Easy Script in Python

80000ST10020a Rev.1 - 18/09/06

Making machines talk.

This document is relating to the following products:

Model	P/N
GM862-QUAD-PY	3990250656
GM862-GPS	3990250657
GE863-PY	3990250654
GE863-PY (leaded balls)	3990250665
GE863-PY	3990250661
GE863-GPS	3990250660
GE864-PY	3990250650
GC864-PY	3990250676
EZ10-QUAD-PY	3990150467
GT863-PY	3990250466

Contents

DISCLAIMER

The information contained in this document is proprietary information of Telit Communications S.p.A..

Telit Communications S.p.A. makes every effort to ensure the quality of the information it makes available. Notwithstanding the foregoing, Telit Communications S.p.A. does not make any warranty as to the information contained herein, and does not accept any liability for any injury, loss or damage of any kind incurred by use of or reliance upon the information.

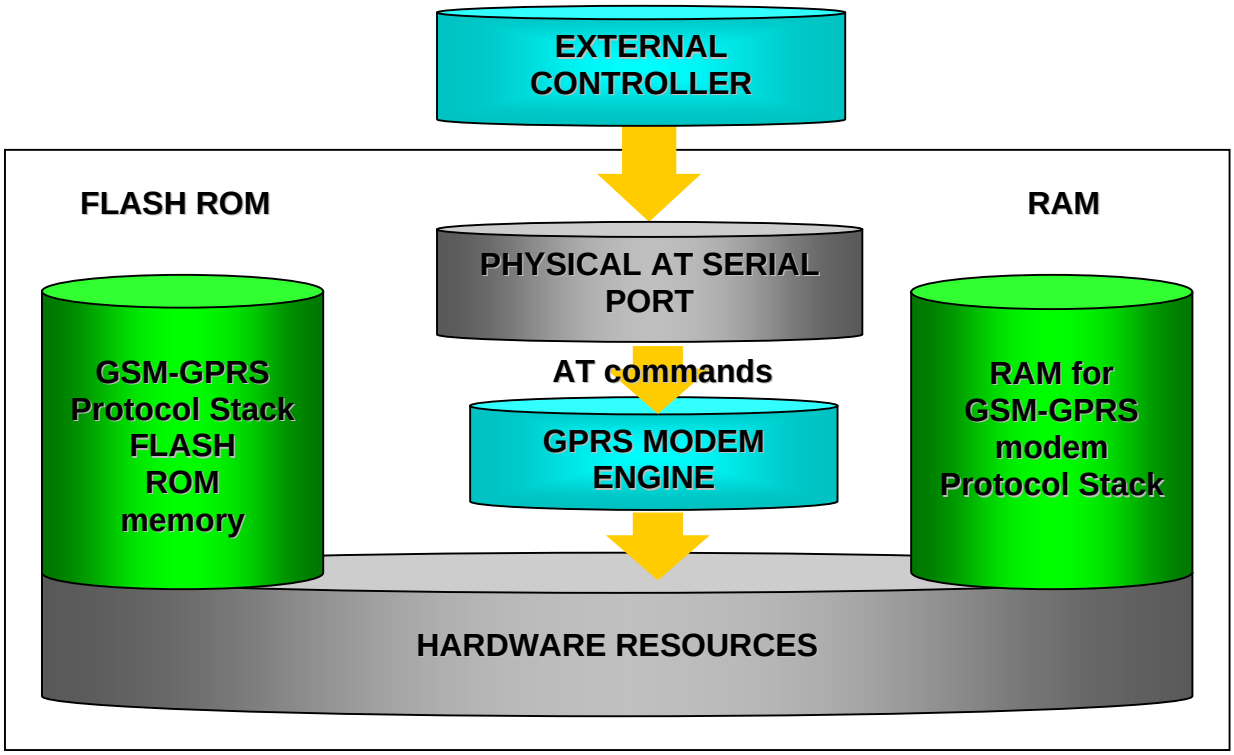
Telit Communications S.p.A. disclaims any and all responsibility for the application of the devices characterized in this document, and notes that the application of the device must comply with the safety standards of the applicable country, and where applicable, with the relevant wiring rules.

Telit Communications S.p.A. reserves the right to make modifications, additions and deletions to this document at any time and without notice.

1 Easy Script Extension - Python interpreter

1.1 Overview

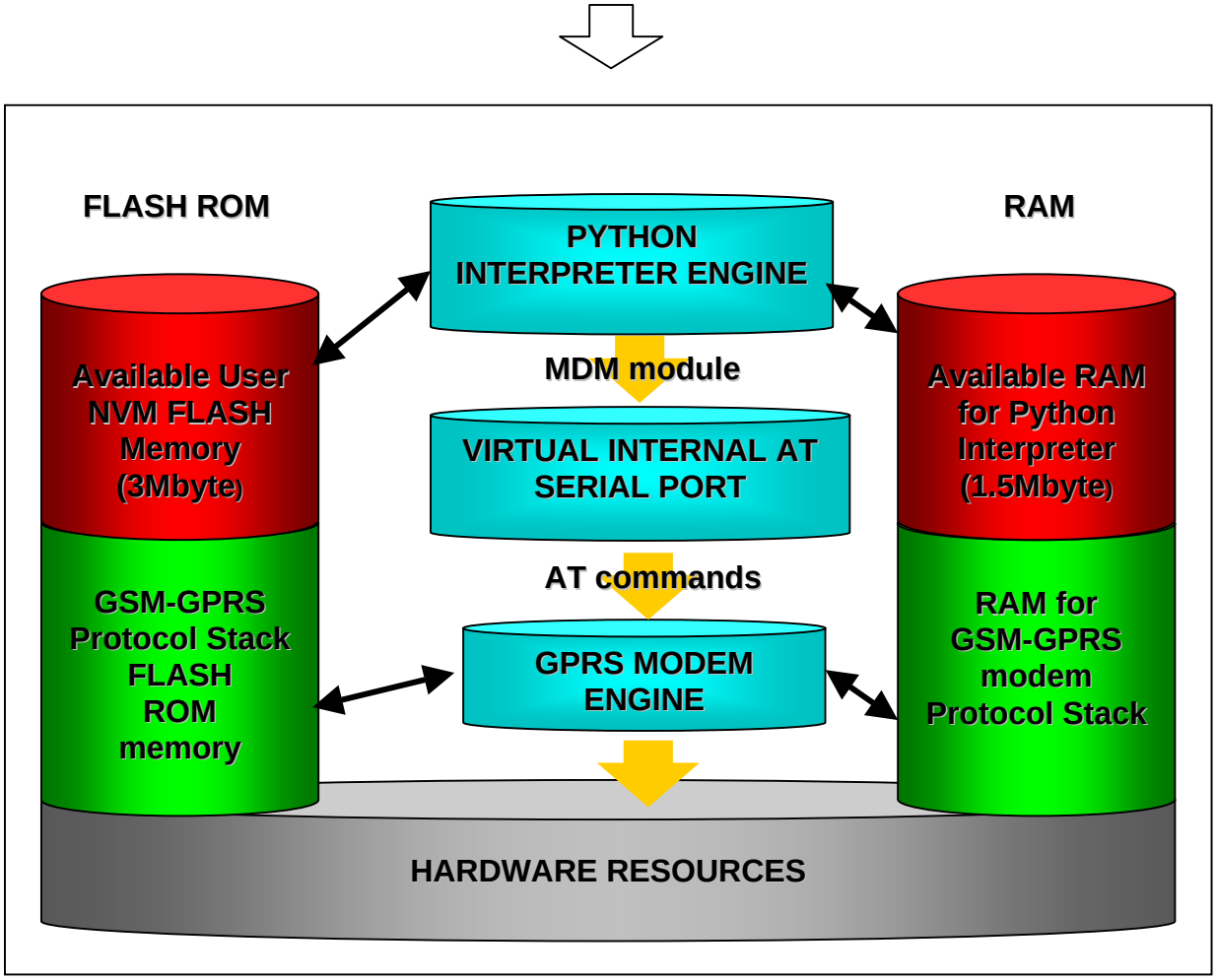
The Easy Script Extension is a feature that allows driving the modem "internally", writing the controlling application directly in a nice high level language: Python.
The Easy Script Extension is aimed at low complexity applications where the application was usually done by a small microcontroller that managed some I/O pins and the module through the AT command interface.
A schematic of such a configuration can be:



In order to eliminate this external controller, and further simplify the programming of the sequence of operations, inside the Python version it is included:

- Python script interpreter engine v. 1.5.2+
- around 3MB of Non Volatile Memory room for the user scripts and data
- 1.5 MB RAM reserved for Python engine usage

A schematic of this approach is:



1.2 Python 1.5.2+ Copyright Notice

The Python code implemented into the module is copyrighted by Stichting Mathematisch Centrum, this is the license:
Copyright © 1991-1995 by Stichting Mathematisch Centrum, Amsterdam, The Netherlands.
All Rights Reserved
Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved.
Copyright (c) 2001-2006 Python Software Foundation; All Rights Reserved.
All Rights Reserved are retained in Python.

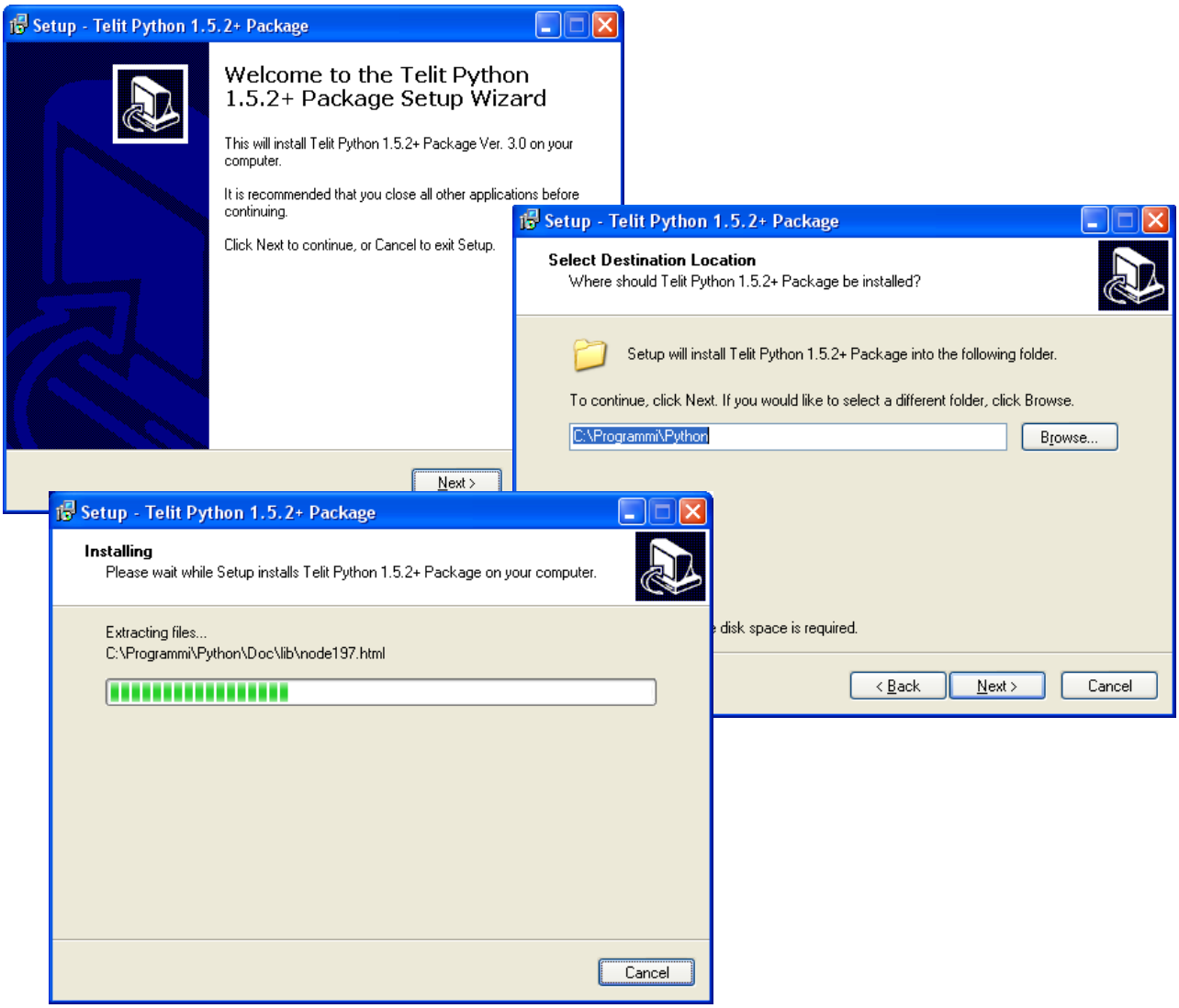
Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Stichting Mathematisch Centrum or CWI or Corporation for National Research Initiatives or CNRI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
While CWI is the initial source for this software, a modified version is made available by the Corporation for National Research Initiatives (CNRI) at the Internet address ftp://ftp.python.org.

STICHTING MATHEMATISCH CENTRUM AND CNRI DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM OR CNRI BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

1.3 Python installation

In order to have software that functions correctly the system requirement is PC running Windows 2000 or XP.
To get PythonWin package 1.5.2+ with the latest version please contact Technical Support at the e-mail: ts-modules@telit.com. For the moment the latest version available is *TelitPy1.5.2+_V3.0.exe.*

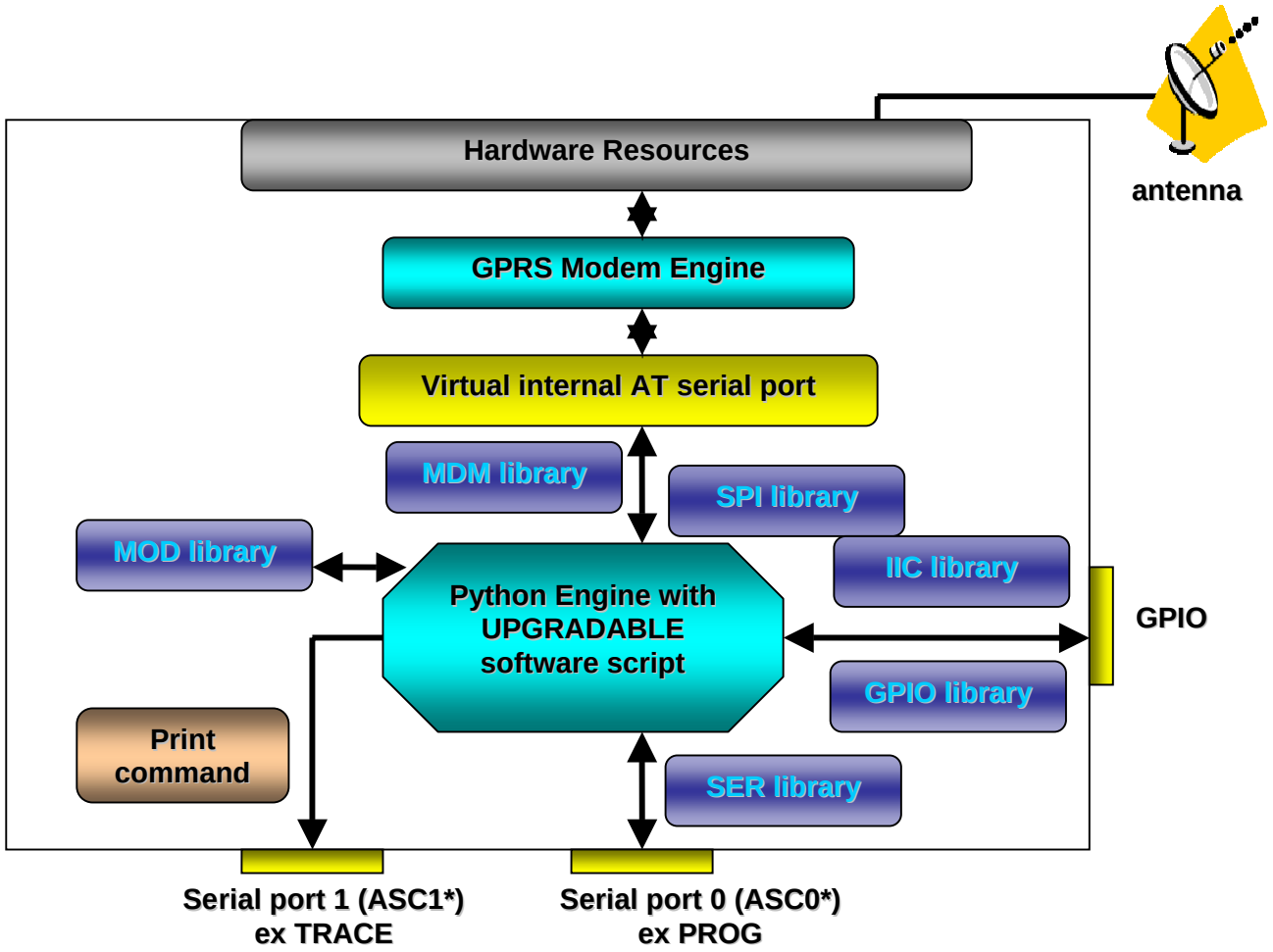
To install *Telit Python package* you need to execute the exe file *TelitPy1.5.2+_V3.0.exe* and let the installer use the default settings. The installation contains the Python compiler package. The *Telit Python package* is placed in the folder C:\Program Files\Python\ .The correct path in the Windows Environmental variables will be set up automatically.



1.4 Python implementation description

Python scripts are text files stored in NVM inside the **Telit module**. There's a file system inside the module that allows to write and read files with different names on one single level (no subdirectories are supported).
Attention: it is possible to run only one Python script at the time.

The Python script is executed in a task inside the **Telit module** at the lowest priority, making sure this does not interfere with GSM/GPRS normal operations. This allows serial ports, protocol stack etc. to run independently from the Python script.
The Python script interacts with the **Telit module** functionality through four build-in interfaces.



- **The MDM interface** is the most important one. It allows 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 the same as the usual serial port interface in the **Telit module**. The difference is that this interface is not a real serial port but just an internal software bridge between Python and mobile internal AT command handling engine. All AT commands working in the **Telit module** are working in this software interface as well. Some of them have no meaning on this interface, such as those regarding serial port settings. The usual concept of flow control keeps its meaning over this interface, but it's managed internally.
- **The SER interface** allows Python script to read from and write to the *real*, physical serial port where usually the AT command interface resides, for example to read NMEA information from a GPS device. When Python is running this serial port is free to be used by Python script because it is not used as AT command interface since the AT parser is mapped into the internal virtual serial port. No flow control is available from Python on this port.
- **The GPIO interface** allows Python script to handle general purpose input output faster than through AT commands, skipping the command parser and going directly to control the pins.
- **The MOD interface** is a collection of useful functions.
- **The IIC interface** is an implementation on the Python core of the IIC bus Master. It allows Python to create one or more IIC bus on the available GPIO pins.
- **The SPI interface** is an implementation on the Python core of the SPI bus Master. It allows Python to create one or more IIC bus on the available GPIO pins.

For the debug, the print command is directly forwarded on the EMMI TX pin (second serial port) at baud rate115200bps 8N1.

1.5 Introduction to Python

Python is a dynamic object oriented programming language that can be used for many kinds of software development. It offers strong support for integration with different tools, comes with extensive standard libraries, and can be learned in a few days time.

1.5.1 Data types

There are three groups of data types in Python:

- Scalars have the subtypes integer, long integer (with an arbitrary number of digits), and strings. For example:

  i = 1; li = 9999999999L; s = 'Hello'

- Sequences contain any number of arbitrary objects in a defined order.

  L = [1, 5, 3, 9, 14];

- Associative lists (more commonly known as dictionaries) allow the access to values based on keys. These keys can be arbitrary but immutable objects. For example:

  D = {'b': 'Python', 'a': 5}; print D['a']

  prints 5.

- Unlike Pascal, C, C++ or Java, Python is a dynamically typed language. Thus, the following code is perfectly valid Python:

```
a = 7                    # 7 (integer)
a = str(2*a) + ' bytes'  # '7 bytes' (string)
```

In Python variables are not defined in the script, they appear only when used.

1.5.2 Operators

Python has the following operators:

- Arithmetic and bitwise operators

  + - * / % ** ~ << >> & ^ |

- Relational and logical operators

  is in < <= > >= == != not and or

- Assignments

  = += -= *= /= %= **= <<= >>= &= ^= |=

- Other operators

  () [ ] { } [:] `` . lambda

1.5.3 Compound statements

- Statements that belong to the same logical group are indented by the same amount of white space:

  ```
  if a > 0:
      b = 1
      c = 2
  ```

  Usually, each statement starts on a new line.

- A statement is continued by putting a backslash \ at the end of a line. This isn't necessary if there are still parentheses (or brackets or braces) open:

  ```
  my_list = [1,              # open bracket, statement continues
             ['abc', 2],     # nested list
             -3+6j]          # closed outermost bracket, statement ends
  print my_list
  ```

1.5.4 Conditional execution

- Python uses *if*, *elif* (not *elsif* or *elseif*), and *else* to denote conditional execution of statements. For example:

```
if a > b:
   print 'a is greater than b.'
elif a < b:
   print 'a is lower than b.'
else:
   print 'a equals b.'
```

- You can use "abbreviated" interval tests:

```
if 2 <= a <= 7:
   print 'a is in the interval [2, 7].'
```

1.5.5 Loops

- Loops in Python are defined by the keywords *for* and *while*.
- The following example uses a *while* loop to collect all numbers from 0 to 99 in a list.

```
numbers = [ ]
i = 0
while i < 100:
   numbers.append(i)
   i = i + 1                    # or  i += 1  since Python 2.0
```

- A similar *for* loop looks like:

```
numbers = [ ]
for i in range(100):
   numbers.append(i)
```

- Instead of the explicit loops above also an implicit loop is possible:

```
numbers = range(100)
```

range(100) generates a list of all integers from 0 to 99 (not 100).

1.5.6 Resources

Some useful manuals for Python can be found on the following links:

http://www.python.org/doc/current/tut/tut.html

http://www.hetland.org/python/instant-python.php

http://rgruet.free.fr/PQR2.2.html

1.6 Python core supported features

The Python core version is 1.5.2+ (string methods added to 1.5.2).
You can use all Python statements and almost all Python built-in types and functions.

Built-in types and functions **not** supported	Available modules (all **others** are **not** supported)
complex	marshal
float	imp
long	_main_
docstring	_builtin_
	sys
	md5

2 Python Build-in Custom Modules

Several build in custom modules have been included in the python core, specifically aimed at the hardware environment of the module.

The build in modules included are:

MDM	interface between Python and mobile internal AT command handling
SER	interface between Python and mobile internal serial port ASC0 direct handling
GPIO	interface between Python and mobile internal general purpose input output direct handling
MOD	interface between Python and mobile miscellaneous functions
IIC	custom software Inter IC bus that can be mapped on creation over almost any GPIO pin available
SPI	custom software Serial Protocol Interface bus that can be mapped on creation over almost any GPIO pin available

2.1 MDM built-in module

MDM built-in module is the interface between Python and the module AT command parser engine.
You need to use MDM built-in module if you want to send AT commands and data from Python script to the network and receive responses and data from the network during connections.
Default start configuration is echo disabled (ATE0) and long form (verbose) return codes (ATV1),
If you want to use MDM built-in module you need to *import* it first:

```
import MDM
```
then you can use MDM built-in module methods like in the following example:
```
a = MDM.send('AT', 0)
b = MDM.sendbyte(0x0d, 0)
c = MDM.receive(10)
```

which sends 'AT' and receives 'OK'.
More details about MDM built-in module methods are in the following paragraphs.

2.1.1 MDM.send(string, timeout)

Sends a string to AT command interface. First input parameter *string* is a Python string which is the string to send to AT command interface. Second input parameter *timeout* is a Python integer, which is measured in 1/10s, and represents the time of waiting for the string to be sent to AT command interface, with maximum value of *timeout*. Waiting time is caused by flow control.
Return value is a Python integer which is -1 if timeout expired otherwise is 1.
Example:

a = MDM.send('AT', 5)
sends string 'AT' to AT command handling, possibly waiting for 0.5 s, assigning return value to a.

2.1.2 MDM.receive(timeout)

Receives a string from AT command interface waiting for it until timeout is expired. Return value will be the first string received no matter of how long the timeout is. Request to Send (RTS) is set to ON. Input parameter *timeout* is a Python integer, which is measured in 1/10s, and represents the time of waiting for the string from AT command interface, with maximum value of *timeout*.
Return value is a Python string which is an empty string if *timeout* expired without any data received otherwise is the string containing data received.
Example:

a = MDM.receive(15)

receives a string from AT command handling, possibly waiting for it for 1.5 s, assigning return value to a.

2.1.3 MDM.read()

Receives a string from AT command interface without waiting for it. Request to Send (RTS) is set to ON. No input parameter.
Return value is a Python string which is an empty string if no data received otherwise is the string containing data received in the moment when command is activated.
Example:

a = MDM.read()

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

2.1.4 MDM.sendbyte(byte, timeout)

Sends a byte to AT command interface. First input parameter *byte* is a Python byte which is any byte value to send to AT command interface. It can be zero. Second input parameter *timeout* is a Python

integer, which is measured in 1/10s, and represents the time of waiting for the string from AT command interface, with maximum value of *timeout*. Waiting time is caused by flow control.
Return value is a Python integer which is -1 if timeout expired otherwise is 1.
Example:

b = MDM.sendbyte(0x0d, 0)

sends byte 0x0d, that stands for CR, to AT command handling, without waiting, assigning return value to b.

2.1.5 MDM.readbyte()

Receives a byte from AT command interface without waiting for it. Request to Send (RTS) is set to ON. No input parameter.
Return value is a Python integer which is -1 if no data received otherwise is the byte value received. It can be zero.
Example:

b = MDM.readbyte()

receives a byte from AT command handling, assigning return value to b.

2.1.6 MDM.getDCD()

Gets Carrier Detect (DCD) from AT command interface. No input parameter.
Return value is a Python integer which is 0 if DCD is OFF or 1 if DCD is ON.
Example:

cd = MDM.getDCD()

gets DCD from AT command handling, assigning return value to cd.

2.1.7 MDM.getCTS()

Gets Clear to Send (CTS) from AT command interface. No input parameter.
Return value is a Python integer which is 0 if CTS is OFF or 1 if CTS is ON.
Example:

cts = MDM.getCTS()

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

2.1.8 MDM.getDSR()

Gets Data Set Ready (DSR) from AT command interface. No input parameter.
Return value is a Python integer which is 0 if DSR is OFF or 1 if DSR is ON.
Example:

dsr = MDM.getDSR()

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

2.1.9 MDM.getRI()

Gets Ring Indicator (RI) from AT command interface. No input parameter.
Return value is a Python integer which is 0 if RI is OFF or 1 if RI is ON.
Example:

ri = MDM.getRI()

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

2.1.10 MDM.setRTS(RTS_value)

Sets Request to Send (RTS) in AT command interface. Input parameter *RTS_value* is a Python integer which is 0 if setting RTS to OFF or 1 if setting RTS to ON.
No return value.
Example:

MDM.setRTS(1)

sets RTS to ON in AT command handling.

2.1.11 MDM.setDTR(DTR_value)

Sets Data Terminal Ready (DTR) in AT command interface. Input parameter *DTR_value* is a Python integer which is 0 if setting DTR to OFF or 1 if setting DTR to ON.
No return value.
Example:

MDM.setDTR(0)

sets DTR to OFF in AT command handling.

2.2 SER built-in module

SER built-in module is the interface between Python core and the device serial port over the RXD/TXD pins direct handling. You need to use SER built-in module if you want to send data from Python script to serial port and to receive data from serial port ASC0 to Python script. This serial port handling module can be used for example to interface the module with an external device such as a GPS and read/send its data (NMEA for example).

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

import SER

then you can use SER built-in module methods like in the following example:

a = SER.set_speed('9600')
b = SER.send('test')
c = SER.sendbyte(0x0d)
d = SER.receive(10)
which sends 'test' followed by CR and receives data waiting for one second.
More details about SER built-in module methods are in the following paragraphs.

2.2.1 SER.send(string)

Sends a string to the serial port TXD/RXD. Input parameter *string* is a Python string which is the string to send to serial port ASC0.
Return value is a Python integer which is -1 if an error occurred otherwise is 1.
Example:

a = SER.send('test')

sends string 'test' to serial port ASC0 handling, assigning return value to a.

Note: the buffer available for SER.send(string) command is 2048bytes

2.2.2 SER.receive(timeout)

Receives a string from serial port TXD/RXD waiting for it until timeout is expired. Return value will be the first string received no matter of how long the timeout is. Input parameter *timeout* is a Python integer, which is measured in 1/10s, and represents the time of waiting for the string from AT command interface, with maximum value of *timeout*.
Return value is a Python string which is an empty string if *timeout* expired without any data received otherwise is the string containing data received.

Example:

a = SER.receive(15)

receives a string from serial port handling, waiting for it for 1.5 s, assigning return value to a.

2.2.3 SER.read()

Receives a string from serial port TXD/RXD without waiting for it. No input parameter.
Return value is a Python string which is an empty string if no data received otherwise is the string containing data received in the moment when command is activated.
Example:

a = SER.read()

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

Note: the buffer available for the SER.receive(timeout) and SER.read() commands is 256bytes

2.2.4 SER.sendbyte(byte)

Sends a byte to serial port TXD/RXD. Input parameter *byte* is a Python byte which is any byte value to send to serial port. It can be zero.
Return value is a Python integer which is -1 if an error occurred otherwise is 1.
Example:

b = SER.sendbyte(0x0d)

sends byte 0x0d, that is CR, to serial port handling, assigning return value to b.

2.2.5 SER.receivebyte(timeout)

Receives a byte from serial port TXD/RXD waiting for it until timeout is expired. Return value will be the first byte received no matter of how long the timeout is. Input parameter *timeout* is a Python integer, which is measured in 1/10s, and represents the time of waiting for the string from AT command interface, with maximum value of *timeout*.
Return value is a Python integer which is -1 if timeout expired without any data received otherwise is the byte value received. It can be zero.
Example:

b = SER.receivebyte(20)

receives a byte from serial port handling, waiting for it for 2.0 s, assigning return value to b.

2.2.6 SER.readbyte()

Receives a byte from serial port TXD/RXD without waiting for it. No input parameter.
Return value is a Python integer which is -1 if no data received otherwise is the byte value received. It can be zero.
Example:

b = SER.readbyte()

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

2.2.7 SER.set_speed(speed, <char format>)

Sets serial port TXD/RXD speed. Default serial port TXD/RXD speed is 9600. Input parameter *speed* is a Python string which is the value of the serial port speed. It can be the same speeds as the +IPR command.
Note: sending the +IPR command to the device is not affecting the physical serial, when using Python engine you must use this function to set the speed of the port.
Optional Parameter <*char format*> is a Python string that represents the character format to be used: first is the number of bits per char (7 or 8), then the parity setting (N - none, E- even, O- odd) and the number of stop bits (1 or 2). Default value is "8N1".
Return value is a Python integer which is -1 if an error occurred otherwise is 1.
Example:

b = SER.set_speed('115200')

sets serial port speed to 115200, assigning return value to b.

Note: in the PythonWin version pervious to *TelitPy1.5.2+_V2.1.exe* and Python on module version previous to Ver6.03.000 a different syntax is implemented depending of the development environment.

For PythonWin application: SER.SetSpeed(speed) without *char format* parameter.
For Python installed on module: SER.set_speed(speed, char format) with *char format* not an optional parameter.

2.3 GPIO built-in module

GPIO built-in module is the interface between Python core and module internal general purpose input output direct handling. You need to use GPIO built-in module if you want to set GPIO values from Python script and to read GPIO values from Python script.
You can control 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: Python core does not verify if the pins are already used for other purposes (IIC module or SPI module) by other functions, it's the customer responsibility to ensure that no conflict over pins occurs.

If you want to use GPIO built-in module you need to import it first:
import GPIO
then you can use GPIO built-in module methods like in the following example:
a = GPIO.getIOvalue(5)
b = GPIO.setIOvalue(4, 1)
this reads GPIO 5 value and sets GPIO 4 to output with value 1.
More details about GPIO built-in module methods are in the following paragraphs.

2.3.1 GPIO.setIOvalue(GPIOnumber, value)

Sets output value of a GPIO pin. First input parameter *GPIOnumber* is a Python integer which is the number of the GPIO. Second input parameter *value* is a Python integer which is the output value. It can be 0 or 1.
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 return value to b.

2.3.2 GPIO.getIOvalue(GPIOnumber)

Gets input or output value of a GPIO. Input parameter *GPIOnumber* is a Python integer which is the number of the GPIO.
Return value is a Python integer which is -1 if an error occurred otherwise is input or output value. It is 0 or 1.
Example:

a = GPIO.getIOvalue(5)

gets GPIO 5 input or output value, assigning return value to b.

2.3.3 GPIO.setIOdir(GPIOnumber, value, direction)

Sets direction of a GPIO. First input parameter *GPIOnumber* is a Python integer which is the number of the GPIO. Second input parameter *value* is a Python integer which is the output value. It can be 0 or 1. It is only used if *direction* value is 1.

Note: when the *direction* value is 1, although the parameter *value* has no meaning, it is necessary to assign it one of the two possible values: 0 or 1

Third input parameter *direction* is a Python integer which is the direction value. It can be 0 for input or 1 for output.
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 *value* having no meaning, assigning return value to c.

2.3.4 GPIO.getIOdir(GPIOnumber)

Gets direction of a GPIO. Input parameter *GPIOnumber* is a Python integer which is the number of the GPIO.
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.
Example:

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

gets GPIO 7 direction, assigning return value to d.

2.4 MOD built-in module

MOD built-in module is the interface between Python and module miscellaneous functions. You need to use MOD built-in module if you want to generate timers in Python script.

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

import MOD

then you can use MOD built-in module methods like in the following example:
MOD.sleep(15)

this blocks Python script execution for 1.5s.
More details about MOD built-in module methods are in the following paragraphs.

2.4.1 MOD.secCounter()

Returns seconds elapsed since 1 January 1970. This method is useful for timers generation in Python script. No input parameter.
Return value is a Python integer which is the value of seconds elapsed since 1 January 1970.
Example:

a = MOD.secCounter()

returns seconds elapsed since 1 January 1970.

2.4.2 MOD.sleep(sleeptime)

Blocks Python script execution for a given time returning the resources to the system. Input parameter *sleeptime* is a Python integer which is measured in 1/10s and used to block script execution for given value.
No return value.
Example:

MOD.sleep(15)

blocks Python script for 1.5 s.

Note: the parameter *sleeptime* can assume integer values is in the following range [0,32767]

2.4.3 MOD.watchdogEnable(timeout) [1]

Protects system against script blocking by performing automatic reboot of the module when the watchdog reaches determined value. Input parameter *timeout* is an integer, which is measured in seconds and represents time to waiting before executing software restart.
No return value.

Example:

MOD.watchdogEnable(50)

after 50sec from execution of this command module will be rebooted.

2.4.4 MOD.watchdogReset()[1]

Restarts watchdog counter that has been previously activated with the command *MOD.watchdogEnable(timeout)* preventing in this way 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 Python watchdog counter.

2.4.5 MOD.watchdogDisable()[1]

Disables Python watchdog that has been previously activated with the command *MOD.watchdogEnable(timeout).* Python watchdog should be disabled before scripts critical lines such as *import,* since it takes a long time and than enabled again after. No input value.
No return value.

Example:

MOD.watchdogDisable()

Disables Python watchdog.

[1] feature available for the modules with the following Order-Num. : 3990250657, 3990250658, 3990250661, 3990250660, 3990250650, 3990250676

2.4.6 MOD.powerSaving(timeout) [1]

This new feature allows Python to put the system in power saving mode for a certain period or until an external event occurs. Input parameter *timeout* is an integer, which is measured in seconds and represents time for which the Python script remains blocked. Python script will exit power saving mode when the determined value of timeout is reached or after unsolicited signal. If the timeout has negative value Python script will exit from power saving mode only when an external event occurs.
No return value.

Example:

MOD.powerSaving(100)

Python script will exit power saving mode after 100sec or when an external event occurs.

2.4.7 MOD.powerSavingExitCause()[1]

This command can be executed after *MOD.powerSaving(timeout)* and gives the cause of unblocking the Python script. No input parameter.
Return value is a Python integer which is 0 if Python script has exit power saving mode after an external event otherwise it is 1 if Python script has exit power saving mode after the timeout is reached.

Example:

MOD.powerSavingExitCause()

gets the cause of exiting of Python script from the power saving mode

2.5 IIC built-in module

IIC built-in module is an implementation on the Python core of the IIC bus[2] Master (No Multi-Master) using the "bit-banging" technique.
You need to use IIC built-in module if you want to create one or more IIC bus on the available GPIO pins. This IIC bus handling module is mapped on creation on two GPIO pins that will become the Serial Data and Serial Clock pins of the bus. It can be created more than one IIC bus over different pins and the pins used must not be used for other purposes.

Note: Python core does not verify if the pins are already used for other purposes (SPI module or GPIO module) by other functions, it's the customer responsibility to ensure that no conflict over pins occurs.
If you want to use IIC built-in module you need to import it first:
import IIC
then you can create the new bus over the GPIO pins (for example over the pins GPIO3, GPIO4) and then use IIC built-in module methods like in the following example:

```
IICbus = IIC.new(3,4)
IICbus.init()
res = IICbus.send('test')
c = IICbus.sendbyte(0x0d)
d = IICbus.dev_read(114,10)
```

which sends 'test' followed by CR and receives a string of 10 bytes from IIC bus device at address 114, assigning it to d.

Note: you must provide external pull-up on SDA line since the line is working as open collector, on the other hand SCLK is driven with a complete push pull.
More details about IIC built-in module object methods are in the following paragraphs.

2.5.1 IIC.new(SDA_pin, SCL_pin)

Creates a new IIC bus object on the GPIO pins number. Input parameter *SDA_pin*, *SCL_pin* are Python bytes which are the GPIO pin number where the SDA (Serial DAta) and SCL (Serial CLock) lines are mapped.
Return value is the Python custom IIC bus object pointer which then shall be used to interface with the IIC bus created.
Example:

```
bus1 = IIC.new(3,4)
bus2 = IIC.new(5,6)
```

this creates two IIC bus, one over the GPIO3 and GPIO4 and one over the GPIO5 and GPIO6.

[2] With the following clock frequency: 0KHz min, 20KHz typical (idle mode), 100KHz max

Note: available pins for the IIC bus are GPIO1 - GPIO13. The only exception is the module family GM862 where available pins are GPIO3 - GPIO13 while GPIO1 and GPIO2 are used for only input or only output and are not available for IIC bus.

2.5.2 IIC object method: init()

Does the first pin initialisation on the IIC bus previously created.
Return value is a Python integer which is -1 if an error occurred otherwise is 1.
Example:

```
a = bus1.init()
```

2.5.3 IIC object method: sendbyte(byte)

Sends a byte to the IIC bus previously created. Input parameter *byte* is a Python byte which is the byte to be sent to the IIC bus. The start and stop condition on the bus are added by the function.
Return value is a Python integer which is -1 if an error occurred otherwise is 1 the byte has been acknowledged by the slave.
Example:

```
a = bus1.sendbyte(123)
```

sends byte 123 to the IIC bus , assigning return result value to a.

2.5.4 IIC object method: send(string)

Sends a string to the IIC bus previously created. Input parameter *string* is a Python string which is the string to send to the IIC bus.
Return value is a Python integer which is -1 if an error occurred otherwise is 1 if all bytes of the string have been acknowledged by the slave.
Example:

```
a = bus1.send('test')
```

sends string 'test' to the IIC bus , assigning return result value to a.

2.5.5 IIC object method: dev_read(addr, len)

Receives a string of *len* bytes from IIC bus device at address *addr*.
Return value is a Python string which is containing data received.

Example:

a = bus1.dev_read(114,10)

receives a string of 10 bytes from IIC bus device at address 114, assigning it to a.

2.5.6 IIC object method: dev_write(addr, string)

Sends a string to the IIC bus device at address *addr*.
Return value is a Python string which is 1 if data is acknowledged correctly, -1 otherwise.
Example:

a = bus1.dev_write(114,'123456789')

sends the string '123456789' to the IIC bus device at address 114, assigning the result to a.

2.5.7 IIC object method: dev_gen_read(addr, start, len)

Receives a string of *len* bytes from IIC bus device whose address is *addr*, starting from address *start*.
Return value is a Python string which is containing data received.
Example:

a = bus1.dev_gen_read(114,122, 10)
receives a string of 10 bytes from IIC bus device at address 114, starting from address 122 assigning it to a.

2.5.8 IIC object method: dev_gen_write(addr, start, string)

Sends a string to the IIC bus device whose address is *addr*, starting from address *start*.
Return value is a Python string which is 1 if data is acknowledged correctly, -1 otherwise.
Example:

a = bus1.dev_gen_write(114, 112, '123456789')

sends the string '123456789' to the IIC bus device at address 114, starting from address 112, assigning the result to a.

2.6 SPI built-in module

SPI built-in module is an implementation on the Python core of the SPI bus Master using the "bit-banging" technique. You need to use SPI built-in module if you want to create one or more SPI bus on the available GPIO pins.

This SPI bus handling module is mapped on creation on three or more GPIO pins that will become the Serial Data In/Out and Serial Clock pins of the bus, plus a number of optional chip select pins up to 8. It can be created more than one SPI bus over different pins and these pins must not be used for other purposes.

Note: Python core does not verify if the pins are already used for other purposes (IIC module or GPIO module) by other functions, it's the customer responsibility to ensure that no conflict over pins occurs.

If you want to use SPI built-in module you need to import it first:
import SPI
then you can create the new bus over the GPIO pins (for example over the pins GPIO3, GPIO4, GPIO5) and then use SPI built-in module methods like in the following example:

```
SPIbus = SPI.new(3,4,5)
SPIbus.init(0,0)
res = SPIbus.send('test')
c = SPIbus.sendbyte(0x0d)
d = SPIbus.readbyte()
```

sends 'test' followed by CR and receives byte from the SPI bus device.

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

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

Creates a new SPI bus object on the GPIO pins number corresponding. Input parameter *SCLK_pin*, *MOSI_pin* and *MISO_pin* are Python bytes which are the GPIO pin number where the SCLK (Serial CLocK), MOSI (Master Output Slave Input), MISO (Master Input Slave Output) lines are mapped. The same is for the *SS0 .. SS9* which are optional Python bytes which are the GPIO pin number where the corresponding Slave Select line is mapped.
Up to 8 slave select lines can be defined (or none if only 1 slave is used).
Return value is the Python custom SPI bus object pointer which shall be used further to interface with the SPI bus created.

Example:

bus3 = SPI.new(3,4,5)
bus4 = SPI.new(6,7,8,9,10)

creates two SPI bus, one over the GPIO3, GPIO4, GPIO5 and one over the GPIO6, GPIO7, GPIO8, GPIO9, GPIO10 where the GPIO9 is the Slave 0 select and GPIO10 is the Slave 1 select pin.
Note: available pins for the SPI bus are GPIO1 - GPIO13. The only exception is the module family GM862 where available pins are GPIO3 - GPIO13 while GPIO1 and GPIO2 are used for only input or only output and are not available for SPI bus.

2.6.2 SPI object method: init(CPOL, CPHA)

Does the first pin initialization on the SPI bus previously created.
Bus clock polarity is controlled by CPOL value:
CPOL = 0 - clock polarity low
CPOL = 1 - clock polarity high
Bus clock phase transmission is controlled by CPHA value:
CPHA = 0 - data bit is clocked/latched on the first edge of the SCLK.
CPHA = 1 - data bit is clocked/latched on the second edge of the SCLK.
Return value is a Python integer which is -1 if an error occurred otherwise is 1.
Example:

a = bus3.init(0,0)

2.6.3 SPI object method: sendbyte(byte, <SS_number>)

Sends a byte to the SPI bus previously created addressed for the Slave number *SS_number* whose Slave Select signal is activated. Input parameter *byte* is a Python byte which is the byte to be sent to the SPI bus. Optional parameter *SS_number* is a Python byte representing the Slave number to be activated; if not present no slave line is activated.
Return value is a Python integer which is -1 if an error occurred otherwise is 1 the byte has been sent.
Example:

a = bus3.sendbyte(123)

sends byte 123 to the SPI bus , assigning return result value to a.

b=bus4.sendbyte(111,1)

sends byte 111 to the SPI bus activating the Slave Select line of the SS1 device (in our example GPIO10)

2.6.4 SPI object method: readbyte(<SS_number>)

Receives a byte from the SPI bus device at Slave Select number *SS_number*. Input optional parameter *SS_number* is a Python byte representing the Slave number to be activated; if not present no slave line is activated.
Return value is a byte (integer) received from the SPI bus device if no data is received the return value will be zero.

Example:

a = bus3.readbyte()

receives a byte from the SPI bus , assigning return result value to a.

b=bus4.readbyte(1)

receives a byte from the SPI bus device on SS1 line, assigning return result value to b.

2.6.5 SPI object method: send(string, <SS_number>)

Sends a string to the SPI bus previously created. Input parameter *string* is a Python string which is the string to send to the SPI bus. Optional parameter *SS_number* is a Python byte representing the Slave number to be activated; if not present no slave line is activated.
Return value is a Python integer which is -1 if an error occurred otherwise is 1 if all bytes of the string have been sent.
Example:

a = bus3.send('test')

sends string 'test' to the SPI bus , assigning return result value to a.

2.6.6 SPI object method: read(len, <SS_number>)

Receives a string of *len* bytes from SPI bus device at Slave Select number *SS_number*. Input optional parameter *SS_number* is a Python byte representing the Slave number to be activated; if not present no slave line is activated.
Return value is a Python string that contains received data.
Example:

a = bus4.read(10,0)

receives a string of 10 bytes from SPI bus device on SS0 line, assigning it to a.

2.6.7 SPI object method: readwrite(string, len, <SS_number>)

Send the string *string* and contemporaneously receives a string of *len* bytes from SPI bus device at Slave Select number *SS_number*. Optional parameter *SS_number* is a Python byte representing the Slave number to be activated; if not present no slave line is activated.
Return value is a Python string that contains received data.
Example:

a = bus4.readwrite("hello",10,0)

send the string "hello" and receives a string of 10 bytes from SPI bus device on SS0 line, assigning it to a.

3 Executing a Python script

The steps required to have a script running by the python engine of the module are:

- write the python script;
- download the python script into the module NVM;
- enable the python script;
- execute the python script.

3.1 Write Python script

A Python script is a simple text file, it can be written with any text editor but for your convenience a complete Integrated Development Environment (IDE) is included in a software package that Telit provides called *Telit Python Package*.
Remembering the supported features described in 1.6, it is simple to write the script and test it directly from the IDE.
The following is the "Hello Word" short Python script that sends the simplest AT command to the AT command parser, waits for response and then ends.

```
import MDM
print 'Hello World!'
result = MDM.send('AT\r', 0)
print result
c = MDM.receive(10)
print c
```

3.2 Download Python script

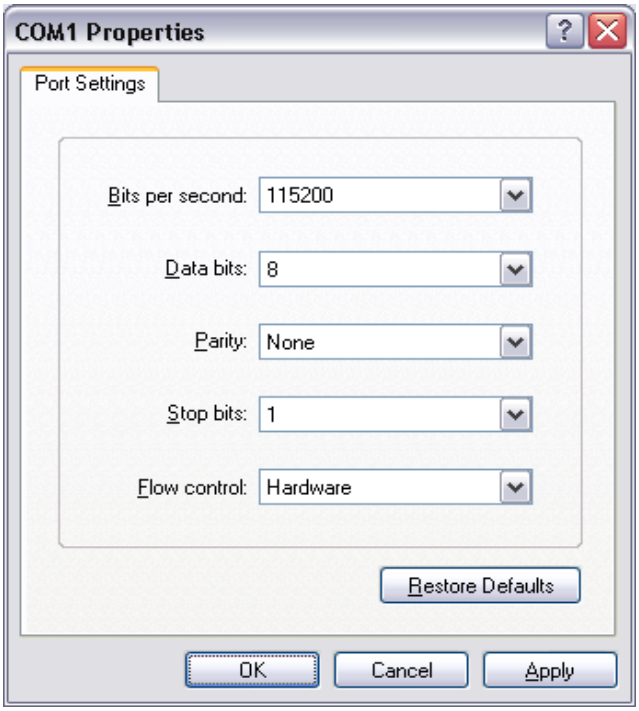
Command: AT#WSCRIPT="< script_name >",< size >,< know-how >

- < script_name >: file name
- < size >: file size (number of bytes)
- < know-how >: know how protection, 1 = on, 0 = off (default)

The script can be downloaded on the module using the #WSCRIPT command. In order to guarantee your company know-how, you have the option to hide the script text so that the #RSCRIPT command does not return the text of the script and keeps it "confidential", you can see only the name of the script with the #LSCRIPT command.
Remember that if you chose to hide the script text it is your responsibility to keep the information about what is executed on the module; for example by naming the script depending from the application and version of the script.
In order to download the script, first you have to choose a name for your script in the module taking care that:

- it must have extension .py;
- the maximum allowed length is 16 characters;
- script name is case sensitive (“Script.py” and “script.py” are two different scripts).

Then you have to find out the exact size in bytes of the script (for example right clicking on the file and selecting “size” in “properties”, attention: not “size on the disc”)
The script download in *Hyper Terminal* is done regardless the previous serial settings at: 115200 baud 8-N-1 with hardware flow control active.

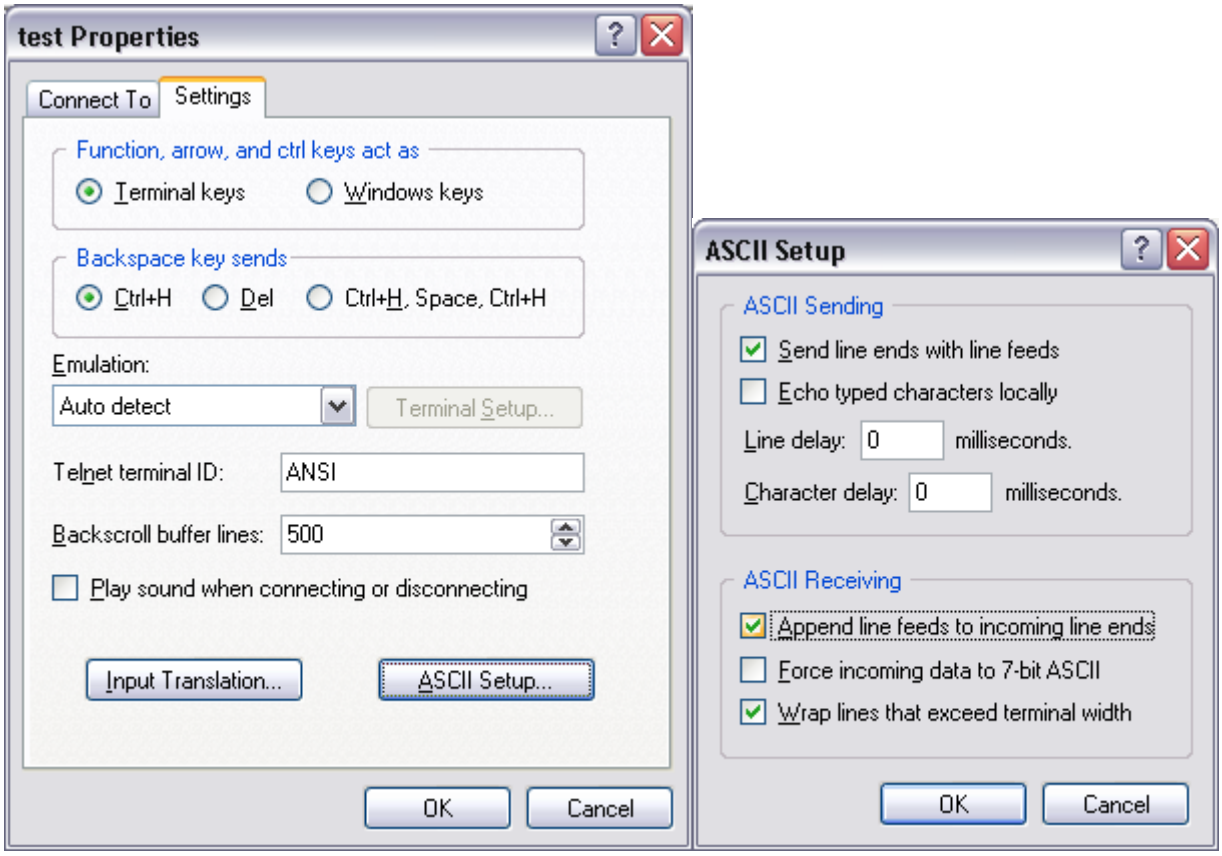


For example:

AT#WSCRIPT=”a.py”,110

wait for the prompt
>>>

and use “Send Text file” with ASCII Setup: “Send line ends with line feeds” and “Append line feeds to incoming line ends” in *HyperTerminal* “Properties” enabled.
Wait for download result: OK or ERROR.



Another way to perform download is: disconnecting in *Hyper Terminal*, when the prompt appears, then right clicking on the file and selecting “download”, when the download ends reconnecting in *Hyper Terminal*.

If instead of *Hyper Terminal* you use *Procomm Plus* application the script text should be sent using "Send File", selecting "Raw ASCII" or "ASCII" as Transfer Protocol. If you use "ASCII" transfer protocol, be sure the options "Expand tabs" and "Expand black lines" are not selected.

3.3 Enable Python script

Command: AT#ESCRIPT="< script_name >"
AT#ESCRIPT?

- < script_name >: file name

Select the Python script which will be executed (the enabled script) from the next start-up and in every future start-up using the AT#ESCRIPT command.
First choose the script you want to enable between the ones you've downloaded:

AT#LSCRIPT? can help you checking the names of the scripts;
AT#ESCRIPT? can help you check 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#ESCRIPT. 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 non having to enabled the same script name every time the script has been changed, deleted and replaced with another script but with the same name.
For example:

AT#ESCRIPT="a.py"

Wait for enable result: OK.

3.4 Execute Python script

The Python script you have 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 -) at start-up, (in this case no AT command interface is connected to the modem port) and if the script name you enabled matches with one of the script names of the scripts you downloaded.

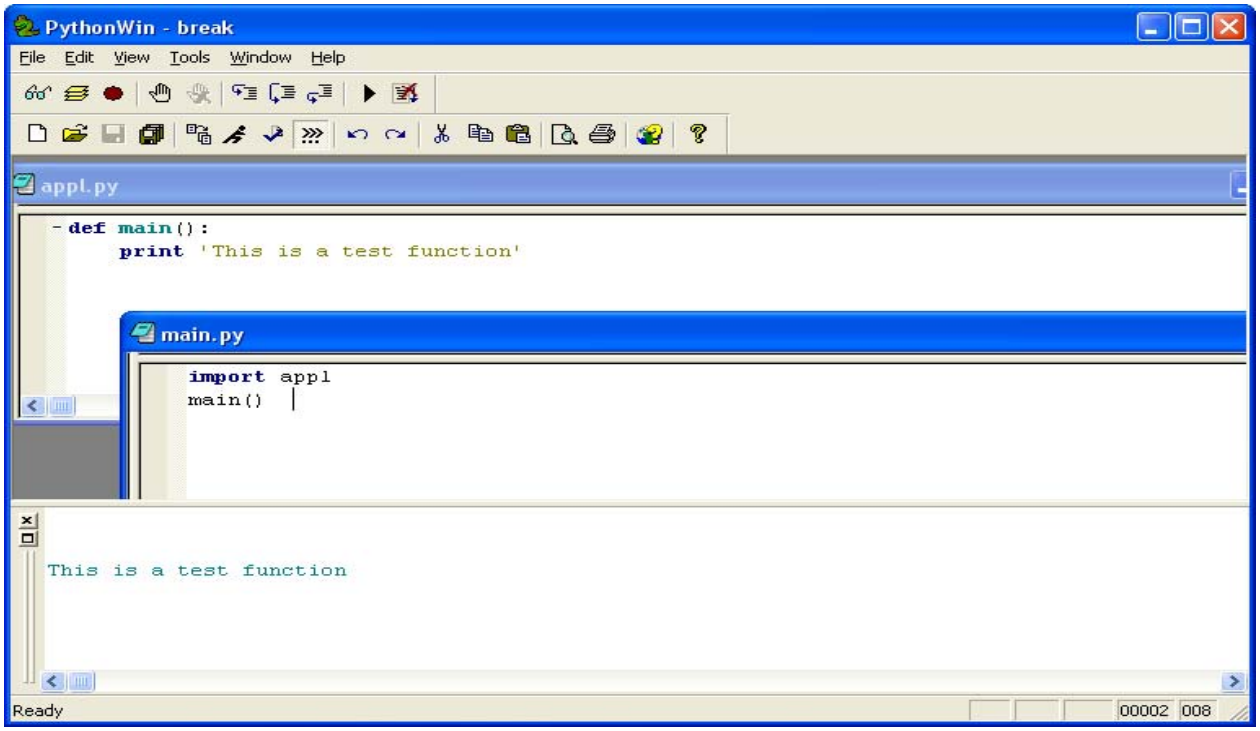
In order to gain again the AT command interface on the modem physical port (for example to update locally a new script) the module shall be powered on with the DTR line HIGH (0V at the module DTR pin) so that the script is not executed and the Python engine is stopped. The real execution of the Python script is delayed from the power on due to the time needed by Python to parse the script. The longer is the script, the longer is this delay.

Note: that only the running script is compiled at run time, all the others that this script may include are compiled once and the compiled result is saved in the NVM as a file with extension .pyo. This delay can be greatly reduced with a simple stratagem:

- type your script normally, and include the main loop in a function, for example "main()", save it to the NVM of the module with a known name, for example appl.py
- write a new script that includes the previous file object, for example "import appl", and this file should call only the main function of the appl.py script, for example main().

In this way the first time the script is executed the imported files will be compiled and the result saved as compiled .pyo files (don't delete them during normal operations, but remember to delete them if you change the corresponding .py script otherwise your changes will not take effect). From the next start-up and in every future start-up the imported files will not be anymore compiled and script execution delay is greatly reduced.

This trick is useful also for long complex scripts, which may run out of memory during compilation; splitting the script into several smaller scripts containing part of the functions/objects definitions will separate the compilation and allow for much bigger script usage.



3.5 Reading Python script

Command: AT#RSCRIPT="< script_name >"

- < script_name >: file name

With the following command AT#RSCRIPT you can read a saved script text. The script text read can be saved using "Capture Text" in HyperTerminal or "Capture File" in *Procomm Plus* application. Port settings should be baud rate 115200bps and hardware flow control.
If know-how protection is activated than AT#RSCRIPT will return only OK: no Python script source code will be returned. In this way nobody will be able to read your Python script from the module.
The Python script will be still in the Python script list and it will be still possible to delete it and to overwrite it.

Example:

AT#RSCRIPT="a.py"

returns Python script source code a.py

3.6 List saved Python scripts

Command: AT#LSCRIPT?

This is a read command that shows the list of the script names currently saved and number of free bytes in memory. No input parameter.

3.7 Deleting Python script

Command: AT#DSCRIPT="< script_name >"

- < script_name >: file name

The Python script can be deleted from the module memory using the #DSCRIPT command.
For example:

AT#DSCRIPT="a.py"

Wait for result: OK.

Note: commands used to write, read and delete script like AT#WSCRIPT, AT#LSCRIPT, AT#RSCRIPT and AT#DSCRIPT can be applied on any type of file, not necessary an executable Python script. For example applied on received data files.

3.8 Restart Python script

Command: AT#REBOOT

This is an execution command that causes the restart of the module and execution of the active script on the start-up.

3.9 Debug Python script

The debug of the active Python script can be done both on the emulated environment of the *Telit Python Package* (refer to its documentation) or directly on the target with the second serial port pin EMMI TX (actually a not translated RS232 serial port as the RXD pin).

Connect to the module serial port EMMI TX at 115200 8-N-1 with hardware flow control active.
Now you can see all Python outputs to stdout and stderr:

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

The Telit GM862-GPS and GE863-GPS have the second serial port pin EMMI TX used for continuous direct output of GPS NMEA sentences that’s why there is another procedure to follow for debugging of the Telit GPS modules. There are two ways to perform direct debugging: activate SSC port or use CMUX.

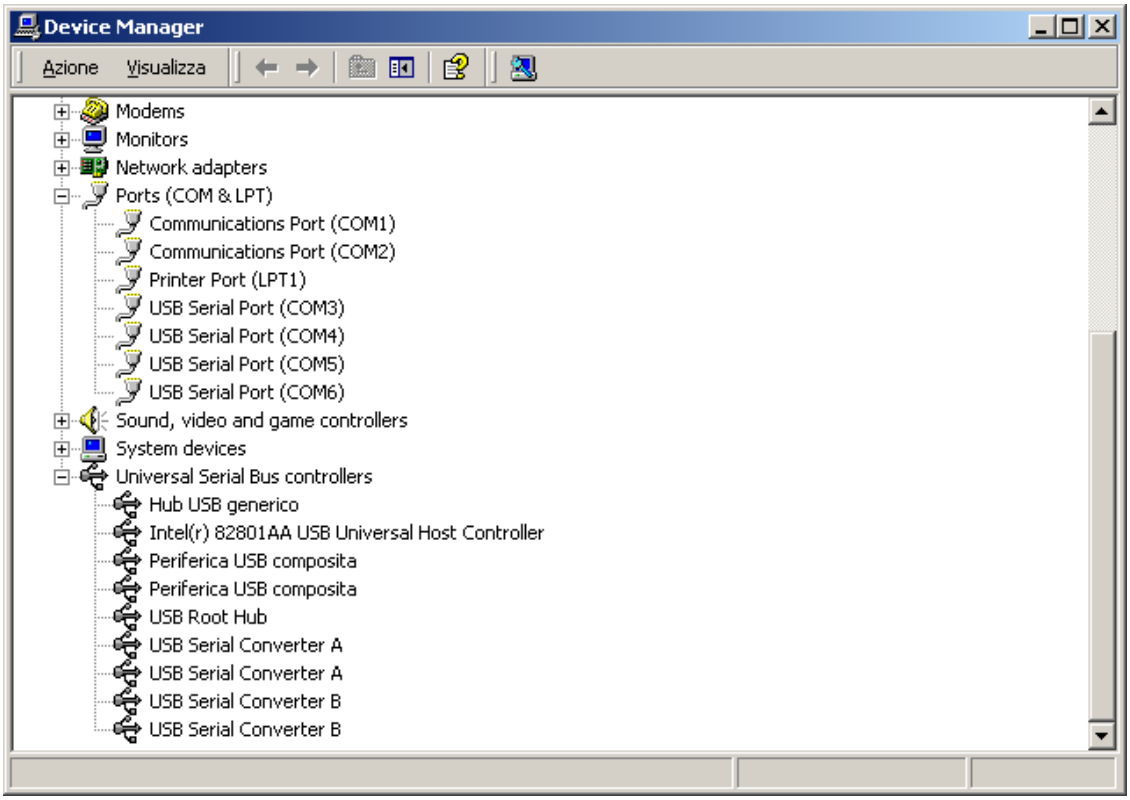
3.9.1 Debug Python script on GPS modules using SSC bus

SSC (Serial Synchronous Controller) port can be configured to be compatible to the SPI Interface, available via 4 GPIO pins. In this case the Python debug data will be read from the USB port placed on the EVK2.
Note: for the direct debug of GPS modules a software version starting from 7.02.001 is needed

3.9.1.1 Installation of the drivers

Before starting the process of debug the drivers should be installed in the following way:

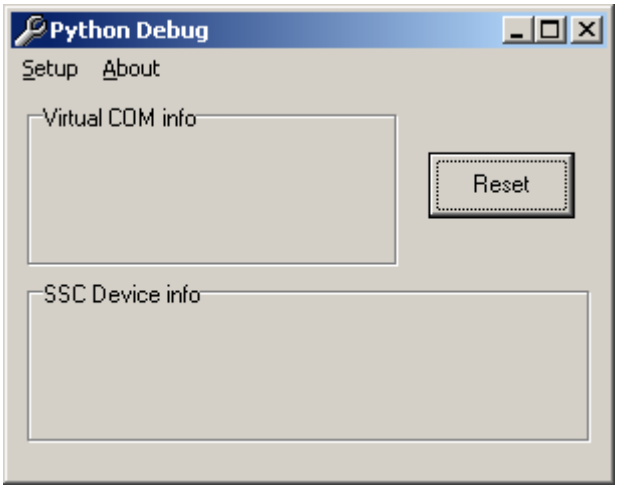
- Download the FTDI drivers and the installation guide in order to use the USB port placed on the EVK2 (http://www.ftdichip.com/Drivers/D2XX.htm)
- Save the drivers (unzipped) on the PC
- After connecting USB cable with PC and USB port placed on the EVK2 (that has been powered on): the installation procedure should start, according to the installation guide instructions
- When the installation is concluded you will have four new COM ports (see Control Panel – System – Hardware – Device Manager) and one not visible SSC port



- close any application controlling the serial ports and install the *Python Debug* application (please contact our technical support to get *Python Debug* application*)*

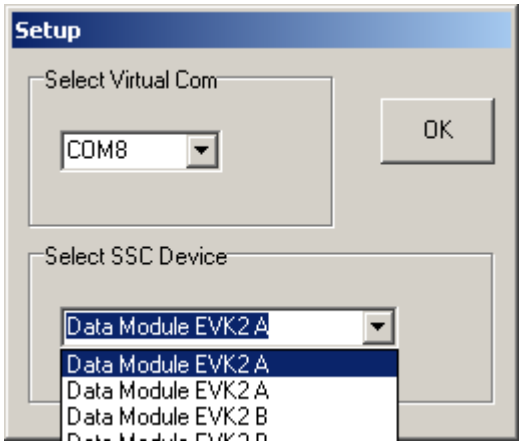
Note: if an error messages appears during the installation, it will be necessary to close any application controlling the serial ports

- the following box should appear when you run the *Pythondebug.exe* for the first time:

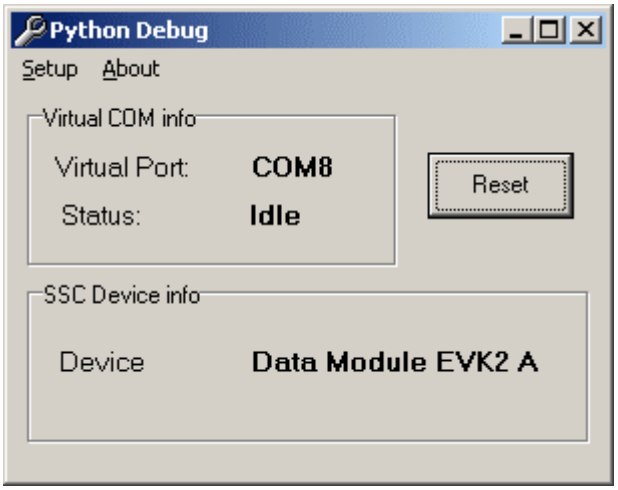


- Select the Setup option.

- Then select a Virtual COM, different from the other COM ports preferably (“COM8” in the figure), and associate to it the first SSC device that appearing in the list (“Data Module EVK2A” in the figure),



- the following figure should appear:



Note: If the PC uses the EVK2 RS232 upper port (ASC0) to send AT commands, remember to put all jumpers to set RS232 mode. This will not effect reading of Python debug data from the USB port

3.9.1.2 Debugging process

After the successful installation of the drivers process of direct debugging of the Telit GPS modules can start. The steps are the following:

- Switch on the module and activate the SSC output with the following AT command: AT#SSCTRACE=1
- Download and enable your Python script, then power OFF the module.
- to be sure that DTR input to the module is HIGH disconnect the RS232 cable from the module side (i.e. RS232 DTR on the modem serial port is LOW);
- check if you have the USB cable connected between the USB port of the PC and the USB port placed on the EVK2
- every time before you power ON the module you have to click on the *Reset* button in the *Python Debug* application (necessary to reactivate the association between the virtual Com port and the SSC device)
- Run a terminal emulator application (e.g. *Hyper Terminal*) to trace the activity of the Python script, with the following setting:

connected COM	virtual Port set in *Python Debug* (COM8 in the example)
Bit rate	115 200
Data bits	8
Parity	No parity
Stop bit	1
Flow control	Hardware

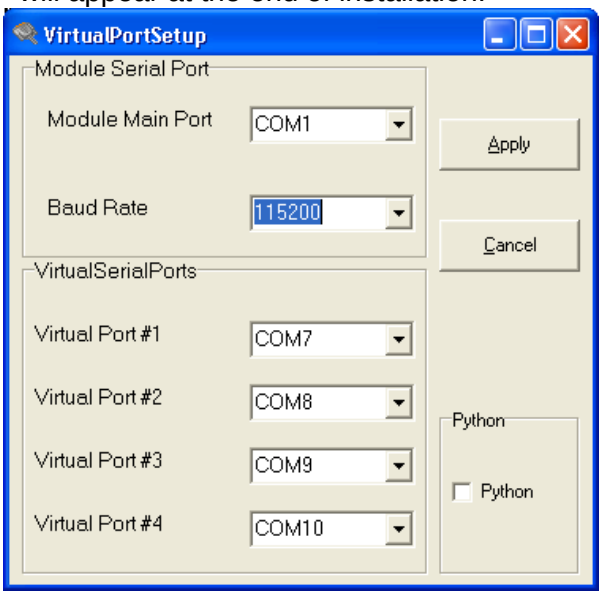
- Power ON the module and you should see the script starting and the debug info appearing on the terminal emulator window.
- If the debug strings do not appear on the screen: power OFF the module, check again if USB cable is correctly connected, reset the *Python Debug* application, than power ON the module and run the terminal emulator application with the same settings as before.

3.9.2 Debug Python script on GPS modules using CMUX

CMUX (Converter-Multiplexer) is a multiplexing protocol implemented in the Telit module that can be used to send data, SMS, fax, TCP data. The Multiplexer mode enables one serial interface to transmit data to four different customer applications. This is achieved by providing four virtual channels using a Multiplexer (Mux).
With activating of the CMUX feature debugging data can be received on the serial ASC0 port mounted on EVK2.
Note: for the direct debug of GPS modules a software version starting from 7.02.X01 is needed.

3.9.2.1 Installation

- Install the *Telit Serial Port Mux* ver 1.08-B001[3] application on your PC. A box similar to this will appear at the end of installation:



- Select the baud rate and then click on the Apply button

3.9.2.2 Debugging process

Note: If the PC uses the EVK2 RS232 upper port (ASC0) to send AT commands, remember to put all jumpers to set RS232 mode.

[3] please contact our technical assistance to get the latest application version

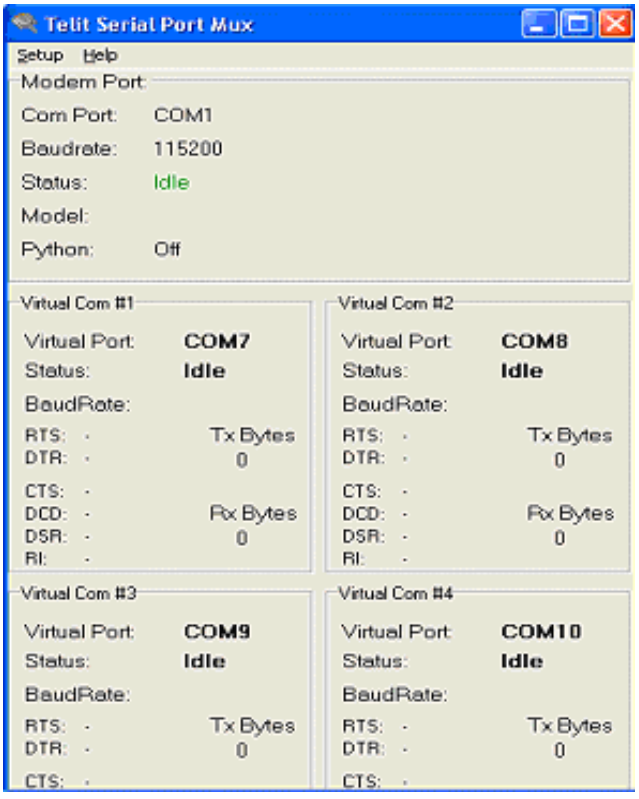
- Switch ON the module and with a terminal emulator (e.g. *Hyper Terminal*) and send the following commands to the module:

  AT#SSCTRACE=0 disabled SSC output

  AT#CMUXSCR=1,<bitrate> activated the CMUX feature on the module; put the desired bit rate (e.g. 115200)

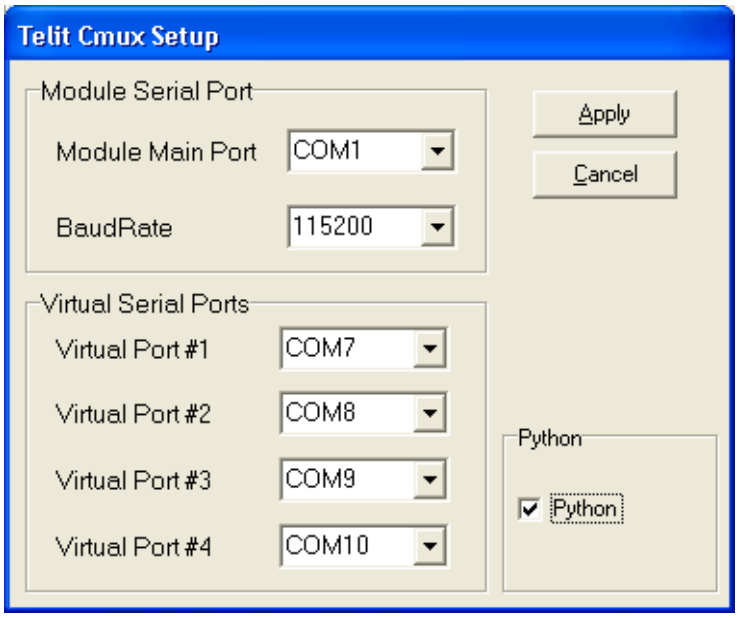
  AT#STARTMODESCR=1,10 module waits for minimum 10 seconds (recommended value; can be changed) and if there is no AT commands sent in this period (except AT<Enter>) start the enabled Python script, regardless of the DTR status (low or high).

- Download and enable[4] your Python script, then power OFF the module.
- Close any application controlling the serial ports (e.g. *Hyper Terminal*)
- Run the *Telit Serial Port Mux* ; a figure similar to the one below will appear:



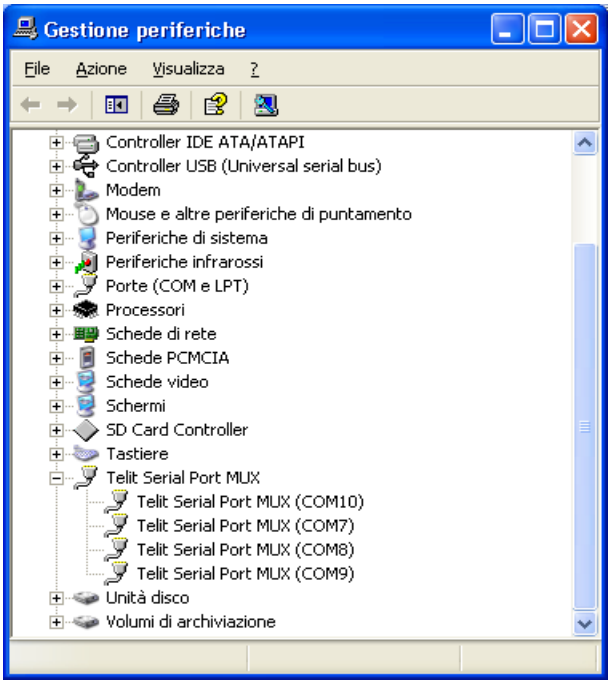
[4] follow the procedure of download and enable of the Python script reported in the paragraph 3.2 and 3.3

Control if the Setup options are the following:



Set the *Module Main Port* as the real COM port you have available (e.g. COM1 in the figure), check the Python box and then select the Apply button.

- After this step, you will have 4 new *Telit Serial Port Mux* ports (see *Control Panel – System – Hardware – Device Manager*) as in the figure below:



- Run a terminal emulator application (e.g. *Hyper Terminal*) to trace the activity of the Python script, with the following setting:

connected COM	virtual port #4 set in Telit CMUX window (COM10 in the figure)
Bit rate	115 200
Data bits	8
Parity	No parity
Stop bit	1
Flow control	Hardware

In the *Telit Serial Port Mux* window, “Status:” of the Virtual Port#4, after establishing connection in *Hyper Terminal,* will change from *Idle* to *Opened*

- Power on the module and wait for at least 10 seconds without sending any AT command (except AT<Enter>);
  In the *Telit Serial Port Mux* window, “Status:” of the *Modem Port:* will change in the following way (before 10 seconds expired):
  Idle → cycle between Connecting and Error → Connected

  After 10 seconds you should see the script starting and the debug info appearing on the terminal emulator window.

- If an ERROR messages appears in the Virtual Port #1,2,3,4 boxes, close any application controlling the serial ports and then restart the Telit CMUX application. If this procedure is not sufficient to avoid ERROR message, reset the PC, run again *Telit Serial Port Mux* with the same settings and repeat the procedure as described above.
- If you need to debug the same Python application again, then:
  - Disconnect the terminal emulator application (eg. *Hyper Terminal*) from the Virtual Port#4 (in this case COM10)
  - "Status:" of the Virtual Port#4 in the *Telit Serial Port Mux* window, should change from Opened to Idle
  - Switch off the module
  - Connect the terminal emulator application to Virtual Port#4 (in this case COM10)
  - "Status:" of the Virtual Port#4 in the *Telit Serial Port Mux* window[5], should change from Idle to Opened
  - Switch on the module and wait for the "Status:" of the *Modem Port* in the *Telit Serial Port Mux* window to go connected

[5] If the *Telit Serial Port Mux* application seems to be freezed, please consider that it becomes active after the module is switched on.

4 List of acronyms

Abbreviation	Description
ACM	Accumulated Call Meter
ASCII	American Standard Code for Information Interchange
AT	Attention Commands
CB	Cell Broadcast
CBS	Cell Broadcasting Service
CCM	Call Control Meter
CLIP	Calling Line Identification Presentation
CLIR	Calling Line Identification Restriction
CMOS	Complementary Metal-Oxide Semiconductor
CR	Carriage Return
CSD	Circuit Switched Data
CTS	Clear To Send
DAI	Digital Audio Interface
DCD	Data Carrier Detected
DCE	Data Communications Equipment
DRX	Data Receive
DSR	Data Set Ready
DTA	Data Terminal Adaptor
DTE	Data Terminal Equipment
DTMF	Dual Tone Multi Frequency
DTR	Data Terminal Ready
EMC	Electromagnetic Compatibility
ETSI	European Telecommunications Equipment Institute
FTA	Full Type Approval (ETSI)
GPRS	General Radio Packet Service
GPIO	General Purpose Input Output
GSM	Global System for Mobile communication
HF	Hands Free
IIC	Inter Integrated Circuit
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
IRA	Internationale Reference Alphabet
ITU	International Telecommunications Union
IWF	Inter-Working Function
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LF	Linefeed

Abbreviation	Description
ME	Mobile Equipment
MISO	Master Input Slave Output
MMI	Man Machine Interface
MO	Mobile Originated
MOSI	Master Output Slave Input
MS	Mobile Station
MT	Mobile Terminated
NVM	Non Volatile Memory
NMEA	National Marine Electronics Association
OEM	Other Equipment Manufacturer
PB	Phone Book
PDU	Protocol Data Unit
PH	Packet Handler
PIN	Personal Identity Number
PLMN	Public Land Mobile Network
PUCT	Price per Unit Currency Table
PUK	PIN Unblocking Code
RACH	Random Access Channel
RLP	Radio Link Protocol
RMS	Root Mean Square
RTS	Ready To Send
RI	Ring Indicator
SCA	Service Center Address
SCL	Serial CLock
SDA	Serial DAta
SIM	Subscriber Identity Module
SMD	Surface Mounted Device
SMS	Short Message Service
SMSC	Short Message Service Center
SPI	Serila Protocol Interface
SS	Supplementary Service
SSC	Synchronous Serial Controllers
TIA	Telecommunications Industry Association
UDUB	User Determined User Busy
USSD	Unstructured Supplementary Service Data

5 Document Change Log

Revision	Date	Changes
ISSUE#0	21/03/06	Release First ISSUE#1
ISSUE#1	13/09/06	1.4 Python Implementation Description: added SPI and IIC libraries that were missing on the graphic 2.4 MOD built-in module: added Python watchdog and power saving mode 2.5 IIC built-in module: added note for the IIC bus clock frequency 3.9 Debug Python Script: new paragraph for GPS modules - clarified meaning of parameter timeout for the following commands: MDM.receive(timeout), SER.receive(timeout) and SER.receivebyte(timeout)