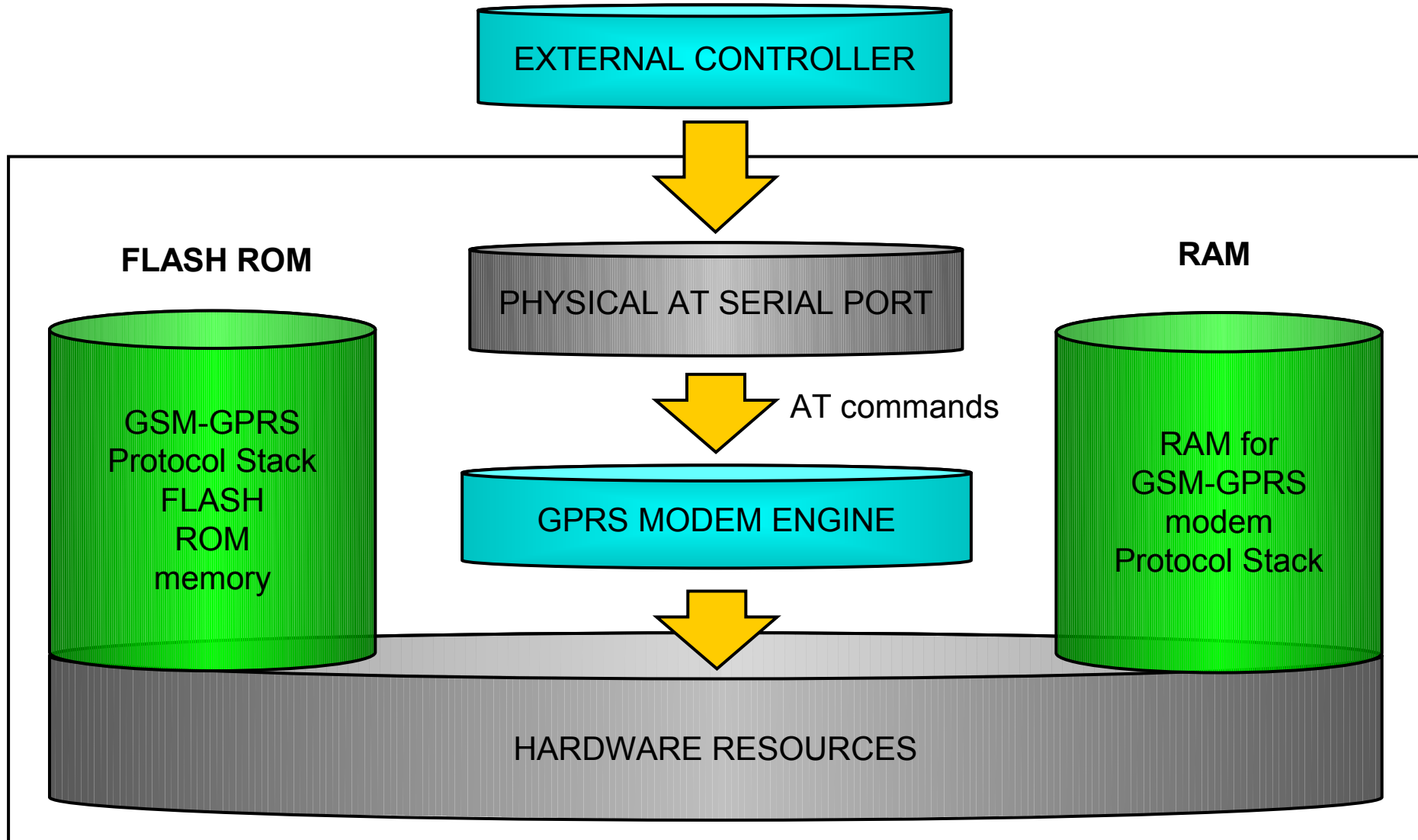


The Easy Script Extension is a feature that allows to drive the modem "internally" writing the software application directly in a high level language:



The Easy Script Extension is aimed at low complexity applications where the application was usually done by a small microcontroller that manage some I/O pins and the module through the AT command and interface.

Standard Module configuration

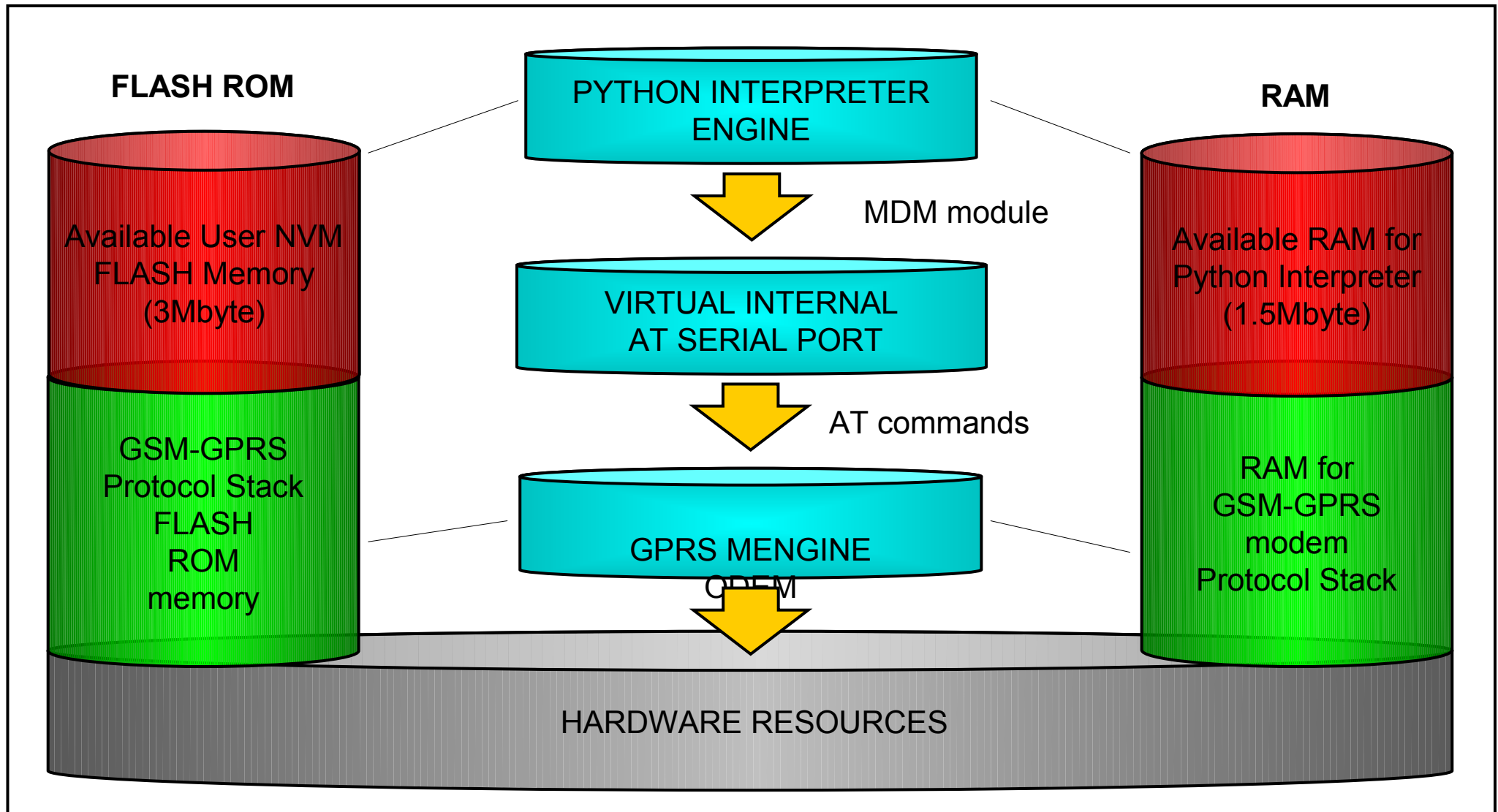


To eliminate the external controller, and further simplify the programming of the sequence of operations, the Python powered modules includes:

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

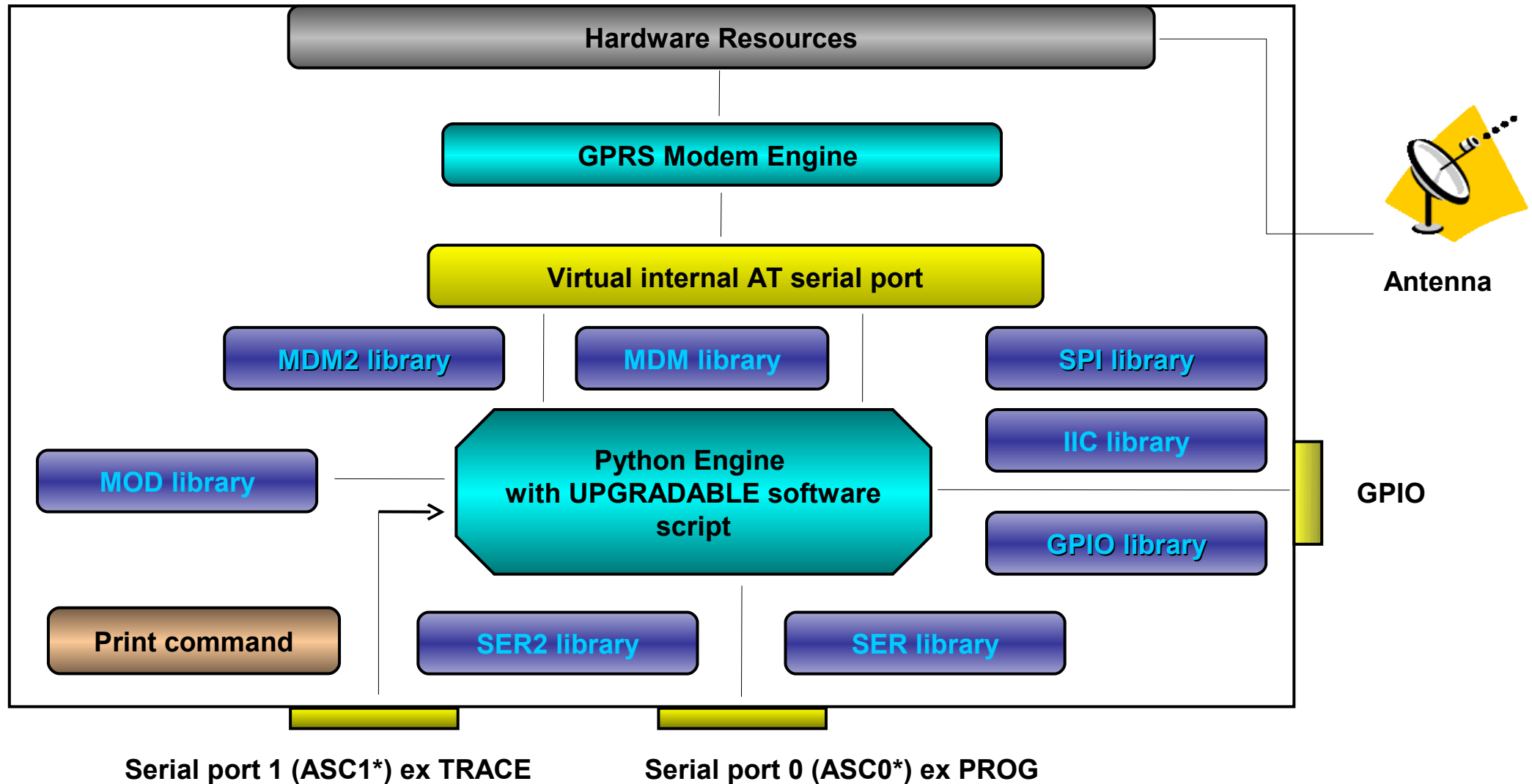


Python Powered Module



- ❑ Python scripts are text files, it's possible to run only one Python script in the Telit PY modules.
- ❑ The Python scripts are stored in NVM inside the module.
- ❑ There's a file system inside the module which allows to write and read files with different names on one single level (no subdirectories are supported).
- ❑ The Python script is executed in a task into the module at the lowest priority, making sure this doesn't interfere with GPRS/GSM normal operations. This allows to the serial ports, protocol stack etc. to run independently from the Python script.
- ❑ The Python script interacts with the module functionality through build-in interfaces.

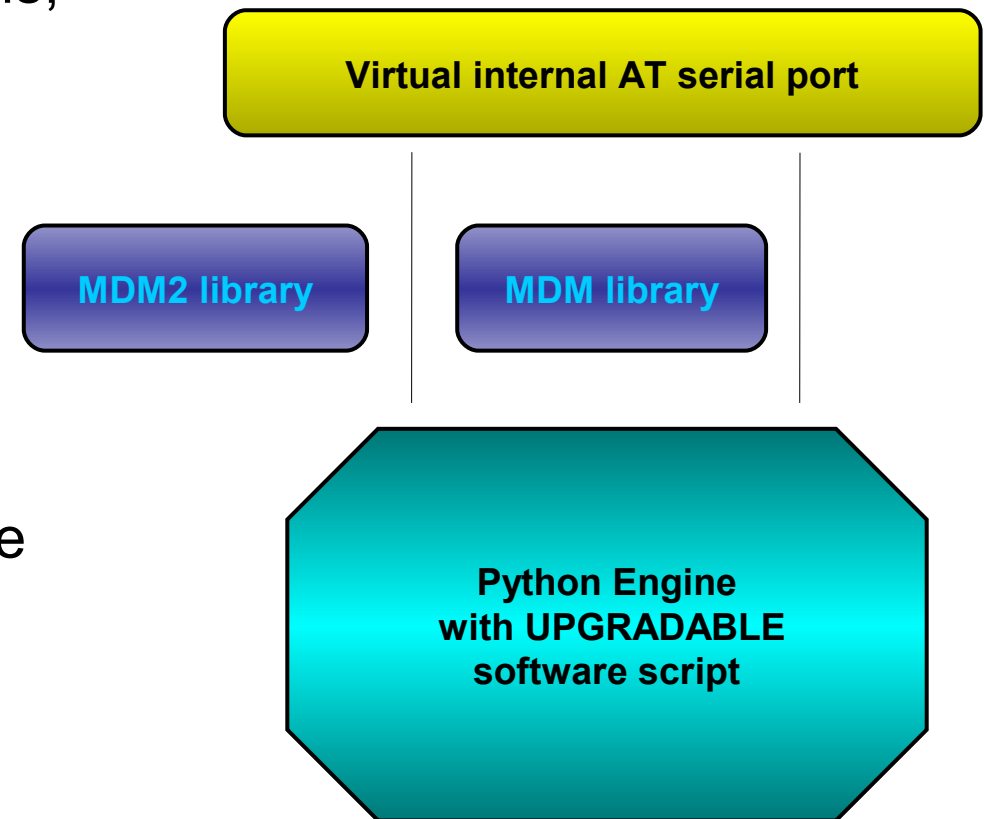
Python interfaces



Python interfaces (MDM)/(MDM2)

MDM and MDM2 interfaces are the most important, allowing Python's scripts to send AT commands, receive responses and unsolicited indications, send data to the network and receive data from the network during connections.

MDM and MDM2 behavior is most likely the same of the common serial port interface in the Telit modules.



Python interfaces (MDM)/(MDM2)



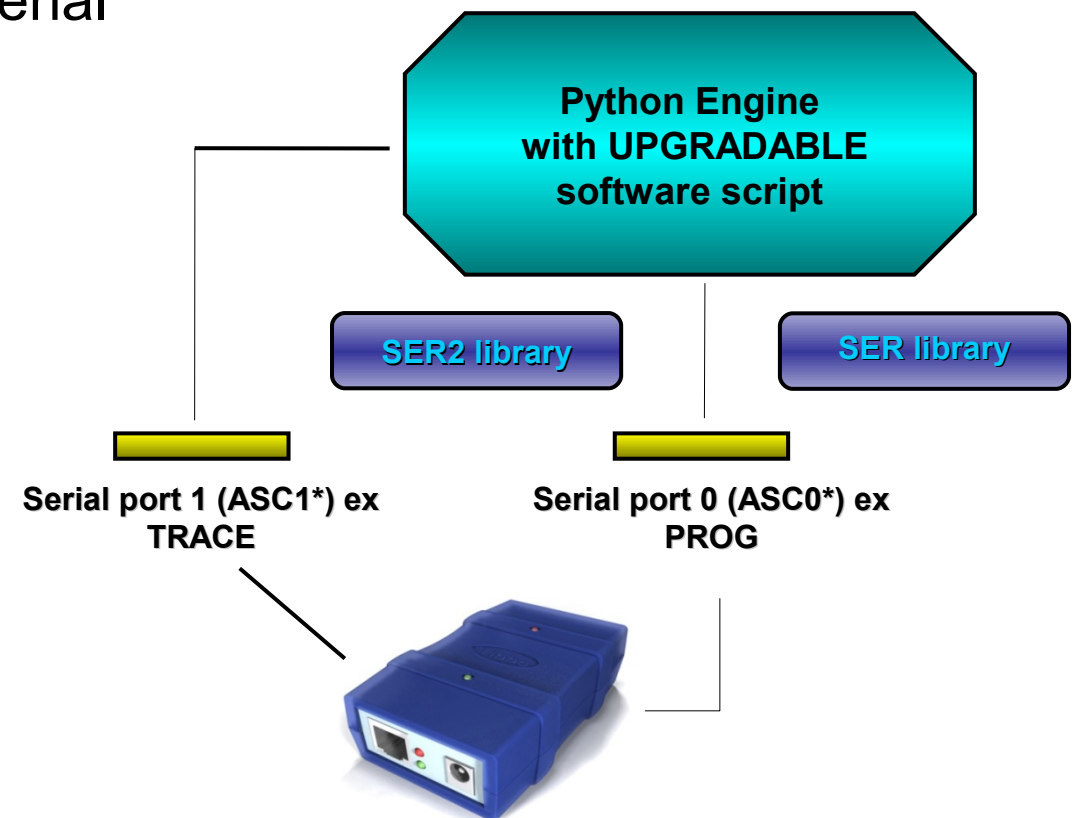
- ❑ They work in parallel like the CMUX serial lines in the Telit modules.
- ❑ All the AT commands available in the Telit modules are working in this software interfaces and follow the same rules as per CMUX.

Note: these interfaces are not real serial ports, but just internal software bridge between Python and mobile internal AT command handling engine.



Python interfaces (SER)

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).



Python interfaces (SER2)



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.

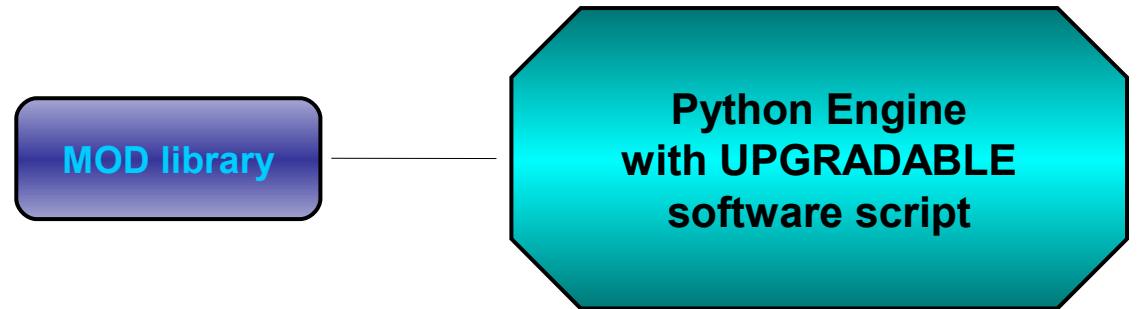
When SER2 built-in module is imported, ASC1 will not be available for trace and debug, in order to have these functionalities you should activate CMUX on ASC0.

Note: SER2 built-in module is available only for non-GPS products.



Python interface MOD

The MOD interface is a collection of useful functions. 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, stop Python execution, manage a Python watchdog, manage the power saving mode from your Python script, etc.

MOD.secCounter()

This method is useful for timers generation in Python script. Return value is a Python integer which is the value of seconds elapsed since 1 January 1970.

MOD.sleep(sleeptime)

Blocks Python script execution for a given time returning the resources to the system. Input parameter timesleep is a Python integer which is the time in 1/10 s to block script execution.

MOD.watchdogxxx(xxx)

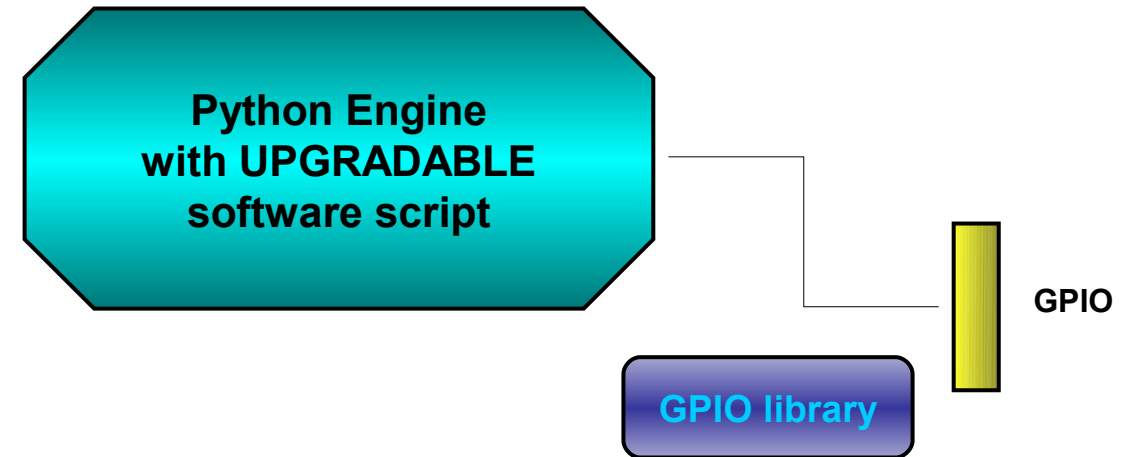
A set of function to manage a Python watchdog .

MOD.powerSavingxxx(xxx)

A set of function to manage in Python the power saving mode .

Python interface GPIO

The **GPIO** interface allows Python script to handle general purpose input output Faster than through AT commands, Skipping the command parser and going directly to control the pins.



GPIO built-in module is the interface between Python core and module internal general purpose input output direct handling.

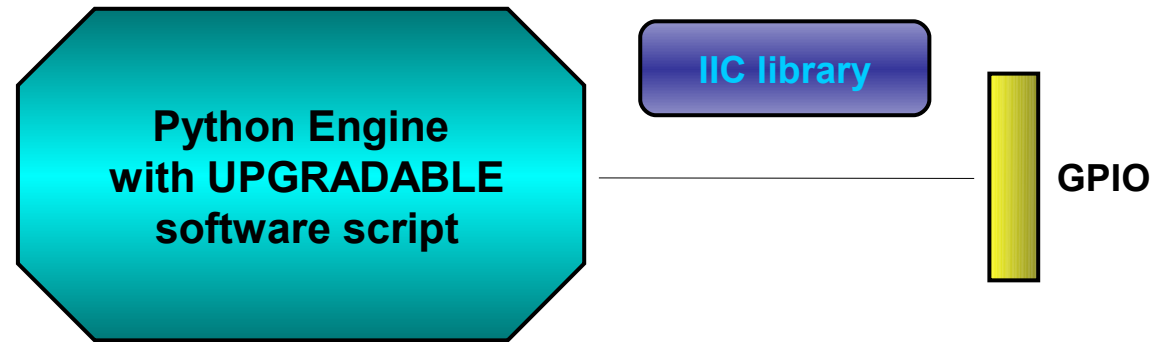
You need to use **GPIO** built-in module if you want to set GPIO values from Python script and to read GPIO values from Python script.

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 suggested.

IIC and SPI built-in module

IIC built-in module is an implementation On the Python core of the IIC bus Master (No Multi-Master).

You need to use IIC built-in module if you want to create one or more IIC bus on the available GPIO pins.



This IIC bus handling module is mapped on creation two GPIO pins that will become the Serial Data and Serial Clock pins of the bus.

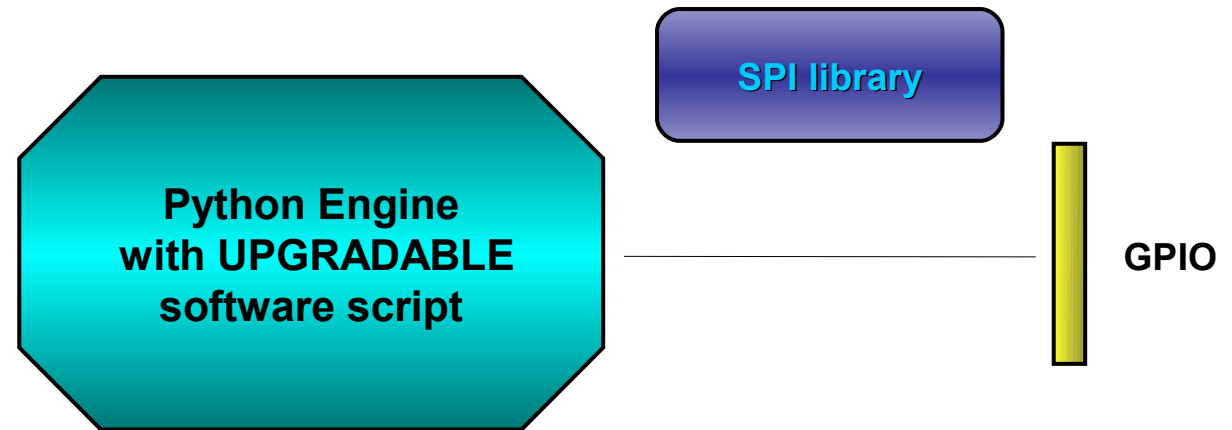
It can be multi-instantiated (you can create more than one IIC bus over different pins) and the pins used must not be used for other purposes.

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

IIC and SPI built-in module

SPI built-in module is an implementation on the Python core of the SPI bus Master.

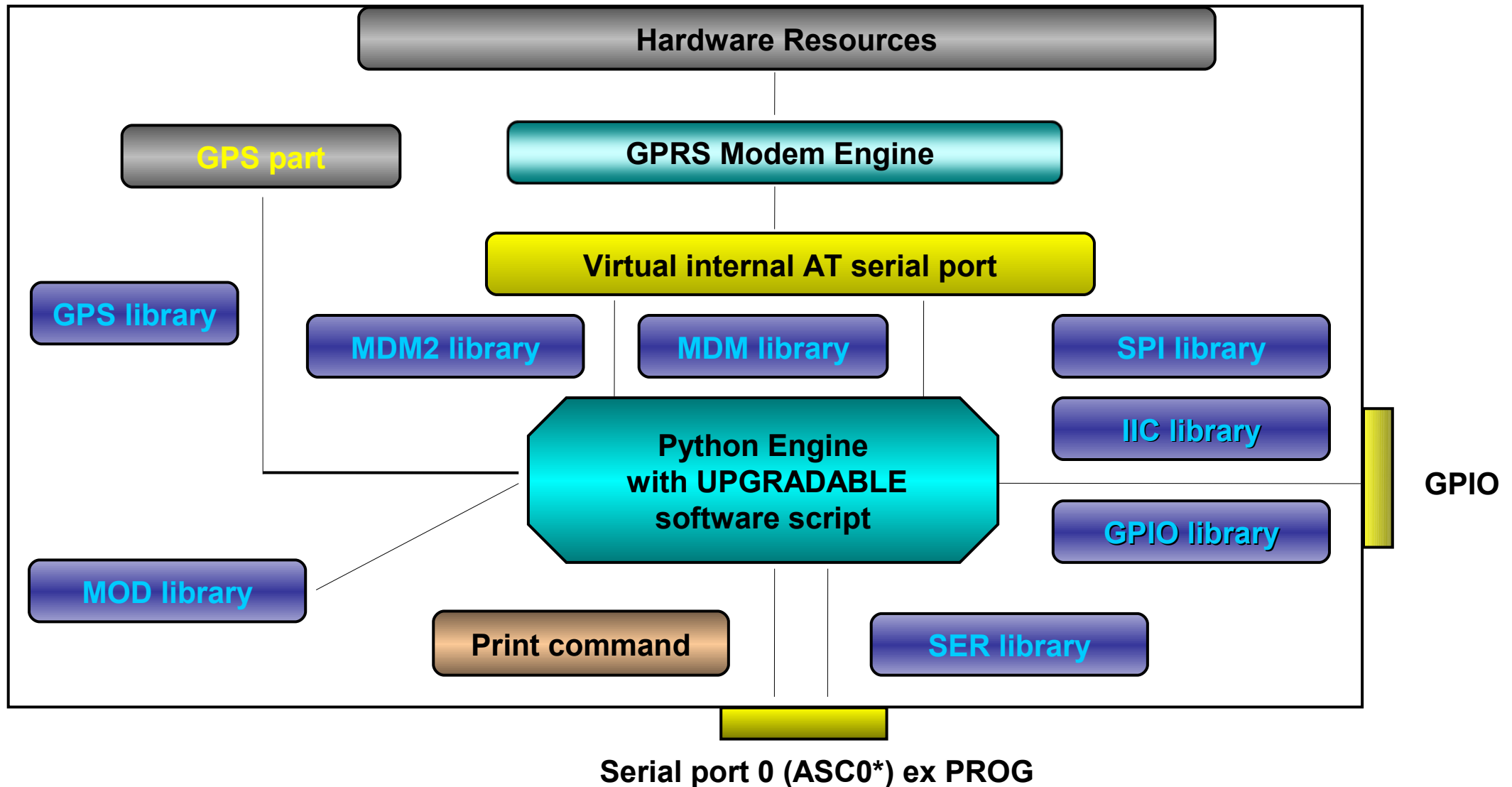
You can use SPI built-in module if you want to create one or more SPI bus on the available GPIO pins.



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 multi-instantiated (you can create more than one SPI bus over different pins) and the pins used must not be used for other purposes.

Python interfaces in GPS modules



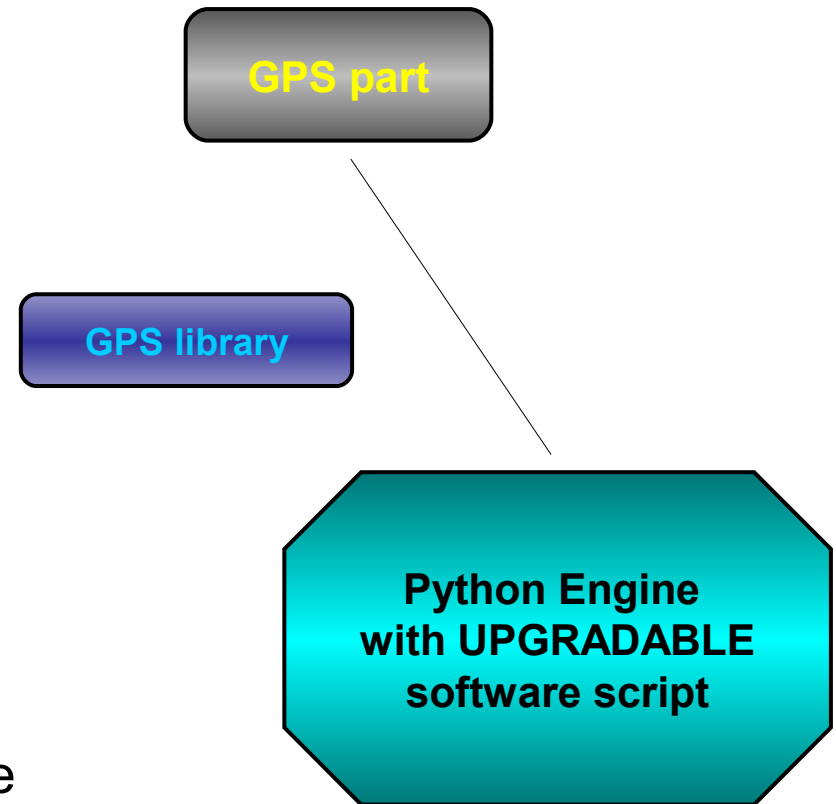
Python interfaces GPS

GPS built-in module is the interface between Python and the mobile internal GPS controller.

It is used to manage GPS controller directly, Instead of using the dedicated AT commands trough MDM built-in module.

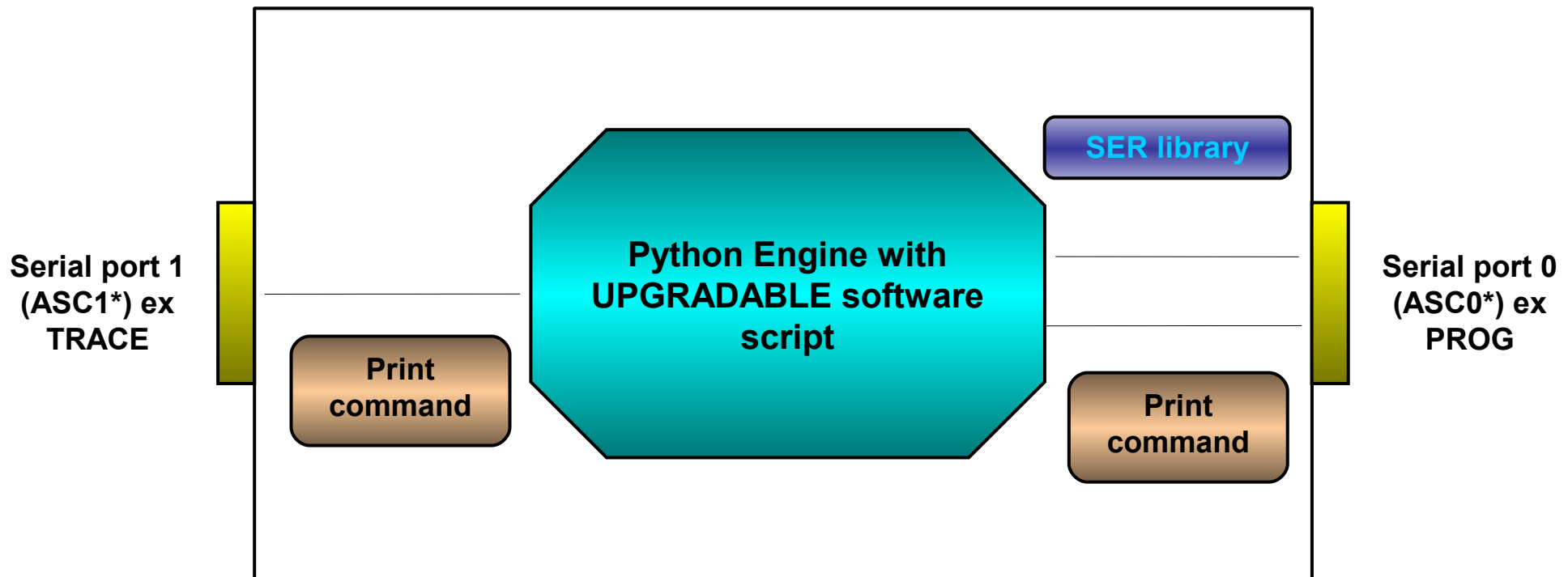
This interface is intended to control the GPS part when the MDM module is busy with other activities (like when GPRS connection is up) and is not possible to use AT Commands.

Using this module you can the read position while you are using GPRS and then send this data trough GPRS, typical tracking solution...



Debug Python script

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



Debug Python script

Python outputs to stdout and stderr:

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

The Telit **GM862-GPS** and **GE863-GPS** have the second serial port pin EMMI TX used for continuous direct output of GPS NMEA sentences that’s why there is another procedure to follow for debugging of the Telit GPS modules.

There are two ways to perform direct debugging:

- SSC port
- CMUX.



Executing a single Python script



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

- 1) **WRITE** the Python Script with any text editor and save it
- 2) **COMPILE** the Python Script using Telit Python Package
- 3) **DOWNLOAD** the Python Script into the module NVM
- 4) **ENABLE** the Python script
- 5) **EXECUTE IT**

There are 4 ways to EXECUTE :

At every power on, with DTR rule, AT command interface lost.

At every power on, NO DTR rule, AT command interface lost.

At every power on, NO DTR rule, AT command interface is available

On user request

AT#STARTMODESCR=0

AT#STARTMODESCR=1

AT#STARTMODESCR=2

AT#EXECSCR

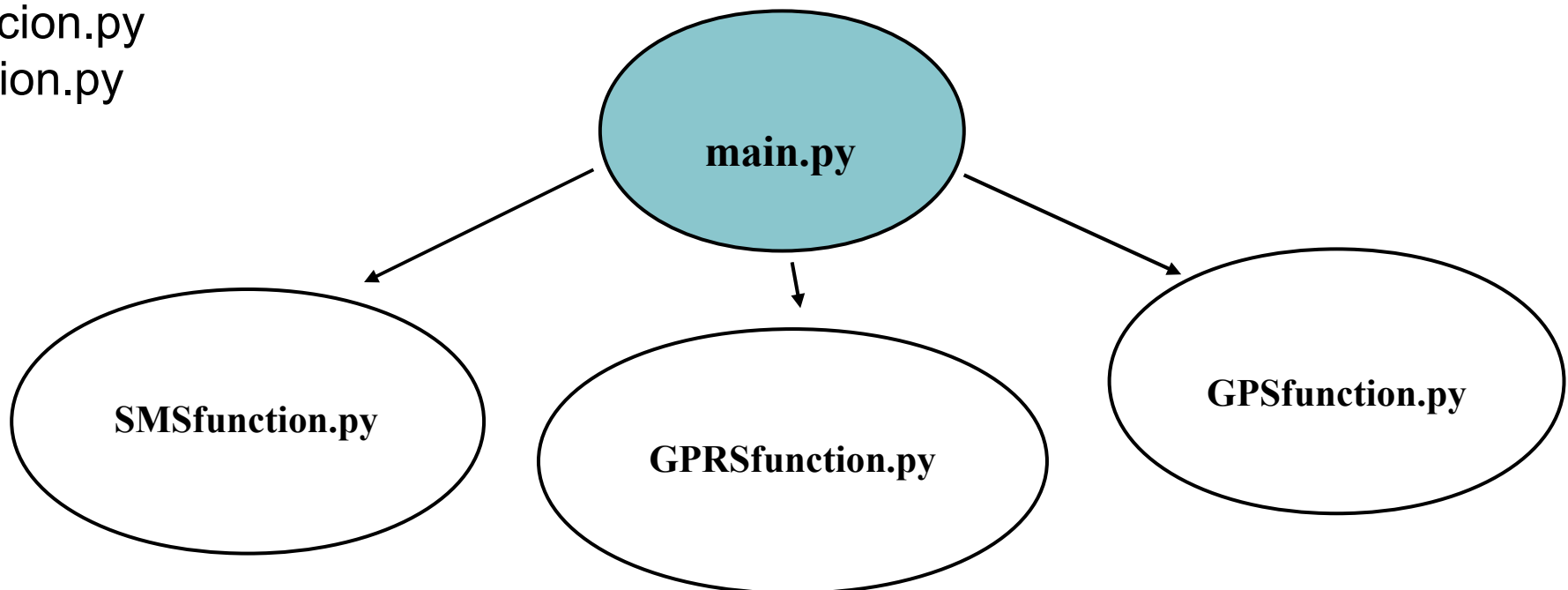
Executing a Python “project”

We call Python “project” a Python program made by a main Python script which calls functions defined in other Python scripts (typical solution).

For example...

Script **main.py** calls functions imported from different script like:

- SMSfunction.py
- GPRSfuncion.py
- GPSfunction.py
-
-



Executing a Python “project”

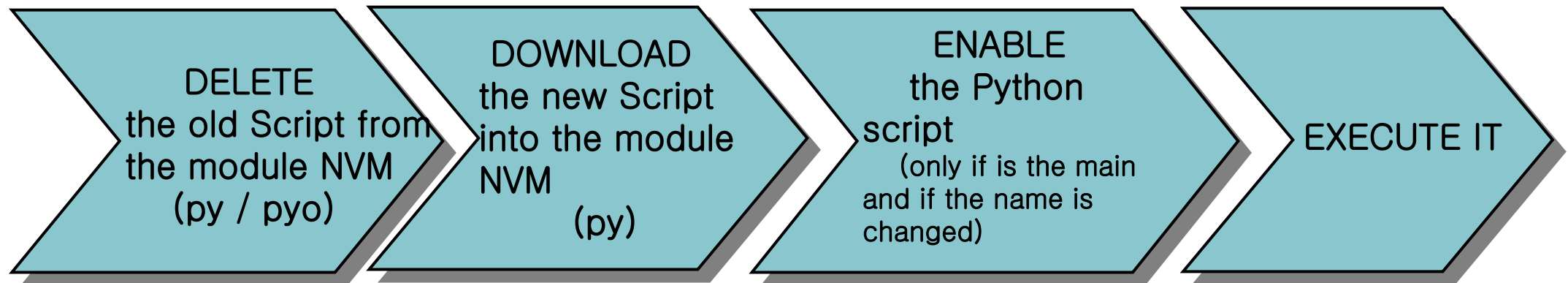


The steps required to have a “project” running in the python engine of the module are:

- 1) **WRITE** all the Python Script of the “project” with any text editor and save
- 2) **COMPILE** all the Python Script of the “project” using Telit Python Package
- 3) **DOWNLOAD** all the Python Script of the “project” into the module NVM
- 4) **ENABLE** the “main” Python script
- 5) **EXECUTE IT**

Updating Your Python script (YOUR APPLICATION)

The steps required to update a python script (your application) are:



How to do this operation ?

Trough serial port (RS232) using a serial terminal interface.

This could be a problem !!!

Updating Your Python script (YOUR APPLICATION)

**APPLICATION ON THE FIELD ! THE BEST WAY?
UPDATE THE APPLICATION
OVER THE AIR.**



Updating Your Python script (YOUR APPLICATION)



Advantages:

- Is not necessary to call back all the applications.
- No physical handling of the applications are required.
- Possibility to update all the applications at same time.

How to do the **OVER THE AIR** update of a Python script?

In your Python application code, you should foresee a step able to:

As soon as the application receives a specific update indication (i.e. SMS with specific string, call from specific number...), an “update function” is executed.

Updating Your Python script (YOUR APPLICATION)

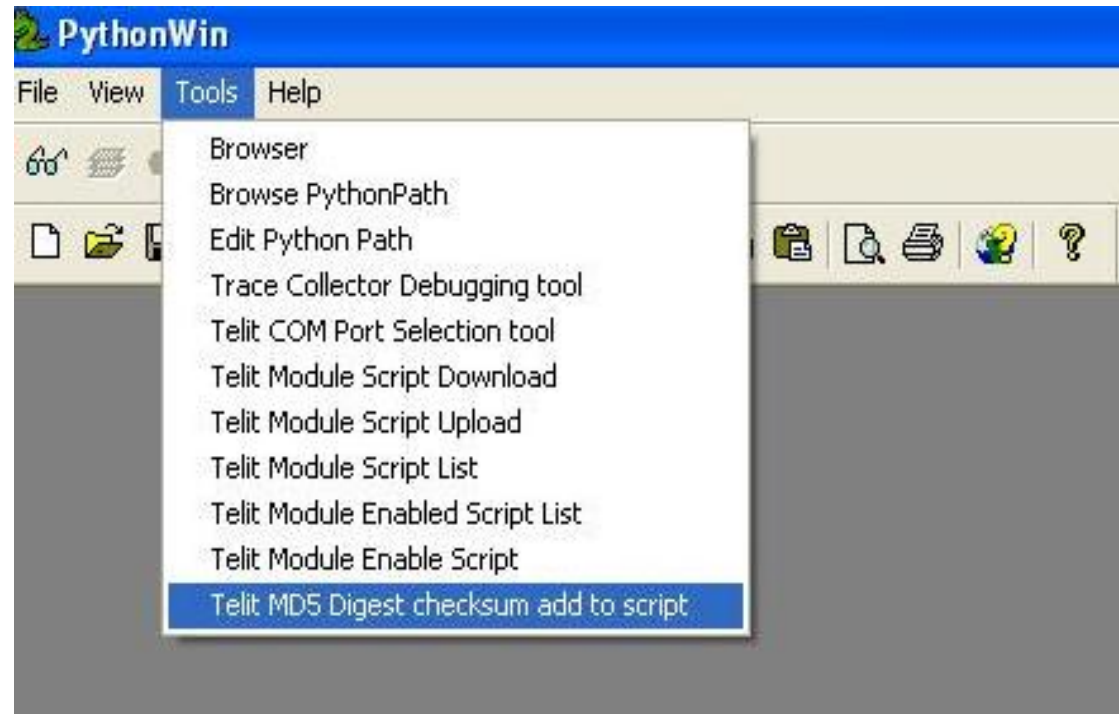


What this “update function” does :

1. The application receives the new (updated) script through:
 - CSD connection.
 - FTP connection on a server where the new script is stored.
2. The “update function” checks the integrity of the new script:
 - Extracting the check sum (put at the end of the new script file and calculated with the md5 function at the moment of the new script creation; do this using *Telit MD5 Digest checksum add to script* tool, in Telit PythonWin).
 - Re calculating the check sum locally (using md5 function).
 - Verifying the match of the two results.

Updating Your Python script (YOUR APPLICATION)

3. If the integrity of the script is OK, the new script can be saved in the NVM with a new name.
4. The “update function” enables the new Script
5. It reboots the system.



Join Telit !

Telit wireless solutions

