Introduction to Using the StarL Framework

The StarL framework is divided into several Java packages based on the functionality offered:

Comms – contains all communication handling classes and Wi-Fi interfaces
Functions – contains built-in distributed algorithms such as leader election and mutual exclusion
Harness – contains simulator specific classes that aren't used in a real implementation. These classes are automatically selected for use when simulating an application.
Interfaces – contains interfaces and abstract classes used throughout the framework
Motion – handles Bluetooth connectivity and robot motion
GVH – contains the classes used by the Global Variable Holder, the core component of StarL

All StarL applications contain a core GlobalVarHolder (GVH) object. The GVH gives all threads access to communications, motion, robot identification, and debugging output. The main StarL application uses LogicThread as its base class. As seen in the simulator's template application TemplateApp, the main GVH is passed into the constructor for StarL applications, which is then passed to the superclass constructor. A protected reference to the GVH is kept by LogicThread and is accessible by TemplateApp.

The provided CommsTestApp.java is a good place to start to understand how to use the built in synchronization and leader election functions. StarL applications have a central state machine which uses an enum of type STAGE called stage to track the current state. In CommsTestApp, a BarrierSynchronizer and RandomLeaderElection are created. BarrierSynchronizer is used to align all agents to the same "barrier", a common point in the execution of an application. Once all agents have reported that they have arrived at this barrier, the application starts the leader election and continues to the state ELECT. Here, the main application waits until a leader has been elected by checking the getLeader() method of RandomLeaderElection. Once a valid leader has been obtained, the application stores the leader's name in the results String array and completes. In simulated applications, the contents of the results array are printed when each robot completes, which lets you easily see what happened in execution.

RaceApp.java is an example application which uses motion and message passing. In RaceApp, the robots are given an ordered list of waypoints and try to travel to them in order. When a robot reaches their destination waypoint, it sends a broadcast message to all other robots informing them that it was the winner. The robots then switch to the next waypoint in the list and race to be the first one there. It might make more sense when you run it.

Message handling in StarL is implemented using the observer pattern. To receive messages, your application will need to implement the MessageListener interface. This includes the messageReceived(RobotMessage m) method. All StarL messages are passed via RobotMessage objects, which have a sender, a recipient, an ID number, and contents. The ID number can be thought of as a mailbox number, if your application is registered to receive messages with ID 50, the framework will call messageReceived in your application whenever a message with ID 50 is received.

To inform the framework that your application is listening for messages, call gvh.comms.addMsgListener(ID, MessageListener) in the constructor of your application to register as a message recipient for a particular message ID. This happens on line 33 in RaceApp.java. A MessageListener object can be registered for multiple message IDs.

A similar system exists to monitor robot events, such as robot motion or position changes. Classes extending RobotEventListener can be registered with the GVH to have their robotEvent() method called whenever a class calls GVH.sendRobotEvent. Robot events have an associated integer “type” and “event” which contain event information. Built in events types are declared in the Common class. All robot events have a type, but not all use the “event” field.

The Common class (in the object package) contains all static definitions used for message IDs and events, as well as a collection of useful simple functions. I recommend looking through them, all have Javadoc comments.