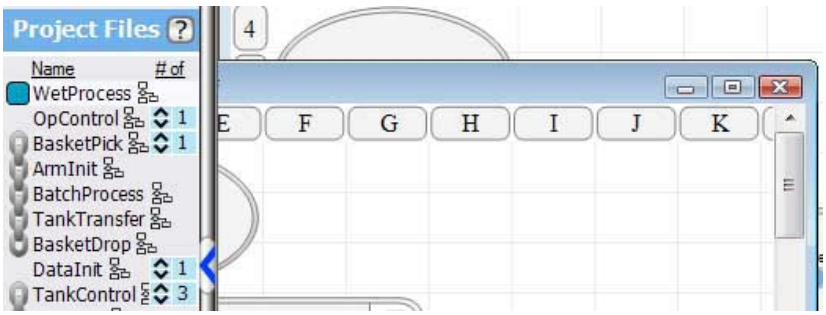
When a wing is collapsed, the middle arrow reverses direction. If you click on the arrow, the wing will reappear.



Closing a Project

To close a project, simply select the project's main program and close it. To close it, click on the red 'X' in the upper right hand corner.

You can close any other file and all that will happen is the file will be closed. You can re-open it by double clicking its name under Project Files. If you close the main program, the whole project will close.



4. Ladder Logic Programming

Velocio Builder's Ladder Logic progamming option provides the most powerful, easiest to use, best documenting ladder logic in the industry. Velocio has built on traditional ladder logic to provide all of the functionality and familiarity to allow long time PLC professionals to pick it right up and go. The additions of tagnames, subroutines, enhanced math, objects and other Velocio Ladder Logic features moves things to another level of productivity, capability, performance and maintainability.

The following is a general list of vBuilder Ladder Logic functionality.

- Typical Ladder Logic functionality
- Number formats including binary (bit), unsigned 8 bit integer (ui8), signed and unsigned 16 bit integer (i16 & ui16), signed 32 bit integer (i32), and floating point (Float)
- Tagnames for all variables
- True Object Subroutines
- Embedded Object Subroutines
- Distributed Processing and seamless intermodule data transfer
- Rung comments for documentation
- General Contacts
- Rising and Falling edge Contacts
- Numeric Comparison Contacts
- Coil outputs (Out, Set and Reset types)
- Calculator (math) operations, including arithmetic and boolean
- Copy (general copy, bit pack and bit unpack)
- Counter (general counter function)
- Drum Sequencer
- Filter
- Loops
- Motion in (high speed general pulse counting and quadrature pulse counting)
- Motion out (stepper motion control)
- PID
- Ramp
- Scale
- Shift/Rotate
- Statistics
- Timer

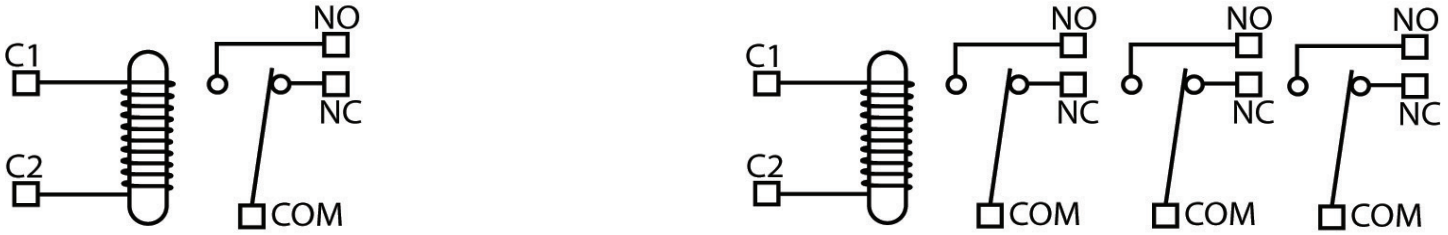
By using this functionality to compose a custom Ladder Logic program, any logical program can be created quickly, easily and graphically.

After a short review of the history of Ladder Logic to provide backgound information for those new to Ladder Logic, the individual vBuilder Ladder Logic program function blocks are detailed.

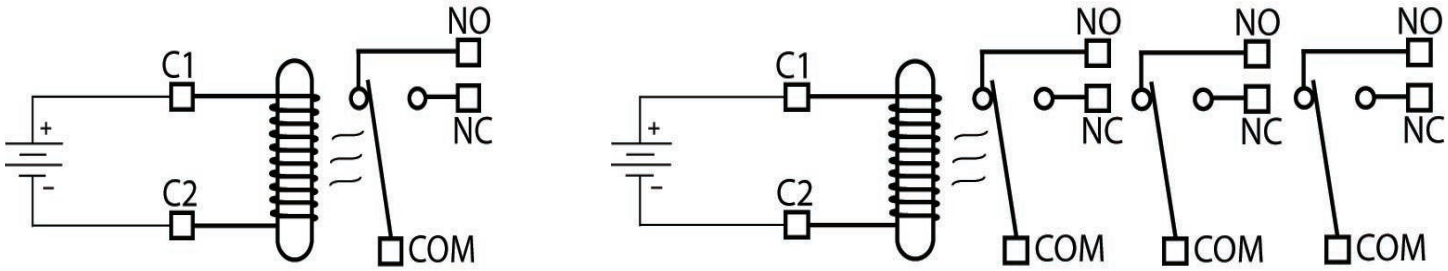
A Little History

Ladder Logic programming actually traces back to the days before PLCs. In 1835, American scientist Joseph Henry invented the relay to improve his version of the telegraph. The relay he invented is basically an electromagnetic switch. It consisted of coiled wires, which formed an electromagnet, and one or more metal contacts. When power is applied to the coil, a magnetic force is applied to the contacts and the contacts move. Typical relay contacts have a common terminal. Connected to the common is a piece of spring metal which creates a switchable electrical contact.

When the coil is not energized, the spring contact connects the common to a normally closed (NC) terminal, as shown on the left, below. A relay, with several contacts is shown in its de-energized state on the right.

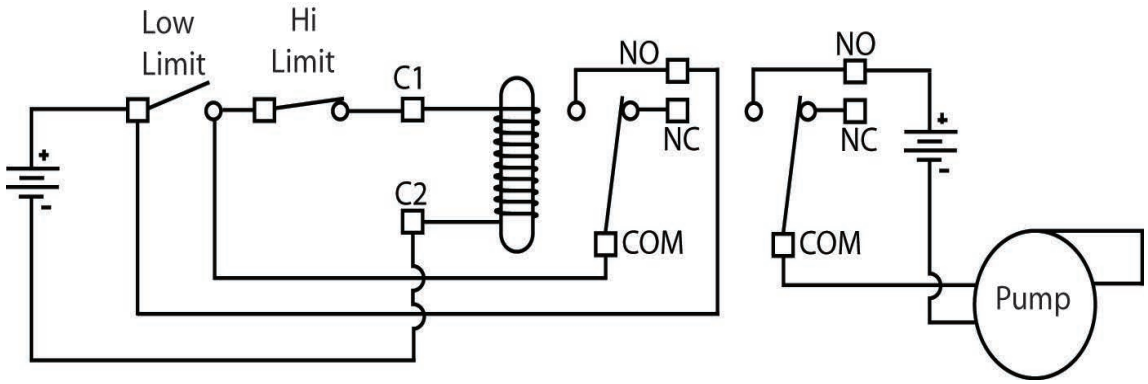


When power is applied to the coil, it creates a magnetic force, which pulls the contact to connect the common to the normally closed (NC) terminal, as shown on the left, below. The figure on the right, below, is an energized, multi-contact relay.

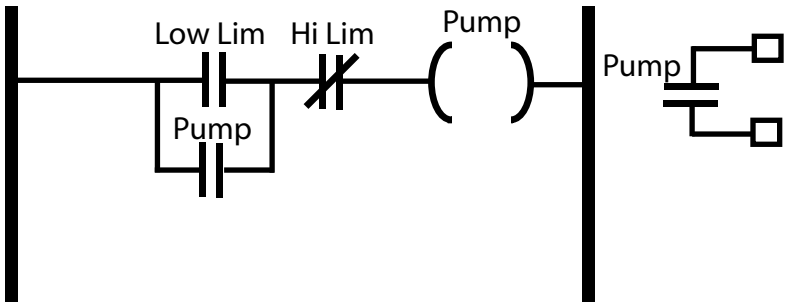


For over 100 years, automation designers used combinations of these simple relays to create controllers. By wiring up multiple relay coils and contacts, combined with switches, in the right pattern, any type of boolean logic operation can be created. The example, above, shows the control of a pump, feeding into a holding tank. There are two float switches in the tank. The low limit switch is normally open and closes when the water level falls below the "low level". The high limit switch is normally closed and opens when the water level rises above the "high limit". The controller has one relay, with two contacts. When the level drops to low, the relay will energize. The contact shown closest to the coil is in parallel with the low limit switch. It will keep the coil energized until the high level is reached. The other relay contact turns on the pump.

While that example, with one relay and two switches, is simple to understand, imagine what it looks like when the application requires tens to hundreds of relays and switches. The wiring diagram would quickly become incomprehensible. That is why an easier to follow form of logic drawing, ladder logic, was developed.



The illustration, below, shows the same example drawn in ladder logic. The two vertical lines on the ends are power rails. The ladder logic diagram shows one rung. The rung has the proper combination of contact connections, followed by the relay coil. This is the basis for ladder logic. The contacts form decision logic. The decision logic is always placed on the left hand side of the rung. The decision logic is followed by an action or execution block. In this case, the execution block is simply the energizing or de-energizing of the coil of a relay that controls the pump.



Using ladder logic diagrams, much more complex logic can be defined, much more clearly than with the typical wiring schematic shown on the previous page. Actual building of controllers still required the physical wiring of the required relay coils and contacts for 100 years or so. More complex programs were drawn in ladder form, by adding more rungs.

In 1968, Dick Morley invented the PLC. The original PLC was designed to be the electronic equivalent to a system of wired relays. Instead of wiring, the sequence of relay connections, or ladder logic, was downloaded from a programming device to the PLC. The PLC interpreted the downloaded ladder logic for execution. The first programming stations were crude, custom electronic devices. Within a few years, program entry began to be performed in graphical form, on personal computers and downloaded over RS232 communication ports to the PLC.

Over the next 40 years, more advanced features, such as math, timers, comparison contacts, drum sequencers, PID, counters and logical operations were added to various manufacturers' PLC ladder logic. It wasn't until after 2000 that floating point math became common place. A lot of incremental improvements to PLCs have enhanced capabilities and improved ease of use, over time. Howver, the pace has been slow. PLC progamming capabilities have fallen further and further behind the general computer and internet technologies on the market.

The first real breakthrough in PLC ladder logic programming, since the invention of the PLC in 1968, is the Velocio Builder (vBuilder) Ladder Logic. vBuilder introduces a number of major new capabilities, as well as providing significant enhancements to a number of others. Those features and capabilities are discussed, in detail, within this manual. With the breakthrough technology enhancements introduced by Velocio, PLC programming, in ladder logic or flow chart programming, has advanced to the technological level of the internet age. The graphical programming aspects for software development have actually moved PLC programming beyond the text based high level language alternatives found in general technology. Graphical progamming enables much higher productivity, higher confidence levels and enhanced documentation. Since graphical programming is so intuitive, it brings that capability for application by any logical person.

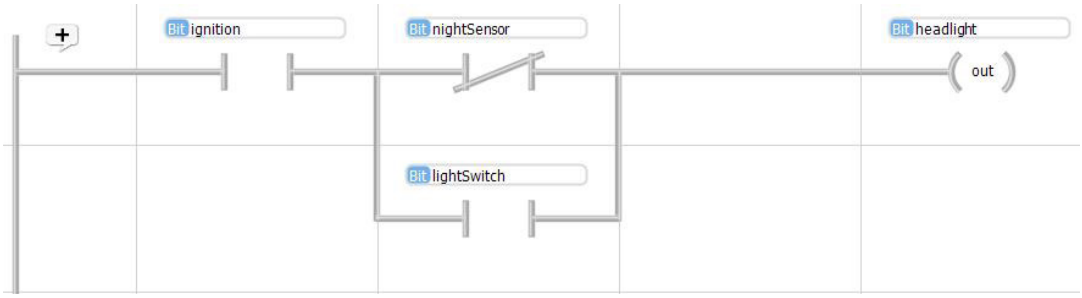
The following pages decribe the individual building block components of vBuilder Ladder Logic. Subsequent chapters decribe more advanced and powerful features, including subroutines, object oriented programming in graphical languages, embedded objects and other topics.

Contacts

In vBuilder Ladder, contacts create the decision logic.

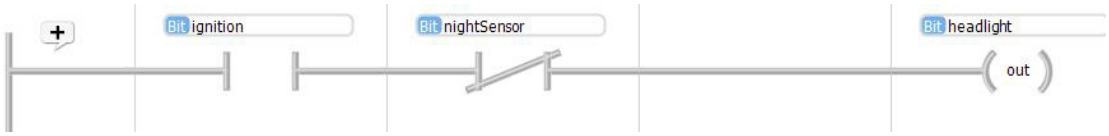
To understand the operation of contacts, within ladder logic, imagine that the rail (the vertical line on the left hand side) is electrified. Connected to the rail are a series of "rungs", containing contacts. Ultimately, at the end of each rung is an operation block. Whether or not the operation block executes is determined by whether an electrical connection, through the rung contacts, connects the operation block to the electified rail. If the electrical connection is made, the operation block executes.

Looking at the example below, the rung controls the control state of an output, tagnamed "headlight". The rung logic which determines whether it is turned on or off consists of three contacts. In order for the power to reach the "headlight" output, it must pass through the normally open "ignition" contact, then pass through either the normally closed "nightSensor" contact or the normally open "lightSwitch" contact. If the power is able to pass through to the "headlight" coil, the headlight will turn on. If not, it will turn off.



Contacts in Series

When contacts are placed in "series", or one after another in a horizontal line, they create an "AND" logic condition. In order for the rung to solve as true, or pass power from the rail to the operation block, each contact in series must evaluate to the true or closed contact state. In order for that to happen, each normally open contact must be "active" (bit value 1), and each normally closed contact must be "inactive" (bit value 0).



In the example, above, if the normally open ignition contact (or bit) is active (or 1) and the normally closed nightSensor contact is inactive (or 0), the headlight output coil will become active (or 1).

Contacts in Parallel

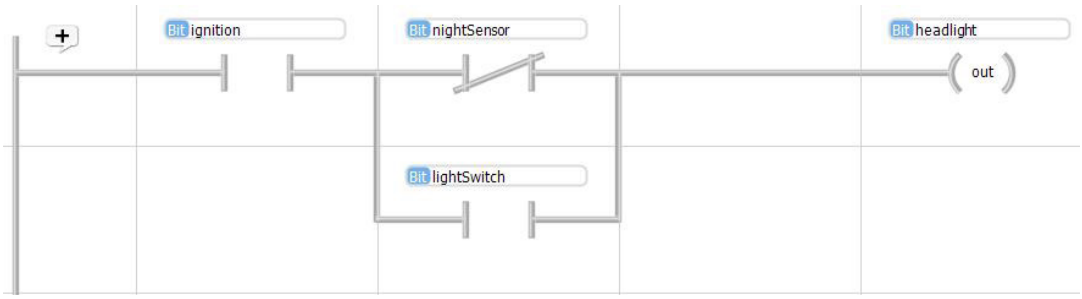
When contacts are placed in "parallel", or one below another with contact connections to the same input and output point, they create an "OR" logic condition. The rung will solve as true, or pass power from the rail to the operational block, if any one of the contact states is evaluated as true. That would happen if any normally open contact, in the parallel ladder,were to be active (bit value 1), or any normally closed contact were inactive (bit value 0).

In the rung above, coil "headlight" will be activated if either normally closed contact "nightSensor" is inactive (or 0) or normally open contact "lightSwitch" is active (or 1).

Contacts in Combination of Series and Parallel

In most applications, a combination of parallel and series contacts are placed to form a rung, like that shown on the previous page and again, below, for convenience. The logic is a combination of AND and OR logic, which translates to -

if "ignition" is active (1) AND if either "nightSensor" is inactive (0) OR lightSwitch is active (1), then "headlight" will become active (1).



Much more detailed information on the operation of individual contact types is contained in the pages which follow.

Normally Open and Normally Closed Contacts

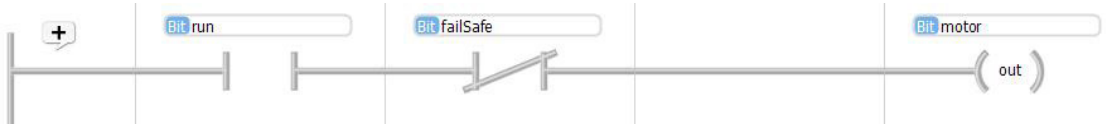
Normally open and normally closed contacts represent the state of a "Bit" type variable. A bit can be associated with digital inputs or outputs, or simply be an internal bit type tag. The "normal" state is the bit's inactive or state. For a digital input, the normal state is when the input is not energized. For an internal or output bit, normal is the state when the bit has the value of '0'. A bit value of '0' indicates that the coil or contact is not activated.

In ladder logic, bits are represented as coils and contacts. Every coil has associated contacts. The coils are representative of output operations on a tagnamed bit - activation or de-activation, while simple normally open or normally closed contacts are decisions made based on the state of a tagnamed bit. In ladder logic a bit (or coil) state of 1 is activated, a bit state of 0 is deactivated. The normally opened or normally closed contacts represent the state of a deactivated coil.

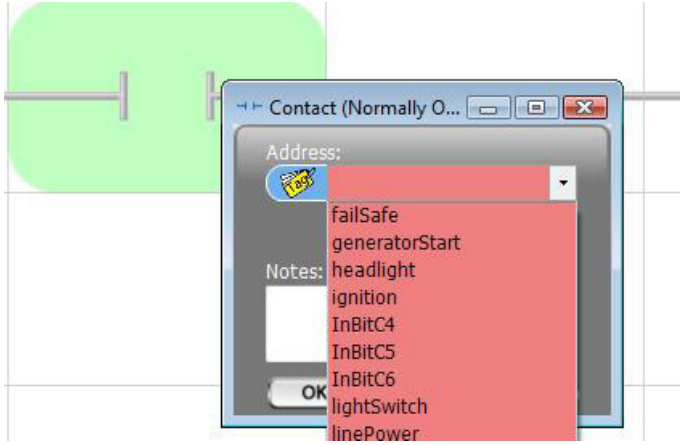
For the contacts in a ladder rung, logical connections occur through inactive (or bit value 0) normally closed contacts and active (or bit value 1) normally open contacts.

◊ Example :

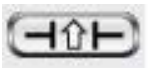
In this example, if the "run" button is active (or 1) and the "failSafe" sensor is inactive (or 0), the "motor" will become active (or 1).



When you place a Normally Open or Normally Closed contact in a rung, a dialog box will pop up. This box is used to select the tagnamed bit to associate with the contact. Select the down arrow to the right of the Address block to get a list of all tagnames available, then pick the one you want and click OK.



Rising Edge Contact



The symbol for a rising edge contact is shown on the left. A rising edge contact is only active (1) when the state of its coil transitions from de-energized to energized. That happens when the bit value of the of the tagnamed bit changes from '0' to '1'.

A rising edge contact remembers the state (or value of the bit) from the last time the rung was evaluated. If the bit was inactive (bit value 0) the last time, and is active (bit value 1) this time, the contact is closed, or active (1) for this one pass. It will not be determined closed again until the tagnamed bit is inactive (bit value 0) first.

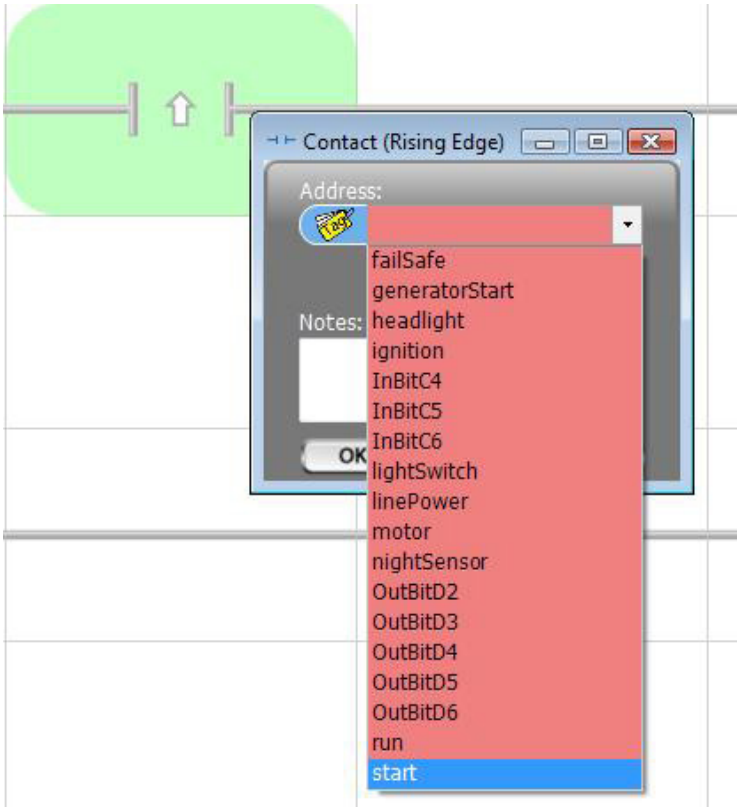
Rising and falling edge contacts are often used for cases when you want to perform an operation one time, when a transition is sensed.

Note : Edge contacts compare the current state of a bit to the state of the same bit the last time the rung was evaluated. If the edge contact is used in a subroutine, remember the logic evaluation is based on the last time the rung in the subroutine was executed for this object.

◊ Example

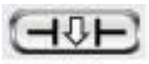
In this example, if the "start" button is pressed, a Copy block is used to set initial values for "targetTemp" and "warmTime"

When you place a Rising Edge contact in a rung, a dialog box will pop up. This box is used to select the tagnamed bit to associate with the contact. Select the down arrow to the right of the Address block to get a list of all tagnames available, then pick the one you want and click OK.



Falling Edge Contact

The symbol for a falling edge contact is shown on the left. A falling edge contact is only closed,or active (1), when the state of its bit value transitions from active to inactive. That happens when the bit value of the of the tag changes from '1' to '0'.



A falling edge contact remembers the state (or value of the bit) from the last time the rung was evaluated. If it was active (bit value 1) the last time, and is inactive (bit value 0) this time, the contact is closed, or active (1), for this one pass. It will not be determined closed again until it is active (bit value 1) again, first.

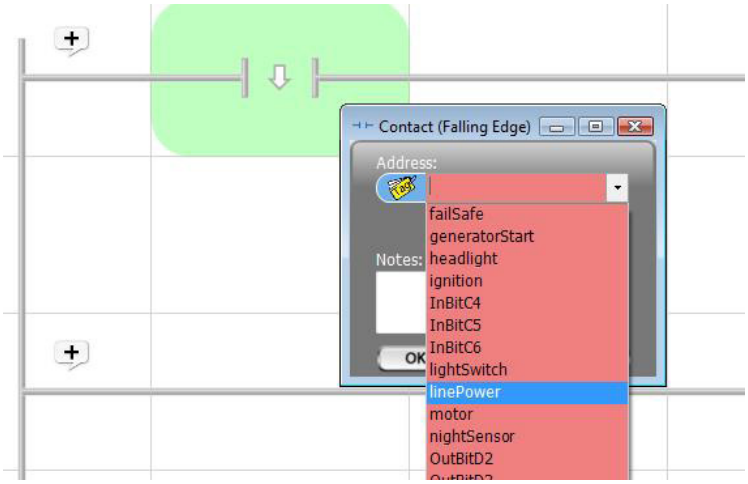
Rising and falling edge contacts are often used for cases when you want to perform an operation one time, when a transition is sensed.

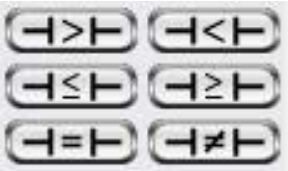
Note : Edge contacts compare the current state of a bit to the state of the same bit the last time the rung was evaluated. If the edge contact is used in a subroutine, remember the logic evaluation is based on the last time the subroutine rung was executed for this object.

◊ Example

In this example, if the "linePower" sensor transitions from active (1) to inactive (0), the "generatorStart" bit will be Set (to 1).

When you place a Falling Edge contact in a rung, a dialog box will pop up. This box is used to select the tagnamed bit to associate with the contact. Select the down arrow to the right of the Address block to get a list of all tagnames available, then pick the one you want and click OK.

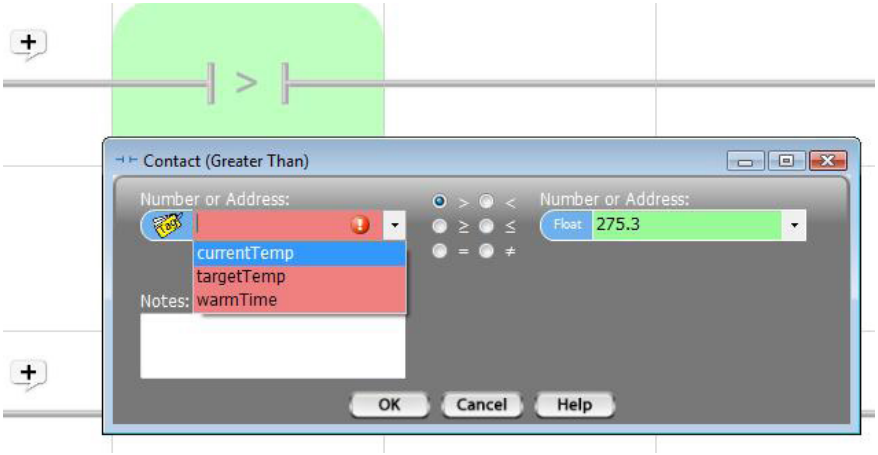




Numeric Comparison Contacts

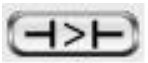
The contacts types shown on the left are numeric comparison contacts. The open/closed, or active/inactive, state of the contact is determine by the result of the defined numeric comparison. There are comparison contacts for greater than, less than, greater than or equal to, less than or equal to, equal to and not equal to. In each case, if the numeric comparison is true, the contact is closed. Otherwise, it is open.

When you place any numeric comparison contact in a rung, a dialog box will pop up. This box is used to define the comparison operation associated with the contact. For any comparison, either select the values to compare from tagnames available in the drop down boxes, or type in a numeric value. Make sure the comparison operation that you want is properly indicated by the radio button selection. If it is not, select the one you want. When you are done defining the comparison, click OK.



It is important to be careful when comparing floating point numbers to integer numbers. Floating point numbers are stored in IEEE format (which is what is used in almost all computer languages). This format incorporates 24 bits of numeric precision and an 8 bit exponent. They are very accurate, but not always exactly precise. A number such as 375.0 might actually be 375.00000001 or 374.9999999999. In such a case a comparison to integer 375 for equality would result in a not true result. When comparing floating point numbers to integers, greater than or less than comparisons are generally preferred.

Greater Than Comparison Contact

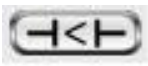


The programming icon for a Greater Than comparison is shown on the left. With a Greater Than comparison, a determination is made as to whether the first value is greater than the second value. If the first is greater than the second, the contact is considered closed. If it is not, the contact is considered open.

◊ Example

In this example, if the currentTemp is Greater Than the MaxTarget, the heater will be reset (turned off).

Less Than Comparison Contact

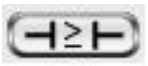


The programming icon for the Less Than comparison is shown on the left. With a Less Than comparison, a determination is made as to whether the first value is less than the second value. If the first is less than the second, the contact is considered closed. If it is not, the contact is considered open.

◊ Example

In this example, if the currentTemp is Less Than the MinTarget, the heater will be set (turned on).

Greater Than or Equal Comparison Contact

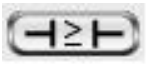


The programming icon for the Greater Than or Equal to comparison is shown on the left. With the Greater Than or Equal to comparison, a determination is made as to whether the first value is greater than or equal to the second value. If the first is greater than or equal to the second, the contact is considered closed. If it is not, the contact is considered open.

◊ Example

In this example, if the cycleCount is Greater Than or Equal to the batchTarget, the processStep will be set to 3.

Less Than or Equal to Comparison Contact

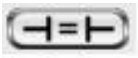


The programming icon for the Less Than or Equal to comparison is shown on the left. With a Less Than or Equal to comparison, a determination is made as to whether the first value is less than or equal to the second value. If the first is less than or equal to the second, the contact is considered closed. If it is not, the contact is considered open.

◊ Example

In this example, if the tankPressure is Less Than or Equal to the minPressure, the tankFill valve will be active.

Equal to Comparison Contact

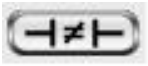


The programming icon for the Equal to comparison is shown on the left. With an Equal to comparison, a determination is made as to whether the first value is equal to the second value. If the two numbers are equal, the contact is considered closed. If not, the contact is considered open.

◊ Example

In this example, if cylceCount is Equal to batchTarget, the number 5 will be Copied into processStep.

Not Equal to Comparison Contact



The programming icon for the Not Equal to comparison is shown on the left. With a Not Equal to comparison, a determination is made as to whether the first value is not equal to the second value. If the two numbers are not equal, the contact is considered closed. If they are equal, it is considered open.

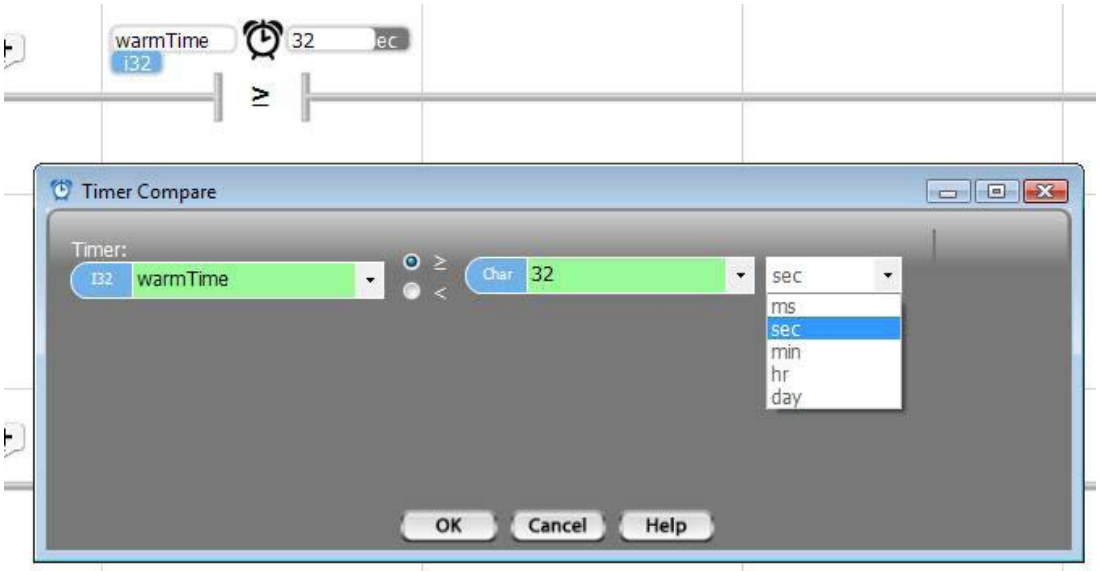
◊ Example

In this example, if statusCode is Not Equal to completeCode, processStep will be incremented.

Timer Comparison Contact

The icon for the Timer Comparison Contact is shown on the left. A timer comparison compares a timer value to a comparison value.

Timers use 32 bit integer (i32) numbers. An i32 used as a timer will keep time in 100ths of a second. It can be compared to values in milliseconds (1/1000s of a second), seconds, minutes, hours, or days. When you place a Timer Compare contact, a dialog box like that shown below, will pop up. In the dialog box, you must select the tagname of the timer in the left hand Timer box. The radio button selections to the right of the Timer box allow you to select whether to check if the Timer is greater than or equal to, or less than the value on the left. In the selection box on the left,



you can either enter a number, or select a tagnamed variable with the number you want to compare to. The last selection you must make is the time units the comparison value represents. You can select time units from the drop down selection list.

If the timer comparison is true, the contact is considered closed. If it is not true, the contact is open.

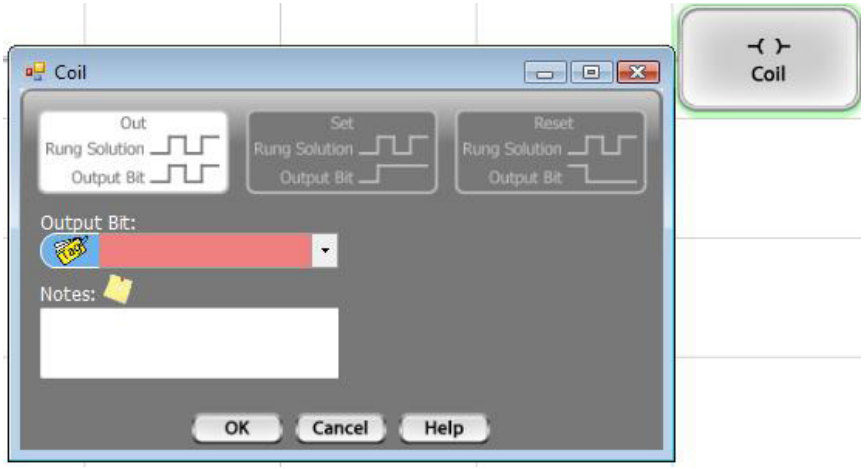
◊ Example

In this example, if warmTime is greater than 32 seconds, generatorStart will be set (to 1).

Coils

Coils are the simplest action blocks used in Ladder Logic. Coils can be used to turn a digital output on or off. They can also be used to set a bit type tagname to a '0' or a '1', for logical operations. There are three Coil types.

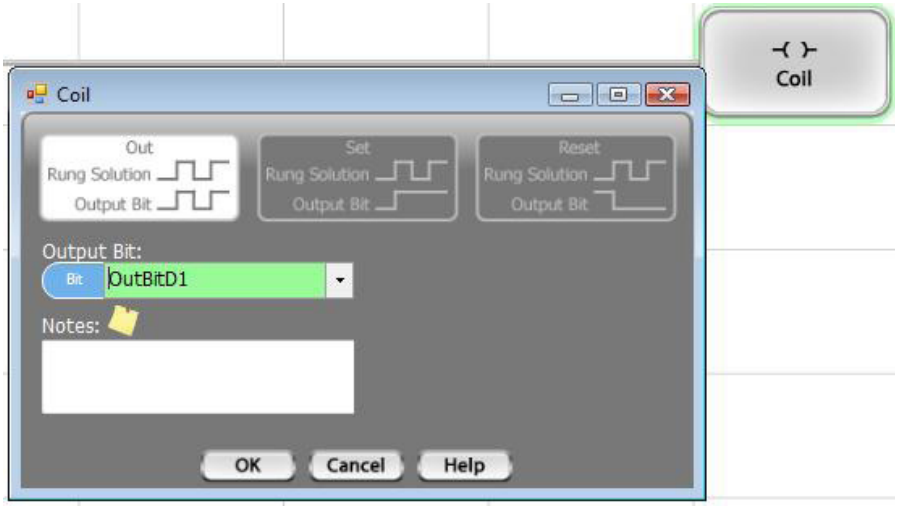
When you place a Coil, a dialog box, showing the three available options, as shown on the right, will pop up.



Out Coil

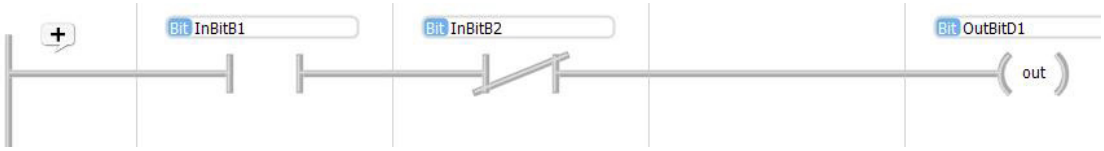
An Out Coil simply places the solution to the contact logic that precedes it into the tagnamed bit. In other words, if the contacts on the rung are in a state that creates a closed contact path all the way from the power rail to the Out Coil, the tagnamed bit will be set active (or 1). If it does not, the tagnamed bit will be set inactive (or 0).

When the dialog box pops up, select the "Out" option and select the tagname of the bit to use, as shown.



◊ Example

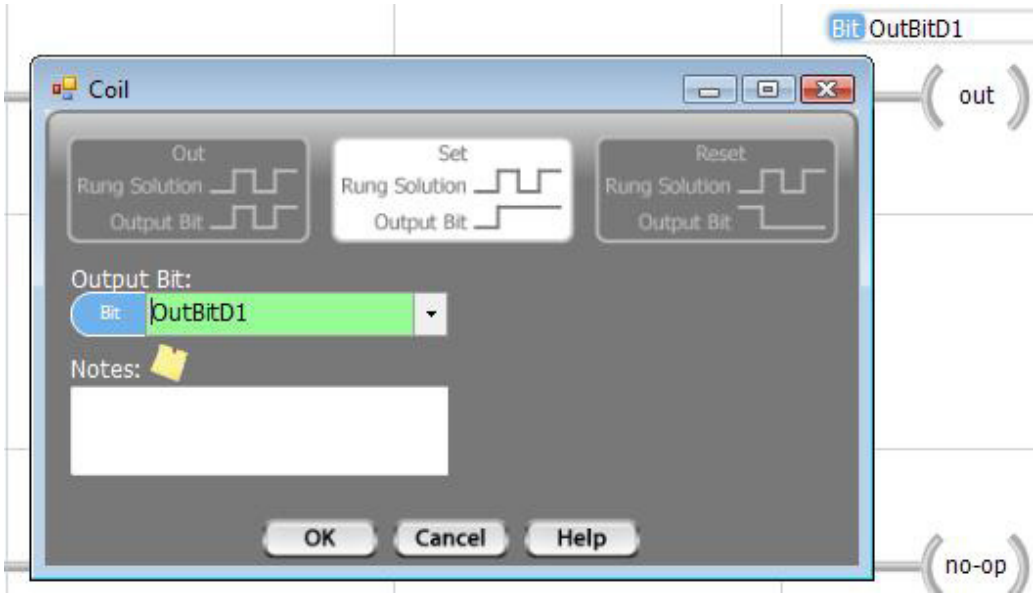
In the example shown below, if InBitB1 is active (1) and InBitB2 is inactive (0), OutBitD1 will be active (1). If either InBitB1 is inactive (0) or InBitB2 is active (1), OutBitD1 will be inactive (0).



Set Coil

A Set Coil will turn the tagnamed bit to active (1) any time the solution to the contact logic that precedes it solves to TRUE. In other words, if the contacts on the rung are in a state that creates a closed contact path all the way from the power rail to the Set Coil, the tagnamed bit will be set active (or 1). If it does not, the tagnamed bit will not be changed.

When the dialog box pops up, select the "Set" option and select the tagname of the bit to use, as shown.



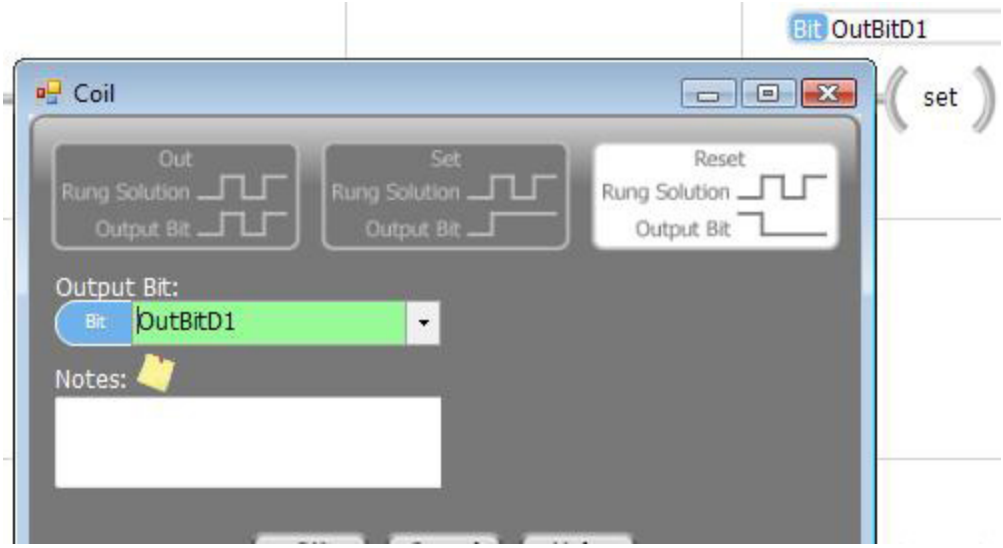
◊ Example

In the example shown below, if InBitB1 is active (1) and InBitB2 is inactive (0), OutBitD1 will be set active (1). If either InBitB1 is inactive (0) or InBitB2 is active (1), nothing will happen.

Reset Coil

A Reset Coil will turn the tagnamed bit to inactive (0) any time the solution to the contact logic that precedes it solves to TRUE. In other words, if the contacts on the rung are in a state that creates a closed contact path all the way from the power rail to the Reset Coil, the tagnamed bit will be set inactive (or 0). If it does not, the tagnamed bit will not be changed.

When the dialog box pops up, select the "Reset" option and select the tagname of the bit to use, as shown.



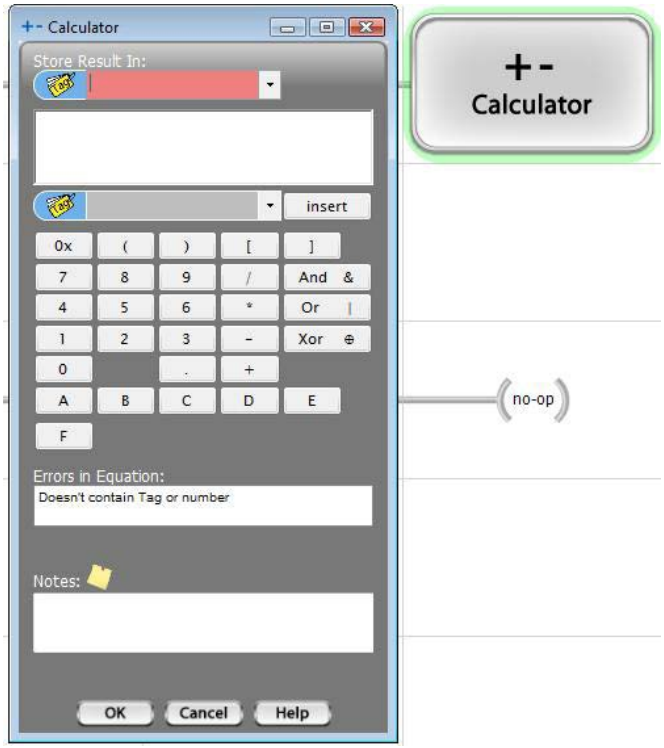
◊ Example

In the example shown below, if InBitB1 is active (1) and InBitB2 is inactive (0), OutBitD1 will be reset inactive (0). If either InBitB1 is inactive (0) or InBitB2 is active (1), nothing will happen.

Calculator

The calculator is the math function of vBuilder. The calculator icon is shown on the left. With a calculator program block you can perform any type of arithmetic or boolean operation. Equations can range from simple to bracketed complex operations.

When you place a calculator block, a dialog box will pop up, as shown on the right.



With the dialog box, first choose the tagnamed variable to use to store the result, from the drop down list at the top of the dialog box. Next, enter your equation. You can pick tagnamed variables using the tag picker below the equation window, then press insert. Select operators and brackets to place in the equation as needed. You can create as complex of an equation as you want.

Once you've entered the equation that you want, click OK. The block will show up in the rung like the illustration to the right.

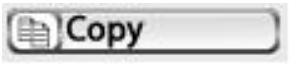
Operators :

+ - * / : Math operators for addition, subtraction, multiplication and division
And &, Or |, Xor + : Boolean And, Or and Exclusive Or operators
( ) : Segmentation brackets
[] : Braces used to contain array index
0x : Hexadecimal number
A, B, C, D, E and F : Hexadecimal 10, 11, 12, 13, 14, and 15

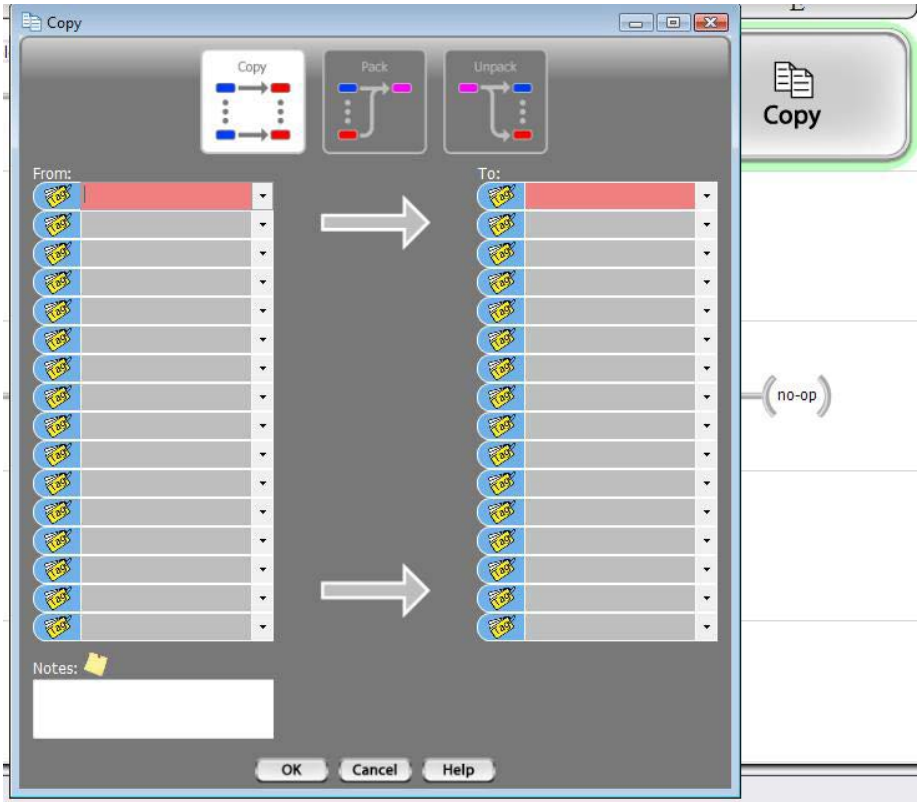
◊ Example

In the example below, if calcBatchWt is active (1), the targetWt will be calculated according to the formula shown.

Copy



The icon for Copy operations is shown on the left. There are actually three different Copy function blocks available : Copy Value, Copy Pack and Copy Unpack.



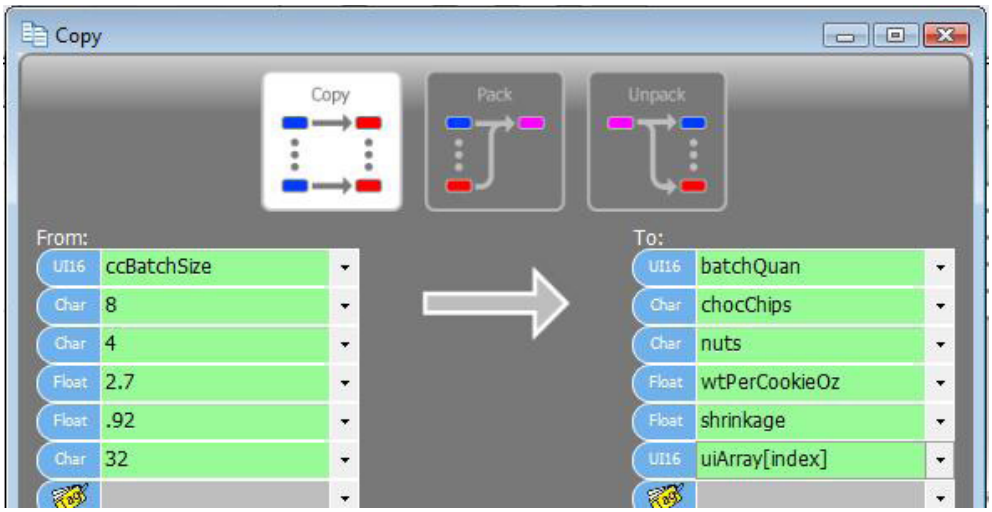
When you place a Copy block in a ladder rung, a dialog box, as shown on the right, will pop up.

The selections across the top of the dialog box are illustrated selection buttons that enable the selection of the appropriate type of copy. As the selection buttons show, the general Copy allows you to copy up to 16 tagnamed variables or constant values to a like number of other selected tagnamed variables. The Copy Pack provides a function for copying a list of bits to a ui16 integer. The Copy Unpack does the opposite - copy individual bits from a ui16 integer to bit tagnamed variables.

When you select the different Copy types, the copy details, below the selection buttons, change accordingly.

Copy Value

The Copy Value function provides the capability to copy up to 16 values or tagnamed variables to a like number of tagnamed variables. Select the "Copy" option in the Copy dialog box and simply select a number or a tagnamed variable to copy from in each box on the left and a corresponding tagnamed variable to copy to in the associated box on the left.



The Copy Value function will convert data from one type to another, if the data types don't match.

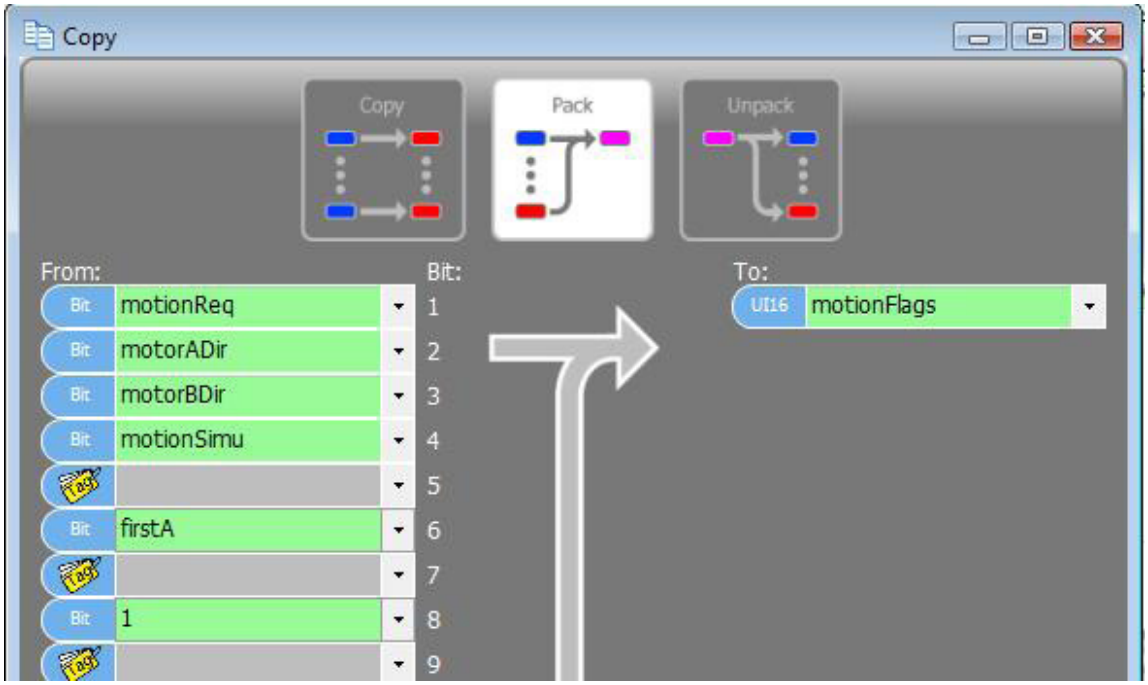
The illustration, above, includes a copy into an indexed variable. When you select an indexed variable, it will appear with the index braces []. You must select inside the braces and type the index number, or type the tagname of a ui16 variable. The case of typing a variable tagname is shown above.

◊ Example

In the example shown below, if initRecipe is active (1), the list of six variables and constants will be copied to the corresponding list of six tagnamed variables.

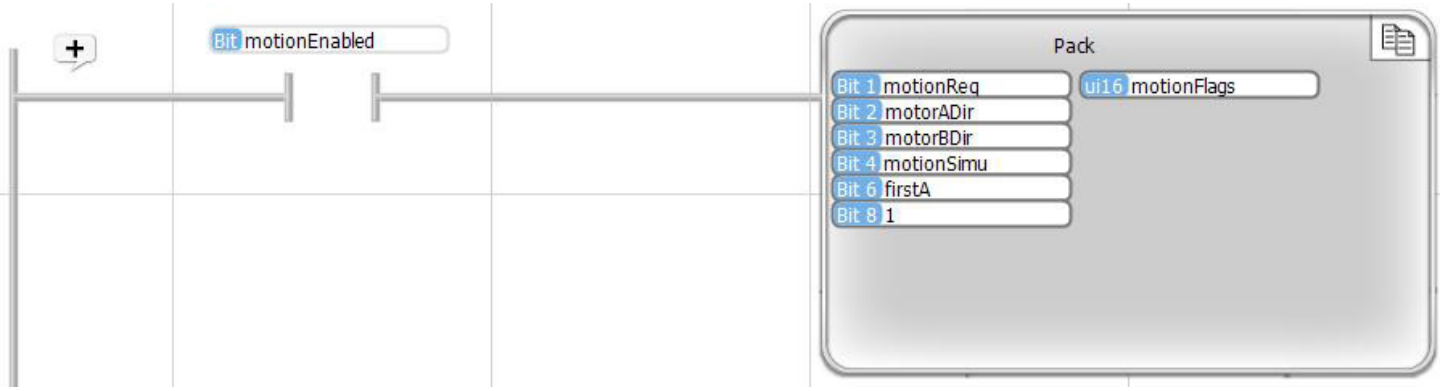
Copy Pack

The Copy Pack function provides the capability to copy up to 16 tagnamed or constant bits to a ui16 variable. Select the "Pack" option in the Copy dialog box and simply select a 0 or 1 or a tagnamed bit to copy to the bit position in each box on the left and a corresponding tagnamed ui16 variable to copy to in the associated box on the left. Bits of the 'To' variable in positions not selected will remain unchanged. This function can be used to 'mask in' selected bit locations within the ui16 variable.



◊ Example

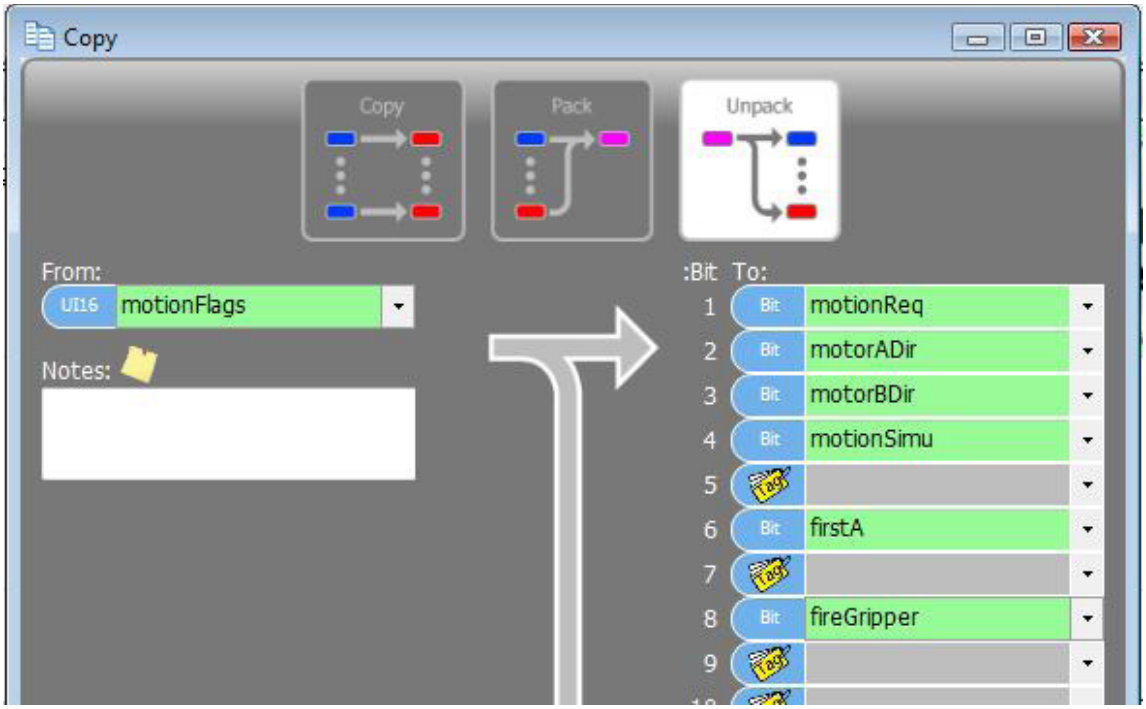
In the example shown below, if motionEnabled is active (1), the bits identified on the left of the Copy Pack block will be copied to the bit positions identified next to the bit tag name or value in the ui16 tagnamed variable listed on the right.



Copy Unpack

The Copy Unpack function provides the capability to extract individual bits from a ui16 variable. Select the "Unpack" option in the Copy dialog box. Unpack extracts bit data from particular bit positions in the From ui16 to the tagnamed bits assigned to each bit position. In the dialog box, the From tag is a ui16 tagnamed variable that you select. Bits of the 'To' variables are extracted from the bit positions identified in the Bit To list.

The source ui16 (From variable) is not changed. There is no requirement that all bits be unpacked. With a Copy Unpack, selected bits can be extracted at will.



◊ Example

In the example shown below, if getMotionFlags is active (1), the bits identified on the right of the Copy Unpack block will be extracted from the bit positions identified next to the bit tag names from the ui16 variable motionFlags.

Counter

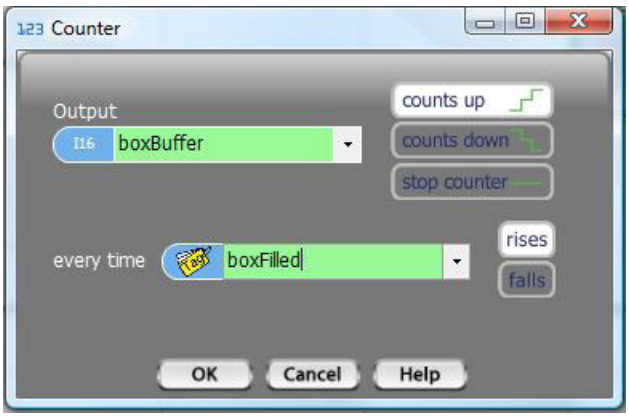
There are both software and high speed hardware counter functions available in vBuilder. The general Counter is a software counter and is described in the following paragraphs. High Speed pulse counter and quadrature counter functions are MotionIn functions and are described in the MotionIn function description.

Counters are background tasks. Once a counter is set up through a Counter block, it will continue to operate, behind the scenes, until it is stopped through a Stop Counter block execution. The counters work in the background, updating once, just prior to the execution of each program logic pass, ***based on the state of the selected bit at that time***. Once you have started an Up or Down Counter, your program logic does not need to execute it again on every program pass.

Note : A counter operation is defined by its Output tag name. Therefore there can be only one count operation active for a given Output tag name.

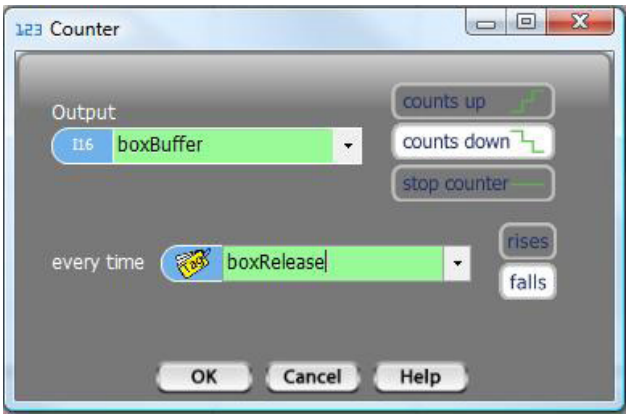
Up Counter

The Up Counter increments a count by one, each time a tagnamed bit transitions (checked just prior to each logic pass) in the direction (either rises or falls) that is selected. In other words, if you want to place an Up Counter block that increments each time a tagnamed bit changes from 0 to 1, select Counter, then select "counts up" and "rises", as well as the Output (the tagnamed variable that keeps the count) and the tag name of the bit that is checked "every time" for the rising transition.



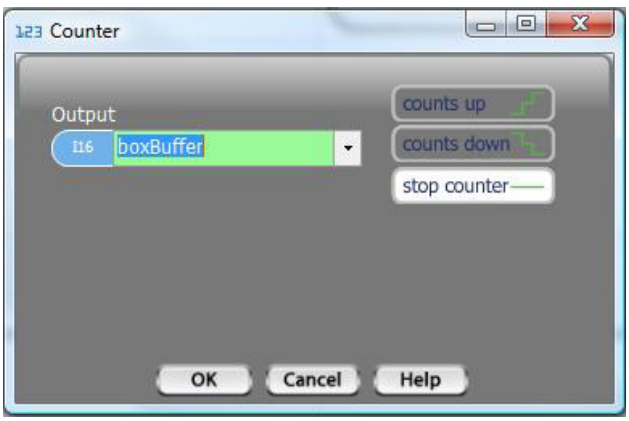
Down Counter

The Down Counter decrements a count by one, each time a tagnamed bit transitions (checked just prior to each logic pass) in the direction (either rises or falls) that is selected. In other words, if you want to place a Down Counter block that decrements each time a tagnamed bit changes from 1 to 0, select Counter, then select "counts down" and "falls", as well as the Output (the tagnamed variable that keeps the count) and the tag name of the bit that is checked "every time" for the falling transition.



Counter Stop

The Counter Stop simply stops the selected counter from counting. The count value will not change. To place a Counter Stop block, select "Counter", then "stop counter" and select the tagname of the counter that you want to stop. Then select 'OK".



◊ Counter Example

The example shown below, shows and application with two counters associated with a breakfast cereal fill line. The fill line has a conveyor that brings empty boxes to a fill chute, where each box is filled, then to a buffer area. In the buffer area, boxes are buffered, while being picked up individually and placed in shipping cartons. This program snippet shown the boxCount incrementing, each time a box is filled. A second counter, boxRemaining is the count down of the remaining box production scheduled for this shift. boxRemaining is decremented each time a box is picked up and placed in a carton. Both of these counters are started on processStep 4. processStep is immediately incremented to 5. In processStep 5, the conveyorRun tag bit is monitored. When conveyorRun becomes inactive (0), both counters will be stopped and processStep will be set to 6 .

Drum Sequencer

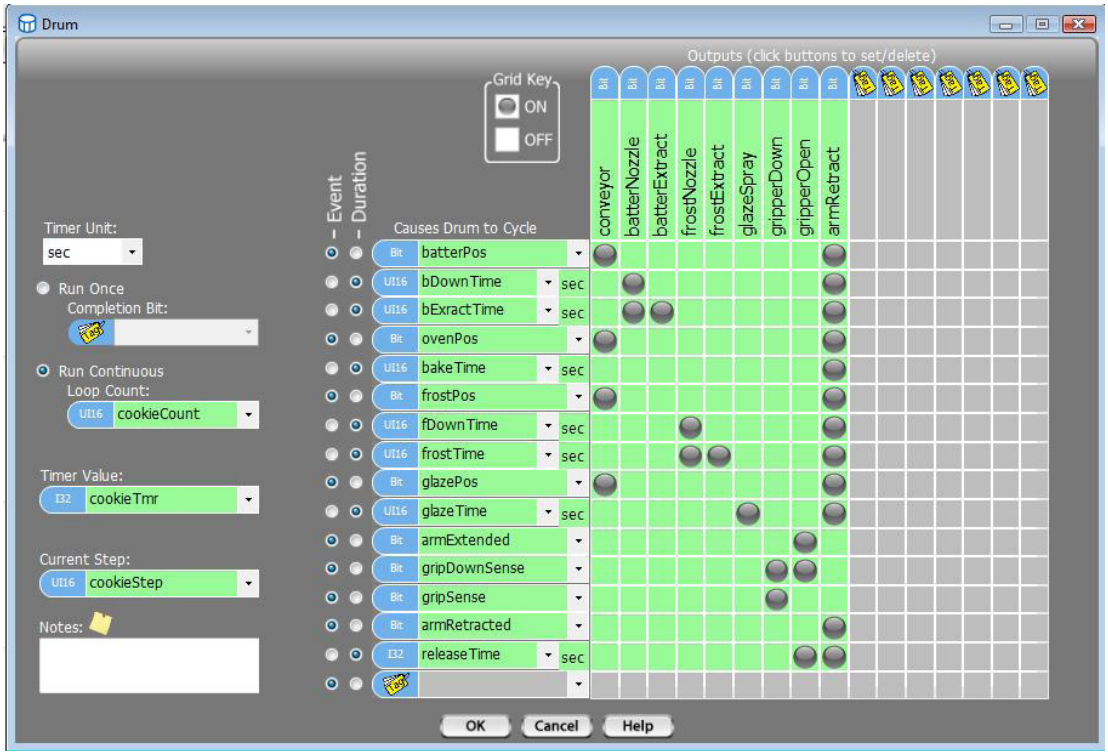
The program icon for the Drum function is shown on the left.

The Drum function is one of the most intricate and powerful functions in vBuilder Ladder Logic. The Drum function is basically a predefined sequencer. It is only available in Ladder.

The drum sequencer operation is like that of the old player pianos that were popular from their invention in 1876 until about the start of the Great Depression. Player pianos were pianos designed to play automatically. They had a rotating drum, where a cylinder with the song music could be placed. The song cylinders had a pattern of holes that could be sensed by mechanical sensors in the piano. As the cylinder rotated, the each hole sensed would cause a particular note to be struck. The cylinder rotated at a constant rate, allowing the hole pattern to completely define the tune.

The Drum function is like that of the player piano. It controls a pattern of bit outputs that are sequenced through. The sequence isn't controlled by a constant speed wheel. The progression of output pattern steps are sequenced instead, by either programmed times, or events.

The best way to get a sense of a Drum Sequencer operation is to take a look at how the dialog box is used. An example is shown below. The first thing to notice is the grid. Across the top of the grid are up to 16 bits that are either activated (1) or deactivated (0), based on the sequence step that is being executed. Listed along the left side are the events or times that cause the sequencer to move to the next step. The pattern of dots are the bits that are set active during each step.



Take a look at the first few sequence steps.

- Step 1 : The conveyor and the armRetract will be active until batterPos is sensed. When that happens, the program moves to sequence step 2.
- Step 2 : The batterNozzle and armRetract will be active for bDownTime. At the completion of bDownTime, the program moves to sequence step 3.
- Step 3 : The batterNozzle, batterExtract and armRetract will be active for bExtractTime. At the completion of bExtractTime, the program moves to step 4 ;
- Step 4 : The conveyor and armRetract will be active until ovenPos is sensed. When that happens, the program moves to sequence step 5.

There are some settings and entries on the right hand side of the dialog box that are important too. The selection at the top is the time increment to use for timed events. This example shows seconds. Other available options from the drop down selection list are millisecond, minutes, hours and days.

The next selection option is whether you want to run the sequece once or continuously. If the selection is to run through once, you must select a bit tagname to use to indicate sequence completion. If the choice is continous, you must select a ui16 integer to hold the loop count.

If you have selected Duration for any step, you must select an i32 variable tagname to be used to hold the time value.

The last selection is for a tagnamed ui16 variable to hold the Current step number.

When building the sequence, for each step, first choose either Event or duration. If an event is chosen, the tagname required for the associated 'Causes Drum to Cycle' field is a bit type. If Duration is chosen, the 'Causes Drum to Cycle' field requires an integer to define the duration time. Any integer type is OK for this entry.

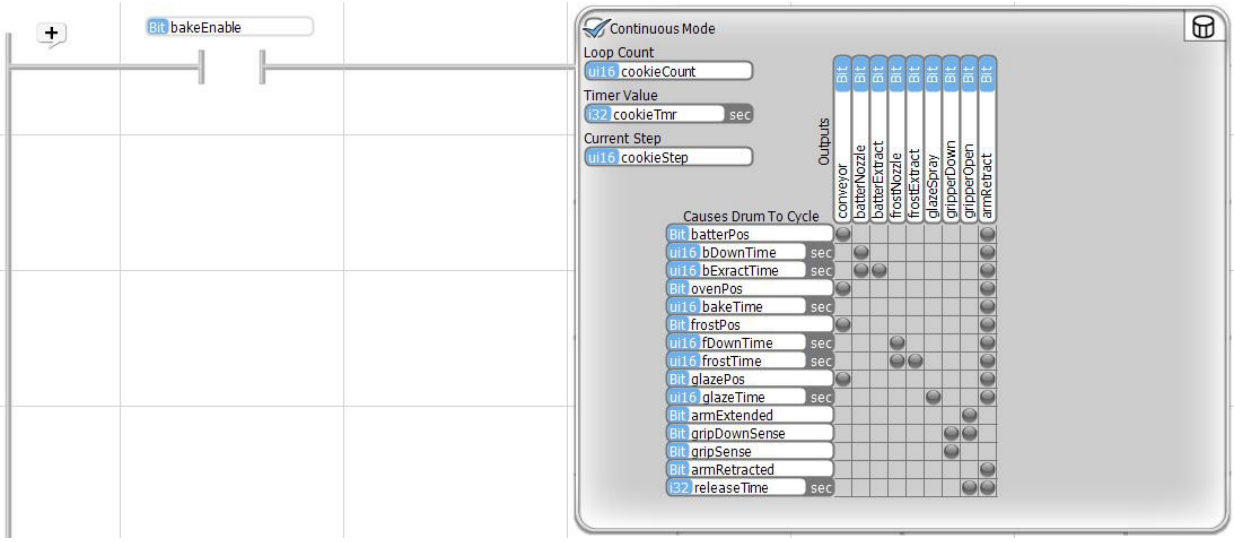
◊ Example

The example we've chosen is a Drum Sequencer for a one cookie at a time baking system.

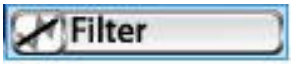
Yeah, we know that such a system doesn't make any real world sense, but we had to come up with an example on short notice. Humor us on this one.

A cookie pan is on a conveyor that takes it to batter extraction station, then to the oven, a frosting application station, glazing station (overkill), then an arm to pick it up, retract and drop it in a box. If you take a good look at the sequence, you should see that its all there - a one block function control for the whole cookie line.

In this example, all of the activation bits control digital outputs. In a Drum sequencer, they can actually be any bit. Other common uses are bits that enable particular sections of program code, or particular subroutines.



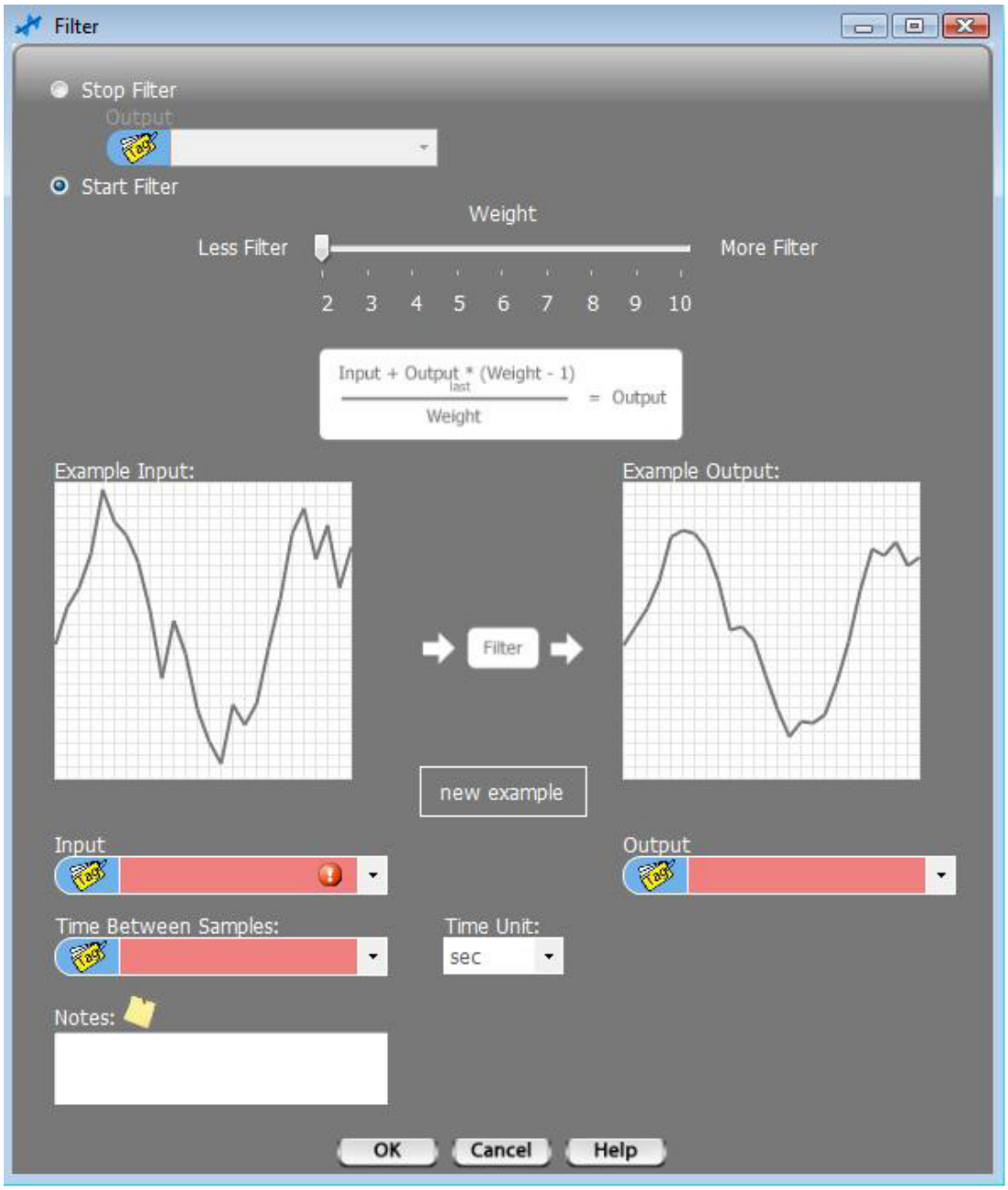
Filter



The program icon for the Filter block is shown on the left. The Filter function is a relatively simple function. It can be used in instances where you are more interested in an average value over a period of time than any particular instantaneous value.

The Filter function is a background task function. You simply tell it what to filter at what weight and time interval and the Filtering will happen behind the scene automatically. All you have to do, once you've set up the Filter is use the filtered result value whenever you need it.

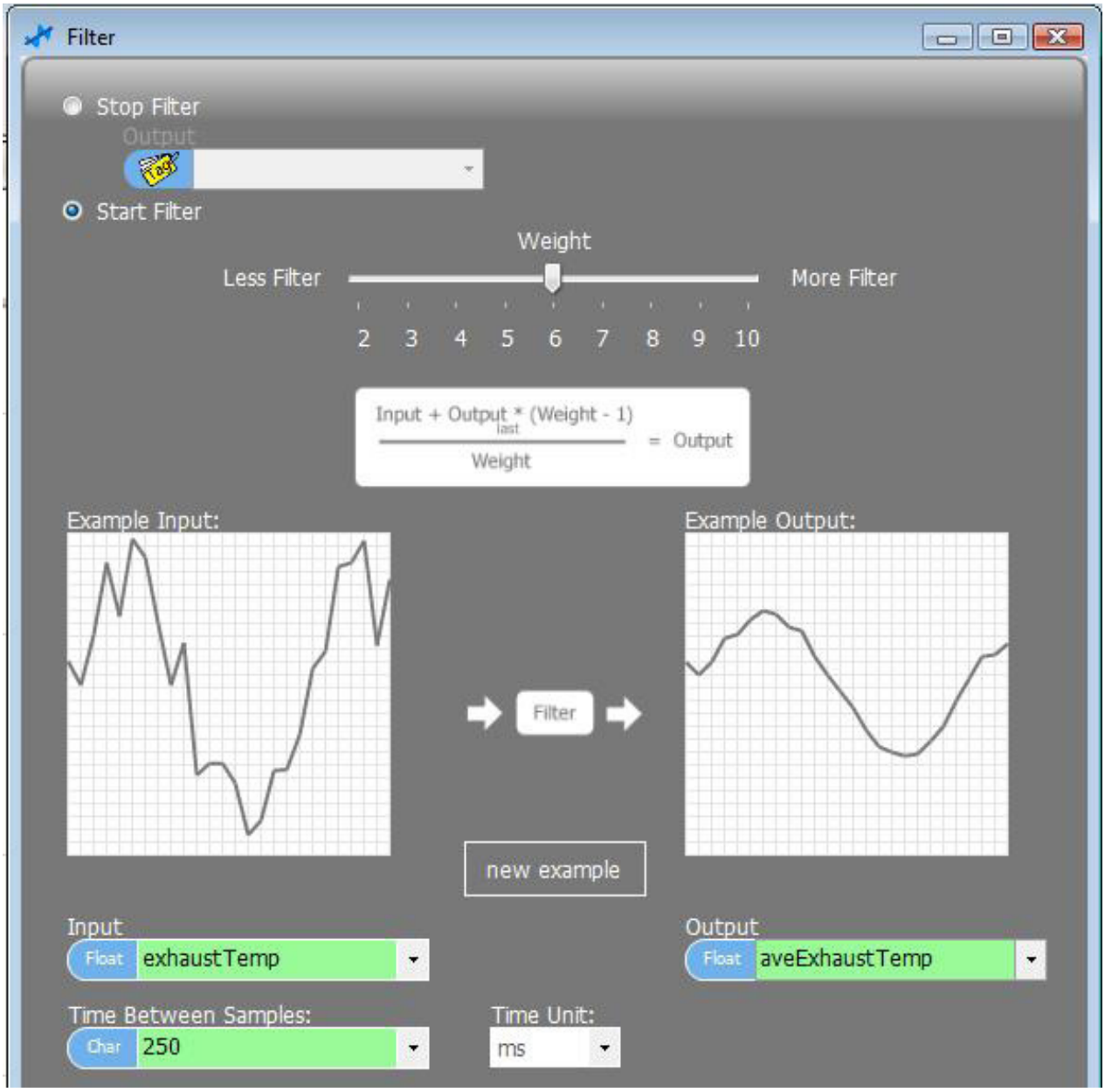
When you place a Filter block a dialog box will pop up. The box look like this :



To get a sense of what Filter does, right click and hold the slider shown next to "Less Filter" and slide it across. The live illustration below the slider shows that with an Example input shown on the left, the resulting output values will be like that shown on the right. Everytime you click on the New Example button, you'll get a new example illustration.

To Start a Filter, select Start Filter, select the tagname of the variable that you want to filter, as Input, select the tagname of the variable where you want the filtered result to be placed, as Output, select a sample rate (either a constant value, or a tagnamed integer variable) and the time units, and select the weight by sliding the Weight bar. Then click OK.

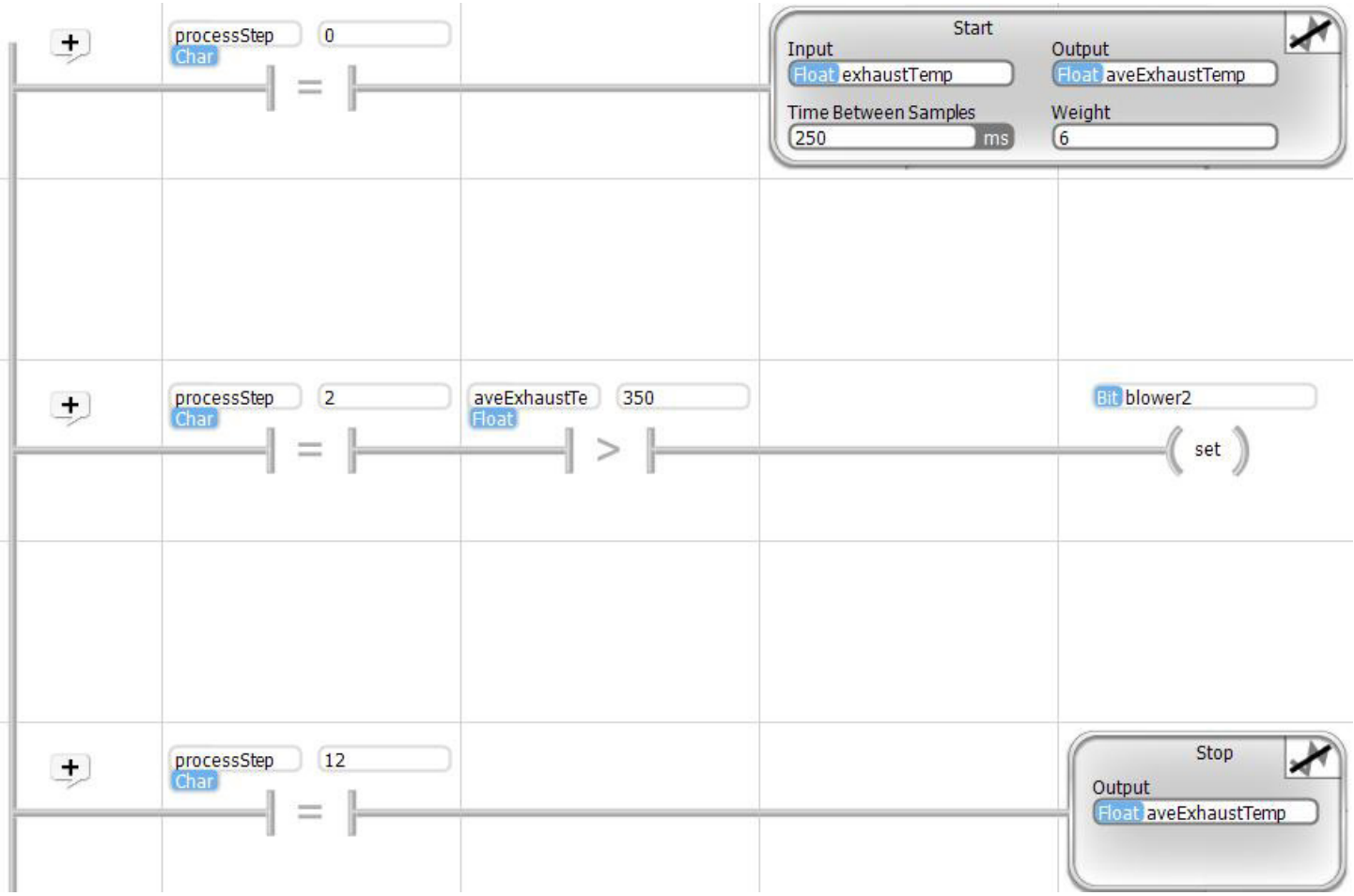
Once you've started a Filter, it will continue to operate in the background until you stop it.



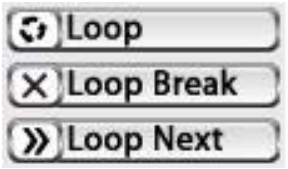
To Stop a Filter, place a Filter block and select Stop Filter and select the filtered output value's tag name. Click OK.

◊ Example

The example, shown below, shows a Filter started in processStep 0 (probably initialization), the filtered value used during processStep 2, and stopped in processStep 12 (probably shut down).



Loop Functions



Icons for the Loop functions are shown on the left.

Program looping is available for Ladder Logic and is an effective tool in certain instances.

Never loop, looking for a change in an input. Inputs are updated between logic passes. Its not going to change. Writing your program as a state machine (see chapter on state machine programming) is a better way to handle waiting for an input change.

Loop

The Loop program icon is shown on the right. The Loop block defines the start of a program block that will loop a defined number of times. For every Loop block, you need to place a corresponding Loop Next block after the last rung of the looped section of your program.

When you place a Loop block, a dialog box, like that shown on the right will appear. Type in any number between 1 and 10000, for the number of loops to execute.

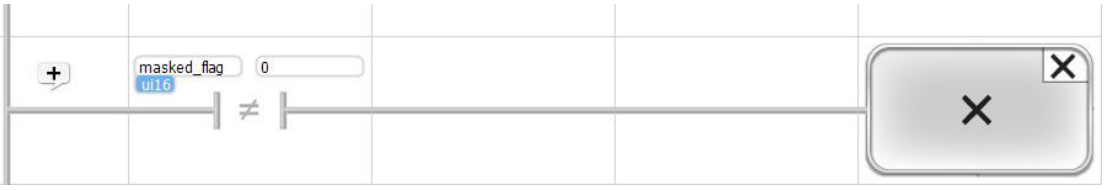
All of the rungs between the Loop and the Loop Next will execute the number of times specific as Number of Loops, unless the program executes a Loop Break.

Once the OK is pressed, the Loop block looks like the image below.

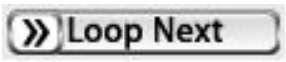
Loop Break

A Loop Break causes the Ladder Logic program to "break" from the loop and continue with the rung after the loop. This is one logic block that requires no additional definition, and therefore has no dialog box.

The image, below, shows a Loop Break placed in a rung. In this case, if masked_flag is Not Equal to 0, the Loop Break will cause the program to break out of the loop.



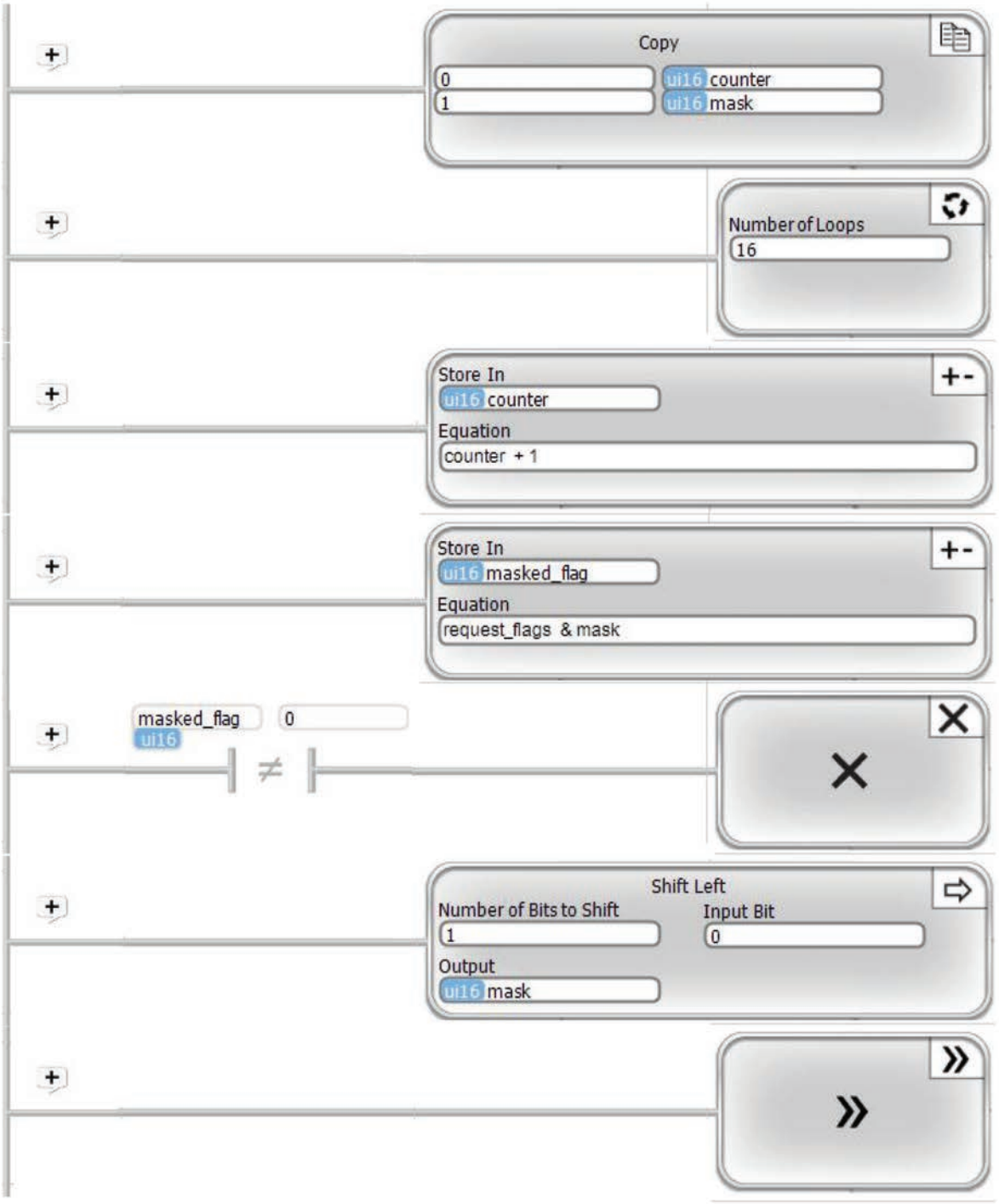
Loop Next



The program icon for Loop Next is shown on the right. A Loop Next block must be placed at the end of a Loop. It requires no further definition and therefore has no dialog box. A loop will execute all rungs between a Loop and a Loop Next, the number of iterations defined in the Loop block - unless a Loop Break is executed.

◊ Example

The example, below, illustrates the use of a Loop. In this example, a variable named request_flags contains a series of individual bits that are requests for different operations. The highest priority operations are requested through the least significant bits. This loop is used to determine whether there is an operation requested, and if so, which one. If a request bit is set, the program will exit the Loop with the number of the request in "counter". If no request bits are set, the Loop will complete all passes and the counter will have the value 16.

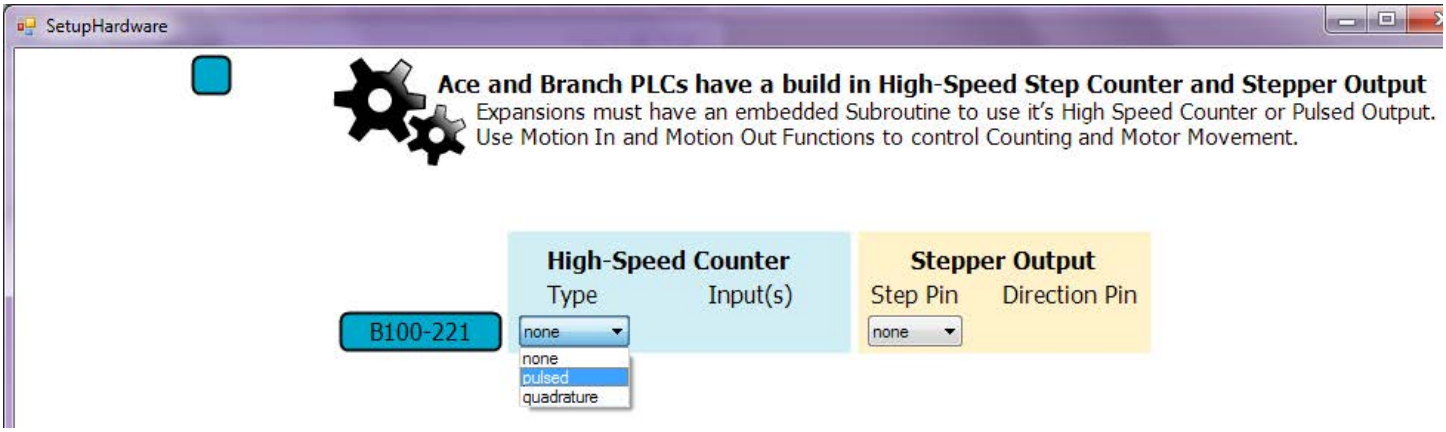


Motion In

Motion In blocks are used for high speed pulse counter inputs. There can be one high speed pulse counter per Ace, Branch or Expansion unit. The high speed counter can be configured for simple high speed pulse counting, or quadrature pulse counting. High speed pulse counting is a background task, which, once started, will continue until a counter stop block is executed.

The high speed pulse counters in each unit are specified for typical operation up to 100KHz. In reality, general high speed pulse inputs will operate at up to 250KHz. Quadrature inputs will operate at up to 200KHz. The general high speed pulse counter consumes approximately 1.6% of total processor time (for the particular PLC unit that it resides in) for every 10KHz pulse rate, while the Quadrature input consumes approximately 2.8% per 10KHz. As you increase the pulse rate, the CPU usage increases proportionately. Lower pulse rates consume proportionately less time. Whether, and at what point that might become an issue is application dependent. Up to 100KHz, it is hard to imagine an issue with most applications.

To implement high speed pulse counting, you must define a dedicated input pin (pulse) or pins (quadrature) during the hardware setup. When you are setting up your hardware from the "Setup Hardware" icon near the upper left corner, the following screen will pop up during the process.



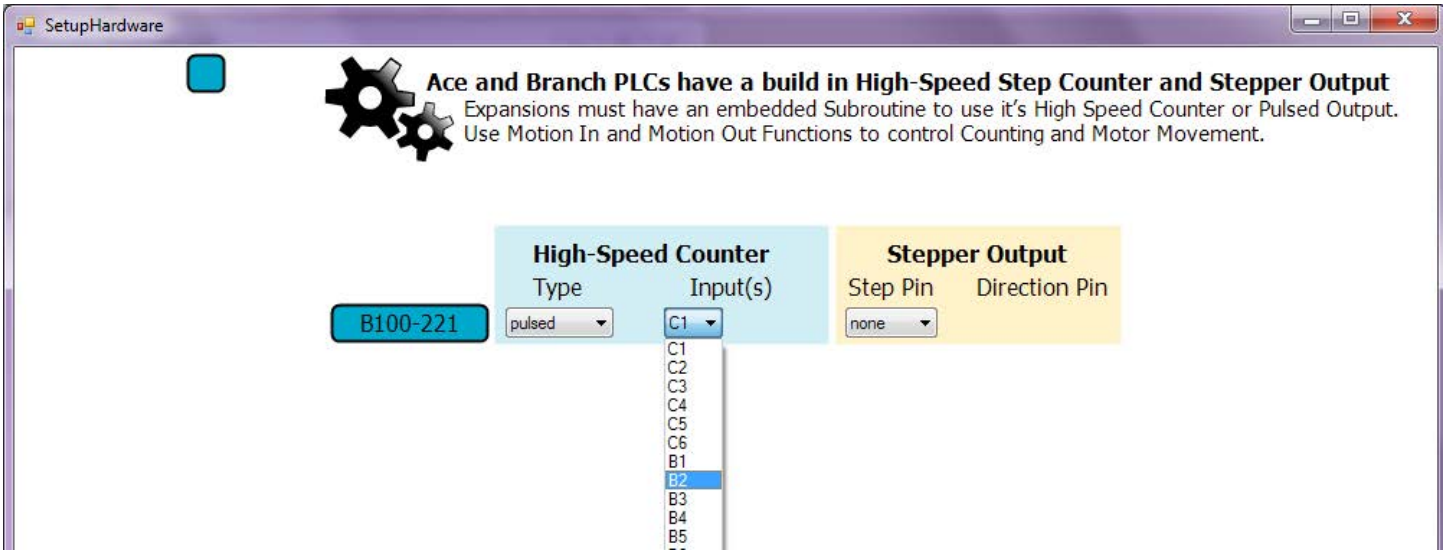
The instructions on the screen are pretty self explanatory. The screen allows you to set up for a high speed pulse counter, stepper motor output or both. To configure for a high speed pulse counter, first select the type that you want (pulse or quadrature) from the drop down selection list shown above.

MotionIn function blocks can only be used for inputs that reside on the PLC unit that contains a program. In other words, MotionIn for the main program can only be applied to digital inputs on the main PLC unit (Ace or Branch). MotionIn blocks can be utilized in the embedded object subroutine and use digital inputs that are local to the Expansion unit that contains the embedded object.

High Speed Pulse Input

A High Speed Pulse input counter will count a single input pulse train. It can be set to count up or down, and count on either a rising or falling edge transition. It is the hardware equivalent of the general Counter function. It can operate at much higher frequencies than the general Counter. It is restricted to counting transitions on a dedicated digital input signal. One high speed pulse counter can be implemented per PLC module (Ace, Branch or Branch Expansion), so up to 15 can be implemented per application.

During hardware setup, the following screen will pop up, with high speed counter and stepper motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure a high speed pulse counter for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expansion), select the Type as "pulsed" and select the particular digital input to use for your count, as shown below.

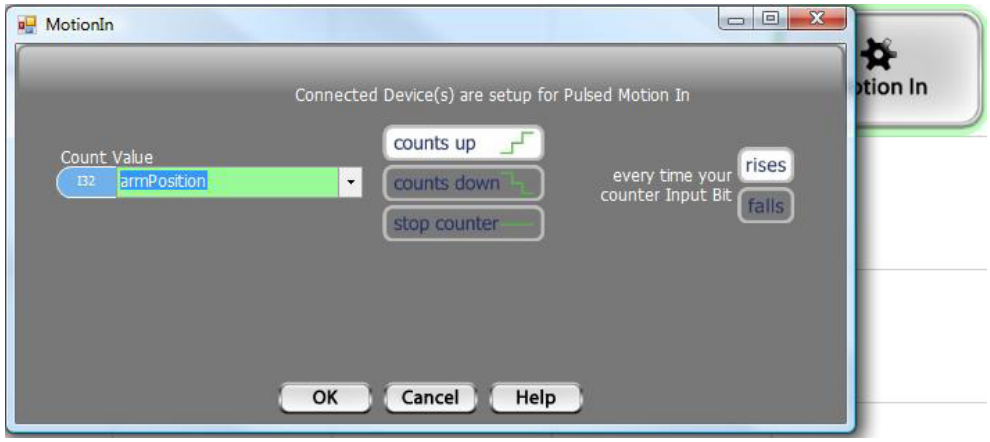


When entering your program, select the MotionIn icon and place it on a rung. A dialog box, like the one shown below, will pop up. Using the dialog box, you can select to count up or down, or stop the count. When selecting count up or down, you can select whether to count on the rising or falling edge of the input signal.

When a high speed pulse counter is started, it will continue to operate in the background until a "stop counter" block is executed.

The Count Value chosen must be an i32 tagnamed variable.

Remember the limit of one high speed pulse counter per main PLC or embedded object.



◊ Example

The example below shows that when enableHScount transitions from 0 to 1, the high speed pulse count is enabled with the count value in armPosition, to Count up on each rising transition of the input that was configured for high speed pulse counting. It will continue to count until disableHScount transitions from 0 to 1. When that happens. the high speed counter will be Stopped.

Quadrature Pulse Input

Quadrature encoders are used to keep track of position in many motion systems. With the combination of two pulse inputs, both the increment of change and the direction of change can be determined. Quadrature encoders output two high speed pulse signals, known as the A and B phase pulses. Clockwise and counterclockwise rotation is detected as a sequence of pulse signal changes, as listed below.

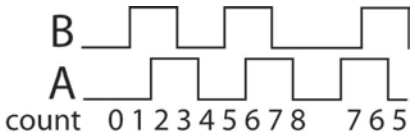
Clockwise rotation

Phase	A	B
1	0	0
2	0	1
3	1	1
4	1	0

Counterclockwise rotation

Phase	A	B
1	1	0
2	1	1
3	0	1
4	0	0

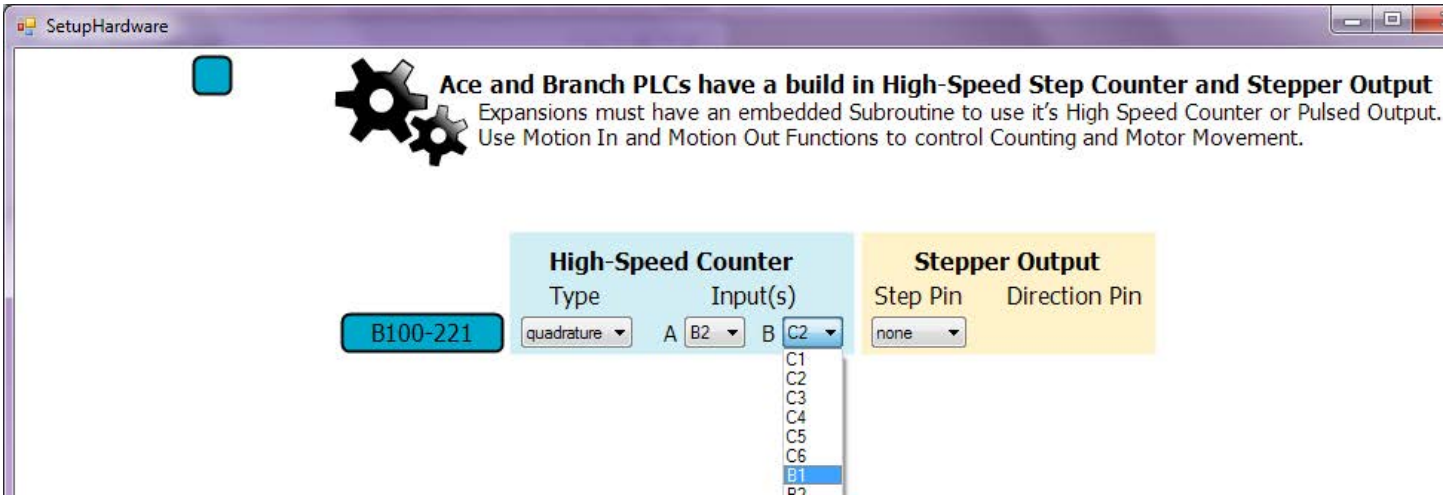
An example of a pulse train and resulting count is shown on the right.



Quadrature encoders are commonly used in servo motion systems, as well as any motion or position application. For more information, there are a number of web sites that give thorough explanations of quadrature encoders, including Wikipedia.

Quadrature Pulse In is restricted to counting transitions on a dedicated pair of digital input signals. One quadrature pulse counter can be implemented per PLC module (Ace, Branch or Expansion), so up to 15 can be implemented per application.

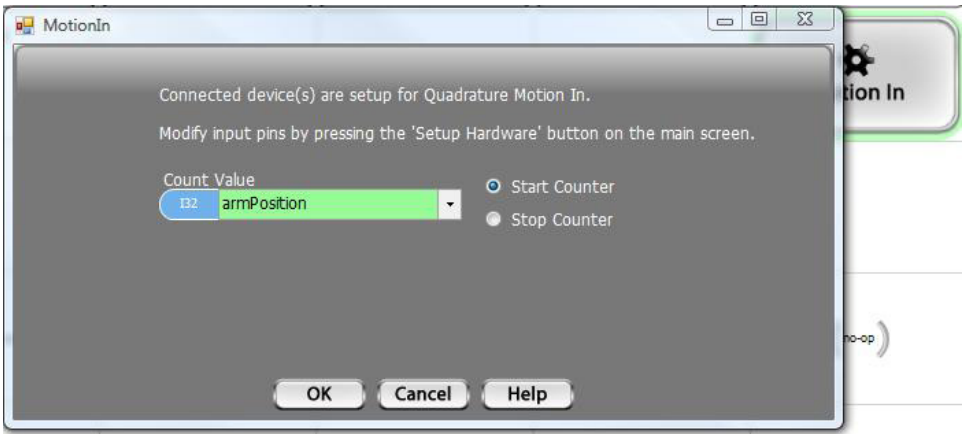
During hardware setup, the following screen will pop up, with high speed counter and stepper and motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure a quadrature pulse counter for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expansion), select the Type as "quadrature" and select the particular digital inputs to use for your count, as shown below.



When entering your program, select the MotionIn icon and place it on a rung. A dialog box, like the one shown below, will pop up. For quadrature MotionIn, you simply need to choose the appropriate Start Counter or Stop Counter selection and an i32 tagname to use for the Count Value.

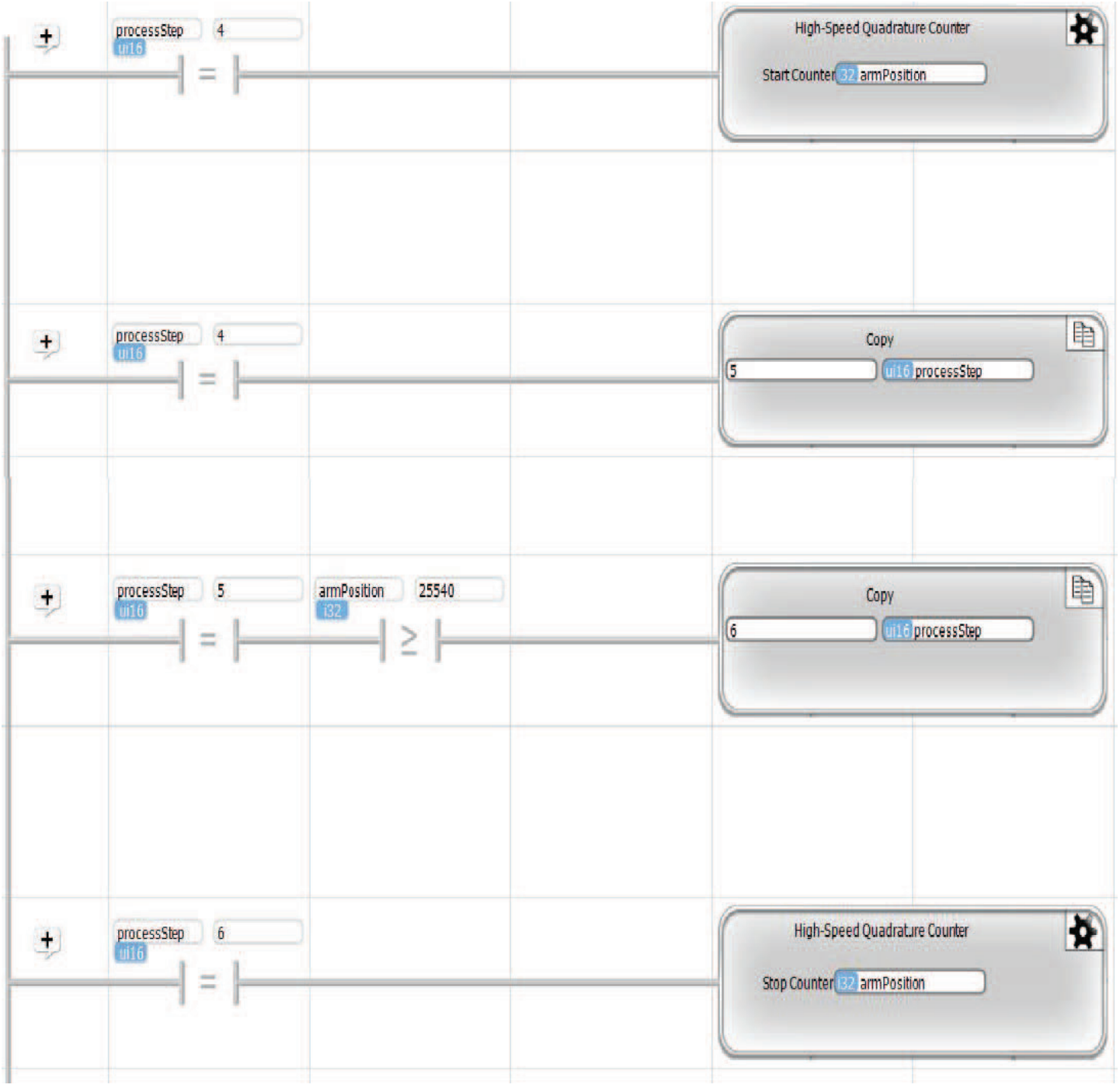
When a quadrature pulse counter is started, it will continue to operate in the background until a "stop counter" block is executed.

vBuilder will not allow you to enter a MotionIn block unless you have already configured for quadrature pulse input and chosen the input pins.



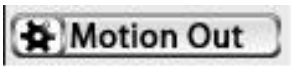
◊ Example

The example below, shows that when processStep is 4, the High Speed Quadrature Counter is Started. The next rung changes processStep to 5. During processStep 5, the armPosition is checked for greater than or equal to 255400. When armPosition becomes greater than or equal to 255400, processStep is changed to 6. In processStep 6, the High Speed Quadrature Counter is Stopped.



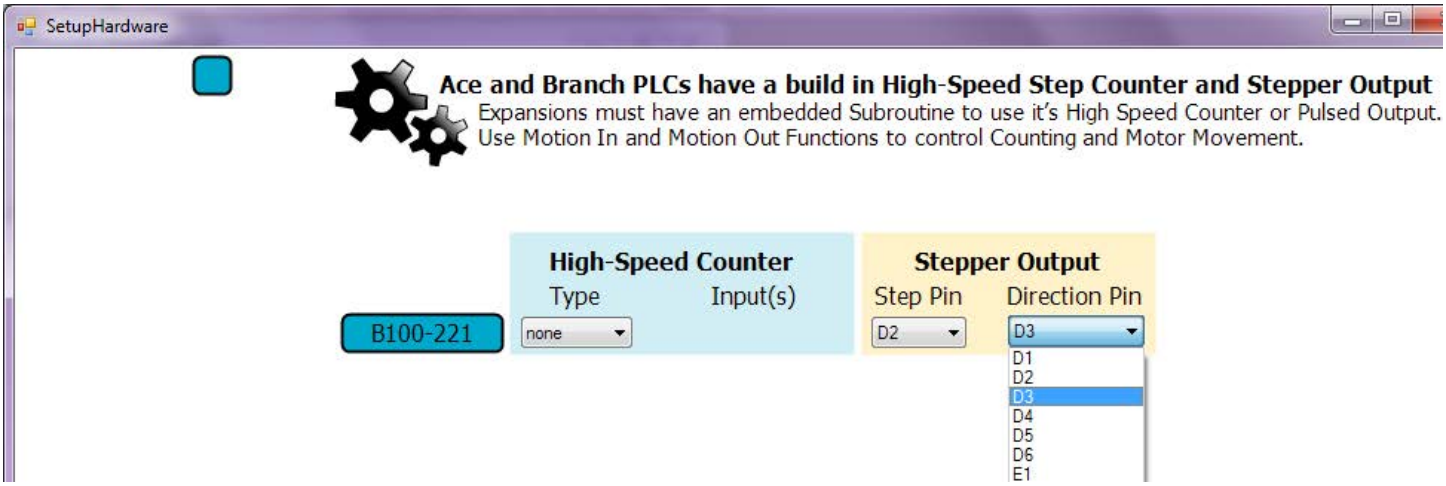
MotionOut

MotionOut blocks are used to control stepper motor motion.



Stepper motion outputs use two digital outputs. One output is a direction signal, telling the motion driver to step in either the positive or negative direction. The other signal is a high speed pulse output, which outputs one pulse to command each stepper motor step. One stepper motion control can be implemented per PLC module (Ace, Branch or Expansion), so up to 15 can be implemented per application.

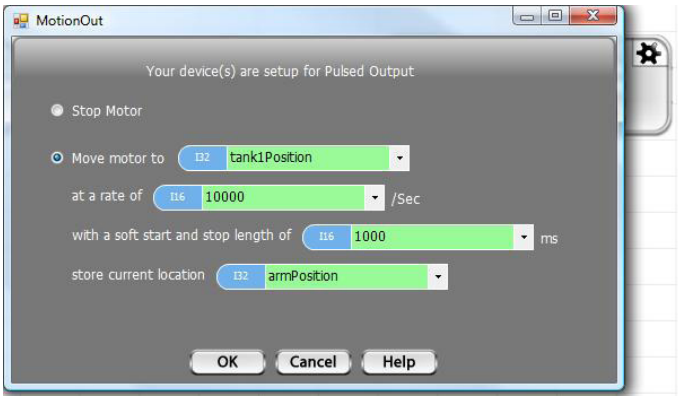
During hardware setup, the following screen will pop up, with high speed counter and stepper motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure stepper motion for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expansion), select the digital output pins to use for the Step and Direction signals. The output pins that you select will be decicated for stepper motion control and not available for any other operation.



When entering your program, select the MotionOut icon and place the MotionOut block at the end of a rung. A dialog box, like the one shown below, will pop up. For a Move command, select 'Move motor to' and either type in the numeric position that you want to move to, or select an i32 tagname that contains the target position. Next enter the pulse rate that you want the motor to move at for the majority of the move (generally the highest pulse rate). If you want the movement to ramp up to full speed to start and ramp down from full speed to stop (which is highly recommended) enter the number of millisecond to allow for the ramp up and ramp down. Lastly, provide an i32 tagnamed variable to be used for the location.

Generally, in a system, the location variable is used for all movements. Usually an initialization program is used to find the "home" position. The home position is normally assigned a location value. All movements after that are to locations relative to the home position.

When a Move is started, it will operate in the background, until the target position is reached.



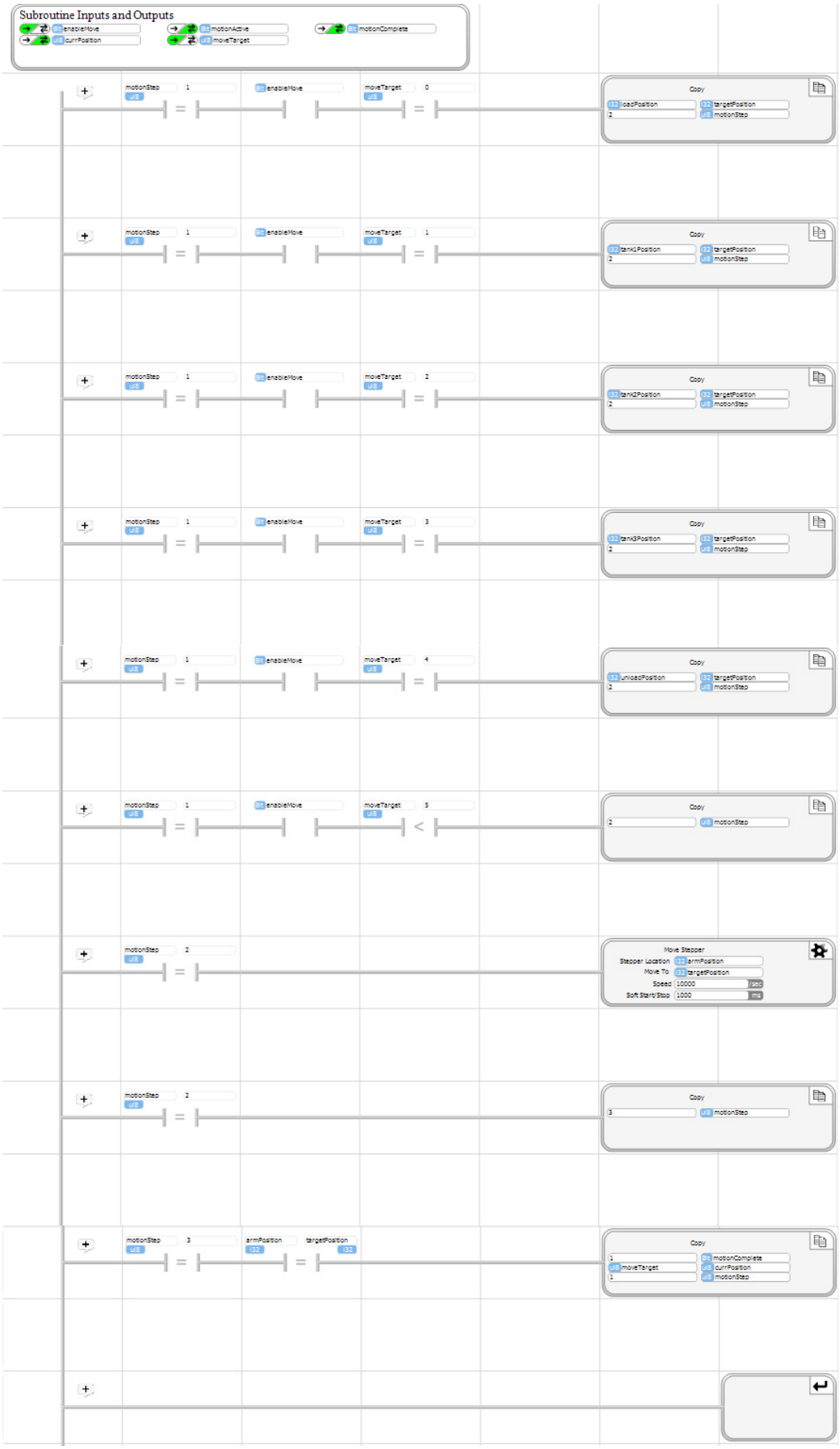
The stepper motion output is specified for a typical pulse rate up to 100KHz. Actually, pulse rates up to 250KHz are possible. Keep in mind that stepper motion control consumes approximately 1.4% of processor time per 10KHz.

◊ Example

The example on the right is subroutine that could be used for a linear transfer arm in a semiconductor wet process tool application. Wet process tools have a series of liquid tanks that perform chemical or cleaning operations on semiconductor wafers. They commonly have a transfer arm that transports the wafers between the tanks in a predefined recipe process. One of the motors would control the lateral position. In this example, we have a machine with a load station, three liquid tanks and an unload station. This example illustrates the control of that lateral position. This subroutine would be called with an enableMove flag indicating that a move is required and the moveTarget indicating where to move. In motionStep 1, the targetPosition is established based on the enable & moveTarget, and motionStep is changed to 2.

If a motion is called for in motionStep 1, motionStep 2 will immediately execute to start the Move, then set motionStep to 3.

In motionStep 3, the subroutine waits until the armPosition is at the targetPosition. When it is, it passes back motionComplete and the tank, or load/unload station number where the arm is currently positioned and sets up to wait for the next move command.



PID

The program block icon for the PID function is shown on the right.

PID is Proportional - Integral - Derivative control. It is the very commonly used in process control, heating control and motion control. The PID function calculates the "error' value between a measured process variable and the desired setpoint. It adjusts an output in an attempt to minimize the error. This adjustment occurs on a regular period. Each time an adjustment is made, it takes into account the magnitude of the error (Proportional), the accumulated total of the error over time (Integral), and the rate of change of the error (Derivative).

PID control is complex enough that we won't provide a detailed discussion in this manual. It is a common control algorithm and there are other excellent reference documents that you should review to get an understanding, if you are new to PID. One very good description is on Wikipedia at the following link.

http://en.wikipedia.org/wiki/PID_controller

We will also put out an application note on PID control and how to use it in the near future.

In vBuilder, PID control is a background task. That means that you Start it and it will continue to run in the background, constantly calculating the error, integrating it and determining its rate of change and updating the output based on these factors combined with the constants that you set up. Once started, the PID will operate until it is Paused

The basic PID equation used in the PID function of vBuilder is :

$$O = P*E + I * \int E\,dt + D * \Delta E/dt$$

where : O : Output
P : Proportional constant (sometimes referred to as the Gain)
I : Integral constant (sometimes referred to as Reset)
D : Derivative constant (sometimes referred to as Rate)
E : Error
dt : change in time

When you place a PID block in your program, a dialog box, like that shown below, will pop up. With this dialog box, you can Start the PID operation, Pause it, or Reset the PID's integral. A PID function is one that you normally need to "tune" to get the right performance. By tuning, we mean adjust the constants, try it, adjust again, until the response works like you want.

PID Start/Continue

The Start/Continue selection in the PID dialog box creates a function block that does exactly what it says. If the PID is not active, it will Start the PID's operation. If it is already running, it will continue operation. A PID actually only has to be started once. It will continue to operate until it is Paused.

When you select Start/Continue, you must select or enter a number of parameters. The following is a list of those parameters, with an explanation of each.

- **Output** : This is what the PID is actually adjusting. The Output should have a direct impact on the value of the Process Variable. For example, a butterfly valve controlling the gas supply to a boiler's burners has a direct impact on the temperature of the boiler.
- **Output Max** : The maximum value allowable for the Output. Regardless of the result of the PID calculation, the Output will be restricted to no more than this value. [any variable type except Bit and ui8]
- **Output Min** : The minimum value allowable for the Output. Regardless of the result of the PID calculation, the Output will be restriced to no less than this value. [any variable type except Bit and ui8]
- **Process Variable** : The is the measurement of the parameter that you are trying to control.
- **Set Point** : This is the desired value of the Process Variable at this time
- **Input Max** : The maximum value of the Process Variable measurement that will be used for the PID calculation. If the actual measurement goes above this value, this value will be used.
- **Input Min** : The minimum value of the Process Variable measurement that will be used for the PID calculation. If the actual measurement goes below this value, this value will be used.
- **Proportional** : The constant multiplier to the error.
- **Integral** : The constant multiplier to the integral of the error.
- **Derivative** : The constant multiplier to the derivative of the error.
- **Sample rate** : The amount of time between each PID calculation/adjustment, in milliseconds.
- **Freeze bias** : This is a selectable option. If selected, it is used to limit the impact of integral portion of the equation to no more than the entire Output range. This is useful in preventing overreaction after a time period where some factor prevented the actual control of the Process Variable [which could possibly result in a huge integral value].

When you Start a PID, it will continue operating at the defined sample rate, behind the scene. In other words, once you start it, you don't need to continue to execute PID Start/Continue blocks for operation. It doesn't hurt anything if you do. Its just not necessary.

PID Reset

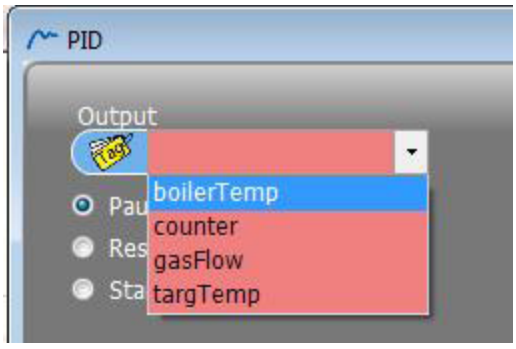
A PID Reset sets the Integral value to the value required to produce an Output equal to the Reset Value. It is something that you should do when starting a PID. The Start/Continue does not initialize the Integral value, because it does not know whether this is the actual Start or whether it is a Continue.

When you place a PID Reset in a ladder rung, select the Output variable of the PID that you want to reset, as shown on the right. Select the value of the PID output to start the PID with.

PID Pause

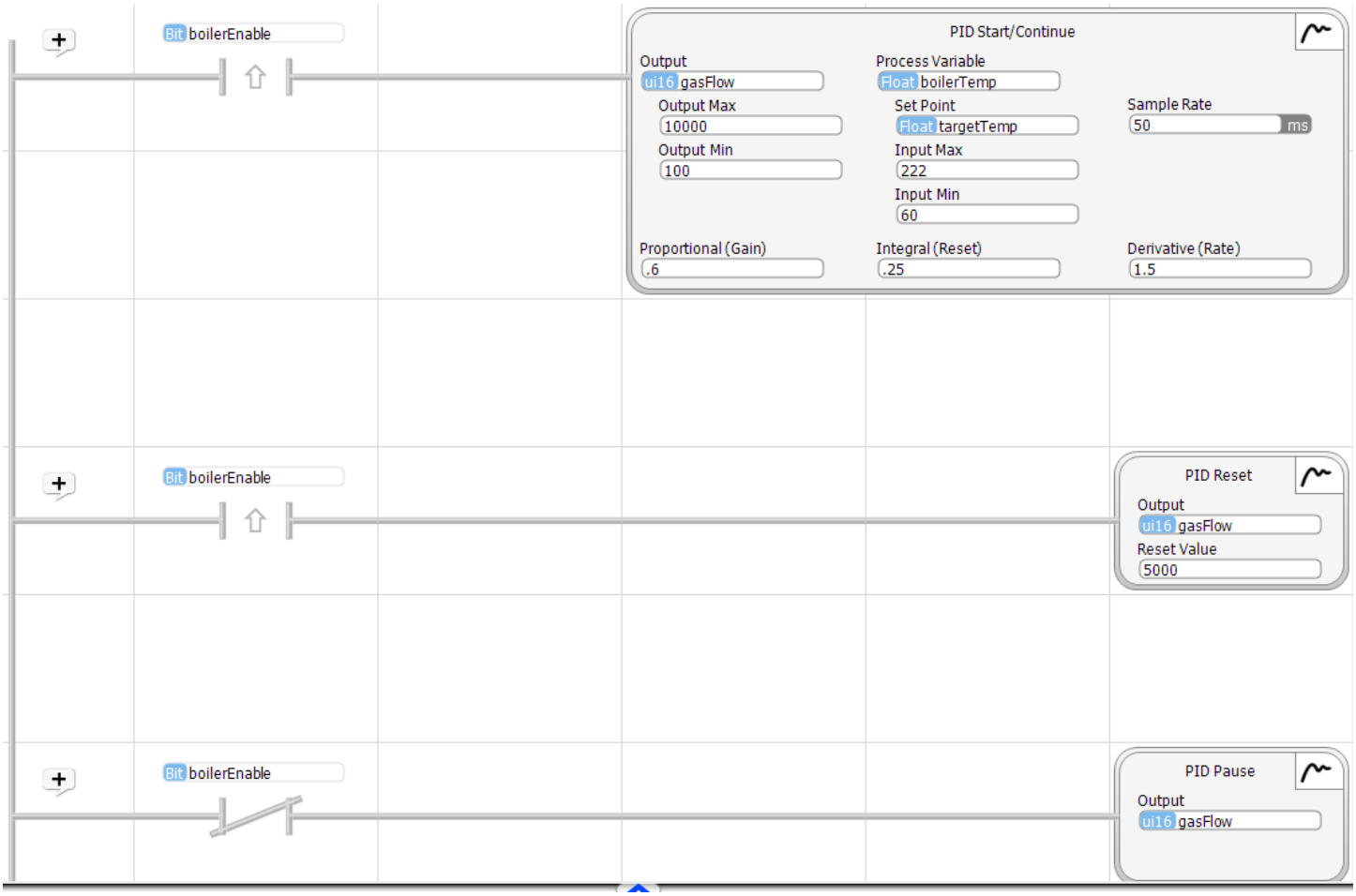
A PID Pause block stops the PID from operating.

When you place a PID Pause in a ladder rung, select the Output variable of the PID that you want to pause, as shown on the right.



◊ Example

The example, shown below, illustrates a PID used to control a boiler. The PID Start and Reset will occur one time when the boilerEnable transitions from inactive (0) to active (1). When boilerEnable is inactive (0), PID is Paused.



Ramp

The icon for the Ramp function is shown on the left.

The Ramp function changes a value from its initial value to a target value, at a defined rate of change. Optionally, a ramp can include a soft start and soft stop. A soft start ramps the rate of change up to the defined rate of change gradually, over the defined soft start/stop period. A soft stop ramps the rate of change from the defined rate to 0, over the defined soft start/stop period.

Ramp is commonly used for motion control. The vBuilder Stepper Motion function has its own ramp feature built in, so this Ramp function would not commonly be used for stepper motion control. It would be applicable to servo motion control, typically coupled with a high speed pulse counter input and PID.

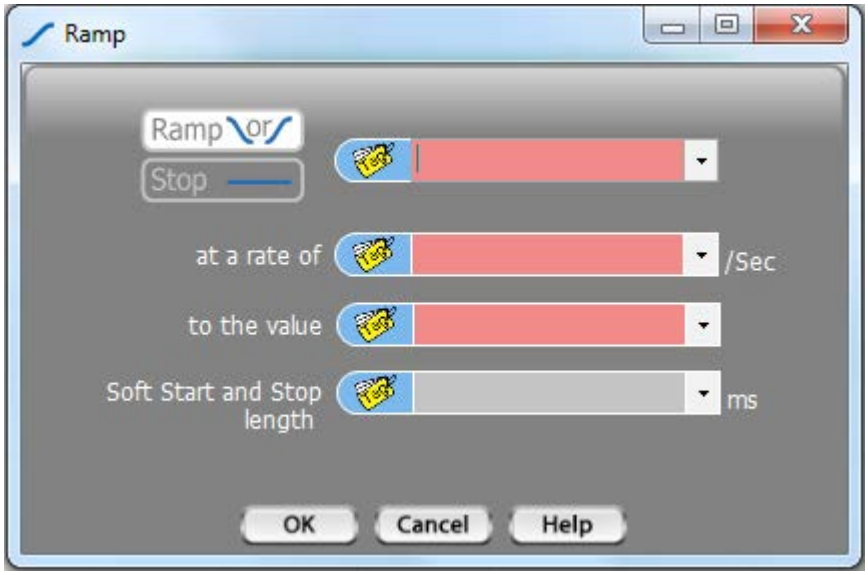
Ramp is also applicable to other machine and process control applications. Any time you want to change a variable at a defined rate, Ramp can be used.

Ramp is a background task. It only needs to be Started. Once Started, it will continue to operate in the background until Stopped.

Ramp Start

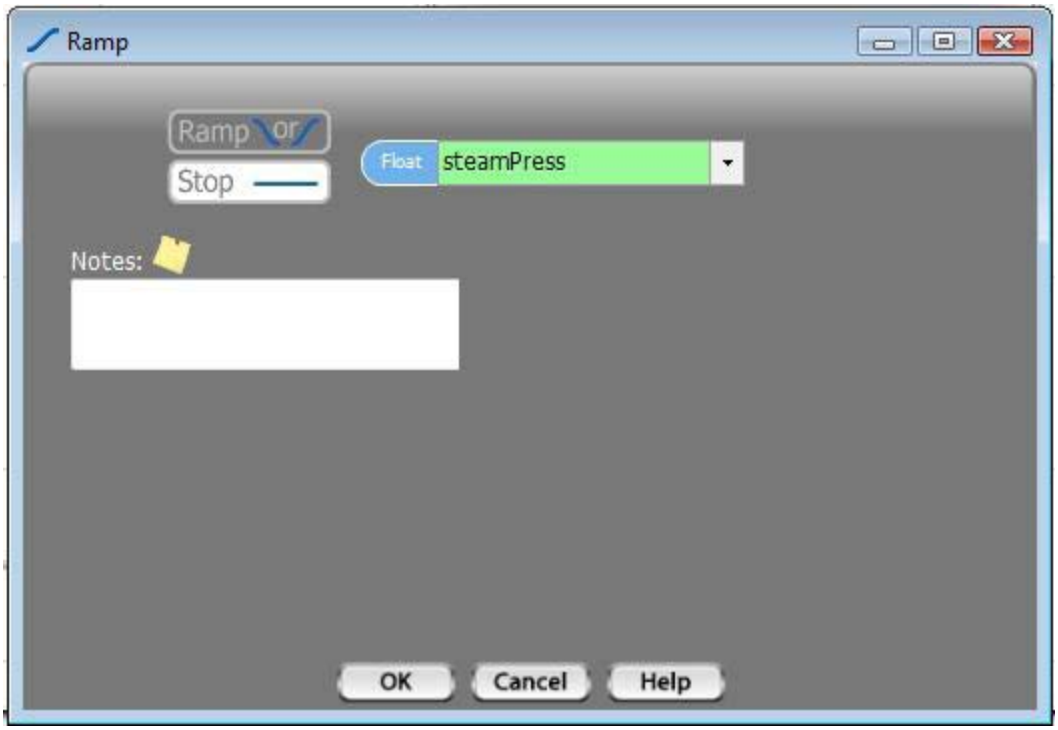
When you place a Ramp block in a flow chart, a dialog box will pop up. This box will allow you to Ramp Start, or Ramp Stop. To Start a Ramp, select the Ramp button, then enter the parameters.

- The top selection is the variable that you want to Ramp.
- Next is the ramp rate in units per second of whatever unit the Ramp variable is. The Ramp Rate is an absolute value. Whether it is a rate of increase or decrease depends on whether the Ramp variable is above or below the target.
- Next is the Ramp Target - the .final value of the variable after ramping
- Optionally, you can enter a Soft Start/Stop time. If you enter a Soft Start/Stop time the ramp rate will ramp up to the Ramp rate over the defined period and ramp down over the same defined period, when approaching the target. This will have the effect of rounding off the value curve, like that shown in the Ramp button graphic.



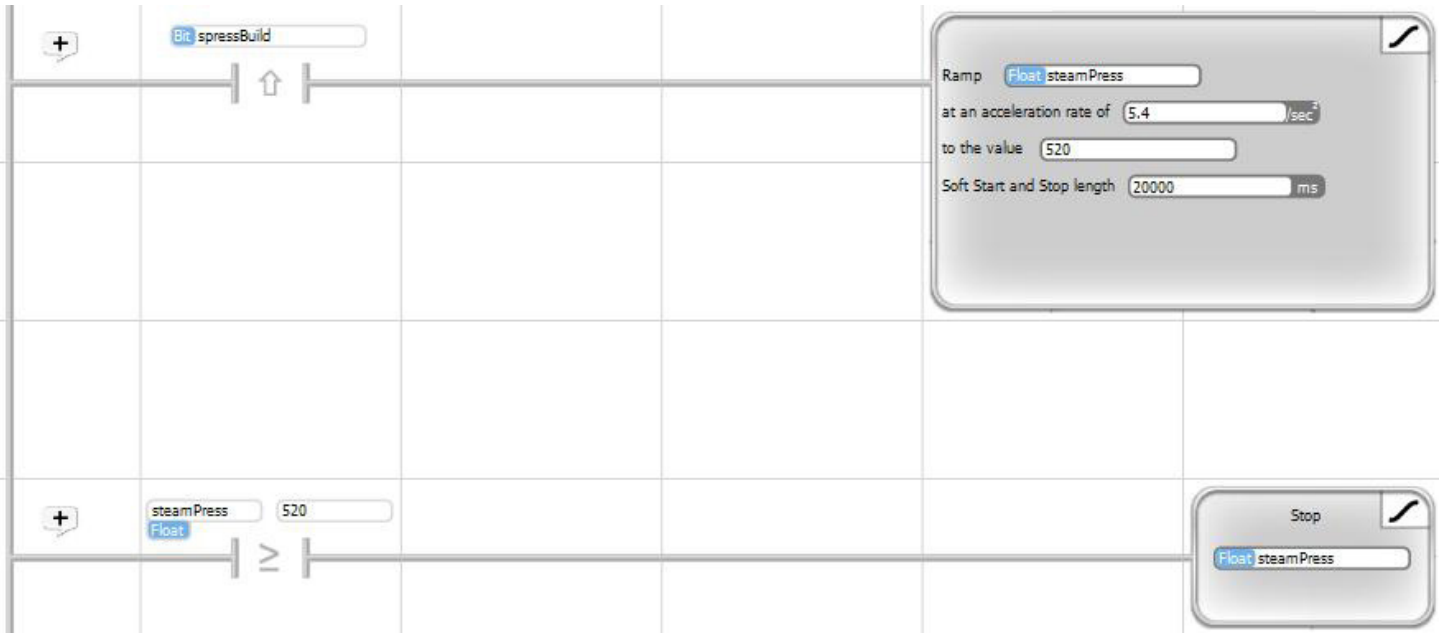
Ramp Stop

Ramp Stop will Stop the Ramp function. This is something that you would normally do, once the target value has been reached. When you place a Ramp block, select Stop, as shown on the right. Next, select the variable that is being ramped. Select OK.



◊ Example

The example, below, shows the ramping of a steamPress setpoint. The Ramping is initiated when pressBuild transitions from inactive (0) to active (1). When the steamPress setpoint becomes greater than or equal to the target value, the Ramp is Stopped.



Scale

The Scale program block icon is shown on the left.

The Scale function is used to automatically convert an input value to an output value. It is commonly used for scaling a raw analog reading to its equivalent value. Scale has the capability to do piecewise linear scaling between up to 16 points. With piecewise linear scaling between such a high number of points, Scale can be used to convert values from non-linear transducers as well as an assortment of other applications.

When you place a Scale block, a dialog box, like that shown below will pop up. In the dialog box, you need to choose the Input and Output variables. You must also define the scaling in the Input and Output tables, as shown. The scale curve will automatically be created on the right to illustrate the scaling that will take place.

◊ Example 1

The example shown below, shows the simple scaling of a 12 bit A/D conversion of a pressure transducer that outputs -14.7 to 3000 psi as a 0 to 5V signal. As shown in the graph, it results in a linear conversion curve.

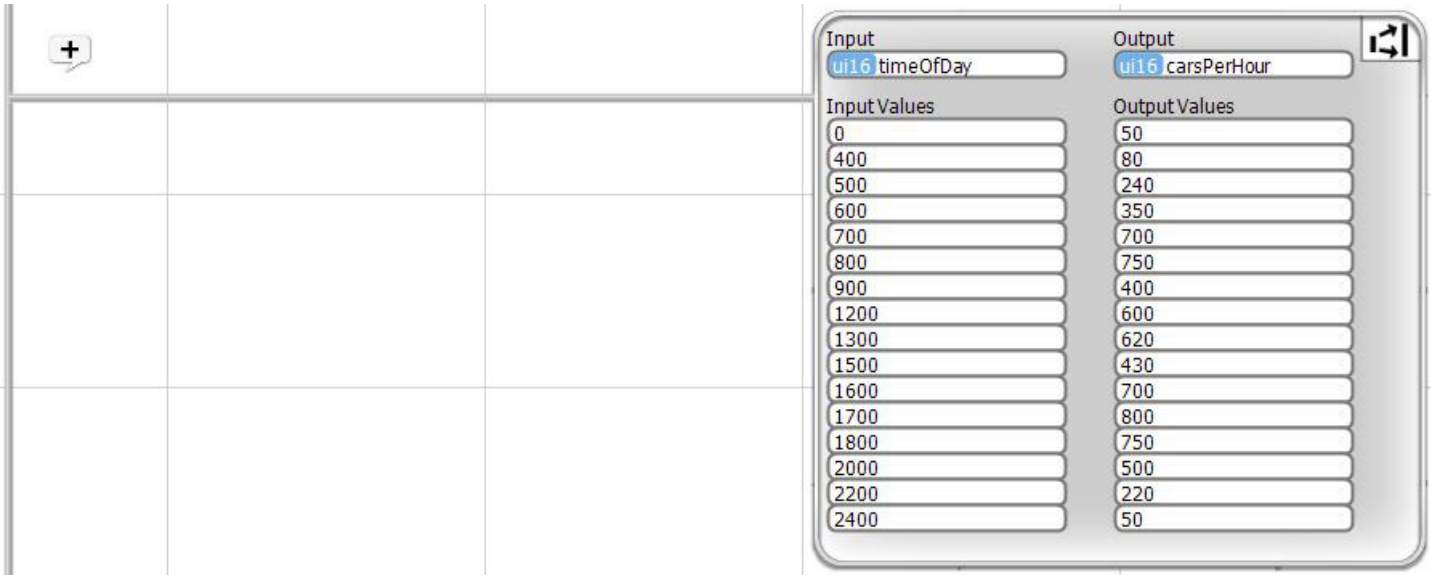
After clicking OK, the program block will be like that shown below.

A more complex use of Scaling is shown on the next page.

◊ Example 2

The example on the right is for a traffic control application. In this application, the average rate of cars per hour that go through a particular intersection is known for 16 different times of the day. The table is set up, as shown. Between any two known points, piecewise interpolation is a good estimate of the expected number of cars per hour at any particular time.

This type of Scaling could be used in a traffic light control application. If seperated Scaling tables were utilized for traffic in different directions, the on and off times for red, and green lights could be dynamically adjusted during the day to minimize traffic delays and maximizing throughput.



Shift/Rotate

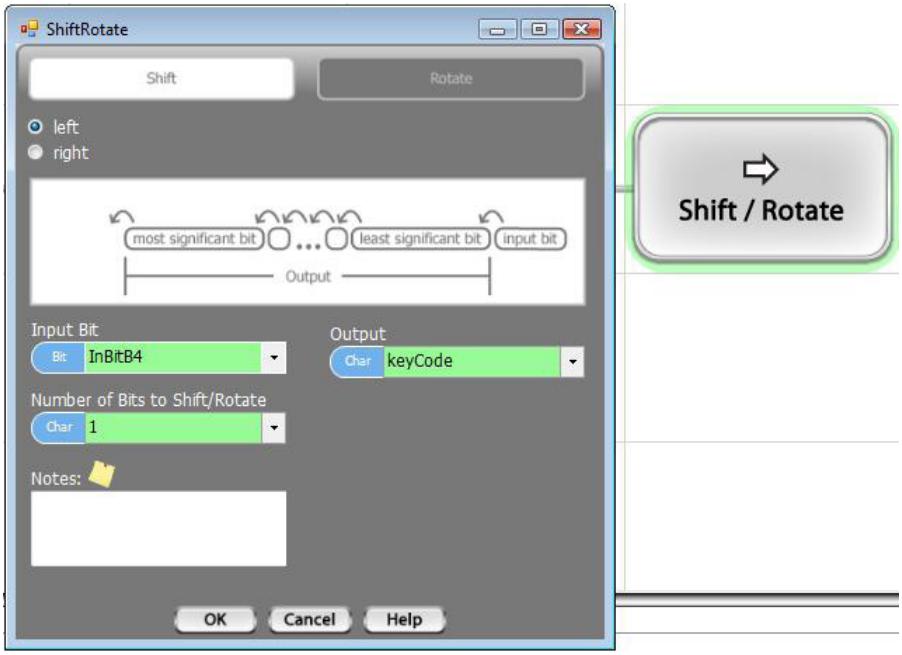
The Shift and Rotate functions provide a mechanism for shifting or rotating bits in an integer number.

The Shift function allows you to shift an integer number left or right, a selected number of bit positions. A bit value that you choose is shifted into the vacated bit position(s). For signed integers, The shift can be defined to include or exclude the sign bit (the most significant bit).

The Rotate function is similar to the Shift. The difference is that the bits that are shifted out of one end of the number are shifted back into the vacated position(s) on the other end.

Shift (unsigned number)

The dialog box for a Shift is shown below. When you place a Shift/Rotate block, select the Shift button in the dialog box. A graphic showing the Shift operation will be displayed in the dialog box. Select the direction that you want to Shift. If you change the direction, the graphic will change to reflect the shifted direction. Select the tagnamed variable that you want to Shift, as Output. Select the bit value that you want to Shift into the vacated bit position. If you are shifting more than one bit position, this value will be shifted into all of the vacated bits. Lastly, if this is an unsigned number, select the number of bit positions to Shift. Click OK.



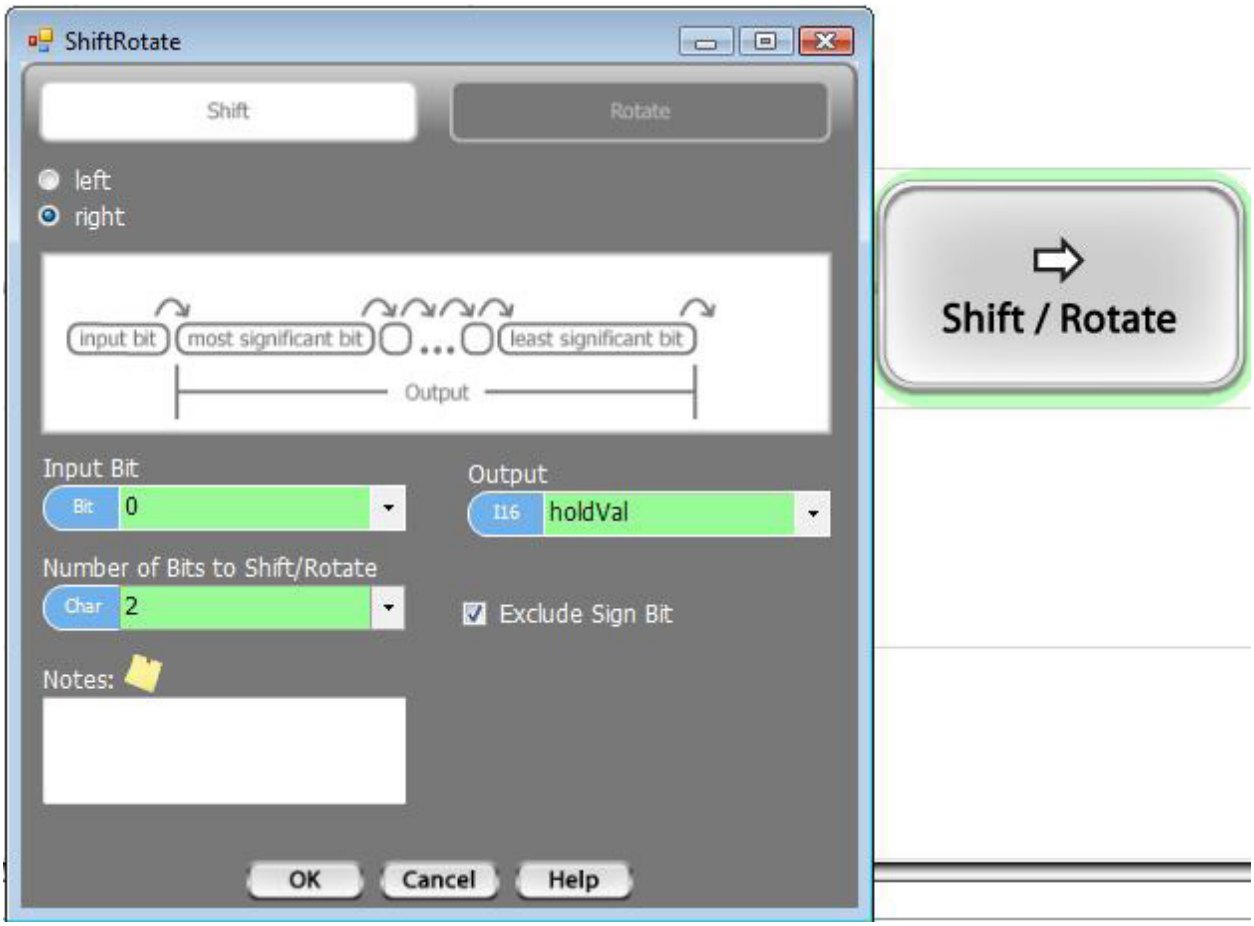
◊ Example Shift of Unsigned Integer

The example, below, shows a shift of a ui8 variable, named keyCode. keyCode will be shifted to the left, one bit position, which the value of InBitB4 shifted into the least significant bit.

Shift (signed number)

There is an additional option, when Shifting a signed number. When you select a signed integer tagname, a checkbox will pop onto the dialog box, which says "Exclude Sign Bit". You can choose to check or uncheck this box. If you check it the Shift operation will not include the sign bit (the most significant bit position). In other words, if you check the box, the sign bit will remain unchanged. If you are shifting right and check the box, the bit that is shifted in will be shifted into the second most significant bit position.

If you uncheck the Exclude Sign Bit box, shifting will include the entire number.

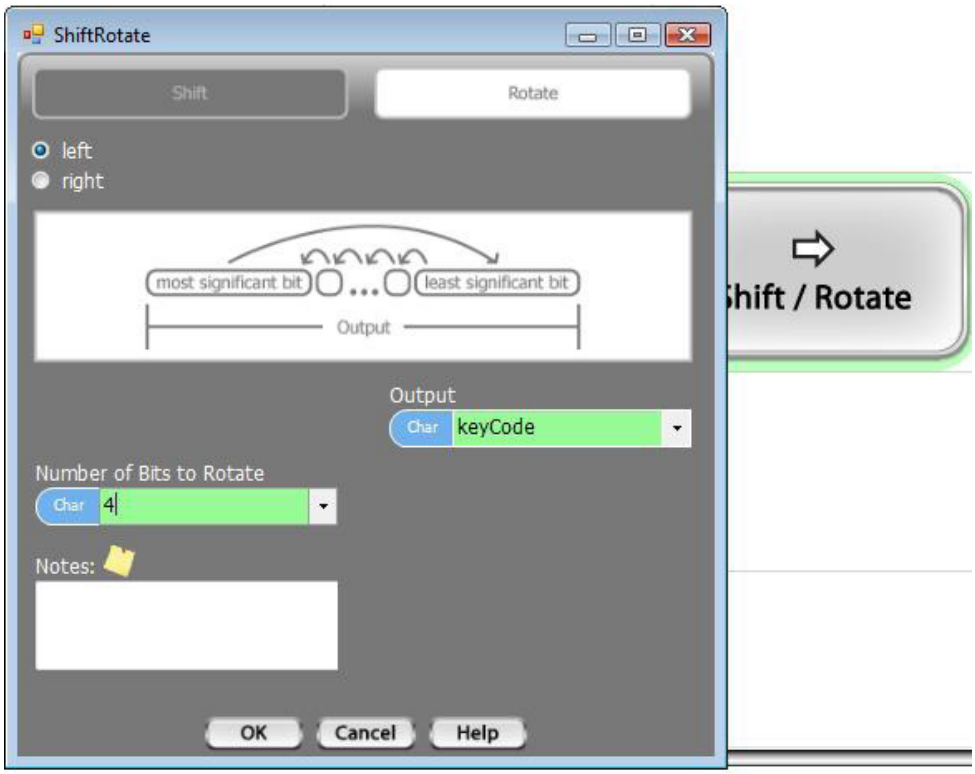


◊ Example Shift of Signed Integer

The example below shows a Shift of an i16 variable, named holdVal, by two bit positions to the right. The sign bit is excluded from the shift. A value of 0 is shifted into the vacated bits.

Rotate (unsigned number)

The dialog box for a Rotate is shown below. When you place a Shift/Rotate block, select the Rotate button in the dialog box. A graphic showing the Rotate operation will be displayed in the dialog box. Select the direction that you want to Rotate. If you change the direction, the graphic will change to reflect the rotated direction. Select the tagnamed variable that you want to Rotate, as Output. The bit rotated out will be rotated back in to the vacant bit position on the opposite end of the variable. Lastly, if this is an unsigned number, select the number of bit positions to Rotate. Click OK.



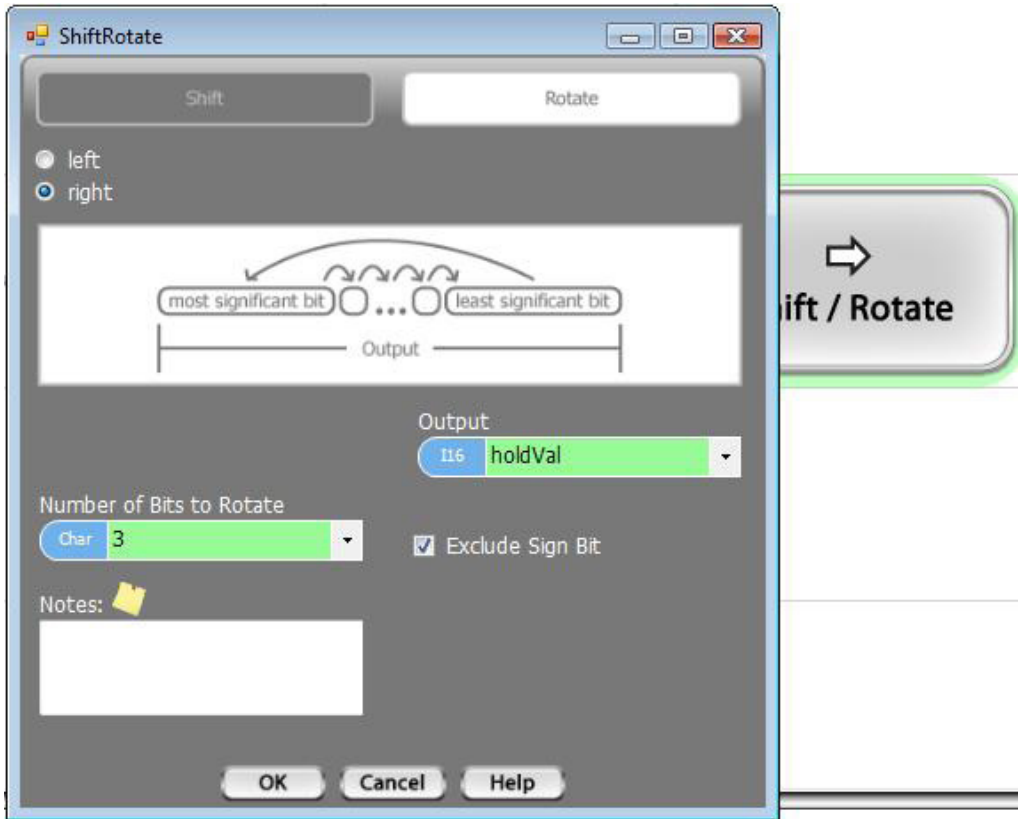
◊ Example Rotate of Unsigned Integer

The example, below, shows a Rotate of a ui8 variable, named keyCode. keyCode will be rotated to the left, four bit positions, with the value of bits shifted out, successively shifted back into the least significant bit.

Rotate (signed number)

There is an additional option, when Rotating a signed number. When you select a signed integer tagname, a checkbox will pop onto the dialog box, which says "Exclude Sign Bit". You can choose to check or uncheck this box. If you check it, the Rotate operation will not include the sign bit (the most significant bit position). In other words, if you check the box, the sign bit will remain unchanged. If you are rotating right and check the box, the bit that is rotated in will be rotated into the second most significant bit position.

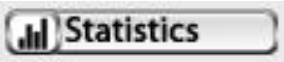
If you uncheck the Exclude Sign Bit box, shifting will include the entire number.



◊ Example Rotate of Signed Integer

The example below shows a Rotate of an i16 variable, named holdVal, by three bit positions to the right. The sign bit is excluded from the rotate.

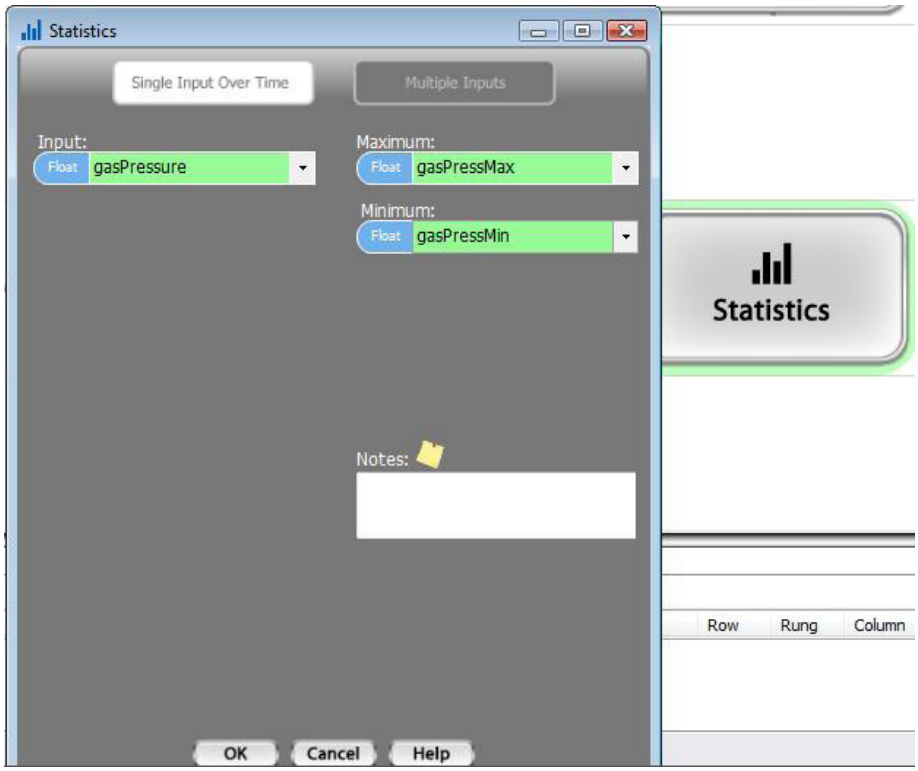
Statistics



Two basic, common statistical functions are available for a Statistics block. The Single Input Over Time option automatically keeps up with the minimum and maximum value of a selected variable, since the time the monitoring was started. The Multiple Inputs option will use a list of up to 16 input values and calculate the Maximum, Minimum, Median and Average values.

Single Input Over Time

When the Single Input Over Time button is selected for a Statistics block, the dialog box will appear as shown on the right. The Input is the tagname of the variable that you want to monitor. The Maximum is the tagname where you want to store the maximum value of the selected input. The Minimum is the tagname for storing the minimum value.



◊ Example Single Input Over Time

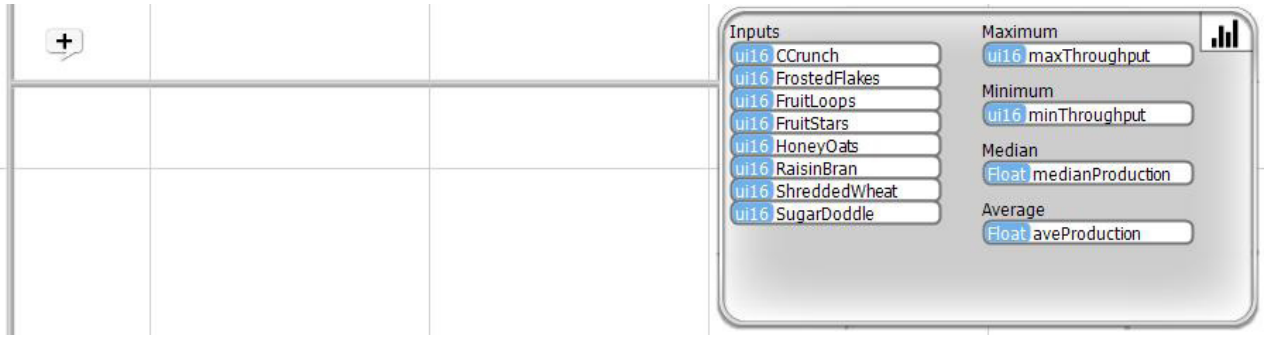
The example below will determine and maintain the Minimum and Maximum value of gasPressure. Each time this program block is executed, the current value will be compared to the previous minimum and maximum. If the value is below the previous minimum, the Minimum will be set equal to the current value. If it is above the previous maximum, the Maximum will be set equal to the current value.

Multiple Input Statistics

When Multiple Inputs is selected for a Statistics block, the dialog box will appear as shown on the right. Up to 16 input tag-names can be selected. The Multiple Input Statistics block can be set up to determine and calculate any of Maximum value, Minimum value, Median value and Average value. If you do not select a tagname for any of the statistical operations, this program block will simply skip that function. You must select at least one statistical operation though.

◊ Example Multiple Inputs

The example below is monitoring eight cereal box fill lines. This cereal factory has eight different lines that are side by side and have identical capabilities and capacities. A Statistics Multiple Inputs block is used to monitor and create meaningful real time management information on how the production lines are doing. In this case, all of the statistics are calculated.



Subroutine

The Subroutine block will place a call to a Subroutine.

Before you can create a Subroutine call, you must have already created the subroutine.

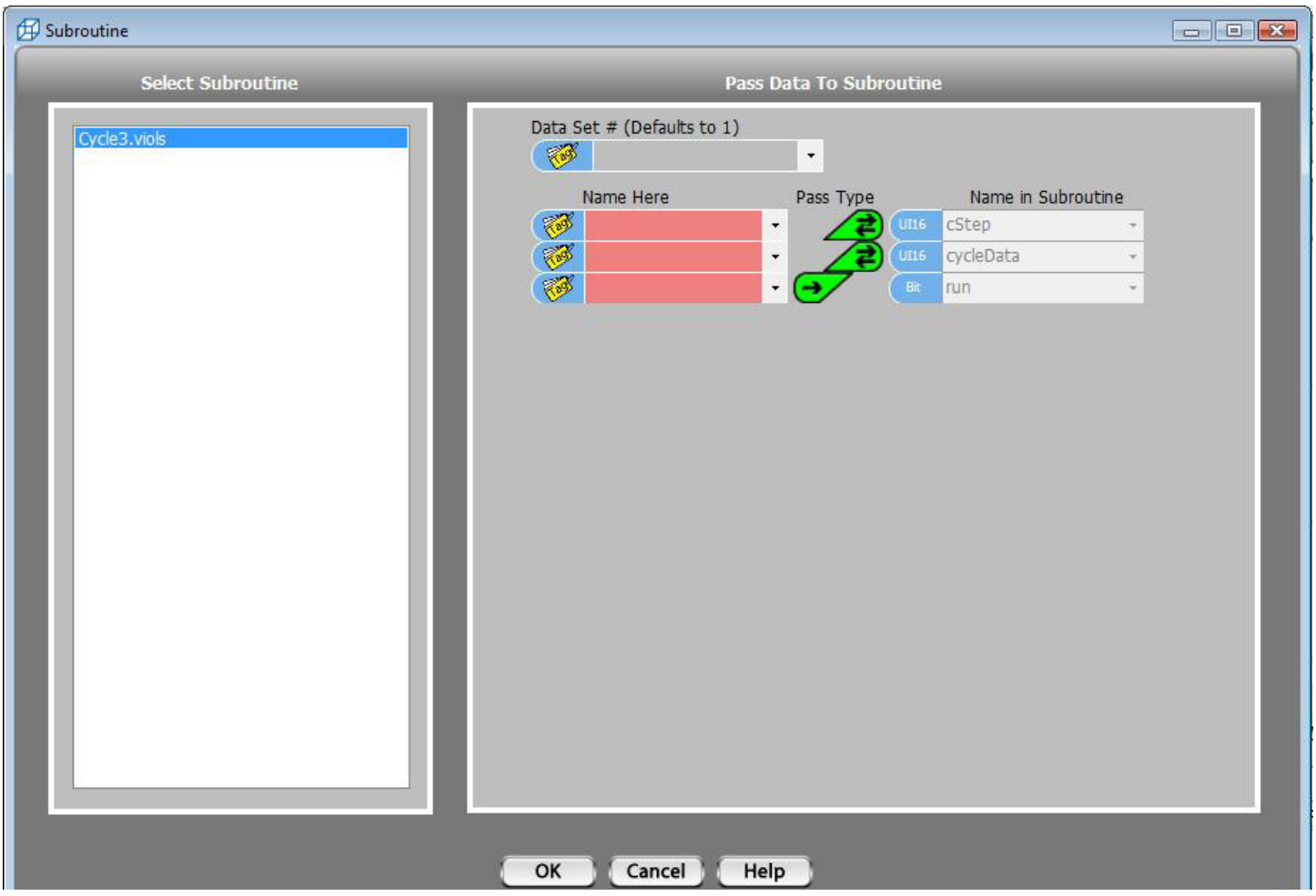
Subroutines are discussed in detail in depth in the chapter on Object Oriented Programming and Subroutines.

When you place a Subroutine Call block, a dialog box, like the one shown below, will pop up. On the left side of the box, is a list of all of the defined subroutines. Choose the one that you want to place. Once you'e done that, the right side of the dialog box will contain a list of Data to Pass to the Subroutine. Since all Subroutines in Velocio PLCs are objects, the first thing you need to do is define the Data Set # (object number) that this Subroutine Call is associated with. If you only have one instance of the Subroutine, you can leave this blank.

Next, you see a list of Passed Data. On the right hand side of the list is the subroutine tag name associated with each item - in other words, the tag name used in the subroutine. On the left are selection boxes to allow you to pick the local tagnames of the data to pass. Between the two columns is the Pass Type for each parameter. The following is an explanation of the Pass Types.

- The symbol with a single arrow pointing to the subroutine indicates pass by value. The numeric value of the data on the left (tagname variable data or constant) is passed into the subroutine. There is no effect on the data in the calling program.
- The symbol with arrows going both directions indicates pass by reference. In this case, actual data is not passed. What is passed into the subroutine is a "reference" to the tagname variable. This reference allows the subroutine to read and/or change the value in the calling program. This is how data is passed back out of a subroutine.

Fill out the Pass Data to Subroutine list and click OK.



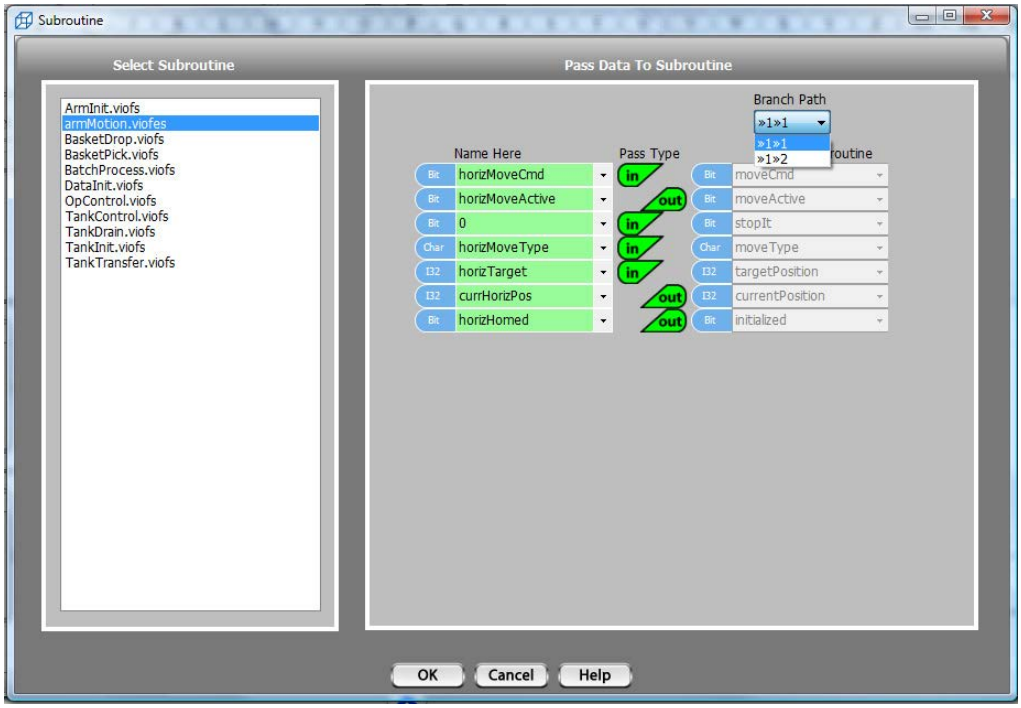
◊ Example Subroutine Call

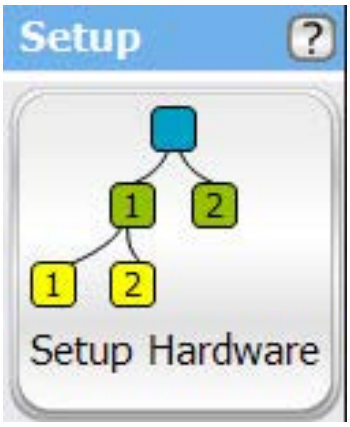
The example below shows a Subroutine Call block. The Call is to Cycle3, object #2 (data set #). Two parameters are passed by reference and one is passed by value.

Calling Embedded Object Subroutines

Calling an Embedded Object Subroutine is basically the same as calling a local subroutine. There are a couple of differences though.

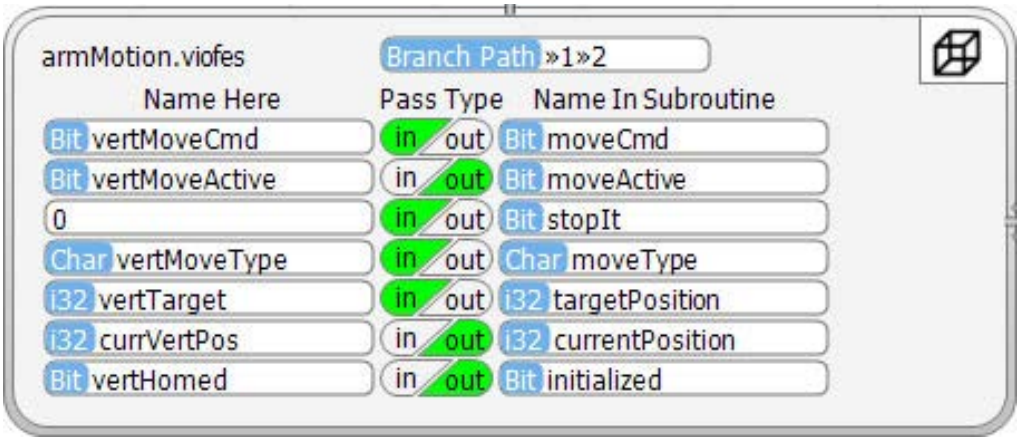
- Parameters passed to and from Embedded Objects are In and Out. There is no pass by reference. You either send data to the object or receive it from the embedded object.
- If there is more than one device configured for the same embedded object, you must select which one you are calling. The particular embedded object is defined by its branch path. The branch path is defined by its port connection relative to the unit the call is made from. For the example shown on the right, the choice is between the unit connected through its first vLink port, then through that unit's first vLink port (>>1>>1), or the one connected through its first vLink port, then through that unit's second vLInk port. The Setup graphic makes it pretty clear. In this case, the two choices are yellow 1 and yellow 2.





◊ Example Embedded Subroutine Call

The example below shows a call to an embedded object subroutine located >>1>>2 relative to the calling program. Since the calling program is in the Branch unit, the call is to yellow 2.



Sub Return

The Subroutine Return block is available only for subroutines. It is simply a block that returns to the program that called the subroutine.

The Subroutine Return has no dialog box. In ladder logic, vBuilder automatically places a Subroutine Return block at the end of a subroutine program. You can place additional Subroutine returns on other rungs, as needed.

◊ Example

The example, below, shows a Subroutine Return block placed in a subroutine.

Timer

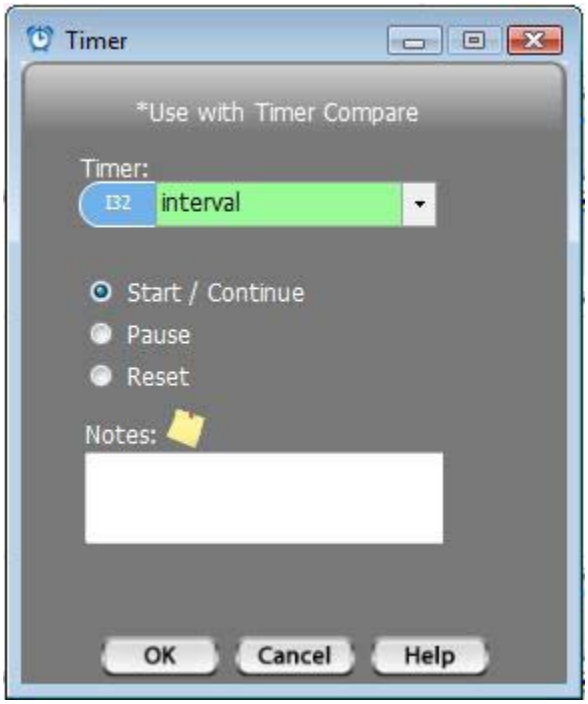
In vBuilder, Timers are background functions. Once you start a Timer, it will continue to run, until you Pause it. While it is running, you can execute Timer Compare blocks to test the time against target values. The Timer value can be Reset at any time with a Timer Reset block.

Timer Start/Continue

When you want to Start a Timer, select the Start/Continue option in the dialog box. The only other thing you need to do is place the tagname of an i32 variable to be used to contain the time value in the box labeled Timer.

Timer Start/Continue will not initialize the timer value to 0. If you want to do that, you should use the Timer Reset function.

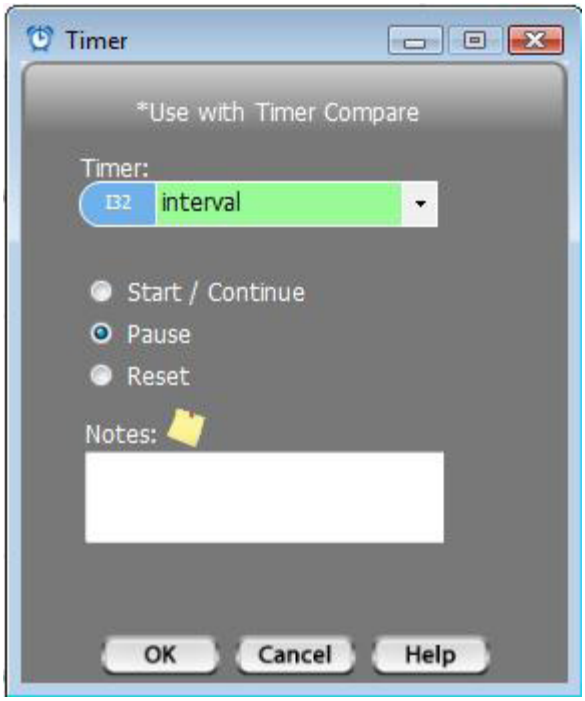
The Timer Start/Continue only needs to be executed once for a Timer to operate. It does not need to execute on every program pass. If it does, that's the Continue. It will continue to operate.



Timer Pause

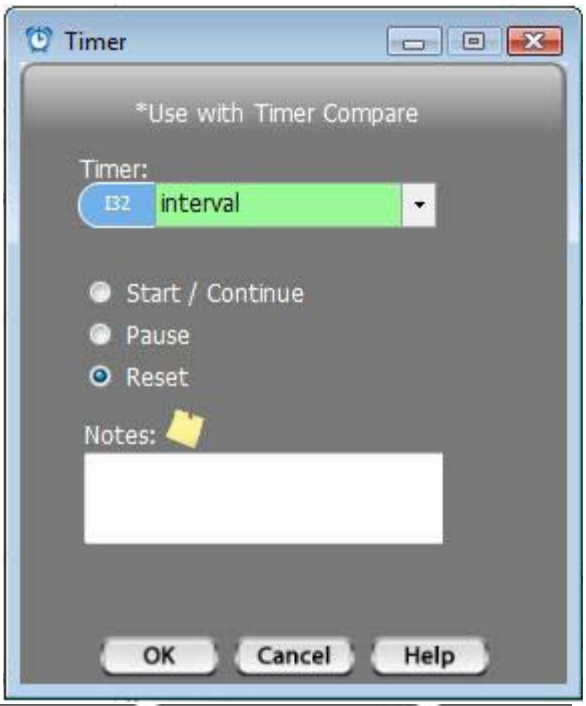
A Timer Pause block is used to stop a Timer from operating. To place a Pause, select the Pause option in the dialog box and enter the i32 tagname of the variable that holds the time value.

Once Paused, the Timer will not run again, until another Timer Start/Continue is executed.



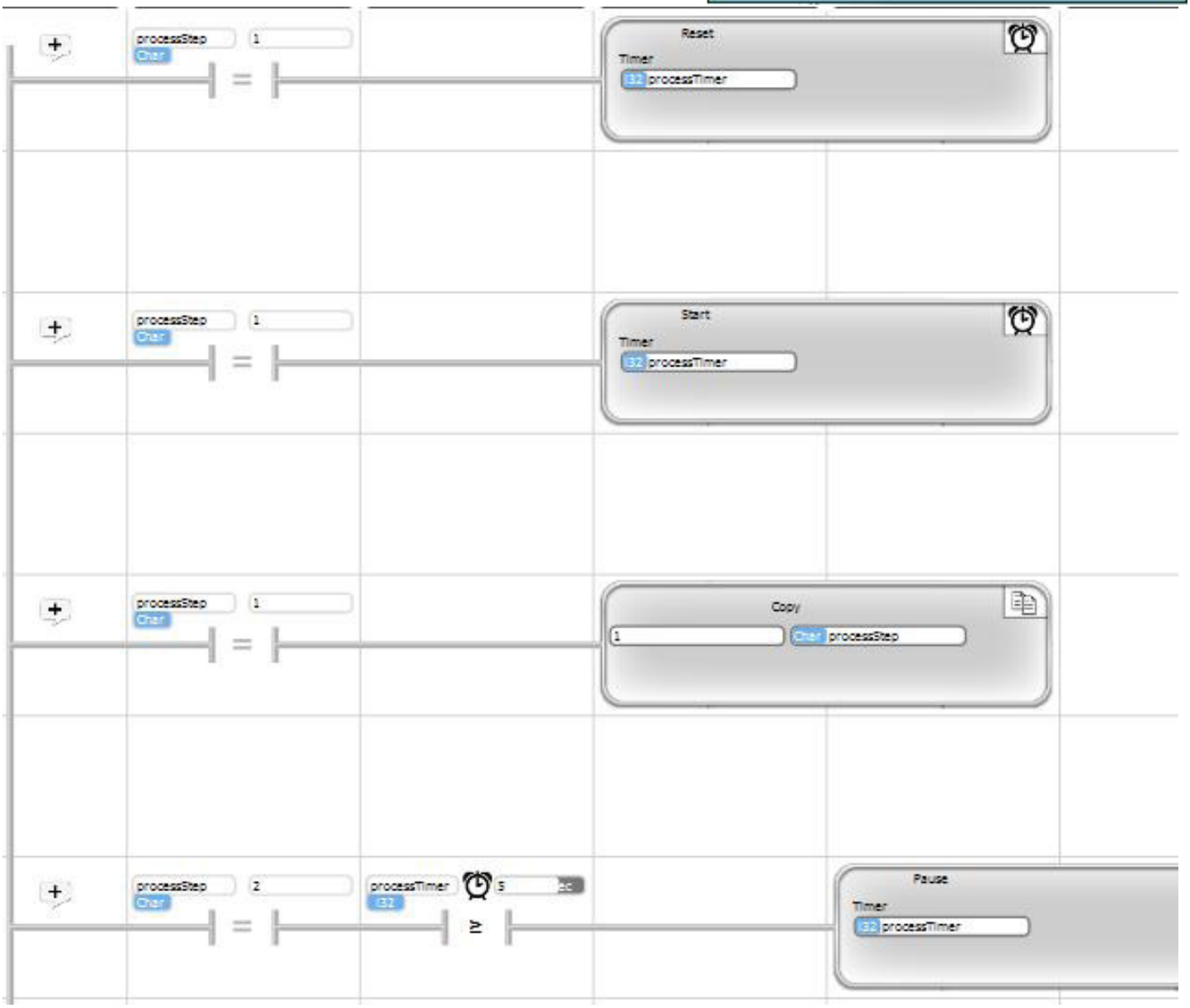
Timer Reset

Timer Reset simply resets the selected timer's value to 0. To perform a Timer Reset, place the timer block, select the 32 bit signed integer tag name or the Timer, then select the Reset option and OK.



◊ Example Timer Application

The example, below, shows a Timer being Reset and Started in processStep 1, then checked in processStep 2. If the processTimer value is greater than or equal to 5 seconds, processTimer will be paused. Obviously, in a real program, you'd likely do other operations as well.



5. Flow Chart Programming

Velocio Builder's Flow Chart programming gives a developer the option of programming in a graphical language that is efficient, easy to understand, self documenting and powerful. Those developers who are long time ladder logic professionals will likely stay with Velocio's Ladder Logic - although many will give flow charts a try. Newer developers will likely find flow chart programming even more natural and easy to use. The functionality is basically the same as our Ladder Logic. The graphics is different and easily understood.

In a lot of applications, whether Ladder Logic or Flow Chart programming is utilized is a matter than personal preference. There are some areas where Flow Charting has distinct advantages. Some of those areas are listed below.

- Test Applications : Test applications are based on pass/fail decisions, which naturally translates to flow chart decision blocks.
- State Machine Applications : Almost any control application is actually a natural state machine. State machine applications are any applications that breaks down to "if we are in this state and this happens, we do that" and then transition to the next state where the next set of "if this happens do that, or if this other happens, do a different that, then enter another state". Almost all control applications, when you break them down, are state machines.
- Applications that are specified with flow charts : In a lot of cases, specifications use flow charts to clearly define requirements. That makes it very easy to translate the specification into the application and to demonstrate to the specifier that the requirements are met.
- Applications that must be maintained by different people over a period of time. Flow charts' self documenting features make learning what the previous developer (or even the earlier you) did, a much easier process.
- Most complex applications : Flow chart logic tends to be easier to follow for complex applications.

The following is a general list of vBuilder Flow Chart functionality.

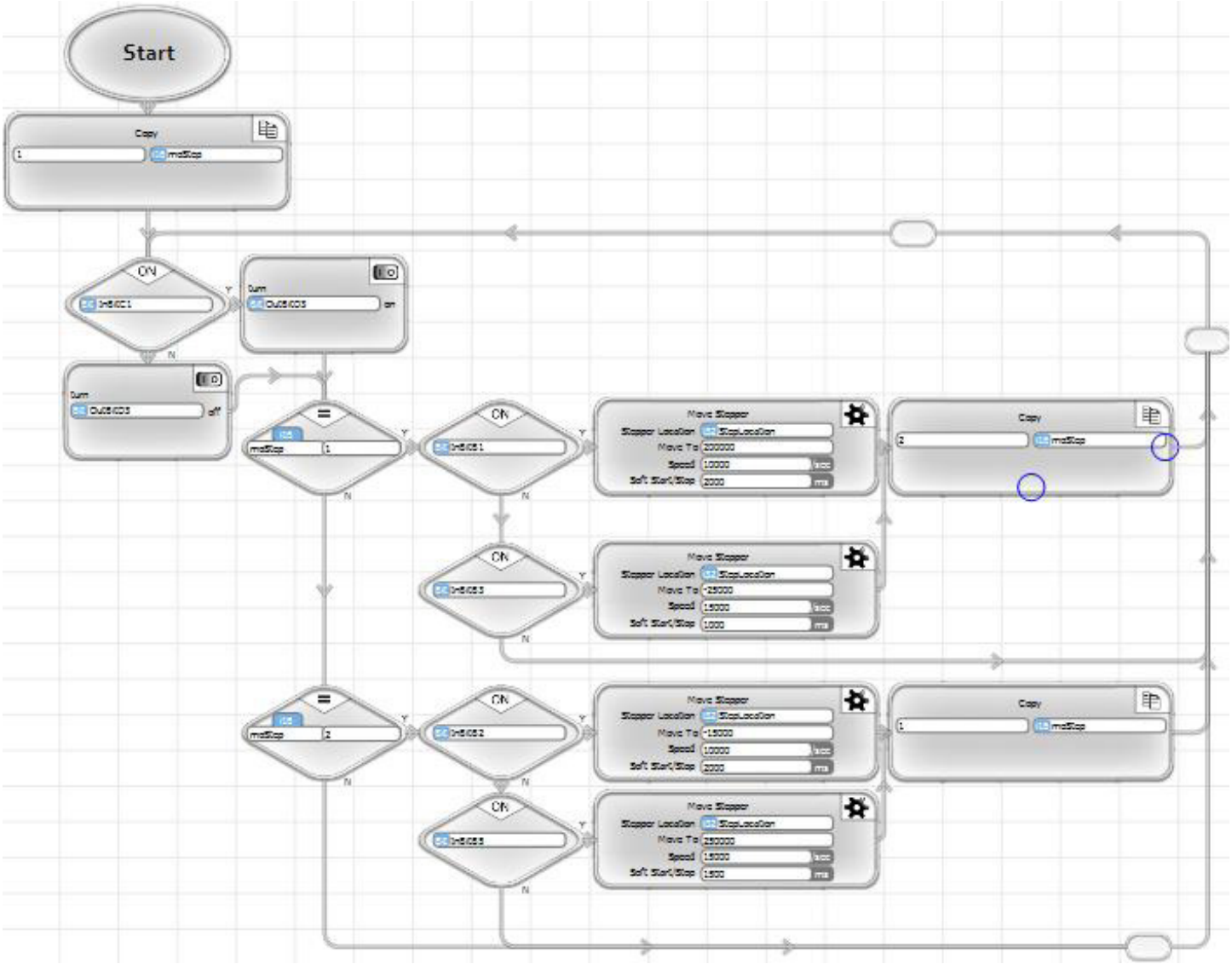
- Number formats including binary, unsigned 8 bit integer (ui8), signed and unsigned 16 bit integer (i16 & ui16), signed 32 bit integer (i32), and floating point (Float)
- Tagnames for all variables
- True Object Subroutines
- Embedded Object Subroutines
- Distributed Processing and seamless intermodule data transfer
- Program flow routing definable by programmer
- Notes for documentation
- General Decision blocks
- Numeric Comparison Decision blocks
- Output Turn On/Off
- Calculator (math) operations, including arithmetic and boolean
- Copy (general copy, bit pack and bit unpack)
- Counter (general counter function)
- Filter
- Motion in (high speed general pulse counting and quadrature pulse counting)
- Motion out (stepper motion control)
- PID
- Ramp
- Scale
- Shift/Rotate
- Statistics
- Timer

By using this functionality to compose a custom Flow Chart program, any logical program can be created quickly, easily and graphically.

After a quick snapshot of a Flow Chart program, the individual vBuilder Flow Chart program function blocks are detailed.

A View of a Flow Chart Program

The screen shot illustration shown below, is a picture of a vBuilder Flow Chart program. This illustration is provided simply to give you a feel for how a Flow Chart program can be constructed as well as how logical and easy to follow the finished program is. The particular details of this program are not important, and we won't go through it. It is simply shown to give you a quick overview of what vBuilder Flow Chart programming is all about.



Decision Blocks

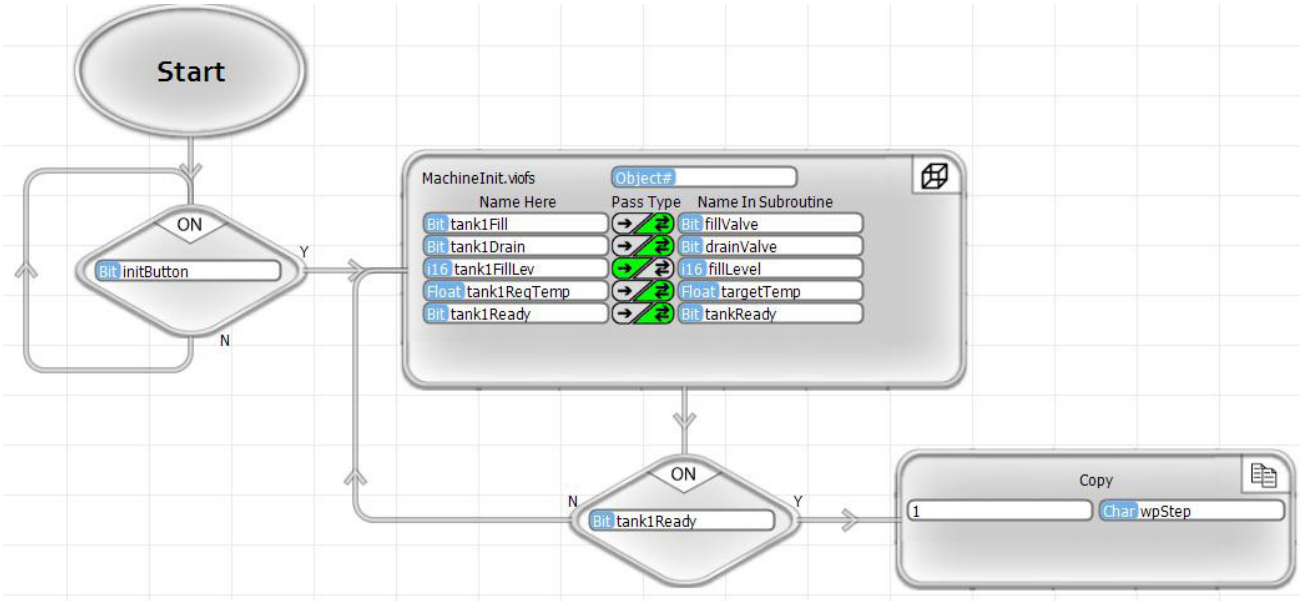
The function block icons at the top of the Toolbox are decision blocks. Decision blocks ask a question. If the answer to the decision is Yes, they exit the decision block one direction. If the answer is No, they exit a different direction.

The decision block icons are shown on the right. Decision blocks are available to check bit status, for numeric comparison and for Timer comparison. With this set of blocks any logical decision that is necessary in an application can be performed.

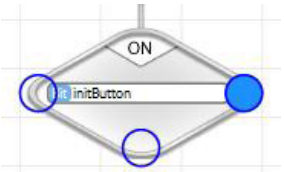
The flow chart segment, shown below, provides a general illustration of the use of decision blocks within Flow Chart programs. This program segment shows the program checking whether the initButton is ON (1), If it is, subroutine MachineInit is called. If it is not, the NO branch from the initButton ON decision block loops back to check again.

Shown after the call to subroutine MachineInit is a decision block that checks if tankReady is ON. If so, the program proceeds to copy 1 into wpStep. If tankReady is not ON, the decision block branches back to the call to MachineInit.

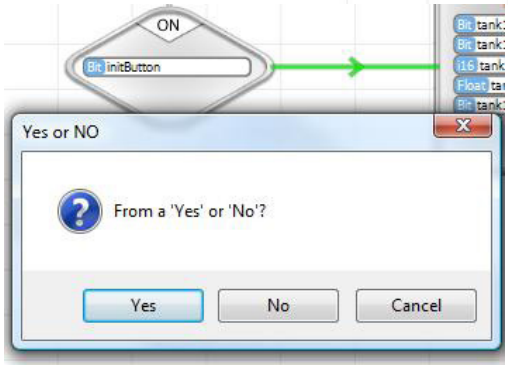
Note that looping in a program, like shown below is generally not the best way to perform control. Read the chapter on State Machine Programming to see a more efficient programming strategy.



When connecting out of a decision block, hover your cursor over the corner where you want to exit. It will turn a solid blue. Left click and continue to hold down while moving to the point you want to connect. When you release, a dialog box will pop up to ask whether this is the Yes or No connection. Click Yes or No to make the selection. A 'Y' for Yes or an 'N" for No will be displayed on the line to indicate the decision type.



For a decision block, you must connect a Yes and a No output.



ON? and OFF? Decision Blocks

ON? and OFF? decision blocks check the state of a "Bit" type variable. A bit can be associated with digital inputs or outputs, or simply be an internal bit type tag. Within vBuilder, a bit tag-named variable is considered ON, if its value is 1. If the variable has the value of 0, it is considerd OFF.

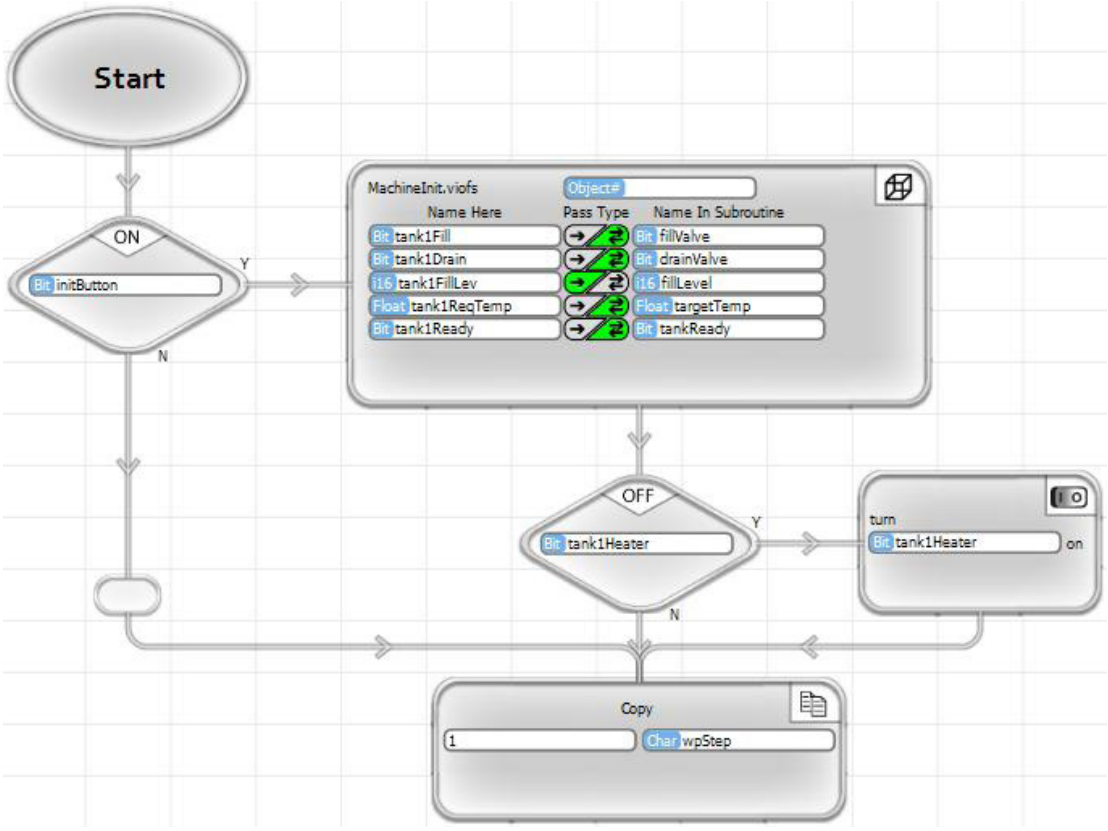
For the ON? decision block, the tagnamed variable is checked to see if it's value is 1. If it is 1, the program will flow in the Yes direction. If it is 0, program flow will proceed in the No direction.

For the OFF? decision block, the tagnamed variable is checked to see if it's value is 0. If it is 0, the program will flow in the Yes direction. If it is 1, program flow will proceed in the No direction.

◊ Example :

In this example, if the initButton is ON (1), the ON? decision block program flow will be to the MachineInit subroutine call. If initButton is not ON, the ON? decision block program flow will be to the Copy block on the bottom of the chart.

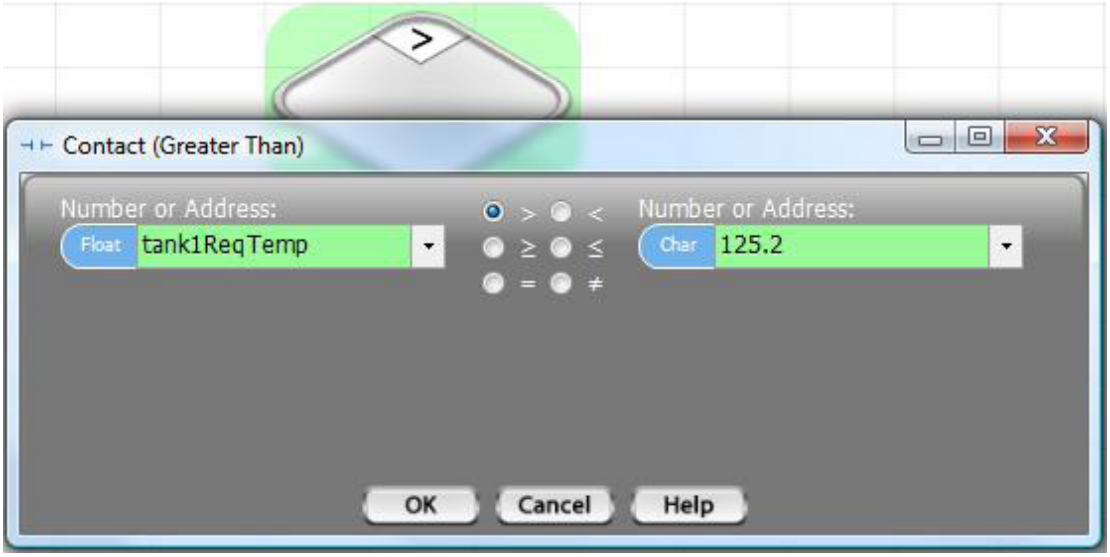
If program flow gets to the OFF? decision block, if tank1Heater is OFF (0), program flow will be to the the block that turns on the tank1Heater. If tank1Heater is not OFF (1), program flow will be to the copy block.



Numeric Comparison Decision Blocks

The decision block icons shown on the left are for numeric comparison decisions. The Yes/No routing from the decision block is determine by the result of the defined numeric comparison. There are comparison decision blocks for greater than, less than, greater than or equal to, less than or equal to, equal to and not equal to. In each case, if the numeric comparison is true, the flow routing is in the Yes direction. If it is false, the program execution will flow in the No direction.

When you place any numeric comparison decision block in a Flow Chart, a dialog box will pop up. This box is used to define the comparison operation associated with the decision. For any comparison, either select the values to compare from tagnames available in the drop down boxes, or type in a numeric value. Make sure the comparison operation that you want is properly indicated by the radio button selection. If it is not, select the one you want. When you are done defining the comparison, click OK.



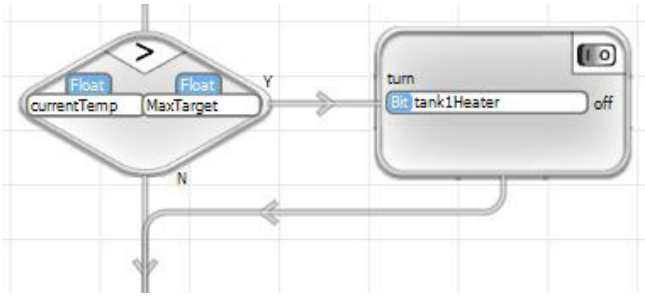
It is important to be careful when comparing floating point numbers to integer numbers. Floating point numbers are stored in IEEE format (which is what is used in almost all computer languages). This format incorporates 24 bits of numeric precision and an 8 bit exponent. They are very accurate, but not always exactly precise. A number such as 375.0 might actually be 375.00000001 or 374.9999999999. In such a case a comparison to integer 375 for equality would result in a not true result. When comparing floating point numbers to integers, greater than or less than comparisons are generally preferred.

Greater Than Comparison Decison Block

The programming icon for a Greater Than comparison is shown on the left. With a Greater Than comparison, a determination is made as to whether the first value is greater than the second value. If the first is greater than the second, the Yes branch will be taken. If it is not, the No branch will be taken.

◊ Example

In this example, if the currentTemp is Greater Than the MaxTarget, the heater will be turned off.

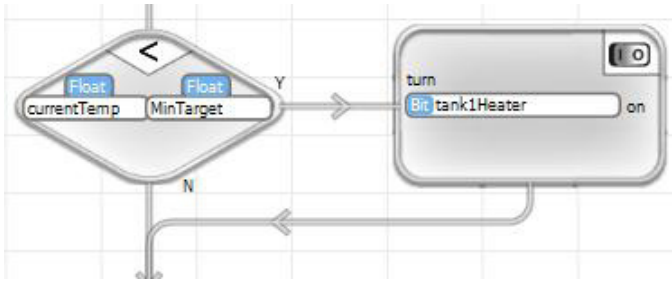


Less Than Comparison Decision Block

The programming icon for the Less Than comparison is shown on the left. With a Less Than comparison, a determination is made as to whether the first value is less than the second value. If the first is less than the second, the Yes branch will be taken. If it is not, the No branch will be taken.

◊ Example

In this example, if the currentTemp is Less Than the MinTarget, the heater will be turned on.

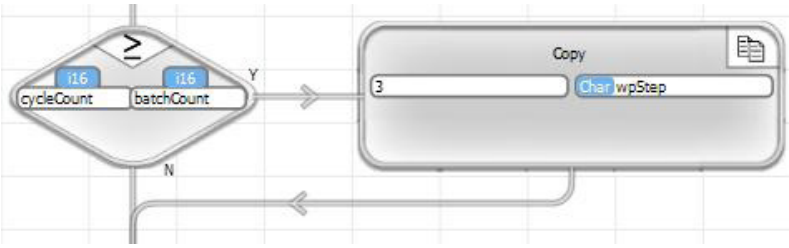


Greater Than or Equal Comparison Decision Block

The programming icon for the Greater Than or Equal to comparison is shown on the left. With the Greater Than or Equal to comparison, a determination is made as to whether the first value is greater than or equal to the second value. If the first is greater than or equal to the second, the Yes branch will be taken. If it is not, the No branch will be taken.

◊ Example

In this example, if the cycleCount is Greater Than or Equal to the batchTarget, wpStep will be set to 3.

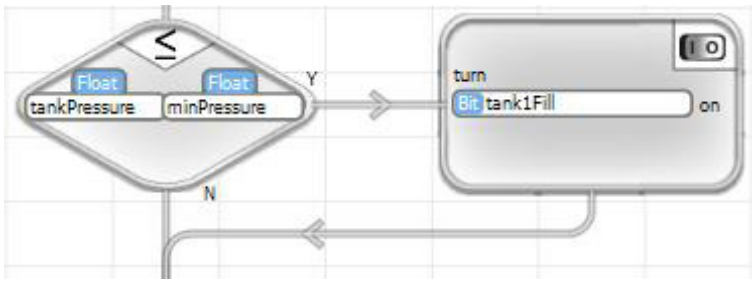


Less Than or Equal to Comparison Decision Block

The programming icon for the Less Than or Equal to comparison is shown on the left. With a Less Than or Equal to comparison, a determination is made as to whether the first value is less than or equal to the second value. If the first is less than or equal to the second, the Yes branch will be taken. If it is not, the No branch will be taken.

◊ Example

In this example, if the tankPressure is Less Than or Equal to the minPressure, the tank1Fill valve will be turned on.

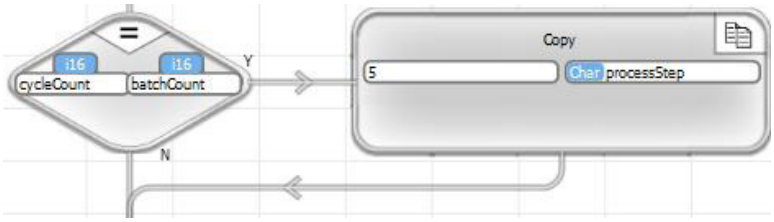


Equal to Comparison Decision Block

The programming icon for the Equal to comparison is shown on the left. With an Equal to comparison, a determination is made as to whether the first value is equal to the second value. If the two numbers are equal, the Yes branch will be taken. If they are not equal, the No branch will be taken.

◊ Example

In this example, if cycleCount is Equal to batchCount, the number 5 will be Copied into processStep.

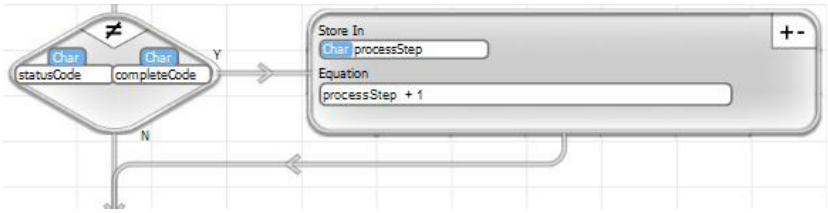


Not Equal to Comparison Decision Block

The programming icon for the Not Equal to comparison is shown on the left. With a Not Equal to comparison, a determination is made as to whether the first value is not equal to the second value. If the two numbers are not equal, the Yes branch will be taken. If they are equal, the No branch will be taken.

◊ Example

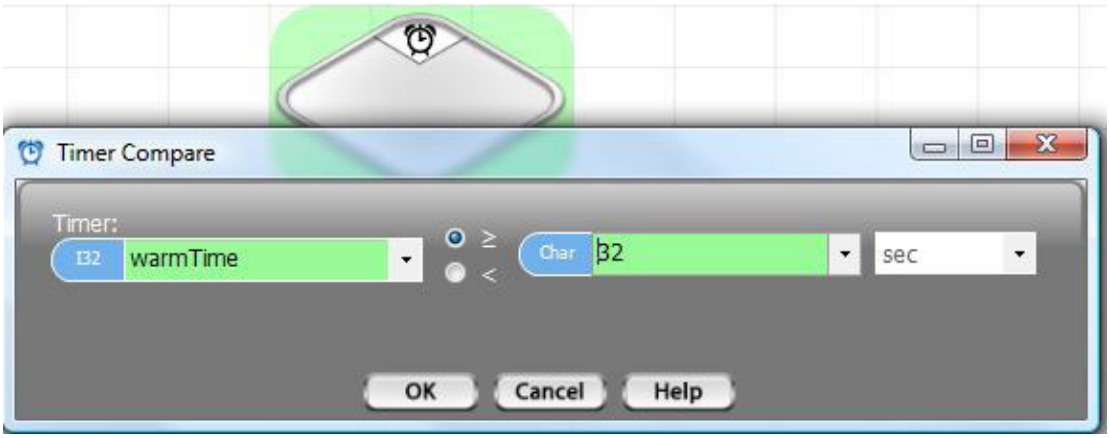
In this example, if statusCode is Not Equal to completeCode, processStep will be incremented.



Timer Comparison Decision Block

The icon for the Timer Comparison Decision Block is shown on the left. A timer comparison compares a timer value to a comparison value.

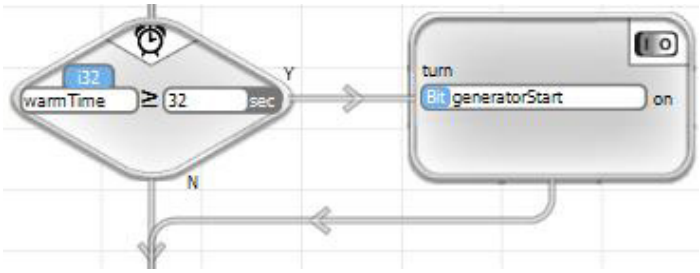
Timers use 32 bit integer (i32) numbers. An i32 used as a timer will keep time in 100ths of a second. It can be compared to values in milliseconds (1/1000s of a second), seconds, minutes, hours, or days. When you place a Timer Compare decision block, a dialog box like that shown below, will pop up. In the dialog box, you must select the tagname of the timer in the left hand Timer box. The radio button selections to the right of the Timer box allows you to select whether to check if the Timer is greater than or equal to, or less than the value on the left. In the selection box on the left, you can either enter a number, or select a tagnamed variable with the number you want to compare to. The last selection you must make is the time units the comparison value represents. You can select time units from the drop down selection list.



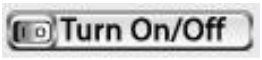
If the timer comparison is true, the Yes branch will be taken. If it is not, the No branch will be taken.

◊ Example

In this example, if warmTime is greater than 32 seconds, generatorStart will be turned on.

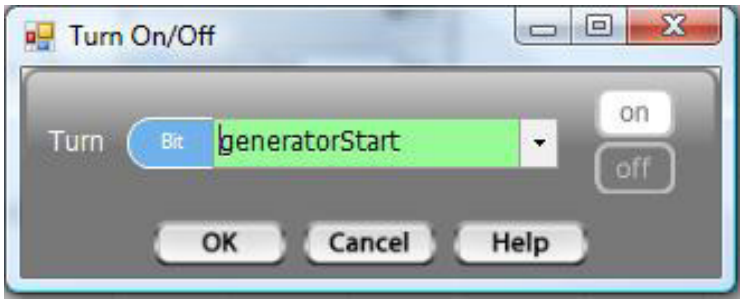


Turn On/Off



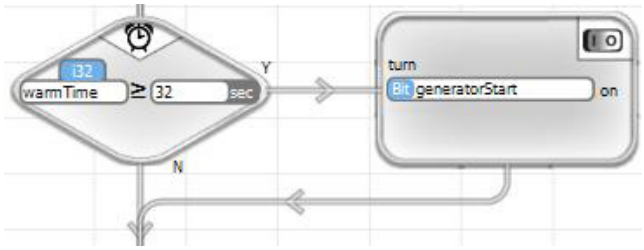
Turn On and Turn Off blocks are the simplest action blocks used in vBuilder Flow Chart programming. Turn On and Turn Off blocks can be used to turn a digital output on or off. They can also be used to set a bit type tagname to a '0' or a '1', for logical operations.

When you place a Turn On or Turn Off block in a program, a dialog box, shown below, will pop up. Select the bit that you want to Turn On (1), or Turn Off (0), and select the On or Off option, then click OK.

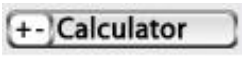


◊ Example

In this example, if warmTimer is greater than 32 seconds, generatorStart will be turned on.



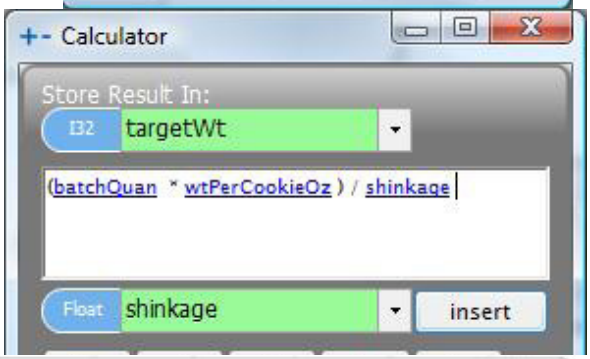
Calculator



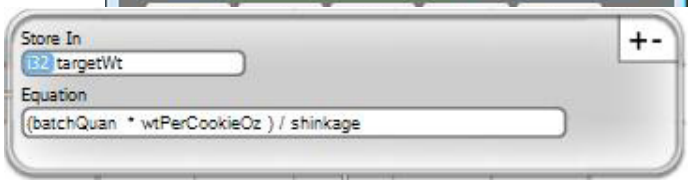
The calculator is the math function of vBuilder. The calculator icon is shown on the left. With a calculator program block you can perform any type of arithmetic or boolean operation. Equations can range from simple to bracketed complex operations.

When you place a calculator block, a dialog box will pop up, as shown on the right.

With the dialog box, first choose the tag-named variable to use to store the result, from the drop down list at the top of the dialog box. Next, enter your equation. You can pick tagnamed variables using the tag picker below the equation window, then press insert. Select operators and brackets to place in the equation as needed. You can create as complex of an equation as you want.



Once you've entered the equation that you want, click OK. The block will show up in the rung like the illustration to the right.

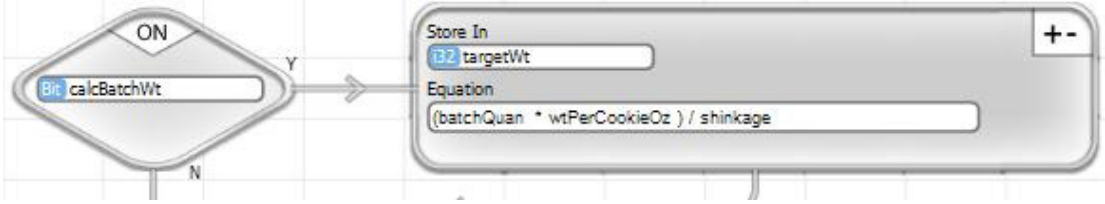


Operators :

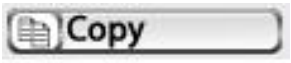
+ - * / : Math operators for addition, subtraction, multiplication and division
And &, Or |, Xor + : Boolean And, Or and Exclusive Or operators
( ) : Segmentation brackets
[] : Braces used to contain array index
0x : Hexadecimal number
A, B, C, D, E and F : Hexadecimal 10, 11, 12, 13, 14, and 15

◊ Example

In the example, if calcBatchWt is On (1), the targetWt will be calculated according to the formula shown.



Copy

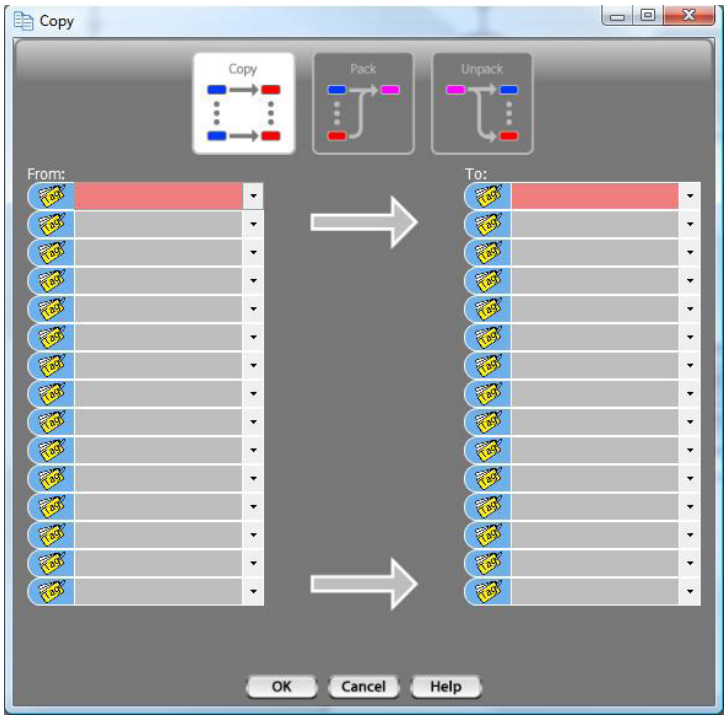


The icon for Copy operations is shown on the left. There are actually three different Copy function blocks available : Copy Value, Copy Pack and Copy Unpack.

When you place a Copy block in a Flow Chart program, a dialog box, as shown on the right, will pop up.

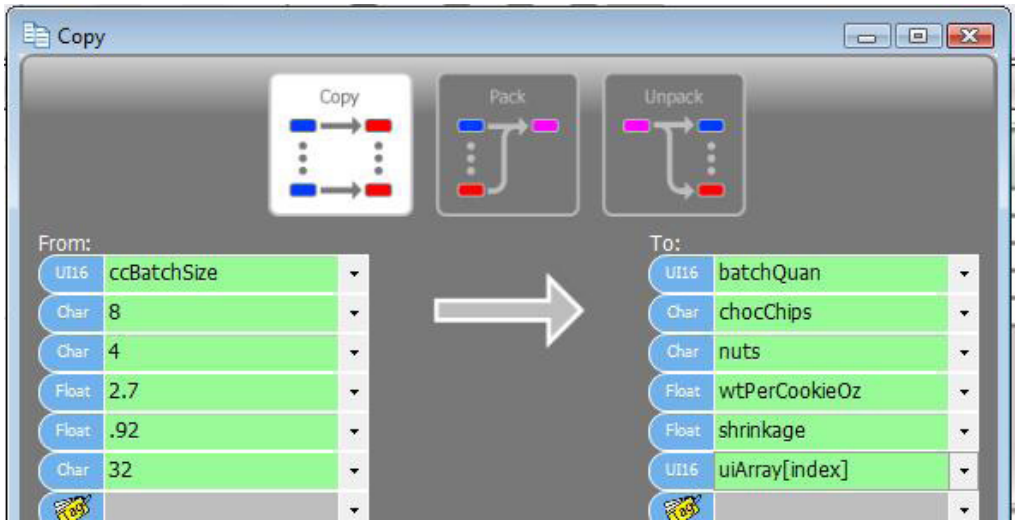
The selections across the top of the dialog box are illustrated selection buttons that enable the selection of the appropriate type of copy. As the selection buttons show, the general Copy allows you to copy up to 16 tagnamed variables or constant values to a like number of other selected tagnamed variables. The Copy Pack provides a function for copying a list of bits to a ui16 integer. The Copy Unpack does the opposite - copy individual bits from a ui16 integer to bit tagnamed variables.

When you select the different Copy types, the copy details, below the selection buttons change accordingly.



Copy Value

The Copy Value function provides the capability to copy up to 16 values or tagnamed variables to a like number of tagnamed variables. Select the "Copy" option in the Copy dialog box and simply select a number or a tagnamed variable to copy from in each box on the left and a corresponding tagnamed variable to copy to in the associated box on the right.



The Copy Value function will convert data from one type to another, if the data types don't match.

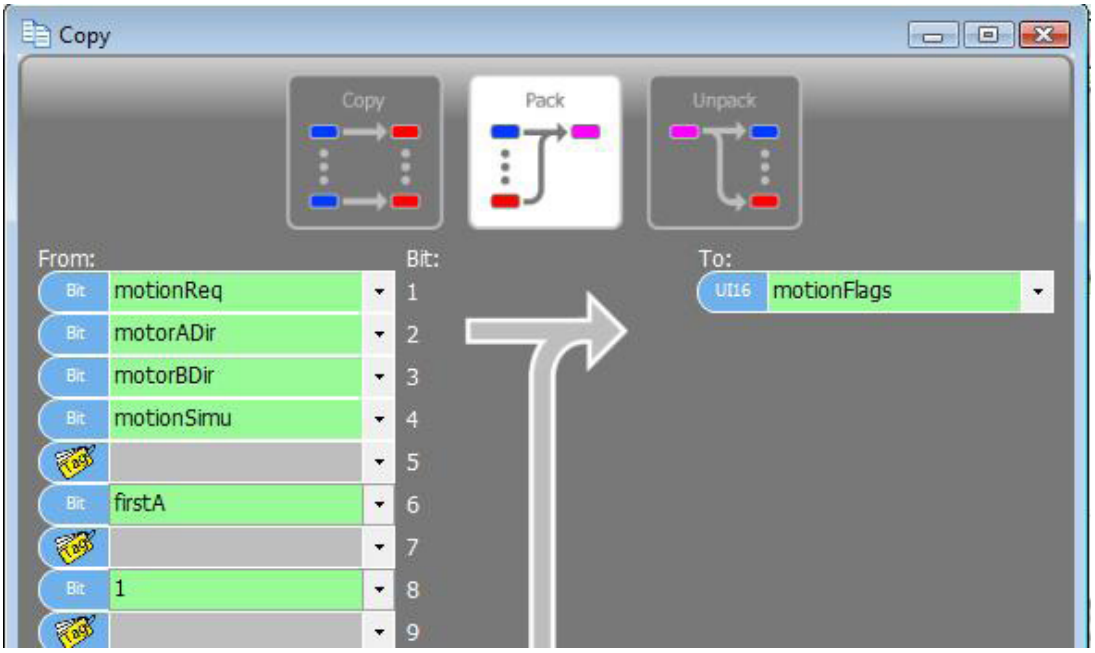
The illustration, above, includes a copy into an indexed variable. When you select an indexed variable, it will appear with the index braces []. You must select inside the braces and type the index number, or type the tagname of a ui16 variable used for the index. The case of typing a variable tagname is shown above.

◊ Example

In the example shown below, if initRecipe is ON (1), the list of six variables and constants will be copied to the corresponding list of six tagnamed variables.

Copy Pack

The Copy Pack function provides the capability to copy up to 16 tagnamed or constant bits to a ui16 variable. Select the "Pack" option in the Copy dialog box and simply select a 0 or 1 or a tagnamed bit to copy to the bit position in each box on the left and a corresponding tagnamed ui16 variable to copy to in the associated box on the right. Bits of the 'To' variable in positions not selected will remain unchanged. This function can be used to 'mask in' selected bit locations within the ui16 variable.



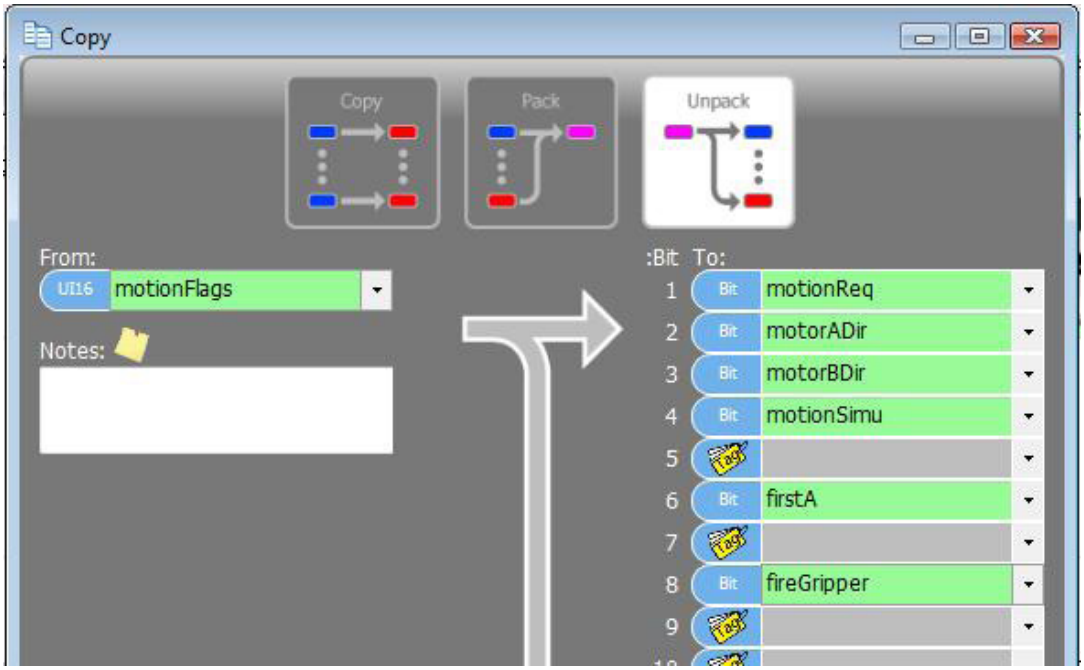
◊ Example

In the example shown below, if motionEnabled is ON (1), the bits identified on the left of the Copy Pack block will be copied to the bit positions identified next to the bit tag name or value in the ui16 tagnamed variable listed on the right.

Copy Unpack

The Copy Unpack function provides the capability to extract individual bits from a ui16 variable. Select the "Unpack" option in the Copy dialog box. Unpack extracts bit data from particular bit positions in the From ui16 to the tagnamed bits assigned to each bit position. In the dialog box, the From tag is a ui16 tagnamed variable that you select. Bits of the 'To' variables are extracted from the bit position s identified in the Bit To list.

The source ui16 (From variable) is not changed. There is no requirement that all bits be unpacked. With a Copy Unpack, selected bits can be extracted at will.



◊ Example

In the example shown below, if getMotionFlags is ON (1), the bits identified on the right of the Copy Unpack block will be extracted from the bit positions identified next to the bit tag names from the ui16 variable motionFlags.

Counter

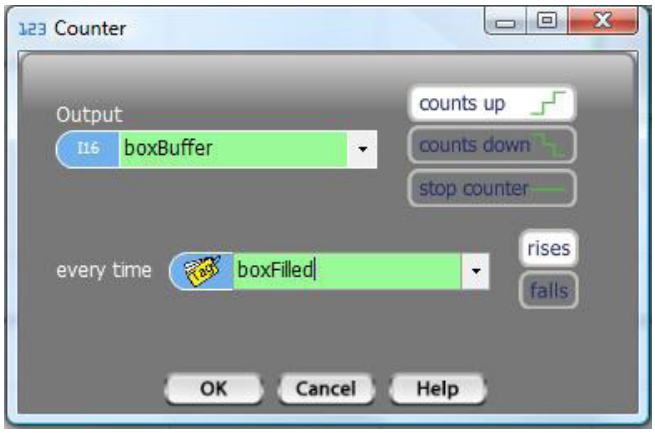
There are both software and high speed hardware counter functions available in vBuilder. The general Counter is a software counter and is described in the following paragraphs. High Speed pulse counter and quadrature counter functions are MotionIn functions and are described in the MotionIn function description.

Counters are background tasks. Once a counter is set up through a Counter block, it will continue to operate, behind the scenes, until it is stopped through a Stop Counter block execution. The counters work in the background, updating once, just prior to the execution of each program logic pass. Once you have started an Up or Down Counter, your program logic does not need to execute it again on every program pass.

Note : A counter operation is defined by its Output tag name. Therefore there can be only one count operation active for a given Output tagname.

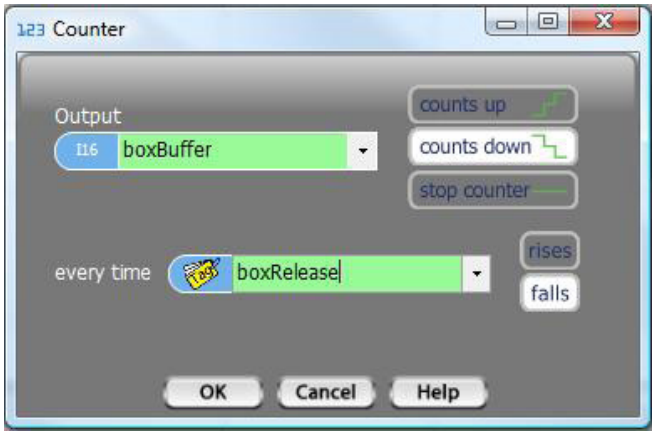
Up Counter

The Up Counter increments a count by one, each time a tagnamed bit transitions (checked just prior to each logic pass) in the direction (either rises or falls) that is selected. In other words, if you want to place an Up Counter block that increments each time a tagnamed bit changes from 0 to 1, select Counter, then select "counts up" and "rises", as well as the Output (the tagnamed variable that keeps the count) and the tag name of the bit that is checked "every time" for the rising transition.



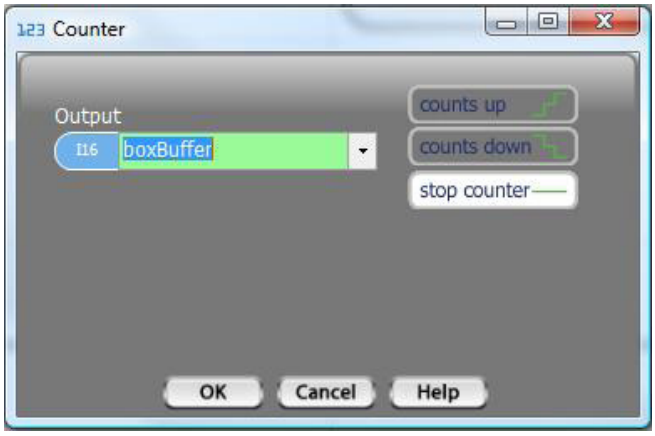
Down Counter

The Down Counter decrements a count by one, each time a tagnamed bit transitions (checked just prior to each logic pass) in the direction (either rises or falls) that is selected. In other words, if you want to place a Down Counter block that decrements each time a tagnamed bit changes from 1 to 0, select Counter, then select "counts down" and "falls", as well as the Output (the tagnamed variable that keeps the count) and the tag name of the bit that is checked "every time" for the falling transition.



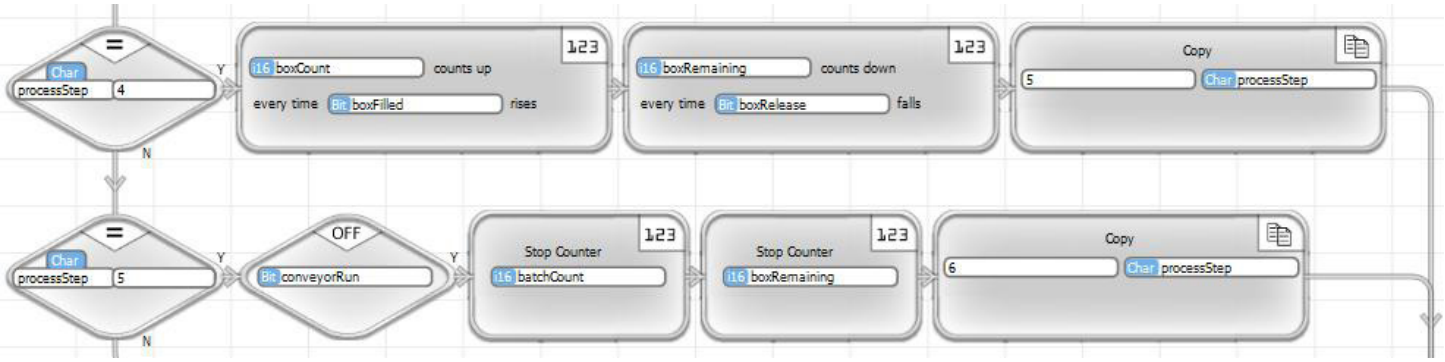
Counter Stop

The Counter Stop simply stops the selected counter from counting. The count value will not change. To place a Counter Stop block, select "Counter", then "stop counter" and select the tagname of the counter that you want to stop. Then select 'OK

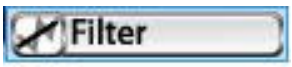


◊ Counter Example

The example, shown below, shows an application with two counters associated with a breakfast cereal fill line. The fill line has a conveyor that brings empty boxes to a fill chute, where each box is filled, then moved to a buffer area. In the buffer area, boxes are buffered, while being picked up, individually, and placed in shipping cartons. This program snippet shows the boxCount incrementing, each time a box is filled. A second counter, boxRemaining is the count down of the remaining box production scheduled for this shift. boxRemaining is decremented each time a box is picked up and placed in a carton. Both of these counters are started on processStep 4. processStep is then immediately incremented to 5. In processStep 5, the conveyorRun tag bit is monitored. When conveyorRun becomes OFF (0), both counters will be stopped and processStep will be set to 6 .



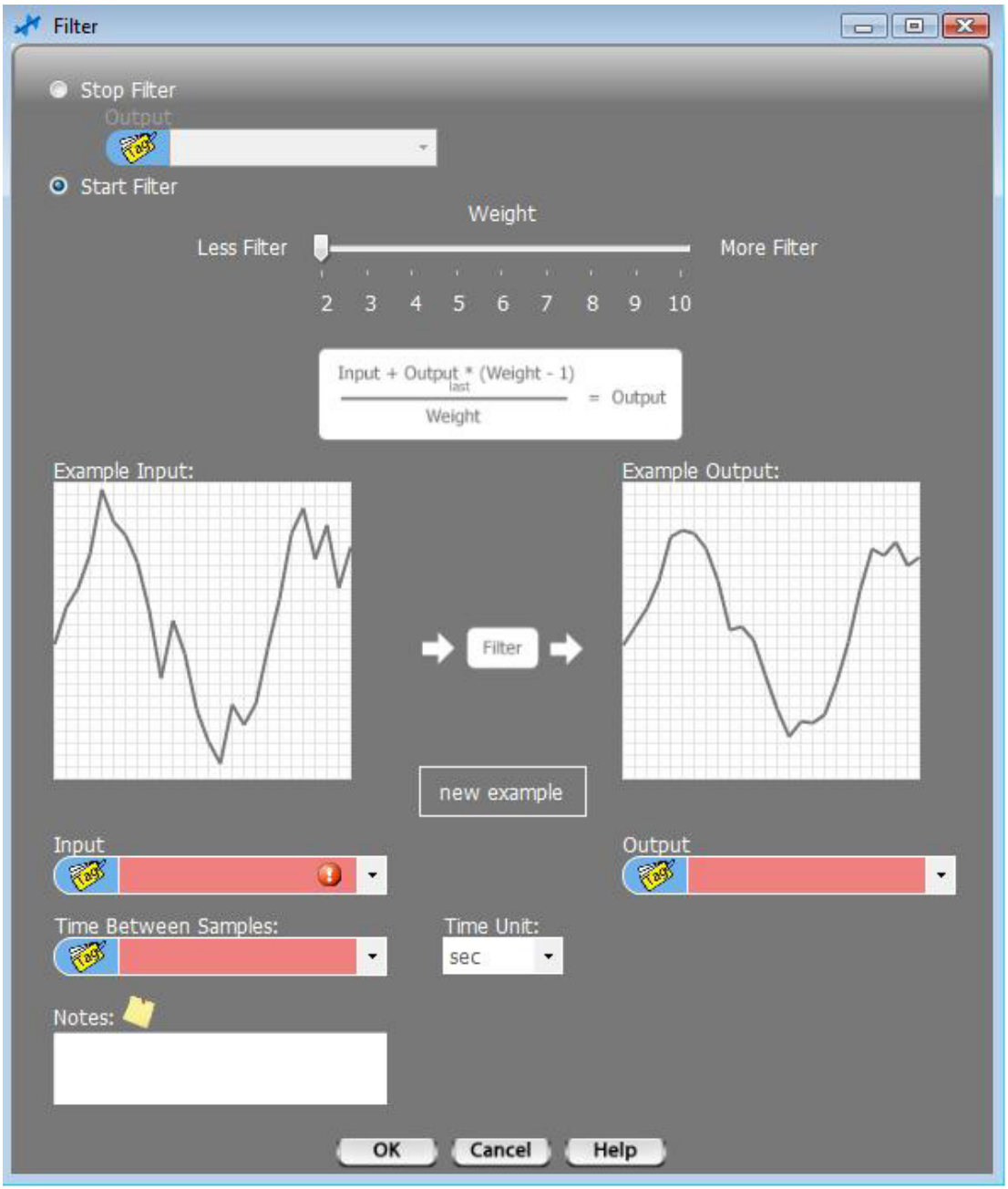
Filter



The program icon for the Filter block is shown on the right. The Filter function is a relatively simple function. It can be used in instances where you are more interested in an average value over a period of time than any particular instantaneous value.

The Filter function is a background task function. You simply tell it what to filter at what weight and time interval and the Filtering will happen behind the scene automatically. All you have to do, once you've set up the Filter is use the filtered result value whenever you need it.

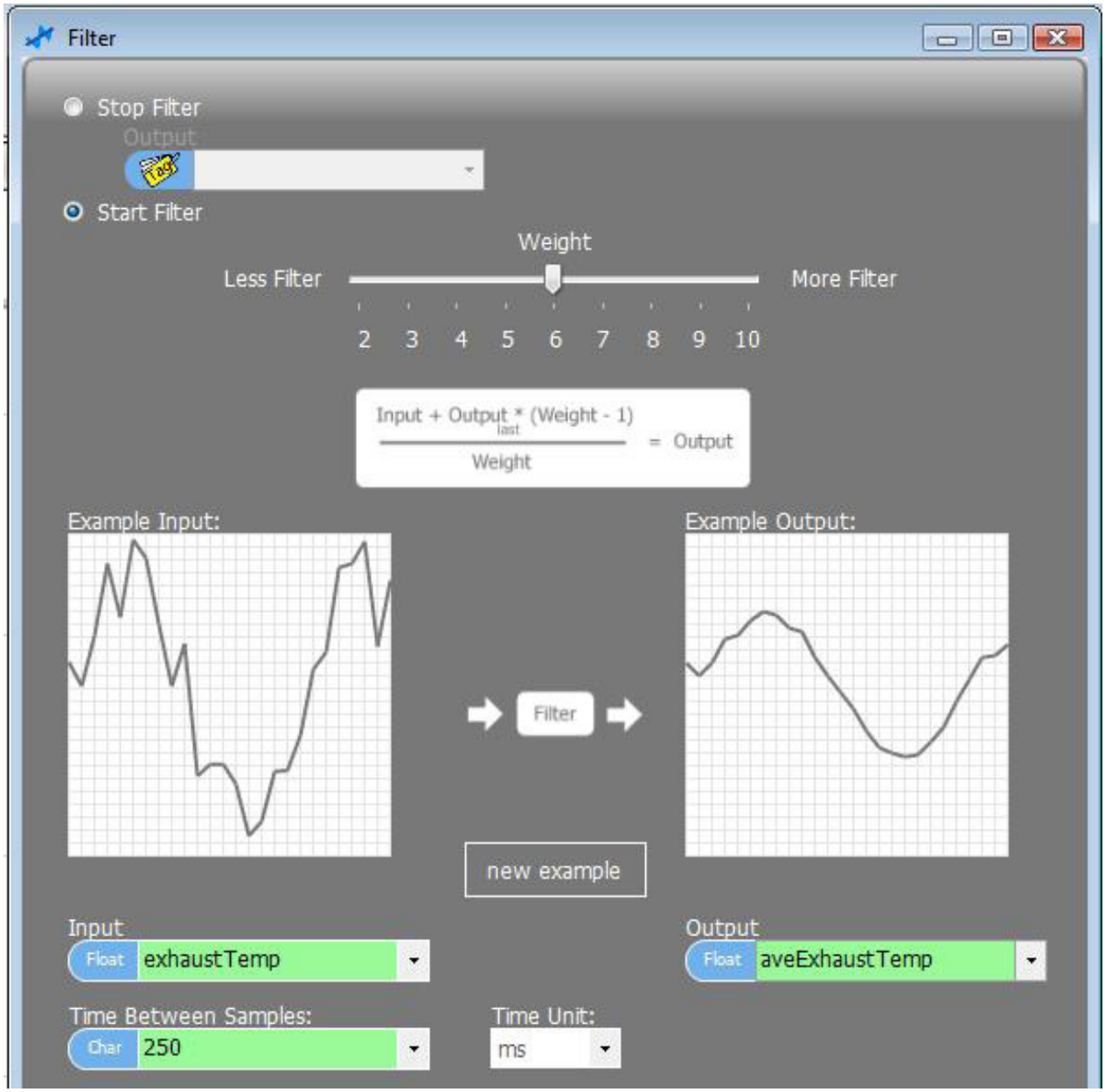
When you place a Filter block a dialog box will pop up. The box look like this :



To get a sense of what Filter does, right click and hold the slider shown next to "Less Filter" and slide it across. The live illustration below the slider shows that with an Example input shown on the left, the resulting output values will be like that shown on the right. Every time you click on the New Example button, you'll get a new example illustration.

To Start a Filter, select Start Filter, select the tagname of the variable that you want to filter, as Input, select the tagname of the variable where you want the filtered result to be placed, as Output, select a sample rate (either a constant value, or a tagnamed integer variable) and the time units, and select the weight by sliding the Weight bar. Then click OK.

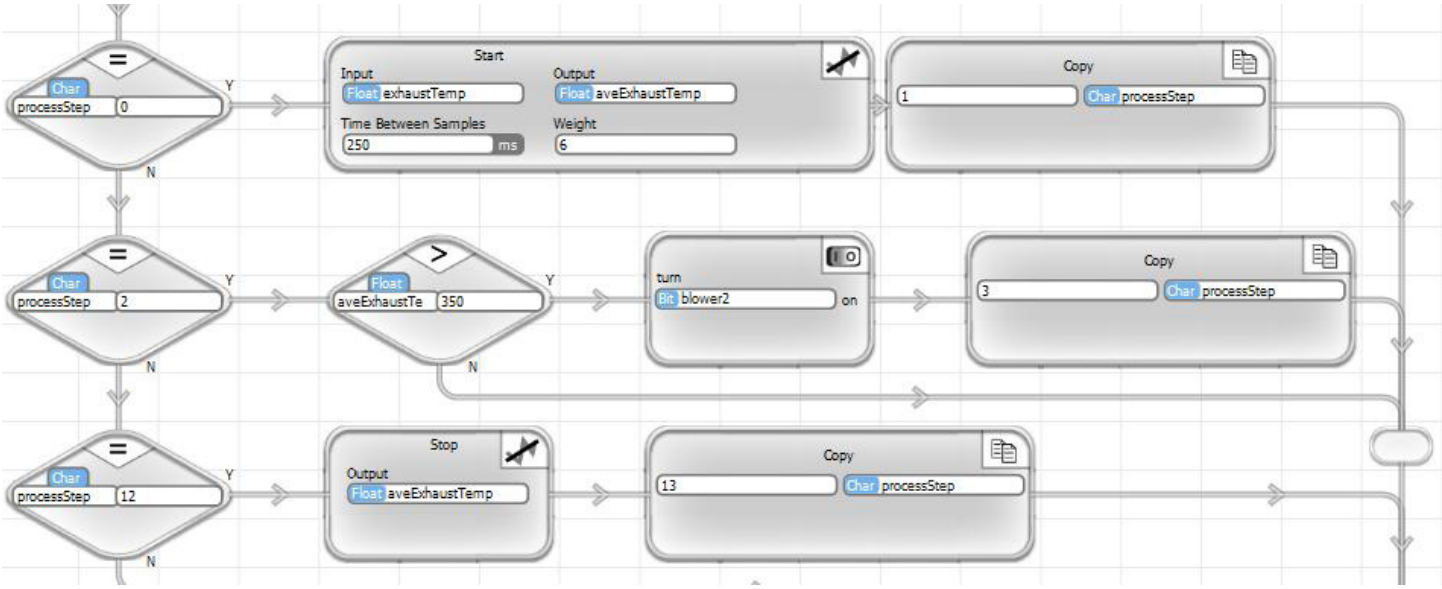
Once you've started a Filter, it will continue to operate in the background until you stop it.



To Stop a Filter, place a Filter block and select Stop Filter and select the filtered Output value's tag name. Click OK.

◊ Example

The example, shown below, shows a Filter started in processStep 0 (probably initialization), the filtered value used during processStep 2, and stopped in processStep 12 (probably shut down).

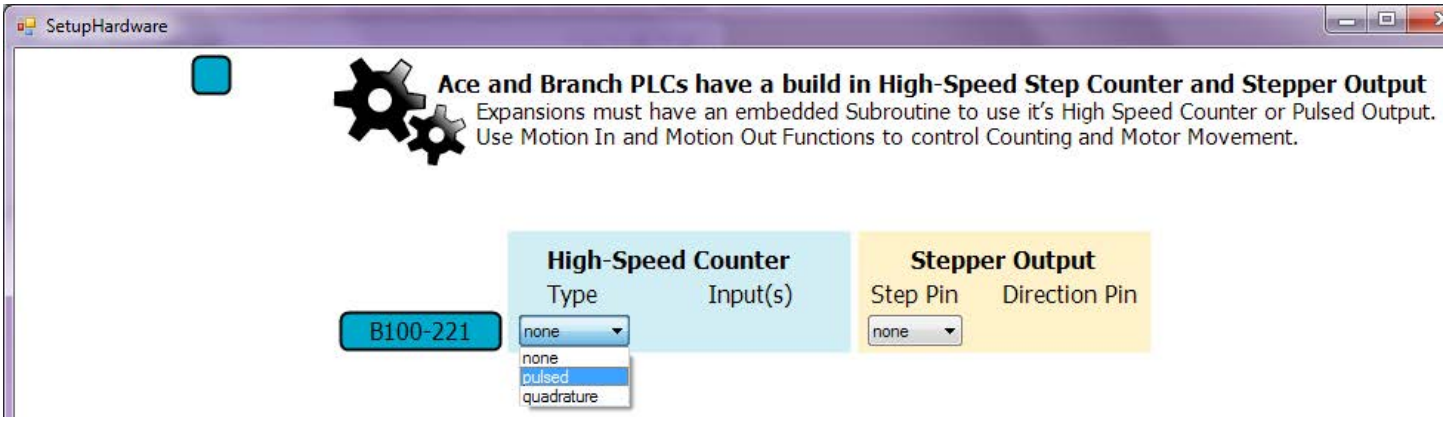


MotionIn

MotionIn blocks are used for high speed pulse counter inputs. There can be one high speed pulse counter per Ace, Branch or Expansion unit. The high speed counter can be configured for simple high speed pulse counting, or quadrature pulse counting. High speed pulse counting is a background task, which, once started, will continue until a counter stop block is executed.

The high speed pulse counters in each unit are specified for typical operation up to 100KHz. In reality, general high speed pulse inputs will operate at up to 250KHz. Quadrature inputs will operate at up to 200KHz. The general high speed pulse counter consumes approximately 1.6% of total processor time (for the particular PLC unit that it resides in) for every 10KHz pulse rate, while the Quadrature input consumes approximately 2.8% per 10KHz. As you increase the pulse rate, the CPU usage increases proportionately. Lower pulse rates consume proportionately less time. Whether, and at what point that might become an issue is application dependent. Up to 100KHz, it is hard to imagine an issue with most applications.

To implement high speed pulse counting, you must define a dedicated input pin (pulse) or pins (quadrature) during the hardware setup. When you are setting up your hardware from the "Setup Hardware" icon near the upper left corner, the following screen will pop up during the process.



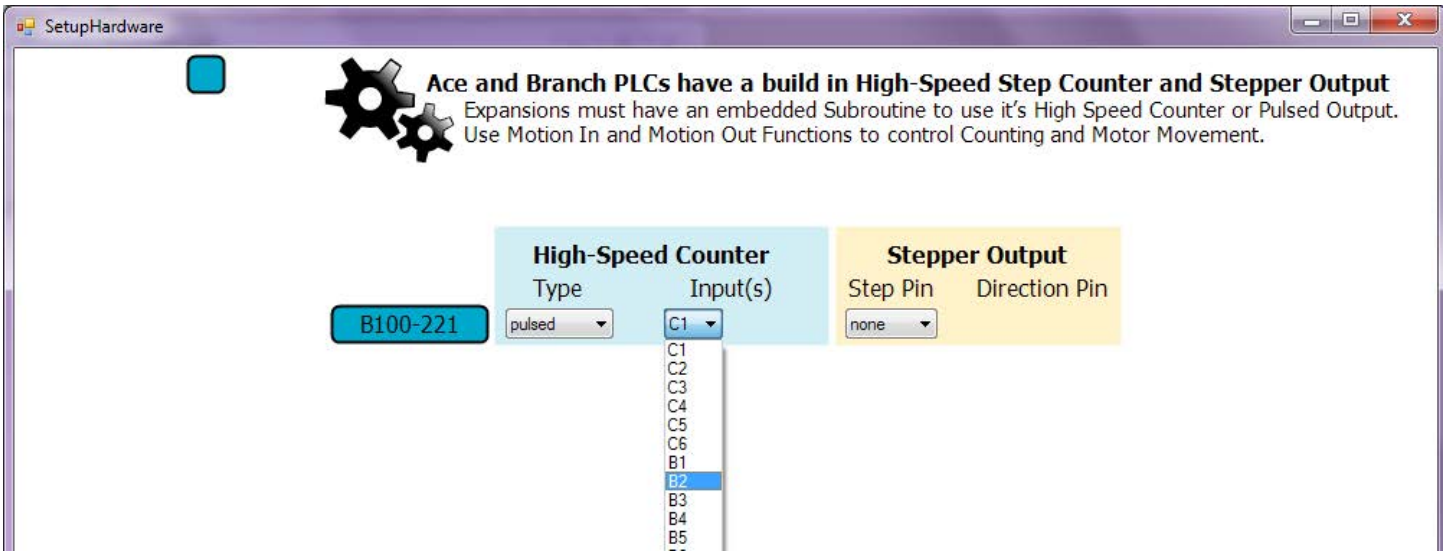
The instructions on the screen are pretty self explanatory. The screen allows you to set up for a high speed pulse counter, stepper motor output or both. To configure for a high speed pulse counter, first select the type that you want (pulse or quadrature) from the drop down selection list shown above.

MotionIn function blocks can only be used for inputs that reside on the PLC unit that contains the program. In other words, MotionIn for the main program can only be applied to digital inputs on the main PLC unit (Ace or Branch). For MotionIn application in Expansion units, an embedded object program must reside in that unit. MotionIn blocks can be utilized in the embedded object subroutine and use digital inputs that are local to the Expansion unit that contains the embedded object.

High Speed Pulse Input

A High Speed Pulse input counter will count a single input pulse train. It can be set to count up or down, and count on either a rising or falling edge transition. It is the hardware equivalent of the general Counter function. It can operate at much higher frequencies than the general Counter. It is restricted to counting transitions on a dedicated digital input signal. One high speed pulse counter can be implemented per PLC module (Ace, Branch or Expansion), so up to 15 can be implemented per application.

During hardware setup, the following screen will pop up, with high speed counter and stepper motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure a high speed pulse counter for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expanstion unit), select the Type as "pulsed" and select the particular digital input to use for your count, as shown below.



When entering your program, select the MotionIn icon and place it in the flow chart. A dialog box, like the one shown below, will pop up. Using the dialog box, you can select to count up or down, or stop the count. When selecting count up or down, you can select whether to count on the rising or falling edge of the input signal.

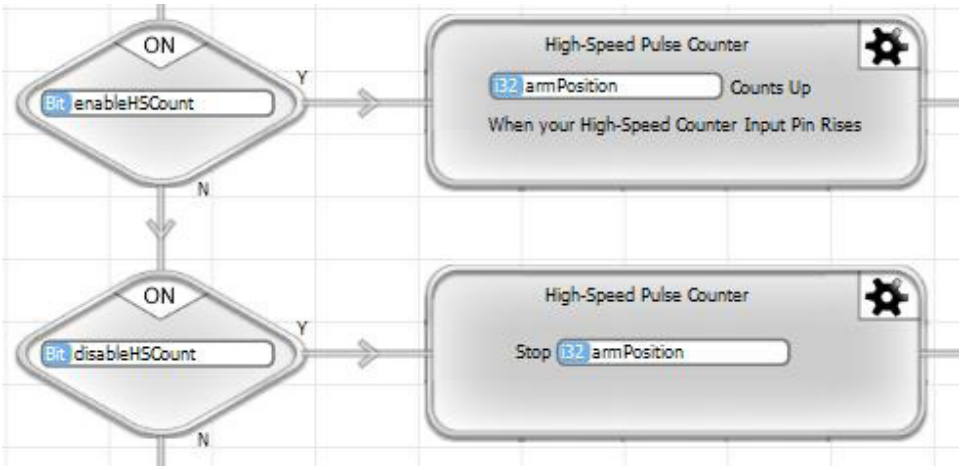
When a high speed pulse counter is started, it will continue to operate in the background until a "stop counter" block is executed.

The tagnamed variable that you choose for Count Value must be an i32.

Remember the limit of one high speed counter per main PLC or embedded object.

◊ Example

The example below shows the when enableHScount is ON (1), the high speed pulse counter for armPosition is enabled to Count up on each rising transition of the input pin configured for high speed pulse counter input. It will continue to count until disableHScount is ON (1). When that happens. the high speed counter will be Stopped.



Quadrature Pulse Input

Quadrature encoders are used to keep track of position in many motion systems. With the combination of two pulse inputs, both the position change and the direction of change can be determined. Quadrature encoders output two high speed pulse signals, known as the A and B phase pulses. Clockwise and counterclockwise rotation is detected as a sequence of pulse signal changes, as listed below.

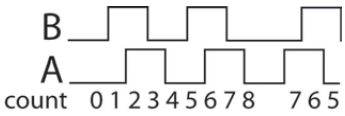
Clockwise rotation

Phase	A	B
1	0	0
2	0	1
3	1	1
4	1	0

Counterclockwise rotation

Phase	A	B
1	1	0
2	1	1
3	0	1
4	0	0

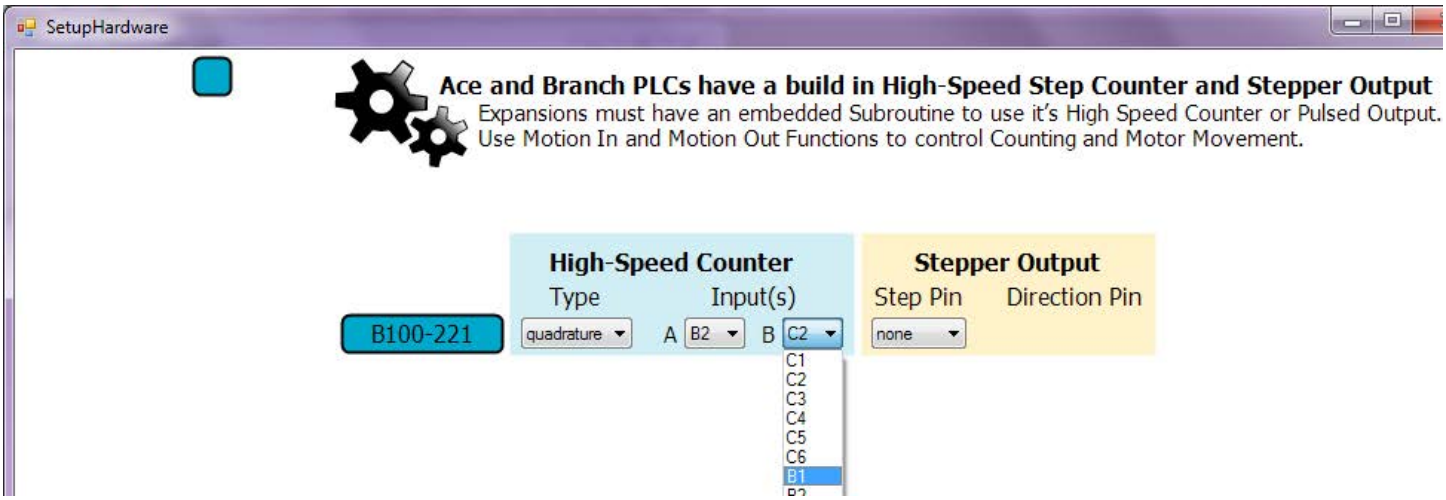
An example of a pulse train and resulting count is shown on the right.



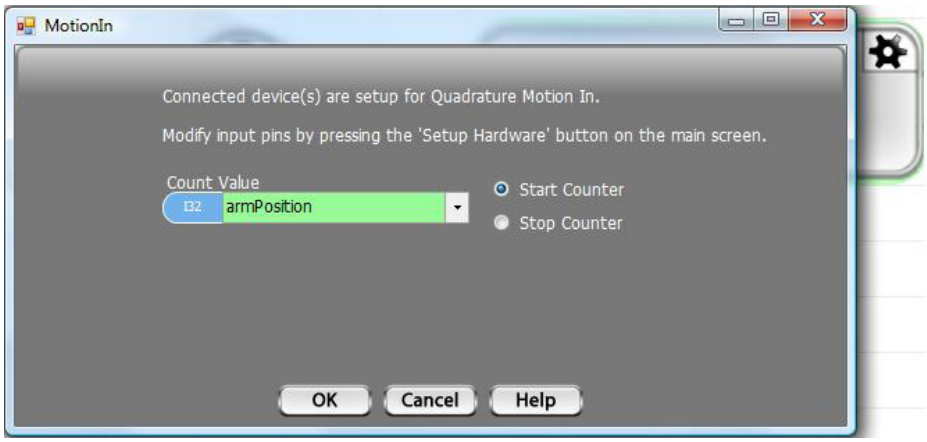
Quadrature encoders are commonly used in servo motion systems, as well as any motion or position application. For more information, there are a number of web sites that give thorough explanations of quadrature endocers, including Wikipedia.

Quadrature Pulse In is restricted to counting transitions on a dedicated pair of digital input signals. One quadrature pulse counter can be implemented per PLC module (Ace, Branch or Expansion), so up to 15 can be implemented per application.

During hardware setup, the following screen will pop up, with high speed counter and stepper and motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure a quadrature pulse counter for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expansion), select the Type as "quadrature" and select the particular digital inputs to use for your count, as shown below.



When entering your program, select the MotionIn icon and place it into the program. A dialog box, like the one shown below, will pop up. For quadrature MotionIn, you simply need to choose the appropriate Start Counter or Stop Counter selection.



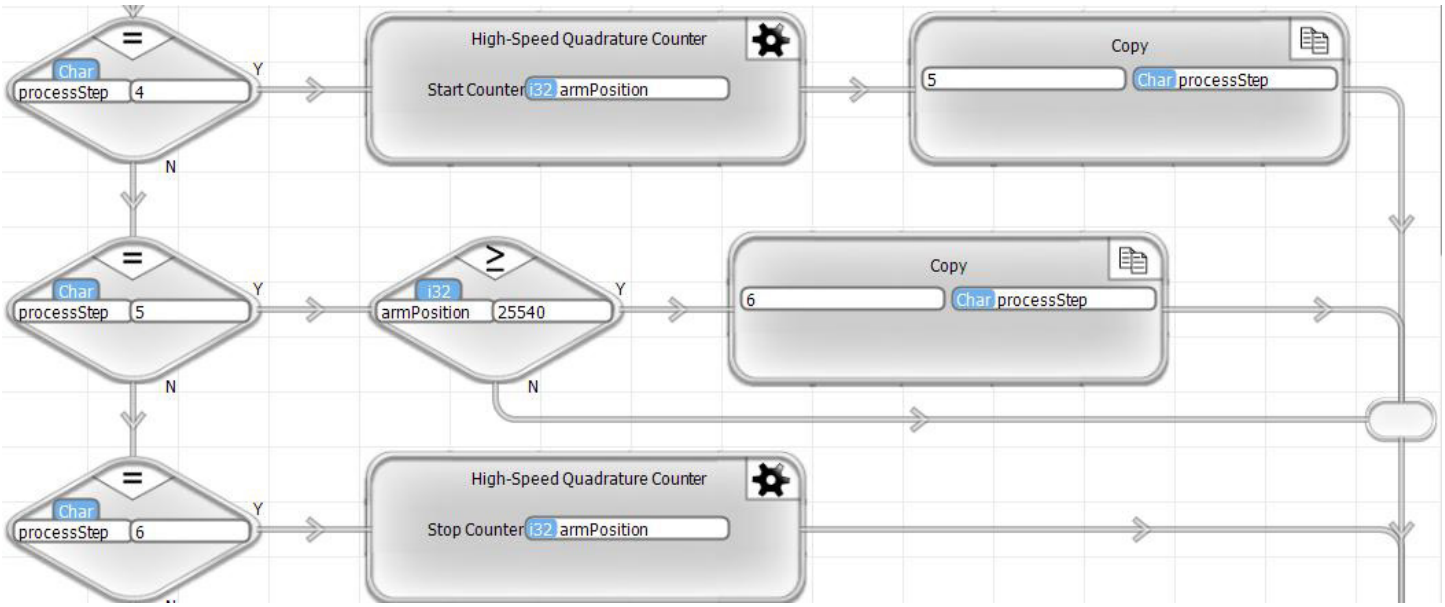
When a quadrature pulse counter is started, it will continue to operate in the background until a "stop counter" block is executed.

The tagnamed variable that you choose for Count Value must be an i32.

Remember the limit of one high speed counter per main PLC or embedded object.

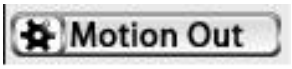
◊ Example

The example below, shows that when processStep is 4, the High Speed Quadrature Counter is Started with the count value as armPosition and processStep is changed to 5. During processStep 5, the armPosition is checked for greater than or equal to 255400. When armPosition becomes greater than or equal to 255400, processStep is changed to 6. In processStep 6, the High-Speed Quadrature Counter is Stopped.



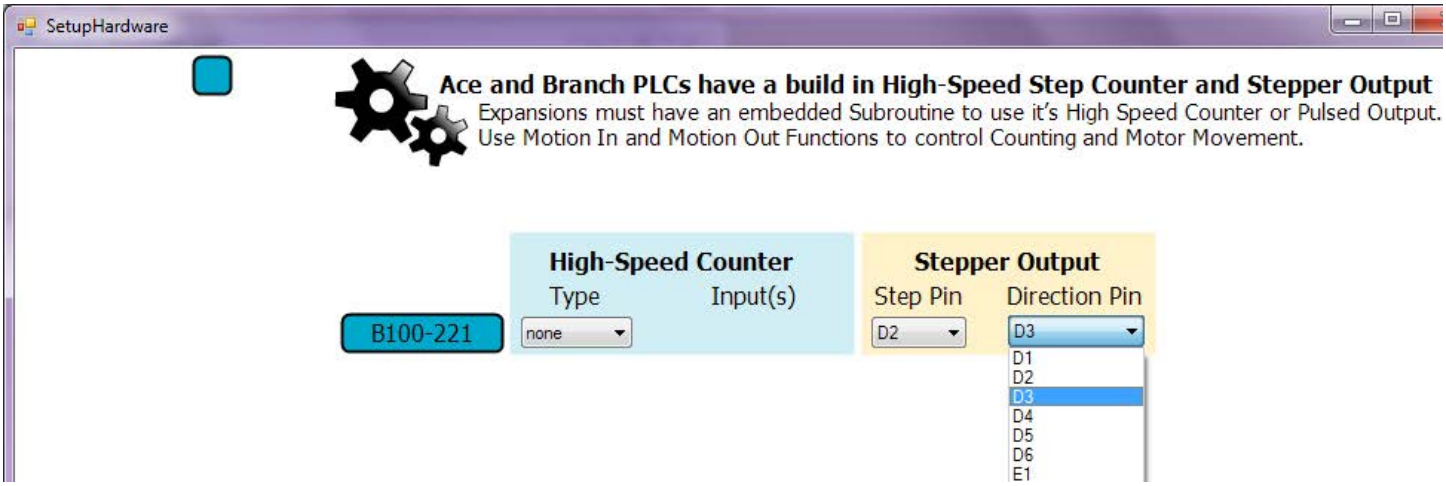
MotionOut

MotionOut blocks are used to control stepper motor motion.

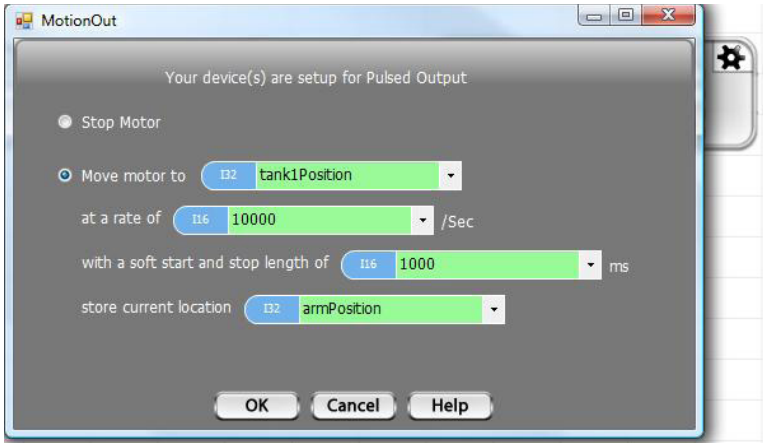


Stepper motion outputs use two digital outputs. One output is a direction signal, telling the motion driver to step in either the positive or negative direction. The other signal is a high speed pulse output, which outputs one pulse to command each stepper motor step. One stepper motion control can be implemented per PLC module (Ace, Branch or Branch Expansion), so up to 15 can be implemented per application.

During hardware setup, the following screen will pop up, with high speed counter and stepper motion configuration options for the main PLC and all other modules that are set up for embedded object programs. To configure stepper motion for that module's program (either the main program for the Ace or Branch, or an embedded object subroutine for an Expansion), select the digital output pins to use for the Step and Direction signals. The output pins that you select will be dedicated for stepper motion control and not available for any other operation.



When entering your program, select the MotionOut icon and place the MotionOut block in your program. A dialog box, like the one shown below, will pop up. For a Move command, select Move motor to and either type in the numeric position that you want to move to, or select an i32 tagname that contains the target position. Next enter the pulse rate that you want the motor to move at for the majority of the move (generally the highest pulse rate). If you want the movement to ramp up to full speed to start and ramp down from full speed to stop (which is highly recommended) enter the number of millisecond to allow for the ramp up and ramp down. Lastly, provide an i32 tagnamed variable to be used for the location.



Generally, in a system, the location variable is used for all movements. Usually an initialization program is used to find the "home" position. The home position is normally assigned a location value. All movements after that are to locations relative to the home position.

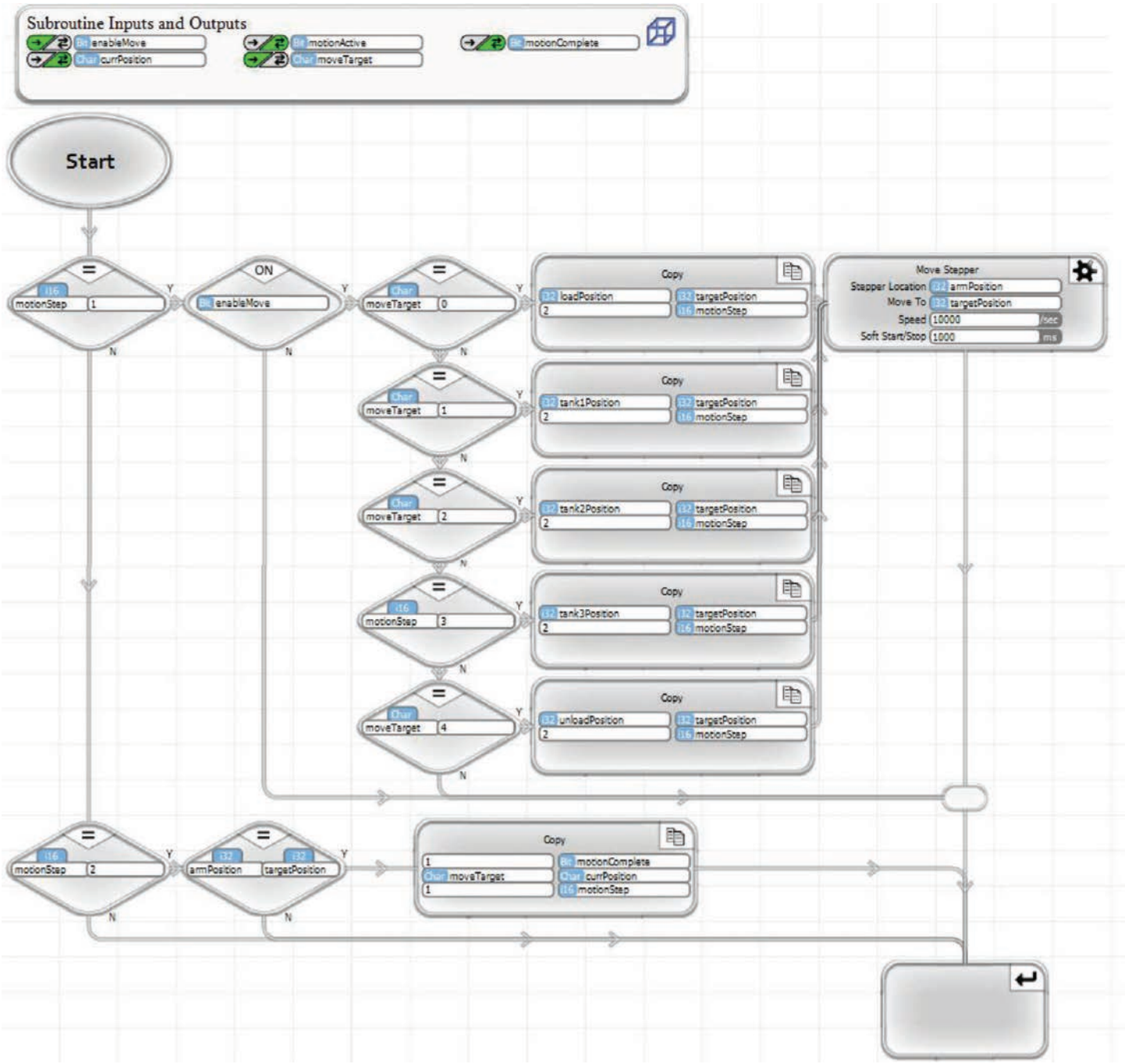
When a Move is started, it will operate in the background, until the target position is reached.

The stepper motion output is specified for a typical pulse rate up to 100KHz. Actually, pulse rates up to 250KHz are possible. Keep in mind that stepper motion control consumes approximately 1.4% of processor time per 10KHz.

◊ Example

The example, below, is subroutine that could be used for a linear transfer arm in a semiconductor wet process tool application. Wet process tools have a series of liquid tanks that perform chemical or cleaning operations on semiconductor wafers. They commonly have a transfer arm that transports the wafers between the tanks in a predefined recipe process. One of the motors would control the lateral position. In this example, we have a machine with a load station, three liquid tanks and an unload station. This example illustrates the control of that lateral position. This subroutine would be called with an enableMove flag indicating that a move is required and the moveTarget indicating where to move. In motionStep 1, the targetPosition is established based on the enable & moveTarget, and the Move is started.

In motionStep 2, the subroutine waits until the armPosition is at the targetPosition. When it is, it stops the motion (which is really not required, since the finished move stops it anyway). Then it passes back motionComplete and the tank, or load/unload station number where the arm is currently positioned and sets up to wait for the next move command.



PID

The program block icon for the PID function is shown on the right.

PID is Proportional - Integral - Derivative control. It is the very commonly used in process control, heating control and motion control. The PID function calculates the "error' value between a measured process variable and the desired setpoint. It adjusts an output in an attempt to minimize the error. This adjustment occurs on a regular period. Each time an adjustment is made, it takes into account the magnitude of the error (Proportional), the accumulated total of the error over time (Integral), and the rate of change of the error (Derivative).

PID control is complex enough that we won't provide a detailed discussion in this manual. It is a common control algorithm and there are other excellent reference documents that you should review to get an understanding, if you are new to PID. One very good description is on Wikipedia at the following link.

http://en.wikipedia.org/wiki/PID_controller

We will also put out an application note on PID control and how to use it in the near future.

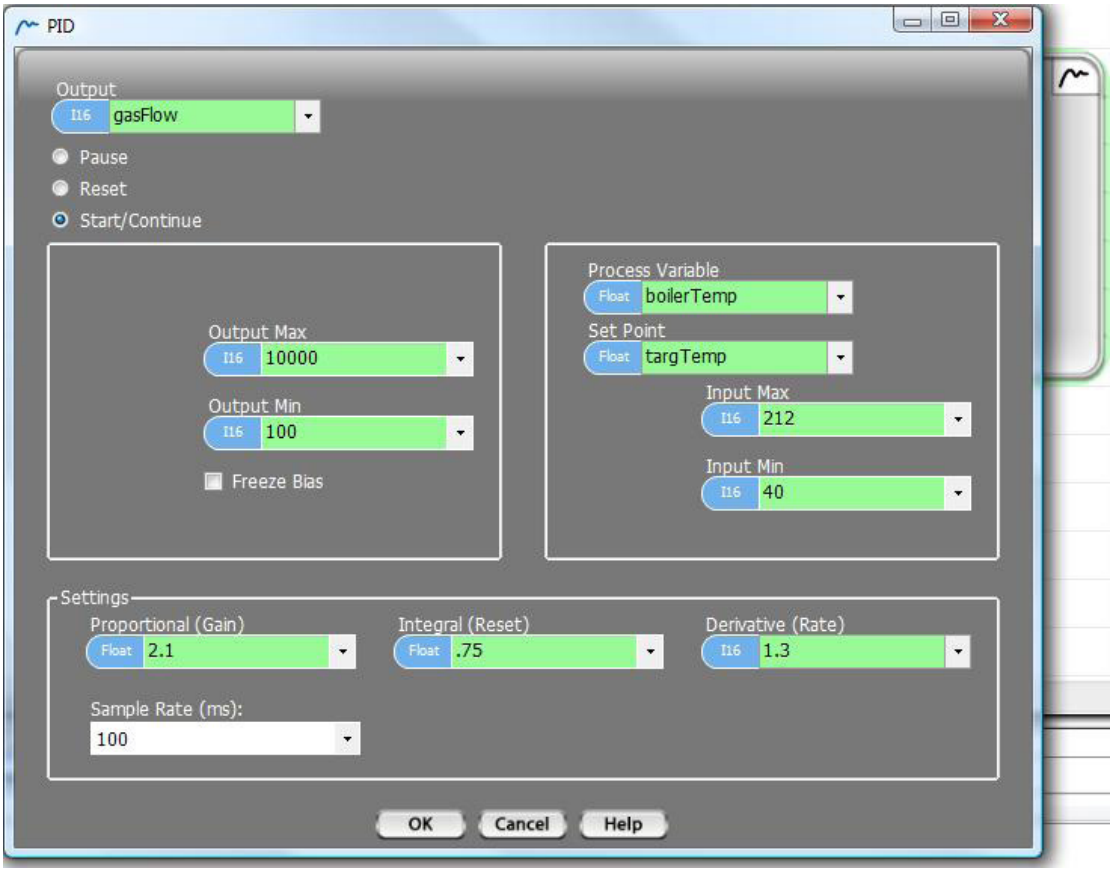
In vBuilder, PID control is a background task. That means that you Start it and it will continue to run in the background, constantly calculating the error, integrating it and determining its rate of change and updating the output based on these factors combined with the constants that you set up. Once started, the PID will operate until it is Paused

The basic PID equation used in the PID function of vBuilder is :

$$O = P*E + I * \int E\,dt + D * \Delta E/dt$$

where : O : Output
P : Proportional constant (sometimes referred to as the Gain)
I : Integral constant (sometimes referred to as Reset)
D : Derivative constant (sometimes referred to as Rate)
E : Error
dt : change in time

When you place a PID block in your program, a dialog box, like that shown on the right, will pop up. With this dialog box, you can Start the PID operation, Pause it, or Reset the PID's integral. A PID function is one that you normally need to "tune" to get the right performance. By tuning, we mean adjust the constants, try it, adjust again, until the response works like you want.



PID Start/Continue

The Start/Continue selection in the PID dialog box creates a function block that does exactly what it says. If the PID is not active, it will Start the PID's operation. If it is already running, it will continue operation. A PID actually only has to be started once. It will continue to operate until it is Paused.

When you select Start/Continue, you must select or enter a number of parameters. The following is a list of those parameters, with an explanation of each.

- **Output** : This is what the PID is actually adjusting. The Output should have a direct impact on the value of the Process Variable. For example, a butterfly valve controlling the gas supply to a boiler's burners has a direct impact on the temperature of the boiler.
- **Output Max** : The maximum value allowable for the Output. Regardless of the result of the PID calculation, the Output will be restricted to no more than this value. [any variable type except Bit and ui8]
- **Output Min** : The minimum value allowable for the Output. Regardless of the result of the PID calculation, the Output will be restriced to no less than this value. [any variable type except Bit and ui8]
- **Process Variable** : This is the measurement of the parameter that you are trying to control.
- **Set Point** : This is the desired value of the Process Variable at this time
- **Input Max** : The maximum value of the Process Variable measurement that will be used for the PID calculation. If the actual measurement goes above this value, this value will be used.
- **Input Min** : The minimum value of the Process Variable measurement that will be used for the PID calculation. If the actual measurement goes below this value, this value will be used.
- **Proportional** : The constant multiplier to the error.
- **Integral** : The constant multiplier to the integral of the error.
- **Derivative** : The constant multiplier to the derivative of the error.
- **Sample rate** : The amount of time between each PID calculation/adjustment, in milliseconds.
- **Freeze bias** : This is a selectable option. If selected, it is used to limit the impact of integral portion of the equation to no more than the entire Output range. This is useful in preventing overreaction after a time period where some factor prevented the actual control of the Process Variable [which could possibly result in a huge integral value].

When you Start a PID, it will continue operating at the defined sample rate, behind the scene. In other words, once you start it, you don't need to continue to execute PID Start/Continue blocks for operation. It doesn't hurt anything if you do. Its just not necessary.

PID Reset

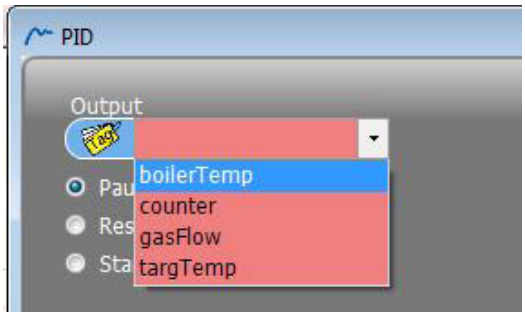
A PID Reset sets the Integral value to the value required to produce an Output equal to the Reset Value. It is something that you should do when starting a PID. The Start/Continue does not initialize the Integral value, because it does not know whether this is the actual Start or whether it is a Continue.

When you place a PID Reset in a flow chart, select the Output variable of the PID that you want to reset, as shown on the right. Select the value of the PID output to start the PID with.

PID Pause

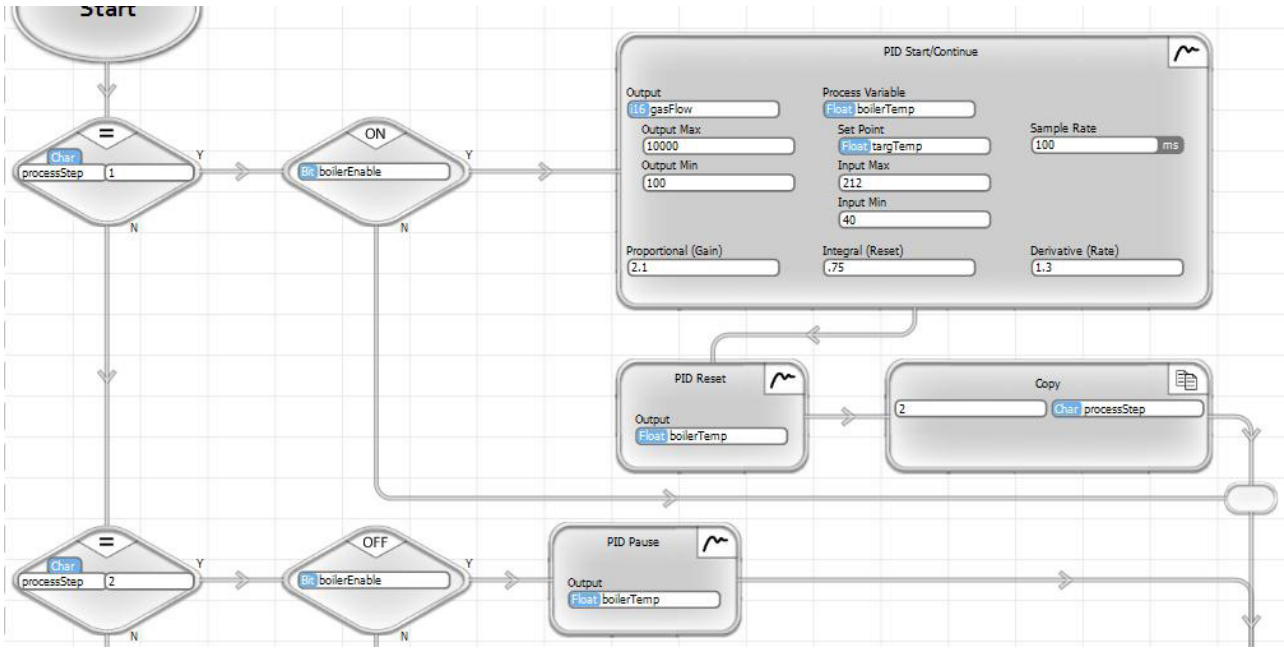
A PID Pause block stops the PID from operating.

When you place a PID Pause in a ladder rung, select the Output variable of the PID that you want to pause, as shown on the right.

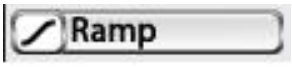


◊ Example

The example, shown below, illustrates a PID used to control a boiler. The PID Start and Reset will occur in processStep 1, the boilerEnable is ON (1). In processStep 2, the PID continues to run until boilerEnable is OFF (0), then PID is Paused.



Ramp



The icon for the Ramp function is shown on the left.

The Ramp function changes a value from its initial value to a target value, at a defined rate of change. Optionally, a ramp can include a soft start and soft stop. A soft start ramps the rate of change up to the defined rate of change gradually, over the defined soft start/stop period. A soft stop ramps the rate of change from the defined rate to 0, over the defined soft start/stop period.

Ramp is commonly used for motion control. The vBuilder Stepper Motion function has its own ramp feature built in, so this Ramp function would not commonly be used for stepper motion control. It would be applicable to servo motion control, typically coupled with a high speed pulse counter input and PID.

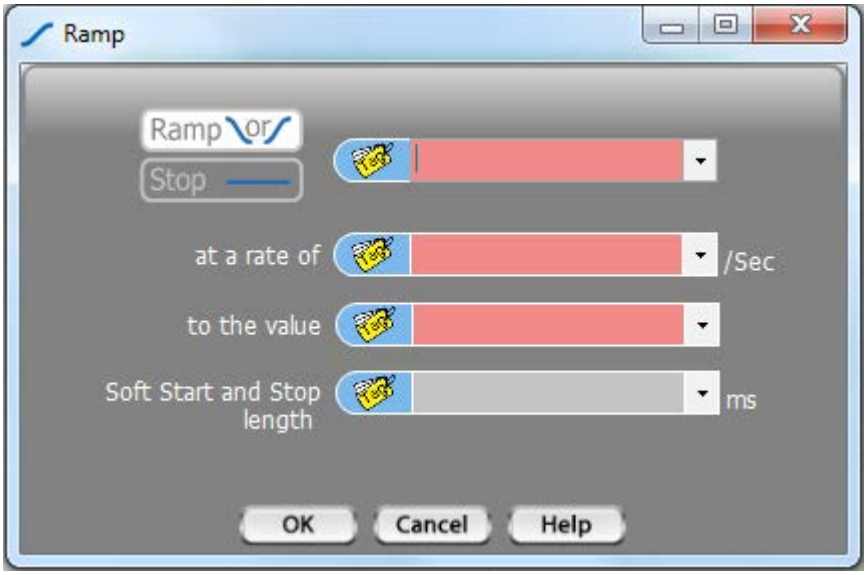
Ramp is also applicable to other machine and process control applications. Any time you want to change a variable at a defined rate, Ramp can be used.

Ramp is a background task. It only needs to be Started. Once Started, it will continue to operate in the background until Stopped.

Ramp Start

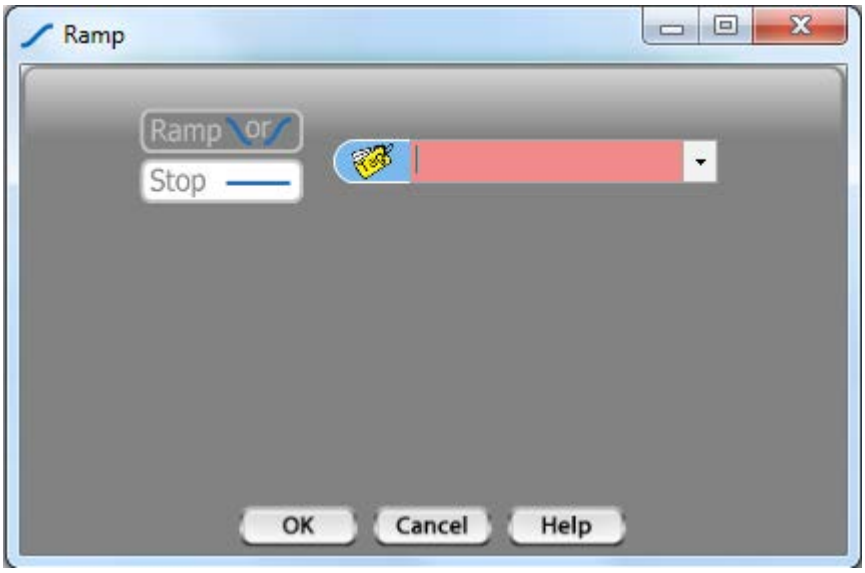
When you place a Ramp block in a flow chart, a dialog box will pop up. This box will allow you to Ramp Start, or Ramp Stop. To Start a Ramp, select the Ramp button, then enter the parameters.

- The top selection is the variable that you want to Ramp.
- Next is the ramp rate in units per second of whatever unit the Ramp variable is. The Ramp Rate is an absolute value. Whether it is a rate of increase or decrease depends on whether the Ramp variable is above or below the target.
- Next is the Ramp Target - the .final value of the variable after ramping
- Optionally, you can enter a Soft Start/Stop time. If you enter a Soft Start/Stop time the ramp rate will ramp up to the Ramp rate over the defined period and ramp down over the same defined period, when approaching the target. This will have the effect of rounding off the value curve, like that shown in the Ramp button graphic.



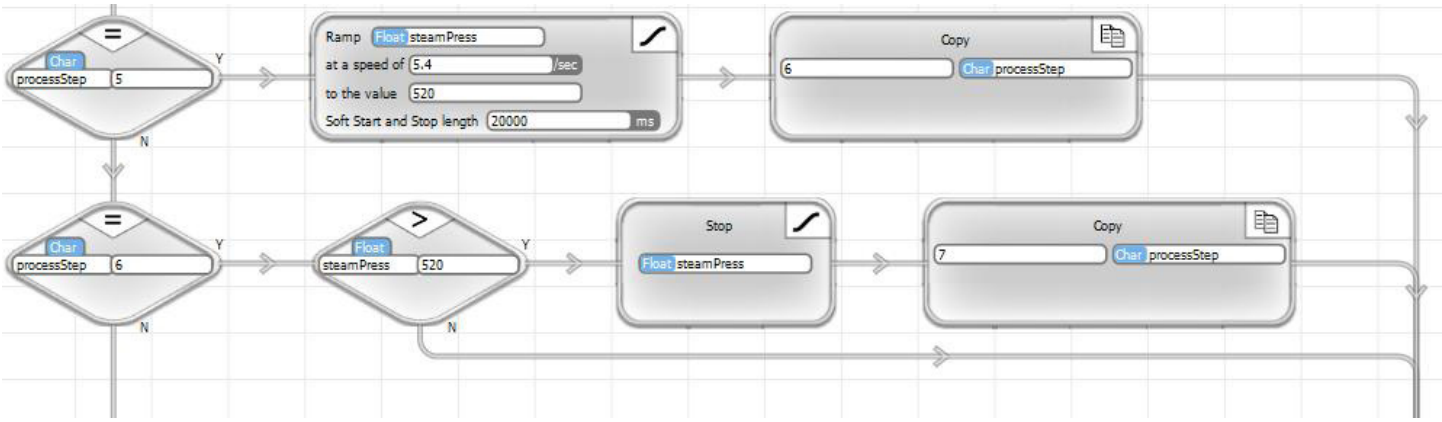
Ramp Stop

Ramp Stop will Stop the Ramp function. This is something that you would normally do, once the target value has been reached. When you place a Ramp block, select Stop, as shown on the right. Next, select the variable that is being ramped. Select OK.



◊ Example

The example, below, shows the ramping of a steamPress setpoint. In processStep 5, the Ramp is initiated, then processStep is set to 6. In processStep 6, when the steamPress setpoint is greater than or equal to the target value (it will actually stop ramping at equal to), the Ramp is Stopped.



Scale

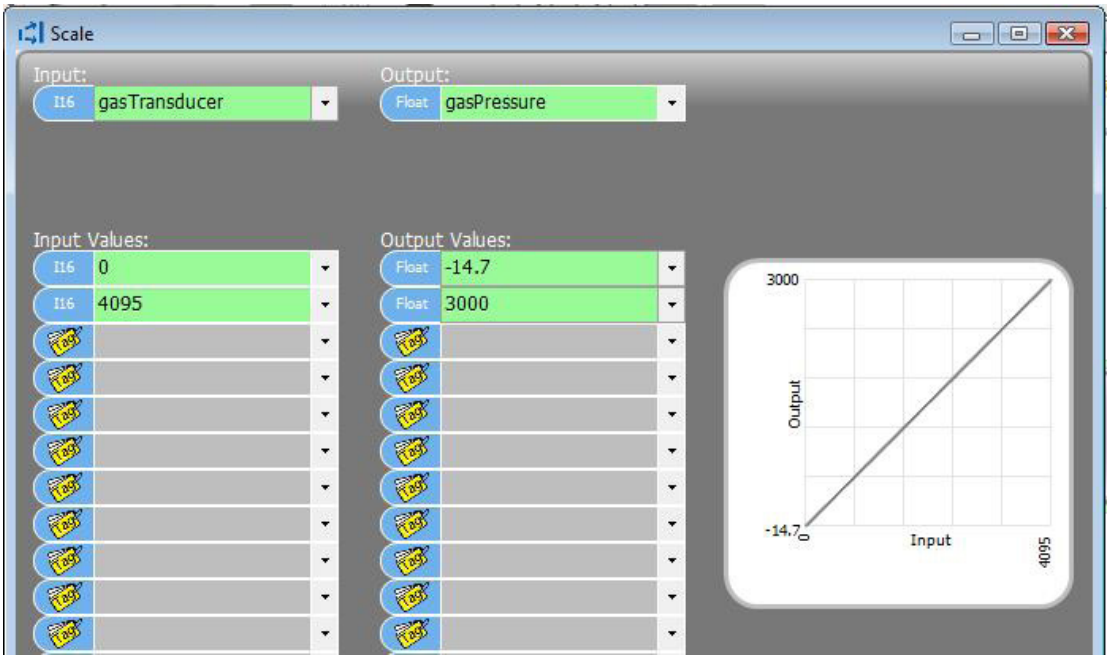
The Scale program block icon is shown on the left.

The Scale function is used to automatically convert an input value to an output value. It is commonly used for scaling a raw analog reading to its equivalent value. Scale has the capability to do piecewise linear scaling between up to 16 points. With piecewise linear scaling between such a high number of points, Scale can be used to convert values from non-linear transducers as well as an assortment of other applications.

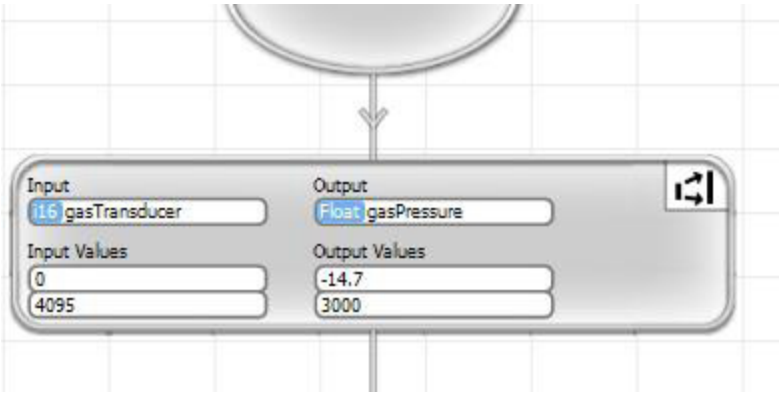
When you place a Scale block, a dialog box, like that shown below will pop up. In the dialog box, you need to choose the Input and Output variables. You must also define the scaling in the Input and Output tables, as shown. The scale curve will automatically be created on the right to illustrate the scaling that will take place.

◊ Example 1

The example shown below, shows the simple scaling of a 12 bit A/D conversion of a pressure transducer that outputs -14.7 to 3000 psi as 0 to 5V. As shown in the graph, it results in a linear conversion curve.



After clicking OK, the program block will be like that shown on the right.

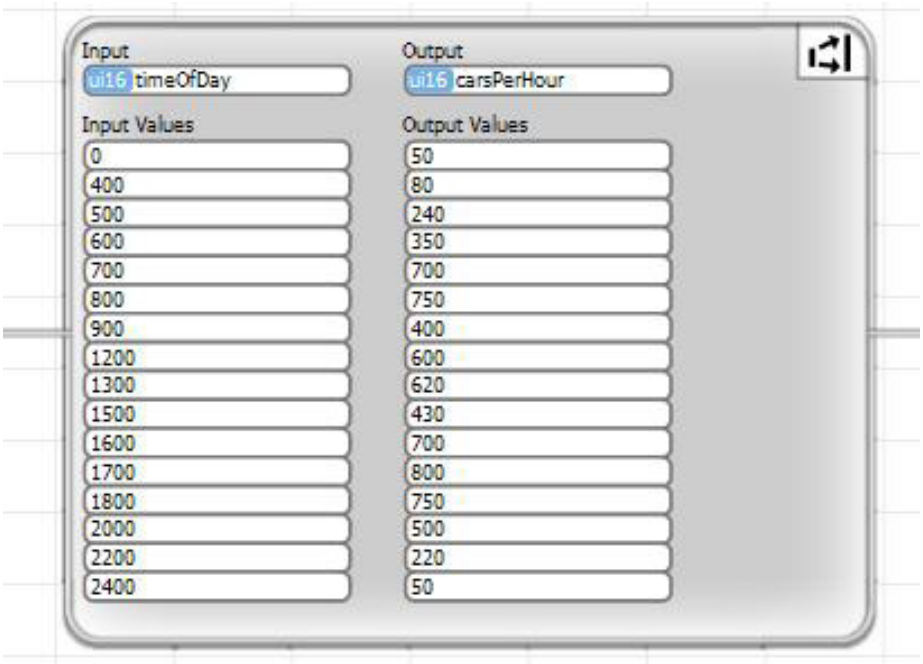


A more complex use of Scaling is shown on the next page.

◊ Example 2

The example on the right is for a traffic control application. In this application, the average rate of cars per hour that go through a particular intersection is known for 16 different times of the day. The table is set up, as shown. Between any two known points, piecewise interpolation is a good estimate of the expected number of cars per hour at any particular time.

This type of Scaling could be used in a traffic light control application. If seperated Scaling tables were utilized for traffic in different directions, the on and off times for red, and green lights could be dynamically adjusted during the day to minimize traffic delays and maximizing throughput.



Shift/Rotate

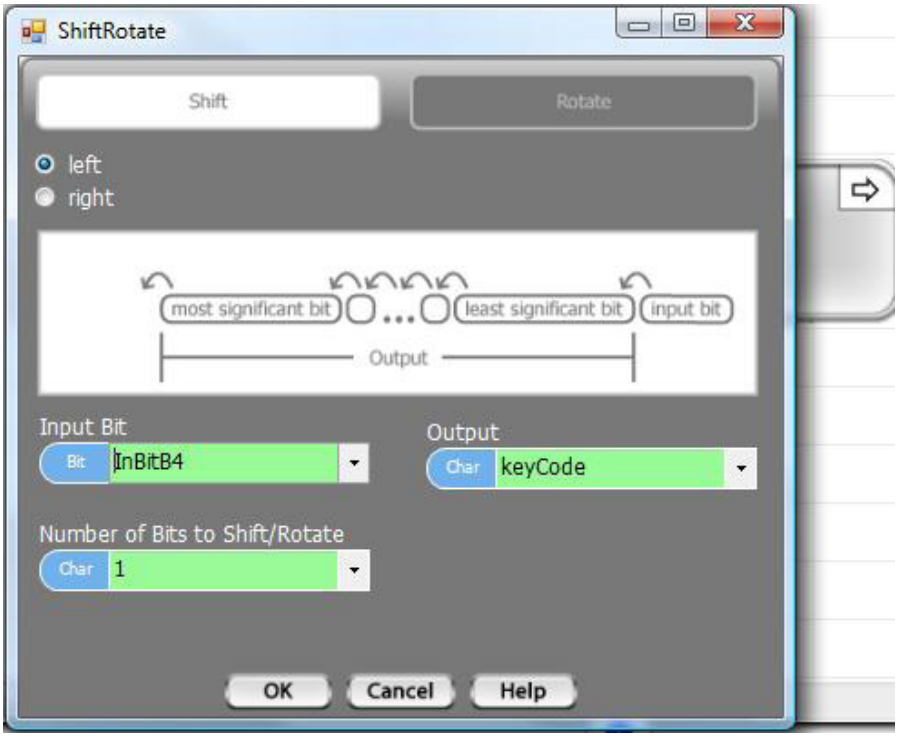
The Shift and Rotate functions provide a mechanism for shifting or rotating bits in an integer number.

The Shift function allows you to shift an integer number left or right, a selected number of bit positions. A bit value that you choose is shifted into the vacated bit position(s). For signed integers, the shift can be defined to include or exclude the sign bit (the most significant bit).

The Rotate function is similar to the Shift. The difference is that the bits that are shifted out of one end of the number are shifted back into the vacated position(s) on the other end.

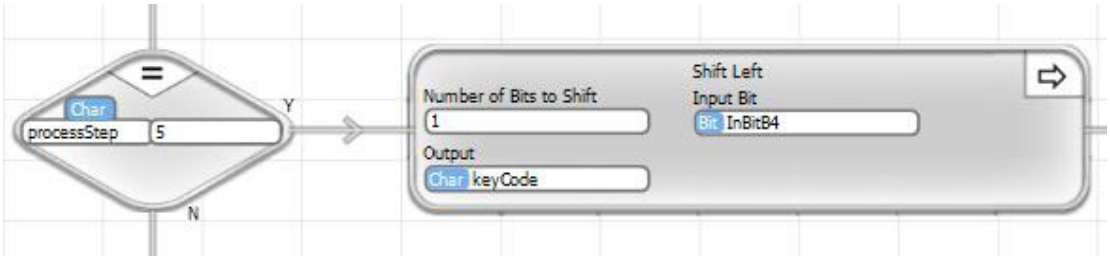
Shift (unsigned number)

The dialog box for a Shift is shown below. When you place a Shift/Rotate block, select the Shift button in the dialog box. A graphic showing the Shift operation will be displayed in the dialog box. Select the direction that you want to Shift. If you change the direction, the graphic will change to reflect the shifted direction. Select the tagnamed variable that you want to Shift, as Output. Select the bit value that you want to Shift into the vacated bit position. If you are shifting more than one bit position, this value will be shifted into all of the vacated bits. Lastly, if this is an unsigned number, select the number of bit positions to Shift. Click OK.



◊ Example Shift of Unsigned Integer

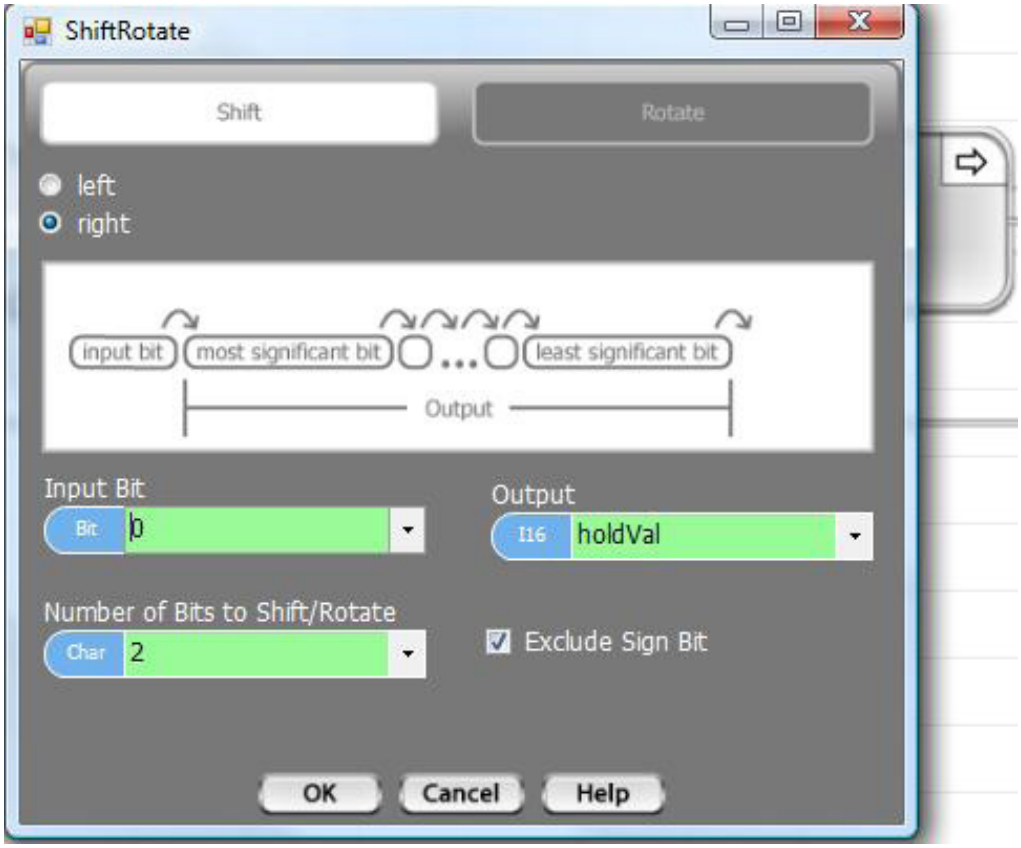
The example, below, shows a shift of a ui8 variable, named keyCode. keyCode will be shifted to the left, one bit position, with the value of InBitB4 shifted into the least significant bit.



Shift (signed number)

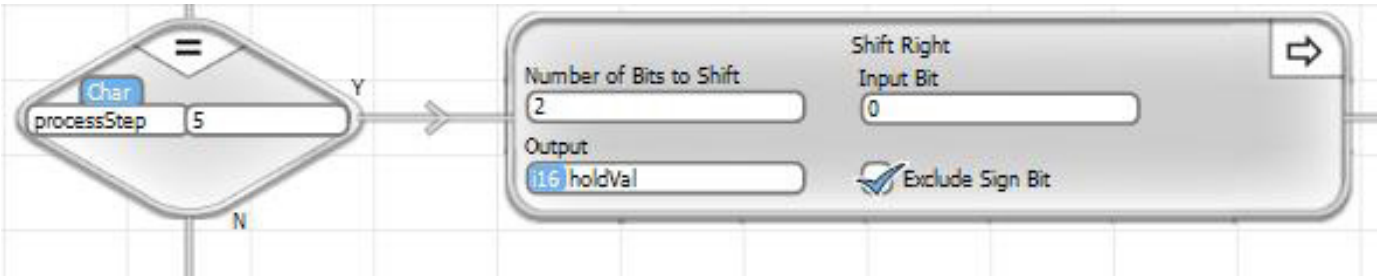
There is an additional option, when Shifting a signed number. When you select a signed integer tagname, a checkbox will pop onto the dialog box, which says "Exclude Sign Bit". You can choose to check or uncheck this box. If you check it the Shift operation will not include the sign bit (the most significant bit position). In other words, if you check the box, the sign bit will remain unchanged. If you are shifting right and check the box, the bit that is shifted in will be shifted into the second most significant bit position.

If you uncheck the Exclude Sign Bit box, shifting will include the entire number.



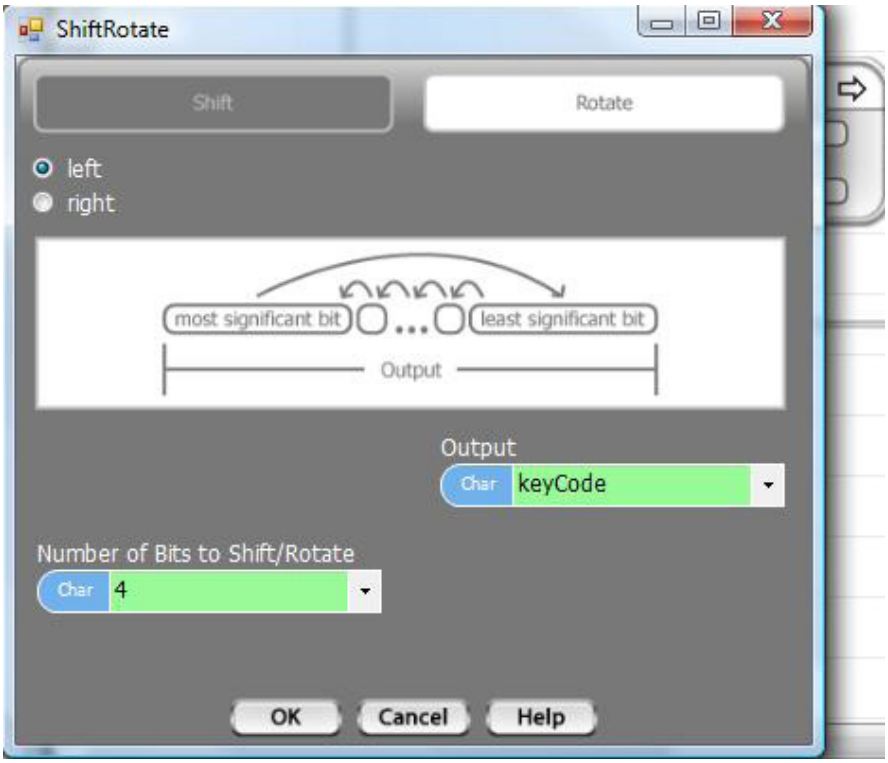
◊ Example Shift of Signed Integer

The example below shows a Shift of an s16 variable, named holdVal, by two bit positions to the right. The sign bit is excluded from the shift. A value of 0 is shifted into the vacated bits.



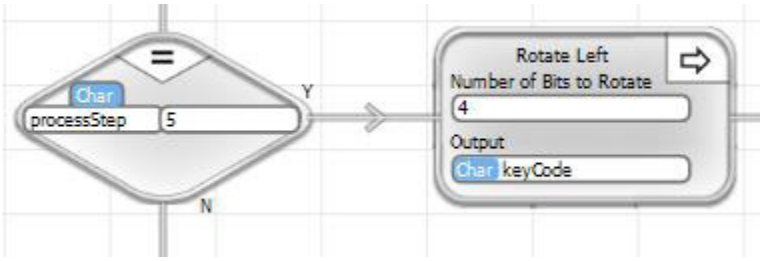
Rotate (unsigned number)

The dialog box for a Rotate is shown below. When you place a Shift/Rotate block, select the Rotate button in the dialog box. A graphic showing the Rotate operation will be displayed in the dialog box. Select the direction that you want to Rotate. If you change the direction, the graphic will change to reflect the rotated direction. Select the tagnamed variable that you want to Rotate, as Output. The bit rotated out will be rotated back in to the vacant bit position on the opposite end of the variable. Lastly, if this is an unsigned number, select the number of bit positions to Rotate. Click OK.



◊ Example Rotate of Unsigned Integer

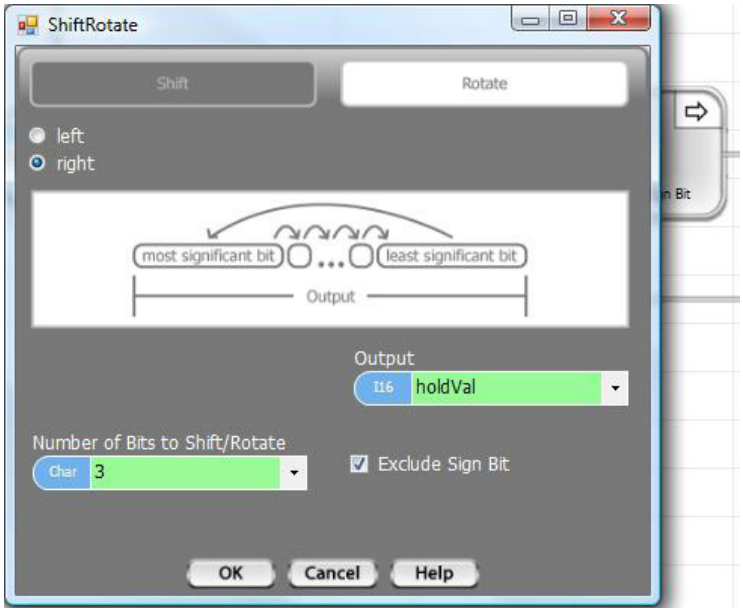
The example on the right shows a Rotate of a ui8 (unsigned 8 bit integer) variable, named keyCode. keyCode will be rotated to the left, four bit positions, with the value of bits shifted out, successively shifted back into the least significant bit.



Rotate (signed number)

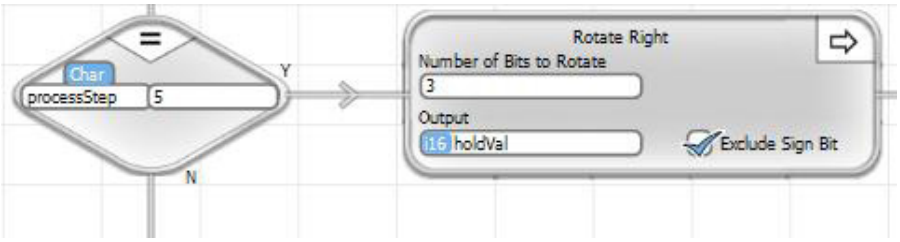
There is an additional option, when Rotating a signed number. When you select a signed integer tagname, a checkbox will pop onto the dialog box, which says "Exclude Sign Bit". You can choose to check or uncheck this box. If you check it the Rotate operation will not include the sign bit (the most significant bit position). In other words, if you check the box, the sign bit will remain unchanged. If you are rotating right and check the box, the bit that is rotated in will be rotated into the second most significant bit position.

If you uncheck the Exclude Sign Bit box, rotating will include the entire number.

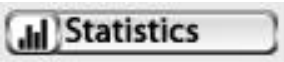


◊ Example Rotate of Signed Integer

The example on the right shows a Rotate of an i16 variable, named holdVal, by three bit positions to the right. The sign bit is excluded from the rotate.



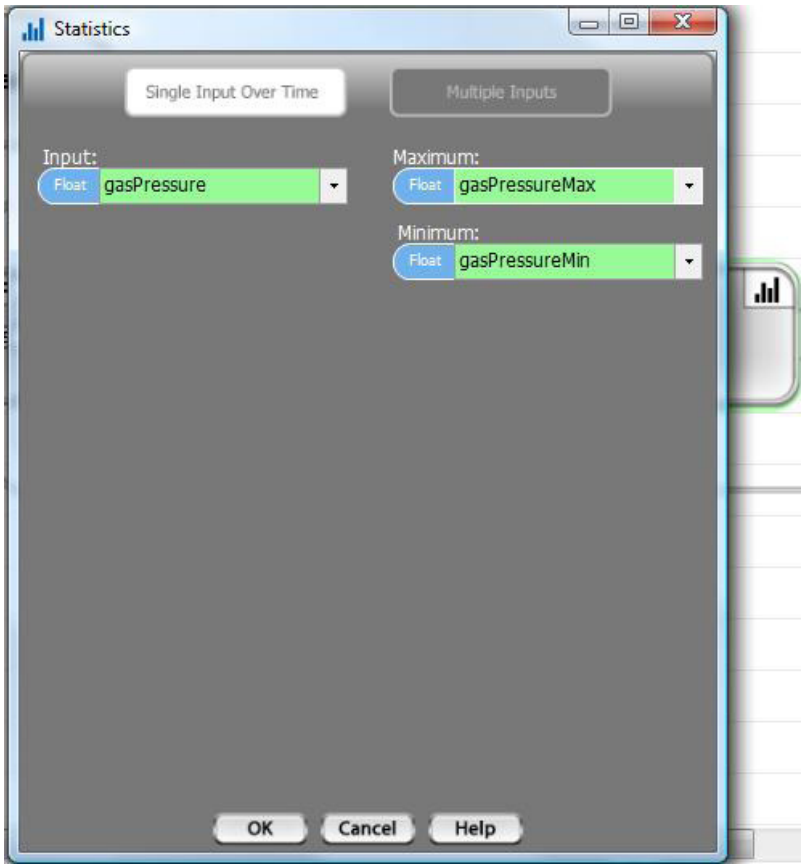
Statistics



Two basic, common statistical functions are available for a Statistics block. The Single Input Over Time option automatically keeps up with the minimum and maximum value of a selected variable, since the time the monitoring was started. The Multiple Inputs option will use a list of up to 16 input values and calculate the Maximum, Minimum, Median and Average values.

Single Input Over Time

When the Single Input Over Time button is selected for a Statistics block, the dialog box will appear as shown on the right. The Input is the tagname of the variable that you want to monitor. The Maximum is the tagname where you want to store the maximum value of the selected input. The Minimum is the tagname for storing the minimum value.



◊ Example Single Input Over Time

The example below will determine and maintain the Minimum and Maximum value of gasPressure. Each time this program block is executed, the current value will be compared to the previous minimum and maximum. If the value is below the previous minimum, the Minimum will be set equal to the current value. If it is above the previous maximum, the Maximum will be set equal to the current value.

Multiple Input Statistics

When Multiple Inputs is selected for a Statistics block, the dialog box will appear as shown on the right. Up to 16 input tagnames can be selected. The Multiple Input Statistics block can be set up to determine and calculate any of Maximum value, Minimum value, Median value and Average value. If you do not select a tagname for any of the statistical operations, this program block will simply skip that function. You must select at least one statistical operation though.

◊ Example Multiple Inputs

The example below is monitoring eight cereal box fill lines. This cereal factory has eight different lines that are side by side and have identical capabilities and capacities. A Statistics Multiple Inputs block is used to monitor and create meaningful real time management information on how the production lines are doing. In this case, all of the statistics are calculated.

Subroutine

The Subroutine block will place a call to a Subroutine.

Before you can create a Subroutine call, you must have already created the subroutine.

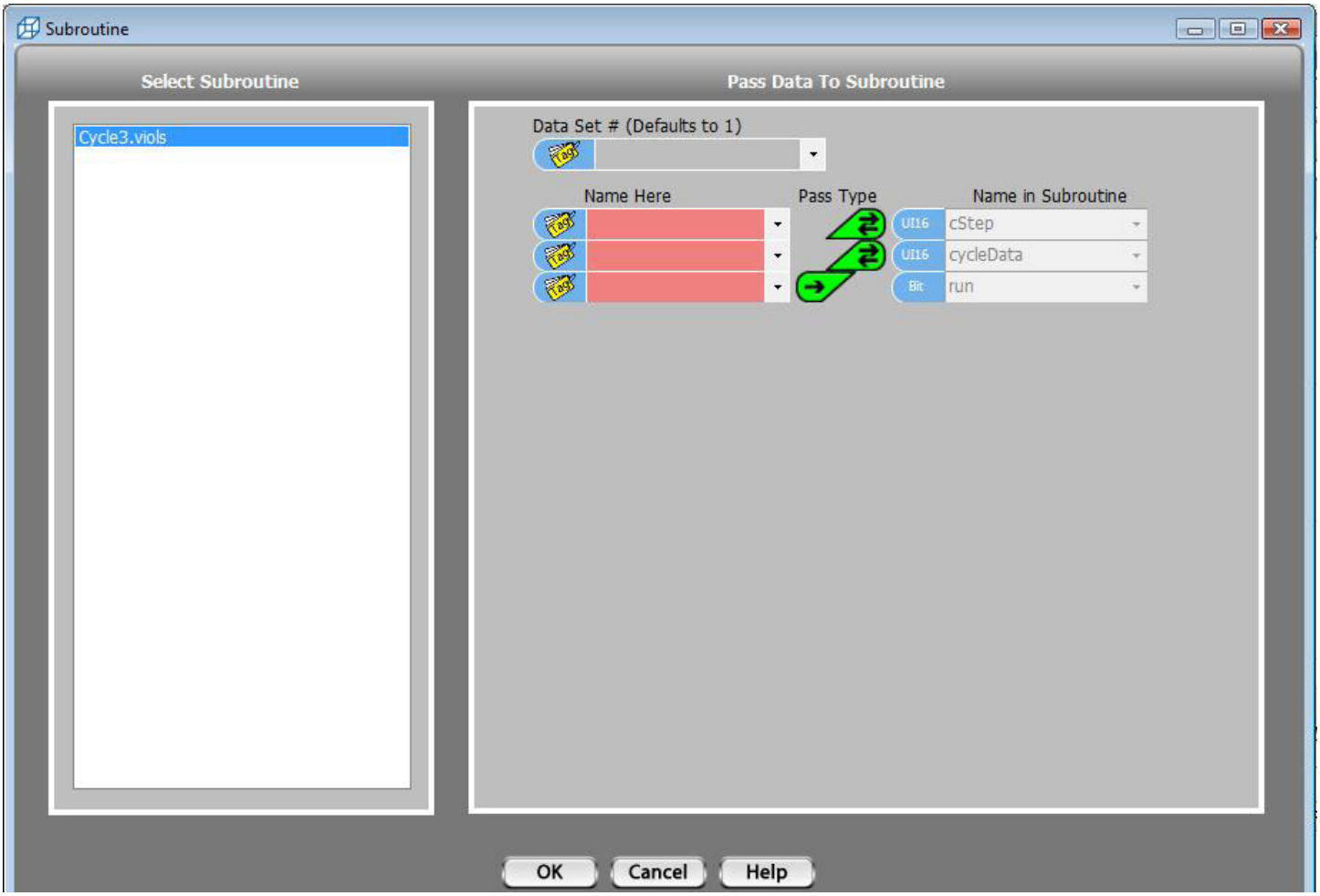
Subroutines are discussed in detail in depth in the chapter on Object Oriented Programming and Subroutines.

When you place a Subroutine Call block, a dialog box, like the one shown below, will pop up. On the left side of the box, is a list of all of the defined subroutines. Choose the one that you want to place. Once you've done that, the right side of the dialog box will contain a list of Data to Pass to the Subroutine. Since all Subroutines in Velocio PLCs are objects, the first thing you need to do is define the Data Set # (object number) that this Subroutine Call is associated with. If you only have one instance of the Subroutine, you can leave this blank.

Next, you see a list of Passed Data. On the right hand side of the list is the subroutine tag name associated with each item - in other words, the tag name used in the subroutine. On the left are selection boxes to allow you to pick the local tagnames of the data to pass. Between the two columns is the Pass Type for each parameter. The following is an explanation of the Pass Types.

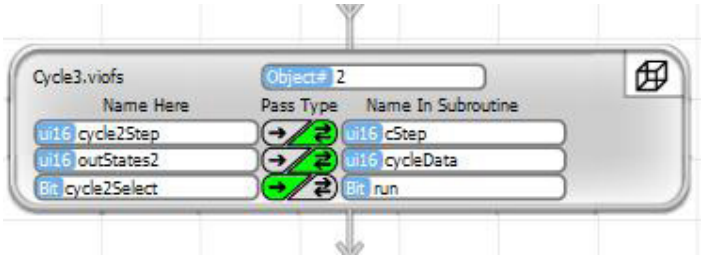
- The symbol with a single arrow pointing to the subroutine indicates pass by value. The numeric value of the data on the left (tagname variable data or constant) is passed into the subroutine. There is no effect on the data in the calling program.
- The symbol with arrows going both directions indicates pass by reference. In this case, actual data is not passed. What is passed into the subroutine is a "reference" to the tagname variable. This reference allows the subroutine to read and/or change the value in the calling program. This is how data is passed back out of a subroutine.

Fill out the Pass Data to Subroutine list and click OK.



◊ Example Subroutine Call

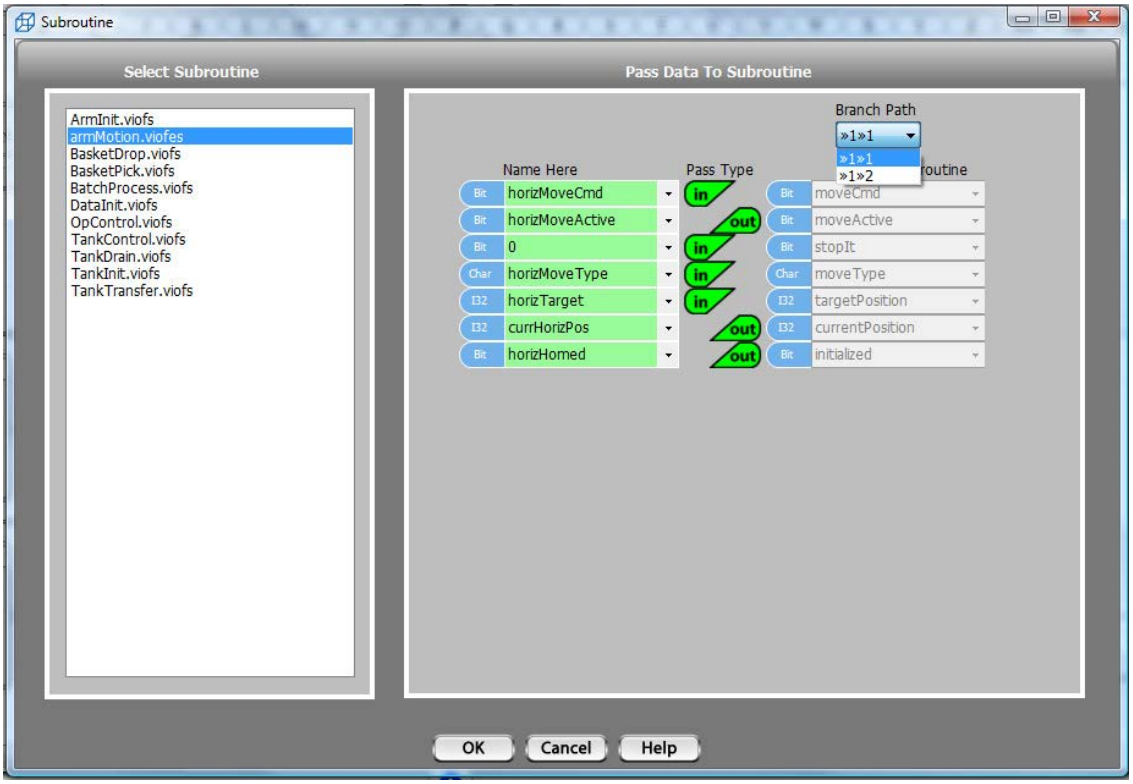
The example on the right shows a Subroutine Call block. The Call is to Cycle3, object #2 (data set #). Two parameters are passed by reference and one is passed by value.

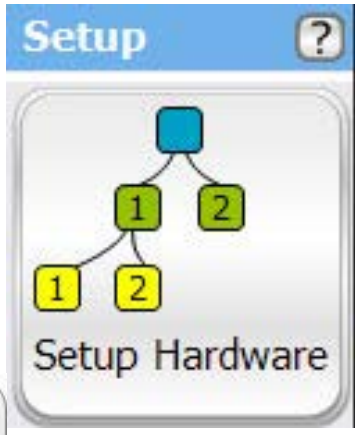


Calling Embedded Object Subroutines

Calling an Embedded Object Subroutine is basically the same as calling a local subroutine. There are a couple of differences though.

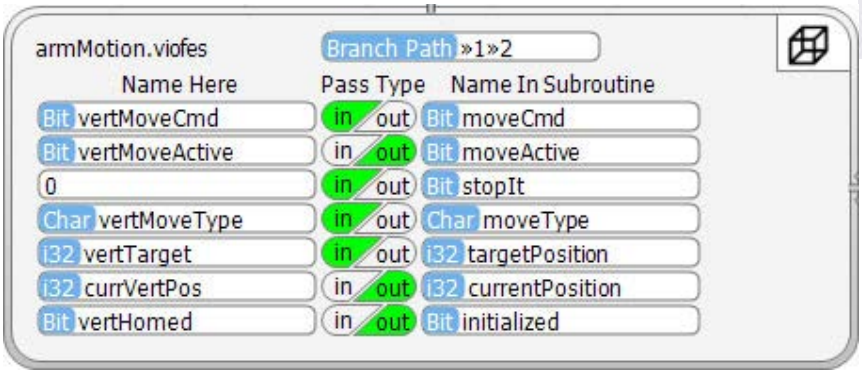
- Parameters passed to and from Embedded Objects are In and Out. There is no pass by reference. You either send data to the object or receive it from the embedded object.
- If there is more than one device configured for the same embedded object, you must select which one you are calling. The particular embedded object is defined by its branch path. The branch path is defined by its port connection relative to the unit the call is made from. For the example shown on th right, the choice is between the unit connected through its first vLink port, then through that unit's first vLink port (>>1>>1), or the one connected through its first vLink port, then through that unit's second vLInk port. The Setup graphic makes it pretty clear. In this case, the two choices are yellow 1 and yellow 2.





◊ Example Embedded Subroutine Call

The example below shows a call to an embedded object subroutine located >>1>>2 relative to the calling program. Since the calling program is in the Branch unit, the call is to yellow 2.



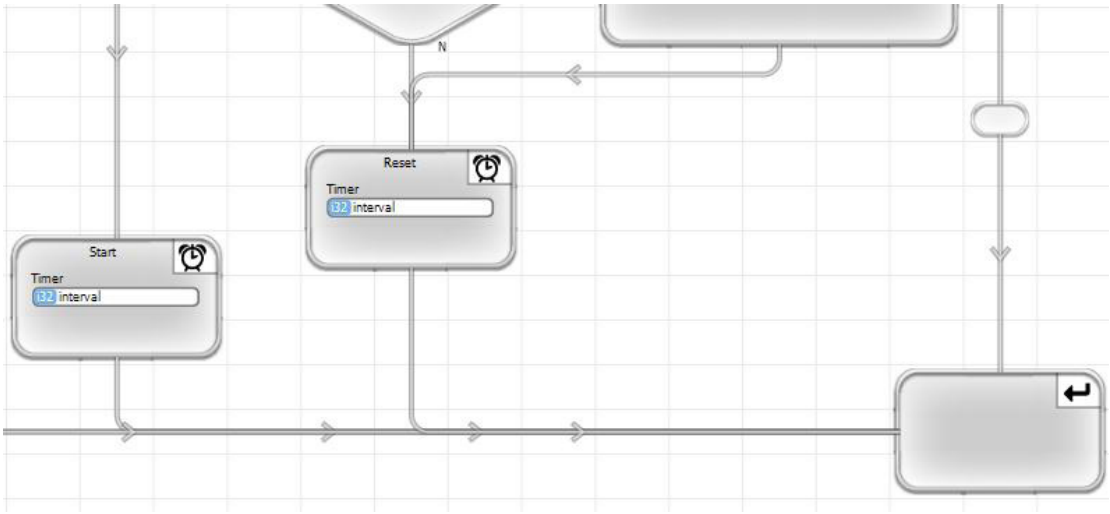
Subroutine Return

The Subroutine Return block is available only for subroutines. It is simply a block that returns to the program that called the subroutine.

The Subroutine Return has no dialog box. It simply can be placed in the Flow Chart.

◊ Example

The example, below, shows a Subroutine Return block placed in a subroutine.



Timer

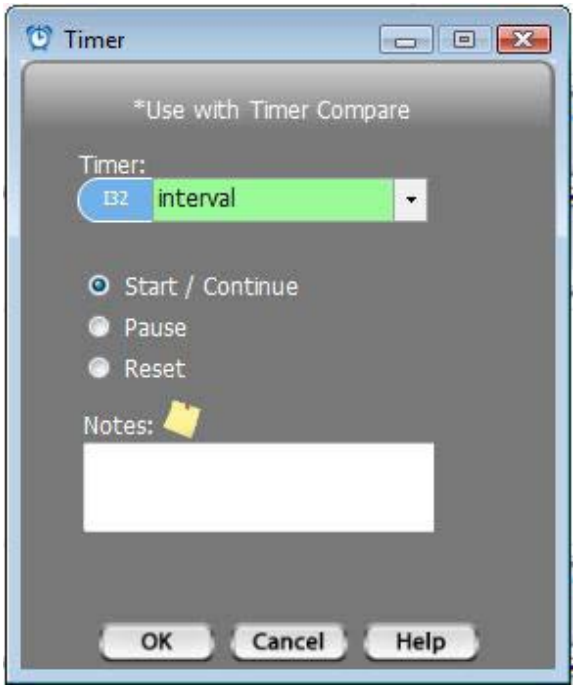
In vBuilder, Timers are background functions. Once you start a Timer, it will continue to run, until you Pause it. While it is running, you can execute Timer Compare blocks to test the time against target values. The Timer value can be Reset at any time with a Timer Reset block.

Timer Start/Continue

When you want to Start a Timer, select the Start/Continue option in the dialog box. The only other thing you need to do is place the tagname of an i32 variable to be used to contain the time value in the box labeled Timer.

Timer Start/Continue will not initialize the timer value to 0. If you want to do that, you should use the Timer Reset function.

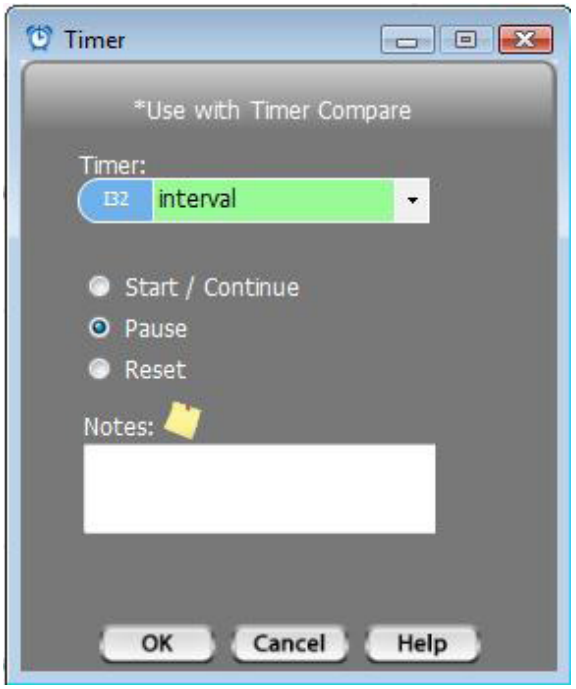
The Timer Start/Continue only needs to be executed once for a Timer to operate. It does not need to execute on every program pass. If it does, that's the Continue. It will continue to operate.



Timer Pause

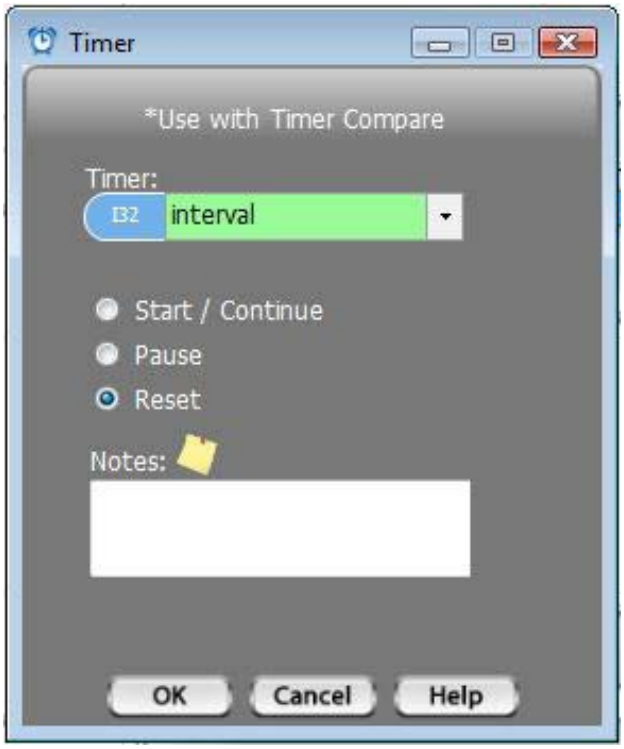
A Timer Pause block is used to stop a Timer from operating. To place a Pause, select the Pause option in the dialog box and enter the i32 tagname of the variable that holds the time value.

Once Paused, the Timer will not run again, until another Timer Start/Continue is executed.



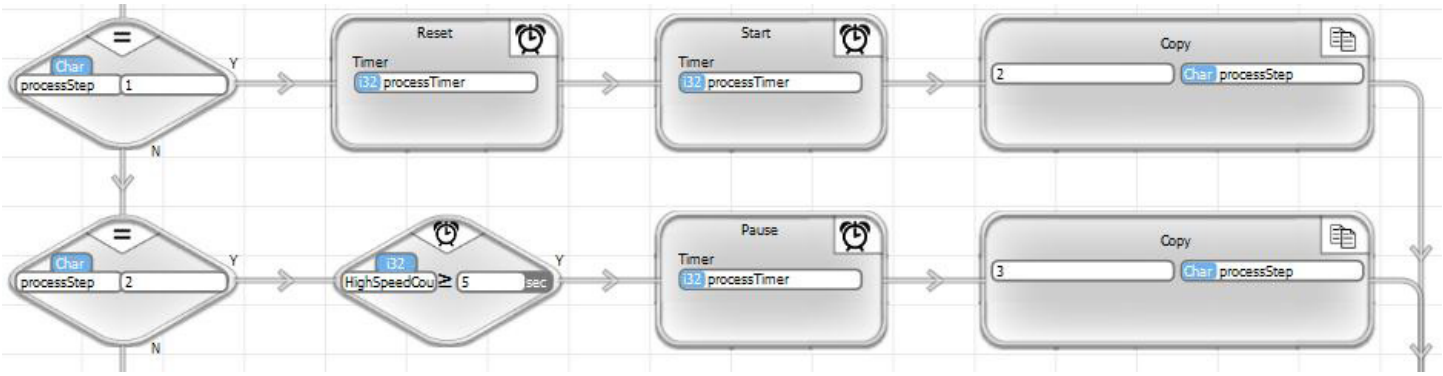
Timer Reset

Timer Reset simply resets the selected timer's value to 0. To perform a Timer Reset, place the timer block, select the 32 bit signed integer tag name or the Timer, then select the Reset option and OK.



◊ Example Timer Applications

The example, below, shows a Timer being Reset and Started in processStep 1, then checked in processStep 2. If the processTimer value is greater than or equal to 5 seconds, processTimer will be paused. Obviously, in a real program, you'd likely do other operations as well.



6 State Machine Programming

We're going to make a very sweeping statement that we know we can back up. Here it is :

Almost all control applications that have any level of complexity are natural state machine applications and are most logically, efficiently and effectively implemented with state machine style of programming.

How do we know this? Through years of developing a myriad of control systems for everything from material handling, to cereal fill line automation, to municipal water and wastewater systems, to semiconductor wet processing, toxic gas distribution, chemical mix and delivery, full semiconductor fabs, hoovercraft control and simulators for NASA's astronaut trainers, automated testers ... and on and on and on. Every single system was designed with a state machine control architecture.

Because state machine design was the best solution for every single application, we didn't use a PLC. Traditional PLCs have been very poor tools for implementing state machine logic. Ladder logic in a single large program doesn't work very well for state machine applications. It's possible to use traditional PLCs to develop one long ladder program using state machine logic, but it is a tortured process at best.

Velocio's Ladder Logic, with object subroutine and embedded object functionality lends itself much better to state machine applications. For PLC programmers who prefer ladder logic, state machine programming in vBuilder Ladder is quite straightforward.

vBuilder Flow Chart programming is an absolute natural for state machine program development. It is perfectly suited for breaking the application down to small, logical components, tying those components together and implementing the sequence logic to implement all of the requirements of any given application. It will also yield a finished program that self documents and is much easier to maintain, modify and propogate than anything else available. If we had a control application to do today, we'd absolutely choose a Velocio PLC as the best solution. We'd utilize vBuilder Flow Chart programming using State Machines.

The next few pages go through the concepts of State Machine programming.

State Machine Principles

What's a state machine? That's a good place to start. You deal with state machine principles every day without knowing it. We don't even have to be dealing with machines or control systems to explain state machines. We can start with you sitting in your living room, watching television.

Imagine yourself sitting in your living room, watching TV. That's your state. Your state is the situation that you are in. In that state, there are a number of things that could happen. Here's a few examples.

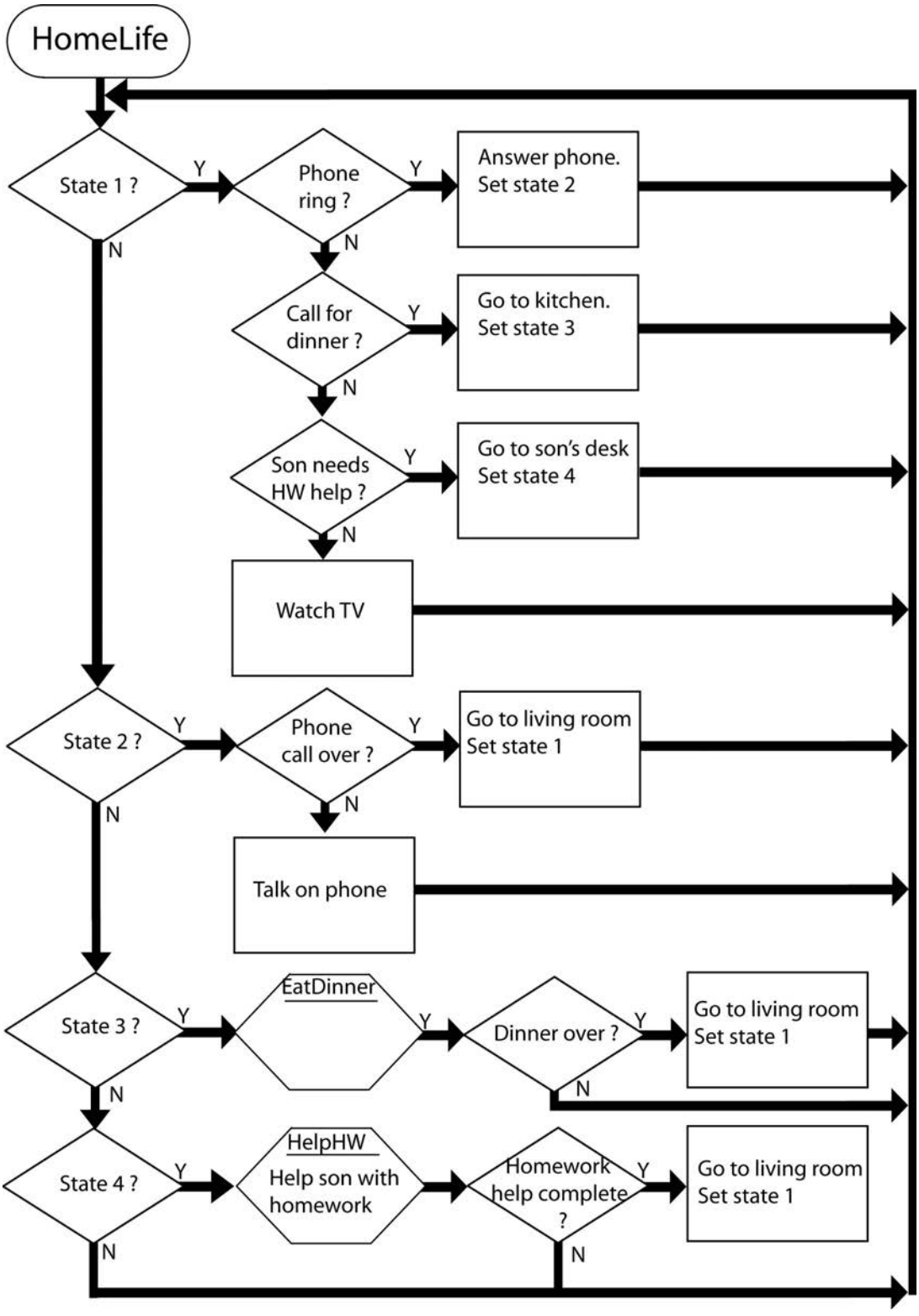
- The phone could ring
- Your spouse could call you for dinner
- Your son could want help with his homework

One of these events might cause you to transition to another state. Possible other states are -

- Talk on the phone
- Eat dinner in kitchen
- Help with homework at son's desk

The flow chart on the right shows your state machine function. There are 4 states 1)watching TV, 2) Talking on the phone, 3) Eating dinner, and 4) Helping with homework. In state machine operation, you do what is required, based on the state you are in, and you look for inputs that can transition you to another state. If you get the input, you make the transition.

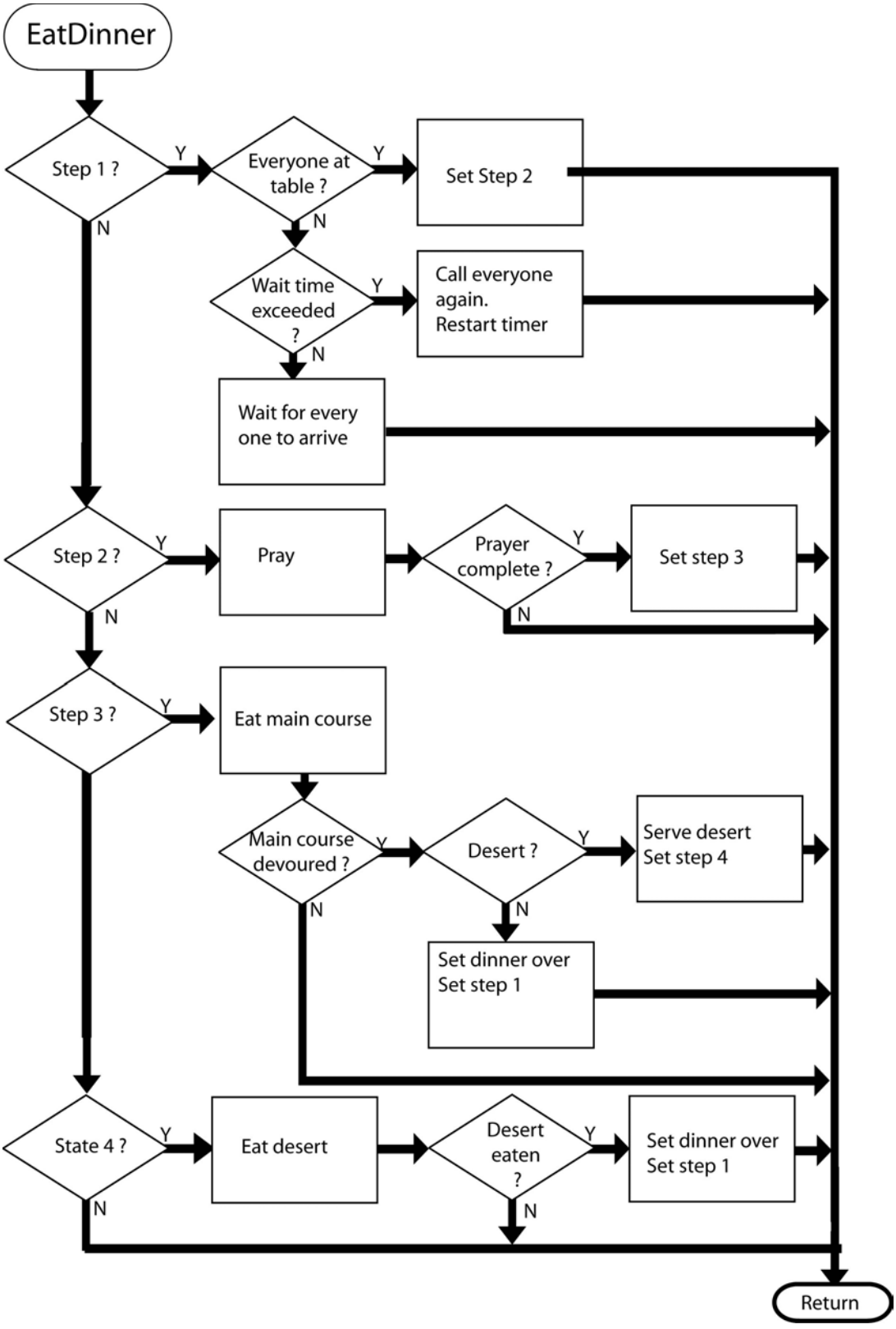
Some of these states may be a state machine in themselves. We'll look at EatDinner, which is a subroutine that we'll implement as a state machine, on the next page.



As shown on the right, EatDinner is a state machine. Notice a couple of key points. Nowhere in either the main program or the subroutine, does the program wait in a loop. It checks status and goes on. The whole program is in a super loop, so we get back to our check point very quickly. By not waiting, anywhere, we can have a number of independent state machines operating simultaneously. That is a key to successful state machine control strategy - never wait in a loop. Instead, check the condition on every pass until it is acheived, then do your operation or transition to a new state.

The other thing to notice is that we are always in a state (or step). What we do and what we are looking for depends on what state we are in. That is true of normal life. Its also how how all control program applications that we have ever seen actually operate.

Even very complex systems break down to a set of simple state machines. By breaking them down and following a few program rules, you can develop nearly any program fairly rapidly and end up with a program that is easy to follow for debug and easy to modify for future changes.

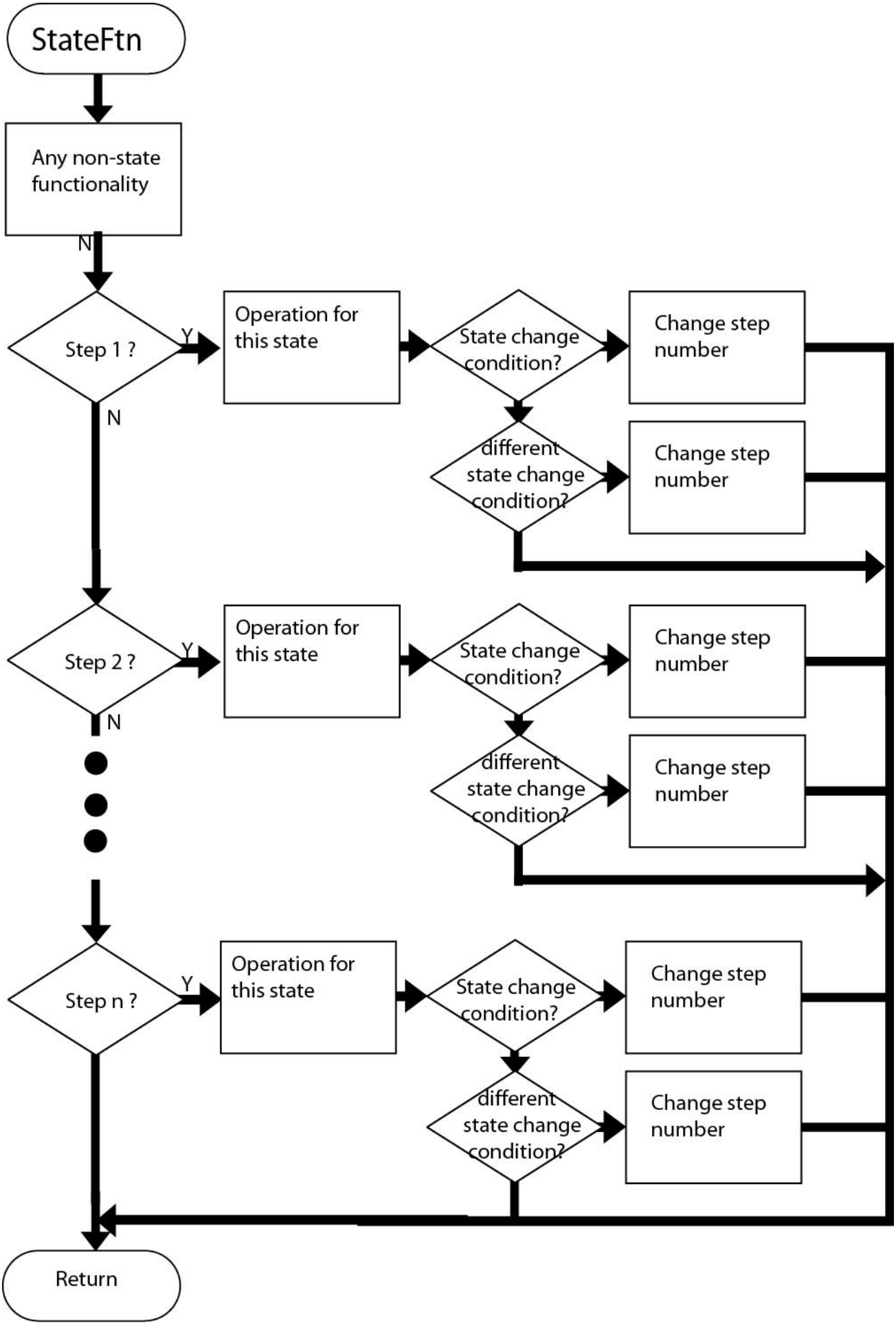


State Machine Rules & Recommendations

State machine progamming consists of logic and following a few simple guidelines. Its very easy to learn and, with a little practice, you can become very proficient and productive. Here are some general guidelings .

- Break your program up into logical subroutines that directly relate to natural components of the application
- If a subroutine still contains a great deal of complexity, break it up into the next level of subroutine
- Sketch out the main program loop with a flow chart. The main program will be a super loop that calls the state machine subroutines for the various logical components of the operation.
- For each subroutine, start breaking it down to a sequence.
  - Start with the initialization state
  - Continue with what happens after initialization is complete
  - Each step in the sequence should be designed along the lines of "while in this state do this; if this happens, do this, then transition to this state"
- Never loop the program waiting for an event to occur
- Sketch out a flow chart, on paper, with your logic. That way you can quickly & easily mark it up.
  - Create an integer as as step number to keep track of your state, or step
  - Start with the state that exists after initialization (or if this program does the initialization, the state at power on). Make that your first step. Put in blocks to do what ever needs to be done in the first step. Add decision blocks that check events that would cause you to move to another state.
  - For each state, put in blocks that perform the operations that are required in this state and the decisions that trigger a transistion to another state.
  - Make sure that you have every potential state covered.
- Repeat for every subroutine

The drawing on the next page shows the general flow of a subroutine written in state machine style of programming. The concept is very simple. When applied to a program of any size the result is very powerful.


StateFtn
Any non-state functionality
N
Step 1 ?
Y
Operation for this state
State change condition?
Change step number
different state change condition?
Change step number
N
Step 2 ?
Y
Operation for this state
State change condition?
Change step number
different state change condition?
Change step number
N
Step n ?
Y
Operation for this state
State change condition?
Change step number
different state change condition?
Change step number
Return

7. Object Oriented Programming and Subroutines

One of the major advances introduced in Velocio Builder is the application of object oriented programming to graphical PLC languages. Object oriented programming has been the mainstay in the general computer software world for over 20 years. The reason object oriented programming (C++, Java, C Sharp & most other commonly used languages) has taken over the software world is that it provides a number of benefits which improve productivity, reliability, maintainability and reusability.

If you Google the advantages of object oriented programming, you will find innumerable lists compiled by various software experts, architects and general users. Everybody has their own list, but almost everyone agrees that there is a long list and that object oriented programming is a major advance in the software industry. We have summarized what we believe to be the high points, as associated with control applications, below

- **Structure** : Application of object oriented programming promotes program development in a structured manner. In fact, to a large degree, it forces structure. Structure, in turn, enhances design productivity, maintainability and documentation.
- **Modularity** : Objects are program components that focus on specific pieces of the application. By definition they form program modules.
- **Partitioning of design focus** : Objects are self contained with a clearly defined interface. When developing an object, you are totally focused on that object alone, without the distraction of other aspects of the total system. That focus leads to higher levels of productivity.
- **Reusability/portability** : A huge benefit for anyone who develops multiple applications which have certain functions in common, is that objects can be re-used. Any object that you develop in Velocio Builder is automatically seperately stored in its own independent files. Those files are portable and can be reused in other applications. This is a huge factor in development efficiency, software quality and reliability. In addition, since multiple instances of the same object can be used in a program, object modules are reusable even in the same system.
- **Isolation/reliability** : Since objects contain (encapsulate) all of their required data and subroutines (functionality), they totally isolate a portion of the program. That isolation enhances debug and maintenance and leads to greater application reliability.
- **Maintainability** : System maintainability is enhanced due to the modularity and isolation of the object components. Maintenence is clearly focused only on those areas that need to be modified. Objects that require no modification do not need to be touched. The simplification of the application makes future maintenance, even by someone other than the original developer, much easier to perform and less susceptible to error.
- **Protection against unintended consequences** : In traditional PLC application programs, all memory is global (accessible everywhere) and the program is just one long sequence of program rungs. That frequently leads to situations where a developer is trying to work on one area of the program causes something unintended to happen in another area. By encapsulating data with functionality, objects prevent that from happening.

If this all sounds high minded to the point of being ovewhelming, don't worry about it. vBuilder makes object oriented programming very easy to do. The following pages will explain.

Object Subroutines

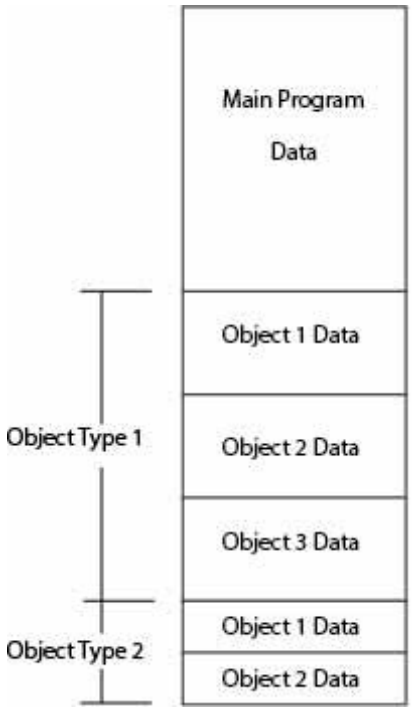
Let's start with the basics. We need to understand subroutines. We need to understand objects. We need to understand how these concepts fit together. Let's start with a couple of short definitions.

- A subroutine is program that is called by another program to perform some special function. It is a way of logically breaking up a larger program into smaller, logical blocks that performs specialized functions. A subroutine is programmed the same way as the main program. The difference is that it can have information passed into it by the program that calls it and it can be designed to pass information back to the program that called it. Another feature of a subroutine is that it can be called from multiple places.
- A program object is like a component. Think of a wheel on an automobile. The wheel is an object. Similarly a program object is a software component that is clearly defined and "encapsulated". It has clear boundaries and a clear interface. Just like a wheel is a clearly defined component of a car, with a clear function (to support the vehicle and roll), and has a clear interface with the car's hub and mounting bolts, a program object has a clear function that you define and a clear interface of data that is passed in and out. All other data that is used by the program object belongs to the program object and is not accessible by any other part of the program.

The following description may appear to be a little abstract at first. Don't let your eyes glaze over. Its short & you'll understand it soon. We'll get into examples that make it make sense soon, but we have to lay out the basic principles first.

All subroutines in vBuilder are object subroutines. What that means is that they have there own section of memory for their own private data. A vBuilder Object is a set of data plus defined functionality. The functionality is the subroutine program functionality. There can actually be more than one subroutine associated with an Object, but we'll get into that later.

The first time you create a subroutine in a project, you create an object. You can define tagname variables for that subroutine, just like you can for the main program. You'll notice that when you define the subroutine tagnames that you don't see the tagnames for the main program or any other subroutine. That is because you are defining the data that this object works with. Any data that comes into the object from the program that calls it or is sent back, is defined in the Subroutine Inputs and Outputs block.



Look at the figure on the right. This is an illustration of how memory is set up in the PLC. The main program has its data, which includes all of the IO and any tagname variables that are defined for the main program. The figure shows a program that has a main program and two types of Objects. These are two subroutines. The first subroutine has three "instances". The second subroutine has two instances. The first subroutine, or object type has more local data than the second. Each "instance" has a seperate set of data.

Objects are generally created to relate to the control of something real. In this case, consider this to be a building control and security system. The building has three main work rooms and two bathrooms. To control it, we might create an object called "workRoom" and a second object called "bathRoom". The workRoom objects would each have a thermostat, ventilation controls, window security, smoke detector, door security, lighting control and audio controls. There would be tagnamed data defined for each of these items, which would constitue the Object Data. The bathroom would have ventilation controls, lighting controls, automatic periodic flush controls and hand dryer controls, constituting its Object Data.

We have three workRooms that all operate the same way, but are likely all in different conditions at any given time. Therefore we need three seperate Objects. Each Object has the same set of data, but each Object's data is seperate from the others, because it needs to hold different status and values and it must not be affected by what is going on in the other rooms. The same for the bathrooms. So that's what we create with our Object Subroutines. We have two subroutines, one for workRoom and one for bathRoom. WorkRoom has three instances or sets of object data. BathRoom has two instances, or sets of object data.

There are a lot of interrelated topics included in vBuilder object subroutine programming. Here is a general list -

- subroutines
- objects
- embedded objects
- multiple instances of objects
- linked subroutines

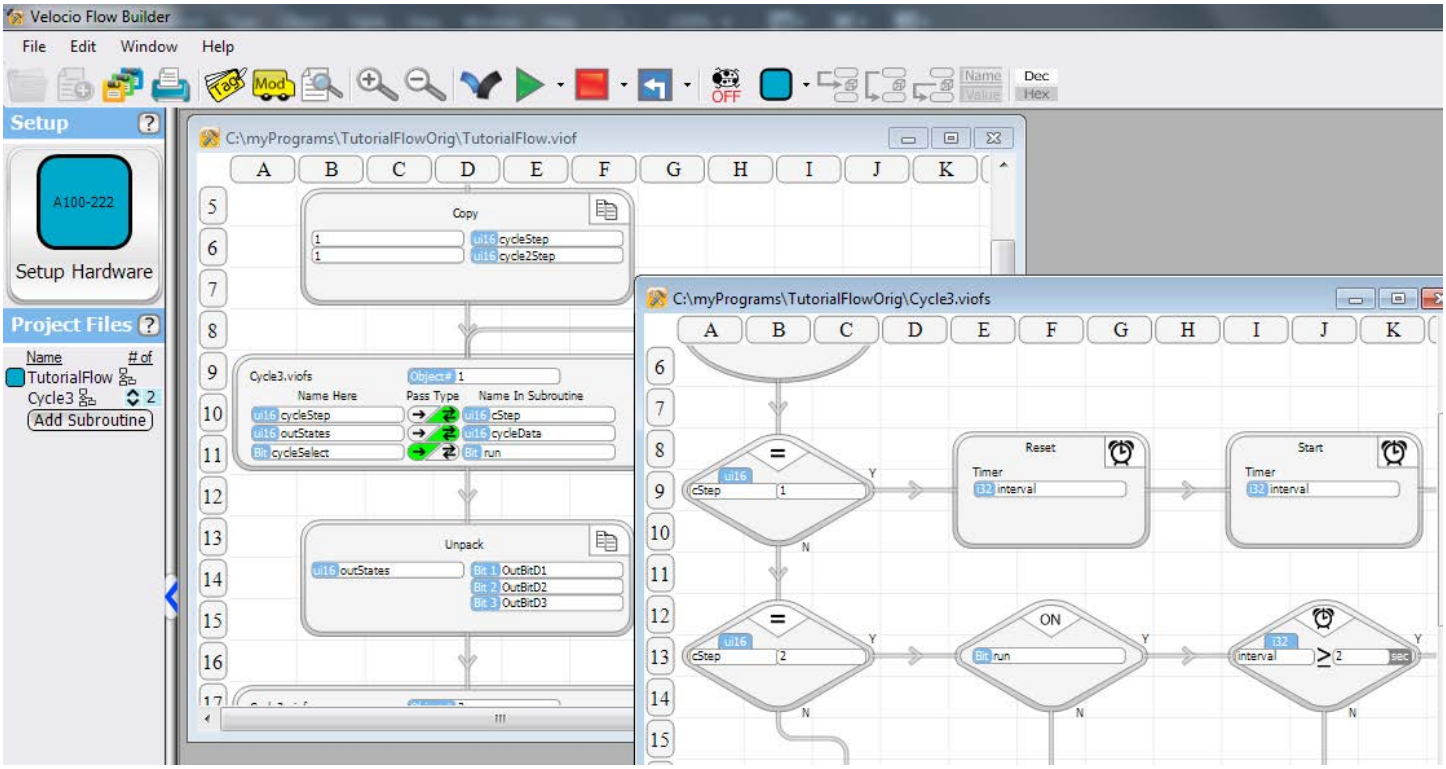
We'll start simple. In fact, if you went through the Tutorials, you've already written a program with object subroutines.

Simple Object Subroutine Example

Tutorial 2 in Chapter 2 goes through an example that includes object subroutines. That example consists of the main program and one object subroutine. The final implementation of that tutorial uses two instances of the object. Let's look at it closer to get an understanding of object subroutines. First, go back to the chapter 2 tutorial and review it.

This is the Flow Chart version of Tutorial 2. From this screen shot, you can see some of the fundamentals.

- The main program is TutorialFlow
- There is one subroutine, Cycle3
- There are two instances of Cycle3 (see the '2' next to Cycle3 under Project Files)
- Both the main program and subroutine are located in the main PLC (actually, this is an Ace - there are no other PLC units in the system)



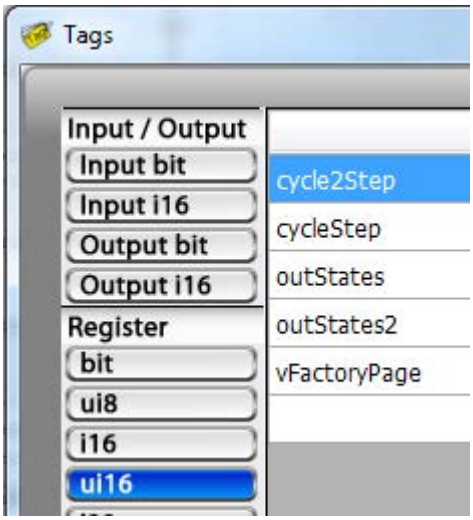
If you have not entered TutorialFlow, go back to chapter 2 and go through its entry and debug. We're going to use it to explain object subroutine concepts.

Look at the main program, TutorialFlow. TutorialFlow starts by initializing data. After initialization, TutorialFlow goes into a super loop.

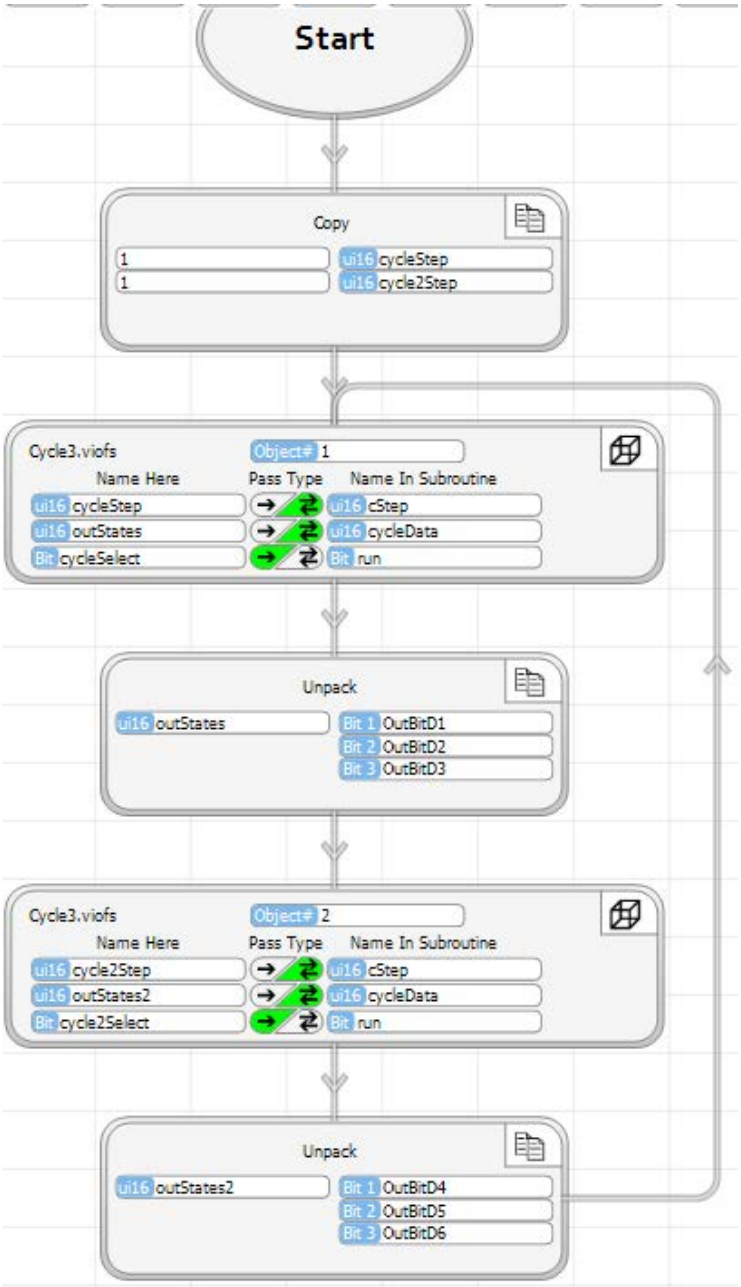
The initialization Copy block initializes step (state) counters for each of the objects to 1. The actual tagnamed data for the Cycle3 subroutine is located in the main program's memory. If you look at the subroutine calls, you can see that a reference to the step counter is passed in.

The super loop calls each instance of Cycle3. Cycle3 will handle determining the required output states of 3 outputs. After the Cycle3 call, the output states are Unpacked to the actual outputs.

Take a closer look at the subroutine calls to Cycle3. Notice that the first call is a call to Object 1 and the second call is to Object 2. Both call the same subroutine, but they pass different data. If you look at the ui16 Tags, you can see that that there are seperate tagnamed variables for cycleStep/cycle2Step and outStates/outStates2.



In the call to Cycle3 object 1, a reference to cycleStep is passed to cStep. In the call to Cycle3 object 2, a reference to cycle2Step is passed to cStep. Inside Cycle3, cStep is what is used. The actual data for cStep is in the main program's tagnamed variable whose reference is passed in. The same applies to outStates/outStates2 and cycleData.



The illustration on the right shows the Tag memory for this program. A careful review, along with a good look at the program should provide a lot of insight into object subroutines.

The program data is divided into three segments. The main program - TutorialFlow - contains all of the Input and Output data, plus all of the tagnamed variables that are defined for TutorialFlow. The ones that are used in this program are shown at the top of the illustration.

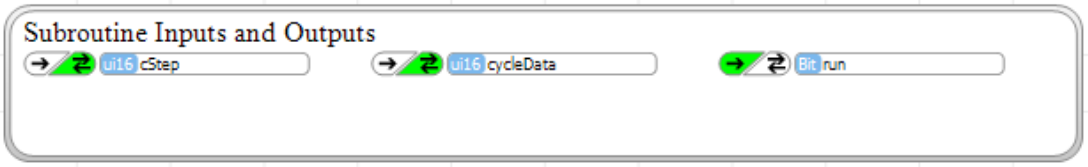
The main program always contains only tagnamed data. There are no references, because references are passed in by a calling program. Nothing calls the main program.

The two 'instances' of Cycle3 each have their own set of data. The data for each is identical. However the data values and references can be different. That way, the individual objects can have the same behavior, but may be in different states at the same time.

The subroutine, Cycle3 has three items defined as Subroutine Inputs and Outputs. The first two, cStep and cycleData, are passed by reference. That means that there is no actual data passed to the subroutine. What is passed is a "reference". A reference is information on where to find the data. The actual data is in the main program data. Since the program calls to the two different objects pass different Tags as references, the two objects use different ui16 tagnamed variables in the main program for cStep and cycleData.

The third Subroutine Inputs and Outputs item is passed by data. That means that an actual value is passed in. In this case, Cycle3 object 1 is passed the state of cycleSelect (B1), which is placed in object variable 'run'.. Cycle3 object 2 is passed the state of cycle2Select for its object variable 'run'. The data that is passed in is stored in the object data for the instance of the object.

Main program data	cycleSelect	data
	cycle2Select	data
	outBitD1	data
	outBitD2	data
	outBitD3	data
	outBitD4	data
	outBitD5	data
	outBitD6	data
	cycleStep	data
	cycle2Step	data
	outStates	data
	outStates2	data
Cycle3 Object 1 data	run	data
	interval	data
	cStep	reference
	cycleData	reference
Cycle3 Object 2 data	run	data
	interval	data
	cStep	reference
	cycleData	reference

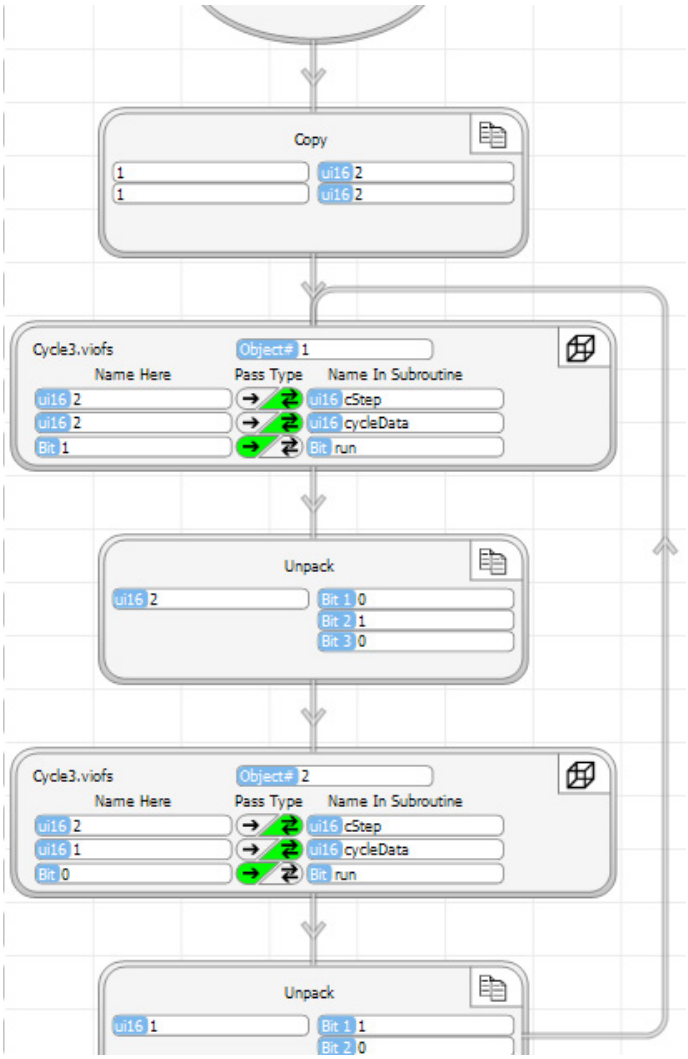


Each time the Subroutine Call to Cycle3 is made, the two references, along with the current data passed to 'run' is passed to the Subroutine.

Program the PLC. Put it in Debug mode. Select Values display. Click Run to start the program.

Flip the B1 input (cycleSelect) on, while leaving the B2 input (cycle2Select) off. You should see the first three outputs cycle. Outputs 4 though 6 should not be cycling.

If you look at the main program, you should see that the value of 1, which is the state or value of the cycleSelect input is being passed to Cycle3 object 1's run bit. The value 0, which is cycle-2Select's bit value when the B2 switch is off, is being passed to Cycle3 object2's run bit. Consequently, even though the same subroutine is called in both cases, object 1 is cycling the outputs while object 2 is not. You can see this happen with the cycleData that is changed by the subroutine and unpacked in the Unpack blocks right after the Subroutine Calls.

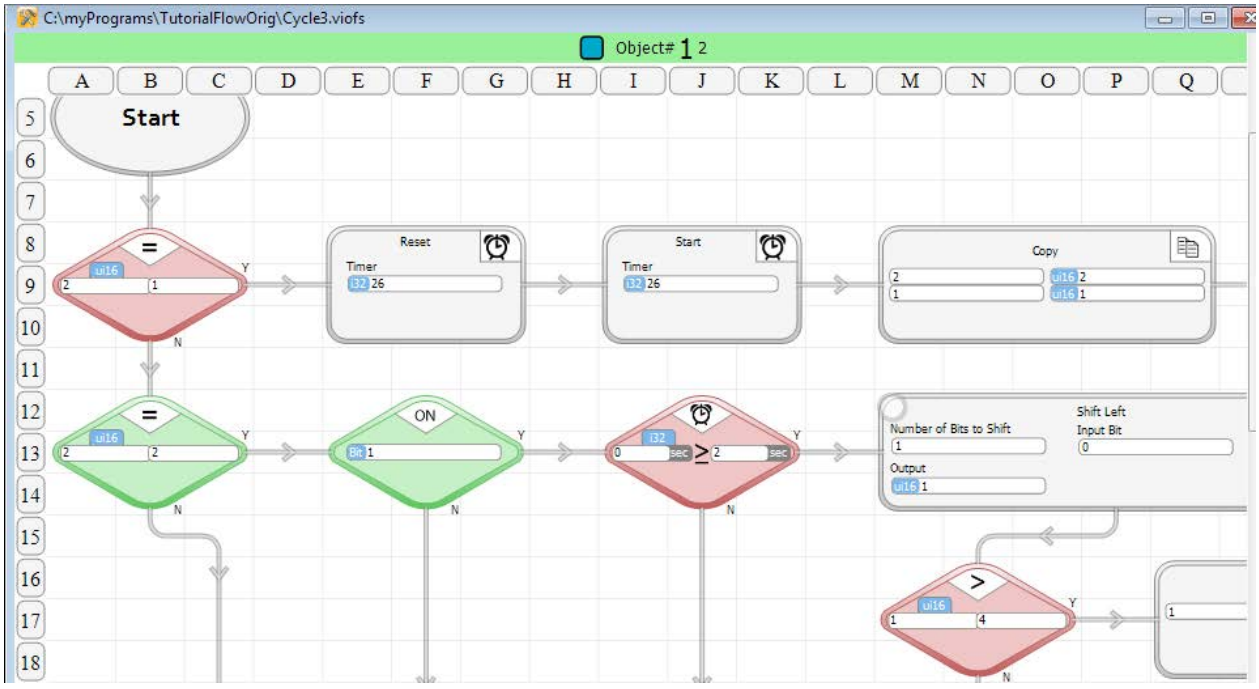


Take a look at Cycle3. Notice that at the top of the Cycle3 program window, there is a blue square followed by the word Object and numbers 1 and 2. All of that provides information that is important when debugging a program with object subroutines.

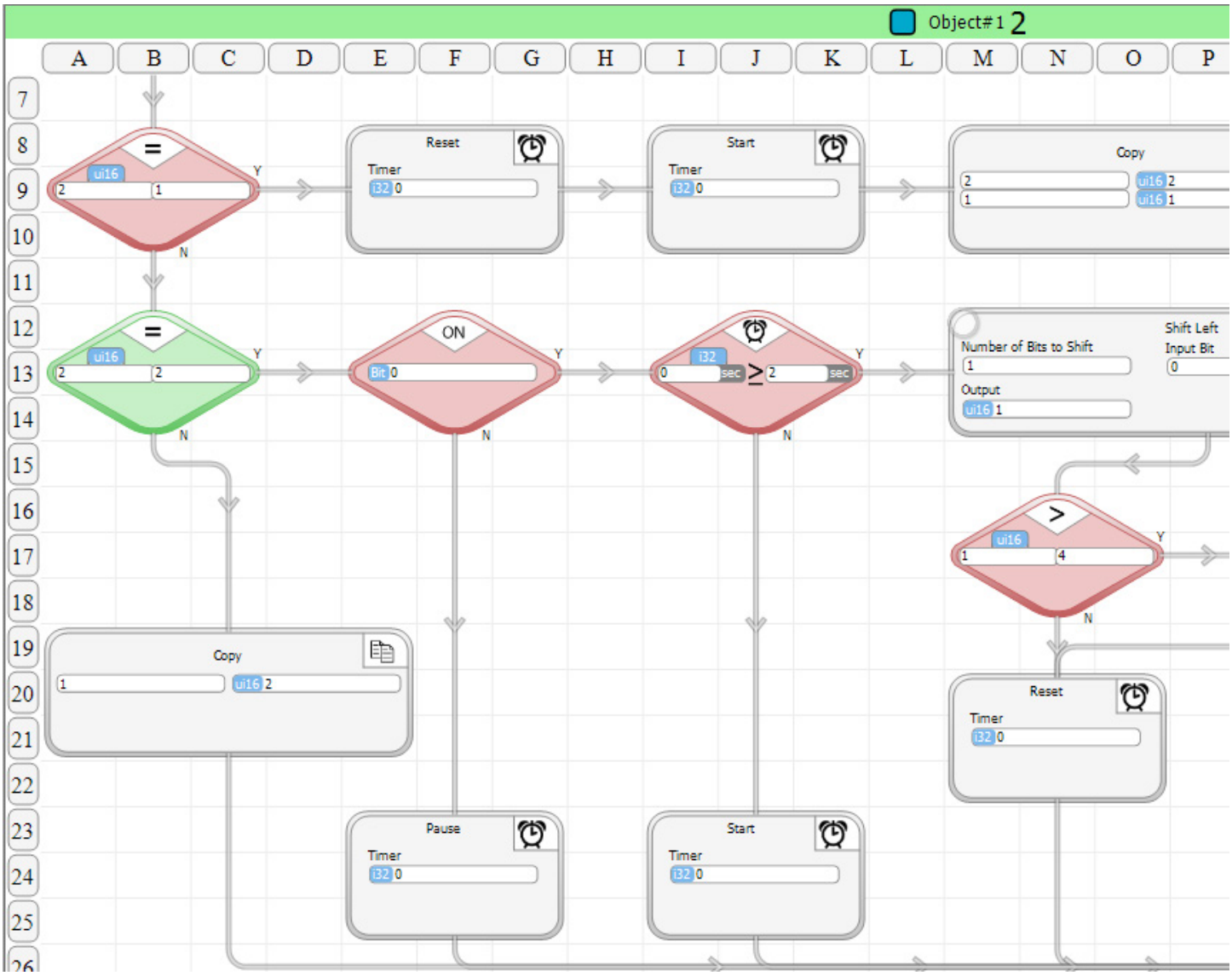
- The colored square shows in which module the routine resides. Currently, there is only one PLC module in this application program, so the square indicator is blue with no number. When we get to embedded objects, there may be other indicators.
- Object # 1 2 informs you that there are two objects associated with this subroutine.
- The fact that the '1' is larger and bold indicates that the "focus" is on object 1. The data display and decision block coloration apply to the object that is in focus. The other object may have different information.
- If you were to click the '2', the focus would change to object 2 and the information display would change to object 2's information.

Watch Cycle3, object 1. Notice that while run (which is the passed value of cycleSelect (B1)) is 1, the interval timer will time up to 2 seconds. When it reaches 2 seconds, the Timer Compare decision block (located at J13 below) determines that it is Greater Than or Equal to 2 seconds and causes logic flow to proceed along the Yes path to the Shift Left block. You should see the cycleData step from shift from 1 to 2 to 4 and back to 1. The back to one doesn't occur in the shift block. It occurs due to the comparison Greater Than 4 decision block and the Copy on the Yes branch, which follows.

Switch back and forth between Name and Value modes so you clearly understand what you are observing.

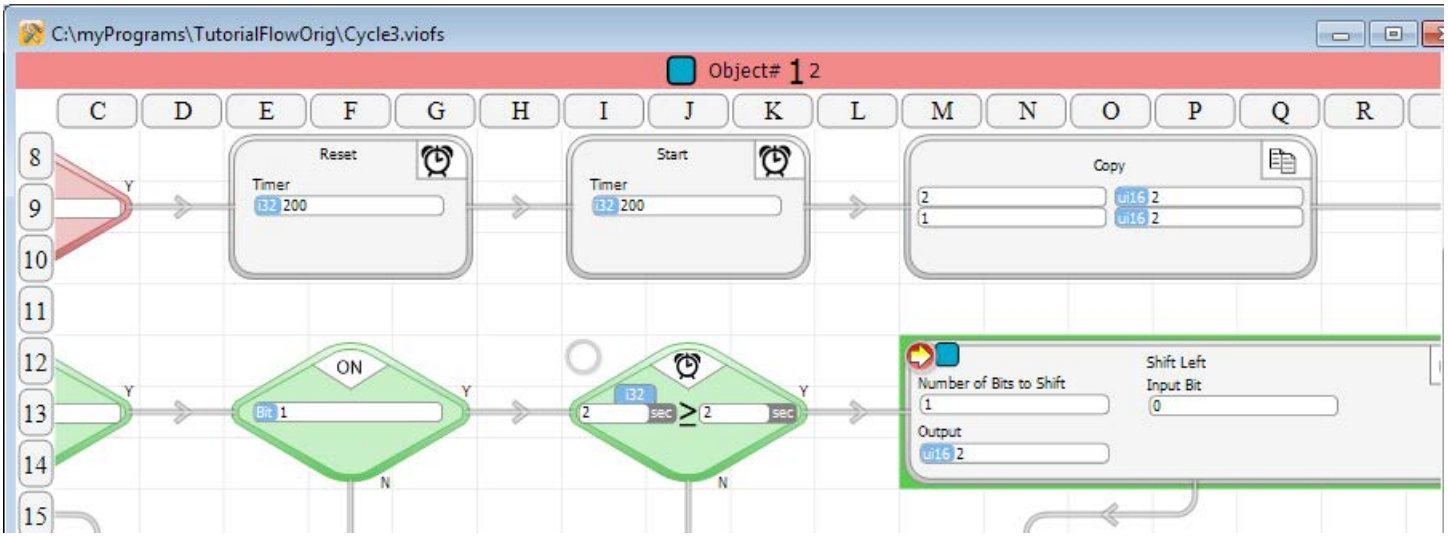


Select object 2. If you have not switched cycle2Select (B2) on, you will see the display like that shown below. The 'run' (passed from cycle2Select) has never been 1. cStep 1 did execute, which Reset and Started the interval timer. In cStep 2, the run bit is never ON (or 1), so the program Paused the interval timer. This happens so fast that the timer value doesn't have a chance to increment.

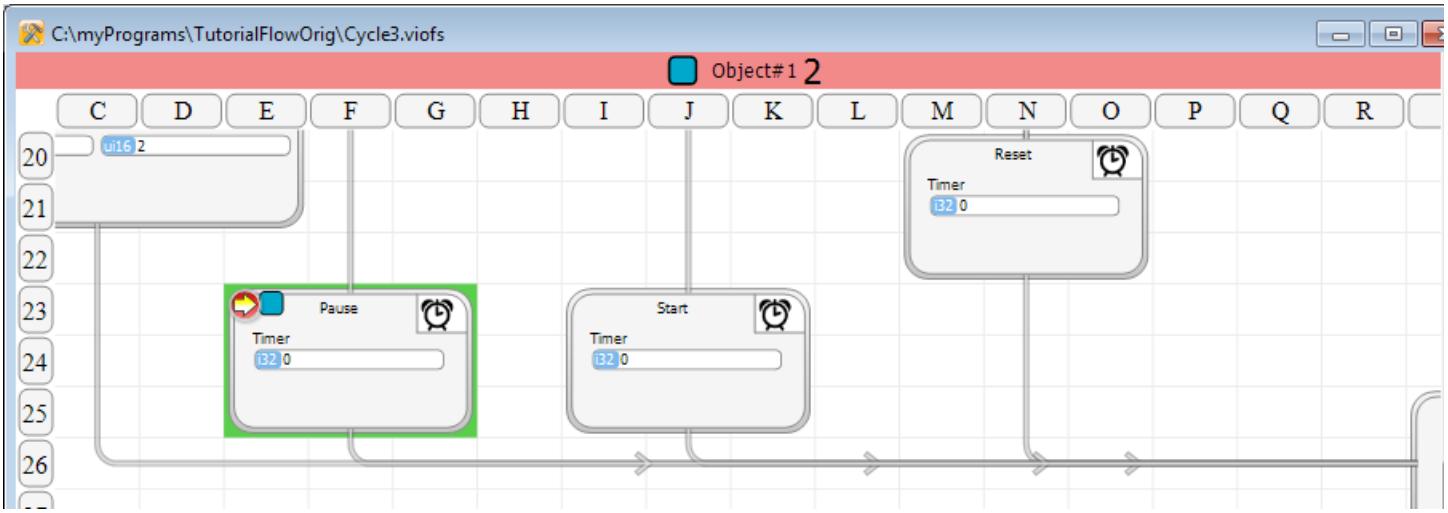


Switch back and forth between the two objects. See how differently they are executing.

Select object 2, then place a breakpoint at the Shift Left block (while Running). The program will break when that block is reached, regardless of which instance. You should see the program stop and the object number switch to 1, as shown below. Breakpoints apply to the subroutine in the selected PLC. They do not differntiate between objects. When the program hits a breakpoint it will change the display to clearly show in which object the breakpoint was reached.



To further illustrate, clear the breakpoint and click Run again. Place a breakpoint on the Timer Pause block. This time, the program will stop at the Pause block and change the display to show object 2.



Turn the cycle2Select (B2) switch on. You should now see both groups of three outputs cycling. They will very likely be cycling at different times. The Cycle3 subroutine was created with one timer - interval, which determines the time to Shift the outputs. The interval timer is part of the Cycle3 object, so each instance of Cycle3 has its own interval timer. If you look at the data illustration a few pages back, you should see the to instances of interval clearly illustrated.

Linked Object Subroutines

In vBuilder programming, an object consists of data plus behavior. Subroutines define "behavior". They are the logical operations that are performed on the object data. Just like real world entities, vBuilder objects can be defined to have different behavior at different times. This behaviour is often most clearly and efficiently deliniated by the creation of different subroutines, tied to the same object data. Linked Subroutines are any set of subroutines that utilize the same object data.

---

vBuilder object data is a separate package of data for each instance of each type of object. That concept was clearly illustrated in the last few pages. If you don't clearly understand object data and instances of object, please review the pages of this chapter, prior to this point, before proceeding.

Any number of suroutines can be linked together as part of an object. By employing seperate subroutines for each type of behavior, program efficiency and clarity can be greatly improved. A typical object might have linked sutboutines for the following purposes.

- Initialization (setting up initial data, plus the acquisition of references used in operation)
- Normal operation
- Special case operation (there many be more than one of these)
- Shutdown

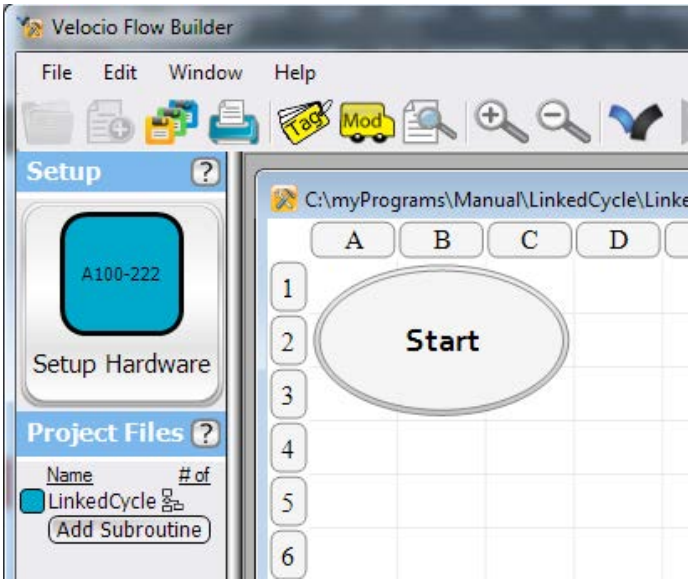
There many other reasons and strategies for creating linked subroutines. Let's get into the concepts through an example.

The example that we will go through is a program that does the same thing as the tutorials that cycle the outputs -Tutorial Example 2, discussed in both chapter 2 and earlier in this chapter. We'll accomplish the same program operation, but do it with linked subroutines.

If you have looked at Tutorial Example 2 recently, you know that the main program handles the actual I/O processing, while the Cycle3 subroutine handles the logical operation of controlling the cycling of the outputs, including timing. Actual I/O information is passed into Cycle 3 and passed back to the main program through references.

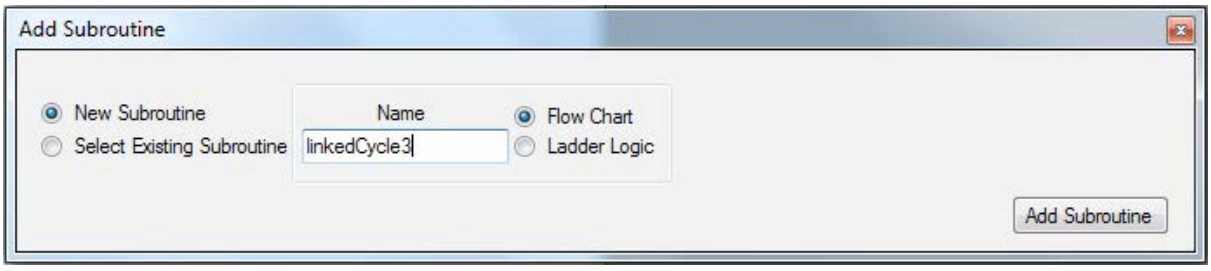
In this example, we will do things a little differently. Initialization will occur in an initization subroutine, linked to the linked-Cycle3 operatational subroutine. Rather than pass data from the main program to Cycle3 for processing then handling outputting the outputs in the main program, we'll do that in linkedCycle3. This will become clear as you go through the example.

Start by creating a new Flow Chart program, called LinkedCycle, as shown on the right.



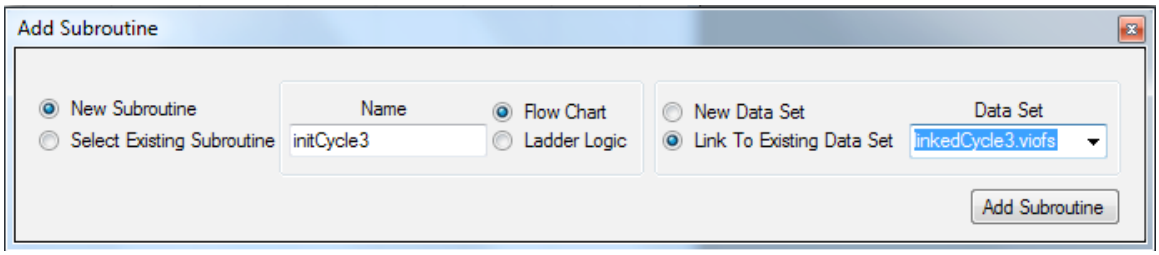
Next, select Add Subroutine and fill out the dialog box to add a New Subroutine named linkedCycle3, using Flow Chart programming. Click Add Subroutine.

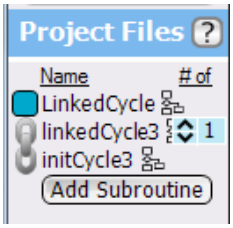
We're adding the main operational subroutine first, simply because the first subroutine of a newly created object becomes the name of the object.



Next, select Add Subroutine again.

When the dialog box comes up, select "New Subroutine". Enter "initCycle3" in the Name box. Select "Link to Existing Data Set". On the far right, a new entry box, labeled "Data Set" will pop up. The selection box will make available all of the objects types that currently exist, for you to choose from. So far, we only have one object type, called linkedCycle3. Select it and click Add Subroutine.





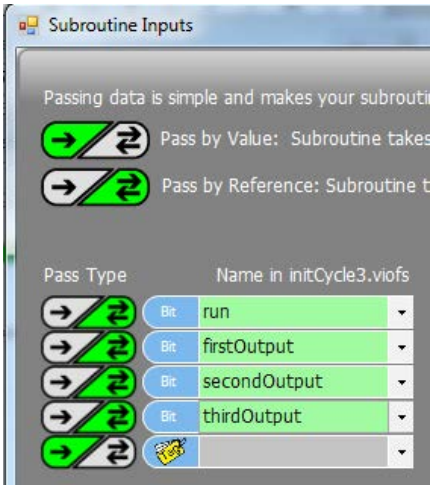
The listing under Project Files will now appear as shown on the right. The main program - LinkedCycle - is shown as residing in the main PLC (the only one necessary for this example). Under it is linkedCycle3. Next to linkedCycle3, the object count adjustment shows that this object is named by the "linkedCycle3" subroutine to its left and there is currently only one object defined. Adjust the number of objects to 2.

Below linkedCycle3 is initCycle3. The chain links that start at linkedCycle3 and end at initCycle3 show that all subroutined within this range are linked to the same object.

Double click to select the Subroutine Inputs and Outputs in initCycle3. Enter the four pass by reference bits shown on the right.

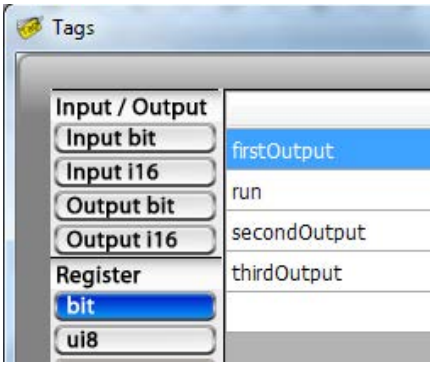
What you are defining to pass into the linkedCycle3 object, when initCycle3 is called, is references to one digital input and three digital outputs. The digital input will serve as the run switch, which will enable or disable the cycling of outputs. The outputs are the three outputs which will be cycled.

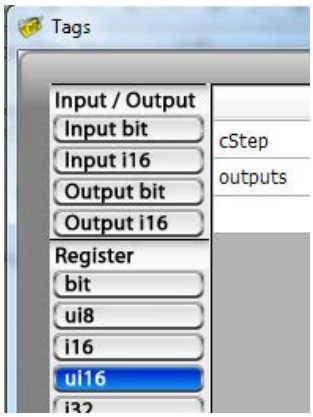
Click OK.

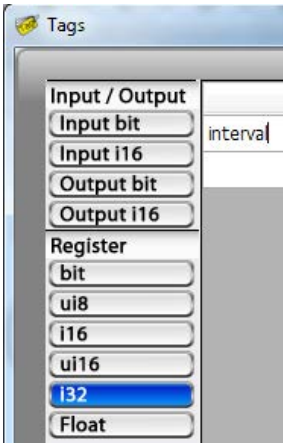


Open up the Tags (while initCycle3 is still selected). You should see the four tags that you just defined as bits passed in by reference, listed as register bits. As far as the linkedCycle3 object subroutines are concerned, these are simply bits that they can use.

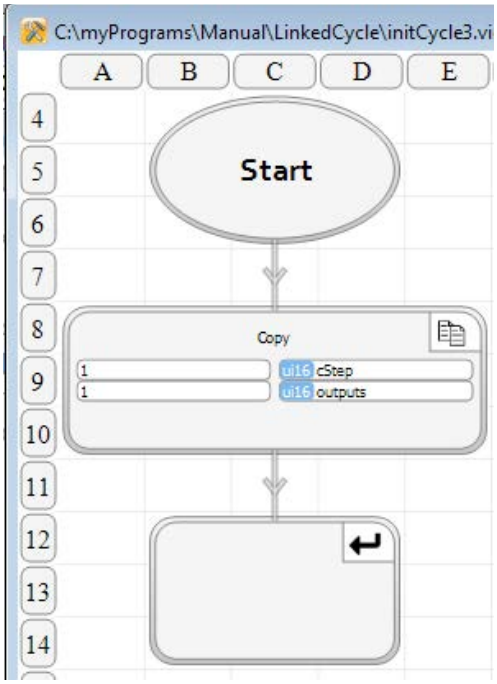
While the Tag dialog box is open, create 'cStep' and 'outputs' as ui16 tagnamed variables and 'interval' as an i32.







Now, let's enter the initCycle3 subroutine program. Enter the program shown on the right.

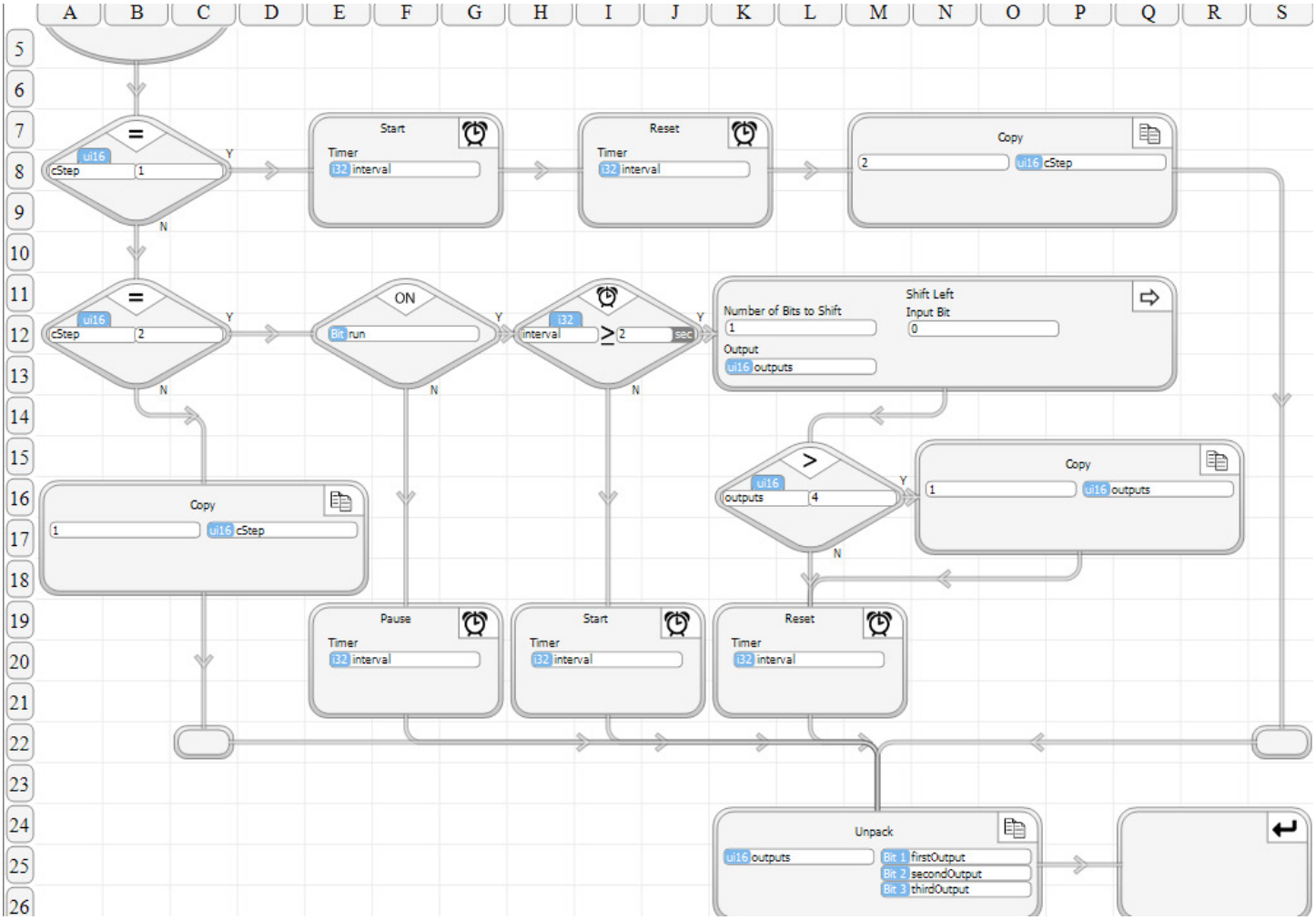


When initCycle3 is called by the main program, it will initial everthing that linkedCycle3 needs initialized. An important aspect of the initialization is actually not in the program logic of the initCycle3 program. It is in the Subroutine Inputs and Outputs. When initCycle3 is called, references to the input used for run and the three outputs are passed into the object. They are thereafter there to use by any of the linked subroutines. By including the reference passes in the initialization routine, these references never need to be passed in again.

The next thing that initCycle3 does is initialize the step or state counter, cStep to 1 and set the initial state of the 3 outputs to outputting the first output on and other two off. This subroutine doesn't actually write to the outputs. The linkedCycle3 subroutine will.

The last block is just a Subroutine Return.

Select and enter the linkedCycle3 subroutine. There are no Subroutine Inputs and Outputs required, so leave that blank. Enter the subroutine as shown below. Everything after the Start block is shown. The flow is very similar to the previous example. A major difference is the fact that this subroutine now directly accesses the inputs and output through the references that were passed in its linked subroutine, initCycle3. An unpack block at the end of the subroutine writes to the actual outputs.



Now select the main program (LinkedCycle) and enter it as shown on the right.

As you can see, this program calls initCycle3 to initialize information used in linkedCycle3. What it passes is references to the IO that is used by each object. Since linkedCycle3 and initCycle3 are linked, any data that is passed into one is available to the other.

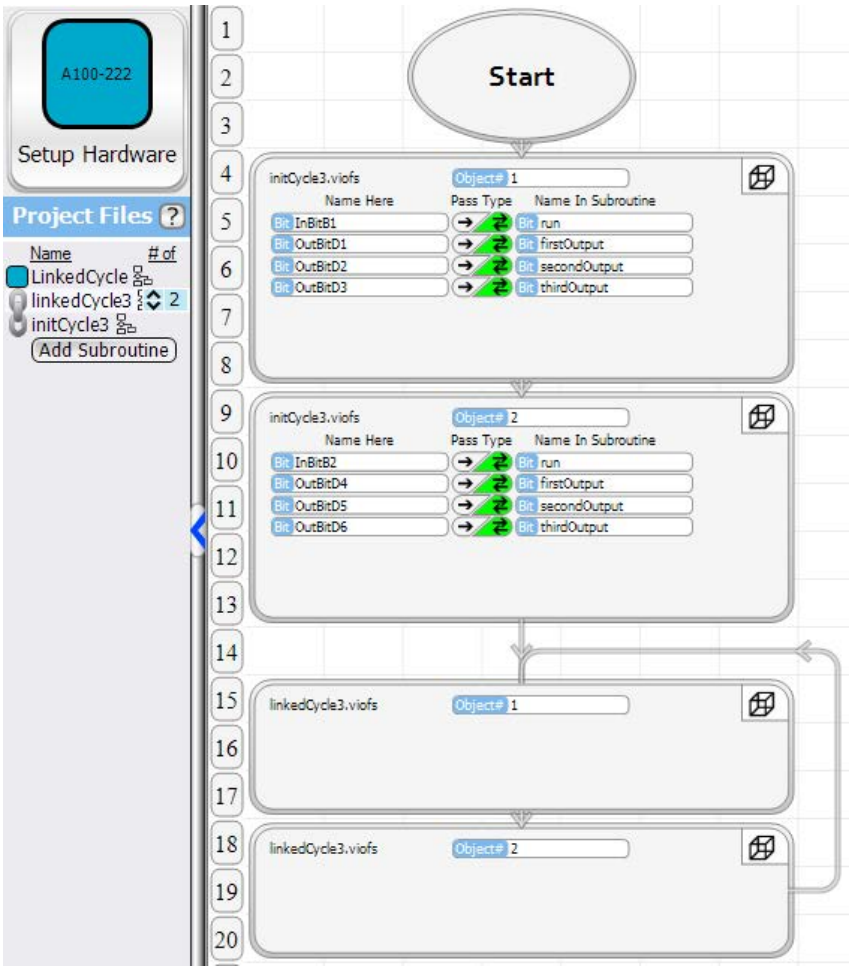
After initialization, the program simply loops through calls to the two objects. No data is passed in because all of the necessary data is already contained in the object data.

We're ready to run. Before programming & running, check a couple of things.

- Check that the program that you have entered is exactly as shown on the previous pages. If it is not and the program doesn't run the way you expect, you will simply be able to use your debugging skills.
- Make sure that the number of objects is set to 2.

Program it and run the program. Put it in Debug mode and watch what the program does.

You can make adjustments to see how the program handles them. One thing to try is passing in different inputs or outputs. The program linkedCycle3 program should use whichever input bit that you pass in to its run. It should cycle through whichever set of outputs that are passed in through initCycle3.



8. Embedded Objects

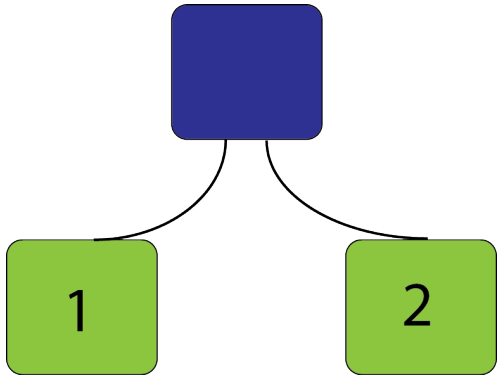
Embedded Objects are an absolutely unique feature of Velocion PLCs, which enable tremendous power, flexibity and system integration. They provide transparent multiprocessing and the ability to program an entire system of interlinked PLCs as an integrated program. They are so unique that they need a concept introduction before we get into their application.

Traditional PLCs have one CPU that executes the application program. IO modules can be added, sometimes in a modular fashion - generally with a backplane bus. Velocio can do that as well, with greater flexibility, through vLink communications.

With traditional PLCs, if you have multiple PLCs that need to share information, or have PLCs that take commands from other PLCs, you must set up communication links or networks. This tends to be an extremely complex, time consuming and error prone undertaking. In addition, debugging such a system generally requires multiple development systems that are not linked.- an unwieldy undertaking.

Velocio PLCs were designed with multiprocessing as a fundamental feature of the architecture. They are designed to provide machine to machine sharing of data, with multiprocessing program execution and do so transparently.

Let's start with the basics. Say that you have a system with one Branch PLC and two Expansion PLCs. Each Expansion is connected directly to the Branch through a vLink communications cable. This configuration is shown on the right.



With this configuration, the Branch Expansion units could simply provide expansion IO for the Branch. The entire program operation could occur in the Branch PLC. In many cases, that is what you want.

Consider the following, very common scenario.

- Each of the Expansions is located at a seperate machine and contains all of the IO for that machine.
- The electrical power to each Expansion is supplied by the seperate machine that it interfaces.
- It may be desirable for the machine to operate even when the machine that the Branch is connected to is off.
- When everything is operating, it is desirable for the Branch PLC to coordinate the independent machines into an integrated system with machine to machine transfer of pertainent information, in real time.

In this scenario, the ideal solution would be an implementation that places the program logic in each individual PLC (a program in the Branch and programs in each Expansion), with data passed between the Branch and Expansion units. Ideally, this data transfer would occur many times per second, so real time control/coordination and interdependent operations can occur.

That is what Velocio's Embedded Objects are designed to do. That is just one of a myriad of scenerios to which they apply. Any time that direct, on the site control, coupled with system integration is required, Embedded Objects can provide the solution. Any time that a system can be subdivided into clearly defined subsystems, Embedded Objects provide a better solution.

As we've done throughout this manual, lets introduce the concepts, through a simple examples, starting on the following page.

Simple Embedded Object Example

To illustrate Embedded Objects through a simple example, we will build on the LinkedCycle example that was presented in the last chapter. If you have not been through that example, please do so before continuing.

We will go through this simple Embedded Object example in three phases. Each phase will add a little complexity and should provide additional clarity to the subject of Embedded Objects.

Phase 1 : Develop a program with one Branch and one Branch Expansion
- Cycle three outputs on the Branch based on the status of a switch in the Expansion.
- Cycle three outputs on the Expansion based on the status of a switch in the Branch

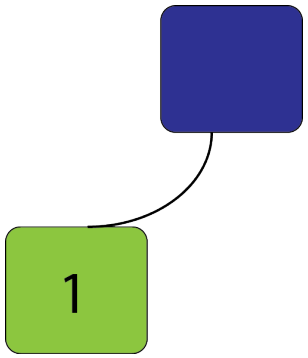
Phase 2 : Add a second Expansion
- Cycle three other outputs on the Branch based on the status of a switch in the Expansion
- Cycle three outputs on the second Expansion based on a second switch in the Branch
- Keep previous operation from Phase 1

Phase 3 : Add "heartbeat" monitoring
- Each of the Branch Espansionss monitor communications from the Branch and adjust their operation accordingly
- Branch unit monitors each Expansion and adjust its operation accordingly

By the time you get through all three phases, you should have a good handle on Embedded Objects.

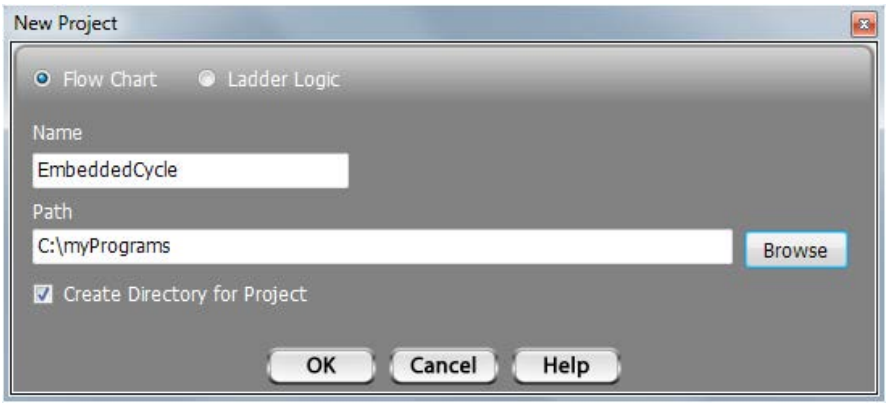
Phase 1 : Basic Single Embedded Operation

Start with a Branch and one Branch Expansion PLC. Connect a vLink cable from the Branch unit's expansion port 1 to the vLink input port of the Expansion unit, as shown on the right. Connect a USB cable from your PLC to the Branch. Power everything up.

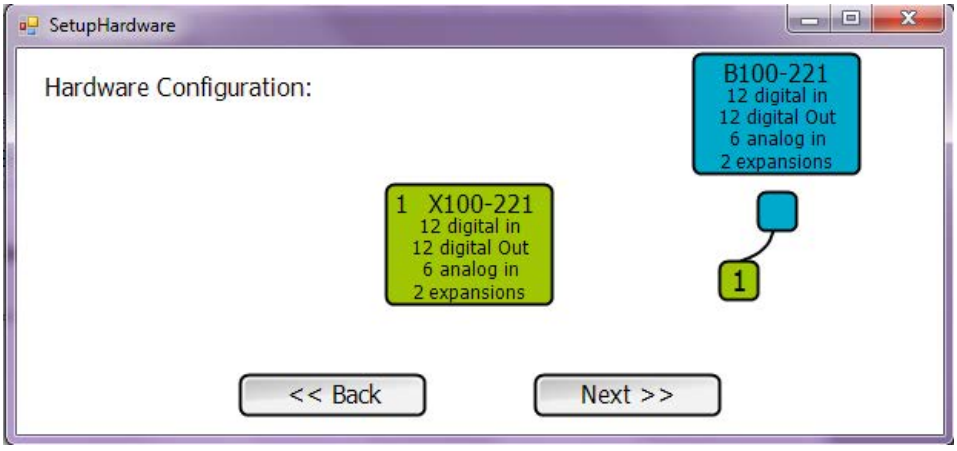


This is ideally done with two Velocio Simulators.

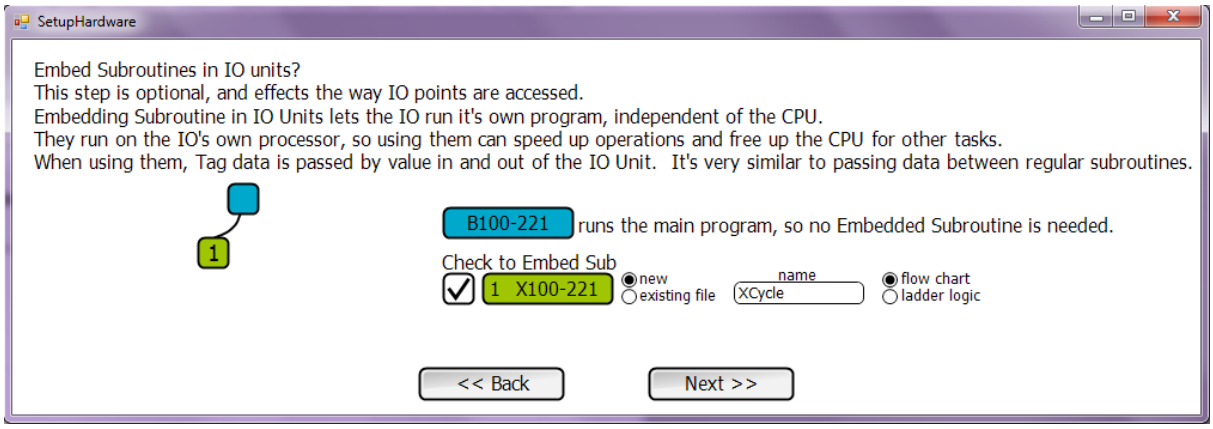
Now, create a new project.



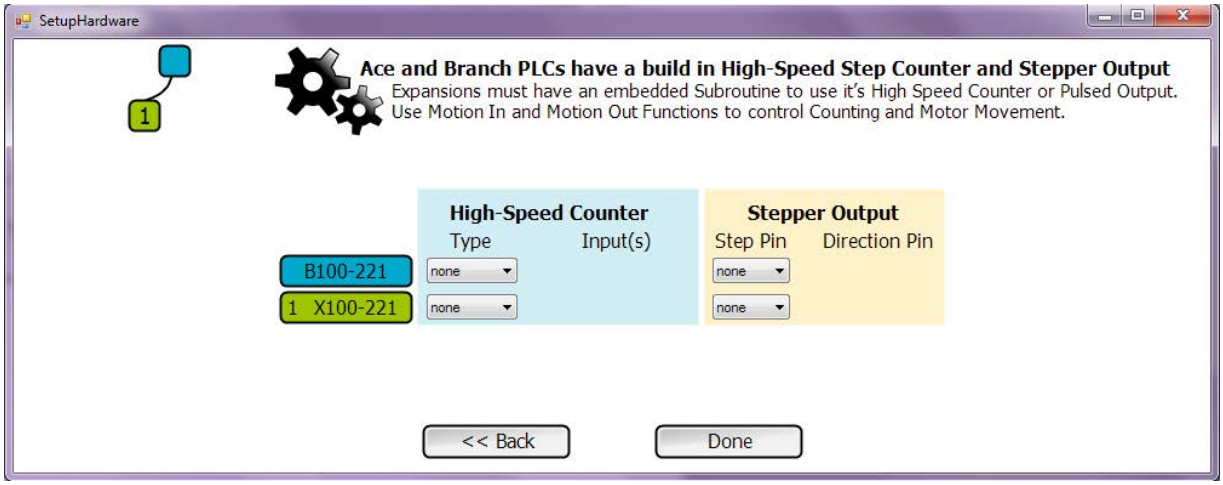
Select "Start Here" and begin configuring. If you have everything connected as described, you should be able to just Auto Configure. The Hardware Configuration should appear as shown on the right. Click Next.



On the next screen, see the checkbox under the label "Check to Embed Sub" and next to the green rectangle labeled '1' and the alphanumeric identifier of the exact unit located in the position connected to the Branch's port 1. Click to check the box. Leave the selection for a new program and enter XCycle as the program name. Leave the selection as flow chart. Click Next.

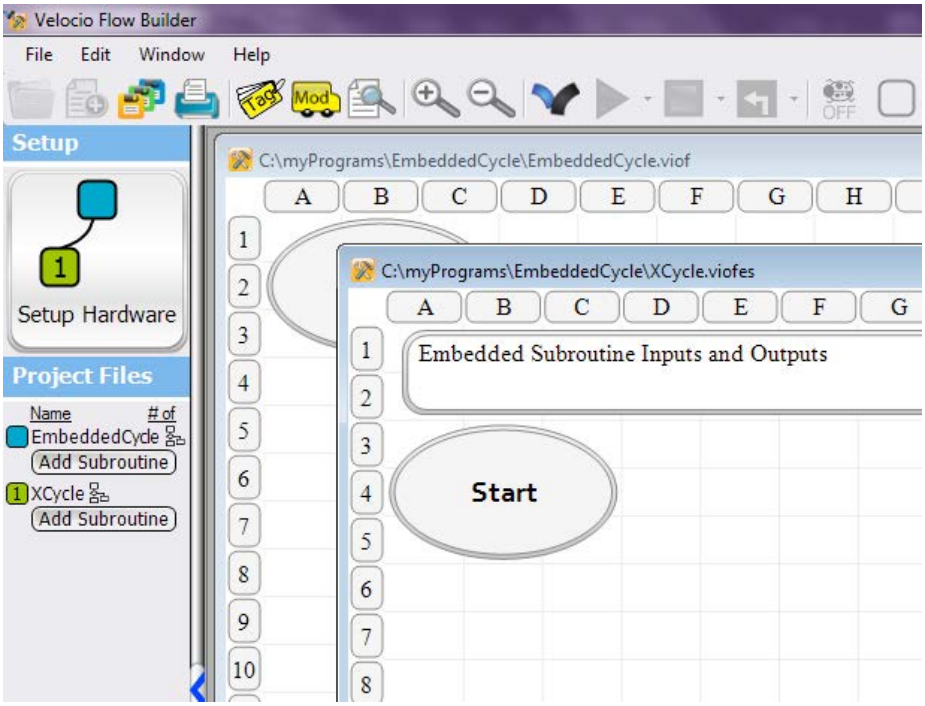


For the next screen, we don't have any high speed counters or stepper motors in this application, so leave everything "none" and press Done.



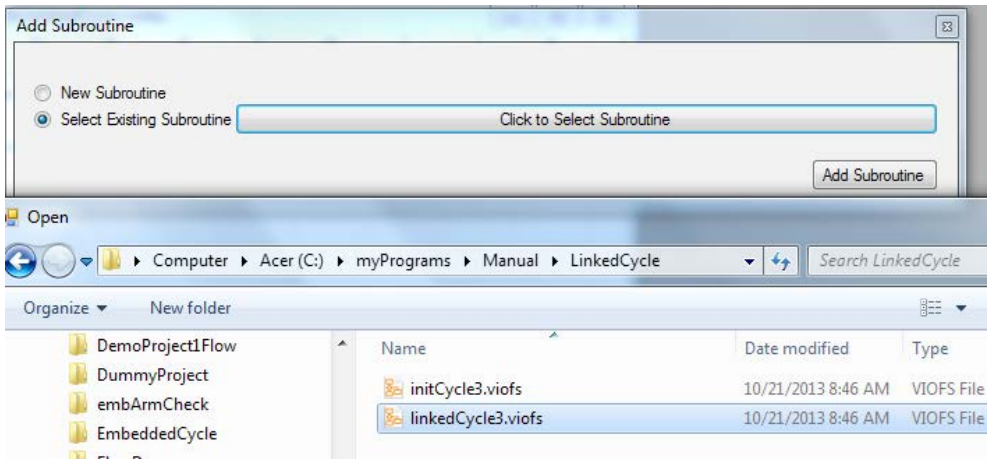
Your screen should now look similar to that shown on the right. The Setup Hardware window shows that you have a Branch (the blue square) connected to a Branch Expansion from the Branch's vLink port 1 (the green square labeled 1).

Under Project Files, there is a blue square. That indicates that all programs (project files) listed next to it and below it, up until you reach another colored square are located in the Branch unit. There is one program listed - EmbeddedCycle. The green square indicates that all of the programs listed next to or below it, until you reach another colored square are to be located in the green number 1 Expansion. Also, there should be program windows open for each of the listed programs.



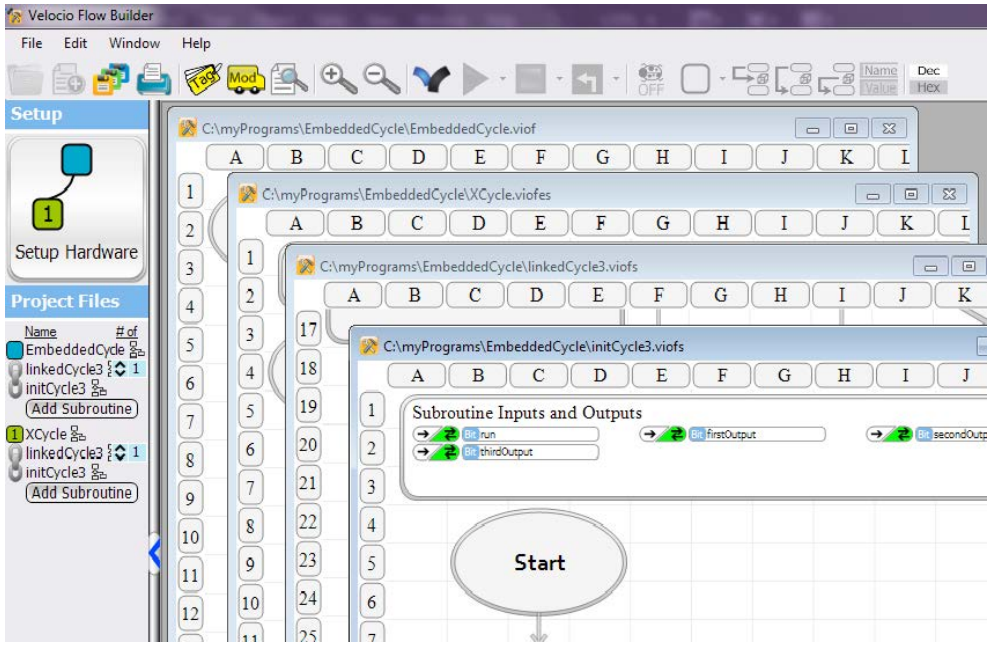
We're going to perform the cycling of three outputs in both the main Branch PLC and the Expansion. We're already written that program three or four different ways. This time, let's reuse the version that we wrote for the Linked Subroutines example.

Click the "Add Subroutine" button under EmbeddedCycle in the Project Files listing. In the dialog box that pops up, select "Select Existing Subroutine". Click the box that says "Click to Select Subroutine". Browse to the directory where you created LinkedCycle.



Select linkedCycle3.viofs. You must select linkedCycle3.viofs and not initCycle3.viofs, because the linked object was created as linkedCycle3. Linked subroutine objects are given the name of the first subroutine created. If they are copied to a new application, you must select the file that has the object name. Click Open, the Add Subroutine.

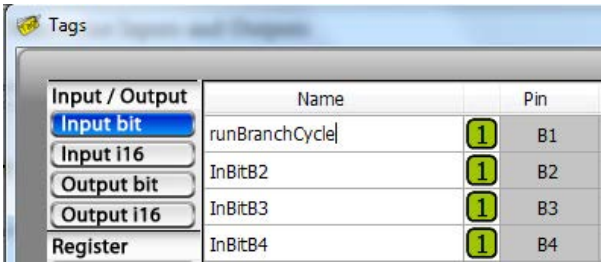
vBuilder will add the linked subroutines to the Branch PLC programs. This is shown at the right.



If you open Windows Explorer and browse to the directory that you defined for this application, you will see that the linkedCycle3 and initCycle3 files were copied to that directory.

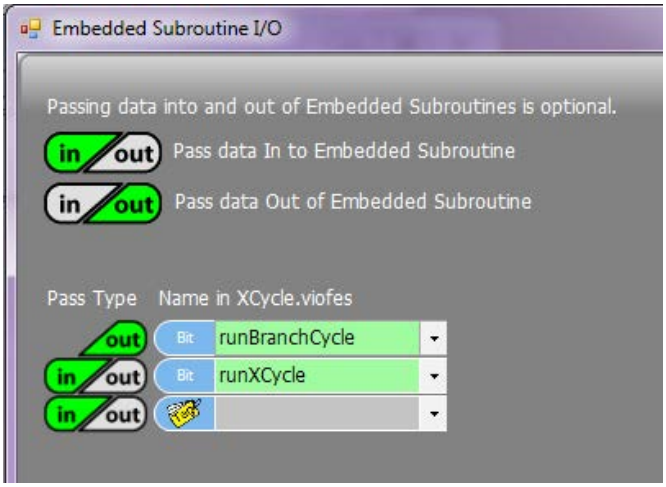
Add Subroutine again, this time under XCycle. Again, add linkedCycle 3. You should find it in your current directory this time.

Select the XCycle program window, then rename the Expansion unit's B1 input, runBranchCycle.

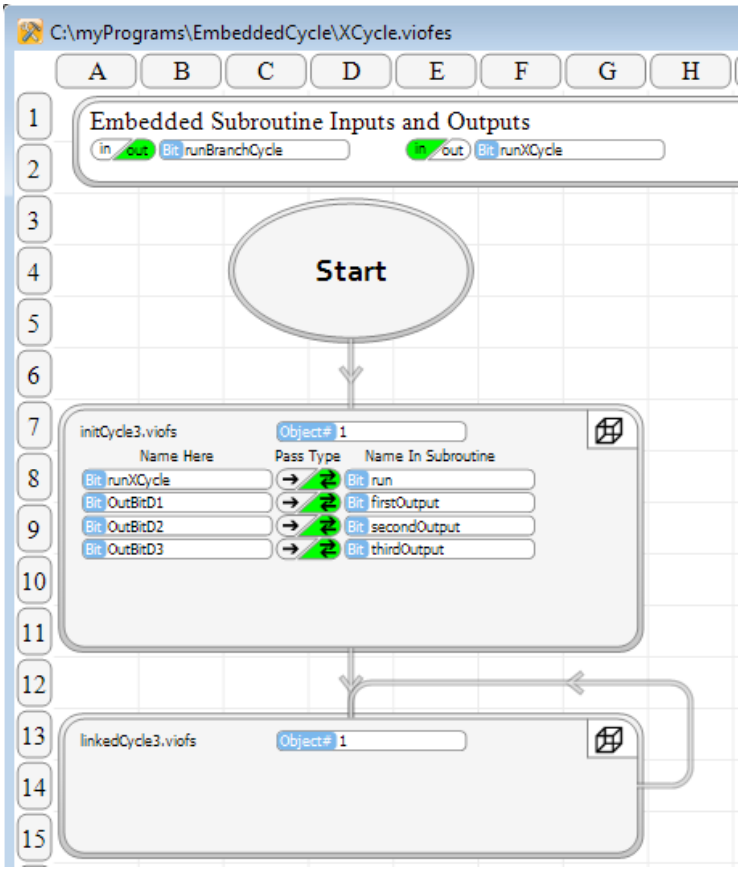


Double click the XCycle "Embedded Subroutine Inputs and Outputs" block and enter runBranchCycle as an output and runXCycle as a bit input.

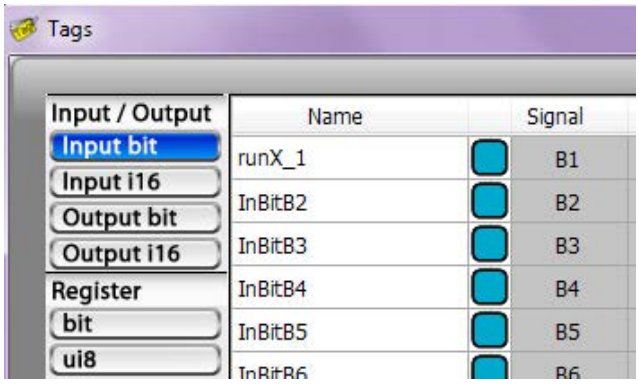
Notice that the Inputs and Outputs to an Embedded Subroutine are a little different than for a local subroutine. For Embedded Subroutines, data is passed in to and out of the Embedded device.



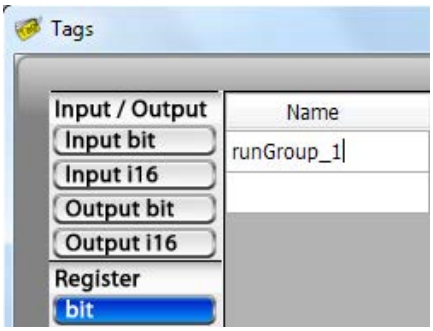
Enter the XCycle program as shown on the right.



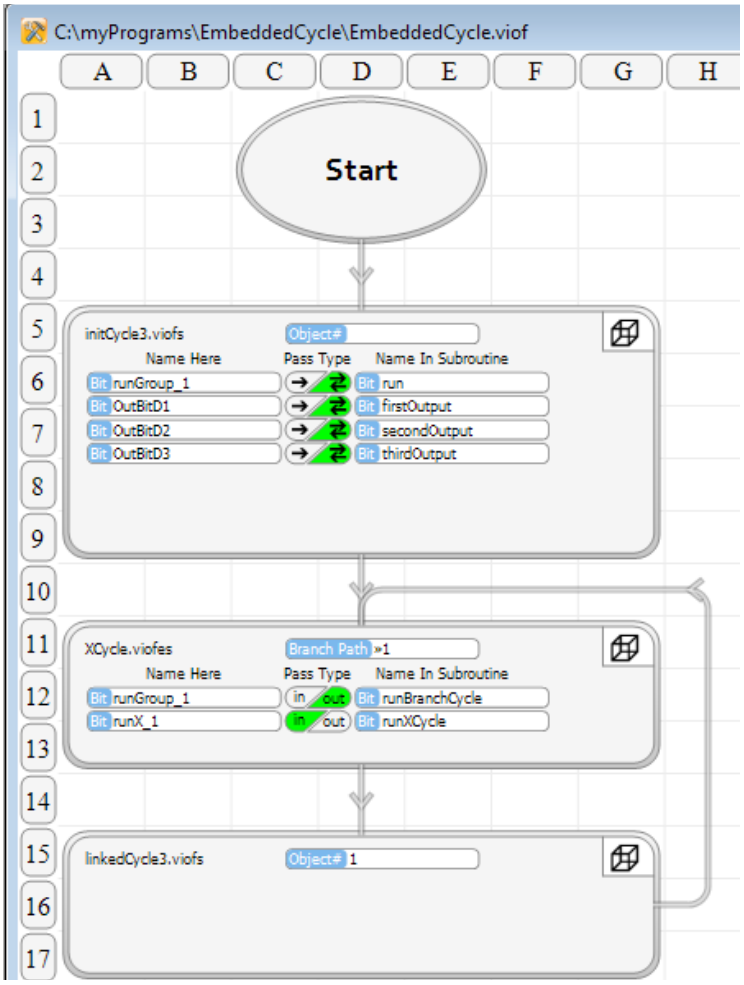
Select the EmbeddedCycle program and change the B1 input to "runX_1".



Create a Register bit tagname - "runGroup_1".



Enter the EmbeddedCycle program shown on the right. Remember that for a Subroutine call to an Embedded Subroutine, the "in" and "out" are with respect to the Embedded device The "runGroup_1" bit is passed "out" of the midCycle Embedded Object.

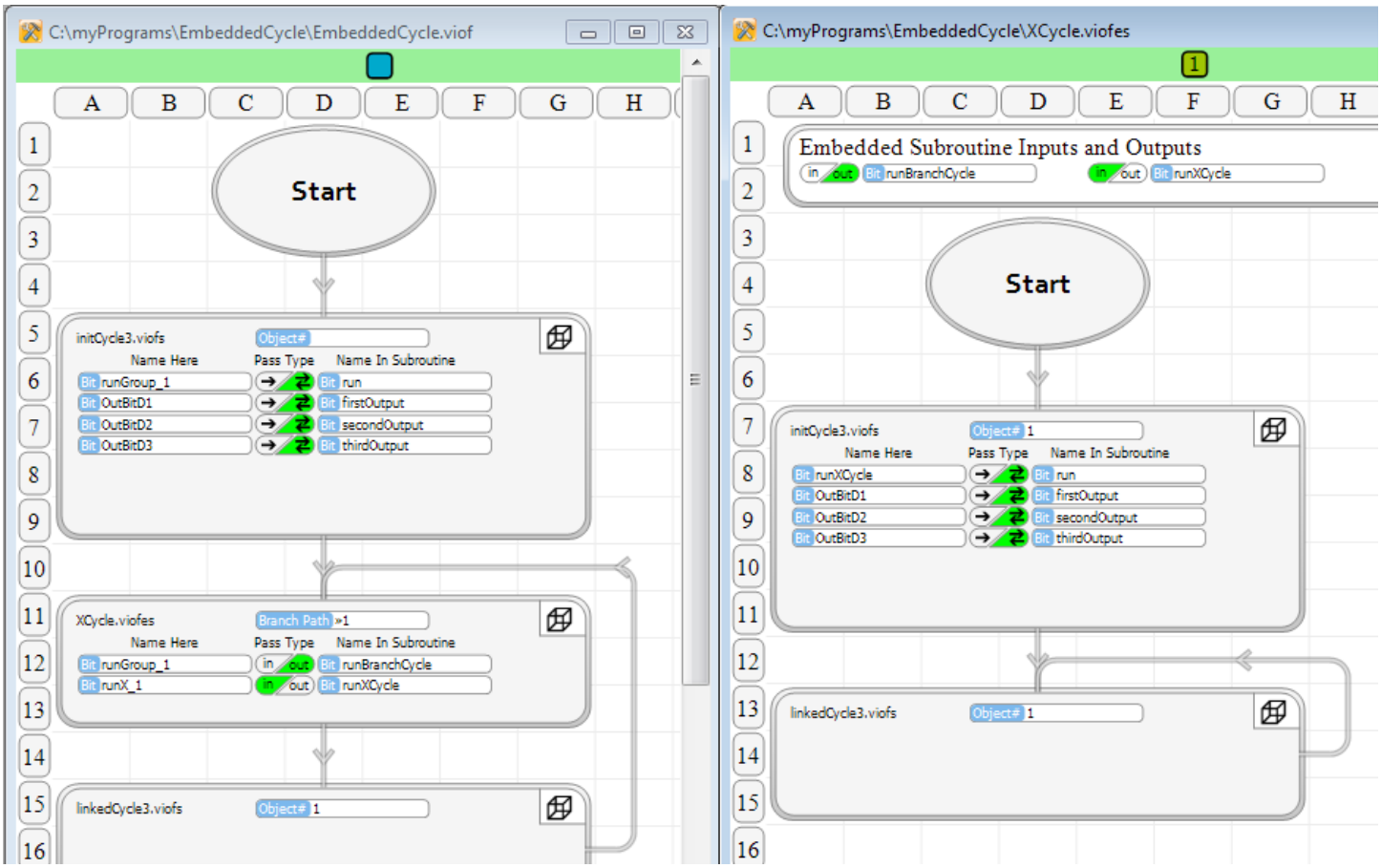


Program the PLCs and put them in the Debug mode. After you select Run :

- If you switch the B1 switch of the Branch unit, the Expansion outputs should cycle.
- If you switch the B1 switch of the Expansion unit, the Branch outputs should cycle.

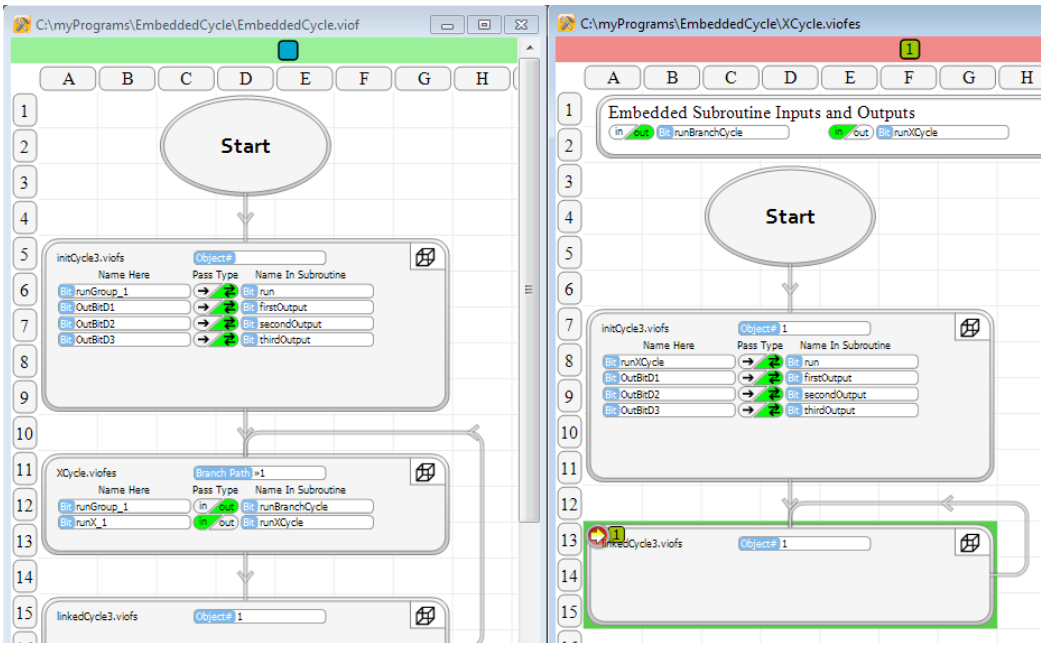
Place vBuilder in Debug Value mode.

Looking at the main programs for the Branch and Expansion units, side by side, switch the Branch unit's B1 switch back and forth. You should see the value change in both programs. The Branch B1 is the bit passed into runXCycle, shown in the EmbeddedCycle Subroutine call to XCycle, and shown as the Embedded Subroutine Inputs and Outputs (the in bit).



Try it with the B1 switch of the Expansion unit. Again, the same status should appear on both sides. You have entered and are running a program where you are communicating real time data between two independently running PLCs. You did it without setting up all kinds of communications functions. Its just a Subroutine call.

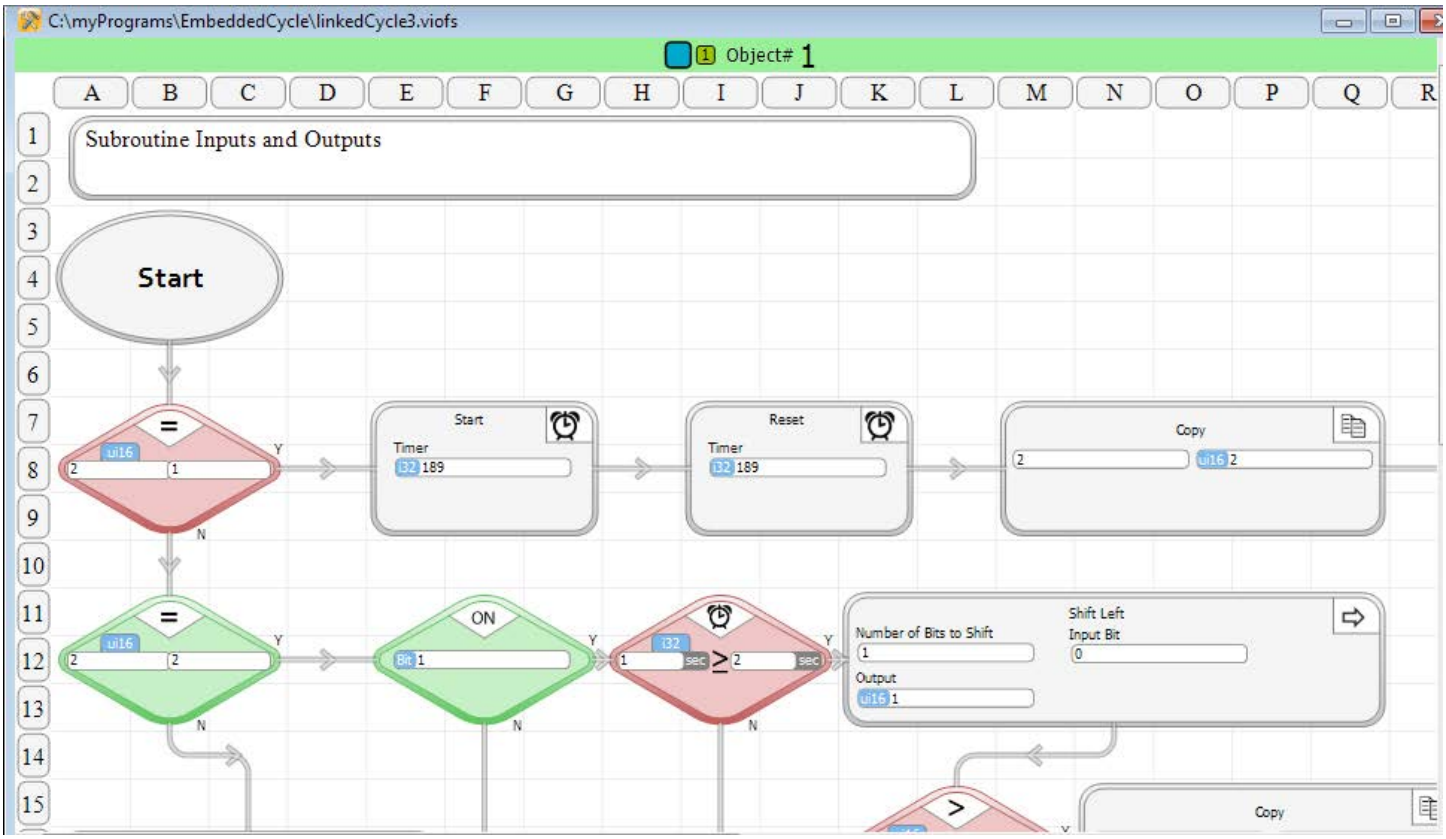
To demonstrate that the two PLCs are running independently, click the linkedCycle3 block on the XCycle program window. You will see that the XCycle program stops at that block, while the EmbeddedCycle program in the Branch unit continues to operate. In addition to the view on the vBuilder screen, you should see this happening in the actual PLC operation.



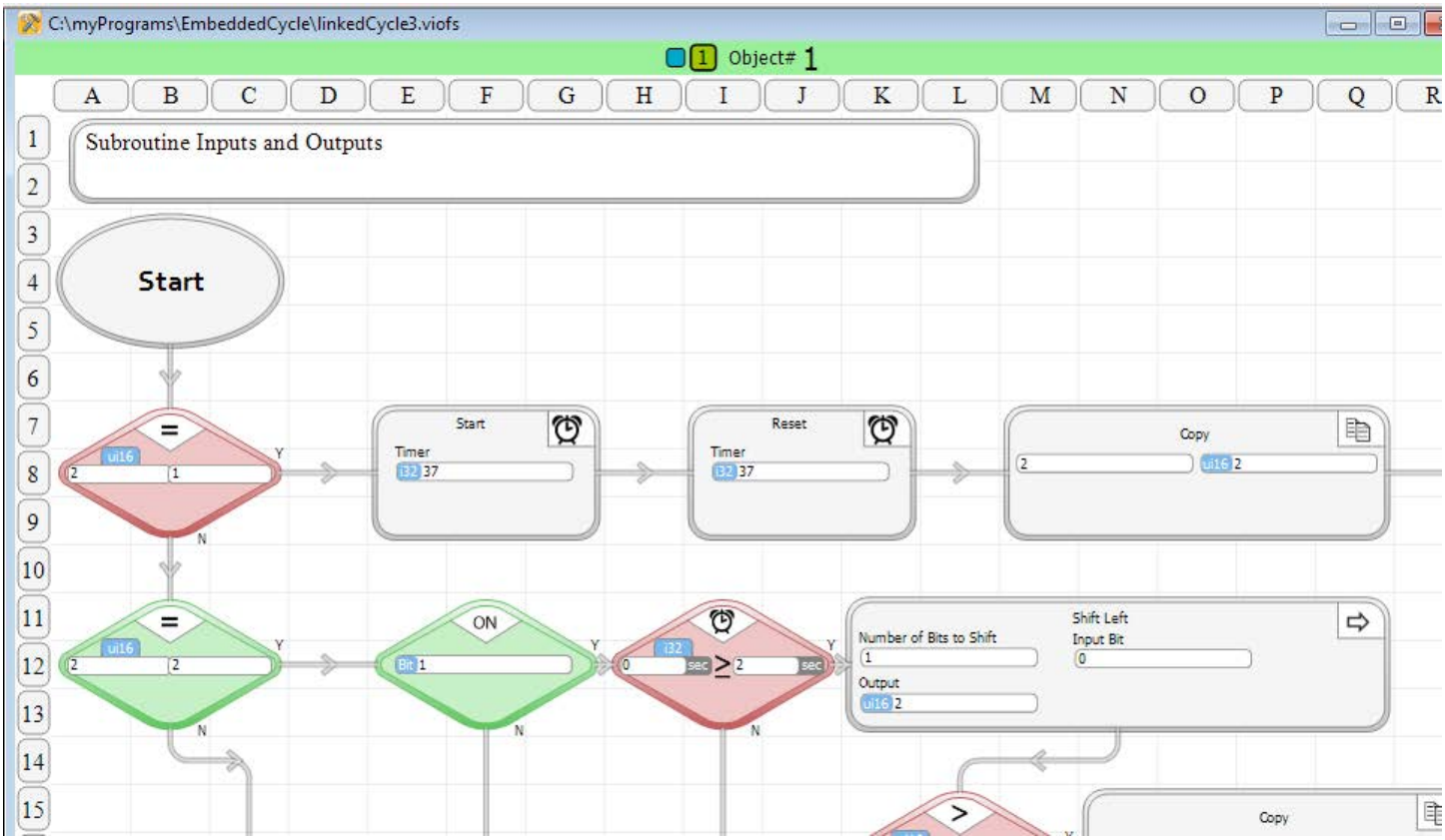
Clear the breakpoint by clicking the block again. Select run. Everything should be running again.

Try setting a breakpoint in the EmbeddedCycle program (one of the two blocks in the loop). You should see the EmbeddedCycle program stop and XCycle continue to run.

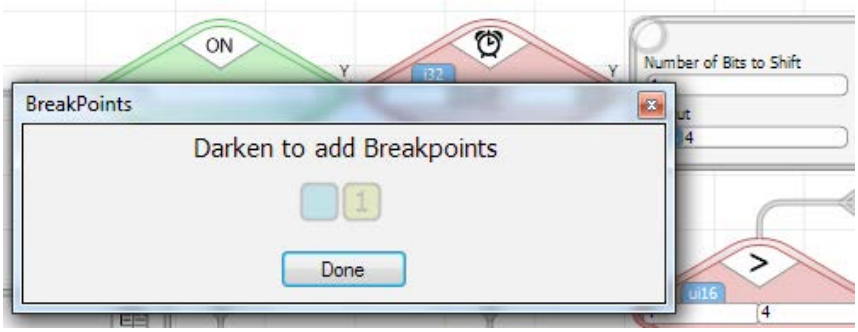
If you select the linkedCycle3 program window, you will see that at the top of the window is a blue and a green square, with one being larger that the other. It also says "object #" with a bold 1. It is displaying real time data for one of the linkedCycle3 subroutines. There is one in each PLC unit. The square that is larger indicates the unit that it is currently focused on. If you



click the other square, the focus will change. The bold 1 indicated that it is focused on object 1 of the unit.



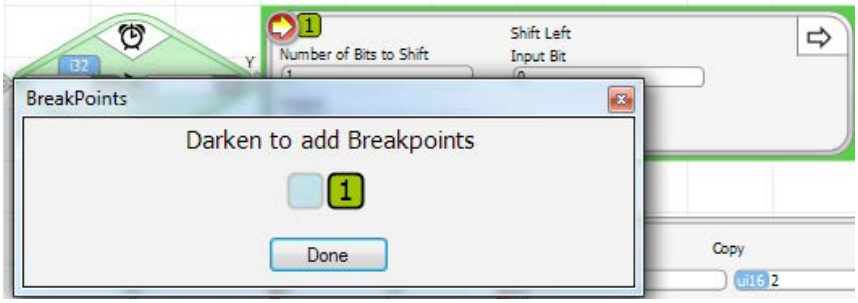
Click to set a breakpoint on a program block in the linkedCycle3 subroutine. A dialog box will pop up. Since this program is in two different PLC units, you must select which unit you want to set the breakpoint in. Click the green square with the 1 to select the Expansion, and click Done.



You should see that the program will stop in the Expansion, when it reaches the block with the breakpoint. The Branch program will continue.

This feature allows you to isolate particular PLC units in a distributed system for Debugging. As you use it in real applications, you will find it to be a very powerful feature.

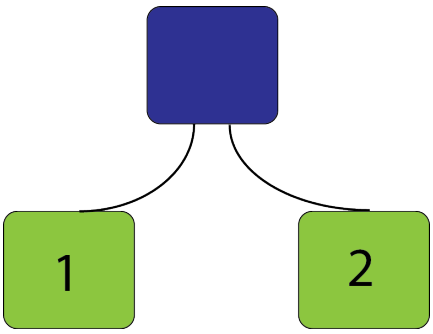
Clear the breakpoint and click run again - or single step through the program blocks. To clear it, click the block that has the breakpoint. When the dialog box pops up, click the highlighted square (which identifies the unit that has the breakpoint set), so it turns to a faded square, then click Done.



Play with the set up for as long as you like. When you are finished, we're ready to go to Phase 2.

Phase 2 : Multiple Instances of the Same Embedded Object

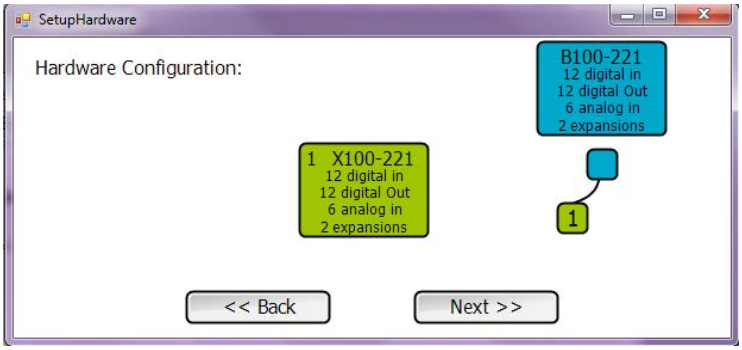
In this phase, we will add a second embedded object, with the same program and expand the Branch PLC's program to both call (communicate with) this second Embedded Object and cycle a second set of outputs, based on the second Embedded Object's input.



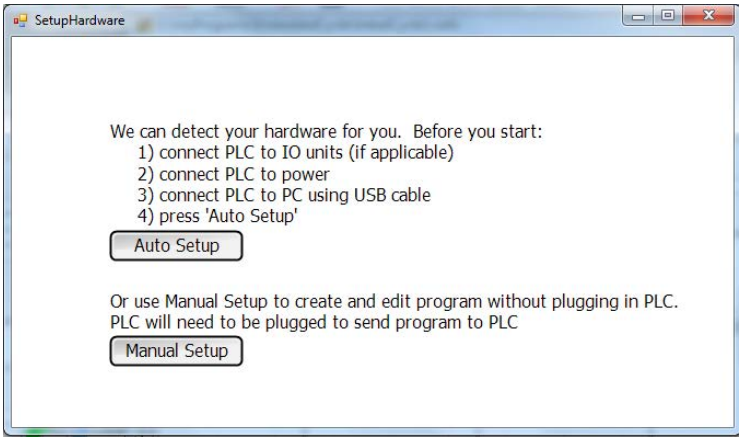
Start by turning off Debug mode and connecting a second Branch Expansion PLC to port 2 of the Branch.

We will configure the Setup Hardware by doing Auto Setup again. We always prefer to Auto Setup, to make sure that everything is present, connected and communicating before we begin programming.

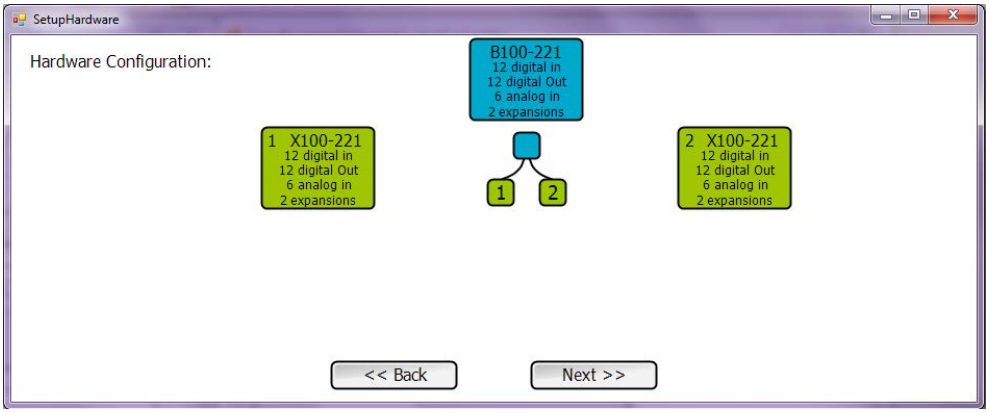
Click Setup Hardware. The current hardware setup will appear as shown on the right.



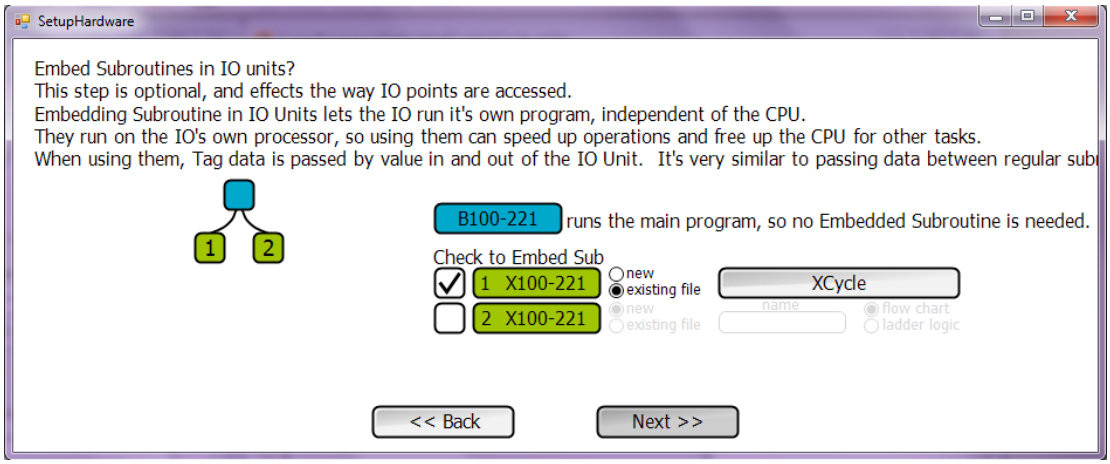
Click the Back button until the Setup Hardware window backs up to the screen that asks whether you want to do Auto or Manual Setup.



Click Auto Setup. The screen should change to appear similar to that shown on the right. It shows that you now have one Branch, with two Expansions connected to the Branch's vLink ports 1 and 2.



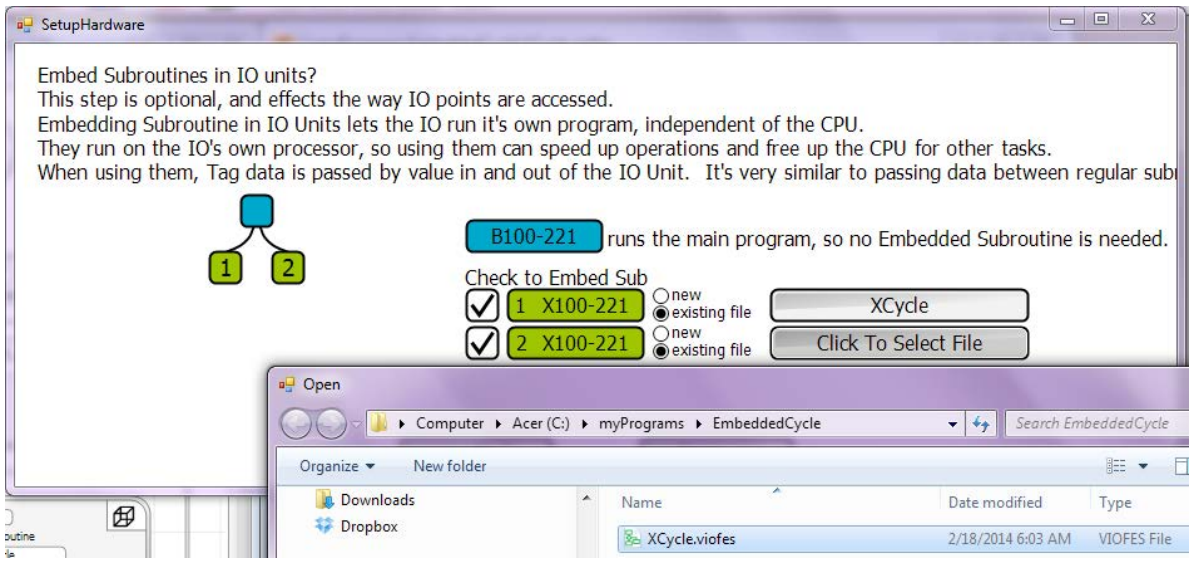
Click Next. The window will change as shown on the right. You already have configured for Embedded Subroutine XCycle to be placed in the Branch Expansion connected to vLink port 1.



Click the checkbox next to the Expansion connected to port 2, and select existing file. The rectangular button to its right will change to "Click to Select File". Click on that button and a browse window with the program directory showing, will pop up.

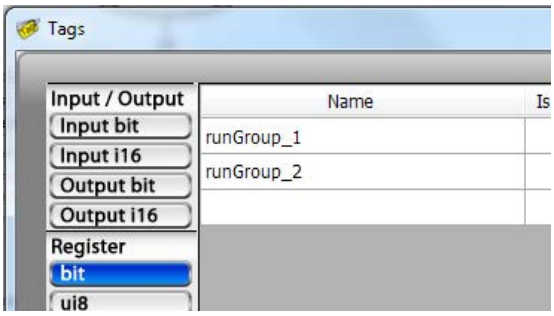
Select "XCycle.viofes" and Open.

Click Next, until you get an option for Done. Click Done.

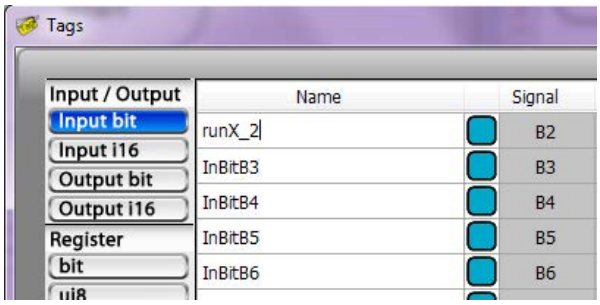


You have now configured the Hardware to place XCycle Embedded Objects in both Expansion units. We do have to make some other program changes before we're ready to go though.

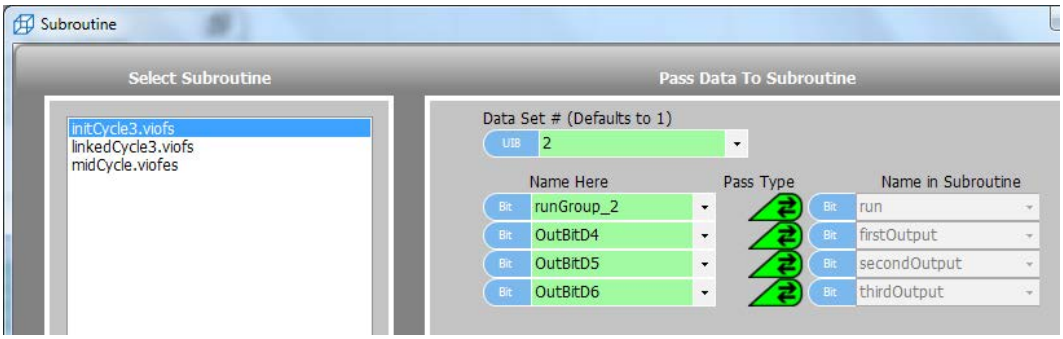
Select the EmbeddedCycle program window to place the focus on the Branch PLC's main program. Add a bit - runGroup_2.



Change the InBitB2 tagname to runX_2.



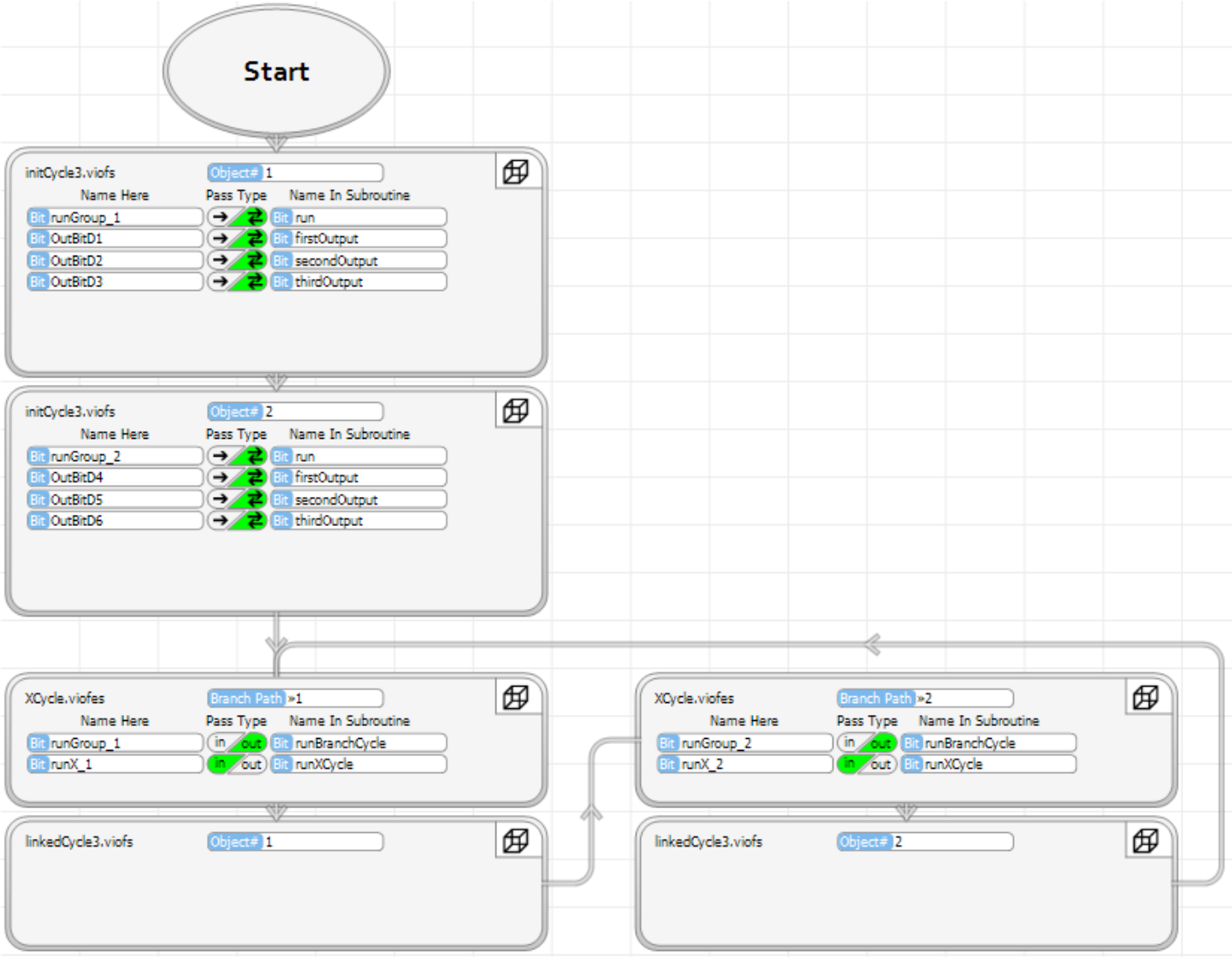
Add a second call to initCycle3, after the first Subroutine call to initCycle3. Set this call for object 2, and set up the parameters to pass like shown on the right.



Place a Subroutine Call to the second Expansion Embedded Object after the linkedCycle3 call. Use the parameters shown.

Add another linkedCycle3 Subroutine call next. This time set it to call for object 2.

Wire the program flow up as-shown on the next page.



Program the PLCs. Select Debug and Run.

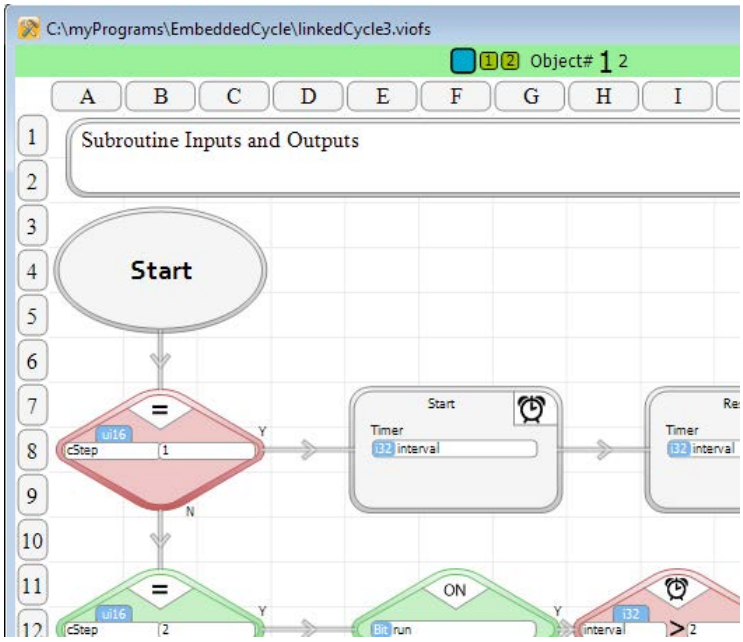
If everything was entered as shown, the program will run.

- If you turn the Branch unit's B1 switch on, the Expansion 1 outputs will cycle.
- If you turn the Branch unit's B2 switch on , the Expansion 2 outputs will cycle.
- If you turn the Expansion 1 unit's B1 switch on, Branch outputs D1 though D3 will cycle.
- If you turn the Expansion 2 unit's B1 switch on, Branch outputs D4 through D6 will cycle.

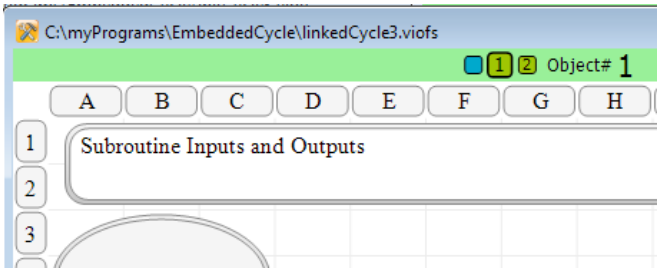
You have created an application in which three PLC units are each executing their program logic simultaneously and passing information seamlessly. Granted, this one is a simple example. It does demonstrate the basic concepts that can be scaled to any level.

Let's look at it closer.

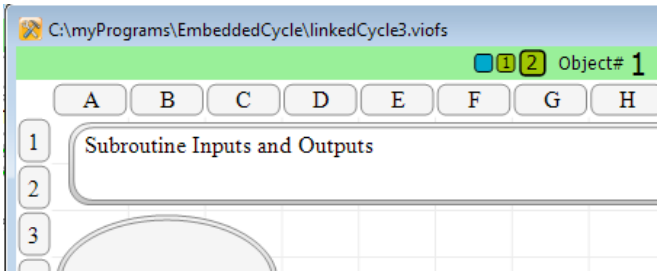
Select the linkedCycle3 program window. Take a look at the top status bar with the green background. Notice that if the Branch is selected (click the blue square), two objects are listed. You can select between the two objects, by clicking either the '1' or the '2'. Whichever one is selected, the status and value information shown in the program window applies to it.



Notice that if you select either Expansion 1,



or Expansion 2, the display only shows one Object. There is only one linkedCycle3 object in each of the Expansion units, whereas there are two in the Branch. vBuilder keeps track of this.



You can select a particular program in a particular device and set breakpoints and perform any type of debugging that you want.

Notice that if you disconnect the vLink cable between the Branch and one of the Expansions, both sides will continue to operate based on the switch status that existed prior to the disconnect. Disconnecting communications means that the data does not get updated. Sometimes this is OK. Sometimes, you want to know if communications is lost. You may need to have a program that performs operation differently when communications is active versus when it is inactive. That is the subject of Phase 3.

Phase 3 : Adding a Heartbeat

In many distributed, multiprocessing applications, you will want to know if device communications is active. It is quite common to design control systems in which each device can operate autonomously, as well as in an integrated system. The operation in stand alone mode, or when the communications link is lost (generally the device connected to it is turned off) operation should not depend on receiving information or commands from the missing link. The program needs to know if communications is active or not.

A very simple solution to this problem is the addition of a "heartbeat" to the communications parameters. The concept is simple.

- The device on one side of the link toggles a tagnamed bit at a predefined frequency (the heartbeat)
- The heartbeat bit is included in the parameters of the Embedded Subroutine Call
- The device on the other side of the link checks that the hearbeat is changing state within a reasonable time period. If it is not, it discerns that the link is not active.

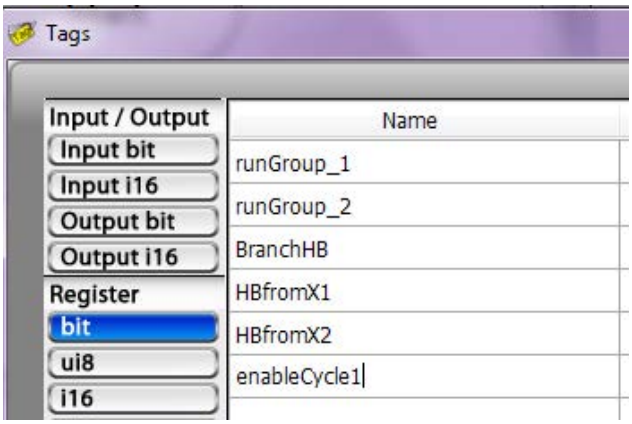
Heartbeats can be sent both directions in a link, if desired.

We'll add heartbeats to our example.

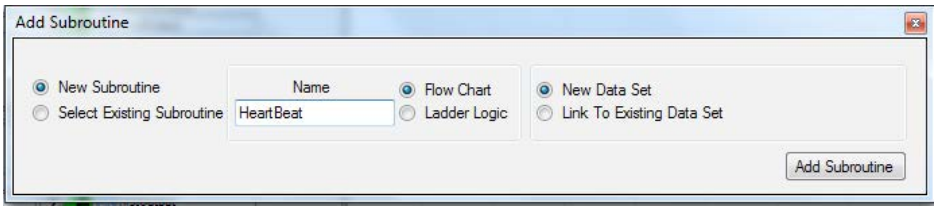
- In the example, heartbeats will be generated in the Branch and both Branch Expansions.
- These heartbeats will be sent to and from the Branch.
- In the Expansion units, we'll change the program so that if a heartbeat is not active, cycling will be disabled.
- In the Branch unit, we'll change the program so that if the heartbeat from Expansion 1 is not active, the cycling of the first set of outputs will be disabled.
- Cycling of the Branch unit's second set of outputs will not be dependent on the heartbeat.

Begin by turning Debug off.

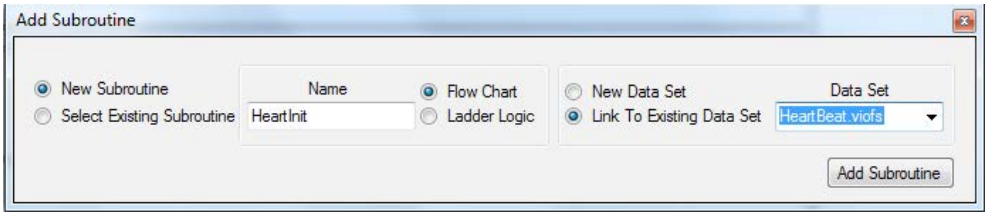
Select the EmbeddedCycle program window and add heartbeat bit tag names for each of the Expansion units and the Branch, as well as a bit tagname labeled 'enableCycle1'.



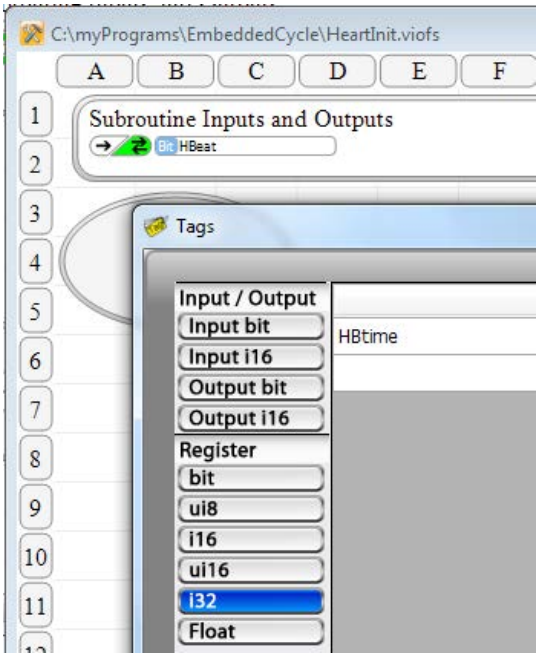
Select Add Subroutine under the EmbeddedCycle list and add a Subroutine called HeartBeat.

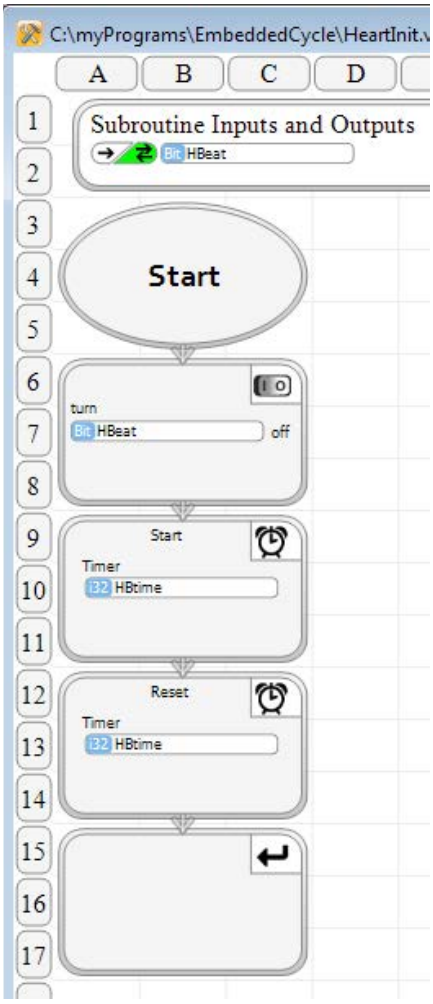


Add a second subroutine, linked to HeartBeat, called HeartInit.



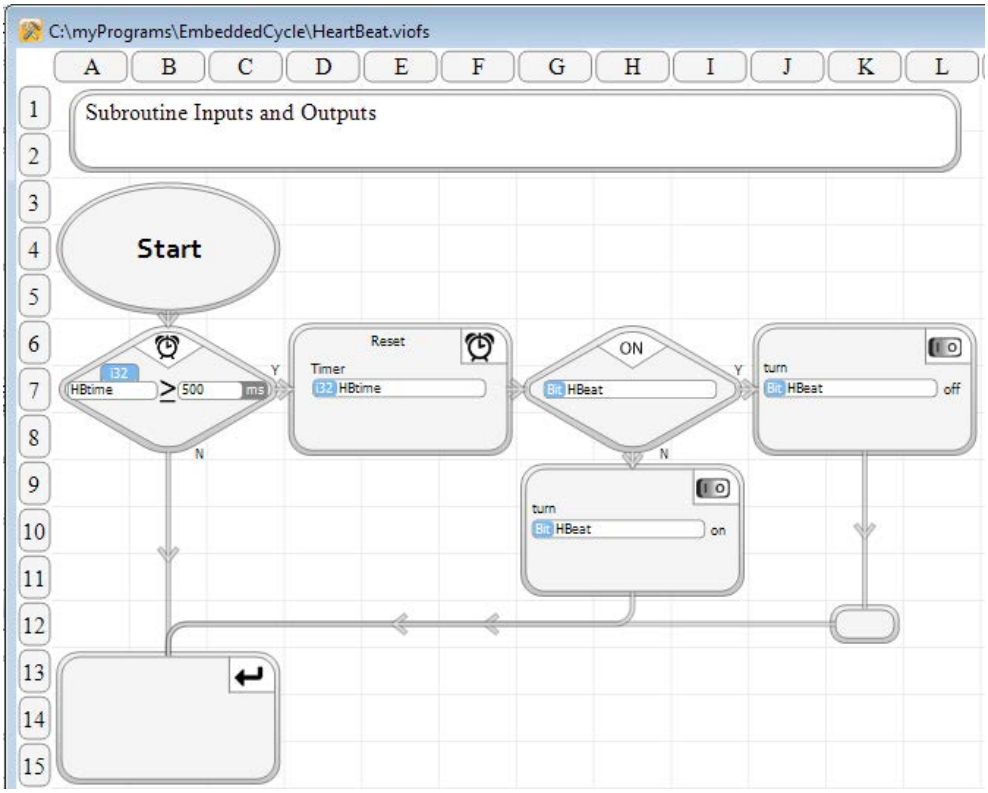
In HeartInit, define HBeat as a pass by reference input Bit. Then select Tags and define an i32 tagname, HBtime.



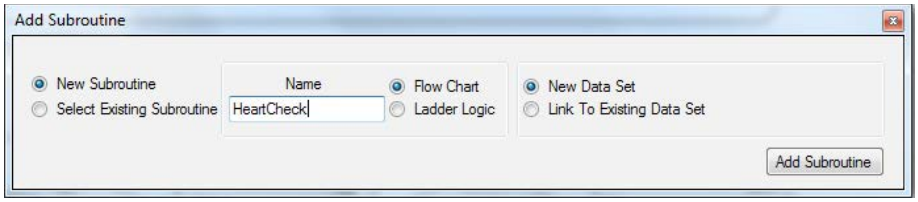


Then enter the HeartInit program as shown on the left. Initialize to HBeat off. Start and Reset the HBtime.

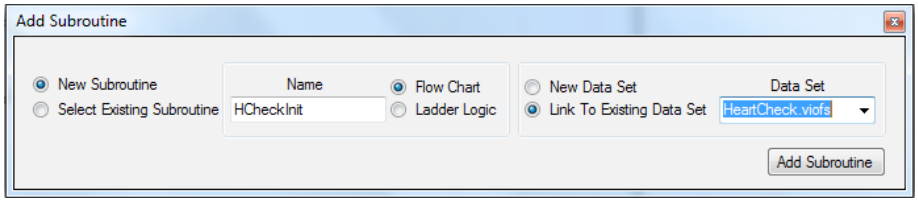
Enter the HeartBeat program as shown below. Each time the HBtime reaches 500 milliseconds, reset the timer and reverse the HBeat bit.



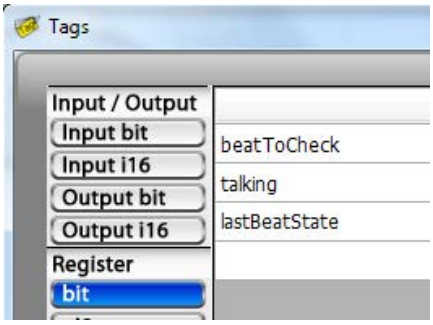
Now create another Subroutine under EmbeddedCycle, called HeartCheck

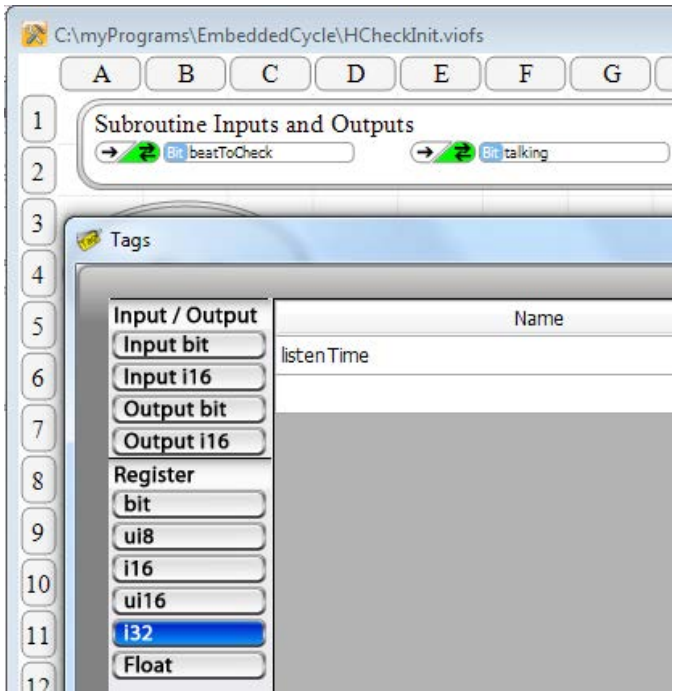


And create HCheckInit as a new Subroutine linked to HeartCheck.

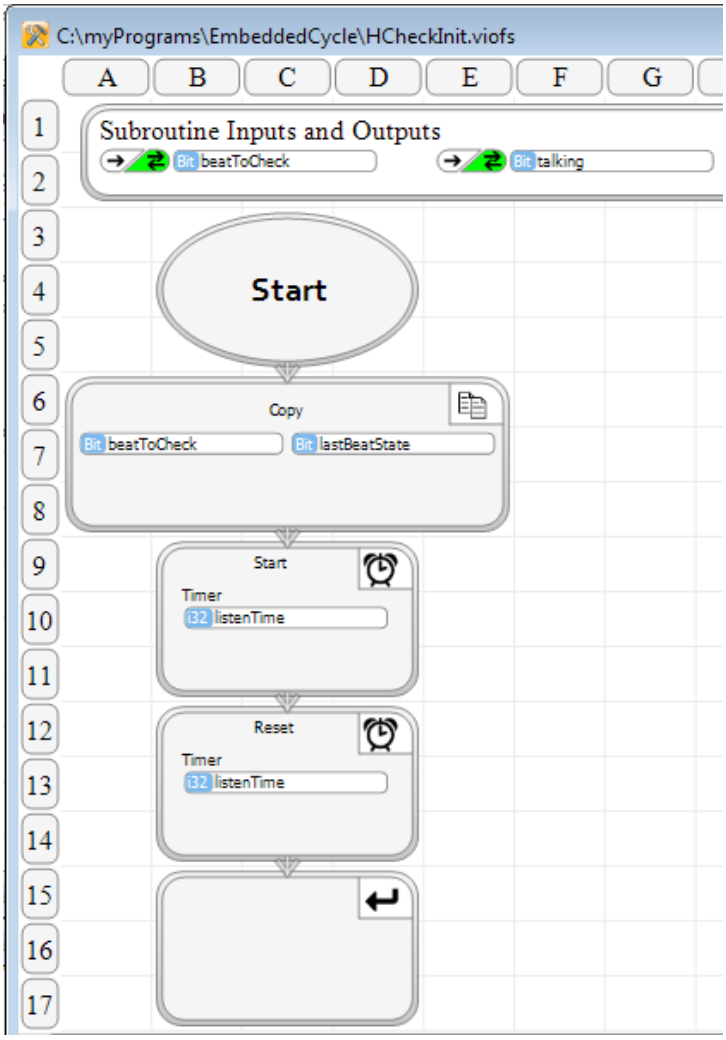


In HCheckInit, define 'beatToCheck' and 'talking' as pass by reference bit Inputs. Create an i32 tagnamed variable called 'listenTime" and a bit called 'lastBeatState'.



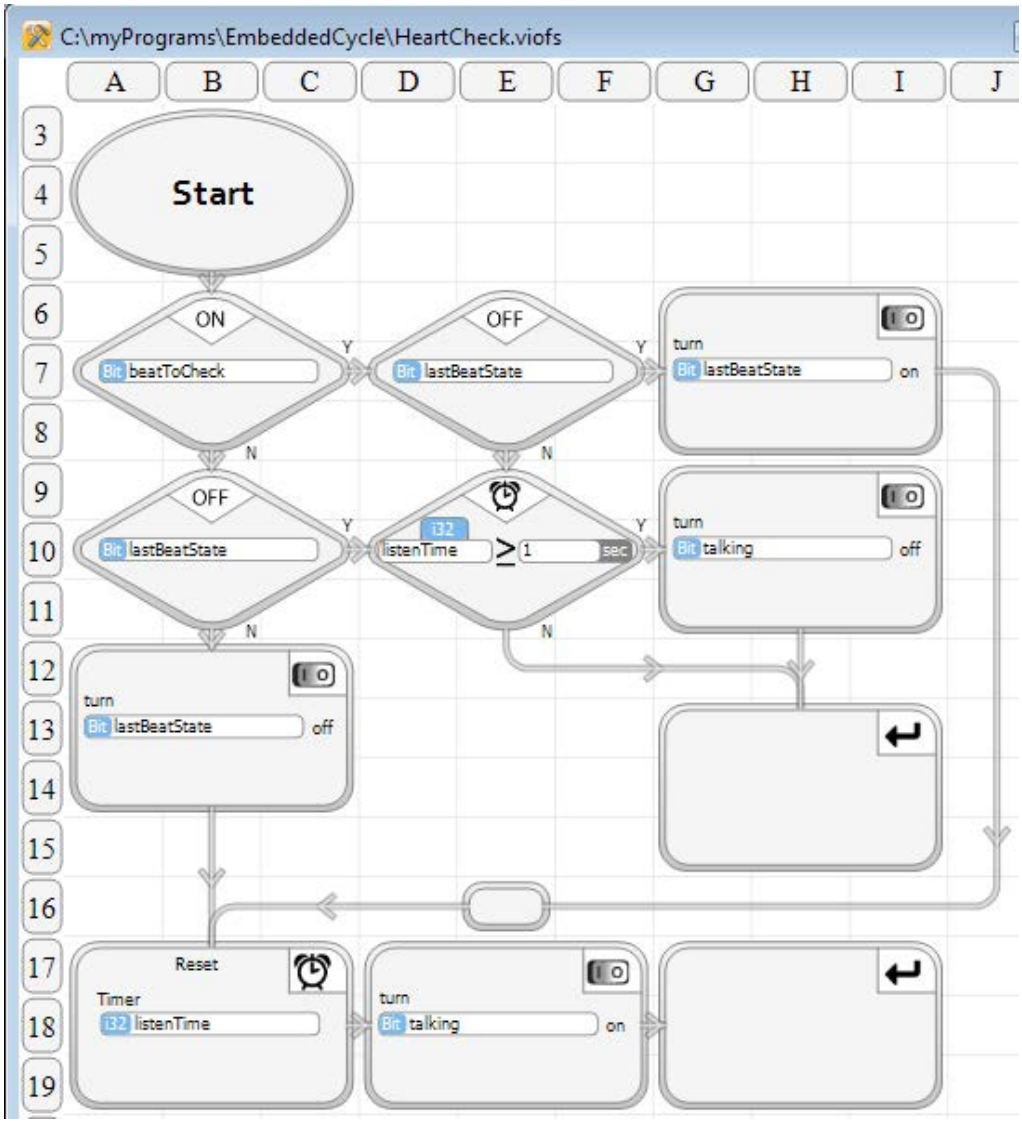


Enter HCheckInit, as shown below.



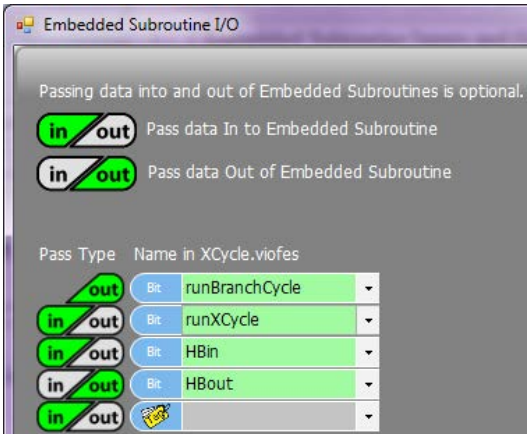
Now, enter HeartCheck, as shown.

Looking at the HeartCheck logic, you should see that as long as the heartbeat from the remote device "beats", or changes state at least once every second, HeartCheck will maintain 'talking' as true. If the heart beat isn't sense as changing within a second - which is what will happen if you disconnect the vLink cable or power off the connected PLC device - 'talking' will be set to OFF.

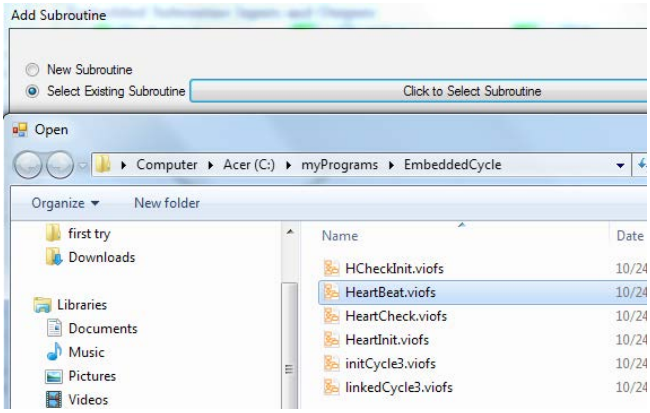


Select XCycle.

Add two bits to the Inputs and Outputs. HBin is the heartbeat from its master. HBout is the Embedded Object's heartbeat to its master.

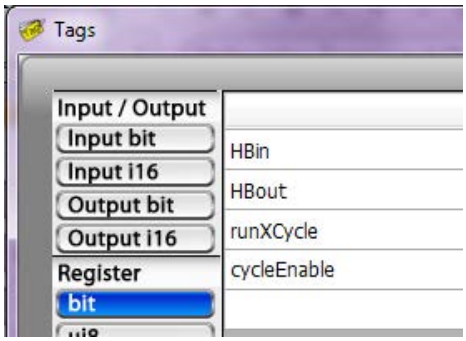


Under Program Files (far left), under XCycle, select Add Subroutine. Click "Select Existing Subroutine", then select HeatBeat. We are putting the same HeartBeat program in the Expansions as in the Branch.



Do it again for HeartCheck.

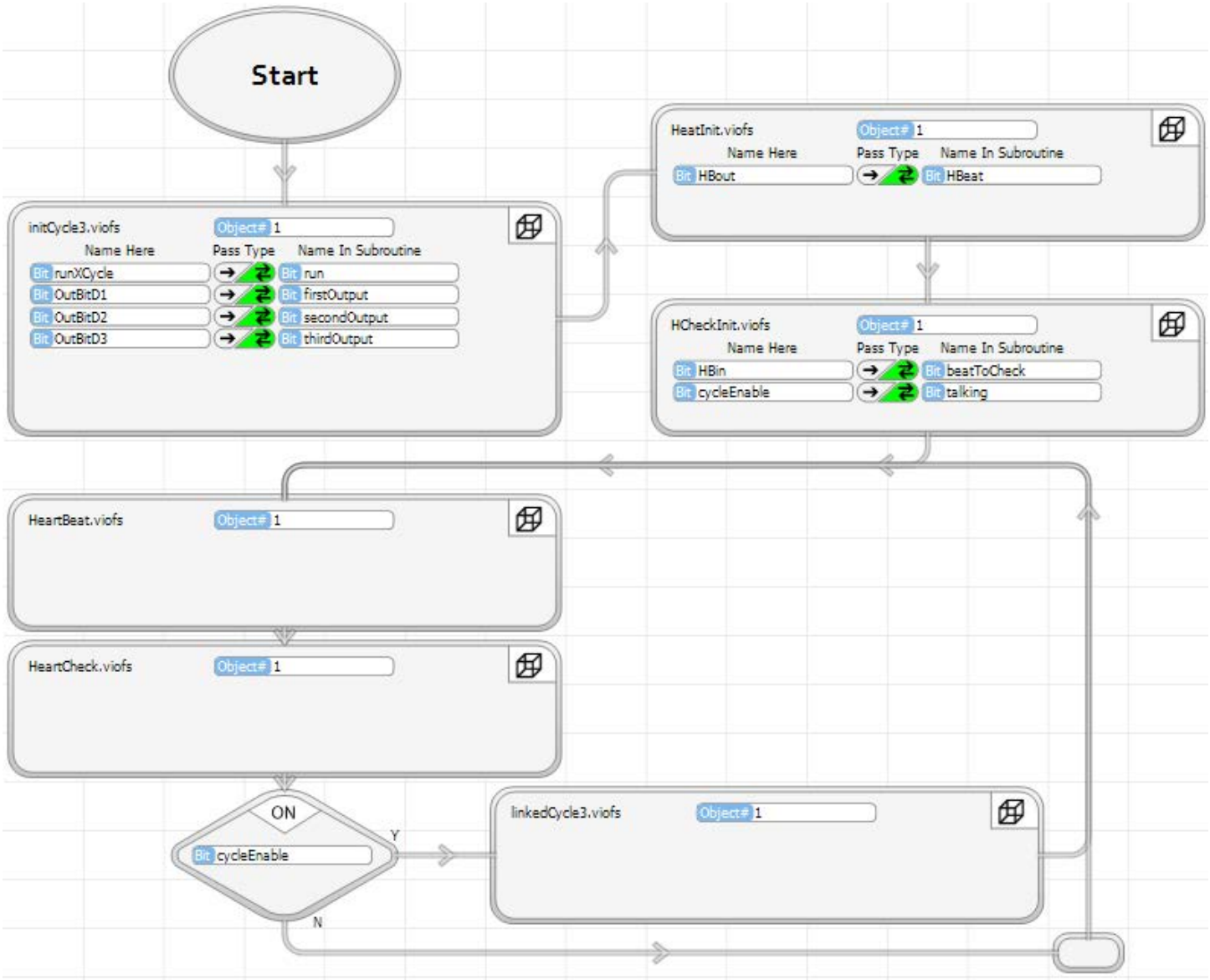
Next, create a new bit, labeled cycleEnable, in XCycle.



Modify the XCycle program as shown below.

- Add HeartInit and HCheckInit to the initialization section, before the infinite loop.
- Add HeartBeat, to generate this unit's heartbeat
- Add HeartCheck to check the heartbeat from the Branch unit
- Add a decision block to check if we are getting a hearbeat from the Branch to determine whether linkedCycle3 should be called.

The expected operations should be fairly obvious. The heartbeat communications is handled transparently in the Subroutine Call in the Branch unit's program.



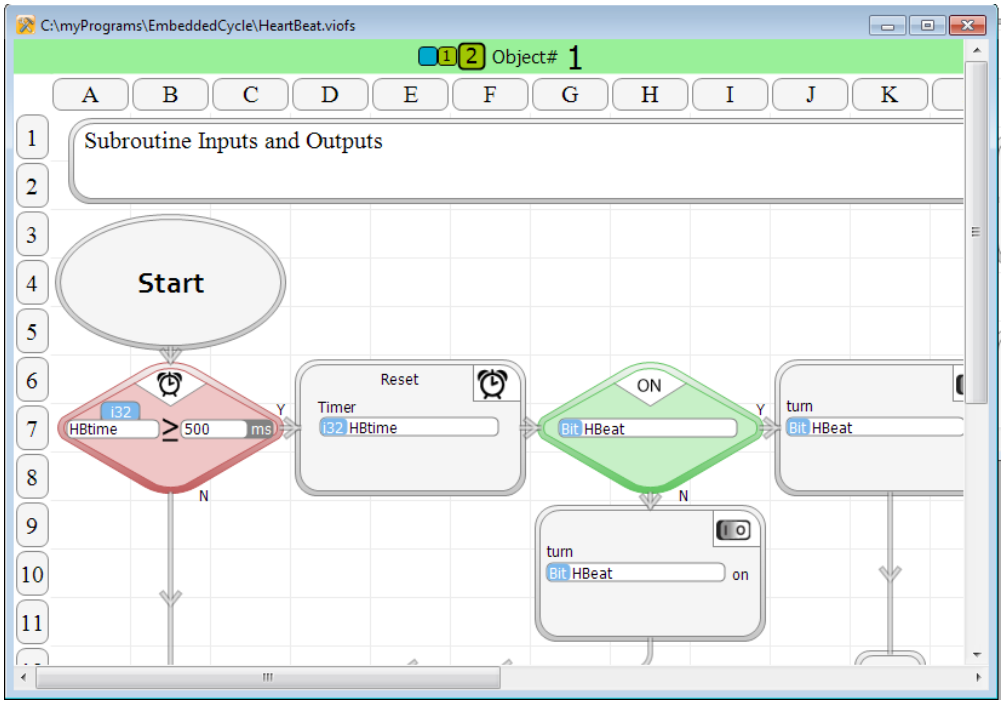
Modify the EmbeddedCycle program as shown below.

- Add HeartInit and HCheckInit to initialize the Branch heartbeat and initialize the check for Expansion #1s heartbeat
- Add HeartBeat to the main program loop to generate the Branch heartbeat
- Add the heartbeat in and out bits to both midCycle subroutine calls
- Add a check for whether we have a heatbeat for XCycle #1 to determine whether to call linkedCycle for the first output group.

Program the PLCs, put them in Debug mode and select Run.

You should see the three PLCs operate like they did before. The two groups of Branch outputs cycle, based on switches from the two Expansions. The Expansions outputs cycle based on the B1 and B2 Branch inputs.

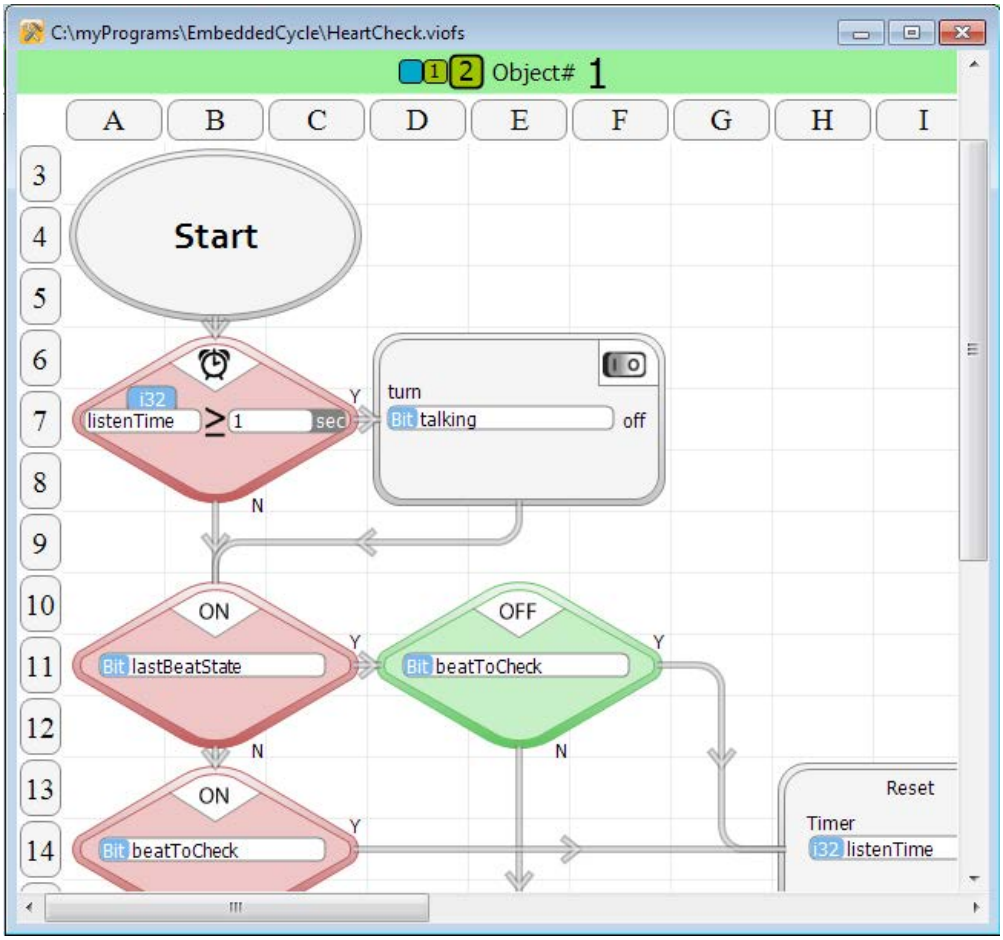
Select the HeartBeat program window. Select each of the PLCs (blue/green 1/green2). What you should see for each of them is the heartbeat generated by this program. Each HeartBeat program is running in a different PLC. If you watch the decision block at H7, it will appear to beat like a heart, turning red/green/red/green on a one second cycle.



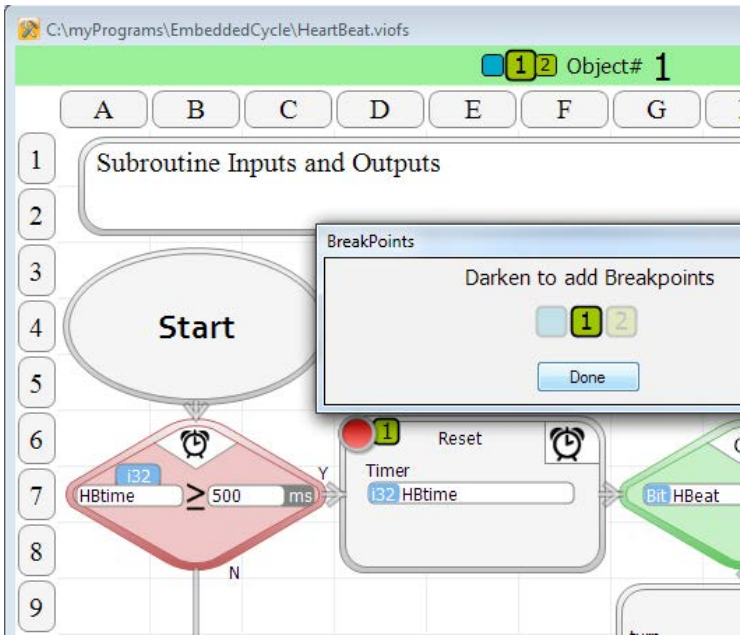
Select the HeartCheck program window. Again, select the three different PLCs. In each case, you should see the heartbeat - sent from another PLS over vLink- as it is received at the selected device. It should look just like the generated heartbeat.

A few things to notice.

- All you had to do to implement communications, in both directions between the Branch and each of the Expansions, was to connect the cables and program a subroutine call.
- The communications occurs transparently. The data is just transferred very similar to a subroutine call in the same PLC.
- Communications is fast. The heartbeat looks the same on both sides.



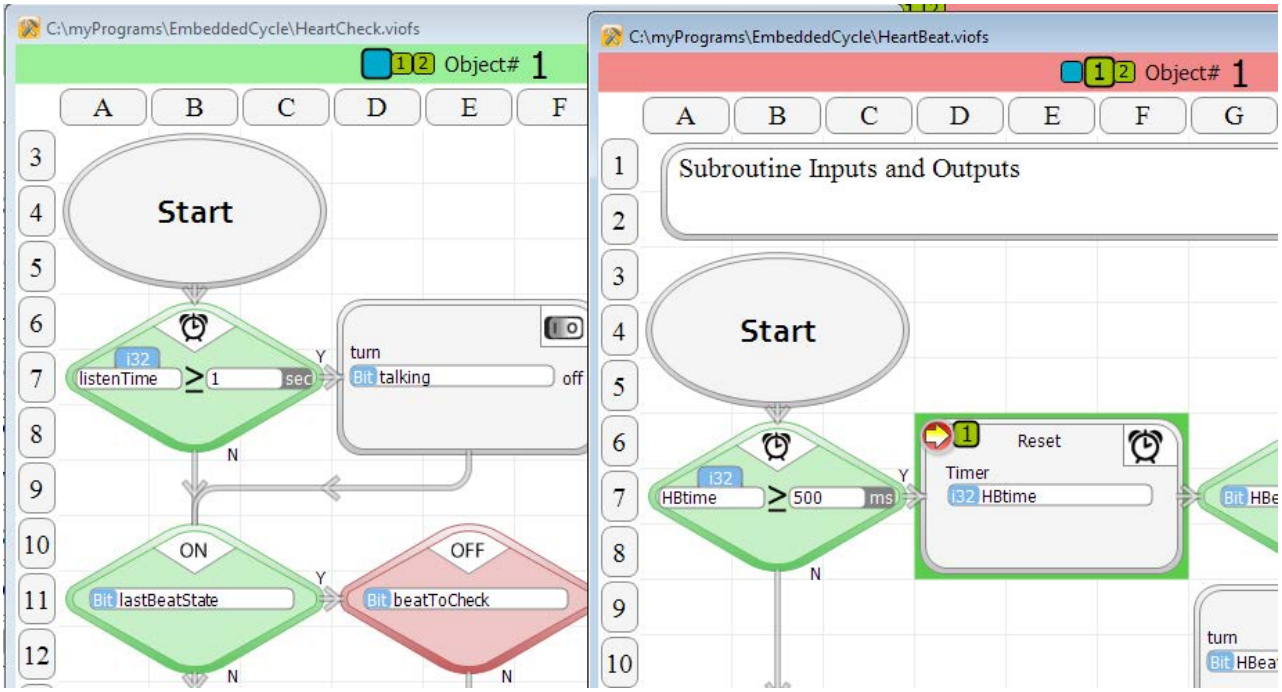
Select HeartBeat again. Place a breakpoint in Expansion 1. To do so, select a program block, then select the green 1 square and click done. When the progam reaches this block, it will stop & therefore it will quit sending a heartbeat.



Look at HeartBeat and HeartCheck side by side. (Remember that only the active program window will show real time data, so you'll have to click back & forth between the two windows) If you select the Branch (blue square) for HeartCheck, you should see that the heartbeat received from Expansion 1 has stopped. If you select Expansion 2, you will see that everything is normal.

If you select Expansion 1, you will see that the received heartbeat from the Branch is still being received, but the program isn't doing anything with it (lastBeatState never changes). That's because the Expansion 1 program is stopped in HeartBeat. The status bar at the top of all the program windows, when the focus is selected for Expansion 1, will be red. This indicates that the program execution is stopped in this unit.

The other thing to notice when Expansion 1 is stopped at a breakpoint, is that the Branch's 1st group of outputs quit cycling. That is due to the logic that we just added. So the heartbeat is doing exactly what we intend for it to do.



You can continue to play with the program. To start Expansion 1 operating again, clear the breakpoint and click Run. Disconnecting cables should have the same affect as stopping a unit's program.

Bottom Line : Developing a distributed processing system is easy with Velocio PLCs & vBuilder. Adding a heartbeat for each unit is recommended in most cases. What you do with the heartbeat is going to be application dependent.

9. Modbus Communications

Modbus RTU communications is the most commonly used, simple communications protocol used in the automation industry. It was developed by the first PLC manufacturer, Modicon, back in the late 1970s. Originally, Modbus was designed to communicate over 1960s physical communications standard - RS232. Over the years, the Modbus RTU communications protocol has been propogated to work over RS485, RS422, Ethernet, and now USB. It may be implemented over other communications links as well.

Protocol definition for Modbus RTU communications is readily available. If you don't already have that documentation, simply Google Modbus protocol. A large number of links to documents will come up. If you are unfamiliar with Modbus, we'd recommend that you go through this documentation first. Modbus RTU is a very simple protocol. To simplify it further, you only need to review the messages that are actually used. Those messages are listed below.

Function code	01 (0x01) :	Read Coils
	02 (0x02) :	Read Discrete Inputs
	03 (0x03) :	Read Holding registers
	04 (0x04) :	Read Input registers
	05 (0x05) :	Write Single Coil
	06 (0x06) :	Write Single Register
	15 (0x0F) :	Write Multiple Coils
	16 (0x10) :	Write Multiple registers
	23 (0x17) :	Read/Write Multiple registers

As a matter of fact, you don't even have to implement all of the function codes. Function codes 01 and 02 read the same information. Function codes 03 and 04 do also. You only need to implement either function code 01 or 02 and either function code 03 or 04.

Velocio Networks supports Modbus RTU communications over USB. The USB standard supports a number of "profiles". A profile is an implementation of communications that works over USB in a particular manner to optimize a particular type of data transfer. Some profiles are standard. Some are proprietary. There are standard profiles for communications with a USB mouse, for doing high speed data transfers to devices like a USB thumb drive. There is also a standard profile, called the CDC profile, that makes the USB port operate like an RS232 serial port. In fact, if you've got a USB to RS232 conversion cable, the USB profile used is the CDC profile.

In the USB world, one side of the communications link is the "host". The other side is the "device". A PC almost always operates as a "host". Velocio PLCs operate as a "device".

If you are implementing Modbus RTU communications to a Velocio PLC, your apparatus must be a "host"

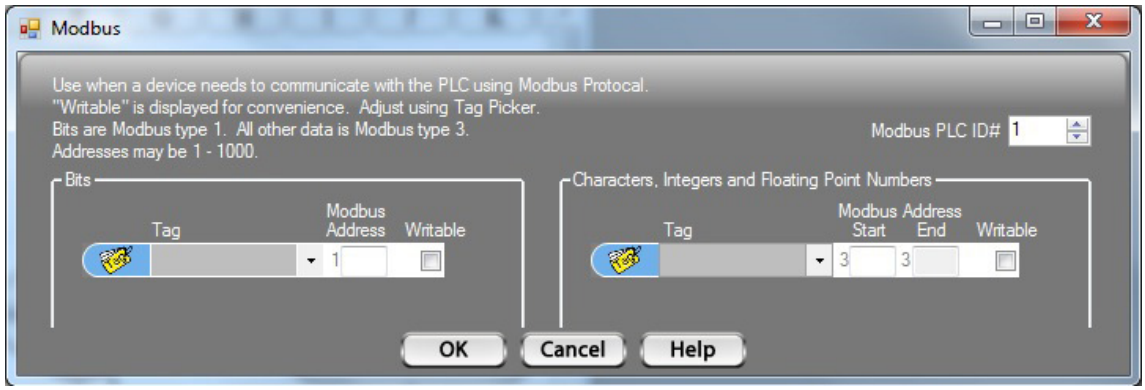
If you are implementing communications from a PC or similar type of machine, it will naturally be configured as a host. If you are communicating from an embedded instrument, like an HMI panel, your product design must incorporate a host USB port.

Configuring a Velocio PLC for Modbus Communications

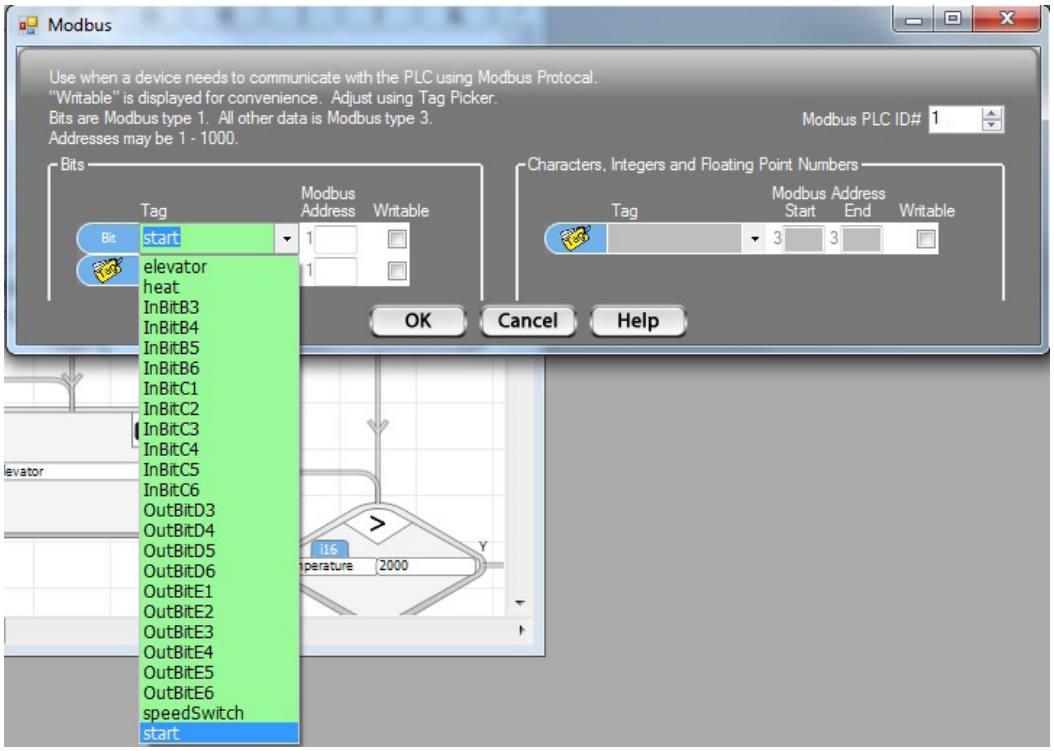
In vBuilder, any global tagname can be selected for Modbus communications. Additionally, each selected tagname can be set for either read only or to allow to be written to by an external device over a communications link (Modbus). The following paragraphs will descibe how to perform Modbus configuration for a vBuilder program. You can write your own programs, configure them as you'd like and test your communications. In addition, Velocio has a vBuilder program already written and configured to communicate via USB Modbus and a demonstration application that runs on any PC and clearly illustrates Modbus communications to a Velocio PLC. Also available from Velocio is an Application Note that goes through the demonstration.

On the vBuilder top tool bar, there is an icon that likes like this :

If you select this icon, you can configure Modbus communications. When you select the icon, a dialog box will appear, as shown on the right.



First, notice the setting for Modbus PLC ID # in the upper right corner. This value will be required in the first byte of all Modbus messages. It defaults to a value of 1. It can be adjusted between 1 and 255, via the up and down arrows. Whatever you set this value to, all Modbus messages must start with this number.



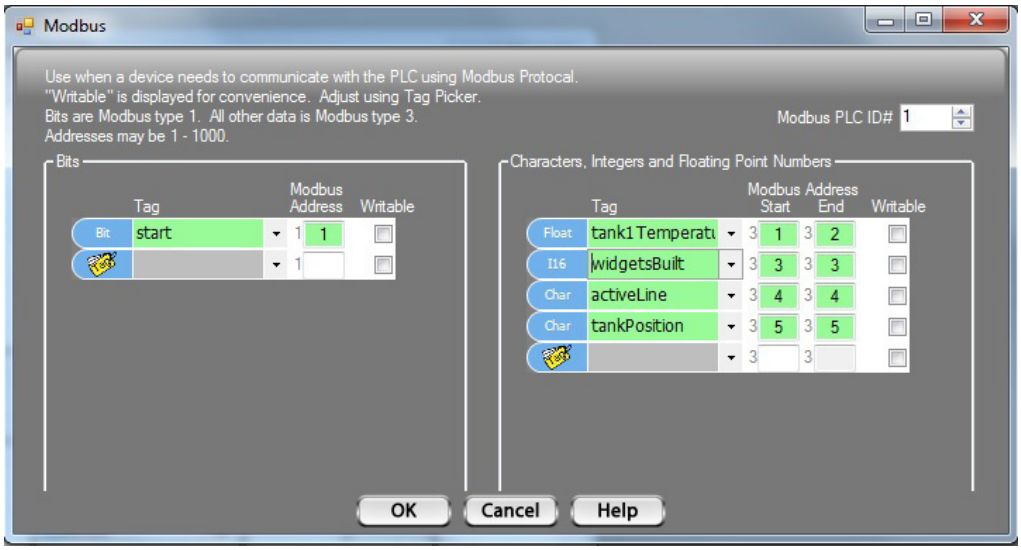
Any tagname available to the main program (not subroutine object data) can be selected for Modbus communications. Global tagnames include all inputs and outputs, as well as any tagnames defined in the main program.

In the dialog box, the list on the left is bits configured for Modbus communications. The list on the right is a list of tagnames of all other data types configured for Modbus communications.

Bits are mapped to Modbus address 1xxxx, where xxxx is between 1 and 65535. Register data is used for all non-bit data (unsigned 8 bit integer, unsigned 16 bit integer, signed 16 bit integer, signed 32 bit integer and floating point). Modbus addresses used for register case is in the 3xxxx range, where xxxx is between 1 and 65535.

To select a tagname for Modbus mapping, select it from the drop down box, or type in the tag name. Next, select a Modbus address to associate with the tagname. If you simply click inside the Modbus address box, next to the tag name, vBuilder will automatically assign the next available address. You can change it if you want, by typing in the address that you would like.

The illustration on the right shows Modbus mapping for one bit (start), one floating point number (tank1Temperature), one signed 16 bit number (widgetsBuilt) and two unsigned 8 bit numbers (activeLine and tankPosition).



Modbus was originally defined when the only data types available were bits and unsigned 16 bit numbers, so all bit reads and writes handled individual bits and groups of bits. All "register" reads and writes dealt with individual 16 bit numbers or groups of 16 bit numbers. To incorporate the varied data types available in Velocio PLCs, we map all non-bit numbers into 16 bit "registers". This is a listing of how various data types are mapped.

unsigned 8 bit	individual 16 bit registers with the high 8 bits unused
unsigned 16 bit	individual 16 bit registers
signed 16 bit	individual 16 bit registers
signed 32 bit	two 16 bit registers, with the most significant bits in the first (or lowest address) register
floating point	two 16 bit registers, with the most significant bits in the first (or lowest address) register

If you look at the dialog box above on the right, you will see that the floating point number is assigned Modbus addresses 1 and 2. The unsigned 16 bit number is in address 3. The two ui8 numbers are in individual registers 4 and 5.

Appendix A : Installing vBuilder

Appendix B : Program Configuration and Limits

(version 1.0 software)
(all limits are per PLC module)

Application Progam Memory : 34K program words
Maximum Program function blocks : 4K
Maximum # of Subroutines : 68
Maximum tagnames : 950
Main program tagname memory :

- Bits : 2048
- ui8s : 512
- ui16s : 512
- s16s : 512
- s32s : 256
- floats : 256

Object Memory : 4096 words (16 bits/word)

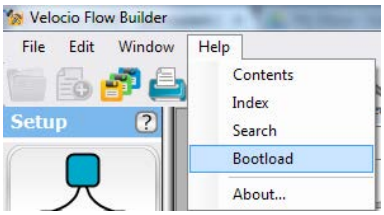
- object bits allocated in groups of 16, as needed; 16 bits = 1 word
- object u8s allocated in groups of 2, as needed; 2 ui8s = 1 word
- object ui16s : 1/word
- object si16s : 1/word
- object si32s : 2 words/ si32
- object floats : 2 words / float
- references : 4 words/ reference

Maximum # objects : 292
Maximum number of tagnamed variables to or from an Embedded Object device : 64

Appendix C : Bootloading PLC Firmware Updates

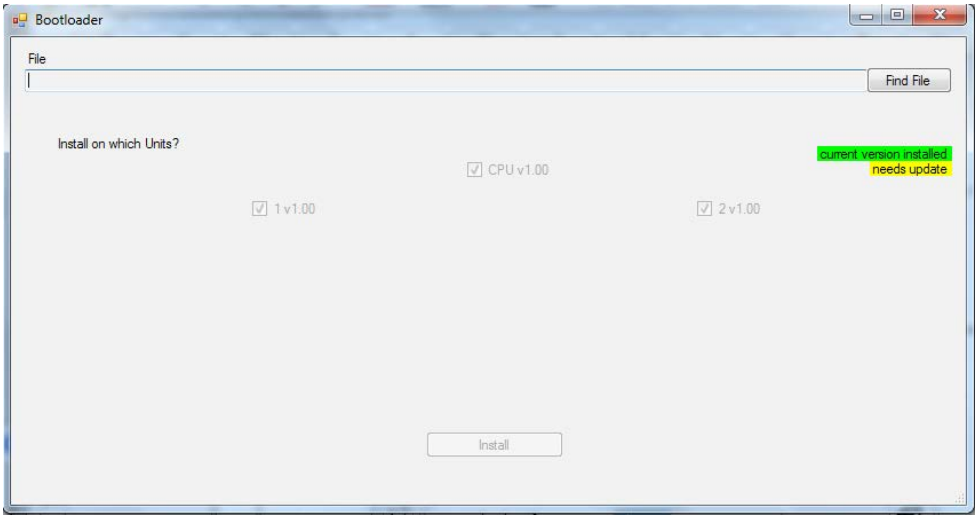
Velocio PLCs are designed so that you can update their operational software in the field. Software updates become available when new features are added. Sometimes software updates are made when a bug is discovered and fixed.

Any available PLC software, for download, is available at the Velocio website - Velocio.net. Download the software from the web site.



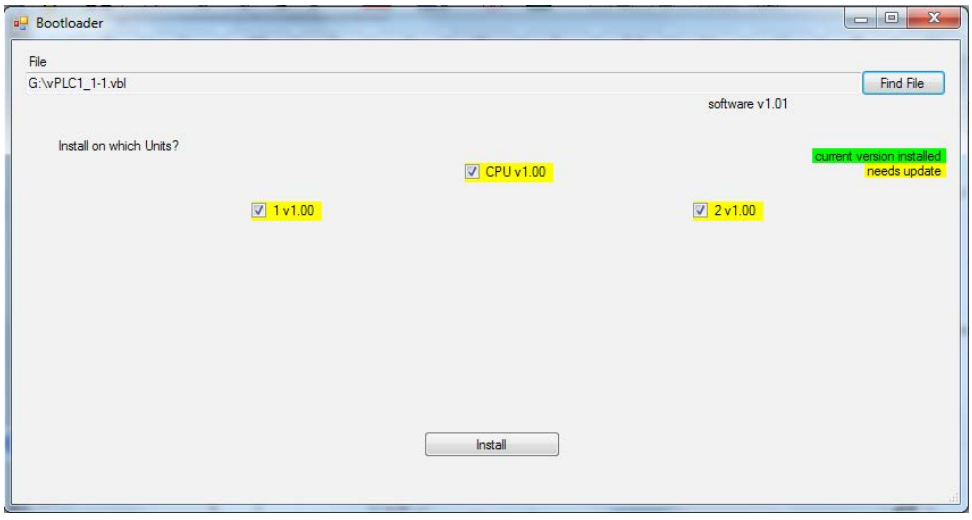
In order to download software to one or more PLCs, the PLC(s) must be connected to the PC with vBuilder and powered up. Check the USB symbol in the lower right hand corner to verify the USB attachement.

Select 'Bootload' under the Help menu. A window, like the one shown below, will pop up. It will show the PLCs that are currently attached and the version of software in each unit. For example, the screen on the right shows that there is a Branch and two Expansions. All three units have version 1.00 software in them.



Ace, Branch and Branch Expansion PLCs all use the same software download.

Click the "Find file" button on the top right. Browse to find the PLC software to download. For an Ace, Branch or Expansion PLC, the software will be named vPLC1_x-x.vbl. x-x is the version number. Select it and press Open.

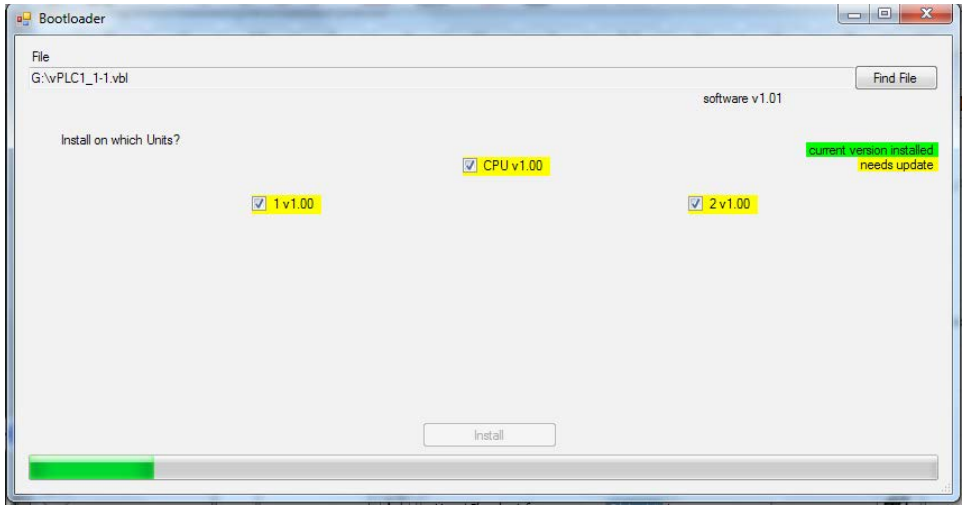


The window will display the download file that you have selected. It will also show the following.

- The version of the software in the selected file will be shown below the File selection, on the right had side.
- If the version number is greater than the version number in each PLC unit, the selection check box for that unit will be auto selected and the PLC will be shown with a yellow background.
- If the version number of the selected file is less than or equal to the version number in each PLC, the PLC will be shown with a green background and the selection box will be unchecked.

The selection box next to each PLC determines whether the selected software will be downloaded to that PLC. You can click to select or deselect any PLC's selection checkbox.

When yo have the units selected for download, click 'Install'. After a couple of seconds of delay, a progress bar should begin to move across the bottom of the Bootloader window.



After the bootload is complete, the screen will update to show the software version of in each PLC unit.

When the bootloader finishes downloading a unit that is attached to the USB cable (an Ace or a Branch), the USB detaches and reattaches. On some PCs, this might not always work properly, and the software version display will not update. Generally, if this happens, the USB icon will be missing from the lower right hand corner of vBuilder. If that should happen, unplug, then replug the USB cable, close the Bootloader, then reopen it. The Bootloader window should show the software currently in each unit.