SCADA README
Andrew Jameson
5/13/2009

The SCADA (Supervisory Control and Data Acquisition) software component has several responsibilities. It records information from the sensors, checks the values, and records them in a database. Based on this information, it maintains the system state, which is described elsewhere in this document. The SCADA system also displays the current values on a webpage, found at URL lprds.aec.lafayette.edu as of May 15th, 2009, along with any events and faults that the system generates. Additionally, the website allows a mechanism for the user to control the system. Finally, the software controls the small LCD display as well as the display board to inform users of the state of the system.

## Required Software Packages

The chosen operating system was Ubuntu distribution of Linux because it is popular and well supported. The following software packages must be installed for the SCADA system to function:

1) MYSQL (found at http://www.mysql.com/)
2) PHP (found at ww.php.net)
3) GNU compiler g++ (found at http://gcc.gnu.org/)
4) Apache2 webserver (found at http://www.apache.org/)
5) gnuplot (found at http://www.gnuplot.info/)
6) cppunit (found at http://sourceforge.net/projects/cppunit/)
7) mysql++ (found at http://tangentsoft.net/mysql++/)
8) phpmyadmin (found at http://www.phpmyadmin.net/home_page/index.php)
9) LCD4Linux Version 0.11.0-SVN (found at http://ssl.bulix.org/projects/lcd4linux/wiki/Download)
10) SimpleTest (http://www.simpletest.org/)
11) libusb-dev – For the picoLCD graphics screen to be recognized (for LCD4Linux)
12) libgd-dev – For displaying images on the LCD (for LCD4Linux)

Development was done in the NetBeans IDE, available at www.netbeans.org. The documentation was created use Doxygen, available at http://www.stack.nl/~dimitri/doxygen/.

The latest copy of the SCADA source code as well as an executable should be available at the project webpage: ww2.lafayette.edu/~ece492-2009.

## Architecture

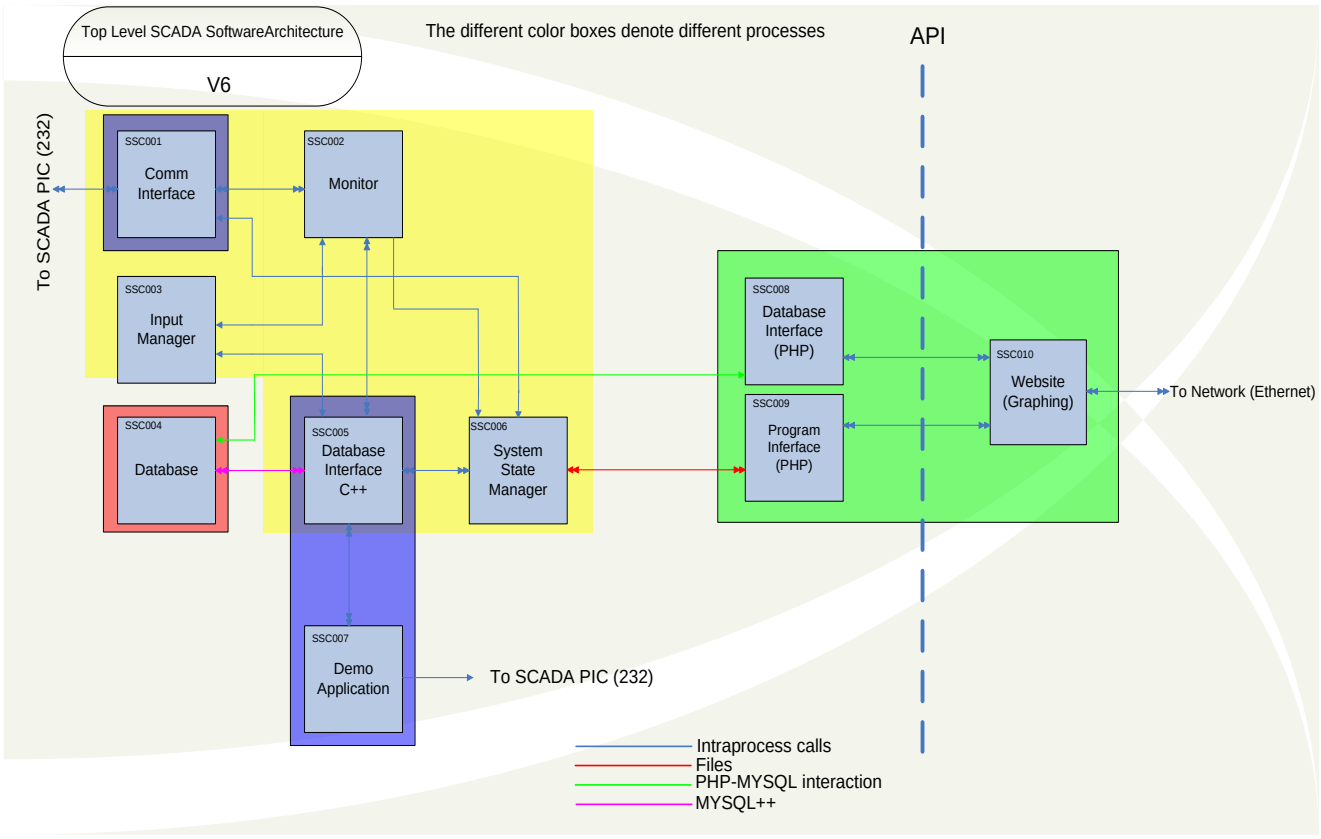



**Figure 1: Top level SCADA software architecture**

## Source Code

The source code can be broken into three main parts and is stored on the website as such.

a) Data collection (C++)
b) Demo manager (C and C++)
c) Website (PHP)

The database can be considered a fourth component. All four components will be discussed in detail later in this report. Two portions of the source code (data collection and demo manager) should be compiled separately. See the compilation directions later in this document. The PHP code does not need to be compiled.

## Compilation of Source Code

The code was compiled using the GNU g++ compiler, version 4:4.3.1-1ubuntu2, found in the standard repository. In addition to including the names of the files in compilation arguments, other libraries must be specified in the form of:

-I /usr/include/mysql++ -L /usr/include/mysql++ -I /usr/include/mysql -L /usr/include/mysql -I /usr/include/cppunit -L /usr/include/cppunit -lmysqlpp –lcppunit

These libraries must be installed prior to the compilation. Since there are two executables which should be run at the same time, it is necessary to compile them separately and run them as two different processes.

## Running the Code

After compilation, the executable should be runnable without any special options specified. Upon starting the Fit-PC, nothing happens automatically. Instead, it is necessary to manually run the programs. The data collection program should be started first, followed by the demo manager program. Currently the data collection portion of the program is set up to write values to the database once a minute.

## Logins/Passwords

For logging into the Fit-PC, the information is:

Username: fit
Password: 111111

For the MYSQL interactions, the login information is hardcoded in the DatabaseInterface class for the C++ as well as the DatabaseInterface class for the PHP portion of the source code. The information is:

Database: LPRDS
Host: localhost
Username: root
Password: 111111

To log into the maintenance portion of the SCADA website, the password is **12345** which has been hardcoded into the PHP.

## Description of the Classes

As mentioned previously, there are 4 main parts to the program:

a) Data collection portion of the software
b) Demo Manager
c) Database
d) Website

Each of these sections is described below:

## Data Collection

The data collection portion of the software is responsible for collecting information from the sensors in the subsystems, processing it, and storing it in the database. The pertinent information about each of the main classes is discussed below.

In the overall flow, the Monitor class requests the latest set of inputs from the Input Manager. The Monitor class then sends commands to the CommInterface to send out queries to PIC boards which then query their sensors. These values are then returned to the Monitor class through the CommInterface. The Monitor class checks these values against known acceptable boundaries. Once every minute these values are written to this database via the DatabaseInterface. Finally, the SystemState class is given control and updates the state based on the new values read in from the sensors, from any faults that the Monitor class may have detected, and from any commands which may have been issued from the website. This means that the overall program flow (found in the main.cpp file) is:

1) Monitor requests latest Inputs from InputManager
2) Monitor polls all of the sensors
3) For each sensor, Monitor checks that the value is OK. If it is not, it informs the SystemState manager.
4) If a minute has passed since the last database write, the values of the sensors are written into the database.
5) The SystemState class updates its state based on current values of the system, faults that may have occurred, and requests from the user which were issued from the website.

**CommInterface:**

*Description:* The CommInterface class is responsible for sending and receiving bytes from the buffer to the PICs. The class has two main methods, sendPacket() and rcvPacket() which can be called to send or receive a certain number of bytes. Currently, the CommInterface class is configured to use a USB port on the Fit-PC. This USB connection is then converted to a standard

RS-232 connection by a crossover cable. This was method was chosen instead of using just a mini RS-232 to RS-232 cable due to issues that were being encountered with the serial port on the Fit-PC.

The format of the queries and commands sent to the PICs can be seen in the communication protocol specification document, available on the project website.

*Known Issues:* There are currently no known issues with the CommInterface class.

*Recommendations:* To ensure that communication is working between two computers, it can be useful to use a known, tested, third party program. On the Ubuntu operating system, we found the SerialGTK program, which can be found in the standard repository, to be useful.

Additionally, the CommInterface class can be tested by running the test cases written for the class, CommInterfaceInitiatorTest.h and CommInterfaceReceiverTest.h files. Two computers are required for running this test. First, on one computer, the CommInterfaceReceiverTest.h test class should be run. On a second computer which is connected by a serial cable the CommInterfaceInitiatorTest.h should be run. Within these two test classes, the methods that should be run are testWrite1() and testWrite2(). The testWrite1() method in the CommInterfaceInitiatorTest.h file sends eight bytes of data to the receiver. The receiver checks the data that it has received against expected values and sends eight bytes to the original sender. The testWrite2() method does the same thing except that it does this 100 times. This is to ensure that a large quantity of data can be correctly transmitted through the program.

Currently, the rcvPacket() method does not have a timeout mechanism built in. It would be useful to include code so that if the specified number of bytes were not received within a certain period of time, the function would return what it had along with a flag indicating what had happened.

**Monitor:**

*Description:* The Monitor class is responsible for creating the requests for information from the sensors (Input objects) as well as processing the returned results. The class sends out requests to the sensors one by one (via the CommInterface class), gets a result, and then checks the value. Each sensor has a set of allowable boundaries, upper and lower, that are associated with it. The Monitor class checks the returned value of each sensor and declares a fault to have occurred if the value exits from of the boundaries. In addition all of the analog sensors, the safety circuit input, which is a logic high if the safety circuit is tripped, is a special digital input. The mappings of all of the sensors can be found in the SCADA hardware documentation.

*Known Issues:* The values that the Monitor class checks are dependent upon the state of the system. For example, if the system is in the shut down state (state 1), it is acceptable and expected for the voltage at the load to be at 0 volts as opposed to the 120 volts that would be expected if the H-Bridge was connected. It is unknown if the system currently handles these conditions.

*Recommendations:*

Currently, the Monitor class has not been formally tested, although it has been informally proven to work in a few instances. The Monitor class should be rewritten for great clarity.

An additional desirable feature would be to add something that would check if the data received from the CommInterface for a poll was correct (i.e. if the CRC was correct). If not, some kind of communication fault could be sent to the SystemState class using the typeTwoFaultDetected() method.

**InputManager:**

*Description:* Each sensor that the system monitors has an Input object associated with it. The Input class provides information about each sensor such as the part number, pin number, type, DateTime of last sensor read, etc. The InputManager class holds a list of all of the Input objects currently in the system. The idea behind this is that the list of Inputs does not need to be pulled from the database for each iteration, causing a large amount of unnecessary inter-process communications.

When the SCADA program is first started up, the InputManager must first poll the database in order to obtain the most recent set of inputs. From here, InputManager passes the current list of the Input objects to the Monitor class each iteration with the Monitor class calls the getSeparatedInputs() method.

*Known Issues:* There are currently no known issues with this class.

*Recommendations:* It is believed that the class overall is functionally correct. Test cases with cppunit are recommended.

**DatabaseInterface:**

*Description:* The database interface provides a method for interacting with the database. At its core, the class uses the mysql++ library, (found at http://tangentsoft.net/mysql++/). Essentially this class acts as a wrapper around the mysql++ library and provides useful calls for the rest of the program. The idea is that the other classes should be insulated from writing complex queries for interactions with the database and being forced to process their results. All methods in this class are static so there is no need to declare an instance. The methods in the class were tested in the DatabaseInterfaceTest.h file, written using the cppunit library (found at http://apps.sourceforge.net/mediawiki/cppunit/index.php?title=Main_Page).

*Known Issues:* There are currently no known issues with the databaseInterface class.

*Recommendations*: The DatabaseInterface class can be tested by running the databaseInterfaceTest.h file. If the program is run and errors are encountered with regards to the database, check that the connection settings for the database are correct, namely the host name,

user name, password, and the name of the database. Additionally, the initialize() method in this class should be called before any interactions take place.

**SystemState:**

The SystemState class is responsible for controlling the state of the system. The system is constantly in one of the five system states. These states should be found on the project website at (http://ww2.lafayette.edu/~ece492-2009/Documents/FinalDocuments/system/STDv6.pdf) The system is updated (by calling the systemStateRun() method) based on four parameters:

a) The current state of the system
   The current state of the system does not in and of itself cause any transitions to take place but it does affect which state the system will move into.
b) User requests which have been issued from the website.
   The website portion of the SCADA system allows the user to send commands to the system (the exact ones are detailed below). When a user issues a request, a value is written to file by the PHP on the website side of things. The SystemState class reads in the file using the processFile() method, which sets variables in the system indicating that certain things should be done.
c) Faults which have been generated by the Monitor class
   When the Monitor class is checking its values, it may find one that is out of bounds. In this case, the Monitor class tells the SystemState class that a fault has occurred using the typeTwoFaultDetected() method in the StateState class. A type two fault is any fault detected by the Monitor class. A type one fault is an undersupply fault, which occurs when the voltage at the load cannot be met. This type of fault will be discussed shortly.

   If a type two fault occurs, the response of the system is always the same. The SCADA system should trip the safety circuit, disconnect the PV and the H-Bridge, and move the system in state 5, the fault state.

   In order to clear a fault, the user presses a button the website to do so. When this happens, the system moves into the shutdown state. However, if the cause of the fault is still present, the system will be bumped immediately back into the fault state.

d) Current values of the voltages at the batteries and the load
   If the voltage at the batteries or the load drops below a certain threshold an undersupply fault, also known as a type one fault, has occurred. If this happens, the system moves into state 4 in which the H-Bridge is disconnected.

The details of the four parameters are:

a) The current state of the system
   1) Safe shut down
   2) PV disconnected, H-Bridge connected
   3) PV connected, H-Bridge connected

4) PV connected, H-Bridge disconnected
5) Type 2 Fault

b) User requests which have been issued from the website
1) Turn the system on
2) Turn the system off
3) Simulate a fault
4) Clear a fault
5) Connect the PV
6) Disconnect the PV
7) Connect the H Bridge
8) Disconnect the H Bridge
9) Put the system into automatic mode
10) Open safety circuit

Not all of the user requests are valid in all states. The table below shows which user requests are valid in which states where a 1 indicates a valid command and a 0 indicates an invalid command. Commands are sent via the website and are read in by the SystemState class.

| State | Turn On | Turn Off | Clear Fault | Simulate Fault | Disconnect PV | Disconnect H-Bridge | Connect PV | Connect H-Bridge |
|---|---|---|---|---|---|---|---|---|
| State 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| State 2 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| State 3 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| State 4 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| State 5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

**Table 1: Allowable actions in the system states**

c) Faults which have been generated by the Monitor class
1) Measured a value which exceeded the allowable threshold
2) Measured a value which was below the allowable threshold
3) Detected the safety circuit being tripped

d) Current value of voltages at the batteries and load
1) If the present voltage at the load is below a threshold, an undersupply fault has occurred.
2) If the present voltage at the batteries is below a threshold, an undersupply fault has occurred.

*Known Issues:* Currently the calls to the CommInterface to do things such as connect and disconnect the PV and H-Bridge are commented out. This was done during the testing phase to cut out these "extra calls".

Note the initialize() method should be called to set the SystemState class up.

*Recommendations:* There are many things which could be improved in the SystemState class. The updateState() method could likely be written in a much more compact way. It is currently quite large and unwieldy.

The SystemState class has test cases written for it in the SystemStateTest.h file. These tests should all run to completion successfully. More thorough testing is required as this is an important class.

In retrospect, the type one fault could be seen as a type two fault but simply interpreted in a special way. This should be considered in the next iteration of this class.

## DemoManager:

The demo application is responsible for starting up the picoLCD display and also for the demo display board. While the display board is actually run by the DemoMain.cc program, the picoLCD is run by LCD4Linux. All DemoMain.cc does is start LCD4Linux, which runs in the background. It does this using a system() call, giving the system the proper command to start LCD4Linux, and pointing it toward the right configuration file. It then moves into a forever loop, where it gets the current system state and then updates the Demo Display, using the Communications interface to send packets telling the SCADA PIC to turn on or off the LEDs. Each LED corresponds to either a state or a switch. The following is the LED response to the system state.

STATE 1:
RPI solid state relay "OPEN", EDS solid state relay "OPEN", EDS inverter "OFF", and "NO LOAD".

STATE 2:
RPI solid state relay "CLOSED", EDS solid state relay "OPEN", EDS inverter "ON", and "LOAD".

STATE 3:
RPI solid state relay "CLOSED", EDS solid state relay "CLOSED", EDS inverter "ON", and "LOAD".

STATE 4:
RPI solid state relay "CLOSED", EDS solid state relay "CLOSED", EDS inverter "OFF", and "NO LOAD".

STATE 5:

RPI solid state relay “OPEN”, EDS solid state relay “OPEN”, EDS inverter “OFF”, and “NO LOAD”.

Additionally the LCD display will change to reflect the current state of the system.

## Database

The SCADA system uses a MYSQL database to store the information about the sensors, measurements, the system states, faults, and events. To set up the database portion of the SCADA system, it is advisable to use the PHPMyAdmin program. The current database is available on the course website. To add this database, create a new database using PHPMyAdmin named LPRDS and import the information into the database using the “Import Data” button.

Currently, we have noticed that the database on the fit-PC appears to freeze regularly. To remedy this problem run:

root@fit: /etc/init.d/mysql restart

The database is comprised of 9 tables as seen below:

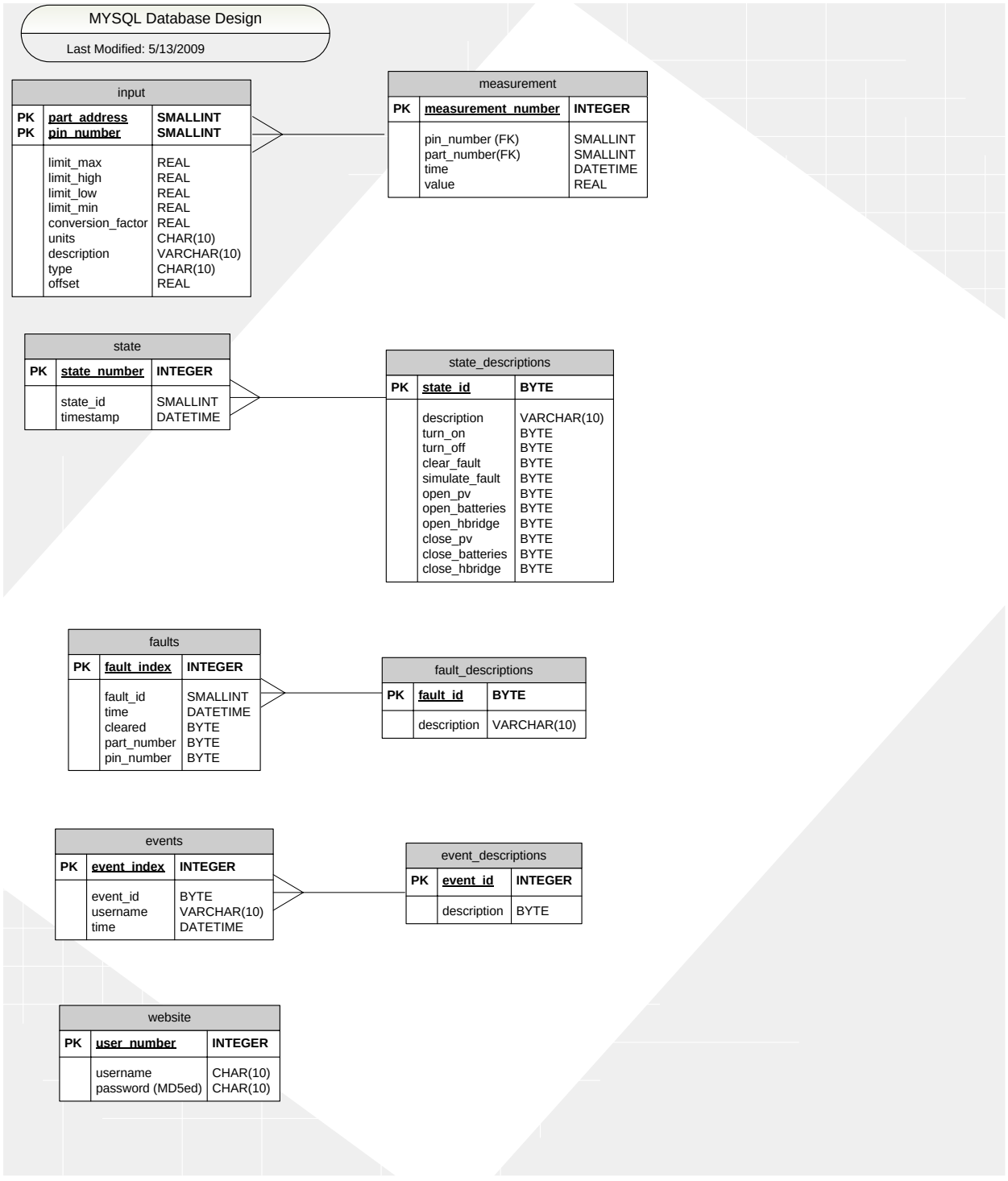



**Figure 2: Database Architecture**

**input:** The input table is responsible for holding the data for all of the inputs that the system is currently observing. The part_number column denotes the PIC number of sensor. A listing of these is found in the SCADA hardware documentation. The pin_number column is somewhat confusing. In the hardware, when an analog sensor value is requested, a "channel" is used as opposed to when a digital value is requested, where the actual pin number is used. In all cases,

from the database portion here, the pin_number is simply the address that the request for the value must be sent regardless of whether or not it is a digital sensor or analog one.

**measurement:** The measurement table is responsible for storing the values of the sensor along with the times that they were read in. These values can be tied to an Input using the part_number and pin_number columns.

**state:** The state table holds a record of the state of the system at all times.

**state_descriptions:** This table holds the state descriptions for each of the five system states. The other columns such as turn_on, turn_off, etc tell whether or not the action is permitted in each of the states. This is the same table as Table 1. These are used to tell the website which actions it should permit the user to take based on the system state.

**fault:** This fault table holds a record of all faults that the system has encountered over its lifetime. The cleared column tells if the fault is still active or if it has been cleared. The part_number and pin_number are used to tie the fault to a specific location.

**fault_descriptions**: This table tells of the 4 different faults that the system can encounter. These are detailed above in the SystemState description.

**events**: Events are written to the database whenever a user issues a request to perform an action on the system.

**event_description:** This table holds the description of each of the events

**website**: This table in theory holds the usernames and passwords for the users which are able to access the website. In actuality, this table is not actually used. The password for the website is hardcoded into the PHP code for the website. This table could possibly be used in the future.

## Website:

The website provides a way for users see the values that are being generated by the sensors and stored in the database. The website runs is built on PHP, mainly due to the easy interactions it provides with MYSQL. Additionally, the website provides a way for the user to send requests to the Data Collection portion of the SCADA system. On the Fit-PC, the website can be found in the /var/www folder, which is the default folder on Ubuntu for the Apache2 webserver. A description of each of the important classes in the website follows:

### DatabaseInterface

*Description:* The DatabaseInterface class on the webpage acts in a manner very similar to the DatabaseInterface class in the Data Collection portion of the website (described above). This class provides useful methods to the rest of the program to insulate them from the details of these interactions.

*Known Issues:* In the getSpecialValues() method, the battery voltage is incorrectly calculated. In the method, the voltages at the four battery terminals are added, which is incorrect. In reality, the battery sensors are placed at ¼, ½, ¾, and the total battery voltage. This means that in order to get the total battery voltage, only one sensor need be read.

*Recommendations:* The DatabaseInterface class has been tested in the DatabaseInterfaceTest class. All methods have been tested except for the getTimeValues() and getSpecialValues() methods due to time constraints. I recommend that test cases be written for these methods.

Overall, I believe that the DatabaseInterface class performs as expected and provides a solid foundation for the rest of the website.

**Plotting**

*Description:* The plotting functionality of the website is handled in the makedata.php and plot.php files. The code first requests the values from the database for the user-specified time period. It then creates the plot using the gnuplot tool. The graph covering the specified time should then be displayed.

*Known Issues:* Currently the code does not appear to be working correctly. I am unsure what the problem is.

*Recommendations:* None at this time.

**Issuing user commands**

*Description:* Commands are issued by writing to a text file. All the commands except the one to silence the alarm are written to a file that is read in by the SystemState class. These commands include ones to turn on and off the system, connect/disconnect the PV and the H-Bridge, and put the system in automatic mode. The command to silence the alarm is read in by the Monitor class and is placed in a separate text file.

*Known Issues:* Currently the command to silence the alarm is expected by the Monitor class to be simply a 1. However, in the ProgramInterface class on the website, a full line of data, containing the name of the user making the request, the DateTime of the request, etc.

*Recommendations:* The ProgramInterface method has been tested and appears to be working. However, full testing with the integrated system has not occurred and is recommended.

**Other Classes**

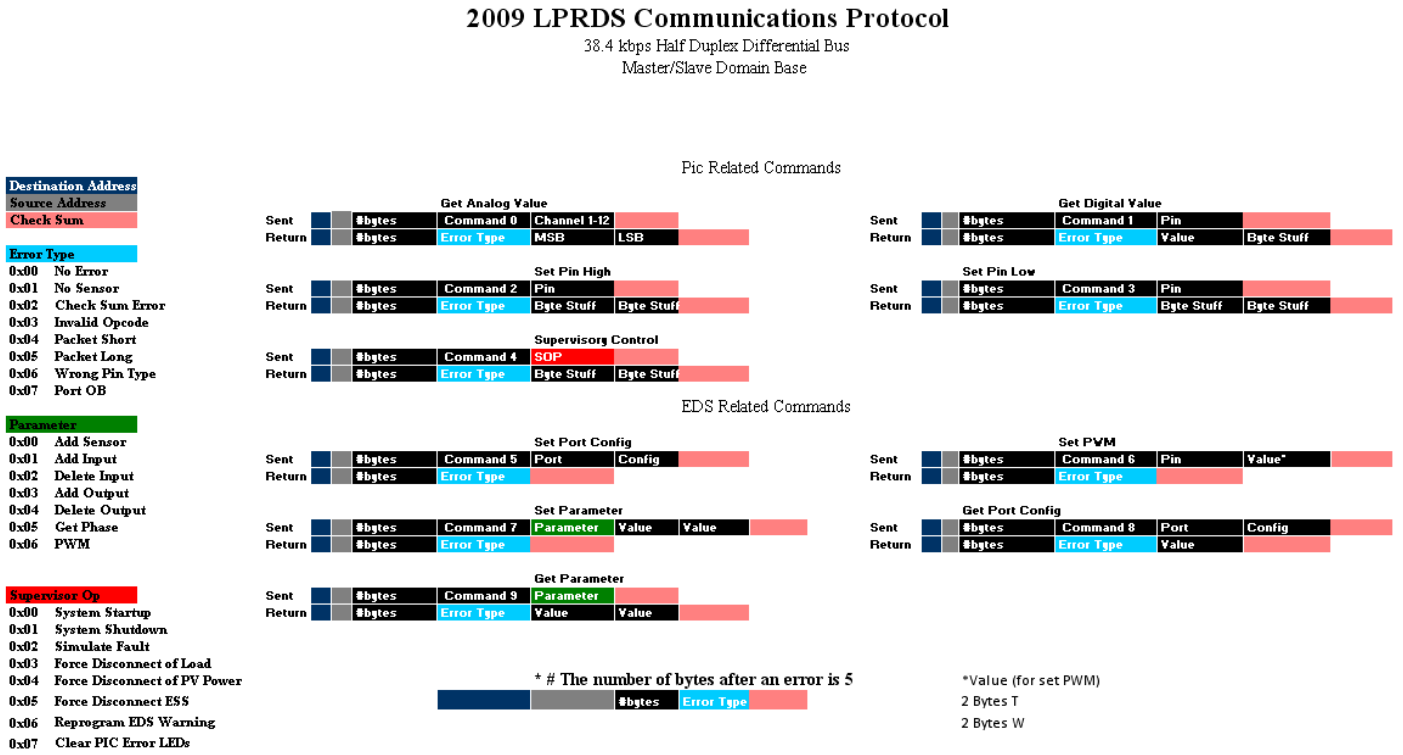
*Description:* The other classes used for displaying the website are detailed in the User's Manual under the SCADA section.

*Known Issues*: None at this time

*Recommendations*: None at this time.

## PIC Interactions

The PC polls values from the sensors by interacting with the PICs on each board. In order to do this, a communication protocol was developed. The bits are sent at 38,000 baud. The communication protocol can be seen below:




All bytes in the protocol are eight bits. As with the SCADA source code, the PIC code should also be found on the project website at ww2.lafayette.edu/~ece492-2009.

## Testing

Limited testing was performed using cppunit, a C++ software library for writing test cases. All test cases were given names corresponding to their functions. For example, the DatabaseInterfaceTest tests the DatabaseInterface class. The test cases are run through the main() function, which contains commented-out code which appears as:

```
CppUnit::TestSuite suite;
suite.addTest(SystemStateTest::suite());
suite.addTest(DatabaseInterfaceTest::suite());
suite.addTest(DateTimeTest::suite());
suite.addTest(CommInterfaceInitiatorTest::suite());
suite.addTest(CommInterfaceReceiverTest::suite());
```

```
suite.addTest(MonitorInitiatorTest::suite());
CppUnit::TextTestResult res;
suite.run( &res );
```

This code is used to run the different test cases written for the system. Each of the test cases be can commented out individually to run only the desired tests. Additionally, some classes, such as the CommInterface require that there be two files for testing them, an initiator and a receiver. In these cases, the two files are run on two separate computers connected by a serial cable. In this case, it is necessary to start the receiver portion of the software before the initiator because the receiver must be listening first.

Testing for the PHP code was performed using the SimpleTest library, found at http://www.simpletest.org/

## LCD4Linux Troubleshooting

All documentation for LCD4Linux can be found at http://ssl.bulix.org/projects/lcd4linux/. However, this documentation is not very helpful, so the following are some tips for writing the .conf file.

First, always make sure that there is a space between the widget name and the brace. For example, this is correct:

```
Widget CPU {
    class  'Text'
    expression  uname('machine')
    prefix 'CPU '
    width  9
    align  'L'
    update tick
}
```

This is incorrect:

```
Widget CPU{  <- NO SPACE
    class  'Text'
    expression  uname('machine')
    prefix 'CPU '
    width  9
    align  'L'
    update tick
}
```

If you do not leave a space, the program will not compile the file correctly.

Next, when creating the layout, ensure that there is a space between the column number and the widget name. Also ensure that there is a space between the row number and the brace. For example, this is correct:

```
Layout testMySQL {
        Row1 {
                Col1 'MySQLtest1'
        }
        Row2 {
                Col1 'MySQLtest2'
        }
}
```

This is incorrect:

```
Layout testMySQL {
        Row1{   <-NO SPACE
                Col1'MySQLtest1' <-NO SPACE
        }
        Row2 {
                Col1 'MySQLtest2'
        }
}
```

Again, the file will not compile if this is not correct.

Some good examples of configuration files for the picoLCD Graphic display can be found in the documentation on the LCD's website, located here: http://www.mini-box.com/picoLCD-256x64-Sideshow-CDROM-Bay;jsessionid=0a0105501f435f9ee863c03848a5be2eade9a28f3a9a.e3eSc34OaxmTe34Pa38Ta38Raxj0.

When creating a plugin for LCD4Linux, make absolutely sure you have followed the instructions for including it when the program is recompiled. These instructions can be found here http://ssl.bulix.org/projects/lcd4linux/wiki/Plugins. When you have finished creating the plugin, make sure you recompile the program with the ./configure command. Then run a make and then a make install. If the compilation worked, you should see your *.o* file included in the compiled plugins list after make finishes.