

Easy Script in Python 2.7

80378ST10106A Rev.5 – 2015-03-03



APPLICABILITY TABLE

| | SW Versions |
|-----------------------|---|
| HE910 Family | 12.00.xx6 |
| HE910 ¹ | |
| HE910-GA | |
| HE910-D | |
| HE910-EUR / HE910-EUD | |
| HE910-EUG / HE910-NAG | |
| HE910-NAR / HE910-NAD | |
| UE910 Family | |
| UE910-EUR / UE910-EUD | |
| UE910-NAR / UE910-NAD | |
| UL865 Family | |
| UL865-EUR / UL865-EUD | |
| UL865-NAR / UL865-NAD | |
| UL865-N3G | |
| GE910 Family | 13.00.xx7 |
| GE910-QUAD | |
| GE910-GNSS | |
| CE910 Family | 18.12.001 (Sprint) 18.12.022 (Verizon) 18.12.012 (Aeris) |
| C910-DUAL | |

Note: the present document covers the SW versions shown in the Applicability Table and may mention features which are not present or behave differently in previous SW versions

¹ HE910 is the “type name” of the products marketed as HE910-G & HE910-DG



USAGE AND DISCLOSURE RESTRICTIONS

License Agreements

The software described in this document is the property of Telit and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

Copyrighted Materials

Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of Telit

High Risk Materials

Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities"). Telit and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

Trademarks

TELIT and the Stylized T Logo are registered in Trademark Office. All other product or service names are the property of their respective owners.

Copyright © Telit Communications S.p.A.



Contents

| | |
|--|-----------|
| 1. Introduction | 11 |
| 1.1. Scope..... | 11 |
| 1.2. Audience..... | 11 |
| 1.3. Contact Information, Support | 11 |
| 1.4. Document Organization | 12 |
| 1.5. Text Conventions | 12 |
| 1.6. Related Documents..... | 13 |
| 2. Easy Script Extension - Python interpreter | 14 |
| 2.1. Overview | 14 |
| 2.2. Python 2.7.2 Copyright Notice | 16 |
| 2.3. Python | 16 |
| 2.4. Python implementation description..... | 16 |
| 2.5. Python supported features | 18 |
| 3. Python Script Operations..... | 19 |
| 3.1. Executing the Python script..... | 19 |
| 3.1.1. Write the Python script | 19 |
| 3.1.2. Compile the Python script | 19 |
| 3.1.3. Download the Python script | 19 |
| 3.1.4. Enable the Python script..... | 22 |
| 3.1.5. Run the Python script..... | 23 |
| 3.1.6. Read the Python script | 23 |
| 3.1.7. List saved Python scripts..... | 24 |
| 3.1.8. Delete the Python script | 24 |
| 3.2. Debug Python script | 25 |
| 4. Python Build-in Custom Modules..... | 26 |
| 4.1. MDM built-in module..... | 27 |
| 4.1.1. MDM.send(string, timeout)..... | 27 |
| 4.1.2. MDM.read()..... | 28 |
| 4.1.3. MDM.sendbyte(byte, timeout)..... | 28 |
| 4.1.4. MDM.readbyte() | 28 |



- 4.1.5. MDM.sendavail() 29
- 4.1.6. MDM.getDCD()..... 29
- 4.1.7. MDM.getCTS()..... 29
- 4.1.8. MDM.getDSR() 30
- 4.1.9. MDM.getRI()..... 30
- 4.1.10. MDM.setRTS(RTS_value)..... 30
- 4.1.11. MDM.setDTR(DTR_value) 30
- 4.2. MDM2 built-in module..... 32
 - 4.2.1. MDM2.send(string, timeout) 32
 - 4.2.2. MDM2.read() 33
 - 4.2.3. MDM2.sendbyte(byte, timeout)..... 33
 - 4.2.4. MDM2.readbyte()..... 34
 - 4.2.5. MDM2.sendavail() 34
 - 4.2.6. MDM2.getDCD() 34
 - 4.2.7. MDM2.getCTS() 34
 - 4.2.8. MDM2.getDSR()..... 35
 - 4.2.9. MDM2.getRI() 35
 - 4.2.10. MDM2.setRTS(RTS_value) 35
 - 4.2.11. MDM2.setDTR(DTR_value)..... 35
- 4.3. SER built-in module 37
 - 4.3.1. SER.send(string, <timeout>) 37
 - 4.3.2. SER.read() 38
 - 4.3.3. SER.sendbyte(byte, <timeout>)..... 38
 - 4.3.4. SER.readbyte()..... 39
 - 4.3.5. SER.sendavail()..... 39
 - 4.3.6. SER.set_speed(speed, <char format>, <flow control>)..... 39
 - 4.3.7. SER.setDCD(DCD_value) 40
 - 4.3.8. SER.setCTS(CTS_value) 40
 - 4.3.9. SER.setDSR(DSR_value) 40
 - 4.3.10. SER.setRI(RI_value) 41
 - 4.3.11. SER.getRTS() 41
 - 4.3.12. SER.getDTR()..... 41
- 4.4. SER2 built-in module 42
 - 4.4.1. SER2.send(string)..... 42
 - 4.4.2. SER2.read() 43
 - 4.4.3. SER2.sendbyte(byte) 43
 - 4.4.4. SER2.readbyte() 43



| | | |
|---------|--|----|
| 4.4.5. | SER2.sendavail() | 44 |
| 4.4.6. | SER2.set_speed(speed, <char format>) | 44 |
| 4.5. | GPIO built-in module | 45 |
| 4.5.1. | GPIO.setIOvalue(GPIOnumber, value) | 45 |
| 4.5.2. | GPIO.getIOvalue(GPIOnumber) | 45 |
| 4.5.3. | GPIO.setIOdir(GPIOnumber, value, direction) | 46 |
| 4.5.4. | GPIO.getIOdir(GPIOnumber) | 46 |
| 4.5.5. | GPIO.getADC(adcNumber) | 47 |
| 4.5.6. | GPIO.setDAC(enable, value) | 47 |
| 4.5.7. | GPIO.setVAUX(vauxNumber, enable) | 48 |
| 4.5.8. | GPIO.getAXE() | 48 |
| 4.5.9. | GPIO.setSLED(status, onDuration, offDuration) | 48 |
| 4.5.10. | GPIO.getCBC() | 49 |
| 4.6. | GPS built-in module | 50 |
| 4.6.1. | GPS.powerOnOff(newStatus) | 50 |
| 4.6.2. | GPS.getPowerOnOff() | 50 |
| 4.6.3. | GPS.resetMode(mode) | 50 |
| 4.6.4. | GPS.getActualPosition() | 51 |
| 4.6.5. | GPS.powerSavingMode(mode, pushToFixPeriod) | 51 |
| 4.6.6. | GPS.powerSavingWakeUp() | 52 |
| 4.6.7. | GPS.getLastGGA() | 52 |
| 4.6.8. | GPS.getLastGLL() | 52 |
| 4.6.9. | GPS.getLastGSA() | 52 |
| 4.6.10. | GPS.getLastGSV() | 53 |
| 4.6.11. | GPS.getLastRMC() | 53 |
| 4.6.12. | GPS.getLastVTG() | 53 |
| 4.6.13. | GPS.getPosition() | 54 |
| 4.7. | IIC built-in module | 55 |
| 4.7.1. | IIC.new(SDA_pin, SCL_pin, ADDR) | 55 |
| 4.7.2. | init() | 56 |
| 4.7.3. | readwrite(string, <read_len>) | 56 |
| 4.8. | SPI built-in module | 58 |
| 4.8.1. | SPI.new(SCLK_pin, MOSI_pin, MISO_pin, <SS0>,<SS1>,...,<SS7>) | 58 |
| 4.8.2. | init(CPOL, CPHA, <SSPOL>, <SS>) | 59 |
| 4.8.3. | readwrite(string, <read_len>, <SS>) | 59 |
| 4.9. | MOD built-in module | 61 |



- 4.9.1. MOD.watchdogEnable(timeout) 61
- 4.9.2. MOD.watchdogReset() 61
- 4.9.3. MOD.watchdogDisable() 62
- 4.9.4. MOD.powerSaving(timeout) 62
- 4.10. USB0 built-in module 63
 - 4.10.1. USB0.send(string, <timeout>) 63
 - 4.10.2. USB0.read() 64
 - 4.10.3. USB0.sendbyte(byte, <timeout>) 64
 - 4.10.4. USB0.readbyte() 64
 - 4.10.5. USB0.sendavail() 65
- 5. Python standard functions..... 66**
 - 5.1. Technical characteristics 66
 - 5.1.1. General 66
 - 5.2. Python supported features 67
 - 5.2.1. Operators, statements, functions 67
 - 5.2.2. Built-in Functions 68
 - 5.2.3. Built-in Constants 69
 - 5.2.4. Truth Value Testing 70
 - 5.2.5. Boolean Operations 70
 - 5.2.6. Comparisons 70
 - 5.2.7. Numeric Types: Integer, Long Integer and Floating Point 70
 - 5.2.8. Numeric Types: Integer and Long Integer 71
 - 5.2.9. Numeric Types: Floating Point 71
 - 5.2.10. Numeric Types: Complex 71
 - 5.2.11. Iterator Types 71
 - 5.2.12. Generator Types 71
 - 5.2.13. Sequence Types: String, List, Tuple, bytearray, Buffer and xrange 72
 - 5.2.14. Sequence Types: Unicode 72
 - 5.2.15. Sequence Types: String 72
 - 5.2.16. Mutable Sequence Types: List and bytearray 73
 - 5.2.17. Set Types: Set and Frozenset 74
 - 5.2.18. Mapping Types: Dictionary 75
 - 5.2.19. File Objects 75
 - 5.2.20. Memoryview Objects 78
 - 5.2.21. Module Objects 78
 - 5.2.22. Classes and Class Instances 78



| | | |
|-----------|--|-----------|
| 5.2.23. | Function Objects | 78 |
| 5.2.24. | Method Objects | 78 |
| 5.2.25. | Code objects | 78 |
| 5.2.26. | Type Objects | 78 |
| 5.2.27. | Null Object | 79 |
| 5.2.28. | Ellipsis Object | 79 |
| 5.2.29. | NotImplemented Object | 79 |
| 5.2.30. | Internal Types: Frame Objects | 79 |
| 5.2.31. | Internal Types: Traceback Objects | 79 |
| 5.2.32. | Slice Objects | 79 |
| 5.2.33. | Built-in Exceptions | 80 |
| 5.2.34. | Built-in Modules: marshal | 81 |
| 5.2.35. | Built-in Modules: imp | 81 |
| 5.2.36. | Built-in Modules: _ast | 81 |
| 5.2.37. | Built-in Modules: __main__ | 81 |
| 5.2.38. | Built-in Modules: __builtin__ | 81 |
| 5.2.39. | Built-in Modules: sys | 82 |
| 5.2.40. | Built-in Modules: exceptions | 83 |
| 5.2.41. | Built-in Modules: gc | 83 |
| 5.2.42. | Built-in Modules: _warnings | 84 |
| 5.2.43. | Built-in Modules: _md5 | 84 |
| 5.2.44. | Built-in Modules: binascii | 84 |
| 5.2.45. | Built-in Modules: _sre | 84 |
| 5.2.46. | Built-in Modules: _weakref | 84 |
| 5.2.47. | Built-in Modules: _syntable | 84 |
| 5.2.48. | Built-in Modules: _functools | 85 |
| 5.2.49. | Built-in Modules: _socket | 85 |
| 5.2.50. | Built-in Modules: time | 87 |
| 5.2.51. | Built-in Modules: posix | 88 |
| 5.2.52. | Built-in Modules: thread | 88 |
| 5.2.53. | Built-in Modules: signal | 89 |
| 5.2.54. | Built-in Modules: errno | 89 |
| 5.2.55. | Built-in Modules: cStringIO | 89 |
| 5.2.56. | Built-in Modules: math | 90 |
| 5.2.57. | Library Modules | 91 |
| 6. | Python non standard functions | 92 |



- 6.1. `_socket` non standard functions..... 92
 - 6.1.1. Non standard socket option flag: `SO_CONTEXTID` 92
- 6.2. `signal` non standard functions 93
 - 6.2.1. Non standard signal: `SIGMDM` 93
 - 6.2.2. Non standard signal: `SIGMDM2` 93
- 7. Python Notes 94**
 - 7.1. Memory Limits 94
 - 7.2. Other Limits..... 94
- 8. Python Script Emulation on PC 95**
 - 8.1. Executing the Python script on PC..... 95
 - 8.1.1. Install Python 95
 - 8.1.2. Install optional serial package..... 95
 - 8.1.3. Copy Python modules 95
 - 8.1.4. Run the Python script..... 96
- 9. Document History 97**



1. Introduction

1.1. Scope

Aim of this document is to give an overview of the Easy Script Extension feature, which lets the developer to drive the modem *internally*, writing the controlling application directly in a high level language such as Python.

1.2. Audience

This document is intended for Telit customers developing functionalities on their applications.

1.3. Contact Information, Support

For general contact, technical support, to report documentation errors and to order manuals, contact Telit Technical Support Center (TTSC) at:

TS-EMEA@telit.com
TS-NORTHAMERICA@telit.com
TS-LATINAMERICA@telit.com
TS-APAC@telit.com

Alternatively, use:

<http://www.telit.com/en/products/technical-support-center/contact.php>

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:

<http://www.telit.com>

To register for product news and announcements or for product questions contact Telit Technical Support Center (TTSC).

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.

Telit appreciates feedback from the users of our information.



1.4. Document Organization

This document contains the following chapters:

[Chapter 1: “Introduction”](#) provides a scope for this document, target audience, contact and support information, and text conventions.

[Chapter 2: “Easy Script Extension – Python interpreter”](#) gives a broad overview about the extension.

[Chapter 3: “Python script operations”](#) deals with the execution of the scripts operatively.

[Chapter 4: “Python built-in custom modules”](#) explains in detail the single custom built-in modules.

[Chapter 5: “Python standard functions”](#) provides a description of Python language supported features.

[Chapter 6: “Python non standard functions”](#) provides a description of non standard functions added to Python language.

[Chapter 7: “Python notes”](#) deals with some Python limits that should be considered while developing scripts.

1.5. Text Conventions



Danger – This information MUST be followed or catastrophic equipment failure or bodily injury may occur.



Caution or Warning – Alerts the user to important points about integrating the module, if these points are not followed, the module and end user equipment may fail or malfunction.



Tip or Information – Provides advice and suggestions that may be useful when integrating the module.

All dates are in ISO 8601 format, i.e. YYYY-MM-DD.



1.6. Related Documents

- HE910 AT Commands Reference Guide, 80378ST10091A (12.xx.xxx)
- HE Family Ports Arrangements User Guide, 1w0300971 (12.xx.xxx)
- AT Commands Reference Guide, 80000ST10025A (13.xx.xxx)
- Telit Modules Software User Guide, 1w0300784 (12.xx.xxx, 13.xx.xxx)
- CE910 Family AT Commands Reference Guide, 80399ST10110A (18.xx.xxx)
- CE910 Family Software User Guide, 1w0301011 (18.xx.xxx)

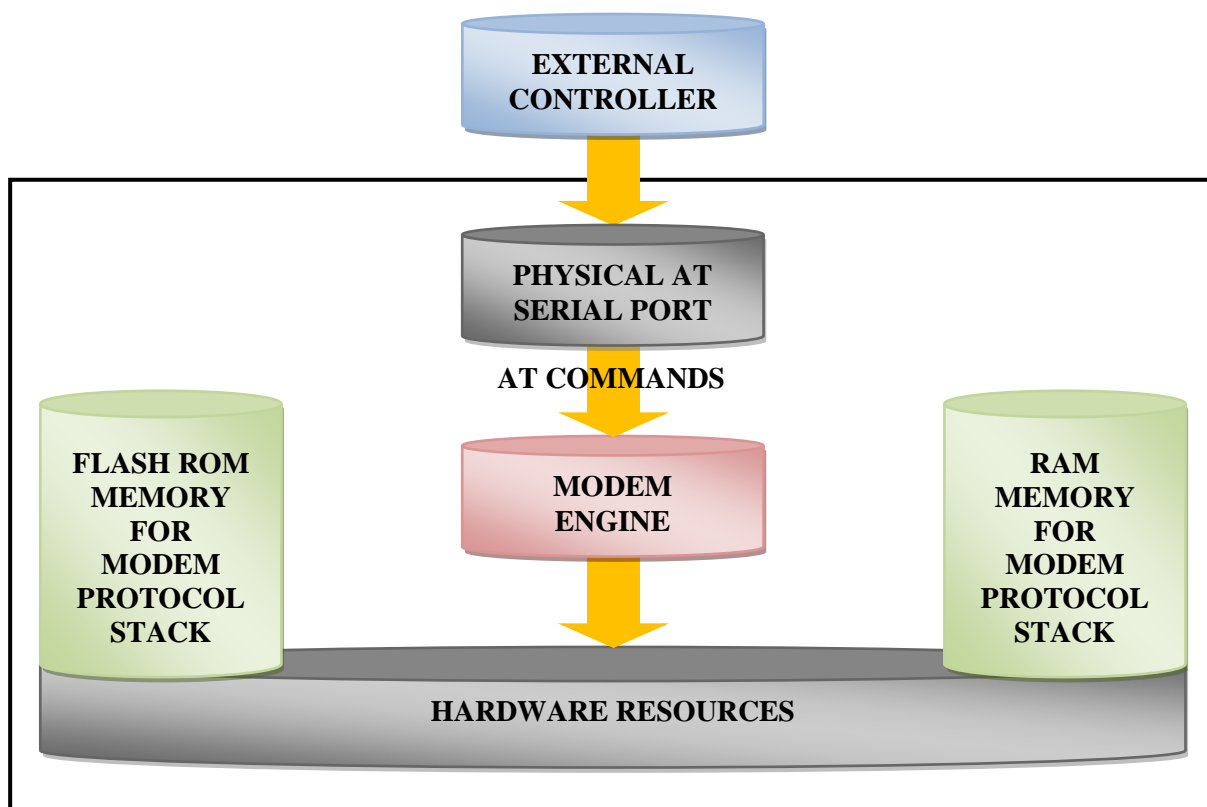


2. Easy Script Extension - Python interpreter

2.1. Overview

The Easy Script Extension is a feature that allows driving the modem *internally*, writing the controlling application directly in the Python high level language. A typical application usually consists of a microcontroller managing data transfer and several I/O pins on the module through the AT command interface.

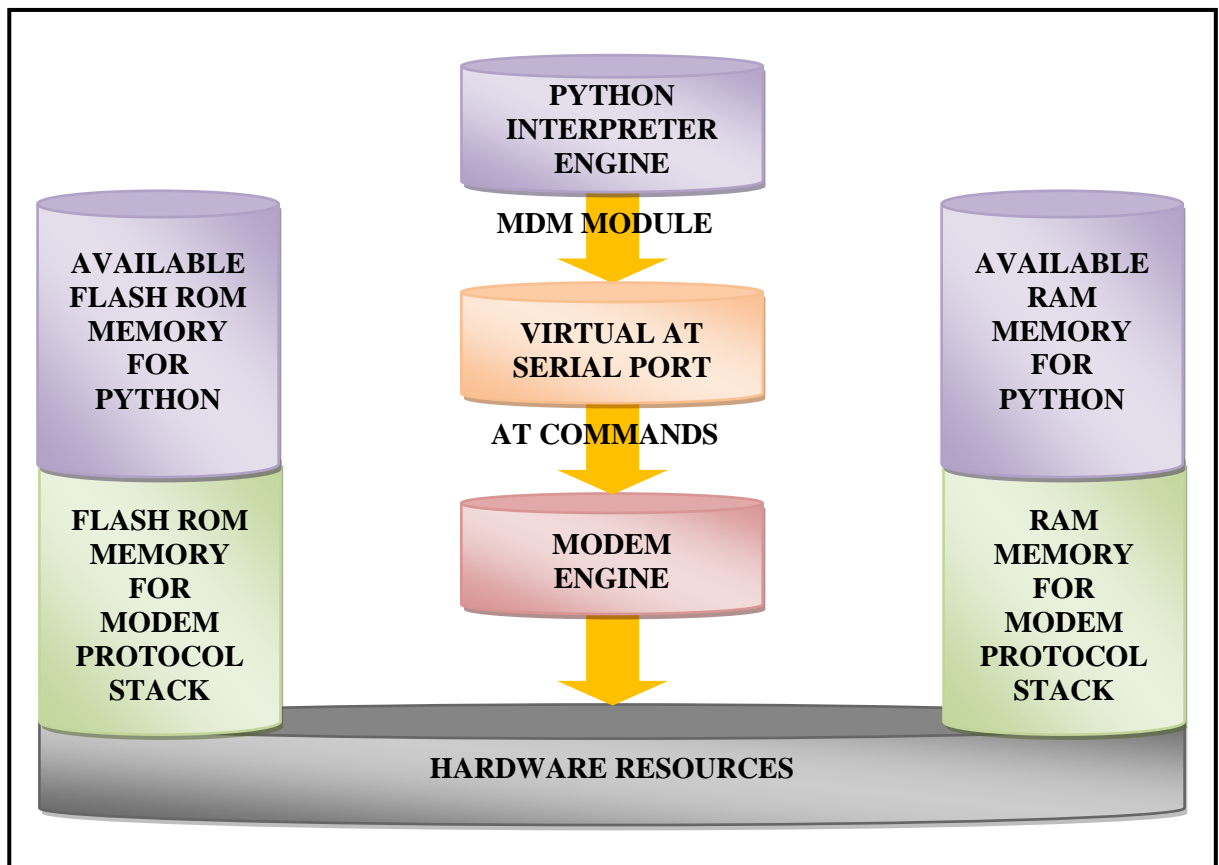
A schematic of such a configuration can be the following:

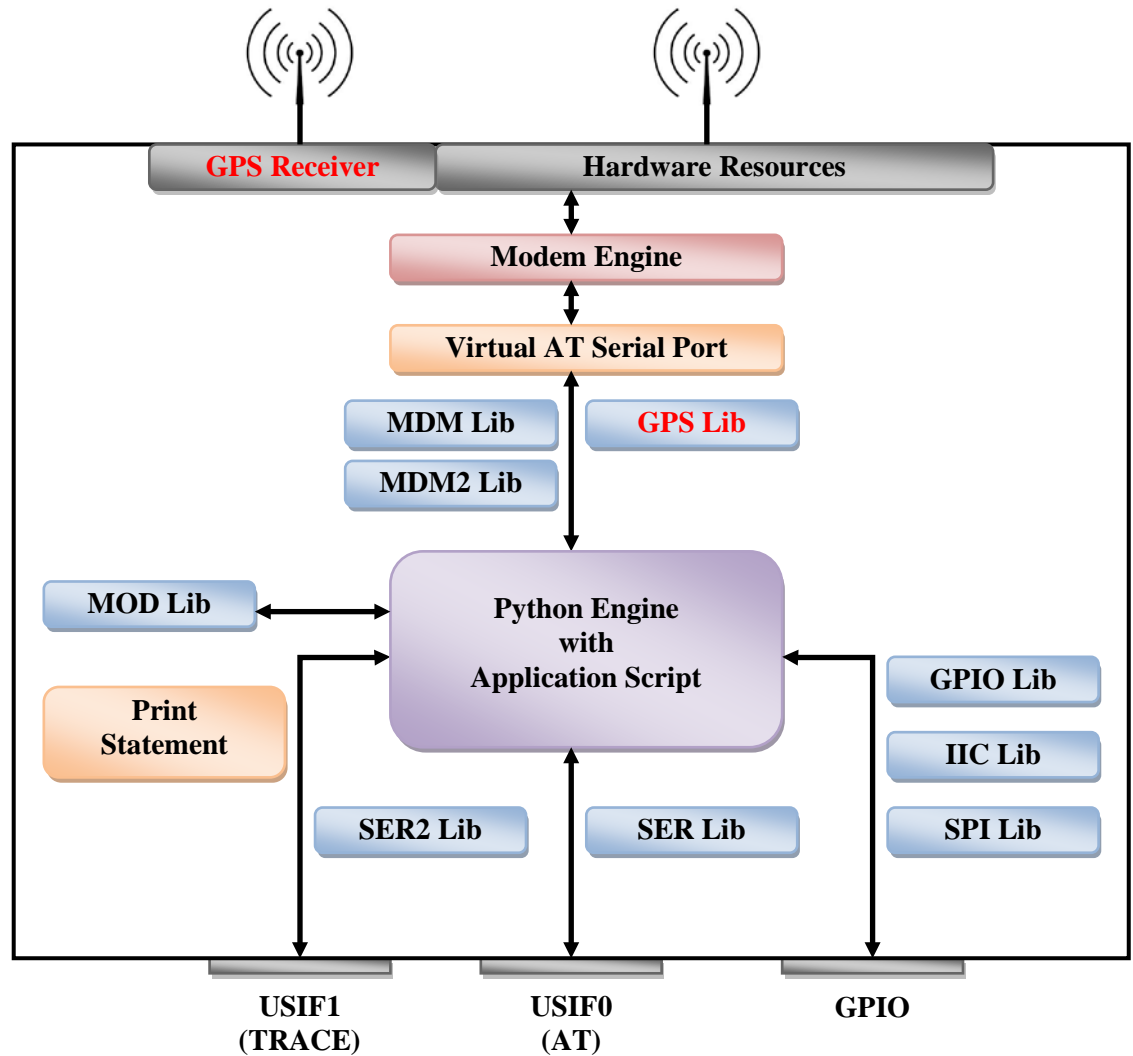


The Easy Script Extension functionality lets the developer to get rid of the external controller and further simplify the programmed sequence of operations. The equipped Python version features the following:

- Python script interpreter engine version 2.7.2
- 2 MB of Non Volatile Memory space for user scripts and data files (12.xx.xxx, 13.xx.xxx and 18.11.004)
- 2 MB RAM available for the Python engine(12.xx.xxx, 13.xx.xxx and 18.11.004)

The following depicts a schematic of this approach:





NOTE:

Antenna GPS, GPS receiver and GPS Library are available exclusively for the GPS modules.

- The MDM interface** is the most important one. It allows the Python script to send AT commands, receive responses and unsolicited indications, send data to the network and receive data from the network during connections. It is quite similar to the regular serial port interface on the **Telit module**. The only difference being that this interface is an internal software bridge between Python and module internal AT command handling engine, and not a physical serial port. All AT commands working on the **Telit module** are working with this software interface as well. Some of them have no meaning for this interface, such as those regarding serial port settings, while others, such as the concept of hardware flow control, keeps its meaning but it's managed internally.



3. Python Script Operations

3.1. Executing the Python script

The steps required to have a script executing by the Python engine on the **Telit module** are:

- write the Python script, in case splitting it in more than one file;
- optionally compile the Python script;
- download the, optionally compiled, Python script into the module NVM;
- enable the Python script;
- run the Python script.

3.1.1. Write the Python script

A Python script is a simple text file with .py extension, it can be written with any text editor. In case of large application it is useful to split it in more than one file.

The following is the "Hello Word" short Python script example that sends the simplest AT command to the AT command parser, immediately reads response and then ends.

```
import MDM
print 'Hello World!'
result = MDM.send('AT\r', 0)
print result
c = MDM.read()
print c
```

3.1.2. Compile the Python script

It is an optional operation.

Compiling the Python script on PC before downloading to module saves time at Python script execution start.

The following procedure allows to compile .py Python files into .pyc Python compiled files:

- install Python version 2.7.2 on your PC (as an example in directory C:\Python27)
- use compileall.py library Python script on your PC to compile all .py files in your working directory (as an example in directory C:\pytemp)

```
cd C:\Python27
python -v -S .\Lib\compileall.py -l -f C:\pytemp
```

3.1.3. Download the Python script

Use the following AT command:

```
AT#WSCRIPT="<script_name>",<size>[,<know-how>]
```

where

<script_name>: file name

<size>: file size (number of bytes)



<know-how>: (optional) know how protection, 1 = on, 0 = off (default)

The script, the compiled script, any text or binary file, can be downloaded to the module using the AT#WSCRIPT command. In order to guarantee your company know-how, you have the option to hide the script text so that the AT#RSCRIPT command does not return the text of the script and keeps it "confidential", you can see the name of the script only using the AT#LSCRIPT command.

In order to download the, optionally compiled, Python script you have to choose a name for your script on the module, taking care of the following:

- the extension for scripts is .py;
- the extension for compiled scripts is .pyc;
- any or no extension is permitted for generic text or binary file;
- the maximum file name length allowed is 16 characters;
- file names are case sensitive.

Then you have to find out the exact size in bytes of the script or compiled script, or generic text or binary file. For example, right clicking on the file and selecting "size" in "properties" (attention: this is different from selecting "size on disk").

It is important for large files, compared to module serial port buffer size of 4096 bytes, to activate hardware flow control on your terminal emulator.

It is possible to overwrite an existing file, there is no need to delete old one first.

When using standard Windows terminal emulator *Hyper Terminal* refer to "Send Text file" function.

In *Hyper Terminal* application select "Hardware" flow control in serial settings.

In ASCII Setup set "Send line ends with line feeds" and "Append line feeds to incoming line ends".

Type for example

```
AT#WSCRIPT="a.py",110
```

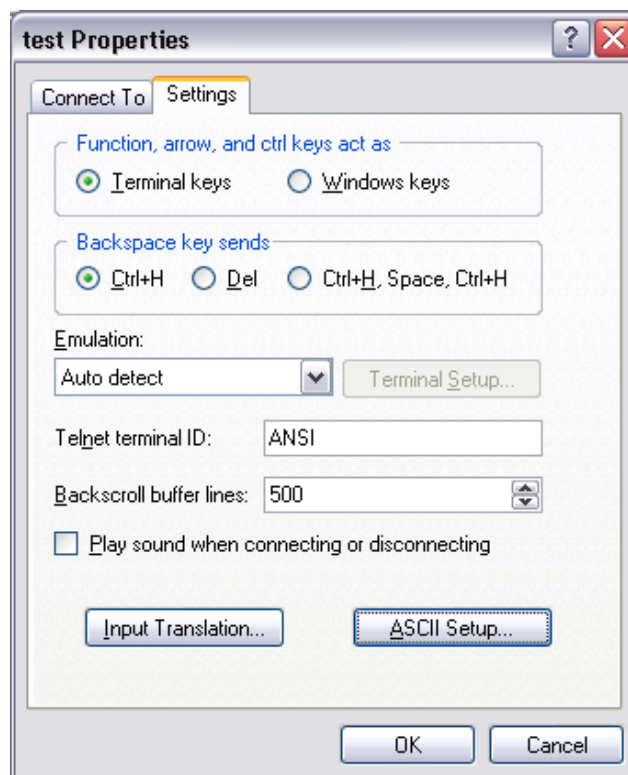
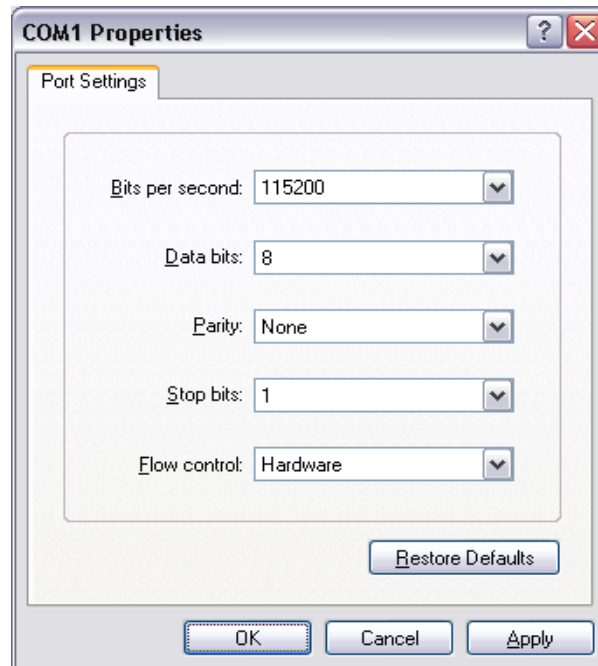
wait for the prompt

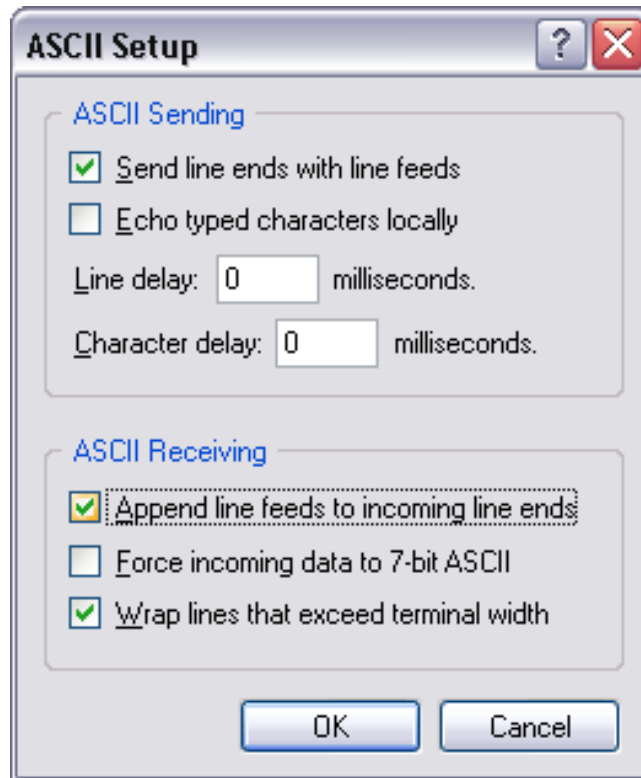
```
>>>
```

and use "Send Text file" selecting the proper file.

Wait for the result: OK or ERROR.







3.1.4. Enable the Python script

Use the following AT command:

```
AT#ESCRIP="<script_name>"
```

where

<script_name>: script name to be executed

Using the AT#ESCRIP command select the Python script which will be executed from the next start-up and in every future start-up or after AT#EXECSCR command. It can be either a .py Python script or a .pyc compiled Python script. In case the application consists of more than one file only the main script must be enabled for execution.

When selecting the script to enable between the ones downloaded to the module:

AT#LSCRIPT - can help checking the names of the scripts;

AT#ESCRIP? - can help checking the name of the script that is enabled at the moment.



NOTE:

There is no error return value for non existing script name in the module memory typed in command AT#ESCRIP. For this reason it's recommended to double check the name of the script that you want to execute. On the other hand this characteristic permits additional possibilities like enabling the Python script before downloading it on the module or not having to enable the same script name every time the script has been changed, deleted and replaced with another script but with the same name.



Type for example

```
AT#ESCRIP="a.py"
```

Wait for the result: OK or ERROR.

3.1.5. Run the Python script

In default configuration the Python script downloaded to module and enabled is executed at every module power on if the DTR line is sensed LOW (2.8V at the module DTR pin - RS232 signals are inverted) on USIF0 at start-up (this means that no AT command interface is connected to the modem serial port), and if the script name enabled matches with one of the script names of the scripts downloaded.

For example the Python script correctly downloaded and enabled is executed when the module is powered on and the serial cable was previously disconnected from USIF0.

In order to block Python script execution and control the module through the AT command interface on the serial port (for example to update locally a new script) the module shall be powered on with the DTR line HIGH (0V at the module DTR pin). In this condition the Python engine is not started and the script is not executed.

The real execution of the .py Python script is delayed from the power on due to the time needed by Python to parse the script. The larger is the script, the longer is this delay. The execution of .pyc compiled Python script is faster because there is no parsing of the script, just reading the file from NVM.

Another possibility is to run the correctly downloaded and enabled Python script from terminal emulator using the following AT command:

```
AT#EXECSCR
```

Another possibility is to select a second way of executing the Python script at every module power on using the following AT command:

```
AT#STARTMODESCR=<script_start_mode>[,<script_start_to>]
```

where

<script_start_mode>: mode (default 0)

<script_start_to>: timeout (default 10)

If the mode is set to 1 than the Python script downloaded to module and enabled is executed at every module power on if the user does not send any AT command on the serial port for the time interval specified in <script_start_to> parameter (default 10s).

3.1.6. Read the Python script

Use the following AT command:

```
AT#RSCRIPT="<script_name>"
```

where

<script_name>: file name

Using the command AT#RSCRIPT read a saved script, compiled script, generic text or binary file. If know-how protection is activated than AT#RSCRIPT will return only OK: no Python



script source code will be returned. In this way no one will be able to read your Python script from the module serial port.

It is important for large files, compared to PC serial port buffer size, to activate hardware flow control on your terminal emulator.

Type for example

```
AT#RSCRIPT="a.py
```

Wait for the prompt

```
<<<
```

Receive file data and wait for the result: OK or ERROR.

3.1.7. List saved Python scripts

Use the following AT command:

```
AT#LSCRIPT
```

This command shows the list of the file names currently saved, their size and the number of free bytes in memory.

3.1.8. Delete the Python script

Use the following AT command:

```
AT#DSCRIPT="<script_name>"
```

where

```
<script_name>: file name
```

Using the AT#DSCRIPT command delete from the module memory a saved script, compiled script, generic text or binary file.

Type for example

```
AT#DSCRIPT="a.py"
```

Wait for the result: OK or ERROR.



3.2. Debug Python script

The debug of the running Python script can be done on the second serial port USIF1 of the module.

For product versions 12.xx.xxx the following configuration must be set.
Use the following AT command:

```
AT#PORTCFG=3
```

to configure ports properly.

Reboot module to make #PORTCFG configuration working.

In #PORTCFG: 3 configuration Python standard output and standard error, including print statements, are redirected to USIF1 at 115200.

In this configuration AT2 parser instance on USIF1 is not available.

In this configuration Python scripts can be debugged with or without USB inserted.

For product versions 13.xx.xxx there is no need of configuration.

Default #PORTCFG: 0 is the proper configuration.

In #PORTCFG: 0 configuration Python standard output and standard error, including print statements, are redirected to USIF1 at 115200.

For product version 18.11.004 there is no need of configuration.

AT #PORTCFG is not available in version 18.11.004.

In default configuration Python standard output and standard error, including print statements, are redirected to USIF1 at 115200.

For every product version (12.xx.xxx, 13.xx.xxx and 18.11.004) proceed in the following way.
Connect to the second module serial port USIF1 at 115200.

Collect Python standard output and standard error:

- Python information messages (for example the version);
- Python error information;
- Results of all Python “print” statements.



4.1. MDM built-in module

The MDM built-in module is the interface between Python and the module AT commands parser engine.

You need to use the MDM built-in module if you want to send AT commands and data from the Python script to the network and receive responses and data from the network during connections.

In the default configuration, echo (ATE0) is disabled and the response format of result codes is set to verbose (ATV1).

If you want to use MDM built-in module you need to import it:

```
import MDM
```

then you can use its methods as in the following example:

```
a = MDM.send('AT', 0)
b = MDM.sendbyte(0x0d, 0)
c = MDM.read()
```

which sends 'AT' and reads 'OK' response.

More details about MDM built-in module methods can be found in the following paragraphs.

4.1.1. MDM.send(string, timeout)

This command sends a string to the AT command interface.

The first input parameter *string* is a Python string to send to the AT command interface.

The second input parameter *timeout* is a Python integer, measured in 1/10s, and is important in online mode when flow control is activated. In fact it represents the maximum time to wait for the string to be sent to the AT command interface when buffer is full and flow control blocks further data. The timeout range is (0 ÷ 32767).

This method returns immediately after the string has been sent to the AT interface or after the timeout period if the whole string could not be sent to the AT interface. The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
a = MDM.send('AT', 5)
```

sends the string 'AT' to AT command handling, waiting up to 0.5 s, assigning the return value to a.



NOTE:

The buffer available for the MDM.send command is 32768 bytes for product versions 12.xx.xxx or 4096 bytes for product versions 13.xx.xxx and 18.11.004.



sets DTR to OFF in AT command handling.



4.2. MDM2 built-in module

MDM2 built-in module is the second interface between Python and the module internal AT command handling. It is used to send AT commands from Python script to module and receive AT responses from module to Python script when the classic MDM built-in module is already in use.

Though MDM2 built-in module is independent from activation of CMUX protocol, it works on the second instance of AT parser in the same way the second CMUX port does. So the rules on AT commands that apply on the first and second CMUX ports (AT parser instances) apply on MDM and MDM2 as well.

See "AT Commands Reference Guide" and "CMUX User Guide" for details on availability of AT commands on all instances and for the rules on parallel execution of AT commands on two instances.

In the default configuration, echo (ATE0) is disabled and the response format of result codes is set to verbose (ATV1).

If you want to use MDM2 built-in module you need to import it:

```
import MDM2
```

than you can use its methods like in the following example:

```
a = MDM2.send('AT', 0)
b = MDM2.sendbyte(0x0d, 0)
c = MDM2.read()
```

which sends 'AT' and reads 'OK' response.

More details about MDM2 built-in module methods can be found in the following paragraphs.

4.2.1. MDM2.send(string, timeout)

This command sends a string to the AT command interface.

The first input parameter *string* is a Python string to send to the AT command interface.

The second input parameter *timeout* is a Python integer, measured in 1/10s, and is important in online mode when flow control is activated. In fact it represents the maximum time to wait for the string to be sent to the AT command interface when buffer is full and flow control blocks further data. The timeout range is (0 ÷ 32767).

This method returns immediately after the string has been sent to the AT interface or after the timeout period if the whole string could not be sent to the AT interface. The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
a = MDM2.send('AT', 5)
```

sends string 'AT' to AT command handling, possibly waiting for 0.5 s, assigning the return value to a.





NOTE:

The buffer available for MDM2.send command is 32768 bytes for product versions 12.xx.xxx or 4096 bytes for product versions 13.xx.xxx and 18.11.004.

4.2.2. MDM2.read()

This command receives a string from the AT command interface.

It has no input parameter.

The return value is a Python string which contains the data received and stored in buffer at the moment of command execution. The value might be empty if no data is received.

Example:

```
a = MDM2.read()
```

receives a string from AT command handling, assigning the return value to a.



NOTE:

The buffer available for MDM2.read command is 32768 bytes for product versions 12.xx.xxx or 4096 bytes for product versions 13.xx.xxx and 18.11.004. The maximum number of bytes returned by each MDM2.read is 1023.



NOTE:

It is up to Python script to keep empty MDM2.read buffer.

4.2.3. MDM2.sendbyte(byte, timeout)

This command sends a byte to the AT command interface.

The first input parameter *byte* can be zero or any Python byte to send to the AT command interface.

The second input parameter *timeout* is a Python integer, measured in 1/10s, and is important in online mode when flow control is activated. In fact it represents the maximum time to wait for the byte to be sent to the AT command interface when buffer is full and flow control blocks further data. The timeout range is (0 ÷ 32767).

This method returns immediately after the byte has been sent to the AT interface or after the timeout period if the byte could not be sent to the AT interface. The return value is a Python integer which is -1 if the timeout expired, 1 otherwise.

Example:

```
b = MDM2.sendbyte(0x0d, 0)
```

sends byte 0x0d, that is <CR>, to AT command handling, without waiting, assigning the return value to b.



Example:

```
cts = MDM2.getCTS()
```

gets CTS from AT command handling, assigning the return value to cts.

4.2.8. **MDM2.getDSR()**

This command gets Data Set Ready (DSR) from the AT command interface.

It has no input parameter.

The return value is a Python integer which is either 0 if DSR is set to OFF or 1 if DSR is set to ON.

Example:

```
dsr = MDM2.getDSR()
```

gets DSR from AT command handling, assigning the return value to dsr.

4.2.9. **MDM2.getRI()**

This command gets Ring Indicator (RI) from the AT command interface.

It has no input parameter.

The return value is a Python integer which is 0 if RI is set to OFF or 1 if RI is set to ON.

Example:

```
ri = MDM2.getRI()
```

gets RI from AT command handling, assigning the return value to ri.

4.2.10. **MDM2.setRTS(RTS_value)**

This command sets Request to Send (RTS) in the AT command interface.

The input parameter *RTS_value* is a Python integer which is 0 if setting RTS to set to OFF or 1 if setting RTS to set to ON.

No return value.

Example:

```
MDM2.setRTS(1)
```

sets RTS to ON in AT command handling.

4.2.11. **MDM2.setDTR(DTR_value)**

This command sets Data Terminal Ready (DTR) in the AT command interface.



4.3.4. **SER.readbyte()**

This command receives a byte from the serial port TXD/RXD.

It has no input parameter.

The return value is a Python integer which is the byte value received and stored in buffer at the moment of command execution or is -1 if no data is received. The return value can also be zero.

Example:

```
b = SER.readbyte()
```

receives a byte from serial port USIF0 handling, assigning the return value to b.

4.3.5. **SER.sendavail()**

This command queries the number of bytes available to send to SER buffer.

It has no input parameter.

The return value is a Python integer which is the number of bytes available to send to SER buffer.

Example:

```
n = SER.sendavail()
```

queries the number of bytes available to send, assigning the return value to n.

4.3.6. **SER.set_speed(speed, <char format>, <flow control>)**

This command sets the serial port TXD/RXD speed. The default serial port TXD/RXD speed is 115200.

The first input parameter *speed* is a Python string which is the value of the serial port speed. It can assume values in the range from '300' to '115200'.

The second optional parameter *<char format>* is a Python string that represents the character format to be used:

the first character is the number of bits per char (7 or 8), then the parity setting (N - none, E- even, O- odd) and in the end the number of stop bits (1 or 2). The default value is "8N1". For version 18.11.004 only "8N1" value is available

The third optional parameter *<flow control>* is only available from version 12.00.xx4, version 13.00.xx5 and version 18.11.004 and is a Python string that represents the flow control mode to be used:

"none" set no flow control enabled,

"hw" set hardware flow control enabled.

The default value is "none".

The return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

```
b = SER.set_speed('9600')
```



sets the serial port USIF0 speed to 9600, assigning the return value to b.



NOTE:

Sending the +IPR command to the device does not affect the physical serial port, you must use this function to set the speed of the port when using the Python engine.

4.3.7. SER.setDCD(DCD_value)

This command sets Carrier Detect (DCD) on the serial port USIF0.
From version 12.00.xx4, version 13.00.xx5 and version 18.11.004 this command has no effect if flow control is enabled.
The input parameter *DCD_value* is a Python integer which is either 0 if DCD is set to OFF or 1 if DCD is set to ON.
No return value.

Example:

```
SER.setDCD(1)
```

sets DCD to ON in USIF0.

4.3.8. SER.setCTS(CTS_value)

This command sets Clear to Send (CTS) on the serial port USIF0.
From version 12.00.xx4, version 13.00.xx5 and version 18.11.004 this command has no effect if flow control is enabled.
The input parameter *CTS_value* is a Python integer which is either 0 if CTS is set to OFF or 1 if CTS is set to ON.
No return value.

Example:

```
SER.setCTS(1)
```

sets CTS to ON in USIF0.

4.3.9. SER.setDSR(DSR_value)

This command sets Data Set Ready (DSR) on the serial port USIF0.
From version 12.00.xx4, version 13.00.xx5 and version 18.11.004 this command has no effect if flow control is enabled.
The input parameter *DSR_value* is a Python integer which is either 0 if DSR is set to OFF or 1 if DSR is set to ON.
No return value.

Example:



```
SER.setDSR(1)
```

sets DSR to ON in USIF0.

4.3.10. **SER.setRI(RI_value)**

This command sets Ring Indicator (RI) on the serial port USIF0.

From version 12.00.xx4, version 13.00.xx5 and version 18.11.004 this command has no effect if flow control is enabled.

The input parameter *RI_value* is a Python integer which is either 0 if RI is set to OFF or 1 if RI is set to ON.

No return value.

Example:

```
SER.setRI(1)
```

sets RI to ON in USIF0.

4.3.11. **SER.getRTS()**

This command gets Request to Send (RTS) from the serial port USIF0.

It has no input parameter.

The return value is a Python integer which is either 0 if RTS is set to OFF or 1 if RTS is set to ON.

Example:

```
rts = SER.getRTS()
```

gets RTS from USIF0, assigning the return value to rts.

4.3.12. **SER.getDTR()**

This command gets Data Terminal Ready (DTR) from the serial port USIF0.

It has no input parameter.

The return value is a Python integer which is either 0 if DTR is set to OFF or 1 if DTR is set to ON.

Example:

```
dtr = SER.getDTR()
```

gets DTR from USIF0, assigning the return value to dtr.



4.4. SER2 built-in module

SER2 built-in module is available only for product versions from 12.00.xx4, from 13.00.xx5 and from 18.11.004.

The SER2 built-in module is an interface between the Python core and the device trace serial port over the RX_AUX/TX_AUX pins direct handling. You need to use the SER2 built-in module if you want to send data from the Python script to the serial port USIF1 and to receive data from the serial port USIF1 to the Python script. This serial port has not the flow control physical lines.

This module is available for all products.

It is Python script developer responsibility to use it properly.

In fact serial port USIF1 is usually dedicated to trace and debug purposes and importing this module will make trace and debug unavailable. Moreover importing this module might interfere with device serial port configuration (see HE910 UE910 Family Ports Arrangements User Guide from version 12.xx.xx4, GE910 Family Ports Arrangements User Guide from version 13.xx.xx5 or CE Family Ports Arrangements User Guide for version 18.11.004).

If you want to use SER2 built-in module you need to import it:

```
import SER2
```

then you can use its methods, like in the following example:

```
a = SER2.set_speed('9600')
b = SER2.send('test')
c = SER2.sendbyte(0x0d)
d = SER2.read()
```

which sends 'test' followed by <CR> and receives data.

More details about SER2 built-in module methods can be found in the following paragraphs.

4.4.1. SER2.send(string)

This command sends a string to the serial port USIF1.

The input parameter *string* is a Python string to send to the serial port USIF1.

The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
a = SER2.send('test')
```

sends the string 'test' to the serial port USIF1 handling, assigning the return value to a.



NOTE:

The buffer available for the SER2.send command is 4096 bytes.



4.4.2. SER2.read()

This command receives a string from the serial port USIF1.

It has no input parameter.

The return value is a Python string which contains the data received and stored in buffer at the moment of command execution. The value might be empty if no data is received.

Example:

```
a = SER2.read()
```

receives a string from the serial port USIF1 handling, assigning the return value to a.



NOTE:

The buffer available for the SER2.read command is 4096 bytes. The maximum number of bytes returned by each SER2.read is 1023.



NOTE:

It is up to Python script to keep empty SER2.read buffer.

4.4.3. SER2.sendbyte(byte)

This command sends a byte to the serial port USIF1.

The input parameter *byte* can be zero or any Python byte to send to the serial port USIF1.

The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
b = SER2.sendbyte(0x0d)
```

sends the byte 0x0d, that corresponds to <CR>, to the serial port USIF1 handling, assigning the return value to b.

4.4.4. SER2.readbyte()

This command receives a byte from the serial port USIF1.

It has no input parameter.

The return value is a Python integer which is the byte value received and stored in buffer at the moment of command execution or is -1 if no data is received. The return value can also be zero.

Example:

```
b = SER2.readbyte()
```

receives a byte from serial port USIF1 handling, assigning the return value to b.



4.4.5. SER2.sendavail()

This command queries the number of bytes available to send to SER2 buffer. It has no input parameter.

The return value is a Python integer which is the number of bytes available to send to SER2 buffer.

Example:

```
n = SER2.sendavail()
```

queries the number of bytes available to send, assigning the return value to n.

4.4.6. SER2.set_speed(speed, <char format>)

This command sets the serial port USIF1 speed. The default serial port USIF1 speed is 115200.

The first input parameter *speed* is a Python string which is the value of the serial port speed. It can assume the values in the range from '300' to '115200'.

The second optional parameter *<char format>* is a Python string that represents the character format to be used:

the first character is the number of bits per char (7 or 8), then the parity setting (N - none, E- even, O- odd) and in the end the number of stop bits (1 or 2). The default value is "8N1".

The return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

```
b = SER2.set_speed('9600')
```

sets the serial port USIF1 speed to 9600, assigning the return value to b.



4.5. GPIO built-in module

The GPIO built-in module is an interface between the Python core and the module internal general purpose input output direct handling. The GPIO built-in module is used to set GPIO values and to read GPIO values from the Python script. You can control the GPIO pins also by sending internal 'AT#GPIO' commands using the MDM module, but using the GPIO module is faster because no command parsing is involved, therefore its use is recommended.



NOTE:

The Python core does not verify if the pins are already used for other purposes by other functions, it's the customer responsibility to ensure that no conflict over pins occurs.

If you want to use the GPIO built-in module you need to import it:

```
import GPIO
```

then you can use its methods as in the following example:

```
a = GPIO.getIOvalue(5)
b = GPIO.setIOvalue(4, 1)
```

this reads the GPIO 5 value and sets GPIO 4 to the output with value 1.

More details about GPIO built-in module methods are in the following paragraphs.

4.5.1. GPIO.setIOvalue(GPIOnumber, value)

This method sets the output value of a GPIO pin.

The first input parameter *GPIOnumber* is a Python integer which is the number of the GPIO. The second input parameter *value* is a Python integer which is the output value. It can be 0 or 1.

The return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

```
b = GPIO.setIOvalue(4, 1)
```

sets GPIO 4 to output with value 1, assigning the return value to b.



NOTE:

For versions up to 12.00.xx5 this method returns -1 if the *GPIOnumber* is not set to output.

4.5.2. GPIO.getIOvalue(GPIOnumber)

This method gets the input value of a GPIO.

The input parameter *GPIOnumber* is a Python integer which is the number of the GPIO.



The return value is a Python integer which is -1 if an error occurred otherwise it is the input value. It can be either 0 or 1.

Example:

```
b = GPIO.getIOvalue(5)
```

gets the GPIO 5 input value, assigning the return value to b.



NOTE:

For versions up to 12.00.xx5 this method returns -1 if the *GPIOnumber* is not set to input.

4.5.3. **GPIO.setIOdir(GPIOnumber, value, direction)**

This method sets the direction of a GPIO.

The first input parameter *GPIOnumber* is a Python integer which is the number of the GPIO. The second input parameter *value* is a Python integer which is the output value. It can be either 0 or 1. It is only used if the *direction* value is 1, it has no meaning if the *direction* value is 0.

The third input parameter *direction* is a Python integer which is the direction value. It can be either 0 for input or 1 for output. Other values are available for alternate functions. Refer to AT#GPIO command in product "AT Commands Reference Guide" for further notes (value is called mode).

The return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

```
c = GPIO.setIOdir(4, 0, 0)
```

sets GPIO 4 to input with the *value* parameter having no meaning, and assigning the return value to c.



NOTE:

When the *direction* value is not 1, although the parameter *value* has no meaning, it is necessary to assign it one of the two possible values: 0 or 1.



NOTE:

For version 18.11.004 the *direction* values for alternate functions are not supported.

4.5.4. **GPIO.getIOdir(GPIOnumber)**

This method gets the direction of a GPIO.

The input parameter *GPIOnumber* is a Python integer which is the number of the GPIO.



The return value is a Python integer which is -1 if an error occurred otherwise is direction value. It is 0 for input or 1 for output. Other values are available for alternate functions. Refer to AT#GPIO command in product "AT Commands Reference Guide" for further notes.

Example:

```
d = GPIO.getIOdir(7)
```

gets GPIO 7 direction, assigning the return value to d.



NOTE:

For version 18.11.004 the return values for alternate functions are not supported.

4.5.5. GPIO.getADC(*adcNumber*)

This method gets ADC value. It is equivalent to the AT#ADC command. The input parameter *adcNumber* is a Python integer which represents the ADC number that will be read and converted in voltage. The return value is a Python integer which is -1 if an error occurred otherwise the converted voltage is returned in mV.

Example:

```
mV = GPIO.getADC(2)
```

gets ADC number 2 input voltage, assigning the return value in mV.



NOTE:

For version 18.11.004 the only available value for *adcNumber* is 1.

4.5.6. GPIO.setDAC(*enable*, *value*)

This method sets the DAC value. It is equivalent to the AT#DAC command. The first input parameter *enable* is a Python integer and can assume values 0 or 1. If it is set to 1 enables DAC output otherwise if it is set to 0 disabled DAC output. The second input parameter *value* is a Python integer and represents the scale factor of output voltage and can assume values in the range 0-1023. The return value is a Python integer that has value -1 if an error occurred otherwise it has value 1.

Example:

```
res = GPIO.setDAC(1, 512)
```

sets DAC output voltage at half the range, assigning the return value to res.





NOTE:

For version 18.11.004 this method is not available.

4.5.7. **GPIO.setVAUX(vauxNumber, enable)**

This method enables or disables the VAUX. It is equivalent to the AT#VAUX command. The first input parameter *vauxNumber* is a Python integer that represents VAUX number that will be enabled or disabled.

The second input parameter *enable* is a Python integer that can assume value 1 in order to enable VAUX output or 0 if VAUX output should be disabled.

The return value is a Python integer that has value -1 if an error occurred otherwise it has the value 1.

Example:

```
res = GPIO.setVAUX(1, 1)
```

enables VAUX number 1 output, assigning the return value to res.



NOTE:

For version 18.11.004 the only available value for *vauxNumber* is 1.

4.5.8. **GPIO.getAXE()**

This method gets the hands free status value. It is equivalent to the AT#AXE command.

It has no input parameter.

The return value is a Python integer that is either 0 if a hand free is not connected or 1 if a hand free is connected.

Example:

```
hf = GPIO.getAXE()
```

gets the AXE value, assigning the return value to hf.



NOTE:

For version 18.11.004 this method has no effect.

4.5.9. **GPIO.setSLED(status, onDuration, offDuration)**

This method sets the status led configuration values. It is equivalent to the AT#SLED command.

The first input parameter *status* is a Python integer that represents the configuration of status led and can assume the following values:



- 0 - ALWAYS OFF
- 1 - ALWAYS ON
- 2 - AUTO
- 3 - BLINKING

The second input parameter *onDuration* is a Python integer which is the period of ON configuration of status led measured in 1/10s.

The third input parameter *offDuration* is a Python integer which is the period of OFF configuration of status led measured in 1/10s.

The return value is a Python integer which is -1 if an error occurred otherwise it is 1.

Example:

```
res = GPIO.setSLED(3, 10, 90)
```

sets status led configuration to blinking with 1s in ON period and 9s in OFF period, assigning the return value to res.

4.5.10. GPIO.getCBC()

This method gets the charger status and battery voltage. It is equivalent to the AT#CBC command.

It has no input parameters.

The return value is a Python tuple formatted in the following way:

(chargerStatus, batteryVoltage).

First element of tuple is a Python integer which is charger status:

- 0 - charger not connected
- 1 - charger connected and charging
- 2 - charger connected and charging process completed

Second element of tuple is a Python integer which is battery voltage in mV.

Example:

```
cbc = GPIO.getCBC()
```

gets charger status and battery voltage values, assigning the return value to cbc tuple.



NOTE:

For version 18.11.004 the return values 1 and 2 for charger status are not supported.



4.6. GPS built-in module

GPS built-in module is available only for product versions 12.xx.xxx and from 13.00.xx4. GPS built-in module is the interface between Python and module internal GPS controller. It is used in order to handle GPS controller without dedicated AT commands through MDM built-in module.

If you want to use GPS built-in module you need to import it:

```
import GPS
```

then you can use its methods like in the following example:

```
position = GPS.getActualPosition()
```

gets a string with position information formatted in the same way as AT\$GPSACP response. More details about GPS built-in module methods can be found in the following paragraphs.

4.6.1. GPS.powerOnOff(newStatus)

This method powers ON/OFF GPS controller. It is equivalent to the AT\$GPSP command. The input parameter *newStatus* is a Python integer and can have the following values:

- 0 - to power OFF GPS controller
- 1 - to power ON GPS controller.

There is no return value.

Example:

```
GPS.powerOnOff(0)
```

GPS controller is powered OFF.

4.6.2. GPS.getPowerOnOff()

This method gets GPS controller current power ON/OFF status.

It has no input parameter.

The return value is a Python integer which is 0 if GPS controller is powered off or 1 if GPS controller is powered on.

Example:

```
status = GPS.getPowerOnOff()
```

gets GPS controller current power ON/OFF status, assigning the return value to status.

4.6.3. GPS.resetMode(mode)

This method resets GPS controller. It is equivalent to the AT\$GPSR command.



4.6.6. **GPS.powerSavingWakeUp()**

This method wakes up GPS controller while in power saving mode. It is equivalent to the AT\$GPSWK command.

It has no input parameter.

The return value is a Python integer which is -1 if an error occurred otherwise is 1.

TeseoII-based GNSS receiver currently does not support power saving.

This method is available only for product versions from 12.00.xx6 and from 13.00.xx7.

Example:

```
res = GPS.powerSavingWakeUp()
```

wakes up GPS controller while in power saving, assigning the return value to res.

4.6.7. **GPS.getLastGGA()**

This method gets GPS last GGA NMEA sentence stored.

It has no input parameter.

The return value is a Python string which is the last GGA NMEA sentence formatted according to NMEA specification.

Example:

```
gga = GPS.getLastGGA()
```

gets last GGA NMEA sentence, assigning the return value to gga.

4.6.8. **GPS.getLastGLL()**

This method gets GPS last GLL NMEA sentence stored.

It has no input parameter.

The return value is a Python string which is the last GLL NMEA sentence formatted according to NMEA specification.

Example:

```
gll = GPS.getLastGLL()
```

gets last GLL NMEA sentence, assigning the return value to gll.

4.6.9. **GPS.getLastGSA()**

This method gets GPS last GSA NMEA sentence stored.

It has no input parameter.



The return value is a Python string which is the last GSA NMEA sentence formatted according to NMEA specification.

Example:

```
gsa = GPS.getLastGSA()
```

gets last GSA NMEA sentence, assigning the return value to gsa.

4.6.10. **GPS.getLastGSV()**

This method gets GPS last GSV NMEA sentence stored.
It has no input parameter.

The return value is a Python string which is the concatenation of the last GSV NMEA sentences formatted according to NMEA specification.

Example:

```
gsv = GPS.getLastGSV()
```

gets last GSV NMEA sentence, assigning the return value to gsv.

4.6.11. **GPS.getLastRMC()**

This method gets GPS last RMC NMEA sentence stored.
It has no input parameter.

The return value is a Python string which is the last RMC NMEA sentence formatted according to NMEA specification.

Example:

```
rms = GPS.getLastRMC()
```

gets last RMC NMEA sentence, assigning the return value to rmc.

4.6.12. **GPS.getLastVTG()**

This method gets GPS last VTG NMEA sentence stored.
It has no input parameter.

The return value is a Python string which is the last VTG NMEA sentence formatted according to NMEA specification.

Example:

```
vtg = GPS.getLastVTG()
```

gets last VTG NMEA sentence, assigning the return value to vtg.



4.6.13. GPS.getPosition()

This method gets GPS last position stored in numeric format.

It has no input parameter.

The return value is a Python tuple formatted in the following way:

(latitude, latNorS, longitude, lonEorW)

where:

- the first element of tuple *latitude* is a Python integer which is latitude in (degrees * 10000000), that is in degrees with 10000000 scale factor
- the second element of tuple *latNorS* is a Python string which is 'N' for north or 'S' for south
- the third element of tuple *longitude* is a Python integer which is longitude in (degrees * 10000000), that is in degrees with 10000000 scale factor
- fourth element of tuple *lonEorW* is a Python string which is 'E' for east or 'W' for west.

If GPS controller has no position information the following tuple will be returned:

(0, "", 0, "").

This method is available only for product versions from 12.00.xx6 and from 13.00.xx7.

Example:

```
pos = GPS.getPosition()
```

gets last position stored, assigning the return value to pos.



4.7. IIC built-in module

IIC built-in module is available only for product versions from 12.00.xx4, from 13.00.xx5 and from 18.11.004.

IIC built-in module is an implementation on the Python core of the IIC bus Master (No Multi-Master) based on general purpose input output (GPIO) using the bit-banging technique.

IIC built-in module allows creating one or more IIC bus Python objects on the available GPIO pins. These IIC bus Python objects are each mapped on creation on two GPIO pins that will be dedicated to the Serial Data and Serial Clock pins of each IIC bus.

If you want to use IIC built-in module you need to import it:

```
import IIC
```

then you can use its methods like in the following example:

```
IICbus = IIC.new(3, 4, 0x50)
IICbus.init()
rec = IICbus.readwrite('\x08'+ 'test', 10)
```

creates a new IIC bus object using GPIOs 3 and 4, with address 0x50, and then sends 08 hex byte followed by 'test' and receives a string of 10 bytes assigning it to rec.

More details about IIC built-in module methods can be found in the following paragraphs.



NOTE:

An external pull-up must be provided on SDA line since the line is working as open collector, on the other hand SCLK is driven with a complete push pull.

4.7.1. IIC.new(SDA_pin, SCL_pin, ADDR)

This command creates a new IIC bus Python object linked to given GPIO pins and with specified address.

The first two input parameters *SDA_pin* and *SCL_pin* are Python bytes, which are the GPIO pin numbers the SDA (Serial DATA) and SCL (Serial CLOCK) lines are mapped to.

The third input parameter *ADDR* is a Python integer and represents the address of IIC bus device. It is the address value of the IIC bus device without the less significant bit required by IIC bus protocol for R/W (read/write) command. It can be a 7 bit address or a 10 bit address. It can be zero.

The return value is the IIC bus Python object which shall then be used to interface to the IIC bus device.



NOTE:

All GPIO pins are available pins for the IIC bus.

It is Python script developer responsibility to avoid conflicts between GPIO pins, no automatic check of GPIO pins already used for other purposes (GPIO module, IIC module, SPI module) is available.





NOTE:

Python core implementation takes the third input parameter *ADDR*, shifts it left by one bit and sets R/W (read/write) bit on the less significant bit of the less significant byte.

Example:

```
IICbus = IIC.new(3, 4, 0x50)
```

creates a new IIC bus object using GPIO 3 for SDA, GPIO 4 for SCL and with address 0x50.

4.7.2. **init()**

This IIC bus Python object method initializes the IIC bus.
No input value.
No return value.

Example:

```
IICbus.init()
```

initializes IIC bus object.

4.7.3. **readwrite(string, <read_len>)**

This IIC bus Python object method sends a string to and/or receives a string from the IIC bus device linked to the IIC bus Python object at its creation.
The first input parameter *string* is a Python string to send to the IIC bus device.
The second optional parameter *<read_len>* is a Python integer which represents the length of the data to be received from the IIC bus device. Default value if omitted is 0. *read_len* value range is (0 ÷254).
The return value is a Python integer with value -1 if an error occurred or is a Python string which contains the data received, in the last case the value might be empty if no data is received.

If the address *ADDR* set at IIC bus Python object creation is not 0:

- if *string* is not empty it will be sent to IIC bus device (Write);
- if *read_len* is greater than 0 a string will be received from IIC bus device (Read);
- if *string* is not empty and *read_len* is greater than 0 first *string* will be sent to IIC bus device (Write) and then a string will be received from IIC bus device (Read).

If the address *ADDR* set at IIC bus Python object creation is 0:

- if *string* is not empty and it is at least 2 bytes long and *read_len* is 0 it will be sent to a generic IIC bus device (Write) whose address and W bit must be set properly by Python script developer in first byte of *string*;



- if *string* is not empty and it is at least one byte long and *read_len* is greater than 0 it will be sent to a generic IIC bus device whose address and R bit must be set properly by Python script developer in first byte of *string* and then a string will be received from IIC bus device (Read);
- it is not possible a Write and Read operation in the same *readwrite* call.

Examples:

```
IICbus = IIC.new(3, 4, 0x50)
IICbus.init()
rec = IICbus.readwrite('\x08'+ 'test', 10)
```

sends 08 hex byte followed by 'test' and then receives a string of 10 bytes assigning it to rec.

```
rec = IICbus.readwrite('\x08'+ 'test', 0)
```

sends only 08 hex byte followed by 'test'.

```
rec = IICbus.readwrite('\x08', 10)
```

sends 08 hex byte and then receives a string of 10 bytes assigning it to rec.

```
rec = IICbus.readwrite("", 10)
```

receives only a string of 10 bytes assigning it to rec.



4.8. SPI built-in module

SPI built-in module is available only for product versions from 12.00.xx4 and from 13.00.xx5. SPI built-in module is an implementation on the Python core of the SPI bus Master based on general purpose input output (GPIO) using the bit-banging technique.

SPI built-in module allows creating one or more SPI bus Python objects on the available GPIO pins. These SPI bus Python objects are each mapped on creation on three GPIO pins that will be dedicated to the Master Input Slave Output, Master Output Slave Input, Serial Clock pins of each SPI bus and optionally on up to 8 other GPIO pins as Chip Select/Slave select pins.

If you want to use SPI built-in module you need to import it:

```
import SPI
```

then you can use its methods like in the following example:

```
SPIbus = SPI.new(3, 4, 5, 6, 7)
SPIbus.init(0, 0, 0, 0)
rec = SPIbus.readwrite('test', 4)
```

creates a new SPI bus object using GPIOs 3, 4, 5, 6 and 7, and then sends 'test' and receives a string of 4 bytes assigning it to rec.

More details about SPI built-in module methods can be found in the following paragraphs.

4.8.1. SPI.new(SCLK_pin, MOSI_pin, MISO_pin, <SS0>, <SS1>, ..., <SS7>)

This command creates a new SPI bus Python object linked to given GPIO pins.

The first three input parameter *SCLK_pin*, *MOSI_pin* and *MISO_pin* are Python bytes, which are the GPIO pin numbers the SCLK (Serial CLock), MOSI (Master Output Slave Input) and MISO (Master Input Slave Output) lines are mapped to.

The following up to 8 optional parameters <SSi> are Python bytes, which are the GPIO pin numbers the SSi (ith Slave Select) lines are mapped to. The SSi parameter are optional because not all SPI devices have a Slave Select (SS) line, called Chip Select (CS) line as well. The SSi parameters must be used for those SPI devices that needs to be selected by Slave Select line.

The return value is the SPI bus Python object which shall then be used to interface to the SPI bus device.

NOTE:



All GPIO pins are available pins for the SPI bus.

It is Python script developer responsibility to avoid conflicts between GPIO pins, no automatic check of GPIO pins already used for other purposes (GPIO module, IIC module, SPI module) is available.

Example:

```
SPIbus = SPI.new(3, 4, 5, 6, 7)
```



creates a new SPI bus object using GPIO 3 for SCLK, GPIO 4 for MOSI, GPIO 5 for MISO, GPIO 6 for SS0, GPIO 7 for SS1.

4.8.2. **init(CPOL, CPHA, <SSPOL>, <SS>)**

This SPI bus Python object method initializes the SPI bus.

The first input parameter *CPOL* is a Python byte, represents Clock polarity and can have the following values:

0 for polarity low;

1 for polarity high.

The second input parameter *CPHA* is a Python byte, represents Clock phase transmission and can have the following values:

0 for data bit clocked/latched on the first edge of SCLK;

1 for data bit clocked/latched on the second edge of SCLK

The third optional parameter *<SSPOL>* is a Python byte, represents the Slave Select polarity and can have the following values:

0 for polarity low (default value if omitted);

1 for polarity high.

The fourth optional parameter *<SS>* is a Python byte, represents the default Slave Select line number to be used among those linked to the SPI bus Python object at its creation and can have values from 0 to 7.

Default behavior if omitted is that, unless SS parameter present in *readwrite*, no SS line will be used.

Behavior if present is that, unless SS parameter present in *readwrite*, this SS line will be used.

The return value is a Python integer, which is -1 if an error occurred, otherwise is 1.

Example:

```
SPIbus.init(0, 0, 0)
```

initializes SPI bus object: no SS line is defined as default and no SS line will be used unless set in *readwrite*.

```
SPIbus.init(0, 0, 0, 1)
```

initializes SPI bus object: SS line 1 is defined as default and second SS line assigned in *SPI.new* will be used unless set in *readwrite*.

4.8.3. **readwrite(string, <read_len>, <SS>)**

This SPI bus Python object method sends a string to and/or receives a string from the SPI bus device linked to the SPI bus Python object at its creation.

The first input parameter *string* is a Python string to send to the SPI bus device.

The second optional parameter *<read_len>* is a Python integer which represents the length of the data to be received from the SPI bus device. Default value if omitted is 0. *read_len* value range is (0 ÷ 254).

The third optional input parameter *<SS>*, is a Python byte, represents the Slave Select line number to be used among those linked to the SPI bus Python object at its creation and can have values from 0 to 7



4.9. MOD built-in module

MOD built-in module is available only for product versions from 12.00.xx4 and from 13.00.xx5.

MOD built-in module is the interface between Python and the module miscellaneous functions.

If you want to use MOD built-in module you need to import it:

```
import MOD
```

then you can use its methods like in the following example:

```
MOD.watchdogEnable(60)
```

starts software watchdog protection.

More details about MOD built-in module methods can be found in the following paragraphs.

4.9.1. MOD.watchdogEnable(timeout)

This method activates the software watchdog protection of the system against script blocking by performing an automatic reboot of the module when the watchdog reaches a determined value. To avoid system reboot *MOD.watchdogReset()* command must be called periodically.

The input parameter *timeout* is an integer, which is measured in seconds and represents the time to wait before executing the system reboot. The timeout range is (1 ÷ 36000 (10 hours)). No return value.

Example:

```
MOD.watchdogEnable(60)
```

activates watchdog that after 60sec from execution of this command will reboot the module unless *MOD.watchdogReset()* is called.

4.9.2. MOD.watchdogReset()

This method restarts the software watchdog counter that has been previously activated with the command *MOD.watchdogEnable(timeout)* preventing in this way the reboot of the module. It should be added in every part of the script that can cause a script blocking (loops, etc.) and is used only when Python watchdog is enabled.

No input value.

No return value.

Example:

```
MOD.watchdogReset()
```

restarts watchdog counter.



4.9.3. MOD.watchdogDisable()

This method disables the software watchdog protection that has been previously activated with the command *MOD.watchdogEnable(timeout)*. Software watchdog should be disabled before scripts critical lines such as *import*, since it takes a long time, and then enabled again after.

No input value.

No return value.

Example:

```
MOD.watchdogDisable()
```

disables watchdog.

4.9.4. MOD.powerSaving(timeout)

This method blocks script execution and sets the system in power saving mode for a given time interval or until an external event occurs (e.g. incoming call RING). Blocking script execution is necessary for power saving conditions to be achieved. Setting power saving mode is equivalent to AT+CFUN=0 command (see Telit Modules Software User Guide for further details).

The input parameter *timeout* is an integer, which is measured in seconds and represents the maximum time the Python script remains blocked. The Python script will exit power saving mode when the given value of timeout is reached or when an external event occurs (e.g. incoming call RING). If the timeout value is -1 the Python script will exit from power saving mode only when an external event occurs. The timeout value can be -1 or in the range (0 ÷ 1800000 (500 hours)).

The return value is a Python integer which is 0 if the Python script has exited power saving mode because an external event has occurred otherwise it is 1 if the Python script has exited power saving mode because the timeout has expired.

Example:

```
cause = MOD.powerSaving(100)
```

blocks Python script and enters power saving mode for a maximum of 100 sec or until an external event occurs.



4.10. USB0 built-in module

USB0 built-in module is available only for product versions from 12.00.xx5 and from 13.00.xx6.

The USB0 built-in module is an interface between the Python core and the device first mini USB port USB0. You need to use the USB0 built-in module if you want to send data from the Python script to the first mini USB port USB0 and to receive data from the first mini USB port USB0 to the Python script. This USB port handling module can be used, for example, to interface the module with an external device and read/send its data.

If you want to use USB0 built-in module you need to import it:

```
import USB0
```

then you can use its methods, like in the following example:

```
b = USB0.send('test')
c = USB0.sendbyte(0x0d)
d = USB0.read()
```

which sends 'test' followed by <CR> and receives data.

More details about USB0 built-in module methods can be found in the following paragraphs.

4.10.1. USB0.send(string, <timeout>)

This command sends a string to the first mini USB port USB0.

The first input parameter *string* is a Python string to send to the first mini USB port USB0.

The second input parameter <timeout> is optional and is a Python integer, measured in 1/10s. It represents the maximum time to wait for the string to be sent to the first mini USB port USB0 when buffer is full and further data is blocked. The timeout range is (0 ÷ 32767).

This method returns immediately after the string has been sent to the first mini USB port USB0 or after the timeout period if the whole string could not be sent to the first mini USB port USB0. The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
a = USB0.send('test')
```

sends the string 'test' to the first mini USB port USB0 handling, assigning the return value to a.



NOTE:

The buffer available for the USB0.send command is 32767 bytes.



4.10.2. USB0.read()

This command receives a string from the first mini USB port USB0.

It has no input parameter.

The return value is a Python string which contains the data received and stored in buffer at the moment of command execution. The value might be empty if no data is received.

Example:

```
a = USB0.read()
```

receives a string from the first mini USB port USB0 handling, assigning the return value to a.



NOTE:

The buffer available for the USB0.read command is 32767 bytes. The maximum number of bytes returned by each USB0.read is 8191.



NOTE:

It is up to Python script to keep empty USB0.read buffer.

4.10.3. USB0.sendbyte(byte, <timeout>)

This command sends a byte to the first mini USB port USB0.

The first input parameter *byte* can be zero or any Python byte to send to the first mini USB port USB0.

The second input parameter *<timeout>* is optional and is a Python integer, measured in 1/10s. It represents the maximum time to wait for the byte to be sent to the first mini USB port USB0 when buffer is full and further data is blocked. The timeout range is (0 ÷ 32767).

This method returns immediately after the byte has been sent to the first mini USB port USB0 or after the timeout period if the byte could not be sent to the first mini USB port USB0. The return value is a Python integer which is -1 if the timeout period has expired, 1 otherwise.

Example:

```
b = USB0.sendbyte(0x0d)
```

sends the byte 0x0d, that corresponds to <CR>, to the first mini USB port USB0 handling, assigning the return value to b.

4.10.4. USB0.readbyte()

This command receives a byte from the first mini USB port USB0.

It has no input parameter.

The return value is a Python integer which is the byte value received and stored in buffer at the moment of command execution or is -1 if no data is received. The return value can also be zero.



Example:

```
b = USB0.readbyte()
```

receives a byte from first mini USB port USB0 handling, assigning the return value to b.

4.10.5. **USB0.sendavail()**

This command queries the number of bytes available to send to USB0 buffer.

It has no input parameter.

The return value is a Python integer which is the number of bytes available to send to USB0 buffer.

Example:

```
n = USB0.sendavail()
```

queries the number of bytes available to send, assigning the return value to n.



5. Python standard functions

In this paragraph you can find detailed description of Python language supported features in **Telit module**. Note that all the functions listed below are available in the Python version 2.7.2.

5.1. Technical characteristics

5.1.1. General

All Python statements and almost all Python built-in types and functions are supported. See in the table below the features not supported:

| |
|-----------|
| complex |
| unicode |
| docstring |
| packages |

Available standard built-in modules are:

| |
|-------------|
| marshal |
| imp |
| _ast |
| __main__ |
| __builtin__ |
| sys |
| exceptions |
| gc |
| _warnings |
| _md5 |
| binascii |
| _sre |
| _weakref |
| _symtable |
| _functools |
| _socket |
| time |
| posix |
| thread |
| signal |
| errno |
| cStringIO |
| math |

All others are not supported.

A small collection of standard Python modules written in Python (not built-in) is available. These .py files are mostly identical to the ones available for PC with minor changes.



5.2. Python supported features

Refer to the documents available online such as: Python 2.7.2 Tutorial, Python 2.7.2 Reference Manual or Python 2.7.2 Library Reference for details about all the features listed in the paragraphs below.

5.2.1. Operators, statements, functions

List of supported operators, statements, functions:

- comments #
- line joining \
- operators +, -, *, /, **, %
- operators <<, >>, &, |, ^, ~
- parentheses
- assignment
- comparison operators <, >, ==, <=, >=, !=, <>
- comparison operators in, not in
- print statement
- if, elif, else statement
- indentation
- and, or, not keywords
- for in statement
- while statement
- range() function
- break and continue statements
- pass statement
- functions (without docstrings) (def)
- return statement
- lambda forms
- objects
- object methods
- del statement
- modules
- import statement
- from statement
- exceptions
- try except finally statements
- raise statement
- classes (class)
- class instances
- global statement
- is, is not tests
- exec statement
- iterators
- generators
- yield statement
- with statement



5.2.2. Built-in Functions

The following built-in functions are supported:

| |
|----------------------------|
| abs |
| all |
| any |
| basestring |
| bin |
| bool |
| bytearray |
| callable |
| chr |
| classmethod |
| cmp |
| compile |
| complex (raises exception) |
| delattr |
| dict |
| dir |
| divmod |
| enumerate |
| eval |
| execfile |
| filter |
| float |
| format |
| frozenset |
| getattr |
| globals |
| hasattr |
| hash |
| help |
| hex |
| id |
| input |
| int |
| isinstance |
| issubclass |
| iter |
| len |
| list |
| locals |
| long |



| |
|---------------|
| map |
| max |
| memoryview |
| min |
| next |
| object |
| oct |
| open |
| ord |
| pow |
| print |
| property |
| range |
| raw_input |
| reduce |
| reload |
| repr |
| reversed |
| round |
| set |
| setattr |
| slice |
| sorted |
| staticmethod |
| str |
| sum |
| super |
| tuple |
| type |
| vars |
| xrange |
| zip |
| <u>import</u> |
| apply |
| buffer |
| coerce |
| intern |

5.2.3. Built-in Constants

The following built-in constants are supported:

| |
|----------------|
| False |
| True |
| None |
| NotImplemented |
| Ellipsis |
| <u>debug</u> |



5.2.4. Truth Value Testing

Truth value testing is supported.

5.2.5. Boolean Operations

The following Boolean operations are supported:

| |
|---------|
| x or y |
| x and y |
| not x |

5.2.6. Comparisons

The following comparisons are supported:

| |
|--------|
| < |
| <= |
| > |
| >= |
| == |
| != |
| is |
| is not |

5.2.7. Numeric Types: Integer, Long Integer and Floating Point

The following operations are supported with the integer, long integer and floating point type:

| |
|---------------|
| x + y |
| x - y |
| x * y |
| x / y |
| x // y |
| x % y |
| -x |
| +x |
| abs(x) |
| int(x) |
| long(x) |
| float(x) |
| divmod(x, y) |
| pow(x, y) |
| x ** y |
| round(x[, n]) |



5.2.13. Sequence Types: String, List, Tuple, Bytearray, Buffer and Xrange

The following operations are supported with the string, list, tuple, bytearray, buffer and xrange types:

| |
|--------------|
| x in s |
| x not in s |
| s + t |
| s * n, n * s |
| s[i] |
| s[i:j] |
| s[i:j:k] |
| len(s) |
| min(s) |
| max(s) |
| s.index(i) |
| s.count(i) |

Xrange type supports only indexing, iteration and len().

5.2.14. Sequence Types: Unicode

Unicode is not supported.

5.2.15. Sequence Types: String

The following methods are supported with the string types:

| |
|------------|
| capitalize |
| center |
| count |
| decode |
| encode |
| endswith |
| expandtabs |
| find |
| format |
| index |
| isalnum |
| isalpha |
| isdigit |
| islower |
| isspace |
| istitle |
| isupper |
| join |
| ljust |



| |
|------------|
| lower |
| rstrip |
| partition |
| replace |
| rfind |
| rindex |
| rjust |
| rpartition |
| rsplit |
| rstrip |
| split |
| splitlines |
| startswith |
| strip |
| swapcase |
| title |
| translate |
| upper |
| zfill |

5.2.16. Mutable Sequence Types: List and bytearray

The following additional operations are supported with the lists and bytearray types:

| |
|--------------------------------|
| s[i] = x |
| s[i:j] = t |
| del s[i:j] |
| s[i:j:k] = t |
| del s[i:j:k] |
| s.append(x) |
| s.extend(x) |
| s.count(x) |
| s.index(x[, i[, j]]) |
| s.insert(i, x) |
| s.pop([i]) |
| s.remove(x) |
| s.reverse() |
| s.sort([cmp[, key[, reverse]]) |



5.2.17. Set Types: Set and Frozenset

The following methods are supported with the set and frozenset types:

| |
|----------------------|
| len(s) |
| x in s |
| x not in s |
| isdisjoint() |
| issubset() |
| set <= other() |
| set < other() |
| issuperset |
| set >= other() |
| set > other() |
| union |
| set other ...() |
| intersection |
| set & other & ...() |
| difference |
| set - other - ...() |
| symmetric_difference |
| set ^ other() |
| copy() |

The following methods are supported with the set type:

| |
|-----------------------------|
| update |
| set = other ...() |
| intersection_update |
| set &= other & ...() |
| difference_update |
| set -= other ...() |
| symmetric_difference_update |
| set ^= other() |
| add |
| remove |
| discard |
| pop |
| clear |



5.2.18. Mapping Types: Dictionary

The following operations are supported with the dictionaries:

| |
|---------------------------|
| len(d) |
| d[key] |
| d[key] = value |
| del d[key] |
| key in d |
| key not in d |
| iter(d) |
| clear() |
| copy() |
| fromkeys(seq,[value]) |
| get(key,[default]) |
| has_key(key) |
| items() |
| iteritems() |
| iterkeys() |
| itervalues() |
| keys() |
| pop(key,[default]) |
| popitem() |
| setdefault(key,[default]) |
| update([other]) |
| values() |
| viewitems() |
| viewkeys() |
| viewvalues() |

5.2.19. File Objects

The following methods are supported with the file objects:

| |
|------------------------|
| close() |
| flush() |
| fileno() |
| isatty() |
| next() |
| read([size]) |
| readline([size]) |
| readlines([sizehint]) |
| xreadlines() |
| seek(offset[, whence]) |
| tell() |
| write(str) |
| writelines(seq) |



The following attributes are supported with the file objects:

| |
|-----------|
| closed |
| encoding |
| errors |
| mode |
| name |
| newlines |
| softspace |

NOTE:

For product versions 12.xx.xxx.

Root directory for Python scripts and in general for text and binary files is:



/sys

and cannot be changed.

Files path name in Python scripts shall contain the root directory.

Path separator is /.

Example:

```
f = open('/sys/example.txt', 'rb')
```

From product versions 12.00.xx4 absolute and relative path names have been added.

Root directory for Python scripts and in general for text and binary files is still:

/sys

and cannot be changed.

Files *absolute* path name (path beginning with /) in Python scripts shall still contain the root directory.

Path separator is /.

Example:

```
f = open('/sys/example.txt', 'rb')
```

Files *relative* path name (path not beginning with /) in Python scripts will automatically refer to the root directory /sys and shall not contain it.

Example:

```
f = open('example.txt', 'rb')
```

NOTE:



For product versions 13.xx.xxx.

Root directory for Python scripts and in general for text and binary files is empty and cannot be changed.

Files path name in Python scripts will refer to the empty root directory.

Example:

```
f = open('example.txt', 'rb')
```



NOTE:



For product versions 18.11.004.

Root directory for Python scripts and in general for text and binary files is empty and cannot be changed.

Files path name in Python scripts will refer to the empty root directory.

Example:

```
f = open('example.txt', 'rb')
```



5.2.20. Memoryview Objects

The following methods are supported with the memoryview objects:

| |
|-----------|
| tobytes() |
| tolist() |

The following attributes are supported with the memoryview objects:

| |
|----------|
| format |
| itemsize |
| shape |
| ndim |
| strides |
| readonly |

5.2.21. Module Objects

Module objects are supported.

The following attributes are supported:

| |
|-------------|
| <u>dict</u> |
| name |

5.2.22. Classes and Class Instances

Classes and class instances are supported.

5.2.23. Function Objects

Function objects and function call are supported.

5.2.24. Method Objects

Method objects are supported.

5.2.25. Code objects

Code objects are supported.

5.2.26. Type Objects

Type objects are supported.



5.2.27. Null Object

Null object is supported.

5.2.28. Ellipsis Object

Ellipsis object is supported.

5.2.29. NotImplemented Object

NotImplemented object is supported.

5.2.30. Internal Types: Frame Objects

Frame objects are supported.

5.2.31. Internal Types: Traceback Objects

Traceback objects are supported.

5.2.32. Slice Objects

Slice objects are supported.



5.2.33. Built-in Exceptions

The following built-in exceptions are supported:

| |
|---------------------|
| BaseException |
| Exception |
| StandardError |
| ArithmeticError |
| BufferError |
| LookupError |
| EnvironmentError |
| AssertionError |
| AttributeError |
| EOFError |
| FloatingPointError |
| GeneratorExit |
| IOError |
| ImportError |
| IndexError |
| KeyError |
| KeyboardInterrupt |
| MemoryError |
| NameError |
| NotImplementedError |
| OSError |
| OverflowError |
| ReferenceError |
| RuntimeError |
| StopIteration |
| SyntaxError |
| IndentationError |
| TabError |
| SystemError |
| SystemExit |
| TypeError |
| UnboundLocalError |
| UnicodeError |
| ValueError |
| ZeroDivisionError |



5.2.34. Built-in Modules: marshal

Built-in marshal module is supported with the following functions:

| |
|-------|
| dump |
| load |
| dumps |
| loads |

The following constant is supported:

| |
|---------|
| version |
|---------|

5.2.35. Built-in Modules: imp

Built-in imp module is supported with the following functions:

| |
|--------------|
| find_module |
| get_magic |
| get_suffixes |
| load_module |
| new_module |
| lock_held |
| acquire_lock |
| release_lock |

The following constants are supported:

| |
|-------------|
| PY_SOURCE |
| PY_COMPILED |
| C_BUILTIN |
| PY_FROZEN |

5.2.36. Built-in Modules: _ast

Built-in _ast module is supported.

5.2.37. Built-in Modules: __main__

Built-in __main__ module is supported.

5.2.38. Built-in Modules: __builtin__

Built-in __builtin__ module is supported.



5.2.42. Built-in Modules: `_warnings`

Built-in `_warnings` module is supported.

5.2.43. Built-in Modules: `_md5`

Built-in `_md5` module is supported.
It is imported by `hashlib.py` module.

5.2.44. Built-in Modules: `binascii`

Built-in `binascii` module is supported with the following functions:

| |
|----------------------------|
| <code>a2b_uu</code> |
| <code>b2a_uu</code> |
| <code>a2b_base64</code> |
| <code>b2a_base64</code> |
| <code>a2b_qp</code> |
| <code>b2a_qp</code> |
| <code>a2b_hqx</code> |
| <code>rledecode_hqx</code> |
| <code>rlecode_hqx</code> |
| <code>b2a_hqx</code> |
| <code>crc_hqx</code> |
| <code>crc32</code> |
| <code>b2a_hex</code> |
| <code>hexlify</code> |
| <code>a2b_hex</code> |
| <code>unexlify</code> |

The following exceptions are supported:

| |
|-------------------------|
| <code>Error</code> |
| <code>Incomplete</code> |

5.2.45. Built-in Modules: `_sre`

Built-in `_sre` module is supported.

5.2.46. Built-in Modules: `_weakref`

Built-in `_weakref` module is supported.

5.2.47. Built-in Modules: `_symtable`

Built-in `_symtable` module is supported.



| |
|-----------------|
| SO_ACCEPTCONN |
| SO_BROADCAST |
| SO_ERR |
| SO_KEEPAIVE |
| SO_LINGER |
| SO_RCVBUF |
| SO_RCVTIMEO |
| SO_REUSEADDR |
| SO_SNDBUF |
| SO_TYPE |
| TCP_MAXSEG |
| TCP_NODELAY |
| AI_ADDRCONFIG |
| AI_ALL |
| AI_CANONNAME |
| AI_DEFAULT |
| AI_MASK |
| AI_NUMERICHOST |
| AI_PASSIVE |
| AI_V4MAPPED |
| AI_V4MAPPED_CFG |
| EAI_ADDRFAMILY |
| EAI_AGAIN |
| EAI_BADFLAGS |
| EAI_BADHINTS |
| EAI_FAIL |
| EAI_FAMILY |
| EAI_MEMORY |
| EAI_NODATA |
| EAI_NONAME |
| EAI_PROTOCOL |
| EAI_SERVICE |
| EAI_SOCKTYPE |
| EAI_SYSTEM |
| NI_DGRAM |
| NI_MAXHOST |
| NI_MAXSERV |
| NI_NAMEREQD |
| NI_NOFQDN |
| NI_NUMERICHOST |
| NI_NUMERICSERV |

The following non standard constant is supported in product versions 12.xx.xxx and from 13.00.xx5 but it is not supported in version 18.11.004:

| |
|--------------|
| SO_CONTEXTID |
|--------------|



Socket objects support the following methods:

| |
|---------------|
| accept |
| bind |
| close |
| connect |
| connect_ex |
| fileno |
| getpeername |
| getsockname |
| getsockopt |
| listen |
| recv |
| recv_into |
| recvfrom |
| recvfrom_into |
| send |
| sendall |
| sendto |
| setblocking |
| settimeout |
| gettimeout |
| setsockopt |
| shutdown |

Socket objects support the following attributes:

| |
|---------|
| family |
| type |
| proto |
| timeout |

Built-in `_socket` module is imported by `socket.py`.

5.2.50. Built-in Modules: time

Built-in time module is supported with the following functions:

| |
|-------|
| time |
| clock |
| sleep |



5.2.51. Built-in Modules: posix

Built-in posix module is supported with the following functions:

| |
|------------|
| stat |
| unlink |
| remove |
| rename |
| open |
| close |
| closerange |
| lseek |
| read |
| write |
| fstat |
| isatty |
| strerror |

The following variable is supported:

| |
|---------|
| environ |
|---------|

The following exception is supported:

| |
|-------|
| error |
|-------|

The following constants are supported:

| |
|------|
| F_OK |
| R_OK |
| W_OK |
| X_OK |

5.2.52. Built-in Modules: thread

Built-in thread module is supported with the following functions:

| |
|----------------------------------|
| start_new_thread |
| exit |
| allocate_lock |
| get_ident |
| stack_size (raises exception) |

Lock objects support the following methods:

| |
|---------|
| acquire |
| release |
| locked |

The following exception is supported:

| |
|-------|
| error |
|-------|



Built-in thread module is imported by threading.py.



NOTE:

It is strongly recommended to terminate main thread only after all secondary threads are terminated.

5.2.53. Built-in Modules: signal

Built-in signal module is supported with the following functions:

| |
|-----------|
| signal |
| getsignal |

The following constants are supported:

| |
|---------|
| SIG_DFL |
| SIG_IGN |

The following non standard constants are supported:

| |
|---------|
| SIGMDM |
| SIGMDM2 |

5.2.54. Built-in Modules: errno

Built-in errno module is supported with the following variable:

| |
|-----------|
| errorcode |
|-----------|

5.2.55. Built-in Modules: cStringIO

Built-in cStringIO module is supported with the following function:

| |
|----------|
| StringIO |
|----------|

The following objects are supported:

| |
|------------|
| InputType |
| OutputType |



5.2.56. Built-in Modules: math

Built-in math module is only supported in product versions from 12.00.xx4 and from 13.00.xx5 with the following functions:

| |
|-----------|
| acos |
| acosh |
| asin |
| asinh |
| atan |
| atan2 |
| atanh |
| ceil |
| copysign |
| cos |
| cosh |
| degrees |
| erf |
| erfc |
| exp |
| expm1 |
| fabs |
| factorial |
| floor |
| fmod |
| frexp |
| fsum |
| gamma |
| hypot |
| isinf |
| isnan |
| ldexp |
| lgamma |
| log |
| log1p |
| log10 |
| modf |
| pow |
| radians |
| sin |
| sinh |
| sqrt |
| tan |
| tanh |
| trunc |

The following constants are supported:

| |
|----|
| e |
| pi |



5.2.57. Library Modules

A small collection of standard Python modules written in Python (not built-in) is available. These .py files are mostly identical to the ones available for PC with minor changes.

- os.py
- posixpath.py
- stat.py
- genericpath.py
- socket.py
- functools.py
- types.py
- threading.py
- hashlib.py



6. Python non standard functions

6.1. socket non standard functions

Only supported in product versions 12.xx.xxx and from 13.00.xx5.
Built-in `_socket` module is imported by `socket.py`.

6.1.1. Non standard socket option flag: `SO_CONTEXTID`

The following non standard constant is supported:

`SO_CONTEXTID`

It is the socket option flag used to link a socket object to a context identifier after PDP Context Activation procedure.



NOTE:

PDP Context Activation procedure can be obtained using `AT+CGDCONT` and `AT#SGACT` commands on MDM or MDM2 interface.

All socket objects must be linked to a context identifier.

In the following example

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_CONTEXTID, 1)
```

a socket object is created and then linked to context identifier number one.



NOTE:

There is a firewall always active on module.

Without firewall proper configuration socket methods might not work as expected.

Firewall can be configured using `AT#FRWL` command on MDM or MDM2 interface.

Sockets used by standard functions (e.g. `gethostbyname`) are silently linked to context identifier number one.



6.2. signal non standard functions

6.2.1. Non standard signal: SIGMDM

The following non standard constant is supported:

SIGMDM

It is the signal number linked to the event of presence of data in MDM.read buffer.

Example:

```
signal.signal(signal.SIGMDM, MDMReadHandler)
```

6.2.2. Non standard signal: SIGMDM2

The following non standard constant is supported:

SIGMDM2

It is the signal number linked to the event of presence of data in MDM2.read buffer.

Example:

```
signal.signal(signal.SIGMDM2, MDM2ReadHandler)
```



8. Python Script Emulation on PC

8.1. Executing the Python script on PC

The steps required to have a script executing by the Python engine on PC in a similar way as on the **Telit module** are:

- install Python on PC;
- install optional serial package on PC;
- copy on PC Python modules that emulates custom built-in modules (MDM, MDM2, SER, GPIO, GPS);
- run the Python script.

8.1.1. Install Python

Download Python 2.7.2 installation from
<http://www.python.org/>
<http://www.python.org/download/releases/2.7.2/>
and install it.

8.1.2. Install optional serial package

Download PythonWin installation related to Python 2.7 from
<http://sourceforge.net/projects/pywin32/files/pywin32/>
and install it.
Download pyserial package installation from
<http://sourceforge.net/projects/pyserial/files/pyserial/>
and install it.

8.1.3. Copy Python modules

A collection of Python modules written in Python (not built-in) that emulates custom built-in modules is available.

MDM.py
MDM2.py
SER.py
SER2.py
GPIO.py
GPS.py
USB0.py

Copy these files on PC.

These modules make the difference between running the script on PC and on module.



8.1.4. Run the Python script

Run the Python script on PC.

Main differences between executing the Python script on PC compared to module are:

- script speed execution;
- different behaviour between emulating modules and custom built-in modules.



9. Document History

| Revision | Date | Changes |
|----------|------------|--|
| 0 | 2012-02-27 | First issue |
| 1 | 2012-10-29 | New title and changes for GE910 product family |
| 2 | 2013-05-29 | Implementation of new functionalities on UE/HE910 series (power saving, watchdog, IIC, SPI, SER2, SER flow control, math module, absolute/relative path) |
| 3 | 2013-07-26 | Porting of the new functionalities on GE910 product family |
| 4 | 2014-05-07 | Implementation of new functionalities on UE/HE910 and GE910 series (USB0) Changes for CE910 product family |
| 5 | 2015-03-03 | Updated GPS section Updated Applicability Table |

