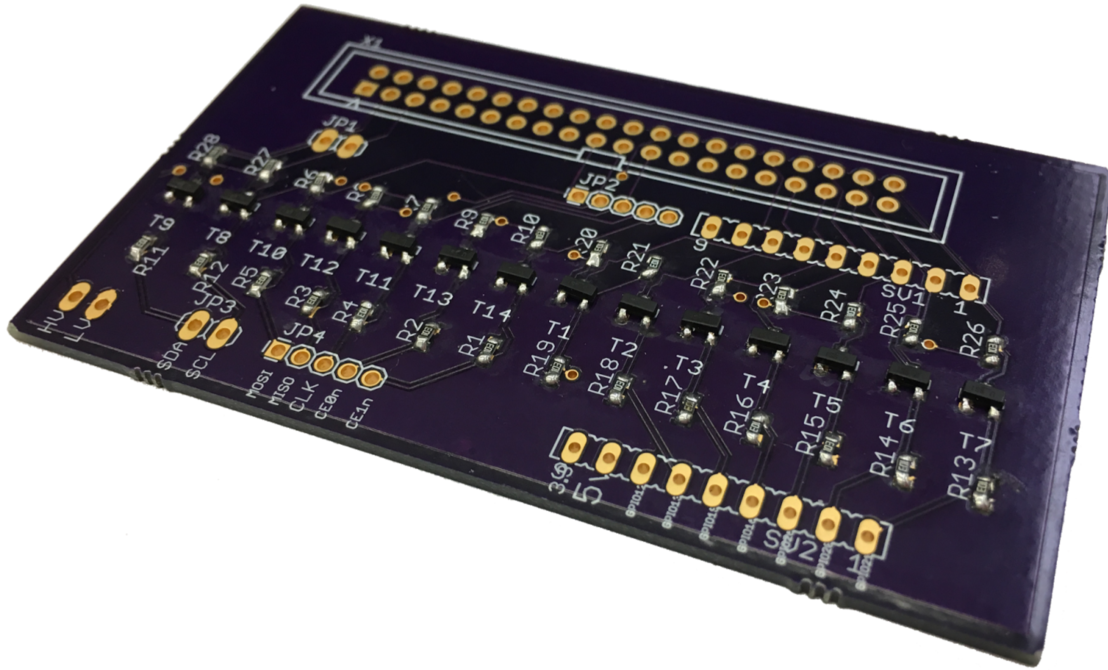
TEAMNAME

Testing Environment for Accessing and Monitoring Networked Automation and Measurement Equipment



Christian Hurst
Braden Rosengren
Antonio Montoya
Benjamin Wiggins
Chris Little

Senior Design | Spring 2017 | MAY1733

1 Introduction **2**
1.1 Project statement 2
1.2 Purpose 2
1.3 Goals 2
2 Design Details and Method **4**
2.1 Design Details 4
2.2 Design Specifications 5
2.2.1 Non-Functional Specifications 5
Software Requirements 5
Hardware Requirements 5
2.2.2 Functional Specifications 5
Software Requirements 5
Hardware Requirements 5
2.3 Design Analysis 5
3 Implementation **7**
4 Testing and Development **11**
4.1 Interface Specifications 11
4.2 Hardware/Software 11
4.3 Design Process 11
4.4 Testing 12
4.4.1 Hardware Testing 12
4.4.2 Software Testing 12
5 Results and Conclusions **13**
Appendix I Operation Manual **14**
1 Building and Installation 14
2 Creating and Running Tests 15
3 Viewing the Results 17
4 Development Resources 17
Appendix II Alternative Designs **18**
Original Project Scope 18
The Operating System 18
The Web Server 18
Appendix III Citations and Resources **19**
Appendix IV Glossary of Terms **20**

1 Introduction

1.1 Project statement

We have set out to create an open source platform for hardware test automation and remote test execution and data collection. The platform allows users access remotely through any web browser, and facilitates the uploading and execution of custom test scripts as well as the viewing and downloading of test results. The platform is built on a Raspberry Pi 3. The Raspberry Pi hosts a web server, and a user interface with the system through the browser based user interface (UI).

The Raspberry Pi can deploy, at the user's command, user defined and uploaded test scripts, store and report back test results, and allow the user to download the raw test data or view it in their browser. The Raspberry Pi can control and query results from common laboratory equipment such as power supplies, digital multimeters, signal generators, and oscilloscopes. The platform also provides the ability to interface with digital or mixed signal devices through the Raspberry Pi's GPIO pins using variable level shifted outputs set to the device's operating voltage.

1.2 Purpose

This platform allows individuals seeking to perform device characterization, or to automate other laboratory tests, to easily design and implement tests that can be launched remotely. It also facilitates remote data collection. The original project was conceived with users of wafer probing stations in mind, however the functionality of allowing remote test execution and result reporting has a wide array of usage cases, for example, allowing remote device demonstration for instructive or educational purposes.

In addition to allowing for remote access, it provides a cost-effective platform that can be used to implement identical laboratory setups at multiple locations that support uniform test code, which could be of great use to teams working in parallel or wishing to do similar characterization or verification on similar or related devices. The platform also offers an excellent opportunity to allow access to test automation to a wider audience, as an open source solution would provide an inexpensive alternative to costly software and hardware that is currently the industry standard.

1.3 Goals

With this project we hope to demonstrate the viability of the concept outlined in the Project Statement. The scope of this project is too broad to expect to be able to deliver a polished and robust platform with only two semesters of work. It is however, completely reasonable to target a demonstration of the utility of the platform in an arbitrary usage case, such as demonstrating a laboratory exercise from a circuits course. Developing a proof of concept will therefore be our primary goal, and expanding and enhancing functionality will be a secondary goal. We will produce supporting documentation

outlining potential usage cases and documenting the additional work required to adapt the platform to be viable for a broader range of usage cases.

- Primary Project Goals
    - Due to the broad scope of the project, our primary goal is to demonstrate the viability of the concept outlined in the Project Statement
    - Demonstrate the utility of the platform with an arbitrary usage case.
        - Demonstrate control over common pieces of test equipment
        - Demonstrate data collection capabilities with a laboratory exercise from a circuits course
    - Produce an operation manual to allow new users to implement the platform
    - Produce documentation showing how to expand the platform's functionality
- Secondary (Stretch) Goals
    - Expand upon the list of proposed usage cases and implementation plans
        - Create proposal for new test platform and equipment configuration for EE 201 and/or EE 230
    - Create interface API for hypothetical equipment set for EE 201/230

2 Design Details and Method

2.1 Design Details

The purpose of our project was to create a system that would allow access, remote or local, to laboratory electrical testing equipment. It is designed to be accessible and usable regardless of the operating system or device being used. The UI will be accessible via any web browser of the user's choice. All server hosting, device communications, and test execution will be handled by a Raspberry Pi 3 that is directly connected to electrical testing equipment via USB, LAN, and/or IEEE-488 (or GPIB) bus.

The project was split into hardware and software components handled by their respective subteams. The software subteam implemented the web UI using a web server hosted on the Raspberry Pi. The browser-based UI allows the users to upload custom test scripts, schedule and execute tests, and easily collect and view test results, with the option to download the results directly to their personal computer.

The test execution software is implemented in Python and stores the tests as scripts. It stores the results or individual tests in JSON files and these files can be accessed by the user for viewing and download.

The server and the test runner must be able to communicate to send information about the tests to the user. This is accomplished through use of a Unix Socket. The test runner software must interface with the testing equipment through GPIB interface which will be accessible to the Raspberry Pi by a number of methods (the IEEE-488 bus, USB, and Ethernet connections).

2.2 Design Specifications

2.2.1 Non-Functional Specifications

Software Requirements

- Web interface is well organized and self explanatory
- Create instruction manual for building software package

Hardware Requirements

- Connectors are spaced in such a way so as to leave room to easily connect peripherals
- Demonstration of some potential usage cases and proof of concept

2.2.2 Functional Specifications

Software Requirements

- Web server and browser-based UI
  - Allow user to easily define and run tests remotely
  - Documented API allows user to define new tests
- Raspberry Pi must be able to interface with the test equipment over:
  - IEEE-488 bus
  - USB connection
  - Connect to devices over the local network (Ethernet)
- System must begin all required processes upon startup without user intervention

Hardware Requirements

- PCB breakout board for computer to chip communication
  - SPI and I^2C headers provided
  - PCB should implement level shifting of digital control signals to acceptable ranges for use on any device under test
  - PCB should fit onto the Raspberry Pi as an expansion board

2.3 Design Analysis

The design of our project was driven primarily by the functionality outlined above. Since our project is based on being able to control various test equipment we needed to support GPIB, the bus over which most electrical lab equipment can be controlled. Additionally, since I^2C and SPI are commonly used on more complex ICs, having support for level-shifted GPIO communication from the Raspberry Pi was also necessary.

We needed a remotely accessible user interface that was capable of rendering content dynamically depending on the tests uploaded and scheduled by the user. For this we decided to implement our interface using Python CGI scripts and the lighttpd web server.

Our choice of language was driven by our desire for an easily extensible system. Python

was chosen as the language of our implementation since it is easy for new users to learn and has a large online community. Additionally, its straight forward approach to module support makes it ideal for defining tests and API files.

Additional details regarding our design choices can be found in Appendix II.

3 Implementation

The project was handled by hardware and software subteams. The software subteam handled the implementation of the web user interface using the lighttpd web server on the Raspberry Pi. The browser-based UI allows the users to upload custom test scripts, schedule and execute tests, and easily collect and view test results, with the option to download results directly to their personal computer. The web server communicates to the rest of the system using Unix Sockets. An overview of the software communication paths can be seen in Figure 1 below.

The test runner software uses a Python based interpreter to parse the user's scripts and convert user instructions into GPIB commands that can be sent to the lab equipment attached to the Raspberry Pi. These conversions are handled by the hardware API, a series of Python modules. The test runner interfaces with the lab equipment over GPIB through several interfaces. The IEEE-488 bus can be accessed through a GPIB-USB Controller [4] that presents itself to the system as a virtual serial port. This makes it easy to send commands and receive results from the lab equipment since communication is achieved through simple file operations. Devices connected directly through USB behave similarly. Devices connected over the local network can be communicated to using web sockets.

The hardware subteam designed a variable logic level shifter in order to interface with digital devices that operate at logic levels different from the Raspberry Pi's 3.3V GPIO bank. The PCB the variable logic level shifter circuits are placed onto is designed to fit directly onto the Raspberry Pi's external pins for easy use with minimal setup. An overview of the hardware connections can be seen in Figure 2 below.

The level shifter schematic can be seen in Figure 3. The breakout board is detailed in Figure 4 (schematic view) and Figure 5 (layout). The completed fabricated board can be seen (without pin headers attached) on the document cover.

As can be seen in Figure 4 the breakout board is simply many individual logic level shifters (Figure 3). It is designed for the voltage from the GPIO pins to be greater than than the voltage of the DUT. But in cases where the DUT has a greater voltage than the GPIO pins the board will accommodate a reverse polarity of up to -5 volts.

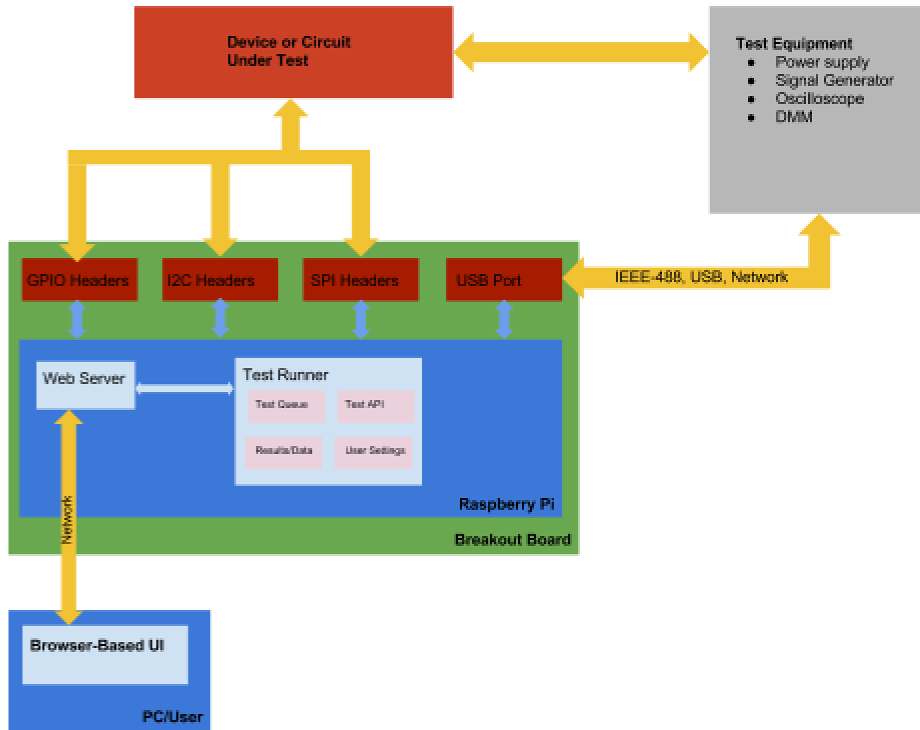


Figure 1 A block diagram outlining the system and modules

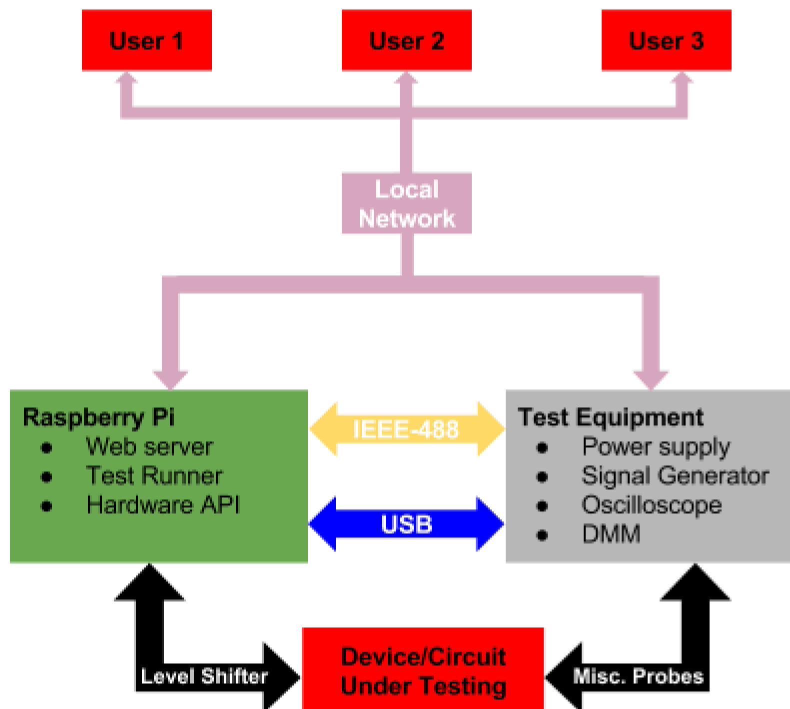


Figure 2 A block diagram outlining the system's hardware connections

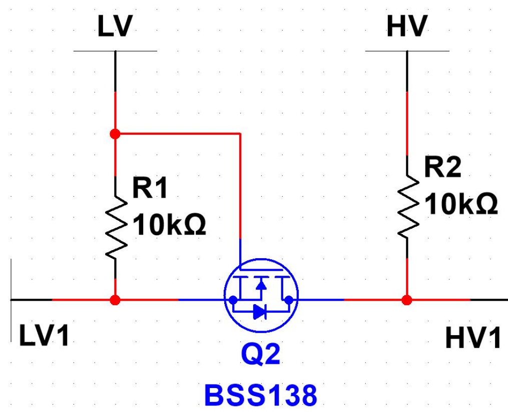


Figure 3 Schematic of Single Level Shifter

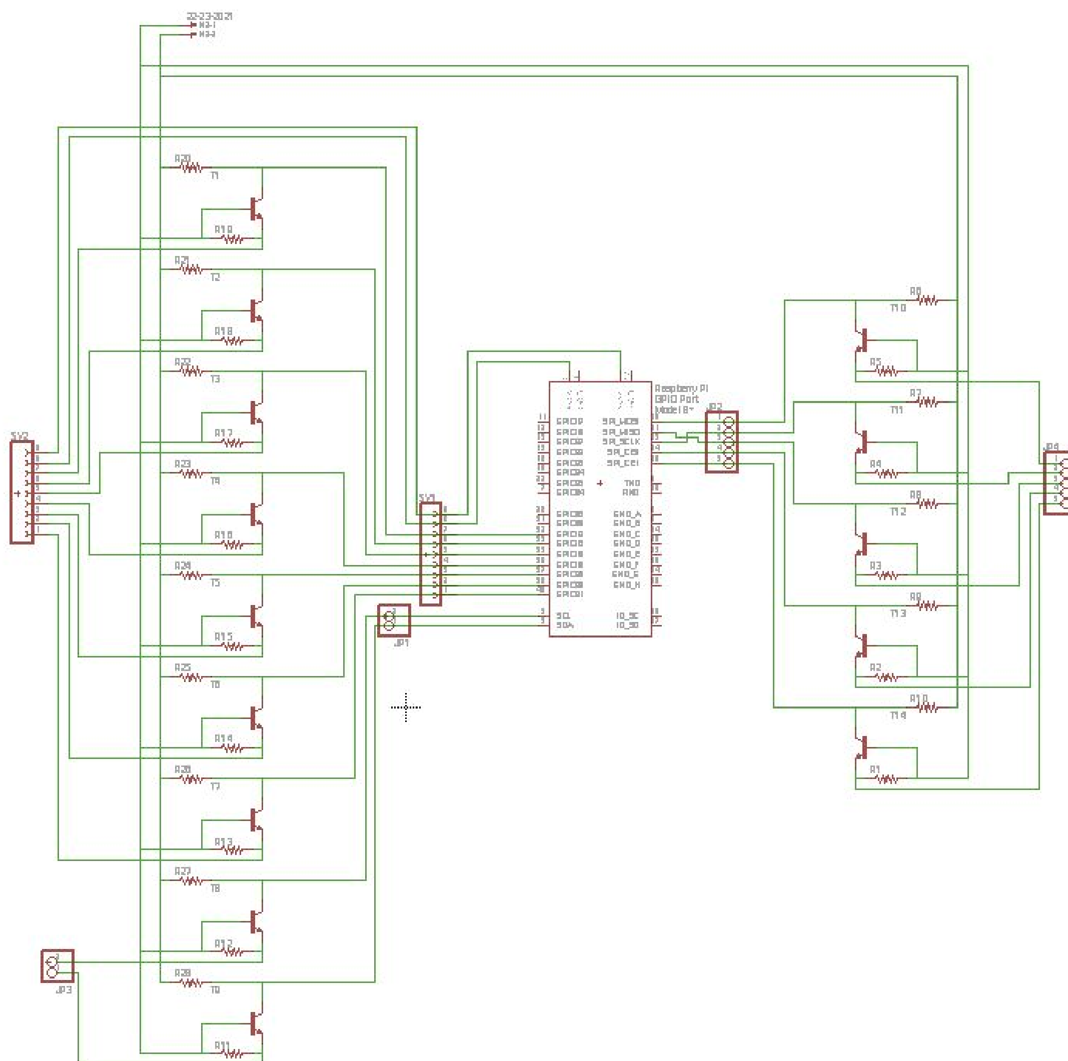


Figure 4 Schematic of Breakout Board

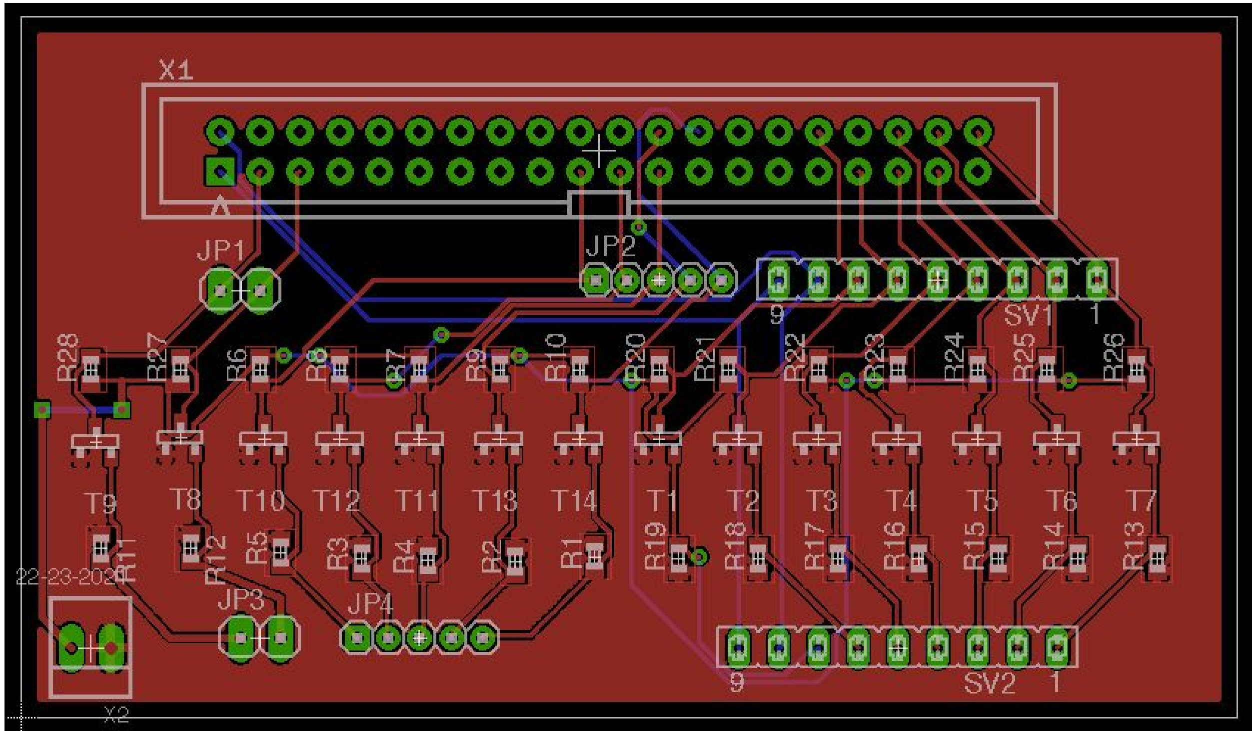


Figure 5 Layout of Breakout Board

4 Testing and Development

4.1 Interface Specifications

GPIB Communication:

- Communication between Raspberry Pi and test equipment requires a GPIB module for older devices only supporting the IEEE-488 bus
- Newer devices communicate using GPIB commands over USB or Ethernet

Device Under Test Communication:

- Breakout board has I2C, SPI and GPIO pins with variable logic level shifting

4.2 Hardware/Software

Web Server with CGI

- Can serve dynamic content (such as current test queue) over HTTP
- Can communicate with test runner process
  - Request information about and results from tests
  - Upload and schedule tests
  - Download test results
- Can send commands over GPIB through testing API or GPIB Console

Test Runner Process

- Executes tests specified by users
- Implements testing API to communicate over the Raspberry Pi's busses and GPIB
- Stores results of tests
- Receives commands from the user interface

4.3 Design Process

The generation of the Kernel, filesystem and other programs for the Raspberry Pi is a tool called Buildroot. Buildroot uses a collection of configuration files and source packages (both existing open source software and our own code) to generate a Linux system image. Once this was done, we were successfully able to load the image onto an SD card and boot it on the Raspberry Pi.

Once the decision to use Buildroot was made, the software framework had to be decided on. A common web server application to use on with Buildroot is lighttpd. This sets up a basic framework on the Raspberry Pi that allows the users to create a website using static HTML files and CGI scripts stored on the device. In order to test the suitability of lighttpd for the scope of our project, we created a simple IO system that could be used to write information into a file stored on the Raspberry Pi and then displayed the information on the website. Because the test scripts will be stored in files, this test results suggest that this process can be expanded upon to allow users to create more complicated tests.

During this time, we also researched the programing languages in order to decide the

one that best fit our needs. The two main contenders were Perl and Python. In the end, we concluded that while Pearl has many features for parsing text that make it more efficient than Python, Python is a more readable and beginner-friendly language that most of the users of this project could easily pick up and debug if necessary.

We then proceeded to research how to implement communication between the Raspberry Pi and test equipment over GPIB. After some research, we discovered the Prologix controller [4] simply opened a virtual serial port on the Raspberry Pi that corresponded to the GPIB connection and could be controlled through simple commands written to it as if it were a file. In order to confirm this, we attached the Raspberry Pi via GPIB to a signal generator in the lab and made a script to change the magnitude, frequency and waveform of the signal generator's output. We found similar results regarding devices connected over USB.

4.4 Testing

4.4.1 Hardware Testing

- The level shifter was initially tested in simulation and constructed on a breadboard to confirm its behavior.
- The breakout board was verified through measurement to ensure it performed as expected.

4.4.2 Software Testing

- Individual features were tested upon implementation.
- Software was systematically tested with actual run-time scenarios
    - Variety of scenarios in order to look for points of failure
    - Scenarios include:
        - Queueing multiple tests
        - Queueing tests while a test is running
        - Sending console commands while a test is running
        - Viewing and downloading test results
    - Intended to mimic actual product use
- Test images were generated from clean builds to prevent possible issues from previous versions of software

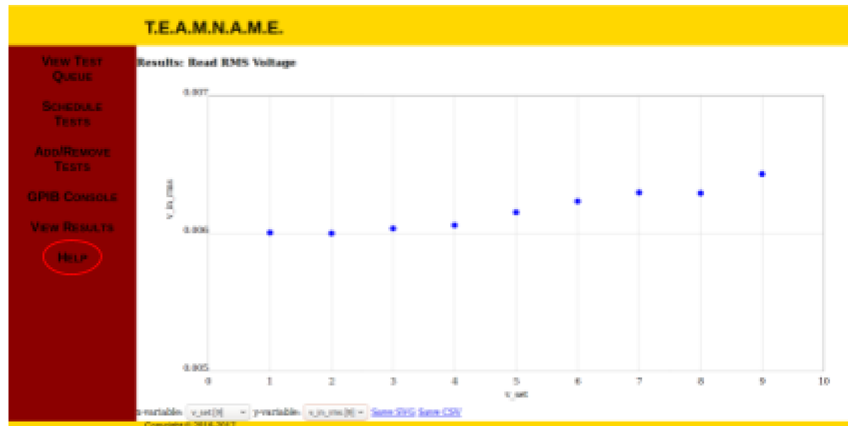
5 Results and Conclusions

We have set out to create an open source platform for hardware test automation and remote test execution and data collection. The platform allows users to access it remotely through any web browser, and allows them to upload and execute custom test scripts as well as to view and download results.

This platform is targeted towards individuals seeking to perform device characterization, implement testing automation, and facilitate remote data collection. The functionality of our platform has a wide array of usage cases from automating hardware verification testing to allowing remote device demonstration for instructive or educational purposes. In addition to providing this functionality, it does so through a cost-effective platform based on open-source software and low-cost hardware.

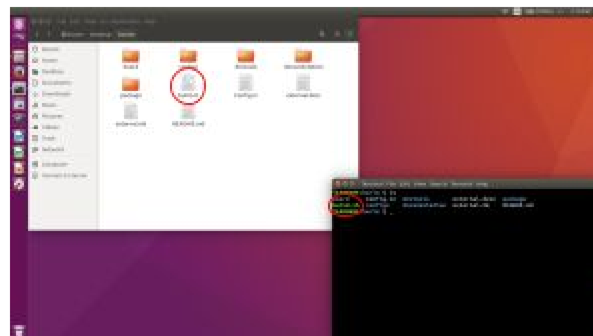
Appendix I Operation Manual

Note: The most up-to-date version of the Operating Manual can be found under “Help” in the main menu on the web interface.



1 Building and Installation

To begin using TEAMNAME to automate testing you must first build the testing environment image. In the root of the repository is the script *build.sh* which when ran produces a directory called *teamname_output* containing the output from Buildroot including the file *teamname_output/output/images/sdcard.img* containing the complete system image.



After generating *sdcard.img*, write it to an SD card through the following steps [10]:

1. Insert a Micro SD card into the computer (note: this may require an adapter)
2. Use `df -h` to view the mounted disks and find the sd card’s location in /dev/ (for example /dev/sdb)
3. Unmount the card using `umount /dev/sdb*` (this unmounts all partitions)
4. Use `dd if=sdcard.img of=/dev/sdb bs=4M` to write the image
    a. This requires root privileges
    b. Be careful that the correct arguments are specified for the if and of arguments when using the dd utility

Below is sample terminal output for building and installing TEAMNAME to an SD card. Note that the card may be named differently depending on your system. Also note that here output from ./build.sh has been suppressed for the sake of demonstration.

```
Terminal File Edit View Search Terminal Help
TEAMNAME:hwrlm $ ls
board     Config.in  devtools       external.desc  package
build.sh  configs    documentation  external.mk    README.md
TEAMNAME:hwrlm $ ./build.sh > /dev/null
TEAMNAME:hwrlm $ cd ../teamname_output/output/images/
TEAMNAME:images $ ls
bcm2710-rpi-3-b.dtb  kernel-marked  rootfs.ext4   sdcard.img
boot.vfat            rootfs.ext2    rpi-firmware  zImage
TEAMNAME:images $ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            1.5G     0  1.5G   0% /dev
tmpfs           294M  8.6M  286M   3% /run
/dev/sda1       291G   17G  259G   7% /
tmpfs           1.5G  256K  1.5G   1% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           1.5G     0  1.5G   0% /sys/fs/cgroup
tmpfs           294M   68K  294M   1% /run/user/1000
/dev/sdb1        32M  7.0M   25M  22% /media/user/26A1-4825
/dev/sdb2        79M   68M  5.7M  93% /media/user/3ebbd418-1995-4e61-a031-86a572b344
61
TEAMNAME:images $ umount /dev/sdb*
umount: /dev/sdb: not mounted
TEAMNAME:images $ sudo dd if=sdcard.img of=/dev/sdb bs=4M
[sudo] password for user:
28+1 records in
28+1 records out
117676544 bytes (118 MB, 112 MiB) copied, 24.3719 s, 4.8 MB/s
TEAMNAME:images $
```

Once the SD card has been written, insert it into the Raspberry Pi. Plug in all desired peripherals and test equipment. Connect it to the local network and power it on.

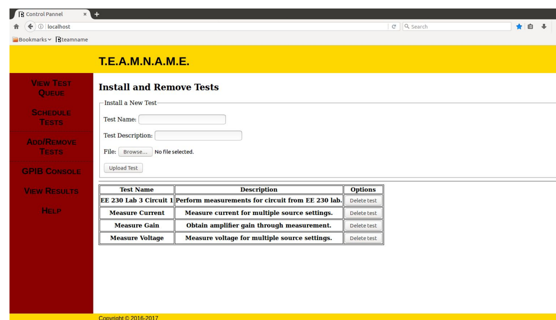
Note: The device will be assigned a random IP address depending on your local network. See the manual for your network hardware or contact your network administrator to obtain its IP address.

2 Creating and Running Tests

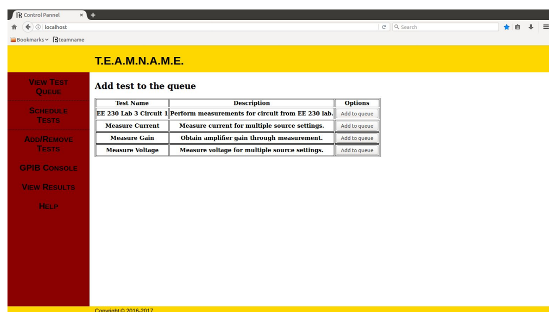
Tests can be created in your text editor of choice. For Python-based tests, all code (including import statements) should be contained in a function called __test(GPIB_Context). It should return a JSON object containing all variables (stored as lists) that you wish to save. An example test provided below sets the voltage on the Agilent 33220A function generator and confirms that the voltage was set. It then reads the AC voltage seen by the Agilent 34410A digital multimeter. It ends by returning the values it stored.

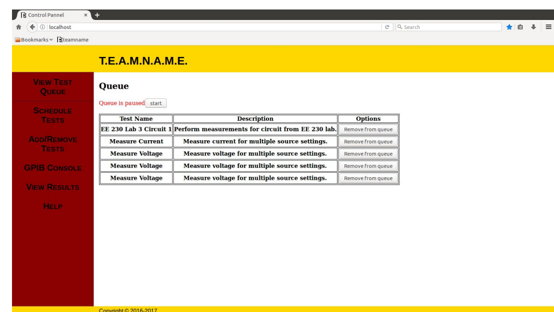
```
def __test(GPIB_Context):
    # Import APIs for devices we are using
    import Agilent33220A        # API for the Function Generator
    import Agilent34401A        # API for the DMM

    import time

    # Allocate storage for data we wish to save
    testVars = {}
    testVars['v_out'] = []
    testVars['v_set'] = []
    testVars['v_in_rms'] = []

    for i in range(1,10):
        testVars['v_out'].append( i )
        Agilent33220A.Agilent33220A_set_voltage(GPIB_Context, i)
        time.sleep(0.5)
        testVars['v_set'].append(float(Agilent33220A.Agilent_33220A_get_voltage(GPIB_Context) ))
        testVars['v_in_rms'].append(float(Agilent34410A.Agilent34410A_measure_ac_voltage(GPIB_Context) ))

    return testVars
```

Once a test is created and saved, it can be uploaded to the device using the “Add/Remove Tests” panel. Choose a name and a provide a description before selecting a file to upload. Then click “Upload test” to install it in TEAMNAME. (Here tests can also be removed with the “Delete test” button.)



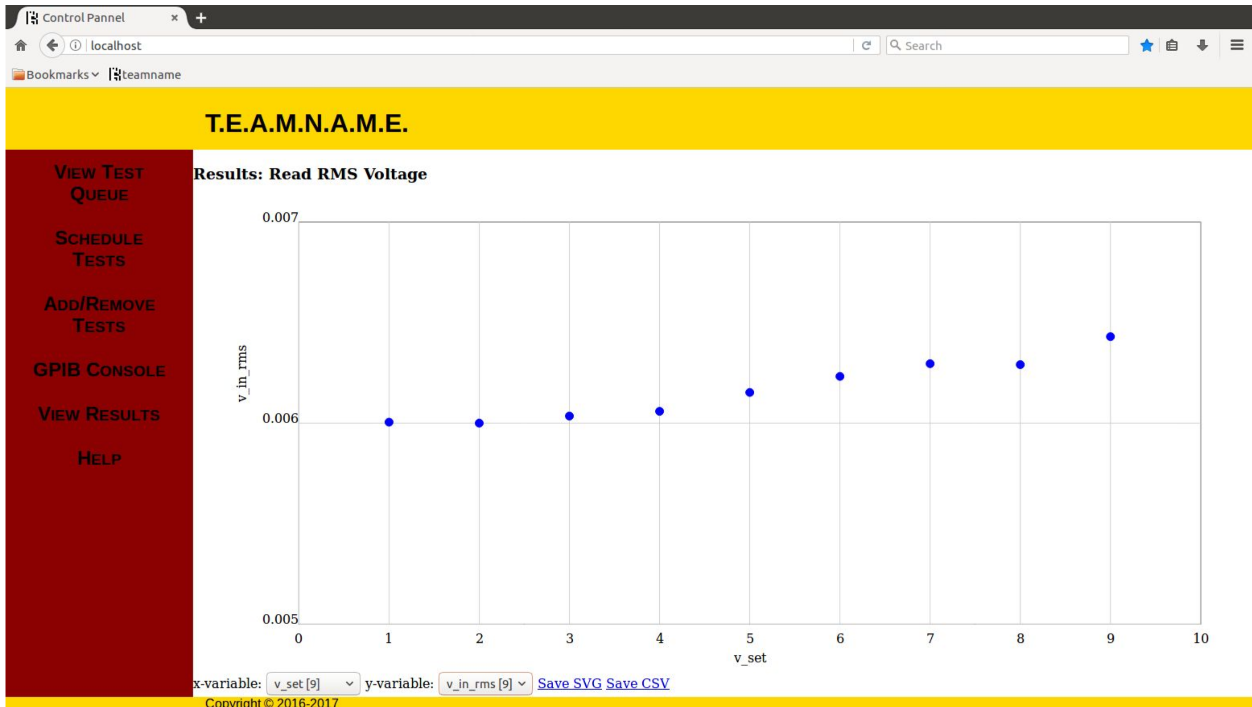
To run a test, navigate to the “Schedule Tests” panel. Select “Add to queue” at add a test to the queue. If the queue is running, the test will begin execution, otherwise navigate to the “View Test Queue” panel and click the “Start queue” button.





3 Viewing the Results

Once a test has completed, TEAMNAME stores the results. To view these results, navigate to the “View Results” panel. Find the entry for our test and select either the “Download JSON” or the “View results” button. The first option allows you to download the complete results including and errors thrown by the test as JSON data. The second option opens an in-browser viewer that can plot variables from the results against each other. It also provides options to download the figure or to download the plotted variables as a CSV spreadsheet.



4 Development Resources

To expand TEAMNAME’s functionality to new test equipment we have made an effort to balance the ability of third parties to build on the available set of supported devices, while allowing the project managers to maintain control over the fundamental structure of the device drivers and server code. In order to allow the program managers to maintain the freedom to change the formatting of the device drivers, which are written in python, as might be required in the future. To achieve this, we set out to refine the process of building out new device driver sets to the most basic information as possible. Because GPIB is an IEEE standard, we feel that by requiring third parties to populate a list of available GPIB commands, or as many as they might require for a project, so that the formatting of the user generated content is unlikely to ever need to change. The API Generation tool works as follows, the developer populates a list of functions that they would like to make available to users writing tests. An example of the formatting for the text file that the developers would write is shown below.

```
 1  Keysight20xx                            #DEVICE_NAME
 2  Keysight 20xx Family of Oscilloscopes   #DEVICE_DESCRIPTION
 3  Oscilloscope                            #DEVICE_TYPE
 4  IEEE-488                                #DEVICE_CONNECTION
 5  Keysight20xx.py                         #TARGET_FILE
 6
 7
 8  #END_HEADER
 9  #NOTES SECTION
10  #
11  #
12  #
13  ##START_DEFINES
14  _clear_status          {clear_device_status}                  *CLS
15  _event_status_enable  Q  {enable_event_status {check_ESE_state}}   *ESE    <mask_argument>
16  ##END
```

A provided perl script “api_gen.pl” will translate the above example into a python file that provides a driver for the Keysight20xx series oscilloscopes. Based on the two lines in the #START_DEFINES section, only two families of functions will be available to users attempting to script test for this device. First the clear device status function will be exposed to test writers as the function “clear_device_status”. When called, the string “*CLS” will be sent to the device. The second line actually creates both a function and a query form of the function signified by the “Q”. The command form of the function is exposed to the user by the alias “enable_event_status”, while the query form is exposed as the alias “check_ESE_state”. Any alias or list of aliases can be entered into the braces to specify the alias/es that can be used by to access a specific action. “<mask_argument>” indicates the name by which the single argument of the command form of the function will be internally referenced. The output of the perl script based on the above command list is shown below.

```python
#!/usr/bin/python
#Driver file for the Keysight20xx device or family of devices
#This Driver file was procedurally generated and may require additional comments and/or
#   documentation in order to meet community standards regarding usability.
# This template assumes the presence of a class with methods for communicating
#   with the GPIB devices. The class here is refered to as GPIB_Context. It is
#   assumed to be the first argument to any function in the API and will let me
#   handle the communication separately. (Just write assuming it will be there.)
#########################
#Function defs go here:
#########################
def Keysight20xx_clear_status(GPIB_Context):
    command = '*CLS'
    GPIB_Context.send(device_data, command)
    return 0

def Keysight20xx_event_status_enable(GPIB_Context):
    command = '*ESE'
    GPIB_Context.send(device_data, command)
    return 0

def Keysight20xx_event_status_enable_query(GPIB_Context):
    command = '*ESE?'
    GPIB_Context.send(device_data, command)
    return GPIB_Context.read(device_data)


device_data = {
        # The device name (cannot contain spaces)
        'name': 'Keysight20xx',
        # The full title of the device
        'description': 'Keysight 20xx Family of Oscilloscopes',
        # 'Signal Generator' | 'Power Supply' | 'DMM' | 'Oscilloscope'
        'Type': 'Oscilloscope',
        # 'IEEE-488' | 'Network' | 'USB'
        'connection': 'IEEE-488',
        # The port the device is on (may be made more specific)
        'port': '10',
        # List of functions available. These name are the ones that will be
        # recognized in the interpreted script.
        'functions': {
            # 'user_visible_alias'  : Keysight20xx_internal_function_name,

            'clear_device_status' : Keysight20xx_clear_status,
            'check_ESE_state' : Keysight20xx_event_status_enable_query,
            'enable_event_status' : Keysight20xx_event_status_enable
        }
```

The template statement that is included in the command list template provided with the project is shown below.

```
//Keysight20xx_example_function    {desired_user alias_functions {query_alias}} :GPIB:COMMand:FOR:FUNCtion <required_input_values> [Optional Arguments {available_inputs_to_required_arg}]
///                              ^  place a 'Q' here to also generate a query version of the function
///                                 use 'QO' for query only
```

Appendix II Alternative Designs

Original Project Scope

This project was originally proposed by IBM to be a proprietary remote hardware testing environment. The setup IBM had would only work on a specific model of computer running a specific firmware. They proposed the idea to us as a server version of this setup that could be accessed from any operating system through a web browser to avoid the struggles they were having with connecting to their current software.

The hardware portion of our project was originally conceived to be a breakout board to communicate between the Raspberry Pi and an embedded microcontroller on a silicon wafer.

Due to numerous strategic and legal hurdles, IBM's legal team eventually pulled their support for IBM's ongoing involvement in the project.

The Operating System

When we began development on the project we chose to use the Raspbian Linux Distribution [6] as our platform's operating system. This seemed a reasonable starting place since our project uses a Raspberry Pi as its hardware platform. As development of the project progressed, however, we realised that Raspbian did not meet all of our project's needs.

When installing our project using Raspbian, we first needed to flash an SD card with the Raspbian image [10] before booting up the Raspberry Pi to install the web server, test runner, and other environment files. Additionally, Raspbian did not include the USBTMC driver necessary to communicate with test and measurement equipment over USB. For these reasons we reimplemented our project as a package with the Buildroot project [1], a build automation tool that allowed us to design and build a complete Linux system image containing all of our project code as well as any external packages we needed for the project to function. This image could then be written to an SD card and would immediately provide a working system on startup.

The Web Server

We began our project using the Apache web server to deliver dynamically-generated content using CGI and Python scripts [9]. When we moved from Raspbian to Buildroot it became convenient to switch to the lighttpd web server [2] due to its simpler setup process for enabling CGI [3] and due to the convenience of Buildroot's out-of-the-box support for it.

Appendix III Citations and Resources

[1] The Buildroot Project: https://buildroot.org/

[2] The Lighttpd Web Server: https://www.lighttpd.net/

[3] CGI on Lighttpd: https://wiki.archlinux.org/index.php/lighttpd

[4] Prologix GPIB-USB Controller: http://prologix.biz/

[5] Linux GPIO Sysfs Interface: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

[6] Raspbian Download: https://www.raspberrypi.org/downloads/raspbian/

[7] Buildroot Documentation: https://buildroot.org/downloads/manual/manual.html

[8] Bi-Directional Level Logic Converter Overview:
https://learn.sparkfun.com/tutorials/bi-directional-logic-level-converter-hookup-guide

[9] Dynamic Content with CGI: https://httpd.apache.org/docs/2.4/howto/cgi.html

[10] Flashing an image to an SD card:
https://www.raspberrypi.org/documentation/installation/installing-images/

Appendix IV Glossary of Terms

DUT: Device under test

GPIB: General Purpose Interface Bus. Also known as IEEE 488, is a short-range communication interface.

GPIO: General Purpose Input and Output. Usually refers to digital communication pins on the Raspberry Pi.