



ATT2.0

2nd Generation Automatic Test Tool for AT Cellular



User Manual



Content

<i>History of change</i>	3
1 Introduction – what is ATT?	4
2 Overview and basic setup	6
2.1 ATT2.0 directory structure	6
2.2 Initial setup of configuration files	7
3 ATT file types	8
4 Start and Initial Setup of ATT2.0	9
4.1 Device setup	10
5 Loading and setting up projects	12
6 Running the project	13
7 Creating new projects	14
8 Scripting	15
8.1 Introduction	15
8.2 ATT Command Levels	16
8.3 ATT command reference	17
8.3.1 High level commands	17
8.3.1.1 AT commands	17
8.3.1.2 Strings, Variables and Python Expressions in AT commands	18
8.3.1.3 WAITFOR / WAIT FOR	20
8.3.1.4 COM	22
8.3.1.5 MESSAGE	23
8.3.1.6 WAIT	24
8.3.2 Mid level commands	25
8.3.2.1 About receive buffers	25
8.3.2.2 Variable argument lists	25
8.3.2.3 ATCMD()	26
8.3.2.4 WAITFOR()	27
8.3.2.5 COM()	29
8.3.2.6 MESSAGE()	30
8.3.2.7 USERTEST()	31
8.3.3 Low level commands	32
8.3.3.1 send(), sendln()	32
8.3.3.2 receive()	33
8.3.3.3 ClearReceiveBuffer() / ClearAllReceiveBuffers()	34
8.3.3.4 SetReceiveTimeout()	35
8.3.3.5 ExtractParameter ()	36
8.4 Compatibility and scripting notes	38

History of change

Version	Date	Author	Changes
1.0_001	17.07.2003	Jillissen	Created new document for ATT1.7
1.1	27.05.2004	Weiden	Added some notes, “First steps” guide introduced
2.0	12.08.2004	Weiden	Started adaptation to ATT2.0
2.1	31.08.2004	Weiden	Page layout improved, added ATT command reference guide
2.2	16.09.2004	Weiden	Last changes, no longer ‘preliminary’.

1 Introduction – what is ATT?

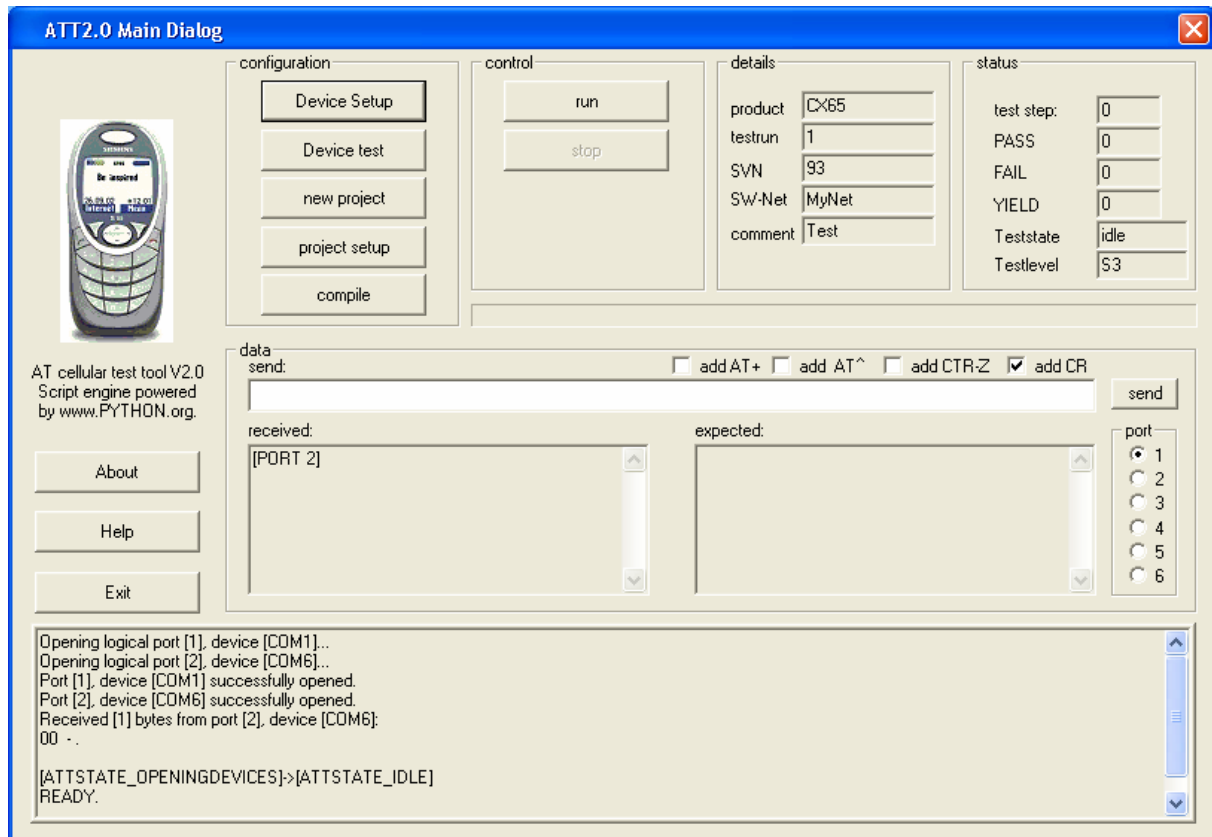


Figure 1.1: ATT2.0 Main dialog

ATT is the “*automatic test tool for AT cellular commands*”.

ATT allows the user to communicate with up to 6 mobiles connected via serial COM ports using AT cellular (ATC) commands. Replies from the mobiles are displayed as soon as ATT receives them. All activated COM ports are internally mapped to 6 logical ATT ports.

ATT2.0 can be used for sending single commands typed in manually at ATT’s command line or automatically by running test scripts.

When running scripts the replies of the mobiles are evaluated and the test results are reported in HTML result files.

It is possible to group several *test scripts* in one *ATT project*, enabling the contained scripts to execute in one go.

ATT2.0 key feature overview:

- Uses PYTHON (www.python.org) as script engine, providing features nowadays expected from high-level scripting languages. The very basic ones: Variables, conditional execution via *if* statement, *for* and *while* loops...
- Easy-to-program test scripts: For most AT command variants, no special syntax is required: They can be scripted as 'pure AT commands'. The ATT2.0 precompiler converts them internally to PYTHON-compatible command syntax.
- Multithreaded I/O: ATT listens for incoming data from all connected mobiles asynchronously in background, thus avoiding data loss or blocked graphical user interface while receiving.
- Detailed result reporting in HTML files.
- Modular device concept using **MAX**, the *Modular Application Toolbox*: For future use, other than serial communication via COM ports can be implemented quite easily, e.g. sockets for network communication. Ideas are always welcome. ☺

2 Overview and basic setup

2.1 ATT2.0 directory structure

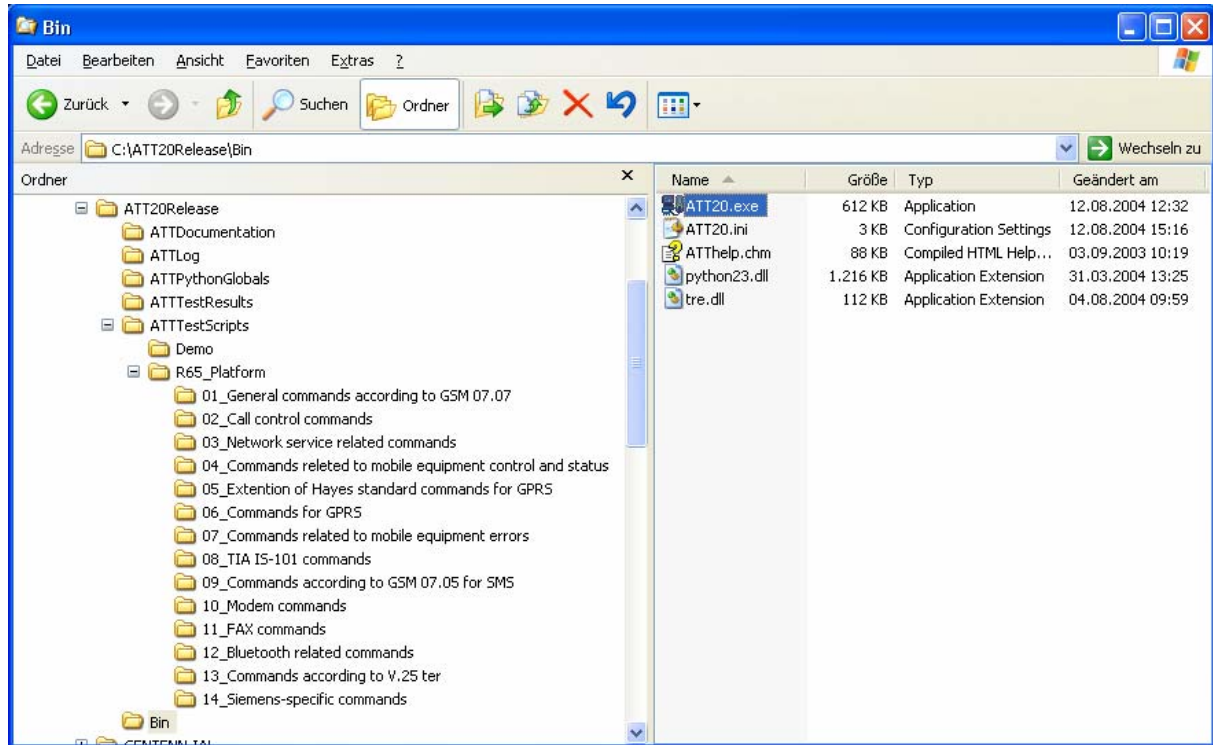


Figure 2.1: ATT2.0 directory structure

After you copied or unzipped the ATT folder to your local drive, you will find all ATT2.0 related stuff in subfolders of *ATT20Release* (see *fig. 1* above).

- In subfolder *Bin* you'll find the executable *ATT20.exe* and the inifile *ATT20.ini*. For convenience, create a shortcut to *ATT20.exe* on your desktop.
- Subfolder *ATTDokumentation* contains all available documentation, e.g. the document you're currently reading. ☺
- *ATTTTestScripts* contains two subfolders: *Demo* contains several example scripts for demonstrating ATT scripting methods. The script file *ATTspecificPythonCommands.txt* demonstrates all currently available ATT-specific PYTHON extension commands. *TwoMobiles.txt* is a typical example showing how two connected mobiles can be tested interactively. All demo files are commented in detail, so looking at these should be a good start for own scripting experiments.

The subfolder *R65_Platform* contains scripts originally written for ATT1.0 and the R65 platform/Ulysses. They were ported to run with ATT2.0. See the README in this folder!

- Folder *AttPythonGlobals* contains all files with global data and functions, imported from most ATT2.0 scripts. This mainly is the file *attglobals.py*, which defines globally used variables (**Phone numbers!**) and constants. Files in this folder are imported from test scripts with the PYTHON `import <FileName> resp. from <FileName> import * statements.`
- *ATTLog* is the subfolder for ATT 's application logging. Per default, ATT writes its application specific logfiles to this directory. In strange error cases, these files might be useful for debugging or tracing of data transferred to/from connected mobiles.
- *ATTTestResults* can be used as output folder for the test result HTML files. This depends on the project settings of the currently loaded ATT2.0 project and can be specified in ATT's *Project setup* dialog (see below).

2.2 Initial setup of configuration files

Open the file *AttPythonGlobals\attglobals.py* (e.g. with notepad) and set up some variables. This file contains variables for phone numbers (*PrimaryMobilePhoneNumber*, *SecondaryMobilePhoneNumber...*), the GPRS access point name and maybe some more. Follow the comments in *attglobals.py*.

That's all for basic setup. You can now start *Att20.exe*.

For more experienced users, the following might be useful (completely optional):

Open *bin/att20.ini*.

You might want ATT2.0 to write its application loggings to somewhere else than the default *ATTLog*, e.g. *C:\temp*. In this case, set variable *LOGDIRECTORY* accordingly. Additionally, you can change variable *PYTHONIMPORTPATH* to your needs, if you have more or other than the standard *AttPythonGlobals* directory for PYTHON files to be imported. Use ';' to separate paths from each other.

3 ATT file types

ATT2.0 uses several types of files:

- **Script files** written by user, containing all required commands for testing. These scripts have the extension **.txt* and can be created with any ASCII text editor. Watch out the examples in the *demo* subdirectory.
- The **PYTHON executable files** with extension **.py*. These files are created from the script files (**.txt*) if they don't already exist or if the **.txt* file modification time is newer than the **.py* file creation time. **.py* files contain the script code transformed into a command syntax that is understandable for ATT's built-in python interpreter. If necessary, creation is done automatically by ATT2.0's precompiler after pressing the *Compile* or *Run* button on ATT's main dialog.
Attention: *Never change the output of the *.py files directly, as they will be overwritten by precompiler again!* Nevertheless, having a look at these files might be informative if the precompiler seems to have become confused about an input script. ☺
- An **ATT Project file** (**.prj*) contains simply the name(s) of one or more script file(s) (without the **.txt*) extension. ATT always runs *projects*, thus executing all script files specified in the currently loaded project file. Projects can be created using ATT's *new project* dialog.
- Probably the python interpreter itself will create **.pyc* files. These are precompiled **.py* files enabling the python interpreter to run scripts faster.

You are missing the reference files **.ref* from ATT1.0? They are no longer required within ATT2.0!

4 Start and Initial Setup of ATT2.0

Start ATT2.0 by executing ATT20.exe. First of all you should setup your devices/ports. Please refer to 4.1, “Device setup”.

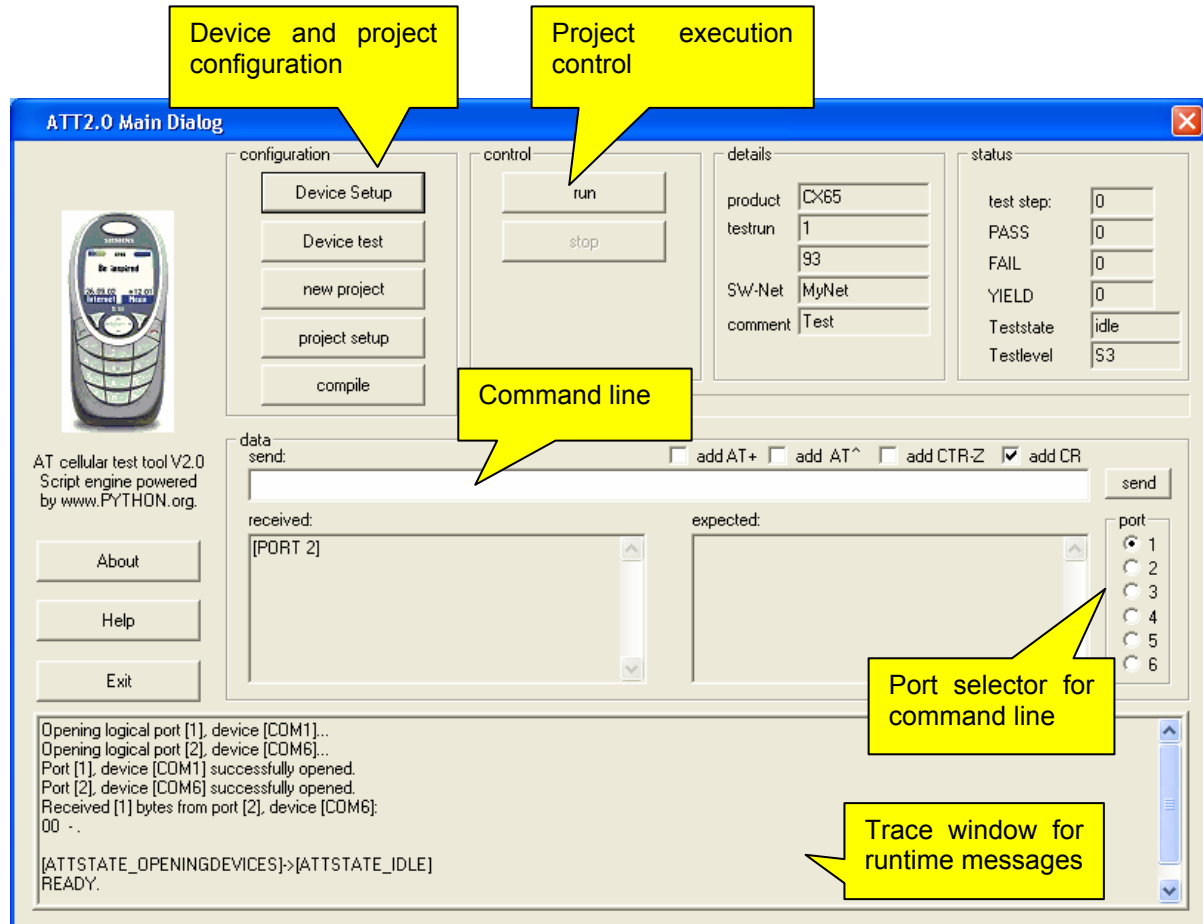


Figure 4.1: The main screen of ATT2.0

For sending an AT command manually enter an AT command in the command line and press the SEND button. By default the command is send via logical port 1. Use the “Port selector” radio buttons under the send button to change this.

To save typing time it is possible to activate the “add AT+” check box. This way instead of typing AT+CGMM only CGMM has to be typed. Ticking “add AT^” does the same with AT^. Most of the commands should be terminated with a carriage return (CR). For this reason “add CR” is selected by default. However, when sending PDU-data etc., the data needs to be terminated with control-Z, so “add CTR-Z” should be ticked instead.

With the button “PROJECT SETUP” all settings for a project like project file, timing, and project specific details can be changed. With the button “run” it is possible to run a previously loaded project. The “COMPILE” buttons creates *.py files from the script files used in the loaded project. This will also be done automatically by pressing “run”, if the *.txt file was changed.

4.1 Device setup

Before it is possible to start any communication it is necessary to set up ATT's logical ports and map them to physical COM ports. Press the button "Device setup" in the main dialog. The dialog shown in Figure 4.2 opens.

The port mapping is done by selecting a logical port in the 'Logical port' dropdown box first and afterwards selecting an available physical device in the 'physical device' list.

Attention: A logical port can only be mapped to exactly one physical device and vice versa. If this is ignored, a message box will open with a warning message.

To delete an existing port mapping, press the "Clear mapping" button. This will clear the mapping between the currently selected logical port and the physical device.

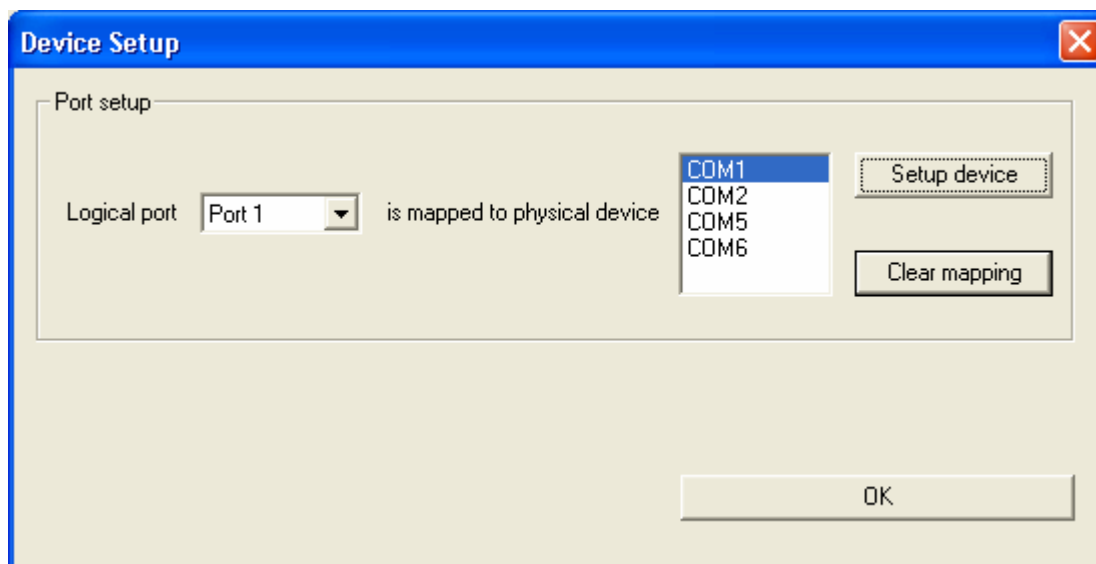


Figure 4.2: Device setup dialog

After selecting the COM port press the "Setup device" button.

The dialog for changing the port properties (Figure 4.3: Port properties) will open. In this window the baud rate, number of data bits, parity, number of stop bits etc. can be configured.

Normally, this would be ≥ 57600 baud, 8 databits, 1 stopbit and no parity (flow control is currently ignored).

Do this for each COM port you mapped to a logical port.

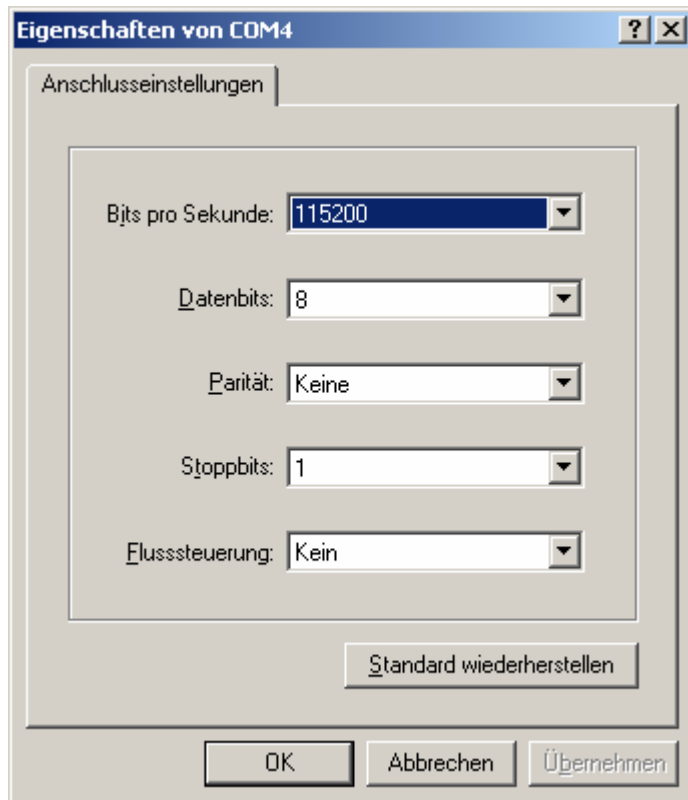


Figure 4.3: Port properties

To test all your settings, return to ATT's main dialog and press the "device test" button. The dialog shown in Figure 4.4: Device test results will inform you about your port mappings, device states and currently connected mobiles, if any.

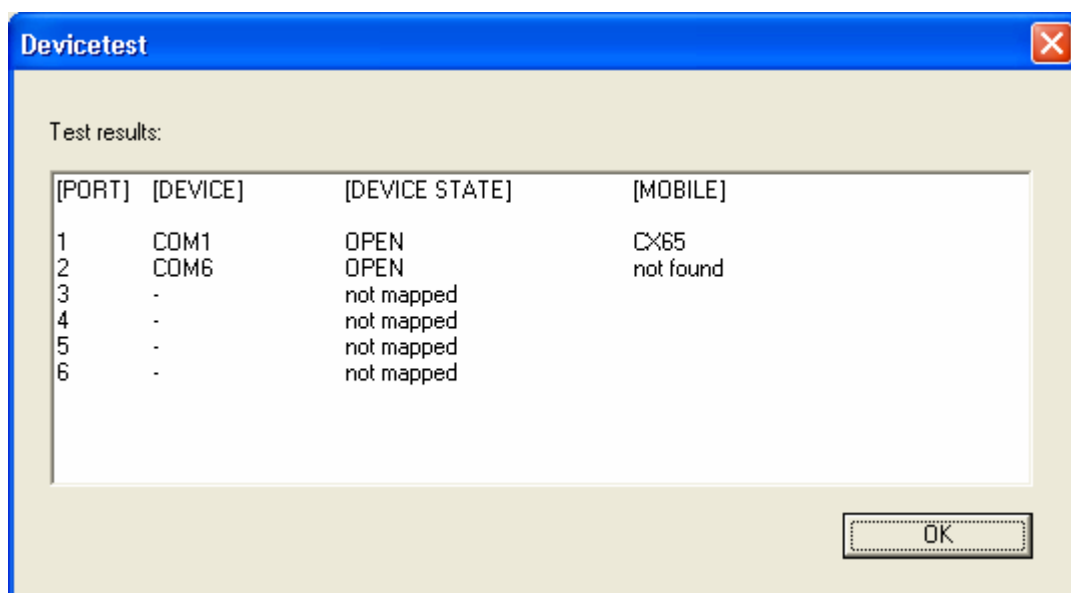


Figure 4.4: Device test results

5 Loading and setting up projects

In the ATT2.0 main dialog select “Project setup”. The project settings window will open:

Project settings

PROJECT SETTINGS

Project path: D:\entw\MAX_Projekte\Max\ATTTestScripts\Demo\

Project name: AttSpecificPythonCommands.prj

Resultfile output directory: D:\ENTW\MAX_PROJEKTE\MAX\ATTTEST

DETAILS

user name: Ralf Weiden

department: ST1

product: CX65

testrun: 1

SVN: 93

SW-Net: MyNet

comment: Test

delay receive-send: 25 ms

delay send-receive: 25 ms

timeout receive: 15000 ms

Testlevel:

Figure 5.1: the Project settings window

The project to be executed can be selected by pressing the “Select project” button. This opens a dialog enabling you to select existing project files *.prj or to specify a folder for new projects to be created.

All HTML result files will be written to the directory selected with the “Select directory” dialog.

All data in the details field, user name, department, product, testrun, SVN, SW-net and comment can be edited for documentation of tests. The value “delay receive-send” is the pause between receiving an answer from the mobile until the next command is send. The “delay send-receive” is the time the program waits after sending an AT command. The “timeout receive” is the maximum time ATT waits for the expected answer from mobile after sending an AT command. The default values *delay recv-send=25ms*, *delay send-recv=25s*, *timeout receive=15000ms* occurred to be useful. If timeouts occur during script runs, probably try to increase the latter.

6 Running the project

After setting up the devices, selecting a project and entering the project settings, pressing the “run” button will start project’s execution. Running projects can be aborted with the “STOP” button.

The testrun results are saved in a HTML result file in the directory selected during project setup (see above). Each result file contains a detailed logging of every test case. A summary is appended at the end of the file, summarizing the project run with the number of executed, passed and failed tests. This summary also includes the general information from the project settings, as shown below:

Summary

project	D:\entw\MAX_Projekte\ATT20 \ATTTestScripts\R65_Platform\01_General commands according to GSM 07.07 \GSM0707_GeneralCommands.prj
date, time	04/09/10, 11:21:17
tester	Ralf Weiden
department	ST1
product	CX65
testrun	1
SVN	_net_
SW net	MyNet
comment	Test
Executed AT commands	40
Executed testcases	44
PASS	42 -> 95 %
FAIL	2 -> 4 %

Tested with ATT2.0

(c) 2004 Siemens AG, Ralf Weiden, ICM MP PD ST1 KLF

based on ATT1.0, initiated by Jan-Carlo Jilissen, now ICM San Diego, California, USA

Naming of result files follows the following conventions:

<Project name>_SVN<software version number>_net<software net>_<date>_<time>.htm

With:

<software version number> and <software net>: Values previously set in the ‘project settings’ dialog.

<date> and <time>: Date/time when test run finished. Format: YYYYMMDD / HHMMSS

7 Creating new projects

An ATT project is a collection of test script (*.txt) files, which are grouped together for being executed one by one. ATT always executes projects, so you need to create one even if it contains only a single script file.

New projects are created using the 'new project' dialog. But first, open the *project setup* dialog (see 5, "*Loading and setting up projects*"), and select the project path for the new project to be located.

In the *new project* dialog, first enter the name of the new project in the corresponding edit field.

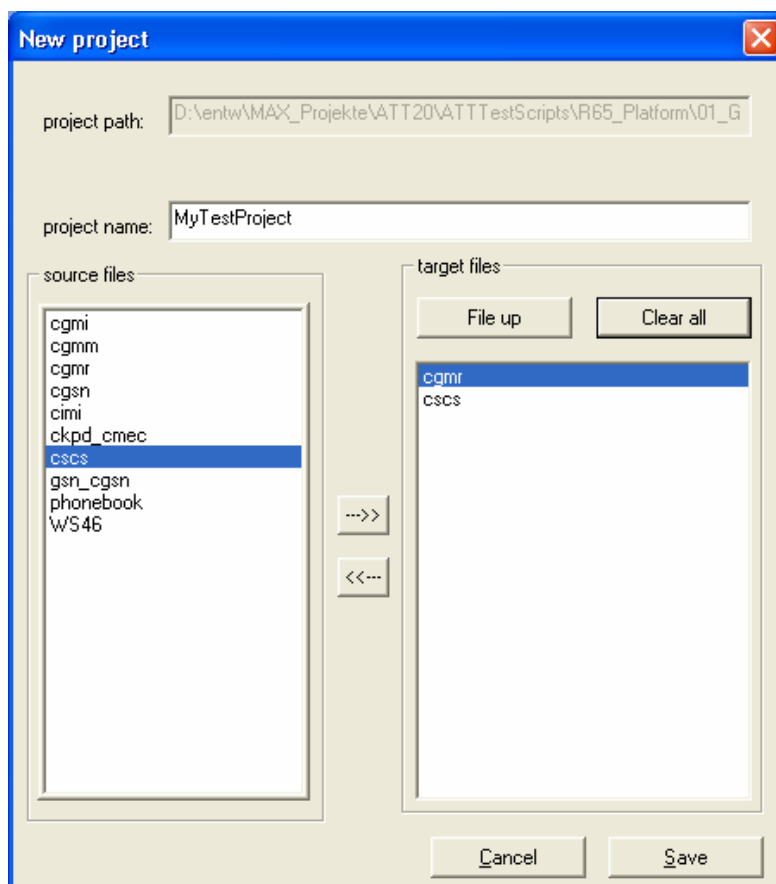


Figure 7.1: The New Project dialog

The list box 'source files' shows all script (*.txt) files found in the current project path. You can add or remove them to / from your new project using the arrow buttons. The order of the scripts in your project can be changed with the 'file up' button.

Finally, press SAVE. Open *project setup* dialog (chapter 5) again and enter additional project information. Afterwards, the project is ready for execution.

The created project file (*.prj) can also be edited using any text editor. That might be required if script files not contained in the current project path should be added to the project. Add them with the complete path, but without the *.txt resp. *.py extension.

8 Scripting

8.1 Introduction

Creating new projects from scratch always starts with writing one or more scripts. Any simple text editor will do. Each line in the file should contain exactly one command.

Commands can be:

- AT cellular commands, like `ATD`, `AT+CCLK?`, `AT+CMGI=?`, etc.
- ATT-specific commands: `ATCMD()`, `WAITFOR()`, `MESSAGE`, `USERTEST()`, `COM...`
- PYTHON build-in statements of the python scripting language: `if`, `for`, `i=i+1`, ...
Shortly, all typical programming language commands. Refer to the *documentation* section at <http://www.python.org> to learn all about PYTHON.

By default, all AT cellular commands in a script will be sent to logical port #1, which is mapped to a physical COM-port via ATT's *device setup* dialog. Sending AT commands to any of the 6 possible logical ports is possible by adding the port number in front of the AT command (or the corresponding `WAITFOR`) like:

```
2:AT+CGMM.
```

Note that there must not be a blank between the port number (`2:`) and the command!

Responses from mobile can be evaluated using the `WAITFOR` command. Each `WAITFOR` defines a testcase for comparing the received response with the expected one, as specified by the `WAITFOR` parameters. Testcase results are written to the HTML result file.

Comments start with `COM` and will be added to the HTML result file for better understanding. A message to the user starts with `MESSAGE`. With messages the user can get notices or instructions, e.g. "MESSAGE This script will overwrite SMS indices 1-10!".

Example of a very simple ATT script:

```
COM Get manufacturer ID code
AT+CGMI
WAITFOR OK
COM Get model ID code from mobile at port 2
2:AT+CGMM
2:WAITFOR OK
MESSAGE Script ended!
```

See also examples in the *demo* folder. All ATT-specific commands are documented in detail in the following command reference.

8.2 ATT Command Levels

ATT2.0's scripting engine knows three hierarchical levels of commands: *High*-, *mid*- and *low level* commands. Of course, all of these commands can be used in script files.

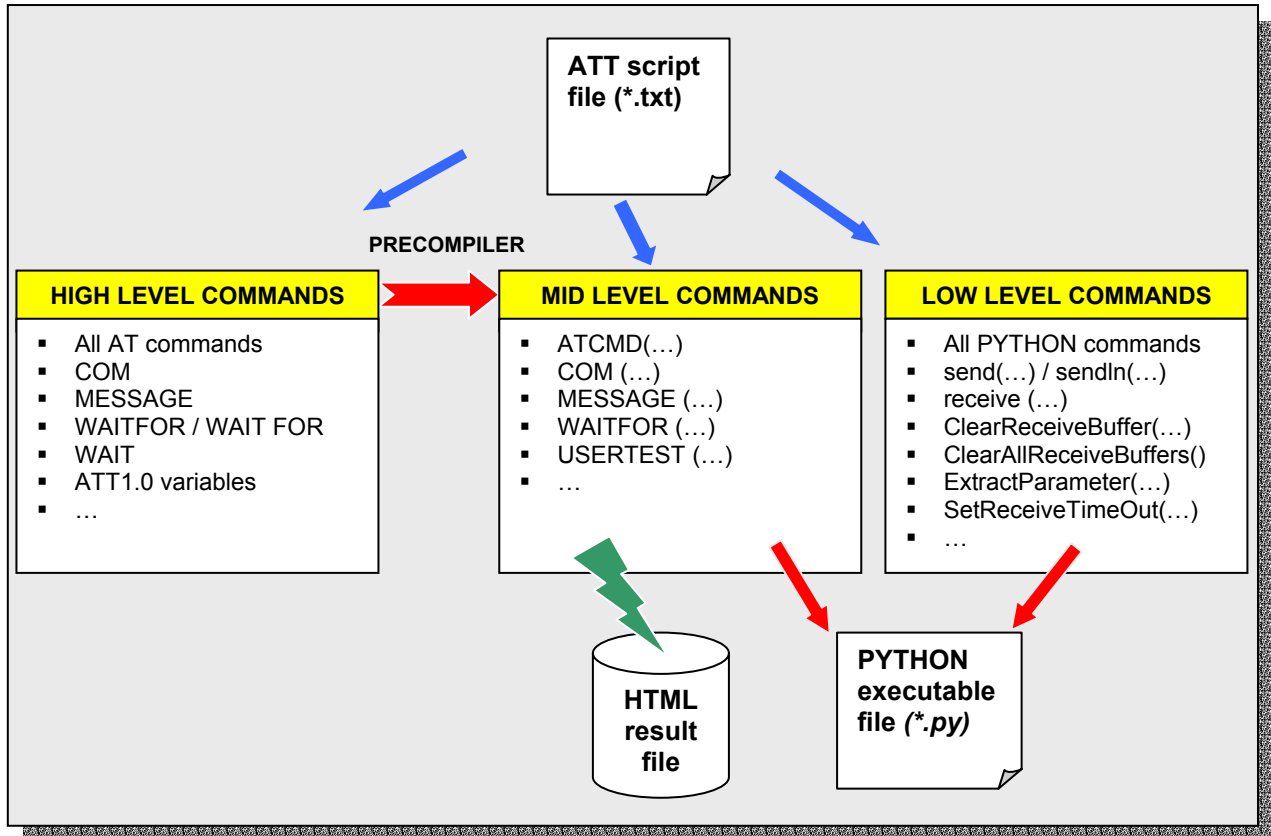


Figure 8.1: ATT command levels

- **High Level Commands** are those without the typical *function call syntax* of common programming languages, that means they don't have the format *functionname (<parameter1>, <parameter2>, ...)*. This obviously are all AT commands, but also those 'simplified' ATT-specific commands like *MESSAGE*, *COM*, *WAITFOR* etc., as already known from ATT1.0. All high level commands need to be processed by ATT's precompiler. The precompiler automatically transforms them into *mid level commands* before script runs, if the script file was changed. Thus, the resulting PYTHON executable files (*.py) don't contain high level commands any more.
- **Mid Level Commands** extend the PYTHON language with ATT-specific functionality. They all have the *function call syntax* directly understandable by ATT's embedded PYTHON interpreter. Most mid level commands are closely related to AT command execution and testing. Usually their results will be written to the HTML result file.

- **Low Level Commands** are quite similar to mid level commands. Main difference is that they don't write output to the HTML result file. They mainly provide basic input/output and control functionality. In this function hierarchy, also python's build-in, "*native*" functions could be subsumed here.

8.3 ATT command reference

8.3.1 High level commands

8.3.1.1 AT commands

ATT2.0 is intended to handle all AT commands according to 3GPP TS 27.007, 3GPP TS 27.005 and ITU-T V250 directly, without any modifications, in ATT scripts. This includes the siemens-specific extensions to the standards. For a basic introduction e.g. refer to the reference manual "*AT command set for Siemens mobile phones and modems*".

All AT commands sent to any connected mobile are written to the HTML result file.

3GPP AT cellular commands always start with 'AT', followed by +, ^, or &, followed by the remainder of the command.

Examples:

AT+CCLK?	# execute command
AT^SACM=?	# check command
AT+CMEC=2	# write command

ITU-T V250 ("Hayes standard commands") are the "classical" modem commands. They all start with "AT", followed by further letters and numbers. They **MUST NOT** contain parenthesis () or the equation sign "=", because otherwise ATT's precompiler assumes them to be PYTHON function calls or variable assignments.

Examples:

```
ATE0
ATD02842950;
ATD>2;
```

These all are valid AT-Commands which can be used directly in ATT scripts. But the following commands will be left unprocessed by the precompiler, either causing PYTHON errors/script termination or strange behaviour during execution:

ATS3=9 will be treated as a variable assignment ("Let variable ATS3 be 9")
ATD(MyPhoneNumbersArray[5]); is recognized as function call to function *ATD()*

Of course, these examples are quite exotic and somewhat constructed. Anyway, if in trouble, use the mid level command *ATCMD()* (see section 8.3.2.3) in conjunction with the questionable AT command.

8.3.1.2 Strings, Variables and Python Expressions in AT commands

Variables:

ATT2.0 recognizes two variants of variables:

- **Old style ATT1.0 variables:** ATT1.0 syntax used angle brackets ‘<’ and ‘>’ to mark variables: `<MyVarName> = ...` or `ATD <MyVarName>;`. For compatibility reasons, this syntax is still recognized by ATT2.0. The precompiler transforms them to ‘new style variable syntax’ as described in section 8.4, “*Compatibility and scripting notes*”. Unlike ATT1.0, ATT2.0 does variable value assignment/resolving during script runtime, *not* while compiling, thus providing much more flexibility. Old style variables shouldn’t be used any more in new scripts.
- **New style variables** are not marked with angle brackets. **Their first character must be either a letter or an underscore ‘_’.** Actually, ATT2.0’s variables are *PYTHON* variables and completely handled by the PYTHON interpreter. Please refer to the python documentation for all about value assignment, PYTHON data types and type conversion. ☺ Of course, this means that also object oriented programming is possible, as provided by PYTHON.

Variables can be used in AT *write* commands (`AT+...=n`) and the dial command `ATD`:

```
# call phone with PrimaryMobilePhoneNumber
ATD PrimaryMobilePhoneNumber;

# reads entry iIndex from short message storage
iIndex=17
AT+CMGR=iIndex

# send message from Index iIndex to Mobile with SecondaryMobilePhoneNumber
AT+CMSS=iIndex, SecondaryMobilePhoneNumber, 145
```

Strings:

Strings in AT *write* commands (`AT+...=n`) always have to be scripted using double quotes, like `"This is my simple demo string"`.

Correct example:

```
AT+CPMS="MT", "MT", "MT"
```

Incorrect example:

```
AT+CPMS=MT, MT, MT
```

In the latter case, the precompiler would handle `MT` as a python variable. This will cause the PYTHON interpreter to terminate with error, because `MT` is an unknown identifier (Of course not if `MT` really is a variable and defined somewhere before... ☺).

The double quotes “...” are part of the AT command and will be sent to the mobile. If that’s not desired, please use the mid level command `ATCMD()` or use single quotes, as used by python itself (see below).

Please note that **phone numbers** in AT *write* commands are also strings, so the above is also valid. The one and only exception is the command *ATD*: Here, phone numbers can be used directly:

```
ATD +492842950;  
ATD 02842950;
```

These both are possible commands.

Note that PYTHON itself uses single quotes `'...'` (the apostrophe) to mark strings: `'This is my simple demo string.'`

So, in all expressions other than AT *write* commands, use this python syntax, e.g. for variable assignment:

```
PrimaryMobilePhoneNumber='+492842950'  
ATD PrimaryMobilePhoneNumber;
```

Globally used phone numbers are assigned to global variables in the file *attglobals.py*. Import this file at the beginning of your script with *from attglobals import **, if required.

Python Expressions:

In fact, everything that is *not* recognized as a string (*not* embedded in double quotes `"..."`) or a simple number will be treated as a python expression. This includes variables. Python expressions are evaluated by the PYTHON interpreter during execution and may contain mathematical expressions, function calls, etc.:

```
# reads entry iIndex with offset 10 from short message storage:  
AT+CMGR=10+iIndex  
  
# reads entry 21 from short message storage:  
AT+CMGR=3*7  
  
# Function call. Of course, GetCurrentIndex() must be defined somewhere!  
AT+CMGR=GetCurrentIndex()  
  
# set service center address SCA to a python string expression  
AT+CSCA='+49234599999'
```

Again, if you really want to send a string like `"3*7"` to the mobile, don't forget the double quotes `"..."` or use the mid level command *ATCMD()*!

8.3.1.3 WAITFOR / WAIT FOR

Most AT commands are followed by *WAITFOR* resp. *WAIT FOR* (no difference in behaviour) to check mobile's answer.

Syntax:

```
n:WAITFOR  strExpectedData      resp.  
n:WAIT FOR strExpectedData
```

Parameters:

n:

Logical port [1..6] to wait for result. Optional, can be left out. Default: 1

strExpectedData

(Sub-) String which has to be part of mobile's response. The command always interprets *strExpectedData* as an ASCII string. Variables and python expressions are not recognized. If required, use the mid level command *WAITFOR()* (see 8.3.2.4, "*WAITFOR()*").

Remarks:

WAITFOR waits until either the expected answer from mobile is received or receive timeout exceeds. Timeout time can initially be set in *project settings* dialog (see 5, "*Loading and setting up projects* ") or be changed during script runtime using the low level command *SetReceiveTimeout()* (see 8.3.3.4).

Each *WAITFOR* defines a testcase. Its result will be written to the HTML result file. The testcase is passed if *strExpectedData* is contained somewhere in the received data. If *strExpectedData* is not part of the received data, the testcase fails after timeout. The corresponding entry in the HTML file will be marked with a red background (or green if passed).

Example:

```
# get time and date from mobile  
AT+CCLK?  
WAITFOR +CCLK:
```

The mobile will respond to AT+CCLK? with something like:

```
AT+CCLK?  
  
+CCLK: "03/01/02,15:59:24"  
OK
```

As '+CCLK:' is part of the received answer, this testcase is passed. For exact matching (received exactly equal to expected data) you would need CR and LF (and probably variables etc.) to be part of *strExpectedData*. Use the mid-level command *WAITFOR()* in such cases.

Please note: Some responses from mobile may be received in multiple segments. It may take some time until the complete response is received. Imagine the following long response to AT+COPN:

```
AT+COPN
+COPN: "21303","STA-MOBILAND"
+COPN: "42402","UAE ETISALAT"
...
... many more lines like above
...
+COPN: "52023","GSM 1800"

OK
```

A statement like `WAITFOR +COPN: "21303"` will pass at the very beginning of mobile's answer, causing ATT to continue script execution immediately, regardless of mobile is still sending its remaining response. In consequence, the data in receive buffer might get corrupted, if in the meanwhile ATT sent subsequent AT commands.

To avoid such problems, simply add a `WAITFOR OK`, insuring that mobile finished its response completely. The complete example script:

```
AT+COPN
WAITFOR +COPN: "21303"
WAITFOR OK
```

8.3.1.4 COM

COM adds a comment to the HTML result file. There, comments always occur with a yellow background and can contain any descriptive information.

Syntax:

```
COM strMyComment
```

Parameters:

strMyComment

Comment string to be written to the result file. The precompiler always interprets *strMyComment* as an ASCII string. Variables and Python expressions are not recognized. If required, use the mid level command *COM()* (see section 8.3.2.5).

Remarks:

You can use all HTML tags in *strMyComment*, allowing a lot of sophisticated output formatting, like <H1>...</H1> for headline style or
 to add a linefeed.

Note that your scripts should also make massive use of the standard python comment (beginning with the '#'), e.g. for structuring and commenting the script file itself!

Example:

```
COM <H1>This is a comment in headline style!</H1>
COM This is the first line <br> This is the second line

# This is a PYTHON comment.
# Python comments of course will not be written to HTML result file.
```

These lines will produce the following output in HTML file:

This is a comment in headline style!

This is the first line
This is the second line

8.3.1.5 MESSAGE

MESSAGE opens a message box, displaying any information to the user. Script execution waits until user clicks the 'OK' button in the message box.

Syntax:

```
MESSAGE strMyMessage
```

Parameters:

strMyMessage

Information string to be displayed in the message box. The precompiler always interprets *strMyMessage* as an ASCII string. Variables and Python expressions are not recognized. If required, use the mid level command *MESSAGE()* (see 8.3.2.6, "*MESSAGE()*").

Remarks:

MESSAGE is intended for displaying simple information. *MESSAGE* does not write any output to the HTML result file.

If you want the user to execute a test manually, also see mid level command *USERTEST()*, section 8.3.2.7.

Example:

```
MESSAGE Attention! This script will overwrite messages in SMS storage!
```

8.3.1.6 WAIT

WAIT pauses the script execution for the time specified. (Equivalent to Windows/MFC's *Sleep()*).

Syntax:

```
WAIT iWaitTime
```

Parameters:

iWaitTime

Non-negative integer: Time to wait in milliseconds.

Remarks:

The precompiler transforms *WAIT* to the Python build-in command *time.sleep()*. This requires the python module *time* to be imported. This is done implicitly if you import *attglobals.py*, which is standard in most cases. So add one of the following statements at the beginning of your script:

```
from attglobals import *
```

OR

```
import time
```

Example:

```
from attglobals import *
```

```
# ...  
# some code  
# ...
```

```
# Wait 500ms  
WAIT 500
```


8.3.2 Mid level commands

8.3.2.1 About receive buffers

Each logical port of ATT has its own, independent receive buffer. Data received from connected mobiles will be appended to the corresponding receive buffer. This continuously happens in background, thus buffering all incoming data for later evaluation without data loss.

Every time an AT command is send, all receive buffers (of all logical ports!) are cleared automatically. This happens implicitly during execution of the mid-level command *ATCMD()* (see 8.3.2.3, “*ATCMD()*”). In this way, ATT is ready to receive the responses caused exactly by the last AT command from all affected mobiles, enabling interactivity tests. As one example, simply imagine one mobile connected to logical port 1, calling another one at port 2:

TEST SCRIPT	Buffer 1 (Responses from Mobile 1)	Buffer 2 (Responses from Mobile 2)
<pre>1:ATD SecondaryPhoneNumber; 1:WAITFOR OK 2:WAITFOR RING</pre>	<pre>...old content... cleared ATD01722595500; OK</pre>	<pre>...old content... cleared dialling RING</pre>

Receive buffers can be cleared explicitly using low level commands *ClearReceiveBuffer()* and *ClearAllReceiveBuffers()* (see section 8.3.3.3) if required.

8.3.2.2 Variable argument lists

Several mid level commands, like *ATCMD()*, *COM()* and *USERTEST()*, accept a variable number of parameters (at least one), known as *variable argument list*. This argument list can be a mixture of python strings, variables and python expressions. The related commands will concatenate all given argument values to one single string. Variable argument lists are identified by the closing *Param1, ...)* in the command's syntax documentation. Examples:

```
ATCMD (1, 'ATD', SecondaryMobilePhoneNumber, ';')
will result in something equivalent to:  ATD01729594857;
```

```
iIndex=17
COM ('Index+4 = ', iIndex+4)
will result in: Index+4 = 21
```

8.3.2.3 ATCMD()

ATCMD() is used for sending an AT command via a specified logical port.

Syntax:

```
ATCMD (iLogicalPort, Param1, ...)
```

Parameters:

iLogicalPort

Logical port (1..6) for sending the AT command

Param1, ...

Variable argument list containing the AT command itself (as a string) and probably several further parameters. See 8.3.2.2, "Variable argument lists".

Remarks:

All AT commands directly used in ATT scripts will be transformed to parameter(s) within *ATCMD()* by ATT's precompiler. Logical port 1 is default, if the AT command does not explicitly specify another.

Each *ATCMD()* command internally clears all receive buffers of all logical ports, as described in section 8.3.2.1, "About receive buffers".

ATCMD() adds all sent AT commands to the HTML result file in the following form:

PORT 1	ATD+4917620132051;	Command 23	[NonAnswered500.py, line 28]
Logical port	AT command, as sent to mobile (after processing variable argument list)	Project-global AT command counter	Command's file location

Script execution will terminate with error if *iLogicalPort* is invalid or not open.

Examples:

```
ATCMD (1, 'AT+CCLK?')           # simple AT command with no variables etc.
```

```
MyPhoneNumber='+492842950'
```

```
ATCMD (1, 'ATD', MyPhoneNumber, ';')      # will result in ATD+492842950;
```

```
ATCMD (1, 'AT+CMGR=', GetCurrentIndex()) # function call also possible
```

8.3.2.4 WAITFOR()

Mid level equivalent to *WAITFOR*. Defines a testcase for checking mobile's response to a previously sent AT command and compares it with the expected answer. *WAITFOR()* allows waiting for variable values, results of function calls, etc., and further processing of received responses.

Syntax:

```
strReceivedData WAITFOR (iLogicalPort, Param1, ...)
```

Parameters:

iLogicalPort

Logical port (1..6) for listening to incoming data.

Param1, ...

Variable argument list defining the expected data to be received. This may be composed using strings, variables and python expressions. See 8.3.2.2, "Variable argument lists".

Return Value:

WAITFOR() returns the complete content of the receive buffer for further evaluation. If no data was received (*WAITFOR()* timed out), python object *None* is returned.

Remarks:

See also the *Remarks* and *Please Note* section of high level *WAITFOR*, chapter 8.3.1.3. *WAITFOR()* writes its result to the HTML file, like follows:

PORT 1	EXPECTED :	Testcase 4	[NonAnswered500.py, line 30]
	OK		
	RECEIVED:		
	ATD+4917620132051;		
	OK	PASSED	

Expected data, as specified in *WAITFOR()*

Project-global testcase counter

Command's file location

Logical port

Complete data received from mobile, if any

Result (PASSED / FAILED)

Example 1:

```
# get current date/time from mobile connected to port 2
2:AT+CCLK?
strResult=WAITFOR (2, '+CCLK:')

if strResult is not None:
    # Mobile answered -> write result to HTML file
    COM ('Response to AT+CCLK:\r\n', strResult)
else:
    # receive timeout -> display message
    MESSAGE No answer from mobile at port 2!
```

Example 2:

```
# check complete response, including variable values and CR+LF (hex 0D/0A)

iValue=2
AT+CMEC=iValue
```

The Mobile will response with:

```
AT+CMEC=2
```

```
OK
```

For verifying the complete response, including variable value and carriage return/linefeed, the appropriate WAITFOR looks like:

```
WAITFOR (1, 'AT+CMEC=', iValue, chr(0x0D), chr (0x0D), chr (0x0A), 'OK')
```

8.3.2.5 COM()

`COM()` adds a comment to the HTML result file. Mid level equivalent to `COM` (see 8.3.1.4, "COM").

Syntax:

```
COM (Param1, ...)
```

Parameters:

Param1, ...

Comment to be written to HTML file, as variable argument list. This may be composed using strings, variables and python expressions. See 8.3.2.2, "Variable argument lists".

Remarks:

See *Remarks* section of high level command `COM`, section 8.3.1.4.

Example:

```
iIndex=17
COM ('Index+4 = ', iIndex+4)
# also possible to add carriage return/linefeed:
COM ('Index+4:\r\n', iIndex+4)
```

8.3.2.6 MESSAGE()

`MESSAGE()` opens a message box, displaying any information to the user. Script execution waits until user clicks the 'OK' button in the message box.

Mid-Level equivalent to `MESSAGE`, (see 8.3.1.5, "`MESSAGE`"), allowing usage of variables etc.

Syntax:

```
MESSAGE (Param1, ...)
```

Parameters:

Param1

Variable argument list specifying information to be displayed in the message box. This may be composed using strings, variables and python expressions. See 8.3.2.2, "*Variable argument lists*".

Remarks:

`MESSAGE()` is intended for displaying simple information. `MESSAGE()` does not write any output to the HTML result file.

If you want the user to execute a test manually, see also mid level command `USERTEST()`, section 8.3.2.7.

Example:

```
PhoneNumber='+492842950'  
MESSAGE ('Dialling number ', PhoneNumber, '! Please wait...')
```

8.3.2.7 USERTEST()

Quite similar to `MESSAGE()`, `USERTEST()` opens a dialog box, asking the user to perform a test manually, e.g. , check a display content or check if a called phone really rings. The opened dialog always has three buttons: 'PASSED', 'FAILED' and 'SKIP', allowing the user to react in an appropriate way. Script execution waits until user clicks any of the three buttons. The usertest result will be written to the HTML result file.

Syntax:

```
iResult USERTEST (Param1, ...)
```

Parameters:

Param1

Variable argument list specifying the test the user should perform. This may be composed using strings, variables and python expressions. See 8.3.2.2, "*Variable argument lists*".

Return value:

-1, 0 or 1, depending on the button the user clicked:

```
-1:  'FAILED'
0:   'SKIP'
1:   'PASSED'
```

Example:

```
PhoneNumber='02842958644'
ATD PhoneNumber;
```

```
iResult=USERTEST ('Dialling ', PhoneNumber, '...\r\nPlease check if called
phone rings!')
```

```
if iResult==0:
    MESSAGE Ooohps! Lazy today? ;-)
```

Depending on the button the user clicked, `USERTEST()` writes an output to the HTML result file like:

USER	USERTEST:	Testcase 1	[CompileTest.py, line 12]
	Dialling 02842958644...	PASSED	
	Please check if called phone rings!		

8.3.3 Low level commands

8.3.3.1 send(), sendln()

Low level functions for sending any data via a specified logical port.

Syntax:

```
send (iLogicalPort, strSendBuffer)
sendln (iLogicalPort, strSendBuffer)
```

Parameters:

iLogicalPort

Logical port (1..6) to be used for sending

strSendBuffer

Python string with data to be sent. Variable argument lists are not supported.

Remarks:

send() sends the content of *strSendBuffer*, without any modification. *sendln()* appends carriage return / linefeed (0x0D / 0x0A).

As low level commands, both don't write any information to the HTML result file.

Script execution will terminate if *iLogicalPort* is invalid or port not opened.

Examples:

```
send (1, 'AT+CCLK?')
```

```
# demonstrate sendln() in conjunction with python string operations
```

```
MyPhoneNumber='+492842950'
```

```
sendln (1, 'ATD'+MyPhoneNumber+';')           # will result in ATD+492842950;
```


8.3.3.2 receive()

Low level function for receiving incoming data via a specified logical port.

Syntax:

```
strReceivedData receive (iLogicalPort)
```

Parameters:

iLogicalPort

Logical port (1..6) for listening to incoming data.

Return Value:

Complete receive buffer content of the corresponding logical port, if data available.
If no data was received (*receive()* timed out), python object *None* is returned.

Remarks:

If the logical port's receive buffer contains previously received data, *receive()* returns immediately, without waiting for further incoming data.

If no data is available, *receive()* waits until either data is received or receiving times out. Timeout time can initially be set in *project settings* dialog (see 5, "Loading and setting up projects ") or be changed during script runtime using the low level command *SetReceiveTimeout()* (see 8.3.3.4).

Received data will be appended to the internal, port-specific receive buffer without deleting already existing content. If an empty buffer is desired before receiving, call *ClearReceiveBuffer()* (see section 8.3.3.3 and 8.3.2.1, "About receive buffers").

Example:

```
# check if a mobile is connected to port 2
# first set timeout time to 1s
iOldTimeout=SetReceiveTimeout(1000)

# now simply send 'AT'
sendln (2, 'AT')
strReceived=receive(2)

if strReceived is not None:
    # Mobile answered -> show receive result in message box
    MESSAGE ('Response to AT:\r\n', strReceived)
else:
    # receive timeout -> display message
    MESSAGE No answer from mobile at port 2!
# reset timeout time to old value

SetReceiveTimeout(iOldTimeout)
```

8.3.3.3 ClearReceiveBuffer() / ClearAllReceiveBuffers()

Functions for clearing internal receive buffers.

Syntax:

```
ClearReceiveBuffer(iLogicalPort)  
ClearAllReceiveBuffers()
```

Parameters:

iLogicalPort

Logical port (1..6) whose receive buffer is to be cleared.

Remarks:

The first function clears the internal receive buffer of the specified logical port, *ClearAllReceiveBuffers()* clears all receive buffers. See 8.3.2.1, “About receive buffers” for more information about receive buffer handling.

8.3.3.4 SetReceiveTimeout()

Sets the timeout time for all commands which actively wait for incoming data (*WAITFOR()*, *receive()*).

Syntax:

```
iOldTimeoutTime SetReceiveTimeout (iNewTimeoutTime)
```

Parameters:

iNewTimeoutTime

Non-negative integer with new timeout time (in milliseconds) to be set.

Return Value:

Old timeout time which was active before the call to *SetReceiveTimeout()*.

Remarks:

Initial timeout time is defined in *project settings* dialog (see 5, "Loading and setting up projects").

As a general idea, it's worth considering setting initial project's timeout time to a relatively short value, e.g. 5 seconds. This will shorten the time ATT waits for the expected response from mobile if a testcase fails, thus accelerating the whole testrun. As most AT commands will respond in a relatively short amount of time (also in error cases), this will mostly be appropriate.

Only a handful of commands take longer. These especially are all commands with network interaction (*ATD*, sending SMS, checking/setting call forwarding, etc.). Before sending such a command, simply increase timeout time temporarily, using *SetReceiveTimeout()*. Afterwards, reset timeout time to the old value again, as returned by the first call to *SetReceiveTimeout()*.

Example:

See example *receive()*, section 8.3.3.2.

8.3.3.5 ExtractParameter ()

ExtractParameter() is useful for extracting a specified parameter value (a substring) out of a complete string returned from mobile.

Syntax:

```
strParameterValue ExtractParameter (strInput, iNoOfParameterToExtract,  
                                   strStartAfter='', strSeparatorTokens=',')
```

Parameters:

strInput

Input string from which a substring is to be extracted as parameter.

iNoOfParameterToExtract

Parameter number (non-negative integer), specifying the parameter to extract. Counting of parameters starts after *strStartAfter*. Parameters are separated by one of the chars contained in *strSeparatorTokens*.

strStartAfter

Starting string sequence (optional). Parameter extraction starts after the first occurrence of *strStartAfter* in the given string *strInput*. If *strStartAfter* is not contained in *strInput*, *ExtractParameter()* fails. This parameter is optional. By default, the parameter extraction starts at the beginning of *strInput*.

strSeparatorTokens

(Optional) string containing the separator(s) which separate(s) the parameters in *strInput*. This parameter is optional. Default: ','.

Return Value:

Extracted parameter value (maybe an empty string).

Example:

This example demonstrates how to extract the SMS service center address (SCA) and its number format from mobile, as returned from command AT+CSCA?.

The mobile will response to AT+CSCA? like:

```
AT+CSCA?  
+CSCA: "+491722270333",145  
  
OK
```

Parameter extraction should start after +CSCA: ", so that's the value for *strStartAfter*. The SCA and its number format ("145") are separated by " (double quote) and , (comma), so these both will be used as separator tokens. Additionally, the number format is followed by a line feed (\n), so that's also part of *strSeparatorTokens*.

The corresponding script:

```
# get service center address (SCA) from mobile  
AT+CSCA?  
strCSCAResult=WAITFOR (1, '+CSCA:')  
  
# extract SCA (first parameter after +CSCA: ")  
strSCA=ExtractParameter (strCSCAResult, 1, '+CSCA: "', '"', '\n')  
  
# extract SCA number format (second parameter after +CSCA: ")  
iSCAFormat=int (ExtractParameter (strCSCAResult, 2, '+CSCA: "', '"', '\n'))
```

See also demo script *AttSpecificPythonCommands.txt* for another example.

8.4 Compatibility and scripting notes

As ATT2.0 uses a completely new architecture, especially the PYTHON interpreter, there might be compatibility problems if you try to run old scripts written for ATT1.0.

ATT2.0's precompiler tries to get the old scripts running without any changes. That should be successful if no variables or some strange command variants are used in ATT1.0 scripts.

For ATT1.0 variable resolving, the [COMPATIBILITY] section was introduced to the ini file *att20.ini*. Old variables contained in this section are simply mapped to new names used by ATT2.0. Names of old variables not found in this section will be left unchanged.

Variable resolving in conjunction with AT commands may lead to ambiguity (with both ATT2.0 and 1.0 scripts), causing the PYTHON interpreter to terminate with errors. Imagine the following AT command in a script file:

```
AT+CPMS=MT,MT,MT
```

Now, what's MT? Is this a PYTHON variable or a string?

Per default, ATT2.0 presumes all not recognized expressions to be a python expression, in this case a variable. But of course, at runtime, the PYTHON interpreter terminates because MT is an unknown identifier.

As described in 8.3.1.1, "AT commands", the solution is: Always use double-quotes "..." for marking string expressions, in this case:

```
AT+CPMS="MT", "MT", "MT"
```

If that's not possible, use the ATT2.0 specific PYTHON command ATCMD():

```
ATCMD (1, 'AT+CPMS=MT, MT, MT')
```

This will be the solution for all (I hope! ...) cases the precompiler fails to interpret a command correctly. This also is valid for *WAITFOR*, *COM* etc.: If in trouble, use the 'mid level' python commands *WAITFOR ()*, *COM ()* etc.

See also the files in *demo* folder for examples!