TELIT PYTHON START GUIDE

80000ST10700A Rev.0 2015-06-09

APPLICABILITY TABLE

PRODUCTS

PYTHON 1.5.2+ SCRIPT INTERPRETER ENGINE

- GT863-PY
- GT864-QUAD/PY
- GE864-QUAD V2
- GE864-GPS
- GE865-QUAD
- GE910-QUAD V3
- GL865-DUAL
- GL865-DUAL V3
- GL865-QUAD
- GL865-QUAD V3
- GL868-DUAL
- GL868-DUAL V3
- GC864-QUAD V2

PYTHON 2.7.2 SCRIPT INTERPRETER ENGINE

- GE910-QUAD
- GE910-GNSS
- UL865 SERIES
- UL865-N3G
- UE910 SERIES
- HE910 SERIES
- HE910 MINI PCIE

SPECIFICATIONS SUBJECT TO CHANGE WITHOUT NOTICE

HIGH RISK MATERIALS

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

TRADEMARKS

You may not and may not allow others to use Telit or its third party licensors’ trademarks. To the extent that any portion of the products, components and any accompanying documents contain proprietary and confidential notices or legends, you will not remove such notices or legends.

THIRD PARTY RIGHTS

The software may include Third Party Right software. In this case you agree to comply with all terms and conditions imposed on you in respect of such separate software. In addition to Third Party Terms, the disclaimer of warranty and limitation of liability provisions in this License shall apply to the Third Party Right software.

TELIT HEREBY DISCLAIMS ANY AND ALL WARRANTIES EXPRESS OR IMPLIED FROM ANY THIRD PARTIES REGARDING ANY SEPARATE FILES, ANY THIRD PARTY MATERIALS INCLUDED IN THE SOFTWARE, ANY THIRD PARTY MATERIALS FROM WHICH THE SOFTWARE IS DERIVED (COLLECTIVELY “OTHER CODE”), AND THE USE OF ANY OR ALL THE OTHER CODE IN CONNECTION WITH THE SOFTWARE, INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO THIRD PARTY LICENSORS OF OTHER CODE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND WHETHER MADE UNDER CONTRACT, TORT OR OTHER LEGAL THEORY, ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE OTHER CODE OR THE EXERCISE OF ANY RIGHTS GRANTED UNDER EITHER OR BOTH THIS LICENSE AND THE LEGAL TERMS APPLICABLE TO ANY SEPARATE FILES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CONTENTS

1 INTRODUCTION

1.1 Scope

Scope of this document is to help you to get you up and running a possible environment to develop Python scripts for Telit modules, to the point where you can edit, compile, download to Telit module, and debug a Python script.
This guide will lead you through the considerations about the limits, available documentation, Python editor installation, some available Telit tools and some configurations of parameters.

1.2 Audience

This document is intended for customers who are developing applications with Easy Script in Python Extension feature

1.3 Contact Information, Support

For general contact, technical support services, technical questions and report documentation errors contact Telit Technical Support at:
TS-EMEA@telit.com
TS-AMERICAS@telit.com
TS-APAC@telit.com

Alternatively, use:
http://www.telit.com/support

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:
http://www.telit.com

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.
Telit appreciates feedback from the users of our information.

1.4 Text Conventions

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

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

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

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

1.5 Related Documents

- Easy Script in Python, 80000ST10020a (10.xx.xxx)
- Easy Script in Python 2.7, 80378ST10106A (12.xx.xxx, 13.xx.xxx)
- AT Commands Reference Guide, 80000ST10025A (10.xx.xxx, 13.xx.xxx, 16.xx.xxx)
- HE910/UE910/UL865 AT Commands Reference Guide, 80378ST10091A (12.xx.xxx)
- Virtual Serial Device Application Note, 80000nt10045A (10.xx.xxx)
- HE Family Ports Arrangements User Guide, 1vv0300971 (12.xx.xxx)
- GE910 Family Ports Arrangements User Guide, 1vv0301049 (13.xx.xxx)

2 BEFORE TO START…

2.1 Considerations about Python limits for Telit modules

Before starting to write your project in Python, please consider the **Python limits** depending on the Python interpreter engine equipped with your module. They are described in details in the ***Easy script in Python*** user guide written for the specified Python interpreter engine you are using (**1.5.2+** or **2.7.2** at present), that it is also the **main reference document** to develop programs in Python for Telit modules.

In general you need to consider that a Python script is executed in a task with the **lowest priority** on the Telit module, so its execution won't interfere with GSM/GPRS/UMTS normal operations. Furthermore, this allows serial ports, protocol stacks etc. to run independently from the Python script.
This also means that you can not achieve **real-time** applications with Python version running on the module or you need to consider if the limitations are acceptable for your application.

Also consider the **memory limits** described in the ***Easy script in Python*** user guide.

Also consider that is highly recommended **to not use the module as a data logger** since all flash memories have limited number of writing and deleting cycles.

Consider that the **execution speed** of the scripts:

- *depends on the particular real scenario* : the Python script is executed in a task with the lowest priority on the Telit module and if GSM/GPRS/UMTS activities are running, the Python operations will be delayed. This is a reason it is not possible to give an absolute evaluation of the execution speed in MIPS

- *depends on the embedded Python interpreter engine version; there are two versions at present: 1.5.2+ and 2.7.2*

- *depends on the specific hardware platform* (specific module) you are using: generally "newer" modules are "faster" than modules of older platforms. Obviously the speed execution of a Python script running on the module and the speed execution of the equivalent Python script running on your PC are not comparable

You can compare the speed execution of different Telit platforms in the same specified conditions with the help of identical scripts running in these different platforms. Pay attention this will not be an absolute measurement for the reasons explained before, also they depend on the real scenario of the test including radio network environment.

Once you have evaluated the limits and decide that it is worth to write an application in Python for your module, then a possible way to start learning about the Python implementation and writing Python code for Telit modules is:

- read this start guide, then

- test the sequence (or a part of it) of the AT commands you need to use in your application~~,~~ sending commands manually using a serial terminal application. This is important to get an idea about the AT commands answer time,the kind of answers you will get and the behavior of the module. "*Command Timing Issues*" information are specified in the AT Commands reference guide of the product you are using or in the Easy script in Python, but it's better to test the commands in a real environment in order to write ~~a~~ good code

- develop first simple pieces of Python code for your application or start by trying some of the Telit_Python_Examples which are available in the *Telit Download zone* of Telit website (if you are not our direct customer, please ask your distributor to provide them for you or contact the Telit Technical Support)

- upload the (eventually compiled) scripts, run and debug the scripts in the module, *not in emulation mode* (emulation mode means Python script running on PC and communicating to the module via a serial port).
  Keep in mind that there are several significant differences between the Python engine installed in the module and the Python environment installed on your PC (look at *Python standard functions* chapter in the *Easy script in Python* user guide) and therefore the same code can behave differently. This also means that a code running on PC might not run on the module and viceversa.

2.2 Python script references for Telit modules

- The **Easy Script in Python user guide** for the Python interpreter engine embedded in the module you are using. These documents are available in the *Downloads* section of the page of the product at www.telit.com web site. There are two documents at the moment:

  - **Easy Script in Python** (for 1.5.2+)

  and

  - **Easy Script in Python 2.7**

- The **AT commands reference guide** of the product you are using

- The **Telit_Python_Examples** zip file package (that contains Python272LibPC Emulation modules needed to compile your scripts). Contact your distributor or Telit Technical Support Center (TTSC) to get the package.

- The **Telit_Virtual_Serial_Device_Application_Note_r0.pdf** for products based on **1.5.2+** Python interpreter version to avoid possible resources conflicts

- the **Telit_HE910_UE910_Family_Ports_Arrangements** based on **2.7.x** Python version

- the **Telit_GE910_Family_Ports_Arrangements** based on **2.7.x** Python version

- the **Telit Technical Forum**

- **References about the Python language**:
  e.g.:
  - the manuals for Python language available with PythonWin IDE and the immense resources available in Internet
  - it may be useful also to test single instructions through *Interactive window* of the *Pythonwin* tool

- Other Telit user guides that can be useful in your application, available on Telit web page of the products or in the Telit Download zone (ask your distributor)

3 INSTALL, EDIT, COMPILE, DOWNLOAD, RUN AND DEBUG A PYTHON SCRIPT

Requirements:

- PC with serial and USB ports
- A module with python engine mounted on a board or starter kit with at least a serial port with hardware flow control lines available (a second serial port or USB is needed in order to debug scripts)
- Terminal emulation application (e.g. hyperterminal)
- Connect your module board or starter kit with a serial cable to your PC,start your serial terminal application and try the first contact sending the command AT<Carriage Return>. You should receive an “OK” as an answer.

<Carriage Return> is the **command line terminator character** (decimal 13, hexadecimal 0x0D) is usually sent hitting the „Enter“ key on the keyboard.

3.1 Install the Python release

It is useful to install the Python release on your PC with the same version of the Python interpreter engine embedded in the module. This is necessary to edit and compile in a fast way the source code of the .py files thus obtaining compiled files on PC that can be downloaded later in the module (but we suggest again to avoid debugging the .py scripts running on PC in emulation mode for the reasons written above).
However, after Python package installation, other applications can be used to edit script files if prefered, from simple ones such as Pspad to Eclipse, and the compiling stage can be implemented in batch files.

3.1.1 Modules with 1.5.2+ Python interpreter version embedded

You need to install the Telit Python package *TelitPy1.5.2+_V4.1*
Contact your distributor or Telit Technical Support to get the package.

3.1.2 Modules with 2.7.2 Python interpreter version embedded:

Look at
http://www.python.org/download/releases/2.7.2/
and download the package for your PC

3.1.3 Main changes in the Python interpreter from 1.5.2+ to 2.7.2 version

This paragraph summarizes, **not exhaustively**, some changes implemented in version 2.7.x which runs in more recent platforms, related to Python interpreter version 1.5.2+, supported in Elite and Xgold platforms~~,~~.
What is declared supported or not supported in this paragraph is related to **2.7.2** version.
This paragraph can be useful if you think to adapt your code developed for 1.5.2+ version to new platforms supporting 2.7.2 version.
For other details refer to *Easy script in Python 2.7* and *Easy script in Python* (for modules with 1.5.2+) **user guides**.

3.1.3.1 Python INTERPRETER VERSION

The Python 2.7.2 version implemented in Telit modules is a subset of the complete original engine: core Python language is supported almost entirely, only a few but essential standard Python modules are supported, and other specialized custom built-in modules are added.
Python **.pyc** compiled files are supported.
Python **.pyo** compiled files are not supported.

3.1.3.2 Python SCRIPTS EXECUTION SPEED

2.7.2 version has increased Python scripts execution speed compared to 1.5.2+ version.

3.1.3.3 float NUMERIC TYPE

Standard floating point numeric type (float) is supported.
Standard complex numeric type (complex) is not supported.

3.1.3.4 NON STANDARD BUILT-IN METHODS

Non standard built-in methods

- unlink
- rename
- flashflush

are not implemented.

3.1.3.5 *posix MODULE*

Standard posix Python module is partially supported.
Instead of non standard built-in unlink method use posix.unlink method.
Instead of non standard built-in rename method use posix.rename method.

3.1.3.6 *MOD MODULE (only available from version 12.00.xx4 and version 13.00.xx5)*

Custom built-in MOD module provides the following methods only:
MOD.watchdogEnable(timeout)
MOD.watchdogReset()
MOD.watchdogDisable()
MOD.powerSaving(timeout)

3.1.4 time MODULE

Standard time Python module is partially supported.
Only time, clock and sleep methods are supported.

Instead of custom MOD.secCounter method use time.time or time.clock methods.
Take care that time.time and time.clock methods return a float.
Instead of custom MOD.sleep method use time.sleep method.
Take care that time.sleep argument is in seconds and is a float.

3.1.4.1 *MDM MODULE*

Custom built-in MDM module is implemented.
MDM module continues to work on the AT0 parser instance.
MDM.receive method is not implemented.
Instead of MDM.receive method use MDM.read asynchronous method. To achieve a behavior equivalent to MDM.receive you can use MDM.read inside a loop that tests the time elapsed since start of procedure and read characters sent by the user in each iteration.
Pseudo code example:

```
set time at start of procedure
set data read from MDM buffer to empty
while (condition to exit  the loop based on data read from MDM buffer not
fulfilled) and (elapsed time since
start of procedure not expired):
        update data read from MDM buffer using MDM.read method
        update elapsed time since start of procedure
        sleep (optional)
```

MDM.receivebyte method is not implemented.
Instead of MDM.receivebyte method use MDM.readbyte method inside a while loop.
MDM.sendavail is a new method that returns free space available in bytes in MDM send buffer.

3.1.5 MDM2 MODULE

Custom built-in MDM2 module is implemented.
MDM2 module continues to work on the AT1 parser instance.
MDM2.receive method is not implemented.
Instead of MDM2.receive method use MDM2.read method inside a loop.
MDM2.receivebyte method is not implemented.
Instead of MDM2.receivebyte method use MDM2.readbyte method inside a while loop.
MDM2.sendavail is a new method that returns free space available in bytes in MDM2 send buffer.

3.1.5.1 SER MODULE
Custom built-in SER module is implemented.
SER module continues working on USIF0.
SER.receive method is not implemented.
Instead of SER.receive method use SER.read method inside a loop.
SER.receivebyte method is not implemented.
Instead of SER.receivebyte method use SER.readbyte method inside a while loop.
SER.sendavail is a new method that returns free space available in bytes in SER send buffer.

3.1.5.2 SER2 MODULE (only available from version 12.00.xx4, version 13.00.xx5)
Custom built-in SER2 module provides the following methods only:

SER2.send(string)
SER2.read()
SER2.sendbyte(byte)
SER2.readbyte()

3.1.5.3 USB0 MODULE (only available from version 12.00.xx5 and version 13.00.xx6)
Custom built-in USB0 module provides the following methods:

USB0.send(string, <timeout>)
USB0.read()
USB0.sendbyte(byte, <timeout>)
USB0.readbyte()
USB0.sendavail()

3.1.5.4 GPS MODULE
Custom built-in GPS module is implemented.
The following GPS methods are not implemented:

- resetMode
- getAntennaVoltage
- getAntennaCurrent
- powerSavingMode
- powerSavingWakeUp

3.1.5.5 IIC MODULE (only available from version 12.00.xx4, version 13.00.xx5)
Custom built-in IIC module provides the following methods only:

IIC.new(SDA_pin, SCL_pin, ADDR)
init()
readwrite(string, <read_len>)

3.1.5.6 SPI MODULE (only available from version 12.00.xx4 and version 13.00.xx5)
Custom built-in SPI module provides the following methods only:

SPI.new(SCLK_pin, MOSI_pin, MISO_pin, <SS0>,<SS1>,…,<SS7>)
init(CPOL, CPHA, <SSPOL>, <SS>)

readwrite(string, <read_len>, <SS>)

3.1.5.7 md5 MODULE

Standard md5 Python module in Python 2.7.x is deprecated.
Instead of standard md5 module use the standard hashlib.py module that is written in Python and imports standard _md5 module.
Standard _md5 Python module is supported.

3.1.5.8 CMUX

If CMUX is active Python scripts do not start.
AT command AT#CMUXSCR is not implemented.

3.1.5.9 File Objects

Please refer to *File Object* section of ***Easy script in Python 2.7***

3.2 Install Python editor

As said before consider that any text editor of your choice be used as Python editor, but text editors that are able to check the Python's syntax are useful to avoid errors during compilation of .py source files.
You can find many Python editors on the web.
Again - we suggest to edit the source code, compile, download the compiled file into the module, then run and debug the Python script running on the module, not in your development environment in emulation mode. So it doesn't matter if the editor has a debug feature included, which is only useful to debug Python scripts running on PC.

E.g. for Windows OS you may use the ***PythonWin*** IDE as indicated below.

3.2.1 Modules with 1.5.2+ Python interpreter version embedded**:**

The PythonWin IDE is included in the Telit Python package *TelitPy1.5.2+_V4.1*
Contact your distributor or Telit Technical Support Center (TTSC) to get the package.

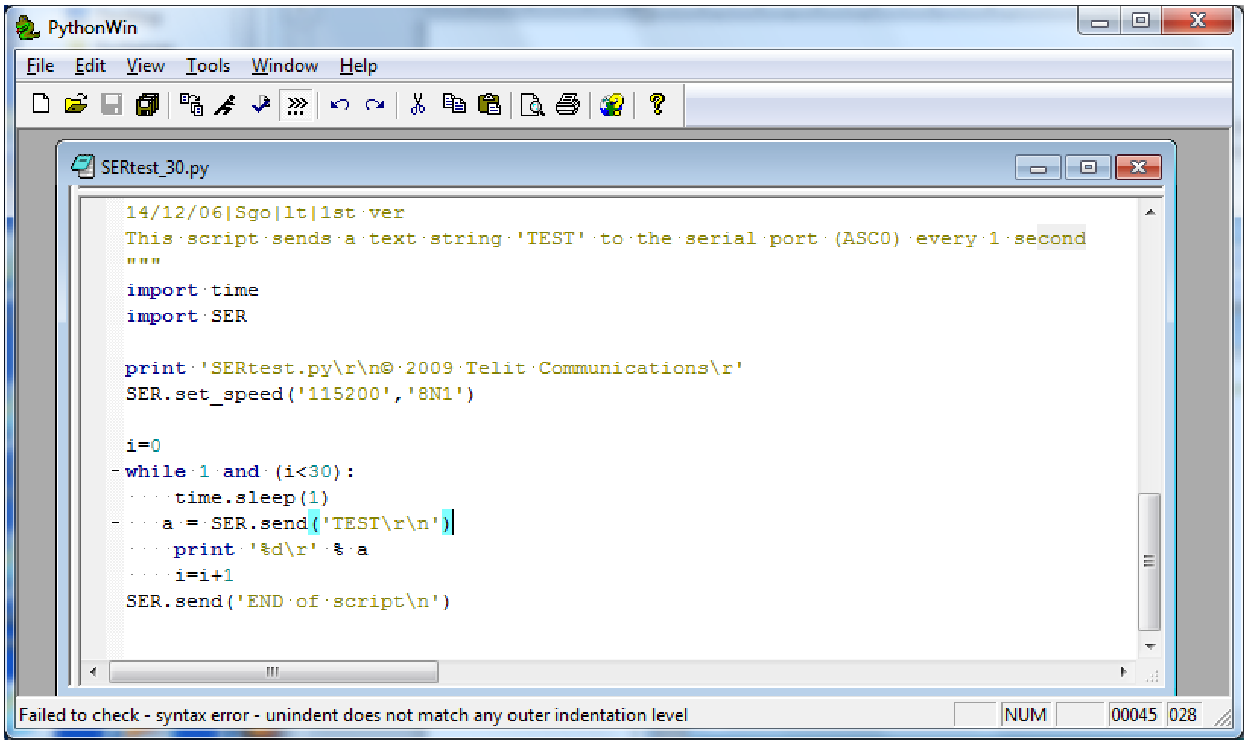
The PythonWin for 1.5.2+ version provides "Compile" and "Download" option added to the contextual menu accessible by right clicking on the selected Python script to compile. The *Compile* option works as default only for 1.5.2+ version.

3.2.2 Modules with 2.7.2 Python interpreter version embedded:

Download PythonWin installation related to Python 2.7 from
http://sourceforge.net/projects/pywin32/files/pywin32/
and install it. Please refer to *Install optional serial package* section of the *Easy script in Python 2.7* user guide for other details. Keep in mind that when want to install additional packages in your Python environment you have to pick a release of the module compatible with your Python engine (for instance if you installed a 32bit Python engine you have to install the 32bit release of the module).
Pay attention that PythonWin for 2.7 version doesn't have the *Compile* option and *Download* option when you select a file and click on the right key of the mouse. It is possible to manually add a *Compile* option by changing a PC registry entry. Please look in the *Compile the py scripts* section of this guide.

3.3 Edit the source code

Edit the source code with a dedicated editor. It is useful to check the syntax of the code before compiling it. E.g. PythonWin editor has a “**Check”** button for this purpose in the menu bar or it is possible to use Ctrl+Shift+C keyboard shortcut. See the example below written for *2.7.2* Python version.



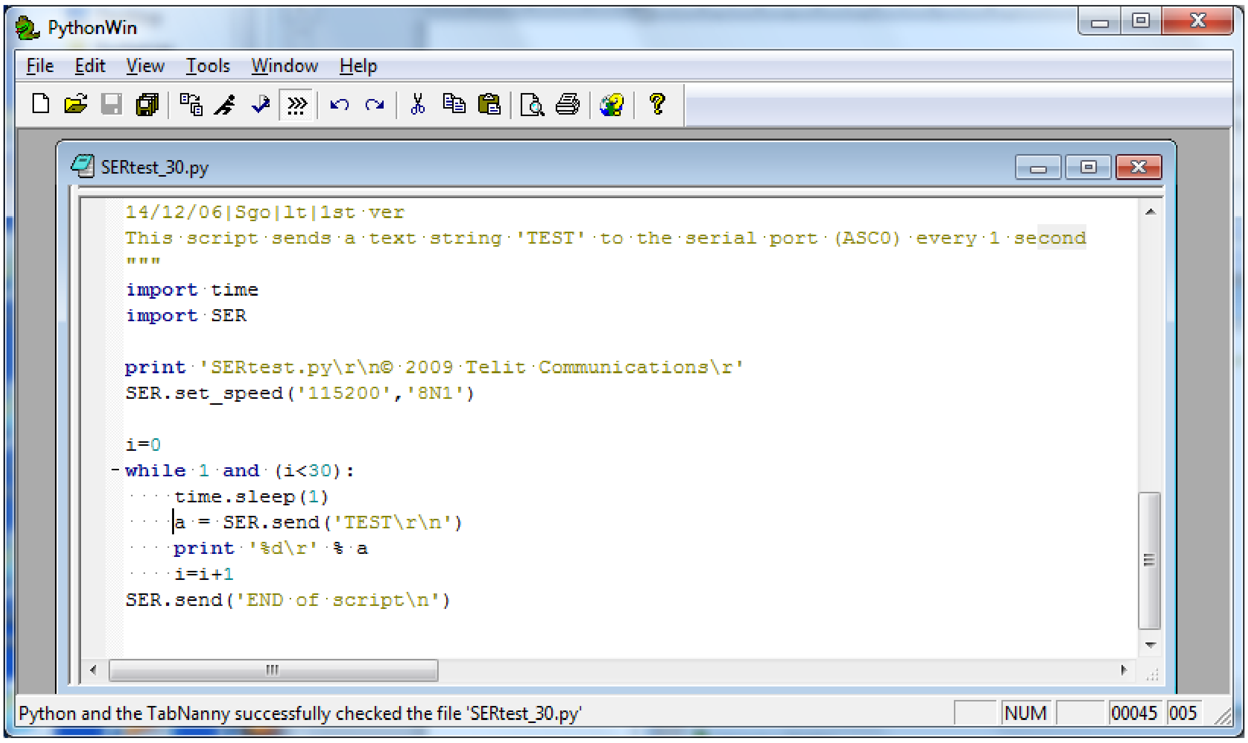
```
14/12/06|Sgo|1t|1st ver
This script sends a text string 'TEST' to the serial port (ASC0) every 1 second
"""
import time
import SER

print 'SERtest.py\r\n© 2009 Telit Communications\r'
SER.set_speed('115200','8N1')

i=0
while 1 and (i<30):
    time.sleep(1)
   a = SER.send('TEST\r\n')
    print '%d\r' % a
    i=i+1
SER.send('END of script\n')
```

```
Failed to check - syntax error - unindent does not match any outer indentation level    NUM    00045 028
```

In this picture above, an error is displayed after the “Check” button was pressed.
The error is displayed, because “a = SER.send('TEST\r\n')” is not correctly indented.
In the bottom picture, the syntax error was corrected and the check operation reports a message “Successful”.



```
14/12/06|Sgo|1t|1st ver
This script sends a text string 'TEST' to the serial port (ASC0) every 1 second
"""
import time
import SER

print 'SERtest.py\r\n© 2009 Telit Communications\r'
SER.set_speed('115200','8N1')

i=0
while 1 and (i<30):
    time.sleep(1)
    a = SER.send('TEST\r\n')
    print '%d\r' % a
    i=i+1
SER.send('END of script\n')
```

Python and the TabNanny successfully checked the file 'SERtest_30.py'

Place sufficient **print** instructions in relevant points in the source code to have your customized debug output.
To visualize these print messages~~,~~ you must have access to the debug port. Look ~~at~~ in the “Configure debugging” section in this document.

As an alternative (or in combination with print) you can use the **SER.send** method to visualize your debug messages on the AT commands port (ASC0 or USIF0 or USB0 for Python modules), but with this strategy it is not possible to capture exceptions logs of messages of premature script stop which are only sent to the debugging port.
Furthermore, in case you want to use SER for debugging, there may be some incompatibilities with particular AT commands (e.g. #SMSATRUN). Look at the *HE Family Ports Arrangements* user guide and *AT Commands reference guide*

Another alternative is to redirect *print* statements to AT commands port (ASC0 or USIF0 or USB0 for Python modules). In this case you can also capture some exception messages if any exception occurs. See the “Configure debugging” section in this document about how to obtain this.

Before compiling the script, **check a last time the syntax correctness** .

If you have both PythonWin versions installed, pay attention which one you are using in the View → *Interactive Window* option of the menu.

3.4 Compile the scripts (py files)

As written in the Easy Script user guide, compiling is an optional operation, but compiling the Python script on PC before downloading to module saves time in Python script execution start.

3.4.1 Modules with 1.5.2+ Python interpreter version embedded:

3.4.1.1 From Command Prompt

Please refer to section “Compiling the Python script” of the *Easy Script in Python* user guide for 1.5.2+ engine or alternatively you can use the following method using Command Prompt:

After you have installed the Telit Python package ***TelitPy1.5.2+_V4.1*** on your PC,

to compile all the .py files in your working directory:

cd *<python.exe directory path>*
python -v -S -OO “<*python.exe directory path* >\Lib\compileall.py" -l –f "< *working directory path* >"

To compile a single .py file in your working directory:

cd *<python.exe directory path>*
python -v -S -OO “<*python.exe directory path* >\Lib\Dircompile.py" "<*file path\filename*>"

e.g.
if the <*python.exe directory path*> is named *C:\Program Files\Python152+* and < *working directory path*> is *C:\MyPythonExamples* , then, to compile all files in the working directory, you need to type and execute in Command Prompt:

```
cd C:\Program Files\Python152+
python -v -S -OO "C:\Program Files\Python152+\Lib\compileall.py" -l -f "C:\MyPythonExamples"
```

or, if the file to compile is named *myfile.py*,

```
python -v -S -OO "C:\Program Files\Python152+\Lib\Dircompile.py" "C:\MyPythonExamples\myfile.py"
```

Please verify that in your *working directory* the corresponding ***pyo*** files have been generated after compilation of the ***.py*** files.

3.4.1.2 Using the Compile option from the contextual menu

If you install PythonWin 4.1 for 1.5.2+ environment, a "*Compile*" option is added to the contextual menu accessible by right clicking on the selected Python script to compile.
This option compiles the file and saves the results in the same directory with a *.pyo extension.
Please refer to section “*Compiling the Python script”* of the *Easy Script in Python* user guide

3.4.2 Modules with 2.7.2 Python interpreter version embedded:

3.4.2.1 From Command Prompt

From the *Easy script in Python 2.7* user guide:

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

after you have installed **Python version 2.7.2** on your PC (as an example in directory C:\Python27):

to compile all the .py files in your working directory:

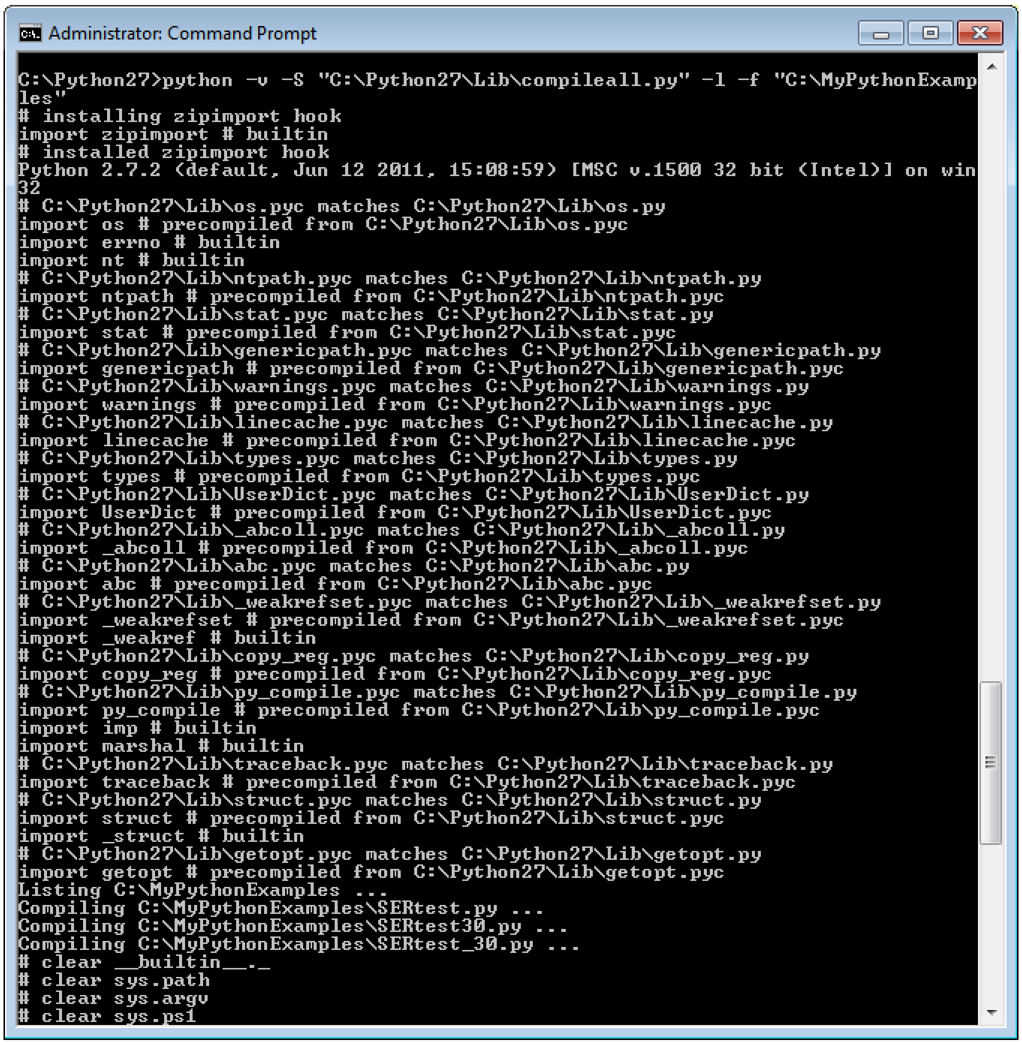
cd <*python.exe directory path*>
python -v -S "<*python.exe directory path* >\Lib\compileall.py" -l –f "< *working directory path* >"

or

python -v -S .\Lib\compileall.py -l –f "< *working directory path* >"

To compile a single .py file in your working directory::
python -v -S "<*python.exe directory path* >\Lib\compileall.py" -l –f "< *file path*\file name>"

Please verify that in your *working directory* the corresponding ***pyc*** files have been generated after compilation of the ***.py*** files.
Below a screenshot of the output of this operation, where the <*python.exe directory*> is the C:\Python27 and <*working directory path*> is *C:\MyPythonExamples*

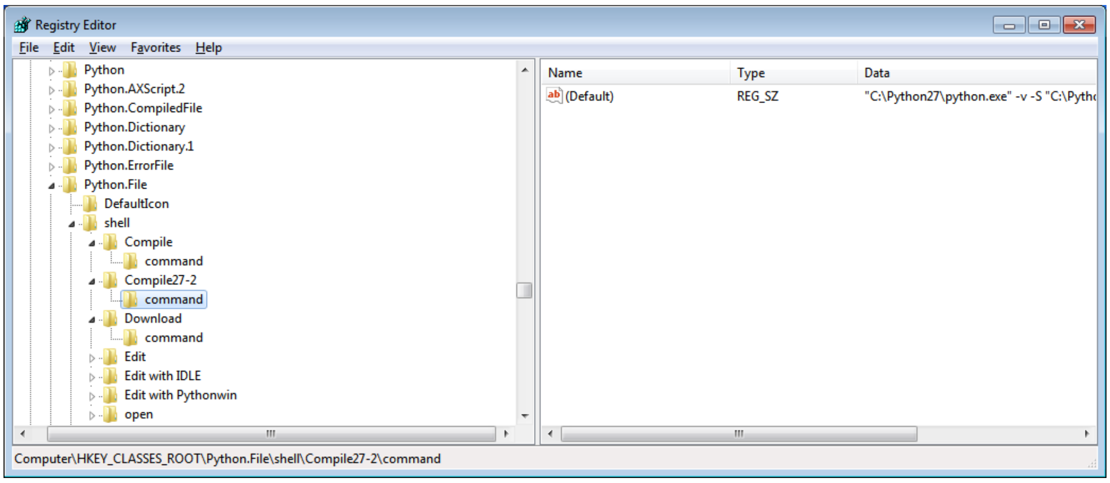

Administrator: Command Prompt

```
C:\Python27>python -v -S "C:\Python27\Lib\compileall.py" -l -f "C:\MyPythonExamp
les"
# installing zipimport hook
import zipimport # builtin
# installed zipimport hook
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win
32
# C:\Python27\Lib\os.pyc matches C:\Python27\Lib\os.py
import os # precompiled from C:\Python27\Lib\os.pyc
import errno # builtin
import nt # builtin
# C:\Python27\Lib\ntpath.pyc matches C:\Python27\Lib\ntpath.py
import ntpath # precompiled from C:\Python27\Lib\ntpath.pyc
# C:\Python27\Lib\stat.pyc matches C:\Python27\Lib\stat.py
import stat # precompiled from C:\Python27\Lib\stat.pyc
# C:\Python27\Lib\genericpath.pyc matches C:\Python27\Lib\genericpath.py
import genericpath # precompiled from C:\Python27\Lib\genericpath.pyc
# C:\Python27\Lib\warnings.pyc matches C:\Python27\Lib\warnings.py
import warnings # precompiled from C:\Python27\Lib\warnings.pyc
# C:\Python27\Lib\linecache.pyc matches C:\Python27\Lib\linecache.py
import linecache # precompiled from C:\Python27\Lib\linecache.pyc
# C:\Python27\Lib\types.pyc matches C:\Python27\Lib\types.py
import types # precompiled from C:\Python27\Lib\types.pyc
# C:\Python27\Lib\UserDict.pyc matches C:\Python27\Lib\UserDict.py
import UserDict # precompiled from C:\Python27\Lib\UserDict.pyc
# C:\Python27\Lib\_abcoll.pyc matches C:\Python27\Lib\_abcoll.py
import _abcoll # precompiled from C:\Python27\Lib\_abcoll.pyc
# C:\Python27\Lib\abc.pyc matches C:\Python27\Lib\abc.py
import abc # precompiled from C:\Python27\Lib\abc.pyc
# C:\Python27\Lib\_weakrefset.pyc matches C:\Python27\Lib\_weakrefset.py
import _weakrefset # precompiled from C:\Python27\Lib\_weakrefset.pyc
import _weakref # builtin
# C:\Python27\Lib\copy_reg.pyc matches C:\Python27\Lib\copy_reg.py
import copy_reg # precompiled from C:\Python27\Lib\copy_reg.pyc
# C:\Python27\Lib\py_compile.pyc matches C:\Python27\Lib\py_compile.py
import py_compile # precompiled from C:\Python27\Lib\py_compile.pyc
import imp # builtin
import marshal # builtin
# C:\Python27\Lib\traceback.pyc matches C:\Python27\Lib\traceback.py
import traceback # precompiled from C:\Python27\Lib\traceback.pyc
# C:\Python27\Lib\struct.pyc matches C:\Python27\Lib\struct.py
import struct # precompiled from C:\Python27\Lib\struct.pyc
import _struct # builtin
# C:\Python27\Lib\getopt.pyc matches C:\Python27\Lib\getopt.py
import getopt # precompiled from C:\Python27\Lib\getopt.pyc
Listing C:\MyPythonExamples ...
Compiling C:\MyPythonExamples\SERtest.py ...
Compiling C:\MyPythonExamples\SERtest30.py ...
Compiling C:\MyPythonExamples\SERtest_30.py ...
# clear __builtin__._
# clear sys.path
# clear sys.argv
# clear sys.ps1
```

3.4.2.2 *Using the Compile option from the contextual menu*

If you install the 2.7.2 version only, there is no "Compile" option added to the contextual menu accessible by right clicking on a Python script. But it is possible to create it in this way:

- Launch “regedit” from the Windows *Run* window
- Navigate to and select the *HKEY_CLASSES_ROOT\Python.File\shell* key
- Create two keys e.g.: *Compile27-2* and *command*
- You will get a situation similar to the one below :



Name	Type	Data
(Default)	REG_SZ	"C:\Python27\python.exe" -v -S "C:\Pyth

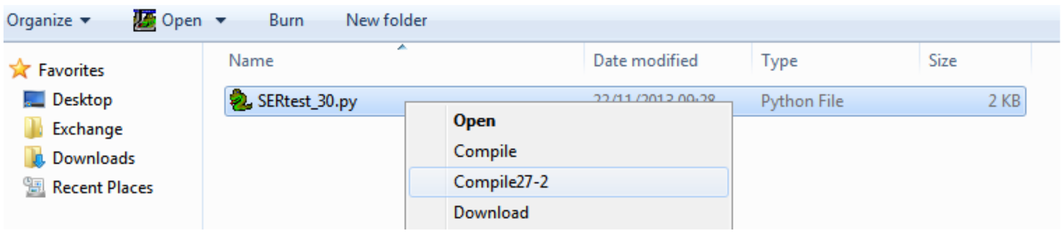
Computer\HKEY_CLASSES_ROOT\Python.File\shell\Compile27-2\command

- In *HKEY_CLASSES_ROOT\Python.File\shell\***Compile27-2***\command key sub-group* create and associate the following String Value:

  "C:\Python27\python.exe" -v -S "C:\Python27\Lib\compileall.py" -l -f "%1"

Then:

- select (click) the Python script to compile
- right click on the mouse
- a contextual menu appears and you can see a similar situation :



If you select *Compile27-2* option the file will be compiled according to *Python 2.7.2* rule

In the screenshot above there is also the "*Compile*" option, because also the Python 152+ version has been installed on the same machine. But in this case in *HKEY_CLASSES_ROOT\Python.File\shell\Compile\command* key,
the following String Value is associated:
"C:\Program Files\Python152+\python.exe" -v -S -OO "C:\Program Files\Python152+\Lib\Dircompile.py" "%1"

3.4.2.3 *Compiling under Linux environment*

Install Python 2.7.2:

```
cd /opt
mkdir Python_272
cd Python_272

wget http://legacy.python.org/ftp//python/2.7.2/Python-2.7.2.tgz

tar -zxvf Python-2.7.2.tgz

cd Python-2.7.2

./configure
make
make install
```

Compile file:

cd < *py file path* >
python2.7 -m py_compile <*py file name*>

3.5 Download the files from PC to the module

Please refere to the section *Download the Python script* in the *Easy script in Python* user guide (according to the Python script interpreter version embedded in your module) for a general script downloading method.

Basically you should use the following AT command:

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

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

Before sending this command, it is obviously needed to have one of the AT command ports of the module (ASC0 or USIF0) connected to a COM port at PC side.

For a successful transfer of the files ~~with~~ of big size it~~'~~ is necessary that all the hardware flow control lines are available at module side and connected along RX and TX lines.
When the size of the file exceeds the size of the serial buffer of the module you risk to loose data without hardware flow control implemented.
In case you can not implement a hardware flow control a possible strategy, but without the guarantee of success in all cases, is to implement via a software ~~a~~ routine that you need to tune yourself , in a similar way as presented below:

1. Issue AT#WSCRIPT command with the right size

2. Send chunks of the file (e.g.1000 bytes for each block) to the serial port . The data arrive in the serial buffer before to be transferred to the module flash memory

3. wait for x seconds (e.g. 10 seconds, you need to tune this time with tests) to allow the module to transfer (write) the data from the serial buffer to the flash memory

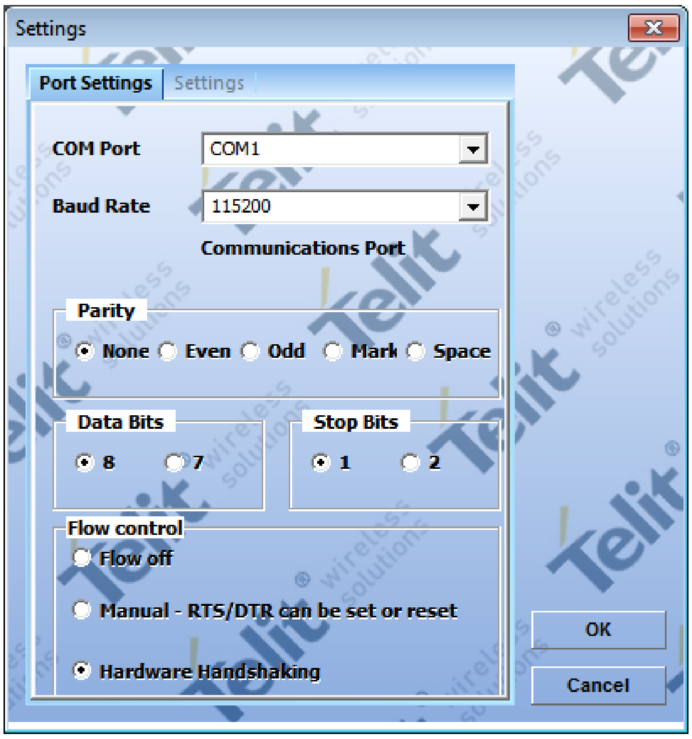
4. repeat the steps 2 and 3 until you receive the "OK" string (the declared number of bytes have been written in the flash, otherwise you will get an error after a timeout]

You can find some other software tools in internet useful to download Python scripts along Telit supplied tools.

3.5.1 File downloading using Telit AT Controller tool

Hereinafter you can find some screenshots related to the use of ***Telit AT controller*** tool.
Contact your distributor or Telit Technical Support to get the package.

Select *Hardware – Handshaking* option as in the screenshot below:



COM Port: COM1

Baud Rate: 115200

Communications Port

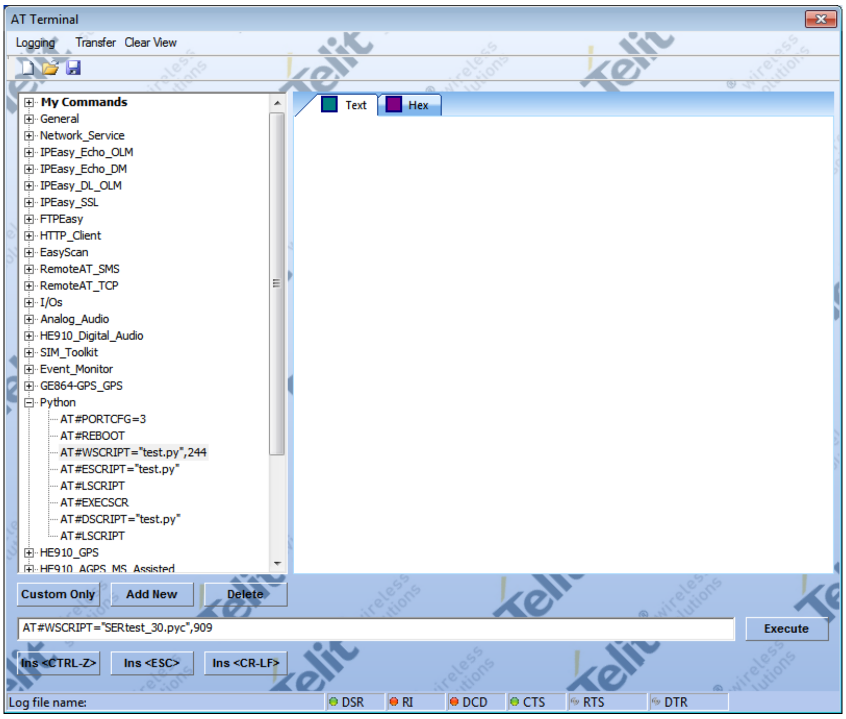
Parity: [✓ None] Even Odd Mark Space

Data Bits: [✓ 8] 7

Stop Bits: [✓ 1] 2

Flow control: Flow off / Manual - RTS/DTR can be set or reset / [✓ Hardware Handshaking]

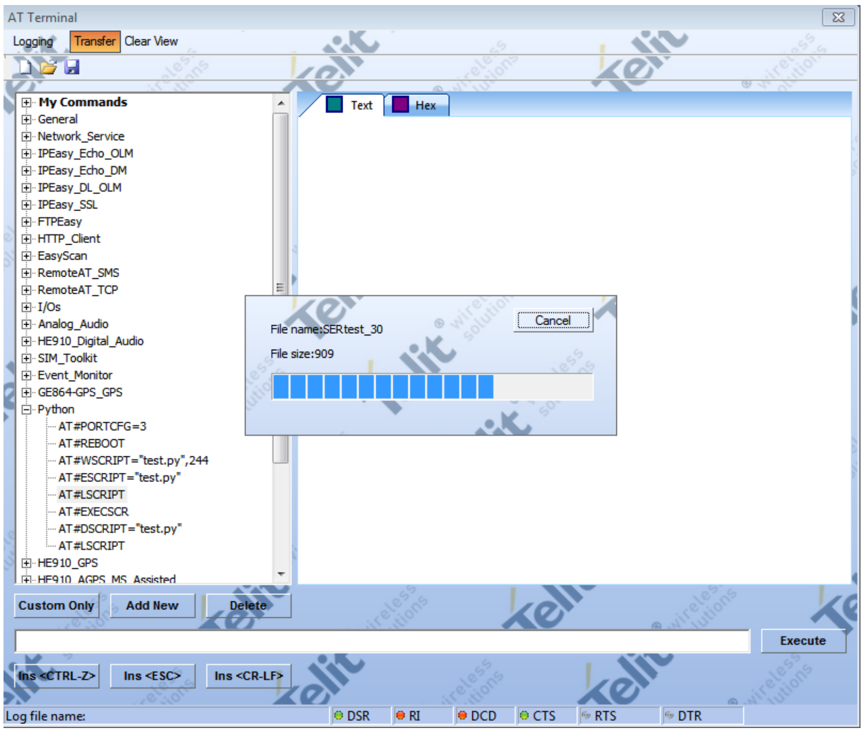
Then you must send the command ***at#wscript*** with the name and size of the file to transfer



```
My Commands
General
Network_Service
IPEasy_Echo_OLM
IPEasy_Echo_DM
IPEasy_DL_OLM
IPEasy_SSL
FTPEasy
HTTP_Client
EasyScan
RemoteAT_SMS
RemoteAT_TCP
I/Os
Analog_Audio
HE910_Digital_Audio
SIM_Toolkit
Event_Monitor
GE864-GPS_GPS
Python
    AT#PORTCFG=3
    AT#REBOOT
    AT#WSCRIPT="test.py",244
    AT#ESCRIPT="test.py"
    AT#LSCRIPT
    AT#EXECSCR
    AT#DSCRIPT="test.py"
    AT#LSCRIPT
HE910_GPS
HE910_AGPS_MS_Assisted
```

```
AT#WSCRIPT="SERtest_30.pyc",909
```

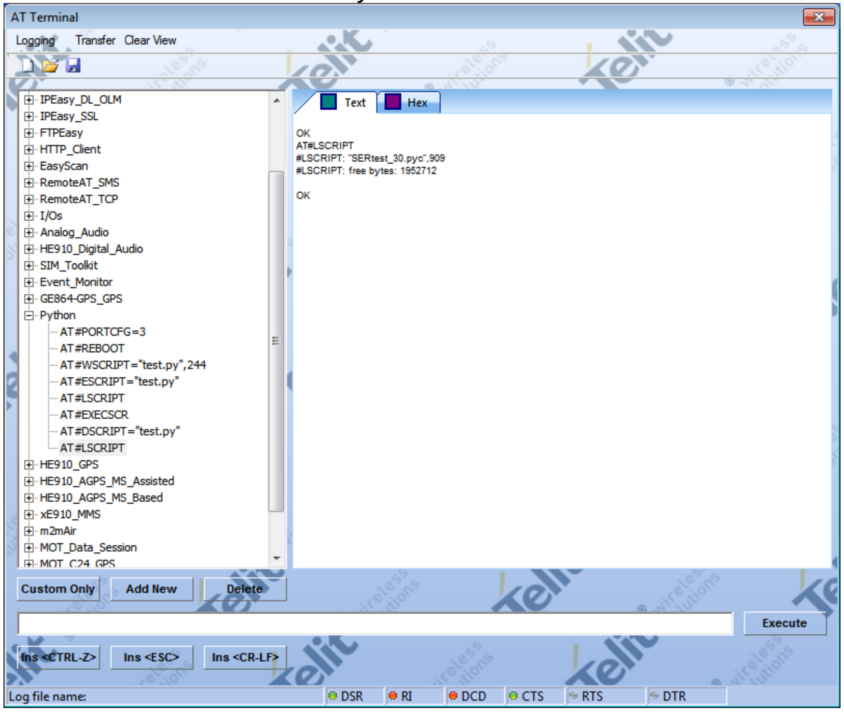
then select theTransfer option to select and transfer the desired file from PC to the module:



File name:SERtest_30

File size:909

Check that the files are transferred to the module successfully (name and size) with **at#lscript** command. Otherwise you should consider if you need to implement hardware flow control if is not yet done.



```
OK
AT#LSCRIPT
#LSCRIPT: "SERtest_30.pyc",909
#LSCRIPT: free bytes: 1952712

OK
```

In case the hardware flow control lines are not available, you can still try to transfer the file by setting *Flow Control* to *Flow off* in the Setting window.

In this case the ">>>" string might not appear after at#wscript. Anyway you can proceed with the file transfer before an internal timeout expires (at expiring of the timeout you would receive an ERROR answer).

3.5.2 File downloading using contextual menu

Hereinafter is also described a method for script downloading if the Telit Python package ***TelitPy1.5.2+_V4.1*** has been installed .
After this package has been installed there is a "*Download*" option added to the contextual menu accessible by right clicking on a Python script.
It is necessary to have the module hardware flow control lines available and connected, eventually through a level translator, to the controller (PC or micro). The EVK2 already has an integrated RS232 level translator.

It is possible to verify which program is associated with this action:

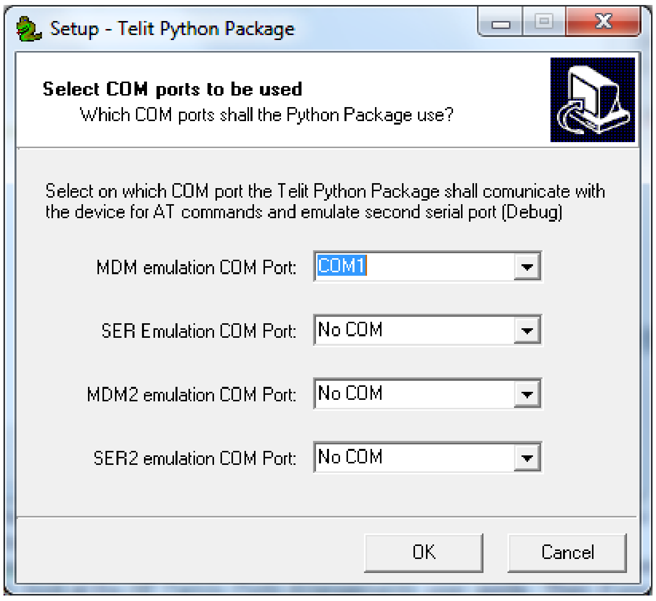
- launch “regedit” from the Windows *Run* window
- navigate to and select the *HKEY_CLASSES_ROOT\Python.File\shell\**Download**\command* key
- read the String Value associated . It should be:

"C:\Program Files\Python152+\python.exe" "C:\Program Files\Python152+\Lib\directDwnld.py" "%1"
So *directDwnld.py* is the file that has the task to download the file to the module through the selected serial port .

Put a RS232 cable between the upper port on EVK2 (PROG) and the COM port on the PC,
to access AT command port of the module (ASC0 or USIF0).
If your PC is not provided with a COM port, you can use a USB to RS232 adapter.

Note: *The EVK2 board comes with 2 DB9 serial ports (PROG/DATA). If you use a RS232 cable between PC and EVK2, pay attention to select the position with label “**RS232”** of the jumpers on EVK2 instead of position with label “USB” (look at “Telit_EVK2_User_Guide.pdf” on www.telit.com)*

With *PythonWin IDE for 1.5.2+ version*, you need to use the *Telit COM portSelection Tool* (available from *Start* menu- *All Programs-Telit Python 1.5.2+ Package*) to set the desired PC COM port for script downloading (e.g. in the screenshot below it is the COM1).
This port must be available with all the hardware flow control physical lines, not with TX and RX lines only.


Setup - Telit Python Package

Select COM ports to be used

Which COM ports shall the Python Package use?

Select on which COM port the Telit Python Package shall comunicate with the device for AT commands and emulate second serial port (Debug)

MDM emulation COM Port: COM1

SER Emulation COM Port: No COM

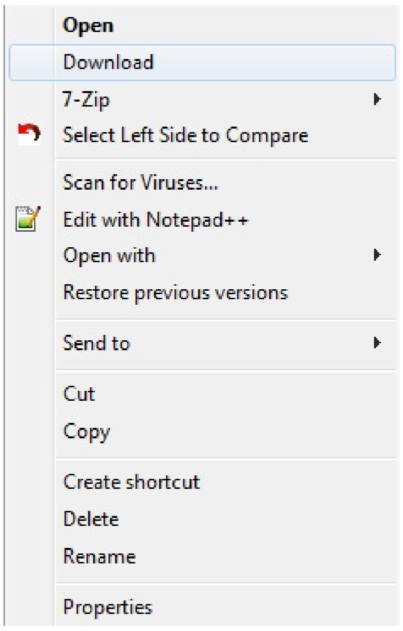
MDM2 emulation COM Port: No COM

SER2 emulation COM Port: No COM

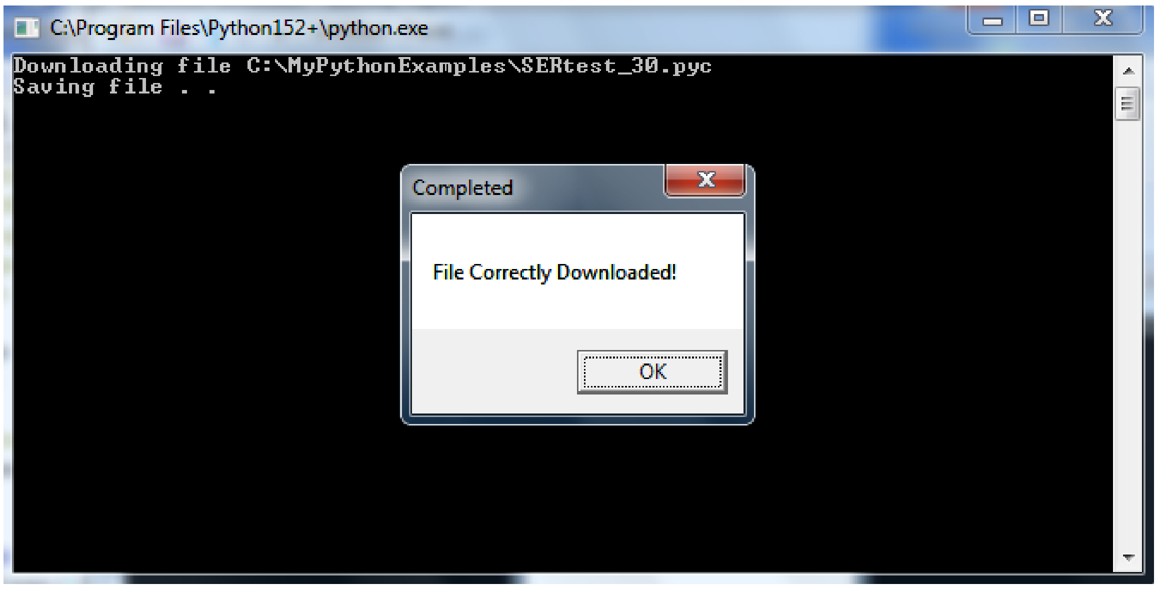
- In case you are using a module with USB port embedded (HE or GE families), you need to refer to HE or GE “*Family Ports Arrangements”* user guide to choose and configure the correct COM port mapped on USB port.
- Connect the EVK2 evaluation kit board to the power supply
- Power on the module

  Then, according to the module’s Python interpreter version embedded:

- Select (clicking on) the **pyo/pyc** file to download, clicking the right key button of the mouse and select the **Download** option as the right as in the screenshot below
- after the downloading, issue send an **ATE1** command to enable echo, if necessary
- Check that the **pyo/pyc** file is in the module after downloading with AT#LSCRIPT command
- Arrange to don’t have any Do not use .py file in the module’s NVM, because the real execution of the .py Python script is delayed from the power on due to the time needed by Python engine to parse the script. The larger is the script, the longer is this delay. The execution of **.pyo/.pyc** compiled Python script is faster because there is no parsing of the script.
- In case there are **py** files in NVM, and if they are not necessary for your application because you have already stored the corresponding **pyo/pyc** files, delete the **py** files only with the command AT#DSCRIPT



If the downloading operation will have success succeeded you will get a similar situation a message as below:



```
Downloading file C:\MyPythonExamples\SERtest_30.pyc
Saving file . .
```

File Correctly Downloaded!

3.6 Enable the script

3.6.1 Modules with 1.5.2+ Python interpreter version embedded:

Enable the script you have downloaded in the module with the command at#escript (e.g. **at#escript="SERtest_30. pyo"**) .
Please refer to section "*Enable the Python script* "of the *Easy Script in Python* user guide

3.6.2 Modules with 2.7.2 Python interpreter version embedded:

Enable the script you have downloaded oin the module with the command at#escript (e.g. **at#escript="SERtest_30. pyc"**) .
Please refer to section "*Enable the Python script* "of the *Easy Script in Python 2.7* user guide

3.7 Run the python script

Please refer to section "*Enable the Python script* "of the *Easy Script in Python* user guide

Run your script in one of the following ways:

- at switch on module startup, according to with the DTR line status (if at#startmodescr=0)
- at switch on module startup, regardless of the DTR line status (if at#startmodescr=1 or 2)
- on at user choice command, with **at#execscr** command (maybe better the best method during debug)

3.8 Configure debugging

The script debug is a process aimed to search and discover the errors and undesirable behaviors of the code and fix them in order to get correct compilation and run.

Here it' is suggested to avoid to debug the Python script in the PC emulated environment of
the *Telit Python Package*, that means running the Python script on PC connected to the module. In fact it may happen that the same script running in the emulated environment could not run in the module.
Many error messages could can result from because of the differences existing between the Python environment embedded in the module and the Python environment in the PC.

The suggested approach to debug Python scripts is to observe the debug messages while the script is running on the module.
Before To get these debug messages you need to decide which what will be your debugging port and consequently which what kind of code you need to write in Python to get the debugging messages on from the selected port and which what kind of AT commands you need to use to configure the debugging.

3.8.1 Modules with 1.5.2+ Python interpreter version embedded:

Please refer to "*Debug Python script"* of the *Easy Script in Python* user guide for further details.

For most of these modules the debugging port is the **ASC1** (auxiliary) port, except for GPS modules (in this case you need to use the *Telit Serial Port Mux* application).

But you could can also decide to use the **ASC0** port as debugging port, with some limitations.

3.8.1.1 Using ASC1 as debug port

In case you decide to use the **ASC1** (auxiliary) port and the module is not a GPS enabled module, you need to put more ***print*** instructions in the parts of the source code that you consider as critical, to get later a flow of your customized debug messages when the script is running.

From an hardware point of view, to see the messages on the second serial port ASC1 it's it is necessary to connect the ASC1 port pins available on your module to the PC COM port, through a level translator as described in the HW user guide of the product you are using.
Pay attention that if you are using a module on its adapter board mounted on EVK2, the level translator is already included in EVK2.

In case you are using a **GPS enabled module**, you need to use physically the port ASC0 port and CMUX features and you will get the *print* debug messages of the *print* instructions on from the fourth virtual port.Look at the section *Using ASC0 as debug port*

3.8.1.2 Using ASC0 as debug port

In case decide to use the **ASC0** port as debugging port (e.g. the ASC1 port is not available on your board), you have three ways to get customized messages:

1. use the **SER.send() method** in your source code
2. **Redirect print** statements and **errors** to ASC0
3. **use *Telit Serial Port Mux* application**

1. It's possible to use the **SER.send()** method to send to ASC0 (AT commands port) your debug messages alternatively or in combination with *print* outputs coming on ASC1 port

   Look at the following Source code snippet:

   ```
   """
   SERtest30.py
   This script sends a text string 'TEST' to the serial
   port (ASC0) and  text string '1' to the debug port
    every 0.5s for 30 times
   """
   import MOD
   import SER

   SER.set_speed('115200','8N1')

    for i in range(30):
       MOD.sleep(5)
       a = SER.send('TEST\n')
       print a

   SER.send('END of script\n')
   ```

   After that the corresponding compiled file SERtest_30.**pyo** file has been enabled and after AT#EXECSCR command has been issued, this example should print a series of "TEST" string on the port associated with AT command port (ASC0) and a series of "1" on the debug port (ASC1). The response "1" on debug port is resulting from successful "SER.send" operations.

   You have to consider that if you have some errors during execution, not managed by your code, the related Python error messages will be visible on the Python debug port only, that is the port where the *print* outputs come.
   In fact on ASC1 port you can see all Python outputs to stdout and stderr.

   So in case ASC1 port is not available in your application and you are using only the SER.send() method to send your debugging messages to ASC0 only, if the script stops prematurely due to not managed exceptions, you can not see any Python error information coming on ASC0 port. But Nevertheless these info sometimes may be necessary to understand the reason for the stop. In this case, it's it is possible to:

2. **redirect print statements** and **errors** to ASC0

    Look at the following Source code snippet:

   ```
   """
   traceonSER.py
   24/01/08|Sgo|bt|1st ver
   Redirect print statements and errors to SERIAL ASC0
   """
   import sys
   import SER

   class SERstdout:
   ```

```
    def __init__(self):
        SER.set_speed("115200","8N1")
    def write(self,s):
        SER.send("DEBUG> " + s+'\n')

print "hello on debug port!"

if(sys.platform != "win32"):
      sys.stdout = SERstdout() # Redirect print
statements
                                       # to SERIAL ASC0
     sys.stderr = SERstdout() # Redirect errors
                               #to SERIAL ASC0

print "hello on serial port!"
```

3. **use *Telit Serial Port Mux* application**

   In case you are using a **GPS enabled module**, you need to use physically the port ASC0 port and CMUX features and you will get the *print* debug messages of the *print* instructions on from the fourth virtual port.Look at the section *Using ASC0 as debug port*

   In respect to with the method described in the section "*Debug Python script on GPS modules using CMUX*" of the *Easy Script in Python* user guide, an alternative method to get these messages is the following:

- run *Telit Serial Port Mux* application
- open and connect one terminal emulator session to the COM port number corresponding to first Virtual port (to send AT commands) and one terminal session to the fourth Virtual port (debug port)
- issue send the following AT commands **at#selint=2, at#cmuxscr=0, at#startmodescr=0** on to the first Virtual port
- Disconnect the terminal emulators if still connected to the Virtual COM ports
- Switch off the module and switch on the module to enable the new *startmodescr* setting
- reconnect the terminal sessions to the above virtual COM ports
- issue on Send with a terminal session connected to first Virtual port: **at#escript="<filename>.pyo"**
- issue send **at#execscr** command to start the execution of the enabled script
- you should observe the debug output (if you have not re-forwarded the print outputs) on in the terminal session connected to the fourth virtual port

3.8.2 Modules with 2.7.2 Python interpreter version embedded:

Please refer to section "*Debug Python script"* of the *Easy Script in Python* 2.7 user guide for further details.

For most of these modules the debugging port is the **USIF1** (auxiliary) port.
For some modules is necessary to set the command **AT#PORTCFG=3** , for others **AT#PORTCFG=0** , before to see the debug messages coming on USF1.

But You could can also decide to use the **USIF0** or **USB0** port as debugging port, with some limitations.

3.8.2.1 Using USIF1 as debug port

In case you decide to use the **USIF1** (auxiliary) port, you need to put more ***print*** instructions in the parts of source code that you consider as critical, to get later a flow of your customized debug messages when the script is running.

From an hardware point of view, to see the messages on the second serial port **USIF1** it's it is necessary to connect the **USIF1** port pins available on your module to the PC COM port, through a level translator as described in the HW user guide of the product you are using.
Pay attention that if you are using a module on its adapter board mounted on EVK2, the level translator is already included in EVK2.

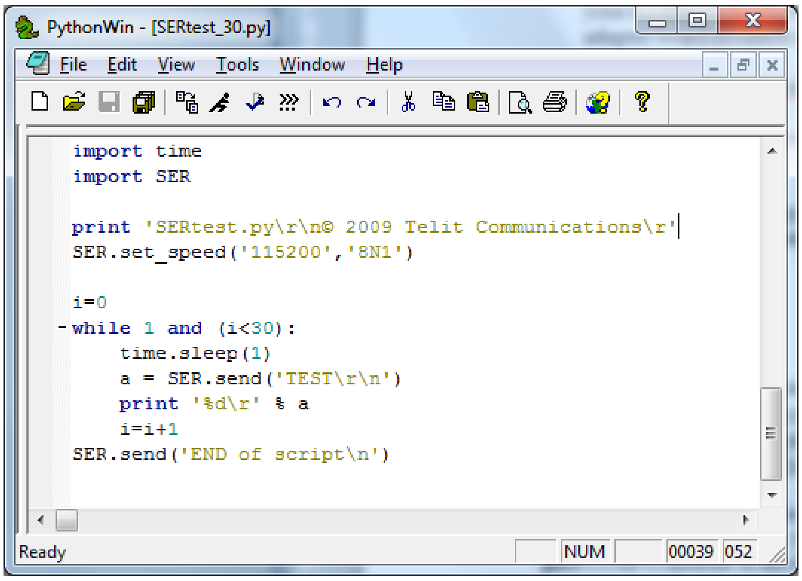
3.8.2.2 Using USIF0 as debug port

In case you decide to use the **USIF0** port as debugging port (e.g. the **USIF1** port is not available on your board), you have two ways to get customized messages:

1. use the **SER.send() method** in your source code
2. **Redirect print** statements and **errors** to **USIF0**

1. It's It is possible to use the **SER.send()** method to visualize on USIF0 (AT commands port) your debug messages alternatively or in combination with *print* outputs coming on USIF1 port

Source code example:

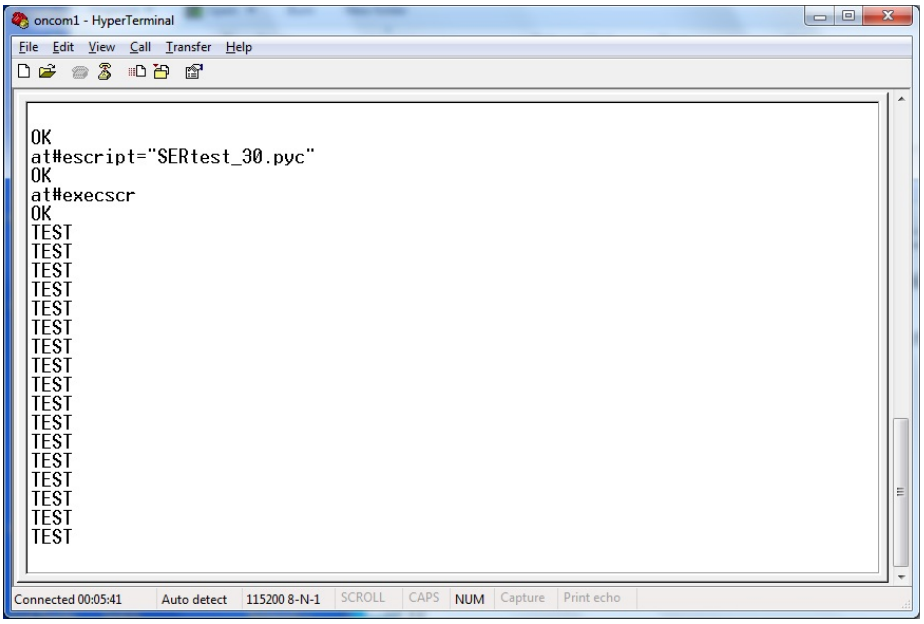


```
import time
import SER

print 'SERtest.py\r\n© 2009 Telit Communications\r'
SER.set_speed('115200','8N1')

i=0
while 1 and (i<30):
    time.sleep(1)
    a = SER.send('TEST\r\n')
    print '%d\r' % a
    i=i+1
SER.send('END of script\n')
```

After that the corresponding compiled file SERtest_30.**pyc** file has been enabled and after AT#EXECSCR command has been issued, what you should observe on USIF0 is the following:



```
OK
at#escript="SERtest_30.pyc"
OK
at#execscr
OK
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
TEST
```

This example should print a series of “TEST” string on the port associated with AT command port (USIF0) and a series of “1” on the debug port (USIF1). The response “1” on debug port is resulting from successful “SER.send” operation.

You have to consider that if you have some errors during execution not managed by your code, the related Python error messages will be visible on the Python debug port only, that is the port where the *print* outputs come.
In fact you can see on USIF1 port you can see all Python outputs to stdout and stderr.

So in case USIF1 port is not available in your application and you are using only the SER.send() method to send your debugging messages to USIF0 only, if the script stops prematurely due to not managed exceptions, you can not see any Python error information coming on USIF0 port. But Nevertheless these info sometimes may be necessary to understand the reason for the stop. In this case, it’s it is possible :

2. **redirect print statements** and **errors** to USIF0

 Look at the following Source code snippet:

```
"""
traceonSER.py
24/01/08|Sgo|bt|1st ver
Redirect print statements and errors to SERIAL USIF0
"""
import sys
import SER

class SERstdout:
    def __init__(self):
        SER.set_speed("115200","8N1")
    def write(self,s):
        SER.send(s)

print "hello on debug port!"

if(sys.platform != "win32"):
      sys.stdout = SERstdout() # Redirect print
statements
```

```
                                                                # to SERIAL USIF0
          sys.stderr = SERstdout() # Redirect errors
                                  #to SERIAL USIF0

     print "hello on serial port!"
```

3.8.2.3 *Using USB0 as debug port*

The USB0 built-in module is an interface between the Python core and the first mini USB port USB0.
The USB0 interface lets the Python script to read from and write to the first mini USB port USB0.

NOTE:

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

To see which is the virtual COM port where the USB0 channel is mapped, please read the following user guides, available on product web page, according to with the product used:

Telit_HE910_UE910_Family_Ports_Arrangements
Telit_GE910_Family_Ports_Arrangements

In case you decide to use the **USB0** port as the debugging port (e.g. you only have only the USB port available on your board), you have two ways to get customized messages:

1. use the **USB0.send() method** in your source code
2. **Redirect print** statements and **errors** to USB0

1. It's It is possible to use the **USB0.send()** method to visualize on USB0 (AT commands port) your debug messages alternatively or in combination with *print* outputs coming on USIF1 port

   Look at the following source code snippet:

```
"""
USB0test_30.py
10/01/14|Sgo|lt|1st ver
This script sends a text string 'TEST' to the to the first
mini USB port USB0 every 1 second.
built-in module USB0 only available from version 12.00.xx5
and version 13.00.xx6
"""

import time
import USB0

print 'USB0test_30.py\r\n© 2014 Telit Communications\r'

 for i in range(30):
    time.sleep(1)
    a = USB0.send('TEST\r\n')
    print '%d\r' % a

USB0.send('END of script\n')
```

After that the corresponding compiled file USB0test_30.**pyc** file has been enabled and after AT#EXECSCR command has been issued, what you should expect to see is that you can see a series of “TEST” strings on the COM port associated to the first mini USB port USB0, whereas you will see and a series of “1” on the debug port (USIF1). “1” is the value returned by the USB0.send method if no error occurred during operation.

You have to consider that if you have there are some errors during execution, not managed by your code, the related Python error messages will be visible on the Python debug port only, that is the port where the *print* outputs come. In fact you can see on the USIF1 port you can see all Python outputs to stdout and stderr.

So in case USIF1 port is not available in your application and you are only using only the USB0.send() method to send your debugging messages to USB0 only, if the script stops prematurely due to not managed exceptions, you can not see any Python error information coming on USB0 port. But Nevertheless these info sometimes might be necessary to understand the reason for the stop. In this case, it’s it is possible :

2. **redirect print statements** and **errors** to USB0

 Look at the following code Source code snippet:

```
"""
printonUSB0.py
10/01/14|Sgo|lt|1st ver
Redirect print statements and errors to the first mini USB
port USB0.
built-in module USB0 only available from version 12.00.xx5
and version 13.00.xx6
"""
import sys
import USB0

class USB0stdout:
    def __init__(self):
        #print "object inizialization\r"
        Pass  #Method definition cannot be empty
    def write(self,s):
        USB0.send(s)

print "print output on debug port!"

if(sys.platform != "win32"):
     sys.stdout = USB0stdout()      #Redirect print
                                    #statements to the
                                    first #miniUSB port
                                    USB0
sys.stderr = USB0stdout()           # Redirect errors to
                                    the #first miniUSB
                                    port USB0

print "print output on serial port!"
```

4 DOCUMENT HISTORY

Revision	Date	Changes
0	2015-06-09	First issue

This image cannot currently be displayed.