LLD Extension Mechanism

Project	TCS 3.x
Document Type	Technical Documentation
Title	LLD Extension Mechanism
Author	Liyi Yu
Creation Date	06.05.2004
Last Modified	
ID and Version	
Status	Being Processed

Copyright © 2002-2003 Texas Instruments, Inc. All rights reserved.

Texas Instruments Proprietary Information – Strictly Private

0 Document Control

© Copyright Texas Instruments, Inc. 2002-2003
All rights reserved.

Every effort has been made to ensure that the information contained in this document is accurate at the time of printing. However, the software described in this document is subject to continuous development and improvement. Texas Instruments reserves the right to change the specification of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of Texas Instruments. Texas Instruments accepts no liability for any loss or damage arising from the use of any information contained in this document.

The software described in this document is furnished under a license agreement and may be used or copied only in accordance with the terms of the agreement. It is an offence to copy the software in any way except as specifically set out in the agreement. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Texas Instruments.

0.1 Document History

ID	Author	Date	Status
	Liyi Yu	06.05.2004	Being processed

0.2 References, Abbreviations, Terms

0.3 Table of Contents

0.4 Table of Figures

1 Introduction

G23 is a software package implementing Layers 2 and 3 of the ETSI-defined GSM air interface signalling protocol, and as such represents that part of a GSM mobile station's protocol software which is both, platform and manufacturer independent. Therefore, G23 can be viewed as a building block providing standardised functionality through generic interfaces for easy integration.

The G23 suite of products consists of the following items:

- Layers 2 and 3 for speech & short message services,
- Layers 2 and 3 for fax & data services,
- Application Control Interface,
- Slim MMI [02.30] and
- Test and integration support tools.

2 Overview

2.1 General

This document is a Low Level Design (LLD) document. It describes the implementation and the use cases of the extension mechanism.

The extension mechanism includes two parts: AT command extension mechanism and ACI extension signal mechanism. AT command extension mechanism gives the customers opportunities to extend the standard AT commands with their own AT commands (customer specific AT-commands) or to handle the standard AT commands in their own way. ACI extension signal mechanism gives customers opportunities to send an extension signal and to handle the signal in their own way. Customers can use this mechanism to implement their own code without accessing the TI source code.

The AT command extension module is not empty when shipping the G23 protocol stack software. Simple examples will be implemented and described in this document.

2.2 Feature List

- Customers can extend the standard AT commands with their own AT commands easily;
- Customers can provide their own handling of standard AT commands;
- Customers can provide their own handling of extension signals.

2.3 Architecture

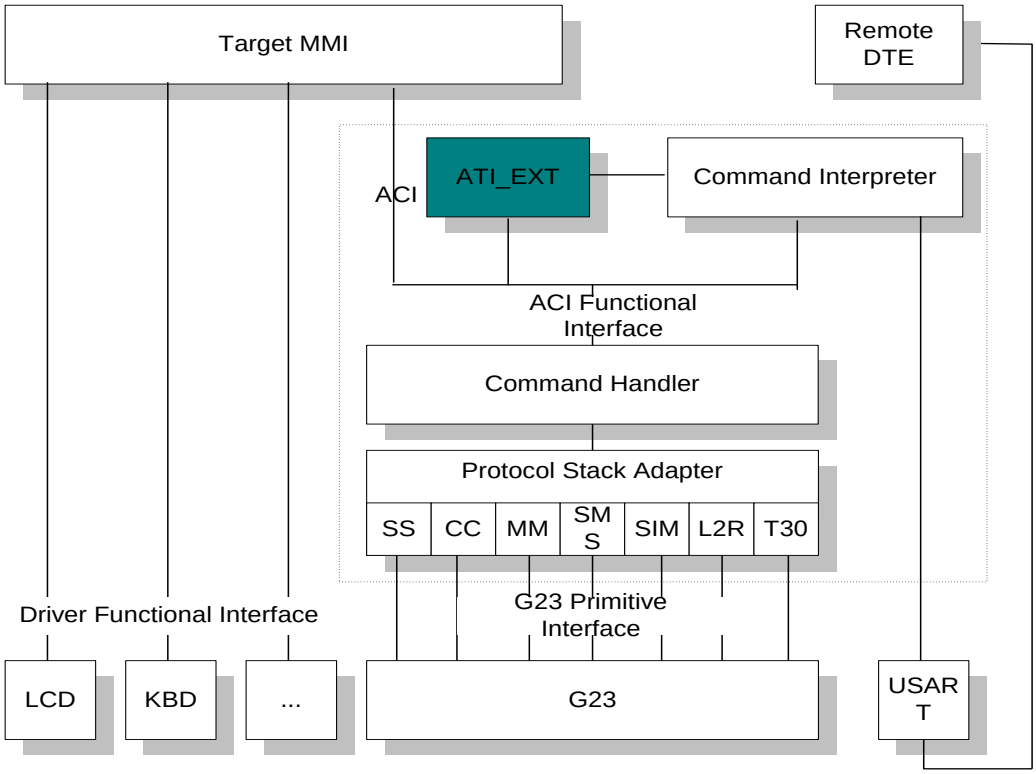


Figure 1 System architecture overview

The picture above is the system architecture of the TI G23 protocol stack software. The extension module (ATI_EXT) is a new module to be implemented for the extension mechanism.

ATI_EXT will run in the same task as ACI. ACI and ATI_EXT have functional interface as shown in the following diagram. Customers can use the ACI functional interface for their implementation.

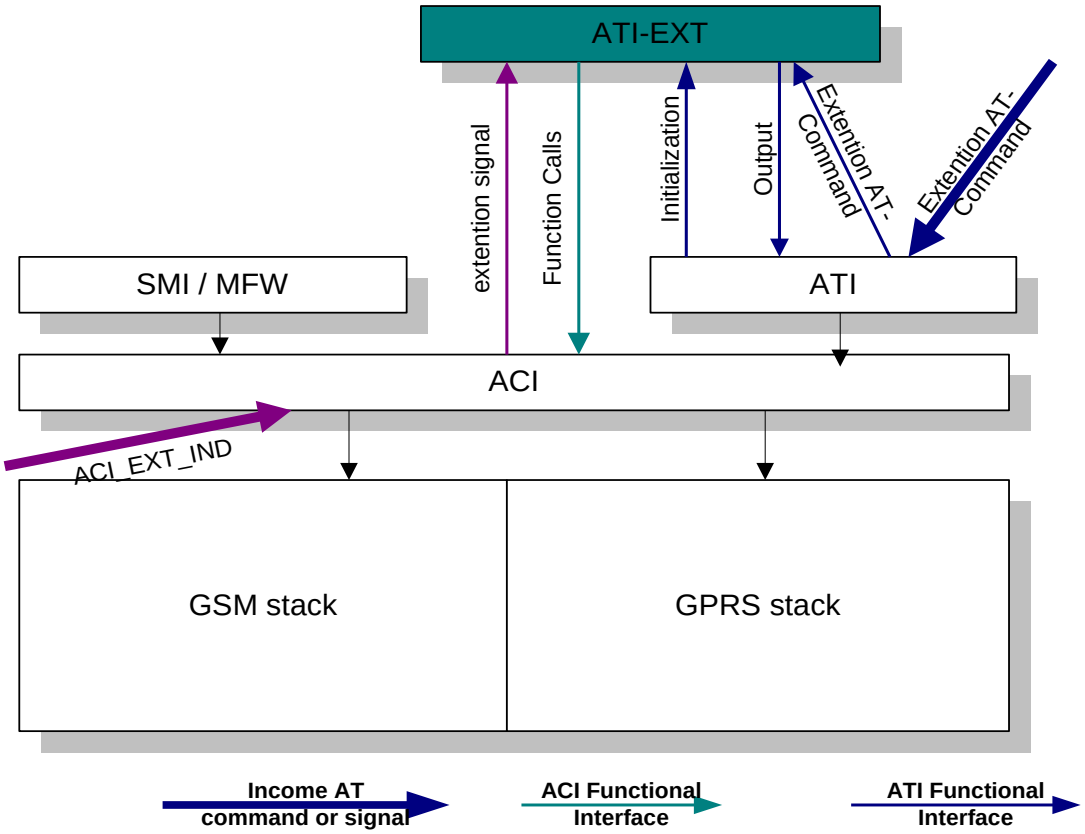


Figure 2 ACI building block

3 Implementation

3.1 Basic Principle

ACI extension mechanism includes two parts: AT command extension mechanism and ACI signal extension mechanism. Customers have access to the module ATI_EXT. They can implement their own handling of extension AT commands in function rEXT_Execute() and implement their own handling of the extension primitive in function rEXT_Signal ().

3.2 AT Command Extension Mechanism

3.2.1 Data types

3.2.1.1 T_ATI_EXT_RETURN

Prototype:

```
typedef enum
{
  ATI_EXT_FAIL     = -1,        /* execution failed, error occurred                    */
  ATI_EXT_CMPL,                 /* execution completed                                 */
  ATI_EXT_EXCT,                 /* execution is in progress                            */
  ATI_EXT_BUSY                  /* execution is rejected due to a busy extension mechanism*/
} T_ATI_EXT_RETURN;
```

Description:

It is used to define the return value of the ATI_EXT functions. Functions in the EXT module should always have this return type.

3.2.1.2 T_ATI_EXT_FORMAT

Prototype:

```
typedef enum
{
  ATI_EXT_PART_UNKNOWN = -1,
  ATI_EXT_PART_BEGIN,           /* beginning part of a complete line          */
  ATI_EXT_PART_LINE,            /* a part from a complete line          */
  ATI_EXT_PART_LAST,            /* the last part from a complete line   */
  ATI_EXT_CMPL_LINE             /* a complete line                      */
} T_ATI_EXT_FORMAT;
```

Description:

It is used to define the format of the output string. Depending on the different output format, ATI outputs the string differently. This format indication is used by the sEXT_output to define which format the output should be.

3.2.2 ATI Interface

3.2.2.1 sEXT_Output () – send output string

Prototype:

GLOBAL T_ATI_RSLT sEXT_Output (UBYTE src_id,
T_ATI_EXT_FORMAT output_format,
*CHAR *output)*

Parameter:

src_id	source ID of the AT command which has sent the output
output_format	format of the output (see T_ATI_EXT_FORMAT)
output	output string

Return:

ATI_FAIL	invalid output format or invalid source ID
ATI_CMPL	the output was successfully sent

Description:

This function can be called from ATI_EXT to send a string output to the AT interpreter. By using the different output format, strings can be outputted differently. E.g. the user wants to output a string in parts to compose a big string he can output the parts saperately with output formats *ATI_EXT_PART_LINE* and then *ATI_EXT_PART_LAST*. Please note that the max length to output is: for GSM is (400-4) bytes and for GPRS is (600-4) bytes.

3.2.2.2 sEXT_Indication() - send unsolicited output

Prototype:

T_ATI_RSLT sEXT_Indication (UBYTE src_id,
*CHAR *indication_string);*

Parameter:

src_id	source ID of the source for the indication 0 means, send indication to all available sources
indication_string	indication string ('C'-Format)

Return:

ATI_FAIL	no extended AT command is running Invalid source ID No indication string
ATI_CMPL	Indication was successfully sent

Description:

This function is used to send an indication unsolicited output.

3.2.2.3 sEXT_Confirm () – execution command is finished

Prototype:

T_ATI_RSLT sEXT_Confirm (UBYTE src_id);

Parameter:

src_id	source ID of the AT command which is finished

Return:

ATI_FAIL	no extension AT command is running Invalid source ID
ATI_CMPL	confirmation was successfully signalled

Description:

If the execution is finished, with this function the successful result "OK" will be signalled. This function should be called to signal the final positive result if there is no final result returned (e.g. only ATI_EXT_EXCT has been returned).

3.2.2.4 sEXT_Error () - error occurred during the command execution

Prototype:

T_ATI_RSLT sEXT_Error (UBYTE src_id, T_ACI_CME_ERR err);

Parameter:

src_id	source ID of the AT command which is finished
err	error code

Return:

ATI_FAIL	no extension AT command is running Invalid source ID
ATI_CMPL	Error message was successfully signalled

Description:

If error happens during the execution of the command, with this function the error information will be outputted. This function should be called to signal the final negative result if there is no final result returned (e.g. only ATI_EXT_EXCT has been returned).

3.2.2.5 sEXT_Init () – initialize the execution command list

Prototype:

*T_ATI_RSLT sEXT_Init (CHAR *cmd_list)*

Parameter:

Cmd_list	The extension command list passed in from the extension module

Return:

ATI_CMPL	initialization was successfully done

Description:

This is the initialization function interface for the extension module in ATI.

3.2.3 ATI_EXT Interface

3.2.3.1 rEXT_Init () - Initialize extension AT command list

Prototype:

T_ATI_EXT_RETURN rEXT_Init ()

Return:

ATI_EXT_FAIL	initialization can not be performed
ATI_EXT_CMPL	command list is successfully initialized

Parameter:

command_list	list of AT command strings

Description:

This function initialize the extension AT command list in ATI.

3.2.3.2 rEXT_Execute () – execute an AT command

Prototype:

T_ATI_EXT_RETURN rEXT_Execute (UBYTE src_id,
*UBYTE *cmd_string,);*

Parameter:

src_id	ID of the source wich has the command received
cmd_string	string with command and parameter ('C'-Format)

Return:

ATI_EXT_FAIL	invalid command string Error in execution
ATI_EXT_BUSY	another ATI extension command is running
ATI_EXT_CMPL	execution is finished
ATI_EXT_EXCT	command runs

Description:

This function starts the execution of a command. The ATI_EXT_FAIL return value indicates only a general error; the extension mechanism is responsible for the specific error indication.

3.2.3.3 rEXT_Abort () – Abort a command execution

Prototype:

T_ATI_EXT_RETURN rEXT_Abort (UBYTE src_id);

Parameter:

src_id	ID of the source which has initiated the abort

Return:

T_ATI_EXT_FAIL	no command is running
T_ATI_EXT_BUSY	abort already running
T_ATI_EXT_CMPL	success

Description:

The function stops the execution of an ATI extension command.

3.3 ACI Signal Extension Mechanism

3.3.1 Data types

3.3.1.1 T_ACI_EXT_IND

Prototype:

```
typedef struct
{
  USHORT signal_id;          /* ID to sign the signal    */
  ULONG  *data;              /* data for the signal      */
} T_ACI_EXT_IND;
```

Description:

The parameter <data> is used to transmit data with the signal. <signal_id> can be used as an internal ID. The ACI does not use this. The data buffer has length of 200 bytes.

3.3.2 Primitive Interface

3.3.2.1 aci_aci_ext_ind () – Interface for ACI_EXT_IND

Prototype:

*void aci_aci_ext_ind (T_ACI_EXT_IND *aci_ext_ind)*

Parameter:

aci_ext_ind	SDU Data element

Return: Void

Description:

This function is the primitive interface for the extended primitive ACI_EXT_IND. This function passes the received data transparently to function rEXT_Signal() in ATI_EXT.

3.3.3 ATI_EXT Interface

3.3.3.1 rEXT_Signal () – extension mechanism signal received

Prototype:

*T_ATI_EXT_RETURN rEXT_Signal (T_ACI_EXT_SIGNAL *ext_signal);*

Return:

T_ATI_EXT_FAIL	signal not supported
T_ATI_EXT_CMPL	The processing was OK.

Parameter:

ext_signal	signal data

Description:

This function will be called by ACI when it receives the extension signal ACI_EXT_SIGNAL. Input data in the primitive will be passed to extension mechanism transparently.

4 Use Cases

4.1 Initialization

The extension AT command list will be initialized when ATI is initialized. The functional interface in ATI_EXT for the initialization is rEXT_Init(). The customer can initialize the extension command list in ACI by the input parameter < *command_list* >.

Example: The customer wants to implement his own AT commands AT$A and AT$B and to override the standard AT command ATD, he should implement the function rEXT_Init() in a way that the input parameter is pointing to a hard coded string: "$A;$B;D;". The extension mechanism can recognize and parse all ACII characters in the AT command name.

The following diagram describes the initialization of the ATI_EXT module.

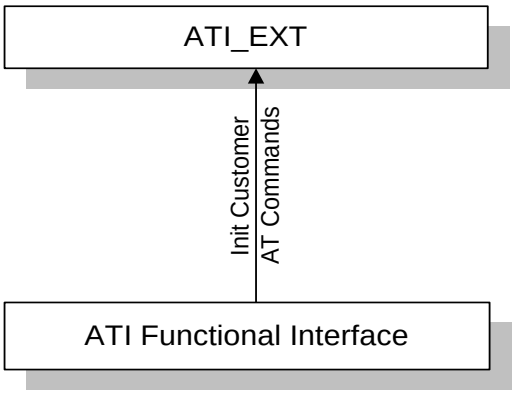


Figure 3 Initialization of the ATI extention mechanism

4.2 Command Execution

When ATI receives the AT command, it first checks the extension command list. If the command is in the extension list ACI calls the extension mechanism. The handling of the extension commands should be implemented in function rEXT_Execute (). When the user implements his own handling, he can make use of the ACI functions. An example of how AT$A, AT$B and ATD is handled is provided. Please note that if the customer specific AT commands are not basic commands (basic commands include: AT+, AT%, AT&, ATA to ATZ), the format of the AT commands passed to the extension mechanism is as following:

Set command: "=" between command name and command parameter. E.g. AT$A=1;
Query command: "?"after the command name. E.g. AT$A? or AT$A?2 if there is a parameter;
Test command: "=?"after the command name. E.g. AT$A=? or AT$A=?2 if there is a parameter.
Other command: nothing after the command name. E.g. AT$A.

The following diagram describes the handling of an extension AT command in the following steps:

1. Extension AT command is received;
2. The AT command string is passed to rEXT_Execute () in the ATI_EXT module;
3. rEXT_Execute () handles the command and uses the ACI and ATI interface if necessary.

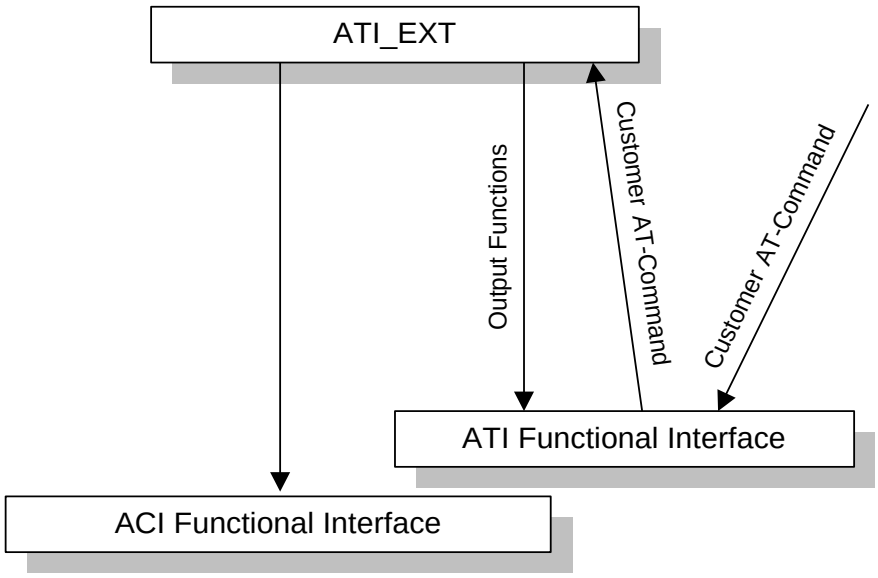


Figure 4 Command execution

4.3 Signal Handling

ACI can receive extension signals from hardware invoked by customer entities. This signal data will be passed to the extension module transparently and will be handled in function rEXT_Signal().

The following diagram describes the handling of an extension signal in the following steps:

1. An extension primitive ACI_EXT_IND is received in the ACI primitive interface;
2. The primitive is passed to rEXT_Signal() transparently;
3. rEXT_Signal() handles the signal.

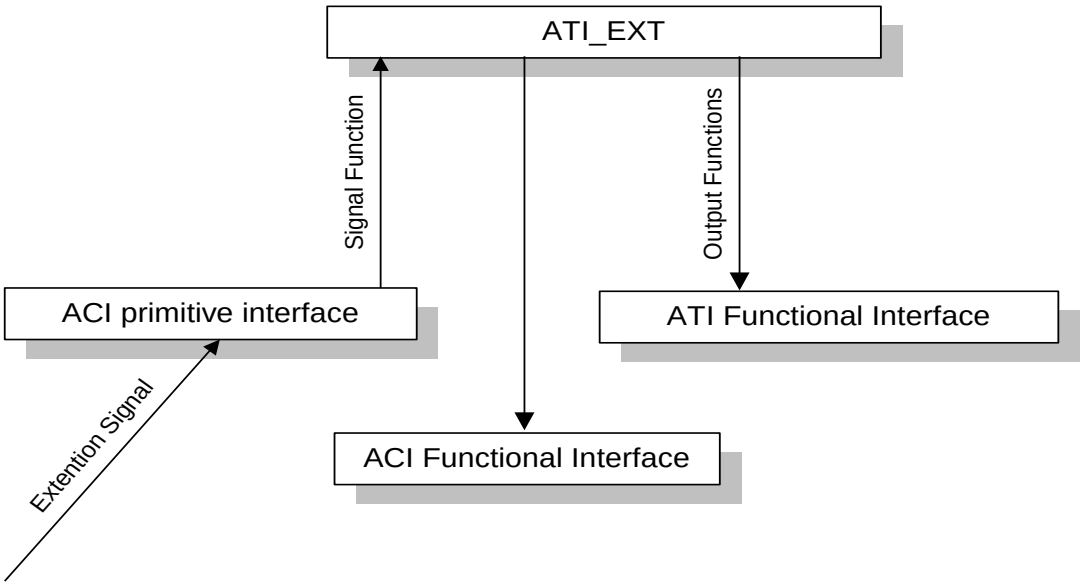


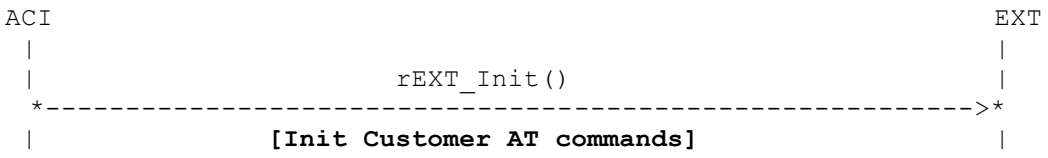
Figure 5 ACI signal extention mechanism

5 Message Sequence Charts

5.1 ATI Extension Mechanism

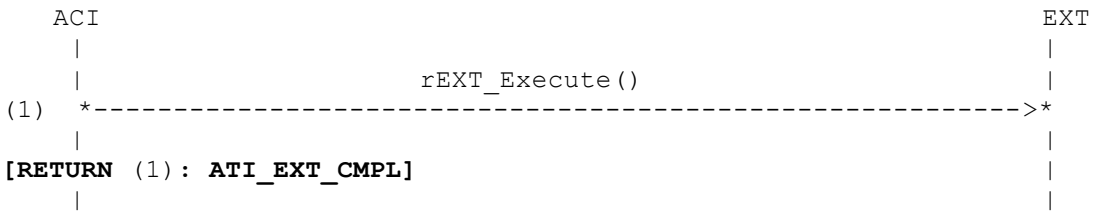
5.1.1 Initialization

ATI initializes the customer specific AT command list when ATI is initialized.

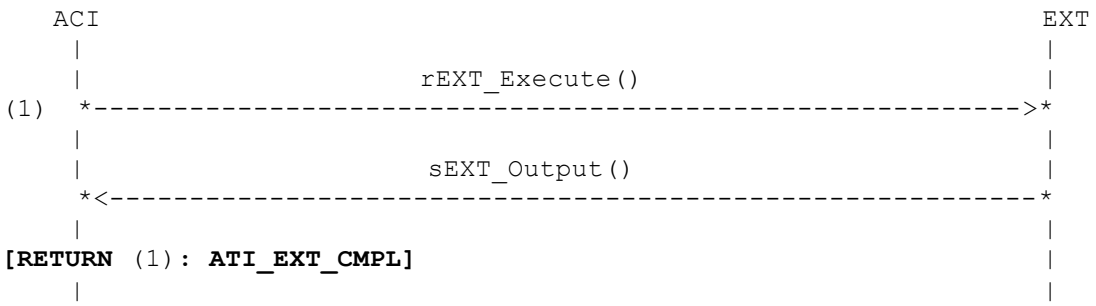


5.1.2 Command Execution

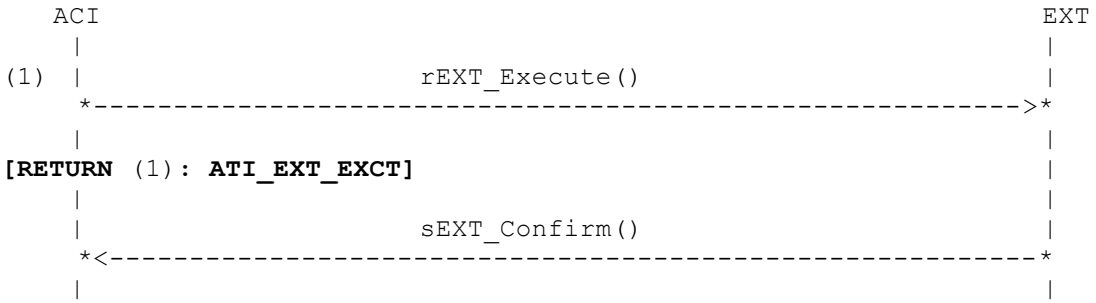
5.1.2.1 Completed without output



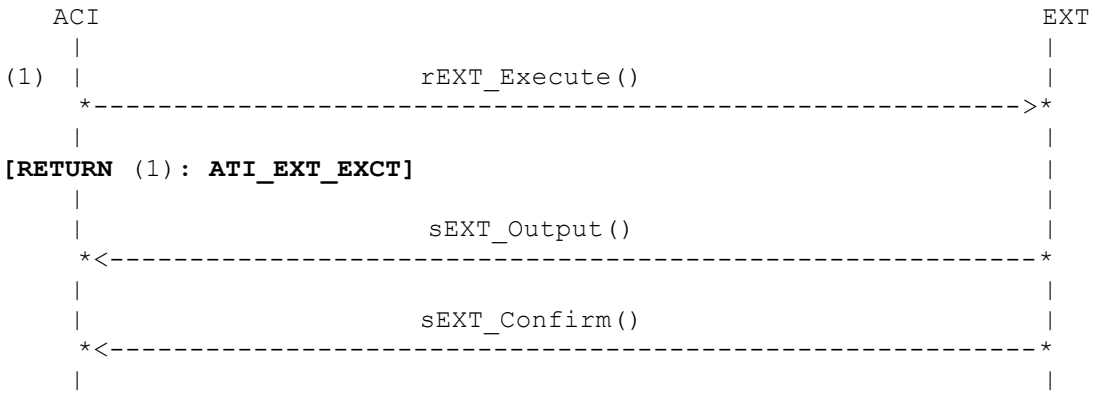
5.1.2.2 Completed with output



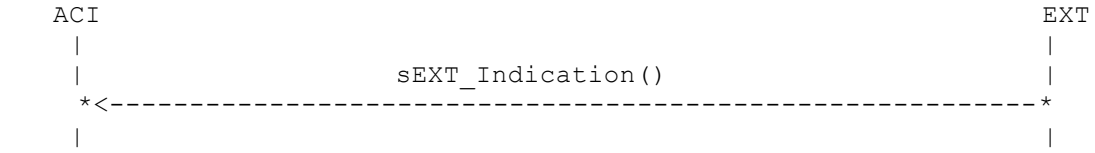
5.1.2.3 Runs longer without output



5.1.3 Runs longer with output

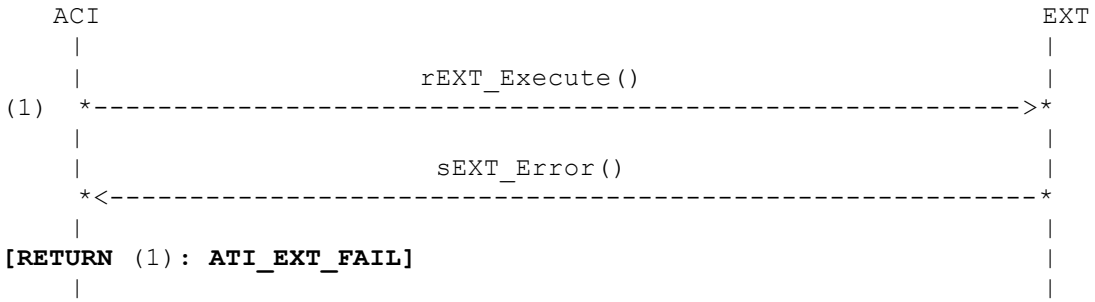


5.1.4 Unsolicited response

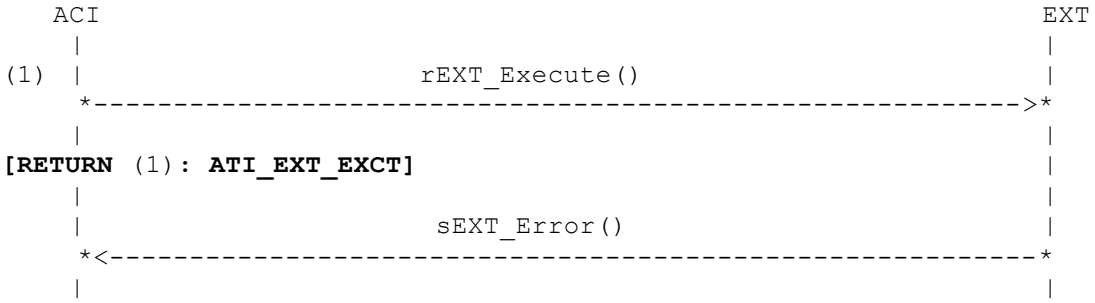


5.1.5 Error cases

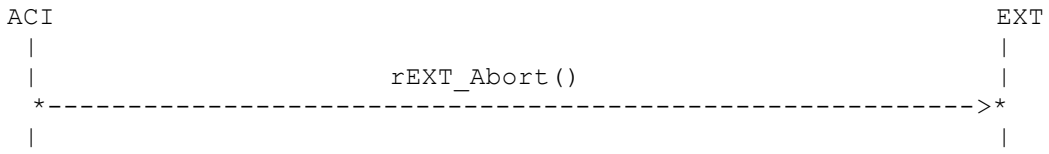
5.1.5.1 Completed



5.1.5.2 Runs longer

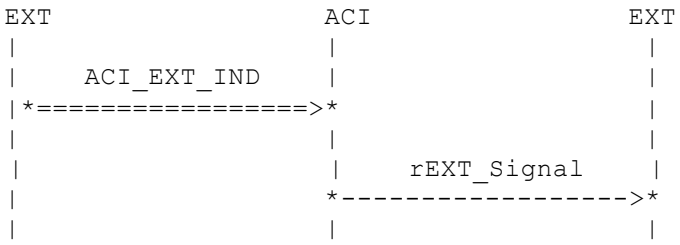


5.1.6 Abort execution



5.2 ACI signal extension mechanism

5.2.1 Receive an Extended Signal



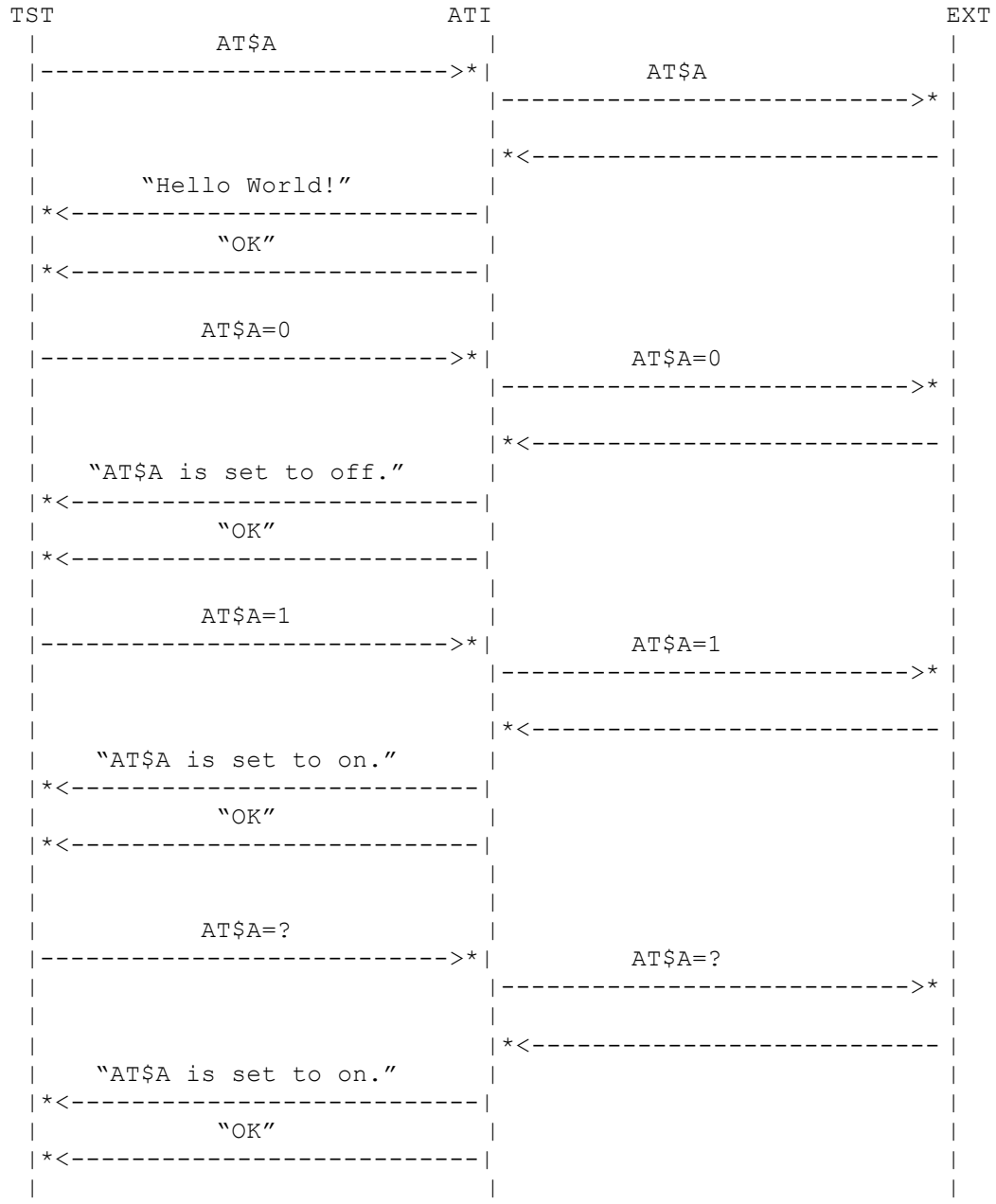
6 Test Plan

6.1 Windows Simulation Tests

The windows simulation test includes the following new test cases:

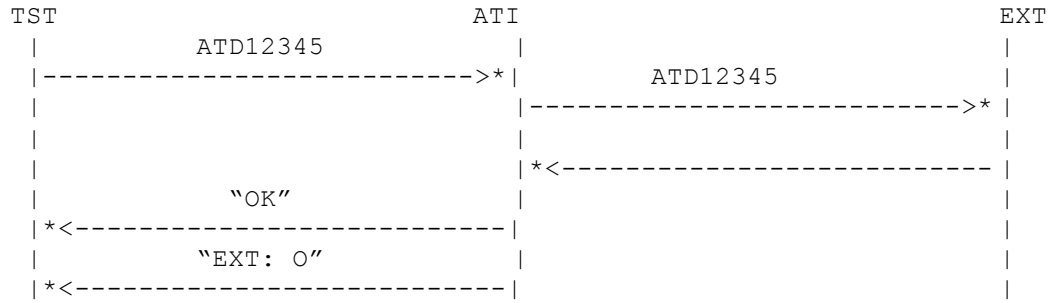
6.1.1 GACI1030 () and GACI1031 ()

This two test cases test the handling of new implemented AT commands AT$A and AT$B.



6.1.2 GACI1032 ()

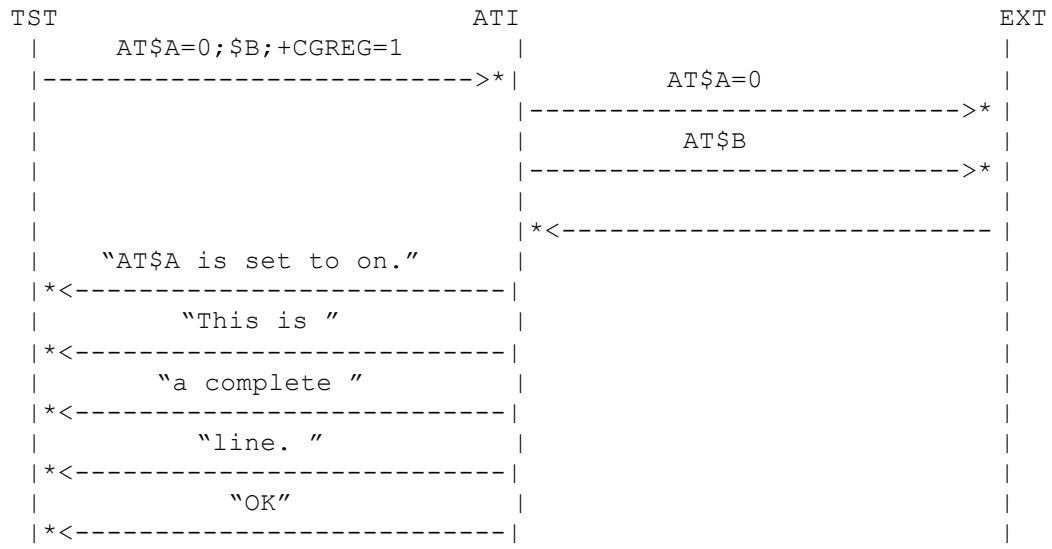
This test case tests the handling of an overriding test case ATD.



6.1.3 GACI1033 ()

This test case tests the handling of a list if AT commands. The format of the extension command list should be separated by a ";".

e.g. AT$A=0;$B;+CGREG=1



6.1.4 GACI1034 ()

This test case tests the handling of the extension signal.

6.2 Target Test

Target test has been done for the same test cases as windows test except the test for the extension signal.

Trace:

AT$A

Hello World!

OK

AT$A=0

AT$A is set to off.

OK

AT$B

This is a complete line.

OK

AT$A;$B

Hello World!

This is a complete line.

OK

AT$A=0;$A=1;$B

AT$A is set to off.

AT$A is set to on.

This is a complete line.

OK