

**KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**COLLEGE OF ENGINEERING**



**DEPARTMENT OF COMPUTER ENGINEERING**

**RECONFIGURABLE VIRTUAL INSTRUMENTATION MIDDLEWARE**

**SUBMITTED FOR FULFILMENT OF THE DEGREE OF MPhil**

**COMPUTER ENGINEERING**

**BY**

**BOAKYE-BOATENG, KWASI**

**(PG2654508)**

**SUPERVISOR:**

**DR. KWAME OSEI BOATENG**

**MAY 2012**

## ABSTRACT

Instruments are integral in our daily lives. From the trader to the engineer, everyone uses instruments to quantify objects or events. Engineering instruments have seen rapid improvement from cathode ray tubes and mechanical inventions to state-of-the-art electronic gizmos. Now there is a new instrument on the block – the Reconfigurable Virtual Instrument (RVI). RVIs make it possible for traditional instruments to be emulated. One moment it can be configured to read voltage and then, within a matter of minutes, it can be reconfigured to read current or even temperature. However, RVIs, in the market so far, are expensive to acquire and maintain especially with the licenses that have to be paid for them periodically. Already, an existing research on creating RVIs using an FPGA has already been achieved. This goes a long way to help developing countries like Ghana with regards to operational expenses on instruments particularly in secondary and tertiary institutions.

The aim of this research is to build upon what has already been done by letting RVIs be accessible by the use of a simple web browser on a PC without the need of special installation. Not only accessible by one PC at a time but by multiple PCs at a go via a distributed system. With the emergence of smart phones that have advanced browsing capabilities, RVIs can also be accessed provided they are connected via a wireless network. This is achieved by using free open-source tools. This research uses an existing RVI research with emphasis on the instruments that were configured on the RVI – the digital frequency meter and the function generator. Tests show promising results with a considerable error margin with regards to the function generator which requires real-time display.

**Keywords:** middleware, interface, distributed systems, application

**DECLARATION**

I hereby declare that, this submission is my own work except for specific references which have been duly acknowledged, this work is the result of my own research and it has not been submitted either in part or whole for any other degree in Kwame Nkrumah University of Science and Technology or any other educational institution elsewhere.

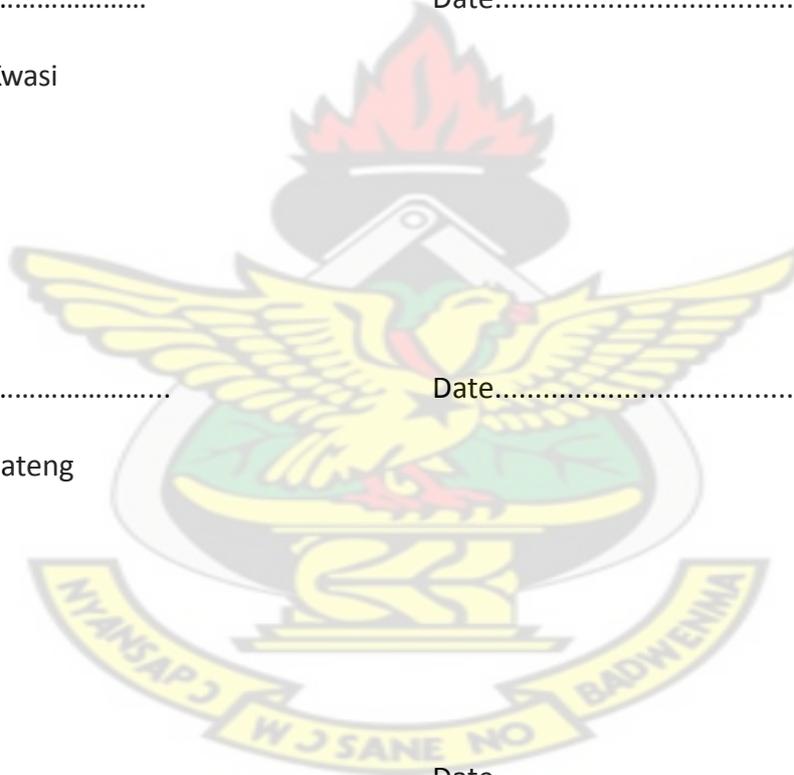
KNUST

Signature.....

Date.....

Boakye-Boateng, Kwasi

(Candidate)



Signature.....

Date.....

Dr. Kwame Osei Boateng

(Supervisor)

Signature.....

Date .....

Dr. Kwame Osei Boateng

(Head, Department of Computer Engineering)

## DEDICATION

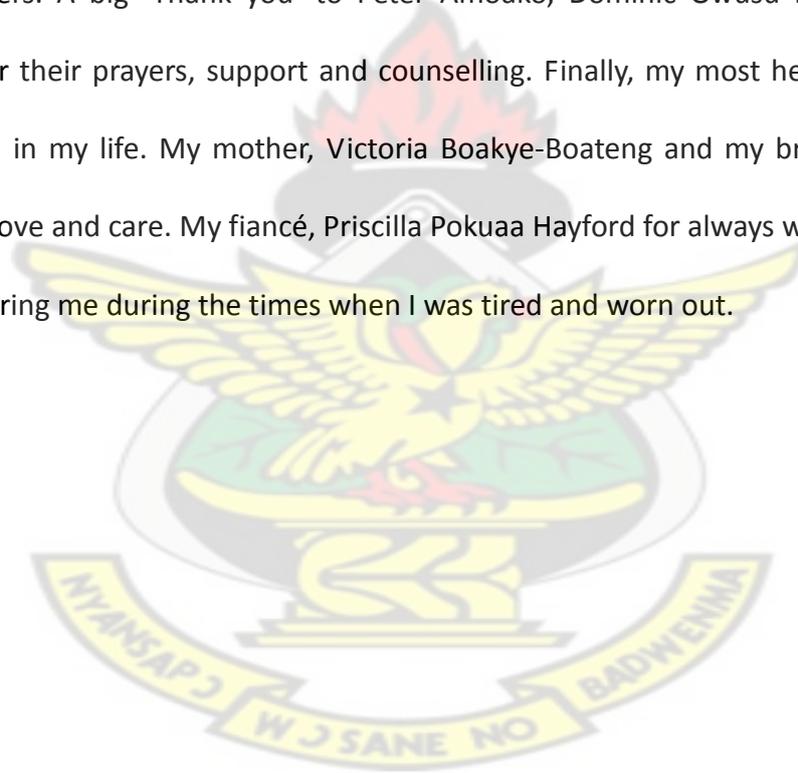
This thesis is dedicated to my father, the late Kwasi Boakye-Boateng. He taught me to never give up in anything I do. He instilled in me the principles of research from the age of 11. His legacy lives on.

# KNUST



## **ACKNOWLEDGEMENT**

I am grateful to God for all that he has done for me. Without him, I would not be who I am. My deepest gratitude to my supervisor, Dr. Kwame Osei Boateng for his unending support and guidance. He sacrificed a lot for me to achieve this feat. I am grateful to Gilbert Osei-Dadzie whose research mine was built upon. He gave me his time, patience and contributions were immeasurable. I am also very grateful to all administrative and academic staffs of faculty of Electrical and Computer Engineering for their support. This work has benefited from valuable discussions with friends and colleagues including Gabriel Agbagba, Kofi Asamoah, Kwaku Aboagye Asare among others. A big 'Thank you' to Peter Amoako, Dominic Owusu Boateng and Kwasi Opoku Mensah for their prayers, support and counselling. Finally, my most heartfelt gratitude to the 'special three' in my life. My mother, Victoria Boakye-Boateng and my brother Kofi Boakye-Boateng for their love and care. My fiancé, Priscilla Pokuaa Hayford for always waking me up late in the night and spurring me during the times when I was tired and worn out.



## TABLE OF CONTENTS

	<b>Page</b>
<b>ABSTRACT</b>	2
<b>DECLARATION</b>	3
<b>DEDICATION</b>	4
<b>ACKNOWLEDGEMENT</b>	5
<b>LIST OF FIGURES</b>	9
<b>CHAPTER ONE – INTRODUCTION</b>	10
1.1 Background to study	10
1.2 Problem definition	11
1.3 Statement of Objectives	12
1.4 Scope of research	13
1.5 Justification of research	13
1.6 Research design/Methodology	14
1.7 Organization	14
<b>CHAPTER TWO – LITERATURE REVIEW</b>	16
2.1 Reconfigurable Virtual Instruments (RVI)	16
2.2 Purpose of Virtual Instruments	18
2.3 Components of Reconfigurable Virtual Instrument	19
2.3.1 Hardware Subsystem	19
2.3.2 Software Subsystem	19
2.4 Challenges Involved in RVI	20
2.5 Existing RVI	20
2.6 What is a distributed system?	21
2.7 What is middleware?	22

<b>CHAPTER THREE: METHODOLOGY AND DESIGN</b>	<b>24</b>
3.1 Methodology	24
3.2 Requirement Specifications	24
3.3 Design	24
3.3.1 Use cases	25
3.3.1.1 User use-case	25
3.3.1.2 Administrator use-case	25
3.3.1.3 System use-case	27
3.3.2 Design structure	28
3.3.2.1 Database Module	28
3.3.2.2 Business Logic (Middleware)	29
3.3.2.3 Web-Based Portal GUI)	29
3.3.3 Sequence diagram	29
3.3.3.1 User sequence diagram	29
3.3.3.2 Administrator sequence diagram	30
<b>CHAPTER FOUR – IMPLEMENTATION AND TESTING</b>	<b>34</b>
4.1 Selected Instruments	34
4.1.1 Digital Frequency Meter	34
4.1.2 Function Generator	34
4.1.3 Instrument Classification	35
4.2 Software Development Tools	35
4.2.1 MySQL	35
4.2.2 Python	37
4.2.3 Django	39
4.2.3.1 Django components	39
4.2.3.2 Features in Django	40
4.2.4 jQuery	41
4.2.4.1 jQuery Features	42
4.2.5 FusionChartsFree (jQuery plugin)	42
4.3 Implementation	43
4.3.1 RVI Architecture	43

4.3.2	Setting Up	44
4.2.3	Database	46
4.2.4	Middleware	47
4.2.5	Graphical User Interface	49
4.4	Debugging and Testing	50
4.5	Results	51

<b>CHAPTER FIVE – CONCLUSION</b>	54
----------------------------------	----

5.1	Conclusion	54
5.2	Future work	54

<b>REFERENCES</b>	56
-------------------	----

<b>GLOSSARY</b>	59
-----------------	----

<b>APPENDIX</b>	60
-----------------	----

KNUST



## LIST OF FIGURES

Figure 1.1: Vendor-defined instruments	11
Figure 2.1: Traditional instruments versus software based virtual instruments	16
Figure 2.2: Reusable hardware and software	18
Figure 2.3: Many applications, one device	18
Figure 2.4: Upgradable hardware system	20
Figure 2.5: Internet	21
Figure 2.6: Distributed system with middleware	22
Figure 3.1 User use-cases	26
Figure 3.2: Administrator use-case	27
Figure 3.3: System use-case	28
Figure 3.4: Structure of software subsystem	28
Figure 3.4: User reading instrument (Sequence diagram)	30
Figure 3.5: Administrator modifying instrument (Sequence diagram)	31
Figure 3.7: Administrator adding instrument (Sequence diagram)	32
Figure 3.8: Administrator deleting instrument (Sequence diagram)	33
Figure 4.1: Example of a python code	38
Figure 4.2: Django powered page	41
Figure 4.3: GUI of Function generator	52
Figure 4.5: GUI of Frequency Meter	53

## **CHAPTER ONE: INTRODUCTION**

### **1.1 Background of Study**

From the creation of time, up till now, measuring instruments have been an important figment in our lives. Everyone benefits when correct measurement is applied in many situations. For example:

- Customers benefit by receiving the exact amount of goods they ordered and paid.
- Traders benefit through accurate stock control
- Ghana (but not limited to) benefits through consumer confidence in a system which delivers consistency and reliability.

Engineers use a vast range of measuring instruments to perform their measurements. These range from simple objects such as rulers and stopwatches to electron microscopes and particle accelerators.

The transformation of instrumentation from mechanical pneumatic transmitters, [3] controllers, and valves to electronic instruments reduced maintenance costs as electronic instruments were more dependable than mechanical instruments. This also increases efficiency and production due to their increase in accuracy [1] [2]. In this modern era, the introduction of Reconfigurable Virtual Instrument has helped reduce the costs significantly due to its versatility. An instrument that can be configured into any instrument helps greatly for those with low purchasing power. Unfortunately, RVIs that have been created so far are expensive and hard to acquire in a country like Ghana for our institutions.

However, research involving RVIs using existing technology such as FPGAs, DACs, ADCs, etc has been carried out [9] [11]. However the interface to take readings is not user-friendly. This research

is to create a middleware to provide services, via a distributed system, that link RVIs to human-machine interfaces (accessed via web browser) designed for them by using free open-source tools. This work was pursued using an RVI system having the design of the digital frequency meter and the function generator [11].

## 1.2 Problem definition

Even though costs involving instruments may have been reduced, further reduction is still necessary. This is because these instruments are vendor-defined. A vendor-defined or traditional instrument (Figure 1.1) is used for a particular type of measurement but cannot be used as a replacement for another type of measurement involving different instrument. For example, a watch (used to measure time) cannot be used to measure current (which is measured by the ammeter).



Figure 1.1 Vendor-defined instruments

A vendor-defined instrument provides an engineer with all software and measurement circuitry packaged into a product with a finite list of fixed-functionality using the instrument front panel.

Thus there is no flexibility. Hence the term vendor-defined is used to describe these instruments [4]. In short, if the instrument acquired cannot measure the object or event in question, buy the instrument that can.

For a country like Ghana, being a third-world country, it is very difficult for institutions to acquire various instruments of different ranges and sizes. As time goes on, new technologies come into play with different instruments measuring even the minutest of objects. These require a lot of money from institutions to acquire them so as to be abreast with science. This is where RVIs come into play.

These devices can be reconfigured into different instruments as per the requirements of the engineer. At one moment it can be configured to be an ammeter, in another moment, it can be reconfigured into a voltmeter.

### **1.3 Statement of Objectives**

The main objective of this research is to develop a distributed system for a network of RVIs and their human-machine interfaces. To achieve this objective, specific objectives have been set. These are:

1. Study a selected set of RVIs
2. To design a middleware to provide services via a distributed system that link RVIs to human-machine interfaces designed for them.
3. To study free open source tools (programming languages and other software)
4. To implement the middleware for the selected set of RVIs

#### **1.4 Scope of Research**

The scope of the research will be focused on creating the interface of the RVI via distributed system with the use of open source tools. With a myriad of open source tools out there, a few will be shortlisted that will fit the objective and a final selection of the instruments made. This research also focuses on institutions that use or are beneficiaries of distributed systems and also use engineering measuring instruments, such as Kwame Nkrumah University of Science and Technology (KNUST).

#### **1.5 Justification for Research**

RVIs have been developed and perfected by multinational corporations such as National Instruments Inc (NI) [5]. RVIs of NI are strictly manufactured by their engineers and are provided with the software required to provide the human-machine interfacing over a distributed system. These tools are commercially available but expensive to institutions in Developing countries. Payments must be made for licenses of such products. This situation does not sit well with the goal of provision of affordable instruments.

For institutions like universities, senior and junior high schools, the amount of money to be paid for such licenses will not be that feasible. The aim of the existing research made in Ghana was to build an RVI system using existing hardware components such as Field Programmable Gate Arrays (FPGA), sensors, analog-to-digital convertors, etc [11]. As much as this objective has been achieved, interfacing it over the distributed network remains an open issue. One can argue that using LabView can be used but then again an issue of licensing appears again. The questions remains as to how interfacing RVIs, that have been created using existing hardware components, over a distributed system using FREE open-source software. The aim of this research is to answer that question and such that the RVIs can be viewed on a PC via a web browser.

## 1.6 Research Methodology

The research will be done in three phases.

Phase 1 involves a review of literature relevant to this study. It will also involve identifying the initial set of which will be used in this research. The next step in the phase will be to design a middleware for distributed RVI system. The last part is to select tools and learn how to use them for the project

Phase 2 involves implementing the middleware for the identified set of RVIs and creating graphical user interface for the identified RVIs.

Phase 3 will be to test and evaluate the results with existing instruments.

## 1.7 Organisation

*Chapter 1:* This chapter provides an introduction to the write up. A background study is given on measuring instruments, its emerging technologies and RVI. Problem definition, research motivation, methodology, scope of research, organization, and thesis objectives are highlighted.

*Chapter 2:* This chapter presents all the necessary background information. The introduction of RVI, its advantages and challenges, and its components will be looked at. Distributed systems and middleware will also be discussed also.

*Chapter 3:* This chapter involves the methodology and design to achieve the objectives of this research. The uses-case, collaborative diagrams and so on are all discussed within this chapter.

*Chapter 4:* In this chapter, the implementation and testing of the RVI will be delved into. The tools

that were used and how they were used will be discussed.

*Chapter 5:* In the concluding part of the thesis, a summary of results and its analysis is given.

Challenges encountered with appropriate recommendations will also be presented.

# KNUST



## CHAPTER TWO: LITERATURE REVIEW

### 2.1 Reconfigurable Virtual Instruments (RVI)

To understand what reconfigurable virtual instruments are, two definitions are given for reconfigurable device and virtual instrumentation.

A **reconfigurable device** is described as a versatile hardware device that can be configured into different electronic devices using a software tool.

**Virtual instrumentation** is described as a software and hardware combination that allows the emulation of an instrument through a custom console or Graphical User Interface (GUI) [4].

Thus a reconfigurable virtual instrument is a versatile hardware instrument combined with software to emulate real-life instruments through a custom console. The primary difference between hardware (traditional) instrumentation and virtual instrumentation is that software is used to replace a large amount of hardware (See Figure 2.1).

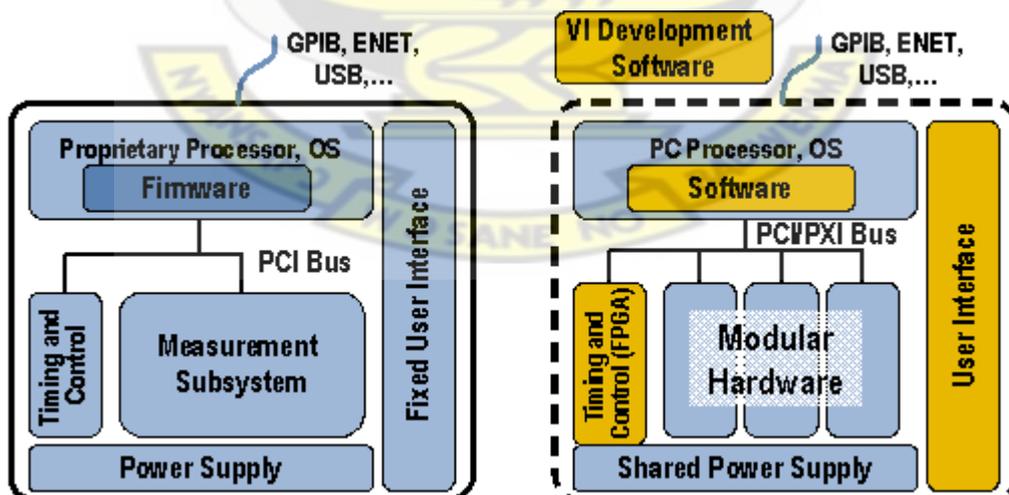


Figure 2.1: Traditional instruments (left) versus software based virtual instruments (right)(Image courtesy of National Instruments)

The software enables complex and expensive hardware to be replaced by already purchased

computer hardware; e.g. analog-to-digital converter can act as a hardware complement of a virtual oscilloscope, a potentiostat enables frequency response acquisition and analysis in electrochemical impedance spectroscopy with virtual instrumentation.

One must not confuse synthetic instruments with virtual instruments. A synthetic instrument is a kind of virtual instrument that is purely software defined. A synthetic instrument performs a specific synthesis, analysis, or measurement function on completely generic, measurement agnostic hardware [4]. Virtual instruments can still have measurement specific hardware, and tend to emphasize modular hardware approaches that facilitate this specificity. Hardware supporting synthetic instruments is by definition not specific to the measurement, nor is it necessarily (or usually) modular.

Leveraging commercially available technologies, such as the PC and the analog-to-digital converter, virtual instrumentation has grown significantly since its inception in the late 1970s. Additionally, software packages like National Instruments' LabVIEW and other graphical programming languages helped grow adoption by making it easier for non-programmers to develop systems.

RVIs typically have a sticker price comparable to and many times less than a similar traditional instrument for the current measurement task. However, the savings compound over time, because RVIs are much more flexible when changing measurement tasks. By not using vendor-defined, pre-packaged software and hardware, engineers get maximum user-defined flexibility. An RVI provides all the software and hardware needed to accomplish the measurement or control task. In addition, with an RVI, engineers can customize the acquisition, analysis, storage, sharing, and presentation functionality using productive, powerful software [5].

## 2.2 Purpose of Virtual Instruments

1. Low-cost reusable common hardware/software platform for the emulation and evaluation of multiple electronic and scientific instrumentation systems (Figure 2.2).

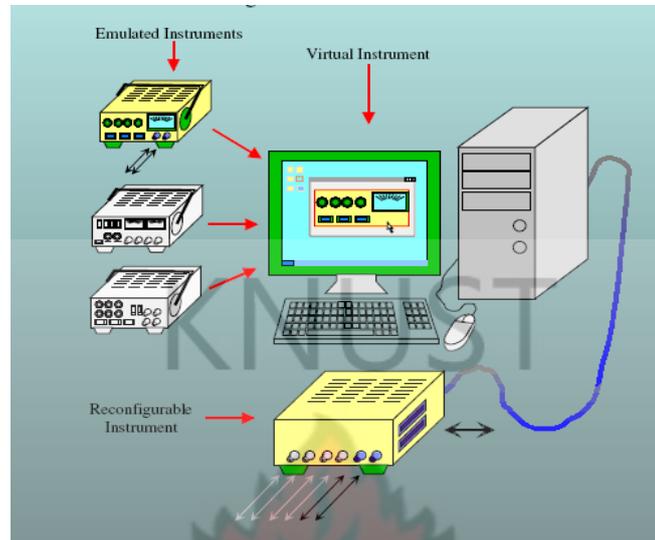


Figure 2.2: Reusable hardware and software

2. Imagine that an engineer uses an RVI to measure the temperature of a certain room and later wants to measure the voltage of a certain device. All that the engineer has to do is to reconfigure the device to measure voltage. Many measurement tasks, one device (Figure 2.3).

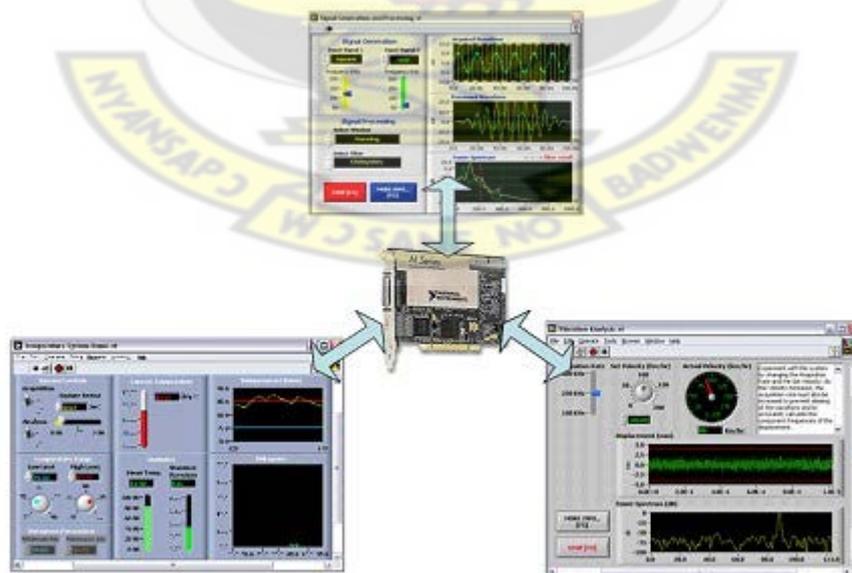


Figure 2.3: Many applications, one device

## 2.3 Components of Reconfigurable Virtual Instrument

RVI system comprises two parts - Hardware Subsystem and Software Subsystem. The RVI can be considered as a “magic box” connected to a PC through a standard port. A virtual instrument is chosen from a library of instruments on the PC and the RVI system is configured to convert it to the selected instrument with its associated console or GUI [6].

### 2.3.1 Hardware Subsystem

- RVI main board

It comprises an FPGA device, extended memory, a block of communication ports, board-to-board connectors and debugging facilities.

- Daughter boards

They are used to assist the main board to connect to the outside world. They can be high performance or low performance.

### 2.3.2 Software Subsystem

- The Computer Software

This software is related to the PC. This contains the GUI, port management programs and data elaboration programs, library of virtual instruments and their custom interfaces, data storage facilities and physical control communication (drivers).

- Synthesizable Hardware Description Code (SHDC)

This is the code corresponding to the FPGA of the RVI. It is responsible for the management of the physical connection with the PC, ADCs and DACs operations, data generation and acquisition, real-

time on-line data processing, and on-board real-time data handling.

## 2.4 Challenges Involved in RVI

Suitable platform must be chosen for the system to work. The right FPGA must be chosen to be the “magic box”. Not just any FPGA device can be chosen.

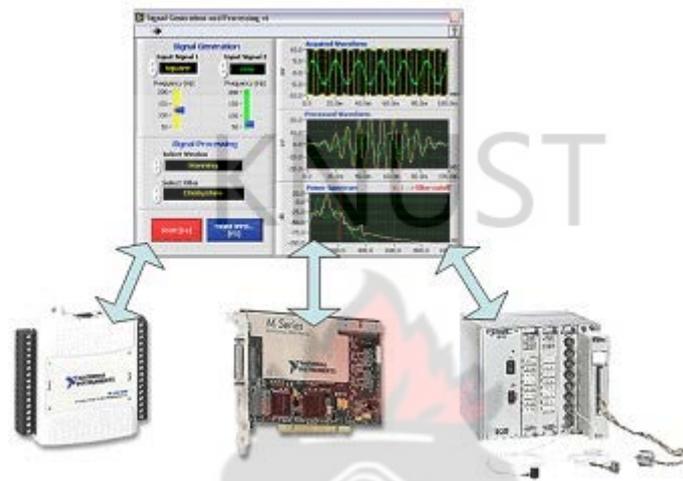


Figure 2.4: Upgradable hardware system

The hardware subsystem must be flexible and must adapt to a wide variety of requirements. The hardware subsystem must also be upgradable to take advantage of new electronics devices and facilities (Figure 2.4). The software subsystem must be portable and adaptable. It must be portable in the sense that it must work on different kinds of operating systems. Adaptable in the sense that it must be able to work on different FPGA vendors and families and use different forms of connectivity be it parallel port, USB, serial port, etc. The software subsystem must also be able to allow the expansion of users or developers base. Like the hardware subsystem, it must also be upgradable, that is, be able to migrate to a new version.

## 2.5 Existing RVI

National instruments already have RVIs created [5]. Their hardware and software that is already in use are proprietary. Existing research made to RVI involved the use of FPGAs to achieve this goal

[6] [9] [11]. The research focuses more on the hardware subsystem and the SHDC of the software system. The computer software portion to interface the RVI was not included in the research. The primary focus of this research is to look at that computer software portion of the RVI on a distributed system.

## 2.6 What is a distributed system?

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility. An easier-to-understand definition is a quote by Leslie Lamport: “A distributed system is one in which the failure of a machine I’ve never heard of can prevent me from doing my work” [7]. Examples of distributed systems are the internet (Figure 2.5), LAN and mobile phone networks.

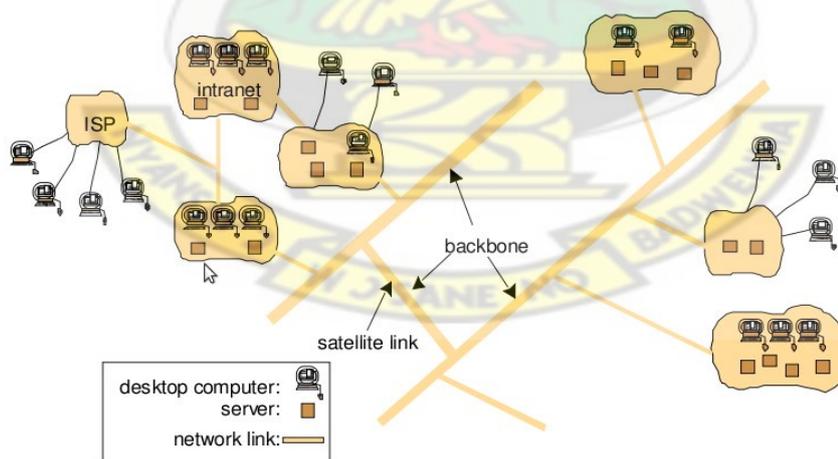


Figure 2.5: Internet

## 2.7 What is middleware?

There are so many definitions for middleware. Popularly, a middleware is described as “the '/' (slash) in client/server”. ObjectWeb® defines middleware as: "The software layer that lies between the operating system and applications on each side of a distributed computing system in a network." Middleware connects software components or applications. Middleware is sometimes informally called “plumbing” because it connects parts of a distributed application with data pipes and then passes data between them[7][8].

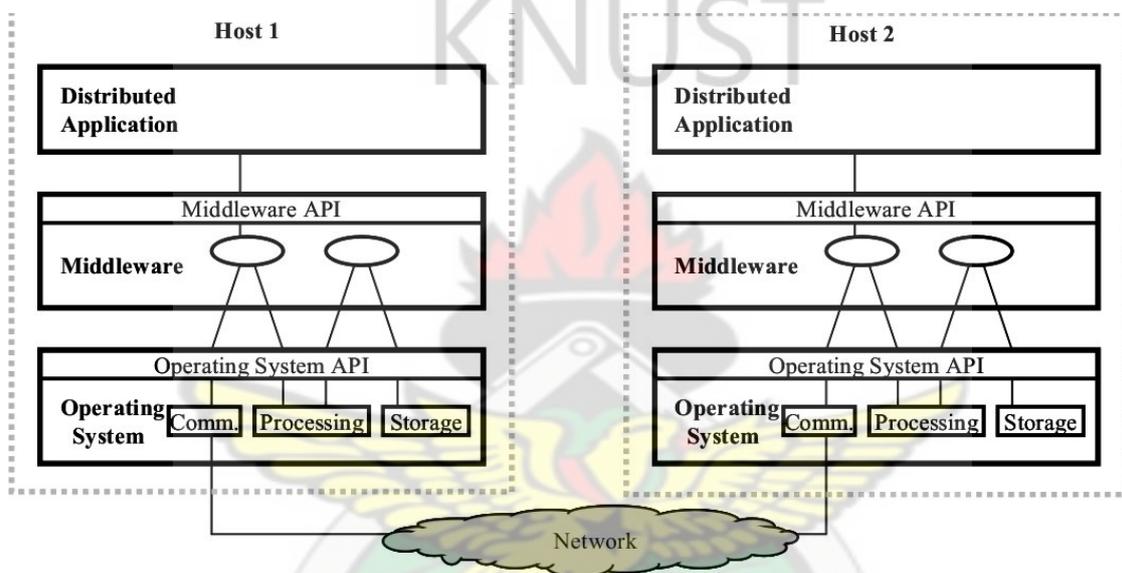


Figure 2.6: Distributed system with middleware

It consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware can also consist of a library of functions, and enables a number of applications to page these functions from the common library rather than re-create them for each application. This technology evolved to provide for interoperability in support of the move to coherent distributed architectures, which are used most often to support and simplify complex, distributed applications. It includes web servers, application servers, and similar tools that support application development and delivery. Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architecture.

It sits "in the middle" (Figure 2.6) between application software working on different operating systems. It is similar to the middle layer of three-tier single system architecture, except that it is stretched across multiple systems or applications. Examples include database systems, telecommunications software, transaction monitors, and messaging-and-queuing software. The distinction between operating system and middleware functionality is, to some extent, arbitrary.

While core kernel functionality can only be provided by the operating system itself, some functionality previously provided by separately sold middleware is now integrated in operating systems. A typical example is the TCP/IP stack for telecommunications, nowadays included in virtually every operating system. There are many types of middleware. The following classifications listed below are the common ones [7] [8]:

- Remote Procedure Call (RPC) – Client makes calls to procedures running on remote systems. Can be asynchronous or synchronous.
- Message Oriented Middleware – Messages sent to the client are collected and stored until they are acted upon, while the client continues with other processing.
- Distributed Object Middleware – This middleware makes it possible for applications to send objects and request services in an object oriented system.
- Distributed Tuples – This is a middleware that sits between applications and database servers.

## **CHAPTER THREE: METHODOLOGY AND DESIGN**

### **3.1 Methodology**

The methodology used in this research was based on the waterfall model. This is because the steps involved in the methodology are straight-forward and simple. The following steps were implemented - Requirements specification, Design, Coding (programming), Testing and Validation, and Installation. Requirements specification and Design will be handled in this chapter. The rest will be handled in the following chapter.

### **3.2 Requirement Specifications**

These requirement specifications describe the system's functions, interface, performance, data, security, etc as expected by the user. From these requirements, will the use-cases, design and sequence diagrams be structured. Below are the requirements:

- The middleware should be accessible across all OS platforms.
- The GUI should be accessible via any web browser.
- The RVI should be accessed via a distributed system.
- The middleware should allow selection of selected instances of instruments loaded on the RVI.
- The middleware should have a library of functions for the RVI.
- The middleware should allow the entry of a new instrument.
- Instrument entry should be allowed only by the administrator.

### **3.3 Design**

This section gives the architecture of the application/middleware. This is achieved by first identifying the use cases. From the use cases will the design be structured and from there the

sequence of operations (sequence diagram) that will happen with the middleware be also identified.

### **3.3.1 Use cases**

Use cases are used to design how the application would be used by the user. Use cases are usually structured with the assumption that the ideal application has been created and ready for use and from this assumption we identify how it will be used by the user. From that other users as well might be structured from this as they might also have needs that might be mutually exclusive from each other. From this, the list of actions to be undertaken by this user will be used to structure how the whole middleware/application will be designed. Three use cases have been identified - User, Administrator and System (middleware).

#### **3.3.1.1 User use-case**

Below are the diagram (Figure 3.2) and the list of actions performed by the user.

1. User opens application via web browser
2. User selects instrument to use.
3. User takes readings from GUI.
4. User closes application.

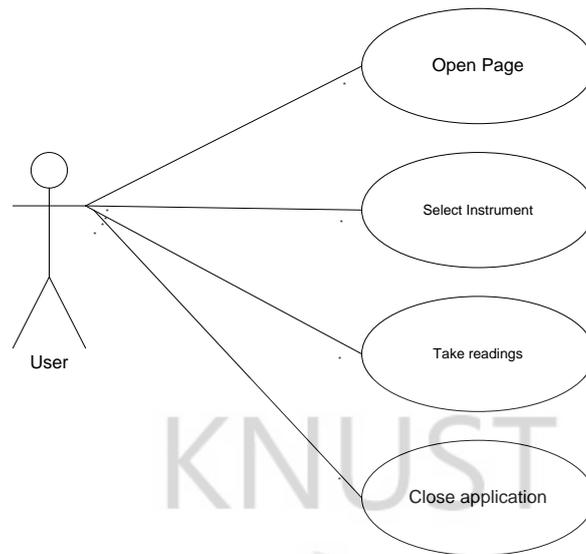


Figure 3.1 User use-cases

### 3.3.1.2 Administrator use-case

Below are the diagram (Figure 3.3) and the list of actions performed by the administrator. It is important that the administrator can perform the same actions as the user with additional actions as well. The additional actions are what are shown.

1. Administrator logs into the system via web browser
2. Administrator adds new instrument function(s) and/or GUI(s)
3. Administrator modifies existing instrument function(s) and/or GUI(s)
4. Administrator logs out.
5. Administrator closes application

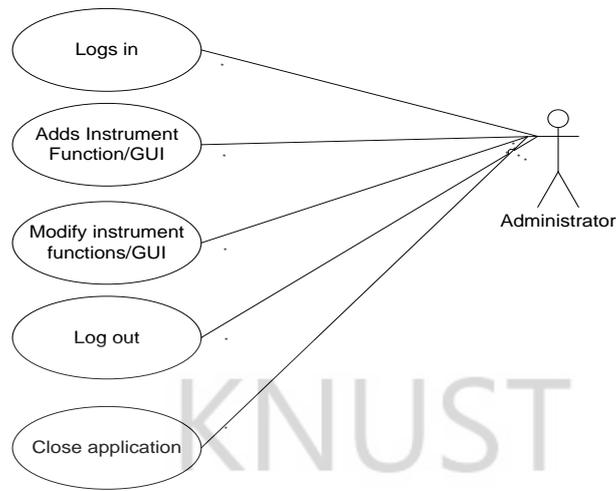


Figure 3.2: Administrator use-case

### 3.3.1.3 System use-case

Even though the System may not be considered as a 'human', it is considered as a user as well since there are certain actions that have to be taken when it is interacted with by the Administrator and the User. Below are the list of actions and the diagram for the System.

1. System checks database to verify if user exists and grants access.
2. System searches the database to find the instrument and loads functions.
3. System retrieves values from RVI and passes it to functions for appropriate decoding.
4. System loads GUI and passes decoded values from function to it.

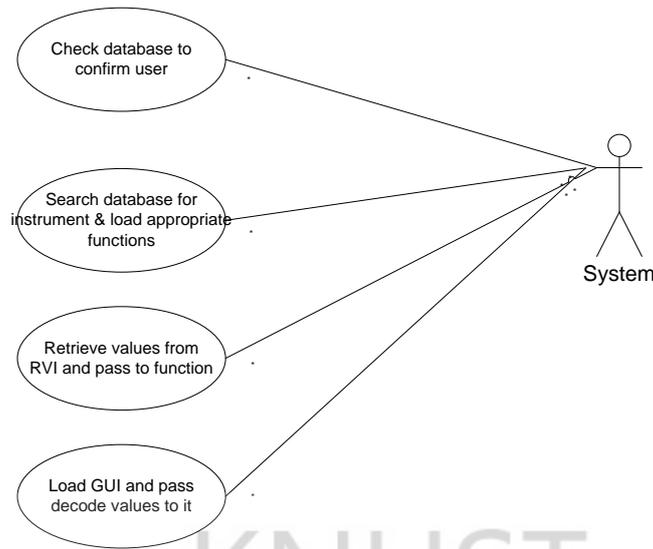


Figure 3.3: System use-case

### 3.3.2 Design structure

From the Use Cases, the application has been classified into the following three modules – Database, Business Logic and Web-based Portal (GUI).



Figure 3.4: Structure of software subsystem

#### 3.3.2.1 Database Module

This module is to hold information about virtual instrument (VI). The database will be used to store records of all virtual instruments configured or added to the RVI subsystem. Information stored about the VI includes Name, Description of VI, and Address Locations/Registers.

### **3.3.2.2 Business Logic (Middleware)**

The middleware for this project is to house all the functions for accessing the database and connection to the FPGA device. It is, also, to serve requests sent to and from the GUI (Figure 3.1).

### **3.3.2.3 Web-Based Portal GUI)**

This is the frontend of the software. It is here that the user will perform visual interactions with the VIs implemented on the FPGA. It is from here that the appropriate interface to mimic “real” instrument is displayed.

### **3.3.3 Sequence diagrams**

This section shows the sequence of events that occur when the application is in use. The sequences are generated by combining the Use Cases with the Design. The sequence diagrams were put into two groups namely User and Administrator sequence diagrams. Under each group, there may be one or more diagrams.

#### **3.3.3.1 User sequence diagram**

This group has only one sequence which involves the User taking an RVI reading. This sequence involves four elements. These are the User, Middleware, Database and RVI (See Figure 3.4). The sequence begins with the User opening the web page and the home page is displayed by the GUI. The User then selects an instrument which then triggers an interface request to the Middleware by the GUI. The Middleware queries the Database for the instrument details and upon retrieving the details, the Middleware loads the RVI with the required instrument. The Middleware then retrieves the raw values from the RVI and loads functions responsible for translating the raw values to meaningful information. Upon generating the right information, the Middleware returns the interface for the GUI to display to the User. The User then takes the readings from the GUI.

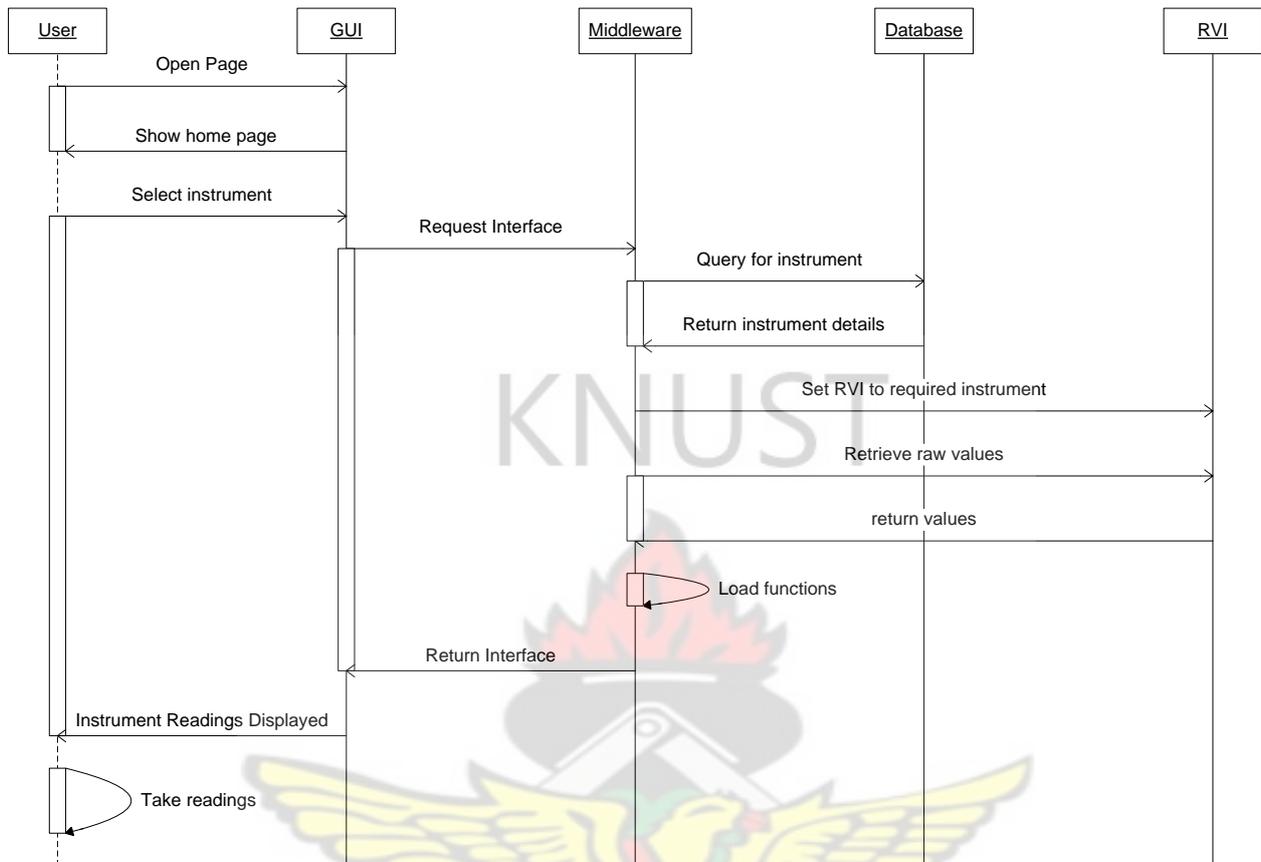


Figure 3.5: User reading instrument (Sequence diagram)

### 3.3.3.2 Administrator sequence diagram

Though it will not be included, a scenario with the administrator taking a reading is similar to that of a User sequence. The Administrator sequence has three diagrams. The first diagram is the administrator modifying an instrument (Figure 3.6), second diagram has the administrator adding a new instrument (Figure 3.7) and the last has the instrument being deleted (Figure 3.8). All three diagrams have four elements in common namely the Administrator, GUI, Middleware and Database. Also all three diagrams have the login sequence in common. The sequence begins with the Administrator opening the web page and the home page is displayed by the GUI. The Administrator then enters log-in credentials which are sent to the Middleware for verification. The

Middleware then queries the Database for credentials and upon verification grants access to the Administrator by informing the GUI to display the admin page. All three diagrams have the stated sequence in common but have different sequence after that.

For modifying an instrument (Figure 3.6), after logging in, the Administrator modifies details of an instrument and the GUI sends the modification request to the Middleware. The Middleware commits the changes to the Database and upon receiving confirmation from the database, the Middleware informs the GUI to give confirmation to the Administrator. The Administrator modifies the instrument's functions directly on the middleware via coding and saves it. The Administrator does likewise to the GUI.

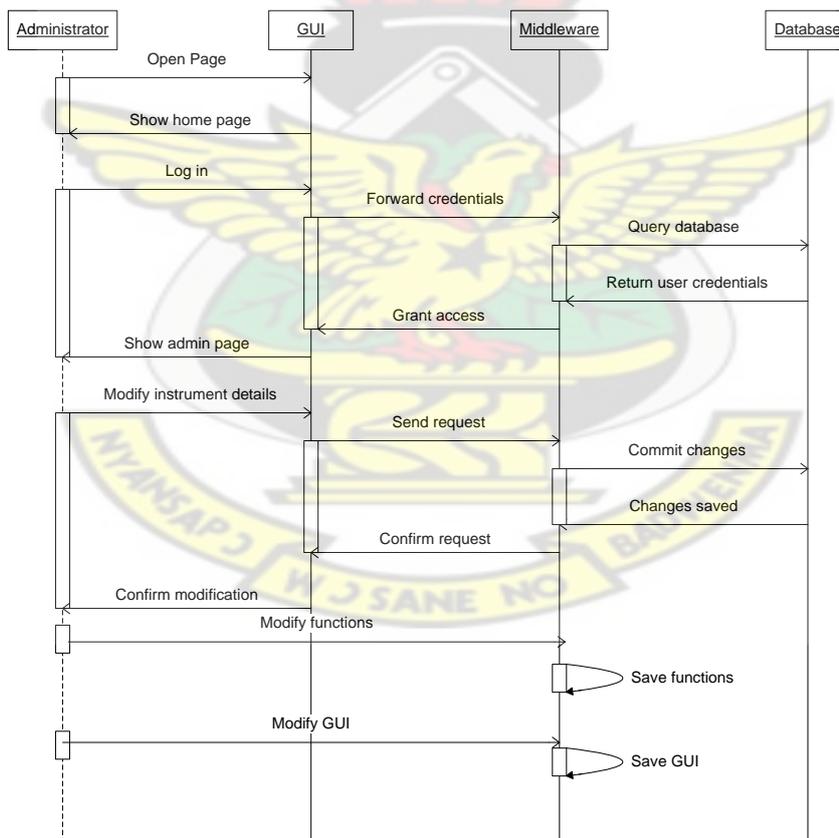


Figure 3.6: Administrator modifying instrument (Sequence diagram)

For adding an instrument (Figure 3.7), after logging in, the Administrator adds new details of an instrument and the GUI sends the addition request to the Middleware. The Middleware commits the addition to the Database and upon receiving confirmation from the database, the Middleware informs the GUI to give confirmation to the Administrator. The Administrator adds the instrument's functions directly on the middleware via coding and saves it. The Administrator does the same thing as well to the GUI.

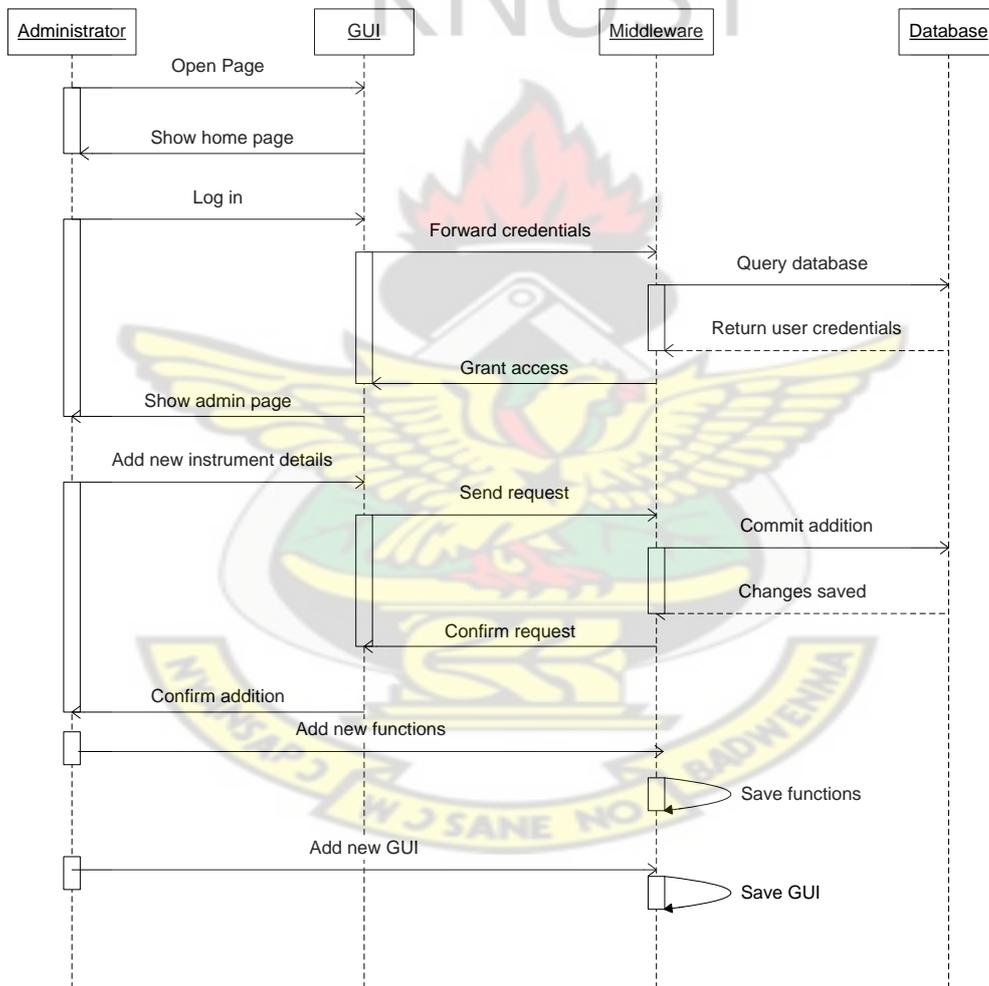


Figure 3.7: Administrator adding instrument (Sequence diagram)

For deleting an instrument (Figure 3.8), after logging in, the Administrator deletes an instrument and the GUI sends the deletion request to the Middleware. The Middleware commits the deletion

to the Database and upon receiving confirmation from the database, the Middleware informs the GUI to give confirmation to the Administrator. The Administrator deletes the instrument's functions directly on the middleware via coding. The Administrator does likewise to the GUI.

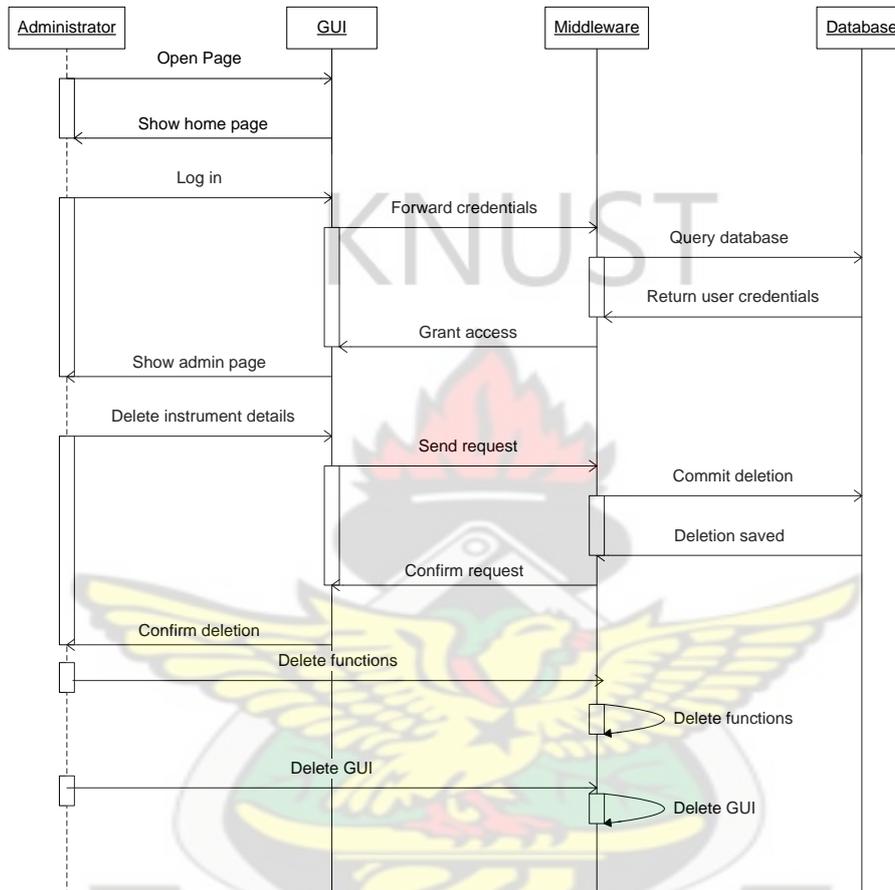


Figure 3.8: Administrator deleting instrument (Sequence diagram)

## **CHAPTER FOUR: IMPLEMENTATION AND TESTING**

This chapter talks about how the middleware is implemented. Before implementation, the instruments based on which the middleware is being designed will be discussed. The software development tools that were used will be discussed next before the implementation itself.

### **4.1 Selected Instruments**

Implementation of the middleware will be created based on a selected RVI configured for Digital Frequency Meter and Function Generator. How these instruments work must be understood before their behaviour and interfaces can be mimicked.

#### **4.1.1 Digital Frequency Meter**

DFMs are widely used items of test equipment within the electronics industry for measuring the frequency of repetitive signals and measuring the elapsed time between events. In particular, DFMs are used for radio frequency (RF) measurements where it is important to test or measure the precise frequency of a particular signal. DFMs are more commonly found as general purpose laboratory test instruments. DFMs operate by counting events within a set period or discovering what a period is by counting a number of precisely timed events. The time periods within which events are counted, or the precisely timed events can be generated using a highly stable quartz crystal oscillator.

#### **4.1.2 Function Generator**

A function generator is a piece of electronic test equipment used to generate different types of electrical waveforms over a wide range of frequencies. Some of the most common waveforms produced by the function generator are the sine, square, triangular and sawtooth shapes. These

waveforms can be either repetitive or single-shot (which requires an internal or external trigger source). Integrated circuits used to generate waveforms may also be described as function generator ICs.

#### **4.1.3 Instrument Classification**

These two instruments can be classified into two types of instruments namely instruments that give discrete values (discrete-value-generating instruments) and instruments that generate graphs (graph-generating instruments). More examples of discrete value generating instruments examples are thermometers, ammeters and voltmeters. Oscilloscope and seismographs are also more examples of graph-generating instruments.

#### **4.2 Software Development Tools**

The software tools to be used for interfacing the RVI system are open-source tools. A selection of tools was considered are based on the design of the application and were scrutinised before a final selection was made. For the database module, PostgreSQL and MySQL because access their databases are available in all major programming languages with language-specific APIs. For the GUI model, FusionChartsFree, FusionChartsFree JQuery plugin, Open Flash Chart and Flot were chosen due to their interactive visual effects. For the middleware module, Django, Apache AXIS for C++ and Apache Axis for Java were chosen for their robustness.

It is based on the middleware module that the programming language was determined. Django uses Python as the others are self-explanatory based on their names. Django was chosen based on its ease of installation and also how easy it was to study Python with a short frame of time. From Django, MySQL was chosen because it could interface easily with Django due to a third party

module created allows Python code to interface with MySQL. FusionCharts Free was chosen because of its ease of installation and usage. The chosen tools are MySQL, Python, Django, jQuery and Fusion Charts with a brief description about them given.

#### 4.2.1 MySQL

MySQL [12] [17] [18] is a relational database management system (RDBMS) which has more than 11 million installations. The program runs as a server providing multi-user access to a number of databases. The project's source code is available under terms of the GNU General Public License, as well as under a variety of proprietary agreements.

MySQL is popular for web applications and acts as the database component of the LAMP, BAMP, MAMP, and WAMP platforms (Linux/BSD/Mac/Windows-Apache-MySQL-PHP/Perl/Python), and for open-source bug tracking tools like Bugzilla. Its popularity for use with web applications is closely tied to the popularity of PHP and Ruby on Rails, which are often combined with MySQL. Wikipedia runs on MediaWiki software, which is written in PHP and uses a MySQL database. Several high-traffic web sites use MySQL for its data storage and logging of user data, including Flickr, Facebook, Wikipedia, Google, [8] Nokia and YouTube.

Libraries for accessing MySQL databases are available in all major programming languages with language-specific APIs. In addition, an ODBC interface called MyODBC allows additional programming languages that support the ODBC interface to communicate with a MySQL database, such as ASP or ColdFusion. The MySQL server and official libraries are mostly implemented in ANSI C/ANSI C++. MySQL will be used to create the database module.

## 4.2.2 Python

Python [13] [14] is a general-purpose high level programming language. It is sometimes referred to as a scripting language as it is often applied in scripting roles. It is commonly defined as an object-oriented scripting language—a definition that blends support for OOP with an overall orientation toward scripting roles. In fact, people often use the word “script” instead of “program” to describe a Python code file. The terms “script” and “program” are used interchangeably, with a slight preference for “script” to describe a simpler top-level file, and “program” to refer to a more sophisticated multi-file application.

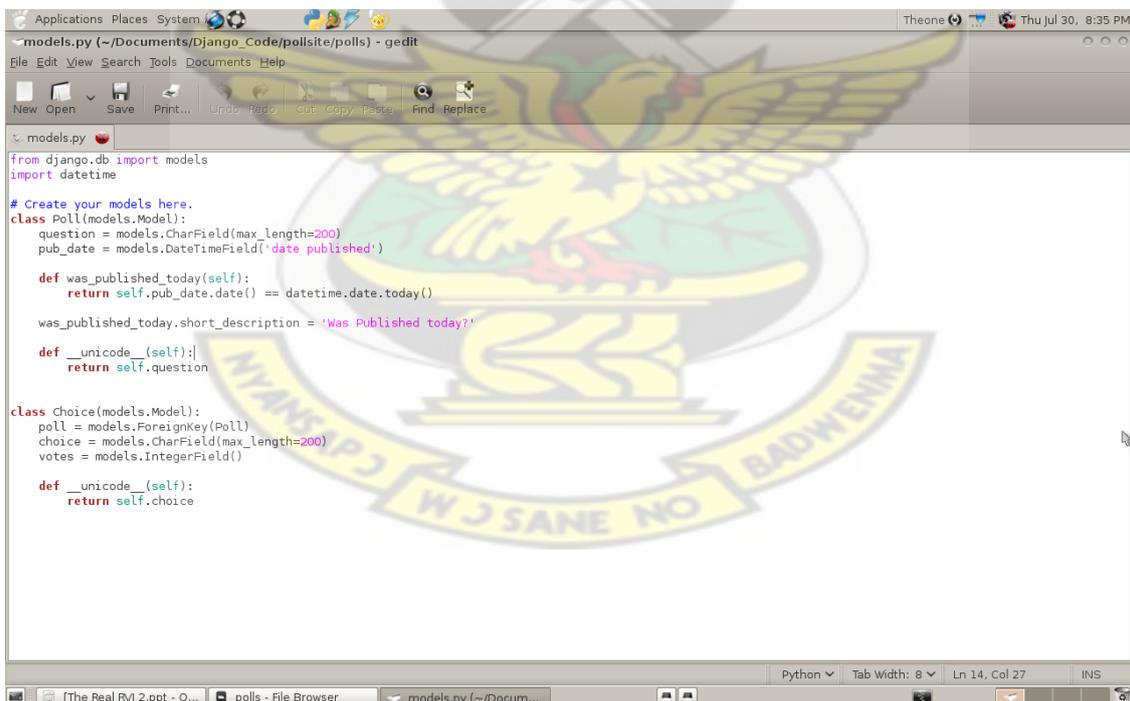
KNUST

The term “scripting language” has so many different meanings to different observers, though in all, three very different associations have been made to it. These are:

- Python is used as a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines (shell scripting) and perform tasks such as processing text files and launching other programs.
- Another association to scripting refers to a “glue” layer used to control and direct (i.e., script) other application components. Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to components that give low-level access to a device. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without having to ship and recompile the entire system’s source code. Python’s simplicity makes it a naturally flexible control tool. Technically, though, this is also just a common Python role; many Python programmers code standalone scripts without ever using or knowing about any integrated components.

- Probably the best way to think of the term “scripting language” is that it refers to a simple language used for quickly coding tasks. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++. Its rapid development cycle fosters an exploratory, incremental mode of programming that has to be experienced to be appreciated. Do not be misconstrued that Python is not just for simple tasks but rather, it makes tasks simple by its ease of use and flexibility. Python has a simple feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for quick tactical tasks and longer-term strategic development.

Python files created are in `.py` extension (Figure 4.1) and they are known as modules. Modules can be called in another file and be used by using the 'import' keyword. Python can be extended to C/C++ and vice versa.



```
models.py
from django.db import models
import datetime

# Create your models here.
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def was_published_today(self):
        return self.pub_date.date() == datetime.date.today()

    was_published_today.short_description = 'Was Published today?'

    def __unicode__(self):
        return self.question

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()

    def __unicode__(self):
        return self.choice
```

Figure 4.1: Example of a python code

### 4.2.3 Django

Django [15] [16] is an open source web application framework, written in Python, which follows the *model-view-controller* [20] [21] [22] (MVC) architectural pattern. It was originally developed to manage several news-oriented sites for The World Company of Lawrence, Kansas, and was released publicly under a BSD license in July 2005.

Django's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes reusability and "pluggability" of components, rapid development, and the principle of DRY (Don't Repeat Yourself). Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative CRUD (create, read, update and delete) interface that is generated dynamically through introspection and configured via admin models.

#### 4.2.3.1 Django components

The core Django framework consists of an object-relational mapper which mediates between data models (defined as Python classes) and a relational database; a regular-expression-based URL dispatcher; a view system for processing requests; and a templating system.

Also included in the core framework are:

- A lightweight, standalone web server for development and testing.
- A form serialization and validation system which can translate between HTML forms and values suitable for storage in the database.
- A caching framework which can use any of several cache methods.
- Support for middleware classes which can intervene at various stages of request processing and carry out custom functions.
- An internal dispatcher system which allows components of an application to communicate

events to each other via pre-defined signals.

- An internationalization system, including translations of Django's own components into a variety of languages.
- A serialization system which can produce and read XML and/or JSON representations of Django model instances.
- A system for extending the capabilities of the template engine.
- An interface to Python's built-in unit test framework.

KNUST

#### 4.2.3.2 Features in Django

The main Django distribution also bundles a number of applications in its *contrib* package, including:

- An extensible authentication system.
- The dynamic administrative interface.
- Tools for generating RSS and Atom syndication feeds.
- A flexible commenting system.
- A sites framework that allows one Django installation to run multiple websites, each with their own content and applications
- Tools for generating Google Sitemaps.
- Tools for preventing cross-site request forgery.
- Template libraries which enable the use of lightweight markup languages such as Textile and Markdown.
- A framework for creating GIS applications.

Figure 4.2 is a screenshot of a Django powered page after creating a project.

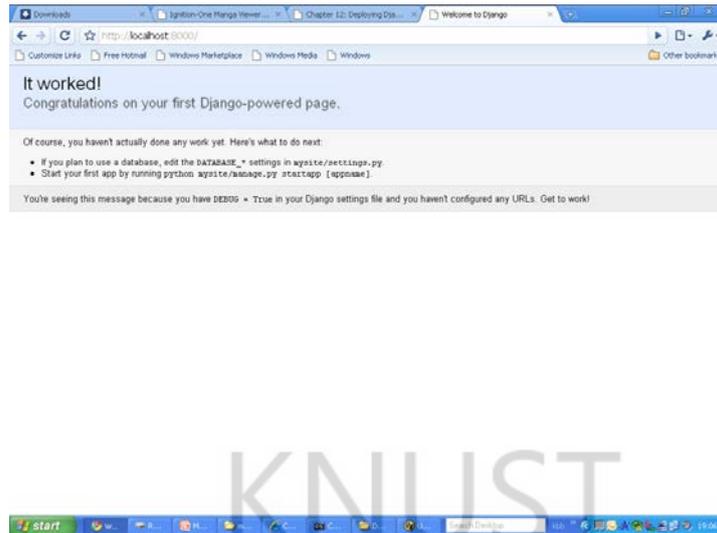


Figure 4.2: Django powered page

#### 4.2.4 jQuery

jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML [23]. It was released in January 2006. Used by over 43% of the 10,000 most visited websites, jQuery is the most popular JavaScript library in use today [24] [25]. jQuery is a free, open source software, dual-licensed under the MIT License and the GNU General Public License, Version 2. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

jQuery also provides capabilities for developers to create plugins on top of the JavaScript library. Using these facilities, developers are able to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. This contributes to the creation of powerful and dynamic web pages.

Microsoft and Nokia have announced plans to bundle jQuery on their platforms [26], Microsoft

adopting it initially within Visual Studio [27] for use within Microsoft's ASP.NET AJAX framework and ASP.NET MVC Framework while Nokia has integrated it into their Web Run-Time widget development platform [28]. jQuery has also been used in MediaWiki since version 1.16[29].

The version of jQuery that will be used for this research is 1.4.2.

#### 4.2.4.1 jQuery Features

jQuery contains the following features:

- DOM element selections using the cross-browser open source selector engine Sizzle, a spin-off out of the jQuery project
- DOM traversal and modification (including support for CSS 1-3)
- Events
- CSS manipulation
- Effects and animations
- Ajax
- Extensibility through plug-ins
- Utilities - such as browser version and the each function.
- Cross-browser support

#### 4.2.5 FusionChartsFree (jQuery plugin)

FusionCharts Free is an open-source free flash charting component that can be used to render data-driven animated charts. Developed in Macromedia Flash MX, FusionCharts can be used with any web scripting language like PHP, ASP, .NET, JSP, ColdFusion, JavaScript, Ruby on Rails, Python etc., to deliver interactive and powerful charts. Using XML as its data interface, FusionCharts makes

full use of Flash to create compact, interactive charts.

FusionCharts Free for jQuery [30] is a jQuery plugin that allows integration FusionCharts Free in a web application using jQuery syntax. FusionCharts Free can be inserted anywhere within a web page, manipulate the chart data and chart cosmetics and even provide data to the chart from simple HTML tables. With the jQuery Plugin for FusionCharts Free, the code becomes concise but coherent.

### **4.3 Implementation**

This section will talk about the implementation of the three modules discussed in chapter three. Before that the architecture of the RVI and how it works must be explained after which the implementation will be discussed.

#### **4.3.1 RVI Architecture**

The RVI is an Actel FPGA that has two types of instruments loaded on it namely the DFM and Function Generator. The FPGA is accessible via a parallel port. The parallel port was used because that was how the FPGA was configured to be accessed. The FPGA is logically designed to have a table that contains 16 registers numbered from 0 to 15 with each register having a size of 1 byte. After the RVI has been loaded with the instruments, selection and reading of these instruments are done by manipulation and/or reading of these registers in the table. The first 8 registers (0 to 7) can only be written whilst the second 8 registers (8 to 15) are read only. To select an instrument that is loaded on the RVI, register 0 must be set to a value.

To access the DFM, the register 0 is set to 2 and the other registers (1 to 7) are set to 0. After it is set, readings are taken on registers (8 to 15) with most significant bit starting from 15. To access the Function Generator, register 0 is set to 1. Register 1 can be manipulated only if register 0 is set

to Function Generator. Register 1 is used to select the waveform type and the values that can be set to it range from 4 to 7. The other registers (2 to 7) are set to 0. After the Function Generator is set, its values are read from register 9. The values change every second because it has been set to a 1Hertz frequency. The values are y-axis plotted against x-axis sequential range of values chosen at will.

#### 4.3.2 Setting Up [20] [21]

A project is first started in Django. A project is a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings. A project named 'RVI', (which is usually a folder) is created, containing the following Python files:

RVI/

`__init__.py`

`manage.py`

`settings.py`

`urls.py`

Despite their small size, these files already constitute a working Django application. The files do the following:

- `__init__.py` is required for Python to treat the 'RVI' directory as a package (i.e., a group of Python modules). It's an empty file, and generally nothing is added to it.
- `manage.py` is command-line utility that enables interaction with the 'RVI' Django project in various ways. There should never be the need to edit this file; it is created in this directory purely for convenience.

- *urls.py* contains the URLs for this Django project. Think of this as the “table of contents” of the Django-powered site.
- *settings.py* is the settings or configuration for this Django project. It contains the database information; path of *urls.py*, templates and template loaders; and the web applications that will be on the server.

After creating a project, the next thing is to create an application (in our case, the middleware). To create a database driven website, an application will have to be created because the ‘models’ (i.e., models in **MVC** architecture) must reside within the application. Like a project, an application, given the name ‘rvi\_middleware’ is created containing the following Python files:

rvi\_middleware/

\_\_init\_\_.py

models.py

views.py

The files do the following:

- *\_\_init\_\_.py*, like that for projects, treats ‘rvi\_middleware’ directory as a package.
- *model.py* holds representations to the database.
- *views.py* contains functions, known as *view* functions, which are responsible for producing the contents of a web page.

It should be noted that files or codes that will be used to access data from the RVI hardware will be stored in the ‘rvi\_middleware’ folder as well.

### 4.3.3 Database

The tools to be used for this module is MySQL and to some extent Python and Django. The database will hold RVI table.

The RVI table holds the following information about the instrument:

- Name (type: string)  
Holds the name of instrument
- Instrument\_type (type: string)  
Type of the instrument
- register\_load1 (type: integer)  
Register on the RVI that data will be sent to load the instrument.
- load\_value1 (type: integer)  
Value passed to register in register\_load1.
- Register\_load2 (type: integer)  
Register on the RVI that data will be sent to load the instrument (this is used for graph-generating) instruments.
- load\_value2 (type: integer)  
Value passed to register in register\_load2.

The queries on these tables will not be done directly on MySQL but through Django (middleware).

The access to the tables and its creation will be done directly from the *models.py* file in the Django application folder *rvi\_middleware*. Each table is represented in the *models.py* file as a class and the columns are represented as variables. The tables are created in MySQL using the *syncdb* command on the *manage.py* terminal. All database queries will be done via the *views.py* table. The code for

*models.py* is shown in Appendix A.

#### 4.3.4 Middleware

The tools to be used for this section are Python and Django. The core functions of the middleware are divided into two. The first is retrieving the values from the RVI and the second is passing the values to the GUI. The modules with Django which are responsible and the functions within the modules will be discussed. The modules (code in Appendix A) are:

- *spp\_fpga\_protocol.py* [6]
- *fpga\_read.py*
- *views.py*

##### spp\_fpga\_protocol.py

This module is responsible for access the RVI via a parallel port and retrieving raw data for processing. The functions responsible for this are:

- *set\_ECP\_ByteMode*

The most common parallel port is the ECP printer port. This function sets the parallel port to byte mode to allow data to be accessed via the port byte-wise.

- *byte\_to\_data\_register*

This function writes one-byte of data from the RVI to the parallel port's data register.

- *write\_command\_register*

This function prepares the RVI 16 one-byte registers to be accessed. After the RVI is prepared, the function calls *byte\_to\_data\_register*.

- *byte\_from\_data\_register*

This function reads one byte of data from the data register.

- *read\_parameters\_reg*

By calling *byte\_from\_data\_register*, this function retrieves data from the 16 registers and puts it in a register table (*reg\_table*).

#### fpga\_read.py

This module modifies the data received before it is passed to the GUI. The modification depends on the type of instrument that is being dealt with. The functions responsible for this are as follows:

- *set\_instrument*

This function sets the RVI to the type of instrument to read.

- *read\_fpga\_for\_graph*

This function retrieves values from the registers in the register table needed to draw the intended graph for the instrument being mimicked. It calls the *read\_parameters\_reg* function. There is a thread in the *read\_fpga\_for\_graph* function that calls it every second later after it has finished executed. This is done to allow periodic retrieval to allow mimic real-time graph plotting.

- *read\_fpga\_for\_discrete*

This function retrieves values from the registers in the register table needed to display the intended discrete value for the instrument being mimicked. It calls the *read\_parameters\_reg* function.

- *xml\_file\_write*

This function uses the register values received from *read\_fpga\_for\_graph* and uses it to create an xml file, *data.xml*, which will be used in FusionChartsFree to draw a graph.

- *discrete\_value\_for\_gui*

This function uses the register values received from *read\_fpga\_for\_discrete* and

manipulates it, and returns one readable value.

### views.py

This module is responsible for producing contents of a webpage. The functions in this module and their details are below:

- *load\_xml*

This function calls the *xml\_file\_write* function to execute. It has a thread that calls the function every ten seconds. This allows mimicking of real-time graph plotting.

- *load\_chart*

This function is responsible for loading the web page for graph-generating instruments. It calls the *load\_xml* function.

- *load\_discrete*

This function is responsible for loading the web page for discrete-value-generating instruments. It calls the *discrete\_value\_for\_gui* function.

#### **4.3.5 Graphical User Interface**

Html files needed to load the GUI for the web browser are also located within the middleware. They are stored in the *templates* folder located in *RVI* project folder. When each instruments URL is keyed in the browser, the *urls.py* file is then accessed by the server for which function in *views.py* to call. The function in turn passes data to the html file assigned to it. The html files for the GUI are:

- *base.html*

This file contains the main design or the look-and-feel of the web-page. Other pages will inherit features from this page.

- *chart.html*

This file generates GUI for graph-generating instruments. It uses FusionChartsFree JQuery plugin to implement this. It inherits the base.html file.

- *value\_read.html*

This file generates GUI for discrete-value-generating instruments. It uses JQuery for visual effects and AJAX to implement this. It inherits the base.html file.

These files are not the only requirements to let the GUI load successfully. There are other files as well that are required and these are found in the *static* folder which is located in the *RVI* project folder. The *static* folder contains JQuery scripts, JavaScript files. Shockwave files and images to let the GUI load as it should be.

#### 4.4 Debugging and Testing

Debugging was done for every function within each module to ensure that it was syntactically correct. All runtime errors encountered were rectified. After that the functions were *unit tested* to ensure that they did the right thing and they do the thing right. This was done with the help of the Python module *unittest* which is also known as *PyUnit*. The module created is *tests.py* which can also be found in the application folder.

Functions created in this module are used to test functions that were created in modules *fpga\_read.py* and *views.py*. Hence the test function names are the names of the functions with the word *test* pre-pended to them. The functions are:

- *test\_read\_fpga\_for\_graph*
- *test\_read\_fpga\_for\_discrete*

- *test\_xml\_file\_write*
- *test\_discrete\_value\_for\_gui*
- *test\_load\_xml*
- *test\_load\_chart*
- *test\_load\_discrete*
- *test\_set\_instrument*

All those tests, tests were done periodically to make sure that the interfaces were displayed correctly. It was observed that switching between Function Generator to DFM throws an exception page during the change. This was due to the fact that the reset switch on the RVI had to be pressed for the DFM to take effect. Thus the time taken for switching from Function Generator to DFM takes about 5 seconds to be effected. The reverse switching is however immediate. Also switching between waveforms for the DFM is also immediate.

#### **4.5 Results**

After debugging and testing to ensure that the code is free of errors, accessing the RVI via a web browser was successful for the two categories of instruments. The Function Generator was loaded on the RVI and accessed via the web browser. See screenshot of the result below:

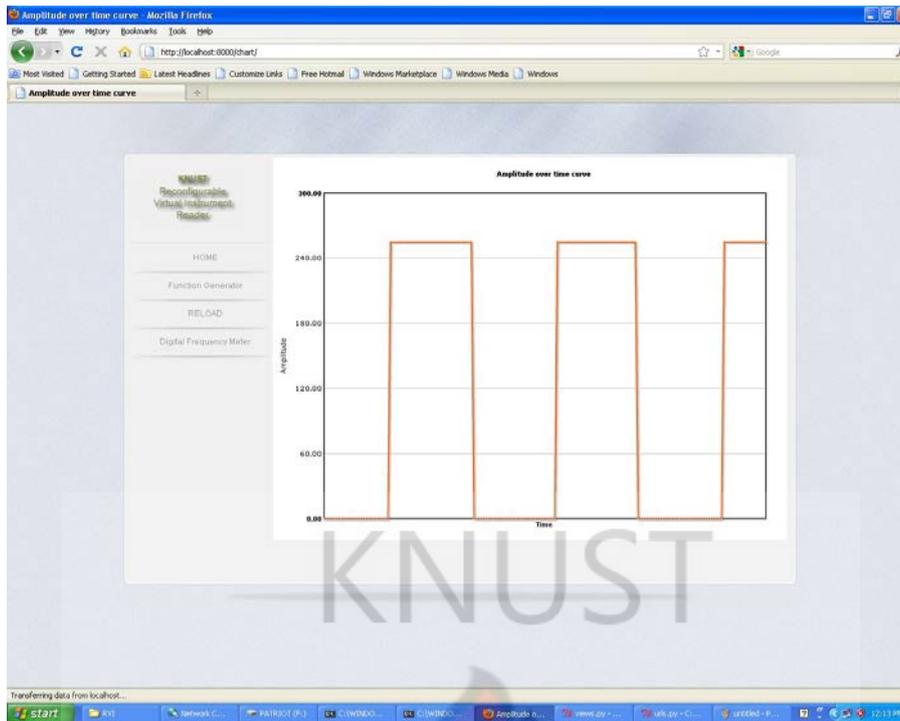


Figure 4.3: GUI of Function generator

All PCs connected via network were able to access the RVI via a URL that was assigned to the server that had the RVI connected. From the graph above, it is evident the waveform is not entirely smooth. This is due to the attempt to ensure real-time processing of RVI data but this cannot be achieved due to the factors such as the servers processing speed. In order to attain this close level of smoothness for the graph, the value change was set to a 1Hertz, the RVI was accessed by the server every second and the x-axis (time) values was given a step-wise increment value of 0.3.

After that the DFM was loaded on the RVI and access via the web browser. The screenshot of the result is also shown below:

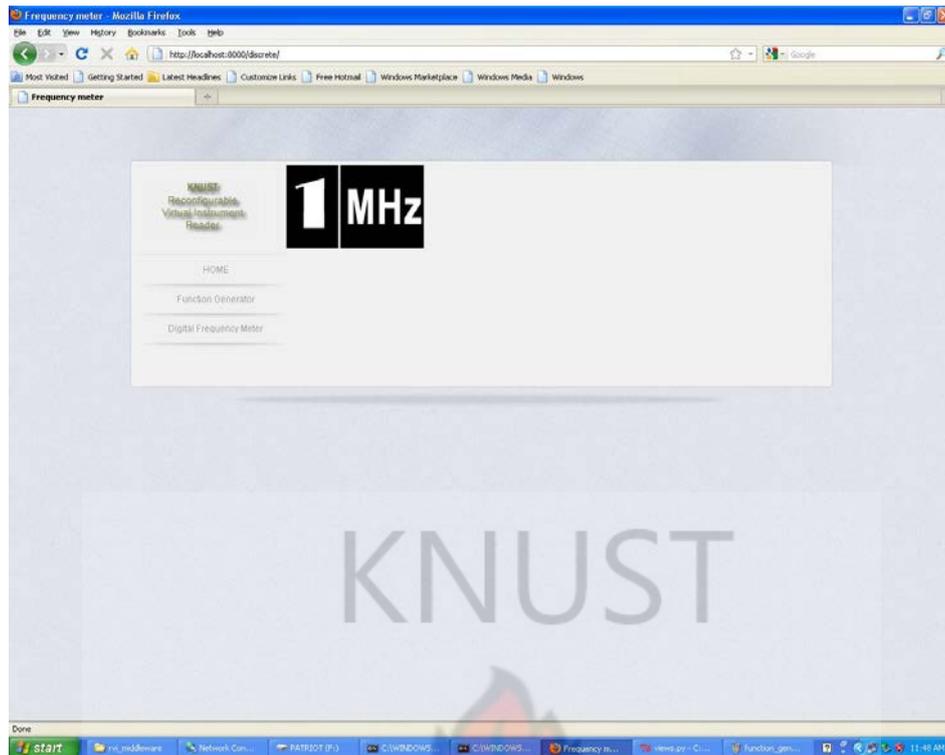


Figure 4.5: GUI of Frequency Meter

The data that is pulled from the RVI for DFM is unlike that of the Function Generator such that the data does not change but remains the same until queried again. So for that, there is no margin of errors for that. What is displayed is for each digit in the screen above is what is pulled from registers 8 to 15 with 15 being the most significant digit.

## **CHAPTER FIVE: CONCLUSION AND RECOMMENDATION**

### **5.1 Conclusion**

Reconfigurable Virtual Instruments (RVIs) make the emulation of traditional instruments possible. However, the commercially available RVIs are created based on proprietary hardware and software making acquisition and maintenance quite costly. Research to make cost-effective RVIs with the use of existing hardware components, such as the FPGA, has been achieved in Ghana[11]. However the interface of the RVI is not user friendly and trying to get special hardware to display readings might also prove costly. The aim of this thesis is to interface the RVI using open source tools that are downloadable from the internet. Not only that but it should be accessible via distributed network too as well.

RVI architecture was discussed in chapter 4 to provide a deeper understanding of the how instruments were loaded on the RVI. The registers were discussed and also the classification of the DFM and Function Generator as discrete-value and graph-generating instruments respectively. This served as a firm foundation for the development of the application enabling users to load instruments and take readings. Apart from the DFM and Function Generator discussed in this thesis, this application can accommodate new instruments should they be available in the future. All that needs to be known for their readings to take effect is the register to activate the instrument and the registers to take the readings from.

Finally this project has made interfacing of the RVI via a web browser easy without the need for any special installation. With other web services that use Java for example, if the user is using a PC devoid of Java, it will have to install the software first. Assuming that the user is connected to a network that has no internet, making use of the web service will be impossible. For this Django-

powered web service, anyone with a web-browser can have access to the RVI.

## 5.2 Future Work

Though an exception is thrown during an instrument switch from Function Generator to DFM causes some delay and uncomfortable situations, it is not a major problem because what can be is to add a button on a web page to just load the instrument and another button to load the readings. Thus right after the button to load the instrument is clicked; the user resets the RVI and then loads the reading. Another alternative is to make configurations on the RVI such that there is no need to make a reset on the RVI when a switch is made.

The communication between the RVI and the PC was done via a parallel port because that was how the RVI was done to communicate with the PC. Had there been more time given for this research, more could have been done. Interaction with the RVI via Ethernet, serial or even USB can be looked at because the parallel port is being phased out. The future at the moment is with Ethernet and USB. Also more research can be done to ensure that the application can work for all sorts of RVI hardware not only Actel FPGA. This will also be dependent on how the hardware is configured as well.

## REFERENCES

1. World Wide Web URL [http://en.wikipedia.org/w/index.php?title=Measuring\\_instrument](http://en.wikipedia.org/w/index.php?title=Measuring_instrument), last accessed February 3rd, 2009
2. World Wide Web URL <http://en.wikipedia.org/w/index.php?title=Instrumentation>, last accessed February 3rd, 2009
3. World Wide Web URL [http://en.wikipedia.org/w/index.php?title=Integrated Instrumentation system](http://en.wikipedia.org/w/index.php?title=Integrated_Instrumentation_system), last accessed February 3rd, 2009
4. World Wide Web URL [http://en.wikipedia.org/w/index.php?title=Virtual Instrumentation](http://en.wikipedia.org/w/index.php?title=Virtual_Instrumentation), last accessed February 3rd, 2009
5. World Wide Web URL <http://www.ni.com/virtualinstrumentation>, last accessed February 3rd, 2009
6. Andres Cicuttin, Maria Liz Crespo, Alexander Shapiro and Nizar Abdallah, "Reconfigurable Virtual Instrumentation", ICTP-INFN Advanced Training Course on FPGA Design and VHDL for Hardware Simulation and Synthesis, Trieste, Italy, Nov. 27-Dec. 22, 2006
7. Andrew S. Tanenbaum, Martin van Steen: Distributed Systems, Principles Distributed Systems and Paradigms; Prentice Hall 2002
8. David E. Bakken, "Middleware", Washington State University, Encyclopedia of Distributed Computing, Kluwer Academic Press, 2003.
9. Gerd Van den Branden, Geert Braeckman, Abdellah Touhafi, Erik Dirckx, "A Dynamically Reconfigurable Virtual Instrument" Erasmushogeschool Brussel, Departement IWT, Nijverheidskaai 170, 1070 Brussel, Belgium
10. World Wide Web URL <http://en.wikipedia.org/w/index.php?title=Middleware>, last accessed February 3rd, 2009
11. Gilbert Osei-Dadzie, Kwame Osei Boateng, "Application of a Block-based Approach in Design of

a Reconfigurable Virtual Instrumentation Platform”, (a) Proceedings of The 2008 IAJC-IJME International Conference, ISBN 978-1-60643-379-9 (b) International Journal of Agile Manufacturing (IJAM), Vol. 11, Issue 1, pp. 39-43, 2009

12. World Wide Web URL <http://en.wikipedia.org/wiki/MySQL>, last accessed July 10th, 2010
13. World Wide Web URL [http://en.wikipedia.org/wiki/Python \(programming language\)](http://en.wikipedia.org/wiki/Python_(programming_language)), last accessed April 15th, 2009
14. World Wide Web URL <http://www.python.org>, last accessed April 15th, 2009
15. World Wide Web URL [http://en.wikipedia.org/wiki/Django \(Web framework\)](http://en.wikipedia.org/wiki/Django_(Web_framework)), last accessed April 15th, 2009
16. World Wide Web URL <http://www.djangoproject.com/>, last accessed January 25th, 2011
17. World Wide Web URL <http://www.mysql.com/>, last accessed January 28th, 2011
18. World Wide Web URL <http://dev.mysql.com/>, last accessed January 28th, 2011
19. World Wide Web URL [http://en.wikipedia.org/wiki/Python syntax and semantics](http://en.wikipedia.org/wiki/Python_syntax_and_semantics), last accessed April 16th, 2009
20. Adrian Holovaty, Jacob Kaplan-Moss, “The Definitive Guide to Django”, Apress 2008
21. World Wide Web URL <http://www.djangobook.com>, last accessed August 13th, 2011
22. World Wide Web URL <http://en.wikipedia.org/wiki/Model-view-controller>, last accessed June 12th, 2011
23. World Wide Web URL <http://jquery.com/>, last accessed May 19th, 2011
24. World Wide Web URL <http://trends.builtwith.com/javascript/JQuery>, last accessed May 19th, 2011
25. World Wide Web URL [http://w3techs.com/technologies/overview/javascript library/all](http://w3techs.com/technologies/overview/javascript_library/all), last accessed May 19th, 2011
26. World Wide Web URL <http://jquery.com/blog/2008/09/28/jquery-microsoft-nokia/>, last accessed May 19th, 2011

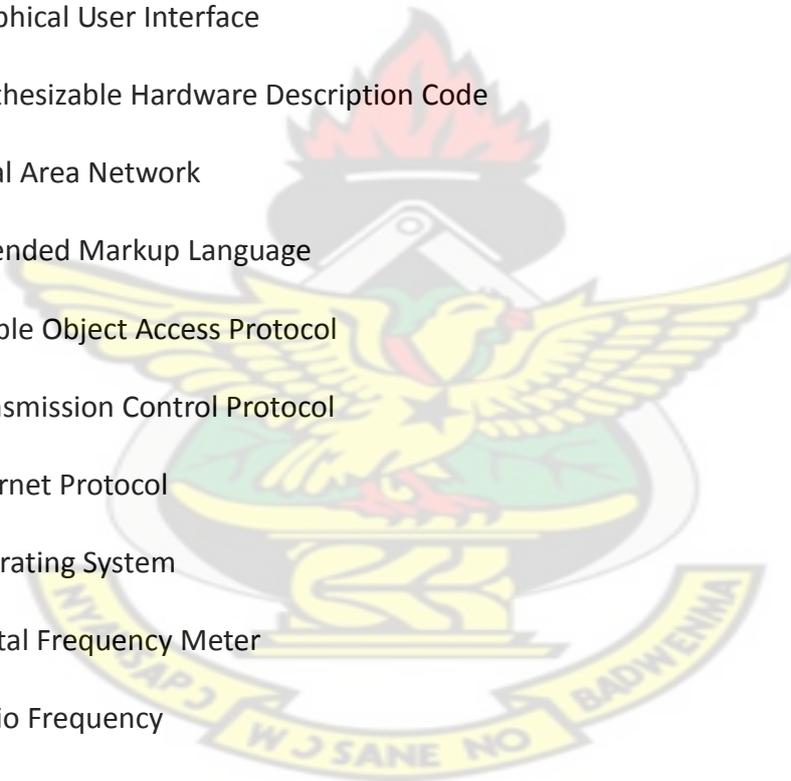
27. World Wide Web URL <http://weblogs.asp.net/scottgu/archive/2008/09/28/jquery-and-microsoft.aspx>, last accessed May 19th, 2011
28. World Wide Web URL [http://wiki.forum.nokia.com/index.php/Guarana\\_UI:\\_a\\_jQuery-Based\\_UI\\_Library\\_for\\_Nokia\\_WRT](http://wiki.forum.nokia.com/index.php/Guarana_UI:_a_jQuery-Based_UI_Library_for_Nokia_WRT), last accessed May 19th, 2011
29. World Wide Web URL <http://www.mediawiki.org/wiki/JQuery>, last accessed May 19th, 2011
30. World Wide Web URL <http://www.fusioncharts.com/jquery/>, last accessed May 19th, 2011



## GLOSSARY

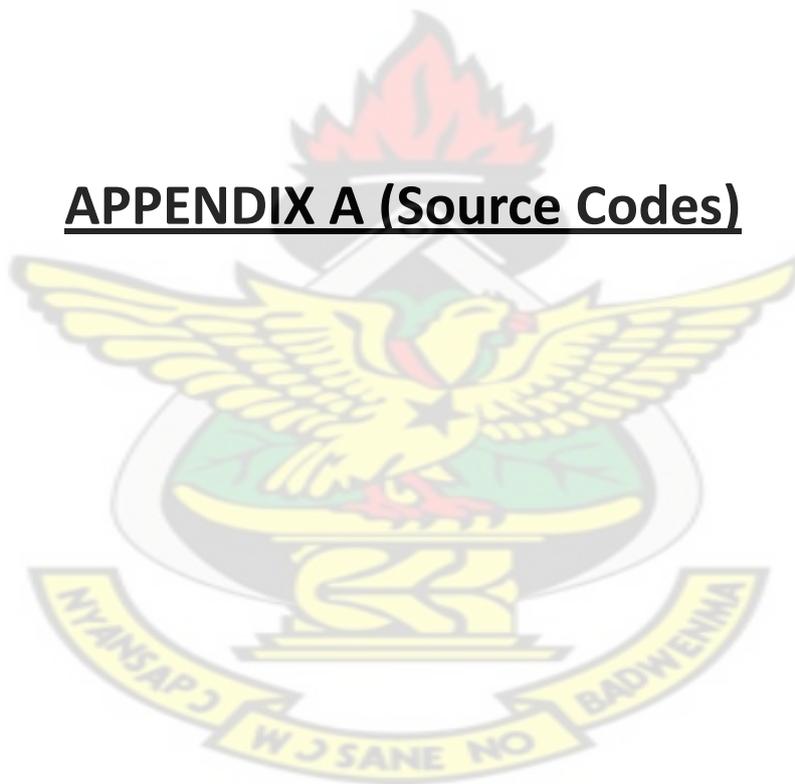
PC	-	Personal Computer
RVI	-	Reconfigurable Virtual Instruments
FPGA	-	Field Programmable Gate Arrays
ADC	-	Analog to Digital Converter
DAC	-	Digital to Analog Converter
VI	-	Virtual Instrument
GUI	-	Graphical User Interface
SHDC	-	Synthesizable Hardware Description Code
LAN	-	Local Area Network
XML	-	eXtended Markup Language
SOAP	-	Simple Object Access Protocol
TCP	-	Transmission Control Protocol
IP	-	Internet Protocol
OS	-	Operating System
DFM	-	Digital Frequency Meter
RF	-	Radio Frequency

KNUST



# KNUST

## APPENDIX A (Source Codes)



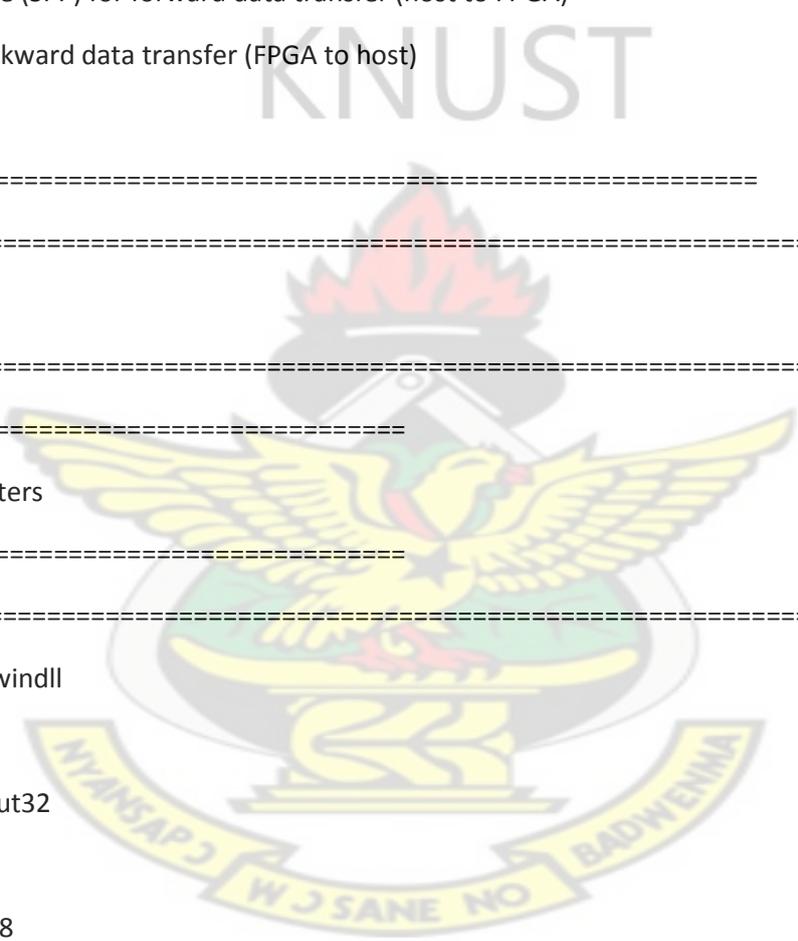
**spp\_fpga\_protocol.py**

```
#=====
# =====
# Source: SPP-FPGA_Protocol.c
# Author: Maria Liz Crespo, ICTP-INFN Mlab, ICTP, Trieste (Italy)
# Description:
# Parallel Port for Virtex FPGA Communication
# compatibility mode (SPP) for forward data transfer (host to FPGA)
# byte mode for backward data transfer (FPGA to host)
#
# =====
#=====
#=====
# =====
# Parallel Port Registers
# =====
#=====
from ctypes import windll

p_port = windll.inpout32

data_register = 0x378
status_register = 0x379
control_register = 0x37a

#=====
# =====
#
# Bit Position in Control Register (0x37a)
```



```

# HostClk(nStrobe)  bit0  SPP:active low (inverted)

# HostBusy(nAutoLinefeed)  bit1  SPP:low(command)&high(data) (inverted)

#          Byte:active low

# nInit(nInit)      bit2  Byte:active low

# Active1284(nSelect)  bit3  SPP:active low (inverted)

# Enable bi-dir port  bit5  Byte:active high

#

# =====

#=====

HostClk_low = 0xfe # bit0=0
HostClk_high = 0x01 # bit0=1

HostBusy_low = 0xfd # bit1=0
HostBusy_high = 0x02 # bit1=1

nInit_low = 0xfb # bit2=0
nInit_high = 0x04 # bit2=1

Active1284_low = 0xf7 # bit3=0
Active1284_high = 0x08 # bit3=1

EnableBidir_low = 0xdf # bit5=0
EnableBidir_high = 0x20 # bit5=1

#=====

# =====

#

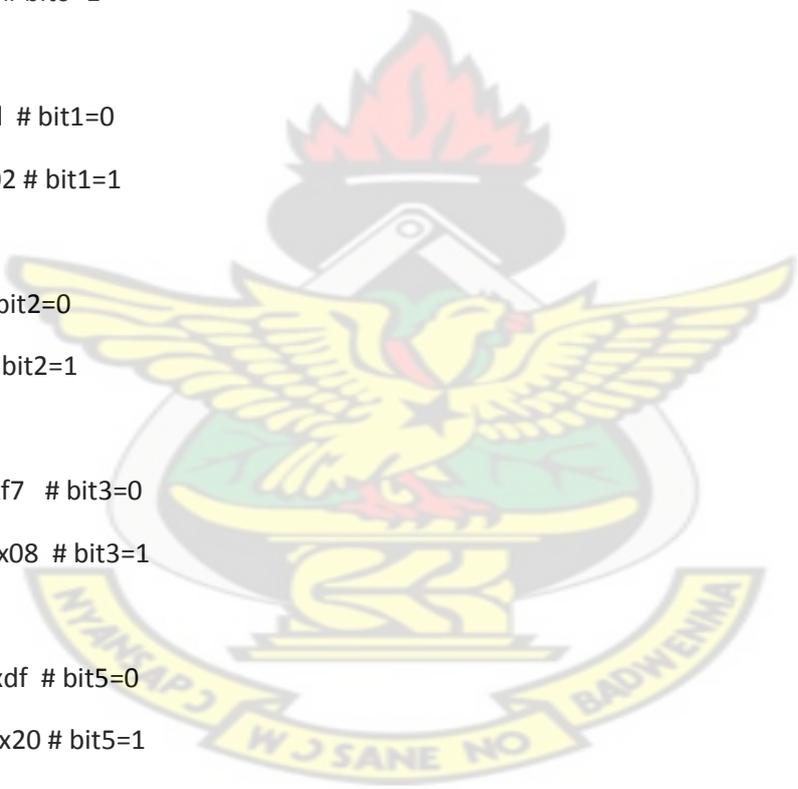
# Bit Position in Status Register (0x379)

# nDataAvail(nError)  bit3  Byte: active low

# Xflag(Select)      bit4  SPP: active high

```

KNUST



# PtrClk(nAck) bit6 SPP: active low

# Byte: active low

# =====

#=====

nDataAvail = 3

Xflag = 4

PtrClk = 6

#=====

# =====

# Bit Position in ECP Control Register (0x77a)

# Operation Mode: bits 7:5 (Standard Mode= 000 (def:0x15))

# (Byte Mode= 001 (0x35))

# (ECP FIFO Mode= 011 (0x75))

# ECP Interrupt Bit: bit 4

# DMA Enable Bit: bit 3

# ECP Service Bit: bit 2 (only read)

# FIFO Full: bit 1 (only read)

# FIFO Empty: bit 0 (only read)

#

# =====

#=====

ECP\_register = 0x77A

#=====

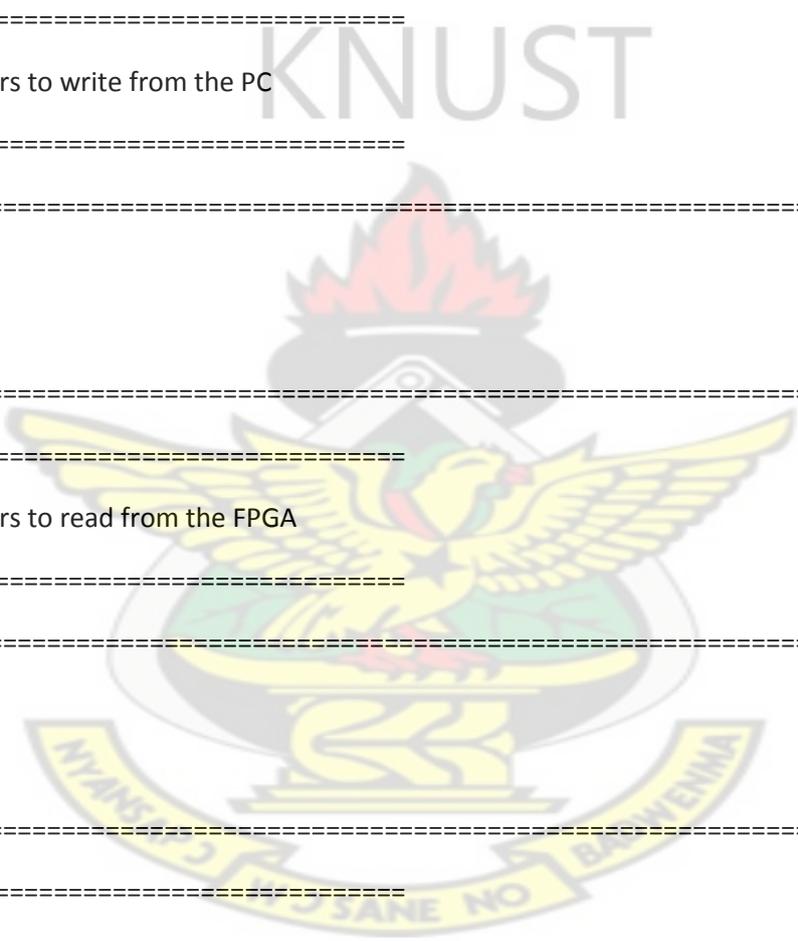
# =====

# FPGA command register ("only wr"):

# bit0 reset (active high),

# bit1 registers (0) or memory (1),

```
# =====  
# =====  
command_reset = 0x01 # bit0=1  
  
command_reg = 0x00 # bit1=0  
command_mem = 0x02 # bit1=1  
  
# =====  
# =====  
# Number of registers to write from the PC  
# =====  
# =====  
reg_num_wr = 8  
  
# =====  
# =====  
# Number of registers to read from the FPGA  
# =====  
# =====  
reg_num_rd = 16  
  
# =====  
# =====  
# Memory size in bytes words  
# =====  
# =====  
mem_size = 256  
  
# =====  
# =====  
# Number of cycles for time-out
```



```

# =====
#=====
TIME_OUT = 100

#=====
# =====
# FPGA parameters registers:
# =====
#=====
MEM_ADDRESS_INI = 0 #Initial FPGA memory address
MEM_LENGTH = 1 #Number of words to read

#=====
# =====
# Global Variables
# =====
#=====

reg_table = [None]*16
mem_table = []

#=====
# =====

# Function: byte_to_data_register
# Parameter: byte to write in the parallel port data register
# Return: void
# Description: Write a byte in the parallel port data register by
#             following the SPP forward data transfer protocol
# =====
#=====

def byte_to_data_register(byte_value):

```

```
p_port.Out32 ( data_register, byte_value )
p_port.Out32 (control_register, p_port.Inp32(control_register) | HostClk_high)
```

```
for i in range(TIME_OUT):
    end_timeout = 1
    bit = p_port.Inp32(status_register) & (1 << PtrClk)
```

```
if ( ( bit >> PtrClk ) == 0 ):
    end_timeout = 0
    break
```

```
if (end_timeout):
    print("\nTimeout Error: Host is waiting for PtrClk(nAck) low\n")
    exit(1)
```

```
#=====
# =====
```

```
# 4- Give the host clock raising edge
#   control_register[HostClk(nStrobe)] = 0 (inverted)
```

```
# =====
#=====
```

```
p_port.Out32 (control_register,p_port.Inp32(control_register) & HostClk_low)
```

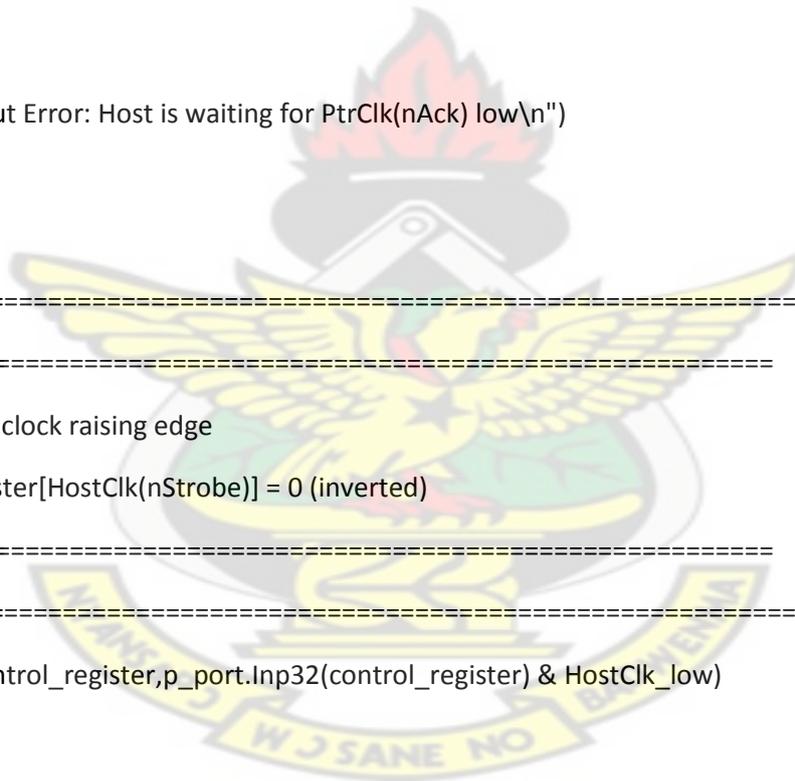
```
#=====
# =====
```

```
# 5- Wait for "PtrClk(nAck) high" in status register
```

```
# =====
#=====
```

```
for i in range(TIME_OUT):
    end_timeout=1
    bit = p_port.Inp32(status_register) & (1 << PtrClk)
```

KNUST



```
if ( ( bit >> PtrClk ) == 1 ):
```

```
    end_timeout = 0
```

```
    break
```

```
if (end_timeout):
```

```
    print("\nTimeout Error: Host is waiting for PtrClk(nAck) high\n")
```

```
    exit(1)
```

# KNUST

```
#=====
```

```
# =====
```

```
# Function: write_command_register
```

```
# Parameter: byte to write in the FPGA command_register
```

```
# Return: void
```

```
# Description: Write a command byte in the FPGA command_register
```

```
# =====
```

```
#=====
```

```
def write_command_register (byte_value):
```

```
    #=====
```

```
    # =====
```

```
    # 1- Init control register
```

```
    # control_register = xx0x 0100 = 0x04
```

```
    # =====
```

```
    #=====
```

```
    p_port.Out32 (control_register,0x04)
```

```
    #=====
```

```
    # =====
```

```
    # 2- Start forward data transfer
```

```

# control_register[Active1284(nSelect)] = 1 (inverted)
# =====
#=====
p_port.Out32 (control_register,p_port.Inp32(control_register) | Active1284_high)

#=====
# =====
# 3- Wait for "Xflag(Select) high" in status register
# =====
#=====
for i in range(TIME_OUT):
    end_timeout=1
    bit = p_port.Inp32(status_register) & (1 << Xflag)

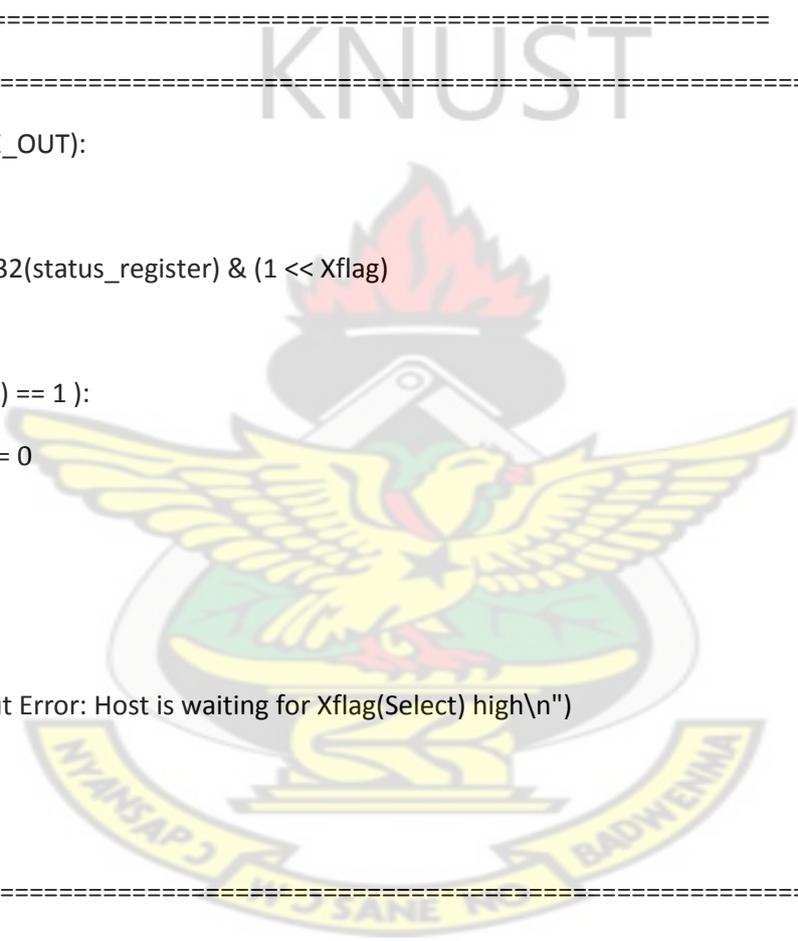
    if ( ( bit >> Xflag ) == 1 ):
        end_timeout = 0
        break

if (end_timeout):
    print("\nTimeout Error: Host is waiting for Xflag(Select) high\n")
    exit(1)

#=====
# =====
# 4- Indicate that the byte to write is a command
# control_register[HostBusy(nAutoLinefeed)] = 1 (inverted)
# =====
#=====
p_port.Out32 (control_register,p_port.Inp32(control_register) | HostBusy_high)

#=====

```



```

# =====
# 5- Write command byte to data register
# =====
#=====
byte_to_data_register (byte_value)

#=====
# =====
# 6- End forward data transfer
#   control_register[Active1284(nSelect)] = 0 (inverted)
# =====
#=====
p_port.Out32 (control_register,p_port.Inp32(control_register) & Active1284_low)

#=====
# =====
# 7- Wait for "Xflag(Select) low" in status register
# =====
#=====

for i in range(TIME_OUT):
    end_timeout=1
    bit = p_port.Inp32(status_register) & (1 << Xflag)

    if ( ( bit >> Xflag ) == 0 ):
        end_timeout = 0
        break

if (end_timeout):
    print("\nTimeout Error: Host is waiting for Xflag(Select) low\n")
    exit(1)

```

```

#=====
# =====
# Function: byte_from_data_register
# Parameter: none
# Return: byte from the parallel port data register
# Description: Read a byte from the parallel port data register by
#             following the byte mode backward data transfer protocol
# =====
#=====
def byte_from_data_register():

#=====
# =====
# 1- Indicate that it can accept data from the peripheral
# control_register[HostBusy(nAutoLinefeed)] = 1 (inverted)
# =====
#=====

p_port.Out32 (control_register,p_port.Inp32(control_register) | HostBusy_high)

#=====
# =====
# 2- Wait for "PtrClk(nAck) low" in status register
# =====
#=====

for i in range(TIME_OUT):
    end_timeout=1
    bit = p_port.Inp32(status_register) & (1 << PtrClk)

    if ( ( bit >> PtrClk ) == 0 ):
        end_timeout = 0

```

```
break
```

```
if (end_timeout):
```

```
    print("\nTimeout Error: Host is waiting for PtrClk(nAck) low\n")
```

```
    return 0
```

```
#=====
```

```
#=====
```

```
# 3- Read byte from data register
```

```
#=====
```

```
#=====
```

```
byte_value = p_port.Inp32 (data_register)
```

```
#=====
```

```
#=====
```

```
# 4- Indicate that it is processing the byte
```

```
# control_register[HostBusy(nAutoLinefeed)] = 0 (inverted)
```

```
#=====
```

```
#=====
```

```
p_port.Out32 (control_register,(p_port.Inp32(control_register) & HostBusy_low))
```

```
#=====
```

```
#=====
```

```
# 5- Wait for "PtrClk(nAck) high" in status register
```

```
#=====
```

```
#=====
```

```
for i in range(TIME_OUT):
```

```
    end_timeout = 1
```

```
    bit = p_port.Inp32(status_register) & (1 << PtrClk)
```

```
if ( ( bit >> PtrClk ) == 1 ):
```

```
    end_timeout = 0
```

```
    break
```

```
if (end_timeout):
```

```
    print("\nTimeout Error: Host is waiting for PtrClk(nAck) high\n")
```

```
    return 0
```

# KNUST

```
#=====
```

```
#=====
```

```
# 6- Return byte from data register
```

```
#=====
```

```
#=====
```

```
return byte_value
```

```
#=====
```

```
#=====
```

```
# Function: read_parameters_reg
```

```
# Parameter: none
```

```
# Return: void
```

```
# Description: Saves the current parameters registers values in
```

```
#     reg_table
```

```
#
```

```
# Read FPGA Parameters Registers
```

```
# -----
```

```
# 1- Indicate in the FPGA command register the access operation on registers
```

```

# (bit 1 of FPGA command register = 0)
#
# 1.1- Init control register
#   control_register = xx0x 0100 = 0x04
# 1.2- Start forward data transfer
#   control_register[Active1284(nSelect)] = 0
# 1.3- Wait for "Xflag(Select) high" in status register
# 1.4- Indicate that the byte to write is a command
#   control_register[HostBusy(nAutoLinefeed)] = 1
# 1.5- Write the command byte to data register by following the
#   SPP forward transfer protocol
#
# 2- Init control register
#   control_register = xx0x 0100 = 0x04
#
# 3- Request backward data transfer
#   control_register[nInit(nInit)] = 0
#
# 4- Wait for "nDataAvail(nError) low" in status register
#
# 5- Read block of registers values (16 registers) from data register
#
# 5.1- Place the data bus in a high impedance state
#   control_register[Enable bi-dir port] = 1
#
# 5.2- For each register value repeat:
#   a- Indicate that it can accept data from the peripheral
#       control_register[HostBusy(nAutoLinefeed)] = 1 (inverted)
#   b- Wait for "PtrClk(nAck) low" in status register
#   c- Read byte from data register
#   d- Indicate that it is processing the byte

```

```

# control_register[HostBusy(nAutoLinefeed)] = 0 (inverted)
# e- Wait for "PtrClk(nAck) high" in status register
#
# 6- Wait for "nDataAvail(nError) high" in status register
#
# 7- End of request backward data transfer
# control_register[nInit(nInit)] = 1
#
# =====
# =====
def read_parameters_reg():
# =====
# =====
# 1- Indicate in the FPGA command register the access operation on registers
# (bit 1 of FPGA command register = 0)
# =====
# =====
write_command_register(command_reg)
# =====
# =====
# 2- Init control register
# control_register = xx0x 0100 = 0x04
# =====
# =====
p_port.Out32 (control_register,0x04)
# =====

```

```

# =====
# 3- Request backward data transfer
#   control_register[nInit(nInit)] = 0
# =====
#=====
p_port.Out32 (control_register,(p_port.Inp32(control_register) & nInit_low))

#=====
# =====
# 4- Wait for "nDataAvail(nError) low" in status register
# =====
#=====

for i in range(TIME_OUT):
    end_timeout = 1
    bit = p_port.Inp32(status_register) & (1 << nDataAvail)

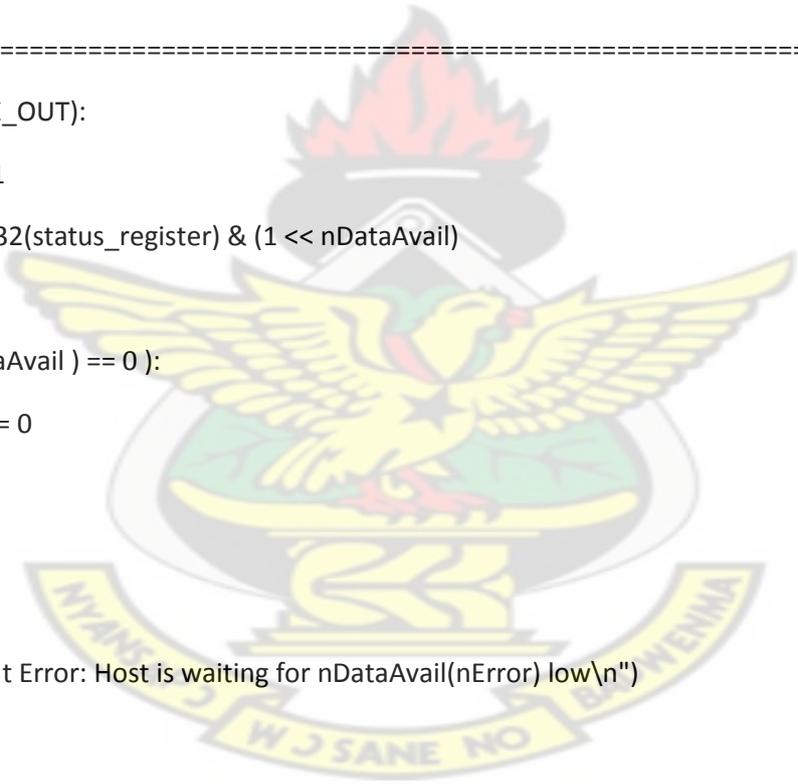
    if ( ( bit >> nDataAvail ) == 0 ):
        end_timeout = 0
        break

if (end_timeout):
    print("\nTimeout Error: Host is waiting for nDataAvail(nError) low\n")
    exit(1)

#=====
# =====
# 5- Read block of registers values
# =====
#

```

KNUST



```

# =====
# 5.1- Place the data bus in a high impedance state
#     control_register[Enable bi-dir port] = 1
# =====
#=====

p_port.Out32 (control_register,(p_port.Inp32(control_register) | EnableBidir_high))

#=====
# =====
# 5.2- For each register value: read data byte from data register
# =====
#=====

global reg_table

for j in range(reg_num_rd):
    reg_table[j] = byte_from_data_register()

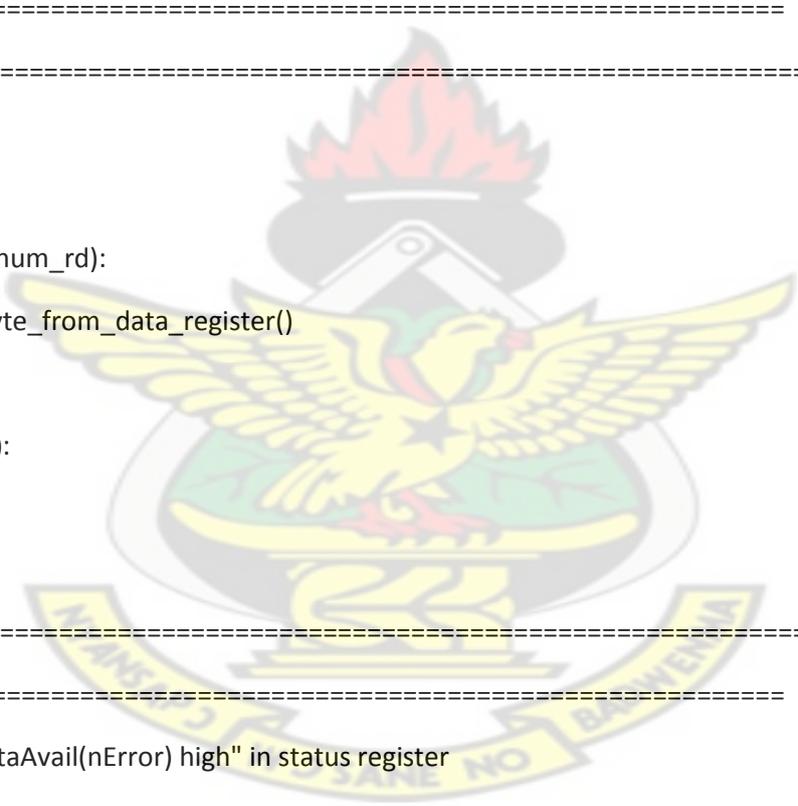
    if (end_timeout):
        exit(1)

#=====
# =====
# 6- Wait for "nDataAvail(nError) high" in status register
# =====
#=====

for i in range(TIME_OUT):
    bit = p_port.Inp32(status_register) & (1 << nDataAvail)

    if ( ( bit >> nDataAvail ) == 1 ):
        end_timeout = 0
        break

```



```

if (end_timeout):
    print("\nTimeout Error: Host is waiting for nDataAvail(nError) high\n")
    exit(1)

```

```

#=====
# =====
# 7- End of request backward data transfer
#   control_register[nInit(nInit)] = 1
# =====
#=====
p_port.Out32 (control_register,(p_port.Inp32(control_register) | nInit_high))

```

```

#=====
# =====
# Function: write_parameters_reg
# Parameter: none
# Return: void
# Description: Copy from reg_table to parameters registers
#
#
# Write FPGA Parameters Registers
# -----
# 1- Indicate in the FPGA command register the access operation on registers
#   (bit 1 of FPGA command register = 0)
#
#   1.1- Init control register
#       control_register = xx0x 0100 = 0x04

```

```

# 1.2- Start forward data transfer
#   control_register[Active1284(nSelect)] = 0
# 1.3- Wait for "Xflag(Select) high" in status register
# 1.4- Indicate that the byte to write is a command
#   control_register[HostBusy(nAutoLinefeed)] = 1
# 1.5- Write the command byte to data register by following the
#   SPP forward transfer protocol
#
# 2- Write block of registers values to data register
#
# 2.1- Indicate that the byte to write is a data
#   control_register[HostBusy(nAutoLinefeed)] = 0 (inverted)
# 2.2- For each register value repeat:
#   Write the data byte to data register by following the
#   SPP forward transfer protocol
#
# 3- End forward data transfer
#   control_register[Active1284(nSelect)] = 0 (inverted)
#
# 4- Wait for "Xflag(Select) low" in status register
#
# =====
# =====
def write_parameters_reg():
# =====
# =====
# 1- Indicate in the FPGA command register the access operation on registers
# (bit 1 of FPGA command register = 0)
# =====
#

```

```

# =====
# 1.1- Init control register
#   control_register = xx0x 0100 = 0x04
# =====
#=====

p_port.Out32 (control_register,0x04)

#=====
# =====
# 1.2- Start forward data transfer
#   control_register[Active1284(nSelect)] = 1 (inverted)
# =====
#=====

p_port.Out32 (control_register,(p_port.Inp32(control_register) | Active1284_high))

#=====
# =====
# 1.3- Wait for "Xflag(Select) high" in status register
# =====
#=====

for i in range(TIME_OUT):
    end_timeout = 1
    bit = p_port.Inp32(status_register) & (1 << Xflag)

    if ( ( bit >> Xflag ) == 1 ):
        end_timeout = 0
        break

if (end_timeout):
    print("\nTimeout Error: Host is waiting for Xflag(Select) high\n")
    exit(1)

```

#=====

# =====

# 1.4- Indicate that the byte to write is a command

# control\_register[HostBusy(nAutoLinefeed)] = 1 (inverted)

# =====

#=====

p\_port.Out32 (control\_register,(p\_port.Inp32(control\_register) | HostBusy\_high))

#=====

# =====

# 1.5- Write command byte to data register

# =====

#=====

byte\_to\_data\_register (command\_reg)

#=====

# =====

# 2- Write block of registers values to data register

# =====

#

# =====

# 2.1- Indicate that the byte to write is a data

# control\_register[HostBusy(nAutoLinefeed)] = 0 (inverted)

# =====

#=====

p\_port.Out32 (control\_register,(p\_port.Inp32(control\_register) & HostBusy\_low))

#=====

# =====

```

# 2.2- For each register value: write data byte to data register

# =====
#=====

for j in range(reg_num_wr):
    byte_to_data_register(reg_table[j])

#=====
# =====

# 3- End forward data transfer
#   control_register[Active1284(nSelect)] = 0 (inverted)
# =====
#=====

p_port.Out32 (control_register,(p_port.Inp32(control_register) & Active1284_low))

#=====
# =====

# 4- Wait for "Xflag(Select) low" in status register
# =====
#=====

for i in range(TIME_OUT):
    end_timeout = 1
    bit = p_port.Inp32(status_register) & (1 << Xflag)

    if ( ( bit >> Xflag ) == 0 ):
        end_timeout = 0
        break

if (end_timeout):
    print("\nTimeout Error: Host is waiting for Xflag(Select) low\n")
    exit(1)

```

```
from fpga_read import xml_file_write, discrete_value_for_gui, set_instrument
from RepeatTimer import RepeatTimer
from django.shortcuts import render_to_response
import os

#path of views file
file_path = os.path.dirname(__file__)

#path of FusionChartsFree data.xml file
xml_file_path = os.path.join(file_path, '..\\static\\data.xml')

#boolean variable to check whether load_xml has already been called
is_graph_Loaded = False

#load_xml thread
load_xml_thread = None

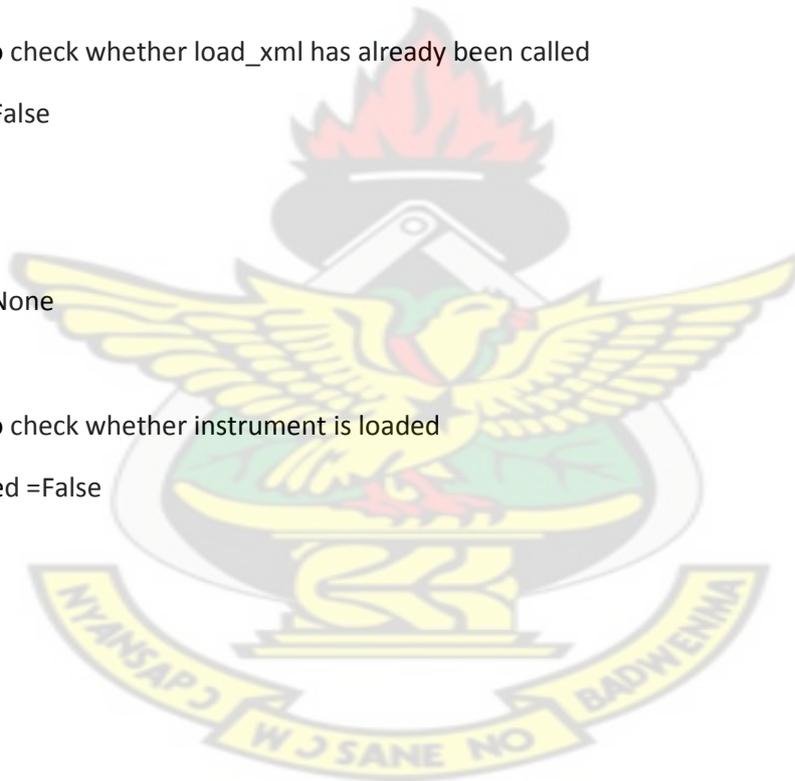
#boolean variable to check whether instrument is loaded
is_instrument_loaded = False

#title of page
title = None

def convert_to_int(str_value):
    float_value = float(''.join(map(str, str_value)))
    a_unit = 'Hz'

    if float_value >= 1000000:
        float_value = float_value / 1000000
        a_unit = 'MHz'

    elif float_value >= 1000:
```



```

float_value = float_value / 1000

a_unit = 'kHz'

else:

    pass

string_value = str(float_value)

count = len(string_value)
dummy_value = []
for i in range(count):
    dummy_value.append(string_value[i])

dummy_value.reverse()
count = len(dummy_value)

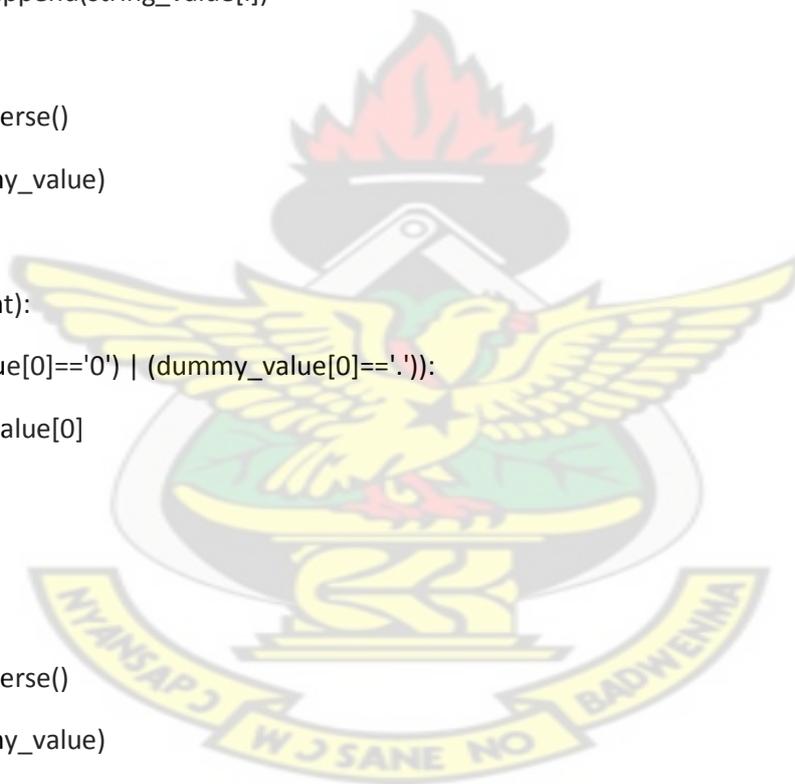
for i in range(count):
    if ((dummy_value[0]=='0') | (dummy_value[0]=='.')):
        del dummy_value[0]
    else:
        break

dummy_value.reverse()
count = len(dummy_value)
final_value = []

for i in range(count):
    if (dummy_value[i]=='.'):
        final_value.append(202)
    else:
        final_value.append(int(dummy_value[i]))

```

KNUST



```
return final_value, a_unit
```

```
def load_chart(request):
```

```
    global is_graph_Loaded, load_xml_thread, title, is_instrument_loaded
```

```
    #check to see if instrument has already been loaded
```

```
    if (is_graph_Loaded==False):
```

```
        register1 = 0
```

```
        value1 = 1
```

```
        register2 = 1
```

```
        value2 = 4
```

```
        is_instrument_loaded = False
```

```
        set_instrument(register1, value1, register2, value2)
```

```
        x_value = "X Axis"
```

```
        y_value = "Y Axis"
```

```
        title = "Amplitude over time curve"
```

```
        load_xml_thread = RepeatTimer(10,xml_file_write,args=[xml_file_path, title, x_value, y_value])
```

```
        load_xml_thread.start()
```

```
        is_graph_Loaded = True
```

```
    return render_to_response('chart.html',{'title': title})
```

```
def load_discrete(request):
```

```
    global is_graph_Loaded, is_instrument_loaded, load_xml_thread, title
```

```
    #check whether graph function has been called
```

```
    if (is_graph_Loaded==True):
```

```
        load_xml_thread.cancel()
```

```
is_graph_Loaded = False

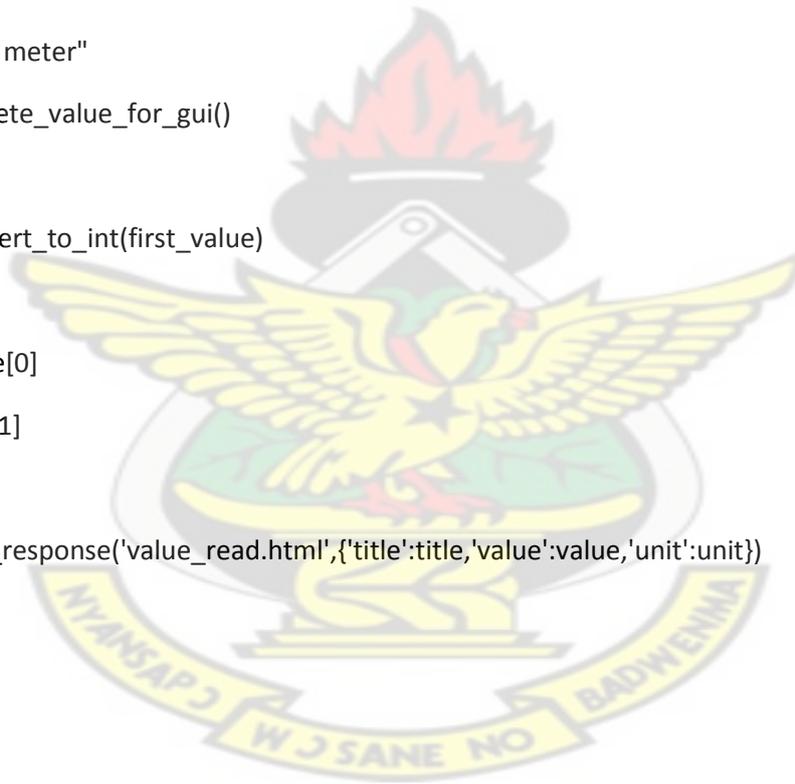
#check whether instrument is already loaded
if (is_instrument_loaded==False):
    register1 = 0
    value1 = 2
    register2 = 1
    value2 = 0
    set_instrument(register1, value1, register2, value2)
    is_instrument_loaded = True

title = "Frequency meter"
first_value = discrete_value_for_gui()

final_tuple = convert_to_int(first_value)

value = final_tuple[0]
unit = final_tuple[1]

return render_to_response('value_read.html',{'title':title,'value':value,'unit':unit})
```



```

#import os

from spp_fpga_protocol import set_ECP_ByteMode, read_parameters_reg, reg_table,
write_parameters_reg

from RepeatTimer import RepeatTimer

import numpy as np

import threading, time

#root_path = os.path.dirname(__file__)

#register table for graph values
graph_value_table = []

#register table for discrete values
discrete_value_table = []

#boolean variable to check if graph function has been loaded
isCalled = False

#thread for read_fpga_for_graph
graph_thread = None

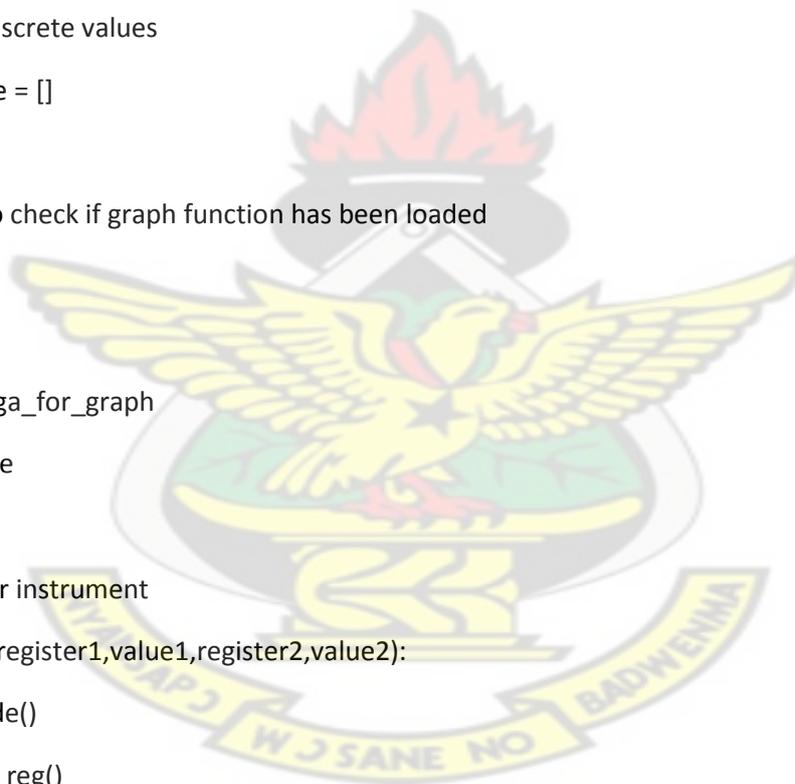
#set RVI to particular instrument
def set_instrument(register1,value1,register2,value2):
    set_ECP_ByteMode()
    read_parameters_reg()
    reg_table[register1] = value1
    reg_table[register2] = value2
    write_parameters_reg()

#=====

# Configuration for graph-generating instrument

```

KNUST



```
#=====
```

```
#Reads registers carry values for drawing a graph
```

```
#loads values into graph_value_table
```

```
def read_fpga_for_graph():
```

```
    global graph_value_table
```

```
    x_variable = np.arange(0,5,0.33)
```

```
    for i in range(len(x_variable)):
```

```
        set_ECP_ByteMode()
```

```
        read_parameters_reg()
```

```
        graph_value_table.append(reg_table[9])
```

```
        time.sleep(1)
```

```
#Creates XML file data.xml to be accessed by FusionChartsFree
```

```
def xml_file_write(a_file_path,graph_title,x_axis_label,y_axis_label):
```

```
    global isCalled, graph_thread
```

```
    if (isCalled==False):
```

```
        graph_thread = RepeatTimer(1,read_fpga_for_graph)
```

```
        graph_thread.start()
```

```
        isCalled = True
```

```
    count = len(graph_value_table)
```

```
    x_variable = np.arange(0,count,0.33)
```

```
    filename = open(a_file_path,"w") #open data.xml file
```

```
filename.write("\n<graph caption="{0}" xAxisName="{1}" yAxisName="{2}" showAnchors="1"
anchorRadius="1" showValues="0">'.format(graph_title,x_axis_label,y_axis_label))
```

```
for x in range(count):
```

```
    filename.writelines("\n <set name="{0:.03f}" value="{1:.03f}"
showName="0"/>'.format(x_variable[x],graph_value_table[x]))
```

```
filename.write("\n</graph>")
```

```
filename.close() #close data.xml file
```

```
#=====
```

```
# End of graph instrument configuration
```

```
#=====
```

```
#=====
```

```
# Configuration of discrete value-generating instrument
```

```
#=====
```

```
#Read register values needed
```

```
def read_fpga_for_discrete():
```

```
    global discrete_value_table, graph_value_table
```

```
    if graph_value_table:
```

```
        graph_value_table = []
```

```
    set_ECP_ByteMode()
```

```
    read_parameters_reg()
```

```
    discrete_value_table = reg_table[8:]
```

```
    discrete_value_table.reverse()
```

```
#process data retrieved into meaningful data
```

```
def discrete_value_for_gui():
```

```
    global discrete_value_table
```

```
    read_fpga_for_discrete()
```

```
    count = len(discrete_value_table)
```

```
    for i in range(count):
```

```
        if (discrete_value_table[0]!=0):
```

```
            break
```

```
        del discrete_value_table[0]
```

```
    discrete_value = discrete_value_table
```

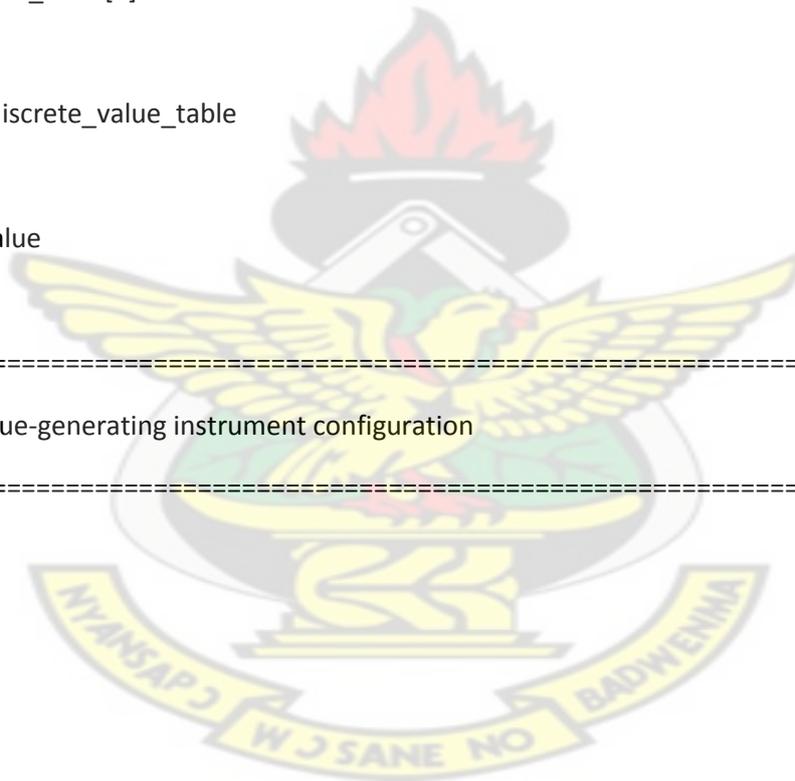
```
    return discrete_value
```

```
#####
```

```
# End of discrete value-generating instrument configuration
```

```
#####
```

KNUST



```
from django.db import models
```

```
# Create your models here.
```

```
class Instrument(models.Model):
```

```
    name = models.CharField(max_length=30)
```

```
    instrument_type = models.CharField(max_length=100)
```

```
    register_load1 = models.IntegerField()
```

```
    load_value1 = models.IntegerField()
```

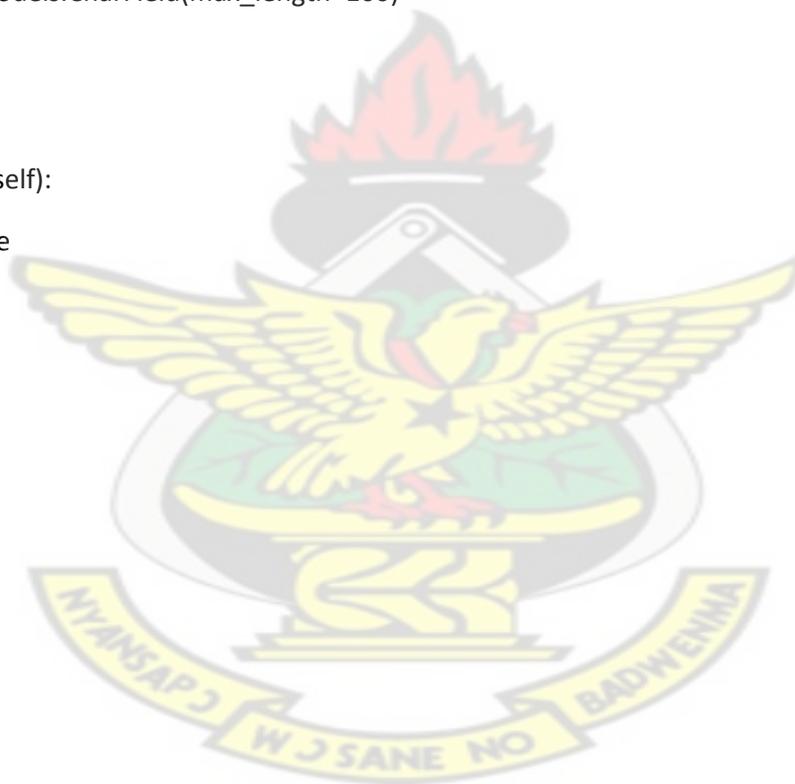
```
    register_load2 = models.IntegerField()
```

```
    load_value2 = models.IntegerField()
```

```
    register_read = models.CharField(max_length=100)
```

```
def __unicode__(self):
```

```
    return self.name
```



```
<!DOCTYPE html >
```

```
<html>
```

```
<head>
```

```
<title>{%block title%}Home - Reconfigurable Virtual Instrument{%endblock%}</title>
```

```
<link href="/static/css/style.css" rel="stylesheet" type="text/css" />
```

```
{%block scripts%}{%endblock%}
```

```
</head>
```

```
<body {%block onload%}{%endblock%}>
```

```
<div id="background">
```

```
<div id="page">
```

```
<div class="header">
```

```
<div class="footer">
```

```
<div class="body">
```

```
<div id="sidebar">
```

```
<a href="/"></a>
```

```
<ul class="navigation">
```

```
<li><a href="/">HOME</a></li>
```

```
Generator</a></li>
```

```
<li><a href="/chart">Waveform
```

```
{%block add_list%}{%endblock%}
```

```
Generator</a></li>
```

```
<li><a href="/discrete">Function
```

```
</ul>
```

```
</div>
```

```
<div id="content" >
    {%block content%}{%endblock%}
</div>
</div>
</div>
<div class="shadow">&nbsp;</div>
</div>
</div>
</div>
</body>
</html>
```

KNUST



```
{% extends "base.html" %}
```

```
{%block title%}{{title}}{%endblock%}
```

```
{%block scripts%
```

```
<script type="text/javascript" src="/static/Javascripts/jquery-1.4.2.js"></script>
```

```
<script type="text/javascript" src="/static/Javascripts/jquery.fusioncharts.js"></script>
```

```
{%endblock%}
```

```
{%block onload%}onload="loadGraph()"{%endblock%}
```

```
{%block add_list%
```

```
<li><a href="" onclick="loadGraph()">RELOAD</li>
```

```
{%endblock%}
```

```
{%block content%
```

Loading Graph... ..

```
<script type="text/javascript">
```

```
function loadGraph(){
```

```
$('#content').insertFusionCharts({
```

```
type: "Line2D", swfPath:"/static/FusionCharts/", width: "726", height: "546",
```

```
data: "/static/data.xml",
```

```
dataFormat: "URIData"
```

```
});
```

```
}
```

```
</script>
```

```
{%endblock%}
```

**value\_read.html**

```
{% extends "base.html" %}
```

```
{%block title%}{{title}}{%endblock%}
```

```
{%block scripts%
```

```
<script type="text/javascript" src="/static/Javascripts/jquery-1.4.2.js"></script>
```

```
<script type = "text/javascript">
```

```
    var reading_array = {{value}}
```

```
function load_images()
```

```
{
```

```
    for (var i =reading_array.length-1; i > -1; i--)
```

```
    {
```

```
        $('#content').prepend('')
```

```
    }
```

```
}
```

```
</script>
```

```
{%endblock%}
```

```
{%block onload%}onload="load_images()"{%endblock%}
```

```
{%block content%}{{unit}}{%endblock%}
```

KNUST

