Department of Electrical Engineering
IIT Bombay

EE712

Project Report

Smart Energy Monitoring with appliance control for enforcing behaviour change in the utilisation of Electrical Energy

Group Members:

1. Rajat Sankhla
2. Dhruv Kachhiya
3. Kaustav Jana

Instructors: Prof. Laxmeshaa Somappa, Prof. Dinesh Sharma

Abstract: In the era of post-pay, we enjoy the comfort of being carefree while using the service. All these fantasies of paying later sound good for using electricity. Paying the electricity bill later often leads to the wastage of energy, generating some units that are not actively used for any purpose. If we are able to save 1 unit of electricity at the user end then unnecessary generation of 2 units of energy can be saved at power stations thereby controlling the pollution levels.

Objective

To develop an IOT-based smart energy metering system that will track the Energy consumption of a household and provide it with energy consumption data and past trends over a mobile application based on which a user can purchase units for the next month and also provide an option to control any load remotely.

Hardware Used

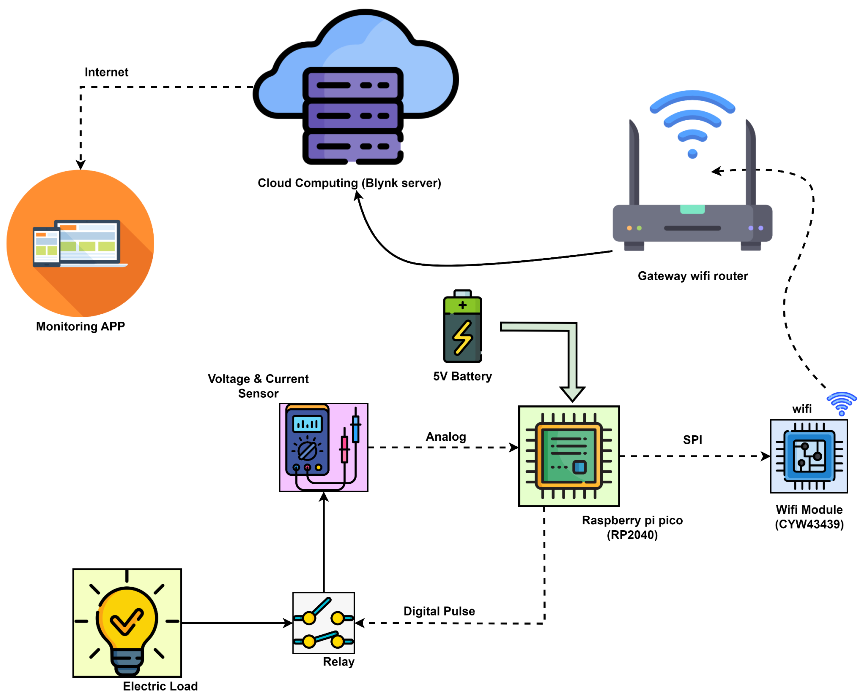
1. Raspberry Pi Pico W
2. ZMPT101B (voltage sensor)
3. ACS712 (current sensor)
4. 250V, 10A relay module
5. 200W light bulbs (Load)
6. BJT
7. 7805 (Linear regulator)

Department of Electrical Engineering
IIT Bombay

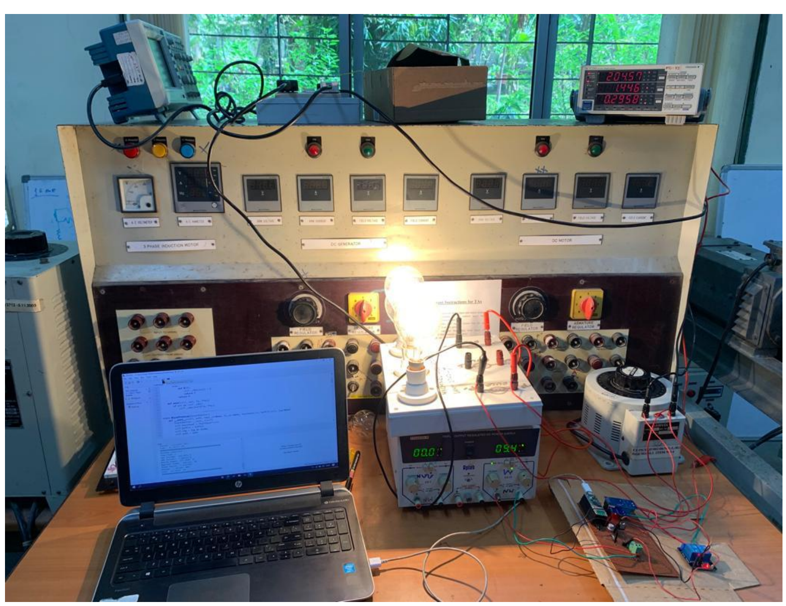
EE712

Project Report

Project Overview Diagram:



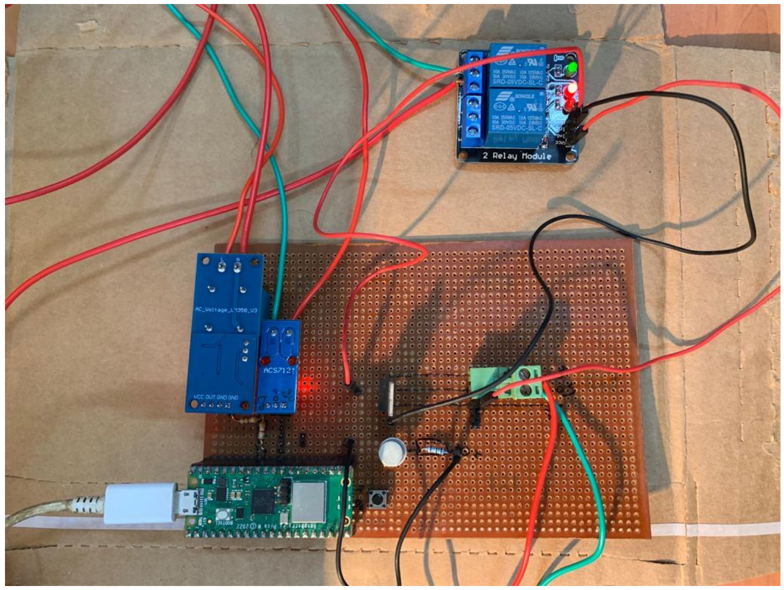
Setup Photo:



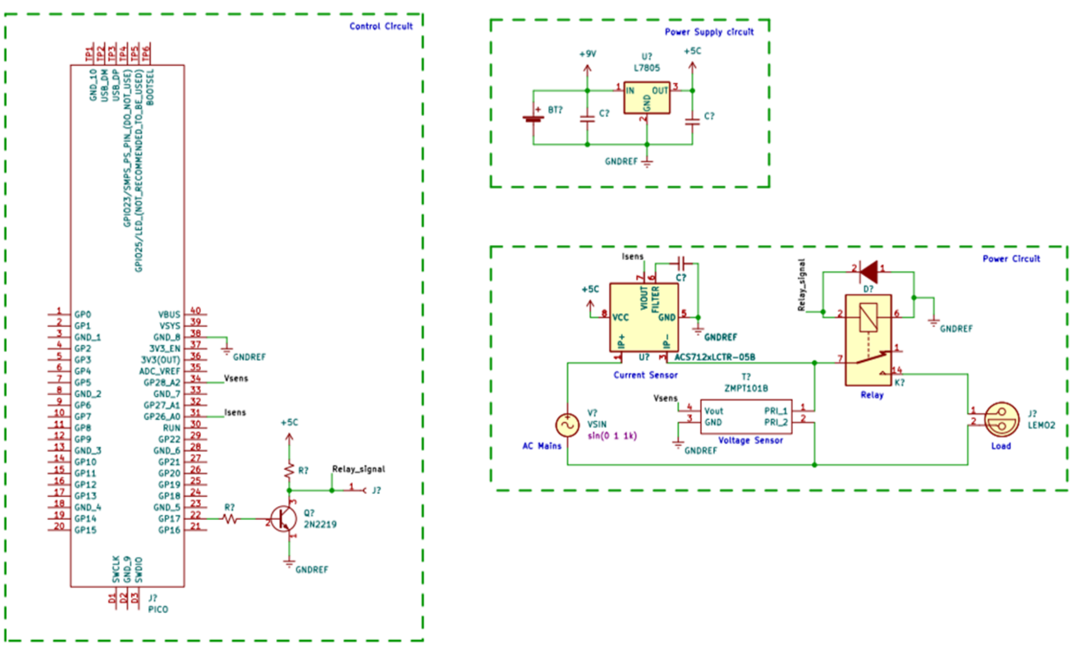
Department of Electrical Engineering
IIT Bombay

EE712

Project Report



Interfacing Diagram



Implementation details

Hardware

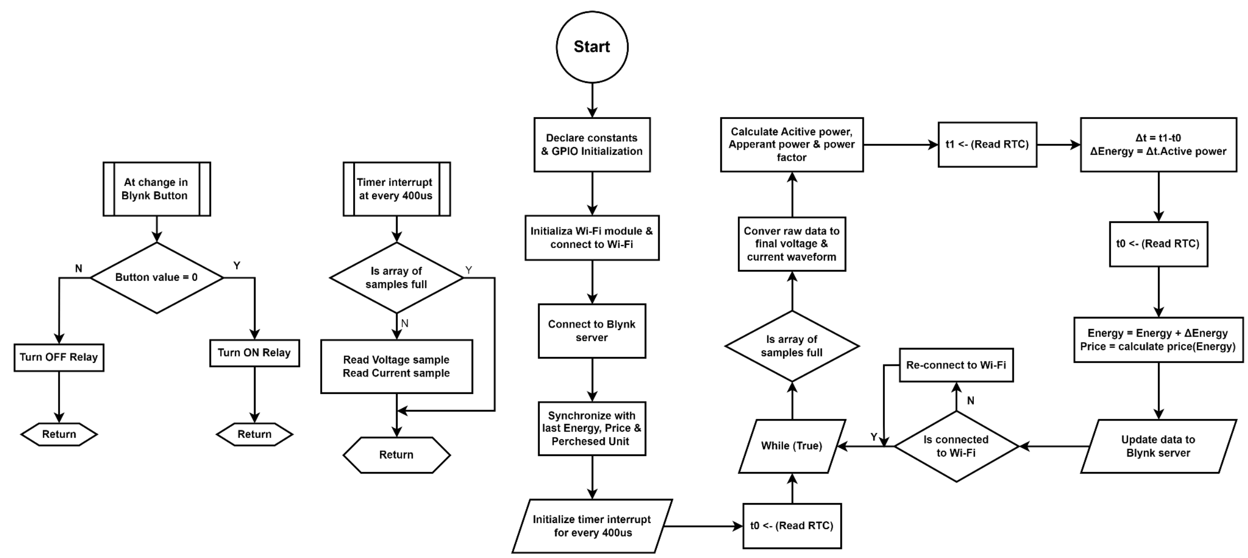
Raspberry Pi Pico W which comes with inbuilt hardware support for Wifi is used because of its low cost, small size and suitable performance for our application.

ZMPT101B is used as a voltage sensor with 5V analog output but 12-bit ADC of PICO can interface analog signals up to 3.3 Volts which it understands as a digital value of 4095 so a voltage divider circuit was used to scale down the maximum analog voltage read by ADC to under 3.3 volts. When no input is provided the output of this sensor is 2.5 volts (Vcc/2).

ACS712 is used as a current sensor which is calibrated to measure currents up to 3 A, this sensor also maps any current between 0 to 5A to 0 to 5V. When no input is provided the output of this sensor is 2.5 volts (Vcc/2).

A relay is used to cut off the load when commanded by the user using input from the GPIO pin of the PICO. A BJT is used to drive the INT pin of the relay.

Firmware logic



Firmware was written in the Thonny IDE, and the complete code is written in micro-python which is a language derived from Python for microcontroller applications.

In the main loop of code, the PICO always checks whether it is connected to the wifi or not, if it is found connected then it continues to the next reading else tries to connect when the wifi network is available. We have used the timer interrupt for sampling the voltage & current values. At every 400us (2.5kHz sampling frequency) PICO goes into the timer interrupt subroutine and reads the ADC for voltage & current. Depending on the current & voltage the rms value of voltage & current is calculated. On top of that the active, reactive & apparent power and power factor are calculated. Based on the active power & time the energy consumption & corresponding cost are calculated.

In addition to that, we also implemented the event handler for the load switch present on the Blynk server. Whenever the switch is toggled on the Blynk server, it goes into the event handler subroutine. Which checks the state of the switch and toggles the relay.

Calculation:

Active power calculation

$$P = \sum_{i=0}^{N} \frac{V_i . I_i}{N}$$

Rms voltage calculation

$$V_{rms} = \sqrt{\sum_{i=0}^{N} \frac{(V_i)^2}{N}}$$

Rms Current calculation

$$I_{rms} = \sqrt{\sum_{i=0}^{N} \frac{(I_i)^2}{N}}$$

Apparent power calculation

$$S = V_{rms} . I_{rms}$$

Power factor calculation

$$PF = \frac{P}{S}$$

Where N is the total number of samples (800) for calculation of one reading.

Department of Electrical Engineering
IIT Bombay

EE712

Project Report

Features Implemented

1. **Protection**: The relay trips when the voltage goes below 185 Volts and above 245 Volts.

2. **Auto-Resumption of Data Flow to the Cloud:** Resumption of power after an electricity cut doesn't reset the data stream in the cloud. When the electricity is disrupted PICO saves the last energy reading and starts updating the data on the cloud from the same point where it left off.

3. Continuous Energy measurement in the absence of Wifi and updation of data to the cloud when the Wifi signal is reached again.

Challenges

1. The challenge in the Accurate Measurement of Active Power and power factor is due to the delay in ADC reading. In the timer interrupt subroutine we are reading the voltage first and then we read the current. As there is an ADC conversion time delay between the measurement of voltage and current is approximately between 35us to 60us. As this current & voltage reading is not sensed at the same time the active power calculation and power factor calculations do not exactly match the pre-calibrated meter readings.

2. Inaccuracy in the voltage reading due to noise in the voltage sensor: The voltage sensor outputs an analog voltage with an offset of 2.5 volts, when no voltage is applied to the sensor reads 2.5 volts, this offset becomes noisy when it is read by the ADC of PICO which when being subtracted from the reading of ADC when voltage is applied introduces an error in measurement due to noise at zero voltage measurement.

Solutions

1. By implementing the same algorithm or at least the timer interrupt subroutine in c or assembly language. We can reduce the time delay between the reading of voltage & current which can potentially reduce the error in the measurement of active power & power factor. This can be done in future work as a future scope of improvement.

2. Challenge 2 was overcome by taking a moving average of the noisy data read by the ADC at 0 voltage and then subtracting this data from the value read by the ADC at non-zero voltage.

Future Work and Scope of Improvement

1. Inaccuracy introduced in the power factor and power calculation due to the challenge one may be eliminated by compensating for the delay in storing current samples from the voltage samples. It was suggested to consider the first reading of voltage as the reading where current samples storage starts.

2. Improvement in the hardware: It was suggested to make the device more compact with 2 output ports to insert the load, the device must look like a big phone charger.

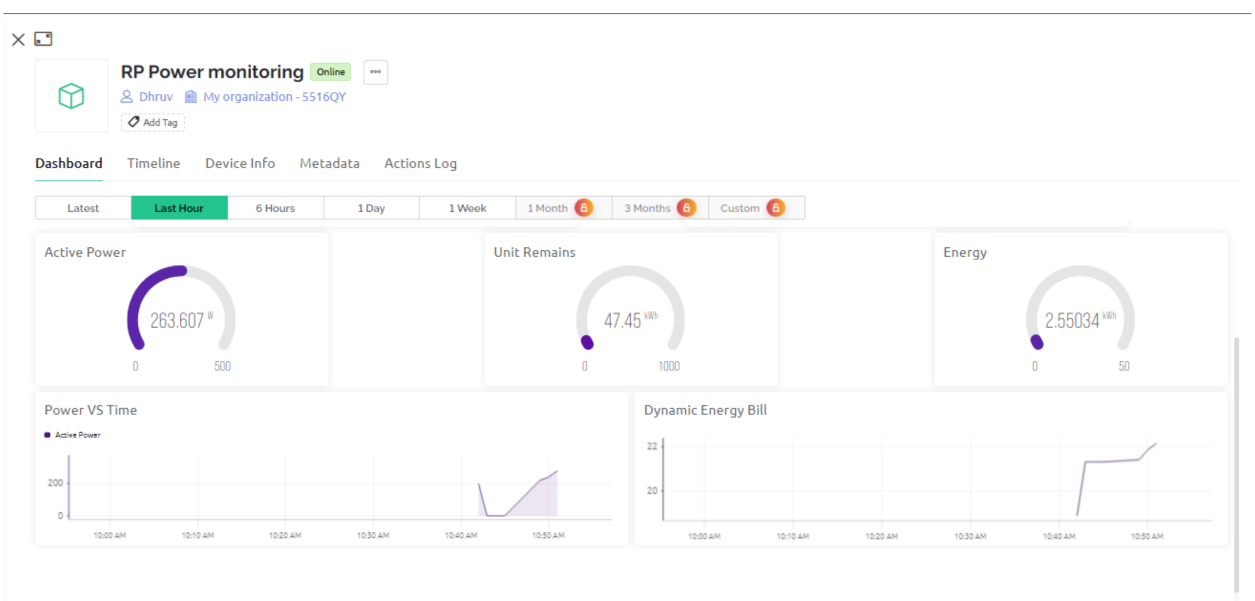
Calibration Procedure and Reference:

Yokogawa power quality analyzer was used to calibrate the voltage and current reading of the sensors.

Voltage and currents are calibrated by matching the Peak analog values of the sensor and the actual supply voltage fed to the sensor.

$$Measured\ value\ of\ mains = \frac{ADC\ reference\ voltage}{65536} \times \frac{Peak\ of\ AC\ mains}{Peak\ value\ of\ sensor\ output}$$

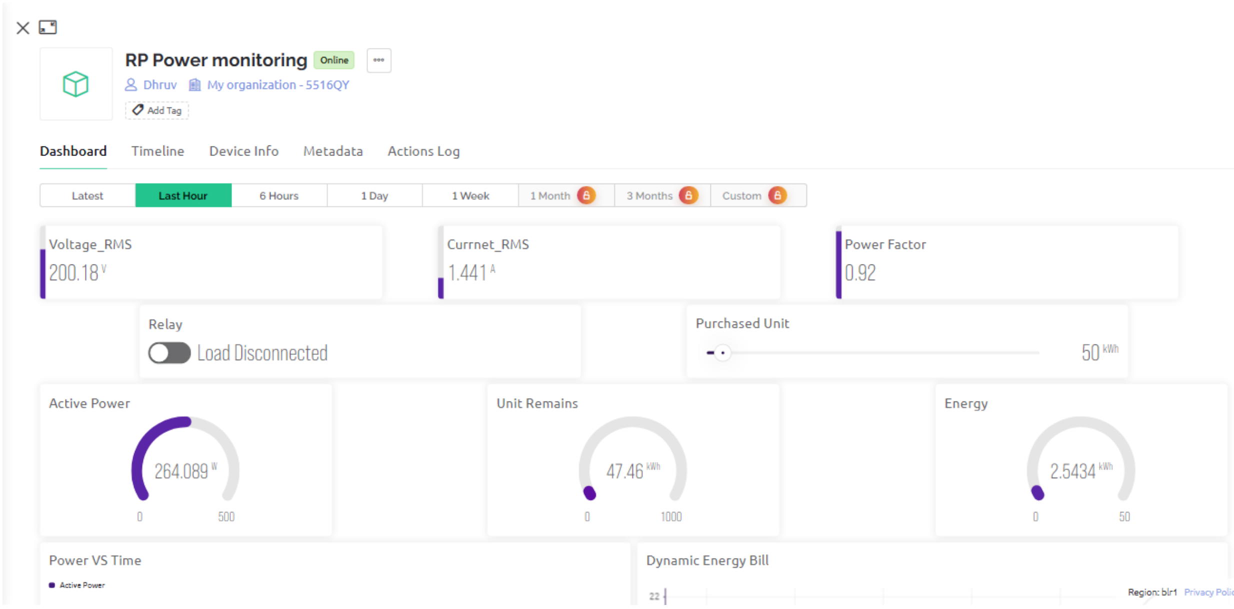
Dashboard Pictures



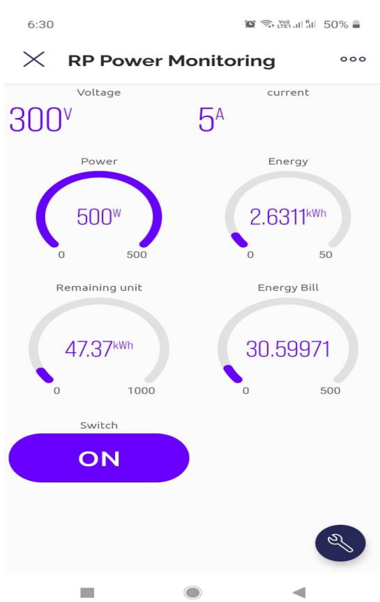
Department of Electrical Engineering
IIT Bombay

EE712

Project Report



Mobile App Pictures



Conclusion: What we proposed:

1. Calculates the energy and displays past energy trend for the user and suggest monthly units the user must purchase.
2. Functionality for loading the number of units from the dashboard.
3. Calculating and Displaying the energy budget for the remaining days of the month.
4. Shedding of load on preset daily unit consumption like AC, and geyser.
5. Dynamic unit pricing.

What we did:

All the above 5 points were implemented along with three extra features.

1. Auto tripping of load in the case of over voltage and under voltage.
2. Auto-Resumption of Data Flow to the Cloud.
3. Continuous Energy measurement in the absence of Wifi and updation of data to the cloud when the Wifi signal is reached again.

Acknowledgement:

We would like to express our utmost gratitude towards Prof. Laxmesha and Prof. Dinesh Sharma for allowing, motivating and pushing us to achieve the feat with our sincere efforts. Without their guidance, it would not have been possible for us to learn the necessary things and complete the project. We would also like to thank WEL lab RAs and staff for helping us throughout the project whether it was related to hardware components.

```python
# Blynk Library specifeid -> ________________________________________________________________-

# Copyright (c) 2015-2019 Volodymyr Shymanskyy. See the file LICENSE for copying permission.

__version__ = "1.0.0"

import struct
import time
import sys
import os

try:
    import machine
    gettime = lambda: time.ticks_ms()
    SOCK_TIMEOUT = 0
except ImportError:
    const = lambda x: x
    gettime = lambda: int(time.time() * 1000)
    SOCK_TIMEOUT = 0.05

def dummy(*args):
    pass

MSG_RSP = const(0)
MSG_LOGIN = const(2)
MSG_PING  = const(6)

MSG_TWEET = const(12)
MSG_NOTIFY = const(14)
MSG_BRIDGE = const(15)
MSG_HW_SYNC = const(16)
MSG_INTERNAL = const(17)
MSG_PROPERTY = const(19)
MSG_HW = const(20)
MSG_HW_LOGIN = const(29)
MSG_EVENT_LOG = const(64)

MSG_REDIRECT  = const(41)  # TODO: not implemented
MSG_DBG_PRINT  = const(55) # TODO: not implemented

STA_SUCCESS = const(200)
STA_INVALID_TOKEN = const(9)

DISCONNECTED = const(0)
CONNECTING = const(1)
CONNECTED = const(2)

print("""
    ___  __          __
   / _ )/ /_ _____  / /__
  / _  / / // / _ \\/  '_/
 /____/_/\\_, /_//_/_/\\_\\
        /___/ for Python v""" + __version__ + " (" + sys.platform + ")\n")

class EventEmitter:
    def __init__(self):
        self._cbks = {}

    def on(self, evt, f=None):
        if f:
            self._cbks[evt] = f
        else:
            def D(f):
                self._cbks[evt] = f
                return f
            return D

    def emit(self, evt, *a, **kv):
```

```
        if evt in self._cbks:
            self._cbks[evt](*a, **kv)


class BlynkProtocol(EventEmitter):
    def __init__(self, auth, tmpl_id=None, fw_ver=None, heartbeat=50, buffin=1024, log=None):
        EventEmitter.__init__(self)
        self.heartbeat = heartbeat*1000
        self.buffin = buffin
        self.log = log or dummy
        self.auth = auth
        self.tmpl_id = tmpl_id
        self.fw_ver = fw_ver
        self.state = DISCONNECTED
        self.connect()

    def virtual_write(self, pin, *val):
        self._send(MSG_HW, 'vw', pin, *val)

    def send_internal(self, pin, *val):
        self._send(MSG_INTERNAL,  pin, *val)

    def set_property(self, pin, prop, *val):
        self._send(MSG_PROPERTY, pin, prop, *val)

    def sync_virtual(self, *pins):
        self._send(MSG_HW_SYNC, 'vr', *pins)

    def log_event(self, *val):
        self._send(MSG_EVENT_LOG, *val)

    def _send(self, cmd, *args, **kwargs):
        if 'id' in kwargs:
            id = kwargs.get('id')
        else:
            id = self.msg_id
            self.msg_id += 1
            if self.msg_id > 0xFFFF:
                self.msg_id = 1

        if cmd == MSG_RSP:
            data = b''
            dlen = args[0]
        else:
            data = ('\0'.join(map(str, args))).encode('utf8')
            dlen = len(data)

        self.log('<', cmd, id, '|', *args)
        msg = struct.pack("!BHH", cmd, id, dlen) + data
        self.lastSend = gettime()
        self._write(msg)

    def connect(self):
        if self.state != DISCONNECTED: return
        self.msg_id = 1
        (self.lastRecv, self.lastSend, self.lastPing) = (gettime(), 0, 0)
        self.bin = b""
        self.state = CONNECTING
        self._send(MSG_HW_LOGIN, self.auth)

    def disconnect(self):
        if self.state == DISCONNECTED: return
        self.bin = b""
        self.state = DISCONNECTED
        self.emit('disconnected')

    def process(self, data=None):
        if not (self.state == CONNECTING or self.state == CONNECTED): return
        now = gettime()
```

```
        if now - self.lastRecv > self.heartbeat+(self.heartbeat//2):
            return self.disconnect()
        if (now - self.lastPing > self.heartbeat//10 and
            (now - self.lastSend > self.heartbeat or
             now - self.lastRecv > self.heartbeat)):
            self._send(MSG_PING)
            self.lastPing = now

        if data != None and len(data):
            self.bin += data

        while True:
            if len(self.bin) < 5:
                break

            cmd, i, dlen = struct.unpack("!BHH", self.bin[:5])
            if i == 0: return self.disconnect()

            self.lastRecv = now
            if cmd == MSG_RSP:
                self.bin = self.bin[5:]

                self.log('>', cmd, i, '|', dlen)
                if self.state == CONNECTING and i == 1:
                    if dlen == STA_SUCCESS:
                        self.state = CONNECTED
                        dt = now - self.lastSend
                        info = ['ver', __version__, 'h-beat', self.heartbeat//1000, 'buff-in', self.buffin, 'dev', sys.platform+'-py']
                        if self.tmpl_id:
                            info.extend(['tmpl', self.tmpl_id])
                            info.extend(['fw-type', self.tmpl_id])
                        if self.fw_ver:
                            info.extend(['fw', self.fw_ver])
                        self._send(MSG_INTERNAL, *info)
                        try:
                            self.emit('connected', ping=dt)
                        except TypeError:
                            self.emit('connected')
                    else:
                        if dlen == STA_INVALID_TOKEN:
                            self.emit("invalid_auth")
                            print("Invalid auth token")
                        return self.disconnect()
            else:
                if dlen >= self.buffin:
                    print("Cmd too big: ", dlen)
                    return self.disconnect()

                if len(self.bin) < 5+dlen:
                    break

                data = self.bin[5:5+dlen]
                self.bin = self.bin[5+dlen:]

                args = list(map(lambda x: x.decode('utf8'), data.split(b'\0')))

                self.log('>', cmd, i, '|', ','.join(args))
                if cmd == MSG_PING:
                    self._send(MSG_RSP, STA_SUCCESS, id=i)
                elif cmd == MSG_HW or cmd == MSG_BRIDGE:
                    if args[0] == 'vw':
                        self.emit("V"+args[1], args[2:])
                        self.emit("V*", args[1], args[2:])
                elif cmd == MSG_INTERNAL:
                    self.emit("internal:"+args[0], args[1:])
                elif cmd == MSG_REDIRECT:
                    self.emit("redirect", args[0], int(args[1]))
                else:
```

```
                    print("Unexpected command: ", cmd)
                    return self.disconnect()

import socket

class Blynk(BlynkProtocol):
    def __init__(self, auth, **kwargs):
        self.insecure = kwargs.pop('insecure', False)
        self.server = kwargs.pop('server', 'blynk.cloud')
        self.port = kwargs.pop('port', 80 if self.insecure else 443)
        BlynkProtocol.__init__(self, auth, **kwargs)
        self.on('redirect', self.redirect)

    def redirect(self, server, port):
        self.server = server
        self.port = port
        self.disconnect()
        self.connect()

    def connect(self):
        print('Connecting to %s:%d...' % (self.server, self.port))
        s = socket.socket()
        s.connect(socket.getaddrinfo(self.server, self.port)[0][-1])
        try:
            s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
        except:
            pass
        if self.insecure:
            self.conn = s
        else:
            try:
                import ussl
                ssl_context = ussl
            except ImportError:
                import ssl
                ssl_context = ssl.create_default_context()
            self.conn = ssl_context.wrap_socket(s, server_hostname=self.server)
        try:
            self.conn.settimeout(SOCK_TIMEOUT)
        except:
            s.settimeout(SOCK_TIMEOUT)
        BlynkProtocol.connect(self)

    def _write(self, data):
        #print('<', data)
        self.conn.write(data)
        # TODO: handle disconnect

    def run(self):
        data = b''
        try:
            data = self.conn.read(self.buffin)
            #print('>', data)
        except KeyboardInterrupt:
            raise
        except socket.timeout:
            # No data received, call process to send ping messages when needed
            pass
        except: # TODO: handle disconnect
            return
        self.process(data)


# MY MAIN CODE START HERE : ->

#load libraries
import machine
import time
```

```
from machine import Timer,ADC,Pin
import array
import math
import network

# dynamic pricing unit price :
RS_PER_KWH_100   =       4.73
RS_PER_KWH_300   =       7.33
RS_PER_KWH_500   =       10.98
RS_PER_KWH_above =       11.63

ENERGY_COFF = 3600e-9    # conversion coffetiant form (W-usec) to (kWH)
ZERO_VOLT1 =  24600#24646        # Median of Values read at zero voltage
ZERO_CURT1 = 49051#49051         # Median of Values read at zero cuurent

SAMPLE = 800             # taking allmost 16 cycles to calculate
SAMPLE_FREQ = 2500       #sample after each 400us

voltage_CF1 = 0.05035025061
current_CF1 = 0.0002730159525

connection_flag = 0              # set if connected to wi-fi

voltage1 = array.array('I', (0 for _ in range(SAMPLE))) #array initialisation with 0
current1 = array.array('I', (0 for _ in range(SAMPLE))) #array initialisation with 0

pre_voltage1 = array.array('i', (0 for _ in range(SAMPLE))) # the previous cycle data
pre_current1 = array.array('i', (0 for _ in range(SAMPLE)))

final_voltage1 = array.array('f', (0 for _ in range(SAMPLE)))
final_current1 = array.array('f', (0 for _ in range(SAMPLE)))

i = 0

# PIN CONFIGRATION -> ______________________________________________________________________---------
#Create outputs for relay
# When the relay in ON the load is connceted and OFF then load is not connected
relay   = machine.Pin(17,machine.Pin.OUT,value = 1)  # to cut the load connected to load1 (by default ON)
led = machine.Pin('LED',machine.Pin.OUT,value = 1)
#Create the ADC pin for voltage & current sensing
voltage1_pin = ADC(Pin(28))  # ADC2 channel
current1_pin = ADC(Pin(26))  # ADC1 channel

# USER DEFINED FUNCTIONS -> __________________________________________________________________------


# energy calculating function depend of energy consumed:
def find_cost(my_energy):
    if(my_energy <= 0.1):
        my_cost = my_energy*RS_PER_KWH_100
        return my_cost
    elif(my_energy > 0.1 and my_energy <= 0.3):
        my_cost = my_energy*RS_PER_KWH_300
        return my_cost
    elif(my_energy > 0.3 and my_energy <= 0.5):
        my_cost = my_energy*RS_PER_KWH_500
        return my_cost
    else:
        my_cost = my_energy*RS_PER_KWH_above
        return my_cost

# rms calculating function:
def RMS_cal(arry):
    squar_sum = sum(arry[value]*arry[value] for value in range(len(arry)))
    avg = squar_sum/(len(arry))
    return math.pow(avg,0.5)
```

```
# def median(arry):
#     my_sorted = sorted(arry)
#     kth = (int)(len(arry)/2)
#     return my_sorted[kth]

# for calculation of all active reactive & apparent power as well as power factor
def all_cal(arr_volt, arr_cur):

    if(len(arr_volt) == len(arr_cur)):
        len_arr = len(arr_volt)
        rms_volt = RMS_cal(arr_volt)
        rms_cur = RMS_cal(arr_cur)
        app_power = rms_volt*rms_cur
        act_power = abs((sum(arr_volt[value]*arr_cur[value] for value in range(len_arr)))/(len_arr))
#         react_power = math.pow(((app_power*app_power)-(act_power*act_power)),0.5)
        pf = act_power/app_power
        return app_power,act_power,pf,rms_volt,rms_cur
    else:
        print("array length must be same")
        return 0

# finction to connect to the wifi need the pass the network switch as argument
def con_wifi(wlan):
    global connection_flag
    wlan.active(True)
    wlan.connect(SSID,PASSWORD)
    max_wait = 1
    while max_wait > 0:
        if (wlan.status() < 0 or wlan.status() >= 3):
            break
        max_wait -= 1
        print('waiting for connection...')
        time.sleep(1)

    # Handle connection error
    if wlan.status() != 3:
        print("network connection failed")
        connection_flag = 0
#         raise RuntimeError('network connection failed')

    else:
        print('connected')
        connection_flag = 1
        status = wlan.ifconfig()
        print( 'ip = ' + status[0] )


# function will check the connection and if desconnected then reconnect
def reconnect(wlan):
    global connection_flag
    if(wlan.isconnected() == 0):
        con_wifi(wlan)
    else:
        print("Already connected !")
        connection_flag = 1


# call back functions for IRQ or Blynk Protocol : -> -----
#Create handlers for the timer IRQ
    # every 400usec store the sample for voltage & cuurent
def ADC_read(Source): #what is the sense of writing Source ?
    global i
    if(i < SAMPLE):
        voltage1[i] = voltage1_pin.read_u16() #store only sample number of values
        current1[i] = current1_pin.read_u16() #store only sample number of values
        #print(voltage1[i])
        #print(current1[i])
```

```
        i = i+1

# conncet to the wifi -> _______________________________________________________________-----------

# TODO: May be need to change based on the wifi
SSID = "MCLAB"
PASSWORD = "MCLAB123"
BLYNK_AUTH = 'CFlM-_422CZzoMKjUeFwvcGwravH69N_' # blynk token

# define the network swithc for wifi connection
net_switch = network.WLAN(network.STA_IF)

#cycles = 0
energy = 0
cost = 0
delta_energy = 0
Unit_purchesed = 0
Unit_remain = 0
started = 0

#  INTIALIZATION CODE -> _______________________________________________________________------------

# connect to the wifi:
for Dh in range(5):
    con_wifi(net_switch)
    time.sleep(1)

"conncet to the blynk server"
# Initialize Blynk
blynk = Blynk(BLYNK_AUTH)

# call Back functions for Blynk : -> _____________________________________-----------

# Register virtual pin handler
@blynk.on("V0") #virtual pin V0 (Relay)
def v0_write_handler(value): #read the value
    if int(value[0]) == 1:
        relay.value(1) #turn the led on
    else:
        relay.value(0)    #turn the led off

# @blynk.on("V*")
# def blynk_handle_vpins(pin, value):
#     print("V{} value: {}".format(pin, value))

@blynk.on("V7")
def blynk_handle_energy(value):
    global energy,started
#     print("inside energy -> started: ",started)
    if(started == 0):
        energy = float(value[0])
#         print("energy recieved: ",energy)

@blynk.on("V8")
def blynk_handle_unit_purches(value):
    global Unit_purchesed,started
#     print("inside unit_purchesed -> started: ",started)
    if(started == 0):
        Unit_purchesed = float(value[0])
#         print("Unit_purchesed: ",Unit_purchesed)

@blynk.on("V9")
def blynk_handle_cost(value):
    global cost,started
#     print("inside cost -> started: ",started)
    if(started == 0):
        cost = float(value[0])
```

```
#         print("cost: ",cost)

#__________________________________----------------------______________________________

blynk.run()
time.sleep(1)

blynk.sync_virtual(7)
blynk.sync_virtual(8)
blynk.sync_virtual(9)

time.sleep(1)

led(0)
# start the interrupt for sampling the voltage & current -> ________________________________________--------
#Enters ADC_read function after every 1/SAMPLE_FREQ seconds
ADC_Timer = Timer(freq=SAMPLE_FREQ, mode=Timer.PERIODIC, callback=ADC_read)

# print("energy: ",energy,"Unit_purchesed:",Unit_purchesed,"cost:",cost)

# main loop (Main Loop) -> ________________________________________________________________________-------
started = 0
flag = 0
# print("started: ",started)

t0 = time.ticks_us()
while True: #run 300 cycles

    blynk.run()

    started = 1
#     print("started: ",started)

    time.sleep_ms(1000)
    reconnect(net_switch)

    if(i >= SAMPLE):
        pre_voltage1 = pre_voltage1[SAMPLE:] # stores first SAMPLE number of elements of pre_voltage1 array
        pre_voltage1.extend(voltage1) #
        pre_current1 = pre_current1[SAMPLE:]
        pre_current1.extend(current1)

        for k in range(SAMPLE):
            pre_voltage1[k] = pre_voltage1[k] - ZERO_VOLT1 #subtracts offset from voltage
            final_voltage1[k] = pre_voltage1[k]*voltage_CF1
            pre_current1[k] = pre_current1[k] - ZERO_CURT1 #subtracts offset from current
            final_current1[k] = pre_current1[k]*current_CF1

        app_power,act_power,power_factor,volt_rms,current_rms = all_cal(final_voltage1,final_current1)

        i = 0

    time_taken = time.ticks_diff(time.ticks_us(),t0)
#     print(time_taken)
    delta_energy =      act_power*(time_taken*1e-6)
#     print(delta_energy)
    t0 = time.ticks_us()
    energy = energy + (delta_energy*ENERGY_COFF)
    cost = find_cost(energy)
    Unit_remain = Unit_purchesed - energy

    print("conn Flag:",connection_flag)
    if(connection_flag == 1):
        blynk.virtual_write(1, volt_rms)
        blynk.virtual_write(2, current_rms)
```

```
        blynk.virtual_write(3, act_power)
        blynk.virtual_write(5, app_power)
        blynk.virtual_write(6, power_factor)
        blynk.virtual_write(7, energy)
        blynk.virtual_write(4, Unit_remain)
        blynk.virtual_write(9, cost)

    if(volt_rms <= 185 or volt_rms >= 245):
        relay(0)
        flag = 1
    else:
        if(flag == 1):
            relay(1)
            flag = 0


    print("rms voltage: ",volt_rms,"V")
    print("rms current: ",current_rms,"A")
    print("Active Power: ",act_power," W")
    print("Apperant Power: ",app_power," VA")
    print("Power factor: ",power_factor)
    print("Energy: ",energy," kwh")
    print("cost: ",cost," RS")


    print("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA")    # end of one cycle
    #cycles = cycles + 1
    #print(cycles)

#Turn off the timers so the program terminates cleanly
ADC_Timer.deinit()
```