FRC Programming Tasks List

Noah Sutton-Smolin

September 3, 2013

Contents

This is a compilation of the introductory tasks for FRC programming. Each of these tasks are to be completed in order.

0.1 Task 00 - Initial lesson

Don't be lazy.
Do things right the first time.
This is the imperative of the programmer.
Don't be lazy.

1 Basic Usage

1.1 Task 01 - `DebugLog` usage

1.1.1 Goal

Learn the use of `DebugLog` to write outputs to the console

1.1.2 Details

One should never use `System.out.print*` to print console outputs. The reasons are varied, but primarily, the outputs become cluttered, unreadable, and generally unusable.

The benefit to `DebugLog` is that it enforces three things:

1. Log message severity, which can be filtered (which is useful for obvious reasons)
2. Log message timestamps, which do not normally come with system outs
3. Log message sources, which show where a message actually came from. This is critically important when diagnosing software issues.

The `RobotTemplate` class is actually what's run by the FRC software, and it interfaces with the `Global` class. You should never modify the `RobotTemplate` class.

Bear in mind, there is a critical difference between the `DEBUG` and `STREAM` levels: Debug is to be used for individual debugging messages. Stream is to be used for messages which would otherwise flood the console, but are useful for low-level debugs (like controller positions, or event triggering, etc.)

1.1.3 Task

Write one output of each severity level to the console when teleop starts, in order, using the `DebugLog` class.

1.1.4 Success

1. Each output is written to the console successfully.
2. No file has been modified except for the `Global` class.
3. Only the `initializeTeleop()` function in the `Global` class has been modified.
4. No `System.out.*` files are used.

1.2 Task 02 - Utilizing the `Global` class

1.2.1 Goal

Understand the function of the `Global` class with respect to shared resources and robot intialization.

1.2.2 Details

The `Global` class is used to hold all global resources and create initalization functions. These functions are wrapped to from RobotTemplate, which you are free to inspect.

The `static` modifier must be used before all `Global` variables; if you do not know what the static modifier does, you are free to do research into it. However, in short, the static modifier makes it so that only one instance of a variable can exist in a class. This means any class resource can be accessed using `[CLASSNAME].[RESOURCENAME]` instead of `[INSTANCENAME].[RESOURCENAME]`.

With respect to our given setup, if we had a jaguar (motor) in `Global` declared as: `public static final Jaguar jag1 = new Jaguar(1,2);` it can be accessed from anywhere in the code by using `Global.jag1`; however, if we declared it like so: `public final Jaguar jag1 = new Jaguar(1,2);` without the `static` modifier, then it could *not* be accessed using `Global.jag1`, which makes it an ineffective declaration for our purposes.

The Watchdog is not something you should ever personally need to deal with, as it is handled by the system, but it is important to understand. The Watchdog makes sure that the robot's code hasn't stopped or frozen. The act of "feeding" the watchdog ensures that the FRC system does not automatically kill our robot. The watchdog is called using:

```
Watchdog.getInstance().feed();
```

This gets the current instance of the Watchdog from the `Watchdog` class (through the static getInstance() function), and feeds it.

1.2.3 Task

Create a `Jaguar` in the global class. Change its power in each of the `initialize` functions. Be sure to call `Watchdog` somewhere once.

1.2.4 Success

1. The power was successfully set in each initialization function.
2. The `Watchdog` was fed once.
3. No file was modified except for `Global`.

1.3 Task 03 - Understanding function timings

1.3.1 Goal

Understand when the initialization functions are called in game.

1.3.2 Details

The `RobotTemplate` class controls the timings in the `Global` class. You need to understand when and under which circumstances the functions in the Global class would be invoked.

- `initializeRobot()` is invoked once when the robot's code starts up.
- `initializeDisabled()` is invoked once when the robot enters the disabled state before and after the game.
- `initializeAutonomous()` is invoked once when the robot's autonomous phase is about to begin.
- `initializeTeleop()` is invoked once when the robot's teleoperated phase is about to begin.

Every robot task in-game will be handled by the `EventManager`. The initialization functions are only used to set up `EventManager`'s sequencing so that everything else follows suit and makes sense.

- `robotKill()` is a predefined function which you can invoke when you wish to shut off the robot entirely.
- `robotStop()` is a function you can define which turns off the robot without disconnecting or disabling it.

1.3.3 Task

None

1.3.4 Success

You read this section, didn't you? Go read it again.

2 Sensor and Motor Usage

2.4 Task 04 - Using raw motors with `MotorLink`

2.4.1 Goal

Learn how to use the `MotorLink` class to more efficiently interface with the `Jaguars`

2.4.2 Details

The `MotorLink` class was written with a primary purpose of simplifying the logic beind encoders (which measure the angle of wheels) and speed controllers (which control the rate that motors move). As such, *all motors should be written using* `MotorLink`.

It is important to understand how to declare a `Jaguar`, though, for this task.

The motor you will be creating will take one argument: a `Jaguar`. You can declare the `Jaguar` in-line. Explore the `MotorLink` class and its functions. Bear in mind that it has been stripped down from its full form, as `Events` have not been introduced yet, and the `MotorSpeedControl` requires events.

2.4.3 Task

Declare a `MotorLink` as a static resource in `Global`. Set its powers scalar to a valid value. Set the motor power.

2.4.4 Success

1. The `MotorLink` is declared as a static resource.
2. The `Jaguar` is *not* declared as a static resource, and is instead consumed by `MotorLink`.
3. The power scalar and powers are set properly.
4. The code is error-free.

2.5 Task 05 - Using and declaring sensors

2.5.1 Goal

Declare sensors and use their values. subsubsectionDetails Sensors are primarily accessible through the `frc3128.HardwareLink` package. For instance, a gyroscope will use the `GyroLink` class (locaed in `frc3128.HardwareLink.GyroLink`) for communications.

`GyroLink` consumes a `Gyroscope` in its constructor.

2.5.2 Task

Declare a new `GyroLink` as a static `Global` resource. Print its value on teleop init.

2.5.3 Success

- The `GyroLink` is declared properly.
- The `Gyroscope` is consumed by the `GyroLink` constructor.
- The value of the gyro is printed out when the robot initializes.

2.6 Task 06 - Using raw encoders and angles

2.6.1 Goal

Create an encoder and read out its angle.

2.6.2 Details

None.

2.6.3 Task

Encoders are devices which keep track of the angle of a given motor. The `MotorLink` class can be linked with an `AbstractEncoder`.

Create a `MagneticPotEncoder` and print out its raw angle on teleop init.

2.6.4 Success

- Only the `MagneticPotEncoder` is kept as a static resource.
- The raw angle is printed out using `DebugLog` at the appropriate level.

2.7 Task 07 - Using motors in conjunction with encoders

2.7.1 Goal

Properly use raw encoders and their angles.

2.7.2 Details

`AbstractEncoder` is an abstract class which has two required functions: `getAngle()` and `getRawValue()`. There are two predefined implementations: `GyroEncoder` and `MagneticPotEncoder`.

There is a static function in `frc3128.Util.RobotMath` called `normalizeAngle()`; this function must be invoked on all angle values continuously, as it keeps the angle values to within normal range. We keep track of angles from 0 to 360 degrees.

2.7.3 Task

Create a new `MotorLink` using a `MagneticPotEncoder`. Read out the angle, and print the normalized value to the print stream.

2.7.4 Success

- Create a `MotorLink` and associated `AbstractEncoder` and `Jaguar`.
- The `AbstractEncoder` and `Jaguar` are fully consumed by the `MotorLink`.
- The angle from the `MotorLink` is normalized and printed out.
- The print out uses `DebugLog`.

2.8 Task 08 - Using the `PneumaticsManager`

2.8.1 Goal

Learn the uses of `PneumaticsManager`.

2.8.2 Details

`PneumaticsManager` is a class designed to control the pneumatic air compression systems on the robot. The pneumatics system uses compressors and two-sided solenoids called Festos to control air flow.

The `PneumaticsManager` must be declared with a compressor. When a new piston is created, it returns a `PistonID`, which is a reference value used by the `PneumaticsManager` to handle piston movements.

2.8.3 Task

Initialize the `PneumaticsManager`. Turn the compressor off, then on. Create a piston, and record the `PistonID` as a static global resource. Turn the piston forward, then flip its state, then reverse its polarity.

2.8.4 Success

All tasks completed without error.

3 Event Manager

3.9 Task 09 - Single run events

3.9.1 Goal

Learn how and when certain events are executed, and how to trigger single-run events.

3.9.2 Details

The `EventManager` is a rather simple block of code which drives the entire robot. It takes `Events`, or small blocks of code, and runs them in the sequence they were queued.

An `Event` is an abstract class. If you don't know what abstract classes are, it is strongly suggested that you look up the documentation for them. An Event must be declared as a class. That means, to create an event, you must do the following:

```
class EventImpl extends Event {
public void execute() {
//your code
}
}
```

If you do not have the `execute()` function, you will not be adhering to the `Event` abstract class, and NetBeans will barf errors in your face.

You can also declare an `Event` inline, but it *must* have an implementation.

```
Event e = new Event() {
public void execute() {
//your code
}
};
```

You can also declare an `Event` inline, and directly queue it:

```
(new Event() {
public void execute() {
//your code
}
}).registerIterableEvent();
```

The `Event` has several critical functions:

- `registerSingleEvent()` will insert the `Event` into the `EventManager`'s queue, and run it once before removing it.
- `registerIterableEvent()` will insert the `Event` into the `EventManager`'s queue, and it won't be removed until it is cancelled directly (or the entire `Event` stack is dropped).
- `registerTimedEvent(int msec)` will create a new instance of a `TimerEvent` and register it as an iterable event. When the timer expires, the selected `Event` will be run. If you're going to call this function, it is *highly* recommended that you call `prepareTimer()` first.

That's it for starting events. For cancelling events:

- `cancelEvent()` will cancel the running event.
- `cancelTimedEvent()` will cancel the event's execution timer.
- `cancelEventAfter(int msec)` will cancel the event after a given amount of time.

The `EventManager` is called automatically in `RobotTemplate`. Do not modfiy `RobotTemplate`. Do not manually invoke the `EventManager` de-queueing function.

3.9.3 Task

Create a single run event which will print out some text.

3.9.4 Success

1. The text is printed out using `DebugLog` and an `Event`.
2. No resources are kept longer than they are needed.