Introduction

The lithium-ion battery test fixture is a distributed system which collects data on up to 20 batteries simultaneously. The main computer running these tests is known as the PC side of the system. The PC side of the system is being designed within the Processing.org programming language. Processing.org is a Java based language with numerous libraries which act as a front end to graphical interfaces. Since it is based on Java, most Java objects/primitives still work in the new environment. The PC side has a USB link to each Arduino (a small embedded system), which runs tests on the batteries in question. The Arduinos are required to send information to the PC host one time a second due to the small amount of memory present on each Arduino. With the help of extra hardware, each Arduino going to be capable of running different tests on each of the two connected batteries. These tests may take 3 to 6 months to finish so stability of the entire system is of great importance. Below is a composite structure diagram of the system.



Figure 1: Composite Structure Diagram for the system. (Note: There is a lot more going on in the hardware than shown, but it is out of scope for my share of the project.)

The value of N is currently set at 10 Arduino connections, but the Java front end is being designed to handle much more for scalability. What will actually determine the value of N will most likely be limitations of the RXTX driver which was used as a means of retrieving data off the serial port. The driver has proven not to be 100% stable (after all, it is an alpha build), but hopefully the next stable version of the driver will fix the potential issues.

Demonstration

# Sequence Diagram

This section describes the software systems implementation, and how they interact with each other. Below is a sequence diagram which helps to explain the PC to Arduino interactions (Figure 2). Basically, once the connection is established and a test is initiated, the system sends predefined program lists to the Arduino, one step at a time. While the Arduino is processing the step, the PC sets the Arduino’s state to busy, and waits for the Arduino to return an idle command in the heartbeat before sending the next step. This is the end goal, but it has not reached this point yet. For example, the resetMillies() command has not need implemented or tested yet. It may interfere with background tasks on the Arduino.

****

Figure2: Sequence diagram from the PC side’s perspective of operations.

# Prototype GUI

Below is a prototype of the GUI. The GUI was designed using Processing.org primitive shapes and elements from the ControlP5 GUI building library. There are a few bugs with it, but for now it is good enough for alpha testing.

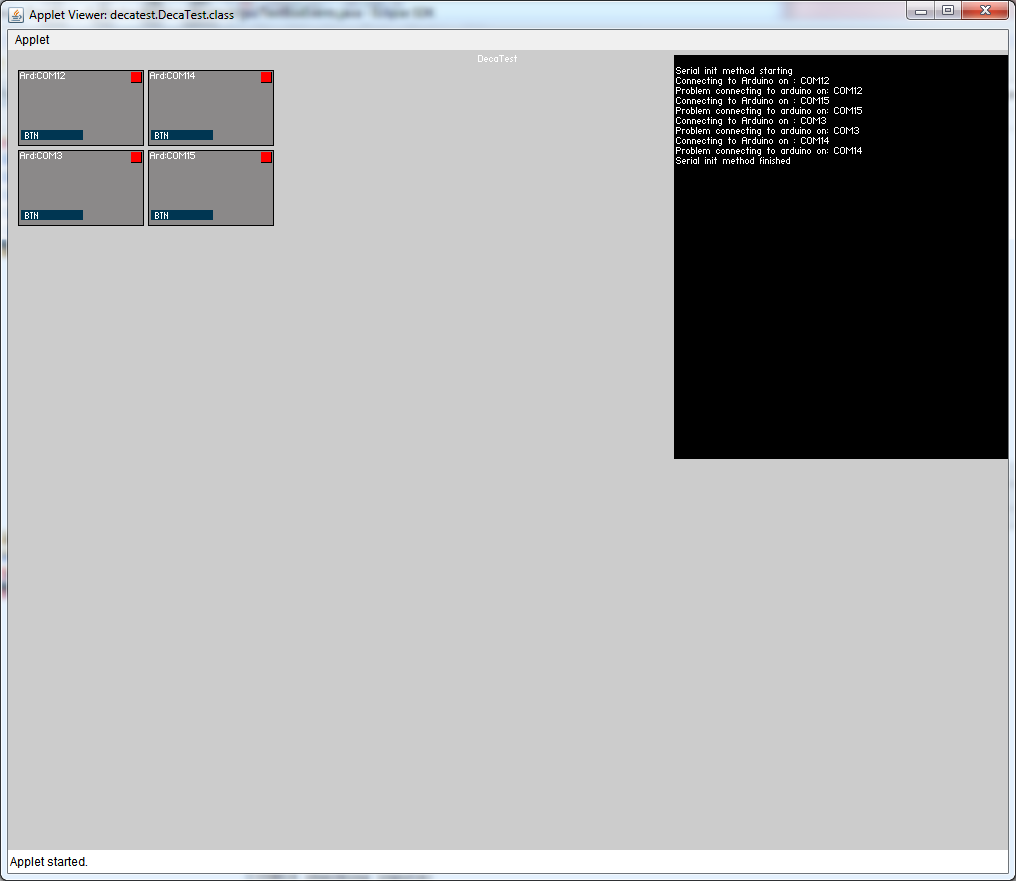


Figure 3: Prototype GUI

Each of the darker grey boxes represents a predefined Arduino unit in the code. The box in the top right corner of each dark grey box represents the connection status for that com port. Since no Arduinos were connected to this computer when the screenshot was taken, all of the lights are red, or in other words, in the “not connected” state. If the light is yellow, that means it is currently trying to connect to that Arduino. If it is green, that means the serial drivers think it has connected to a serial device. A limitation of this is that Bluetooth devices will always connect. The button belonging to each dark grey box has previously been used for testing purposes, but will eventually be used to input a scheduler file. More buttons can be added at anytime. The black box contains a text area object which is used as a means of outputting status messages to the user. Overall the GUI is far from complete and it is recommended to reengineer it with a more suitable Java based tool.

# Class Diagrams

Below is a class diagram of the serial driver front end to the RXTX dll.

Figure 4: Serial Driver class diagram (Note: Yellow boxes represent re-used libraries/classes which I did not write)

|  |  |
| --- | --- |
| Class | Description |
| WatchDogPacket | This class contains the data to pass up the callback latter |
| DacPacketEvent | This is an interface which is implemented in the main application code. The DacConnection class sends new packets to wherever this interface is implemented. |
| Serial\_Event | The serial\_event interface is inherited by the class which will receive notifications of when the lowest layer's buffer finds an end of transmit byte array within the byte buffer. |
| SerialDriver | The SerialDriver class manages the incoming/outgoing data via the RXTX DLL. It reads data from the incoming buffer and stores it locally in a byte array until we've received an entire packet's worth of data. It also contains methods to reconnect to an Arduino after it is disconnected from the PC. |
| DacConnection | The DacConnection's largest method defined as abstract in its super class, and it gets called each time the buffer contains a full packet. It passes in the location of the last byte and creates a DacPacket out of the incoming data. Once completed, it gets forwarded to the next highest level. |
| NotConnectedException | This is simply an exception to throw when the user tries to send data to an Arduino which is not connected to the PC. |

Below is a class diagram of the main application code.

Figure 5: Main program structure (Note: Yellow boxes represent re-used libraries/classes which I did not write)

|  |  |
| --- | --- |
| Class | Description |
| DecaTest | This class contains the program's main method and is responsible for all initialization. It also routes all button presses to the appropriate ArdUnit. |
| ArdUnit | One of these classes is instantiated for each defined Arduino. It contains the highest level of the serial driver callbacks and also handles all button events belonging to its assigned Arduino. |
| StatusLight | One StatusLight class is defined for each Arduino. It contains a thread which periodically checks for the state of the Arduino and updates the GUI's StatusLight according to the current state. |
| WatchDogPacket | The WatchDogPacket class is a data class which is created by the serial drivers to return all data to the main program. It also translates the data to an easier to read format. |
| DacPacketEvent | The DacPacketEvent is an interface. The class that extends it will be able to receive WatchDogPackets from the lower layers. |
| FileIO | The FileIO class reads and writes data from all associated files. It is responsible for writing the data to the correct folder and keeping track of how far along we are in the current test. It also contains the code to parse program and schedule files. |
| TextBoxEvents | Contains all static methods to write data to the black text box on the screen. |

# Threads

The Java/Processing program contains several threads of execution.

* The GUI and the all events generated by the objects on the screen (ex: button click events) are processed in one thread of execution. The event handling sometimes locks up the screen briefly, so a possible refactor would be to create an event queue for the events to enter, and then have another thread dequeue the events and process it. Since this is only a minor annoyance, this refactor will not be completed this quarter.
* When the program is initializing, a thread is spawned in the background to connect to each of the Arduinos, one at a time. This thread stops once all Arduino serial objects are initialized and at least one attempt to connect each has occurred.
* The previous thread creates a thread for each of the com ports the program tries to connect to. Currently, these threads wakeup once every second to check if the serial driver reports that a connection is established. If the connection status changes, it updates the color of the light on the screen. A possible refactor would be to combine all these threads into one since the code in each of them is extremely minimal. This may not be completed this quarter due to time constraints.
* Once the serial connection is initialized, the RXTX driver spawns a thread for each connection. This thread uses a callback method defined in the SerialPortEventListener interface to inform the class that new data has arrived (or of any other type of events you’ve added listeners to). Once this occurs, the SerialDriver class stores the data in a linked list of bytes. If this class detects an end of packet transmission, it notifies the DacConnection class through the Serial\_Event interface that a complete packet is waiting. It then reads the data out of the buffer and forms a DacPacket object out of the information. Finally, it returns this packet to the ArdUnit class though the DacPacketEvent interface. This is all done in the thread spawned by the dll.

# Flow of Events at start-up

//TODO: this