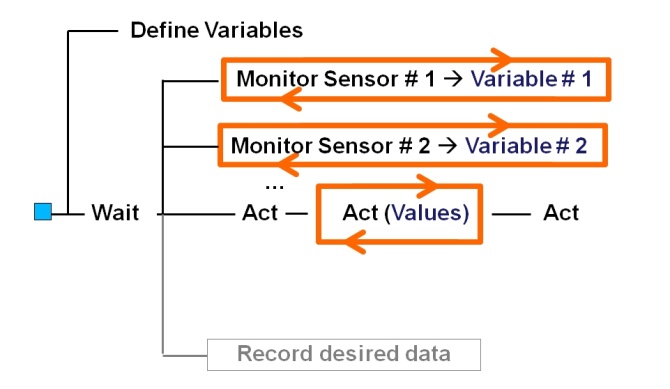
Title: Program Structure

Many times I see teams I'm supporting decide to run their GUI-based programs in a straight line from start to finish. But I'm not sure that's always a great idea.

Consider the program structure outlined below:



In this model, the first thing that happens after the blue start block is the definition of variables and setting their first values as determined by the programmer. Don't limit the concept to just writing variables. If calibration for a sensor was required, then the initialization phase of the program could check for existing calibration values or files, and if they did not exist, the program would request the operator to perform the calibration.

* For example, most gyros experience drift of a few degrees per minute. This would be the place to determine that drift and write that value to a variable.

In this first phase of the program, the robot physically does nothing. It sits still while the variable initialization runs its course.

After the initialization is done, the program starts parallel loops to monitor sensors and write the sensor values to variables. Some considerations:

* If data needs to be corrected this would be the place to apply the corrections. Continuing the gyro example from above, the gyro's data would be corrected probably by subtracting the drift value before the sensor reading would be written to the variable.
* Some sensors are noisy, so this would also be a place to smooth the data, perhaps by averaging the raw readings between loops, and then writing the smoothed data to the variable.

This enables the main program branch to act based on the variables, or as shown in the diagram, to Act(values) or Act, as a function of variable values.

It also gives some flexibility and modularity to the program. If a sensor is replaced, then only that sensor's loop will have to be changed, and it will be easier to find all the parts of the program that need changed.

If a new program action is created, then another action can be added to the main program branch, without risking the introduction of error by disturbing the sensor program elements providing the data.

Finally, if desired or important, data can be logged in parallel with the program - this allows the robot to do what the operator wants, while keeping the file management or data sharing complexities away from the main program structure. Again, this should reduce the opportunity for programming error introduction.

**Discussion:**

By running the sensors in their own loops, the robot simulates a *publish and subscribe* architecture. In other words, the sensor loops read and smooth the data and publish it to the variable. The main program subscribes to the variables and reads each of them when required. By executing the loops in parallel, the robot can do accomplish more because it's not limited to acting on only one sensor.

If you continue in more advanced robotics, it's likely you'll encounter machines using *finite state machines* to publish and subscribe data for action by the main CPU and core program.

Of course, the single CPU of LEGO Robots doesn't really work in parallel, but it's not a bad simulation.

**Pitfalls**:

* **CPU limitations.** Trying to do too much at one time can lead to the CPU granting priority to some program branches over others. This can sometimes lead to the robot waiting for a specific value to be reached, even as it careens into the obstacle it was supposed to avoid. In each of the sensor loops, consider adding a slight delay time. Even a "wait for a hundredth of a second" can limit the emphasis placed on that sensor loop without reducing its effectiveness.
* **Sensor response time**. Most LEGO and compatible sensors do not need to checked continuously throughout the program's run. For example, there's almost no value gained by checking a thermometer every hundredth of a second - use a wait block to check it less frequently, freeing the CPU to work other tasks. For another example, consider the NXT Ultrasonic sensor - given the limit of the speed of sound, the fastest it should be expected to respond is on the order of 15 milliseconds (or 15 thousandths of a second). Unless the robot is moving really fast, an Ultrasonic sensor probably only needs checked every half-second or so.
* **Synchronization challenges**. One potential challenge that could occur is that related variables could fall out of synchronization with each other. For example, imagine a line following robot having a turn variable giving it power for a turn in one direction, while the related light sensor variable indicated a turn in the opposite direction was more appropriate. This can usually be mitigated by comparing the delays in the sensor loops and ensuring the more critical sensors get checked more often.
* **Infinite loops**. I don't like infinite loops. In fact, I admit to having a deep aversion to them... Perhaps because *The Sorcerer's Apprentice* was the result of an infinite loop... c'mon kid, why did you forget to put an upper limit on how many trips the brooms could make with the water buckets? More seriously, how do we avoid them with this program structure? I use a boolean variable usually called **run done**, and set it to false in the initialization phase. Then as each sensor loop runs, it checks to see if **run done** is true, and ends the loop if that's the case. I then change the **run done** variable to true at the end of either the main program action branch or the data logging branch, depending on which is a higher priority. This allows the program to end gracefully instead of requiring a person to end it.

**Further reading:**

[Publish and Subscribe](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)

[Finite State Machines](https://en.wikipedia.org/wiki/Finite-state_machine)