SMS-Control software

Table des matières

1.Document history

Version	Date	Description
1.0	Jan 8, 2025	Initial document

Copyright MC Hobby SPRL – All right reserved

2.Requirements

A **SIM card without PinCode** is required (this can be done with a smartphone).

The **SIM must be properly initialized**, this is usually done by inserting the SIM into a smartphone and doing a first call over the mobile network (eg : Belgium).

Only after, the SIM can be inserted within the **gsms-control** to power it on and start the configuration.

3.Quick tour

The gate-control project uses a 4G Base Station kit.

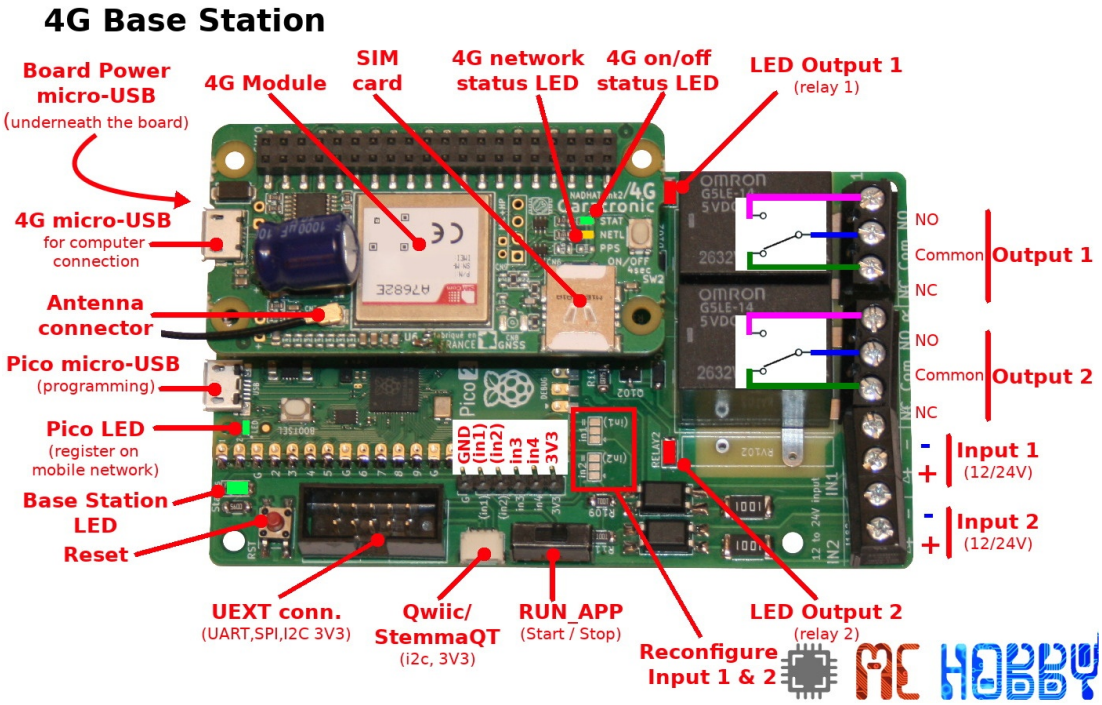


Fig : features of the 4G Base station board.

The key features of the board are :

•**2x relay output** : this galvanic isolation allow to control high power devices. By design this output can be used up to 24V DC or 48V AC.

•**2x opto-isolated input** : able to read 12V / 24V input signal. See https://github.com/mchobby/micropython-4G-BASE-STATION code repository for wiring examples.

•**2x 3,3V input** : for extra application

•**1x UEXT port** : exposing a SPI bus, I2C bus UART and 3,3V power for custom application

•**1x Qwiic/StemmaQt port** : exposing 3V3 I2C to connect i2C board/sensor for custom application

•**Pico Powered** : powered with Raspberry-Pi Pico 2 microcontroler.

•**MicroPython Powered** : this solution is written in python. The code can be tuned to your need. See the technical documentation to schematic and details.

•**5V Powered** : the entire project runs with 5V micro-USB power supply.

Copyright MC Hobby SPRL – All right reserved

The relays are able to handle grid power switching. The board also have location to solder varistor (transient effect suppressor) coupled to the relays. The relay traces (copper, 3,5mm wide) will allow substantial power switching. All-in-All this board could withstand higher voltage handling but it is not intended to by design! If you do so, it is at your own risk.

4.Quick Setup

Insert the **configured** SIM card into the 4G board.

Power-up the board on the **board power micro-USB**.

After few seconds, the base station LED will pulse every 3 seconds signaling the normal operation.

Use your mobile phone to send SMS to the sms-control. Any attempt of phone call will be immediately rejected.

From now, the user can send one of the following SMS command :

- **say** : return the first parameter (or replace it by « hello »).
- **info** : return several SMS with information on the system and the two inputs.
- **error** : return an error message (programming example).
- **punish** : show how to send a message to phone number given as first parameter.
- **on1** : turn on the first relay (use off1 to switch it off).
- **on2** : turn on the second relay (use off2 to switch it off).

5.Status LEDs

5.1.Pico LED

The LED on the Raspberry-Pi Pico (next to its USB connector) will **lit when the board gets registered on a mobile network**..

Note : the Base Station LED will lit (fixed) while powering-up the 4G module THEN pulse at 500ms while registering the mobile network.

5.2.4G module LEDs

The 4G module fits 3 status LEDs :

- **STAT (green)** : Powered and internal initialization completed
- **NETL (orange)** :
  - Fixed = not registered on network
  - 2 blink/sec = registered on network
- **PPS (blue)** : -to be defined-

Note : 4G module can be power-off by pressing the on/off button for 5 seconds then release it.

Copyright MC Hobby SPRL – All right reserved

5.3.Base Station LED

The base station LED can use multiples modes to communicates about the software state.

•**Heartbeat** : just a short blink every Nth seconds. The time is a useful information.

•**Pulsing** : where the LED progressively light-up then progressively light down. The time for a full cycle is a useful information.

•**Error** : series of fast blinking followed by a given number of slow blinks (this is the numerical error code).

Mode	Information	Description
Pulse	3sec/pulse	Slow pulse indicates processing of the main loop.
Fixe + Pulse	Fix ~10 sec + 500ms	Power-up 4G and connecting to mobile network. The status LED is ON while powering-up the 4G module and training the UART. Note : Pico LED will lit when registered !
Error	2	Cannot connect mobile network. See REPL output on Pico MicroUSB for more details.
Error	3	Unexpected error in treatment loop. See REPL output for more details

6.SMS command structure

SMS are used to send **keyword** command to the sms-control.

The keyword can be accompanied with optional parameters (up to two parameters, coma separated).

The valid formats are the following:

```
keyword
keyword,param1
keyword,param1,param2
```

with:

•**instruction** : max 10 chars

•**param1** : value of first parameter (max 20 chars)

•**param2** : value of second parameter (max 30 chars)

7.SMS Processing

Incoming SMS are treated one the time in their incoming order (incoming order may depend on the mobile network latency).

The incoming message is checked on authorized phone numbers:

• Check only when `is_auth()` is implemented otherwise everything is authorized.

• Returns « Denied ! » message on un-authorized access.

Copyright MC Hobby SPRL – All right reserved

Sms-control parse the incoming message on coma to extract the keyword, the first parameter, the second parameter. As parameters are optional, the message can be made of a single keyword.

The extracted keyword is compared to the internal list supported keywords (uppercase comparison).

•An « ERROR ! » message is returned when no keyword match is found! The error information is also shown on the REPL session.

•When a keyword match exists :

- □Execution is switched to the *handler function* attached to the keyword.
- □The « Done » message is sent back after *handler_function* execution.

If an error occured while executing the handler function then an « ERROR ! » message is return. The second message contains the error message.

Handler function can also send messages (as response to SMS or to any phone number). The messages are sent asynchronously by the main treatment loop. As a result, such message are sent after the "Done!" notification.

8.Output control

The board is fitted with 2 outputs (namely Output 1 and Output 2), each one controlling a relay. Each relay offers a normally open and normally close contact.

The control of the Outputs (the relays) are straightforward.

•Send SMS with "on1" or "on2" to activates the Output 1 or Output 2 (relay).

•Send SMS with "off1" or "off2" to deactivate the corresponding output.

9.Input control

The board is fitted with 2 opto-isolated 12~24V input (IN1 and IN2) as well as 2 additional 3,3V inputs (IN3 and IN4).

Opto-coupled entries have also been tested under 5V and looks to work properly.

9.1.Unmanaged IN1 and IN2

Signal on input is not managed by the sms-control software. However, user script can read then whenever it is necessary.

As exampple, ssending the SMS with "i**nfo**" will return messages containing various information including the IN1 & IN2 status.

Copyright MC Hobby SPRL – All right reserved

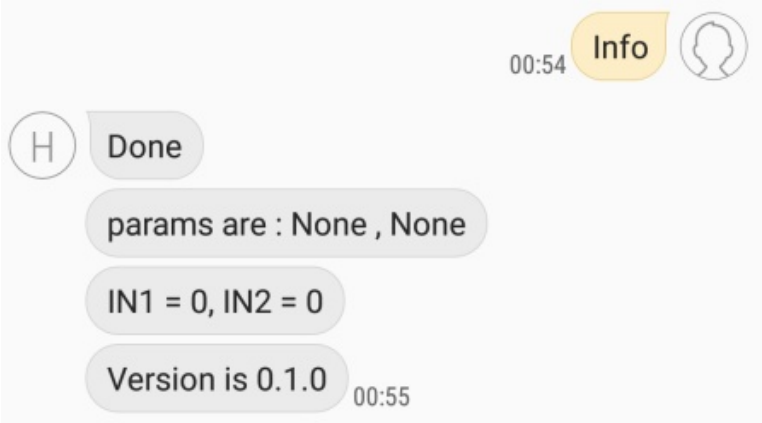


Fig : Response to info message

9.2.Un-initialized IN3 and IN4

Unlike gate-control, the `IN3` and `IN4` pins are not yet initialized as input. As consequence, the user script can define another usage for them (output or PWM).

The IN3 and IN4 constants (in `station.py`) identifies the pin number when defining alternate usage.

The IN3 or IN4 will automatically initialized as input when accessing the first time at the `BaseStation.in3` or `BasseStation.in4` properties.

10.Right management

There is no user management and no right management in the sms-control software.

Vocal call are always rejected.

By default, the SmsControl base class come with not restriction on the SMS sender.

However, the user code can define a list accepted phone numbers by surchaging the `is_auth()` method.

```
class MyApp( SmsControlApp ):
  def __init__( self ):
    super().__init__()
    # *** CREATE YOUR OBJECTs GHERE ***
    self.register_sms_handler( 'say', self.say_handler )

  def is_auth( self, phone_nr ):
    return phone_nr in ('+32444661122','+32444998877')

  def say_handler( self, msg, params ):
    # Send a response message
    self.register_message( notif_for=msg.phone,
            msg='I say %s!' %
            ("Hello" if params[0]==None else params[0])
            )
```

Copyright MC Hobby SPRL – All right reserved

11.Demonstration application

SMS-Control project come with an example script `examples/sms-control/main.py` implementing several keywords.

```
01: from smsctrl import SmsControlApp, HandlerError
02: from smsctrl import __version__ as smsctrl_version
03:
04: __version__ = '0.1.0' # My App Version
05:
06: class MyApp( SmsControlApp ):
07:   def __init__( self ):
08:     super().__init__()
09:     # *** CREATE YOUR OBJECTs GHERE ***
10:
11:     self.register_sms_handler( 'say'   , self.say_handler  )
12:     self.register_sms_handler( 'info'  , self.info_handler )
13:     self.register_sms_handler( 'error' , self.send_error_handler )
14:     self.register_sms_handler( 'punish', self.punish_handler )
15:     self.register_sms_handler( 'on1'   , self.relay_handler )
17:     self.register_sms_handler( 'off1'  , self.relay_handler )
17:     self.register_sms_handler( 'on2'   , self.relay_handler )
18:     self.register_sms_handler( 'off2'  , self.relay_handler )
19:
20:   # UNCOMMENT TO RESTRICT ACCESS to given phone number
21:   #def is_auth( self, phone_nr ):
22:   #  return phone_nr in ('+32444661122','+32444998877')
23:
24:   def update( self ):
25:     # *** PERFORM YOUR OBJECTs UPDATES HERE ***
26:     super().update()
27:
28:   # --- SMS Handlers ---
29:   def say_handler( self, msg, params ):
30:      self.register_message( notif_for=msg.phone, msg='I say %s!' %
              ("Hello" if params[0]==None else params[0]) )
31:
32:   def info_handler( self, msg, params ):
33:     self.register_message( msg.phone, "MyApp Version is %s" %
                                          __version__ )
34:     self.register_message( msg.phone, "SmsCtrl Version is %s" %
                                           smsctrl_version )
35:     self.register_message( msg.phone, "IN1 = %s, IN2 = %s" %
                    (self.base.in1.value(), self.base.in2.value() ) )
36:     self.register_message( msg.phone, "params are : %r , %r" %
                    (params[0],params[1]) ) # Params may be None
37:
38:
39:   def send_error_handler( self, msg, params ):
40:     raise HandlerError( 'This error message will be send back' )
41:
42:   def punish_handler( self, msg, params ):
43:     if not self.is_phone_nr( params[0] ):
44:       raise HandlerError( 'Invalid phone Number as parameter!' )
45:     self.register_message( params[0], "You have been punished!",
                              source_nr=msg.phone )
46:
47:   def relay_handler( self, msg, params ):
48:     """ This method is called for on1 & off1 """
49:     relay = None
50:     if '1' in msg.message:
51:       relay = self.base.rel1
52:     elif '2' in msg.message:
53:       relay = self.base.rel2
54:     else:
55:       raise HandleError( 'Invalid relay number!')
56:
57:     if 'ON' in msg.message.upper(): # msg.message is case-sensitive.
58:       relay.on()
59:     elif 'OFF' in msg.message.upper():
60:       relay.off()
61:     else:
62:       raise HandlerError( 'only "on" or "off" maybe used!' )
63:
64: app = MyApp()
```

Copyright MC Hobby SPRL – All right reserved

```
65: app.power_up()
66: app.run()
```

•Line 1 : Import of the `SmsControlApp` from `smsctrl` library. The class encapsulates the application logic. The `HandleError` class is used by user code to report errors.

•Line 2 : import of the `__version__` from `smsctrl` library under a new name.

•Line 3 : declaration of the `MyApp` application class inheriting fril the `SmsControlApp`. By overloading some methods, the user code will be able to implement its own keywords (say your « SMS commands).

•Lines 7 to 18 : Initializer method. Needs first to initialize the ancestor with a `super()` call. Next, the line 9 is where the user script should initialize its own objects (eg: temperature sensor). Finally, a series of `register_sms_handler()` associate the **keyword** together with the **method to call** when the keyword appears in the SMS.

•Lines 25 to 26 : Update method is called at each processing cycle. In the ancestor class this method updates the LED state and handle messages coming from 4G module. By overcharging the `update()` method, the user code can update its internal states, query the sensors, etc. Just remember to make a `super()` call to the ancestor!

•Lines 29 to 62 : implementation of the function handlers to respond to incoming SMS. Those handler will be detailed later.

•Lines 64 to 66 : Creates and start the customized application.

•Line 64 : Creates an instance of the `MyApp` customized application class.

•Line 65 : Call the `power_up()` method to power up and initialize the 4G module.

•Line 66 : Calling the `run()` method to start sms-control application and its treatment loops.

It is now time to inspect the various handler functions. Each implementation demonstrate a feature of the sms-control !

11.1.say_handler : get parameter and send message

The `say_handler` is put in place by the line 11 for the keyword « say ».

```
11:     self.register_sms_handler( 'say'    , self.say_handler  )
```

As shown in the code below:

•The handler is responsible to check the existence the required parameters (and their values).

•Parameters are available in the `params` list ! The first parameter is `params[0]`, the second is `params[1]` .

•Parameters are set to `None` when omitted

•Parameters are always `Str` (strings)

```
29:   def say_handler( self, msg, params ):
30:      self.register_message( notif_for=msg.phone,
                msg='I say %s!' %
               ("Hello" if params[0]==None else params[0])
         )
```

Copyright MC Hobby SPRL – All right reserved

The `register_message()` call is used to prepare a SMS message to be send. SMS are sent asynchronously after the handler finished its task.

Regarding register_message() :

•The first parameter (`notif_for`) is the recipient. In this case, the command sender then the phone number is grabbed from the original message (`msg.phone`).

•The second parameter (`msg`) is a string containing the message.

•The third parameter (`source_nr`, optional) is the phone number of a third party that will be integrated to the message. This is quite useful when repporting error messages to a master phone number, the message will then identify the third party phone causing the error.

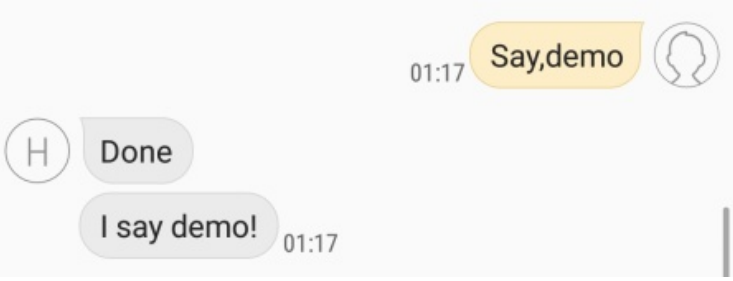


Fig : say keyword example

11.2.info_handler : send several message

The `info_handler` is put in place by the line 12 for the keyword « info ».

```
12:     self.register_sms_handler( 'info'  , self.info_handler )
```

As shown in the code below:

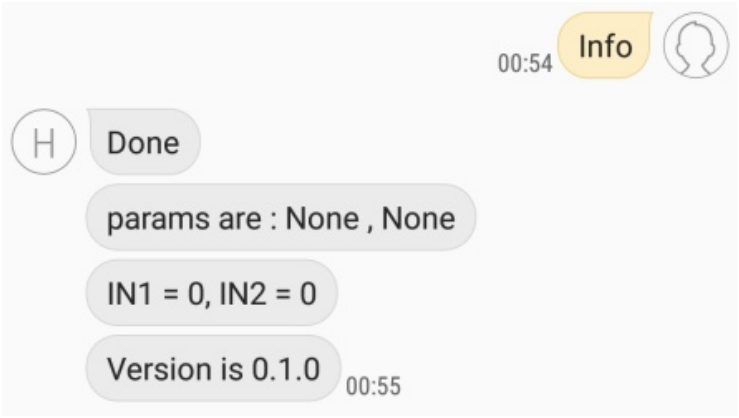
•Several messages can be send by the handler.

•The handler can easyly **access the 4G-base-station** feature. See call at linne 35. The `station.py` library is mentionned in the Resources section.

•The parameters value are also send back when mentionned otherwise the default value (`None`) is returned.

```
32:   def info_handler( self, msg, params ):
33:     self.register_message( msg.phone,
                "MyApp Version is %s" % __version__ )
34:     self.register_message( msg.phone,
                "SmsCtrl Version is %s" % smsctrl_version )
35:     self.register_message( msg.phone,
                 "IN1 = %s, IN2 = %s" % (
                     self.base.in1.value(),self.base.in2.value()
                                         ) )
36:     self.register_message( msg.phone,
                  "params are : %r , %r" % (params[0],params[1]) )
```

Copyright MC Hobby SPRL – All right reserved



11.3.error_handler : returning error to sender

The `error_handler` is put in place by the line 12 for the keyword « error ».

```
13:  self.register_sms_handler( 'error', self.send_error_handler )
```

Using the **HandlerError** exception is an handy way to returns error message to the sender and to interrupt the processing of the handler function.

```
39:   def send_error_handler( self, msg, params ):
40:     raise HandlerError('This error message will be send back')
```

Which returns the error message instead of “Done”.

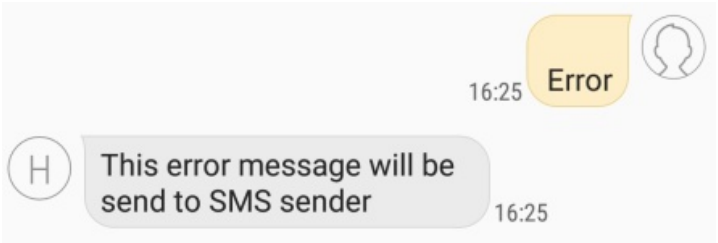


Fig: result of HandlerError exception

11.4.punish_handler : sending message to third person

The punish keyword is used to send a punishment sentence to a third party phone number (encoded as first parameter).

This keyword is used as follow :

```
punish,+32477445566
```

The `punish_handler` is put in place by the line 14 for the keyword « punish ».

```
14:    self.register_sms_handler( 'punish', self.punish_handler )
```

This time, the `register_message()` takes the first parameter as phone number to send the punishement sentence.

Notice that sender phone number (`msg.phone`) is mentioned as third parameter in the `register_message()` call.
This way, the recipient will receive the message containing the sentence and the punisher phone number (because the SMS message is issued by the 4G-Base-Station SIM card).

```
42:   def punish_handler( self, msg, params ):
```

Copyright MC Hobby SPRL – All right reserved

```
43:     if not self.is_phone_nr( params[0] ):
44:       raise HandlerError('Invalid phone Number as parameter!')
45:     self.register_message( params[0],
                       "You have been punished!",
                       source_nr=msg.phone )
```

when incorrect phonr number is used with punish keyword, the 4G-Base-Station will get an SMSError when sending the message. This error will be trapped by the 4G-Base-Station… error notification will nbot be returned to the sender.

11.5.relay_handler : controling relays with SMS

The `relay_handler` handler is attached to several keywords with the lines 15 to 18.

```
15:     self.register_sms_handler( 'on1'   , self.relay_handler )
17:     self.register_sms_handler( 'off1'  , self.relay_handler )
17:     self.register_sms_handler( 'on2'   , self.relay_handler )
18:     self.register_sms_handler( 'off2'  , self.relay_handler )
```

The code is quite simple and straightforward.

The keyword comparison is not case-sensitive. However, the message content (`msg.message`) contains the text as written by the sender (mixed case). Any treatment on the msg.message must take care about the case!

```
47:   def relay_handler( self, msg, params ):
48:     """ This method is called for on1 & off1 """
49:     relay = None
50:     if '1' in msg.message:
51:       relay = self.base.rel1
52:     elif '2' in msg.message:
53:       relay = self.base.rel2
54:     else:
55:       raise HandleError( 'Invalid relay number!')
56:
57:     if 'ON' in msg.message.upper():
58:       relay.on()
59:     elif 'OFF' in msg.message.upper():
60:       relay.off()
61:     else:
62:       raise HandlerError( 'only "on" or "off" maybe used!' )
```

11.6.Unexpected error

The `SmsControlApp` class can also manage un-expected error occuring in the handler function.

In the following example, the `send_error_handler()` function is slightly modified to cause a runtime error.

```
def send_error_handler( self, msg, params ):
  """ Just throw an error to the sender """
  # Use raise HandlerError( 'blablabla' ) to return an
  # error to the SMS sender
  i = 1
  y = 0
  print( i / y ) # will cause an error
```

Copyright MC Hobby SPRL – All right reserved

```
  raise HandlerError( 'This error message will be send to SMS
sender' )
```

As expected, a division by zero error will occurs in the `print()` statement before the `HandlerError` can even occurs.

The error is catched and returns the error message to the sender.

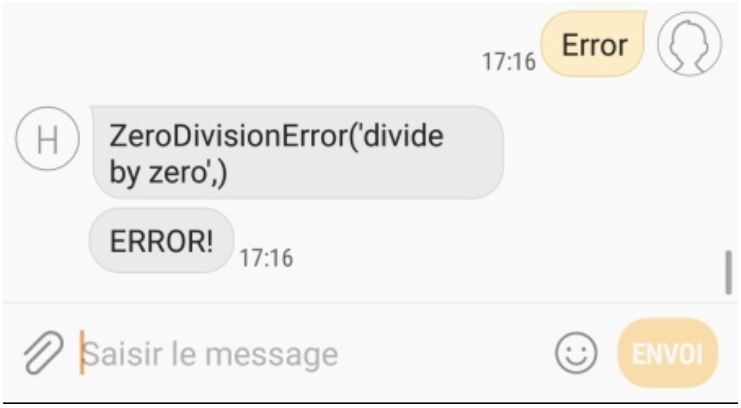


Fig : Capturing unexpected error

12.Debugging

Terminal software, MPRemote or Thonny IDE can be used to inspect the messages sent over the REPL session.

```
Connecting mobile network
connected!
Creating objects
Clearing stored SMS
Starting URC supervisor 0.1.0
----------------------------------------
1 notifications availables
[Notif(_time=1609459570, _type=3, source='+CMTI: "SM",1',
cargo=1)]
----------------------------------------
SMS received @ id 1
HandlerError for message Error from +32477668811
      HandlerError('This error message will be send to SMS
sender',)
```

The following REPL example shows the SMSError occuring because of an invalid phone number (used with the punish keyword).

```
----------------------------------------
1 notifications availables
[Notif(_time=1609459382, _type=3, source='+CMTI: "SM",1',
cargo=1)]
----------------------------------------
SMS received @ id 1
_loop: Unexpected error SMSError('AT+CMGS="+3244779"\r\n : +CMS
ERROR: unknown error : ERROR',)
_loop: Ignoring SMSError
```

13.Resources

4G-Base-Station kit

Copyright MC Hobby SPRL – All right reserved

https://shop.mchobby.be/fr/nouveaute/2888-4g-base-station-4g-controled-board-with-relays-and-optocoupled-input-micopython-ready-3232100028883.html

Code Repository

https://github.com/mchobby/micropython-4G-BASE-STATION

Station API Library

The `station.py` is the library file offering access to the 4G-Base-Station hardware. This file can be read from the reppository for more information.

https://github.com/mchobby/micropython-4G-BASE-STATION/blob/main/lib/station.py

Copyright MC Hobby SPRL – All right reserved