FRC Team 3128 Software Framework Summary v0.2

Noah Sutton-Smolin

September 28, 2013

Contents

1 Introduction

Welcome to the FRC Team 3128's software framework! We are making this available for public use by any FIRST team (attribution requested), as we have fleshed out the details and are decently pleased with the results. May it help you on your further quests.

This framework is designed to simplify the conceptualization and execution of robotic code, specifically designed for the FIRST Robotics Competition API. It is an event-driven system; this means that the robot is controlled solely by `Events`. An `Event` is simply a block of code which runs at a set interval.

This document assumes confident Java and Object Oriented Programming knowledge, including abstract classes and their implementations. It also assumes you have some familiarity with the FRC API. Additionally, you may wish to check out our code from Google Code (titled "frc-team-3128"). You must be using *the latest FRC Java API*. These things will not be explained over the course of the document, as this is neither a Java nor Netbeans nor FRC API introduction; rather, it is an FRC 3128 architecture introduction. Additionally, the entire program has javadoc, so all functions are explained in themselves.

While this code architecture does take some getting used to, it has increased our speed by over a factor of 100, and we hope it does the same for you. As the only setup is done during initialization, it also allows for dynamic and easy testing; again, we hope it does the same for you. Here's a short list detailing some capacity of this architecture:

- Sequence together tasks during autonomous in a seamless way, which would otherwise be difficult to attain
- Control the entire drive train during teleop
- Dynamically add and remove buttons and their effects
- Allow dynamic decision-making trees and heuristic algorithms

Should you find any errors, questions, concerns, bugs, questions, etc., please email Noah Sutton-Smolin at `noahsutsmo@gmail.com`. If so, we apologize for the inconvenience, and will be more than happy to update and/or fix. We will also be happy to help answer questions about how to use this API.

If there are any features you would like to see, please send me an email as well!

1.1 Why use an Event-driven system?

Event systems are particularly useful in robotic applications, as they enable responsive behavior which remains decentralized. As opposed to many other styles, which involve writing specifically what you want to do in order, the `Event`-driven system allows you to dynamically change the order and timings of events. In addition, it allows for several tasks which would otherwise be inordinately difficult: the creation of autonomous programs (through the use of `EventSequencer`), the creation of events which run after a certain amount of time, and high efficiency increases.

One of the primary benefits belongs with controllers (called "Joysticks" in the FRC API; we will call them "controllers" for the duration of this document). Controllers are also `Events`; however, instead of controlling the drive train directly, they call events through the `ListenerManager` utility. The `ListenerManager` allows certain events to be run when the triggers are called.

Here's an example:

```
ListenerManager.addListener(new Event() {
    public void execute() {Global.motorSpin.setPower(0.7);}
}, "buttonADown");
```

In this statement, we've done two things: we've first created a new `Event`. That event simply contains code which spins up a motor. We've then registered it with the `ListenerManager` under the listener `"buttonADown"`. Now, if the listener `"buttonADown"` is called elsewhere in the code, all associated `Events` will be run. This is as simple as:

```
ListenerManager.callListener("buttonADown");
```

In this case, when the user presses the `A` button, the Controller event will register the change and call the appropriate listener, `"buttonADown"`. The `ListenerManager` then looks for any events which should run (are linked with the key), and executes them.

2 Core program structure

This section will detail where the code you write actually goes, then will detail the actual package structure for the program.

The first thing to cover is the `DebugLog`, as it prettifies all of the debug outputs. It makes everything organized and neat in your outputs. It does this by enforcing three things:

1. Log message severity, which can be filtered, as we do not always want to see the output.
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 severity levels in order are:

1. `DebugLog.LVL_ERROR`: This is used when the program has encountered an error which will cause it to exit.
2. `DebugLog.LVL_SEVERE`: This is used when a part of the program cannot continue as requested.
3. `DebugLog.LVL_WARN`: This is used when a part of the program may not function as intended, or something has the possibilty to fail.
4. `DebugLog.LVL_INFO`: This is used for important operating information *only*.
5. `DebugLog.LVL_DEBUG`: This is used for debugging outputs which are temporary and/or infrequent.
6. `DebugLog.LVL_STREAM`: This is used for constant outputs. For instance, controller positions are under the stream level. The difference between `STREAM` and `DEBUG` is critical; don't mix them up.

2.1 Code execution

The `RobotStartup` class is what the API actually runs. The robot is based of FRC's `IterativeRobot`. This class will link directly with the Global class, and is actually what calls the `EventManager`'s `processEvents()` function. You should never need to modify this file, as it serves only to link the default FRC API to the rest of the code.

The file that you *will* need to modify is `Global`. The `Global` class actually controls the initialization and setup of the robot. Put all global resources in this file as `public static [final]` values. As this system is `Event`-driven, you won't need to actually execute any code yourself after initialization. Instead, put all `Event`s you'd like to run in this class under the appropriate `initialize*` function.

Let's say I wanted to create a simple drive program. Here's how we would do this most effectively:

```
public static final XControl xControl1 = new XControl(1);
public static final MotorLink mLeft = new MotorLink(new Jaguar(1,1));
public static final MotorLink mRight = new MotorLink(new Jaguar(1,2));

public static void initializeTeleop() {
    ListenerManager.addListener(new Event() {
        public void execute() {
            Global.mLeft.setPower(Global.xControl1.y1+Global.xControl1.x1);
            Global.mLeft.setPower(Global.xControl1.y1-Global.xControl1.x1);
        }
    }, ListenerConst.UPDATE_JOY1);
}
```

(Bear in mind, all foreign classes will be explained later. For now, all you need to know is that `XControl` is an XBox 360 Controller, `MotorLink` is a better way to set motor powers, and `ListenerConst` is a class which holds the listener strings so you don't have to type them every time.)

So, what does this code do? First, it creates the `mLeft, mRight, xControl1` global resources. These can be used everywhere simply by specifying `Global.[varname]` Then, in `initializeTeleop()`, the program adds an `Event` to the `ListenerManager` which will be run when joystick 1 on the controller changes. This way, the function doesn't run every iteration, but only runs when it needs to.

2.2 Package structure

The following is the full package structure for our architecture:

```
\---frc3128
    +---EventManager : Used for Event control
    |   \---EventSequence : Used to sequence events together, as in autonomous
    +---HardwareLink
    |   +---Controller : Used for the various allowed controllers
    |   +---Encoder : Used for encoder declarations
    |   +---Motor : Used for motor declarations (Jaguar not recommended)
    |   |   \---SpeedControl : Used for motor speed controllers (incomplete)
    |   \---Pneumatics : Used for pneumatic control
    \---Util : Used for generic utilities, such as DebugLog
        \---Connection : Used for external connections (incomplete)
```

3 What are Events, and how do I use them effectively?

`Event`s are blocks of code which can be run dynamically. The code for an event goes inside the event's abstract `public void execute()` function. Events have three modes: single run, iterable, and timed.

- A single run `Event` will be run once, then removed from the `EventManager`'s queue.
- An iterable `Event` will run continuously until it is stopped. There are three things which will stop an `Event`:
  1. It causes itself to halt.
  2. It is caused to halt by an outside command.
  3. It throws an exception/error, which will not be handled by `EventManager`.
- A timed `Event` will be run after a certain amount of time. These work in a particular way; when you call a timed event, it creates an iterable `TimerEvent`. The `TimerEvent` will run until the time has expired, then delete itself and trigger the associated `Event`. In this way, timed events are really just normal iterable events with a timed trigger.

3.1 Small blocks of code

`Event`s can be used to handle small blocks of code or small, linear instructions. This is primarily useful when you're dealing with buttons, but has other uses. This also works well in conjunction with `ListenerManager`. One **always declares and names these events based on their *effects***, not their triggers. For instance, you wouldn't name an event `ButtonADownEvent`, but rather `MotorSpinUp` and add it to the trigger for `"buttonADown"`.

```
class MotorSpinUp extends Event {
        public void execute() {Global.motorSpin.setPower(0.7);}
}

class MotorSpinDown extends Event {
        public void execute() {Global.motorSpin.setPower(-0.2);}
}
```

Then, elsewhere:

```
ListenerManager.addListener(new MotorSpinUp(), "buttonADown");
ListenerManager.addListener(new MotorSpinDown(), "buttonAUp");
```

3.2 Drive events

Drive events work in much the same way as above, but they contain more code, and are on broader triggers. For instance, you'd put a drive event on the joystick update event, not a button event. This should be fairly straightforward, given the previous event description.

3.3 Controller events

Controller events are much more complex. They are one of the few iterable events you'll ever run in your program. The constructor typically inserts the event into the `EventManager`. This event needs to keep track of the state of the controller, and call appropriate listeners when there is a change. For instance, our `AttackControl` event is defined as follows:

```
public class AttackControl extends Event {
    public  double    x, y, throttle;
    public  Joystick  aControl;
    private final int controlID;
    private boolean[] buttonsPressed = {false, false, false, false, false,
        false, false, false, false, false, false};

    public AttackControl(int port) {
        aControl = new Joystick(port);
        controlID = port;
        this.registerIterableEvent();
        DebugLog.log(DebugLog.LVL_DEBUG, this, "AttackControl added self to
            event manager!");
    }

    public void execute() {
        boolean updateJoy = false, updateThrottle = false;

        if(x != aControl.getAxis(Joystick.AxisType.kX)) updateJoy = true;
        if(y != aControl.getAxis(Joystick.AxisType.kY)) updateJoy = true;
        if(throttle != aControl.getAxis(Joystick.AxisType.kThrottle))
            updateThrottle = true;

        x = aControl.getAxis(Joystick.AxisType.kX);
        y = aControl.getAxis(Joystick.AxisType.kY);
        throttle = aControl.getAxis(Joystick.AxisType.kThrottle);

        if(updateJoy)
            ListenerManager.callListener(ListenerConst.UPDATE_ATK_JOY);
        if(updateThrottle)
            ListenerManager.callListener(ListenerConst.UPDATE_ATK_THROTTLE);
        if(updateJoy || updateThrottle)
            ListenerManager.callListener(ListenerConst.UPDATE_DRIVE);

        for(int i = 1; i < 11; i++) {
            if(buttonsPressed[i] != aControl.getRawButton(i)) {
                ListenerManager.callListener(ListenerConst.getAtkCtrlListenerKey(
                    this.controlID, i, aControl.getRawButton(i)));
                DebugLog.log(DebugLog.LVL_STREAM, this, "Button " +
                    (this.controlID + "-" + i) +
                    (aControl.getRawButton(i)==true?" pressed.":" released."));
            }
            buttonsPressed[i] = aControl.getRawButton(i);
        }
    }
}
```

As can be seen, the `AttackControl` event will trigger listeners whenever something in the controlelr state changes. For instance, if the throttle updates, it invokes `ListenerConst.UPDATE_ATK_THROTTLE`. This is an example; you are free to add your own controller types as well as modify ours.

3.4 Sequencing events in autonomous

The `EventSequencer` is the primary class used during autonomous. It allows for sequencing of as many `SequenceEvents` as you'd like.

A `SequenceEvent` is almost the same as an `Event`, wtih two major differences: First, it has an ad-

ditional abstract function called `exitConditionMet()` to indicate whether the current task has finished. Additionally, it keeps track of *how long* it's been running, which allows for timings and such. There are two default variants of the `SequenceEvent`: the `SingleSequence` which runs once and exits, and the `TimedSequence` which runs for a certain amount of time then exits. The default `SequenceEvent` can be used for custom exit conditions, such as a tilt motor angled properly, or a camera in line, or some such.

Let's look at an example autonomous program:

```
public static void initializeAuto() {
    EventSequencer autoSeq = new EventSequencer();

    autoSeq.addEvent(new TimedSequence(1000) {
        public void execute() {
            Global.mLeft.setSpeed(0.7);
            Global.mRight.setSpeed(0.7);
        }
    });

    autoSeq.addEvent(new SequenceEvent() {
        public void execute() {
            Global.mLeft.setSpeed(-1);
            Global.mRight.setSpeed(0);
        }

        public boolean exitConditionMet() {
            return !(Global.mLeft.getEncoderAngle() < 50);
        }
    });

    autoSeq.addEvent(new SingleSequence() {
        public void execute() {
            Global.mLeft.setSpeed(0);
            Global.mRight.setSpeed(0);
        }
    });

    autoSeq.startSequence();
}
```

So, what does this do? It should be rather clear from the way it's written: When the autonomous program initializes, the it adds a timed event that moves the robot forward for one second. Then, it adds a `SequenceEvent` which turns the robot counterclockwise until `mLeft`'s encoder angle is less than 50 degrees. Then, it adds a single event which stops the robot from moving. Then, `startSequence()` is called to insert `autoSeq` into the `EventManager`'s stack. These events will be executed in order once the `EventManager` runs at the start of autonomous.

4 Hardware interface classes

The FIRST API has some organizational issues which do not fit well with the event-driven system. As a result, we've created a `*Link` class (e.g. `MotorLink, GyroLink`) for each piece of hardware we use. This not only allows us to customize the way hardware works, but also allows us to change the hardware interface without changing code function should their API change.

All of the harware interface classes take their respective FRC API items as arguments. Why did we do this? Because otherwise it becomes atrociously unclear what you're sending to the class. For instance:

```
new MotorLink(1, 2, 1, 5, 2, 3, new LinearAngleTarget(0.1, 0.1));
```

is vastly less clear than:

```
new MotorLink(new Jaguar(1, 2), new MagneticPotEncoder(1, 5, 2, 3), new
    LinearAngleTarget(0.1, 0.1));
```

4.1 Motor encoders

The encoders and speed controllers feed into the `MotorLink` class, so we will discuss those first. The `AbstractEncoder` class effectively creates a way for a `MotorLink` to have an associated encoder, regardless of its type. For instance, we want the motor to handle the optical encoder as well as the magnetic potentiometer in the same way - internally, at least.

All encoders must have two functions: `getAngle()` and `getRawValue()`. This class is abstract instead of an interface as abstract classes, for an unknown reason, are *much* faster on the cRIO than interfaces. To create an encoder, simply insert `extends AbstractEncoder`.

4.2 Motor speed controllers

The speed controllers are similar to encoders, but are significantly more complex. They work off individual time steps. They are events so as to easily facilitate this. There are four abstract functions: `setControlTarget(), speedTimestep(), clearControlRun(), isComplete()`.

- `setControlTarget()` accepts the value to be controlled to, whether it is a speed, position, or other. The class accepts this value, and does whatever it needs to do with it.
- `clearControlRun()` resets the speed controller after a control run is complete.
- `isComplete()` indicates whether the speed controller's goal is complete.
- `speedTimestep()` computes the power for the associated `MotorLink` class. It returns a `double`, which becomes the new power of the motor.

None of the default speed controllers are tested or complete, and this package is still in testing. Use it with some degree of caution.

4.3 MotorLink

The motor itself is controlled by the `MotorLink` class. The motor can be associated both with a speed controller and encoder, but can also be set normally. There are a few important behaviors and features for `MotorLink`. Bear in mind that running erroneous actions (such as deleting a nonexistent speed controller, replacing an active speed controller, setting the power while a motor is under speed control, etc.) will generally **not** stop the `MotorLink`.

- Encoders and speed controllers can be added, changed, and set at any time. Changing an active speed controller produces a severe level debug message (see DebugLog). Changing an encoder at all will produce a warning.
- The `reverseMotor()` function will change the direction of a motor. This is important, as positive powers should generally always mean clockwise.
- The power scalar on the motor adjusts what the maximum bound for power output is, where $0 < x < 1$.
- Trying to set the motor's power while a speed controller is active will cause the speed controller to halt.

This class is largely untested as of 9/28/13, so there may still be errors.

4.4 Pneumatics

The `PneumaticsLink` controls the pneumatic systems on the robot. The `PistonID` class keeps track of which piston you're handling. All it holds is an index, though it can also be used to reverse the polarity of the piston (e.g. change out to in, in to out). To set a piston, simply invoke the correct `PneumaticsLink` function on the correct `PistonID`. The `PneumaticsLink` currently only supports dual solenoid valves, such as Festos.

The functions will not be detailed here, as they are rather self-explanatory.

5 Other utilities

`DebugLog` was covered at the start of this document. The `Constants` class is effectively a pile of settings and flags. The comments in the constants file will explain what they are for. `RobotMath` contains a couple basic robot math functions, and any others you feel should be added.

6 Conclusion

Hopefully this document helps introduce you to the FRC Team 3128 even-driven control system! And, additionally, we hope this helps you during competition. The programming intro document contains information on setting up NetBeans and the FRC API. The programming tasks document contains a tutorial for newcomers on how to actually write code in this API.

See you at the field!