ADL User Guide for Open AT® V4.10

Revision: 002
Date: September 2006

wavecom
Make it wireless

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

ADL User Guide for Open AT® V4.10

Revision: **002**
Date: **September 11, 2006**
Reference: **WM_DEV_OAT_UGD_019**

Document History

Index	Date	Versions	
001	May, 2006	Creation	
002	September 2006	Creation	

Copyright

Trademarks

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® 4.10 and higher (until next version of this document).

Table of Contents

DOCUMENT HISTORY 2

COPYRIGHT 3

TRADEMARKS 3

OVERVIEW 4

TABLE OF CONTENTS 5

LIST OF FIGURES 12

1 INTRODUCTION 13
1.1 Important Remarks 13
1.2 References 13
1.3 Glossary 13
1.4 Abbreviations 14

2 DESCRIPTION 15
2.1 Software Architecture 15
2.2 Imported APIs from Open AT® library 16
2.3 ADL Limitations 16
2.4 Open AT® Memory Resources 16
2.5 Defined Compilation Flags 17
2.6 Inner AT Commands Configuration 18
2.7 Open AT® Specific AT Commands 19
2.7.1 AT+WDWL Command 19
2.7.2 AT+WOPEN Command 19
2.8 Notes on Wavecom OS 20
2.9 Security 21
2.9.1 Software Security: Memory Access Protection 21
2.9.2 Hardware Security: Watchdog Protection 21
2.9.3 Safe Boot Mode 21
2.10 RTE limitations 22
2.10.1 Sending large buffers through an ADL API 22
2.10.2 IRQ services 22

3 API 23

3.1 Mandatory Application Code 23
3.1.1 Required Header File 23
3.1.2 Call Stack Sizes 23
3.1.3 The adl_main function 25

3.2 AT Commands Service 26
3.2.1 Required Header File 26
3.2.2 Unsolicited Responses 26
3.2.3 Responses 28
3.2.4 Incoming AT Commands 31
3.2.5 Run AT Commands 35

3.3 Timers 42
3.3.1 Required Header Files 42
3.3.2 The adl_tmrSubscribe Function 42
3.3.3 The adl_tmrUnSubscribe Function 43
3.3.4 Example 44

3.4 Memory Service 45
3.4.1 Required Header File 45
3.4.2 The adl_memGetInfo Function 45
3.4.3 The adl_memGet Function 46
3.4.4 The adl_memRelease Function 47
3.4.5 Heap Memory Block Status 47
3.4.6 Example 48

3.5 Debug Traces 48
3.5.1 Required Header File 48
3.5.2 Debug Configuration 48
3.5.3 Full Debug Configuration 50
3.5.4 Release Configuration 50

3.6 Flash 51
3.6.1 Required Header File 51
3.6.2 Flash Objects Management 51
3.6.3 The adl_flhSubscribe Function 52
3.6.4 The adl_flhExist Function 53
3.6.5 The adl_flhErase Function 53
3.6.6 The adl_fhWrite Function 54
3.6.7 The adl_flhRead Function 55
3.6.8 The adl_flhGetFreeMem Function 55
3.6.9 The adl_flhGetIDCount Function 56
3.6.10 The adl_flhGetUsedSize Function 56

3.7 FCM Service 57
3.7.1 Required Header File 58
3.7.2 The adl_fcmIsAvailable Function 58
3.7.3 The adl_fcmSubscribe Function 59
3.7.4 The adl_fcmUnsubscribe Function 63
3.7.5 The adl_fcmReleaseCredits Function 63
3.7.6 The adl_fcmSwitchV24State Function 64
3.7.7 The adl_fcmSendData Function 64
3.7.8 The adl_fcmSendDataExt Function 66
3.7.9 The adl_fcmGetStatus Function 67

3.8 GPIO Service 68
3.8.1 Required Header File....68
3.8.2 GPIO Types....68
3.8.3 The adl_ioEventSubscribe Function....74
3.8.4 The adl_ioHdlr_f Call-back Type 74
3.8.5 The adl_ioEventUnsubscribe Function 75
3.8.6 The adl_ioSubscribe Function 76
3.8.7 The adl_ioUnsubscribe Function 77
3.8.8 The adl_ioSetDirection Function....77
3.8.9 The adl_ioRead Function....78
3.8.10 The adl_ioReadSingle Function 78
3.8.11 The adl_ioWrite Function 79
3.8.12 The adl_ioWriteSingle Function....80
3.8.13 The adl_io GetProductType Function....80
3.8.14 The adl_ioGetFeatureGPIOList Function 80
3.8.15 The adl_ioIsFeatureEnabled Function 81
3.8.16 Example....82

3.9 Bus Service 84
3.9.1 Required Header File....84
3.9.2 Bus Types 84
3.9.3 The adl_busSubscribe Function 94
3.9.4 The adl_busUnsubscribe Function 96
3.9.5 The adl_busRead Function....97
3.9.6 The adl_busWrite Function 97
3.9.7 The adl_busDirectRead Function....98
3.9.8 The adl_busDirectWrite Function 99
3.9.9 Example....100

3.10 Error Management 103
3.10.1 Required Header File....103
3.10.2 The adl_errSubscribe Function....103
3.10.3 The adl_errUnsubscribe Function....104
3.10.4 The adl_errHalt Function....105
3.10.5 The adl_errEraseAllBacktraces Function....105
3.10.6 The adl_errStartBacktraceAnalysis Function 105
3.10.7 The adl_errGetAnalysisState Function....106
3.10.8 The adl_errRetrieveNextBacktrace Function 107

3.11 SIM Service 109
3.11.1 Required Header File....109
3.11.2 The adl_simSubscribe Function 109
3.11.3 The adl_simUnsubscribe Function 110
3.11.4 The adl_simGetState Function 111

3.12 Open SIM Access Service....112
3.12.1 Required Header File....112
3.12.2 The adl_osaSubscribe Function 112
3.12.3 The adl_osaHandler_f call-back Type....113
3.12.4 The adl_osaSendResponse Function 115
3.12.5 The adl_osaUnsubscribe Function 116
3.12.6 Example....118

3.13 SMS Service....119
3.13.1 Required Header File....119

3.13.2 The adl_smsSubscribe Function 119
3.13.3 The adl_smsSend Function 120
3.13.4 The adl_smsUnsubscribe Function 121

3.14 Message Service 122
3.14.1 Required Header File....122
3.14.2 The adl_msgMailBox_e Type....122
3.14.3 The adl_msgIdComparator_e Type 122
3.14.4 The adl_msgFilter_t Structure 123
3.14.5 The adl_msgSubscribe Function 124
3.14.6 The adl_msgHandler_f call-back Type 125
3.14.7 The adl_msgUnsubscribe Function 126
3.14.8 The adl_msgSend Function....126
3.14.9 Example....128

3.15 Call Service 129
3.15.1 Required Header File....129
3.15.2 The adl_callSubscribe Function....129
3.15.3 The adl_callSetup Function 132
3.15.4 The adl_callSetupExt Function 132
3.15.5 The adl_callHangup Function....133
3.15.6 The adl_callHangupExt Function....133
3.15.7 The adl_callAnswer Function 134
3.15.8 The adl_callAnswerExt Function 134
3.15.9 The adl_callUnsubscribe Function....134

3.16 GPRS Service 136
3.16.1 Required Header File....136
3.16.2 The adl_gprsSubscribe Function 136
3.16.3 The adl_gprsSetup Function 139
3.16.4 The adl_gprsSetupExt Function 140
3.16.5 The adl_gprsAct Function 141
3.16.6 The adl_gprsActExt Function 141
3.16.7 The adl_gprsDeact Function 142
3.16.8 The adl_gprsDeactExt Function....143
3.16.9 The adl_gprsGetCidInformations Function 144
3.16.10 The adl_gprsUnsubscribe Function 145
3.16.11 The adl_gprsIsAnIPAddress Function....145
3.16.12 Example....146

3.17 Semaphore ADL Service 148
3.17.1 Required Header File....148
3.17.2 The adl_semSubscribe Function 148
3.17.3 The adl_semConsume Function 149
3.17.4 The adl_semConsumeDelay Function....149
3.17.5 The adl_semProduce Function 150
3.17.6 The adl_semUnsubscribe Function 150
3.17.7 Example....152

3.18 Application Safe Mode Service 153
3.18.1 Required Header File....153
3.18.2 The adl_safeSubscribe Function....153
3.18.3 The adl_safeUnsubscribe Function....154
3.18.4 The adl_safeRunCommand Function....155

3.19 AT Strings Service 156
3.19.1 Required Header File 156
3.19.2 The adl_strID_e Type 156
3.19.3 The adl_strGetID Function 157
3.19.4 The adl_strGetIDExt Function 157
3.19.5 The adl_strIsTerminalResponse Function 158
3.19.6 The adl_strGetResponse Function 158
3.19.7 The adl_strGetResponseExt Function 159
3.20 Application & Data Storage Service 160
3.20.1 Required Header File 160
3.20.2 The adl_adSubscribe Function 160
3.20.3 The adl_adUnsubscribe Function 161
3.20.4 The adl_adEventSubscribe Function 162
3.20.5 The adl_adEventHdlr_f Call-back Type 162
3.20.6 The adl_adEventUnsubscribe Function 164
3.20.7 The adl_adWrite Function 164
3.20.8 The adl_adInfo Function 165
3.20.9 The adl_adFinalise Function 165
3.20.10 The adl_adDelete Function 166
3.20.11 The adl_adInstall Function 166
3.20.12 The adl_adRecompact Function 168
3.20.13 The adl_adGetState Function 169
3.20.14 The adl_adGetCellList Function 170
3.20.15 The adl_adFormat Function 171
3.20.16 Example 172
3.21 AT/FCM IO Ports Service 173
3.21.1 Required Header File 173
3.21.2 AT/FCM IO Ports 173
3.21.3 Ports Test Macros 175
3.21.4 The adl_portSubscribe Function 176
3.21.5 The adl_portUnsubscribe Function 177
3.21.6 The adl_portIsAvailable Function 177
3.21.7 The adl_portGetSignalState Function 178
3.21.8 The adl_portStartSignalPolling Function 178
3.21.9 The adl_portStopSignalPolling Function 180
3.22 RTC Service 181
3.22.1 Required Header File 181
3.22.2 RTC service Types 181
3.22.3 The adl_rtcGetTime Function 183
3.22.4 The adl_rtcConvertTime Function 183
3.22.5 The adl_rtcDiffTime Function 184
3.22.6 Example 184
3.23 IRQ Service 185
3.23.1 Required Header File 185
3.23.2 The adl_irqID_e Type 185
3.23.3 The adl_irqNotificationLevel_e Type 186
3.23.4 The adl_irqPriorityLevel_e Type 186
3.23.5 The adl_irqEventData_t Structure 187
3.23.6 The adl_irqSubscribe Function 187
3.23.7 The adl_irqHandler_f Call-back Type 190

3.23.8 The adl_irqUnsubscribe Function 191
3.23.9 Example 192
3.24 SCTU Service 193
3.24.1 Required Header File 194
3.24.2 The adl_sctuInfo_t Structure 194
3.24.3 The adl_sctuSubscribe Function 195
3.24.4 The adl_sctuSetChannelConfig Function 196
3.24.5 The adl_sctuStart Function 197
3.24.6 The adl_sctuRead Function 198
3.24.7 The adl_sctuStop Function 198
3.24.8 The adl_sctuUnsubscribe Function 198
3.24.9 Example 199
3.25 Extint ADL Service 201
3.25.1 Required Header File 202
3.25.2 The adl_extintSettings_t Structure 202
3.25.3 The adl_extintInfo_t Structure 203
3.25.4 The adl_extintSubscribe Function 203
3.25.5 The adl_extintConfig Function 204
3.25.6 The adl_extintRead function 205
3.25.7 The adl_extintUnsubscribe Function 205
3.25.8 Example 207
3.26 Execution Context Service 208
3.26.1 Required Header File 209
3.26.2 The adl_ctxID_e Type 209
3.26.3 The adl_ctxGetID Function 209
3.26.4 The adl_ctxGetTaskID Function 209
3.26.5 The adl_ctxGetDiagnostic Function 210
3.26.6 The adl_ctxSuspend Function 210
3.26.7 The adl_ctxResume Function 211
3.26.8 Example 212
3.27 ADL VariSpeed Service 213
3.27.1 Required Header File 213
3.27.2 The adl_vsMode_e Type 213
3.27.3 The adl_vsSubscribe Function 214
3.27.4 The adl_vsSetClockMode Function 214
3.27.5 The adl_vsUnsubscribe function 215
3.27.6 Example 215
3.28 ADL DAC Service 216
3.28.1 Required Header File 216
3.28.2 The adl_dacSubscribe Function 216
3.28.3 The adl_dacUnsubscribe Function 217
3.28.4 The adl_dacWrite Function 217
3.28.5 Example 218

4 ERROR CODES 219

4.1 General Error Codes 219
4.2 Specific FCM Service Error Codes 220
4.3 Specific Flash Service Error Codes 220

4.4 Specific GPRS Service Error Codes 220
4.5 Specific A&D Storage Service Error Codes.....220

List of Figures

Figure 1: General software architecture 15
Figure 2: Error when trying to send too large a data buffer through an API 22
Figure 3: Open AT® RAM Mapping (Call Stack Space) 24
Figure 4: ADL AT Command Stacks Architecture 41
Figure 5: Open AT® RAM Mapping, with `adl_memInfo_t` Structure Field Names 46
Figure 6: Flow Control Manager Representation 57
Figure 7 Motorola Modes Timing Example 88
Figure 8: Intel Mode Timing - Read Process Example 89
Figure 9: Intel Mode Timing - Write Process Example 89
Figure 10: A&D cell content install window 167
Figure 11: Q268X Wireless CPUs SCTU Architecture 193
Figure 12: ADL External Interruption Service: Example of Interruption with Debounce Period 201
Figure 13:ADL External Interruption Service: Example of Interruption with Stretching Process 201

1 Introduction

1.1 Important Remarks

- It is strongly recommended before reading this document, to read the ADL User Guide Open AT® 4.10 and specifically the Introduction (chapter 1) 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

[1] Basic Development Guide for Open AT® 4.10, (ref. WM_DEV_OAT_UGD_017)
[2] AT commands Interface Guide for OS 6.61 (ref WM_DEV_OAT_UGD_014)
[3] Tools Manual for Open AT® IDE 1.00 (ref. WM_DEV_OAT_UGD_018)

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 OS	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 OS functions.
Wavecom OS	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
APN	Access Point Name
CID	Context IDentifier
CPU	Central Processing Unit
DAC	Digital Analog Converter
EXTINT	External Interruption
FCM	Flow Control Manager
GPIO	General Purpose Input Output
GGSN	Gateway GPRS Support Node
GPRS	General Packet Radio Service
IP	Internet Protocol
IR	Infrared
KB	Kilobyte
MS	Mobile Station
OS	Operating System
PDP	Packet Data Protocol
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

2 Description

2.1 Software Architecture

The Application Development Layer library provides a high level interface for the Open AT® software developer. The ADL set of services has to be used to access all the Wavecom Wireless CPUs capabilities & interfaces.

The Open AT® environment relies on the following software architecture:

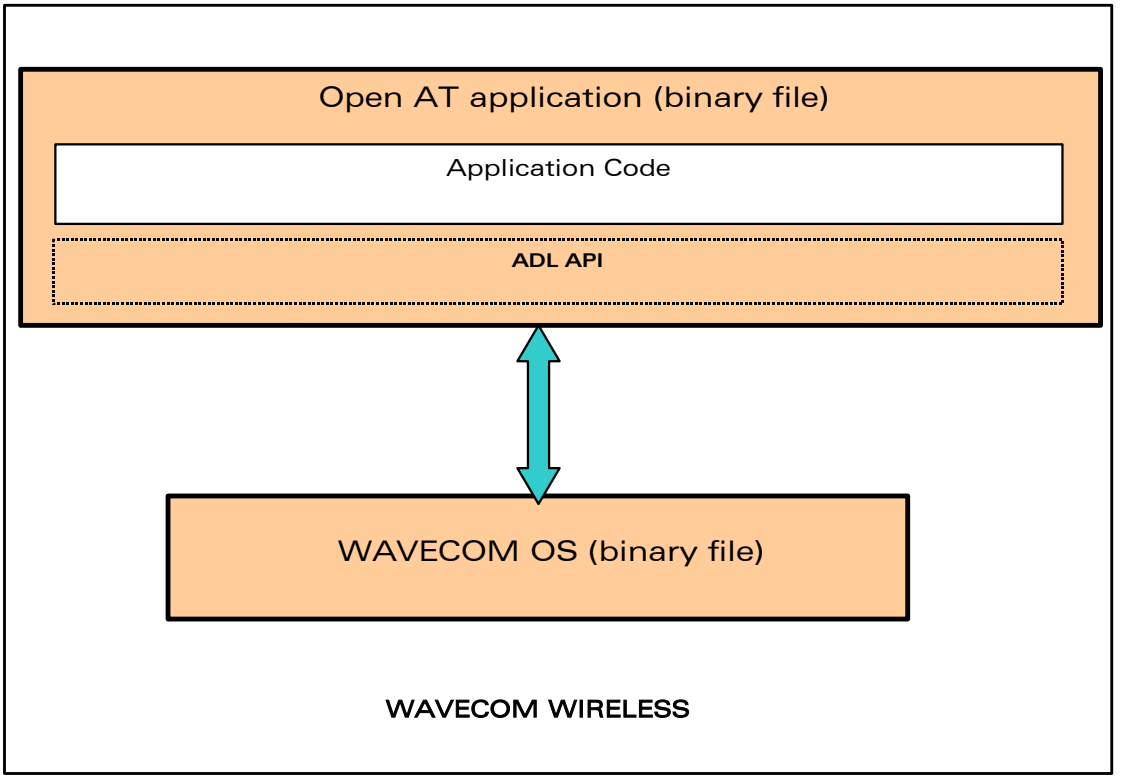


Figure 1: General software architecture

The different software elements on a Wavecom product are described in this section.
The **Open AT® application**,which includes the following items:

- the application code,
- as an option (according to the application needs), one or several Open AT® plug-in libraries (such as the IP connectivity library),
- the Wavecom Application Development Layer library, which provides all the services used by the application
- the **Wavecom OS**, which manages the Wavecom Wireless CPU.

2.2 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® ADL User Guide (document [1]) for these APIs description.

2.3 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.

2.4 Open AT® Memory Resources

The available memory resources for the Open AT® applications are listed below.

OS 6.61 Q2686 Wireless CPU H type memory
256-1600 Kbytes of ROM (application code) (default: 832 Kbytes) (for configuration, see AT+WOPEN command)
256 Kytes of RAM
128 Kbytes of Flash Object Data
0-1344 Kbytes of Application & Data Storage Volume (default: 768 Kbytes) (for configuration, see AT+WOPEN command)

The total available flash space for both Open AT® application place and A&D storage place is 1600 Kbytes.

The maximum A&D storage place size is 1344 Kbytes (about 1.3 Mbytes: usable for Wavecom OS upgrade capability). In this case the Open AT® application maximum size will be 256 Kbytes.

The minimum A&D storage place size is 0 Kbytes (usable for applications with huge hard coded data). In this case the Open AT® application maximum size will be about 1.5 Mbytes.

Caution:

Any A&D size change will lead to this area format process (some seconds on start-up; all A&D cells data will be erased).

2.5 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® V4.10 interface, it is defined as "__OAT_API_VERSION__=410".

__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.

2.6 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
AT+CRC	1
AT+CGEREP	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 caution:

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 OS without a running ADL application).

2.7 Open AT® Specific AT Commands

See document WM_DEV_OAT_UGD_014-001, AT Commands Interface Guide (document [2]).

2.7.1 AT+WDWL Command

The AT+WDWL command is usable to download .dwl files trough the serial link, using the 1K Xmodem protocol.

Dwl files may be Wavecom OS updates, Open AT® application binaries, or E2P configuration files.

By default this command is not pre-parsed (it can not be filtered by the Open AT® application), except if the Application Safe Mode service is used.

Note:
The AT+WDWL command is described in the document [2].

2.7.2 AT+WOPEN Command

The AT+WOPEN command allows to control Open AT® applications mode & parameters.

Parameters :

0 Stop the application (the application will be stopped on all product resets)

1 Start the application (the application will be started on all product resets)

2 Get the Open AT® libraries versions

3 Erase the objects flash of the Open AT® Embedded Application (allowed only if the application is stopped)

4 Erase the Open AT® Embedded Application

(allowed only if the application is stopped)

5 Suspend the Open AT® application, until the AT+WOPENRES command is used, or an hardware interruption occurs

6 Configures the Application & Data storage place and Open AT® application place sizes.

7 Requires the current Open AT® application state (e.g. to check if the application binary has correctly been built or if the application is running in Target or RTE mode).

8 Configures the Safe Boot mode (refer to §2.9 for more information).

Note:
Refer to the document [2] for more information about this command.

By default this command is not pre-parsed (it can not be filtered by the Open AT® application), except if the Application Safe Mode service is used.

2.8 Notes on Wavecom OS

The Open AT® application runs within a single task managed by the Wavecom OS: event handlers are always called sequentially by ADL (no specific re-entrancy protection has to be implemented in the application code, due to ADL architecture).

The Wavecom OS and the Open AT® application manage their own RAM area. Any access from one of these entities to the other's RAM area is prohibited and causes an exception.

Global variables, call stack and dynamic memory are all part of the RAM allocated to the Open AT® application.

2.9 Security

Security mechanisms are implemented in the Wavecom OS in order to protect the Wireless CPU against software errors. When this occurs, the Wireless CPU resets and a function call log (called "back-trace") is stored in the Wireless CPU non-volatile memory. After reset, the **adl_main** function is called with the **ADL_INIT_REBOOT_FROM_EXCEPTION** value.

After a reset caused by a software crash, the application is started only 20 seconds after the start of the Wavecom OS. This allows at least 20 seconds delay to re-download a new application, or to stop the currently running one. In case of a normal reset, the application restarts immediately.

2.9.1 Software Security: Memory Access Protection

A specific RAM area is allocated to the Open AT® application.
The Open AT® application is seen as a Real-Time task in the Wavecom OS, and each time this task runs, the Wavecom RAM protection is activated.
If the Open AT® application tries to access this RAM, then an exception occurs and the software resets.

In case of illegal RAM access, the stored back-trace will display the "ARM exception 1 xxx" statement, where "xxx" is the address that the application was attempting to access.

2.9.2 Hardware Security: Watchdog Protection

All software (both Open AT® application & Wavecom OS) is protected from reaching a dead-end lock by a 5 seconds external watchdog reset circuit.

If one task uses the CPU for more than the allowed time, the external watchdog circuit resets the Wireless CPU.
If a crash due to this watchdog protection is detected, the stored back-trace will display the "Watchdog Reset" statement.

2.9.3 Safe Boot Mode

A specific Safe Boot mode is available on the Wireless CPU.

This mode is activated when a key combination (configured through the AT + WOPEN = 8 command mode) is pressed during Wireless CPU reset.
It is useful when the embedded application causes an exception soon after the Wireless CPU resets, without any possibility for the external application to send any AT command to disable the Open AT command.

2.10 RTE limitations

2.10.1 Sending large buffers through an ADL API

Large data buffers (greater than 1600 data bytes) cannot be sent through an ADL API (Eg. adl_busWrite) in RTE mode. If the application tries to do so, an error message (see Figure 2: Error when trying to send too large a data buffer through an API) will be displayed, and the RTE application will stop with an error.

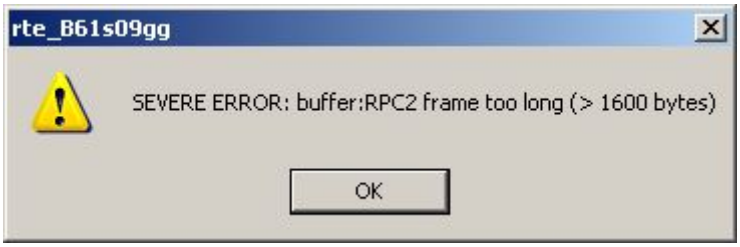


Figure 2: Error when trying to send too large a data buffer through an API

2.10.2 IRQ services

Due to the RTE architecture and to the very low latency & processing times required in IRQ based applications the IRQ service & all the related services (such as SCTU, ExtInt services, etc..) are not available in this mode. The subscription function will always fail when called in RTE.

3 API

3.1 Mandatory Application Code

3.1.1 Required Header File

Mandatory application API header file is:
`adl_AppliInit.h`
(This file is already included by adl_global.h)

3.1.2 Call Stack Sizes

This constant below must be defined by every Open AT® application, in order to provide the Wavecom OS with the required Open AT® task call stack size.

```
const u16 wm_apmCustomStackSize = 1024;   // The 1024 value is an example
```

If the application whishes to handle interruptions (cf. IRQ service chapter & Execution context service chapter (see § 3.25), it has also to define the required contexts (low level and/or high level) call stack sizes.

The constant used for low level interruption handlers execution context is the one below:

```
const u16 wm_apmIRQLowLevelStackSize = 1024;  // Example value
```

The constant used for high level interruption handlers execution context is the one below:

```
const u16 wm_apmIRQHighLevelStackSize = 1024;  // Example value
```

Please note that these definitions are optional: if the application either does not supply one or both of these call stack sizes, or set them to 0, the associated context(s) will not be available at runtime.

- **Reminder:**

  A call stack is the Open AT® RAM area which contains the local variables and return addresses for function calls. Call stack sizes are dedued from the total available RAM size for Open AT® application.

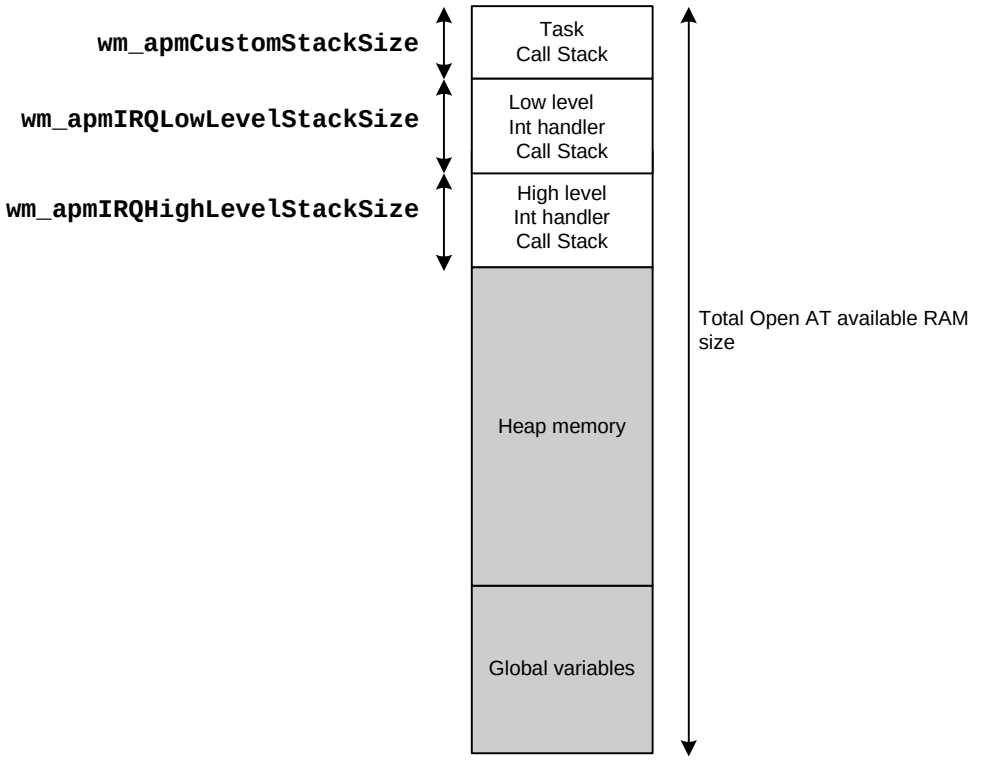


Figure 3: Open AT® RAM Mapping (Call Stack Space)

Note 1:
Previous Open AT® versions required the **wm_apmCustomStack** constant definition. This constant is now completely internally processed by the Wavecom OS, and its definition has to be removed from the Open AT® application code.

Note 2:
In RTE mode, the call stacks are processed by the host's operating system, and are not configurables (declared sizes are just removed from the available RAM space for the heap memory). It also means that stack overflows cannot be debugged within the RTE mode.

Note 3:
The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than the ARM compiler. If the GCC compiler is used, the Open AT® application is declared with greater stack sizes. An automatic multiplier tool is integrated in the generation tools, which applies a factor to stack sizes declared in the Open AT® application source file. By default this factor is set to 3 (eg. If the application source code requires a 1 kilobyte stack size, the effective size with GCC compiler will be 3 kilobytes). It can be modified (and even set to 0) by the **-gssf** option of the **wmnew** script.

3.1.3 The adl_main function

This function must be defined by each Open AT® application, in order to provide the Wavecom OS with the required Open AT® application entry point. Once the application is started (using the **AT+WOPEN=1** command), it is called by the Wavecom OS each time the Wireless CPU is powered-on, and after each hardware or software reset.

- **Prototype**

  ```
  void adl_main ( adl_InitType_e InitType )
  ```

- **Parameters**

  InitType:
  Wireless CPU power-on or reset reason; based on the following type:

  ```
  typedef enum
  {
        ADL_INIT_POWER_ON,              // Normal power on
        ADL_INIT_REBOOT_FROM_EXCEPTION, // Reboot after an 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;
  ```

Important note:
The **adl_main** function is NOT like a standard "C" main function. The application does not end when **adl_main** returns. An Open AT® application is stopped only if the "AT+WOPEN=0" command is used. The **adl_main** function is only the application entry point, and has to subscribe to some services and events to go further. In addition the whole software is protected by a watchdog mechanism, the application shall not use infinite loops and loops having a too long duration, the Wireless CPU will reset due to the watchdog hardware security (please refer to § 2.9.2 Hardware Security: Watchdog Protection for more information).

3.2 AT Commands Service

3.2.1 Required Header File

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

`adl_at.h`

3.2.2 Unsolicited Responses

An unsolicited response is a string sent by the Wavecom OS to applications in order to provide them unsolicited event information (ie. not in response to an AT command).

ADL applications may subscribe to an unsolicited response in order to receive the event in the provided handler.

Once an application has subscribed to an unsolicited response, it will have to unsubscribe from it to stop the callback function being executed every time the matching unsolicited response is sent from the Wavecom OS.

Multiple subscriptions: each unsolicited response may be subscribed several times. If an application subscribes to an unsolicited response with handler 1 and then subscribes 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.

3.2.2.1 The adl_atUnSoSubscribe Function

This function subscribes to a specific unsolicited response with an associated callback function: when the required unsolicited response is sent from the Wavecom OS, 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 § 3.19 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 (it is declared in the adl_at.h header file):

```
typedef struct
{
      adl_strID_e RspID;  // Standard response ID
      adl_atPort_e Dest;  // Unsolicited response destination port
      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 § 3.19 for more information.
The Dest field is the unsolicited response original destination port. If it is set to ADL_PORT_NONE, unsolicited response is required to be broadcasted on all ports.
The return value of the callback function will have to be TRUE if the unsolicited string is to be sent to the external application (on the port indicated by the Dest field, if not set to ADL_PORT_NONE, otherwise on all ports), 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 to be sent to the external application.

- **Returned values**
  - OK on success
  - ERROR if an error occurred.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.2.2.2 The adl_atUnSoUnSubscribe Function

This function unsubscribes from 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context)

3.2.2.3 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);
}
```

3.2.3 Responses

3.2.3.1 The adl_atSendResponse function

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

- **Prototype**

```
s32 adl_atSendResponse (  u16 Type,
                          ascii * String )
```

- **Parameters**

  Type:
  This parameter is composed of the response type, and the destination port where to send the response. The type & destination combination has to be done with the following macro :

  ```
  ADL_AT_PORT_TYPE ( _port, _type )
  ```

  The `_port` argument has to be a defined value of the `adl_atPort_e` type, and this required port has to be available (cf. the AT/FCM port Service) ; sending a response on an Open AT® the GSM or GPRS based port will have no effects).

Note:
that with the **ADL_AT_UNS** type value, if the **ADL_AT_PORT_TYPE** macro is not used, the unsolicited response will be broadcasted on all currently opened ports.
If the **ADL_AT_PORT_TYPE** macro is not used with the **ADL_AT_RSP** & **ADL_AT_INT** types, responses will be by default sent on the UART 1 port. If this port is not opened, responses will not be displayed.

The _type argument has to be one of the values defined below:

- ADL_AT_RSP :
  Terminal response (have to ends an incoming AT command).
  A destination port has to be specified.
  Sending such a response will flush all previously buffered unsolicited responses on the required port.

- ADL_AT_INT :
  Intermediate response (text to display while an incoming AT command is running).
  A destination port has to be specified.
  Sending such a response will just display the required text, without flushing all previously buffered unsolicited responses on the required port.

- ADL_AT_UNS :
  Unsolicited response (text to be displayed out of a currently running command process).
  For the required port (if any) or for each currently opened port (if the **ADL_AT_PORT_TYPE** macro is not used), if an AT command is currently running (ie. the command was sent by the external application, but this command answer has not be sent back yet), any unsolicited response will automatically be buffered, until a terminal response is sent on this port.

String:
The text to be sent.
Please note that this is exactly the text string to be displayed on the required port (ie. all carriage return & line feed characters ("\r\n" in C language) have to be sent by the application itself).

- **Returned values**

o ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
o OK if the function is successfully executed.

3.2.3.2 The adl_atSendStdResponse Function

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

- **Prototype**

```
s32 adl_atSendStdResponse (   u8 Type,
                              adl_strID_e RspID )
```

- **Parameters**

Type:
Same use as the `adl_atSendResponse` Type parameter.

RspID:
Standard response ID to be sent (see §3.19 for more information).

- **Returned value**

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
- OK if the function is successfully executed.

3.2.3.3 The adl_atSendStdResponseExt Function

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

- **Prototype**

```
s32 adl_atSendStdResponseExt (  u8 Type,
                                adl_strID_e RspID,
                                u32 arg )
```

- **Parameters**

Type:
Same use as the `adl_atSendResponse` Type parameter.

RspID:
Standard response ID to be sent (see §3.19 for more information).

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

- **Returned value**

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
- OK if the function is successfully executed.

3.2.3.4 Additional Macros for Specific Port Access

The above Response sending functions may be also used with the macros below, which provide the additional Port argument : it should avoid heavy code including each time the **ADL_AT_PORT_TYPE** macro call.

```
#define adl_atSendResponsePort(_t,_p,_r)
        adl_atSendResponse(ADL_AT_PORT_TYPE(_p,_t),_r)

#define adl_atSendStdResponsePort(_t,_p,_r)
        adl_atSendStdResponse(ADL_AT_PORT_TYPE(_p,_t),_r)

#define adl_atSendStdResponseExtPort(_t,_p,_r,_a)
        adl_atSendStdResponseExt(ADL_AT_PORT_TYPE(_p,_t),_r,_a)
```

3.2.4 Incoming AT Commands

An ADL application may subscribes to an AT command string, in order to receive events each time an external application sends this AT command on one of the Wireless CPU's ports.

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

Multiple subscriptions: if an application subscribes to a command with a handler and subscribes 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).

3.2.4.1 The adl_atCmdSubscribe Function

This function subscribes to a specific command with an associated callback function, so that next time the required command is sent by an 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. Since this service only handles AT commands, this string has to begin by the "AT" characters.

  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 (it is declared in the adl_at.h header file):

```
typedef struct
{
    u16             Type;       // Incoming Command Type
    u8              NbPara;     // Parameters number
    adl_atPort_e    Port;       // Source port
    wm_lst_t        ParaList;   // Parameters list
    u16             StrLength;  // Command string length
    ascii           StrData[1]; // Command string
} adl_atCmdPreParser_t;
```

This structure members are defined below:

- Type :
  Incoming command type (will be one of the required ones at subscription time), detected by the ADL pre-processing.

- NbPara :
  Non NULL parameters number (if Type is **ADL_CMD_TYPE_PARA**), or 0 (with other type values).

- Port :
  Port on which the command was sent by the external application.

- ParaList :
  Parameters list (if Type is ADL_CMD_TYPE_PARA).
  Each parameter may be accessed by the ADL_GET_PARAM(_p,_i) macro, where _p is the command handler parameter (adl_atCmdPreParser_t * pointer), and _i is the parameter index (from 0 to NbPara – 1). NbPara is the number of arguments received and it is a number between the minimum arguments number ('a') and the maximum arguments number ('b') (eg. a=1, b=5 and "AT+MYCMD=0,1,2", _i can be between 0 and 3 – 1 = 2).
  If a string parameter is provided (eg. AT+MYCMD="string"), the quotes will be removed from the returned string (eg. ADL_GET_PARAM(para,0) will return "string" (without quotes) in this case).
  If a parameter is not provided (eg. AT+MYCMD=1), the matching list element will be set to NULL (eg. ADL_GET_PARAM(para,0) will return NULL in this case).

- StrLength, StrData :
  Incoming command string buffer length and address. If the incoming command from the external application is containing useless spaces (" ") or semi-colon (";") characters, those will automatically be removed from the command string (e.g. if an

external application sends "AT+MY CMD;" string, the command handler will receive "AT+MYCMD").

Options:
This flag combines with a bitwise 'OR' ('|' in C language) the following information:

-

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.*
Minimum arguments number	0x000X	Minimum argument number for incoming command. Usable only if the **ADL_CMD_TYPE_PARA** type is required.
Maximum arguments number	0x00X0	Maximum argument number for incoming command. Usable only if the **ADL_CMD_TYPE_PARA** type is required.
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,however the only string "at-" is not received.*

Incoming commands which are matching with these options combinations will lead to the callback function execution. If options do not match, the command will be forwarded to be processed by the Wavecom OS.

- **Returned values**
  - OK
  - ERROR if an error occurred.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Important note about incoming concatenated command :

ADL is able to recognize and process concatenated commands coming from external applications (Please refer to AT Commands Interface Guide (document [2]) for more information on concatenated commands syntax).

In this case, this port 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, all other external command requests will be refused on this port: any external application connected on this port will receive a "+CME ERROR: 515" code if it tries to send another command. The embedded application can continue using this port for its specific processes, but it has to be careful to send one (at least one, and only one) terminal response for each subscribed 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 internal and external commands on this one.

3.2.4.2 The adl_atCmdUnSubscribe Function

This function unsubscribes from 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 from.

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

- **Returned values**
  - OK if the command was found,
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
  - ERROR otherwise.

3.2.4.3 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_atSendResponsePort(ADL_AT_RSP, paras->Port,
                              "\r\nhandling at+abc?\r\n");
   else if(paras->Type == ADL_CMD_TYPE_TEST)
       adl_atSendResponsePort(ADL_AT_RSP, paras->Port,
                              "\r\nhandling at+abc=?\r\n");
   else if(paras->Type == ADL_CMD_TYPE_ACT)
       adl_atSendResponsePort(ADL_AT_RSP, paras->Port,
                              "\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_atSendResponsePort(ADL_AT_RSP, paras->Port, buffer);
    }
    adl_atSendResponsePort(ADL_AT_RSP, paras->Port, "\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);
}
```

3.2.5 Run AT Commands

3.2.5.1 The adl_atCmdCreate Function

This function sends a command on the required port 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**

```
s8 adl_atCmdCreate (   ascii * Cmdstr,
                       u16 Rspflag,
                       adl_atRspHandler_t Rsphdl,
                       [...,]
                       NULL)
```

- **Parameters**

Cmdstr:
The string (name) of the command we want to send. If the string does not ends with the CR character ("\r" in C language), it will be added by ADL. In case of text mode commands (as +CMGW for example), text end character has to be the ^Z ("\x1A" in C language) one.

Rspflag:
This parameter is composed of the unsubscribed responses destination flag, and the port where to send the command. The flag & destination combination has to be done with the following macro :

```
ADL_AT_PORT_TYPE ( _port, _flag )
```

The `_port` argument has to be a defined value of the `adl_atPort_e` type, and this required port has to be available (cf. the AT/FCM port Service). If this port is not available, or if it is a GSM or GPRS based one, the command will not be executed.

The `_flag` argument has to be one of the values defined below:
If set to TRUE: the responses and intermediate responses of the sent command that are not subscribed (ie. not listed in the **`adl_atCmdCreate`** function arguments) will be sent on the required port.
If set to FALSE they will not be sent to the external application.

If the **`ADL_AT_PORT_TYPE`** macro is not used, by default the command will be sent to the Open AT® virtual port (see next paragraph for more information about At commands ports).

Rsphdl:
Handler of the callback function associated to all the responses and intermediate responses subscribed in the **`adl_atCmdCreate`** function call.

Note that the callback function will be called one time on each response line sent back by the Wavecom OS. For example, since the "AT+CGMR" commands replies with two lines (Software version response, and then "OK" response), the response handler will be called two times if all responses are subscribed.

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 received response.
The 'adl_atResponse_t' structure is defined as follows (declared in the adl_at.h header file):

```
typedef struct
{
  adl_strID_e           RspID;
  adl_atPort_e          Dest;
  u16                   StrLength;
  ascii                 StrData[1];
} adl_atResponse_t;
```

This structure members are defined below:

- RspID :
  Detected standard response ID if the received response is a standard one. See § 3.19 for more information.

- Dest :
  Port on which the command has been executed ; it is also the destination port where the response will be forwarded if the handler returns TRUE.

- StrLength & StrData :
  Response string length & value.

The return value of the callback function has to be TRUE if the response string has to be sent to the provided port, FALSE otherwise.

This allows a variable number of arguments, where ADL expects a list of response and intermediate response to subscribe to. When the command is executed, its responses are compared with each item of this list. For each matching response, the callback function is called ; the other responses are processed as required by the RspFlag parameter.
Note:
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.

The elements of this response list can also be set as an adl_rsp_ID_e response ID. Please refer to § 3.19 for more information.

- **Returned values**
  - OK on success (the command will be executed on the required port as soon as possible)
  - ADL_RET_ERR_PARAM on parameter error (NULL command string, or "a/" command required (this command can not be used with the **`adl_atCmdCreate`** function))
  - ADL_RET_ERR_UNKNOWN_HDL if the required port is not available.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Note 1 :
This function can be associated with the **adl_atCmdSubscribe** one for filtering or spying any intermediate response or response of a specific command send by the external application. (See the example below).
Note 2:
Commands sent through the adl_atCmdCreate function are directly submitted to the Wavecom OS AT interface: they can not be filtered by an adl_atCmdSubscribe mechanism. The adl_atCmdSubscribe function filters only the commands coming from external Wireless CPU ports.
Note 3:
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 on this port will just store the required command, in order to send those ones as soon as the running "Text Mode" command has ended.
Note 4:
A command sent through the **adl_atCmdCreate** function must be canceled by another command sent later through the same function. E.g. if ATD & ATH commands are sent through the function (before ATD answers), the ATD command does not receive any response, since its execution has been canceled by the ATH command.

- **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",
                    ADL_AT_PORT_TYPE ( paras->Port, 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,
                    ADL_AT_PORT_TYPE ( paras->Port, TRUE),
                    "+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);
}
```

3.2.5.2 The adl_atCmdSendText Function

This function allows to provide a running "Text Mode" command on a specific port (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 ( adl_port_e Port,
                       ascii * Text )
```

- **Parameters**

  Port:
  Port on which is currently running the "Text Mode" command, waiting for some text input.

  Text:
  Text to be provided to the running "Text Mode" command on the required port. 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_UNKNOWN_HDL if the required port is not available.
  - ADL_RET_ERR_BAD_STATE if there is no "Text Mode" command currently running on the required port.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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 on this port).
It is possible to insert new lines (\'r' chracters) in the text body.

3.2.5.3 AT Commands Ports Processing

Several AT commands ports are available on the Wireless CPU ; an application may know each port's current state using the AT/FCM Port service.

When an AT command is sent using the adl_atCmdCreate function, this one is pushed on the required port inner command stack. ADL is processing one command stack by available port on the Wireless CPU.

When an AT command is sent from an external application on a specific port, this command is also pushed on the required port inner command stack.

For each command stack, while this stack is not empty, ADL sends the commands one by one (ie. ADL sends the command on the required port, waits until the terminal response is received, and then continue with the next command) until reaching the stack's end.

In addition to Wireless CPU physical UART ports and logical 27.010 channel ports, there is an additional Open AT® virtual port, usable to send commands only with Open AT® applications (in order not to be disturbed, or not to disturb applications running on the Wireless CPU physical ports).

The ADL AT command stacks architecture is resumed with the scheme below:

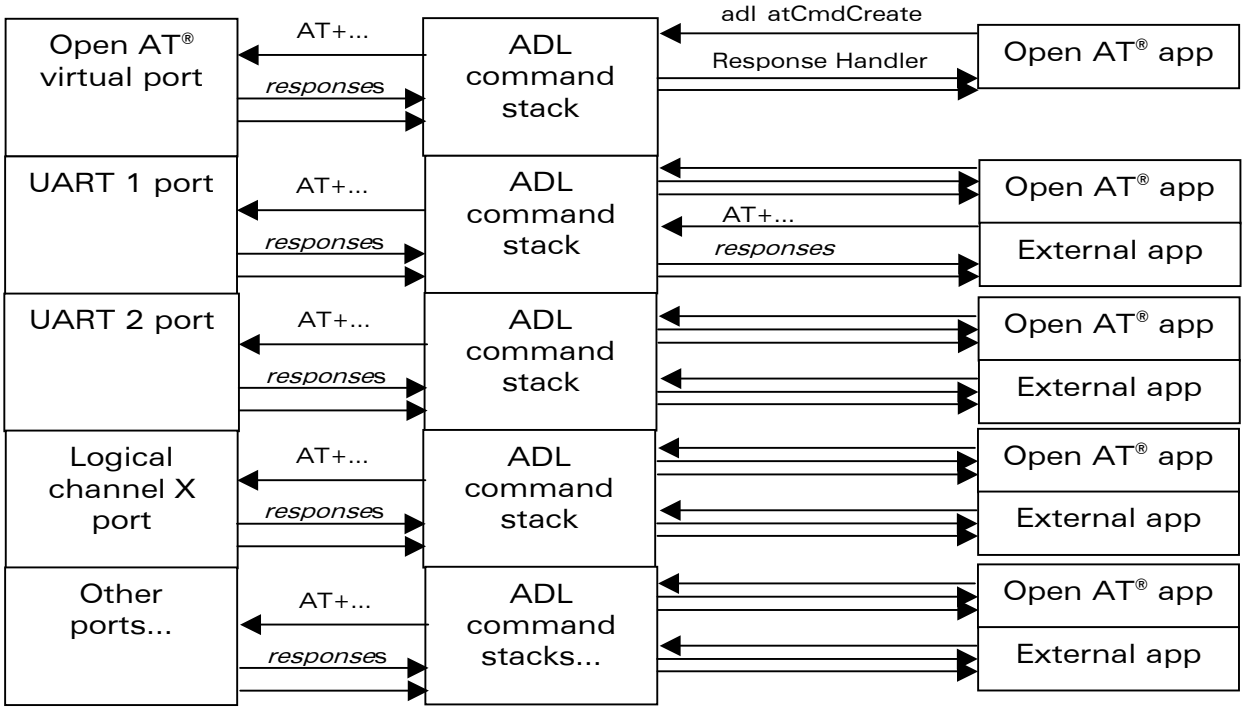


Figure 4: ADL AT Command Stacks Architecture

3.3 Timers

3.3.1 Required Header Files

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

3.3.2 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**

o 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.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Note:
The function will return a NULL pointer if the timer value is zero. The timer will not be started.

3.3.3 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.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.3.4 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);
}
```

3.4 Memory Service

3.4.1 Required Header File

The header file for the memory functions is:

`adl_memory.h`

3.4.2 The adl_memGetInfo Function

This function returns information about the Open AT® RAM areas sizes.

- **Prototype**

  `s32 adl_memGetInfo ( adl_memInfo_t * Info )`

- **Parameters**

  Info:
  Structure updated by the function, using the following type:

```
typedef struct
{
    u32 TotalSize;
    u32 StackSize;
    u32 HeapSize;
    u32 GlobalSize;
} adl_memInfo_t;
```

  - **`TotalSize`**

  Total RAM size for the Open AT® application (in bytes).

  Please refer to the **2.4Memory Resources** chapter for more information.

  - **`StackSize`**

  Open AT® application call stack area size (in bytes).

  This size is defined by the Open AT® application through the **`wm_apmCustomStackSize`** constant (Please refer to the 3 Mandatory API chapter for more information).

  Note:
  This field is set to 0 under Remote Task Environment.

  - **`HeapSize`**

  Open AT® application total heap memory area size (in bytes).

  This size is the difference between the total Open AT® memory size and the Global & Stack areas sizes.

  Note:
  This field is set to 0 under Remote Task Environment.

- **GlobalSize**

Open AT® application global variables area size (in bytes).

This size is defined at the binary link step; it includes the ADL library, plug-in libraries (if any) and Open AT® application global variables.

Note:
This field is set to 0 under Remote Task Environment.

- **Reminder:**

  The Open AT® RAM is divided in three areas (Call stack, Heap memory & Global variables). This function returns the area sizes.

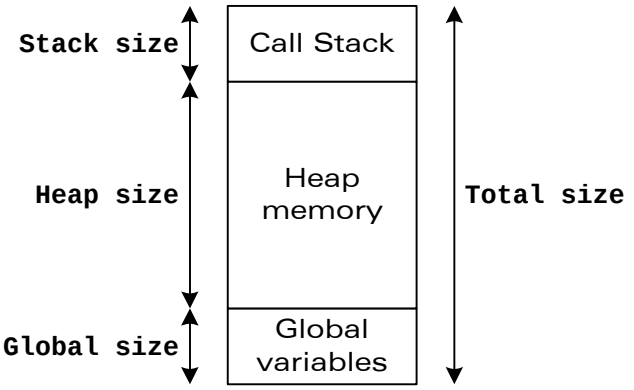


Figure 5: Open AT® RAM Mapping, with `adl_memInfo_t` Structure Field Names

- **Returned values**
  - OK on success; the **Info** parameter is updated in the Open AT® RAM information.
  - ADL_RET_ERR_PARAM on parameter error

3.4.3 The adl_memGet Function

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

- **Prototype**

```
void * adl_memGet ( u32 size )
```

- **Parameters**

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

- **Returned values**
  - A pointer to the allocated memory buffer on success.

- If the memory allocation fails, this function will lead to a ADL_ERR_MEM_GET error, which can be handled by the Error Service. If this error is filtered and refused by the error handler, the function will return NULL. Please refer to the paragraph 3.10 for more information.

- Memory allocation may also fail due to an unrecoverable corrupted memory state; one of the following exceptions is then generated (these exceptions cannot be filtered by the Error service, and systematically lead to a reset of the Wireless CPU):

    - **`RTK exception 166`**
      A buffer header or footer data is corrupted: a write overflow has occurred on this block.

3.4.4 The adl_memRelease Function

This function releases the allocated memory buffer designed by the supplied pointer.

- **Prototype**

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

- **Parameters**

  ptr:
  A pointer on the allocated memory buffer.

- **Returned values**

  - TRUE if the memory was correctly released.
    In this case, the provided pointer is set to NULL.
  -
  - If the memory release fails, one of the following exceptions is generated (these exception cannot be filtered by the Error service, and systematically lead to a reset of the Wireless CPU):

    - **`RTK exception 155`**
      The supplied address is out of the heap memory address range.
    - **`RTK exception 161`** or **`RTK exception 166`**
      The supplied buffer header or footer data is corrupted: a write overflow has occurred on this block
    - **`RTK exception 159`** or **`RTK exception 172`**
      The heap memory release process has failed due to a global memory corruption in the heap area.

3.4.5 Heap Memory Block Status

A list of the currently reserved heap memory blocks can be displayed at any time using the Target Monitoring Tool "Get RTK Status" command. Please refer to the Tools Manual (document 3) for more information.

3.4.6 Example

This example demonstrates how to use the Memory service in a nominal case (error cases are not handled).
Complete examples using the Memory service are also available on the SDK (this service is used by almost all examples).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    adl_memInfo_t MemInfo;
    u8 * MyByteBuffer

    // Gets Open AT RAM information
    adl_memGetInfo ( &MemInfo );

    // Allocates a 10 bytes memory buffer
    MyByteBuffer = ( u8 * ) adl_memGet ( 10 );

    // Releases the previously allocated memory buffer
    adl_memRelease ( MyByteBuffer );
}
```

3.5 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 (document 3).

3.5.1 Required Header File

The header file for the flash functions is:

`adl_traces.h`

3.5.2 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).

3.5.3 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.

3.5.4 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).

3.6 Flash

3.6.1 Required Header File

The header file for the flash functions is:

```
adl_flash.h
```

3.6.2 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.

3.6.2.1 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.

3.6.2.2 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 Wireless CPU's flash memory. The two storage places (Wireless CPU and PC one) may be synchronized using the RTE Monitor interface (cf. the Tools Manual [3]for more information).

3.6.3 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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.

3.6.4 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
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.6.5 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)
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.6.6 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.6.7 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).
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.6.8 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.

3.6.9 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**

  - On success:
    - ID count allocated on the provided handle if any;
    - the total remaining ID count if the handle is set to NULL
  - ADL_RET_ERR_UNKNOWN_HDL if handle is not subscribed
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.6.10 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

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.7 FCM Service

ADL provides a FCM (Flow Control Manager) service to handle all FCM events, and to access to the data ports provided on the product.
An ADL application may subscribe to a specific flow (UART 1, UART 2 or USB physical/virtual ports, GSM CSD call data port, GPRS session data port or Bluetooth virtual data ports) to exchange data on it.

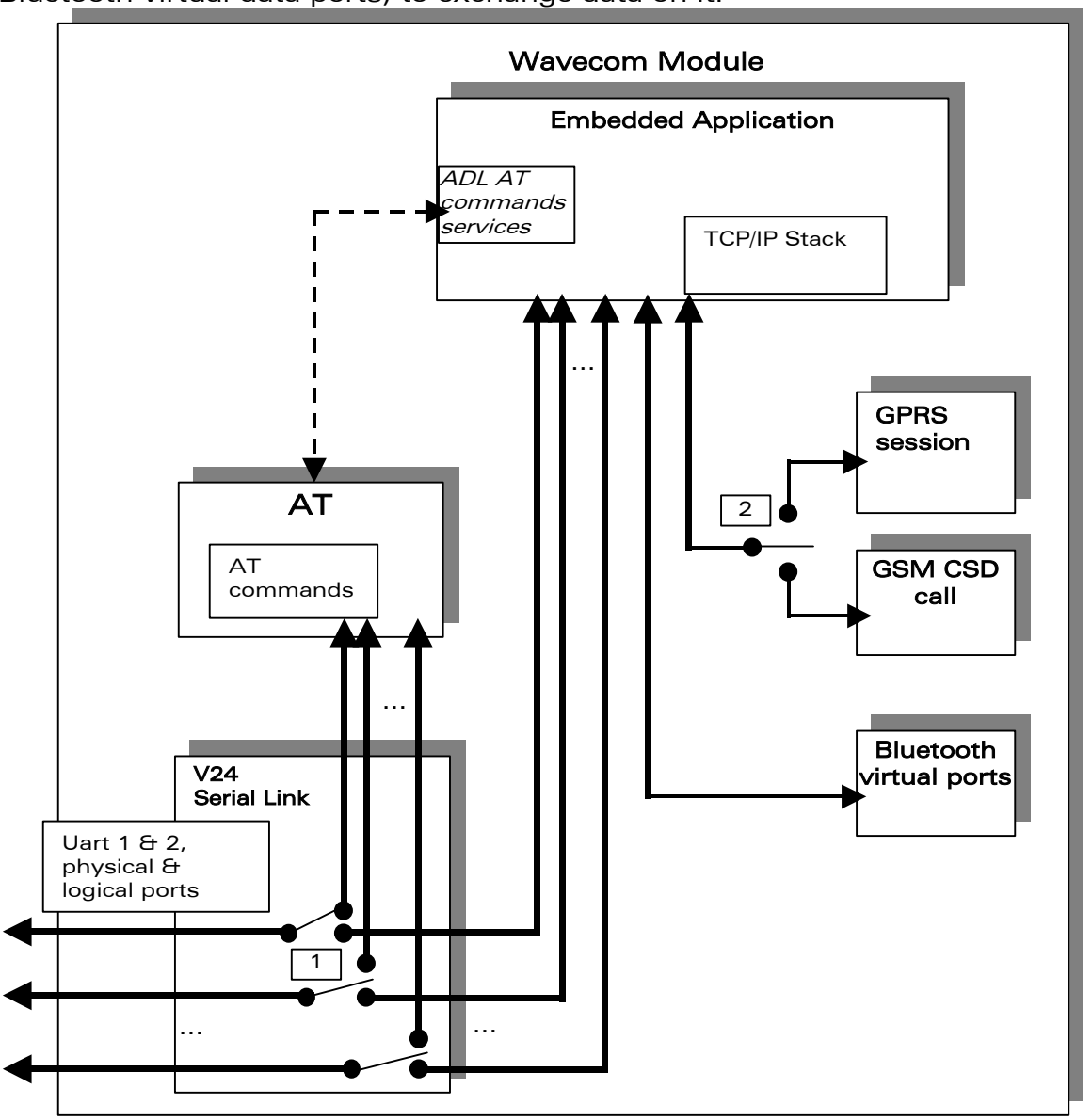


Figure 6: Flow Control Manager Representation

By default (ie. without any Open AT® application, or if the application does not use the FCM service), all the Wireless CPU's ports are processed by the Wavecom OS. The default behaviors are :

- When a GSM CSD call is set up, the GSM CSD data port is directly connected to the UART port where the ATD command was sent ;
- When a GPRS session is set up, the GPRS data port is directly connected to the UART port where the ATD or AT+CGDATA command was sent ;
- When a Bluetooth peripheral is detected & connected through an SPP based profile, a local data bridge may be set up between a Bluetooth virtual data port and the required UART port, using the AT+WLDB command.

Once subscribed by an Open AT® application with the FCM service, a port is no more available to be used with the AT commands by an external application. The available ports are the ones listed in the ADL AT/FCM Ports service :

- ADL_PORT_UART_X / ADL_PORT_UART_X_VIRTUAL_BASE identifiers may be used to access to the Wireless CPU's physicals UARTS, or logical 27.010 protocol ports ;
- ADL_PORT_GSM_BASE identifier may be used to access to a remote modem (connected through a GSM CSD call) data flow ;
- ADL_PORT_GPRS_BASE identifier may be used to exchange IP packets with the operator network and the Internet ;
- ADL_PORT_BLUETOOTH_VIRTUAL_BASE may be used to access to a connected Bluetooth device data stream with the Serial Port Profile (SPP).

The "1" switchs on the figure above means that UART based ports may be used with AT commands or FCM services as well. These switches are processed by the adl_fcmSwitchV24State function.
The "2" switch on the figure above means that either the GSM CSD port or the GPRS port may be subscribed at one time, but not both together.

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.

3.7.1 Required Header File

The header file for the FCM functions is:

```
adl_fcm.h
```

3.7.2 The adl_fcmIsAvailable Function

This function allows to check if the required port is available and ready to handle the FCM service.

- **Prototype**

```
bool  adl_fcmIsAvailable   (  adl_fcmFlow_e Flow );
```

- **Parameters**

  Flow:
  Port from which to require the state.

- **Returned values**

  - TRUE if the port is ready to handle the FCM service
  - FALSE if it is not ready

  Notes :
  All ports should be available for the FCM service, except :
  - The Open AT® virtual one, which is only usable for AT commands,
  - The Bluetooth virtual ones with enabled profiles other than the SPP one,
  - If the port is already used to handle a feature required by an external application through the AT commands (+WLDB command, or a CSD/GPRS data session is already running)

3.7.3 The adl_fcmSubscribe Function

This function subscribes to the FCM service, opening the requested port 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 port 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 (UART physical & logical based ports), 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 (UART physical & logical based ports), the corresponding port has to be opened first with the AT+WMFM command (for physical ports), or with the 27.010 protocol driver on the external application side (for logical ports), otherwise the subscription will fail. See AT Commands Interface guide (document [2]) for more information.
By default, only the UART1 physical port is opened.

A specific port state may be known using the ADL AT/FCM port service.

- **Prototype**

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

- **Parameters**

  Flow:
  The allowed values are the available ports of the **adl_port_e** type. Only ports with the FCM capability may be used with this service (ie. all ports except the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** and not SPP **ADL_PORT_BLUETOOTH_VIRTUAL_BASE** based ones).

  Please note that the **adl_fcmFlow_e** type is the same than the **adl_port_e** one, except the fact that is may handle some additional FCM specific flags (see below). Previous versions FCM flows identifiers have been kept for ascendant compatibility. However, these constants should be considered as deprecated, and the **adl_port_e** type members should now be used instead.

```
#define ADL_FCM_FLOW_V24_UART1    ADL_PORT_UART1
#define ADL_FCM_FLOW_V24_UART2    ADL_PORT_UART2
#define ADL_FCM_FLOW_V24_USB      ADL_PORT_USB
#define ADL_FCM_FLOW_GSM_DATA     ADL_PORT_GSM_BASE
#define ADL_FCM_FLOW_GPRS         ADL_PORT_GPRS_BASE
```

  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_PORT_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 (UART physical & logical based ports) : 120 bytes ;
- On GSM CSD data port : 270 bytes ;
- On GPRS port : 1500 bytes ;
- On Bluetooth virtual ports : 120 bytes.

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.
Please note that 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 already subscribed in master mode,
  - ADL_RET_ERR_NOT_SUBSCRIBED if a slave subscription is made when 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 port is not available.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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 Wireless CPU 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, 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.

3.7.4 The adl_fcmUnsubscribe Function

This function unsubscribes from a previously subscribed FCM service, closing the previously opened flows. The unsubscription 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.7.5 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.
- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.7.6 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
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.7.7 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 based flows : 2000 bytes ;
- On GSM data flow : no limitation (memory allocation size) ;
- On GPRS flow : 1500 bytes ;
- On Bluetooth virtual ports : 2000 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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.

3.7.8 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];
      u32 Reserved3;
      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 based flows : 2000 bytes ;
  - On GSM data flow : no limitation (memory allocation size) ;
  - On GPRS flow : 1500 bytes ;
  - On Bluetooth virtual ports : 2000 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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).

3.7.9 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*.

3.8 GPIO Service

ADL provides a GPIO service to handle GPIO operations.

3.8.1 Required Header File

The header file for the GPIO functions is:

`adl_gpio.h`

3.8.2 GPIO Types

3.8.2.1 The adl_ioConfig_t Structure

This structure is used by the **`adl_ioSubscribe`** function in order to set the reserved GPIO parameters.

```
typedef struct
{
   adl_ioLabel_u       eLabel;
   u32                 Pad;
   adl_ioDirection_e   eDirection;
   adl_ioState_e       eState;
} adl_ioConfig_t;
```

The **`eLabel`** member represents the GPIO label.
The **`eDirection`** member represents the required GPIO direction.
The **`eState`** member represents the GPIO state at subscription time (for outputs only).

3.8.2.2 The adl_ioLabel_u Union

This union represents the different GPIO labels, depending on the Wireless CPU used.

```
typedef union
{
   adl_ioLabel_Q2686_e  Q2686_Label;
   adl_ioLabel_Q2687_e  Q2687_Label;
} adl_ioLabel_u;
```

The **`Q2686_Label`** member has to be used on the Q2686 Wireless CPU.
The **`Q2687_Label`** member has to be used on the Q2687 Wireless CPU.

Wismo QUIK Q2686 Gpio Labels

The following table lists the Gpio labels for the Wismo Quik Q2686 Wireless CPU and their specific linked features when applicable.

GPIO Identifier	Linked Feature (if any)
ADL_IO_Q2686_GPIO_1	
ADL_IO_Q2686_GPIO_2	
ADL_IO_Q2686_GPIO_3	ADL_IO_FEATURE_INT0
ADL_IO_Q2686_GPIO_4	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_5	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_6	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_7	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_8	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_9	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_10	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_11	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_12	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_13	ADL_IO_FEATURE_KBD
ADL_IO_Q2686_GPIO_14	ADL_IO_FEATURE_UART2
ADL_IO_Q2686_GPIO_15	ADL_IO_FEATURE_UART2
ADL_IO_Q2686_GPIO_16	ADL_IO_FEATURE_UART2
ADL_IO_Q2686_GPIO_17	ADL_IO_FEATURE_UART2
ADL_IO_Q2686_GPIO_18	ADL_IO_FEATURE_SIMPRES
ADL_IO_Q2686_GPIO_19	
ADL_IO_Q2686_GPIO_20	
ADL_IO_Q2686_GPIO_21	
ADL_IO_Q2686_GPIO_22	ADL_IO_FEATURE_BT_RST
ADL_IO_Q2686_GPIO_23	
ADL_IO_Q2686_GPIO_24	
ADL_IO_Q2686_GPIO_25	ADL_IO_FEATURE_INT1
ADL_IO_Q2686_GPIO_26	ADL_IO_FEATURE_BUS_I2C
ADL_IO_Q2686_GPIO_27	ADL_IO_FEATURE_BUS_I2C
ADL_IO_Q2686_GPIO_28	ADL_IO_FEATURE_BUS_SPI1_CLK
ADL_IO_Q2686_GPIO_29	ADL_IO_FEATURE_BUS_SPI1_IO
ADL_IO_Q2686_GPIO_30	ADL_IO_FEATURE_BUS_SPI1_I
ADL_IO_Q2686_GPIO_31	ADL_IO_FEATURE_BUS_SPI1_CS
ADL_IO_Q2686_GPIO_32	ADL_IO_FEATURE_BUS_SPI2_CLK
ADL_IO_Q2686_GPIO_33	ADL_IO_FEATURE_BUS_SPI2_IO
ADL_IO_Q2686_GPIO_34	ADL_IO_FEATURE_BUS_SPI2_I
ADL_IO_Q2686_GPIO_35	ADL_IO_FEATURE_BUS_SPI2_CS
ADL_IO_Q2686_GPIO_36	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_37	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_38	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_39	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_40	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_41	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_42	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_43	ADL_IO_FEATURE_UART1
ADL_IO_Q2686_GPIO_44	

```
typedef enum
{
   ADL_IO_Q2686_GPIO_1  = 3,
   ADL_IO_Q2686_GPIO_2,
   ADL_IO_Q2686_GPIO_3,
   …
   ADL_IO_Q2686_GPIO_43,
   ADL_IO_Q2686_GPIO_44,
   ADL_IO_Q2686_PAD     = 0x7FFFFFFF
} adl_ioLabel_Q2686_e;
```

Note:
As soon as a multiplexed feature is used or not used, an **ADL_IO_EVENT_FEATURE_ENABLED / ADL_IO_EVENT_FEATURE_DISABLED** event is notified to the Open AT® application in order to inform it that the related GPIO is available or not available.

Wismo QUIK Q2687 Gpio Labels

The Gpio labels for the Wismo Quik Q2687 Wireless CPU are defined by the values below; for each GPIO label, it is also indicated if this label is linked to a specific feature.

GPIO Identifier	Linked Feature (if any)
ADL_IO_Q2687_GPIO_1	ADL_IO_FEATURE_BUS_PARALLEL_ADDR2_CS2
ADL_IO_Q2687_GPIO_2	ADL_IO_FEATURE_BUS_PARALLEL_ADDR1
ADL_IO_Q2687_GPIO_3	ADL_IO_FEATURE_INT0
ADL_IO_Q2687_GPIO_4	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_5	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_6	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_7	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_8	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_9	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_10	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_11	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_12	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_13	ADL_IO_FEATURE_KBD
ADL_IO_Q2687_GPIO_14	ADL_IO_FEATURE_UART2
ADL_IO_Q2687_GPIO_15	ADL_IO_FEATURE_UART2
ADL_IO_Q2687_GPIO_16	ADL_IO_FEATURE_UART2
ADL_IO_Q2687_GPIO_17	ADL_IO_FEATURE_UART2
ADL_IO_Q2687_GPIO_18	ADL_IO_FEATURE_SIMPRES
ADL_IO_Q2687_GPIO_19	
ADL_IO_Q2687_GPIO_20	
ADL_IO_Q2687_GPIO_21	
ADL_IO_Q2687_GPIO_22	ADL_IO_FEATURE_BT_RST
ADL_IO_Q2687_GPIO_23	
ADL_IO_Q2687_GPIO_24	

GPIO Identifier	Linked Feature (if any)
ADL_IO_Q2687_GPIO_25	ADL_IO_FEATURE_INT1
ADL_IO_Q2687_GPIO_26	ADL_IO_FEATURE_BUS_I2C
ADL_IO_Q2687_GPIO_27	ADL_IO_FEATURE_BUS_I2C
ADL_IO_Q2687_GPIO_28	ADL_IO_FEATURE_BUS_SPI1_CLK
ADL_IO_Q2687_GPIO_29	ADL_IO_FEATURE_BUS_SPI1_IO
ADL_IO_Q2687_GPIO_30	ADL_IO_FEATURE_BUS_SPI1_I
ADL_IO_Q2687_GPIO_31	ADL_IO_FEATURE_BUS_SPI1_CS
ADL_IO_Q2687_GPIO_32	ADL_IO_FEATURE_BUS_SPI2_CLK
ADL_IO_Q2687_GPIO_33	ADL_IO_FEATURE_BUS_SPI2_IO
ADL_IO_Q2687_GPIO_34	ADL_IO_FEATURE_BUS_SPI2_I
ADL_IO_Q2687_GPIO_35	ADL_IO_FEATURE_BUS_SPI2_CS
ADL_IO_Q2687_GPIO_36	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_37	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_38	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_39	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_40	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_41	ADL_IO_FEATURE_UART1
ADL_IO_Q2687_GPIO_42	ADL_IO_FEATURE_UART1
ADL_IO Q2687_GPIO_43	ADL_IO_FEATURE_UART1
ADL_IO Q2687_GPIO_44	

```
typedef enum
{
   ADL_IO_Q2687_GPIO_1  = 2,
   ADL_IO_Q2687_GPIO_2,
   ADL_IO_Q2687_GPIO_3,
   …
   ADL_IO_Q2687_GPIO_43,
   ADL_IO_Q2687_GPIO_44,
   ADL_IO_Q2687_PAD     = 0x7FFFFFFF
} adl_ioLabel_Q2687_e;
```

Note:
As soon as a multiplexed feature is used (or no longer used), an **ADL_IO_EVENT_FEATURE_ENABLED / ADL_IO_EVENT_FEATURE_DISABLED** event is notified to the Open AT® application in order to inform it that the related GPIOs are no longer available (or available again).

3.8.2.3 The adl_ioDirection_e Type

This type represents the direction used for a GPIO.

```
typedef enum
{
   ADL_IO_OUTPUT,
   ADL_IO_INPUT
} adl_ioDirection_e;
```

The **ADL_IO_OUTPUT** constant is used to set a GPIO as an output.
The **ADL_IO_INPUT** constant is used to set a GPIO as an input.

3.8.2.4 The adl_ioState_e Type

This type represents the state of a GPIO.

```
typedef enum
{
   ADL_IO_LOW,
   ADL_IO_HIGH
} adl_ioState_e;
```

The **ADL_IO_LOW** constant represents the low state of a GPIO.
The **ADL_IO_HIGH** constant represents the high state of a GPIO.

3.8.2.5 The adl_ioSetDirection_t Structure

This type is used by the **adl_ioSetDirection** function to set a GPIO to a new direction.

```
typedef struct
{
   adl_ioLabel_u      eLabel;
   adl_ioDirection_e  eDirection;
} adl_ioSetDirection_t;
```

The **eLabel** member represents the GPIO label.
The **eDirection** member represents the new GPIO direction.

3.8.2.6 The adl_ioRead_t Structure

This type is used by the **adl_ioRead** function to read one or several GPIO values.

```
typedef struct
{
   adl_ioLabel_u  eLabel;
   adl_ioState_e  eState;
} adl_ioRead_t;
```

The **eLabel** member represents the GPIO label.
The **eState** member is set by the function to the GPIO read value.

3.8.2.7 The adl_ioWrite_t Structure

This type is used by the **adl_ioWrite** function to write one or several GPIO values.

```
typedef struct
{
   adl_ioLabel_u  eLabel;
   adl_ioState_e  eState;
} adl_ioWrite_t;
```

The **eLabel** member represents the GPIO label.
The **eState** member represents the GPIO value to be set on the output.

3.8.2.8 The adl_ioFeature_e Type

This type enumerates the Wireless CPU features which are multiplexed with one or several GPIOs.

```
typedef enum
{
   ADL_IO_FEATURE_NONE,
   ADL_IO_FEATURE_BUS_SPI1_CLK,          // SPI 1 bus clock pin
   ADL_IO_FEATURE_BUS_SPI1_IO,           //   SPI   1   bus   bi-
                                         directional data pin
   ADL_IO_FEATURE_BUS_SPI1_I,            //   SPI   1   bus   uni-
                                         directional input pin
   ADL_IO_FEATURE_BUS_SPI1_CS,           // SPI 1 bus hardware chip
                                         select pin
   ADL_IO_FEATURE_BUS_SPI2_CLK,          // SPI 2 bus clock pin
   ADL_IO_FEATURE_BUS_SPI2_IO,           //   SPI   2   bus   bi-
                                         directional data pin
   ADL_IO_FEATURE_BUS_SPI2_I,            //   SPI   2   bus   uni-
                                         directional input pin
   ADL_IO_FEATURE_BUS_SPI2_CS,           // SPI 2 bus hardware chip
                                         select pin
   ADL_IO_FEATURE_BUS_I2C,               // I2C bus
   WM_IO_FEATURE_BUS_PARALLEL_ADDR1,     // Parallel bus Address pin
   WM_IO_FEATURE_BUS_PARALLEL_ADDR2_CS2, // Parallel bus chip select
                                         2 / Address pin
   ADL_IO_FEATURE_KBD,                   // Keypad
   ADL_IO_FEATURE_SIMPRES,               // SIM presence signal
   ADL_IO_FEATURE_UART1,                 // UART 1 port
   ADL_IO_FEATURE_UART2,                 // UART 2 port
   ADL_IO_FEATURE_INT0,                  // External interrupt 0
   ADL_IO_FEATURE_INT1,                  // External interrupt 1
   ADL_IO_FEATURE_BT_RST,                // Bluetooth reset signal
   ADL_IO_FEATURE_LAST
} adl_ioFeature_e;
```

3.8.3 The adl_ioEventSubscribe Function

This function allows the Open AT® application to provide ADL with a call-back for GPIO related events.

- **Prototype**

```
s32 adl_ioEventSubscribe ( adl_ioHdlr_f GpioEventHandler );
```

- **Parameters**

  GpioEventHandler:
  Application provided event call-back function. Please refer to next chapter for event descriptions.

- **Returned values**

  - A positive or null value on success:
    - GPIO event handle, to be used in further GPIO API functions calls;
  - A negative error value otherwise:
    - ADL_RET_ERR_PARAM if a parameter has an incorrect value,
    - ADL_RET_ERR_NO_MORE_HANDLES if the GPIO event service has been subscribed to more than 128 timers.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

  Note:
  In order to set-up an automatic GPIO polling process, the **adl_ioEventSubscribe** function has to be called before the **adl_ioSubscribe**.

3.8.4 The adl_ioHdlr_f Call-back Type

Such a call-back function has to be provided to ADL through the **adl_ioEventSubscribe** interface, in order to receive GPIO related events.

- **Prototype**

```
typedef void (*adl_ioHdlr_f) (  s32             GpioHandle,
                                adl_ioEvent_e   Event,
                                u32             Size,
                                void *          Param );
```

- **Parameters**

  GpioHandle:
  Read GPIO handle for the **ADL_IO_EVENT_INPUT_CHANGED** event. For other events, this parameter value is not relevant.

  Size:
  Number of items (read inputs or updated features) in the **Param** table.

Event & Param:
Event is the received event identifier; other parameters use depends on the event type. The events (defined in the **adl_ioEvent_e** type) are described in the following table.

Event	Meaning	Param
ADL_IO_EVENT_FEATURE_ENABLED	One or several multiplexed features are now enabled on the Wireless CPU. Associated GPIO are not available for subscription.	Updated features identifiers table (using the **adl_ioFeature_e *** type)
ADL_IO_EVENT_FEATURE_DISABLED	One or several multiplexed features are now disabled on the Wireless CPU. Associated GPIO are available for subscription.	Updated features identifiers table (using the **adl_ioFeature_e *** type)
ADL_IO_EVENT_INPUT_CHANGED	One or several of the subscribed inputs have changed. This event will be received only if a polling process is required at GPIO subscription time.	Read values table (using the **adl_ioRead_t *** type)

Note :
If the GPIO event is subscribed in the **adl_main** function, an **ADL_IO_EVENT_FEATURE_ENABLED** event will be generated just after the application has started, in order to notify it with all the enabled features on the Wireless CPU power-on.

3.8.5 The adl_ioEventUnsubscribe Function

This function allows the Open AT® application to unsubscribe from the GPIO events notification.

- **Prototype**

  ```
  s32 adl_ioEventUnsubscribe ( s32 GpioEventHandle );
  ```

- **Parameters**

  GpioEventHandle:
  Handle previously returned by the **adl_ioEventSubscribe** function.

- **Returned values**

  - OK on success,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_NOT_SUBSCRIBED if no GPIO event handler has been subscribed,
  - ADL_RET_ERR_BAD_STATE if a polling process is currently running with this event handle.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.8.6 The adl_ioSubscribe Function

This function subscribes to some GPIOs. For subscribed inputs, a polling system can be configured in order to notify a previously subscribed GPIO event handler with an **ADL_IO_EVENT_INPUT_CHANGED** event.

- **Prototype**

```
s32    adl_ioSubscribe  (  u32          GpioNb,
                           adl_ioConfig_t * GpioConfig,

                           u8           PollingTimerType,
       u32              PollingTime,
                           s32          GpioEventHandle );
```

- **Parameters**

GpioNb:
Size of the **GpioConfig** array.

GpioConfig:
GPIO subscription configuration array, which contains **GpioNb** elements. For each element, the **adl_ioConfig_t** structure members have to be configured.

PollingTimerType:
Type of the polling timer (if required); defined values are:

ADL_TMR_TYPE_100MS	100 ms granularity timer
ADL_TMR_TYPE_TICK	18.5 ms tick granularity timer

If no polling is requested, this parameter is ignored.

PollingTime:
If some GPIO are allocated as inputs, this parameter represents the time interval between two GPIO polling operations (unit is dependent on the **PollingTimerType** value).
Please note that each required polling process uses one of the available ADL timers (Reminder: up to 32 timers can be simultaneously subscribed).
If no polling is requested, this parameter has to be 0.
GpioEventHandle:
GPIO event handle (previously returned by **adl_ioEventSubscribe** function). Associated event handler will receive an **ADL_IO_EVENT_INPUT_CHANGED** event each time one of the subscribed inputs state has changed.
If no polling is requested, this parameter is ignored.

- **Returned values**
  - A positive or null value on success:
    - GPIO handle to be used on further GPIO API functions calls;

- A negative error value:
  - 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_NO_MORE_TIMERS if there is no timer available to start the polling process required by application,
  - ADL_RET_ERR_NO_MORE_HANDLES if no more GPIO handles are available.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.8.7 The adl_ioUnsubscribe Function

This function un-subscribes from a previously allocated GPIO handle.

- **Prototype**

```
s32     adl_ioUnsubscribe   ( s32        GpioHandle );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by adl_ioSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.8.8 The adl_ioSetDirection Function

This function allows the direction of one or more previously allocated GPIO to be modified.

- **Prototype**

```
s32  adl_ioSetDirection  ( s32                    GpioHandle,
                           u32                    GpioNb,
                           adl_ioSetDirection_t * GpioDir );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by **`adl_ioSubscribe`** function.

  GpioNb:
  Size of the **`GpioDir`** array.

  GpioDir:
  GPIO direction configuration structure array, using the **`adl_ioSetDirection_t`** type. For each array element:
  - the **`eLabel`** field identifies which GPIO direction has to be modified,
  - the **`eDirection`** field is the new GPIO direction to be set.

- **Returned values**
  - OK on success,
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.8.9 The adl_ioRead Function

This function allows several GPIOs to be read from a previously allocated handle.

- **Prototype**

```
s32     adl_ioRead          ( s32        GpioHandle,
                              u32        GpioNb,
                              adl_ioRead_t *GpioRead );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by adl_ioSubscribe function.

  GpioNb:
  Size of the **GpioRead** array.

  GpioArray:
  GPIO read structure array, using the **adl_ioRead_t** type. For each array element:
  - the eLabel field has to be provided by the application to identify which GPIO has to be read;
  - the eState field will be updated by the function, with the read value on the input.

- **Returned values**
  - OK on success (read values are updated in the **GpioArray** parameter),
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_BAD_STATE if one of the required GPIOs was not subscribed as an input.

3.8.10 The adl_ioReadSingle Function

This function allows one GPIO to be read from a previously allocated handle.

- **Prototype**

```
s32  adl_ioReadSingle ( s32  GpioHandle,
                        u32  Gpio );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by **`adl_ioSubscribe`** function.

  Gpio:
  Identifier of the GPIO (using one of the **`adl_ioLabel_u`** union type).

- **Returned values**
  - GPIO read value on success,
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_BAD_STATE if one of the required GPIO was not subscribed as an input.

3.8.11 The adl_ioWrite Function

This function writes on several GPIOs from a previously allocated handle.

- **Prototype**

```
s32     adl_ioWrite         ( s32        GpioHandle,
                              u32        GpioNb,
                              adl_ioWrite_t *     GpioWrite );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by adl_ioSubscribe function.

  GpioNb:
  Size of the **`GpioWrite`** array.

  GpioWrite:
  GPIO write structure array, using the **`adl_ioWrite_t`** type. For each array element:

- the **`eLabel`** field identifies which GPIO has to be written;
- the **`eState`** field is the value to be set on the output.

- **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_BAD_STATE if one of the required GPIOs was not subscribed as an output.

3.8.12 The adl_ioWriteSingle Function

This function allows one GPIO to be written from a previously allocated handle.

- **Prototype**

```
s32  adl_ioWriteSingle ( s32  GpioHandle,
                         u32  Gpio,
                         u32  State );
```

- **Parameters**

  GpioHandle:
  Handle previously returned by **adl_ioSubscribe** function.

  Gpio:
  Identifier of the GPIO (using one of the **adl_ioLabel_u** union type).

  State:
  Value to be set on the output.

- **Returned values**

  - OK on success,
  - ADL_RET_ERR_PARAM on parameter error,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_BAD_STATE if one of the required GPIO was not subscribed as an output.

3.8.13 The adl_io GetProductType Function

This function returns the Wireless CPU type.

- **Prototype**

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

- **Returned values**

  This function returns the Wireless CPU type, with the following defined values:

Identifier	Product type
ADL_IO_PRODUCT_TYPE_Q2686	Q2686 Wireless CPU
ADL_IO_PRODUCT_TYPE_Q2687	Q2687 Wireless CPU

3.8.14 The adl_ioGetFeatureGPIOList Function

This function retrieves a list of GPIOs multiplexed with a specific feature (ie. the GPIO which are not available for subscription when this feature is enabled).

- **Prototype**

```
s32  adl_ioGetFeatureGPIOList  ( adl_ioFeature_e  Feature,
                                 u8 *             GpioTab,
                                 u8               GpioNb );
```

- **Parameters**

  Feature:
  Required feature, to retrieve from multiplexed GPIO list.

  GpioTab:
  Returned multiplexed GPIO list containing feature.
  If set to NULL, the required table size is returned by the function.

  GpioNb:
  Size of **`GpioTab`** array.

- **Returned values**
  - Associated feature multiplexed GPIO count on success (if **`GpioTab`** is NULL, or if the provided size matches with the required one,
  - ADL_RET_ERR_PARAM if the provided array size does not match with the required one,
  - ADL_RET_ERR_UNKNOWN_HDL if the feature is unknown.

3.8.15 The adl_ioIsFeatureEnabled Function

This function checks if the required feature is enabled or not.

- **Prototype**

```
bool adl_ioIsFeatureEnabled ( adl_ioFeature_e Feature );
```

- **Parameters**

  Feature:
  Required feature.

- **Returned values**
  - FALSE if the feature is unknown or if it is disabled (the associated GPIO are available for subscription),
  - TRUE if the feature is enabled (the associated GPIO is not available for subscription).

3.8.16 Example

This example demonstrates how to use the GPIO service in a nominal case (error cases not handled) on the Q2686 Wireless CPU.

Complete examples using the GPIO service are also available on the SDK (generic Telemetry sample, generic Drivers library sample).

```
// Global variables & constants

// Subscription data
#define GPIO_COUNT1 2
#define GPIO_COUNT2 1
const adl_ioConfig_t MyGpioConfig1 [ GPIO_COUNT1 ] =
{ { ADL_IO_Q2686_GPIO_1, 0, ADL_IO_OUTPUT, ADL_IO_LOW },
  { ADL_IO_Q2686_GPIO_2, 0, ADL_IO_INPUT } };
const adl_ioConfig_t MyGpioConfig2 [ GPIO_COUNT2 ] =
{ { ADL_IO_Q2686_GPIO_19, 0, ADL_IO_INPUT } };
// Event Handle
s32 MyGpioEventHandle;

// Gpio Handles
s32 MyGpioHandle1, MyGpioHandle2;
// GPIO event handler
void MyGpioEventHandler ( s32 GpioHandle, adl_ioEvent_e Event, u32
Size, void * Param )
{
```

```
    // Check event
    switch ( Event )
    {
        case ADL_IO_EVENT_INPUT_CHANGED :
            // The subscribed input has changed
            TRACE (( 1, "GPIO %d new value: %d",
            ( ( adl_ioRead_t * ) Param ) [0].eLabel,
            ( ( adl_ioRead_t * ) Param ) [0].eState ));
        break;
    }
}

...
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    s32 ReadValue;

    // Subscribe to the GPIO event service
    MyGpioEventHandle = adl_ioEventSubscribe ( MyGpioEventHandler );

    // Subscribe to the GPIO service (One handle without polling,
     // one with a 100ms polling process)
    MyGpioHandle1 = adl_ioSubscribe ( GPIO_COUNT1, MyGpioConfig1, 0, 0,
0 );
    MyGpioHandle2 = adl_ioSubscribe ( GPIO_COUNT2, MyGpioConfig2,
     ADL_TMR_TYPE_100MS, 1, MyGpioEventHandle );

    // Set output
    adl_ioWriteSingle ( MyGpioHandle1, ADL_IO_Q2686_GPIO_1, ADL_IO_HIGH
);

    // Read inputs
    ReadValue = adl_ioReadSingle (MyGpioHandle1, ADL_IO_Q2686_GPIO_2 );
    ReadValue = adl_ioReadSingle (MyGpioHandle2, ADL_IO_Q2686_GPIO_19
);

    // Unsubscribe from the GPIO services
    adl_ioUnsubscribe ( MyGpioHandle1 );
    adl_ioUnsubscribe ( MyGpioHandle2 );

    // Unsubscribe from the GPIO event service
    adl_ioUnsubscribe ( MyGpioEventHandle );
}
```

3.9 Bus Service

ADL provides a bus service to handle SPI I2C & Parallel bus operations.

3.9.1 Required Header File

The header file for the bus functions is:

```
adl_bus.h
```

3.9.2 Bus Types

3.9.2.1 The adl_busType_e Type

This structure defines the bus available for subscription on the Wireless CPU.

```
typedef enum
{
   ADL_BUS_SPI1 = 1,
   ADL_BUS_SPI2,
   ADL_BUS_I2C = 4
   ADL_BUS_PARALLEL = 6
} adl_busType_e;
```

The **ADL_BUS_SPI1** constant is used to access the Wireless CPU SPI1 bus.
The **ADL_BUS_SPI2** constant is used to access the Wireless CPU SPI2 bus.
The **ADL_BUS_I2C** constant is used to access the Wireless CPU I2C bus.
The **ADL_BUS_PARALLEL** constant can be used to access the module Parallel bus.

3.9.2.2 The adl_busSPISettings_t Structure

This structure defines the SPI bus settings for subscription.

```
typedef struct
{
   u32 Clk_Speed;
   u32 Clk_Mode;
   u32 ChipSelect;
   u32 ChipSelectPolarity;
   u32 LsbFirst;
   u32 GpioChipSelect;
   u32 WriteHandling;
   u32 DataLinesConf;
} adl_busSPISettings_t;
```

3.9.2.2.1 Clk_Speed

The **Clk_Speed** parameter is the SPI bus clock speed. Allowed values are in the [0 – 127] range. The SPI clock speed is defined using the formula below:

```
Pclk / ( ( 2 * Clk_Speed ) + 2 )
```

Where **Pclk** is the Wireless CPU Peripheral Clock speed.
On the Q2686 Wireless CPU, **Pclk** rate is set to 26 MHz.

Example: if **Clk_Speed** is set to 0, the SPI bus clock speed is set to 13 MHz.

3.9.2.2.2 Clk_Mode

The **Clk_Mode** parameter is the SPI clock mode. Defined values are:

ADL_BUS_SPI_CLK_MODE_0	rest state 0, data valid on rising edge
ADL_BUS_SPI_CLK_MODE_1	rest state 0, data valid on falling edge
ADL_BUS_SPI_CLK_MODE_2	rest state 1, data valid on rising edge
ADL_BUS_SPI_CLK_MODE_3	rest state 1, data valid on falling edge

3.9.2.2.3 ChipSelect

The **ChipSelect** parameter sets the pin used to handle the Chip Select signal. Defined values are:

ADL_BUS_SPI_ADDR_CS_GPIO	Use a GPIO as Chip Select signal (the **GpioChipSelect** parameter has to be used)
ADL_BUS_SPI_ADDR_CS_HARD	Use the reserved hardware chip select pin for the required bus.
ADL_BUS_SPI_ADDR_CS_NONE	The Chip Select signal is not handled by the ADL BUS service. The application should allocate a GPIO to handle itself the Chip Select signal.

3.9.2.2.4 ChipSelectPolarity

The **ChipSelectPolarity** parameter sets the polarity of the Chip Select signal; defined values are :

ADL_BUS_SPI_CS_POL_LOW	Chip Select signal is active in Low state.
ADL_BUS_SPI_CS_POL_HIGH	Chip Select signal is active in High state.

3.9.2.2.5 LsbFirst

The **LsbFirst** parameter defines the priority for data transmission through the SPI bus, LSB or MSB first. This applies only to data. The Opcode and Address fields sent are always sent with MSB first. Defined values are :

ADL_BUS_SPI_MSB_FIRST	Data buffer is sent with MSB first.
ADL_BUS_SPI_LSB_FIRST	Data buffer is sent with LSB first.

3.9.2.2.6 GpioChipSelect

The **GpioChipSelect** parameter is used only if the **ChipSelect** parameter is set to the **ADL_BUS_SPI_ADDR_CS_GPIO** value. It sets the GPIO label to use as the chip select signal. It has to be a member of the **adl_ioLabel_u** union (see the GPIO service description).
Example: in order to use the Q2686 Wireless CPU GPIO 1 to handle the SPI bus chip select signal, **ChipSelect** parameter has to be set to **ADL_BUS_SPI_ADDR_CS_GPIO** value, and **GpioChipSelect** parameter has to be set to **ADL_IO_Q2686_GPIO_1** value.

3.9.2.2.7 WriteHandling

The **WriteHandling** parameter defines the chip select signal behavior. Defined values are:

ADL_BUS_SPI_WORD_HANDLING	Chip select signal state changes on each written or read word; word size is defined on read or write process request, in the **Size** parameter of the **adl_busAccess_t** configuration structure
ADL_BUS_SPI_FRAME_HANDLING	Chip select signal is enabled at the beginning of the read/write process, and is disabled at the end of this process. Note: This mode is not available with the **ADL_BUS_SPI_ADDR_CS_HARD** Chip Select configuration.

3.9.2.2.8 DataLinesConf

The **DataLinesConf** parameter defines if the SPI bus uses one single pin to handle both input and output data signals, or two pins to handle them separately; defined values are:

ADL_BUS_SPI_DATA_BIDIR	One bi-directional pin is used to handle both input & output data signals.
ADL_BUS_SPI_DATA_UNIDIR	Two pins are used to handle separately input & output data signals

3.9.2.3 The adl_busI2CSettings_t Structure

This structure defines the I2C bus settings for subscription.

```
typedef struct
{
   u32 ChipAddress;
   u32 Clk_Speed;
} adl_busI2CSettings_t;
```

3.9.2.3.1 ChipAddress

The **ChipAddress** parameter sets the remote chip 7 bit address on the I2C bus. Only b1 to b7 bits are used (b0 bit and the 3 most significant bytes are ignored).

Example: if the remote chip address is set to A0, the **ChipAddress** parameter has to be set to the 0xA0 value.

3.9.2.3.2 Clk_Speed

The **Clk_Speed** parameter sets the required I2C bus speed; defined values are:

ADL_BUS_I2C_CLK_STD	Standard I2C bus speed (100 Kbits/s)
ADL_BUS_I2C_CLK_FAST	Fast I2C bus speed (400 Kbits/s)

3.9.2.4 The adl_busParallelCs_t Structure

This type defines the Parallel bus Chip Select.

```
typedef struct
{

   u8 Type;
   u8 Id;
   u8 pad[2]
} adl_busParallelCs_t;
```

3.9.2.4.1 Type

The **Type** parameter defines the Chip Select signal type. The only available value is **ADL_BUS_PARA_CS_TYPE_CS**. All other values are reserved for future use.

3.9.2.4.2 Id

The **Id** parameter defines the Chip Select identifier used; available values are 2 (for CS2 pin) and 3 (for CS3 pin)

3.9.2.5 The adl_busParallelTimingCfg_t Structure

This type defines the Parallel bus timings.

```
typedef struct
{
        u8 AccessTime;
        u8 SetupTime;
        u8 HoldTime;
        u8 TurnaroundTime;
        u8 OpToOpTurnaroundTime;
        u8 pad[3];
} adl_busParallelTimingCfg_t
```

The **OpToOpTurnaroundTime** parameter is reserved for future use.

The other parameters configuration defines the parallel bus timing, in 26 MHz cycles number (one cycle duration is 1/26 MHz = ~38.5 ns), according to the bus mode required at subscription time.

3.9.2.5.1 Motorola Modes Timing

The following timing behavior applies when the **ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW** (E signal low polarity) or **ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH** (E signal high polarity) modes are required at subscription time. In the example given, the Access, Setup & Hold times are set to 1, and the Turnaround time is set to 2.

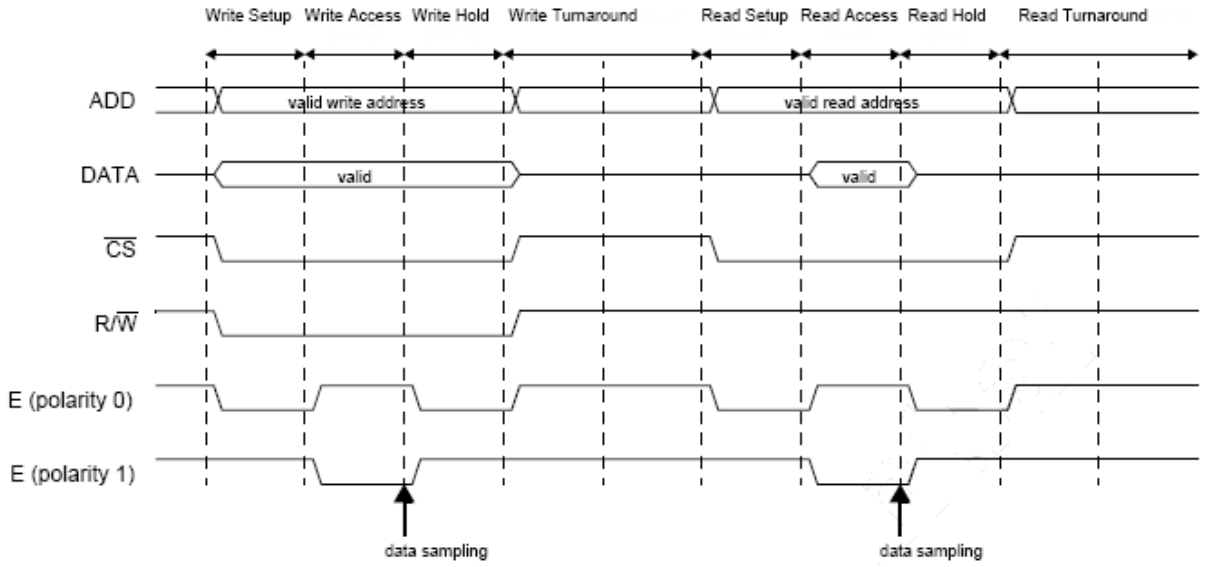


Figure 7 Motorola Modes Timing Example

3.9.2.5.2 Intel Mode Timing

The following timing behavior applies when the **ADL_BUS_PARALLEL_MODE_ASYNC_INTEL** mode is required at subscription time. In this mode, the timing configuration must not be the same for the read or write process.

In this read process example, Setup & Hold times are set to 1, and Access & Turnaround times are set to 3.

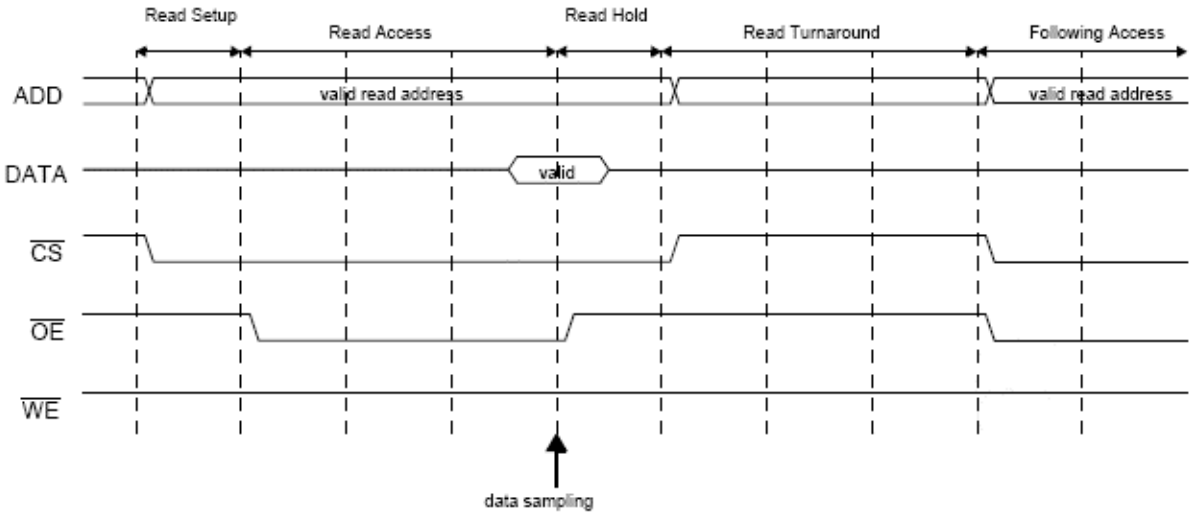


Figure 8: Intel Mode Timing - Read Process Example

In this write process example, Setup & Access time are set to 2, Hold time is set to 1 and Turnaround time is set to 3.

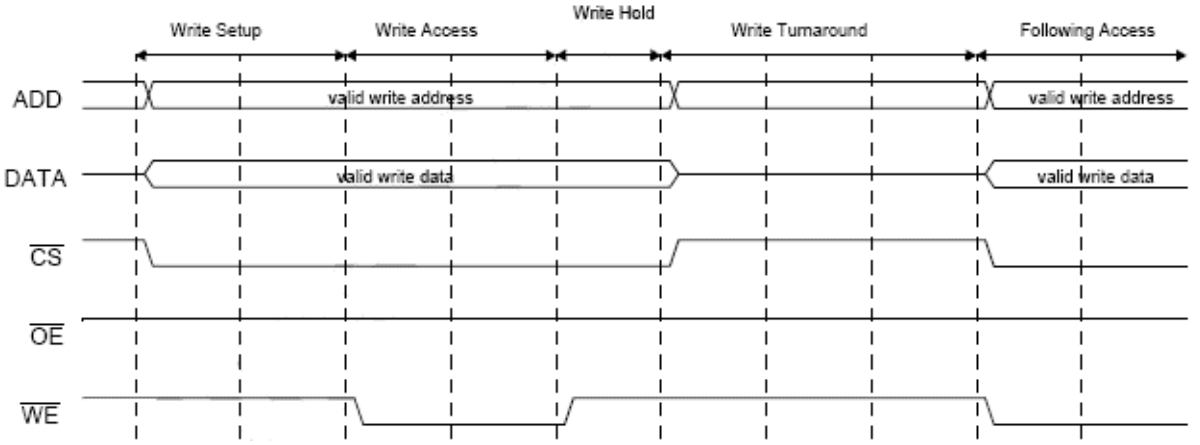


Figure 9: Intel Mode Timing - Write Process Example

3.9.2.6 The adl_busParallelSettings_t Structure

This type defines the Parallel bus settings for subscription.

```
typedef struct
{
   union
   {
        struct
        {
             u8                              Width;
             u8                              Mode;
             u8                              pad[2]
             adl_busParallelTimingCfg_t      ReadCfg;
             adl_busParallelTimingCfg_t      WriteCfg;
             adl_busParallelCs_t             Cs;
             u8                              NbAddNeeded;
        } In;
        struct
        {
             volatile void *  PrivateAddress;
             u32              MaskValidAddress;
        } Out;
   } u;
} adl_busParallelSettings_t;
```

The structure is usable:

- Before the subscription function call, in order to set the required parallel bus configuration (using the **In** union member),
- After the subscription function call, in order to retrieve the address mask to be set at read/write process time (using the **Out** union member).

Caution:

adl_busParallelSettings_t is an union structure, and the required parallel bus configuration cannot be defined with a const.

3.9.2.6.1 Width

The **Width** parameter defines the read/write process data buffer items bit size (8 or 16 bits), using the **adl_busSize_e** type.

3.9.2.6.2 Mode

The **Mode** parameter defines the required parallel bus standard mode to be used.

The following values are available:

ADL_BUS_PARALLEL_MODE_ASYNC_INTEL	Standard Intel mode
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH	Standard Motorola mode, with E signal high polarity
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW	Standard Motorola mode, with E signal low polarity

Please refer to **adl_busParallelTimingCfg_t** type description for information & chronograms about the available modes.

3.9.2.6.3 ReadCfg, WriteCfg

The **ReadCfg & WriteCfg** parameters define the timing configuration for each read and write process, using the **adl_busParallelTimingCfg_t** type.

3.9.2.6.4 Cs

The **Cs** parameter defines the Chip Select signal configuration, using the **adl_busParallelCs_t** type.

3.9.2.6.5 NbAddNeeded

The **NbAddNeeded** parameter defines the required number of address pins to be used by this bus configuration. Maximum address pins number depends on the required Chip Select pin.

Chip Select pin	Maximum address pins number
CS2	2 (A1, A24)
CS3	3 (A1, A24, A25)

Address pins are always reserved by decimal order; Eg:

- If one address pin is required, A1 signal will be reserved.
- If two address pins are required, A1 & A24 signals will be reserved.

As CS2 & A25 signals are multiplexed on the same pin, CS2 signal can be used only if CS3 one was not previously subscribed with three address pins. Please note that CS3 signal can be used with three address pins only if CS2 one is not subscribed.

3.9.2.6.6 PrivateAddress

This output parameter is reserved for ADL private usage.

3.9.2.6.7 MaskValidAddress

The **MaskValidAddress** output parameter provides the application with the bit mask which can be set at direct read/write process time. Each bit is associated with one of the required address pins (Eg. If two address pins are required, two bits will be set in the mask). For each set bit, the associated address pin is the pin corresponding to the bit index (Eg. If the b1 bit is set, the A1 address pin will be usable at read/write process time).

3.9.2.7 The adl_busSettings_u Type

This structure sets the bus configuration parameters that are provided at subscription time.

```
typedef union
{
   adl_busSPISettings_t  SPI;
   adl_busI2Csettings_t  I2C;
   adl_busParallelsettings_t   Parallel;
} adl_busSettings_u;
```

The union **SPI** member is used by the API if an SPI bus subscription is required.

The union **I2C** member is used by the API if an I2C bus subscription is required.

The union **Parallel** member will be used by the API if a Parallel bus subscription is required.

3.9.2.8 The adl_busAccess_t Type

This structure sets the bus access configuration parameters, to be used on a standard read or write process request (for SPI or I2C bus only).

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

3.9.2.8.1 Address & AddressLength

Usable for the both SPI & I2C buses.

The **Address** parameter allows up to 32 bits to be sent on the bus, before starting the read or write process. The number of bits to send is set by the **AddressLength** parameter. If less than 32 bits are required to be sent, only the most significant bits are sent on the bus.

Allowed values for the **AddressLength** parameter are:

- the [0 – 32] range for the SPI bus;
- the [0, 8, 16, 24, 32] values for the I2C bus.

Example: in order to send the "AA" byte on the bus prior to a read or write process, the **Address** parameter has to be set to the **0xAA000000** value, and the **AddressLength** parameter has to be set to 8.

3.9.2.8.2 Opcode & OpcodeLength

Usable only for SPI bus (ignored for I2C bus).

The **Opcode** parameter allows up to 32 bits to be sent on the bus, before starting the read or write process. The number of bits to send is set by the **OpcodeLength** parameter. If less than 32 bits are required to be sent, only the most significant bits are sent on the bus.

Allowed values for the **OpcodeLength** parameter are the [0 – 32] range.

Example: in order to send the "BBB" word on the bus prior to a read or write process, the **Opcode** parameter has to be set to the **0xBBB00000** value, and the **OpcodeLength** parameter has to be set to 12.

3.9.2.8.3 Size

Usable for the both SPI & I2C buses.

The **Size** parameter allows the bit size of read/ write process data buffer items to be set using the **adl_busSize_e** type:

3.9.2.9 The adl_busSize_e Type

This type sets the bit size for read & write processes data buffer items.

```
typedef enum
{
   ADL_BUS_SIZE_1_BIT = 1,
   ADL_BUS_SIZE_2_BITS,
   ADL_BUS_SIZE_3_BITS,
   ADL_BUS_SIZE_4_BITS,
   ADL_BUS_SIZE_5_BITS,
   ADL_BUS_SIZE_6_BITS,
   ADL_BUS_SIZE_7_BITS,
   ADL_BUS_SIZE_BYTE,
   ADL_BUS_SIZE_9_BITS,
   ADL_BUS_SIZE_10_BITS,
   ADL_BUS_SIZE_11_BITS,
   ADL_BUS_SIZE_12_BITS,
   ADL_BUS_SIZE_13_BITS,
   ADL_BUS_SIZE_14_BITS,
   ADL_BUS_SIZE_15_BITS,
   ADL_BUS_SIZE_HALF,
   ADL_BUS_SIZE_WORD = ADL_BUS_SIZE_HALF
} adl_busSize_e;
```

Size parameter values can be:

- From **ADL_BUS_SIZE_1_BIT** to **ADL_BUS_SIZE_ HALF** values for the SPI bus.
- Only the **ADL_BUS_SIZE_BYTE** value for the I2C bus.
- **ADL_BUS_SIZE_BYTE** or **ADL_BUS_SIZE_HALF** for the Parallel bus.

Note:
The **ADL_BUS_SIZE_WORD** value may be used on future Wireless CPU versions to define 32 bit-wide data item size.

The read/write functions data buffer format depends on this bitsize configuration:

- From **ADL_BUS_SIZE_1_BIT** to **ADL_BUS_SIZE_BYTE** values, the data buffer will be considered to be an **u8*** buffer.
- From **ADL_BUS_SIZE_9_BITS** to **ADL_BUS_SIZE_ HALF** values, the data buffer will be considered to be an **u16*** buffer.

Moreover, if the required size is not the **ADL_BUS_SIZE_BYTE** or the **ADL_BUS_SIZE_HALF**, only the least significant bits of each buffer item will be used.

Examples:

- In order to send bytes on the bus, the bit size parameter has to be set to the **ADL_BUS_SIZE_BYTE** value (read/write functions data buffer format will be considered as an **u8*** buffer).
- In order to send 10 bits words on the bus, the bit size parameter has to be set to the **ADL_BUS_SIZE_10_BITS** value (read/write functions data buffer format will be considered as an **u16*** buffer, and only the 10 least significant bits of each **u16** will be sent on the bus).

3.9.3 The adl_busSubscribe Function

This function subscribes to a specific bus, in order to write and read values to/from a remote chip. Up to 8 bus configurations can simultaneously be subscribed, in order to drive un to 8 remote chips.

- **Prototype**

```
s32     adl_busSubscribe    ( adl_busType_e  BusType,
                              adl_busSettings_u * BusSettings);
```

- **Parameters**

  BusType:
  Type of the bus to subscribe to, using the **adl_busType_e** type values.
  BusSettings:
  Subscribed bus configuration parameters, using the **adl_busSettings_u** type.

- **Returned values**

  - A positive or null value on success:
    - BUS handle, to be used in further BUS API functions calls;

- A negative error value:
  - ADL_RET_ERR_PARAM if a parameter has an incorrect value
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the required bus is already subscribed with the provided configuration
  - ADL_RET_ERR_NO_MORE_HANDLES if there are no more free bus handles (8 bus configurations have already been subscribed)
  - ADL_RET_ERR_BAD_HDL if a GPIO required by the provided bus configuration is currently subscribed by an Open AT® application.
  - ADL_RET_ERR_NOT_SUPPORTED if the required bus type is not supported by the Wireless CPU on which the application is running.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Notes :
A bus is available only if the GPIO multiplexed with the corresponding feature is not yet subscribed by an Open AT® application. Concerned features depend on the bus type and configuration:

Bus Type & Configuration	Multiplexed Features
ADL_BUS_I2C bus	**ADL_IO_FEATURE_BUS_I2C**
ADL_BUS_SPI1 bus, not using the hardware chip select pin, using one bi-directional data pin	**ADL_IO_FEATURE_BUS_SPI1_CLK,** **ADL_IO_FEATURE_BUS_SPI1_IO**
ADL_BUS_SPI1 bus, using the hardware chip select pin, using one bi-directional data pin	**ADL_IO_FEATURE_BUS_SPI1_CLK,** **ADL_IO_FEATURE_BUS_SPI1_IO,** **ADL_IO_FEATURE_BUS_SPI1_CS**
ADL_BUS_SPI1 bus, not using the hardware chip select pin, using two data pins	**ADL_IO_FEATURE_BUS_SPI1_CLK,** **ADL_IO_FEATURE_BUS_SPI1_IO,** **ADL_IO_FEATURE_BUS_SPI1_I**
ADL_BUS_SPI1 bus, using the hardware chip select pin, using two data pins	**ADL_IO_FEATURE_BUS_SPI1_CLK,** **ADL_IO_FEATURE_BUS_SPI1_IO,** **ADL_IO_FEATURE_BUS_SPI1_I,** **ADL_IO_FEATURE_BUS_SPI1_CS**
ADL_BUS_SPI2 bus, not using the hardware chip select pin, using one bi-directional data pin	**ADL_IO_FEATURE_BUS_SPI2_CLK,** **ADL_IO_FEATURE_BUS_SPI2_IO**
ADL_BUS_SPI2 bus, using the hardware chip select pin, using one bi-directional data pin	**ADL_IO_FEATURE_BUS_SPI2_CLK,** **ADL_IO_FEATURE_BUS_SPI2_IO,** **ADL_IO_FEATURE_BUS_SPI2_CS**

Bus Type & Configuration	Multiplexed Features
ADL_BUS_SPI2 bus, not using the hardware chip select pin, using two data pins	**ADL_IO_FEATURE_BUS_SPI2_CLK,** **ADL_IO_FEATURE_BUS_SPI2_IO,** **ADL_IO_FEATURE_BUS_SPI2_I**
ADL_BUS_SPI2 bus, using the hardware chip select pin, using two data pins	**ADL_IO_FEATURE_BUS_SPI2_CLK,** **ADL_IO_FEATURE_BUS_SPI2_IO,** **ADL_IO_FEATURE_BUS_SPI2_I,** **ADL_IO_FEATURE_BUS_SPI2_CS**
ADL_BUS_PARALLEL bus, using one hardware chip select CS3, using one address pin	None
ADL_BUS_PARALLEL bus, using one hardware chip select CS3, using two address pins	**ADL_IO_FEATURE_BUS_PARALLEL_ADDR1**
ADL_BUS_PARALLEL bus, using one hardware chip select CS3, using three address pins	**ADL_IO_FEATURE_BUS_PARALLEL_ADDR1** **ADL_IO_FEATURE_BUS_PARALLEL_ADDR2_CS2**
ADL_BUS_PARALLEL bus, using one hardware chip select CS2, using one address pin	**ADL_IO_FEATURE_BUS_PARALLEL_ADDR2_CS2**
ADL_BUS_PARALLEL bus, using one hardware chip select CS2, using two address pins	**ADL_IO_FEATURE_BUS_PARALLEL_ADDR1** **ADL_IO_FEATURE_BUS_PARALLEL_ADDR2_CS2**

Once the bus is subscribed, the features corresponding to the required configuration are enabled, and the multiplexed GPIO are not available for subscription by the Open AT® application, or through the standard AT commands.

3.9.4 The adl_busUnsubscribe Function

This function unsubscribes from a previously subscribed SPI or I2C bus type. This function is not usable with the Parallel bus.

- **Prototype**

```
s32     adl_busUnsubscribe   ( s32      Handle );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_busSubscribe function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.9.5 The adl_busRead Function

This function reads data from a previously subscribed bus SPI or I2C type. This function is not usable with the Parallel bus.

- **Prototype**

```
s32     adl_busRead          (s32              Handle,
                              adl_busAccess_t  * pAccessMode,
                              u32              DataLen,
                              void *           Data );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_busSubscribe function.

  pAccessMode:
  Bus access mode, defined according to the **adl_busAccess_t** structure. Please refer to this structure description (§ 3.9.2.8 The adl_busAccess_t Type) for more information.

  DataLen:
  Number of items to read from the bus.

  Data:
  Buffer where to copy the read items.
  Note:
  items bit size is defined in the **pAccessMode** configuration structure..

- **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,
  - ADL_RET_ERR_BAD_STATE if there is no acknowledgement from the remote chip on the bus (I2C bus only).

3.9.6 The adl_busWrite Function

This function writes on a previously subscribed SPI or I2C bus type. This function is not usable with the Parallel bus.

- **Prototype**

```
s32     adl_busWrite         ( s32              Handle,
                               adl_busAccess_t  * pAccessMode,
                               u32              DataLen,
                               void *           Data );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_busSubscribe function.

  pAccessMode:
  Bus access mode, defined according to the **adl_busAccess_t structure**; please refer to this structure description (§ 3.9.2.8 The adl_busAccess_t Type) for more information.

  DataLen:
  Number of items to write on the bus.

  Data:
  Data buffer to write on the bus.

  Note:
  Items bit size is defined in the **pAccessMode** configuration structure.

- **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,
  - ADL_RET_ERR_BAD_STATE if there is no acknowledgement from the remote chip on the bus (I2C bus only).

3.9.7 The adl_busDirectRead Function

This function reads data about previously subscribed Parallel bus type.

This function is not usable with the SPI or I2C bus.

- **Prototype**

```
s32     adl_busDirectRead    ( s32            Handle,
                               u32            ChipAddress,
                               u32            DataLen,
                               void *         Data );
```

- **Parameters**

  Handle:
  Handle previously returned by the **adl_busSubscribe** function.

  ChipAddress:
  Chip address configuration. This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.

  DataLen:
  Number of items to read from the bus.

Data:
Buffer into which the read items are copied (items bit size (8 or 16 bits) is defined at subscription time in the configuration structure).

- **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.

3.9.8 The adl_busDirectWrite Function

This function writes on a previously subscribed bus.

- **Prototype**

```
s32     adl_busDirectWrite   ( s32              Handle,
                               u32              ChipAddress,
                               u32              DataLen,
                               void *           Data );
```

- **Parameters**

Handle:
Handle previously returned by the **adl_busSubscribe** function.

ChipAddress:
Chip address configuration. This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.

DataLen:
Number of items to write on the bus.

Data:
Data buffer to write on the bus (item bit size (8 or 16 bits) is defined at subscription time in the configuration structure).

- **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.

3.9.9 Example

This example simply demonstrates how to use the BUS service in a nominal case (error cases are not handled) on the Q2687 Wireless CPU.

Complete examples of BUS service used are also available on the SDK (generic Data_Storage sample, generic Drivers library sample).

```
// Global variables & constants

// SPI Subscription data
const adl_busSPISettings_t MySPIConfig =
{
    0,                          // 13MHz SPI clock speed
    ADL_BUS_SPI_CLK_MODE_0,     // Mode 0 clock
    ADL_BUS_SPI_ADDR_CS_GPIO,   // Use a GPIO to handle the Chip
                                Select signal
    ADL_BUS_SPI_CS_POL_LOW,     // Chip Select active in low state
    ADL_BUS_SPI_MSB_FIRST,      // Data are sent MSB first
    ADL_IO_ Q2687_GPIO_31,      // Use GPIO 31 to handle the Chip
                                Select signal
    ADL_BUS_SPI_FRAME_HANDLING, // Chip Select active during the whole
                                frame
    ADL_BUS_SPI_DATA_BIDIR      // Chip connected using one data pin
};

// I2C Subscription data
const adl_busI2CSettings_t MyI2CConfig =
{
    0x20,                       // Chip address is 0x20
    ADL_BUS_I2C_CLK_STD         // Chip uses the I2C standard clock
speed
};

// Parallel Subscription data
adl_busParallelSettings_t MyParaConfig =
{
    ADL_BUS_SIZE_BYTE,                          // 8 bits parallel
bus
    ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW,   //Motorola mode,
                                                Low E signal polarity
    {0x00,0x00},
    { 1, 1, 1, 2, 0, 0, 0 },                    //Read timing
                                                settings
    { 1, 1, 1, 2, 0, 0, 0 },                        //Write timing
                                                settings
 {
     ADL_BUS_PARA_CS_TYPE_CS,
     3                                          // Use CS3 pin
    { 0,0 }
  },
    3                                           //3 address pins
                                                required
};
```

```
// Write/Read buffer sizes
#define WRITE_SIZE 5
#define READ_SIZE  3

// Access configuration structure
adl_busAccess_t AccessConfig =
{
    0, 0, 0, 0,         // No Opcode, No Address
    ADL_BUS_SIZE_BYTE   // Byte (u8) data buffer
};

// BUS Handles
s32 MySPIHandle, MyI2Chandle, MyParaHandle;

// Data buffers
u8 WriteBuffer [ WRITE_SIZE ], ReadBuffer [ READ_SIZE ];

...

// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    s32 ReadValue;

    // Subscribe to the SPI1 BUS
    MySPIHandle = adl_busSubscribe ( ADL_BUS_SPI1, &MySPIConfig );

    // Subscribe to the I2C BUS
    MyI2CHandle = adl_busSubscribe ( ADL_BUS_I2C, &MyI2CConfig );
    // Subscribe to the Parallel BUS
    MyParaHandle = adl_busSubscribe ( ADL_BUS_PARALLEL, &MyParaConfig
);

    // Write 5 bytes set to '0' on the SPI & I2C bus
    wm_memset ( WriteBuffer, WRITE_SIZE, 0 );
    adl_busWrite ( MySPIHandle, &AccessConfig, WRITE_SIZE, WriteBuffer
);
    adl_busWrite ( MyI2CHandle, &AccessConfig, WRITE_SIZE, WriteBuffer
);
    // Write 5 bytes set to '0' on the Parallel bus, at address 0
    adl_busDirectWrite ( MyParaHandle, 0, WRITE_SIZE, WriteBuffer );

    // Read 3 bytes from the SPI & I2C bus
    adl_busRead ( MySPIHandle, &AccessConfig, READ_SIZE, ReadBuffer );
    adl_busRead ( MyI2CHandle, &AccessConfig, READ_SIZE, ReadBuffer );

    // Read 3 bytes from the Parallel bus, at address 0
    adl_busDirectRead ( MyParaHandle, 0, READ_SIZE, ReadBuffer );

    // Unsubscribe from subscribed BUS
    adl_busUnsubscribe ( MySPIHandle );
    adl_busUnsubscribe ( MyI2CHandle );
```

```
    adl_busUnsubscribe ( MyParaHandle );
}
```

3.10 Error Management

3.10.1 Required Header File

The header file for the error functions is:

```
adl_errors.h
```

3.10.2 The adl_errSubscribe Function

This function subscribes to error service and gives an error handler: this allows the application to handle errors generated by ADL or by the **adl_errHalt** function. Errors generated by the Wavecom OS can not be handled by such 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 backtrace.
A backtrace is composed of the provided message, and a call stack "footprint" taken at the function call time. It is readable by the Target Monitoring Tool (Please refer to the Tools Manual (document 3) for more information).

Note:
That **ErrorID** below 0x0100 are for internal purpose so the application should only use **ErrorID** above 0x0100.

The reboot is performed once the handler has returned TRUE. In order to ensure the downloading of a new binary file after a fatal error has been detected, the Open AT® application software startup is done after a 20 seconds delay.
Therefore, in order not to miss any event, any application has to handle the case of a startup delay of 20 seconds.
Moreover, if the product reset is due to a fatal error (from Open AT® application, or from Wavecom OS), the **adl_main** function's **adlInitType** parameter will be set to the **ADL_INIT_REBOOT_FROM_EXCEPTION** value.
ADL may generates errors which will be handled by an error handler:

ErrorID	ADL function	Cause
ADL_ERR_MEM_GET	**adl_memGet**	The product ran out of heap memory, or the heap memory is composed of free blocks smaller than the required size.
ADL_ERR_MEM_RELEASE	**adl_memRelease**	The provided pointer was not provided by the **adl_memGet** function, or it was already released.
ADL_ERR_IO_ALLOCATE	**adl_ioSubscribe**	Abnormal error on Gpio subscription: should be signalized to Wavecom support.
ADL_ERR_IO_RELEASE	**adl_ioUnsubscribe**	Abnormal error on Gpio unsubscription: should be signalized to Wavecom support.
ADL_ERR_IO_READ	**adl_ioRead**	Abnormal error on Gpio read: should be signalized to Wavecom support.
ADL_ERR_IO_WRITE	**adl_ioWrite**	Abnormal error on Gpio write: should be signalized to Wavecom support.
ADL_ERR_FLH_READ	**adl_flhRead**	Abnormal error on Flash object read: should be signalized to Wavecom support.
ADL_ERR_FLH_DELETE	**adl_flhErase**	Abnormal error on Flash object erasure: should be signalized to Wavecom support.

- **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
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.10.3 The adl_errUnsubscribe Function

This function unsubscribes from error service. Errors generated by ADL or by the **adl_errHalt** function will no more be handled by the error handler.

- **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

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.10.4 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
                    const ascii *ErrorString );
```

- **Parameters**

ErrorID:
Error ID

ErrorString:
Error string to be provided to the error handler, and to be stored in the resulting backtrace if a fatal error is required.
Please note that only the string address is stored in the backtrace, so this parameter has not to be a pointer on a RAM buffer, but a constant string pointer. Moreover, the string will only be correctly displayed if the current application is still present in the Wireless CPU's flash memory. If the application is erased or modified, the string will not be correctly displayed when retrieving the backtraces.

3.10.5 The adl_errEraseAllBacktraces Function

Backtraces (caused by the adl_errHalt function, ADL or the Wavecom OS) are stored in the product non-volatile memory. A limited number of backtraces may be stored in memory (depending on each backtrace size, and other internal parameters stored in the same storage place). The **`adl_errEraseAllBacktraces`** function allows to free and re-initialize this storage place.

- **Prototype**

```
s32   adl_errEraseAllBacktraces ( void );
```

- **Returned value**

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
- OK if the function is successfully executed.

3.10.6 The adl_errStartBacktraceAnalysis Function

In order to retrieve backtraces from the product memory, a backtrace analysis process has to be started with the **`adl_errStartBacktraceAnalysis`** function.

- **Prototype**

```
s8    adl_errStartBacktraceAnalysis ( void );
```

- **Returned values**
  - A positive or null handle on success. This handle have to be used in the next **adl_errRetrieveNextBacktrace** function call. It will be valid until this function returns a ADL_RET_ERR_DONE code.
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if a backtrace analysis is already running.
  - ERROR if an unexpected internal error occurred.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Note :
Only one analysis may be running at a time. The **adl_errStartBacktraceAnalysis** function will return the ADL_RET_ERR_ALREADY_SUBSCRIBED error code if it is called while an analysis is currently running.

3.10.7 The adl_errGetAnalysisState Function

This function may be used in order to know the current backtrace analysis process state.

- **Prototype**

```
adl_errAnalysisState_e adl_errGetAnalysisState ( void );
```

- **Returned values**

The current backtraces analysis state, which use the type below:

```
typedef enum
{
    ADL_ERR_ANALYSIS_STATE_IDLE,          // No running analysis
    ADL_ERR_ANALYSIS_STATE_RUNNING        // An analysis is running
} adl_errAnalysisState_e;
```

3.10.8 The adl_errRetrieveNextBacktrace Function

This function allows the application to retrieve the next backtrace buffer stored in the product memory. The backtrace analysis may have been started first with the **adl_errStartBacktraceAnalysis** function.

- **Prototype**

```
s32 adl_errRetrieveNextBacktrace  (  u8    Handle
                                     u8 *  BacktraceBuffer
                                     u16   Size );
```

- **Parameters**

  Handle:
  Backtrace analysis handle, returned by the **adl_errStartBacktraceAnalysis** function.

  BacktraceBuffer:
  Buffer in which the next retrieved backtrace will be copied. This parameter may be set to **NULL** in order to know the next backtrace buffer required size.

  Size:
  Backtrace buffer size. If this size is not large enough, the ADL_RET_ERR_PARAM error code will be returned.

- **Returned values**

  - OK if the next stored backtrace was successfully copied in the BacktraceBuffer parameter.
  - The required size for next backtrace buffer if the BacktraceBuffer parameter is set to **NULL**.
  - ADL_RET_ERR_PARAM if the provided Size parameter is not large enough.
  - ADL_RET_ERR_NOT_SUBSCRIBED if the **adl_errStartBacktraceAnalysis** function was not called before.
  - ADL_RET_ERR_UNKNOWN_HDL if the provided Handle parameter is invalid.
  - ADL_RET_ERR_DONE if the last backtrace buffer has already been retrieved. The Handle parameter will now be unsubscribed and not usable any more with the **adl_errRetrieveNextBacktrace** function. A new analysis has to be started with the **adl_errStartBacktraceAnalysis** function.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Notes :

- Once retrieved, the backtrace buffers may be stored (separately or concatenated), in order to be sent (using the application's protocol/bearer choice) to a remote server or PC. Once retrieved as one or several files on a PC, this(these) one(s) may be read using the Target Monitoring Tool and the Serial Link Manager in order to decode the backtrace buffer(s). Please refer to the Tools Manual (document 3) [3]) in order to know how to process these files.

- If **`adl_errRetrieveNextBacktrace`** is used you have to retrieve all next backtraces. Otherwise it is impossible to retrieve the first backtraces.

3.11 SIM Service

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

3.11.1 Required Header File

The header file for the SIM related functions is:

`adl_sim.h`

3.11.2 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**

```
s32 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.

- **Returned value**

o ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
o OK if the function is successfully executed.

3.11.3 The adl_simUnsubscribe Function

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

- **Prototype**

```
s32   adl_simUnsubscribe  ( adl_simHdlr_f       Handler)
```

- **Parameters**

Handler:
Handler used with adl_SimSubscribe function.

- **Returned value**

o ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).
o OK if the function is successfully executed.

3.11.4 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;
```

3.12 Open SIM Access Service

The ADL Open SIM Access (OSA) service allows the application to handle APDU requests & responses with an external SIM card, connected through one of the Wireless CPU interfaces (UART, SPI, I2C).

Note:
The Open SIM Access feature has to be enabled on the Wireless CPU in order to make this service available.
The Open SIM Access feature state can be read thanks to the AT+WCFM=5 command response value: this feature state is represented by the bit 5 (00000020 in hexadecimal format).
Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU.

3.12.1 Required Header File

The header file for the OSA service definitions is:
adl_osa.h

3.12.2 The adl_osaSubscribe Function

This function allows the application to supply an OSA service handler, which will then be notified on each OSA event reception.
Moreover, by calling this function, the application requests the Wavecom firmware to close the local SIM connection, and to post SIM requests to the application from now.

- **Prototype**

```
s32 adl_osaSubscribe ( adl_osaHandler_f OsaHandler );
```

- **Parameters**

OsaHandler:
OSA service handler supplied by the application.
Please refer to **`adl_osaHandler_f`** type definition for more information (see paragraph 3.12.3).

- **Returned values**

o A positive or null value on success:
OSA service handle, to be used in further OSA service function calls.
A confirmation event will then be received in the service handler:
  - ADL_OSA_EVENT_INIT_SUCCESS if the local SIM connection was closed successfully,
  - ADL_OSA_EVENT_INIT_FAILURE if a Bluetooth SAP connection is running.

- A negative error value otherwise:
  - ADL_RET_ERR_PARAM on a supplied parameter error,
  - ADL_RET_ERR_NOT_SUPPORTED if the Open SIM access feature is not enabled on the Wireless CPU
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service was already subscribed (the OSA service can only be subscribed one time).
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.12.3 The adl_osaHandler_f call-back Type

Such a call-back function has to be supplied to ADL on the OSA service subscription. It will be notified by the service on each OSA event.

- **Prototype**

```
typedef void (* adl_osaHandler_f) ( adl_osaEvent_e         Event,
                                    adl_osaEventParam_u *  Param );
```

- **Parameters**

  Event:
  OSA service event identifier, using one of the following defined values.

Event Type	Use
ADL_OSA_EVENT_INIT_SUCCESS	*The OSA service has been successfully subscribed:* ▪ *The local SIM card has been shut down, and,* ▪ *From now on, all SIM requests will be posted to on the application through the OSA service.*
ADL_OSA_EVENT_INIT_FAILURE	*The OSA service subscription has failed:* ▪ *The Wireless CPU is already connected to a remote SIM through the Bluetooth SAP profile (the SAP connection has to be closed prior to subscribing to the OSA service).*
ADL_OSA_EVENT_ATR_REQUEST	*The application is notified with this event after the* ADL_OSA_EVENT_INIT_SUCCESS *one:* ▪ *The Wavecom firmware is required for the Answer To Reset data.* ▪ *The application has to reset the remote SIM card, and to get the ATR data in order to post it back to the Wavecom firmware through the* **adl_osaSendResponse** *function.*

Event Type	Use
ADL_OSA_EVENT_APDU_REQUEST	*This event is received by the application each time the Wavecom firmware has to send an APDU request to the SIM card. This request (notified to the application through the* **`Length`** *&* **`Data`** *parameters) has to be forwarded to the remote SIM by the application, and has to read the associated response in order to post it back to the Wavecom firmware through the* **`adl_osaSendResponse`** *function.*
ADL_OSA_EVENT_SIM_ERROR	*This event is notified to the application:* ▪ *If an error was notified to the Wavecom firmware in a SIM response (posted through the* **`adl_osaSendResponse`** *function), or,* ▪ *If the internal response time-out has elapsed (a request event was sent to the application, but no response was posted back to the Wavecom firmware).* *When this event is received, the OSA service is automatically un-subscribed and the Wavecom firmware resumes the local SIM connection.*
ADL_OSA_EVENT_CLOSED	*The application will receive this event after un-subscribing from the OSA service. The Wavecom firmware has resumed the local SIM connection.*

Param

Event parameters, using the following type:

```
typedef union
{
    adl_osaStatus_e   ErrorEvent;
    struct {
    {
        u16 Length;
        u8 * Data;
    }                 RequestEvent;
} adl_osaEventParam_u;
```

This union is used depending on the event type.

Event Type	Event Parameter
ADL_OSA_EVENT_INIT_SUCCESS	*Set to NULL*
ADL_OSA_EVENT_INIT_FAILURE	*Set to NULL*
ADL_OSA_EVENT_ATR_REQUEST	*Set to NULL*
ADL_OSA_EVENT_APDU_REQUEST	**RequestEvent** s*tructure set:* **Length:** *APDU request buffer length* **Data:** *APDU request data buffer address*
ADL_OSA_EVENT_SIM_ERROR	**ErrorEvent** *value set, according to the status previously sent back through the* **adl_osaSendResponse** *function, or set by the firmware on unsolicited errors.* *Please refer to the* **adl_osaSendResponse** *function description for more information.*
ADL_OSA_EVENT_CLOSED	*Set to NULL*

3.12.4 The adl_osaSendResponse Function

This function allows the application to post back ATR or APDU responses to the Wavecom firmware, after receiving an ADL_OSA_EVENT_ATR_REQUEST or ADL_OSA_EVENT_APDU_REQUEST event.

- **Prototype**

```
s32 adl_osaSendResponse ( s32             OsaHandle,
                          adl_osaStatus_e Status,
                          u16             Length,
                          u8 *            Data );
```

- **Parameters**

OsaHandle:
OSA service handle, previously returned by the **adl_osaSubscribe** function.

Status
Status to be supplied to the firmware, in response to an ATR or APDU request, using the following defined values.

Event Type	Use
ADL_OSA_STATUS_OK	*Response data buffer has been received from the SIM card.*
ADL_OSA_STATUS_CARD_NOT_ACCESSIBLE	*SIM card does not seem to be accessible (no response from the card).*
ADL_OSA_STATUS_CARD_REMOVED	*The SIM card has been removed.*
ADL_OSA_STATUS_CARD_UNKNOWN_ERROR	*Generic code for all other error cases.*

Length:
ATR or APDU request response buffer length, in bytes.
Note:
Should be set to 0 if the SIM card status is not OK.

Data:
ATR or APDU request response buffer address. This buffer content will be copied and sent by ADL to the Wavecom firmware.

Note:
Should be set to 0 if the SIM card status is not OK.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on a supplied parameter error.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied OSA handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the OSA service is not waiting for an APDU or ATR request response.

3.12.5 The adl_osaUnsubscribe Function

This function un-subscribes from the OSA service: the local SIM connection is resumed by the Wavecom firmware, and the application supplied handler is not any longer notified of OSA events.

- **Prototype**

```
s32 adl_osaUnsubscribe ( s32 OsaHandle );
```

- **Parameters**

OsaHandle:
OSA service handle, previously returned by the **`adl_osaSubscribe`** function.

- **Returned values**
  - OK on success.
    An ADL_OSA_EVENT_CLOSED confirmation event will then be received in the service handler.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied OSA handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.12.6 Example

This example simply demonstrates how to use the OSA service in a nominal case (error cases are not handled).

```
// Global variables


// OSA service handle
s32 OsaHandle;

// SIM request response data buffer length & address
u16 SimRspLen;
u8 * SimRspData;


  // OSA service handler
void MyOsaHandler ( adl_osaEvent_e Event, adl_osaEventParam_u * Param )
{
    // Switch on the event type
    switch ( Event )
    {
        case ADL_OSA_EVENT_ATR_REQUEST :
        case ADL_OSA_EVENT_APDU_REQUEST :
            // Reset the SIM card or transmit request
            // Get the related response data buffer
            // To be copied to SimRspLen & SimRspData global variables
            // Post back the response to the Wavecom firmware
            adl_osaSendResponse (     OsaHandle,ADL_OSA_STATUS_OK,
                                      SimRspLen, SimRspData );
        break;
    }
}
// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the OSA service
    OsaHandle = adl_osaSubscribe ( MyOsaHandler );
}
void MyFunction2 ( void )
{
    // Un-subscribes from the OSA service
    adl_osaUnsubscribe ( OsaHandle );
}
```

3.13 SMS Service

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

3.13.1 Required Header File

The header file for the SMS related functions is:

`adl_sms.h`

3.13.2 The adl_smsSubscribe Function

This function subscribes to the SMS service in order to receive SMSs 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 an 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.

Note:
Whatever is the handler's returned value, the incoming message has been internally processed by ADL; if it is read later via the +CMGR or +CMGL command, its status will be 'REC READ', instead of 'REC UNREAD'.

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 n 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 to receive SMSs:

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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.13.3 The adl_smsSend Function

This function sends an 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 to send SMSs:
ADL_SMS_MODE_PDU
to send a SMS in PDU mode.
ADL_SMS_MODE_TEXT
to send a SMS in Text mode.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if a parameter has an incorrect 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 an SMS (initialization not yet performed, or sending an SMS already in progress)
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.13.4 The adl_smsUnsubscribe Function

This function unsubscribes from the SMS service. The associated handler with provided handle will no longer receive SMS events.

- **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 an SMS
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.14 Message Service

ADL provides this service to allow applications to post and handle messages. Messages are used to exchange data between the different application components (application task, interruption handler...).

3.14.1 Required Header File

The header file for message-related functions is:

`adl_msg.h`

3.14.2 The adl_msgMailBox_e Type

This type defines the available mailboxes for the message service. Mailboxes are used to identify a message destination.

```
typedef enum
{
      ADL_MSG_MBX_OAT_TASK,
      ADL_MSG_MBX_OAT_TASK_LAST,  //Reserved for internal use
} adl_msgMailBox_e;
```

The **ADL_MSG_MBX_OAT_TASK** constant identifies the Open AT® application task.Other values are reserved for internal use.

3.14.3 The adl_msgIdComparator_e Type

This type defines the different message identifier comparison operators available.

```
typedef enum
{
      ADL_MSG_ID_COMP_EQUAL,
      ADL_MSG_ID_COMP_DIFFERENT,
      ADL_MSG_ID_COMP_GREATER,
      ADL_MSG_ID_COMP_GREATER_OR_EQUAL,
      ADL_MSG_ID_COMP_LOWER,
      ADL_MSG_ID_COMP_LOWER_OR_EQUAL,
      ADL_MSG_ID_COMP_LAST,                //Reserved for internal use
} adl_msgIdComparator_e;
```

The meaning of each comparison operator is defined below

Comparison Operator	Description
ADL_MSG_ID_COMP_EQUAL	*The two identifiers are equal.*
ADL_ MSG_ID_COMP_DIFFERENT	*The two identifiers are different.*
ADL_ MSG_ID_COMP_GREATER	*The received message identifier is greater than the subscribed message identifier.*
ADL_ MSG_ID_COMP_GREATER_OR_EQUAL	*The received message identifier is greater or equal to the subscribed message identifier.*
ADL_ MSG_ID_COMP_LOWER	*The received message identifier is lower than the subscribed message identifier.*
ADL_ MSG_ID_COMP_LOWER_OR_EQUAL	*The received message identifier is lower or equal to the subscribed message identifier.*

3.14.4 The adl_msgFilter_t Structure

This structure allows the application to define a message filter at service subscription time.

```
typedef struct
{
      u32                     MsgIdentifierMask;
      u32                     MsgIdentifierValue;
      adl__msgIdComparator_e  Comparator;
      adl__ctxID_e            Source;
} adl_msgFilter_t;
```

3.14.4.1 Structure Fields

The structure fields are defined below:

- **`MsgIdentifierMask`**:
  Bit mask to be applied to the incoming message identifier at reception time. Only the bits set to 1 in this mask will be compared for the service handlers notification. If the mask is set to 0, the identifier comparison will always match.

- **`MsgIdentifierValue`**:
  Message identifier value to be compared with the received message identifier. Only the bits filtered by the **`MsgIdentifierMask`** mask are significant.

- **`Comparator`**:
  Operator to be used for incoming message identifier comparison,

using the **adl_msgIdComparator_e** type. Please refer to the type description for more information (see § 3.14.3).

- **Source**:
  Required incoming message source context: the handler will be notified with messages received from this context. The **ADL_CTX_ALL** constant should be used if the application wishes to receive all messages, whatever the source context.

3.14.4.2 Filter Examples

- With the following filter parameters:
  MsgIdentifierMask = 0x0000F000
  MsgIdentifierValue = 0x00003000
  Comparator = ADL_MSG_ID_COMP_EQUAL
  Source = ADL_CTX_ALL
  the comparison will match if the message identifier fourth quartet is strictly equal to 3, whatever the other bit values, and whatever the source context.

- With the following filter parameters:
  MsgIdentifierMask = 0
  MsgIdentifierValue = 0
  Comparator = ADL_MSG_ID_COMP_EQUAL
  Source = ADL_CTX_ALL
  the comparison will always match, whatever the message identifier & the source context values

- With the following filter parameters:
  MsgIdentifierMask = 0xFFFF0000
  MsgIdentifierValue = 0x00010000
  Comparator = ADL_MSG_ID_COMP_GREATER_OR_EQUAL
  Source = ADL_CTX_HIGH_LEVEL_IRQ_HANDLER
  the comparison will match if the message identifier two most significant bytes are greater or equal to 1, and if the message was posted from high level interruption handler.

3.14.5 The adl_msgSubscribe Function

This function allows the application to receive incoming user-defined messages, sent from any application components (the application task itself or interruption handlers).

- **Prototype**

```
s32    adl_msgSubscribe(adl_mgsFilter_t_ * Filter,
                        adl_msgHandler_f  msgHandler);
```

- **Parameters**

  Filter:
  Identifier and source context conditions to check each message reception in order to notify the message handler. Please refer to the **adl_msgFilter_t** structure description for more information.

  MsgHandler:
  Application defined message handler, which will be notified each time a received message matches the filter conditions. Please refer to **adl_msgHandler_f** call-back type definition for more information.

- **Returned values**

  - A positive or null value on success:
    - Message service handle, to be used in further Message service functions calls.
  - A negative error value otherwise:
    - ADL_RET_ERR_PARAM if a parameter has an incorrect value.
    - ADL_RET_ERR_NO_MORE_HANDLES if there are no more free message service handles (up to 128 message filters can be subscribed).
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.14.6 The adl_msgHandler_f call-back Type

Such a call-back function has to be supplied to ADL through the **adl_msgSubscribe** interface in order to receive incoming messages. Messages will be received through this handler each time the supplied filter conditions are fulfilled.

- **Prototype**

```
typedef void (*adl_msgHandler_f) (      u32    MsgIdentifier,
                                        adl_ctxID_e  Source,
                                        u32    Length,
                                        void * Data );
```

- **Parameters**

  MsgIdentifier:
  Incoming message identifier.

  Source:
  Source context identifier from which the message was sent.

  Lenght:
  Message body length, in bytes. This length should be 0 if the message does not include a body.

Data:
Message body buffer address. This address should be NULL if the message does not include a body.

3.14.7 The adl_msgUnsubscribe Function

This function un-subscribes from a previously subscribed message filter. Associated message handler will no longer receive the filtered messages.

- **Prototype**

```
S32    adl_msgUnsubscribe  ( s32   MsgHandle );
```

- **Parameters**

MsgHandle:
Handle previously returned by the **adl_msgSubscribe** function.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.14.8 The adl_msgSend Function

This function allows the application to send a message at any time.

- **Prototype**

```
s32    adl_msgSend      ( adl_msgMailBox_e DestinationMbx,
                          u32              MessageIdentifier,
                          u32              Length,
                          void *           Data );
```

- **Parameters**

DestinationMbx:
Destination mailbox to which the message is to be posted.
The only value allowed is ADL_MSG_MBX_OAT_TASK.

MessageIdentifier:
The application defined message identifier. Message reception filters will be applied to this identifier before notifying the concerned message handlers.

Length:
Message body length, if any. Should be set to 0 if the message does not include a body.

Data:
Message body buffer address, if any. Should be set to 0 if the message does not include a body. This buffer data content will be copied into the message.

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

Note:
When a message is posted, the source context identifier is automatically set accordingly to the current context:

- If the message is sent from the application task, the source context identifier is set to ADL_CTX_OAT_TASK.
- If the message is sent from a low level interruption handler, the source context identifier is set to ADL_CTX_LOW_LEVEL_IRQ_HANDLER.
- If the message is sent from a high level interruption handler, the source context identifier is set to ADL_CTX_HIGH_LEVEL_IRQ_HANDLER.

3.14.9 Example

This example simply demonstrates how to use the message service in a nominal case (error cases are not handled).

```
// Global variables & constants

// Message filter definition
const adl_msgFilter_t MyFilter =
{
  0xFFFF0000,                       // Compare only the 2 MSB
  0x00010000,                       // Compare with 1
  ADL_MSG_ID_COMP_GREATER_OR_EQUAL, // Msg ID has to be >= 1
  ADL_CTX_OAT_TASK                  //  Application  task  incoming  msg
only
};

// Message service handle
s32 MyMsgHandle;

...

// Incoming message handler
void MyMsgHandler (  u32 MsgIdentifier, adl_ctxID_e Source,
                     u32 Length, void * Data )
{
  // Message processing
}

...
{
    // Subscribe to the message service
    MyMsgHandle = adl_busSubscribe ( &MyFilter, MyMsgHandler );

    // Send an empty message
    adl_msgSend ( 0x00010055, ADL_MSG_MBX_OAT_TASK, 0, NULL );

    // Unsubscribe from the message service
    adl_msgUnsubscribe ( MyMsgHandle );
}
```

3.15 Call Service

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

3.15.1 Required Header File

The header file for the call related functions is:

`adl_call.h`

3.15.2 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 ; each event is received according to an "event type", which can be :
  - MO *(Mobile Originated call related event)*
  - MT *(Mobile Terminated call related event)*
  - CMD *(Incoming AT command related event)*

Event / Call ID	Description	Type
ADL_CALL_EVENT_RING_VOICE / 0	*if voice phone call*	*MT*
ADL_CALL_EVENT_RING_DATA / 0	*if data phone call*	*MT*
ADL_CALL_EVENT_NEW_ID / X	*if wind: 5,X*	*MO*
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*	*MO MT*
ADL_CALL_EVENT_ALERTING / 0	*if wind: 2*	*MO*
ADL_CALL_EVENT_NO_CARRIER / 0	*phone call failure, 'NO CARRIER'*	*MO MT*
ADL_CALL_EVENT_NO_ANSWER / 0	*phone call failure, no answer*	*MO*
ADL_CALL_EVENT_BUSY / 0	*phone call failure, busy*	*MO*

Event / Call ID	Description	Type
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.*	*MO*
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*	*MT*
ADL_CALL_EVENT_CIEV / Speed	*OK response after a performed call setup; in data call setup case, the connection <Speed> (in bps) is also provided*	
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)*	*MO MT*
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.*	*MO*
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*	*MT*
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)*	*MO MT*
ADL_CALL_EVENT_AUDIO_OPENNED / 0	*if +WIND: 9*	*MO MT*
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*	*MT*
ADL_CALL_EVENT_RING_GPRS / 0	*if GPRS phone call*	*MT*

Event / Call ID	Description	Type
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.*	*CMD*
ADL_CALL_EVENT_SETUP_ERROR_NO_SIM / 0	*A call setup (from embedded or external application) has failed (no SIM card inserted)*	*MO*
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)*	*MO*
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)*	*MO*

The events returned by this handler are defined below:

Event	Description
ADL_CALL_FORWARD	*the call event shall be sent to the external application* *On unsolicited events, these ones will be forwarded to all opened ports.* *On responses events, these ones will be forwarded only on the port on which the request was executed.*
ADL_CALL_NO_FORWARD	*the call event shall not be sent to the external application*
ADL_CALL_NO_FORWARD_ATH	*the call event shall not be sent to the external application and the application shall terminate the call by sending an 'ATH' command.*
ADL_CALL_NO_FORWARD_ATA	*the call event shall not be sent to the external application and the application shall answer the call by sending an 'ATA' command.*

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM on parameter error
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.15.3 The adl_callSetup Function

This function just runs the **`adl_callSetupExt`** one on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_callSetupExt`** description for more information). Please note that events generated by the **`adl_callSetup`** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.15.4 The adl_callSetupExt Function

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

- **Prototype**

```
s8      adl_callSetupExt ( ascii *          PhoneNb,
                           u8               Mode,
                           adl_port_e       Port );
```

- **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

  Port:
  Port on which to run the call setup command. When setup return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM on parameter error (bad value, or unavailable port)
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.15.5 The adl_callHangup Function

This function just runs the **`adl_callHangupExt`** one on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_callHangupExt`** description for more information). Please note that events generated by the **`adl_callHangup`** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.15.6 The adl_callHangupExt Function

This function hangs up the phone call.

- **Prototype**

```
s8    adl_callHangupExt ( adl_port_e Port );
```

- **Parameters**

  Port:
  Port on which to run the call hang-up command. When hang-up return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**
  - OK on success

- ADL_RET_ERR_PARAM on parameter error (unavailable port)
- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.15.7 The adl_callAnswer Function

This function just runs the **`adl_callAnswerExt`** one on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_callAnswerExt`** description for more information). Please note that events generated by the **`adl_callAnswer`** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.15.8 The adl_callAnswerExt Function

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

- **Prototype**

```
s8    adl_callAnswerExt ( adl_port_e Port );
```

- **Parameters**

  Port:
  Port on which to run the call hang-up command. When hang-up return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_PARAM on parameter error (unavailable port)
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.15.9 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.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.16 GPRS Service

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

3.16.1 Required Header File

The header file for the GPRS related functions is:

```
adl_gprs.h
```

3.16.2 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*

Event / Call ID	Description
ADL_GPRS_EVENT_ME_CLASS_B	*If the ME has changed to class B*
ADL_GPRS_EVENT_ME_CLASS_CG	*If the ME has changed to class CG*
ADL_GPRS_EVENT_ME_CLASS_CC	*If the ME has changed to class CC*
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 was successful on the Cid X*
ADL_GPRS_EVENT_DEACTIVATE_OK_FROM_EXT / X	*If the deactivation requested by the external application was successful on the Cid X*
ADL_GPRS_EVENT_ANSWER_OK	*If the acceptance of the incoming PDP activation with adl_gprsAct()was successful*
ADL_GPRS_EVENT_ANSWER_OK_FROM_EXT	*If the acceptance of the incoming PDP activation by the external application was successful*
ADL_GPRS_EVENT_ACTIVATE_OK / X	*If the activation requested with adl_gprsAct() on the Cid X was successful*
ADL_GPRS_EVENT_GPRS_DIAL_OK_FROM_EXT / X	*If the activation requested by the external application with 'ATD*99' (PPP dialing) was successful on the Cid X*
ADL_GPRS_EVENT_ACTIVATE_OK_FROM_EXT / X	*If the activation requested by the external application on the Cid X was successful*
ADL_GPRS_EVENT_HANGUP_OK_FROM_EXT	*If the rejection of the incoming PDP activation by the external application was successful*
ADL_GPRS_EVENT_DEACTIVATE_KO / X	*If the deactivation requested with adl_gprsDeact() on the Cid X failed*
ADL_GPRS_EVENT_DEACTIVATE_KO_FROM_EXT / X	*If the deactivation requested by the external application on the Cid X failed*

Event / Call ID	Description
ADL_GPRS_EVENT_ACTIVATE_KO_FROM_EXT / X	*If the activation requested by the external application on the Cid X failed*
ADL_GPRS_EVENT_ACTIVATE_KO / X	*If the activation requested with adl_gprsAct() on the Cid X failed*
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()was successful*
ADL_GPRS_EVENT_SETUP_KO / X	*If the set up of the Cid X with adl_gprsSetup() failed*
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 for registration.*

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

The possible returned values for this handler are defined below:

Event	Description
ADL_GPRS_FORWARD	*the event shall be sent to the external application.* *On unsolicited events, these one be forwarded to all opened ports.* *On responses events, these one be forwarded only on the port on which the request was executed.*
ADL_GPRS_NO_FORWARD	*the event is not sent to the external application*
ADL_GPRS_NO_FORWARD_ATH	*the event is not sent to the external application and the application will terminate the incoming activation request by sending an 'ATH' command.*
ADL_GPRS_NO_FORWARD_ATA	*the event is not sent to the external application and the application will accept the incoming activation request by sending an 'ATA' command.*

- **Returned values for adl_gprsSubscribe**

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

Error value	Description
ADL_RET_ERR_PARAM	*In case of parameter error*
ADL_RET_ERR_SERVICE_LOCKED	*If the function was called from a low level interruption handler (the function is forbidden in this context).*

3.16.3 The adl_gprsSetup Function

This function runs the **`adl_gprsSetupExt`** on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_gprsSetupExt`** description for more information). Please note that events generated by the **`adl_gprsSetup`** will not be able to be forwarded to any external port, since the setup command runs on the Open AT® port.

3.16.4 The adl_gprsSetupExt Function

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

- **Prototype**

```
s8 adl_gprsSetupExt (  u8                     Cid,
                       adl_gprsSetupParams_t  Params,
                       adl_port_e             Port );
```

- **Parameters**

  Cid:
  The Cid of the PDP context to setup (integer value between 1 and 4).

  Params:
  The parameters to set up are contained in the following type:

```
typedef struct
{
 ascii* APN;
 ascii* Login;
 ascii* Password;
 ascii* FixedIP;
 bool   HeaderCompression;
 bool   DataCompression;
}adl_gprsSetupParams_t;
```

  - APN:
    Address of the Provider GPRS Gateway (GGSN)
    maximum 100 bytes string

  - Login:
    GPRS account login
    maximum 50 bytes string

  - Password:
    GPRS account password
    maximum 50 bytes string

  - FixedIP:
    Optional fixed IP address of the MS (used only if not set to NULL)
    maximum 15 bytes string

  - HeaderCompression:
    PDP header compression option (enabled if set to TRUE)

  - DataCompression:
    PDP data compression option (enabled if set to TRUE)

  Port:
  Port on which to run the PDP context setup command. Setup return events are received in the GPRS event handler. If the application requires

ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**

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

Error value	Description
ADL_RET_ERR_PARAM	*parameter error: bad Cid value or unavailable port*
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	*problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*if 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*
ADL_RET_ERR_SERVICE_LOCKED	*If the function was called from a low level interruption handler (the function is forbidden in this context).*

3.16.5 The adl_gprsAct Function

This function just runs the **`adl_gprsActExt`** one on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_gprsActExt`** description for more information). Please note that events generated by the **`adl_gprsAct`** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.16.6 The adl_gprsActExt Function

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

- **Prototype**

```
s8 adl_gprsActExt ( u8          Cid,
                    adl_port_e  Port );
```

- **Parameters**

  Cid:
  The Cid of the PDP context to activate (integer value between 1 and 4).

  Port:
  Port on which to run the PDP context activation command. Activation return events are received in the GPRS event handler.If the application

requires ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**

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

  Possible error values are:

Error Value	Description
ADL_RET_ERR_PARAM	*parameters error: bad Cid value or unavailable port*
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	*problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*if 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*
ADL_RET_ERR_SERVICE_LOCKED	*If the function was called from a low level interruption handler (the function is forbidden in this context).*

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

3.16.7 The adl_gprsDeact Function

This function runs the **`adl_gprsDeactExt`** on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. **`adl_gprsDeactExt`** description for more information). Please note that events generated by the **`adl_gprsDeact`** will not be able to be forwarded to any external port, since the setup command runs on the Open AT® port.

3.16.8 The adl_gprsDeactExt Function

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

- **Prototype**

```
s8 adl_gprsDeactExt (  u8           Cid
                       adl_port_e   Port );
```

- **Parameters**

  Cid:
  The Cid of the PDP context to deactivate (integer value between 1 and 4).

  Port:
  Port on which to run the PDP context deactivation command. Deactivation return events are received in the GPRS event handler.If the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- **Returned values**

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

  Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	*parameters error: bad Cid value or unavailable port*
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	*problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*if 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*
ADL_RET_ERR_SERVICE_LOCKED	*If the function was called from a low level interruption handler (the function is forbidden in this context).*

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 Wireless CPU lock.

3.16.9 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 (integer value between 1 and 4).

  Infos:
  Information of the activated PDP context is contained in the following type:

```
typedef struct
{
    u32 LocalIP; // Local IP address of the MS
    u32 DNS1;    // First DNS IP address
    u32 DNS2;    // Second DNS IP address
    u32 Gateway; // Gateway IP address
}adl_gprsInfosCid_t;
```

  This parameter fields will be set only if the GPRS session is activated ; otherwise, they all will be set to 0.

- **Returned values**

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

Error value	Description
ADL_RET_ERR_PARAM	*parameters error: bad Cid value*
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	*problem to set up the Cid (the CID is already activated)*
ADL_NO_GPRS_SERVICE	*if 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*

3.16.10 The adl_gprsUnsubscribe Function

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

- **Prototype**

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

- **Parameters**

  Handler:
  Handler used with adl_gprsSubscribe function.

- **Returned values**

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

Error value	Description
ADL_RET_ERR_PARAM	*parameter error*
ADL_RET_ERR_UNKNOWN_HDL	*the provided handler is unknown*
ADL_RET_ERR_NOT_SUBSCRIBED	*the service is not subscribed*
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*
ADL_RET_ERR_SERVICE_LOCKED	*If the function was called from a low level interruption handler (the function is forbidden in this context).*

3.16.11 The adl_gprsIsAnIPAddress Function

This function checks if the provided string is a valid IP address. Valid IP address strings arebased on the "a.b.c.d" format, where a, b, c & d are integer values between 0 and 255.

- **Prototype**

```
bool adl_gprsIsAnIPAddress ( ascii * AddressStr );
```

- **Parameters**

  AddressStr:
  IP address string to check.

- **Returned values**

  - TRUE if the provided string is a valid IP address one, and FALSE otherwise.
  - NULL & empty string ("") are not considered as a valid IP address.

3.16.12 Example

This example just demonstrates how to use the GPRS service in a nominal case (error cases are not handled).

Complete examples using the GPRS service are also available on the SDK (Ping_GPRS sample).

```
// Global variables
adl_gprsSetupParams_t MyGprsSetup;
adl_gprsInfosCid_t    InfosCid;

// GPRS event handler
s8 MyGprsEventHandler ( u16 Event, u8 CID )
{
    // Trace event
    TRACE (( 1, "Received GPRS event %d/%d", Event, CID ));

    // Switch on event
    switch ( Event )
    {
        case ADL_GPRS_EVENT_SETUP_OK :
            TRACE (( 1, "PDP Ctxt Cid %d Setup OK", CID ));
            // Activate the session
            adl_gprsAct ( 1 );
        break;

        case ADL_GPRS_EVENT_ACTIVATE_OK :
            TRACE (( 1, "PDP Ctxt %d Activation OK", CID ));
            // Get context information
            adl_gprsGetCidInformations ( 1, &InfosCid );
            // De-activate the session
            adl_gprsDeAct ( 1 );
        }
        break;

        case ADL_GPRS_EVENT_DEACTIVATE_OK :
            TRACE (( 1, " PDP Ctxt %d De-activation OK", CID ));
            // Un-subscribe from GPRS event handler
           adl_gprsUnsubscribe ( MyGprsEventHandler );
        break;
    }

    // Forward event
    return ADL_GPRS_FORWARD;
}

// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Fill Setup structure
    MyGprsSetup.APN = "myapn";
    MyGprsSetup.Login = "login";
    MyGprsSetup.Password = "password";
```

```
    MyGprsSetup.FixedIP = NULL;
    MyGprsSetup.HeaderCompression = FALSE;
    MyGprsSetup.DataCompression = FALSE;

    // Subscribe to GPRS event handler
    adl_gprsSubscribe ( MyGprsEventHandler );

    // Set up the GPRS context
    adl_gprsSetup ( 1, MyGprsSetup );
}
```

3.17 Semaphore ADL Service

The ADL Semaphore service allows the application to handle the semaphore resources supplied by the Open AT® OS.
Semaphores are used to synchronize processes between the application task and high level interruption handlers.

Note:
Semaphores cannot be used in a low level interruption handler context.

3.17.1 Required Header File

The header file for the Semaphore service definitions is:

```
adl_sem.h
```

3.17.2 The adl_semSubscribe Function

This function allows the application to reserve and initialize a semaphore resource.

- **Prototype**

```
s32 adl_semSubscribe ( u16 SemCounter );
```

- **Parameters**

  SemCounter:
  Semaphore inner counter initialization value (reflects the number of times the semaphore can be consumed before the calling task must be suspended).

- **Returned values**
  - A positive or null value on success:
    - Semaphore service handle, to be used in further service function calls.
  - A negative error value otherwise:
    - ADL_RET_ERR_NO_MORE_SEMAPHORES when there are no more free semaphore resources.
      Up to 7 semaphore resources are available.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.17.3 The adl_semConsume Function

This function allows the application to reduce the required semaphore counter by one.
If this counter value falls under zero, the calling execution context is suspended until the semaphore is produced from another context.

- **Prototype**

```
s32 adl_semConsume ( s32 SemHandle );
```

- **Parameters**

  SemHandle:
  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL when the supplied handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.17.4 The adl_semConsumeDelay Function

This function allows the application to reduce the required semaphore counter by one.
If this counter value falls under zero, the calling execution context is suspended until the semaphore is produced from another context.
Moreover, if the semaphore is not produced during the supplied time-out duration, the calling context is automatically resumed.

- **Prototype**

```
s32 adl_semConsumeDelay (    s32 SemHandle,
                             u32 TimeOut );
```

- **Parameters**

  SemHandle:
  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

  Timeout:
  Time to wait before resuming context when the semaphore is not produced (must not be 0). Time measured is in 18.5 ms ticks.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL when the supplied handle is unknown.
  - ADL_RET_ERR_PARAM when a supplied parameter value is wrong.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

- **Exceptions**

  The following exception must be generated on this function call.
  - RTK exception 206 if the semaphore has been consumed too many times.
    A semaphore can be consumed a number of times equal to its initial value + 256.

3.17.5 The adl_semProduce Function

This function allows the application to increase the required semaphore counter by one.
If this counter value gets above zero, the execution contexts that were suspended due to using this semaphore are resumed.

- **Prototype**

```
s32 adl_semProduce ( s32 SemHandle );
```

- **Parameters**

  SemHandle:
  Semaphore service handle, previously returned by the **`adl_semSubscribe`** function.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

- **Exceptions**

  The following exception must be generated on this function call.
  - RTK exception 133 if the semaphore has been produced too many times.
    A semaphore can be produced until its inner counter reaches its initial value.

3.17.6 The adl_semUnsubscribe Function

This function allows the application to unsubscribe from the Semaphore service, in order to release the previously reserved resource.
A semaphore can be unsubscribed only if its inner counter value is the initial one (the semaphore has been produced as many times as it has been consumed).

- **Prototype**

```
s32 adl_semUnsubscribe ( s32 SemHandle );
```

- **Parameters**

  SemHandle:
  Semaphore service handle, previously returned by the **`adl_semSubscribe`** function.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL when the supplied handle is unknown
  - ADL_RET_ERR_BAD_STATE when the semaphore inner counter value is different from the initial value.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.17.7 Example

This example shows how to use the Semaphore service in a nominal case (error cases are not handled).

```
// Global variable: Semaphore service handle
s32 MySemHandle;



// Somewhere in the application code, used as high level interruption
handler
void MyHighLevelHandler ( void )
{
    // Produces the semaphore, to resume the application task context
    adl_semProduce ( MySemHandle );
}



// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the semaphore service
    MySemHandle = adl_semSubscribe ( 0 );

    // Consumes the semaphore, with a 37 ms time-out delay
    adl_semConsumeDelay ( MySemHandle, 2 );

    // Consumes the semaphore: has to be produced from another context
    adl_semConsume ( MySemHandle );

void MyFunction2 ( void )
{
    // Un-subscribes from the semaphore service
    adl_semUnsubscribe ( MySemHandle );
}
```

3.18 Application Safe Mode Service

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

3.18.1 Required Header File

The header file for the Application safe mode service is:

`adl_safe.h`

3.18.2 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:
  Additionnal options for +WDWL command subscription. This command is at least subscribed in ACTION and READ mode. Please see adl_atCmdSubscribe API for more details about these options.

  WOPENopt:
  Additionnal options for +WOPEN command subscription. This command is at least subscribed in READ, TEST and PARAM mode, with minimum of one mandatory parameter. Please see adl_atCmdSubscribe API for more details about 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_AD_GET_SIZE,   // AT+WOPEN=6 command
   ADL_SAFE_CMD_WOPEN_AD_SET_SIZE,   // AT+WOPEN=6,<size> 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 the commands used for adl_atCmdSubscribe API.

If the Handler returns FALSE, the command will not be forwarded to the Wavecom OS.
If the Handler returns TRUE, the command will be processed by the Wavecom OS, 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
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.18.3 The adl_safeUnsubscribe Function

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

- **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
- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.18.4 The adl_safeRunCommand Function

This function allows +WDWL or +WOPEN command with any standard syntax.

- **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 cannot be suspended (the function has no effect). Please note that the current Open AT® application process is suspended immediately on the adl_safeRunCommand process; if there is any code after this function call, it will be executed only when the process is resumed.

  RspHandler:
  Response handler to get command results. All responses are subscribed and the command is executed on the Open AT® virtual port. Instead of providing a response handler, a port identifier may be specified (using `adl_port_e` type): the command will be executed on this port, and the resulting responses sent back on this port.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter has an incorrect value
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.19 AT Strings Service

This service provides APIs to process AT standard response strings.

3.19.1 Required Header File

The header file for the AT strings service is:

`adl_str.h`

3.19.2 The adl_strID_e Type

All predefined AT strings for this service are defined in the following type:

```
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:"
      ADL_STR_CMEE,                             // "+CMEE:"
      ADL_STR_CREG,                             // "+CREG:"
      ADL_STR_CGREG,                            // "+CGREG:"
      ADL_STR_CRC,                              // "+CRC:"
      ADL_STR_CGEREP,                           // "+CGEREP:"
      // Last string ID
      ADL_STR_LAST
} adl_strID_e;
```

3.19.3 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.

3.19.4 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.

3.19.5 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 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.

3.19.6 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 caution:

The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application. This means that the embedded application will have to release the returned pointer.

3.19.7 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. Depending on the 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 caution:**
>
> The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application. This means that the embedded application will have to release the returned pointer.

3.20 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 (Wavecom OS updates, new Open AT® applications or E2P configuration files) in order to be installed later on the product.

The default storage size is 768 Kbytes. It may be configured with the AT+WOPEN command (Please refer to the AT commands interface guide (document [2]) for more information).

This storage size has to be set to the maximum (about 1.2 Mbytes) in order to have enough place to store a Wavecom OS update.

Caution:

Any A&D size change will lead to an area format process (some additional seconds on start-up, all A&D cells data will be erased).

Legal mention:

The Download Over The Air feature enables the Wavecom OS to be remotely updated.

The downloading and OS updating processes have to be activated and managed by an appropriate Open AT® based application to be developed by the customer. The security of the whole process (request for update, authentication, encryption, etc) has to be managed by the customer under his own responsibility. Wavecom shall not be liable for any issue related to any use by customer of the Download Over The Air feature.

Wavecom AGREES AND THE CUSTOMER ACKNOWLEDGES THAT THE SDK Open AT® IS PROVIDED "AS IS" BY Wavecom WITHOUT ANY WARRANTY OR GUARANTEE OF ANY KIND.

3.20.1 Required Header File

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

```
adl_ad.h
```

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

  When subscribing, the size is rounded up to the next multiple of 4.

- **Returned values**

  - A positive or null value on success:
    - The A&D cell handle on success, to be used on further A&D API functions calls,
  - A negative error value:
    - ADL_RET_ERR_ALREADY_SUBSCRIBED if the cell is already subscribed;
    - ADL_AD_RET_ERR_OVERFLOW if there is not enough allocated space,
    - 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 an 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.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.3 The adl_adUnsubscribe Function

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

- **Prototype**

```
s32   adl_adUnsubscribe ( s32  CellHandle );
```

- **Parameters**

  CellHandle:
  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_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.4 The adl_adEventSubscribe Function

This function allows the application to provide ADL with an event handler to be notified with A&D service related events.

- **Prototype**

```
s32   adl_adEventSubscribe ( adl_adEventHdlr_f Handler );
```

- **Parameters**

  Handler:
  Call-back function provided by the application. Please refer to next chapter for more information.

- **Returned values**

  - A positive or null value on success:
    - A&D event handle, to be used in further A&D API functions calls,
  - A negative error value:
    - ADL_RET_ERR_PARAM if the **`Handler`** parameter is invalid,
    - ADL_RET_ERR_NO_MORE_HANDLES if the A&D event service has been subscribed more than 128 times.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Notes:
In order to format or re-compact the A&D storage volume, the **`adl_adEventSubscribe`** function has to be called before the **`adl_adFormat`** or the **`adl_adRecompact`** functions.

3.20.5 The adl_adEventHdlr_f Call-back Type

This call-back function has to be provided to ADL through the **`adl_adEventSubscribe`** interface, in order to receive A&D related events.

- **Prototype**

```
typedef void (*adl_adEventHdlr_f) ( adl_adEvent_e  Event,
                                    u32            Progress );
```

- **Parameters**

  Event:
  Event is the received event identifier. The events (defined in the **`adl_adEvent_e`** type) are described in the table below.

Event	Meaning
ADL_AD_EVENT_FORMAT_INIT	*The **adl_adFormat** function has been called by an application (a format process has just been required).*
ADL_AD_EVENT_FORMAT_PROGRESS	*The format process is on going. Several "progress" events should be received until the process is completed.*
ADL_AD_EVENT_FORMAT_DONE	*The format process is over. The A&D storage area is now usable again. All cells have been erased, and the whole storage place is available.*
ADL_AD_EVENT_RECOMPACT_INIT	*The adl_adRecompact function has been called by an application (a re-compaction process has been required).*
ADL_AD_EVENT_RECOMPACT_PROGRESS	*The re-compaction process is on going. Several "progress" events should be received until the process is completed.*
ADL_AD_EVENT_RECOMPACT_DONE	*The re-compaction process is over: the A&D storage area is now usable again. The space previously used by deleted cells is now free.*
ADL_AD_EVENT_INSTALL	*The adl_adInstall function has been called by an application (an install process has just been required and the Wireless CPU is going to reset).*

  Progress:
  On ADL_AD_EVENT_FORMAT_PROGRESS & ADL_AD_EVENT_RECOMPACT_PROGRESS events reception, this parameter is the process progress ratio (considered as a percentage).
  On ADL_AD_EVENT_FORMAT_DONE & ADL_AD_EVENT_RECOMPACT_DONE events reception, this parameter is set to 100%.
  Otherwise, this parameter is set to 0.

3.20.6 The adl_adEventUnsubscribe Function

This function allows the Open AT® application to unsubscribe from the A&D events notification.

- **Prototype**

```
s32 adl_adEventUnsubscribe ( s32 EventHandle );
```

- **Parameters**

  EventHandle:
  Handle previously returned by the **adl_adEventSubscribe** function.

- **Returned values**

  - OK on success,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_NOT_SUBSCRIBED if no A&D event handler has been subscribed,
  - ADL_RET_ERR_BAD_STATE if a format or re-compaction process is currently running with this event handle.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.7 The adl_adWrite Function

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

- **Prototype**

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

- **Parameters**

  CellHandle:
  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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.8 The adl_adInfo Function

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

- **Prototype**

```
s32  adl_adInfo  (  s32              CellHandle
                    adl_adInfo_t *   Info );
```

- **Parameters**

  CellHandle:
  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 or an undefined size.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.9 The adl_adFinalise Function

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

- **Prototype**

```
s32  adl_adFinalise ( s32  CellHandle );
```

- **Parameters**

  CellHandle:
  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.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.10 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   ( s32  CellHandle );
```

- **Parameters**

  CellHandle:
  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_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

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

3.20.11 The adl_adInstall Function

This function installs the content of the requested cell, if it is a **`.DWL`** file. This file should be an Open AT® application, an EEPROM configuration file, an XModem downloader binary file, or a Wavecom OS binary file.
Caution:

This API resets the Wireless CPU on success.

- **Prototype**

```
s32  adl_adInstall  ( s32  CellHandle );
```

- **Parameters**

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

- **Returned values**

  - Wireless CPU 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.
    Before the Wireless CPU reset, all subscribed event handlers (if any) will receive the ADL_AD_EVENT_INSTALL event, in order to let them perform last operations.
  - ADL_RET_ERR_BAD_STATE if the cell is not finalized,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle was not subscribed.

- ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Note for RTE:
In RTE mode, calling this API will cause a message box display, prompting the user for installing the desired A&D cell content or not (see Figure 10: A&D cell content install window).

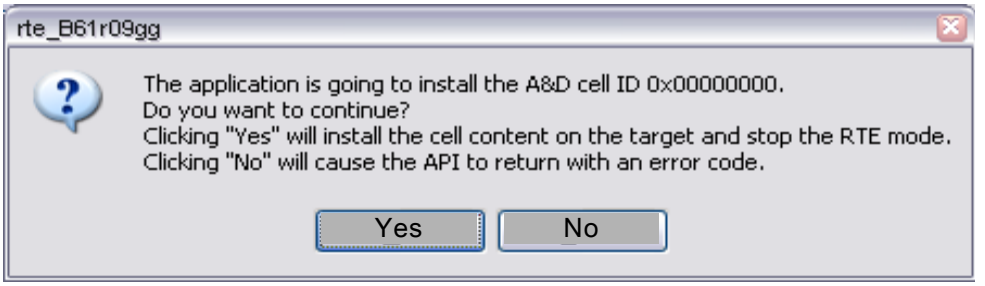


Figure 10: A&D cell content install window

If the user selects "No", the API will fail and return the ADL_AD_RET_ERROR code.
If the user selects "Yes", the cell content is installed, the Wireless CPU resets, and the RTE mode is automatically closed.

3.20.12 The adl_adRecompact Function

This function starts the re-compaction process, which will release the deleted cell spaces and IDs.

Caution:

If some A&D cells are deleted, and the recompaction process is not performed regularly, the deleted cell space will not be freed.

- **Prototype**

```
s32 adl_adRecompact ( s32 EventHandle );
```

- **Parameters**

  EventHandle:

  Event handle previously returned by the **adl_adEventSubscribe** function. The associated handler will receive the re-compaction process events sequence.

- **Returned values**
  - OK on success. Event handlers will receive the following event sequence:
    - ADL_AD_EVENT_RECOMPACT_INIT just after the process is launched,
    - ADL_AD_EVENT_RECOMPACT_PROGRESS several times, indicating the process progression,
    - ADL_AD_EVENT_RECOMPACT_DONE when the process is completed.
  - ADL_RET_ERR_BAD_STATE if a re-compaction or format process is currently running,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_NOT_SUBSCRIBED if no A&D event handler has been subscribed,
  - ADL_AD_RET_ERR_NOT_AVAILABLE if there is no A&D space available on the product.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.13 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 the 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_AD_RET_ERR_NEED_RECOMPACT if a power down or a reset occurred when a re-compaction process was running. The application has to launch the adl_adRecompact function before using any other A&D service function.
  - ADL_RET_ERR_PARAM on parameter error.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.14 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.
  The list is ordered by cell id values, from the lowest to the highest.

  Caution:

  The list memory is allocated by the `adl_adGetCellList` function and has to 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.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.15 The adl_adFormat Function

This function re-initializes the A&D storage volume. It is only allowed if there is currently no subscribed cells, or if there are no currently running re-compaction or format process.

Important caution:

All the A&D storage cells will be erased by this operation. The A&D storage format process can take several seconds.

- **Prototype**

```
s32 adl_adFormat (s32 EventHandle );
```

- **Parameters**

  EventHandle:
  Event handle previously returned by the **`adl_adEventSubscribe`** function. The associated handler will receive the format process events sequence.

- **Returned values**
  - OK on success. Event handlers will receive the following event sequence:
    - ADL_AD_EVENT_FORMAT_INIT just after the process is launched,
    - ADL_AD_EVENT_FORMAT_PROGRESS several times, indicating the process progression,
    - ADL_AD_EVENT_FORMAT_DONE once the process is done,
  - ADL_RET_ERR_UNKNOWN_HDL if the handle is unknown,
  - ADL_RET_ERR_NOT_SUBSCRIBED if no A&D event handler has been subscribed,
  - 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, or if a re-compaction or format process is already running.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.20.16 Example

This example demonstrates how to use the A&D service in a nominal case (error cases not handled).
Complete examples using the A&D service are also available on the SDK (DTL Application_Download sample, generic Download library sample).

```
// Global variables & constants

// Cell & event handles
s32 MyADCellHandle;
s32 MyADEventHandle;

// Info & state structure
adl_adInfo_t Info;
adl_adState_t State;

// A&D event handler
void MyADEventHandler ( adl_adEvent_e Event, u32 Progress )
{
    // Check event
    switch ( Event )
    {
        case ADL_AD_EVENT_RECOMPACT_DONE :
        case ADL_AD_EVENT_FORMAT_DONE :
            // The process is over
            TRACE (( 1, "Format/Recompact process over…" ));
        break;
    }
}

...

// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    u8 DataBuffer [ 10 ];

    // Get state
    adl_adGetState ( &State );

    // Subscribe to the A&D event service
    MyADEventHandle = adl_adEventSubscribe ( MyADEventHandler );

    // Subscribe to an A&D cell
    MyADCellHandle = adl_adSubscribe ( 0x00000000, 20 );

    // Write data buffer
    wm_memset ( DataBuffer, 10, 0 );
    adl_adWrite ( MyADCellHandle, 10, DataBuffer );

    // Get info
    adl_adInfo ( MyADCellHandle, &Info );
```

```
    // Install the cell (will fail, not finalized)
    adl_adInstall ( MyADCellHandle );

    // Finalize the cell
    adl_adFinalise ( MyADCellHandle );

    // Delete the cell
    adl_adDelete ( MyADCellHandle );

    // Launch the re-compaction process
    adl_adRecompact ( MyADEventHandle );

    // Launch the format process
    // (will fail, re-compaction process is running)
    adl_adFormat ( MyADEventHandle );

    // Unsubscribe from the A&D event service
    // (will fail, re-compaction process is running)
    adl_adEventUnsubscribe ( MyADEventHandler );
}
```

3.21 AT/FCM IO Ports Service

ADL applications may use this service to be informed about the product AT/FCM IO ports states.

3.21.1 Required Header File

The header file for the AT/FCM IO Ports service is:

`adl_port.h`

3.21.2 AT/FCM IO Ports

AT Commands and FCM services can be used to send and receive AT Commands or data blocks, to or from one of the product ports. These ports are linked either to product physical serial ports (as UART1 / UART2 / USB ports), or virtual ports (as Open AT® virtual AT port, GSM CSD call data port, GPRS session data port or Bluetooth virtual ports).
AT/FCM IO Ports are identified by the type below :

```
typedef enum
{
    ADL_PORT_NONE,
    ADL_PORT_UART1,
    ADL_PORT_UART2,
    ADL_PORT_USB,

    ADL_PORT_UART1_VIRTUAL_BASE       = 0x10,
    ADL_PORT_UART2_VIRTUAL_BASE       = 0x20,
    ADL_PORT_USB_VIRTUAL_BASE         = 0x30,
    ADL_PORT_BLUETOOTH_VIRTUAL_BASE   = 0x40,
    ADL_PORT_GSM_BASE                 = 0x50,
    ADL_PORT_GPRS_BASE                = 0x60,
    ADL_PORT_OPEN_AT_VIRTUAL_BASE     = 0x80
} adl_port_e;
```

The available ports are described hereafter :

- ADL_PORT_NONE
  Not usable
- ADL_PORT_UART1
  Product physical UART 1
  Please refer to the AT+WMFM command documentation to know how to open/close this product port.
- ADL_PORT_UART2
  Product physical UART 2
  Please refer to the AT+WMFM command documentation to know how to open/close this product port.
- ADL_PORT_USB
  Product physical USB port (reserved for future products)
- ADL_PORT_UART1_VIRTUAL_BASE
  Base ID for 27.010 protocol logical channels on UART 1
  Please refer to AT+CMUX command & 27.010 protocol documentation to know how to open/close such a logical channel.
- ADL_PORT_UART2_VIRTUAL _BASE
  Base ID for 27.010 protocol logical channels on UART 2
  Please refer to AT+CMUX command & 27.010 protocol documentation to know how to open/close such a logical channel.
- ADL_PORT_USB_VIRTUAL _BASE
  Base ID for 27.010 protocol logical channels on USB link (reserved for future products)
- ADL_PORT_BLUETOOTH_VIRTUAL _BASE
  Base ID for connected Bluetooth peripheral virtual port.
  ONLY USABLE WITH THE FCM SERVICE
  Please refer to the Bluetooth AT commands documentation to know how to connect, and how to open/close such a virtual port.
- ADL_PORT_GSM_BASE
  Virtual Port ID for GSM CSD data call flow
  ONLY USABLE WITH THE FCM SERVICE

Please note that this port will be considered as always available (no OPEN/CLOSE events for this port ; adl_portIsAvailable function will always return TRUE)

- ADL_PORT_GPRS_BASE
  Virtual Port ID for GPRS data session flow
  ONLY USABLE WITH THE FCM SERVICE
  Please note that this port will be considered as always available (no OPEN/CLOSE events for this port ; adl_portIsAvailable function will always return TRUE) if the GPRS feature is supported on the current product.
- ADL_PORT_OPEN_AT_VIRTUAL_BASE
  Base ID for AT commands contexts dedicated to Open AT® applications
  ONLY USABLE WITH THE AT COMMANDS SERVICE
  This port is always available, and is opened immediately at the product's start-up. This is the default port where are executed the AT commands sent by the AT Command service.

3.21.3 Ports Test Macros

Some ports & events test macros are provided. These macros are defined hereafter.

- ADL_PORT_IS_A_SIGNAL_CHANGE_EVENT(_e)
  Returns TRUE if the event "_e" is a signal change one, FALSE otherwise.
- ADL_PORT_GET_PHYSICAL_BASE(_port)
  Extracts the physical port identifier part of the provided "_port". E.g. if used on a 27.010 virtual port identifier based on the UART 2, this macro will return ADL_PORT_UART2*.*
- ADL_PORT_IS_A_PHYSICAL_PORT(_port)
  Returns TRUE if the provided "_port" is a physical output based one (E.g. UART1, UART2 or 27.010 logical ports), FALSE otherwise.
- ADL_PORT_IS_A_PHYSICAL_OR_BT_PORT(_port)
  Returns TRUE is the provided "_port" is a physical output or a bluetooth based one, FALSE otherwise.
- ADL_PORT_IS_AN_FCM_PORT(_port)
  Returns TRUE if the provided "_port" is able to handle the FCM service (i.e. all ports except the Open AT® virtual base ones), FALSE otherwise.
- ADL_PORT_IS_AN_AT_PORT(_port)
  Returns TRUE if the provided "_port" is able to handle AT commands services (i.e. all ports except the GSM & GPRS virtual base ones), FALSE otherwise.

3.21.4 The adl_portSubscribe Function

This function subscribes to the AT/FCM IO Ports service in order to receive specific ports related events.

- **Prototype**

```
s8      adl_portSubscribe ( adl_portHdlr_f PortHandler );
```

- **Parameters**

PortHandler:
Port related events handler defined using the following type:

```
typedef void (*adl_portHdlr_f) ( adl_portEvent_e Event,
                                 adl_port_e Port,
                                 u8 State );
```

The events received by this handler are defined below:

ADL_PORT_EVENT_OPENED
*Informs the ADL application that the specified **Port** is now opened. According to its type, it may now be used with either AT Commands service or FCM service.*

ADL_PORT_EVENT_CLOSED
*Informs the ADL application that the specified **Port** is now closed. It is not usable anymore with neither AT Commands service nor FCM service.*

ADL_PORT_EVENT_DSR_STATE_CHANGE
*Informs the ADL application that the specified **Port** DSR signal state has changed to the new **State** value (0/1). This event will be received by all subscribers which have started a polling process on the specified **Port** DSR signal with the adl_portStartSignalPolling function.*

ADL_PORT_EVENT_CTS_STATE_CHANGE
*Informs the ADL application that the specified **Port** CTS signal state has changed to the new **State** value (0/1). This event will be received by all subscribers which have started a polling process on the specified **Port** CTS signal with the adl_portStartSignalPolling function.*

The handler **Port** parameter uses the **adl_port_e** type described above.
The handler **State** parameter is set only for the ADL_PORT_EVENT_XXX_STATE_CHANGE events.

- **Returned values**

  - 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 (the service is able to process up 127 subscriptions).
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.21.5 The adl_portUnsubscribe Function

This function unsubscribes from the AT/FCM IO Ports service. The related handler will not receive ports related events any more. If a signal polling process was started only for this handle, it will be automaticaly stopped.

- **Prototype**

```
s8      adl_portUnsubscribe ( u8 Handle );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_portSubscribe function.

- **Returned values**

  - OK on success ;
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown ;
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.21.6 The adl_portIsAvailable Function

This function checks if the required port is currently opened or not.

- **Prototype**

```
bool    adl_portIsAvailable ( adl_port_e Port );
```

- **Parameters**

  Port:
  Port from which to require the current state.

- **Returned values**

  - TRUE if the port is currently opened ;
  - FALSE if the port is closed, or if it does not exists.

Notes

- The function will always return TRUE on the ADL_PORT_GSM_BASE port ;
- The function will always return TRUE on the ADL_PORT_GPRS_BASE port if the GPRS feature is enabled (always FALSE otherwise).

3.21.7 The adl_portGetSignalState Function

This function returns the required port signal state.

- **Prototype**

```
s8      adl_portGetSignalState ( adl_port_e Port,
                                 adl_portSignal_e Signal );
```

- **Parameters**

  Port:
  Port from which to require the current signal state. Only physical output related ports (UARTX & USB ones, used as physical ports, or with the 27.010 protocol) may be used with this function.

  Signal:
  Signal from which to query the current state, based on the following type :

```
typedef enum
{
    ADL_PORT_SIGNAL_CTS,
    ADL_PORT_SIGNAL_DSR,

    ADL_PORT_SIGNAL_LAST
} adl_portSignal_e;
```

  Signal are detailed below:

  ADL_PORT_SIGNAL_CTS
  Required port CTS input signal : physical pin in case of a physical port (UARTX), emulated logical signal in case of a 27.010 logical port.

  ADL_PORT_SIGNAL_DSR
  Required port DSR input signal : physical pin in case of a physical port (UARTX), emulated logical signal in case of a 27.010 logical port.

- **Returned values**
  - The signal state (0/1) on success ;
  - ADL_RET_ERR_PARAM on parameter error;
  - ADL_RET_ERR_BAD_STATE if the required port is not opened.

3.21.8 The adl_portStartSignalPolling Function

This function starts a polling process on a required port signal for the provided subscribed handle.
Only one polling process can run at a time. A polling process is defined on one port, for one or several of this port's signals.
It means that this function may be called several times on the same port in order to monitor several signals ; the polling time interval is set up by the first function call (polling tme parameters are ignored or further calls). If the

function is called several times on the same port & signal, additional calls will be ignored.

Once a polling process is started on a port's signal, this one is monitored : each time this signal state changes, a ADL_PORT_EVENT_XXX_STATE_CHANGE event is sent to all the handlers which have required a polling process on it.

Whatever is the number of requested signals and subscribers to this port polling process, a single cyclic timer will be internally used for this one.

- **Prototype**

```
s8      adl_portStartSignalPolling (u8 Handle,
                                    adl_port_e Port,
                                    adl_portSignal_e Signal,
                                    u8 PollingTimerType,
                                    u32 PollingTimerValue );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_portSubscribe function.

  Port:
  Port on which to run the polling process. Only physical output related ports (UARTX & USB ones, used as physical ports, or with the 27.010 protocol) may be used with this function.

  Signal:
  Signal to monitor while the polling process. See the adl_portGetSignalState function for information about the available signals.

  PollingTimerType:
  PollingTimerValue parameter value's unit. The allowed values are defined below :

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

  This parameter is ignored on additional function calls on the same port.

  PollingTimerValue:
  Polling time interval (uses the PollingTimerType parameter's value unit).

  This parameter is ignored on additional function calls on the same port.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_PARAM on parameter error ;
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown ;
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed ;
  - ADL_RET_ERR_BAD_STATE if the required port is not opened ;
  - ADL_RET_ERR_ALREADY_SUBSCRIBED if a polling process is already running on another port.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.21.9 The adl_portStopSignalPolling Function

This function stops a running polling process on a required port signal for the provided subscribed handle.

The associated handler will not receive the ADL_PORT_EVENT_XXX_STATE_CHANGE events related to this signal port anymore.

The internal polling process cyclic timer will be stopped as soon as the last subscriber to the current running polling process has call this function.

- **Prototype**

```
s8      adl_portStopSignalPolling ( u8 Handle,
                                    adl_port_e Port,
                                    adl_portSignal_e Signal );
```

- **Parameters**

  Handle:
  Handle previously returned by the adl_portSubscribe function.

  Port:
  Port on which the polling process to stop is running.

  Signal:
  Signal on which the polling process to stop is running.

- **Returned values**
  - OK on success ;
  - ADL_RET_ERR_PARAM on parameter error ;
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown ;
  - ADL_RET_ERR_NOT_SUBSCRIBED if the service is not subscribed ;
  - ADL_RET_ERR_BAD_STATE if the required port is not opened ;
  - ADL_RET_ERR_BAD_HDL if there is no running polling process for this Handle / Port / Signal combination.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.22 RTC Service

ADL provides a RTC service to access to the Wireless CPU's inner RTC, and to process time related data.

3.22.1 Required Header File

The header file for the RTC functions is:

`adl_rtc.h`

3.22.2 RTC service Types

3.22.2.1 The adl_rtcTime_t Structure

The following structure is used by the Wavecom OS in order to retrieve the current RTC time:

```
typedef struct
{
    u16  Year;            // Year (Four digits)
    u8   Month;           // Month (1-12)
    u8   Day;             // Day of the month (1-31)
    u8   Hour;            // Hour (0-23)
    u8   Minute;          // Minute (0-59)
    u8   Second;          // Second (0-59)
    u8   Pad;             // Not used    u16 SecondFracPart;      //
Second fractional part
} adl_rtcTime_t;
```

Second fractional part step is the `ADL_RTC_SECOND_FRACPART_STEP` constant. This field most significant bit is not used (values are in the [0 – 0x7FFF] range).

3.22.2.2 The adl_rtcTimeStamp_t Structure

The following structure may be used to perform arithmetic operations on time data:

```
typedef struct
{
    u32 TimeStamp;        // Seconds elapsed since 1st January 1970
    u16 SecondFracPart;   // Second fractional part
} adl_rtcTimeStamp_t;
```

The timestamp uses the Unix format (seconds elapsed since the 1st January 1970).

Second fractional part step is the `ADL_RTC_SECOND_FRACPART_STEP` constant. This field most significant bit is not used (values are in the [0 – 0x7FFF] range).

3.22.2.3 Constants

RTC service constants are defined below.

Constant	Value	Use
ADL_RTC_SECOND_FRACPART_STEP	30.5	*Second fractional part step value (in µs) ; The real value is 1/215*
ADL_RTC_DAY_SECONDS	24x60x60	*Seconds count in a day*
ADL_RTC_HOUR_SECONDS	60x60	*Seconds count in an hour*
ADL_RTC_MINUTE_SECONDS	60	*Seconds count in a minute*
ADL_RTC_MS_US	1000	*µseconds count in a millisecond*

3.22.2.4 Macros

RTC service macros are defined below.

Macro	Parameter	Use
ADL_RTC_GET_TIMESTAMP_SECONDS(_t)	adl_rtcTimeStamp_t structure	*Timestamp seconds part (0-59)*
ADL_RTC_GET_TIMESTAMP_MINUTES(_t)	adl_rtcTimeStamp_t structure	*Timestamp minutes part (0-59)*
ADL_RTC_GET_TIMESTAMP_HOURS(_t)	adl_rtcTimeStamp_t structure	*Timestamp hours part (0-23)*
ADL_RTC_GET_TIMESTAMP_DAYS(_t)	adl_rtcTimeStamp_t structure	*Timestamp days part*
ADL_RTC_GET_TIMESTAMP_MS(_t)	adl_rtcTimeStamp_t structure	*Timestamp milliseconds part (0-999)*
ADL_RTC_GET_TIMESTAMP_US(_t)	adl_rtcTimeStamp_t structure	*Timestamp microseconds part (0-999)*

These macros may be used in order to extract durations parts from a given timestamp ; the logical equations below are always true:

```
_t.TimeStamp == ADL_RTC_GET_TIMESTAMP_SECONDS(_t) +
                ADL_RTC_GET_TIMESTAMP_MINUTES(_t) * ADL_RTC_MINUTE_SECONDS +
                ADL_RTC_GET_TIMESTAMP_HOURS(_t)   * ADL_RTC_HOUR_SECONDS +
                ADL_RTC_GET_TIMESTAMP_DAYS(_t)    * ADL_RTC_DAY_SECONDS

_t.SecondFracPart * ADL_RTC_SECOND_FRACPART_STEP ==
                ADL_RTC_GET_TIMESTAMP_MS(_t) * ADL_RTC_MS_US +
                ADL_RTC_GET_TIMESTAMP_US(_t)
```

3.22.3 The adl_rtcGetTime Function

This function retrieves the current RTC time structure.

- **Prototype**

```
s32   adl_rtcGetTime ( adl_rtcTime_t * TimeStructure );
```

- **Parameters**

TimeStructure:
Retrieved current time structure.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM if the parameter is incorrect.

3.22.4 The adl_rtcConvertTime Function

This function is able to convert RTC time structure to timestamp structure, and timestamp structure to RTC time structure.

- **Prototype**

```
s32   adl_rtcConvertTime ( adl_rtcTime_t *          TimeStructure,
                           adl_rtcTimeStamp_t *     TimeStamp,
                           adl_rtcConvert_e         Conversion );
```

- **Parameters**

TimeStructure:
Input / output RTC time structure

TimeStamp:
Input / output timestamp structure

Conversion:
Conversion mode, using the type below:

```
typedef enum
{
    ADL_RTC_CONVERT_TO_TIMESTAMP,
    ADL_RTC_CONVERT_FROM_TIMESTAMP
} adl_rtcConvert_e;
```

ADL_RTC_CONVERT_TO_TIMESTAMP
This mode converts the TimeStructure parameter to the TimeStamp parameter.

ADL_RTC_CONVERT_FROM_TIMESTAMP
This mode converts the TimeStamp parameter to the TimeStructure parameter.

- **Returned values**
  - OK on success,
  - ERROR if conversion failed (internal error),
  - ADL_RET_ERR_PARAM if one parameter value is incorrect.

3.22.5 The adl_rtcDiffTime Function

This function calculates the difference between two timestamp structures.

- **Prototype**

```
s32   adl_rtcDiffTime (    adl_rtcTimeStamp_t *    TimeStamp1,
                           adl_rtcTimeStamp_t *    TimeStamp2,
                           adl_rtcTimeStamp_t *    Result );
```

- **Parameters**

  TimeStamp1:
  First timestamp

  TimeStamp2:
  Second timestamp
  Result:
  Time difference between the two provided timestamps.

- **Returned values**

  - O on success, and if TimeStamp1 equals to TimeStamp2,
  - 1 on success, and if TimeStamp1 is greater than TimeStamp2,
  - -1 on success, and if TimeStamp2 is greater than TimeStamp1,
  - ADL_RET_ERR_PARAM if one parameter value is incorrect.

3.22.6 Example

This example demonstrates how to use the RTC service in a nominal case (error cases are not handled).

Complete examples using the RTC service are also available on the SDK (generic Download library sample).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    adl_rtcTime_t Time1, Time2;
    adl_rtcTimeStamp_t Stamp1, Stamp2, Diff;
    s32 Way;

    // Get time
    adl_rtcGetTime ( &Time1 );
    adl_rtcGetTime ( &Time2 );

    // Convert to time stamps
    adl_rtcConvertTime ( &Time1, &Stamp1, ADL_RTC_CONVERT_TO_TIMESTAMP
);
    adl_rtcConvertTime ( &Time2, &Stamp2, ADL_RTC_CONVERT_TO_TIMESTAMP
);

    // Reckon time difference
    Way = adl_rtcDiffTime ( &Stamp1, &Stamp2, &Diff );
}
```

3.23 IRQ Service

The ADL IRQ service allows interruption handlers to be defined.
These handlers are usable with other services (External Interruption Pins, SCTU) to monitor specific interruption sources.

Interruption handlers are running in specific execution contexts of the application. Please refer to the Execution Contexts Service for more information (§ 3.25).

Note:

- The Real Time Enhancement feature has to be enabled on the Wireless CPU in order to make this service available.
- The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value: this feature state is represented by the bit 4 (00000010 in hexadecimal format)
- Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU.

3.23.1 Required Header File

The header file for the IRQ functions is:

```
adl_irq.h
```

3.23.2 The adl_irqID_e Type

This type defines the interruption sources that the service is able to monitor.

```
typedef enum
{
      ADL_IRQ_ID_EXTINT0,
      ADL_IRQ_ID_EXTINT1,
      ADL_IRQ_ID_SCTU1,

      ADL_IRQ_ID_LAST   // Reserved for internal use
} adl_irqID_e;
```

The ADL_IRQ_ID_EXTINTX constants identify interruption sources raised by the External Interrupt Pins service.

The ADL_IRQ_ID_SCTUX constants identify interruption sources raised by the SCTU service.

3.23.3 The adl_irqNotificationLevel_e Type

This type defines the notification level of a given interruption handler.

```
typedef enum
{
      ADL_IRQ_NOTIFY_LOW_LEVEL,
      ADL_IRQ_NOTIFY_HIGH_LEVEL,

      ADL_IRQ_NOTIFY_LAST     // Reserved for internal use
} adl_irqNotificationLevel_e;
```

The ADL_IRQ_NOTIFY_LOW_LEVEL constant allows low level interruption handlers to be defined.

The ADL_IRQ_NOTIFY_HIGH_LEVEL constant allows high level interruption handlers to be defined.
For more information on specific high and low level handlers behavior, please refer to the Execution Context Service description (§ 3.25).

3.23.4 The adl_irqPriorityLevel_e Type

This structure supplies interruption handlers with data related to the interruption source.

```
typedef struct
{
      ADL_IRQ_PRIORITY_LOW_LEVEL,
      ADL_IRQ_PRIORITY_HIGH_LEVEL,

      ADL_IRQ_PRIORITY_LAST   // Reserved for internal use
} adl_irqPriorityLevel_e;
```

The ADL_IRQ_PRIORITY_LOW_LEVEL constant allows a low priority interruption handler to be defined.

The ADL_IRQ_PRIORITY_HIGH_LEVEL constant allows a low priority interruption handler to be defined.

The priority level of an handler allows the notification order to be set in case of event conflict:

- A low priority level handler cannot be interrupted by other low priority level handlers.
- A high priority level handler cannot be interrupted by other high or low priority level handlers;
- A low priority level handler can be interrupted by any high priority level handler.

3.23.5 The adl_irqEventData_t Structure

This structure supplies interruption handlers with data related to the interruption source.

```
typedef struct
{
      union
      {
          void * LowLevelOuput;
          void * HighLevelInput;
      } UserData;
      void * SourceData;
} adl_irqEventData_t;
```

3.23.5.1 The UserData Field

This field allows the application to exchange data between low level and high level interruption handlers.

The **`LowLevelOuput`** member address has to be modified by the low level handler before it returns. This address will be supplied to the high level handler through the **`HighLevelInput`** member.

3.23.5.2 The SourceData Field

This field provides interruption handlers (which have used the ADL_IRQ_OPTION_AUTO_READ option at subscription time) with interruption source specific data. Please refer to each interruption source related service for more information about this field data structure.

3.23.6 The adl_irqSubscribe Function

This function allows the application to supply an interruption handler, to be used later in Interruption source related service subscription.

- **Prototype**

```
s32 adl_irqSubscribe (     adl_irqHandler_f        IrqHandler,
      adl_irqNotifyLevel_e NotificationLevel,
      adl_irqPriorityLevel_e  PriorityLevel,
      u32                  Options );
```

- **Parameter**

  IrqHandler:
  Interruption handler supplied by the application.
  Please refer to **`adl_irqHandler_f`** type definition for more information (see § 3.23.7).

  NotificationLevel:
  Interruption handler notification level; allows the supplied handler to be identified as a low level or a high level one.

Please refer to **`adl_irqNotifyLevel_e`** type definition for more information (see § 3.23.3).

PriorityLevel:
Interruption handler priority level; allows the supplied handler to be set as a low priority or a high priority one.
Please refer to **`adl_irqPriorityLevel_e`** type definition for more information (see § 3.23.4).

Options:
Interruption handler notification options.
A bitwise OR combination of the following options has to be used:

Options	Comment
ADL_IRQ_OPTION_AUTO_READ	*When the interruption occurs, the source related information structure is automatically read by the service, and supplied to the interruption handler.* *Please refer to* **`adl_irqHandler_f`** *type and interruption source related services description for more information (see §.3.23.7)* *Note:* *When used with a high level interruption handler, this option allows the application to get the source related information structure read at interruption time.*
ADL_IRQ_OPTION_PRE_ACKNOWLEDGEMENT	*Set by default* *This option forces ADL acknowledge the interruption source prior to calling the low level handler. It is ignored for high level handlers subscription.* *Please note, that in this configuration, the low level interruption handler should be re-entrant, as, if another interruption occurs before the first one's process is not over, the low level handler will be called one more time.*

Options	Comment
ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT	*This option forces ADL acknowledge the interruption source after having called the low level handler. It is ignored for high level handlers subscription.* *Please note that in this configuration, the low level interruption handler will always be called sequentially (it does not have to be re-entrant).* ***Caution:*** ***This option is not supported by all interruption source services; please refer to these services de scription for more information.***

- **Returned values**
  - A positive or null value on success:
    - IRQ service handle, to be used in further IRQ & interruption source services function calls.
  - A negative error value otherwise:
    - ADL_RET_ERR_PARAM on a supplied parameter error.
    - ADL_RET_ERR_NOT_SUBSCRIBED if a low or high level handler subscription is required while the associated context call stack size was not supplied by the application (please referto the Mandatory Service description (§ 3.1)).
    - ADL_RET_ERR_NOT_SUPPORTED if the Real Time enhancement feature is not enabled on the Wireless CPU.
    - ADL_RET_ERR_BAD_STATE if the function is called in RTE mode.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Note:
The IRQ service will always return an error code in RTE mode (the service is not supported in this mode). The only way to debug interruption handlers is to use the Debug service to send traces readable by the Target Monitoring Tool. Use of the IRQ service should be flagged in order to make an application working correctly in RTE.

3.23.7 The adl_irqHandler_f Call-back Type

This type has to be used by the application in order to provide ADL with an interruption hander.

- **Prototype**

```
typedef bool (*adl_irqHandler_f) (adl_irqID_e      Source,
                                  adl_irqNotifyLevel_e
      NotificationLevel,
                                  adl_irqEventData_t * Data );
```

- **Parameter**

  Source:
  Interruption source identifier.
  Please refer to **adl_irqID_e** type definition for more information. (see § 3.23.2).

  NotificationLevel:
  Interruption handler current notification level.
  Please refer to **adl_irqNotifyLevel_e** type definition for more information (see § 3.23.3).

  Data:
  I Interruption handler input/output data field.
  Please refer to **adl_irqEventData_e** type definition for more information. (see § 3.23.5).

- **Returned values**

  - Not relevant for high level interruption handlers.
  - For low level interruption handlers:
    - A TRUE return value will force ADL to call the subscribed high level handler for this interruption source.
    - A FALSE return value will force ADL not to call any high level handler for this interruption source.

  Note:

  For low level interruption handlers, 1 ms can be considered as a maximum latency time before being notified with the interruption source event.

3.23.8 The adl_irqUnsubscribe Function

This function allows the application to un-subscribe from the interruption service. The associated handler will no longer be notified of interruption events.

- **Prototype**

  ```
  s32 adl_irqUnsubscribe ( s32 IrqHandle );
  ```

- **Parameter**

  IrqHandle:
  Interruption service handle, previously returned by the **`adl_irqSubscribe`** function.

- **Returned values**

  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the supplied handle is still used by an interruption source service.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.23.9 Example

This example simply demonstrates how to use the interruption service in a nominal case (error cases are not handled).

```
// Global variable: IRQ service handle
s32 MyIRQHandle;



// Interruption handler
bool MyIRQHandler (adl_irqID_e Source, adl_irqNotifyLevel_
                   e NotificationLevel, adl_irqEventData_t * Data )
{
    // Interruption process…
    // Notify the High Level handler, if any
    return TRUE;
}



// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribe to the IRQ service

MyIRQHandle = adl_irqSubscribe ( MyIRQHandler, ADL_IRQ_NOTIFY_LOW_LEVEL
,
                ADL_IRQ_PRIORITY_HIGH_LEVEL,   ADL_IRQ_OPTION_AUTO_READ
);

    // Interruption source service subscription
}
    // Un-subscribe from the IRQ service
    adl_irqUnsubscribe ( MyIRQHandle );
}
```

3.24 SCTU Service

ADL provides this service (System Controller Timer Unit (SCTU)) to handle the Wireless CPU hardware timerconfiguration and interruptions.

SCTU are hardware timers, able to raise interruptions on comparisons and/or overflow. SCTU block settings depend on Wireless CPU type.

Q2686 Wireless CPU

SCTU block number	Clock rate	Prescaler width	Counter width	Comparator channels
1	13 MHz	8 bits	16 bits	4

Q2687 Wireless CPU

SCTU block number	Clock rate	Prescaler width	Counter width	Comparator channels
1	13 MHz	8 bits	16 bits	4

Q268X Wireless CPUs SCTU architecture is shown in the diagram below.

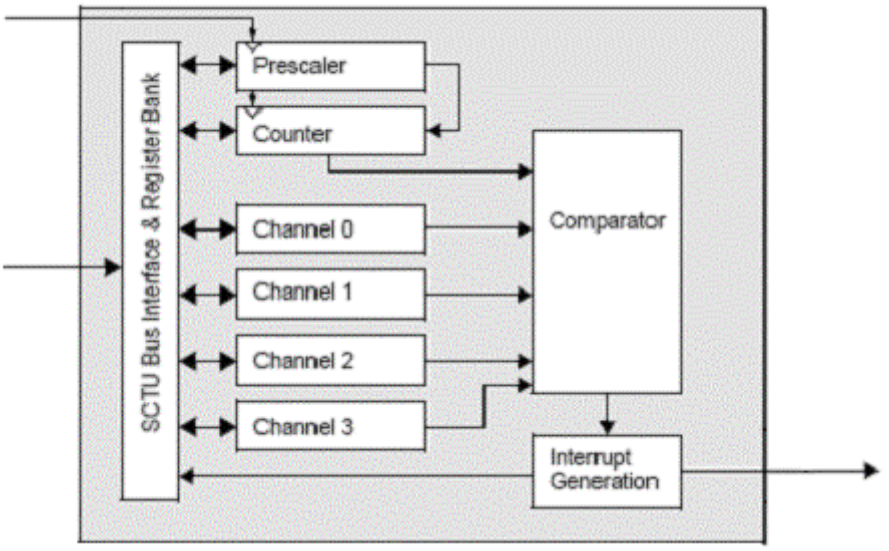


Figure 11: Q268X Wireless CPUs SCTU Architecture

Global SCTU functioning is described below:

- A prescaler is cadenced by the internal clock signal
  (the prescaler reload value is configurable at service subscription)
- The counter is clocked by the prescaler overflow signal
  (the counter reload value is configurable at service subscription)
- The comparator channels are configurable separately (comparison values & interruption generation flag)
- The counter overflow interruption flag is also configurable
- Once the service is subscribed, the timer can be started & stopped by the application.

3.24.1 Required Header File

The header file for the SCTU functions is:

```
adl_sctu.h
```

3.24.2 The adl_sctuInfo_t Structure

This structure allow the application to get the interruption source(s) identifier(s) when an SCTU interruption occurs. When an interruption handler is subscribed with the **ADL_IRQ_OPTION_AUTO_READ** option and connected to the SCTU service, the **SourceData** field in the **adl_irqEventData_t** input parameter of this handler will have to be cast to this type in order to handle correctly the information.

```
typedef struct
{
      u8 SourceMask;    // Interruption sources mask
} adl_sctuInfo_t;
```

- **Parameter**

  SourceMask:
  SCTU interruption source bits mask: this will be a bit-wise OR of one or several of the constants below.

Constant	Use
ADL_SCTU_IT_SRC_OVERFLOW	*Counter overflow interruption source*
ADL_SCTU_IT_SRC_COMP_CHANNEL_0	*Comparator channel 0 interruption source*
ADL_SCTU_IT_SRC_COMP_CHANNEL_1	*Comparator channel 1 interruption source*
ADL_SCTU_IT_SRC_COMP_CHANNEL_2	*Comparator channel 2 interruption source*
ADL_SCTU_IT_SRC_COMP_CHANNEL_3	*Comparator channel 3 interruption source*

3.24.3 The adl_sctuSubscribe Function

This function allows the application to subscribe to the SCTU service. Moreover, by calling this function, the application powers on the required Wireless CPU System Controller Timer Unit block.

- **Prototype**

```
s32 adl_sctuSubscribe (    adl_sctuID_e            BlockID
                           s32                     LowLevelIrqHandle
                           s32
      HighLevelIrqHandle
                           adl_sctuSettings_t *    Settings );
```

- **Parameters**

BlockID:
SCTU block identifier to be subscribed, using the type below.

```
typedef enum
{
    ADL_SCTU_BLOCK1,        // SCTU block 1
    ADL_SCTU_BLOCK_LAST     // Internal use only
} adl_sctuID_e;
```

LowLevelIrqHandle:
Low level interruption handler identifier, previously returned by the **adl_irqSubscribe** function.
This parameter is optional if the **HighLevelIrqHandle** parameter is supplied.

HighLevelIrqHandle:
High level interruption handler identifier, previously returned by the **adl_irqSubscribe** function.
This parameter is optional if the **LowLevelIrqHandle** parameter is supplied.

Settings:
SCTU block configuration, to be defined using the following structure:

```
typedef struct
{
    u8 PrescalerReload;     // Prescaler reload value
    u8 Pad;                 // Reserved
    u16 CounterReload;      // Counter reload value
    bool OverflowITFlag;    // Overflow interruption flag
} adl_sctuSettings_t;
```

PrescalerReload
Value to be reloaded in the prescaler register on each overflow (ie. when the prescaler reaches the 0xFF value).
The prescaler time period can be calculated by the formula:

PrescalerTime = (0xFF – PrescalerReload + 1) / ClockFrequency

Eg. With a 13 MHz frequency & a prescaler reload value set to 80, the prescaler time period is ~13.54 µs ((255-80+1)/13000000).

OverflowITFlag:
Boolean value which controls the counter overflow interruption. If this flag is set, each time counter overflow occurs, an SCTU interruption is generated.

- **Returned values**
  - A positive or null value on success:
    - SCTU service handle, to be used in further SCTU service function calls..
  - A negative error value otherwise:
    - ADL_RET_ERR_PARAM on a supplied parameter error.
    - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service was already subscribed for this block (the SCTU service can only be subscribed one time on each block).
    - ADL_RET_ERR_BAD_HDL if one or both supplied interruption handler identifiers are invalid.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

Important note:
While the SCTU service is subscribed, the Wireless CPU will never enter low power consumption mode, since the full speed internal clock is used. Low power consumption mode will be usable again as soon as the SCTU service is unsubscribed.

3.24.4 The adl_sctuSetChannelConfig Function

This function allows the application to configures one of the SCTU block comparator channels.

- **Prototype**

```
s32 adl_sctuSetChannelConfig (s32                  SctuHandle,
                              adl_sctuChannel_e    ChannelID,
                              u16                  CompareValue,
                              bool
      InterruptionFlag);
```

- **Parameters**

SctuHandle:
SCTU service handle, previously returned by the **adl_sctuSubscribe** function.

ChannelID:
Comparator channel identifier, using the following type:

```
typedef enum
{
    ADL_SCTU_CHANNEL_0,
    ADL_SCTU_CHANNEL_1,
    ADL_SCTU_CHANNEL_2,
    ADL_SCTU_CHANNEL_3
} adl_sctuChannel_e;
```

CompareValue:
Counter value to be monitored by the comparator channel.
The default value is set to 0 at service subscription time.

InterruptionFlag:
Boolean value which controls the comparator channel interruption. If this flag is set, an SCTU interruption is raised each time the counter value matches the comparator channel value.
The default value is set to **FALSE** at service subscription time.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on a supplied parameter error.
  - ADL_RET__ERR_UNKNOWN_HDL if the supplied SCTU handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the SCTU timer is running.

3.24.5 The adl_sctuStart Function

This function allows the application to start the SCTU block timer. Once started, the SCTU interruptions are generated, according to the counter overflow & comparator channel settings.

- **Prototype**

```
s32 adl_sctuStart ( s32 SctuHandle );
```

- **Parameters**

  SctuHandle:
  SCTU service handle, previously returned by the **adl_sctuSubscribe** function.

- **Returned values**
  - OK on success.
  - ADL_RET__ERR_UNKNOWN_HDL if the supplied SCTU handle is unknown.
  - ADL_RET_ERR_NOT_SUBSCRIBED if no interruption source has been defined in the SCTU block configuration.
  - ADL_RET_ERR_BAD_STATE if the SCTU timer is already running.

3.24.6 The adl_sctuRead Function

This function allows the application to retrieve the SCTU interruption source mask. It should be called only from low level interruption handlers subscribed with the ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT option; in other cases, the SCTU interruption source(s) will always be acknowledged, and the function will return an error.

- **Prototype**

```
s32 adl_sctuRead (  s32                SctuHandle,
                    adl_sctuInfo_t *   Info );
```

- **Parameters**

  SctuHandle:
  SCTU service handle, previously returned by the **adl_sctuSubscribe** function.

  Info:
  SCTU interruption source information structure. Please refer to **adl_sctuInfo_t** structure definition (see § 3.24.2).

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on a supplied parameter error.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied SCTU handle is unknown.
  - ADL_RET_ERR_BAD_STATE if no SCTU interruption has occurred.

3.24.7 The adl_sctuStop Function

This function allows the application to stop the SCTU block timer. SCTU interruptions will no longer be generated for this block.

- **Prototype**

```
s32 adl_sctuStop ( s32 SctuHandle );
```

- **Parameters**

  SctuHandle:
  SCTU service handle, previously returned by the **adl_sctuSubscribe** function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied SCTU handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the SCTU timer is not running.

3.24.8 The adl_sctuUnsubscribe Function

This function allows the application to unsubscribe from the SCTU service. Associated interruption handlers are disconnected from the SCTU interruption source.

- **Prototype**

```
s32 adl_sctuUnsubscribe ( s32 SctuHandle );
```

- **Parameters**

  SctuHandle:
  SCTU service handle, previously returned by the **adl_sctuSubscribe** function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied SCTU handle is unknown.
  - ADL_RET_ERR_BAD_STATE if the SCTU timer is running.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.24.9 Example

This example simply demonstrates how to use the SCTU service in a nominal case (error cases are not handled).

```
// Global variables

// SCTU service handle
s32 SctuHandle;

// IRQ service handle
s32 IrqHandle;

// SCTU configuration: counter overflow each 5ms
adl_sctuSettings_t Config =
{ 80, 0, 65166, TRUE };

// SCTU interruption handler
bool    MySctuHandler    (adl_irqID_e    Source,    adl_irqNotifyLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
  // Read the interruption source
  adl_sctuInfo_t Source, * AutoReadSource;
  adl_sctuRead ( SctuHandler, &Source );

  // Interruption source can also be obtained from the auto read
option.
  AutoReadSource = ( adl_sctuInfo_t * ) Data->SourceData;

  return TRUE;
}
```

```
// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

  // Subscribes to the IRQ service
  IrqHandle    =     adl_irqSubscribe  (     MySctuHandler,
 ADL_IRQ_NOTIFY_LOW_LEVEL, ADL_IRQ_PRIORITY_HIGH_LEVEL,  ADL
_IRQ_OPTION_AUTO_READ | ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT );

  // Subscribes to the SCTU service
    SctuHandle  =  adl_sctuSubscribe  (  ADL_SCTU_BLOCK1,  IrqHandle,  0,
                                        &Config );
    // Configures comparator channel
    adl_sctuSetChannelConfig  (  SctuHandle,  ADL_SCTU_CHANNEL_0,  65500,
                                  TRUE );

    // Starts timer
    adl_sctuStart ( SctuHandle );
}
void MyFunction2 ( void )
{
    // Stops the timer
    adl_sctuStop ( SctuHandle );

    // Un-subscribes from the SCTU service
    adl_sctuUnsubscribe ( SctuHandle );
}
```

3.25 Extint ADL Service

The ADL External Interruption (ExtInt) service allows the application to handle Wireless CPU External Interruption pin configuration & interruptions.
External interruption pins are multiplexed with the Wireless CPU GPIO, and are available only if configured correctly through the AT commands:

- The "INTERRUPT" feature must be enabled in the AT+WFM command.
- The required pin must be configured as an interruption (not a GPIO) in the AT+WIPC command.

The global External Interruption pin operation is described below:

- The interruption is generated either on the falling or the rising edge of the input signal, or both (with debounce filter only).
- The input signal is filtered by one of the following processes:
  - Bypass (no filter)
  - Debounce (a stable state is required for a configurable duration before generating the interruption)

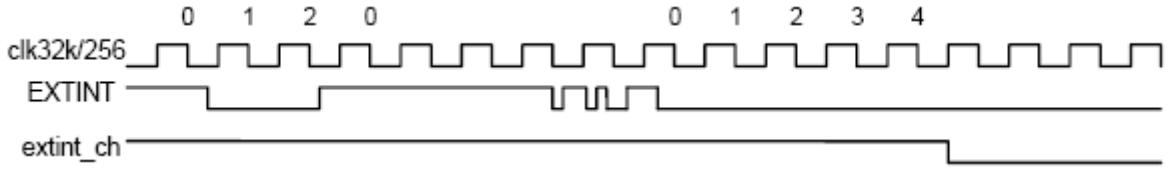


Figure 12: ADL External Interruption Service: Example of Interruption with Debounce Period

E.g. EXTINT is the input signal, extint_ch is the generated interruption. When the debounce period equals 4, the Wireless CPU waits for a stable signal during 4 cycles (7.8 ms steps) before generating the interruption.

- Stretching (the signal is stretched in order to detect even small glitches in the signal)

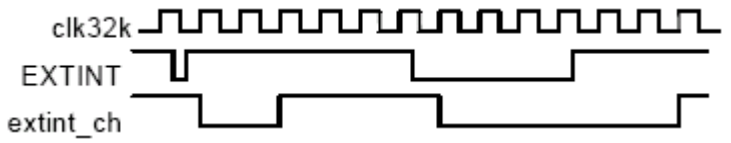


Figure 13:ADL External Interruption Service: Example of Interruption with Stretching Process

E.g. EXTINT is the input signal, extint_ch is the generated interruption. With the stretching process, the generated interruptions are stretched in time, in order not to miss any pulses on the input signal.

- Interruption generated because an External Interruption pin is always pre-acknowledged, whatever is the subscribed option in the IRQ service.

3.25.1 Required Header File

The header file for the ExtInt service definitions is:

`adl_extint.h`

3.25.2 The adl_extintSettings_t Structure

This structure allows the application to configure external interruption pin behavior.

```
typedef struct
{
  adl_extintPolarity_e  Polarity;        // Input signal polarity
  adl_extintProcess_e    Process;        // Input signal process
  u8                    DebounceDuration;// Debounce process
duration
} adl_extintSettings_t;
```

- **Parameters**

  Polarity:
  Interruption generation polarity, using the following type:

```
typedef enum
{
    ADL_EXTINT_POLARITY_LOW,  // Falling edge interruption
    ADL_EXTINT_POLARITY_HIGH, // Rising edge interruption
    ADL_EXTINT_POLARITY_BOTH, // Both edges interruption
                              // only usable with debounce process

    ADL_EXTINT_POLARITY_LAST  // Internal use only
} adl_extintPolarity_e;
```

  Process:
  Filter process applied to the input signal:

```
typedef enum
{
    ADL_EXTINT_PROCESS_BYPASS,      // No filter
    ADL_EXTINT_PROCESS_DEBOUNCE,    // Debounce filter
    ADL_EXTINT_PROCESS_STRETCHING,  // Stretching filter

    ADL_EXTINT_PROCESS_LAST   // Internal use only
} adl_extintProcess_e;
```

  DebounceDuration:
  Used only with the debounce process (ignored with other processes).
  Time during which the signal must be stable before generating the interruption, in 7.8 ms steps.
  The values allowed range is from 1 to 7

3.25.3 The adl_extintInfo_t Structure

This structure allows the application to get the external interruption pin input status at any time. When an interruption handler is subscribed with the ADL_IRQ_OPTION_AUTO_READ option and plugged on the ExtInt service, the **SourceData** field in the **adl_irqEventData_t** input parameter of this handler must be cast to this type in order to handle the information correctly.

```
typedef struct
{
    u8 PinState;  // External Interruption Pin input status
} adl_extintInfo_t;
```

- **Parameter**

  PinState:
  Current state (0/1) of the input signal plugged on the external interruption pin.

3.25.4 The adl_extintSubscribe Function

This function allows the application to subscribe to the ExtInt service. Each External Interruption pin can only be subscribed one time. Once subscribed, the pin is no more configurable through the AT commands interface (with AT+WIPC or AT+WFM commands).

- **Prototype**

```
s32 adl_extintSubscribe ( adl_extintID_e         ExtIntID
                          s32                    LowLevelIrqHandle
                          s32                    HighLevelIrqHandle
                          adl_extintSettings_t * Settings );
```

- **Parameters**

  ExtIntID:
  External Interruption pin identifier to be subscribed, using the type below.

```
typedef enum
{
    ADL_EXTINT_PIN0,    // External interruption pin 0
    ADL_EXTINT_PIN1,    // External interruption pin 1

    ADL_EXTINT_PIN_LAST // Internal use only
} adl_extintID_e;
```

  LowLevelIrqHandle:
  Low level interruption handler identifier, previously returned by the **adl_irqSubscribe** function.
  This parameter is optional if the **HighLevelIrqHandle** parameter is supplied.

HighLevelIrqHandle:
High level interruption handler identifier, previously returned by the **adl_irqSubscribe** function.
This parameter is optional if the **LowLevelIrqHandle** parameter is supplied.

Settings:
External interruption pin configuration, to be defined using the **adl_extintSettings_t** structure.

- **Returned values**
  - A positive or null value on success:
    - ExtInt service handle, to be used in further ExtInt service function calls.
  - A negative error value otherwise:
    - ADL_RET_ERR_PARAM on a supplied parameter error
    - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service was already subscribed for this external interruption pin (the ExtInt service can only be subscribed one time for each pin).
    - ADL_RET_ERR_BAD_HDL if one or both supplied interruption handler identifiers are invalid.
    - ADL_RET_ERR_BAD_STATE if the external interruption pin is not correctly configured with the AT Commands interface (please refer to the AT+WIPC command description).
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.25.5 The adl_extintConfig Function

This function allows the application to modify an external interruption pin configuration.

- **Prototype**

```
s32 adl_extintConfig (  s32                     ExtIntHandle,
                        adl_extintSettings_t *  Settings );
```

- **Parameters**

ExtIntHandle:
ExtInt service handle, previously returned by the **adl_extintSubscribe** function.

Settings:
External interruption pin configuration, to be defined using the **adl_extintSettings_t** structure.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on a supplied parameter error.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied ExtInt handle is unknown.

3.25.6 The adl_extintRead function

This function allows the application to retrieve the external interruption pin input status.

- **Prototype**

```
s32 adl_extintRead (s32                 ExtIntHandle,
                    adl_extintInfo_t *  Info );
```

- **Parameters**

ExtIntHandle:
ExtInt service handle, previously returned by the **adl_extintSubscribe** function.

Info:
External interruption pin information structure.
Refer to **adl_extintInfo_t** type definition (see §3.25.3).

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on a supplied parameter error.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied ExtInt handle is unknown.

3.25.7 The adl_extintUnsubscribe Function

This function allows the application to unsubscribe from the ExtInt service. Associated interruption handlers are unplugged from the ExtInt interruption source. Pin configuration control is resumed by the AT+WIPC command.

- **Prototype**

```
s32 adl_extintUnsubscribe ( s32 ExtIntHandle );
```

- **Parameters**

ExtIntHandle:
ExtInt service handle, previously returned by the **adl_extintSubscribe** function.

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied ExtInt handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.25.8 Example

This example demonstrates how to use the ExtInt service in a nominal case (error cases are not handled).

```
// Global variables

// ExtInt service handle
s32 ExtIntHandle;

// IRQ service handle
s32 IrqHandle;

// ExtInt configuration: debounce mode on both edges, with 7.8 ms
debounce period
adl_extintSettings_t Config =
{ ADL_EXTINT_POLARITY_BOTH, ADL_EXTINT_PROCESS_DEBOUNCE, 1 };


// ExtInt interruption handler
bool MyExtIntHandler (adl_irqID_e Source, adl_irqNotifyLevel_e
                      NotificationLevel,
                      adl_irqEventData_t * Data )
{
    // Read the input status
    adl_extintInfo_t Status, * AutoReadStatus;
    adl_extintRead ( ExtIntHandle, &Status );

    // Input status can also be obtained from the auto read option.
    AutoReadStatus = ( adl_extintInfo_t * ) Data->SourceData;

    return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the IRQ service
IrqHandle = adl_irqSubscribe ( MyExtIntHandler, ADL IRQ NOTIFY_LOW_LEVE
L,                ADL_IRQ_PRIORITY_HIGH_LEVEL, ADL_IRQ_OPTION_AUTO_READ
);

    // Subscribes to the ExtInt service
    SctuHandle  =  adl_sctuSubscribe  (  ADL_SCTU_BLOCK1,  IrqHandle,  0,
                                                              &Config
);
    // Configures comparator channel
    ExtIntHandle = adl_ExtIntSubscribe ( ADL_EXTINT_PIN0, IrqHandle, 0,
 &Config );
}
void MyFunction2 ( void )
{
    // Un-subscribes from the ExtInt service
    adl_extintUnsubscribe ( ExtIntHandle );
}
```

3.26 Execution Context Service

The application runs under several execution contexts, according to the monitored event (ADL service event, or interruption event).
The execution contexts are:

- **The application task context;**
  This is the main application context, initialized on the **`adl_main`** function call, and scheduled each time a message is received; each message is then converted to an ADL service event, according to its content.
  This context has a low priority and should be interrupted by the other ones.
- **The high level interruption handler context;**
  This is also a task context, but with an higher priority that the main application task. High level interruption handlers run in this context.
  This context has a middle priority: when an interruption raises an event monitored by a high level handler.
  This context will be immediately activated, even if the application task was running; however, this context could be interrupted by low level interruption handlers.
- **The low level interruption handler context;**
  This is a context designed to be activated as soon as possible on an interruption event.
  This context has a high priority: when an interruption raises an event monitored by a low level handler.
  This context will be immediately activated, even if a task (whatever it is: application task, high level handler or a WAVECOM OS task) was running.
  On the other hand, the execution time spent in this context has to be as short as possible; moreover, some service calls are forbidden while this context is running.

As the application code should run in different contexts at the same time, the user should protect his critical functions against re-entrancy.
Critical code sections should be protected through a semaphore mechanism (cf. Semaphores service). The ADL services are all re-entrant.

Data can be exchanged between contexts through a message system (cf. Messages service).
However, the RAM area is global and accessible from all contexts.

3.26.1 Required Header File

The header file for the Execution Context function is:

```
adl_ctx.h
```

3.26.2 The adl_ctxID_e Type

This type defines the execution context identifiers.

```
typedef enum
{
    ADL_CTX_OAT_TASK,

    ADL_CTX_LOW_LEVEL_IRQ_HANDLER = 0xFD,
    ADL_CTX_HIGH_LEVEL_IRQ_HANDLER = 0xFE,
    ADL_CTX_ALL = 0xFF,    // Reserved for internal use
    ADL_CTX_WAVECOM = 0xFF,// Wavecom tasks context
} adl_ctxID_e;
```

The ADL_CTX_OAT_TASK constant identifies the Open AT application task context.
The ADL_CTX_LOW_LEVEL_IRQ_HANDLER constant identifies the low level interruption handler context.
The ADL_CTX_HIGH_LEVEL_IRQ_HANDLER constant identifies the high level interruption handler context.
The ADL_CTX_WAVECOM constant identifies the Wavecom OS context.

3.26.3 The adl_ctxGetID Function

This function allows the application to retrieve the current execution context identifier.

- **Prototype**

```
adl_ctxID_e adl_ctxGetID ( void );
```

- **Returned values**
  - ADL_CTX_OAT_TASK if the function is called from an ADL service event handler.
  - ADL_CTX_LOW_LEVEL_IRQ_HANDLER if the function is called from a low level interruption handler.
  - ADL_CTX_HIGH_LEVEL_IRQ_HANDLER if the function is called from an high level interruption handler.

3.26.4 The adl_ctxGetTaskID Function

This function allows the application to retrieve the current running task identifier:

- In Open AT® task or high level interruption handler contexts, this function will behave like the **`adl_ctxGetID`** function.
- But in a low level handler execution context, the retrieved identifier will be the active task identifier when the interruption signal is raised.

- **Prototype**

```
adl_ctxID_e adl_ctxGetTaskID ( void );
```

- **Returned values**
  - ADL_CTX_OAT_TASK if the function is called from an ADL service event handler.
  - ADL_CTX_HIGH_LEVEL_IRQ_HANDLER if the function is called from a high level interruption handler.
  - If called from a low level interruption handler, the returned value depends on the interrupted task:
    - ADL_CTX_OAT_TASK if the Open AT® task was running.
    - ADL_CTX_WAVECOM if a Wavecom OS task was running.
    - ADL_CTX_HIGH_LEVEL_IRQ_HANDLER if a high level interruption handler was running.

3.26.5 The adl_ctxGetDiagnostic Function

This function allows the application to retrieve information about the current application's execution contexts.

- **Prototype**

```
u32 adl_ctxGetDiagnostic ( void );
```

- **Returned values**

  Bit mask where one or several of the following values are set (with a bitwise OR combination):
  - ADL_CTX_DIAG_NO_IRQ_PROCESSING if the Open AT® IRQ processing mechanism has not been started (interruption handlers stack sizes have not been supplied).
  - ADL_CTX_DIAG_BAD_IRQ_PARAM (*reserved for future use).*
  - ADL_CTX_DIAG_NO_HIGH_LEVEL_IRQ_HANDLER if high level interruption handlers are not supported (high level handler stack size is not supplied).

3.26.6 The adl_ctxSuspend Function

This function allows the application to suspend the Open AT® task process. This process can be resumed later thanks to the **`adl_ctxResume`** function, which should be called from an interruption handler.

- **Prototype**

```
s32 adl_ctxSuspend ( adl_ctxID_e Task );
```

- **Parameters**

  Task:
  Task identifier to be suspended.
  The only value allowed is ADL_CTX_OAT_TASK.

- **Returned values**

  - OK on success:
    The application task is now suspended.
  - ADL_RET_ERR_PARAM if a wrong parameter is supplied.

Notes:

- If the function was called in the application task context, it will not return but just suspend the task.
  The OK value will be returned when the task process is resumed.
- While the application is suspended, received events are queued until the process is resumed.
  If too many events occur, the application mailbox would be overloaded, and this would lead the Wireless CPU to reset (**the application task should not be suspended for a long time**).
- For the same reason, while the application is suspended, the subscribed AT commands cannot be processed by the application.

3.26.7 The adl_ctxResume Function

This function allows the application to resume the Open AT® task process, previously suspended with to the **`adl_ctxSuspend`** function.

- **Prototype**

```
s32 adl_ctxResume ( adl_ctxID_e Task );
```

- **Parameters**

  Task:
  Task identifier to be suspended.
  The only value allowed is ADL_CTX_OAT_TASK.

- **Returned values**

  - OK on success
    The application's task process is now resumed.
  - ADL_RET_ERR_PARAM if a wrong parameter is supplied.

3.26.8 Example

This example simply demonstrates how to use the execution context service in a nominal case (error cases are not handled).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Get the execution context state
    u32 Diagnose = adl_ctxGetDiagnose();

    // Get the execution context
    adl_ctxID_e CurCtx = adl_ctxGetID();

    // Check for low level handler context
    if ( CurCtx == ADL_CTX_LOW_LEVEL_IRQ_HANDLER )
    {

        // Get the interrupted context
        adl_ctxID_e InterruptedCtx = adl_ctxGetTaskID();
    }
}



// Somewhere in the application code, used within an interruption
handler
void MyIRQFunction ( void )
{
    // Suspend the application task
    adl_ctxSuspend ( ADL_CTX_OAT_TASK );

    // Resume the application task
    adl_ctxResume ( ADL_CTX_OAT_TASK );
}
```

3.27 ADL VariSpeed Service

The ADL VariSpeed service allows the Wireless CPU clock frequency to be controlled, in order to temporarily increase application performance.

Note:

- The Real Time Enhancement feature must be enabled on the Wireless CPU in order to make this service available.
- The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value:
  This feature state is represented by the bit 4 (00000010 in hexadecimal format).
- Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU.

3.27.1 Required Header File

The header file for the VariSpeed service is:

`adl_vs.h`

3.27.2 The adl_vsMode_e Type

This type defines the available CPU modes for the VariSpeed Service.

```
typedef enum
{
  ADL_VS_MODE_STANDARD,
  ADL_VS_MODE_BOOST,

  ADL_VS_MODE_LAST              // Reserved for internal use
} adl_vsMode_e;
```

The ADL_VS_MODE_STANDARD constant identifies the standard CPU clock mode (default CPU mode on startup).

The ADL_VS_MODE_BOOST constant can be used by the application to make the Wireless CPU enter a specific boost mode, where the CPU clock frequency is set to its maximum value.

Caution:
In boost mode, the Wireless CPU power consumption increases significantly. For more information, refer to the Wireless CPU Power Consumption Mode documentation.

Depending on the Wireless CPU type, the clock frequencies of the available modes are listed below:

Modes	Wireless CPU	
	Q2686	Q2687
ADL_VS_MODE_STANDARD	26 MHz	26 MHz
ADL_VS_MODE_BOOST	104 MHz	104 MHz

3.27.3 The adl_vsSubscribe Function

This function allows the application to get control over the VariSpeed service. The VariSpeed service can only be subscribed one time.

- **Prototype**

```
s32 adl_vsSubscribe ( void );
```

- **Parameters**

  None

- **Returned values**

  - A positive or null value on success:
    - VariSpeed service handle, to be used in further service function calls.
  - A negative error value otherwise:
    - ADL_RET_ERR_ALREADY_SUBSCRIBED if the service has already been subscribed.
    - ADL_RET_ERR_NOT_SUPPORTED if the Real Time enhancement feature is not enabled on the Wireless CPU.
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.27.4 The adl_vsSetClockMode Function

This function allows the application to modify the speed of the CPU clock.

- **Prototype**

```
s32 adl_vsSetClockMode (        s32 VsHandle,
                                adl_vsMode_e ClockMode );
```

- **Parameters**

  VsHandle:
  VariSpeed service handle, previously returned by the **adl_vsSubscribe** function.

  ClockMode:
  Required clock mode.
  Refer to **adl_vsMode_e** type definition for more information (see § 3.27.2).

- **Returned values**

  - OK on success
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied handle is unknown.
  - ADL_RET_ERR_PARAM if the supplied clock mode value is wrong.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.27.5 The adl_vsUnsubscribe function

This function allows the application to unsubscribe from the VariSpeed service control. The CPU mode is reset to the standard speed.

- **Prototype**

```
s32 adl_vsUnsubscribe ( s32 VsHandle );
```

- **Parameters**

  VsHandle:
  VariSpeed service handle, previously returned by the **adl_vsSubscribe** function.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_UNKNOWN_HDL if the supplied handle is unknown.
  - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level interruption handler (the function is forbidden in this context).

3.27.6 Example

This example demonstrates how to use the VariSpeed service in a nominal case (error cases are not handled).

```
// Global variable: VariSpeed service handle
s32 MyVariSpeedHandle;



// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribe to the VariSpeed service
    MyVariSpeedHandle = adl_vsSubscribe();

    // Enter the boost mode
    adl_vsSetClockMode ( MyVariSpeedHandle, ADL_VS_MODE_BOOST );
}
void MyFunction2 ( void )
{
    // Un-subscribe from the VariSpeed service
    adl_vsUnsubscribe ( MyVariSpeedHandle );
}
```

3.28 ADL DAC Service

3.28.1 Required Header File

The header file for the functions dealing with the DAC interface is:

`adl_dac.h`

3.28.2 The adl_dacSubscribe Function

This function subscribes to one of the module DAC block interface.

- **Prototype**

```
s32 adl_dacSubscribe (   adl_dacChannel_e Channel,
                         adl_dacParam_t * Parameters )
```

- **Parameters**

Channel:
The DAC channel identifier to be subscribed, using the type below:

```
typedef enum
{
   ADL_DAC_CHANNEL_1,
   ADL_DAC_NUMBER_OF_CHANNEL,
   ADL_DAC_CHANNEL_PAD = 0x7fffffff
} adl_dacChannel_e;
```

Channel identifiers depend on the current module type (please refer to the module Product Technical Specification document for more information):

Module type	Channel identifier	Output DAC PIN name	Output DAC PIN number
Q2687	**ADL_DAC_CHANNEL_1**	AUXDAC	82

Parameters:
DAC channel initialization parameters, using the type below:

```
typedef struct {
u32   InitialValue;
} adl_dacParam_t;
```

InitialValue:
Initial value to be written on the DAC just after it has been opened. Significant bits and output voltage depend on the module type (please refer to the module Product Technical Specification document for more information).

Module type	Significant bits	Max. output voltage
Q2687	8 less significant bits	2.2 V (for 0xFF value)

- **Returned values**
  - A positive or null value on success:
    - DAC service handle, to be used with further DAC service functions calls..
  - A negative error value otherwise:
    - ADL_RET_ERR_ALREADY_SUBSCRIBED if the required channel has already been subscribed.
    - ADL_RET_ERR_NO_MORE_HANDLES if there is no more free DAC handles.
    - ADL_RET_ERR_NOT_SUPPORTED if the current module does not support the DAC service.
    - ADL_RET_ERR_PARAM on parameter error.

Note:
The DAC service is only available on the Q2687 product.

3.28.3 The adl_dacUnsubscribe Function

This function unsubscribes from a previously subscribed DAC block.

- **Prototype**

```
s32 adl_dacUnsubscribe ( s32 Handle )
```

- **Parameters**

  Handle:
  DAC service handle previously returned by the **adl_dacSubscribe** function.

- **Returned values**
  - OK on success
  - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown.

3.28.4 The adl_dacWrite Function

This function allows to set the output value of a subscribed DAC block.

- **Prototype**

```
s32 adl_dacWrite ( s32 Handle,
                   u32 Value )
```

- **Parameters**

  Handle:
  DAC service handle previously returned by the **adl_dacSubscribe** function.

  Value:
  Value to be written on the DAC output. Significant bits and output voltage depend on the module type (Refer to the module Product Technical Specification document for more information).

Module type	Significant bits	Max. output voltage
Q2687	8 less significant bits	2.2 V (for 0xFF value)

- **Returned values**
  - OK on success.
  - ADL_RET_ERR_PARAM on parameter error.

3.28.5 Example

This example shows how to use the DAC service in a nominal case (error cases not handled).

```
// Global variable
s32 MyDACHandle;



// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Initialization structure
    adl_dacParam_t InitStruct = { 0 };

    // Subscribe to the DAC service
    MyDACHandle = adl_dacSubscribe ( ADL_DAC_CHANNEL_1, &InitStruct );

    // Write a value on the DAC block
    adl_dacWrite ( MyDACHandle, 80 );

    ...

    // Write another value on the DAC block
    adl_dacWrite ( MyDACHandle, 190 );

    ...

    // Unsubscribe from the DAC service
    adl_dacUnsubscribe ( MyDACHandle );
}
```

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_DONE	-11	The required iterative process is now terminated
ADL_RET_ERR_OVERFLOW	-12	The required operation has exceeded the function capabilities
ADL_RET_ERR_NOT_SUPPORTED	-13	An option, required by the function, is not enabled on the Wireless CPU, the function is not supported in this configuration
ADL_RET_ERR_NO_MORE_TIMERS	-14	The function requires a timer subscription, but no more timers are available
ADL_RET_ERR_NO_MORE_SEMAPHORES	-15	The function requires a semaphore allocation, but there are no more free resource
ADL_RET_ERR_SERVICE_LOCKED	-16	If the function was called from a low lewel interruption handler (the function is forbidden in this case)
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

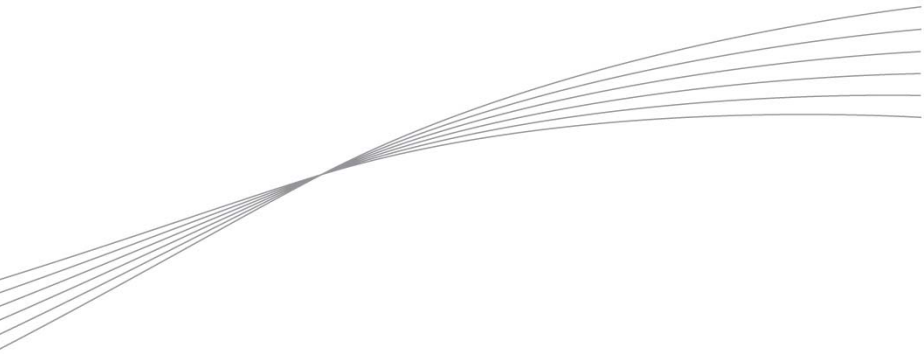
Error Code	Error Value
ADL_FLH_RET_ERR_OBJ_NOT_EXIST	ADL_RET_ERR_SPECIFIC_BASE
ADL_FLH_RET_ERR_MEM_FULL	ADL_RET_ERR_SPECIFIC_BASE-1
ADL_FLH_RET_ERR_NO_ENOUGH_IDS	ADL_RET_ERR_SPECIFIC_BASE-2
ADL_FLH_RET_ERR_ID_OUT_OF_RANGE	ADL_RET_ERR_SPECIFIC_BASE-3

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 A&D Storage Service Error Codes

Error Code	Error Value
ADL_AD_RET_ERR_NOT_AVAILABLE	ADL_RET_ERR_SPECIFIC_BASE
ADL_AD_RET_ERR_OVERFLOW	ADL_RET_ERR_SPECIFIC_BASE - 1
ADL_AD_RET_ERROR	ADL_RET_ERR_SPECIFIC_BASE - 2
ADL_AD_RET_ERR_NEED_RECOMPACT	ADL_RET_ERR_SPECIFIC_BASE - 3



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. - Unit 201-207, 2nd Floor, Bio-Informatics Centre – No.2 Science Park West Avenue - Hong Kong Science Park, Shatin - New Territories, Hong Kong