Physics with Home-made Equipment and Innovative Experiments.

PHOENIX

Ajith Kumar B P, VVV Satyanarayana, Kundan Singh & Parmanand Singh

Inter-University Accelerator Centre, New Delhi 110 067

Pramode C E, IC Software, Trissur, Kerala

Contents

Chapter 1

Introduction

Experiments are an important part of learning science and the capability to perform them with some confidence in the results opens up an entirely new method of learning. Experimental data can be compared with the results from mathematical models to examine the fundamental laws governing various phenomena. Research laboratories around the world performing science experiments use various types of sensors interfaced to computers for data acquisition. They formulate hypotheses, design and perform experiments, mathematically analyze the data to check whether they agree with the theory. Unfortunately the data acquisition hardware used is too expensive for school and college laboratories where teaching is the main goal and not research.

PHOENIX project is a modular, flexible and inexpensive system for experiment control, data acquisition and analysis utilizing the power of the personal computers and micro-controllers. Phoenix is developed at Inter-University Accelerator Centre, an autonomous research institute of University Grants Commission, India, providing particle accelerator based research facilities to the Universities. This document provides and overview of the equipment, its operation at various levels of sophistication and several experiments that can be done using it.

1.1 The Hardware

A photograph of Phoenix is shown in figure 1.1. The connection to the PC can be made through the Serial/USB ports depending on the version of hardware used. The unit is powered by a 9V DC adapter but the USB version has a jumper option to run on the 5V supply of USB. The hardware schematic is shown in figure 1.2. The hardware features are listed below.

- 10 bit resolution ADCs, 4 channels
- Digital Outputs, 4 channels
- Digital Inputs, 4 channels

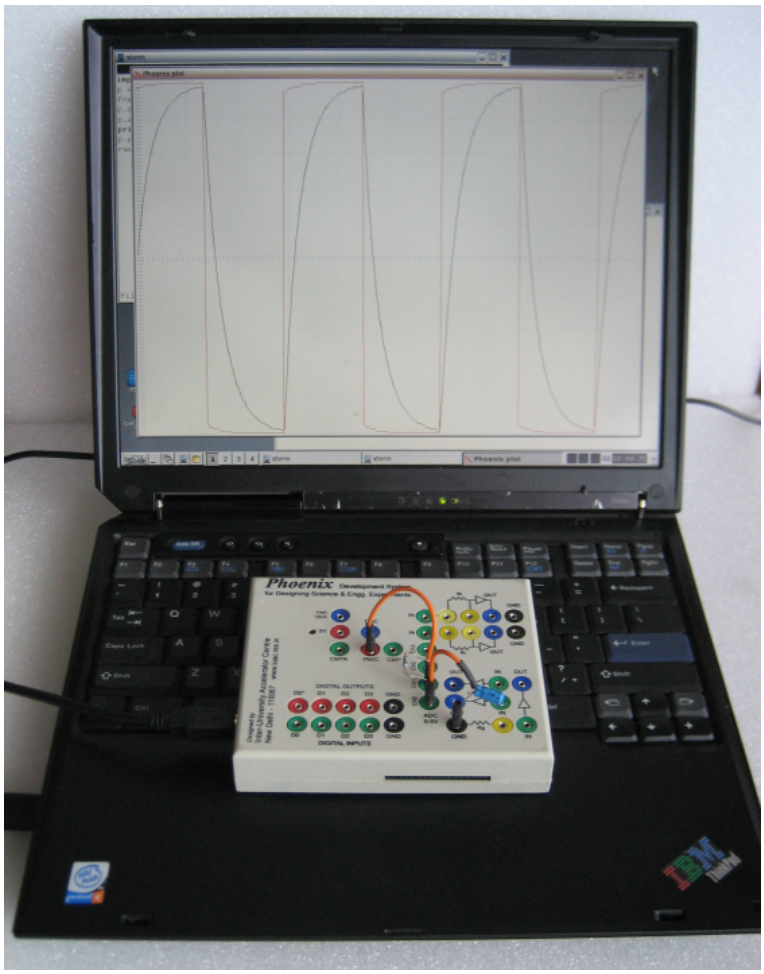


Figure 1.1: A Photograph of the Phoenix connected to USB port of a laptop PC showing RC integration of a square wave.

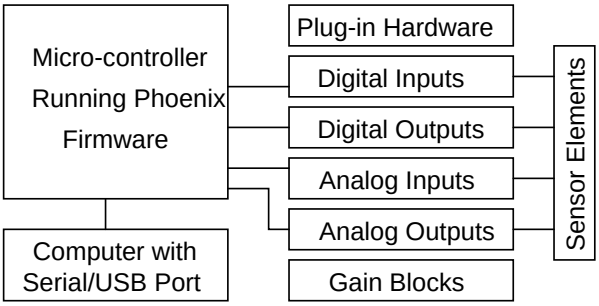


Figure 1.2: A schematic block diagram of Phoenix Hardware.

- 8 bit DAC, 1 channel
- Frequency Counter, up to 1 MHz
- Square wave Generator, up to 4 MHz
- 1 mA Constant Current Source
- Variable gain Amplifiers
- 5V regulated DC output
- Serial/USB interface to Computer

The analog inputs can digitize 0 to 5V range signals into a number ranging from 0 to 1023. The minimum conversion time is 10 microseconds. The voltage level changes at Digital Input/Output pins can be monitored/controlled with microsecond timing accuracy. Time intervals can be measured with micro-second resolution using the the Digital I/O sockets. The Analog output is implemented using a Pulse Width Modulated DAC having 8 bit accuracy. The control and sensor elements used in the experimental setup are connected to the I/O channels. The plug-in feature allows expanding the hardware capabilities. Since most of the sensor elements give small output voltages, variable gain Amplifiers are provided to make them into the 0 to 5V range before feeding to the ADC inputs. A one milli Ampere constant current source and a 5V regulated DC output are also available.

1.1.1 Accessories

Most of the experiments require some extra circuits connected to Phoenix. For example measuring the period of a pendulum uses a Light Barrier. The oscillating pendulum intercepts the light falling on a photo-transistor whose output is connected to one of the Digital Inputs for time interval measurement. Several such accessories with different levels of comnplexities have been developed and documented. Commercially available sensors for temperature, pressure, light etc. are used in making the accessories.

1.2 The Software

The role of the software is to communicate to the control/sensor elements of the experiment as per the instructions from the user. To make program development easier, we have followed a layered approach as outlined in the figure 1.3. The actual interaction with the hardware is done by the Firmware, written in C, running on the micro-controller, as per the commands from the PC through the serial link. The user programs communicate to the Python library to get the job done. Experimenter has the freedom to use the system at various levels of sophistication.

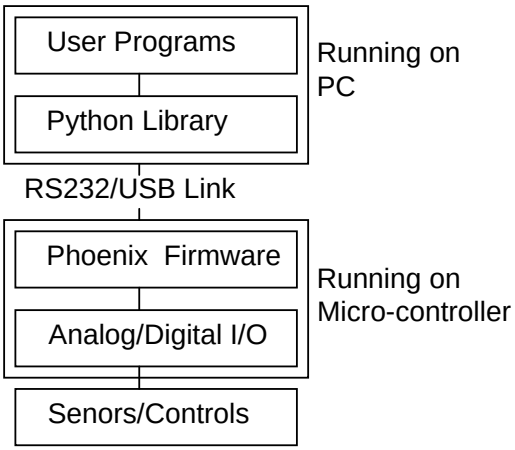


Figure 1.3: Schematic of the Software architecture of Phoenix.

1.2.1 Level 1: GUI Programs

Completed GUI programs are at the topmost level, where no programming is required. The user connects the sensors/control elements used in the experiment and collect the data using the GUI provided. Several such programs have been written for different experiments. GUI is useful for school level experiments where the teachers are not expected to do any programming. GUI programs are also written to make Phoenix to function as a low frequency storage oscilloscope, frequency counter, function generator etc.

1.2.2 Level 2: Writing Programs using the Python Library

This is the level meant for those who want to develop new experiments using Phoenix. The hardware features can be accessed by calling the functions from the Python library, described in chapter 4. Python is amazingly simple to learn with its clean syntax and already has a huge collection of libraries for scientific computation and graphics. Python can be used either from inside the interpreter or by running the python programs stored in files. Phoenix can be accessed using single line commands, as illustrated below.

```
import phm
p = phm.phm()
v = p.read_block(200, 20, 1)
p.plot(v)
```

This program samples the voltage signal connected to the ADC input 200 times, with a 20 microseconds interval between samples. The acquired data plotted and a screen-shot of the program output is shown in figure 1.4. Similarly there are

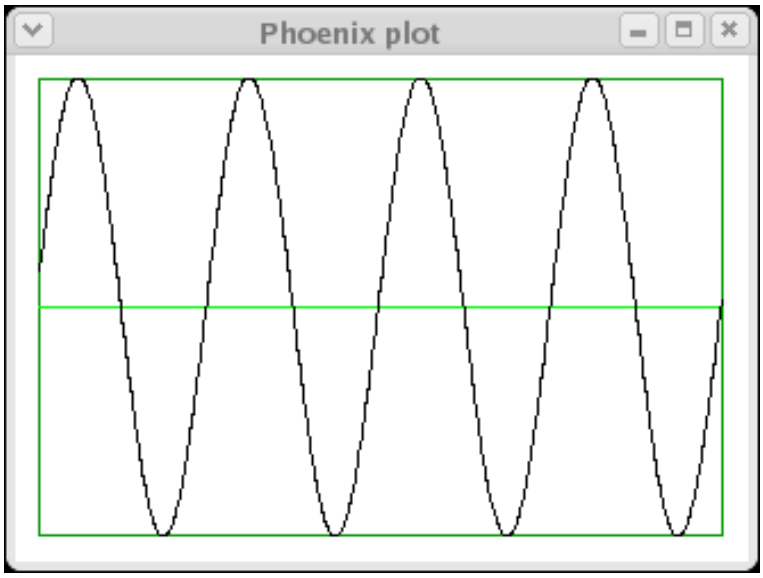


Figure 1.4: Plot of a 1 KHz sine wave signal connected to Phoenix ADC input.

functions for setting/reading the voltage level of Digital Input/Outputs and for measuring the time interval between level transitions on Digital Input Sockets.

Phoenix can be used with any computer or operating system having a Python Interpreter and a Python module to communicate to Serial/USB port. We recommend Free Software platforms like GNU/Linux since they come with non-restrictive licenses, better suited for academic work where sharing of knowledge is encouraged. Phoenix can also be used just by booting from the live CD which contains all the required software. You can also install it to your existing operating system.

1.2.3 Level 3: Micro-controller programming and making Stand-alone systems

This level is of interest to engineering students who wish to develop micro-controller based projects. Phoenix is designed around an AVR series micro-controller, ATmega16, from Atmel. It provides enough processing power and there is a free cross-compiler available. The Phoenix hardware functions as a micro-controller development system. One can write programs in C language, compile it using the cross-compiler of AVR, and upload the binary to ATmega16 chip through a cable connected to the parallel port of the PC. A LCD display is provided for easy program development. Several examples on how to convert Phoenix hardware into stand-alone devices like temperature monitor, frequency counter, rolling display etc. has been given in the documentation [5].

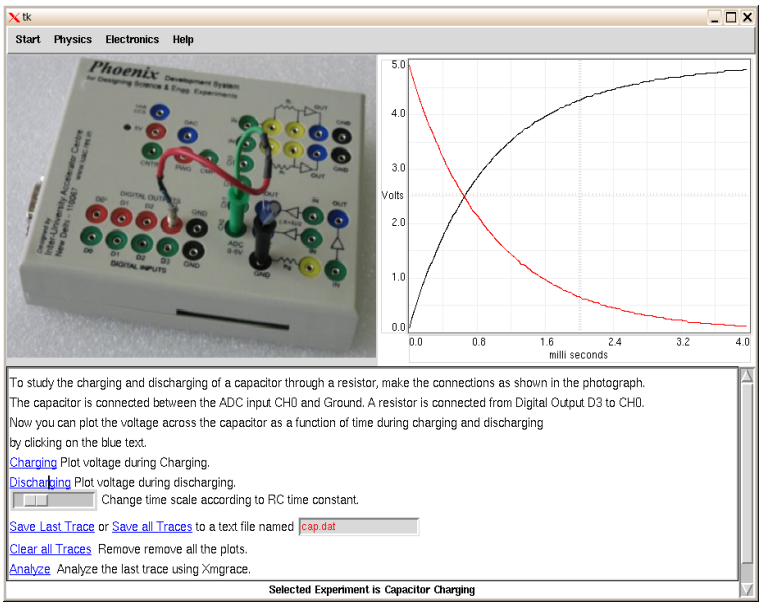


Figure 1.5: Screen-shot of the program showing the voltage across a capacitor during charging and discharging through a resistor. The program shows a photograph of the connections to be made.

1.3 Developing Science Experiments

Any science experiment involves control and measurement of physical parameters like temperature, pressure, voltage etc., as a function of time in most of the cases. The parameters to be measured need to be converted into electrical signals, which is done by appropriate sensor elements. The sensor outputs are processed and connected to the input sockets of Phoenix. Some experiments in electricity and electronics can be carried out without using any sensors since they directly generate electrical signals. The method can be illustrated better using the example given below. The details of more experiments are given in chapter 3 and at www.iuac.res.in, the phoenix website.

1.3.1 Charging and Discharging of a Capacitor

If a positive voltage step is applied to a capacitor through a resistor, the capacitor will start charging and the voltage across the capacitor will rise exponentially. This is taught in the theory classes but experimentally proving it require a fast digitizer and other electronics circuits. To do this using Phoenix, we connect a resistor from one of the Digital Outputs to an Analog input and the Capacitor from there to ground. Through software, we can change the voltage level at the Digital Output and then capture the resulting waveform across the capacitor by reading the ADC input channel CH0. Figure 1.5 shows a screenshot of the GUI

program doing this experiment. All we have done to design this experiment is to combine the Digital Output and Analog Input features of Phoenix.

1.4 Objectives and present status

The usage of computers for experiment control, data acquisition and analysis at the research level is well established. Several aspects needs to be considered while using the same for teaching science. Commercially available solutions does not allow the user to explore the internals of the system. What we have designed is a system that can be used at different levels depending on the programming and electronics expertise of the user. With this approach we hope to train teachers to use and later develop computer interfaced experiments based on their own ideas. Tight integration between data collection and analysis has been avoided. The stress is on getting the data into the computer with minimum amount of effort. The ability of the system to function as a micro-controller development may be useful for engineering students and student projects leading to equipment for physics experiments is an expected possibility. Phoenix can be used as a starting point for designing stand-alone systems like temperature controller, weather monitoring station, frequency counter etc.

The project started in 2005 by designing a parallel port based interface and several experiments using it. Feedback from the academic community has been instrumental in deciding the direction. The initial versions used C language for developing user programs but a Python option [3] was added later. The micro-controller version was done in 2006 to make the hardware more compact, expandable and less costly. The new version runs on any computer or operating system having Serial/USB ports and Python language support. Several universities have shown interest in using Phoenix and currently more than four hundred pieces are in circulation.

IUAC tries to make the system available to the maximum number of users. We conduct one day workshops at different places to demonstrate the system to physics teachers. Training programs are conducted at IUAC periodically for physics teachers from the universities. The hardware can be manufactured royalty free and we already have several vendors selling the hardware at a very low price. A live CD has been made to run the system without installing any software to the hard disk. A mailing list has been setup for user interaction. Complete details of the project are available on-line[1].

Chapter 2

How it Works

2.1 The hardware

The most common Input/Output devices for a computer are Keyboard, Mouse, Monitor and Printer. However, you cannot feed a voltage signal to a computer for measuring its properties. Phoenix provides precisely this capability to your computer. It acts as a gateway to your computer through which Analog and Digital signals can travel. The analog signal are converted into digital form by Phoenix.

As you can see from figure 2.1, the sockets on the top panel are grouped into different categories. You can connect external world signals to these sockets and Control/Monitor them using simple function calls written in Python Language. The major features are listed below

- Digital Inputs (Read the voltage level through software)
- Digital Outputs (Set the voltage level to 0 or 5V using software)
- Analog Inputs (Measure the value of an applied voltage)
- Analog Output (Generates a voltage under software control)
- Time interval measurements between voltage level changes
- Frequency counter
- Square wave generator
- Constant current source
- Amplifiers whose gain can be set using resistors

The functions of the various groups of top panel sockets are briefly explained below.

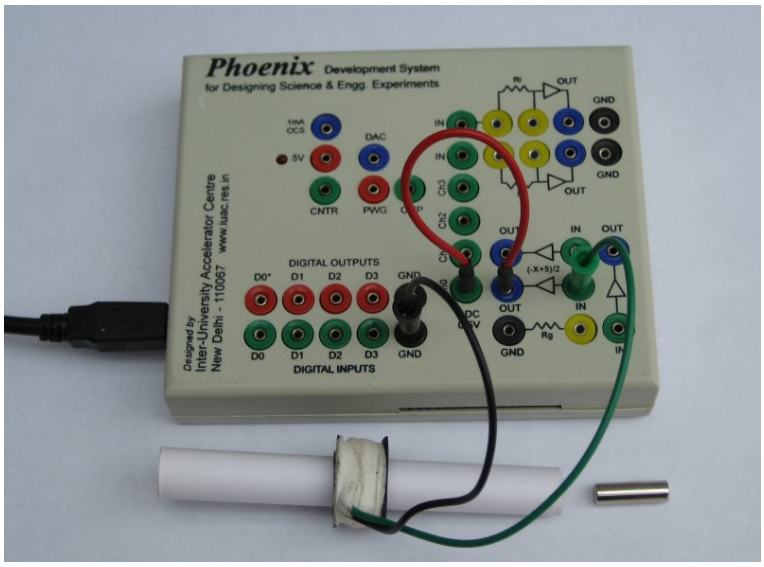


Figure 2.1: The top panel of Phoenix Interface. A coil is connected to one of the analog inputs through a level shifting amplifier.

1. 5V OUT - This is a regulated 5V power supply that can be used for powering external circuits. It can deliver only upto 100mA current , which is derived from the 9V unregulated DC supply from the adapter.

2. Digital outputs - four RED sockets at the lower left corner . The socket marked $D0^*$ is buffered with a transistor; it can be used to drive 5V relay coils. The logic HIGH output on D0 will be about 4.57V whereas on D1, D2, D3 it will be about 5.0V. D0 should not be used in applications involving precise timing of less than a few milli seconds.

3. Digital inputs - four GREEN sockets at the lower left corner. It might sometimes be necessary to connect analog outputs swinging between -5V to +5V to the digital inputs. In this case, you MUST use a 1K resistor in series between your analog output and the digital input pin.

4. ADC inputs - four GREEN sockets marked CH0 to CH3

5. PWG - Programmable Waveform Generator

6. DAC - 8 bit Digital to Analog Converter output

7. CMP - Analog Comparator negative input, the positive input is tied to the internal 1.23 V reference.

8. CNTR - Digital frequency counter (only for 0 to 5V pulses)

9. 1 mA CCS - Constant Current Source, BLUE Socket, mainly for Resistance Temperature Detectors, RTD.

10. Two variable gain inverting amplifiers, GREEN sockets marked IN and BLUE sockets marked OUT with YELLOW sockets in between to insert resistors. The amplifiers are built using TL084 Op-Amps and have a certain offset which has to be measured by grounding the input and accounted for when making precise measurements.

11. One variable gain non-inverting amplifier. This is located on the bottom right corner of the front panel. The gain can be programmed by connecting appropriate resistors from the Yellow socket to ground.

12. Two offset amplifiers to convert -5V to +5V signals to 0 to 5V signals. This is required since our ADC can only take 0 to 5V input range. For digitizing signals swinging between -5V to +5V we need to convert them first to 0 to 5V range. Input is GREEN and output is BLUE.

To reduce the chances of feeding signals to output sockets by mistake, the following Color Convention is followed.

- GREEN - Inputs, digital or analog
- RED - Digital Outputs and the 5V regulated DC output
- BLUE - Analog Outputs
- YELLOW - Gain selection resistors
- BLACK - Ground connections

2.2 The Software

To access the Phoenix hardware, programs running on the PC should communicate to the micro-controller inside Phoenix. This is done over RS232 or USB interface, depending on the model used. The communication is handled by a Python library. User programs call simple functions to access various features of Phoenix. In order to use Phoenix, your computer must have the following software installed:

1. The Python interpreter.[1]
2. Pyserial module to access the RS232 port from Python.
3. PyUSB to access the USB ports.

[1] Strictly speaking, any program having the ability to talk to the USB /Serial port can access Phoenix. A C library is also available on the CD but it is incomplete.

If you are running from the Phoenix liveCD, all these things are available and ready to use. If you are running some other GNU/Linux distribution you need to install the Pyserial and PyUSB modules from the files provided on the Phoenix CD. You need to copy the Phoenix library 'phm.py' from the directory 'phoenix/software/interface' to the 'site-packages' directory inside the Python home directory [2]. All the software required to run Phoenix under MSWindows is located inside a directory named *winPython* on the CD. Install all of them, by clicking on the icons.

2.3 Communicating to Phoenix

You can use Python interpreter in two different ways. Start the Python interpreter and type the commands from it is one option. The '>>>' is the Python command prompt.

$ python

>>> import phm
>>> p = phm.phm()

Perhaps an easier option is to type your code into file using a Text Editor and invoke the interpreter with your program name as the first argument:

$ python mycode.py

Every Phoenix program should have the following two lines in the beginning

import phm
p = phm.phm()

The first line loads the Phoenix library called 'phm'. The second line invokes the function phm() from from the library, that returns an object of a class named 'phm'. All the functions to access Phoenix is inside this class. We call them by prefixing the object name, for example

```
print p.read_inputs()
```

prints the status of the Digital Inputs.

The Python library to communicate to the hardware is explained in 4. You need them to develop new experiments. In the next chapter, we discuss some of the experiments that has already been developed and code is written.

[2] On most systems this will be /usr/lib/python2.x, where x is the version number

Chapter 3

Experiments

Several experiments on electricity and electronics can be done without much extra accessories. Science experiments require sensor elements that converts physical parameters into electrical signals. The number of science experiments one can do with Phoenix is limited mainly by the availability of sensor elements. Here we describe several experiments that can be done using some sensor elements that are easily available.

Almost all the experiments described below can be done using Python program named *Experiments*, having a Graphical User Interface. For better understanding of the internal working , one can also carry out them using the simple programs listed below.

3.1 A sine wave for free - Power line pickup

There are two types of electric power available ,generally known as AC and DC power. The Direct Current or DC flows in the same direction and is generally made available from battery cells. The electricity coming to our houses is Alternating Current or AC, which changes the direction of flow continuously. What is the nature of this direction change. The frequency of AC power available in India is 50 Hz. Let us explore this using Phoenix-M and a piece of wire. A frequency of 50 Hz means the period of the wave is 20 milliseconds. If we capture the signal for 100 milliseconds there will be 5 cycles during that time interval. Let us digitize 200 samples at 500 microsecond intervals and analyze it.

Connect one and of a 25 cm wire to the Ch0 input of the ADC and let the other end float. The 50 Hz signals picked up by the ADC can be displayed as a function of time by the *read_block()* and *plot_data()* functions. With few lines of code you are making a simple CRO !

Type in the following program in a text editor [1] and save it as a file named

[1] if you are not familiar with standard GNU/Linux editors like vi or emacs, you can use 'Nedit', which is available from the start menu of the Live-CD. Notepad under MSWindows

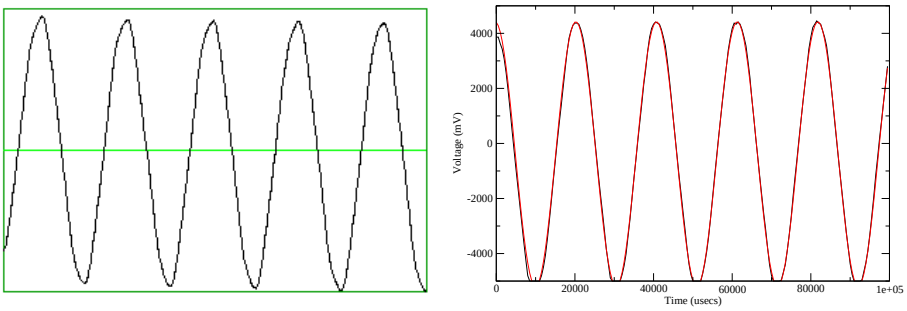


Figure 3.1: Power line Pickup

say 'pickup.py'.

```
import phm
p = phm.phm()
p.select_adc(0)
while p.read_inputs() == 15:
    v = p.read_block(200, 500,1)
    p.plot_data(v)
p.save_data(v, 'pickup.dat')
```

Run the program (by typing 'python pickup.py' at the Operating System command prompt) after plugging one end of 25 cm wire to Ch0 of the ADC. Make sure that none of the digital input pins are grounded. You should see a waveform similar to that of figure 3.1. Adjust the position of the wire or touch the floating end with your hand to see the changes in the waveform.

How do you terminate the program? The 'while' loop is continuously reading from the digital inputs and checking whether the value is 15 - if none of the sockets D0 to D3 are grounded, the value returned by read_inputs() will definitely be 15 and the loop body will execute. If you ground one of the digital inputs, the value returned by read_inputs will be something other than 15; this will result in the loop terminating. Terminate the program when a good trace is on the screen, last sample collected is saved to the disk file 'pickup.dat' just before exiting.

You can do the same selecting *Explore* from the menubar of Program *Experiments*. For capyuring the waveform and doing mathematical analysis, use the program named 'Analog Box'.

3.1.1 Mathematical analysis of the data

By counting the number of waves within a given time interval one can roughly figure out the frequency of the line pickup but it won't be accurate and don't tell us much about the nature of the wave. Let us approach the problem in a more systematic manner. We have measured the value of the voltage at 200

different instances of time and want to find out the function that governs the time dependency of the voltage. Problems of this class are solved by fitting the experimental data with some mathematical formula provided by the theory governing the physical phenomena under investigation. Curve fitting is a method of comparing experimental results with a theoretical model.

Here the theoretical value of voltage as a function of time is given by a sinusoidal wave represented by the equation $V = V_0 \sin 2\pi f$, where V_0 is the amplitude and f is the frequency. The experimental data can be 'fitted' using this equation to extract these parameters. We use the two dimensional plotting package *xmgrace* for plotting a fitting the data. Xmgrace is free software and included on the CD along with user manual and a tutorial. Xmgrace is started from the command prompt with file 'pickup.dat' , saved by the python program, as the argument.

```
# xmgrace pickup.dat
```

Select $Data->Transformation->NonLinearCurveFitting$ from the main menu and enter the equation $V(t) = V_0 \sin(2\pi ft/1000000.0 + \theta) + C$. Xmgrace accepts the adjustable parameters as A0, A1 etc.

- $V(t)$ is the value of voltage at time $= t$
- $V_0$is the amplitude. The value will be close to 5000 milli volts (represented by parameter A0)
- f is the frequency of the wave (parameter A1)
- t is the value of time, divided by 1000000 to convert micro seconds to seconds
- $\theta$is the phase offset since we are not starting the digitization at zero crossing (parameter A2)
- C is the amplitude offset that may be present (parameter A4)

Reasonable starting values should be given by the user for V_0 and f to guide the fitting algorithm. Try different values until you get a good fit. The figure **??** shows the data plotted along with the fitted curve. The Curve fitting window 3.2 shows the parameter values.

The extracted value of frequency is 48.73 Hz ! Did not believe it and cross checked it by feeding a 50 Hz sine wave from a precision function generator to the ADC input. The result of a similar analysis gave 49.98 Hz. Checked with the power distribution people and confirmed that the line frequency was really below 49 Hz.

Exercise: Repeat the experiment by changing the length of the wire, touching one end by your hand and rising the other hand, moving it near any electrical equipment etc. (do not touch any power line). You can also analyze other wave forms if you have a signal generator.

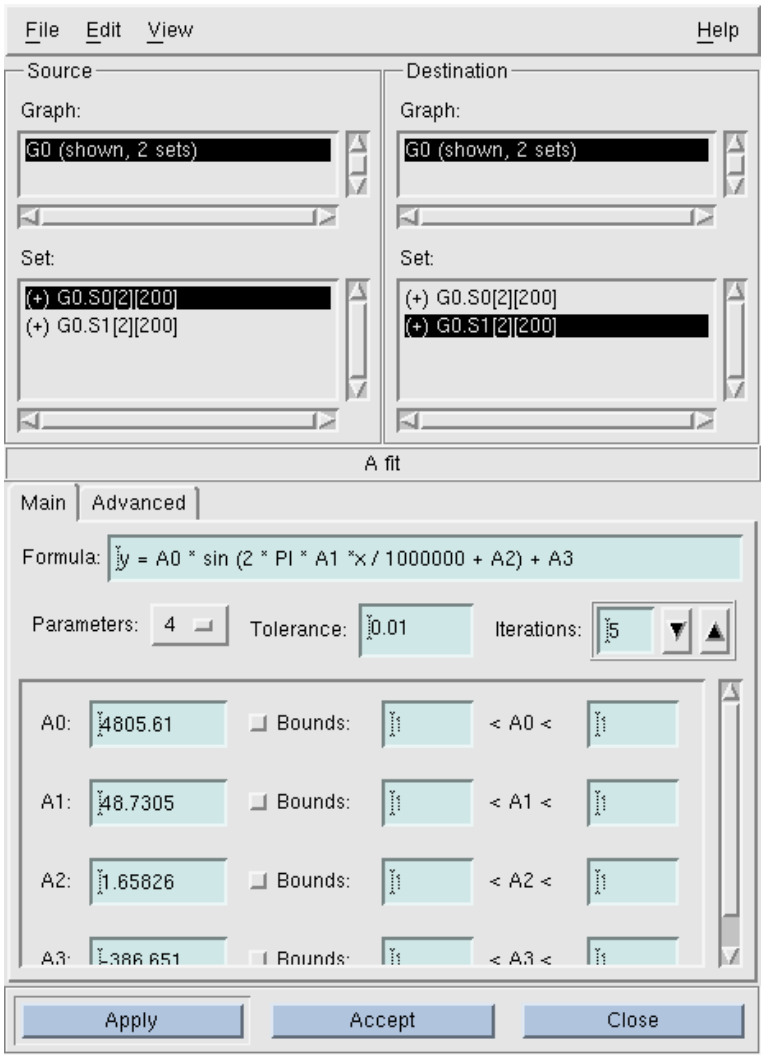


Figure 3.2: Curve fitting window of *xmgrace*

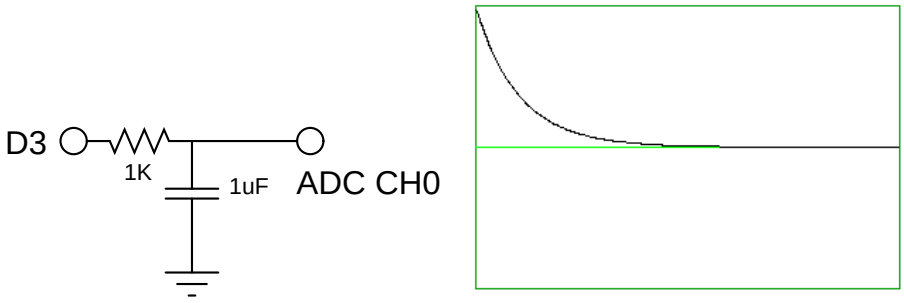


Figure 3.3: (a)Circuit to study Capacitor. (b) The discharge curve.

3.2 Capacitor charging and discharging

Every student learning about electricity knows that a capacitor charges and discharges exponentially but not very many has seen it doing so. Such experiments require fast data acquisition since the entire process is over within milli seconds. Let us explore this phenomena using Phoenix. All you need is a capacitor and a resistor.

Refer figure 3.3 for the experimental setup. The RC circuit under study is connected between the Digital output socket D3 and Ground. The voltage across the capacitor is monitored by the ADC channel 0. The voltage on D3 can be set to 0V or 5V under software control. Taking D3 to 5V will make the capacitor charge to 5V through the resistor R and then taking D3 to 0V will cause it to discharge. All we need to do is digitize the voltage across C just after changing the output of D3. Let us study the discharge process first. The python program *cap.py* listed below does the job.

```
import phm, time
p = phm.phm()
p.select_adc(0)
p.write_outputs(8)
time.sleep(1)
p.enable_set_low(3)
data = p.read_block(200,50, 0)
p.plot_data(data)
raw_input()         # wait for keypress
```

We make the digital output pins go high and sleep for 1 second (allowing the capacitor to charge to full 5V). The call to function *p.enable_set_low(3)* is similar to *select_adc()* or *add_channel()* ,whose effect is seen only later, when a read_block or multi_read_block is called. The idea is this - in certain situations, an ADC read should begin immediately after a few digital outputs are set to 1 or 0 - so we can combine the two together and ask the ADC read functions themselves to do the 'set to LOW or HIGH' and then start reading. In this

case, it brings to logic LOW pin D3, thereby starting the capacitor discharge process. The function then starts reading the voltage across the capacitor applied on ADC channel 0 at 250 microsecond intervals [2]. The voltage across the capacitor as a function of time is shown in Figure 3.3(b), which looks like an exponential function. When the rate of change of something is proportional to its instantaneous value the change is exponential.

Let us examine why it is exponential and what is an exponential function with the help of some elementary relationships.

The discharge of the capacitor results in a current I through the resistor and according to Ohm's law $V = IR$.

Voltage across the capacitor at any instant is proportional to the stored charge at that instant, $V = Q/C$.

These two relations imply $I = \frac{Q}{RC}$ and we current is nothing but the rate of flow of charge, $I = \frac{dQ}{dt}$.

Solving the differential equation $\frac{dQ}{dt} = \frac{Q}{RC}$ results in $Q(t) = Q_0 e^{-\frac{t}{RC}}$ which also implies $V(t) = V_0 e^{-\frac{t}{RC}}$

Exercise: Modify the python program to watch the charging process. Change the code to make D3 LOW by calling p.write_outputs(0) and set it to HIGH just before digitization with p.enable_set_high(3). Extract the RC value by fitting the data using the equation using xmgrace package.

Dielectric constant of materials can be calculated by fabricating capacitors and measuring the capacitance. The experimental setup used is shown in figure 3.4.

3.2.1 Linear Charging of a Capacitor

Exponential charging and discharging of capacitors are explained in the previous section. If we can keep the current flowing through the resistor constant, the capacitor will charge linearly. Let's wire up the circuit shown in Figure 3.5 When D3 is HIGH no charging occurs. When D0 goes LOW the capacitor starts charging through the 1mA constant current source.

and run the following Python script:

```
import phm, time
p = phm.phm()
p.enable_set_low(3)
p.write_outputs(8)
time.sleep(1)
```

[2] You may wonder as to why such a seemingly complicated function like enable_set_low is required. We can as well make the digital output pin go high by calling write_outputs and then call read_block. The problem is that all these functions communicate with the Phoenix box using a slow-speed serial cable. For example, the read_block function simply sends a request (which is encoded as a number) over the serial line asking the micro-controller in the Phoenix box to digitize some input and send it back. By the time this request reaches the micro-controller over the serial line, the capacitor would have discharged to a certain extend! So we have to instruct the Phoenix micro-controller in just ONE command to set a pin LOW and then start the digitization process. For more details refer to secion 4.5.5.4

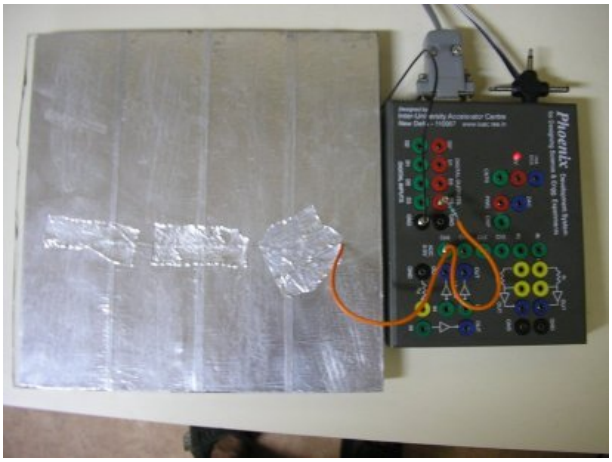


Figure 3.4: Dielectric constant of glass from capacitance

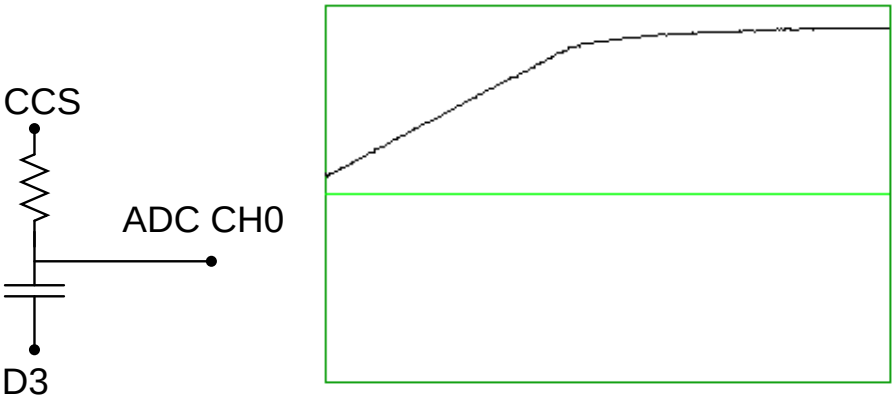


Figure 3.5: (a) Circuit for Linear Charging of Capacitor. (b) The voltage waveform

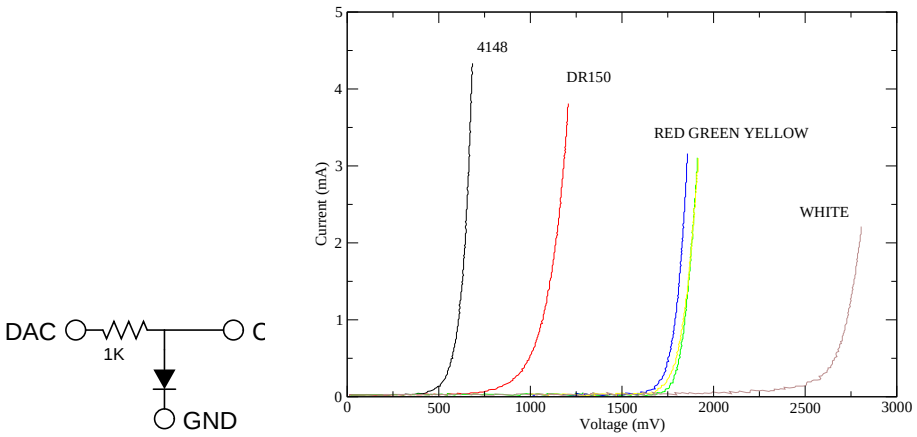


Figure 3.6: (a)Circuit for Diode IV Characteristic. (b) VI curve of several diodes.

```
v = p.read_block(400, 20, 0)
p.plot_data(v)
raw_input()  # wait for keypress
```

You will obtain a graph like the one shown in Figure 3.5.

3.3 IV Characteristics of Diodes

Diode IV characteristic can be obtained easily using the DAC and ADC features. The circuit for this is shown in figure 3.6(a). Connect one end of a 1 KOhm resistor to the DAC output. The other end is connected to the ADC Ch0 Input. Positive terminal of the diode also is connected to the ADC Ch0 and negative to ground. The Voltage across the diode is directly measured by the ADC and the current is calculated using Ohm's law since the voltage at both ends of the 1 KOhm resistor is known.

We have tried to study different diodes including Light Emitting Diodes with different wavelengths. The code 'iv.py' is ran for each diode and the output redirected to different files. For example; 'python iv.py > red.dat' after connecting the RED LED. The code 'iv.py' is listed below.

```
import phm, time
p=phm.phm()
p.set_adc_size(2)
p.set_adc_delay(200)
va = 0.0
while va <= 5000.0:
   p.set_voltage(va)
   time.sleep(0.001)
```

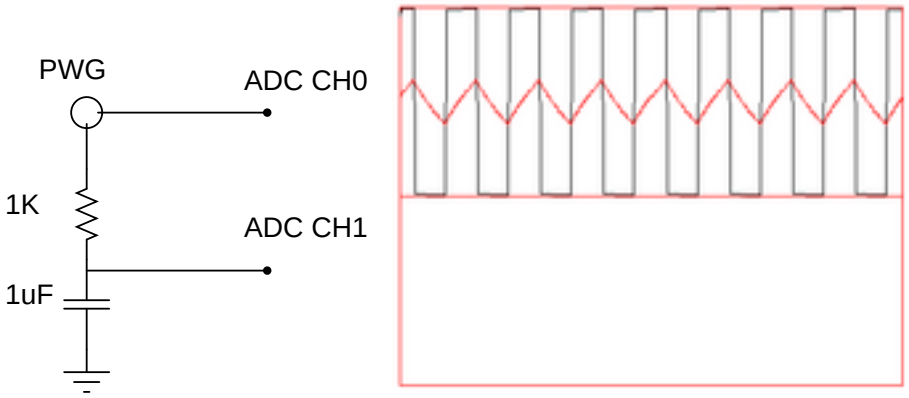


Figure 3.7: (a) RC Integration circuit (b) Result with 1 KHz square wave

```
vb = p.zero_to_5000()[1]
va = va + 5000.0/255
print vb, ' ' , (va-vb)/1000.0
```

The program output is redirected to a file and plotted using the program 'xmgrace', by specifying all the data files as command line arguments. The output is shown in the figure 3.6(b). Note the difference between different diodes. If the frequency of the LEDs are known it is possible to estimate the value of Plank's constant from these results.

3.4 Mathematical operations using RC circuits

RC circuits can be used for integration and differentiation of waveforms with respect to time. For example a square-wave of a particular frequency can be integrated to a triangular wave using proper RC values. In this experiment, we will apply a square wave (produced by the PWG) to CH0 of the Phoenix ADC. We will apply the same signal to an RC circuit (R=1K, C=1uF) and observe the waveform across the capacitor. The circuit is shown in figure 3.7(a) .We will repeat the experiment for 3 different cases by varying the Period of the square wave to show the different results.

1. RC $\approx$T , Results in a Triangular wave form3.8(b)
2. RC $>>$ T, The result is a DC level with some ripple 3.8(a)
3. RC $<<$ T, The sharp edges becomes exponential. 3.8(b)

The code 'sqintegrate.py' which generated these three plots is as follows:

```
"""data was taken with 1K resistor, 1uF capacitor
```

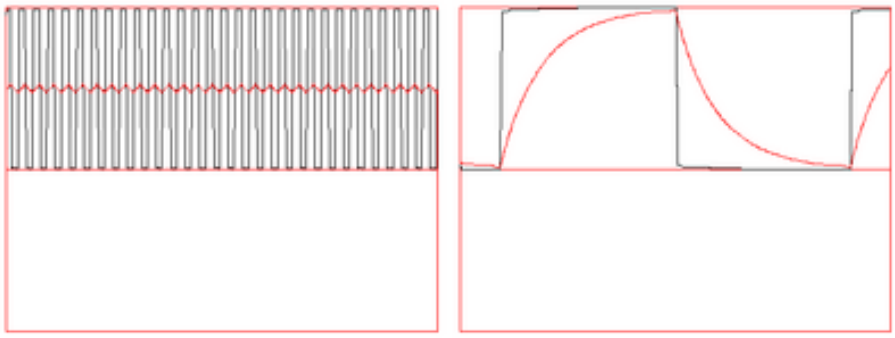


Figure 3.8: (a) RC > T (b) RC < T

```
Three sets are taken:
a) freq=1000 Hz and sampling delay = 10micro seconds, samples=400
b) freq=5000 Hz and sampling delay = 10micro seconds, samples=300
c) freq=100 Hz sampling delay = 20micro seconds, samples=300
"""
import phm
p = phm.phm()
freq = 1000
samples = 300
delay = 10
p.add_channel(0)
p.add_channel(1)
print p.set_frequency(freq)
while p.read_inputs() == 15:
   p.plot_data(p.multi_read_block(samples, delay, 0 ))
```

Run the code by changing the frequency to study the relation between RC and T

3.5 Electric field produced by a changing magnetic field

A voltage will be generated across a conductor kept in a changing magnetic field. We can create a change in the magnetic field by moving a permanent magnet near it. The magnitude of the voltage is decided by the length of the conductor and the rate of change of magnetic field. That means, to get larger voltage you can

- increase the length of the conductor
- use a stronger magnet

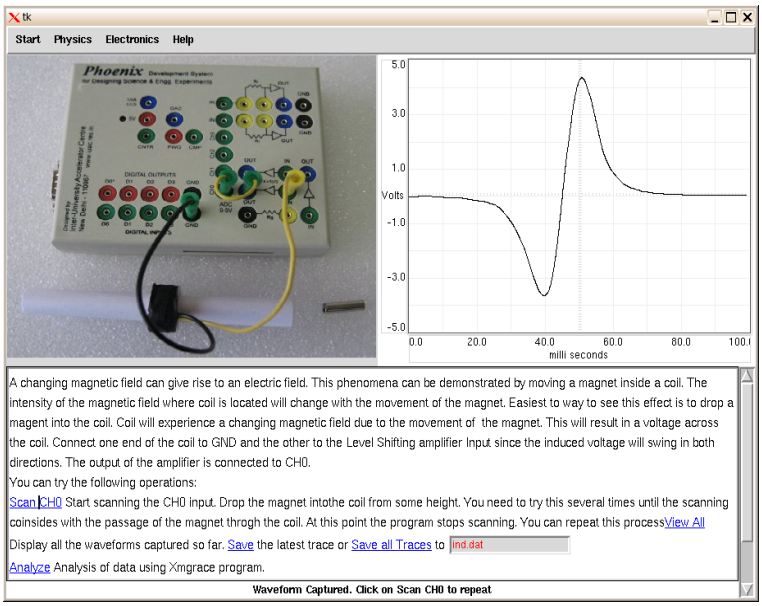


Figure 3.9: Experiment to demonstrate electromagnetic induction. The wave form induced on a solenoid by moving a magnet through it is captured using Phoenix.

- move the magnet faster

In this experiment we will drop a small magnet into a coil having 5000 turns of insulated copper wire. The coil is connected between ground and the input of the level shifter, and the level shifter output to ADC input Ch0. We need the level shifter since the induced voltage can go to negative values depending upon the direction of motion. From the physics menu select Electromagnetic Induction. A screen as shown in figure will come up. Clicking on the *Scan CH0* tag inside the text will tell the program to scan for the induced voltage, at regular intervals. Now drop the magnet into the coil. If the movement of the magnet through the coil coincides with the scanning, the waveform is captured and a message is displayed. You need to repeat dropping the magnet until this happens. If the waveform similar to the one shown in figure 3.9 is not obtained, click on *Scan CH0* to repeat the process.

The induced voltage first builds up in one direction and then in the opposite direction. How do we explain this waveform. According to Faraday's law, the induced voltage across a coil with N turns is given by

$$\varepsilon = -N\frac{d\Phi_B}{dt}$$

where $d\Phi_B/dt$ is the rate of change of magnetic flux intercepted by the coil. From figure 3.10, it can be seen that the direction of the magnetic field is radially

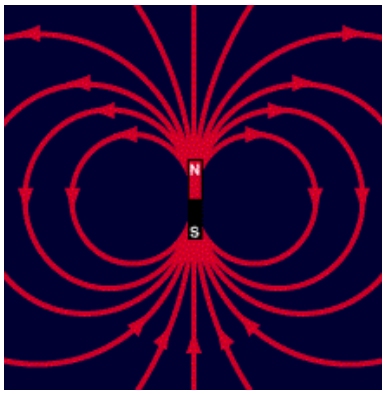


Figure 3.10: Magnetic field of a cylindrical magnet.

outwards near the north pole. Imagine the magnet entering the coil with the north pole first. The induced voltage goes up due to the effect of the outgoing flux from the north pole. Once the magnet enters inside the coil, one half of the coil will intercept a magnetic field in the outward direction and the other half a field in the inward direction. This results in a zero induced voltage when the magnet is in the middle of the coil. Once the magnet moves out of the coil through the opposite side the contribution from the south pole dominates and we find an induced voltage of opposite polarity.

Since the induced voltage is proportional to the rate of change of magnetic field, the amplitude of the waveform will increase with the velocity of the magnet. This can be verified by dropping the magnet from different heights.

3.5.1 Mutual Induction

In the previous experiment, the changing magnetic field was generated by a moving magnet. We can create a varying magnetic field by changing the current flowing through a coil, by connecting it to an AC source. We can generate a sinusoidal voltage on the DAC socket. For that select *Mutual Induction* from the *physics* menu. The program will display a photograph of the connections to be made. The DAC socket is programmed to generate a 50 Hz sine wave ranging from 0 to 5V. Passing it through a series capacitor will make amplitude ranging from -2.5 to +2.5V. This is given to the primary coil and also to the level shifter input for monitoring. The secondary coil is connected between ground and the input of the other level shifter. The level shifter outputs are connected to CH0 and CH1.

At this point you should see the sinusoidal primary waveform on the screen. The secondary waveform will be a horizontal line. Now bring the coils closer. Once you make the magnetic circuit using the ferrite core, the secondary output also will be displayed as shown in figure 3.11. Try reversing the mutual orientation of the coil to see its effect on the phase difference between primary and

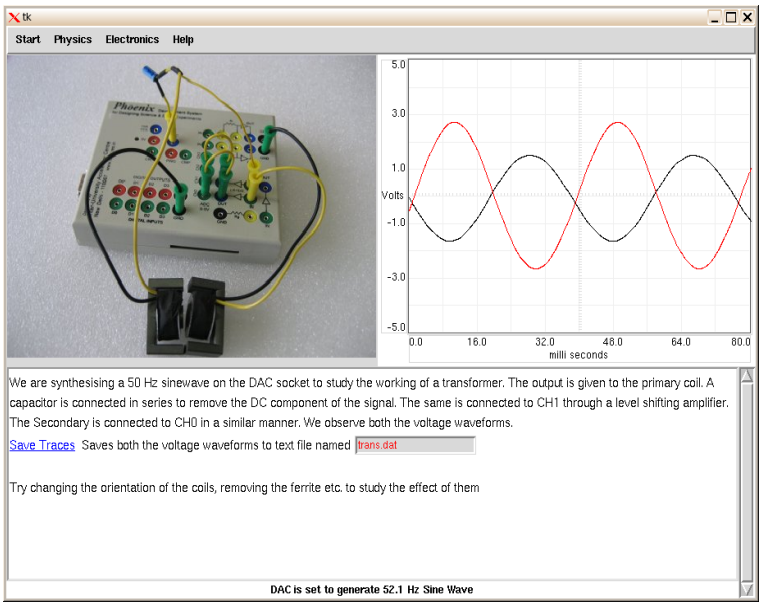


Figure 3.11: Mutual Induction between two coils

secondary waveforms.

3.6 Generate Sound from Electrical Signals

In the previous experiments, we have seen that a magnet moving inside a coil induces a voltage across the coil. Moving the coil inside a magnetic field also does the same since only the relative motion matters. In a similar manner, *if you allow a current to flow through a conductor kept inside a magnetic field the conductor will experience a force*. This phenomenon is the basis of devices like electric motor, loudspeaker etc. We will take the case of the loudspeaker and explore the conversion of electrical energy into sound energy.

A loudspeaker has a movable coil placed inside a magnetic field and a paper cone connected to the coil. When a current is made to flow through the coil, it moves. By applying an AC voltage across the coil we can make the paper cone move back and forth to generate sound waves. We can change the frequency of the waveform to change the frequency of the sound generated.

To study this experimentally using Phoenix, connect a loudspeaker between PGW and Ground sockets along with a 100Ω resistor to prevent overloading of the PWG output. Now click on the PWG socket and change the frequency, using the slider that pops up. You will observe that at some particular frequency, the intensity of sound shows a maximum. This is due to resonance.

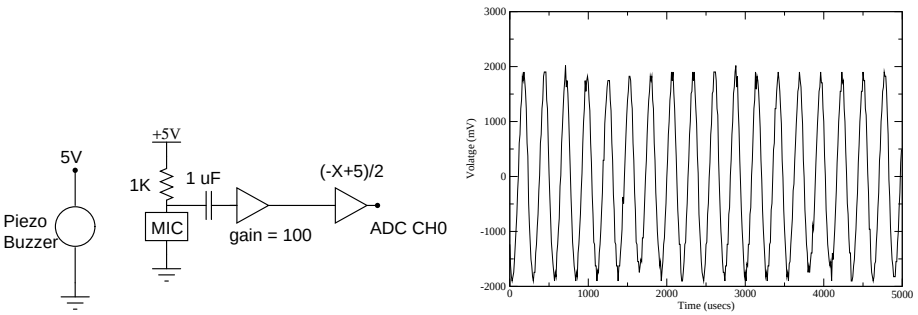


Figure 3.12: (a) Circuit to digitize the buzzer output sound using the microphone (b) The captured waveform.

3.6.1 Creating music

The musical notes are produced by certain frequencies, for example a 261.63 Hz frequency represents the note 'Sa'. One can produce some music by programming PWG to generate the notes in some order. From the help screen, clicking on the PWG Socket provides an option to play some tunes.

3.7 Digitizing audio signals using a condenser microphone

A condenser microphone is wired as shown in figure 3.12(a) to capture the audio signals. One end of the microphone goes to Vcc through a resistor, the other end is grounded. The output is taken via a capacitor to block the DC used for biasing the microphone. The signal is amplified by a variable gain inverting amplifier. The amplified output is level shifted and connected to ADC channel 0. The program 'analog box' from the main menu can be used to capture the waveform and a screen-shot is shown in figure 3.12(b).

The frequency can be roughly estimated by looking through the data file for time interval between two zero crossings. For more accurate results, use the curve fitting option of the program.

3.8 Synchronizing Digitization with External 'Events'

In the previous examples we have seen how to digitize a continuous waveform. We can start the digitization process at any time and get the results. This is not the case for transient signals. We have to synchronize the digitization process with the process that generates the signal. For example, the signal induced in a coil if you drop a magnet into it. Phoenix does this by making the 'read_block()' and 'multi_read_block()' calls to wait on a transition on the Digital Inputs or Analog Comparator Input.

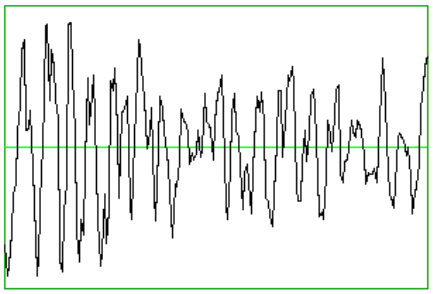


Figure 3.13: Collision sound.microphone

Connect the condenser microphone as shown in figure 3.12(a). Connect the output of the inverting amplifier to Digital Input D3 through a 1K resistor. The same is given to ADC through the level shifting amplifier.

Make some sound to the microphone. The 'p.enable_wait_high(3)' will make the read_block() function to wait until D3 goes HIGH. With no input signal the input to D0 will be near 0V, that is taken as LOW. The program 'wcro.py' used is listed below.

```
import phm
p = phm.phm()
p.select_adc(0)
p.enable_wait_high(3)
while 1:
   v = p.read_block(200,20,1)
   if v != None:
      p.plot_data(v)
```

Exercise: Use a similar setup to study the voltage induced on a coil when a magnet is suddenly dropped into it.

3.9 Temperature Measurements

In certain experiments it is necessary to measure temperature at regular time intervals. This can be done by connecting the output of a temperature sensor to one of the ADC inputs of Phoenix and record the value at regular intervals. There are several sensors available for measuring temperature, like thermocouples, platinum resistance elements and solid state devices like AD590 and LM35. They work on different principles and require different kind of signal processing circuits to convert their output into the 0 to 5V range required by the ADC. We will examine some of the sensors in the following sections.

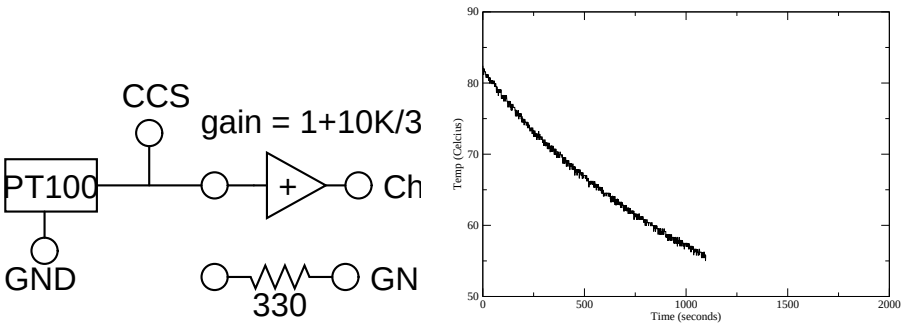


Figure 3.14: (a) PT100 Circuit. (b) The cooling curve

3.9.1 Temperature of cooling water using PT100

PT100 is an easily available Resistance Temperature Detector , RTD, that can be used from -200^{0}C to 800^{0}C. It has a resistance of 100 Ohms at zero degree Celsius; the temperature vs resistance charts are available. The circuit for connecting PT100 with Phoenix is shown in figure 3.14

The PT100 sensor is connected between the 1mA Constant Current Source and ground. The voltage across PT100 is given by Ohm's law, for example if the resistance is 100Ω the voltage will be 100 * 1 mA = 100 mV. This must be amplified before giving to the ADC. The gain is chosen in such a way that that amplifier output is close to 5V at the maximum temperature we are planning to measure. In the present experiment we just observe the cooling curve of hot water in a beaker. The maximum temperature is 100^{0}C and the resistance of PT100 is 138Ω at that point that gives 138 mV across it. We have chosen a gain of roughly 30 to amplify this voltage. The gain is provided by the non-inverting amplifier with a 330Ω resistor from the Yellow socket to ground.

How do we calculate the temperature from the measured voltage ? The resistance is easily obtained by dividing the measured voltage by the gain of the amplifier. To get the temperature from the resistance one need the calibration chart of P100 or use the equation to calculate it.

R_{T}= $\mathrm{R}_0\left[1 + AT + BT^2 - 100CT^3 + CT^4\right]$

- R_T = Resistance at temperature T
- $R_0$is the resistance at 0^0Celsius.
- $A = 3.908310^{-3}$
- $B = -5.77510^{-7}$

The first three terms are enough for temperatures above zero degree Celsius and the resulting quadratic equation can be solved for T. The program 'pt100.py' listed below prints the temperature at regular intervals. The output of the program is redirected to a file named 'cooling_pt100.dat' and plotted used xmgrace as shown in figure 3.14.

```
import phm, math, time
p = phm.phm()
gain = 30.7       # amplifier gain
offset = 0.0      # Amplifier offset, measured with input grounded
ccs_current = 1.0 # CCS output 1 mA
def r2t(r):       # Convert resistance to temperature for PT100
   r0 = 100.0
   A = 3.9083e-3
   B = -5.7750e-7
   c = 1 - r/r0
   b4ac = math.sqrt( A*A - 4 * B * c)
   t = (-A + b4ac) / (2.0 * B)
   return t
def v2r(v):
  v = (v + offset)/gain
  return v / ccs_current
p.select_adc(0)
p.set_adc_size(2)
p.set_adc_delay(200)
strt = p.zero_to_5000()[0]
for x in range(1000):
   res = p.zero_to_5000()
   r = v2r(res[1])
   temp = r2t(r)
   print '%5.2f %5.2f' %(res[0]-strt, temp)
   time.sleep(1.0)
```

Even though the experiment looks simple there are several errors that need to be eliminated. The CCS is marked as 1 mA but the resistors in the circuit implemented that can have upto 1% error. To find out the actual current do the following. Take a 100 Ohm resistor and measure its resistance 'R' with a good multimeter. Connect it from CCS to ground and measure the voltage 'V' across it. Now V/R gives you the actual current output from CCS.

For measurements around room temperature the voltage output is under a couple of hundred millivolts. For better precision this need to be amplified to 5V, to utilize the full range of the ADC. A gain of 20 to 30, depends on the upper limit of measurement, can be implemented using the variable gain amplifiers. The offset voltage of the amplifier should be measured by grounding the input and subtracted from the actual readings. The actual gain should also should be calculated by measuring the input and output at a a couple of voltages.

Another method of calibrating the setup is to measure the ADC output at 0^0and100^0 and assume a linear relation, which may not be very accurate, between the ADC output and the temperature.

Distance (cm)	Timeusec)	Dist. difference	Time diff.	Vel. (m/s
4	224			
5	253	1	29	344.8
6	282	2	58	344.8
7	310	3	86	348.8

Table 3.1: Velocity of sound

3.10 Measuring Velocity of sound

The simplest way to measure the velocity of anything is to divide the distance travelled by time taken. Since phoenix can measure time intervals with microsecond accuracy we can apply the same method to measure the velocity of sound. We will first try to do this with a pair of piezo electric crystals and later by using a microphone.

3.10.1 Piezo transceiver

A piezo electric crystal has mechanical and electrical axes. It deforms along the mechanical axis if a voltage is applied along the electrical axis. If a force is applied along the mechanical axis a voltage is generated along the electrical axis. We are using a commercially available piezo transmitter and receiver pair that has a resonant frequency of 40 KHz. The experimental setup is shown in figure 3.15.

The transmitter piezo is excited by sending a 13 micro seconds wide pulse on Digital Output Socket D3 to generate a sound wave. The sound wave reaches the receiver piezo kept several centimeters away and induces a small voltage across it. This signal is amplified by two variable gain amplifiers in series, each with a gain of 100. The output is fed to Digital Input D3 through a 1K resistor[3]. The interval between the output pulse and the rising edge of D3 is measured by the following program 'piezo.py'. The output is redirected to a file

```
import phm
p=phm.phm()
p.set_pulse_width(13)
p.set_pulse_polarity(0)
p.write_outputs(0)
for x in range(10):
   print p.pulse2rtime(3,3)
```

To avoid gross errors in this experiment one should be aware of the following. Applying one pulse to the transmitter piezo is like banging a metal plate to make sound, it generates a train of waves whose frequency is around 40 KHz. The

[3]It is very important to use this resistor. The amplifier output is bipolar and goes negative values. Feeding negative voltage to D3 may damage the micro-controller. The 1KOhm resistor acts as a current limiter for the diode that protects the micro-controller from negative inputs.

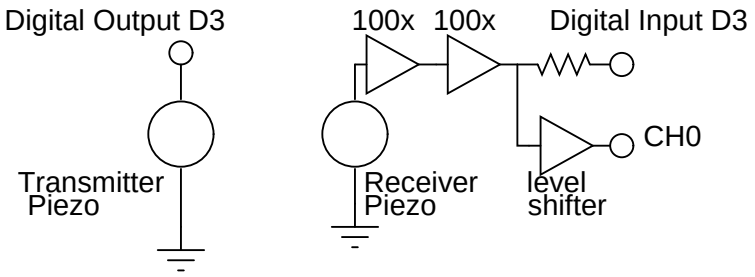


Figure 3.15: Piezo Transceiver setup measuring velocity of sound

receiver output is a wave envelope whose amplitude rises quickly and then goes down rather slowly. When we amplify this signal one of the crests during the building up of the envelope makes the Digital Input HIGH. When we increase the distance between the crystals the amplitude of the signal also goes down. At some point this will result in sudden jump of 25 microseconds in the time measurement which is caused by D3 going HIGH by the next pulse. This can be avoided by taking groups of reading at different distances varying it by 3 to 4 centi meters.

3.10.2 Condenser microphone

Velocity of sound can be measured by banging two metal plates together and recording the time of arrival of sound at a microphone kept at a distance. One metallic plate is connected to ground, another one is connected to a digital input say D0. The generated sound travels through air and reaches the microphone and induces an electrical signal. The electrical signal is amplified 200 times by two amplifiers in series and connected to D3. The experimental setup is shown figure 3.16. We have used 1 mm thick aluminium plates to generate the sound. When we strike one by the other, the digital input D0 gets grounded resulting in a falling edge at D0. The amplified sound signal causes a rising edge on D1 [4]. The software measures the time interval between two falling edges using the following lines of code. To get better results repeat the measurement several times and take average.

```
import pm
p = phm.phm()
print p.f2ftime(0,1)
```

Here is a table of measurements obtained experimentally:

[4] Rising or falling edge depends on the amplifier offset etc. If the amplifier output will start oscillating when the sound signal arrives. If is already HIGH it will go LOW when the sound signal arrives and we should look for a falling edge.

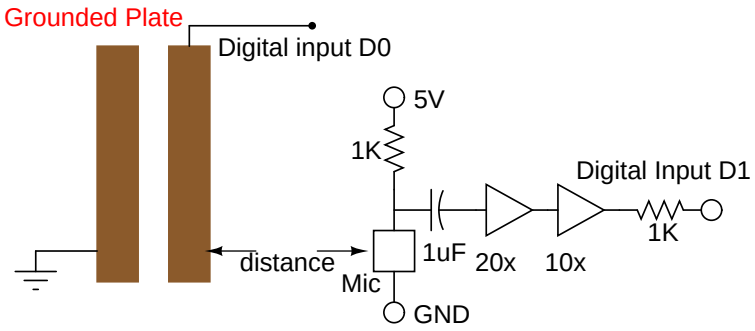


Figure 3.16: Velocity of sound by microphone

Distance (cm)	Time (milli seconds)	Speed = distance/time
0	0.060	To be treated as offset
10	0.350	344.8
20	0.645	341.8
30	0.925	346.8
40	1.218	345.4
50	1517	343.1
60	1810	342.8

3.11 Study of Pendulum

Studying the oscillations of a pendulum is part of any elementary physics course. Since the time period of a pendulum is a function of acceleration due to gravity, one can calculate g by doing a pendulum experiment. The accuracy of the result depends mainly on how accurate we can measure the period T of the pendulum. Let us explore the pendulum using phoenix.

3.11.1 A Rod Pendulum - measuring acceleration due to gravity

A rod pendulum is very easy to fabricate. We took a cylindrical rod and fixed a knife edge at one end of it to make a T shaped structure. The pendulum is suspended on the knife edges and its lower end intercepts a light barrier while oscillating. The light barrier is made of an LED and photo transistor. The output of the light barrier is connected to Digital Input D3. The program 'rodpend.py' is used for measuring T and calculating the value of g. The code is listed below.

```
import phm, math
p=phm.phm()
length = 14.65                  # length of the rod pendulum
```

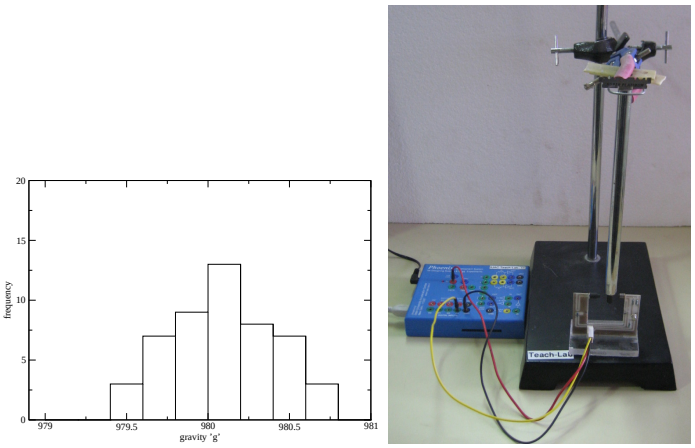


Figure 3.17: (a) Histogram of measured values of g. (b) The experimental setup.

```
pisqr = math.pi * math.pi
for i in range(50):
   T = p.pendulum_period(3)/1000000.0
   g = 4.0 * pisqr * 2.0 * length / (3.0 * T * T)
   print i, ' ',T, ' ', g
```

The output of the program is redirected to a file and a histogram is made using 'xmgrace' program as shown in figure 3.17. The mean value and percentage error in the measurement can be obtained from the width of the histogram peak.

3.11.2 Nature of oscillations of the pendulum

A simple pendulum can be studied in several different ways depending on the sensor you have got. If you have an angle encoder the angular displacement of the pendulum can be measured as a function of time. What we used is a DC motor with the pendulum attached to its shaft. When the pendulum oscillates it rotates the axis of the motor and a small time varying voltage is induced across the terminal of the motor. This voltage is amplified and plotted as a function of time. The experimental setup and output are shown in figure 3.18 on the following page. The program pend_digitize.py is listed below.

The output of the program is send to a file and plotted using *xmgrace*. The period of oscillation can be extracted by fitting the data with the equation of an exponentially decaying sinusoidal wave. The equation used for fitting the data is the following.

$A(t) = A_0 \sin(\omega t + \theta)e^{-\mathrm{dt}} + C$

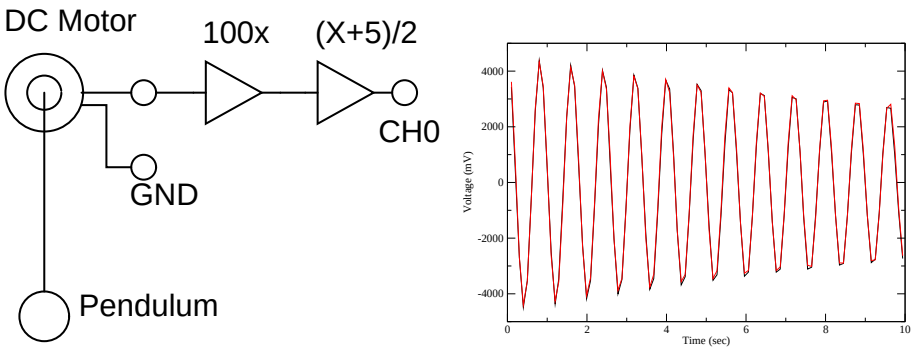


Figure 3.18: (a) Experimental setup. (b) Angular velocity of the pendulum as a function of time.

- $A(t)$ - Displacement at time t
- A_0 - Maximum displacement
- ω - Angular velocity
- θ - displacement at t=0
- d - Damping factor
- C - Constant to take care of DC offset from amplifiers

The angular velocity $\omega$is found to be 7.87 and the length of the pendulum is 15.7 cm. The calculated value of 'g' using the simple pendulum equation $\omega^2 L = 972$ cm/sec^2. The errors are due to the simple pendulum approaximation and the error in measurement of length.

3.12 Acceleration due to gravity by time of flight method

The motion of an object falling under gravity is governed by the relation $s = ut + \frac{1}{2}gt^2$ where s is the height from which the object falls, u the initial velocity of the object, g the acceleration due to gravity, and t the time taken. So, if we can accurately measure the time taken by an object with a known initial velocity, to fall through a known height, the acceleration due to gravity can be directly found from this relation. If the object is initially at rest, i.e. $u = 0$, the relation reduces to $s = \frac{1}{2}gt^2$. This experiment is straight forward if we can measure the time of flight with the desired accuracy.

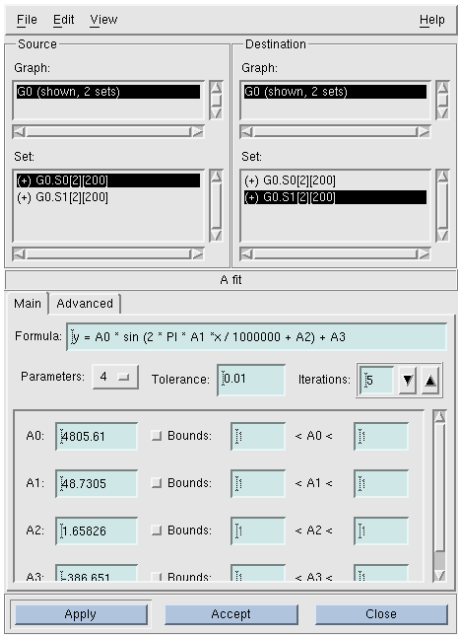
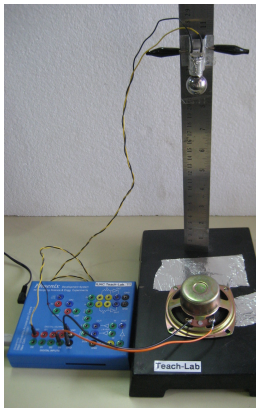


Figure 3.19: Pendulum Data fitted with equation using xmgrace program.



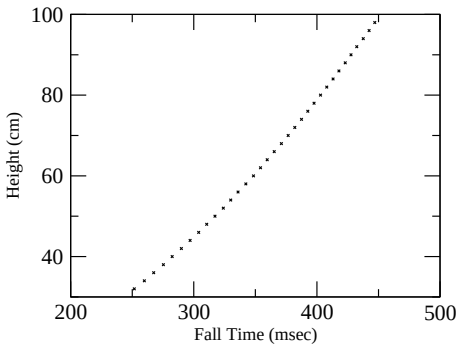


Figure 3.20: (a) Apparatus used for measuring the time of fall of a mild steel ball. (b) Plot of Fall Time Vs height.

3.12.1 The experimental Setup

The experimental setup used is shown in figure 3.20(a). We drop a spherical object of high density material from a known height and measure the time taken by it to fall thru a certain distance. High density is required to minimize the effect of air resistance and the spherical shape to avoid any dependence on the change in orientation while falling. The object has to be released under computer control and the arrival time of it on the ground also needs to be marked. We have used a mild steel ball of about 2 cm diameter magnetically held by a solenoid, taken from a commercially available relay. The solenoid is attached to a clamp, which can slide along a metallic meter scale mounted vertically. The accuracy of height measurement is around one fifth of a millimeter. For better accuracy one can use a height gauge with vernier attachment, giving an accuracy of 0.02 mm.

The solenoid is powered from Digital Output $D0$, that can be controlled through software. When the ball is released from the solenoid it falls on the metal plate at the bottom. The arrival time can be marked by using a touch sensor or a light barrier. We have tried those methods but found it is very easy to use a loudspeaker for this purpose. The loudspeaker is attached to the base to capture the electrical signal generated due to the vibrations created by the impact. The loudspeaker output is amplified and connected to one of the Digital Input $D0$ of Phoenix. The procedure is repeated for different heights and the time intervals are measured. The Python program used for carrying out the experiment is listed below. The last line of the code measures the time interval from clearing Digital Output (D0) to attaining a logic HIGH level on Digital Input (D0).

```
import phm                      # Load the library
p = phm.phm()                   # Create Phoenix object
p.write_outputs(1)              # Power the electro-magnet
print 'Attach the Iron Ball...'
time.sleep(3)
print p.clr2rtime(0,0)          # Measure time
```

3.12.2 Observations and Analysis

The fall time is measured for different heights. The height is measured from the top of the surface on which the ball falls to the bottom of the ball, when it is held by the solenoid. The results are shown in table 3.2. The value of 'g' calculated using the expression $s = \frac{1}{2}gt^2$ results in lower values of g . It happens due to the fact that the measured time of flight is more than the actual time of flight since the solenoid does not release the ball immediately after switching off the current. The correction is more pronounced for lower time of flight because the percentage error increases with the reduction in the total value. The calculated g shows better agreement with the expected value if a correction of 4 milliseconds is applied. However it is difficult to justify such an arbitrary correction. In fact we measured the delay in releasing the ball and found it to

Height $h(cm)$	Time $t(s)$	$g = \frac{2h}{t^2}(cm/s^2)$	$g' = \frac{2h}{(t-0.004)^2}(cm/s^2)^*$
93.13	0.4406	959.5	977.1
88.13	0.4285	960.0	978.1
83.13	0.4162	959.8	978.5
78.13	0.4037	958.8	978.1
73.13	0.3907	958.2	978.1
68.13	0.3773	957.2	977.8
58.13	0.3489	955.1	977.3
53.13	0.3337	954.2	977.5
48.13	0.3180	951.9	976.3
43.13	0.3013	950.2	975.9
38.13	0.2831	951.5	979.0
33.13	0.2643	948.5	977.9
28.13	0.2438	946.5	978.4
23.13	0.2218	940.3	975.2
18.13	0.1965	939.1	978.5
13.13	0.1680	930.4	976.4
8.13	0.1331	917.8	975.6

Table 3.2: Measured values of the Time of flight for different heights. The last column shows the value of g calculated after subtracting 4 milliseconds from the the time of flight.

be around 2 ms. The remaining 2 ms seems to be coming from the delay in sensing the time of arrival of the ball.

The next section describes a better method of analyzing the data, that eliminates the systematic errors in a nice way.

3.12.3 Data analysis by fitting the data

In this method, we really don't use the known relationship $s = \frac{1}{2}gt^2$. We just express the distance as an arbitrary polynomial function of time and extract the coefficients by fitting it with the experimental data.

$$s = a_0 + a_1 t + a_2 t^2 \tag{3.1}$$

From the definition of acceleration, we know that it is nothing but the second derivative of distance with respect to time. Differentiating equation 3.1 twice with respect to t, gives

$$acceleration = \frac{d^2 s}{dt^2} = 2a_2$$

The experimental data is shown in table 3.2 . The curve fitting is done using the program *xmgrace* , no correction is applied to the measured values. The

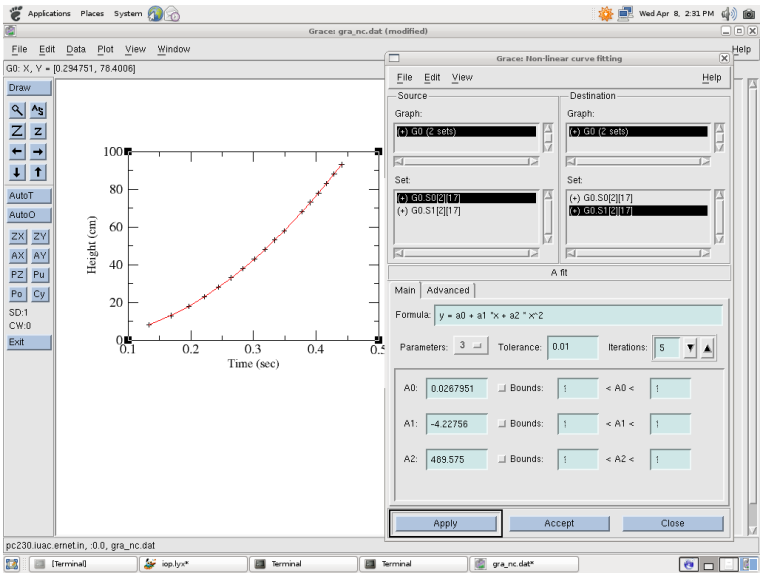


Figure 3.21: Fall Time Vs Height data is plotted and fitted with the equation $a_0 + a_1 t + a_2 t^2$ using the program xmgrace

value of the coefficient a_2 is 489.57, as shown in figure 3.21, which gives the value of acceleration to be 979.14 cm/s^2, which is very close to the accepted standard of 981 cm/s^2. The beauty of the method of curve fitting as a tool for analysis, lies in the fact that the systematic errors are absorbed by the coefficients a_0 and a_1.

3.13 Study of Timer and Delay circuits using 555 IC

Constructing astable and monostable multi-vibrators using $IC555$ is done in elementary electronics practicals. Using phoenix one can measure the frequency and duty-cycle of the output with micro second accuracy. In the case of monostable Phoenix can apply the trigger pulse and measure the width of the output.

3.13.1 Timer using 555

An astable multi-vibrator is wired using IC 555 as shown in figure 3.22(a). The output of the circuit is fed to CNTR input for frequency measurement and then to Digital Input D0 for duty cycle measurement. The code used is shown below along with the results obtained at each step.

```
print p.measure_frequency()      # signal connected to CNTR
```

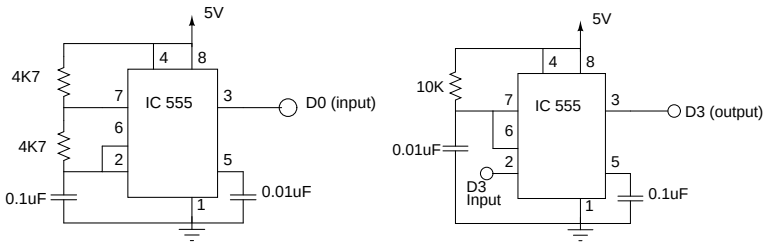


Figure 3.22: (a) IC555 oscillator circuit. (b) IC555 Monoshot circuit.

```
904
print p.r2ftime(0,0)            # signal to D0 input
733
print p.f2rtime(0,0)
371
```

Exercise: Cross check the above results with that predicted by the equation for frequency and duty cycle. The resistor values used are of 1% tolerance and capacitor of 5% tolerance.

3.13.2 Mono-stable multi-vibrator

The monostable circuit is wired up as shown in figure 3.22(b). The 555 IC require a LOW TRUE signal at pin 2 to trigger it. The output goes HIGH for a duration decided by the R and C values and comes back to LOW. The following lines of code is used for triggering 555 and measuring the time interval from trigger to the falling edge of the signal from pin 3.

```
p.set_pulse_width(1)
p.set_pulse_polarity(1)
p.write_outputs(8)          # keep D3 high
p.pulse2ftime(3,3)          # pulse on D3 to a falling edge on D3
123
```

Again it is left as an exercise to the reader to verify whether 123 microseconds is acceptable based on the RC values used.

3.14 Counting Gamma Rays

Gamma rays can be detected using a Geiger-Muller Counter. The GM tube is a coaxial tube filled with low pressure gas, with a high voltage applied to a thin wire along the axis. The passage of radiation causes a momentary discharge that is counted electronically. The Phoenix GM counter accessory generates the necessary high voltage and counts the pulses produced by radiation. The

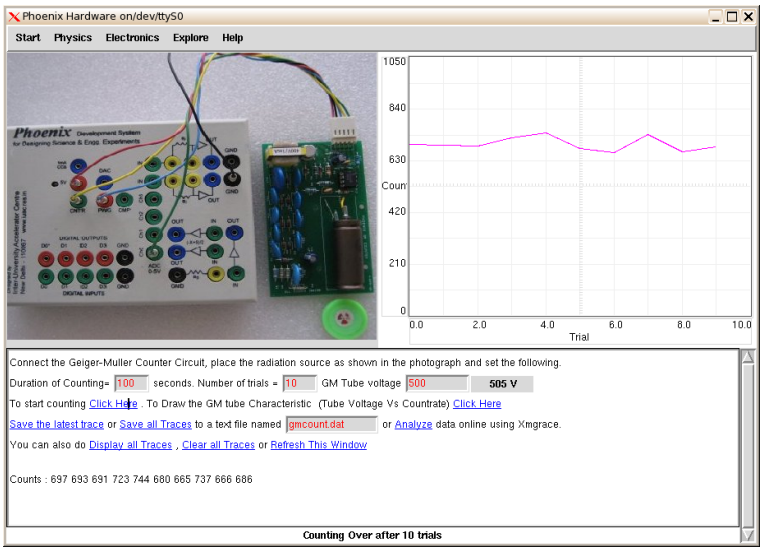


Figure 3.23: Screeshot of the Geiger-Muller Counter Experiment.

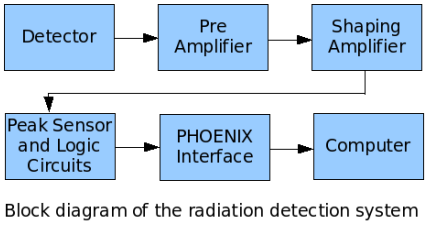


Figure 3.24: Radiation Detection Circuit block diagram.

experiment can be done using the GUI shown in figure 3.23 or using the simple Python program listed below.

```
#Count the GM tune output for one second , ten times.
import phm, time
p=phm.phm()
TIME = 1
print 'Tube voltage = ',p.gm_set_voltage(500)
for x in range(10):
     print p.gm_get_count(TIME)
```

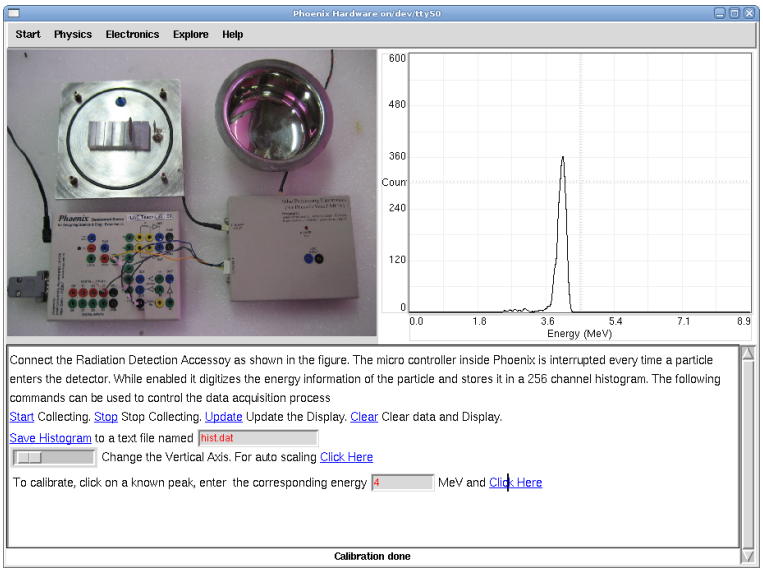


Figure 3.25: Screenshot of the radiation detection program. A photograph of the experimental setup is shown on the screen to guide the user.

3.15 Energy Spectrum of Alpha Particles

The radiation detection system has been designed as an accessory of the Phoenix Interface. The focus is on measuring the energy spectrum of the observed radiation. An inexpensive PN junction is used for alpha particle detection. Gamma radiation can be studied using scintillation or HPGe detectors. The required signal processing electronics (Pre-Amplifier, Shaping amplifier, discriminator etc. as shown in figure 3.24) has been packaged in to a small box. The digitization and the PC interfacing is provided by PHOENIX. A screenshot of the GUI program available for this experiment is shown in figure 3.25.

3.16 Amplitude Modulation

Use the program 'Analog Box' from the menu. (to be written)

3.17 Frequency Modulation

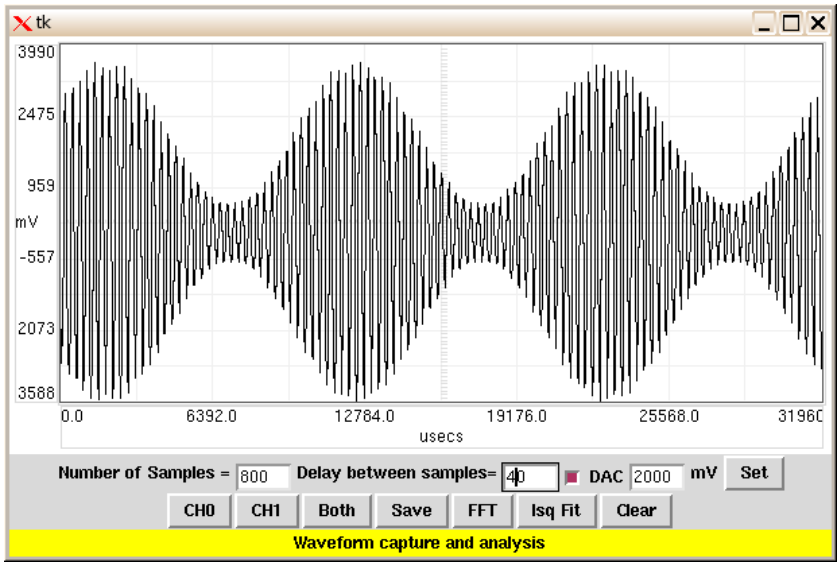


Figure 3.26: Amplitude modulated waveform. Generated using the Analog Box accessory asd captured by Phoenix Analog input.

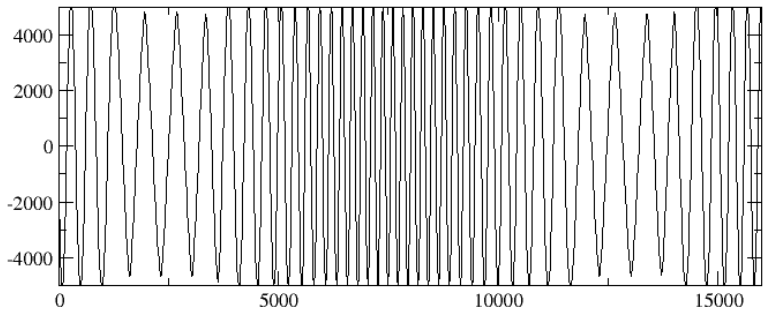


Figure 3.27: Frequency Modulation

Chapter 4

Python Library of Phoenix

This chapter explains the Python Libray functions for accessing the Phoenix hardware. They are grouped into sections like Digital I/O, Analog I/O, Time interval measurement functions etc. The functions belong to a class named 'phm' and will be invoked along with the object name. Any phoenix program will have the two lines of code shown below, to import the library and create an object using one class defined in the library.

```
import phm      # import the phoenix library
p = phm.phm()   # object of 'phm' class of 'phm' library
```

Phoenix library has online help. From the Python interpreter give the command 'help(phm)' after importing the 'phm' library.

Documentation Conventions

We define a function by its *name, type of data returned by it, its arguments and the data type of all the arguments*. Python functions can have a variable argument list. For example a function with two arguments can be called with a single argument if the second argument has a default value. They can also have named arguments. These features are illustrated with the an example function that accepts coordinate data in a list variable, and plots it.

None = plot(list data, int width=400, int height=300, Widget parent = None)

- This function returns the Python data type 'None'.
- The first argument is a list containing the coordinates to be plotted, which MUST be provided.
- The remaining arguments are having default values. If the calling program does not specify them, the default value is used.
- Second and third specify the dimensions of the plot window to be created.

- The last argument is the name of the parent window, inside which the new plot window will be created.

We can invoke this function in many different ways like:

- plot(data)
- plot(data,500,300) # width and height are specified in that order
- plot(data, height = 500) # height only is specified, skipping width

The following sections will use a format like

return_type = function_name (type arg1, ..., type argN = value, ...)

If no arguments are required, we will show and empty paranthesis. *The main data types in used are 'int', 'float', and 'list'. None* is the Python data type used if the function returns nothing. The convention will be clear from the simple examples in the beginning. There are only few functions that accepts variable number of arguments or named arguments. Most of them have one or two arguments only.

4.1 Digital Inputs

There are four digital input sockets. You can connect them externally to Ground or 5 volts [1], to make the voltage level HIGH or LOW. The software can read the voltage level present on all sockets. It can also monitor the level transitions on these sockets with microsecond timing resolution, a feature that will be explained later.

4.1.1 read_inputs

PROTOTYPE

```
int read_inputs()
```

DESCRIPTION

The function reurns an integer whose 4 LSBs represents the voltage level present at the Digital Input Sockets.

USAGE

```
data = p.read_inputs()
```

EXAMPLES

```
print p.read_inputs()
```

will print the number 15 (1111_{bin}) if nothing is connected to the sockets, they are all internally pulled up to a HIGH. Invoking the same function with D0 grounded will return 14.

[1] They are TTL inputs. Any voltage less than 0.8V is taken as a LOW and greater than 2V is taken as a HIGH.

4.2 Analog Comparator Input

The socket marked as CMP behaves in a manner similar to that of Digital Inputs. In most of the cases it can be considered as the fifth digital input. Advanced features of CMP will be discussed later.

4.2.1 read_acomp

PROTOTYPE

```
int read_acomp()
```

DESCRIPTION

The function reurns one if the CMP input is less that 1.23 volts, otherwise it returns zero.

USAGE

```
level = p.read_acomp()
```

EXAMPLES

```
print p.read_acomp()
```

4.3 Digital Outputs

There are four digital output sockets. You can set the voltage level on them to zero or five volts using software. The first Digital Output, D0*, is transistor buffered and capable of driving up to 100 mA current, it should not be used for timing applications. All other outputs can provide only up to 5 mA. If you connect LEDs to them, use a 1KΩresistor in series with the LED for current limiting.

4.3.1 write_outputs

PROTOTYPE

```
None write_outputs(int)
```

DESCRIPTION

The function takes an integer as argument whose 4 LSBs are used for setting the voltage level on the four Digital Output sockets.

USAGE

```
p.write_outputs(15)
```

EXAMPLES

```
p.write_outputs(15)
p.write_outputs(8)
```

The first line will make all 4 digital outputs HIGH (15 is 1111_{binary}), the second one will make D3 HIGH and all other LOW. Measure the outputs with a voltmeter or by connecting an LED from the sockets to ground with a 1KΩresistor in series.

4.4 Analog Output

Phoenix has one Programmable Voltage Source, marked as DAC. The voltage level on the DAC socket can be set from 0 to 5V. The resolution is only 8 bits, the voltage will change in about 19 mV steps. The DAC is implemented by controlling the duty cycle of a 31.25 KHz Pulse Width Modulated waveform generated on PWG socket. The PWG is internally connected to the DAC socket through a low pass filter. Due to this reason, PWG and DAC cannot be used at the same time. The quality of the DC output on DAC output can be improved by adding external filters.

4.4.1 set_voltage

PROTOTYPE

```
None set_voltage(float mV)
```

DESCRIPTION

Set the output voltage of the DAC. The value of mV should be from 0 to 5000. It represents voltage in milli volts.

USAGE

```
p.set_voltage(2000)       # Sets 2 volts on DAC socket
```

4.4.2 set_dac

PROTOTYPE

None set_ dac(int k)

DESCRIPTION

Write a one byte value to the 8 bit DAC. The DAC output varies from 0 to 5000 mV. Writing a 0 to the DAC results in an output voltage of 0 and writing a 255 results in an output voltage of 5V. Intermediate values give appropriately scaled outputs. Almost always, you will not have to use this function in your code - the *set_ voltage* function is much more convenient.

USAGE

```
p.set_dac(127)     # set DAC to nearly 2.5 volts
```

4.5 Analog Inputs

Phoenix has four channels of analog inputs, CH0, CH1, CH2 and CH3. The input voltage MUST be within the 0V to 5V range. The input voltage can be digitized with 10 bit resolution, requiring 2 bytes to store the data. It is also possible to ignore the two LSBs and return a 1 byte data.

Phoenix supports two modes of digitizing the analog voltage present at the input sockets, *single conversion and block mode conversion*. In the Single conversion mode, the voltage is measured only once and the result is returned. In the block mode, more than one measurements are done during a single function call. The calling program can specify the number of measurements to be done and the time interval between two consecutive measurements. Block reads are necessary for digitizing waveforms.

4.5.1 ADC Settings

4.5.1.1 select_adc

PROTOTYPE

```
None select_adc(int chan)
```

DESCRIPTION

Selects one from the four channels available. The voltage at this analog input will be digitized during the subsequent calls to measure the voltage. The channel number ranges from 0 to 3.

USAGE

```
select_adc(0)    # selects the first channel
```

4.5.1.2 set_adc_size

PROTOTYPE

```
None set_adc_size(int size)
```

DESCRIPTION

The Phoenix ADC resolution can be set to 8 or 10 bits. Calling this function with argument 1 will choose 8 bits, an argument of 2 chooses 10 bits. It is set to 8 bits on powering. Programs using ADC must call this function once after a power up. *If a program sets the data size to 2 bytes and Phoenix is reset after that, a mismatch will occur resulting in communication error since the default is 1 byte at the micro-controller side.*

USAGE

```
p.set_adc_size(1) #set the resolution to 8 bits
p.set_adc_size(2) #set it to 10 bits
```

4.5.2 Single Reads

4.5.2.1 get_voltage

PROTOTYPE

```
tuple get_voltage()
```

DESCRIPTION

Reads the analog voltage on the current ADC channel and returns a tuple. First element of the tuple is the PC time-stamp and the second element is the voltage in milli-volts. The timestamp is required for plotting slowly varying parameters as a function of time.

This function was earlier called zero_to_5000().

USAGE

```
res = p.get_voltage()
```

EXAMPLES

Connect DAC to CH0 using a piece of wire and run the following program several times. The result will be fluctuating by around 20 mV. Connect a 1KΩ resistor from DAC to CH0 and a 1 uF capacitor from CH0 to GND. Run the program again several times to observe the difference.[2]

```
p.set_voltage(3000)
p.select_adc(0)
p.set_adc_size(2)
print p.get_voltage()[1] # print voltage only
```

4.5.2.2 get_voltage_bip

PROTOTYPE

```
tuple get_voltage_bip()
```

DESCRIPTION

The Phoenix ADC input accepts only voltages within the 0 to 5V range. In many experiments the sensors may generate voltages going to negative values also. The level shifting amplifiers, (-x+5)/2 , convert a -5V to +5V range signal into a 0 to 5V signal. Calling get_voltage_bip() will return the voltage given to the input of the level shifter, so that the user program need not calculate it. This function was earlier called minus5000_to_5000().

USAGE

```
res = p.get_voltage_bip()  # read a bipolar signal
```

[2] The ripple on the DAC output is reduced by the extra RC filter, resulting in a better DC output.

EXAMPLES

Connect the input of a level shifter to GND, output to CH0 and run the following program. The third line will print around 2500 mV and the fourth one around zero millivolts. You may find the values differing by around 10 mV, that is the level of accuracy you can get from the ADC used. This can be corrected to some extend by calibrating the ADCs using known voltages.

```
p.set_adc_size(2)
p.select_adc(0)
print p.get_voltage()[1]
print p.get_voltage_bip()[1]
```

4.5.2.3 read_adc

PROTOTYPE

```
tuple read_adc()
```

DESCRIPTION

Digitizes the analog voltage on the current ADC channel (set by the 'select_adc' call) and returns a number in the range 0-255 or 0-1023 (depending on the ADC sample size set by the 'set_adc_size' function) and the system time stamp as a tuple. It is easier to use get_voltage()

USAGE

```
print p.read_adc()
```

4.5.2.4 adc_input_period

PROTOTYPE

int adc_input_period(int chan)

DESCRIPTION

The period of a voltage waveform, in microseconds, on any of the ADC inputs can be measured by this function. This works fine for square wave inputs with amplitude greater than 1.5 volts. The ADC input is internally connected to the Analog Comparator for doing this measurement.

USAGE

```
print p.adc_input_period(0)
```

4.5.3 Block Reads

To capture waveforms having frequency more than several Hertz, we need to digitize several hundred points with a single function call. The time interval between consecutive digitizations must be kept same. The Block Read functions are used for capturing waveforms. There are several other supporting functions in this group for setting various parameters related to block mode digitization.

4.5.3.1 read_block

PROTOTYPE

```
list read_block(int np, int delay, int bipolar = 0)
```

DESCRIPTION

The first argument is the number of voltage measurements to be done and the second is the time interval between them in microseconds. The third argument is used for converting the raw data to the voltage value. If you are feeding the signal through the level shifters, use **1** as the third argument. If you do not use the third argument, it is taken as zero by default.

The channel to digitized and the datasize are set by the select_adc() and set_adc_size() functions. The maximum number of samples is limited by the size of the buffer inside the microcontroller. The buffer size is 800 bytes[3], means the maximum is 800 with 8 bit size and 400 with 10 bit size.

The return value is a list containing tuples like *[(t1,v1), (t2,v2).....]* , where each tuple contains the time and voltage values of one sample. In case of any error a single element list containing the error message is returned. It is the calling programs responsibility to check this before trying to use the returned value.

USAGE

```
data = p.read_block(100, 20, 1)
data = p.read_block(100,20)
```

EXAMPLES

Connect PWG to CH0 and run the following program.

```
p.select_adc(0)
p.set_adc_size(2)
p.set_frequency(1000)
v = p.read_block(400, 20)
if len(v) == 1:
     print v         #error message
p.plot(v)
```

4.5.3.2 multi_read_block

PROTOTYPE

```
list multi_read_block(int np, int delay, int bipolar = 0)
```

[3] The buffersize for the ATmega32 version of Phoenix is 1800 bytes

DESCRIPTION

This call is similar to read_block() but capable of digitizing data from multiple channels. The channels to be digitized are set by *add_channel() and del_channel()* functions, explained below. *The channel selected by select_adc() function has no effect on multi_read_block().* The arguments have the same meaning as in read_block(). The size of each item in the returned list is decided by the number of active channels. Some example results are shown below.

All channels added : [[t1, a1, b1, c1, d1], [t2, a2, b2, c2, d2], ...]

Channels 0 and 1 : [[t1, a1, b1], [t2, a2, b2], ...]

Channels 2 and 3 : [[t1, a1, b1], [t2, a2, b2], ...]

It should be noted that the returned list does not have information about the active channels. The calling program should get this from the value of channel mask. With all channels selected, the maximum number of samples is 200 with 1 byte resolution and 100 with 2 byte resolution, decided by the 800 byte buffersize. In case of any error a single element list containing the error message is returned. It is the calling programs responsibility to check this before trying to use the returned value.

USAGE

```
v = p.multi_read_block(100, 10)
```

EXAMPLES

```
p.add_channel(1)
p.set_adc_size(1)
v = p.multi_read_block(5, 10)
print v
```

If you run this code after powering the micro-controller, the result will have data from CH0 and CH1, since CH0 is selected by default. Each element in the list will have three values, time and two voltages. You can explicitly deselect CH0 by calling del_channel(0).

4.5.4 Block Read Channel Selection

The firmware inside Phoenix keeps a *4 bit channel mask,* whose bits can be set by add_channel() and cleared by del_channel(). The channel mask decides which all channels will be read during a multi_read_block() call. For example, if the channel mask is 5 (0101_{binary}), CH0 and CH2 will be read. The function get_chanmask() returns the current values of the channel mask. The number of elements in each item of the list returned by multi_read_block() depends on the number of currently active channels. If all the channels are set, each item will have five elements, where the first one is the time value and the remaining four are the voltage levels at the four channels.

4.5.4.1 add_channel , del_channel

PROTOTYPE

```
None add_channel(int chan)
None del_channel(int chan)
```

DESCRIPTION

add_channel() sets the specified bit in the Channel Mask and del_channel() clears it.

USAGE

```
p.add_channel(1)
p.del_channel(3)
```

EXAMPLES

```
for k in range(4):
    p.del_channel(k)  # deselect all channels
p.add_channel(3)      # Add the fourth channel
v = p.multi_read_block(100, 10, 0)
p.plot(v)
```

4.5.4.2 get_chanmask

PROTOTYPE

```
int get_chanmask()
```

DESCRIPTION

This function returns the value of the current channel mask. The four LSBs of the returned integer contains the selected channel information. This call is used by programs like CRO to interpret the data returned by multi_read_block() function.

USAGE

```
mask = p.get_chanmask()
```

EXAMPLES

```
print p.get_chanmask()
```

4.5.5 Block Read Modifiers

The behavior of block reads calls can be controlled in several ways to enhance their flexibility. They can be made to start only when the input is between some specified limits, this feature is essential for getting a stable trace for CRO applications.

You can also synchronize the beginning of digitization process with some external event. Digitization is made to wait for specified level changes on a digital input. This feature is useful for digitizing transient waveforms. The synchronizing signal is derived from the waveform itself and applied to a digital input.

It is also possible to SET, CLEAR or PULSE one of the digital outputs just before starting the digitization process. The control functions only changes the settings at the micro-controller end, the actions are visible only when a block read call is made.

4.5.5.1 set_adc_trig

PROTOTYPE

None set_adc_trig(float lower, float upper, int shifted = 0)

DESCRIPTION

A CRO application typically reads a number of samples from the ADC in bulk and plots it on to the screen; this process is repeated. If every time we start digitizing from a different part of the signal (say a periodic sine wave), the display will not remain static and will tend to 'run around'. The solution is to fix the starting point for each scan. The set_adc_trig() tells the read block calls to start the action only when the input voltage is changing from lower to upper. The trigger levels are specified in millivolts. The third argument is 1 when the level shifters are used. If you omit that argument, zero is taken by default. *This function assumes that the ADC is set to 8 bit resolution.*

USAGE

```
p.set_adc_trig(2000, 2200,0)
p.set_adc_trig(-200, 200, 1)
```

EXAMPLE

```
p.set_frequency(500)
p.select_adc(0)
p.set_adc_size(1)
# p.set_adc_trig(1000, 4000, 0)
p.set_adc_trig(0, 200)
while 1:
    x = p.read_block(400, 20, 0)
    p.plot_data(x)
```

Connect PWG to CH0 and run this code. Run it again after commenting line 3 instead of line 4 and observe the change in the stability of the trace.

4.5.5.2 enable_wait_high, enable_wait_low

PROTOTYPE

```
None enable_wait_high(int digin)
None enable_wait_low(int digin)
```

DESCRIPTION

A block_read or multi_read_block called after invoking this will wait for the specified digital input socket to go HIGH / LOW before starting digitization. If that does not happen, a timeout error will happen and the read_block() will return a None.

USAGE

```
p.enable_wait_high(0)
p.enable_wait_low(0)
```

EXAMPLE

```
p.enable_wait_low(0)
p.enable_wait_high(2)
x = p.read_block(200,20,0)
```

The first call to wait for a LOW on D0 is reset by the second call and the read_block waits only for the digital input D2 to go HIGH. An example to capture a transient waveform is explained below.

Connect a loudspeaker between the input of the non-inverting amplifier and GND. Set the gain resistor to 100 Ω. Connect the output of the amplifier to CH0 through the level shifter. Connect it to Digital Input D0 through a 1KΩ resistor. Run the following code and immediately tap the loudspeaker lightly.

```
p.enable_wait_high(0)
p.select_adc(0)
x = p.read_block(400,20,0)
```

The read_block() will wait until D0 goes high. The electrical signal from the loudspeaker will make this and the digitization will start from that point.

4.5.5.3 disable_wait

PROTOTYPE

None disable_wait()

DESCRIPTION

This function will cancel the effect of calling enable_wait_low or enable_wait_high.

USAGE

```
disable_wait()
```

4.5.5.4 enable_set_high, enable_set_low

PROTOTYPE

None enable_set_high(int digout)

None enable_set_low(int digout)

DESCRIPTION

In some applications, it would be necessary to make a digital output socket go high/low before digitization starts. It is ofcourse possible to do this by first calling write_outputs and then starting digitization - but the in-between delay may not be acceptable in some applications. This function, when called with a digital output socket number as argument, makes a subsequent ADC block digitization function set/clear the socket before it begins the digitization process. Action can be defined on only one digital output at a time.

USAGE

```
p.enable_set_high(0)
p.enable_set_low(1)
```

EXAMPLE

Capturing the voltage across a capacitor while charging / discharging is a typical application of this feature. Connect a 1uF capacitor between CH0 and GND. Connect a 1KΩ resistor from digital output D3 to CH0 and run the following code.

```
p.write_outputs(0)
time.sleep(1)
p.enable_set_high(3)
x = p.read_block(200, 20)
p.plot(x)
raw_input()    # wait until a key is pressed
```

Due to the third line, D3 is taken to HIGH just before digitizing the voltage on CH0. The voltage at CH0 will follow it to 5V exponentially.

4.5.5.5 enable_pulse_high, enable_pulse_low

PROTOTYPE

None enable_pulse_high(int digout)

None enable_pulse_low(int digout)

DESCRIPTION

In some applications, it would be useful to send a Pulse on a digital output before digitization starts. The enable_pulse_high() maks the speficied output HIGH for some duration and then makes it LOW. The duration is set by the set_pulse_width() function. The calling program should make sure that the socket is set to LOW before calling read_block, else a HIGH to LOW transition will result instead of a pulse. The enable_pulse_low() takes the output LOW and then HIGH after some duration.

Pulse action can be defined on only one digital output at a time.

USAGE

```
p.enable_pulse_high(0)
p.enable_pulse_low(1)
```

EXAMPLE

This function can be used to capture a waveform that is triggered by an input signal. For example, connect the digital output D3 to the input of a IC555 mono-shot circuit and connect the 555 output to CH0. Set the mono-shot delay to around a millisecond and run the following code.

```
p.write_outputs(8)
p.set_pulse_width(1)
p.set_pulse_polarity(1) #  LOW TRUE pulse
p.enable_pulse_low(3)
x = p.read_block(300, 10)
p.plot(x)
raw_input()    # wait until a key is pressed
```

4.5.5.6 disable_set

PROTOTYPE

None disable_set()

DESCRIPTION

This function cancels the effect of enable_set and enable_pulse calls mentioned above.

EXAMPLE

```
p.disable_set()
```

4.6 Waveform Generation and Frequency Counter

Phoenix can generate waveforms and measure the frequency of an input signal in several different ways as explained below.

4.6.1 Programmable Waveform Generator (PWG)

The socket marked as 'PWG' can be programmed to generate a square wave. The voltage level swings between zero and five volts. The frequency can be set to values from 15 Hz to 4 MHz. However, all intermediate values are not possible since the frequencies are generated by dividing the 8 MHz clock frequency and comparing with set point registers.

4.6.1.1 set_frequency

PROTOTYPE

```
float set_frequency(float freq)
```

DESCRIPTION

The function generates a square waveform, having 50% duty cycle, on the PWG socket of the Phoenix box whose frequency is 'n' Hz. The frequency can vary from 15Hz to 4MHz. We may not get the exact frequency which we have specified, only something close to it. The function returns the actual frequency set in Hz. Note that waveform generation is done purely in hardware - the Phoenix box can perform some other action while the waveform is being generated.

The DAC unit, explained later, should not be used while PWG is running - doing so will result in a 31.25 KHz waveform whose duty cycle proportional to the value set to the DAC.

USAGE

```
m = p.set_frequency(1000)
```

EXAMPLES

```
print p.set_frequency(1000)
print p.set_frequency(1005)
```

The first line will print '1000.0' but the second line will print '1008.06', that is the possible frequency just above the requested one.

Connect PWG to CH0 and run the following code to capture the waveform.

```
p.set_frequency(1000)
x = p.read_block(300,20)
p.plot(x)
raw_input()
```

4.6.2 Arbitrary Waveform Generation

As discussed earlier, the voltage on the DAC socket can be set to any value between 0 to 5V. Under software control, it is also possible to change this value in a periodic manner to generate a waveform. The DAC input values required to generate a single cycle of the waveform is loaded into the memory of the micro-controller and they are send to the DAC one by one. Note that the DAC is set by periodically triggered interrupt service routines consuming the processor time. Resuts of block reads issued during arbitrary waveform generation may not be very accurate.

4.6.2.1 load_wavetable

PROTOTYPE

int load_wavetable(list wavetable)

DESCRIPTION

Loads the wavetable, a list of 100 numbers, to the eeprom memory of the micro-controller. Returns the number of bytes loaded, less than 100 indicates an error.

EXAMPLE

```
p.load_wavetable(v)
```

EXAMPLE

```
v = []
for k in range(100):
    v.append(k)
p.load_wavetable(v)
```

4.6.2.2 start_wave

PROTOTYPE

float start_wave(float freq, int external_dac = 0)

DESCRIPTION

Starts the wave generation on the DAC socket or on the external plug-in DAC, using the loaded wavetable. This function uses the micro-controller interrupts and the maximum frequency is limited to around 150 Hz. It is more useful in the low frequency range where frequencies less than a Hertz need to be generated.

EXAMPLE

```
p.load_wave(10,0)
```

EXAMPLE

```
v = []
for k in range(100):
    v.append(k)
p.load_wavetable(v)
p.start_wave(20,0) # no external DAC used
```

4.6.2.3 pulse_d0d1

PROTOTYPE

None pulse_d0d1(float freq)

DESCRIPTION

Generates a squarewave on Digital outputs D0 and D1. Only low frequencies are possible.

EXAMPLE

```
p.pulse_d0d1(20)
```

4.6.2.4 stop_wave

PROTOTYPE

None stop_wave()

DESCRIPTION

Stops the wave generation on the DAC socket and digital outputs. Done by disabling the MCU interrupts.

EXAMPLE

```
p.stop_wave()
```

4.6.3 measure_frequency

PROTOTYPE

int measure_frequency()

DESCRIPTION

Measure the frequency of the square waveform at the CNTR input. Returns the frequency in Hz. The input frequency can be from several Hertz to one MHz. Input to CNTR is monitored for one second and the number of pulses are counted.

EXAMPLE

Connect PWG to CNTR and run the following code

```
p.set_frequency(500)
print p.measure_frequency()
```

4.7 Passive Time Interval Measurements

Digital Inputs and the CMP socket of Phoenix can be used for measuring time intervals with microsecond resolution. The time between two level transitions can be measured. The transitions defining the start and finish could be on the same socket or on different socket.

4.7.0.1 r2ftime, f2rtime

PROTOTYPE

```
int r2ftime(int digin1, int digin2)
int f2rtime(int digin1, int digin2)
```

DESCRIPTION

r2ftime returns delay in microseconds between a rising edge on digin1 and falling edge on digin2 - the sockets can be the same. socket numbers 0 to 3 indicate digital input sockets D0 to D3 and socket number 4 stands for the CMP input. f2rtime() measures time from a falling edge to a riding edge.

USAGE

```
p.r2ftime(0, 1)
```

EXAMPLES

Connect PWG to digital input D0 and run the following code, should print around 500 usecs.

```
p.set_frequency(1000) # half period = 500 usecs
print p.r2ftime(0,0)
```

4.7.0.2 r2rtime, f2ftime

PROTOTYPE

```
int r2rtime(int digin1, int digin2)
int f2ftime(int digin1, int digin2)
```

DESCRIPTION

r2rtime returns delay in microseconds between a rising edge on digin1 and rising edge on digin2 - the sockets MUST be distinct. If you want to measure the time between two rising edges, use multi_r2rtime(). f2ftime() measures the time between two falling edges.

USAGE

```
p.r2rtime(0, 1)
```

EXAMPLES

```
print p.r2rtime(0, 1)
```

4.7.0.3 multi_r2rtime

PROTOTYPE

```
int multi_r2rtime(int digin, int skipcycles)
```

DESCRIPTION

Measures time interval between two rising edges of a waveform applied to a digital input socket (D0 to D3) or CMP socket. If 'skipcycles' is zero, period of the waveform is returned. In general, 'skipcycles' number of consecutive rising and falling edges are skipped between the two rising edges.

USAGE

p.multi_r2rtime(0, 3)

EXAMPLE

Connect PWG to digital input D0 and run the following code.

```
p.set_frequency(1000)
a = p.multi_r2rtime(0,9) # time for 10 cycles in usecs
frequency = 10.0e6/t10 # in Hz
```

For a periodic waveform input, the first line returns the time for one cycle and the second one returns the time for 10 cycles (9 rising edges in between skipped). This call can be used for frequency measurement. The accuracy can be improved by measuring larges number of cycles.

4.7.0.4 pendulum_period

PROTOTYPE

```
int pendulum_period(int digin)
```

DESCRIPTION

This is equivalent to multi_r2r() with one cycle skipped. Some software delay is added to get rid of the noise present at the level transitions.

4.8 Active Time Interval Measurements

During some experiments, we need to initiate some action and measure the time interval to the result of it. The actions are initiated by setting, clearing or by sending pulses on the Digital Outputs. The results will generate voltage transitions on Digital Inputs or on CMP.

4.8.0.5 set2rtime, set2ftime, clr2rtime, clr2ftime

PROTOTYPE

int set2rtime(digout, digin)

(remaining functions have similar prototypes)

DESCRIPTION

These functions SET/CLEAR a digital output socket specified by 'digout' and wait for the digital input (or analog comparator) socket specified by 'digin' to go HIGH /LOW.

USAGE

p.set2rtime(0, 1)

4.8.0.6 pulse2rtime, pulse2ftime

PROTOTYPE

```
int pulse2rtime(int digout, digin)
int pulse2rtime(int digout, digin)
```

DESCRIPTION

Sends out a single pulse on a Digital Output and waits for a rising/falling edge on a Digital Input or CMP. The duration and the polarity of the pulse is set by set_pulse_width() and set_pulse_polarity() functions. On powerup the width is 13 microseconds and polarity is positive (voltage goes from 0V to 5V and comes back to 0V). The initial level of 'digout' should be set according to the polarity setting. If the polarity is LOW TRUE, the level must be set high beforehand and it should be set low for HIGH TRUE pulse.

USAGE

```
p.pulse2rtime(0, 0)
```

EXAMPLE

```
p.set_pulse_width(1)
p.set_pulse_polarity(1)
print p.pulse2rtime(0, 1)
```

4.8.0.7 set_pulse_width

PROTOTYPE

None set_pulse_width(int width)

DESCRIPTION

Sets the pulse width, in microseconds, to be used by the pulse2ftime(), pulse2rtime() and pulse_out() functions.

USAGE

```
p.set_pulse_width(10)
```

4.8.0.8 set_pulse_polarity

PROTOTYPE

None set_pulse_polarity(int pol)

DESCRIPTION

Sets the pulse polarity to be used by the pulse2ftime(), pulse2rtime() and pulse_out() functions. pol = 0 means a HIGH TRUE pulse and pol=1 means a LOW TRUE pulse.

USAGE

```
p.set_pulse_polarity(1)
```

4.9 Histogram Generation

A level change on the CMP socket of Phoenix can trigger an interrupt service routine. This feature is used by the accessory for Radiation Detection and Analysis. When a charged particle or gamma ray photon is incident on the radiation detector, it produces a voltage pulse whose amplitude is proportional to the energy of incident radiation. This pulse is amplified and given to the ADC CH0 input. A logical pulse is given to the CMP socket if the voltage level is above the noise threshold. Every pulse on CMP runs a program to digitize the voltage present at CH0 and make a 256 channel histogram using that data. Each channel is of 2 byte size and can hold upto 65535 counts. If any of the channels read this level, the histogramming is automatically stopped to avoid overflow. *Read block calls should not be issued while collecting histogram since the same buffer area is used for storing the data.*

The histogramming feature is controlled by the functions explained below.

4.9.1 start_hist

PROTOTYPE

```
None start_hist()
```

DESCRIPTION
Enable the CMP interrupts to start histogramming.

USAGE

```
p.start_hist()
```

4.9.2 stop_hist

PROTOTYPE

```
None stop_hist()
```

DESCRIPTION
Disable the CMP interrupts to stop histogramming.

```
USAGE
p.stop_hist()
```

4.9.3 clear_hist

PROTOTYPE

```
None clear_hist()
```

DESCRIPTION
Clear the 512 byte histogram buffer in the micro-controller

USAGE

```
p.clear_hist()
```

4.9.4 read_hist

PROTOTYPE

```
None read_hist()
```

DESCRIPTION
Returns the 256 element (2 byte) histogram data.

USAGE

```
v= p.read_hist()
```

EXAMPLE

To test this feature, connect PWG to CMP using a cable. Connect the level shifter output (will be at 2.5 V when the input is not connected) to CH0. Run the following program and plot it using 'xmgrace hist.dat' .

```
import phm, time
p=phm.phm()
p.set_frequency(1000)
p.clear_hist()
p.start_hist()
time.sleep(1)
p.stop_hist()
v = p.read_hist()
f = open('hist.dat','w')
for k in v:
    ss = '%d %d\n'%(k[0], k[1] )
    f.write(ss)
```

4.10 Serial Periferal Interface (SPI) Modules

There are several devices like memory chips, high resolution ADCs, DACs etc. that can be accessed using the SPI protocol. The SPI uses three lines (CLOCK, DATA OUT and DATA IN) for commmunication. There will be one Chip Select signal also for each SPI slave device connected. Phoenix implements a Software based SPI communication, where the Soclets CH3, CH2 and CH1 are used as CLOCK, DATA OUT and DATA IN respectively. The Digital Output D3 is used as Chip Select of the ADC, D2 for the DAC and D1 for Serial EEPROM.

Library functions are available to access the ADC and DAC plug-in modules already developed. SPI functions are also accessible from Python library so that new circuits can be incorporated without changing the code at the micro-controller side.

4.10.1 Raw SPI Functions

These functions are useful for testing new SPI devices. Some SPI devices require the SCLK to be HIGH during CS is enabled, they are handled by the _bar functions.

4.10.1.1 chip_enable , chip_enable_bar

PROTOTYPE

```
None chip_enable(int device)
```

DESCRIPTION

Enable the selected SPI device by pulling the CS pin LOW. Values 0, 1, 2 selcts the devices connected to D3, D2 and D1 respectively. Some SPI devices require the SCLK to be HIGH while taking CS LOW, we need to call chip_enable_bar() for them.

USAGE

```
chip_enable(1)  # Selects the SPI whose CS is from Socket D2
```

EXAMPLE

chip_enable_bar(0) # device on D3

4.10.1.2 chip_disable

PROTOTYPE

```
None chip_disable()
```

DESCRIPTION
Disable all SPI devices by taking the CS pins (digital outputs D3, D2 and D1) HIGH.

USAGE

```
chip_disable()
```

4.10.1.3 spi_push, spi_push_bar

PROTOTYPE

```
None spi_push(int data)
```

DESCRIPTION
Sends the 8 bit number to the currently selected SPI device.

USAGE

```
spi_push(255)
```

EXAMPLE

```
chip_enable(1) # selcted the DAC on D2
data = 1000
spi_push( (data >> 8) & 255) # push high byte
spi_push(data & 255)         # then low byte
chip_disable()
```

4.10.1.4 spi_pull, spi_pull_bar

PROTOTYPE

```
int spi_pull()
```

DESCRIPTION
Reads an 8 bit number from the currently selected SPI device.

USAGE

```
dat = spi_pull()
```

EXAMPLE

```
chip_enable(0) # selcted the ADC on D3
cmd = 64 + 15
spi_push(cmd) # push command
print spi_pull()         # print ADC ID register
chip_disable()
```

4.10.2 High Resolution ADC / DAC module

(picture)
The AD/DA card has an 8 channel 24 bit ADC and a single channel 16 bit DAC. The circuit is optically isolated from Phoenix and powered by +/-12V DC supply. The software is similar to that of the built-in ADC. The maximum input voltage range of the ADC is from 0 to 2.5V, with a resolution of few microvolts. This is suitable for measuring the output of various sensors without any amplification. The input range can be reduced down to 20 mV to do more accurate measurements.

4.10.2.1 hr_set_voltage

PROTOTYPE

```
None hr_set_voltage(float mv)
```

DESCRIPTION
Sets the output of the Serial DAC from 0 to 2500 mV. Minimum stepsize is 38.1 microvolts.

USAGE

```
hr_set_voltage(500.23)
```

EXAMPLE

hr_set_voltage(100.0)

4.10.2.2 hr_select_adc

PROTOTYPE

```
None hr_select_adc(int chan)
```

DESCRIPTION

Selects any channel from 0 to 7. When differential inputs are given, channel numbers from 8 to 11 are used. The selected channel is used for subsequent digitizations.

USAGE

```
hr_select_adc(0)    # selects the first channel
```

EXAMPLE

```
hr_select_adc(8)    # select CH0 and CH1 as differential Input
```

4.10.2.3 hr_get_voltage

PROTOTYPE

```
list hr_get_voltage()
```

DESCRIPTION

Reads the analog voltage on the current ADC channel and returns a tuple. First element of the tuple is the PC time-stamp and the second element is the voltage in milli-volts. The timestamp is required for plotting slowly varying parameters as a function of time.

USAGE

```
res = p.hr_get_voltage()
```

EXAMPLES

Connect Serial DAC to Serial ADC CH0 using a piece of wire and run the following program several times. The result will be fluctuate in the microvolts range.

```
p.hr_set_voltage(500.0)
p.hr_select_adc(0)
print p.hr_get_voltage()[1] # print voltage only
```

4.10.2.4 hr_select_range

PROTOTYPE

```
None hr_select_range(int ran)
```

DESCRIPTION

Set the fullscale range for the currently selected channel. Range can be set from 0 to 7. The minimum setting for range (ran = 0) is 20 mV and the maximum range is 2.56V (ran = 7) . The voltage ranges in millivolts are [20, 40, 80, 160, 320, 640, 1280, 2560]. By default the range is set to 2.56V.

USAGE

```
hr_select_range(0)    # selects 20 mV total range.
```

EXAMPLE

```
hr_select_range(7)    # select 2.56V range
```

4.10.2.5 hr_internal_cal

PROTOTYPE

```
list hr_internal_cal(int ran)
```

DESCRIPTION

Does an internal calibration of the specified channel and returns the OFFSET and GAIN coefficients. As such there is no need of calibration if the temperature is near 25^0 .

USAGE

```
print hr_internal_cal(0)    # print OFFSET and GAIN of channel 0
```

4.10.2.6 hr_external_cal

PROTOTYPE

```
list hr_external_cal(int zero_or_fs)
```

DESCRIPTION

A system calibration can be performed by connecting the zero and full scale voltages externally. Connect the external Zero Level and call this function with 'zero_or_fs' set to 0. Then connect the full scale voltage and call the function with 'zero_or_fs' = 1. Calibration is done for the current channel.

USAGE

```
hr_external_cal(0)          # Connect Zero Level to the input
print hr_external_cal(1)    # Connect Full Scale Voltage to the input
```

4.10.3 Serial EEPROM

The Serial EEPROM module uses AT25HP512 eeprom with 64 kbytes memory and SPI interface. This can be used for data recording applications, without the PC. The SPI connections are same as explained above. The Digital Output D1 is the Chipselect for EEPROM. The function calls to read/write the SEEPROM are explained below.

4.10.3.1 seeprom_read

PROTOTYPE

```
print read_seeprom(int addr, int nbytes)
```

DESCRIPTION

Reads data from the Serial EEPROM plugin module. The starting address and the number of bytes to be read are specified. Maximum number of bytes that can be read in a single call is 256. Data is returned in a list.

USAGE

```
v = p.seeprom_read(2000, 256)
```

4.10.3.2 seeprom_verify

PROTOTYPE

list seeprom_verify(int blocknum, list data)

DESCRIPTION

This function is for checking the Serial EEPROM plugin memory. We are writing data to SEEPROM in 128 byte blocks. The first argument is the block number (from 0 to 511 in a 64K chip) and the second is a list containing the data. The data provided is first loaded in to the internal EEPROM and then copied to the SEEPROM plugin. Then it is read back and returned. If everything is fine the retunred data must be the same as the one written.

USAGE

```
v = p.seeprom_verify(0, x)
```

EXAMPLE

```
x = range(128) # makes 128 element list
v = p.seeprom_verify(0, x)
print v
```

4.11 Plotting Functions

Currently Phoenix uses the Tkinter graphics toolkit for doing graphics. Tkinter provides the basic drawing routines on its Canvas Widget. Phoenix library contains some routines that plots data returned by Phoenix, using Tkinter module.

4.11.1 plot

PROTOTYPE

None plot(data, width=400, height=300, parent=None)

DESCRIPTION

Plots the data returned by read_block and multi_read_block. Provides grid, window resizing and coordinate measurement facilities, using mouse. Any previous plot existing on the window will be deleted.

USAGE

```
p.plot(v) # v is a list returned by read_block
```

EXAMPLE

```
v = p.read_block(200, 10, 1)
p.plot(v)
```

4.11.2 plot_data

PROTOTYPE

```
None plot_data(data, width=400, height=300)
```

DESCRIPTION

Similar to plot() but with limited features. Use this inside infinite loops with fast updating, where plot() function will not work properly.

EXAMPLE

```
p.plot_data(v)
```

EXAMPLE

```
while 1:
    v = p.read_block(200, 10, 1)
    p.plot_data(v)
```

4.11.3 window

PROTOTYPE

None = window(int width=400, int height=300, Widget parent = None)

DESCRIPTION

The plot() function is the simplest way to plot the data returned by block read functions. However, if you need a finer control over the process, open a window and use functions like line(), box() etc. The function window() creates a Tkinter window on the screen. If no parent window given, a new root window is created and used as the parent. This window will be used for subsequent calls to draw objects like line, box etc. If you do not specify the dimensions, default values will be used.

USAGE

```
canvas = p.window()
```

EXAMPLE

```
p.window()
raw_input() # show it until a key is pressed
```

4.11.4 set_scale

PROTOTYPE

```
None = set_scale(float xmin, float ymin, float xmax, float ymax)
```

DESCRIPTION

For plotting graphs, it is convenient to define a global coordinate system according to the range of the values to be plotted. The set_scale() function defines the upper and lower limits of the X and Y coordinates we will be plotting. Ones it is set, the conversion from the global coordinate system to the screen coordinates will be taken care by the drawing functions.

USAGE

```
p.set_scale(0, -5000, 500, 5000)
```

EXAMPLE

```
p.set_scale(0, 100, 0, 5000)
```

4.11.5 line

PROTOTYPE

None = line(list xy, color = 'black')

DESCRIPTION

The line() function accepts a list of XY coordinates to plot a line using it. The coordinate input is of the form [(x1,y1), (x2,y2),.....]. The color is black by default.

USAGE

```
line([(0,0), (100,100)],'red')
```

EXAMPLE

```
p.window(200,200)
#p.set_scale(0,0,200,200)
p.line([(0,0),(50,100),(100,100)],'blue')
p.line([(0,0),(100,100)],'red')
raw_input()
```

Run the program to see the output. Uncomment line 2 and run it again to see the effect of set-scale function. The last line is to keep the graph on the screen until a key is pressed.

4.11.6 remove_lines

PROTOTYPE

```
None = remove_lines()
```

DESCRIPTION

Delete all the lines plotted on the window.

USAGE

```
p.remove_lines()
```

EXAMPLE

```
p.window(200,200)
p.set_scale(0,0,200,200)
p.line([(0,0), (100,100)],'blue')
raw_input('Press Enter')
p.remove_lines()
raw_input('Press Enter')
```

4.11.7 box

PROTOTYPE
None = box(list xy, color = 'black')

DESCRIPTION

The box() function accepts a list having the XY coordinates of bottom-keft and top-right corners. The coordinate input is of the form [(x1,y1), (x2,y2)].

USAGE

```
box([(0,0), (100,100)],'red')
```

EXAMPLE

```
p.window(200,200)
p.set_scale(0,0,200,200)
p.box([(0,0), (100,100)],'blue')
raw_input()
```

4.11.8 remove_boxes

PROTOTYPE

```
None = remove_boxes()
```

DESCRIPTION

Delete all the boxes plotted on the window.

USAGE

```
p.remove_boxes()
```

4.12 Disk Writing

4.12.1 save_data

PROTOTYPE

```
None save_data(data, fn='plot.dat')
```

DESCRIPTION
Save the data returned by the ADC block read functions into a file in multi column format. Default filename is 'plot.dat'; this can be overriden.

EXAMPLE

```
v = p.read_block(200, 10, 1)
p.save_data(v, 'sine.dat')
```

4.13 Plugin LCD Display

Phoenix hardware has a front side socket where an alphanumeric LCD display can be connected. The alphanumeric LCD display is not used normally, it is meant for developing and debugging micro-controller program. It is also used when stand-alone devices are made using Phoenix. However there are some Python functions also to send characters to the LCD display from the PC.

4.13.0.1 init_LCD_display

PROTOTYPE

```
None init_LCD_display()
```

DESCRIPTION
Initialize the LCD display by sending the necessary commands to the display module.

USAGE

```
p.init_LCD_display()
```

4.13.0.2 write_LCD

PROTOTYPE

```
None write_LCD(char ch)
```

DESCRIPTION
Writes a single character to the LCD display at the current cursor position.

USAGE

```
p.write_LCD_('A')
```

4.13.0.3 message_LCD

PROTOTYPE

```
None message_LCD(char message)
```

DESCRIPTION
Initialize the LCD display and writes a character string (up to 16 characters in length) to it.

USAGE

```
p.message_LCD('hello world')
```

Chapter 5

Elementary Python

5.1 Python Basics

The first reaction from most of the readers would be like "Oh. No. One more programming language. I already had tough time learning the one know now" . Cool down, Python is not harmful as the name sounds. A high school student with average intelligence can pick it up within two weeks. What is there in a programming language anyway. It allows you to create variables, like elementary algebra, and allows you to manipulate them using arithmetic and logical operators. A sequence of such statements makes the program.

Our approach here will be to study the example programs, try to guess the output and compare it with the actual output you get after executing it. Correct your understanding if they differ. To execute the program, enter the programs listed below in to a file , for example one.py , using a text editor and use the command;

```
$ python one.py
```

Remember that spaces are important, adding extra spaces in the beginning will result in a syntax error. Indentation is used for marking group of lines under loops, conditionals, functions etc. Anything on the right side of a # is treated as a comment.

5.2 Variables and Data types

Python support numeric data types like Integer, float and complex numbers. Type of the variable is decided by the value you assign to it, this method is called dynamic typing. The variables belong to certain types like 'integer', 'float', 'string' etc.

```
a = 4 # integer type
b = 1.5 # floating point type
```

```
c = 2 + 3j # imaginary part is folloed by a 'j'
print a, a * c # guesss the output
```

5.3 Compound Data types

Types that comprise smaller pieces are called compound data types. They can be used as a single item or as individual elements using the bracket [] operator.

```
a = 'I am a string' # String is a compound data type
b = [3, 4.5, 'myname'] # this is a list data type
b = (3, 4.5, 'myname') # but this is a tuple type
print a, b, a[0], b[2] # remember ! Indexing starts with zero.
```

List is a very versatile data type in Python. String and Tuple are immutable, ie. you cannot change their individual elements afterwards. But a list can be modified in many different ways.

```
x = (10, 5.4)
x[1] = 6.7 # Will give an error
s = 'no change'
s[1] = 'a' # Will also give error
x = [10, 5.4]
x[1] = 6.7
print x # will print 6.7 as the second element
x.append(100)
print x
x = range(10) # creates a list
print x , x[1] # print full list and then the second element
size = len(x)
for k in range(size):
print k, x[k] # print index and the list element indexed
for item in x: # easier way to access list elements in a for loop
print item
```

5.4 Control Statements (while , for, in , if , else & elif)

Programs generally do not execute from first to last line as shown in the previous examples. Depending upon the values of some variables it may skip some portion of the code or execute some portions several times in a loop. Looping is done by using keywords while and for . Conditional execution is achieved by using keywords if , else and elif .

```
x = 1 # program to print multiplication table of 4
while x <= 10 : # colon and indentation are important
```

```
print x * 4
x = x + 1 # increment value of x by 1
print 'Done'
x = 1
while x < 6 :
if x < 3: # conditional execution
print 'small ', x # one more level of indentation
x = x + 1
x = 1
while x < 6 :
if x < 3: # conditional execution
print 'small ', x # one more level of indentation
elif x > 3:
print 'big ', x
else
print 'equal'
x = x + 1
```

5.5 for loops in Python

The for loop in Python is more flexible than similar constructs in other languages and written using for and in keywords. It iterates on a compound data type like a list or tuple. To implement loops that will be executed a fixed number of times can be implemented using the range() function

```
a = [12, 3.2, 'hello']
for x in a:
print x
print range(10)
for n in range(10):
print n
```

5.6 Keyborad Input

Python functions to get a numeric value from the keyboard is input(). There is another function named raw_input() that accepts the input as a character string. A message can be given as argument to both the functions.

```
x = input('Enter a number ')
print x , type(x) # try this program with different inputs
x = raw-input('Enter a string ')
print 2 * x # Is it a bit crazy !
```

The function raw_input() gets the input as string. To input numeric data types with error checking, one should use this function and then convert it into numeric type.

```
s = raw-input( 'Enter a number ')
try: # try and except are error handling stuff in Python
x = float(s)
except:
print 'Invalid input. Assignng zero'
x = 0
print x
```

5.7 Formatted Output

Python supports an output formatting similar to that of the printf function in C. The format string will contain % operator followed by format specifiers. After the format string the values to be printed are given as a tuple, as shown in the examples below .

```
a = 2.0 /3
print a
print 'a = %5.3f' %(a) # upto 3 decimal places
for k in range(1,11): # A formatted multiplication table
print '5 x %2d = %2d' %(k, k*5)
```

Operators

Some of the important arithmetic and logical operators in Python are listed below.

Symbol
Function
Example
=
Assignment
a = 5
+
Addition
a = a + 5
-
Subtraction
a = a - 1
*
Multiplication
a = a * 3
**
Exponential
b = a ** 3
/
Division (integer or float)
2 / 4 (zero) , 2.0 / 4 (0.5)

==
Equality test
if x == 5:
!=
Negated equality test
if x != 5:
not
Logical negation
x = True ; y = not x
and
Logical AND
c = a and b
or
Logical OR
if a or b : print a * b
a, b = raw_input('Enter two numbers separated by comma ') # like 2,3
print a == b
print a / b # Add more below if you like

5.8 User defined functions

Large programs need to be divided into logical subsections. Python allows you to define functions using the keyword def . Python functions can be a single variable or a list.

```
def sum(a,b): # a trivial function
return a + b
print sum(3, 4)
def factorial(n): # a recursive function
if n == 0:
return 1
else:
return n * factorial(n-1)
print factorial(10)
```

5.9 Python Modules

One of the major advantages of Python is the large number of function libraries, generally called modules, to do tasks like graphics, networking, scientific computation etc. Modules are loaded by using the keyword import . Several ways of using import is explained below.

```
import math
print math.sin(0.5) # module_name.method_name is the syntax
from math import sin
```

```
print sin(0.5) # sin is imported as a local method
from math import * # import everything from math library
print sin(0.5)
```

In the second method, we need not type the module name every time. But there could be trouble if two modules imported has a function with same name. Try the following example to see that.

```
from scipy import *
x = linspace(0, 5, 10) # make a 10 element array
print sin(x) # sin function of scipy can handle array arguments
from math import * # sin function of math will be called now
print sin(x) # and will fail with the array argument
```

5.10 File Input/Output

Disk files can be opened using the function named open() that returns a file object. Files can be opened for reading or writing. There are several methods belonging to the file class that can be used for reading and writing data.

```
f = open('test.dat' , 'w')
print f # will print description of the object
f.write ('This is a test file')
f.close()
```

Above program creates a new file named 'test.dat' (any existing file with the same name will be removed in that process) and writes a string to it. The following program opens this file for reading the data.

```
f = open('test.dat' , 'r')
print f.read()
f.close()
```

Note that the data written/read are character strings. read() function can be used to read a fixed number of characters also, as shown below.

```
f = open('test.dat' , 'r')
print f.read(7) # get first seven characters
print f.read() # get the remaining ones
f.close()
```

Now we will examine how to read a text file and convert it into numeric type. First we will create a file with two columns of numbers.

```
f = open('xy.dat' , 'w')
for k in range(4):
s = '%3d %3d\n' %(k, 2*k)
f.write(s)
f.close()
```

The contents of the file will look like this.

0 0
1 2
2 4
3 6

Now we write a program to read this file and print the numbers.

```
f = open('xy.dat' , 'r')
while 1: # infinite loop
s = f.readline()
if s == '' : # Empty string means end of file
break # terminate the loop
m = s.split() # split the character string
print m # result is a list with two elements
sum = int(m[0]) , int(m[1]) # convert strings into intergers
print sum
f.close()
```

5.11 The pickle module

In the previous section we have seen that the data going to file is always treated as character strings. To preserve the data type information, we can use the pickle module, as shown below.

import pickle
f = open('test.pck' , 'w')
pickle.dump(12.3, f) # write a float type
f.close()

Now read it back and check the data type.

import pickle
f = open('test.pck' , 'r')
x = pickle.load(f)
print x , type(x) # check the type of data read
f.close()

5.12 Python in Science and Mathematics

We will use Python for plotting mathematical functions, solving equations, doing numerical computation etc. Most of the operations require working with one or two dimensional matrices. Functions capable of working directly on arrays and matrices makes the code very simple. In the beginning we will use the math module, maily to see its limitations, and then go to packages like numpy, scipy, matplotlib etc.

The first example generates xy coordinates to plot a sine wave, plotting will be done using some external program. Enter the following code to a file named sine.py , using any text editor.

```
import math
x = 0.0
while x < math.pi * 4: # plot from zero 4 pi
print x , math.sin(x)
```

The output coming to the screen can be redirected to a file as shown below, from the command prompt. You can plot the data using some program like xmgrace.

```
$ python sine.py > sine.dat
$ xmgrace sine.dat
```

There are two things we do not like in the above program. One is to deal with the loops to generate the data and the other is to use an extarnal program to plot the data. To solve the first problem, we need to use some modules that supports vectorized functions. We will use the module numpy that supports operation on compound data types like arrays and matrices. First we will learn howto make arrays (and also matrices) using numpy.

```
import numpy
x = numpy.arange(1.0, 2.0, 0.1) # start, end and step need to be given
print x , type(x)
If you want to save some typing, use an alternate form of import as shown below.
import numpy as np
x = np.arange(1.0, 2.0, 0.1) # start, end and step need to be given
print x , type(x)
Now let us examine some more functions that generate arrays.
import numpy
x = numpy.ones(5) # make some one dimensional arrays
print x
x = numpy.zeroes(5)
print x
x = numpy.random.rand(5)
print x
a = [1,2,3,4,5] # make a list
x = numpy.array(a) # and convert to a numpy array
print a
```

We can also make multi-dimensional arrays. Remember that a member of a list can be another list. The following example shows how to make a two dimensional array.

```
import numpy
a = [ [1,2] , [3,4] ] # make a list of lists
x = numpy.array(a) # and convert to a numpy array
print a
```

Dimensions of arrays can be changed using the reshape function as shown below.

```
import numpy
a = arange(10)
print a
a = a.reshape(5,2) # 5 rows and 2 columns
print a
```

We can perform operations like addition, multiplication etc. on arrays. The operation is performed on each element. To understand the behavior of array arithmetic, see the results of the following example program.

```
import numpy as np # give a shorter name
a = np.array([ [1,2] , [3,4] ]) # make a numpy array
print a
print 5 * a
print a * a
print a / a
```

You can also do matrix algebra on arrays by using certain functions. The same can be done in an easier manner if you define a matrix instread of array.

```
import numpy as np # give a shorter name
a = np.array([ [1,2] , [3,4] ]) # make a numpy array
print dot(a,a)
ia = np.linalg.inv(a) # linear algebra subpackage of numpy
print ia , ia * a # should print identity matrix
print a.T # print transpose
```

Compare the results of array and matrix data types from the example below.

```
import numpy as np # give a shorter name
a = np.array([ [1,2] , [3,4] ]) # make a numpy array
m = np.mat([ [1,2] , [3,4] ]) # make a numpy array
print m * m # Matrix multiplication
print a ** 3 # element wise cube
print m ** 3 # multiply matrix 3 times
```

5.13 The matplotlib package

There are several packages available for plotting under Python. We will use matplotlib package that produces good quality outputs and is also well documented. Matplotlib also supports a Matlab like interface.

```
import matplotlib.pyplot as p
import numpy as np
x = [1,2,3]
y = [4,5,3]
p.plot(x,y)
```

```
p.xlabel('X')
p.show()
```

Another example

```
import matplotlib.pyplot as p
import numpy as np
t = np.arange(0.0, 5.0, 0.2)
p.plot(t, t**2,'x')
p.plot(t, t**3,'ro')
p.show()
```

The following example shows plotting in multiple windows.

```
import matplotlib.pyplot as p
import numpy as np
t = np.arange(0.0, 5.0, 0.2)
p.figure(1)
p.subplot(211)
p.xlabel('x - axis')
p.plot(t , t**2, 'o')
p.subplot(212)
p.plot(t, t**3)
p.show()
```

To learn more about matplotlib refer to the manual.

5.14 Plotting mathematical functions

One of our objectives is to understand different mathematical functions by plotting them graphically. We will use nump arange, linspace and logspace functions from numpy to generate the input dataset and the vectorized functions to calculate the functions. The results are plotted using the pyplot subpackage from matplotlib.

```
import numpy as np
import matplotlib.pyplot as p
x = np.linspace(0, 2 * np.pi, 101) # why not 100 points ?
y = np.sin(x)
p.plot(x,y)
p.show()
```

Bibliography

[1] www.iuac.res.in

[2] http://www.python.org/

[3] http://linuxgazette.net/111/pramode.html

[5] http://www.iuac.res.in/%7Eelab/phoenix/docs/phm_book.pdf