ADL User Guide for Open AT® OS 6.35

User Guide

WM_DEV_OAT_UGD_080
014
February 16, 2011

Important Notice

Due to the nature of wireless communications, transmission and reception of data can never be guaranteed. Data may be delayed, corrupted (i.e., have errors) or be totally lost. Although significant delays or losses of data are rare when wireless devices such as the Sierra Wireless modem are used in a normal manner with a well-constructed network, the Sierra Wireless modem should not be used in situations where failure to transmit or receive data could result in damage of any kind to the user or any other party, including but not limited to personal injury, death, or loss of property. Sierra Wireless accepts no responsibility for damages of any kind resulting from delays or errors in data transmitted or received using the Sierra Wireless modem, or for failure of the Sierra Wireless modem to transmit or receive such data.

Safety and Hazards

Do not operate the Sierra Wireless modem in areas where blasting is in progress, where explosive atmospheres may be present, near medical equipment, near life support equipment, or any equipment which may be susceptible to any form of radio interference. In such areas, the Sierra Wireless modem **MUST BE POWERED OFF**. The Sierra Wireless modem can transmit signals that could interfere with this equipment. Do not operate the Sierra Wireless modem in any aircraft, whether the aircraft is on the ground or in flight. In aircraft, the Sierra Wireless modem **MUST BE POWERED OFF**. When operating, the Sierra Wireless modem can transmit signals that could interfere with various onboard systems.

Note: *Some airlines may permit the use of cellular phones while the aircraft is on the ground and the door is open. Sierra Wireless modems may be used at this time.*

The driver or operator of any vehicle should not operate the Sierra Wireless modem while in control of a vehicle. Doing so will detract from the driver or operator's control and operation of that vehicle. In some states and provinces, operating such communications devices while in control of a vehicle is an offence.

Limitations of Liability

This manual is provided "as is". Sierra Wireless makes no warranties of any kind, either expressed or implied, including any implied warranties of merchantability, fitness for a particular purpose, or noninfringement. The recipient of the manual shall endorse all risks arising from its use.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Sierra Wireless. SIERRA WIRELESS AND ITS AFFILIATES SPECIFICALLY DISCLAIM LIABILITY FOR ANY AND ALL DIRECT, INDIRECT, SPECIAL, GENERAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES INCLUDING, BUT NOT LIMITED TO, LOSS OF PROFITS OR REVENUE OR ANTICIPATED PROFITS OR REVENUE ARISING OUT OF THE USE OR INABILITY TO USE ANY SIERRA WIRELESS PRODUCT, EVEN IF SIERRA WIRELESS AND/OR ITS AFFILIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR THEY ARE FORESEEABLE OR FOR CLAIMS BY ANY THIRD PARTY.

Notwithstanding the foregoing, in no event shall Sierra Wireless and/or its affiliates aggregate liability arising under or in connection with the Sierra Wireless product, regardless of the number of events, occurrences, or claims giving rise to liability, be in excess of the price paid by the purchaser for the Sierra Wireless product.

Patents

This product may contain technology developed by or for Sierra Wireless Inc.

This product includes technology licensed from QUALCOMM® 3G.

This product is manufactured or sold by Sierra Wireless Inc. or its affiliates under one or more patents licensed from InterDigital Group.

Copyright

Trademarks

AirCard® and Watcher® are registered trademarks of Sierra Wireless. Sierra Wireless™, AirPrime™, AirLink™, AirVantage™ and the Sierra Wireless logo are trademarks of Sierra Wireless.

, , ®, inSIM®, “YOU MAKE IT, WE MAKE IT WIRELESS®”, WAVECOM®, WISMO®, Wireless Microprocessor®, Wireless CPU®, Open AT® are filed or registered trademarks of Sierra Wireless S.A. in France and/or in other countries.

Windows® and Windows Vista® are registered trademarks of Microsoft Corporation.

Macintosh and Mac OS are registered trademarks of Apple Inc., registered in the U.S. and other countries.

QUALCOMM® is a registered trademark of QUALCOMM Incorporated. Used under license.

Other trademarks are the property of the respective owners.

Contact Information

Sales Desk:	Phone:	1-604-232-1488
	Hours:	8:00 AM to 5:00 PM Pacific Time
	E-mail:	sales@sierrawireless.com
Post:	Sierra Wireless 13811 Wireless Way Richmond, BC Canada V6V 3A4	
Fax:	1-604-231-1109	
Web:	www.sierrawireless.com	

Consult our website for up-to-date product descriptions, documentation, application notes, firmware upgrades, troubleshooting tips, and press releases: www.sierrawireless.com

Document History

Index	Date	Versions	
001	June 19, 2008	Creation for Open AT® OS v6.10	
002	August 20, 2008	Updates for Open AT® OS v6.10	
003	September 16, 2008	Updates for Open AT® OS v6.10	
004	October 14, 2008	Creation for Open AT® OS v6.11	
005	November 26, 2008	Creation for Open AT® OS v6.20	
006	December 5, 2008	Updates for Open AT® OS v6.20	
007	February 26, 2009	Updates for Open AT® OS v6.21	
009	April 24, 2009	Updates for Open AT® OS v6.30	
010	July 10, 2009	Updates for Open AT® OS v6.31	
011	September 30, 2009	Updates for Open AT® OS v6.31	
012	June 15, 2010	Updates for Open AT® OS 6.32 .	
013	July 15, 2011	Updates for Open AT® OS 6.33	
014	February 16, 2011	Updates for Open AT® OS 6.35	

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

Note: *Though all features are documented in this manual, new features may still be in beta stage at publication and therefore may not yet be validated. Please refer to the Customer Release Note for complete and detailed information regarding beta and validated features at time of release.*

Contents

List of Figures

1. Introduction

1.1. Important Remark

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. AT Commands Interface Guide for FW 7.45 (Ref. WM_DEV_OAT_UGD_079)
2. Developer Studio (http://www.sierrawireless.com/developer_studio) online help

1.3. Glossary

Term	Definition
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 Sierra Wireless, 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 Sierra Wireless product.
Embedded OS	Software that includes the Embedded Application and the Sierra Wireless library.
Embedded software	User application binary: set of Embedded Application sources + Sierra Wireless library.
External Application	Application external to the Sierra Wireless product that sends AT commands through the serial link.
Developer Studio	Integrated development environment for developing embedded cellular Mobile to Mobile (M2M) applications
Target	Open AT® compatible product supporting an Embedded Application.
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.
Sierra Wireless library	Library delivered by Sierra Wireless to interface Embedded Application sources with Sierra Wireless Firmware functions.
Sierra Wireless Firmware	Set of GSM and open functions supplied to the User.

1.4. Abbreviations

A&D	Application & Data
ADL	Application Development Layer
AMS	AirPrime Management Services
API	Application Programming Interface
APN	Access Point Name
CID	Context IDentifier
CLSP	Core Layer Service Provider
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
MCC	Mobile Country Code
MNC	Mobile Network Codes
MS	Mobile Station
OS	Operating System
PDP	Packet Data Protocol
PDU	Protocol Data Unit
PLMN	Public Land Mobile Network
RAM	Random-Access Memory
ROM	Read-Only Memory
RTK	Real-Time Kernel
RSSI	Received Signal Strength Indication
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 Sierra Wireless embedded module's capabilities & interfaces.

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

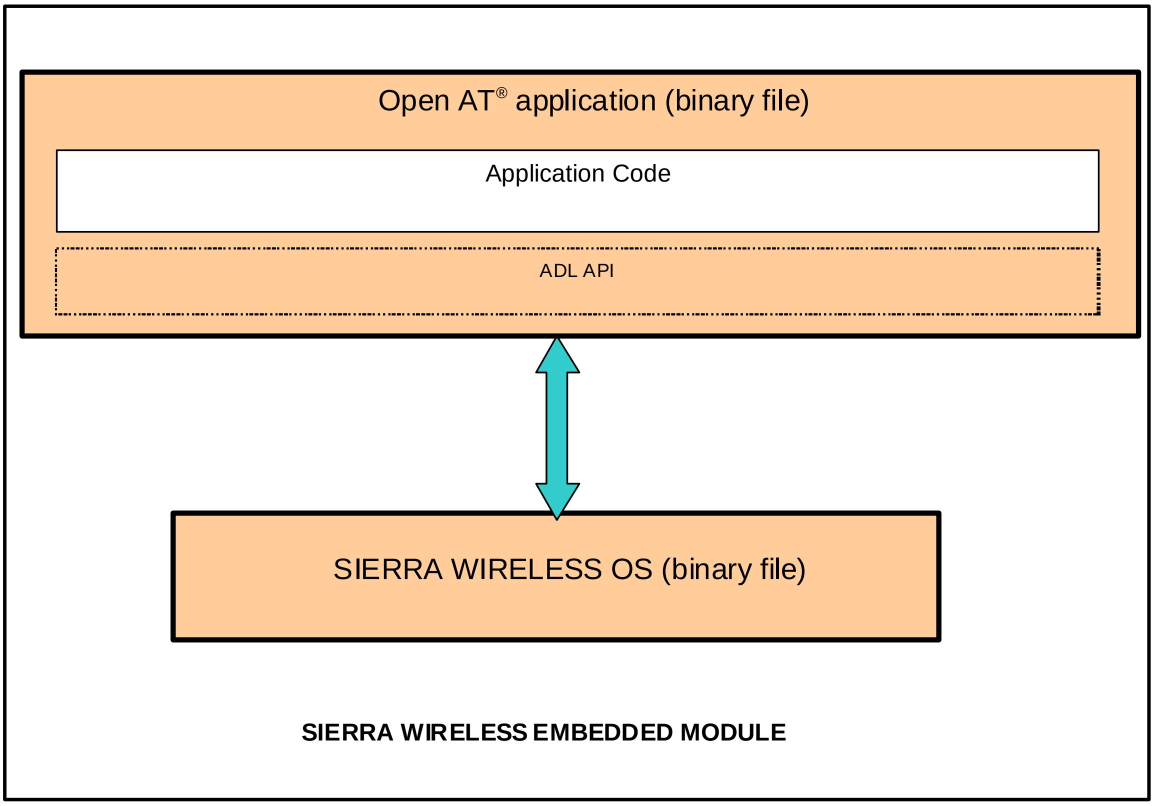


Figure 1. General software architecture

The different software elements on a Sierra Wireless 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 Plug-In libraries (such as the IP connectivity library),
- the Sierra Wireless Application Development Layer library, which provides all the services used by the application,
- the **Sierra Wireless Firmware**, which manages the Sierra Wireless embedded module.

2.2. ADL Limitations

2.2.1. AT+WIND command state

ADL is internally using +WIND indications in order to be kept informed of events raised by the embedded module. It has its own +WIND configuration, and this introduces the following behaviour when the application is started/stopped with the AT+WOPEN=0/1 command:

- The AT+WIND configuration is stored in two different places in AT+WOPEN=0 or AT+WOPEN=1 modes; consequently the enabled +WIND indications are not the same in these two modes.
- Moreover, when switching from AT+WOPEN=1 back to AT+WOPEN=0, all the +WIND indications will be enabled (whatever was the AT+WIND? configuration before switching to AT+WOPEN=1 mode).

2.2.2. Multitasking limitations

When an application declares several tasks, events which come following to a service subscription or in response to a service function will always notify the associated handlers in the first (more prioritary) task context (**except for Timers and Messages services**).

Examples:

- Even if the **`adl_atCmdSend`** function is called by the application in the task 2 context, the provided response handler will be called by ADL in the task 0 context.
- Even if the **`adl_smsSubscribe`** function is called by the application is the task 1 context, incoming SMS events will be notified by ADL in the task 0 context.
- But event handlers provided to Timers & Messages services will always be notified in the task contexts where the subscription functions were called.

2.3. Open AT® Memory Resources

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

Reminder:

- KB stands for Kilobytes
- MB stands for Megabytes
- Mb stands for Megabits

2.3.1. RAM Resources

The maximum RAM size available for Open AT® applications depends on the embedded module RAM capabilities, and on the used memory option at project creation time (please refer to Developer Studio online help for more information):

Total RAM SizeLink Option	8Mb of Total RAM	16Mb of Total RAM or more
"256KB" link option	256KB	256KB
"1MB+" link option	NC*	1MB or more

*"NC" stands for "Not Compatible", i.e. such a linked application will not start if downloaded on such a embedded module.

The available RAM for an Open AT application is always 1MB less than the actual value of the actual RAM and hence the same is displayed as 1 MB less when adl_memGETinfo() API is used.

For example, if the customer has a total of 2 Mb of RAM, then the RAM available for his Open AT(R) application is 1 Mb, and this value will be displayed with adl_memGETinfo() API.

2.3.2. Flash Resources

Total Flash Size	ROM(Application code)	Application & Data Storage Volume	Flash Objects Data
32Mb	256-1600KB (default: 832KB)	0-1344KB (default: 768KB)	128KB
64Mb or more	256-(1600+X)KB (default: (832+X)KB)	0-(1344+X)KB (default: 768KB)	384KB

For all flash sizes greater than 32Mb, all additional space is available for A&D and Application Code areas. X stands for this additional flash space in KB. X is reckoned using the following formula:

X = ((S – 32)/8) * 1024

Where S is the total Flash size in Mb; E.g. for a 64Mb Flash: X = 4096KB.

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

The maximum A&D storage place size is 1344+X KB (usable for Firmware upgrade capability). In this case the Open AT® application maximum size will be 256 KB.

The minimum A&D storage place size is 0 KB (usable for applications with huge hard coded data).

For more information about the A&D and Application Code areas size configuration, please refer to the AT+WOPEN command description in the AT Commands Interface Guide.

For both 32Mb and 64Mb flash types, the maximum FLASH object size that can be set with DWLWin is 1728KB.

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).*

2.4. Defined Compilation Flags

Developer Studio defines some compilation flags, related to the chosen generation environment. Please refer to Developer Studio online help for more information.

2.5. 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 Command	Fixed value
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).

Caution: *User is strongly advised against modifying the current values of these commands from any Open AT® application. Sierra Wireless 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 are 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 Sierra Wireless AT OS without a running ADL application).

2.6. Open AT® Specific AT Commands

Please refer to the AT Commands Interface Guide.

2.6.1. 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 Sierra Wireless Firmware 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 AT Commands Interface Guide.*

2.6.2. 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 interrupt 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.

Note: *Refer to the document AT Commands Interface Guide for more information about this command.*

Note: *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.*

2.7. Notes on Sierra Wireless Firmware

The Open AT® application runs within several tasks managed by the Sierra Wireless Firmware: event handlers are almost always called sequentially by ADL in the first task context, except for the Timers & Messages service (please refer to these services description for more information). The whole ADL API is reentrant and can be called from anymore in the application. If the application offers an API which is supposed to be called from several execution contexts, it is recommended to implement a reentrancy protection mechanism, using the semaphore service

The Sierra Wireless Firmware 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.

2.8. RTE limitations

2.8.1. 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) will be displayed, and the RTE application will stop with an error.

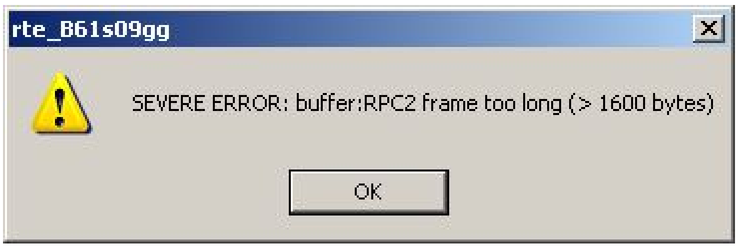


Figure 2. Error when trying to send too large a data buffer through an API

2.8.2. Services Limitations

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 ExtInt services, etc..) are not available in this mode. Moreover, the OpenDevice and the Event services are not available in this mode. The subscription function will always fail when called in RTE.

2.9. Recovery Mechanism

This mechanism has been introduced in the Sierra Wireless firmware with the IDS service. It allows to avoid infinite and uncontrolled reset loop in the firmware. A reset loop can occur when:

- a new unstable Sierra Wireless firmware is downloaded;
- a new unstable application is downloaded;
- the new application downloaded is not compatible with the Sierra Wireless firmware.

When a reset loop is detected by the Sierra Wireless firmware, a recovery mechanism is launched with the following 3 steps:

- Firstly, it tries to go back to the old firmware or application;
- If the first step does not work, it stops the Open AT® application (if started);
- Lastly, it starts the Xmodem downloader in interactive mode in order to download a new firmware.

When a reset loop occurs, Open AT® application is stopped after 8 resets.

3. API

3.1. Application Entry Points Interface

ADL supplies Application Entry Points Interface to allow applications to define the generic settings of the application tasks and contexts.

The application will have to define its entry points settings using the **`adl_InitTasks`** table. Each line of this table represents a task, which is characterized by the following parameters:

- the task entry point, called at the embedded module boot time, in the priority order
- the task call stack size
- the task priority level
- the task name

If the application wishes to use the IRQ service, it will also have to define the call stack sizes for its low level (**`adl_InitIRQLowLevelStackSize`**) and high level (**`adl_InitIRQHighLevelStackSize`**) interrupt handlers.

Moreover, some operations related to the initialization are available:

- An **Init type check** function (**`adl_InitGetType`**) to retrieve at any time the embedded module initialization type.

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. Tasks Declaration

3.1.2.1. Task Definition Table

Mandatory tasks definition table to be provided by the application. For more information on each task's parameters, please refer to the adl_InitTasks_t description. Each line of this table allows to intialize one task. To let the system know how many tasks are required, all the elements of the last line of this table have to be set to 0.

Task entry points declared in the table will be called on embedded module boot, in the priority order (the highest priority level is called first).

```
Const adl_InitTasks_t  adl_InitTasks[]
```

Note: *At least one task shall be declared in this table. If no tasks are declared in the table, the Firmware will refuse to launch the application, and the application launch status will be set to 16 (No task declared) Please refer to* ***AT+WOPEN=7*** *description in AT Commands Interface Guide for more information.*

Note: *There is maximum limit to the number of tasks which shall be declared in this table (Please refer to the Resources chapter for more information). If more tasks than the authorized maximum are declared in the table, the Firmware will refuse to launch the application, and the application launch status will be set to 5 (Too many tasks)*
Please refer to ***AT+WOPEN=7*** *description in AT Commands Interface Guide for more information.*

Warning: *Since ADL processing is running in the first application's task context, this one has always to be declared with the highest priority level, otherwise the Firmware will refuse to launch the application, and the application launch status will be set to 11 (Application binary init failure). Please refer to AT+WOPEN=7 description in AT Commands Interface Guide for more information.*

3.1.2.2. The adl_InitTasks_t Structure

Open AT® application's tasks declaration structure, used to format the **adl_InitTasks** table.

Code:

```
typedef struct
{
        void            (* EntryPoint)(void);
        u32             StackSize;
        const ascii*    Name;
        u8              Priority;
} adl_InitTasks_t;
```

Description

EntryPoint(void)

Task initialization handler, which aims to be called each time the Embedded module boots, as soon as the application is started with the **AT+WOPEN=1** command.

Notes

Note: *A task entry point function is NOT like a standard "C" main function. The task does not end when returns. An Open AT® application is stopped only if the **AT+WOPEN=0** command is used. Such a call-back 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 embedded module will reset due to the watchdog hardware security (please refer to Hardware Security: Watchdog Protection for more information).*

StackSize

Used to provide to the system the required call stack size (in bytes) for the current task. A call stack is the Open AT® RAM area which contains the local variables and return addresses for function calls. Call stack sizes are deduced from the total available RAM size for the Open AT® application.

Note: *In RTE mode, the call stacks are processed by the host's operating system, and are not configurable (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.*
The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than ARM compilers. If the GCC compiler is used, the Open AT® application has to be declared with greater stack sizes.
Call stack sizes shall be declared with some extra bytes margin. It is not recommended to try to reckon exactly the required call stack size of each task.
If the total call stack sizes (including the tasks ones & the interrupt contexts ones) is too large, the Firmware will refuse to launch the application, and the application launch status will be set to 9 (Bad memory configuration).
Please refer to AT+WOPEN=7 description in AT Commands Interface Guide for more information.

Note: *Stack memory is limited to 64 kBytes and if allocated above 64 kBytes correct behavior is not guaranteed.*

Name

Task identification string, used for debug purpose with Traces & Errors services.

Priority

Task priority level, relatively to the other tasks declared in the table. The higher is the number, the higher is the priority level. Priorities values declared in the table should be from 1 to the tasks count. This priority determines the order in which the events are notified to the several tasks when several ones receive information at the same time.

Note: *All the priorities declared in the table have to be different (two tasks can not have the same priority level).*
If there is an error in the priorities declaration, the Firmware will refuse to launch the application, and the application launch status will be set to 17 (Bad priority value)
Please refer to AT+WOPEN=7 description in AT Commands Interface Guide for more information.

3.1.2.3. [Deprecated] Single task initialization

For ascendant compatibility purpose, the former way of declaring the application entry point is still supported.

As soon as the tasks initialisation table is *NOT* provided, ADL looks for a single entry point function:

```
void adl_main ( adl_InitType_e init )
{
        // TODO: add your init code here
}
```

and for a constant defining the application's call stack size, in bytes:

```
const u16 wm_apmCustomStackSize = 3*1024;
```

Note: *Note: as soon as the tasks initialization table is provided, any adl_main function will be ignored.*

3.1.3. Interrupt Handlers Call Stack Sizes Declaration

Interfaces dedicated to the interrupt handlers call stack sizes declaration.

3.1.3.1. Low level interrupt handler call stack size.

Call stack size (in bytes) of the Low level interrupt handler execution context. If the application wishes to handle interruptions (cf. IRQ Service chapter & Execution Context Service chapter), it has also to define the required contexts (low level and/or high level) call stack sizes.

```
const u32 adl_InitIRQLowLevelStackSize
```

Note: *This definition is optional if the application does not plan to use the IRQ service.*
The Real Time Enhancement feature has to be enabled on the embedded module if the application requires this call stack to be greater than zero.
The Real Time Enhancement feature state can be read thanks to the ***AT+WCFM=5*** *command response value: Please refer to the AT Commands Interface Guide for more information.*

Note: *Please contact your Sierra Wireless distributor for more information on how to enable this feature on the embedded module.*

Note: *If this call stack is declared, and if the feature is not enabled on the embedded module, the Firmware will refuse to launch the application, and the application launch status will be set to 19 (Real Time feature not enabled)*
Please refer to. ***AT+WOPEN=7*** *description in AT Commands Interface Guide for more information.*

3.1.3.2. High level interrupt handler call stack size

Call stack size (in bytes) of the High level interrupt handler execution context. If the application whishes to handle interruptions (cf. IRQ Service chapter & Execution Context Service chapter), it has also to define the required contexts (low level and/or high level) call stack sizes.

```
const u32 adl_InitIRQHighLevelStackSize
```

Note: *This definition is optional if the application does not plan to use the IRQ service, or just low level interrupt handlers.*
The Real Time Enhancement feature has to be enabled on the embedded module if the application requires this call stack to be greater than zero.
*The Real Time Enhancement feature state can be read thanks to the **AT+WCFM=5** command response value: Please refer to the AT Commands Inteface guide 1 for more information.*

Note: *Please contact your Sierra Wireless distributor for more information on how to enable this feature on the embedded module.*

Note: *If this call stack is declared, and if the feature is not enabled on the embedded module, the Firmware will refuse to launch the application, and the application launch status will be set to 19 (Real Time feature not enabled). Please refer to. **AT+WOPEN=7** description in AT Commands Interface Guide for more information.*

3.1.4. Initialization information

3.1.4.1. The adl_InitType_e Type

Details of the reason of the embedded module boot.

Code

```
typedef enum
{
        ADL_INIT_POWER_ON,
        ADL_INIT_REBOOT_FROM_EXCEPTION,
        ADL_INIT_DOWNLOAD_SUCCESS,
        ADL_INIT_DOWNLOAD_ERROR,
        ADL_INIT_RTC,
} adl_InitType_e;
```

Description

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_INIT_RTC:	Power-on due to an RTC alarm (cf. the AT+CALA command documentation for more information).

3.1.4.2. The adl_InitGetType function

Returns the last embedded module power-on or reset reason.

Prototype

```
adl_InitType_e adl_InitGetType (void )
```

Returned value

- The embedded module reset reason. (Please refer to adl_InitType_e description for more information).

Example:

This example demonstrates how to use the function **adl_InitGetType** in a nominal case.

```
// Anywhere in the application code, to retrieve init type.
  adl_InitType_e InitType = adl_InitGetType();
```

3.1.5. Miscellaneous name and version related information

The constants defined below allows the application to define some information readable by the Sierra Wireless Firmware. These constants definitions are optional, and automatically considered as empty strings if not provided by the application.

3.1.5.1. Application name

This constant string should be defined by the application in order to provide a name readable by the Sierra Wireless Firmware.

```
const ascii adl_InitApplicationName[]
```

3.1.5.2. Company name

This constant string should be defined by the application, in order to provide a company name readable by the Sierra Wireless Firmware.

```
const ascii adl_InitCompanyName[]
```

3.1.5.3. Application version

This constant string should be defined by the application in order to provide a version readable by the Sierra Wireless Firmware.

```
const ascii adl_InitApplicationVersion[]
```

3.1.5.4. Example

```
// Application name definition
const ascii adl_InitApplicationName[] = "My Application";

// Company name definition
const ascii adl_InitCompanyName[] = "My Company";

// Application version definition
const ascii adl_InitApplicationVersion[] = "v1.0.0";
```

3.1.6. Stack Sizes Macro

The constants defined below allows the application to define the stack sizes.

3.1.6.1. The ADL_DECLARE_CALL_STACK

Application stack size Macro.

Code

```
#define ADL_DECLARE_CALL_STACK (X) const u16    wm_apmCustomStackSize = X
```

Description

`ADL_DECLARE_CALL_STACK:`

This macro declares the right **`wm_apmCustomStackSize`** value according to the compilers.

The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than ARM compilers.

If the GCC compiler is used, the allocation has to be declared with greater stack sizes (the X parameter is then multiplied by 3).

3.1.6.2. The ADL_DECLARE_LOWIRQ_STACK

Low level interrupt handler call stack size Macro.

Code

```
#define ADL_DECLARE_LOWIRQ_STACK(X) const u32 adl_InitIRQLowLevelStackSize = X
```

Description

`ADL_DECLARE_LOWIRQ_STACK:`

This macro declares the right **`adl_InitIRQLowLevelStackSize`** value according to the compilers.

The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than ARM compilers.

If the GCC compiler is used, the allocation has to be declared with greater stack sizes (the X parameter is then multiplied by 3).

3.1.6.3. The ADL_DECLARE_HIGHIRQ_STACK

High level interrupt handler call stack size Macro.

Code

```
#define ADL_DECLARE_HIGHIRQ_STACK(X)constu32 adl_InitIRQHighLevelStackSize = X
```

Description

`ADL_DECLARE_HIGHIRQ_STACK:`

This macro declares the right **`adl_InitIRQHighLevelStackSize`** value according to the compilers.

The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than ARM compilers.

If the GCC compiler is used, the allocation has to be declared with greater stack sizes (the X parameter is then multiplied by 3).

3.1.7. Interrupt priorities change

3.1.7.1. Detailed description

The constants defined below allows the application to change some interrupt priorities in the Firmware. This possibility is optional, and automatically considered as the default priority if not provided by the application. The default priorities for the Firmware interrupts are:

- Priority 0 (highest priority): UART1
- Priority 1: FINT1
- Priority 2: FINT0
- Priority 3: PIO
- Priority 4: EXTINT1
- Priority 5: EXTINT2
- Priority 6 (lowest priority): EXTINT3, RTC (Real Time Clock), USB Fiq, USB irq, KBS (Keyboard), SCTU1, SCTU2, UART2, SPI1, SPI2, SPI3, I2C, DMAU, USIM

Warning: *Changing the interrupt priority is at the whole customer responsibility. The Firmware was tested and validated only in the default configuration.*

3.1.7.2. Example

```
// Change UART 2 interrupt priority
const adl_InitInterrupts_t adl_InitApplicationInterruptPrio[] =
{
    { ADL_IRQ_TYPE_UART, ADL_IRQ_INSTANCE_2,   ADL_IRQ_PRIORITY_0   },
    { ADL_IRQ_TYPE_MAX,  ADL_IRQ_INSTANCE_MAX, ADL_IRQ_PRIORITY_MAX } // This
line should always be the last line of this table
};
```

Notes

When an interrupt table is present in the Open AT application, a trace is displayed under TMT at device start up. For the above example, the following traces are displayed:

```
Trace       1   ***************  WARNING  ***************
Trace       1   Interruption priority change: UART2 current priority 0, default
priority 6
Trace       1   ************* WARNING  END **************
```

If an interrupt is defined several time in this table, only the last priority change will be taken into account.

When an interrupt table is present in the Open AT application and when an error is present in this table, the Open AT application is not started (the AT+WOPEN=7 returns +WOPEN: 7,21 response) and a trace is displayed under TMT at device start up.

```
Trace       1   ***************  WARNING  ***************
Trace       1   Interrupt priority table is not correct
Trace       1   ************* WARNING  END **************
```

3.1.7.3. The adl_InitInterrupts_t Structure

Firmware interrupts priorities declaration structure, used to format the adl_InitApplicationInterruptPrio table.

Code:

```
typedef struct
{
      u16         InterruptType;
      u8          InterruptInstance;
      u8          InterruptPriority;
} adl_InitInterrupts_t;
```

Description

InterruptType

Interrupt Type on which the priority has to be changed by the application.

Note: *If the interrupt is not supported by the platform, the Firmware will refuse to change the Firmware interrupt priorities, and the application launch status will be set to 21. (cf.* **AT+WOPEN=7** *description in* *AT Commands Interface Guide* *for more information)*

InterruptInstance

Interrupt instance on which the priority has to be changed by the application.

InterruptPriority

Priority of the interrupt.

Define the priority required by the application for the corresponding interrupt.

3.1.7.4. Type Definition : The adl_InitInterrupts_t Type

Firmware interrupts priorities declaration structure, used to format the adl_InitApplicationInterruptPrio table.

```
typedef struct _adl_InitInterrupts_t adl_InitInterrupts_t;
```

3.1.7.5. The adl_InterruptCategoryId_e Type

Details on the Embedded module boot reason.

Code

```
typedef enum
{
        ADL_IRQ_TYPE_FINT,
        ADL_IRQ_TYPE_UART,
        ADL_IRQ_TYPE_USB,
        ADL_IRQ_TYPE_EXTINT,
        ADL_IRQ_TYPE_RTC,
        ADL_IRQ_TYPE_KBD,
        ADL_IRQ_TYPE_TIMER,
        ADL_IRQ_TYPE_SPI,
        ADL_IRQ_TYPE_I2C,
        ADL_IRQ_TYPE_DMAU,
        ADL_IRQ_TYPE_USIM,
        ADL_IRQ_TYPE_LAST,
        ADL_IRQ_TYPE_MAX = 0xFF
} adl_InterruptCategoryId_e;
```

Description

`ADL_IRQ_TYPE_FINT:`	Mask for FINT.
`ADL_IRQ_TYPE_UART:`	Mask for UART.
`ADL_IRQ_TYPE_USB:`	Mask for USB.
`ADL_IRQ_TYPE_EXTINT:`	Mask for External Interrupt.
`ADL_IRQ_TYPE_RTC:`	Mask for RTC.
`ADL_IRQ_TYPE_KBD:`	Mask for Keyboard.
`ADL_IRQ_TYPE_TIMER:`	Mask for Timer.
`ADL_IRQ_TYPE_SPI:`	Mask for SPI.
`ADL_IRQ_TYPE_I2C:`	Mask for I2C.
`ADL_IRQ_TYPE_DMAU:`	Mask for DMA.
`ADL_IRQ_TYPE_USIM:`	Mask for SIM.
`ADL_IRQ_TYPE_LAST:`	Non significant value (should not be used).
`ADL_IRQ_TYPE_MAX:`	Non significant value (should not be used).

3.1.7.6. The adl_InterruptId_e Type

Details the instances for the Firmware interrupts. See PTS of the platform for more details. Examples: FINT has 2 instances, UART has 2 instances, RTC has 1 instance Possibilities for WMP100:

- FINT: **ADL_IRQ_INSTANCE_1** for FINT0, **ADL_IRQ_INSTANCE_2** for FINT1
- UART: **ADL_IRQ_INSTANCE_1** for UART1, **ADL_IRQ_INSTANCE_2** for UART2
- USB: **ADL_IRQ_INSTANCE_1** for USB FIQ, **ADL_IRQ_INSTANCE_2** for USB IRQ
- EXTINT: **ADL_IRQ_INSTANCE_1** for EXTINT1, **ADL_IRQ_INSTANCE_2** for EXTINT2, **ADL_IRQ_INSTANCE_3** for EXTINT3
- RTC: **ADL_IRQ_INSTANCE_1**

- KEYBOARD: **ADL_IRQ_INSTANCE_1**
- Timer: **ADL_IRQ_INSTANCE_1** for Hardware Timer 1, **ADL_IRQ_INSTANCE_2** for Hardware Timer 2
- SPI: **ADL_IRQ_INSTANCE_1** for SPI1, **ADL_IRQ_INSTANCE_2** for SPI2, **ADL_IRQ_INSTANCE_3** for SPI3
- I2C: **ADL_IRQ_INSTANCE_1**
- DMA: **ADL_IRQ_INSTANCE_1**
- SIM: **ADL_IRQ_INSTANCE_1**

Code

```
typedef enum
{
        ADL_IRQ_INSTANCE_1 = 1,
        ADL_IRQ_INSTANCE_2,
        ADL_IRQ_INSTANCE_3,
        ADL_IRQ_INSTANCE_LAST,
        ADL_IRQ_INSTANCE_MAX = 0xFF
} adl_InterruptId_e;
```

Description

`ADL_IRQ_INSTANCE_1:`	Instance 1 of the mask.
`ADL_IRQ_INSTANCE_2:`	Instance 2 of the mask
`ADL_IRQ_INSTANCE_3:`	Instance 3 of the mask
`ADL_IRQ_INSTANCE_LAST:`	Non significant value (should not be used)
`ADL_IRQ_INSTANCE_MAX:`	Non significant value (should not be used).

3.1.7.7. The adl_InterrupPriority_e Type

Details the priority for the Firmware interrupts.

Code

```
typedef enum
{
        ADL_IRQ_PRIORITY_0,
        ADL_IRQ_PRIORITY_1,
        ADL_IRQ_PRIORITY_2,
        ADL_IRQ_PRIORITY_3,
        ADL_IRQ_PRIORITY_4,
        ADL_IRQ_PRIORITY_5,
        ADL_IRQ_PRIORITY_6,
        ADL_IRQ_PRIORITY_LAST,
        ADL_IRQ_PRIORITY_MAX = 0xFF
} adl_InterrupPriority_e;
```

Description

`ADL_IRQ_PRIORITY_0:`	Priority 0: highest priority.
`ADL_IRQ_PRIORITY_1:`	Priority 1.
`ADL_IRQ_PRIORITY_2:`	Priority 2.
`ADL_IRQ_PRIORITY_3:`	Priority 3.
`ADL_IRQ_PRIORITY_4:`	Priority 4.
`ADL_IRQ_PRIORITY_5:`	Priority 5.
`ADL_IRQ_PRIORITY_6:`	Priority 6.
`ADL_IRQ_PRIORITY_LAST:`	Non significant value (should not be used).
`ADL_IRQ_PRIORITY_MAX:`	Non significant value (should not be used).

3.1.7.8. Variable : Firmware interrupt priority change requested by the application

This table allows an application to change the priority of the Firmware interrupts.

```
const adl_InitInterrupts_t adl_InitApplicationInterruptPrio[]
```

3.1.8. Example

The code sample below illustrates a nominal use case of the ADL Application Entry Points public interface.

```
// Application tasks declaration table
const adl_InitTasks_t adl_InitTasks [] =
{
    { MyFirstEntryPoint,  1024, "MYTASK1", 3 },
    { MySecondEntryPoint, 1024, "MYTASK2", 2 },
    { MyThirdEntryPoint,  1024, "MYTASK3", 1 },
    { 0, 0, 0, 0 }
};

// Low level handlers execution context call stack size
const u32 adl_InitIRQLowLevelStackSize = 1024;

// High level handlers execution context call stack size
const u32 adl_InitIRQHighLevelStackSize = 1024;
```

3.2. Basic Features

3.2.1. Data Types

The available data types are described in the **wm_types.h** file. They ensure compatibility with the data types used in the functional prototypes and are used for both Target and RTE generation.

3.2.2. List Management

3.2.2.1. Type Definition

3.2.2.1.1. The wm_lst_t Type

This type is used to handle a list created by the list API.

```
typedef void * wm_lst_t;
```

3.2.2.1.2. The wm_lstTable_t Structure

This structure is used to define a comparison callback and an Item destruction callback:

```
typedef struct
{
       s16  ( * CompareItem )   ( void *, void * );
       void ( * FreeItem )   ( void * );
} wm_lstTable_t;
```

The **CompareItem** callback is called every time the list API needs to compare two items.

It returns:

- OK when the two provided elements are considered similar.
- –1 when the first element is considered smaller than the second one.
- 1 when the first element is considered greater than the second one.

If the **CompareItem** callback is set to NULL, the **wm_strcmp** function is used by default.

The **FreeItem** callback is called each time the list API needs to delete an item. It should then perform its specific processing before releasing the provided pointer.

If the **FreeItem** callback is set to NULL, the **wm_osReleaseMemory** function is used by default.

3.2.2.2. The wm_lstCreate Function

The **wm_lstCreate function** allows to create a list, using the provided attributes and callbacks.

Prototype

```
wm_lst_t wm_lstCreate (  u16              Attr,
                         wm_lstTable_t *  funcTable );
```

Parameters

Attr:

List attributes, which can be combined by a logical OR among the following defined values:

- **`WM_LIST_NONE`**: no specific attribute ;
- **`WM_LIST_SORTED`**: this list is a sorted one (see the wm_lstAddItem section and wm_lstinsertitem section descriptions for more details);
- **`WM_LIST_NODUPLICATES`**: this list does not allow duplicate items (see the wm_lstAddItem section and wm_lstinsertitem section descriptions for more details).

funcTable:

Pointer on a structure containing the comparison and the item destruction callbacks.

Returned values

- This function returns a list pointer corresponding to the created list. This must be used in all further operations on this list.

3.2.2.3. The wm_lstDestroy Function

The wm_lstDestroy function allows to clear and then destroy the provided list.

Prototype

```
void wm_lstDestroy ( wm_lst_t   list );
```

Parameters

list:

The list to destroy.

Note: *This function calls the **`FreeItem`** callback (if defined) on each item to delete it, before destroying the list:*

3.2.2.4. The wm_lstClear Function

The **`wm_lstClear`** function allows to clear all the provided list items, without destroying the list itself (please refer to wm_lstdeleteitem function for notes on item deletion).

Prototype

```
void wm_lstClear ( wm_lst_t     list );
```

Parameters

list:

The list to clear.

Note: *This function calls the **`FreeItem`** callback (if defined) on each item to delete it.*

3.2.2.5. The wm_lstGetCount Function

The **`wm_lstGetCount`** function returns the current item count.

Prototype

```
u16 wm_lstGetCount ( wm_lst_t   list );
```

Parameters

list:

The list from which to get the item count.

Returned values

- The number of items of the provided list. The function returns 0 if the list is empty.

3.2.2.6. The wm_lstAddItem Function

The **`wm_lstAddItem`** function allows to add an item to the provided list.

Prototype

```
s16 wm_lstAddItem (   wm_lst_t      list,
                      void *        item );
```

Parameters

list:

The list to add an item to.

item:

The item to add to the list.

Returned values

- The position of the added item, or ERROR if an error occurred.

*Note: The **`item`** pointer should not point on a **`const`** or local buffer, as it is released in any item destruction operation.*

*Note: If the list has the WM_LIST_SORTED attribute, the item is inserted in the appropriate place after calling of the **`CompareItem`** callback (if defined). Otherwise, the item is appended at the end of the list.*

*Note: If the list has the WM_LIST_NODUPLICATES, the item is not inserted when the **`CompareItem`** callback (if defined) returns 0 on any previously added item. In this case, the returned index is the existing item index.*

3.2.2.7. The wm_lstInsertItem Function

The **`wm_lstInsertItem`** function allows to insert an item to the provided list at the given location.

Prototype

```
s16 wm_lstInsertItem (   wm_lst_t  list,
                         void *    item,
                         u16       index );
```

Parameters

list:

The list to add an item to.

item:

The item to add to the list.

index:

The location where to add the item.

Returned values

- The position of the added item, or **`ERROR`** if an error occured.

Note: The item pointer should not point on a const or local buffer, as it is released in any item destruction operation.

Note: This function does not take list attributes into account and always inserts the provided item in the given index.

3.2.2.8. The wm_lstGetItem Function

The **wm_lstGetItem** function allows to read an item from the provided list, in the given index.

Prototype

```
void * wm_lstGetItem (   wm_lst_t  list,
                         u16       index );
```

Parameters

list:

The list from which to get the item.

index:

The location where to get the item.

Returned values

- A pointer on the requested item, or **NULL** if the index is not valid.

3.2.2.9. The wm_lstDeleteItem Function

The **wm_lstDeleteItem** function allows to delete an item of the provided list in the given indices.

Prototype

```
s16 wm_lstDeleteItem (   wm_lst_t  list,
                         u16       index );
```

Parameters

list:

The list to delete an item from.

index:

The location where to delete the item.

Returned values

- The number of remaining items in the list, or **ERROR** if an error did occur.

*Note: This function calls the **FreeItem** callback (if defined) on the requested item to delete it.*

3.2.2.10. The wm_lstFindItem Function

The **wm_lstFindItem** function allows to find an item in the provided list.

Prototype

```
s16 wm_lstFindItem (   wm_lst_t  list,
                       void *    item );
```

Parameters

list:

The list where to search.

item:

The item to find.

Returned values

- The index of the found item if any, ERROR otherwise.

Note: *This function calls the* **`CompareItem`** *callback (if defined) on each list item, until it returns 0.*

3.2.2.11. The wm_lstFindAllItem Function

The **`wm_lstFindAllItem`** function allows to find all items matching the provided one, in the given list.

Prototype

```
s16 * wm_lstFindAllItem (   wm_lst_t  list,
                            void *    item );
```

Parameters

list:

The list where to search.

item:

The item to find.

Returned values

- A s16 buffer containing the indices of all the items found, and ending with ERROR.

Note: *This buffer should be released by the application when its processing is done.*

Note: *This function calls the* **`CompareItem`** *callback (if defined) on each list item to get all those which match the provided item*

This function should be used only if the list cannot be changed during the resulting buffer processing. Otherwise the **`wm_lstFindNextItem`** *should be used.*

3.2.2.12. The wm_lstFindNextItem Function

The **`wm_lstFindNextItem`** function allows to find the next item index of the given list, which corresponds with the provided one.

Prototype

```
s16 wm_lstFindNextItem ( wm_lst_t  list,
                         void *    item );
```

Parameters

list:

The list to search in.

item:

The item to find.

Returned values

- The index of the next found item if any, otherwise ERROR.

*Note: This function calls the **`CompareItem`** callback (if defined) on each list item to get those which match with the provided item. It should be called until it returns **`ERROR`**, in order to get the index of all items corresponding to the provided one. The difference with the **`wm_lstFindAllItem`** function is that, even if the list is updated between two calls to **`wm_lstFindNextItem`**, the function does not return a previously found item. To restart a search with the **`wm_lstFindNextItem`**, the **`wm_lstResetItem`** should be called first.*

3.2.2.13. The wm_lstResetItem Function

The **`wm_lstResetItem`** function allows to reset all previously found items by the **`wm_lstFindNextItem`** function.

Prototype

```
void wm_lstResetItem (   wm_lst_t  list,
                         void *    item );
```

Parameters

list:

The list to search in.

item:

The item to search, in order to reset all previously found items.

*Note: This function calls the **`CompareItem`** callback (if defined) on each list item to get those which match with the provided one.*

3.2.3. Standard Library

3.2.3.1. Standard C Function Set

The available standard APIs are defined below:

```
ascii *   wm_strcpy     ( ascii * dst, ascii * src );
ascii *   wm_strncpy    ( ascii * dst, ascii * src, u32 n );
ascii *   wm_strcat     ( ascii * dst, ascii * src );
ascii *   wm_strncat    ( ascii * dst, ascii * src, u32 n );
u32       wm_strlen     ( ascii * str );
s32       wm_strcmp     ( ascii * s1, ascii * s2 );
s32       wm_strncmp    ( ascii * s1, ascii * s2, u32 n );
s32       wm_stricmp    ( ascii * s1, ascii * s2 );
s32       wm_strnicmp   ( ascii * s1, ascii * s2, u32 n );
ascii *   wm_memset     ( ascii * dst, ascii c, u32 n );
ascii *   wm_memcpy     ( ascii * dst, ascii * src, u32 n );
s32       wm_memcmp     ( ascii * dst, ascii * src, u32 n );
ascii *   wm_itoa       ( s32 a, ascii * szBuffer );
s32       wm_atoi       ( ascii * p );
u8        wm_sprintf    ( ascii * buffer, ascii * fmt, ... );
```

Important remark about GCC compiler:

When using GCC compiler, due to internal standard C library architecture, it is strongly not recommended to use the "%f" mode in the wm_sprintf function in order to convert a float variable to a string. This leads to an ARM exception (product reset).

A way around for this conversion is:

```
float MyFloat;                      // float to display
ascii MyString [ 100 ];             // destination string
s16 d,f;
d = (s16) MyFloat * 1000;           // Decimal precision: 3 digits
f = ( MyFLoat * 1000 ) - d;         // Decimal precision: 3 digits
wm_sprintf ( MyString, "%d.%03d", (s16)MyFloat, f );
                                    // Decimal precision: 3 digits
```

3.2.3.2. String Processing Function Set

Some string processing functions are also available in this standard API.

All the following functions leads to an ARM exception if a requested **ascii *** parameter is NULL.

ascii wm_isascii (ascii *c*);

Returns c if it is an ascii character (‘a’/’A’ to ‘z’/’Z’), 0 otherwise.

ascii wm_isdigit (ascii *c*);

Returns c if it is a digit character (‘0’ to ‘9’), 0 otherwise.

ascii wm_ishexa (ascii *c*);

Returns c if it is a hexadecimal character (‘0’ to ‘9’, ‘a’/’A’ to ‘f’/’F’), 0 otherwise.

bool wm_isnumstring (ascii * *string*);

Returns TRUE if string is a numeric one, FALSE otherwise.

bool wm_ishexastring (ascii * *string*);

Returns TRUE if string is a hexadecimal one, FALSE otherwise.

bool wm_isphonestring (ascii * *string*);

Returns TRUE if string is a valid phone number (national or international format), FALSE otherwise.

u32 wm_hexatoi (ascii * *src*, u16 *iLen*);

If src is a hexadecimal string, converts it to a returned u32 of the given length, and 0 otherwise. As an example: wm_hexatoi ("1A", 2) returns 26, wm_hexatoi ("1A", 1) returns 1

u8 * wm_hexatoibuf (u8 * *dst*, ascii * *src*);

If src is a hexadecimal string, converts it to an u8 * buffer and returns a pointer on dst, and NULL otherwise. As an example, wm_hexatoibuf (dst, "1F06") returns a 2 bytes buffer: 0x1F and 0x06

ascii* wm_itohexa (ascii * *dst*, u32 *nb*, u8 *len*);

Converts nb to a hexadecimal string of the given length and returns a pointer on dst. For example, wm_itohexa (dst, 0xD3, 2) returns "D3", wm_itohexa (dst, 0xD3, 4) returns "00D3".

```
ascii* wm_ibuftohexa    ( ascii * dst, u8 * src, u16 len );
```

Converts the u8 buffer src to a hexadecimal string of the given length and returns a pointer on dst. Example with the src buffer filled with 3 bytes (0x1A, 0x2B and 0x3C), wm_ibuftohexa (dst, src, 3) returns "1A2B3C").

```
u16    wm_strSwitch     ( const ascii * strTest, ... );
```

This function must be called with a list of strings parameters, ending with NULL. strTest is compared with each of these strings (on the length of each string, with no matter of the case), and returns the index (starting from 1) of the string which matches if any, 0 otherwise.

Example:

wm_strSwitch ("TEST match", "test", "no match", NULL") returns 1, wm_strSwitch ("nomatch", "nomatch a", "nomatch b", NULL) returns 0.

```
ascii * wm_strRemoveCRLF ( ascii * dst, ascii * src, u16 size );
```

Copy in dst buffer the content of src buffer, removing CR (0x0D) and LF (0x0A) characters, from the given size, and returns a pointer on dst.

```
ascii * wm_strGetParameterString (   ascii *          dst,
                                     const ascii *    src,
                                     u16              Position );
```

If src is a string formatted as an AT response (for example "+RESP:P1,P2,P3") or as an AT command (for example "AT+CMD=P1,P2,P3"), the function copies the parameter at Position offset (starting from 1) if it is present in the src buffer, and returns a pointer on dst. It returns NULL otherwise.

Note: *The response RESP: P1,,P3 is considered to contain three parameters:*

** Parameter number 1 is present, and has the value "P1";*

** Parameter number 2 is present, and has a null value;*

** Parameter number 3 is present, and has the value "P3";*

** Parameters numbered 4 & above are not present.*

Example:

wm_strGetParameterString (dst, "+WIND: 4", 1) returns "4",

wm_strGetParameterString (dst, "+WIND: 5,1", 2) returns "1",

wm_strGetParameterString (dst, "AT+CMGL=\"ALL\"", 1) returns "ALL".

3.3. AT Commands Service

3.3.1. Required Header File

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

`adl_at.h`

3.3.2. Unsolicited Responses

An unsolicited response is a string sent by the Sierra Wireless Firmware 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 Sierra Wireless Firmware.

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.

3.3.2.1. 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 Sierra Wireless Firmware, 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.

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_port_e  Dest;         // Unsolicited response destination port
        u16         StrLength;    /* the length of the string (name) of the
                                  unsolicited response */
        bool        RiPulse;      // Indicate if RI signal must be pulsed
        u8          Pad[3];       // not used
        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.

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.

The RiPulse field indicates if RI signal will be pulsed or not. The RI signal is pulsed if this field is set to TRUE. If it is set to TRUE, application can set to FALSE to not pulse the RI signal. If it is set to FALSE, modification of this field by application has no impact, RI signal will not be pulsed. Refer to "+WRIM" AT command in the "AT command interface guide" to get more information about RI signal.

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.*
In case of several handlers associated to the same unsolicited response, by default if RiPulse is set to TRUE, if at least one handler set RiPulse to FALSE, RI signal is not pulsed.
If the unsolicited string is not sent to the external application then RI signal is not pulsed even if "RiPulse" flag is set to TRUE.

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).

3.3.2.2. 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 interrupt handler (the function is forbidden in this context)

Note: *The RI pulse generation behaviour depends on "+WRIM" AT command parameter: RI pulse duration depends on **pulse_width** parameter of "+WRIM" AT command.*

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);
}
```

3.3.3. Responses

ADL AT responses sending.

The defined operations are:

- **adl_atSendResponse** sending of the text provided as a response with the type provided to the port provided
- **adl_atSendResponseSpe** with the NI provided the command associated is found if it had subscribed to the response provided the response handler is called else the response is sent to the port provided
- **adl_atSendStdResponse** sending of the standard response provided as a response with the type provided to the port provided
- **adl_atSendStdResponseSpe** with the NI provided the command associated is found if it had subscribed to the standard response provided the response handler is called else the standard response is sent to the port provided
- **adl_atSendStdResponseExt** sending of standard response with an argument provided as a response with the type provided to the port provided
- **adl_atSendStdResponseExtSpe** with the NI provided the command associated is found if it had subscribed to the standard response with an argument provided the response handler is called else the standard response with an argument is sent to the port provided
- **adl_atSendStdResponseExtStr** sending of standard response with a string argument provided as a response with the type provided to the port provided
- **adl_atSendUnsoResponse** sending of an unsolicited response with a string argument provided as a reponse with the port provided and RI flag provided

Note: ***adl_atSendResponseSpe**, **adl_atSendStdResponseSpe**, **adl_atSendStdResponseExtSpe** are to be used with **adl_atCmdSendExt** function.*

Note: ***adl_atCmdSendExt*** *stacks command when call in a command handler to resend the command whereas* ***adl_atSendResponseSpe****,* ***adl_atSendStdResponseSpe****,* ***adl_atSendStdResponseExtSpe*** *unstacks the command and call the appropriate response handler (if any).*

3.3.3.1. Required Header File

The header file for the functions dealing with ADL AT Response Sending Service public interface is:

adl_RspHandler.h

3.3.3.2. 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 *   Text );
```

Parameters

Type:

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

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

Text:

The string of the response.

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.3. The adl_atSendResponseSpe Function

This function sends the provided text as a response, an unsolicited response or an intermediate response, according to the requested type. With the NI provided, the associated command is found. If the command had subscribed to this reponse, then the response handler is called. Otherwise, the response is sent to the port provided.

Prototype

```
s32 adl_atSendResponseSpe ( u16       Type,
                            ascii*    Text,
                            u16       NI );
```

Parameters

Type:

This parameter is composed of the response type, and the destination port where to send the response. The type and destination combination has to be done with the following macro:
ADL_AT_PORT_TYPE (_port,   _type).

Text:

The string of the response.

NI:

Notification Identifier to find the associate command.

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.4. 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:

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)**.

RspID:

The ID of the response.

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.5. The adl_atSendStdResponseSpe Function

This function sends the provided standard response as a response, an unsolicited response or an intermediate response, according to the requested type. With the NI provided, the associated command is found. If the command had subscribed to this standard response, then the response handler is called. Otherwise, the standard response is sent to the port provided.

Prototype

```
s32 adl_atSendStdResponseSpe ( u16            Type,
                               adl_strID_e    RspID,
                               u16            NI );
```

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)**.

RspID:

The ID of the response.

NI:

Notification Identifier to find the associate command.

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.6. 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 ( u16          Type,
                               adl_strID_e  RspID,
                               u32          arg );
```

Parameters

Type:

This parameter is composed of the response type, and the destination port where to send the response. The type and destination combination has to be done with the following macro: **ADL_AT_PORT_TYPE (_port, _type)**.

RspID:

The ID of the response.

arg:

Standard response argument (being a u32).

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.7. The adl_atSendStdResponseExtSpe Function

This function sends the provided standard response with an argument as a response, an unsolicited response or an intermediate response, according to the requested type. With the NI provided, the associated command is found. If the command had subscribed to this standard response with an argument, then the response handler is called. Otherwise, the standard response with an argument is sent to the port provided.

Prototype

```
s32 adl_atSendStdResponseExtSpe ( u16          Type,
                                  adl_strID_e  RspID,
                                  u32          arg,
                                  u16          NI );
```

Parameters

Type:

This parameter is composed of the response type, and the destination port where to send the response. The type and destination combination has to be done with the following macro: **ADL_AT_PORT_TYPE (_port, _type)**.

RspID:

The ID of the response.

arg:

Standard response argument (being a u32).

NI:

Notification Identifier to find the associate command.

Returned values

- `OK` on success
- `ADL_RET_ERR_SERVICE_LOCKED` if the function was called from a low level interrupt handler

3.3.3.8. The adl_atSendStdResponseExtStr 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_atSendStdResponseExtStr (  u8           Type,
                                   adl_strID_e  RspID,
                                   ascii*       arg );
```

Parameters

Type:

This parameter is composed of the response type, and the destination port where to send the response. The type and destination combination has to be done with the following macro: **`ADL_AT_PORT_TYPE (_port, _type)`**.

RspID:

The ID of the response

arg:

Standard response argument (being a string).

Returned values

- `OK` on success
- `ADL_RET_ERR_SERVICE_LOCKED` if the function was called from a low level interrupt handler

3.3.3.9. The adl_atSendUnsoResponse Function

This function sends the text provided to the required port as an unsolicited response with the RIpulse flag to allows to generate a RI pulse. Refer to "+WRIM" AT command in the "AT command interface guide" to get more information about RI signal.

Prototype

```
s32 adl_atSendUnsoResponse  (  adl_port_e   Port,
                               ascii*       Text,
                               bool         RIpulse );
```

Parameters

Port:

The destination port where to send the response.

Text:

The text to be sent.

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

RIpulse:

RI pulse flag, if TRUE, RI signal is pulsed.

Note: *The RI pulse generation behaviour depends on "+WRIM" AT command parameter:*
- if mode parameter of "+WRIM" AT command is set to 0, RI signal cannot be pulsed by ***adl_atSendUnsoResponse****.*
- RI pulse duration depends on pulse_width parameter of "+WRIM" AT command.

Returned values

- **OK** on success
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler

3.3.3.10. 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_AT_RSP0X01
#define ADL_AT_UNS0X02
#define ADL_AT_INT0X04
#define ADL_AT_PORT_TYPE(_port_, _type_)(u16) (_port_ << 8 | _type_)
#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)
```

Description

ADL_AT_RSP:	The text/ID associated to this type will be sent as a standard or 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_UNS:	The text/ID associated to this type will be sent as an unsollicited 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 (i.e. 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.
ADL_AT_INT:	The text/ID associated to this type will be sent as an 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_PORT_TYPE:`	The _port argument has to be a defined value of the **`adl_port_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: *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 opened ports.* Note: *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 3.31 port. If this port is not opened, responses will not be displayed.*
`adl_atSendResponsePort:`	Additional Port parameter definition for response sending function **`adl_atSendResponse`**.
`adl_atSendStdResponsePort:`	Additional Port parameter definition for response sending function **`adl_atSendStdResponse`**.
`adl_atSendStdResponseExtPort:`	Additional Port parameter definition for response sending function **`adl_atSendStdResponseExt`**.

3.3.4. Incoming AT Commands

An ADL application may subscribe to an AT command string, in order to receive events each time either an external application sends this AT command on one of the embedded module's ports or this AT command is sent with **adl_atCmdSendExt** API (and an appropriate NI parameter). 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 either by an external application or with **adl_atCmdSendExt** API.

Multiple subscriptions: An application subscribes to a command with a handler (handler1) and subscribes then to the same command with another handler (handler2). Every time this command is sent either by the external application or with **adl_atCmdSendExt** API the last subscribed handler (handler2) will be called. Handler1 will only be called if handler2 resends the subscribed command with **adl_atCmdSendExt** API and the provided NI.

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 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 to 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.

The defined operations are:

- A **adl_atCmdSubscribeExt** function to subscribe to a command with providing a Context.
- A **adl_atCmdSubscribe** function to subscribe to a command without providing a Context.
- A **adl_atCmdUnSubscribe** function to unsubscribe to a command.

3.3.4.1. Required Header File

The required header file is:

adl_CmdHandler.h

3.3.4.2. The adl_atCmdPreParser_t Structure

This structure contains information about AT command.

Code

```
typedef struct
{
      u16          Type;          // Type
      u8           NbPara;        // Number of parameters
      adl_port_e   Port;          // Port
      wm_lst_t     ParaList;      // List of parameters
      u16          StrLength;     // Incoming command length
      u16          NI;            // Notification Identifier
      void *       Contxt;        // Context
      ascii        StrData[1];    // Incoming command address
} adl_atCmdPreParser_t;
```

Description

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:

Only if Type is **ADL_CMD_TYPE_PARA**. Each parameter may be accessed by the **ADL_GET_PARAM(_p,_i)** macro. If a string parameter is provided (e.g. 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 (e.g. AT+MYCMD), the matching list element will be set to NULL (e.g. **ADL_GET_PARAM(para,0)** will return NULL in this case).

StrLength:

Incoming command string buffer length.

NI:

This parameter is to hold the Notification Identifier provided by the command handler.

Contxt:

A context holding information gathered at the time the command is subscribed (if provided).

StrData[1]:

Incoming command string buffer 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").

3.3.4.3. The adl_ atCmdSubscriptionPort_e Type

Basic required subscription port affected.

Code

```
typedef enum
{
        ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT,
        ADL_CMD_SUBSCRIPTION_ALL_PORTS
} adl_atCmdSubscriptionPort_e;
```

Description

ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT:	The subscription is only concerning command received on the external port.
ADL_CMD_SUBSCRIPTION_ALL_PORTS:	The subscription is concerning command received on all ports.

Note: *In this current release* ***ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT*** *is the only valid choice.*

3.3.4.4. ADL_GET_PARAM

Macro to get the requested parameter.

Code

```
#define ADL_GET_PARAM  ( _P_,
                         _i_ )((ascii*)wm_lstGetIitem(_P_->ParaList,_i_))
```

Parameters

_P_:

command handler parameter (refer to adl_atCmdPreParser_t structure about pointer to use).

_i_:

parameter index from 0 to NbPara (refer to adl_atCmdPreParser_t structure for more information about NbPara).

3.3.4.5. The adl_atCmdHandler_t Command Handler

Such a call-back function has to be supplied to ADL through the **adl_atCmdSubscribe** interface in order to process AT command subscribed.

Prototype

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

Parameters

Params:

Contains information about AT response (refer to adl_atCmdPreParser_t for more information).

Note: *The command handler has the responsability to send unsollicited/intermediate reponses and at least one terminal response.*

3.3.4.6. 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 exclusively by an external application, the callback function will be executed.

Prototype

```
s16 adl_atCmdSubscribe ( ascii *             Cmdstr,
                         adl_atCmdHandler_t  Cmdhdl,
                         u16                 Cmdopt );
```

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. (Refer to adl_atCmdHandler_t for more information about callback function).

Cmdopt:

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. Information about number of arguments is combined with a bitwise 'OR' : **`ADL_CMD_TYPE_PARA | 0xXY`** , where X which defines maximum argument number for incoming command and Y which defines minimum argument number for incoming command.
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 but without the ending character ";" which is parsed for concatenated commands mode. 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.
ADL_CMD_TYPE_ROOT_EXT	0x2000	All commands starting with the subscribed string are allowed even with the ending character ";" this means that such a command will not be usable in a concatenated AT commands string. 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. **Note:** In this current release **`ADL_CMD_TYPE_ROOT_EXT`** is behaving like **`ADL_CMD_TYPE_ROOT`**

Note: *If **`ADL_CMD_TYPE_ROOT_EXT`** is associated with others it has priority and therefore the command cannot be recognized as a concatenated one.*

Note: In this current release ***ADL_CMD_TYPE_ROOT_EXT*** *is behaving like* ***ADL_CMD_TYPE_ROOT.***

Returned values

- **OK** on success.
- **ERROR** if an error occurred.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.3.4.7. The adl_atCmdSubscribeExt 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 or on all ports (depending on the Cmdport parameter), the callback function will be executed.

Prototype

```
s16 adl_atCmdSubscribeExt ( ascii *                        Cmdstr,
                            adl_atCmdHandler_t             Cmdhdl,
                            u16                            Cmdopt,
                            void *                         Contxt,
                            adl_atCmdSubscriptionPort_e    Cmdport );
```

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. (Refer to adl_atCmdHandler_t for more information about callback function).

Cmdopt:

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. Information about number of arguments is combined with a bitwise 'OR' : ADL_CMD_TYPE_PARA | 0xXY , where X which defines maximum argument number for incoming command and Y which defines minimum argument number for incoming command.
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 but without the ending character ";" which is parsed for concatenated commands mode. 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.

Command type	Value	Meaning
ADL_CMD_TYPE_ROOT_EXT	0x2000	All commands starting with the subscribed string are allowed even with the ending character ";" this means that such a command will not be usable in a concatenated AT commands string. 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. Note: In this current release **ADL_CMD_TYPE_ROOT_EXT** is behaving like **ADL_CMD_TYPE_ROOT**

Note: *If **ADL_CMD_TYPE_ROOT_EXT** is associated with others it has priority and therefore the command cannot be recognized as a concatenated one.*

In this current release **ADL_CMD_TYPE_ROOT_EXT** is behaving like **ADL_CMD_TYPE_ROOT**

Contxt:

Context made to hold information gathered at the time the command is subscribed.

Cmdport:

Port on which the command is subscribed (type of to **adl_atCmdSubscriptionPort_e**).

ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT

ADL_CMD_SUBSCRIPTION_ALL_PORTS

Note: *In this current release **ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT** is the only valid choice*

Returned values

- **OK** on success.
- **ERROR** if an error occurred.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler.

3.3.4.8. 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** on success,
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler.
- **ERROR** otherwise.

3.3.4.9. The adl_atCmdSetQuietMode Function

This function allows to set Quiet mode. In this mode, terminal responses are not send. This function has the same behaviour as ATQ command behaviour.

Prototype

```
void adl_atCmdSetQuietMode ( bool  IsQuiet )
```

Parameters

IsQuiet:

Quiet mode setting:

- **TRUE:** Quiet mode is activated
- **FALSE:** Quiet mode is deactivated. Default value.

3.3.4.10. Example

This example demonstrates how to use the AT Command Subscription/Unsubscriptions service in a nominal case (error cases not handled) with a embedded module.

Complete examples using the AT Command service are also available on the SDK.

```
// ati callback function
    void ATI_Handler(adl_atCmdPreParser_t *paras)
    {
        // we send a terminal response
        adl_atSendStdResponsePort(ADL_AT_RSP, paras->Port, ADL_STR_OK);
    }

    // function 2
    void function2(adl_InitType_e adlInitType)
    {
        // We unsubscribe the command ;
        adl_atCmdUnSubscribe("ati",
                            (adl_atCmdHandler_t)ATI_Handler);
    }

    // function 1
    void function1(adl_InitType_e adlInitType)
    {
        // Subscribe to the 'ati' command.
        adl_atCmdSubscribe("ati",
                            (adl_atCmdHandler_t)ATI_Handler,
                            ADL_CMD_TYPE_ACT);
    }
```

3.3.5. Outgoing AT Commands

The following functions allow to send a command on the required port and allows the subscription to several responses and intermediate responses with one associated callback function, so that when any of the responses or intermediate responses we subscribe to will be received by the ADL parser, the callback function will be executed.

The defined operations are:

- **`adl_atCmdCreate`** function to send a command on the required port and allow the subscription to several responses and intermediate responses with one associated callback function, so that when any of the responses or intermediate responses we subscribe to will be received by the ADL parser, the callback function will be executed.
- **`adl_atCmdSend`** same function as **`adl_atCmdCreate`** without the rspflag argument and instead sending the command to the Open AT® internal port.
- **`adl_atCmdSendExt`** same function as **`adl_atCmdCreate`** without the rspflag argument and instead the port argument plus a Notification Identifier and a Context.
- **`adl_atCmdSendText`** function to allow 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`**/**`adl_atCmdSend`**/ **`adl_atCmdSendExt`** function call.

Note: *Now **`adl_atCmdSendExt`** (with a NI parameter different from 0) finds out if the command has been subscribed. If the command has been subscribed the handler is called otherwise the command is executed (as it is when called with **`adl_atCmdSend`** or **`adl_atCmdCreate`**). If the command has multiple subscription the last handler subscribed is called. In order for any other handler to be called the last handler has to resend the command with **`adl_atCmdSendExt`** API and the NI parameter provided so that the penultimate handler will be called and so on.*

Note: *For any multiply subscribed command sent by an external application on one of the embedded module's ports all handlers were called at the same time. Now there is a change of behaviour where only the last subscribed handler is called (by resending the command using **`adl_atCmdSendExt`** API and the provided NI the penultimate handler is called and so on ...).*

Note: *If any Inner AT Command (as decribed in section Inner AT Commands Configuration of ADL UGD) is subscribed its handler has to resend the command with **`adl_atCmdSendExt`** API and the NI parameter provided so that ADL internal handler is called. Otherwise as explained in section Inner AT Commands Configuration of ADL UGD it may affect ADL correct behaviour.*

Note: *If a command is only subscribed once. Sending this command will call the handler. If the handler resends the command with **`adl_atCmdSendExt`** API and the NI parameter provided the command will be sent for execution. Likewise if a command is multiply subscribed. Sending this command with **`adl_atCmdSendExt`** API and the NI parameter provided will call the last handler if at some point (after re-sending the command with **`adl_atCmdSendExt`** API and the NI parameter provided) the first handler is called re-sending the command with **`adl_atCmdSendExt`** API and the NI parameter provided will send the command for execution.*

Note: *If the required port is not opened, the functions return an error(ADL_RET_ERR_PARAM). In the USB case, the cable must be plugged and the enumeration with the host has to succeed before proceeding to one of these operations.*

3.3.5.1. Required Header File

The header file is:

`adl_CmdStackHandler.h`

3.3.5.2. The adl_atResponse_t Structure

This structure contains information about AT command response.

code

```
typedef struct
{
      adl_strID_e    RspID;       // RspID
      adl_port_e     Dest;        // Dest
      u16            StrLength;   // Response length
      void *         Contxt;      // Context
      bool           IsTerminal;  // Terminal response flag
      u8             NI;          //  Notification Identifier
      u8             Type;        //  Type of the response
      u8             Pad [1];     //  Reserved for future use
      ascii          StrData[1];  // Response address
} adl_atResponse_t;
```

Description

RspID:

Detected standard response ID if the received response is a standard one.

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:

Response string buffer length.

Contxt:

A context holding information gathered at the time the command is sent (if provided).

IsTerminal:

A boolean flag indicating if the received response is the terminal one (TRUE) or an intermediate one (FALSE).

NI:

This parameter is to hold the Notification Identifier provided by the command initiating the response.

Type:

Type of the response.

StrData[1]:

Response string buffer address.

3.3.5.3. The adl_atRspHandler_t

Such a call-back function has to be supplied to ADL through the **`adl_atCmdCreate`**/ **`adl_atCmdSend`**/**`adl_atCmdSendExt`** interface in order to process AT response subscribed.

Prototype

```
typedef bool(*) adl_atRspHandler_t ( adl_atResponse_t   *Params )
```

Parameters

Params:

Contains information about AT response (refer to adl_atResponse_t for more information).

Returned 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.

3.3.5.4. The ADL_NI_LAUNCH

ADL_NI_LAUNCH to enable searching handler process.

Code

```
#define ADL_NI_LAUNCH   0xFE
```

Description

`ADL_NI_LAUNCH:`

To enable searching handler process.

If ADL_NI_LAUNCH is provided in API, adl_atCmdSendExt searching handler process will be launched: If the command is subscribed, the handler will be called. Otherwise, the command will be executed.

3.3.5.5. The adl_atCmdCreate Function

Add command to the required port command stack, in order to be executed as soon as this port is ready.

Prototype

```
s8 adl_atCmdCreate ( ascii *                atstr,
                     u16                    rspflag,
                     adl_atRspHandler_t     rsphdl,
                                            … );
```

Parameters

atstr:

The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

rspflag:

This parameter is composed of the unsubscribed responses destination flag, and the port where to send the command. The flag and 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_port_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:

  The response handler of the callback function associated to the command.

  ...:

  A list of strings of the response to subscribed to. This list has to be terminated by NULL.

Returned values

- **`OK`** on success
- **`ADL_RET_ERR_PARAM`** if an error occurred
- **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interruption handler
- **`ADL_RET_ERR_UNKNOWN_HDL`** when the _port argument correspond to a port which is closed.

Note: *Arguments **`rsphdl`** and the list of subscribed responses can be set to NULL to only send the command.*

Note: *If the _port parameter is set to **`ADL_PORT_NONE`** the command will be sent on **`ADL_PORT_OPEN_AT_VIRTUAL_BASE`** port.*

Note: *ATQ commands should not be used with **`adl_atCmdCreate`** / **`adl_atCmdSend`** / **`adl_atCmdSendExt`** API but instead **`adl_atCmdSetQuietMode`** API is to be used.*

3.3.5.6. The adl_atCmdSend Function

Add command to the internal default port command stack, in order to be executed as soon as this port is ready.

Prototype

```
s8 adl_atCmdSend  (  ascii *                atstr,
                     adl_atRspHandler_t     rsphdl,
                                            … );
```

Parameters

atstr:

The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

rsphdl:

The response handler of the callback function associated to the command.

...:

A list of strings of the response to subscribed to. This list has to be terminated by NULL.

Returned values

- **`OK`** on success
- **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interruption handler
- **`ADL_RET_ERR_PARAM`** if an error occurred

Note: *Arguments rsphdl and the list of subscribed responses can be set to NULL to only send the command.*

3.3.5.7. The adl_atCmdSendExt Function

This function sends AT command with 2 added arguments compared to **`adl_atCmdCreate / adl_atCmdSend`**: a NI (Notification Identifier) and a Context.

Add command to the required port command stack, in order to be executed as soon as this port is ready.

Prototype

```
s8 adl_atCmdSendExt   (   ascii *              atstr,
                          adl_port_e           port,
                          u16                  NI,
                          void *               Contxt,
                          adl_atRspHandler_t   rsphdl,
                                               … );
```

Parameters

atstr:

The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

port:

The required port on which the command will be executed.

NI:

This parameter is to hold the Notification Identifier. The NI parameter can have the following values:

- 0 (default value): the command is directly sent for execution (as when using **`adl_atCmdCreate`** or **`adl_atCmdSend`**)
- **`ADL_NI_LAUNCH`**: the searching handler process is launched:
  - If the command is subscribed the handler will be called
  - Else the command will be executed
- any para->NI provided by the handler (if called inside a handler)

Contxt:

Context made to hold information gathered at the time the command was sent.

rsphdl:

The response handler of the callback function associated to the command (see *Note* below).

...:

A list of strings of the response to subscribed to. This list has to be terminated by NULL (see *Note* below).

Returned values

- **`OK`** on success
- **`ADL_RET_ERR_PARAM`** if an error occurred
- **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler

Note: *Arguments rsphdl and the list of subscribed responses can be set to NULL to only send the command.*

Note: *The command AT+CPIN=<pin_code> is automatically subscribed by the Open AT OS. So if the user wants to send the command AT+CPIN=<pin_code> through the OPEN AT application, API adl_atCmdSendExt() with a NI parameter needs to be used . This way the ADL internal handler would be called and the correct SIM state would be maintained by the Open AT OS.*

3.3.5.8. The adl_atCmdSendText Function

Sends text for a running text command.

Prototype

```
s8 adl_atCmdSendText (   adl_port_e   Port,
                         ascii *      Text );
```

Parameters

Port:

Port on which is 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
- `ADL_RET_ERR_PARAM` if an error occurred
- `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level Interruption handler.

3.3.5.9. Examples

This example demonstrates how to use the AT Command Sending service in a nominal case (error cases not handled) with a embedded module.

Complete examples using the AT Command service are also available on the SDK.

Example 1

```
// ati responses callback function
    s16 ATI_Response_Handler(adl_atResponse_t *paras)
    {
       TRACE((1, "Reponse handled"));
       TRACE((1, paras->StrData));
       return FALSE;
    }

    // function 1
    void function1(adl_InitType_e adlInitType)
    {
        // We send ati and subscribe to its responses
        adl_atCmdSend("ati",
                      (adl_atRspHandler_t)ATI_Response_Handler,
                      "*",
                      NULL);
    }
```

Example 2

```
// at+bbb responses handler function
    bool B_RspHandler ( adl_atResponse_t * paras )
    {
        TRACE (( 1, "In B_RspHandler - printing out response" ));
        // the return value is TRUE to print out responses
        return TRUE;
    }
    // at+aaa command handler function
    void A_CmdHandler(adl_atCmdPreParser_t * paras )
    {
        TRACE (( 1, "In A_CmdHandler - sending AT+BBB cmd" ));
        // sending at+bbb command with adl_atCmdSendExt and provided NI
        // at+bbb is subscribed so command handler B_CmdHandler is to be
   called
        adl_atCmdSendExt( "at+bbb", paras->Port, paras->NI, NULL,
      B_RspHandler, "*", NULL );
    }

    // ati responses handler function
    bool C_RspHandler ( adl_atResponse_t * paras )
    {
        TRACE (( 1, "In C_RspHandler - transferring response" ));
        // ati responses are handled and transfered to the previous
   responses handler subscribes with the same NI
        adl_atSendResponseSpe ( ADL_AT_PORT_TYPE (paras->Dest, paras->Type),
                paras->StrData, paras->NI );
        return FALSE;
    }
    // at+bbb command handler function
    void B_CmdHandler(adl_atCmdPreParser_t * paras )
    {
        TRACE (( 1, "In B_CmdHandler - sending ATI cmd" ));
        // sending ati command with adl_atCmdSendExt and provided NI
        // ati is not subscribed hence the AT command is sent for execution
        adl_atCmdSendExt( "ati", TRUE, paras->NI, NULL, C_RspHandler, "*",
          NULL );
    }

    void adl_main ( adl_InitType_e InitType )
    {
        TRACE (( 1, "Embedded Application : Main" ));

        // at+aaa is subscribed with A_CmdHandler command handler
        adl_atCmdSubscribe("AT+AAA",A_CmdHandler,ADL_CMD_TYPE_ACT);
        // at+bbb is subscribed with B_CmdHandler command handler
        adl_atCmdSubscribe("AT+BBB",B_CmdHandler,ADL_CMD_TYPE_ACT);
    }
```

Example 3

```
// ati responses handler function
    bool ATI_RspHandler2 ( adl_atResponse_t * paras )
    {
        TRACE (( 1, "In ATI_RspHandler2 - printing out response" ));
        // ati responses are handled
        // the return value is TRUE to print out responses
        return TRUE;
    }
    // ati command handler function
    void ATI_CmdHandler1(adl_atCmdPreParser_t * paras )
    {
        TRACE (( 1, "In ATI_CmdHandler1 - re-sending AT cmd" ));
        // This handler is the last subscribed so the first called
        // sending ati command with adl_atCmdSendExt() and provided NI
        // ati is again subscribed so next command handler ATI_CmdHandler2()
   is to be called
        adl_atCmdSendExt( paras->StrData, paras->Port, paras->NI, NULL,
          ATI_RspHandler2, "*", NULL );
    }
    // ati responses handler function
    bool ATI_RspHandler3 ( adl_atResponse_t * paras )
    {
        TRACE (( 1, "In ATI_RspHandler3 - transferring response" ));
        // ati responses are handled and transfered to the previous
   responses handler subscribes with the same NI
        adl_atSendResponseSpe ( ADL_AT_PORT_TYPE (paras->Dest, paras->Type),
                 paras->StrData, paras->NI );
        return FALSE;
    }
    // ati command handler function
    void ATI_CmdHandler2(adl_atCmdPreParser_t * paras )
    {
        TRACE (( 1, "In ATI_CmdHandler2 - sending AT cmd for execution (no
   more handlers)" ));
        // sending ati command with adl_atCmdSendExt() and provided NI
        // ati is not subscribed anymore (both subscribed handler have been
   called) hence the AT command is sent for execution
        adl_atCmdSendExt( paras->StrData, paras->Port, paras->NI, NULL,
   ATI_RspHandler3, "*", NULL );
    }

    void adl_main ( adl_InitType_e InitType )
    {
        TRACE (( 1, "Embedded Application : Main" ));

        // ati is subscribed twice
        // - first with ATI_CmdHandler2 command handler
        // - then  with ATI_CmdHandler1 command handler
        adl_atCmdSubscribe("ati",ATI_CmdHandler2,ADL_CMD_TYPE_ACT);
        adl_atCmdSubscribe("ati",ATI_CmdHandler1,ADL_CMD_TYPE_ACT);
    }
```

3.4. Timers

ADL supplies Timers Service interface to allow application tasks to require and handle timer related events.

The defined operations are:

- **subscription** functions (**adl_tmrSubscribe** & **adl_tmrSubscribeExt**) usable to require a timer event for the current task
- A **handler** call-back type (**adl_tmrHandler_t**) usable to receive timer related events
- An **unsubscription** function (**adl_tmrUnSubscribe**) usable to stop a currently running timer.

3.4.1. Required Header Files

The header file for the functions dealing with timers is:

adl_TimerHandler.h

3.4.2. The adl_tmr_t Structure

This structure is used to store timers related parameters. **adl_tmrSubscribe** and **adl_tmrSubscribeExt** return a pointer on this structure, which will be usable later to unsubscribe from the timer through **adl_tmrUnSubscribe**.

Code:

```
typedef struct
{
      u8                        TimerId;
      adl_tmrCyclicMode_e       bCyclic;
      adl_tmrType_e             TimerType;
      u32                       TimerValue;
      adl_tmrHandler_t          TimerHandler;
} adl_tmr_t;
```

Description

TimerId

0 based internal timer identifier. This identifier will be provided to **adl_tmrHandler_t handler** on each call.

bCyclic

Remembers the associated timer cyclic mode.

TimerType

Remembers the programmed timer granularity.

TimerValue

Remembers the programmed timer duration.

TimerHandler

Remembers the timer handler address, provided at subscription time.

3.4.3. Defines

3.4.3.1. ADL_TMR_100MS_MAX_VALUE

ADL_TMR_100MS_MAX_VALUE defines the maximal value that can be set for a timer with a granularity of 100 ms. Refer to the **TimerValue** parameter in adl_tmrSubscribe function and adl_tmrSubscribeExt function. The maximal period of the timer is about 7 days.

Code

```
#define ADL_TMR_100MS_MAX_VALUE 0x5E9000
```

Description

ADL_TMR_100MS_MAX_VALUE : Max value for 100ms-timer

3.4.3.2. ADL_TMR_MS_TO_TICK

Several conversion from timing unit to ticks.

Code

```
#define ADL_TMR_MS_TO_TICK(MsT) ((u32)(((MsT)*7)+64)>>7)
```

Description

ADL_TMR_MS_TO_TICK(MsT): Timer conversion from milliseconds to ticks

3.4.3.3. ADL_TMR_100MS_TO_TICK

Several conversion from timing unit to ticks.

Code

```
#define ADL_TMR_100MS_TO_TICK(MsT) ((u32)(((MsT)*693L)+64)>>7)
```

Description

ADL_TMR_100MS_TO_TICK(MsT): From 100 milliseconds to ticks

3.4.3.4. ADL_TMR_S_TO_TICK

Several conversion from timing unit to ticks.

Code

```
#define ADL_TMR_S_TO_TICK(SecT) ((u32)(((SecT)*6934L)+64)>>7)
```

Description

ADL_TMR_S_TO_TICK(SecT): From seconds to ticks

3.4.3.5. ADL_TMR_MN_TO_TICK

Several conversion from timing unit to ticks.

Code

```
#define ADL_TMR_MN_TO_TICK(MnT) ((u32)(((MnT)*416034L)+64)>>7)
```

Description

ADL_TMR_MN_TO_TICK(MnT): From minutes to ticks

3.4.4. The adl_tmrType_e

Allows to define the granularity (time unit) for the **`adl_tmrSubscribe`**, **`adl_tmrSubscribeExt`** & **`adl_tmrUnSubscribe`** functions.

Code

```
typedef enum
{
        ADL_TMR_TYPE_100MS,
        ADL_TMR_TYPE_TICK,
        ADL_TMR_TYPE_LAST
} adl_tmrType_e;
```

Description

`ADL_TMR_TYPE_100MS:`	100ms granularity timer.
`ADL_TMR_TYPE_TICK:`	18.5ms ticks granularity timer.
`ADL_TMR_TYPE_LAST:`	Reserved for internal use.

3.4.5. The adl_tmrCyclicMode_e

Allows to define the required cyclic option at timer subscription time.

Note: *When using the **`ADL_TMR_CYCLIC_OPT_ON_EXPIRATION`** option, there is no minimum time guaranteed between two timer events, since if the application is preempted for some time, timer events will continue to be generated even if the application is not notified.*

Note: *This is not the case with the **`ADL_TMR_CYCLIC_OPT_ON_RECEIVE`** option: since the timer is re-programmed only when the application is notified, the duration between two events is guaranteed to be at least equal to the timer period.*

Code

```
typedef enum
{
        ADL_TMR_CYCLIC_OPT_NONE,
        ADL_TMR_CYCLIC_OPT_ON_EXPIRATION,
        ADL_TMR_CYCLIC_OPT_ON_RECEIVE,
        ADL_TMR_CYCLIC_OPT_LAST
} adl_tmrCyclicMode_e;
```

Description

`ADL_TMR_CYCLIC_OPT_NONE:`	One shot timer: the timer will be automatically be unsubscribed as soon as the event is notified to the application.
`ADL_TMR_CYCLIC_OPT_ON_EXPIRATION:`	Cyclic timer, which will be re-programmed on expiration, just before the event is sent to the application.
`ADL_TMR_CYCLIC_OPT_ON_RECEIVE:`	Cyclic timer, which will be re-programmed on event reception, just before notifying the application's handler.
`ADL_TMR_CYCLIC_OPT_LAST:`	Reserved for internal use.

3.4.6. The adl_tmrHandler_t

Call-back function, provided in an **`adl_tmrSubscribe`** or **`adl_tmrSubscribeExt`** call, and notified each time the related timer occurs.

Prototype:

```
typedef void(*) adl_tmrHandler_t ( u8     ID,
                                   void * Context );
```

Parameters

ID

Timer internal identifier (readable from the **`adl_tmr_t`** pointer returned at subscription time).

Context

Pointer on the application context provided to **`adl_tmrSubscribeExt`** function. Will be set to NULL is the timer was programmed with **`adl_tmrSubscribe`** function.

Note: *Such a call-back function will always be called in the task context where the timer was programmed with **`adl_tmrSubscribe`** or **`adl_tmrSubscribeExt`**.*

Note: *Timer events should be delayed if the applicative task is pre-empted due to higher priority (applicative or firmware) tasks processing.*

3.4.7. 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, in the task context where the **`adl_tmrSubscribe`** function was called.

Prototype

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

Parameters

bCyclic:

This boolean flag indicates whether the timer is cyclic (**TRUE**) or not (**FALSE**). A cyclic timer is automatically restarted before calling the application event handler.

TimerValue:

The number of periods after which the timer expires (depends on TimerType parameter required time unit).

If an **`ADL_TMR_TYPE_100MS`** timer is subscribed, the maximal value of this parameter is **`ADL_TMR_100MS_MAX_VALUE`**.

TimerType:

Unit of the TimerValue parameter (uses the **`adl_tmrType_e`** type).

Timerhdl:

The callback function associated to the timer (using the **`adl_tmrHandler_t`** type).

Returned values

- A positive timer handle (an **adl_tmr_t** pointer) on success, usable to unsubscribe later from the timer service; a NULL or negative value (the timer is not started).
- On failure, a negative error value:
- NULL If TimerValue is 0 or too big, or if there is no additional timer resource available.
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

Note: *Since the embedded module 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. E.g., if a 20 * 100ms timer is required, the real time value will be 1998 ms (108 * 18.5ms).*

Note: *The maximal value of "TimerValue" parameter is 0x5E9000 when "**ADL_TMR_TYPE_100MS**" timer is subscribed.*

Note: *Timers started with this function are not strict. Please refer to **adl_tmrSubscribeExt** for more information about strict timers.*

Note: *A task can use up to 32 timers at the same time and all tasks can use about 40 timers at the same time. If no additional timer is available, returned value will be NULL.*

Note: *The embedded module time granularity is approximately 18.5 ms. The exact value is equal to the duration of 4 GSM frames, which is 24/1300s (18.461 ms).*

Note: *Any application that uses the Timer service in a periodic mode, should consider this exact tick duration. For example, if it calls **adl_tmrSubscribe(Ext)** with TimerType=ADL_TMR_TYPE_TICK and TimerValue=1, the elapsed time after 389190 timer expirations will be 7185s, which is 15s lower than the expected one (7200s).*

3.4.8. The adl_tmrSubscribeExt Function

This function starts a timer with an associated callback function. The callback function will be executed as soon as the timer expires, in the task context where the **adl_tmrSubscribe** function was called.

Prototype

```
adl_tmr_t *adl_tmrSubscribeExt (   adl_tmrCyclicMode_e    CyclicOpt,
                                   u32                    TimerValue,
                                   adl_tmrType_e          TimerType,
                                   adl_tmrHandler_t       Timerhdl,
                                   void *                 Context,
                                   bool                   Strict );
```

Parameters

CyclicOpt:

This option flag allows to set the required cyclic mode of the timer, using the **adl_tmrCyclicMode_e** type.

TimerValue:

The number of periods after which the timer expires (depends on TimerType parameter required time unit).

If an **ADL_TMR_TYPE_100MS** timer is subscribed, the maximal value of this parameter is **ADL_TMR_100MS_MAX_VALUE**.

TimerType:

Unit of the TimerValue parameter (uses the **adl_tmrType_e** type).

Timerhdl:

The callback function associated to the timer (using the **adl_tmrHandler_t** type).

Context:

Pointer on an application defined context, which will be provided to the handler when the timer event will occur. This parameter should be set to NULL if not used.

Strict:

Boolean flag, allowing to start a strict timer.

If set to FALSE, like **`adl_tmrSubscribe`**, the timer occurence will not lead the embedded module to wake up from SLEEP mode with GSM stack in idle. This means that the timer occurence will be delayed to the next embedded module regular wake up.

If set to TRUE, the timer is strict, and will awake the embedded module from the SLEEP mode with GSM stack in idle when it occurs.

Please note that out of the SLEEP mode with GSM stack in idle, this parameter is ignored.

Returned values

- A positive timer handle (an **`adl_tmr_t`** pointer) on success, usable to unsubscribe later from the timer service; on error, a NULL or negative value (the timer is not started).
- NULL If TimerValue is 0 or too big, or if there is no additional timer resource available.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

Note: *Since the embedded module 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. E.g., if a 20 * 100ms timer is required, the real time value will be 1998 ms (108 * 18.5ms).*

Note: *The maximal value of "TimerValue" parameter is 0x5E9000 when "**`ADL_TMR_TYPE_100MS`**" timer is subscribed.*

Note: *A task can use up to 32 timers at the same time and all tasks can use about 40 timers at the same time. If no additional timer is available, returned value will be NULL.*

Note: *The embedded module time granularity is approximately 18.5 ms. The exact value is equal to the duration of 4 GSM frames, which is 24/1300s (18.461 ms).*

Note: *Any application that uses the Timer service in a periodic mode, should consider this exact tick duration. For example, if it calls **`adl_tmrSubscribe(Ext)`** with TimerType=ADL_TMR_TYPE_TICK and TimerValue=1, the elapsed time after 389190 timer expirations will be 7185s, which is 15s lower than the expected one (7200s).*

3.4.9. 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 has not expired yet.

Prototype

```
s32 adl_tmrUnSubscribe(  adl_tmr_t*          t,
                         adl_tmrHandler_t    Timerhdl,
                         adl_tmrType_e       TimerType );
```

Parameters

t:

Timer handle to be unsubscribed, previously returned by **`adl_tmrSubscribe`** or **`adl_tmrSubscribeExt`**.

Timerhdl:

The callback function associated to the timer. This parameter is only used to verify the coherence of **`t`** parameter. It has to be the timer handler used in the subscription procedure.

For example:

```
PhoneTaskTimerPtr = adl_tmrSubscribe (TRUE, 10, OneSecond,
                    ADL_TMR_TYPE_100MS, PhoneTaskTimer);
// Later ......
adl_tmrUnSubscribe (PhoneTaskTimerPtr, PhoneTaskTimer,
                    ADL_TMR_TYPE_100MS) ;
```

TimerType:

Time unit of the returned value, using the **adl_tmrType_e** enumeration.

Returned values

- On success, a positive value indicating the remaining time of the timer before it expires (time unit depends on the TimerType parameter value);

  On failure, a negative error value:
- **ADL_RET_ERR_BAD_HDL** if the provided timer handle is unknown
- **ADL_RET_ERR_BAD_STATE** if the timer has already expired.
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

Note: *When the **ADL_RET_ERR_BAD_STATE** error code is returned, the timer is correctly unsubscribed. This error code occurs when the function is called after the timer has elapsed at hardware level, but before the timer handler is notified.*

Note: *Once a "one shot" (non cyclic) timer has expired and the handler is called, there is no need to unsubscribe from the Timer service: such a timer is automatically unsubscribed once elapsed.*

3.4.10. Example

The code sample below illustrates a nominal use case of the ADL Timers Service public interface (error cases are not handled).

```
adl_tmr_t *tt, *tt2;
u16 timeout_period = 5;     // in 100 ms steps;

void Timer_Handler( u8 Id, void * Context )
{
    // We do not unsubscribe to the timer because it has ‘naturally’ expired
    adl_atSendResponse(ADL_AT_RSP, "\r\Timer timed out\r\n");
}

void Timer_Handler2( u8 Id, void * Context )
{
    // Unsubscribe from the timer resource
    adl_tmrUnSubscribe ( tt2, Timer_Handler2 );
}

// main function
void adl_main ( adl_InitType_e adlInitType )
{
    // We set up a one-shot timer
    tt = adl_tmrSubscribe ( FALSE,
                            timeout_period,
                            ADL_TMR_TYPE_100MS,
                            Timer_Handler );

    // We set up a cyclic timer
    tt2 = adl_tmrSubscribeExt ( ADL_TMR_CYCLIC_OPT_ON_RECEIVE,
                                timeout_period,
                                ADL_TMR_TYPE_100MS,
                                Timer_Handler2,
                                NULL,
                      FALSE );
}
```

3.5. Memory Service

The ADL Memory Service allows the applications to handle dynamic memory buffers, and get information about the platform's RAM mapping.

The defined operations are:

- get & release functions **`adl_memGet`** & **`adl_memRelease`** usable to manage dynamic memory buffers
- An information function **`adl_memGetInfo`** usable to retrieve information about the platform's RAM mapping

3.5.1. Required Header File

The header file for the memory functions is:

`adl_memory.h`

3.5.2. Data Structures

3.5.2.1. The adl_memInfo_t Structure

This structure contains several fields containing information about the platform's RAM mapping.

Note: The RAM dedicated to the Open AT® application is divided in three areas (Call stack, Heap memory & Global variables). The **`adl_memGetInfo`** *function returns these area current sizes.*

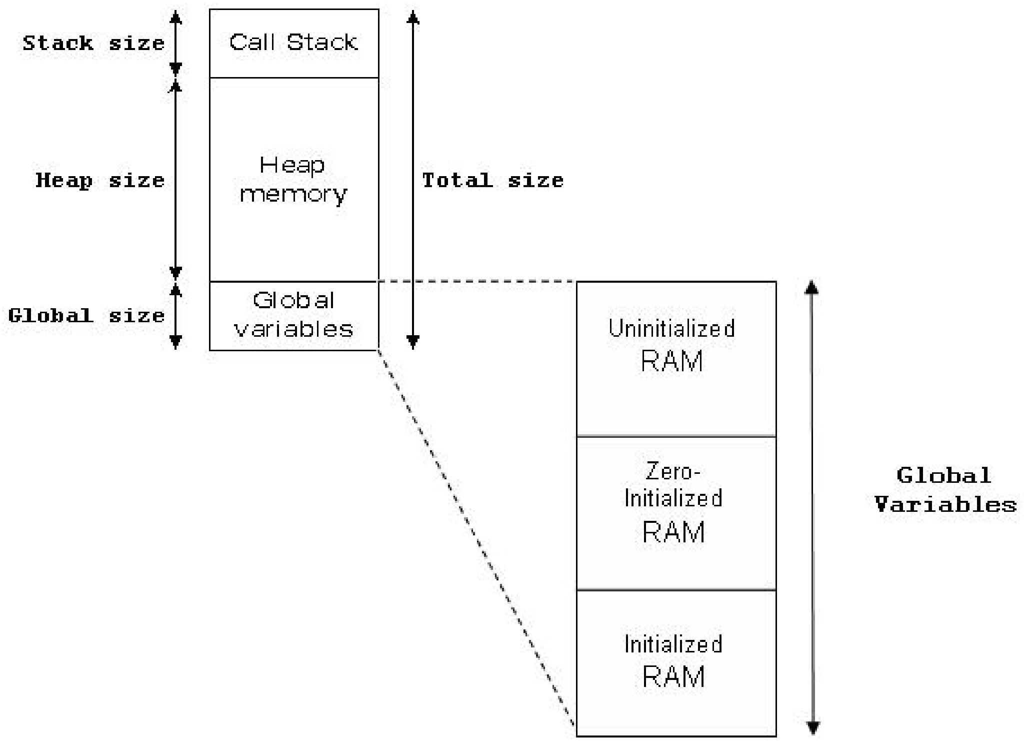


Figure 3. Open AT® RAM Mapping

Code

```
typedef  structure
{
        u32     TotalSize,
        u32     StackSize,
        u32     HeapSize,
        u32     GlobalSize
} adl_memInfo_t
```

Description

TotalSize

Total RAM size for the Open AT® application (in bytes). Please refer to the Memory Resources chapter for more information.

StackSize

Open AT® application call stacks area size (in bytes). This size is defined by the Open AT® application in the **`adl_InitTasks`** task table, and thanks to the **`adl_InitIRQLowLevelStackSize`** and **`adl_InitIRQHighLevelStackSize`** constants. (Please refer to the 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.*

3.5.3. Defines

3.5.3.1. The adl_memRelease

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

Parameters

_p_

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.

Note: *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 embedded module).*

Exceptions

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

3.5.3.2. The ADL_MEM_UNINIT

This macro is used to define a global variable in the uninitialized part of RAM. This part is not cleared after a hard or soft reset, only when power supply is OFF. So when an application restarts, global variable defined with this macro keep the last saved value before the last reset.

Code

```
#define ADL_MEM_UNINIT ( _X ) _X __attribute__((section("UNINIT")));
```

Parameters

_X

This parameters corresponds to global variable to define. The type and the name of the variable have to be defined. Refer to Example below to get more information

Note: *Rules on the syntax:*
- at the end of variable declaration,
- there is no semi-colonglobal variable cannot be initialized with a value when it is declared

Warning: *It is not functional in RTE mode; the global variable will be intialized to 0 at starting.*

Example

```
// Global variable definition
  ADL_MEM_UNINIT( u32 MyGlobal )

  void adl_main ( adl_InitType_e InitType )
  {
      ...
      MyGlobal = 500;
      ...
  }
```

3.5.4. 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:

Please refer to the 3.5.2.1 `adl_memInfo_t` structure.

- TotalSize

  Total RAM size for the Open AT® application (in bytes).
  Please refer to the Memory 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 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. Please refer to the Figure 3.

Returned values

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

3.5.5. 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.

Exceptions

- **`ADL_ERR_MEM_GET`** 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 section Error Management for more information.
- **`RTK exception 166`**

  A buffer header or footer data is corrupted: a write overflow has occurred on this block.

Note: 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 embedded module).

3.5.6. The adl_memRelease Function

Internal memory release function, which should not be called directly. The **`adl_memRelease`** macro has to be used in order to release memory buffer.

Prototype

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

Parameters

ptr:

A pointer on the allocated memory buffer.

Returned values

- Please refer to the section adl_memRelease macro definition.

3.5.7. Heap Memory Block Status

A list of the currently reserved heap memory blocks can be displayed at any time using the Developer Studio Heap Status view. Please refer to Developer Studio online help 2 for more information.

3.5.8. Example

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

```
// 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 );
}
```

3.6. ADL Registry Service

The ADL Registry Service allows to give to Open AT® applications an access to the platform registry, used to store generic information about the software & hardware capabilities or configuration.

The defined operations are:

- An **`adl_regGetWCPUType`** function to retrieve information from the registry about current embedded module identifier (deprecated function)
- An **`adl_regGetWCPUTypeExt`** function to retrieve from the registry the current embedded module identifier.
- An **`adl_regGetHWInteger`** function to retrieve integer value of a registry entry
- An **`adl_regGetHWData`** function to retrieve the data value of a registry entry
- An **`adl_regGetHWDataChunk`** function to retrieve the data value of a registry entry

3.6.1. Required Header File

The header file is:

```
adl_reg.h
```

3.6.2. The adl_regGetWCPUTypeExt Function

This function allows the application to retrieve the current embedded module identifier

Prototype

```
s32 adl_regGetWCPUTypeExt  ( ascii *     CPUType );
```

Parameters

CPUType:

String buffer where the embedded module type identifier has to be copied.

Can be set to NULL in order just to retrieve the required string buffer size.

Returned values

- Positive number of copied characters to the supplied string buffer (including terminal 0).

3.6.3. The adl_regGetHWInteger Function

This function allows the application to retrieve the integer value of a registry entry.

Prototype

```
s32 adl_regGetHWInteger  (  ascii *   Label,
                            s32 *     Value );
```

Parameters

Label

Label of the entry in the registry.

Value

Integer buffer where the value of the registry label has to be copied.

Returned values

- A `OK` on success.
- A negative error value otherwise:
  - `ADL_RET_UNKNOWN_HDL` if the registry Label is not found.
  - `ADL_RET_BAD_HDL` if the registry type required is not good.
  - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value

3.6.4. The adl_regGetHWData Function

This function allows the application to retrieve the data value of a registry entry.

Prototype

```
s32 adl_regGetHWData (  ascii *      Label,
                        void *       Data);
```

Parameters

Label

Label of the entry in the registry.

Data

Data buffer where the information of the registry label has to be copied,

This is an optional parameter and must be set to 0 if not used.

Returned values

- The size of the Data information on success.
- A negative error value otherwise:
  - `ADL_RET_UNKNOWN_HDL` if the registry Label is not found.
  - `ADL_RET_BAD_HDL` if the registry type required is not good.
  - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value.

3.6.5. The adl_regGetHWDataChunk Function

This function allows the application to retrieve the data value of a registry entry.

Prototype

```
s32 adl_regGetHWDataChunk   (  ascii *   Label,
                               void *    Data,
                               u32       BeginOffset,
                               u32       ByteCount  );
```

Parameters

Label

Label of the entry in the registry.

Data

Data buffer where the information of the registry label has to be copied.

This is an optional parameter and must be set to 0 if not used.

BeginOffset

Offset within the data value, this is an optional parameter must be set to 0 if not used

ByteCount

Number of bytes to get, this is an optional parameter must be set to 0 if not used. if it set to 0, all data from offset to the end of entry are copied.

Returned values

- The size of the Data information on success.
- A negative error value otherwise:
  - **ADL_RET_UNKNOWN_HDL** if the registry Label is not found.
  - **ADL_RET_BAD_HDL** if the registry type required is not good.
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.

Note: *If BeginOffset and/or ByteCount is not 0 and Data is 0 the size of the Data information returned will not take into account the BeginOffset and/or ByteCount parameter.*

3.6.6. Example

```
// Retrieve embedded module identifier
  void * function_1()
  {
      // Retrieve required size for embedded module identifier
      u32 NameSize = adl_regGetWCPUType ( NULL );

      // Allows enough memory
      ascii * Name = adl_memGet ( NameSize );

      // Retrieve embedded module type
      adl_regGetWCPUType ( Name );

      // Check current embedded module type
      if ( !wm_strcmp ( Name, "WMP100" ) )
      {
        // WMP100 embedded module
      }
      else if ( !wm_strcmp ( Name, "Q2686" ) )
      {
        // Q2686 embedded module
      }
      else if ( !wm_strcmp ( Name, "Q2687" ) )
      {
        // Q2687 embedded module
      }
  }

  // Retrieve hardware integer information
  void * function_2()
  {
      u32 Hardware_info;

      // Retrieve the integer information
      adl_regGetHWInteger ( "Hardware_info_label", &Hardware_info );
      ...
  }
```

```
// Retrieve hardware data information
void * function_3()
{
    // Retrieve required size for hardware data information
    u32 Hardware_info_size = adl_regGetHWData ( "Hardware_info_label",
                                              NULL );

    // Allows enough memory
    adl_HardwareInfoExample_t * Hardware_info_data = adl_memGet
                                                ( Hardware_info_size );

    // Retrieve the adl_HardwareInfoExample_t information
    adl_regGetHWData ( "Hardware_info_label", Hardware_info_data );
    ...
}

// Retrieve hardware data information
void * function_4()
{
    // Allows enough memory for a part of hardware data information
    ascii * Hardware_info_data_chunk = adl_memGet ( 10 );

    // Retrieve the adl_HardwareInfoExample_t information
    adl_regGetHWDataChunk ( "Hardware_info_label",
                            Hardware_info_data_chunk , 5 , 10 );
    ...
}
```

3.7. Debug Traces

This service allows to display debug trace strings on Developer Studio. The different ways to embed these trace strings in an Open AT® application depends on the selected configuration in the used Developer Studio (see below).

For more information on the configurations of Developer Studio, please refer to Developer Studio online help 2.

The defined operations are:

- Trace function & macros (**`adl_trcPrint`**, **`TRACE`** & **`FULL_TRACE`**) to print the required trace string
- Dump function & macros (**`adl_trcDump`**, **`DUMP`** & **`FULL_DUMP`**) to dump the required buffer content

3.7.1. Required Header File

The header file for the flash functions is:

`adl_traces.h`

3.7.2. Build Configuration Macros

According to the chosen build configuration in Developer Studio, following macros will be defined or not, allowing the user to embed none, part or the entire debug traces information in its final application.

3.7.2.1. Debug Configuration

When the Debug configuration is selected in Developer Studio, the **__DEBUG_APP__** compilation flag is defined, and also the **`TRACE & DUMP`** macros.

Traces & dumps declared with these macros will be embedded at compilation time.

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

3.7.2.2. Full Debug Configuration

When the Full Debug configuration is selected in Developer Studio, both the **__DEBUG_APP__** and **__DEBUG_FULL__** compilation flags are defined, and also the **`TRACE, FULL_TRACE, DUMP`** & **`FULL_DUMP`** macros.

Traces & dumps declared with these macros will be embedded at compilation time.

3.7.2.3. Release Configuration

When the Release configuration is selected in Developer Studio, neither the **__DEBUG_APP__** nor **__DEBUG_FULL__** compilation flags are defined.

The **`TRACE, FULL_TRACE, DUMP`** 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 Developer Studio at runtime).

3.7.2.4. Defines

3.7.2.4.1. TRACE

This macro is a shortcut to the **`adl_trcPrint`** function. Traces declared with this macro are only embedded in the application if it is compiled with in the Debug or Full Debug configuration, but not in the Release configuration.

```
#define TRACE  ( _X_    )
```

3.7.2.4.2. DUMP

This macro is a shortcut to the **`adl_trcDump`** function. Dumps declared with this macro are only embedded in the application if it is compiled with in the Debug or Full Debug configuration, but not in the Release configuration.

```
#define DUMP   (   _lvl_,
                   _P_,
                   _L_  )
```

3.7.2.4.3. FULL TRACE

This macro is a shortcut to the adl_trcPrint function. Traces declared with this macro are only embedded in the application if it is compiled with in Full Debug configuration, but not in the Debug or Release configuration.

```
#define FULL_TRACE   ( _X_  )
```

3.7.2.4.4. FULL DUMP:

This macro is a shortcut to the **`adl_trcDump`** function. Dumps declared with this macro are only embedded in the application if it is compiled with in Full Debug configuration, but not in the Debug or Release configuration.

```
#define FULL_DUMP (   _lvl_,
                      _P_,
                      _L_  )
```

3.7.3. The adl_trcPrint Function

This function displays the required debug trace on the provided trace level. The trace will be displayed in Developer Studio, according to the current context:

- for tasks: on the trace element name defined in the tasks declaration table (cf. Application Initialization service)

- for Low Level Interrupt handlers: on the "LLH" trace element
- for High Level Interrupt handlers: on the "HLH" trace element

In addition to the trace information, a embedded module local timestamp is also displayed in the tool.

Example1:

```
u8 I = 123;
TRACE (( 1, "Value of I: %d", I ));
```

At runtime, this will display the following string on the CUS4 level 1 on Developer Studio:

```
Value of I: 123
```

Prototype

```
s8 adl_trcPrint ( u8            Level,
                  const ascii*  strFormat,
                                …  );
```

Parameters

Level:

Trace level on which the information has to be sent. Valid range is **1 - 32**.

strFormat:

String to be displayed, using a standard C "sprintf" format.

...:

Additional arguments to be dynamically inserted in the provided constant string.

Note: *Direct use of the **`adl_trcPrint`** function is not recommended. The **`TRACE`** & **`FULL_TRACE`** macros should be used instead, to take benefit of the build configurations features.*

Note: *'%s' character, normally used to insert strings, is not supported by the trace function.*

Note: *The trace display should be limited to 255 bytes. If the trace string is longer, it will be truncated.*

Note: *ADL trace function only supports up to 6 parameters; additional parameters are ignored.*

3.7.4. The adl_trcDump Function

This function dumps the required buffer content on the provided trace level. The dump will be displayed in Developer Studio, according to the current context:

- for tasks: on the trace element name defined in the tasks declaration table (cf. Application Initialization service)
- for Low Level Interrupt handlers: on the "LLH" trace element
- for High Level Interrupt handlers: on the "HLH" trace element

In addition to the trace information, a embedded module local timestamp is also displayed in the tool.

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 "..." characters 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 level 1 in Developer Studio:

```
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 level 1 in Developer Studio:

```
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
```

Prototype

```
void adl_trcDump (    u8         Level,
                      u8 *       DumpBuffer,
                      u16        DumpLength );
```

Parameters

Level:

Trace level on which the information has to be sent. Valid range is **1 - 32**.

DumpBuffer:

Buffer address to be dumped.

DumpLength:

Number of bytes to be displayed at required address.

Note: *Direct use of the* ***adl_trcDump*** *function is not recommended. The* ***DUMP*** *&* ***FULL_DUMP*** *macros should be used instead, to take benefit of the build configurations features.*

3.7.5. Example

The code sample below illustrates a nominal use case of the ADL Debug Traces service public interface (error cases are not handled).

```
u8 MyInt = 12;
ascii * MyString = "hello";

// Print a debug trace for current context on level 1
TRACE (( 1, "My Sample Trace: %d", MyInt ));

// Dump a buffer content for current context on level 2
DUMP ( 2, MyString, strlen ( MyString ) );
```

3.8. Flash

3.8.1. Required Header File

The header file for the flash functions is:

`adl_flash.h`

3.8.2. 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.

Note: *The default number of ID's is 2560 for 32Mb flash and 5120 for 64Mb flash.*

Note: *The maximum number of flash objects that can exist at any given time is 7936. Using WPK along with DWLWIN, the user can change the default value in the range 2560 to 7936.*

3.8.2.1. 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.

Caution: *The Garbage Collector process is a time consuming operation. Performing numerous flash write operations in the same event handler increases the probability of Garbage Collector occurence, and should lead to a watchdog reset of the embedded module. It is not recommended to perform too many flash write operations in the same event handler. If numerous operations are required, it is advised to regularly "give back the hand" to the Firmware (by introducing timers) in the write loop, in order to avoid the Watchdog reset to occur.*

3.8.2.2. 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 embedded module's flash memory. The two storage places (embedded module and PC one) may be synchronized using the RTE Monitor interface (cf. Developer Studio (http://www.sierrawireless.com/developer_studio) online help for more information).

3.8.3. 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 Interrupt handler (the function is forbidden in this context).

Note: *Only one subscription is necessary. It is not necessary to subscribe to the same handle at each application start.*

Note: *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.*

3.8.4. 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
- **`0K`** 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 Interrupt handler (the function is forbidden in this context).

3.8.5. 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.

Caution: *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 Interrupt handler (the function is forbidden in this context).

3.8.6. The adl_flhWrite Function

This function writes the flash object from the given Handle at the given ID, for the length provided with the buffer 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 buffer 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 Interrupt handler (the function is forbidden in this context).

3.8.7. 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 buffer.

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 buffer 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 Interrupt handler (the function is forbidden in this context).

3.8.8. 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.

3.8.9. The adl_flhGetIDCount Function

This function returns the ID count for the provided handle.

Prototype

```
s32 adl_flhGetIDCount (  ascii*    Handle );
```

Parameters

Handle:

The Handle of the subscribed set of objects. If set to NULL, an error is returned.

Returned values

- On success:
  - ID count allocated on the provided handle if any;
  - an error is returned 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 Interrupt handler (the function is forbidden in this context).

3.8.10. 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 Interrupt handler (the function is forbidden in this context).

3.9. 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.

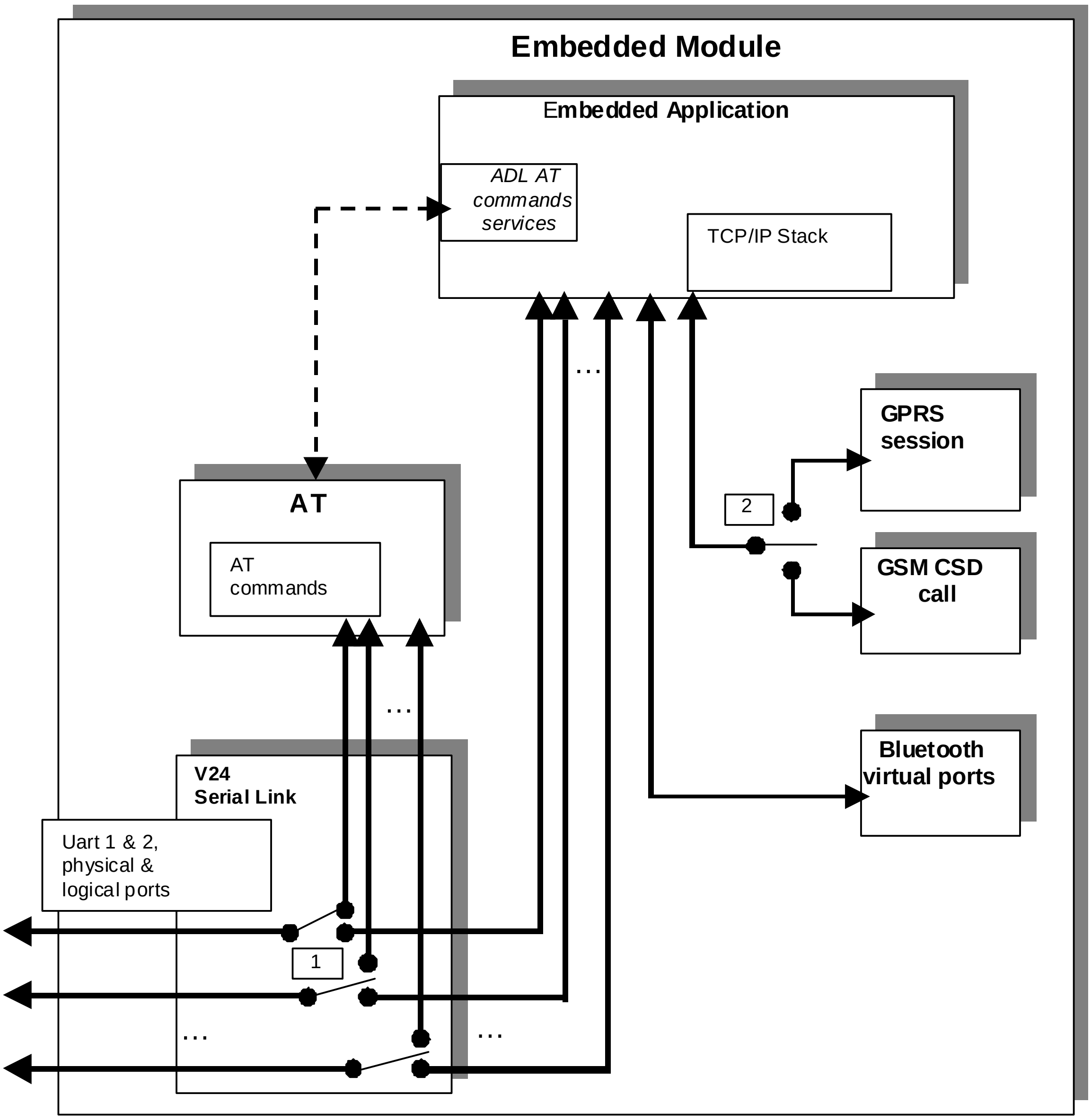


Figure 4. Flow Control Manager Representation

By default (ie. without any Open AT® application, or if the application does not use the FCM service), all the embedded module's ports are processed by the Sierra Wireless Firmware. 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;

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 embedded module'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;

The "1" switch 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.

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

3.9.1. Required Header File

The header file for FCM functions is:

```
adl_fcm.h
```

3.9.2. 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

Note: *All ports should be available for the FCM service, except:*

Note: *The Open AT® virtual one, which can only be used for AT commands,*

Note: *If the port is already used to handle a feature required by an external application through the AT commands (a CSD/GPRS data session is already running)*

3.9.3. 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_OPENED` 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 and receive data on/from the flow, but will know some limitations:

- For serial-line flows (UART physical and 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.

Caution: *For serial-link related flows (UART physical and 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 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 it 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 these 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 subscribed data handlers (master and slaves) 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 next time the data handler will return TRUE. The **adl_fcmReleaseCredits** should also be used to release credits outside 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;.

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 Interrupt handler (the function is forbidden in this context).

Note: *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.*

Note: *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. Application can emulate the sequence "1s delay; +++; 1s delay" behaviour with* ***adl_fcmSwitchV24State*** *API and* ***ADL_FCM_EVENT_V24_STATE_OFFLINE*** *parameter.*

Then the behaviour 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 embedded module side (with the adl_callHangUp function), the corresponding GSM or GPRS flow have to be unsubscribed. These flows will have to be subscribed again before starting up a new GSM data call, or a new GPRS session.
- For serial link flows, 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 to be first 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.
- To perform a GSM data call, the GSM flow should be open first. Only when the flow opened event (**`ADL_FCM_EVENT_FLOW_OPENED`**) is received, then a data call can be done or answered.

3.9.4. 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 interrupt handler (the function is forbidden in this context).

3.9.5. 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 higher 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 Interrupt handler (the function is forbidden in this context).

3.9.6. 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_AT`**, equivalent to "offline" modem state, DCD/DSR off.

- **`ADL_FCM_V24_STATE_DATA, ADL_FCM_V24_STATE_DATA`**, equivalent to "online connected" modem state, DCD/DSR on..
- **`ADL_FCM_V24_STATE_OFFLINE,`** equivalent to "offline connected" modem state, DCD on, DSR off.

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 Interrupt handler (the function is forbidden in this context).

3.9.7. 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 ;

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 Interrupt 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.

3.9.8. 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 ;.

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 Interrupt 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).

3.9.9. 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.

3.10. GPIO Service

ADL provides a GPIO service to handle GPIO operations.

The defined operations are:

- A **`adl_ioGetCapabilitiesList`** function to retrieve a list of GPIO capablities informations.
- A **`adl_ioSubscribe`** function to set the reserved GPIO parameters
- A **`adl_ioUnsubscribe`** function to un-subscribes from a previously allocated GPIO handle
- A **`adl_ioEventSubscribe`** function to provide ADL with a call-back for GPIO related events
- A **`adl_ioEventUnsubscribe`** function to unsubscribe from the GPIO events notification
- A **`adl_ioSetDirection`** function to allow the direction of one or more previously allocated GPIO to be modified
- A **`adl_ioRead`** function to allow several GPIOs to be read from a previously allocated handle
- A **`adl_ioReadSingle`** function to allow one GPIO to be read from a previously allocated handle
- A **`adl_ioWrite`** function to write on several GPIOs from a previously allocated handle
- A **`adl_ioWriteSingle`** function to allow one GPIO to be written from a previously allocated handle

3.10.1. Required Header File

The header file for the GPIO functions is:

```
adl_gpio.h
```

3.10.2. GPIO Types

3.10.2.1. The adl_ioCap_t structure

This structure gives information about io capabilities.

```
typedef struct
{
        u32  NbGpio;    // The number of GPIO managed by ADL.
        u32  NbGpo;     // The number of GPO managed by ADL.
        u32  NbGpi;     // The number of GPI managed by ADL.
} adl_ioCap_t;
```

3.10.2.2. The adl_ioDefs_t type

This type defines the GPIO label.

This is a bit field:

- b0-b15 are use to identify the io
  - see section adl_ioLabel_etype
- b16-b31 usage depends of the command
  - see section adl_ioLevel_etype
  - see section adl_ioDir_etype
  - see section adl_ioStatus_etype
  - see section adl_ioCap_etype
  - see section adl_ioError_etype

3.10.2.3. The adl_ioLabel_e type

This type lists the label field definition (b0-b15 of **adl_ioDefs_t**). Each IO is identified by a number and a type. Please see also section adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_NUM_MSK        = (0xFFF),
        ADL_IO_TYPE_POS       = 12,
        ADL_IO_TYPE_MSK       = (3UL<<ADL_IO_TYPE_POS),
        ADL_IO_GPI            = (1UL<<ADL_IO_TYPE_POS),
        ADL_IO_GPO            = (2UL<<ADL_IO_TYPE_POS),
        ADL_IO_GPIO           = (3UL<<ADL_IO_TYPE_POS),
        _IO_LABEL_MSK         = ADL_IO_NUM_MSK | ADL_IO_TYPE_MSK
} adl_ioLabel_e
```

Description

`ADL_IO_NUM_MSK`	**Number field (b0-b11; 0->4095)**
`ADL_IO_TYPE_MSK`	Type field (b12-b13):
`ADL_IO_GPI`	- To identify a GPI
`ADL_IO_GPO`	- To identify a GPO
`ADL_IO_GPIO`	- To identify a GPIO (GPO + GPI)
`ADL_IO_LABEL_MSK`	Mask including **`ADL_IO_NUM_MSK`** and **`ADL_IO_TYPE_MSK`**

Note: *b14-b15 are reserved.*

Note: *This type is only used to identify an IO pin of the embedded module, and not to configure the current direction. E.g. to identify the GPIO 12 pin of a embedded module, the "ADL_IO_GPIO | 12" statement shall be used. In order to configure or get the current direction of a given pin, the **`adl_ioDir_e`** type must be used (please refer to adl_ioDir_etype for more information). Please also note that valid labels are described in the related Embedded module Product Technical Specification, and are also retrievable from the GPIO service capabilities.*

3.10.2.4. The adl_ioLevel_e type

This type lists the level field definition (b16 of **`adl_ioDefs_t`**). Please see also adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_LEV_POS        = 16,
        ADL_IO_LEV_MSK        = (1UL<<ADL_IO_LEV_POS),
        ADL_IO_LEV_HIGH       = (1UL<<ADL_IO_LEV_POS),
        ADL_IO_LEV_LOW        = (0UL<<ADL_IO_LEV_POS)
} adl_ioLabel_e
```

Description

`ADL_IO_LEV_MSK`	Level field: the Level of GPIO
`ADL_IO_LEV_HIGH`	- High Level
`ADL_IO_LEV_LOW`	- Low Level

3.10.2.5. The adl_ioDir_e type

This type lists the direction field definition (b17-b18 of **`adl_ioDefs_t`**). Please see also adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_DIR_POS     = 17,
        ADL_IO_DIR_MSK     = (3UL<<ADL_IO_DIR_POS),
        ADL_IO_DIR_OUT     = (0UL<<ADL_IO_DIR_POS),
        ADL_IO_DIR_IN      = (1UL<<ADL_IO_DIR_POS),
        ADL_IO_DIR_TRI     = (2UL<<ADL_IO_DIR_POS)
} adl_ioDir_e type
```

Description

ADL_IO_DIR_MSK	- Dir field: The direction of GPIO
ADL_IO_DIR_OUT	- Set as Output
ADL_IO_DIR_IN	- Set as Input
ADL_IO_DIR_TRI	- Set as a Tristate

Note: *This type is only used to identify the current direction of a given pin. Pin labels are identified by the* ***adl_ioLabel_e*** *type (Please refer to [adl_ioLabel_etype](#) for more information).*

3.10.2.6. The adl_ioError_e type

This type lists the error field definition (b28-b31 of **adl_ioDefs_t**). Please see also adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_ERR_POS         = 28,
        ADL_IO_ERR_MSK         = (7UL<<ADL_IO_ERR_POS),
        ADL_IO_ERR             = (0UL<<ADL_IO_ERR_POS),
        ADL_IO_ERR_UNKWN       = (1UL<<ADL_IO_ERR_POS),
        ADL_IO_ERR_USED        = (2UL<<ADL_IO_ERR_POS),
        ADL_IO_ERR_BADDIR      = (3UL<<ADL_IO_ERR_POS),
        ADL_IO_ERR_NIH         = (4UL<<ADL_IO_ERR_POS),
        ADL_IO_GERR_POS        = 31,
        ADL_IO_GERR_MSK        = (1UL<<ADL_IO_GERR_POS),
        ADL_IO_GNOERR          = (0UL<<ADL_IO_GERR_POS),
        ADL_IO_GERR            = (1UL<<ADL_IO_GERR_POS)
 } ioError_e type
```

Description

ADL_IO_ERR_MSK	**Error cause (b28-b30):**
ADL_IO_ERR	- Unidentified error
ADL_IO_ERR_UNKWN	- Unknown GPIO
ADL_IO_ERR_USED	- Already used
ADL_IO_ERR_BADDIR	- Bad direction
ADL_IO_ERR_NIH	- GPIO is not in the handle
ADL_IO_GERR_MSK	**General error field (b31):**
ADL_IO_GNOERR	- No Error (b28-30 are unsignificant)
ADL_IO_GERR	- Error during the treatment (see b28-b30 for the cause)

Example

```
#define NUM_GPIO_OUT 2
adl_ioDefs_t Gpio_Out_Config[NUM_GPIO_OUT] = {

      (ADL_IO_GPO | 20 | ADL_IO_DIR_OUT | ADL_IO_LEV_LOW ) ,
      (ADL_IO_GPIO | 23 | ADL_IO_DIR_OUT | ADL_IO_LEV_LOW) };

        s32 myGpioOut_Handle;

        void adl_main ( adl_InitType_e InitType )
    {
               TRACE (( 1, "Embedded Application : Main" ));

   //Subscribe to outputs
    myGpioOut_Handle = adl_ioSubscribe(NUM_GPIO_OUT,Gpio_Out_Config,0,0,0);
    TRACE (( 1, "handler returns %d", myGpioOut_Handle ));

        switch(myGpioOut_Handle)
    {
               case ADL_RET_ERR_PARAM:
               TRACE (( 1, "if a parameter has an incorrect value" ));
               break;
               case ADL_RET_ERR_DONE:
               TRACE (( 1, "refers to the field 3.10.2.6 adl_ioError_e" ));
               TRACE ((1,"is there any error %x",Gpio_Out_Config[0] &
ADL_IO_GERR_MSK )); // if the result is 80000000, this means that there is an
error. actually the b31 indicates if b28-b31 are significant or not.

               TRACE ((1," the return value of adl_io_defs_t is  %x",
Gpio_Out_Config[0] & ADL_IO_ERR_MSK )); // then to get the error result, use
the mask ADL_IO_ERR_MSK . in Our case, as GPO20 is not recognized, then the
returned error will be 10000000 which corresponds to adl_io_err_unkwm (unkown
GPIO).

                    break;
                case ADL_RET_ERR_NO_MORE_TIMERS:
                TRACE (( 1, "there is no timer available to start" ));
                    break;
                case ADL_RET_ERR_NO_MORE_HANDLES:
                TRACE (( 1, "no more GPIO handles are available" ));
                    break;
                case ADL_RET_ERR_SERVICE_LOCKED:
               TRACE (( 1, "the function was called from a low level
               Interrupt handler" ));
                    break;
                           }
            TRACE((1,"myGpioOut_Handle = %d",myGpioOut_Handle));
}
```

3.10.2.7. The adl_ioCap_e type

This type lists the capabilities field definition (b21-b22 of **adl_ioDefs_t**). It is only an output. Please see also adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_CAP_POS      = 21,
        ADL_IO_CAP_MSK      = (3UL<<ADL_IO_CAP_POS),
        ADL_IO_CAP_OR       = (1UL<<ADL_IO_CAP_POS),
        ADL_IO_CAP_IW       = (2UL<<ADL_IO_CAP_POS)
 } adl_ioCap_e type
```

Description

ADL_IO_CAP_MSK	**Capabilities field: Specials capabilities**
ADL_IO_CAP_OR	- Output is readable
ADL_IO_CAP_IW	- Input is writable

3.10.2.8. The adl_ioStatus_e type

This type lists the status field definition (b19-b20 of **adl_ioDefs_t**). it is only an output. Please see also adl_ioDefs_t for the other fields.

Code

```
type def enum
{
        ADL_IO_STATUS_POS      = 19,
        ADL_IO_STATUS_MSK      = (3UL<<ADL_IO_STATUS_POS),
        ADL_IO_STATUS_USED     = (1UL<<ADL_IO_STATUS_POS),
        ADL_IO_STATUS_FREE     = (0UL<<ADL_IO_STATUS_POS)
} adl_ ioStatus_e type
```

Description

ADL_IO_STATUS_MSK	**Status field: to get the status of the fields**
ADL_IO_STATUS_USED	- The IO is used by task
ADL_IO_STATUS_FREE	- The IO is available

3.10.2.9. The adl_ioEvent_e type

This type describes the GPIOs events received.

Code

```
type def enum
{
        ADL_IO_EVENT_INPUT_CHANGED   = 2
} adl_ ioEvent_e type
```

Description

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.

3.10.3. The adl_ioGetCapabilitiesList Function

This function returns the embedded module GPIO capabilities list. For each hardware available GPIO, the embedded module shall add an item in the GPIO capabilities list. A GPIO is hardware available when it is not used by any feature.

Caution: *The returned GpioTab array must be released by the customer application when the information is not useful any more.*

Prototype

```
s32 adl_ioGetCapabilitiesList   (   u32 *            GpioNb,
                                    adl_ioDefs_t **  GpioTab,
                                    adl_ioCap_t *    GpioTypeNb );
```

Parameters

GpioNb:

Number of GPIO treated, it is the size of **`GpioTab`** array.

GpioTab:

Returns a pointer to a list containing GPIO capablities informations (using **`adl_ioDefs_t **`** type).

Outputs available for each array element:

- the GPIO label (see section adl_ioLabel_etype).
- the GPIO direction (see section adl_ioDir_etype).
- the GPIO capabilities (see section adl_ioCap_e type).
- the GPIO status (see section adl_ioStatus_e type).

GpioTypeNb:

Returned the number of each GPIO, GPO and GPI. **GpioTypeNb** is an optional parameter, not used if set to NULL.

Returned values

- **`OK`** on success.
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.

3.10.4. 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 on 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 called from a low level Interrupt handler.

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

3.10.5. 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.

Event:

Event is the received identifier; other parameters use depends on the event type.

Size:

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

Param:

Read value tables (using **adl_ioDefs_t *** type) for the **ADL_IO_EVENT_INPUT_CHANGED** event.

Outputs available for each array element:

- the GPIO label (see section adl_ioLabel_etype).
- the GPIO level (see section adl_ioLevel_etype).
- the GPIO error information (see section adl_ioError_etype).

3.10.6. 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

- A **`OK`** on success
- A negative error value otherwise:
  - **`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 interrupt handler (the function is forbidden in this context).

Example:

```
void my_ioGetCapabilitiesList ()
    {
        u32 My_Loop;
        ascii * My_Message = adl_memGet ( 100 );
        u32 My_GpioNb;
        adl_ioDefs_t * My_GpioTab = NULL;
        adl_ioCap_t GpioTypeNb;

        adl_ioGetCapabilitiesList ( &My_GpioNb , &My_GpioTab ,
        &GpioTypeNb );

        wm_sprintf ( My_Message , "\r\nRessources : %d GPIO, %d GPI and
        %d GPO \r\n" , GpioTypeNb.NbGpio , GpioTypeNb.NbGpi ,
        GpioTypeNb.NbGpo );
        adl_atSendResponse ( ADL_AT_UNS, My_Message );

        adl_atSendResponse ( ADL_AT_UNS, "\r\nList of GPIO :\r\n" );

        for ( My_Loop = 0 ; My_Loop < My_GpioNb ; My_Loop++ )
        {
            switch ( My_GpioTab [ My_Loop ] & ADL_IO_TYPE_MSK )
            {
                case ADL_IO_GPI :
                    wm_sprintf ( My_Message, "GPI %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
                case ADL_IO_GPIO :
                    wm_sprintf ( My_Message, "GPIO %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
                case ADL_IO_GPO :
                    wm_sprintf ( My_Message, "GPO %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
            }
            adl_atSendResponse ( ADL_AT_UNS, My_Message );

            ... // customer treatment

        }

        adl_memRelease ( My_Message );

        // My_GpioTab must be released by the customer application
        adl_memRelease ( My_GpioTab );
    }
```

3.10.7. 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_ioDefs_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_ioDefs_t`** structure members have to be configured.

- Inputs to set for each array element:
  - the label of the GPIO to subscribe (see section adl_ioLabel_etype**)**.
  - the GPIO direction (see section adl_ioDir_etype).
  - the GPIO level, only if the GPIO is an output (see section adl_ioLevel_etype).
- Outputs available for each array element:
  - the GPIO error information (see section adl_ioError_etype**)**.

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

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 otherwise (No GPIO is reserved):
  - **`ADL_RET_ERR_PARAM`** if a parameter has an incorrect value,

   - **`ADL_RET_ERR_DONE`** refers to the field 3.10.2.6 **`adl_ioError_e`** for more information.
   - **`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 Interrupt handler (the function is forbidden in this context).

3.10.8. 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

- A **`OK`** on success.
- A negative error value otherwise:
   - **`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 Interrupt handler (the function is forbidden in this context).

3.10.9. 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_ioDefs_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_ioDefs_t *` type).

- Inputs to set for each array element:
   - the label of the GPIO to modify (see section adl_ioLabel_etype).
   - the new GPIO direction (see section adl_ioDir_etype).
- Outputs available for each array element:
   - the GPIO error information (see section adl_ioError_etype)

Returned values

- `OK` on success.
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`ADL_RET_ERR_DONE`** refers to the field adl_ioError_e for more information for each GPIO. If the error information is **`ADL_IO_GNOERR`**, the process has been completed with success for this GPIO.
  - **`ADL_RET_ERR_UNKNOWN_HDL`** if the handle is unknown.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.10.10. 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_ioDefs_t*     GpioRead );
```

Parameters

GpioHandle:

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

GpioNb:

Size of the **`GpioRead`** array.

GpioRead:

GPIO read structure array (using the **`adl_ioDefs_t *`** type).

- Inputs to set for each array element:
  - the label of the GPIO to read (see section adl_ioLabel_etype).
- Outputs available for each array element:
  - the GPIO level value (see section adl_ioLevel_etype).
  - the GPIO error information (see section adl_ioError_etype)

Returned values

- `OK` on success (read values are updated in the **`GpioArray`** parameter).
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`ADL_RET_ERR_DONE`** refers to the field adl_ioError_e for more information. If the error information is **`ADL_IO_GNOERR`**, the process has been completed with success for this GPIO.
  - **`ADL_RET_ERR_UNKNOWN_HDL`** if the handle is unknown.

3.10.11. The adl_ioReadSingle Function

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

Prototype

```
s32    adl_ioReadSingle ( s32              GpioHandle,
                          adl_ioDefs_t*    Gpio );
```

Parameters

GpioHandle:

Handle previously returned by **adl_ioSubscribe** function.

Gpio:

Identifier of the GPIO (see **adl_ioLabel_e**).

Returned values

- GPIO read value on success (1 for a high level or 0 for a low level),
- A negative error value otherwise
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
  - **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.

3.10.12. The adl_ioWrite Function

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

Prototype

```
s32    adl_ioWrite (  s32              GpioHandle,
                      u32              GpioNb,
                      adl_ioDefs_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_ioDefs_t * type).

- Inputs to set for each array element:
  - the label of the GPIO to write (see section adl_ioLabel_etype).
  - the new GPIO level (see section adl_ioLevel_etype).
- Outputs available for each array element:
  - the GPIO error information (see section adl_ioError_etype).

Returned values

- **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
  - **ADL_RET_ERR_DONE** refers to the field adl_ioError_e for more information. If the error information is **ADL_IO_GNOERR**, the process has been completed with success for this GPIO.

- **`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 output.

3.10.13. The adl_ioWriteSingle Function

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

Prototype

```
s32   adl_ioWriteSingle (  s32            GpioHandle,
                           adl_ioDefs_t*  Gpio,
                           bool           State );
```

Parameters

GpioHandle:

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

Gpio:

Identifier of the GPIO (see section adl_ioLabel_etype).

State:

Value to be set on the output:

- TRUE for a high level.
- FALSE for a low level.

Returned values

- **`OK`** on success.
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`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.

3.10.14. Example

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

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

u32 My_Gpio_Label1 [ GPIO_COUNT1 ] = { 1 , 2 };
u32 My_Gpio_Label2 [ GPIO_COUNT2 ] = { 3 };

adl_ioDefs_t* MyGpioConfig1 [ GPIO_COUNT1 ] =
{
    ( ADL_IO_GPIO | 1| ADL_IO_DIR_OUT | ADL_IO_LEV_LOW ) ,
    ( ADL_IO_GPIO | 2| ADL_IO_DIR_IN)
};
 adl_ioDefs_t* MyGpioConfig2 [ GPIO_COUNT2 ] =
 { ADL_IO_GPIO | 3| ADL_IO_DIR_IN };


// Gpio 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 :
         {
                u32 My_Loop;
                // The subscribed input has changed
                for ( My_Loop = 0 ; My_Loop < Size ; My_Loop++)
                {
                    if (( ADL_IO_TYPE_MSK & ((adl_ioDefs_t *)Param)[ My_Loop ]
)
                          && ADL_IO_GPO )
                    {
                        TRACE (( 1, "GPO %d new value: %d",
                        (((adl_ioDefs_t *)Param)[ My_Loop ] ) & ADL_IO_NUM_MSK
,
                        ((((adl_ioDefs_t *)Param)[ My_Loop ]) & ADL_IO_LEV_MSK
) &
                          ADL_IO_LEV_HIGH  ));
```

```
            }
            else
            {
                TRACE (( 1, "GPIO %d new value: %d",
                ( ((adl_ioDefs_t *)Param)[ My_Loop ] ) & ADL_IO_NUM_MSK ,
                ( (((adl_ioDefs_t *)Param)[ My_Loop ] ) & ADL_IO_LEV_MSK ) &
                  ADL_IO_LEV_HIGH  ));
             }
          }
       }
       break;
   }
 }

    ...
// Somewhere in the application code, used as an event handler
    void MyFunction ( void )
    {
        // Local variables
        s32 ReadValue;
        adl_ioDefs_t Gpio_to_write1 = ADL_IO_GPIO | My_Gpio_Label1 [ 0 ] ;
        adl_ioDefs_t Gpio_to_read1 = ADL_IO_GPIO | My_Gpio_Label1 [ 1 ] ;
        adl_ioDefs_t Gpio_to_read2 = ADL_IO_GPIO | My_Gpio_Label2 [ 0 ] ;

        // 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, &Gpio_to_write1 , TRUE );

        // Read inputs
        ReadValue = adl_ioReadSingle (MyGpioHandle1, &Gpio_to_read1 );
        ReadValue = adl_ioReadSingle (MyGpioHandle2, &Gpio_to_read2 );

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

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

3.11. Bus Service

The ADL supplies interface to handle bus operations.

The defined operations are:

- **adl_busSubscribe** to open a bus
- **adl_busUnsubscribe** to close a bus
- **adl_busIOCtl** to modify the behavior of the bus
- **adl_busRead** & **adl_busReadExt** to read on the a SPI or I2C bus
- **adl_busWrite** & **adl_busWriteExt** to write on the a SPI or I2C bus
- **adl_busDirectWrite** & **adl_busDirectRead** to write on the Parallel bus

3.11.1. Required Header File

The header file for the bus functions is:

adl_bus.h

3.11.2. Capabilities Registry Informations

3.11.2.1. The adl_busSpiCommonCap_e Type

SPI block common capabilities.

Code:

```
typedef enum
{
      ADL_BUS_SPI_COMMON_CAP_MASTER  = (1<<0),
      ADL_BUS_SPI_COMMON_CAP_SLAVE   = (1<<1),
      ADL_BUS_SPI_COMMON_CAP_2W      = (1<<2),
      ADL_BUS_SPI_COMMON_CAP_3W      = (1<<3),
      ADL_BUS_SPI_COMMON_PADDING     = 0x7fffffff
} adl_busSpiCommonCap_e;
```

Description:

ADL_BUS_SPI_COMMON_CAP_MASTER	The block can be used in master mode.
ADL_BUS_SPI_COMMON_CAP_SLAVE	The block can be used in slave mode. Reserved for future use.
ADL_BUS_SPI_COMMON_CAP_2W	The block can be configured to use 2 wires (DAT and CLK).
ADL_BUS_SPI_COMMON_CAP_3W	The block can be configured to use 3 wires (MISO, MOSI and CLK).

3.11.2.2. The adl_busSpiCap_e Type

SPI block capabilities in Master or Slave mode.

Code:

```
typedef enum
{
       ADL_BUS_SPI_CAP_BUSY        = (1<<0),
       ADL_BUS_SPI_CAP_LOAD        = (1<<1),
       ADL_BUS_SPI_CAP_CS_NONE     = (1<<2),
       ADL_BUS_SPI_CAP_CS_GPIO     = (1<<3),
       ADL_BUS_SPI_CAP_CS_HARD     = (1<<4),
       ADL_BUS_SPI_CAP_MSB         = (1<<5),
       ADL_BUS_SPI_CAP_LSB         = (1<<6),
       ADL_BUS_SPI_CAP_MICROWIRE   = (1<<7),
       ADL_BUS_SPI_CAP_MASK        = (1<<8),
       ADL_BUS_SPI_CAP_SHIFT       = (1<<9),
       ADL_BUS_SPI_CAP_PADDING     = 0x7fffffff
} adl_busSpiCap_e;
```

Description:

`ADL_BUS_SPI_CAP_BUSY`	The block can use a BUSY signal.
`ADL_BUS_SPI_CAP_LOAD`	The block can use a LOAD signal.
`ADL_BUS_SPI_CAP_CS_NONE`	The block can work without Chip Select.
`ADL_BUS_SPI_CAP_CS_GPIO`	The block can work with a GPIO as Chip Select.
`ADL_BUS_SPI_CAP_CS_HARD`	The block can work with a dedicated hardware pin as Chip Select.
`ADL_BUS_SPI_CAP_MSB`	The block can send data MSB first.
`ADL_BUS_SPI_CAP_LSB`	The block can send data LSB first.
`ADL_BUS_SPI_CAP_MICROWIRE`	The block can be used in Microwire mode.
`ADL_BUS_SPI_CAP_MASK`	The block has a mask possibility.
`ADL_BUS_SPI_CAP_SHIFT`	The block has a shift possibility.

3.11.2.3. The adl_busI2CCap_e Type

I2C block capabilities.

Code:

```
typedef enum
{
       ADL_BUS_I2C_CAP_ADDR_10_BITS   = (1<<0),
       ADL_BUS_I2C_CAP_MASTER         = (1<<1),
       ADL_BUS_I2C_CAP_SLAVE          = (1<<2),
       ADL_BUS_I2C_CAP_CLK_FAST       = (1<<3),
       ADL_BUS_I2C_CAP_CLK_HIGH       = (1<<4),
       ADL_BUS_I2C_CAP_ADD_SIZE_8     = (1<<5),
       ADL_BUS_I2C_CAP_ADD_SIZE_16    = (1<<6),
       ADL_BUS_I2C_CAP_ADD_SIZE_24    = (1<<7),
       ADL_BUS_I2C_CAP_ADD_SIZE_32    = (1<<8),
       ADL_BUS_I2C_CAP_PADDING        = 0x7fffffff
} adl_busI2CCap_e;
```

Description:

`ADL_BUS_I2C_CAP_ADDR_10_BITS`	The block can use 10 bits addressing mode. **`Reserved for future use`**
`ADL_BUS_I2C_CAP_MASTER`	The block can be used in master mode.
`ADL_BUS_I2C_CAP_SLAVE` .	The block can be used in slave mode.
`ADL_BUS_I2C_CAP_CLK_FAST`	The block can use Fast clock (400 kbits/s).
`ADL_BUS_I2C_CAP_CLK_HIGH`	The block can use High Speed clock (3.4 Mbits/s).
`ADL_BUS_I2C_CAP_ADD_SIZE_8`	The address size can be 8 bits (see `ADL_BUS_CMD_SET_ADD_SIZEe IOCtl` command).
`ADL_BUS_I2C_CAP_ADD_SIZE_16`	The address size can be 16 bits (see `ADL_BUS_CMD_SET_ADD_SIZE IOCtl` command).
`ADL_BUS_I2C_CAP_ADD_SIZE_24`	The address size can be 24 bits (see `ADL_BUS_CMD_SET_ADD_SIZE IOCtl` command).
`ADL_BUS_I2C_CAP_ADD_SIZE_32`	The address size can be 32 bits (see `ADL_BUS_CMD_SET_ADD_SIZE IOCtl` command).

3.11.3. Common Data Structures and Enumerations

ADL provides capabilities information about the BUS service, thanks to the registry service.

The following entries are defined in the registry:

Registry entry	Type	Description
i2c_NbBlocks[3]	INTEGER	The number of i2c blocks managed by the embedded module
i2c_xx_Cap	INTEGER	The capabilities of the block, defined as a combination of the **`adl_busI2CCap_e`** type values.
i2c_xx_MaxLength	Unsigned INTEGER[4]	The maximum amount of items that can be passed in a I2C read/write operation
spi_NbBlocks[3]	INTEGER	The number of spi blocks managed by the embedded module
spi_xx_Common	INTEGER	The generic capabilities of the block, defined as a combination of the **`adl_busSpiCommonCap_e`** type values.
spi_xx_ClockDivStep	INTEGER	The number of steps of the clock divider (see adl_busSPISettings_t **`::Clk_Speed`** field description)
spi_xx_MaxLength	INTEGER	The maximum amount of items that can be passed in a SPI read/write operation
spi_xx_DataSizes[2]	INTEGER	Available data sizes for **`ADL_BUS_CMD_SET_DATA_SIZE IOCtl`** command
spi_xx_Master_OpcodeSizes[2]	Unsigned INTEGER[4]	Available Opcode sizes for **`ADL_BUS_CMD_SET_OP_SIZE IOCtl`** command
spi_xx_Master_AddressSizes[2]	Unsigned INTEGER[4]	Available Address sizes for **`ADL_BUS_CMD_SET_ADD_SIZE IOCtl`** command
spi_xx_Master_Cap	INTEGER	The capabilities of the block in Master mode, defined as a combination of the **`adl_busSpiCap_e type`**
spi_xx_Master_MaxFreqClock	INTEGER	The maximum frequency (in kHz) of the clock in Master mode (see adl_busSPISettings_t**`::Clk_Speed`** field description).

Registry entry	**Type**	**Description**
Para_NbBlocks[3]	INTEGER	The number of parallel bus blocks managed by the embedded module
Para_NbCS	INTEGER	The number of chip select available to the customer
Para_CS	INTEGER	The list of currently accessible chip select * This is a bitfield, each bit represents a CS available. e.g. : Para_CS = 5, the Parallel bus 1 has 2 CS available : CS0 (b0) and CS2 (b2)
Para_xx_Addr	INTEGER	Current address of the Chip select XX
Para_xx_Freq	INTEGER	Current frequency of the Chip select XX

Note: *1. For the registry entry the **xx** part has to be replaced by the number of the instance.*
*Example: if you want the capabilities of the I2C1 block the registry entry to use will be **i2c_01_Cap.** **Example:** if you want the common capabilities of the SPI2 block the registry entry to use will be **spi_02_Common**.*

Note: *2. Sizes are coded in a bit field, where size n is available when the n-1 bit is set.*
*Example: **0x80008003** means sizes 32 bits, 16 bits, 2 bits and 1 bit are available.*

Note: *3. A SPI/I2C/Parallel bus block will be identified with a number from 1 to **spi_NbBlocks** or **i2c_NbBlocks** or **Parallel_NbBlocks**.*

Note: *4. Entries using the Unsigned INTEGER type have to be casted to an u32 value after being retrieved from adl_regGetHWInteger function.*

3.11.3.1. The adl_busSettings_u Type

Generic bus settings union.

Code

```
typedef struct
  {
       adl_busSPISettings_t  SPI;
       adl_busI2Settings_t   I2C;
  }adl_busSettings_u;
```

Description

SPI

SPI member, previously handle SPI related settings.

I2C

I2C member, previously to handle 12C related settings.

3.11.3.2. The adl_busID_e Type

This type allows to identify the bus types supported by the service.

Code:

```
typedef enum
{
      ADL_BUS_ID_SPI,           //SPI Bus
      ADL_BUS_ID_I2C,           //I2C Bus
      ADL_BUS_ID_PARALLEL,      //Parallel Bus
      ADL_BUS_ID_LAST,          //Reserved for internal use
} adl_busID_e;
```

3.11.3.3. The adl_busType_e Type

Former enumeration used to identify BUS types.

Code:

```
typedef enum
{
      ADL_BUS_SPI1,
      ADL_BUS_SPI2,
      ADL_BUS_I2C,
      ADL_BUS_PARALLEL
} adl_busType_e;
```

Description:

`ADL_BUS_SPI1`	This constant was previously used to access the embedded module SPI1 bus.
`ADL_BUS_SPI2`	This constant was previously used to access the embedded module SPI2 bus
`ADL_BUS_I2C`	This constant was previously used to access the embedded module I2C bus
`ADL_BUS_PARALLEL`	This constant was previously used to access the embedded module Parallel bus

3.11.4. SPI Bus Subscription Data Structures and Enumerations

3.11.4.1. The adl_busSPISettings_t Type

SPI bus settings.

Code:

```
typedef struct
{
        u32             Clk_Speed;
        u32             Clk_Mode;
        u32             ChipSelect;
        u32             ChipSelectPolarity;
        u32             LsbFirst;
        adl_ioDefs_t    GpioChipSelect;
        u32             LoadSignal;
        u32             DataLinesConf;
        u32             MasterMode;
        u32             BusySignal;
} adl_busSPISettings_t;
```

Description:

Clk_Speed

The Clk_Speed parameter is a divider that allows to modify SPI bus clock speed.

Valid values are in the [0 – (N-1)] range, where N is the spi_xx_ClockDivStep capability.

The SPI clock speed (in kHz) is defined using the formula below:

MaxFrequency / (1 + ClkSpeed)

Where MaxFrequency is the embedded module maximum frequency for the current SPI block (spi_xx_Master_MaxFreqClock capability).

Example: if Clk_Speed is set to 0, and Max_Frequency is 13000 kHz, the SPI bus clock speed is set to 13000 kHz.

Note: *The MaxFrequency can be changed by the command AT+WCPS=1,x.*

While subscribing to SPI bus, check the current SPI MaxFrequency to know which Clk_Speed value to use by reading the spi_xx_Master_MaxFreqClock capability using adl_regGetHWInteger.

Clk_Mode

This parameter is the SPI clock mode (see adl_busSPI_Clk_Mode_e).

ChipSelect

This parameter sets the pin used to handle the Chip Select signal (see adl_busSPI_ChipSelect_e).

ChipSelectPolarity

This parameter sets the polarity of the Chip Select signal (see adl_busSPI_ChipSelectPolarity_e).

LsbFirst

This 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 (see adl_busSPI_LSBfirst_e).

GpioChipSelect

This parameter defines the GPIO Chip Select. This 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 (see adl_ioDefs_t).

LoadSignal

This parameter defines the LOAD signal behavior (see adl_busSPI_Load_e).

DataLinesConf

This 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 (see adl_busSPI_DataLinesConf_e).

MasterMode

This parameter is the SPI master or slave running mode (see adl_busSPI_MS_Mode_e).

BusySignal

This parameter defines the LOAD signal behavior (see adl_busSPI_Busy_e).

Note: *The BUSY and LOAD signals cannot be used on the WMP100. These signals will be available in a forthcoming update.*

3.11.4.2. The adl_busSPI_Clk_Mode_e Type

SPI bus Clock Modes. See also adl_busSPISettings_t for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_CLK_MODE_0,
        ADL_BUS_SPI_CLK_MODE_1,
        ADL_BUS_SPI_CLK_MODE_2,
        ADL_BUS_SPI_CLK_MODE_3,
        ADL_BUS_SPI_CLK_MODE_MICROWIRE,
} adl_busSPI_Clk_Mode_e;
```

Description:

ADL_BUS_SPI_CLK_MODE_0	Mode 0: rest state 0, data valid on rising edge.
ADL_BUS_SPI_CLK_MODE_1	Mode 1: rest state 0, data valid on falling edge.
ADL_BUS_SPI_CLK_MODE_2	Mode 2: rest state 1, data valid on falling edge.
ADL_BUS_SPI_CLK_MODE_3	Mode 3: rest state 1, data valid on rising edge
ADL_BUS_SPI_CLK_MODE_MICROWIRE	Microwire mode. See also `ADL_BUS_SPI_CAP_MICROWIRE` Capability.

3.11.4.3. The adl_busSPI_ChipSelect_e Type

SPI bus Chip Select. See also adl_busSPISettings_t for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_ADDR_CS_GPIO,
        ADL_BUS_SPI_ADDR_CS_HARD,
        ADL_BUS_SPI_ADDR_CS_NONE,
 } adl_busSPI_ChipSelect_e;
```

Description:

`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.

3.11.4.4. The adl_busSPI_ChipSelectPolarity_e Type

SPI bus Chip Select Polarity. See also adl_busSPISettings_t for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_CS_POL_LOW,
        ADL_BUS_SPI_CS_POL_HIGH,
 } adl_busSPI_ChipSelectPolarity_e;
```

Description:

`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.

3.11.4.5. The adl_busSPI_LSBfirst_e Type

SPI bus MSB/LSB First. See also adl_busSPISettings_t for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_MSB_FIRST,
        ADL_BUS_SPI_LSB_FIRST
 } adl_busSPI_LSBfirst_e;
```

Description:

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

3.11.4.6. The adl_busSPI_WriteHandling_e Type

SPI bus Write Handling.

Kept for ascendant compatibility. The adl_busSPI_Load_e type shall be used instead.

Code:

```
typedef enum
{
        ADL_BUS_SPI_FRAME_HANDLING,
        ADL_BUS_SPI_WORD_HANDLING
 } adl_busSPI_WriteHandling_e;
```

Description:

ADL_BUS_SPI_FRAME_HANDLING	LOAD signal is enabled at the beginning of the read/write process, and is disabled at the end of this process.
ADL_BUS_SPI_WORD_HANDLING	LOAD signal state changes on each written or read word.

3.11.4.7. The adl_busSPI_Load_e Type

SPI bus LOAD signal configuration. See also adl_busSPISettings_t & ADL_BUS_SPI_CAP_LOAD for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_LOAD_UNUSED,
        ADL_BUS_SPI_LOAD_USED
 } adl_busSPI_Load_e;
```

Description:

ADL_BUS_SPI_LOAD_UNUSED	The LOAD signal is not used.
ADL_BUS_SPI_LOAD_USED	The LOAD signal is used (LOAD signal state changes on each written or read word; word size is defined thanks to ADL_BUS_CMD_SET_DATA_SIZE IOCtl command. Please refer to the Product Technical Specification document for more information about the LOAD signal).

Note: *The BUSY and LOAD signals cannot be used on the WMP100. These signals will be available in a forthcoming update.*

3.11.4.8. The adl_busSPI_DataLinesConf_e Type

SPI bus Data Lines configuration. See also adl_busSPISettings_t, ADL_BUS_SPI_COMMON_CAP_2W & ADL_BUS_SPI_COMMON_CAP_3W capabilities for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_DATA_BIDIR,
        ADL_BUS_SPI_DATA_UNIDIR
 } adl_busSPI_DataLinesConf_e;
```

Description:

`ADL_BUS_SPI_DATA_BIDIR`	2 wires mode (DAT and CLK), one bi-directional pin is used to handle both input & output data signals.
`ADL_BUS_SPI_DATA_UNIDIR`	3 wires mode (MISO, MOSI and CLK), two pins are used to handle separately input & output data signals.

3.11.4.9. The adl_busSPI_MS_Mode_e Type

Master/Slave bus mode configuration. See also adl_busSPISettings_t, ADL_BUS_SPI_COMMON_CAP_MASTER & ADL_BUS_SPI_COMMON_CAP_SLAVE capabilities for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_MASTER_MODE,
        ADL_BUS_SPI_SLAVE_MODE
 } adl_busSPI_MS_Mode_e;
```

Description:

`ADL_BUS_SPI_MASTER_MODE`	The SPI bus is running in master mode (default value when adl_busSubscribe function is used).
`ADL_BUS_SPI_SLAVE_MODE`	The SPI bus is running in slave mode. **Reserved for future use.**

3.11.4.10. The adl_busSPI_Busy_e Type

SPI bus BUSY signal configuration. See also adl_busSPISettings_t & ADL_BUS_SPI_CAP_BUSY capability for more information.

Code:

```
typedef enum
{
        ADL_BUS_SPI_BUSY_UNUSED,
        ADL_BUS_SPI_BUSY_USED
 } adl_busSPI_Busy_e;
```

Description:

ADL_BUS_SPI_BUSY_UNUSED	The BUSY signal is not used (default value when adl_busSubscribe function is used).
ADL_BUS_SPI_BUSY_USED	The BUSY signal is used

Note: *The BUSY and LOAD signals cannot be used on the WMP100. These signals will be available in a forthcoming update.*

3.11.5. I2C Bus Subscription Data Structures and Enumerations

3.11.5.1. The adl_busI2CSettings_t Type

This structure defines the I2C bus settings for subscription.

Note: *Please refer to the Product Technical Specification for more information.*

Code:

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

Description:

ChipAddress

This parameter sets the remote chip **N** bit address on the I2C bus.

b0 to b6 bits are used.

Example:

If the remote chip address is set to A0, the ChipAddress parameter has to be set to the 0xA0 value.

Clk_Speed

This parameter sets the required I2C bus speed (see adl_busI2C_Clk_Speed_e).

AddrLength

This parameter sets the remote chip address length configuration (see adl_busI2C_AddrLength_e).

MasterMode

This parameter is the I2C master or slave running mode (see adl_busI2C_MS_Mode_e).

3.11.5.2. The adl_busI2C_Clk_Speed_e Type

I2C bus Clock Speed. See also adl_busI2CSettings_t, ADL_BUS_I2C_CAP_CLK_FAST & ADL_BUS_I2C_CAP_CLK_HIGH capabilities for more information.

Code:

```
typedef enum
{
        ADL_BUS_I2C_CLK_STD,
        ADL_BUS_I2C_CLK_FAST,
        ADL_BUS_I2C_CLK_HIGH
 } adl_busI2C_Clk_Speed_e;
```

Description:

`ADL_BUS_I2C_CLK_STD`	Standard I2C bus speed (100 kbits/s).
`ADL_BUS_I2C_CLK_FAST`	Fast I2C bus speed (400 kbits/s).
`ADL_BUS_I2C_CLK_HIGH`	High I2C bus speed (3.4 Mbits/s).

3.11.5.3. The adl_busI2C_AddrLength_e Type

I2C bus chip address length. See also adl_busI2CSettings_t & ADL_BUS_I2C_CAP_ADDR_10_BITS capability for more information.

Code:

```
typedef enum
{
        ADL_BUS_I2C_ADDR_7_BITS,
        ADL_BUS_I2C_ADDR_10_BITS
 } adl_busI2C_AddrLength_e;
```

Description:

`ADL_BUS_I2C_ADDR_7_BITS`	Chip address is 7 bits long (default value if adl_busSubscribe function is used).
`ADL_BUS_I2C_ADDR_10_BITS`	Chip address is 10 bits long. .

3.11.5.4. The adl_busI2C_MS_Mode_e Type

Master/Slave bus mode configuration. See also adl_busI2CSettings_t & ADL_BUS_I2C_CAP_MASTER capability for more information.

Code:

```
typedef enum
{
        ADL_BUS_I2C_MASTER_MODE,
        ADL_BUS_I2C_SLAVE_MODE
 } adl_busI2C_MS_Mode_e;
```

Description:

`ADL_BUS_I2C_MASTER_MODE`	The I2C bus is running in master mode (default value when adl_busSubscribe function is used).
`ADL_BUS_I2C_SLAVE_MODE`	The I2C bus is running in slave mode. Reserved for future use.

3.11.6. Parallel Bus Subscription Data Structures and Enumerations

Note: *This is only applicable for WMP100. The parallel interface parameters are already automatically set up on chip select CS0 for Flash and CS1 for Memory and therefore inaccessible to the user. Two chip selects remain for user usage.*

3.11.6.1. The adl_busParallelCs_t Type

This type defines the Parallel bus Chip Select.

Please refer to the Product Technical Specification for more information.

Code:

```
typedef struct
{
        u8      Type;       //Chip select type
        u8      Id;         //Chip select identifier
        u8      Pad[2];     //Needed to be compliant with GCC alignment
} adl_busParallelCs_t;
```

Description:

Type

This 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 (see adl_busParallel_CS_Type_e).

Id

This parameter defines the Chip Select identifier used.

3.11.6.2. The adl_busParallelPageCfg_t Type

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

Code:

```
typedef struct
{
        u8      PageSize;            //Page size
        u8      PageAccessCycles;    //Between address change and valid
                                        data output
} adl_busParallelPageCfg_t;
```

3.11.6.3. The adl_busParallelSettings_t Type

Parallel bus settings.

Code

```
typedef struct
{
      u8                                  Width;
      u8                                  Mode;
      u8                                  pad [2];
      adl_busParallelTimingsCfg_t         ReadCfg;
      adl_busParallelTimingsCfg_t         WriteCfg;
      adl_busParallelCs_t                 Cs;
      adl_busParallelPageCfg_t            PageCfg;
      adl_busParallelSynchronousCfg_t     SynchronousCfg;
      u32                                 AddressPin;
} adl_busSPISettings_t;
```

Description:

Width

This parameter defines the read/write process data buffer items bit size, using the **adl_busParallelSize_e** type.

Mode

This parameter defines the required parallel bus standard mode to be used, using the **adl_busParallel_Bus_Mode_e** type.

ReadCfg

Define the timing configuration for each read and write process, using the **adl_busParallelTimingCfg_t** type.

WriteCfg

Define the timing configuration for each read and write process, using the **adl_busParallelTimingCfg_t** type.

Cs

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

PageCfg

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

SynchronousCfg

Configuration of the synchronous mode.

This structure hosts the parameters used to configure the synchronous mode accesses.

AddressPin

Select the pin used for the parallel bus.
This is a bitfield, each bit represents a pin of the parrallel bus.
e.g.: 0x03, two address pin are used (A0 and A1).

3.11.6.4. The adl_busParallelSynchronousCfg_t Type

Configuration parameters for the page mode.

This structure hosts the parameters used to configure the synchronous mode accesses.

Code:

```
typedef struct
{
        u8      BurstSize;          //Size of Burst size
        u8      ClockDivisor;       //Main Memory clock divider
        s32     UseWaitEnable:1;    //WS generation using WAIT#
        s32     WaitActiveDuringWS:1;//WAIT# during or 1-cycle before WS
        s32     Reserved:30;        //unused
  } adl_busParallelSynchronousCfg_t;
```

3.11.6.5. The adl_busParallelTimingCfg_t Type

Parallel bus Timing structure.

This type defines the Parallel bus timings.

Note: *The parameters configuration defines the parallel bus timing, in cycles number (please refer to the Product Technical Specification for more information), according to the bus mode required at subscription time (see **adl_busParallel_Bus_Mode_e**).*
Example: In 26 MHz cycles number, one cycle duration is 1/26 MHz = ~38.5 ns

Note: *The Para_xx_Freq value can be changed by the command AT+WCPS=1,x. You must query the Para_xx_Freq value at Parallel bus subscription to know the timing values to be used.*

Code

```
typedef struct
{
        u8      AccessTime;
        u8      SetupTime;
        u8      HoldTime;
        u8      TurnaroundTime;
        u8      OptoOpTurnaroundTime;
        u8      pad[3];             // Internal use only
} adl_busParallelTimingCfg_t;
```

Description:

AccessTime

Access Time (see adl_busParallel_Bus_Mode_e and the Product Technical Specification).

SetupTime

Setup Time (see adl_busParallel_Bus_Mode_e and the Product Technical Specification).

HoldTime

Hold Time (see adl_busParallel_Bus_Mode_e and the Product Technical Specification).

TurnaroundTime

Turnaround Time (see adl_busParallel_Bus_Mode_e and the Product Technical Specification).

OptoOpTurnaroundTime

Read-to-read/write-to-write turnaround Time.

(see adl_busParallel_Bus_Mode_e and the Product Technical Specification)

3.11.6.6. The adl_busParallelSize_e Type

Bus access width.

Multiplexed modes spare pins by multiplexing data and addresses on the same pins. All the access widths and access modes are not available, valid combinations depend on the platform.

Code

```
typedef enum
{
      ADL_BUS_PARALLEL_WIDTH_INVALID,                 // reserved
      ADL_BUS_PARALLEL_WIDTH_8_BITS,                  // 8-bit device
      ADL_BUS_PARALLEL_WIDTH_16_BITS,                 // 16-bit device
      ADL_BUS_PARALLEL_WIDTH_32_BITS,                 // 32-bit device
      ADL_BUS_PARALLEL_WIDTH_16_BITS_MULTIPLEXED,     // 16-bit multiplexed
                                                      device
      ADL_BUS_PARALLEL_WIDTH_32_BITS_MULTIPLEXED      //32-bit multiplexed
                                                      device
} adl_busParallelSize_e;
```

3.11.6.7. The adl_busParallel_Bus_Mode_e Type

Types of access.

Intel 8080 compatible and Motorola 6800 compatible asynchronous accesses modes can be configured:

- Intel mode uses an output enable or read enable signal and a write enable signal. In this read process example, Setup & Hold times are set to 1, and Access & Turnaround times are set to 3.

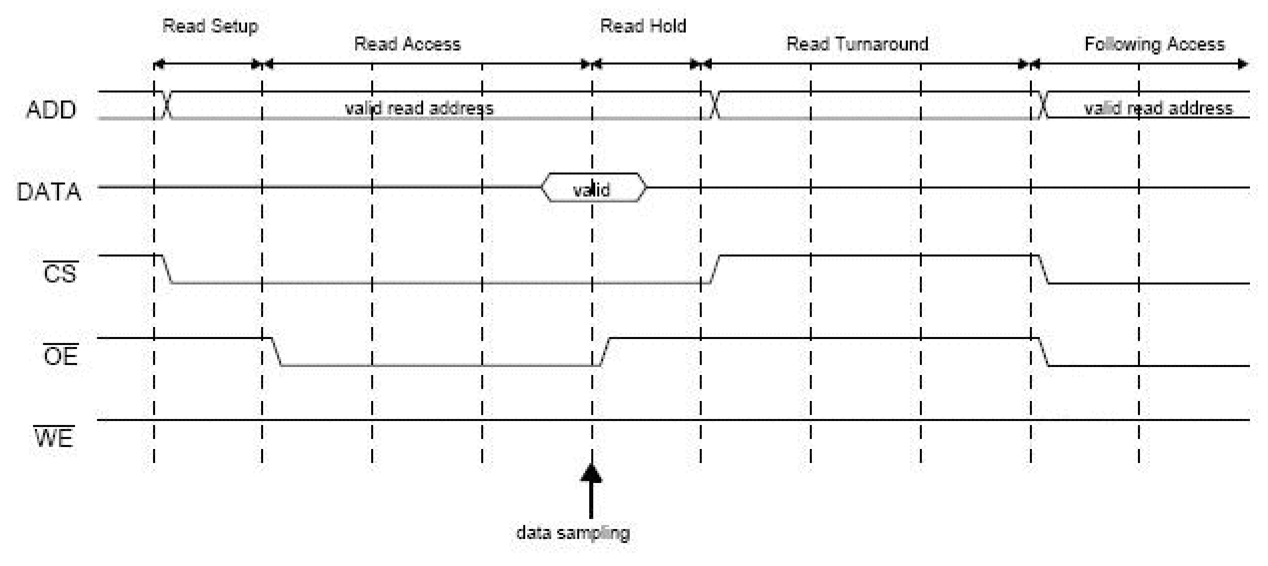


Figure 5. Intel Mode Timing - Read Process Example

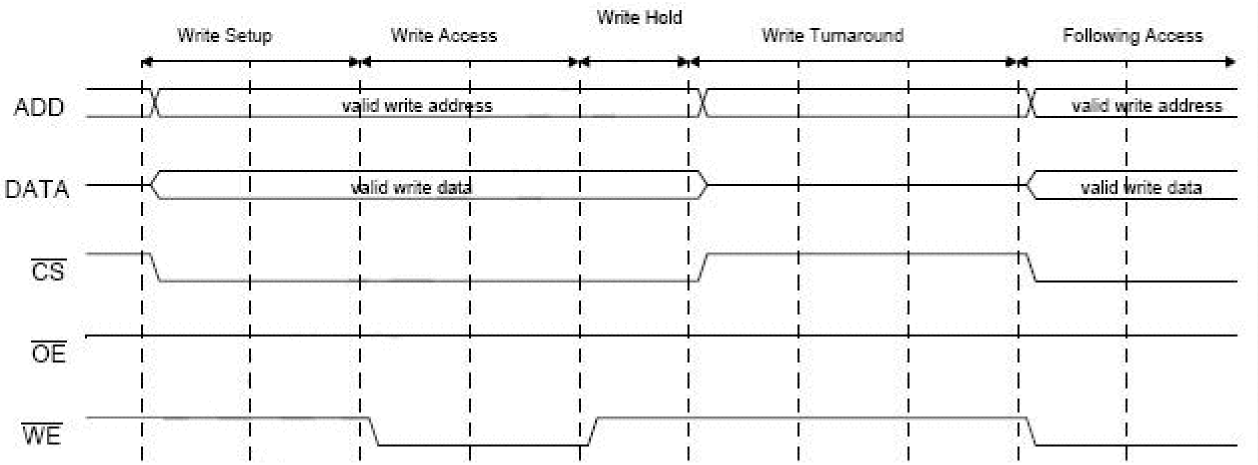


Figure 6. Intel Mode Timing - Write Process Example

- Motorola mode uses a read not write signal and an enable signal. The polarity of the enable signal can be configured:
  - E is active at high level with mode Motorola 0 (LOW)
  - E is active at low level with mode Motorola 1 (HIGH)

  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.

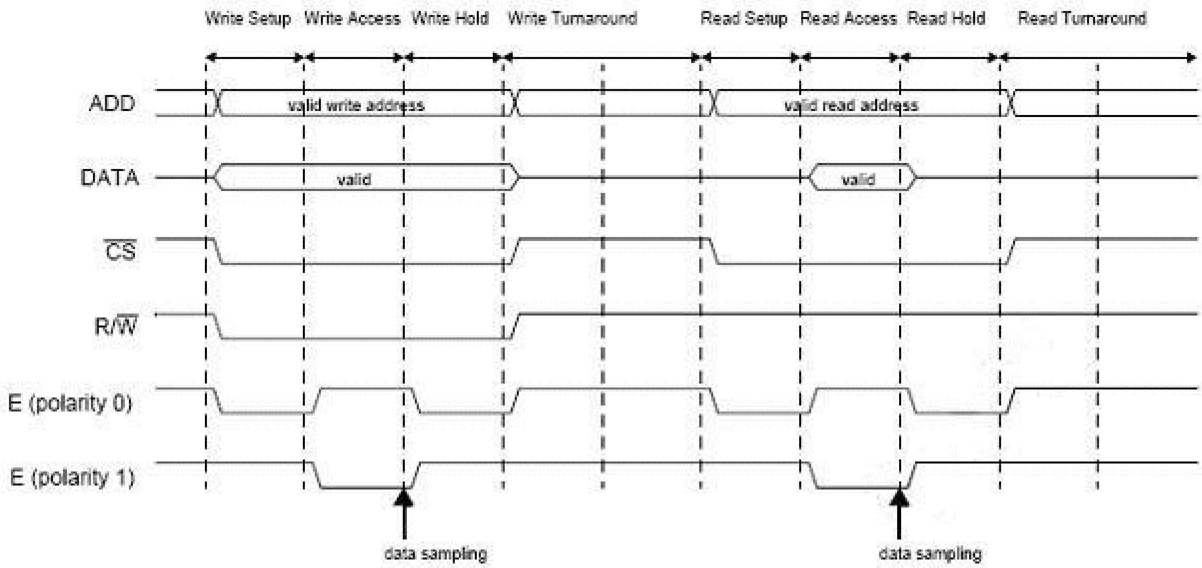


Figure 7. Motorola Modes Timing Example

Code

```
enum
{
ADL_BUS_PARALLEL_MODE_INVALID,                 // reserved
ADL_BUS_PARALLEL_MODE_ASYNC_INTEL,             // Intel 8080 compatible
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW,      // Motorola 6800 compatible,
                                               with E signal low polarity
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH,     // Motorola 6800 compatible,
                                               with E signal high polarity
ADL_BUS_PARALLEL_MODE_ASYNC_PAGE,              // Page mode
ADL_BUS_PARALLEL_MODE_SYNC_READ_ASYNC_WRITE,   // Synchronous only in reads
ADL_BUS_PARALLEL_MODE_SYNC_READ_WRITE          // Full synchronous mode
} adl_busParallel_Bus_Mode_e
```

3.11.6.8. The adl_busParallel_CS_Type_e Type

Parallel bus chip select type.

See also section adl_busParallelCs_t for more information.

Code

```
enum
{
       ADL_BUS_PARA_CS_TYPE_CS,    // Chip select type
} adl_busParallel_CS_Type_e
```

Description

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.

3.11.7. IOCtl Operations Data Structures and Enumerations

3.11.7.1. The adl_busAsyncInfo_t Type

This structure lists the information returned when an asynchronous read/write operation end event occurs.

Code:

```
typedef struct
{
       s32        Result;
} adl_busAsyncoInfo_t;
```

Description:

Result

Asynchronous read/write operation result code. See also adl_busWrite & adl_busRead functions return values description for more information.

3.11.7.2. The adl_busEvt_t Type

This structure allows to define the interrupt handlers which will be notified when the end of an asynchronous read/write operation event occurs.

Interrupt handlers defined in the IRQ service - using the adl_irqHandler_f type - are notified with the following parameters:

- the Source parameter will be set to **`ADL_IRQ_ID_SPI_EOT`** (for SPI bus operation) or **`ADL_IRQ_ID_I2C_EOT`** (for I2C bus operation).
- the **`adl_irqEventData_t`**::SourceData field of the Data parameter should be casted to the **`adl_busAsyncInfo_t * type`**, usable to retrieve information about the current interrupt event (if the **`ADL_IRQ_OPTION_AUTO_READ`** option has been required)
- the **`adl_irqEventData_t`**::Instance field of the Data parameter will have to be considered as an **`u32`** value, usable to identify which block has raised the current interrupt event (i.e. the BlockId provided at subscription time in adl_busSubscribe function).
- the **`adl_irqEventData_t`**::Context field of the Data parameter will be the application context, provided when the **`adl_busReadExt`** or **`adl_busWriteExt`** function was called. (It will be set to NULL if **`adl_busRead`** or **`adl_busWrite`** function was used)

Code:

```
typedef struct
{
      s32         LowLevelIrqHandle;
      s32         HighLevelIrqHandle;
} adl_busEvt_t;
```

Description:

LowLevelIrqHandle

Low level interrupt handler, previously returned by the **`adl_irqSubscribe`** function.

This parameter is optional if the **`HighLevelIrqHandle`** parameter is supplied.

HighLevelIrqHandle

High level interrupt handler, previously returned by the **`adl_irqSubscribe`** function. This parameter is optional if the **`LowLevelIrqHandle`** parameter is supplied.

3.11.7.3. The adl_busSpiMaskShift_t Type

The parameter type for the **`ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT`** and **`ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT IoCtl`** commands.

Code:

```
typedef struct
{
      u32                w_Mask;
      u32                w_Value;
      adl_busMaskSPI_e   Option;
      u8                 Pad [3];
} adl_busSpiMaskShift_t;
```

Description:

w_Mask

Each bit to "1" will stay unchanged and each bit to "0" will be replaced by the w_Value ones.

w_Value

The value to set in the masked bits.

Option

Enabled/disabled Mask and Shift modes.

Pad

Internal use only.

3.11.7.4. The adl_busMaskSPI_e Type

Definition of the parameters to enable/disable Mask and Shift modes.

Code:

```
typedef enum
{
      ADL_BUS_SPI_MASK_ENA        = (1L<<0),
      ADL_BUS_SPI_SHIFT_ENA       = (1L<<1),
 } adl_busMaskSPI_e;
```

Description:

`ADL_BUS_SPI_MASK_ENA`	Mask mode is enabled.
`ADL_BUS_SPI_SHIFT_ENA`	Shift mode is enabled.

3.11.7.5. The adl_busIoCtlCmd_e Type

Definition of the commands for adl_busIOCtl function.

Code:

```
typedef enum
{
      ADL_BUS_CMD_SET_DATA_SIZE
      ADL_BUS_CMD_GET_DATA_SIZE
      ADL_BUS_CMD_SET_ADD_SIZE
      ADL_BUS_CMD_GET_ADD_SIZE
      ADL_BUS_CMD_SET_OP_SIZE
      ADL_BUS_CMD_GET_OP_SIZE
      ADL_BUS_CMD_LOCK
      ADL_BUS_CMD_UNLOCK
      ADL_BUS_CMD_GET_LAST_ASYNC_RESULT
      ADL_BUS_CMD_SET_ASYNC_MODE
      ADL_BUS_CMD_GET_ASYNC_MODE
      ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT
      ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT
      ADL_BUS_CMD_SET_PARALLEL_CFG
      ADL_BUS_CMD_GET_PARALLEL_CFG
      ADL_BUS_CMD_PARA_GET_ADDRESS
      ADL_BUS_CMD_PARA_GET_MAX_SETTINGS
      ADL_BUS_CMD_PARA_GET_MIN_SETTINGS
      ADL_BUS_CMD_PADDING            = 0x7fffffff
} adl_busIoCtlCmd_e;
```

Description:

`ADL_BUS_CMD_SET_DATA_SIZE`	Set the size in bits of one data element. **Parameters**: The Param of adl_busIoCtl is defined as a pointer to an u32 value. See also spi_xx_DataSizes Capabilities for the available values, default value is 8. *Note: Available for the SPI Bus only.*
`ADL_BUS_CMD_GET_DATA_SIZE`	Get the size in bits of one data element. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32 value. *Note: Available for the SPI Bus only.*
`ADL_BUS_CMD_SET_ADD_SIZE`	Set the size in bits of the address. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32 value. See also spi_xx_MasterAddressSizes and adl_busI2CCap_e capabilities for the available values, default value is zero (address is not used).
`ADL_BUS_CMD_GET_ADD_SIZE`	Set the size in bits of the address. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32 value. *Note: Available for the SPI and I2C Bus only.*
`ADL_BUS_CMD_SET_OP_SIZE`	Set the size in bits of the Opcode. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32 value. *Note: Available for the SPI Bus only.*
`ADL_BUS_CMD_GET_OP_SIZE`	Get the size in bits of the Opcode. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32 value. *Note: Available for the SPI Bus only.*
`ADL_BUS_CMD_LOCK`	Lock a bus to avoid concurrent access and to allow access to the bus in interrupt context. After this call, the block is locked and only the handle which has locked it can use this block. **Parameters:**The Param of adl_busIoCtl is not relevant and can be set to NULL. *Note: Available for the SPI and I2C Bus only.* Trying to lock a second time a given block with the same handle will lead to an **ADL_RET_ERR_BAD_HDL** error. Trying to lock a bus which is already locked by another handle will lead the current task context to be suspended, until the block is unlocked, thanks to the **ADL_BUS_CMD_UNLOCK** command. Assuming several handles have subscribed in the same block. If handle1 has locked the block and handle2 attempts to access the

	same block handle1 will be suspended so that handle2 accesses the block. When handle2 releases the block handle1 will resume its operation. **Warning:** *This command is available only in asynchronous mode.*
ADL_BUS_CMD_UNLOCK	Unlock a bus previously locked by **ADL_BUS_CMD_LOCK** command. **Parameters:** The Param of adl_busIoCtl is not relevant and can be set to NULL. Note: *Available for the SPI and I2C Bus only.* If a task context was suspended due to a **ADL_BUS_CMD_LOCK** command on this block, it will be resumed as soon as the block is unlocked.
ADL_BUS_CMD_GET_LAST_ASYNC_RESULT	Get the last asynchronous read/write operation of return value. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an adl_busAsyncInfo_t structure. Note: *Available for the SPI and I2C Bus only.*
ADL_BUS_CMD_SET_ASYNC_MODE	Configure the Synchronous/asynchronous mode settings **Parameters:** The Param of adl_busIOCtl is defined as pointer on adl_busEvt_t. When this parameter is set to a value different of NULL, adl_busWrite and adl_busRead behaviour become asynchronous. When it is set to NULL, read/write operations are synchronous (default value). Note: *Available for the SPI and I2C Bus only.*
ADL_BUS_CMD_GET_ASYNC_MODE	Get the current value of the synchronous/asynchronous mode settings. **Parameters:** The Param of adl_busIOCtl is defined as a pointer on adl_busEvt_t. If the current mode is synchronous, all elements of Param\ are NULL. Available for the SPI and I2C Bus only.
ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT	Enable/disable and set the parameters for the mask and shift modes. **Parameters:** The Param of adl_busIOCtl is defined as a pointer on adl_busSpiMaskShift_t. Note: *Available for the SPI Bus only.* **Warning:** *Reserved for future use*
ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT	Get the status and the parameters for the mask and shift modes. **Parameters:**The Param of adl_busIOCtl is defined as a pointer on adl_busSpiMaskShift_t.

	Note: Available for the SPI Bus only.
ADL_BUS_CMD_SET_PARALLEL_CFG	Set the Parallel configuration for one subscribed bus. **Parameters:** The Param of adl_busIoCtl is defined as a pointer on adl_busParallelSettings_t. *Note: Available for the Parallel Bus only.* Parameter AddressPin and CS.Id are specific to the subscribed bus, therefore they cannot be changed. If they are changed, it will have no effect and no error will be returned.
ADL_BUS_CMD_GET_PARALLEL_CFG	Get the Parallel configuration for one subscribed bus. **Parameters:** The Param of adl_busIoCtl is defined as a pointer on adl_busParallelSettings_t. *Note: Available for the Parallel Bus only.* AddressPin and CS.Id parameters can not be changed. If changed, values are ignored and no error is returned.
ADL_BUS_CMD_PARA_GET_ADDRESS	Gets Parallel bus base where the chip select can be addressed for one subscribed bus. **Parameters:** The Param of adl_busIoCtl is defined as a pointer to an u32. *Note: Available for the Parallel Bus only.*
ADL_BUS_CMD_PARA_GET_MAX_SETTINGS	Provides settings for the maximum IO performances. **Parameters:** The Param of adl_busIoCtl is defined as a pointer on adl_busParallelSettings_t. Only the ReadCfg, the WriteCfg and the SynchronousCfg informations are available *Note: Available for the Parallel Bus only.*
ADL_BUS_CMD_PARA_GET_MIN_SETTINGS	Provides settings for the minimum IO performances **Parameters:** The Param of adl_busIoCtl is defined as a pointer on adl_busParallelSettings_t. Only the ReadCfg, the WriteCfg and the SynchronousCfg informations are available. *Note: Available for the Parallel Bus only.*

3.11.8. Read/Write Data Structures

3.11.8.1. 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).

Code:

```
typedef struct
{
      u32         Address;
      u32         Opcode;
} adl_busAccess_t;
```

Description

Address

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 **`ADL_BUS_CMD_SET_ADD_SIZE`** IOCtl command. If less than 32 bits are required to be sent; only the most significant bits are sent on the bus.

Opcode

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 **`ADL_BUS_CMD_SET_OP_SIZE`** command. If less than 32 bits are required to be sent, only the most significant bits are sent on the bus.

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

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.

3.11.9. The adl_busSubscribe Function

This function subscribes to a specific bus, in order to write and read values to/from a remote chip.

Prototype

```
s32    adl_busSubscribe (    adl_busID_e      BusId,
                             u32              BlockId,
                             void *           BusParam );
```

Parameters

BusId:

Type of the bus to subscribe to, using the **`adl_busID_e`** type values.

BlockId:

ID of the block to use (in the range 1-N, where N is specific to each bus type & embedded module platform; cf. the i2c_NbBlocks & spi_NbBlocks & Para_NbBlocks Capabilities).

BusParam:

Subscribed bus configuration parameters, using specific parameters of the bus (considered as an **adl_busSPISettings_t ***, **an adl_busI2CSettings_t *** or an **adl_busParallelSettings_t *** pointer).

Returned values

- Handle: 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 one parameter has an incorrect value
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the bus is already open with this chip select or in configuration uncompatible with this chip select.
  - **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 embedded module on which the application is running.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *A bus is available only if the GPIO multiplexed with the corresponding feature is not yet subscribed by an Open AT® application.*

Note: *Once the bus is subscribed, the multiplexed GPIO with the required configuration are not available for subscription by the Open AT® application, or through the standard AT commands.*

3.11.10. The adl_busUnsubscribe Function

This function unsubscribes from a previously subscribed.

Prototype

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

Parameters

Handle:

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

Returned values

- **OK** on success.
- A negative error value otherwise.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** either transfer is on-going or the Bus is locked hence cannot be closed.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *If a bus is locked it can't be closed otherwise error **ADL_RET_ERR_BAD_STATE** is received. Only unlocked bus can be closed.*

3.11.11. The adl_busIOCtl Function

This function permits to modify the configuration and the behavior of a subscribed bus.

Prototype

```
s32    adl_busIOCtl ( u32                  Handle,
                      adl_busIoCtlCmd_e    Cmd,
                      void *               Param );
```

Parameters

Handle:

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

Cmd:

Command to be executed. (see adl_busIoCtlCmd_e for more information).

Param:

Parameter associated to the command. (see adl_busIoCtlCmd_e for more information).

Returned values

- **OK** on success
- A negative error value:
  - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
  - **ADL_RET_ERR_DONE** if an error occurs during the operation.
  - **ADL_RET_ERR_BAD_HDL** if the required command is not usable for the current handle.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
  - **ADL_RET_ERR_NOT_SUPPORTED** If the capabilities inform that no Asynchronous mode is possible

3.11.12. The adl_busRead Function

This function reads data from a previously subscribed bus SPI or I2C type.

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

Note: *By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.*

Prototype

```
s32    adl_busRead (  s32                  Handle,
                      adl_busAccess_t *    pAccessMode,
                      u32                  Length,
                      void *               pDataToRead );
```

Parameters

Handle:

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

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t** structure.

Length:

Number of items to read from the bus.

pDataToRead:

Buffer where to copy the read items.

Returned values

- **OK** on success if the operation is pending (asynchronous mode).
- A negative error value otherwise:
  - **ERROR** If a error during the operation occurs
  - **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_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note: *Items bit size is defined thanks to the **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command.*

Note: *In asynchronous mode, the end of the read operation will be notified to the application through an interrupt event. Please refer to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command for more information.*

Note: *For correct behaviour, any parameter passed to this command has to be global and not local variable.*

3.11.13. The adl_busReadExt Function

This function reads data from a previously subscribed bus SPI or I2C type.

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

Note: *By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.*

Prototype

```
s32    adl_busReadExt (   s32                 Handle,
                          adl_busAccess_t *   pAccessMode,
                          u32                 Length,
                          void *              pDataToRead,
                          void *              context );
```

Parameters

Handle:

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

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t** structure.

Length:

Number of items to read from the bus.

pDataToRead:

Buffer where to copy the read items.

context:

Pointer on an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

Returned values

- **OK** on success
- A negative error value otherwise:
  - **Error** If a error during the operation occurs.**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_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note: *Items bit size is defined thanks to the* ***ADL_BUS_CMD_SET_DATA_SIZE IOCtl*** *command.*

Note: *In asynchronous mode, the end of the read operation will be notified to the application through an interrupt event. Please refer to* ***ADL_BUS_CMD_SET_DATA_SIZE IOCtl*** *command for more information.*

Note: *For correct behaviour, any parameter passed to this command has to be global and not local variable.*

3.11.14. The adl_busWrite Function

This function writes on a previously subscribed SPI or I2C bus type.

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

Note: *By default the access is synchronous. This behavior can be changed with the* ***ADL_BUS_CMD_SET_ASYNC_MODE IOCtl*** *command.*

Prototype

```
s32    adl_busWrite ( s32                Handle,
                      adl_busAccess_t*   pAccessMode,
                      u32                Length,
                      void *             pDataToWrite );
```

Parameters

Handle:

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

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t structure**;

Length:

Number of items to write on the bus.

pDataToWrite:

Data buffer to write on the bus.

Returned values

- **OK** on success if the operation is pending (asynchronous mode).
- A negative error value otherwise.
  - **ERROR** If a error during the operation occurs
  - **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_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note: *Items bit size is defined thanks to the* ***ADL_BUS_CMD_SET_DATA_SIZE IOCtl*** *command.*

Note: *In asynchronous mode, the end of the write operation will be notified to the application through an interrupt event. Please refer to* ***ADL_BUS_CMD_SET_DATA_SIZE IOCtl*** *command for more information.*

Note: ***pDataToWrite*** *should point to either a global static or dynamic buffer. Stack variables should not be used to hold data to bus services. If the write is synchronous, the data buffer may be released or reused after the* ***adl_busWrite API*** *call. For asynchronous access, the application should wait for the confirmation via the interrupt event before releasing the buffer.*

3.11.15. The adl_busWriteExt Function

This function writes on a previously subscribed SPI or I2C bus type.

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

Note: *By default the access is synchronous. This behavior can be changed with the* ***ADL_BUS_CMD_SET_ASYNC_MODE IOCtl*** *command.*

Prototype

```
s32    adl_busWrite ( s32                Handle,
                      adl_busAccess_t*   pAccessMode,
                      u32                Length,
                      void *             pDataToWrite,
                      void *             context );
```

Parameters

Handle:

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

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t structure**;

Length:

Number of items to write on the bus.

pDataToWrite:

Data buffer to write on the bus.

context:

Pointer on an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

Returned values

- `OK` on success
- A negative error value otherwise.
  - `Error` If a error during the operation occurs,`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_SERVICE_LOCKED` if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note: *Items bit size is defined thanks to the **`ADL_BUS_CMD_SET_DATA_SIZE IOCtl`** command.*

Note: *In asynchronous mode, the end of the write operation will be notified to the application through an interrupt event. Please refer to **`ADL_BUS_CMD_SET_DATA_SIZE IOCtl`** command for more information.*

Note: *For correct behaviour, any parameter passed to this command has to be global and not local variable*

3.11.16. 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.

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

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 (see adl_busParallelSettings_t).

Returned values

- `OK` on success
- A negative error value otherwise.
  - `ADL_RET_ERR_UNKNOWN_HDL` if the provided handle is unknown,
  - `ADL_RET_ERR_PARAM` if a parameter has an incorrect value.

3.11.17. The adl_busDirectWrite Function

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

Warning: *This function is not protected against reentrancy by consequently several tasks may access to the same resource. A protection mechanism has to be implemented by application to share a same resource and avoid two tasks access to the same resource at the same time.*

Prototype

```
s32    adl_busDirectWrite    (  s32       Handle,
                                u32       ChipAddress,
                                u32       Length,
                                void *    pDataToWrite );
```

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.

Length:

Number of items to write on the bus.

pDataToWrite:

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

Returned values

- **OK** on success
- A negative error value otherwise.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
  - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.

3.11.18. Example

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

Complete examples of BUS service used are also available on the SDK.

```
// Global variables & constants

// SPI Subscription data
const adl_busSPISettings_t MySPIConfig =
{
    1,                          // No divider, use full 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_GPIO | 31,           // Use GPIO 31 to handle the Chip Select
                                   signal
    ADL_BUS_SPI_LOAD_UNUSED,    // LOAD signal not used
    ADL_BUS_SPI_DATA_BIDIR,     // 2 Wires configuration
    ADL_BUS_SPI_MASTER_MODE,    // Master mode
```

```
    ADL_BUS_SPI_BUSY_UNUSED     // BUSY signal not used
};

// 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
    ADL_BUS_I2C_ADDR_7_BITS,    // 7 bits address length
    ADL_BUS_I2C_MASTER_MODE     // Master mode
};

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

// Access configuration structure
adl_busAccess_t AccessConfig =
{
    0, 0    // No Opcode, No Address
};

// BUS Handles
s32 MySPIHandle, MyI2Chandle;

// 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;
    u32 AddSize=0;

    // Subscribe to the SPI1 BUS
    MySPIHandle = adl_busSubscribe ( ADL_BUS_ID_SPI, 1, &MySPIConfig );

    // Subscribe to the I2C BUS
    MyI2CHandle = adl_busSubscribe ( ADL_BUS_ID_I2C, 1, &MyI2CConfig );

    // Configure the Address length to 0 (rewrite the default value)
    adl_busIOCtl ( MySPIHandle, ADL_BUS_CMD_SET_ADD_SIZE, &AddSize );
    adl_busIOCtl ( MyI2CHandle, ADL_BUS_CMD_SET_ADD_SIZE, &AddSize );

    // 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 );

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

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

3.12. Error Management

ADL supplies Error service interface to allow the application to cause & intercept fatal errors, and also to retrieve stored back-trace logs. For the ADL standard error codes, please refer to section Error Codes.

The defined operations are:

- A subscription function (**adl_errSubscribe**) to register an error event handler
- An unsubscription function (**adl_errUnsubscribe**) to cancel this event handler registration
- An error handler callback (**adl_errHdlr_f**) to be notified each time a fatal error occurs
- An error request function (**adl_errHalt**) to cause a fatal error
- A cleaning function (**adl_errEraseAllBacktraces**) to clean the back-traces storage area
- An analysis status function (**adl_errGetAnalysisState**) to retrieve the current back-trace analysis status
- An analysis start function (**adl_errStartBacktraceAnalysis**) to start the back-trace analysis
- A retrieve function (**adl_errRetrieveNextBacktrace**) to retrieve the next back-trace buffer for the current analysis.

3.12.1. Required Header File

The header file for the error functions is:

```
adl_error.h
```

3.12.2. Enumerations

3.12.2.1. The adl_ errInternalID_e Type

This type lists the error identifiers which should be generated by ADL.

Code

```
typedef enum
{
      ADL_ERR_LEVEL_MEM     = 0x0010,
      ADL_ERR_MEM_GET       = ADL_ERR_LEVEL_MEM,
      ADL_ERR_MEM_RELEASE,
      ADL_ERR_LEVEL_FLH     = 0x0020,
      ADL_ERR_FLH_READ      = ADL_ERR_LEVEL_FLH,
      ADL_ERR_FLH_DELETE,
      ADL_ERR_LEVEL_APP     = 0x0100
} adl_errInternalID_e;
```

Description

`ADL_ERR_LEVEL_MEM:`	Base level for generated ADL memory errors.
`ADL_ERR_MEM_GET:`	The platform runs out of dynamic memory.
`ADL_ERR_MEM_RELEASE:`	Internal error on dynamic memory release operation.

Note: Internal usage only. An application has no way to produce such an error.

`ADL_ERR_LEVEL_FLH:`	Base level for generated ADL flash errors.
`ADL_ERR_FLH_READ:`	Internal error on flash object read operation.

Note: Internal usage only. An application has no way to produce such an error

`ADL_ERR_FLH_DELETE:`	Internal error on flash object deletes operation.

Note: Internal usage only. An application has no way to produce such an error

`ADL_ERR_LEVEL_APP:`	Base level for application generated errors.

3.12.2.2. The adl_errAnalysisState_e Type

This type is used to enumerate the possible states of the backtraces analysis.

Code

```
typedef enum
{
      ADL_ERR_ANALYSIS_STATE_IDLE     // No running analysis
      ADL_ERR_ANALYSIS_STATE_RUNNING // A backtrace analysis is running
} adl_errAnalysisState_e;
```

3.12.3. Error event handler

Such a call-back is called each time a fatal error is caused by the application or by ADL.

Errors which should be generated by ADL are described in the **`adl_errInternalID_e`** type.

An error is described by an identifier 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 embedded module to execute a fatal error reset with a backtrace. A backtrace is composed of the provided message, and a call stack dump taken at the function call time. It is readable by the Developer Studio (Please refer to the Developer Studio online help 2 for more information).

Prototype

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

Parameters

ErrorID

Error identifier, defined by the application or by ADL

ErrorString

Error string, defined by the application or by ADL

Returned values

- TRUE If the handler decides to let the embedded module reset
- FALSE If the handler refuses to let the embedded module reset

Note: *An error event handler is called in the same execution context than the code which has caused the error.*

Note: *If the error handler returns FALSE, the back-trace log is not registered in the embedded module non-volatile memory.*

3.12.4. 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 Firmware can not be handled by such an error handler.

Prototype

```
s8     adl_errSubscribe ( adl_errHdlr_f   ErrorHandler );
```

Parameters

ErrorHandler:

Error Handler, Error event handler, defined using the **`adl_errHdlr_f`** type

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 Interrupt handler (the function is forbidden in this context).

3.12.5. The adl_errUnsubscribe Function

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

Prototype

```
s8     adl_errUnsubscribe ( adl_errHdlr_f ErrorHandler );
```

Parameters

ErrorHandler:

Error event handler, defined using the **`adl_errHdlr_f`** type, and previously provided to **`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 Interrupt handler (the function is forbidden in this context).

3.12.6. The adl_errHalt Function

This function causes an error, defined by its ID and string. If an error handler is defined (using **`adl_errHdlr_f`** type), it will be called, otherwise a embedded module reset will occur.

When the Embedded module resets (if there is no handler, or if this one returns TRUE), a back-trace log is registered in a non-volatile memory area, and also sent to Developer Studio (if this one is running).

Such a back-trace log contains:

- the call stack dump when the error occurs
- the provided error identifier & string
- the context name which has caused the error, following the same behaviour than a trace display operation (please refer to the Debug Traces service for more information).

Prototype

```
void   adl_errHalt (   u16               ErrorID,
                       const ascii *     ErrorStr );
```

Parameters

ErrorID:

Error ID Error identifier. Shall be at least equal to **`ADL_ERR_LEVEL_APP`** (lower values are reserved for ADL internal error events)

ErrorStr:

Error string to be provided to the error handler, and to be stored in the resulting backtrace if a fatal error is required.

Note: *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 embedded module's flash memory. If the application is erased or modified, the string will not be correctly displayed when retrieving the backtraces.*

Note: *Error identifiers below **`ADL_ERR_LEVEL_APP`** are for internal purpose so the application should only use an identifier above **`ADL_ERR_LEVEL_APP`***

Note: *When the embedded module reset is due to a fatal error, the init type parameter will be set to the **`ADL_INIT_REBOOT_FROM_EXCEPTION`** value (Please refer to the Tasks Initialization Service for more information).*

3.12.7. The adl_errEraseAllBacktraces Function

Backtraces (caused by the **`adl_errHalt`** function, ADL or the Firmware) are stored in the embedded module 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 values

- **`OK`** on success.**`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.12.8. 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

- Handle A positive or null handle on success. This handle has 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 Interrupt 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.*

3.12.9. 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 analysis state, using the **adl_errAnalysisState_e** type.

3.12.10. The adl_errRetrieveNextBacktrace Function

This function allows the application to retrieve the next backtrace buffer stored in the embedded module memory. The backtrace analysis has to be 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.
- **Size**: 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 Interrupt handler (the function is forbidden in this context).

Note: *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 Developer Studio in order to decode the backtrace buffer(s). Please refer to Developer Studio online help 2 in order to know how to process these files.*

Note: *If **adl_errRetrieveNextBacktrace** is used you have to retrieve all next backtraces. Otherwise it is impossible to retrieve the first backtraces. There is no way to cancel a backtrace analysis; an analysis has always to be completed until all the backtraces are retrieved.*

3.12.11. Example

The code sample below illustrates a nominal use case of the ADL Error service public interface (error cases are not handled).

```
// Error Event handler
bool MyErrorHandler ( u16 ErrorID, ascii * ErrorStr )
{
    // Nothing to do but accept the reset
    return TRUE;
}

// Error string
const ascii * MyErrorString = "Application Generated Error";

// Error launch function
void MyFunction1 ( void )
{
    // Subscribe to error service
    adl_errSubscribe ( MyErrorHandler );

    // Cause an error
    adl_errHalt ( ADL_ERR_LEVEL_APP + 1, MyErrorString );
}

// Error service unsubscription function
void MyFunction2 ( void )
{
    // Unsubscribe from error service
    adl_errUnsubscribe ( MyErrorHandler );
}

// Backtraces analysis event handler
u8 * MyAnalysisFunction ( void )
{
    // Start analysis
```

```
    s8 AnalysisHandle = adl_errStartBacktraceAnalysis();

    // Get state
    adl_errAnalysisState_e State = adl_errGetAnalysisState();

    // Retrieve next backtrace size
    u8 * Buffer = NULL;
    u32 Size = adl_errRetrieveNextBacktrace ( AnalysisHandle, Buffer, 0 );

    // Retrieve next backtrace buffer
    Buffer = adl_memGet ( Size );
    adl_errRetrieveNextBacktrace ( AnalysisHandle, Buffer, Size );

    // Erase all backtraces
    adl_errEraseAllBacktraces();

    // Return backtrace buffer
    return Buffer;
}
```

3.13. SIM Service

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

3.13.1. Required Header File

The header file for the SIM related functions is:

```
adl_sim.h
```

3.13.2. 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_WAIT

if the argument PinCode is set to NULL

On the last three events, the service is waiting for either the external application or the adl_simEnterPIN API 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.

ADL_SIM_EVENT_NET_LOCK

The phone is locked on a network

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 either the external application or the adl_simEnterPIN API.

This argument is used only the first time the service is subscribed. It is ignored on all further subscriptions.

Returned value

- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
- **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service was already subscribed with the same handler.
- **ADL_RET_ERR_PARAM** if the function was called with a null handler.
- **OK** if the function is successfully executed.

3.13.3. 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

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

3.13.4. The adl_simGetState Function

This function gets the current SIM service state.

Prototype

```
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_NET_LOCK,   // The phone is locked on a network
        ADL_SIM_STATE_LAST
} adl_simState_e;
```

3.13.5. The adl_simEnterPIN Function

The **adl_simEnterPIN** interface enables the user to enter the Pin Code of the inserted SIM.

Prototype

```
s32 adl_simEnterPIN ( ascii *   PinCode );
```

Parameters

PinCode

a string holding the Pin Code

Returned values

- **0** if the Pin Code has been correctly processed
- **ADL_RET_ERR_PARAM** if the Pin Code is not informed
- **ADL_RET_ERR_BAD_STATE** if the SIM is not waiting for any Pin Code to be entered

Note: *The Pin Code value is not definitively saved by the ADL SIM service and it is lost after each reset.*

Note: *The ADL SIM service doesn't try to used the Pin Code provided if there is only one attempt left to entered the right PIN code.*

3.13.6. The adl_simEnterPUK Function

This interface enables the user to enter the puk code and a new pin code.

Prototype

```
s32 adl_simEnterPUK (    ascii *    PukCode,
                         ascii *    NewPinCode );
```

Parameters

PukCode

a string holding the puk Code

NewPinCode

a string holding the new pin code

Returned values

- **`OK`** ADL try the given PUK code
- **`ADL_RET_ERR_PARAM`** if the PukCode or NewPinCode is not informed
- **`ADL_RET_ERR_BAD_STATE`** if the SIM is not waiting for PIN or PUK, and nothing entered yet from ext.

3.13.7. The adl_simRemAttempt Function

This function allows to get the number of remaining attempts on PIN and PUK codes.

Prototype

```
s32 adl_simRemAttempt ( void );
```

Returned values

- **`adl_simRem_e`** structure which holds the PIN and PUK remaining attempts

  The description of **`adl_simRem_e`** structure is as follows:

```
typedef struct
{
    s8 PinRemaining; //Contains remaining attempts on PIN before lock PIN
    s8 PukRemaining; //Contains remaining attempts on PUK before lock PUK
} adl_simRem_e;
```

3.14. 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 embedded module interfaces (UART, SPI, I2C).

Note: *The Open SIM Access feature has to be enabled on the embedded module in order to make this service available.*

Note: *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).*

Note: *Please contact your Sierra Wireless distributor for more information on how to enable this feature on the embedded module.*

3.14.1. Required Header File

The header file for the OSA service definitions is:

```
adl_osa.h
```

3.14.2. The adl_osaVoltage_e type

This voltage for power up the external SIM (in bit-wise).

```
typedef enum
{
        ADL_OSA_ISO     = 0x00,
        ADL_OSA_1V8     = 0x01,
        ADL_OSA_3V      = 0x02,
        ADL_OSA_5V      = 0x04
} adl_osaVoltage_e;
```

Description

ADL_OSA_ISO:	The card can be activated at a VCC of 3V or 5V.
ADL_OSA_1V8:	The card can be activated at a VCC of 1.8V.
ADL_OSA_3V:	The card can be activated at a VCC of 3V.
ADL_OSA_5V:	The card can be activated at a VCC of 5V.

3.14.3. The adl_osaATRparam_t Structure

This structure allows the application to power up the external SIM by given voltage.

```
typedef struct
{
        adl_osaVoltage_e  voltage
} adl_osaATRparam_t;
```

3.14.4. 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 Sierra Wireless 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 3.14.6).

Returned values

- 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 embedded module
  - **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 Interrupt handler (the function is forbidden in this context).

3.14.5. The adl_osaSubscribeExt Function

This function allows the application to supply an OSA service handler and support voltage, which will then be notified on each OSA event reception.

Moreover, by calling this function, the application requests the Sierra Wireless 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,
                        adl_osaVoltage_e    SupportVoltage );
```

Parameters

OsaHandler:

OSA service handler supplied by the application.

Please refer to adl_osaHandler_f type definition for more information.

SupportVoltage:

The voltage supported by SIM card reader. Bitwise OR combination of the voltage listed in the **adl_osaVoltage_e** type.

Returned values

- 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 or voltage not listed in the `adl_osaVoltage_e` type
  - `ADL_RET_ERR_NOT_SUPPORTED` if the Open SIM access feature is not enabled on the embedded module
  - `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 Interrupt handler (the function is forbidden in this context)

3.14.6. 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 embedded module 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 Sierra Wireless 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 Sierra Wireless firmware through the `adl_osaSendResponse` function.

Event Type	Use
ADL_OSA_EVENT_APDU_REQUEST	This event is received by the application each time the Sierra Wireless 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 Sierra Wireless 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 Sierra Wireless 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 Sierra Wireless firmware). When this event is received, the OSA service is automatically un-subscribed and the Sierra Wireless firmware resumes the local SIM connection.
ADL_OSA_EVENT_CLOSED	The application will receive this event after un-subscribing from the OSA service. The Sierra Wireless firmware has resumed the local SIM connection.
ADL_OSA_EVENT_POWER_OFF_REQUEST	The application has to power off SIM card when receiving this event

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	**RequestEvent** structure set: **Length:** Size of (**adl_osaATRparam_t**) **Data:** **adl_osaATRparam_t** structure set
ADL_OSA_EVENT_APDU_REQUEST	**RequestEvent** structure set: **Length:** APDU request buffer length **Data:** APDU request data buffer address

Event Type	**Event Parameter**
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
ADL_OSA_EVENT_POWER_OFF_REQUEST	Set to NULL

3.14.7. The adl_osaSendResponse Function

This function allows the application to post back ATR or APDU responses to the Sierra Wireless 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 Sierra Wireless 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.

3.14.8. The adl_osaUnsubscribe Function

This function un-subscribes from the OSA service: the local SIM connection is resumed by the Sierra Wireless 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 Interrupt handler (the function is forbidden in this context).
- **ADL_RET_ERR_NOT_SUBSCRIBED** The OSA service is not subscribed, so it is not possible to unsubscribe it.
- **ADL_RET_ERR_BAD_STATE** Firmware is waiting for an ATR or APDU request from the simcard, and unsubscription is forbidden until the simcard's request is granted.

3.14.9. 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 Sierra Wireless 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_osaSubscribeExt ( MyOsaHandler, ADL_OSA_1V8|ADL_OSA_3V );
}
void MyFunction2 ( void )
{
    // Un-subscribes from the OSA service
    adl_osaUnsubscribe ( OsaHandle );
}
```

3.15. SMS Service

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

3.15.1. Required Header File

The header file for the SMS related functions is:

```
adl_sms.h
```

3.15.2. 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 has not been forwarded (i.e. the +CMTI indication is not generated and the SMS is not stored in the SIM memory).

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 Interrupt handler (the function is forbidden in this context).

3.15.3. The adl_smsSubscribeExt Function

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

Prototype

```
s8    adl_smsSubscribeExt   (  adl_smsHdlrExt_f    SmsHandler,
                               adl_smsCtrlHdlr_f   SmsCtrlHandler,
                               u8                  Mode );
```

Parameters

SmsHandler:

SMS handler defined using the following type:

```
typedef s32 ( * adl_smsHdlrExt_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 **ADL_SMS_FORWARD_INDICATION_AND_STORE** 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 **`ADL_SMS_FILTER_INDICATION_AND_DELETE`** if the SMS should not be forwarded. And it returns **`ADL_SMS_FILTER_INDICATION_AND_STORE`** if the SMS must be forwarded to the external application (it is then stored in SIM memory) and the external application is not notified.

If the SMS service is subscribed several times, a received SMS will be forwarded to the external application only if each of the handlers returns 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 Interrupt handler (the function is forbidden in this context).

3.15.4. 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 Interrupt handler (the function is forbidden in this context).

3.15.5. 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 Interrupt handler (the function is forbidden in this context).

3.16. 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, Interrupt handler…).

The defined operations are:

- **subscription** & **unsubscription** functions (**adl_msgSubscribe** & **adl_msgUnsubscribe**) usable to manage message reception filters.
- **reception** callbacks (**adl_msgHandler_f**) usable to receive incoming messages.
- A **sending** function (**adl_msgSend**) usable to send messages to an application task.

3.16.1. Required Header File

The header file for message-related functions is:

```
adl_msg.h
```

3.16.2. The adl_msgIdComparator_e Type

Enumeration of comparison operators, usable to define a message filter through the **adl_msgFilter_t** structure..

```
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.

3.16.3. 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;
```

Structure Fields

The structure fields are defined below:

- **`MsgIdentifier`**Mask:
  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 section adl_msgIdComparator_e).
- **`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.

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 Interrupt handler.

3.16.4. 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 Interrupt 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_SERVICE_LOCKED`** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Note: *Messages filters definition is specific to each task: the filter will apply only to incoming messages for the current task context. The associated call-back will be called in this task context when the filter conditions are fulfilled.*

3.16.5. 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.

Length:

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.

Note: *A message handler callback will be called by ADL in the execution context where it has been subscribed.*

3.16.6. 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 Interrupt handler (the function is forbidden in this context).

3.16.7. The adl_msgSend Function

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

Prototype

```
s32    adl_msgSend    (  adl_ctxID_e      DestinationTask,
                         u32              MessageIdentifier,
                         u32              Length,
                         void *           Data );
```

Parameters

DestinationTask:

Destination task to which the message is to be posted, using the adl_ctxID_e type. Only tasks identifiers are valid (it is not possible to post messages to interrupt handler contexts).

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:*

Note: *If the message is sent from the application task, the source context identifier is set to the sending task identifier.*

Note: *If the message is sent from a low level Interrupt handler, the source context identifier is set to* ***ADL_CTX_LOW_LEVEL_IRQ_HANDLER****.*

Note: *If the message is sent from a high level Interrupt handler, the source context identifier is set to* ***ADL_CTX_HIGH_LEVEL_IRQ_HANDLER****.*

Note: *If data body is provided for the message, this one will be copied in an allocated heap memory buffer. This buffer will be automatically released after the message has been notified to all the matching message reception filters.*

Note: *Beware for task 0 Message Identifier 0xFFFFFFFF and 0xFFFFFFFE are internally used by ADL.*

3.16.8. Example

The code sample below illustrates a nominal use case of the ADL Messages Service public interface (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
    0                                  // Application task 0 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
}

// Somewhere in the application code
void MyFunction ( void )
{
    // Subscribe to the message service
    MyMsgHandle = adl_msgSubscribe ( &MyFilter, MyMsgHandler );

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

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

3.17. Call Service

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

3.17.1. Required Header File

The header file for the call related functions is:

```
adl_call.h
```

3.17.2. 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 pair 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 MT[1]
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

[1] In case of Call Waiting only; please refer to the AT Commands Interface Guide for more information.

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
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

Event / Call ID	Description	Type
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
ADL_CALL_EVENT_DTR_RELEASE / 0	If the call is released by switching DTR upon ON to OFF	MO MT

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 Interrupt handler (the function is forbidden in this context).

3.17.3. 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.

3.17.4. 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 Interrupt handler (the function is forbidden in this context).

3.17.5. 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.

3.17.6. 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 Interrupt handler (the function is forbidden in this context).

3.17.7. 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.

3.17.8. 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 Interrupt handler (the function is forbidden in this context).

3.17.9. 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 Interrupt handler (the function is forbidden in this context).

3.18. GPRS Service

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

3.18.1. Required Header File

The header file for the GPRS related functions is:

```
adl_gprs.h
```

3.18.2. The adl_gprsSubscribe Function

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

Prototype

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

Parameters

GprsHandler:

GPRS handler defined using the following type:

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

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

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

Event / Call ID	Description
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
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 Interrupt handler (the function is forbidden in this context).

3.18.3. 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.

3.18.4. 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 Interrupt handler (the function is forbidden in this context).

3.18.5. 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.

3.18.6. 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 Interrupt handler (the function is forbidden in this context).

Caution: *This function must be called before opening the GPRS FCM Flows.*

3.18.7. 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.

3.18.8. 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 Interrupt handler (the function is forbidden in this context).

Caution: *CIf 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 embedded module lock.*

3.18.9. 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

3.18.10. 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 Interrupt handler (the function is forbidden in this context).

3.18.11. 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.

3.18.12. 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 );
}
```

3.19. 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 Interrupt handlers.

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

The defined operations are:

- A subscription function **`adl_semSubscribe`** to get a semaphore resource control
- An unsubscription function **`adl_semUnsubscribe`** to release a semaphore resource
- Consumption functions **`adl_semConsume`** and **`adl_semConsumeDelay`** to consume a semaphore counter
- A produce function **`adl_semProduce`** to produce a semaphore counter
- A test function **`adl_semIsConsumed`** to check a semaphore current state
- A capabilities function **`adl_semGetResourcesCount`** to retrieve the currently free semaphore resources count

3.19.1. Required Header File

The header file for the Semaphore service definitions is:

```
adl_sem.h
```

3.19.2. The adl_semGetResourcesCount Function

This function retrieves the count of currently free semaphore resources for the application usage.

Prototype

```
u32 adl_semGetResourcesCount ( void );
```

Returned values

- Free semaphore resources count.

3.19.3. 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

- **`Handle`** A positive semaphore service handle 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.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.19.4. 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 Interrupt handler (the function is forbidden in this context).

Exceptions

The following exception must be generated on this function call

- 205 If the semaphore has been consumed too many times. A semaphore can be consumed a number of times equal to its initial value + 256.

3.19.5. 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 Interrupt handler (the function is forbidden in this context).
- **`ADL_RET_ERR_BAD_STATE`** when the semaphore has not been consumed and timeout has elapsed. Even if the semaphore has not been consumed at timeout, the semaphore counter has been decreased by one. Therefore, after this code is returned, it is mandatory to call Adl_SemProduce once to get the semaphore counter to the correct value.

Exceptions

The following exception must be generated on this function call.

- 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.

3.19.6. 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 Interrupt handler (the function is forbidden in this context).

Exceptions

The following exception must be generated on this function call.

- 133 if the semaphore has been produced too many times.
  A semaphore can be produced until its inner counter reaches its initial value.

3.19.7. 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 Interrupt handler (the function is forbidden in this context).

3.19.8. The adl_semIsConsumed Function

This function allows the application to check if a semaphore is currently consumed (the internal counter value is lower than the initial value) or not (the counter value is the initial one).

Prototype

```
s32 adl_semIsConsumed ( s32    SemHandle );
```

Parameters

SemHandle:

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

Returned values

- TRUE if the semaphore resource is consumed.
- FALSE If the semaphore resource is not consumed.
- `ADL_RET_ERR_UNKNOWN_HDL` when the supplied handle is unknown.

3.19.9. 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 interrupt 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 );
}
```

3.20. 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.

3.20.1. Required Header File

The header file for the Application safe mode service is:

```
adl_safe.h
```

3.20.2. 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 Sierra Wireless Firmware.

If the Handler returns TRUE, the command will be processed by the Sierra Wireless Firmware, 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 Interrupt handler (the function is forbidden in this context).

3.20.3. 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 Sierra Wireless Firmware.

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 Interrupt handler (the function is forbidden in this context).

3.20.4. 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, ADL_SAFE_CMD_WOPEN_OTHER and ADL_SAFE_CMD_WOPEN_ERASE_OBJ** 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 interrupt handler (the function is forbidden in this context).

3.21. AT Strings Service

This service provides APIs to process AT standard response strings.

3.21.1. Required Header File

The header file for the AT strings service is:

```
adl_str.h
```

3.21.2. 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:"
        ADL_STR_LAST                           // Last string ID
} adl_strID_e;
```

3.21.3. 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.

3.21.4. 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.

3.21.5. 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.

3.21.6. 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.

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.*

3.21.7. 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.

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.*

3.22. 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 (Sierra Wireless Firmware 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 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 Sierra Wireless Firmware 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 Sierra Wireless Firmware 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. Sierra Wireless shall not be liable for any issue related to any use by customer of the Download Over The Air feature.

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

The defined operations are:

- adl_adSubscribe
- adl_adUnsubscribe
- adl_adWrite
- adl_adInfo
- adl_adGetState
- adl_adFinalise
- adl_adDelete
- adl_adInstall
- adl_adRecompact
- adl_adGetCellList
- adl_adFormat
- adl_adEventSubscribe
- adl_adEventUnsubscribe
- adl_adGetInstallResult

3.22.1. Required Header File

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

```
adl_ad.h
```

3.22.2. 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 interrupt handler (the function is forbidden in this context).

3.22.3. 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 interrupt handler (the function is forbidden in this context).

3.22.4. 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 interrupt 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.

3.22.5. 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.

Event	Meaning
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 embedded module 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.

3.22.6. 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 running with this event handle.
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.7. The adl_adWrite Function

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

Note: *On unsubscribing an AD cell and then re-subscribing the same cell any '0xFF' characters stored in the cell originally would be reassigned as free space.*

Note: *If it is required to append data to a cell from which the application was unsubscribed, it is strongly recommended to recompact the memory as further attempts of appending data will result in an error -22 (even though the resubscription is successful).*

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 exceeds the cell size.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.8. The adl_adInfo Function

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

Note: The A&D memory data cannot be read in RTE mode.

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 of an undefined size.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.9. The adl_adFinalise Function

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

Note that it also sets the limits for a cell. For instance, if a cell of undefined size is subscribed, then all A&D memory space is reserved for this cell.After writing data on this cell, it is important to finalise this cell, which will then mark the boundaries for the cell, (fix its size) and allow other cell subscriptions (if there is any cell of undefined size, which is not finalized, then it is not possible to subscribe to another cell of undefined size).

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 interrupt handler (the function is forbidden in this context).

3.22.10. The adl_adDelete Function

This function deletes the provided A&D cell. The used space 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_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 interrupt handler (the function is forbidden in this context).

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

3.22.11. 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 Sierra Wireless Firmware binary file.

Caution: *This API resets the embedded module on success.*

Prototype

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

Parameters

CellHandle:

A&D cell handle returned by **adl_adSubscribe** function.

Returned values

- Embedded module 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 embedded module 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 interrupt handler (the function is forbidden in this context).

Note: *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 A&D cell content install window).*

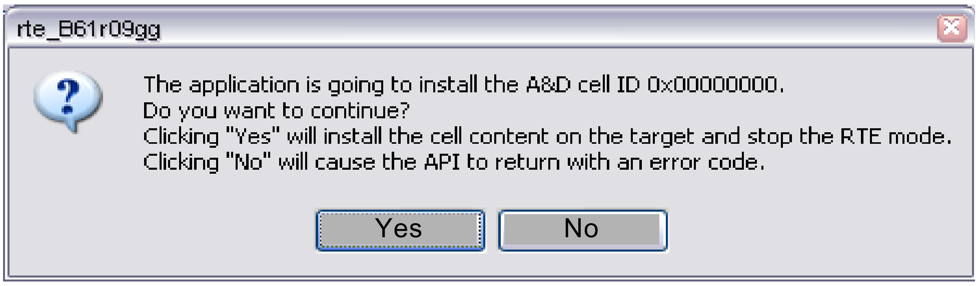


Figure 8. A&D cell content install window

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 embedded module resets, and the RTE mode is automatically closed.

3.22.12. The adl_adRecompact Function

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

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 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 interrupt handler (the function is forbidden in this context).

Note: *It is strongly recommended after Recompact process to unsubscribe (and then re-subscribe) any already subscribed cell.*

3.22.13. The adl_adGetState Function

This function provides information structure on the 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;         // Free space 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 interrupt handler (the function is forbidden in this context).

3.22.14. The adl_adGetCellList Function

This function provides the list of the 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 with the wm_lstDestroy function 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 interrupt handler (the function is forbidden in this context).

Note: *The number of elements in the returned list are limited by* ***ADL_AD_MAX_CELL_RETRIEVE;***

Note: *If the number of cell IDs to get is superior to* ***ADL_AD_MAX_CELL_RETRIEVE****, use* ***adl_adFindInit*** *and* ***adl_adFindNext*** *functions.*

3.22.15. The adl_adFormat Function

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

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 performed,
- **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 subscribed cell, or if a re-compaction or format process is running.

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

3.22.16. The adl_adFindInit Function

This function initializes a cell search between the two provided cell identifiers.

Prototype

```
s32    adl_adFindInit (  u32                 MinCellId,
                         u32                 MaxCellId,
                         adl_adBrowse_t*     BrowseInfo );
```

Parameters

MinCellId:

Minimum cell value for wanted cell identifiers.

MaxCellId:

Maximum cell value for wanted cell identifiers.

BrowseInfo:

Returned browse information, to be used with the adl_adFindNext function. Based on the following type:

```
typedef struct
{
      u32     hidden[4];    // Fields of Cell browse info have not to be
                               modified by the application
}adl_adBrowse_t;
```

Returned values

- **OK** on success.
- **ADL_AD_RET_ERR_NOT_AVAILABLE** if A&D space is not available
- **ADL_RET_ERR_PARAM** on parameter error.

3.22.17. The adl_adFindNext Function

This function performs a search on cell ID with the browse information provided by the **adl_adFindInit** function.

Prototype

```
s32    adl_adFindNext  ( adl_adBrowse_t*     BrowseInfo,
                         u32*                CellId );
```

Parameters

BrowseInfo:

Browse information.

CellId:

ID of cell found.

Returned values

- **OK** on success.
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_AD_RET_ERR_REACHED_END** no more elements to enumerate.

- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler.

3.22.18. The adl_adGetInstallResult Function

The **adl_adGetInstallResult** interface enables the user to retrieve the result of **adl_adInstall**.

Prototype

```
s32   adl_adGetInstallResult ( void );
```

Returned values

- **OK** on success
- **ADL_AD_RET_ERR_UPDATE_FAILURE** if last update failed
- **ADL_AD_RET_ERR_RECOVERY_DONE** if last update succeeded, but the OS was unstable. The system had to do a recovery
- **ADL_AD_RET_ERR_OAT_DEACTIVATED** if the Open AT® application was deactivated at start-up because of reset loops

3.22.19. The adl_factoryReadCell Function

The **adl_factoryReadCell** interface enables the user to read cell on the factory volume and get the size of cell.

Prototype

```
s32   adl_factoryReadCell  (  adl_factoryCell_e   Cell,
                              ascii*              data );
```

Parameters

Cell:

Cell to be read, based on the following information:

```
typedef enum
{
      ADL_FACTORY_CELL_SERIAL,
      ADL_FACTORY_CELL_TX,
      ADL_FACTORY_CELL_RX,
} adl_factoryCell_e;
```

data:

String read. This is an optional parameter, it could be set to NULL just to retrieve size of cell.

Returned values

- The size of the Cell on success.
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_BAD_STATE** if the factory volume is not accessible
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler.

3.23. AT/FCM IO Ports Service

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

3.23.1. Required Header File

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

```
adl_port.h
```

3.23.2. 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_RDMS_VIRTUAL_BASE         = 0x70,
       ADL_PORT_RDMS_SERVER_VIRTUAL_BASE
       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.
- 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_RDMS_VIRTUAL BASE

  Virtual Port ID for IDS service supporting the flow of internal messages (only internal use)
- ADL_PORT_RDMS_SERVER_VIRTUAL_BASE
  Virtual Port ID for IDS service supporting the flow of messages exchanged with the server (only internal use).
- 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.
- ADL_PORT_RDMS_VIRTUAL BASE

  Virtual Port ID for IDS service supporting the flow of internal messages (only internal use)
- ADL_PORT_RDMS_SERVER_VIRTUAL_BASE
  Virtual Port ID for IDS service supporting the flow of messages exchanged with the server (only internal use).

3.23.3. 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.

3.23.4. 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 are identified by the type below:

```
typedef enum
{
       ADL_PORT_EVENT_OPENED,
       ADL_PORT_EVENT_CLOSED,
       ADL_PORT_EVENT_RTS_STATE_CHANGE,
       ADL_PORT_EVENT_DTR_STATE_CHANGE
} adl_portEvent_e;
```

The events are described 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_RTS_STATE_CHANGE
  *Informs the ADL application that the specified **Port** RTS 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** RTS 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.*

- ADL_PORT_EVENT_DTR_STATE_CHANGE

  *Informs the ADL application that the specified **Port** DTR 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** DTR signal with the adl_portStartSignalPolling function.*

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 interrupt handler (the function is forbidden in this context).

3.23.5. 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 interrupt handler (the function is forbidden in this context).

3.23.6. 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.

Note: *The function will always return TRUE on the **`ADL_PORT_GSM_BASE`** port ;*

Note: *The function will always return TRUE on the **`ADL_PORT_GPRS_BASE`** port if the GPRS feature is enabled (always FALSE otherwise).*

3.23.7. 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_RTS,
      ADL_PORT_SIGNAL_DTR,
      ADL_PORT_SIGNAL_LAST
} adl_portSignal_e;
```

Signals are detailed below:

- ADL_PORT_SIGNAL_RTS
  Required port RTS 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_DTR
  Required port DTR 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.

3.23.8. 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 interrupt handler (the function is forbidden in this context).

3.23.9. 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 interrupt handler (the function is forbidden in this context).

3.24. RTC Service

ADL provides a RTC service to access to the embedded module's inner RTC, and to process time related data.

The defined operations are:

- A **adl_rtcGetTime**
- A **adl_rtcSetTime**
- A **adl_rtcConvertTime**
- A **adl_rtcDiffTime**

3.24.1. Required Header File

The header file for the RTC functions is:

```
adl_rtc.h
```

3.24.2. RTC service Types

3.24.2.1. The adl_rtcTime_t Structure

Holds a RTC time:

```
typedef struct
{
      u32     Pad0            // Not used
      u32     Pad1            // Not used
      u16     Year;           // Year (Four digits)
      u8      Month;          // Month (1-12)
      u8      Day;            // Day of the Month (1-31)
      u8      WeekDay;        // Day of the Week (1-7)
      u8      Hour;           // Hour (0-23)
      u8      Minute;         // Minute (0-59)
      u8      Second;         // Second (0-59)
      u32     SecondFracPart; // Second fractional part
      u32     Pad2;           // Not used
} adl_rtcTime_t;
```

Second fractional part (0-MAX) The MAX value is available from the registry field rtc_PreScalerMaxValue. See panel "Capabilities registry informations".

3.24.2.2. The adl_rtcTimeStamp_t Structure

Used to perform arithmetic operations on time data:

```
typedef struct
{
      u32     TimeStamp;          // Seconds elapsed since 1st January 1970
      u32     SecondFracPart;     // Second fractional part
} adl_rtcTimeStamp_t;
```

Second fractional part (0-MAX) The MAX value is available from the registry field rtc_PreScalerMaxValue. See panel "Capabilities registry informations".

3.24.2.3. Constants

RTC service constants are defined below:

Constant	Value	Use
ADL_RTC_DAY_SECONDS	24 * ADL_RTC_HOUR_SECONDS	*Seconds count in a day*
ADL_RTC_HOUR_SECONDS	60 * ADL_RTC_MINUTE_SECONDS	*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*

3.24.2.4. Macros

RTC service macros are defined below:

Macro	Parameter	Use
ADL_RTC_SECOND_FRACPART_STEP	adl_rtcGetSecondFracPartStep structure	*Second fractional part step value (in µs) extraction macro*
ADL_RTC_GET_TIMESTAMP_DAYS(_t)	(_t.TimeStamp / ADL_RTC_DAY_SECONDS) structure	*Days number extraction macro.*
ADL_RTC_GET_TIMESTAMP_HOURS(_t)	((_t.TimeStamp % ADL_RTC_DAY_SECONDS) / ADL_RTC_HOUR_SECONDS) structure	*Hours number extraction macro*
ADL_RTC_GET_TIMESTAMP_MINUTES(_t)	((_t.TimeStamp % ADL_RTC_HOUR_SECONDS) / ADL_RTC_MINUTE_SECONDS) structure	*Minutes number extraction macro*
ADL_RTC_GET_TIMESTAMP_SECONDS(_t)	(_t.TimeStamp % ADL_RTC_MINUTE_SECONDS) structure	*Seconds number extraction macro*
ADL_RTC_GET_TIMESTAMP_MS(_t)	(((u32)(_t.SecondFracPart * ADL_RTC_SECOND_FRACPART_STEP)) / ADL_RTC_MS_US) structure	*Milliseconds number extraction macro.*

Macro	Parameter	Use
ADL_RTC_GET_TIMESTAMP_US(_t)	(((u32)(_t.SecondFracPart * ADL_RTC_SECOND_FRACPART_STEP)) % ADL_RTC_MS_US) structure *Note:* *This macro returns the number of microseconds within the millisecond. For example, if the part of the timestamp is 12345 = 0.3767395 sec, the macro returns 739 or 740.*	*µseconds number extraction macro*

3.24.3. Enumerations

3.24.3.1. The adl_rtcConvert_e Type

This structure contains the available conversion modes.

Code

```
typedef enum
{
        ADL_RTC_CONVERT_TO_TIMESTAMP,
        ADL_RTC_CONVERT_FROM_TIMESTAMP
} adl_rtcConvert_e;
```

Description

`ADL_RTC_CONVERT_TO_TIMESTAMP:` Conversion mode to TimeStamp.

`ADL_RTC_CONVERT_FROM_TIMESTAMP:` Conversion mode from TimeStamp.

3.24.4. The adl_rtcGetSecondFracPartStep Function

This function retrieves the second fractional part step (in µs), reading the rtc_PreScalerMaxValue register field.

Prototype

```
float  adl_rtcGetSecondFracPartStep ( void );
```

Returned values

- The second fractional part step of the embedded module, in µs.

3.24.5. The adl_rtcGetTime Function

This function retrieves the current RTC time into an adl_rtcTime_t structure.

Prototype

```
s32    adl_rtcGetTime ( adl_rtcTime_t *   TimeStructure );
```

Parameters

TimeStructure:

RTC structure where to copy current time.

Returned values

- **OK** on success.
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler

3.24.6. The adl_rtcSetTime Function

This function sets a RTC time from a adl_rtcTime_t structure.

Prototype

```
s32    adl_rtcSetTime ( adl_rtcTime_t *   TimeStructure );
```

Parameters

TimeStructure:

RTC structure where to get current time.

Returned values

- **OK** on success.
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interruption handler

Note: *The input parameter cannot be a constant since it is modified by the API*

Note: *When setting the RTC time SecondFracPart and WeekDay field are ignored.*

3.24.7. The adl_rtcConvertTime Function

This function is able to convert RTC time structure to timestamp structure, and timestamp structure to RTC time structure thanks to a third agument precising the way of conversion.

Prototype

```
s32    adl_rtcConvertTime (  adl_rtcTime_t*      TimeStructure,
                             adl_rtcTimeStamp_t* TimeStamp,
                             adl_rtcConvert_e    Conversion );
```

Parameters

TimeStructure:

RTC structure where to get/set current time

TimeStamp:

Timestamp structure where to get/set current time

Conversion:

Conversion way:

- ADL_RTC_CONVERT_TO_TIMESTAMP
- ADL_RTC_CONVERT_FROM_TIMESTAMP

Returned values

- **`OK`** on success,
- **`ERROR`** if conversion failed (internal error),
- **`ADL_RET_ERR_PARAM`** on parameter error.

3.24.8. The adl_rtcDiffTime Function

This function reckons the difference between two timestamps.

Prototype

```
s32   adl_rtcDiffTime ( adl_rtcTimeStamp_t *   TimeStamp1,
                        adl_rtcTimeStamp_t *   TimeStamp2,
                        adl_rtcTimeStamp_t *   Result );
```

Parameters

TimeStamp1:

First timestamp to compare

TimeStamp2:

Second timestamp to compare

Result:

Reckoned time difference

Returned values

- **1** if TimeStamp1 is greater than TimeStamp2,
- **-1** if TimeStamp2 is greater than TimeStamp1,
- **0** if the provided TimeStamps are the same,
- **`ADL_RET_ERR_PARAM`** on parameter error.

3.24.9. Capabilities

ADL provides informations to get the RTC Second Frac Part capabilities.

The following entry is defined in the registry:

Registry entry	Type	Description
rtc_PreScalerMaxValue	INTEGER	0: No second fractional part xxx: Second fractional part resolution

3.24.10. Example

This example demonstrates how to use the RTC service in a nominal case (error cases are not handled) with a embedded module.

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, DiffStamp;
    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, &DiffStamp );

    //Convert the time difference from time stamps
    adl_rtcConvertTime (&Diff, &DiffStamp, ADL_RTC_CONVERT_FROM_TIMESTAMP );

    //Set back the initial time
    adl_rtcSetTime ( &Time1 );
}
```

3.25. IRQ Service

The ADL IRQ service allows interrupt handlers to be defined.

These handlers are usable with other services (External Interrupt Pins, Audio) to monitor specific interrupt sources.

Interrupt handlers are running in specific execution contexts of the application. Please refer to the Execution Context Service for more information.

The defined operations are:

- Subscription functions **adl_irqSubscribe** & **adl_irqSubscribeExt** to define interrupt handlers
- **Configuration** functions **adl_irqSetConfig** & **adl_irqGetConfig** to handle interrupt handlers configuration
- An **Unsubscription** function **adl_irqUnsubscribe** to remove an IRQ handler definition
- A **Get Capabilities** function **adl_irqGetCapabilities** to retrieve the IRQ service capabilities

Note: *The Real Time Enhancement feature has to be enabled on the embedded module in order to make this service available.*

Note: *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)*

Note: *Please contact your Sierra Wireless distributor for more information on how to enable this feature on the embedded module.*

3.25.1. Required Header File

The header file for the IRQ functions is:

```
adl_irq.h
```

3.25.2. The adl_irqID_e Type

This type defines the interrupt sources that the service is able to monitor.

```
typedef enum
{
        ADL_IRQ_ID_AUDIO_RX_LISTEN,
        ADL_IRQ_ID_AUDIO_TX_LISTEN,
        ADL_IRQ_ID_AUDIO_RX_PLAY,
        ADL_IRQ_ID_AUDIO_TX_PLAY,
        ADL_IRQ_ID_EXTINT,
        ADL_IRQ_ID_TIMER,
        ADL_IRQ_ID_EVENT_CAPTURE
        ADL_IRQ_ID_EVENT_DETECTION
        ADL_IRQ_ID_SPI_EOT,
        ADL_IRQ_ID_I2C_EOT,
        ADL_IRQ_ID_LAST         // Reserved for internal use
} adl_irqID_e;
```

The **ADL_IRQ_ID_AUDIO_RX_LISTEN** constant identifies RX path interrupt sources raised by the Audio Stream Listen service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_TX_LISTEN** constant identifies TX path interrupt sources raised by the Audio Stream Listen service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_RX_PLAY** constant identifies RX path interrupt sources raised by the Audio Stream Play service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_TX_PLAY** constant identifies TX path interrupt sources raised by the Audio Stream Play. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_EXTINT** constant identifies interrupt sources raised by the External Interrupt Pin source. For more information, please refer to the Extint ADL Service.

The **ADL_IRQ_ID_TIMER** constant identifies interrupt sources raised by the Timer Interrupts source. For more information, please refer to the TCU Service.

The **ADL_IRQ_ID_EVENT_CAPTURE** constant identifies capture interrupt sources raised by the Timer Interrupts source. For more information, please refer to the TCU Service.

The **ADL_IRQ_ID_EVENT_DETECTION** constant identifies detection interrupt sources raised by the Timer Interrupt source. For more information, please refer to the TCU Service.

The **ADL_IRQ_ID_SPI_EOT** constant identifies SPI bus asynchronous end of transmission event. Please refer to the Bus Service for more information.

The **ADL_IRQ_ID_I2C_EOT** constant identifies I2C bus asynchronous end of transmission event. Please refer to the Bus Service for more information.

3.25.3. The adl_irqNotificationLevel_e Type

This type defines the notification level of a given interrupt handler.

For more information on specific high and low level handlers behavior, please refer to the Execution Context Service description.

```
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 interrupt handlers to be defined.

The **ADL_IRQ_NOTIFY_HIGH_LEVEL** constant allows high level interrupt handlers to be defined.

3.25.4. The adl_irqPriorityLevel_e Type

This type defines the priority level of a given interrupt handler.

The lowest priority level is always 0.

The highest priority level shall be retrieved thanks to the **adl_irqGetCapabilities** function.

Please refer to each interrupt related service for more information about the available priority levels.

The priority level of a handler allows the notification order to be set in case of event conflict:

- A **N** priority level handler cannot be interrupted by other handlers with the same **N** priority level, or with a lower **N - X** priority level.
- A **N** priority level handler can be interrupted by any other handlers with an higher **N + X** priority level.

Note: *Priority levels settings are significant only for low level interrupt handlers. There is no way to define priority levels for high level interrupt handlers.*

Note: Priority levels settings are only efficient with external interrupt service, allowing to configure the several external interrupt pins priority. Other interrupt source services priorities are not configurable, and always have the values listed in the table below.
Trying to modify the priority of such services will have no effect.

Service	Events	Priority value
Audio Service	ADL_IRQ_ID_AUDIO_RX_LISTEN ADL_IRQ_ID_AUDIO_TX_LISTEN ADL_IRQ_ID_AUDIO_RX_PLAY ADL_IRQ_ID_AUDIO_TX_PLAY	Max
BUS & TCU Services	ADL_IRQ_ID_SPI_EOT ADL_IRQ_ID_I2C_EOT ADL_IRQ_ID_TIMER ADL_IRQ_ID_EVENT_CAPTURE ADL_IRQ_ID_EVENT_DETECTION	0

MAX value represents the maximum priority value.

3.25.5. The adl_irqEventData_t Structure

This structure supplies interrupt handlers with data related to the interrupt source.

```
typedef struct
{
        union
        {
              void *    LowLevelOuput;
              void *    HighLevelInput;
        }UserData;
              void *    SourceData;
              u32       Instance
              void *    Context
} adl_irqEventData_t;
```

3.25.5.1. The UserData Field

This field allows the application to exchange data between low level and high level interrupt handlers.

3.25.5.2. The Source Data Field

This field provides to handlers an interrupt source specific data. Please refer to each interrupt source related service for more information about this field data structure.

When the interrupt occurs, the source related information structure is automatically provided by the service to the low level interrupt handler, whatever if the **ADL_IRQ_OPTION_AUTO_READ** option is enabled or not.

In an high level interrupt handler, this field will be set only if the **ADL_IRQ_OPTION_AUTO_READ** option is enabled.

3.25.5.3. The Instance Field

Instance identifier of the interrupt event which has just occurred. Please refer to each interrupt source related service for more information on the instance number use.

3.25.5.4. The Context Field

Application context, given back by ADL on event occurrence. This context was provided by the application to the interrupt source related service, when using the operation which enables the interrupt event occurrences.
If the interrupt source related service does not offer a way to define an application context, this member will be set to NULL.
Please refer to each interrupt source related service for more information on the instance number use.

3.25.6. The adl_irqCapabilities_t Structure

This structure allows the application to retrieve information about the IRQ service capabilities.

```
typedef struct
{
      u8      PriorityLevelsCount,
      u8      Pad [3]             // Reserved for internal use
      u8      InstancesCount [ADL_IRQ_ID_LAST]
} adl_irqCapabilities_t;
```

3.25.6.1. The PriorityLevelsCount Field

This field provides the priority levels count, usable to set an **adl_irqPriorityLevel_e** type value (see adl_irqPriorityLevel_e)

Such a value shall use a range from 0 to **PriorityLevelsCount**–1.

3.25.6.2. The InstancesCount Field

This field provides the instances count, for each interrupt source identifier. Please refer to each interrupt source related service for more information. If an instance count value is set to 0, the corresponding interrupt related event is not supported on the current platform.

3.25.7. The adl_irqConfig_t Structure

This structure allows the application to configure interrupt handlers behaviour.

```
typedef struct
{
      adl_irqPriorityLevel_e   PriorityLevel,
      bool                     Enable,
      u8                       Pad[2]    // Reserved for future use
      adl_irqOptions_e         Options
} adl_irqConfig_t;
```

3.25.7.1. The PriorityLevel Field

This field defines the interrupt handler priority level. Please refer to the adl_irqPriorityLevel_e type definition for more information.

Note: *If different services are plugged on an interrupt handler, the priority value will be applied to all services, if possible. If the priority value is not applicable for a given service, it will be ignored.*

3.25.7.2. The Enable Field

This field defines if the interrupt handler is enabled or not.
If set to **TRUE**, the interrupt handler is enabled and any interrupt event on which is plugged this handler will call the related function.
If set to **FALSE**, the interrupt handler is disabled: all interrupt events on which are plugged this handler are masked, and will be delayed until the handler is enabled again.

Note: *This is the default behaviour. If specified in the related service, the event shall be just delayed until the handler is enabled again.*

3.25.7.3. The Options Field

This field defines the interrupt handler notification options. A bitwise OR combination of the option constants has to be used. Please refer to the adl_irqOptions_e type definition for more information.

3.25.8. The adl_irqOptions_e type

These options have to be used with a bit-wise OR in order to specify the interrupt handler behaviour.

```
typedef enum
{
        ADL_IRQ_OPTION_AUTO_READ                =1UL,
        ADL_IRQ_OPTION_PRE_ACKNOWLEDGEMENT      =0UL,
        ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT     =0UL
} adl_ adl_irqOptions_e;
```

ADL_IRQ_OPTION_AUTO_READ: Automatic interrupt source information read.

When the interrupt occurs, the source related information structure is automatically read by the service, and supplied to the low level interrupt handler.
When used with a high level interrupt handler, this option allows the application to get the source related information structure read at interrupt time.

Note: *This option has no effect with a low level interrupt handler (**adl_irqEventData_t::SourceData** field will always be provided by the related interrupt service in this case).*
***ADL_IRQ_OPTION_PRE_ACKNOWLEDGEMENT**: Interrupt source pre-acknowledgement.*
***ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT:** Interrupt source post-acknowledgement.*

3.25.9. The adl_irqHandler_f Type

This type has to be used by the application in order to provide ADL with an interrupt hander.

Prototype

```
typedef bool (*)adl_irqHandler_f (adl_irqID_e             Source,
                                  adl_irqNotificationLevel_e NotificationLevel,
                                  adl_irqEventData_t *       Data );
```

Parameter

Source:

Interrupt source identifier.

Please refer to adl_irqID_e type definition for more information.

NotificationLevel:

Interrupt handler current notification level.

Please refer to adl_irqNotificationLevel_e type definition for more information.

Data:

Interrupt handler input/output data field.

Please refer to adl_irqEventData_t type definition for more information.

Returned values

- Not relevant for high level interrupt handlers.
- For low level interrupt handlers
  - TRUE: requires ADL to call the subscribed high level handler for this interrupt source.
  - FALSE: requires ADL not to call any high level handler for this interrupt source.

Note: *For low level interrupt handlers, 1 ms can be considered as a maximum latency time before being notified with the interrupt source event.*

3.25.10. 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_irqNotificationLevel_e   NotificationLevel,
                         adl_irqPriorityLevel_e       PriorityLevel,
                         adl_irqOptions_e             Options );
```

Parameter

IrqHandler:

Interrupt handler supplied by the application.

NotificationLevel:

Interrupt handler notification level; allows the supplied handler to be identified as a low level or a high level one.

PriorityLevel:

Interrupt handler priority level; Please refer to adl_irqPriorityLevel_e type definition for more information.

Options:

Interrupt handler notification options.

A bitwise OR combination of the options constant has to be used. Please refer to the adl_irqOptions_e type definition for more information.

Returned values

- Handle: A positive or null IRQ service handle on success, to be used in further IRQ & interrupt source services function calls.
- `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.

Note: When subscribing to a high level handler, both Low Level & High Level Interrupt contexts stack sizes have to be defined, since ADL internally uses the Low level context to process events.

- **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 interrupt 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). .Use of the IRQ service should be flagged in order to make an application working correctly in RTE.

*Note: This function is a shortcut to the **adl_irqSubscribeExt** one. Provided **PriorityLevel** and **Options** parameters values will be used to fill the configuration structure. The **adl_irqConfig_t::Enable** field will be set to **TRUE** by default.*

3.25.11. The adl_irqSubscribeExt Function

This function allows the application to supply an interrupt handler, to be used later in Interrupt source related service subscription.

Prototype

```
s32 adl_irqSubscribeExt ( adl_irqHandler_f              IrqHandler,
                          adl_irqNotificationLevel_e    NotificationLevel,
                          adl_irqConfig_t*              Config );
```

Parameter

IrqHandler:

Interruption handler supplied by the application.

Please refer to adl_irqHandler_f type definition for more information.

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_irqNotificationLevel_e type definition for more information.

Config:

Interrupt handler configuration. Please refer to the adl_irqConfig_t structure definition for more information.

Returned values

- Handle: A positive or null IRQ service handle on success, to be used in further IRQ & interrupt source services function calls.
- **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.

Note: When subscribing to a high level handler, both Low Level & High Level Interrupt contexts stack sizes have to be defined, since ADL internally uses the Low level context to process events.

- **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 interrupt 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). Use of the IRQ service should be flagged in order to make an application working correctly in RTE.

3.25.12. The adl_irqUnsubscribe Function

This function allows the application to unsubscribe from the interrupt service. The associated handler will no longer be notified of interrupt events.

Prototype

```
s32 adl_irqUnsubscribe ( s32   IrqHandle );
```

Parameter

IrqHandle:

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

3.25.13. The adl_irqSetConfig function

This function allows the application to update an interrupt handler's configuration.

Prototype

```
s32 adl_irqSetConfig  (  s32                IrqHandle,
                         adl_irqConfig_t *  Config );
```

Parameter

IrqHandle:

IRQ service handle, previously returned by the **adl_irqSubscribe** function.

Config:

Interrupt handler configuration structure. Please refer to the adl_irqConfig_t structure definition for more information.

Returned values

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

3.25.14. The adl_irqGetConfig function

This function allows the application to retrieve an interrupt handler's configuration.

Prototype

```
s32 adl_irqGetConfig (  s32                IrqHandle,
                        adl_irqConfig_t *  Config );
```

Parameter

IrqHandle:

IRQ service handle, previously returned by the **`adl_irqSubscribe`** function.

Config:

Interrupt handler configuration structure. Please refer to the adl_irqConfig_t structure definition for more information.

Returned values

- **`OK`** on success.
- **`ADL_RET_ERR_UNKNOWN_HDL`** if the supplied handle is unknown.
- **`ADL_RET_ERR_PARAM`** on a supplied parameter error.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.25.15. The adl_irqGetCapabilities Function

This function allows the application to retrieve information about the IRQ service capabilities on the current platform.

Prototype

```
s32 adl_irqGetCapabilities  ( adl_irqCapabilities_t *    Capabilities );
```

Parameter

Capabilities

IRQ service capabilities information structure. Please refer to the adl_irqCapabilities_t structure definition for more information.

Returned values

- **`OK`** on success.
- **`ADL_RET_ERR_PARAM`** on parameter error.

3.25.16. Example

The code sample below illustrates a nominal use case of the ADL IRQ Service public interface (error cases are not handled).

```
// Global variable: IRQ service handle
  s32 MyIRQHandle;

  // Interrupt handler
  bool MyIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
  NotificationLevel, adl_irqEventData_t * Data )
  {
      // Interrupt process...
      // Notify the High Level handler, if any
      return TRUE;
  }

  // Somewhere in the application code, used as event handler
  void MyFunction1 ( void )
  {
      // Local variables
      adl_irqCapabilities_t Caps;
      adl_irqConfig_t Config;

      // Get capabilities
      adl_irqGetCapabilities ( &Caps );

      // Set configuration
      Config.PriorityLevel = Caps.PriorityLevelsCount - 1; // Highest priority
      Config.Enable = TRUE;              // Interrupt handler enabled
      Config.Options = ADL_IRQ_OPTION_AUTO_READ;          // Auto-read option
set

      // Subscribe to the IRQ service
      MyIRQHandle = adl_irqSubscribeExt ( MyIRQHandler,
      ADL_IRQ_NOTIFY_LOW_LEVEL, &Config );

      // TODO: Interrupt source service subscription
      ...

      // Mask the interrupt
      adl_irqGetConfig ( MyIRQHandle, &Config );
      Config.Enable = FALSE;
      adl_irqSetConfig ( MyIRQHandle, &Config );

      ...

      // Unmask the interrupt
      adl_irqGetConfig ( MyIRQHandle, &Config );
      Config.Enable = TRUE;
      adl_irqSetConfig ( MyIRQHandle, &Config );

      ...

      // TODO: Interrupt source service un-subscription
      ...

      // Un-subscribe from the IRQ service
      adl_irqUnsubscribe ( MyIRQHandle );
  }
```

3.26. TCU Service

ADL supplies Timer & Capture Unit Service interface to handle operations related to the embedded module hardware timers & capture units.

The defined operations are:

- A **subscription** function (**`adl_tcuSubscribe`**) to subscribe to the TCU service
- An **unsubscription** function (**`adl_tcuUnsubscribe`**) to unsubscribe from the TCU service
- **Start & Stop** functions (**`adl_tcuStart`** & **`adl_tcuStop`**) to control the TCU service event generation

3.26.1. Required Header File

The header file for the TCU function is:

`adl_tcu.h`

3.26.2. Capabilities Registry Informations

ADL provides capabilities information about the TCU service, thanks to the registry service.

The following entries have been defined in the registry:

Registry entry	Type	Description
tcu_TmrSrvAvailable	INTEGER	Availability of the Accurate Timer service (boolean value)
tcu_CaptSrvAvailable	INTEGER	Availability of the Event Capture service (boolean value)
tcu_DetectSrvAvailable	INTEGER	Availability of the Event Detection service (boolean value)
tcu_EvPinsNb	INTEGER	Number of pins usable to monitor events with the Capture & Detection services
tcu_TimersNb	INTEGER	Maximum number of Accurate Timer service instances which can be running at the same time
tcu_TimerBoundaries	DATA	Minimum & maximum duration values which can be used for the Accurate Timer service, using the **`adl_tcuTimerBoundaries_t`** structure format.
tcu_TimerTick	DATA	Timer resolution used by the Accurate Timer Service, using the **`adl_tcuTimerDuration_t`** structure format.
tcu_EvDetectUnit	INTEGER	Time granularity used (in µs steps) in the event detection service: for inactivity period settings (**`_adl_tcuEventDetectionSettings_t::Duration`**) for last stable state duration information (**`_adl_tcuEventDetectionInfo_t::LastStateDuration`**)
tcu_EvCaptUnit	INTEGER	Time granularity used (in µs steps) in the event capture service, for capture duration setting (**`_adl_tcuEventCaptureSettings_t::Duration`**)

3.26.3. Data Structures

3.26.3.1. The adl_tcuEventCaptureSettings_t Structure

TCU configuration structure, when the **`ADL_TCU_EVENT_CAPTURE`** service is used.

```
typedef struct
{
      u16                     CapturePinID,
      adl_tcuEventType_e      EventType,
      u32                     Duration,
      u32*                    EventCounter
} adl_tcuEventCaptureSettings_t;
```

Fields

CapturePinID:

Identifier of the pin on which the service has to monitor events.Please refer to the PTS for more information. The allowed values range is from 0 to the value returned by the tcu_EvPinsNb capability - 1.

EventType:

Event capture type, using one of the **`adl_tcuEventType_e`** type values.

Duration:

Duration of the capture period (in the unit provided by the **`tcu_EvCaptUnit`** capability). This duration is used only if the **`adl_tcuEventCaptureSettings_t::EventCounter`** address is set to **`NULL`**, otherwise it will be ignored. When the parameter is used, the related IRQ service handlers are called on each duration expiration, indicating to the application how many events have occurred since the previous handler call.

Note: *When the Event Capture is configured with a period duration greater than 0, an Accurate Timer resource is internally used to handle the service.*

See also `adl_tcuTimerDuration_t` description, for more information about the boundaries and the time resolution of a Timer resource.

EventCounter:

Address of a 32 bits variable provided by the application, where the events counter value has to be stored. If this address is provided, no interrupt events will be generated, but the event counter value will be incremented each time a new event is detected. Please note that in this case, none of IRQ service handles provided to the **`adl_tcuSubscribe`** function will be used (parameters values will be ignored). If this address is set to **`NULL`**, the service will regularly generate events, on the time base defined by the **`adl_tcuEventCaptureSettings_t::Duration`** parameter.

Note: *The provided variable address has to be accessible from the Firmware until the service is unsubscribed. This means that the variable has to be either a global/static one, or an allocated heap buffer.*

If provided, the event counter content is reset by the TCU service at each TCU service starting (including restarting) and is incremented while changes occur on the selected capture pin.

3.26.3.2. The adl_tcuEventDetectionInfo_t Structure

This structure contains the information provided to event handlers when **`ADL_IRQ_ID_EVENT_DETECTION`** events are generated, following a **`ADL_TCU_EVENT_DETECTION`** service subscription.

```
typedef struct
{
        u32                     LastStateDuration,
        adl_tcuEventType_e      EventType
} adl_tcuEventDetectionInfo_t;
```

Fields

LastStateDuration:

Duration (in the unit provided by the **`tcu_EvDetectUnit`** capability) of the last stable state of the monitored signal, before the handler notification occured.

EventType:

Type of the event which has caused the notification. If the value is positive or null, it represents the detected event type, using the **`adl_tcuEventType_e`** enumeration type. If the value is **`ADL_TCU_EVENT_TYPE_NONE`**, it means that no event has been detected since the last handler notification when the timeout programed thanks to the **`adl_tcuEventDetectionSettings_t::Duration`** parameter has elapsed.

3.26.3.3. The adl_tcuEventDetectionSettings_t Structure

TCU configuration structure, when the **`ADL_TCU_EVENT_DETECTION`** service is used.

```
typedef struct
{
       u16                     DetectionPinID,
       adl_tcuEventType_e      EventType,
       u32                     Duration
} adl_tcuEventDetectionSettings_t;
```

Fields

DetectionPinID

Identifier of the pin on which the service has to monitor events. Please refer to the Product Technical Specification for more information. The allowed values range is from 0 to the value returned by the **`tcu_EvPinsNb`** capability - 1.

EventType

Event detection type, using one of the **`adl_tcuEventType_e`** type values.

Duration

Optional inactivity detection period duration, used to cause an handler notification if no event occurred for a given time slot. If this value is set to 0, the inactivity detection will be disabled. If this value is greater than 0, it is the inactivity detection period duration (in the unit provided by the **`tcu_EvDetectUnit`** capability): if no event has occurred since the last notification (or since the **`adl_tcuStart`** function call) when the duration expires, the associated handlers will be called to warn the application about this inactivity.

Note: *When the Event Detection is configured with an inactivity period duration greater than 0, an Accurate Timer resource is internally used to handle the service.*

See also adl_tcuTimerDuration_t description, for more information about the boundaries and the time resolution of a Timer resource.

3.26.3.4. The adl_tcuTimerBoundaries_t Structure

This structure is usable to retrieve the TCU capabilities about the Accurate Timer service duration boundaries.

```
typedef struct
{
      adl_tcuTimerDuration_t    MinDuration,
      adl_tcuTimerDuration_t    MaxDuration
} adl_tcuTimerBoundaries_t;
```

Fields

MinDuration

Minimum timer duration, using the **adl_tcuTimerDuration_t** structure.

MaxDuration

Maximum timer duration, using the **adl_tcuTimerDuration_t** structure.

3.26.3.5. The adl_tcuTimerDuration_t Structure

Configuration structure usable to represent a timer duration.

Note: *Valid boundaries for a Timer duration should be retrieved from the* ***tcu_TimerBoundaries*** *capability.*

Note: *Please note that only the product of the two fields (**DurationValue * DurationUnit**) is considered for boundaries checking.*

Note: *Values of the ADL_TCU_TIMER_UNIT_XXX constants are recommended ones, but any other combination which fit with the platform capabilities is allowed. E.g. the following configuration (2ms) is allowed:* ***adl_tcuTimerDuration_t MyDuration = { 1, 2000 };***

Note: *Please note also that whatever is the configured duration, it will however be rounded down to the nearest multiple of the tick resolution, retrievable through the* ***tcu_TimerTick*** *capability.*

```
typedef struct
{
       u32         DurationValue,
       u32         DurationUnit
} adl_tcuTimerDuration_t;
```

Fields

DurationValue

Timer duration value, in the unit set by the **_adl_tcuTimerDuration_t::DurationUnit** field.

DurationUnit

Timer duration multiplier, in µs steps. For user convenience, it is advised to use defined duration unit constants (**ADL_TCU_TIMER_UNIT_US**, **ADL_TCU_TIMER_UNIT_MS** or **ADL_TCU_TIMER_UNIT_S**).

3.26.3.6. The adl_tcuTimerSettings_t Structure

TCU configuration structure, when the **ADL_TCU_ACCURATE_TIMER** service is used.

```
typedef struct
{
       adl_tcuTimerDuration_t   Duration,
       u32                      Periodic
} adl_tcuTimerSettings_t;
```

Fields

Duration

Timer duration, using the **adl_tcuTimerDuration_t** configuration structure.

Periodic

Boolean periodic timer configuration:

if set to **TRUE**, the timer is reloaded after each event occurrence.

Otherwise, the timer is stopped after the first event occurrence.

Note: *Beware if the timer is periodic and the Duration parameter is low, the handle will be called at high frequency. Hence, this handle needs to have little to do, otherwise a reset might occur.*

3.26.4. Enumerators

3.26.4.1. The adl_tcuService_e Type

This enumeration lists the available TCU services types.

Code

```
enum
{
       ADL_TCU_ACCURATE_TIMER,
       ADL_TCU_EVENT_CAPTURE,
       ADL_TCU_EVENT_DETECTION
 } adl_tcuService_e;
```

Description

ADL_TCU_ACCURATE_TIMER

Accurate timer service

Allows the application to subscribe to the accurate timer service.

Please refer to the Accurate Timers Service configuration for more information.

ADL_TCU_EVENT_CAPTURE

Event capture service.

Allows the application to subscribe to the event capture service.

Please refer to the Event Capture Service configuration for more information

ADL_TCU_EVENT_DETECTION

Event detection service.

Allows the application to subscribe to the event detection service.

Please refer to the Event Detection Service configuration for more information.

3.26.4.2. The adl_tcuEventType_e Type

This enumeration lists the available event types usable for the capture & detection services.

Code

```
enum
{
        ADL_TCU_EVENT_TYPE_NONE         = (s16)0xFFFF,// No event detected
        ADL_TCU_EVENT_TYPE_RISING_EDGE = 0,           // Capture or detect
                                                      rising edge events only
        ADL_TCU_EVENT_TYPE_FALLING_EDGE,              // Capture or detect
                                                      falling edge events only
        ADL_TCU_EVENT_TYPE_BOTH_EDGE                  // Capture or detect
                                                         events on both edges
} adl_tcuEventType_e;
```

Note: ***ADL_TCU_EVENT_TYPE_NONE*** *is only used for event detection information, as a _adl_tcuEventDetectionInfo_t::EventType parameter value.*

3.26.5. Accurate Timers Service

This service is usable to generate (periodically or not) accurate timer events, configured thanks to the **`adl_tcuTimerSettings_t`** structure (such a structure has to provided to the **`adl_tcuSubscribe`** function).

Output parameter of the **`adl_tcuStop`** function is used as an **`adl_tcuTimerDuration_t`** pointer to return the remaining time until the timer expiration when the stop operation has been performed.

Interrupt handlers defined in the IRQ service - using the **`adl_irqHandler_f`** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **`adl_tcuStart`** function is called:

- the Source parameter will be set to **`ADL_IRQ_ID_TIMER`**
- the **`adl_irqEventData_t::SourceData`** field of the Data parameter will be set to **`NULL`**.
- the **`adl_irqEventData_t::Instance`** field of the Data parameter will be set to 0.
- the **`adl_irqEventData_t::Context`** field of the Data parameter will be set to the application context, provided at subscription time.

Note: *Even though the periodic TCU timer is hardware driven, when selecting a periodic timer, the next timer start is delayed due to interrupt handler exiting the timer. In order not to stretch a periodic timer from the time period desired, it is important to spend as little time as possible within the interrupt handler, because the time spent in the handler will be added to the periodic time of the next timer.*

3.26.5.1. Example

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_ACCURATE_TIMER** mode.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Accurate timer configuration: periodic 5ms timer
adl_tcuTimerSettings_t Config = { { 5, ADL_TCU_TIMER_UNIT_MS }, TRUE };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Timer event
    if ( Source == ADL_IRQ_ID_TIMER )
    {
        // Trace event
        TRACE (( 1, "Timer event" ));
    }
      return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL, 0, 0
);

    // Subscribes to the TCU service, in Accurate Timer mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_ACCURATE_TIMER, IrqHandle, 0,
&Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
}
void MyFunction2 ( void )
{
    // Stops event generation, and gets remaining time
    adl_tcuTimerDuration_t RemainingTimer ;
    adl_tcuStop ( TCUHandle, &RemainingTimer );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.6. Event Capture Service

This service is usable to count events on a given embedded module pin, and is configured thanks to the **`adl_tcuEventCaptureSettings_t`** structure (such a structure has to provided to the **`adl_tcuSubscribe`** function).

Output parameter of the **`adl_tcuStop`** function is not used for this service, and shall be set to **`NULL.`**

Interrupt handlers defined in the IRQ service - using the **`adl_irqHandler_f`** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **`adl_tcuStart`** function is called:

- the Source parameter will be set to **`ADL_IRQ_ID_EVENT_CAPTURE`**
- the **`adl_irqEventData_t::SourceData`** field of the Data parameter will have to be casted as an u32 value, indicating the number of events which have occured since the last event handler call.

  The notification period is configured by the

  `adl_tcuEventCaptureSettings_t::Duration` parameter.
- the **`adl_irqEventData_t::Instance`** field of the Data parameter will be set to the monitored pin identifier, required at subscription time in the **`adl_tcuEventCaptureSettings_t::CapturePinID`**.
- the **`adl_irqEventData_t::Context`** field of the Data parameter will be set to the application context, provided at subscription time.

3.26.6.1. Example (without handler notification)

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_EVENT_CAPTURE** mode, without handler notification.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// Event counter to be provided to the API
u32 MyEventCounter;

// TCU Event capture configuration: on pin 0, count falling edges, with a
provided event counter
adl_tcuEventCaptureSettings_t Config = { 0, ADL_TCU_EVENT_TYPE_FALLING_EDGE, 0,
&MyEventCounter };

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the TCU service, in Event Capture mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_CAPTURE, 0, 0, &Config, NULL
);

    // Reset counter to 0, and starts event generation
    MyEventCounter = 0;
    adl_tcuStart ( TCUHandle );
}

void MyFunction2 ( void )
{
    // Periodically monitor the events counter, whenever in the application's
life
    TRACE (( 1, "Current events count: %d", MyEventCounter ));
}

void MyFunction3 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.6.2. Example (with handler notification)

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **`ADL_TCU_EVENT_CAPTURE`** mode, with handler notification.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Event capture configuration: on pin 0, counts rising edge events, and
notify the handler every second
adl_tcuEventCaptureSettings_t Config = { 0, ADL_TCU_EVENT_TYPE_RISING_EDGE, 8,
NULL };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Event Capture
    if ( Source == ADL_IRQ_ID_EVENT_CAPTURE )
    {
        // Check for pin identifier
        if ( Data->Instance == 0 )
        {
            // Get Source Data
            u32 SourceData = ( u32 ) Data->SourceData;

            // Trace event count
            TRACE (( 1, "%d events capture since last notification", SourceData
));
        }
    }

    return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL, 0,
ADL_IRQ_OPTION_AUTO_READ );

    // Subscribes to the TCU service, in Event Capture mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_CAPTURE, IrqHandle, 0,
&Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
}

void MyFunction2 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.7. Event Detection Service

This service is usable to detect events on a given embedded module pin, and is configured thanks to the **`adl_tcuEventDetectionSettings_t`** structure (such a structure has to provided to the **`adl_tcuSubscribe`** function.

Output parameter of the **`adl_tcuStop`** function is not used for this service, and shall be set to **`NULL.`**

Interrupt handlers defined in the IRQ service - using the **`adl_irqHandler_f`** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **`adl_tcuStart`** function is called.

- the Source parameter will be set to **`ADL_IRQ_ID_EVENT_DETECTION`**
- the **`adl_irqEventData_t::SourceData`** field of the Data parameter will have to be casted as a pointer on an **`adl_tcuEventDetectionInfo_t`** structure.
- the **`adl_irqEventData_t::Instance`** field of the Data parameter will be set to the monitored pin identifier, required at subscription time in the **`adl_tcuEventDetectionSettings_t::DetectionPinID.`**
- the **`adl_irqEventData_t::Context`** field of the Data parameter will be set to the application context, provided at subscription time.

3.26.7.1. Example

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **`ADL_TCU_EVENT_DETECTION`** mode.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Event detection configuration: on pin 0, detects rising edge events, and
set a 200 ms timeout
adl_tcuEventDetectionSettings_t Config = { 0, ADL_TCU_EVENT_TYPE_RISING_EDGE,
200 };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Event Detection
    if ( Source == ADL_IRQ_ID_EVENT_DETECTION )
    {
        // Check for pin identifier
        if ( Data->Instance == 0 )
        {
            // Get Source Data
            adl_tcuEventDetectionInfo_t * SourceData =
           ( adl_tcuEventDetectionInfo_t * ) Data->SourceData;

            // Check for true or inactivity event
            if ( SourceData->EventType < 0 )
            {
                // Trace inactivity
                TRACE (( 1, "Event detection timeout" ));
            }
```

```
            else
            {
                // Trace event detection
                TRACE (( 1, "%d event detected; last state duration: %d ms",
                SourceData->EventType, SourceData->LastStateDuration ));
            }
        }
    }

    return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL,
                                   0, ADL_IRQ_OPTION_AUTO_READ );

    // Subscribes to the TCU service, in Event Detection mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_DETECTION, IrqHandle, 0,
                                   &Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
}

void MyFunction2 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.8. The adl_tcuSubscribe Function

This function allows the application to subscribe to the TCU service.

Prototype

```
s32 adl_tcuSubscribe (  adl_tcuService_e    SrvID,
                        s32                 LowLevelIrqHandle,
                        s32                 HighLevelIrqHandle,
                        void *              Settings,
                        void *              Context );
```

Parameters

SrvID:

Service type to be subscribed, using the **`adl_tcuService_e`** type.

LowLevelIrqHandle:

Low level interrupt handler identifier, previously returned by the **`adl_irqSubscribe`** function. This parameter is optional if the **HighLevelIrqHandle** parameter is supplied..

HighLevelIrqHandle:

High level interrupt handler identifier, previously returned by the **`adl_irqSubscribe`** function. This parameter is optional if the **LowLevelIrqHandle** parameter is supplied..

Settings:

TCU service configuration, to be defined according to the SrvID parameter value (Please refer to adl_tcuService_e type for more information).

Context:

Pointer on an application context, which will be provided back to the application when the related TCU events will occur.

Returned values

- Handle: A positive TCU service handle on success, to be used in further TCU service function calls.
- **`ADL_RET_ERR_PARAM`** on a supplied parameter error.
- **`ADL_RET_ERR_ALREADY_SUBSCRIBED`** if the service was already subscribed for this configuration. Only for **`ADL_TCU_EVENT_CAPTURE`** & **`ADL_TCU_EVENT_DETECTION`** service types.
- **`ADL_RET_ERR_NO_MORE_HANDLES`** if there are no more available internal resources for the required service. Only for **`ADL_TCU_ACCURATE_TIMER`** service type; cf. **`tcu_TimersNb`** capability.
- **`ADL_RET_ERR_BAD_HDL`** if one or both supplied interrupt handler identifiers are invalid.
- **`ADL_RET_ERR_BAD_STATE`** If the function was called in RTE mode (The TCU service is not available in RTE mode).
- **`ADL_RET_ERR_NOT_SUPPORTED`** If the required service is not supported on the current plateform.
- **`ADL_RET_ERR_SERVICE_LOCKED`** If the function was called from a low level interrupt handler (the function is forbidden in this context.

Note: *In some configuration cases, both **`LowLevelIrqHandle & HighLevelIrqHandle`** parameters are optional. Please refer to adl_tcuEventCaptureSettings_t **`::EventCounter`** description for more information.*

Note: *Whatever is the configuration, events are generated only after a call to the **`adl_tcuStart`** function.*

3.26.9. The adl_tcuUnsubscribe Function

This function allows the application to unsubscribe from the TCU service.

Prototype

```
s32 adl_tcuUnsubscribe   ( s32    Handle );
```

Parameters

Handle:

TCU service handle, previously returned by the **`adl_tcuSubscribe`** function.

Returned values

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

Note: *If the service was started thanks to the adl_tcuStart function, an unsubscription operation will implicitely stop it, without having to call the adl_tcuStop function.*

3.26.10. The adl_tcuStart Function

This function allows the application to start the TCU service event generation. Once started, the related interrupt events are generated, according to the service configuration.
Please refer to the adl_tcuService_e type for more information.

Prototype

```
s32 adl_tcuStart  ( s32  Handle );
```

Parameters

Handle:

TCU service handle, previously returned by the **adl_tcuSubscribe** function.

Returned values

- **OK** on success.
- **ADL_RET__ERR_UNKNOWN_HDL** if the supplied TCU handle is unknown.
- **ADL_RET_ERR_SERVICE_LOCKED** If the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *If the service was already started, using this function will start it again by reprograming the events generation.*

3.26.11. The adl_tcuStop Function

This function allows the application to stop the TCU service event generation. Once stopped, the related interrupt events not are generated anymore.
The function has no effect and returns **OK** if the service is already stopped.

Prototype

```
s32 adl_tcuStop ( s32                       Handle,
                  adl_tcuTimerDuration_t*   OutParam );
```

Parameters

Handle:

TCU service handle, previously returned by the **adl_tcuSubscribe** function.

OutParam:

Output parameter of the stop operation, depending on the service type. Please refer to adl_tcuService_e type for more information on this parameter usage.

This parameter should either be set to a **adl_tcuTimerDuration_t*** type or **NULL**.

Returned values

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

3.27. Extint ADL Service

The ADL External Interrupt (ExtInt) service allows the application to handle embedded module External Interrupt pin configuration & interruptions.

External interrupt pins are multiplexed with the embedded module GPIO, please refer to the embedded module Product Technical Specification for more information.

The global External Interrupt pin operation is described below:

- The interruption is generated either on:
  - the falling or the rising edge of the input signal, or both.
  - the low or high level of the input signal (currently not supported).
- 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) e.g. EXTINT is the input signal, extint_ch is the generated interruption. When the debounce period equals 4, the embedded module waits for a stable signal during 4 cycles before generating the interruption.

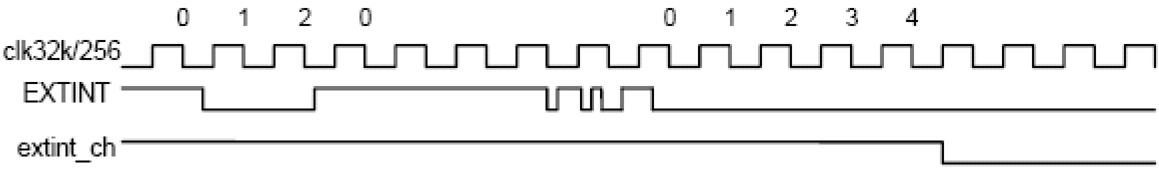


Figure 9. ADL External Interrupt service: Example of Interruption with Debounce Period

  - Stretching (the signal is stretched in order to detect even small glitches in the signal)

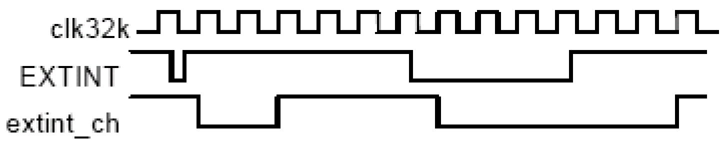


Figure 10. ADL External Interrupt service: Example of Interruption with Stretching Process

  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 Interrupt pin is always pre-acknowledged, whatever is the subscribed option in the IRQ service.

The ADL supplies interface to handle External Interruptions.

The defined operations are:

- A function adl_extintGetCapabilities to retrieve the External Interruption capablities informations.
- A function adl_extintSubscribe to subscribe to the External Interruption service.
- A function adl_extintConfig to modify an external interruption pin configuration.
- A function adl_extintGetConfig to get an external interruption pin configuration.
- A function adl_extintRead to retrieve the external interruption pin input status.
- A function adl_extintUnsubscribe to unsubscribe from the External Interruption service.
- A function adl_extintSetFIQStatus to set the FIQ status
- A function adl_extintGetFIQStatus to get the FIQ status

3.27.1. Required Header File

The header file for the ExtInt service definitions is:

```
adl_extint.h
```

3.27.2. The adl_extintID_e

This type defines the external interruption pin. Using **adl_extintGetCapabilities** to know the valid value of **adl_extintID_e**. Valid values range start from 0 to **adl_extintCapabilities_t::NbExternalInterrupt - 1**.

```
typedef u8 adl_extintID_e;
```

3.27.3. The adl_extintConfig_t Structure

This structure allows the application to configure external interrupt pin behavior. Using **adl_extintGetCapabilities** to know the available external interruption settings of the embedded module.

```
typedef struct
{
      adl_extintSensitivity_e     Sensitivity;
      adl_extintFilter_e          Filter;
      u8                          FilterDuration;
      u8                          Pad;          // Internal use only
      void *                      Context
} adl_extintConfig_t;
```

Fields

Sensitivity:

Interruption generation sensitivity, using the following type:

```
typedef enum
{
      ADL_EXTINT_SENSITIVITY_RISING EDGE,    // Rising edge (edge
                                             sensitivity) interruption
      ADL_EXTINT_SENSITIVITY_FALLING_EDGE,   // Falling edge (edge
                                             sensitivity) interruption
      ADL_EXTINT_SENSITIVITY_BOTH_EDGE,      // Rising & Falling edges (edge
                                             sensitivity)interruption.
                                             ADL_EXTINT_FILTER_STRETCHING_MODE
                                             cannot be used  with this mode.
      ADL_EXTINT_SENSITIVITY_LOW LEVEL       // Low level (level sensitivity)
                                             interruption (currently not
                                             supported). No Filter can be used
                                             with this mode,
                                             adl_extintConfig_t::Filter value
                                             must be equal to
                                             ADL_EXTINT_FILTER_BYPASS_MODE
      ADL_EXTINT_SENSITIVITY_HIGH LEVEL      // High level(level sensitivity)
                                             interruption(currently not
                                             supported). No Filter can be used
                                             with this mode,
                                             adl_extintConfig_t::Filter value
                                             must be equal to
                                             ADL_EXTINT_FILTER_BYPASS_MODE
      ADL_EXTINT_SENSITIVITY_LAST            // Internal use only
} adl_extintSensitivity_e;
```

Filter:

Filter process applied to the input signal:

```
typedef enum
{
      ADL_EXTINT_FILTER_BYPASS_MODE,     // No filter. It is the bypass mode
      ADL_EXTINT_FILTER_DEBOUNCE_MODE,   // Debounce filter.
                                         adl_extintConfig_t::
                                         FilterDuration value must not
                                         be equal to zero.
      ADL_EXTINT_FILTER_STRETCHING_MODE, // Stretching filter.
                                         adl_extintConfig_t::
                                         FilterDuration value must be
                                         equal to zero.
      ADL_EXTINT_FILTER_LAST             // Internal use only
} adl_extintFilter_e;
```

FilterDuration:

Time (in number of steps) during which the signal must be stable before generating the interruption. Refers to the function **`adl_extintGetCapabilities,`** to know the values allowed range.

This parameter is used only with the following filter:

- **`ADL_EXTINT_FILTER_DEBOUNCE_MODE`**.

Context:

Application context pointer, which will be given back to the application when an interruption event occurs.

3.27.4. The adl_extintExtConfig_e

This enumerator allows the application to configure some extended configuration for an external interrupt. This enumerator is used in the adl_extintSetConfigExt and adl_extintGetConfigExt APIs. These APIs should be used after calling the adl_extintSubscribe, adl_extintConfig APIs (these APIs do not take into account of the extended configuration):

```
typedef enum
{
   ADL_EXTINT_EXTCONFIG_ONE_SHOT_MODE,     // One shot mode: When the One
                                           Shot Mode is enabled, the
                                           External Interrupt will occur
                                           only one time. In order to
                                           reactivate the interrupt, the
                                           application should call under
                                           task (and not in the interrupt
                                           low level handler) the
                                           adl_irqGetConfig API, set the
                                           Enable field to TRUE and call
                                           the adl_irqSetConfig API.
                                           If this extended configuration
                                           is not set using the
                                           adl_extintSetConfigExt API,
                                           the default value for this
                                           extended configuration is
                                           FALSE.
                                           To activate this extended
                                           mode, the Value parameter in
                                           the adl_extintSetConfigExt API
                                           should be set to TRUE.
                                           To deactivate this extended
                                           mode, the Value parameter in
                                           the adl_extintSetConfigExt API
                                           should be set to FALSE.

   ADL_EXTINT_EXTCONFIG_LAST,              // Internal use only
 } adl_extintExtConfig_e;
```

3.27.5. The adl_extintInfo_t Structure

This structure allows the application to get the external interrupt pin input status at any time. When an interrupt handler is plugged on the ExtInt service, the SourceData field in the **`adl_irqEventData_t`** input parameter of this handler must be cast to * **`adl_extintInfo_t`** type in order to handle the information correctly.

```
typedef struct
{
      u8   PinState;
} adl_extintInfo_t;
```

Fields

PinState:

External Interrupt pin input status. Current state (0/1) of the input signal plugged on the external interrupt pin.

3.27.6. Capabilities

ADL provides informations to get EXTINT capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
extint_NbExternalInterrupt	INTEGER	Number of external interrupt pins
extint_RisingEdgeSensitivity	INTEGER	Rising edge sensitivity supported
extint_FallingEdgeSensitivity	INTEGER	Falling edge sensitivity supported
extint_BothEdgeSensitivity	INTEGER	Both edge detector supported
extint_LowLevelSensitivity	INTEGER	Low level sensitivity not supported
extint_HighLevelSensitivity	INTEGER	High level sensitivity not supported
extint_BypassMode	INTEGER	Bypass mode supported
extint_StretchingMode	INTEGER	Stretching mode supported
extint_DebounceMode	INTEGER	Debounce mode supported
extint_MaxDebounceDuration	INTEGER	Debounce max duration in ms
extint_DebounceNbStep	INTEGER	Number of step for debounce duration
extint_NbPriority	INTEGER	Available priority levels for the EXTINT service (to be used as a **adl_irqPriorityLevel_e** value in the IRQ service)

3.27.6.1. The adl_extintCapabilities_t type

This structure allows the application to read external interruptioncapabilities.

```
typedef struct
{
        u8                          NbExternalInterrupt;
        bool                        RisingEdgeSensitivity;
        bool                        FallingEdgeSensitivity;
        bool                        BothEdgeSensitivity;
        bool                        LowLevelSensitivity;
        bool                        HighLevelSensitivity
        bool                        BypassMode
        bool                        StretchingMode
        bool                        DebounceMode
        u8                          MaxDebounceDuration
        u8                          DebounceNbStep
        u8                          PriorityLevelsCount
        u8                          Pad [3]
} adl_extintCapabilities_t
```

Fields

NbExternalInterrupt:

Number of external interruption

RisingEdgeSensitivity:

Rising edge sensitivity supported

FallingEdgeSensitivity:

Falling edge sensitivity supported

BothEdgeSensitivity:

Both edge detector supported

LowLevelSensitivity:

Low level sensitivity not supported

HighLevelSensitivity:

High level sensitivity not supported

BypassMode:

Bypass mode supported

StretchingMode:

Stretching mode supported

DebounceMode:

Debounce mode supported

MaxDebounceDuration:

Debounce max duration in ms

DebounceNbStep:

Number of step for debounce duration

PriorityLevelsCount:

Available priority levels for the EXTINT service (to be used as a adl_irqPriorityLevel_e value in the IRQ service).

Pad [3]:

Internal use

3.27.6.2. The adl_extintGetCapabilities Function

This function returns the embedded module External Interruption capabilities. Capabilities are the same for all available pins on the embedded module.

Prototype

```
s32 adl_extintGetCapabilities ( adl_extintCapabilities_t *
                                    PinCapabilities )
```

Parameters

PinCapabilities

Returned External Interruption capabilities (using adl_extintCapabilities_t).

Returned values

- **OK** on success
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value

Example

This example demonstrates how to use the function **adl_extintGetCapabilities** in a nominal case (error cases not handled).

Complete examples using the External Interruption service are also available on the SDK (generic Signal Replica).

```
void My_extintGetcapabilities ( )
    {
        ascii * My_Message = adl_memGet ( 1000 );

        adl_extintCapabilities_t My_WCPU_ExtInt_Capabilities;

        adl_extintGetCapabilities ( &My_WCPU_ExtInt_Capabilities );

        wm_sprintf ( My_Message,
                    "\r\nMy WCPU have %d Ext. Int.\r\n
                    supported sensitivity :\r\n
                    RisingEdgeSensitivity %d\r\n
                    FallingEdgeSensitivity %d\r\n
                    BothEdgeSensitivity %d\r\n
                    LowLevelSensitivity %d\r\n
                    HighLevelSensitivity %d\r\n
                    supported filter :\r\n
                    Bypass %d\r\n
                    Stretching %d\r\n
                    Debounce %d\r\n
                    filter options :\r\n
                    MaxDebounceDuration %d ms in %d steps\r\n",
                    My_WCPU_ExtInt_Capabilities.NbExternalInterrupt ,
                    My_WCPU_ExtInt_Capabilities.RisingEdgeSensitivity ,
                    My_WCPU_ExtInt_Capabilities.FallingEdgeSensitivity ,
                    My_WCPU_ExtInt_Capabilities.BothEdgeSensitivity ,
                    My_WCPU_ExtInt_Capabilities.LowLevelSensitivity ,
                    My_WCPU_ExtInt_Capabilities.HighLevelSensitivity ,
                    My_WCPU_ExtInt_Capabilities.BypassMode ,
                    My_WCPU_ExtInt_Capabilities.StretchingMode ,
                    My_WCPU_ExtInt_Capabilities.DebounceMode ,
                    My_WCPU_ExtInt_Capabilities.MaxDebounceDuration ,
                    My_WCPU_ExtInt_Capabilities.DebounceNbStep
                    ) );
        adl_atSendResponse ( ADL_AT_UNS, My_Message );
        adl_memRelease ( My_Message );

    }
```

3.27.7. The adl_extintSubscribe Function

This function allows the application to subscribe to the ExtInt service. Each External Interrupt 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).

Interrupt handlers defined in the IRQ service - using the **adl_irqHandler_f** type - are notified with the following parameters:

- the **Source** parameter will be set to **ADL_IRQ_ID_EXTINT**

- the **`adl_irqEventData_t::SourceData`** field of the Data parameter has to be casted to an **`adl_extintInfo_t *`** type, usable to retrieve information about the current external interrupt pin state.
- the **`adl_irqEventData_t`**::Instance field of the **`Data`** parameter will have to be considered as an **`adl_extintID_e`** value, usable to identify which block has raised the current interrupt event.
- the **`adl_irqEventData_t::Context`** field of the Data parameter will be set to the application context, provided at subscription time.

Prototype

```
s32 adl_extintSubscribe (   adl_extintID_e         ExtIntID,
                            s32                    LowLevelIrqHandle,
                            s32                    HighLevelIrqHandle,
                            adl_extintConfig_t *   Settings );
```

Parameters

ExtIntID:

External interrupt pin identifier to be subscribed. (see section adl_extintID_e).

LowLevelIrqHandle:

Low level interrupt handler identifier, previously returned by the **`adl_irqSubscribe`** function.

This parameter is optional if the **`HighLevelIrqHandle`** parameter is supplied.

HighLevelIrqHandle:

High level interrupt handler identifier, previously returned by the **`adl_irqSubscribe`** function.

This parameter is optional if the **`LowLevelIrqHandle`** parameter is supplied.

Settings:

External interrupt pin configuration, (see section adl_extintConfig_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`** if one parameter has an incorrect value
  - **`ADL_RET_ERR_NOT_SUPPORTED`** if one parameter refers to a mode or a configuration not supported by the embedded module
  - **`ADL_RET_ERR_ALREADY_SUBSCRIBED`** if the service was already subscribed for this external interrupt pin (the External Interrupt service can only be subscribed one time for each pin).
  - **`ADL_RET_ERR_BAD_HDL`** if one or both supplied interrupt handler identifiers are invalid.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *When interrupt event generated by the EXTINT service are masked (thanks to* ***`adl_irqConfig_t::Enable`*** *field configuration of the IRQ service), events are just delayed until the related handler is enabled again.*

3.27.8. The adl_extintConfig Function

This function allows the application to modify an external interrupt pin configuration.

Prototype

```
s32 adl_extintConfig (  s32                    ExtIntHandle,
                        adl_extintConfig_t *   Settings );
```

Parameters

ExtIntHandle:

External Interrupt service handle, previously returned by the **adl_extintSubscribe** function.

Settings:

External interrupt pin configuration, (see section adl_extintConfig_t structure).

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
  - **ADL_RET_ERR_NOT_SUPPORTED** if one parameter refers to a mode or a configuration not supported by the embedded module
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown.

3.27.9. The adl_extintGetConfig Function

This function allows the application to get an external interrupt pin configuration.

Prototype

```
s32 adl_extintGetConfig (  s32                    ExtIntHandle,
                           adl_extintConfig_t *   Settings );
```

Parameters

ExtIntHandle:

External Interrupt service handle, previously returned by the **adl_extintSubscribe** function.

Settings:

External interrupt pin configuration, (see section adl_extintConfig_t structure).

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown

3.27.10. The adl_extintSetConfigExt Function

This function allows the application to set an extended configuration for an external interruption pin.

Prototype

```
s32 adl_extintSetConfigExt  (  s32                      ExtIntHandle,
                               adl_extintExtConfig_e    ExtConfig,
                               u32                      Value );
```

Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the adl_extintSubscribe function.

ExtConfig:

Extended configuration (see adl_extintExtConfig_e)

Value:

Extended configuration value

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown
  - **ADL_RET_ERR_NOT_SUPPORTED** if the API is not supported by the Sierra Wireless stack

3.27.11. The adl_extintGetConfigExt Function

This function allows the application to get an extended configuration for an external interruption pin.

Prototype

```
s32 adl_extintGetConfigExt  (  s32                      ExtIntHandle,
                               adl_extintExtConfig_e    ExtConfig,
                               u32*                     Value );
```

Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the adl_extintSubscribe function.

ExtConfig:

Extended configuration (see adl_extintExtConfig_e)

Value:

Extended configuration value

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown
  - **ADL_RET_ERR_NOT_SUPPORTED** if the API is not supported by the Sierra Wireless stack

3.27.12. The adl_extintRead function

This function allows the application to retrieve the external interrupt pin input status.

Prototype

```
s32 adl_extintRead ( s32                ExtIntHandle,
                     adl_extintInfo_t *  Info );
```

Parameters

ExtIntHandle:

External Interrupt service handle, previously returned by the **adl_extintSubscribe** function.

Info:

External interrupt pin information structure (see section adl_extintInfo_t type).

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied ExtInt handle is unknown.

3.27.13. The adl_extintUnsubscribe Function

This function allows the application to unsubscribe from the ExtInt service. Associated interrupt handlers are unplugged from the External Interruption source. Pin configuration control is resumed by the AT+WIPC command.

Prototype

```
s32 adl_extintUnsubscribe ( s32       ExtIntHandle );
```

Parameters

ExtIntHandle:

External Interrupt service handle, previously returned by the **adl_extintSubscribe** function.

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.27.14. The adl_extintSetFIQStatus function

This function sets the FIQ status. TRUE - Enables / FALSE - Disables the fast mode for the external interrupt specified by the provided handler.

Prototype

```
s32 adl_extintSetFIQStatus  (  s32       ExtIntHandle,
                               bool      Status );
```

Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

Status:

FIQ Status to be set.

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interruption handle is unknown
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the FIQ status is tried to be set on more than one handle

3.27.15. The adl_extintGetFIQStatus function

This function gets the FIQ status. Check if the fast mode for the external interrupt (specified by the provided handler) is enabled or not.

Prototype

```
s32 adl_extintGetFIQStatus  (  s32       ExtIntHandle,
                               bool *    Status );
```

Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

Status:

FIQ Status to be retrieved.

Returned values

- A **OK** on success.
- A negative error value otherwise:
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interruption handle is unknown

3.27.16. Example

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

Complete example using the External Interrupt service are also available on the SDK (generic Signal Replica sample).

```
// Global variables

    // use the PIN0 for the Ext Int
    #define EXTINT_PIN0 0

    // ExtInt service handle
    s32 ExtIntHandle;
```

```
// IRQ service handle
s32 IrqHandle;

// ExtInt configuration: both edge detection without filter
adl_extintConfig_t extintConfig =
{ ADL_EXTINT_SENSITIVITY_BOTH_EDGE, ADL_EXTINT_FILTER_BYPASS_MODE, 0, 0,
  NULL };


// ExtInt interruption handler
bool MyExtIntHandler ( adl_irqID_e Source,
                       adl_irqNotificationLevel_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 )
{
    adl_extintCapabilities_t My_ExtInt_Capa;

    adl_extintGetCapabilities ( &My_ExtInt_Capa );

    // Test if the WCPU have Ext Int pin
    if ( My_ExtInt_Capa.NbExternalInterrupt >= 1 )
    {
        // Subscribes to the IRQ service
        IrqHandle = adl_irqSubscribe ( MyExtIntHandler,
      ADL_IRQ_NOTIFY_LOW_LEVEL, ADL_IRQ_PRIORITY_HIGH_LEVEL,
      ADL_IRQ_OPTION_AUTO_READ );

        // Configures comparator channel
        ExtIntHandle = adl_extintSubscribe ( EXTINT_PIN0 , IrqHandle, 0,
         &extintConfig );

        if( ExtIntHandle > 0 )
        {
            s32 OneShotMode = 0;
            s32 s32Result = adl_extintGetConfigExt( ExtIntHandle,
            ADL_EXTINT_EXTCONFIG_ONE_SHOT_MODE, &OneShotMode);
            // Set the EXT INT in one shot mode
            if( !OneShotMode )
            {
                OneShotMode = TRUE;
                s32Result = adl_extintSetConfigExt( ExtIntHandle,
               ADL_EXTINT_EXTCONFIG_ONE_SHOT_MODE, OneShotMode );
            }
        }
    }
}
void MyFunction2 ( void )
{
    // Un-subscribes from the ExtInt service
    adl_extintUnsubscribe ( ExtIntHandle );
}
```

3.28. Execution Context Service

ADL supplies the Execution Context Service interface to handle operations related to the several execution contexts available for an Open AT® application.
The application runs under several execution contexts, according to the monitored event (ADL service event, or interrupt event).

The execution contexts are:

- **The application task context**;

  This is the main application context, initialized on the task entry point functions, 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 global low priority and should be interrupted by the other ones.

- **The high level interrupt handler context**;

  This is also a task context, but with a higher priority that the main application task. High level interrupt handlers run in this context.

  This context has a global middle priority: when an interrupt 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 interrupt handlers.

- **The low level interrupt handler context**;

  This is a context designed to be activated as soon as possible on an interrupt event.

  This context has a global high priority: when an interrupt 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 SIERRA WIRELESS Firmware 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. Semaphore ADL Service), and/or by temporary disabling interrupts (cf. IRQ Service).
The ADL services are all re-entrant.

Data can be exchanged between contexts through a message system (cf. Message Service).
However, the RAM area is global and accessible from all contexts.

The defined operations of the Execution Context service are:

- Current context identification functions (**`adl_ctxGetID`** & **`adl_ctxGetTaskID`**) to retrieve the current context identifiers.
- A Tasks count function (**`adl_ctxGetTasksCount`**) to retrieve the current tasks count in the runing application.
- A Diagnostic function (**`adl_ctxGetDiagnostic`**) to retrieve information about the current contexts configuration.
- A State function (**`adl_ctxGetState`**) to retrieve the required execution context's current state.
- Suspend functions (**`adl_ctxSuspend & adl_ctxSuspendExt`**) to suspend at any time a running application task.
- Resume functions (**`adl_ctxResume & adl_ctxResumeExt`**) to resume at any time a suspended application task.
- A Sleep function (**`adl_ctxSleep`**) to put the current context to sleep for a required duration.

3.28.1. Required Header File

The header file for the Execution Context function is:

```
adl_ctx.h
```

3.28.2. The adl_ctxID_e Type

This type defines the execution context identifiers. Low or High level interrupt handlers, and Sierra Wireless Firmware tasks are identified by specific contants. Application tasks are identified by values between **0** and the **adl_ctxGetTasksCount** function return.

```
typedef enum
{
        ADL_CTX_LOW_LEVEL_IRQ_HANDLER      = 0xFD, //Low level interrupt handler
                                                     context
        ADL_CTX_HIGH_LEVEL_IRQ_HANDLER     = 0xFE, // High level interrupt
                                                     handler context
        ADL_CTX_ALL                        = 0xFF, // Reserved for internal use
        ADL_CTX_WAVECOM                    = 0xFF, // Sierra Wireless Firmware
                                                     tasks context
} adl_ctxID_e;
```

3.28.3. The adl_ctxDiagnostic_e Type

This type defines the available diagnostics, to be retrieved by the **adl_ctxGetDiagnostic** function.

```
typedef enum
{
        ADL_CTX_DIAG_NO_IRQ_PROCESSING            = 0x01,
        ADL_CTX_DIAG_BAD_IRQ_PARAM                = 0x02,
        ADL_CTX_DIAG_NO_HIGH_LEVEL_IRQ_HANDLER    = 0x04,
} adl_ctxDiagnostic_e;
```

Description

ADL_CTX_DIAG_NO_IRQ_PROCESSING:	The Open AT® IRQ processing mechanism has not been started (interrupt 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:	High level interrupt handlers are not supported (high level handler stack size is not supplied).

3.28.4. The adl_ctxState_e Type

This type defines the various states for a given execution context, to be retrieved by the **adl_ctxGetState** function.

```
typedef enum
{
        ADL_CTX_STATE_ACTIVE
        ADL_CTX_STATE_WAIT_EVENT
        ADL_CTX_STATE_WAIT_SEMAPHORE
        ADL_CTX_STATE_WAIT_INNER_EVENT
        ADL_CTX_STATE_SLEEPING
        ADL_CTX_STATE_READY
        ADL_CTX_STATE_PREEMPTED
        ADL_CTX_STATE_SUSPENDED
} adl_ctxState_e;
```

Description

ADL_CTX_STATE_ACTIVE:	The context is currently active (the current code is executed in this context).
ADL_CTX_STATE_WAIT_EVENT:	The context is currently waiting for events (there are currently no events to process).
ADL_CTX_STATE_WAIT_SEMAPHORE:	The context is currently waiting for a semaphore to be produced. The code execution is currently frozen on a semaphore consumption function. This can be either an applicative semaphore, or an internal one, consumed within an ADL function call.
ADL_CTX_STATE_WAIT_INNER_EVENT:	The context is currently waiting for an internal event. The code execution is currently frozen, waiting for an internal event within an ADL function call.
ADL_CTX_STATE_SLEEPING:	The context is currently sleeping, after a call to **adl_ctxSleep** function.
ADL_CTX_STATE_READY:	The context has events to process, but is not currently processing them yet, since an higher priority context is processing events.
ADL_CTX_STATE_PREEMPTED:	The context has been pre-empted while it was processing events. It will resume its processing as soon as the higher priority context which is currently running will have terminated his own processing.
ADL_CTX_STATE_SUSPENDED:	The task context is currently suspended, thanks to a call to the adl_ctxSuspend function.

3.28.5. 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

- Current application's execution context identifier. Please refer to adl_ctxID_e for more information.
- **ID** An application task's zero-based index 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 interrupt handler.
- **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER** if the function is called from a high level interrupt handler.

3.28.6. The adl_ctxGetTaskID Function

This function allows the application to retrieve the current running task identifier:

- In Open AT® task or high level interrupt handler contexts, this function will behave like the **adl_ctxGetID** function.
- In a low level handler execution context, the retrieved identifier will be the active task identifier when the interrupt signal is raised.

Prototype

```
adl_ctxID_e adl_ctxGetTaskID ( void );
```

Returned values

- Current task's execution context identifier. Please refer to adl_ctxID_e for more information.
- **ID** An application task's zero-based index 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 interrupt handler.
- **Interrupted TaskID** If called from a low level interrupt handler, the returned value depends on the interrupted task:
  - An application task's zero-based index, if an Open AT® application task was running.
  - **ADL_CTX_SIERRAWIRELESS** if a Sierra Wireless Firmware task was running.
  - **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER** if a high level interrupt handler was running.

3.28.7. The adl_ctxGetTasksCount Function

This function allows the application to retrieve the current application's tasks count.

Prototype

```
u8 adl_ctxGetTasksCount ( void );
```

Returned value

- Current application's tasks count.

3.28.8. 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 value

- Bitwise OR combination of the diagnostics listed in the **adl_ctxDiagnostic_e** type.

3.28.9. The adl_ctxGetState Function

This function allows the application to retrieve the current state of the required execution context.

Prototype

```
s32  adl_ctxGetState ( adl_ctxID_e    Context );
```

Parameters

Context:

Execution context from which the current state has to be queried.

Returned values

- On success, returns the (positive or null) current execution context state, using the **adl_ctxState_e** type.
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_BAD_HDL** If the low level interrupt handler execution context state is required.

Note: *It is not possible to query the current state of the contexts below (**ADL_RET_ERR_BAD_HDL** error will be returned):*

Note: *the low level interrupt handler execution context (in any case)*

Note: *the high level interrupt handler execution context, if the related **adl_InitIRQHighLevelStackSize** call stack has not be declared in the application.*

3.28.10. The adl_ctxSuspend Function

This function allows the application to suspend an application task process. This process can be resumed later thanks to the **adl_ctxResume** function, which should be called from interrupt handlers or from any other application task.

Prototype

```
s32 adl_ctxSuspend ( adl_ctxID_e      Task );
```

Parameters

Task:

Task identifier to be suspended.

Valid values are in the **0** - **adl_ctxGetTasksCount** range.

Returned values

- **OK** on success:
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_BAD_STATE** if the required task is already suspended.

Note: *If the function was called in the application task context, it will not return but just suspend the task.*

Note: *The **OK** value will be returned when the task process is resumed.*

Note: *While a task 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 embedded module to reset (an application task should not be suspended for a long time, if it is assumed to continue to receive messages).*

Note: *When task 0 is suspended, embedded module will not respond to any AT commands coming from external ports.*

3.28.11. The adl_ctxSuspendExt Function

This function allows the application to suspend several application tasks processes. Theses process can be resumed later thanks to the **adl_ctxResume** or **adl_ctxResumeExt** functions, which should be called from interrupt handlers or from any other application task.

Prototype

```
s32 adl_ctxSuspendExt    (  u32             TasksCount,
                            adl_ctxID_e*    TasksIDArray );
```

Parameters

TasksCount:

Size of the **TasksIDArray** array parameter (number of tasks to be suspended).

TasksIDArray:

Array containing the identifiers of the tasks to be suspended. Valid values are in the **0 - adl_ctxGetTasksCount** range.

Returned values

- **OK** on success:
- **ADL_RET_ERR_PARAM** on parameter error (no task will be suspended).
- **ADL_RET_ERR_BAD_STATE** if the required task is already suspended (no task will be suspended).

Note: *If the function was called in the application task context, it will not return but just suspend the task.*

Note: *The **OK** value will be returned when the task process is resumed.*

Note: *While a task 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 embedded module to reset (an application task should not be suspended for a long time, if it is assumed to continue to receive messages).*

3.28.12. 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.

Valid values are in the **0** - **adl_ctxGetTasksCount** range.

Returned values

- **OK** on success:
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_BAD_STATE** If the required task is not currently suspended.

Note: *The required task is resumed as soon as the function is called.*

Note: *If the resumed task has a lower priority level than the current one, it will be scheduled as soon as the current task process will be over.*

Note: *If the resumed task has a higher priority level than the current one, it will be scheduled as soon as the function is called.*

3.28.13. The adl_ctxResumeExt Function

This function allows the application to resume several Open AT® tasks processes, previously suspended with to the **adl_ctxSuspend** or **adl_ctxSuspendExt** functions.

Prototype

```
s32 adl_ctxResumeExt (  u32             TasksCount,
                        adl_ctxID_e*    TasksIDArray );
```

Parameters

TasksCount:

Size of the **TasksIDArray** array parameter (number of tasks to be suspended).

TasksIDArray:

Array containing the identifiers of the tasks to be suspended. Valid values are in the **0** - **adl_ctxGetTasksCount** range.

Returned values

- **OK** on success:
- **ADL_RET_ERR_PARAM** on parameter error.
- **ADL_RET_ERR_BAD_STATE** If the required task is not currently suspended (no task will be resumed).

Note: *The required task is resumed as soon as the function is called.*

Note: *If the resumed task has a lower priority level than the current one, it will be scheduled as soon as the current task process will be over.*

Note: *If some resumed task have an higher priority level than the current one, it will be scheduled as soon as the function is called.*

3.28.14. The adl_ctxSleep Function

This function allows the application to put the current execution context to sleep for the required duration. This context processing is frozen during this time, allowing other contexts to continue their processing. When the sleep duration expires, the context is resumed and continues its processing.

Prototype

```
s32 adl_ctxSleep ( u32    Duration );
```

Parameters

Duration:

Required sleep duration, in ticks number (18.5 ms granularity).

Returned values

- `OK` on success (when the function returns, the sleep duration has already elapsed).
- `ADL_RET_ERR_SERVICE_LOCKED` If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.28.15. Example

The code sample below illustrates a nominal use case of the ADL Execution Context Service public interface (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_ctxGetDiagnostic();

    // Get the application tasks count
    u8 TasksCount = adl_ctxGetTasksCount();

    // 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();
    }
    else
    {
        // Get the current task state
        adl_ctxState_e State = adl_ctxGetState ( CurCtx );
    }
}


// Somewhere in the application code, used within an high level interrupt
handler
void MyIRQFunction ( void )
{
    // Suspend the first application task
    adl_ctxSuspend ( 0 );

    // Resume the first application task
    adl_ctxResume ( 0 );

    // Put to sleep for some time...
    adl_ctxSleep ( 10 );
}
```

3.29. ADL VariSpeed Service

The ADL VariSpeed service allows the embedded module clock frequency to be controlled, in order to temporarily increase application performance.

Note: *The Real Time Enhancement feature must be enabled on the embedded module in order to make this service available.*

Note: *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).

Note: *Please contact your Sierra Wireless distributor for more information on how to enable this feature on the embedded module.*

3.29.1. Required Header File

The header file for the VariSpeed service is:

```
adl_vs.h
```

3.29.2. 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 embedded module enter a specific boost mode, where the CPU clock frequency is set to its maximum value.

Caution: *In boost mode, the embedded module power consumption increases significantly.*
For more information, refer to the Embedded module Power Consumption Mode documentation.

The CPU clock frequencies of the available modes are listed below:

Modes	CPU Clock Frequency
..._STANDARD	26 MHz
..._BOOST	104 MHz"

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

3.29.4. 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.

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 interrupt handler (the function is forbidden in this context).

3.29.5. 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 interrupt handler (the function is forbidden in this context).

3.29.6. 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 );
}
```

3.30. ADL DAC Service

The Digital Analog Converter service offers to the customer entities the ability to convert a digital value code of a certain resolution into an analog signal level voltage.

The defined operations are:

- A function **adl_dacSubscribe** to set the reserved DAC parameters.
- A function **adl_dacUnsubscribe** to un-subscribes from a previously allocated DAC handle.
- A function **adl_dacWrite** to allow a DACs to be write from a previously allocated handle.
- A function **adl_dacAnalogWrite** to allow a DAC to be write from a previously allocated handle.
- A function **adl_dacRead** to allow a DAC to be read from a previously allocated handle.
- A function **adl_dacAnalogRead** to allow a DAC to be read from a previously allocated handle.

3.30.1. Required Header File

The header file for the functions dealing with the DAC interface is:

```
adl_dac.h
```

3.30.2. Data Structure

3.30.2.1. The adl_dacParam_t Structure

DAC channel initialization parameters.

Code

```
typedef struct
{
      u32     InitialValue
}adl_dacparam_t
```

Description

InitialValue

Raw value to set in the register of the DAC.

3.30.3. Defines

3.30.3.1. ADL_DAC_CHANNEL_1

Former constant used to identify the first DAC channel.

```
#define ADL_DAC_CHANNEL_1   0
```

3.30.4. Enumerations

3.30.4.1. The adl_dacType_e

Definition of DAC type.

Code

```
typedef enum
{
      ADL_DAC_TYPE_GEN_PURPOSE    // General Purpose DAC
} adl_dacType_e
```

3.30.5. The adl_dacSubscribe Function

This function subscribes to a DAC channel.

Prototype

```
s32 adl_dacSubscribe (   u32                  Channel,
                         adl_dacParam_t *     DacConfig );
```

Parameters

Channel:

DAC channel identifier.

DacConfig

DAC subscription configuration (using **adl_dacParam_t**).

Returned values

- A positive or null value on success:
  - DAC handle to be used on further DAC API functions calls.
- A negative error value otherwise (**No DAC is reserved**):
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the required channel has already been subscribed.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler.
  - **ADL_RET_ERR_NOT_SUPPORTED** if the current embedded module does not support the DAC service.

3.30.6. The adl_dacUnsubscribe Function

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

Prototype

```
s32 adl_dacUnsubscribe ( s32   DacHandle );
```

Parameters

DacHandle:

Handle previously returned by **`adl_dacSubscribe`** function.

Returned values

- **`OK`** on success
- A negative error value otherwise:
  - **`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 interrupt handler.

3.30.7. The adl_dacWrite Function

This function writes the digital value on DACs previously allocated.

Prototype

```
s32 adl_dacWrite  (  s32       DacHandle,
                     u32       DacWrite );
```

Parameters

DacHandle:

Handle previously returned by **`adl_dacSubscribe`** function.

DacWrite

New DAC settings to set.

Returned values

- **`OK`** on success
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`ADL_RET_ERR_UNKNOWN_HDL`** if the handle is unknown
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler and the DAC used cannot be called under interrupt context.

3.30.8. The adl_dacAnalogWrite Function

This function writes a analog value in mV on a DAC previously allocated.

Prototype

```
s32 adl_dacAnalogWrite   (  s32       DacHandle,
                            s32       DacWritemV );
```

Parameters

DacHandle:

Handle previously returned by `adl_dacSubscribe` function.

DacWritemV

New DAC settings to set (in mV).

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value.
  - `ADL_RET_ERR_UNKNOWN_HDL` if the handle is unknown
  - `ADL_RET_ERR_SERVICE_LOCKED` if the function was called from a low level interrupt handler and the DAC used cannot be called under interrupt context.

3.30.9. The adl_dacRead Function

This function reads the last written value on a DAC.

Prototype

```
s32 adl_dacRead   (  s32    DacHandle,
                     u32*   DacRead );
```

Parameters

DacHandle:

Handle previously returned by `adl_dacSubscribe` function.

DacRead

DAC digital value.

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value.
  - `ADL_RET_ERR_UNKNOWN_HDL` if the handle is unknown

3.30.10. The adl_dacAnalogRead Function

This function reads the last written value on a DAC.

Prototype

```
s32 adl_dacAnalogRead   (  s32    DacHandle,
                           s32*   DacReadmV );
```

Parameters

DacHandle:

Handle previously returned by `adl_dacSubscribe` function.

DacReadmV

DAC analog value in mV.

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value.
  - `ADL_RET_ERR_UNKNOWN_HDL` if the handle is unknown.

3.30.11. Capabilities

ADL provides informations to get DAC capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
dac_NbBlocks	INTEGER	The number of DAC blocks available
dac_xx_DigitInitValue	INTEGER	Digital value at DAC resource allocation. dac_xx_DigitInitValue is set at -1 if the default value is unknown.
dac_xx_MaxRefVoltage	INTEGER	Reference voltage of the DAC output when the maximal digital value is set.
dac_xx_MinRefVoltage	INTEGER	Reference voltage of the DAC output when the minimal digital value is set.
dac_xx_Resolution	INTEGER	DAC resolution in steps.
dac_xx_DacType	INTEGER	DAC type, see section adl_dacType_e.
dac_xx_InterruptContextUsed	INTEGER	This value is set to 1 if DAC write operations can be called under interrupt context

Note: *For the registry entry the **xx** part must be replaced by the number of the instance.*
*Example: if you want the Resolution capabilities of the DAC02 block, the registry entry to use will be **dac_02_Resolution**.*

Note: *DACs will be identified with a number as 0, 1, 2, ... dac_NbBlocks-1.*

Note: *For each block, the settling time capabilities are defined in the PTS.*

3.30.12. Example

The sample DAC illustrates a nominal use case of the ADL DAC Service public interface.

```
// Global variable
    s32 MyDACHandle;
    u32 MyDACID = 1;

    …

    // 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 ( MyDACID , &InitStruct );

        // Write a value on the DAC block
        adl_dacWrite ( MyDACHandle, 80 );

        ...

        // Write another value on the DAC block
        adl_dacWrite ( MyDACHandle, 190 );

        ...

        // Write a analog value on the DAC block (1500 mV)
        adl_dacAnalogWrite ( MyDACHandle, 1500 );

        ...

        {
            s32 AnalogValue;
            // Read the last analog value write on the DAC block
            adl_dacAnalogRead ( MyDACHandle , &AnalogValue );

            ...
         }

         ...

        {
            u32 Value;
            // Read the last register value write on the DAC block
            adl_dacRead ( MyDACHandle , &Value );

            ...
         }

        // Unsubscribe from the DAC service
        adl_dacUnsubscribe ( MyDACHandle );
    }
```

3.31. ADL ADC Service

The goal of the ADC service is to offer all the interfaces to handle application using ADC for voltage level measurement such as temperature and battery level monitoring purposes. The ADC interface provides also a way to get analog value from various sources. The ADC is a circuit section that converts low frequency analog signals, like battery voltage or temperature, to digital value.

The defined operations are:

- A function **`adl_adcRead`** to read a ADC register value.
- A function **`adl_adcAnalogRead`** to read a ADC analog value in mV.

3.31.1. Required Header File

The header file for the functions dealing with the ADC interface is:

```
adl_adc.h
```

3.31.2. The adl_adcRead Function

This function allows ADCs to be read. For this operation, it is not necessary to subscribe to ADC previously.

Prototype

```
s32 adl_adcRead   (  u32    ChannelID,
                     u32*   AdcRawValue );
```

Parameters

ChannelID:

Channel ID of the ADC to read.

AdcRawValue

The value of the ADC register.

Returned values

- A **`OK`** on success (read values are updated in the **`AdcRawValue`** parameter)
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler and the ADC used cannot be called under interrupt context.

3.31.3. The adl_adcAnalogRead Function

This function allows ADCs to be read. For this operation, it is not necessary to subscribe to ADC previously.

Prototype

```
s32 adl_adcAnalogRead   (  u32    ChannelID,
                           s32*   AdcValuemV );
```

Parameters

ChannelID:

Channel ID of the ADC to read.

AdcValuemV

The value corresponding to the register Value of the ADC voltage in mV.

Returned values

- A **`OK`** on success (read values are updated in the **`AdcValuemV`** parameter)
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if one parameter has an incorrect value.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler and the ADC used can not be called under interrupt context.

3.31.4. Capabilities

ADL provides informations to get ADC capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
adc_NbBlocks	INTEGER	The number of ADC blocks available
adc_xx_ResolutionsBits	INTEGER	To get on how many bits, is coded the result.
adc_xx_ MaxInputRange	INTEGER	The minimum input voltage in mV supported by each ADC.
adc_xx_ MinInputRange	INTEGER	The maximum input voltage in mV supported by each ADC.
adc_xx_InterruptContextUsed	INTEGER	This value is set to 1, if ADC read functions can be called under interrupt context

Note: *For the registry entry the **xx** part must be replaced by the number of the instance.*
Example: *if you want the Resolution Bits capabilities of the ADC02 block the registry entry to use will be **adc_02_ResolutionBits**.*

Note: *ADCs will be identified with a number as 0, 1, 2, ... adc_NbBlocks-1.*

Note: *For each block, the sampling time capability is defined in the PTS.*

3.31.5. Example

The code sample below illustrates a nominal use case of the ADL ADC Service public interface (error cases are not handled).

```
// ADC read functions

    // Read ADC Raw Value
    u32 My_adcReadRawValue ( u32 My_adcID )
    {
        // Variable to store ADC voltage information
        u32 My_adcValue;

        // Read the ADC
        adl_adcRead ( My_adcID , &My_adcValue );

        return ( My_adcValue );
    }

    // Read ADC value in mV
    u32 My_adcReadValue ( u32 My_adcID )
    {
        // Variable to store ADC voltage information
        s32 My_adcValue_mV;

        // Read the ADC
        adl_adcAnalogRead ( My_adcID , &My_adcValue_mV );

        return ( My_adcValue_mV );
    }
```

3.32. ADL Queue Service

ADL supplies this interface to provide to applications thread-safe queue service facilities, usable from any execution context.

The defined operations are:

- A subscription function **`adl_queueSubscribe`** to create a queue resource.
- An unsubscription function **`adl_queueUnsubscribe`** to delete a queue resource.
- A state query function **`adl_queueIsEmpty`** to check if it remains items in the queue.
- item handling functions **`adl_queuePushItem`** & **`adl_queuePopItem`** to queue and de-queue items.

3.32.1. Required Header File

The header file for the functions dealing with the Queue interface is:

```
adl_queue.h
```

3.32.2. The adl_queueOptions_e Type

This type allows to define the behaviour of a queue resource.

```
typedef enum
{
      ADL_QUEUE_OPT_FIFO,
      ADL_QUEUE_OPT_LIFO,
      ADL_QUEUE_OPT_LAST   //Reserved for internal use
} adl_queueOptions_e;
```

Description

`ADL_QUEUE_OPT_FIFO:`	First In, First Out: the first pushed item will be retrieved first.
`ADL_QUEUE_OPT_LIFO:`	Last In, First Out: the last pushed item will be retrieved first.

3.32.3. The adl_queueSubscribe Function

This function allows the application to create a thread-safe queue resource. The obtained handle is then usable with the other service operations.

Prototype

```
s32 adl_queueSubscribe ( adl_queueOptions_e     Option);
```

Parameter

Option

Allows to configure the behaviour of the queue resource, using one of the **`adl_queueOptions_e`** type values.

Returned values

- **`Handle`** A positive queue service handle on success.
- **`ADL_RET_ERR_PARAM`** on parameter error.
- **`ADL_RET_ERR_SERVICE_LOCKED`** If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.32.4. The adl_queueUnsubscribe Function

This function allows the application to release a previously subscribed queue resource, if this one is empty.

Prototype

```
s32 adl_queueUnsubscribe ( s32  Handle );
```

Parameters

Handle:

A queue service handle, previously returned by the **`adl_queueSubscribe`** function.

Returned values

- **`OK`** on success
- **`ADL_RET_ERR_BAD_STATE`** If the provided queue resource is not empty; it shall be firstly emptied thanks to the **`adl_queuePopItem`** function.
- **`ADL_RET_ERR_UNKNOWN_HDL`** If the provided handle is invalid
- **`ADL_RET_ERR_SERVICE_LOCKED`** If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.32.5. The adl_ queueIsEmpty Function

This function informs the application, if items remain in the provided queue.

Prototype

```
s32 adl_queueIsEmpty ( s32  Handle );
```

Parameters

Handle:

A queue service handle, previously returned by the **`adl_queueSubscribe`** function.

Returned values

- **`FALSE`** If it remains at least one item in the queue
- **`TRUE`** If the queue is empty.
- **`ADL_RET_ERR_UNKNOWN_HDL`** If the provided handle is invalid.

3.32.6. The adl_ queuePushItem Function

This function allows the application to add an item at the end of the provided queue resource.

Prototype

```
s32 adl_queuePushItem (  s32       Handle,
                         void*     Item );
```

Parameters

Handle:

A queue service handle, previously returned by the **`adl_queueSubscribe`** function.

Item

Pointer on the application item; this parameter cannot be **`NULL`**

Returned values

- **`OK`** on success.
- **`ADL_RET_ERR_UNKNOWN_HDL`** If the provided handle is invalid.
- **`ADL_RET_ERR_PARAM`** on parameter error (Bad item pointer).

Exceptions

- **`144:`** Raised if too many items are pushed in the queue.

Note: *This function is thread-safe, and shall be called from any execution context.*
This means that operations on queue items are performed under a critical section, in which the current context cannot be pre-empted by any other context.

3.32.7. The adl_ queuePopItem Function

This function allows the application to retrieve an item from the provided queue resource, according to the defined behaviour at subscription time (cf. **`adl_queueSubscribe`** function):

- If the queue option is **`ADL_QUEUE_OPT_FIFO`**, the first pushed item is retrieved by the function
- If the queue option is **`ADL_QUEUE_OPT_LIFO`**, the last pushed item is retrieved by the function.

Prototype

```
void* adl_queuePopItem ( s32   Handle );
```

Parameters

Handle:

A queue service handle, previously returned by the **`adl_queueSubscribe`** function.

Returned values

- **`Item`** on success, a pointer on the de-queued item.
- **`NULL`** If the provided handle is unknown, or if the related queue is empty.

Note: *This function is thread-safe, and shall be called from any execution context.*

Note: *This means that operations on queue items are performed under a critical section, in which the current context cannot be pre-empted by any other context.*

3.32.8. Example

The code sample below illustrates a nominal use case of the ADL Queue service public interface (error cases are not handled).

```
// Event handler, somewhere in the application
void MyFunction ( void )
{
    // Queue handle
    s32 MyHandle;

    // Queue state
    s32 State;

    // Item definitions
    u32 MyItem1, MyItem2, *GotItem1, *GotItem2;

    // Create a FIFO queue resource
    MyHandle = adl_queueSubscribe(ADL_QUEUE_OPT_FIFO);

    // Check the queue state (shall be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Push items
    adl_queuePushItem ( MyHandle, &MyItem1 );
    adl_queuePushItem ( MyHandle, &MyItem2 );

    // Check the queue state (shall not be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Pop items (retrieved in FIFO order)
    GotItem1 = adl_queuePopItem ( MyHandle );
    GotItem2 = adl_queuePopItem ( MyHandle );

    // Check the queue state (shall be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Delete the queue resource
    adl_queueUnsubscribe ( MyHandle );
}
```

3.33. ADL Audio Service

The ADL Audio Service allows to handle audio resources, and play or listen supported audio formats on these resources (single/dual tones, DTMF tones, melodies, PCM audio streams, decoded DTMF streams).

The defined operations are:

- An **`adl_audioSubscribe`** function to subscribe to an audio resource.
- An **`adl_audioUnsubscribe`** function to unsubscribe from an audio resource.
- An **`adl_audioTonePlay`** function to play a single/dual tone.
- An **`adl_audioDTMFPlay`** function to play a DTMF tone.
- An **`adl_audioMelodyPlay`** function to play a melody.
- An **`adl_audioTonePlayExt`** function to play a single/dual tone (extension).
- An **`adl_audioDTMFPlayExt`** function to play a DTMF tone (extension).
- An **`adl_audioMelodyPlayExt`** function to play a melody (extension).
- An **`adl_audioStreamPlay`** function to play an audio stream.
- An **`adl_audioStreamListen`** function to listen to an audio stream.
- An **`adl_audioStop`** function to stop playing or listening.
- An **`adl_audioSetOption`** function to set audio options.
- An **`adl_audioGetOption`** function to get audio options

3.33.1. Required Header File

The header file for the functions dealing with the Audio service interface is:

```
adl_audio.h
```

3.33.2. Data Structures

3.33.2.1. The adl_audioDecodedDtmf_u Union

This union defines different types of buffers which are used according to the decoding mode (Raw mode enable or disable) when listening to an audio DTMF stream.
(refer to ADL_AUDIO_DTMF_DETECT_BLANK_DURATION for more information about Raw mode).

Code

```
typedef union
{
        ascii                               DecodedDTMFChars
                                            [ADL_AUDIO_MAX_DTMF_PER_FRAME]
        adl_audioPostProcessedDecoder_t     PostProcessedDTMF
} adl_audioDecodedDtmf_u;
```

Description

DecodedDTMFChars:

This field contains decoded DTMF in Raw mode.

PostProcessedDTMF:

This field contains information about decoded DTMF and decoding post-process. (Refer to adl_audioPostProcessedDecoder_t for more information).

3.33.2.2. The adl_audioPostProcessedDecoder_t Structure

This structure allows the application to handle post-processed DTMF data when listening to an audio DTMF stream with Raw mode deactivated.
(Refer to ADL_AUDIO_DTMF_DETECT_BLANK_DURATION for more information about Raw mode).

Code

```
typedef struct
{
      u32      Metrics;
      u32      Duration;
      ascii    DecodedDTMF
} adl_audioPostProcessedDecoder_t;
```

Description

Metrics:

Processing metrics, contains information about DTMF decoding process. **Reserved for Future Use**.

Duration:

DTMF duration, contains post-processed DTMF duration, in ms

DecodedDTMF:

PostProcessed DTMF buffer contains decoded DTMF.

3.33.2.3. The adl_audioStream_t Structure

This structure allows the application to handle data buffer according to the audio format when an audio stream interrupt occurs during a playing (**adl_audioStreamPlay**) or a listening to (**adl_audioStreamListen**) an audio stream.

Code

```
typedef struct
{
      adl_audioFormats_e                audioFormat;
      adl_audioStreamDataBuffer_u *     DataBuffer;
      bool *                            BufferReady;
      bool *                            BufferEmpty;
      bool *                            BufferOverwrite
} adl_audiostream_t;
```

Description

audioFormat:

Stream audio format (refer to **`adl_audioFormats_e`** for more information)

DataBuffer:

Audio data exchange buffer:

- This field stores audio sample during an audio PCM stream listening or decoded DTMF during an audio DTMF stream listening.
- It contains audio sample to play during an audio PCM stream playing. (Refer to 3.33.2.4 **`adl_audioStreamDataBuffer_u`** structure for more information).

BufferReady:

This flag is used for audio stream playing and listening process:

- When an audio stream is played, each time an interruption occurs this flag has to set to TRUE when data buffer is filled. If this flag is not set to TRUE, an 'empty' frame composed of 0x0 will be sent and set the BufferEmpty flag to TRUE. Once the sample is played BufferReady is set to FALSE by the firmware.
- When an audio stream is listened, each time an interruption occurs this flag has to be set to FALSE when data buffer is read. If this flag is not set to FALSE, then firmware will set BufferOverwrite flag to TRUE. **This pointer is initialized only when an audio stream is played or listened. Currently, it is only used for PCM stream playing and listening.**

BufferEmpty:

When an audio stream is played, this flag is set to TRUE when empty data buffer is played (for example, when an interruption is missing). This flag is used only for information and it has to be set to FALSE by application. **This pointer is initialized only when an audio stream is played. Currently, it is used only for PCM stream playing.**

BufferOverwrite:

When an audio stream is listened, this flag is set to TRUE when the last fame has been overwritten (for example, when an interruption is missing). This flag is used just for information, it has to be set to FALSE by application each time it accesses the data buffer. **This pointer is initialized only when an audio stream is listened. Currently, it is only used for PCM stream listening.**

3.33.2.4. The adl_audioStreamDataBuffer_u Union

This union defines different types of buffers, which are used according to the audio format when an audio stream interruption occurs.

Code

```
typedef union
{
      u8                          PCMData [1];
      u8                          AMRData [1];
      adl_audioDecodedDtmf_u      DTMFData
} adl_audiostreamDataBuffer_u;
```

Description

PCMData [1]:

PCM stream data buffer.

This buffer is used when playing or listening to an audio PCM stream.

AMRData [1]:

AMR stream data buffer.

This buffer is used when playing to an audio AMR / AMR-WB stream.

DTMFData:

DTMF stream data buffer.

This buffer stores decoded DTMF when listening to an audio DTMF stream according to the decoding mode which is used. Please refer to 3.33.2.1 **`adl_audioDecodedDtmf_u`** for more information about DTMF buffer structure and ADL_AUDIO_DTMF_DETECT_BLANK_DURATION for more information about decoding modes.

3.33.3. Defines

3.33.3.1. ADL_ AUDIO_MAX_DTMF_PER_FRAME

This constant defines maximal number of received DTMFs each time interrupt handlers are called when a listening to a DTFM stream in Raw mode (Refer to ADL_AUDIO_DTMF_DETECT_BLANK_DURATION for more information about Raw mode).

Code:

```
#define ADL_AUDIO_MAX_DTMF_PER_FRAME   2
```

3.33.3.2. ADL_AUDIO_NOTE_DEF

This macro is used to define the note value to play according to the note definition, the scale and the note duration.

To play a melody, each note defines in the melody buffer has to be defined with this macro (see section adl_audioMelodyPlay function).

Code:

```
#define ADL_AUDIO_NOTE_DEF  (  ID,
                               Scale,
                               Duration )(((ID)+(Scale*12))<<8)+(Duration));
```

Parameters

ID :

This parameter corresponds to the note identification. Please refer to the code below for the Group Notes identification for melody.

```
#define ADL_AUDIO_C         0x01     //C
#define ADL_AUDIO_CS        0x02     //C #
#define ADL_AUDIO_D         0x03     //D
#define ADL_AUDIO_DS        0x04     //D #
#define ADL_AUDIO_E         0x05     //E
#define ADL_AUDIO_F         0x06     //F
#define ADL_AUDIO_FS        0x07     //F #
#define ADL_AUDIO_G         0x08     //G
#define ADL_AUDIO_GS        0x09     //G #
#define ADL_AUDIO_A         0x0A     //A
#define ADL_AUDIO_AS        0x0B     //A #
define ADL_AUDIO_B          0x0C     //B
#define ADL_AUDIO_NO_SOUND  0xFF     //No sound
```

Scale:

This parameter defines the note scale (0 - 7).

Duration:

This parameter defines the note duration. Please refer to the Group Notes Durations code below to see the set of note durations which are available.

```
#define ADL_AUDIO_WHOLE_NOTE       0x10   //Whole note
#define ADL_AUDIO_HALF             0x08   //Half note
#define ADL_AUDIO_QUARTER          0x04   //Quarter note
#define ADL_AUDIO_EIGHTH           0x02   //Eighth note
#define ADL_AUDIO_SIXTEENTH        0x01   //Sixteenth note
#define ADL_AUDIO_DOTTED_HALF      0x0C   //Dotted half note
#define ADL_AUDIO_DOTTED_QUARTER   0x06   //Dotted quarter
#define ADL_AUDIO_DOTTED_EIGHTH    0x03   //Dotted Eighth
```

3.33.4. Enumerations

3.33.4.1. The adl_ audioResources_e Type

This type lists the available audio resources of the embedded module, including the local ones (plugged to the embedded module itself) and the ones related to any running voice call. These resources are usable either to play a pre-defined/stream audio format (output resources), or to listen to an incoming audio stream (input resources).

Code

```
typedef enum
{
      ADL_AUDIO_SPEAKER,
      ADL_AUDIO_BUZZER,
      ADL_AUDIO_MICROPHONE,
      ADL_AUDIO_VOICE_CALL_RX,
      ADL_AUDIO_VOICE_CALL_TX
} adl_audioResources_e;
```

Description

ADL_AUDIO_SPEAKER:	Current speaker (output resource; please refer to the AT Command interface guide for more information on how to select the current speaker).
ADL_AUDIO_BUZZER:	Buzzer (output resource, just usable to play single frequency tones & melodies).
ADL_AUDIO_MICROPHONE:	Current microphone (input resource; please refer to the AT Command interface guide for more information on how to select the current microphone).
ADL_AUDIO_VOICE_CALL_RX:	Running voice call incoming channel (input resource, available when a voice call is running to listen to audio streams).
ADL_AUDIO_VOICE_CALL_TX:	Running voice call outgoing channel (output resource, available when a voice call is running to play audio streams).

3.33.4.2. The adl_audioResourceOption_e Type

This type defines the audio resource subscription options.

Code

```
typedef enum
{
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION  = 0x00,
      ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION   = 0x01
} adl_audioResourceOption_e;
```

Description

ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION:

Never allows prioritary uses of the resource (the resource subscriber owns the resource until unsubscription time).

ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION:

Allows prioritary uses of the resource (such as incoming voice call melody, outgoing voice call tone play, SIM Toolkit application tone play).

3.33.4.3. The adl_audioFormats_e Type

This type defines the audio stream formats for audio stream playing/listening processes.

Code

```
typedef enum
{
      ADL_AUDIO_DTMF                //Decoded DTMF sequence
      ADL_AUDIO_PCM_MONO_8K_16B     //PCM mono 16 bits/8 KHz Audio sample
      ADL_AUDIO_PCM_MONO_16K_16B    //PCM mono 16 bits/16 KHz Audio sample
      ADL_AUDIO_AMR                 //AMR/AMR-WB Audio sample
} adl_audioFormats_e;
```

3.33.4.4. The adl_audioInstance_e Type

Instance set of the audio interrupt event which occurs when audio stream listening or playing is started. Refer to **Instance** field in adl_irqEventData_t structure for more information.

Code

```
typedef enum
{
        ADL_AUDIO_DTMF_INSTANCE,
        ADL_AUDIO_PCM_INSTANCE,
        ADL_AUDIO_AMR_INSTANCE
} adl_audioInstance_e;
```

Description

`ADL_AUDIO_DTMF_INSTANCE:`

For DTMF decoding interruption.

`ADL_AUDIO_PCM_INSTANCE:`

When audio stream recording or playing is started with **`ADL_AUDIO_PCM_MONO_8K_16B`** or **`ADL_AUDIO_PCM_MONO_16K_16B`** format.

`ADL_AUDIO_AMR_INSTANCE:`

When audio stream recording or playing is started with **`ADL_AUDIO_AMR`** format

3.33.4.5. The adl_audioAmrCodecRate_e Type

Available speech codec rate for audio ARM / AMR-WB stream playing process.

Code

```
typedef enum
{
        ADL_AUDIO_AMR_RATE_4_75,
        ADL_AUDIO_AMR_RATE_5_15,
        ADL_AUDIO_AMR_RATE_5_90,
        ADL_AUDIO_AMR_RATE_6_70,
        ADL_AUDIO_AMR_RATE_7_40,
        ADL_AUDIO_AMR_RATE_7_95,
        ADL_AUDIO_AMR_RATE_10_20,
        ADL_AUDIO_AMR_RATE_12_20,
        ADL_AUDIO_AMR_WB_RATE_6_60,
        ADL_AUDIO_AMR_WB_RATE_8_85,
        ADL_AUDIO_AMR_WB_RATE_12_65,
        ADL_AUDIO_AMR_WB_RATE_14_25,
        ADL_AUDIO_AMR_WB_RATE_15_85,
        ADL_AUDIO_AMR_WB_RATE_18_25,
        ADL_AUDIO_AMR_WB_RATE_19_85,
        ADL_AUDIO_AMR_WB_RATE_23_05,
        ADL_AUDIO_AMR_WB_RATE_23_85
} adl_audioAmrCodecRate_e;
```

Description

ADL_AUDIO_AMR_RATE_4_75:
AMR codec rate 4.75 kb/s.

ADL_AUDIO_AMR_RATE_5_15:
AMR codec rate 5.15 kb/s.

ADL_AUDIO_AMR_RATE_5_90:
AMR codec rate 5.90 kb/s.

ADL_AUDIO_AMR_RATE_6_70:
AMR codec rate 6.70 kb/s.

ADL_AUDIO_AMR_RATE_7_40:
AMR codec rate 7.40 kb/s.

ADL_AUDIO_AMR_RATE_7_95:
AMR codec rate 7.95 kb/s.

ADL_AUDIO_AMR_RATE_10_20:
AMR codec rate 10.20 kb/s.

ADL_AUDIO_AMR_RATE_12_20:
AMR codec rate 12.20 kb/s.

ADL_AUDIO_AMR_WB_RATE_6_60:
AMR-WB codec rate 6.60 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**.

ADL_AUDIO_AMR_WB_RATE_8_85:
AMR-WB codec rate 8.85 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_12_65:
AMR-WB codec rate 12.65 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_14_25:
AMR-WB codec rate 14.25 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_15_85:
AMR-WB codec rate 15.85 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_18_25:
AMR-WB codec rate 18.25 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_19_85:
AMR-WB codec rate 19.85 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_23_05:
AMR-WB codec rate 23.05 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

ADL_AUDIO_AMR_WB_RATE_23_85:
AMR-WB codec rate 23.85 kb/s, refer to **ADL_::AUDIO_AMR_WB_AVAILABLE**

3.33.4.6. The adl_audioEvents_e Type

Set of events that will be notified by ADL to audio event handlers.

Code

```
typedef enum
{
        ADL_AUDIO_EVENT_NORMAL_STOP,
        ADL_AUDIO_EVENT_RESOURCE_RELEASED
} adl_audioEvents_e;
```

Description

`ADL_AUDIO_EVENT_NORMAL_STOP:`

A pre-defined audio format play has ended (please refer to adl_audioDTMFPlay, adl_audioTonePlay or adl_audioMelodyPlay for more information). This event is not sent on a request to stop from application.

`ADL_AUDIO_EVENT_RESOURCE_RELEASED:`

Resource has been automatically unsubscribed due to a prioritary use by the embedded module (please refer to the ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION option and adl_audioSubscribe for more information).

3.33.4.7. The adl_audioOptionTypes_e Type

This type includes a set of options readable and writable through the **`adl_audioSetOption`** and **`adl_audioGetOption`** functions. These options allow to configure the embedded module audio service behaviour, and to get this audio service capabilities and parameters ranges.

For each option, the value type is specified, and a specific keyword indicates the option access:

- **R:** the option is only readable.
- **RW:** the option is both readable & writable.

Note: *For more information about indicative values which should be returned when reading options for MIN/MAX values, please refer to the Audio Commands chapter of the AT Commands Interface Guide.*

Code

```
typedef enum
{
        ADL_AUDIO_DTMF_DETECT_BLANK_DURATION,
        ADL_AUDIO_MAX_FREQUENCY,
        ADL_AUDIO_MIN_FREQUENCY,
        ADL_AUDIO_MAX_GAIN,
        ADL_AUDIO_MIN_GAIN,
        ADL_AUDIO_MAX_DURATION,
        ADL_AUDIO_MIN_DURATION,
        ADL_AUDIO_MAX_NOTE_VALUE,
        ADL_AUDIO_MIN_NOTE_VALUE,
        ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE,
        ADL_AUDIO_DTMF_PROCESSED_STREAM_BUFFER_SIZE,
        ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE,
        ADL_AUDIO_PCM_16K_16B_MONO_BUFFER_SIZE,
        ADL_AUDIO_AMR_WB_AVAILABLE,
        ADL_AUDIO_AMR_SPEECH_CODEC_RATE,
        ADL_AUDIO_AMR_MIXED_VOICE,
        ADL_AUDIO_AMR_BUFFER_SIZE,
        ADL_AUDIO_RAW_DTMF_SAMPLE_DURATION
} adl_audioOptionTypes_e;
```

Description

`ADL_AUDIO_DTMF_DETECT_BLANK_DURATION`

RW: DTMF decoding option (u16); it allows to define the blank duration (ms) in order to detect the end of a DTMF. This value will act on the embedded module behaviour to return information about DTMF when listening to a DTMF audio stream. The value must be a multiple of value returned by **`ADL_AUDIO_RAW_DTMF_SAMPLE_DURATION`** option multiplied by **`ADL_AUDIO_MAX_DTMF_PER_FRAME`**.

If a NULL value is specified, DTMF decoder will be in **Raw mode** (default), Raw data coming from DTMF decoder are sent via interrupt handlers with a frequency which depends on value returned by **`ADL_AUDIO_RAW_DTMF_SAMPLE_DURATION`** option multiplied by **`ADL_AUDIO_MAX_DTMF_PER_FRAME`**. This mode requires to implement an algorithm to detect the relevant DTMF. (Refer to adl_audioDecodedDtmf_u **`::DecodedDTMFChars`** for more information about buffer type used).

Otherwise, the Raw mode is disabled. The value specifies the blank duration which notifies the end of DTMF. Each time a DTMF is detected, interrupt handlers are called. (Refer to adl_audioPostProcessedDecoder_t structure for more information about stored data).

`ADL_AUDIO_MAX_FREQUENCY`

R: allows to get the maximum frequency allowed to be played on the required output resource (please refer to adl_audioResourceOption_e for more information). The returned frequency value is defined in Hz (u16).

`ADL_AUDIO_MIN_FREQUENCY`

R: allows to get the minimum frequency allowed to be played on the required output resource (please refer to adl_audioResourceOption_e for more information). The returned frequency value is defined in Hz (u16).

ADL_AUDIO_MAX_GAIN

R: supplies the maximum gain which can be set to play a pre-defined audio format (please refer to adl_audioDTMFPlayExt, adl_audioTonePlayExt or adl_audioMelodyPlayExt for more information). The returned gain value is defined in 1/100 of dB (s16). This value can be retrieved only with **ADL_AUDIO_SPEAKER** and **ADL_AUDIO_BUZZER** audio resource handle. Otherwise, an error will be returned.

ADL_AUDIO_MIN_GAIN

R: supplies the minimum gain which can be set to play a pre-defined audio format (please refer to adl_audioDTMFPlayExt, adl_audioTonePlayExt or adl_audioMelodyPlayExt for more information). The returned gain value is defined in 1/100 of dB (s16). This value can be retrieved only with **ADL_AUDIO_SPEAKER** and **ADL_AUDIO_BUZZER** audio resource handle. Otherwise, an error will be returned.

ADL_AUDIO_MAX_DURATION

R: supplies the maximum duration which can be set to play a DTMF tone or a single/dual tone (please refer to adl_audioDTMFPlay or adl_audioTonePlay for more information). The returned duration value is defined in ms (u32). This value can be retrieved only with **ADL_AUDIO_SPEAKER** and **ADL_AUDIO_BUZZER** audio resource handle. Otherwise, an error will be returned.

ADL_AUDIO_MIN_DURATION

R: supplies the minimum duration which can be set to play a DTMF tone or a single/dual tone (please refer to adl_audioDTMFPlay or adl_audioTonePlay for more information). The returned duration value is defined in ms (u32). This value can be retrieved only with **ADL_AUDIO_SPEAKER** and **ADL_AUDIO_BUZZER** audio resource handle. Otherwise, an error will be returned.

ADL_AUDIO_MAX_NOTE_VALUE

R: supplies the maximum duration for a note (tempo) which can be set to play a melody (please refer to adl_audioMelodyPlay for more information). This value is the maximal value which can be defined with **ADL_AUDIO_NOTE_DEF** macro (u32).

ADL_AUDIO_MIN_NOTE_VALUE

R: supplies the minimum duration for a note (tempo) which can be set to play a melody (please refer to adl_audioMelodyPlay for more information). This value is the minimal value which can be defined with **ADL_AUDIO_NOTE_DEF** macro (u32).

ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE

R: allows to get the buffer type to allocate for listening to a DTMF stream in Raw mode or playing a DTMF stream, defined in number of bytes (u8).

ADL_AUDIO_DTMF_PROCESSED_STREAM_BUFFER_SIZE

R: allows to get the buffer type to allocate for listening to a DTMF stream in Pre-processed mode, defined in number of bytes (u8).

ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE

R: allows to get the buffer type to allocated for playing or listening to on a PCM 8KHz 16 bits Mono stream, defined in number of bytes (u16).

ADL_AUDIO_PCM_16K_16B_MONO_BUFFER_SIZE

R: allows to get the buffer type to allocated for playing or listening to on a PCM 16KHz 16 bits Mono stream, defined in number of bytes (u16).

ADL_AUDIO_AMR_WB_AVAILABLE

R: allows to know if AMR Wideband codec rates are available. TRUE if they are available, FALSE otherwhise (bool). Refer to adl_audioAmrCodecRate_e to get available codec rates.

ADL_AUDIO_AMR_SPEECH_CODEC_RATE

RW: allows to define which codec rate will be used for AMR stream playing. Refer to adl_audioAmrCodecRate_e to get available codec rates. By default, Codec rate is ADL_AUDIO_AMR_RATE_4_75.

ADL_AUDIO_AMR_MIXED_VOICE

RW: allows to define if the AMR sample should be mixed to the voice when an AMR audio sample is played. This value is set to FALSE to mute vocoder, TRUE otherwise. By default, option is set to FALSE (bool).

ADL_AUDIO_AMR_BUFFER_SIZE

R: allows to define the buffer type to allocated for playing or listening to on an AMR stream, defined in number of bytes (u32). By default, option is set to 0.

According to the selected codec rate, the buffer has to be defined with a multiple of one speech frame size, "0" is not available (refer to adl_audioStreamPlay to get more information about buffer to allocated.

The option value has to match with size of AMR buffer which has been allocated. Otherwise, AMR player (/recorder) risks not to work properly.

ADL_AUDIO_RAW_DTMF_SAMPLE_DURATION

R: allows to get the duration of one DTMF sample when DTMF decoding is on Raw mode, defined in ms (u8). This value depends on the embedded module which is used.

3.33.5. Audio events handler

This call-back function has to be supplied to ADL through the **adl_audioSubscribe** interface in order to receive audio resource related events

prototype

```
typedef void(*) adl_audioEventHandler_f( s32                 audioHandle,
                                         adl_audioEvents_e   Event);
```

parameters

audioHandle

This is the handle of the audio resource which is associated to the event (refer to adl_audioSubscribe for more information about the audio resource handle).

Event

This is the received event identifier (refer to adl_audioEvents_e for more information about the different events).

3.33.6. Audio resources control

3.33.6.1. The adl_audioSubscribe Function

This function allows to subscribe to the one of the available resources and specify its behaviour when another client attempts to subscribe it.
A call-back function is associated for audio resources related events, the **adl_audioPostProcessedDecoder_t** Type.

Prototype

```
s32 adl_audioSubscribe  ( adl_audioResources_e      audioResource,
                          adl_audioEventHandler_f   audioEventHandler,
                          adl_audioResourceOption_e Options );
```

Parameters

audioResource

Requested audio resource.

audioEventHandler

Application provided audio event call-back function (refer to adl_audioEventHandler_f for more information.

Options

Option about the audio resource behaviour (refer to adl_audioResourceOption_e for more information).

Returned values

- Positive or NULL if allocation succeeds, to be used on further audio API functions calls.
- **ADL_RET_ERR_PARAM** if the parameter has an incorrect value.
- **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the resource is already subscribed.
- **ADL_RET_ERR_NOT_SUPPORTED** if the resource is not supported.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

*Note: ERROR values are defined in **adl_error.h**.*

3.33.6.2. The adl_audioUnsubscribe Function

This function allows to unsubscribe to one of the resources which have been previously subscribed.

A resource cannot be unsubscribed if it is running, process on this resource has to be previously stopped (refer to adl_audioStop for more information).

Prototype

```
s32 adl_audioUnsubscribe ( s32  audioHandle );
```

Parameter

audioHandle

Handle of the audio resource which has to be unsubscribed.

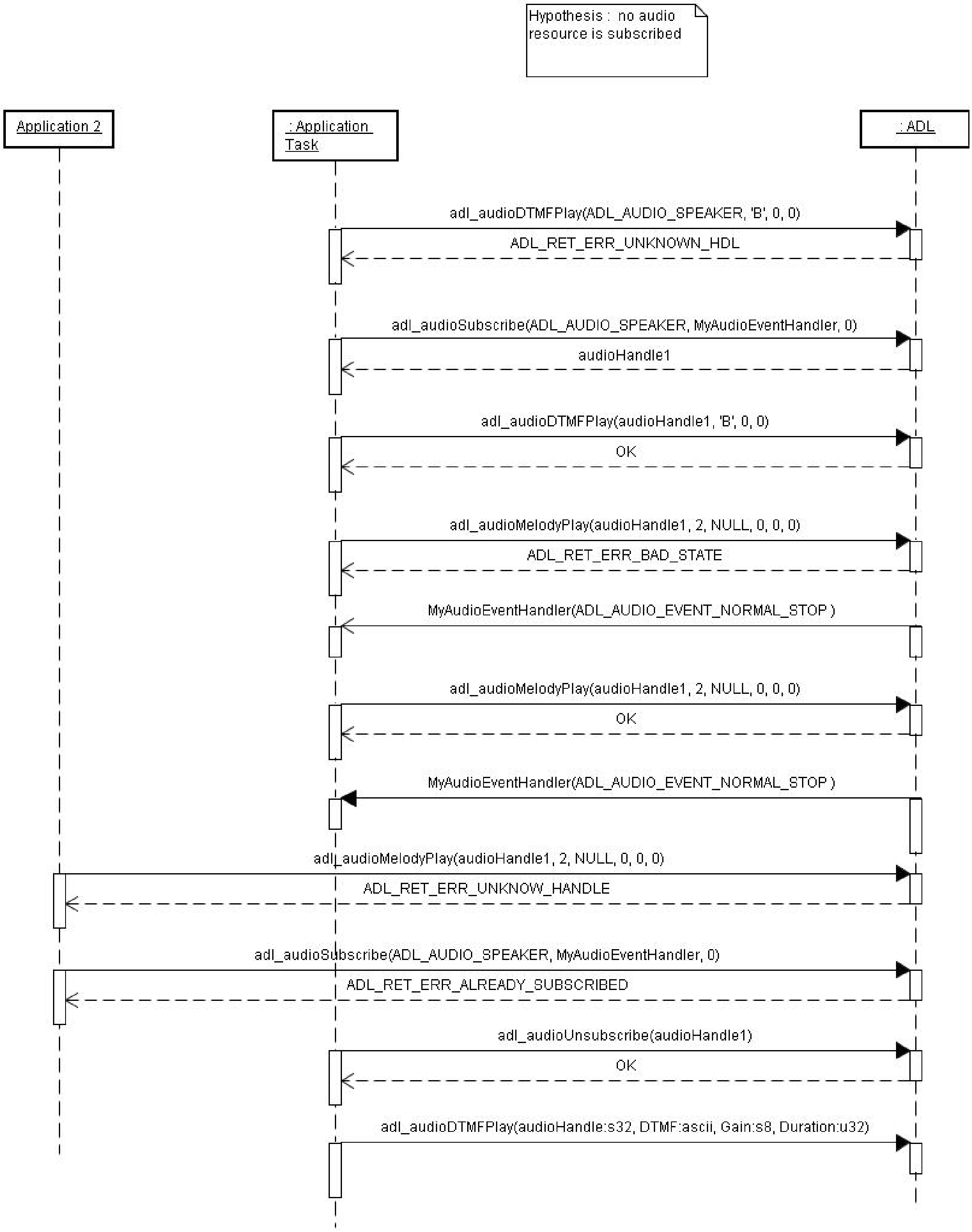
Returned values

- **OK** on success
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_NOT_SUBSCRIBED** if no audio resource has been subscribed.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.33.7. Play a pre-defined audio format

These functions allow to play a melody, a tone or a DTMF on the available audio outputs.

The following diagram illustrates a typical use of the ADL Audio Service interface to play a predefined audio format.


Hypothesis : no audio resource is subscribed
Application 2
: Application Task
: ADL
adl_audioDTMFPlay(ADL_AUDIO_SPEAKER, 'B', 0, 0)
ADL_RET_ERR_UNKNOWN_HDL
adl_audioSubscribe(ADL_AUDIO_SPEAKER, MyAudioEventHandler, 0)
audioHandle1
adl_audioDTMFPlay(audioHandle1, 'B', 0, 0)
OK
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
ADL_RET_ERR_BAD_STATE
MyAudioEventHandler(ADL_AUDIO_EVENT_NORMAL_STOP)
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
OK
MyAudioEventHandler(ADL_AUDIO_EVENT_NORMAL_STOP)
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
ADL_RET_ERR_UNKNOW_HANDLE
adl_audioSubscribe(ADL_AUDIO_SPEAKER, MyAudioEventHandler, 0)
ADL_RET_ERR_ALREADY_SUBSCRIBED
adl_audioUnsubscribe(audioHandle1)
OK
adl_audioDTMFPlay(audioHandle:s32, DTMF:ascii, Gain:s8, Duration:u32)

3.33.7.1. The adl_audioTonePlay Function

This function plays a single or dual tone on current speaker and only a single tone on buzzer.
Only the speaker output is able to play tones in two frequencies. The second tone parameters are ignored on buzzer output.
The specified output stops to play at the end of tone duration or on an application request (refer to adl_audioStop for more information).
Use **`adl_audioGetOption`** function to obtain the parameters range. Please also refer to AT commands Interface User Guide 1 for more information.

Prototype

```
s32 adl_audioTonePlay (  s32       audioHandle,
                         u16       Frequency1,
                         s8        Gain1,
                         u16       Frequency2,
                         s8        Gain2,
                         u32       Duration );
```

Parameters

audioHandle

Handle of the audio resource which will play tone (current speaker or buzzer).

Frequency1

Frequency for the 1st tone (Hz).

Gain1

This parameter sets the tone gain which will be applied to the 1st frequency value (dB).

Frequency2

Frequency for the 2nd tone (Hz), only processed on current speaker. Frequency2 has to set to 0 to play a single tone on current speaker.

Gain2

This parameter sets the tone gain which will be applied to the 2nd frequency value (dB).

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

Returned values

- **`OK`** on success.
- **`ADL_RET_ERR_PARAM`** if parameters have an incorrect value.
- **`ADL_RET_ERR_UNKNOWN_HDL`** if the provided handle is unknown.
- **`ADL_RET_ERR_BAD_STATE`** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **`ADL_RET_ERR_BAD_HDL`** if the audio resource is not allowed for tone playing.
- **`ADL_RET_ERR_NOT_SUPPORTED_`** if the audio resource is not available for tone playing.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler.

*Note: An event **`ADL_AUDIO_EVENT_NORMAL_STOP`** is sent to the owner resource when a tone is stopped automatically at the end of the duration time.*

Example

```
// audio resource handle
  s32 handle;

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {
            switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
               TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the speaker
              Ret = adl_audioUnsubscribe ( handle );
              break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandle,
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a single tone
      Ret = adl_audioTonePlay( handle, 300, -10, 0, 0, 50 );
  }
```

3.33.7.2. The adl_audioDTMFPlay Function

This function allows a DTMF tone to be played on the current speaker or on voice call TX (in communication only).

It is not possible to play DTMF on the buzzer.
The specified output stops to play at the end of tone duration or on an application request (refer to adl_audioStop for more information).
Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide 1 for more information.

Prototype

```
s32 adl_audioDTMFPlay (   s32       audioHandle,
                          ascii     DTMF,
                          s8        Gain,
                          u32       Duration );
```

Parameters

audioHandle

Handle of the audio resource which will play DTMF tone (current speaker or voice call TX).

DTMF

DTMF to play (0-9,A-D,*,#).

Gain

This parameter sets the tone gain (dB), and is only for the speaker.

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple. For voice call TX, duration is not guaranteed, which depends on operator.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for DTMF playing.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for DTMF playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a DTMF is stopped automatically at the end of the duration time.*

Note: *A DTMF cannot be stopped on client request when DTMF is played on voice call TX.*

Note: *When DTMF is played on voice call TX, no **ADL_AUDIO_EVENT_NORMAL_STOP** is received in audio event handler.*

Example

```
// audio resource handle
s32 handle;

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
              TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the current speaker
              Ret = adl_audioUnsubscribe ( handle );
          break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
     handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler,
     ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a DTMF tone
      Ret = adl_audioDTMFPlay( handle, 'A', -10, 10);
  }
```

3.33.7.3. The adl_audioMelodyPlay Function

This function allows to play a defined melody on current speaker or buzzer.
The specified output stops the playing process on an application request (refer to adl_audioStop for more information) or when the melody has been played the same number of time than that is specified in CycleNumber.
Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide 1 for more information.

Prototype

```
s32 adl_audioMelodyPlay (   s32       audioHandle,
                            u16 *     MelodySeq,
                            u8        Tempo,
                            u8        CycleNumber,
                            s8        Gain   );
```

Parameters

audioHandle

Handle of the audio resource which will play Melody (current speaker or buzzer).

MelodySeq

Melody to play. A melody is defined by an u16 table , where each element defines a note event, duration and sound definition.
The melody sequence has to finish by a NULL value.
(refer to ADL_AUDIO_NOTE_DEF for more information)

Tempo

Tempo is defined in bpm (1 beat = 1 quarter note).

CycleNumber

Number of times the melody should be played.
If not specified, the cycle number is infinite, Melody should be stopped by client.

Gain

This parameter sets melody gain (dB).

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for melody playing.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for melody playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a Melody is stopped automatically at the end of the cycle number.*

Example

```
// audio resource handle
  s32 handle;

// Melody buffer
u16*MyMelody={ADL_AUDIO_NOTE_DEF( ADL_AUDIO_A,3,ADL_AUDIO_DOTTED_QUARTER),
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_CS,5,ADL_AUDIO_DOTTED_HALF),
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_E,1,ADL_AUDIO_WHOLE_NOTE ),
              ... ,
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_AS,3,ADL_AUDIO_EIGHTH),
              0 };

// audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {
      s32 Ret;

      switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
              TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the buzzer
              Ret = adl_audioUnsubscribe ( handle );

          break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_BUZZER, MyAudioEventHandler ,
ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a Melody
      Ret = adl_audioMelodyPlay( handle, MyMelody, 10, 2, -10);
  }
```

3.33.7.4. The adl_audioTonePlayExt Function

This function plays a single or dual tone on current speaker and only a single tone on buzzer.
Only the speaker output is able to play tones in two frequencies. The second tone parameters are ignored on buzzer output.
The specified output stops to play at the end of tone duration or on an application request (refer to adl_audioStop for more information).
Use **`adl_audioGetOption`** function to obtain the parameters range. Please also refer to AT commands Interface User Guide 1 for more information.

Prototype

```
s32 adl_audioTonePlayExt(   s32       audioHandle,
                            u16       Frequency1,
                            s16       Gain1,
                            u16       Frequency2,
                            s16       Gain2,
                            u32       Duration );
```

Parameters

audioHandle

Handle of the audio resource which will play tone (current speaker or buzzer).

Frequency1

Frequency for the 1st tone (Hz).

Gain1

This parameter sets the tone gain which will be applied to the 1st frequency value (unit: 1/100 of dB).

Frequency2

Frequency for the 2nd tone (Hz), only processed on current speaker.
Frequency2 has to set to 0 to play a single tone on current speaker.

Gain2

This parameter sets the tone gain which will be applied to the 2nd frequency value (unit : 1/100 of dB).

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

Returned values

- **`OK`** on success.
- **`ADL_RET_ERR_PARAM`** if parameters have an incorrect value.
- **`ADL_RET_ERR_UNKNOWN_HDL`** if the provided handle is unknown.
- **`ADL_RET_ERR_BAD_STATE`** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **`ADL_RET_ERR_BAD_HDL`** if the audio resource is not allowed for tone playing.
- **`ADL_RET_ERR_NOT_SUPPORTED_`** if the audio resource is not available for tone playing.
- **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler.

Note: *An event **`ADL_AUDIO_EVENT_NORMAL_STOP`** is sent to the owner resource when a tone is stopped automatically at the end of the duration time.*

3.33.7.5. The adl_audioDTMFPlayExt Function

This function allows a DTMF tone to be played on the current speaker or on voice call TX (in communication only).

It is not possible to play DTMF on the buzzer.

The specified output stops to play at the end of tone duration or on an application request (refer to adl_audioStop for more information).
Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide 1 for more information.

Prototype

```
s32 adl_audioDTMFPlayExt(   s32      audioHandle,
                            ascii    DTMF,
                            s16      Gain,
                            u32      Duration );
```

Parameters

audioHandle

Handle of the audio resource which will play DTMF tone (current speaker or voice call TX).

DTMF

DTMF to play (0-9,A-D,*,#).

Gain

This parameter sets the tone gain (unit: 1/100 of dB), and is only for the speaker.

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple. For voice call TX, duration is not guaranteed, which depends on operator.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for DTMF playing.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for DTMF playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a DTMF is stopped automatically at the end of the duration time.*

Note: *A DTMF cannot be stopped on client request when DTMF is played on voice call TX.*

Note: *When DTMF is played on voice call TX, no **ADL_AUDIO_EVENT_NORMAL_STOP** is received in audio event handler.*

3.33.7.6. The adl_audioMelodyPlayExt Function

This function allows to play a defined melody on current speaker or buzzer.

The specified output stops the playing process on an application request (refer to adl_audioStop for more information) or when the melody has been played the same number of time than that is specified in CycleNumber.

Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface Guide for more information.

Prototype

```
s32 adl_audioMelodyPlayExt (    s32       audioHandle,
                                u16 *     MelodySeq,
                                u8        Tempo,
                                u8        CycleNumber,
                                s16       Gain   );
```

Parameters

audioHandle

Handle of the audio resource which will play Melody (current speaker or buzzer).

MelodySeq

Melody to play. A melody is defined by an u16 table , where each element defines a note event, duration and sound definition.
The melody sequence has to finish by a NULL value.
(refer to ADL_AUDIO_NOTE_DEF for more information)

Tempo

Tempo is defined in bpm (1 beat = 1 quarter note).

CycleNumber

Number of times the melody should be played.
If not specified, the cycle number is infinite; Melody should be stopped by client.

Gain

This parameter sets melody gain (unit: 1/100 of dB).

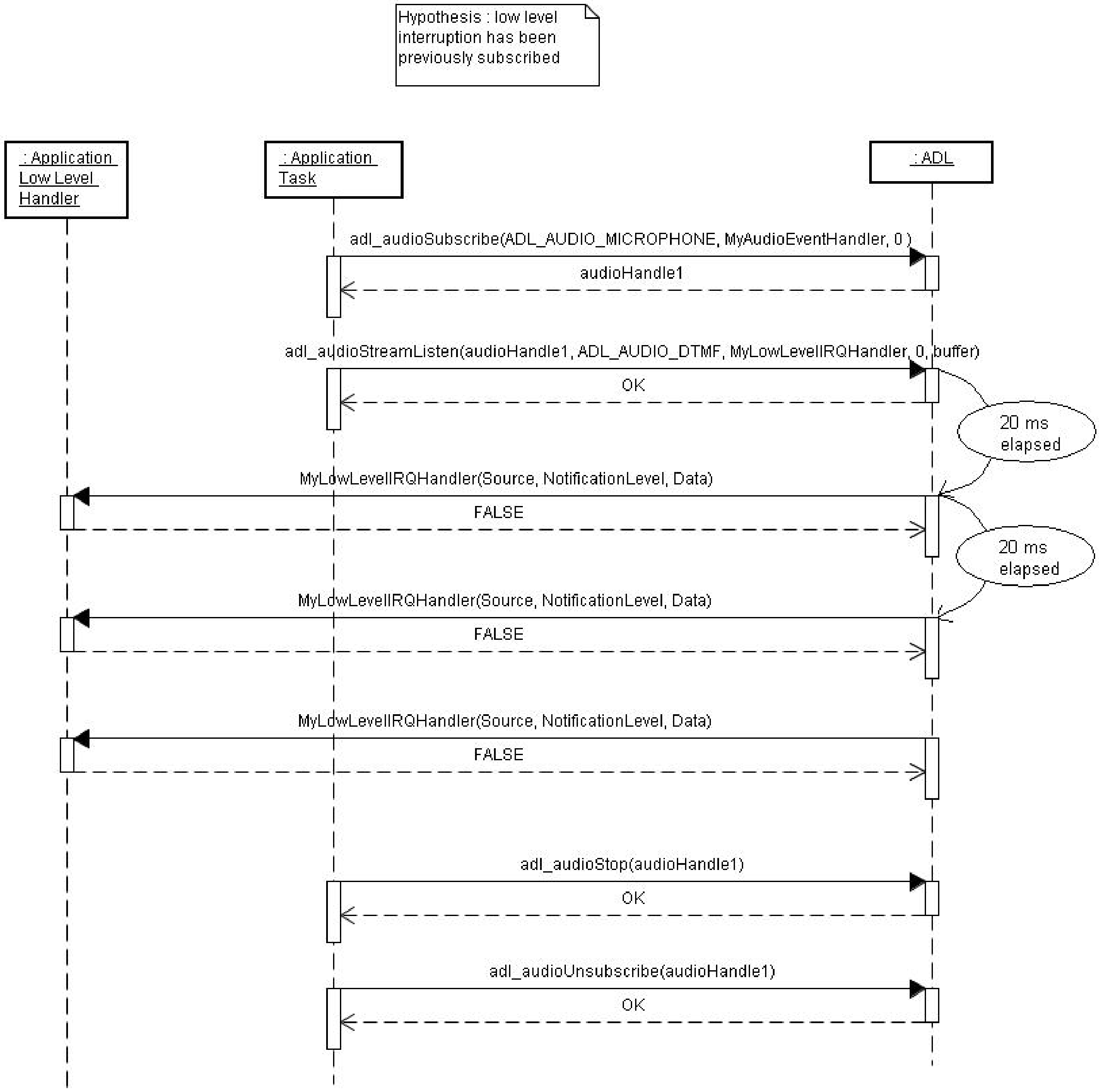
Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for melody playing.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for melody playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a Melody is stopped automatically at the end of the cycle number.*

3.33.8. Audio stream

These functions allows to play or listen an audio stream.



3.33.8.1. The adl_audioStreamPlay Function

This function allows to play an audio sample stream on the current speaker or on voice call TX.

Playing an audio sample stream implies that low level interruption and/or high level interruption have been previously subscribed

(Refer to adl_irqSubscribe in ADL user guide for more information).

Moreover, memory space has to be allocated for the audio stream buffer before playing starts and it has to be released after playing stops.

Each time the interruption related to playing process occurs, allocated buffer has to be filled with audio data to play in IRQ low or high level notification handler. Currently, this functions allows to play 3 audio formats:

- audio signal sampled at 8KHz on16 bits (**ADL_AUDIO_PCM_MONO_8K_16B)**
- audio signal sampled at 16KHz on 16 bits (**ADL_AUDIO_PCM_MONO_16K_16B**).Only available on current speaker
- audio signal compressed by an AMR / AMR-WB codec (**ADL_AUDIO_AMR**). Refer to adl_audioAmrCodecRate_e to get more information about available codecs.Playing audio file compressed by AMR-WB codec is only available on current speaker

Play PCM audio format

Before starting a PCM audio playing process, the application has to set the embedded module audio configuration according to the sample rate (8KHz or 16KHz) of audio file to be played. Refer to the **AT+SPEAKER** command in "AT Commands Interface Guide" to get more information about audio resource configuration.

According to the audio configuration a different space memory size will be allocated (use **adl_audioGetOption** function to get the size):

- refer to **ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE** for a sample rate at 8KHz
- refer to **ADL_AUDIO_PCM_16K_16B_MONO_BUFFER_SIZE** for a sample rate at 16KHz

Warning: *If allocated buffer size does not match with the sample rate, audio playing process may not work properly.*

Play AMR audio format

This function can play only AMR / AMR-WB audio file stored in RTP format (refer to RFC4867 to get more information about RTP format for AMR and AMR-WB). Before starting an AMR audio playing process, the application has to set parameters such as codec rate (refer to adl_audioAmrCodecRate_e to get available codec rate), buffer size (refer to ADL_AUDIO_AMR_BUFFER_SIZE to get more information), mixed voice option (refer to ADL_AUDIO_AMR_MIXED_VOICE to get more information). According to audio configuration, an audio signal compressed either with AMR codec or with AMR-WB codec could be played:

- for an audio signal with sample rate at 8 KHZ, an AMR codec has to be used
- for an audio signal with sample rate at 16 KHZ, an AMR-WB codec has to be used. **AMR-WB audio recording is only available on speaker.** Refer to the **AT+SPEAKER** command in "AT Commands Interface Guide" to get more information about audio resource configuration. The buffer size, which has to be allocated, depends on the codec rate selected by the application. For each codec rate, a minimal space memory size has to be allocated. Buffer size has to be either an audio AMR file size or multiple of one 20-ms audio AMR speech frame size (this last one depends on codec rate).

Warning: *If allocated buffer size does not match with codec rate, quality of played audio signal may be altered.*

When AMR audio file is played on voice call and high level IRQ notification has been subscribed with ***ADL_IRQ_OPTION_AUTO_READ*** *option (refer to [adl_irq.h](#) to get more information about this option) and audio buffer is too huge then a network de-registration may occur. In this case,* ***ADL_IRQ_OPTION_AUTO_READ*** *option should not be used or audio buffer size should be a small AMR speech frame size.*

Prototype

```
s32 adl_audioStreamPlay  (  s32                  audioHandle,
                            adl_audioFormats_e   audioFormat,
                            s32                  LowLevelIRQHandle,
                            s32                  HighLevelIRQHandle,
                            void *               buffer );
```

Parameters

audioHandle

Handle of the audio resource which will play audio stream (current speaker or voice call TX).

audioFormat

Stream audio format. Only **ADL_AUDIO_DTMF** format is not available to be played (Refer to [adl_audioFormats_e](#) for more information).

LowLevelIRQHandle

Low level IRQ handle previously returned by IRQ subscription (please refer to [adl_irqSubscribe](#) for more information).

HighLevelIRQHandle

High level IRQ handle previously returned by IRQ subscription (please refer to [adl_irqSubscribe](#) for more information).

buffer

contains sample to play.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for audio stream playing or if interrupt handler identifiers are invalid.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for audio stream playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *To work properly, LowLevelIRQHandle is mandatory. The low level interrupt has to be previously subscribed with* ***ADL_IRQ_OPTION_AUTO_READ*** *option.*

Note: *The HighLevelIRQHandle is optional.*

Note: *Each time an audio sample is required, an interrupt handler will be notified to send the data. The interrupt identifier will be set to* ***ADL_IRQ_ID_AUDIO_RX_PLAY*** *or* ***ADL_IRQ_ID_AUDIO_TX_PLAY****, according to the resource used to start the stream play.*

Note: *in order to work properly, data should be sent in low level interruption handler*

Note: *Some audio filters will be deactivated for audio sample playing (refer to "audio command" chapter in the AT command Interface Guide 1 for more information).*

Note: *For audio interrupt subscription **ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT** option is not available.*

Note: *Before to play an audio file, header of file has to be removed, only data has to be send.*

Example

```
Start PCM audio playing process

// audio resource handle
  s32 handle;

  // audio stream buffer
  void * StreamBuffer;

  // PCM samples
  u16 PCM_Samples[160] = { ... , ... , ... , ... , ... , ... , 0 };         //
size of PCM sample = 320 bytes

  // PCM samples index
  u8 indexPCM = 0;

  // Low level interrupt handler
  bool MyLowLevelIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
                              Notification Level, adl_irqEventData_t * Data )
  {
      // copy PCM sample to play
      wm_strcpy( StreamBuffer, PCM_Samples );
      // Set BufferReady flag to TRUE
      *( ( adl_audioStream_t * )Data->SourceData )->BufferReady = TRUE;

      //...

      return FALSE;
  }

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;
      s32 BufferSize;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler ,
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Memory allocation
      Ret = adl_audioGetOption ( handle,
      ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE, &BufferSize )
      StreamBuffer = adl_memGet( BufferSize );   // release memory after
                                                    audio stream playing

      // Play an audio PCM stream
      Ret = adl_audioStreamPlay( handle, ADL_AUDIO_PCM_MONO_8K_16B
      MyLowLevelIRQHandler, 0, StreamBuffer);
  }
```

```
Start AMR audio playing process

#define AMR_SIZE = 160

// audio resource handle
  s32 handle;

  // audio stream buffer
  void * StreamBuffer;

  // AMR samples
  u8 AMR_Samples[AMR_SIZE] = { ... , ... , ... , ... , ... , ... , 0 };      //
size of AMR audio sample to play = 160 bytes

  // Low level interruption handler
  bool MyLowLevelIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
                              Notification Level, adl_irqEventData_t * Data )
  {
      // copy PCM sample to play
      wm_strcpy( StreamBuffer, AMR_Samples );

      //...

      return FALSE;
  }

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;
      s32 BufferSize = AMR_SIZE;
      bool MixedOption = FALSE;
      adl_audioAmrCodecRate_e CodecRate = ADL_AUDIO_AMR_RATE_5_15;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler ,
                               ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Set Mixed voice option
      Ret = adl_audioSetOption ( handle, ADL_AUDIO_AMR_MIXED_VOICE,
                                 &MixedOption );
```

```
    // Set Codec Rate
    Ret = adl_audioSetOption ( handle, ADL_AUDIO_AMR_CODEC_RATE,
                              &CodecRate )

    // Memory allocation
    Ret = adl_audioSetOption ( handle, ADL_AUDIO_AMR_BUFFER_SIZE,
                              &BufferSize );
    StreamBuffer = adl_memGet( BufferSize );   // release memory after
                                                  audio stream playing

    // Play an audio AMR stream
    Ret = adl_audioStreamPlay( handle, ADL_AUDIO_AMR,
                               MyLowLevelIRQHandler, 0, StreamBuffer);
}
```

3.33.8.2. The adl_audioStreamListen Function

This function allows listening to a DTMF tone or an audio sample from microphone or voice call RX.

Listening to an audio sample stream implies that low level interrupt and/or high level interrupt have been previously subscribed (refer to adl_irqSubscribe for more information).

Moreover, memory space has to be allocated for the audio stream buffer before listening starts and it has to be released after listening stops.

Each time the interruption related to playing process occurs, recorded audio data has to be saved in allocated buffer in IRQ low or high level notification handler. Currently, this functions allows to record 4 audio formats:

- decoded DTMF (**ADL_AUDIO_DTMF**).
- audio signal sampled at 8KHz on 16 bits (**ADL_AUDIO_PCM_MONO_8K_16B**).
- audio signal sampled at 16KHz on 16 bits (**ADL_AUDIO_PCM_MONO_16K_16B**). **Only available on microphone**.
- audio signal compressed by an AMR or AMR-WB codec (**ADL_AUDIO_AMR**). Refer to adl_audioAmrCodecRate_e to get more information about available codecs.**Recording with AMR-WB codec is only available on microphone.**

DTMF decoding

Function allow to listen to a DTMF stream in Raw mode or in Pre-processed mode according to blank duration set initially. (refer to ADL_AUDIO_DTMF_DETECT_BLANK_DURATION for more information about Raw mode).

According to the mode of DTMF decoding, a different buffer size has to be allocated:

- for Raw mode , refer to ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE
- for Pre-processed mode, refer to ADL_AUDIO_DTMF_PROCESSED_STREAM_BUFFER_SIZE

Record PCM audio format

Before starting a PCM audio recording process, the application has to set embedded module audio configuration to define recording sample rate. Refer to the **AT+SPEAKER** command in "AT Commands Interface Guide" to get more information about audio resource configuration.

According to audio configuration a different space memory size will be allocated (use **adl_audioGetOption** function to get the size):

- refer to ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE for a sample rate at 8KHz
- refer to ADL_AUDIO_PCM_16K_16B_MONO_BUFFER_SIZE for a sample rate at 16KHz

Warning: *If allocated buffer size does not match with the sample rate, audio recording process may not work properly.*

Record AMR audio format

This function can record audio signal, compress it with AMR or AMR-WB codec and store it in RTP audio format (refer to RFC4867 to get more information about RTP format for AMR and AMR-WB). Before starting an AMR or AMR-WB audio recording process, the application has to set parameters such as codec rate (refer to adl_audioAmrCodecRate_e to get available codec rate), buffer size (refer to ADL_AUDIO_AMR_BUFFER_SIZE to get more information), mixed voice option (refer to ADL_AUDIO_AMR_MIXED_VOICE to get more information about it). According to the audio configuration, an audio signal could be compressed either with AMR codec or with AMR-WB codec:

- for an audio signal with a sample rate at 8 KHZ, an AMR codec has to be used
- for an audio signal with sample rate at 16 KHZ, an AMR-WB codec has to be used. **AMR-WB audio recording is only available on microphone** Refer to the **AT+SPEAKER** command in "AT Commands Interface Guide" to get more information about audio resource configuration. Buffer size, which has to be allocated, depends on the codec rate selected by application. For each codec rate, a minimal space memory size has to be allocated. Buffer size has to be either an audio AMR file size or multiple of one 20-ms audio AMR speech frame size, moreover one octet has to be allocated for frame header (this last one depends on codec rate).

Warning:

If allocated buffer size does not match with the codec rate, quality of played audio signal may be altered.

*When AMR audio file is listened on voice call and high level IRQ notification has been subscribed with **ADL_IRQ_OPTION_AUTO_READ** option (refer to adl_irq.h to get more information about this option) and audio buffer is too huge then a network de-registration may occur. In this case, **ADL_IRQ_OPTION_AUTO_READ** option should not be used or audio buffer size should be a small AMR speech frame size.*

Prototype

```
s32 adl_audioStreamListen ( s32                  audioHandle,
                            adl_audioFormats_e   audioFormat,
                            s32                  LowLevelIRQHandle,
                            s32                  HighLevelIRQHandle,
                            void *               buffer );
```

Parameters

audioHandle

Handle of the audio resource from which to listen the audio stream (microphone or voice call RX).

audioFormat

Stream audio format (refer to adl_audioFormats_e for more information).

LowLevelIRQHandle

Low level IRQ handle previously returned by IRQ subscription (please refer to adl_irqSubscribe for more information).

HighLevelIRQHandle

High level IRQ handle previously returned by IRQ subscription (please refer to adl_irqSubscribe for more information).

buffer

contains received decoded DTMF or audio samples.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.

- **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio signal is playing on the required audio resource.
- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for audio stream listening or if interrupt handler identifiers are invalid.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for audio stream listening.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note: *The **LowLevelIRQHandle** is optional if the **HighLevelIRQHandle** is supplied.*

Note: *The **HighLevelIRQHandle** is optional if the **LowLevelIRQHandle** is supplied.*

Note: *Each time an audio sample or DTMF sequence is detected, an interrupt handler will be notified to require the data. The interrupt identifier will be set to **ADL_IRQ_ID_AUDIO_RX_LISTEN** or **ADL_IRQ_ID_AUDIO_TX_LISTEN**, according to the resource used to start the stream listen.*

Note: *All audio filters will be deactivated for DTMF listening and only some audio filters for audio sample listening (refer to "audio command" chapter in the AT command Interface Guide 1 for more information).*

Note: *For audio interrupt subscription, **ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT** option is not available.*

Example

```
// audio resource handle
  s32 handle;

  // audio stream buffer
  void * StreamBuffer;


  // Low level interruption handler
  bool MyLowLevelIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
                              Notification Level, adl_irqEventData_t * Data )
  {
      TRACE (( 1, "DTMF received : %c, %c ", StreamBuffer[0],
               StreamBuffer[1] ));

      return FALSE;
  }
  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;
      s32 BufferSize;

      // Subscribe to the current microphone
      handle = adl_audioSubscribe ( ADL_AUDIO_MICROPHONE,
        MyAudioEventHandler , ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Memory allocation
      Ret = adl_audioGetOption ( handle,
                        ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE, &BufferSize )
      StreamBuffer = adl_memGet( BufferSize );   // release memory after
                                                    audio stream listening

      // Listen to audio DTMF stream
      Ret = adl_audioStreamListen( handle, ADL_AUDIO_DTMF
                                    MyLowLevelIRQHandler, 0, StreamBuffer);
  }
```

3.33.9. Stop

3.33.9.1. The adl_audioStop Function

This function allows to:

- stop playing a tone on the current speaker or on the buzzer,
- stop playing a DTMF on the current speaker or on the voice call TX,
- stop playing a melody on the current speaker or on the buzzer,
- stop playing an audio PCM stream on the current speaker or on the voice call TX,
- stop listening to an audio DTMF stream from current microphone or voice call RX,
- stop listening to an audio sample stream from current microphone or voice call RX.

ADL_AUDIO_EVENT_NORMAL_STOP event will not be sent to application.

Prototype

```
s32 adl_audioStop ( s32     audioHandle );
```

Parameters

audioHandle

Handle of the audio resource which has to stop its process.

Returned values

- **OK** on success.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_BAD_STATE** if no audio process is running on the required audio resource.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Example

```
// audio resource handle
  s32 handle;

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler ,
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a single tone
      Ret = adl_audioTonePlay( handle, 300, -10, 0, 0, 50 );

      // Stop playing the single tone
      Ret = adl_audioStop( handle );

      // unsubscribe to the current speaker
      Ret = adl_audioUnsubscribe ( handle );

  }
```

3.33.10. Set/Get options

3.33.10.1. The adl_audioSetOption Function

This function allows to set an audio option according to audio resource and option type specified. Several option types are only readable, so this function cannot be used with them (refer to adl_audioOptionTypes_e for more information).

Prototype

```
s32 adl_audioSetOption ( s32                       audioHandle,
                         adl_audioOptionTypes_e    audioOption,
                         s32                       value   );
```

Parameters

audioHandle

Handle of the audio resource.

audioOption

This parameter defines audio option to set (refer to adl_audioOptionTypes_e for more information).

value

Defines setting value for option.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_NOT_SUPPORTED** if the requested option is associated with a feature not available on the platform.

3.33.10.2. The adl_audioGetOption Function

This functions allows to get information about audio service according to audio resource and option type specified.

Prototype

```
s32 adl_audioGetOption ( s32                       audioHandle,
                         adl_audioOptionTypes_e    audioOption,
                         s32 *                     value );
```

Parameters

audioHandle

Handle of the audio resource.

audioOption

audio option which wishes to get information (refer to adl_audioOptionTypes_e for more information).

value

option value according to audio option which has been set.

Returned values

- value option value according to audio option which has been set.
- **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
- **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- **ADL_RET_ERR_NOT_SUPPORTED** if all features associated to the option are not available

3.34. ADL Secure Data Storage Service

The ADL supplies Secure Data Storage Service interface to

- read/write/query data stored in ciphered format in non volatile memory,
- update cryptographic keys in order to block replay/re-download attacks.

The defined operations are:

- An **`adl_sdsWrite`** function to write secured data.
- An **`adl_sdsRead`** function to read secured data.
- An **`adl_sdsQuery`** function to require size of one of secured entries.
- An **`adl_sdsDelete`** function to delete one of secured entries.
- An **`adl_sdsStats`** function to get statistics about secured data storage.
- An **`adl_sdsUpdateKeys`** function to update the cryptographic keys.

Note: *These functions are available only if:*
- they are used with a compatible platform.
- the Secured Data Storage feature is properly activated on the production line
- the objects are not erased, otherwise embedded module has to be returned in production line
Otherwise, every function cited above will return the error code ***`ADL_RET_ERR_NOT_SUPPORTED.`***

Note: *Secure Data Storage is only available on AirPrime WMP Series modules*

3.34.1. Required Header File

The header file for the functions dealing with the ADL Secure Data Storage Service public interface is:

`adl_sds.h`

3.34.2. Data Structure

3.34.2.1. The adl_sdsStats_t Structure

Data storage statistics contains information about secured data storage. It has to be used with adl_sdsStats API. .

Code

```
typedef struct
{
        u32     MaxEntrySize
        u32     FreeSpace
        u32     TotalSpace
        u16     MaxEntry
        u16     EntryCount
}adl_sdsStats_t;
```

Description

MaxEntrySize

Maximal size of one secured entry. It is defined in number of bytes.

FreeSpace

Available space for secured entries.

Warning: *This figure does not depend only on written data but depends on the state of the underlying storage media too. It might increase or decrease as data entries sharing the same space as ciphered entries are created or deleted.*

TotalSpace

Total space allocated for ciphered entries. This figure is a quota, and must be treated as such. Because ciphered entries share storage media with other information, this quota might be unaccessible if, for example, the underlying storage medium is near its full capacity.

MaxEntry

Maximal number of secured entry.

Note: *The maximal number of secured entries depends on the underlying storage service. There might be less available entries if this storage service is near its maximum capacity.*

EntryCount

Total number of secured entries.

3.34.3. Defines

3.34.3.1. ADL_SDS_RET_ERR_ENTRY_NOT_EXIST

Entry does not exist.

```
#define ADL_SDS_RET_ERR_ENTRY_NOT_EXIST    ADL_RET_ERR_SPECIFIC_BASE
```

3.34.3.2. ADL_SDS_RET_ERR_MEM_FULL

Not enough space memory to write secured data.

```
#define ADL_SDS_RET_ERR_MEM_FULL      ADL_RET_ERR_SPECIFIC_BASE - 1
```

3.34.4. The adl_sdsWrite Function

This function allows to store data in a secured entry, data are ciphered. This function creates a new entry or updates an existing one.

Prototype

```
s32 adl_sdsWrite   (   u32        ID,
                       u32        Length,
                       void *     Source );
```

Parameters

ID:

Numeric ID of the entry. The ID range is from 0 to **MaxEntry** (returned by **`adl_sdsStats`**). Refer to adl_sdsStats_t to get more information about **MaxEntry**.

Length

Size of the data to write in the entry. Use **`adl_sdsStats`** to get the maximum size for one secured entry (refer to **MaxEntrySize** in adl_sdsStats_t to get more information).

Source

Pointer to the source buffer. It contains data to write.

Returned values

- **`OK`** on success
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if parameters have an incorrect value.
  - **`ADL_SDS_RET_ERR_MEM_FULL`** no enough memory is available for writing.
  - **`ADL_RET_ERR_NOT_SUPPORTED`** writing operation is not available.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler.

3.34.5. The adl_sdsRead Function

This function allows to retrieve data from a secured entry. Data which has been previously written with **`adl_sdsWrite`** API.

Prototype

```
s32 adl_sdsRead (      u32                 ID,
                       u32                 Offset,
                       u32                 Length,
                       void *              Destination );
```

Parameters

ID:

Numeric ID of the entry. The ID range is from 0 to **MaxEntry** (returned by **`adl_sdsStats`**). Refer to adl_sdsStats_t to get more information about **MaxEntry**.

Offset

Offset in the secured entry, defined in number of bytes. It allows to retrieve a part of the entry. It is an offset in relation to the first byte of the entry.

Length

Size of data to read in the secured entry. Use **`adl_sdsQuery`** API to get the maximal length for the required entry.

Destination

Pointer to the destination buffer. It contains data to retrieve.

Returned values

- **`OK`** on success
- A negative error value otherwise:
  - **`ADL_RET_ERR_PARAM`** if parameters have an incorrect value.
  - **`ADL_SDS_RET_ERR_ENTRY_NOT_EXIST`** if entry ID does not exist.
  - **`ADL_RET_ERR_NOT_SUPPORTED`** reading operation is not available.
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler.

3.34.6. The adl_sdsQuery Function

This function allows to check if a secured entry exists and gets its size.

Prototype

```
s32 adl_sdsQuery (    u32        ID,
                      u32*       Length );
```

Parameters

ID:

Numeric ID of the entry. The ID range is from 0 to **MaxEntry** (returned by **`adl_sdsStats`**). Refer to adl_sdsStats_t to get more information about **MaxEntry**.

Length

Output pointer for the entry size. It can be set to NULL.

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if parameters have an incorrect value.
  - `ADL_SDS_RET_ERR_ENTRY_NOT_EXIST` if entry ID does not exist.
  - `ADL_RET_ERR_NOT_SUPPORTED` operation is not available.
  - `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interrupt handler.

3.34.7. The adl_sdsDelete Function

This function allows to delete one of existing entries.

Prototype

```
s32 adl_sdsDelete (   u32    ID );
```

Parameters

ID:

Numeric ID of the entry. The ID range is from 0 to **MaxEntry** (returned by **`adl_sdsStats`**). Refer to adl_sdsStats_t to get more information about **MaxEntry**.

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if parameters have an incorrect value or secured entry does not exist.
  - `ADL_SDS_RET_ERR_ENTRY_NOT_EXIST` if entry ID does not exist.
  - `ADL_RET_ERR_NOT_SUPPORTED` deletion operation is not available.
  - `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interrupt handler.

3.34.8. The adl_sdsStats Function

This function allows to retrieve information about secured data storage as free memory space or total memory space.

Prototype

```
s32 adl_sdsStats ( adl_sdsStats_t*    Stats );
```

Parameters

Stats:

Pointer on statistical information of secured data storage. (refer to adl_sdsStats_t to have more information about statistics).

Returned values

- `OK` on success
- A negative error value otherwise:
  - `ADL_RET_ERR_PARAM` if parameters have an incorrect value.
  - `ADL_RET_ERR_NOT_SUPPORTED` operation is not available.
  - `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interrupt handler.

3.34.9. The adl_sdsUpdateKeys Function

This function allows to re-generate the internal cryptographic keys. This function has to be used to defeat possible replay or re-download attacks.
Once the keys are re-generated, all the stored data remain available and still readable by application, but the processor will not be able to re-use a previous image of the non-volatile memory with old cryptographic keys.

Prototype

```
s32 adl_sdsUpdateKeys ( void );
```

Note: *This function is synchronous and its exectution time is independent of the number of entries.*

Warning: *This must be used with caution because of the limited life expectancy of the non-volatile memory implied in this process. For example, a WMP100 processor can, at most, withstand 2x10^6 key changes: changing them every second would therefore wear out the processor after 1.5 year.*

Returned values

- **OK** on success
- A negative error value otherwise:
  - **ADL_RET_ERR_NOT_SUPPORTED** updating operation is not available.
  - **ADL_RET_ERR_FATAL** EEPROM cannot be written.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.10. Example

The code sample below illustrates a nominal use case of the ADL Secure Data Storage Service public interface (error cases are not handled).

```
// ...
// decrement counter
u32 n=10;
u32 size;
u32 offset=0;
adl_sdsWrite( COUNTER_ID, offset, sizeof(u32), &n );
adl_sdsQuery( COUNTER_ID, &size );
adl_sdsRead( COUNTER_ID, offset, size, &n );
n--;
adl_sdsWrite( COUNTER_ID, size, &n );

// ensure that from now on, any previously
// stored memory image becomes incompatible
// with this processor
adl_sdsUpdateKeys();
// ...

adl_sdsRead( COUNTER_ID, offset, sizeof(u32), &n );
// delete entry
adl_sdsDelete( COUNTER_ID );
```

3.35. ADL WatchDog Service

ADL provides a watchdog service to access to the embedded module's WatchDog. There are 2 watchdogs: The Firmware watchdog (also called the hardware watchdog) and the Software watchdog (also called the application watchdog). The software watchdog is unique, meaning that there is only one and that it may be armed by one task and rearmed by another task. All applications tasks share one software watchdog. The watchdog duration is absolute and not a function of the application CPU use. The hardware and software watchdogs are independent. Either may expire first.

Note: *The timing unit is a tick which corresponds to 18.5 ms.*

- Hardware watchdog put to sleep

Because an application may launch heavy treatments that can take more than the hardware watchdog duration (one minute for example) and because the watchdog cannot be stopped once it had been started, system provides a way to deactivate the hardware watchdog from the application point of view for a given time. In fact, during this time, system rearms by itself the hardware watchdog application in a high priority task because the IDLE task cannot take the focus while the application treatments are not finished.

The defined operations are:

- A **`adl_wdPut2Sleep`**
- A **`adl_wdAwake`**

- Application watchdog Management

Application watchdog can be activated with a given duration. Once the application watchdog is activated, the application binary has to rearm regularly the application watchdog to indicate that it is still alive. Else, a back trace is generated and a reset occurs. Application watchdog can be deactivated or reactivated with a new duration.

The defined operations are:

- A **`adl_wdRearmAppWd`**
- A **`adl_wdActiveAppWd`**
- A **`adl_wdDeActiveAppWd`**

3.35.1. Required Header File

The header file for the functions dealing with the ADL WatchDog Service public interface is:

`adl_wd.h`

3.35.2. The adl_wdPut2Sleep Function

This function enables to launch an automatic hardware watchdog relaunch for a given duration. Thanks to this function, during the watchdog sleep duration, application treatments can take more than hardware watchdog duration even if IDLE task cannot have the CPU focus for more than hardware watchdog duration. Once the sleep duration expired, the IDLE task must receive back the CPU focus in less than the hardware watchdog duration, else a watchdog reset occurs.

Note: *This must be called just before an heavy treatment to avoid watchdog reset. The argument has to be strictly positive.*

Prototype

```
u32 adl_wdPut2Sleep ( u32      i_u32_SleepDuration );
```

Parameters

i_u32_SleepDuration:

Watchdog sleep duration in number of ticks (timer macro **ADL_TMR_S_TO_TICK(SecT)** - can be used for duration conversion).

Returned values

- **OK** or **ADL_RET_ERR_PARAM** if wrong argument.

3.35.3. The adl_wdAwake Function

The adl_wdAwake function enables to cancel watchdog inactivation.

Note: *This should be called just after an heavy treatment if watchdog had been inactivated to force the restore of default behavior. If not called, default behavior will be restored automatically at the expiration of watchdog sleep duration.*

Prototype

```
u32 adl_wdAwake ( void );
```

Returned values

- Remaining time before automatic watchdog reactivation in number of ticks.

3.35.4. Example

Here is an example of how to use the watchdog API access functions.

```
void CallMyHeavyTreatpments(void)
  {
     // To store remaining time before the end of watchdog inactivation
     u32 i_u32_ReaminingTime;

     // Watchdog inactivation for 30 seconds
     adl_wdPut2Sleep(ADL_TMR_S_TO_TICK(30));

     // Watchdog reactivation
     i_u32_ReaminingTime = adl_wdAwake();

     printf("Watchdog is to be awaken in %d number of ticks",
     i_u32_ReaminingTime );
  }
```

3.35.5. The adl_wdRearmAppWd Function

Enable to rearm the application watchdog with the stored watchdog duration.

Note: *Application can use a cyclic timer to regularly rearm the application watchdog.*
OK is returned and nothing happens if ***adl_wdActiveAppWd*** *has not been called before.*

Prototype

```
s32 adl_wdRearmAppWd ( void );
```

Returned values

- **OK** or **ADL_RET_ERR_NOT_SUPPORTED** if the watchdog service is not supported.

3.35.6. The adl_wdActiveAppWd Function

Once started, application watchdog must be rearmed regularly (no matter how) to indicate that it is still alive. If the watchdog timer expired, the hardware watchdog will not be rearmed anymore and the embedded module's will reset.

Prototype

```
s32 adl_wdActiveAppWd ( u32  i_u32_Duration );
```

Note: *Argument has to be strictly positive.*

Parameters

i_u32_Duration:

Software application watchdog duration in number of ticks (timer macro ADL_TMR_S_TO_TICK(SecT) - can be used for duration conversion).

Returned values

- **OK**
- **ADL_RET_ERR_PARAM** on parameter error
- **ADL_RET_ERR_NOT_SUPPORTED** if the watchdog service is not supported.

3.35.7. The adl_wdDeActiveAppWd Function

The **adl_wdDeActiveAppWd** function enables to stop watchdog.

Note: *OK is returned and nothing happens if **adl_wdActiveAppWd** has not been called before.*

Prototype

```
s32 adl_wdDeActiveAppWd ( void );
```

Returned values

- **OK** or **ADL_RET_ERR_NOT_SUPPORTED** if the watchdog service is not supported.

3.35.8. Example

Here is an example of how to use the application watchdog API access functions.

```
void CallMyHeavyAppliTreatpments(void)
{
   adl_tmr_t *tt;

   // Lets activate the application watchdog for 30 seconds
   adl_wdActiveAppWd(ADL_TMR_S_TO_TICK(30));

   // Lets suscribe to a 25 sec timer
   tt = (adl_tmr_t *)adl_tmrSubscribe (TRUE,
                                       25,
                                       ADL_TMR_TYPE_100MS,
                                       (adl_tmrHandler_t)Timer_Handler);

   // Launch heavy appli treatment
   MyHeavyAppliTreatemnt();
}

void Timer_Handler( u8 Id, void * Context )
{
   if ( (process has not ended)
   {
       if (there is some activities)
       {
           // Rearm the application watchdog for another go
           adl_wdRearmAppWd();
       }
       else
       {
           // the process has not ended and there is no activities ->
             application watchdog reset
       }
   }
   else // process has ended
   {
       // the process has ended we can now deactivate the application
          watchdog
       adl_wdDeActiveAppWd();
   }
}
```

3.36. ADL Layer 3 Service

The ADL supplies Layer3 Service interface allows to get information about Layer 3 as PLMN scan information.

The defined operations are:

- A **`adl_L3infoSubscribe`** function to subscribe to the L3 information service
- A **`adl_L3infoUnsubscribe`** function to unsubscribe to the L3 information service.

Note: *The L3 layer interface is not available on the AirPrime Q26Extreme module*

3.36.1. Required Header File

The header file for the functions dealing with the ADL Layer 3 Service public interface is:

```
adl_L3info.h
```

3.36.2. The adl_L3infoChannelList_e

List of available information channel.

Code

```
typedef enum
{
        ADL_L3INFO_SCAN
        ADL_L3INFO_CELL
        ADL_L3INFO_RSM
}adl_L3infoChannelList_e;
```

Description

ADL_L3INFO_SCAN

This channel allows to retrieve information about PLMN Scan:

- power min, max, average
- cell synchronization. Refer to Channel Identity for more details on information structure, which are returned by Scan channel

ADL_L3INFO_CELL

This channel allows to retrieve information about current cell and proximate cells. Refer to wm_l3info_Cell_SyncCellInfo_t for more details on information structure, which are returned by CELL channel.

ADL_L3INFO_RSM

This channel allows to retrieve RSM information which is reported once PLMN scan is finished. Refer to wm_l3info_RSM_freq_t for more details on information structure, which are returned by RSM channel.

Note: *Some L3INFO channels are not defined in* ***`adl_L3infoChannelList_e`*** *but they are used by the firmware and Open AT® application cannot access to them. So when application subscribes to a channel which is not defined in* ***`adl_L3infoChannelList_e`**,* *then a valid handle (positive or NULL value) will be returned instead of "ADL_RET_ERR_PARAM" in some cases.*

3.36.3. The Layer3 infoEvent Handler

Such a call-back function has to be supplied to ADL through the adl_L3infoSubscribe interface in order to receive L3 information according to channels and related events.

Prototype

```
typedef void(*)adl_L3infoEventHandler_f( u32                      Time,
                                         adl_L3infoChannelList_e  ChannelId,
                                         u32                      EventId,
                                         u32                      Length,
                                         void *                   Info );
```

Parameters

Time

Reserved for Future Use.

ChannelId

Channel identity which provides information. (Refer to adl_L3infoChannelList_e for more information).

EventId

Event identity according to ChannelId. Refer to l3info_trace for more information about possible event.

Length

Length of "Info" content.

Info

Information content according to ChannelID and EventID. Refer to l3info_trace for more information about the type of "Info".

3.36.4. The adl_L3infoSubscribe Function

This function allows to subscribe several times to one of the available information channel of the Layer 3.

A call-back function is associated for Layer 3 events and to retrieve information relative to the channel requested.

Prototype

```
s32 adl_L3infoSubscribe  (     adl_L3infoChannelList_e    ChannelId,
                               adl_L3infoEventHandler_f   L3infoHandler );
```

Parameters

ChannelId

Information channel requested. (Refer to adl_L3infoChannelList_e for more information).

L3infoHandler

Application provides Layer 3 event call-back function (Refer to **`adl_L3infoEventHandler_f`** for more information).

Returned values

- **Positive or NULL** if allocation succeed, returns handle which has to to be used on further L3 info API functions calls
- **`ADL_RET_ERR_PARAM`** if parameter has an incorrect value.
- **`ADL_RET_ERR_NOT_SUPPORTED`** if the Raw Spectrum Information feature is not enabled on the embedded module.

- `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interruption handler.

3.36.5. The adl_L3infoUnsubscribe Function

This function allows to unsubscribe from the specific channel L3 information flow which has been subscribed previously with **adl_L3infoSubscribe** function.

Prototype

```
s32 adl_L3infoUnsubscribe   (   u32    Handle );
```

Parameters

Handle

handle previously returned by **adl_L3infoSubscribe** function.

Returned values

- **OK** on success
- `ADL_RET_ERR_UNKNOWN_HDL` if the provided handle is unknown.
- `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interruption handler.

3.36.6. Example

These function allows to subscribe or unsubscribe to one of information channel available from Layer 3.

```
// Channel info handle
  s32 handle;

  // info channel event call-back function
  void MyChannelEventHandler( u32 Time, adl_L3infoChannelList_e ChannelId,
  u32 EventId, u32 Length, void * Info )
  {

      switch ( EventId)
      {
          ...
      }

      adl_L3infoUnsubscribe( handle );

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {

      // Subscribe to PLMN Scan channel information
      handle = adl_L3infoSubscribe ( ADL_L3INFO_SCAN,
                                     MyChannelEventHandler);

  }
```

3.36.7. Channel Identity List

Channel Identity list.

Note: Only PLMN Scan and Cell Information Channels are opened in ADL.

3.36.7.1. The l3info_ChannelList_t

Code

```
typedef enum
{
      L3INFO_SCAN          //PLMN scan information
      L3INFO_CELL          //Cell information
      BATT_CHANNEL         //BATT channel information [Internal use]
      SMS_CHANNEL          //SMS information [Internal use]
      DATA_INFO_CHANNEL    //Data information [Internal use]
      CELL_INFO_CHANNEL    //Cell Information channel [Internal use]
      OAT_CHANNEL          //OAT Information channel [Internal use]
      L3INFO_NBCHANNEL     //Number of channel
  }l3info_ChannelList_t;
```

3.36.8. Cell Information Channel Interface

This section describes events and associated data structure to provide information about serving cell and surrounding cells.

3.36.8.1. Cell Information [WM_L3_INFO_SYNC_CELL_INFO event]

The Synchronized cell nformation is reported every 5 seconds if embedded module is under GSM coverage.

3.36.8.2. WM_Cx_NOT_AVAILABLE

WM_Cx_NOT_AVAILABLE define.

if not C1, C2, C31 or C32 is not available

```
#define WM_BSIC_NOT_AVAILABLE  0x40
```

3.36.8.3. WM_BSIC_NOT_AVAILABLE

if BSIC not available.

```
#define WM_BSIC_NOT_AVAILABLE  0xFF
```

3.36.8.4. WM_L3_INFO_SYNC_CELL_INFO

Synchronized Cell Information event identity.

```
#define WM_L3_INFO_SYNC_CELL_INFO   0
```

3.36.9. PLMN SCAN Information Channel Interface

This section describes events and associated data structure to provide information about PLMN SCAN procedure.

The PLMN Scan procedure is composed by the following steps :

- At first a power measurement on each supported frequency is performed.
- Then if sufficent power (> noise power level(~ -105dBm)) is detected on one cell or more , cell synchronisation attempt is performed on these cells.

The PLMN scan procedure can be initiated by the embedded module, for initial PLMN selection or automatic PLMN reselection purposes, or can be initiated by the user with AT+COPS command for instance.

3.36.9.1. Measurements Information [WM_L3_INFO_SCAN_PWR event]

The Measurement information are reported each time a power measurement is required on all frequencies.

The corresponding reported data are statistics on the low band, high band and low+high band.

The total number of cells with a power level greater than the noise power level is also reported.

3.36.9.2. Cell Synchronisation Information [WM_L3_INFO_SCAN_SYNC_CELL event]

The Cell Sychronisation information are reported when a cell synchronisation attempt was executed during the PLMN Scan procedure and

- if the embedded module is not camped on a cell (the number of synchro failure is updated)
- if the embedded module has just camped on a cell (CellCamped flag set): no other WM_L3_INFO_SCAN_SYNC_CELL event is reported after.

3.36.9.3. Cell Information [WM_L3_INFO_CELL_INFO event]

The Cell Information are reported each time a cell is synchronized during the scan procedure.

3.36.9.4. Scan end Information [WM_L3_INFO_SCAN_END event]

This event is reported once the scan is finished.

3.36.10. PLMN SCAN Information Channel : Event List

3.36.10.1. WM_L3_INFO_SCAN_PWR

Power level information event identity.

```
#define WM_L3_INFO_SCAN_PWR   0
```

3.36.10.2. WM_L3_INFO_SCAN_SYNC_CELL

Cell Synchronisation information event identity.

```
#define WM_L3_INFO_SCAN_SYNC_CELL  1
```

3.36.10.3. WM_L3_INFO_SCAN_END

Scan ended.

```
#define WM_L3_INFO_SCAN_END   2
```

3.36.10.4. WM_L3_INFO_CELL_INFO

Cell Information event identity.

```
#define WM_L3_INFO_CELL_INFO  3
```

3.36.11. Radio Spectrum Monitoring (RSM) Channel Interface

This section describes events and associated data structure to provide information about Radio environment.

RSM information is updated and reported at each PLMN scan (initiated by the user using AT+COPS=? command or initiated by the embedded module itself) .

3.36.11.1. Cell Information [WM_L3_INFO_CELL_INFO event]

The RSM information is reported once the PLMN scan is finished. The RSM information is composed for each frequency:

- Rxlev (0 to 63)
- Synchronized Status (synchronized, synchronisation failed, synchronisation not tried)
- if Synchronized status is equal to synchronized:
  - BSIC
  - Location Area Information (MCC / MNC / LAC)
  - Cell identity (equal to 0xFFFF if unknown)

3.36.11.2. WM_L3_INFO_RSM_EVT

RSM event identity.

```
#define WM_L3_INFO_RSM_EVT   0
```

3.36.11.3. WM_L3_INFO_RSM_EVT event

Maximum number of frequency.

```
#define L3INFO_MAX_NB_RSM_FREQ   971
```

3.36.12. Layer 3 Information Status

Status or Error returned by any Layer 3 function.

3.36.12.1. L3INFO_ERR_CHANNEL_UNKNOWN

Unknown Channel Identity.

```
#define L3INFO_ERR_CHANNEL_UNKNOWN   ((s32) (-1))
```

3.36.12.2. L3INFO_ERR_CHANNEL_ALREADY_OPENED

Channel already opened.

```
#define L3INFO_ERR_CHANNEL_ALREADY_OPENED   ((s32) (-2))
```

3.36.12.3. L3INFO_ERR_CHANNEL_ALREADY_CLOSED

Scan ended.

```
#define L3INFO_ERR_CHANNEL_ALREADY_CLOSED   ((s32) (-3))
```

3.36.12.4. L3INFO_ERR_INVALID_HANDLE

Invalid Handle.

```
#define L3INFO_ERR_INVALID_HANDLE   ((s32) (-4))
```

3.36.12.5. L3INFO_OK

Successful operation.

```
#define L3INFO_OK   ((s32) (-0))
```

3.36.13. Function interface for information provider

This function is used by any Software Element providing information on a defined channel.

3.36.13.1. The l3info_trace Function

l3info_trace : Event trace function.

This function is called each time a event shall be reported whatever the channel state is (open, closed).

Prototype

```
void l3info_trace (   l3info_ChannelList_t    ChannelId,
                      u32                     EventId,
                      u32                     Length,
                      u8*                     Ptr );
```

Parameters

ChannelId

Channel Identity

EventId

Event Identity

Length

Length of the information content

Ptr

Information content

3.36.13.2. The l3info_IsChannelActivated Function

l3info_IsChannelActivated : Channel status.

This function returns the channel state (open, closed).

Prototype

```
bool l3info_IsChannelActivated  (  l3info_ChannelList_t   ChannelId );
```

Parameters

ChannelId

Channel Identity

Returns

- TRUE if channel is open, otherwise, FALSE.

3.36.14. User Interface

The User Interface is composed by:

- a subscribe function. At subscription, the user shall provide a callback function
  - This callback function will be used each time information has to be reported.
  - This callback function shall follow the _pl3infoCallBackProto function prototype described below.
- an unsubscribe function.

3.36.14.1. The l3info_infoSubscribe Function

l3info_infoSubscribe : Layer 3 information channel subscription function.

Prototype

```
s32 l3info_infoSubscribe (  l3info_ChannelList_t      ChannelId,
                            _pl3infoCallBackProto*    pFunc );
```

Parameters

ChannelId

Channel Identity

pFunc

Callback function pointer

Return value

- Handle (positive value) or negative value if error

3.36.14.2. The l3info_infoUnSubscribe Function

l3info_infoUnSubscribe : Layer 3 information channel subscription function.

Prototype

```
s32 l3info_infoUnSubscribe  (   s32    Handle );
```

Parameters

Handle

Handle of the channel to close

Return value

- **L3INFO_OK** if OK or negative value if invalid handle.

3.36.14.3. The _pl3infoCallBackProto Function

CallBack function prototype.

Prototype

```
void _pl3infoCallBackProto (   u32                    Time,
                               l3info_ChannelList_t   ChannelId,
                               u8                     EventId,
                               u32                    Length,
                               u8                     *Ptr );
```

Parameters

Time

Not used

ChannelId

Channel Identity

EventId

Event Identity

Length

Length of the information content

Ptr

Information content

3.36.15. Layer 3 Information Interface Specification Data Structures

3.36.15.1. The wm_l3info_Cell_SyncCellInfo_t Structure

Synchronized Cell Information structure.

This information is reported every 5 seconds, or when a first cell is synchronized.

Code

```
typedef struct
{
      u8                                   NbSyncCell,
      u8                                   Pad[3],
      wm_l3info_Cell_SyncCellParameter_t   SyncCell[7]
}wm_l3info_Cell_SyncCellInfo_t;
```

Description

NbSyncCell

Number of synchronized cell.

SyncCell[7]

Synchronized cell information (First serving cell then neighbor cells)

3.36.15.2. The wm_l3info_Cell_SyncCellParamater_t Structure

Synchronized Cell Parameter Information structure.

Code

```
typedef struct
{
      u16    Arfcn,
      u8     Rssi,
      u8     Lai[5],
      u8     CellId[2],
      u8     Bsic,
      s8     C1,
      s16    C2,
      s16    C31,
      s16    C32,
      bool   GprsIndication,
      u8     MsTxPwrMaxCcch
}wm_l3info_Cell_SyncCellParamater_t;
```

Description

Arfcn

ARFCN

Rssi

RSSI: Range [0 to 63]

Lai[5]

Location area identity: including MCC, MNC and LAC.

--8--7--6--5-|-4--3--2--1
Byte 1 :MCC digit 2 | MCC digit 1

Byte 2 :MNC digit 3 | MCC digit 3

Byte 3 :MNC digit 2 | MNC digit 1

Byte 4 :LAC

Byte 5 :LAC (cont)

CellId[2]

Cell identity

Bsic

Base Station Identity code.

C1

C1 value: cell selection criteria (only available in idle mode - **`WM_Cx_NOT_AVAILABLE`** if not available).

C2

C2 value: GSM cell reselection criteria (only available in idle mode - **`WM_Cx_NOT_AVAILABLE`** if not available).

C31

C31 value: GPRS cell reselection criteria (only available in idle mode - **`WM_Cx_NOT_AVAILABLE`** if not available).

C32

C32 value: GPRS cell reselection criteria (only available in idle mode - **`WM_Cx_NOT_AVAILABLE`** if not available).

GprsIndication

Gprs support indication.

MsTxPwrMaxCcch

Power control level: The maximum TX power level an MS may use when accessing a Control Channel CCH. (Range: 0 to 31).

3.36.15.3. The wm_l3info_RSM_freq_t Structure

Frequency information structure.

Code

```
typedef enum
{
        L3INFO_FREQ_NOT_TRIED             //No synchronisation performed in this
                                          frequency
        L3INFO_FREQ_SYNCHRONIZED          //GSM cell found on this frequency
        L3INFO_FREQ_NOT_SYNCHRONIZED      //No GSM cell found on this frequency
}
```

Code

```
typedef struct
{
      u16     Arfcn
      u8      Rxlev
      u8      Bsic
      u8      Lai[4]
      u16     CellIdentity
}wm_l3info_RSM_freq_t;
```

Description

Arfcn

ARFCN.

Rxlev

Rx Level (0 to 63).

Bsic

Base Station Identity Code.

Lai[4]

Location area identity : including MCC, MNC and LAC.

--8--7--6--5-|-4--3--2--1
Byte 1 :MCC digit 2 | MCC digit 1

Byte 2 :MNC digit 3 | MCC digit 3

Byte 3 :MNC digit 2 | MNC digit 1

Byte 4 :LAC

CellIdentity

Cell Identity.

3.36.15.4. The wm_l3info_RSM_t Structure

RSM information.

Code

```
typedef struct
{
      u16                    NumberOfFrequency
      u16                    Pad
      wm_l3info_RSM_freq_t   FreqInfo[L3INFO_MAX_NB_RSM_FREQ]
}wm_l3info_RSM_t;
```

Description

NumberOfFrequency

Number of frequency reported.

FreqInfo[L3INFO_MAX_NB_RSM_FREQ]

RSM table information.

3.36.15.5. The wm_l3info_Scan_PowerInfo_t Structure

Power Measurement Information structure.

Code

```
typedef struct
{
      wm_l3info_Scan_PowerStat_t    Total,
      wm_l3info_Scan_PowerStat_t    LowBand,
      wm_l3info_Scan_PowerStat_t    HighBand,
      u16                           NumberOfCellAboveNoise,
      bool                          CellCamped
}wm_l3info_Scan_PowerInfo_t;
```

Description

Total

Power Measurement statistics for all bands.

LowBand

Power Measurement statistics for the low band (GSM/850).

HighBand

Power Measurement statistics for the high band (DCS/PCS).

NumberOfCellAboveNoise

Number of cells with a power level greater than the noise's one.

CellCamped

TRUE if embedded module is camped on a cell, else FALSE.

3.36.15.6. The wm_l3info_Scan_PowerStat_t Structure

Power Measurement structure.

Code

```
typedef struct
{
      u32     NbFreq
      u8      Min
      u8      Max
      u8      Mean
      u32     Variance
}wm_l3info_Scan_PowerStat_t;
```

Description

NbFreq

Number of frequencies.

Min

Minimal power level detected.

Max

Maximal power level detected.

Mean

Mean power level.

Variance

Variance.

3.36.15.7. The wm_l3info_Scan_SynchroCellInfo_t Structure

Cell Synchronization Information structure.

This information is reported each time a cell synchronisation is unsucessful and no cell has been already synchronised, or when a first cell is synchronised.

Code

```
typedef struct
{
      u16     NbCellTriedInLowBand,
      u16     NbCellTriedInHighBand,
      bool    CellCamped
}wm_l3info_Scan_SynchroCellInfo_t;
```

Description

NbCellTriedInLowBand

Number of tried cell in low band since the start of the scan.

NbCellTriedInHighBand

Number of tried cell in high band since the start of the scan.

CellCamped

TRUE if embedded module is camped on a cell, else FALSE.

3.36.15.8. The wm_l3info_Scan_End_t structure

End Scan Information structure.

These information is reported at the end of the scan procedure

Code

```
typedef struct
{
      bool    CellCamped
} wm_l3info_Scan_End_t;
```

Description

CellCamped

TRUE if embedded module is camped on a cell, else FALSE.

3.36.15.9. The wm_l3info_CellInfo_t structure

Cell Information structure.

These information is reported each time a cell is synchronised during a scan.

Code

```
typedef struct
{
      u16     Arfcn
      u16     CellId
      u8      Rssi
      u8      Lai[3]
} wm_l3info_CellInfo_t;
```

Description

Arfcn

Cell Frequency (Arfcn)

CellId

Cell Identity

Rssi

RSSI on the corresponding frequency

Lai[3]

Cell PLMN (MCC/MNC coded as in 3GPP 04.18)

3.37. ADL Event Service

ADL provides an Event service to access to the embedded module's Event.

Please make a note that the timing unit is a tick which corresponds to 18.5 ms.

Events are communication objects between tasks and interrupt routines (ISRs) or between tasks and other tasks.

Dynamic Event creation:

- Events are dynamically created, please refer to adl_eventCreate API
- A handle is returned at Event creation to handle it

Wait for an Event:

- A task is allowed to wait for an Event under given conditions, using **adl_eventWait** API
- If the Event wait condition is already TRUE, the task continues its execution and its context is ADL_CTX_STATE_WAIT_EVENT
- Else the task status becomes **ADL_CTX_STATE_WAIT_INNER_EVENT** (wait for Event) and the task cannot be scheduled when it is in this state
- A task which is waiting for an Event that is set becomes **ADL_CTX_STATE_READY** and can be scheduled again
- In fact, several tasks can wait for the same Event. In such case, they will all be in **ADL_CTX_STATE_WAIT_EVENT** state
- If more than one task is waiting for an Event and when this Event is set, all the tasks waiting for that Event can be reactivated (depending on the selected wait mode / see below)
- If several tasks are waiting for an Event that is set, those tasks become **ADL_CTX_STATE_READY** and will be scheduled (states become **ADL_CTX_STATE_ACTIVE**) according to their priorities

Event Wait condition:

- When a task calls the **adl_eventWait** API to wait for an Event, it provides a mask to be compared with the internal Event bit field
- The task stays in the **ADL_CTX_STATE_WAIT_INNER_EVENT** status, while the wait condition is not TRUE
- There are 2 waiting modes:
  - **ADL_EVENT_WAIT_ANY**: The wait condition is TRUE if at least one bit of the provided mask matches with the Event internal bit field
  - **ADL_EVENT_WAIT_ALL**: The wait condition is TRUE if all bits of the provided mask match with the Event internal bit field

Event set and clear:

- An Event can be set or cleared by a task or an interrupt routine (please refer to adl_eventSet and adl_eventClear API)
- When an Event is set, tasks that are waiting for this Event can be reactivated (**ADL_CTX_STATE_READY** state)

Event internal mask:

- An Event contains an internal bit field which is a private attribute
- Initial value of this internal mask is provided at the creation of the Event
- A task that is waiting for an Event waits, in fact for one or several bits of the internal bit field to be raised
- It is possible to set or to clear each bit of the internal bit field individually or in group, please refer to adl_eventSet and adl_eventClear API

- **adl_eventSet** API sets one or several bits and can make **ADL_CTX_STATE_READY** one or several tasks that are waiting for this bits to be raised
- **adl_eventClear** API clears one or several bits

Note about Event versus Semaphore:

- Whereas semaphore production are not allowed before consumption, all API of the Event service can be used on a given Event, whatever the order is (on condition the Event is firstly created)
- Hence **adl_eventWait** / **adl_eventSet** and **adl_eventClear** functions can be called in any order
- If more than one task is waiting for a semaphore and when this semaphore is produced (in a task context for example), then only the task with higher priority (among all the tasks that are waiting for the semaphore) becomes **ADL_CTX_STATE_READY** and can be scheduled
- If more than one task is waiting for an Event and when this Event occurs, then all tasks waiting for this Event can become **ADL_CTX_STATE_READY** and can be scheduled.

The defined operations are:

- A **adl_eventCreate** function
- A **adl_eventWait** function
- A **adl_eventClear** function
- A **adl_eventSet** function

3.37.1. Required Header File

The header file for the functions dealing with the ADL Event Service public interface is:

```
adl_event.h
```

3.37.2. Defines

3.37.2.1. The ADL_EVENT_NO_TIMEOUT

No timeout definition

```
#define ADL_EVENT_NO_TIMEOUT0xFFFFFFFF
```

3.37.3. Enumerations

3.37.3.1. The adl_eventWaitMode_e

For adl_eventWait API.

Code

```
typedef enum
{
        ADL_EVENT_WAIT_ANY
        ADL_EVENT_WAIT_ALL
}adl_eventWaitMode_e;
```

Description

ADL_EVENT_WAIT_ANY

Wait for any Event (ANY).

ADL_EVENT_WAIT_ALL

Wait for all Event (ALL).

3.37.4. The adl_eventCreate Function

Enable to create a new event: Allocate the event and initialize internal mask with initial value.

Prototype

```
s32 adl_eventCreate  ( u32  eventFlags );
```

Parameters

eventFlags

Initial value for event mask.

Returned values

- `eventHandle` if creation is successful
- `ADL_RET_ERR_SERVICE_LOCKED` If the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *A reset will be caused for the following exception: Out of memory*

3.37.5. The adl_eventWait Function

Enable to wait for all or only some event depending on mode.

Prototype

```
s32  adl_eventWait ( s32                    eventHandle,
                     u32                    inEventFlags,
                     u32*                   outEventFlags,
                     adl_eventWaitMode_e    eventMode,
                     u32                    eventTimeOut );
```

Parameters

eventHandle

Event handle (returned by adl_eventCreate).

inEventFlags

Event wait mask.

outEventFlags

Affected with event mask when the task is reactivated (outEventFlags can be NULL. In this case, current mask is not returned).

eventMode

Selected wait mode (**`ADL_EVENT_WAIT_ANY`** or **`ADL_EVENT_WAIT_ALL`**).

- If wait mode is **`ADL_EVENT_WAIT_ANY`**
  - If inEventFlags matches with at least one bit of internal mask then the call of **`adl_eventWait`** function stays activated
  - Otherwise, the call of adl_eventWait function is blocking the current task is deactivated (state is ADL_CTX_STATE_WAIT_INNER_EVENT) until inEventFlags matches with at least one bit of internal mask
- Else (if wait mode is **`ADL_EVENT_WAIT_ALL`**)
  - If inEventFlags matches with all bits of internal mask then the call of **`adl_eventWait`** function stays activated
  - Otherwise, the call of **`adl_eventWait`** function is blocking, the current task is deactivated (state is ADL_CTX_STATE_WAIT_INNER_EVENT) until inEventFlags matches with all the bits of internal mask

eventTimeOut

Wait timeout in number tick (18.5 ms)

- If programmed timeout is not ADL_EVENT_NO_TIMEOUT and if the task is waiting for the event, then a timeout timer is launched.
- If the timer has expired, the task is unblocked and **`ADL_RET_ERR_DONE`** code is returned

Returned values

- **`OK`** Operation is successful
- **`ADL_RET_ERR_DONE`** if the Timer has expired and the task is activated
- **`ADL_RET_ERR_PARAM`** If eventMode parameter is neither **`ADL_EVENT_WAIT_ANY`** nor **`ADL_EVENT_WAIT_ALL`** or if inEventFlags is set to 0
- **`ADL_RET_ERR_BAD_STATE`** If the function was called from Task 0 context
- **`ADL_RET_ERR_SERVICE_LOCKED`** If the function was called from a low level interrupt handler (the function is forbidden in this context).

Note: *A reset will be caused for the following exceptions:*

bad Event handle

bad Event wait mode

3.37.6. The adl_eventClear Function

Enable to clear one or several bits in internal event mask.

Prototype

```
s32 adl_eventClear(   s32       eventHandle,
                      u32       inEventFlags,
                      u32*      outEventFlags );
```

Parameters

eventHandle

Event handle (returned by adl_eventCreate).

inEventFlags

Mask indicates which bit to clear into event internal mask.

outEventFlags

Affected with event mask before the operation; (this parameter can be NULL. In this case current mask is not returned)

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** when InEventFlags =0

3.37.7. The adl_eventSet Function

Enable to set one or several bits in internal event mask and to reactivate task waiting for this event.

If event internal mask is modified and if at least one task is waiting for this event then, for each task waiting for the event, according to the wait mode:

- If wait mode is **ADL_EVENT_WAIT_ANY** and if wait mask matches with at least one bit of internal mask, then the task is reactivated.
- Otherwise if wait mode is ADL_EVENT_WAIT_ALL and if wait mask matches with all bits of internal mask then the task is reactivated.

Prototype

```
s32 adl_eventSet( s32    eventHandle,
                  u32    inEventFlags );
```

Parameters

eventHandle

Event handle

inEventFlags

Mask indicates which bit to set into event internal mask.

Returned values

- **OK** on success
- **ADL_RET_ERR_PARAM** when InEventFlags =0

3.37.8. Example

Here is an example of how to use the application Event API access functions.

```
// Global definitions

    // Event object handler
    static u32 l_u32_MyEvent = NULL;

    // External interrupt handler
    static void MyExternal InterruptHandler(void);

    // My task entry point
    void MyTask(void)
    {
        // External interrupt registration
        // ...

        // Event creation
        l_u32_MyEvent = adl_eventCreate(0);

        // Task infinite loop
        while (1)
        {
            // Wait for bit 0 of my Event to be raised (without timeout)
            adl_eventWait(
                l_u32_MyEvent,
                1,
                NULL,
                ADL_EVENT_WAIT_ANY,
                ADL_EVENT_NO_TIMEOUT);

            // Launch my treatments
            // ...

            // Clear Event
            l_s_ErrorCode = adl_eventClear(
                l_u32_MyEvent,
                1,
                NULL);
        }
    }

    void MyExternal InterruptHandler(void)
    {
       // Signal Event
       l_s_ErrorCode = adl_eventSet(l_u32_MyEvent, 1);

       // Interrupt acknowledgement
       // ...
    }
```

3.38. ADL AirPrime Management Services

ADL provides a AirPrime Management Services (AMS).

- AirPrime Management Services Monitoring Service:

This service enables the parameters monitoring with AirPrime Management Services.

The defined operations are:

- A **`adl_idsMonitorSubscribe`** function
- A **`adl_idsMonitorUnsubscribe`** function
- A **`adl_idsMonitorTrace`** function
- A **`adl_idsMonitorDeleteUnused`** function

- AirPrime Management Services Provisioning Service:

This service enables the provisioning of parameters with AirPrime Management Services.

The defined operations are:

- A **`adl_idsProvSubscribe`** function
- A **`adl_idsProvUnsubscribe`** function

3.38.1. Required Header File

The header file for the ADL AirPrime Management Services Service public interface function is:

```
adl_ids.h
```

3.38.2. Data Structure for Monitoring Process

3.38.2.1. The adl_idsMonitorCfg_t Structure

Structure for New Monitoring Configuration on reception of server message.

Code

```
typedef struct
{
      bool                         OnDemand,
      bool                         Cumul,
      u32                          Timing,
      adl_idsMonitorDataType_e     DataType,
      void                         *TriggerValueData,
      u32                          TriggerValueLen,
      adl_idsMonitorTrig_e         TrigMode,
      s32                          TriggerHysteresis,
      adl_idsMonitorFlagReset      Reset
} adl_idsMonitorCfg_t;
```

Description

State:

OnDemand flag: the server can request an alert/report at any time then the device looks for all monitoring parameters that are marked as being "On Demand" and generates a report containing all those parameters and sends it back to the server.

Cumul:

Cumulate parameter definition only available if DataType is **`ADL_IDS_MONITOR_INTEG_DATA`**. If set to TRUE when **`adl_idsMonitorTrace`** is called the value given will be added to the previous one when reported to the server.

Timing:

Timer for monitoring: 0 is no timing, otherwise timing in minutes when timer elapsed the parameter set through the **`adl_idsMonitorTrace`** is reported to the server (this is an internal monitoring process).

DataType:

Paramater Type.

TriggerValueData:

Trigger value only valid when TrigMode is not **`ADL_IDS_MONITOR_NO_TRIG.`**

TriggerValueLen:

Trigger length when TrigMode is not **`ADL_IDS_MONITOR_NO_TRIG`**.

TrigMode:

Trigger mode only valid when DataType is **`ADL_IDS_MONITOR_INTEG_DATA`**.

TriggerHysteresis:

Behaviour depends on TrigMode

Reset:

When should the monitoring parameter value be reset?

3.38.2.2. The adl_idsMonitorDataType_e Type

This enumeration for Monitoring parameter type.

Code

```
enum
{
      ADL_IDS_MONITOR_INTEG_DATA,
      ADL_IDS_MONITOR_BUFF_DATA
 } adl_idsMonitorDataType_e;
```

Description

ADL_IDS_MONITOR_INTEG_DATA

Data type is an integer.

ADL_IDS_MONITOR_BUFF_DATA

Data type is buffer.

3.38.2.3. The adl_idsMonitorFlagReset_e Type

This enumeration for Monitoring state.

Code

```
enum
{
      ADL_IDS_MONITOR_RESET_NOW,
      ADL_IDS_MONITOR_RESET_ON_TRIGGER,
      ADL_IDS_MONITOR_RESET_ON_TIMER,
      ADL_IDS_MONITOR_RESET_ON_DEMAND
 } adl_idsMonitorFlagReset_e;
```

Description

ADL_IDS_MONITOR_RESET_NOW

Reset monitoring parameter value now (on subscription time).

ADL_IDS_MONITOR_RESET_ON_TRIGGER

Reset monitoring parameter value when Trigger is happening.

ADL_IDS_MONITOR_RESET_ON_TIME

Reset monitoring parameter value when timer ends.

ADL_IDS_MONITOR_RESET_ON_DEMAND

Reset monitoring parameter value when monitoring starts.

3.38.2.4. The adl_idsMonitorTrig_e Type

This enumeration for Monitoring Trigger mode.

Code

```
enum
{
      ADL_IDS_MONITOR_NO_TRIGGER,
      ADL_IDS_MONITOR_TRIGGER_UP,
      ADL_IDS_MONITOR_TRIGGER_DOWN,
      ADL_IDS_MONITOR_TRIGGER_BOTH,
      ADL_IDS_MONITOR_TRIGGER_EQUAL,
      ADL_IDS_MONITOR_TRIGGER_NOT_EQUAL,
      ADL_IDS_MONITOR_TRIGGER_DELTA
 } adl_idsMonitorTrig_e;
```

Description

ADL_IDS_MONITOR_NO_TRIGGER

No Trigger.

ADL_IDS_MONITOR_TRIGGER_UP

Trigger when value is higher than TriggerValueData

ADL_IDS_MONITOR_TRIGGER_DOWN

Trigger when value is lower than TriggerValueData

ADL_IDS_MONITOR_TRIGGER_BOTH

Trigger when the TriggerValueData is reached

ADL_IDS_MONITOR_TRIGGER_EQUAL

Trigger when the value is equal to the TriggerValueData

ADL_IDS_MONITOR_TRIGGER_NOT_EQUAL

Trigger when the value is not equal to the TriggerValueData

ADL_IDS_MONITOR_TRIGGER_DELTA

Trigger when the value is higher than TriggerValueData + Hysteresis or less than TriggerValue – Hysteresis

3.38.3. Data structure for Provisioning Process

3.38.3.1. The adl_idsProvCfg_t Structure

Structure for provisioning Configuration.

```
typedef struct
{
      void*                           Context,
      adl_idsProvCallBackRead         idsProvRead,
      adl_idsProvCallBackWrite        idsProvWrite,
      adl_idsProvCallBackGetLength    idsProvGetLength
} adl_idsProvCfg_t;
```

Fields

Context:

Buffer to specify a context available during the whole process.

idsProvRead:

Read function pointer.

idsProvWrite:

Write function pointer.

idsProvGetLength:

Get Length function pointer.

3.38.3.2. The adl_idsProvCallBackRead

When the server requests a READ, this function is called to read the parameter associated with the handle in the provided Buffer with the length returned by **adl_idsProvCallBackGetLength** function.

Prototype

```
typedef s32(*) adl_idsProvCallBackRead ( u32    sHandle,
                                          void   *Ctx,
                                          void   *Ptr,
                                          u32    Len );
```

Parameters

sHandle

Handle

Ctx

Context that will be given back once the callback is called

Ptr

Buffer read

Len

Buffer Length to be read

Return values

- `OK` on success
- `Error` otherwise

3.38.3.3. The adl_idsProvCallBackWrite

When the server requests a WRITE, this function is called to write the provided Buffer with the provided length in the parameter associated with the handle.

Prototype

```
typedef s32(*) adl_idsProvCallBackWrite ( u32    sHandle,
                                          void   *Ctx,
                                          void   *Ptr,
                                          u32    Len );
```

Parameters

sHandle

Handle

Ctx

Context that will be given back once the callback is called

Ptr

Buffer write

Len

Buffer Length to be written

Return values

- `OK` on success
- `Error` otherwise

3.38.3.4. The adl_idsProvCallBackGetLength

When the server requests a READ this function is called to get the length of the parameter associated to the handle to be read to allocate the desired memory.

Prototype

```
typedef s32(*) adl_idsProvCallBackGetLength  (  u32       sHandle,
                                                 void      *Ctx   );
```

Parameters

sHandle

Handle

Ctx

Context that will be given back once the callback is called

Return values

- **`Length`** if positive
- **`Error`** otherwise

3.38.4. AirPrime Management Services Monitoring API Access Functions

3.38.4.1. The adl_idsMonitorSubscribe Function

The aim of this function is to activate Monitoring on given name with associated configuration.

In the provided configuration, the user has to specify a callback function (**`idsMonitorNewConfig`**) to handle any message from the server suggesting to use a new configuration.

Prototype

```
s32  adl_idsMonitorSubscribe ( ascii*                Name,
                               adl_idsMonitorCfg_t*   Config );
```

Parameters

Name

Parameter Name (up to 50 characters)

Config

Parameter Configuration

Returned values

- **`Handle`** If positive value (AirPrime Management Services handle to be used on further AirPrime Management Services API functions calls)
- **`ADL_RET_ERR_PARAM`** If one parameter is NULL
- **`ADL_RET_ERR_NOT_SUBSCRIBED`** If AirPrime Management Services service is not started
- **`ADL_RET_ERR_BAD_STATE`** If AirPrime Management Services service is busy (a session with server is already opened and an Open AT® parameter is accessed)
- **`ADL_RET_ERR_NO_MORE_HANDLES`** If no more parameters can be monitored
- **`ADL_RET_ERR_ALREADY_SUBSCRIBED`** If a parameter with such name is already monitored
- **`ADL_RET_ERR_NOT_SUPPORTED`** If the device is not allowing this feature

Note: *Up to 50 Open AT® parameters can be monitored at the same time.*

3.38.4.2. The adl_idsMonitorUnsubscribe Function

The aim of this function is to remove a parameter under Monitoring by providing its Handle (given at Activation).

Prototype

```
s32  adl_idsMonitorUnsubscribe ( s32     sHandle );
```

Parameters

sHandle

Handle associated with the parameter (returned by **adl_idsMonitorSubscribe** API)

Returned values

- **OK** on success
- **ADL_RET_ERR_UNKNOWN_HANDLE** If the handle provided is unknown
- **ADL_RET_ERR_NOT_SUBSCRIBED** If AirPrime Management Services service has not started
- **ADL_RET_ERR_BAD_STATE** If AirPrime Management Services service is busy (a session with server is already open and an Open AT® parameter is accessed)
- **ADL_RET_ERR_NOT_SUPPORTED** If the device is not allowing this feature

3.38.4.3. The adl_idsMonitorTrace Function

The aim of this function is to Trace a parameter under Monitoring by providing its Handle (given at Activation) and data with length of updated value.

Prototype

```
s32  adl_idsMonitorTrace (  s32    sHandle,
                            void*  Data,
                            u32    Len );
```

Parameters

sHandle

Handle associated with the parameter (returned by **adl_idsMonitorSubscribe** API)

Data

Pointer on Data

Len

Data Length

Returned values

- **OK** on success
- **ADL_RET_ERR_UNKNOWN_HANDLE** If the handle provided is unknown
- **ADL_RET_ERR_NOT_SUBSCRIBED** If AirPrime Management Services service has not started
- **ADL_RET_ERR_BAD_STATE** If AirPrime Management Services service is busy (a session with server is already open and an Open AT® parameter is accessed)
- **ADL_RET_ERR_NOT_SUPPORTED** If the device is not allowing this feature

Note: *If when subscribing with **adl_idsMonitorSubscribe** the cumul parameter was set to TRUE, the value of the data here traced will be added to the previous one when reported to the server.*

3.38.4.4. The adl_idsMonitorDeleteUnused Function

The aim of this function is to delete unused parameter under Monitoring. Unused parameter are the ones that have been subscribed but are not anymore. Flash object entries containing the configuration for these parameters have been allocated. So calling this API cleans flash entries of unused parameters..

Prototype

```
s32  adl_idsMonitorDeleteUnused ( void );
```

Returned values

- **OK** on success
- **ADL_RET_ERR_NOT_SUBSCRIBED** If AirPrime Management Services service is not started
- **ADL_RET_ERR_BAD_STATE** If AirPrime Management Services service is busy (a session with server is already opened and an Open AT® parameter is accessed)
- **ADL_RET_ERR_NOT_SUPPORTED** If the device is not allowing this feature

3.38.5. AirPrime Management Services Provisioning API Access Functions

3.38.5.1. The adl_idsProvSubscribe Function

The aim of this function is to activate Provisioning on given Name with associated configuration.

In the provided configuration, the user has to specify:

- a callback function **(idsProvRead**) to handle any READ message coming from the server
- a callback function (**idsProvWrite**) to handle any WRITE message coming from the server
- a callback function (**idsProvGetLength**) to get the length of the parameter in case of a READ message coming from the server

Prototype

```
s32  adl_idsProvSubscribe   (  ascii*                 Name,
                               adl_idsProvCfg_t*      Config );
```

Parameters

Name

Parameter Name (up to 50 characters)

Config

Parameter Configuration

Returned values

- **Handle** If positive value (AirPrime Management Services handle to be used on further AirPrime Management Services API functions calls)
- **ADL_RET_ERR_PARAM** If one parameter is NULL
- **ADL_RET_ERR_NOT_SUBSCRIBED** If AirPrime Management Services service has not started
- **ADL_RET_ERR_BAD_STATE** If AirPrime Management Services service is busy (a session with server is already open and an Open AT® parameter is accessed)
- **ADL_RET_ERR_NO_MORE_HANDLES** If no more parameters can be provided
- **ADL_RET_ERR_ALREADY_SUBSCRIBED** If a parameter with such name is already provided
- **ADL_RET_ERR_NOT_SUPPORTED** If the device is not allowing this feature

Note: Up to 50 Open AT® parameters can be provided at the same time.

3.38.5.2. The adl_idsProvUnsubscribe Function

The aim of this function is to remove a parameter for provisioning by providing its Handle (given at Activation).

Prototype

```
s32  adl_idsProvUnsubscribe ( s32     sHandle );
```

Parameters

sHandle

Handle associated with the parameter (returned by **adl_idsProvSubscribe** API)

Returned values

- **OK** on success
- **ADL_RET_ERR_UNKNOWN_HANDLE** If the handle is unknown
- **ADL_RET_ERR_NOT_SUBSCRIBED** If AirPrime Management Services service has not started
- **ADL_RET_ERR_BAD_STATE** If AirPrime Management Services service is busy (a session with server is already open and an Open AT® parameter is accessed)
- **ADL_RET_ERR_NOT_SUPPORTED** If the device is not allowing this feature

3.38.6. Example

This example demonstrates how to use the AirPrime Management Services in a nominal case (error cases not handled) with a embedded module.

Complete examples using the AT Command service are also available on the SDK.

```
 s32 MonitorHandle;
s32 ProvHandle;

static s32 MonTemp = 5;
static s32 ProvTemp = 10;
 char Number[32];
u32 value = 15;


void TemperatureHasChanged (s32 NewTemperature )
{
    s32 sRet;

    MonTemp = NewTemperature;

    // The temperature has changed notify the serveur
    adl_idsMonitorTrace(MonitorHandle, &MonTemp, sizeof(MonTemp));

    TRACE (( 1, "TemperatureHasChanged : temperature %d",
MonTemp ));
}
```

```
void InitMonitor()
{
    adl_idsMonitorCfg_t MyMonitorConfig;

 MyMonitorConfig.OnDemand = FALSE;
    MyMonitorConfig.Cumul = TRUE;
    MyMonitorConfig.Timing = 0;
    MyMonitorConfig.DataType = ADL_IDS_MONITOR_INTEG_DATA;
    MyMonitorConfig.TriggerValueData = (void*)&value;
    MyMonitorConfig.TriggerValueLen = sizeof(value);
    MyMonitorConfig.TrigMode = ADL_IDS_MONITOR_TRIGGER_UP;
 MyMonitorConfig.TriggerHysteresis = 0;
    MyMonitorConfig.Reset = ADL_IDS_MONITOR_RESET_NOW ;


    // now subscribe with the set configuration
    MonitorHandle = adl_idsMonitorSubscribe("Temperature",
                &MyMonitorConfig);

  // get rid of all unused Monitor parameter
    adl_idsMonitorDeleteUnused();

    // Set the parameter value
    sRet = adl_idsMonitorTrace(MonitorHandle, &value,
                                  sizeof(value));

    TRACE (( 1, "InitMonitor : MonitorHandle %d", MonitorHandle ));
}


s32 MyProvRead (s32 Handle, void *Ctx, void * Ptr, u32 Len )
{
  TRACE (( 1, "MyProvRead is called" ));
    // Read temperature from device measuring
    wm_itoa(MonTemp, Number);
    wm_memcpy(Ptr, Number, wm_strlen(Number));
    return OK;
}

s32 MyProvWrite (s32 Handle, void *Ctx, void * Ptr, u32 Len )
{
    TRACE (( 1, "MyProvWrite is called" ));
    // Write temperature to device controller
    wm_memcpy(Number, Ptr, Len);
    Number[Len] = 0;
    // Write temperature to device controller
 ProvTemp = (s32) wm_atoi(Ptr);
       return OK;

}
```

```
s32 MyProvGetLength (void *Ctx )
{
    TRACE (( 1, "MyProvGetLength is called" ));
    wm_itoa(MonTemp, Number);
    return wm_strlen(Number);
}


void InitProvision()
{
    adl_idsProvCfg_t MyProvConfig;

    MyProvConfig.Context = 0;
    MyProvConfig.idsProvRead = (adl_idsProvCallBackRead) MyProvRead;
    MyProvConfig.idsProvWrite = (adl_idsProvCallBackWrite)
           MyProvWrite;
    MyProvConfig.idsProvGetLength = (adl_idsProvCallBackGetLength)
           MyProvGetLength;

    // now subscribe with the set configuration
    ProvHandle = adl_idsProvSubscribe("Temperature", &MyProvConfig);

    TRACE (( 1, "InitProvision : ProvHandle %d", ProvHandle ));
}
```

3.39. ADL Open Device Service

The ADL Open Device service provides a raw access to any device behaving as a serial port. Each device is defined in a class, refer to **`eDfClid_t`** structure in wm_factory.h file to get more information about the existing classes.

In order to get a raw access to the device, a software block component supplies APIs which allows to manipulate the device. it is called **Service Provide** (SP) and APIs of this SP are based on a **Generic Interface**. For each existing device class, there is only one generic interface. For example, a SP which allows to access to an UART, the SP is based on UART generic interface, refer to wm_uart.h to get more information. These SP could be either existing SP on the Firmware or SP which are defined by Open AT® application. Each SP supplies the following functions:

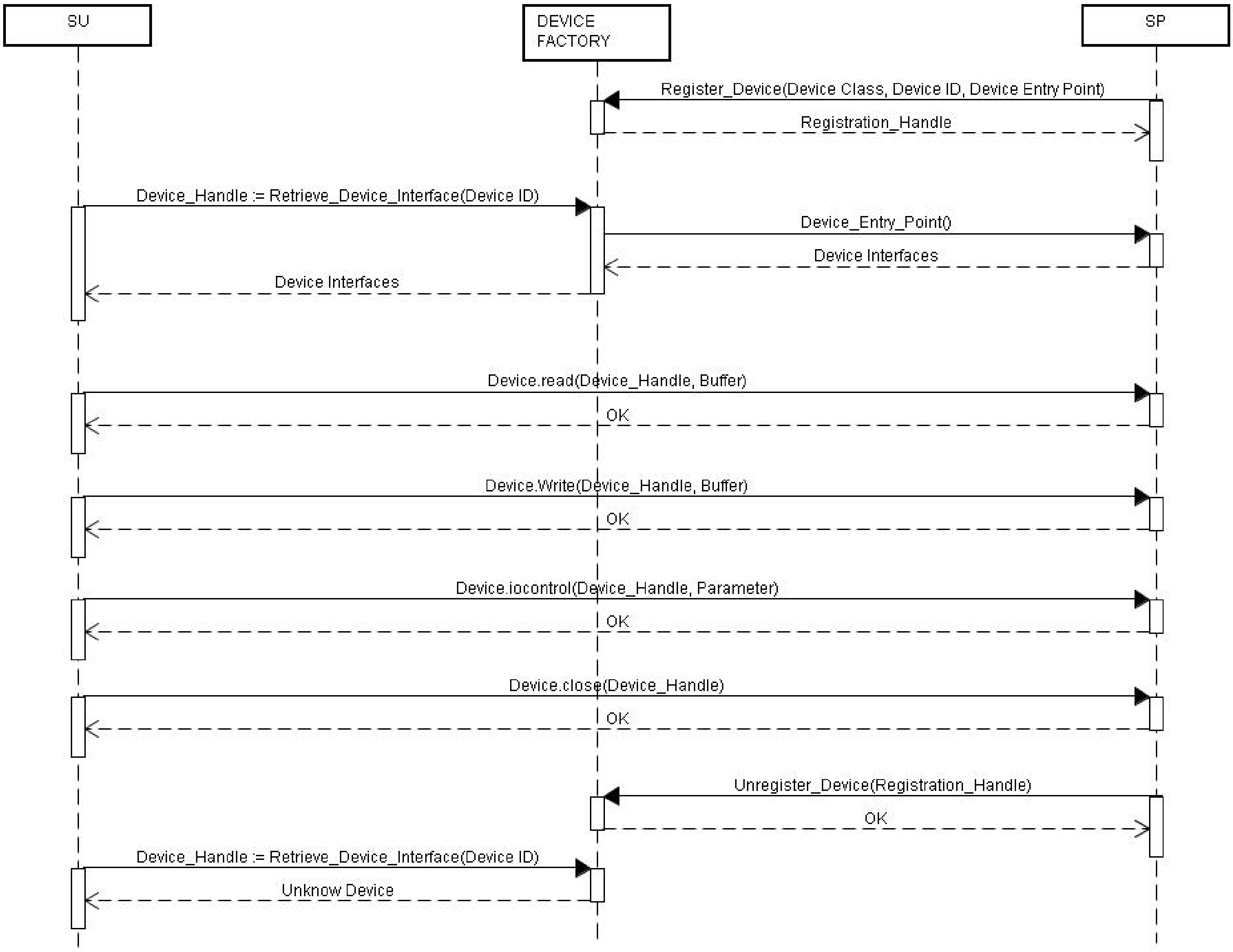
- read function: to read data from a device buffer
- write function: to write data to a device buffer
- I/O control function: to set/get device parameters
- close function: to release the device

Services supply by a SP can be accessed either by Firmware or by Open AT® application. In this case, Firmware and Open AT® application are called **Service User** (SU). A SU can:

- get information about the existing SPs
- retrieves the SP's interfaces at runtime to access to the raw device configuration (read, write, I/O control, close functions)

To be accessible, a SP of device has to be previously registred in the Firmware. Then its services can be accessed by the SU.

Typical use diagram



The following diagram illustrates a typical mechanism between SU and SP.

The ADL Open Device service allows to:

- register to a new SP defined by Open AT® application
- unregister to the SPs which are defined by an Open AT® application
- get a raw access to a device behaving via SPs defined by Firmware or Open AT® application

Note: The ADL Open Device service is not available in RTE mode.

The defined operations are:

- An **`adl_odRegister`** function to register a new SP
- An **`adl_odUnregister`** function to unregister a SP previously registered
- An **`adl_odOpen`** function to access to SP of a device

3.39.1. Required Header File

The header file for the ADL Open Device Service public interface function is:

`adl_OpenDevice.h`

3.39.2. The adl_odOpen_f function

Such entry point function has to be supplied to ADL through the **`adl_odRegister`** interface to access to a device SP defined by customer application through **`adl_odOpen`** function.

This function has to allow to supply to SU the interface of SP (read , write, IO control, close functions).

Refer to **"Device registration"** part to get an use case example.

Prototype

```
typedef s32(*)  adl_odOpen_f ( void   *param );
```

Parameters

param

pointer on parameter structure according to device class which is registered

Returned values

- **`handle`** if application has succeeded to open the device
- **`ERROR`** Otherwise

3.39.3. The adl_odOpen function

This function allows to access to one of available device. According to the device class id, this function can:

- initialise and configurate the port
- provide event callbacks to the device
- retrieve functions interface from the device

Prototype

```
s32 adl_odOpen (  eDfClid_t     dev_clss_id,
                  void*         param );
```

Parameters

dev_clss_id

device class identifier (refer to **`wm_factory.h`** to get more information)

param

pointer on device settings

Note: *For instance, only UART device can be opened.*
To open an UART like device (as detailed in the code sample):

- **`dev_clss_id`** has to be DF_UART_CLID
- param has to be a pointer on a **`sUartSettings_t`** structure (refer to wm_uart.h). User will provide the UART Id, UART role and events callbacks if needed. Device will provide back functions interface as read, write or io_control. Refer to Open UART Interface description for more information.

Returned values

- **`Handle`** Completed operation
- **`ERROR`** Failed operation
- **`ADL_RET_ERR_PARAM`** no param provided

3.39.4. Example

The code sample below illustrates a nominal use case of the ADL Open Device Service public interface.

```
 //MyFunction allows to open an UART.
//Opening parameters are based on UART Interface Pattern

#include "adl_OpenDevice.h"
#include "wm_uart.h"

static psGItfCont_t uart_if;
static u32 uart2_hdl;

...

void MyFunction( void )
{
   sUartSettings_t settings;
   sUartLc_t       line_coding;

  // Set the line coding parameters
    line_coding.valid_fields = UART_LC_ALL;
    line_coding.rate = (eUartRate_t)( UART_RATE_USER_DEF | 57600 );
    line_coding.stop = UART_STOP_BIT_1;
    line_coding.data = UART_DATALENGTH_8;
    line_coding.parity = UART_PARITY_NONE;

  // UART2 will be opened in NULL MODEM role / with synchronous read/write
    settings.identity = "UART2";
    settings.role = UART_ROLE_NM;
    settings.capabilities = NULL;
    settings.event_handlers = NULL;
    settings.interface = &uart_if;
    settings.line_coding = &line_coding;

  uart_hdl = adl_odOpen( DF_UART_CLID, &settings );
    if( !uart_hdl )
  {
    // UART2 opening failed...
      return;
    }

  // UART2 successfully opened, write some bytes
  uart_if.write( uart_hdl, "Tx Some bytes", 13 );

}
```

3.39.5. The adl_odRegister function

This function allows to register a new device which could be used by application or firmware. This device has to be based on one of available device class (UART, SIM, ...)

Prototype

```
s32 adl_odRegister ( eDfClid_t       dev_class,
                     ascii*          pub_id,
                     u32             priv_id,
                     adl_odOpen_f    OpenDevice );
```

Parameters

dev_class

device class identifier (refer to `wm_factory.h` to get more information)

pub_id

string which defines the public ID of device to be registered

priv_id

private ID, it is linked to the public ID in the device table register

OpenDevice

entry point function to open the registered device

Returned values

- `Handle` if registration succeeded, to be used with unregistration API
- `ERROR` if registration failed
- `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interruption handler

3.39.6. The adl_odUnregister function

This function allows to unregister a device which has been previously subscribed with `adl_odRegister` API.

Prototype

```
s32 adl_odUnregister ( s32  odHandle );
```

Parameters

odHandle

Handle of the device which has to be unregistered

Returned values

- `OK` on success
- `ADL_RET_ERR_UNKNOWN_HDL` if the provided handle is unknown
- `ADL_RET_ERR_SERVICE_LOCKED` if called from a low level interruption handler

3.39.7. Example

The code sample below illustrates a nominal use case of the ADL Open Device Service public interface

```
//Register/ unregister a new service provider for UART device

  // Device SP handle
  s32 MySpHandle    // UART Service Provider handle

  // Uart interface prototype
  static eChStatus_t MyUartReadFunction( u32 Handle, void* pData, u32 amount );
  static eChStatus_t MyUartWriteFunction( u32 Handle, void* pData, u32 length
);
  static eChStatus_t MyUartIOControlFunction( u32 Handle, u32 Cmd, void* pParam
);
  static eChStatus_t MyUartCloseFunction( u32 Handle );

  // code hereafter is for SP block
  static eChStatus_t MyUartReadFunction( u32 Handle, void* pData, u32 amount )
  {
     ...
  }

  static eChStatus_t MyUartWriteFunction( u32 Handle, void* pData, u32 length )
  {
     ...
  }

  static eChStatus_t MyUartIOControlFunction( u32 Handle, u32 Cmd, void* pParam
)
  {
     ...
  }
  static eChStatus_t MyUartCloseFunction( u32 Handle )
  {
     ...
  }

  static void MyOpenDeviceFunction( psUartSettings_t  UartSettings )
  {
      s32 Handle = 0x10;

      if( !UartSettings )
            return 0;
      ...

      // Supply interface
      if( UartSettings->interface )
      {
          // Device interface structure

          static const sGItfCont_t UartItf =
          {
              MyUartReadFunction,
              MyUartWriteFunction,
              MyUartIOControlFunction,
              MyUartCloseFunction,
          };

          *UartSettings->interface = (psGItfCont_t)&UartItf;
      }

      ...
```

```
        // return an handle
        return Handle;
    }


    s32 MyRegisterFunction (void)
    {
        s32 sReturn;

        ...

        // Register a new UART device
        MySpHandle = adl_odRegister( DF_UART_CLID, "MY_UART", MyPrivateID,
(adl_odOpen_f)MyOpenDeviceFunction );

        ...
    }

    void MyUnregisterFunction (void)
    {
        s32 sReturn;

        ...

        // Unregister device
        sReturn = adl_odUnregister( MySpHandle );

        ...
    }
```

3.40. ADL OS Clock Interface Specification

ADL provides an API to get the time from the embedded module initialization.

The defined operation is:

- A adl_GetOsClockTime function

3.40.1. Required Header File

The header file for the function dealing with the ADL OS clock is:

```
adl_osclk.h
```

3.40.2. The adl_osclkGetTime Function

This function returns the elapsed time from the embedded module initialization. The time granularity is 4,615 ms.

Prototype

```
s32  adl_osclkGetTime (  u64*      pTime );
```

Parameters

pTime

Time (unit : ms)

Returned values

- **`OK`** on success
- **`ADL_RET_ERR_NOT_SUPPORTED`** if the API is not supported by the Sierra Wireless stack. In this case, the pTime pointer value is set to 0.

4. Device Services

The following subsections describe in detail the Device Services available for use.

4.1. Open UART Interface

ADL provides Open UART Interface to give **direct** access to the embedded module UART Service Providers. A UART Service Provider should be assigned as a software component managing either a physical or an emulated UART. Whatever category it belongs to, UART Service Provider **is required** to implement the Open UART Interface. As the main consequence UART Service Users do not have to be concerned with the real UART implementation they will deal with.

By default (i.e. without any Open AT® application, or if the application does not use the Open UART Interface) all embedded module UART service providers are managed by and are available for the Sierra Wireless firmware.

A UART service provider handled by the Sierra Wireless firmware is not available for an Open AT® application.

A UART service provider handled by an Open AT® application is not available for the Sierra Wireless firmware.

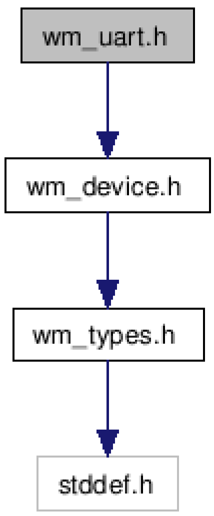
In both previous cases, an attempt to get access to an already used UART service provider returns an error to the requestor.

4.1.1. Required Header File

The Open UART Interface is defined by the following header file:

wm_uart.h

Note that the wm_uart.h header is self-sufficient (or auto-compilable), which means there is no need (for the service user) to include any other header files to get access to a UART service provider (except for the opening stage, where the adl_OpenDevice.h file is required). The way the wm_uart.h is built is illustrated by the following include figure.



4.1.2. Data Structures

4.1.2.1. The sUartSettings_t structure

The UART configuration structure, when the ADL open device service (adl_OpenDevice) function is used.

```
typedef struct
{
        // GENERIC Fields, is a sGenSettings_t type (wm_device.h)
        char* identity;
         psUartEvent_t    event_handlers;
         ppsGItfCont_t    interface;
        // END of the GENERIC fields

        // SPECIFIC IN parameters
        enum eUartRole
         {
            UART_ROLE_NM     = 0x10,
            UART_ROLE_DTE    = 0x14,
            UART_ROLE_DCE    = 0x1B,
            UART_ROLE_CAP    = 0xFF,
            UART_ROLE_MAP    = 0x70000000L
         } role;
            psUartLc_t    line_coding;

        // SPECIFIC output
         psUartCap_t   capabilities;
} sUartSettings_t, *psUartSettings_t;
```

Fields

identity:

This field is mandatory. It allows the service user to choose the UART service provider it wants to work with. Setting this field to NULL generates an error. Any non NULL value is considered by the UART service provider as a pointer of a NULL terminated string. Supported string contents are listed below:

- "UART1": to get access to the embedded module UART1
- "UART2": to get access to the embedded module UART2
- "UART3": to get access to the embedded module USB serial port

Attempting to provide an identity other than those listed above generates an error.

Note: This list is not exhaustive and will be updated when new UART service providers are available.

event_handlers:

This field is optional. It allows the service user to provide its event handlers during the opening stage. Please note the G_IOC_EVENT_HANDLERS IO control can also be used to manage the event handler configuration.

Setting this field to NULL means there is no event handling information available for the UART service provider. Setting this field to a non NULL value is considered by the UART service provider as a pointer of sUartEvent_t structure.

Caution: *When this field is not provided then the UART service provider handles read and write operations in a synchronous (blocking) mode. See also the operation mode clause for further information about the synchronous and asynchronous operating modes.*

See also sUartEvent_t description for further information about event handler configuration management.

interface:

This field is optional. It allows the service user to dynamically retrieve UART service provider interface inside a generic interface container.

Setting this field to NULL means the UART service provider shall not provide its interface to its service user. Any non NULL value is considered by the UART service provider as the address of a pointer of sGItfCont_t structure.

Caution: *Setting this field to* ***NULL is hazardous****, and should not be done except when the service user has already retrieved a UART service provider interface of the same type (e.g. the embedded module UART1 and UART2 share the same type).*

See also sGItfCont_t description for further information about generic interface containers and interface retrieving at run time.

role:

This field is mandatory. It allows the service user to indicate to the UART service provider which running mode shall be established. Running modes list is given hereafter:

- UART_ROLE_NM: To handle NULL MODEM connections with a maximum of 4 signals wired: RX, TX, RTS and CTS. Supported by the embedded module UART1 and UART2.
- UART_ROLE_DTE: To behave as a DTE (set/get DTR; get DSR, RI & DCD). Currently not supported.
- UART_ROLE_DCE: To behave as a DCE (set/get DSR, RI & DCD; get DTR) by using a maximum 8 signals: RX, TX, RTS, CTS, DTR, DSR, RI and DCD. Supported by the embedded module UART1 and UART3.
- UART_ROLE_CAP: ***deprecated***, attempting to use this identifier generates an error.
- Any other value is not supported by the UART service provider and shall generate an error.

line_codings:

This field is optional. It allows the service user to set the UART speed and the character format (amount of data bits, amount of stop bits and parity type) configuration.

Setting this field to NULL means the UART service provider shall apply its default configuration (115200, 8N1). Any non NULL value is considered by the UART service provider as a pointer of sUartLc_t structure.

See also sUartLc_t description for further information about the line coding setting.

capabilities:

This field is optional. It allows the UART service provider to return its capabilities to the service user during the opening stage. Please note the generic G_IOC_CAPABILITIES IO control can also be used to retrieve the capabilities after the opening stage.

Setting this field to NULL means the UART service provider shall not return its capabilities to the service user. Any non NULL value is considered by the UART service provider as a pointer of s sUartCap_t structure.

See also sUartCap_t description for further information about the UART capabilities content and the G_IOC_CAPABILITIES for further information about the capabilities gathering after the opening stage.

4.1.2.2. The sUartCap_t structure

The UART capabilities structure, when the ADL open device service (adl_OpenDevice) function is used (nested in the sUartSettings_t structure) or when the generic IO control G_IOC_CAPABILITIES is used.

```
typedef struct
{
        u32 speed; /* see the eUartRate_t enum */
        enum ioc_cap   /* IO command capability */
        {
          IOC_UART_LC_SUP = IOC_LAST_SUP, /* see wm_device.h */
          IOC_UART_SS_SUP = IOC_LAST_SUP << 1,
          IOC_UART_FL_SUP = IOC_LAST_SUP << 2,
          IOC_UART_FC_SUP = IOC_LAST_SUP << 3,
          IOC_UART_TE_SUP = IOC_LAST_SUP << 4,
          IOC_UART_SM_SUP = IOC_LAST_SUP << 5,
          IOC_UART_ER_SUP = IOC_LAST_SUP << 6,
          IOC_UART_TO_SUP = IOC_LAST_SUP << 7,
          IOC_UART_FD_SUP = IOC_LAST_SUP << 8,
        } ioc;
        enum stop_cap
        {
          UART_STOP_BIT_1_CAP     =   0x01,
          UART_STOP_BIT_2_CAP     =   0x02,
          UART_STOP_BIT_1_5_CAP   =   0x04
        } stop;
        enum par_cap
        {
          UART_PAR_NONE_CAP   = 0x01,
        UART_PAR_EVEN_CAP     = 0x02,
        UART_PAR_ODD_CAP      = 0x04,
        UART_PAR_SPACE_CAP    = 0x08,
        UART_PAR_MARK_CAP     = 0x10
        } parity;
        enum data_cap
        {
          UART_DATA_AUTO_CAP  = 0x01,
          UART_DATA_5BITS_CAP = 0x02,
          UART_DATA_6BITS_CAP = 0x04,
          UART_DATA_7BITS_CAP = 0x08,
          UART_DATA_8BITS_CAP = 0x10,
          UART_DATA_8BITS_CAP = 0x20,
        } parity;
```

```
        enum fc_cap /* flow control capability */
        {
         UART_FC_NONE_CAP    = 0x01,
         UART_FC_RTS_CTS_CAP = 0x02,
         UART_FC_XONXOFF_CAP = 0x04,
         UART_FC_ALL_CAP     = (UART_FC_NONE_CAP |
                                UART_FC_RTS_CTS_CAP |
                                UART_FC_XONXOFF_CAP)
        } fc;
         u8 fd_cap[6];        /* FIFO depth capability */
         u16 min_dur_tx;      /* in tenth of second */
         u16 max_dur_rx;      /* in tenth of second */
} sUartCap_t, *psUartCap_t;
```

Fields

speed:

This field describes the speeds supported by the UART service provider. The field structure is similar to the eUartStop_t enumerator field structure. A bit set to 1 means the service user is allowed using the corresponding bit of the eUartRate_t enumerator. A bit set to 0 means the service user is not allowed using the corresponding bit in the eUartRate_t enumerator.

See also eUartRate_t description for further information about the speed configuration.

ioc:

This field describes the IO commands supported by the UART service provider. Each bits in the [Bit0...Bit9] range is related to an IO command identity. Bit X set to 1 means UART service provider implements the related X IO command, bit Y set to 0 means UART service provider does not implement related Y IO command.

See also eUartIoCmd_t description for further information about the speed configuration.

stop:

This field describes the stop bit configurations supported by the UART service provider. The field structure is similar to the eUartStop enumerator field structure. A bit set to 1 means the service user is allowed using the corresponding bit of the eUartStop enumerator. A bit set to 0 means the service user is not allowed using the corresponding bit in the eUartStop enumerator.

See also eUartStop enumerator description (sUartLc_t structure) for further information about the stop bits configuration.

parity:

This field describes the parity bit configurations supported by the UART service provider. The field structure is similar to the eUartParity enumerator field structure. A bit set to 1 means the service user is allowed to use the corresponding bit of the eUartParity enumerator. A bit set to 0 means the service user is not allowed to use the corresponding bit in the eUartParity enumerator.

See also eUartParity enumerator (sUartLc_t structure) description for further information about the stop bits configuration.

data:

This field describes the data bits configurations supported by the UART service provider. The field structure is similar to the eUartData enumerator field structure. A bit set to 1 means the service user is allowed to use the corresponding bit of the eUartData enumerator. A bit set to 0 means the service user is not allowed to use the corresponding bit in the eUartData enumerator.

See also eUartData enumerator (sUartLc_t structure) description for further information about the data bits configuration.

fc:

This field describes the flow control configurations supported by the UART service provider. The field structure is similar to the eFcType enumerator field structure. A bit set to 1 means the service user is allowed to use the corresponding bit of the eFcType enumerator. A bit set to 0 means the service user is not allowed to use the corresponding bit in the eUartData enumerator.

See also eFcType enumerator (sUartFlowCtrl_t structure) description for further information about the flow control configuration.

fd_cap:

This field describes the RX FIFO threshold configurations supported by the UART service provider. Any non NULL entry in this array can be later on used for the RX FIFO threshold configuration. In case all entries in the array are set to a NULL value it means the UART service provider does not support the IOC_UART_FD operation.

See also sUartFd_t structure description for further information about the UART RX FIFO depth management.

min_dur_tx, max_dur_rx:

Always set to 0 **(deprecated).**

4.1.2.3. The sUartLc_t structure

The UART Line Coding structure, when the ADL open device service, adl_OpenDevice function, is used (nested in the sUartSettings_t structure) or when the IOC_UART_LC IO control is used.

```
typedef struct
{
        eGIocSo_t op;  /* generic get/set operation */
        enum eUartLcField
        {/* indicates which following fields are significant */
          UART_LC_RATE  = 1,
          UART_LC_STOP  = 2,
          UART_LC_PAR   = 4,
          UART_LC_DATA  = 8,
          UART_LC_ALL   = (UART_LC_RATE |UART_LC_STOP
                            |UART_LC_PAR | UART_LC_DATA),
          UART_LC_MAP = 0x70000000L
        } valid_fields;
          eUartRate_t rate;
          enum eUartStop
        {
          UART_STOP_BIT_1,
          UART_STOP_BIT_2,
          UART_STOP_BIT_1_5,
          UART_STOP_BIT_LAST,
          UART_STOP_BIT_MAP    = 0x7F,
        } stop;
```

```
        enum eUartParity
        {
         UART_PARITY_NONE,
         UART_PARITY_EVEN,
         UART_PARITY_ODD,
         UART_PARITY_SPACE,
         UART_PARITY_MARK,
         UART_PARITY_LAST,
         UART_PARITY_MAP  = 0x7F
        } stop;
        enum eUartData
        {
         UART_ DATALENGTH_AUTOFRAME,
         UART_DATALENGTH_5   = 5,
         UART_DATALENGTH_6,
         UART_DATALENGTH_7,
         UART_DATALENGTH_8,
         UART_DATALENGTH_16  = 16,
         UART_DATALENGTH_LAST,
         UART_DATALENGTH_MAP = 0x7F,
        } data;
} sUartLc_t, *psUartLc_t;
```

Fields

op:

This field describes the sub operation to be executed either to set a line coding configuration or to get the current line coding configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

Note: *UART service provider shall assume a set operation when this structure is embedded in a sUartSettings_t one.*

valid_fields:

This field describes the validity of the others sUartLc_t' fields. One bit affects one structure's field. Setting a bit to one means the associated field is valid and shall be taken into account by the UART service provider. Consequently the values are as follows:

- UART_LC_RATE: describes the validity of the rate structure's field.
- UART_LC_STOP: describes the validity of the stop structure's field.
- UART_LC_PAR: describes the validity of the parity structure's field.
- UART_LC DATA: describes the validity of the data structure's field.
- UART_LC_ALL: OR operation of the 4, previously described bits.

Note: *Any combination (up to 16) of those four bits is valid. Setting the four bits to 0 is allowed but shall not have any impact on the current line coding configuration.*

rate:

This field describes the transmission rate to be applied by the UART service provider.

See also eUartRate_t enumerator description for further information about the rate selection.

stop:

This field describes the stop bits configuration to be applied by the UART service provider. According to the UART service provider stop bit capability, the following configuration are supported:

- UART_STOP_BIT_1: each transmitted byte is tailed by 1 stop bit.
- UART_STOP_BIT_2: each transmitted byte is tailed by 2 stop bits.
- UART_STOP_BIT_1_5: each transmitted byte is tailed by 1.5 stop bits.

Note: *Attempting to set a stop bits configuration not indicated by the stop bits capability or not known shall generate an error.*

See also Product Technical Specification for further information about the stop / data bits supported configuration combinations.

parity:

This field describes the parity bit configuration to be applied by the UART service provider. According to the UART service provider parity bit capability, the following configuration are supported:

- UART_PARITY_NONE: transmission without parity bit.
- UART_PARITY_EVEN: transmission with parity bit: even parity.
- UART_PARITY_ODD: transmission with parity bit: odd parity.
- UART_PARITY_SPACE: transmission with parity bit: space parity (idle state forced).
- UART_PARITY_MARK: transmission with parity bit: space parity (active state forced).

Note: *Attempting to set a parity bit configuration not indicated by the parity bit capability or not known shall generate an error.*

data:

This field describes the data bits configuration to be applied by the UART service provider. According to the UART service provider data bits capability, the following configuration are supported:

- UART_DATALENGTH_AUTOFRAME: the UART service provider determine by itself character format: amount of data bits / kind of parity / amount of stop bit(s).
- UART_ DATALENGTH_5: transmission of 5 bits characters.
- UART_ DATALENGTH_6: transmission of 6 bits characters.
- UART_ DATALENGTH_7: transmission of 7 bits characters.
- UART_ DATALENGTH_8: transmission of 8 bits characters.
- UART_ DATALENGTH_16: not supported

Note: *Attempting to set a data bits configuration not indicated by the data bits capability or not known shall generate an error.*

See also Product Technical Specification for further information about the stop / data bits supported configuration combinations (including the supported auto-framing combinationl).

4.1.2.4. The sUartEvent_t structure

The UART events setting structure, when the ADL open device service, adl_OpenDevice function, is used (nested in the sUartSettings_t structure) or when the generic G_IOC_EVENT_HANDLERS IO control is used.

Code

```
typedef struct
{
        eGIocSo_t       op;         /* generic get/set operation */
        void*           user_data;
        eUartEvId_t     valid_cb;
        sGCbDesc_t      cb_list[6];
} sUartEvent_t, *psUartEvent_t;
```

Fields

op:

This field describes the sub operation to be executed to either set event handlers configuration or to get the current event handlers configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

Note: *UART service provider shall assume a set operation when this structure is embedded in a sUartSettings_t one.*

user_data:

This field allows the service user to provide the UART SP with a "global" value which shall be given back as first parameter of any valid event handler set in this structure. Valid event handlers are defined by the valid_cb field and the cb_list field content. Setting all the bits of this field to 1 means the UART service provider shall not use this field. Any other value shall be interpreted by the UART service provider as valid.

valid_cb:

This field describes the validity of each entry of the cb_list (see below) field. Bits 0 to 5 are associated to cb_list[0] … cb_list[5]. Setting a 1 at the bit X position means the cb_list[X] content is valid for the UART service provider. At the other hand setting a 0 at the bit X position means the cb_list[X] content shall not taken into account by the UART service provider.

Note: *Setting this field to 0 instructs the UART service provider to uninstall all currently installed event handlers. In this case application does not have to fill in the cb_list[0...5] field.*

See also eUartEvId_t description for further information about UART event handler identification.

cb_list:

This field allows the service user to provide the UART SP with its event handlers and "local" optional values UART service provider is required to give as first parameter when calling event handlers. This field can store the following event handler configuration information:

- cb_list[0]: ON TX COMPLETE.
- cb_list[1]: ON TX EMPTY (UART shift register empty).
- cb_list [2]: ON RX DATA AVAILABLE.
- cb_list [3]: ON SIGNAL STATE CHANGE.
- cb_list [4]: ON ERROR CHANGE.
- cb_list [5]: ON RX COMPLETE.

Important note Service user is not allowed to provide event handlers configuration which will lead to get, simultaneously, two installed:

- TX event handlers (i.e. ON TX COMPLETE and ON TX EMPTY)
- RX event handlers (i.e. ON RX DATA AVAILABLE and ON RX COMPLETE)

See also sGEvent_t for further information about event handler configuration management.

Note: *Uninstalling an event handler is achieved by setting a 1 at the bit X position in the valid_cb field and simultaneously set the evt_hdl field of the cb_list[X] with a NULL value. More than 1 event handler can be simultaneously uninstalled.*

4.1.2.5. The sUartFlowCtrl_t structure

The UART flow control structure, when the IOC_UART_FC IO command is used.

Code

```
typedef struct
{
        eGIocSo_t op;  /* generic get/set operation */
        enum eFcType
        {
          UART_FC_NONE,
          UART_FC_XON_XOFF,
          UART_FC_RST_CTS,
          UART_FC_XON_XOFF_RTS_CTS
        } type[2]; /* [0] DCEbyDTE and [1] DTEbyDCE */
        u8 xon;
        u8 xoff;
} sUartFlowCtrl_t, *psUartFlowCtrl_t;
```

Fields

op:

This field describes the sub operation to be executed either to set flow control configuration or to get the current flow control configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

type:

This field allows the service user to provide the UART SP type of flow control in both transmission directions. The type[0] entry specifies the DCEbyDTE flow control and the type[1] entry specifies the DTEbyDCE flow control.

xon:

This field allows the service user to specify the ASCII code of the xon character. Setting this field to 0 means the UART service provider has to use the default xon character (ASCII DC1).

xoff:

This field allows the service user to specify the ASCII code of the xoff character. Setting this field to 0 means the UART service provider has to use the default xoff character (ASCII DC3).

4.1.2.6. The sUartSsIoc_t structure

The UART signal state structure, when the IOC_UART_SS IO command is used.

Code

```
typedef struct
{
      eGIocSo_t   op;    /* generic get/set operation */
      eUartSs_t   sig_id;
      eUartSs_t   state;
} sUartSsIoc_t, *psUartSsIoc_t;
```

Fields

op:

This field describes the sub operation to be executed either to signal state configuration or to get the current signal state configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

sig_id:

Allows the application to specify the identities of the signal to be set or retrieved.

See also eUartSs_t for further information about signal state management.

state:

Allows the application to specify / retrieve the state of the signals identified in the sig_id field.

See also eUartSs_t for further information about signal state management.

4.1.2.7. The sUartFd_t structure

The UART FIFO configuration structure, when the IOC_UART_FD IO command is used.

Code

```
typedef struct
{
        eGIocSo_t op;     /* generic get/set operation */
        enum
        {
         UART_FD_DEPTH_0,
         UART_FD_DEPTH_1,
         UART_FD_DEPTH_2,
         UART_FD_DEPTH_3,
         UART_FD_DEPTH_4,
         UART_FD_DEPTH_5
        } rx_size;        // only valid in case mode set to UART_FD_BOTH_ON
        enum
        {
         UART_FD_BOTH_OFF,
         UART_FD_TX_ON,
         UART_FD_BOTH_ON,
         UART_FD_MAP = 0x7000
        }mode;
} sUartFd_t,   *psUartFd_t;
```

Fields

op:

This field describes the sub operation to be executed either to signal state configuration or to get the current signal state configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

rx_size:

Allows the service user to configure, according to its capabilities, the UART service provider's reception trigger level.

See also sUartCap_t for further information about the supported reception trigger level.

mode:

Allows the service user to configure, according to its capabilities, the UART service provider' FIFO. Following possibilities are offered to the service user:

- UART_FD_BOTH_OFF: RX and TX UART FIFO are disabled.
- UART_FD_TX_ON: UART service provider uses its internal transmission FIFO.
- UART_FD_BOTH_ON: UART service provider uses its internals transmission & reception FIFO.

4.1.2.8. The sUartCbOssc_t structure

The UART signal states notification structure, when the On Signal State Changes notification is fired.

Code

```
typedef struct
{
        eUartSs_t delta;
        eUartSs_t state;
} sUartCbOssc_t, *psUartCbOssc_t;
```

Fields

delta:

Identifies which signals are concerned by the state change.

A bit set to 1 indicates the corresponding signal state has changed. More than one signal state change can be encoded (bits "oring").

See also eUartSs_t description for further information about UART signal identification.

state:

Contains the modified signal state. Only signals identified by the delta field are relevant.

A bit set to one indicates the corresponding signal is active.

A bit set to zero indicates the corresponding signal is inactive.

4.1.3. Enumerators

4.1.3.1. The eUartEvId_t type

This enumeration lists the UART event handler identities.

Code

```
typedef enum
{
        UART_CB_ON_TX_COMPLETE          = G_CB_LAST,
        UART_CB_TX_EMPTY                = 2,
        UART_CB_ON_RX_DATA_AVAILABLE    = 4,
        UART_CB_ON_SIG_STATE_CHANGE     = 8,
        UART_CB_ON_ERROR                = 16,
        UART_CB_ON_RX_COMPLETE          = 32,
        UART_CB_ON_ALL                  = 63,
        UART_CB_MAP                     = 0x70000000L
} eUartEvId_t;
```

Description

UART_CB_ON_TX_COMPLETE

Allows the application to configure (install / uninstall) the ON TX COMPLETE event handler.

UART_CB_ON_TX_EMPTY

Allows the application to configure (install / uninstall) the ON TX EMPTY event handler.

UART_CB_ON_RX_DATA_AVAILABLE

Allows the application to configure (install / uninstall) the ON RX DATA AVAILABLE event handler.

UART_CB_ON_SIG_STATE_CHANGE

Allows the application to configure (install / uninstall) the ON SIG STATE CHANGE event handler.

UART_CB_ON_ERROR

Allows the application to configure (install / uninstall) the ON ERROR event handler.

UART_CB_ON_RX_COMPLETE

Allows the application to configure (install / uninstall) the ON RX COMPLETE event handler.

Note: *Additional value G_CB_ON_NOTHING is implicitly defined as this enumeration is derived from the eGEvId_t generic one. It allows the application to erase all currently installed event handlers at the UART service provider side.*

4.1.3.2. The eUartRate_t type

This enumeration lists the UART the available transmission speeds.

Code

```
typedef enum
{
      UART_RATE_UNDEF       = 0x00000000,
      UART_RATE_AUTO        = 0x00000001,
      UART_RATE_300         = 0x00000002,
      UART_RATE_600         = 0x00000004,
      UART_RATE_1200        = 0x00000008,
      UART_RATE_2400        = 0x00000010,
      UART_RATE_4800        = 0x00000020,
      UART_RATE_9600        = 0x00000040,
      UART_RATE_19200       = 0x00000080,
      UART_RATE_38400       = 0x00000100,
      UART_RATE_57600       = 0x00000200,
      UART_RATE_115200      = 0x00000400,
      UART_RATE_230400      = 0x00000800,
      UART_RATE_460800      = 0x00001000,
      UART_RATE_921600      = 0x00002000,
      UART_RATE_1_84_M      = 0x00004000,
      UART_RATE_3_25_M      = 0x00008000,
      UART_RATE_USER_DEF    = 0x00010000,
      UART_RATE_AB          = 0x00020000
} eUartEvId_t;
```

Description

UART_RATE_UNDEF

Returned by the UART service provider in case the service user previously asked for the UART_RATE_AUTO and no characters were received preventing the UART service provider to detect the transmission speed. When application attempts to set this value the UART service provider shall return an error.

UART_RATE_AUTO

Allows the application to configure the UART service provider (according to its capabilities) to detect the transmission speed on character reception.

UART service provider shall generate an error in case service user attempt to "or" any other value with this one.

From UART_RATE_300

To UART_RATE_3_25_M

Allows the application to configure the UART service provider (according to its capabilities) speed from 300 bps to 3,25 Mbps.

UART_RATE_USER_DEF value shall not be "ored" with any value in the range [UART_RATE_300…UART_RATE_3_M].

UART service provider shall generate an error in case more than one discrete speed is set by the service user.

UART_RATE_USER_DEF

Allows the application to configure the UART service provider (according to its capabilities) speed in an open way. This value has just to be "ored" with the actual speed wished by the service user.

Set, by the UART service provider, and "ored" with the actual transmission speed on service user interrogations (get operations).

UART_RATE_AB

Returned by the UART service provider, in case the service user previously put the UART service provider in UART_RATE_AUTO mode and transmission speed was successfully detected.

UART service provider shall generate an error in case service user attempts to set this value.

4.1.3.3. The eUartIoCmd_t type

This enumeration lists the available IO commands identities for configure, or obtain information from, the UART service provider. Values of this enumeration have to be used when IO control operation is invoked.

Code

```
typedef enum
{
      IOC_UART_EH    = G_IOC_EVENT_HANDLERS
      IOC_UART_CAP   = G_IOC_CAPABILITIES,
      IOC_UART_LC    = G_IOC_LAST,
      IOC_UART_SS,
      IOC_UART_FL,
      IOC_UART_FC,
      IOC_UART_TE,
      IOC_UART_SM,
      IOC_UART_ER,
      IOC_UART_TO,
      IOC_UART_FD
} eUartIoCmd_t;
```

Description

IOC_UART_EH

Allows the application to set or get the event handling parameters.

See also sUartEvent_t for further information about event handler configuration.

IOC_UART_CAP

Allows application to get the UART service provider capabilities.

See also sUartCap_t for further information about event handler configuration.

IOC_UART_LC

Allows the application to set or get the line coding parameters (speed, data bits amount, parity type, stop bits amount).

See also sUartLc_t for further information about line coding configuration.

IOC_UART_SS

Allows the application to set or get the UART signal states.

See also sUartSsIoc_t for further information about line coding configuration.

IOC_UART_FC

Allows the application to set or get the flow control parameters.

See also sUartFlowControl_t for further information about line coding configuration.

IOC_UART_TE

DEPRECATED. Attempting to use this value shall generate an error.

IOC_UART_SM

DEPRECATED. Attempting to use this value shall generate an error.

IOC_UART_ER

DEPRECATED. Attempting to use this value shall generate an error.

IOC_UART_TO

DEPRECATED. Attempting to use this value shall generate an error.

IOC_UART_FD

Allows the application to set or get the UART FIFO control parameters.

4.1.3.4. The eUartFl_t type

This enumeration lists the flushing operations implemented by the UART service provider. The parameter for the IOC_UART_FL IO command.

Code

```
typedef enum
{
        UART_FLUSH_RX  = 1,
        UART_FLUSH_RX,
        UART_FLUSH_ALL
} eUartFl_t;
```

Description

UART_FLUSH_RX

Allows the application to flush the UART RX FIFO content.

UART_FLUSH_RX

Allows the application to flush the UART TX FIFO content.

UART_FLUSH_ALL

Allows the application to flush the UART TX & RX FIFO content.

4.1.3.5. The eUartSs_t type

This enumeration lists the available UART signals identifiers.

Code

```
typedef enum
{
      UART_SIG_DCD   = 0x0001,
      UART_SIG_DSR   = 0x0002,
      UART_SIG_DTR   = 0x0004,
      UART_SIG_RI    = 0x0008,
      UART_SIG_BREAK = 0x0010,
      UART_SIG_RTS   = 0x0020,
      UART_SIG_CTS   = 0x0040,
      UART_SIG_ALL   = 0x007F
} eUartSs_t;
```

Description

UART_SIG_DCD

Allows the service user to:

Get the DCD state when UART service provider is acting as DTE or DCE.

Set the DCD state when UART service provider is acting as DCE.

Not supported when the UART service provider is acting in NULL MODEM mode.

UART_SIG_DSR

Allows the service user to:

Get the DSR state when UART service provider is acting as DTE or DCE.

Set the DSR state when UART service provider is acting as DCE.

Not supported when the UART service provider is acting in NULL MODEM mode.

UART_SIG_DTR

Allows the service user to:

Get the DTR state when UART service provider is acting as DTE or DCE.

Set the DTR state when UART service provider is acting as DTE.

Not supported when the UART service provider is acting in NULL MODEM mode.

UART_SIG_RI

Allows the service user to:

Get the RI state when UART service provider is acting as DTE or DCE.

Set the RI state when UART service provider is acting as DTE.

Not supported when the UART service provider is acting in NULL

MODEM mode.

UART_SIG_BREAK

Allows the service user to set a "break condition" on the TX line. Getting the BREAK signal is not possible and returns always 0.

UART_SIG_RTS

Allows the service user to set the RTS signal state.

UART_SIG_CTS

Allows the service user to get the CTS signal state.

4.1.3.6. The eUartErr_t type

This enumeration lists the available UART error codes.

Code

```
typedef enum
{
        UART_OE     = 0x02,
        UART_PE     = 0x04,
        UART_FE     = 0x08,
        UART_BE     = 0x10,
        UART_TX_TO  = 0x20
} eUartSs_t;
```

Description

UART_OE

UART overrun error identifier

UART_PE

UART parity error identifier

UART_FE

UART framing error identifier

UART_BE

UART break error identifier.

UART_TX_TO

This error identifier is deprecated and is not any longer managed by the UART service provider.

4.1.4. Operations

There are two types of operations defined by the Open UART Interface:

- Requests: Allow a service user to directly handle any UART service provider.
- Notifications: Allow a UART service provider to notify event occurrences to service user.

Note: *Before requesting, or being notified by, a UART service provider an Open AT® application shall retrieve a, direct, access (by using the ADL Open Device service) to this UART service provider.*

5 request functions are offered:

- An **open** function to:
  - Optionally retrieve the UART service provider's interface (through a generic interface container) **and**
  - retrieve a unique UART service provider reference (handle) which shall be subsequently provided as parameter to the rest of the request functions **and**
  - optionally install event handlers to manage the UART service provider notifications **and**
  - optionally set the line coding parameters **and**
  - optionally retrieve the UART service provider's capabilities.
- A **read** function to retrieve characters received by the UART service provider.
- A **write** function to instruct the UART to send characters over the serial line.
- An **io_control** function to configure, or get information from, the UART service provider.
- A **close** function to release the UART interface (and the handle previously allocated).

6 notifications are offered to the UART service user to inform it of the occurrence of:

- The completion of the current emission (write completion at the byte level).
- The completion of the current emission (write completion at the bit level).
- The availability of received data (UART service provider uses an internal buffer).
- The changing in the signal states.
- Errors (parity, framing, break detection, overrun).
- The completion of the current reception (reception in zero copy mode)

Note: *Calling request functions while application event handlers are running is not supported. Doing such a call might generate system instabilities.*

4.1.4.1. The open function

There is no, at strictly speaking, specific function provided to open (get a direct access to) a UART service provider. The ADL Open Device service provides a generic function allowing getting, direct, access to numerous kinds of service providers. Hereafter a description of what is needed to open a UART service provider.

Prototype

```
s32 adl_OpenDevice ( eDfClid_t    dev_clss_id,
                     void *       param );
```

Parameters

dev_clss_id:

The device class identifier the service provider to be opened belongs to. To open a UART service provider application has to use the **DF_UART_CLID** value.

param:

Service provider configuration, to be defined accordingly to the dev_clss_id parameter in the UART case address of a sUartSettings_t structure is required.

Returned values

- Handle: A positive UART service provider handle on success, to be used in further Open UART service function calls.
- 0: UART service provider opening failed.

4.1.4.1.1. Example: How to open the UART2 (57600,N81)

```
#include "adl_OpenDevice.h"
#include "wm_uart.h"

static psGItfCont_t uart_if;
static u32 uart2_hdl;

void adl_main( adl_InitType_e InitType )
{
   sUartSettings_t settings;
   sUartLc_t       line_coding;

   // Set the line coding parameters
   line_coding.valid_fields = UART_LC_ALL;
   line_coding.rate = (eUartRate_t)( UART_RATE_USER_DEF | 57600 );
   line_coding.stop = UART_STOP_BIT_1;
   line_coding.data = UART_DATALENGTH_8;
   line_coding.parity = UART_PARITY_NONE;

   // UART2 will be opened in NULL MODEM role / with synchronous read/write
   settings.identity = "UART2";
   settings.role = UART_ROLE_NM;
   settings.capabilities = NULL;
   settings.event_handlers = NULL;
   settings.interface = &uart_if;
   settings.line_coding = &line_coding;

   uart_hdl = adl_OpenDevice( DF_UART_CLID, &settings );
   if( !uart_hdl )
   {
      // UART2 opening failed...
     return;
   }

   // UART2 successfully opened, write some bytes
   uart_if.write( uart_hdl, "Tx Some bytes", 13 );
   …
}
```

4.1.4.2. The read request

This function allows the application to read the bytes received by the UART service provider. Before using this function the application shall open the UART service provider (shall own the UART interface as well as a valid UART handle).

Two running modes are supported: zero copy (ZC) or non zero copy (NZC) modes. The running mode selection is achieved by the application when it provides the UART SP with **either** the On Rx Data Complete (ZC selected) **or** the On Rx Data Available (NZC selected) event handlers.

When UART SP is running in (ZC) mode the read function runs in an asynchronous way. Application provisions a read providing the UART SP with reception buffer address and size information. UART SP returns an operation pending indication. While an asynchronous read operation is pending application is allowed to invoke the read function which will have the following effects:

- Current reception buffer address and size information are erased by the UART SP. According to the new parameters provided the asynchronous read operation is:
  - Either cancelled in case reception buffer address and size are set to NULL.
  - Or continued in case buffer address and size are set to non NULL values.

UART SP completes the pending read operation by firing the On Rx Data Complete event.

When UART SP is running in (NZC) mode the read function runs in a synchronous way. Application should trigger the read function call after UART SP called its On Rx Data Available event handler. Application provides UART SP with the reception buffer address and size parameters. These parameters shall contain non NULL values otherwise the UART SP returns an error. UART SP returns the amount of data actually stored in the reception buffer. Application should call the read function while UART SP returns a non NULL amount of copied bytes.

Prototype

```
eChStatus_t read (    u32       Handle,
                      void *    pData,
                      u32       len );
```

Parameters

Handle:

Handle of the UART service provider previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

pData:

Address where the received data shall be put. NULL value is not supported when UART SP is running in NZC mode.

len:

Size (in bytes) of the memory area provided to store the received data. NULL value is not supported when UART SP is running in NZC mode.

Returned values

Synchronous mode

- Any positive value greater or equal than CH_STATUS_NORMAL and strictly lower than CH_STATUS_PENDING indicates the amount (including 0) of bytes copied from the UART service provider to the application reception buffer.
- CH_STATUS_ERROR: either pData or len or both parameters set with NULL values, or invalid UART service provider handle.

Asynchronous mode

- CH_STATUS_ERROR: Invalid UART service provider handle.
- CH_STATUS_NORMAL: Asynchronous read cancellation successfully completed.
- CH_STATUS_PENDING: Asynchronous read operation is pending.

Both modes

- CH_STATUS_ERROR: no reception event handler installed.

4.1.4.2.1. Example: how to select asynchronous/synchronous read operation

```
#include "adl_OpenDevice.h"
#include "wm_uart.h"

static psGItfCont_t uart_if;
static u32 uart2_hdl;
static u8 rx_buf[ 256 ];

static void on_rxc_handler( u32 user_data, psGData_t evt_par){
   // Code to obtain new Rx buffer and size to be returned to the UART SP
   // Just in case where there is no more available reception buffer
   *(u64*)evt_par.buf = 0LL;
}
```

```
static void on_rxda_handler( u32 user_data, psGData_t evt_par){
   // Code to set an ADL event to unlock an synchronous read
}

void adl_main( adl_InitType_e InitType ) {
   sUartEvent_t evt_setting;
   u8* p_rx_buf;
   u32 nb_tb_read;
   u32 nb_read;
   …
   // somewhere in the application
   // refer to Example for the UART2 opening code
   // Here the UART2 has been successfully opened (uart_itf & uart2_hdl valid)

   // select the asynchronous read operation assuming there is no RX event
handler
   // installed
   evt_setting.op = G_IOC_OP_SET;
   evt_setting.valid_cb = UART_CB_ON_RX_COMPLETE;
   evt_setting.user_data = (void*)-1L;  // not used
   evt_setting.cb_list[5].evt_hdl = (pGEvtNotif_t)on_rxc_handler;
   evt_setting.cb_list[5].user_data = (void*)-1L; // not used
   if( uart_if.io_control( uart_hdl, IOC_UART_CB, &evt_setting ) ){
      // an error occurred …
      return;
   }
   if(CH_STATUS_PENDING != uart_if( uart2_hdl, rx_buf, sizeof(rx_buf))){
      // an error occurred …
   }
   …
   // somewhere in the application switch from asynchronous to synchronous read
   p_rx_buf = rx_buf;
   amount_tb_read = sizeof( rx_buf);

   evt_setting.op = G_IOC_OP_SET;
   evt_setting.valid_cb = (eUartEvId_t)(UART_CB_ON_RX_DATA_AVAILABLE |
                UART_CB_ON_RX_DATA_AVAILABLE);
   evt_setting.user_data = (void*)-1L;  // not used
   evt_setting.cb_list[5].evt_hdl = NULL;
   evt_setting.cb_list[2].evt_hdl = (pGEvtNotif_t)on_rxda_handler;
   evt_setting.cb_list[2].user_data = (void*)-1L; // not used
   if( uart_if.io_control( uart_hdl, IOC_UART_CB, &evt_setting ) )
   {
      // an error occurred …
      return;
   }
   // Code to wait an ADL Event set by the On Rx Data Available handler
   …
   while(0 != (nb_read = uart_if.read( uart2_hdl,p_rx_buf, nb_tb_read))){
      nb_tb_read -= nb_read;
      p_rx_buf += nb_read;
   }
}
```

4.1.4.3. The write request

This function allows the application to instruct the UART service provider to send bytes. Before using this function the application shall open the UART service provider (owning the UART interface as well as a valid UART handle).

Two running modes are supported: Asynchronous (A) and synchronous (S) modes. The running mode selection is achieved by the application when it provides the UART SP with **either** the On TX Complete **or** the On TX Empty event handlers (A). When there is no transmission completion event handler the write operation is executed in synchronously (S).

When UART SP is running in (S) mode the application is blocked while the byte transmission occurs.

When UART SP is running in (A) mode the application enables a write providing the UART SP with the transmission buffer address and size parameters. UART SP returns an operation pending indication. While an asynchronous write operation is pending application is allowed to invoke the write function with both transmission buffer address and size parameters set with a NULL value. As consequence the pending write operation is cancelled.

According to the current transmission event handler installed UART SP completes the pending write operation by firing either the On TX Complete/Empty event.

Prototype

```
eChStatus_t write (   u32       Handle,
                      void *    pData,
                      u32       len );
```

Parameters

Handle:

Handle of the UART service provider previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

pData:

Address of the data block to be sent. NULL value is not supported when UART SP is running in (S) mode.

len:

Size (in bytes) of the data block to be sent. NULL value is not supported when UART SP is running in (S) mode.

Returned values

Synchronous mode

- CH_STATUS_NORMAL operation successfully completed.
- CH_STATUS_ERROR: either pData or len or both parameters set with NULL values.

Asynchronous mode

- CH_STATUS_ERROR: Asynchronous write operation is already pending.
- CH_STATUS_NORMAL: Asynchronous write cancellation successfully completed.
- CH_STATUS_PENDING: Asynchronous write operation successfully started.

Both modes

- CH_STATUS_ERROR: invalid UART service provider handle.

4.1.4.4. The io_control request

This function allows to set configuration, or to get configuration information from the UART service provider. Before using this function the application shall open the UART service provider (shall own the UART interface as well as a valid UART handle).

This function is generic and supports several IO commands. To choose among the supported IO commands the application has to set the Cmd parameter with a supported IO command identifier (see also eUartIoCmd_t for further information about the supported IO commands).

Prototype

```
eChStatus_t io_control ( u32       Handle,
                         u32       Cmd,
                         void*     pParam );
```

Parameters

Handle:

Handle of the UART service provider previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

Cmd:

IO command identifier.

See also eUartIocId_t for further information about the supported UART IO commands.

pParam:

IO command parameter. Type of this parameter depends on the Cmd parameter value. Following sub clauses will detail the actual type to be used.

Returned values

- Depend on the IO command type. Following sub clauses will detail actual return values.

Note: *IO Commands support can be obtained during the opening stage or by using the IOC_UART_CAP IO command (which is mandatorily implemented). Attempting to invoke an unsupported IO command shall generate an error.*

4.1.4.4.1. The IOC_UART_EH IO command

This function allows setting events handling configuration, or to get configuration information about the events handling configuration currently used by the UART service provider.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_EH.

pParam:

Address of a sUartEvent_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / unknown operation / pParam set to NULL / invalid configuration.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.2. The IOC_UART_CAP IO command

This function allows getting the capabilities of the UART service provider.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_CAP.

pParam:

Address of a sUartCap_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle // pParam set to NULL.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.3. The IOC_UART_LC IO command

This function allows setting the line coding configuration or getting the current line coding configuration used by the UART SP.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_LC.

pParam:

Address of a sUartLc_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / unsupported operation / pParam set to NULL / invalid configuration.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.4. The IOC_UART_SS IO command

This function allows setting the signal state configuration or getting the current signal state configuration used by the UART SP.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_LC.

pParam:

Address of a sUartSsIoc_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / unsupported operation / pParam set to NULL / invalid configuration.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.5. The IOC_UART_FL IO command

This function allows flushing the UART service provider transmission and/or reception FIFO.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_FL.

pParam:

Value from the eUartFl_t enumerated.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / invalid configuration.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.6. The IOC_UART_FC IO command

This function allows setting the flow control configuration or getting the current flow control configuration used by the UART service provider.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_FC.

pParam:

Address of a sUartFlowControl_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / unsupported operation / invalid configuration / pParam set to NULL.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.4.7. The IOC_UART_FC IO command

This function allows setting the FIFO (RX & TX) configuration or getting the current FIFO configuration used by the UART service provider.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_UART_FD.

pParam:

Address of a sUartFd_t structure.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle / unsupported operation / invalid configuration / pParam set to NULL.
- CH_STATUS_NORMAL: command succeeded.

4.1.4.5. The Close request

This function allows the application to stop all pending, read and write operations and to release the UART SP. Before using this function the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle).

Prototype

```
eChStatus_t close (   u32    Handle );
```

Parameters

handle:

Handle of the UART service provider previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

Returned values

- CH_STATUS_ERROR: invalid UART service provider handle.
- CH_STATUS_NORMAL: close operation successfully completed.

4.1.4.6. The On TX Complete notification handler

This notification allows the application to be aware of the completion of the pending asynchronous write operation. It occurs when the last byte, of the previously submitted data block, is being transmitted by the UART service provider.

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On TX Complete notification handler.

Prototype

```
void on_txc (  void*         user_data,
               psGData_t     evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on every occurrence of the transmission completion.

evt_param:

Address of a sGData_t structure allowing the application to provide the UART service provider with address and size parameters of a new data block to be transmitted. In case application does not have any more data block to be transmitted it shall set the buf **and** len fields of the sGData_t structure to a NULL value.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.1.4.7. The On TX Empty notification handler

This notification allows the application to be aware of the completion of the pending asynchronous write operation. It occurs when the last bit, of the previously submitted data block, is being transmitted by the UART service provider.

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On TX Empty notification handler.

Prototype

```
void on_txe ( void*       user_data,
              psGData_t   evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on each transmitter empty notification.

evt_param:

Address of a sGData_t structure allowing the application to provide the UART service provider with address and size parameters of a new data block to be transmitted. In case application does not have any more data block to be transmitted it shall set the buf **and** len fields of the sGData_t structure to a NULL value.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.1.4.8. The On Rx Complete notification handler

This notification allows the application to be aware of the completion of the pending asynchronous read operation. It occurs when;

- Either the previously provided Rx buffer is full.
- Or on reception timeout. Which means at least 1 character has been stored in the previously provided Rx buffer and no activity occurred on the RX line for a time comprises in the range [3.5 ··· 4.5] characters time.

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On Rx Complete notification handler.

Prototype

```
void on_rxc ( void*       user_data,
              psGData_t   evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on each receive complete notification.

evt_param:

Address of a sGData_t structure allowing the application to provide the UART service provider with address and size parameters of a new reception data block. In case application does not have any more available reception data block it shall set the buf **and** len fields of the sGData_t structure with NULL values.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.1.4.9. The On Rx Data Available notification handler

This notification allows the application to trigger a (NZC) read. It occurs when at least one byte has been received by the UART service provider. While application has not extracted all the received bytes (in other words while the synchronous read function does not return 0) this notification will not be re-generated.

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On Rx Data Available notification handler.

Prototype

```
void on_rxda ( void*         user_data,
               psGData_t     evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on each occurrence of the received data available notification.

evt_param:

This parameter is mandatory but UART service provider does not use it. Application shall ignore its content.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.1.4.10. The On Signal State Change notification handler

This notification allows the application to be aware of any UART signal state change. It occurs when at least one input signal state, from the embedded module point of view, is modified.

In DCE mode: the DTR signals state changes are notified.

In DTE mode: the RI, DCD and DSR signals state changes are notified.

In Null Modem, DTE and DCE modes: the CTS and BREAK signals state changes are notified.

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On Rx Data Available notification handler.

Prototype

```
void on_ssc (    void*                 user_data,
                 psUartCbOssc_t        evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on each occurrence of the received data available notification.

evt_param:

Provide to application the identities and current states of the modified signals.

See also sUartCbOssc_t for further information about signal identities and states.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.1.4.11. The On Error notification handler

This notification allows the application to be informed of UART errors occurrences. It is fired when errors occur at the UART service provider side..

Before being notified the application shall open the UART service provider (must own the UART SP interface as well as a valid UART SP handle) and configure the UART service provider with its On Rx Data Available notification handler.

Prototype

```
void on_ssc (    void*           user_data,
                 eUartErr_t      evt_param );
```

Parameters

user_data:

Generic 32 bits information the application previously provided during the event handler configuration. The UART service provider is required to give back this information to the application on each occurrence of the received data available notification.

evt_param:

Provide to application the error identities.

See also eUartErr_t for further information about error identities.

Returned values

Not Applicable.

Note: *This handler is called in an interrupt context. The stack size for this context is 1024 bytes, defined in ADL. Consequently, this handler must not be used to make heavy operations or allocate large space memory.*

Note: *Even if it is strongly not recommended to use traces in interrupt handlers, if they are temporarily used for debug purpose, traces will be emited on the "LLH" flow. Such event handlers are considered as Low Level Handlers anywhere in the API (adl_ctxGetContextID() returns ADL_CTX_LOW_LEVEL_IRQ_HANDLER), and all related restrictions apply.*

4.2. Open USB Interface

ADL provides Open USB Interface to give access to the module USB Core Layer Service Provider. The USB Core Layer Service Provider should be understood as the software component handling the USB Device framework. The USB CL SP acronym will be used in the rest of this chapter.

By default (i.e. without any Open AT® application or in case such an application does not use the Open USB Interface) the module USB CL SP is managed by the Sierra firmware only.

When an USB CL SP is handled by the Sierra firmware, it is not available for an Open AT® application.

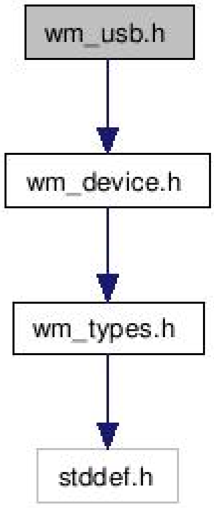
Similarly, when an USB CL SP is handled by an Open AT® application, it is not available for the Sierra firmware.

In both the above cases, any attempt to get access to already used USB CL SP will returns with an error to the requestor.

4.2.1. Required Header File

The Open USB Interface is defined by the following header file **wm_usb.h**

It should be noticed that the wm_usb.h header is self-sufficient (or auto-compilable), that means there is no need (for the service user) to include any other header files (except for the opening stage, where the adl_OpenDevice.h file is required) to get access to the USB CL SP. The way the wm_usb.h header file is built is illustrated by the following dependency graph.



4.2.2. Data Structures

4.2.2.1. The sOpUsbSettings_t Structure

The USB CL SP configuration structure, when the ADL open device service (adl_OpenDevice) function is used.

```
typedef struct
{
        // GENERIC Fields, is a sGenSettings_t type (wm_device.h)
        char* identity;
        psObUsbEvent_t event_handlers;
        ppsGItfCont_t interface;
        // END of the GENERIC fields

        // SPECIFIC IN parameters
        u32 ousb_itf_version;

        // To support the enumeration stage
        psOpUsbDevInfo_t p_device_fs;
        psOpUsbDevInfo_t p_device_hs;
        char** ad_if_id_list; // A char* array[] NULL TERMINATED

        // SPECIFIC IN/OUT parameters
        psOpUsbCapabilities_t capabilities;
} sOpUsbSettings_t, *psOpUsbSettings_t;
```

Fields:

identity:

This field is mandatory present. It allows the service user to choose the USB CL SP it wants to work with. Setting this field to NULL generates an error. Any non null value is considered by the USB CL SP as a pointer of a null terminated string. Supported string contents are listed hereafter:

- "UDEV0": to get access to the module USB 2.0 in Full Speed mode.

Attempting to provide an identity not contained in the previous list generates an error.

Note: *This list is not exhaustive and will be updated when new USB CL SP will be available.*

event_handlers:

This field is optional. It allows the service user to provide its event handlers during the opening stage. Please note the IOC_USB_EH IO control cannot be used to later on set the event handler configuration.

See also sOpUsbEvent_t description for further information about event handler configuration management.

interface:

This field is optional. It allows the service user to dynamically retrieve the interface of the USB CL SP inside a generic interface container.

Setting this field to NULL generates an error. Any non NULL value is considered by the USB CL SP as the address of a pointer of sGItfCont_t structure (see wm_device.h file).

See also sGItfCont_t description for further information about generic interface containers and interface retrieving at run time.

ousbitf_version:

This field will be mandatory. It allows the service user to use USB CL SP versioned interfaces. **This field is currently not used.**

p_device_fs:

This field is optional. It allows the service user to describe its USB function for a *Full Speed (FS)* USB device controller. This field shall not be set to NULL expect in the cases listed hereafter:

- The device controller used is a USB HS one and given the p_device_hs field is set to a non NULL value or
- The ad_if_id_list field contains, at least, one USB function/class identifier.

Attempting to set this field to NULL when conditions listed below are not met generates an error.

Any non null value is considered by the USB CL SP as a pointer of sOpUsbDevInfo_t structure.

p_device_hs:

This field is optional. It allows the service user to describe its USB function for a *High Speed (HS)* USB device controller. This field shall not be set to NULL expect in the cases listed hereafter:

- The device controller used is a USB FS one and given the p_device_fs field is set to a non NULL value or
- The ad_if_id_list field contains, at least, one USB function/class identifier.

Attempting to set this field to NULL when conditions listed below are not met generates an error.

Any non null value is considered by the USB CL SP as a pointer of sOpUsbDevInfo_t structure.

Note: *Some USB device controllers are capable to work either in Full or High speed mode. In case such a controller would be used within the Sierra Wireless module both p_device_fs and p_device_fs fields should be set to a non NULL value. The USB CL SP shall autonomously handle the Device Qualifier and Other Speed Configuration USB descriptor building, querying and switching processes.*

See also sOpUsbDevInfo_t description for further information about the Device Information setting.

ad_if_id_list:

This field is optional. It allows service user to build an USB composite function by reusing USB functions/classes already coded and located at the Sierra firmware side**.** Hereafter the list of identifiers for such supported functions/classes:

“UFLCDC”: to reuse the existing CDC ACM (serial port) class.

Attempting to provide an identity not contained in the previous list generates an error.

Note: *This list is not exhaustive and will be updated when new USB firmware functions/classes will be available.*

capabilities:

This field is optional. It allows the USB CL SP to return its capabilities to the service user during the opening stage. Please note the IOC_USB_CAP IO control can also be used to retrieve the capabilities after the opening stage. **This field is currently not used.**

4.2.2.2. The sOpUsbCapabilities_t Structure

The USB SP CL capabilities structure, when the ADL open device service (adl_OpenDevice) function is used (nested in the sOpUsbSettings_t structure) or when the generic IO control G_IOC_CAPABILITIES is used.

```
typedef struct
{
      u32 currently_not_identified;
} sOpUsbIocCapabilities_t, *psOpUsbIocCapabilities_t, sOpUsbCapabilities_t,
      *psOpUsbCapabilities_t ;
```

Fields:

currently_not_identified:

This field is to be defined.

4.2.2.3. The sOpUsbDevInfo_t Structure

The Open USB **Device** Information set, when the ADL open device service, adl_OpenDevice function, is used (nested in the sOpUsbSettings_t structure) by the USB CL SP to build the Device and Device qualifier standard descriptors.

```
typedef struct
{
   u8   bDeviceClass;
   u8   bDeviceSubClass;
   u8   bDeviceProtocol;
   u8   bNumConfigurations;
   u16  bcdDevice;
   u16  idVendor;
   u16  idProduct;
   enum eDevInfoCust
   {
      CF_NOCUSTOMIZATION    = 0x0000,
      CF_BDEVICECLASS       = 0x0001,
      CF_BDEVICESUBCLASS    = 0x0002,
      CF_BDEVICEPROTOCOL    = 0x0004,
      CF_BNUMCONFIGURATIONS = 0x0008,   // Not implemented
      CF_IDVENDOR           = 0x0010,
      CF_IDPRODUCT          = 0x0020,
      CF_BCDDEVICE          = 0x0040,
      CF_ACONFIG            = 0x0080,
      CF_IMANUFACTURER      = 0x0100,
      CF_IPRODUCT           = 0x0200,
      CF_ISERIALNUMBER      = 0x0400,
      CF_DEVICE_ALL         = 0x07FF,
      CF_MAP                = 0x7FFF
   } cust_fields;

   psOpUsbConfInfo_t a_config  /*[bNumConfigurations]*/;
   ascii* iManufacturer;
   ascii* iProduct;
   ascii* iSerialNumber;
} sOpUsbDevInfo_t, *psOpUsbDevInfo_t;
```

Fields:

bDeviceClass:

bDeviceSubClass:

bDeviceProtocol:

bNumConfigurations:

bcdDevice:

idVendor:

idProduct;

iManufacturer:

iProduct:

iSerialNumber:

For the meaning of the above fields see the USB specification revision 2.0 – table 9-8. Standard Device Descriptor.

Note: *Any character string is accepted, when sets to NULL or points to an empty string the corresponding fields in the USB standard Device descriptor are set to 0.*

cust_fields:

This field contains the device information set customization descriptor. In case the Service User just wants to customize the USB Device descriptor of a class implemented at the firmware side it shall set this field with the needed pre-defined constant to indicate to the USB Core Layer which fields of the Device information set must be overwritten.

a_config:

This field contains the address of the array of Open USB Configuration Information set.

Note: *The size of the configuration array is given by the bNumConfigurations field.*

See also sOpUsbConfInfo_t structure for further details about the Open USB Configuration Information set.

4.2.2.4. The sOpUsbEvent_t Structure

The USB CL SP events setting structure, when the ADL open device service, adl_OpenDevice function, is used (then nested in the sOpUsbSettings_t structure).

```
typedef struct
{
       eGIocSo_t op;  /* generic get/set operation */
       void* user_data;
       enum
       {
         OUSB_UNINSTALL_ALL = G_CB_ON_NOTHING,
         OUSB_ON_STATUS     = G_CB_LAST,
         OUSB_ON_REQUEST    = (OUSB_ON_STATUS << 1),
         OUSB_ON_COMPLETE   = (OUSB_ON_REQUEST << 1),
         OUSB_INSTALL_ALL   = (OUSB_ON_STATUS | OUSB_ON_REQUEST |
                              OUSB_ON_COMPLETE),
         USB_ON_MAP         = 0x70000000L
       }
       valid_cb;
       sGCbDesc_t cb_list[3];
} sOpUsbIocEventH_t, *psOpUsbIocEventH_t, *psOpUsbEvent_t;
```

Fields:

op:

This field describes the sub-operation to be executed either *set* event handlers configuration or *get* the current event handlers configuration.

See also eGIocSo_t description for further information about the sub-operation selection.

Note: *USB CL SP shall assume a set sub-operation when this structure is embedded in a sOpUsbSettings_t one. When this structure is used as parameter for the IO control operation IOC_USB_EH only the get sub-operation is supported.*

user_data:

This field allows the service user to provide the USB CL SP with a "global" value (a service user context address for example) which shall be given back as first parameter of any valid event handler set in this structure. Valid event handlers are defined by the valid_cb field and the cb_list field content. Setting all the bits of the *user_data* field to 1 means the USB CL SP shall ignore the field content. Any other value, including NULL, shall be interpreted by the USB CL SP as valid.

valid_cb:

This field describes the validity of each entry of the cb_list (see below) field. Bits 0 to 2 are associated to cb_list[0] … cb_list[2]. Setting a 1 at the bit X position means the cb_list[X] content is valid for the USB CL SP. At the other hand setting a 0 at the bit X position means the cb_list[X] content shall not taken into account by the USB CL SP.

Note: *Setting this field to 0 instructs the USB CL SP to uninstall all currently installed event handlers. In such a case service user is not required to fill in the cb_list[0...2] fields.*

cb_list:

This field allows the service user to provide the USB CL SP with its event handlers and "local" optional values the USB CL SP is required to give as first parameter when calling event handlers. This field can store the following event handler configuration information:

- cb_list[0]: ON STATUS.
- cb_list[1]: ON REQUEST.
- cb_list [2]: ON COMPLETE.

See also sGEvent_t for further information about event handler configuration management.

Note: *The content of the user_data field of each cb_list entry shall not be used by the USB CL SP (as "global" user_data is required to be used).*

4.2.2.5. The sOpUsbConfInfo_t Structure

The Open USB **Configuration** Information set, when the ADL open device service, adl_OpenDevice function, is used (nested in the sOpUsbDevInfo_t structure) by the USB CL SP to build the Configuration and Configuration Other Speed standard descriptors.

```
typedef struct
{
       ascii* iConfiguration;

       enum ebBmAttributes
       {
          BUS_ONLY_POWERED,
          REMOTE_WAKEUP_DISABLED   = BUS_ONLY_POWERED,
          REMOTE_WAKEUP_ALLOWED    = 0x20,
          SELF_POWERED = 0x40
       }bmAttributes;

       // Can be set even if SELF_POWERED is select (dual powering scheme
       support)
       u8  bMaxPower;
       u8 bNumInterfaces; // May be set to NULL for customization purpose.
       enum eConfInfoCust
       {
          CF_BMATTRIBUTES      = 0x01,
          CF_BMAXPOWER         = 0x02,
          CF_CONFIGURATION_ALL = 0x03
       } cust_fields;
       // Array of array of psOpUsbItfInfo_t.  Array's size is given by the
       // bNumInterfaces field. May be set to NULL for customization
       // purpose.
       pppsOpUsbItfInfo_t aap_itf/*[.bNumInterfaces]*/;
} sOpUsbConfInfo_t, *psOpUsbConfInfo_t;
```

Fields:

iConfiguration:

Note: *Any character string is accepted, when sets to NULL or points to an empty string the corresponding field in the USB standard Device descriptor is set to 0.*

bmAttributes:

Note: *The bmAttributes field's content returned to the USB host during the enumeration stage is the result of an AND logical operation between the content of the bmAttributes field of the sOpUsbConfInfo_t structure and the remote wakeup capability of the USB chip of the WCPU® actually used within the customer's application.*

bMaxPower:

bNumInterfaces:

Note: *The bNumInterfaces shall be set to 0 when the application is using the Open USB Service for customizing existing USB function/class's descriptors (CONFIGURATION and DEVICE) located at the firmware side. The customization of the INTERFACE descriptor (located at the firmware side) is currently not supported.*

For the meaning of those fields see the USB specification revision 2.0 – table9-10/table9-11. Standard Configuration Descriptor and Other Speed Configuration Descriptor.

cust_fields:

This field contains the configuration information set customization descriptor. In case the User just wants to customize the USB Configuration descriptor of a class implemented at the firmware side it shall set this field with the needed pre-defined constant to indicate to the USB Core Layer which fields of the configuration information set must be overwritten.

aap_itf:

This field contains the address of the array of array of Open USB Interface Information set. When the bNumInterfaces field contains 0 the content of the aap_itf is useless for the Open USB CL

Note the size of this array of Interface Information set arrays is given by the bNumInterfaces field. Index in the first array's dimension (aap_itf[***first dimension***][y]) is used by the USB CL SP to compute the bInterfaceNumber field of the Standard Interface Descriptor. Index in the second's array dimension (aap_itf[x][***second dimension***]) is used by the USB CL SP to compute the bAlternateSetting field of the Standard Interface Descriptor. The last entry of the second dimension shall always be set to NULL to inform the USB CL SP of the amount of default + alternate interface settings.

See also sOpUsbItfInfo_t structure for further details about the Open USB Configuration Information set.

4.2.2.6. The sOpUsbItfInfo_t Structure

The Open USB **Interface** Information set, when the ADL open device service, adl_OpenDevice function, is used (nested in the sOpUsbConfInfo_t structure) by the USB CL SP to build the Interface standard descriptor.

```
typedef struct
{
   psOpUsbIaInfo_t p_interface_association;
   ascii* iInterface;

   u8 bNumEndpoints;
   u8 bInterfaceClass;
   u8 bInterfaceSubClass;
   u8 bInterfaceProtocol;

   u8* p_classinfo;

   ppsOpUsbEpInfo_t ap_ep/*[bNumEndpoints]*/;
} sOpUsbItfInfo_t, *psOpUsbItfInfo_t, **ppsOpUsbItfInfo_t,
  ***pppsOpUsbItfInfo_t;
```

Fields:

p_interface_association:

For the meaning of this field see the USB specification revision 3.0 – table9-16. Standard Interface Association Descriptor.

iInterface:

Any character string is accepted, when sets to NULL or points to an empty string the corresponding field in the USB standard Device descriptor is set to 0

bInterfaceClass:

bInterfaceSubClass:

bInterfaceProtocol:

bNumEndpoints:

iInterface:

For the meaning of those fields see the USB specification revision 2.0 – table9-12. Standard Device Descriptor

p_classinfo:

Address of an array of bytes. Allows the service user in providing any class specified information. The field shall be set to NULL in case there is no class specified information.

ap_ep:

Address of an array of sOpUsbEpInfo_t pointers. The field shall be set to NULL in case this interface uses the control endpoint only.

See also sOpUsbEpInfo_t for further information about endpoint information set.

4.2.2.7. The sOpUsbEpInfo_t Structure

The Open USB **Endpoint** Information set, when the ADL open device service, adl_OpenDevice function, is used (nested in the sOpUsbItfInfo_t structure) by the USB CL SP to build the Endpoint standard descriptor.

```
typedef struct
{
   Enum
   {
       ZLP,
       NO_ZLP
   }zlp_generation;
   enum
   {
       OUT,
       IN
   }bDirection;
   u8 bEndpointId;
                 enum
           {
        TT_CTL,
        TT_ISO,
        TT_BULK,
        TT_INT,
        TT_MASK = TT_INT,

        ST_NOSYNC= 0,
        ST_ASYNC = (1 << 2),
        ST_ADAPT = (2 << 2),
        ST_SYNC  = (3 << 2),
        ST_MASK  = ST_SYNC,

             UT_DATA           = 0,
             UT_FEED           = (1 << 4),
             UT_IMP_FEED_DATA  = (2 << 4),
             UT_RES            = (3 << 4),
             UT_MASK           = UT_RES
          } bmAttributes;
          u16 wMaxPacketSize;
          u8 bInterval;
  u8* p_classinfo;
} sOpUsbEpInfo_t, *psOpUsbEpInfo_t, **ppsOpUsbEpInfo_t;
```

Fields:

zlp_generation:

Indicate to the USB CL SP how to act with transfer's lengths (BULK / INTERRUPT) multiple of the endpoint's maximum packet size (only known by the USB CL SP).

bDirection:

The endpoint's direction: either OUT or IN.

bEndpointId:

The endpoint's logical identity (not related to the physical identity managed at the USB device controller level). Valid values are in the [1...15] range.

bmAttributes:

bInterval:

wMaxPacketSize:

For the meaning of those fields see the USB specification revision 2.0 – table9-13. Standard Interface Descriptor

Note: *wMaxPacketSize is used only for isochronous endpoints, for the control, bulk and interrupt endpoints the USB CL SP determines by itself (thanks to the USB device controller capabilities) the right values.*

p_classinfo:

Address of an array of bytes. Allows the service user in providing any endpoint class specified information. The field shall be set to NULL in case there is no endpoint class specified information.

4.2.2.8. The sOpUsbIaInfo_t Structure

The Open USB **Interface Association** information set, when the ADL open device service, adl_OpenDevice function, is used (nested in the sOpUsbItfInfo_t structure) by the USB CL SP to build the Endpoint standard descriptor.

```
typedef struct
{
   u8       bInterfaceCount;
   u8       bFunctionClass;
   u8       bFunctionSubClass;
   u8       bFunctionProtocol;
   ascii*   iInterface;
} sOpUsbIaInfo_t, *psOpUsbIaInfo;
```

Fields:

bInterfaceCount:

bFunctionClass:

bFunctionSubClass:

bFunctionProtocol:

iInterface:

For the meaning of this field see the USB specification revision 3.0 – Table 9-16 Standard Interface Association Descriptor.

Note: The bFirstIntrefaceCount field specified in the USB specification revision 3.0 –Table 9-16 is automatically generated by the USB CL SP.

4.2.2.9. The sOpUsbIocInterrupt_t Structure

The USB CL SP events setting structure, when the ADL open device service, io_control function with the cmd parameter sets to **IOC_USB_INT**.

```
typedef struct
{
   eGIocSo_t op;  /* generic get/set operation */
   enum
   {
      OUSB_INTR_MASKED,
      OUSB_INTR_DEMASKED,
      OUSB_INTR_MAP = 0x70000000L
   }intr_mask;
} sOpUsbIocInterrupt_t, *psOpUsbIocInterrupt_t;
```

Fields:

op:

This field describes the sub-operation to be executed either *set* or *get* the state of the USB device controller interrupt mask.

intr_mask:

The supported values for the interrupt mask.

4.2.2.10. The sOpUsbIocFlush_t Structure

The USB CL SP endpoint flushing structure, when the ADL open device service, io_control function with the cmd parameter sets to **IOC_OUSB_FLUSH**.

```
typedef struct
{
   u8 identity;
   u8 direction;
} sOpUsbIocFlush_t, *psOpUsbIocFlush_t;
```

Fields:

direction:

Direction of the endpoint that must be flushed, valid directions are in the [0:OUT, 1:IN] range.

identity:

Identity of the endpoint that must be flushed, valid identities are in the [0…15] range.

4.2.2.11. The sOpUsbObjectId_t Structure

The USB Objects Identity structure, when the ADL open device service, io_control function with the cmd parameter sets to **IOC_USB_OBJECT_ID**.

```
typedef struct
{
   enum
   {
      OUSB_INTERFACE_OBJ,
      OUSB_ENDPOINT_OBJ,
      OUSB_LAST_OBJ
   }object_selector;
   union //IN: local object ID (UFL) OUT: actual object ID (UCL)
   {
      u16 itfid;
      u8 epid[2];    //[0] identity [1..15] [1]:direction [IN, OUT]
   } __attribute__((packed,aligned(2))) u;
} sOpUsbObjectId_t, *psOpUsbObjectId_t;
```

Fields:

object_selector:

Indicate to the USB Core Layer the kind of USB object the actual identity has to be retrieved. Interface and endpoints are supported.

u:

According to the selected object (interface or endpoint) indicates to the USB Core Layer the local identity of the object to be processed.

Interface identifier (u.itfid field) shall be in the [0…255] range.

Endpoint identifiers (see comments).

4.2.2.12. The sOpUsbTransAttr_t Structure

The transfer attributes structure, when the read and write operations are invoked by the USB SU. Instead of providing a void* as second parameter for the read and write operations (see generic read a write operations prototype in the wm_device.h file) the USB SU is required to use a pointer on this structure.

```
typedef struct
{
   u8       identity;
   u8       spare[3];
   void*    data;
} sOpUsbTransAttr_t, *psOpUsbTransAttr_t;
```

Fields:

Identity:

Indicates in the [1...15] range the logical endpoint identity on which the transfer shall occur.

data:

According to the requested operation (read or write) indicates where to put the received data or to get the data to be transmitted.

4.2.2.13. The sOpUsbOnStatus_t Structure

The ON STATUS event parameter structure sets by the USB CL SP when it warms the OnStatus event.

```
typedef struct
{
   enum eOpUsbStatus
           {
        OUSB_STATUS_CONFIGURED,      // id.config
        OUSB_STATUS_DECONFIGURED,    // id.cause
        OUSB_STATUS_ITF_STARTED,     // id.itf
        OUSB_STATUS_LAST,
        OUSB_STATUS_MAP = 0x7FFF
          } status;
          union
          {
             u16 config;
             u8  itf[ 2 ];         // [0]:bNumInterface,[1]:bAlternateSetting
             enum
             {
           BUS_RESET,
                CABLE_UNPLUGGED,
           CONFIGURATION_CHANGE
             }cause;
  } __attribute__((packed,aligned(2))) id;
} sOpUsbOnStatus_t, *psOpUsbOnStatus_t;
```

Fields:

status:

This field describes the current status of the USB CL SP. Two levels are defined, one for the whole device and the other for the interfaces. The device can be in the:

- Configured state: after a Set Configuration with a wValue not set to 0
- De-configured state: after a bus reset, an USB cable unplugging or a bus reset.

When the host issues a Set Interface request the OUSB_STATUS_IT_STARTED event is warmed by the USB CL SP.

id:

This field contains the configurations and interfaces identity:

- For configurations identity select the id.config field.
- For interfaces identity select the id.itf field, id.itf[0] contains the bNumInterface value and id.itf[1] contains the bAlternateSetting value.

This field hosts too the de-configuration's cause: bus reset, cable unplugged and configuration change causes are supported.

Caution: *It pertains to a Service User having pending read / write operations to properly act on an unexpected status notification (STATUS_DECONFIGURED / STATUS CONFIGURED / ITF STARTED). Particular attention should be given to buffer management in case such notifications would occur.*

4.2.2.14. The sOpUsbOnComplete_t Structure

The ON COMPLETE event parameter structure, sets by the USB CL SP when it warms the OnComplete event.

```
typedef struct
{
   u8       identity;
   u8       direction;
   u8       spare[2];
   void*    data;
   u32      length;
} sOpUsbComplete_t, *psOpUsbComplete_t;
```

Fields:

data:

In case of a read completion the address of the next RX buffer, in case of a write completion the address of the next buffer to be transmitted. USB SU shall set this field to NULL in case there is no other transfer (read or write) to be done.

length:

USB SP CL shall set this field to indicate to USB SU actual amount of received data. USB SU shall set this field in the following way:

In case of a read completion the size of the next RX buffer, in case of a write completion the size of the next buffer to be transmitted. Buffer size is expressed in bytes. In case no other transfer is required this field shall be set to 0.

identity:

Is the logical endpoint identifier supported values are in the [0...15] range.

direction:

Indicates the completion direction: when set to 0 (OUT direction) RX completion, when set to 1 (IN direction) write completion.

4.2.2.15. The uOpUsbOnRequest_t Union

The ON REQUEST event parameter structure sets by the USB CL SP when it warms the OnRequest event.

```
typedef union
{
            struct   // Set by USB CL
          {
             enum eOpUsbReqType
             {
                RECIPIENT_DEVICE,
                RECIPIENT_INTERFACE,
                RECIPIENT_ENDPOINT,
                RECIPIENT_OTHER,
                RECIPIENT_RESERVED,
                RECIPIENT_MASK = 0x1F,

                TYPE_STANDARD  = 0x00,
                TYPE_CLASS     = 0x20,
                TYPE_VENDOR    = 0x40,
                TYPE_RESERVED  = 0x60,
                TYPE_MASK      = TYPE_RESERVED,

                DIRECTION_OUT  = 0,
                DIRECTION_IN   = 0x80,
                DIRECTION_MASK = DIRECTION_IN,
             }bmRequestType;
             u8 bRequest;
             union
             {
                u8  b[2];
                u16 w;
             } __attribute__((packed,aligned(2))) wValue;
             union
             {
                u8  b[2];
                u16 w;
             } __attribute__((packed,aligned(2))) wIndex;
             u16 wLength;
          } setup;
          struct  // Set by USB FL
          {
        void* data_stage;
        s16 status;  // 0: OK, -1: ERROR (see wm_types.h)
        u16 length;
          }out;
} uOpUsbOnRequest_t, *puOpUsbOnRequest_t;
```

Fields:

setup (structure):

To forward class, vendor specific and some standard requests to USB SU. Refer to the clause 9.3 of the USB specification revision 2.0 for further details about this field. This field is written by the USB CL SP and is read by the USB SU.

out (structure):

To provide information to USB CL SP in order to it to handle data or status stages. This field is written by the USB SU and is used by the USB CL SP.

out.data_stage: address of the transmission or reception buffer required for the data stage or a valid (byte for example) address to handle the status stage.

out.status: USB SU is required to set this field to -1 (will generate a stall in the control endpoint) in case an error occurred, otherwise the USB SU shall set it to 0 (data stage can occur or status stage ok) or to 1 for delayed IN status stages (IN status stage shall be triggered later on by the Service User and IN status stage's completion is automatically handled by the USB Core Layer)

out.length: actual length for the data stage. It shall be set to 0 for the status stage management.

Caution: *It pertains to a Service User waiting for a pending data stage completion to properly act on an unexpected request notification (see USB specification for further details).*

4.2.3. Enumerators

4.2.3.1. The eOpUsbIoCmd_t type

This enumeration lists the available IO commands identities for configuring, or obtaining information from, the USB CL SP. Enumeration values are used as second parameter for the io_control operation.

```
typedef enum
{
      IOC_OUSB_EH    = G_IOC_EVENT_HANDLERS,
      IOC_OUSB_CAP   = G_IOC_CAPABILITIES,
      IOC_OUSB_INT,
      IOC_OUSB_FLUSH,
      IOC_OUSB_OBJECT_ID,   // Retrieving actual INTERFACE and ENDPOINTS
                            // identity (GET_ONLY)
      IOC_OUSB_LAST,
      IOC_OUSB_MAP   = 0x70000000L
} eOpUsbIoCmd_t, *peOpUsbIoCmd_t;
```

Description:

IOC_OUSB_EH

To allow the application to retrieve the event handling configuration used at the USB CL SP side.

IOC_OUSB_CAP NOT IMPLEMENTED

IOC_OUSB_INT

Allows the application to set and get the state of the device controller interrupt mask.

IOC_OUSB_FLUSH

Allows the application to flush an endpoint.

IOC_OUSB_OBJECT_ID

Allows the application (USB Function Layer) to retrieve the actual INTERFACE & ENDPOINTS identities (those communicated to the host during the enumeration stage) in order to build some class requests.

4.2.3.2. The E_ALREADY_BOUND_t constant

This constant is returned by adl_OpenDevice function in case a USB Function Layer (located either at the ADL or the firmware side) is already bound with the USB Core Layer.

4.2.4. Operations

There are two types of operations defined by the Open USB Interface:

- Requests: Allow the service user to directly handle the USB CL SP.
- Notifications: Allow the USB CL SP to notify event occurrences to service user.

Note: *Before requesting, or being notified by, an USB CL SP an Open AT® application shall retrieve a, direct, access (by using the ADL Open Device service) to this USB CL SP.*

5 request functions are offered:

- An open function to:
  - Retrieve the USB CL service provider's interface (through a generic interface container) and
  - retrieve a unique USB CL SP reference (handle) which shall be subsequently provided as parameter to the rest of the request functions and
  - install event handlers to manage the USB CL SP notifications and
  - Optionally retrieve the USB CL service provider’s capabilities.
- A read function to retrieve data chunk sent by the USB host.
- A write function to instruct the USB CL SP to send data chunk to the USB host.
- An io_control function to configure, or get information from, the USB CL SP.
- A close function to release the USB CL SP interface (and the handle previously allocated).

3 notifications are offered to the USB SU to inform it of the occurrence of:

- The status changes (ON_STATUS) at the device and interface levels.
- The completion of the pending read/write operations (ON_COMPLETE).
- The arrival of a class or vendor specific request (ON_REQUEST).

Warning: *Calling request functions (read/write/io_control/close) while application event handlers are running is not supported. Such call shall not be managed and shall return an error.*

4.2.4.1. The open Function

There is no, at strictly speaking, specific function provided to open (get a direct access to) the USB CL SP. The ADL Open Device service provides a generic function allowing getting, direct, access to numerous kinds of service providers, including the USB CL SP one. Hereafter a description of what is needed to open USB CL SP.

Prototype

```
s32 adl_OpenDevice ( eDfClid_t     dev_clss_id,
                     void *        param );
```

Parameters

dev_clss_id:

The device class identifier the service provider to be opened belongs to. To open USB CL SP the application has to use the **DF_USB_CLID** value.

param:

Service provider configuration, to be defined accordingly to the dev_clss_id parameter in the USB CL SP case address of a sOpUsbSettings_t structure is required.

Returned values

- `Handle`: A positive USB CL SP handle on success, to be used in further Open USB service function calls.
- Otherwise the USB CL SP opening failed (check your input parameters)

Example: How to open the USB CL SP

```
#include "adl_OpenDevice.h"
#include "wm_usb.h"

static psGItfCont_t usb_if;
static u32 usb_hdl;

void adl_main( adl_InitType_e InitType )
{
   sOpUsbSettings_t settings;
   static sOpUsbIocInterrupt_t usb_it = { .op = G_IOC_OP_SET, .intr_mask=
OUSB_INTR_MASKED };

   // Set the settings parameters
   settings.identity = "USBDEV0";
   settings.interface = &usb_if;
   // fill-in the rest of the settings fields hereafter
   …

   usb_hdl = adl_OpenDevice( DF_USB_CLID, &settings );
   if( !usb_hdl )
   {
      // USB CL opening failed...
     return;
   }

   // USBDEV0 successfully opened, mask the device controller interrupt
   usb_if.io_control( usb_hdl, IOC_OUSB_INT, (void*)&usb_it);
   …
}
```

4.2.4.2. The read Request

This function allows the USB SU to read the data received by the USB CL SP. Before using this function the USB SU shall open the USB CL SP (hat to own the USB CL interface as well as a valid USB CL handle).

The read function works asynchronously. USB SU when calling the reads functions provides the USB CL SP with a reception buffer address, the buffer size and logical endpoint in which the read operation is applying. USB CL SP returns an operation pending indication. While an asynchronous read operation is pending USB SU is allowed to invoke the read function with both reception buffer address and size parameters set to a NULL value in order to cancel it. When a read operation is cancelled by the USB SU the read completion event handler is not called by the USB CL SP.

The read operation completion occurs when USB CL SP invokes the ON COMPLETE event manager.

Prototype

```
eChStatus_t read (   u32                   Handle,
                     psOpUsbTransAttr_t    trsf_attr,
                     u32                   len );
```

Parameters

Handle:

Handle of the USB CL previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

trsf_attr structure (by address):

trsf_attr.data This field contains the reception buffer's address. NULL value is supported only in case a read operation is pending

trsf_attr.identity the endpoint logical identity on which read operation applies [1...15].

len:

This field contains the size of the reception buffer. NULL value is supported to cancel a pending read operation.

Returned values

- **CH_STATUS_ERROR:** Invalid USB CL SP handle or a read operation is already pending.
- **CH_STATUS_NORMAL: OK**, read cancellation successfully completed.
- **CH_STATUS_PENDING**: **OK**, read operation is pending.

Note: *USB chip's interrupts are automatically disabled by the USB Core Layer during this request processing.*

Example: How to launch an asynchronous read operation

```
#include "adl_OpenDevice.h"
#include "wm_usb.h"

// To be coded
```

4.2.4.3. The write Request

This function allows the USB SU to send data block to the USB host. Before using this function the USB SU shall open the USB CL SP (has to own the USB CL SP interface as well as a valid USB CL SP handle).

The write operation works asynchronously. USB SU when calling the write operation is provisioning USB SP CL with the transmission buffer address, the buffer size and endpoint logical identifier in which the transfer applies. USB CL SP returns an operation pending indication. While an asynchronous write operation is pending USB SU is allowed to invoke the write function with both transmission buffer address and size parameters set to a NULL value in order to cancel it. When a pending write operation is cancelled by the USB SU the write completion event handler is not called by the USB CL SP.

Prototype

```
eChStatus_t write(   u32                  Handle,
                     psOpUsbTransAttr_t   trsf_attr,
                     u32                  len );
```

Parameters

Handle:

Handle of the USB CL previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

trsf_attr structure (by address):

trsf_attr.data This field contains the transmission buffer's address. NULL value is supported only in case a write operation is pending.

trsf_attr.identity the endpoint logical identity on which write operation applies [1...15].

len:

This field contains the size of the transmission buffer. NULL value is supported to cancel a pending write operation.

Returned values

- **`CH_STATUS_ERROR`**: Invalid USB CL SP handle or a write operation is already pending.
- **`CH_STATUS_NORMAL`**: **`OK`**, write cancellation successfully completed.
- **`CH_STATUS_PENDING`**: **`OK`**, write operation is pending.

Note: *USB chip's interrupts are automatically disabled by the USB Core Layer during this request processing.*

4.2.4.4. The io_control Request

This function allows to set or to get configuration information from the SB CL SP. Before using this function the application shall open the USB CL SP (has to own the USB CL SP interface as well as a valid USB CL SP handle).

This function is generic and supports several IO commands. To choose among the supported IO commands the application has to set the Cmd parameter with a supported IO command identifier.

Prototype

```
eChStatus_t io_control (   u32            Handle,
                           eOpUsbIoCmd_t  Cmd,
                           void*          pParam );
```

Parameters

Handle:

Handle of the USB CL previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

Cmd:

Open USB IO command identifier.

See also eOpUsbIoCmd_t for further information about the supported Open USB IO commands.

pParam:

IO command parameter. Type of this parameter depends on the Cmd parameter value. Following sub clauses will detail the actual type to be used.

Returned values

Depend on the IO command type, the following sub clauses will detail actual return values.

4.2.4.4.1. The IOC_OUSB_EH IO Command

This function allows getting (**read only**) the USB CL SP event handling configuration information.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_OUSB_EH.

pParam:

Address of a **`sOpUsbEvent_t`** structure.

Note: *The op field of the sOpUsbEvent_t structure shall be set to G_IOC_GET constant.*

Returned values

- **`CH_STATUS_ERROR`**: invalid USB CL SP handle / unknown operation / pParam set to NULL / invalid configuration.
- **`CH_STATUS_NORMAL`**: command succeeded.

4.2.4.4.2. The IOC_OUSB_CAP IO Command

This function allows getting the capabilities of the USB CL SP. **[Currently not supported]**

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_OUSB_CAP.

pParam:

Address of a **`sOpUsbCapabilities_t`** structure.

Note: *The op field of the sOpUsbCapabilities_t structure shall be set to G_IOC_GET constant*

Returned values

- **`CH_STATUS_ERROR`**: invalid CL SP handle // pParam set to **`NULL`**.
- **`CH_STATUS_NORMAL`**: command succeeded.

4.2.4.4.3. The IOC_OUSB_FLUSH IO Command

This function allows flushing any endpoints.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_OUSB_FLUSH.

pParam:

Address of a **`sOpUsbIocFlush_t`** structure.

Returned values

- **`CH_STATUS_ERROR`**: invalid USB CL SP handle / pParam set to NULL / invalid configuration.
- **`CH_STATUS_NORMAL`**: command succeeded.

4.2.4.4.4. The IOC_OUSB_INT IO Command

This function allows setting or getting the USB device controller interrupt mask.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_OUSB_INT.

pParam:

Address of a `sOpUsbIocInterrupt_t` structure.

Returned values

- `CH_STATUS_ERROR`: invalid USB CL SP handle / pParam set to NULL / invalid configuration.
- `CH_STATUS_NORMAL`: command succeeded.

4.2.4.4.5. The IOC_OUSB_OBJECT_ID IO Command

This function allows a function layer retrieving the actual USB identifier for USB objects such as Interfaces or Endpoints.

Prototype

See also the io_control request for further information about io_control prototype and parameter description.

Parameters

Handle:

See also the io_control request for further information about this parameter.

Cmd:

Set to IOC_OUSB_OBJECT_ID.

pParam:

Address of a `sOpUsbObjectId_t` structure.

Returned values

- `CH_STATUS_ERROR`: invalid USB CL SP handle / pParam set to NULL / invalid configuration.
- `CH_STATUS_NORMAL`: command succeeded.

4.2.4.5. The close Request

This function allows the USB SU to stop all pending, read and write operations and to release the USB CL SP. Before using this function the USB SU shall open the USB CL SP (has to own the USB CL SP interface as well as a valid USB CLSP handle).

Prototype

```
eChStatus_t close ( u32     Handle );
```

Parameters

Handle:

Handle of the USB CL previously returned by the adl_OpenDevice function. Setting this parameter with a value different from the one obtained by the call to the adl_OpenDevice function generates an error.

Returned values

- **`CH_STATUS_ERROR`**: invalid USB CL handle.
- **`CH_STATUS_NORMAL`**: close operation successfully completed.

4.2.4.6. The ON COMPLETE Notification Handler

This notification allows the application to be aware of the completion of the pending asynchronous read/write operation (including the control ones).

Before being notified the USB SU shall open the USB CL SP (must own the USB CL SP interface as well as a valid USB CL SP handle) and configure the USB CL service provider with its on_complete notification handler.

Prototype

```
void on_complete (   void*                  user_data,
                     psOpUsbOnComplete_t    evpar );
```

Parameters

user_data:

Information the USB SU (a context for example) provided during the event handler configuration stage. The USB CL SP is required to give back this information to the application on every occurrence of the ON COMPLETE event.

evpar:

Address of a **`sOpUsbOnComplete_t`** structure allowing the USB CL SP to inform the USB SU on the completion of read or write operations

USB SU to provide additional buffer to be read or written (according to the transfer direction).

Returned values

Not Applicable.

4.2.4.7. The ON REQUEST Notification Handler

This notification allows the application to be aware of the arrival of a USB request sent by the USB host.

Before being notified the USB SU shall open the USB CL SP (must own the USB CL SP interface as well as a valid USB CL SP handle) and configure the USB CL service provider with its on_request notification handler.

Prototype

```
void on_request ( void*                user_data,
                  puOpUsbOnRequest_t   evpar );
```

Parameters

user_data:

Information the USB SU (a context for example) provided during the event handler configuration stage. The USB CL SP is required to give back this information to the application on every occurrence of the ON REQUEST event.

evpar:

Address of a **`uOpUsbRequest_t`** union allowing the USB CL SP to provide the USB SU with the USB request issued by the USB host

USB SU to provide USB CL SP with additional information for handling the status or data stage.

Returned values

Not Applicable.

4.2.4.8. The ON STATUS Notification Handler

This notification allows the USB SU to be aware of the device or interface state changes. It occurs either when the device is configured (on a host Set Configuration request) or the device is de-configured (on cable unplugging, bus reset or on host Set Configuration (0) request).

Before being notified the USB SU shall open the USB CL SP (must own the USB CL SP interface as well as a valid USB CL SP handle) and configure the USB CL service provider with its on_status notification handler.

Prototype

```
void on_status (  void*               user_data,
                  sOpUsbOnStatus_t    evt_param);
```

Parameters

user_data:

Information the USB SU (a context for example) provided during the event handler configuration stage. The USB CL SP is required to give back this information to the application on every occurrence of the ON STATUS event.

evt_param:

`sOpUsbOnStatus_t` structure allowing USB CL SP to provide the USB SU with the kind of changes occurring and configuration or interface identities.

Returned values

Not Applicable

5. Error Codes

5.1. 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 embedded module, 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

5.2. 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

5.3. 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

5.4. 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

5.5. 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
ADL_AD_RET_ERR_REACHED_END	ADL_RET_ERR_SPECIFIC_BASE - 4
ADL_AD_RET_ERR_UPDATE_FAILURE	ADL_RET_ERR_SPECIFIC_BASE - 5
ADL_AD_RET_ERR_RECOVERY_DONE	ADL_RET_ERR_SPECIFIC_BASE - 6
ADL_AD_RET_ERR_OAT_DEACTIVATED	ADL_RET_ERR_SPECIFIC_BASE - 7
ADL_AD_SIZE_UNDEF	0XFFFFFFFF
ADL_AD_MAX_CELL_RETRIEVE	600

6. Resources

Here are listed the available resources of the Open AT® OS.

Resource name	Value
Maximum tasks count	64
Maximum running timers	40
Maximum running timers count per task	32
Semaphore resources	100

Index