

Easy Script in Python

80000ST10020a Rev.3 - 24/05/07



Contents

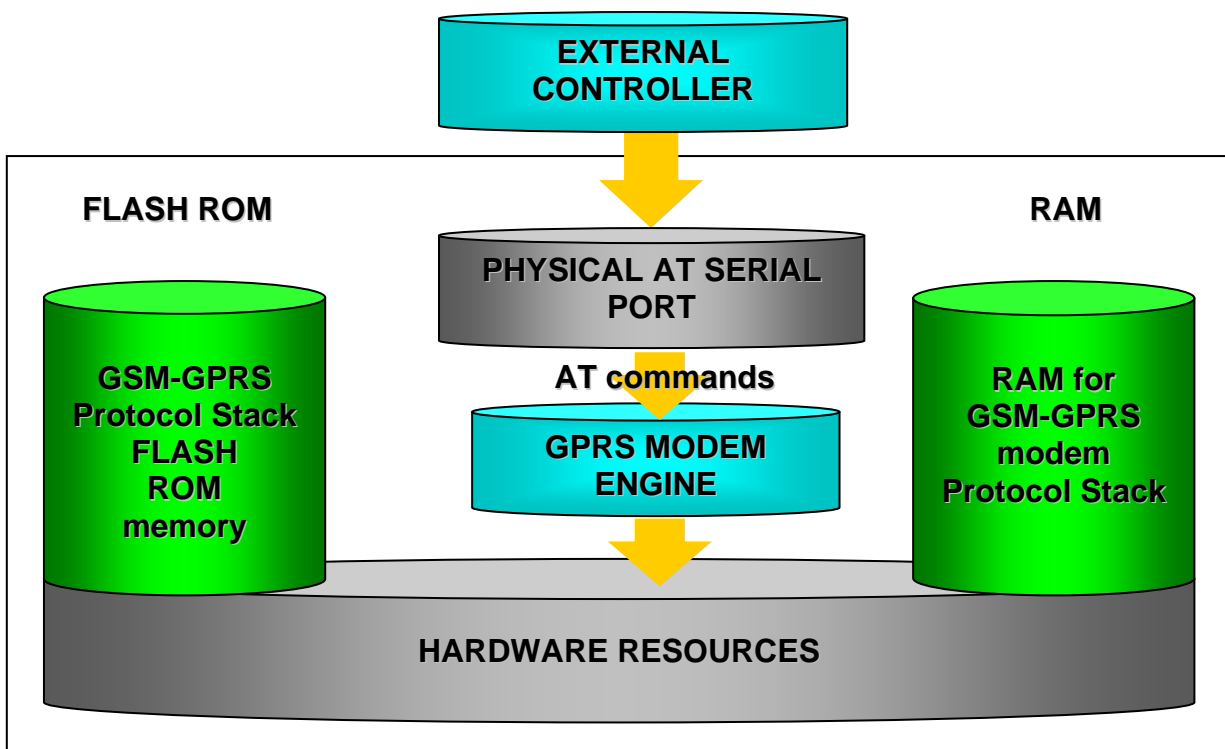
- 1 Easy Script Extension - Python interpreter 7**
 - 1.1 Overview.....7**
 - 1.2 Python 1.5.2+ Copyright Notice9**
 - 1.3 Python installation.....10**
 - 1.4 Python implementation description11**
 - 1.5 Introduction to Python.....13**
 - 1.5.1 Data types 13
 - 1.5.2 Operators..... 14
 - 1.5.3 Compound statements..... 14
 - 1.5.4 Conditional execution 15
 - 1.5.5 Loops 15
 - 1.5.6 Resources 16
 - 1.6 Python core supported features.....17**
- 2 Python Build-in Custom Modules 18**
 - 2.1 CMUX and Python18**
 - 2.2 MDM built-in module19**
 - 2.2.1 MDM.send(string, timeout) 19
 - 2.2.2 MDM.receive(timeout) 19
 - 2.2.3 MDM.read()..... 20
 - 2.2.4 MDM.sendbyte(byte, timeout)..... 20
 - 2.2.5 MDM.receivebyte(timeout) 20
 - 2.2.6 MDM.readbyte() 21
 - 2.2.7 MDM.getDCD() 21
 - 2.2.8 MDM.getCTS()..... 21
 - 2.2.9 MDM.getDSR() 21
 - 2.2.10 MDM.getRI() 22
 - 2.2.11 MDM.setRTS(RTS_value)..... 22
 - 2.2.12 MDM.setDTR(DTR_value) 22
 - 2.3 MDM2 built-in module23**
 - 2.3.1 MDM2.send(string, timeout) 23
 - 2.3.2 MDM2.receive(timeout) 24
 - 2.3.3 MDM2.read()..... 24
 - 2.3.4 MDM2.sendbyte(byte, timeout)..... 24
 - 2.3.5 MDM2.receivebyte(timeout) 25
 - 2.3.6 MDM2.readbyte() 25
 - 2.3.7 MDM2.getDCD() 25
 - 2.3.8 MDM2.getCTS()..... 25
 - 2.3.9 MDM2.getDSR() 26
 - 2.3.10 MDM2.getRI() 26
 - 2.3.11 MDM2.setRTS(RTS_value)..... 26
 - 2.3.12 MDM2.setDTR(DTR_value) 26



1 Easy Script Extension - Python interpreter

1.1 Overview

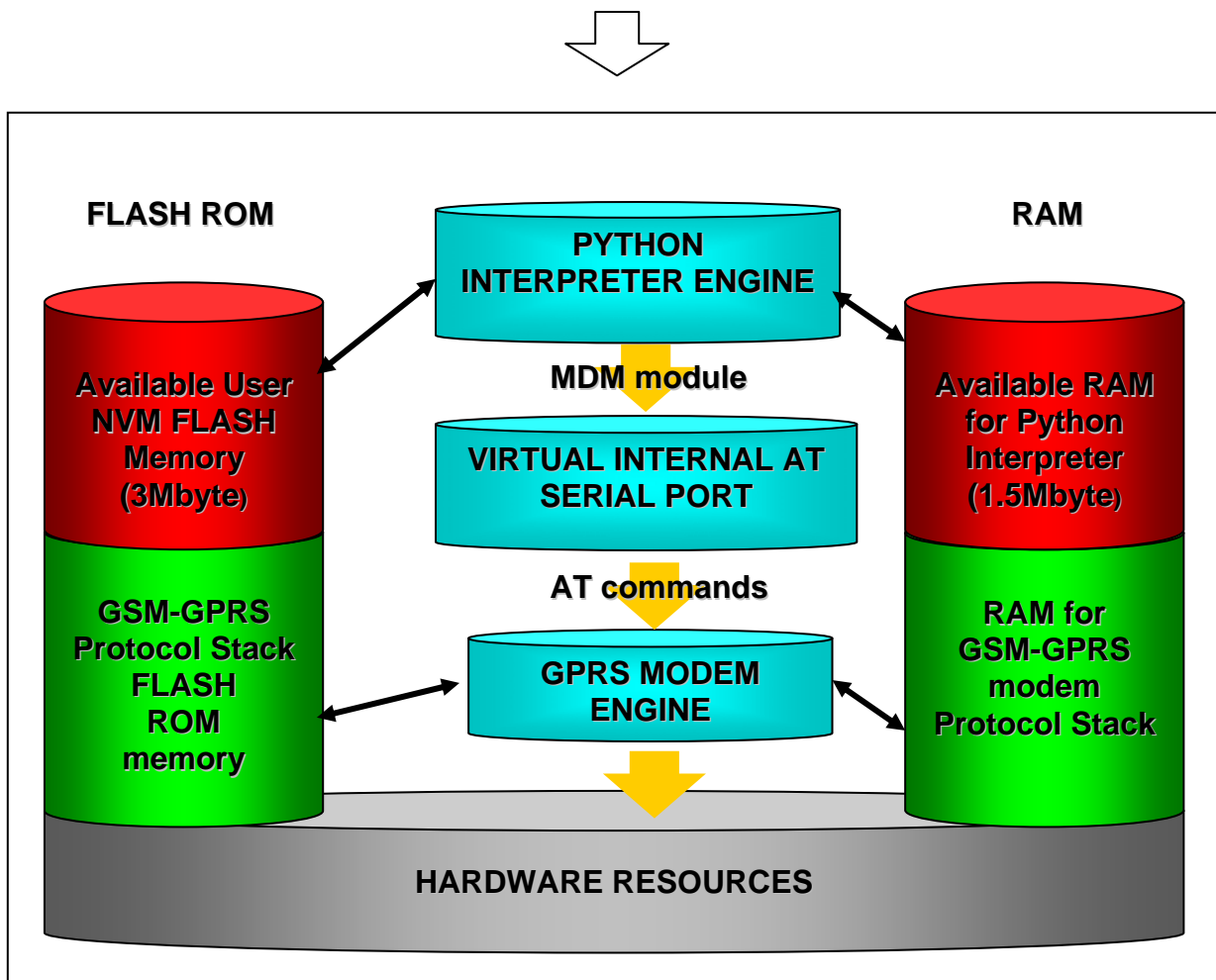
The Easy Script Extension is a feature that allows driving the modem "internally", writing the controlling application directly in a nice high level language: Python. The Easy Script Extension is aimed for the application usually done by a small microcontroller that managed some I/O pins and the module 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
- 1.2 MB / 1.5 MB ¹ RAM reserved for Python engine usage

A schematic of this approach is:



¹ Available only for the products with the following Order-Num.: 3990250655, 3990250656, 3990250653 and 3990250654

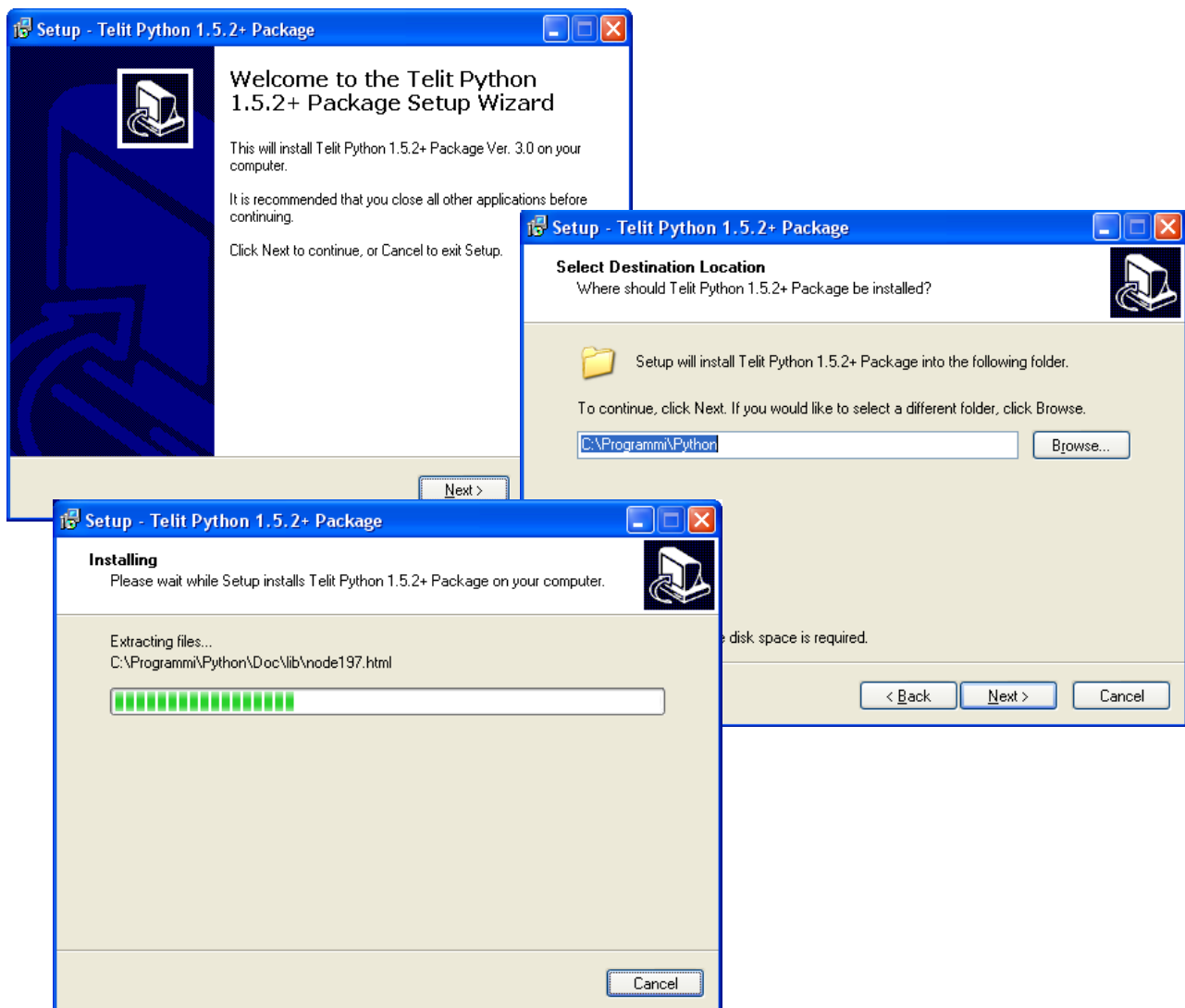


1.3 Python installation

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.



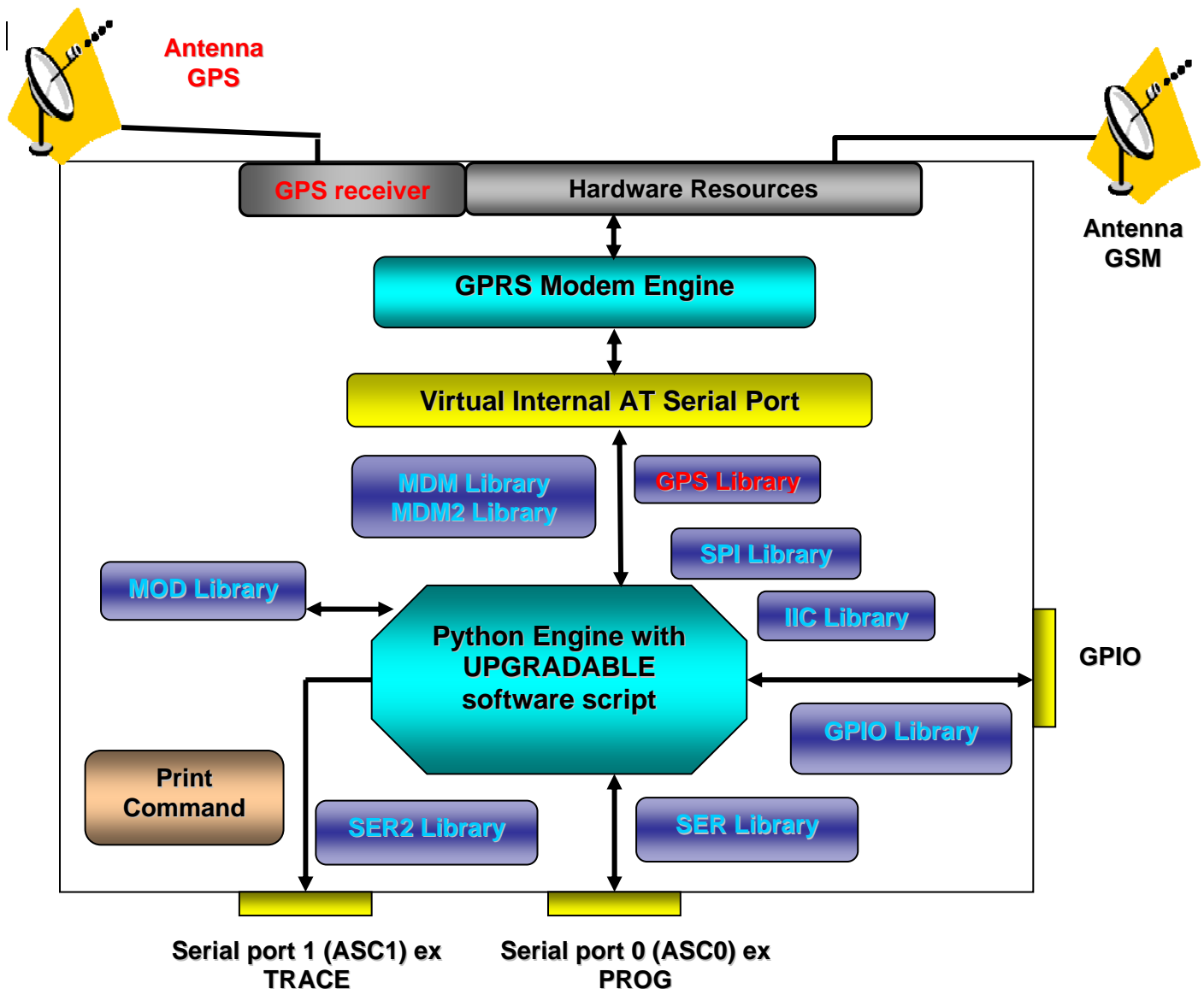
1.4 Python implementation description

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.



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



2.2 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.

For the default start configuration echo (ATE0) is disabled and long form (verbose) is 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 can be found in the following paragraphs.

2.2.1 MDM.send(string, timeout)

This command 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.

2.2.2 MDM.receive(timeout)

This command 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 maximum time of waiting for the string from AT command interface. Return value is a Python string which is an empty string if *timeout* has expired without any data received otherwise the string contains data received.



Example:

```
dsr = MDM.getDSR()
```

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

2.2.10 MDM.getRI()

This command gets Ring Indicator (RI) from AT command interface. It has no input parameter. 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 = MDM.getRI()
```

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

2.2.11 MDM.setRTS(RTS_value)

This command 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.

2.2.12 MDM.setDTR(DTR_value)

This command 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.



2.3.2 MDM2.receive(timeout)

This command 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 maximum time of waiting for the string from AT command interface. Return value is a Python string which is an empty string if *timeout* has expired without any data received otherwise the string contains data received.

Example:

```
a = MDM2.receive(15)
```

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

2.3.3 MDM2.read()

This command 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 the string contains data received in the moment when command is activated.

Example:

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

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

2.3.4 MDM2.sendbyte(byte, timeout)

This command sends a byte to AT command interface. First input parameter *byte* is any Python byte that will be to send to AT command interface. It can also be zero. Second input parameter *timeout* is a Python integer which is the value in 1/10 s to wait for the byte to be sent to AT command interface before timeout expires. Waiting time is caused by flow control. Return value is a Python integer which is -1 if timeout expired otherwise is 1.

Example:

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

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




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

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

2.3.9 MDM2.getDSR()

This command gets Data Set Ready (DSR) from AT command interface. It has no input parameter. Return value is a Python integer which is 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 return value to dsr.

2.3.10 MDM2.getRI()

This command gets Ring Indicator (RI) from AT command interface. It has no input parameter. 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 return value to ri.

2.3.11 MDM2.setRTS(RTS_value)

This command sets Request to Send (RTS) in AT command interface. 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. It has no return value.

Example:

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

sets RTS to ON in AT command handling.

2.3.12 MDM2.setDTR(DTR_value)

This command sets Data Terminal Ready (DTR) in AT command interface. Input parameter *DTR_value* is a Python integer which is 0 if setting DTR to set to OFF or 1 if setting DTR to set to ON. It has no return value.



2.4 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). SER built-in module has also been improved lately with the possibility to control physical lines.

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 can be found in the following paragraphs.

2.4.1 SER.send(string)

This command sends a string to the serial port TXD/RXD. Input parameter *string* is a Python string that will be send to serial port ASC1.

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 4096 bytes⁴

⁴ For the products with the following Order-Num. 3390250655, 3390250656, 3390250653 and 3390250654 the available buffer is 2048 bytes



2.4.2 SER.receive(timeout)

This command 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 maximum time of waiting for the string from the serial port ASC0.

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 ASC0 handling, waiting for it for 1.5 s, assigning return value to a.

2.4.3 SER.read()

This command receives a string from serial port TXD/RXD without waiting for it. It has 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 ASC0 handling, assigning return value to a.

NOTE: the buffer available for the SER.receive(timeout) and SER.read() commands is 4096 bytes⁵.

2.4.4 SER.sendbyte(byte)

This command sends a byte to serial port TXD/RXD. Input parameter *byte* is any Python byte that will be send to serial port. It can also 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 ASC0 handling, assigning return value to b.

⁵ For the products with the following Order-Num. 3390250655, 3390250656, 3390250653 and 3390250654 the available buffer is 256 bytes



2.4.5 SER.receivebyte(timeout)

This command 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 maximum time of waiting for the string from serial port ASC0.

Return value is a Python integer which is -1 if timeout expired without any data received otherwise is the byte value received. It can also be zero.

Example:

```
b = SER.receivebyte(20)
```

receives a byte from serial port ASC0 handling, waiting for it for 2.0 s, assigning return value to b.

2.4.6 SER.readbyte()

This command receives a byte from serial port TXD/RXD without waiting for it. It has no input parameter.

Return value is a Python integer which is -1 if no data received otherwise is the byte value received. It can also be zero.

Example:

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

receives a byte from serial port ASC0 handling, assigning return value to b.

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

This command 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')
```



2.4.11 SER.setRI(RI_value)

This command sets Ring Indicator (RI) in serial port ASC0. Input parameter *RI_value* is a Python integer which is 0 if RI is set to OFF or 1 if RI is set to ON. It has no return value.

Example:

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

sets RI to ON in ASC0.

2.4.12 SER.getRTS()

This command gets Request to Send (RTS) from serial port ASC0. It has no input parameter. Return value is a Python integer which is 0 if RTS is set to OFF or 1 if RTS is set to ON.

Example:

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

gets RTS from ASC0, assigning return value to rts.

2.4.13 SER.getDTR()

This command gets Data Terminal Ready (DTR) from serial port ASC0. It has no input parameter. Return value is a Python integer which is 0 if DTR is set to OFF or 1 if DTR is set to ON.

Example:

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

gets DTR from ASC0, assigning return value to dtr.



2.5 SER2 built-in module⁶

SER2 built-in module is the interface between Python and mobile internal serial port ASC1 direct handling. It is used when you want to send data from Python script to serial port ASC1 and receive data from serial port ASC1 to Python script.

SER2 built-in module is available only for non-GPS products. When SER2 built-in module is imported, ASC1 will not be available for trace and debug, in order to have these functionalities you should activate CMUX on ASC0.

If you are using CMUX on ASC0 (AT#CMUXSCR=1) then fourth CMUX port will be available for trace and debug.

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

```
import SER2
```

then you can use SER2 built-in module methods like in the following example:

```
a = SER2.send('test')
b = SER2.sendbyte(0x0d)
c = SER2.receive(10)
```

which sends 'test' followed by <CR> and receives data waiting for one second.

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

2.5.1 SER2.send(string)

This command sends a string to the serial port ASC1. Input parameter *string* is a Python string that will be send to serial port ASC1.

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

Example:

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

sends string 'test' to serial port ASC1 handling, assigning return value to a.

2.5.2 SER2.receive(timeout)

This command receives a string from serial port ASC1 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 maximum time of waiting for the string from the serial port ASC1.

⁶ feature available for the modules with the following Order-Num. : 3990250657, 3990250658, 3990250661, 3990250660, 3990250650, 3990250676



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 = SER2.receive(15)
```

receives a string from serial port ASC1 handling, waiting for it for 1.5 s, assigning return value to a.

2.5.3 SER2.read()

This command receives a string from serial port ASC1 without waiting for it. It has 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 = SER2.read()
```

receives a string from serial port ASC1 handling, assigning return value to a.

NOTE: the buffer available for the SER2.receive(timeout) and SER2.read() commands is 4096 bytes.

2.5.4 SER2.sendbyte(byte)

This command sends a byte to serial port ASC1. Input parameter *byte* is any Python byte that will be send to serial port ASC1. It can also be zero. Return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

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

sends byte 0x0d, that is <CR>, to serial port ASC1 handling, assigning return value to b.

2.5.5 SER2.receivebyte(timeout)

This command receives a byte from serial port ASC1 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 maximum time of waiting for the string from serial port ASC1.

Return value is a Python integer which is -1 if timeout expired without any data received otherwise is the byte value received. It can also be zero.



Example:

```
b = SER2.receivebyte(20)
```

receives a byte from serial port ASC1 handling, waiting for it for 2.0 s, assigning return value to b.

2.5.6 SER2.readbyte()

This command receives a byte from serial port ASC1 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 also be zero.

Example:

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

receives a byte from serial port ASC1 handling, assigning return value to b.

2.5.7 SER2.set_speed(speed, <format>)

This command sets serial port ASC1 speed. Default serial port ASC1 speed is 9600. First input parameter *speed* is a Python string which is the value of the serial port speed. It can be in the range from '300' to '115200'. Second input parameter format is optional and is a Python string which is the serial port format. It can be '8N1', '8N2', '8E1', '8O1', '7N1', '7N2', '7E1', '7O1', or '8E2'. Return value is a Python integer which is -1 if an error occurred otherwise is 1.

Example:

```
b = SER2.set_speed('115200')
```

sets serial port ASC1 speed to 115200, assigning return value to b.



2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 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.

2.6.4 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.



2.7 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.

2.7.1 MOD.secCounter()

Returns seconds elapsed since 1 January 2000. This method is useful for timers generation in Python script. No input parameter.

Return value is a Python long integer which is the value of seconds elapsed since 1 January 2000.

AT+CCLK command allows to read and set current date and time.

Here are some useful constants:

- 1 day = 86400 seconds
- 1 year = 31536000 or 31622400 seconds
- 30 years from 1 January 1970 to 1 January 2000 = 946684800 seconds
(simply add this constant if you need seconds elapsed since 1 January 1970)

Example:

```
a = MOD.secCounter()
```

returns seconds elapsed since 1 January 2000.

2.7.2 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]

2.7.3 MOD.watchdogEnable(timeout)⁷

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.

2.7.4 MOD.watchdogReset()⁷

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.

2.7.5 MOD.watchdogDisable()⁷

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 then enabled again after. No input value. No return value.

Example:

```
MOD.watchdogDisable()
```

Disables Python watchdog.

⁷ feature available for the modules with the following Order-Num. : 3990250657, 3990250658, 3990250661, 3990250660, 3990250655, 3990250676



2.7.6 MOD.powerSaving(timeout)⁷

This new feature allows Python to put the system in power saving mode⁸ for a certain period or until an external event⁹ occurs in the same way as AT command AT+CFUN=0 does. 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.

2.7.7 MOD.powerSavingExitCause()⁷

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

⁸ ATTENTION: when the script debugging is activated the module does not enter in power saving mode

⁹ an external event in this case can be: URC unsolicited message (ex. RING of incoming calls) or putting RTS high (when it goes back low the power saving mode remains disabled)



Easy Script in Python

80000ST10020a Rev.3 - 24/05/07

This creates one IIC bus over the lines SDA=GPIO3 and SCL=GPIO4 with default address 0x50. If no address defined, the address anyway can be feed with the *readwrite()* function as first part of the string.

NOTE: available pins for the IIC bus are GPIO1 - GPIO13. The only exception is the module family GM862 where available pins are GPIO2 - GPIO13 where GPIO2 must be used only for output.

2.8.2 IIC object method: init()

This command does the first pin initialisation on the IIC bus previously created. It has no input parameter. Return value is a Python integer that is -1 if an error occurred otherwise is 1.

Example:

```
status = myIIC1.init()
```

2.8.3 IIC object method: readwrite(string, read_len)¹¹

This command can send a string or receive a string of *read_len* bytes from from IIC bus device at address *addr*. First input parameter *string* is Python character while the second parameter *read_len* is a Python integer used in process of reading data from the IIC bus and can assume values in the range from 0 to 254.

Return value is a Python string which contains received data.

Example of use:

#start of I2C example for a particular I2C device

```
import IIC
import MOD
```

```
I2C_SCL = 12    # GPIO used for SCL pin
I2C_SDA = 4     # GPIO used for SDA pin
I2C_ADDR = 0x50 # myIIC1 address
NUM_REGS = 8    # max # of registers
```

```
myIIC1 = IIC.new(I2C_SDA, I2C_SCL, I2C_ADDR)
status = myIIC1.init()
#MOD.sleep(5)
```

```
print ' Writing from ADDR = 0x08, DATA = "Telit" '
if myIIC1.readwrite('\x08'+ 'Telit',0) == -1:
    print 'Error acknowledged'
#MOD.sleep(5)
```

¹¹ NOTE: it is not possible to perform at the same time reading and writing of the string



```
ret = myIIC1.readwrite('\x08', 14) # Random read
print ' RANDOM READ FROM ADDR = 0x08, 14 bytes: %s' % ret
#MOD.sleep(5)
```

```
print ' Writing DATA= "hello!" from ADDR = 0x00'
myIIC1.readwrite('\x00'+ 'hello!', 0)
#MOD.sleep(5)
```

```
print ' SETTING CURRENT ADDRESS = 0x00'
if myIIC1.readwrite('\x00',0) == -1:
    print 'Error acknowledged'
#MOD.sleep(5)
```

```
ret = myIIC1.readwrite("", 22) # Current address read
print ' CURRENT ADDR READ = 0x00, 22 bytes: %s' % ret
MOD.sleep(5)
```

```
ret = myIIC1.readwrite('\x00', 254) # Current address read
print ' read 254 bytes with readwrite: %s' % ret
```

2.8.4 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:

Supposing to set:

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

then:

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

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

2.8.5 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.

2.8.6 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.

2.8.7 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.

2.8.8 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.



2.8.9 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.



2.9 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
```

the functions to use are: create your SPI object (new), initialize (init) and transfer data (readwrite) like in the example:

```
mySPIobject = SPI.new(3, 4, 5, 6)
mySPIobject.init(0, 0, 0, 0)
d = mySPIobject.readwrite('test', 4)
```

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

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

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

This command creates a new SPI bus object on the corresponding GPIO pins. Input parameter SCLK_pin, MOSI_pin and MISO_pin are Python bytes that represent the GPIO pin number where the SCLK (Serial CLock), MOSI (Master Output Slave Input), MISO (Master Input Slave Output) lines are mapped.

Input parameter SS_i_line is a mandatory Python byte if the SPI device needs to be selected by Slave Select line: it is the GPIO pin number where the SS_i (ith Slave Select) line is mapped. Up to 8 Slave Select lines can be defined.

The SS_i_pin is optional because not all SPI devices have a Slave Select line, otherwise named Chip Select (CS) line.



Return value is the Python custom SPI object pointer that will be used further to interface with this specific SPI object created.

Example:

```
mySPIobject = SPI.new(6, 7, 8, 9, 10)
```

Creates an SPI object “mySPIobject” where SCLK=GPIO6, MOSI=GPIO7, MISO=GPIO8 and SS0=GPIO9, SS1=GPIO10

NOTE: available pins for the SPI bus are GPIO1 - GPIO13. The only exception is the module family GM862 where available pins are GPIO3 - GPIO13.

2.9.2 SPI object method: init (CPOL, CPHA, <SSPOL>, <SS>)

This command performs the initialization on the SPI bus previously created, and can be reused as many time as necessary if some of its parameters need changes during work.

First input parameter *CPOL* represents clock polarity and is controlled in the following way:

- *CPOL* = 0 - clock polarity low
- *CPOL* = 1 - clock polarity high

Second input parameter *CPHA* represents clock phase transmission and is controlled in the following way:

- *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.

The combinations of polarity and phases are often referred to as SPI modes.

Third parameter *SSPOL* is optional and represents the Slave Select Polarity and can assume the following values:

- *SSPOL* = 0 - polarity low (default)
- *SSPOL* = 1 - polarity high

Fourth parameter *SS* is optional and represents the Default Slave Select line number to use among the already defined slave select (SS) lines for this SPI object and then it can assume values from 0 to 7.

- *SS* = unused - means that if not SS settled in *readwrite()* parameter’s function, no SS will be moved.
- *SS* = 0...7 – Defined the default SS line to move if not SS settled in *readwrite()* parameter’s function.

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

Example:

```
status = mySPIobject.init(0, 0, 0)
```



In this initialization no SS line is defined as default, and no SS line will be moved if not set in *readwrite* function.

```
status = mySPIobject.init(0, 0, 0, 1)
```

In this initialization the SS=1 refers to the use of SS1, already defined in *SPI.new(6, 7, 8, 9, 10)* as GPIO10.

```
status = mySPIobject.init(0, 0, 0, 0)
```

In this initialization the SS=0 refers to the use of SS0, already defined in *SPI.new(6, 7, 8, 9, 10)* as GPIO9.

2.9.3 SPI object method: *readwrite(string, <read_len>, <SS>)*

This command sends *string* and receives *read_len* bytes at the same time from SPI bus device at Slave Select line number *SS*.

First input parameter *string* is a Python string while the second optional¹² parameter *read_len* is a Python integer used only for reading data and can assume values in the range from 0 to 254.

The third optional input parameter *SS*, if defined, selects which of *SS* line number defined in *SPI.new* will be used and, if not defined, the default Slave Select line number in *SPI.init*, if any, will be used.

Return value is a Python string that contains (*read_len bytes*) of sent and/or received data, in case an error occurs return value will be -1 or NULL if is memory error.

Example:

```
myString = mySPIobject.readwrite('hello', 10)
```

send the string "hello" and receives a string of 10 bytes from the SPI device, assigning it to *myString*

Example of writing and reading of a memory in a particular SPI device (addressable by a Slave Select pin)

```
#start of SPI example
import SPI
import MOD
import MDM
import GPIO
```

```
CMD_WRITE = '\x02'    #this value is not the value for any SPI device, but for tested one only!
CMD_READ = '\x03'    #this value is not the value for any SPI device, but for tested one only!
CMD_RESET = '\xC0'
REG_ADDR_X = '\x0E'  #this value is not the value for any SPI device, but for tested one only!
REG_ADDR_Y = '\x0F'  #this value is not the value for any SPI device, but for tested one only!
```

¹² In case of read operation this parameter is obligatory



```
TWO_READ_ACCESS = '\x00\x00' # string of the length equal to the number of bytes to receive

MySPI1 = SPI.new (3,10,8,6) # (SCLK, MOSI, MISO, SS0)
MySPI1.init (0,0,0,0)      # (CPOL, CPHA, SSPOL, SS)

#Power Up, Reset and Clock ON routine for the SPI device can also be implemented from outside

MySPI1.readwrite (CMD_WRITE + REG_ADDR_Y + '\xAC') # write 0xAC in REG_ADDR_Y

myString = MySPI1.readwrite (CMD_READ + REG_ADDR_X, 4)
print "DATA VALUE AT ADDR_X and _Y =",hex(ord(myString[2]))+' and '+ hex(ord(myString[3]))
# values of myString [0] and myString [1] correspond to the status output while writing

myString = MySPI1.readwrite(CMD_READ + REG_ADDR_X + TWO_READ_ACCESS,4)
# does the same as previous: this is to maintain the backward compatibility with the past version of the
readwrite method

ret = MySPI1.readwrite (CMD_READ, 129) # read first byte (cmd) plus 128 bytes starting from the last
position

i = 0
print "STATUS =",hex(ord(ret[i]))
while (i < 128):
    print "REGISTER[" ,hex(i), "]:",hex(ord(ret[i + 1]))
    i = i + 1

#end of SPI example
```

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

NOTE: We advice to use the new *readwrite* (2.9.3) method to send bytes or strings.

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)

2.9.5 SPI object method: `readbyte(<SS_number>)`

NOTE: We advice to use the new *readwrite* (2.9.3) method to send bytes or strings.

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.

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

NOTE: We advice to use the new *readwrite* (2.9.3) method to send bytes or strings.

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.

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

NOTE: We advice to use the new *readwrite* (2.9.3) method to send bytes or strings.

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.



2.10GPS¹³ built-in module

GPS built-in module is the interface between Python and mobile 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 first:

```
import GPS
```

After this you can start using GPS built-in module 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.

2.10.1 GPS. powerOnOff(newStatus)

This module powers ON/OFF GPS controller in the same way as AT command: AT\$GPSP.

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.

2.10.2 GPS.getPowerOnOff()

This module gets GPS controller current power ON/OFF status. It has no input parameter.

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 return value to status.

¹³ Available only for modules with following Order-Num.: 3990250657 and 3990250660



2.10.3 GPS.resetMode(mode)

This module resets GPS controller in the same way as AT command: AT\$GPSR. Input parameter *mode* is a Python integer and can have the following values:

- 0 for Hardware reset
- 1 for Coldstart (No Almanac, No Ephemeris);
- 2 for Warmstart (No Ephemeris);
- 3 for Hotstart (with stored Almanac and Ephemeris).

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

Example:

```
res = GPS.resetMode(1)
```

executes a cold restart of GPS controller, assigning return value to res.

2.10.4 GPS.getAntennaVoltage()

This module gets GPS antenna supply voltage in the same way as AT command: AT\$GPSAV. It has no input parameter.

Return value is a Python integer which represents antenna supply voltage in mV.

Example:

```
mV = GPS.getAntennaVoltage()
```

gets GPS antenna supply voltage, assigning return value to mV.

2.10.5 GPS.getAntennaCurrent()

This module gets GPS antenna current consumption in the same way as AT command: AT\$GPSAI. It has no input parameter.

Return value is a Python integer which represents antenna current consumption in mA.

Example:

```
mA = GPS.getAntennaCurrent()
```

gets GPS antenna current consumption, assigning return value to mA.

2.10.6 GPS.getActualPosition()

This module gets GPS last position information stored in the same way as with AT command: AT\$GPSACP. It has no input parameter.



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 return value to gga.

2.10.10 GPS.getLastGLL()

This command gets GPS last GLL NMEA sentence stored. It has no input parameter. 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 return value to gll.

2.10.11 GPS.getLastGSA()

This command gets GPS last GSA NMEA sentence stored. It has no input parameter. 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 return value to gsa.

2.10.12 GPS.getLastGSV()

This command gets GPS last GSV NMEA sentence stored. It has no input parameter. 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 return value to gsv.



2.10.13 GPS.getLastRMC()

This command gets GPS last RMC NMEA sentence stored. It has no input parameter. 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 return value to rms.

2.10.14 GPS.getLastVTG()

This command gets GPS last VTG NMEA sentence stored. It has no input parameter. 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 return value to vtg.

2.10.15 GPS.getPosition()

This command gets GPS last position stored in numeric format. It has no input parameter. Return value is a Python tuple formatted in the following way <latitude, latNorS, longitude, lonEorW> where:

- First element of tuple is a Python integer which is latitude in (degrees * 10000000), that is in degrees with 10000000 scale factor.
- Second element of tuple in a Python string which is 'N' for north or 'S' for south.
- Third element of tuple is a Python integer which is longitude in (degrees * 10000000), that is in degrees with 10000000 scale factor.
- Fourth element of tuple in a Python string which is 'E' for east or 'W' for west.

If GPS controller has no position information the following tuple is returned:
(0, "", 0, "").

Example:

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

gets last position stored, assigning return value to pos.



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)



Easy Script in Python

80000ST10020a Rev.3 - 24/05/07

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.

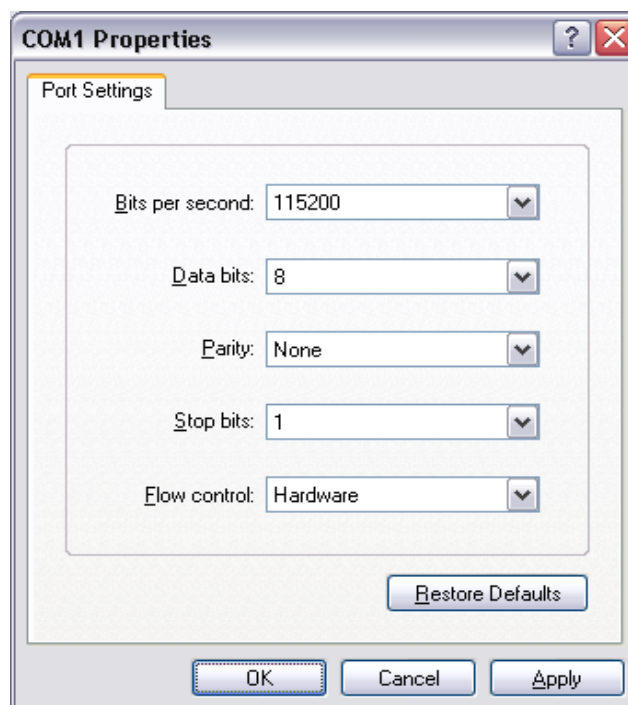
Remember that if you chose to hide the script text it is your responsibility to keep the information about what is executed on the module; for example by naming the script depending from the application and version of the script.

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:

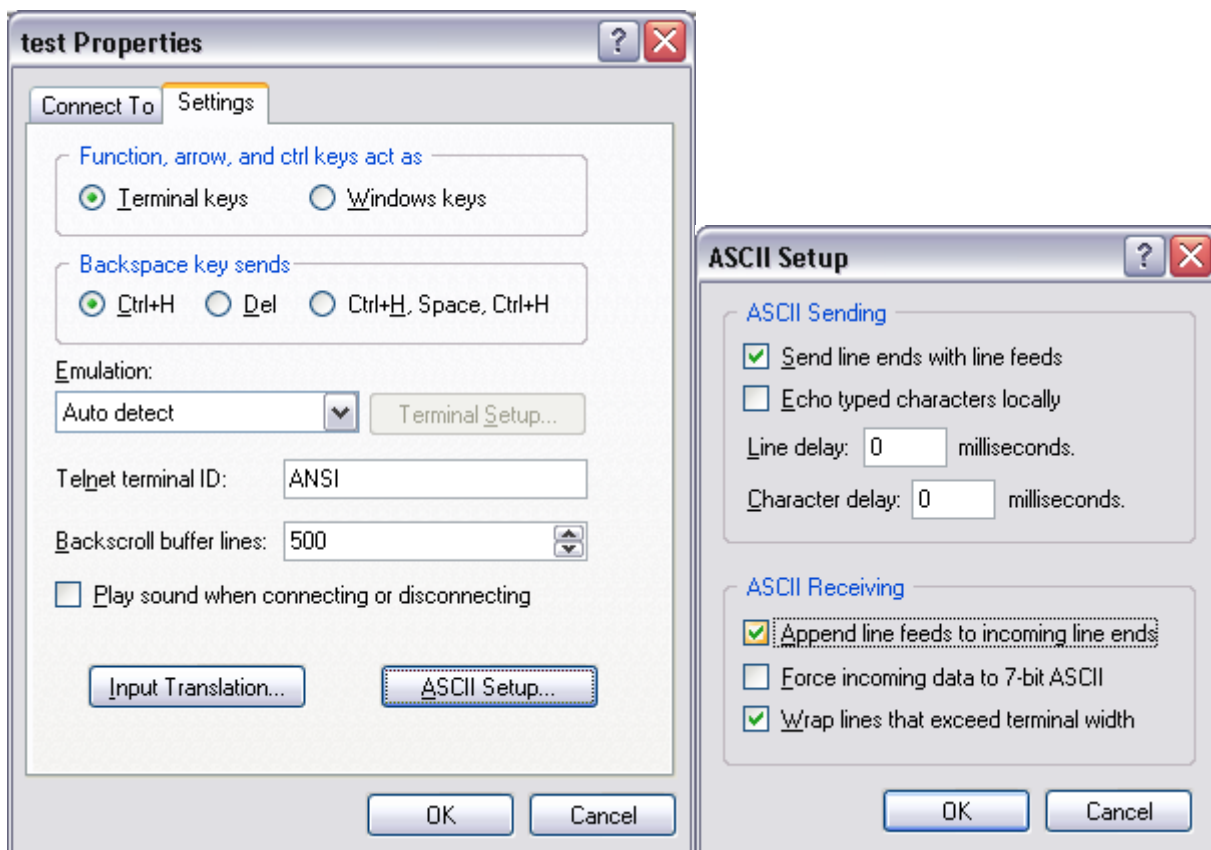
```
AT#WSCRIPT="a.py",110
```

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

- < 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#ESCRIP command.

First choose the script you want to enable between the ones you’ve downloaded:

AT#LSCRIP? can help you checking the names of the scripts;

AT#ESCRIP? 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#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 non 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.

For example:

AT#ESCRIP="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.

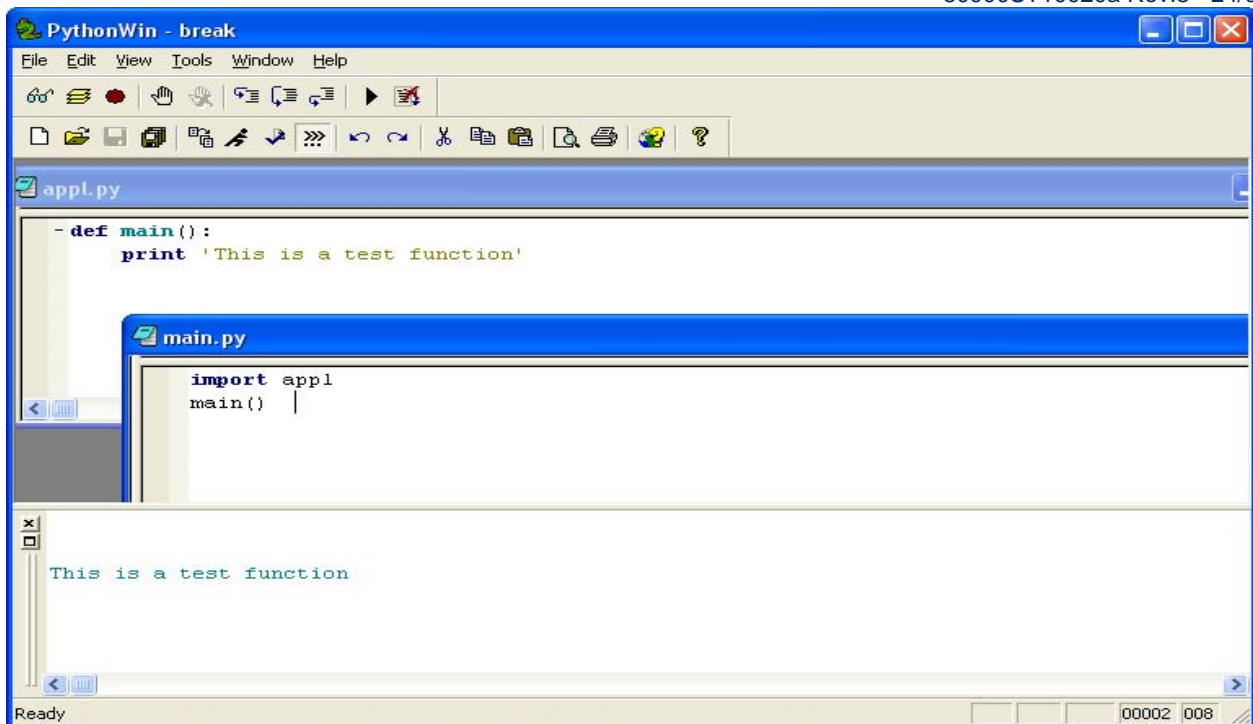
NOTE: that only the running script is compiled at run time, all the others that this script may include are compiled once and the compiled result is saved in the NVM as a file with extension .pyo. This delay can be greatly reduced with a simple stratagem:

- 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.

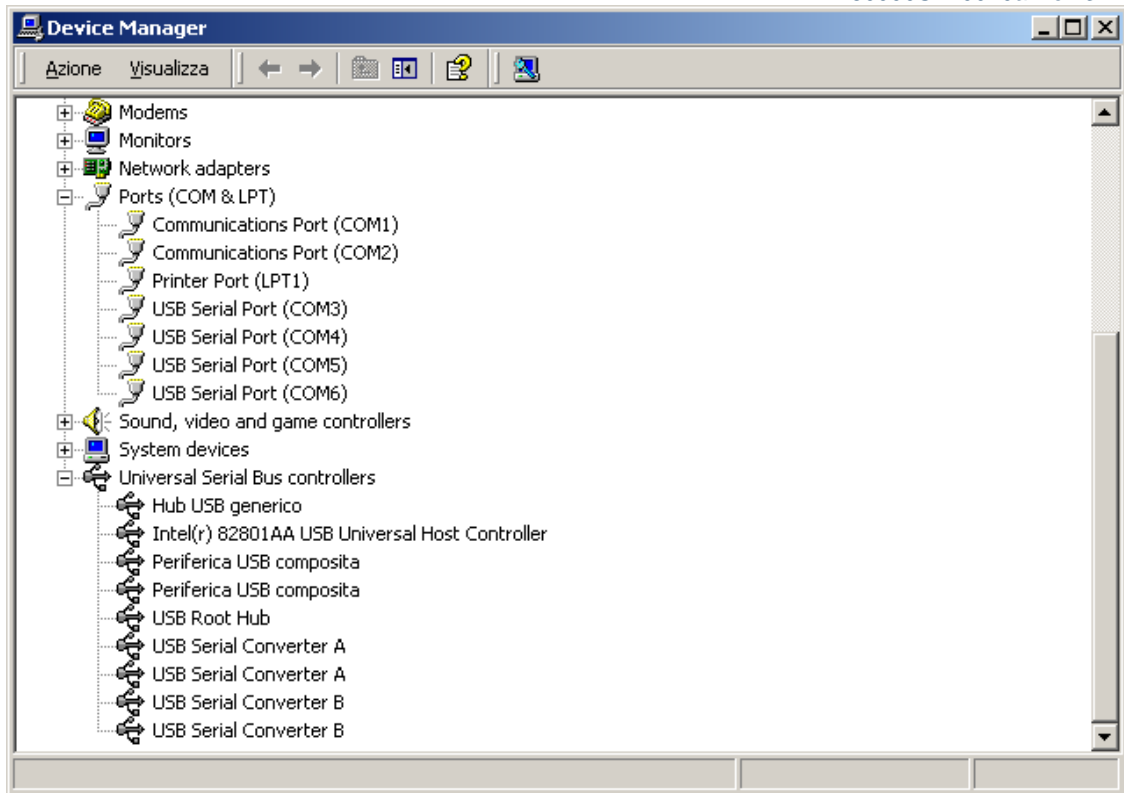
NOTE: commands used to write, read and delete script like AT#WSCRIPT, AT#LSCRIPT, AT#RSCRIPT and AT#DSCRIPT can be applied on any type of file, not necessary an executable Python script. For example applied on received data files.

3.8 Restart Python script

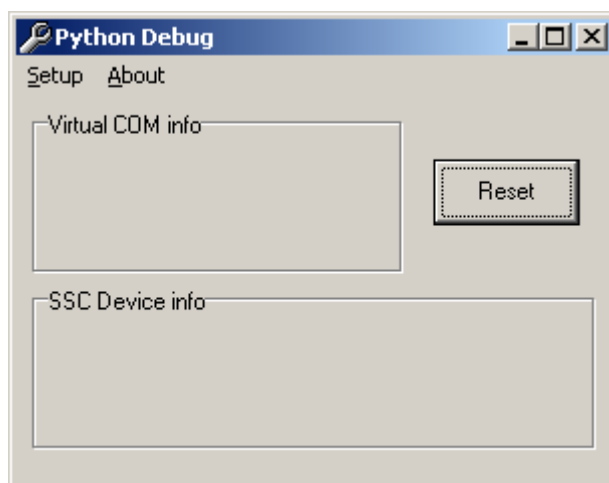
Command: AT#REBOOT

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





- 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:



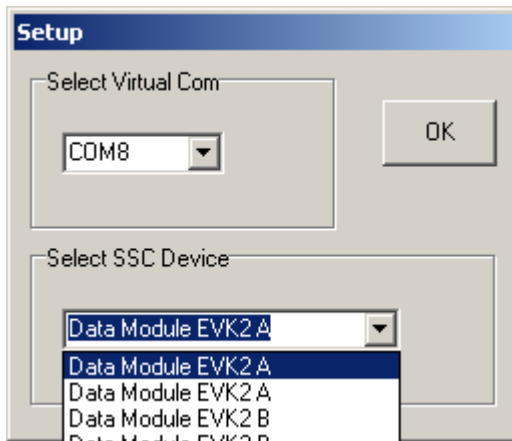
- Select the Setup option.



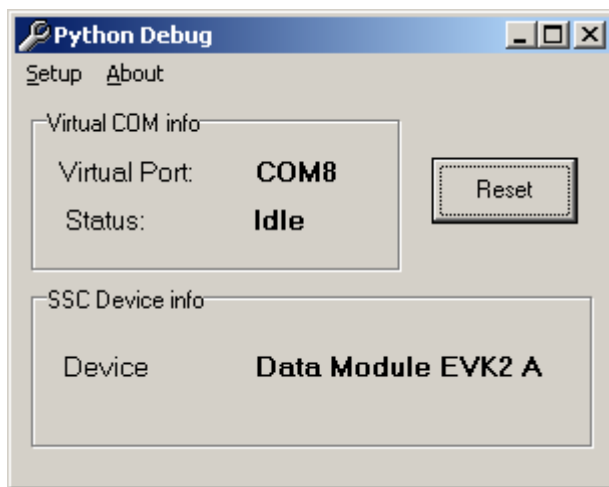
Easy Script in Python

80000ST10020a Rev.3 - 24/05/07

- 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



3.9.2 Debug Python script on GPS modules using CMUX

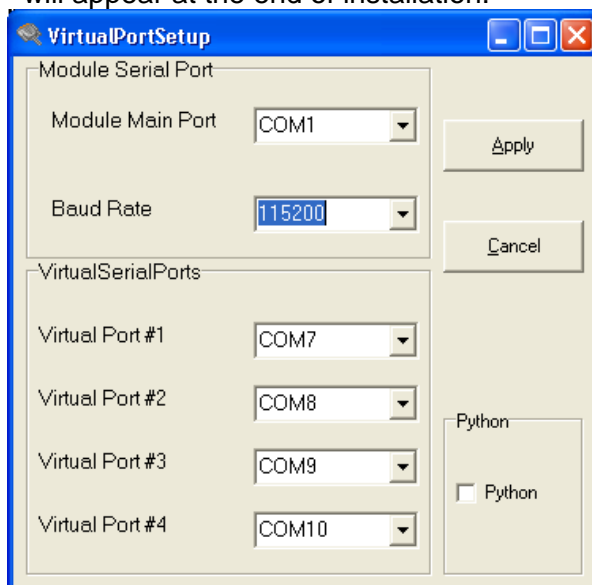
CMUX (Converter-Multiplexer) is a multiplexing protocol implemented in the Telit module that can be used to send data, SMS, fax, TCP data. The Multiplexer mode enables one serial interface to transmit data to four different customer applications from which one is dedicated to Python debug. This is achieved 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-B001¹⁴ 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

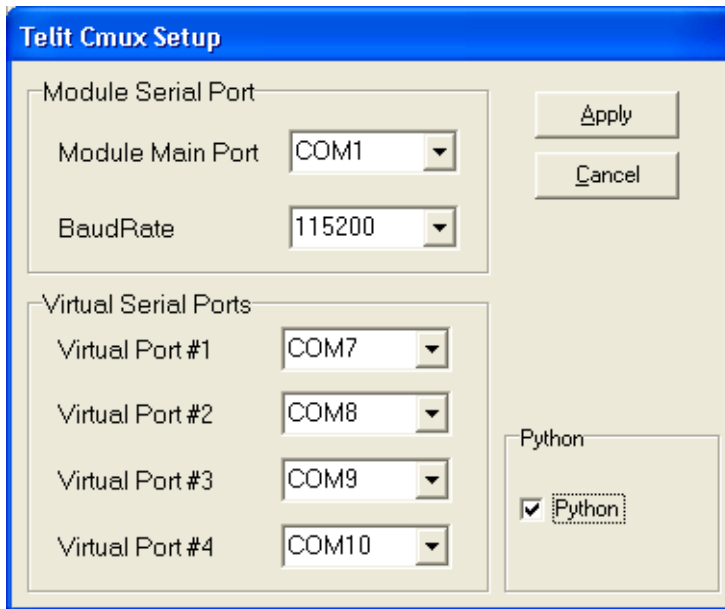
3.9.2.2 Debugging process

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

¹⁴ please contact our technical assistance to get the latest application version



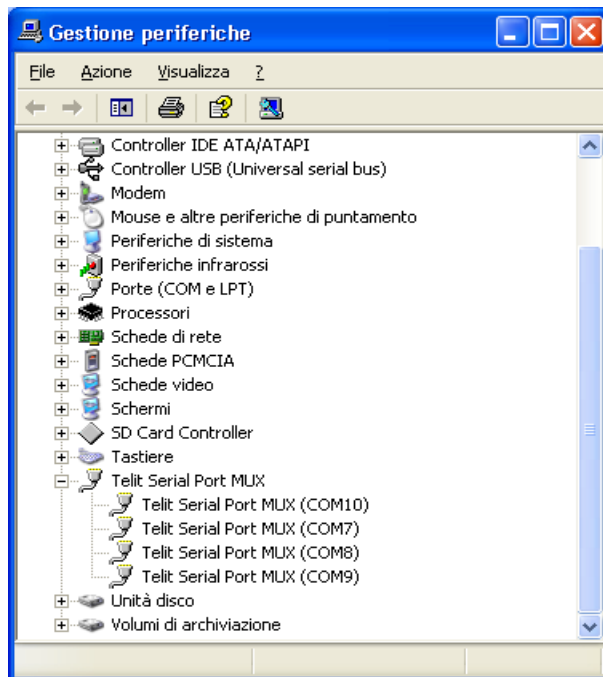
Control if the Setup options are the following:



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.



4 Python notes

4.1 Memory Limits

In order to prevent *memory error*, in phase of compilation or execution of the script, we advise you to consider the following limits:

- allocated memory for each variable
- number of the variables that can cause RAM overflow.

The memory available on Telit Python modules includes:

- around 3MB of Non Volatile Memory for the user scripts and data
- 1.2 MB / 1.5 MB¹⁷ RAM reserved for Python engine usage
- 16KB of memory for each variable

In order to give rough idea of the impact of these constrains consult the table below that contains limits for different types of variables. Please note that these limits are estimated values and should be used only to give general information in Python script development.

Type of variable	number of elements ¹⁸	example
string	16 000	'data'
list	4 000	[23,'data','c']
tuple	4 000	(23,'data','c')
range	4 000	range(3)=[0,1,2,3]
dictionary	worst condition 682	{ 'aaa':1000, 'bbb':1001}

NOTE: each element of list, tuple, range, or dictionary has up to 16KB of memory available.

At each startup the Python task loads a list of:

- variable names
- module names
- methods names

¹⁷ Available only for the products with the following Order-Num.: 3990250655, 3990250656, 3990250664, 3990250665, 3990250653 and 3990250654

¹⁸ 1 element = 1 byte



- strings delimited by “ “ or by ‘ ‘ if not terminated with \r

All these names are included in the main script and in all the files .py called directly or indirectly by the main. The number of names that can be loaded at each startup from the Python task is around 500.

We advise you to use the same variable names in different .py files of the same project, in case this is possible.

The recommended dimension of the compiled file .pyo should be <16KByte

NOTE: It is highly recommended not to use the module as a data logger since all flash memories have limited number of writing and deleting cycles.

4.2 Other Limits

Some other Python limits that should be considered while developing your Python script in order to find an appropriate solution are listed below:

- Python script should not interfere with GSM/GPRS standard operations¹⁹, for this reason there is a pause of **50 ms** each second during the Python task activity.
- GPIO polling frequency **<100Hz**;
- I2C and SPI speed: from 10Kb/s to 20Kb/s

¹⁹ This means that operations such as serial port, protocol stack etc, can run independently from the Python script.



6 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)
ISSUE#2	16/03/07	Added new modules such as: 2.3 MDM2 built-in module 2.5 SER2 built-in module 2.10 GPS built-in module Added new function under IIC and SPI
ISSUE#3	24/05/07	New disclaimer Added introduction for the new modules in paragraph 1.4 and 2 Introduced new chapter with Python notes

