Developing Safety Critical Code in Labview ECE 3502 : Mariobot

Joseph Weate (jfw6fp)

01May2015

With an every growing reliance on safety critical software, it is imperative that programmers following strict coding guidelines. Deviating from a set of guidelines could cause unintended situations. As was the case in Toyota's lawsuit for their Camry model accelerating unexpectedly. Not only did Toyota find themselves facing over a billion dollars in fines, innocent people lost their lives. Safety critical code should always be considered a programmers' requirement.

When working in a team, as is the setting for University of Virginia's Mariobot course, it is the upmost importance that all team members are aware of and follow the same practices. This will help account for simple mistakes that could cause life threatening results. There are lists upon lists of general rules to adhere to when programming, making it a nuisance to follow. Out of the seemingly infinite options that go into fine detail about both what to do and what not to do, a simplified list of 8 rules has been generated. While these rules do not cover all that needs to be done, it creates a point that is worth starting out.

Some of the rules have been modified specifically for data-flow programming such as Labview, and other rules have been added specifically for the type of programming.

-------------------------------------------------------------------------------------------------------------------------------

**Rule 1:** *Document your style practices to ensure consistency (*"Development Guide." *)*

Especially when programming in a team, always follow the same styling conventions. During the planning stage of a project, a team should go over and document what styles they will use on both the front and backend. This includes but is not limited to fonts, text characteristics, wiring techniques, positioning, and sizing. Labview presents the options to customize your style as in depth as you want. However, as discussed in the next rule, simplicity is often better. Having multiple colors and text fonts that symbolize different features and data flow paths complicate the matter. Too much going is distracting and causes an increase in the probability of mixing up the meaning of something. When deciding team styles, it is important to have overlap. Relying, for example, on one color being the sole indicator for a state could result in problems for factors such as color-blind or forgetful partner.

Having your team's styling practices predefined and understood will allow for more concrete building and debugging.

**Rule 2:** *"Restrict all code to very simple control flow constructs"*(Holzmann)

Avoid dynamic states- banish recursion. Keeping a simple control data flow allows for simpler analysis and reduces errors that could occur. With everything laid out, if each input produced one output, the controls would be clearer. While odds are, every program you build will not be linearly laid out, having simplified states allow you to reuse the state at any time. Simplified states also allow simple but powerful error protection to occur.

For example, we take a look at the University of Virginia's Mariobot class again. In the second lab, students were tasked with building a barcode scanner. In most implementations, the code utilized a dynamic structure to read the barcode. The state would transition back and forth before two states, light and dark, reading the barcode. Within the states, the index for the barcode was constantly changing- whether the scanner was waiting for the second, third, or forth barcode to be seen. This was done by using a global counter that was incrementing every time the stage was hit. When the counter struck four, the code would know the barcode was scanned and then go on to interpreting the barcode. The global variable would then reset. So what's wrong here? The students did not keep the data flow simple. What if the scanner didn't read the barcode? What if the scanner read the barcode but not right? What if the counter never incremented? What if the counter was actually a local variable and was lost upon leaving the state? What if the counter was never reset? What if I read 3 lines and never got a fourth?

Covering all the possible cases in a dynamic state require both a lot of thought and confusion. Keeping a simple control flow will allow for much better clarity.

**Rule 3:** *"No function should be longer than what can fit on one sheet of paper in standard format"* (Holzmann)

All code should be verifiable. If code gets too long, the process of checking all possible cases draws out and more than often one case (the one that will break) is left out. Long, spaghetti like flow generally relates to poor planning, and poor planning generally relates to problems. Keep your states understandable.

**Rule 4:** *"The code's assertion density should average to minimally two assertions per state"* (Holzmann)

Each state in data-flow programming deals with inputs and outputs. Each state should check the inputs and outputs when received and before sending them out in order to check for erroneous values. Labview allows this to happen simply by selecting a "Data Range" on the block diagrams.

**Rule 5:** *"Declare all data objects at the smallest possible level of scope"* (Holzmann)

If an object is not in scope, a state cannot access the object. The state can thus not alter the object or corrupt the object. In Labview, avoid global and local variables as often as possible. This will help avoid dynamic memory allocations and help keep performance steady. If you must use a global variable, Labview has the option to create a functional global variable. A functional global variable "does not create extra copies of data and allows certain actions such as initialize, read, write, and empty" ("Development Guide."). This will help avoid errors such as race conditions, which are difficult to debug.

**Rule 6:** *"Each calling function must check the return value of nonvoid functions"* (Holzmann)

Within states, even though the statement was reached correctly and the next part was implemented does not mean the statement executed properly. This piggybacks the idea of having a Data Range for the inputs and outputs of every state, as discussed earlier. Labview, once again, simplifies the process for us with the help of prebuilt error handling techniques, such as error VI's and clusters. If something went wrong in the middle of a state, but was somehow corrected, there is still a possibility that in the future a larger scale error can occur. Implementing Labview's error handling VI's properly will help mitigate these larger scale errors.

**Rule 7:** *"Control Redundancy"* (Olson)

Don't be afraid of redundancy, but do not over kill it. Control it. Redundancy is having two or more states do the same thing, at the same time. If one of the states fails, there is a back up. If the states produce different results, an error occurred in one of the states. Redundancy can save you in times of trouble.

Redundancy can also result in major problems such as overusing power and errors in the sensor. For example, if something is wrong with a sensor on a Mariobot used at the University of Virginia but the data flow is running parallel modules, no error will be discovered. Parallel modules will both access the sensor data, and assuming everything else is constant, will produce the same result. If the results are compared, they are the same and the state advances when it should not have.

Redundancy can be your friend so long as you control it.

**Rule 8:** *"All code must be compiled, from the first day of development, with all compiler warnings enabled at the most pedantic setting available."* (Holzmann)

First off, being able to compile the code at the end of the day ensures that when you begin work tomorrow, you won't forget why it would not compile. This ensures more transparency from day to day. All compiler warnings should be taken seriously, even if it appears to be nothing. If the compiler is getting confused, then the platform you are running your code on could easily get confused as well.

In Labview, compiler errors will help show things like unlinked states. This would be a good sign that you are using spaghetti flow. Errors like such will not allow running.

Compiler warnings in Labview will however allow you to run the program. Code modules with warnings may cause run time failures. Two examples of compiler warnings in Labview that prove useful in determining how well you followed the previously defined 7 rules are "Passing an incompatible pointer as a parameter" and "Extraneous return value" (Why Are Some). Receiving any errors should be a red flag that your product is not ready.

------------------------------------------------------------------------------------------------------------------------------

Following this set of 8 rules can help make analysis of critical data flow components more reliable. The rules outlined should be used in combination, a balance that is specific for the project and user. In using the rules it is important to realize that the rules will not ensure safety critical code, but instead create the environment for safety critical code.

"If these rules seem draconian at first, bear in mind that they are meant to make it possible to check safety-critical code where human lives can very literally depend on its correctness. The rules are like the seat belts in a car: Initially, using them is perhaps a little uncomfortable, but after a while, it becomes second nature, and not using them is unimaginable" (Holzmann)

Works Cited

1. Holzmann, Gerald J. "The Power of 10: Rules for Developing Safety- Critical Code." Software Technologies. N.p., June 2006. Web. Apr. 2015.

2. "Development Guide." Labview. National Instrument, Apr. 2003. Web. Apr. 2015. <ni.com>.

3. Olson, Zach. "LabVIEW Real-Time for Safety-Critical and Mission-Critical Applications." National Instrument. N.p., n.d. Web. Apr. 2015. <ftp://ftp.ni.com/pub/events/nits/presentations/2007/realtime.pdf>.

4. "Why Are Some Compiler Errors Returned as Warnings in LabWindows/CVI 2013?" National Instruments. N.p., June 2013. Web. Apr. 2015. <http://digital.ni.com/public.nsf/allkb/59379736F0B18A5886257B86007D2C2E>.