Below are the UML Class diagrams I came up with and some short descriptions of how I see everything working. In case you forgot the 3 permission symbols are "-" which is private, "#" which is protected, and "+" which is public



ChronoTime Class:

* Attributes
  + powerState: this just keeps track of whether or not the system is on
  + currentRun: this is the active run
  + pastRuns: this is some form of collection of previous runs, though it may not be necessary upon rereading the project description
  + channels: this will be an array of potentially up to 8 sensor object
  + log: this will be a text log of every command received by the system
  + currentCommand: I will talk about this below
* Methods
  + setCommand(string Command) this is how we are actually going to take input from the simulator, they will pass the ChronoTimer a string which will be parsed in the execute method then depending on the result of the parsing one of several methods will be called. we may have to implement some form of ready for command signal that the simulator people can use depending on implementation as well
  + execute(): this will be essentially our listener loop, if the system is on it will be checking for new commands and/or sensor events being triggered, parse the commands or react to the triggering events then reset back to a waiting state, if the system is "off" the only command we will accept is power
  + power(): simply toggles the powerState boolean
  + reset(): resets all variables to null/init states
  + conn(sensor, int): will take a sensor object and an int and store that sensor in that location, one of the 3 public methods the simulation people have to interface with the system
  + disc(int): removes the sensor object from the indicated position
  + print(run): will take a run object and print to the console the results of it, most of the work on this one will be done in the Run class
  + export(run): will take a run object and print to some file the results.
  + newRun(): will generate a new run object to be stored in the currentRun, must return exception if currentRun already is not null
  + endRun(): will close out the currentRun and move it to the pastRuns collection, leaving currentRun null in preparation for the next time newRun is called



Run Class:

* Attributes
  + type: this will just be a simple storage of what type it is
  + systemTime: this is where the system time will be stored, this may move up to the ChronoTimer class
* Methods (these are all abstract methods to be implemented by the specific types of runs, except for maybe the set and get time functions)
  + trig(int): will take an int that represents which sensor triggered and act accordingly depending on the type of run and current situation
  + swap(): this will swap two racers
  + dnf(): this will trigger that a certain racer did not finish
  + cancel(): this will void the latest start input and reset the affected racers
  + num(int): this will create a new racer object
  + clr(int): this will delete a racer object that exists
  + end(): this will be used to close out a run before it is moved to the pastRuns collection
  + setTime(time): sets system time
  + printOrderByBib(): optional print method to adjust display order
  + printOrderByTime(): optional print method to adjust display order
  + printOrderByStart(): optional print method to adjust display order
  + getTime(): returns system time



IND Run Class (inherits Run class):

* Attributes
  + queuedRacers: racers that have been created but have not started
  + runningRacers: racers that are currently running
  + finishedRacers: racers that have finished, or had DNF triggered on them
* Methods
  + SEE RUN CLASS



Racer Class

* Attributes
  + bibNum: unique int used to keep track of racers
  + startTime: time racer starts
  + endTime: time racer finished
  + dnf: whether or not a racer dnf'ed
* Methods
  + Racer: Constructor method with several different formats depending on situation
  + Setters and Getters for every attribute
  + getTime(): returns a time that is the finishTime-startTime, can only be called if both exist



Sensor Class

This class will function as an interface between the sim and the system.

* Attributes
  + type: stores the type of sensor
  + isTriggered: stored whether or not the sensor has been triggered
* Methods
  + Sensor(): constructor, should always be given a sensor type
  + Trigger(): Simulation team calls this to set triggered to true
  + Reset(): System team calls this to acknowledge the trigger was received
  + getStatus(): returns isTriggered

Sim team, when creating Sensors, a reference should be stored so they can be accessed by you later.

Most of the commands listed on the project description will be handled by the system, with the exception of the TRIG command, and the EXIT command. The CONN and DISC commands will have to be handled by both, as both systems should be keeping track of the connected sensors.