White Paper on How to Apply Encapsulation (an Object-Oriented Programming (OOP) Idea) to Programmable Logic Controller (PLC) Software Development.

By Eric Rouse

The benefits of data encapsulation and abstraction (common attributes of object oriented programming) have long been realized in more traditional programming environments. Languages like Visual Basic, C# and Python provide PC programmers easy access to these benefits. But what about PLCs? Is there a benefit for PLC programs to also follow these structures? If so, how can a sort of "object oriented" approach to abstraction and encapsulation be implemented on PLC architectures?

At the outset, it must be noted that PLCs and PCs are very different pieces of hardware, designed to solve very different problems. There are too many differences to list, but two are of primary importance here:

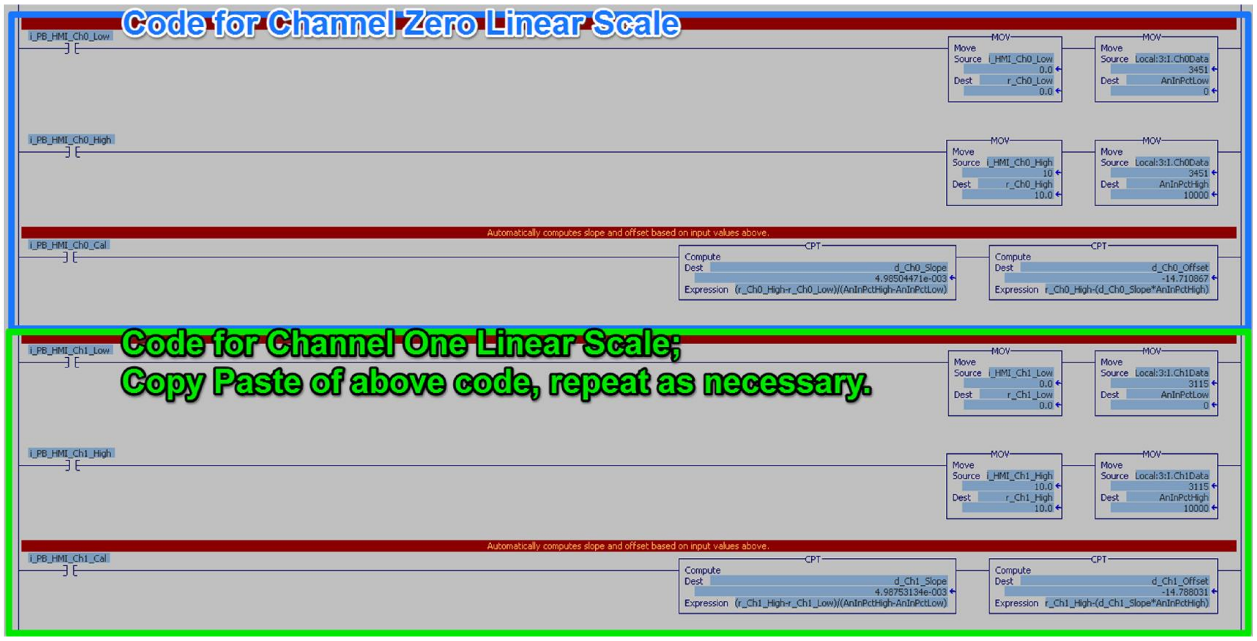
1. PLCs are most often programmed in Ladder Logic, a graphical, rule-base programming language. As such, it is fundamentally different from the procedural languages that are most often used on PCs. In ladder logic, each logical element, called a "rung," represents a rule that must be followed. If certain input conditions are true, then the output conditions are applied. PC programming usually takes an imperative approach, defining steps that programs must take to reach a desired end.
2. Another fundamental difference between the PLC and PC environments is the matter of memory management. Any PLC memory to be used during execution is reserved at compile time. It has static memory and data types. A PC has the ability to allocate memory dynamically, even while a program is executing.

OOP makes sense for PLCs because PLCs control literal, real-world objects. They read sensors, turn on valves, start motors, communicate with databases; some can even execute complex servo motion. It would be ideal if one could simply instantiate a "servo" object and all the necessary data structures were implemented to control the servo. Unfortunately Ladder Logic is not designed to do that.

But, with careful implementation, Ladder Logic can have the OOP-like encapsulation abilities. Here, encapsulation refers to the act of bringing all the control code and data structures necessary to control a particular object or device into one place. Preferably with the ability to abstract, or hide away the details, making the device code a kind of "black box."

Take, for instance, an analog input. Almost all analog inputs need to be scaled, and they are almost always scaled linearly.

A common practice to accomplish this is to write a routine that has unique code for every single analog input, as shown in Figure 1. This method is the one of the simplest forms of attempting to encapsulate PLC code. It is a good start, but it has many problems. If there is a mistake in the code, there are many places that need to be searched out and fixed. In this case there are 192 analog inputs. And if there is a copy/paste/search/replace error it can be extremely difficult to track down.


Code for Channel Zero Linear Scale
Code for Channel One Linear Scale;
Copy Paste of above code, repeat as necessary.

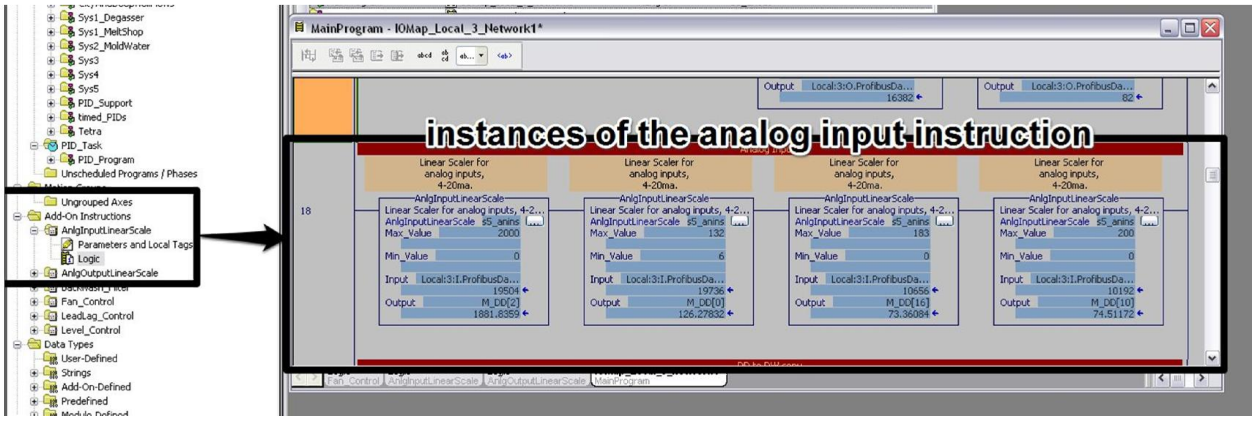
```
i_PB_HMI_Ch0_Low
  MOV  Move  Source i_HMI_Ch0_Low 0.0   Dest r_Ch0_Low 0.0
  MOV  Move  Source Local:3:I.Ch0Data 3451   Dest AnInPctLow 0

i_PB_HMI_Ch0_High
  MOV  Move  Source i_HMI_Ch0_High 10   Dest r_Ch0_High 10.0
  MOV  Move  Source Local:3:I.Ch0Data 3451   Dest AnInPctHigh 10000

Automatically computes slope and offset based on input values above.
i_PB_HMI_Ch0_Cal
  CPT  Compute  Dest d_Ch0_Slope 4.98504471e-003
       Expression (r_Ch0_High-r_Ch0_Low)/(AnInPctHigh-AnInPctLow)
  CPT  Compute  Dest d_Ch0_Offset -14.710867
       Expression r_Ch0_High-(d_Ch0_Slope*AnInPctHigh)

i_PB_HMI_Ch1_Low
  MOV  Move  Source i_HMI_Ch1_Low 0.0   Dest r_Ch1_Low 0.0
  MOV  Move  Source Local:3:I.Ch1Data 3115   Dest AnInPctLow 0

i_PB_HMI_Ch1_High
  MOV  Move  Source i_HMI_Ch1_High 10.0   Dest r_Ch1_High 10.0
  MOV  Move  Source Local:3:I.Ch1Data 3115   Dest AnInPctHigh 10000

Automatically computes slope and offset based on input values above.
i_PB_HMI_Ch1_Cal
  CPT  Compute  Dest d_Ch1_Slope 4.98753134e-003
       Expression (r_Ch1_High-r_Ch1_Low)/(AnInPctHigh-AnInPctLow)
  CPT  Compute  Dest d_Ch1_Offset -14.788031
       Expression r_Ch1_High-(d_Ch1_Slope*AnInPctHigh)
```

Figure 1 - Repeating Code; Copy and Paste and Search and Replace

Another method is to separate the unique code into its own routine, then copy/paste/search/replace the routine. This doesn't solve either of the previous problems, but it does make code harder to read. As such, it is not recommended.

A slight improvement to this method is to have a separate program and create copies of it using local tags. At least in this case the code and data are kept together. So, encapsulation is attained, but it uses a lot of PLC overhead for very limited gains.

The best way, at least on PLC platforms that offer the ability, is to write one's own instructions. For instance, RSLogix5000, from Rockwell Automation, offers the Add-On Instruction (AOI). So the analog input scaling is done in a single program block, like in Figure 2.



```
Output  Local:3:O.ProfibusDa...  16382        Output  Local:3:O.ProfibusDa...  82

18
Linear Scaler for analog inputs, 4-20ma.
AnlgInputLinearScale
Linear Scaler for analog inputs, 4-2...
AnlgInputLinearScale  $5_anins
Max_Value  2000
Min_Value  0
Input   Local:3:I.ProfibusDa...  19504
Output  M_DD[2]  1881.8359

Linear Scaler for analog inputs, 4-20ma.
AnlgInputLinearScale
Linear Scaler for analog inputs, 4-2...
AnlgInputLinearScale  $5_anins
Max_Value  132
Min_Value  6
Input   Local:3:I.ProfibusDa...  19736
Output  M_DD[0]  126.27832

Linear Scaler for analog inputs, 4-20ma.
AnlgInputLinearScale
Linear Scaler for analog inputs, 4-2...
AnlgInputLinearScale  $5_anins
Max_Value  183
Min_Value  0
Input   Local:3:I.ProfibusDa...  10656
Output  M_DD[16]  73.36084

Linear Scaler for analog inputs, 4-20ma.
AnlgInputLinearScale
Linear Scaler for analog inputs, 4-2...
AnlgInputLinearScale  $5_anins
Max_Value  200
Min_Value  0
Input   Local:3:I.ProfibusDa...  10192
Output  M_DD[10]  74.51172
```

Figure 2 - Add-On Instruction provides very good encapsulation.

This method solves all those problems. If the code is wrong, it need only be edited in one place. The rest of the program references that AOI. Change it once, and it updates everywhere. Also, copy and paste errors are bypassed.

There are further implications; often there are many objects on a machine or parts of a process that operate/interact the same way. Motors are a good example. Every motor has interlocks, and starts and stops in some way. The particulars of motor control code can be solved once, then implemented everywhere. Figure 3 shows a single rung that starts 3 motors. This would normally take several lines of code. Or, take a machine or process with multiple robots, for instance. All the code necessary for each robot can be written, tested and then instantiated multiple times. This makes the code very scalable/adaptive.

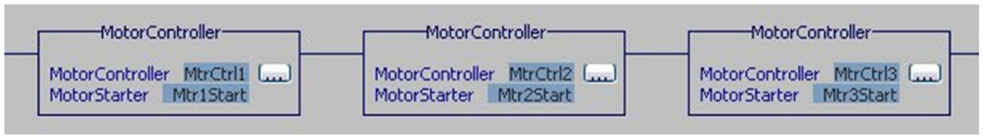


Figure 3 - Three motors controlled in a single rung.

Encapsulation is a great method for abstracting low-level details from main control code. It can be hard to implement on PLC architectures, but there are some methods that can approximate it. The best technique is to create one’s own instructions.