*Event-Based Autonomous*

Our autonomous code takes advantage of the object-oriented aspects of programming available in Java, including abstraction and encapsulation. Each autonomous action that the robot performs, such as turning or pausing, is represented by an “object” in the code. Each of these objects is called an event, and contains all the unique information and code needed to complete the task. This event-based setup is similar to “event-driven programming” systems used by many professional programmers, and even the Java programming language itself.

*Advantages*

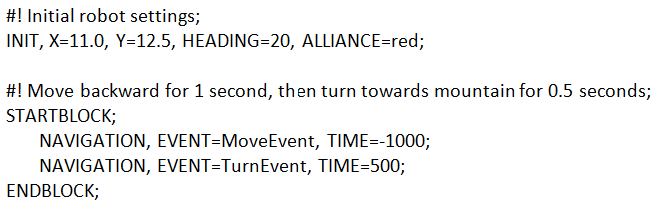
* Simplifies many parts of the code, as events contain (“encapsulate”) everything they need (except for a few details that are independent of the event, such as the initial robot position)
* Modifying an autonomous variation or adding a new one is easier, as each variation is just a different combination of the same events with different details (e.g. how long to pause)

*Disadvantages*

* Information used by multiple events must be stored in a separate location (e.g. the current position of the robot on the field)

*Autonomous Event Configurations*

Our autonomous event configuration system (or “event configs” for short) is a mini-programming language developed by us. It allows us to easily create, modify, and manage many different autonomous variations, with as little change to our main code as possible. Event configs, like the example below, are set up to work with how our event-driven system works. For more info, see the “Event Config” section under the “Misc. Documentation” tab.



*Advantages*

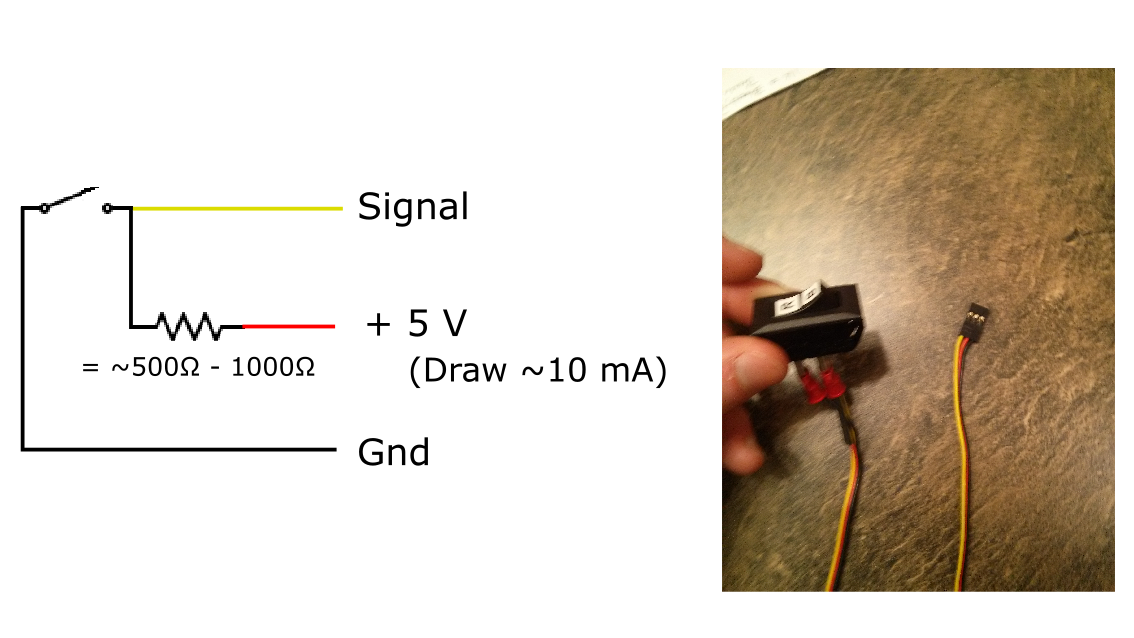
* Makes setting up, modifying, and managing many different autonomous variations significantly easier & quicker (only a few additional lines in the actual code for each event config)
* Minimal programming knowledge is needed to create an event config

*Disadvantages*

* A system to parse/translate the event config text into events in the code needed to be programmed
* A new type of event added to the navigation or action system will also require the addition of code to the event config parser/translator to be able to fully use the new event

*Custom Alliance Switch*

We created a switch that allows us to tell the robot which alliance we are on in a match. If we want to make a new configuration for autonomous, we do not have to create one version for the red alliance and one version for the blue alliance; this switch does that for us. The robot reads this switch only once and stores its state, just in case the switch is hit during autonomous. The parts of the code that vary for each side of the field read this stored value, and if necessary they reflect using equations we found (for more info, see the “Grid Reflection” section under the “Development” tab.) We had to wire this switch up to work correctly and safely with the robot’s device interface module.



*Advantages*

* Halves the number of autonomous variations we have to manage

*Disadvantages*

* If the drivers or coach forget to flip the switch to the right alliance, autonomous would likely fail, and the robot could interfere with our alliance partner’s program or get penalized

*Modular Design*

We utilized abstraction, encapsulation, cohesion and other object-oriented programming aspects to create a very modular program. Each “module” or component only handles a small portion of the entire program; instead it asks other components for information or to complete certain tasks. For example, we have a single “Drive System” class in the program that directly accesses the robot’s drive motors, and it provides a standard interface to the other components for accessing the motors (e.g. setting the left and right sides of the drive instead of each individual motor). So, if we added 200 new motors to each side of the drive, ONLY the drive system class would have to change to utilize the new motors.

*Advantages*

* Great opportunity for expansion
* Breaks down complex problems and systems into small, manageable components
* Allows for many different parts of the software to be developed simultaneously

*Disadvantages*

* Modular design is generally less efficient when it is run (for our team though this is not significant enough to be a problem)
* Modular design usually requires more code to set it up initially

*Separate Navigation and Action Systems*

We have two separate systems that handle events. The navigation system handles the events that move the robot to different places on the field ( such as moving forward or turning), and the actions system handles less common events such as pausing or using the arm system to score in a bucket.

*Advantages*

* Allows the robot to move around on the field while doing other actions; this allows for the limited autonomous time to be used more effectively

*Disadvantages*

* Since the two systems are running at the same time, there are only a few ways to synchronize events between the two systems (we implemented a way to group events together that helps alleviate this issue)

*A Few Other Features*

* A “debug hook” which is used to locate where in the program serious errors/”exceptions” in the code are occurring
* A safety feature of our code, where if an error/exception occurs while autonomous is running, the program basically stops running our other code. This helps prevent issues such as penalties or interfering with our alliance partner, and also aids us in fixing the bug (prevents a possibly confusing “cascade” of bugs caused by the same problematic code).