PLC Programming goes Imperative

Decades ago, computer science emerged from the dark ages of assembly language programming and created two new languages: Lisp and Fortran. These are two very important computer languages because they exist at opposite ends of an imagined spectrum in the eyes of computer scientists: functional languages vs. imperative languages.

Fortran "won" the first battle, not least because imperative languages are closer to how the CPU actually does things, so back in the day when every little CPU cycle mattered it was easier to understand the performance implications of a Fortran program than a Lisp program. Plus, if you were already programming in assembly, then you were already thinking about how the computer was executing your code. In fact, the next big imperative language, C, is often referred to as "portable assembly language." Fast forward to now, and modern languages like C#, Java, Python and Ruby have all grafted a lot of functional programming features onto their imperative programming basic syntax. In C#, for instance, Linq is a direct rip-off of Lisp's S-Expressions and it now has Closures and lambda functions. Functional languages provide ways to think at a higher level than imperative languages. In a functional program you describe what you want and in an imperative program you describe how to do it.

Here's an example in C#, using imperative programming:

```
var data = new int[] { 1, 2, 3, 4, 5 };
var sumOfSquares = 0;
for(var i = 0; i < data.Length; i++)
{
    sumOfSquares += data[i] * data[i];
}
```

…and the same thing done functionally:

```
var data = new int[] { 1, 2, 3, 4, 5 };
var sumOfSquares = data.Select(x => x * x).Sum();
```

In the second case, I'm taking the list of numbers, using Select to translate that into a list of their squares (also known as a Map operation) and then using Sum on the resulting list to compute an aggregate sum (also known as a Reduce operation). It has some interesting advantages. For instance, the original code can't be split across multiple cores, but the latter can. Also, if you know both syntaxes, the latter is easier to read and understand.

Now take ladder logic. I've made the claim before that basic ladder logic (with contacts and coils) is actually a functional language. A simple example might be ANDing two inputs to get an output, which in C# would look like this:

```
var output = inputA && inputB;
```

That's actually functional. If I wanted to write it imperatively I'd have to do something like:

```
var output = false;
if(inputA && inputB)
{
  output = true;
}
```

In ladder logic, that would be the equivalent of using an unlatch (or reset) instruction to turn off an output and then using a latch (or set) instruction to turn on the output if the A and B contacts were true. Clearly that's not considered "good" ladder logic.

Similarly, a start/stop circuit goes like this:

```
var run = (start || run) && !stop;
```

Now historically, mathematicians and physicists preferred functional languages because they just wanted to describe what they wanted, not how to do it. It's worth noting that electricians, looking at ladder logic, prefer to see functional logic (with contacts and coils) rather than imperative logic (with sets, resets, and move instructions).

In recent years we've seen all major PLC brands start to include the full set of IEC-61131-3 languages, and the most popular alternative to ladder logic is structured text. Now that it's available, there are a lot of newer automation programmers who only ever knew imperative programming and never took the time to learn ladder logic properly, and they just start writing all of their logic in structured text. That's why we're seeing automation programming slowly shift away from the functional language (ladder) towards the imperative language (structured text).

Now I'm not suggesting that structured text is bad. I prefer to have more tools at my disposal, and there are definitely times when structured text is the correct choice for automation programming. However, I'd like to point out that the history of computer science has been a progressive shift away from Fortran-like imperative languages towards Lisp-like functional languages. At the same time, we're seeing automation programming move in the opposite direction, and I think alarm bells should be going off.

It's up to each of us to make an intelligent decision about what language to choose. In that respect, I want everyone to think about how your brain is working when you program in an imperative style vs. a functional style.

When you're doing imperative programming, you're holding a model of the computer in your mind, with its memory locations and CPU and you're "playing computer" in your head, simulating the effect of each instruction on the overall state of the CPU and memory. It's only your intimate knowledge of how computers work that actually allows you to do this, and it's the average electrician's inability to do this which makes them dislike structured text, sets, resets, and move instruction. They know how relays work, and they don't know how CPUs work.

If you know how CPUs work, then I understand why you want to use structured text for everything. However, if you want electricians to read your logic, then you can't wish-away the fact that they aren't going to "get" it.

As always, be honest with yourself about who will read your logic, and choose your implementation appropriately.

4

This entry was posted in Industrial Automation and tagged ladder-logic, plc on March 26, 2017 [http://www.contactandcoil.com/automation/industrial-automation/plc-programming-goes-imperative/] .

7 thoughts on "PLC Programming goes Imperative"

Ed Stevens
March 27, 2017 at 1:20 am

Totally agree that there's a general trend towards functional programming, also agree that it's a good trend.

Your argument about Ladder being a functional language and structured text is also interesting.

What I don't believe is that the shift from ladder towards structured text has too much to do with the former being more functional than the latter.

The shift is rather caused by on the one hand new generations of engineers being more comfortable with ST, and on the other hand programs becoming bigger and needing to handle more data.

The concept of Functional Programming is different from a Functional Language. Both LAD and ST allow for a functional way of programming, the same way as both allow for making a mess of it.

Scott Whitlock Post author
March 27, 2017 at 7:08 pm

@EdStevens – True. My main concern is that I'm seeing ST replace the lowest level logic of the program, where LD is likely better suited.

There are many cases where, e.g., you never want cylinder "A" to extend if cylinder "B" isn't retracted, and this is both better expressed and better visualized in ladder. When things go wrong with a machine, it's typically a sensor problem or a mechanical issue. You know cylinder "A" should be extending, so you can just see that cylinder "B" retracted isn't on, so you go and check the sensor.

If you're setting the Extend Cylinder "A" output from 5 different places in your ST program, your program might be correct, but it's almost impossible for someone unfamiliar with the code to figure out why it's not working.

Now sure, if you're going to have a single ST program that's responsible for turning on the extend output and you only ever set that output from a single line, then sure, you're back to writing functional code. For me, that raises two more issues: is the ST editor as good of a debugging tool as the LD editor (can I see the state of each variable)? And can your average ST programmer who was raised on C resist the urge to set that output variable from more than one place?

I don't have a problem with people using ST if they've thought through some of these issues, but from the examples I've seen, I don't think that's the case.

Bill
April 3, 2017 at 6:19 pm

Thanks for the very thoughtful article. I personally think the major difference between ST and Ladder is that in ST you can easily skip blocks code from execution. That is efficient for implementing, for example, a state-machine. It's not too difficult to program functionally in ST for a Ladder programmer, and software like TwinCAT or Codesys makes it easy for online debugging.

In my mind, Ladder is only a functional language for bit operations. We will probably never get true functional languages like Erlang for automation applications due to reliance on recursions and inherently not time deterministic.

Ryan Maw

April 26, 2017 at 12:12 am

I have a lot of function blocks written in ST (TwinCAT) and the line-for-line code-equivalent function blocks in Ladder (RSLogix5000). Neither language is inherently more imperative. We can write functional or imperative code in either language just as easily. However, I can see how someone coming from a more mainstream programming background may pick up ST and write functional style code. Having written a lot of Javascript and a lot of Ladder over the years, when I first started writing ST it took me a few minutes before I realized that I should write things like:

output := condition and (input1 or input2);

As opposed to an if..then..else statement. I'm not sure this would have occurred to me if I hadn't already had a lot of experience in Ladder and been translating a Ladder routine to ST at the time. The problem, then, is not ST itself but how it makes programmers, who are new to PLCs, comfortable in doing new things using familiar old patterns.

The advantage I find with ST is that it's easy to type, copy, paste, email… and its much more compact. The big advantage I find with ladder is that I can intuitively understand the basics of what a rung does with just a glance. A rung is a picture that has meaning, and our brains interpret pictures much faster than text, even "structured" text. Online debugging is also easier because the picture comes alive with "power flow" highlights, so now its almost like watching a machine work (Ladder) instead of reading about how it works (ST). At times, I find these advantages compelling enough that I wonder why the computer science world hasn't invented a graphical language for mainstream computer programming. Giving instructions to computers in a string of monotone words seems like we are catering to the computer's nature more than our own. I suspect it's because, when writing the code, the patterns aren't as important to us as when reading, and because making computers smarter is expensive, and even because there is a geeky cachet that comes with knowing how to use a convoluted tool.

Structured Text is just that: text with some supporting structure. Ladder is the inverse: structure with a little supporting text. One day I hope to have a tablet with an app where I can just draw the rungs, contacts, and coils! RSLogix is pretty good, but if we had better ladder editing tools, the language would be more appealing to newcomers.

Jasper Felix
May 23, 2017 at 6:05 am

PLC is a digital computer used for automation of typically industrial electromechanical processes, such as control of machinery on factory assembly lines, amusement rides, or light fixtures. Great post full of useful tips.

Thomas
May 29, 2021 at 5:51 am

Wish we would have Lambda Expressions within Structured Text. That would be a big plus. Structured text would be easier to adapt for software engineers familiar with high level languages as C# and Java.

Furthermore the Structured Text code would be much cleaner if something like Lambda Expressions or LINQ would be possible.

Scott Whitlock Post author
June 1, 2021 at 8:17 am

Yes, lambda expressions would be awesome!