Phoenix Programmer's Manual

Ajith Kumar B.P
Inter-University Accelerator Centre
New Delhi 110 067

Version 1 (Jul, 2008)

Contents

Chapter 1

Introduction

Phoenix[1] is an equipment that can be used for developing computer interfaced science experiments without getting into the details of electronics or computer programming. For developing experiments this is a middle path between the *pushbuttonsystems* and the *developfromscratch* approach. Emphasis is on leveraging the power of personal computers for experiment control, data acquisition and processing. Phoenix is developed by Inter-University Accelerator Centre and the details can be found at www.iuac.res.in. IUAC is an autonomous research institute of University Grants Commission, India, providing particle accelerator based research facilities to the Universities. This manual explains the Python language library handling the communication between the PC and Phoenix. A brief description of Phoenix Hardware is given in the beginning.

1.1 Phoenix Top Panel

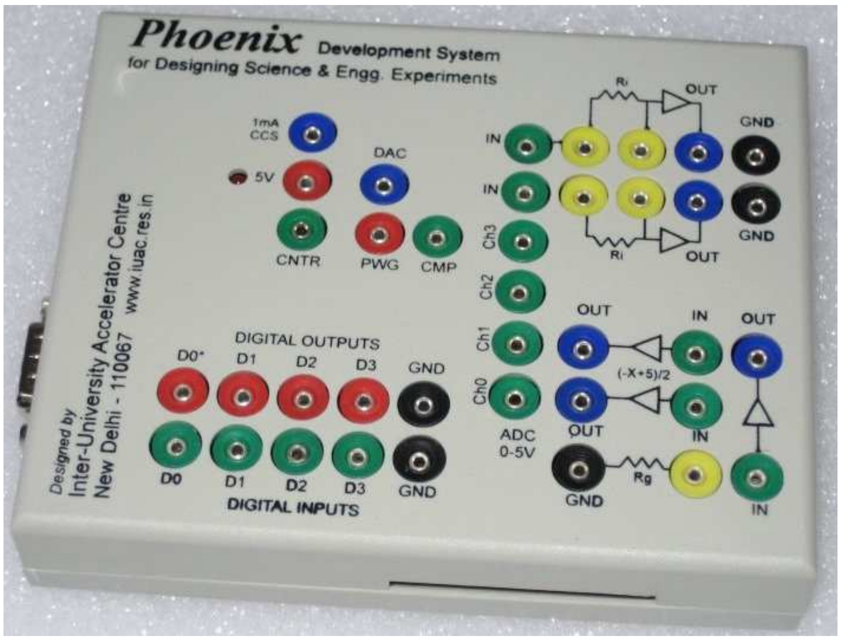
On the top panel you will find several 2mm banana sockets with different colors. Their functions are briefly explained below. The 9V DC input and the PC communication port (USB or Serial) connector are on the left side. The slot on the front side is for connecting the alphanumeric LCD display module. The connector for programming the ATmega16 micro-controller inside is on the backside.

1. 5V OUT - This is a regulated 5V power supply that can be used for powering external circuits. It can deliver only up to 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. Input to them should be 0V or 5V normally. It might sometimes be necessary to connect analog outputs swinging

[1]Physics with Homemade Equipment and Innovative Experiments.



Phoenix Development System
for Designing Science & Engg. Experiments
1mA CCS
5V
DAC
CNTR
PWG
CMP
DIGITAL OUTPUTS
D0* D1 D2 D3 GND
D0 D1 D2 D3 GND
DIGITAL INPUTS
IN
IN
Ch3
Ch2
Ch1
Ch0
ADC 0-5V
OUT
OUT
GND
GND
Ri
OUT
IN
(-X+5)/2
IN
Rg
GND
IN
Designed by Inter-University Accelerator Centre New Delhi - 110067 www.iuac.res.in

Figure 1.1: Phoenix hardware.

between -5V to +5V to the digital inputs. In this case, you MUST feed the input through a 1K resistor in series to protect it from damage.

4. ADC inputs - four GREEN sockets marked CH0 to CH3. The input voltage should be between 0V and 5V. When you need to digitize -5V to +5V range signals, feed them though the level shifting amplifiers. They are marked as (-x+5)/2 on the panel and that is what they do. Invert your signal, add 5Volts to it and divide the result by two.

5. PWG - Programmable Waveform Generator. This output can be programmed to generate a 5V square wave. The frequency can be varied from 15 Hz to 4 MHz, but all the intermediate frequencies are not possible.

6. DAC - 8 bit Digital to Analog Converter output. We really do not have a proper DAC on Phoenix. It is made by filtering a variable duty cycle pulse generated on the PWG socket. Due to this you can only use either PWG or DAC at a time.

7. CMP - Analog Comparator negative input, the positive input is tied to the internal 1.23 V reference. In most of the cases, the software treats this like the Digital Inputs. CMP input is capable of triggering a hardware interrupt of the micro-controller inside.

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 used

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

1.1.1 Things to be careful about when using Phoenix

1. The digital output pins can drive only 5mA. Don't connect any load less than 1K Ohm to them.
2. Digital output D0 is transistor buffered and can provide 100 mA. Don't use it for timing controls.
3. Digital and ADC inputs should not be negative or above 5V, ie. should be from 0 to 5V.
4. Variable gain amplifier outputs should be connected to Digital Inputs only through a One KOhm series resistor.
5. Amplifier inputs should be within -5 to +5V range.
6. Output pins should not be tied together.
7. Do not draw more than 100mA current from the 5V regulated supply. Take necessary protections against back emf when connecting inductive loads like motors or relay coils.
8. Forcibly Inserting Multi Meter probes with diameter more than 2 mm to the front panel sockets will damage them.
9. And, don't pour coffee into the sockets !

1.2 Software Installation

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

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, inside directory 'phoenix/software'. The installation process is explained below.

```
#unzip pyserial-2.2.zip
#cd pyserial-2.2
#python setup.py build
#cd ..
#tar zxf pyusb-0.4.1.tar.gz
#cd pyusb-0.4.1
#python setup.py install
```

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 [3]. 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.

GUI programs are available for most of the Phoenix based experiments. You need to read this manual only if you want to design new experiments using Phoenix.

1.3 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.

```
ajith@pc230:~$ python
```

[2]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.

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

Python 2.4.4 (#2, Apr 5 2007, 20:11:18) [GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2 Type "help", "copyright", "credits" or "license" for more information.

> > > 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:

ajith@pc230:~$ 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.

Chapter 2

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.

2.1 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.

2.1.1 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.

2.2 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.

2.2.1 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()
```

2.3 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.

2.3.1 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.

2.4 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.

2.4.1 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
```

2.4.2 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
```

2.5 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.

2.5.1 ADC Settings

2.5.1.1 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
```

2.5.1.2 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
```

2.5.2 Single Reads

2.5.2.1 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
```

2.5.2.2 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]
```

2.5.2.3 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()
```

2.5.2.4 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)
```

2.5.3 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.

2.5.3.1 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)
```

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

2.5.3.2 multi_read_block

PROTOTYPE

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

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).

2.5.4 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.

2.5.4.1 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)
```

2.5.4.2 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()
```

2.5.5 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.

2.5.5.1 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.

2.5.5.2 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.

2.5.5.3 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()
```

2.5.5.4 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.

2.5.5.5 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
```

2.5.5.6 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()
```

2.6 Waveform Generation and Frequency Counter

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

2.6.1 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.

2.6.1.1 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()
```

2.6.2 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.

2.6.2.1 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)
```

2.6.2.2 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
```

2.6.2.3 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)
```

2.6.2.4 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()
```

2.6.3 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()
```

2.7 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.

2.7.0.1 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)
```

2.7.0.2 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)
```

2.7.0.3 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.

2.7.0.4 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.

2.8 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.

2.8.0.5 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)
```

2.8.0.6 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)
```

2.8.0.7 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)
```

2.8.0.8 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)
```

2.9 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.

2.9.1 start_hist

PROTOTYPE

```
None start_hist()
```

DESCRIPTION
Enable the CMP interrupts to start histogramming.

USAGE

```
p.start_hist()
```

2.9.2 stop_hist

PROTOTYPE

```
None stop_hist()
```

DESCRIPTION
Disable the CMP interrupts to stop histogramming.

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

2.9.3 clear_hist

PROTOTYPE

```
None clear_hist()
```

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

USAGE

```
p.clear_hist()
```

2.9.4 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)
```

2.10 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.

2.10.1 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.

2.10.1.1 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

2.10.1.2 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()
```

2.10.1.3 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()
```

2.10.1.4 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()
```

2.10.2 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.

2.10.2.1 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)

2.10.2.2 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
```

2.10.2.3 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
```

2.10.2.4 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
```

2.10.2.5 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
```

2.10.2.6 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
```

2.10.3 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.

2.10.3.1 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)
```

2.10.3.2 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
```

2.11 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.

2.11.1 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)
```

2.11.2 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)
```

2.11.3 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
```

2.11.4 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)
```

2.11.5 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.

2.11.6 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')
```

2.11.7 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()
```

2.11.8 remove_boxes

PROTOTYPE

```
None = remove_boxes()
```

DESCRIPTION

Delete all the boxes plotted on the window.

USAGE

```
p.remove_boxes()
```

2.12 Disk Writing

2.12.1 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')
```

2.13 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 few functions to send characters to the LCD display from the PC.

2.13.0.1 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()
```

2.13.0.2 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')
```

2.13.0.3 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 3

Programs for Simple Experiments

This section is to provide more elaborate, complete working examples for the functions explained in the previous chapter.

3.0.1 Capacitor Charging/Discharging

```
# Connect the capacitor from CH0 to GND. resistor from D3 to CH0
import phm, time
p=phm.phm()
p.select_adc(0)
p.set_adc_size(2)
p.write_outputs(8)
time.sleep(1)
p.enable_set_low(3)
v=p.read_block(200,25,0)
p.plot_data(v)
p.save_data(v, 'cap.dat')
print 'Press any Key to Exit'
raw_input()
```