Raspberry Pi software

2.0

Generated by Doxygen 1.8.13

Contents

1 Raspberry Pi code 1

2 Namespace Index 3

2.1 Packages . . . 3

3 Class Index 5

3.1 Class List . . . 5

4 File Index 7

4.1 File List . . . 7

5 Namespace Documentation 9

5.1 main3 Namespace Reference . . . 9

5.1.1 Function Documentation . . . 10

5.1.1.1 on_connect() . . . 10

5.1.1.2 on_disconnect() . . . 11

5.1.2 Variable Documentation . . . 11

5.1.2.1 bad_connection_flag . . . 11

5.1.2.2 broker . . . 11

5.1.2.3 client . . . 11

5.1.2.4 con . . . 12

5.1.2.5 con_local . . . 12

5.1.2.6 con_user . . . 12

5.1.2.7 con_user_local . . . 12

5.1.2.8 connected_flag . . . 12

5.1.2.9 control_pin 12
5.1.2.10 cur 13
5.1.2.11 cur_local 13
5.1.2.12 cur_user 13
5.1.2.13 cur_user_local 13
5.1.2.14 current_time 13
5.1.2.15 data 13
5.1.2.16 dataRef1 14
5.1.2.17 dataSend 14
5.1.2.18 err_cnt 14
5.1.2.19 Message 14
5.1.2.20 meter1 14
5.1.2.21 meter2 15
5.1.2.22 meter3 15
5.1.2.23 meter4 15
5.1.2.24 meter5 15
5.1.2.25 on_connect 15
5.1.2.26 on_disconnect 15
5.1.2.27 path 16
5.1.2.28 path_local 16
5.1.2.29 path_local_user 16
5.1.2.30 path_user 16
5.1.2.31 reading 16
5.1.2.32 readingCT1 16
5.1.2.33 readingPT1 17
5.1.2.34 readingPT2 17
5.1.2.35 setPoint 17
5.1.2.36 time_send 17
5.2 MIC3 Namespace Reference 17
5.2.1 Variable Documentation 18

5.2.1.1 CRC_error 18
5.2.1.2 Data_error 18
5.2.1.3 No_error 18
5.2.1.4 ser 18
5.2.1.5 Trans_error 18
5.3 SQLfunction Namespace Reference 19
5.3.1 Function Documentation 20
5.3.1.1 new_photonMeasure_callback() 20
5.3.1.2 old_photonMeasure_callback() 21
5.3.1.3 on_connect() 22
5.3.1.4 on_disconnect() 22
5.3.1.5 send_admin() 23
5.3.1.6 send_email() 24
5.3.1.7 update_callback() 24
5.3.2 Variable Documentation 26
5.3.2.1 bad_connection_flag 26
5.3.2.2 broker 26
5.3.2.3 client 26
5.3.2.4 con 26
5.3.2.5 con_local 27
5.3.2.6 connected_flag 27
5.3.2.7 cur 27
5.3.2.8 cur_local 27
5.3.2.9 current_time 27
5.3.2.10 dataRef1 27
5.3.2.11 DISCONNECT_TIME 28
5.3.2.12 email_cntr 28
5.3.2.13 email_context 28
5.3.2.14 email_message 28
5.3.2.15 err_cnt 28
5.3.2.16 on_connect 29
5.3.2.17 on_disconnect 29
5.3.2.18 path 29
5.3.2.19 path_local 29
5.3.2.20 receiver_email 29
5.3.2.21 sender_email 30
5.3.2.22 sender_password 30
5.3.2.23 smtp_server 30
5.3.2.24 SSLport 30

6 Class Documentation 31
6.1 MIC3.MIC1 Class Reference 31
6.1.1 Detailed Description 32
6.1.2 Constructor & Destructor Documentation 32
6.1.2.1 __init__() 32
6.1.3 Member Function Documentation 33
6.1.3.1 readApparentPower() 33
6.1.3.2 readCT1() 34
6.1.3.3 readFrequency() 35
6.1.3.4 readPhaseCurrent() 37
6.1.3.5 readPhasePower() 38
6.1.3.6 readPhaseVoltage() 39
6.1.3.7 readPowerFactor() 40
6.1.3.8 readPT1() 41
6.1.3.9 readPT2() 43
6.1.3.10 readReactivePower() 44
6.1.4 Member Data Documentation 45
6.1.4.1 __Address 45
6.1.4.2 __Control 45
6.1.4.3 __CT1 46
6.1.4.4 __F 46
6.1.4.5 __I1 46
6.1.4.6 __I2 46
6.1.4.7 __I3 46
6.1.4.8 __P1 47
6.1.4.9 __P2 47
6.1.4.10 __P3 47
6.1.4.11 __PF1 47
6.1.4.12 __PF2 47
6.1.4.13 __PF3 48

6.1.4.14 __PT1 48
6.1.4.15 __PT2 48
6.1.4.16 __Q1 48
6.1.4.17 __Q2 48
6.1.4.18 __Q3 49
6.1.4.19 __S1 49
6.1.4.20 __S2 49
6.1.4.21 __S3 49
6.1.4.22 __V1 49
6.1.4.23 __V2 50
6.1.4.24 __V3 50
6.2 MIC3.MIC2 Class Reference 50
6.2.1 Detailed Description 51
6.2.2 Constructor & Destructor Documentation 51
6.2.2.1 __init__() 51
6.2.3 Member Function Documentation 51
6.2.3.1 readFrequency() 51
6.2.3.2 readPhaseCurrent() 52
6.2.3.3 readPhasePower() 53
6.2.3.4 readPhaseVoltage() 54
6.2.4 Member Data Documentation 55
6.2.4.1 __Address 55
6.2.4.2 __Control 56
6.2.4.3 __F 56
6.2.4.4 __I1 56
6.2.4.5 __I2 56
6.2.4.6 __I3 57
6.2.4.7 __P1 57
6.2.4.8 __P2 57
6.2.4.9 __P3 57
6.2.4.10 __V1 57
6.2.4.11 __V2 58
6.2.4.12 __V3 58

7 File Documentation 59

7.1 Modbus/main3.py File Reference 59
7.2 Modbus/MIC3.py File Reference 60
7.3 README.md File Reference 61
7.4 userID/SQLfunction.py File Reference 61

Index 63

Chapter 1

Raspberry Pi code

The Raspberry Pi subsystem software: The folder Modbus contains Modbus network-related files, while the userID contains the MQTT network and users database-related files. The programs are written in Python 3.7.

Welcome to the project!

/Modbus folder:

This is the source folder that contains the firmware files for the Modbus part of the Pi. The 'main3.py' is the main script run at boot. 'MIC3.py' is the updated Python 3.7 library that contains the MIC1 and MIC2 energy meter classes and member functions needed by 'main3.py'.

/userID folder:

This is the source folder that contains the firmware files for the MQTT and user DB part of the Pi. The 'SQ↩ Lfunction.py' is the main script run at boot.

Executing the scripts

These programs are to be run from bash with 'python3 <filename>'. They are normally run at boot concurrently from their respective .sh scripts.

Chapter 2

Namespace Index

2.1 Packages

Here are the packages with brief descriptions (if available):

main3 9
MIC3 17
SQLfunction 19

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

MIC3.MIC1

Class for reading Modbus data from MIC1 energy meter 31

MIC3.MIC2

Unused class for MIC2 energy meter; it is missing the PT!, PT2, CT1 control variables 50

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Modbus/main3.py 59
Modbus/MIC3.py 60
userID/SQLfunction.py 61

Chapter 5

Namespace Documentation

5.1 main3 Namespace Reference

Functions

- def on_connect (client, userdata, flags, rc)

  Executes on MQTT client connect to broker and sets flags.
- def on_disconnect (client, userdata, rc)

  Executes on MQTT client disconnect and sets flags.

Variables

- string broker = "broker.hivemq.com"

  MQTT broker.
- string path_local = "/media/DATABASE/modbusData.db"

  Path for modbus database.
- string path = "/mnt/dav/Data/modbusData.db"
- string path_local_user = "/media/DATABASE/usertable.sqlite3"

  Path for users database.
- string path_user = "/mnt/dav/Data/usertable.sqlite3"
- con_user_local = lite.connect(path_local_user)
- cur_user_local = con_user_local.cursor()
- con_user = lite.connect(path_user)

  Initial DB connection check.
- cur_user = con_user.cursor()
- con_local = lite.connect(path_local)
- cur_local = con_local.cursor()
- con = lite.connect(path)
- cur = con.cursor()
- dataRef1 = cur.fetchone()
- int err_cnt = 0
- client = mqtt.Client()
- connected_flag
- bad_connection_flag
- on_connect
- on_disconnect

- int control_pin = 18
- meter1 = MIC.MIC1(0x01, control_pin)

  Initializes meter.
- meter2 = MIC.MIC1(0x02, control_pin)
- meter3 = MIC.MIC1(0x03, control_pin)
- meter4 = MIC.MIC1(0x04, control_pin)
- meter5 = MIC.MIC1(0x05, control_pin)
- int time_send = 1
- current_time = time.ctime(time.time())
- readingPT1 = meter1.readPT1()
- readingPT2 = meter1.readPT2()
- readingCT1 = meter1.readCT1()
- reading = meter1.readPhaseVoltage()
- string Message
- data = cur.fetchone()
- tuple setPoint = (data[0]+data[1]+data[2])
- dictionary dataSend

5.1.1 Function Documentation

5.1.1.1 on_connect()

```
def main3.on_connect (
              client,
              userdata,
              flags,
              rc )
```

Executes on MQTT client connect to broker and sets flags.

Definition at line 58 of file main3.py.

References on_connect.

```
58 def on_connect(client, userdata, flags, rc):
59     if rc == 0:
60         #print("Connection start")

61         client.bad_connection_flag = False
62         client.connected_flag = True
63         err_cnt = 0
64 
65         client.publish("HANevse/testmodbus", "Hello from Modbus function", 1, False)
66         print("Connected OK")
67 
68     else:
69         print("Bad connection, RC = ", rc)
70         client.bad_connection_flag = True
71 
```

5.1.1.2 on_disconnect()

```
def main3.on_disconnect (
                client,
                userdata,
                rc )
```

Executes on MQTT client disconnect and sets flags.

Definition at line 73 of file main3.py.

References on_disconnect.

```
def on_disconnect(client, userdata, rc):
    client.connected_flag = False
    if rc != 0:
        print("Unexpected disconnection.")
        client.bad_connection_flag = True
    else:
        print("Normal disconnection.")

```

5.1.2 Variable Documentation

5.1.2.1 bad_connection_flag

```
main3.bad_connection_flag
```

Definition at line 84 of file main3.py.

5.1.2.2 broker

```
string main3.broker = "broker.hivemq.com"
```

MQTT broker.

Definition at line 20 of file main3.py.

5.1.2.3 client

```
main3.client = mqtt.Client()
```

Definition at line 81 of file main3.py.

5.1.2.4 con

```
main3.con = lite.connect(path)
```

Definition at line 46 of file main3.py.

5.1.2.5 con_local

```
main3.con_local = lite.connect(path_local)
```

Definition at line 42 of file main3.py.

5.1.2.6 con_user

```
main3.con_user = lite.connect(path_user)
```

Initial DB connection check.

Definition at line 37 of file main3.py.

5.1.2.7 con_user_local

```
main3.con_user_local = lite.connect(path_local_user)
```

Definition at line 32 of file main3.py.

5.1.2.8 connected_flag

```
main3.connected_flag
```

Definition at line 83 of file main3.py.

5.1.2.9 control_pin

```
int main3.control_pin = 18
```

Definition at line 90 of file main3.py.

5.1.2.10 cur

```
main3.cur = con.cursor()
```

Definition at line 47 of file main3.py.

5.1.2.11 cur_local

```
main3.cur_local = con_local.cursor()
```

Definition at line 43 of file main3.py.

5.1.2.12 cur_user

```
main3.cur_user = con_user.cursor()
```

Definition at line 38 of file main3.py.

5.1.2.13 cur_user_local

```
main3.cur_user_local = con_user_local.cursor()
```

Definition at line 33 of file main3.py.

5.1.2.14 current_time

```
main3.current_time = time.ctime(time.time())
```

Definition at line 131 of file main3.py.

5.1.2.15 data

```
main3.data = cur.fetchone()
```

Definition at line 678 of file main3.py.

5.1.2.16 dataRef1

```
main3.dataRef1 = cur.fetchone()
```

Definition at line 49 of file main3.py.

5.1.2.17 dataSend

```
main3.dataSend
```

Initial value:

```
=  {
            "setPoint":setPoint,
            }
```

Definition at line 702 of file main3.py.

5.1.2.18 err_cnt

```
int main3.err_cnt = 0
```

Definition at line 55 of file main3.py.

5.1.2.19 Message

```
string main3.Message
```

Initial value:

```
=  current_time + """
        V1: %.2f   V2: %.2f   V3: %.2f
        """%(meter1._MIC1__V1, meter1._MIC1__V2, meter1._MIC1__V3)
```

Definition at line 153 of file main3.py.

5.1.2.20 meter1

```
main3.meter1 = MIC.MIC1(0x01, control_pin)
```

Initializes meter.

Definition at line 115 of file main3.py.

5.1.2.21 meter2

```
main3.meter2 = MIC.MIC1(0x02, control_pin)
```

Definition at line 116 of file main3.py.

5.1.2.22 meter3

```
main3.meter3 = MIC.MIC1(0x03, control_pin)
```

Definition at line 117 of file main3.py.

5.1.2.23 meter4

```
main3.meter4 = MIC.MIC1(0x04, control_pin)
```

Definition at line 118 of file main3.py.

5.1.2.24 meter5

```
main3.meter5 = MIC.MIC1(0x05, control_pin)
```

Definition at line 119 of file main3.py.

5.1.2.25 on_connect

```
main3.on_connect
```

Definition at line 85 of file main3.py.

Referenced by on_connect().

5.1.2.26 on_disconnect

```
main3.on_disconnect
```

Definition at line 86 of file main3.py.

Referenced by on_disconnect().

5.1.2.27 path

```
string main3.path = "/mnt/dav/Data/modbusData.db"
```

Definition at line 26 of file main3.py.

5.1.2.28 path_local

```
string main3.path_local = "/media/DATABASE/modbusData.db"
```

Path for modbus database.

Definition at line 25 of file main3.py.

5.1.2.29 path_local_user

```
string main3.path_local_user = "/media/DATABASE/usertable.sqlite3"
```

Path for users database.

Definition at line 29 of file main3.py.

5.1.2.30 path_user

```
string main3.path_user = "/mnt/dav/Data/usertable.sqlite3"
```

Definition at line 30 of file main3.py.

5.1.2.31 reading

```
main3.reading = meter1.readPhaseVoltage()
```

Definition at line 150 of file main3.py.

5.1.2.32 readingCT1

```
main3.readingCT1 = meter1.readCT1()
```

Definition at line 146 of file main3.py.

5.1.2.33 readingPT1

```
main3.readingPT1 = meter1.readPT1()
```

Definition at line 144 of file main3.py.

5.1.2.34 readingPT2

```
main3.readingPT2 = meter1.readPT2()
```

Definition at line 145 of file main3.py.

5.1.2.35 setPoint

```
int main3.setPoint = (data[0]+data[1]+data[2])
```

Definition at line 679 of file main3.py.

5.1.2.36 time_send

```
int main3.time_send = 1
```

Definition at line 122 of file main3.py.

5.2 MIC3 Namespace Reference

Classes

- class MIC1

  Class for reading Modbus data from MIC1 energy meter.
- class MIC2

  Unused class for MIC2 energy meter; it is missing the PT!, PT2, CT1 control variables.

Variables

- ser = serial.Serial("/dev/ttyS0", 38400)

  This library is made for reading MIC/MIC2 energy meters with a MAX485 module MIC2 only reads data from registers.
- int Data_error = -3
- int CRC_error = -2
- int Trans_error = -1
- int No_error = 0

5.2.1 Variable Documentation

5.2.1.1 CRC_error

```
int MIC3.CRC_error = -2
```

Definition at line 20 of file MIC3.py.

5.2.1.2 Data_error

```
int MIC3.Data_error = -3
```

Definition at line 19 of file MIC3.py.

5.2.1.3 No_error

```
int MIC3.No_error = 0
```

Definition at line 22 of file MIC3.py.

5.2.1.4 ser

```
MIC3.ser = serial.Serial("/dev/ttyS0", 38400)
```

This library is made for reading MIC/MIC2 energy meters with a MAX485 module MIC2 only reads data from registers.

This is not the correct value. To calculate correct value, PT1, PT2, CT1 need to be read. Please take MIC1 as an example MIC1: 8-bit data, no parity, 1 stop bit, 38400 BAUD

Definition at line 16 of file MIC3.py.

5.2.1.5 Trans_error

```
int MIC3.Trans_error = -1
```

Definition at line 21 of file MIC3.py.

5.3 SQLfunction Namespace Reference

Functions

- def on_connect (client, userdata, flags, rc)

  Executes on MQTT client connect to broker, sets flags and subscribes.
- def on_disconnect (client, userdata, rc)

  Executes on MQTT client disconnect and sets flags.
- def update_callback (client, userdata, message)

  Callback function that parses RFID swipe message from Photon and checks in the DB what to publish as answer Publish output is structured as "1;2" where 1=socket number and 2=answer to Photon.
- def new_photonMeasure_callback (client, userdata, message)

  New Callback for Photon measurements that parses, checks DB for user data like name and carname, then logs into 'measurements'.
- def old_photonMeasure_callback (client, userdata, message)

  Old Photon measurements callback that parses '' separated values.
- def send_admin ()

  Function to send admin email if one user has been plugged in at a socket for over 4 hours still in beta mode and needs improvements.
- def send_email ()

Variables

- DISCONNECT_TIME = int(4 $*$ 60 $*$ 60)

  Const.
- int email_cntr = 0
- int SSLport = 465
- string smtp_server = "smtp.gmail.com"
- string sender_email = "tpi97364@gmail.com"

  Email sender address for Pi.
- string receiver_email = ""

  Holder for email addresses of receivers.
- string sender_password = "controlsystem"

  Pi email password.
- string email_message

  Default email message.
- email_context = ssl.create_default_context()
- con = None

  Initial DB connection check.
- string broker = "broker.hivemq.com"

  MQTT broker address.
- string path_local = "/media/DATABASE/usertable.sqlite3"

  Path to users database path = "./userList" #Use internal memory - old DB.
- string path = "/mnt/dav/Data/usertable.sqlite3"
- con_local = lite.connect(path_local)
- cur_local = con_local.cursor()
- cur = con.cursor()
- dataRef1 = cur.fetchone()
- int err_cnt = 0
- client = mqtt.Client()
- connected_flag
- bad_connection_flag
- on_connect
- on_disconnect
- current_time = time.ctime(time.time())

5.3.1 Function Documentation

5.3.1.1 new_photonMeasure_callback()

```
def SQLfunction.new_photonMeasure_callback (
                client,
                userdata,
                message )
```

New Callback for Photon measurements that parses, checks DB for user data like name and carname, then logs into 'measurements'.

Definition at line 208 of file SQLfunction.py.

```
208 def new_photonMeasure_callback(client, userdata, message):
209     try:
210         con = lite.connect(path)
211         cur = con.cursor()
212     except Exception as e:
213         print (e)
214     con_local = lite.connect(path_local)
215     cur_local = con_local.cursor()
216     data = json.loads(message.payload)
217     print(data)
218     V1 = float(data.get("V1"))
219     V2 = float(data.get("V2"))
220     V3 = float(data.get("V3"))
221     I1 = float(data.get("I1"))
222     I2 = float(data.get("I2"))
223     I3 = float(data.get("I3"))
224     #P = float(data.get("P"))
225     #E = float(data.get("E"))
226     F = float(data.get("F"))
227     Time = int(data.get("Time"))
228     SocketID = int(data.get("SocketID"))
229     UserID = str(data.get("UserID")).upper()
230 
231     try:
232         cur.execute("SELECT name, rowid FROM users WHERE uidTag = ? ", (UserID,) )
233         dataUser = cur.fetchone()
234     except Exception as e:
235         print (e)
236         cur_local.execute("SELECT name, rowid FROM users WHERE uidTag = ? ", (UserID,) )
237         dataUser = cur_local.fetchone()
238 
239     try:
240         dataUser[1]
241     except Exception as e:
242         print (e)
243         print("WARNING: Unauthorized user " + UserID + " charging at socket " + str(SocketID))
244         dataUser = (’unknown’, 31)
245     try:
246         cur.execute("SELECT carId FROM car_of_user WHERE userId = ? ", (dataUser[1],) )
247         if (cur.fetchone() is None):
248             carId = 404
249         else:
250             carId = cur.fetchone()[0]
251     except Exception as e:
252         print (e)
253         cur_local.execute("SELECT carId FROM car_of_user WHERE userId = ? ", (dataUser[1],) )
254         if (cur.fetchone() is None):
255             carId = 404
256         else:
257             carId = cur_local.fetchone()[0]
258 
259     try:
260         cur.execute("SELECT brand || ’ ’ || type FROM cars WHERE id = ? ", (carId,) )
261         carName = cur.fetchone()[0]
262     except Exception as e:
263         print (e)
264         cur_local.execute("SELECT brand || ’ ’ || type FROM cars WHERE id = ? ", (carId,) )
265         carName = cur_local.fetchone()[0]
266     try:
```

```
267         cur.execute("INSERT INTO measurements(userId, userName, carId, carName, socketId, V1, V2, V3, I1, I2, I3, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?)",
268                     (UserID, dataUser[0], carId, carName, SocketID, V1, V2, V3, I1, I2, I3, F, Time))
269         con.commit()
270     except Exception as e:
271         print (e)
272     finally:
273         cur_local.execute("INSERT INTO measurements(userId, userName, carId, carName, socketId, V1, V2, V3, I1, I2, I3, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?)",
274                 (UserID, dataUser[0], carId, carName, SocketID, V1, V2, V3, I1, I2, I3, F, Time))
275 
276     con_local.commit()
277     #-#Insert with P and E measurements
278     #cur.execute("INSERT INTO measurements(userId, userName, carId, carName, socketId, V1, V2, V3, I1, I2, I3, P, E, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)",
279     #            (UserID, dataUser[0], carId, carName, SocketID, V1, V2, V3, I1, I2, I3, P, E, F, Time))
280 
281     # Or skip all .get() and do it in cur.execute
282     #cur.execute("INSERT INTO measurements(userId, socketId, V1, V2, V3, I1, I2, I3, P, E, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?)",
283     #            (str(data["UserID"]).upper(), int(data["SocketID"]), float(data["V1"]), float(data["V2"]), float(data["V3"]), float(data["I1"]), float(data["I2"]), float(data["I3"]), float(data["P"]), float(data["E"]), float(data["F"]), int(data["Time"]) ))
284 
285    #for readable timestamp use this at end of INSERT: time.strftime('%Y-%m-%d %T', time.localtime(int(data["Time"]) ))
286 
287 
288 
```

5.3.1.2 old_photonMeasure_callback()

```
def SQLfunction.old_photonMeasure_callback (
                client,
                userdata,
                message )
```

Old Photon measurements callback that parses '' separated values.

Definition at line 290 of file SQLfunction.py.

```
290 def old_photonMeasure_callback(client, userdata, message):
291     con = lite.connect(path)
292     cur = con.cursor()
293     data = message.payload
294     data = data.decode('UTF-8')
295     print(data)
296 
297     index = []
298     for i in range(len(data)):
299         if (data[i] == '%'):
300             index.append(i)
301     V1 = float(data[:index[0]])
302     V2 = float(data[index[0]+1:index[1]])
303     V3 = float(data[index[1]+1:index[2]])
304     I1 = float(data[index[2]+1:index[3]])
305     I2 = float(data[index[3]+1:index[4]])
306     I3 = float(data[index[4]+1:index[5]])
307     P = float(data[index[5]+1:index[6]])
308     E = float(data[index[6]+1:index[7]])
309     F = float(data[index[7]+1:index[8]])
310     Time = int(data[index[8]+1:index[9]])
311     SocketID = int(data[index[9]+1:index[10]])
312     UserID = data[index[10]+1:index[11]]
313     cur.execute("INSERT INTO photonMeasure(UIDtag, SocketID, V1, V2, V3, I1, I2, I3, P, E, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?)",
314                 (UserID, SocketID, V1, V2, V3, I1, I2, I3, P, E, F, Time))
315     con.commit()
316     #print(V1)
317 
```

5.3.1.3 on_connect()

```
def SQLfunction.on_connect (
                client,
                userdata,
                flags,
                rc )
```

Executes on MQTT client connect to broker, sets flags and subscribes.

Definition at line 63 of file SQLfunction.py.

References on_connect.

```
63 def on_connect(client, userdata, flags, rc):
64     if rc == 0:
65         #print("Connection start")
66         print(path)
67         client.bad_connection_flag = False
68         client.connected_flag = True
69         err_cnt = 0
70 
71         client.publish("HANevse/testsql", "Hello from SQLfunction",1 ,False)
72 #         client.subscribe([("HANevse/getUsers", 2), ("HANevse/UpdateUser", 2), ("HANevse/photonMeasure",
       2)])
73         client.subscribe([("HANevse/updateUser", 2), ("HANevse/photonMeasure", 2)])
74         print("Connected OK")
75     else:
76         print("Bad connection, RC = ", rc)
77         client.bad_connection_flag = True
78 
```

5.3.1.4 on_disconnect()

```
def SQLfunction.on_disconnect (
                client,
                userdata,
                rc )
```

Executes on MQTT client disconnect and sets flags.

Definition at line 80 of file SQLfunction.py.

References on_disconnect.

```
80 def on_disconnect(client, userdata, rc):
81     client.connected_flag = False
82     if rc != 0:
83         print("Unexpected disconnection.")
84         client.bad_connection_flag = True
85     else:
86         print("Normal disconnection.")
87 
88 # def SendUser_callback(client, userdata, message):
89 #     #print(message.payload)
90 #     con = lite.connect(path)
91 #     cur = con.cursor()
92 #     cur.execute('select * from list')
93 #
94 #     data = cur.fetchall()
95 #     dataSend = ""
96 #
97 #     for element in data:
98 #         print(element)
99 #         dataSend +=
     (str(element[0])+'%'+element[1]+'%'+element[2]+'%'+str(element[3])+'%'+element[4]+'%'+str(element[5])+'%'+element[6]+'
100 #
101 #     client.publish("HANevse/UserList", dataSend, 2, True)
102 #     #publish.single("HANevse/UserList", dataSend, hostname=broker)
103 #     #print(dataSend)
104 
```

5.3.1.5 send_admin()

```
def SQLfunction.send_admin ( )
```

Function to send admin email if one user has been plugged in at a socket for over 4 hours still in beta mode and needs improvements.

Definition at line 320 of file SQLfunction.py.

```
320 def send_admin():
321     try:
322         con = lite.connect(path)
323         cur = con.cursor()
324         cur.execute("SELECT rowid FROM measurements WHERE Time <= ? AND Time >= ? LIMIT 1", ((str(int(
  time.time()) - DISCONNECT_TIME)),(str(int(time.time()) - DISCONNECT_TIME - 29)),) )
325         dataRef = cur.fetchone()
326         if dataRef is None:
327             pass
328         else:
329             cur.execute("SELECT rowid FROM measurements WHERE Time >= ? LIMIT 1", ((str(int(time.time()) -
  60)),) )
330             dataRef = cur.fetchone()
331             if dataRef is None:
332                 pass
333             else:
334                 with smtplib.SMTP_SSL(smtp_server, SSLport, context=email_context) as server:
335                     server.login(sender_email, sender_password)
336                     server.sendmail(sender_email, "nguyenxuan.trung@han.nl", email_message)
337     except Exception as e:
338         print (e)
339         con_local = lite.connect(path_local)
340         cur_local = con_local.cursor()
341         cur_local.execute("SELECT rowid FROM measurements WHERE Time <= ? AND Time >= ? LIMIT 1", ((str(int
  (time.time()) - DISCONNECT_TIME)),(str(int(time.time()) - DISCONNECT_TIME - 29)),) )
342         dataRef = cur_local.fetchone()
343         if dataRef is None:
344             pass
345         else:
346             cur_local.execute("SELECT rowid FROM measurements WHERE Time >= ? LIMIT 1", ((str(int(time.time
  ()) - 60)),) )
347             dataRef = cur_local.fetchone()
348             if dataRef is None:
349                 pass
350             else:
351                 with smtplib.SMTP_SSL(smtp_server, SSLport, context=email_context) as server:
352                     server.login(sender_email, sender_password)
353                     server.sendmail(sender_email, "nguyenxuan.trung@han.nl", email_message)
354 
355 #     if dataRef is None:
356 #         pass
357 #     else:
358 #         try:
359 #             cur.execute("SELECT rowid FROM measurements WHERE Time >= ? LIMIT 1", ((str(int(time.time())
   - 60)),) )
360 #             dataRef = cur.fetchone()
361 #         except Exception as e:
362 #             print (e)
363 #             cur_local.execute("SELECT rowid FROM measurements WHERE Time >= ? LIMIT 1",
   ((str(int(time.time()) - 60)),) )
364 #             dataRef = cur_local.fetchone()
365 #
366 #         if dataRef is None:
367 #             pass
368 #         else:
369 #             with smtplib.SMTP_SSL(smtp_server, SSLport, context=email_context) as server:
370 #                 server.login(sender_email, sender_password)
371 #                 server.sendmail(sender_email, "nguyenxuan.trung@han.nl", email_message)
372 
373 # Function (executed every 5min) that checks in DB for users charging for over 4 hours and sends them
   email to d/c
```

5.3.1.6 send_email()

```
def SQLfunction.send_email ( )
```

Definition at line 374 of file SQLfunction.py.

```
374 def send_email():
375 
376     con_local = lite.connect(path_local)
377     cur_local = con_local.cursor()
378     try:
379         con = lite.connect(path)
380         cur = con.cursor()
381         cur.execute("SELECT email, rowid FROM users WHERE LastStartOrStop <= ? AND email <> ’’ AND mailed <
       1 AND socketId IS NOT NULL", ((str(int(time.time()) - DISCONNECT_TIME)),) )
382         dataRef = cur.fetchall()
383     except Exception as e:
384         print (e)
385         cur_local.execute("SELECT email, rowid FROM users WHERE LastStartOrStop <= ? AND email <> ’’ AND
       mailed < 1 AND socketId IS NOT NULL", ((str(int(time.time()) - DISCONNECT_TIME)),) )
386         dataRef = cur_local.fetchall()
387 
388     if dataRef is None:
389         return
390     url = "http://www.kite.com"
391     timeout = 5
392     try:
393         request = requests.get(url, timeout=timeout)
394         print("Connected to the Internet")
395     except (requests.ConnectionError, requests.Timeout) as exception:
396         return
397     with smtplib.SMTP_SSL(smtp_server, SSLport, context=email_context) as server:
398         server.login(sender_email, sender_password)
399         for element in dataRef:
400             server.sendmail(sender_email, element[0], email_message)
401             try:
402                 cur.execute("UPDATE users SET mailed = 1 WHERE rowid=?", (element[1],))
403             except Exception as e:
404                 print (e)
405             cur_local.execute("UPDATE users SET mailed = 1 WHERE rowid=?", (element[1],))
406     #cur.execute("UPDATE users SET mailed = 0 WHERE LastStartOrStop > ? AND mailed > 0 AND socketId IS
       NULL", ((str(int(time.time()) - DISCONNECT_TIME)),) )
407     try:
408         cur.execute("UPDATE users SET mailed = 0 WHERE mailed > 0 AND socketId IS NULL")
409         con.commit()
410     except Exception as e:
411         print (e)
412     cur_local.execute("UPDATE users SET mailed = 0 WHERE mailed > 0 AND socketId IS NULL")
413     con_local.commit()
414 
415 #setup mqtt
```

5.3.1.7 update_callback()

```
def SQLfunction.update_callback (
                client,
                userdata,
                message )
```

Callback function that parses RFID swipe message from Photon and checks in the DB what to publish as answer Publish output is structured as "1;2" where 1=socket number and 2=answer to Photon.

Definition at line 107 of file SQLfunction.py.

```
def update_callback(client, userdata, message):
    try:
        con = lite.connect(path)
        cur = con.cursor()
    except Exception as e:
        print (e)
    con_local = lite.connect(path_local)
    cur_local = con_local.cursor()
    data = json.loads(message.payload)
    print(data)

    UserId = str(data.get("UserId")).upper()
    socketId = int(data.get("Charger"))
    StartTime = int(data.get("StartTime"))
    #print(UserId)
    try:
        cur.execute("SELECT LastStartOrStop, socketId, verified FROM users WHERE uidTag=? LIMIT 1", (UserId,))
        dataUser = cur.fetchone() # returns a tuple
    except Exception as e:
        print (e)
        cur_local.execute("SELECT LastStartOrStop, socketId, verified FROM users WHERE uidTag=? LIMIT 1", (UserId,))
        dataUser = cur_local.fetchone() # returns a tuple

    dataSend = str(socketId) + ";"
    try:
        cur.execute("SELECT socketId FROM users WHERE socketId=? LIMIT 1", (socketId,))
        socketUsed = cur.fetchone()
    except Exception as e:
        print (e)
        cur_local.execute("SELECT socketId FROM users WHERE socketId=? LIMIT 1", (socketId,))
        socketUsed = cur_local.fetchone()
    #The socketUsed can be either None or the socket number, so parsing it can give error without check
    if socketUsed is not None:
        socketUsed = socketUsed[0]

    #This is the filter for checking and preparing the answer to the EV charger
    if dataUser is not None: # if user ID is in list
        if (dataUser[2] == "true"):
            if ((StartTime - dataUser[0]) >= 20): # if last swipe is over 20s ago
                if (socketUsed == socketId): #if this socket is used now
                    if (socketId == dataUser[1]): # if user already uses this socket
                        dataSend += "4" # successfully stop charging
                        try:
                            cur.execute("UPDATE users SET socketId=?, LastStartOrStop=? WHERE uidTag=?", (None, StartTime, UserId))
                        except Exception as e:
                            print (e)
                        finally:
                            cur_local.execute("UPDATE users SET socketId=?, LastStartOrStop=? WHERE uidTag=?", (None, StartTime, UserId))
                    else:
                        dataSend += "3" # socket is occupied by another user
                else: #if this socket is free
                    if dataUser[1] is None: #if user was not using any socket
                        dataSend += "1" # successfully start new charge
                        try:
                            cur.execute("UPDATE users SET socketId=?, LastStartOrStop=? WHERE uidTag=?", (socketId, StartTime, UserId))
                        except Exception as e:
                            print (e)
                        finally:
                            cur_local.execute("UPDATE users SET socketId=?, LastStartOrStop=? WHERE uidTag=?", (socketId, StartTime, UserId))
                    else:
                        dataSend += "6" # user already at another socket
            else: #if swiped less than 20s ago
                if (socketUsed == socketId): #if this socket is used now
                    if (socketId == dataUser[1]): # if user already uses this socket
                        dataSend += "5" # you just started using this socket less than 20s ago
                    else:
                        dataSend += "3" # socket is occupied by another user
                else: #if this socket is free
                    if dataUser[1] is None: #if user was not using any socket
                        dataSend += "2" #charger is free, but you already swiped less than 20s ago
                    else:
                        dataSend += "6"  # user already at another socket
        else:
             dataSend += "7" #user not verified by admin
    else:
        dataSend += "8" #user not in the userlist
    try:
        con.commit()
    except Exception as e:
        print (e)
    con_local.commit()

    client.publish("HANevse/allowUser", dataSend, 0, False)

# def Update_callback(client, userdata, message):
#     con = lite.connect(path)
#     cur = con.cursor()
#     data = message.payload
#     index = []
#     for i in range(len(data)):
#         if (data[i] == '%'):
#             index.append(i)
#     UserId = int(data[:index[0]])
#     PendingCharger = int(data[index[0]+1:index[1]])
#     StartTime = int(data[index[1]+1:index[2]])
#     cur.execute("UPDATE list SET PendingCharger=? WHERE Id=?", (PendingCharger, UserId))
#     cur.execute("UPDATE list SET StartTime=? WHERE Id=?", (StartTime, UserId))
#     con.commmit()


```

5.3.2 Variable Documentation

5.3.2.1 bad_connection_flag

`SQLfunction.bad_connection_flag`

Definition at line 419 of file SQLfunction.py.

5.3.2.2 broker

`string SQLfunction.broker = "broker.hivemq.com"`

MQTT broker address.

Definition at line 38 of file SQLfunction.py.

5.3.2.3 client

`SQLfunction.client = mqtt.Client()`

Definition at line 416 of file SQLfunction.py.

5.3.2.4 con

`SQLfunction.con = None`

Initial DB connection check.

Definition at line 35 of file SQLfunction.py.

5.3.2.5 con_local

```
SQLfunction.con_local = lite.connect(path_local)
```

Definition at line 47 of file SQLfunction.py.

5.3.2.6 connected_flag

```
SQLfunction.connected_flag
```

Definition at line 418 of file SQLfunction.py.

5.3.2.7 cur

```
SQLfunction.cur = con.cursor()
```

Definition at line 53 of file SQLfunction.py.

5.3.2.8 cur_local

```
SQLfunction.cur_local = con_local.cursor()
```

Definition at line 48 of file SQLfunction.py.

5.3.2.9 current_time

```
SQLfunction.current_time = time.ctime(time.time())
```

Definition at line 446 of file SQLfunction.py.

5.3.2.10 dataRef1

```
SQLfunction.dataRef1 = cur.fetchone()
```

Definition at line 55 of file SQLfunction.py.

5.3.2.11 DISCONNECT_TIME

```
SQLfunction.DISCONNECT_TIME = int(4 * 60 * 60)
```

Const.

for max time before email is sent to charging user

Definition at line 14 of file SQLfunction.py.

5.3.2.12 email_cntr

```
int SQLfunction.email_cntr = 0
```

Definition at line 16 of file SQLfunction.py.

5.3.2.13 email_context

```
SQLfunction.email_context = ssl.create_default_context()
```

Definition at line 33 of file SQLfunction.py.

5.3.2.14 email_message

```
string SQLfunction.email_message
```

Initial value:

```
=  """\
Subject: Unplug car

Please unplug your car from the EV charger. Over 4 hours have passed since it was plugged in.

This is an automatically generated email. A response to this email will not be read."""
```

Default email message.

Definition at line 26 of file SQLfunction.py.

5.3.2.15 err_cnt

```
int SQLfunction.err_cnt = 0
```

Definition at line 60 of file SQLfunction.py.

5.3.2.16 on_connect

```
SQLfunction.on_connect
```

Definition at line 420 of file SQLfunction.py.

Referenced by on_connect().

5.3.2.17 on_disconnect

```
SQLfunction.on_disconnect
```

Definition at line 421 of file SQLfunction.py.

Referenced by on_disconnect().

5.3.2.18 path

```
string SQLfunction.path = "/mnt/dav/Data/usertable.sqlite3"
```

Definition at line 45 of file SQLfunction.py.

5.3.2.19 path_local

```
string SQLfunction.path_local = "/media/DATABASE/usertable.sqlite3"
```

Path to users database path = "./userList" #Use internal memory - old DB.

Definition at line 44 of file SQLfunction.py.

5.3.2.20 receiver_email

```
string SQLfunction.receiver_email = ""
```

Holder for email addresses of receivers.

Definition at line 22 of file SQLfunction.py.

5.3.2.21 sender_email

```
string SQLfunction.sender_email = "tpi97364@gmail.com"
```

Email sender address for Pi.

Definition at line 20 of file SQLfunction.py.

5.3.2.22 sender_password

```
string SQLfunction.sender_password = "controlsystem"
```

Pi email password.

Definition at line 24 of file SQLfunction.py.

5.3.2.23 smtp_server

```
string SQLfunction.smtp_server = "smtp.gmail.com"
```

Definition at line 18 of file SQLfunction.py.

5.3.2.24 SSLport

```
int SQLfunction.SSLport = 465
```

Definition at line 17 of file SQLfunction.py.

Chapter 6

Class Documentation

6.1 MIC3.MIC1 Class Reference

Class for reading Modbus data from MIC1 energy meter.

Public Member Functions

- def __init__ (self, Id, Control)
- def readPT1 (self)

  Reads PT1 variable needed for all other calculations.
- def readPT2 (self)

  Reads PT2 variable needed for all other calculations.
- def readCT1 (self)

  Reads CT1 variable needed for all other calculations.
- def readPhaseVoltage (self)

  Reads and calculates phase Voltages with the help of PT1 and PT2.
- def readPhaseCurrent (self)

  Reads and calculates phase Currents with the help of CT1.
- def readPhasePower (self)

  Reads and calculates phase Power values with the help of PT1, PT2, CT1.
- def readReactivePower (self)

  Reads and calculates Reactive Power (Q) values with the help of PT1, PT2, CT1.
- def readApparentPower (self)

  Reads and calculates Apparent Power (S) values with the help of PT1, PT2, CT1 This function is diferent because the CRC value overflows and is to be edited to not error.
- def readPowerFactor (self)

  Reads and calculates Power Factors (PF)
- def readFrequency (self)

  Reads and calculates Frequency (F)

Private Attributes

- __Control
- __Address
- __PT1
- __PT2
- __CT1
- __V1
- __V2
- __V3
- __I1
- __I2
- __I3
- __P1
- __P2
- __P3
- __Q1
- __Q2
- __Q3
- __S1
- __S2
- __S3
- __PF1
- __PF2
- __PF3
- __F

6.1.1 Detailed Description

Class for reading Modbus data from MIC1 energy meter.

Definition at line 248 of file MIC3.py.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 __init__()

```
def MIC3.MIC1.__init__ (
              self,
              Id,
              Control )
```

Definition at line 249 of file MIC3.py.

```
249     def __init__(self, Id, Control):
250         self.__Control = Control
251         self.__Address = Id
252         self.__PT1 = 1.0
253         self.__PT2 = 1.0
254         self.__CT1 = 1.0
255         self.__V1 = 0.0
256         self.__V2 = 0.0
257         self.__V3 = 0.0
258         self.__I1 = 0.0
259         self.__I2 = 0.0
260         self.__I3 = 0.0
261         self.__P1 = 0.0
262         self.__P2 = 0.0
263         self.__P3 = 0.0
264         self.__Q1 = 0.0
265         self.__Q2 = 0.0
266         self.__Q3 = 0.0
267         self.__S1 = 0.0
268         self.__S2 = 0.0
269         self.__S3 = 0.0
270         self.__PF1 = 0.0
271         self.__PF2 = 0.0
272         self.__PF3 = 0.0
273         self.__F  = 0.0
274
```

6.1.3 Member Function Documentation

6.1.3.1 readApparentPower()

```
def MIC3.MIC1.readApparentPower (
                self )
```

Reads and calculates Apparent Power (S) values with the help of PT1, PT2, CT1 This function is diferent because the CRC value overflows and is to be edited to not error.

Definition at line 709 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.↩ MIC1.__CT1, MIC3.MIC1.__PT1, MIC3.MIC1.__PT2, MIC3.MIC1.__S1, MIC3.MIC1.__S2, and MIC3.MIC1.__S3.

```
709     def readApparentPower(self):
710         #Calculate CRC16-MODBUS

711         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
712         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x46, 0x00, 0x03])))
713         crc_Tx = crc_Tx.replace(" ", "0")
714         #The crc_Tx must include 4 hexadecimal characters.

715         #If crc_Tx = 10, function hex() will return 0xa, which is not expected

716         #Therefore, String format operator was used

717
718         #Send request

719         GPIO.output(self.__Control, GPIO.HIGH)
720         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x46, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
721
722         #There is a delay caused by the converter. The program must wait before reading the result

723         sleep(0.02)
724
725         #Receive data

726         GPIO.output(self.__Control, GPIO.LOW)
727         cnt = 0
728         data_left = ser.inWaiting()
```

```
729         while (data_left == 0):
730             #wait for data

731             cnt=cnt+1
732             if (cnt < 50000): #wait for maximum 5 seconds

733                 sleep(0.0001)
734                 data_left = ser.inWaiting()
735             else:
736                 print("Transmitting error: Time out")
737                 return Trans_error
738         received_data = ser.read()
739         sleep(0.02)
740         data_left = ser.inWaiting()
741         received_data += ser.read(data_left)
742 
743         if ((received_data[0]) != self.__Address):
744             print("Transmitting error: Data corrupted")
745             return Data_error
746         if (len(received_data) != 11):
747             print("Transmitting error: Data corrupted")
748             return Data_error
749 
750         #Check the CRC code

751         crc_cal = hex(crc16(received_data[:9]))
752         crc_Rx = hex(struct.unpack(’H’,received_data[9:])[0])
753 
754         if crc_cal == crc_Rx:
755             self.__S1 = float(struct.unpack(’H’, received_data[4:2:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
756             self.__S2 = float(struct.unpack(’H’, received_data[6:4:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
757             self.__S3 = float(struct.unpack(’H’, received_data[8:6:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
758             return No_error
759         else:
760             print("Transmitting error: Incorrect CRC")
761             return CRC_error
762 
```

6.1.3.2 readCT1()

```
def MIC3.MIC1.readCT1 (
                self )
```

Reads CT1 variable needed for all other calculations.

Definition at line 410 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, and MIC3.MIC1.__CT1.

```
410     def readCT1(self):
411         #Calculate CRC16-MODBUS

412         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
413         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x08, 0x00, 0x01])))
414         #The crc_Tx must include 4 hexadecimal characters.

415         #If crc_Tx =  10, function hex() will return 0xa, which is not expected

416         #Therefore, String format operator was used

417 
418         #Send request

419         GPIO.output(self.__Control, GPIO.HIGH)
420         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x08, 0x00, 0x01, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
421 
422         #There is a delay caused by the converter. The program must wait before reading the result
```

```
        sleep(0.01)

        #Receive data
        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data
            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds
                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error

        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        #Check if the data is correct
        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error
        if (len(received_data) != 7):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:5]))

        #DEBUG ONLY-----------------------------------------------
        #retval = ""
        #for character in received_data:
        #    retval += ('0123456789ABCDEF'[int((character)/16)])
        #    retval += ('0123456789ABCDEF'[int((character)%16)])
        #    retval += ':'
        #print (retval[:-1])
        #print (crc_cal) #use for debugging only
        #---------------------------------------------------------

        crc_Rx = hex(struct.unpack('H',received_data[5:])[0])

        #print (crc_Rx) #use for degugging only

        if crc_cal == crc_Rx:
            self.__CT1 = float(struct.unpack('H', received_data[4:2:-1])[0])
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error
```

6.1.3.3 readFrequency()

```
def MIC3.MIC1.readFrequency (
                self )
```

Reads and calculates Frequency (F)

Definition at line 818 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.↩ MIC2.__F, and MIC3.MIC1.__F.

```
    def readFrequency(self):
        #Calculate CRC16-MODBUS
        crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
        crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x30, 0x00, 0x01])))
        #The crc_Tx must include 4 hexadecimal characters.
        #If crc_Tx =  10, function hex() will return 0xa, which is not expected
        #Therefore, String format operator was used

        #Send request
        GPIO.output(self.__Control, GPIO.HIGH)
        ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x30, 0x00, 0x01, int(crc_Tx[3:],16), int(
  crc_Tx[1:3],16)]))

        #There is a delay caused by the converter. The program must wait before reading the result
        sleep(0.01)

        #Receive data
        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data
            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds
                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error
        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        #DEBUG ONLY-----------------------------------------------
        #retval = ""
        #for character in received_data:
        #    retval += ('0123456789ABCDEF'[int((character)/16)])
        #    retval += ('0123456789ABCDEF'[int((character)%16)])
        #    retval += ':'
        #print (retval[:-1])
        #print (crc_cal) #use for debugging only
        #---------------------------------------------------------

        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error
        if (len(received_data) != 7):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:5]))
        crc_Rx = hex(struct.unpack('H',received_data[5:])[0])

        if crc_cal == crc_Rx:
            self.__F = float(struct.unpack('H', received_data[4:2:-1])[0])/100
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error
#---------------------------------END OF MIC1------------------------------------
```

```
879 #--------------------------------------------------------------------------------

880
```

6.1.3.4 readPhaseCurrent()

```
def MIC3.MIC1.readPhaseCurrent (
                self )
```

Reads and calculates phase Currents with the help of CT1.

Definition at line 546 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.MIC1.__CT1, MIC3.MIC2.__I1, MIC3.MIC1.__I1, MIC3.MIC2.__I2, MIC3.MIC1.__I2, MIC3.MIC2.__I3, and MIC3.MIC1.__I3.

```
546     def readPhaseCurrent(self):
547         #Calculate CRC16-MODBUS

548         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
549         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x39, 0x00, 0x03])))
550         #The crc_Tx must include 4 hexadecimal characters.

551         #If crc_Tx =  10, function hex() will return 0xa, which is not expected

552         #Therefore, String format operator was used

553
554         #Send request

555         GPIO.output(self.__Control, GPIO.HIGH)
556         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x39, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
557
558         #There is a delay caused by the converter. The program must wait before reading the result

559         sleep(0.01)
560
561         #Receive data

562         GPIO.output(self.__Control, GPIO.LOW)
563         cnt = 0
564         data_left = ser.inWaiting()
565         while (data_left == 0):
566             #wait for data

567             cnt=cnt+1
568             if (cnt < 50000): #wait for maximum 5 seconds

569                 sleep(0.0001)
570                 data_left = ser.inWaiting()
571             else:
572                 print("Transmitting error: Time out")
573                 return Trans_error
574         received_data = ser.read()
575         sleep(0.01)
576         data_left = ser.inWaiting()
577         received_data += ser.read(data_left)
578
579         if ((received_data[0]) != self.__Address):
580             print("Transmitting error: Data corrupted")
581             return Data_error
582         if (len(received_data) != 11):
583             print("Transmitting error: Data corrupted")
584             return Data_error
585
586         #Check the CRC code

587         crc_cal = hex(crc16(received_data[:9]))
588         crc_Rx = hex(struct.unpack('H',received_data[9:])[0])
589
590         if crc_cal == crc_Rx:
```

```
591             self.__I1 = float(struct.unpack(’H’, received_data[4:2:-1])[0])*(self.__CT1/5)/1000
592             self.__I2 = float(struct.unpack(’H’, received_data[6:4:-1])[0])*(self.__CT1/5)/1000
593             self.__I3 = float(struct.unpack(’H’, received_data[8:6:-1])[0])*(self.__CT1/5)/1000
594             return No_error
595         else:
596             print("Transmitting error: Incorrect CRC")
597             return CRC_error
598
```

6.1.3.5 readPhasePower()

```
def MIC3.MIC1.readPhasePower (
                self )
```

Reads and calculates phase Power values with the help of PT1, PT2, CT1.

Definition at line 600 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.MIC1.__CT1, MIC3.MIC2.__P1, MIC3.MIC1.__P1, MIC3.MIC2.__P2, MIC3.MIC1.__P2, MIC3.MIC2.__P3, MIC3.MIC1.__P3, MIC3.MIC1.__PT1, and MIC3.MIC1.__PT2.

```
600     def readPhasePower(self):
601         #Calculate CRC16-MODBUS

602         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
603         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x3E, 0x00, 0x03])))
604         #The crc_Tx must include 4 hexadecimal characters.

605         #If crc_Tx =  10, function hex() will return 0xa, which is not expected

606         #Therefore, String format operator was used

607
608         #Send request

609         GPIO.output(self.__Control, GPIO.HIGH)
610         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x3E, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
611
612         #There is a delay caused by the converter. The program must wait before reading the result

613         sleep(0.01)
614
615         #Receive data

616         GPIO.output(self.__Control, GPIO.LOW)
617         cnt = 0
618         data_left = ser.inWaiting()
619         while (data_left == 0):
620             #wait for data

621             cnt=cnt+1
622             if (cnt < 50000): #wait for maximum 5 seconds

623                 sleep(0.0001)
624                 data_left = ser.inWaiting()
625             else:
626                 print("Transmitting error: Time out")
627                 return Trans_error
628         received_data = ser.read()
629         sleep(0.01)
630         data_left = ser.inWaiting()
631         received_data += ser.read(data_left)
632
633         if ((received_data[0]) != self.__Address):
634             print("Transmitting error: Data corrupted")
635             return Data_error
636         if (len(received_data) != 11):
637             print("Transmitting error: Data corrupted")
638             return Data_error
639
640         #Check the CRC code
```

```
641         crc_cal = hex(crc16(received_data[:9]))
642         crc_Rx = hex(struct.unpack('H',received_data[9:])[0])
643 
644         if crc_cal == crc_Rx:
645             self.__P1 = float(struct.unpack('h', received_data[4:2:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
646             self.__P2 = float(struct.unpack('h', received_data[6:4:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
647             self.__P3 = float(struct.unpack('h', received_data[8:6:-1])[0])*(self.__PT1/self.__PT2)*(self.
      __CT1/5)
648             return No_error
649         else:
650             print("Transmitting error: Incorrect CRC")
651             return CRC_error
652 
```

6.1.3.6 readPhaseVoltage()

```
def MIC3.MIC1.readPhaseVoltage (
                self )
```

Reads and calculates phase Voltages with the help of PT1 and PT2.

Definition at line 477 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.MIC1.__PT1, MIC3.MIC1.__PT2, MIC3.MIC2.__V1, MIC3.MIC1.__V1, MIC3.MIC2.__V2, MIC3.MIC1.__V2, MIC3.MIC2.__V3, and MIC3.MIC1.__V3.

```
477     def readPhaseVoltage(self):
478         #Calculate CRC16-MODBUS

479         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
480         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x31, 0x00, 0x03])))
481         #The crc_Tx must include 4 hexadecimal characters.

482         #If crc_Tx =  10, function hex() will return 0xa, which is not expected

483         #Therefore, String format operator was used

484 
485         #Send request

486         GPIO.output(self.__Control, GPIO.HIGH)
487         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x31, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
488 
489         #There is a delay caused by the converter. The program must wait before reading the result

490         sleep(0.01)
491 
492         #Receive data

493         GPIO.output(self.__Control, GPIO.LOW)
494         cnt = 0
495         data_left = ser.inWaiting()
496         while (data_left == 0):
497             #wait for data

498             cnt=cnt+1
499             if (cnt < 50000): #wait for maximum 5 seconds

500                 sleep(0.0001)
501                 data_left = ser.inWaiting()
502             else:
503                 print("Transmitting error: Time out")
504                 return Trans_error
505 
506         received_data = ser.read()
507         sleep(0.01)
508         data_left = ser.inWaiting()
509         received_data += ser.read(data_left)
```

```

        #Check if the data is correct
        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error
        if (len(received_data) != 11):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:9]))

        #DEBUG ONLY-----------------------------------------------
        #retval = ""
        #for character in received_data:
        #    retval += (’0123456789ABCDEF’[int((character)/16)])
        #    retval += (’0123456789ABCDEF’[int((character)%16)])
        #    retval += ’:’
        #print (retval[:-1])
        #print (crc_cal) #use for debugging only
        #---------------------------------------------------------

        crc_Rx = hex(struct.unpack(’H’,received_data[9:])[0])

        #print (crc_Rx) #use for degugging only

        if crc_cal == crc_Rx:
            self.__V1 = float(struct.unpack(’H’, received_data[4:2:-1])[0])*(self.__PT1/self.__PT2)/10
            self.__V2 = float(struct.unpack(’H’, received_data[6:4:-1])[0])*(self.__PT1/self.__PT2)/10
            self.__V3 = float(struct.unpack(’H’, received_data[8:6:-1])[0])*(self.__PT1/self.__PT2)/10
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error

```

6.1.3.7 readPowerFactor()

```
def MIC3.MIC1.readPowerFactor (
              self )
```

Reads and calculates Power Factors (PF)

Definition at line 764 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.MIC1.__PF1, MIC3.MIC1.__PF2, and MIC3.MIC1.__PF3.

```
764     def readPowerFactor(self):
765         #Calculate CRC16-MODBUS
766         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
767         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x4A, 0x00, 0x03])))
768         #The crc_Tx must include 4 hexadecimal characters.
769         #If crc_Tx =  10, function hex() will return 0xa, which is not expected
770         #Therefore, String format operator was used
771 
772         #Send request
```

```
773         GPIO.output(self.__Control, GPIO.HIGH)
774         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x4A, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
775 
776         #There is a delay caused by the converter. The program must wait before reading the result
777         sleep(0.02)
778 
779         #Receive data
780         GPIO.output(self.__Control, GPIO.LOW)
781         cnt = 0
782         data_left = ser.inWaiting()
783         while (data_left == 0):
784             #wait for data
785             cnt=cnt+1
786             if (cnt < 50000): #wait for maximum 5 seconds
787                 sleep(0.0001)
788                 data_left = ser.inWaiting()
789             else:
790                 print("Transmitting error: Time out")
791                 return Trans_error
792         received_data = ser.read()
793         sleep(0.02)
794         data_left = ser.inWaiting()
795         received_data += ser.read(data_left)
796 
797         if ((received_data[0]) != self.__Address):
798             print("Transmitting error: Data corrupted")
799             return Data_error
800         if (len(received_data) != 11):
801             print("Transmitting error: Data corrupted")
802             return Data_error
803 
804         #Check the CRC code
805         crc_cal = hex(crc16(received_data[:9]))
806         crc_Rx = hex(struct.unpack('H',received_data[9:])[0])
807 
808         if crc_cal == crc_Rx:
809             self.__PF1 = float(struct.unpack('h', received_data[4:2:-1])[0])/1000
810             self.__PF2 = float(struct.unpack('h', received_data[6:4:-1])[0])/1000
811             self.__PF3 = float(struct.unpack('h', received_data[8:6:-1])[0])/1000
812             return No_error
813         else:
814             print("Transmitting error: Incorrect CRC")
815             return CRC_error
816 
```

6.1.3.8 readPT1()

```
def MIC3.MIC1.readPT1 (
                self )
```

Reads PT1 variable needed for all other calculations.

Definition at line 276 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, and MIC3.MIC1.__PT1.

```
276     def readPT1(self):
277         #Calculate CRC16-MODBUS
278         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
279         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x05, 0x00, 0x02])))
280         #The crc_Tx must include 4 hexadecimal characters.
281         #If crc_Tx =  10, function hex() will return 0xa, which is not expected
```

```
        #Therefore, String format operator was used

        #Send request
        GPIO.output(self.__Control, GPIO.HIGH)
        ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x05, 0x00, 0x02, int(crc_Tx[3:],16), int(crc_Tx[1:3],16)]))

        #There is a delay caused by the converter. The program must wait before reading the result
        sleep(0.01)

        #Receive data
        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data
            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds
                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error

        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        #Check if the data is correct
        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error
        if (len(received_data) != 9):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:7]))

        #DEBUG ONLY------------------------------------------------
        #retval = ""
        #for character in received_data:
        #    retval += (’0123456789ABCDEF’[int((character)/16)])
        #    retval += (’0123456789ABCDEF’[int((character)%16)])
        #    retval += ’:’
        #print (retval[:-1])
        #print (crc_cal) #use for debugging only
        #----------------------------------------------------------

        crc_Rx = hex(struct.unpack(’H’,received_data[7:])[0])

        #print (crc_Rx) #use for degugging only

        if crc_cal == crc_Rx:
            self.__PT1 = float(struct.unpack(’I’, received_data[6:2:-1])[0])
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error

```

6.1.3.9 readPT2()

```
def MIC3.MIC1.readPT2 (
                self )
```

Reads PT2 variable needed for all other calculations.

Definition at line 343 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, and M↩ IC3.MIC1.__PT2.

```
343     def readPT2(self):
344         #Calculate CRC16-MODBUS
345         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
346         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x07, 0x00, 0x01])))
347         #The crc_Tx must include 4 hexadecimal characters.
348         #If crc_Tx =  10, function hex() will return 0xa, which is not expected
349         #Therefore, String format operator was used
350 
351         #Send request
352         GPIO.output(self.__Control, GPIO.HIGH)
353         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x07, 0x00, 0x01, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
354 
355         #There is a delay caused by the converter. The program must wait before reading the result
356         sleep(0.01)
357 
358         #Receive data
359         GPIO.output(self.__Control, GPIO.LOW)
360         cnt = 0
361         data_left = ser.inWaiting()
362         while (data_left == 0):
363             #wait for data
364             cnt=cnt+1
365             if (cnt < 50000): #wait for maximum 5 seconds
366                 sleep(0.0001)
367                 data_left = ser.inWaiting()
368             else:
369                 print("Transmitting error: Time out")
370                 return Trans_error
371 
372         received_data = ser.read()
373         sleep(0.01)
374         data_left = ser.inWaiting()
375         received_data += ser.read(data_left)
376 
377         #Check if the data is correct
378         if ((received_data[0]) != self.__Address):
379             print("Transmitting error: Data corrupted")
380             return Data_error
381         if (len(received_data) != 7):
382             print("Transmitting error: Data corrupted")
383             return Data_error
384 
385         #Check the CRC code
386         crc_cal = hex(crc16(received_data[:5]))
387 
388         #DEBUG ONLY-----------------------------------------------
389         #retval = ""
390         #for character in received_data:
391         #    retval += (’0123456789ABCDEF’[int((character)/16)])
392         #    retval += (’0123456789ABCDEF’[int((character)%16)])
393         #    retval += ’:’
```

```
394         #print (retval[:-1])

395         #print (crc_cal) #use for debugging only

396         #----------------------------------------------------------

397
398         crc_Rx = hex(struct.unpack('H',received_data[5:])[0])
399
400         #print (crc_Rx) #use for degugging only

401
402         if crc_cal == crc_Rx:
403             self.__PT2 = float(struct.unpack('H', received_data[4:2:-1])[0])
404             return No_error
405         else:
406             print("Transmitting error: Incorrect CRC")
407             return CRC_error
408
```

6.1.3.10 readReactivePower()

```
def MIC3.MIC1.readReactivePower (
                self )
```

Reads and calculates Reactive Power (Q) values with the help of PT1, PT2, CT1.

Definition at line 654 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC1.__Address, MIC3.MIC2.__Control, MIC3.MIC1.__Control, MIC3.MIC1.__CT1, MIC3.MIC1.__PT1, MIC3.MIC1.__PT2, MIC3.MIC1.__Q1, MIC3.MIC1.__Q2, and MIC3.MIC1.__Q3.

```
654     def readReactivePower(self):
655         #Calculate CRC16-MODBUS

656         crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
657         crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x01, 0x42, 0x00, 0x03])))
658         #The crc_Tx must include 4 hexadecimal characters.

659         #If crc_Tx =  10, function hex() will return 0xa, which is not expected

660         #Therefore, String format operator was used

661
662         #Send request

663         GPIO.output(self.__Control, GPIO.HIGH)
664         ser.write(serial.to_bytes([self.__Address, 0x03, 0x01, 0x42, 0x00, 0x03, int(crc_Tx[3:],16), int(
      crc_Tx[1:3],16)]))
665
666         #There is a delay caused by the converter. The program must wait before reading the result

667         sleep(0.02)
668
669         #Receive data

670         GPIO.output(self.__Control, GPIO.LOW)
671         cnt = 0
672         data_left = ser.inWaiting()
673         while (data_left == 0):
674             #wait for data

675             cnt=cnt+1
676             if (cnt < 50000): #wait for maximum 5 seconds

677                 sleep(0.0001)
678                 data_left = ser.inWaiting()
679             else:
680                 print("Transmitting error: Time out")
681                 return Trans_error
682         received_data = ser.read()
683         sleep(0.02)
```

```
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error
        if (len(received_data) != 11):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:9]))
        crc_Rx = hex(struct.unpack('H',received_data[9:])[0])

        if crc_cal == crc_Rx:
            self.__Q1 = float(struct.unpack('h', received_data[4:2:-1])[0])*(self.__PT1/self.__PT2)*(self.__CT1/5)
            self.__Q2 = float(struct.unpack('h', received_data[6:4:-1])[0])*(self.__PT1/self.__PT2)*(self.__CT1/5)
            self.__Q3 = float(struct.unpack('h', received_data[8:6:-1])[0])*(self.__PT1/self.__PT2)*(self.__CT1/5)
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error

```

6.1.4 Member Data Documentation

6.1.4.1 __Address

MIC3.MIC1.__Address [private]

Definition at line 251 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readCT1(), MIC3.MIC1.readFrequency(), MIC3.MIC1.readPhaseCurrent(), MIC3.MIC1.readPhasePower(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPowerFactor(), MIC3.MIC1.readPT1(), MIC3.MIC1.readPT2(), and MIC3.MIC1.readReactivePower().

6.1.4.2 __Control

MIC3.MIC1.__Control [private]

Definition at line 250 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readCT1(), MIC3.MIC1.readFrequency(), MIC3.MIC1.readPhaseCurrent(), MIC3.MIC1.readPhasePower(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPowerFactor(), MIC3.MIC1.readPT1(), MIC3.MIC1.readPT2(), and MIC3.MIC1.readReactivePower().

6.1.4.3 __CT1

```
MIC3.MIC1.__CT1  [private]
```

Definition at line 254 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readCT1(), MIC3.MIC1.readPhaseCurrent(), MI↩C3.MIC1.readPhasePower(), and MIC3.MIC1.readReactivePower().

6.1.4.4 __F

```
MIC3.MIC1.__F  [private]
```

Definition at line 273 of file MIC3.py.

Referenced by MIC3.MIC1.readFrequency().

6.1.4.5 __I1

```
MIC3.MIC1.__I1  [private]
```

Definition at line 258 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseCurrent().

6.1.4.6 __I2

```
MIC3.MIC1.__I2  [private]
```

Definition at line 259 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseCurrent().

6.1.4.7 __I3

```
MIC3.MIC1.__I3  [private]
```

Definition at line 260 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseCurrent().

6.1.4.8 __P1

MIC3.MIC1.__P1 [private]

Definition at line 261 of file MIC3.py.

Referenced by MIC3.MIC1.readPhasePower().

6.1.4.9 __P2

MIC3.MIC1.__P2 [private]

Definition at line 262 of file MIC3.py.

Referenced by MIC3.MIC1.readPhasePower().

6.1.4.10 __P3

MIC3.MIC1.__P3 [private]

Definition at line 263 of file MIC3.py.

Referenced by MIC3.MIC1.readPhasePower().

6.1.4.11 __PF1

MIC3.MIC1.__PF1 [private]

Definition at line 270 of file MIC3.py.

Referenced by MIC3.MIC1.readPowerFactor().

6.1.4.12 __PF2

MIC3.MIC1.__PF2 [private]

Definition at line 271 of file MIC3.py.

Referenced by MIC3.MIC1.readPowerFactor().

6.1.4.13 __PF3

```
MIC3.MIC1.__PF3  [private]
```

Definition at line 272 of file MIC3.py.

Referenced by MIC3.MIC1.readPowerFactor().

6.1.4.14 __PT1

```
MIC3.MIC1.__PT1  [private]
```

Definition at line 252 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readPhasePower(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPT1(), and MIC3.MIC1.readReactivePower().

6.1.4.15 __PT2

```
MIC3.MIC1.__PT2  [private]
```

Definition at line 253 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readPhasePower(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPT2(), and MIC3.MIC1.readReactivePower().

6.1.4.16 __Q1

```
MIC3.MIC1.__Q1  [private]
```

Definition at line 264 of file MIC3.py.

Referenced by MIC3.MIC1.readReactivePower().

6.1.4.17 __Q2

```
MIC3.MIC1.__Q2  [private]
```

Definition at line 265 of file MIC3.py.

Referenced by MIC3.MIC1.readReactivePower().

6.1.4.18 __Q3

```
MIC3.MIC1.__Q3  [private]
```

Definition at line 266 of file MIC3.py.

Referenced by MIC3.MIC1.readReactivePower().

6.1.4.19 __S1

```
MIC3.MIC1.__S1  [private]
```

Definition at line 267 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower().

6.1.4.20 __S2

```
MIC3.MIC1.__S2  [private]
```

Definition at line 268 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower().

6.1.4.21 __S3

```
MIC3.MIC1.__S3  [private]
```

Definition at line 269 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower().

6.1.4.22 __V1

```
MIC3.MIC1.__V1  [private]
```

Definition at line 255 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseVoltage().

6.1.4.23 __V2

```
MIC3.MIC1.__V2 [private]
```

Definition at line 256 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseVoltage().

6.1.4.24 __V3

```
MIC3.MIC1.__V3 [private]
```

Definition at line 257 of file MIC3.py.

Referenced by MIC3.MIC1.readPhaseVoltage().

The documentation for this class was generated from the following file:

- Modbus/MIC3.py

6.2 MIC3.MIC2 Class Reference

Unused class for MIC2 energy meter; it is missing the PT!, PT2, CT1 control variables.

Public Member Functions

- def __init__ (self, Id, Control)
- def readPhaseVoltage (self)
- def readPhaseCurrent (self)
- def readPhasePower (self)
- def readFrequency (self)

Private Attributes

- __Control
- __Address
- __V1
- __V2
- __V3
- __I1
- __I2
- __I3
- __P1
- __P2
- __P3
- __F

6.2.1 Detailed Description

Unused class for MIC2 energy meter; it is missing the PT!, PT2, CT1 control variables.

Definition at line 25 of file MIC3.py.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 __init__()

```
def MIC3.MIC2.__init__ (
              self,
              Id,
              Control )
```

Definition at line 26 of file MIC3.py.

```
26     def __init__(self, Id, Control):
27         self.__Control = Control
28         self.__Address = Id
29         self.__V1 = 0
30         self.__V2 = 0
31         self.__V3 = 0
32         self.__I1 = 0
33         self.__I2 = 0
34         self.__I3 = 0
35         self.__P1 = 0
36         self.__P2 = 0
37         self.__P3 = 0
38         self.__F  = 0
39
```

6.2.3 Member Function Documentation

6.2.3.1 readFrequency()

```
def MIC3.MIC2.readFrequency (
              self )
```

Definition at line 196 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC2.__Control, and MIC3.MIC2.__F.

```
    def readFrequency(self):
        #Calculate CRC16-MODBUS
        crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
        crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x40, 0x00, 0x00, 0x02])))
        #The crc_Tx must include 4 hexadecimal characters.
        #If crc_Tx =  10, function hex() will return 0xa, which is not expected
        #Therefore, String format operator was used

        #Send request
        GPIO.output(self.__Control, GPIO.HIGH)
        ser.write(serial.to_bytes([self.__Address, 0x03, 0x40, 0x00, 0x00, 0x02, int(crc_Tx[3:],16), int(crc_Tx[1:3],16)]))

        #There is a delay caused by the converter. The program must wait before reading the result
        sleep(0.004)

        #Receive data
        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data
            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds
                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error
        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:7]))
        crc_Rx = hex(struct.unpack('H',received_data[7:])[0])

        if crc_cal == crc_Rx:
            self.__F = struct.unpack('f', received_data[6:2:-1])[0]
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error
#---------------------------------END OF MIC2------------------------------------
#--------------------------------------------------------------------------------


```

6.2.3.2 readPhaseCurrent()

```
def MIC3.MIC2.readPhaseCurrent (
                self )
```

Definition at line 96 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC2.__Control, MIC3.MIC2.__I1, MIC3.MIC2.__I2, and MIC3.MIC2.__I3.

```
    def readPhaseCurrent(self):
        #Calculate CRC16-MODBUS
        crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
        crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x40, 0x12, 0x00, 0x06])))
        #The crc_Tx must include 4 hexadecimal characters.
        #If crc_Tx =  10, function hex() will return 0xa, which is not expected
        #Therefore, String format operator was used

        #Send request
        GPIO.output(self.__Control, GPIO.HIGH)
        ser.write(serial.to_bytes([self.__Address, 0x03, 0x40, 0x12, 0x00, 0x06, int(crc_Tx[3:],16), int(crc_Tx[1:3],16)]))

        #There is a delay caused by the converter. The program must wait before reading the result
        sleep(0.004)

        #Receive data
        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data
            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds
                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error
        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code
        crc_cal = hex(crc16(received_data[:15]))
        crc_Rx = hex(struct.unpack('H',received_data[15:])[0])

        if crc_cal == crc_Rx:
            self.__I1 = struct.unpack('f', received_data[6:2:-1])[0]
            self.__I2 = struct.unpack('f', received_data[10:6:-1])[0]
            self.__I3 = struct.unpack('f', received_data[14:10:-1])[0]
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error

```

6.2.3.3 readPhasePower()

```
def MIC3.MIC2.readPhasePower (
            self )
```

Definition at line 146 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC2.__Control, MIC3.MIC2.__P1, MIC3.MIC2.__P2, and MIC3.MIC2.__P3.

```
    def readPhasePower(self):
        #Calculate CRC16-MODBUS

        crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
        crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x40, 0x1c, 0x00, 0x06])))
        #The crc_Tx must include 4 hexadecimal characters.

        #If crc_Tx =  10, function hex() will return 0xa, which is not expected

        #Therefore, String format operator was used


        #Send request

        GPIO.output(self.__Control, GPIO.HIGH)
        ser.write(serial.to_bytes([self.__Address, 0x03, 0x40, 0x1c, 0x00, 0x06, int(crc_Tx[3:],16), int(crc_Tx[1:3],16)]))

        #There is a delay caused by the converter. The program must wait before reading the result

        sleep(0.004)

        #Receive data

        GPIO.output(self.__Control, GPIO.LOW)
        cnt = 0
        data_left = ser.inWaiting()
        while (data_left == 0):
            #wait for data

            cnt=cnt+1
            if (cnt < 50000): #wait for maximum 5 seconds

                sleep(0.0001)
                data_left = ser.inWaiting()
            else:
                print("Transmitting error: Time out")
                return Trans_error
        received_data = ser.read()
        sleep(0.01)
        data_left = ser.inWaiting()
        received_data += ser.read(data_left)

        if ((received_data[0]) != self.__Address):
            print("Transmitting error: Data corrupted")
            return Data_error

        #Check the CRC code

        crc_cal = hex(crc16(received_data[:15]))
        crc_Rx = hex(struct.unpack(’H’,received_data[15:])[0])

        if crc_cal == crc_Rx:
            self.__P1 = struct.unpack(’f’, received_data[6:2:-1])[0]
            self.__P2 = struct.unpack(’f’, received_data[10:6:-1])[0]
            self.__P3 = struct.unpack(’f’, received_data[14:10:-1])[0]
            return No_error
        else:
            print("Transmitting error: Incorrect CRC")
            return CRC_error

```

6.2.3.4 readPhaseVoltage()

```
def MIC3.MIC2.readPhaseVoltage (
                self )
```

Definition at line 40 of file MIC3.py.

References MIC3.MIC2.__Address, MIC3.MIC2.__Control, MIC3.MIC2.__V1, MIC3.MIC2.__V2, and MIC3.MIC2.__V3.

```
def readPhaseVoltage(self):
    #Calculate CRC16-MODBUS
    crc16 = crcmod.mkCrcFun(0x18005, rev=True, initCrc = 0xFFFF, xorOut = 0x0000)
    crc_Tx = ".%4x"%(crc16(serial.to_bytes([self.__Address, 0x03, 0x40, 0x02, 0x00, 0x06])))
    #The crc_Tx must include 4 hexadecimal characters.
    #If crc_Tx =  10, function hex() will return 0xa, which is not expected
    #Therefore, String format operator was used

    #Send request
    GPIO.output(self.__Control, GPIO.HIGH)
    ser.write(serial.to_bytes([self.__Address, 0x03, 0x40, 0x02, 0x00, 0x06, int(crc_Tx[3:],16), int(crc_Tx[1:3],16)]))

    #There is a delay caused by the converter. The program must wait before reading the result
    sleep(0.004)

    #Receive data
    GPIO.output(self.__Control, GPIO.LOW)
    cnt = 0
    data_left = ser.inWaiting()
    while (data_left == 0):
        #wait for data
        cnt=cnt+1
        if (cnt < 50000): #wait for maximum 5 seconds
            sleep(0.0001)
            data_left = ser.inWaiting()
        else:
            print("Transmitting error: Time out")
            return Trans_error

    received_data = ser.read()
    sleep(0.01)
    data_left = ser.inWaiting()
    received_data += ser.read(data_left)

    if ((received_data[0]) != self.__Address):
        print("Transmitting error: Data corrupted")
        return Data_error

    #Check the CRC code
    crc_cal = hex(crc16(received_data[:15]))

    #print (crc_cal) #use for debugging only

    crc_Rx = hex(struct.unpack('H',received_data[15:])[0])

    #print (crc_Rx) #use for degugging only

    if crc_cal == crc_Rx:
        self.__V1 = struct.unpack('f', received_data[6:2:-1])[0]
        self.__V2 = struct.unpack('f', received_data[10:6:-1])[0]
        self.__V3 = struct.unpack('f', received_data[14:10:-1])[0]
        return No_error
    else:
        print("Transmitting error: Incorrect CRC")
        return CRC_error

```

6.2.4 Member Data Documentation

6.2.4.1 __Address

MIC3.MIC2.__Address [private]

Definition at line 28 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readCT1(), MIC3.MIC2.readFrequency(), MIC3.MIC1.readFrequency(), MIC3.MIC2.readPhaseCurrent(), MIC3.MIC1.readPhaseCurrent(), MIC3.MIC2.readPhasePower(), MIC3.MIC1.readPhasePower(), MIC3.MIC2.readPhaseVoltage(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPowerFactor(), MIC3.MIC1.readPT1(), MIC3.MIC1.readPT2(), and MIC3.MIC1.readReactivePower().

6.2.4.2 __Control

```
MIC3.MIC2.__Control  [private]
```

Definition at line 27 of file MIC3.py.

Referenced by MIC3.MIC1.readApparentPower(), MIC3.MIC1.readCT1(), MIC3.MIC2.readFrequency(), MIC3.MIC1.readFrequency(), MIC3.MIC2.readPhaseCurrent(), MIC3.MIC1.readPhaseCurrent(), MIC3.MIC2.readPhasePower(), MIC3.MIC1.readPhasePower(), MIC3.MIC2.readPhaseVoltage(), MIC3.MIC1.readPhaseVoltage(), MIC3.MIC1.readPowerFactor(), MIC3.MIC1.readPT1(), MIC3.MIC1.readPT2(), and MIC3.MIC1.readReactivePower().

6.2.4.3 __F

```
MIC3.MIC2.__F  [private]
```

Definition at line 38 of file MIC3.py.

Referenced by MIC3.MIC2.readFrequency(), and MIC3.MIC1.readFrequency().

6.2.4.4 __I1

```
MIC3.MIC2.__I1  [private]
```

Definition at line 32 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseCurrent(), and MIC3.MIC1.readPhaseCurrent().

6.2.4.5 __I2

```
MIC3.MIC2.__I2  [private]
```

Definition at line 33 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseCurrent(), and MIC3.MIC1.readPhaseCurrent().

6.2.4.6 __I3

```
MIC3.MIC2.__I3  [private]
```

Definition at line 34 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseCurrent(), and MIC3.MIC1.readPhaseCurrent().

6.2.4.7 __P1

```
MIC3.MIC2.__P1  [private]
```

Definition at line 35 of file MIC3.py.

Referenced by MIC3.MIC2.readPhasePower(), and MIC3.MIC1.readPhasePower().

6.2.4.8 __P2

```
MIC3.MIC2.__P2  [private]
```

Definition at line 36 of file MIC3.py.

Referenced by MIC3.MIC2.readPhasePower(), and MIC3.MIC1.readPhasePower().

6.2.4.9 __P3

```
MIC3.MIC2.__P3  [private]
```

Definition at line 37 of file MIC3.py.

Referenced by MIC3.MIC2.readPhasePower(), and MIC3.MIC1.readPhasePower().

6.2.4.10 __V1

```
MIC3.MIC2.__V1  [private]
```

Definition at line 29 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseVoltage(), and MIC3.MIC1.readPhaseVoltage().

6.2.4.11 __V2

```
MIC3.MIC2.__V2 [private]
```

Definition at line 30 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseVoltage(), and MIC3.MIC1.readPhaseVoltage().

6.2.4.12 __V3

```
MIC3.MIC2.__V3 [private]
```

Definition at line 31 of file MIC3.py.

Referenced by MIC3.MIC2.readPhaseVoltage(), and MIC3.MIC1.readPhaseVoltage().

The documentation for this class was generated from the following file:

- Modbus/MIC3.py

Chapter 7

File Documentation

7.1 Modbus/main3.py File Reference

Namespaces

- main3

Functions

- def main3.on_connect (client, userdata, flags, rc)

  Executes on MQTT client connect to broker and sets flags.
- def main3.on_disconnect (client, userdata, rc)

  Executes on MQTT client disconnect and sets flags.

Variables

- string main3.broker = "broker.hivemq.com"

  MQTT broker.
- string main3.path_local = "/media/DATABASE/modbusData.db"

  Path for modbus database.
- string main3.path = "/mnt/dav/Data/modbusData.db"
- string main3.path_local_user = "/media/DATABASE/usertable.sqlite3"

  Path for users database.
- string main3.path_user = "/mnt/dav/Data/usertable.sqlite3"
- main3.con_user_local = lite.connect(path_local_user)
- main3.cur_user_local = con_user_local.cursor()
- main3.con_user = lite.connect(path_user)

  Initial DB connection check.
- main3.cur_user = con_user.cursor()
- main3.con_local = lite.connect(path_local)
- main3.cur_local = con_local.cursor()
- main3.con = lite.connect(path)
- main3.cur = con.cursor()
- main3.dataRef1 = cur.fetchone()
- int main3.err_cnt = 0

- main3.client = mqtt.Client()
- main3.connected_flag
- main3.bad_connection_flag
- main3.on_connect
- main3.on_disconnect
- int main3.control_pin = 18
- main3.meter1 = MIC.MIC1(0x01, control_pin)

  Initializes meter.
- main3.meter2 = MIC.MIC1(0x02, control_pin)
- main3.meter3 = MIC.MIC1(0x03, control_pin)
- main3.meter4 = MIC.MIC1(0x04, control_pin)
- main3.meter5 = MIC.MIC1(0x05, control_pin)
- int main3.time_send = 1
- main3.current_time = time.ctime(time.time())
- main3.readingPT1 = meter1.readPT1()
- main3.readingPT2 = meter1.readPT2()
- main3.readingCT1 = meter1.readCT1()
- main3.reading = meter1.readPhaseVoltage()
- string main3.Message
- main3.data = cur.fetchone()
- tuple main3.setPoint = (data[0]+data[1]+data[2])
- dictionary main3.dataSend

7.2 Modbus/MIC3.py File Reference

Classes

- class MIC3.MIC2

  Unused class for MIC2 energy meter; it is missing the PT!, PT2, CT1 control variables.
- class MIC3.MIC1

  Class for reading Modbus data from MIC1 energy meter.

Namespaces

- MIC3

Variables

- MIC3.ser = serial.Serial("/dev/ttyS0", 38400)

  This library is made for reading MIC/MIC2 energy meters with a MAX485 module MIC2 only reads data from registers.
- int MIC3.Data_error = -3
- int MIC3.CRC_error = -2
- int MIC3.Trans_error = -1
- int MIC3.No_error = 0

7.3 README.md File Reference

7.4 userID/SQLfunction.py File Reference

Namespaces

- SQLfunction

Functions

- def SQLfunction.on_connect (client, userdata, flags, rc)

  Executes on MQTT client connect to broker, sets flags and subscribes.
- def SQLfunction.on_disconnect (client, userdata, rc)

  Executes on MQTT client disconnect and sets flags.
- def SQLfunction.update_callback (client, userdata, message)

  Callback function that parses RFID swipe message from Photon and checks in the DB what to publish as answer Publish output is structured as "1;2" where 1=socket number and 2=answer to Photon.
- def SQLfunction.new_photonMeasure_callback (client, userdata, message)

  New Callback for Photon measurements that parses, checks DB for user data like name and carname, then logs into 'measurements'.
- def SQLfunction.old_photonMeasure_callback (client, userdata, message)

  Old Photon measurements callback that parses '' separated values.
- def SQLfunction.send_admin ()

  Function to send admin email if one user has been plugged in at a socket for over 4 hours still in beta mode and needs improvements.
- def SQLfunction.send_email ()

Variables

- SQLfunction.DISCONNECT_TIME = int(4 $*$ 60 $*$ 60)

  Const.
- int SQLfunction.email_cntr = 0
- int SQLfunction.SSLport = 465
- string SQLfunction.smtp_server = "smtp.gmail.com"
- string SQLfunction.sender_email = "tpi97364@gmail.com"

  Email sender address for Pi.
- string SQLfunction.receiver_email = ""

  Holder for email addresses of receivers.
- string SQLfunction.sender_password = "controlsystem"

  Pi email password.
- string SQLfunction.email_message

  Default email message.
- SQLfunction.email_context = ssl.create_default_context()
- SQLfunction.con = None

  Initial DB connection check.
- string SQLfunction.broker = "broker.hivemq.com"

  MQTT broker address.
- string SQLfunction.path_local = "/media/DATABASE/usertable.sqlite3"

  Path to users database path = "./userList" #Use internal memory - old DB.

- string SQLfunction.path = "/mnt/dav/Data/usertable.sqlite3"
- SQLfunction.con_local = lite.connect(path_local)
- SQLfunction.cur_local = con_local.cursor()
- SQLfunction.cur = con.cursor()
- SQLfunction.dataRef1 = cur.fetchone()
- int SQLfunction.err_cnt = 0
- SQLfunction.client = mqtt.Client()
- SQLfunction.connected_flag
- SQLfunction.bad_connection_flag
- SQLfunction.on_connect
- SQLfunction.on_disconnect
- SQLfunction.current_time = time.ctime(time.time())

Index

__F
  MIC3::MIC1, 46
  MIC3::MIC2, 56
__Address
  MIC3::MIC1, 45
  MIC3::MIC2, 55
__CT1
  MIC3::MIC1, 45
__Control
  MIC3::MIC1, 45
  MIC3::MIC2, 56
__I1
  MIC3::MIC1, 46
  MIC3::MIC2, 56
__I2
  MIC3::MIC1, 46
  MIC3::MIC2, 56
__I3
  MIC3::MIC1, 46
  MIC3::MIC2, 56
__P1
  MIC3::MIC1, 46
  MIC3::MIC2, 57
__P2
  MIC3::MIC1, 47
  MIC3::MIC2, 57
__P3
  MIC3::MIC1, 47
  MIC3::MIC2, 57
__PF1
  MIC3::MIC1, 47
__PF2
  MIC3::MIC1, 47
__PF3
  MIC3::MIC1, 47
__PT1
  MIC3::MIC1, 48
__PT2
  MIC3::MIC1, 48
__Q1
  MIC3::MIC1, 48
__Q2
  MIC3::MIC1, 48
__Q3
  MIC3::MIC1, 48
__S1
  MIC3::MIC1, 49
__S2
  MIC3::MIC1, 49
__S3
  MIC3::MIC1, 49
__V1
  MIC3::MIC1, 49
  MIC3::MIC2, 57
__V2
  MIC3::MIC1, 49
  MIC3::MIC2, 57
__V3
  MIC3::MIC1, 50
  MIC3::MIC2, 58
__init__
  MIC3::MIC1, 32
  MIC3::MIC2, 51

bad_connection_flag
  main3, 11
  SQLfunction, 26
broker
  main3, 11
  SQLfunction, 26

CRC_error
  MIC3, 18
client
  main3, 11
  SQLfunction, 26
con
  main3, 11
  SQLfunction, 26
con_local
  main3, 12
  SQLfunction, 26
con_user
  main3, 12
con_user_local
  main3, 12
connected_flag
  main3, 12
  SQLfunction, 27
control_pin
  main3, 12
cur
  main3, 12
  SQLfunction, 27
cur_local
  main3, 13
  SQLfunction, 27
cur_user
  main3, 13

cur_user_local
  main3, 13
current_time
  main3, 13
  SQLfunction, 27

DISCONNECT_TIME
  SQLfunction, 27
data
  main3, 13
Data_error
  MIC3, 18
dataRef1
  main3, 13
  SQLfunction, 27
dataSend
  main3, 14

email_cntr
  SQLfunction, 28
email_context
  SQLfunction, 28
email_message
  SQLfunction, 28
err_cnt
  main3, 14
  SQLfunction, 28

MIC3, 17
  CRC_error, 18
  Data_error, 18
  No_error, 18
  ser, 18
  Trans_error, 18
MIC3.MIC1, 31
MIC3.MIC2, 50
MIC3::MIC1
  __F, 46
  __Address, 45
  __CT1, 45
  __Control, 45
  __I1, 46
  __I2, 46
  __I3, 46
  __P1, 46
  __P2, 47
  __P3, 47
  __PF1, 47
  __PF2, 47
  __PF3, 47
  __PT1, 48
  __PT2, 48
  __Q1, 48
  __Q2, 48
  __Q3, 48
  __S1, 49
  __S2, 49
  __S3, 49
  __V1, 49
  __V2, 49
  __V3, 50
  __init__, 32
  readApparentPower, 33
  readCT1, 34
  readFrequency, 35
  readPT1, 41
  readPT2, 42
  readPhaseCurrent, 37
  readPhasePower, 38
  readPhaseVoltage, 39
  readPowerFactor, 40
  readReactivePower, 44
MIC3::MIC2
  __F, 56
  __Address, 55
  __Control, 56
  __I1, 56
  __I2, 56
  __I3, 56
  __P1, 57
  __P2, 57
  __P3, 57
  __V1, 57
  __V2, 57
  __V3, 58
  __init__, 51
  readFrequency, 51
  readPhaseCurrent, 52
  readPhasePower, 53
  readPhaseVoltage, 54
main3, 9
  bad_connection_flag, 11
  broker, 11
  client, 11
  con, 11
  con_local, 12
  con_user, 12
  con_user_local, 12
  connected_flag, 12
  control_pin, 12
  cur, 12
  cur_local, 13
  cur_user, 13
  cur_user_local, 13
  current_time, 13
  data, 13
  dataRef1, 13
  dataSend, 14
  err_cnt, 14
  Message, 14
  meter1, 14
  meter2, 14
  meter3, 15
  meter4, 15
  meter5, 15
  on_connect, 10, 15
  on_disconnect, 10, 15

path, 15
path_local, 16
path_local_user, 16
path_user, 16
reading, 16
readingCT1, 16
readingPT1, 16
readingPT2, 17
setPoint, 17
time_send, 17
Message
main3, 14
meter1
main3, 14
meter2
main3, 14
meter3
main3, 15
meter4
main3, 15
meter5
main3, 15
Modbus/MIC3.py, 60
Modbus/main3.py, 59

new_photonMeasure_callback
SQLfunction, 20
No_error
MIC3, 18

old_photonMeasure_callback
SQLfunction, 21
on_connect
main3, 10, 15
SQLfunction, 21, 28
on_disconnect
main3, 10, 15
SQLfunction, 22, 29

path
main3, 15
SQLfunction, 29
path_local
main3, 16
SQLfunction, 29
path_local_user
main3, 16
path_user
main3, 16

README.md, 61
readApparentPower
MIC3::MIC1, 33
readCT1
MIC3::MIC1, 34
readFrequency
MIC3::MIC1, 35
MIC3::MIC2, 51
readPT1
MIC3::MIC1, 41
readPT2
MIC3::MIC1, 42
readPhaseCurrent
MIC3::MIC1, 37
MIC3::MIC2, 52
readPhasePower
MIC3::MIC1, 38
MIC3::MIC2, 53
readPhaseVoltage
MIC3::MIC1, 39
MIC3::MIC2, 54
readPowerFactor
MIC3::MIC1, 40
readReactivePower
MIC3::MIC1, 44
reading
main3, 16
readingCT1
main3, 16
readingPT1
main3, 16
readingPT2
main3, 17
receiver_email
SQLfunction, 29

SQLfunction, 19
bad_connection_flag, 26
broker, 26
client, 26
con, 26
con_local, 26
connected_flag, 27
cur, 27
cur_local, 27
current_time, 27
DISCONNECT_TIME, 27
dataRef1, 27
email_cntr, 28
email_context, 28
email_message, 28
err_cnt, 28
new_photonMeasure_callback, 20
old_photonMeasure_callback, 21
on_connect, 21, 28
on_disconnect, 22, 29
path, 29
path_local, 29
receiver_email, 29
SSLport, 30
send_admin, 22
send_email, 23
sender_email, 29
sender_password, 30
smtp_server, 30
update_callback, 24
SSLport
SQLfunction, 30

send_admin
SQLfunction, 22
send_email
SQLfunction, 23
sender_email
SQLfunction, 29
sender_password
SQLfunction, 30
ser
MIC3, 18
setPoint
main3, 17
smtp_server
SQLfunction, 30

time_send
main3, 17
Trans_error
MIC3, 18

update_callback
SQLfunction, 24
userID/SQLfunction.py, 61