ADL User Guide for Open AT® v3.03

Revision: 001
Date: December 2005

OPEN AT

wavecom
Make it wireless

Operating Systems | Integrated Development Environments | Plug-Ins | Wireless CPUs | Services

ADL User Guide for Open AT® V3.03

Revision: **001**

Date: **12ve December 2005**

Reference: **WM_ASW_OAT_UGD_00053**

Document History

Index	Date	Versions	
001	November 28th 2005	Created	

Trademarks

®, WAVECOM®, WISMO® Open AT® and certain other trademarks and logos appearing on this document, are filed or registered trademarks of Wavecom S.A. in France or in other countries. All other company and/or product names mentioned may be filed or registered trademarks of their respective owners.

Copyright

Overview

This user guide describes the Application Development Layer (ADL).
The aim of the Application Development Layer is to ease the development of Open AT® embedded application. It applies to revision Open AT® 3.03 and upper until further notice.

Table of Contents

List of figures

1 Introduction

1.1 Important remarks

- It is strongly recommended before reading this document, to read the Open AT® Basic Development Guide and specifically the Introduction (chapter 1) and the Description (chapter 2) for having a better overview of what Open AT® is about.
- The ADL library and the standard embedded Open AT® API layer must not be used in the same application code. As ADL APIs will encapsulate commands and trap responses, applications may enter in error modes if synchronization is no more guaranteed.

1.2 References

I. Open AT® Basic Development Guide for revision 3.03 (ref WM_ASW_OAT_UGD_00052 revision 001).

1.3 Glossary

Application Mandatory API	Mandatory software interfaces to be used by the Embedded Application.
AT commands	Set of standard modem commands.
AT function	Software that processes the AT commands and AT subscriptions.
Embedded API layer	Software developed by Wavecom, containing the Open AT® APIs (Application Mandatory API, AT Command Embedded API, OS API, Standard API, FCM API, IO API, and BUS API).
Embedded Application	User application sources to be compiled and run on a Wavecom product.
Embedded Core software	Software that includes the Embedded Application and the Wavecom library.
Embedded software	User application binary: set of Embedded Application sources + Wavecom library.
External Application	Application external to the Wavecom product that sends AT commands through the serial link.
IDE	Integrated Development Environment
Target	Open AT® compatible product supporting an Embedded Application.

Target Monitoring Tool	Set of utilities used to monitor a Wavecom product.
Receive command pre-parsing	Process for intercepting AT responses.
Send command pre–parsing	Process for intercepting AT commands.
Standard API	Standard set of "C" functions.
Wavecom library	Library delivered by Wavecom to interface Embedded Application sources with Wavecom Core Software functions.
Wavecom Core Software	Set of GSM and open functions supplied to the User.

1.4 Abbreviations

A&D	Application & Data
ADL	Application Development Layer
API	Application Programming Interface
CPU	Central Processing Unit
IR	Infrared
KB	Kilobyte
OS	Operating System
PDU	Protocol Data Unit
RAM	Random-Access Memory
ROM	Read-Only Memory
RTK	Real-Time Kernel
SDK	Software Development Kit
SMA	Small Adapter
SMS	Short Message Services
WAP	Wireless Application Protocol

2 Description

2.1 Software Architecture

The Application Development Layer software library, based on standard embedded Open AT® API layer, is included in the Wavecom library since Open AT® release 2.00 (as defined in section 2.1.1 "Software Organization" of the Basic Development Guide).

The aim of the ADL is to provide a high level interface to the Open AT® software developer. The ADL supplies the mandatory software skeleton for an embedded application, for instance the message parser (see 2.2: "Minimum Embedded Application Code" of Open AT® Basic Development Guide) and some messages states machines for given complex services (SIM service, SMS service...).

Thus, the Open AT® software developer can concentrate on the contents of his application. He or she simply has to write the callback functions associated to each service he or she wants to use.

Therefore the software supplied by Wavecom contains the items listed below:

- ADL software library wmadl.lib,
- A set of header files (.h) defining the ADL API functions,
- Source code samples,

It relies on the following software architecture:

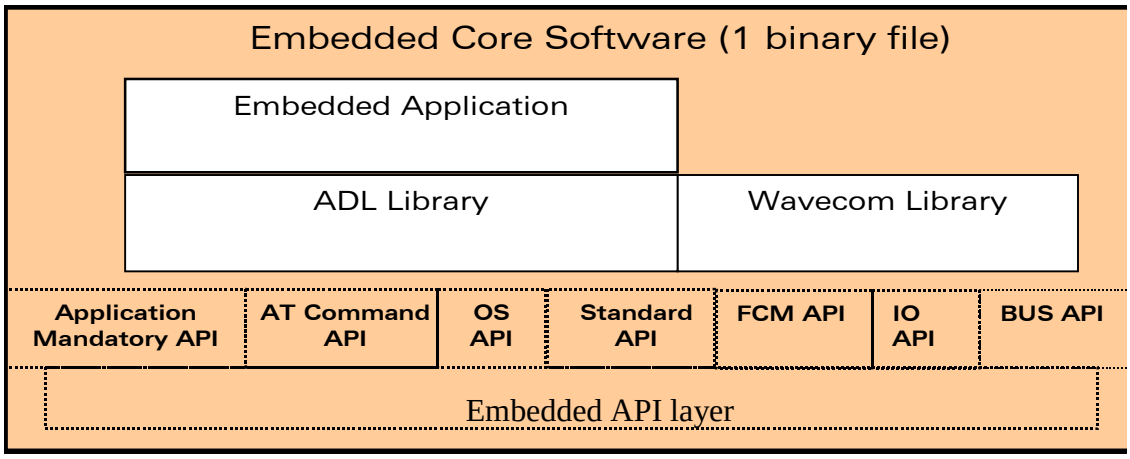


Figure 1: Software architecture

2.2 Minimum Embedded Application Code

The minimum embedded application code requested for ADL is the following:

```
u32 wm_apmCustomStack [ 256 ];
/* The value 256 is an example */
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
```

And the entry point to the ADL code is the main function `adl_main()`:

```
/*main function */
void adl_main(adl_apmInitType_e InitType) {}
```

The **`adl_InitType_e`** is described below:

```
typedef enum
{
      ADL_INIT_POWER_ON,      // Normal power on
      ADL_INIT_REBOOT_FROM_EXCEPTION, // Reboot after an embedded
                              application exception
      ADL_INIT_DOWNLOAD_SUCCESS,    // Reboot after a successful install
                              process (cf. adl_adInstall API)
      ADL_INIT_DOWNLOAD_ERROR// Reboot after an error in install process
                              (cf. adl_adInstall API)
} adl_InitType_e;
```

`wm_apmCustomStack` and **`wm_apmCustomStackSize`** are two mandatory variables, used to define the application call stack size (see §"Minimum Embedded Application Code" and § "Mandatory Functions" of Open AT® Basic Development Guide).

For more information about AT command size, downloading, memory limitation or security, please see § "Description" of Open AT® Basic Development Guide.

2.3 Imported APIs from Open AT® library

The following APIs can be used like in Open AT® standard applications. The required headers are already included in the global ADL header file. The APIs available by this way are listed below:

- Standard API (defined in wm_stdio.h file) ;
- List API (defined in wm_list.h file) ;
- Sound API (defined in wm_snd.h file) ;

Please refer to Open AT® Basic Development Guide for these APIs description.

2.4 ADL limitations

- ADL is not designed to run in ATQ1 mode (quiet mode, meaning that there is no answer to AT commands).
- While an ADL application is running, the ATQ command always replies +CME ERROR:600 ("Not allowed by embedded application).
- Concatenated commands (for example "AT+CREG?;+CGMR") may be used from the embedded application, but not from external applications while ADL is running. If subscribed commands are concatenated, command handlers will not be notified.
- Since ADL uses its own internal process of the +WIND indications, the current value of the AT+WIND command may not be the same when the AT+WOPEN command state is 0 or 1.

2.5 UART 2 and GPIOs shared resources

When the product's second UART is used (started with the AT+WMFM command, or reserved for the GPS component in internal mode on a Q25X1-based product), some of the GPIOs are no more available for the embedded application. The impacted GPIOs depend on the product type, as described hereafter:

WAVECOM module series	Unavailable GPIOs
Q24X6	• GPIO 0 and GPIO 5 • GPO 2 • GPI
Q24X0	• GPIO 0 and GPIO 5 • GPO 2 • GPI
Q25X1	• GPIO 0 and GPIO 5 • GPO 2 • GPI
P32X6	• GPIO 2 • GPI
Q31X6	• GPIO 4 and GPIO 5 • GPO 2 • GPI
P51X6	• GPIO 5 • GPO 0 and GPO 1

2.6 Q2501 product external battery charging mechanism GPIO shared resource

On the Q2501 product, if the external battery charging mechanism is implemented (please refer to the AT+WHCNF command documentation), the GPIO 3 is locked on start-up, and is not available for Open AT® applications.

2.7 Open AT® Memory resources

The available memory resources for the Open AT® applications depend on the product memory size:

- For 16 Mbits flash size products ('A' WISMO module series memory):
  - 256 Kbytes of ROM (application code)
  - 32 Kbytes of RAM
  - 5 Kbytes of Flash Object Data
  - 0 Kbytes of Application & Data Storage Volume

- For 32 Mbits flash size products (B memory):
  - 512 Kbytes of ROM (application code)
  - 128 Kbytes of RAM
  - 128 Kbytes of Flash Object Data
  - 512 Kbytes of Application & Data Storage Volume

2.8 Defined compilation flags

Default compilation flags are defined for all Open AT® projects. These flags are defined below:

__DEBUG_APP__

If this flag is defined (by default), the TRACE & DUMP macros (cf. traces service chapter) will be compiled, and will display debug information on Target Monitoring Tool. Otherwise, these macro will be ignored.

__OAT_API_VERSION__

Numeric flag which contains the current used API version level. For Open AT® V3.03 interface, it is defined as "__OAT_API_VERSION__=303".

__DEBUG_FULL__

If this flag is defined (using the wmmake script with the `-fulldebug` option), the FULL_TRACE & FULL_DUMP macros (cf. traces service chapter) will be compiled, and will display debug information on Target Monitoring Tool. Otherwise, these macros will be ignored.

2.9 Inner AT commands configuration

The ADL library needs for its internal processes to set-up some AT command configurations, that differ from the default values. The concerned commands are listed hereafter:

AT Command	Fixed value
AT+CMEE	1
AT+WIND	All indications (*)
AT+CREG	2
AT+CGREG	2
ATV	1
ATQ	0

(*) *All +WIND unsolicited indications are always required by the ADL library. The "+WIND: 3" indication (product reset) will be enabled only if the external application required it.*

The above fixed values are set-up internally by ADL. This means that all related error codes (for +CMEE) or unsolicited results are always all available to all Open AT® ADL applications, without requiring them to be sent (using the corresponding configuration command).

Important Warning :

User is strongly advised against modifying the current values of these commands from any Open AT® application. Wavecom would not guarantee ADL correct processing if these values are modified by any embedded application.

External applications may modify these AT commands' parameter values without any constraints. These commands and related unsolicited results behavior is the same with our without a running ADL application.

If errors codes or unsolicited results related to these commands are subscribed and then forwarded by an ADL application to an external one, these results will be displayed for the external application only if this one has required them using the corresponding AT commands (same behavior than the Wavecom AT firmware without a running ADL application).

3 API

3.1 AT Commands

3.1.1 Required Header File

The header file for the functions dealing with AT commands is:

adl_at.h

3.1.2 Unsolicited Responses

An unsolicited response is seen as a message received as argument to the ADL `wm_apmAppliParser()` function, with it's the 'MsgTyp' parameter set to WM_AT_UNSOLICITED (see "wm_apmAppliParser Function" in Open AT® Basic Development Guide).

Once you have subscribed to an unsolicited response, you have to unsubscribe to it to stop the callback function being executed every time the ADL parser receives this unsolicited response.

Multiple subscriptions: if you subscribe to an unsolicited response with handler 1 and then you subscribe to the same unsolicited response with handler 2, every time the ADL parser receives this unsolicited response handler 1 and then handler 2 will be executed.

3.1.2.1 The adl_atUnSoSubscribe function

This function subscribes to a specific unsolicited response with an associated callback function: when the unsolicited response we subscribed to is received by the ADL parser the callback function will be executed.

- **Prototype**

```
s16 adl_atUnSoSubscribe(ASCII *UnSostr,
            adl_atUnSoHandler_t UnSohdl)
```

- **Parameters**

UnSostr:
The name (as a string) of the unsolicited response we want to subscribe to. This parameter can also be set as an adl_rspID_e response ID. Please refer to §3.15 for more information.

UnSohdl:
A handler to the callback function associated to the unsolicited response.

The callback function is defined as follow:

```
typedef bool (* adl_atUnSoHandler_t) (adl_atUnsolicited_t *)
```

The argument of the callback function will be a 'adl_atUnsolicited_t' structure, holding the unsolicited response we subscribed to.

The 'adl_atUnsolicited_t' structure defined as follow:

```
typedef struct
{
      adl_strID_e RspID;      // Standard response ID
      u16 StrLength;    /* the length of the string (name) of the
                        unsolicited response*/
      ascii StrData[1]; /* a pointer to the string (name) of the
                        unsolicited response*/
      } adl_atUnsolicited_t;
```

The RspID field is the parsed standard response ID if the received response is a standard one. Refer to §3.15 for more information.

The return value of the callback function is TRUE if the unsolicited string is to be sent to the external application, and FALSE otherwise.
Note that in case of several handlers associated to the same unsolicited response, all of them have to return TRUE for the unsolicited response can be sent to the external application.

- **Returned values**

  OK if no error
  ERROR (-1) if an error occurred.

3.1.2.2 The adl_atUnSoUnSubscribe function

This function unsubscribes to an unsolicited response and its handler.

- **Prototype**

```
s16 adl_atUnSoUnSubscribe(ASCII *UnSostr,
            adl_atUnSoHandler_t UnSohdl)
```

- **Parameters**

  UnSostr:
  The string of the unsolicited response we want to unsubscribe to.

  UnSohdl:
  The callback function associated to the unsolicited response.

- **Returned values**

  OK if the unsolicited response was found,
  ERROR otherwise.

3.1.2.3 Example

```
/* callback function */
bool Wind4_Handler(adl_atUnsolicited_t *paras)
{
    /* Unsubscribe to the '+WIND: 4' unsolicited response */
    adl_atUnSoUnSubscribe("+WIND: 4",
                    (adl_atUnSoHandler_t)Wind4_Handler);
    adl_atSendResponse(ADL_AT_RSP, "\r\nWe have received a Wind 4\r\n");
    /* We want this response to be sent to the external application,
    * so we return TRUE */
    return TRUE;
}

/*main function */
void adl_main(adl_InitType_e adlInitType)
{
    /* Subscribe to the '+WIND: 4' unsolicited response */
    adl_atUnSoSubscribe("+WIND: 4",
                    (adl_atUnSoHandler_t)Wind4_Handler);
}
```

3.1.3 Responses

3.1.3.1 The adl_atSendResponse function

This function sends the provided text to the external application, as a response, an unsolicited response or an intermediate response, according to the requested type.

- **Prototype**
  `void adl_atSendResponse(u8 Type, ascii*String)`

- **Parameters**

  Type:
  - ADL_AT_RSP (response)
  - ADL_AT_UNS (unsolicited response)
  - ADL_AT_INT (intermediate response)

  The ADL_AT_PORT_TYPE Macro may be used with this parameter in order to specify the port where to send the response.
  The specified port has to be previously opened with the AT+WMFM command.
  For example, to send a Response on the UART2 port, the ADL_AT_PORT_TYPE (ADL_AT_UART2, ADL_AT_RSP) syntax has to be used for the Type parameter.
  String:
  The text to be sent.

3.1.3.2 The adl_atSendStdResponse function

This function sends the provided standard response to the external application, as a response, an unsolicited response or an intermediate response, according to the requested type.

- **Prototype**
  `void adl_atSendStdResponse ( u8 Type, adl_strID_e RspID )`

- **Parameters**

  Type:
- ADL_AT_RSP (response)
- ADL_AT_UNS (unsolicited response)
- ADL_AT_INT (intermediate response)

The ADL_AT_PORT_TYPE Macro may be used with this parameter in order to specify the port where to send the response.
The specified port has to be previously opened with the AT+WMFM command.
For example, to send a Response on the UART2 port, the ADL_AT_PORT_TYPE (ADL_AT_UART2, ADL_AT_RSP) syntax has to be used for the Type parameter.

RspID:
Standard response ID to be sent (see §3.15 for more information).

3.1.3.3 The adl_atSendStdResponseExt function

This function sends the provided standard response with an argument to the external application, as a response, an unsolicited response or an intermediate response, according to the requested type.

- **Prototype**
  `void adl_atSendStdResponseExt ( u8 Type, adl_strID_e RspID, u32 arg )`

- **Parameters**

  Type:
- ADL_AT_RSP (response)
- ADL_AT_UNS (unsolicited response)
- ADL_AT_INT (intermediate response)

The ADL_AT_PORT_TYPE Macro may be used with this parameter in order to specify the port where to send the response.
The specified port has to be previously opened with the AT+WMFM command.
For example, to send a Response on the UART2 port, the ADL_AT_PORT_TYPE (ADL_AT_UART2, ADL_AT_RSP) syntax has to be used for the Type parameter.

RspID:
Standard response ID to be sent (see §3.15 for more information).

arg:
Standard response argument. According to response ID, this argument should be an **`u32`** integer, or an **`ascii *`** string.

3.1.4 Incoming AT Commands

A command is a message that is received as an argument by the `wm_apmAppliParser()`function of the ADL with its 'MsgTyp' parameter set to WM_AT_CMD_PRE_PARSER.

Once you have subscribed to a command, you have to unsubscribe to stop the callback function being executed every time this command is sent by the external application.

Multiple subscriptions: if you subscribe to a command with a handler and you subscribe then to the same command with another handler, every time this command is sent by the external application both handlers will be successively executed (in the subscription order).

3.1.4.1 The adl_atCmdSubscribe function

This function subscribes to a specific command with an associated callback function, so that next time the command we subscribed to is sent by the external application, the callback function will be executed.

- **Prototype**

```
s16 adl_atCmdSubscribe(ASCII *Cmdstr,
            adl_atCmdHandler_t Cmdhdl,
            u16 Options)
```

- **Parameters**

  Cmdstr:
  The string (name) of the command we want to subscribe to.

  Cmdhdl:
  The handler of the callback function associated to the command.

  The callback function is defined as follow:

```
typedef void (* adl_atCmdHandler_t) (adl_atCmdPreParser_t *)
```

  The argument of the callback function will be an 'adl_atCmdPreParser_t' structure holding the command we subscribed to.

The 'adl_atCmdPreParser_t' structure is defined as follow:

```
typedef struct
{
      u16         Type;          /* the type of the command (from
                                    ADL_CMD_TYPE_PARA, ADL_CMD_TYPE_TEST,
                                    ADL_CMD_TYPE_READ, ADL_CMD_TYPE_ACT and
                                    ADL_CMD_TYPE_ROOT as defined below) */
      u8          NbPara;        /* the number of valid arguments (different from
                                    "") of the command (if command is from
                                    ADL_CMD_TYPE_PARA type)*/
      adl_at_Port_e     Port;    // Source port on which the command was sent.

      wm_lst_t    ParaList;      /* the parameters list (if command is
                                    from ADL_CMD_TYPE_PARA type). The
                                    ADL_GET_PARAM(_P_,_i_) macro should be used to
                                    get elements of this list (_P_ is the pointer to
                                    the adl_atCmdPreParser_t structure, _i_ is the
                                    requested parameter index (starting from 0)).*/

      u16         StrLength;     /* the length of the command */
      ascii       StrData[1];    /* a pointer to the string of the command*/
} adl_atCmdPreParser_t;
```

Options:
This flag combines with a logical 'OR' the following information:

- Its minimum number of arguments 'a' stored in the least significant byte as in 0x000a
- Its maximum number of arguments 'b' stored in the second least significant byte as in 0x00b0
- Its 'type':

Command type	Value	Meaning
ADL_CMD_TYPE_PARA	0x0100	'AT+cmd=x, y'is allowed. The execution of the callback function also depends on whether the number of argument is valid or not.
ADL_CMD_TYPE_TEST	0x0200	'AT+cmd=?' is allowed.
ADL_CMD_TYPE_READ	0x0400	'AT+cmd?' is allowed.
ADL_CMD_TYPE_ACT	0x0800	'AT+cmd' is allowed.
ADL_CMD_TYPE_ROOT	0x1000	All commands starting with the subscribed string are allowed. The handler will only receive the whole AT string (no parameters detection). For example, if the "at-" string is subscribed, all "at-cmd1", "at-cmd2", etc. strings will be received by the handler.

- **Returned values**

  OK
  ERROR (-1) if an error occurred.

- **Important note about incoming concatenated command**

  ADL is able to recognize and process concatenated commands coming from external application (Please refer to AT Commands Interface Guide for more information on concatenated commands syntax).

  In this case, ADL enters a specific concatenation processing mode, which will end as soon as the last command replies OK, or if one of the used command replies an ERROR code. During this specific mode, any external application connected on this port will receive a "+CME ERROR: 515" code if it tries to send another command.

  If a subscribed command is used in a concatenated command string, the corresponding handler will be notified as if the command was used alone.

  In order to handle properly the concatenation mechanism, each subscribed command has to finally answer with a single terminal response (`ADL_STR_OK, ADL_STR_ERROR` or other ones), otherwise the port will stay in concatenation processing mode, refusing all external commands on this one.

3.1.4.2 The adl_atCmdUnSubscribe function

This function unsubscribes to a command and its handler.

- **Prototype**

```
s16 adl_atCmdUnSubscribe(ascii *Cmdstr,
            adl_atCmdHandler_t Cmdhdl)
```

- **Parameters**

  Cmdstr:
  The string (name) of the command we want to unsubscribe to.

  Cmdhdl:
  The handler of the callback function associated to the command.

- **Returned values**

  OK if the command was found,
  ERROR otherwise.

3.1.4.3 Example

```
/* callback function */
void atabc_Handler(adl_atCmdPreParser_t *paras)
{
    /* Unsubscribe (therefore the command at+abc will only work once) */
    adl_atCmdUnSubscribe("at+abc",
                    (adl_atCmdHandler_t)atabc_Handler);
    if(paras->Type == ADL_CMD_TYPE_READ)
       adl_atSendResponse(ADL_AT_RSP, "\r\nhandling at+abc?\r\n");
   else if(paras->Type == ADL_CMD_TYPE_TEST)
       adl_atSendResponse(ADL_AT_RSP, "\r\nhandling at+abc=?\r\n");
   else if(paras->Type == ADL_CMD_TYPE_ACT)
       adl_atSendResponse(ADL_AT_RSP, "\r\nhandling at+abc\r\n");
   else if(paras->Type == ADL_CMD_TYPE_PARA)
   {
        ascii buffer[25];
        wm_strcpy(buffer, "\r\nhandling at+abc=");
        wm_strcat(buffer, ADL_GET_PARAM(paras, 0));
        wm_strcat(buffer, "\r\n");
        adl_atSendResponse(ADL_AT_RSP, buffer);
    }
    adl_atSendResponse(ADL_AT_RSP, "\r\nOK\r\n");
}

/*main function */
void adl_main(adl_InitType_e adlInitType)
{
 /* Subscribe to the 'at+abc' command in all modes and accepting 1 parameter */
    adl_atCmdSubscribe("at+abc",
                    (adl_atCmdHandler_t)atabc_Handler,
                    ADL_CMD_TYPE_TEST|ADL_CMD_TYPE_READ|
                    ADL_CMD_TYPE_ACT|ADL_CMD_TYPE_PARA|0x0011);
}
```

3.1.5 Sending AT commands

3.1.5.1 The adl_atCmdCreate function

This function sends a command and allows the subscription to several responses and intermediates responses with one associated callback function, so that when any of the responses or intermediates responses we subscribe to will be received by the ADL parser, the callback function will be executed.

- **Prototype**

```
void adl_atCmdCreate(ASCII *Cmdstr,
            bool Rspflag,
            adl_atRspHandler_t Rsphdl,
            [...,]
            NULL)
```

- **Parameters**

  Cmdstr:
  The string (name) of the command we want to send.
  NULL value and "a/" command will be ignored by this function.

  Rspflag: Boolean
  If set to TRUE: the responses and intermediate responses of the command created that are not subscribed will be sent to the external application,
  If set to FALSE they won't be sent to the external application.

  The ADL_AT_PORT_TYPE(_port,_type) macro may be used with this parameter to select a specific UART in order to :
  - forward unsubscribed responses to this specific port ;
  - set-up this specific port serial line settings (as speed with AT+IPR, character framing with AT+ICF, etc...)

  Examples :
  to forward unsubscribed responses to UART1, use
  ```
  ADL_AT_PORT_TYPE ( ADL_AT_UART1, TRUE )
  ```

  to setup UART2 speed (without response forward), use
  ```
  ADL_AT_PORT_TYPE ( ADL_AT_UART2, FALSE )
  ```

  Rsphdl:
  Handler of the callback function associated to all the responses and intermediate responses we are subscribing to.

  The callback function is defined as follow:
  typedef bool (* adl_atRspHandler_t) (adl_atResponse_t *)
  The argument of the callback function will be an 'adl_atResponse_t' structure holding the response we subscribed to.
  The 'adl_atResponse_t' structure is defined as follows:

```
typedef struct
{
      adl_strID_e RspID;      // Standard response ID
      u16   StrLength; // the length of the unsolicited response
      ascii StrData[1];// the string (name) of the unsolicited
                       response
} adl_atResponse_t;
```

  The RspID field is the parsed standard response ID if the received response is a standard one. See § 3.15 for more information.
  The return value of the callback function will be TRUE if the response string must be sent to the external application, FALSE otherwise.

  This allows a variable number of arguments, where we expect a list of response and intermediate response to subscribe to.
  Note that the last element of the list must be NULL.

If the list is set to only 2 elements "*"and NULL, when the command will be sent, all the responses and intermediate responses received by the ADL parser will execute the callback function until a terminal response is received by the ADL parser. This can be useful if you don't know what will be the response of a command, so you can't properly subscribe to it.

The elements of this response list can also be set as an adl_rsp_ID_e response ID. Please refer to §3.15 for more information.

- **Note 1**

With this function we can subscribe to intermediate responses as well as responses.

An intermediate response is a message that is received as an argument by the wm_apmAppliParser() function with its 'MsgTyp' field set to WM_AT_INTERMEDIATE.
A response is a message that is received as an argument by the wm_apmAppliParser() function with its 'MsgTyp' field set to WM_AT_RESPONSE.

Note that all the responses and intermediate responses that have been subscribed to when the command has been created will be un-subscribed when the next terminal response is received by the ADL parser.

- **Note 2**

This function can be associated with the adl_CmdSubscribe one for filtering or spying any intermediate response or response of a specific command send by the external application. (See the example below).

- **Note 3**

Commands sent through the adl_atCmdCreate function are directly submitted to the Wavecom Core Software AT interface: they can not be filtered by an adl_atCmdSubscribe mechanism. The adl_atCmdSubscribe function filters only the commands coming from any external application.

- **Note 4**

This function can be used to send "Text Mode" commands (such as "AT+CMGW", etc.); in order to provide the text related to this command, the adl_atCmdSendText function has then to be used as soon as the prompt ('> ') response is received in the response handler.
Any further calls to adl_atCmdCreate will just store the required commands, in order to send those ones as soon as the running "Text Mode" command has ended.

- **Example**

In the following example, we spy the ATD command by sending the AT+CLCC command every time a subscribed intermediate response or response is received by the ADL parser

```
/* atd responses callback function */
s16 ATD_Response_Handler(adl_atResponse_t *paras)
{
    /* None of the response of the 'at+clcc' command is subscribed but
because
    * the 2nd argument is set to TRUE, all will be sent to the external
application */
    adl_atCmdCreate("at+clcc",
                    TRUE,
                    (adl_atRspHandler_t)NULL,
                    NULL);
    Return TRUE;
}

/* atd callback function */
void ATD_Handler(adl_atCmdPreParser_t *paras)
{
    adl_atCmdUnSubscribe("atd",
                    (adl_atCmdHandler_t) ATD_Handler);
    /* We unsubscribe the command so that when we resend the command
    * it won't be received by the ADL parser anymore.*/
    /* We resend the command (for the phone call to be made) and
subscribe to some
    * of its responses. We also set the 2nd argument to TRUE so that the
response not
    * subscribed will be directly sent to the external application */
    adl_atCmdCreate(paras->StrData,
                    TRUE,
                    (adl_atRspHandler_t)ATD_Response_Handler,
                    "+WIND: 5,1",
                    "+WIND: 2",
                    "OK",
                    NULL);
}

/*main function */
void adl_main(adl_InitType_e adlInitType)
{
    /* Subscribe to the 'atd' command.*/
    adl_atCmdSubscribe("atd",
                    (adl_atCmdHandler_t)ATD_Handler,
                    ADL_CMD_TYPE_ACT);
}
```

3.1.5.2 The adl_atCmdSendText function

This function allows to provide a running "Text Mode" command (e.g. "AT+CMGW") with the required text. This function has to be used as soon as the prompt response ("> ") comes in the response handler provided on adl_atCmdCreate function call.

- **Prototype**

```
s8 adl_atCmdSendText ( ascii * Text )
```

- **Parameters**

  Text:
  Text to be provided to the running "Text Mode" command. If the text does not end with a 'Ctrl-Z' character (0x1A code), the function will add it automatically.

- **Returned values**
  - OK on success; the text has been provided to the running "Text Mode" command: the response handler provided on adl_atCmdCreate call will be notified with the command responses.
  - ADL_RET_ERR_PARAM on parameter error (NULL text)
  - ADL_RET_ERR_BAD_STATE if there is no "Text Mode" command currently running.

- **Note**

  It is not possible to send the text in several times. As soon as the adl_atCmdSendText function is used, the provided text will immediately be sent, and the command will be executed (further calls to adl_atCmdSendText will return ADL_RET_ERR_BAD_STATE, until a new "Text Mode" command is sent).
  It is possible to insert new lines (\'r' chracters) in the text body.

3.2 Timers

3.2.1 Required Header Files

The header file for the functions dealing with timers is:
adl_TimerHandler.h

3.2.2 The adl_tmrSubscribe function

This function starts a timer with an associated callback function. The callback function will be executed as soon as the timer expires.
Note :
Since the WAVECOM products time granularity is 18.5 ms, the 100 ms steps are emulated, reaching a value as close as possible to the requested one modulo 18.5. For example, if a 20 * 100ms timer is required, the real time value will be 1998 ms (108 * 18.5ms).

- **Prototype**

```
adl_tmr_t *adl_tmrSubscribe( bool bCyclic,
                             u32 TimerValue,
                             u8 TimerType,
                             adl_tmrHandler_t Timerhdl )
```

- **Parameters**

bCyclic:
This boolean flag indicates whether the timer is cyclic (TRUE) or not (FALSE). The cyclic timer is automatically set up when a cycle is over.

TimerValue:
The number of periods after which the timer expires (TimerType dependant).

TimerType:
Unit of the TimerValue parameter. The allowed values are defined below:

Timer type	Timer unit
ADL_TMR_TYPE_100MS	TimerValue is in 100 ms steps
ADL_TMR_TYPE_TICK	TimerValue is in 18.5 ms tick steps

Timerhdl:
The handler of the callback function associated to the timer.
It is defined following the type below:

```
typedef void (*adl_tmrHandler_t) ( u8 )
```

The argument of the callback function will be the timer ID received by the ADL parser.

- **Returned values**

A pointer to the timer started (that will be later used, for instance for the un-subscription). There can only be 32 timers running at the same time, if you try to get more this function will return a NULL pointer.
Note: The function will return a NULL pointer if the timer value is zero. The timer will not be started.

3.2.3 The adl_tmrUnSubscribe function

This function stops the timer and unsubscribes to it and his handler.
The call to this function is only meaningful to a cyclic timer or a timer that hasn't expired yet.

- **Prototype**

```
s32 adl_tmrUnSubscribe( adl_tmr_t *tim,
                        adl_tmrHandler_t Timerhdl,
                        u8 TimerType )
```

- **Parameters**

tim:
The timer we want to unsubscribe to.

Timerhdl:
The handler of the callback function associated to the timer.
Note: this parameter is only used to verify the coherence of tim parameter. Timerhdl has to be the timer handler used in the subscription procedure.
For example

```
PhoneTaskTimerPtr = adl_tmrSubscribe (TRUE, OneSecond,
                    ADL_TMR_TYPE_100MS, PhoneTaskTimer) ;
......
adl_tmrUnSubscribe (PhoneTaskTimerPtr, PhoneTaskTimer,
                    ADL_TMR_TYPE_100MS) ;
```

TimerType:
Unit of the TimerValue parameter. The allowed values are defined below:

Timer type	Timer unit
ADL_TMR_TYPE_100MS	TimerValue is in 100 ms steps
ADL_TMR_TYPE_TICK	TimerValue is in 18.5 ms tick steps

- **Returned values**
  - ERROR if the timer wasn't found or couldn't be stopped,
  - the remaining time of the timer before it expires (unit according to the TimerValue parameter)
  - ADL_RET_ERR_BAD_HDL if the provided handler is not the timer's one
  - ADL_RET_ERR_BAD_STATE if the handler has already expired.

3.2.4 Example

```
adl_tmr_t *tt;
u16 timeout_period = 5;         // in 100 ms steps;

void Timer_Handler( u8 Id )
{
    /* We don't unsubscribe to the timer because it has 'naturally'
expired */
    adl_atSendResponse(ADL_AT_RSP, "\r\Timer timed out\r\n");}

/*main function */
void adl_main(adl_InitType_e adlInitType)
{
    /* We set up a timer */
    tt = (adl_tmr_t *)adl_tmrSubscribe, (FALSE,
                    timeout_period,
                          ADL_TMR_TYPE_100MS,
                    (adl_tmrHandler_t)Timer_Handler);
}
```

3.3 Memory

3.3.1 Required Header File

The header file for the memory functions is:

`adl_memory.h`

3.3.2 The adl_memGet function

This function allocates the memory for the requested **size** into the client application RAM memory.

- **Prototype**

  `void * adl_memGet ( u16 size )`

- **Parameters**

  size:
  The size of memory requested (in bytes).

- **Returned values**

  A pointer to the memory allocated if any,
  NULL otherwise.

3.3.3 The adl_memRelease function

This function releases the memory allocated to the supplied pointer.

- **Prototype**
  `bool adl_memRelease ( void ptr )`

- **Parameters**
  ptr:
  The pointer holding the memory.

- **Returned values**
  TRUE if the memory was correctly released,
  FALSE otherwise.

3.4 Debug traces

This service allow to display software « trace » strings on the Target Monitoring Tool. The different ways to embed these trace strings in an Open AT® application depends on the selected configuration in the used IDE (or with the `wmmake` command).
For more information about the Target Monitoring Tool, the configurations and the Integrated Development Environments, please refer to the Tools Manual.

3.4.1 Required Header File

The header file for the flash functions is:
`adl_traces.h`

3.4.2 Debug configuration

When the Debug configuration is selected in the used IDE (or with the `wmmake` command), the __DEBUG_APP__ compilation flag is defined, and also the following macros.

- **`TRACE (( u8 TL, ascii * T, … ))`**
  Prints a “trace” in the Target Monitoring Tool.

  `TL` defines the trace level (traces will be displayed on the CUS4 element of the Target Monitoring Tool).
  Trace levels range is from 1 to 32.
  `T` is the trace string, which may use the standard C “`sprintf`” syntax.
  Please note that the maximum displayed string length is 256 bytes. If the string is longer, it will be truncated on display.

  Example :
  ```
  u8 I = 123;
  TRACE (( 1, “Value if I : %d”, I ));
  ```

  At runtime, this will display the following string on the CUS4 level 1 on the Target Monitoring Tool:
  ```
  Value of I: 123
  ```

- **DUMP (u8 TL, u8 * P, u16 L)**
  Displays the content (each byte in hexadecimal format) of the provided buffer in the Target Monitoring Tool.

  TL defines the trace level (traces will be displayed on the CUS4 element of the Target Monitoring Tool).
  Trace levels range is from 1 to 32.
  P is the buffer's address to dump.
  L is the length (in bytes) of the required dump.

  Since a display line maximum length is 255 bytes, if the display length is greater than 80 (each byte is displayed on 3 ascii characters), the dump will be segmented on several lines. Each 80 bytes truncated line will end with the "..." character sequence.

  Example 1 :
  ```
  u8 * Buffer = "\x0\x1\x2\x3\x4\x5\x6\x7\x8\x9";
  DUMP ( 1, Buffer, 10 );
  ```

  At runtime, this will display the following string on the CUS4 level 1 on the Target Monitoring Tool:
  ```
  00 01 02 03 04 05 06 07 08 09
  ```

  Example 2 :
  ```
  u8 Buffer [ 200 ], i;
  for ( i = 0 ; i < 200 ; i++ ) Buffer [ i ] = i;
  DUMP ( 1, Buffer, 200 );
  ```

  At runtime, this will display the following three lines on the CUS4 level 1 on the Target Monitoring Tool:
  ```
  00 01 02 03 04 05 06 07 08 09 0A [bytes from 0B to 4D] 4E 4F...
  50 51 52 53 54 55 56 57 58 59 5A [bytes from 5B to 9D] 9E 9F...
  A0 A1 A2 A3 A4 A5 A6 A7 [bytes from A8 to C4] C5 C6 C7
  ```

In this Debug configuration, the FULL_TRACE and FULL_DUMP macros are ignored (even if these ones are used in the application source code, they will neither be compiled, nor displayed on Target Monitoring Tool at runtime).

3.4.3 Full Debug configuration

When the Full Debug configuration is selected in the used IDE (or with the `wmmake` command), the __DEBUG_APP__ and __DEBUG_FULL__ compilation flags are both defined, and also the following macros.

- **TRACE ((u8 TL, ascii * T, …))**
  Cf. the Debug configuration

- **DUMP (u8 TL, u8 * P, u16 L)**
  Cf. the Debug configuration

- **FULL_TRACE ((u8 TL, ascii * T, …))**
  Works exactly as the TRACE macro.

- **FULL_DUMP (u8 TL, u8 * P, u16 L)**
  Works exactly as the DUMP macro.

3.4.4 Release configuration

When the Release configuration is selected in the used IDE (or with the `wmmake` command), neither the __DEBUG_APP__ nor __DEBUG_FULL__ compilation flags are defined.
In this configuration, the TRACE, DUMP, FULL_TRACE and FULL_DUMP macros are ignored (even if these ones are used in the application source code, they will neither be compiled, nor displayed on Target Monitoring Tool at runtime).

3.5 Flash

3.5.1 Required Header File

The header file for the flash functions is:

```
adl_flash.h
```

3.5.2 Flash Objects Management

An ADL application may subscribe to a set of objects identified by an handle, used by all ADL flash functions.
This handle is chosen and given by the application at subscription time.
To access to a particular object, the application gives the handle and the ID of the object to access.
At first subscription, the Handle and the associated set of IDs are saved in flash.
The number of flash object IDs associated to a given handle may be only changed after have erased the flash objects (with the AT+WOPEN=3 command).
For a particular handle, the flash objects ID take any value, from 0 to the ID range upper limit provided on subscription.

Important note: due to the internal storage implementation, only up to 2000 object identifiers can exist at the same time.

3.5.2.1 Flash objects write/erase inner process overview

Written flash objects are queued in the flash object storage place. Each time the adl_flhWrite function is called, the process below is done :

- If the object already exists, it is now considered as "erased" (ie. "adl_flhWrite(X);" <=> "adl_flhDelete(X); adl_flhWrite(X);")
- The flash object driver checks if there is enough place the store the new object. If not, a Garbage Collector process is done (see below).
- The new object is created.

About the erase process, each time the adl_flhDelete (or adl_flhWrite) function is called on a ID, this object is from this time "considered as erased", even if it is not physicaly erased (an inner "erase flag" is set on this object).
Objects are physically erased only when the Garbage Collector process is done, when an adl_flhWrite function call needs a size bigger than the available place in the flash objects storage place. The Garbage Collector process erases the flash objects storage place, and re-write only the objects which have not their "erase flag" set.
Please note that the flash memory physical limitation is the erasure cycle number, which is granted to be at least 100.000 times.

3.5.2.2 Flash objects in Remote Task Environment

When an application is running in Remote Task Environment, the flash object storage place is emulated on the PC side : objects are read/written from/to files on the PC hard disk, and not from/to the module's flash memory. The two storage places (module and PC one) may be synchronized using the RTE Monitor interface (cf. the Tools Manual for more information).

3.5.3 The adl_flhSubscribe function

This function subscribes to a set of objects identified by the given Handle.

- **Prototype**

- `s8 adl_flhSubscribe ( ascii* Handle, u16 NbObjectsRes )`**Parameters**

  Handle:
  The Handle of the set of objects to subscribe to.

  NbObjectRes :
  The number of objects related to the given handle. It means that the IDs available for this handle are in the range [0 , (NbObjectRes – 1)].

- **Returned values**
  - OK on success (first allocation for this handle)
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if space is already created for this handle,
  - ADL_FLH_RET_ERR_NO_ENOUGH_IDS if there are no more enough object IDs to allocate the handle.

Notes:

- Only one subscription is necessary. It is not necessary to subscribe to the same handle at each application start.
- It is not possible to unsubscribe from an handle. To release the handle and the associated objects, the user must do an AT+WOPEN=3 to erase the flash objects of the Open AT® Embedded Application.

3.5.4 The adl_flhExist function

This function checks if a flash object exists from the given Handle at the given ID in the flash memory allocated to the ADL developer.

- **Prototype**
  `s32 adl_flhExist (ascii* Handle, u16 ID )`
- **Parameters**

  Handle:
  The Handle of the subscribe set of objects.

  ID:
  The ID of the flash object to investigate (in the range allocated to the provided Handle).
- **Returned values**
  - the requested Flash object length on success
  - 0 if the object does not exist.
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_FLH_RET_ERR_ID_OUT_OF_RANGE if ID is out of handle range

3.5.5 The adl_flhErase function

This function erases the flash object from the given Handle at the given ID.

- **Prototype**
  `s8 adl_flhErase (ascii* Handle, u16 ID )`
- **Parameters**

  Handle:
  The Handle of the subscribed set of objects.

  ID:
  The ID of the flash object to be erased.
  Important note: If ID is set to ADL_FLH_ALL_IDS, all flash objects related to the provided handle will be erased.
- **Returned values**
  - OK on success
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_FLH_RET_ERR_ID_OUT_OF_RANGE if ID is out of handle range

- ADL_FLH_RET_ERR_OBJ_NOT_EXIST if the object does not exist
- ADL_RET_ERR_FATAL if a fatal error occurred (ADL_ERR_FLH_DELETE error event will then be generated)

3.5.6 The adl_fhWrite function

This function writes the flash object from the given Handle at the given ID, for the length provided with the string provided. A single flash object can use up to 30 Kbytes of memory.

- **Prototype**

```
s8 adl_flhWrite (ascii* Handle, u16 ID, u16 Len, u8 *WriteData )
```

- **Parameters**

  Handle:
  The Handle of the subscribed set of objects.

  ID:
  The ID of the flash object to write.
  Len:
  The length of the flash object to write.
  WriteData:
  The provided string to write in the flash object.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM if one at least of the parameters has a bad value.
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_FLH_RET_ERR_ID_OUT_OF_RANGE if ID is out of handle range
  - ADL_RET_ERR_FATAL if a fatal error occurred (ADL_ERR_FLH_WRITE error event will then occur).
  - ADL_FLH_RET_ERR_MEM_FULL if flash memory is full.
  - ADL_FLH_RET_ERR_NO_ENOUGH_IDS if the object can not be created due to the global ID number limitation.

3.5.7 The adl_flhRead function

This function reads the flash object from the given Handle at the given ID, for the length provided and stores it in a string.

- **Prototype**
  `s8 adl_flhRead (ascii* Handle, u16 ID, u16 Len, u8 *ReadData )`

- **Parameters**
  Handle:
  The Handle of the subscribed set of objects

  ID:
  The ID of the flash object to read.

  Len:
  The length of the flash object to read.

  ReadData:
  The string allocated to store the read flash object.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM if one at least of the parameters has a bad value.
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_FLH_RET_ERR_ID_OUT_OF_RANGE if ID is out of handle range
  - ADL_FLH_RET_ERR_OBJ_NOT_EXIST if the object does not exist.
  - ADL_RET_ERR_FATAL if a fatal error occurred (ADL_ERR_FLH_READ error event will then occur).

3.5.8 The adl_flhGetFreeMem function

This function gets the current remaining flash memory size.

- **Prototype**
  `u32 adl_flhGetFreeMem ( void )`

- **Returned values**
  Current free flash memory size in bytes.

3.5.9 The adl_flhGetIDCount function

This function returns the ID count for the provided handle, or the total remaining ID count.

- **Prototype**
  `s32 adl_flhGetIDCount (ascii* Handle)`

- **Parameters**
  Handle:
  The Handle of the subscribed set of objects. If set to NULL, the total remaining ID count will be returned.

- **Returned values**
  - ID count on success: allocated on the provided handle if any, or the total remaining ID count if the handle is set to NULL.
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed

3.5.10 The adl_flhGetUsedSize function

This function returns the used size by the provided ID range from the provided handle. The handle should also be set to NULL to get the whole used size.

- **Prototype**
  `s32 adl_flhGetUsedSize (ascii* Handle, u16 StartID, u16 EndID)`

- **Parameters**
  Handle:
  The Handle of the subscribed set of objects. If set to NULL, the whole flash memory used size will be returned.

  StartID:
  First ID of the range from which to get the used size ; has to be lower than EndID.

  EndID:
  Last ID of the range from which to get the used size ; has to be greater than StartID. To get the used size by all an handle IDs, the [0 , ADL_FLH_ALL_IDS] range may be used

- **Returned values**
  - Used size on success: from the provided Handle if any, otherwise the whole flash memory used size
  - ADL_RET_ERR_PARAM on parameter error
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_FLH_RET_ERR_ID_OUT_OF_RANGE if ID is out of handle range

3.6 FCM Service

ADL provides a FCM service to handle all FCM events.

An ADL application may subscribe to a specific flow (V24 UART 1, UART 2, USB, GSM DATA or GPRS) to exchange data on it. Once a flow is subscribed, the application gets a handle, which must be used in all further FCM operations.

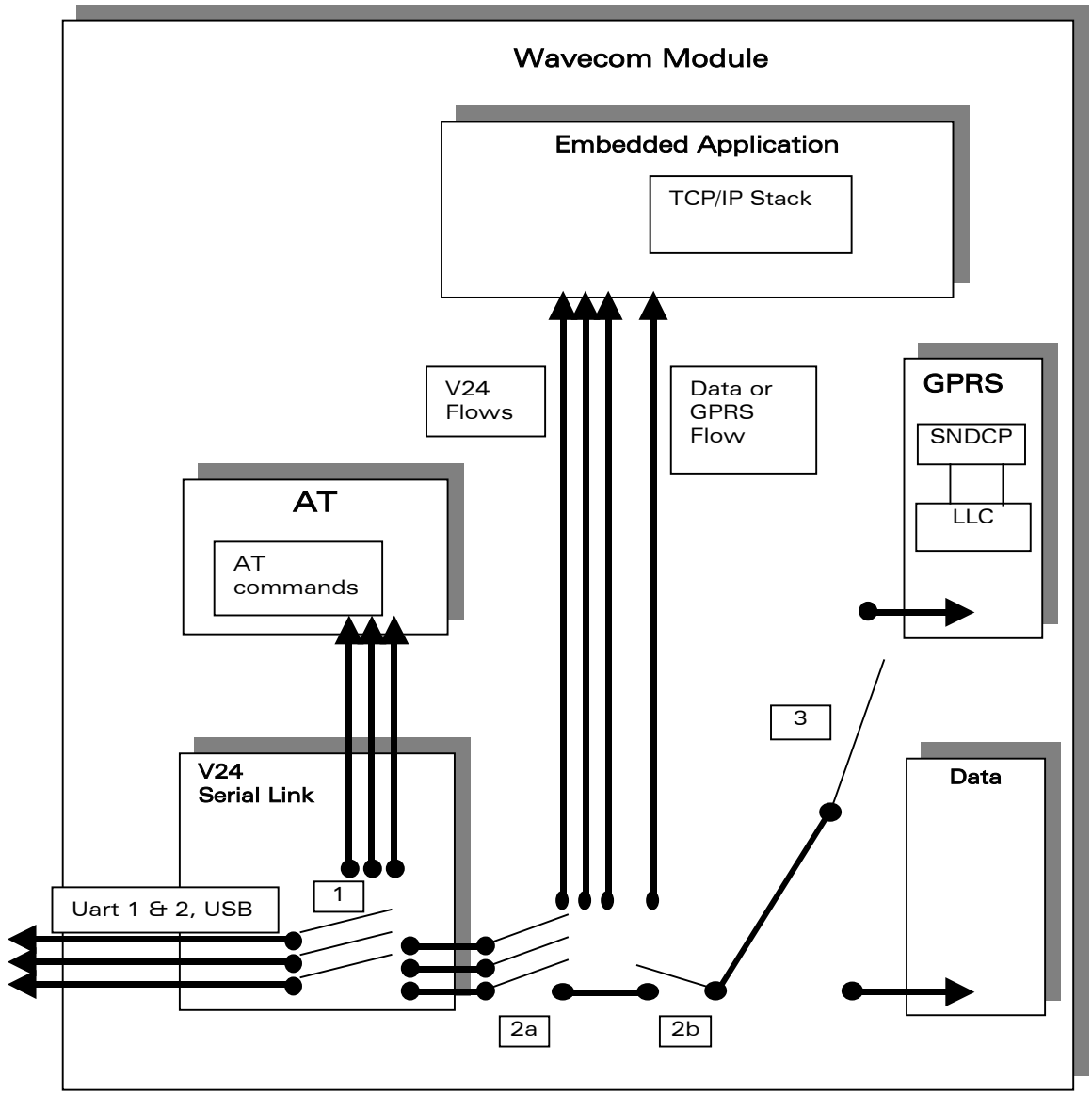


Figure 2: Flow Control Management representation

By default, the flows are closed (in the figure above, all the "2a" and "2b" switches are closed to transmit all data directly between the V24 serial links and Data or GPRS communication).
The "2a" and "2b" switches are processed by the adl_fcmSubscribe and adl_fcmUnsubscribe functions.

The "3" switch means that either the GSM data or the GPRS flow may be subscribed at one time, but not both together.
The "1" switch is processed by the adl_fcmSwitchV24State function.

Important note
GPRS provides only **packet** mode transmission. This means that the embedded application can only send/receive **IP packets** to/from the GPRS flow.

3.6.1 Required Header File

The header file for the FCM functions is:

```
adl_fcm.h
```

3.6.2 The adl_fcmSubscribe function

This function subscribes to the FCM service, opening the requested flow and setting the control and data handlers. The subscription will be effective only when the control event handler has received the ADL_FCM_EVENT_FLOW_OPENNED event.
Each flow may be subscribed only one time.

Additional subscriptions may be done, using the ADL_FCM_FLOW_SLAVE flag (see below). Slave subscribed handles will be able to send & receive data on/from the flow, but will know some limitations:

- For serial-line flows (UART1, UART2, USB), only the main handle will be able to switch the Serial Link state between AT & Data mode ;
- If the main handle unsubscribe from the flow, all slave handles will also be unsubscribed.

Important note:
For serial-link related flows (ADL_FCM_FLOW_V24_UART1 & 2, ADL_FCM_FLOW_V24_USB), the corresponding UART has to be opened first with the AT+WMFM command (See AT Commands Interface guide for more information), otherwise the subscription will fail.
By default, only the UART1 is opened.

- **Prototype**

```
s8    adl_fcmSubscribe     (  adl_fcmFlow_e        Flow,
                              adl_fcmCtrlHdlr_f    CtrlHandler,
                              adl_fcmDataHdlr_f    DataHandler );
```

- **Parameters**

  Flow:
  The allowed values are:
  ADL_FCM_FLOW_GSM_DATA,
  ADL_FCM_FLOW_GPRS,
  ADL_FCM_FLOW_V24_UART1,
  ADL_FCM_FLOW_V24_UART2,
  ADL_FCM_FLOW_V24_USB

  To perform a slave subscription (see above), a bit-wise or has to be done with the flow ID and the ADL_FCM_FLOW_SLAVE flag ; for example:

```
adl_fcmSubscribe (  ADL_FCM_FLOW_V24_UART1 | ADL_FCM_FLOW_SLAVE,
                    MyCtrlHandler, MyDataHandler );
```

  CtrlHandler:
  FCM control events handler, using the following type:
```
typedef bool ( * adl_fcmCtrlHdlr_f ) (adl_fcmEvent_e event );
```

  The FCM control events are defined below (All handlers related to the concerned flow (master and slaves) will be notified together with this events):
  - ADL_FCM_EVENT_FLOW_OPENNED (related to adl_fcmSubscribe),
  - ADL_FCM_EVENT_FLOW_CLOSED (related to adl_fcmUnsubscribe),
  - ADL_FCM_EVENT_V24_DATA_MODE (related to adl_fcmSwitchV24State),
  - ADL_FCM_EVENT_V24_DATA_MODE_EXT (see note below),
  - ADL_FCM_EVENT_V24_AT_MODE (related to adl_fcmSwitchV24State),
  - ADL_FCM_EVENT_V24_AT_MODE_EXT (see note below),
  - ADL_FCM_EVENT_RESUME (related to adl_fcmSendData and adl_fcmSendDataExt),
  - ADL_FCM_EVENT_MEM_RELEASE (related to adl_fcmSendData and adl_fcmSendDataExt) ,

  This handler return value is not relevant, except for ADL_FCM_EVENT_V24_AT_MODE_EXT.

  DataHandler:
  FCM data events handler, using the following type:
```
typedef bool ( * adl_fcmDataHdlr_f ) ( u16 DataLen, u8 * Data );
```

  This handler receives data blocks from the associated flow.
  Once the data block is processed, the handler must return TRUE to release the credit, or FALSE if the credit must not be released. In this case, all credits will be released next time the handler will return TRUE.

  On all flows, all data handlers (master and slaves) subscribed are notified with a data event, and the credit will be released only if all handlers return TRUE: each handler should return TRUE as default value.

  If a credit is not released on the data block reception, it will be released the next time the data handler will return TRUE. The adl_fcmReleaseCredits() should also be used to release credits outside of the data handler.

Maximum size of each data packets to be received by the data handlers depends on the flow type :

- On serial link flows (UART1 / UART2 / USB) : 120 bytes ;
- On GSM data flow : 270 bytes ;
- On GPRS flow : 1500 bytes.

On serial link and GSM flows, if data size to be received by the Open AT® application exceeds this maximum packet size, data will be segmented by the Flow Control Manager, which will call several times the Data Handlers with the segmented packets.
On GPRS flow, whole IP packets will always be received by the Open AT® application.

- **Returned values**
  - A positive or null handle on success (which will have to be used in all further FCM operations). The Control handler will also receive a ADL_FCM_EVENT_FLOW_OPENNED event when flow is ready to process,
  - ADL_RET_ERR_PARAM if one parameter has an incorrect value,
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the flow is not available,
  - ADL_RET_ERR_NOT_SUBSCRIBED if a V24 subscription is made when V24 MASTER flow is not subscribed,
  - ADL_FCM_RET_ERROR_GSM_GPRS_ALREADY_OPENNED if a GSM or GPRS subscription is made when the other one is already subscribed.
  - ADL_RET_ERR_BAD_STATE if the required UART flow was not previously opened with the AT+WMFM command.

- **Notes**

  - When « 7 bits » mode is enabled on a v24 serial link, in data mode, payload data is located on the 7 least significant bits (LSB) of every byte.

  - When a serial link is in data mode, if the external application sends the sequence “1s delay ; +++ ; 1s delay”, this serial link is switched to AT mode, and corresponding handler is notified by the ADL_FCM_EVENT_V24_AT_MODE_EXT event. Then the behavior depends on the returned value.
  If it is TRUE, all this flow remaining handlers are also notified with this event. The main handle can not be un-subscribed in this state.
  If it is FALSE, this flow remaining handlers are not notified with this event, and this serial link is switched back immediately to data mode.
  In the first case, after the ADL_FCM_EVENT_V24_AT_MODE_EXT event, the main handle subscriber should switch the serial link to data mode with the adl_fcmSwitchV24State API, or wait for the ADL_FCM_EVENT_V24_DATA_MODE_EXT event. This one will come when the external application sends the “ATO” command: the serial link is switched to data mode, and then all V24 clients are notified.
  - When a GSM data call is released from the remote part, the GSM flow will automatically be unsubscribed (the ADL_FCM_EVENT_FLOW_CLOSED event will be received by all the flow subscribers).
  - When a GPRS session is released, or when a GSM data call is released from the module side (with the adl_callHangUp function), the

corresponding GSM or GPRS flow have to be unsubscribed. These flows will have to be subscribed again before starting up a new GSM data call, or a new GPRS session.

- For serial link flows (UART1/UART2/USB), the serial line parameters (speed, character framing, etc...) must not be modified while the flow is in data state. In order to change these parameters' value, the concerned flow has firstly to be switched back in AT mode with the `adl_fcmSwitchV24State` API. Once the parameters changed, the flow may be switched again to data mode, using the same API.

3.6.3 The adl_fcmUnsubscribe function

This function unsubscribes from a previously subscribed FCM service, closing the previously opened flows. The cancel of subscription will be effective only when the control event handler has received the ADL_FCM_EVENT_FLOW_CLOSED event.

If slave handles were subscribed, as soon as the master one unsubscribes from the flow, all the slave one will also be unsubscribed.

- **Prototype**

```
s8    adl_fcmUnsubscribe   (  u8 Handle );
```

- **Parameters**

  Handle:
  Handle returned by the adl_fcmSubscribe function.

- **Returned values**
  - OK on success. The Control handler will also receive a ADL_FCM_EVENT_FLOW_CLOSED event when flow is ready to process
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is incorrect,
  - ADL_RET_ERR_NOT_SUBSCRIBED if the flow is already unsubscribed,
  - ADL_RET_ERR_BAD_STATE if the serial link is not in AT mode.

3.6.4 The adl_fcmReleaseCredits function

This function releases some credits for requested flow handle.
The slave subscribers should not use this API.

- **Prototype**

```
s8   adl_fcmReleaseCredits (  u8   Handle,
                              u8   NbCredits );
```

- **Parameters**

  Handle:
  Handle returned by the adl_fcmSubscribe function.
  NbCredits:
  Number of credits to release for this flow. If this number is greater than the number of previously received data blocks, all credits are released. If

an application wants to release all received credits at any time, it should call the adl_fcmReleaseCredits API with NbCredits parameter set to 0xFF.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown,
  - ADL_RET_ERR_BAD_HDL if the handle is a slave one.

3.6.5 The adl_fcmSwitchV24State function

This function switches a serial link state to AT mode or to Data mode. The operation will be effective only when the control event handler has received an ADL_FCM_EVENT_V24_XXX_MODE event. Only the main handle subscriber can use this API.

- **Prototype**

```
s8   adl_fcmSwitchV24State   (  u8  Handle,
                                u8  V24State );
```

- **Parameters**

Handle:
Handle returned by the adl_fcmSubscribe function.
V24State:
Serial link state to switch to. Allowed values are defined below:
ADL_FCM_V24_STATE_AT,
ADL_FCM_V24_STATE_DATA

- **Returned values**
  - OK on success. The Control handler will also receive a ADL_FCM_EVENT_V24_XXX_MODE event when the serial link state has changed
  - ADL_RET_ERR_PARAM if one parameter has an incorrect value
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown
  - ADL_RET_ERR_BAD_HDL if the handle is not the main flow one

3.6.6 The adl_fcmSendData function

This function sends a data block on the requested flow.

- **Prototype**

```
s8 adl_fcmSendData          (  u8                  Handle,
                               u8 *                Data,
                               u16                 DataLen );
```

- **Parameters**

Handle:
Handle returned by the adl_fcmSubscribe function.
Data:
Data block buffer to write.

DataLen:
Data block buffer size.
Maximum data packet size depends on the subscribed flow :
- On serial link flows (UART1 / UART2 / USB) : 2000 bytes ;
- On GSM data flow : no limitation (memory allocation size) ;
- On GPRS flow : 1500 bytes.

- **Returned values**
  - OK on success. The Control handler will also receive a ADL_FCM_EVENT_MEM_RELEASE event when the data block memory buffer will be released ;
  - ADL_FCM_RET_OK_WAIT_RESUME on success, but the last credit was used. The Control handler will also receive a ADL_FCM_EVENT_MEM_RELEASE event when the data block memory buffer will be released ;
  - ADL_RET_ERR_PARAM is a parameter has an incorrect value,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown,
  - ADL_RET_ERR_BAD_STATE if the flow is not ready to send data,
  - ADL_FCM_RET_ERR_WAIT_RESUME if the flow has no more credit to use.

  On ADL_FCM_RET_XXX_WAIT_RESUME returned value, the subscriber has to wait for a ADL_FCM_EVENT_RESUME event on Control Handler to continue sending data.

3.6.7 The adl_fcmSendDataExt function

This function sends a data block on the requested flow. This API do not perform any processing on provided data block, which is sent directly on the flow.

- **Prototype**

```
s8 adl_fcmSendDataExt      (  u8                    Handle,
                              adl_fcmDataBlock_t * DataBlock);
```

- **Parameters**

  Handle:
  Handle returned by the adl_fcmSubscribe function.

  DataBlock:
  Data block buffer to write, using the following type:

```
typedef struct
{
      u16 Reserved1[4];
      u16 DataLength;   /* Data length */
      u16 Reserved2[5];
      u8 Data[1]; /* Data to send */
} adl_fcmDataBlock_t;
```

  The block must be dynamically allocated and filled by the application, before sending it to the function. The allocation size has to be **`sizeof ( adl_fcmDataBlock_t ) + DataLength`**, where DataLength is the value to be set in the **`DataLength`** field of the structure.

  Maximum data packet size depends on the subscribed flow :

- On serial link flows (UART1 / UART2 / USB) : 2000 bytes ;
- On GSM data flow : no limitation (memory allocation size) ;
- On GPRS flow : 1500 bytes.

- **Returned values**
  - OK on success. The Control handler will also receive a ADL_FCM_EVENT_MEM_RELEASE event when the data block memory buffer will be released,
  - ADL_FCM_RET_OK_WAIT_RESUME on success, but the last credit was used. The Control handler will also receive a ADL_FCM_EVENT_MEM_RELEASE event when the data block memory buffer will be released ;
  - ADL_RET_ERR_PARAM is a parameter has an incorrect value,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown,
  - ADL_RET_ERR_BAD_STATE if the flow is not ready to send data,
  - ADL_FCM_RET_ERR_WAIT_RESUME if the flow has no more credit to use.

  On ADL_FCM_RET_XXX_WAIT_RESUME returned value, the subscriber has to wait for an ADL_FCM_EVENT_RESUME event on Control Handler to continue sending data.

Important Remark :
The Data block will be released by the adl_fcmSendDataExt() API on OK and ADL_FCM_RET_OK_WAIT_RESUME return values (the memory buffer will be effectively released once the ADL_FCM_EVENT_MEM_RELEASE event will be received in the Control Handler). The application has to use only dynamic allocated buffers (with adl_memGet function).

3.6.8 The adl_fcmGetStatus function

This function gets the buffer status for requested flow handle, in the requested way.

- **Prototype**

```
s8 adl_fcmGetStatus (  u8              Handle,
                       adl_fcmWay_e    Way );
```

- **Parameters**

  Handle:
  Handle returned by the adl_fcmSubscribe function.

  Way:
  As flows have two ways (from Embedded application, and to Embedded application), this parameter specifies the direction (or way) from which the buffer status is requested. The possible values are:

```
typedef enum {
     ADL_FCM_WAY_FROM_EMBEDDED,
     ADL_FCM_WAY_TO_EMBEDDED
} adl_fcmWay_e;
```

- **Returned values**
  - ADL_FCM_RET_BUFFER_EMPTY *if the requested flow and way buffer is empty,*
  - ADL_FCM_RET_BUFFER_NOT_EMPTY *if the requested flow and way buffer is not empty ; the Flow Control Manager is still processing data on this flow,*
  - ADL_RET_ERR_UNKNOWN_HDL *if the provided handle is unknown,*
  - ADL_RET_ERR_PARAM *if the way parameter value in out of range*.

3.7 GPIO Service

ADL provides a GPIO service to handle GPIO operations.

3.7.1 Required Header File

The header file for the GPIO functions is:

`adl_gpio.h`

3.7.2 The adl_ioSubscribe function

This function subscribes to some GPIO and sets up a polling system if required. Note: using the product's second UART locks some GPIOs, which will not be available for allocation by the application ; please refer to §2.5for more information.

- **Prototype**

```
s8      adl_ioSubscribe  (  u32          GpioMask,
                           u32          GpioDir,
                           u32          GpioDefValues,
                           u32          PollingTime,
                           adl_ioHdlr_f    GpioHandler );
```

- **Parameters**

GpioMask:

Mask of GPIOs to subscribe, using the following defined values. One or several GPIOs may be subscribed, by performing a logical OR between the requested identifiers.

*For **Wismo Pac P31X3 and P32X3** products:*

ADL_IO_P32X3_GPI,
ADL_IO_P32X3_GPIO_0,
ADL_IO_P32X3_GPIO_2,
ADL_IO_P32X3_GPIO_3,
ADL_IO_P32X3_GPIO_4,
ADL_IO_P32X3_GPIO_5

*For **Wismo Pac P32X6** product:*

ADL_IO_P32X6_GPI,
ADL_IO_P32X6_GPO_0,
ADL_IO_P32X6_GPIO_0,
ADL_IO_P32X6_GPIO_2,
ADL_IO_P32X6_GPIO_3,
ADL_IO_P32X6_GPIO_4,
ADL_IO_P32X6_GPIO_5,
ADL_IO_P32X6_GPIO_8

*For **Wismo Quik Q23X3 and Q24X3** products:*

ADL_IO_Q24X3_GPI,
ADL_IO_Q24X3_GPO_1,
ADL_IO_Q24X3_GPO_2,
ADL_IO_Q24X3_GPIO_0,
ADL_IO_Q24X3_GPIO_4,
ADL_IO_Q24X3_GPIO_5

*For **Wismo Quik Q24X6** products:*

ADL_IO_Q24X6_GPI,
ADL_IO_Q24X6_GPO_0,
ADL_IO_Q24X6_GPO_1,
ADL_IO_Q24X6_GPO_2,
ADL_IO_Q24X6_GPO_3,
ADL_IO_Q24X6_GPIO_0,
ADL_IO_Q24X6_GPIO_4,
ADL_IO_Q24X6_GPIO_5

*For **Wismo Quik Q2400** products:*

ADL_IO_Q24X0_GPI,
ADL_IO_Q24X0_GPO_0,
ADL_IO_Q24X0_GPO_1,
ADL_IO_Q24X0_GPO_2,
ADL_IO_Q24X0_GPO_3,
ADL_IO_Q24X0_GPIO_0,
ADL_IO_Q24X0_GPIO_4,
ADL_IO_Q24X0_GPIO_5

*For **Wismo Quik Q31X6** product:*

ADL_IO_Q31X6_GPI,
ADL_IO_Q31X6_GPO_1,
ADL_IO_Q31X6_GPO_2,
ADL_IO_Q31X6_GPIO_3,
ADL_IO_Q31X6_GPIO_4,
ADL_IO_Q31X6_GPIO_5,
ADL_IO_Q31X6_GPIO_6,
ADL_IO_Q31X6_GPIO_7

*For **Wismo Pac P5186** product:*

ADL_IO_P51X6_GPO_0
ADL_IO_P51X6_GPO_1,
ADL_IO_P51X6_GPIO_0,
ADL_IO_P51X6_GPIO_4,
ADL_IO_P51X6_GPIO_5,
ADL_IO_P51X6_GPIO_8,
ADL_IO_P51X6_GPIO_9,
ADL_IO_P51X6_GPIO_10,
ADL_IO_P51X6_GPIO_11,
ADL_IO_P51X6_GPIO_12

*For **Wismo Quik Q25X1** product:*

ADL_IO_Q25X1_GPI
ADL_IO_Q25X1_GPO_0
ADL_IO_Q25X1_GPO_1
ADL_IO_Q25X1_GPO_2
ADL_IO_Q25X1_GPO_3
ADL_IO_Q25X1_GPIO_0
ADL_IO_Q25X1_GPIO_1
ADL_IO_Q25X1_GPIO_2
ADL_IO_Q25X1_GPIO_3
ADL_IO_Q25X1_GPIO_4
ADL_IO_Q25X1_GPIO_5

GpioDir:
Mask of GPIO directions to subscribe. For each allocated GPIO, the corresponding bit in the mask should be set to one of the following values:

- 1: input
- 0: output.

The "GpioMask" constants should be used also for this parameter. If this parameter is set to 0, all subscribed GPIOs are allocated as outputs. If it is set to 0xFFFFFFFF, all subscribed GPIOs are allocated as inputs.
Note: this parameter is only relevant for GPIOs ; GPIs are always subscribed as inputs, and GPOs are always subscribed as outputs, whatever is the **GpioDir** corresponding bit value.

GpioDefValues:
Mask of GPIO default values when set as an output. For each subscribed output GPIO, the corresponding bit in the mask is the default value after allocation (0 or 1). The "GpioMask" constants should be used also for this parameter. If this parameter is set to 0, all subscribed output GPIOs are set to 0. If it is set to 0xFFFFFFFF, all subscribed output GPIOs are set to 1.

PollingTime:
If some IO is allocated as input, this parameter represents the time interval between two GPIO polling operations (unit is 100ms) ;
If no polling is requested, this parameter must be 0.

GpioHandler:
Handler receiving the status of the GPIOs specified by the mask. Must be NULL if no polling is requested. The following type is used:

```
typedef void (*adl_ioHdlr_f) ( u8 GpioHandle, u32 GpioState );
```

GpioHandle: handle on which the polling GPIOs are allocated
GpioState: mask of read values on polling GPIOs.

This handler is called every time the "GpioState" value changes (ie. one of the allocated GPIOs has changed).

- **Returned values**
  - A positive or null GPIO handle on success,
  - ADL_RET_ERR_PARAM if a parameter has an incorrect value,
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if a requested GPIO was not free, .
  - ADL_RET_ERR_FATAL if a fatal error occurred (a ADL_ERR_IO_ALLOCATE error event will also be sent)

Note : Some product hardware related features (UART2, external battery charging mechanism on Q2501) may lock some GPIOs, which will not be available for allocation by the application ; please refer to § 2.5 & 2.6 for more information.

3.7.3 The adl_ioUnsubscribe function

This function unsubscribes from a GPIO handle previously allocated.

- **Prototype**

```
s8      adl_ioUnsubscribe   ( u8         Handle );
```

- **Parameters**

Handle:
Handle previously returned by a call to adl_ioSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown
  - ADL_RET_ERR_FATAL if a fatal error occurred (a ADL_ERR_IO_RELEASE error event will also be sent)

3.7.4 The adl_ioRead function

This function reads all GPIOs from a handle previously allocated.

- **Prototype**

```
u32     adl_ioRead          ( u8         Handle );
```

- **Parameters**

Handle:
Handle previously returned by a call to adl_ioSubscribe function.

- **Returned values**

4 bytes mask of the read GPIO states, or
0 if the handle is unknown.

3.7.5 The adl_ioWrite function

This function writes on one or more GPIOs from a handle previously allocated.

- **Prototype**

```
s8      adl_ioWrite         ( u8        Handle,
                              u32       GpioMask,
                              u32       GpioValues );
```

- **Parameters**

  Handle:
  Handle previously returned by a call to adl_ioSubscribe function.

  GpioMask:
  Mask of GPIO to write.

  GpioValues:

  Mask of GPIO values to write.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown
  - ADL_RET_ERR_PARAM if one parameter has an incorrect value
  - ADL_RET_ERR_FATAL if a fatal error occurred (a ADL_ERR_IO_WRITE error event will also be sent)

3.7.6 The adl_io GetProductType function

This function returns the product type.

- **Prototype**

```
adl_ioProductTypes_e adl_ioGetProductType  ( void );
```

- **Returned values**

  This function returns the product type, with the following defined values:

ADL_IO_PRODUCT_TYPE_Q24X3	*(for Q23X3 and Q24X3 products)*
ADL_IO_PRODUCT_TYPE_Q24X6	
ADL_IO_PRODUCT_TYPE_P32X3	*(for P31X3 and P32X3 products)*
ADL_IO_PRODUCT_TYPE_P32X6	
ADL_IO_PRODUCT_TYPE_Q31X6	
ADL_IO_PRODUCT_TYPE_P5186	
ADL_IO_PRODUCT_TYPE_Q24X0	
ADL_IO_PRODUCT_TYPE_Q25X1	

3.8 Bus Service

ADL provides a bus service to handle all SPI, I2C soft and Parallel bus operations.
Note: for bus management operations, the Q25x1 series module behaves as Q2406 modules.

3.8.1 Required Header File

The header file for the bus functions is:

adl_bus.h

3.8.2 The adl_busSubscribe function

This function subscribes to a specific bus type.

- **Prototype**

```
s8      adl_busSubscribe    ( u32       BusAddress,
                              u32       Param );
```

- **Parameters**

  BusAddress:

  Type and address of the bus to subscribe to, using following defined values, by performing a logical OR between **type** and **address**.

	Type possible values	*Address* possible values
SPI bus	ADL_BUS_TYPE_SPI	ADL_BUS_SPI_ADDR_CS_SPI_EN: use SPI_EN pin as Chip Select *(for Q24X6 and Q2400 products, this setting is automatically mapped on GPO 3 used as Chip Select ;* *for P32X6 product, this setting is automatically mapped on GPIO 8 used as Chip Select);* ***Not available for P5186 product**).* ADL_BUS_SPI_ADDR_CS_SPI_AUX: use SPI_AUX pin as Chip Select *(for Q24X6, Q2400 and P32X6 products, this setting is automatically mapped on GPO 0 used as Chip Select ;* ***Not available for P5186 product*** ***Not available for Q31X6 product**).* ADL_BUS_SPI_ADDR_CS_GPIO : a GPIO or GPO is used as Chip Select. The used GPIO index is given by a logical OR with the index defined in IO service *This IO must not be allocated by any application.* ADL_BUS_SPI_ADDR_CS_NONE The Chip Select signal is not handled by the ADL BUS service. The application should subscribe to a GPIO in order to handle itself the SPI Chip Select signal.

	Type possible values	*Address* possible values
IC2 soft bus	ADL_BUS_TYPE_I2C_SOFT	Less Significant Byte of BusAddress parameter is used as 7 bits slave address for devices on I2C bus.
Parallel bus	ADL_BUS_TYPE_PARALLEL	ADL_BUS_PARA_LCDEN_AS_CS: use LCD_EN pin as Chip Select *On P32X6 product, the LCD_EN pin is the same than the GPIO 8 one ; it must not be allocated by any application.* ADL_BUS_PARA_CSUSR_AS_CS: use CS_USER pin as Chip Select (GPIO 5 on Pac products, GPIO 3 on Q31X6 product). ***This GPIO pin must not be allocated by any application.***

Param:
Bus parameters, defined by following values, using a logical OR to combine the different settings:

for SPI bus:

- **Clock speed:**

Speed constant	Supported on Q2XX3 and P3XX3 products	Supported on QXXX6 and P32X6 products	Supported on P5186 product
ADL_BUS_SPI_SCL_SPEED_13Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_6_5Mhz		Yes	Yes
ADL_BUS_SPI_SCL_SPEED_4_33Mhz		Yes	Yes
ADL_BUS_SPI_SCL_SPEED_3_25Mhz	Yes	Yes	Yes
ADL_BUS_SPI_SCL_SPEED_2_6Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_2_167Mhz		Yes	Yes
ADL_BUS_SPI_SCL_SPEED_1_857Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_1_625Mhz	Yes	Yes	
ADL_BUS_SPI_SCL_SPEED_1_44Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_1_3Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_1_181Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_1_083Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_1Mhz		Yes	
ADL_BUS_SPI_SCL_SPEED_926Khz		Yes	
ADL_BUS_SPI_SCL_SPEED_867Khz		Yes	
ADL_BUS_SPI_SCL_SPEED_812Khz	Yes	Yes	
ADL_BUS_SPI_SCL_SPEED_101Khz	Yes		

- **Clock mode:**
  ADL_BUS_SPI_CLK_MODE_0
  (the rest state is 0, data valid on rising edge)
  ADL_BUS_SPI_CLK_MODE_1
  (the rest state is 0, data valid on falling edge)
  ADL_BUS_SPI_CLK_MODE_2
  (the rest state is 1, data valid on rising edge)
  ADL_BUS_SPI_CLK_MODE_3
  (the rest state is 1, data valid on falling edge)

- **Chip Select Polarity:**
  ADL_BUS_SPI_CS_POL_LOW, *for low polarity*
  ADL_BUS_SPI_CS_POL_HIGH, *for high polarity*

- **Lsb or Msb first:**
  ADL_BUS_SPI_MSB_FIRST, *to send data MSB first*
  ADL_BUS_SPI_LSB_FIRST, *to send data LSB first*

- **Gpio Handling:**
  (only when an IO is used as Chip Select)
  ADL_BUS_SPI_BYTE_HANDLING,
  the IO signal pulse on each data byte,
  ADL_BUS_SPI_FRAME_HANDLING,
  the IO signal works as a normal chip select.

For I2C bus:

- **SCL signal GPIO:**
  The GPIO index to use to handle the SCL signal (shifted to the two MSBytes)

- **SDA signal GPIO:**
  The GPIO index to use to handle the SDA signal (on the two LSBytes)

Remark: the ADL_IO_ID_U32_TO_U16 macro should be used to convert the used GPIO ID to u16 type before calling the API.
Example:

```
Adl_busSubscribe( ADL_BUS_TYPE_IC2_SOFT,
      ADL_IO_ID_U32_TO_U16(MySDAGpio) |
      (ADL_IO_ID_U32_TO_U16(MySCLGpio)<<16));
```

For Parallel bus:

- **Data Order:**
  ADL_BUS_PARA_DATA_DIRECT_ORDER,
  to send data on direct order
  ADL_BUS_PARA_DATA_REVERSE_ORDER,
  to send data on reverse order

- **LCD_EN signal polarity (only for LCD_EN chip select):**
  ADL_BUS_PARA_LCDEN_POL_LOW
  data is sampled on the rising edge from low state to high state of LCD_EN.
  ADL_BUS_PARA_LCDEN_POL_HIGH
  data is sampled on the falling edge from high state to low state of LCD_EN.

- **LCD_EN Address Setup Time (only for LCD_EN chip select):**
  It is the time interval between the setting of an address for the Parallel bus and the activation of the LCD_EN pin. It is the T1 time on the figure below.
  The allowed values are from 0 to 31 (using bits 0 to 4).
  The resulting time interval is:
  For P32X3 product: (X * 38.5) ns ;
  For P32X6 product: (1 + 2 X) * 19 ns.

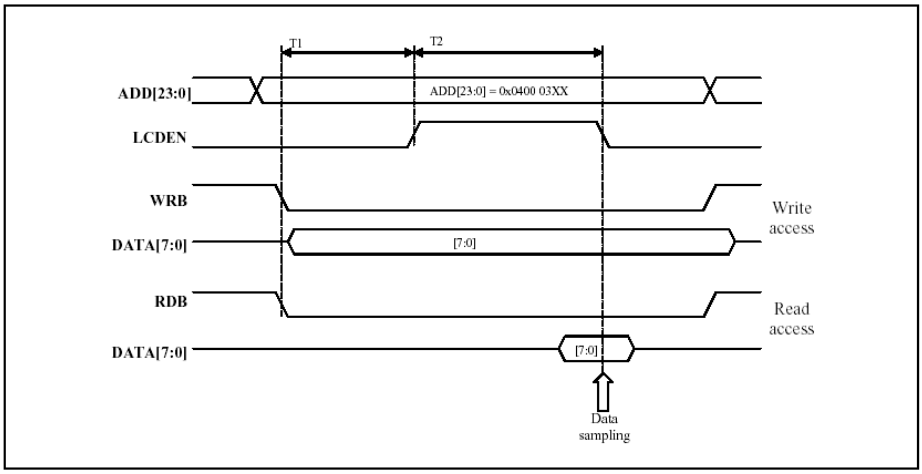


Figure 3: LCD_EN Address Setup chronogram

- **LCD_EN Signal Pulse Duration (only for LCD_EN chip select):**
  It is the time interval during which the LCD_EN pin is valid. It is the T2 time on the figure above.
  The allowed values are from 0 to 31 (using bits 5 to 10).
  The resulting time interval is:
  For P32X3 product: (X + 1.5) * 38.5 ns ;
  For P32X6 product: (1 + 2 * (X + 1)) * 19 ns.
  (Warning, for the P32X6 product, the 0 value in considered as 32).

- **CS_USER number of wait states (only for CS_USER chip select):**
  It is the time interval during which the data is valid on the bus, using the defined values:
  ADL_BUS_PARA_CSUSR_0_WAIT_STATE *(62 ns)*
  ADL_BUS_PARA_CSUSR_1_WAIT_STATE *(100 ns)*
  ADL_BUS_PARA_CSUSR_2_WAIT_STATE *(138 ns)*
  ADL_BUS_PARA_CSUSR_3_WAIT_STATE *(176 ns)*

- **Returned values**

  A positive or null bus handle on success.
  ADL_RET_ERR_PARAM if one parameter has an incorrect value
  ADL_RET_ERR_ALREADY_SUBSCRIBED if requested bus and address is already subscribed
  For other negative errors, please refer to the BUS API chapter of the Open AT® Basic Development Guide.

- **Remark**

  If one or more IOs are required to open a bus, these IOs must not be subscribed by any application. On the bus unsubscribe operation, the IOs can be subscribed again.

3.8.3 The adl_busUnsubscribe function

This function unsubscribes from a previously subscribed bus type

- **Prototype**

```
s8      adl_busUnsubscribe   ( u8       Handle );
```

- **Parameters**

  Handle:
  Handle previously returned by adl_busSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown.
  - For other negative errors, please refer to the BUS API chapter of the Open AT® Basic Development Guide.

3.8.4 The adl_busRead function

This function reads data from a previously subscribed bus type

- **Prototype**

```
s8      adl_busRead          (u8                Handle,
                              adl_busAccess_t    *pAccessMode,
                              u32                DataLen,
                              void *             Data );
```

- **Parameters**

  Handle:
  Handle previously returned by adl_busSubscribe function.

  pAccessMode:
  Bus access mode, defined according to the following type:

```
typedef struct
{
    u32 Address;
    u32 Opcode;
    u8 OpcodeLength;
    u8 AddressLength;
} adl_busAccess_t;
```

  This parameter is processed differently according the bus type:

  - For SPI bus:
    For Q24X3 and P32X3 products:

    one byte can be sent through the **Opcode** parameter
    (only the LSByte is used ; if **OpcodeLength** is less than 8 bits, only the MSBits of the LSByte are used),

    two bytes can be sent through the **Address** parameter
    (only the two LSBytes are used ; if **OpcodeLength** is less than 24 bits, only the MSBits of the two LSBytes are used),

    the **OpcodeLength** is the sum of **Opcode** and **Address** lengths in bits
    (if **OpcodeLength** is 0, nothing is sent ;
    if **OpcodeLength** < 9, just **Opcode** is sent ;
    if 8 < **OpcodeLength** < 25, **Opcode** then **Address** are sent),

    the **AddressLength** parameter is not used.

    For Q24X6, Q2400 and P32X6 products:

    Up to 32 bits can be sent through the **Opcode** parameter, according to the **OpcodeLength** parameter (in bits).
    if **OpcodeLength** is less than 32 bits, only MSBits are used.

    Up to 32 bits can be sent through the **Address** parameter, according to the **AddressLength** parameter (in bits).
    if **AddressLength** is less than 32 bits, only MSBits are used.

- For I2C soft bus:
  Not used, this parameter should be NULL.

- For Parallel bus:
  Only the **Address** parameter is used.
  This parameter is used to set the A2 pin value ; it can be set to following values:
  WM_BUS_PARA_ADDRESS_A2_SET, to set the A2 pin ;
  WM_BUS_PARA_ADDRESS_A2_RESET, to reset the A2 pin

DataLen:
Number of bytes to read from the bus.

Data:
Buffer where to copy the read bytes.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown,
  - ADL_RET_ERR_PARAM if a parameter has an incorrect value,
  - For other negative errors, please refer to the BUS API chapter of the Open AT® Basic Development Guide.

3.8.5 The adl_busWrite function

This function writes on a previously subscribed bus.

- **Prototype**

```
s8      adl_busWrite          ( u8               Handle,
                                adl_busAccess_t  * pAccessMode,
                                u32              DataLen,
                                void *           Data );
```

- **Parameters**

Handle:
Handle previously returned by adl_busSubscribe function.

pAccessMode:
Bus access mode, defined with the following type:

```
typedef struct
{
    u32 Address;
    u32 Opcode;
    u8 OpcodeLength;
    u8 AddressLength;
} adl_busAccess_t;
```

This parameter is processed differently according the bus type:

- **For SPI bus**:
  - *For Q24X3 and P32X3 products:*

    one byte can be sent through the **Opcode** parameter (only the LSByte is used ; if **OpcodeLength** is less than 8 bits, only the MSBits of the LSByte are used),

    two bytes can be sent through the **Address** parameter (only the two LSBytes are used ; if **OpcodeLength** is less than 24 bits, only the MSBits of the two LSBytes are used),

    the **OpcodeLength** is the sum of **Opcode** and **Address** lengths in bits
    (if **OpcodeLength** is 0, nothing is sent ;
    if **OpcodeLength** < 9, just **Opcode** is sent ;
    if 8 < **OpcodeLength** < 25, **Opcode** then **Address** are sent),

    the **AddressLength** parameter is not used.

    For Q24X6, Q2400 and P32X6 products:

    Up to 32 bits can be sent through the **Opcode** parameter, according to the **OpcodeLength** parameter (in bits).
    if **OpcodeLength** is less than 32 bits, only MSBits are used.

    Up to 32 bits can be sent through the **Address** parameter, according to the **AddressLength** parameter (in bits).
    if **AddressLength** is less than 32 bits, only MSBits are used.

- **For I2C soft bus**:
  Not used, this parameter should be NULL.

- **For Parallel bus:**
  Only the **Address** parameter is used.
  This parameter is used to set the A2 pin value ; it can be set to following values:
  WM_BUS_PARA_ADDRESS_A2_SET, to set the A2 pin ;
  WM_BUS_PARA_ADDRESS_A2_RESET, to reset the A2 pin

DataLen:
Number of bytes to write on the bus.

Data:
Data buffer to write on the bus.

- **Returned values**

  OK on success.
  ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown,
  ADL_RET_ERR_PARAM if a parameter has an incorrect value,
  For other negative errors, please refer to the BUS API chapter of the Open AT® Basic Development Guide.

3.9 Errors management

3.9.1 Required Header File

The header file for the error functions is:

```
adl_errors.h
```

3.9.2 The adl_errSubscribe function

This function subscribes to error service and gives an error handler.

- **Prototype**

```
s8      adl_errSubscribe     ( adl_errHdlr_f  Handler );
```

- **Parameters**

  Handler:
  Error Handler, defined on following type:

```
typedef bool ( * adl_errHdlr_f ) ( u16 ErrorID, ascii * ErrorStr );
```

  An error is described by an Id and a string (associated text), that are sent as parameters to the adl_errHalt function.
  If the error is processed and filtered the handler should return FALSE. The return value TRUE will cause the product to execute a fatal error reset with a back trace.
  Note that ErrorID below 0x0100 are for internal purpose so you should only use ErrorID above 0x0100.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter has an incorrect value
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service is already subscribed

3.9.3 The adl_errUnsubscribe function

This function unsubscribes from error service.

- **Prototype**

```
s8      adl_errUnsubscribe   ( adl_errHdlr_f  Handler);
```

- **Parameters**

  Handler:
  Handler returned by adl_errSubscribe function

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter has an incorrect value
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handler is unknown
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed

3.9.4 The adl_errHalt function

This function causes an error, defined by its ID and string. If an error handler is defined, it will be called, otherwise a product reset will occur.

- **Prototype**

```
void    adl_errHalt     (  u16          ErrorID
                           ascii *      ErrorString );
```

- **Parameters**

  ErrorID:
  Error ID

  ErrorString:
  Error string available to the error handler.

3.10 SIM Service

ADL provides this service to handle SIM and PIN code related events.

3.10.1 Required Header File

The header file for the SIM related functions is:

`adl_sim.h`

3.10.2 The adl_simSubscribe function

This function subscribes to the SIM service, in order to receive SIM and PIN code related events. This will allow to enter PIN code (if provided) if necessary.

- **Prototype**

```
void adl_simSubscribe   ( adl_simHdlr_f   SimHandler,
                          ascii *         PinCode );
```

- **Parameters**

SimHandler:
SIM handler defined using the following type:

```
typedef void ( * adl_simHdlr_f ) ( u8 Event );
```

The events received by this handler are defined below.
Normal events:

ADL_SIM_EVENT_PIN_OK
if PIN code is all right
ADL_SIM_EVENT_REMOVED
if SIM card is removed
ADL_SIM_EVENT_INSERTED
if SIM card is inserted
ADL_SIM_EVENT_FULL_INIT
when initialization is done

Error events:

ADL_SIM_EVENT_PIN_ERROR
if given PIN code is wrong
ADL_SIM_EVENT_PIN_NO_ATTEMPT
if there is only one attempt left to entered the right PIN code
ADL_SIM_EVENT_PIN_WAIT
if the argument PinCode is set to NULL
On the last three events, the service is waiting for the external application to enter the PIN code.
Please note that the deprecated ADL_SIM_EVENT_ERROR event has been removed since the ADL version 3. This code was mentioned in version 2 documentation, but was never generated by the SIM service.

PinCode:
It is a string containing the PIN code text to enter. If it is set to NULL or if the provided code is incorrect, the PIN code will have to be entered by the external application.
This argument is used only the first time the service is subscribed. It is ignored on all further subscriptions.

3.10.3 The adl_simUnsubscribe function

This function unsubscribes from SIM service. The provided handler will not receive SIM events any more.

- **Prototype**

```
void    adl_simUnsubscribe  ( adl_simHdlr_f       Handler)
```

- **Parameters**

  Handler:
  Handler used with adl_SimSubscribe function.

3.10.4 The adl_simGetState function

This function gets the current SIM service state.

- **Prototype**

```
void  adl_simState_e adl_simGetState ( void );
```

- **Returned values**

  The returned value is the SIM service state, based on following type:

```
typedef enum
{
      ADL_SIM_STATE_INIT,  // Service init state (PIN state not known yet)
      ADL_SIM_STATE_REMOVED,  // SIM removed
      ADL_SIM_STATE_INSERTED, // SIM inserted (PIN state not known yet)
      ADL_SIM_STATE_FULL_INIT, // SIM Full Init done
      ADL_SIM_STATE_PIN_ERROR, // SIM error state
      ADL_SIM_STATE_PIN_OK,   // PIN code OK, waiting for full init
      ADL_SIM_STATE_PIN_WAIT, // SIM inserted, PIN code not entered yet

      /* Always last State */
      ADL_SIM_STATE_LAST
} adl_simState_e;
```

3.11 SMS Service

ADL provides this service to handle SMS events, and to send SMS to the network.

3.11.1 Required Header File

The header file for the SMS related functions is:

`adl_sms.h`

3.11.2 The adl_smsSubscribe function

This function subscribes to the SMS service in order to receive SMS from the network.

- **Prototype**

```
s8      adl_smsSubscribe ( adl_smsHdlr_f       SmsHandler,
                           adl_smsCtrlHdlr_f SmsCtrlHandler,
                           u8                Mode );
```

- **Parameters**

SmsHandler:
SMS handler defined using the following type:

```
typedef bool ( * adl_smsHdlr_f ) (  ascii * SmsTel,
                                    ascii * SmsTimeLength,
                                    ascii * SmsText );
```

This handler is called each time a SMS is received from the network.
SmsTel contains the originating telephone number of the SMS (in text mode), or NULL (in PDU mode).
SmsTimeLength contains the SMS time stamp (in text mode), or the PDU length (in PDU mode).
SmsText contains the SMS text (in text mode), or the SMS PDU (in PDU mode).
This handler returns TRUE if the SMS must be forwarded to the external application (it is then stored in SIM memory, and the external application is then notified by a "+CMTI" unsolicited indication).
It returns FALSE if the SMS should not be forwarded.
If the SMS service is subscribed several times, a received SMS will be forwarded to the external application only if each of the handlers return TRUE.

SmsCtrlHandler:
SMS event handler, defined using the following type:

```
typedef void ( * adl_smsCtrlHdlr_f ) ( u8 Event, u16 Nb );
```

This handler is notified by following events during a SMS sending process.

ADL_SMS_EVENT_SENDING_OK
*the SMS was sent successfully, **Nb** parameter value is not relevant.*

ADL_SMS_EVENT_SENDING_ERROR
*An error occurred during SMS sending, **Nb** parameter contains the error number, according to "+CMS ERROR" value (cf. AT Commands Interface Guide).*

ADL_SMS_EVENT_SENDING_MR
*the SMS was sent successfully, **Nb** parameter contains the sent Message Reference value. A ADL_SMS_EVENT_SENDING_OK event will be received by the control handler.*

Mode:
Mode used for SMS reception from the following values:

ADL_SMS_MODE_PDU
SmsHandler will be called in PDU mode on each SMS reception.

ADL_SMS_MODE_TEXT
SmsHandler will be called in Text mode on each SMS reception.

- **Returned values**
  - On success, this function returns a positive or null handle, requested for further SMS sending operations.
  - ADL_RET_ERR_PARAM if a parameter has a wrong value.

3.11.3 The adl_smsSend function

This function sends a SMS to the network.

- **Prototype**

```
s8      adl_smsSend     ( u8            Handle,
                          ascii *       SmsTel,
                          ascii *       SmsText,
                          u8            Mode );
```

- **Parameters**

  Handle:
  Handle returned by adl_smsSubscribe function.

  SmsTel:
  Telephone number where to send the SMS (in text mode), or NULL (in PDU mode).

  SmsText:
  SMS text (in text mode), or SMS PDU (in PDU mode).

  Mode:
  Mode used for SMS sending from the following values:
  ADL_SMS_MODE_PDU
  to send a SMS in PDU mode.
  ADL_SMS_MODE_TEXT
  to send a SMS in Text mode.

- **Returned values**
  - This function returns OK on success.
  - ADL_RET_ERR_PARAM if a parameter has a wrong value.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the product is not ready to send a SMS (initialization not done yet, or sending a SMS already in progress)

3.11.4 The adl_smsUnsubscribe function

This function unsubscribes from SMS service. The associated handler with provided handle will not receive SMS events any more.

- **Prototype**

```
s8      adl_smsUnsubscribe  ( u8          Handle)
```

- **Parameters**

  Handle:
  Handle returned by adl_smsSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handler is unknown.
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed.
  - ADL_RET_ERR_BAD_STATE if the service is processing a SMS

3.12 Call Service

ADL provides this service to handle call related events, and to setup calls.

3.12.1 Required Header File

The header file for the call related functions is:

```
adl_call.h
```

3.12.2 The adl_callSubscribe function

This function subscribes to the call service in order to receive call related events.

- **Prototype**

```
s8      adl_callSubscribe ( adl_callHdlr_f CallHandler  );
```

- **Parameters**

  CallHandler:
  Call handler defined using the following type:

  typedef s8 (* adl_callHdlr_f) (u16 Event, u32 Call_ID);

  The pairs events / call Id received by this handler are defined below:

Event / Call ID	Description
ADL_CALL_EVENT_RING_VOICE / 0	*if voice phone call*
ADL_CALL_EVENT_RING_DATA / 0	*if data phone call*
ADL_CALL_EVENT_NEW_ID / X	*if wind: 5,X*
ADL_CALL_EVENT_RELEASE_ID / X	*if wind: 6,X ; on data call release, X is a logical OR between the Call ID and the ADL_CALL_DATA_FLAG constant*
ADL_CALL_EVENT_ALERTING / 0	*if wind: 2*
ADL_CALL_EVENT_NO_CARRIER / 0	*phone call failure, 'NO CARRIER'*
ADL_CALL_EVENT_NO_ANSWER / 0	*phone call failure, no answer*
ADL_CALL_EVENT_BUSY / 0	*phone call failure, busy*
ADL_CALL_EVENT_SETUP_OK / Speed	*ok response after a call setup performed by the adl_callSetup function; in data call setup case, the connection <Speed> (in bits/second) is also provided.*
ADL_CALL_EVENT_ANSWER_OK / Speed	*ok response after an ADL_CALL_NO_FORWARD_ATA request from a call handler ; in data call answer case, the connection <Speed> (in bps) is also provided*

Event / Call ID	Description
ADL_CALL_EVENT_HANGUP_OK / Data	*ok response after a ADL_CALL_NO_FORWARD_ATH request, or a call hangup performed by the adl_callHangup function ; on data call release, Data is the ADL_CALL_DATA_FLAG constant (0 on voice call release)*
ADL_CALL_EVENT_SETUP_OK_FROM_EXT / Speed	*ok response after an 'ATD' command from the external application; in data call setup case, the connection <Speed> (in bits/second) is also provided.*
ADL_CALL_EVENT_ANSWER_OK_FROM_EXT / Speed	*ok response after an 'ata' command from the external application ; in data call answer case, the connection <Speed> (in bps) is also provided*
ADL_CALL_EVENT_HANGUP_OK_FROM_EXT / Data	*ok response after an 'ATH' command from the external application ; on data call release, Data is the ADL_CALL_DATA_FLAG constant (0 on voice call release)*
ADL_CALL_EVENT_AUDIO_OPENNED / 0	*if +WIND: 9*
ADL_CALL_EVENT_ANSWER_OK_AUTO / Speed	*OK response after an auto-answer to an incoming call (ATS0 command was set to a non-zero value) ; in data call answer case, the connection <Speed> (in bps) is also provided*
ADL_CALL_EVENT_RING_GPRS / 0	*if GPRS phone call*
ADL_CALL_EVENT_SETUP_FROM_EXT / Mode	*if the external application has used the 'ATD' command to setup a call. Mode value depends on call type (Voice: 0, GSM Data: ADL_CALL_DATA_FLAG, GPRS session activation: binary OR between ADL_CALL_GPRS_FLAG constant and the activated CID). According to the notified handlers return values, the call setup may be launched or not: if at least one handler returns the ADL_CALL_NO_FORWARD code (or higher), the command will reply "+CME ERROR: 600" to the external application ; otherwise (if all handlers return ADL_CALL_FORWARD) the call setup is launched.*

Event / Call ID	Description
ADL_CALL_EVENT_SETUP_ERROR_NO_SIM / 0	*A call setup (from embedded or external application) has failed (no SIM card inserted)*
ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY / 0	*A call setup (from embedded or external application) has failed (the PIN code is not entered)*
ADL_CALL_EVENT_SETUP_ERROR / Error	*A call setup (from embedded or external application) has failed (the <Error> field is the returned +CME ERROR value ; cf. AT Commands interface guide for more information)*

The events returned by this handler are defined below:

Event	Description
ADL_CALL_FORWARD	*the event of the call is to be sent to the external application*
ADL_CALL_NO_FORWARD	*the event of the call is not to be sent to the external application*
ADL_CALL_NO_FORWARD_ATH	*the event of the call is not to be sent to the external application and the application shall terminate the call by sending an 'ATH' command.*
ADL_CALL_NO_FORWARD_ATA	*the event of the call is not to be sent to the external application and the application shall answer the call by sending an 'ATA' command.*

- **Returned values**

  This function returns a positive or null handle on success, or a negative error value.

3.12.3 The adl_callSetup function

This function sets up a call to a specified phone number.

- **Prototype**

```
s8      adl_callSetup    ( ascii *          PhoneNb,
                           u8               Mode );
```

- **Parameters**

  PhoneNb:
  Phone number to use to set up the call.

  Mode:
  Mode used to set up the call:
  ADL_CALL_MODE_VOICE,
  ADL_CALL_MODE_DATA

- **Returned values**

  This function returns a negative error value, or 0 on success.

3.12.4 The adl_callHangup function

This function hangs up the phone call.

- **Prototype**

```
s8      adl_callHangup   ( void );
```

- **Returned values**

  This function should return a negative error value, or 0 on success.

3.12.5 The adl_callAnswer function

This function allows the application to answer a phone call out of the call events handler.

- **Prototype**

```
s8      adl_callAnswer   ( void );
```

- **Returned values**

  This function should return a negative error value, or 0 on success.

3.12.6 The adl_callUnsubscribe function

This function unsubscribes from the Call service. The provided handler will not receive Call events any more.

- **Prototype**

```
s8      adl_callUnsubscribe ( adl_callHdlr_f Handler );
```

- **Parameters**

  Handler:
  Handler used with adl_callSubscribe function.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM on parameter error
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handler is unknown
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed.

3.13 GPRS Service

ADL provides this service to handle GPRS related events and to setup, activate and deactivate PDP contexts.

3.13.1 Required Header File

The header file for the GPRS related functions is:

`adl_gprs.h`

3.13.2 The adl_gprsSubscribe function

This function subscribes to the GPRS service in order to receive GPRS related events.

- **Prototype**

```
s8      adl_gprsSubscribe ( adl_gprsHdlr_f GprsHandler  );
```

- **Parameters**

  GprsHandler:
  GPRS handler defined using the following type:

```
typedef s8 (*adl_gprsHdlr_f)(u16 Event, u8 Cid);
```

  The pairs events/Cid received by this handler are defined below:

Event / Call ID	Description
ADL_GPRS_EVENT_RING_GPRS	*If incoming PDP context activation is requested by the network*
ADL_GPRS_EVENT_NW_CONTEXT_DEACT / X	*If the network has forced the deactivation of the Cid X*
ADL_GPRS_EVENT_ME_CONTEXT_DEACT / X	*If the ME has forced the deactivation of the Cid X*
ADL_GPRS_EVENT_NW_DETACH	*If the network has forced the detachment of the ME*
ADL_GPRS_EVENT_ME_DETACH	*If the ME has forced a network detachment or lost the network*
ADL_GPRS_EVENT_NW_CLASS_B	*If the network has forced the ME on class B*
ADL_GPRS_EVENT_NW_CLASS_CG	*If the network has forced the ME on class CG*
ADL_GPRS_EVENT_NW_CLASS_CC	*If the network has forced the ME on class CC*
ADL_GPRS_EVENT_ME_CLASS_B	*If the ME has changed his class to class B*
ADL_GPRS_EVENT_ME_CLASS_CG	*If the ME has changed his class to class CG*
ADL_GPRS_EVENT_ME_CLASS_CC	*If the ME has changed his class to class CC*

Event / Call ID	Description
ADL_GPRS_EVENT_NO_CARRIER	*If the activation of the external application with 'ATD*99' (PPP dialing) did hang up.*
ADL_GPRS_EVENT_DEACTIVATE_OK / X	*If the deactivation requested with adl_gprsDeact() function did succeed on the Cid X*
ADL_GPRS_EVENT_DEACTIVATE_OK_FROM_EXT / X	*If the deactivation requested by the external application succeed on the Cid X*
ADL_GPRS_EVENT_ANSWER_OK	*If the acceptance of the incoming PDP activation with adl_gprsAct() did succeed*
ADL_GPRS_EVENT_ANSWER_OK_FROM_EXT	*If the acceptance of the incoming PDP activation by the external application did succeed*
ADL_GPRS_EVENT_ACTIVATE_OK / X	*If the activation requested with adl_gprsAct() on the Cid X did succeed*
ADL_GPRS_EVENT_GPRS_DIAL_OK_FROM_EXT / X	*If the activation requested by the external application with 'ATD*99' (PPP dialing) did succeed on the Cid X*
ADL_GPRS_EVENT_ACTIVATE_OK_FROM_EXT / X	*If the activation requested by the external application on the Cid X did succeed*
ADL_GPRS_EVENT_HANGUP_OK_FROM_EXT	*If the rejection of the incoming PDP activation by the external application did succeed*
ADL_GPRS_EVENT_DEACTIVATE_KO / X	*If the deactivation requested with adl_gprsDeact() on the Cid X did fail*
ADL_GPRS_EVENT_DEACTIVATE_KO_FROM_EXT / X	*If the deactivation requested by the external application on the Cid X did fail*
ADL_GPRS_EVENT_ACTIVATE_KO_FROM_EXT / X	*If the activation requested by the external application on the Cid X did fail*
ADL_GPRS_EVENT_ACTIVATE_KO / X	*If the activation requested with adl_gprsAct() on the Cid X did fail*
ADL_GPRS_EVENT_ANSWER_OK_AUTO	*If the incoming PDP context activation was automatically accepted by the ME*
ADL_GPRS_EVENT_SETUP_OK / X	*If the set up of the Cid X with adl_gprsSetup() did succeed*
ADL_GPRS_EVENT_SETUP_KO / X	*If the set up of the Cid X with adl_gprsSetup() did fail*
ADL_GPRS_EVENT_ME_ATTACH	*If the ME has forced a network attachment*
ADL_GPRS_EVENT_ME_UNREG	*If the ME is not registered*
ADL_GPRS_EVENT_ME_UNREG_SEARCHING	*If the ME is not registered but is searching a new operator to register to.*

Note: If Cid X is not defined, the value ADL_CID_NOT_EXIST will be used as X.

The events returned by this handler are defined below:

Event	Description
ADL_GPRS_FORWARD	*the event shall be sent to the external application*
ADL_GPRS_NO_FORWARD	*the event shall not be sent to the external application*
ADL_GPRS_NO_FORWARD_ATH	*the event shall not be sent to the external application and the application shall terminate the incoming activation request by sending an 'ATH' command.*
ADL_GPRS_NO_FORWARD_ATA	*the event shall not be sent to the external application and the application shall accept the incoming activation request by sending an 'ATA' command.*

- **Returned values**

  This function returns 0 on success, or a negative error value.

3.13.3 The adl_gprsSetup function

This function sets up a PDP context identified by its CID with some specific parameters.

- **Prototype**

```
s8 adl_gprsSetup(u8 Cid, adl_gprsSetupParams_t  Params);
```

- **Parameters**

  Cid:
  The Cid of the PDP context to setup.

  Params:
  Structure containing the parameters to set up using the following type:

```
typedef struct
{

 ascii* APN; // Address of the Provider GPRS Gateway GGSN
                 //(max length 100 bytes)
 ascii* Login; // Login of the GPRS account (max length 50 bytes)
 ascii* Password; // Password of the GPRS account (max lng 50 bytes)
 ascii* FixedIP; // Optional Fixed IP address of the MS
}adl_gprsSetupParams_t;
```

- **Returned values**

  This function returns 0 on success, or a negative error value.
  Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	*In case of parameter error: Cid value must be included between 1 to 4*
ADL_RET_ERR_PIN_KO	*If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet.*
ADL_GPRS_CID_NOT_DEFINED	*in case of problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*f the GPRS service is not supported by the product.*
ADL_RET_ERR_BAD_STATE	*The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function.*

3.13.4 The adl_gprsAct function

This function activates a specific PDP context identified by its Cid.

- **Prototype**

```
s8 adl_gprsAct(u8 Cid);
```

- **Parameters**

  Cid:
  The Cid of the PDP context to activate.

- **Returned values**

  This function returns 0 on success, or a negative error value.

  Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	*in case of parameters error: Cid value must be included between 1 to 4*
ADL_RET_ERR_PIN_KO	*If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet.*
ADL_GPRS_CID_NOT_DEFINED	*in case of problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*f the GPRS service is not supported by the product.*
ADL_RET_ERR_BAD_STATE	*The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function.*

  Important Note: This function must be called before opening the GPRS FCM Flows.

3.13.5 The adl_gprsDeact function

This function deactivates a specific PDP context identified by its Cid.

- **Prototype**

```
s8 adl_gprsDeact(u8 Cid);
```

- **Parameters**

  Cid:
  The Cid of the PDP context to deactivate.

- **Returned values**

  This function returns 0 on success, or a negative error value.

  Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	*in case of parameters error: Cid value must be included between 1 to 4*
ADL_RET_ERR_PIN_KO	*If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet.*
ADL_GPRS_CID_NOT_DEFINED	*in case of problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*f the GPRS service is not supported by the product.*
ADL_RET_ERR_BAD_STATE	*The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function.*

IMPORTANT NOTE: if the GPRS flow is running, please do wait for the ADL_FCM_EVENT_FLOW_CLOSED event before calling the adl_gprsDeact function, in order to prevent module lock.

3.13.6 The adl_gprsGetCidInformations function

This function gets information about a specific activated PDP context identified by its Cid.

- **Prototype**

```
s8 adl_gprsGetCidInformations (u8 Cid,adl_gprsInfosCid_t * Infos);
```

- **Parameters**

Cid:
The Cid of the PDP context.

Infos:
Structure containing the information of the activated PDP context using the following type:

```
typedef struct
{
    u32 LocalIP; // Local IP address of the MS (only if is activated,
else 0)
    u32 DNS1; // First DNS IP address (only if is activated, else 0)
    u32 DNS2; // Second DNS IP address (only if is activated, else 0)
    u32 Gateway; // Gateway IP address (only if is activated, else 0)
}adl_gprsInfosCid_t;
```

- **Returned values**

This function returns 0 on success, or a negative error value.

Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	*in case of parameters error: Cid value must be included between 1 to 4*
ADL_RET_ERR_PIN_KO	*If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet.*
ADL_GPRS_CID_NOT_DEFINED	*in case of problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*f the GPRS service is not supported by the product.*
ADL_RET_ERR_BAD_STATE	*The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function.*

3.13.7 The adl_gprsUnsubscribe function

This function unsubscribes from the GPRS service. The provided handler will not receive GPRS events any more.

- **Prototype**

```
s8      adl_gprsUnsubscribe ( adl_gprsHdlr_f Handler );
```

- **Parameters**

  Handler:
  Handler used with adl_gprsSubscribe function.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM on parameter error
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handler is unknown
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed.

3.14 Application Safe Mode Service

By default, the +WOPEN and +WDWL commands can not be filtered by any embedded application. This service allows one application to get these commands events, in order to prevent any external application to stop or erase the current embedded one.

3.14.1 Required Header File

The header file for the Application safe mode service is:

`adl_safe.h`

3.14.2 The adl_safeSubscribe function

This function subscribes to the Application safe mode service in order to receive +WOPEN and +WDWL commands events.

- **Prototype**

```
s8      adl_safeSubscribe ( u16            WDWLopt,
                            u16            WOPENopt,
                            adl_safeHdlr_f SafeHandler  );
```

- **Parameters**

  WDWLopt:
  Additional options for +WDWL command subscription. This command is at least subscribed in ACTION and READ mode. Please see adl_atCmdSubscribe API for more details on these options.

  WOPENopt:
  Additional options for +WOPEN command subscription. This command is at least subscribed in READ, TEST and PARAM mode, with minimum one mandatory parameter. Please see adl_atCmdSubscribe API for more details on these options.

SafeHandler:
Application safe mode handler defined using the following type:

```
typedef bool (*adl_safeHdlr_f) ( adl_safeCmdType_e CmdType,
                                 adl_atCmdPreParser_t * paras );
```

The CmdType events received by this handler are defined below:

```
typedef enum
{
   ADL_SAFE_CMD_WDWL,                // AT+WDWL command
   ADL_SAFE_CMD_WDWL_READ,           // AT+WDWL? command
   ADL_SAFE_CMD_WDWL_OTHER,          // WDWL other syntax

   ADL_SAFE_CMD_WOPEN_STOP,          // AT+WOPEN=0 command
   ADL_SAFE_CMD_WOPEN_START,         // AT+WOPEN=1 command
   ADL_SAFE_CMD_WOPEN_GET_VERSION,   // AT+WOPEN=2 command
   ADL_SAFE_CMD_WOPEN_ERASE_OBJ,     // AT+WOPEN=3 command
   ADL_SAFE_CMD_WOPEN_ERASE_APP,     // AT+WOPEN=4 command
   ADL_SAFE_CMD_WOPEN_SUSPEND_APP,   // AT+WOPEN=5 command
   ADL_SAFE_CMD_WOPEN_READ,          // AT+WOPEN? command
   ADL_SAFE_CMD_WOPEN_TEST,          // AT+WOPEN=? command
   ADL_SAFE_CMD_WOPEN_OTHER          // WOPEN other syntax
} adl_safeCmdType_e;
```

The **paras** received structure contains the same parameters as is the commands were subscribed with adl_atCmdSubscribe API.

If the Handler returns FALSE, the command will not be forwarded to the Wavecom core software.
If the Handler returns TRUE, the command will be processed by the Wavecom core software, which will send responses to the external application.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameters have an incorrect value
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service is already subscribed

3.14.3 The adl_safeUnsubscribe function

This function unsubscribes from Application safe mode service. The +WDWL and +WOPEN commands are not filtered anymore and always processed by the Wavecom core software.

- **Prototype**

```
s8      adl_safeUnsubscribe  ( adl_safeHdlr_f Handler);
```

- **Parameters**

  Handler:
  Handler used with adl_safeSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter has an incorrect value
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handler is unknown
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed

3.14.4 The adl_safeRunCommand function

This function allows to run +WDWL or +WOPEN command with any standard syntax, and to get its answers.

- **Prototype**

```
s8      adl_safeRunCommand   ( adl_safeCmdType_e  CmdType,
                               adl_atRspHandler_t RspHandler );
```

- **Parameters**

  CmdType:
  Command type to run ; please refer to adl_safeSubscribe description. ADL_SAFE_CMD_WDWL_OTHER and ADL_SAFE_CMD_WOPEN_OTHER values are not allowed.
  The ADL_SAFE_CMD_WOPEN_SUSPEND_APP may be used to suspend the Open AT® application task. The execution may be resumed using the AT+WOPENRES command, or by sending a signal on the hardware Interrupt product pin (The INTERRUPT feature has to be enabled on the product : please refer to the AT+WFM command). Open AT® application running in Remote Task Environment can not be suspended (the function has no effect).
  RspHandler:
  Response handler to get ran commands' results. All responses are subscribed. If no response handler is provided (NULL parameter), the responses are forwarded to the external application.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter has an incorrect value

3.15 AT Strings Service

This service provides APIs to process AT standard response strings.

3.15.1 Required Header File

The header file for the AT strings service is:

`adl_str.h`

3.15.2 The adl_strID_e type

This type defines all pre-defined AT strings by this service, defined below:

```
typedef enum
{
      ADL_STR_NO_STRING,  // Unknown string

      ADL_STR_OK,         // "OK"
      ADL_STR_BUSY,       // "BUSY"
      ADL_STR_NO_ANSWER,  // "NO ANSWER"
      ADL_STR_NO_CARRIER, // "NO CARRIER"
      ADL_STR_CONNECT,    // "CONNECT"
      ADL_STR_ERROR,      // "ERROR"
      ADL_STR_CME_ERROR,  // "+CME ERROR:"
      ADL_STR_CMS_ERROR,  // "+CMS ERROR:"
      ADL_STR_CPIN,       // "+CPIN:"

      ADL_STR_LAST_TERMINAL, // Terminal resp. are before this line

      ADL_STR_RING = ADL_STR_LAST_TERMINAL,   // "RING"
      ADL_STR_WIND,       // "+WIND:"
      ADL_STR_CRING,      // "+CRING:"
      ADL_STR_CPINC,      // "+CPINC:"
      ADL_STR_WSTR,       // "+WSTR:"

      // Last string ID
      ADL_STR_LAST
} adl_strID_e;
```

3.15.3 The adl_strGetID function

This function returns the ID of the provided response string.

- **Prototype**

```
adl_strID_e adl_strGetID (  ascii * rsp  );
```

- **Parameters**

  rsp:
  String to parse to get the ID.

- **Returned values**
  - ADL_STR_NO_STRING if the string is unknown.
  - Id of the string otherwise.

3.15.4 The adl_strGetIDExt function

This function returns the ID of the provided response string, with an optional argument and its type.

- **Prototype**

```
adl_strID_e adl_strGetIDExt (  ascii * rsp
                               void *  arg
                               u8 *    argtype  );
```

- **Parameters**

  rsp:
  String to parse to get the ID.

  arg:
  Parsed first argument ; not used if set to NULL.

  argtype:
  Type of the parsed argument:
  if argtype is **`ADL_STR_ARG_TYPE_ASCII`**, arg is an **`ascii *`** string ;
  if argtype is **`ADL_STR_ARG_TYPE_U32`**, arg is an **`u32 *`** integer.

- **Returned values**
  - ADL_STR_NO_STRING if the string is unknown.
  - Id of the string otherwise.

3.15.5 The adl_strIsTerminalResponse function

This function checks whether the provided response ID is a terminal one. A terminal response is the last response that a response handler will receive from a sent command.

- **Prototype**

```
bool  adl_strIsTerminalResponse  ( adl_strID_e RspID );
```

- **Parameters**

  RspID:
  Response ID to check.

- **Returned values**
  - TRUE if the provided response ID is a terminal one.
  - FALSE otherwise.

3.15.6 The adl_strGetResponse function

This function provides the standard response string from its ID.

- **Prototype**

```
ascii * adl_strGetResponse  ( adl_strID_e   RspID );
```

- **Parameters**

  RspID:
  Response ID from which to get the string.

- **Returned values**
  - Standard response string on success ;
  - NULL if the ID does not exist.

IMPORTANT WARNING:
The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application ; ie. the embedded application will have to release the returned pointer.

3.15.7 The adl_strGetResponseExt function

This function provides a standard response string from its ID, with the provided argument.

- **Prototype**

```
ascii * adl_strGetResponseExt  ( adl_strID_e RspID,
                                 u32         arg );
```

- **Parameters**

  RspID:
  Response ID from which to get the string.

  arg:
  Response argument to copy in the response string ; according to response ID, this argument should be an **u32** integer value, or an **ascii *** string.

- **Returned values**

  Standard response string on success ;
  NULL if the ID does not exist.

IMPORTANT WARNING:
The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application ; ie. the embedded application will have to release the returned pointer.

3.16 Application & Data storage Service

This service provides APIs to use the Application & Data storage volume. This volume may be used to store data, or ".dwl" files (new Open AT® applications) in order to be later installed on the product. The maximum storage size is 512 KBytes.

3.16.1 Required Header File

The header file for the Application & Data storage service is:

`adl_ad.h`

3.16.2 The adl_adSubscribe function

This function subscribes to the required A&D space cell identifier.

- **Prototype**

```
s32   adl_adSubscribe  (  u32   CellID
                          u32   Size );
```

- **Parameters**

  CellID:
  A&D space cell identifier to subscribe to ; this cell may already exist or not. If the cell does not exist, the given size is allocated.

  Size:

  New cell size in bytes (this parameter is ignored if the cell already exists). It may be set to ADL_AD_SIZE_UNDEF for a variable size. In this case, new cells subscription will fail until the undefined size cell is finalized.

  Total used size in flash will be data size + header size ; header size is variable (with an average value of 16 bytes).

  When Subscribing, the size is rounded at the multiple of 4 superior.

- **Returned values**
  - The cell positive or null handle on success ;
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the cell is already subscribed;
  - ADL_AD_RET_ERR_OVERFLOW if there is not enough space for the allocation;
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product.
  - ADL_RET_ERR_PARAM if the CellId parameter is 0xFFFFFFFF (this value should not be used as a A&D Cell ID).
  - ADL_RET_ERR_BAD_STATE (when subscribing an undefined size cell) if another undefined size cell is already subscribed and not finalized.

3.16.3 The adl_adUnsubscribe function

This function unsubscribes from the given A&D cell handle.

- **Prototype**

```
s32   adl_adUnsubscribe ( u32  Handle );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed.

3.16.4 The adl_adWrite function

This function writes data at the end of the given A&D cell.

- **Prototype**

```
s32   adl_adWrite ( u32     Handle
                    u32     Size
                    void *  Data  );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

  Size:
  Data buffer size in bytes.

  Data:
  Data buffer.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed ;
  - ADL_RET_ERR_PARAM on parameter error ;
  - ADL_RET_ERR_BAD_STATE if the cell is finalized ;
  - ADL_AD_RET_ERR_OVERFLOW if the write operation exceed the cell size.

3.16.5 The adl_adInfo function

This function provides information on the requested A&D cell.

- **Prototype**

```
s32  adl_adInfo  (  u32              Handle
                    adl_adInfo_t *   Info );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

  Info:
  Information structure on requested cell, based on following type:

```
typedef struct
{
 u32   identifier;   // identifier
 u32   size;         // entry size
 void  *data;        // pointer to stored data
 u32   remaining;    // remaining writable space unless finalized
 bool  finalised;    // TRUE if entry is finalized
}adl_adInfo_t;
```

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_PARAM on parameter error ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed.
  - ADL_RET_ERR_BAD_STATE if the required cell is a not finalized undefined size one.

3.16.6 The adl_adFinalise function

This function set the provided A&D cell in read-only (finalized) mode. The cell content can not be modified anymore.

- **Prototype**

```
s32  adl_adFinalise ( u32  Handle );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed ;
  - ADL_RET_ERR_BAD_STATE if the cell was already finalized.

3.16.7 The adl_adDelete function

This function deletes the provided A&D cell. The used space and the ID will be available on next re-compaction process.

- **Prototype**

```
s32  adl_adDelete   ( u32  Handle );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed.

Note: calling **`adl_adDelete`** will unsubscribe the allocated handle.

3.16.8 The adl_adInstall function

This function installs the content of the requested cell, if it is a **`.DWL`** file. This file may be an Open AT®application, an EEPROM configuration file, an XModem downloader binary file, or a Wavecom Core software binary file.
WARNING: This API resets the product on success.

- **Prototype**

```
s32  adl_adInstall  ( u32  Handle );
```

- **Parameters**

  Handle:
  A&D cell handle returned by `adl_adSubscribe` function.

- **Returned values**
  - Product resets on success ; the parameter of the `adl_main` function is then set to ADL_INIT_DOWNLOAD_SUCCESS, or
  - ADL_INIT_DOWNLOAD_ERROR, according to the **`.DWL`** file update success or not.
  - ADL_RET_ERR_BAD_STATE if the cell is not finalized ;
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed.

3.16.9 The adl_adRecompact function

This function starts the re-compaction process, which will release the deleted cells spaces and IDs. The process is also launched as soon as deleted memory space exceeds 50% of the total A&D volume memory space.

- **Prototype**

```
s32 adl_adRecompact ( adl_adRecompactHdlr_f Handler );
```

- **Parameters**

  Handler:
  Re-compaction handler, which be called at the end of the process. The handler is based on the following type:

```
typedef void ( * adl_adRecompactHdlr_f ) ( void );
```

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_BAD_STATE if the re-compaction process is already running, or if there is a not finalized undefined size cell in the A&D volume ;
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product.

3.16.10 The adl_adGetState function

This function provides an information structure on the current A&D volume state.

- **Prototype**

```
s32 adl_adGetState ( adl_adState_t * State );
```

- **Parameters**

  State:
  A&D volume information structure, based on following type:

```
typedef struct
{
    u32 freemem;        // Space free memory size
    u32 deletedmem;     // Deleted memory size
    u32 totalmem;       // Total memory
    u16 numobjects;     // Number of allocated objects
    u16 numdeleted;     // Number of deleted objects
    u8  pad;            // not used
} adl_adState_t;
```

- **Returned values**
  - OK on success ;
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product
  - ADL_RET_ERR_PARAM on parameter error.

3.16.11 The adl_adGetCellList function

This function provides the list of the current allocated cells.

- **Prototype**

```
s32  adl_adGetCellList ( wm_lst_t * CellList );
```

- **Parameters**

  CellList:
  Return allocated cell list. The list elements are the cell identifiers and are based on u32 type.

  WARNING: the list used memory is allocated by the `adl_adGetCellList` function and must be released by the application.

- **Returned values**
  - OK on success ;
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product ;
  - ADL_RET_ERR_PARAM on parameter error.

3.16.12 The adl_adFormat function

This function allows to completely re-initialize the A&D storage volume. It is allowed only if there is currently no subscribed cells.

Important warning:
All the A&D storage cells will be erased by this operation. The A&D storage format process should take up to several seconds.

- **Prototype**

```
s32  adl_adFormat  ( void );
```

- **Returned values**
  - OK on success ;
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product ;
  - ADL_RET_ERR_BAD_STATE if there is at least one currently subscribed cell.

- **Note**

  If a re-compaction process was running, this one will be stopped, and the provided handler will not be notified.

3.17 GPS Service

ADL applications may use this service to access to the GPS device information on Q2501 products.
Note: the product uses the module's second UART to access to the GPS component. This will lock some GPIOs, which will not be available for allocation by the application ; please refer to §2.5 for more information.

3.17.1 Required Header File

The header file for the GPS service is:

`adl_gps.h`

3.17.2 GPS Data structures

3.17.2.1 Position

GPS Position data are stored in the following structure:

```
typedef struct
{
    ascii UTC_time [_S_UTC_TIME];            // hhmmss.sss
    ascii date [_S_DATE];                    // ddmmyy
    ascii latitude [_S_POSITION];            // ddmm.mmmm
    ascii latitude_Indicator[_S_INDICATOR];  // N - S
    ascii longitude [_S_POSITION];           // dddmm.mmmm
    ascii longitude_Indicator[_S_INDICATOR]; // E - W
    ascii status[_S_INDICATOR];
    ascii P_Fix[_S_INDICATOR];
    ascii sat_used [_S_SAT];                 // Satellites used
    ascii HDOP [_S_HDOP];                    // Horizontal Dilution of
                                                Precision
    ascii altitude [_S_ALTITUDE];            // MSL Altitude
    ascii altitude_Unit[_S_INDICATOR];
    ascii geoid_Sep [_S_GEOID_SEP];          // geoid correction
    ascii geoid_Sep_Unit[_S_INDICATOR];
    ascii Age_Dif_Cor [_S_AGE_DIF_COR];      // Age of Differential
                                                correction
    ascii Dif_Ref_ID [_S_DIF_REF_ID];        // Diff Ref station ID
    ascii magneticVariation[_S_COURSE];      // magnetic variation: not
                                                available for sirf
                                                technology

} adl_gpsPosition_t;
```

All fields are ascii zero terminated strings containing GPS information.

3.17.2.2 Speed

GPS Speed data are stored in the following structure:

```
typedef struct
{
    ascii course [_S_COURSE];          // Degrees from true North
    ascii speed_knots [_S_SPEED];      // Speed in knots
    ascii speed_km_p_hour [_S_SPEED];  // Speed in km/h
} adl_gpsSpeed_t;
```

All fields are ascii zero terminated strings containing GPS information.

3.17.2.3 Satellite View

GPS satellite view data are stored in the following structure:

```
typedef struct
{
    u8 id;          // range 1 to 32
    u8 elevation;   // maximum 90
    u32 azimuth;    // range 0 to 359
    s8 SNR ;        // range 0 to 99, -1 when not tracking
} adl_gpsSatellite_t;
```

All fields are integers containing GPS information about current satellite.

```
typedef struct
{
    u8 NB_Msg ;                           // Number of messages
    u8 MSG_Number ;                       // Message Number
    u8 Sat_view ;                         // Satellites in view
    adl_gpsSatellite_t sat [_NB_SAT_MAX]; // array for informations
                                             about differents
                                             satellites
} adl_gpsSatView_t;
```

The different fields contain information about the current satellite view. Each satellite information details are contained in the “sat” field.

3.17.3 The adl_gpsSubscribe function

This function subscribes to the GPS service in order to receive GPS related events.

- **Prototype**

```
s8      adl_gpsSubscribe ( adl_gpsHdlr_f    GpsHandler
                           u32              PollingTime );
```

- **Parameters**

GpsHandler:
GPS events handler defined using the following type:

```
typedef bool (*adl_gpsHdlr_f) ( adl_gpsEvent_e Event,
                                adl_gpsData_t* GpsData );
```

The events received by this handler are defined below:

ADL_GPS_EVENT_RESETING_HARDWARE

If the ADL GPS service needs to reset the product, in order to enable the GPS device internal mode. The handler may refuse this reset by returning FALSE. If at least one handler refuses the reset, the service goes to ADL_GPS_STATE_EXT_MODE state.
*The **GpsData** parameter is set to NULL.*

ADL_GPS_EVENT_EXT_MODE

*If the at least one Handler refused the ADL_GPS_EVENT_RESETING_HARDWARE event, the service entered in ADL_GPS_STATE_EXT_MODE state, and will be available on next product reset. The **GpsData** parameter is set to NULL. Handler's returned value is not relevant.*

ADL_GPS_EVENT_IDLE

*If the service entered the ADL_GPS_STATE_IDLE state: the service is ready to read GPS data. The **GpsData** parameter is set to NULL. Handler's returned value is not relevant.*

ADL_GPS_EVENT_POLLING_DATA

*If a Polling Time was required on subscription. The **GpsData** contains all GPS data read from the GPS device. Handler's returned value is not relevant.*

The GpsData parameter is based on the following type:

```
typedef struct
{
    adl_gpsPosition_t   Position;   // Current GPS position
    adl_gpsSpeed_t      Speed;      // Current GPS speed
    adl_gpsSatView_t    SatView;    // Current GPS satellite view
} adl_gpsData_t;
```

Position:
Current GPS position data ; please refer to GPS service data structures in § 3.17.2

Speed:
Current GPS speed data ; please refer to GPS service data structures in § 3.17.2

SatView:
Current GPS satellite view data ; please refer to GPS service data structures in § 3.17.2

PollingTime:
Time interval (in seconds) between each GPS data polling event (ADL_GPS_EVENT_POLLING_DATA) reception by the GPS handler.

- **Returned values**
  - This function returns a positive or null handle on success ;
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_NO_MORE_HANDLES if there is no more free handles,
  - ADL_GPS_RET_ERR_NO_Q25_PRODUCT if the current product is not a Q2501 one.

3.17.4 The adl_gpsUnsubscribe function

This function un-subscribes from the GPS service. The corresponding GPS handler will not receive any GPS events any more.

- **Prototype**

```
s8 adl_gpsUnsubscribe ( u8 Handle );
```

- **Parameters**

  Handle:
  The handle returned by the `adl_gpsSubscribe` function.

- **Returned values**
  - This function returns 0 on success,
  - ADL_RET_ERR_NOT_SUBSCRIBED if the GPS service was not subscribed,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is not a valid one,
  - ADL_RET_ERR_BAD_STATE if the service is in INIT state.

3.17.5 The adl_gpsGetState function

This function returns the current GPS service state.

- **Prototype**

```
adl_gpsState_e adl_gpsGetState ( void );
```

- **Returned values**

  The current GPS service state, based on following type:

```
typedef enum
{
   ADL_GPS_STATE_INIT, // Service initialization state
   ADL_GPS_STATE_NO_Q25, // Not a Q25 product
   ADL_GPS_STATE_RESETING_HARDWARE, // Trying to reset product after
                                    have set the GPS internal mode
   ADL_GPS_STATE_EXT_MODE, // Reset refused: will be on internal mode
                           on next product start-up
   ADL_GPS_STATE_IDLE // GPS driver in IDLE mode, ready to read data
} adl_gpsState_e;
```

3.17.6 The adl_gpsGetPosition function

This function gets the current position read from the GPS device.

- **Prototype**

```
s8 adl_gpsGetPosition ( u8 Handle, adl_gpsPosition_t * Position );
```

- **Parameters**

  Handle:
  The handle returned by the `adl_gpsSubscribe` function.

  Position:
  Position data read from the GPS device. please refer to GPS service data structures in § 3.17.2

- **Returned values**
  - This function returns OK on success.
  - ADL_RET_ERR_NOT_SUBSCRIBED if the GPS service was not subscribed,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is not a valid one,
  - ADL_RET_ERR_BAD_STATE if the GPS service is out of IDLE state.

3.17.7 The adl_gpsGetSpeed function

This function gets the current speed read from the GPS device.

- **Prototype**

```
s8 adl_gpsGetSpeed ( u8 Handle, adl_gpsSpeed_t * Speed );
```

- **Parameters**

  Handle:
  The handle returned by the `adl_gpsSubscribe` function.

  Speed:
  Speed data read from the GPS device. please refer to GPS service data structures in § 3.17.2

- **Returned values**
  - This function returns OK on success.
  - ADL_RET_ERR_NOT_SUBSCRIBED if the GPS service was not subscribed,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is not a valid one,
  - ADL_RET_ERR_BAD_STATE if the GPS service is out of IDLE state.

3.17.8 The adl_gpsGetSatView function

This function gets the current satellite view read from the GPS device.

- **Prototype**

```
s8 adl_gpsGetSatView ( u8 Handle, adl_gpsSatView_t * SatView );
```

- **Parameters**

  Handle:
  The handle returned by the `adl_gpsSubscribe` function.

  SatView:
  SatView data read from the GPS device. please refer to GPS service data structures in § 3.17.2

- **Returned values**
  - This function returns OK on success.
  - ADL_RET_ERR_NOT_SUBSCRIBED if the GPS service was not subscribed,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is not a valid one,
  - ADL_RET_ERR_BAD_STATE if the GPS service is out of IDLE state.

4 Error codes

4.1 General error codes

Error code	Error value	Description
OK	0	No error response
ERROR	-1	general error code
ADL_RET_ERR_PARAM	-2	parameter error
ADL_RET_ERR_UNKNOWN_HDL	-3	unknown handler / handle error
ADL_RET_ERR_ALREADY_SUBSCRIBED	-4	service already subscribed
ADL_RET_ERR_NOT_SUBSCRIBED	-5	service not subscribed
ADL_RET_ERR_FATAL	-6	fatal error
ADL_RET_ERR_BAD_HDL	-7	Bad handle
ADL_RET_ERR_BAD_STATE	-8	Bad state
ADL_RET_ERR_PIN_KO	-9	Bad PIN state
ADL_RET_ERR_NO_MORE_HANDLES	-10	The service subscription maximum capacity is reached
ADL_RET_ERR_SPECIFIC_BASE	-20	Beginning of specific errors range

4.2 Specific FCM service error codes

Error code	Error value
ADL_FCM_RET_ERROR_GSM_GPRS_ALREADY_OPENNED	ADL_RET_ERR_SPECIFIC_BASE
ADL_FCM_RET_ERR_WAIT_RESUME	ADL_RET_ERR_SPECIFIC_BASE-1
ADL_FCM_RET_OK_WAIT_RESUME	OK+1
ADL_FCM_RET_BUFFER_EMPTY	OK+2
ADL_FCM_RET_BUFFER_NOT_EMPTY	OK+3

4.3 Specific flash service error codes

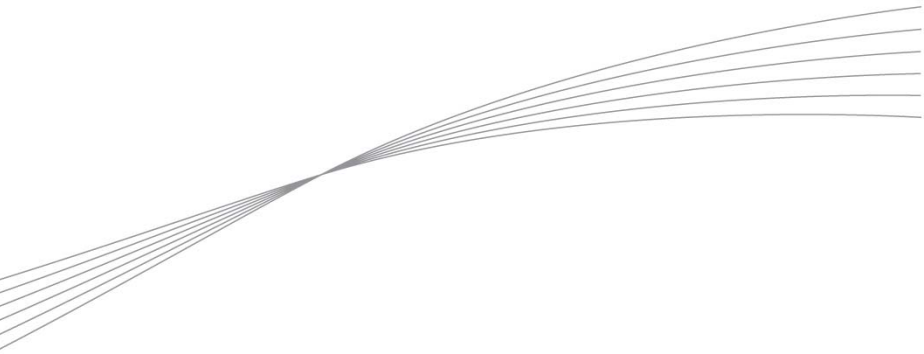
Error code	Error value
ADL_FLH_RET_ERR_OBJ_NOT_EXIST	ADL_RET_ERR_SPECIFIC_BASE
ADL_FLH_RET_ERR_MEM_FULL	ADL_RET_ERR_SPECIFIC_BASE-1
ADL_FLH_RET_ERR_NO_ENOUGH_IDS	ADL_RET_ERR_SPECIFIC_BASE-2
ADL_FLH_RET_ERR_ID_OUT_OF_RANGE	ADL_RET_ERR_SPECIFIC_BASE-3

4.4 Specific GPRS service error codes

Error code	Error value
ADL_GPRS_CID_NOT_DEFINED	-3
ADL_NO_GPRS_SERVICE	-4
ADL_CID_NOT_EXIST	5

4.5 Specific GPS service error codes

Error code	Error value
ADL_GPS_RET_ERR_NO_Q25_PRODUCT	ADL_RET_ERR_SPECIFIC_BASE



wavecom

Make it wireless

WAVECOM S.A. - 3 esplanade du Foncet - 92442 Issy-les-Moulineaux Cedex - France - Tel: +33(0)1 46 29 08 00 - Fax: +33(0)1 46 29 08 08
Wavecom, Inc. - 4810 Eastgate Mall - Second Floor - San Diego, CA 92121 - USA - Tel: +1 858 362 0101 - Fax: +1 858 558 5485
WAVECOM Asia Pacific Ltd. - 4/F, Shui On Centre - 6/8 Harbour Road - Hong Kong - Tel: +852 2824 0254 - Fax: +852 2824 025