EMCOR Bench-Test

Design Document

Revision:	0.4
Status:	Released
Repository:	Share Point
Project:	LCLS MCOR AIP
Folder:	N/A
Document ID:	
File:	EMCOR Bench-Test Design Document
Owner:	
Last modification:	February 26, 2016
Created:	October 24, 2012

Prepared by	Reviewed by	Approved by
Tomo Cesnik	Briant Lam	

Document History

Revision	Date	Changed/reviewed	Section(s)	Modification
0.1	2014-11-27	Tomo Cesnik		First version
0.2	2014-12-15	Gasper Jansa	1 and 2	Added requirements
0.3	2015-01-06	Tomo Cesnik	4, 5, 6	Updated APIs & GUIs
0.4	2015-04-09	Tomo Cesnik	4	Update according to the split of RemotePCI functionality

Confidentiality

This document is classified as a public document. As such, it or parts thereof are openly accessible to anyone listed in the Audience section, either in electronic or in any other form.

Scope

This document represents requirements and design for EMCOR controller test environment.

Audience

This document is targeted to all persons involved in the EMCOR controller test environment.

Table of Contents

Figures

Tables

No table of figures entries found.

Glossary of Terms

API	Application Programming Interface
CPU	Central Processing Unit
EPICS	Experimental Physics and Industrial Control System
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HW	Hardware
IO	Input / Output
SW	Software
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

References

[1] EMCOR Bench-Test Procedure

[2] FUNCTIONAL REQUIREMENTS FOR MCOR SLOT- 00 CONTROLLER CARD, Rev 3, 20 September 2011

[3] EMCOR Hardware User's Manual

[4] EMCOR Tester Programming Guide

[5] EMCOR EVR Bench-Test Design Document

1. Introduction

Automatic testing of the hardware usually includes development of the simulator that can check the outputs of the hardware and feed some data back to its inputs. The simulator should verify that the hardware under the test is behaving correctly in all test cases and that it meets its design requirements. For EMCOR controller board testing, EMCOR tester board is used to verify the correctness of the controller. In this document we present the functional requirements and design for the test environment used for testing EMCOR controller, from hardware and software to the user's point of view. The test cases are explained in the test plan document [1].

2. Functional Requirements

This chapter lists the functional requirements for EMCOR bench-test software.

1. The bench-test software should provide user interface for automated tests.
2. The bench-test software should generate a report for the automated tests.
3. The bench-test software should provide a user interface for manually testing DACs and ADCs.
4. The bench-test software should provide user interface for displaying the EMCOR board voltages and temperatures.
5. The bench-test software should test all digital I/O, DACs, ADCs by writing to, and reading from the firmware interface.
6. The bench-test software should test the following EMCOR requirements: R01, R02, R03, R05, R06, R11, R13, R15, R18, R19, R20, R25, R26, R28, R29, R31, R33. [2]

3. System Overview

The test setup consists of the following hardware:

- EMCOR controller board
- EMCOR tester board
- P2 Marx Hardware Manager
- Testing machine (i.e. Linux or PC Host)
- Cat6 Ethernet cable
- RS232 serial cables. A straight through cat6 Ethernet cable can be used, if connecting from the EMCOR serial port to a DIGI terminal server port.
- Kontron CPU (COMe-CDcD2 N270) with 2GB, PC2-6400, 800MHz, SoDimm RAM and heat sink installed.

As shown below in Figure 1, the EMCOR Controller board is connected, with all of its I/O, to the EMCOR Tester board. The Controller board's I/O is managed by the on-board FPGA and CPU through the PCI bus. A special program that runs on a Controller board enables PCI access over Ethernet. On the other side, the Tester board runs similar software that enables manipulation of the IO signals through the same medium. Both boards are connected to the testing machine, or host over Ethernet.

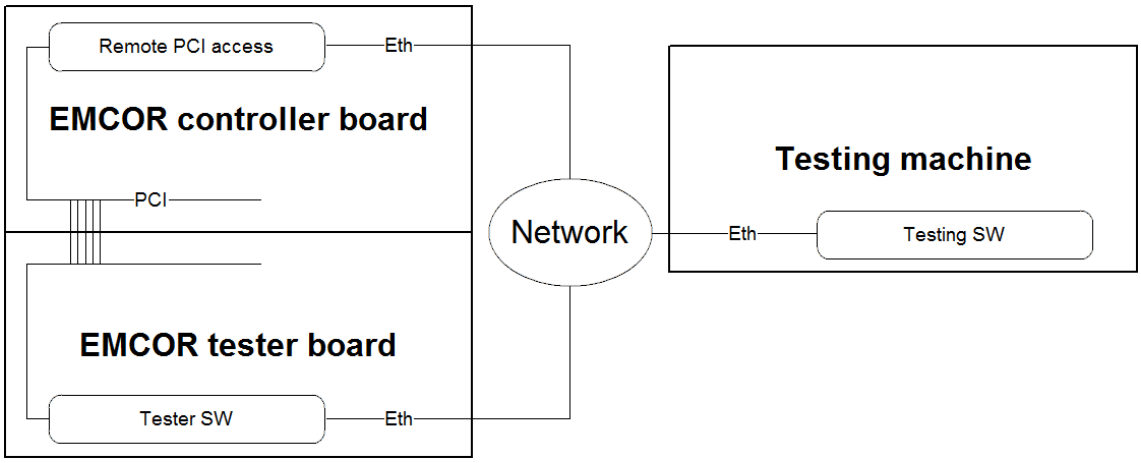

Figure 1: System Architecture

Software running on a Tester board uses UDP Ethernet protocol and additionally specifies how each packet should look like. The software expects the UDP packet follow a specific format, including a header and data. The header contains the protocol version, task ID, memory address and size (i.e. Number of bytes). The data contains the actual data to be written to or read from a specified memory location. Both the user and server sending UDP packets are expected to format the information accordingly. Using this UDP service, the bench-test

software is able to access the necessary data, and verify that the Controller board is working correctly. For protocol details, reference the EMCOR Tester board document. [4]

On EMCOR Controller board, a special server must be run that exposes PCI registers on the network. The protocol for accessing these registers is similar to the one in the tester board software. Communication can take place using UDP or TCP Ethernet protocol. Network packets are text-based, so the user must just provide a string in appropriate format to construct the message. A user's requests contain the version, signature, operation mode (R/W), width and offset of data to read/write. The response from the server is structured in the same way. With this functionality the EMCOR Controller board can be managed remotely without EPICS. For details on the protocol, see EMCOR Controller board document. [3]

The bench-test software uses Python (version 2.7.4) to test the EMCOR Controller board functionality. Since the Controller and the Test board can both be accessed on the network through standard Ethernet protocols, the bench-test software uses the Python ***socket*** module, to communicate with the servers. Protocols for reading and writing are abstracted into separate classes, hiding the implementation of message construction and parsing. A special API is available to the user, which work with both servers; creating appropriate and reliable test cases. For details on this API, reference chapters 4 and 4.1.

Construction of automatic test cases is made available with the Python ***unittest*** module. Test cases should be sensibly structured into classes by the functionality that they test. For test cases that require user interaction, the ***PyQt*** Python module (version 4.11.3) is used for displaying the GUI elements. Two separate GUIs are available one is used for setting and monitoring all the channels on the EMCOR Controller board, while the other one is used for running the tests automatically. The GUIs are described in chapters 5 and 6.

4. EMCOR Controller API

Testing software needs to communicate with EMCOR controller to write/read its values and check if they were appropriately applied and handled. On the controller board, a server that exposes board registers on the network must be running. Special API that connects to this server is implemented in Python. It hides all implementation details from the user and only exposes certain function calls. How the API looks can be seen on Figure 2.

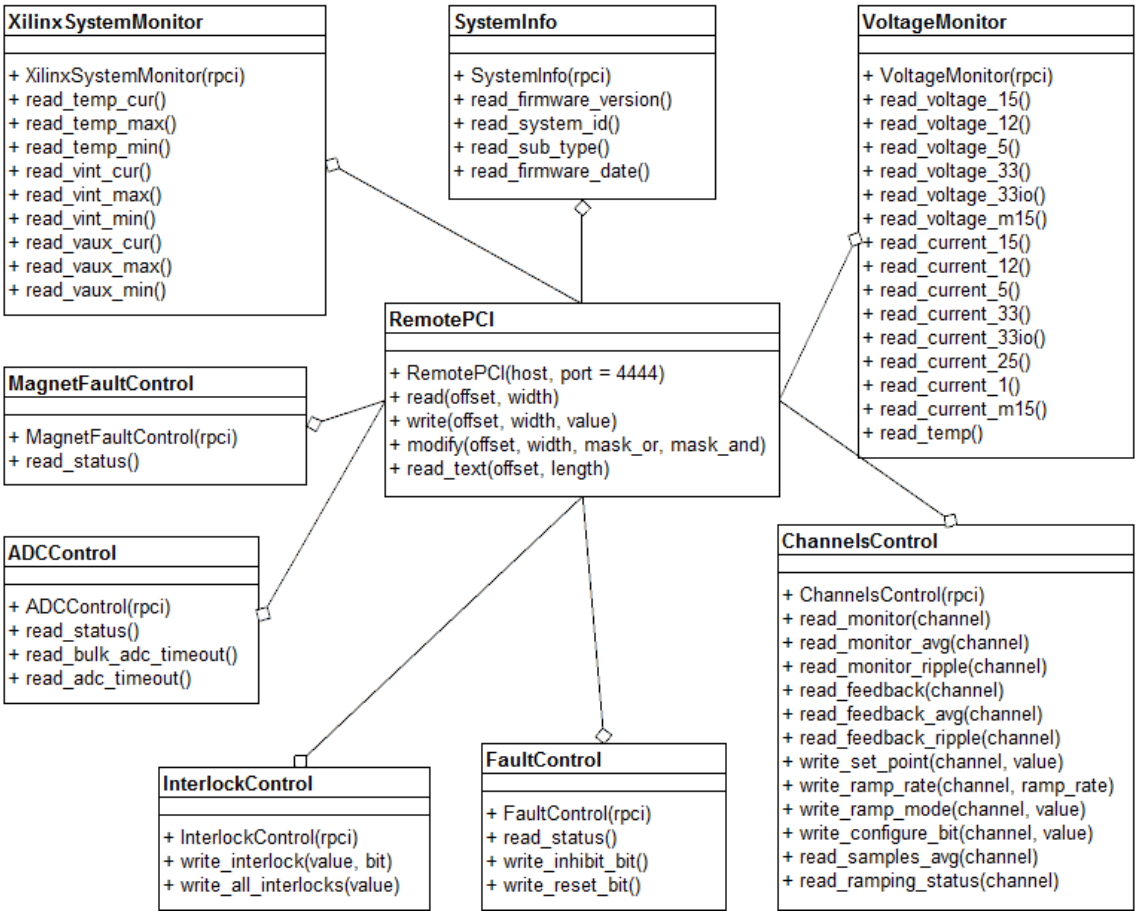

Figure 2: EMCOR Controller API

Implementation is split into two parts. The first and most important part is the ***RemotePCI*** library for communicating just with the server directly. The core class is ***RemotePCI***. User can create a new instance of it by passing host and port values of the server to its constructor. The value for a port is optional and if not provided will be set to default value of 4444. When the instance is created, the connection with the server will be established and ready to use. The used resources are automatically cleaned up when the instance goes out of scope in the user code and the de-constructor is automatically called.

The user can use three basic functions to work with the controller: read/write/modify. All of them have the same first two arguments. Values for register offset and width must always be specified, since all the commands work with the controller board registers directly. To check

which register offsets are valid for the board, check its documentation. There are three different register widths supported: 8/16/32 bits. If invalid value is specified, ***ValueError*** exception will be raised.

Read method doesn't need any additional parameters. It returns the actual register data that was requested in the format of an integer number. Any additional data needed by the protocol is stripped and not included in the return value. In case that the communication with the server fails or the passed values are rejected on the server, an ***RPCIProtocolException*** will be raised. This exception is unique to the ***RemotePCI*** API, so it can be used by the user to catch problems regarding to the communication protocol.

Write method requires value parameter, which is a new value that should be written to the specified register. The method doesn't return anything, but it raises an ***RPCIProtocolException*** if the communication with the server fails.

Modify method requires two additional parameters, OR and AND masks for modifying register's existing value. The new register value is set in the following way:

```
new_value = (old_value | mask_or) & mask_and
```

The method returns a newly set value in integer format or raises an ***RPCIProtocolException*** if the communication fails.

There exists additional method for reading the text directly from the specified register offset. Its advantage is automatic conversion of numbers to the character strings. Note that its length argument is the number of characters and can be any positive number.

The second part of the implementation consists of ***EmcorController*** library. Since using ***RemotePCI*** functions requires user to know the register map of the device, additional utility classes are provided that make working with the device easier. Each of them represents certain functionality of the device and exposes functions to use this functionality. They are all created in the same manner, by passing the instance of ***RemotePC***I class to their constructor, because they all use its read/write functions underneath. For details on the utility classes, see Figure 2 or check the source documentation.

4.1. EMCOR Tester API

For communication with EMCOR tester, simple Python API is provided that enables sending of UDP packets and receiving the responses. All low-level implementation is hidden from the user through the usage of Python ***EmcorTester*** class (see Figure 3).

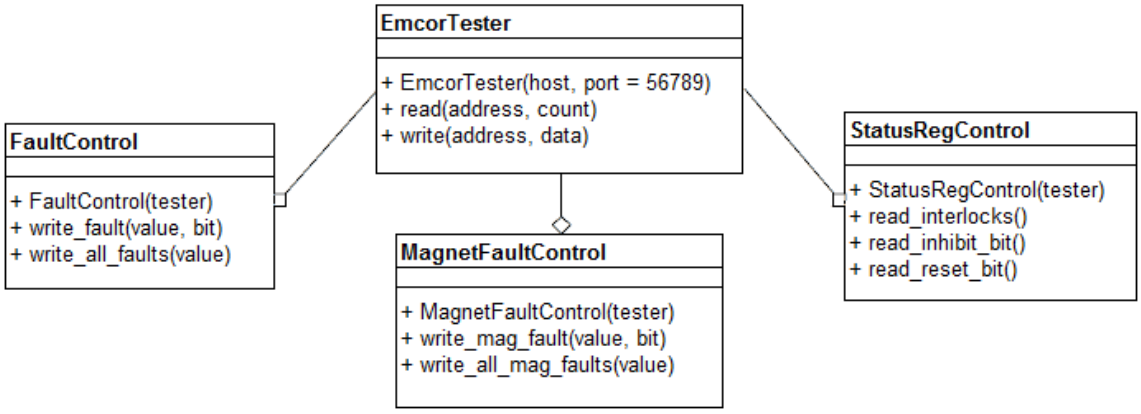

Figure 3: EMCOR Tester API

The instance of this class can be created by providing two parameters to its constructor, hostname and port number of the server. If port number is not provided, the default one 56789 will be used. Similar as in ***RemotePCI*** Python class, when its instance goes out of scope in the user code, all used resources will be automatically cleaned.

There are two functions available to the user of this class: read and write. The first and most important parameter for both of them is the memory address. Read function takes another parameter - count - which signifies how many bytes should be read from the tester board. The function returns read data as an array of bytes or raises an ***ETProtocolException***, which denotes that there was a problem in the communication protocol. Write function requires an array of bytes as a parameter, which it then sends to the server to get written on the specified memory location. If the procedure fails, an ***ETProtocolException*** is raised.

Additional utility classes are also provided that hide the complexity of addressing the memory space of the device. They can be constructed by passing the instance of ***EmcorTeste***r class to their constructor. Underneath they use read/write functions of the core ***EmcorTester*** class, but by using them the user doesn't need to know the register map of the device. For details on the utility classes see Figure 3 or check the source documentation.

5. EMCOR Controller Channel GUI

GUI for EMCOR channels controller enables the user to get instant feedback from the device. First the connection to the device must be established. This is done by providing hostname or IP address to the most top-left text box and pressing the `Connect` button. If the connection cannot be established, the dialog box will be displayed with the details of the error. If the connection is successful the button's text will change to "Disconnect" and the whole GUI will be populated with the values read from the board. On the GUI's left side, basic device information is displayed, like its name, temperatures and voltages.

On the right side, IO channels can be controlled and their actual values are displayed. Therefore each channel has several GUI components. For setting the currents, there is a textbox with optional up/down arrows for changing the values in a discrete way. Besides this textbox, there is a button for actually sending and applying the values on the controller. There are also four textboxes that display the monitor and feedback voltages and ripple. Provided are also controls for setting the voltages on all the channels simultaneously. This is especially useful for testing and calibration. Note that the GUI is updated continuously while the connection is active, so the user can see the changes immediately. Mockup of the GUI can be seen on Figure 4.

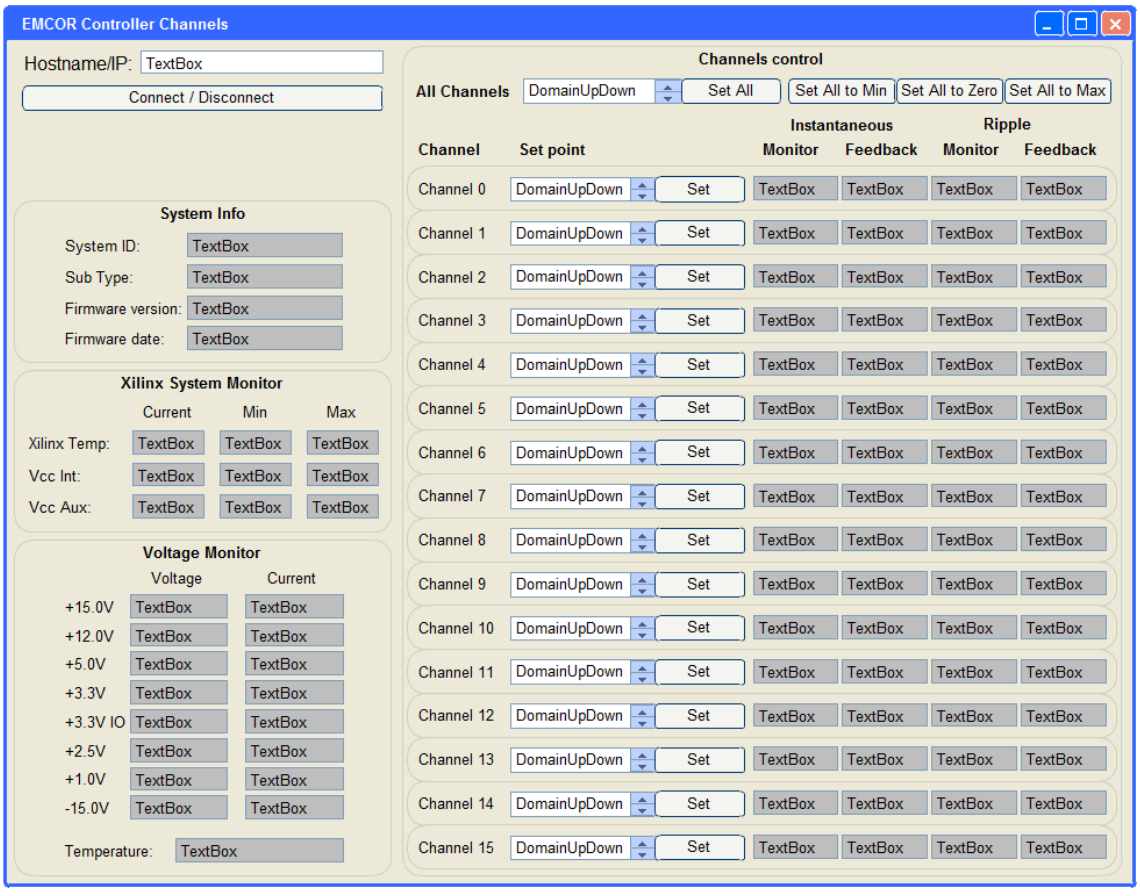

Figure 4: EMCOR controller channels GUI mockup

6. EMCOR Controller Automated Test GUI

For easier testing, simple GUI for running all the tests is provided. It consists of two parts. On the left side, there is a test selector box that displays all the tests in a tree structure. The top node contains all the tests, on the sublevels there are test suites and on the last level test cases are displayed. User can select if he/she would like to run just the test case, test suite or all the tests automatically. When selection is made, Run button can be clicked to run the test/s in the background. The button title is then changed to "Stop" and by clicking it, the test that is currently being run is stopped. On the right side of the GUI, the details of the selected test case are displayed. The visible attributes are name, description, status and duration of the last run as well as output of the last run, which enables user to see why the test failed or the details of the testing process. If test suite is selected in the tests tree view, the details of the suite are displayed on the GUI's right side. The details include name, status, duration and number of the tests with aggregated results. While the tests are run, the GUI is automatically updated with the test results, so the user can immediately see failure reasons or the details of the testing process. For the mockup of the GUI screen, see Figure 5.

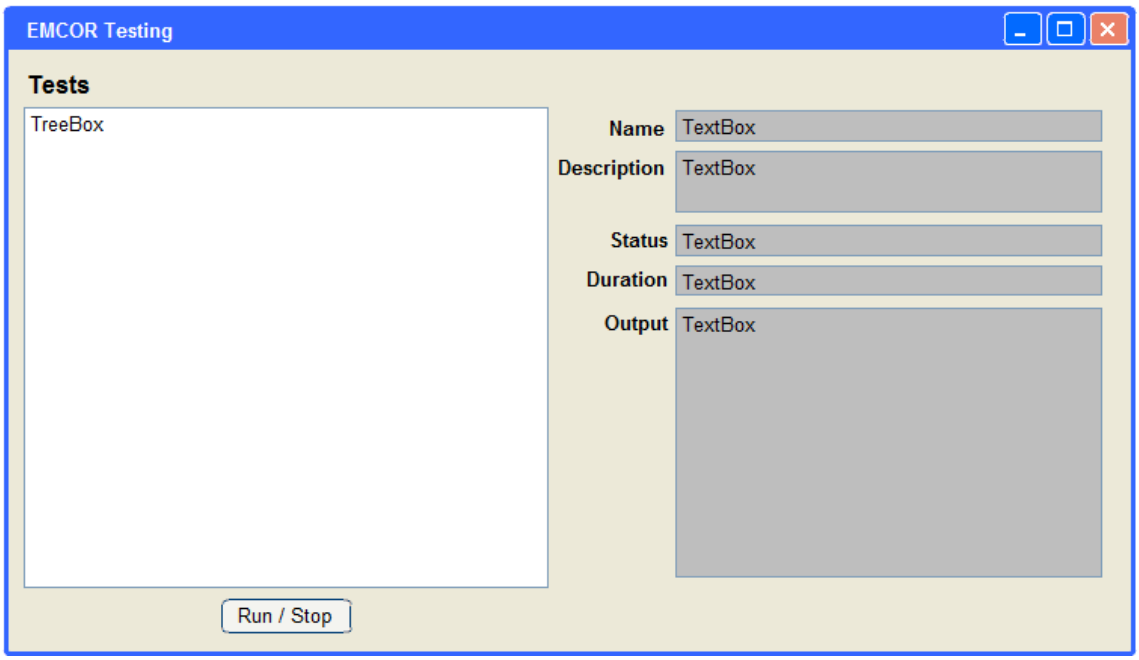

Figure 5: EMCOR controller tests GUI mockup

GUI also contains a menu, which is used for managing the test sessions. Its mockup can be seen on Figure 6.

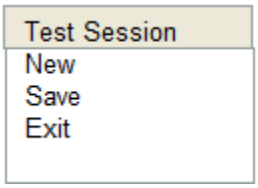

Figure 6: GUI menu mockup

By clicking on "New" menu item, dialog box is shown for choosing the directory from which to load the tests. Also all the test statuses in the GUI are cleared. Menu item "Save" shows file selection box for selecting the location to save the report to. By default, test reports are named as test-report-<ISO8601 DATE>.xml, where the date is the current date and is specified in such form: YYYYMMDDThhmmss. Since time zone designator is omitted, local time is assumed. The user is also able to override the name.

6.1. Test Reports

After the tests are run, the report of the test session can be saved to a file. The file uses XML file format that tries to be as similar as possible to the JUnit/XUnit XML test report file format, which is an unofficial standard for test reports. Report generation mimics other Python tools that produce such reports, but saves all the tests from the tests hierarchy tree that were run, not only the erroneous ones. The XML therefore contains all information about test suites and test cases, their names, grouping, number of tests run, test run durations and error messages. It also contains configuration variables that were used for testing. Example report file can be seen below:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="AllTests" errors="0" tests="0" failures="0" time="0">
    <testsuite name="TestSuite" errors="0" skipped="1" tests="3" failures="1"
time="0.006">
      <properties>
         <property name="hostname" value="host.slac.stanford.edu" />
      </properties>
      <testcase name="testId" time="0.006">
         <failure message="test failure">Assertion failed</failure>
      </testcase>
      <testcase name="testTemperature" time="0">
         <skipped />
      </testcase>
      <testcase name="testResponse" time="0" />
   </testsuite>
</testsuite>
```

6.2. Test Configurations

The tests in ideal world are self-sufficient and use as little external configuration as possible. There are still some variables that can vary in different testing environments, for example IP address of the testing board, controller board ... For defining such configurable parameters the

EMCOR testing framework uses configuration file in `INI` format. Each section in the file specifies the Python module and optionally class for which the configuration values should be applied. The module and the class are separated by the dot character. If the section name is "*", the configuration will be applied to all test modules. In each section more variable names and values can be specified. Example:

```
[*]
IP=111.111.111.111

[input_tests]
IP=123.123.123.123

[input_tests.TestFaults]
channels=5
```

Note that the configuration is injected only if the module variable or class variable already exists. The type of the variable is casted to the same type as it was defined before. More specific definition of the module configuration variable takes precedence over the generic one. In the provided example all the test modules will get their "IP" variable set to "111.111.111.111", but for the "input_tests" module, this variable will be set to value "123.123.123.123".