Tricks of the Trade: SCADA Development Shortcuts

Always be on the lookout for Time-Saving techniques

By Shawn Hustins and Graham Nasby

The craft of SCADA software development is one that requires a multitude of skills. The average SCADA system includes a wide variety of components and modules that all have to work together to make the system work. The effort and time to write code for SCADA systems can also vary greatly depending on what component or module is being developed.

There are many instances where we find ourselves bogged down some of the aspects of programming SCADA systems. No matter the size and complexity of a project, there will always be tedious elements involved. As they mount up, these simpler tasks can put quite a burden on progress towards project completion when they're not planned for and dealt with accordingly.

But what methods are available to combat these issues? The traditional approach is to divide up the work, so that individuals with particular skills in certain areas can work on the area as per their specialty. But what about those mundane tasks like setting up hundreds of data tags, or populating data arrays, or setting up a large number of similar workstations? To be honest, no one wants to do those!

When looking at the issue from a developer's point of view, there are potentially much more powerful solutions available with the aid of the computer. The more mindless a task is, the easier it is to tell a computer how to do it. (Humans, on the other hand, are much better suited for the more creative aspects of software development.) So although it is highly unlikely one could automate their entire job, it's often possible to come up with ways to automated some of the more repetitive or dull aspects. Automating tasks does not have to be difficult; often just a few lines of scripting can be used to free up time for more interesting (and productive) work. Automation also has the benefit of reducing mistakes and improving consistency.

Here are a few examples of time saving solutions that I have been using lately. Keep in mind that every SCADA project, and its associated programming environment, is unique. As such, the intention of the following is not to provide the best/only way of doing it, but rather to serve as examples. Hopefully these case studies can give you some ideas about how to automate parts of your own software development.

Note that these examples come from a SCADA programming environment which consists of Allen-Bradley PLCs and GE Proficy iFix software. Other environments will likely have their own opportunities for time saving techniques as well.

Automatically Generating PLC Code

In this example, an Excel spreadsheet was used to auto-generate ladder logic code segments that can be pasted into a larger PLC program.

For this project, we had an operator interface terminal (GE QuickPanel+) which we wanted use to read real-time data from an Allen-Bradley CompactLogix PLC via the serial port. We were using the GE QuickPanel product because it has a store and forward collector feature that tightly integrates in with our process historian to provide us with redundant data-logging to our normal PLC-to-historian data-logging.

For reasons of communications redundancy, we chose to have the QuickPanel+ read data via its serial port rather than via the PLC's Ethernet connection. The challenge was that the serial connection from the QuickPanel needed to use the older Allen-Bradley DF1 protocol, which does not support the CompactLogix's tag based memory registers. Thus, to get this to work we would have to create a special outbox array in the CompactLogix and then map it to a numbered memory file number that could be read by the DF1 communications. Part of this work also involved writing a ladder logic routine in the CompactLogix that would copy and scale all the various data tags into the outbox array. Writing this ladder program would normally be a tedious task, but instead we used Excel and scripting to automate the code generation.

Upon analysis, the programming task manifested itself as follows: given the list of tags to be accessed on the Logix controller by an external device, create a ladder logic routine to copy the tags into an array and assign each tag on the external device to its new address.

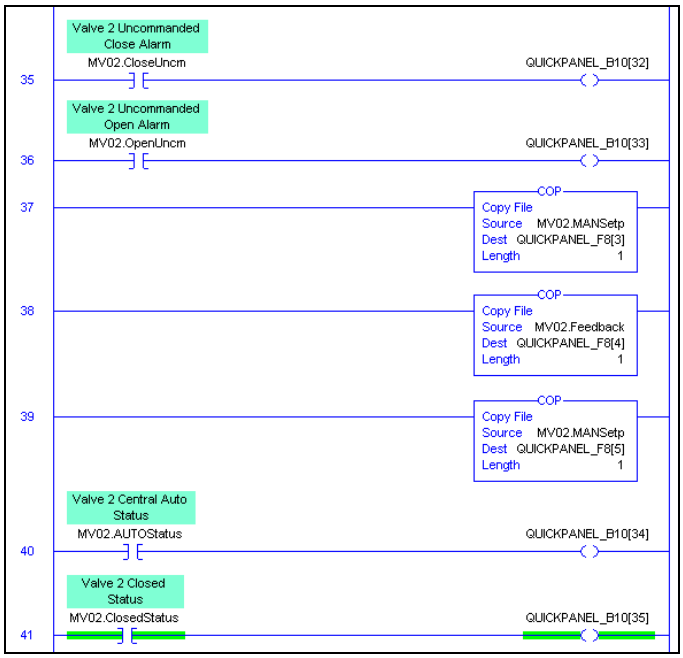

Figure 1 – PLC Ladder logic rungs in the RS Logix 5000 programming environment that copy data from the PLCs tag-based memory registers into the outbox array

One of the key points to note in this task is the repetition; each rung has to contain instructions to move data from the memory tag into the outbox array. This simple pattern provoked the idea of creating a few simple spreadsheet formulas that would generate all the ladder logic code and the addresses that would be imported to the tag database of the external device. To feed the spreadsheet formulas, we exported a list of tags from using a tool within the RSLogix 5000 PLC Programing software.

A	B	C	D	E	F	G	H	I	J	K
START RUNG	RUNG LOGIC				END RUNG		TAG TYPE	IS BOOL	IS REAL	I/O Addr. to Read
SOR	XIC	MV01.ModeReq	OTE	QP_B10[5]	EOR		BOOL	1	0	B10:0/5
SOR	XIC	MV01.MANCloseReq	OTE	QP_B10[6]	EOR		BOOL	1	0	B10:0/6
SOR	XIC	MV01.MANOpenReq	OTE	QP_B10[7]	EOR		BOOL	1	0	B10:0/7
SOR	XIC	MV01.PosFBErrorEnable	OTE	QP_B10[8]	EOR		BOOL	1	0	B10:0/8
SOR	XIC	MV01_IN.Failed	OTE	QP_B10[9]	EOR		BOOL	1	0	B10:0/9
SOR	XIC	MV01.PosDevAlarm	OTE	QP_B10[10]	EOR		BOOL	1	0	B10:0/10
SOR	XIC	MV01.CloseFail	OTE	QP_B10[11]	EOR		BOOL	1	0	B10:0/11
SOR	XIC	MV01.OpenFail	OTE	QP_B10[12]	EOR		BOOL	1	0	B10:0/12
SOR	XIC	MV01.GenAlarm	OTE	QP_B10[13]	EOR		BOOL	1	0	B10:0/13
SOR	XIC	MV01.PosFBErrorAlarm	OTE	QP_B10[14]	EOR		BOOL	1	0	B10:0/14
SOR	XIC	MV01.CloseUncm	OTE	QP_B10[15]	EOR		BOOL	1	0	B10:0/15
SOR	XIC	MV01.OpenUnCM	OTE	QP_B10[16]	EOR		BOOL	1	0	B10:1/0
SOR	COP	MV01.MANSetp	QP_F8[0]	1	EOR		LREAL	0	1	F8:0
SOR	COP	MV01.Feedback	QP_F8[1]	1	EOR		LREAL	0	1	F8:1
SOR	COP	MV01.MANSetp	QP_F8[2]	1	EOR		LREAL	0	1	F8:2
SOR	XIC	MV01.AUTOStatus	OTE	QP_B10[17]	EOR		BOOL	1	0	B10:1/1

	B	C	D	E	G	I	J	K
1	RUNG LOGIC					IS BOOL	IS REAL	I/O Addr. to Read
2	=IF(H2="BOOL","XIC","COP")		=IF(H2="BOOL","OTE","QP_F8["&(SUM(J$2:J2) - 1)&"]")	=IF(H2="BOOL","QP_B10["&(SUM(I$2:I2) - 1)&"]",1)		=IF(H2="BOOL",1,0)	=IF(H2="LREAL",1,0)	=IF(H2="BOOL","B10:"&QUOTIENT(SUM(I$2:I2) - 1,16)&"/"&MOD(SUM(I$2:I2) - 1,16),"F8:"&(SUM(J$2:J2) - 1))
3	=IF(H3="BOOL","XIC","COP")		=IF(H3="BOOL","OTE","QP_F8["&(SUM(J$2:J3) - 1)&"]")	=IF(H3="BOOL","QP_B10["&(SUM(I$2:I3) - 1)&"]",1)		=IF(H3="BOOL",1,0)	=IF(H3="LREAL",1,0)	=IF(H3="BOOL","B10:"&QUOTIENT(SUM(I$2:I3) - 1,16)&"/"&MOD(SUM(I$2:I3) - 1,16),"F8:"&(SUM(J$2:J3) - 1))
4	=IF(H4="BOOL","XIC","COP")		=IF(H4="BOOL","OTE","QP_F8["&(SUM(J$2:J4) - 1)&"]")	=IF(H4="BOOL","QP_B10["&(SUM(I$2:I4) - 1)&"]",1)		=IF(H4="BOOL",1,0)	=IF(H4="LREAL",1,0)	=IF(H4="BOOL","B10:"&QUOTIENT(SUM(I$2:I4) - 1,16)&"/"&MOD(SUM(I$2:I4) - 1,16),"F8:"&(SUM(J$2:J4) - 1))
5	=IF(H5="BOOL","XIC","COP")		=IF(H5="BOOL","OTE","QP_F8["&(SUM(J$2:J5) - 1)&"]")	=IF(H5="BOOL","QP_B10["&(SUM(I$2:I5) - 1)&"]",1)		=IF(H5="BOOL",1,0)	=IF(H5="LREAL",1,0)	=IF(H5="BOOL","B10:"&QUOTIENT(SUM(I$2:I5) - 1,16)&"/"&MOD(SUM(I$2:I5) - 1,16),"F8:"&(SUM(J$2:J5) - 1))

Figure 2 – (top) Excel spreadsheet that was used to auto-generate the code, inputs are highlighted in blue, (bottom) close up of the formulas used to assemble the Instruction mnemonics for each Ladder Rung

Each rung is written as follows: (a) Start of rung, (b) if the tag is a Boolean value, move it into the array using relay logic, (c) if it is a real value and can be moved using the MOV instruction and associated parameters, (d) end of rung. Since all the values needed to be placed in two distinct arrays – one for Booleans and one for Floats, it was very simple to keep count of indices on each. Each array has its elements numbered as [0], [1], [2], etc. Additionally, a column was added to record the array address, which was then imported back to the QuickPanel's tag database.

The 10 minutes of time required to put this together allowed much more time worth of manually written code to be generated in just seconds. Estimated time savings: 6 hours.

Populating Memory Register Data into PLC Code Files

In this example, we used text files (and a text editor) to side load a large number of memory register values into a PLC.

In most PLC programming environments there is usually an interface provided for hand-entering in data register values, both for offline and online editing. These PLC programming tools work well for entering a few values at a time or for viewing/editing values for troubleshooting. However, for entering large numbers of values, using these provided data file editing tools can quickly become tedious (and error prone). Thus when we needed to enter several 100 element lookup tables into a PLC to calculate water volumes in ellipsoid water towers, we needed to find a better way.

Since the PLCs we were using for this project were Allen-Bradley SLC500s, the programming environment was a package called RSLogix 500. The RSLogix 500 software provides several options for saving PLC code files. The first is the native .RSS file, which is a binary format. The second is a .SLC file, which although it is less space efficient, it has the feature of being in plant text. It is through the .SLC files that we were able to automate the task of entering in several 100 element lookup tables that will be used to calculate the volume of several ellipsoid water towers based on level readings.

So the next logical step would be to open up the SLC file in a text editor (for example, Notepad++) and start edit the memory register values in place, and then load it back into the PLC program. However, we decided to go one step further.

Our tool of choice was to use Microsoft Excel in conjunction with a text editor. Using the text editor, copied the memory register portions of the SLC file (were tab delimited across columns, and rows separated by Carriage Returns) into the clipboard. We then pasted these into Excel, and Excel automatically neatly organized the data values into columns/rows. The Excel interface was much easier/faster for editing the data value ranges. After the edits were complete, we then copied/pasted the data from Excel back into the SLC file. After a few minor formatting adjustments, we could then import this back into the SLC Program. Estimated time savings: 12 hours.

Food for thought: The nice thing about text files (and their cousin XML files) is that they are readily adaptable for a wide variety of editing techniques. Most modern PLC programming environments now have text-based or XML-based file import/export options. The real beauty of being able to use text files is that it provides the opportunity for any developer to make their own tools that are not offered within the usual PLC programming environment. Also, depending on the capabilities of the text export/import file format, there may be other aspects of the PLC program that could be manually edited using a text-based tool, such as ladder instructions, data file setup, program files, or other configuration aspects.

Data File F8 -- FLOAT -- Analog Input Scaled Values

Offset	0	1	2	3	4
F8:0	2209.045	10.20678	0.9186632	0.6409279	84.67496
F8:5	0	4.807195	507.2083	719.8068	17.45002
F8:10	0	0.5417366	3.245957	3.329141	1.373417
F8:15	0	-23.8806	1.192202	0.9785594	32.64917
F8:20	16.84725	12.61254	719.6816	2.524645	19.65918
F8:25	0	0	0	0	0
F8:30	0	0	919.8077	0	0
F8:35	0	480.141	0	0	0
F8:40	84.50328	1.373417	0	0	0
F8:45	104.5514	0.5951473	103.9562	716.6048	0
F8:50	0	0.01	6	16	0
F8:55	0	0	0	0	0
F8:60	430	11.58019	17.45918	1.373417	0
F8:65	-30.2	37.04	3200	13738	-28
F8:70	30	3000	11000	-30.2	37.04

Figure 3 – Snapshot of SLC file showing data register values organized in rows/columns – ideal for importing into a spreadsheet program like Excel for large scale edits

Deployment of Multiple SCADA View Terminals

In this example, virtual machines were used to rapidly deploy six new SCADA view terminals. Using virtual machines (VMs) that run on top of a hypervisor significantly reduces amount of time to deploy multiple similar computers.

Traditionally setting up a new SCADA view terminal would involve getting a new desktop computer, installing the operator system on it, tweaking the operating system settings, installing the SCADA view software, installing several helper application and hot fixes, and then configuring the SCADA software. The focus being consistency to ensure that the terminals are identical so the user experience would be consistent and correct. The entire process would typically take about 2-3 days to complete from start to finish, and have to be repeated for each SCADA view terminal.

Not wanting to spend 12-15 days setting up view terminals, we instead adopted a much more time efficient approach. We went to an existing SCADA view terminal and took an image from it in the form of a virtual machine (VM). We used a hypervisor utility to do this, and most hypervisors come with a VM creating tool such as this.

We then installed a copy of Windows 7 onto a new desktop machine, and installed the hypervisor software, which in our case was VMware Workstation. The next step was to copy over the VM over to the computer, configuring it to use a new static IP address, and installing the required licensing keys. Lastly, we added a script to automatically start up the VM and the SCADA software within it whenever the computer was powered on. For the rest of the terminals we just repeated the same process (including using a tool called Norton Ghost to copy the operating system images), which reduced our deployment time down to about 2 hours per view terminal.

Our last step was to prepare a document that outlined the entire process, which we saved into a project folder so we would have the tools and procedure readily available for next time. As part of this project we also made backup copies of the VM, so we can quickly/easily rebuilt a corrupted view terminal if the need arises. Estimated time savings: 60 hours

Why Time Savings are Important

When looking at most SCADA software development, the 80-20 Pareto rule usually applies. About 20% of the programming will be complex and require careful thought on how to design and program. The remaining 80% will often be mundane, and be tedious and time consuming to implement.

Ironically when it comes to debugging, the reverse is usually true: it's usually that 20% of the code (the complex part) that requires 80% of the effort during the testing/adjusting phase. Thus it's important to try to reduce the amount of time that has to be spent on programming mundane parts of SCADA programs; this will allow you to spend more time working on the more complex (and more interesting) portion of your code.

Guidelines for Identifying Time Saving Opportunities

Here are some guidelines how to identify opportunities for automating parts of your SCADA programming workflow:

Always get familiar with the software tools used on the job. Many offer import and export tools or additional file formatting options. This may provide alternate methods of data manipulation or data transfer between software. Text-based files and comma separated value files can sometimes be quicker to interact with than the default software tools provided by the vendor.

Keep an eye out for any discernable patterns that arise in work being done. Identifying task that have to be repeated over and over is usually a good indicator. One method is to take a few moments, before beginning any programming task, to think about how a computer might complete the job automatically. What inputs would it require? What output is desired? What processing or logic would be needed? Etc.

The most important factor to consider is the return on investment in terms of time. The time taken to develop an automated tool must be weighed against the time that would be saved with its use. If the investment doesn't appear to be worthwhile, are there other use cases that can be tied in? Can the task be broken down further to an automatic portion and manual portion? Some tasks may not be worth automating.

Make sure to document it! Regardless of whether your automated solution is a quick-and-dirty hack or a carefully-thought-out utility, always take the time to write up a short README file on how it works and how to use it. Also gather all the required files into a project folder, plus some examples. This will allow you to use to the tool again in the future, when no-one including yourself can remember how it was used the first time. This sort of documentation also comes in handy for future troubleshooting of automatically generated code.

Looking Ahead

No matter the project, there are usually opportunities to use automation to reduce the time and effort required to complete certain tasks on the job. Investing a little extra brainpower into the development process can yield big results. Just remember to actively evaluate each task as it manifests itself and always think about where development time can be saved.

About the Authors

Shawn Hustins is a third year engineering student at the University of Guelph. He recently completed a SCADA developer co-op placement at Guelph Water. Shawn, originally from Mississauga Ontario, is majoring in the B.Sc.(Eng.) Engineering Systems and Computing program. Contact: hustinss@uoguelph.ca

Graham Nasby, P.Eng, PMP, CAP holds the position of Water SCADA & Security Specialist at City of Guelph Water Services, a publicly-owned water utility located in Guelph, Ontario, Canada. Graham is the co-chair of the IS112 SCADA Systems standards committee. Contact: graham.nasby@guelph.ca